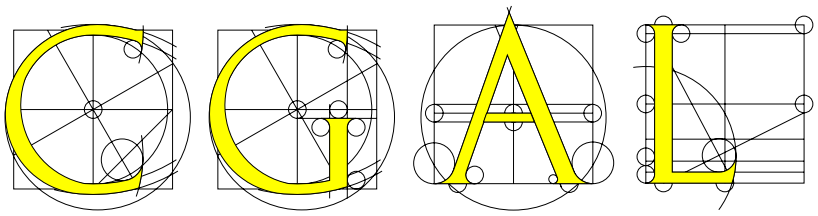


CGAL Reference Manual

Part 2: Basic Library

Release 1.0, April 1998

Preface

CGAL is a *Computational Geometry Algorithms Library* written in C++, which is developed by the ESPRIT project CGAL. This project is carried out by a consortium consisting of Utrecht University (The Netherlands), ETH Zürich (Switzerland), Freie Universiät Berlin (Germany), INRIA Sophia-Antipolis (France), Max-Planck Institut für Informatik, Saarbrücken (Germany), RISC Linz (Austria) and Tel-Aviv University (Israel). You find more information on the project on the CGAL home page at URL `http://www.cs.uu.nl/CGAL/`.

Should you have any questions, comments, remarks or criticism concerning CGAL, please send a message to `cgal@cs.uu.nl`.

Editors

Hervé Brönnimann, Andreas Fabri (INRIA Sophia-Antipolis)
Stefan Schirra (Max-Planck Institut für Informatik)
Remco Veltkamp (Utrecht University)

Authors

Chapter 2: Wieger Wesselink (Utrecht University).

Chapter 3: Lutz Kettner (ETH Zürich).

Chapter 4: Lutz Kettner (ETH Zürich).

Chapter 5: Eyal Flato, Dan Halperin, Iddo Hanniel, Sariel Har-Peled, Michal Ozery (Tel-Aviv University).

Chapter 6: Hervé Brönnimann, Olivier Devillers, Andreas Fabri, Monique Teillaud, Mariette Yvinec (INRIA Sophia-Antipolis).

Chapter 7: Wolfgang Freiseisen, Carl Van Geem (RISC Linz).

Chapter 8: Stefan Schirra (Max-Planck Institut für Informatik).

Chapter 9: Bernd Gärtner, Michael Hoffmann (ETH Zürich); Sven Schönherr (Freie Universiät Berlin).

Chapter 10: Gabriele Neyer (ETH Zürich); Eyal Flato, Dan Halperin, Sariel Har-Peled (Tel-Aviv University).

Acknowledgement

This work was supported by the ESPRIT IV Long Term Research Project No. 21957 (CGAL).

Contents

Chapter 1

Introduction

CGAL, the *Computational Geometry Algorithms Library*, is written in C++ and consists of three 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 parameterized by a representation class, which specifies the underlying number types used for calculations. 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. The third part 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 basic library. The collection of basic geometric algorithms and data structures currently includes polygons, half-edge data structure, planar maps, boolean operations, triangulations, convex hulls, optimisation algorithms, and multidimensional search trees.

1.1 Organisation of the manual

The following chapters successively treat polygons, half-edge data structure, planar maps, boolean operations, triangulations, convex hulls, optimisation algorithms, and multidimensional search trees.

Some functionality of a number of classes and algorithms is considered more advanced, for example because integrity of the underlying data structure is not guaranteed. In such cases the programmer is responsible for integrity. These functionalities are described in sections like this one, bounded by horizontal brackets.

Chapter 2

Polygon

2.1 2D Polygon (*CGAL_Polygon_2<Traits,Container>*)

Definition

The class *CGAL_Polygon_2* represents a simple polygon in the two-dimensional Euclidean plane $\mathbb{E}_2$. A polygon is called *simple* if there is no pair of nonconsecutive edges sharing a point (see [PS85]).

An object p of the data type *CGAL_Polygon_2* is defined by the sequence of its vertices. A simple polygon p is oriented, i.e., its boundary has clockwise or counterclockwise orientation. The side to the left of the boundary is called the positive side and the side to the right of the boundary is called the negative side. As any Jordan curve, the boundary of a polygon divides the plane into two open regions, a bounded one and an unbounded one.

An object p of *CGAL_Polygon_2* is a dynamic data structure, i.e. vertices can be added and removed. These operations may destroy the simplicity of the polygon, which is a precondition to most predicates of polygons.

The data type *CGAL_Polygon_2* is parameterized with two template parameters: a traits class *Traits* and a container class *Container*. The parameter *Traits* defines the types and predicates that are used in the polygon class and the polygon algorithms. For example *Traits::Point_2* denotes the type of the vertices of the polygon. A default polygon traits class *CGAL_Polygon_traits_2<R>* is provided (see Section 2.4), where R is a representation class. The parameter *Container* specifies the type of container that is used to store the sequence of vertices of the polygon, e.g. a list, a vector, a tree, etc. The type *Container* should fulfill the requirements of a sequence container given in [MS96]. The value type of the container should be the same as the point type of the traits class.

Note: Currently, a polygon declaration looks like *CGAL_Polygon_2<Traits, list<Traits::Point_2> >*. When nested templates become available this might be simplified to *CGAL_Polygon_2<Traits, list>*.

Assertions

The polygon code uses infix *POLYGON* in the assertions, e.g. defining the compiler flag *CGAL_POLYGON_NO_PRECONDITIONS* switches precondition checking off, cf. Section 2 of the Reference Manual Part 0, General Introduction.

Changes

The most important changes with respect to the previous release are:

- There is now a proper traits parameter. It is no longer possible to use the representation class R parameter for the template parameter *Traits*. Instead the default traits class *CGAL_Polygon_traits_2<R>* should be used, or when appropriate a user defined one.
- The method *append* is now called *push_back*. This was done to support back insert iterators.
- The *transform* method has been removed. It is replaced by a global *transform* function.
- Random access methods have been added. These methods are only available for random access containers.
- The window stream output operator has been added.
- A few bugs have been fixed (in the area computation and the simplicity test).

Types

CGAL_Polygon_2<Traits,Container>:: Traits

The traits type.

CGAL_Polygon_2<Traits,Container>:: Container

The container type.

typedef Traits::FT FT;

typedef Traits::Point_2

Point_2;

typedef Traits::Segment_2

Segment_2;

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 with 'const' in their name are non-mutable, the others are mutable. The iterator category is in all cases bidirectional, except for *Vertex_iterator* and *Vertex_const_iterator*, which have 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. This will be corrected when iterator traits become available. The consequence of using iterators / circulators with an incorrect iterator category is that when an STL algorithm is applied to such a range, the wrong (i.e. inefficient) version of an STL algorithm may be selected.

For vertices we define

CGAL_Polygon_2<Traits,Container>:: Vertex_iterator

CGAL_Polygon_2<Traits,Container>:: Vertex_const_iterator

CGAL_Polygon_2<Traits,Container>:: Vertex_circulator

CGAL_Polygon_2<Traits,Container>:: Vertex_const_circulator

Their value type is *Point_2*.

For edges we define

CGAL_Polygon_2<Traits,Container>:: Edge_const_circulator

CGAL_Polygon_2<Traits,Container>:: Edge_const_iterator

Their value type is *Segment_2*.

Creation

template <class InputIterator>
CGAL_Polygon_2<Traits,Container> p(InputIterator first, InputIterator last);

Introduces a polygon p with vertices from the sequence defined by the range *[first,last)*.
Precondition: The value type of points in the range *[first,last)* is *Point_2*.

Operations

The following operations allow to modify a polygon.

Vertex_iterator *p.insert(Vertex_iterator i, Point_2 q)*

Inserts the vertex q before i. The return value points to the inserted vertex.

void *p.insert(Vertex_iterator i, InputIterator first, InputIterator last)*

Inserts the vertices in the range *[first, last)* before i.
Precondition: The value type of points in the range *[first,last)* is *Point_2*. Note: this method is only available if the compiler supports member templates.

void *p.push_back(Point_2 q)*

Has the same semantics as *p.insert(p.vertices_end(), q)*.

void *p.erase(Vertex_iterator i)*

Erases the vertex pointed to by i.

void *p.erase(Vertex_iterator first, Vertex_iterator last)*

Erases the vertices in the range *[first, last)*.

void *p.reverse_orientation()*

Reverses the orientation of the polygon. The vertex pointed to by *p.vertices_begin()* remains the same.

Traversal of a polygon

The following methods of the class *CGAL_Polygon_2* return circulators and iterators that allow to traverse the vertices and edges.

Vertex_iterator *p.vertices_begin()*

Returns a mutable iterator that allows to traverse the vertices of the polygon p.

Vertex_iterator *p.vertices_end()*

Returns the corresponding past-the-end iterator.

Vertex_const_iterator

p.vertices_begin()

Returns a non-mutable iterator that allows to traverse the vertices of the polygon p.

Vertex_const_iterator

p.vertices_end()

Returns the corresponding past-the-end iterator.

Vertex_circulator *p.vertices_circulator()*

Returns a mutable circulator that allows to traverse the vertices of the polygon p.

Vertex_const_circulator

p.vertices_circulator()

Returns a non-mutable circulator that allows to traverse the vertices of the polygon p.

Edge_const_iterator *p.edges_begin()*

Returns a non-mutable iterator that allows to traverse the edges of the polygon p.

Edge_const_iterator *p.edges_end()*

Returns the corresponding past-the-end iterator.

Edge_const_circulator

p.edges_circulator()

Returns a non-mutable circulator that allows to traverse the edges of the polygon p.

Predicates

bool *p.is_simple()*

Returns whether p is a simple polygon.

bool *p.is_convex()*

Returns whether p is convex.

CGAL_Orientation *p.orientation()*

Returns the orientation of p. If the number of vertices $p.size() < 3$ then *CGAL_COLLINEAR* is returned.
Precondition: *p.is_simple()*.

CGAL_Oriented_side *p.oriented_side(Point_2 q)*

Returns *CGAL_POSITIVE_SIDE*, *CGAL_NEGATIVE_SIDE*, or *CGAL_ON_ORIENTED_BOUNDARY*, depending on where point q is.
Precondition: *p.is_simple()*.

CGAL_Bounded_side *p.bounded_side(Point_2 q)*

Returns the symbolic constant *CGAL_ON_BOUNDED_SIDE*, *CGAL_ON_BOUNDARY* or *CGAL_ON_UNBOUNDED_SIDE*, depending on where point q is.
Precondition: *p.is_simple()*.

CGAL_Bbox_2 *p.bbox()* Returns the smallest bounding box containing p.

Traits::FT *p.area()* Returns the signed area of the polygon p. This means that the area is positive for counter clockwise polygons and negative for clockwise polygons.

Vertex_iterator *p.left_vertex()*

Returns the leftmost vertex of the polygon p with the smallest y-coordinate.

Vertex_iterator *p.right_vertex()*

Returns the rightmost vertex of the polygon p with the largest y-coordinate.

Vertex_iterator *p.top_vertex()*

Returns topmost vertex of the polygon p with the largest x-coordinate.

Vertex_iterator *p.bottom_vertex()*

Returns the bottommost vertex of the polygon p with the smallest x-coordinate.

For convenience we provide the following boolean functions:

bool *p.is_counterclockwise_oriented()*

bool *p.is_clockwise_oriented()*

bool *p.is_collinear_oriented()*

bool *p.has_on_positive_side(Point_2 q)*

bool *p.has_on_negative_side(Point_2 q)*

bool *p.has_on_boundary(Point_2 q)*

bool *p.has_on_bounded_side(Point_2 q)*

bool *p.has_on_unbounded_side(Point_2 q)*

Random access methods

These methods are only available for random access containers.

Point_2 *p.vertex(int i)*

Returns a (const) reference to the i-th vertex.

Point_2 *p[int i]* Returns a (const) reference to the i-th vertex.

Segment_2 *p.edge(int i)*

Returns a const reference to the i-th edge.

Miscellaneous

int *p.size()*

Returns the number of vertices of the polygon p.

bool *p.is_empty()*

Returns $p.size() == 0$.

Container *p.container()*

Returns a const reference to the sequence of vertices of the polygon p.

Globally defined operators

template <class Traits, class Container1, class Container2>

bool *operator==(CGAL_Polygon_2<Traits,Container1> p1,*
CGAL_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 *operator!=(CGAL_Polygon_2<Traits,Container1> p1,*
CGAL_Polygon_2<Traits,Container2> p2)

Test for inequality.

template <class Transformation, class Traits, class Container>

CGAL_Polygon_2<Traits,Container>

transform(Transformation t, p)

Returns the image of the polygon p under the transformation t.

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.

ostream& *ostream& os << CGAL_Polygon_2<Traits, Container> p*

Inserts the polygon p into the stream os.
Precondition: The insert operator is defined for class *Point_2*.

istream& *istream& is >> CGAL_Polygon_2<Traits, Container> p*

Reads a polygon from stream is and assigns it to p.

#include <CGAL/IO/Window_stream.h>

CGAL_Window_stream&

CGAL_Window_stream& W << CGAL_Polygon_2<Traits, Container> p

Inserts the triangulation p into the window stream W. The insert operator must be defined for *Point_2*.

Example

The following code fragment creates a polygon and checks if it is convex.

```
#include <CGAL/basic.h>
#include <CGAL/Cartesian.h>
#include <CGAL/Polygon_2.h>
#include <list.h>

typedef CGAL_Cartesian<double> R;
typedef CGAL_Polygon_traits_2<R> Traits;
typedef Traits::Point_2 Point;
```

```
typedef list<Point> Container;
typedef CGAL_Polygon_2<Traits,Container> Polygon;

#include <iostream.h>

int main()
{
  Polygon p;

  p.push_back(Point(0,0));
  p.push_back(Point(1,0));
  p.push_back(Point(1,1));
  p.push_back(Point(0,1));

  cout << "The polygon is " << (p.is_convex() ?  "" :  "not ") << "convex." << endl;

  return 0;
}
```

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 described in section 2.2. There you can find information about which algorithms were used and what their complexity they have.

2.2 Algorithms on sequences of 2D points

A number of algorithms on sequences of 2D points is supplied as global functions. In all algorithms the points in the range *[first,last)* are interpreted as the vertices of a polygon. The vertices in this range should have value type *Traits::Point_2*. Most functions take a traits class as their last argument. This is just a technical matter: the template argument *Traits* must appear in the arguments of the function. Modern compilers should be able to optimize this traits argument away. In all cases a default version of the function exists without the *Traits* argument. These default versions use the predefined polygon traits class from section 2.4.

template <*class ForwardIterator, class Traits*>
ForwardIterator *CGAL_left_vertex_2(* *ForwardIterator first,*
ForwardIterator last,
Traits traits)

Returns the leftmost point from the range *[first,last)* with the smallest y-coordinate. TRAITS: uses *Traits::Less_xy*.

template <*class ForwardIterator, class Traits*>
ForwardIterator *CGAL_right_vertex_2(* *ForwardIterator first,*
ForwardIterator last,
Traits traits)

Returns the rightmost point in the range *[first,last)* with the largest y-coordinate.
TRAITS: uses *Traits::Less_xy*.

template <class ForwardIterator, class Traits>
ForwardIterator *CGAL_top_vertex_2(ForwardIterator first,*
ForwardIterator last,
Traits traits)

Returns the topmost point from the range *[first,last)* with the largest x-coordinate.
TRAITS: uses *Traits::Less_yx*.

template <class ForwardIterator, class Traits>
ForwardIterator *CGAL_bottom_vertex_2(ForwardIterator first,*
ForwardIterator last,
Traits traits)

Returns the bottommost point from the range *[first,last)* with the smallest x-coordinate.
TRAITS: uses *Traits::Less_yx*.

template <class InputIterator>
CGAL_Bbox_2 *CGAL_bbox_2(InputIterator first, InputIterator last)*

Returns the smallest bounding box of the points in the range *[first,last)*.
Precondition: The range *[first,last)* is not empty.

template <class ForwardIterator, class Numbertype, class Traits>
void *CGAL_area_2(ForwardIterator first,*
ForwardIterator last,
Numbertype& result,
Traits traits)

Computes the signed area of the polygon formed by the points in the range *[first,last)*.
TRAITS: uses *Traits::determinant_2*.

template <class ForwardIterator, class Traits>
bool *CGAL_is_convex_2(ForwardIterator first,*
ForwardIterator last,
Traits traits)

Returns *true* if the polygon formed by the points in the range *[first,last)* is convex. The implementation checks if the polygon is locally convex and if there is only one local minimum and one local maximum with respect to the lexicographical ordering determined by *Traits::lexicographically_xy_smaller*. The complexity of the algorithm is $O(n)$, where n is the number of points in the range *[first,last)*.
TRAITS:
uses *Traits::lexicographically_xy_smaller* and *Traits::orientation*.

template <class ForwardIterator, class Traits>
bool CGAL_is_simple_2(ForwardIterator first,
ForwardIterator last,
Traits traits)

Returns *true* if the polygon formed by the points in the range *[first,last)* is simple. The simplicity test of a polygon is a line-segment intersection test. For the implementation a plane sweep algorithm is used, see also [PS85], p.276. The complexity of the algorithm is $O(n \log n)$ (worst case), where n is the number of points in the range *[first,last)*.
TRAITS:
uses *Traits::compare_x*, *Traits::compare_y*, *Traits::cross_product_2*, *Traits::is_negative*, *Traits::lexicographically_yx_smaller_or_equal*, *Traits::do_intersect*, *Traits::have_equal_direction*.

template <class ForwardIterator, class Point, class Traits>
CGAL_Oriented_side CGAL_oriented_side_2(ForwardIterator first,
ForwardIterator last,
Traits::Point_2 point,
Traits traits)

This determines the location of the point q with respect to the polygon formed by the points in the range *[first,last)*. Returns *CGAL_NEGATIVE_SIDE*, *CGAL_POSITIVE_SIDE*, or *CGAL_ON_ORIENTED_BOUNDARY*, depending on where point q is. For the implementation a *crossing test* algorithm is used with complexity $O(n)$, see [Hai94].
Precondition: The points in the range *[first,last)* form the vertices of a simple polygon.
TRAITS: uses *Traits::Less_xy*, *Traits::compare_x*, *Traits::compare_y*, *Traits::determinant_2*, *Traits::orientation* and *Traits::sign*.

template <class ForwardIterator, class Point, class Traits>
CGAL_Bounded_side CGAL_bounded_side_2(ForwardIterator first,
ForwardIterator last,
Traits::Point_2 point,
Traits traits)

Determines the location of the point q with respect to the polygon formed by the points in the range *[first,last)*. Returns *CGAL_ON_BOUNDED_SIDE*, *CGAL_ON_BOUNDARY* or *CGAL_ON_UNBOUNDED_SIDE*, depending on where point q is.
Precondition: The points in the range *[first,last)* form the vertices of a simple polygon.
TRAITS:
uses *Traits::compare_x*, *Traits::compare_y*, *Traits::determinant_2* and *Traits::determinant_2*.

template <class ForwardIterator, class Traits>
CGAL_Orientation CGAL_orientation_2(ForwardIterator first,
ForwardIterator last,
Traits traits)

Returns the orientation of the polygon formed by the points in the range *[first,last)*. If the number of points is smaller than three, *CGAL_COLLINEAR* is returned.
Precondition: The points in the range *[first,last)* form the vertices of a simple polygon.
TRAITS: uses *Traits::Less_xy* and *Traits::orientation*.

2.3 Polygon Traits class requirements

The polygon algorithms in section 2.2 are parameterized with a traits class *Traits* that defines the basic types and predicates that the algorithms use. This section describes the minimal requirements for the traits class. N.B. Currently the list of requirements contains several redundant predicates that can be easily expressed in others. For example, the lexicographical comparison functions can be expressed in the functions *compare_x* and *compare_y*). These predicates will probably be removed from the traits class.

Types

Traits:: FT The coordinate type of the points of the polygon (i.e. the *field type*).

Traits:: Point_2 The point type on which the polygon algorithms operate.

Traits:: Segment_2

The segment type on which the polygon algorithms operate.

Traits:: Vector_2

The vector type on which the polygon algorithms operate.

Traits:: Less_xy 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 resp.

Traits:: Less_yx Same as *Less_xy* with the roles of x and y interchanged.

Creation

Traits traits; Default constructor.

Traits traits(t);

Copy constructor.

Operations

bool *traits.lexicographically_xy_smaller(Point_2 p, Point_2 q)*

Returns *true* iff $p_x < q_x$ or $p_x = q_x$ and $p_y < q_y$.

bool *traits.lexicographically_yx_smaller_or_equal(Point_2 p, Point_2 q)*

Returns *true* iff $p_y \leq q_y$ or $p_y = q_y$ and $p_x \leq q_x$.

FT *traits.cross_product_2(Vector_2 p, Vector_2 q)*

Returns $p_x q_y - p_y q_x$.

FT *traits.determinant_2(Point_2 p, Point_2 q, Point_2 r)*

Returns $\begin{vmatrix} p_x & q_x & r_x \\ p_y & q_y & r_y \\ 1 & 1 & 1 \end{vmatrix} = (p_x - q_x)(q_y - r_y) - (p_y - q_y)(q_x - r_x)$.

int *traits.sign(FT x)*

Returns $\begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ 1 & \text{if } x > 0 \end{cases}$

bool *traits.is_negative(FT x)*

Returns *true* iff $x<0$.

bool *traits.do_intersect(Point_2 p1, Point_2 q1, Point_2 p2, Point_2 q2)*

Returns *true* iff the segments *[p1,q1]* and *[p2,q2]* intersect.

CGAL_Orientation *traits.orientation(Point_2 p, Point_2 q, Point_2 r)*

Returns *CGAL_LEFTTURN*, if r lies to the left of the oriented line l defined by p and q, returns *CGAL_RIGHTTURN* if r lies to the right of l, and returns *CGAL_COLLINEAR* if r lies on l.

CGAL_Comparison_result

traits.compare_x(Point_2 p, Point_2 q)

$$\text{Returns} \begin{cases} CGAL\_SMALLER & \text{if } p_x < q_x \\ CGAL\_EQUAL & \text{if } p_x = q_x \\ CGAL\_LARGER & \text{if } p_x > q_x \end{cases}$$

CGAL_Comparison_result

traits.compare_y(Point_2 p, Point_2 q)

$$\text{Returns} \begin{cases} CGAL\_SMALLER & \text{if } p_y < q_y \\ CGAL\_EQUAL & \text{if } p_y = q_y \\ CGAL\_LARGER & \text{if } p_y > q_y \end{cases}$$

bool *traits.have_equal_direction(Vector_2 v1, Vector_2 v2)*

Returns *true* iff the vectors *v1* and *v2* have the same direction.

2.4 Polygon default Traits class (*CGAL_Polygon_traits_2<R>*)

The polygon class *CGAL_Polygon_2* and the polygon algorithms are parameterized with a traits class *Traits*. The default polygon traits class *CGAL_Polygon_traits_2<R>* is parameterized with a representation class *R*.

#include <CGAL/Polygon_traits_2.h>

Types

typedef R::FT *FT;*

typedef CGAL_Point_2<_R>

Point_2;

typedef CGAL_Segment_2<_R>

Segment_2;

typedef CGAL_Vector_2<_R>

Vector_2;

typedef CGAL_p_Less_xy<Point_2>

Less_xy;

typedef CGAL_p_Less_yx<Point_2>

Less_yx;

Creation

CGAL_Polygon_traits_2<R> traits;

Operations

bool *traits.lexicographically_xy_smaller(Point_2 p, Point_2 q)*

Returns *CGAL_lexicographically_xy_smaller(p,q).*

bool *traits.lexicographically_yx_smaller_or_equal(Point_2 p, Point_2 q)*

Returns *CGAL_lexicographically_yx_smaller_or_equal(p,q).*

FT *traits.cross_product_2(Vector_2 p, Vector_2 q)*

Returns *p.x() * q.y() - q.x() * p.y().*

FT *traits.determinant_2(Point_2 p, Point_2 q, Point_2 r)*

Returns *cross_product_2(p-q, p-r).*

int *traits.sign(FT x)*

Returns *CGAL_sign(x).*

bool *traits.is_negative(FT x)*

Returns *CGAL_is_negative(x).*

bool *traits.do_intersect(Point_2 p1, Point_2 q1, Point_2 p2, Point_2 q2)*

Returns *CGAL_do_intersect(Segment_2(p1,q1), Segment_2(p2,q2).*

CGAL_Orientation *traits.orientation(Point_2 p, Point_2 q, Point_2 r)*

Returns *CGAL_orientation(p, q, r).*

CGAL_Comparison_result

traits.compare_x(Point_2 p, Point_2 q)

Returns *CGAL_compare_x(p, q).*

CGAL_Comparison_result

traits.compare_y(Point_2 p, Point_2 q)

Returns *CGAL_compare_y(p, q)*.

bool *traits.have_equal_direction(Vector_2 v1, Vector_2 v2)*

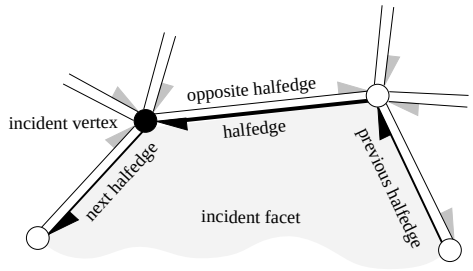
Returns the value of *CGAL_Direction_2<R>(v1) == CGAL_Direction_2<R>(v2)*.

Chapter 3

Halfedge Data Structures

3.1 Introduction

A halfedge data structure is an edge-centered data structure capable of maintaining incidence informations of vertices, edges and facets, for example for planar maps or polyhedrons. Each edge is decomposed into two halfedges with opposite orientations. One incident facet and one incident vertex are stored in each halfedge. For each facet and each vertex one incident halfedge is stored. Reduced variants of the halfedge data structure can omit some of these informations, for example the halfedge pointers in vertices or the storage of vertices at all.



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 different data structure. We continue naming it halfedge data structure (HDS). This data structure can also be seen as one of the variants of the quad-edge data structure [GS85], reduced to support only orientable surfaces. In general, the quad-edge data structure has a richer modeling space and can represent non-orientable 2-manifolds. The dual and the primal graph are symmetrically represented; the same algorithms can be applied to the dual graph as well as to the primal graph. This implies a lack of static type checking to distinguish between facets and vertices. Even though halfedges are similar to the winged edge data structure [Bau75] or the original reference for the DCEL [MP78], they are not equivalent, since the traversal algorithms are forced to search for certain incidences where the information is directly available in the halfedge data structure. An overview and comparison of these different data structures together with a thorough description of the design implemented here can be found in [Ket98].

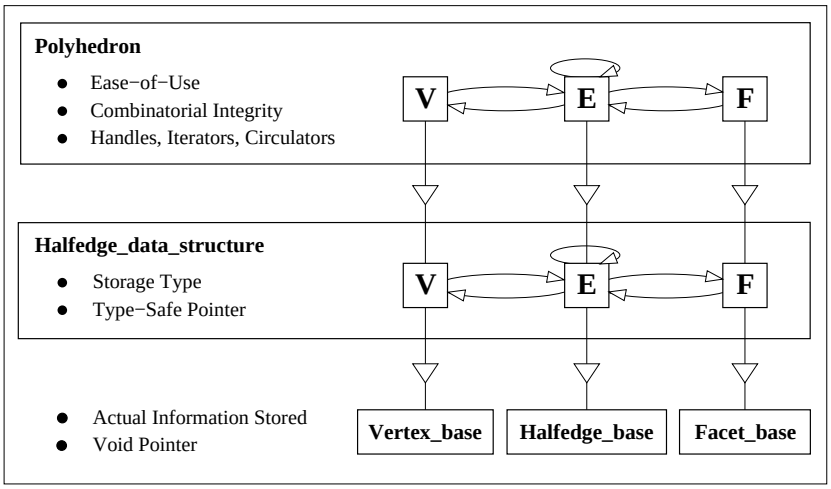


Figure 3.1: Responsibilities of the different layers in the halfedge data-structure design.

Design Overview

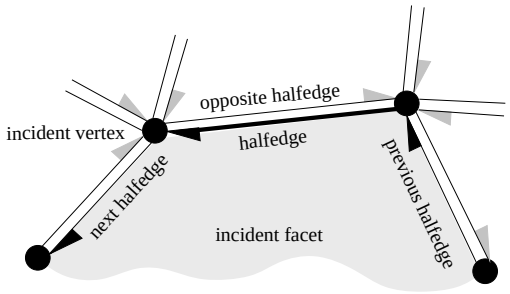
The design strictly separates topology and geometry. Vertices, halfedges and facets carry both kinds of information. The *Halfedge_data_structure* acts as a container class and stores all items and manages their incidence relations. For example the *CGAL_Polyhedron* from Chapter 4 uses the *Halfedge_data_structure* and adds geometric operations. It imposes further restrictions on the data structure, for example, that an edge always has two distinct endpoints.

Figure 3.1 offers a closer look at the design using *CGAL_Polyhedron* as example. Note that *Halfedge_data_structure*, *Vertex_base*, *Halfedge_base* and *Facet_base* are concepts, where each can have multiple models. There are many different possibilities for vertices, edges and facets. Currently, two different models are provided for the *Halfedge_data_structure*. Many combinations are possible and result in a different *CGAL_Polyhedron*. We make use of the implicit instantiation of template classes in C++. The requirement set for a model consists of a mandatory part that every model must comply with and certain optional parts a model must only comply with if the corresponding functionality is actually used. For example, a vertex is allowed to be empty. If we want to use it for the polyhedral surface, then the normal vector computation imposes the additional requirements that the vertex must contain a three-dimensional point and gives access to it with the member function *point()*.

The responsibilities of the base classes for vertices, halfedges and facets are the actual storage of the incidences in terms of *void*-pointers, the geometry and other attributes. Especially the storage of incidences with *void*-pointers allows, for example, the facet class to be exchanged without rewriting the halfedge class. The advantage of strong type checking will be reestablished in the next layer. Implementations for vertices, edges and facets are provided that fulfill the minimal set of requirements. They can be used as base classes for own extensions. Richer implementations are provided as defaults; for polyhedrons they provide a three-dimensional point in the vertices and a plane equation in the facets.

The *Halfedge_data_structure* is responsible for the storage organization of the vertices, halfedges and facets. Currently, implementations are provided that use a bidirectional list or an STL vector internally. The *Halfedge_data_structure* derives new classes for vertices, halfedges and facets. They replace the *void*-pointer incidence information with type-safe pointers at the interface. Additional information besides the incidence information simply stays unaffected and will be inherited.

Different models are possible for the *Halfedge_data_structure* (two are available right now). Thus the



Vertex
Halfedge* halfedge()
Point& point()

Halfedge
Halfegde* opposite()
Halfedge* next()
Halfedge* prev()
Vertex* vertex()
Facet* facet()

Facet
Halfedge* halfedge()

Figure 3.2: The three classes *Vertex*, *Halfedge*, and *Facet* of the halfedge data structure. Member functions with shaded background are mandatory. The others are optionally supported.

set of requirements for the *Halfedge_data_structure* is kept small. To support the implementation of high-level operations, a helper class *Halfedge_data_structure_decorator* is provided, which is not shown in Figure 3.1 but would be placed at the side of the *Halfedge_data_structure*, since it broadens that interface but does not hide it. It adds Euler operations and adaptive functionality. For example, if the *prev()* function is not provided for halfedges, a *find_prev()* function searches the previous halfedge along the facet. If the *prev()* function is available, the *find_prev()* 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 *CGAL_Polyhedron* (Chapter 4) adds an ease-of-use layer in terms of high-level functions, high-level concepts for accessing the items, i.e. handles, iterators and circulators (pointers are no longer visible at this interface), and the protection of the combinatorial integrity. It derives new vertices, halfedges and facets to provide the handles and hide the pointers.

Organization of this Chapter

Section 3.2 describes the requirements an *Halfedge_data_structure* with its three local classes for vertices, halfedges and facets must fulfill. Section 3.3 names the models currently available. These models make use of the *Vertex_base*, *Halfedge_base* and *Facet_base* concepts, whose requirements are stated in Section 3.4. The predefined base classes are described in Section 3.5. The helper class *Halfedge_data_structure_decorator* follows in Section 3.6. Several examples of halfedge data structures in Section 3.7 conclude the chapter.

3.2 Requirements for a *Halfedge_data_structure*

Definition

A *Halfedge_data_structure* 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 and facets are optional. See Figure 3.2 for the incidences stored and the mandatory or optional member functions possible for vertices, halfedges and facets.

A model for the *Halfedge_data_structure* concept must provide the following types and operations. In addition, the local types *Vertex*, *Halfedge* and *Facet* must fulfill the requirements listed in the Sections 3.2.1 to 3.2.3.

Types

Halfedge_data_structure:: Vertex	vertex type.
Halfedge_data_structure:: Halfedge	halfedge type.
Halfedge_data_structure:: Facet	facet type.
Halfedge_data_structure:: Size	type for size values.
Halfedge_data_structure:: Difference	type for difference values.
Halfedge_data_structure:: Point	type for the (optionally) associated geometry for vertices. If no point geometry is supported, the type is *void**.

For the following iterators exist appropriate const iterators too.

Halfedge_data_structure:: Vertex_iterator	iterator over all vertices.
Halfedge_data_structure:: Halfedge_iterator	iterator over all halfedges.
Halfedge_data_structure:: Facet_iterator	iterator over all facets.
Halfedge_data_structure:: iterator_category	category for all iterators.

Creation

Halfedge_data_structure hds;	the empty data structure *hds*.
Halfedge_data_structure hds(Size v, Size h, Size f);	a data structure *hds* with storage reserved for v vertices, h halfedges, and f facets. The reservation sizes are a hint for optimizing storage allocation.
Halfedge_data_structure hds(hds2);	copy constructor performing deep copy.
Halfedge_data_structure& hds = hds2	assignment performing deep copy.

void *hds.reserve(Size v, Size h, Size 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 be invalidated.

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_facets() const*	number of facets.
Size	*hds.capacity_of_vertices() const*	space reserved for vertices.
Size	*hds.capacity_of_halfedges() const*	space reserved for halfedges.
Size	*hds.capacity_of_facets() const*	space reserved for facets.
size_t	*hds.bytes() const*	bytes used for the polyhedron.
size_t	*hds.bytes_reserved() const*	bytes reserved for the polyhedron.

The following member functions return the non-mutable iterator or non-mutable circulator respectively if the halfedge data structure will be 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()*	
Facet_iterator	*hds.facets_begin()*	iterator over all facets (excluding holes).
Facet_iterator	*hds.facets_end()*	

Insertion

The following operations allocate a new element of the named item. Halfedges are always allocated in pairs of opposite halfedges. The opposite pointers are automatically set. Note that *new_vertex()* and *new_facet()* might not be present for halfedge data structures that do not support vertices or facets respectively.

*Vertex**	*hds.new_vertex()*	creates a default vertex.
*Vertex**	*hds.new_vertex(const Vertex* v)*	creates a copy of v.
*Vertex**	*hds.new_vertex(const Point& p)*	creates a new vertex initialized to p.
*Halfedge**	*hds.new_edge()*	creates a new pair of opposite halfedges.
*Halfedge**	*hds.new_edge(const Halfedge* h)*	creates a copy of the two halfedges h and *h->opposite()*.
*Facet**	*hds.new_facet()*	creates a default facet.
*Facet**	*hds.new_facet(const Facet* f)*	creates a copy of f.

Removal

The following operations erase an element referenced by a pointer. Halfedges are always deallocated in pairs of opposite halfedges, however, the halfedge pointer might denote any of the two possible halfedges. Erasing single elements is optional and indicated with the type tag *Supports_removal*. The *pop_back* member functions and the deletion of all items are mandatory, if the item is supported.

void	*hds.delete_vertex(Vertex* v)*	deletes the vertex v.
void	*hds.delete_edge(Halfedge* h)*	deletes the pair of opposite halfedges h.
void	*hds.delete_facet(Facet* f)*	deletes the facet f.
void	*hds.vertex_pop_back()*	removes last vertex. Mandatory.
void	*hds.edge_pop_back()*	removes last pair of halfedges. Mandatory.
void	*hds.facet_pop_back()*	removes last facet. Mandatory.
void	*hds.delete_all()*	deletes all elements. Mandatory.

Operations with Border Halfedges

The following notion of border halfedges is particular useful where the halfedge data structure is used to model surfaces that might contain holes. Halfedges incident to an hole are called *border halfedges*. A halfedge is a *border edge* if the halfedge 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 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. Halfedge insertion or removal and changing the border status of a halfedge may invalidate these values, which are not automatically updated.

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 facet, the halfedge iterator will reference the halfedge incident to the facet right before the halfedge incident to the hole.

Size *hds.size_of_border_halfedges() const*

number of border halfedges. An edge with no incident facet 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 facet on one side and to a hole 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.

Types for Tagging Optional Features

The nested types below are either equal to *CGAL_Tag_true* or *CGAL_Tag_false*, depending on whether the named member function is supported or not. Those related to vertices, halfedges and facets are identical to the definitions for the local vertex, halfedge or facet class respectively.

Halfedge_data_structure:: Supports_vertex_halfedge	*Vertex::halfedge()*.
Halfedge_data_structure:: Supports_vertex_point	*Vertex::point()*.
Halfedge_data_structure:: Supports_halfedge_prev	*Halfedge::prev()*.
Halfedge_data_structure:: Supports_halfedge_vertex	*Halfedge::vertex()*.
Halfedge_data_structure:: Supports_halfedge_facet	*Halfedge::facet()*.
Halfedge_data_structure:: Supports_facet_halfedge	*Facet::halfedge()*.

Halfedge_data_structure:: Supports_removal	the halfedge data structure supports the removal of individual elements of a surface, i.e. *delete_edge()*, *delete_vertex()* if vertices are supported, and *delete_facet()* if facets are supported.

The following dependencies among these options must be regarded:

Vertices are supported $\Longleftrightarrow$ *Supports_halfedge_vertex* $\equiv$ *CGAL_Tag_true*.
Facets are supported $\Longleftrightarrow$ *Supports_halfedge_facet* $\equiv$ *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_facet_halfedge $\equiv$ *CGAL_Tag_true* $\Longrightarrow$ *Supports_halfedge_facet* $\equiv$ *CGAL_Tag_true*.

See Also

CGAL_Halfedge_data_structure_decorator in Section 3.6. *CGAL_Polyhedron_3* in Chapter 4.

3.2.1 Requirements for a *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 3.2 depicts the relationship between a halfedge and its incident halfedges, vertices, and facets.

Types

Vertex:: Vertex	self.
Vertex:: Halfedge	corresponding halfedge type.
Vertex:: Facet	corresponding facet type.

Type for (optionally) associated geometry. If a point is not supported the type is *void**.

Vertex:: Point	point type stored in vertices.

Operations

*Halfedge**	*v.halfedge()*	an incident halfedge pointing to v.
*const Halfedge**	*v.halfedge()*	

Point&	*v.point()*	the point.
Point	*v.point()*	
void	*v.set_halfedge(Halfedge* h)*	set incident halfedge.

Types for Tagging Optional Features

The nested types below are either equal to *CGAL_Tag_true* or *CGAL_Tag_false*, depending on whether the named member function is supported or not.

Vertex:: Supports_vertex_halfedge	*halfedge()*.
Vertex:: Supports_vertex_point	*point()*.

3.2.2 Requirements for a *Halfedge*

Definition

A halfedge must store pointers to the next halfedge and its opposite halfedge. It optionally stores pointer to its incident vertex and incident facet. Type tags indicate whether these member functions are supported. Figure 3.2 depicts the relationship between a halfedge and its incident halfedges, vertices, and facets. The *is_border()* predicate is mandatory, but may return always *false*. A halfedge is an oriented edge between two vertices. It is always paired with its counterpart that has the opposite direction. They are mutually linked with the *opposite()* member function.

Types

Halfedge:: Vertex	corresponding vertex type.
Halfedge:: Halfedge	self.
Halfedge:: Facet	corresponding facet type.

Operations

*Halfedge**	*h.opposite()*	the opposite halfedge.
*const Halfedge**	*h.opposite()*	
*Halfedge**	*h.next()*	the next halfedge around the facet.
*const Halfedge**	*h.next()*	
*Halfedge**	*h.prev()*	the previous halfedge around the facet.
*const Halfedge**	*h.prev()*	
*Vertex**	*h.vertex()*	the incident vertex.
*const Vertex**	*h.vertex()*	
*Facet**	*h.facet()*	the incident facet. If h is a border halfedge the result might be *NULL* or a unique facet representing this hole or a unique facet representing all holes.
*const Facet**	*h.facet()*	
bool	*h.is_border()*	is true if h is a border halfedge.

void	*h.set_next(* next)*	
void	*h.set_prev(* prev)*	
void	*h.set_vertex(Vertex* v)*	
void	*h.set_facet(Facet* f)*	if $f == NULL$ h becomes a border edge.

Types for Tagging Optional Features

The nested types below are either equal to *CGAL_Tag_true* or *CGAL_Tag_false*, depending on whether the named member function is supported or not.

Halfedge:: Supports_halfedge_prev	*prev()*.
Halfedge:: Supports_halfedge_vertex	*vertex()*.
Halfedge:: Supports_halfedge_facet	*facet()*.

3.2.3 Requirements for a *Facet*

Definition

A facet optionally stores a reference to an incident halfedge that points to the facet. Type tags indicate whether this member function is supported. Figure 3.2 depicts the relationship between a halfedge and its incident halfedges, vertices, and facets.

Types

Facet:: Vertex	corresponding vertex type.
Facet:: Halfedge	corresponding halfedge type.
Facet:: Facet	self.

Creation

Facet f;	default constructor.
Facet f(Facet);	copy constructor.

Operations

*Halfedge**	*f.halfedge()*	an incident halfedge pointing to f.
*const Halfedge**	*f.halfedge()*	
void	*f.set_halfedge(Halfedge* h)*	

Types for Tagging Optional Features

The nested type below is either equal to *CGAL_Tag_true* or *CGAL_Tag_false*, depending on whether the named member function is supported or not.

Facet:: Supports_facet_halfedge	*halfedge()*.

3.3 Models of *Halfedge_data_structure*

Four models are currently provided for the *Halfedge_data_structure* concept in CGAL. Two generic models are parameterized with models of the *Vertex_base*, *Halfedge_base* and *Facet_base* concepts: *CGAL_Halfedge_data_structure_using_list* manages the items with a doubly-linked list, and *CGAL_Halfedge_data_structure_using_vector* manages the items internally with an STL vector. Two further models provide default solutions based on the generic model using lists. The model *CGAL_Halfedge_data_structure_default* implements all incidences and allows to choose a point type for vertices. *CGAL_Halfedge_data_structure_polyhedron_default_3* is prepared for polyhedral surfaces. It implements again all incidences and stores a three-dimensional point in vertices and a plane equation in facets, see Section 4.5.2 for details.

3.3.1 The Default Halfedge Data Structure

Definition

The default *CGAL_Halfedge_data_structure_default*<*Point*> implements all incidences supported by the *Halfedge_data_structure* concept. It stores a point of type *Point* in each vertex.

#*include* <*CGAL/Halfedge_data_structure_default.h*>

Implementation

The default halfedge data structure is a wrapper for *CGAL_Halfedge_data_structure_using_list* instantiated with *CGAL_Vertex_max_base*<*Point*>, *CGAL_Halfedge_max_base* and *CGAL_Facet_max_base*, see Section 3.3.2 to 3.5. Note that the copy constructor and the assignment operator are suboptimal for the list representation.

3.3.2 A Halfedge Data Structure Using Lists

Definition

CGAL_Halfedge_data_structure_using_list<*V,H,F*> is a model of the *Halfedge_data_structure* concept. It is parameterized with a model V of the *Vertex_base* concept, a model H of the *Halfedge_base* concept, and a model F of the *Facet_base* concept, whose requirements can be found in Section 3.4. Predefined models can be found in Section 3.5.

CGAL_Halfedge_data_structure_using_list<*V,H,F*> uses doubly-linked lists to store the items. This implies bidirectional iterators to access the elements and support for random insertions and deletions. It adds internally two additional pointers to each element. The capacity is not influenced by calls to *reserve()* and calling *reserve()* do not invalidate any iterator or circulator.

#*include* <*CGAL/Halfedge_data_structure_using_list.h*>

Types

It supports all types required for the *Halfedge_data_structure* concept, specifically the following two types are fixed.

typedef CGAL_Tag_true *Supports_removal;*
typedef bidirectional_iterator_tag *iterator_category;*

Operations

CGAL_Halfedge_data_structure_using_list<V,H,F> complies to the requirements stated for the concept *Halfedge_data_structure* in Section 3.2.

Implementation

It uses *CGAL_In_place_list* to implement the internal list storage, see the CGAL support library manual. The copy constructor and the assignment operator must update all internal pointers, which is currently done using the STL map, thus resulting in a runtime of $O(|V|\log|V|+|H|\log|H|+|F|\log|F|)$ instead of optimal linear time.

3.3.3 A Halfedge Data Structure Using Vectors

Definition

CGAL_Halfedge_data_structure_using_vector<V,H,F> is a model of the *Halfedge_data_structure* concept. It is parameterized with a model V of the *Vertex_base* concept, a model H of the *Halfedge_base* concept, and a model F of the *Facet_base* concept, whose requirements can be found in Section 3.4. Predefined models can be found in Section 3.5.

CGAL_Halfedge_data_structure_using_vector<V,H,F> uses an STL *vector* to store the elements. This implies random access iterators to access the elements, but no support for random insertions and deletions. The capacity is restricted to the reserved size. Allocations are not possible beyond the capacity without calling reserve again. All pointers, iterators and circulators are invalidated upon a reserve call that increases the capacity. (This restriction might be dropped in the future and the polyhedron will automatically reallocate if the capacity exceeds.)

#include <CGAL/Halfedge_data_structure_using_vector.h>

Types

It supports all types required for the *Halfedge_data_structure* concept, specifically the following two types are fixed.

typedef CGAL_Tag_false *Supports_removal;*
typedef random_access_iterator_tag *iterator_category;*

Operations

CGAL_Halfedge_data_structure_using_vector<V,H,F> complies to the requirements for the concept *Halfedge_data_structure* as stated in Section 3.2.

Implementation

It uses STL *vector* to implement the internal storage.

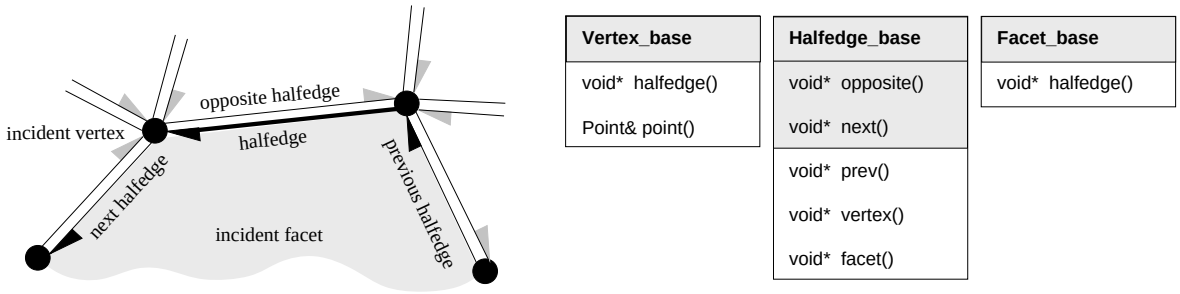


Figure 3.3: The three concepts *Vertex_base*, *Halfedge_base*, and *Facet_base* for the halfedge data structure. Member functions with shaded background are mandatory. The others are optionally supported.

3.4 Requirements for *Vertex_base*, *Halfedge_base* and *Facet_base*

Two predefined models of the *Halfedge_data_structure* concept make use of the concepts of *Vertex_base*, *Halfedge_base* and *Facet_base*, whose requirements are defined in the following subsections. The incidence relations and the mandatory and optional member functions are summarized in Figure 3.3 .

3.4.1 Requirements for *Vertex_base*

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 3.3 depicts the relationship between a halfedge and its incident halfedges, vertices, and facets.

Types

Type for (optional) associated geometry. If a point is not supported the type is *void**.

Vertex_base:: Point	point type stored in vertices.

Creation

Vertex_base v;	default constructor.
Vertex_base v(const Vertex&);	copy constructor.
Vertex_base v(const Point& p);	must exists if points are supported.

Operations

*void**	*v.halfedge()*	an incident halfedge pointing towards v.
*const void**	*v.halfedge() const*	
Point&	*v.point()*	the point.

const Point&	*v.point() const*	
void	*v.set_halfedge(void* h)*	set incident halfedge.

Types for Tagging Optional Features

The following types are equal to either *CGAL_Tag_true* or *CGAL_Tag_false*, depending on whether the named member function is supported or not.

Vertex_base:: Supports_vertex_point	*point()*.
Vertex_base:: Supports_vertex_halfedge	*halfedge()*.

3.4.2 Requirements for *Halfedge_base*

Definition

A halfedge must store pointers for the next halfedge and its opposite halfedge. It optionally stores pointer to its incident vertex and incident facet. Type tags indicate whether these member functions are supported. Figure 3.3 depicts the relationship between a halfedge and its incident halfedges, vertices, and facets. The *is_border()* predicate is mandatory, but may return always *false*.

Creation

Halfedge_base h;	default constructor.
Halfedge_base h(const Halfedge&);	copy constructor.

Operations

*void**	*h.opposite()*	the opposite halfedge.
*const void**	*h.opposite() const*	
*void**	*h.next()*	the next halfedge around the facet.
*const void**	*h.next() const*	
*void**	*h.prev()*	the previous halfedge around the facet.
*const void**	*h.prev() const*	
*void**	*h.vertex()*	the incident vertex.
*const void**	*h.vertex() const*	
*void**	*h.facet()*	the incident facet. If h is a border halfedge the result might be *NULL* or a unique facet representing this hole or a unique facet representing all holes.
*const void**	*h.facet() const*	
bool	*h.is_border() const*	is true if h is a border halfedge.
void	*h.set_next(void* next)*	
void	*h.set_prev(void* prev)*	
void	*h.set_opposite(void* h)*	
void	*h.set_vertex(void* v)*	
void	*h.set_facet(void* f)*	if f ==*NULL* h becomes a border edge.

Types for Tagging Optional Features

The following types are equal to either *CGAL_Tag_true* or *CGAL_Tag_false*, depending on whether the named member function is supported or not.

Halfedge_base:: Supports_halfedge_prev	*prev()*.
Halfedge_base:: Supports_halfedge_vertex	*vertex()*.
Halfedge_base:: Supports_halfedge_facet	*facet()*.

3.4.3 Requirements for *Facet_base*

Definition

A facet optionally stores a reference to an incident halfedge that points to the facet. Type tags indicate whether this member function is supported. Figure 3.3 depicts the relationship between a halfedge and its incident halfedges, vertices, and facets.

Creation

Facet_base f;	default constructor.
Facet_base f(const Facet&);	copy constructor.

Operations

*void**	*f.halfedge()*	an incident halfedge pointing to f.
*const void**	*f.halfedge() const*	
void	*f.set_halfedge(void* h)*	

Types for Tagging Optional Features

The nested type below is either equal to *CGAL_Tag_true* or *CGAL_Tag_false*, depending on whether the named member function is supported or not.

Facet_base:: Supports_facet_halfedge	*halfedge()*.

3.5 Models of *Vertex_base*, *Halfedge_base* and *Facet_base*

CGAL currently provides the following models for the concepts of *Vertex_base*, *Halfedge_base* and *Facet_base*. The first set of models implements the minimal set of requirements demanded for the concepts. The second set provides all described incidences and a point type for vertices. Another model for the *Facet_base* concept is tailored for polyhedral surfaces and includes a plane equation, see Section 4.5.3. These models can be used directly or they can serve as base classes for further developments.

#include <CGAL/Halfedge_data_structure_bases.h>

class CGAL_Vertex_min_base {};	defines the minimal vertex functionality with no data at all.
template <class Point> *class CGAL_Vertex_max_base {};*	defines the maximal vertex functionality including halfedge pointer and a template parameter for the point.
class CGAL_Halfedge_min_base {};	defines the minimal halfedge functionality with next and opposite pointers.
class CGAL_Halfedge_max_base {};	defines the maximal halfedge functionality including previous, vertex and facet pointers.
class CGAL_Facet_min_base {};	defines the minimal facet functionality with no data at all.
class CGAL_Facet_max_base {};	defines the maximal functionality for facets including halfedge pointer.

See Also

CGAL_Halfedge_data_structure_using_list<V,H,F> and
CGAL_Halfedge_data_structure_using_vector<V,H,F>.

3.6 Decorator for Halfedge Data Structures

The class *CGAL_Halfedge_data_structure_decorator<HDS>* provides additional functions to examine and to modify a halfedge data structure. All these functions take care of the different capabilities a certain halfedge data structure may have or may not have. The functions evaluate the support type tags of the halfedge data structure to decide on the actions. If the 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/Halfedge_data_structure_decorator.h>

Types

CGAL_Halfedge_data_structure_decorator<HDS>:: HDS	the type of the halfedge data structure.
CGAL_Halfedge_data_structure_decorator<HDS>:: Vertex	the vertex type of the halfedge data structure.
CGAL_Halfedge_data_structure_decorator<HDS>:: Halfedge	the halfedge type of the halfedge data structure.
CGAL_Halfedge_data_structure_decorator<HDS>:: Facet	the facet type of the halfedge data structure.
CGAL_Halfedge_data_structure_decorator<HDS>:: Point	the point type used in the vertex.

Creation

CGAL_Halfedge_data_structure_decorator<HDS> D;

Access Functions

*Halfedge**	*D.get_vertex_halfedge(Vertex* v)*	returns the incident halfedge of v if supported, *NULL* otherwise.
*Vertex**	*D.get_vertex(Halfedge* h)*	returns the incident vertex of h if supported, *NULL* otherwise.
*Halfedge**	*D.get_prev(Halfedge* h)*	returns the previous halfedge of h if supported, *NULL* otherwise.
*Halfedge**	*D.find_prev(Halfedge* h)*	returns the previous halfedge of h. Uses the *prev()* method if supported or performs a search around the facet using *next()*.
*Facet**	*D.get_facet(Halfedge* h)*	returns the incident facet of h if supported, *NULL* otherwise.
*Halfedge**	*D.get_facet_halfedge(Facet* f)*	returns the incident halfedge of f if supported, *NULL* otherwise.

Creation of New Primitives

*Vertex**	*D.new_vertex(HDS& hds)*	returns a new vertex from *hds* if vertices are supported, *NULL* otherwise.
*Vertex**	*D.new_vertex(HDS& hds, const Vertex* v)*	returns a copy of v from *hds* if vertices are supported, *NULL* otherwise.
*Vertex**	*D.new_vertex(HDS& hds, Point p)*	returns a new vertex from *hds* initialized to p if vertices are supported, *NULL* otherwise.
*Facet**	*D.new_facet(HDS& hds)*	returns a new facet from *poly* if facets are supported, *NULL* otherwise.
*Facet**	*D.new_facet(HDS& hds, const Facet* f)*	returns a copy of f from *hds* if facets are supported, *NULL* otherwise.

Creation of New Composed Items

*Halfedge**	*D.create_loop(HDS& hds)*	returns a halfedge from a newly created loop in *hds* consisting of a single closed edge, one vertex and two facets (if supported respectively).
*Halfedge**	*D.create_segment(HDS& hds)*	returns a halfedge from a newly created segment in *hds* consisting of a single open edge, two vertices and one facet (if supported respectively).

Removal of Elements

void	*D.delete_vertex(HDS& poly, Vertex* v)*	removes the vertex v if vertices are supported.
void	*D.delete_facet(HDS& poly, Facet* f)*	removes the facet f if facets are supported.

Modifying Functions (Composed)

void	*D.close_tip(Halfedge* h)*	makes *h->opposite()* the successor of h.
void	*D.close_tip(Halfedge* h, Vertex* v)*	makes *h->opposite()* the successor of h and sets the vertex to v.
void	*D.insert_tip(Halfedge* h, Halfedge* v)*	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 the one v points to if vertices are supported.
void	*D.remove_tip(Halfedge* h)*	removes the edge *h->next()->opposite()* from the halfedge circle around the vertex pointed to by h. The new successor halfedge of h will be *h->next()->opposite()->next()*.

void *D.insert_halfedge(Halfedge* h, Halfedge* f)*

inserts the halfedge h between f and *f->next()*. The facet of h will be the one f points to if facets are supported.

void *D.remove_halfedge(Halfedge* h)*

removes edge *h->next()* from the halfedge circle around the facet pointed to by h. The new successor of h will be *h->next()->next()*.

void *D.set_vertex_in_vertex_loop(Halfedge* h, Vertex* v)*

loops around the vertex incident to h and sets all vertex pointers to v.
Precondition: *h != NULL*.

void *D.set_facet_in_facet_loop(Halfedge* h, Facet* f)*

loops around the facet incident to h and sets all facet pointers to f.
Precondition: *h != NULL*.

*Halfedge**

D.flip_edge(Halfedge h)*

performs an edge flip, i.e. Delaunay flip. It returns h after rotating the edge h one vertex in the direction the facet orientation points to.
Precondition: *h != NULL* and the facet cycles on both sides of h are triangles.

Modifying Functions (For Border Halfedges)

*Halfedge** *D.make_hole(HDS& hds, Halfedge* h)*

removes the facet incident to h from hds and creates a hole.
Precondition: *h != NULL* and *!(h->is_border())*. If facets are supported, *Supports_removal* must be equivalent to *CGAL_Tag_true*.

*Halfedge** *D.fill_hole(HDS& hds, Halfedge* h)*

creates a new facet from hds and fills the hole incident to h.
Precondition: *h != NULL* and *h->is_border()*.

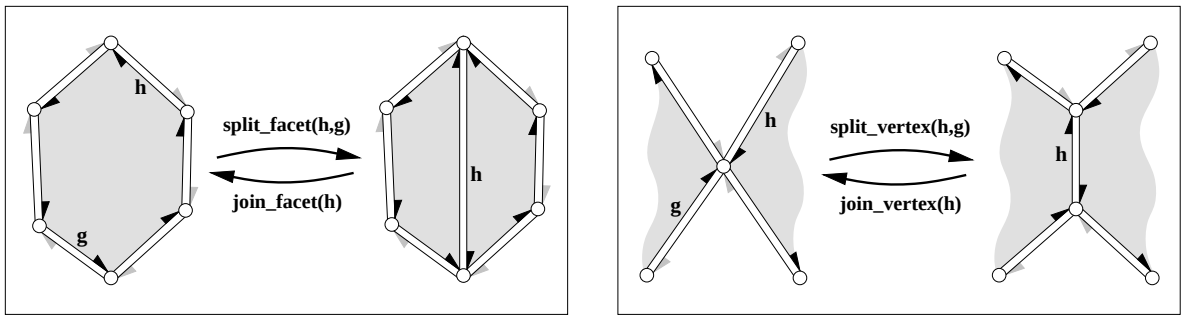
*Halfedge** *D.add_facet_to_border(HDS& hds, Halfedge* h, Halfedge* g)*

creates a new facet from hds within the hole incident to h and g by connecting the tip of g with the tip of h with a new halfedge from hds and filling this separated part of the hole with a new facet from hds, 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 != NULL*, *g != NULL*, *h->is_border()*, *g->is_border()* and g can be reached along the same 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 an *join_vertex()* operation, or an edge removal to *join_facet()*.

Given a halfedge data structure *hds* and a halfedge pointer h four special applications of the Euler operators are: *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_facet(h,h)* results in a loop directly following h. *split_facet(h,h->next())* results in a bridge parallel to the halfedge *h->next* with a new facet in-between.



*Halfedge** *D.split_facet(HDS& hds, Halfedge* h, Halfedge* g)*

split 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 obtained from *hds* is a copy of the first facet. The new diagonal is returned. The time is proportional to the distance from h to g around the facet.

*Halfedge** *D.join_facet(HDS& hds, Halfedge* h)*

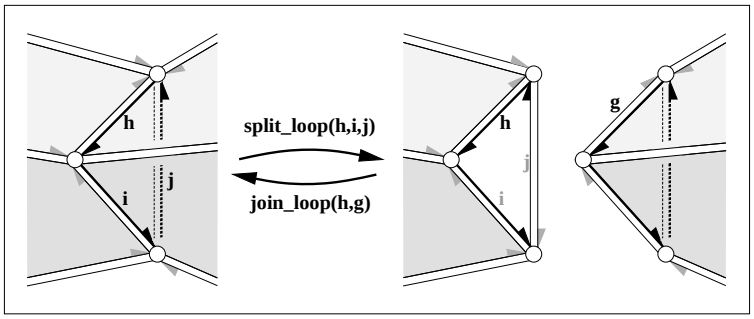
join the two facets incident to h. The facet incident to *h->opposite()* gets removed by *hds*. Both facets might be holes. Returns the predecessor of h around the facet. The invariant *join_facet(split_facet(h, g))* returns h and keeps the data structure unchanged. The time is proportional to the size of the facet removed and the time to compute *h->prev()*.
Precondition: *HDS* supports removal of facets.

*Halfedge** *D.split_vertex(HDS& hds, Halfedge* h, Halfedge* g)*

split 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. The new edge is returned. The time is proportional to the distance from h to g around the vertex.

*Halfedge** *D.join_vertex(HDS& hds, Halfedge* h)*

join the two vertices incident to h. The vertex denoted by *h->opposite()* gets removed by *hds*. Returns the predecessor of h around the vertex. 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()*.
Precondition: *HDS* supports removal of vertices.



*Halfedge** *D.split_loop(HDS& hds, Halfedge* h, Halfedge* i, Halfedge* j)*

cut the halfedge data structure into two parts along the cycle (h, i, j). Three copies of the vertices, three copies of the halfedges, and two new triangles will be created. h, i, j will be incident to the first new triangle. The return value will be a halfedge denoting the new halfedge of the second new triangle which was *h-opposite()* beforehand.
Precondition: h, i, j are 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** *D.join_loop(HDS& hds, Halfedge* h, Halfedge* g)*

glues the boundary of two facets together. Both facets and the vertices of the facet loop g gets removed. Returns h. The invariant *join_loop(h, split_loop(h, i, j))* returns h and keeps the data structure unchanged.
Precondition: *HDS* supports removal of vertices and facets. The facets denoted by h and g are different and have an equal degree (i.e. number of edges).

Modifying Functions (Primitives)

void	*D.set_point(Vertex* v, Point p)*	sets the point of v to p.
void	*D.set_point(Halfedge* h, Point p)*	sets the point of the vertex incident to h to p.
void	*D.set_vertex_halfedge(Vertex* v, Halfedge* g)*	sets the incident halfedge of v to g.
void	*D.set_vertex_halfedge(Halfedge* h)*	sets the incident halfedge of the vertex incident to h to h.

void	*D.set_vertex(Halfedge* h, Vertex* v)*	sets the incident vertex of h to v.
void	*D.set_prev(Halfedge* h, Halfedge* g)*	sets the previous link of h to g.
void	*D.set_facet(Halfedge* h, Facet* f)*	sets the incident facet of h to f.
void	*D.set_facet_halfedge(Facet* f, Halfedge* g)*	sets the incident halfedge of f to g.
void	*D.set_facet_halfedge(Halfedge* h)*	sets the incident halfedge of the facet incident to h to h.

Miscellaneous

bool *D.is_valid(HDS hds, bool verbose = false, int level = 0)*

returns *true* is the halfedge data structure *hds* according to the *level* value. If *verbose* is *true*, statistics are printed to *cerr*. 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 facet pointer for border halfedges are unequal to *NULL*. 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 facets f: the incident facet of the incident halfedge of f is equal to f. The halfedges around f starting with the incident halfedge of f form a cycle. Some checks that redundancies among internal variables holds, e.g. that the 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 is equal to the incident vertex of the opposite of the next halfedge. The incident facet of h is equal to the incident facet 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 facets must be equal to the number of all halfedges, i.e. all halfedges surrounding a facet must form a single cycle (no holes in facets).

Level 4: All tests of level 3 and *normalized_border_is_valid*.

bool *D.normalized_border_is_valid(HDS hds, bool verbose = false)*

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. 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 printed to *cerr*.

3.7 Examples of Halfedge Data Structures

3.7.1 The Default Halfedge Data Structure

The default halfedge data structure from Section 3.3.1 expects a point as template argument, which we choose to be an `int` for simplicity. The example program defines a halfedge data structure and a decorator, see Section 3.6 for the decorator. The program creates a loop, consisting of two halfedges, one vertex and two facets, and checks its validity.

```
/* hds_prog_default.C      */
/* ----------------------- */
#include <CGAL/Halfedge_data_structure_default.h>
#include <CGAL/Halfedge_data_structure_decorator.h>

typedef int                                          Point;
typedef CGAL_Halfedge_data_structure_default<Point>  HDS;
typedef CGAL_Halfedge_data_structure_decorator<HDS>  Decorator;

int main() {
    HDS hds;
    Decorator decorator;
    decorator.create_loop( hds);
    CGAL_assertion( decorator.is_valid( hds));
    return 0;
}
```

3.7.2 A Minimal Halfedge Data Structure

The following program declares a minimal structure `HDS` using the minimal bases provided in Section 3.5 and a list-bases halfedge data structure from Section 3.3.2. The result is a data structure maintaining only halfedges with next and opposite pointers. No vertices or facets are stored. The data structure models an *undirected graph*.

```
/* hds_prog_graph.C        */
/* ----------------------- */
#include <CGAL/Halfedge_data_structure_bases.h>
#include <CGAL/Halfedge_data_structure_using_list.h>
#include <CGAL/Halfedge_data_structure_decorator.h>

typedef CGAL_Halfedge_data_structure_using_list <
            CGAL_Vertex_min_base,
            CGAL_Halfedge_min_base,
            CGAL_Facet_min_base >                    HDS;
typedef CGAL_Halfedge_data_structure_decorator<HDS>  Decorator;

int main() {
    HDS hds;
    Decorator decorator;
    decorator.create_loop( hds);
    CGAL_assertion( decorator.is_valid( hds));
    return 0;
}
```

3.7.3 The Default with a Vector Instead of a List

The default halfedge data structure uses a list internally and the maximal base classes. Assuming we have a point type `Point` we can declare an alternative data structure `HDS` using the same base classes but the vector storage from Section 3.3.3. Note that for the vector storage the size of the halfedge data structure must be reserved beforehand, either with the constructor as shown in the example or with the `reserve()` member function.

```
/* hds_prog_vector.C      */
/* ---------------------- */
#include <CGAL/Halfedge_data_structure_bases.h>
#include <CGAL/Halfedge_data_structure_using_vector.h>
#include <CGAL/Halfedge_data_structure_decorator.h>

typedef int                                          Point;
typedef CGAL_Halfedge_data_structure_using_vector <
            CGAL_Vertex_max_base< Point >,
            CGAL_Halfedge_max_base,
            CGAL_Facet_max_base >                    HDS;
typedef CGAL_Halfedge_data_structure_decorator<HDS>  Decorator;

int main() {
    HDS hds(1,2,2);
    Decorator decorator;
    decorator.create_loop( hds);
    CGAL_assertion( decorator.is_valid( hds));
    return 0;
}
```

3.7.4 Example Adding Color to Facets

This example re-uses the base class available for facets and adds another member variable *color*. The facet in the halfedge data structure and data structures building upon that will inherit the new member variable, as the main function indicates. The same can be done with any kind of functionality and any of the three items; vertices, halfedges or facets. However, note that additional pointers to vertices, halfedges, and facets cannot be introduced this way, since they are only maintained as `void*` pointers at this level. The internal classes, which are responsible for the type safe pointers, must be extended to achieve this.

```
/* hds_prog_color.C       */
/* ---------------------- */
#include <CGAL/basic.h>
#include <CGAL/IO/Color.h>
#include <CGAL/Halfedge_data_structure_bases.h>
#include <CGAL/Halfedge_data_structure_using_list.h>

/* A facet with a color member variable. */
class My_facet :  public CGAL_Facet_max_base {
public:
    CGAL_Color color;
};

typedef int                                          Point;
typedef CGAL_Halfedge_data_structure_using_list <
```

```
            CGAL_Vertex_max_base< Point >,
            CGAL_Halfedge_max_base,
            My_facet >                                  HDS;

int main() {
    HDS hds;
    My_facet* f = hds.new_facet();
    f→color = CGAL_RED;
    return 0;
}
```

3.7.5 Example Extending the Minimal Halfedge Base with a Previous Pointer

The minimal halfedge data structure above can be extended to a data structure also supporting a previous pointer for halfedges. We replace the *CGAL_Halfedge_min_base* with our own *Halfedge_base* derived from the *CGAL_Halfedge_min_base*.

```
class My_halfedge : public CGAL_Halfedge_min_base {
    void* prv;
public:
    typedef  CGAL_Tag_true   Supports_halfedge_prev;
    void*       prev()                  { return prv;}
    const void* prev() const            { return prv;}
    void        set_prev( void* h)      { prv = h;}
};
```

3.7.6 Example Using own Halfedge Structure

The halfedge data structure as presented here is not as 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 side they do not need any search operations during traversals. A comparison can be found in [Ket98].

Computation time versus memory space can also be traded with the halfedge data structure design as presented here. We will use traditional C techniques like type casting and make the assumption that pointers (especially those from `malloc` or `new`) points to even addresses. The idea is as follows: The halfedge data structure allocate 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 coding the sign of this difference. We will store this bit as the least significant bit together with the next halfedge pointer. The example also omits the previous pointer. What remains are three pointers per halfedge. The same solution can be applied to the list based polyhedron traits class with the difference that the `SIZE` constant must reflect the additional pointers needed for the list management. They are added internally of the halfedge data structure using the base class *CGAL_In_place_list_base*<*My_halfedge*>.

```
/* hds_prog_compact.C    */
```

```
/* ------------------ */
#include <CGAL/Halfedge_data_structure_bases.h>
#include <CGAL/Halfedge_data_structure_using_vector.h>
#include <CGAL/Halfedge_data_structure_decorator.h>

/* Define a new halfedge class. */
class My_halfedge {
protected:
    size_t  nxt;
    void*   v;
    void*   f;
public:
    typedef CGAL_Tag_false Supports_halfedge_prev;
    typedef CGAL_Tag_true  Supports_halfedge_vertex;
    typedef CGAL_Tag_true  Supports_halfedge_facet;

    My_halfedge() :  f(NULL), nxt(0) {}

    void*       opposite()       {
        const size_t SIZE = sizeof( My_halfedge);
        if ( nxt & 1)
            return (char*)this + SIZE;
        return (char*)this - SIZE;
    }
    const void* opposite() const {
        const size_t SIZE = sizeof( My_halfedge);
        if ( nxt & 1)
            return (const char*)this + SIZE;
        return (const char*)this - SIZE;
    }
    void*       next()           { return (void*)(nxt & (~ size_t(1)));}
    const void* next() const     { return (void*)(nxt & (~ size_t(1)));}

    void*       vertex()         { return v;}
    const void* vertex() const   { return v;}

    void*       facet()          { return f;}
    const void* facet() const    { return f;}

    bool is_border() const       { return f == NULL;}

    void  set_opposite( void* g) {
        const size_t SIZE = sizeof( My_halfedge);
        char* h = (char*)g;
        CGAL_assertion( abs( h - (char*)this) == SIZE);
        if ( h > (char*)this)
            nxt |= 1;
        else
            nxt &= (~ size_t(1));
    }
    void  set_next( void* h)     {
    CGAL_assertion( ((size_t)h & 1) == 0);
        nxt = ((size_t)(h)) | (nxt & 1);
    }
    void  set_vertex( void* _v)  { v = _v;}
```

```
    void  set_facet( void* _f)   { f = _f;}
};

typedef int                                          Point;
typedef CGAL_Halfedge_data_structure_using_vector <
            CGAL_Vertex_max_base< Point >,
            My_halfedge,
            CGAL_Facet_max_base >                    HDS;
typedef CGAL_Halfedge_data_structure_decorator<HDS>  Decorator;

int main() {
    HDS hds(1,2,2);
    Decorator decorator;
    decorator.create_loop( hds);
    CGAL_assertion( decorator.is_valid( hds));
    return 0;
}
```

3.7.7 Example Using the Halfedge Iterator

Three edges are created in the default halfedge data structure. The halfedge iterator is used to count the halfedges.

```
/* hds_prog_halfedge_iterator.C */
/* ------------------------- */
#include <CGAL/Halfedge_data_structure_default.h>
#include <CGAL/Halfedge_data_structure_decorator.h>

typedef int                                          Point;
typedef CGAL_Halfedge_data_structure_default<Point>  HDS;
typedef CGAL_Halfedge_data_structure_decorator<HDS>  Decorator;
typedef HDS::Halfedge_iterator                       Halfedge_iterator;

int main() {
    HDS hds;
    Decorator decorator;
    decorator.create_loop( hds);
    decorator.create_segment( hds);
    decorator.create_loop( hds);
    CGAL_assertion( decorator.is_valid( hds));
    int n = 0;
    Halfedge_iterator begin = hds.halfedges_begin();
    for( ; begin ≠ hds.halfedges_end(); ++begin)
    ++n;
    CGAL_assertion( n == 6);  /* == 3 edges */
    return 0;
}
```

3.7.8 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 an edge iterator, which is used to count the edges.

```
/* hds_prog_edge_iterator.C */
/* ------------------------ */
#include <CGAL/Halfedge_data_structure_default.h>
#include <CGAL/Halfedge_data_structure_decorator.h>
#include <CGAL/N_step_adaptor.h>

typedef int                                          Point;
typedef CGAL_Halfedge_data_structure_default<Point>  HDS;
typedef CGAL_Halfedge_data_structure_decorator<HDS>  Decorator;
typedef HDS::Halfedge                                Halfedge;
typedef HDS::Halfedge_iterator                       Halfedge_iterator;
typedef HDS::iterator_category                       Iterator_category;
typedef CGAL_N_step_adaptor< Halfedge_iterator, 2,
                 Halfedge&, Halfedge*,
                Halfedge,  ptrdiff_t,
                Iterator_category>      Edge_iterator;

int main() {
    HDS hds;
```

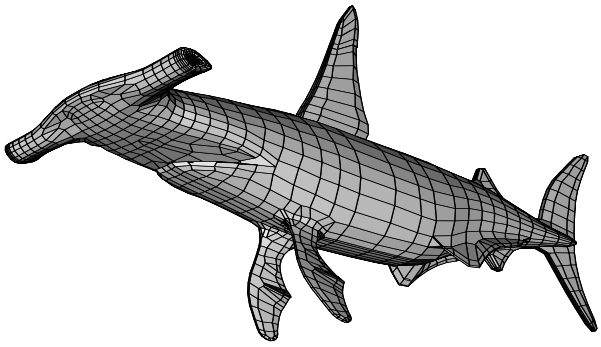
```
    Decorator decorator;
    decorator.create_loop( hds);
    decorator.create_segment( hds);
    decorator.create_loop( hds);
    CGAL_assertion( decorator.is_valid( hds));
    int n = 0;
    Edge_iterator begin = hds.halfedges_begin();
    for( ; begin ≠ hds.halfedges_end(); ++begin)
    ++n;
    CGAL_assertion( n == 3);  /* == 3 edges */
    return 0;
}
```

Chapter 4

3D-Polyhedral Surfaces

4.1 Introduction

This chapter presents polyhedral surfaces in three dimensions. They are a collection of vertices, edges and facets with 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. In the case of a closed surface we call it a *polyhedron*.



Design Rationale

Chapter 3 gives a design overview of the *Halfedge_data_structure* concept, its flexibility, predefined models and their potential use in data structures such as polyhedral surfaces, see also [Ket98]. A halfedge data structure is an edge-centered data structure. Each edge is decomposed into two halfedges with opposite orientations. Functions are provided to access incident facets and vertices. For a facet or a vertex one of the incident halfedges can be accessed. A halfedge data structure is responsible of storing the halfedges, vertices and facets, as well as maintaining their incidences through a pointer based interface.

A polyhedral surface is based on a model of the *Halfedge_data_structure* concept and adds combinatorial integrity (e.g. internal pointers cannot be simply written), abstract concepts to access the items, such as iterators and circulators, high level operations and ease-of-use, for example Euler operators. It also

adds knowledge about the geometric information kept in the data structure, such as points or plane equations, with a traits class.

The class *CGAL_Polyhedron* can store polyhedrons and polyhedral surfaces. The template argument for the *Halfedge_data_structure* allows to choose among the different possible models, for example a list or 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 provided default model for the halfedge data structure *CGAL_Halfedge_data_structure_polyhedron_default_3<R>* chooses the list representation and *CGAL_Point_3<R>* for the geometric information stored with vertices and *CGAL_Plane_3<R>* for the geometric information stored with facets.

A utility class *CGAL_Polyhedron_incremental_builder_3* helps in creating polyhedral surfaces from a list of points followed by a list of facets represented as indices into the point list. This is particularly useful in combination with usual file formats for polyhedrons.

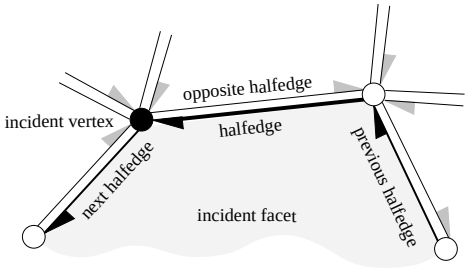
Organization of this Chapter

Section 4.2 introduces the polyhedral surface class *CGAL_Polyhedron_3*, its three local classes *Vertex*, *Halfedge*, and *Facet*. Section 4.3 defines the *Polyhedron_traits* concept, which adds the geometric knowledge to the polyhedral surface, and Section 4.4 names the provided models for this concept. Section 4.5 presents additional requirements for the *Halfedge_data_structure* concept that were recognized by the polyhedral surface and states the default models that fulfills these requirements, see also Chapter 3 on halfedge data structure. Section 4.6 continues with the description of the utility class for incremental construction of polyhedral surfaces. Section 4.7 concludes this chapter with examples using polyhedral surfaces.

4.2 Polyhedral Surfaces (*CGAL_Polyhedron_3<Traits,HDS>*)

Definition

A polyhedral surface *CGAL_Polyhedron_3<Traits,HDS>* 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.



Vertices represent points in 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 is allowed to have holes. The halfedges along the boundary of a hole are called *border halfedges* and have no incident facets. 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 polyhedrons 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).

One implication of the 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 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. *CGAL_Polyhedron_3<Traits,HDS>* 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 *CGAL_Polyhedron_3<Traits,HDS>* can represent polyhedral surfaces as well as polyhedrons. The interface is designed in such a way that it is easy to ignore border edges and work only with polyhedrons.

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 came first and the halfedge incident to the hole came 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 during further modifications. There is no automatic caching done for auxiliary information.

The class *CGAL_Polyhedron_3<Traits,HDS>* expects a model of the *Polyhedron_traits* concept from Section 4.3 as a first template argument, and a model of the *Halfedge_data_structure* concept from Section 3.2 with the additional requirements from Section 4.5 as a second template argument

#include <CGAL/Polyhedron_3.h>

Types

CGAL_Polyhedron_3<Traits,HDS>:: Traits	traits class.
CGAL_Polyhedron_3<Traits,HDS>:: Halfedge_data_structure	halfedge data structure *HDS*.
CGAL_Polyhedron_3<Traits,HDS>:: Vertex	vertex type.
CGAL_Polyhedron_3<Traits,HDS>:: Halfedge	halfedge type.
CGAL_Polyhedron_3<Traits,HDS>:: Facet	facet type.
CGAL_Polyhedron_3<Traits,HDS>:: Size	type for size values.
CGAL_Polyhedron_3<Traits,HDS>:: Difference	type for difference values.

Types for the (optionally) associated geometry. If the specific item is not supported the type is *void**.

CGAL_Polyhedron_3<Traits,HDS>:: Point	point stored in vertices.
CGAL_Polyhedron_3<Traits,HDS>:: Plane	plane equation stored in facets.
CGAL_Polyhedron_3<Traits,HDS>:: Normal	normal vector stored in facets.

The following handles, iterators and circulators have appropriate non-mutable counterparts. 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 appropriate iterator can be used as well.

The iterator category is defined through the polyhedron traits class for all iterators. The circulators are bidirectional if the halfedge in the polyhedron traits class provides a member function *prev()*, otherwise they are of the forward category.

CGAL_Polyhedron_3<Traits,HDS>:: Vertex_handle	handle to vertex.
CGAL_Polyhedron_3<Traits,HDS>:: Halfedge_handle	handle to halfedge.
CGAL_Polyhedron_3<Traits,HDS>:: Facet_handle	handle to facet.

CGAL_Polyhedron_3<Traits,HDS>:: Vertex_iterator	iterator over all vertices.
CGAL_Polyhedron_3<Traits,HDS>:: Halfedge_iterator	iterator over all halfedges.
CGAL_Polyhedron_3<Traits,HDS>:: Facet_iterator	iterator over all facets.

CGAL_Polyhedron_3<Traits,HDS>:: Halfedge_around_vertex_circulator

circulator of halfedges around a vertex.

CGAL_Polyhedron_3<Traits,HDS>:: Halfedge_around_facet_circulator

circulator of halfedges around a facet.

CGAL_Polyhedron_3<Traits,HDS>:: iterator_category	iterator category of all the iterators.
CGAL_Polyhedron_3<Traits,HDS>:: circulator_category	circulator category of all the circulators.

Creation

CGAL_Polyhedron_3<Traits,HDS> P(Traits traits = Traits());

CGAL_Polyhedron_3<Traits,HDS> P(Size v, Size h, Size 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 v, Size h, Size 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 an arbitrary 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 with p_1, p_2, p_3 and p_4. Returns that halfedge of the tetrahedron which incident vertex is initialized with p_1, the incident vertex of the next halfedge with p_2, and the vertex thereafter with p_3. The remaining fourth vertex is initialized with p_4

Halfedge_handle *P.make_triangle()*

a triangle with border edges is added to the polyhedral surface. Returns an arbitrary, 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 with p_1, p_2 and p_3. Returns that non-border halfedge of the triangle which incident vertex is initialized with p_1, the incident vertex of the next halfedge with p_2, and the vertex thereafter with p_3.

Access Member Functions

Size	*P.size_of_vertices()*	number of vertices.
Size	*P.size_of_halfedges()*	number of halfedges (incl. border halfedges).
Size	*P.size_of_facets()*	number of facets.
Size	*P.capacity_of_vertices()*	space reserved for vertices.
Size	*P.capacity_of_halfedges()*	space reserved for halfedges.
Size	*P.capacity_of_facets()*	space reserved for facets.
size_t	*P.bytes()*	bytes used for the polyhedron.
size_t	*P.bytes_reserved()*	bytes reserved for the polyhedron.
Vertex_iterator	*P.vertices_begin()*	iterator over all vertices.
Vertex_iterator	*P.vertices_end()*	past-the-end iterator and null-handle.
Halfedge_iterator	*P.halfedges_begin()*	iterator over all halfedges.
Halfedge_iterator	*P.halfedges_end()*	past-the-end iterator and null-handle.
Facet_iterator	*P.facets_begin()*	iterator over all facets (excluding holes).
Facet_iterator	*P.facets_end()*	past-the-end iterator and null-handle.
Traits	*P.traits()*	returns the traits class.

Geometric Predicates

bool *P.is_triangle(Halfedge_const_handle h)*

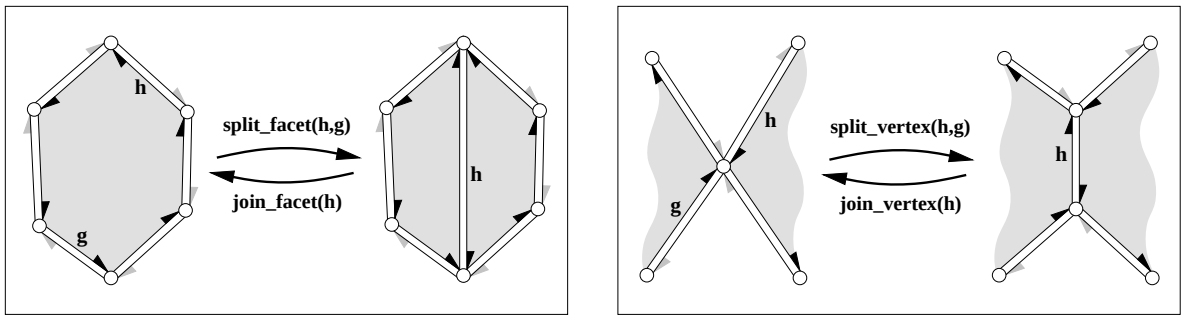
true iff the connected component denoted by h is a triangle.

bool *P.is_tetrahedron(Halfedge_const_handle h)*

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)*

split 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. It returns the new diagonal. 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)*

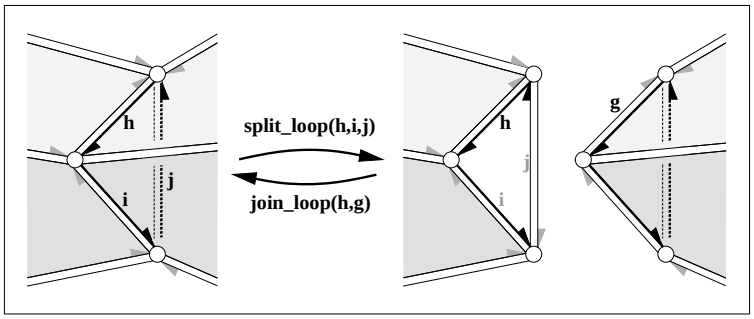
join the two facets incident to h. The facet incident to h*->opposite()* gets removed. Both facets might be holes. Returns the predecessor of h around the facet. The invariant *join_facet(split_facet(h, g))* returns h and keeps the polyhedron unchanged. The time is proportional to the size of the facet removed and the time to compute h*->prev()*.
Precondition: *Traits* supports removal of facets. The degree of both vertices incident to h is at least three (no antennas).

Halfedge_handle *P.split_vertex(Halfedge_handle h, Halfedge_handle g)*

split the vertex incident to h and g into two vertices and connects them with a new edge. The second (new) vertex is a copy of the first vertex. It returns the new edge. 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 (no antennas).

Halfedge_handle *P.join_vertex(Halfedge_handle h)*

join the two vertices incident to h. The vertex denoted by h*->opposite()* gets removed. Returns the predecessor of h around the vertex. 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()*.
Precondition: *Traits* supports removal of vertices. The size of both facets incident to h is at least four (no multi-edges).



Halfedge_handle *P.split_loop(Halfedge_handle h, Halfedge_handle i, Halfedge_handle j)*

cut the polyhedron into two parts along the cycle (h, i, j). Three copies of the vertices, three copies of the halfedges, and two new triangles will be created. h, i, j will be incident to the first new triangle. The return value will be a halfedge iterator denoting the new halfedge of the second new triangle which was *h->opposite()* beforehand.
Precondition: h, i, j are 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 two facets together. Both facets and the vertices of the facet loop g gets removed. Returns h. The invariant *join_loop(h, split_loop(h, i, j))* returns h and keeps the polyhedron unchanged.
Precondition: *Traits* supports removal of vertices and facets. The facets denoted by h and g are different and have an equal degree (i.e. number of edges) ≥ 3.

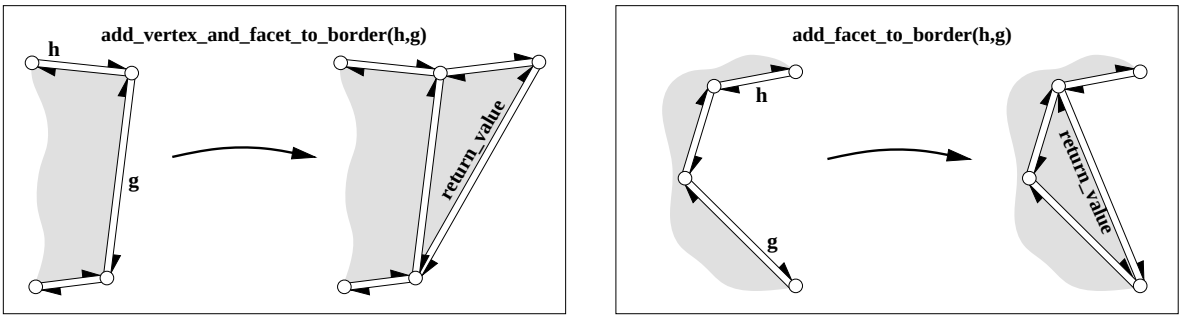
Modifying Facets and Holes

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.
Precondition: *Traits* supports removal of facets, and *! h.is_border()*.

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 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.
Precondition: *h->is_border()*, *g->is_border()*, *h != g*, *h->next() != g*, and g can be reached along the same hole starting with h.

Operations with Border Halfedges

Halfedges incident to an 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 precedes 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 may invalidate these values, which are not automatically updated.

void P.normalize_border()

sorts halfedges such that the non-border edges precedes 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 *P.size_of_border_halfedges()*

number of 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 *P.size_of_border_edges()*

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: *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

P.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.

Miscellaneous

void *P.inside_out()* reverse facet orientation (incl. normal or plane equations).

void *P.delegate(Modifier_base<HDS>& m)*

calls the *operator()* of the modifier m. See *CGAL_Modifier_base* in the Support Library Manual for a description of modifier design and its usage.
Precondition: The polyhedral surface must be valid when the modifier returns from execution.

bool *P.is_valid(bool verbose = false, int level = 0)*

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 Section 3.6 and that each facet is at least a triangle and that the incident facets of a non-border edge are distinct.

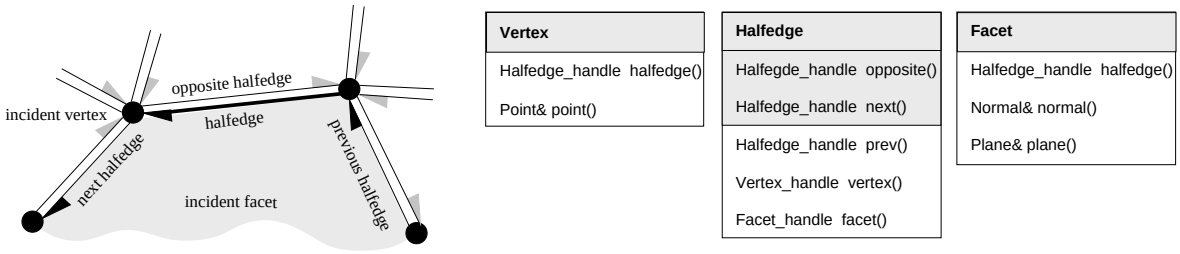


Figure 4.1: The three classes *Vertex*, *Halfedge*, and *Facet* of the polyhedral surface. Member functions with shaded background are mandatory. The others are optionally supported.

bool *P.normalized_border_is_valid(_HDS hds, bool verbose = false)*

returns *true* if the border halfedges are in normalized representation, which is when enumerating all halfedges with the iterator: The non-border edges precedes 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*.

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.

CGAL_Polyhedron_3<Traits,HDS>::Supports_vertex_halfedge	*Vertex::halfedge()*.
CGAL_Polyhedron_3<Traits,HDS>::Supports_vertex_point	*Vertex::point()*.
CGAL_Polyhedron_3<Traits,HDS>::Supports_halfedge_prev	*Halfedge::prev()*.
CGAL_Polyhedron_3<Traits,HDS>::Supports_halfedge_vertex	*Halfedge::vertex()*.
CGAL_Polyhedron_3<Traits,HDS>::Supports_halfedge_facet	*Halfedge::facet()*.
CGAL_Polyhedron_3<Traits,HDS>::Supports_facet_halfedge	*Facet::halfedge()*.
CGAL_Polyhedron_3<Traits,HDS>::Supports_facet_plane	*Facet::plane()*.
CGAL_Polyhedron_3<Traits,HDS>::Supports_facet_normal	*Facet::normal()*.
CGAL_Polyhedron_3<Traits,HDS>::Supports_removal	supports the removal of individual elements of a surface.

4.2.1 Vertex of a Polyhedral Surface

Definition

A vertex optionally stores a point and a reference to an incident halfedge that points to the vertex. Type tags as defined in *CGAL_Polyhedron_3* indicate whether these member functions are supported. Figure 4.1 depicts the relationship between a halfedge and its incident halfedges, vertices, and facets.

Types

Vertex defines the same types as *CGAL_Polyhedron_3* except the traits class.

Access Functions

Halfedge_handle	*v.halfedge()*	an incident halfedge pointing to v.
Point&	*v.point()*	the point.

Halfedge_around_vertex_circulator

	v.vertex_begin()	circulator of halfedges around the vertex (clockwise).

4.2.2 Halfedge of a Polyhedral Surface

Definition

Figure 4.1 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 its counterpart that has the opposite direction. They are mutually linked with the *opposite()* member function. 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 some hole (which might not be uniquely determined in situations with more than one hole at a vertex). An invariant is that successive assignments of the form h = *h->next()* cycles counterclockwise around the facet and traverses all halfedges incident to this facet. A similar invariant is that successive assignments of the form h = *h->next()->opposite()* cycles clockwise around the vertex and traverses all halfedges incident to this vertex. Two circulators are provided for these circular orders. *next()* and *opposite()* are mandatory functions for each instantiation of polyhedral surfaces. The other member functions are optionally as indicated with the type tags defined in *CGAL_Polyhedron_3*, see Section 4.2. The *prev()* member function points to the preceding halfedge around the same facet. Handles to the incident vertex and facet are optionally stored.

Types

Halfedge defines the same types as *CGAL_Polyhedron_3* except the traits class.

Access Functions

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.
Vertex_handle	*h.vertex()*	the incident vertex.
Facet_handle	*h.facet()*	the incident facet. If h is a border halfedge the result is a singular value for the handle.
Halfedge_handle	*h.next_on_vertex()*	the next halfedge around the vertex (clockwise). Is equal to *h.next()->opposite()*.
Halfedge_handle	*h.prev_on_vertex()*	the previous halfedge around the vertex (counterclockwise). Is equal to *h.opposite()->prev()*.

Halfedge_around_vertex_circulator

	h.vertex_begin()	circulator of halfedges around the vertex (clockwise).

Halfedge_around_facet_circulator

	h.facet_begin()	circulator of halfedges around the facet (counterclockwise).

bool	*h.is_border()*	is true if h is a border halfedge.
bool	*h.is_border_edge()*	is true if h or *h.opposite()* is a border halfedge.

Implementation

The methods *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.

4.2.3 Facet of a Polyhedral Surface

Definition

A facet optionally stores a normal vector, a plane, and a reference to an incident halfedge that points to the facet. If a plane is supported the normal is taken from the plane equation. The type tags as defined in *CGAL_Polyhedron_3* indicates whether these member functions are supported, see Section 4.2. Figure 4.1 depicts the relationship between a halfedge and its incident halfedges, vertices, and facets.

Types

Facet defines the same types as *CGAL_Polyhedron_3* except the traits class.

Access Functions

Halfedge_handle	*f.halfedge()*	an incident halfedge pointing to f.
Normal`&`	*f.normal()*	the normal vector. If a plane equation is supported, only a value of the normal vector is returned (no reference).
Plane`&`	*f.plane()*	the supporting plane of the facet.

Halfedge_around_facet_circulator

	f.facet_begin()	circulator of halfedges around the facet (counterclockwise).

4.3 Requirements for a *Polyhedron_traits*

Definition

A model for the *Polyhedron_traits* concept must provide the following types, operations and restrictions with respect to the halfedge data structure *HDS* used in the polyhedral surface.

Types

Polyhedron_traits:: Point	point type equal to *HDS::Point*, if points are supported.
Polyhedron_traits:: Normal	normal vector equal to *HDS::Facet::Normal*, if normals are supported.
Polyhedron_traits:: Plane	plane equation equal to *HDS::Facet::Plane*, if plane equations are supported.

Operations

void *traits.reverse_normal(Normal& n) const*

reverses the orientation of the normal vector n. Only needed if normal vectors are supported.

void *traits.reverse_plane(Plane& p) const*

reverses the orientation of the plane equation p. Only needed if plane equations are supported.

See Also

CGAL_Polyhedron_3 and *CGAL_Polyhedron_default_traits_3*.

4.4 Models of *Polyhedron_traits*

Definition

CGAL_Polyhedron_default_traits_3<R> is a model of the *Polyhedron_traits* concept from Section 4.3 based on the CGAL-kernel. It defines *CGAL_Point_3<R>* as points, *CGAL_Vector_3<R>* as surface normal vectors, and *CGAL_Plane_3<R>* as plane equations.

#include <CGAL/Polyhedron_default_traits_3.h>

4.5 Additional Requirements and a Default Model for a *Halfedge_data_structure*

The additional requirements allow to work with normal vectors or plane equations associated to facets as flexible as with the point type associated to vertices. The storage of either a normal vector or a plane equation in a facet is optional.

4.5.1 Additional Requirements

Definition

A halfedge data structure used for polyhedral surfaces must be a model of the *Halfedge_data_structure* concept as described in Section 3.2 and additionally provides the following types and operations for the local *Facet* type to support optionally surface normals or plane equations for facets.

Types

Types for (optionally) associated geometry in facets. If they are not supported the respective types are *void**.

Facet:: Normal	surface normal vector stored in facets.
Facet:: Plane	plane equation stored in facets.

Operations

The member functions for the normal vector or the plane equation are only needed if the respective feature is supported. If plane equations are supported, a member function for the normal vector is necessary, though only with a value semantic of the return value (no reference as return value).

Normal&	*f.normal()*	the surface normal vector.
const Normal&	*f.normal() const*	
Plane&	*f.plane()*	the plane equation.
const Plane&	*f.plane() const*	

Types for Tagging Optional Features

The nested types below are either equal to *CGAL_Tag_true* or *CGAL_Tag_false*, depending on whether the named member function is supported or not.

Facet:: Supports_facet_normal	*normal()*.
Facet:: Supports_facet_plane	*plane()*.

The following dependencies among these options and those of the *Halfedge* type must be regarded:

$Supports\_facet\_normal \equiv CGAL\_Tag\_true \Longrightarrow Supports\_halfedge\_facet \equiv CGAL\_Tag\_true.$
$Supports\_facet\_plane \equiv CGAL\_Tag\_true \Longrightarrow Supports\_halfedge\_facet \equiv CGAL\_Tag\_true.$

4.5.2 The Default Halfedge Data Structure for Polyhedrons

Definition

The default *CGAL_Halfedge_data_structure_polyhedron_default_3<R>* implements all incidences supported by the *Halfedge_data_structure* concept. It chooses geometry types from the CGAL-kernel as determined with the representation class *R*. It stores a point of type *CGAL_Point_3<R>* in each vertex and a plane equation of type *CGAL_Plane_3<R>* in each facet.

#include <CGAL/Halfedge_data_structure_polyhedron_default_3.h>

Implementation

The default halfedge data structure provided for polyhedrons is a wrapper for the class *CGAL_Halfedge_data_structure_using_list* using the *CGAL_Vertex_max_base<CGAL_Point_3<R>>*, *CGAL_Halfedge_max_base* and *CGAL_Polyhedron_facet_base_3* models, see Section 3.3.2 to 3.5 and 4.5.3.

4.5.3 Models of *Facet_base*

CGAL currently provides a model for the *Facet_base* concept specifically tailored for polyhedral surfaces. Other models can be found in Section 3.5.

template <class R>
class CGAL_Polyhedron_facet_base_3 {}; — defines the maximal facet functionality for polyhedrons including halfedge pointer and a plane equation of type *CGAL_Plane_3<R>*.

See Also

CGAL_Vertex_min_base, ..., *CGAL_Facet_max_base*, and *CGAL_Halfedge_data_structure_polyhedron_default_3*.

4.6 An Incremental Builder for Polyhedral Surfaces

Definition

CGAL_Polyhedron_incremental_builder_3<HDS> is an auxiliary class that supports the incremental construction of polyhedral surfaces. This is for example convenient when constructing polyhedral surfaces from file formats like the Object File Format (OFF) [Phi94], OpenInventor [Wer94] or VRML [BPP95, VRML96]. *CGAL_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* in the Support Library Manual.

The incremental builder might be of broader interest for other halfedge data structures as well, 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 printed 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*). The correct protocol of method calls to build a polyhedral surface is given as a regular expression below. Vertices and facets can be added in arbitrary order as long as *add_vertex_to_facet()* refers only to vertex indices that are already known.

begin_surface (*add_vertex* | (*begin_facet add_vertex_to_facet*$*$ *end_facet*)) $*$ *end_surface*

Types

CGAL_Polyhedron_incremental_builder_3<HDS>:: Halfedge_data_structure

halfedge data structure *HDS*.

CGAL_Polyhedron_incremental_builder_3<HDS>:: Point its point type.
CGAL_Polyhedron_incremental_builder_3<HDS>:: Size its size type.

Creation

CGAL_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.

Operations

void B.begin_surface(Size v, Size f, Size h = 0)

starts the construction. v is the number of vertices to expect, f the number of facets, and h the number of 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 polyhedrons. If the representation does not support insertion the object must fit into the reserved sizes.

void B.add_vertex(Point p) adds p to the vertex list.
void B.begin_facet() starts a facet.

void	*B.add_vertex_to_facet(Size i)*	adds a vertex with index i to the current facet. The first point added with *add_vertex()* has the index 0.
void	*B.end_facet()*	ends a facet.
void	*B.end_surface()*	ends the construction.
bool	*B.error()*	returns error status of the builder.
void	*B.rollback()*	undoes all changes made to the halfedge data structure since the last *begin_surface()*.
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.

4.7 Examples Using Polyhedral Surfaces

Examples of different halfedge data structures can be found in Section 3.7.

4.7.1 First Example Using Defaults

The first example instantiates a polyhedron using the default traits class, the default halfedge data structure, and creates a tetrahedron.

```
/* polyhedron_prog_simple.C */
/* ------------------------ */
#include <CGAL/Cartesian.h>
#include <CGAL/Halfedge_data_structure_polyhedron_default_3.h>
#include <CGAL/Polyhedron_default_traits_3.h>
#include <CGAL/Polyhedron_3.h>

typedef CGAL_Cartesian<double>                                  R;
typedef CGAL_Halfedge_data_structure_polyhedron_default_3<R>    HDS;
typedef CGAL_Polyhedron_default_traits_3<R>                    Traits;
typedef CGAL_Polyhedron_3<Traits,HDS>                          Polyhedron;
typedef Polyhedron::Halfedge_handle                            Halfedge_handle;

int main() {
    Polyhedron P;
    Halfedge_handle h = P.make_tetrahedron();
    CGAL_assertion( P.is_tetrahedron( h));
    return 0;
}
```

4.7.2 Example with Geometry in Vertices

This example creates a tetrahedron initialized with four points. In addition it demonstrates the use of the vertex iterator and the access to the point in the vertices. The output of the program will be `(0 0 0) (1 0 0) (0 1 0) (0 0 1)`.

```
/* polyhedron_prog_tetra.C */
/* ----------------------- */
#include <CGAL/Cartesian.h>
#include <iostream.h>
#include <CGAL/Halfedge_data_structure_polyhedron_default_3.h>
#include <CGAL/Polyhedron_default_traits_3.h>
#include <CGAL/Polyhedron_3.h>

typedef CGAL_Cartesian<double>                                  R;
typedef CGAL_Halfedge_data_structure_polyhedron_default_3<R>    HDS;
typedef CGAL_Polyhedron_default_traits_3<R>                     Traits;
typedef CGAL_Polyhedron_3<Traits,HDS>                           Polyhedron;
typedef Polyhedron::Point                                       Point;
typedef Polyhedron::Vertex_iterator                             Vertex_iterator;

int main() {
    Point p( 0.0, 0.0, 0.0);
    Point q( 1.0, 0.0, 0.0);
    Point r( 0.0, 1.0, 0.0);
    Point s( 0.0, 0.0, 1.0);

    Polyhedron P;
    P.make_tetrahedron( p, q, r, s);
    CGAL_set_ascii_mode( cout);
    Vertex_iterator begin = P.vertices_begin();
    for ( ; begin ≠ P.vertices_end(); ++begin)
        cout ≪ "(" ≪ begin→point() ≪ ") ";
    cout ≪ endl;
    return 0;
}
```

4.7.3 Example Declaring a Point Iterator

It might be preferable to have an iterator enumerating all points directly instead of all vertices as in the previous example. Such an iterator could be used in other algorithms, for example a convex hull computation. The following declaration gives us such a point iterator.

```
#include <CGAL/Iterator_project.h>
#include <CGAL/function_objects.h>

typedef Polyhedron::Vertex                                      Vertex;
typedef Polyhedron::Vertex_iterator                             Vertex_iterator;
typedef Polyhedron::Point                                       Point;
typedef Polyhedron::Difference                                  Difference;
typedef Polyhedron::iterator_category                           iterator_category;
typedef CGAL_Project_point<Vertex>                              Project_point;
typedef CGAL_Iterator_project<Vertex_iterator, Project_point,
        Point&, Point*, Difference, iterator_category>          Point_iterator;
```

The `for`-loop of the previous example could now be replaced with the following loop:

```
Point_iterator begin = P.vertices_begin();
for ( ; begin ≠ P.vertices_end(); ++begin)
    cout ≪ "(" ≪ (*begin) ≪ ") ";
```

4.7.4 Example Writing Object File Format (OFF) with STL Algorithms

The following example creates a tetrahedron and writes it to *cout* using the Object File Format (OFF) [Phi94]. The example makes advanced use of STL algorithms (*copy*, *distance*), STL *ostream_iterator* and CGAL circulators.

```
/* polyhedron_prog_off.C  */
/* ---------------------- */
#include <CGAL/Cartesian.h>
#include <iostream.h>
#include <CGAL/Halfedge_data_structure_polyhedron_default_3.h>
#include <CGAL/Polyhedron_default_traits_3.h>
#include <CGAL/Polyhedron_3.h>
#include <CGAL/Iterator_project.h>
#include <CGAL/function_objects.h>

typedef CGAL_Cartesian<double>                                  R;
typedef CGAL_Halfedge_data_structure_polyhedron_default_3<R>  HDS;
typedef CGAL_Polyhedron_default_traits_3<R>                  Traits;
typedef CGAL_Polyhedron_3<Traits,HDS>                        Polyhedron;
typedef Polyhedron::Difference                               Difference;
typedef Polyhedron::iterator_category                        Iterator_category;
typedef Polyhedron::Point                                    Point;
typedef Polyhedron::Vertex                                   Vertex;
typedef Polyhedron::Vertex_iterator                          Vertex_iterator;
typedef Polyhedron::Facet_iterator                           Facet_iterator;
typedef Polyhedron::Halfedge_around_facet_circulator
                                        Halfedge_around_facet_circulator;

typedef CGAL_Project_point<Vertex>                            Project_point;
typedef CGAL_Iterator_project<Vertex_iterator, Project_point,
        Point&, Point*, Difference, Iterator_category>       Point_iterator;

int main() {
    Point p( 0.0, 0.0, 0.0);
    Point q( 1.0, 0.0, 0.0);
    Point r( 0.0, 1.0, 0.0);
    Point s( 0.0, 0.0, 1.0);

    Polyhedron P;
    P.make_tetrahedron( p, q, r, s);

    /* Write polyhedron on Object File Format (OFF). */
    CGAL_set_ascii_mode( cout);
    cout ≪ "OFF" ≪ endl;
    cout ≪ P.size_of_vertices() ≪ ' ' ≪ P.size_of_facets() ≪ " 0" ≪ endl;
    copy( Point_iterator( P.vertices_begin()),
      Point_iterator( P.vertices_end()),
```

```
    ostream_iterator<Point>(cout,"\n"));
  Facet_iterator i = P.facets_begin();
  for ( ; i ≠ P.facets_end(); ++i) {
      Halfedge_around_facet_circulator j = i→facet_begin();
      /* Facets in polyhedral surfaces are at least triangles. */
      CGAL_assertion( CGAL_circulator_size(j) ≥ 3);
      cout ≪ CGAL_circulator_size(j) ≪ " ";
      do {
          size_t d = 0;
          distance( P.vertices_begin(), j→vertex(), d);
          cout ≪ " " ≪ d;
      } while ( ++j ≠ i→facet_begin());
      cout ≪ endl;
  }
  return 0;
}
```

4.7.5 Example Using the Incremental Builder and Modifier Mechanism

The *CGAL_Polyhedron_incremental_builder_3* class is used to create a triangle using the modifier mechanism as described in the Support Library Manual.

```
/* polyhedron_prog_incr_builder.C */
/* ------------------------------ */
#include <CGAL/Cartesian.h>
#include <CGAL/Halfedge_data_structure_polyhedron_default_3.h>
#include <CGAL/Polyhedron_default_traits_3.h>
#include <CGAL/Polyhedron_incremental_builder_3.h>
#include <CGAL/Polyhedron_3.h>

typedef CGAL_Cartesian<double>                                  R;
typedef CGAL_Halfedge_data_structure_polyhedron_default_3<R>  HDS;
typedef CGAL_Polyhedron_default_traits_3<R>                  Traits;
typedef CGAL_Polyhedron_3<Traits,HDS>                        Polyhedron;

/* 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::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();
```

```
    }
};

int main() {
    Polyhedron P;
    Build_triangle<HDS> triangle;
    P.delegate( triangle);
    CGAL_assertion( P.is_triangle( P.halfedges_begin()));
    return 0;
}
```

Chapter 5

Planar Map

5.1 Introduction

Our package is concerned with subdivisions of the plane induced by collections of curves. In this section we briefly review the geometric concepts underlying the data structures described in the following sections.

Curve We will use the term *curve* to describe the image of a continuous 1–1 mapping into the plane of any one of the following: the closed unit interval (arc), the open unit interval (unbounded curve), or the unit circle (closed curve). In all cases a curve is non self-intersecting.

x-Monotone curve We say that a curve c is x-monotone if c either intersects any vertical line in at most one point, or that c is a vertical segment.

Planar subdivision (planar-map), vertex, edge, face A planar subdivision (or planar-map) is an embedding of a planar graph G into the plane, such that each arc of G is embedded as a bounded

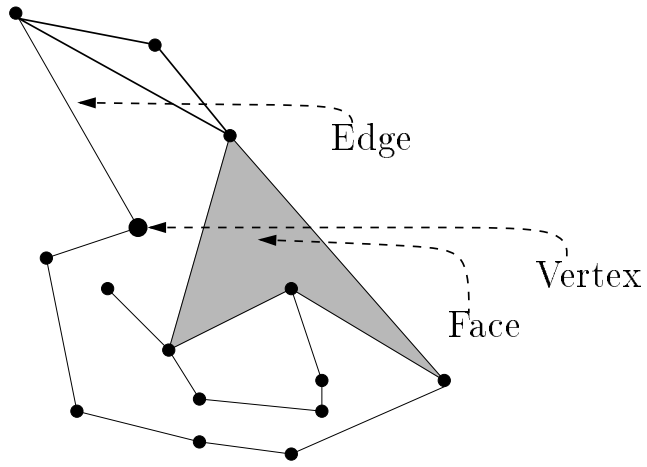


Figure 5.1: A face, an edge, and a vertex

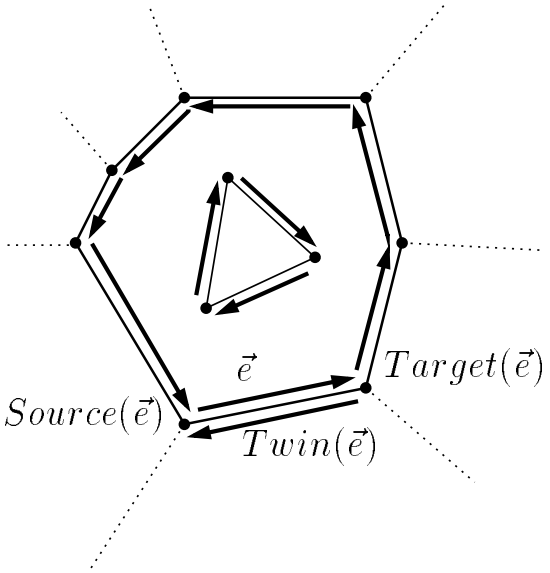


Figure 5.2: DCEL

curve. The image of a node of G is called a *vertex*, and the image of an arc is called an *edge*. In this embedding no pair of edges intersect in their interiors. A *face* of the subdivision is a maximal connected region of the plane that does not contain any vertex or edge. See Figure 5.1. We consider a face to be open, and its boundary is formed by vertices and edges of the subdivision.

Incidence If a vertex v is an endpoint of an edge e, then we say that v and e are *incident* to one another. Similarly, a face and an edge on its boundary are incident, and a face and a vertex on its boundary are incident.

Half-edge, twin, source, target We consider each edge e to be two-sided, representing it by two directed *half-edges* $\vec{e}$ and Twin($\vec{e}$). A half-edge $\vec{e}$ is an ordered pair (u, v) of its endpoints, and it is directed from u, the *source*, to v, the *target*. We consider each half edge to lie on the border of a single face—the face lying to our left as we traverse the edge from source to target.

Connected Component of the Boundary (CCB) Each connected component of the boundary of a face is represented by a circular list of half-edges. The list of half-edges of the outer boundary component of a face is oriented counterclockwise, and the list for each inner boundary component is oriented clockwise, see Figure 5.2. For a face f of a planar-map, we call each connected component of the boundary of f a *CCB*. If f is bounded, we call its outer boundary component the outer-CCB. Any other connected component of the boundary of f is called a hole.

Edges Around a Vertex Every maximal group of half-edges which share the same origin is represented by a special object. It can be viewed as a circular list of half-edges ordered clockwise around their origin vertex.

Doubly connected edge list (DCEL) For a planar-map, its *DCEL* representation consists of a connected list of half-edges for every CCB of every face in the subdivision, with additional incidence

information that enables us to traverse the subdivision. In particular, for each half-edge the DCEL stores a pointer to its twin edge and to the next clockwise/counterclockwise half-edge with the same source. In addition, for each half-edge the DCEL stores a pointer to the incident face. See Figure 5.2. For more information about the DCEL representations see [dBvKOS97]. In the following specifications, we implement the subdivision by a DCEL.

5.2 Planar Map

5.2.1 2D Planar Map (*CGAL_Planar_map_2<Dcel, Traits>*)

Definition

An object *pm* of the class *CGAL_Planar_map_2<Dcel, Traits>* is the planar subdivision induced by a set of x-monotone curves that do not intersect in their interiors. The available traits and dcel classes will be described later.

Types

CGAL_Planar_map_2<Dcel, Traits>:: Vertex

Represents a vertex of the planar-map.

CGAL_Planar_map_2<Dcel, Traits>:: Halfedge

Represents a half-edge of the planar-map.

CGAL_Planar_map_2<Dcel, Traits>:: Face

Represents a face of the planar-map.

CGAL_Planar_map_2<Dcel, Traits>:: Vertex_iterator

A bidirectional iterator over the vertices of the planar-map. Its value-type is *Vertex*.

CGAL_Planar_map_2<Dcel, Traits>:: Halfedge_iterator

A bidirectional iterator over the halfedges of the planar-map. Its value-type is *Halfedge*.

CGAL_Planar_map_2<Dcel, Traits>:: Face_iterator

A bidirectional iterator over the faces of the planar-map. Its value-type is *Face*.

CGAL_Planar_map_2<Dcel, Traits>:: Ccb_halfedge_circulator

A forward circulator over the edges of a CCB (connected components of the boundary). Its value-type is *Halfedge*.

CGAL_Planar_map_2<Dcel, Traits>:: Halfedge_around_vertex_circulator

A forward circulator over the half-edges which have the vertex as their source. The half-edges are traversed in their clockwise order around the vertex. Its value-type is *Halfedge*.

CGAL_Planar_map_2<Dcel, Traits>:: Holes_iterator

A bidirectional iterator to traverse all the holes (i.e., inner CCBs) of a face (*Holes_iterator++* is the next hole in the face). Its value type is *Ccb_halfedge_circulator*.

CGAL_Planar_map_2<Dcel, Traits>:: Locate_type

Enumeration of the features of the planar map: VERTEX=1, EDGE, FACE, UNBOUNDED_FACE. This specifies the result of point location and ray shooting operations.

Creation

CGAL_Planar_map_2<Dcel, Traits> pm; construct an empty map - containing one unbounded face corresponding to the whole plane

Operations

Halfedge

pm.insert(Traits::X_curve &cv) insert the curve cv into the map
Precondition: No curve *cv1* of *pm* intersects *cv* in the interiors of *cv*,*cv1*

Halfedge

pm.insert_at_vertices(Traits::X_curve cv, Vertex v1, Vertex v2)

insert *cv* as a new edge between *v1* and *v2*, where *v1* and *v2* are vertices of the map.
Precondition: *v1* and *v2* represent the source and target of *cv* respectively.

Halfedge

pm.insert_from_vertex(Traits::X_curve cv, Vertex v1, bool source)

insert a new edge for which one endpoint, *v1*, is already in the map. If *source* is *true* (resp. *false*) then the source (resp. target) of *cv* is *v1*.
Precondition: the second endpoint of *cv* is not in the map.

Halfedge

pm.insert_in_face_interior(Traits::X_curve cv, Face f)

insert *cv* as a new inner component of *f*.
Precondition: *cv* is contained completely in the interior of *f* (no vertex of *cv* meets any vertex on the map).

Halfedge

pm.locate(Traits::Point p, Locate_type& lt)

return the location in *pm* where *p* lies. If *lt* returns VERTEX then *p* lies on the vertex which is the source of the returned *Halfedge*. If *lt* returns EDGE then *p* lies on the returned *Halfedge*. If *lt* returns FACE then *p* lies on the face which is on the left of the returned *Halfedge*. If *lt* returns UNBOUNDED_FACE then *p* lies on the unbounded face, and the returned *Halfedge* is on a hole in the unbounded face

Halfedge

pm.vertical_ray_shoot(Traits::Point p, Locate_type& lt, bool up_direction)

if *up_direction* is *true* (resp. *false*) return the first edge of *pm* that intersects the upward (resp. downward) vertical ray emanating from *p*. If several edges intersect the vertical ray in the same (end) point *q*, the function returns the first edge encountered when moving clockwise from $\vec{pq}$ around *q*. In that case the value of *lt* will be VERTEX. If the ray doesn't intersect the value of *lt* will be UNBOUNDED_FACE. Otherwise the value of *lt* will be EDGE.
Precondition: *p* lies in the interior of a face of *pm*.

Halfedge

pm.split_edge(Halfedge e, Traits::X_curve& c1, Traits::X_curve& c2)

split the edge e into *e1* and *e2*, and add a vertex in the splitting point. *e1* and *e2* will represent *c1* and *c2* accordingly. *c1* is the left-low part of e and *c2* is the rest. One vertex, two edges and zero faces are added.
Precondition: the curve $c1 \cup c2$ is identical to the curve to the curve represented by e. $c1 \cap c2$ is the splitting point.

Face

pm.unbounded_face() return the unbounded face of the planar-map.

Face_iterator

pm.faces_begin() return the begin-iterator of the faces in pm.

Face_iterator

pm.faces_end() return the past-the-end iterator of the faces in pm.

Halfedge_iterator

pm.halfedges_begin() return the begin-iterator of the half-edges in pm.

Halfedge_iterator

pm.halfedges_end() return the past-the-end iterator of the half-edges in pm.

Vertex_iterator

pm.vertices_begin() return the begin-iterator of the vertices in pm.

Vertex_iterator

pm.vertices_end() return the past-the-end iterator of the vertices in pm.

bool *pm.is_valid()* check the validity of all the features of pm. Check for each vertex v whether all incident half-edges (by performing *get_twin().get_next()*)have v as their origin. Check that each half-edge e hold the right curve – *Traits.curve_source(e.curve())==e.source().point()* and *Traits.curve_target(e.curve())== e.target().point()*. Check that all the edges on the boundary of each face f point to f as their face.

5.2.2 2D Planar Map Vertex (*CGAL_Planar_map_2<Dcel,Traits>::Vertex*)

Definition

An object v of the class *Vertex* is a vertex of a planar-map. It represents a 2D point.

Operations

Traits::Point

	v.point()	return the point of v.
bool	*v.is_incident_edge(Halfedge e)*	return *true* if e incident to v (i.e., if it is the source or the target of v).
bool	*v.is_incident_face(Face f)*	return *true* if f incident to v.
int	*v.degree()*	return the degree of v.

Halfedge_around_vertex_circulator

	v.incident_halfedges()	return a circulator that allows to traverse the halfedges that have v as their source. The edges are traversed around v in a clockwise order. This circulator can also be used to access incident faces and vertices.

5.2.3 2D Planar Map Halfedge (*CGAL_Planar_map_2<Dcel,Traits>::Halfedge*)

Definition

An object e of the class *Halfedge* is a directed edge in a planar map, which represents a half-edge of the Dcel.

Operations

Traits::X_curve&

e.curve()	return the curve that this edge represents.

Vertex

e.source()	return the source vertex of e.

Vertex

e.target()	return the destination vertex of e.

Face

e.face()	return the face that e is incident to, that lies to the left of e.

Halfedge

e.twin()	return the twin half-edge (the same half-edge in the opposite direction).

Halfedge

e.next_halfedge()	return the next half-edge on the CCB of the face that lies to the left of e.

Ccb_halfedge_circulator

e.ccb()	return a circulator to traverse the half-edges of the CCB containing e.

5.2.4 2D Planar Map Face (*CGAL_Planar_map_2<Dcel,Traits>::Face*)

Definition

An object f of the class *Face* is a face in a planar map.

Operations

bool

f.is_unbounded()	return *true* if the face is unbounded

Holes_iterator

f.holes_begin()	return an iterator for traversing all the holes (inner CCBs) of f.

Holes_iterator

f.holes_end()	return the corresponding past-the-end iterator.

Halfedge

f.halfedge_on_outer_ccb()	return a half-edge on the outer boundary of f. *Precondition*: The face has an outer CCB; namely f is a bounded face.

Ccb_halfedge_circulator

f.outer_ccb()	return a circulator for traversing the outer boundary of f. *Precondition*: The face has an outer CCB; namely f is a bounded face.

bool *f.is_halfedge_on_inner_ccb(Halfedge e)*

return *true* if e belongs to an inner CCB of f.

bool *f.is_halfedge_on_outer_ccb(Halfdge e)*

return *true* if e belongs to the outer CCB of f.

bool *f.is_outer_ccb_exist()* return *true* if f has an outer CCB.

5.2.5 (*CGAL_Planar_map_2<Dcel,Traits>::Ccb_halfedge_circulator*)

Definition

A circulator over the half-edges of a connected component of a boundary of a face in the planar map. The circulator conforms to the requirements of forward circulators. The value type is *CGAL_Planar_map_2<Dcel,Traits>::Halfedge*.

5.2.6 (*CGAL_Planar_map_2<Dcel,Traits>::Halfedge_around_vertex_circulator*)

Definition

A circulator over the half-edges with a common origin. The circulator conforms to the requirements of forward circulators. The value type is *CGAL_Planar_map_2<Dcel,Traits>::Halfedge*.

5.3 Predefined Planar Map Interface Classes

The Planar Map Class is parameterized with the interface classes *Dcel* and *Traits*. *Dcel* defines the underlying combinatorical data structure (halfedge structure) used by the planar map. *Traits* defines the abstract interface between planar maps and the *geometric* primitives they use. In the future we plan to supply the users with the formal requirements of the interface class, so they can construct their own class. However, at the moment only the predefined dcel and traits classes given below can be used as parameters.

We supply a default traits class for exact arithmetic — *CGAL_Pm_segment_exact_traits<R>* where R is the representation type. There is also a class for finite precision arithmetic — *CGAL_Pm_segment_epsilon_traits<R>*, which is not recommended unless it is known that robustness issues will not arise. Both traits handle finite line segments in the plane (*X_curve* is of type *CGAL_segment_2<R>* and *Point* is of type *CGAL_Point_2<R>*).

5.3.1 Default Dcel Structure (*CGAL_Pm_default_dcel<Traits>*)

Definition

The class *CGAL_Pm_default_dcel<Traits>* is used as the default (at the moment, the only) class for the underlying halfedge structure of the planar map.

#include < CGAL/Pm_default_dcel.h>

5.3.2 Exact (Rationals) Traits (*CGAL_Pm_segment_exact_traits<R>*)

Definition

The class *CGAL_Pm_segment_exact_traits<R>* is used as the recommended class. To avoid robustness problems, the representation type R must be an exact type such as *CGAL_Cartesian<leda_rational>*, *CGAL_Homogeneous<leda_integer>* or any other exact type.

#include < CGAL/Pm_segment_exact_traits.h>

Types

CGAL_Pm_segment_exact_traits<R>:: Point

of type *CGAL_Point_2<R>*

CGAL_Pm_segment_exact_traits<R>:: X_curve

of type *CGAL_Segment_2<R>*

Creation

CGAL_Pm_segment_exact_traits<R> T_exact;

5.3.3 Floating Point Epsilon Traits (*CGAL_Pm_segment_epsilon_traits<R>*)

Definition

The class *CGAL_Pm_segment_epsilon_traits<R>* is used for finite precision (floating-point) arithmetic. This is a temporary Traits Class. (We plan to implement more sophisticated methods to handle floating point arithmetic at a later stage.) The class uses a predefined ε (of type *double*) to decide whether two values should be considered the same. Two real values a and b are considered equal if $|a - b| < \varepsilon$. Thus when L_∞ norm is used, two points p_1, p_2 are treated as the same point if $\|p_1 - p_2\|_\infty = \max(|p_1.x() - p_2.x()|, |p_1.y() - p_2.y()|) < \varepsilon$. The value of ε can be determined through the class's constructor.

#include < CGAL/Pm_segment_epsilon_traits.h >

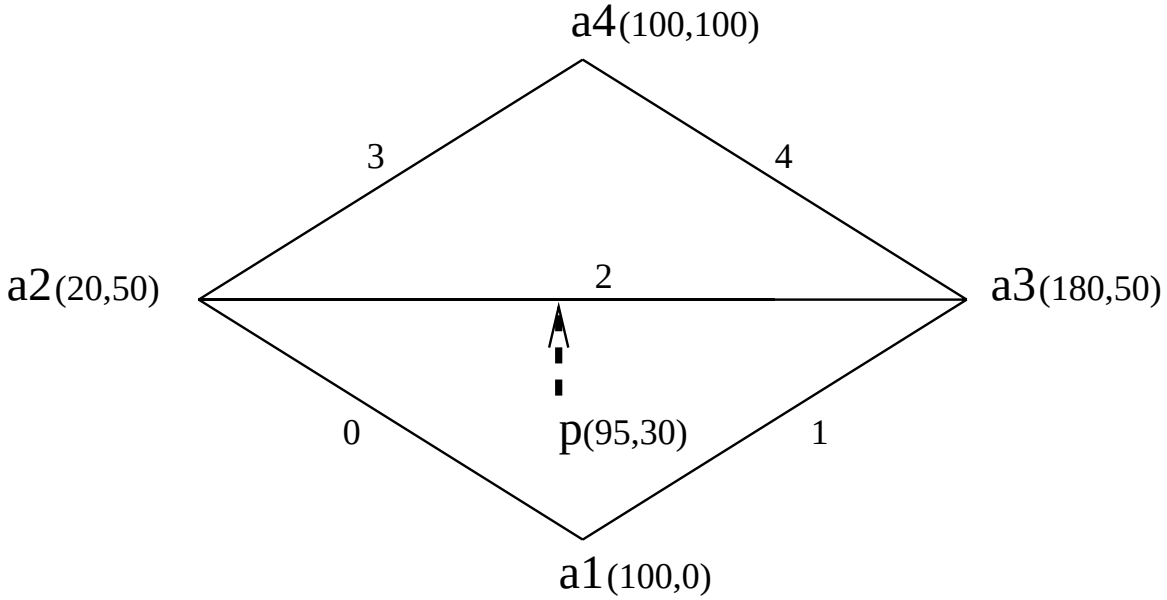


Figure 5.3: The map generated by the example program

Types

CGAL_Pm_segment_epsilon_traits<R>:: Point

of type *CGAL_Point_2<R>*

CGAL_Pm_segment_epsilon_traits<R>:: X_curve

of type *CGAL_Segment_2<R>*

Creation

CGAL_Pm_segment_epsilon_traits<R> T_fp(double eps = 0.0001);

5.4 Example Program

The following program creates a planar map from five curves (see figure 5.3). It uses the floating-point arithmetics interface class (*CGAL_Pm_segment_epsilon_traits<R>*). The first part of the code initializes five segments. The next part inserts them to the map and checks the validity of the map. In the last part, a vertical ray shooting is performed from the point p.

```
#include <CGAL/Cartesian.h>
#include <CGAL/Pm_segment_epsilon_traits.h>
#include <CGAL/Pm_default_dcel.h>
#include <CGAL/Planar_map_2.h>
```

```
typedef double                                          number_type;
typedef CGAL_Cartesian<number_type>                     coord_t;
typedef CGAL_Pm_segment_epsilon_traits<coord_t>         pmtraits;
typedef typename pmtraits::Point                        point;
typedef typename pmtraits::X_curve                      curve;

typedef CGAL_Pm_default_dcel<pmtraits>                  pmdcel;


int main()
{
  // creating an instance of CGAL_Planar_map_2<pmdcel,pmtraits>
  CGAL_Planar_map_2<pmdcel,pmtraits> pm;

  curve cv[5];
  int i;

  point a1(100, 0), a2(20, 50), a3(180, 50), a4(100, 100);

  // those curves are about to be inserted to pm
  cv[0] = curve(a1, a2);
  cv[1] = curve(a1, a3);
  cv[2] = curve(a2, a3);
  cv[3] = curve(a2, a4);
  cv[4] = curve(a3, a4);

  cout << "the curves of the map:" << endl;
  for (i = 0; i < 5; i++)
    cout << cv[i] << endl;

  cout << endl;

  // insert the five curves to the map
  cout << "inserting the curves to the map..." << endl;
  for (i = 0; i < 5; i++)
  {
    cout << "inserting curve" << i << endl;
    pm.insert(cv[i]);
  }

  // check the validity of the map
  cout << "check map validity... ";
  if (pm.is_valid())
    cout << "map valid!" << endl;
  else
    cout << "map invalid!" << endl;
  cout << endl;

  // vertical ray shooting upward from p
  point p(95, 30);
  typename CGAL_Planar_map_2<pmdcel,pmtraits>::Halfedge e;
  typename CGAL_Planar_map_2<pmdcel,pmtraits>::Locate_type lt;

  cout << endl << "upward vertical ray shooting from " << p << endl;
```

```
  e=pm.vertical_ray_shoot(p, lt, true);
  cout << "returned the edge:" << e << endl;

  return 0;
}
```

The output of this program is

```
the curves of the map:
100 0 20 50
100 0 180 50
20 50 180 50
20 50 100 100
180 50 100 100

inserting the curves to the map...
inserting curve0
inserting curve1
inserting curve2
inserting curve3
inserting curve4
check map validity... map valid!

upward vertical ray shooting from 95 30
returned the edge: {(20 50)->(180 50)}
```

Chapter 6

Triangulations

6.1 Introduction

This chapter presents a framework for triangulations that are 2-manifolds without singularities that may be embedded in the 3D space. Examples are the triangulation of a simple polygon in the plane, the Delaunay triangulations of points in the plane, or triangulated irregular networks (TINs) which are used as terrain model in GIS. All these triangulations are based on vertices and triangular faces, and on incidence information between vertices and faces. On top of that *abstract triangulation* come different *geometric layers*: They vary in the geometric information associated to a vertex (e.g., two-dimensional points for triangulations in the plane, or three-dimensional points for TINs), and they vary in the functionality. The insertion of a point into a Delaunay triangulation or a TIN requires different update steps.

Design Rationale

As triangulations are basic data structures they are not implemented as a layer on top of a planar map, but they have a proper internal representation. The elementary objects of the triangulation are vertices and triangular faces with pointers to their three neighbors and their three vertices. This representation needs less storage and should result in faster algorithms. Notice that edges are not represented in triangulation. Therefore, to attach information on edges, one should use an edge-based data structure such as *CGAL_Planar_map_2*.

The abstract triangulation layer is implemented by two classes for the faces and vertices of a triangulation. The geometric layer is implemented by several triangulation classes which allow to interact with a collection of faces and vertices. Triangulations can be traversed with iterators and circulators.

A triangulation may or may not have a boundary. A triangle having an edge on the boundary has no neighbor through this edge. For triangulations in the plane, it is sometimes convenient to artificially close the triangulation by adding an auxiliary point so that it is homeomorphic to a 3D sphere.

Organization of this Chapter

Sections 6.2 to 6.3 introduce triangulation classes which decompose the plane in triangles. Section 6.2 presents a class that maintains a triangulation of the convex hull of points in the plane. Section 6.3

presents a class that maintains the *Delaunay property* for a set of points. Both classes are dynamic, that is points can be inserted in and removed from the triangulation.

The triangulation classes are parameterized with a triangulation traits class, that defines the types of the primitives the triangulation uses. CGAL provides some traits class implementations for the CGAL kernel that are presented in section 6.4.

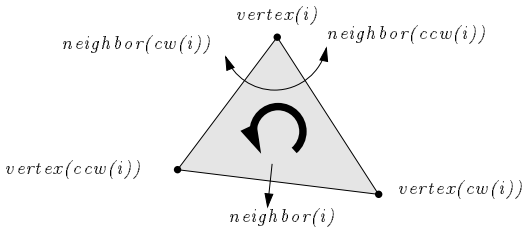
Section 6.5 presents the syntactical and semantical requirements that must be fulfilled by a triangulation traits class. This is at the same time a specification for people that want to use the classes provided by CGAL as for people that want to extend the framework.

Section 6.6 gives the necessary informations for people who want to reuse classes provided by CGAL.

Operations on Indices

The three vertices of a face are indexed with 0, 1 and 2. This order defines a counterclockwise order. Given an index of a vertex the following functions allow to compute the index of the (counter-) clockwise neighbor vertex. Thus, all the classes provide the two functions *int ccw(int i)* and *int cw(int i)*. Note that the functions are static member functions although they do not depend on the actual face.

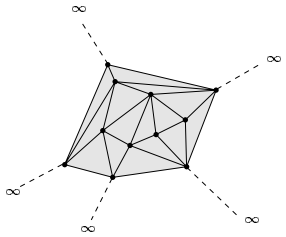
Note further that *f.neighbor(f.ccw(i))* is the neighbor of f, turning counterclockwise around vertex i of f and that *f.vertex(f.ccw(i))* is the vertex of f, turning counterclockwise inside f from vertex i.



6.2 Triangulation of Points in the Plane

The usual definition of the triangulation of a set of points specifies that the interior of the convex hull of the points is divided into non overlapping triangles having the given points as vertices. Thus the faces of the planar subdivision are triangles except the unbounded face which has as many vertices as there are points on the convex hull. However, for many applications, such as Kirkpatrick's hierarchy or incremental Delaunay construction, it is convenient that the unbounded face is also triangulated. In that way, special cases at the boundary of the convex hull are simpler to deal with.

The classes described in the following sections have one auxiliary vertex "at infinity". All vertices on the boundary of the convex hull are incident to this vertex.



Thus the triangulation of the set of points consists of finite and infinite faces. Although it is convenient to draw a triangulation as in the above figure you should not take the coordinates of the vertex at infinity, nor should you apply a geometric predicate on an infinite face.

Definition

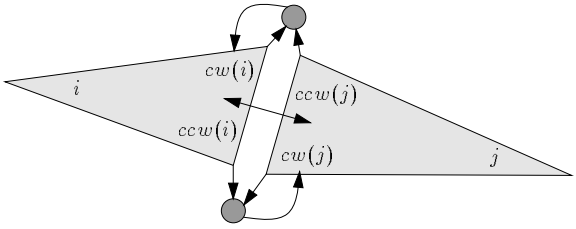
The class *CGAL_Triangulation_2<Traits>* maintains a triangulation of the convex hull of points in the plane.

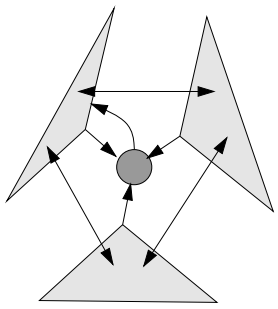
A triangulation is a collection of vertices and faces that are linked together. Each face has pointers to three vertices and to three neighbor faces. Each vertex has a pointer to an adjacent face. The orientation of a face is defined by the order of its vertices, which is referred to as the *counterclockwise order*.

We call a triangulation *valid* if the following is true.

(a) Two faces that are neighbors have pointers to each other and they have two vertices in common. The faces have a coherent orientation, that is, the common vertices have opposite order in the two faces.

(b) All faces that are incident to a vertex v must be linked with neighbor pointers. Vertex v points to an arbitrary incident face.





The template parameter *Traits* is the triangulation traits class. It defines the types of the primitives the triangulation uses. For example *Traits::Point* is a mapping on a point class. Think of it as 2D points in the Euclidean plane.

CGAL provides some traits class implementations for the CGAL kernel which are described in section 6.4. Customizing own triangulation traits classes can be done according to the requirements given in section 6.5.1.

#include <CGAL/Triangulation_2.h>

Types

CGAL_Triangulation_2<Traits>:: Traits

the triangulation traits type.

typedef Traits::Point Point;

typedef Traits::Segment

Segment;

typedef Traits::Triangle

Triangle;

The following types denote the vertex and face type of the triangulation. They have the functionality as specified in sections 6.5.3 and 6.5.4.

typedef Traits::Vertex

Vertex; Vertex type.

typedef Traits::Face Face; Face type.

typedef pair<Face_handle, int>

Edge; Edge type. An *Edge(f,i)* is edge number i of face f.

The following types are handles on *Vertex* and *Face*. A handle can be dereferenced with *operator** or *operator->* to get an object, see the chapter on handles.

typedef Vertex::Vertex_handle

Vertex_handle;

typedef Face::Face_handle

Face_handle;

The following types denote iterators that allow to visit all finite vertices, edges and faces of the triangulation.

CGAL_Triangulation_2<Traits>:: Vertex_iterator

Bidirectional, non-mutable, with value-type *Vertex*, inherits from *Vertex_handle*.

CGAL_Triangulation_2<Traits>:: Edge_iterator

Bidirectional, non-mutable, with value-type *Edge*.

CGAL_Triangulation_2<Traits>:: Face_iterator

Bidirectional, non-mutable, with value-type *Face*, inherits from *Face_handle*.

CGAL_Triangulation_2<Traits>:: Line_face_circulator

Bidirectional, non-mutable, with value-type *Face*, inherits from *Face_handle*.

The following types denote circulators that allow to visit vertices, edges and faces incident to a given vertex.

typedef Vertex::Vertex_circulator

Vertex_circulator;

Bidirectional, non-mutable, with value-type *Vertex*, inherits from *Vertex_handle*.

typedef Vertex::Edge_circulator

Edge_circulator;

Bidirectional, non-mutable, with value-type *Edge*.

typedef Vertex::Face_circulator

Face_circulator;

Bidirectional, non-mutable, with value-type *Face*, inherits from *Face_handle*.

enum Locate_type { VERTEX=0, EDGE, FACE, OUTSIDE, COLLINEAR_OUTSIDE};

Specifies which case occurs during a point location operation. *OUTSIDE* means outside the convex hull. *COLLINEAR_OUTSIDE* means outside the convex hull but collinear to all points of the triangulation, that is the convex hull is 1-dimensional.

Creation

CGAL_Triangulation_2<Traits> *T(Traits t = Traits());*

Introduces an empty triangulation T.

CGAL_Triangulation_2<Traits> *T(Vertex_handle v, Traits t = Traits());*

Introduces a triangulation T that is initialized with the vertices and faces that are linked to vertex v. If v has no incident face the triangulation consists only of v. Otherwise v must be the vertex at infinity.

CGAL_Triangulation_2<Traits> *T(T0);*

Copy constructor. The triangulation is duplicated, $T$and *T0* refer to different triangulations. After the copy, if *T0* is modified, $T$is not.

CGAL_Triangulation_2<Traits>

T = T0 Assignment. The triangulation is duplicated, and modifying one after the copy does not modify the other.

void *T.swap(& T0)*

The triangulations *T0* and $T$are swapped. *T.swap(T0)* should be preferred to *T=T0* or to *T(T0)* if *T0* is deleted after that.

Setting

void *T.set_finite_vertex(Vertex_handle v)*

void *T.set_infinite_vertex(Vertex_handle v)*

Point Location

Face_handle *T.locate(Point query, Face_handle f = Face_handle())*

If the point lies inside the convex hull of the points, a finite face that contains the query in its interior or on its boundary is returned.
If the point *query* does not lie inside the convex hull of the points, an infinite face with vertices (∞, p, q) is returned such that the following holds: Let P be the current point set of the triangulation plus the query point. If the dimension of the convex hull of P is 2, the point *query* lies to the left of the oriented line passing through p and$q$. If the dimension of the convex hull of P is 1, q lies between p and *query*, and there is no other vertex on the convex hull that lies between them.
The last argument f is an indication to the *locate* step where to start.
Precondition: T has at least two vertices.

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 *li* is the index of the vertex or the index of the vertex opposite to the edge, if the point lies on a vertex or on an edge. Otherwise *li* has no meaning.
Precondition: T has at least two vertices.

Insertion and Removal

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.
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 f =Face_handle())*

Same as above. Additionally, parameter *lt* describes where point p was located before updating the triangulation (see *locate*).

Vertex_handle *T.push_back(Point p)*

Equivalent to *insert(p)*.

template < *class InputIterator* >
int *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 *first* and *last* is *Point*.

void *T.remove(Vertex_handle v)*

Removes the vertex from the triangulation. The created hole is retriangulated.
Precondition: Vertex v must be finite.

void *T.clear()*

Removes all finite vertices and faces.

void *T.flip(Face_handle f, int i)*

Flips face f with its neighbor i.

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).

Vertex Iterator

The triangulation defines an iterator that allows to visit all finite vertices. This iterator is non-mutable and bidirectional. Its value type is *Vertex*.

Vertex_iterator *T.vertices_begin()*

Returns an iterator that refers to the "first" finite vertex of T.

Vertex_iterator *T.vertices_end()*

Returns the corresponding past-the-end iterator.

A vertex iterator is invalidated by any modification of the triangulation.

Edge Iterator

The triangulation class defines an iterator that allows to visit all edges with finite vertices. This iterator is non-mutable and bidirectional. Its value type is *Edge*.

Edge_iterator *T.edges_begin()*

Returns an iterator that refers to the "first" finite edge of T.

Edge_iterator *T.edges_end()*

Returns the corresponding past-the-end iterator.

An edge iterator is invalidated by any modification of the triangulation.

Face Iterator

The triangulation class defines an iterator that allows to visit all finite faces. This iterator is non-mutable and bidirectional. Its value type is *Face*.

Face_iterator *T.faces_begin()*

Returns an iterator that refers to the "first" finite face of T.

Face_iterator *T.faces_end()*

Returns the corresponding past-the-end iterator.

A face iterator is invalidated by any modification of the triangulation.

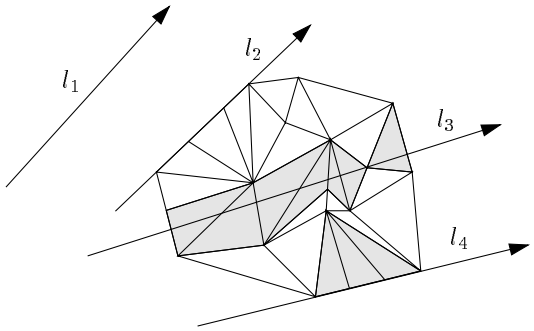
Line Face Circulator

The triangulation defines a circulator that allows to visit all faces that are intersected by a line. 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())*

Let l be the line defined by p and q. This function returns a circulator that allows to traverse the finite faces whose interior intersects with l, or that have an edge that is collinear with l and whose interior lies completely to the left of l.
The starting point of the circulator is the face f, if f != *NULL*, or the first such face along l. If there is no such face the circulator has a singular value.
The circulator performs a wraparound and refers to the infinite faces adjacent to the first and last finite face that is traversed.
Precondition: If f != *NULL*, the point p must be inside or on the boundary of f.

The following figure illustrates which 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 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.

Vertex Circulator

The triangulation defines a circulator that allows to visit all vertices adjacent to a vertex *Vertex_handle* v. This circulator is non-mutable and bidirectional, where *operator++* moves the circulator counterclockwise. Its value type is *Vertex_handle*. The

Vertex_circulator *v.incident_vertices()*

Returns a vertex circulator that refers to an arbitrary vertex incident to vertex v.

A vertex circulator *circ* is invalidated by any modification of a face incident to v and **circ*.

Edge Circulator

The triangulation class defines a circulator that allows to visit all edges adjacent to a vertex *Vertex_handle* v. This circulator is non-mutable and bidirectional, where *operator++* moves the circulator counterclockwise. Its value type is *Edge*.

Edge_circulator *v.incident_edges()*

Returns an edge circulator that refers to an arbitrary edge incident to vertex v.

Edge_circulator *v.incident_edges(Face_handle f)*

Returns an edge circulator that refers to the edge incident to vertex v, going counterclockwise in f from v.

An edge circulator *circ* is invalidated by any modification of a face incident to the edge it refers to.

Face Circulator

The triangulation class defines a circulator that allows to visit all faces adjacent to a vertex *Vertex_handle v*. This circulator is non-mutable and bidirectional, where *operator++* moves the circulator counterclockwise. Its value type is *Face_handle*.

Face_circulator *v.incident_faces()*

Returns a face circulator that refers to an arbitrary face incident to vertex v.

Face_circulator *v.incident_faces(Face_handle f)*

Returns a face circulator that refers to face f.

A face circulator is invalidated by any modification of the face it refers to.

Traversal of the Convex Hull

The above three functions applied on the infinite vertex of the triangulation allow to visit the vertices on the convex hull, as well as 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 T.infinite_vertex()->incident_vertices()

Finite and Infinite Vertices and Faces

bool *T.is_infinite(Vertex_handle v)*

Returns *true*, iff vertex v is infinite.

bool *T.is_infinite(Face_handle f)*

Returns *true*, iff one of the vertices of f is infinite.

bool *T.is_infinite(Edge e)*

Returns *true*, iff edge e is infinite.

bool *T.is_infinite(Edge_circulator e)*

Returns *true*, iff edge e is infinite.

bool *T.is_infinite(Edge_iterator e)*

Returns *true*, iff edge e is infinite.

bool *T.is_infinite(Face_handle f, int i)*

Returns *true*, iff the edge of face f with index i is infinite.

Face_handle *T.infinite_face()*

Returns an arbitrary face that has an infinite vertex.

Vertex_handle *T.infinite_vertex()*

Returns the infinite vertex.

Vertex_handle *T.finite_vertex()*

Returns an arbitrary finite vertex.

Geometric Predicates

As the faces are oriented counterclockwise, we can identify the "interior" of a finite face with the positive side.

CGAL_Oriented_side *T.oriented_side(Face_handle f, Point p)*

Returns on which side of the oriented boundary of f lies the point p.
Precondition: f is finite.

Miscellaneous

int *T.ccw(int i)*

Returns $i + 1$ modulo 3.
Precondition: $0 \leq i \leq 2$.

int *T.cw(int i)*

Returns $i + 2$ modulo 3.
Precondition: $0 \leq i \leq 2$.

Traits *T.traits()*

Returns a const reference to the triangulation traits object.

int *T.dimension()*

Returns the dimension of the convex hull.

int *T.number_of_vertices()*

Returns the number of finite vertices.

int *T.number_of_faces()*

Returns the number of finite and infinite faces.

Triangle *T.triangle(Face_handle f)*

Returns the triangle formed by the three vertices of f.
Precondition: The face is finite.

Segment *T.segment(Face_handle f, int i)*

Returns the line segment formed by the vertices *f->ccw(i)* and *f->cw(i)* of face f.
Precondition: $0 \leq i \leq 2$. *f->ccw(i)* and *f->cw(i)* are finite.

Segment *T.segment(Edge e)*

Returns the line segment corresponding to edge e.

Segment *T.segment(Edge_circulator ec)*

Returns the line segment corresponding to edge **ec*.

Segment *T.segment(Edge_iterator ei)*

Returns the line segment corresponding to edge **ei*.

bool *T.is_valid(bool verbose = false, int level = 0)*

Tests the validity of all faces, if the orientation of the faces is consistent. This method serves mainly for debugging user-defined triangulation.

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/ostream_2.h>

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*.

#include <CGAL/IO/Window_stream.h>

CGAL_Window_stream&

CGAL_Window_stream& W << T

Inserts the triangulation T into the window stream W. The insert operator must be defined for *Point* and *Segment*.

Example

The following code fragment creates a triangulation of 2D points for the usual Euclidean metric. We read points from `cin` and writes the points on the convex hull to `cout`. The `while` loop can be replaced by the code in the comment, as the triangulation class has an *insert* member function that takes an STL range of points as argument.

```
typedef CGAL_Cartesian<CGAL_Rational> Rep;
typedef CGAL_Point_2<Rep> Point;
typedef CGAL_Triangulation_euclidean_traits_2<Rep> Traits;

typedef CGAL_Triangulation_2<Traits> Triangulation;
typedef Triangulation::Vertex_circulator Vertex_circulator;

{
    Triangulation T();

    while (cin){
        Point p;
        cin ≫ p;
        T.insert(p);
    }

    Vertex_circulator hvc = T.infinite_vertex()→incident_vertices(),
                      done(hvc);

    if (hvc ≠ NULL) {
        do{
            cout ≪ hvc→point();
        }while(++hvc ≠ done);
    }
}
```

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 while maintaining the convex hull if the point falls outside. Apart from the location, insertion takes time $O(1)$.

Removal of a vertex is done by removing all adjacent triangles, and retriangulating 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 are incremented by following a geometric traversal of the triangulation as defined in [dBvOO96]. The iterators take space $O(1)$. Incrementing them takes amortized time $O(1)$.

6.3 Delaunay Triangulation of Points

Definition

The class *CGAL_Delaunay_triangulation_2<Traits>* is a triangulation of points in a plane that maintains the *Delaunay property*: The circumscribing circle of each face does not contain any vertex.

The template parameter *Traits* is the Delaunay triangulation traits class. It defines the primitives the Delaunay triangulation uses.

CGAL provides some traits class implementations for the CGAL kernel that are described in section 6.4. Customizing own triangulation traits classes can be done according to the requirements given in section 6.5.1.

#include <CGAL/Delaunay_triangulation.h>

Inherits From

CGAL_Triangulation_2<Traits>

The modifying functions *insert* and *remove* overwrite the inherited functions to maintain the Delaunay property.

Types

typedef Traits::Distance

Distance;

Creation

CGAL_Delaunay_triangulation_2<Traits> DT(Traits t = Traits());

Introduces an empty Delaunay triangulation DT.

CGAL_Delaunay_triangulation_2<Traits> DT(Vertex_handle v; Traits t = Traits());

Introduces a Delaunay triangulation DT that is initialized with the vertices and faces that are linked to vertex v. If v has no incident face the triangulation consists only of v. Otherwise v must be the vertex at infinity.

Insertion and Removal

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 initialized the location.

Vertex_handle *DT.insert(Point p, Locate_type& lt. Face_handle f=Face_handle())*

Same as above. Additionally, parameter *lt* describes where point p was located before updating the triangulation.

Vertex_handle *DT.push_back(Point p)*

Equivalent to *insert(p)*.

template < class InputIterator >
int *DT.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 *DT.remove(Vertex_handle v)*

Removes the vertex from the triangulation.

Queries

Vertex_handle *DT.nearest_vertex(Point p, Face_handle f=Face_handle())*

Returns any nearest vertex of p. f may be used to initialize the location

Geometric Predicates

CGAL_Oriented_side *DT.side_of_oriented_circle(Face_handle f, Point p)*

Returns the side of p with respect to the circle circumscribing the triangle associated with f

Miscellaneous

bool *DT.is_valid(bool verbose = false, int level = 0)*

Tests the validity of all faces and verifies that the Delaunay property and the orientation of the triangles. This method serves mainly for debugging Delaunay triangulation algorithms designed by the user.

Example

The following code fragment 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 and the predicates which are defined in the Delaunay triangulation traits class forget about the z-coordinate of these points. The terrain model is then visualized on a 3D viewer.

```
typedef CGAL_Homogeneous<leda_integer>  Rep;
typedef CGAL_Triangulation_euclidean_xy_traits_3<Rep>  Terrain;
typedef CGAL_Delaunay_triangulation_2<Terrain> Delaunay;

{
    Delaunay DT(Terrain());

    while(cin){
        CGAL_Point_3<Rep> p;
        cin >> p;
        DT.insert(p);
    }
    CGAL_Geomview_stream G;
    G << DT;
}
```

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 retriangulates the hole but this time using the Delaunay criterion. Removal of a vertex of degree d takes time $O(d^2)$, which is $O(1)$ for a random vertex in the triangulation.

A nearest neighbor is found in time $O(n)$ in the worst case, but $O(1)$ for vertices distributed uniformly at random, and any query point.

6.4 Predefined Triangulation Traits Classes

The classes *CGAL_Triangulation_2<Traits>* and *CGAL_Delaunay_triangulation_2<Traits>* are parameterized with a triangulation traits class. CGAL provides some traits class implementations for the CGAL kernel. These implementations allow to vary the metric used in the triangulations and to operate on 2d-projections of three-dimensional points.

6.4.1 Euclidean Metric

Definition

The class *CGAL_Triangulation_euclidean_traits_2<R>* is used for the usual Euclidean metric for 2D points.

#include <CGAL/Triangulation_euclidean_traits_2.h>

Types

typedef CGAL_Point_2<R> Point;

typedef CGAL_Segment_2<R> Segment;

typedef CGAL_Triangle_2<R> Triangle;

CGAL_Triangulation_euclidean_traits_2<R>:: Vertex

CGAL_Triangulation_euclidean_traits_2<R>:: Face

Creation

CGAL_Triangulation_euclidean_traits_2<R> E;

6.4.2 Euclidean Metric in a Projection Plane

Definition

The class *CGAL_Triangulation_euclidean_xy_traits_3<R>* is used for the usual Euclidean metric applied on the projection of 3D points on the the xy-plane. Instead of really projecting the 3D points and maintaining a map between them (which costs space and is error prone) this class supplies geometric predicates that ignore the z-coordinate of the points.

#include <CGAL/Triangulation_euclidean_xy_traits_3.h>

Types

typedef CGAL_Point_3<R> Point;

typedef CGAL_Segment_3<R> Segment;

typedef CGAL_Triangle_3<R> Triangle;

CGAL_Triangulation_euclidean_xy_traits_3<R>:: Vertex

CGAL_Triangulation_euclidean_xy_traits_3<R>:: Face

Creation

CGAL_Triangulation_euclidean_xy_traits_3<R> E;

Variants

Instead of projecting on the xy-plane, the classes *CGAL_Triangulation_euclidean_xz_traits_3<R>* and *CGAL_Triangulation_euclidean_yz_traits_3<R>* project on the xz- and the yz-plane, respectively.

#include <CGAL/Triangulation_euclidean_xz_traits_3.h>
#include <CGAL/Triangulation_euclidean_yz_traits_3.h>

6.5 Requirements

6.5.1 Requirements for Triangulation Traits Classes

The classes *CGAL_Triangulation_2<Traits>* and *CGAL_Delaunay_triangulation_2<Traits>* are parameterized with a triangulation traits class, that defines the types of the primitives the triangulation uses. The following requirement catalog lists the primitives, i.e. types, member functions etc., that must be defined for a class that can be used to parameterize triangulations. The traits class implementations for the CGAL kernel from section 6.4 can be used as a starting point for customizing own triangulation traits classes, for example through derivation and specialization.

Requirements for a Triangulation Traits (*Traits*)

Definition

A class *Traits* that satisfies the requirements for a triangulation traits class must provide the following types and operations.

Be careful that the constructed triangulation must be a triangulation of the convex hull (convex hull edges, must be Delaunay edges). Otherwise problems may occur. This property is verified for any smooth convex metric. It can also be ensured for other metrics by the insertion of the vertices of a bounding box enclosing the sites.

Types

Traits:: Vertex	A type to hold a vertex in a triangulation. The requirements for a triangulation vertex from section 6.5.3 must be fulfilled.
Traits:: Face	A type to hold a face in a triangulation. The requirements for a triangulation face from section 6.5.4 must be fulfilled.
Traits:: Point	The type must provide a copy constructor, assignment and equality test.
Traits:: Segment	The type must provide a constructor that takes two points as argument.
Traits:: Triangle	The type must provide a constructor that takes three points as argument.
Traits:: Distance	Only needed for *CGAL_Delaunay_triangulation_2<Traits>:: nearest_vertex(..)*. The requirements for a Distance type must be fulfilled, see section 6.5.2 below.

Creation

Only a default constructor is required. Note that further constructors can be provided.

Traits t;	A default constructor.

Operations

CGAL_Comparison_result

t.compare_x(Point p0, Point p1)

Compares the x-coordinates.

CGAL_Comparison_result

t.compare_y(Point p0, Point p1)

Compares the y-coordinates.

CGAL_Orientation *t.orientation(Point p0, Point p1, Point p2)*

Performs the orientation test.

CGAL_Oriented_side *t.side_of_oriented_circle(Point p0, Point p1, Point p2, Point test)*

Performs the incircle test. Points *p0, p1* and *p2* must be oriented counterclockwise and the method tests point *test* with respect to the circumscribing circle.

6.5.2 Requirement for Distances between Points

Definition

This class stores up to three points *p0*, *p1* and *p2*. It can tell which of *p1* and *p2* is closer to *p0*. The three points can be replaced dynamically. Note that "closer" does not necessarily mean the Euclidean metric. As it may depend on an instance of *Traits*, such an instance can be passed as an argument to the constructors.

Types

Distance:: Point

Distance:: Traits A type that satisfies the requirements for a triangulation traits class.

Creation

Distance d(Point p0=Point(), Point p1=Point(), Point p2=Point(), Traits i = NULL);*

Creates a distance object. Missing points are not initialized.

Operations

void *d.set_point(int i, Point pi)*

Replaces point i.
Precondition: $0 \leq i \leq 2$.

Point *d.get_point(int i)*

Returns point i.
Precondition: $0 \leq i \leq 2$.

CGAL_Comparison_result

d.compare()

Compares the distance between *p0* and *p1* with the distance between *p0* and and *p2*.
Precondition: *p1* and *p2* are initialized.

6.5.3 Requirements for a Triangulation Vertex

Definition

A class must fulfill the following requirements in order that it can be used as a vertex of a triangulation.

The type *Point* should contain some geometric information such as coordinates or other, and is typically a point (for example *Point* could be *CGAL_Point_2<R>* for a triangulation of a planar point set, or *CGAL_Point_3<R>* for a terrain model). A vertex provides access to a face of the triangulation having v as vertex.

Types

Vertex:: Point	The type of the geometric information associated to a vertex.
Vertex:: Vertex_handle	Handle to vertex type.
Vertex:: Face_handle	Handle to face type.
Vertex:: Vertex_circulator	A circulator for traversal of adjacent vertices.
Vertex:: Edge_circulator	A circulator for traversal of adjacent edges.
Vertex:: Face_circulator	A circulator for traversal of adjacent faces.

Creation

For triangulation algorithms designed by the user, vertices need to be explicitly constructed. If the modification of the triangulation is done through a triangulation class, there is no need for these functions.

Vertex v; — Introduces a variable v. The geometric information is initialized by the default constructor of class P. The pointer to the incident face is initialized with $NULL$.

Vertex v(Point p); — Introduces a variable v, and initializes the geometric information. The pointer to the incident face is initialized with $NULL$.

Vertex v(Point p, Face_handle f);

Introduces a variable v, and initializes the geometric information and the pointer to the incident face.

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.

Access Functions

Point *v.point()*

Returns the geometric information of v.

Face_handle *v.face()* — Returns a face of the triangulation having v as vertex. The face is the first adjacent face in counterclockwise order, if v is on the boundary.

Miscellaneous

int *v.ccw(int i)*

Returns $i+1$ modulo 3.
Precondition: $0 \leq i \leq 2$.

int *v.cw(int i)*

Returns $i+2$ modulo 3.
Precondition: $0 \leq i \leq 2$.

int *v.degree()*

Returns the degree of v in the triangulation.

Vertex_handle *v.handle()*

Returns a handle to the vertex.

Traversal of the Adjacent Vertices, Edges and Faces.

Three circulator classes allow to traverse the vertices, edges, and faces incident to a given vertex. The *operator++* moves the circulator counterclockwise and *operator--* moves the circulator clockwise.

Vertex_circulator *v.incident_vertices()*

Returns a vertex circulator that refers to an arbitrary vertex incident to vertex v.

Edge_circulator *v.incident_edges(Face_handle f=v.face())*

Returns an edge circulator that refers to an arbitrary edge incident to vertex v if f is omitted. Otherwise the circulator reference edge incident to vertex v, going counterclockwise in f from v.

Face_circulator *v.incident_faces(Face_handle f=v.face())*

Returns a face circulator that refers to face f or to an arbitrary face incident to v.

A face circulator is invalidated by any modification of the face it points 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 the two faces that are incident to v and w.

6.5.4 Requirements for a Triangulation Face

Definition

A class must fulfill the following requirements in order that it can be used as a face of a triangulation.

A face contains pointers to three vertices whose order defines the counterclockwise order. It further contains three pointers to neighbor faces.

The vertices and neighbors are indexed 0,1, and 2. Neighbor i lies opposite to vertex i.

Types

Face:: Vertex_handle — Handle to vertex type.

Face:: Face_handle — Handle to face type.

Creation

For user defined triangulation algorithms, faces need to be explicitly constructed and linked to their neighbors.

Face f; — Introduces a variable f and initializes all vertices and neighbors with *NULL*.

Face f(Vertex_handle v0, Vertex_handle v1, Vertex_handle v2);

Introduces a variable f, and initializes the vertices. The neighbors are initialized with *NULL*.

```
Face f( Vertex_handle v0,
        Vertex_handle v1,
        Vertex_handle v2,
        Face_handle n0,
        Face_handle n1,
        Face_handle n2)
```

Introduces a variable f, and initializes the vertices and the neighbors.

Setting

void *f.set_vertex(int i, Vertex_handle v)*

Set vertex i to be v.
Precondition: $0 \leq i \leq 2$.

void *f.set_neighbor(int i, Face_handle n)*

Set neighbor i to be n.
Precondition: $0 \leq i \leq 2$.

Vertex Access Functions

Vertex_handle *f.vertex(int i)*

Returns the vertex i of f.
Precondition: $0 \leq i \leq 2$.

int *f.index(Vertex_handle v)*

Returns the index of vertex v in f.
Precondition: v is a vertex of f

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)*

Returns *true* if v is a vertex of f, and computes the index i of the vertex.

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)*

Returns the neighbor i of f. The result can be *NULL*.
Precondition: $0 \leq i \leq 2$.

int *f.index(Face_handle n)*

Returns the index of face n.
Precondition: n is a neighbor of f.

bool *f.has_neighbor(Face_handle n)*

Returns *true* if n is a neighbor of f.

bool *f.has_neighbor(Face_handle n, int& i)*

Returns *true* if n is a neighbor of f, and compute the index i of the neighbor.

Miscellaneous

int *f.ccw(int i)*

Returns $i+1$ modulo 3.
Precondition: $0 \leq i \leq 2$.

int *f.cw(int i)*

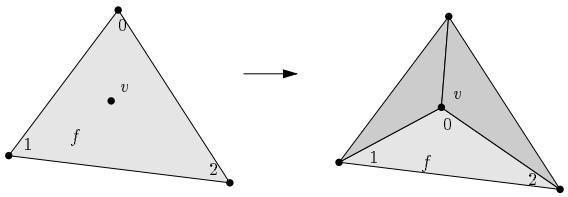
Returns $i+2$ modulo 3.
Precondition: $0 \leq i \leq 2$.

Face_handle *f.handle()*

Returns a handle to the face.

void *f.insert(Vertex_handle v)*

Splits f in three faces, by modifying f and by creating two new faces. Vertex v becomes vertex 0 of f.



It is in your responsibility to keep a triangulation valid, if you use the advanced operations. Obviously you have to make a triangulation invalid at times in order to implement higher level algorithms. To help you in the development of new triangulation classes, CGAL provides a predicate to verify the validity of the constructed structure. This function serves mainly for debugging.

bool *f.is_valid()*

Returns *true* if the triangle shares two vertices with its neighbors, and these neighbors have a correct reciprocal neighboring link and coherent orientations. It further checks if f is in the adjacency list of each of its three vertices, and if the vertices point to the right face if they are on the boundary.

6.6 Implementations

This section gives further information about classes that can be used as starting point if you want to write your own vertex or face classes.

6.6.1 Vertex (*CGAL_Triangulation_vertex<P>*)

Definition

The class *CGAL_Triangulation_vertex* satisfies the requirements for vertices from section 6.5.3.

#include <CGAL/Triangulation_2_vertex.h>

Types

typedef P Point;

typedef CGAL_Triangulation_vertex<P>

Vertex;

typedef CGAL_Pointer<Vertex>

Vertex_handle;

typedef CGAL_Triangulation_face<Vertex>::Face_handle

Face_handle;

6.6.2 Face (*CGAL_Triangulation_face<V>*)

Definition

The class *CGAL_Triangulation_face* satisfies the requirements for faces from section 6.5.4. The type V must fulfill the requirements of a vertex class.

#include <CGAL/Triangulation_face.h>

Types

typedef CGAL_Triangulation_face<V>

Face;

typedef CGAL_Pointer<Face>

Face_handle;

typedef V::Vertex_handle

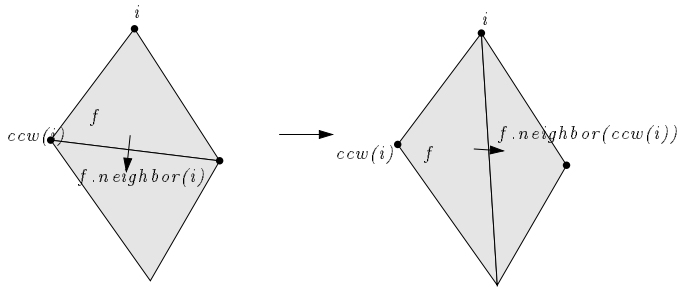
Vertex_handle;

Additional Operations

The following operations allow to modify the abstract triangulation while keeping a valid triangulation, that is the neighbor pointers and the face pointer of the vertices are updated as well. Of course these modifications do not necessarily preserve geometric properties of the triangulation.

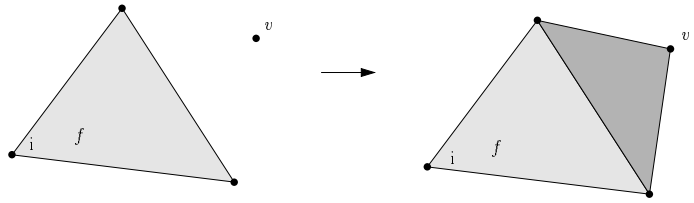
void *f.flip(int i)*

Flips f with $f.n = neighbor(i)$. Let j be *n->index(f)* before the flip. The vertices i and *ccw(i)* of face f do not change. The same holds for the vertices j and *ccw(j)* of n.
Precondition: $0 \leq i \leq 2$, and *f.neighbor(i)* exists.



void *f.insert(Vertex_handle v, int i)*

Adds a triangulation face to the triangulation, that becomes neighbor i of f.
Precondition: f has no neighbor i.



The following vertex removal function is the inverse operation of both insertion functions.

bool *f.remove(Vertex_handle & v)*

Removes v from the triangulation, if one of the two conditions is fulfilled.
Precondition: Node v is not on the boundary and has degree 3 or node v is on the boundary and has degree 2 and f has a neighbor face.
In the first case three triangles are merged to one triangle. The two triangles that are adjacent to f and v are deleted.
In the second case vertex v and face f are deleted. The function returns *true* and sets v to *NULL*, if the removal was successful.

void *f.set_vertices()*

Set vertices to *NULL*

void *f.set_neighbors()*

Set neighbors to *NULL*

void *f.set_vertices(Vertex_handle v0, Vertex_handle v1, Vertex_handle v2)*

void *f.set_neighbors(Face_handle n0, Face_handle n1, Face_handle n2)*

6.6.3 Vertex Circulator (*CGAL_Triangulation_vertex_circulator<V,F>*)

Definition

The types V and F must fulfill the requirements for a vertex and face class, respectively.

#include <CGAL/Triangulation_vertex_circulator.h>

Inherits From

CGAL_Bidirectional_circulator_base<,ptrdiff_t>* and *V::Vertex_handle*.

6.6.4 Edge Circulator (*CGAL_Triangulation_edge_circulator<V,F>*)

typedef pair<Face_handle, int>

Edge;

Definition

The types V and F must fulfill the requirements for a vertex and face class, respectively.

#include <CGAL/Triangulation_edge_circulator.h>

Inherits From

CGAL_Bidirectional_circulator_base<pair<Face_handle, int>,ptrdiff_t>

6.6.5 Face Circulator (*CGAL_Triangulation_face_circulator<V,F>*)

Definition

The types V and F must fulfill the requirements for a vertex and face class, respectively.

#include <CGAL/Triangulation_face_circulator.h>

Inherits From

CGAL_Bidirectional_circulator_base<Face_handle,ptrdiff_t> and *F::Face_handle*.

6.6.6 Vertex Iterator (*CGAL_Triangulation_2_vertex_iterator<Traits>*)

Definition

The type I must fulfill the requirements for a triangulation traits class.

#include <CGAL/Triangulation_2_vertex_iterator.h>

Inherits From

bidirectional_iterator<Traits::Vertex_handle,ptrdiff_t> and *I::Vertex_handle*.

6.6.7 Edge Iterator (*CGAL_Triangulation_2_edge_iterator<Traits>*)

Definition

The type *Traits* must fulfill the requirements for a triangulation traits class.

typedef pair<Face_handle, int>

Edge;

#include <CGAL/Triangulation_2_edge_iterator.h>

Inherits From

bidirectional_iterator<*pair*<*Face_handle, int*>,*ptrdiff_t*>

6.6.8 Face Iterator (*CGAL_Triangulation_2_face_iterator*<*Traits*>)

Definition

The type *Traits* must fulfill the requirements for a triangulation traits class.

#*include* <*CGAL/Triangulation_2_face_iterator.h*>

Inherits From

bidirectional_iterator<*Traits::Face_handle,ptrdiff_t*> and *Traits::Face_handle*.

Chapter 7

Boolean Operations in 2D

7.1 Introduction

CGAL provides boolean operations for polytopes in the 2-dimensional Euclidean space. The functions described in the next section apply to two simple polygons. Both boundary and interior are considered as part of a polygon. Boolean operations on polygons are therefore not limited to operations on the boundary of the objects. The functions described in this chapter allow to perform the purely geometric operations intersection, union, and difference on two polygons in the 2-dimensional Euclidean space. They should not be confused with the regularised operations in the context of solid modeling.

A clear distinction should be made between two sorts of functions described in this chapter. Some operations perform an intersection test without computing the actual result of the intersection. Other operations perform a boolean operation (intersection, union, or difference): they explicitly compute and return the result of the boolean operation on two polygons.

Note that particular classes of polytopes are not closed under boolean operations. For instance, the union of two simple polygons is not necessarily a simple polygon, or the intersection of two triangles is not necessarily a triangle.

All functions described below are template functions. Some are parameterized only by the number type (denoted by *R*), others are parameterized with a traits class (denoted by *Traits*). Where we use polygons of type *CGAL_Polygon_2* the functions are parameterized by a number type (denoted by *R*) and container (denoted by *Container*), as it is the case for *CGAL_Polygon_2*. For more details we refer to the description of *CGAL_Polygon_2*. In a boolean operations traits class some types and functions needed for the computation of boolean operations are defined. We provide a default version of the traits class for boolean operations, which we describe in detail below. So the user need not (but can if desired) provide ones own boolean operations traits class.

Note: The current version of boolean operations is robust only when exact arithmetic, for instance *CGAL_Cartesian<CGAL_Rational>*, is used. In nearly degenerate configurations, correct results are not guaranteed when floating point numbers are used.

7.2 Boolean Operations on Polygons

Definition

Boolean operations are provided for two simple polygons (for the definition of simple polygons, see Chapter 2.1). A triangle and an iso-oriented rectangle clearly are special cases of simple polygons.

A polygon on which the boolean operations can be performed can be stored in one of the following CGAL-objects:

- *CGAL_Triangle_2*, a 2-dimensional triangle.
- *CGAL_Iso_rectangle_2*, a 2-dimensional iso-oriented rectangle.
- *CGAL_Polygon_2*, a 2-dimensional polygon. It must be simple, but not necessarily convex. So, non-adjacent edges do not intersect and there are no holes in the polygon. The vertices of such a polygon must be ordered in counterclockwise order.

We consider polygons as being closed and filled objects, i.e. we consider both its boundary and its interior. The edge cycle of the polygon will be referred to explicitly as the polygon boundary. For boolean operations the distinction between the interior and the exterior of a polygon is determined by the order of its vertices.

The result of an intersection test is one of the boolean values *true* or *false*. The result of an intersection, union, or difference can be empty, a single object, or several objects. An object can be a point, a segment, a triangle, an iso-oriented rectangle, a polygon, or a polygon with one or several holes.

There, where the result cannot be more than one object (in the case of the intersection of two triangles, and in the case of the intersection of two iso-rectangles) the corresponding function returns a possibly empty object. In all other cases the functions return a possibly empty sequence of objects.

Note that whenever we refer to a "sequence" of objects we actually mean a collection of objects independent of the way in which they are stored. We provide a mechanism which allows the user to define the type of "sequence" in which the output will be stored (the template *OutputIterator*, see further for details).

Vertices of input polygons must be ordered counterclockwise. Vertices of output polygons are ordered counterclockwise. However, vertices of polygons representing holes (as part of the output) are ordered clockwise. For instance, if the result of a boolean operation is a polygon with some holes, then this result will be represented as a sequence of polygons, where the first one representing the outer boundary of the contour is ordered counterclockwise and the following ones representing the inner contours (holes) are ordered clockwise.

For two simple polygons A and B, the boolean operations are defined:

Intersection test of two polygons (*CGAL_do_intersect(A,B)*): This checks if the two polygons A and B do intersect without computing the intersection area. It returns *true* if the polygons A and B do intersect, otherwise *false* will be returned.

Intersection of two polygons (*CGAL_intersection(A,B)*): It performs the operation $C := A \cap B$ and returns the result C as a maybe empty sequence of objects (*CGAL_Object*), i.e. a sequence containing objects of type *CGAL_Point_2*, *CGAL_Segment_2*, and *CGAL_Polygon_2*. When A and B are both triangles (*CGAL_Triangle_2*) or when A and B are both iso-oriented rectangles (*CGAL_Iso_rectangle*) the result can only be a single object and therefore the intersection will return a *CGAL_Object* instead of a sequence of objects.

Union of two polygons (*CGAL_union(A,B)*): It performs the operation $C := A \cup B$ and returns the result C as a maybe empty sequence of objects (*CGAL_Object*), i.e. a sequence containing objects of type *CGAL_Polygon_2*. The first polygon gives the counterclockwise-ordered outer boundary of the contour of the union and the following polygons (if existing) define the inner clockwise-ordered contours (the holes). If $A \cap B$ is exactly one point (a vertex of at least one of the two input polygons), the union returns one non-simple polygon. If $A \cap B$ is empty, then a sequence will be returned consisting of A and B both represented as polygons with counterclockwise ordered contour.

Difference of two polygons (*CGAL_difference(A,B)*): This performs the operation $C := A \setminus interior(B)$ and returns the result C as a maybe empty sequence of objects (*CGAL_Object*), i.e. a sequence containing objects of type *CGAL_Point_2*, *CGAL_Segment_2*, and *CGAL_Polygon_2*. If $A \cap B$ is empty then A will be returned. If $A \cap B = A$ (i.e. $A \subseteq B$), then the empty sequence will be returned. If $A \cap B = B$ (i.e. $B \subseteq A$, that means B is a hole of A), then a sequence containing (two) objects (A, B) of type *CGAL_Polygon_2* will be returned, where the second one represents a hole and has clockwise order.

Parameters

The boolean operations described in the following have one or more template parameters. The number of template parameters as well as the type of template parameters differs for the different operations. Here we give a list of all template parameters which might occur and some explanation of where they stand for.

Traits:

A value for the template parameter *Traits* is a boolean operations traits class. We provide a standard boolean operations traits class for convenience to the user, but the user can define ones own traits class as well. For more details about boolean operations traits classes look at the table in the following paragraph on Types, and in the section on the standard traits class (7.3.1).

R:

This template parameter defines the representation class[1]. Common examples of *R* are: *CGAL_Cartesian<double>*, *CGAL_Homogeneous<float>*, or *CGAL_Cartesian<CGAL_Rational>*.

Container:

The container type *Container* for a polygon. The user must pre-instantiate this for instance by *list<CGAL_Point_2<CGAL_Cartesian<CGAL_Rational> > >*.

OutputIterator:

The type of the iterator pointing to the container in which the output is stored: *OutputIterator*. The user must pre-instantiate this for instance by *list<Object>::const_iterator*.

ForwardIterator:

The type of the iterator pointing to the container in which the input is stored: *ForwardIterator*. The user must pre-instantiate this for instance by *list<Point>::const_iterator*.

In the table below we describe the types which are defined in a boolean operations treats class. Such a traits class contains the most important type definitions used in the algorithms. These types can

[1] A detailed description of the representation class can be found in Part 1 of the CGAL-Reference Manual

be changed by the user, if needed. In the first column of the table the types present in a boolean operations treats class are listed. In the second column a brief description of the respective types is given, and in the third column the default value for each of the types in given as they are defined in the standard boolean operations treats class. For more details on the standard boolean operations treats class we refer to Section 7.3.1.

type	description	standard
Traits::Point	2D point	*CGAL_Point_2<R> >*
Traits::Segment	2D segment	*CGAL_Segment_2<R> >*
Traits::Triangle	2D triangle	*CGAL_Triangle_2<R >*
Traits::Iso_rectangle	2D iso-oriented rectangle	*CGAL_Iso_rectangle_2<R >*
Traits::Container	container type for polygon	*list<CGAL_Point_2<R> >*
Traits::Polygon	2D polygon	*CGAL_Polygon_2<CGAL_Polygon_traits_2<R>, Container >*

Operations

Operations on 2D Iso rectangles

#include <CGAL/bops_Iso_rectangle_2.h>

template <class R>
bool CGAL_do_intersect(CGAL_Iso_rectangle_2<R> A, CGAL_Iso_rectangle_2<R> B)

returns *true* if the iso-oriented rectangles A and B do intersect.

template <class R, class OutputIterator>
OutputIterator

CGAL_intersection(CGAL_Iso_rectangle_2<R> A,
CGAL_Iso_rectangle_2<R> B,
OutputIterator object_it)

computes the intersection of two iso-oriented rectangles and places the resulting object of type *CGAL_Object* in a container of the type corresponding to the output iterator (*OutputIterator*) *object_it* which points to the resulting object. The function returns an output iterator (*OutputIterator*) pointing to the position beyond the end of the container. Each object part of the output is either a point (*CGAL_Point_2*), or a segment (*CGAL_Segment_2*), or an iso-oriented rectangle (*CGAL_Iso_rectangle_2*). In case of an empty intersection no objects are put into the output operator.

template <class R, class OutputIterator>
OutputIterator

CGAL_union(CGAL_Iso_rectangle_2<R> A,
CGAL_Iso_rectangle_2<R> B,

OutputIterator list_of_objects_it)

computes the union of two iso-oriented rectangles and places all resulting objects as a sequence of objects of type *CGAL_Object* in a container of type corresponding to the type of output iterator (*OutputIterator*) *list_of_objects_it* which points to the first object in the sequence. The function returns an output iterator (*OutputIterator*) pointing to the position beyond the end of the sequence. The sequence may contain an iso-oriented rectangle (*CGAL_Iso_rectangle_2*), or a simple polygon (*CGAL_Polygon_2*), or two iso-oriented rectangles (*CGAL_Iso_rectangle_2*, in case $A \cap B$ is empty).

template <*class R, class OutputIterator*>
OutputIterator

CGAL_difference(CGAL_Iso_rectangle_2<*R*> *A,*
CGAL_Iso_rectangle_2<*R*> *B,*
OutputIterator list_of_objects_it)

computes the difference ($A \setminus B$) of two iso-oriented rectangles and places the resulting objects as *CGAL_Object* in a container of type corresponding to the type of output iterator (*OutputIterator*) *list_of_objects_it* which points to the first object in the sequence. The function returns an output iterator (*OutputIterator*) pointing to the position beyond the end of the sequence. Each object is either a simple polygon (*CGAL_Polygon_2)* or an iso-oriented rectangle (*CGAL_Iso_rectangle_2*). If $B \subseteq A$ no object will be put into the output iterator.

Operations on 2D Triangles

#include <*CGAL/bops_Triangle_2.h*>

template <*class R*>
bool CGAL_do_intersect(CGAL_Triangle_2<*R*> *A, CGAL_Triangle_2*<*R*> *B)*

returns *true* if the triangles A and B do intersect.

template <*class R, class OutputIterator*>
OutputIterator

CGAL_intersection(CGAL_Triangle_2<*R*> *A,*
CGAL_Triangle_2<*R*> *B,*

OutputIterator object_it)

computes the intersection of two triangles and places the resulting object of type *CGAL_Object* in a container of the type corresponding to the output iterator (*OutputIterator*) *object_it* which points to the resulting object. The function returns an output iterator (*OutputIterator*) pointing to the position beyond the end of the container. The resulting object is either a point (*CGAL_Point_2*), or a segment (*CGAL_Segment_2*), or a triangle (*CGAL_Triangle_2*), or a convex polygon (*CGAL_Polygon_2* with *is_convex() == true*). In case of an empty intersection no object is put into the output operator.

template <class R, class OutputIterator>
OutputIterator

CGAL_union(CGAL_Triangle_2<R> A,
CGAL_Triangle_2<R> B,
OutputIterator list_of_objects_it)

computes the union of two triangles and places all resulting objects as a sequence of objects of type *CGAL_Object* in a container of type corresponding to the type of output iterator (*OutputIterator*) *list_of_objects_it* which points to the first object in the sequence. The function returns an output iterator (*OutputIterator*) pointing to the position beyond the end of the sequence. The sequence may contain a triangle (*CGAL_Triangle_2*), or a simple polygon (*CGAL_Polygon_2*), or two triangles (*CGAL_Polygon_2*, in case $A \cap B$ is empty).

template <class R, class OutputIterator>
OutputIterator

CGAL_difference(CGAL_Triangle_2<R> A,
CGAL_Triangle_2<R> B,
OutputIterator list_of_objects_it)

Computes the difference $(A \setminus B)$ of two triangles and places all resulting objects as a sequence of objects of type *CGAL_Object* in a container of type corresponding to the type of output iterator (*OutputIterator*) *list_of_objects_it* which points to the first object in the sequence. It returns an output iterator (*OutputIterator*) pointing to position beyond the end of the sequence. Each object in the sequence is either a triangle (*CGAL_Triangle_2*), or a simple polygon (*CGAL_Polygon_2*). If $B \subseteq A$ no object will be put into the output iterator.

Operations on 2D Polygons

#include <CGAL/bops_Polygon_2.h>

template <class R, class Container>
bool CGAL_do_intersect(CGAL_Polygon_2<CGAL_Polygon_traits_2<R>,Container> A,

CGAL_Polygon_2<CGAL_Polygon_traits_2<R>,Container> B)

returns *true* if the polygons A and B do intersect.
Precondition: A and B simple polygons, their vertices are in counterclockwise order.

template <class R, class Container, class OutputIterator>
OutputIterator

CGAL_intersection(CGAL_Polygon_2<CGAL_Polygon_traits_2<R>,Container> A,
CGAL_Polygon_2<CGAL_Polygon_traits_2<R>,Container> B,
OutputIterator list_of_objects_it)

computes the intersection of two simple polygons and places all resulting objects as a sequence of objects of type *CGAL_Object* in a container of type corresponding to the type of output iterator (*OutputIterator*) *list_of_objects_it* which points to the first object in the sequence. The function returns an output iterator (*OutputIterator*) pointing to the position beyond the end of the sequence. In case of an empty intersection no objects are put into the output operator.
Precondition: A and B simple polygons, their vertices are in counterclockwise order.

template <class R, class Container, class OutputIterator>
OutputIterator

CGAL_union(CGAL_Polygon_2<CGAL_Polygon_traits_2<R>,Container> A,
CGAL_Polygon_2<CGAL_Polygon_traits_2<R>,Container> B,
OutputIterator list_of_objects_it)

computes the union of two simple polygons and places all resulting objects as a sequence of objects of type *CGAL_Object* in a container of type corresponding to the type of output iterator (*OutputIterator*) *list_of_objects_it* which points to the first object in the sequence. The function returns an output iterator (*OutputIterator*) pointing to the position beyond the end of the sequence.
Precondition: A and B are simple polygons, their vertices are in counterclockwise order.

template <class R, class Container, class OutputIterator>
OutputIterator

CGAL_difference(CGAL_Polygon_2<CGAL_Polygon_traits_2<R>,Container> A,
CGAL_Polygon_2<CGAL_Polygon_traits_2<R>,Container> B,
OutputIterator list_of_objects_it)

computes the difference of two simple polygons ($A \setminus B$) and places the resulting object as *CGAL_Object* in a container of type corresponding to the type of output iterator (*OutputIterator*) *list_of_objects_it* which points to the first object in the sequence. The function returns an output iterator (*OutputIterator*) pointing to the position beyond the end of the sequence. The difference can be empty in which case no object will be put in the output iterator.
Precondition: A and B simple polygons, their vertices are in counterclockwise order.

Operations on 2D Polygons defined by containers

#include <CGAL/bops_Container_Polygon_2.h>

```
template <class ForwardIterator, class Traits>
bool    CGAL_do_intersect( ForwardIterator Afirst,
                           ForwardIterator Alast,
                           ForwardIterator Bfirst,
                           ForwardIterator Blast,
                           Traits &)
```

with polygon A of type *Traits::Polygon* defined by the vertices of type *Traits::Point* in the range $[Afirst, Alast)$ and for polygon B of type *Traits::Polygon* defined by the vertices of type *Traits::Point* in the range $[Bfirst, Blast)$. It returns *true* if the polygons A and B do intersect.

Precondition: A and B simple polygons, their vertices are in counterclockwise order.

```
template <class ForwardIterator, class OutputIterator, class Traits>
OutputIterator

        CGAL_intersection( ForwardIterator Afirst,
                           ForwardIterator Alast,
                           ForwardIterator Bfirst,
                           ForwardIterator Blast,
                           Traits &,
                           OutputIterator list_of_objects_it)
```

with polygon A of type *Traits::Polygon* defined by the vertices of type *Traits::Point* in the range $[Afirst, Alast)$ and with polygon B of type *Traits::Polygon* defined by the vertices of type *Traits::Point* in the range $[Bfirst, Blast)$, computes the intersection of two simple polygons and places all resulting objects as a sequence of objects of type *CGAL_Object* in a container of type corresponding to the type of output iterator (*OutputIterator*) *list_of_objects_it* which points to the first object in the sequence. The function returns an output iterator (*OutputIterator*) pointing to the position beyond the end of the sequence. In case of an empty intersection no objects are put into the output operator. If an object in the sequence to which the output iterator refers is a polygon, then this polygon is of type *Traits::Polygon* with vertices of type *Traits::Point* and container *Traits::Container*.

Precondition: A and B simple polygons, their vertices are in counterclockwise order.

```
template <class ForwardIterator, class OutputIterator, class Traits>
OutputIterator

        CGAL_union( ForwardIterator Afirst,
                    ForwardIterator Alast,
                    ForwardIterator Bfirst,
                    ForwardIterator Blast,
                    Traits &,
                    OutputIterator list_of_objects_it)
```

with polygon A of type *Traits::Polygon* defined by the vertices of type *Traits::Point* in the range $[Afirst, Alast)$ and with polygon B of type *Traits::Polygon* defined by the vertices of type *Traits::Point* in the range $[Bfirst, Blast)$. Computes the union of two simple polygons ($A \cup B$) and places all resulting

objects as a sequence of objects of type *CGAL_Object* in a container of type corresponding to the type of output iterator (*OutputIterator*) *list_of_objects_it* which points to the first object in the sequence. It returns an output iterator (*OutputIterator*) pointing to position beyond the end of the sequence. If an object in the sequence to which the output iterator refers is a polygon, then this polygon is of type *Traits::Polygon* with vertices of type *Traits::Point* and container *Traits::Container*.

Precondition: A and B simple polygons, their vertices are in counterclockwise order.

template <*class ForwardIterator, class OutputIterator, class Traits*>
OutputIterator

CGAL_difference(ForwardIterator Afirst,
ForwardIterator Alast,
ForwardIterator Bfirst,
ForwardIterator Blast,
Traits &,
OutputIterator list_of_objects_it)

with polygon A of type *Traits::Polygon* defined by the vertices of type *Traits::Point* in the range $[Afirst, Alast)$ and with polygon B of type *Traits::Polygon* defined by the vertices of type *Traits::Point* in the range $[Bfirst, Blast)$, computes the difference of two simple polygons $(A \setminus B)$ and places all resulting objects of type *CGAL_Object* in a container of type corresponding to the type of output iterator (*OutputIterator*) *list_of_objects_it* which points to the first object in the sequence. The function returns an output iterator (*OutputIterator*) pointing to the position beyond the end of the sequence. The difference can be empty in which case no object will be put in the output iterator. If an object in the sequence to which the output iterator refers is a polygon, then this polygon is of type *Traits::Polygon* with vertices of type *Traits::Point* and container *Traits::Container*.

Precondition: A and B simple polygons, their vertices are in counterclockwise order.

Example

Principally, Boolean operations work as follows, illustrated by the example of intersecting two polygons:

1. To use the predefined boolean operations traits class *CGAL_bops_traits_2*<*R*>, include the file *bops_traits_2.h*. For details see Section 7.3.1.
2. Instantiation of two polygons that can be triangles, iso oriented rectangles, or simple polygons. A polygon can be represented as an object (e.g. *CGAL_Polygon_2*) or as a sequence of points held in a sequence container (e.g. *STL*-list).
3. Performing the boolean operation:

```
CGAL_intersection(A.begin(), A.end(), B.begin(), B.end(), traits, result);
CGAL_intersection(A, B, result);
```

4. Taking the result of the operation:

   The result consists of a sequence of points, segments, and simple polygons. Hence, the user has to check what type of element has to be performed for further computations.

The full example (depicted in figure 7.1) instantiates two simple polygons and computes their intersection. Note: here the non-exact (builtin) number type *float* is used.

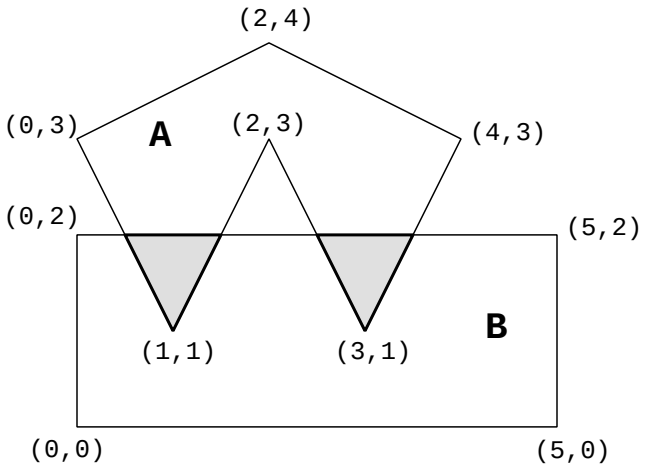


Figure 7.1: Intersection example of two simple polygons (in cartesian coordinates).

```
#include <CGAL/Homogeneous.h>
#include <CGAL/Cartesian.h>
#include <CGAL/basic.h>
#include <iostream.h>
#include <CGAL/bops_Polygon_2.h>

typedef float TestNum;

#ifdef USE_CARTESIAN_COORDINATES
      typedef CGAL_Cartesian<TestNum>        R_type;
#else
      typedef CGAL_Homogeneous<TestNum>      R_type;
#endif

typedef CGAL_Point_2<R_type>                 Point_2;
typedef CGAL_Segment_2<R_type>               Segment_2;

typedef list< Point_2 >                       Container;
typedef CGAL_Polygon_traits_2<R_type>        Polygon_traits_2;
typedef CGAL_Polygon_2< Polygon_traits_2, Container >  Polygon_2;
typedef vector<Point_2>                       Input_container;


int example_intersection(
  const Input_container& container_A,
  const Input_container& container_B
) {
  /* instantiate Polygon A and B with containers */
  Polygon_2 A(container_A.begin(), container_A.end());
  Polygon_2 B(container_B.begin(), container_B.end());

  /* declaration of the result container */
  list<CGAL_Object> result;
```

```
  /* performing intersection of A and B */
  CGAL_intersection(A, B, back_inserter(result));

  cout << "result size=" << result.size() << endl;

  /* possible results */
  Point_2   point;
  Segment_2 segment;
  Polygon_2 polygon;

  list<CGAL_Object>::const_iterator it;
  for( it= result.begin(); it != result.end(); it++) {
    if( CGAL_assign( polygon, *it) ) {
      cout << "PGN: " << polygon << endl;    /* polygon detected */
    }
    else if( CGAL_assign( segment, *it) ) {
      cout << "SEG: " << segment << endl;    /* segment detected */
    }
    else if( CGAL_assign( point, *it) )  {
      cout << "PNT:" << point << endl;       /* point detected */
    }
    else {
      cout << "undefined object " << endl;   /* nothing detected */
    }
  }

  return result.size();
}



int main(void)
{
  Input_container container_A(6), container_B(4);

  container_A[0]= Point_2(2,4); /* description of polygon A */
  container_A[1]= Point_2(0,3);
  container_A[2]= Point_2(1,1);
  container_A[3]= Point_2(2,3);
  container_A[4]= Point_2(3,1);
  container_A[5]= Point_2(4,3);

  container_B[0]= Point_2(0,2); /* description of polygon B */
  container_B[1]= Point_2(0,0);
  container_B[2]= Point_2(5,0);
  container_B[3]= Point_2(5,2);

  example_intersection( container_A, container_B);
  return 0;
}
```

The output of our small example program looks like as follows:

```
result size=2
PGN: 3 1 1 1 15 20 10 5 20 10
PGN: 3 25 20 10 3 1 1 35 20 10
```

Its interpretation says that the result consists of two polygons with size three (i.e. two triangles) given by their vertices in homogeneous coordinates and counterclockwise order: ((1,1,1), (15,20,10), (5,20,10)) and ((25,20,10),(3,1,1),(35,20,10)).

From this (full) example can be seen how a boolean operation could be applied in a safe and practical way. The following sketch of a more advanced example shows how traits classes can be used for performing boolean operations.

```
#include <CGAL/bops_traits_2.h>
#include <CGAL/bops_Polygon_2_container.h>

typedef CGAL_Cartesian<double> R;
typedef CGAL_bops_traits_2<R> Traits;
typedef Traits::Polygon Polygon;
typedef Traits::Output_object_container Output_container;

void myFunction() {
  Polygon polygon;
  Traits  traits_class;

  polygon A, B; /* instantiate A and B */
  /* ... */

  Output_container result;

  /* apply a boolean operation: */
  CGAL_intersection(A.begin(), A.end(), B.begin(), B.end(),
                    traits_class, back_inserter(result));

  /* do something with the result: */
  /* ... */

}
```

Implementation

The algorithms for boolean operations of two polygons are (efficient) specialized methods for specific objects. Depending on the polygon type we switch internally to the best suited routines. For instance, for two triangles we switch to a more efficient algorithm than the general one on polygons.

The memory consumption is $O(n)$ (where n is the whole number of vertices of the input polygons). The time complexity is $O(n^2)$ for simple polygons, and $O(1)$ for triangles and iso-oriented rectangles.

Note: As mentioned above, the result is sometimes returned as iterators pointing to a *list<CGAL_Object>*, where *list* is a *STL list container*, which implements a double connected list (include file: *list.h*).

See Also

CGAL_Intersection, *CGAL_Polygon*, *CGAL_Triangle*, and *CGAL_Iso_rectangle*.

7.3 Boolean Operations Traits Class Implementations

7.3.1 Traits Class Implementation using the two-dimensional CGAL Kernel (*CGAL_bops_traits_2<R>*)

Since all geometric objects on which boolean operations act are parameterized with a template for the representation class, also the predefined boolean operations traits class is a template class with a parameter for the representation class <*R*>. The predefined boolean operations traits class is called *CGAL_bops_traits_2<R>*.

#include <CGAL/bops_traits_2.h>

Types

typedef CGAL_Object — *Object;*

typedef CGAL_Bbox_2 — *Bbox;*

typedef CGAL_Point_2<R> — *Point;*

typedef CGAL_Segment_2<R> — *Segment;*

typedef CGAL_Triangle_2<R> — *Triangle;*

typedef CGAL_Iso_rectangle_2<R> — *Iso_rectangle;*

typedef list<Object> — *Output_object_container;*

typedef list<Point> — *Container;*

typedef Container — *Input_polygon_container;*

typedef list<Point> — *Output_polygon_container;*

typedef CGAL_Polygon_2<CGAL_Polygon_traits_2<R>,Container> — *Polygon;*

typedef Polygon — *Input_polygon;*

typedef Polygon::Vertex_const_iterator — *Polygon_vertex_const_iterator;*

typedef CGAL_Polygon_2<R,Output_polygon_container> — *Output_polygon;*

For more information about these types see 7.4.

Creation

CGAL_bops_traits_2<R> Traits;

default constructor

Operations

bool *Traits.do_overlap(Bbox a, Bbox b)*

returns *CGAL_do_overlap(a,b).*

bool *Traits.box_is_contained_in_box(Bbox a, Bbox b)*

returns *true* iff *xmin(b) <= xmin(a) and xmax(b) >= xmax(a) and ymin(b) <= ymin(a) and ymax(b) >= ymax(a).*

bool *Traits.is_equal(Point p1, Point p2)*

returns *p1 == p2.*

bool *Traits.less_x(Point p1, Point p2)*

returns *p1.x() < p2.x();*

bool *Traits.is_leftturn(Point p0, Point p1, Point p2)*

returns *CGAL_leftturn(p0, p1, p2).*

Object *Traits.Make_object(Point p)*

returns *CGAL_make_object(p).*

Object *Traits.Make_object(Segment seg)*

returns *CGAL_make_object(seg).*

Object *Traits.Make_object(Output_polygon pgon)*

returns *CGAL_make_object(pgon).*

Object *Traits.Make_object(Iso_rectangle irect)*

returns *CGAL_make_object(irect).*

Object *Traits.Make_object(Triangle triangle)*

returns *CGAL_make_object(triangle).*

Bbox *Traits.get_Bbox(Polygon pgon)*

returns *pgon.bbox().*

bool *Traits.has_on_bounded_side(Polygon pgon, Point pt)*

returns *pgon.has_on_bounded_side(pt).*

void *Traits.reverse_orientation(Polygon& pgon)*

performs *pgon.reverse_orientation().*

Polygon_vertex_const_iterator

Traits.most_left_vertex(Polygon pgon)

returns *pgon.left_vertex().*

7.4 Boolean Operations Traits Class Requirements (*Traits<R>*)

Types

Traits<R>:: Object; — A common object type like *CGAL_object*. The "real" objects (e.g. *Output_polygon*, *Segment*, *Point*, ...) are mapped into *Object* using the member function *Make_object* (see below to the operations description).

Traits<R>:: Point; — The point type on which the boolean operations algorithms operate. It should represent a two dimensional point with coordinates of type *R::NT*. The type must provide a copy constructor, an assignment, and an equality test. Furthermore a constructor for cartesian coordinates (*Point(R::NT,R::NT)*), a constructor for homogenous coordinates (*Point(R::NT,R::NT,R::NT)*), and the operations *R::NT Point::x()* and *R::NT Point::y()* are needed, which returns the value of the x- or y-coordinate, respectively. This point type represents the vertices of segments, triangles, iso-oriented rectangles, and polygons, which occur as possible results of a boolean operation.

Traits<R>:: Segment; — Line segment type. It should represent a two dimensional segment defined by the two extremal points of type *Point*. The type must provide a copy constructor, an assignment, and an equality test. Furthermore a constructor by two points (*Segment(Point,Point)*) will be needed.

Traits<R>:: Triangle; — The type of triangles on which boolean operations can be performed. It should represent a triangle in two dimensional Euclidean space with vertices of type *Point*. The type must provide a copy constructor, an assignment, and an equality test. Furthermore a constructor by three points (*Triangle(Point,Point,Point)*) will be needed.

Traits<R>:: Iso_rectangle; — The type of iso-oriented rectangles on which boolean operations can be performed. It should represent an iso-oriented rectangle in two dimensional Euclidean space with vertices of type *Point*. The type must provide a copy constructor, an assignment, and an equality test. Furthermore a constructor by three points (*Iso_rectangle(Point,Point,Point)*) will be needed.

Traits<R>:: Polygon; — The type of polygons on which boolean operations will be performed (i.e. *Input_polygon*, there also exists an *Output_polygon*). This type should represent a simple polygon in two dimensional Euclidean space with vertices of type *Point*. Usually it is based on *Container*. The type must provide a copy constructor, an assignment, and an equality test.

Traits<R>:: Polygon_vertex_const_iterator;

A (const) vertex forward iterator for *Polygon*.

typedef Polygon — *Input_polygon;*

Traits<R>:: Output_polygon; — The resulting polygon type. Usually based on *Output_polygon_container*. This type must represent a simple polygon in two dimensional Euclidean space with vertices of type *Point*.

Traits<R>:: Bbox; — A bounding box type used internally in the computation of boolean operations. The type must provide an equality test.

Traits<R>:: Container; — The container in which points of type *Point* are stored for further use in polygons on which boolean operations are to be performed (*Input_polygon_container*). Note: The only usage of *Container* is to define *Polygon*.

typedef Container — *Input_polygon_container;*

Traits<R>:: Output_polygon_container;

The container in which points of type *Point* are stored for further use in polygons (*Output_polygon*) which result by performing boolean operations. Note: The only usage of *Output_polygon_container* is to define *Output_polygon*.

Traits<R>:: Output_object_container;

The container in which objects of type *Object* are stored. This container represents the result of a boolean operation, since each resulting object (e.g. *Output_Polygon*, *Segment*, ...) is mapped into a *Object*. and afterwards stored into a container of this type *Output_object_container*. It must provide a operation *Output_object_container::push_back(Object)*.

Creation

Traits<R> bops_traits;

default constructor

Operations

bool *bops_traits.do_overlap(Bbox a, Bbox b)*

returns *true* iff the boxes do overlap.

bool *bops_traits.box_is_contained_in_box(Bbox a, Bbox b)*

returns *true* iff *a* is contained in *b*, entirely.

bool *bops_traits.is_equal(Point p1, Point p2)*

returns *true* iff the value of *p1* is equal to the value of *p2*.

bool *bops_traits.less_x(Point p1, Point p2)*

returns *true* iff the x-ccordinate of *p1* is smaller than that of *p2*.

bool *bops_traits.is_leftturn(Point p0, Point p1, Point p2)*

returns *true* iff *p2* lies on the left of the oriented line through *p0* and *p1*.

Object *bops_traits.Make_object(Point p)*

maps *Point* into *Object* and returns it.

Object *bops_traits.Make_object(Segment p)*

maps *Segment* into *Object* and returns it.

Object *bops_traits.Make_object(Output_polygon p)*

maps *Output_polygon* into *Object* and returns it.

Object *bops_traits.Make_object(Iso_rectangle p)*

maps *Iso_rectangle* into *Object* and returns it.

Object *bops_traits.Make_object(Triangle p)*

Maps *Triangle* into *Object* and returns it.

Bbox *bops_traits.get_Bbox(Polygon pgon)*

returns the bounding box of *Polygon*.

bool *bops_traits.has_on_bounded_side(Polygon pgon, Point pt)*

returns *true* iff point *pt* lies on the bounded side of polygon *pgon*.

void *bops_traits.reverse_orientation()*

reverses the orientation of polygon *pgon*.

Polygon_vertex_const_iterator

bops_traits.most_left_vertex(Polygon pgon)

returns the most left vertex of polygon *pgon*.

Chapter 8

Convex Hulls and Extreme Points

8.1 Planar Convex Hull and Extreme Point Computation

Definition

A subset S of the plane is convex if with 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 computing the counterclockwise sequence of extreme points for a set of points, i.e. the counterclockwise sequence of points on the convex hull of the set of points. Furthermore, functions computing special extreme points are provided. Finally, there are functions that check whether a given sequence of points forms a (counter)clockwise convex polygon.

The last parameter *Traits* in the convex hull and extreme point functions is a traits class that defines the primitives that are used in the algorithms. For example *Traits::Point_2* is a mapping on a point class.

CGAL provides implementations of convex hull traits classes as described in 8.2. A default implementation *CGAL_convex_hull_traits_2<R>* is provided for the two-dimensional CGAL kernel. This traits class is used in default versions of the functions that do not have a traits class parameter. Such default versions exists for all the functions specified below. Own convex hull traits classes can be designed according to the requirements for such traits classes listed in 8.4.

Assertions

As supposed for all code in CGAL the code in this module checks preconditions, postconditions and assertions. It uses infix *CH* in the name of the assertions. For the convex hull algorithm(s), postcondition check tests only convexity (if not disabled), but not containment of the input points in the polygon defined by the output points. The latter is considered an expensive checking and can be enabled by defining *CGAL_CH_CHECK_EXPENSIVE*. See also Section 2 of the Reference Manual Part 0, General Introduction.

8.1.1 Convex Hull

CGAL provides a function *CGAL_convex_hull_points_2()* to compute the counterclockwise sequence of extreme points for a set of points.

#include <CGAL/convex_hull_2.h>

template <class InputIterator, class OutputIterator>
OutputIterator CGAL_convex_hull_points_2(InputIterator first,
InputIterator beyond,
OutputIterator result,
Traits ch_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*.
TRAITS: uses *Traits::Point_2*, *Traits::Less_xy*, *Traits::Less_yx*, *Traits::Right_of_line*, and *Traits::Leftturn*.

Besides the default algorithm for computing convex hulls, implementations of some classical convex hull algorithms are available, with the same semantics as the default algorithm.

#include <CGAL/ch_akl_toussaint.h>

template <class ForwardIterator, class OutputIterator, class Traits>
OutputIterator CGAL_ch_akl_toussaint(ForwardIterator first,
ForwardIterator beyond,
OutputIterator result,
Traits ch_traits)

TRAITS: uses *Traits::Point_2*, *Traits::Less_xy*, *Traits::Less_yx*, *Traits::Right_of_line*, and *Traits::Leftturn*.

#include <CGAL/ch_graham_andrew.h>

template <class InputIterator, class OutputIterator, class Traits>
OutputIterator CGAL_ch_graham_andrew(InputIterator first,
InputIterator beyond,
OutputIterator result,
Traits ch_traits)

TRAITS: operates on *Traits::Point_2* using *Traits::Leftturn* and *Traits::Less_xy*.

#include <CGAL/ch_eddy.h>

template <*class InputIterator, class OutputIterator, class Traits*>
OutputIterator *CGAL_ch_eddy(InputIterator first,*
InputIterator beyond,
OutputIterator result,
Traits ch_traits)

TRAITS: uses the types *Traits::Point_2*, *Traits::Less_dist_to_line*, *Traits::Right_of_line*, and *Traits::Less_xy*.

#include <*CGAL/ch_bykat.h*>

template <*class InputIterator, class OutputIterator, class Traits*>
OutputIterator *CGAL_ch_bykat(InputIterator first,*
InputIterator beyond,
OutputIterator result,
Traits ch_traits)

TRAITS: uses the types *Traits::Point_2*, *Traits::Less_dist_to_line*, *Traits::Right_of_line*, and *Traits::Less_xy*.

#include <*CGAL/ch_jarvis.h*>

template <*class ForwardIterator, class OutputIterator, class Traits*>
OutputIterator *CGAL_ch_jarvis(ForwardIterator first,*
ForwardIterator beyond,
OutputIterator result,
Traits ch_traits)

TRAITS: uses types *Traits::Point_2*, *Traits::Less_rotate_ccw*, and *Traits::Less_xy*.

Implementation

CGAL_ch_akl_toussaint() follows [AT78], *CGAL_ch_graham_andrew()* follows [And79] and the presentation given in [Meh84]. Both have worst-case running time $O(n \log n)$, where n is the number of points. *CGAL_ch_eddy()* follows [Edd77], *CGAL_ch_bykat()* is a non-recursive variant of *CGAL_ch_eddy()* as described in [Byk78], and *CGAL_ch_jarvis()* follows [Jar73]. The latter algorithms have worst-case running time $O(nh)$, where n is the number of points and h is the number of extreme points.

Example

In the following example we use the STL-compliant interface of *CGAL_Polygon_2* to construct the convex hull polygon from the sequence of extreme points. Point data are read from standard in, the convex hull polygon is shown in a *leda_window*. Remember, that the default traits class *CGAL_convex_hull_traits_2*<*R*> is used with the points of the two-dimensional CGAL kernel, if no traits class is passed to the convex hull algorithm.

```
#include <CGAL/basic.h>
#include <list.h>
```

```
#include <CGAL/Cartesian.h>
#include <CGAL/Point_2.h>
#include <CGAL/convex_hull_2.h>
#include <CGAL/Polygon_2.h>

typedef CGAL_Cartesian<int>                          RepCls;
typedef CGAL_Point_2<RepCls>                         Point_2;
typedef CGAL_Polygon_2<RepCls, list<Point_2> >       Polygon_2;

int
main()
{
  Polygon_2                                    CH;
  istream_iterator< Point_2, ptrdiff_t >       in_start( cin );
  istream_iterator< Point_2, ptrdiff_t >       in_end;

  CGAL_convex_hull_points_2( in_start, in_end,
                             inserter(CH, CH.vertices_begin()) );
  CGAL_Window_stream  W( 512, 512, 50, 50);
  W.init( -256.0, 255.0, -256.0);
  W << CH;

  Point_2  p;
  W >> p;          // wait for mouse click in window
  return 0;
}
```

8.1.2 Lower Hull and Upper Hull

CGAL also provides functions to compute the counterclockwise sequence of extreme points on the lower hull as well as on the upper hull.

#include <CGAL/convex_hull_2.h>

template <class InputIterator, class OutputIterator>
OutputIterator CGAL_lower_hull_points_2(InputIterator first,
InputIterator beyond,
OutputIterator result,
Traits ch_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*.
TRAITS: uses *Traits::Point_2*, *Traits::Less_xy*, *Traits::Less_yx*, *Traits::Right_of_line*, and *Traits::Leftturn*.

template <*class InputIterator, class OutputIterator*>
OutputIterator *CGAL_upper_hull_points_2(InputIterator first,*
InputIterator beyond,
OutputIterator result,
Traits ch_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. leftmost and rightmost point are equal) the extreme point is not reported.
Precondition: The source range [*first*,*beyond*) does not contain *result*.
TRAITS: uses *Traits::Point_2*, *Traits::Less_xy*, *Traits::Less_yx*, *Traits::Right_of_line*, and *Traits::Leftturn*.

The different treatment of the case that all points are equal ensures that concatenation of lower and upper hull points gives the sequence of extreme points.

8.1.3 Convex Hull Utilities

CGAL provides some functions to compute certain subsequences of the sequence of extreme points on the convex hull.

template <*class ForwardIterator, class OutputIterator, class Traits*>
OutputIterator *CGAL_ch_jarvis_march(ForwardIterator first,*
ForwardIterator beyond,
Traits::Point_2 start_p,
Traits::Point_2 stop_p,
OutputIterator result,
Traits ch_traits)

generates the counterclockwise ordered 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*).
TRAITS: uses *Traits::Less_rotate_ccw*.

template <*class BidirectionalIterator, class OutputIterator, class Traits*>
OutputIterator *CGAL_ch_graham_andrew_scan(BidirectionalIterator first,*
BidirectionalIterator beyond,
OutputIterator result,

Traits ch_traits)

computes the sorted sequence of extreme points which are not left of pq, where p is the value of *first* and q is the value of *beyond* -1. It reports this sequence counterclockwise in a range starting at *result*, starting with p; point q is omitted.
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 works.
TRAITS: uses *Traits::Leftturn* operating on the point type *Traits::Point_2*.

Example

In the following example *CGAL_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* predicate, see 8.4. According to the definition of *Less_rotate_ccw*, 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 precondition of *CGAL_ch_graham_andrew_scan()* are satisfied.

```
template <class InputIterator, class OutputIterator, class Traits>
OutputIterator
CGAL_ch_graham_anderson( InputIterator  first, InputIterator  beyond,
                         OutputIterator result,
                         const Traits&  ch_traits)
{
  typedef typename Traits::Less_xy          Less_xy;
  typedef typename Traits::Point_2          Point_2;
  typedef typename Traits::Less_rotate_ccw  Less_rotate_ccw;

  if (first == beyond) return result;
  vector< Point_2 >  V;
  copy( first, beyond, back_inserter(V) );
  vector< Point_2 >::iterator it = min_element(V.begin(), V.end(), Less_xy());
  sort( V.begin(), V.end(), Less_rotate_ccw(*it) );
  if ( *(V.begin()) == *(V.rbegin()) )
  {
      *result = *(V.begin());  ++result;
      return result;
  }
  return CGAL_ch_graham_andrew_scan( V.begin(), V.end(), res, ch_traits);
}
```

Anderson's variant of the Graham scan is usually inferior to Andrew's variant because of its higher arithmetic demand.

8.1.4 Extreme Points in Coordinate Directions

CGAL provides functions computing extreme points with respect to the directions of the underlying

Cartesian coordinate system. Each function name has a part that tells you which extreme points are reported in the reference arguments and in which order. n means north, w west, s south, and e east, which stands for a point with maximal y-, minimal x-, minimal y-, and maximal x-coordinate resp. For the x-coordinate comparison, xy-lexicographical order is used, for y-coordinate comparison yx-lexicographical order.

#include <CGAL/ch_selected_extreme_points_2.h>

```
template <class ForwardIterator>
void                CGAL_ch_nswe_point( ForwardIterator first,
                                        ForwardIterator beyond,
                                        ForwardIterator& n,
                                        ForwardIterator& s,
                                        ForwardIterator& w,
                                        ForwardIterator& e,
                                        Traits ch_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 inequations $*s \leq_{yx} *it$, $*w \leq_{xy} *it$, and $*e \geq_{yx} *it$ hold respectively for all iterators it in the range.
TRAITS: uses the types *Traits::Less_xy*, and *Traits::Less_yx*.

Analogously, we have

```
template <class ForwardIterator>
void                CGAL_ch_we_point( ForwardIterator first,
                                      ForwardIterator beyond,
                                      ForwardIterator& w,
                                      ForwardIterator& e,
                                      Traits ch_traits)
```

```
template <class ForwardIterator>
void                CGAL_ch_ns_point( ForwardIterator first,
                                      ForwardIterator beyond,
                                      ForwardIterator& n,
                                      ForwardIterator& s,
                                      Traits ch_traits)
```

```
template <class ForwardIterator>
void                CGAL_ch_n_point( ForwardIterator first,
                                     ForwardIterator beyond,
                                     ForwardIterator& n,
                                     Traits ch_traits)
```

```
template <class ForwardIterator>
void                CGAL_ch_s_point( ForwardIterator first,
                                     ForwardIterator beyond,
                                     ForwardIterator& s,
                                     Traits ch_traits)
```

template <*class ForwardIterator*>
void *CGAL_ch_w_point(ForwardIterator first,*
ForwardIterator beyond,
ForwardIterator& w,
Traits ch_traits)

template <*class ForwardIterator*>
void *CGAL_ch_e_point(ForwardIterator first,*
ForwardIterator beyond,
ForwardIterator& e,
Traits ch_traits)

8.1.5 Convexity Checking

template <*class ForwardIterator, class Traits*>
bool *CGAL_is_ccw_strongly_convex_2(ForwardIterator first,*
ForwardIterator beyond,
Traits ch_traits)

returns *true*, iff the point elements in [*first*,*beyond*) form a counterclockwise oriented strongly convex polygon.
TRAITS: uses *Traits::Leftturn* and *Traits::Less_xy*.

template <*class ForwardIterator, class Traits*>
bool *CGAL_is_cw_strongly_convex_2(ForwardIterator first,*
ForwardIterator beyond,
Traits ch_traits)

returns *true*, iff the point elements in [*first*,*beyond*) form a clockwise oriented strongly convex polygon.
TRAITS: uses *Traits::rightturn()* and *Traits::Less_xy*.

8.2 Convex Hull Traits Class Implementations

The convex hull algorithms in Chapter 8 are parameterized with a traits class *Traits*. Besides the traits class *CGAL_convex_hull_traits_2<R>* used in the default versions of the functions, CGAL provides a traits class *CGAL_convex_hull_constructive_traits_2<R>* which also uses CGAL primitives and reuses intermediate results to speed up computation.

8.2.1 A Convex Hull Traits Class for the two-dimensional Kernel (*CGAL_convex_hull_traits_2<R>*)

Definition

The class *CGAL_convex_hull_traits_2<R>* is a convex hull traits class for the two-dimensional CGAL kernel, especially the two-dimensional point class. The class is parameterized with a representation class R. It is used in the default versions of the convex hull and extreme point algorithms.

#include <CGAL/convex_hull_traits_2.h>

Types

typedef CGAL_Point_2<R> Point_2;

CGAL_convex_hull_traits_2<R>:: Less_xy

CGAL_convex_hull_traits_2<R>:: Less_yx

CGAL_convex_hull_traits_2<R>:: Leftturn

CGAL_convex_hull_traits_2<R>:: Rightturn

CGAL_convex_hull_traits_2<R>:: Right_of_line

CGAL_convex_hull_traits_2<R>:: Less_dist_to_line

CGAL_convex_hull_traits_2<R>:: Less_rotate_ccw

Creation

CGAL_convex_hull_traits_2<R> ch_traits;

8.2.2 Another Convex Hull Traits Class for the two-dimensional Kernel (*CGAL_convex_hull_constructive_traits_2<R>*)

Definition

Like *CGAL_convex_hull_traits_2<R>*, the class *CGAL_convex_hull_constructive_traits_2<R>* is a convex hull traits class parameterized with a representation class for the two-dimensional CGAL kernel. However, unlike the default convex hull traits class *CGAL_convex_hull_traits_2<R>*, the class *CGAL_convex_hull_constructive_traits_2<R>* makes use of previously computed results to avoid redundancy. For example, in the sidedness tests, lines (of type *CGAL_Line_2<R>*) 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>

Types

typedef CGAL_Point_2<R> Point_2;

typedef CGAL_Line_2<R> Line_2;

CGAL_convex_hull_constructive_traits_2<R>:: Less_xy

CGAL_convex_hull_constructive_traits_2<R>:: Less_yx

CGAL_convex_hull_constructive_traits_2<R>:: Leftturn

CGAL_convex_hull_constructive_traits_2<R>:: Rightturn

CGAL_convex_hull_constructive_traits_2<R>:: Right_of_line

CGAL_convex_hull_constructive_traits_2<R>:: Less_dist_to_line

CGAL_convex_hull_constructive_traits_2<R>:: Less_rotate_ccw

Creation

CGAL_convex_hull_constructive_traits_2<R> ch_traits;

8.3 Convex Hull Traits Classes for LEDA geometry

We provide traits classes for the two-dimensional geometry of LEDA. Note, that the usage of this traits class is not recommended with *CGAL_ch_eddy()* for LEDA versions prior to 3.6. The reason for this is simply, that older versions of LEDA do not provide appropriate efficient predicates used in *CGAL_ch_eddy()*.

8.3.1 Convex Hull Traits Class for LEDA Rational Points (*CGAL_convex_hull_rat_leda_traits_2*)

Definition

The class *CGAL_convex_hull_rat_leda_traits_2* is a convex hull traits class for the two-dimensional rational geometry provided in LEDA, especially *leda_rat_point*s.

#include <*CGAL/convex_hull_rat_leda_traits_2.h*>

Types

typedef leda_rat_point Point_2;

CGAL_convex_hull_rat_leda_traits_2:: Less_xy

CGAL_convex_hull_rat_leda_traits_2:: Less_yx

CGAL_convex_hull_rat_leda_traits_2:: Leftturn

CGAL_convex_hull_rat_leda_traits_2:: Rightturn

CGAL_convex_hull_rat_leda_traits_2:: Right_of_line

CGAL_convex_hull_rat_leda_traits_2:: Less_dist_to_line

CGAL_convex_hull_rat_leda_traits_2:: Less_rotate_ccw

Creation

CGAL_convex_hull_rat_leda_traits_2 ch_traits;

8.3.2 Convex Hull Traits Class for LEDA Points (*CGAL_convex_hull_leda_traits_2*)

Definition

The class *CGAL_convex_hull_leda_traits_2* is a convex hull traits class for the two-dimensional plain geometry provided in LEDA, especially *leda_point*s. Please note, that plain geometry in LEDA is potentially non-robust due to the straightforward use of floating-point arithmetic. For robust, reliable computation use LEDA's rational geometry!

#include <CGAL/convex_hull_leda_traits_2.h>

Types

typedef leda_point Point_2;

CGAL_convex_hull_leda_traits_2:: Less_xy

CGAL_convex_hull_leda_traits_2:: Less_yx

CGAL_convex_hull_leda_traits_2:: Leftturn

CGAL_convex_hull_leda_traits_2:: Rightturn

CGAL_convex_hull_leda_traits_2:: Right_of_line

CGAL_convex_hull_leda_traits_2:: Less_dist_to_line

CGAL_convex_hull_leda_traits_2:: Less_rotate_ccw

Creation

CGAL_convex_hull_leda_traits_2 ch_traits;

8.4 Convex Hulls Traits Class Requirements

The convex hull algorithms in Chapter 8 are parameterized with an traits class *Traits* which defines the primitives (objects and predicates) that the convex hull algorithms use. The following catalog lists the primitives involved in the convex hull and extreme point functions. The primitives that are actually required for a function are listed with the specification of the function in Chapter 8.

Available convex hull traits classes implementations provided by CGAL are described in Section 8.2.

Types

Traits:: Point_2	The point type on which the convex hull functions operate.
Traits:: Less_xy	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 resp.
Traits:: Less_yx	Same as *Less_xy* with the roles of x and y interchanged.
Traits:: Leftturn	Predicate object type. 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.
Traits:: Rightturn	Predicate object type. Must provide *bool operator()(Point_2 p,Point_2 q,Point_2 r)*, which returns *true* iff r lies to the right of the oriented line through p and q.
Traits:: Right_of_line	Unary predicate object type. Must provide a constructor taking two *Point_2* s p and q and *bool operator()(Point_2 r)*, which returns *true* iff r lies right of the directed line through p and q.
Traits:: Less_dist_to_line	Binary predicate object type. Must provide a constructor taking two *Point_2*s p and q and *bool operator()(Point_2 r,Point_2 s)*, which returns *true* iff the signed distance of r to the line l_{pq} through p and q is smaller as the the distance of s to l_{pq}. It is used to compute the point right of a line with maximum unsigned distance to the line. The binary predicate must provide a total order compatible to 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_dist_to_line*.
Traits:: Less_rotate_ccw	Binary predicate object type. Must provide a constructor taking a *Point_2* e and *bool operator()(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 default and copy constructor are required.

Traits ch_traits;	A default constructor.
Traits ch_traits(tbc);	A copy constructor.

Operations

There are no operations in this traits class.

Chapter 9

Geometric Optimisation

Introduction

This chapter describes routines for solving geometric optimisation problems. The first two sections contain algorithms for computing and updating the

- smallest enclosing circle (Section 9.1) and the
- smallest enclosing ellipse (Section 9.2), respectively,

of a finite point set. The remaining sections describe algorithms for searching in matrices with specific properties and some applications. In particular, there are general implementations of

- monotone matrix search (Section 9.6), which can be applied to compute
  - extremal polygons of a convex polygon (Section 9.3) *or*
  - all furthest neighbors for the vertices of a convex polygon (Section 9.4),
- and sorted matrix search (Section 9.7), which can be used to compute the p-centers of a planar point set (Section 9.5).

Traits Class

The class and function templates are parameterized with a traits class which defines the abstract interface between the optimisation algorithm and the primitives it uses. We provide traits class implementations that interface the optimisation algorithms with the CGAL kernel. For some algorithms, in addition, we provide traits class adapters to user supplied point classes. Finally, we describe the requirements that must be fulfilled by traits classes for optimisation algorithms. This is at the same time a specification for using the provided traits class implementations as for users who want to supply their own traits class.

Assertions

The optimisation code uses infix *OPTIMISATION* in the assertions, e.g. defining the compiler flag *CGAL_OPTIMISATION_NO_PRECONDITIONS* switches precondition checking off, cf. Section 2 of the Reference Manual Part 0, General Introduction.

9.1 Smallest Enclosing Circles

This section describes routines for computing and updating the smallest enclosing circle of a finite point set. Formally, the 'smallest enclosing circle' is the boundary of the closed disk of minimum area covering the point set. It is known that this disk is unique. We usually identify the disk with its bounding circle, allowing us to talk about points being on the boundary of the circle, etc.

The algorithm works in an incremental manner. It is implemented as semi-dynamic data structures, thus allowing to insert points while maintaining the smallest enclosing circle.

9.1.1 2D Smallest Enclosing Circle (*CGAL_Min_circle_2<Traits>*)

Definition

An object of the class *CGAL_Min_circle_2<Traits>* is the unique circle of smallest area enclosing a finite set of points in two-dimensional Euclidean plane $\mathbb{E}_2$. 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 $mc(P)$. If $mc(P)$ has more than three points on the boundary, 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.

Note: In this release correct results are only guaranteed if exact arithmetic is used, see Section 9.1.3.

#include <CGAL/Min_circle_2.h>

Traits Class

The template parameter *Traits* is a traits class that defines the abstract interface between the optimisation algorithm and the primitives it uses. For example *Traits::Point* is a mapping on a point class. Think of it as 2D points in the Euclidean plane.

We provide a traits class implementation using the CGAL 2D kernel as described in Section 9.1.3. Traits class adapters to user supplied point classes are available, see Sections 9.1.4 and 9.1.5. Customizing own traits classes for optimisation algorithms can be done according to the requirements for traits classes listed in Section 9.1.8.

Types

CGAL_Min_circle_2<Traits>:: Traits

typedef Traits::Point	*Point;*	Point type.
typedef Traits::Circle	*Circle;*	Circle type.

The following types denote iterators that allow to traverse all points and support points of the smallest enclosing circle, resp. The iterators are non-mutable and their value type is *Point*. The iterator category is given in parentheses.

CGAL_Min_circle_2<Traits>:: Point_iterator (bidirectional).

CGAL_Min_circle_2<Traits>:: Support_point_iterator (random access).

Creation

A *CGAL_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 >
CGAL_Min_circle_2<Traits> min_circle(InputIterator first,
InputIterator last,
bool randomize = false,
CGAL_Random& random = CGAL_random,
Traits traits = Traits())

creates a variable *min_circle* of type *CGAL_Min_circle_2<Traits>*. It is initialized 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).
Precondition: The value type of *first* and *last* is *Point*.

Note: In case a compiler does not support member templates yet, we provide specialized constructors instead. In the current release there are constructors for C arrays (using pointers as iterators), for the STL sequence containers *vector<Point>* and *list<Point>* and for the STL input stream iterator *istream_iterator<Point>*.

CGAL_Min_circle_2<Traits> min_circle(Traits traits = Traits());

creates a variable *min_circle* of type *CGAL_Min_circle_2<Traits>*. It is initialized to $mc(\emptyset)$, the empty set.
Postcondition: *min_circle.is_empty() = true*.

CGAL_Min_circle_2<Traits> min_circle(Point p, Traits traits = Traits());

creates a variable *min_circle* of type *CGAL_Min_circle_2<Traits>*. It is initialized to $mc(\{p\})$, the set $\{p\}$.
Postcondition: *min_circle.is_degenerate() = true*.

CGAL_Min_circle_2<Traits> *min_circle(Point p1, Point p2, Traits traits = Traits());*

creates a variable *min_circle* of type *CGAL_Min_circle_2<Traits>*. It is initialized to $mc(\{p1, p2\})$, the circle with diameter equal to the segment connecting $p1$ and $p2$.

CGAL_Min_circle_2<Traits> *min_circle(Point p1, Point p2, Point p3, Traits traits = Traits());*

creates a variable *min_circle* of type *CGAL_Min_circle_2<Traits>*. It is initialized to $mc(\{p1, p2, p3\})$.

Access Functions

int *min_circle.number_of_points()*

returns the number of points of *min_circle*, i.e. $|P|$.

int *min_circle.number_of_support_points()*

returns the number of support points of *min_circle*, i.e. $|S|$.

Point_iterator *min_circle.points_begin()*
returns an iterator referring to the first point of *min_circle*.

Point_iterator *min_circle.points_end()*
returns the corresponding past-the-end iterator.

Support_point_iterator *min_circle.support_points_begin()*
returns an iterator referring to the first support point of *min_circle*.

Support_point_iterator *min_circle.support_points_end()*
returns the corresponding past-the-end iterator.

Point *min_circle.support_point(int i)*

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()*

returns the current circle of *min_circle*.

Predicates

By definition, an empty *CGAL_Min_circle_2<Traits>* has no boundary and no bounded side, i.e. its unbounded side equals the whole plane $\mathbb{E}_2$.

CGAL_Bounded_side *min_circle.bounded_side(Point p)*

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_circle*, resp.

bool *min_circle.has_on_bounded_side(Point p)*

returns *true*, iff p lies properly inside *min_circle*.

bool *min_circle.has_on_boundary(Point p)*

returns *true*, iff p lies on the boundary of *min_circle*.

bool *min_circle.has_on_unbounded_side(Point p)*

returns *true*, iff p lies properly outside of *min_circle*.

bool *min_circle.is_empty()*

returns *true*, iff *min_circle* is empty (this implies degeneracy).

bool *min_circle.is_degenerate()*

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*).
Precondition: The value type of *first* and *last* is *Point*.

Note: In case a compiler does not support member templates yet, we provide specialized *insert* functions instead. In the current release there are *insert* functions for C arrays (using pointers as iterators), for the STL sequence containers *vector<Point>* and *list<Point>* and for the STL input stream iterator *istream_iterator<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.

Using the traits class implementation for the CGAL kernel with exact arithmetic as described in Section 9.1.3 guarantees validity of *min_circle*. The following function is mainly intended for debugging user supplied traits classes but also for convincing the anxious user that the traits class implementation is correct.

bool *min_circle.is_valid(bool verbose = false, int level = 0)*

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

Traits *min_circle.traits()*

returns a const reference to the traits class object.

I/O

ostream& *ostream& os << min_circle*

writes *min_circle* to output stream *os*.
Precondition: The output operator is defined for *Point* (and for *Circle*, if pretty printing is used).

istream& *istream& is >> & min_circle*

reads *min_circle* from input stream *is*.
Precondition: The input operator is defined for *Point*.

#include <CGAL/IO/Window_stream.h>

CGAL_Window_stream&

CGAL_Window_stream& ws << min_circle

writes *min_circle* to window stream *ws*.
Precondition: The window stream output operator is defined for *Point* and *Circle*.

See Also

CGAL_Min_ellipse_2 (Section 9.2.1), *CGAL_Min_circle_2_traits_2* (Section 9.1.3), *CGAL_Min_circle_2_adapterC2* (Section 9.1.4), *CGAL_Min_circle_2_adapterH2* (Section 9.1.5).

Implementation

We implement the algorithm of Welzl, with move-to-front heuristic [Wel91]. 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 *CGAL_Min_circle_2<Traits>* and to show that randomization can be useful in certain cases, we give an example.

```
#include <CGAL/Gmpz.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Point_2.h>
#include <CGAL/Min_circle_2_traits_2.h>
#include <CGAL/Min_circle_2.h>

typedef  CGAL_Gmpz                      NT;
typedef  CGAL_Homogeneous<NT>           R;
typedef  CGAL_Point_2<R>                Point;
typedef  CGAL_Min_circle_2_traits_2<R>  Traits;
typedef  CGAL_Min_circle_2<Traits>      Min_circle;

int main()
{
    int     n = 1000;
    Point*  P = new Point[ 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);           // very slow
    Min_circle  mc2( P, P+n, true);     // fast

    delete[] P;
    return( 0);
}
```

9.1.2 2D Optimisation Circle (*CGAL_Optimisation_circle_2<R>*)

Definition

An object of the class *CGAL_Optimisation_circle_2<R>* is a circle in the two-dimensional Euclidean plane $\mathbb{E}_2$. Its boundary splits $\mathbb{E}_2$ into a bounded and an unbounded side. Note that the circle can be degenerated, i.e. it can be empty or contain only a single point. By definition, an empty *CGAL_Optimisation_circle_2<R>* has no boundary and no bounded side, i.e. its unbounded side equals the whole plane $\mathbb{E}_2$. A *CGAL_Optimisation_circle_2<R>* containing exactly one point p has no bounded side, its boundary is $\{p\}$, and its unbounded side equals $\mathbb{E}_2 \backslash \{p\}$.

#include <CGAL/Optimisation_circle_2.h>

Types

typedef CGAL_Point_2<R> *Point;* Point type.

typedef R::FT *Distance;* Distance type.

Creation

void *circle.set()*

sets *circle* to the empty circle.

void *circle.set(CGAL_Point_2<R> p)*

sets *circle* to the circle containing exactly p.

void *circle.set(CGAL_Point_2<R> p, CGAL_Point_2<R> q)*

sets *circle* to the circle with diameter $\overline{pq}$.
Precondition: p and q are distinct.

void *circle.set(CGAL_Point_2<R> p,*
CGAL_Point_2<R> q,
CGAL_Point_2<R> r)

sets *circle* to the unique circle through p,q,r.
Precondition: p,q,r are not collinear.

void *circle.set(CGAL_Point_2<R> center, R::FT squared_radius)*

sets *circle* to the circle with center *center* and squared radius *squared_radius*.

Access Functions

CGAL_Point_2<R> *circle.center()*

returns the center of *circle*.

R::FT *circle.squared_radius()*

returns the squared radius of *circle*.

Equality Tests

bool *circle* == *circle2*

returns *true*, iff *circle* and *circle2* are equal, i.e. if they have the same center and same squared radius.

bool *circle* != *circle2*

returns *true*, iff *circle* and *circle2* are not equal.

Predicates

CGAL_Bounded_side *circle.bounded_side(CGAL_Point_2<R> p)*

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(CGAL_Point_2<R> p)*

returns *true*, iff p lies properly inside *circle*.

bool *circle.has_on_boundary(CGAL_Point_2<R> p)*

returns *true*, iff p lies on the boundary of *circle*.

bool *circle.has_on_unbounded_side(CGAL_Point_2<R> p)*

returns *true*, iff p lies properly outside of *circle*.

bool *circle.is_empty()*

returns *true*, iff *circle* is empty (this implies degeneracy).

bool *circle.is_degenerate()*

returns *true*, iff *circle*is degenerate, i.e. if *circle* is empty or equal to a single point.

I/O

ostream& *ostream& os << circle*

writes *circle* to output stream *os*.

istream& *istream& is >> & circle*

reads *circle* from input stream *is*.

#include <CGAL/IO/Window_stream.h>

CGAL_Window_stream&

CGAL_Window_stream& ws << circle

writes *circle* to window stream *ws*.

9.1.3 Traits Class Implementation using the two-dimensional CGAL Kernel (*CGAL_Min_circle_2_traits_2<R>*)

Definition

The class *CGAL_Min_circle_2_traits_2<R>* interfaces the 2D optimisation algorithm for smallest enclosing circles with the CGAL 2D kernel.

#include <CGAL/Min_circle_2_traits_2.h>

Types

typedef CGAL_Point_2<R> *Point;*

typedef CGAL_Optimisation_circle_2<R> *Circle;*

Creation

CGAL_Min_circle_2_traits_2<R> traits;

CGAL_Min_circle_2_traits_2<R> traits(CGAL_Min_circle_2_traits_2<R>);

Operations

CGAL_Orientation traits.orientation(Point p, Point q, Point r)

returns *CGAL_orientation(p, q, r)*.

See Also

CGAL_Min_circle_2 (Section 9.1.1), *CGAL_Min_circle_2_adapterC2* (Section 9.1.4), *CGAL_Min_circle_2_adapterH2* (Section 9.1.5), Requirements of Traits Classes for 2D Smallest Enclosing Circle (Section 9.1.8).

Example

See example for *CGAL_Min_circle_2* (Section 9.1.1).

9.1.4 Traits Class Adapter for 2D Smallest Enclosing Circle to 2D Cartesian Points (*CGAL_Min_circle_2_adapterC2<PT,DA>*)

Definition

The class *CGAL_Min_circle_2_adapterC2<PT,DA>* interfaces the 2D optimisation algorithm for smallest enclosing circles with the point class *PT*. The data accessor *DA* [KW97] is used to access the x- and y-coordinate of *PT*, i.e. *PT* is supposed to have a Cartesian representation of its coordinates.

#include <CGAL/Min_circle_2_adapterC2.h>

Types

CGAL_Min_circle_2_adapterC2<PT,DA>:: DA	Data accessor for Cartesian coordinates.
CGAL_Min_circle_2_adapterC2<PT,DA>:: Point	Point type.
CGAL_Min_circle_2_adapterC2<PT,DA>:: Circle	Circle type.

Creation

CGAL_Min_circle_2_adapterC2<PT,DA> adapter(DA da = DA());

CGAL_Min_circle_2_adapterC2<PT,DA> adapter(CGAL_Min_circle_2_adapterC2<PT,DA>);

Operations

CGAL_Orientation *adapter.orientation(Point p, Point q, Point r)*

returns constants *CGAL_LEFTTURN*, *CGAL_COLLINEAR*, or *CGAL_RIGHTTURN* iff r lies properly to the left, on, or properly to the right of the oriented line through p and q, resp.

See Also

CGAL_Min_circle_2 (Section 9.1.1), *CGAL_Min_circle_2_traits_2* (Section 9.1.3), *CGAL_Min_circle_2_adapterH2* (Section 9.1.5), Requirements of Traits Class Adapters to 2D Cartesian Points (Section 9.1.6).

Example

The following example illustrates the use of the traits class adapters with your own point class. For the sake of simplicity, we expect a point class with Cartesian `double` coordinates and access functions `x()` and `y()`. Based on this we show how to implement and use data accessors.

Note: In this release correct results are only guaranteed if exact arithmetic is used, so this example (using inexact floating-point arithmetic) is only intended to illustrate the techniques.

```
#include <CGAL/Min_circle_2_adapterC2.h>
#include <CGAL/Min_circle_2.h>

// your own point class (Cartesian)
class PtC {
    // ...
  public:
    PtC( double x, double y);
    double  x( ) const;
    double  y( ) const;
    // ...
};

// the data accessor for PtC
class PtC_DA {
  public:
    typedef  double  FT;
    void  get( const PtC& p, double& x, double& y) const {
        x = p.x(); y = p.y();
    }
    double  get_x( const PtC& p) const { return( p.x()); }
    double  get_y( const PtC& p) const { return( p.y()); }
    void  set( PtC& p, double x, double y) const { p = PtC( x, y); }
};

// some typedefs
typedef  CGAL_Min_circle_2_adapterC2 < PtC, PtC_DA >  AdapterC;
typedef  CGAL_Min_circle_2 < AdapterC >               Min_circle;

// do something with Min_circle
Min_circle   mc( /*...*/ );
```

9.1.5 Traits Class Adapter for 2D Smallest Enclosing Circle to 2D Homogeneous Points (*CGAL_Min_circle_2_adapterH2<PT,DA>*)

Definition

The class *CGAL_Min_circle_2_adapterH2<PT,DA>* interfaces the 2D optimisation algorithm for smallest enclosing circles with the point class PT. The data accessor DA [KW97] is used to access the hx-, hy- and hw-coordinate of PT, i.e. PT is supposed to have a homogeneous representation of its coordinates.

#include <CGAL/Min_circle_2_adapterH2.h>

Types

CGAL_Min_circle_2_adapterH2<PT,DA>:: DA	Data accessor for homogeneous coordinates.
CGAL_Min_circle_2_adapterH2<PT,DA>:: Point	Point type.
CGAL_Min_circle_2_adapterH2<PT,DA>:: Circle	Circle type.

Creation

CGAL_Min_circle_2_adapterH2<PT,DA> adapter(DA da = DA());

CGAL_Min_circle_2_adapterH2<PT,DA> adapter(CGAL_Min_circle_2_adapterH2<PT,DA>);

Operations

CGAL_Orientation *adapter.orientation(Point p, Point q, Point r)*

returns constants *CGAL_LEFTTURN*, *CGAL_COLLINEAR*, or *CGAL_RIGHTTURN* iff r lies properly to the left, on, or properly to the right of the oriented line through p and q, resp.

See Also

CGAL_Min_circle_2 (Section 9.1.1), *CGAL_Min_circle_2_traits_2* (Section 9.1.3), *CGAL_Min_circle_2_adapterC2* (Section 9.1.4), Requirements of Traits Class Adapters to 2D Homogeneous Points (Section 9.1.7).

Example

The following example illustrates the use of the traits class adapters with your own point class. For the sake of simplicity, we expect a point class with homogeneous `int` coordinates and access functions `hx()`, `hy()`, and `hw()`. Based on this we show how to implement and use data accessors.

Note: In this release correct results are only guaranteed if exact arithmetic is used, so this example (using integer arithmetic with possible overflows) is only intended to illustrate the techniques.

```
#include <CGAL/Min_circle_2_adapterH2.h>
#include <CGAL/Min_circle_2.h>

// your own point class (homogeneous)
class PtH {
    // ...
  public:
    PtH( int hx, int hy, int hw);
    int  hx( ) const;
    int  hy( ) const;
    int  hw( ) const;
    // ...
};

// the data accessor for PtH
class PtH_DA {
  public:
    typedef  int  RT;
    void  get( const PtH& p, int& hx, int& hy, int& hw) const {
        hx = p.hx(); hy = p.hy(); hw.phw();
    }
    int  get_x( const PtH& p) const { return( p.hx()); }
    int  get_y( const PtH& p) const { return( p.hy()); }
    int  get_w( const PtH& p) const { return( p.hw()); }
    void  set( PtH& p, int hx, int hy, int hw) const { p = PtH( hx, hy, hw); }
};

// some typedefs
typedef  CGAL_Min_circle_2_adapterH2< PtH, PtH_DA >  AdapterH;
typedef  CGAL_Min_circle_2< AdapterH >               Min_circle;

// do something with Min_circle
Min_circle   mc( /*...*/ );
```

9.1.6 Requirements of Traits Class Adapters to 2D Cartesian Points

The family of traits class adapters *..._adapterC2* to 2D Cartesian points is parameterized with a point type PT and a data accessor DA [KW97]. The latter defines the coordinates-based interface between the traits class adapter and the point type. The following requirements catalog lists the primitives, i.e. types, member functions etc., that must be defined for classes PT and DA that can be used to parameterize *..._adapterC2*.

Point Type (PT)

$PT\ \ p;$		Default constructor.
$PT\ \ p(\ PT);$		Copy constructor.
PT&	$p = q$	Assignment.
bool	$p == q$	Equality test.

The following I/O operators are only needed, if the corresponding I/O operators of the optimisation algorithm are used.

ostream&	*ostream*& $os << p$	writes p to output stream os.

istream&	*istream& is >> & p*	reads p from input stream is.

Read/Write Data Accessor (DA)

DA:: FT		The number type FT has to fulfill the requirements of a CGAL field type.
DA da;		Default constructor.
DA da(DA);		Copy constructor.
void	*da.get(Point p, FT& x, FT& y)*	returns the Cartesian coordinates of p in x and y, resp.
FT	*da.get_x(Point p)*	returns the Cartesian x-coordinate of p.
FT	*da.get_y(Point p)*	returns the Cartesian y-coordinate of p.
void	*da.set(Point& p, FT x, FT y)*	sets p to the point with Cartesian coordinates x and y.

9.1.7 Requirements of Traits Class Adapters to 2D Homogeneous Points

The family of traits class adapters *..._adapterH2* to 2D homogeneous points is parameterized with a point type PT and a data accessor DA [KW97]. The latter defines the coordinates-based interface between the traits class adapter and the point type. The following requirements catalog lists the primitives, i.e. types, member functions etc., that must be defined for classes PT and DA that can be used to parameterize *..._adapterH2*.

Point Type (PT)

PT p;		Default constructor.
PT p(PT);		Copy constructor.
PT&	$p = q$	Assignment.
bool	$p == q$	Equality test.

The following I/O operators are only needed, if the corresponding I/O operators of the optimisation algorithm are used.

ostream&	*ostream& os << p*	writes p to output stream os.
istream&	*istream& is >> & p*	reads p from input stream is.

Read/Write Data Accessor (*DA*)

DA:: RT		The number type RT has to fulfill the requirements of a CGAL ring type.
DA da;		Default constructor.
DA da(DA);		Copy constructor.
void	*da.get(Point p, RT& hx, RT& hy, RT& hw)*	returns the homogeneous coordinates of p in hx, hy and hw, resp.
RT	*da.get_hx(Point p)*	returns the homogeneous x-coordinate of p.
RT	*da.get_hy(Point p)*	returns the homogeneous y-coordinate of p.
RT	*da.get_hw(Point p)*	returns the homogeneous w-coordinate of p.
void	*da.set(Point& p, RT hx, RT hy, RT hw)*	sets p to the point with homogeneous coordinates hx, hy and hw.

9.1.8 Requirements of Traits Classes for 2D Smallest Enclosing Circle

The class template *CGAL_Min_circle_2* is parameterized with a *Traits* class which defines the abstract interface between the optimisation algorithm and the primitives it uses. The following requirements catalog lists the primitives, i.e. types, member functions etc., that must be defined for a class that can be used to parameterize *CGAL_Min_circle_2*. A traits class implementation using the CGAL 2D kernel is available and described in Section 9.1.3. In addition, we provide traits class adapters to user supplied point classes, see Sections 9.1.4 and 9.1.5. Both, the implementation and the adapters, can be used as a starting point for customizing own traits classes, e.g. through derivation and specialization.

Traits Class (*Traits*)

Definition

A class that satisfies the requirements of a traits class for *CGAL_Min_circle_2* must provide the following primitives.

Types

Traits:: Point	The point type must provide default and copy constructor, assignment and equality test.
Traits:: Circle	The circle type must fulfill the requirements listed below in the next section.

In addition, if I/O is used, the corresponding I/O operators for *Point* and *Circle* have to be provided, see topic **I/O** in Section 9.1.1.

Variables

Circle	*circle;*	The actual 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. Note that further constructors can be provided.

Traits traits;	A default constructor.
Traits traits(Traits);	A copy constructor.

Operations

The following predicate is only needed, if the member function *is_valid* of *CGAL_Min_circle_2* is used.

CGAL_Orientation *traits.orientation(Point p, Point q, Point r)*

returns constants *CGAL_LEFTTURN*, *CGAL_COLLINEAR*, or *CGAL_RIGHTTURN* iff r lies properly to the left, on, or properly to the right of the oriented line through p and q, resp.

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 $\mathbb{E}_2 \backslash \{p\}$.

Types

Circle:: Point	Point type.

The following type is only needed, if the member function *is_valid* of *CGAL_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 $\overline{pq}$. 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)*

returns *true*, iff p lies properly outside of *circle*.

Each of the following predicates is only needed, if the corresponding predicate of *CGAL_Min_circle_2* is used.

CGAL_Bounded_side *circle.bounded_side(Point p)*

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)*

returns *true*, iff p lies properly inside *circle*.

bool *circle.has_on_boundary(Point p)*

returns *true*, iff p lies on the boundary of *circle*.

bool *circle.is_empty()*

returns *true*, iff *circle* is empty (this implies degeneracy).

bool *circle.is_degenerate()*

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 *CGAL_Min_circle_2* is used.

bool *circle* == *circle2*

returns *true*, iff *circle* and *circle2* are equal.

Point *circle.center()*

returns the center of *circle*.

Distance *circle.squared_radius()*

returns the squared radius of *circle*.

I/O

The following I/O operators are only needed, if the corresponding I/O operators of *CGAL_Min_circle_2* are used.

ostream& *ostream& os* << *circle*

writes *circle* to output stream *os*.

istream& *istream& is* >> & *circle*

reads *circle* from input stream *is*.

CGAL_Window_stream&

CGAL_Window_stream& ws << *circle*

writes *circle* to window stream *ws*.

9.2 Smallest Enclosing Ellipses

This section describes routines for computing and updating the smallest enclosing ellipse of a finite point set. Formally, the 'smallest enclosing ellipse' is the boundary of the closed disk of minimum area covering the point set. It is known that this disk is unique. We usually identify the disk with its bounding ellipse, allowing us to talk about points being on the boundary of the ellipse, etc.

The algorithm works in an incremental manner. It is implemented as semi-dynamic data structures, thus allowing to insert points while maintaining the smallest enclosing ellipse.

9.2.1 2D Smallest Enclosing Ellipse (*CGAL_Min_ellipse_2<Traits>*)

Definition

An object of the class *CGAL_Min_ellipse_2<Traits>* is the unique ellipse of smallest area enclosing a finite 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 $me(P)$. If $me(P)$ has more than five points on the boundary, 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.

Note: In this release correct results are only guaranteed if exact arithmetic is used, see Section 9.2.3.

#include <CGAL/Min_ellipse_2.h>

Traits Class

The template parameter *Traits* is a traits class that defines the abstract interface between the optimisation algorithm and the primitives it uses. For example *Traits::Point* is a mapping on a point class. Think of it as 2D points in the Euclidean plane.

We provide a traits class implementation using the CGAL 2D kernel as described in Section 9.2.3. Traits class adapters to user supplied point classes are available, see Sections 9.2.4 and 9.2.5. Customizing own traits classes for optimisation algorithms can be done according to the requirements for traits classes listed in Section 9.2.8.

Types

CGAL_Min_ellipse_2<Traits>:: Traits

typedef Traits::Point	*Point;*	Point type.
typedef Traits::Ellipse	*Ellipse;*	Ellipse type.

The following types denote iterators that allow to traverse all points and support points of the smallest enclosing ellipse, resp. The iterators are non-mutable and their value type is *Point*. The iterator category is given in parentheses.

CGAL_Min_ellipse_2<Traits>:: Point_iterator (bidirectional).

CGAL_Min_ellipse_2<Traits>:: Support_point_iterator (random access).

Creation

A *CGAL_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 >
CGAL_Min_ellipse_2<Traits> min_ellipse(InputIterator first,
InputIterator last,
bool randomize = false,
CGAL_Random& random = CGAL_random,
Traits traits = Traits())

creates a variable *min_ellipse* of type *CGAL_Min_ellipse_2<Traits>*. It is initialized 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).
Precondition: The value type of *first* and *last* is *Point*.

Note: In case a compiler does not support member templates yet, we provide specialized constructors instead. In the current release there are constructors for C arrays (using pointers as iterators), for the STL sequence containers *vector<Point>* and *list<Point>* and for the STL input stream iterator *istream_iterator<Point>*.

CGAL_Min_ellipse_2<Traits> min_ellipse(Traits traits = Traits());

creates a variable *min_ellipse* of type *CGAL_Min_ellipse_2<Traits>*. It is initialized to $me(\emptyset)$, the empty set.
Postcondition: *min_ellipse.is_empty() = true*.

CGAL_Min_ellipse_2<Traits> min_ellipse(Point p, Traits traits = Traits());

creates a variable *min_ellipse* of type *CGAL_Min_ellipse_2<Traits>*. It is initialized to $me(\{p\})$, the set $\{p\}$.
Postcondition: *min_ellipse.is_degenerate() = true*.

CGAL_Min_ellipse_2<Traits> min_ellipse(Point p, Point q, Traits traits = Traits());

creates a variable *min_ellipse* of type *CGAL_Min_ellipse_2<Traits>*. It is initialized to $me(\{p,q\})$, the set $\{(1-\lambda)p + \lambda q \mid 0 \leq \lambda \leq 1\}$.
Postcondition: *min_ellipse.is_degenerate() = true*.

CGAL_Min_ellipse_2<Traits> *min_ellipse(Point p1, Point p2, Point p3, Traits traits = Traits());*

creates a variable *min_ellipse* of type *CGAL_Min_ellipse_2<Traits>*. It is initialized to $me(\{p1, p2, p3\})$.

CGAL_Min_ellipse_2<Traits> *min_ellipse(Point p1,*
Point p2,
Point p3,
Point p4,
Traits traits = Traits())

creates a variable *min_ellipse* of type *CGAL_Min_ellipse_2<Traits>*. It is initialized to $me(\{p1, p2, p3, p4\})$.

CGAL_Min_ellipse_2<Traits> *min_ellipse(Point p1,*
Point p2,
Point p3,
Point p4,
Point p5,
Traits traits = Traits())

creates a variable *min_ellipse* of type *CGAL_Min_ellipse_2<Traits>*. It is initialized to $me(\{p1, p2, p3, p4, p5\})$.

Access Functions

int *min_ellipse.number_of_points()*

returns the number of points of *min_ellipse*, i.e. $|P|$.

int *min_ellipse.number_of_support_points()*

returns the number of support points of *min_ellipse*, i.e. $|S|$.

Point_iterator *min_ellipse.points_begin()*
returns an iterator referring to the first point of *min_ellipse*.

Point_iterator *min_ellipse.points_end()*
returns the corresponding past-the-end iterator.

Support_point_iterator *min_ellipse.support_points_begin()*
returns an iterator referring to the first support point of *min_ellipse*.

Support_point_iterator *min_ellipse.support_points_end()*
returns the corresponding past-the-end iterator.

Point *min_ellipse.support_point(int i)*

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()*

returns the current ellipse of *min_ellipse*.

Predicates

By definition, an empty *CGAL_Min_ellipse_2<Traits>* has no boundary and no bounded side, i.e. its unbounded side equals the whole plane $\mathbb{E}_2$.

CGAL_Bounded_side *min_ellipse.bounded_side(Point p)*

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_ellipse*, resp.

bool *min_ellipse.has_on_bounded_side(Point p)*

returns *true*, iff p lies properly inside *min_ellipse*.

bool *min_ellipse.has_on_boundary(Point p)*

returns *true*, iff p lies on the boundary of *min_ellipse*.

bool *min_ellipse.has_on_unbounded_side(Point p)*

returns *true*, iff p lies outside of *min_ellipse*.

bool *min_ellipse.is_empty()*

returns *true*, iff *min_ellipse* is empty (this implies degeneracy).

bool *min_ellipse.is_degenerate()*

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*).
Precondition: The value type of *first* and *last* is *Point*.

Note: In case a compiler does not support member templates yet, we provide specialized *insert* functions instead. In the current release there are *insert* functions for C arrays (using pointers as iterators), for the STL sequence containers *vector<Point>* and *list<Point>* and for the STL input stream iterator *istream_iterator<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.

Using the traits class implementation for the CGAL kernel with exact arithmetic as described in Section 9.2.3 guarantees validity of *min_ellipse*. The following function is mainly intended for debugging user supplied traits classes but also for convincing the anxious user that the traits class implementation is correct.

bool *min_ellipse.is_valid(bool verbose = false, int level = 0)*

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

Traits *min_ellipse.traits()*

returns a const reference to the traits class object.

I/O

ostream& *ostream& os << min_ellipse*

writes *min_ellipse* to output stream *os*.
Precondition: The output operator is defined for *Point* (and for *Ellipse*, if pretty printing is used).

istream& *istream& is >> & min_ellipse*

reads *min_ellipse* from input stream *is*.
Precondition: The input operator is defined for *Point*.

#include <CGAL/IO/Window_stream.h>

CGAL_Window_stream&

CGAL_Window_stream& ws << min_ellipse

writes *min_ellipse* to window stream *ws*.
Precondition: The window stream output operator is defined for *Point* and *Ellipse*.

See Also

CGAL_Min_circle_2 (Section 9.1.1), *CGAL_Min_ellipse_2_traits_2* (Section 9.2.3), *CGAL_Min_ellipse_2_adapterC2* (Section 9.2.4), *CGAL_Min_ellipse_2_adapterH2* (Section 9.2.5).

Implementation

We implement the algorithm of Welzl, with move-to-front heuristic [Wel91], using the primitives as described in [GS97a, GS97b]. 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 creation of *CGAL_Min_ellipse_2<Traits>*and to show that randomization can be useful in certain cases, we give an example.

```
#include <CGAL/Gmpz.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Point_2.h>
#include <CGAL/Min_ellipse_2_traits_2.h>
#include <CGAL/Min_ellipse_2.h>

typedef  CGAL_Gmpz                      NT;
typedef  CGAL_Homogeneous<NT>           R;
typedef  CGAL_Point_2<R>                Point;
```

```
typedef  CGAL_Min_ellipse_2_traits_2<R>  Traits;
typedef  CGAL_Min_ellipse_2<Traits>      Min_ellipse;

int main()
{
    int     n = 1000;
    Point*  P = new Point[ 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_ellipse  me1( P, P+n);          // very slow
    Min_ellipse  me2( P, P+n, true);    // fast

    delete[] P;
    return( 0);
}
```

9.2.2 2D Optimisation Ellipse (*CGAL_Optimisation_ellipse_2<R>*)

Definition

An object of the class *CGAL_Optimisation_ellipse_2<R>* is an ellipse in the two-dimensional Euclidean plane $\mathbb{E}_2$. Its boundary splits $\mathbb{E}_2$ into a bounded and an unbounded side. Note that the ellipse can be degenerated, i.e. it can be empty, equal to a single point or equal to a segment. By definition, an empty *CGAL_Optimisation_ellipse_2<R>* has no boundary and no bounded side, i.e. its unbounded side equals the whole plane $\mathbb{E}_2$. A *CGAL_Optimisation_ellipse_2<R>* equal to a single point p or equal to a segment s, resp., has no bounded side, its boundary is $\{p\}$ or s, resp., and its unbounded side equals $\mathbb{E}_2\backslash\{p\}$ or $\mathbb{E}_2\backslash s$, resp.

#include <CGAL/Optimisation_ellipse_2.h>

Types

typedef CGAL_Point_2<R> Point; Point type.

Creation

void *ellipse.set()*

sets *ellipse* to the empty ellipse.

void *ellipse.set(CGAL_Point_2<R> p)*

sets *ellipse* to the ellipse equal to the single point p.

void *ellipse.set(CGAL_Point_2<R> p, CGAL_Point_2<R> q)*

sets *ellipse* to the ellipse equal to the segment $\overline{pq}$.
Precondition: p and q are distinct.

void *ellipse.set(CGAL_Point_2<R> p1, CGAL_Point_2<R> p2, CGAL_Point_2<R> p3)*

sets *ellipse* to the ellipse of smallest area through *p1*, *p2* and *p3*. *Precondition*: *p1*,*p2*,*p3* are not collinear.

void *ellipse.set(CGAL_Point_2<R> p1, CGAL_Point_2<R> p2, CGAL_Point_2<R> p3, CGAL_Point_2<R> p4)*

sets *ellipse* to the ellipse of smallest area through *p1*, *p2*, *p3* and *p4*. *Precondition*: *p1*,*p2*,*p3*,*p4* are in convex position.

void *ellipse.set(CGAL_Point_2<R> p1, CGAL_Point_2<R> p2, CGAL_Point_2<R> p3, CGAL_Point_2<R> p4, CGAL_Point_2<R> p5)*

sets *ellipse* to the unique ellipse through *p1*, *p2*, *p3*, *p4* and *p5*. *Precondition*: There exists an ellipse through *p1*,*p2*,*p3*,*p4*,*p5*.

Access Functions

Equality Tests

bool *ellipse* == *ellipse2*

returns *true*, iff *ellipse* and *ellipse2* are equal.

bool *ellipse* != *ellipse2*

returns *true*, iff *ellipse* and *ellipse2* are not equal.

Predicates

CGAL_Bounded_side *ellipse.bounded_side(CGAL_Point_2<R> p)*

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(CGAL_Point_2<R> p)*

returns *true*, iff *p* lies properly inside *ellipse*.

bool *ellipse.has_on_boundary(CGAL_Point_2<R> p)*

returns *true*, iff p lies on the boundary of *ellipse*.

bool *ellipse.has_on_unbounded_side(CGAL_Point_2<R> p)*

returns *true*, iff p lies properly outside of *ellipse*.

bool *ellipse.is_empty()*

returns *true*, iff *ellipse* is empty (this implies degeneracy).

bool *ellipse.is_degenerate()*

returns *true*, iff *ellipse*is degenerate, i.e. if *ellipse* is empty, equal to a single point or equal to a segment.

I/O

ostream& *ostream& os << ellipse*

writes *ellipse* to output stream *os*.

istream& *istream& is >> & ellipse*

reads *ellipse* from input stream *is*.

#include <CGAL/IO/Window_stream.h>

CGAL_Window_stream&

CGAL_Window_stream& ws << ellipse

writes *ellipse* to window stream *ws*.

9.2.3 Traits Class Implementation using the two-dimensional CGAL Kernel (*CGAL_Min_ellipse_2_traits_2<R>*)

Definition

The class *CGAL_Min_ellipse_2_traits_2<R>* interfaces the 2D optimisation algorithm for smallest enclsoing ellipses with the CGAL 2D kernel.

#include <CGAL/Min_ellipse_2_traits_2.h>

Types

typedef CGAL_Point_2<R> *Point;*

typedef CGAL_Optimisation_ellipse_2<R> *Ellipse;*

Creation

CGAL_Min_ellipse_2_traits_2<R> traits;

CGAL_Min_ellipse_2_traits_2<R> traits(CGAL_Min_ellipse_2_traits_2<R>);

See Also

CGAL_Min_ellipse_2 (Section 9.2.1), *CGAL_Min_ellipse_2_adapterC2* (Section 9.2.4), *CGAL_Min_ellipse_2_adapterH2* (Section 9.2.5), Requirements of Traits Classes for 2D Smallest Enclosing Ellipse (Section 9.2.8).

Example

See example for *CGAL_Min_ellipse_2* (Section 9.2.1).

9.2.4 Traits Class Adapter for 2D Smallest Enclosing Ellipse to 2D Cartesian Points (*CGAL_Min_ellipse_2_adapterC2<PT,DA>*)

Definition

The class *CGAL_Min_ellipse_2_adapterC2<PT,DA>* interfaces the 2D optimisation algorithm for smallest enclosing ellipses with the point class PT. The data accessor DA [KW97] is used to access the x- and y-coordinate of PT, i.e. PT is supposed to have a Cartesian representation of its coordinates.

#include <CGAL/Min_ellipse_2_adapterC2.h>

Types

CGAL_Min_ellipse_2_adapterC2<PT,DA>:: DA	Data accessor for Cartesian coordinates.
CGAL_Min_ellipse_2_adapterC2<PT,DA>:: Point	Point type.
CGAL_Min_ellipse_2_adapterC2<PT,DA>:: Ellipse	Ellipse type.

Creation

CGAL_Min_ellipse_2_adapterC2<PT,DA> adapter(DA da = DA());

CGAL_Min_ellipse_2_adapterC2<PT,DA> adapter(CGAL_Min_ellipse_2_adapterC2<PT,DA>);

See Also

CGAL_Min_ellipse_2 (Section 9.2.1), *CGAL_Min_ellipse_2_traits_2* (Section 9.2.3), *CGAL_Min_ellipse_2_adapterH2* (Section 9.2.5), Requirements of Traits Class Adapters to 2D Cartesian Points (Section 9.2.6).

Example

The following example illustrates the use of the traits class adapters with your own point class. For the sake of simplicity, we expect a point class with Cartesian `double` coordinates and access functions `x()` and `y()`. Based on this we show how to implement and use data accessors.

Note: In this release correct results are only guaranteed if exact arithmetic is used, so this example (using inexact floating-point arithmetic) is only intended to illustrate the techniques.

```
#include <CGAL/Min_ellipse_2_adapterC2.h>
#include <CGAL/Min_ellipse_2.h>

// your own point class (Cartesian)
class PtC {
    // ...
  public:
    PtC( double x, double y);
    double  x( ) const;
    double  y( ) const;
    // ...
};

// the data accessor for PtC
class PtC_DA {
  public:
    typedef  double  FT;
    void  get( const PtC& p, double& x, double& y) const {
        x = p.x(); y = p.y();
    }
    double  get_x( const PtC& p) const { return( p.x()); }
    double  get_y( const PtC& p) const { return( p.y()); }
    void  set( PtC& p, double x, double y) const { p = PtC( x, y); }
};

// some typedefs
typedef  CGAL_Min_ellipse_2_adapterC2 < PtC, PtC_DA >  AdapterC;
typedef  CGAL_Min_ellipse_2 < AdapterC >               Min_ellipse;

// do something with Min_ellipse
Min_ellipse   me( /*...*/ );
```

9.2.5 Traits Class Adapter for 2D Smallest Enclosing Ellipse to 2D Homogeneous Points (*CGAL_Min_ellipse_2_adapterH2<PT,DA>*)

Definition

The class *CGAL_Min_ellipse_2_adapterH2<PT,DA>* interfaces the 2D optimisation algorithm for smallest enclosing ellipses with the point class PT. The data accessor DA [KW97] is used to access the hx-,

hy- and hw-coordinate of PT, i.e. PT is supposed to have a homogeneous representation of its coordinates.

#include <CGAL/Min_ellipse_2_adapterH2.h>

Types

CGAL_Min_ellipse_2_adapterH2<PT,DA>:: DA	Data accessor for homogeneous coordinates.
CGAL_Min_ellipse_2_adapterH2<PT,DA>:: Point	Point type.
CGAL_Min_ellipse_2_adapterH2<PT,DA>:: Ellipse	Ellipse type.

Creation

CGAL_Min_ellipse_2_adapterH2<PT,DA> adapter(DA da = DA());

CGAL_Min_ellipse_2_adapterH2<PT,DA> adapter(CGAL_Min_ellipse_2_adapterH2<PT,DA>);

See Also

CGAL_Min_ellipse_2 (Section 9.2.1), *CGAL_Min_ellipse_2_traits_2* (Section 9.2.3), *CGAL_Min_ellipse_2_adapterC2* (Section 9.2.4), Requirements of Traits Class Adapters to 2D Homogeneous Points (Section 9.2.7).

Example

The following example illustrates the use of the traits class adapters with your own point class. For the sake of simplicity, we expect a point class with homogeneous `int` coordinates and access functions `hx()`, `hy()`, and `hw()`. Based on this we show how to implement and use data accessors.

Note: In this release correct results are only guaranteed if exact arithmetic is used, so this example (using integer arithmetic with possible overflows) is only intended to illustrate the techniques.

```
#include <CGAL/Min_ellipse_2_adapterH2.h>
#include <CGAL/Min_ellipse_2.h>

// your own point class (homogeneous)
class PtH {
    // ...
  public:
    PtH( int hx, int hy, int hw);
    int  hx( ) const;
    int  hy( ) const;
    int  hw( ) const;
    // ...
};

// the data accessor for PtH
class PtH_DA {
```

```
  public:
    typedef  int  RT;
    void  get( const PtH& p, int& hx, int& hy, int& hw) const {
        hx = p.hx(); hy = p.hy(); hw.phw();
    }
    int  get_x( const PtH& p) const { return( p.hx()); }
    int  get_y( const PtH& p) const { return( p.hy()); }
    int  get_w( const PtH& p) const { return( p.hw()); }
    void  set( PtH& p, int hx, int hy, int hw) const { p = PtH( hx, hy, hw); }
};

// some typedefs
typedef  CGAL_Min_ellipse_2_adapterH2< PtH, PtH_DA >  AdapterH;
typedef  CGAL_Min_ellipse_2< AdapterH >               Min_ellipse;

// do something with Min_ellipse
Min_ellipse   me( /*...*/ );
```

9.2.6 Requirements of Traits Class Adapters to 2D Cartesian Points

The family of traits class adapters *..._adapterC2* to 2D Cartesian points is parameterized with a point type PT and a data accessor DA [KW97]. The latter defines the coordinates-based interface between the traits class adapter and the point type. The following requirements catalog lists the primitives, i.e. types, member functions etc., that must be defined for classes PT and DA that can be used to parameterize *..._adapterC2*.

Point Type (PT)

PT p;		Default constructor.
PT $p(PT)$;		Copy constructor.
PT&	$p = q$	Assignment.
bool	$p == q$	Equality test.

The following I/O operators are only needed, if the corresponding I/O operators of the optimisation algorithm are used.

ostream&	*ostream*& $os << p$	writes p to output stream os.
istream&	*istream*& $is >>$ & p	reads p from input stream is.

Read/Write Data Accessor (DA)

$DA::FT$		The number type FT has to fulfill the requirements of a CGAL field type.
DA da;		Default constructor.
DA $da(DA)$;		Copy constructor.
void	*da.get(Point p, FT*& *x, FT*& *y)*	returns the Cartesian coordinates of p in x and y, resp.

FT	*da.get_x(Point p)*	returns the Cartesian x-coordinate of p.
FT	*da.get_y(Point p)*	returns the Cartesian y-coordinate of p.
void	*da.set(Point& p, FT x, FT y)*	sets p to the point with Cartesian coordinates x and y.

9.2.7 Requirements of Traits Class Adapters to 2D Homogeneous Points

The family of traits class adapters *..._adapterH2* to 2D homogeneous points is parameterized with a point type PT and a data accessor DA [KW97]. The latter defines the coordinates-based interface between the traits class adapter and the point type. The following requirements catalog lists the primitives, i.e. types, member functions etc., that must be defined for classes PT and DA that can be used to parameterize *..._adapterH2*.

Point Type (PT)

PT p;		Default constructor.
PT p(PT);		Copy constructor.
PT&	$p = q$	Assignment.
bool	$p == q$	Equality test.

The following I/O operators are only needed, if the corresponding I/O operators of the optimisation algorithm are used.

ostream&	*ostream& os* $<<$ p	writes p to output stream *os*.
istream&	*istream& is* $>>$ *&* p	reads p from input stream *is*.

Read/Write Data Accessor (DA)

DA:: RT		The number type RT has to fulfill the requirements of a CGAL ring type.
DA da;		Default constructor.
DA da(DA);		Copy constructor.
void	*da.get(Point p, RT& hx, RT& hy, RT& hw)*	returns the homogeneous coordinates of p in hx, hy and hw, resp.
RT	*da.get_hx(Point p)*	returns the homogeneous x-coordinate of p.
RT	*da.get_hy(Point p)*	returns the homogeneous y-coordinate of p.
RT	*da.get_hw(Point p)*	returns the homogeneous w-coordinate of p.

void	*da.set(Point& p, RT hx, RT hy, RT hw)*
	sets p to the point with homogeneous coordinates hx, hy and hw.

9.2.8 Requirements of Traits Classes for 2D Smallest Enclosing Ellipse

The class template *CGAL_Min_ellipse_2* is parameterized with a *Traits* class which defines the abstract interface between the optimisation algorithm and the primitives it uses. The following requirements catalog lists the primitives, i.e. types, member functions etc., that must be defined for a class that can be used to parameterize *CGAL_Min_ellipse_2*. A traits class implementation using the CGAL 2D kernel is available and described in Section 9.2.3. In addition, we provide traits class adapters to user supplied point classes, see Sections 9.2.4 and 9.2.5. Both, the implementation and the adapters, can be used as a starting point for customizing own traits classes, e.g. through derivation and specialization.

Traits Class (*Traits*)

Definition

A class that satisfies the requirements of a traits class for *CGAL_Min_ellipse_2* must provide the following primitives.

Types

Traits:: Point	The point type must provide default and copy constructor, assignment and equality test.
Traits:: Ellipse	The ellipse type must fulfill the requirements listed below in the next section.

In addition, if I/O is used, the corresponding I/O operators for *Point* and *Ellipse* have to be provided, see topic **I/O** in Section 9.2.1.

Variables

Ellipse	*ellipse;*	The actual 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. Note that further constructors can be provided.

Traits traits;	A default constructor.
Traits traits(Traits);	A copy constructor.

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 $\overline{pq}$. 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)*

returns *true*, iff p lies properly outside of *ellipse*.

Each of the following predicates is only needed, if the corresponding predicate of *CGAL_Min_ellipse_2* is used.

CGAL_Bounded_side *ellipse.bounded_side(Point p)*

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)*

returns *true*, iff p lies properly inside *ellipse*.

bool *ellipse.has_on_boundary(Point p)*

returns *true*, iff p lies on the boundary of *ellipse*.

bool *ellipse.is_empty()*

returns *true*, iff *ellipse* is empty (this implies degeneracy).

bool *ellipse.is_degenerate()*

returns *true*, iff *ellipse* is degenerate, i.e. if *ellipse* is empty or equal to a single point.

I/O

The following I/O operators are only needed, if the corresponding I/O operators of *CGAL_Min_ellipse_2* are used.

ostream& *ostream& os << ellipse*

writes *ellipse* to output stream *os*.

istream& *istream& is >> & ellipse*

reads *ellipse* from input stream *is*.

CGAL_Window_stream&

CGAL_Window_stream& ws << ellipse

writes *ellipse* to window stream *ws*.

9.3 Computing Extremal Polygons

This section describes several functions to compute a maximal k-gon P_k that can be inscribed into a given convex polygon P. The criterion for maximality can be chosen freely by defining an appropriate traits class as specified in section 9.3.4. For CGAL point classes there are two predefined traits classes to compute a maximum area (see section 9.3.1) resp. perimeter (see section 9.3.2) inscribed k-gon.

9.3.1 Computing a Maximum Area Inscribed k-gon

This section describes a function to compute a maximal 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.

#include <CGAL/extremal_polygon_2.h>

```
template < class RandomAccessIC, class OutputIterator >
OutputIterator          CGAL_maximum_area_inscribed_k_gon(
                                RandomAccessIC points_begin,
                                RandomAccessIC 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. Value type of *RandomAccessIC* has to be *CGAL_Point_2<R>* for some representation class R.
2. *OutputIterator* accepts the value type of *RandomAccessIC* as value type,
3. the – at least three – points denoted by the range [*points_begin*, *points_end*) form the boundary of a convex polygon (oriented clock– or counterclockwise) *and*
4. $k \geq 3$.

Note

On compilers not supporting member function templates, the parameter *RandomAccessIC* is fixed to *vector<Point_2>::iterator* where *Point_2* is the value type of *RandomAccessIC*.

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/Point_2.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.h>
#include <vector.h>

int main() {

  typedef double                              FT;
  typedef CGAL_Cartesian< FT >                R;
  typedef CGAL_Point_2< R >                   Point_2;
  typedef CGAL_Polygon_traits_2< R >          P_traits;
  typedef vector< Point_2 >                   Cont;
  typedef CGAL_Polygon_2< P_traits, Cont >    Polygon_2;
  typedef CGAL_Random_points_in_square_2<
    Point_2,
    CGAL_Creator_uniform_2< FT, Point_2 > >
  Point_generator;

  Polygon_2 p;
  int number_of_points( 10);
  int k( 5);

  CGAL_random_convex_set_2( number_of_points,
                            back_inserter( p),
                            Point_generator( 1));
  cout << "Generated Polygon:\n" << p << endl;

  Polygon_2 k_gon;
  CGAL_maximum_area_inscribed_k_gon(
    p.vertices_begin(),
    p.vertices_end(),
    k,
    back_inserter( k_gon));
  cout << "Maximum area " << k << "-gon:\n" << k_gon << endl;

}
```

9.3.2 Computing a Maximum Perimeter Inscribed k-gon

This section describes a function to compute a largest perimeter k-gon P_k that can be inscribed in a given convex polygon P. Note that P_k is not unique in general, but we know that its vertices form a subset of the vertex set of P.

#include <CGAL/extremal_polygon_2.h>

template < class RandomAccessIC, class OutputIterator >
OutputIterator CGAL_maximum_perimeter_inscribed_k_gon(
RandomAccessIC points_begin,
RandomAccessIC 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. Value type of *RandomAccessIC* has to be *CGAL_Point_2<R>* for some representation class R,
2. there is a global function *R::FT CGAL_sqrt(R::FT)* defined that computes the squareroot of a number,
3. *OutputIterator* accepts the value type of *RandomAccessIC* as value type,
4. the – at least three – points denoted by the range [*points_begin*, *points_end*) form the boundary of a convex polygon (oriented clock– or counterclockwise) *and*
5. $k \geq 2$.

Note

On compilers not supporting member function templates, the parameter *RandomAccessIC* is fixed to *vector<Point_2>::iterator* where *Point_2* is the value type of *RandomAccessIC*.

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/Point_2.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.h>
#include <vector.h>
```

```
int main() {

  typedef double                              FT;
  typedef CGAL_Cartesian< FT >                R;
  typedef CGAL_Point_2< R >                   Point_2;
  typedef CGAL_Polygon_traits_2< R >          P_traits;
  typedef vector< Point_2 >                   Cont;
  typedef CGAL_Polygon_2< P_traits, Cont >    Polygon_2;
  typedef CGAL_Random_points_in_square_2<
    Point_2,
    CGAL_Creator_uniform_2< FT, Point_2 > >
  Point_generator;

  Polygon_2 p;
  int number_of_points( 10);
  int k( 5);

  CGAL_random_convex_set_2( number_of_points,
                            back_inserter( p),
                            Point_generator( 1));
  cout << "Generated Polygon:\n" << p << endl;

  Polygon_2 k_gon;
  CGAL_maximum_perimeter_inscribed_k_gon(
    p.vertices_begin(),
    p.vertices_end(),
    k,
    back_inserter( k_gon));
  cout << "Maximum perimeter " << k << "-gon:\n" << k_gon << endl;

}
```

9.3.3 Computing General Extremal Polygons

This section describes a general function to compute a maximal k-gon P_k that can be inscribed in a given convex polygon P. The criterion for maximality and some basic operations have to specified in an appropriate traits class as specified in section 9.3.4.

#include <CGAL/extremal_polygons_2.h>

template < class RandomAccessIC, class OutputIterator, class Traits >
OutputIterator CGAL_extremal_polygon(
RandomAccessIC points_begin,
RandomAccessIC 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. *Traits* has to satisfy the requirements stated in section 9.3.4,
2. Value type of *RandomAccessIC* must be *Traits::Point_2*,
3. *OutputIterator* accepts *Traits::Point_2* as value type,
4. the – at least three – points denoted by the range [*points_begin*, *points_end*) form the boundary of a convex polygon (oriented clock– or counterclockwise) *and*
5. $k \geq t.min\_k()$.

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.

9.3.4 Requirements for Extremal Polygon Traits Classes

Definition

A class *Exp_traits* has to provide the following types and operations in order to qualify as a traits class for *CGAL_extremal_polygon*.

Types

Exp_traits:: Point_2	class used for representing the input points.
Exp_traits:: FT	class used for doing computations on point coordinates (has to fulfill field-type requirements).
Exp_traits:: Operation	AdaptableBinaryFunction class *op*: *Point_2* × *Point_2* → *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 RandomAccessIC, class OutputIterator >
OutputIterator *t.compute_min_k_gon(RandomAccessIC points_begin,*
RandomAccessIC 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()*).

template < class RandomAccessIC >
bool *t.is_convex(RandomAccessIC points_begin,*
RandomAccessIC points_end) const

returns true, iff the points [*points_begin*, *points_end*) form a convex chain.

Notes

- *Exp_traits::is_convex* is only used for precondition checking. Therefore it needs not to be specified, in case that precondition checking is disabled.
- On compilers not supporting member function templates, *RandomAccessIC* is fixed to *vector<Point_2>::iterator* and *OutputIterator* is fixed to *vector<int>::reverse_iterator*.

See Also

#include <CGAL/Extremal_polygon_traits_2.h>

The classes *CGAL_Kgon_area_traits<R>* and *CGAL_Kgon_perimeter_traits<R>* (templatized with a CGAL representation class) both fulfill these requirements.

9.4 All Furthest Neighbors

This section describes a function to compute all furthest neighbors for the vertices of a convex polygon.

#include <CGAL/all_furthest_neighbors_2.h>

template < class RandomAccessIC, class OutputIterator, class Traits >
OutputIterator CGAL_all_furthest_neighbors(
RandomAccessIC points_begin,
RandomAccessIC 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

1. If t is specified explicitly, *Traits* satisfies the requirements stated in section 9.4.1,
2. Value type of *RandomAccessIC* is *Traits::Point_2* or – if t is not specified explicitly – *CGAL_Point_2<R>* for some representation class R,
3. *OutputIterator* accepts *int* as value type *and*
4. the points denoted by the non-empty range [*points_begin*, *points_end*) form the boundary of a convex polygon P (oriented clock– or counterclockwise).

Implementation

The implementation uses monotone matrix search[AKM+87] (see also section 9.6). Its runtime complexity is linear in the number of vertices of P.

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/Point_2.h>
#include <CGAL/Polygon_2.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/random_convex_set_2.h>
```

[1] i.e. the furthest neighbor of *points_begin[i]* is *points_begin[*i-th number written to *o]*

```
#include <CGAL/all_furthest_neighbors_2.h>
#include <CGAL/IO/Ostream_iterator.h>
#include <iostream.h>
#include <vector.h>

typedef double                                    FT;
typedef CGAL_Cartesian< FT >                      R;
typedef CGAL_Point_2< R >                         Point_2;
typedef CGAL_Polygon_traits_2< R >                P_traits;
typedef vector< Point_2 >                         Point_cont;
typedef CGAL_Polygon_2< P_traits, Point_cont >    Polygon_2;
typedef CGAL_Random_points_in_square_2<
  Point_2,
  CGAL_Creator_uniform_2< FT, Point_2 > >
Point_generator;
typedef CGAL_Ostream_iterator< int, ostream >  Ostream_iterator;

int
main()
{
  // generate random convex polygon:
  Polygon_2 p;
  CGAL_random_convex_set_2( 10,
                            back_inserter( p),
                            Point_generator( 1));

  // compute all furthest neighbors:
  CGAL_all_furthest_neighbors(
    p.vertices_begin(),
    p.vertices_end(),
    Ostream_iterator( cout));
  cout << endl;

}
```

9.4.1 Requirements for All Furthest Neighbors Traits Classes

Definition

A class *Afn_traits* has to provide the following types and operations in order to qualify as a traits class for *CGAL_all_furthest_neighbors*.

Types

Afn_traits:: Point_2	class used for representing the input points.
Afn_traits:: FT	class used for doing computations on point coordinates (has to fulfill field-type requirements).

Afn_traits:: Distance — AdaptableBinaryFunction class: $Point\_2 \times Point\_2 \to FT$ computing the squared Euclidean distance between two points.

Operations

template < class RandomAccessIC >
bool *t.is_convex(RandomAccessIC points_begin, RandomAccessIC points_end) const*

returns true, iff the points [*points_begin*, *points_end*) form a convex chain.

Notes

- *Afn_traits::is_convex* is only used for precondition checking.
- On compilers not supporting member function templates, *RandomAccessIC* is fixed to *vector<Point_2>::iterator*.

9.5 Rectangular p-Centers

This section describes a function to compute rectilinear p-centers of a planar point set, i.e. a set of p points such that the maximum minimal 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 Traits >
OutputIterator          CGAL_rectangular_p_center_2(
                                ForwardIterator f,
                                ForwardIterator l,
                                OutputIterator o,
                                Traits::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

1. If t is specified explicitly, *Traits* satisfies the requirements stated in section 9.5.1,
2. Value type of *RandomAccessIC* is *Traits::Point_2* or – if t is not specified explicitly – *CGAL_Point_2<R>* for some representation class R,
3. *OutputIterator* accepts the value type of *RandomAccessIC* as value type,
4. the range $[f, l)$ is not empty *and*
5. $p = 2$.

Note

On compilers not supporting member function templates, the parameter *RandomAccessIC* is fixed to *vector<Point_2>::iterator* and *OutputIterator* is fixed to *back_insert_iterator<vector<Point_2>>* where *Point_2* is *Traits::Point_2*.

Implementation

The implementation uses sorted matrix search (see section 9.7) and fast algorithms for piercing rectangles[SW96] which leads to a runtime complexity of $\mathcal{O}(n \cdot \log n)$ where n is the number of input points.

Example

The following code generates a random set of ten points and computes its two-centers.

```
#include <CGAL/Cartesian.h>
#include <CGAL/Point_2.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/rectangular_p_center_2.h>
#include <CGAL/copy_n.h>
#include <CGAL/IO/Ostream_iterator.h>
#include <iostream.h>
#include <vector.h>

typedef double                              FT;
typedef CGAL_Cartesian< FT >                R;
typedef CGAL_Point_2< R >                   Point_2;
typedef vector< Point_2 >                   Cont;
typedef CGAL_Random_points_in_square_2<
  Point_2,
  CGAL_Creator_uniform_2< FT, Point_2 > >
Point_generator;
typedef CGAL_Ostream_iterator< Point_2, ostream >
  Ostream_iterator_point;

int main() {

  int number_of_points( 10);
  int p( 2);
  Ostream_iterator_point cout_ip( cout);
  CGAL_set_pretty_mode( cout);

  Cont points;
  CGAL_copy_n( Point_generator( 1),
               number_of_points,
               back_inserter( points));
  cout << "Generated Point Set:" << endl;
  copy( points.begin(), points.end(), cout_ip);

  Cont centers;
  FT p_radius;
  CGAL_rectangular_p_center_2(
    points.begin(),
    points.end(),
    back_inserter( centers),
    p_radius,
    p);
  cout << "\n\n" << p << "-centers:" << endl;
  copy( centers.begin(), centers.end(), cout_ip);
  cout << "\n\n" << p << "-radius = " << p_radius << endl;

}
```

9.5.1 Requirements for Rectangular p-center Traits Classes

Definition

A class *Rpc_traits* has to provide the following types in order to qualify as a traits class for *CGAL_rectangular_p_center_2*.

Types

Rpc_traits:: Iso_rectangle_2	class used for representing axis-parallel rectangles.
Rpc_traits:: Point_2	class used for representing the input points.
Rpc_traits:: FT	class used for doing computations on point coordinates (has to fulfill field-type requirements).
Rpc_traits:: X	adaptable unary function class: $Point\_2 \rightarrow FT$ computing the x-coordinate of a given point.
Rpc_traits:: Y	adaptable unary function class: $Point\_2 \rightarrow FT$ computing the y-coordinate of a given point.
Rpc_traits:: Build_point	adaptable binary function class: $FT \times FT \rightarrow Point\_2$ creating a point with the given coordinates.

9.6 Monotone Matrix Search

In this section we describe a general function to compute 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    CGAL_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

1. *Matrix* satisfies the requirements stated in section 9.6.1,
2. Value type of *RandomAccessIC* is *int*,
3. t points to a structure of size at least *m.number_of_rows() and*
4. if *compare_strictly* is defined, it has to be an adaptable binary function: *Matrix::Value* $\times$ *Matrix::Value* $\rightarrow$ *bool* describing a strict (non-reflexive) total ordering on *Matrix::Value*.

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.

9.6.1 Requirements for Monotone Matrix Search Matrix Classes

Definition

A class *Mms_matrix* has to provide the following types and operations in order to qualify as argument to *CGAL_monotone_matrix_search*.

Types

Mms_matrix:: 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:
$0 \leq row < number\_of\_rows()$ and
$0 \leq column < number\_of\_columns()$.

*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 in section 9.6), 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 *CGAL_Dynamic_matrix* (see section 9.6.2) that can be used to add most of the functionality described above to arbitrary matrix classes.

9.6.2 An Adaptor for Adding Dynamic Operations to Matrix Classes

#include <CGAL/Dynamic_matrix.h>

Definition

The class *CGAL_Dynamic_matrix<M>* can be used as an adaptor for an arbitrary matrix class M to provide the dynamic operations needed for monotone matrix search (see section 9.6). The template argument class has to fulfill some requirements as stated in section 9.6.3.

Creation

CGAL_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()*

returns the number of columns.

int *d.number_of_rows()*

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:
$0 \leq row < number\_of\_rows()$ and
$0 \leq column < number\_of\_columns()$.

*Matrix** *d.extract_all_even_rows()*

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:
$number\_of\_rows() == number\_of\_columns()$.

Implementation

All operations take constant time except for *extract_all_even_rows* which needs time linear in the number of rows.

9.6.3 Basic Requirements for Matrix Classes

In this section we list some basic requirements for matrix classes.

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
$0 \leq column < number\_of\_columns()$.

9.7 Sorted Matrix Search

This section describes a general function to select the smallest entry in a set of sorted matrices that fulfills a certain criterion.

More exactly, a matrix $M = (m_{ij}) \in S^{r \times l}$ (over a totally ordered set S) is sorted, iff

$$\forall\, 1 \le i \le r,\ 1 \le j < l \ :\ m_{ij} \le m_{i(j+1)} \ \ and$$
$$\forall\, 1 \le i < r,\ 1 \le j \le l \ :\ m_{ij} \le m_{(i+1)j} \ .$$

Now let $\mathcal{M}$ be a set of n sorted matrices over S and f be a monotone predicate on S, i.e.

$$f \ :\ S \longrightarrow bool \quad \text{with} \quad f(r) \implies \forall t \in S\,,\, t > r \ :\ f(t)\ .$$

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. The requirements for this class are described in section 9.7.1.

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

template < class RandomAccessIC, class Traits >
Traits::Value          CGAL_sorted_matrix_search(
                           RandomAccessIC f,
                           RandomAccessIC 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. *Traits* satisfies the requirements for sorted matrix search traits classes stated in section 9.7.1,
2. Value type of *RandomAccessIC* is *Traits::Matrix*,
3. All matrices in $[f, l)$ are sorted according to *Traits::compare_non_strictly and*
4. there is at least one entry x in a matrix $M \in [f, l)$ for which *Traits::is_feasible(x)* is true.

Implementation

The implementation uses an algorithm by Frederickson and Johnson[FJ83],[FJ84] and runs in $\mathcal{O}(n \cdot k + f \cdot \log(n \cdot k))$, 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 *CGAL_sorted_matrix_search* to compute the upper bound for the maximal entry of a in M.

```
#include <CGAL/Random.h>
#include <CGAL/function_objects.h>
#include <CGAL/Cartesian_matrix.h>
#include <CGAL/sorted_matrix_search.h>
#include <vector.h>

int main() {

  typedef int                                Value;
  typedef vector< Value >                    Vector;
  typedef Vector::iterator                   Value_iterator;
  typedef vector< Vector >                   Vector_cont;
  typedef CGAL_Cartesian_matrix<
    plus< int >,
    Value_iterator,
    Value_iterator >                         Matrix;

  // set of vectors the matrices are build from:
  Vector_cont vectors;

  // generate a random vector and sort it:
  Vector a;
  int i;
  cout << "a = ( ";
  for ( i = 0; i < 5; ++i) {
    a.push_back( CGAL_random( 100));
    cout << a.back() << " ";
  }
  cout << ")" << endl;
  sort( a.begin(), a.end(), less< Value >());

  // build a cartesian from a:
  Matrix M( a.begin(), a.end(), a.begin(), a.end());

  // search an upper bound for max(a):
  Value bound( a[4]);
  Value upper_bound(
    CGAL_sorted_matrix_search(
      &M,
      &M + 1,
      CGAL_sorted_matrix_search_traits_adaptor(
        bind2nd( greater_equal< Value >(), bound),
        M)));
  cout << "upper bound for " << bound << " is "
       << upper_bound << endl;

}
```

9.7.1 Requirements for Sorted Matrix Search Traits Classes

Definition

A class *Sms_traits* that satisfies the requirements of a traits class for the sorted matrix search algorithm has to provide the following types and operations.

Types

Sms_traits:: Matrix	The class used for representing matrices. It has to fulfill the requirements listed in section 9.6.3.
typedef Matrix::Value Value;	The class used for representing the matrix elements.
Sms_traits:: 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.
Sms_traits:: 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)*.

See Also

In section 9.7.2 we define an adaptor *CGAL_Sorted_matrix_search_traits_adaptor* to create sorted matrix search traits classes for arbitrary feasibility test and matrix classes.

9.7.2 A Traits Class Adaptor for Sorted Matrix Search

#include <CGAL/Sorted_matrix_search_traits_adaptor.h>

Definition

The class *CGAL_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. Any proper instantiation fulfills the requirements stated in section 9.7.1.

The compare operation *Compare_strictly* is set to *less<M::Value>* and *Compare_non_strictly* accordingly to *less_equal<M::Value>*.

Requirements

1. M has to fulfill the requirements listed in section 9.6.3 *and*
2. F has to define a copy constructor and a monotone *bool operator()(const Value&)*.

Creation

CGAL_Sorted_matrix_search_traits_adaptor<F,M> t(const F& m);

initializes t to use m for feasibility testing.

Chapter 10

Search Structures

10.1 Introduction

This chapter presents the CGAL range tree, segment tree, and KD-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.

First the range and segment trees are introduced in sections 10.2 through 10.7. Then the KD-tree is presented in Sections 10.8 through 10.10.

10.2 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 10.1, 10.2 and 10.4 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.). These classes are defined in Section 10.3 (Section 10.4 resp.). These classes are parameterized with a traits class that defines, among other things, the key and interval type of a tree (cf. Section 10.5). In Section 10.6 we give the requirements tree traits classes must fulfill.

In case you want to create a combined tree structure, refer Section 10.7. The classes described in that section enable you to define each layer of a tree separately. These tree classes are templatized with the type of the data item, the query interval and a traits class which is used as an interface to the data. In Section 10.7.1 we describe the ready-to-use traits CGAL provides and the requirements a traits class has to fulfill. By this, the advanced user can develop his own traits classes.

We now give a short definition of the version of the range tree and segment tree implemented here. The presentation closely follows [dBvKOS97].

10.2.1 Definition of a Range Tree

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 10.1 shows a two-dimensional range tree, Figure 10.2 a d-dimensional one.

10.2.2 Definition of a Segment Tree

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 enclose a given d-dimensional interval (*enclosing_query*).

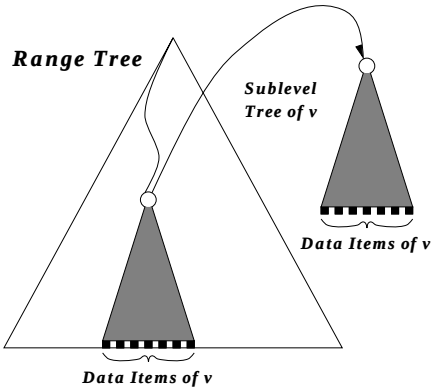


Figure 10.1: A two-dimensional range tree. The tree is a binary search tree on the first dimension. Each sublayer tree of a vertex v is a binary search tree on the second dimension. The data items in a sublayer tree of v are all data items of the subtree of v.

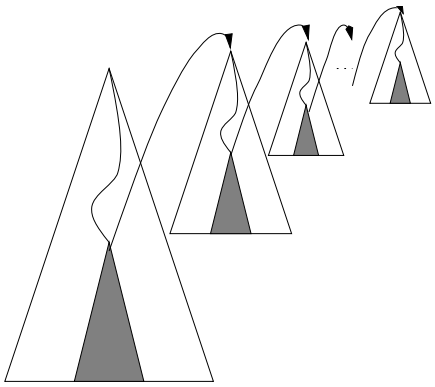


Figure 10.2: A d-dimensional range tree. For each layer of the tree, one sublayer tree is illustrated.

- Determine all d-dimensional intervals that partially overlap or are contained in a given d-dimensional interval ($window\_query$).

In Figure 10.3 an example of a one-dimensional segment tree is given. Figure 10.4 shows a two-dimensional segment tree.

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.

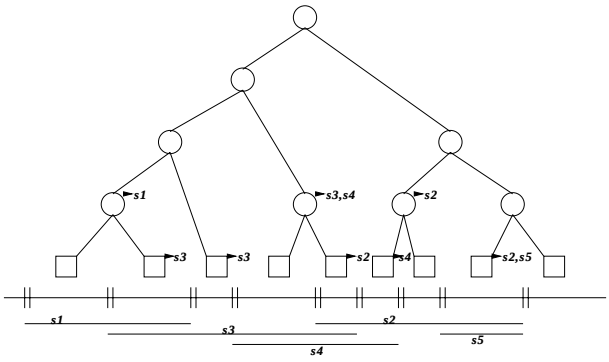


Figure 10.3: A one-dimensional segment tree. The segments and the corresponding elementary intervals are shown below the tree. The arcs from the nodes point to their subsets.

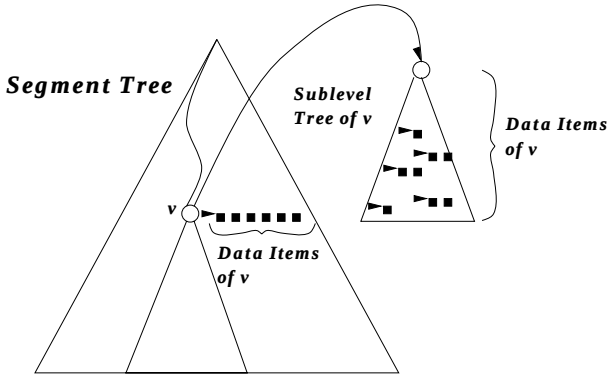


Figure 10.4: A two-dimensional segment tree. The first layer of the tree is built according to the elementary intervals of the first dimension. Each sublayer tree of a vertex v is a segment tree according to the second dimension of all data items of v.

10.3 k-dimensional Range Trees

CGAL provides four range tree classes for $k \in \{1, \ldots, 4\}$.

Definition

An object of the class *CGAL_Range_tree_k* 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 *CGAL_Range_tree_k* 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 presented in Section 10.5. In Section 10.6 we give 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

CGAL_Range_tree_k<Traits>:: Traits

the type of the range tree traits class.

typedef Traits::Key Key;

typedef Traits::Interval

Interval;

Creation

CGAL_Range_tree_k<Traits> R;

Introduces an empty range tree R.

template < class ForwardIterator >
CGAL_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 piece of code creates a number of 2-dimensional points that have a character associated with it, and constructs a 2-dimensional range tree for these data. Then a window query is performed. The result is written to standard output.

```
#include <CGAL/Cartesian.h>
#include <CGAL/Point_2.h>
#include <CGAL/Range_segment_tree_traits.h>
#include <CGAL/Range_tree_k.h>

typedef CGAL_Cartesian<double> Rep;
typedef CGAL_Range_tree_map_traits_2<Rep, char> Traits;
typedef CGAL_Range_tree_2<Traits> Range_tree_2_type;

void main()
{
  /* pair<CGAL_Point_2<Rep>,char> */
  typedef Traits::Key Key;
  /* pair<CGAL_Point_2<Rep>,CGAL_Point_2<Rep> > */
  typedef Traits::Interval Interval;

  vector<Key> InputList, OutputList;
  InputList.push_back(Key(CGAL_Point_2<Rep>(8,5.1), 'a'));
  InputList.push_back(Key(CGAL_Point_2<Rep>(1,1.1), 'b'));
  InputList.push_back(Key(CGAL_Point_2<Rep>(3,2.1), 'c'));

  Range_tree_2_type Range_tree_2(InputList.begin(),InputList.end());
  Interval win(Interval(CGAL_Point_2<Rep>(4,8.1),CGAL_Point_2<Rep>(5,8.2)));
  cerr << "\n Window Query:\n ";
  Range_tree_2.window_query(win, back_inserter(OutputList));
  vector<Key>::iterator current=OutputList.begin();
  while(current≠OutputList.end())
```

```
  {
    cout << (*current).first.x() << "," << (*current).first.y()
         << ":" << (*current++).second << endl;
  }
}
```

10.4 k-dimensional Segment Trees

Definition

An object of the class *CGAL_Segment_tree_k* is a k-dimensional segment tree that can store k-dimensional intervals of type *Interval*. The class allows to perform window queries on the keys. The class *CGAL_Segment_tree_k* is parameterized with a segment tree traits class *Traits* that defines, among other things, the type of the *Interval*.

CGAL provides traits class implementations that allow to use the segment tree with point classes from the CGAL kernel as keys. These classes are presented in Section 10.5. In Section 10.6 we give 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

CGAL_Segment_tree_k<Traits>:: Traits

the type of the segment tree traits class.

typedef Traits::Key Key;

typedef Traits::Interval

Interval;

Creation

CGAL_Segment_tree_k<Traits> S;

Introduces an empty segment tree S.

template < class ForwardIterator >
CGAL_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

The following piece of code creates some 2-dimensional intervals and constructs a 2-dimensional segment tree for these data. Then a window query and an enclosing query is performed. The result is written to standard output.

```
#include <CGAL/Homogeneous.h>
#include <CGAL/Point_2.h>
#include <CGAL/Segment_tree_k.h>
#include <CGAL/Range_segment_tree_traits.h>

typedef CGAL_Cartesian<double> Rep;
typedef CGAL_Range_segment_tree_traits_2<Rep> Traits;
typedef CGAL_Segment_tree_2<Traits > Segment_tree_2_type;

int main()
{
  typedef Traits::Key Key;              /* is a CGAL_Point_2<Rep> */
  typedef Traits::Interval Interval;  /* is a pair<Key,Key> */
  list<Interval> InputList, OutputList, N;

  InputList.push_back(Interval(Key(1,5), Key(2,7)));
  InputList.push_back(Interval(Key(2,7), Key(3,8)));
```

```
  InputList.push_back(Interval(Key(6,9), Key(9,13)));
  InputList.push_back(Interval(Key(1,3), Key(3,9)));

  Segment_tree_2_type Segment_tree_2(InputList.begin(), InputList.end());

  Interval a(Key(3,6), Key(7,12));
  Segment_tree_2.window_query(a,back_inserter(OutputList));

  /* output of the query elements on stdout */
  list<Interval>::iterator j = OutputList.begin();
  cout << "\n window_query \n";
  while(j≠OutputList.end())
  {
    cout << (*j).first.x() << "-" << (*j).second.x() << " "
         << (*j).first.y() << "-" << (*j++).second.y() << endl;
  }
  Interval b(Key(6,10),Key(7,11));
  Segment_tree_2.enclosing_query(b,back_inserter(OutputList));
  cout << "\n enclosing_query \n";
  while(j≠OutputList.end())
  {
    cout << (*j).first.x() << "-" << (*j).second.x() << " "
         << (*j).first.y() << "-" << (*j++).second.y() << endl;
  }
}
```

10.5 Tree Traits Class Implementations

This section provides range tree and segment tree traits class implementations for the CGAL kernel.

10.5.1 Set-like Tree Traits Classes

The following traits classes are set-like, since no data is associated to the keys.

Set-like Tree Traits Class for 2D Kernel Points

Definition

The class *CGAL_Range_segment_tree_traits_set_2* 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

CGAL_Point_2<R> Key;

pair<Key, Key> Interval;

Definition

The class *CGAL_Range_segment_tree_traits_set_3* 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

CGAL_Point_3<R> *Key;*

pair<Key, Key> *Interval;*

10.5.2 Map-like Range Tree Traits Classes

The following traits classes are map-like, since they allow to associate data to the keys.

Map-like Range Tree Traits Class for 2D Kernel Points

Definition

The class *CGAL_Range_tree_traits_map_2* 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

pair<CGAL_Point_2<R>, T>

Key;

pair<CGAL_Point_2<R>, CGAL_Point_2<R> >

Interval;

Map-like Range Tree Traits Class for 3D Kernel Points

Definition

The class *CGAL_Range_tree_traits_map_3* 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

pair<CGAL_Point_3<R>, T>

Key;

pair<CGAL_Point_3<R>, CGAL_Point_3<R> >

Interval;

10.5.3 Map-like Segment Tree Traits Classes

The following traits classes are map-like, since they allow to associate data to the keys.

Map-like Segment Tree Traits Class for 2D Kernel Points

Definition

The class *CGAL_Segment_tree_traits_map_2* 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

CGAL_Point_2<R> *Key;*

pair<pair<Key,Key>, T>

Interval;

Map-like Segment Tree Traits Class for 3D Kernel Points

Definition

The class *CGAL_Segment_tree_traits_map_3* 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

pair<CGAL_Point_3<R> >

Key;

pair<pair<Key, Key>, T>

Interval;

10.6 Tree Traits Class Requirements

This section describes the requirements for range and segment tree traits classes.

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

Range_segment_tree_traits_*`k`:: Key*

The `k`-dimensional key type.

Range_segment_tree_traits_*`k`:: Interval*

The `k`-dimensional interval type.

Range_segment_tree_traits_*`k`:: Key_i*

The type in dimension i, with $1 \leq i \leq k$.

Range_segment_tree_traits_*`k`:: key_i*

function object providing an *operator()* that takes an argument of type *Key* and returns a component of type *Key_i*.

Range_segment_tree_traits_*`k`:: low_i*

function object providing an *operator()* that takes an argument of type *Interval* and returns a component of type *Key_i*.

Range_segment_tree_traits_*`k`:: high_i*

function object providing an *operator()* that takes an argument of type *Interval* and returns a component of type *Key_i*.

Range_segment_tree_traits_*`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 pair<int, double> Key;
  typedef int Key_1;
  typedef double Key_2;
  typedef pair<Key,Key> Interval;

  class _Key_1{
  public:
    Key_1 operator()(const Key& k)
    { return k.first;}
  };
  class _Key_2{
  public:
    Key_2 operator()(const Key& k)
    { return k.second;}
  };
  class _Low_1{
  public:
    Key_1 operator()(const Interval& i)
    { return i.first.first;}
  };
  class _High_1{
  public:
    Key_1 operator()(const Interval& i)
    { return i.second.first;}
  };
  class _Low_2{
  public:
    Key_2 operator()(const Interval& i)
    { return i.first.second;}
  };
  class _High_2{
  public:
    Key_2 operator()(const Interval& i)
    { return i.second.second;}
  };
  class _Compare_1{
  public:
    bool operator()(Key_1 k1, Key_1 k2)
    { return less<int>()(k1,k2);}
  };
  class _Compare_2{
  public:
    bool operator()(Key_2 k1, Key_2 k2)
    { return less<double>()(k1,k2);}
  };
  typedef _Compare_1 compare_1;
  typedef _Compare_2 compare_2;
  typedef _Low_1 low_1;
```

```
  typedef _High_1 high_1;
  typedef _Key_1 key_1;
  typedef _Low_2 low_2;
  typedef _High_2 high_2;
  typedef _Key_2 key_2;
};
```

10.7 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.

10.7.1 Tree Traits class and its Requirements

Tree class *Range_tree_d* and class *Segment_tree_d* are templatized with a parameter *CGAL_Traits*. *CGAL_Traits* builds the interface between the tree and the data items. We now describe the traits class for a *Range_tree_d* layer and for a *Segment_tree_d* layer. The traits classes are implemented as template classes. If you do not want to use these traits classes you can also define your own class which has at least to provide the functionality of the traits class described below.

CGAL_tree_point_traits — Requirements

Definition

CGAL_tree_point_traits is a template class that provides an interface to data items.

Types

typedef
CGAL_tree_point_traits<Data, Window, Key, Data_func, Window_left_func, Window_right_func, Compare> Point_traits;

Point_traits:: Data	the container *Data* — defines the Data type.
Point_traits:: Window	the container *Window* — defines the type of the query rectangle.
Point_traits:: Key	the type *Key* of the data slot this traits class provides access to.

Creation

#include <CGAL/Tree_traits.h>
CGAL_tree_point_traits<Data, Window, Key, Data_func, Window_left_func, Window_right_func, Compare> d();

Data and *Window* are two containers, that may consist of several data slots. One of these data slots has to be of type *Key*. *Data_func* is a function object providing an *operator()* that takes an argument of type *Data* and returns a component of type *Key*. *Window_left_func* and *Window_right_func* are function objects that allow to access the left and right data slot of container *Window* which has type *Key*. *Compare* defines a comparison relation which must define a strict ordering of the objects of type *Key*. If defined, *less<Key>* is sufficient.

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*.

bool *d.comp(Key& key1, Key& key2)*

returns *Compare(key1, key2)*.

bool *d.key_comp(Data& data1, Data& data2)*

returns *Compare(get_key(data1), get_key(data2))*.

CGAL_tree_interval_traits

Definition

CGAL_tree_interval_traits is a template class that provides an interface to data items. It is similar to *CGAL_tree_interval_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*.

Types

typedef
CGAL_tree_interval_traits<Data, Window, Key, Data_left_func, Data_right_func, Window_left_func, Window_right_func, Compare> Interval_traits;

Interval_traits:: Data — the container *Data* — the data type.

Interval_traits:: Window — the container *Window* — the query window type.

Interval_traits:: Key — the type *Key* of the data slot this traits class provides access to.

Creation

#include <CGAL/Tree_traits.h>
CGAL_tree_interval_traits<Data, Window, Key, Data_left_func, Data_right_func, Window_left_func, Window_right_func, Compare> d();

Data and *Window* are two containers, that may consist of several data slots. Two of these data slots has to be of type *Key*. *Data_left_func* and *Data_left_func* are two function objects that provide access to two data slot of container *Data* which have type *Key*. *Data_left_func* (*Data_right_func* resp.) are two function objects that returns the Key of which is associated with the left (right resp.) boundary of an interval. *Window_left_func* and *Window_right_func* are function objects that return the left and right data slot of container *Window* which have type *Key*. *Compare* defines a comparison relation which must define a strict ordering of the objects of type *Key*. If defined, *less<Key>* is sufficient.

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*.

bool *d.comp(Key& key1, Key& key2)*

returns Compare(key1, key2).

10.7.2 CGAL_Range_tree and CGAL_Segment_tree

Definition

The tree classes were first intended to have a template argument defining the type of the sublayer tree. This leads to nested template arguments, where the internal class and function identifier got longer than a compiler dependent limit. Even for dimension 2 this happened. Therefore we chose another design. Now a tree is created with a prototype of the sublayer tree. With this prototype the sublayer tree can be cloned. The design pattern corresponds to the Prototype design pattern in [GHJV95].

In this sense, an instance of a three-dimensional range tree (segment tree) would have to be created as a range tree (segment tree) with creation variable *Sublayer_type s*, which is a prototype of a two-dimensional range tree (segment tree). Because a range tree of segment tree is expecting a prototype for its creation, a recursion anchor which builds dimension "zero" is needed. *CGAL_Tree_anchor* described in section 10.7.3 fulfills all these requirements. All tree classes (range tree, segment tree, tree anchor) are derived from an abstract base class *CGAL_Tree_base*.

Additionally a range tree (segment tree) is build in function *make_tree* using iterators. The iterator concept is realized in the Standard Template Library and in many other libraries, see [MS96]. As long as the GNU, SUN and SGI compiler do not support template member functions, we only support member functions parameterized with iterators working on *STL list, STL vector* and *C-arrays*.

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 requirements are described in Section 10.7.1.

CGAL_Range_tree

Types

CGAL_Range_tree_d<Data, Window, Traits>:: Data

container *Data*.

CGAL_Range_tree_d<Data, Window, Traits>:: Window

container *Window*.

CGAL_Range_tree_d<Data, Window, Traits>:: Traits

container *Traits*.

Creation

#include <CGAL/Range_tree_d.h>
CGAL_Range_tree_d<Data, Window, Traits> r(CGAL_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 Section 10.7.1. 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 *CGAL_Tree_anchor<Data, Window> sublayer_tree* with the same template parameters (*Data* and *Window*) *CGAL_Range_tree_d* is defined. In order to construct a two-dimensional range tree, create *CGAL_Range_tree_d* with a one-dimensional *CGAL_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(Q 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 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 weather 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(const Q win, const C 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 $\mathcal{O}(n \log n^d)$ time. The points in the query window are reported in time $\mathcal{O}(k + \log^d n)$, where k is the number of reported points. The tree uses $\mathcal{O}(n \log n^d)$ storage.

CGAL_Segment_tree

Types

CGAL_Segment_tree_d<Data, Window, Traits>:: Data

container *Data*.

CGAL_Segment_tree_d<Data, Window, Traits>:: Window

container *Window*.

CGAL_Segment_tree_d<Data, Window, Traits>:: Traits

class *Traits*.

Creation

#include <CGAL/Segment_tree_d.h>
CGAL_Segment_tree_d<Data, Window, Traits> s(

CGAL_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 as described in Section 10.7.1. 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 *CGAL_Tree_anchor<Data, Window> sublayer_tree* with the same template parameters *Data* and *Window* *CGAL_Segment_tree_d* is defined. In order to construct a two-dimensional segment tree, create *CGAL_Segment_tree_d* with a one-dimensional *CGAL_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.
d-dimensional intervals $[a_1, b_1), \ldots, [a_d, b_d)$, $a_i, b_i \in T_i$, $1 \leq i \leq d$ with $a_j \geq b_j$ produce a *CGAL warning*.

OutputIterator *s.window_query(Q 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.

OutputIterator *s.enclosing_query(Q 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 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(const Q win, const C 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 $\mathcal{O}(n \log n^d)$ time. The points in the query window are reported in time $\mathcal{O}(k + \log^d n)$, where k is the number of reported points. The tree uses $\mathcal{O}(n \log n^d)$ storage.

10.7.3 Sublayer_type

A *Sublayer_type* of class *CGAL_Range_tree_d* or *CGAL_Segment_tree_d* has to fulfill the following requirements: First of all, the class has to be derived from the abstract base class *CGAL_Tree_base*. The traits class of the sublayer type is defined by this abstract base class. E.g., the class has to provide methods *make_tree*, *window_query*, *enclosing_query* and *is_inside*. with the same parameter types as the instantiated class *CGAL_Range_tree_d* or *CGAL_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 *CGAL_Segment_tree_d*, this function is expected to return *true*, *false* otherwise.

Such a recursion anchor class is provided by the following class.

CGAL_Tree_anchor

Definition

CGAL_Tree_anchor is also derived from *CGAL_Tree_base*. Therefore, it provides the same methods as *CGAL_Range_tree_d* and *CGAL_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* (*CGAL_Segment_tree_d* respectively) with *CGAL_Tree_anchor* and the container classes for the data items (*Data* and *Window*.

Definition

CGAL_Tree_anchor<Data, Window>:: Data

container *Data*.

CGAL_Tree_anchor<Data, Window>:: Window

container *Window*.

Creation

#include <CGAL/Tree_base.h>

CGAL_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(const Q key, const C 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.

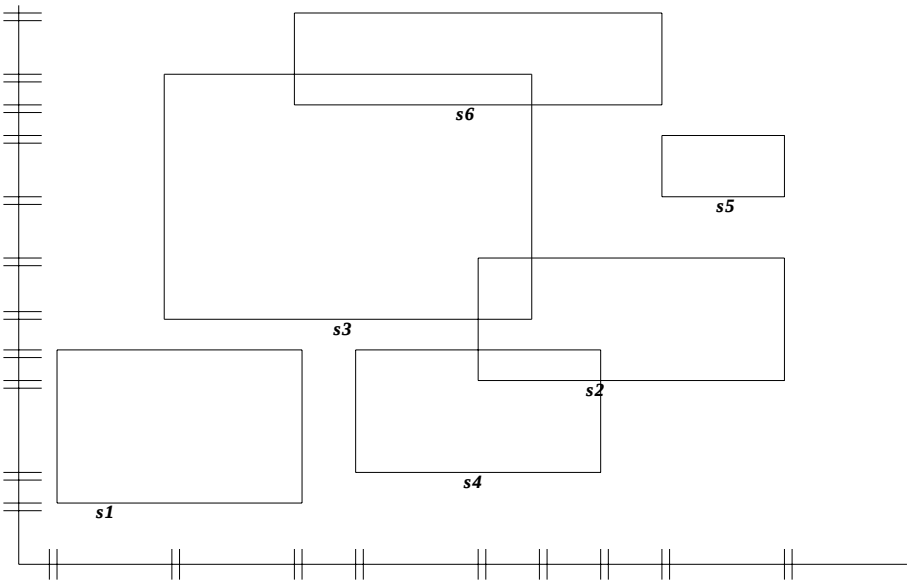


Figure 10.5: Two dimensional interval data.

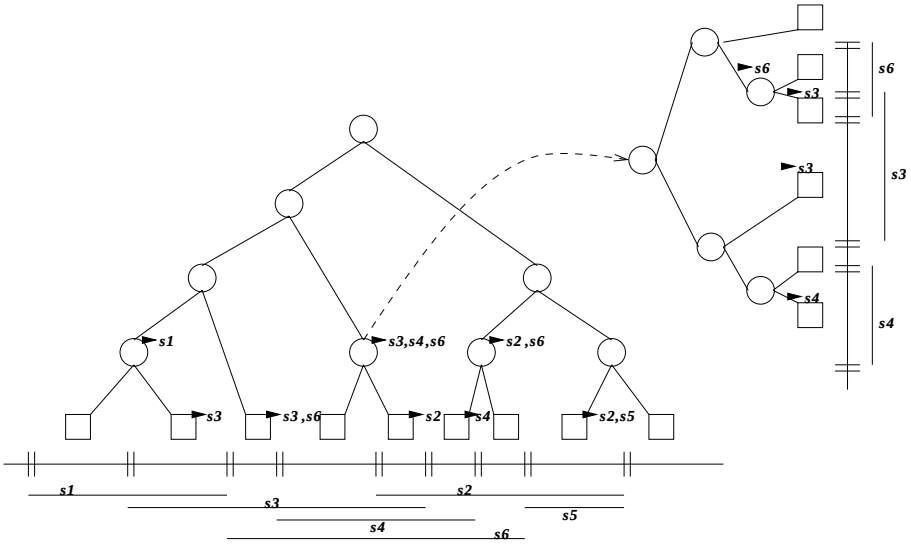


Figure 10.6: Two dimensional segment tree according to the interval data of Figure 10.5.

10.8 KD-trees

The implementation of the KD-tree is independent of the implementation of the rest of CGAL and can be used separately from CGAL.

The KD-tree class is parameterized with an interface class, that defines the interface between the KD-tree and the geometric primitives used. CGAL provides ready-made interface class that is presented in Section 10.9. The formal requirements for a class to be a KD-tree interface class is described in Section 10.9.

For a given set $S = \{p_1, \ldots, p_n\}$ on n points in $\mathbb{R}^d$, it is sometimes useful to be able to answer *orthogonal range-searching* on S; namely, given an axis parallel query box B in $\mathbb{R}^d$, one would like to "quickly" determine the subset of points of S lying inside B.

Several data structures were suggested for that problem. Foremost among those, at least theoreticly, is the range-tree data structure with $O(n \log^d(n))$ preprocessing time, $O(n \log^{d-1}(n))$ space, and $O(\log^d(n)+k)$ query time, where k is the output size of the query. A theoreticly inferior data structure is the KD-tree, which offers $O(n \log n)$ preprocessing time, $O(n)$ space, and $O(n^{1-1/d})$ query time. The KD-tree is a binary tree constructed as follows: we compute the point p_i of S such that its first coordinate is the median value among $p_1^1, \ldots, p_n^1$, where p_i^k denote the k-th coordinate of the i-th point of S. Let S_1 denote all the points of S with first coordinate smaller than p_i's first coordinate, and let $S_2 = S \setminus S_1$. Let T_1, T_2 be the KD-trees constructed recursively for S_1, S_2, respectively. The KD-tree of S is simply the binary tree having T_1 as its left subtree and T_2 as its right subtree. We apply this algorithm recursively, splitting the sets in the i-level of the KD-tree using the median point in the k-th coordinate, where $k = (i \mod n) + 1$. See Figure 10.7 for an illustration.

The resulting data structure has linear size, $O(n \log n)$ preprocessing time, and can answer a query in $O(n^{1-1/d} + k)$ time, where k is the size of the query output. See [dBvKOS97].

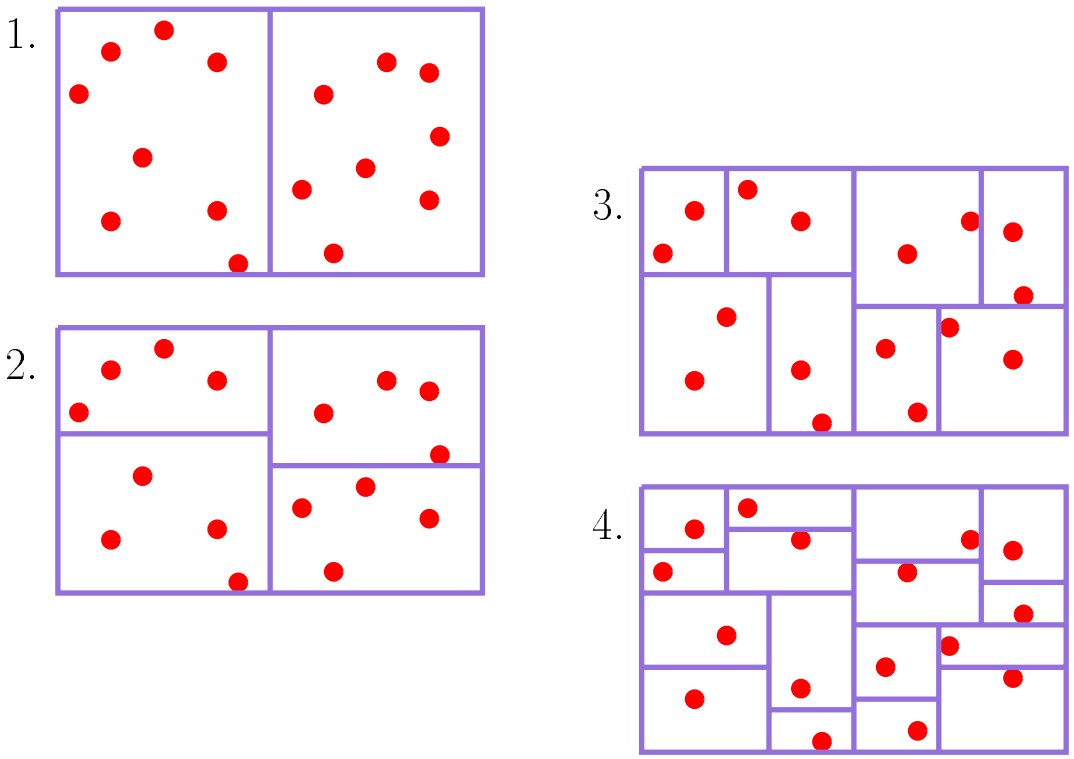


Figure 10.7: The partition of a set of points induced by a KD-tree of the points

In this section we define the class *CGAL_Kdtree_d* that implements the CGAL KD-tree.

CGAL_Kdtree_d<I>

Definition

An object T of the class *CGAL_Kdtree_d<I>* is the KD-tree induced by a set of points in d-dimensions.
#include < CGAL/kdtree_d.h>

Types

CGAL_Kdtree_d<I>:: Box represents an axis-parallel box in d-dimensions. The box might be unbounded.

typedef I::Point Point;

typedef list<Point> List_points;

Creation

CGAL_Kdtree_d<I> kd_tree(int dim = 2);

construct an empty KD-tree of dimension *dim*.

Operations

bool kd_tree.is_valid(bool verbose = false, int level = 0)

perform internal consistency checks to verify the correctness of the KD-tree

void kd_tree.build(list<Point> &l)

construct the KD-tree from the points stored in l.
Precondition: all the points in l are of dimension no smaller than the dimension of *kd_tree*itself.

void kd_tree.search(back_insert_iterator<List_points> result, Box & query_box)

return into *result* all the points of the KD-tree that lie inside *query_box*

CGAL_Kdtree_d<I>::Box

Definition

An object B of the class *Box* is a d-dimensional box (it may be unbounded). A d-dimensional box is the set defined by the Cartesian set $[l_1, r_1) \times [l_2, r_2) \times \cdots \times [l_d, r_d)$.

Creation

CGAL_Kdtree_d<I>::Box box(int d);

Construct a box corresponding to the whole d-dimensional space

CGAL_Kdtree_d<I>::Box box(Point left, Right right, int d);

Construct the axis parallel box in the d-dimensional space defined by the points *left*, *right*.

Operations

void *box.set_coord_left(int k, Point & left)*

set the left endpoint of the k-th dimensional interval of *box* to be the k-th coordinate of *left*

void *box.set_coord_right(int k, Point & right)*

set the right endpoint of the k-th dimensional interval of *box* to be the k-th coordinate of *right*

bool *box.is_in(Point pnt)*

return *true* if *pnt* lies inside *box*.

bool *box.is_coord_in_range(int k, Point pnt)*

return *true* if the k-th coordinate of *pnt* lies inside the k-th dimensional interval of *box*.

10.9 KD-tree Default Interface Class

CGAL contains a default implementation for the KD-tree interface class. In this section we describe how to use it.

KD-tree implemented using the default interface may be applied to any standard class of point provided by CGAL.

The default interface class *CGAL_Kdtree_Interface<Point>* is templated with a parameter *Point*, which is required to supply the following methods:

- Constructor, and copy constructor.
- *int dimension();* - return the dimension of the point.
- *operator[](int dim);* - an operator for accessing the various coordinates of the given point. Used also to copy coordinates between points.

There are two other default interface classes *CGAL_Kdtree_Interface_2d<Point>* and *CGAL_Kdtree_Interface_3d<Point>* which should be used when using 2D and 3D points from the CGAL kernel. This is done since the points in the kernel do not support changing their coordinates through direct access.

10.10 KD-tree Interface Class Specification

In this section we present the formal requirements for a KD-treeinterface class, that can be used to instantiate a variable of type *CGAL_Kdtree_d<I>*.

The KD-tree class is parameterized with the interface class I which defines the abstract interface between the KD-tree and the data (i.e., points). The following requirement catalog lists the primitives, i.e., types, member functions etc., that must be defined for a class that can be used to parameterize KD-trees. Ready-made implementation is available and described in Section 10.9.

I (I)

Definition

A class I that satisfies the requirements of an interface class for a KD-tree class must provide the following types and operations.

Types

I :: Point	A type to hold a input item.

Operations

static CGAL_Comparison_result	*i.compare(int k, Point p0, Point p1)*	compare the k-th coordinate of $p0$ and $p1$. Return *CGAL_LARGER* if $p0_k > p1_k$, *CGAL_SMALLER* if $p0_k < p1_k$, or else *CGAL_EQUAL*.
static void	*i.copy_coord(int k, Point & dest, Point src)*	Copy the k-th coordinate of *src* to the k-th coordinate of *dest*.
static int	*i.dimension(Point pnt)*	return the dimension of *pnt*

Bibliography

[AKM+87] A. Aggarwal, M. M. Klawe, S. Moran, P. Shor, and R. Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2:195–208, 1987.

[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.

[AT78] S. G. Akl and G. T. Toussaint. A fast convex hull algorithm. *Inform. Process. Lett.*, 7(5):219–222, 1978.

[Bau75] B. G. Baumgart. A polyhedron representation for computer vision. In *Proc. AFIPS Natl. Comput. Conf.*, volume 44, pages 589–596, 1975.

[BFH95] H. Bendels, D. W. Fellner, and S. 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.

[BPP95] G. Bell, A. Parisi, and M. Pesce. Vrml the virtual reality modeling language: Version 1.0 specification. `http://www.vrml.org/`, May 26 1995. Third Draft.

[Byk78] A. Bykat. Convex hull of a finite set of points in two dimensions. *Inform. Process. Lett.*, 7:296–298, 1978.

[dBvKOS97] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 1997.

[dBvOO96] M. de Berg, R. van Oostrum, and M. Overmars. Simple traversal of a subdivision without extra storage. In *Proc. 12th Annual Symp. on Comput. Geom.*, 1996, pages C5–C6.

[Edd77] W. F. Eddy. A new convex hull algorithm for planar sets. *ACM Trans. Math. Softw.*, 3:398–403 and 411–412, 1977.

[FJ83] G. N. Frederickson and D. B. Johnson. Finding kth 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.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Gra72] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Inform. Process. Lett.*, 1:132–133, 1972.

[GS85] L. J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Trans. Graph.*, 4:74–123, 1985.

[GS97a] B. Gärtner and S. Schönherr. Exact primitives for smallest enclosing ellipses. In *Proc. 13th Annu. ACM Symp. on Computational Geometry*, pages 430–432, 1997.

[GS97b] B. Gärtner and S. Schönherr. Smallest enclosing ellipses – fast and exact. Serie B – Informatik B 97-03, Freie Universität Berlin, Germany, June 1997.

[Hai94] E. Haines. Point in polygon strategies. In Paul Heckbert, editor, *Graphics Gems IV*, pages 24–46. Academic Press, Boston, MA, 1994.

[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.

[Ket98] L. Kettner. Designing a data structure for polyhedral surfaces. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, 1998. to appear, preliminary version available as Technical Report #278, Department Informatik, ETH Zürich, Switzerland.

[KW97] D. Kühl and Karsten Weihe. Data access templates. *C++ Report*, June 1997.

[Män88] M. Mäntylä. *An Introduction to Solid Modeling*. Computer Science Press, Rockville, MD, 1988.

[Meh84] K. Mehlhorn. *Multi-dimensional Searching and Computational Geometry*, volume 3 of *Data Structures and Algorithms*. Springer-Verlag, Heidelberg, West Germany, 1984.

[MP78] D. E. Muller and F. P. Preparata. Finding the intersection of two convex polyhedra. *Theoret. Comput. Sci.*, 7:217–236, 1978.

[MS96] D. R. Musser and Atul Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley, 1996.

[Phi94] M. Phillips. *Geomview Manual: Geomview Version 1.5 for Silicon Graphics Workstations*. The Geometry Center, University of Minnesota, October 1994. `http://www.geom.umn.edu/software/download/geomview.html`.

[PS85] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, NY, 1985.

[Sam90] H. Samet. *Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.

[SW96] M. Sharir and E. Welzl. Rectilinear and polygonal p-piercing and p-center problems. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 122–132, 1996.

[SL95] A. Stepanov and M. Lee. The standard template library. `http://www.cs.rpi.edu/~musser/doc.ps`, October 1995.

[VRML96] The virtual reality modeling language specification: Version 2.0, ISO/IEC CD 14772. `http://www.vrml.org/`, August 4 1996.

[Wei85] K. Weiler. Edge-based data structures for solid modeling in curved-surface environments. *IEEE Computer Graphics and Application*, 5(1):21–40, January 1985.

[Wel91] E. 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] J. Wernicke. *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor, Release 2*. Addison-Wesley, 1994.