Chapter 1

Boolean Operations in 2D

1.1 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 explicitely 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 parametrized 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.

1.2 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 independ 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 (1.3.1).

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 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 1.3.1.

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.

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

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 1.3.1.
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 1.1) instantiates two simple polygons and computes their intersection. Note: here the non-exact (builtin) number type *float* is used.

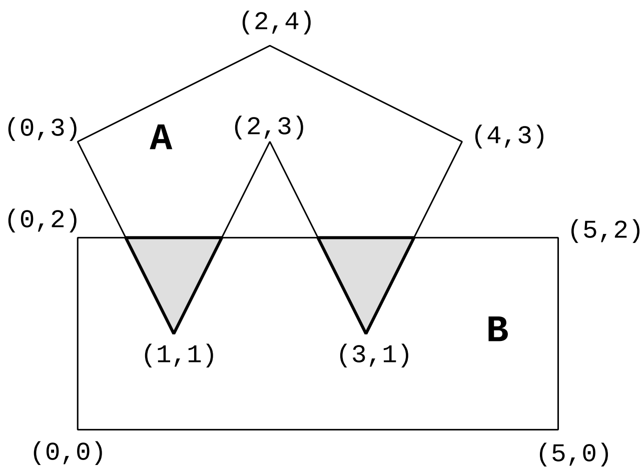

Figure 1.1: Intersection example of two simple polygons (in cartesian coordinates).

```
#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*).

1.3 Boolean Operations Traits Class Implementations

1.3.1 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 1.4.

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.is_equal(Point p1, Point p2)*

returns $p1 == p2$.

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()*.

double *Traits.xmin(Bbox bbox)*
returns *bbox.xmin()*.

double *Traits.xmax(Bbox bbox)*
returns *bbox.xmax()*.

double *Traits.ymin(Bbox bbox)*
returns *bbox.ymin()*.

double *Traits.ymax(Bbox bbox)*
returns *bbox.ymax()*.

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()*.

1.4 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_containe::push_back(Object)*.

Creation

Traits<R> bops_traits; default constructor

Operations

bool *bops_traits.do_overlap(Bbox a, Bbox b)*

returns *true* iff the do boxes do verlap.

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.is_leftturn(Point p0, Point p1, Point p2)*

returns *true* iff $p2$ lies onf 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*.
double	*bops_traits.xmin(Bbox bbox)*	
		returns *xmin* of *bbox*.
double	*bops_traits.xmax(Bbox bbox)*	
		returns *xmax* of *bbox*.
double	*bops_traits.ymin(Bbox bbox)*	
		returns *ymin* of *bbox*.
double	*bops_traits.ymax(Bbox bbox)*	
		returns *ymax* of *bbox*.
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*.