Colored DCEL for Boolean Operations in 2D *

Wolfgang Freiseisen

31. January 1998

Abstract

Finding the intersection, union, or difference of two simple polygons are well known problems in computational geometry. In this paper, new algorithms and their implementation solving these problems are presented. Assuming that line segment intersection is done already, the common main idea is to build up a colored doubly connected edge list (DCEL), independently of the actual operation. Colors indicate the relationship between polygons and elements, i.e. every element covered by a polygon will get its color. In order to perform the respective boolean operation the DCEL is traversed for elements with a certain color.

It will be shown that the DCEL can be constructed in $O(n \log(n))$ time and using this constructed DCEL the boolean operations need $O(n)$ time.

The given implementation, which is part of the CGAL[1] library, is parameterized with traits classes that define geometric predicates, number types, and representation classes. This approach makes it easy to adapt the algorithms to other geometric types and predicates.

1 Introduction

In technical applications like CAD, CAM, GIS, and many others it is often necessary to find the common interior of two or more objects (e.g. polygons or polyhedra), i.e. to calculate the intersection area/volume of these objects. Other, similar, problems are to compute the union, i.e. the area/volume covered by all objects, or the difference, i.e. the area/volume covered only by the first one of these objects, but not the others. Analogous to set theory we speak of boolean operations. In this paper we consider boolean operations on polygons in the 2-dimensional Euclidean space. We describe also how they can be implemented using the programming language C++. The current implementation is part of the CGAL-library[2], which is a C++-library that contains primitives, data structures, and algorithms for computational geometry. CGAL is developed by seven sites (Utrecht Univ., INRIA Sophia Antipolis, MPI Saarbrücken, ETH Zürich, Free Univ. Berlin, Tel Aviv. Univ., and RISC Linz). For the description of the algorithms (and the implementation) a basic knowledge of C++, STL[3], and CGAL will be presumed. Good introductions can be found, for instance in [Str91], [MS96], and [FGK+96], respectively.

The algorithms described in the next sections 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. Note that simple polygons 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. Hence, a typical result of a boolean operation will be a list of objects of mixed types, namely polygons, holes, segments, and points.

An intersection of two simple polygons will result in a list containing polygons, segments, and points. A union (or a difference) will result in a list containing polygons, holes (e.g. polygons with different orientation), segments, and points. Therefore, it is necessary to have a suitable data structure for

*This research is supported by the ESPRIT IV LTR Project No. 21957 (CGAL) of the European Union.

[1] Computational Geometry Algorithms Library

[2] http://www.cs.ruu.nl/CGAL

[3] Standard Template Library[SL95]

finding the correct results for each operation. Furthermore, computing the result should be robust and efficient.

The solution we choose uses a planar graph as the underlying data structure, which allows an easy traversing of the result. This data structure contains vertices, edges, faces, and holes, and each of these element types has a color. The result needs neither necessarily be connected nor is it guaranteed that there are no dangling elements.

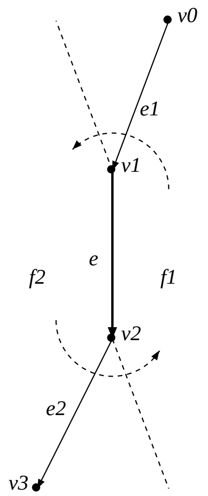
The Algorithm:

In order to perform boolean operations on simple polygons we use the following strategy:

1. Find all intersections by doing a sweep-line over the edges of the polygons ([MS88], [MN94], [BO79], [PS85]).
2. Construct a planar graph by inserting all edges and all intersections.
3. Traverse this planar graph in order to perform an intersection, union, or difference.

In this paper we will concentrate on the second and third step, constructing a planar graph from two simple polygons, and traversing it in order to get the respective result. For that reason we have to choose a suitable representation or data structure of a planar graph: the doubly connected edge list (DCEL).

This data structure contains the following elements: faces, edges, and vertices. Each element is able to store additional information like a color. This data structure also allows an easy and fast access of all elements and the insertion of polygons with associated colors.



E	V1	V2	F1	F2	P1	P2
e	v1	v2	f1	f2	e1	e2
e1	v0	v1	f1	f2		e
e2	v2	v3	f1	f2	e	

Figure 1: Illustration of the DCEL

1.1 A DCEL Represents a Planar Graph

Let $V = \{v_1, v_2, \ldots, v_n\}$ be a set of vertices and $E = \{e_1, e_2, \ldots, e_m\}$ be a set of edges. A planar graph $G = (V, E)$, not necessarily connected, can be represented in different ways. One straightforward representation would be to use a list of vertices V and a list of edges E, whereas each element V contains the vertex coordinates and each element of E represents an edge, which contains its source-vertex and its destination-vertex. The main disadvantage of this representation is that an efficient traversal of the graph is not possible without additional structures.

A more convenient representation, especially for our purposes, is the so called *Doubly Connected Edge List (DCEL)*. Its idea is based on this straightforward representation sketched above, however it is of course more efficient and a traversal, for instance around a face, can be done faster. The DCEL was invented by [MP78] and a detailed description also can be found in [PS85].

Topological Point of View

The most important role, as the name already states, will be played by the edges in a DCEL. Each edge is split into two half edges with opposite directions. Each half edge is defined by its two vertices, source and target vertex, where the opposite half edge is defined by the same two vertices in reverse order.

If an half edge is preceded (or followed) by another one, then its source (or target) vertex is identical with the target (or source) vertex of the preceding (or following) half edge. Two non identical vertices v_1, v_2 are connected iff there exists a chain of one or more edges, with v_1 and v_2 as its endpoints.

A subgraph of G where all vertices are connected is a called a connected component of G. Each connected component consists of at least one vertex, that means there can also exist connected components without any edges. This can be seen as an inconsistency of an edge based system, however, it is necessary to be able to represent each possible graph which could occur by applying boolean operations.

The cells bounded by a closed chain of edges are called faces. The interior of a face can contain dangling edges or singular vertices, but it cannot contain a chain which is not part of the boundary and connects two vertices of the boundary. However, a face inside a face is called a hole.

Usually, a face lies on the left side of a half edge. If both twins of an edge are incident of the same face, then this edge is a dangling edge or it is part of an dangling chain.

Technical Point of View

In principle a DCEL consists of

- the vertex-list (V),
- the edge-list (E),
- and two header lists, one for vertices (HV) and one for faces (HF).

The vertex-list contains the world-coordinates of each vertex and assigns an unique index to each vertex.

As shown in Figure 1 the main component of a DCEL representing a planar graph G is the edge-list, which consists of edge nodes, where each node has the following information:

- *Vertices*, where $v1$ is the source vertex and $v2$ the target vertex.
- *Faces*, with $f1$ is the left face and $f2$ the right face.
- *Edges*, with $p1$ pointing to the next edge in counter-clockwise order seen from $v1$ and $p2$ pointing to the next edge in counter-clockwise order seen from $v2$, respectively.

If the graph has n_v vertices and n_f faces, we use two arrays $HV[1 : n_v]$ and $HF[1 : n_f]$ of headers of the vertex and face lists. These header-lists are filled by a scan of arrays $v1$ and $f1$ in time $O(n)$.

2 DCEL for Boolean Operations

2.1 DCEL Elements

There are three types of elements of a DCEL: *vertex* (*Dcel_vertex*), *edge* (*Dcel_edge*), and *face* (*Dcel_face*). They have a common base class *Dcel_element_type*, where the color information (*Dcel_Color*) and the container index is stored. All other informations will be held in the derived classes respectively.

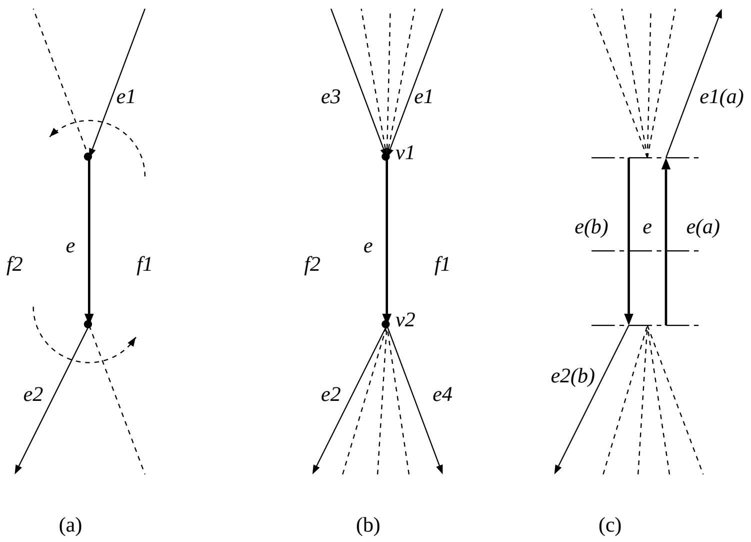


Figure 2: edge, winged-edge, and half-edge structure

Edge

An edge consists of four information fields corresponding to figure 1: two *vertices*, $V1$ source vertex and $V2$ target vertex, and two *faces*, $F1$ left face and $F2$ right face, Additionally there are two pointers to *edges*, pointer $P1$ points to the next edge in counterclockwise order seen from $V1$ and pointer $P2$ points to the next edge in counterclockwise order seen from $V2$, respectively ([MP78], [PS85]).

This structure is more or less a half-edge structure (figure 2a) and it depends on the concrete implementation how to follow a certain path. In order to save memory we identify by a half-edge the whole edge. Therefore, each edge represents both half-edges and the vertices of each edge are sorted by their indices in increasing order.

For a better speed up one could extend the information of one edge by two pointers, which point to the next edge in clockwise order seen from $V1$ and $V2$, respectively. This would be similar to the winged-edge data structure ([Bau83]) or to its extension the quad-edge data structure ([GS85]) (figure 2b). This will lead to a traversing speed up and to a higher memory consumption.

It is also simple to get out a real half-edge structure (figure 2c) from this. The only thing that has to be done is to store both directions with a pointer to each other in the described structure ([Mar88]). This means that a half-edge, in a real half-edge structure, would contain a pointer to its twin, its source vertex, a pointer to the face on the right hand, and a pointer to the next following half-edge. Of course, this would increase the memory consumption.

Vertex

Each vertex includes an additional header pointer to one incident edge, usually that edge with the smallest index is chosen. In a half-edge data structure one of the half-edges emanating from it would be chosen. A vertex also holds information of the outside world: points in cartesian or homogeneous coordinates and its degree in the graph structure, i.e. the number of incident edges.

Face

Similar to a vertex, each face includes an additional header pointer to one adjacent edge, usually that edge with the smallest index is chosen.

A face in the DCEL can be open or closed, however it does not contain any other DCEL elements. Usually we have a "zero-face", this is the only existing face of an empty DCEL. In that case the boundary of the face is empty, however, usually the boundary consists of an alternating sequence of vertices and edges. Since each edge defines its vertices explicitly, a connected sequence of edges describes a face sufficient. If this sequence is circular the face is called bounded. *Note:* This definition does not depend on what kind of geometric structure a DCEL represents, a (connected) planar graph in 2D or a polyhedron in 3D.

Example 1. The following example depicted in Figure 3 shows a simple polygon with a dangling edge represented as a DCEL, where A is the outer face an B is the interior one. Note: the dangling edge a has two equal adjacent faces, i.e. ($F1 = F2 = A$).

E	V1	V2	F1	F2	P1	P2
a	1	2	A	A	a	c
b	2	5	A	B	a	e
c	2	3	B	A	b	d
d	3	4	B	A	c	e
e	4	5	B	A	d	b

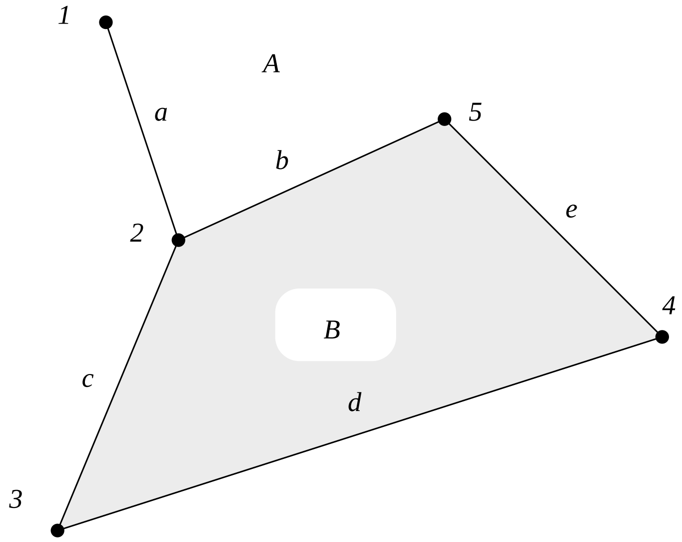


Figure 3: Example 1 with dangling edge.

Example 2. In Figure 4 is shown a more practical but not too complicated example with three faces and without dangling elements.

E	V1	V2	F1	F2	P1	P2
a	1	5	A	B	e	b
b	1	2	B	A	a	f
c	2	3	B	C	b	d
d	3	4	B	C	c	e
e	4	5	B	C	d	f
f	2	5	C	A	c	a

Here we give some important type definitions, which will be used in some of the programs. I is an interface or traits class. It holds a lot of special technical definitions and types necessary for an actual implementation of the given programs (for a detailed information of I see the appendix).
For storing the elements of a DCEL we use the following STL-containers:

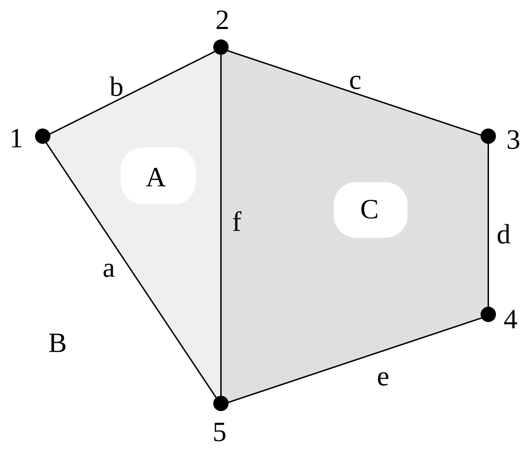


Figure 4: Example 2 without dangling edges.

Type-Definition 1 :

```
typedef vector<Dcel_edge_type<I> >   Edges_container;
typedef vector<Dcel_face_type<I> >   Faces_container;
typedef vector<Dcel_vertex_type<I> > Vertices_container;
```

Also the most important geometric types are predefined in I:

Type-Definition 2 :

```
typedef typename I::Point   Point;
typedef typename I::Segment Segment;
typedef typename I::Polygon Polygon;
```

2.2 DCEL Traverse

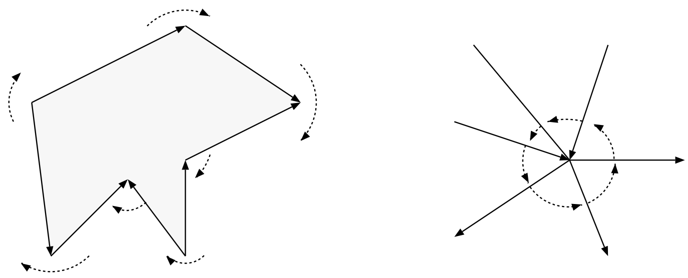
A DCEL can be traversed in several ways:

- Iterating over all edges, vertices, or faces.
- Iterating over all edges incident to a certain vertex or face.
- Walking around a given face and collecting all vertices in a certain order, i.e. clockwise or counter-clockwise.

Since we are iterating over a set of elements or around a vertex or face, we are using the so called *iterators*. The *iterator* concept can be seen as a generalization of pointer concept and was invented in the STL-library ([SL95],[MS96]).

For iterating on the element containers of the DCEL we use the following iterator types:

Type-Definition 3 :



(a) (b)

Figure 5: Traversing around a face (a) and around a vertex (b)

```
typedef typename Edges_container::iterator          edges_iterator;
typedef typename Faces_container::iterator          faces_iterator;
typedef typename Vertices_container::iterator       vertices_iterator;
```

In Figure 5 these iterating processes are illustrated, where each arc-arrow symbolizes exactly on iterator step. Note: The segment arrows represent a directed edge in the DCEL. The beginning element of an iterating process is usually the element in corresponding header list (see Program 1 or Program 2).

Traversing Around Vertices

We assume that the DCEL is built-up correctly (see Section 2.3). With the following definition, it is easy to see how the edges incident on a given vertex can be obtained from the DCEL.

Type-Definition 4 :

```
edges_iterator begin(vertices_iterator v);
                /* returns the header of v */
edges_iterator next(edges_iterator e, vertices_iterator v);
                /* returns the next incident edge to v in counter-clockwise order */
```

By using the header array HV of the vertex list, we obtain the following straight-forward procedure stated in Program 1 for gathering all edges in counter-clockwise order around a given vertex v.

Program 1 (Traverse around vertex v):

```
list<edges_iterator> get_all_edges( vertices_iterator v ) const {
  /* traces the edges counterclockwise about vertex v */
  list<edges_iterator> A;
  edges_iterator a, e= begin(v);
  A.push_back(e);
  for( a= next(e,v); a ≠ e; a= next(a,v) ) A.push_back(a);
  return A;
}
```
program end

Traversing Around Faces

Similar to this procedure the edges enclosing a given face can be obtained from the DCEL using a similar definition.

Type-Definition 5 :

```
edges_iterator begin(faces_iterator v);
               /* returns the header of v */
edges_iterator next(edges_iterator e, faces_iterator f);
               /* returns the next adjacent edge to f in clockwise order */
```

By using the header array HF face lists, we obtain the following straight-forward procedure stated in Program 2 for gathering all edges in clockwise order around a given face f .

Program 2 (Traverse around face f):

```
list<edges_iterator> get_all_edges( face_iterator f ) const {
  /* traces the edges clockwise about face f */
  list<edges_iterator> A;
  edges_iterator a, e= begin(f);
  A.push_back(e);
  for( a= next(e,f); a ≠ e; a= next(a,f) ) A.push_back(a);
  return A;
}
```
program end

Clearly, these procedures run in time proportional to the number of edges incident on vertex v or face f, respectively. Notice that edges are traced counterclockwise about vertex v while they are traced clockwise around face f. It is not possible to trace around a face in counter-clockwise order.

In the boolean operations section we will obtain a procedure *walk_around*, based on Program 2, which traverses a face in the DCEL in an arbitarty direction and puts the vertices as points into a polygon.

2.3 DCEL Construction

We start by using a list of points and a list of edges, represented by a list of integer (*int*) pairs. Each pair contains two indices into the point list defining an edge. This simple structure is also called vertex-edge representation of a (planar) graph.

This input is valid if it represents a planar graph. The validity can be easily checked and can be achieved using any algorithm for intersecting a set of line segments. Thus, from now on we will assume that the input is valid.

The constructing procedure, sketched below, uses another special structure, the so called vertex-to-edge representation that is a graph presentation, which enables us to sort all edges in counterclockwise order around their vertices. It should no be confused with the simple vertex-edge representation.

1. Insert all vertices and edges into the DCEL, without any further information like next edges or adjacent faces.
2. Create vertex-to-edge representation: insert edges and vertices into the vertex-to-edge representation and sort all edges around each vertex in counterclockwise order in this vertex-to-edge representation.

3. Construct the vertex and face cycles using the vertex-to-edge representation and update the header fields.

Now the "built-up" is finished, i.e. no more insertions or deletions are allowed. The DCEL can only be traversed with functions stated in Section 2.2.

Step 1 is straight forward and can be done in one scan without additional memory. We will show that Step 2 can be done in $O(n \log(n))$ time using $O(n)$ space. Step 3 is partitioned in three subparts: constructing the vertex cycles, i.e. to update all consecutive edge pointers in counterclockwise order, constructing the face cycles, i.e. to find for each edge both adjacent faces, and updating the header fields. The third step can be solved easily in $O(n)$ time:

> Given n vertices and f faces, the header fields $HV[1:n]$ (vertex headers) and $HF[1:f]$ (face headers) are filled by a scan of all $V1$ and $F1$ in $O(n)$ time, respectively.

The other parts of step 3 need more effort and as mentioned they are based on the vertex-to-edge representation.

Vertex-To-Edge Representation of a Planar Graph

Let a planar graph $G = (V, E)$ be defined as usual. Let n and m be the number of vertices and edges, respectively.

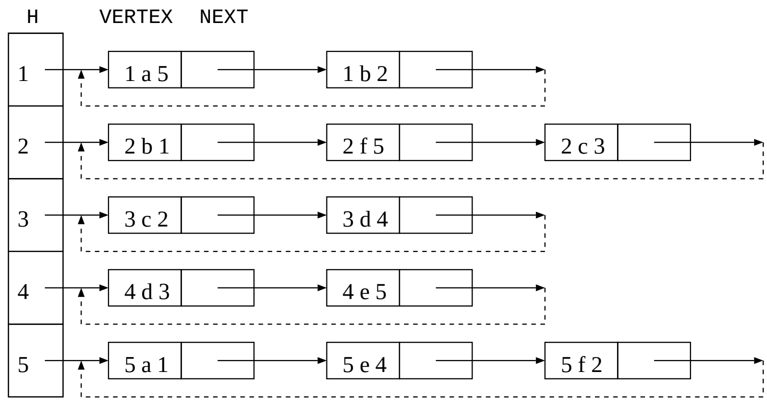


Figure 6: Vertex-to-edge representation in cyclic lists

The *vertex-to-edge* representation of G consists of three arrays:

1. Array `H[1..n]`, where each cell `H[j]` corresponds to vertex $v_j \in V$ and contains a pointer to the first term of the cyclic list of edges incident on v_j. These edges are arranged in the order in which they appear as one proceeds counterclockwise around v_j.

2. Array `VERTEX[1..2m]`, where `VERTEX[i]` contains vertex $v \in V$ and eventually a pointer to an edge incident to v. Clearly, each edge appears twice in this array.

3. Array `NEXT[1..2m]`, where `NEXT[i]` points to next item of the cyclic list of edges. Hence, (`VERTEX[i]`, `NEXT[i]`) is the format of the list nodes of a single linked (cyclic) list as is depicted in figure 6.

One can see immediately that this structure uses $O(n) + 2O(2m) = O(m)$ space. An example of this vertex-to-edge representation is shown in figure 4.

j	H
1	1
2	3
3	6
4	8
5	10

i	VERTEX	edge	NEXT
1	1	a	2
2	1	b	1
3	2	b	4
4	2	f	5
5	2	c	3
6	3	c	7
7	3	d	6
8	4	d	9
9	4	e	8
10	5	a	11
11	5	e	12
12	5	f	10

This representation of a planar graph is precisely that one obtained by the algorithm of Preparata and Hong ([PH77]), which constructs the convex hull of a set of points in three dimensions. Indeed, the surface of a convex polyhedron is topologically a planar graph.

It is clear that filling the vertex-to-edge representation can be done in $O(n)$ time by inserting each edge with its vertices (indices of vertices) into the structure. One insertion can be done in constant time. It remains to show that sorting of the edges around their vertices can be done in $O(n\log(n))$ time.

Theorem 1. *Given a planar graph G in vertex-to-edge representation with n vertices and m edges, then sorting the edges around their vertices in a certain order (e.g. counter-clockwise) requires $O(n\log(n))$ time.*

Proof. Each vertex v_i has $m_i \geq 1$ consecutive edges. Since each edge has exactly two vertices we get

$$\sum_{i=1}^{n} m_i = 2m. \tag{1}$$

Sorting of all consecutive edges around their vertex takes $O(m_i\log(m_i)) = k_i m_i \log(m_i)$ time (with a constant k_i).

$$\begin{aligned}
\sum_{i=1}^{n} O(m_i\log(m_i)) &= \sum_{i=1}^{n} k_i m_i \log(m_i) \\
&= k\sum_{i=1}^{n} m_i\log(m_i) \qquad (\text{with } k = \max\{k_i : i = 1..n\}) \\
&= k\log(m)\sum_{i=1}^{n} m_i \qquad (\text{with } \log(m_i) \leq \log(m)) \\
&= 2mk\log(m) \\
&= O(m\log(m))
\end{aligned}$$

□

A straightforward approach for performing this problem, is going through all vertices and sorting their incident edges in angular order. An easy way for doing comparisons is to compare the edge directions.

Although the vertex-to-edge representation was one of the most commonly used representations for a planar graph, it has the disadvantage that the dual graph, i.e. the graph whose vertices correspond to faces of the original graph, is not readily available. For this one has to develop the *face-to-edge* representation of the original graph, where each face refers to a cyclically ordered list of enclosing edges.

The DCEL is a more convenient representation for this purpose and has other advantages. But at least for building up the DCEL this vertex-to-edge representation is very useful.

Constructing the vertex cycles:

This step updates all edge pointers, that means for both vertices v_1, v_2 of an edge e the next edges e_1, e_2 in counterclockwise order seen vertex v_1, v_2, respectively, have to be found and the corresponding pointers p_1, p_2 have to be updated. In other words: the pointer arrays $P1$ and $P2$ have correctly be filled.

Example 3. In the case of Example 2 (Figure 4) we have:

Input

E	V1	V2
a	1	5
b	1	2
c	2	3
d	3	4
e	4	5
f	2	5

Output

E	V1	V2	P1	P2
a	1	5	e	b
b	1	2	a	f
c	2	3	b	d
d	3	4	c	e
e	4	5	d	f
f	2	5	c	a

In order to have an more readable code we use some type-definitions for using the vertex-to-edge representation (*V2E_rep_base_dcel*):

Type-Definition 6 :

```
/* vertex-to-edge representation: */
typedef typename Dcel_base<I>::V2E_rep_base_dcel V2E_rep_base_dcel;
typedef typename V2E_rep_base_dcel::header_iterator header_iterator;
typedef typename V2E_rep_base_dcel::vertex_iterator vertex_iterator;
```

For constructing the vertex cycles we need the following preconditions: the vertex list and the edge list of the DCEL are initialized and the vertex-to-edge representation $v2e$ is initialized and sorted.

Program 3 (Construct Vertex Cycles):

```
template <class I>
void Dcel_base<I>::construct_vertex_cycles( V2E_rep_base_dcel& v2e ) {
  header_iterator h;            /* header of the vertex-to-edge rep. */
  vertex_iterator v2, v2_begin; /* vertex of the vertex-to-edge rep. */
  vertices_iterator v;          /* DCEL - vertices */
  edges_iterator e;             /* DCEL - edges */
  Dcel_edge_id  u;              /* which edge pointer: P1 or P2 */

  /* for all    headers v1 */
  for( h= v2e.header_begin(); h ≠ v2e.header_end(); h++) {
    v= (vertices_iterator)(*(*h).vertex()).vertex(); /* dcel-vertex */
    (*v)._degree= (*h).size(); /* set the degree of the vertex */
```

```
        /* for all vertices v2 in "header-list" */
        v2_begin= v2e.vertex_begin(h);
        /* fill dcel vertex header */
        (*v)._header= (edges_iterator)(*v2_begin).edge();
        for( v2= v2_begin; v2 ≠ v2e.vertex_end(v2); v2= v2e.vertex_next(v2) )
        { /* edge [v1,v2] */
          e= (edges_iterator)(*v2).edge();
          u= (*e).V1()==v ?  EDGE_1 :  EDGE_2; /* update edge pointer */
          (*e).P(u)= (*v2e.vertex_next(v2)).edge();
        }
        (*e).P(u)= (*v2_begin).edge(); /* closing the circle */
      }
      return;
    }
```
program end

Theorem 2. *Given a sorted vertex-to-edge representation of a graph G with n vertices and m edges, then the vertex cycles of the according DCEL can be constructed in $O(n)$ time.*

Proof. Since we visit each edge exactly two times, the algorithm takes $O(2m) = O(n)$ time. □

Constructing the face cycles:

This step updates all faces in the dcel-structure, such that looking from vertex v_1 to v_2 of an edge e, f_1 will be the left face and f_2 the right one. In other words: the face arrays $F1$ and $F2$ have to be filled correctly. For constructing the face cycles the construction of the vertex cycles is a assumed.

Example 4. In the case of Example 2 (Figure 4) we have:

Input

E	V1	V2	P1	P2
a	1	5	e	b
b	1	2	a	f
c	2	3	b	d
d	3	4	c	e
e	4	5	d	f
f	2	5	c	a

Output

E	V1	V2	F1	F2	P1	P2
a	1	5	A	B	e	b
b	1	2	B	A	a	f
c	2	3	B	C	b	d
d	3	4	B	C	c	e
e	4	5	B	C	d	f
f	2	5	C	A	c	a

The construction of the face cycles is done in two steps: first the face-list is created and filled, then by iterating through the edges their face fields ($F1$ and $F2$) are initialized. This is done in the following way: the first face (s) and the first edge a are taken and it is checked whether both face fields are initialized or not. If at least on of them is not set, then that one is initialized with s and all edges of the boundary of s are also initialized with s. This procedure is repeated until all edges are initialized.

Program 4 (Construct Face Cycles):

```
template <class I>
void Dcel_base<I>::construct_face_cycles() {
    int m= number_of_edges();
    /* create face-list */
    int f= 2 + m − number_of_vertices();
```

```
  _f_list.erase(_f_list.begin(), _f_list.end());
  _f_list.reserve(f+1);
  for(int i= 0; i < f; i++)
    _f_list.push_back( Dcel_face_type<I>(i) );

  faces_iterator s= _f_list.begin();
  edges_iterator a= _e_list.begin();

  for( int k= 1; k≤ 2*m; ) /* k .. edge-counter */ {
    if( (*a).F1() ≠ NULL && (*a).F2() ≠ NULL )
      /* if F1(a) and F2(a) are set, then take next edge a */
      a++;
    else { /* at least one of F1(a) or F2(a) is not initialized */
      /* which face is not initialized ? -> u */
      Dcel_edge_id u= (*a).F1() == NULL ? FACE_1 :  FACE_2;
      k++;
      (*a).F(u)= s;       /* initialize face */
      (*s).header()= a; /*           and its header */
      /* get the corresponding vertex */
      const_vertices_iterator c= (*a).V(u);

      /* initialize all edges on the boundary of face s */
      edges_iterator e= a;
      for( e= (*e).P(u); e ≠ a; e= (*e).P(u), k++ ) {
        u= (*e).V1() == c  ?  VERTEX_2 :  VERTEX_1;
        (*e).F(u)= s; /* initialize face Fu(e) */
        c= (*e).V(u); /* set (new) vertex */
      }
      s++; /* next face */
    }
  } /* end for */
  return;
}
```
program end

Theorem 3. *Given a sorted vertex-to-edge representation of a graph G with n vertices and m edges and the vertex cycles of the according DCEL, then its face cycles can be constructed in $O(n)$ time.*

Proof. Since we visit each edge exactly two times, the algorithm takes $O(2m) = O(n)$ time. □

Summarizing the discussions before we get the following theorem:

Theorem 4. *Given a valid vertex-edge representation with n vertices and m edges, the DCEL can be constructed in $O(n \log(n))$ time using $O(n)$ space.*

Proof. This result follows immediately from Theorem 1, Theorem 2, and Theorem 3. The only additional used space is that of the vertex-to-edge representation, which takes $O(m) = O(n)$ space. □

2.4 DCEL Colorizing

In order to perform boolean operations on simple polygons we consider each polygon to be filled with a color. The colors of the polygon are different (e.g. red and black).

To colorize a DCEL means to add some additional (color-)information to a certain subset of all DCEL elements. All elements that are fully covered by one polygon get its color. From that we can decide to which polygon an element belongs, or by which polygon an element is covered. It is also possible that an element belongs to both polygons. We use different colors of type *Dcel_Color*: red (*RED*), black (*BLACK*), red and black (*RED_AND_BLACK*), and no color (*NO_COLOR*).

The colorizing mechanism, given in Program 6, colorize-s the interior of a simple polygon *pgon* given by a list of its vertices in counterclockwise order. The color is given by the input variable *col* and will be set using the *binary OR* operation (Program 5), that means colors of an element will not be deleted but added.

Program 5 (Colorizing Dcel elements):

```
template <class I>
void Dcel<I>::colorize(Dcel_element_type& el, const Dcel_Color& col) {
   el.set_color( el.color() | col); /* colorize el with col */
}
```
program end

Since the routine compares points (vertices of the polygon with the vertices of the Dcel of type *Point*) this method can lead to robustness problems, when using inexact arithmetic. In that case we have two possibilities:

- to use a geometric predicate *is_inside_polygon(point)* with its implemented epsilon arithmetic.
- to find a unique mapping from points to vertices, that is to use the DCEL vertices as grid points.

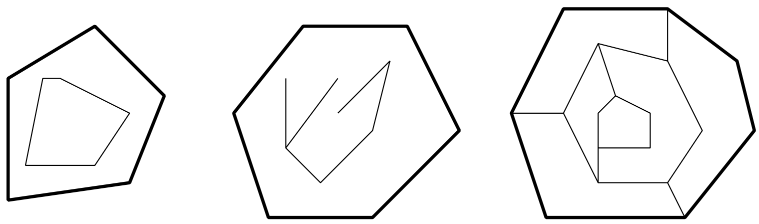
The routine returns *true* iff everything is ok otherwise it returns *false* if for example polygon *pgon* is not valid.

Program 6 (Colorizing):

```
template <class I>
bool Dcel<I>::colorize(const list<Point>& pgon, const Dcel_Color& col) {
  if( pgon.size() < 3 ) return false;  /* polygon not valid: abort */
  Point pt= (*(pgon.rbegin())); /* get last point */
  vertices_iterator v0= find( pt ); /* find first vertex in O(log(n)) time */
  colorize( *v0, col);     /* colorize v0 */

  /* for all vertices of polygon pgon */
  list<Point>::const_iterator itp;
  for(itp= pgon.begin(); itp ≠ pgon.end(); itp++) {
    /* find second vertex 'v1' and edge [v0,v1] */
    edges_iterator  e0= begin(v0);
    edges_iterator a= e0;
    vertices_iterator v1= (*a).opposite_vertex(v0);
    if( (*v1).point() ≠ *itp ) {
      for( a= next(e0,v0); a ≠ e0; a= next(a,v0) ) {
        v1= (*a).opposite_vertex(v0);
        if( (*v1).point() == *itp ) break; /* found next vertex (v1) !! */
```

```
      }
      if( a == e0) return false; /* polygon not valid: abort */
    }
    colorize( *a, col);     /* colorize edge a */;
    colorize( *v1, col);    /* colorize vertex v1 */
    faces_iterator f= (*e).left_face(v1); /* v0 -> v1 ... pgon is CCW */;
    colorize( *f, col);     /* colorize face f */
    v0= v1;
  }
  return true;
}
```
program end



(a) (b) (c)

Figure 7: Examples, where the colorizing mechanism fails.

Note: The Program 6 works only in the case of intersecting two polygons, it does not handle more general cases, like nested faces or nested non-connected subgraphs, depicted in Figure 7, where the polygon drawn with thick line width indicates the face to be colored. To handle these cases we have to put in more effort, which will enlarge the running time significantly. However, for computing boolean operations of two simple polygons the given Program 6 is sufficient and fast.

Theorem 5. *Given a planar graph G (in a DCEL) as the result of the intersection of two simple polygons $pgon_1, pgon_2$, then colorizing the interior of of one of these polygons, say $pgon_1$, in the DCEL takes $O(\log(n) + k)$ time, where n the number of vertices of the DCEL and k the number of vertices of the $pgon_1$.*

Proof. Finding the first vertex of the polygon in the corresponding DCEL takes $O(\log(n))$, if we have a suitable data-structure, e.g. based on a binary tree.

The *for*-loop takes $O(k)$ time iff the inner loop takes $O(1)$ time. Since $pgon_1$ and $pgon_2$ are simple polygons, the do not have self intersections. Therefore we get for the corresponding DCEL

$$\max\{\deg(v) | v \text{ vertex in DCEL}\} = 4.$$

Hence, the inner loop needs constant time and we get $O(\log(n) + k)$ running time. □

In the actual implementation this step requires $O(n+k)$ time in the worst case, but since this procedure is called only twice it is not really expensive compared to other parts of boolean operations.

3 Boolean Operations of two simple polygons

In this section we consider the first step, finding all segment intersections, and the second step, constructing the colored DCEL representing a planar Graph, as already done. For defining the algorithms we need the following type definitions, where I is a given traits class.

Type-Definition 7 :

```
typedef typename I::Point                       Point_2;
typedef typename I::Segment                     Segment_2;
typedef typename I::Output_polygon              Polygon_2;
typedef typename I::Output_polygon_container    Polygon_Container;
typedef typename I::Output_object_container     Output_container;
typedef Bops_dcel<I>                            Dcel;
```

Also the following "global" variables will be used. (Of course, they are not really global in the actual implementation, they are encapsulated in a C++-class.)

Type-Definition 8 :

```
Dcel               dcel;     /* colored DCEL */
Polygon_2          _pgon1;   /* input polygon 1 */
Polygon_2          _pgon2;   /* input polygon 2 */
Output_container   _result;  /* boolean operations result */
```

3.1 Intersecting a Set of Segments

Although we assume that intersecting the boundary of the polygons has already been done, we will say a few words on this topic. This problem can be viewed as a simple intersection problem of a set of line segments and can be solved, for instance by the algorithm of Bentley and Ottman [BO79] in $O(m \log(m) + k)$ time. Where m is the number of segments and k is the number of intersections. Hence, in the worst case we get a $O(n^2)$ complexity, because of the output sensitivity of this problem.

But this intersection problem can also be viewed as a intersection of two sets of line segments, where each of these sets is a set of non-intersecting line segments. This problem is called sometimes red/blue segment intersections, since these both sets are thought to have different colors similar to our approach of performing boolean operations. A very efficient solution to this problem was given in [PS93].

For our algorithm it is necessary to get the result of an intersection algorithm in an appropriate form, that means to know the intersection point or segment and its relationship to the given polygons.

The procedures declared in the following are used for computing the intersection, building up the DCEL, and for coloring the interior of each polygon in the corresponding DCEL.

intersection_result

Intersection(Input_polygon polygon_A, Input_polygon polygon_B)

computes the boundary intersection of two simple polygons A, B and returns the result encapsulated in the type *intersection_result*.

int *get_result(intersection_result result,*
list<Point>& point_list,
list<pair<int,int> >& edge_list)

returns a list of all points and edges extracted from *intersection_result*.

int *get_result_on_A(intersection_result result, list<Point>& point_list)*

returns a list of points ordered on the boundary of the given polygon A with all intersection points.

int *get_result_on_B(intersection_result result, list<Point>& point_list)*

returns a list of points ordered on the boundary of the given polygon B with all intersection points.

3.2 Handling the Special Cases

Concerning the intersection result $I = \partial A \cap \partial B$ of the boundaries of two simple polygons A and B we consider five cases: $I = \emptyset$, $A = B$, $A \subset B$, $B \subset A$, and the general case, where the result of a boolean operation has to be calculated explicitly. Program 7 determines what case occurs. For this purpose the following enumeration type is defined.

Type-Definition 9 :

```
enum Intersection_type {
    is_empty         = 0,
    is_identical     = 1,
    A_is_subset_of_B  = 2,
    B_is_subset_of_A  = 3,
    is_intersection  = 4
};
```

The procedure *calc_intersection_type()* tries to make a first check by using the bounding boxes of the polygons. If the result of this check is not unique, the check is more costly using the polygons themselves.

Program 7 (Determine Intersection Type):

```
template <class I>
Intersection_type
Bops_Simple_Polygons_2<I>::calc_intersection_type(int) const {
     typename I::Bbox a_box= I::get_Bbox(_pgon1);
     typename I::Bbox b_box= I::get_Bbox(_pgon2);

     if( a_box == b_box ) {
       if( _pgon1 == _pgon2 ) return is_identical;
     }
     if( !I::do_overlap(a_box, b_box) ) return is_empty;

     if( I::box_is_contained_in_box(a_box, b_box) )
       return A_is_subset_of_B;
     if( I::box_is_contained_in_box(b_box, a_box) )
       return B_is_subset_of_A;

     typename Polygon_Container::const_iterator it;
     int sum1= 0, n1= _pgon1.size();
     for( it= _pgon1.vertices_begin(); it ≠ _pgon1.vertices_end(); it++)
```

```
      sum1 += I::has_on_bounded_side(_pgon2, *it) ?  +1 :  −1;

    if( sum1 ==  abs(n1) ) {
      int sum2= 0, n2= _pgon2.size();
      for( it= _pgon2.vertices_begin(); it ≠ _pgon2.vertices_end(); it++)
        sum2 += I::has_on_bounded_side(_pgon1,*it) ?  +1 :  −1;
      if( sum2 ==  abs(n2) ) {
        if( sum1 == −n1 && sum2 == −n2 ) /* polygons are separated
      return !_inter_res.size() ? is_empty : is_intersection;
     if( sum1 == n1 && sum2 == -n2 ) /* A is subset B */
          return A_is_subset_of_B;
        if( sum1 == −n1 && sum2 == n2 ) /* B is subset A */
          return B_is_subset_of_A;
      }
    }
    return is_intersection; /* intersections occur */
}
```
program end

Lemma 1. *Given two simple polygons it requires $O(n)$ time to decide their type of intersection. (n is the number of vertices of both polygons.)*

Proof. With respect to Program 7 the proof is straight forward. □

If a special case occur tabular 3.2 gives an overview what the boolean operations will return.

	Intersection $A \cap B$	Difference $A \setminus B$	Union $A \cup B$
$\partial A \cap \partial B = \emptyset$	$\emptyset$	A	A,B
$A = B$	A	$\emptyset$	A
$A \subset B$	A	$\emptyset$	B
$B \subset A$	B	A, hole B	A

3.3 Intersection

The intersection algorithm for two simple polygons is based on the following assumptions:

- The two polygons have been inserted into a colored DCEL.
- All intersections are computed and the DCEL is valid, i.e. its a planar graph, where all elements (faces, edges, and vertices) are defined.
- The colors of faces, edges, and vertices indicate the connection to the inserted polygons, i.e. color *RED_AND_BLACK* indicates that this element belongs to both polygons, *RED* or *BLACK* indicates that this element belongs to the first or second polygon, respectively, and *NO_COLOR* indicate elements that do not belong to any polygon (e.g. the outer face).

Now, it is nearly straightforward how to traverse the DCEL for gathering the solution, i.e. a set of polygons, segments, and points. Principally, we have to find all elements with both colors. Hence, this is done by calling Program 10 (*get_colored_elements(RED_AND_BLACK)*), which uses Program 8 and Program 9 for walking around the faces.

Program 8 (similar to Program 2 in Section 2.2) traverses a face of the DCEL, puts the vertices as points in into a list in clockwise order, or in counter-clockwise order in the case of the outer face. The return value is the number of visited vertices. This list of points defines a polygon. Additionally, the visited edges and vertices will be marked, such that a calling program is able to decide if this element was visited before or not.

Program 8 (Walking Around a Face):

```
int walk_around(faces_iterator face, list<Point_2>& result) const {
  int n= 0; /* number of vertices */
  edges_iterator a= dcel.begin(face);
  mark(a);
  vertices_iterator v0= ((*a).F1() == face) ?  (*a).V1() :  (*a).V2();
  result.push_back(dcel.point(v0)); n++;
  mark(v0);
  for(edges_iterator e= dcel.next(a,face); e ≠ a; e= dcel.next(e, face) ) {
      mark(e);
      vertices_iterator v= (*e).opposite_vertex( v0 );
      mark(v);
      result.push_back(dcel.point(v)); n++;
      v0= v;
  }
  return n;
}
```
program end

The result of this program is a list of points, Program 9 converts it into a polygon with counter-clockwise order, or respectively with clockwise order, if the second parameter *ccw* is set to *false*.

Program 9 (Convert a Face into a Polygon):

```
Polygon_2 walk_around(faces_iterator face, bool ccw = true) const {
  list<Point_2> pt_list;
  if( walk_around( face, pt_list) )
    return ccw ?  Polygon_2(pt_list.rbegin(), pt_list.rend()) :
                 Polygon_2(pt_list.begin(),  pt_list.end());
  return Polygon_2();
}
```
program end

After finding all faces (i.e. polygons) we have only to look through the non visited edges and vertices in order to gather all edges (i.e. segments) and vertices (i.e. points) having both colors.

Program 10 (Find all elements with color *search_color*):

```
void get_colored_elements( const Dcel_Color& search_color ) {
  /* get all faces, i.e. polygons */
  Polygon_2  pgon;
  for(face_iterator f= dcel.face_begin(); f ≠ dcel.face_end(); f++) {
    if( (*f).color() == search_color ) {
      pgon= walk_around(f);
      add_to_result(pgon);
    }
  }
  Segment_2 *seg;  /* find all edges, i.e. segments */
  for( edge_iterator e= dcel.begin(); e ≠ dcel.end(); e++ ) {
    if( is_unmarked(e)  && (*e).color() == search_color ) {
      seg= new Segment_2( dcel.point((*e).V1()), dcel.point((*e).V2()) );
```

```
      add_to_result(*seg);
      mark( (*e).V1() );
      mark( (*e).V2() );
      delete seg;
    }
  }
  /* find all vertices, i.e. points */
  for(vertex_iterator v= dcel.vertex_begin(); v ≠ dcel.vertex_end(); v++) {
    if( is_unmarked(v) && (*v).color() == search_color )
       add_to_result(dcel.point(v));
  }
  return;
}
```
program end

Theorem 6. *Given a colored DCEL representing a planar graph G and a color col, then gathering all elements having color col needs $O(n)$ time and takes $O(n+m) = O(n)$ additional space, where n respective m are the number of vertices and edges of G.*

Proof. Since each edge has at maximum two adjacent faces the first loop requires $O(2m)$ operations in the worst case. Thus, we get for all three loops a worst case time complexity of $O(3m+n) = O(n)$.

The additional space is required for marking the elements. This marking information is held in two arrays with size n and m, respectively. □

This result implies immediately the main theorem of this section:

Theorem 7. *Given a colored DCEL representing a planar graph G as the result of the intersection of the boundary of two simple polygons $pgon_1, pgon_2$, then performing the intersection of the polygons interior can be done in $O(n)$ time with using $O(n)$ additional memory.*

3.4 Difference

The difference algorithm of two simple polygons is based on the same assumptions as the intersection algorithm (see 3.3). After handling the special cases of different intersection-types (see Section 3.2), the difference is performed, similar to the intersection, by gathering all elements having the color of the first polygon, say *RED*, by calling Program 10 (*get_colored_elements(RED)*).

Hence, we get the according theorem for the difference operation.

Theorem 8. *Given a colored DCEL representing a planar graph G as the result of the intersection of the boundary of two simple polygons $pgon_1, pgon_2$, then performing the difference of the polygons interior can be done in $O(n)$ time with using $O(n)$ additional memory.*

3.5 Union

The union of two simple polygons can also be seen as a gathering of all elements with a certain color, similarly to intersection and difference. In the case of performing the union this color will be *NO_COLOR*, i.e. only the outer boundary and the inner holes have to be found. However, since the result should have a certain order, namely the outer boundary should be in front of the resulting list, we have to use a slightly modified procedure.

We consider the special cases to be handled as stated in Section 3.2. The outer boundary is found be taking the point with the minimal x-coordinate and finding its incident edge with maximal slope. Hence, this edge has to part of the outer boundary. This algorithm is stated in Program 11.

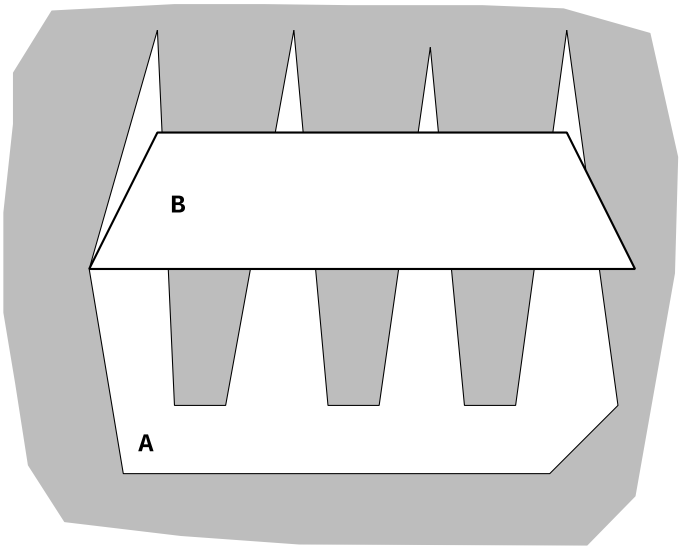


Figure 8: Union of two simple polygons A and B. The colored faces (outer boundary and holes) indicate the result.

Program 11 (Find the outer boundary):

```
faces_iterator outer_boundary() const {
  typedef typename Polygon_2::Vertex_const_iterator polygon_vertex_iterator;
  polygon_vertex_iterator p1it = _pgon1.left_vertex();
  polygon_vertex_iterator p2it = _pgon2.left_vertex();
  polygon_vertex_iterator pit  = (*p1it).x() < (*p2it).x() ?  p1it :  p2it ;
  Point_2 pt = *pit;   /* the most left point of polygons A and B */
  vertex_iterator vit = dcel.find(pt);
      /*  find the edge lit starting at vit with largest slope */
  edges_iterator ait= dcel.begin(vit);
  edges_iterator lit= ait, eit;
  for(eit= dcel.next(ait,vit); eit ≠ ait; eit= dcel.next(eit, vit) ) {
    Point_2 pt1= dcel.point(vit);
    Point_2 pt2= dcel.point((*eit).opposite_vertex(vit) ) ;
    Point_2 pt3= dcel.point((*lit).opposite_vertex(vit) ) ;
    if( is_leftturn(pt1, pt3, pt2) ) lit = eit;
  }
  return (*lit).V1() == vit ?  (*lit).F1() :  (*lit).F2();
}
```
program end

Lemma 2. *Finding the outer boundary of a colored DCEL representing a planar graph, which is a result of the intersection of the boundary of two simple polygons can be done in $O(\log(n))$.*

Proof. To find the point *pt* with minimal x-coordinate of a polygon can be done in $O(\log(n))$ time using a suitable data structure. Also finding the concerning vertex of *pt* in the DCEL can be done in $O(\log(n))$ time using e.g. a binary tree. Since the maximum degree of a vertex is four (see proof of Program 5), finding the edge with largest slope requires constant time. Thus, in summary the algorithm stated in Program 11 can be done in $O(\log(n))$ time. □

In the case of not using data structures that allow a $O(\log(n))$ searching, the algorithm can be performed in $O(n)$ time doing sequential searching, anyway.

In general the result of a union of two simple polygons consists of a polygon describing the outer boundary and a set of polygons describing the inner boundary, the holes, which are oriented clockwise. (For example in Figure 8 there are three holes.) Hence, first we have to find the outer boundary (Program 11) and to convert it into a polygon with counter-clockwise order

Program 12 (Perform Union):

```
void Union::perform() {
  faces_iterator out= outer_boundary();
  add_to_result( walk_around(out, false) );

  for(face_iterator f= dcel.face_begin(); f ≠ dcel.face_end(); f++)
    if( f ≠ out && (*f).color() == NO_COLOR )
      add_to_result( walk_around(f, false) );
}
```

program end

Lemma 2 implies immediately the main theorem of this section:

Theorem 9. *Given a colored DCEL representing a planar graph G as the result of the intersection of the boundary of two simple polygons $pgon_1, pgon_2$, then performing the union of the polygons interior can be done in $O(n)$ time with using $O(n)$ additional memory.*

Proof. With Lemma 2 and the same argument as given in the proof of Theorem 6 that each edge is visited at most twice leads to the assertion. □

4 Conclusion

Given two simple polygons and the intersection of their boundaries as a planar graph in a simple vertex-edge representation, we have shown that the corresponding colored DCEL can be constructed from this in $O(n\log(n))$ time using $O(n)$ additional memory (the vertex-to-edge representation) and using this DCEL that boolean operations can performed in $O(n)$ time using $O(n)$ additional space.

What remains is to

- make the algorithm more robust, e.g. for using non-exact arithmetic: Although this problem depends mainly on the intersection algorithm it can cause problems in finding vertices and colorizing.
- put more effort in the colorizing mechanism in order to be able to colorize general planar graphs or maps.
- extend the boolean operations, such that one can compute the exact difference of two simple polygons or the union of a set of polygons.
- make boolean operations closed, that means the result of an operation should be used as input for the next one. An approach for performing that is to use planar maps as a generalization of simple polygons.

Acknowledgments

The author would like to thank Sabine Stifter and Petru Pau for their helpful suggestions and Peter Stadelmeyer for fruitful discussions, which lead to many improvements.

References

[Bau83] B. G. Baumgart. A polyhedron representation for computer vision. *AFIPS Conference Proc.*, vol. 44:589–596, 1983.

[BO79] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, C-28:643–647, September 1979.

[FGK+96] Andreas Fabri, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr. The cgal kernel: A basis for geometric computation. In M. C. Lin and D. Manocha, editors, *ACM Workshop on Applied Computational Geometry*, pages 191–202, Philadelphia, Pennsylvenia, May, 27–28 1996. Lecture Notes in Computer Science 1148.

[GS85] Leonidas Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and computation of voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, April 1985.

[Mar88] Martti Mäntylä. *An Introduction to Solid Modeling*. Computer Science Press, Rockville, MD, 1988.

[MN94] Kurt Mehlhorn and Stefan Naeher. Implementation of a sweep line algorithm for the straight line segment intersection problem. Technical report, IB-B951220 Saarbruecken, Saarbruecken, 1994.

[MP78] Muller and Preparata. Finding the intersection of two convex polyhedra. *Theoretical Computer Science*, 7:217–236, 1978.

[MS88] Harry G. Mairson and Jorge Stolfi. Reporting and counting intersections between two sets of line segments. In R. A. Earnshaw, editor, *Theoretical Foundations of Computer Graphics and CAD*, volume F40 of *NATO ASI*, pages 307–325. Springer-Verlag, 1988.

[MS96] David R. Musser and Atul Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley, 1996.

[PH77] F. P. Preparata and S. J. Hong. Convex hulls of finite sets of points in two and three dimensions. *Communications of the ACM*, 20:87–93, 1977.

[PS85] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, 1985.

[PS93] Larry Palazzi and Jack Snoeyink. Counting and reporting red/blue segment intersections. Technical Report TR-93-32, ftp.cs.ubc.ca, oct 1993.

[SL95] Alexander Stepanov and Meng Lee. The standard template library. Technical report, HP Labs, Palo Alto, October 1995.

[Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2nd edition, 1991.

A Implementation of the DCEL

#include <CGAL/bops_traits.h>

In a boolean operations traits class some types and functions needed for the computation of boolean operations are defined.

A.1 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 homogeneous 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 do boxes do overlap.

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 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.box_is_contained_in_box(Bbox bbox1, Bbox bbox2)* returns true iff *bbox1* is subset or equal *bbox2*.
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*.

A.2 The colors of the DCEL (*CGAL__Dcel_Color*)

#include <CGAL/bops_dcel_color.h>

Definition
The internal structure of the DCEL deals with colors, i.e. each vertex, edge, or face may be colored with one or both of two colors (*CGAL__RED*, *CGAL__BLACK*).

Creation

CGAL__Dcel_Color col((CGAL__Dcel_Color c));

CGAL__Dcel_Color col(int c);

Operations

int	*int(col)*	converts to int()
CGAL__Dcel_Color	*col | c*	binary *or*–operator

Constants

const CGAL__Dcel_Color	*CGAL__NO_COLOR = 0;*
const CGAL__Dcel_Color	*CGAL__UNCOLORED = 0;*
const CGAL__Dcel_Color	*CGAL__RED = 1;*
const CGAL__Dcel_Color	*CGAL__BLACK = 2;*
const CGAL__Dcel_Color	*CGAL__RED_AND_BLACK = 3;*
const CGAL__Dcel_Color	*CGAL__TWICE_COLORED = 3;*

A.3 The DCEL Element (*CGAL__Dcel_element_type*)

#include <CGAL/bops_dcel_element.h>

Creation

CGAL__Dcel_element_type f(CGAL__Dcel_Color col = CGAL__NO_COLOR);
default constructor

CGAL__Dcel_element_type f(int ind, CGAL__Dcel_Color col = CGAL__NO_COLOR);
constructs a new element with setting a new index number *ind*.

Operations

CGAL__Dcel_Color	*f.color()*	returns the actual color
CGAL__Dcel_Color	*f.set_color(CGAL__Dcel_Color c)*	sets the color of *f*
int	*f.index()*	returns the actual index number

A.4 The DCEL vertex (*CGAL__Dcel_vertex_type*)

#include <CGAL/bops_dcel_vertex.h>

Creation

CGAL__Dcel_vertex_type f; default constructor

CGAL__Dcel_vertex_type f(CGAL_Object pt,
int ind,
CGAL__Dcel_Color col = CGAL__NO_COLOR)
constructs a new vertex with setting a new index number *ind* and a new Point *p* (*CGAL_Object*&).

Operations

CGAL__Dcel_edge	*f.header()*	returns the vertex header info

A.5 The DCEL face (*CGAL__Dcel_face_type*)

#include <CGAL/bops_dcel_face.h>

Creation

CGAL__Dcel_face_type f; default constructor

CGAL__Dcel_face_type f(int ind, CGAL__Dcel_Color col = CGAL__NO_COLOR);
constructs a new face with setting a new index number *ind*.

Operations

CGAL__Dcel_edge	*f.header()*	returns the face header info

A.6 The DCEL edge (*CGAL__Dcel_edge_type*)

#include <CGAL/bops_dcel_edge.h>

Creation

CGAL__Dcel_edge_type e; default constructor

CGAL__Dcel_edge_type e(CGAL__Dcel_vertex v1, CGAL__Dcel_vertex v2, CGAL__Dcel_Color col = CGAL__NO_COLOR)

construction by two vertices and a color

CGAL__Dcel_edge_type e(CGAL__Dcel_edge_type e);

copy constructor

CGAL__Dcel_edge_type e(CGAL__Dcel_edge_type e, int ind);

copy constructor which sets also a new edge index *ind*.

Operations

CGAL__Dcel_vertex	*e.V1()*	returns *V1*.
CGAL__Dcel_vertex	*e.V2()*	returns *V2*.
CGAL__Dcel_face	*e.F1()*	returns *F1*.
CGAL__Dcel_face	*e.F2()*	returns *F2*.
CGAL__Dcel_edge	*e.P1()*	returns *P1*.
CGAL__Dcel_edge	*e.P2()*	returns *P2*.
CGAL__Dcel_vertex	*e.opposite_vertex(CGAL__Dcel_vertex v)*	
CGAL__Dcel_face	*e.left_face(CGAL__Dcel_vertex v1)*	
CGAL__Dcel_face&	*e.left_face(CGAL__Dcel_vertex v1)*	
CGAL__Dcel_vertex	*e.V(CGAL__Dcel_edge_id i)*	returns vertex i (with $i \in \{1,2\}$)
CGAL__Dcel_face	*e.F(CGAL__Dcel_edge_id i)*	returns face i (with $i \in \{1,2\}$)
CGAL__Dcel_edge	*e.P(CGAL__Dcel_edge_id i)*	returns edge i (with $i \in \{1,2\}$)

A.7 The DCEL base class (*CGAL__Dcel_base<I>*)

#include <CGAL/bops_dcel_base.h>

Definition

This template class is parameterized by a traits class <*I*>(see section A.1).

Types

typedef I::edge_iterator	*edge_iterator;*	iterator on a container of type *CGAL__Dcel_edge*.
typedef I::face_iterator	*face_iterator;*	iterator on a container of type *CGAL__Dcel_face*.
typedef I::vertex_iterator	*vertex_iterator;*	iterator on a container of type *CGAL__Dcel_vertex*.
typedef I::const_points_iterator	*const_points_iterator;*	const iterator on a container of type *Point*.
typedef I::const_vertices_iterator	*const_vertices_iterator;*	const iterator on a container of type *CGAL__Dcel_vertex*.
typedef I::const_edges_iterator	*const_edges_iterator;*	const iterator on a container of type *CGAL__Dcel_edge*.

Creation

CGAL__Dcel_base<I> L; default constructor

CGAL__Dcel_base<I> L(CGAL__Dcel_base dl);

copy constructor

void *L.construct_face_cycles(void)*

construction of face cycles
Precondition: vertex list *_v_list* is initialized: edge list *_e_list* is initialized: V1, V2 vertex-to-edge representation is initialized and sorted

void *L.construct_face_cycles(void)*

construction of face cycles
Precondition: face list *_f_list* is not initialized, it will be cleared and filled edge list *_e_list* is initialized: V1, V2, P1, P2

edge_iterator *L.insert(CGAL__Dcel_edge_type et)*

vertex_iterator *L.insert(CGAL__Dcel_vertex_type vt)*

face_iterator *L.insert(CGAL__Dcel_face_type ft)*

Operations

int	*L.number_of_vertices()*	returns the number of vertices in the DCEL
int	*L.number_of_edges()*	returns the number of edges in the DCEL
int	*L.number_of_faces()*	returns the number of faces in the DCEL

edge_iterator	*L.begin()*	returns the begin-iterator of the edge-list
edge_iterator	*L.end()*	returns the end-iterator of the edge-list
const_vertices_iterator	*L.vertex_begin()*	returns the begin-iterator of the vertex-list
const_vertices_iterator	*L.vertex_end()*	returns the end-iterator of the vertex-list
face_iterator	*L.face_begin()*	returns the begin-iterator of the face-list
face_iterator	*L.face_end()*	returns the end-iterator of the face-list
edge_iterator	*L.begin(vertex_iterator v)*	returns header of vertex v
edge_iterator	*L.next(edge_iterator e, vertex_iterator v)*	returns next edge in counterclockwise order about vertex v
edge_iterator	*L.begin(face_iterator f)*	return header of face f
edge_iterator	*L.next(edge_iterator e, face_iterator f)*	returns next edge in clockwise order about face f
list<edge_iterator>	*L.get_all_edges(vertex_iterator v)*	traces the edges counterclockwise about vertex v
list<edge_iterator>	*L.get_all_edges(face_iterator f)*	traces the edges clockwise about face f
edge_iterator	*L.insert(CGAL__Dcel_edge_type<I> et)*	
const_vertices_iterator	*L.insert(CGAL__Dcel_vertex_type<I> vt)*	
const_points_iterator	*L.insert(Point pt)*	
face_iterator	*L.insert(CGAL__Dcel_face_type<I> ft)*	

A.8 The DCEL class (*CGAL__DCEL<R>*)

#include <CGAL/bops_dcel.h>

Types

typedef pair<int,int>	*epair;*	global typedef of edge-pair
typedef vector<CGAL__Dcel_edge_type> ::const_iterator	*edge_iterator;*	
typedef vector<CGAL__Dcel_face_type> ::const_iterator	*face_iterator;*	
typedef vector<CGAL__Dcel_vertex_type>::const_iterator	*vertex_iterator;*	

Creation

CGAL__DCEL<R> dcel; — default constructor

CGAL__DCEL<R> dcel(CGAL__Dcel<R> l);

copy constructor

CGAL__DCEL<R> dcel(list< epair > eds, list< CGAL__Point_2<R> > pts);

constructs dcelby a set of points and its edge definitions. Each epair defines an edge by the indices of two points of the point list pts. Note: pts is of type list< CGAL_Point_2<R> >.

void dcel.insert(list< epair > eds, list< CGAL__Point_2 > pts)

constructs dcelby a set of points and its edge definitions. Each epair defines an edge by the indices of two points of the point list pts. Note: pts is of type list< CGAL_Point_2<R> >.

Operations

void dcel.colorize(CGAL__Dcel_Element_type el)

colorizes an element (vertex, face, or edge).

bool dcel.colorize(list< CGAL__Point_2<R> > pgon, CGAL__Dcel_Color col)

Colorizes all elements of dcelbeing inside of the simple polygon pgon with color col. pgon is defined by a list of points where the connection between the last and the first point is also assumed as an polygon edge.

CGAL__Point_2<R> dcel.point(vertex_iterator v) — returns the point represented by vertex v in dcel.

vertex_iterator dcel.find(CGAL__Point_2<R> pt)

returns the vertex representing point pt in dcel

Implementation Remarks

The DCEL is implemented by three vectors of vertices, edges, and faces, respectively. Therefore, all iterators get invalid after insertion or deletion. However, this DCEL is thought to be used for gathering information after building up.

A.9 Boolean Operations of Two Simple Polygons in the 2-dimensional Euclidean Space (*CGAL_Bops_Simple_Polygons_2<R,Polygon_Container>*)

#include <CGAL/bops_simple_polygons_2.h>

This is the base class for all boolean operations (intersection, difference, and union). It provides the most important features for this operations, and in the derived classes has only be defined a function *perform()*.

Types

typedef CGAL_Point_2< R >	*Point_2;*	
typedef CGAL_Segment_2< R >	*Segment_2;*	
typedef CGAL_Polygon2< R, Polygon_Container >	*Polygon_2;*	
typedef list<CGAL_Object>::const_iterator	*result_iterator;*	iterator on the operation result
typedef list<CGAL_Object>::size_type	*size_type;*	
typedef CGAL__Dcel< R >	*Dcel;*	
typedef CGAL__Dcel< R >::face_iterator	*face_iterator;*	
typedef CGAL__Dcel< R >::edge_iterator	*edge_iterator;*	
typedef CGAL__Dcel< R >::vertex_iterator	*vertex_iterator;*	
typedef CGAL__intersecting_polygons<R, Polygon_Container>	*Intersect_Polygons;*	

Creation

CGAL_Bops_Simple_Polygons_2<R,Polygon_Container> bops;

default constructor

CGAL_Bops_Simple_Polygons_2<R,Polygon_Container> bops(Polygon_2 pgon1, Polygon_2 pgon2);

constructor by two simple polygons

Operations

bool	*bops.operation()*	performs the operation
result_iterator	*bops.begin()*	returns begin of the operation result
result_iterator	*bops.end()*	returns end of the operation result
size_type	*bops.size()*	returns result size
bool	*bops.empty()*	returns true if the result is empty
bool	*bops.do_intersect()*	returns true if the input polygons do intersect
virtual void	*bops.perform(void)*	this virtual function has to be overloaded, by the child class. *perform()* defines the special boolean operation, i.e. intersection, union, or difference
int	*bops.calc_intersection_type(void)*	calculates and returns the intersection type of the two simple polygons:

n	type
0	$A \cap B = \emptyset$
1	$A = B$
2	$A \subset B$
3	$B \subset A$
4	partial intersection

void *bops.create_dcel(void)* creates the DCEL

Traversing

int *bops.walk_around(face_iterator face, list<Point_2>& result)*

traverses a face in the DCEL and puts the vertices as points into a list. These points define a polygon. Additionally, the visited edges will be marked in the marked-vector. The size of the marked-vector should be at least the number of edges in the DCEL, i.e. *marked.size()* $\geq$ *dcel.number_of_edges* (The result is always be ordered clockwise, except on traversing the outer boundary.)

Polygon_2 *bops.polygon_2(list<Point_2> pt_list, bool reverse = true)*

converts a point-list into a polygon, by default in counter clockwise order. (if reverse == false in clockwise order).

Polygon_2 *bops.walk_around(face_iterator face, bool ccw = true)*

traverses a face in the DCEL and puts the vertices as points into a polygon. Additionally, the visited edges will be marked in the marked-vector. The size of the marked-vector should be at least the number of edges in the DCEL, i.e. *marked.size()* $\geq$ *dcel.number_of_edges*

void *bops.get_colored_elements(CGAL__Dcel_Color search_color)*

finds all elements of the DCEL having the given color *search_color* and puts them into the result.

faces_iterator *bops.outer_boundary()* returns *a face_iterator* to the outer boundary of the DCEL.

Marking

void *bops.marked_vector_init()* initializes the marked vectors, i.e. creates a marked vector for edges and one for vertices, each with the appropriate size.

void *bops.unmark()* all markers for vertices and edges will be set false.

void *bops.mark(vertex_iterator v)* marks vertex v.

void	*bops.unmark(vertex_iterator v)*	
		unmarks vertex v.
bool	*bops.is_marked(vertex_iterator v)*	
		returns true if vertex v is marked.
bool	*bops.is_unmarked(vertex_iterator v)*	
		returns true if vertex v is unmarked.
bool	*bops.is_unmarked(edge_iterator e)*	
		returns true if edge e is unmarked.
bool	*bops.is_marked(edge_iterator e)*	
		returns true if edge e is marked.
void	*bops.mark(edge_iterator e)*	marks edge e.
void	*bops.unmark(edge_iterator e)*	
		unmarks edge e.