Advances in Engineering Software 64 (2013) 11–19

Contents lists available at SciVerse ScienceDirect

Advances in Engineering Software

journal homepage: www.elsevier.com/locate/advengsoft

A simple algorithm for Boolean operations on polygons

Francisco Martínez *, Carlos Ogayar, Juan R. Jiménez, Antonio J. Rueda

Departamento de Informática, Universidad de Jaén, Campus Las Lagunillas, s/n, 23071 Jaén, Spain

ARTICLE INFO

Article history:
Received 8 June 2012
Received in revised form 21 January 2013
Accepted 14 April 2013
Available online 7 June 2013

Keywords:
Boolean operations polygons
Polygon clipping
Polygon overlay
Computational geometry
Computer graphics
Geometric operations

ABSTRACT

In this paper a simple and efficient algorithm for computing Boolean operations on polygons is presented. The algorithm works with almost any kind of input polygons: concave polygons, polygons with holes, several contours and self-intersecting edges. Important topological information, as the holes of the result polygon, is computed.

1. Introduction

Boolean operations on polygons play an important role in different applied fields such as Computer Graphics, GIS or CAD.

Many algorithms have been developed for polygon clipping, in which several polygons are clipped against a clipping polygon. However, these algorithms often impose strong restrictions on the clipping polygon. For example, some algorithms only work with convex [1,2] or rectangular [3] clipping polygons.

For the general case of Boolean operations on polygons, i.e., concave polygons with holes, several contours and self-intersections less solutions are available. Greiner and Hormann propose a simple and elegant algorithm [4], but it does not properly deal with degenerate cases. However, the algorithm has been extended to deal with degenerate cases [5] and even to cope with holes and polygons with several contours [6]. Unfortunately, this last algorithm is not so elegant as the original one.

Rivero and Feito [7] and Peng et al. [8] present simple methods, from a mathematical point of view, for computing Boolean operations on polygons that are based on the simplex theory proposed by Feito [9]. Unfortunately, these methods are difficult to implement, inefficient and produce a result polygon with almost null topological information—just a list of unconnected edges.

From the field of Computational Geometry some algorithms have been proposed for the more general problem of the overlay of two subdivisions of the plane [10,11]. These algorithms extend the one by Bentley and Ottmann [12] for computing the intersection points in a set of line segments using the plane sweep technique. They compute the overlay in $O((n+k)\log n)$ time, where n denotes the total number of edges of the input overlays and k is the number of intersections between their edges. The algorithm described in [10] does not deal with some degenerate cases, as it is the case of input polygons with overlapping edges. Vatti [13] has also presented a, less efficient algorithm, based on the plane sweep paradigm. In [14] the sweep-line technique is used to compute a data structure based on trapezoidal maps that allows to clip polygons, a drawback of this approach is the $O(n^2)$ size of the data structure.

In [15] we have also extended the algorithm by Bentley and Ottmann [12], but only for computing Boolean operations on polygons in $O((n+k)\log n)$ time. Because we solve a simpler problem our algorithm is also quite simpler and easier to understand than [10,11]. However, the algorithm does not compute which of the result polygon contours are holes, this information is neither computed by the other algorithms for computing Boolean operations [4,7,8,13]. Owing to this information is important in some applications—for instance, it is needed for computing a polygon area—, in this paper we present a modified algorithm that computes the holes of the result polygon. Furthermore, the new algorithm uses a simpler criterion for selecting and joining the edges belonging to the result polygon.

The paper is structured as follows. In Section 2 the algorithm and the format of the result polygon are sketched. Sections 3–5 give a detailed description of the algorithm. Section 6 analyzes its running time and Section 7 explains how the degenerate and special cases are dealt with. Section 8 explains some optimizations,

* Corresponding author. Tel.: +34 953212887.
E-mail address: fmartin@ujaen.es (F. Martínez).

0965-9978/$ - see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.advengsoft.2013.04.004

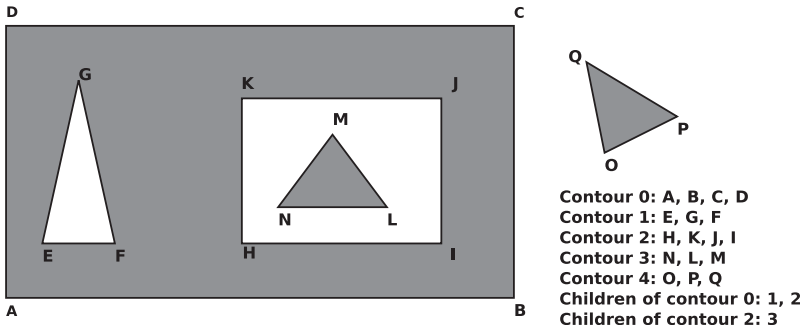

Fig. 1. Polygon specification.

in Section 9 an experimental comparison with Vatti's algorithm is made. Finally, Section 10 draws some conclusions.

2. Foundations

In this section the algorithm for computing Boolean operations on polygons is outlined. First, the format of the result polygon is explained.

2.1. Polygon specification

Our algorithm works with polygons that consist of several contours. Each contour is a simple polygon and the edges of the contours are interior disjoint. Next, we make the following definitions:

- *External contour*: it is a contour not included in any of the other polygon contours.
- *Internal contour*: it is a contour included in at least one of the other polygon contours.
- *Parent contour*. Given an internal contour C, let P be the contour equals to the intersection of all the polygon contours that contain C—i.e., the smaller contour that contains C. Then, we say that P is the *parent contour of* C and that C is a *child contour of* P

A polygon consisting of several contours can be represented as follows:

- The vertices of the contours included in an even number of contours are listed in counter-clockwise order.
- The vertices of the contours included in an odd number of contours are listed in clockwise order.
- For every parent contour its children contours are listed.

For example, Fig. 1 shows a polygon represented this way. Given this representation the area of a polygon can be computed as the sum of the signed areas of its contours. Note that the amount of storage required by this representation is linear in the number of vertices of the polygon.

The result polygon computed by our algorithm follows this representation. However, our algorithm does not impose constraints about the orientation of the vertices of the input polygons. The contours of the input polygons can be listed clockwise or counter-clockwise, regardless they are included in an odd or even number of contours. It is neither necessary to specify child contours. The only restriction imposed by our algorithm on an input polygon is that two edges of the same polygon cannot overlap.

2.2. Outline of the algorithm

The boundary of the result of a Boolean operation on two polygons consists of those portions of the boundary of each polygon that lie inside—or outside, depending on the kind of operation—the other polygon, see Fig. 2. For example, the intersection consists of those portions lying inside, whereas the union consists of those portions lying outside.

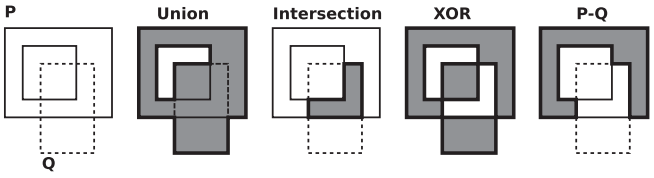

Fig. 2. Boolean operations on polygons.

We propose the following scheme to compute a Boolean operation between two polygons:

1. Subdivide the edges of the polygons at their intersection points.
2. Select those subdivided edges that lie inside—or outside—the other polygon.
3. Join the selected edges to form the contours of the result polygon and compute the child contours.

The algorithm is based on the following idea: after subdividing polygons edges at their intersection points—see Fig. 3—, a subdivided edge lies inside or outside the other polygon and therefore it belongs or not to the result polygon.

In the next sections the algorithm, which has two stages, is described. In the first stage the polygon edges are subdivided and the edges in the result polygon are selected. In the second stage the selected edges are joined to form the contours of the result polygon and the child contours are computed.

2.3. Computing the child contours

In this subsection we describe the approach used to compute the child contours. Given a contour C and its bottom left vertex v we shoot a vertical ray from v that goes downward. Let $e \in C_2$ be the first edge of the polygon crossed by the ray. e represents an

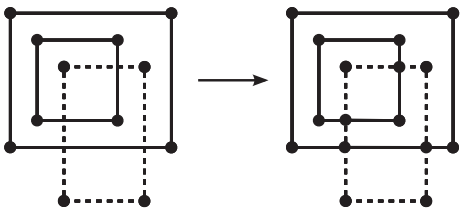

Fig. 3. Subdividing polygon edges at their intersection points.

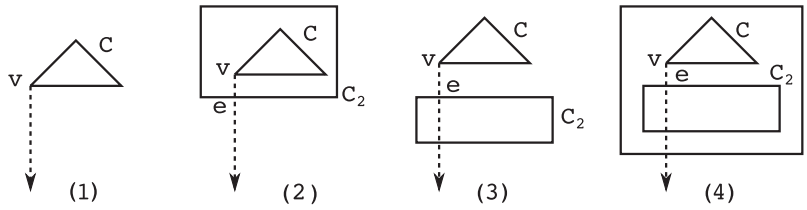

Fig. 4. Cases for determining the parent contour of *C*.

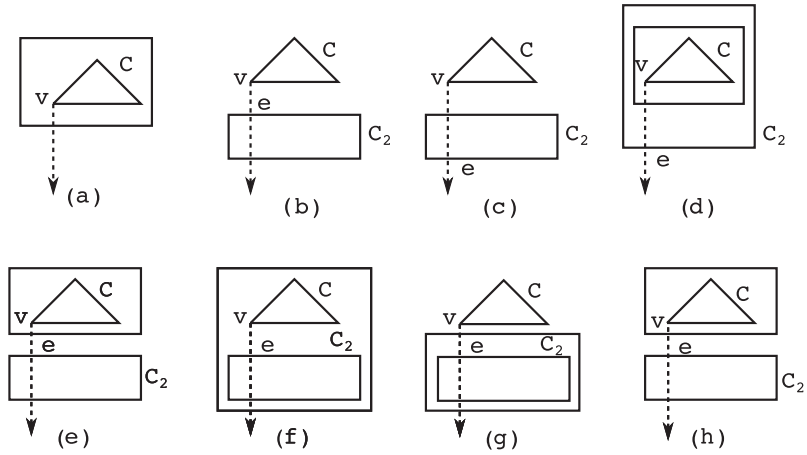

Fig. 5. Impossible cases.

inside-outside or outside-inside transition into C_2 for a vertical ray that starts below C_2 and goes upward.

Then, we can distinguish four possible cases for determining the parent contour of *C*, see Fig. 4:

1. *e* does not exist. Then, it can be proved that *C* is an external contour.
2. *e* represents an outside-inside transition into C_2. Then, it can be proved that C_2 is the parent contour of *C*.
3. *e* represents an inside-outside transition into C_2 that is an external contour. Then, it can be proved that *C* is an external contour.
4. *e* represents an inside-outside transition into C_2 that is an internal contour. Then, it can be proved that the parent contour of C_2 is the parent contour of *C* as well.

Now, let us prove the different cases. We take into account that the edges of the contours are interior disjoint and that the contours are simple polygons so that they are topologically equivalent to circles.

1. Let us suppose that *C* is an internal contour, see Fig. 5a. Then, the ray must cross an edge of its parent contour, which contradicts our premise. So, *C* must be an external contour.
2. Let us suppose first that C_2 does not contain *C*, see Fig. 5b and c. However, these figures contradict our premises, in Fig. 5b *e* represents an inside-outside transition into *C* and in Fig. 5c *e* is not the first edge crossed by the ray. Now, let us suppose that C_2 contains *C*, but it is not its parent, see Fig. 5d. This, again, contradicts our premises, because *e* is not the first edge crossed by the ray. So we conclude that C_2 is the parent contour of *C*.
3. Let us suppose that *C* is an internal contour, see Fig. 5e and f. These figures contradict our premises, in Fig. 5e *e* is not the first edge crossed by the ray and in Fig. 5f C_2 is an internal contour. So we conclude that *C* must be an external contour.
4. Let us suppose first that *C* is an external contour, see Fig. 5g. This figure contradicts our premises because *e* is not the first edge crossed by the ray. Now, let us suppose that *C* and C_2 have a different parent, see Fig. 5h. Again, this figure contradicts our premises because *e* is not the first edge crossed by the ray. So, we conclude that *C* and C_2 must have the same parent.

3. The first stage of the algorithm: subdividing the edges

In this section the first stage of the algorithm is described: the subdivision of the polygons edges at their intersection points and the selection of the edges that belong to the result polygon.

```
struct SweepEvent {
  Point point; // point associated with the event
  bool left;   // is the left endpoint of the edge?
  SweepEvent* otherEvent; // other event of the edge
  PolygonType pol;        // it can be SUBJ or CLIP
   ...
};
```

Listing 1. Part of the information associated to a sweep event.

```
1  Compute the events, insert them into Q
2  while (! Q.empty ()) {
3    event = Q.top ();
4    Q.pop ();
5    if (event.left) {
6      pos = S.insert (event);
7      setInformation (event, S.prev (pos));
8      possibleInter (pos, S.next (pos));
9      possibleInter (pos, S.prev (pos));
10   } else { // the event is a right endpoint
11     pos  = S.find (event->otherEvent);
12     next = S.next (pos);
13     prev = S.prev (pos);
14     S.erase (pos);
15     possibleInter (prev, next);
16   }
17   if (event->inResult)
18     result.push_back (event);
19 }
```

Listing 2. First stage of the algorithm: subdividing the edges.

Furthermore, at this stage key information for the second stage of the algorithm is computed. This stage is described in [15], but we have decided to include it again to make the paper self-contained.

In order to find the intersection points between the edges—and subdivide them—we have adapted the well-known algorithm by Bentley and Ottmann [12] for computing the intersection points in a set of line segments. This algorithm is based on the following principle: let us suppose that the plane is swept by a vertical line and a data structure, *S*, stores the line segments intersecting the sweep line sorted from bottom to top as they intersect the sweep line. Then, the following can be proved: (1) the status of *S* only changes when an endpoint or intersection point of the line segments is found and (2) if two line segments intersect, they will be adjacent along *S* at some moment of the plane sweep.

Our algorithm sweeps the plane from left to right with a vertical line. The sweep events set consists of the endpoints of the polygons edges. A priority queue *Q* holds the sweep events set sorted lexicographically. When an intersection between two edges is found the intersecting edges are subdivided and four new events are added to *Q*. This way our algorithm lacks of intersection point event.

Listing 1 shows part of the information associated to an event of the plane sweep. The point field stores its coordinates. The left field indicates if the event is the first endpoint of the edge found by the sweep line. The otherEvent field stores a pointer to the other event of the edge, so that the event can represent its associated edge. When an edge is subdivided this pointer is updated to point to the new event of the edge—associated to the intersection point. Finally, the pol field is used to hold the polygon associated to the event.

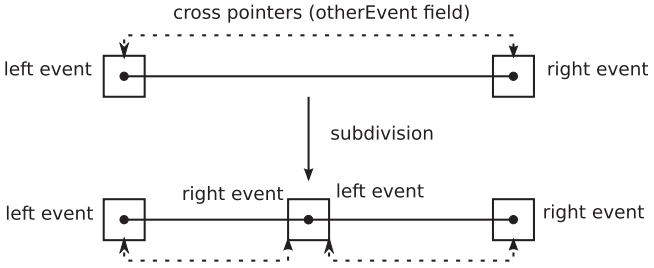

Fig. 6. Update of cross-pointers after edge subdivision.

To represent the sweep line status a balanced binary search tree is used—in the current C++ implementation the *set* container from the STL standard library.

Next, the algorithm is described—see Listing 2. Firstly, the endpoints of the edges are inserted into the priority queue in lexicographical order. Next events are processed—from left to right—as follows. When the left endpoint of an edge is found, the event, and implicitly the edge, is inserted into the sweep line status (*S*). Then, the setInformation function determines if the edge belongs to the result polygon taking into account its preceding event in *S*, this function is described in Section 4. Possible intersections between the edge and its neighboring edges in *S* are also computed. When the right endpoint of an edge is found, the edge is removed from *S*. The neighboring edges of the removed edge are now adjacent along *S*, so they are tested for intersection. Finally, if the edge

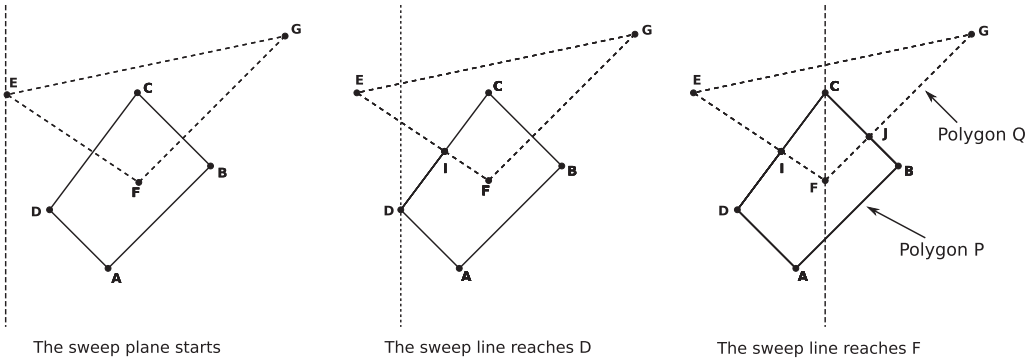

Fig. 7. Example of edge subdivision.

belongs to the result polygon its event is stored in the result array for the second stage of the algorithm.

The *possibleInter* routine is used to test and process a possible intersection between two edges. If the edges intersect at an interior point of, at least, one of the edges, then the edges are subdivided. When an edge is subdivided, two new events are inserted into Q and the otherEvent fields of the events representing the subdivided edge are updated—see Fig. 6.

Let us see an example of edge subdivision—Fig. 7. The plane sweep starts at vertex E and the edges $\overline{EF}$ and $\overline{EG}$ are inserted into S ($S = \{\overline{EF}, \overline{EG}\}$). Next, the sweep line reaches vertex D and the edges $\overline{DA}$ and $\overline{DC}$ are inserted into S ($S = \{\overline{DA}, \overline{DC}, \overline{EF}, \overline{EG}\}$). $\overline{DC}$ intersects with its neighboring edge $\overline{EF}$, so the edges are subdivided at I and S implicitly changes to $\{\overline{DA}, \overline{DI}, \overline{EI}, \overline{EG}\}$.

In this section we have not considered the degenerate cases: vertical and overlapping edges, they will be explained in Section 7.

4. The first stage of the algorithm: selecting the edges

In this section we explain how the edges belonging to the result polygon are selected. As described in Section 2.2, an edge belongs to the result polygon if it lies inside—or outside, depending on the kind of Boolean operation—the other polygon.

In order to determine whether an edge e lies inside the other polygon the following criterion is used: when e is inserted into the sweep line status (S) we select the closest edge downwards in S that belongs to the other polygon. Then, we take into account whether the edge represents an inside-outside or outside-inside transition into its polygon for a vertical ray starting below the polygon and going upwards. If the transition is outside-inside e lies inside the other polygon, otherwise e lies outside. For example, in Fig. 8 when the edge $\overline{IJ}$ is inserted into S the closest edge belonging to the other polygon (P) downwards in S is $\overline{AB}$. $\overline{AB}$ represents an outside-inside transition into P, so $\overline{IJ}$ lies inside P.

To implement this approach in an efficient way every edge/event store some flags. Given an edge e that is inserted into S, e determines whether it lies inside the other polygon checking a flag of its preceding edge/event in S—namely, the otherInOut flag. Next, we describe the meaning of the fields associated to an edge/event e related to the selection of the edges of the result polygon:

- inOut: it indicates if e determines an inside-outside transition into the polygon, to which e belongs, for a vertical ray that starts below the polygon and intersects e.
- otherInOut: it has the same meaning as the previous flag, but referred to the closest edge to e downwards in S that belongs to the other polygon.
- inResult: it indicates whether e belongs to the result polygon.
- prevInResult: pointer to the closest edge to e downwards in S that belongs to the result polygon. This field is used in the second stage of the algorithm to compute child contours.

These fields can be easily computed taking into account the values of these fields in the preceding edge in S. As an example, in Fig. 7 the edge $\overline{FJ}$ represents an outside-inside transition into its polygon (Q), so its inOut field is set to false. The closest edge to $\overline{FJ}$ downward in S and belonging to P is $\overline{AB}$. $\overline{AB}$ represents an outside-inside transition into its polygon (P), so the otherInOut field of edge $\overline{FJ}$ is also set to false and, therefore, $\overline{FJ}$ lies inside P.

To correctly compute the fields endpoints with the same coordinate must be processed—that is, sorted into the priority queue—as follows:

- Right endpoints must be processed before the left ones.
- Left endpoints must be processed in the ascending order of their associated edges in S.

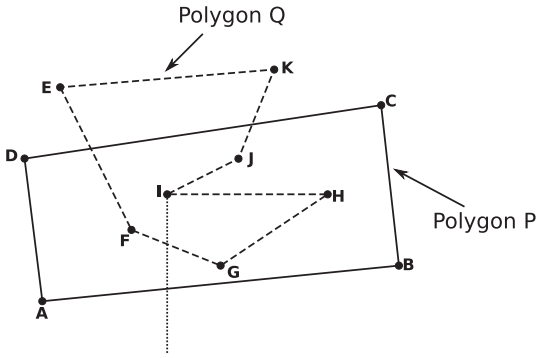

Fig. 8. Determining whether edge $\overline{IJ}$ lies inside P.

5. The second stage of the algorithm: joining the edges

In this section the second stage of the algorithm is described: the edges of the result polygon are joined to form contours and the child contours are assigned to their parent contours.

The contours are computed from the information stored in the result array, that was computed in the first stage of the algorithm. This array stores the events associated to the result polygon edges in lexicographical order. For example, Fig. 9 shows a result polygon and its associated result array.

Listing 3 shows the algorithm for computing a contour given its first event/vertex e. The first two vertices of the contour are trivially computed as e->point and e->otherEvent->point. In the sample polygon of Fig. 9, and supposing that we start with the first event of the result array, the first two vertices added to the contour are the endpoints of the edge $\overline{AB}$. To find the next vertex of the contour we have to find an edge with an endpoint equal to the last vertex computed: the other endpoint of the edge is the vertex that we are looking for. For example, in Fig. 9, and after adding vertices A and B to the contour, the edge starting at vertex B is $\overline{BC}$, so vertex C should be added to the contour. This process must be repeated until all the vertices of the contour are computed, that is, until the starting vertex is reached. In the algorithm in Fig. 3 the function neighbor is used to find an event placed at the same coordinates that the event that receives as a parameter. Owing to result is lexicographically sorted the event to be found is placed in result at an adjoining position to the event received as a parameter—so, it can be efficiently found.

The second stage of the algorithm uses some new fields of the SweepEvent structure—see Listing 4. Next, we explain the meaning

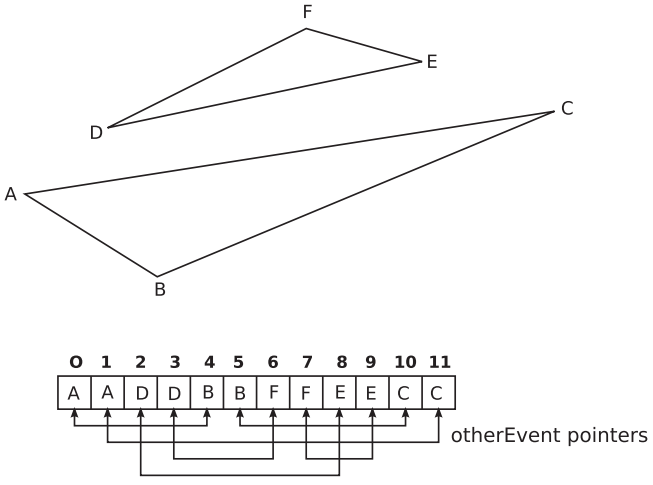

Fig. 9. Result polygon consisting of two contours and its result array.

```
void computeContour (SweepEvent* e, Contour& C) {
  C.add (e->point);
  SweepEvent* current = e->otherEvent;
  while (current->point != e->point) {
    C.add (current->point);
    current = neighbor (current, result);
    current = current->otherEvent;
  }
}
```

Listing 3. Algorithm for computing a contour.

```
struct SweepEvent {
  ...
  int pos;
  bool resultInOut;
  int contourId;
  int parentId;
  bool processed;
  int depth;
};
```

Listing 4. Information associated to a sweep event used in the second stage of the algorithm.

```
for (int i = 0; i < result.size (); ++i) {
  result[i]->pos = i;
  result[i]->processed = false;
}
Polygon P;                    // result polygon
int currentContourId = 0;
for (int i = 0; i < result.size (); ++i) {
  if (! result[i]->processed) {
    Contour C;
    int depth, parent;
    computeDepth(result[i]->prevInResult,depth,parent);
    computeContour(result[i], C);
    if (depth is odd)
      C.changeOrientation ();
    P.add (C);
    if (parent != -1)
      P(parent).addChild (currentContourId);
    currentContourId++;
  }
}
```

Listing 5. Algorithm for computing the contours.

of these fields for an event e stored in result that belongs to contour C:

- pos: it stores the position of e in result.
- resultInOut: it indicates if the edge associated to e determines an inside-outside transition into C for a vertical ray that starts below C and intersects the edge associated to e.
- contourId: an integer value that identifies the contour C.
- parentId: if C is an internal contour, then it stores the identifier of its parent contour, otherwise it stores the same value as the contourId field—i.e., the C identifier.
- processed: it indicates whether the point associated to e has been added or not to a contour.
- depth: the amount of contours that contain C.

Next we describe the algorithm—see Listing 5. The first loop is used to initialize the pos and processed fields. In the second loop the contours of the result polygon are computed. When an event that has not still been processed is found, the contour starting at its associated point is computed. The computeDepth function computes the depth and the parent contour of the new contour; this function is explained later. In order to compute the new contour the computeContour function is used, see Listing 3. As explained before, this function takes advantage of the fact that result is lexicographically sorted—see Fig. 9. In Fig. 9 the first event of result is the left endpoint of edge $\overline{AB}$, so the first contour computed starts at vertex A. The right event of $\overline{AB}$ is at position 4 in result—this position is stored in result[0]->otherEvent->pos. Owing to result is lexicographically sorted, the next edge of the contour must have an event at an adjoining position to 4—in this case position 5. This way all the linked edges of the first contour can be found. As the events of a contour are found their resultInOut, contourId, parentId, processed and depth fields are updated—this is not included in the algorithm in Listing 3. The field resultInOut is set to true if the right event precedes the left event in the contour, otherwise it is set to false.

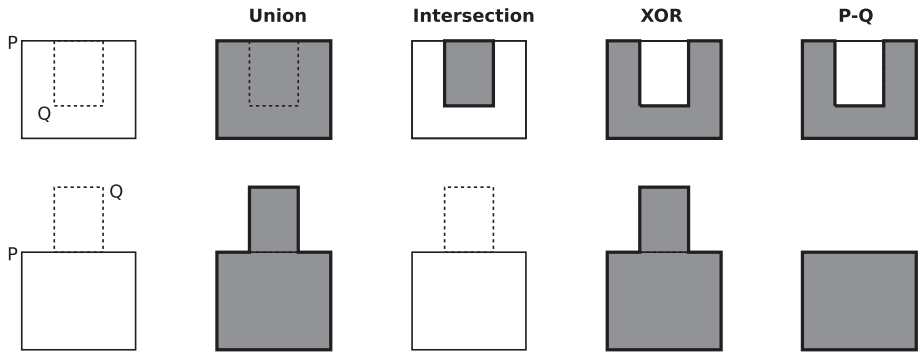

Fig. 10. Polygons with overlapping edges.

Now let us explain the computeDepth function. This function computes the depth and the parent contour, if it exists, of a new contour. Owing to result is lexicographically sorted, when the algorithm finds the first event e of a new contour C the following holds: (1) e is the bottom left vertex of C and (2) if there is a result contour C_2 intersected by a vertical ray starting from e and going downwards, then C_2 has previously been computed by the algorithm. Therefore, we can compute the parent contour of C taking into account the four cases shown in Fig. 4.

The algorithm computes the contours in counter-clockwise order. If a contour is included in an odd number of contours, then its orientation is changed before inserting it into the result polygon.

6. Running time analysis

In this section we analyze the running time of the algorithm. In the analysis n denotes the total number of edges of the input polygons and k denotes the number of intersection points between the edges of the input polygons.

Firstly, we analyze the first stage of the algorithm, shown in Listing 2. In the first sentence, the sweep events are inserted into Q—$O(n \log n)$. Then, the while loop processes the events. Let us analyze its body:

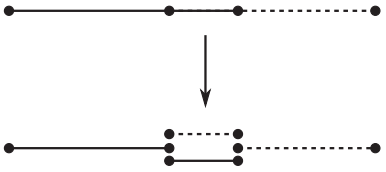

Fig. 11. Overlapping edges subdivision.

- All the operations on the balanced binary search tree S—insert, remove, find, next and previous—take $O(\log n)$ because S stores at most n edges.
- The possibleInter function can insert four events into Q; as Q has $O(\log (n+k))$ size the function takes $O(\log (n+k))$ time.
- The other instructions of the loop, including the setInformation function, take constant time.

The loop is executed $2(n+2k)$ times, so it takes $O((n+k) \log (n+k))$, i.e., $O((n+k) \log n)$, since $k \leqslant n^2$. This time dominates the time needed for initializing Q, so the first stage of the algorithm runs in time $O((n+k) \log n)$.

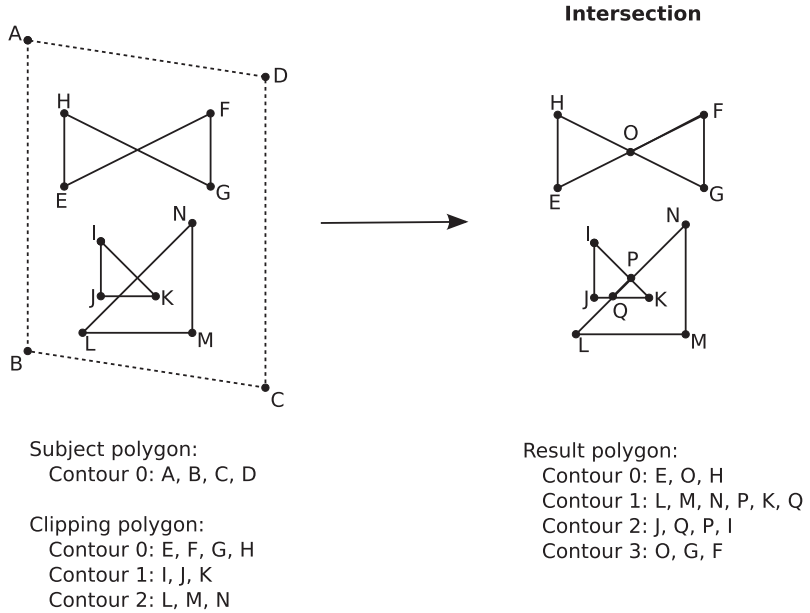

Fig. 12. Intersection of a self-intersecting polygon.

To analyze the second stage of the algorithm—Listing 5— we take into account that the result array stores at most $2(n + 2k)$ events, so the first loop takes $O(n + k)$ time. Let us analyze the second loop. The body of the if sentence is only executed when a new contour is found. The computeDepth function takes constant time. The computation of a contour, the change of its orientation and its addition to the result polygon takes $O(c)$ time, where c is the number of vertices of the contour. Owing to $n + k$ is an upper bound on the number of vertices of the result polygon we can conclude that the second loop, and so the second stage of the algorithm, takes $O(n + k)$ time.

Therefore, we conclude that the algorithm runs in time $O((n + k) \log n)$.

7. Degenerate and special cases

In this section we sketch how the degenerate cases of the algorithm are handled.

7.1. Vertical edges

The algorithm sorts the events lexicographically, so the lower endpoint of a vertical edge is considered its "left event" and the upper endpoint its "right event". Other things about vertical edges to be taken into account are the following:

- To order the sweep-line status (S) we consider that a vertical edge intersects the sweep line at the y–coordinate of its lower endpoint.
- If a non-vertical edge intersects the sweep line at the lower endpoint of a vertical edge, then the vertical edge is placed in S after the non-vertical edge.

7.2. Overlapping edges

Overlapping edges are special because they are not in the result polygon depending on whether they are inside the other polygon. For example, in Fig. 10 we can see that the overlapping edge belongs to the union of the upper polygons, but it does not belong to the union of the lower polygons.

When two edges overlap they are subdivided so that their overlapping fragments become an edge of each polygon—see Fig. 11. The algorithm must select at most one of these two "equal edges" as part of the result polygon. This selection is done taking into account whether the overlapping edges have the same value in the *inOut* flag and the kind of Boolean operation. Edges that have the same *inOut* flag are only included in the result of union and intersection operations, whereas edges that have different *inOut* flags are only included in the result of set theoretic difference operations—see Fig. 10.

7.3. Self-intersecting polygons

Our algorithm can work with self-intersecting polygons, but it subdivides their edges at their intersection points. For example, Fig. 12 shows the result of a Boolean operation with an self-intersecting polygon.

7.4. Several edges meeting at an endpoint

In Section 6, when analyzing the running time for computing a new contour, we said that the edges of a contour can be linked in $O(c)$ time, where c is the number of vertices of the contour. This is true if, given an edge, the next edge of the contour can be found in constant time in the result array. This is the case when exactly two edges of a contour meet at every vertex of the contour—see Fig. 9, but it is not the case when an arbitrary number of edges can meet at a vertex—see Fig. 13. The current implementation of the algorithm uses a very similar, but slightly different strategy, from that explained in Section 5, in order to join the edges of the contours in $O(c)$ time, regardless the number of edges that meet any vertex. The strategy is based on storing the coincident events in the same position of the result array.

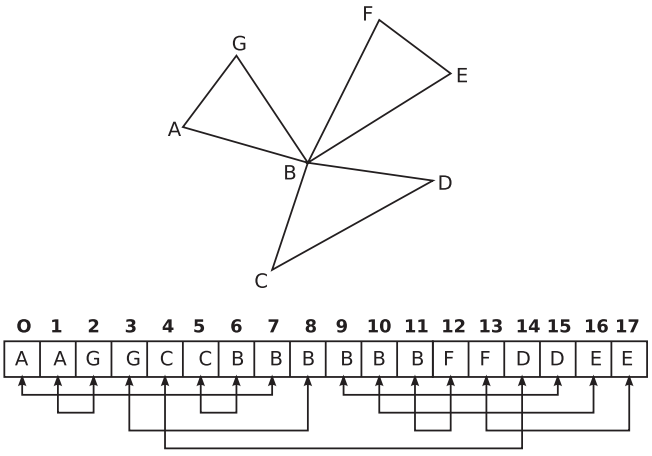

Fig. 13. Result polygon with several edges meeting at a vertex.

8. Optimizations

In this section we describe some optimizations developed in the implementation of the algorithm:

1. Before computing the Boolean operation the bounding boxes of the input polygons are computed. If the bounding boxes do not overlap, then the Boolean operation can be trivially computed.
2. When computing the intersection between polygons, when the rightmost vertex of an input polygon is reached by the sweep line, then we can stop the plane sweep because no new edges can be added to the result polygon. Likewise, we can stop the plane sweep when computing $P - Q$ and the rightmost vertex of P is reached by the sweep line.
3. When the algorithm of Listing 2 reaches the right event of an edge its left event is found in S—line 11—, an operation that takes $O(\log n)$ time. We can avoid this operation if the left event stores its position in S in the SweepEvent structure.

9. Experimentation

In this section an implementation of our algorithm is compared with the Greiner and Hormann's and Vatti's algorithms. For the Vatti's algorithm we have used the well-known implementation by Alan Murta.[1] The experiments have been conducted on an Intel Core i5 processor under Linux. Table 1 shows the time needed for intersecting two polygons representing continents at different resolutions. The last column of Table 1 presents the number of intersections between the edges of the polygons. Clearly, our algorithm performs better than the other algorithms when the number of vertices is increased. We think the main reason behind these results is the way wherein the different algorithms compute the intersections between the polygon edges—it must be taken into account that the

[1] General Polygon Clipper library, by Alan Murta, http://www.cs.man.ac.uk/toby/alan/software/.

Table 1
Execution times of intersection operations (in seconds).

Number of vertices	Greiner	Vatti	Our algorithm	Number of intersections
25,786 × 18,065	0.02	0.11	0.04	10,405
124,636 × 55,648	6.1	4.2	0.66	33,786
223,834 × 175,629	120.4	35.3	1.3	91,034

algorithms spend the majority of their CPU time computing these intersections points. Let us see how these points are computed by the different algorithms:

- Greiner and Hormann's algorithm uses a naive approach: each edge is tested against the other edges for intersection. Of course, this approach is inefficient, especially when the number of intersections is sublinear in the number of vertices, as it is the case with the tested polygons.
- Our algorithm uses the classical plane sweep approach. Edges are only tested for intersection when they become adjacent in the status line. At most a pair of intersection tests are computed during the processing of a plane sweep event. If the number of intersections is sublinear in the number of vertices—a very common situation—then the intersections are computed very efficiently. Our algorithm's running time is $O((n+k)\log n)$, however, as in the tested polygons $k \leqslant n$ the algorithm exhibits a $O(n\log n)$ running time for these test cases.
- Although Vatti's algorithm is also based on the plane sweep technique, it is very different from our algorithm. For example, it does not use the classical plane sweep approach for computing the intersection points: in Vatti's algorithm during the processing of a plane sweep event each edge in the status line has to be tested for intersection with its immediate predecessor edge in the status line. Obviously, this approach is slower than our method that, as mentioned above, only needs a pair of intersection tests for each plane sweep event processed.

10. Conclusions

In this paper we have redesigned a previous algorithm for computing Boolean operations on polygons so that it can compute the child contours of the result polygon contours. This information is not computed by previous algorithms. Furthermore, the final algorithm is easier to understand and implement than the original algorithm. Experimental results show that when the number of intersections between the polygons is linear in the number of vertices of the polygons—a common situation—the algorithm is faster than Vatti's and Greiner and Hormann's algorithms.

References

[1] Andreev RD. Algorithm for clipping arbitrary polygons. Comput Graph Forum 1989;8(3):183–91.
[2] Sutherland IE, Hodgman GW. Commun ACM 1974;17(1):32–42.
[3] Liang YD, Barsky BA. An analysis and algorithm for polygon clipping. Commun ACM 1983;26(11):868–77.
[4] Greiner G, Hormann K. Efficient clipping of arbitrary polygons. ACM Trans Graph 1998;17(2):71–83.
[5] Kim D, Kim M. An extension of polygon clipping to resolve degenerate cases. Comput-Aid Des Appl 2006;3:447–56.
[6] Kui Liu Y, Qiang Wang X, Zhe Bao S, Gomboši M, Žalik B. An algorithm for polygon clipping, and for determining polygon intersections and unions. Comput Geosci 2007;33(5):589–98.
[7] Rivero M, Feito FR. Boolean operations on general planar polygons. Comput Graph 2000;24(6):881–96.
[8] Peng Y, Yong J-H, Dong W-M, Zhang H, Sun J-G. A new algorithm for boolean operations on general polygons. Comput Graph 2005;29(1):57–70.
[9] Feito FR, Rivero M. Geometric modelling based on simplicial chains. Comput Graph 1998;22(5):611–9.
[10] Nievergelt J, Preparata FP. Plane-sweep algorithms for intersecting geometric figures. Commun ACM 1982;25(10):739–47.
[11] de Berg M, Cheong O, van Kreveld M, Overmars M. Computational geometry: algorithms and applications. 3rd ed. Berlin: Springer; 2008.
[12] Bentley J, Ottmann T. Algorithms for reporting and counting geometric intersections. IEEE Trans Comput 1979;28(9):643–7.
[13] Vatti BR. A generic solution to polygon clipping. Commun ACM 1992;35(7):56–63.
[14] Wang J, Cui C, Gao J. An efficient algorithm for clipping operation based on trapezoidal meshes and sweep-line technique. Adv Eng Softw 2012;47(1):72–9.
[15] Martínez F, Rueda AJ, Feito FR. A new algorithm for computing boolean operations on polygons. Comput Geosci 2009;35(6):1177–85.