Computers & Geosciences 35 (2009) 1177–1185

Contents lists available at ScienceDirect

Computers & Geosciences

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

A new algorithm for computing Boolean operations on polygons ☆

Francisco Martínez *, Antonio Jesús Rueda, Francisco Ramón Feito

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

ARTICLE INFO

Article history:
Received 21 January 2008
Received in revised form
12 June 2008
Accepted 25 August 2008

Keywords:
Polygon clipping
Boolean operations polygons
Polygon overlay

ABSTRACT

This paper presents a new algorithm for computing Boolean operations on polygons. These kind of operations are frequently used in the geosciences in order to get spatial information from spatial data modeled as polygons. The presented algorithm is simple and easy to understand and implement. Let n be the total number of edges of all the polygons involved in a Boolean operation and k be the number of intersections of all the polygon edges. Our algorithm computes the Boolean operation in time $O((n+k)\log(n))$.

Finally, the proposed algorithm works with concave polygons with holes, and with regions composed of polygon sets. Furthermore, it can be easily adapted to work with self-intersecting polygons.

1. Introduction

Polygon overlay, which is also referred to as polygon clipping or polygon intersection, operations determine the spatial coincidence (if any) of two polygon data layers, usually creating a new polygon layer in the process. Polygon overlay techniques are often used by field scientists to explore the relationships between spatial attributes, stored as layers in a geophysical data model. Examples of polygon data are: tectonic plates, biomes, watersheds or sea ice. For example, Fig. 1 shows two polygons representing the sea ice coverage over two time periods. A Boolean operation on the polygons can be used to visualize and get information about the changes in sea ice coverage over time. For example, $P-Q$ represents the coverage area gained during the period, and $Q-P$ the coverage area lost—it is supposed that Q represents a time previous to P.

There is a slight difference between polygon clipping and polygon intersection. The former refers to when several polygons are clipped against the same clipping polygon. Many efficient algorithms exist for polygon clipping (Foley et al., 1990). However, most of them are limited to certain types of polygons, for instance, Andereev (1989) and Sutherland and Hodgeman (1974) algorithms require a convex clip polygon, while the algorithm by Liang and Barsky (1983) requires a rectangular clip polygon.

For the general case of polygons, i.e. concave polygons with holes and self-intersections, less solutions are available. Furthermore, some of the solutions need complex, specific data structures as it is the case of Weiler (1980) algorithm.

Greiner and Hormann (1998) propose a new algorithm for clipping polygons. The algorithm is very easy to understand and implement. In addition, it is very fast, especially for self-intersecting polygons. Nevertheless, the algorithm treats degeneracy, which occurs when a vertex of a polygon lies on an edge of the other, by perturbing the position of the vertex. Fig. 2 shows that perturbation is not always a good solution. As can be seen the result of the Boolean operation $P-Q$ depends on the kind of perturbation, on the left figure a polygon with a hole is obtained, on the right figure a polygon with two regions is obtained. Liu et al. (2007) propose some optimizations to Greiner and Hormann's algorithm, and explain how the algorithm

☆ Code available from server at http://www.iamg.org/CGEditor/index.htm.

* Corresponding author. Tel.: +34 953212887; fax: +34 953212420.
E-mail addresses: fmartin@ujaen.es (F. Martínez), ajrueda@ujaen.es (A.J. Rueda), ffeito@ujaen.es (F.R. Feito).

0098-3004/$ - see front matter © 2008 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cageo.2008.08.009

can be adapted to work with polygons with holes and regions composed of polygon sets. Unfortunately, they do not bring new solutions to the degeneracy problem.

In this paper we propose a new algorithm for computing Boolean operations on polygons. The algorithm is very easy to understand, among other things because it can be seen as an extension of the classical algorithm, based on the plane sweep, for computing the intersection points between a set of segments, see Preparata and Shamos (1985). When a new intersection between the edges of polygons is found, our algorithm subdivides the edges at the intersection point. This produces a plane sweep algorithm with only two kind of events: left and right endpoints, making the algorithm quite simple. Furthermore, the subdivision of edges provides a simple way of processing degeneracies.

It must be noted that Vatti (1992) algorithm is also based on the plane sweep paradigm. However, our algorithm is quite different than Vatti's one. Concretely, our algorithm takes a different, more efficient, approach for computing the intersections between the edges of the polygons involved in the Boolean operation, making our algorithm much faster than Vatti's one for the computation of Boolean operations for large polygons.

The remainder of the paper is structured as follows. In the next section the theoretical background on which the algorithm is based is stated. In Section 3 the overall algorithm is described. Sections 4 and 5 explain how the edges belonging to the result of the Boolean operation are selected and connected to form the solution. Section 6 makes a complexity analysis of the algorithm. Section 7 describes how the special cases of the algorithm are processed. In Section 8 a comparison with Vatti's and Greiner and Hormann's algorithms is made. Finally, Section 9 brings some conclusions.

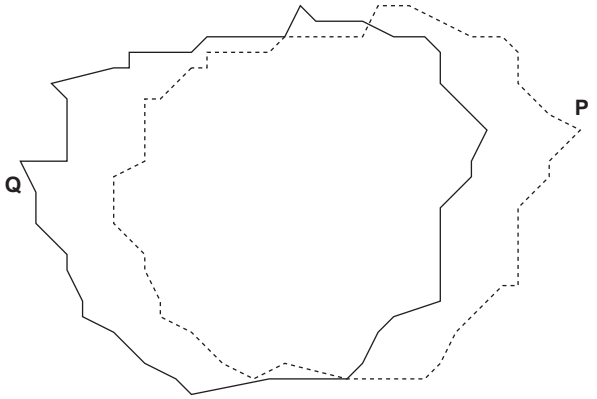

Fig. 1. Sea ice coverage over two time periods.

2. Basics

A natural way to represent a polygon is by listing its vertices in counter-clockwise order: $v_0, v_1, v_2, \ldots, v_n$. The ordered list of edges $\overline{v_0v_1}, \overline{v_1v_2}, \ldots, \overline{v_nv_0}$ defines the polygon boundary.

The boundary of the result of a Boolean operation on two polygons consists of those portions of the boundary of each polygon that lie or do not lie, depending on the type of operation, inside the other polygon. For example, Fig. 3 shows the results of different Boolean operations on two polygons. Therefore, the computation of a Boolean operation is reduced to finding these portions. Once found, they must be connected to form the result polygon.

Suppose that the edges of two polygons are subdivided at their intersection points, see Fig. 4. In this case the boundaries of the polygons intersect at endpoints of some of their edges. Therefore, the problem of computing the boundary of the result of a Boolean operation on the polygons is reduced to finding those edges of each polygon

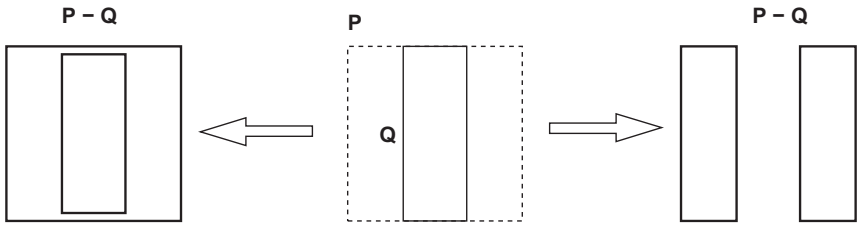

Fig. 2. Depending on kind of perturbation different solutions are obtained.

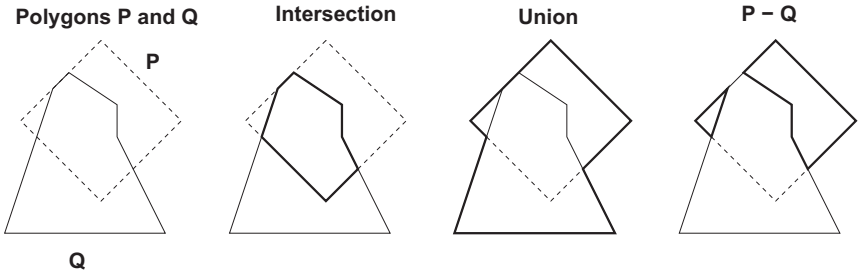

Fig. 3. Boolean operations on polygons.

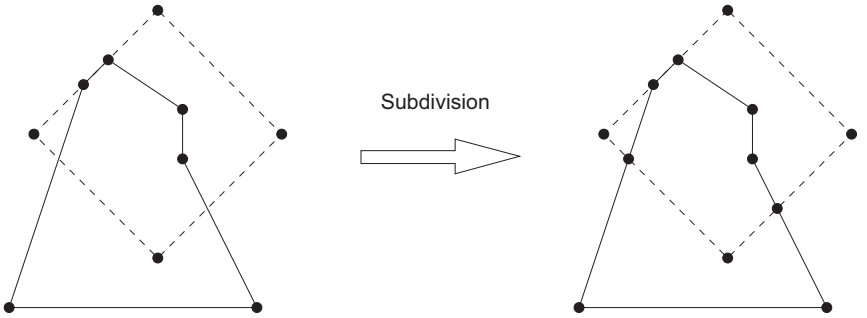

Fig. 4. Subdivision of edges of polygons at their intersection points.

that lie or do not lie, depending on the type of operation, inside the other polygon. Again, once these edges are found they must be connected to form the result polygon.

We can therefore sketch the following approach for computing Boolean operations on polygons:

(1) Subdivide the edges of the polygons at their intersection points.
(2) Select those subdivided edges that lie inside the other polygon—or that do not lie depending on the operation.
(3) Join the edges selected in step 2 to form the result polygon.

The algorithm proposed in this paper uses the plane sweep technique to efficiently implement this approach.

3. The algorithm

In this section we describe the algorithm for computing Boolean operations on polygons. In order to subdivide the edges of the polygons we must first find their intersection points. This task can be efficiently done using the following principle: suppose that the plane is swept with a vertical line. At every moment the edges that intersect the sweep-line are stored, ordered from bottom to top as they intersect the sweep-line, in a data structure S. Then, it can be proved that: (1) the status of S only changes when the sweep-line reaches an endpoint or an intersection point of the edges and (2) only edges that are adjacent along S can intersect. The classical algorithm for computing the intersection points between a set of segments is based on this principle given by Preparata and Shamos (1985).

Our algorithm also uses this approach to efficiently find the intersection points between the edges of the polygons. Furthermore, the information available during the plane sweep is used to subdivide the edges and decide which of them should be included in the result of the Boolean operation. The algorithm is described next.

We use a vertical line to sweep the plane from left to right. The *sweep-line status*, S, consists of the ordered sequence of the edges of both polygons intersecting the vertical line. S will only change at the endpoints of the edges:

- When the left endpoint of an edge is reached the edge must be added to S.
- When the right endpoint is reached the edge must be removed from S.

Therefore, the *event-point set* is formed by the endpoints of the edges of the polygons. This set changes dynamically because when an edge is subdivided two new endpoints appear. The algorithm implements the event-point set using a priority queue that holds the endpoints sorted from left to right.

Now, we can describe the algorithm, see Fig. 5. Firstly, the endpoints of the edges are placed into a priority queue sorted by x coordinate. Then the endpoints are processed—from left to right—as follows. When a left endpoint is found its associated edge is inserted into the sweep line status (S). Then, following the approach explained in Section 4, it is computed if the edge lies inside the other polygon. Possible intersections with its neighbors along S must also be processed. When a right endpoint is found its associated edge is removed from S. Now, its two neighbors along S become adjacent, and are tested for intersection. The removed edge is also considered for inclusion in the result of the Boolean operation.

The procedure *possibleInter* is used to detect and process a possible intersection between two edges. If the edges belong to the same polygon or they only intersect at one of their endpoints no extra processing is required. If the edges belong to different polygons and they intersect at a point interior to one of the edges then they must be subdivided. When an edge is subdivided the data structures Q and S are updated to reflect the new status. Fig. 6 shows the types of intersections that lead to the subdivision of an edge and how they are processed. We have used the intersection routine described in Schneider and Eberly (2003) for detecting a possible intersection between two edges.

Let us see an example of edge subdivision, see Fig. 7. When the sweep-line reaches the point p_1, we have S is $\{\overline{q_2q_3}, \overline{q_1q_2}\}$. Then, the left endpoint of $\overline{p_1p_2}$ is processed, and $\overline{p_1p_2}$ is inserted into S ($S = \{\overline{q_2q_3}, \overline{q_1q_2}, \overline{p_1p_2}\}$). $\overline{p_1p_2}$ intersects with its neighbor $\overline{q_1q_2}$ at point i, so $\overline{p_1p_2}$ and $\overline{q_1q_2}$ must be subdivided into edges $\overline{p_1i}$, $\overline{ip_2}$,$\overline{q_1i}$ and $\overline{iq_2}$. Q must be updated to include the endpoints of these new

```
01. Insert the endpoints of the edges of polygons into priority queue Q
02. while (! Q.empty ()) {
03.   event = Q.top ();
04.   Q.pop ();
05.   if (event.left_endpoint ()) {
06.     pos = S.insert (event);
07.     event.setInsideOtherPolygonFlag (S.prev (pos));
08.     possibleInter (pos, S.next (pos));
09.     possibleInter (pos, S.prev (pos));
10.   } else { // the event is a right endpoint
11.     pos  = S.find (*event.other);
12.     next = S.next (pos);
13.     prev = S.prev (pos);
14.     if (event.insideOtherPolygon ()) Intersection.add (event.segment ());
15.     if (! event.insideOtherPolygon ()) Union.add (event.segment ());
16.     S.erase (pos);
17.     possibleInter (prev, next);
18.   }
19. }
```

Fig. 5. Algorithm.

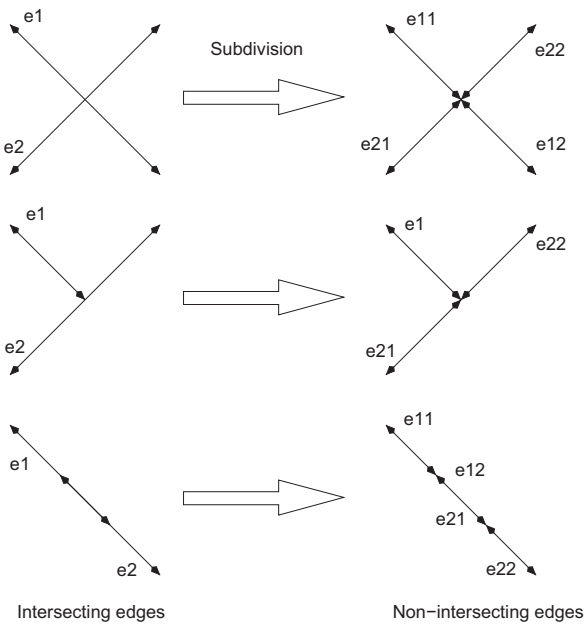

Fig. 6. Types of intersections that lead to a subdivision.

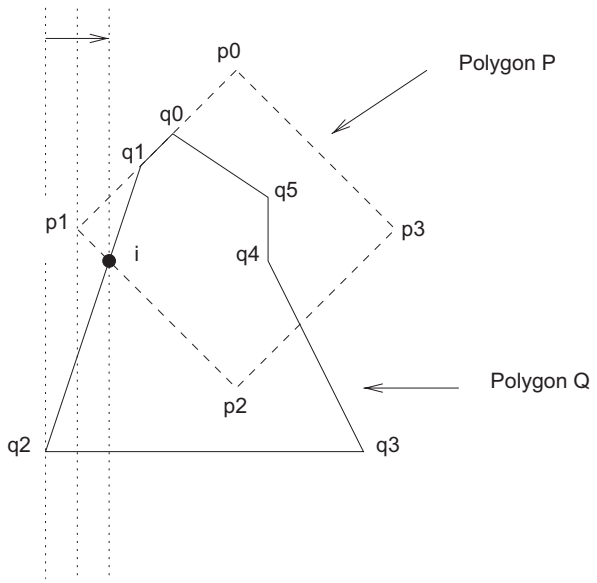

Fig. 7. Sweep-line.

edges. S will also change to $S = \{\overline{q_2q_3}, \overline{iq_2}, \overline{p_1i}\}$. After this, the left endpoint of $\overline{p_0p_1}$ is processed, and $\overline{p_0p_1}$ is inserted into S ($S = \{\overline{q_2q_3}, \overline{iq_2}, \overline{p_1i}, \overline{p_0p_1}\}$).

4. Selecting the result edges

When the sweep-line reaches the right endpoint of an edge e the algorithm decides if e belongs to the result of the Boolean operation. As outlined in Section 2, this

```
struct SweepEvent {
  Point p;          // point associated with the event
  SweepEvent *other; // event associated to the other endpoint of the edge
  bool left;  // is the point the left endpoint of the edge (p, other->p)?
  PolygonType pl;    // it can be SUBJECT or CLIPPING
  bool inOut;        // inside-outside transition into the polygon
  bool inside;       // is the edge (p, other->p) inside the other polygon?
  EdgeType type;     // used for overlapping edges
};
```

Fig. 8. Data structure for representing events/edges.

```
void setInsideFlag (SweepEvent* le, SweepEvent* ple) {
   if (ple == NULL) {
      le->inside = le->inOut = false;
   } else if (le->pl == ple->pl) { // same polygon ?
      le->inside = ple->inside;
      le->inOut  = ! ple->inOut;
   } else {
      le->inside = ! ple->inOut;
      le->inOut  = ple->inside;
   }
}
```

Fig. 9. Routine to set inside and inOut flags of edges.

decision is made by testing if e lies inside the other polygon P.

In the algorithm we compute if e lies inside P when e is inserted into S. We can easily make this computation after reading three flags of information from the edge that precedes e in S. In Fig. 8 we suggest a data structure for representing an endpoint of an edge—an event-point—that implicitly represents its associated edge—we insert into S the left endpoint of the segments. The three flags from the preceding edge that have to be read are:

- *pl*: indicates if the edge belongs to the subject or clipping polygon.
- *inOut*: indicates if the edge determines an inside–outside transition into the polygon, to which the edge belongs, for a vertical semi-line that goes up and intersects the edge.
- *inside*: indicates if the edge is inside the other polygon.

Fig. 9 shows a routine that computes the *inOut* and *inside* flags of a left endpoint event *le*, that has been inserted into S, given the left endpoint event *ple* of the immediate predecessor of *le* in S. If *ple* is null then *le* is the first event in S and the flags can be trivially set to false.

To correctly apply this routine endpoints placed at the same x coordinate must be processed—that is, sorted into the priority queue—from bottom to top. If two endpoints share the same point the right endpoints must be processed before the left ones. If two left endpoints share the same point then they must be processed in the ascending order of their associated edges in S.

5. Connecting the result edges to form the solution

The result of a Boolean operation on two polygons is a set, possibly empty, of polygons. In the previous sections we have described how to find the edges of these polygons. Next, we show how these edges can be connected to form the result polygons.

We must hold a set C—initially empty—of chains of connected edges and a set R that holds the result polygons. Every edge e that belongs to the solution must be processed as follows:

- If e cannot be connected at any of the ends of any chain of C, then a new chain, formed by e, is added to C.
- If e can be connected to only one chain c of C, then e is added to c. If the first and last edges in c are connected, then c holds a result polygon and it is moved to R.
- If e can be connected to two chains c_1 and c_2 of C, then the edges of c_2 and e are added to c_1, and c_2 is removed from C. If the first and last edges in c_1 are connected then c_1 is moved to R.

6. Performance analysis

In this section we analyze the performance of the algorithm shown in Fig. 5. We will use the following notation: let n be the total number of edges of all the polygons involved in the Boolean operation and k be the number of intersections of all the polygon edges.

The algorithm starts inserting all the endpoints of the edges on Q, which takes $O(n\log(n))$. Then the plane sweep starts and all the events are processed in the cycle. Let us analyze the cycle body:

- Lines 6, 11–13 and 16 are operations on S. S holds at most n edges and it can be implemented as a dictionary, so these lines take each $O(\log(n))$.
- Line 7 runs in time $O(\log(n))$, since this is the time needed to determinate the immediate predecessor of the event in S—the routine used to set the *inside* and *inOut* flags runs in time $O(1)$.

- The function *possibleInter* takes $O(\log(n+k))$, because after an intersection test, which uses constant time, four insertions on Q can be done, and Q has an $O(n+k)$ size.
- Line 3 runs in constant time, and line 4 takes $O(\log(n+k))$.
- Finally, the inclusion of an edge in the result polygons—lines 14 and 15—which is treated in Section 5, takes $O(\log(n))$. There can be at most n chains of connected edges in which to include the edge. The endpoints of the chains can be stored in a dictionary, so that finding the chain that joins with the edge runs in time $O(\log(n))$. The remainder of operations—joining and edge to a chain or joining two chains—can be implemented in constant time.

Therefore, we can conclude that the cycle body runs in time $O(\log(n+k))$. The cycle is executed $(n+4k)$ times so the cycle takes $O((n+k)\log(n+k))$, i.e. $O(n+k)\log(n)$, since $k \leqslant n^2$. This time clearly dominates the initial $O(n\log(n))$ step 1, so the whole algorithm runs in time $O(n+k)\log(n)$.

7. Special cases

In this section we discuss the special cases of the algorithm. As it will be shown they are treated in a simple, elegant way.

7.1. Vertical edges

Vertical edges are special because their two endpoints are placed at the same x coordinate. However, they can be processed by the algorithm as "normal edges" as long as the following simple rules are met:

(1) The lower endpoint of a vertical edge must be considered as its left endpoint and the upper endpoint as its right endpoint.
(2) To order the sweep-line status (S) it must be considered that a vertical edge intersects the sweep-line at the y coordinate of its lower endpoint—remember that S is ordered by the y coordinate at which edges intersect the sweep-line. 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

When two edges overlap they are subdivided so that their overlapping fragments become an edge of each polygon—see the last type of intersection in Fig. 6. The algorithm must select at most one of these two "equal edges" as part of the result of the Boolean operation. Unfortunately, the methods explained in Section 4 to select the result edges does not work for overlapping edges. Therefore, the two "equal edges" representing an overlapping fragment need a special processing, which is described next.

When overlapping between two edges is detected one of the edges representing the overlapping fragment is labeled as NON_CONTRIBUTING, meaning that the edge will not be considered for inclusion in the result of the Boolean operation. The other edge is labeled as SAME_TRANSITION or DIFFERENT_TRANSITION depending on the overlapping edges having the same *inOut* flag, and it will be included in the result depending on its label and on the type of Boolean operation. Edges labeled as

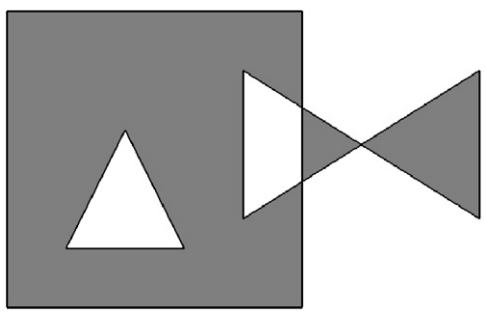

Fig. 11. Example of self-intersecting polygon.

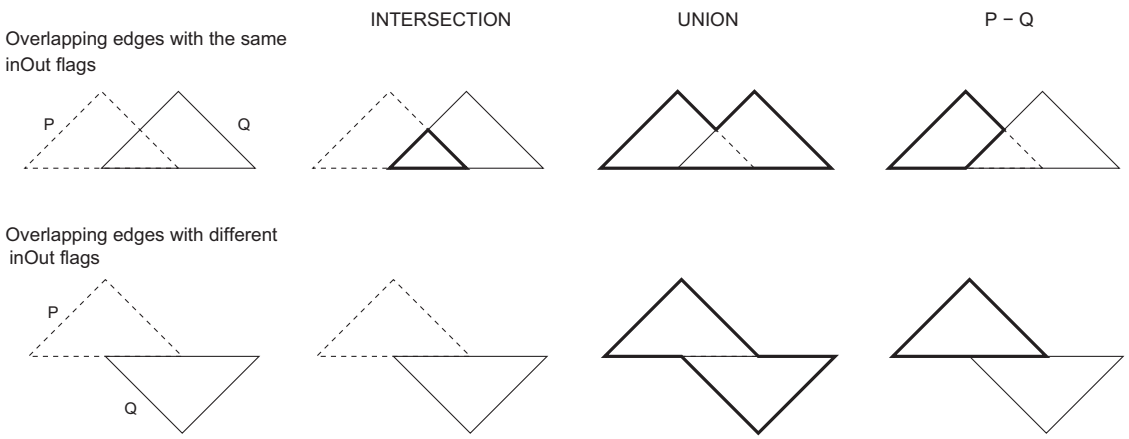

Fig. 10. Inclusion of overlapping edges in result of Boolean operations.

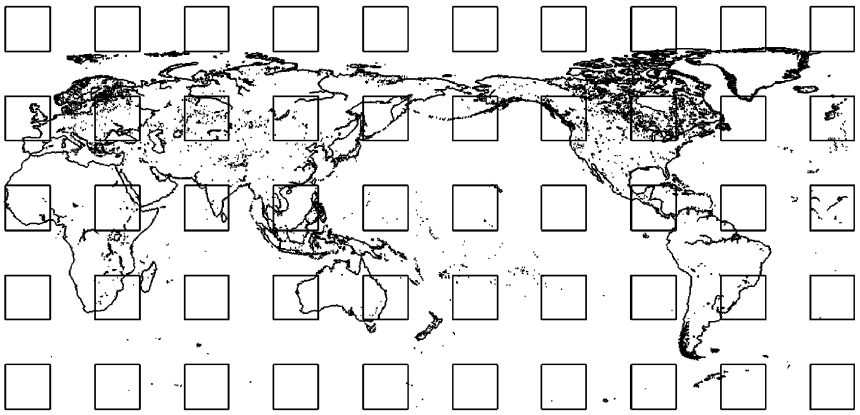

Fig. 12. Subject and clipping polygons.

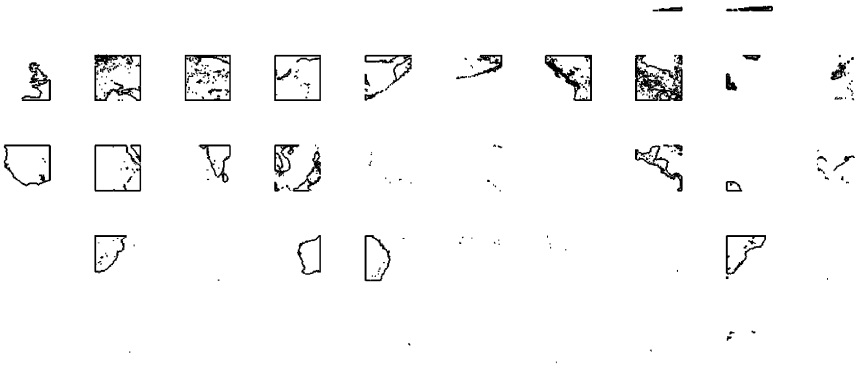

Fig. 13. Intersection.

SAME_TRANSITION are only included in the result of union and intersection operations. Edges labeled as DIFFERENT_TRANSITION are only included in the result of set theoretic difference operations. Fig. 10 shows how overlapping fragments are included in the result of Boolean operations depending on the *inOut* flag of the overlapping edges and on the type of Boolean operation.

7.3. Self-intersecting polygons

When the boundary of a polygon crosses itself, the polygon is called self-intersecting. Fig. 11 shows a polygon set consisting of three individual polygons: a square, a triangle inside the square—a hole—and a self-intersecting bow-tie shaped polygon that, in turn, intersects with the square. To know whether a point belongs to the interior of the polygon the even–odd rule can be applied: let r be a ray thrown from the point to infinity in any direction, such as the ray does not cross any polygon vertex or self-intersecting point, and let c be the number of times that r crosses the boundary of the polygon. Then, the point is inside the polygon if c is odd—and outside if c is even.

The algorithm does not work for polygons with self-intersections. The reason is simple: the algorithm is not aware of self-intersection points. However, these points should be processed as events of the plane sweep because self-intersecting edges should exchange their positions at the sweep-line status at their intersection points.

Fortunately, a small change in the algorithm can make it work for this kind of polygons. It is enough to find and process intersection points not only between the edges of different polygons, but also of the same polygon. In this case, self-intersecting edges will be also subdivided at their intersection points, and therefore, the result polygons will not contain self-intersections.

8. Evaluation

In this section we compare Greiner and Hormann's and Vatti's algorithms with the one presented in this paper. We have implemented Greiner and Hormann's and our algorithm in C++, due to Vatti's algorithm being difficult to implement we have used the implementation available at the website.[1] To implement our algorithm we have used an STLs priority queue container to represent the event queue and an STLs set container to represent the status line. The programs have been executed on a Intel Pentium IV processor at 2.4 GHz under Linux. Fig. 12 shows a polygon representing the coastline of the

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

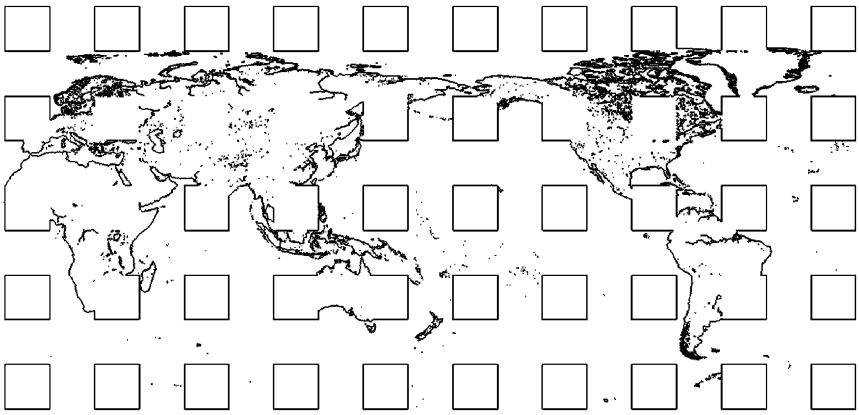

Fig. 14. Union.

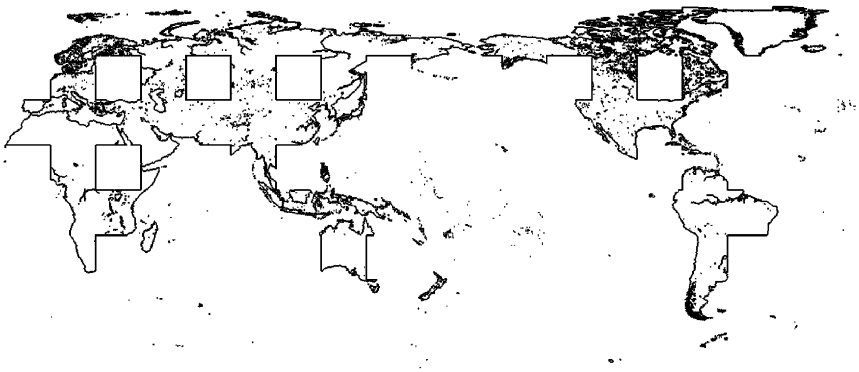

Fig. 15. Difference.

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

Number of vertices	Greiner	Vatti	New algorithm	Number of intersections
76 696 × 32	0.03	0.55	0.16	178
76 696 × 648	0.21	0.58	0.16	814
76 696 × 3895	4.01	2.16	0.22	4294
76 696 × 15 580	17.04	6.30	0.38	8978

Earth and its main lakes and a second polygon set consisting in several squares. Figs. 13, 14 and 15 show the result of the Boolean intersection, union and difference, respectively.

We have computed the intersection of the polygon representing the Earth with several polygon sets with an increasing number of squares, the result are presented in Table 1. The last column of the table presents the number of intersections between the edges of the polygons. Clearly, our algorithm performs better when the number of edges is increased. To understand this it must be said that Boolean operation algorithms spend the majority of their CPU time computing the intersection points between the polygons. The analyzed algorithms use different approaches to compute these points:

- Greiner and Hormann's algorithm uses the brute force approach. Of course, Greiner and Hormann's algorithm could also use a plane sweep technique to compute the intersection points for large 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.
- 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.

9. Conclusions

In this paper we have proposed a new algorithm for computing Boolean operations on polygons. The algorithm

is based on the classical plane sweep technique for computing the intersection points between a set of segments. Our algorithm subdivides the edges of the polygons at their intersection points. This subdivision makes the algorithm quite simple, allowing an elegant way of processing degeneracies.

The proposed algorithm computes a Boolean operation in time $O((n+k)\log(n))$, where n is the total number of edges of all the polygons involved in the Boolean operation and k is the number of intersections of all the polygon edges.

Unlike some approaches, the proposed algorithm does not need to be adapted to work with polygons with holes, and with regions composed of polygon sets.

Acknowledgments

This work has been partially granted by the Ministerio de Ciencia y Tecnología of Spain and the European Union by means of the ERDF funds, under the research project TIN2007-67474-CO3-03, and by the Conserjería de Innovación, Ciencia y Empresa of the Junta de Andalucía and the European Union by means of the ERDF funds, under the research projects P06-TIC-01403 and P07-TIC-02773.

We also would like to thank the anonymous reviewers for their helpful comments.

References

Andereev, R.D., 1989. Algorithm for clipping arbitrary polygons. Computer Graphics Forum 8 (2), 183–191.

Foley, J.D., Van Dam, A., Feiner, S.K., Hughes, J.F., 1990. Computer Graphics: Principles and Practice. Addison-Wesley, Reading, MA, 1174 pp.

Greiner, G., Hormann, K., 1998. Efficient clipping of arbitrary polygons. Association for Computing Machinery—Transactions on Graphics 17 (2), 71–83.

Liang, Y.D., Barsky, B.A., 1983. An analysis and algorithm for polygon clipping. Communications of the Association for Computing Machinery 26 (11), 868–877.

Liu, Y.K., Wang, X.Q., Bao, S.Z., Gomboi, M., Žalik, B., 2007. An algorithm for polygon clipping, and for determining polygon intersections and unions. Computers & Geosciences 33, 589–598.

Preparata, F., Shamos, M., 1985. Computational Geometry an Introduction. Springer, New York, NY, 398 pp.

Schneider, P.J., Eberly, D.H., 2003. Geometric Tools for Computer Graphics. Elsevier Science, San Francisco, CA, 1060 pp.

Sutherland, I.E., Hodgeman, G.W., 1974. Reentrant polygon clipping. Communications of the Association for Computing Machinery 17 (1), 32–42.

Vatti, B.R., 1992. A generic solution to polygon clipping. Communications of the Association for Computing Machinery 35 (7), 56–63.

Weiler, K., 1980. Polygon comparison using a graph representation. In: Proceedings of the 7th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH), Seattle, Washington, USA, pp. 10–18.