A CLOSED SET OF ALGORITHMS FOR PERFORMING SET OPERATIONS ON POLYGONAL REGIONS IN THE PLANE

Michael V. Leonov, Alexey G. Nikitin

A. P. Ershov Institute of Information Systems, Siberian Branch of Russian Academy of Sciences

This paper presents a simple and fast algorithm for computing the union, the intersection, the difference and the symmetrical difference of polygonal regions in the plane. The algorithm explicitly copes with degenerate cases and vertices of high degree so that the output of the algorithm satisfies its input restrictions. An expected running time of the algorithm is $O(n \log^* n + k + z \log n)$, where n (resp. z) is the total number of edges (resp. contours) in polygons describing input regions and k is the number of intersections between the edges. The presented algorithm outperforms most of known algorithms for polygon set operations and is relatively easy to implement.

INTRODUCTION

Boolean set operations on polygons are essential in various applications such as computer graphics, GIS, CAD/CAM, circuit design etc. The result of a set operation on even two simple convex polygons can possibly be a set of concave self-touching polygons with holes (see Fig. 1).

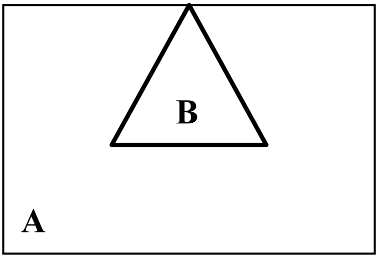


Figure 1

Unfortunately, the traditional algorithms either do not cope with degenerate cases at all or break complex output polygons into simple ones, making the operations on the resulting polygon(s) impossible without additional preprocessing steps. Nevertheless, in various GIS and CAD applications there is a need for a closed set of polygon operations. This means that the output of an algorithm must satisfy its input restrictions. It is also desireable to allow holes and vertices of degree higher than two. Let us make a brief survey of the existing algorithms for polygon set operations.

Weiler and Atherton [5. , 12. , 13.] proposed the following algorithm. First, all intersections of input polygon edges are determined. A complex set of rules is used for the different classes of intersections to rearrange the contours so that none of them intersects. Then each contour is traversed to collect the information about its owner(s). Finally, the nesting structure of the contours is determined and the resulting polygon subset is selected from this structure. The main disadvantages of this algorithm are:

- the result of an operation must not necessarily consist of the minimum contours (see Table 4, $A \neq B$),
- the problem of handling and describing self-touching polygons is not considered at all, though the algorithm can yield these polygons as its output,
- the output of the algorithm can contain vertices of high degree, which are not allowed on its input, and it is extremely difficult to extend the algorithm to cope with such cases.

Schutte [9. , 10.] modified the Weiler algorithm by much more clear division into steps and the different algorithm for edge labeling. However, his algorithm has stronger input restrictions.

The above algorithms give a non-closed set of polygon operations, because their output can contain self-touching contours (which are not allowed by both of them) and holes, which are not allowed by the Schutte algorithm.

In this paper we describe a simple and fast algorithm which is free of the disadvantages mentioned above. The paper is organized as follows. In section 2 the basic definitions and denotations are given, section 3 describes the algorithm itself, in sections 4 and 5 we give the analysis of the algorithm efficiency and discuss some implementation details.

BASIC DEFINITIONS

Let $E(a, b)$ denote the edge starting at point a and ending at point b, which is not coincident with a. For b (resp. a) we will call the edge E previous (resp. next). A set of points x satisfying the conditions $0 < \angle(bax) < \varphi$ and $2\pi-\varphi < \angle(abx) < 2\pi$ is called the left neighborhood of edge $E(a, b)$. Similarly, a set of points x satisfying the conditions $0 < \angle(abx) < \varphi$ and $2\pi-\varphi < \angle(bax) < 2\pi$ is called the right neighborhood of edge $E(a, b)$ (see Fig. 2a).

Definition 1. A contour C is the ordered set of n edges $\{E_0, E_1, \dots, E_{n-1}\}$, $n \geq 3$, so that $\forall\ i \in \{0\dots n-1\}$ $E_{i-1} = E(v_{i-1}, v_i)$ and $E_i = E(v_i, v_{i+1})$[1].

Point v_i, for which the edge E_{i-1} (resp. E_i) is previous (resp. next), is called ith vertex of the contour C, in this case the edges E_{i-1} and E_i are called *incident*. The order of tracing the edges of the contour $C = \{E_0, E_1, \dots, E_{n-1}\}$ by increasing the edge index ($i \to i + 1$) is called *straight*, the opposite order is called *reverse*.

Note that the same geometric point in the plane can correspond to several vertices of a contour. Such vertices are called the *high degree* vertices. The *angle of next* (resp. *previous*) *edge* $E(a,v)$ (resp. $E(v,b)$) for a vertex v is defined as the directed angle between vectors e_1 (unit vector of the x-axis) and $E(v,a)$ (resp. $E(v,b)$).

[1] Here and furthermore all index ariphmetics is made in modulo n, where n is a number of vertices in currently considered contour.

Consider a pair of incident edges E_{i-1} and E_i of a contour C: $0 \le i \le n - 1$. A set of points x satisfying the conditions $0 < \angle(v_{i+1}\, v_i\, x) < \angle(v_{i+1}\, v_i\, v_{i-1})$ and $0 < |x - v_i| < \varepsilon$, is called the *inner* ε*-neighborhood of vertex* v_i and denoted as $\varepsilon^{+}(v_i)$. A set of points x satisfying the conditions $0 < \angle(v_{i-1}\, v_i\, x) < \angle(v_{i-1}\, v_i\, v_{i+1})$ and $0 < |x - v_i| < \varepsilon$ is called the *outer* ε*-neighborhood of vertex* v_i and denoted as $\varepsilon^{-}(v_i)$ (see Fig. 2b).

From now our setting will be Eucledian plane with usual Cartesian system.

Definition 2. *Domain* is a finite non-empty polygonal open connected set on Eucledian plane, defined with a precision of set of measure zero.

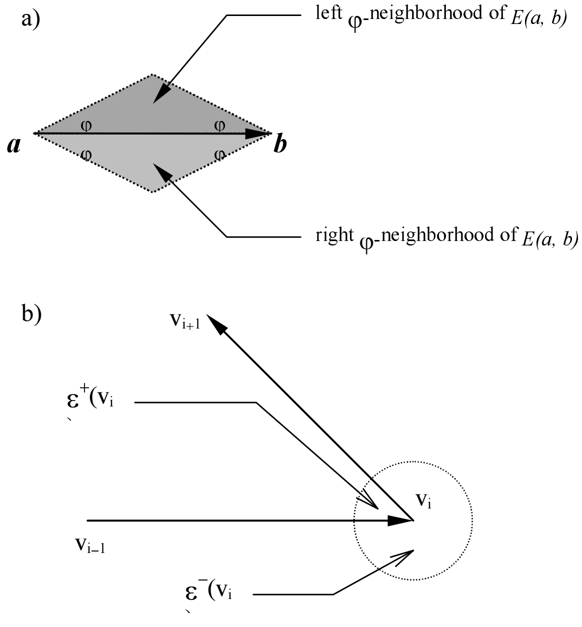


Figure 2

“Polygonal” means that the domain boundary consists of closed polylines. The precision of set of measure zero removes from consideration domains with “gaps” and “point holes”. Note that domain can be both single-connected and multi-connected set.

Let ***dA*** be the boundary of a domain ***A***, so that the orientations of ***dA*** and ***A*** are coordinated. To obtain bijection between ***dA*** and ***A***, we add to the traditional boundary representation two important restrictions. Consider contour ***C*** consisting of **n** edges.

Definition 3. Contour C is called the *bounding contour* of a domain A, if it satisfies the following conditions:

1. $C \subset dA$;
2. when C is traced in straight order, domain A is on the left;
3. $\forall\ i \in \{0 \dots n-1\}\ \exists\ \ \varepsilon > 0 \colon \forall\ \ x \in \varepsilon^{+}(v_i)\quad x \in A$;

   $\forall\ i \in \{0 \dots n-1\}\ \forall\ \ \varepsilon > 0 \colon \exists\ \ x \in \varepsilon^{-}(v_i)\quad x \notin A$.

Definition 4. *Polygon* is a set of all bounding contours of a domain.

The *outer* (resp. *inner*) contours of A are defined as the contours of its polygon with counter-clockwise (resp. clockwise) orientation. Obviously there is exactly one outer contour in an arbitrary polygon

Definition 5. *Region* is a set of polygons describing a set of non-intersecting domains.

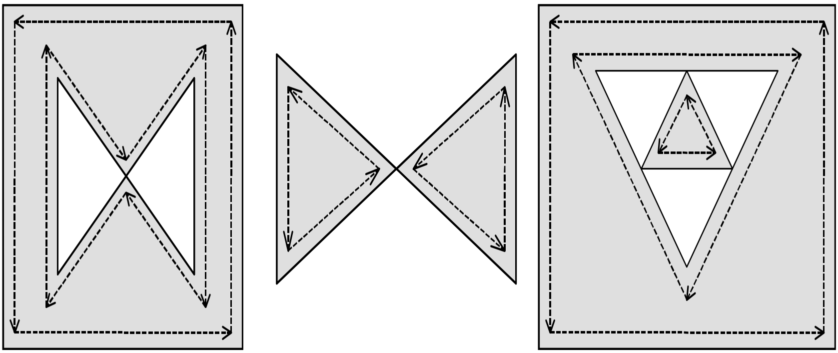


Figure 3

Thus we have a bijection between a set of non-intersecting domains (i.e. arbitrary polygonal area in the plane) and the set of its bounding contours. The examples of the bijection are shown in Fig. 3.

THE ALGORITHM

Input: two regions A and B, operation op = {*union*, *intersection*, *difference*, *symmetrical difference*}.

Output: region $R = A\ op\ B$.

The algorithm consists of the following steps.

Step 1. Processing of the edge intersections.

Step 2. Edge and contour labeling.

Step 3. Collecting the resulting contours.

Step 4. Placing the resulting contours in R.

Fig. 4 shows example regions *A* and *B*. According to our definitions, region A is a polygon consisting of one outer and one inner contours, and region B is a polygon consisting of one outer self-touching contour.

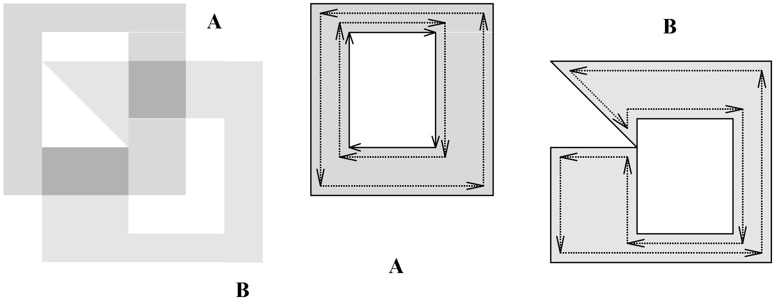


Figure 4

Processing of the edge intersections

Assume we have an algorithm, which reports all intersections between non-incident edges of regions *A* and *B*. If the edges share common line segment, we will treat its endpoints as the intersection points. All new intersection points are added as vertices to the input regions. Hence insertion of the intersection points does not change domains whose interiors are described by *A* and *B*, furthermore *A* and *B* will mean input regions *A* and *B* with all their intersection points added. Vertices corresponding to intersection points are called *cross-vertices* (Fig. 5). Note that one geometric intersection point corresponds to at least two cross-vertices.

Let v be a cross-vertex of some region. Let $E^{+}(v)$ (resp. $E^{-}(v)$) denote its previous (resp. next) edge. The cross-vertex is processed as follows.

Two cross-vertex descriptors $D^{+}(v)$ and $D^{-}(v)$ are created, which correspond to $E^{+}(v)$ and $E^{-}(v)$. Into these descriptors are placed the values of their edge angles and flags about if the edges are previous or next to v.

1. The pointers between v and its descriptors $D^+(v)$ and $D^-(v)$ are established, allowing determining the descriptors for a vertex and vice versa.

2. $D^+(v)$ and $D^-(v)$ are placed into *connectivity list* $L(x)$ *of intersection point* x. This list contains descriptors of all cross-vertices corresponding to x and is sorted by edge angles. Furthermore, $L(x)$ will be used for searching for counter-clockwise and clockwise neighbors of an edge. The order of descriptors with equal edge angles is not specified.

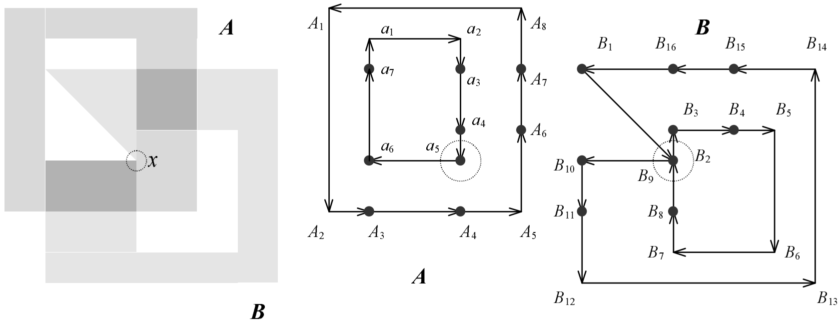


Figure 5

Fig. 6 shows the connectivity list of point x (marked in Fig. 5). The corresponding cross-vertices are a_5, B_2 and B_9.

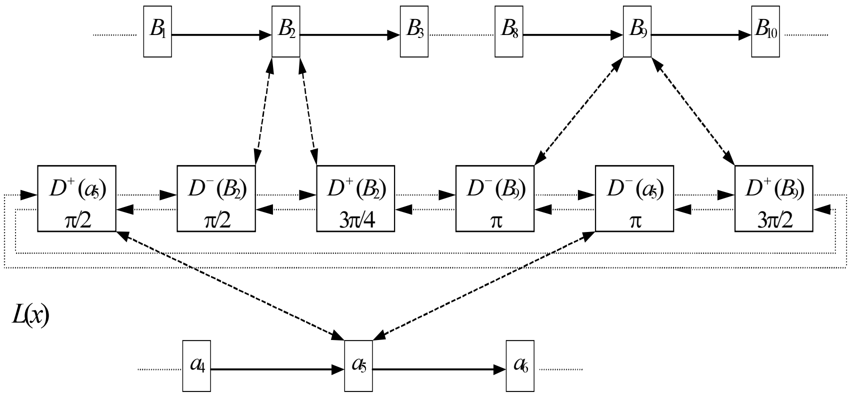


Figure 6

A cross-vertex is considered to belong to a connectivity list when its descriptors belong to the list. Since a cross-vertex may belong only to the connectivity list of corresponding geometric point, we introduce the notion of a *cross-vertex connectivity list*.

Edge and contour labeling

Let C be a bounding contour of a region A or B. Let M be region, which C does not belong to.

Since in step 1 all intersection points were added to the input regions, taking into account the definitions 3 — 5 implies the following predicate.

Predicate 1. After the first step of the algorithm:

1) Unequal edges of regions A and B intersect at their endpoint or not at all;

2) Every edge of C either coincides with an edge of M (such edges are called *shared*), or lies inside or outside (possibly except its endpoints) M;

3) If C does not have cross-vertices, it lies inside or outside M.

Let E be an edge of a contour C. Then its *label* has one of the following values:

INSIDE — when E (possibly except its endpoints) lies inside M,

OUTSIDE — when E (possibly except its endpoints) lies outside M,

SHARED1 — when E coincides with an edge from region M with the same direction,

SHARED2 — when E coincides with an edge from region M with the opposite direction.

The *label* of contour C has one of the following values:

ISECTED — when C contains at least one cross-vertex,

INSIDE — when C lies inside M,

OUTSIDE — when C lies outside M.

The correctness of edge and contour label definitions immediately follows from Predicate 1.

Here is an algorithm for labeling C and its edges.

1. *C does not contain any cross-vertices.*

If C lies inside M, it is labeled as *INSIDE,* otherwise — as *OUTSIDE*. The edges of C are not labeled at all.

2. *C contains at least one cross-vertex.*

C is labeled as *ISECTED* and all its edges are sequentially labeled. Let $E_i(a, b)$ ($i \in \{0\ldots n-1\}$, where n is the number of edges in C) be the edge to label.

2.1 *E_i does not contain cross-vertices as its endpoints.*

1) $i \neq 0$. The edge label value is copied from the edge E_{i-1}.

2) $i = 0$. If a lies inside M, then E_i is labeled as *INSIDE*, otherwise E_i is labeled as *OUTSIDE*.

2.2 *E_i contains one or two cross-vertices as its endpoints.*

1) $\exists$ *edge* $F(c, d)$: $F \in M \wedge a = c \wedge b = d$. E_i is labeled as *SHARED1*.

2) $\exists$ *edge* $F(c, d)$: $F \in M \wedge a = d \wedge b = c$. E_i is labeled as *SHARED2*.

3) *Cross-vertices connectivity list(s) does not contain any vertices from M.*
Such situation is possible if C is not intersected by M and touches itself. The edge is labeled similarly to step 2.1.

4) *Cross-vertices connectivity list(s) contain vertices from M.*
For each such vertex w_k the check is performed if E_i lies inside $\angle(w_{k-1}w_kw_{k+1})$. If E_i is inside one of such "labeling" angles, then it lies inside M and therefore is labeled as *INSIDE*. Otherwise, E_i is labeled as *OUTSIDE* (see Fig. 7).

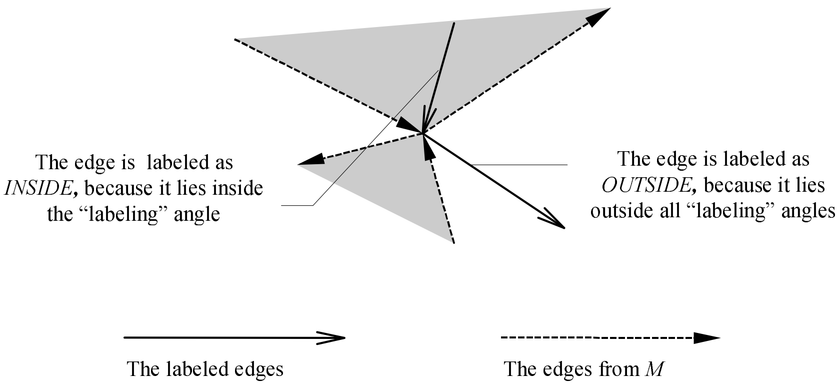


Figure 7

Collecting the resulting contours

Preliminaries

One of the essential ideas of the presented algorithm is that bounding contours of R (furthermore they are called *resulting contours*) are collected using the values of edge and contour labels instead of their coordinates. Since an edge cannot appear in a region more than once, we are to find the *rules* for edge and contour inclusion into R.

Fig. 8 shows our example regions A and B after performing labeling step.

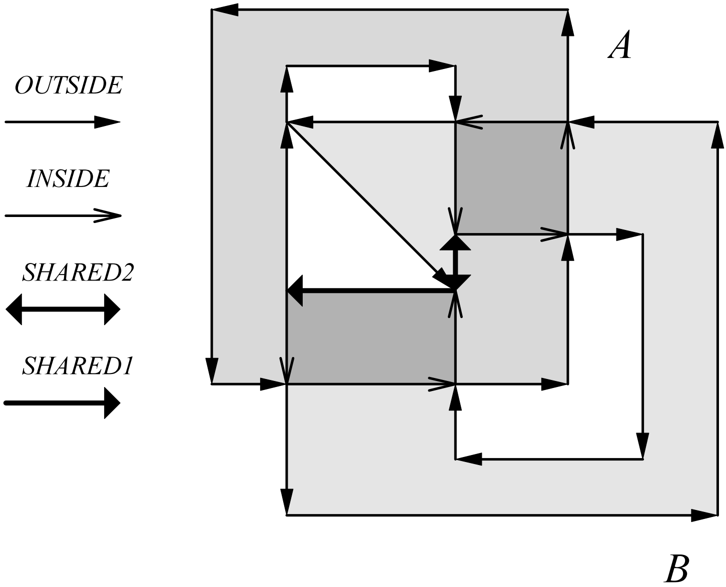


Figure 8

The rules for edge inclusion into a resulting contour

Consider some edge E belonging to either A or B. Let $\varphi > 0$ be the maximum real number such that neither left, nor right φ-neighbourhood of edge E do not contain any other vertices or edges from A or B. These neighbourhoods are called the *minimum* left and right neighbourhoods of E and are denoted respectively as $\varphi^{+}(E)$ and $\varphi^{-}(E)$. The choice of φ implies that all points from a minimum neighbourhood of an edge are inside or outside A or B at the same time.

Now we show how the rules for edge inclusion into a resulting contour follow from those of minimum neighbourhoods.

1. If $\varphi^{+}(E)$ belongs to R, but $\varphi^{-}(E)$ does not, E will be included into a resulting contour with its original direction.

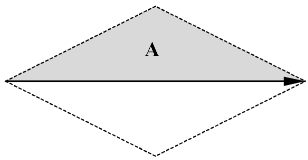
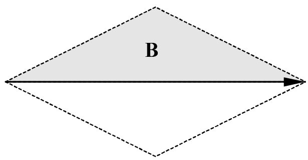
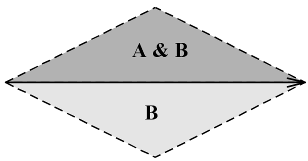
2. If $\varphi^{-}(E)$ belongs to R, but $\varphi^{+}(E)$ does not, E will be included into a resulting contour with inverse direction.

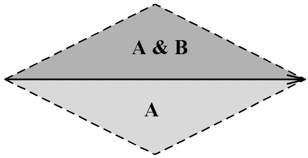
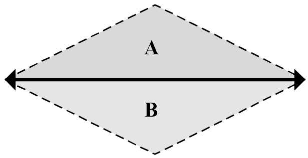
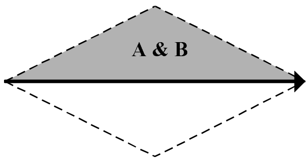
3. If both $\varphi^{-}(E)$ and $\varphi^{+}(E)$ do not belong to R, E will not be included into a resulting contour.

4. If both $\varphi^{-}(E)$ and $\varphi^{+}(E)$ belong to R, E will not be included into a resulting contour (this would contradict definition 3, clause 4).

Note that only one edge from a pair of shared edges can be included in result. Table 1 shows the inclusion rules for all combinations of edge labels for both regions.

Table 1

<table>
<tr><td>
</td><td>Case 1.
E belongs to A and its label is OUTSIDE. $\varphi^{+}(E)$ belongs to A, $\varphi^{-}(E)$ does not belong to any of the input regions. E will be added with the original direction into the union, the difference and the symmetrical difference of A and B.</td></tr>
<tr><td>
</td><td>Case 2.
E belongs to B and its label is OUTSIDE. $\varphi^{+}(E)$ belongs to B, $\varphi^{-}(E)$ does not belong to any of the input regions. E will be added with the original direction into the union and the symmetrical difference of A and B.</td></tr>
<tr><td>
</td><td>Case 3.
E belongs to A and its label is INSIDE. $\varphi^{+}(E)$ belongs to both A and B, $\varphi^{-}(E)$ belongs to B. E will be added with the original direction into the intersection of A and B, and with the inverse direction into the symmetrical difference of A and B.</td></tr>
</table>

<table>
<tr>
<td>
</td>
<td>Case 4.
E belongs to B and its label is INSIDE. $\varphi^{+}(E)$ belongs to both A and B, $\varphi^{-}(E)$ belongs to A. E will be added with the original direction into the intersection of A and B, and with the inverse direction into the difference and the symmetrical difference of A and B.</td>
</tr>
<tr>
<td>
</td>
<td>Case 5.
A pair of edges $E \in A$ and $F \in B$ with labels SHARED2. $\varphi^{+}(E)$ belongs to A, a $\varphi^{+}(F)$ belongs to B. One of the edges will be added with the original direction of E into the difference of A and B.</td>
</tr>
<tr>
<td>
</td>
<td>Case 6.
A pair of edges A and B with labels SHARED1. The minimum left neighbourhood of the edges belongs to both A and B, the minimum right neighbourhood of the edges does not belong to any of the input regions. One of the edges will be added with the original direction into the union and the intersection of A and B.</td>
</tr>
</table>

The jump rules at the intersection points

Suppose while collecting the resulting edges we just came to a cross-vertex v. Where will we go next? Let x be geometric point corresponding to v. Consider all cross-vertices corresponding to x. Let m be the number of such vertices, $m \geq \mathbf{2}$. Then the number of their descriptors is $2m$ (see Step 1). With a help of $L(x)$ we can obtain the set of edges F_{j}, $j \in \{\mathbf{0} \ldots 2m - \mathbf{1}\}$ so that $F_0 = E^{+}(v)$ and the descriptor of $F_{\mathrm{j+1}}$ is the nearest in the clockwise direction to the descriptor of F_{j}.

Let $\varepsilon > 0$ be the maximum real number such that no inner or outer ε-neighbourhood of considered vertices contain vertices or edges different from F_{j}. Consider the sectors into which geometric ε-neighbourhood of point x is divided by edges F_{j}. Such sectors are called the *minimum* sectors. Apparently, all points of a minimum sector (possibly except its boundaries) are inside or outside A or B at the same time. If the points of two neighbour minimum sectors belong to R, by definition 3 they should be bounded by the same resulting contour.

We find the desired edge by sequentially testing edges F_j, beginning from F_0 and increasing j, to satisfy the inclusion rule for desired operation. Then we can go along this edge. Note that for the search the geometric information is not required at all, because descriptors are sorted in connectivity list by corresponding edge angles.

The collecting algorithm

Since the previous discussion applies to all operations, the algorithm collects resulting contour for all operation in the similar way. In this step we sequentially consider each bounding contour of A and B. Let C be the contour currently being considered.

The label of contour or edge X is furthermore denoted as *X.Flags*. Also *FORWARD* and *BACKWARD* will denote the original and the inverse directions of edges. Here is the algorithm for collecting the resulting contours.

1. If $C.Flags \neq ISECTED$, C is added to R depending on its label and operation *op* (see Table 2).

Table 2

op	The rule for inclusion of C into R	Direction
$\cap$	$(C.Flags = INSIDE)$	*FORWARD*
$\cup$	$(C.Flags = OUTSIDE)$	*FORWARD*
$-$	$(C.Flags = OUTSIDE) \wedge (C \in A)$	*FORWARD*
	$(C.Flags = INSIDE) \wedge (C \in B)$	*BACKWARD*
$\neq$	$(C.Flags = OUTSIDE)$	*FORWARD*
	$(C.Flags = INSIDE)$	*BACKWARD*

2. If $C.Flags = ISECTED$, we need to find in C all edges which resulting contours can start from. Here is the algorithm for processing C (n is the number of vertices in C). Each edge E has a bit flag *E.Mark* indicating if the edge has already included into one of the resulting contours, initially the flag is set to false for all edges.

for i := **0 to n−1 do**
begin

```
        if (EdgeRule(Ei, dir) and not Ei .Mark)
        begin
                contour r;
                if (dir = FORWARD)
                        r := Collect(vi, dir);
                else
                        r := Collect(vi+1, dir);
                Include r into a set of resulting contours;
        end;
end;
```

where **function EdgeRule(E:edge; var dir:(FORWARD, BACKWARD)): boolean** is the edge inclusion rule for currently performed operation (see Table 3).

Table 3

Op	The rule for inclusion of E into R	Direction
$\cap$	$(E.Flags = INSIDE) \vee (E.Flags = SHARED)$	*FORWARD*
$\cup$	$(E.Flags = OUTSIDE) \vee (E.Flags = SHARED1)$	*FORWARD*
$-$	$((E.Flags = OUTSIDE) \vee (E.Flags = SHARED2)) \wedge (E \in A)$	*FORWARD*
	$((E.Flags = INSIDE) \vee (E.Flags = SHARED2)) \wedge (E \in B)$	*BACKWARD*
$\oplus$	$(E.Flags = OUTSIDE)$	*FORWARD*
	$(E.Flags = INSIDE)$	*BACKWARD*

```
function Collect(v:vertex; dir:(FORWARD, BACKWARD)):contour;
begin
        Create an empty contour r;
        repeat
                Add v to r;
                if (dir = FORWARD)
                        E := edge next to v;
                else
                        E := edge before v;
                E.Mark := true;
                if ((E.Flags = SHARED1) or  (E.Flags = SHARED2))
```

```
                    for edge, shared with E, set its Mark to true;
            v := vertex next to v in direction dir;
            if (v is a cross-vertex) then
                    Jump(v, dir);
        until (current edge is marked);
        return r;
end;

procedure Jump(var v:vertex; var dir:(FORWARD, BACKWARD));
begin
        if (dir = FORWARD) then
                d := prev(D+(v));
        else
                d := prev(D−(v));
        { prev(D) denotes descriptor nearest to D in clockwise direction }
        found := FALSE;
        repeat
                e := edge corresponding to d;
                if (not (e.Mark) and EdgeRule(e, newdir)) then
                begin
                        v := vertex corresponding to d;
                        if ((e is next to v) and (newdir = FORWARD) or
                        (e is previous to v) and (newdir = BACKWARD)) then
                        begin
                                dir := newdir;
                                found := TRUE;
                        end;
                end;
                d := prev(d);
        until (found);
end;
```

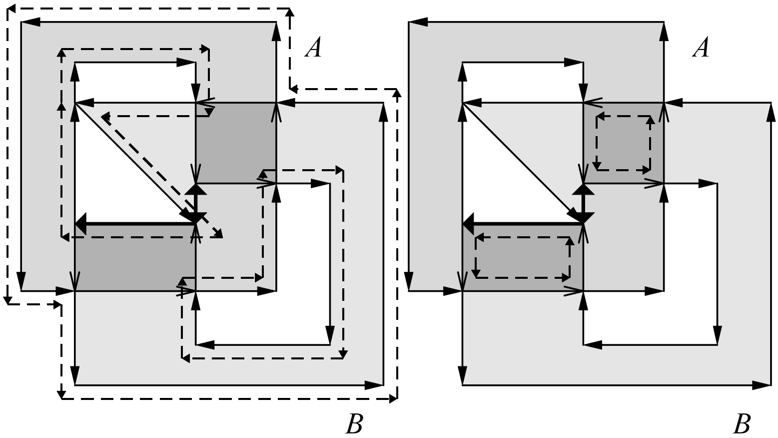
Fig. 9 shows the resulting contours for different operations on example regions *A* and *B*.

The *union* is composed of one outer and two inner contours, both the *difference* and the *intersection* are composed of two outer contours, the *symmetrical difference* is composed of three outer contours.

Placing the resulting contours in R

After previous steps, we have obtained a set of bounding contours of *R*. Arranging them into *R* is very simple because they already have proper orientation.

For each outer contour, a new polygon is created. Inner contours were stored in temporary list in previous stage, now they should be placed in proper polygons. For finding a proper polygon for an inner contour, it suffices to determine the minimum outer contour containing the inner contour. Region *R* is composed of polygons formed in this step.



$A \cup B$ $A \cap B$

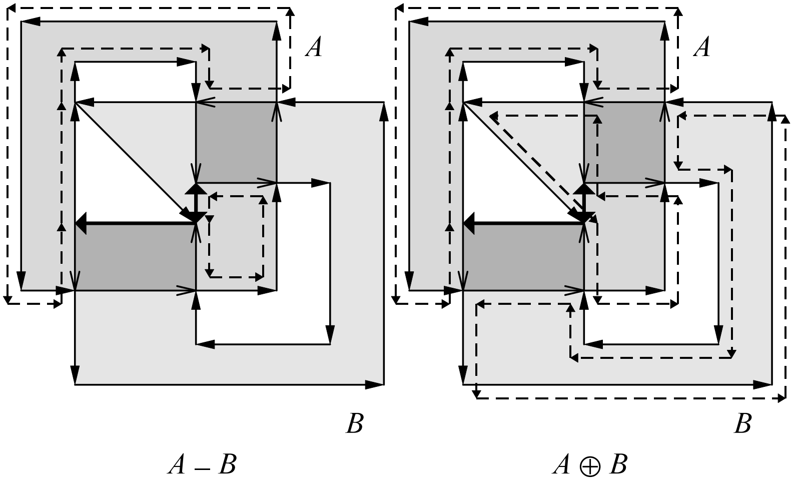


Figure 9

THE ANALYSIS OF THE EFFICIENCY

Now we analyze the efficiency of the presented algorithm. Let A and B be input regions with n vertices and z contours in total.

Let k be the number of new vertices added in step 1. In the worst case $k = O(n^2)$. Time complexity of step 1 depends on the algorithm for reporting edge intersections. The worst-case time complexity of testing all pairs of edges for intersection $O(n^2)$. There are more efficient algorithms [1. — 3.] with the time complexity $O(n \log n + k)$. Finke and Hinrichs [4.] presented an algorithm which can be extended to compute overlay of two path-connected planar subdivisions in $O(n \log^* n + k + z \log n)$ expected time.

The time complexity of step 2 is $O(n + z\, p(n))$, where $p(n)$ is the time for determining if a given point is inside a region consisting of n vertices. Without using additional data structures $p(n) = O(n)$. To speed up point location queries one can use a search structure created by Seidel's trapezoidation algorithm, which can be constructed in $O(n \log^* n + z \log n)$ time [11.]. This structure allows point location queries in $O(\log n)$ expected time.

Since in step 3 each edge is used not more than two times, its time complexity is apparently $O(n + k)$. Step 4 takes $O(z\, p(n + k))$ time.

Computational expenses of the algorithm can also be decreased by using minimum bounding boxes which can be computed in $O(n)$ time.

Thus, the presented algorithm works in $O(n \log^* n + k + z \log n)$ expected time and $O(n + k)$ space.

CONCLUSION

Let us summarize the main results of this paper.

- Mathematically non-ambiguous definitions are given for description of polygonal regions in the plane.
- The technique for explicit handling vertices of high degree is developed, which avoids degeneracy problems of traditional algorithms.
- A set of intuitive clear rules for collecting the resulting contours is introduced, which allow compact and efficient implementation of the presented algorithm.

The main contribution of the presented paper is a *closed* set of operations.

The implementation of the algorithm consists of about 850 lines of C-code. In practice, the performance of the algorithm meets theoretical requirements very well.

The first author would like to thank Klamer Schutte for helpful discussions on polygon clipping and Ralf Gueting, Raimund Seidel and Jack Snoeyink for assistance with materials for the paper.

References

1. Further comparison of algorithms for geometric intersection problems / D. S. Andrews, J. Snoeyink, J. Boritz a. o. // Proc. 6th Internat. Sympos. on Spatial Data Handling.-1994.- P. 709-724.

2. An optimal algorithm for finding segments intersections / I. J. Balaban // Proc. 11th Annual ACM Sympos. on Computational geometry.-1995.- P. 211-219.

3. A simple trapezoid sweep algorithm for reporting red/blue segment intersections / T. Chan // Proc. 6th Canadian Conf. on Computational Geometry.- 1994.- P. 263-268.

4. Overlaying simply connected planar subdivisions in linear time / U. Finke, K. H. Hinrichs // Proc. 11th Annual ACM Sympos. on Computational geometry.- 1995.- P.119-126.

5. Computer graphics: principles and practice / J. D. Foley, A. van Dam, S. K. Feiner, J. F. Hughes // Addison-Wesley, 1990.

6. Realms: a foundation for spatial data types in database systems / R. H. Gueting, M. Schneider // Proc. 3rd Internat. Sympos. on Large Spatial Databases.- 1993.- P. 14-35.

7. Realm-based spatial data types: the ROSE algebra / R. H. Gueting, M. Schneider // VLDB Journal 4.- 1995.- P. 100-143.

8. Implementation of the ROSE algebra: efficient algorithms for realm-based spatial data types / R. H. Gueting, M. Schneider, Th. de Ridder // Proc. 4th Internat. Sympos. on Large Spatial Databases.- 1995.- P. 216-239.

9. An edge labeling approach to concave polygon clipping / K. Schutte // unpublished manuscript, online version is available at [http://www.ph.tn.tudelft.nl/~klamer/clip.ps.gz].

10. Knowledge based recognition of man-made objects / K. Schutte // Ph.D. Thesis.- University of Twente, ISBN 90-9006902-X, 1994.

11. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons / R. Seidel // Computational Geometry: Theory and Applications.- 1991.- Vol.1, N1.-P. 51-64.

12. Hidden surface removal using polygon area sorting / K. Weiler, P. Atherton // Computer Graphics.- 1977.- Vol.11, N2.- P. 214-222.

13. Polygon comparison using a graph representation / K. Weiler // Computer Graphics.- 1980.- Vol. 14, N3.- P. 10-18.