A Quick Intersection Algorithm for Arbitrary Polygons

Borut Žalik, Matej Gomboši, and David Podgorelec
University of Maribor
Faculty of Electrical Engineering and Computer Science
Smetanova 17, SI-2000 Maribor, Slovenia.
e-mails: {zalik, matej.gombosi, david.podgorelec}@uni-mb.si

Abstract. The paper describes an algorithm which finds an intersection between any two well-defined polygons. The algorithm works in three steps. In the first step, the intersection points of the polygon edges are calculated by the use of a sweep line. These points are needed in the second part of the algorithm where they are equipped with so-called navigation data. In the last part of the algorithm, an intersection polygon is determined by performing a walk-about through polygon edges and intersection points. By the use of the developed data structure, this task can be done safely and quickly.

Keywords: computational geometry, sweep-line algorithm, polygon intersection.

1 INTRODUCTION

Polygon intersection is one of the fundamental problems of the computational geometry with applications in various areas of computer graphics and CAD. The problem is considered from two viewpoints:

- there are only convex polygons at input, and
- at least one of the polygons is not convex (can be concave, can include nested holes to any level of depth).

The algorithms of the first type are well known and are relatively easy to implement (a robust implementation has, however, solve traps connected with the finite arithmetic) [PREP85, O'ROU93]. On the other hand, there is still a lack of quick and robust algorithms copping with intersections between general polygons and they are also frequently missing in a professional CAD software (for example in GIS). Many times, it is suggested to split such polygons into convex parts (the most popular methods are triangulation [BERG97] and trapezoidation [ZALI98]) and solve them individually. However, in real applications like in geodesy, this does not seem always a good idea (see Figure 1). In the case of such polygons, it is better to have an algorithm which copes directly with the intersection problem. Therefore, we have developed an algorithm with the following requirements in mind: at input, it should accept any well-defined polygon (self-intersected polygons are excluded), it should be quick enough to cope with the polygons like the one in Figure 1, it should be robust regarding finite arithmetic, and the implementation should not be too complicated. Such an algorithm is described in the paper.

Before we proceed, a few words are dedicated to the used vocabulary adopted from the theory of geometric modelling [MORT85]:

- *A loop* represents a boundary of a polygon and consists of a closed sequence of oriented polygon edges. Each polygon must have exactly one loop.
- *A ring* represents a hole in the polygon. There can be zero or finite number of rings in a considered polygon. The rings can be nested to any level of depth.

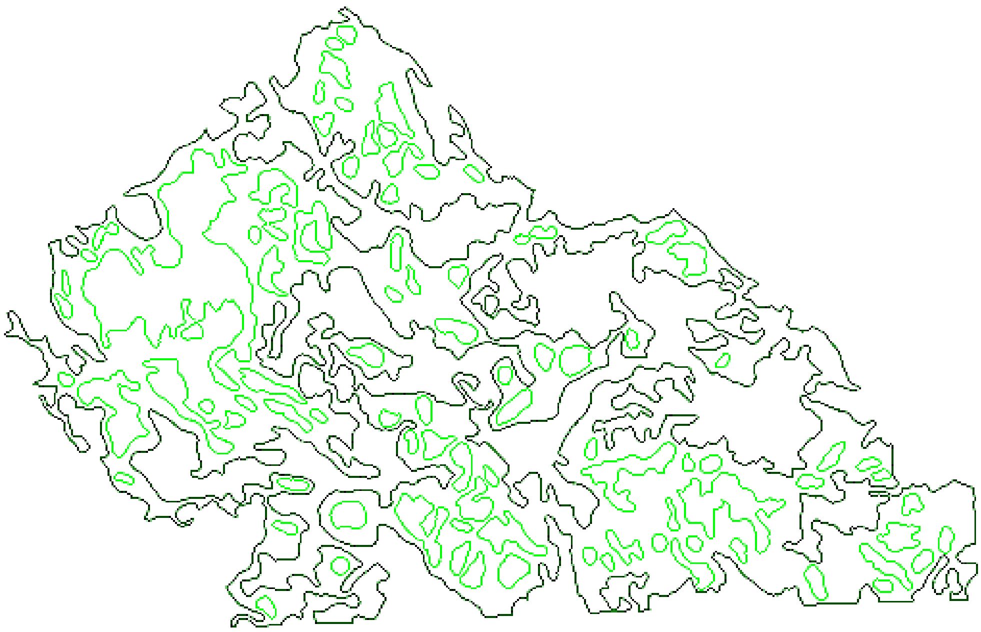

Figure 1. A polygon from a geodesic problem consisting of more than 28000 edges

2 THE ALGORITHM

Our algorithm for determination of the intersection of two arbitrary non-self-intersected polygons works in three steps:

1. It finds the intersection points between all edges of both polygons. A sweep line algorithm is developed for this task and it is described in Section 2.1.
2. When the intersection points are calculated, they are equipped by some additional, so-called navigation data, needed in the last part of the algorithm (Section 2.2).
3. In the last part, the algorithm takes the data prepared by previous two steps and constructs the intersection polygon by performing a walk-about from one intersection point to another.

2.1 Calculating the intersection points

A calculation of the intersection points between polygon edges is the most critical operation of the polygon intersection algorithm considering a time complexity. The use of the brute-force approach (trying to intersect each edge of the first polygon with each edge of the second one) leads us to the $\Omega(n \times m)$ time complexity, where n and m are the numbers of both polygon vertices. However, in some "ugly" cases, this time complexity cannot be avoided (see Figure 2).

Fortunately, in general, the number of intersections between polygon edges is much smaller than $n \times m$. For example, in Figure 3, polygon P has 7 and polygon Q 5 vertices. In the naive (brute-force) implementation, the algorithm checks all 35 pairs of polygon edges although only 8 intersections exist. It would be nice, however, to have an algorithm which would behave cleverer and "would guess" which edges intersect.

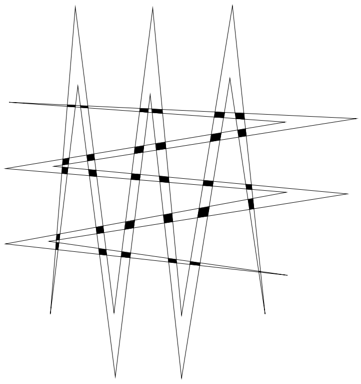

Figure 2. Number of intersection points between polygon edges is $n \times m$

For this purpose, a sweep line algorithm is used. The usage of the sweep line algorithms is very frequent in the computational geometry and computer graphics (i.e. scan line fill algorithm, trapezoidation algorithms), and all of them use the same simple idea. We imagine that the sweep line s (let us say it is a vertical) glides over the plane from $-\infty$ to $+\infty$. It stops only at special event points - at polygon vertices in our case - where the algorithm performs some tasks. In the case of searching for intersections of polygon edges, we are interested in those edges, which are pierced by the sweep line at the moment of observing. They are good candidates for the intersection. It is obvious that a set of edges pierced by the sweep line is changed only at the polygon vertices. This information is considered as the sweep-line status. The most well-known solution of the problem of finding all intersections between line segments in the plane uses two AVL trees - one for sweep-line status and the second for event-point schedule [PREP85, BERG97]. It can be easily adopted to polygon intersection problem. However, in the paper, a less efficient method considering time complexity is used, but it is much easier for an implementation.

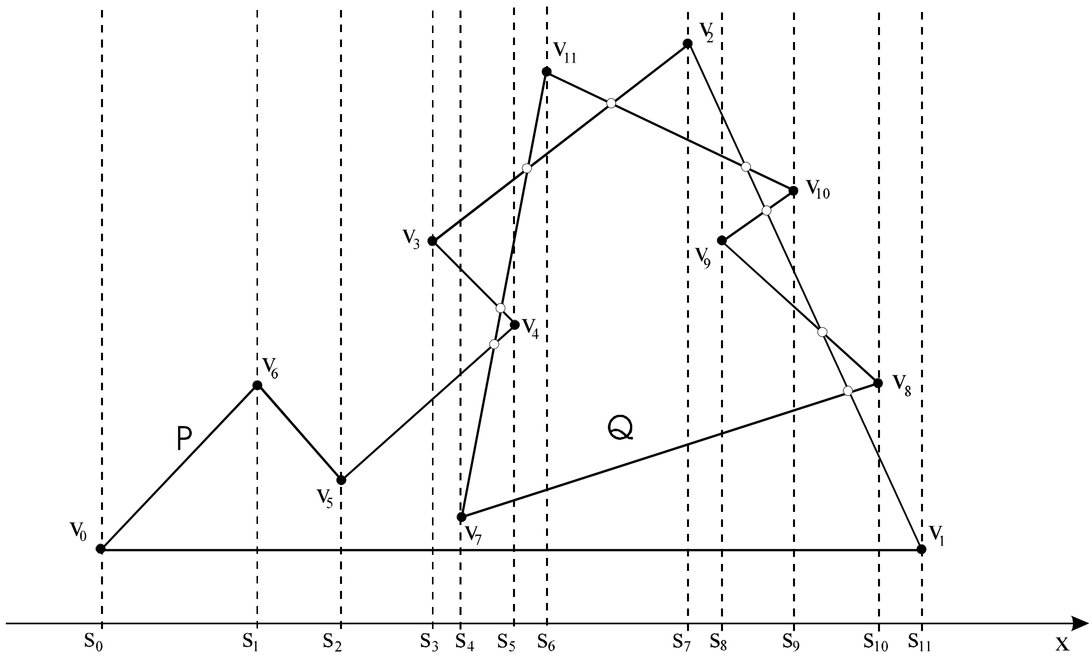

Figure 3. Polygon intersection points determination using the sweep line

Let us say vertex v_k lays at sweep line s_r. At this vertex, two edges $e_{k,i}$ and $e_{k,j}$ connecting it with vertices v_i and v_j meet each other. Let us introduce two sets of currently processed edges. The set S_p contains the edges of polygon P and the set S_q the edges of polygon Q. The situations shown in Figure 4 can occur and are treated by the following rules:

Rule a: $v_i > v_k$ & $v_j > v_k$; edges $e_{k,i}$ and $e_{k,j}$ are inserted into S_p ($v_k \in P$) or S_q ($v_k \in S_q$),

Rule b: $v_i < v_k$ & $v_j < v_k$; edges $e_{k,i}$ and $e_{k,j}$ are removed from S_p ($v_k \in P$) or S_q ($v_k \in S_q$),

Rule c: $v_j < v_k$ & $v_i > v_k$; edge $e_{k,i}$ is added and $e_{k,j}$ removed from S_p ($v_k \in P$) or S_q($v_k \in S_q$),

Rule d: $v_k > v_i$ & $v_j < v_k$; edge $e_{k,i}$ is removed and $e_{k,j}$ added to S_p ($v_k \in P$) or S_q ($v_k \in S_q$),.

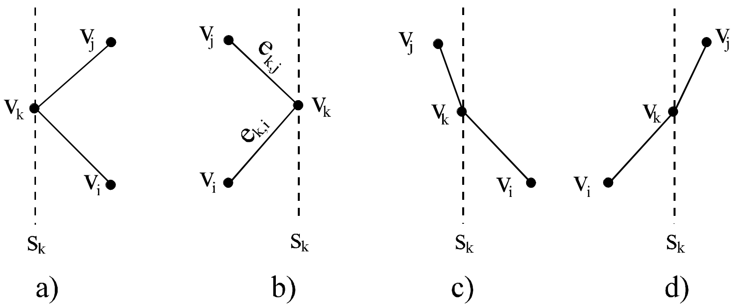

Figure 4. possible position of polygon vertices regarding the sweep line

Of course, it can happen that more than one vertex exist on the sweep line. Such cases can be easily solved, and are considered at the end of this section. Let us explain first how the algorithm works.

To enable our sweep line to glide smoothly over the plane (without jumps), the vertices of the input polygons have to be ordered in the direction of the sweep line movement (in the positive x direction in our case). There will be at most 12 stops of the sweep line in our example (see Figure 3). Let us see what the algorithm does during each of them:

1. stop: The first stop of the sweep line is at vertex v_0 (Figure 3). The **Rule a** is employed in this case and two edges $e_{0,1}$ and $e_{0,6}$ are inserted in the set of processed edges. Since these edges are the members of polygon P, they are added into set S_p. Set S_q is empty at the moment because sweep line has not yet reached polygon Q, and therefore there is not necessary to perform any check for the intersection.

$S_p = \{e_{0,1}, e_{0,6}\}$, $\quad S_q = \{ \}$.

2. stop: The sweep line moves to vertex v_6 and **Rule d** is used. Edge $e_{0,6}$ is removed and $e_{6,5}$ is inserted into S_p ($v_6 \in P$):

$S_p = \{e_{6,5}, e_{0,1}\}$, $\quad S_q = \{ \}$.

3. stop: The sweep line moves to vertex v_5 and **Rule d** is implemented again. The situation in our set is the following:

$S_p = \{e_{5,4}, e_{0,1}\}$, $\quad S_q = \{ \}$.

4. stop: The sweep line passes through the vertex v_3. The usage of **Rule a** adds two additional edges into set S_p:

$S_p = \{e_{3,2}, e_{3,4}, e_{5,4}, e_{0,1}\}$, $\quad S_q = \{ \}$.

5. stop: The sweep line reaches the first vertex of polygon Q - vertex v_7 which is connected with vertices v_8 and v_{11}. Because of this, there exists a possibility that edges $e_{7,8}$ and $e_{7,11}$ intersect the edges from set S_p. The following tests are applied:

$e_{7,11}$ & $e_{3,2}$: tried and found an intersection point	$e_{7,8}$ & $e_{3,2}$: not tried $y_{max}(e_{7,8}) < y_{min}(e_{3,2})$
$e_{7,11}$ & $e_{3,4}$: tried and found an intersection point	$e_{7,8}$ & $e_{3,4}$: not tried $y_{max}(e_{7,8}) < y_{min}(e_{3,4})$
$e_{7,11}$ & $e_{5,4}$: tried and found an intersection point	$e_{7,8}$ & $e_{5,4}$: tried and not found
$e_{7,11}$ & $e_{0,1}$: not tried $y_{max}(e_{0,1}) < y_{min}(e_{7,11})$	$e_{7,8}$ & $e_{0,1}$: not tried $y_{max}(e_{0,1}) < y_{min}(e_{7,8})$

After check, the status of set S_p and the status of set S_q are updated:

$S_p = \{e_{3,2}, e_{3,4}, e_{5,4}, e_{0,1}\}$, $S_q = \{e_{7,8}, e_{7,11}\}$.

6. stop: Vertex v_4 is reached. By the use of **Rule b** edges $e_{5,4}$ and $e_{4,3}$ are removed from S_p. Thus:

$S_p = \{e_{3,2}, e_{0,1}\}$, $S_q = \{e_{7,8}, e_{7,11}\}$.

The reader can now imagine how the process continues. For clarity, let us consider the last step when the sweep line reaches vertex v_8. The situation in our sets just before reaching vertex v_8 is

$S_p = \{e_{2,1}, e_{0,1}\}$, $S_q = \{e_{9,8}, e_{7,8}\}$.

At vertex v_8, **Rule b** is implemented giving us:

$S_p = \{e_{2,1}, e_{0,1}\}$, $S_q = \{\}$.

Since one of the sets of currently processed edges becomes empty, there do not exist any intersections more and we can terminate.

In our example, only one unsuccessful call to the line intersection routine has been made. Of course, with a simple check, it can be discovered that edge $e_{0,1}$ does not intersect for sure non of the edges of the polygon Q and it does not need at all to be inserted in the set of considered edges.

Special case arises when more than one vertex exist on the sweep line. Possible cases are shown in Figure 5 and can be handled very easily by the following rules:

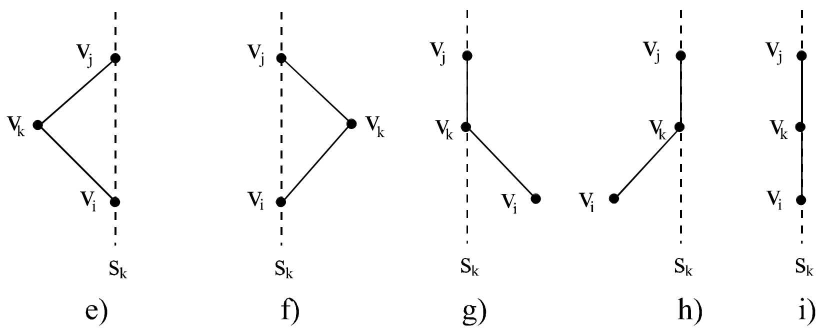

Figure 5. Boundary cases during the "sweep" with the line

Rule e: Both edges $e_{k,i}$ and $e_{k,j}$ are removed from $S_p(v_k\in P)$ or $S_q(v_k\in S_q)$,

Rule f: Both edges $e_{k,i}$ and $e_{k,j}$ are tried for intersection and added into $S_p(v_k\in P)$ or $S_q(v_k\in S_q)$,

Rule g: Both edges $e_{k,i}$ and $e_{k,j}$ are checked for intersection with edges from S_q $(v_i\in P)$ or S_p $(v_i\in S_q)$, but only edge $e_{k,i}$ is inserted into appropriate set.

Rule h: Edge $e_{k,j}$ is checked for intersection with edges from S_q $(v_i\in P)$ or S_p $(v_i\in S_q)$, but it is not added into any of the set. Edge $e_{k,i}$ is removed from the set S_p $(v_i\in P)$ or S_q $(v_i\in S_q)$.

Rule i: Both edges $e_{k,i}$ and $e_{k,j}$ are checked for intersections, but they are not inserted into any of the sets of considered edges.

2.2 Adding navigation information to intersection points

In the second step of the algorithm, each intersection point is equipped with additional, so-called navigation information. To show how the algorithm handles polygons with holes, our polygons P and Q are extended with some of them. Polygon P gets ring R_{P1} and nested ring R_{P2}, while polygon Q is equipped with one ring R_Q (see Figure 6). For this step, edges of input polygons have to be oriented properly. The algorithm requires all loops (and rings on even levels of depth) to be counter-clockwise and all rings on odd levels of depth to be clockwise oriented (see Figure 6). Beside intersection coordinates, each intersection point is in this step equipped with the following data:

- **a consecutive position (CP_P, CP_Q).** An intersection point laying between polygon vertices v_i and v_j ($i < j$) obtains a floating point pair of indexes regarding a normalised distance from vertex v_i taking into account the orientation of edges. This position is needed to determine quickly which vertices lay between two neighbouring intersection points in the direction of orientation. Table 1 shows the calculated consecutive positions for our example.

Table 1. Consecutive positions

Intersection	CP_P	CP_Q	Intersection	CP_P	CP_Q
a	4.13	18.60	i	12.45	22.21
b	3.85	18.51	j	12.30	14.58
c	2.66	18.21	k	8.83	21.25
d	7.32	22.64	l	1.75	17.16
e	7.21	14.31	m	9.50	14.69
f	2.32	17.74	n	1.66	16.62
g	13.81	14.39	o	1.43	15.36
h	13.63	22.55	p	1.31	14.92

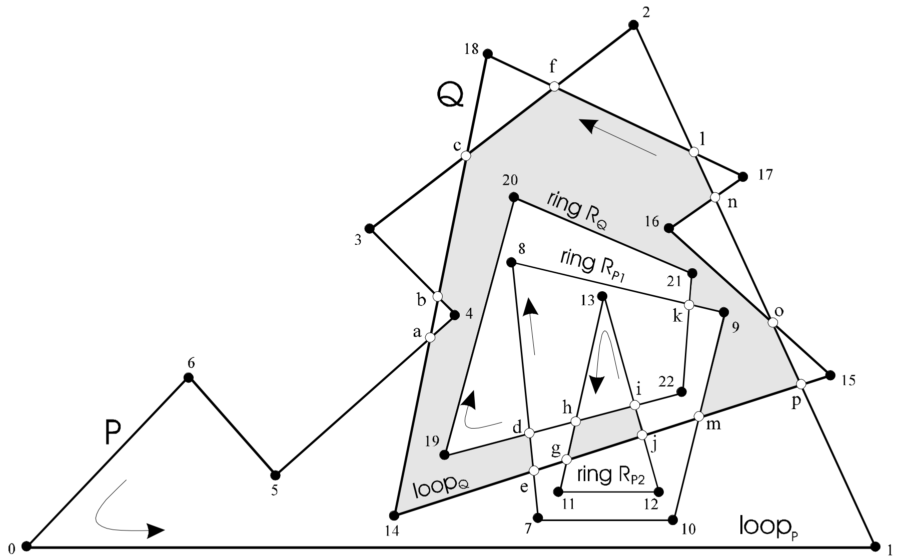

Figure 6. Intersection between two polygons with rings

- **the next polygon flag**. This value represents the polygon on which the walk-about from a considered intersection point will continue. Let us say there are two vectors **a** and **b**; **a** belongs to polygon P and **b** to polygon Q. The function *left_of(**a**, **b**)* returns one of the vectors **a** and **b** which points to the left half-plane determined by the other vector. In Figure 7b vector **b** is *left of* vector **a**. Therefore, in the case of Figure 7, we shall continue from intersection point a in the direction pointed by vector **b,** i.e. we shall follow the loop of polygon Q. The direction of walk-about always corresponds to the orientation of loop or ring.

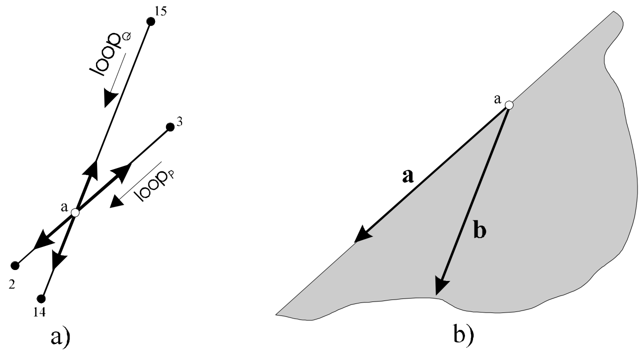

Figure 7a) Possible direction of movement from an intersection point
b) Rule *left_of(**a**, **b**)*

So far only intersection points have been discussed. However, it could happen that intersection points do not exist. Such cases are easy solved because there are only two possibilities:

- the polygons do not intersect at all, or
- one polygon is completely inside the other one.

In the first case, the resulting intersection polygon does not exist. In the second case, a classical containment test is used to determine which polygon is inside and which outside.

2.3 Finding resulting intersection polygon

In this part of the algorithm, the walk-about through intersection points is finally performed. With a new efficient data structure and with prepared navigation information, the walk through the “labyrinth” of edges and intersection points can be done very efficiently. Let us take a closer look at the idea. The list of found intersection points with determined navigation information is split into three arrays, and two of them are sorted regarding the values of consecutive positions (Figure 8).

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Sorted CPs of polygon P	p	o	n	l	f	c	b	a	-1	e	d	k	m	-1	j	i	h	g	-1
Consecutive position P	1.31																		

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Sorted CPs of polygon Q	e	g	j	m	p	o	n	l	f	c	b	a	-1	k	i	h	d	-1
Consecutive position Q	14.31																	

Keys		a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p
	Index P	7	6	5	10	9	4	17	16	15	14	11	3	12	2	1	0
	Index Q	11	10	9	16	0	8	1	15	14	2	13	7	3	6	5	4

Figure 8. Arranging of data representing consecutive positions of intersection points

At the beginning, the first point is taken from the array *Keys* and remembered. We have to return to this point to terminate our walk-about. From the start point, the algorithm proceeds on the polygon determined by the navigation information until the next intersection point is met. This point is simply the right point in the sorted array of consecutive point positions. All polygon vertices between two consecutive position including intersection points have to be included into result, too. Let us take a few lines to explain how easy the algorithm determines which polygon vertices have to be added between two intersection points x and y. Consider that we are moving from the intersection point x to the y. There are two possibilities:

a) y is immediate right neighbour of x. In this case $CP(x) < CP(y)$ and all polygon vertices with indexes between $\lceil x \rceil$ and $\lfloor y \rfloor$ are added to the result.
b) the right immediate neighbour of x is pointer to the y (see Figure 8). This means that $CP(x) > CP(y)$. The polygon vertices between x and y are obtained in two steps:
    - vertices with indexes i: $\lceil x \rceil \leq i \leq$ the last vertex index in the loop (or appropriate ring)
    - vertices with indexes j: the last vertex index in the loop(or appropriate ring) vertex index < j < y.

The name of the next consecutive point serves as the key into the array *Keys* where the position of this point in the sorted array of the second polygon is stored. In this way, we "jump" from one array of sorted consecutive position to another without any search. Each visited intersection point is marked as being already used. The process continues until the starting point, which has been remembered at the beginning, is reached what means that the resulting polygon is closed. In the case of convex input polygons without holes, this is already the solution, but in general, the result can consist of more non-connected closed polygons. Because of this, the described algorithm continues until all the intersection points in array *Keys* are marked as used.

To highlight the described algorithm, let us do the walk-about in our example (see Figure 6 and Table 1). The first intersection point is point a. We already know from the navigation data that we shall continue the walk-about on polygon Q. Intersection a is located at the position 11 in the array of sorted consecutive position of polygon Q (see Figure 8). The right neighbour in that array is not an intersection point. Instead, it is a pointer to the next intersection point on the loop of polygon Q in the direction of orientation. In this way, the intersection point e is obtained. By already described procedure, vertex v_{14} is found as being located between intersection points a and e. It is obtained:

$P \cap Q = \{a, v_{14}, e\}$.

At intersection point e, we switch to polygon P. From the array *Keys* we discover that e is stored at position 9 in sorted array of consecutive points of polygon P. Its right neighbour is vertex d which is added to the result:

$P \cap Q = \{a, v_{14}, e, d\}$.

Now we turn to polygon Q again. The position of d is 16 in the sorted array of consecution points of polygon Q. In this way, we continue until the resulting intersection polygon is completely constructed.

3 TIME COMPLEXITY ESTIMATION

The worst case time complexity of the polygon intersection problem is $\Omega(n \times m)$ where n is the number of vertices of polygon P and m the number of vertices of polygon Q. This time complexity refers to the first part of the algorithm where the intersection points have to be determined. In some cases, there exist exactly $n \times m$ intersection points (see Figure 2) and in such cases, this time complexity cannot be avoided. Fortunately, in the most cases, less intersection points can be expected. It has been shown that the sweep-line solution with AVL trees performs this task in $O((n + r) \log n)$, where n is the common number of both polygon edges and r the number of intersections [PREP85, BERG97]. Theoretical estimation of our solution still have the quadratic time complexity but in general it works much better than $\Omega(n \times m)$. Let us consider which time complexity have individual parts of the algorithm.

In the first step, all $k = m + n$ vertices of input polygons have to be sorted in x direction. Using quicksort this require $O(k \log_2 k)$ time. Then, for each sweep line stop, one of the following actions is performed (see section 2.1):

- Applying **Rule a**: two edges of polygon P, $v_i \in P$ (or of polygon Q, $v_i \in Q$) are checked for intersections with edges of polygon Q (or P) which are currently located in corresponding sets S_q (or S_p). Suppose, there is r ($r < k$) edges in the considered set. This task is always finished in time 2 O(r).
- Applying **Rule b**: two edges from a given set of edges can be removed in linear time O(r).
- Applying **Rule c** and **Rule d**: in this case, one edge is checked for intersection, and the other one is removed from the corresponding set of edges. Using the estimations mentioned before, this task is finished in time O(r), too.
- Rules which solve special cases (see Figure 5 in Section 2.1) can be considered using the analogy of **Rules a-d.**

These tasks have to be performed at each of the polygon vertex v_i ($i \leq k$). Let us suppose that at each vertex, **Rule a** has to be applied. In this case, we have to do 2kO(r) checks giving us

$$T1 = O(k^2).$$

However, the upper is very rude, because also other rules are used for sure, and in general, the number of edges $r << k$.

The processing time of the second part of the intersection algorithm depends on the number of found intersection points $w \leq k$. This part of the algorithm is passed only once. Therefore,

$$T2 = O(w) = O(k).$$

The third step of the algorithm requires two sortings of $w1 = w + r + 1$ values where w is the number of intersection points, and r is the number of rings of individual polygon (see Figure 8). This can be done in time

$$T3 = 2\, O(w1 \log_2 w1)$$

We can expect $w1 \approx k$ and therefore T3 becomes

$T3 = O(k \log_2 k)$.

The common expected time complexity of the algorithm which finds the intersection of two arbitrary non-self-intersected polygons, is:

$$T + T1 + T2 + T3 = O(k^2) + O(k) + O(k \log_2 k) = O(k^2).$$

Practical measurement shows that theoretical estimation is to strong and that an average time complexity is closer to the $O(k \log_2 k)$.

4 CONCLUSIONS

In the paper, a new universal approach to find the intersection between two arbitrary non-self-intersected polygons has been presented. The algorithm can be implemented easily. Problems of finite arithmetic arise only at actual calculation of two polygon edges, and thus they can be well controlled. In this way, a quick and very stable algorithm can be easily constructed. The algorithm can be extended to the other Boolean operations by minor changes in the second part of the algorithm where the navigation information is determined. At the moment, the algorithm for construction of the Boolean union is in the final stage.

The algorithm has been developed for the use in geodesy as a supporting algorithm for existing GIS systems. In these cases, polygons represent pieces of land. With additional small piece of code, which calculates the area of intersected polygon, the algorithm is used at Faculty of Civil Engineering at University of Maribor, Slovenia, and at department for Department of Soil, Water, and Climate at the University of Minnesota, USA. Until now, over 100000 polygon intersections have been calculated without failures. The algorithm can be checked by visiting http://www.uni-mb.si/~cgm.

Acknowledgements. We greatly acknowledge Dr. Daniel Rebolj (University of Maribor) and Mr. Donald Kackman (University of Minnesota) with supplying us with “strange” polygons and for their patience during the testing phase of the algorithm. The polygon from Figure 1 has been shown by permission of Mr. Donald Kackman. We are also thankful the anonymous referees for suggestions to improvement of the paper.

5 REFERENCES

[BERG97] Berg, M., M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geometry, Algorithms and Applications*, Springer-Verlag, Berlin, 1997.

[MORT85] Mortenson, M. E., *Geometric Modeling*, John Wiley & Sohns, 1985.

[O’ROU93] O’Rourke, J: *Computational Geometry in C*, Cambridge University Press, 1993.

[PREP85] Preparata, F. P. and M. I. Shamos, *Computational Geometry An Introduction*, Springer-Verlag, 1985.

[ŽALI98] Žalik, B., and G. J. Clapwhorthy, *A Universal Trapezoidation Algorithm for Planar Polygons*, sent for review in Comput. & Graphics.