Efficient Clipping of Arbitrary Polygons

Günther Greiner* Kai Hormann†

Computer Graphics Group, University of Erlangen

Abstract

Clipping 2D polygons is one of the basic routines in computer graphics. In rendering complex 3D images it has to be done several thousand times. Efficient algorithms are therefore very important. We present such an efficient algorithm for clipping arbitrary 2D polygons. The algorithm can handle arbitrary closed polygons, specifically where the clip and subject polygons may self-intersect. The algorithm is simple and faster than Vatti's [11] algorithm, which was designed for the general case as well. Simple modifications allow determination of union and set-theoretic difference of two arbitrary polygons.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation—Display Algorithms I.3.5 [Computer Graphics]: Computational Geometry and Object Modelling

Keywords: Clipping, Polygon Comparison

1 Introduction

Clipping 2D polygons is a fundamental operation in image synthesis. For example, it can be used to render 3D images through hidden surface removal [10], or to distribute the objects of a scene to appropriate processors in a multiprocessor ray tracing system. Several very efficient algorithms are available for special cases: Sutherland and Hodgeman's algorithm [4, 10] is limited to convex clip polygons. That of Liang and Barsky [4, 5] require that the clip polygon be rectangular. More general algorithms were presented in [1, 6, 8, 9, 13]. They allow concave polygons with holes, but they do not permit self-intersections, which may occur, e.g., by projecting warped quadrilaterals into the plane.

For the general case of arbitrary polygons (i.e., neither the clip nor the subject polygon is convex, both polygons may have self-intersections), little is known. To our knowledge, only the Weiler algorithm [12, 4] and Vatti's algorithm [11] can handle the general case in reasonable time. Both algorithms are quite complicated.

In this paper we present an algorithm for the clipping of arbitrary polygons, that is conceptually simple, for example, the data structure for the polygons we use is less complex. While in Weiler's algorithm the input polygons are combined into a single graph structure, we represent all polygons (input as well as output) as doubly linked lists. In all three approaches all the intersections between the two input polygons have to be determined first (in Vatti's algorithm, self-intersections of each input polygon as well). Merging these intersection points into the data structure is the decisive step. We think that our approach is more intuitive and considerably simpler than Weiler's algorithm. Finally, we obtain each output polygon by a simple traversal of the (modified) input polygons. In Weiler's algorithm, traversals of the tree are necessary. A runtime comparison with Vatti's algorithm is given in the final section. Weiler's algorithm as well as the one presented here can also determine other Boolean operations of two arbitrary polygons: union and set-theoretic difference.

*greiner@cs.fau.de
†hormann@cs.fau.de

This paper is organized as follows. In the next section we specify what the interior of an arbitrary polygon is. In Section 3 we outline the basic concept of the algorithm. We then describe the data structure used to represent polygons in Section 4. In Section 5 we describe how the intersection points are merged into the data structure and give details of the implementation. In the final section, results are discussed and compared to Vatti's algorithm.

2 Basics

A closed polygon $\boldsymbol{P}$ is described by the ordered set of its vertices $P_0, P_1, P_2, \ldots, P_n = P_0$. It consists of all line segments consecutively connecting the points P_i, i.e. $\overline{P_0P_1}$, $\overline{P_1P_2}, \ldots, \overline{P_{n-1}P_n} = \overline{P_{n-1}P_0}$.

For a convex polygon it is quite simple to specify the interior and the exterior. However, since we allow polygons with self-intersections we must specify more carefully what the interior of such a closed polygon is. We base the definition on the *winding number* [4]. For a closed curve γ and a point A not lying on the curve, the winding number $\omega(\gamma, A)$ tells how often a ray centered at A and moving once along the whole closed curve winds around A, counting counterclockwise windings by $+1$ and clockwise windings by -1 (see Figure 1). The winding number has several important properties:

- When A is moved continuously and/or the curve γ is deformed continuously in such a way that A always keeps a positive distance to γ, the winding number will not change.
- For a fixed curve γ the winding number $\omega(\gamma, \cdot)$ is constant on each component of the complement $\mathbb{R}^2 \setminus \gamma$. Moreover, if A lies in the unbounded component of $\mathbb{R}^2 \setminus \gamma$ then $\omega(\gamma, A) = 0$.
- If A moves and thereby crosses the curve once, the winding number decreases or increases by exactly 1.

The third statement is the basis for the algorithm presented below. It can be derived from the first as is illustrated in Figure 2.

The interior of a closed curve (e.g., a closed polygon) now is defined as follows:

Definition 1 *A point* A *lies in the interior of the closed curve γ if and only if the winding number $\omega(\gamma, A)$ is odd.*

This definition and the third property of the winding number stated above then imply the following: *A path that intersects the polygon exactly once traverses either from the interior to the exterior of the polygon, or vice versa.*

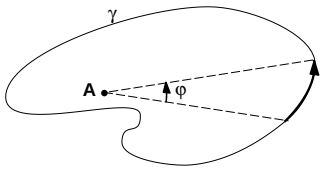

Figure 1: Winding number: $\omega(\gamma, A) = \frac{1}{2\pi} \int d\varphi$.

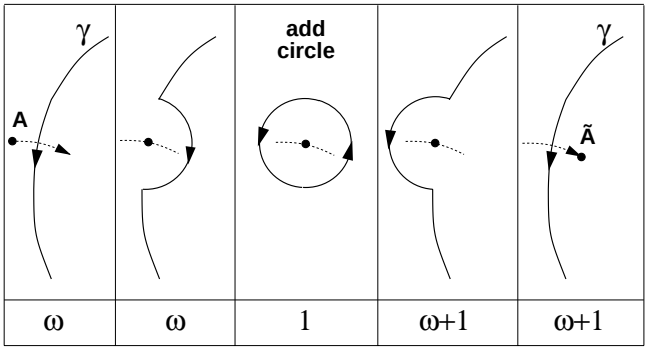

Figure 2: Change of the winding number when a point crosses the curve.

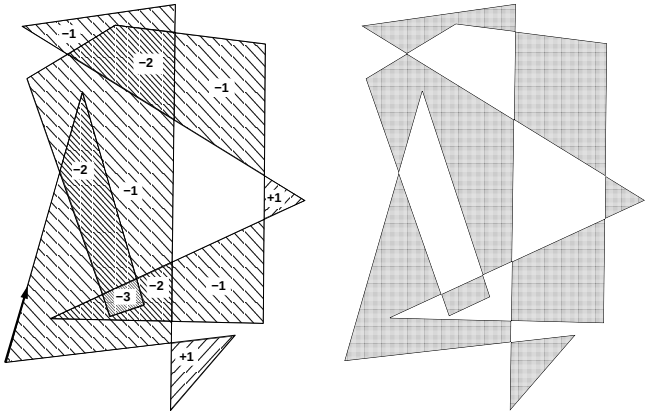

Figure 3: Winding numbers ($\neq 0$) and interior for an arbitrarily complex polygon.

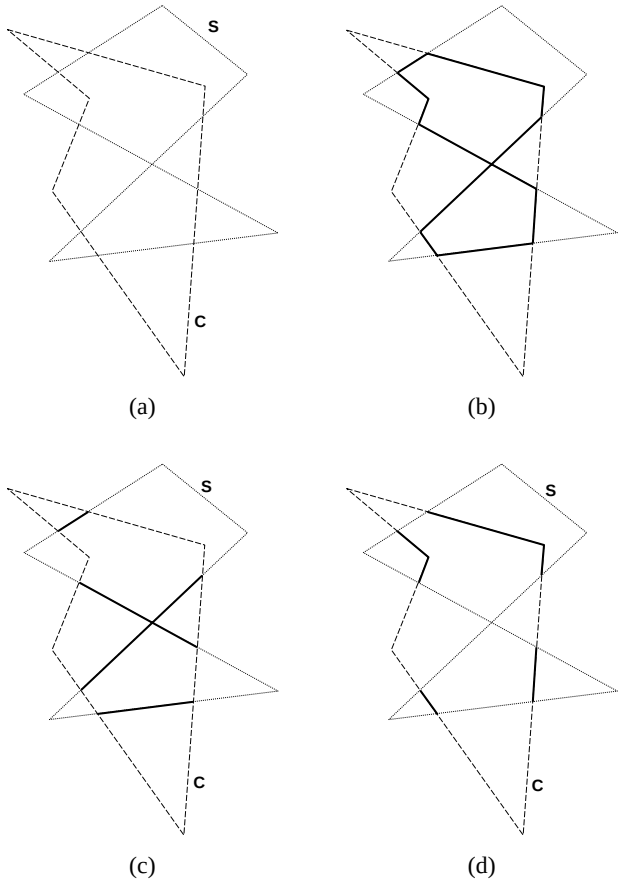

Figure 4: Example of clipping a subject polygon S against a clip polygon C. The lower row shows parts of subject polygon inside clip polygon ($S \cap C_{\text{int}}$) and parts of clip polygon inside the subject polygon ($C \cap S_{\text{int}}$), respectively.

This property leads to an efficient algorithm to detect whether a point lies inside or outside a polygon, namely the *even-odd rule* (see [4]). In Figure 3 the winding numbers and the interior of a closed polygon are shown.

Given two polygons, a clip (clipper) and a subject polygon (clippee), the *clipped polygon* consists of all points interior to the clip polygon that lie inside the subject polygon. This set will be a polygon or a set of polygons. Thus, clipping a polygon against another polygon means determining the intersection of two polygons. In general, this intersection consists of several closed polygons. Instead of intersection, one can perform other Boolean operations (to the interior): e.g., *union* and *set-theoretic difference*. (see Figure 5).

3 General Concept

The process of clipping an arbitrary polygon against another arbitrary polygon can be reduced to finding those portions of the boundary of each polygon that lie inside the other polygon. These partial boundaries can then be connected to form the final clipped polygon.

To clarify this, consider the example in Figure 4 where the task is to clip the polygon with the dotted lines (referred to as the subject polygon S) against the polygon with the broken lines (referred to as the clip polygon C). We start by determining which parts of the subject polygon boundary lie inside the clip polygon (Figure 4.c). We can find those parts by considering the following analogous situation:

Imagine pushing a chalk cart along the subject polygon boundary. We start at some vertex of the polygon, and open the distribution hatch at the start if the vertex lies inside the clip polygon. Then we push the cart along the subject polygon toggling the position of the hatch (open/closed) whenever we cross an edge of the clip polygon. We stop when we reach our starting vertex. Then the parts of the subject polygon that lie inside the clip polygon will be marked with chalk.

We use the same technique, but this time running our chalk cart along the clip polygon in order to discover those parts of the clip polygon that lie inside the subject polygon (Figure 4.d): Once we have found all those parts of the polygon edges that lie inside the other polygon, we merge these parts to obtain the clipped polygon (Figure 4.b).

The process of merging is easy, considering the fact that each part of the subject polygon that will be in the outcome is bounded by two intersection points of subject and clip polygon. These vertices are also the beginning or end of one of the clip polygon's relevant parts. Therefore, if you keep track of the intersection points and the parts they come from, connecting the supporting parts in the correct order is easy and shown in Section 5.

Set-theoretic difference and the union of the two polygons can also be calculated by making the following modification to the algorithm. To determine $S \setminus C$, one first marks the parts of the subject polygon that are *exterior* to the clip polygon. These will be merged with the relevant parts of the clip polygon. The procedure is illustrated in the left part of Figure 5. Determination of the union is sketched in the middle of Figure 5 and the right part shows how the difference $C \setminus S$ can be obtained.

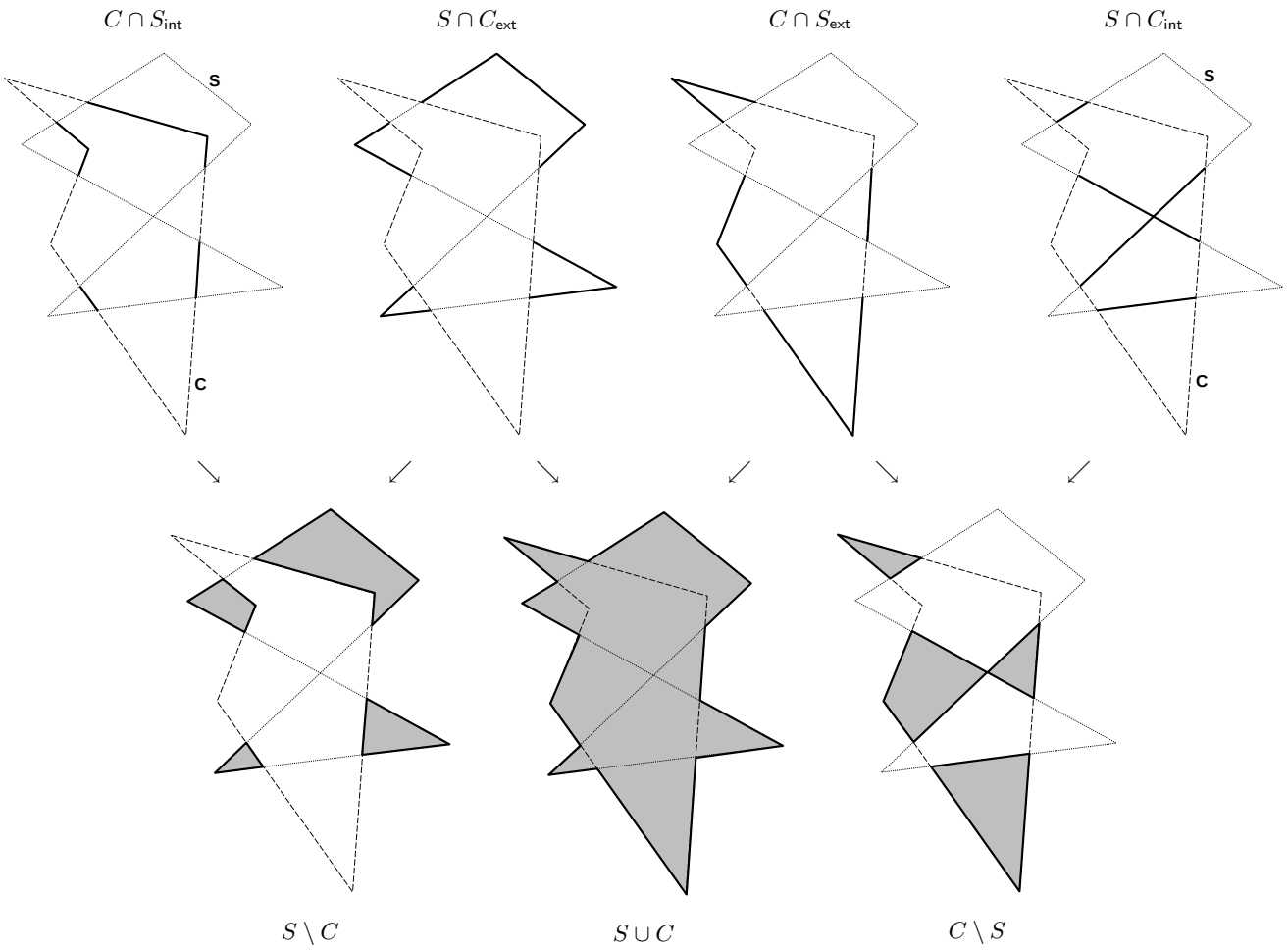

Figure 5: Set theoretic differences and union of two polygons

4 Data structures

Our algorithm requires a data structure for representing polygons. As shown later, a doubly linked list of nodes is most suitable. Each node represents one of the polygon's vertices and contains the following information:

```
vertex = { x, y       : coordinates;
           next, prev : vertexPtr;
           nextPoly   : vertexPtr;
           intersect  : boolean;
           neighbor   : vertexPtr;
           alpha      : float;
           entry_exit : boolean;
         }
```

Figure 6: Vertex data structure.

Normally a vertex only needs `x` and `y` to store its coordinates and `next` and `prev` as links to the neighboring vertices. As the clipping process may result in a set of n polygons $(\boldsymbol{P}_1, \ldots, \boldsymbol{P}_n)$, we use `nextPoly` to be able to handle a linked list of polygons, i.e., we let the `nextPoly` pointer of the first vertex of the k-th polygon ($\boldsymbol{P}_{k,0} \rightarrow$ `nextPoly`) point at the first vertex of the $(k+1)$th polygon $\boldsymbol{P}_{k+1,0}$, $k = 1, \ldots, n-1$.

The remaining fields (`intersect, neighbor, alpha, entry_exit`) are used internally by the algorithm. Intersection points of subject and clip polygon are marked by the `intersect` flag. During execution of the algorithm, all intersection points will be determined and two copies, linked by the `neighbor` pointer, will be inserted into the data structures of both the subject and the clip polygon. That means, the intersection point which is inserted into the subject polygon's data structure will be connected to the one inserted into the clip polygon's data structure using the `neighbor` pointer and vice versa. To accelerate the sorting process, we store an `alpha`-value indicating where the intersection point lies relatively to start and end point of the edge. Remembering the chalk cart analogy we also need an `entry_exit` flag to record whether the intersecting point is an entry or an exit point to the other polygon's interior.

Figure 7 shows an example clipping problem and the data structure generated by the algorithm.

5 The algorithm

The algorithm operates in three phases:

In phase one (see Figure 8), we search for all intersection points by testing whether each edge of the subject polygon and each of the clip polygon intersect or not. If they do, the intersection routine (Figure 11) will deliver two numbers between 0 and 1, the `alpha`-values, which indicate where the intersection point lies relatively to

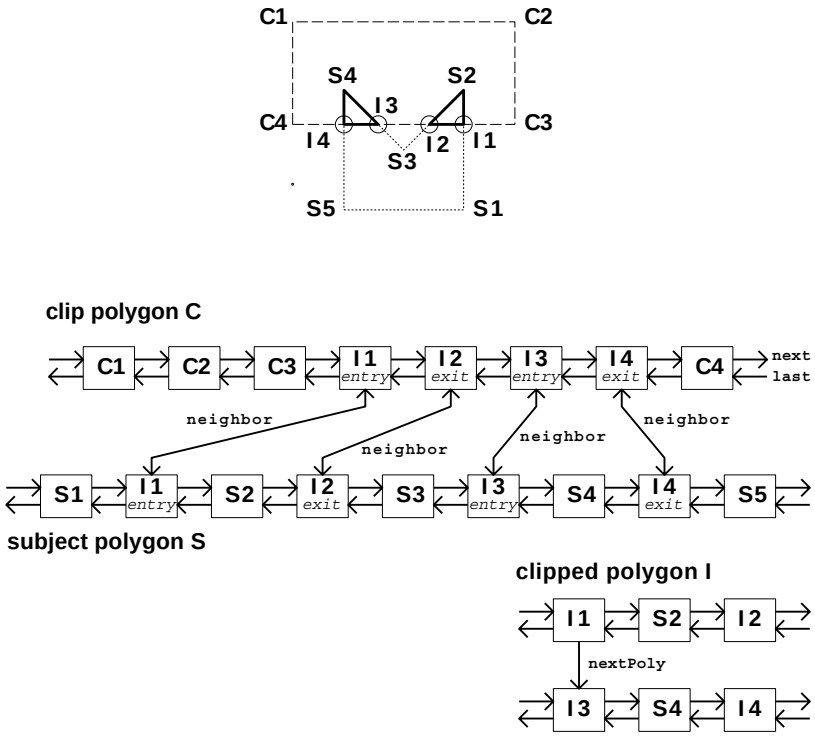

Figure 7: Data structure for polygons.

```
for each vertex Si of subject polygon do
  for each vertex Cj of clip polygon do
    if intersect (Si,Si+1,Cj,Cj+1,a,b)
      I1 = CreateVertex(Si,Si+1,a)
      I2 = CreateVertex(Cj,Cj+1,b)
      link intersection points I1 and I2
      sort I1 into subject polygon
      sort I2 into clip polygon
    end if
  end for
end for
```

Figure 8: Pseudo-code for phase one.

start and end point of both edges. With respect to these `alpha`-values, we create new vertices and insert them into the data structures of subject and clip polygon between the start and end point of the edges that intersect. If no intersection points are detected we know that either the subject polygon lies entirely inside the clip polygon or vice versa or that both polygons are disjoint. By performing the even-odd rule we can easily decide which case we have and simply return either the inner polygon as the clipped polygon or nothing at all.

Phase two (see Figure 9) is analogous to the chalk cart in Section 3. We trace each polygon once and mark entry and exit points to the other polygon's interior. We start at the polygon's first vertex and detect using the even-odd rule whether this point lies inside the other polygon or not. Then we move along the polygon vertices and mark the intersecting points that have been inserted in phase one (and marked by the `intersect` flag) alternately as entry and exit points respectively.

```
for both polygons P do
  if P0 inside other polygon
    status = exit
  else
    status = entry
  end if
  for each vertex Pi of polygon do
    if Pi->intersect then
      Pi->entry_exit = status
      toggle status
    end if
  end for
end for
```

Figure 9: Pseudo Code for phase two.

In phase three (see Figure 10) we create the desired clipped polygon by filtering it out of the enhanced data structures of subject and clip polygon. In order to build the clipped polygon we use two routines: `newPolygon` and `newVertex`. `newPolygon` registers the beginning of a new polygon while the vertices of that polygon are transferred by `newVertex`; for example, the sequence

```
newPolygon
newVertex(A)
newVertex(B)
newVertex(C)
newPolygon
newVertex(D)
newVertex(E)
newVertex(F)
newVertex(G)
```

```
while unprocessed intersecting points in
                                subject polygon
  current = first unprocessed intersecting
                    point of subject polygon
  newPolygon
  newVertex (current)
  repeat
    if current->entry
      repeat
        current = current->next
        newVertex (current)
      until current->intersect
    else
      repeat
        current = current->prev
        newVertex (current)
      until current->intersect
    end if
    current = current->neighbor
  until PolygonClosed
end while
```

Figure 10: Pseudo-code for part three.

```
intersect(P1,P2,Q1,Q2,alphaP,alphaQ)
  WEC_P1 = <P1 - Q1 | (Q2 - Q1)^⊥>
  WEC_P2 = <P2 - Q1 | (Q2 - Q1)^⊥>
  if (WEC_P1*WEC_P2 <= 0)
    WEC_Q1 = <Q1 - P1 | (P2 - P1)^⊥>
    WEC_Q2 = <Q2 - P1 | (P2 - P1)^⊥>
    if (WEC_Q1*WEC_Q2 <= 0)
      alphaP = WEC_P1/(WEC_P1 - WEC_P2)
      alphaQ = WEC_Q1/(WEC_Q1 - WEC_Q2)
      return (1); exit
    end if
  end if
  return (0)
end intersect
```

Figure 11: Pseudo-code for the intersection.

generates a set of two Polygons $P_1 = ABC$ and $P_2 = DEFG$ and $A \rightarrow$ `nextPoly` points at D.

To illustrate the pseudo-code of phase three (Figure 10) we use our chalk cart again, here called 'current'. First we place it at one of the intersection points. Since we want to mark the clipped polygon we open the hatch (`newPolygon`) and move the cart along the subject polygon's edge into the interior of the clip polygon. The `entry_exit` flag tells us which direction to choose: 'entry' means forward direction (`next`) while 'exit' signals us to go backward (`prev`). Each time we reach a vertex we remember it by calling `newVertex`. We leave the clip polygon's interior as soon as we come to the next intersection point. This is where we turn the cart (`current = current->neighbor`) in order to move along the clip polygon's edges. Again the `entry_exit` flag tells us which route leads to the other polygon's interior. We continue this process until we arrive at the starting vertex and close the hatch (and the polygon). If there are still intersection points that have not yet been chalked (i.e., the clipped polygon is a set of polygons) we move the chalk cart there and repeat the whole procedure until there are no unmarked intersection points left.

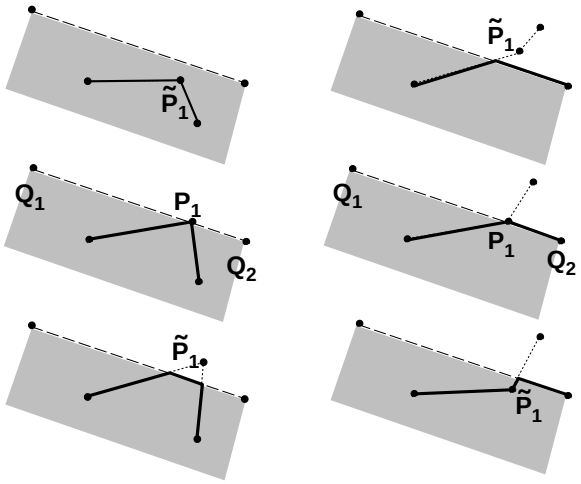

Figure 12: Two degenerate configurations (left and right) and two possible perturbations for each example (upper and lower row).

Clearly finding the intersection of two lines, say $\overline{P_1P_2}$ and $\overline{Q_1Q_2}$, is a basic operation of the algorithm. This can be done effectively in the following way. Determine *window edge coordinates*, *outcodes* and α-values (see [3, 4]) of P_i with respect to $\overline{Q_1Q_2}$ and, if necessary, also for Q_i with respect to $\overline{P_1P_2}$. By this algorithm, many cases where there is no intersection will be detected early. When there is an intersection, the procedure `intersect` (Figure 11) will return the α-values `alphaP` and `alphaQ` for the point of intersection with respect to $\overline{P_1P_2}$ and $\overline{Q_1Q_2}$ respectively.

So far, we tacitly assumed that there are no *degeneracies*, i.e., each vertex of one polygon does not lie on an edge of the other polygon (see also [2]). Degeneracies can be detected in the `intersect` procedure. For example, P_1 lies on the line $\overline{Q_1Q_2}$ if and only if `alphaP` $= 0$ and $0 \leq$ `alphaQ` ≤ 1. In this case, we perturb P_1 slightly such that for the perturbed point $\tilde{P}_1$ we have `alphaP` $\neq 0$. We allow the algorithm to continue, replacing P_1 with $\tilde{P}_1$. Two typical examples are given in Figure 12. For each case two possible perturbations are sketched. If we take care that the perturbation is less than pixel width, the output on the screen will be correct.

6 Evaluation

Both Vatti's algorithm and the one presented here have been implemented in C on a Silicon Graphics Indigo work station. Given an integer n, a subject and a clip polygon with n vertices were generated at random and clipped against one other, first using Vatti's algorithm and then the on described above. This was done a thousand times, and the running times of both algorithms were recorded. The resulting average times (in *ms*) are listed in columns two and three of Table 1. The improvement factors of our method over Vatti's algorithm are shown in the next column. The table demonstrates that the improvement factor increases with the size of n. An explanation for that will be given below.

As explained above, the intersection points of subject and clip polygon are part of the clipped polygon, hence there is no way to avoid calculating them. In order to illustrate the typical number of intersections, we recorded them for each trial. The averages of these numbers are listed in the last column of Table 1. Inspecting these values, we observe that they grow with n^2. Figure 13 shows that if we have a polygon with n edges and another with m edges, the number of intersections can be nm in the worst case. So the aver-

n	Vatti	New Algorithm	Improvement	Intersections
3	0.272	0.1754	1.55	1.98
5	0.644	0.3657	1.76	5.88
10	2.093	1.163	1.80	23.40
20	8.309	4.218	1.97	91.31
50	66.364	30.724	2.16	583.40

Table 1: Results of the implementation.

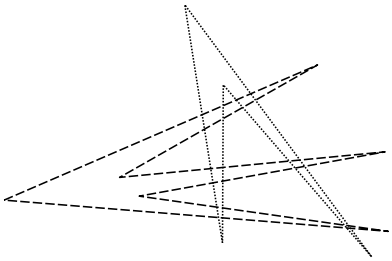

Figure 13: Worst case for intersection of two polygons.

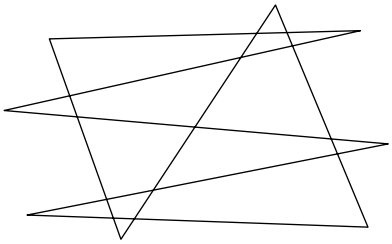

Figure 14: Worst case for self-intersection.

age number of intersections grows by the order of $O(nm)$. There is a well-known result in computational geometry based on the plane sweep algorithm, which says that if there are N line segments generating k intersections, then these intersections can be reported in time $O((N+k)\log(N))$ [7]. Note that this relation yields an even worse complexity in the worst case. Since the computation of the intersections involves floating point operations, it is a complex task compared to the remaining work that has to be done by the algorithm (sorting, pointer assignments, etc.). Measurements revealed that intersection calculation accounts for roughly 80% of the running time.

Consequently, any clipping algorithm supporting arbitrary polygons must have complexity $O(nm)$ with n and m being the edge numbers of the polygons. This statement is confirmed by the average timings of both algorithms.

The reason for the poorer performance of Vatti's algorithm is that it also has to compute the self-intersection points of both polygons. Figure 14 indicates that the number of self-intersection points for a polygon with n edges can be $O(n^2)$ in the worst case. This might be the reason why the improvement factor of our algorithm (compared to Vatti's algorithm) grows with increasing number of edges.

Acknowledgments

The authors wish to thank the anonymous referees for their critical comments which helped to improve this paper.

References

[1] Andreev, R. D. Algorithm for clipping arbitrary polygons. *Comput. Graph. Forum 8* (1989), 183–191

[2] Blinn, J. Fractional Invisibility. *IEEE Comput. Graph. Appl. 8* (1988), 77–84

[3] Blinn, J. Line Clipping. *IEEE Comput. Graph. Appl. 11* (1991), 98–105

[4] Foley, J. D., A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics, Principles and Practice.* Addison-Wesley, Reading, MA, 1990

[5] Liang, Y. and B. A. Barsky. An analysis and algorithm for polygon clipping. *Commun. ACM 26* (1983), 868–877

[6] Montani, C. and M. Re. Vector and raster hidden surface removal using parallel connected stripes. *IEEE Comput. Graph. Appl. 7* (1987), 14–23

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

[8] Rappaport, A. An efficient algorithm for line and polygon clipping. *Visual Comput. 7* (1991), 19–28

[9] Sechrest, S. and D. Greenberg. A visible polygon reconstruction algorithm. *Comput. Graph. 15* (1981), 17–26

[10] Sutherland E. E. and G. W. Hodgeman. Reentrant polygon clipping. *Commun. ACM 17* (1974), 32–42

[11] Vatti, B. R. A generic solution to polygon clipping. *Commun. ACM 35* (1992), 56–63

[12] Weiler, K. Polygon comparison using a graph representation. In *SIGGRAPH '80* (1980), 10–18

[13] Weiler, K. and P. Atherton. Hidden surface removal using polygon area sorting. In *SIGGRAPH '77* (1977), 214–222