'91 KACC 1991. 10. 22~24

Robust Plane Sweep Algorithm for Planar Curve Segments

In-Kwon Lee, Hwan-Yong Lee, and *Myung-Soo Kim*

Department of Computer Science, POSTECH

Abstract

Plane sweep is a general method in computational geometry. There are many efficient theoretical algorithms designed using plane sweep technique. However, their practical implementations are still suffering from the topological inconsistencies resulting from the numerical errors in geometric computations with finite-precision arithmetic. In this paper, we suggest new implementation techniques for the plane sweep algorithms to resolve the topological inconsistencies and construct the planar object boundaries from given input curve segments.

1 Introduction

Plane sweep is a general method in computational geometry. A variant of this method known as scan-line algorithm in computer graphics has been extensively used in the fundamental computer graphics algorithms for hidden-line elimination and polygon-filling. The basic idea is sweeping a vertical (resp. horizontal) line in the plane from left to right (resp. from bottom to top) and examining the status change of geometric entities lying on the current sweep line. As the sweep line moves, the corresponding cross-sections change their positions and shapes dynamically. At certain discrete instances, the topological arrangement of these cross sections changes. For example, in Figure 1, the relative position for the intersection points of C and D on the current sweep line L switches as L passes across the curve intersection point p. In-between these events the relative positions are the same. By scheduling these events for status changes correctly in advance and further taking appropriate actions to record the corresponding status change at each scheduled event, one can determine the correct topological arrangement of geometric entities in the plane.

There are potentially many geometric reasoning problems in a plane which can be solved using this general method. These include line and curve segment intersection, hidden line elimination, polygon filling, boolean operations (union, intersection, difference) for 2D regions, etc. The plane sweep algorithm solves these problems essentially by computing the line and curve contour intersections and additionally doing problem-specific geometric reasonings about the regions bounded by these contours. In this paper, we consider the problem of constructing object boundaries using plane sweep, from a set of planar curve segments representing geometric constraints.

Many authors in computational geometry have designed efficient intersection algorithms using plane sweep technique. For example, Bently and Ottmann [1] designed an $O((n+k)\log n)$ plane sweep algorithm to compute the line segment intersections in a plane, where k is the number of intersections. Chazelle and Edelsbrunner [2] recently improved this result to an optimal $O(n\log n + k)$ algorithm. For the curved case, Johnstone and Goodrich [4] presented an algorithm to compute the intersections between two plane algebraic curve segments using plane sweep technique. This algorithm can also compute the intersections among any number of plane algebraic curve segments. These algorithms assume the intersection points can be computed exactly, i.e., the topological arrangement of arbitrarily close intersection points can be determined correctly. For intersecting line segments with rational coefficient equations, this requirement is relatively easy to satisfy since one can compute each intersection point exactly as a pair of rational numbers by doing only a few simple rational arithmetic with no errors. For intersecting plane algebraic curve segments, Johnstone and Goodrich [4] guarantee the exactness of curve intersections by using arbitrarily small tracing steps determined by the Gap theorem. However, in some degenerate cases, it may require very high precision arithmetic to satisfy this exactness condition.

To implement practically efficient plane sweep algorithm for planar algebraic curve segments, we have to devise a new mechanism which can enforce robustness to the algorithm using finite precision arithmetic for geometric computations. For this purpose, we present two techniques called *multiple intersection merging* and *intersection splitting* which can guarantee certain robustness conditions during the execution of algorithm. Our approach is somewhat similar to the data normalization technique of Milenkovic [7] which can guarantee the robustness of determining the topological arrangement of line segments in a plane using finite-precision arithmetic. However, there are many differences between our approach and Milenkovic's since algebraic curve segments have more general properties than line segments.

The rest of this paper is organized as follows. In §2, we describe the data structures for implementing the plane sweep algorithm to be discussed in this paper. In §3, we present a simple plane sweep (*SPS*) algorithm under the basic assumption that there are no multiple intersections or no (almost) overlapping edges among the given input curve segments. In §4, the method of uncertainty modeling is described. In §5, a robust plane sweep algorithm (*RPS*) which can resolve the topological inconsistencies from numerical errors in geometric computations are described. Finally, in §6 we conclude this paper.

2 Data Structures

Each object has a boundary representation with edges of algebraic curve segments. There is one outer loop of directed edges in counter-clockwise order and k inner loops of edges in clockwise order, where k is the number of holes in the object. Each edge has its begin and end points, and its defining implicit and/or parametric equation(s). Each point is given as a pair of x and y coordinates. The input is given as a set of directed edges. The output is an object with an outer loop and k inner loops of directed edges. The problem is how to construct a planar graph represent the correct topological arrangement of input edges, see Figure 2. By discarding certain redundant edges (i.e., those in the object interior), we construct $k+1$ loops for the object boundary.

First, by adding singular, inflection, and x and y-extreme points as extra vertices if necessary, each edge is subdivided into monotone curve segments, see Figure 2-(b). There are various geometric entities for the internal representation of an object.

- **Point**

There are two types of points, i.e., "begin" or "end" point. The data structure of each point has various fields such as its "begin" or "end" classification, the x, y-coordinate pair, and the pointer to edge.

- **Edge**

Each edge has its class as "line", "circular arc" or general "curve" edge. Each edge has one pointer to its begin point and another pointer to its end point. The normals of a monotone edge are uniquely determined by the two extreme normals at its begin and end points. We can determine the unique quadrant direction (NE, NW, SW, or SE) of each edge from the range of these normals. The other fields are the center and radius for circular arc, the defining implicit and/or parametric equations for general curves, and convexity.

- **Vertex**

Vertex is a data structure for the adjacency information of the two edges sharing a common vertex. Each vertex with two adjacent edges is duplicated into two points, i.e., one as a begin point and the other as an end point. Vertex structure has its (x, y) coordinate pair and two pointers to the adjacent edges, begin-edge is the edge beginning and end-edge is the edge ending at the vertex. Depending on which quadrants the normals of the two adjacent edges direct, each vertex is classified as *ISO-MIN* (isolated-minimum), *ISO-MAX* (isolated-maximum), *HOLE-MIN* (hole-minimum), *HOLE-MAX* (hole-maximum), *OBJ-MIN* (object-minimum), *OBJ-MAX* (object-maximum), *LEFT-SIDE* (left-side), or *RIGHT-SIDE* (right-

side), see Figure 3. We call *ISO-* classes as *isolated-type* and others as *connected-type*.

- **Side**

Side is a data structure for a connected sequence of edges which form a partial boundary of the object. There are two types of sides, "left" or "right" side. Sides are the contents to be kept on a heap in the same way as the conventional plane sweep algorithms keep the edges intersecting with the current sweep line in a heap.

The conventional plane sweep algorithms use two fundamental data structures, i.e., *event queue* and *sweeping status*. We use two major data structures, a list of vertices (*vertex-queue*) as *event queue* and a list of sides (*side-heap*) as *sweeping status*.

3 Plane Sweep Algorithm

In this section, we describe a simple plane sweep (*SPS*) algorithm assuming there are no numerical errors in the geometric computations with used in the algorithm. The *SPS* algorithm is described by sweeping a horizontal line from bottom to top. We initialize *vertex-queue* by sorting all the input points into a sorted list of vertices. Two coincident points are combined into a single vertex, and at the same time the vertex classification is done. The first version of *SPS* algorithm is as follows.

```
Algorithm PlaneSweep
(* first version of SPS algorithm *)
begin
Sort the vertices into vertex-queue and classify the vertices;
side-heap = empty;
while vertex-queue is not empty do begin
   Take the bottom vertex in vertex-queue;
   Do local processing according to the vertex class;
   Delete the bottom vertex of vertex-queue; end
end
```

3.1 Local Processing

Local processings on event vertices are divided into two group. The types of *ISO-MIN*,*OBJ-MIN*, and *HOLE-MIN* form one group. For these types, we construct new side(s) originating from the current event vertex and insert them into a right position in *side-heap*. Each edge adjacent to the vertex constructs a side of type *left* or *right*. If the edge direction is from top to bottom (resp. bottom to top), the side type is *left* (resp. *right*), see Figure 4.

Before inserting a new side to *side-heap*, we check whether this side intersects with an adjacent side in *side-heap*. The details of this intersection processing are explained in the §3.2. The pseudo code for local processings on *-MIN* vertices is given below.

```
procedure LocalMin(v)
(* Local processing on a -MIN vertex v*)
begin
new1 = MakeNewSide(v.begin-edge,v);
new2 = MakeNewSide(v.end-edge,v);
if new1 is empty then new1 = new2;
if new2 is to the left of new1 then swap new1 and new2;
previous = the side previous to the location for new1 in side-heap;
next = the side next to the location for new2 in side-heap;
IntersectSides(new1, previous);
if new2 is empty
   then IntersectSides(new1, next);
   else IntersectSides(new2, next);
Insert new1 and new2 between previous and next;
end
```

The procedure "MakeNewSide" returns a new side containing only a single edge. The edge direction determines the side type. In the cases of *OBJ-MIN* and *HOLE-MIN*, the vertex has two pointers to the two adjacent sides. `Left-side` (resp. `right-side`) field points the side which is to the left (resp. to the right) of the vertex, see Figure 4.

```
procedure MakeNewSide(e,v)
(* Make a new side with edge e and bottom vertex v *)
begin
Allocate memory for new-side;
if the direction of e is from top to bottom
   then new-side.type = left;
   else new-side.type = right;
new-side.edges = e;
new-side.bottom = v;
if v.type = OBJ-MIN then begin
   if new-side.type = left then v.left-side = new-side;
   else v.right-side = new-side; end
else if v.type = HOLE-MIN then begin
   if new-side.type = left then v.right-side = new-side
   else v.left-side = new-side; end
Set new-side.top to the top point of e;
end
```

Local processing on the other type event vertices does not create new sides. At first, we search *side-heap* for the side(s) having the same top point(s) with the current event vertex. We call this side as *current-side*. If the vertex type is *LEFT-SIDE* or *RIGHT-SIDE* (Figure 5-(a)), the new edge must be appended to *current-side*. If it is *ISO-MAX* (Figure 5-(b)), *current-side* must be deleted from *side-heap* since this side does not appears in the final result. In *OBJ-MAX* and *HOLE-MAX* cases, *current-side* and the next side (denoted *next-side*) must have the same top points, and they must be connected at the top. If the two sides have the same bottom vertices (Figure 5-(c)), these sides construct either an inner loop or an outer loop. When one of the two sides has *ISO-MIN* bottom, both sides are redundant and we delete them, see Figure 5-(d). Otherwise, *current-side* and *next-side* have different bottom vertices, and both bottom vertices have two adjacent sides. Thus, there are at least four sides connected in a sequence. We may treat any three connected sides as a single side, see the cases (a), (b), and (c) in Figure 6. As before, after inserting a new edge, we must check whether the new edge intersects with an edge of adjacent side. The procedure "LocalOther" describes a sequence of these processings.

```
procedure LocalOther(v)
(* The local processing on the vertex v with type ISO-MAX,
RIGHT-SIDE, LEFT-SIDE, OBJ-MAX, or HOLE-MAX *)
begin
current = the side with top coordinates the same as that of v;
previous = the side previous to current in side-heap;
next = the side next to current;
if v.type = LEFT-SIDE then begin
  Add v.end-edge to the tail of current.edges list;
  Change current.top to the begin point of v.end-edge; end
else if v.type = RIGHT-SIDE then begin
  Add v.end-edge to the tail of current.edges list;
  Change current.top to the end point of v.end-edge; end
else if v.type = ISO-MAX then
  Delete current from side-heap;
(* Next cases are for OBJ-MAX or HOLE-MAX *)
else if current.bottom = next.bottom then begin
  result = Concatenate(Reverse(next.edges),current.edges);
  if v.type is OBJ-MAX
     then result.type = outer-loop;
     else result.type = inner-loop;
  Output(result);
  Delete current and next from side-heap; end
else if current.bottom.type = ISO-MIN
or next.bottom.type = ISO-MIN then
  Delete current and next from side-heap;
else if current.bottom.left-side = current then begin
  Add Concatenate(Reverse(next.edges),current.edges)
  to the tail of next.bottom.right-side.edges;
  bottom-vertex = current.bottom.right-side.bottom;
  next.bottom.right-side.bottom = bottom-vertex;
  bottom-vertex.left-side = next.bottom.right-side;
  bottom-vertex.right-side = current.bottom.right-side;
```

```
  Delete current and next from side-heap; end
else if next.bottom.left-side = next then begin
  Add Concatenate(Reverse(next.edges),current.edges)
  to the tail of next.bottom.right-side.edges;
  bottom-vertex = current.bottom.left-side.bottom;
  next.bottom.right-side.bottom = bottom-vertex;
  bottom-vertex.left-side = current.bottom.left-side;
  bottom-vertex.right-side = next.bottom.right-side;
  Delete current and next from side-heap; end
else begin
  Add Concatenate(Reverse(current.edges),next.edges)
  to the tail of current.bottom.left-side.edges;
  bottom-vertex = next.bottom.left-side.bottom;
  current.bottom.left-side.bottom = bottom-vertex;
  bottom-vertex.left-side = next.bottom.left-side;
  bottom-vertex.right-side = current.tottom.left-side;
  Delete current and next from side-heap; end
IntersectSides(previous, the side next to previous in side-heap);
if v.type = LEFT-SIDE or RIGHT-SIDE
  then IntersectSides(current, next);
end
```

Using the above procedures, we describe the complete version of *SPS* algorithm as follows.

```
Algorithm PlaneSweep
(* complete version of SPS algorithm *)
begin
Sort the vertices into vertex-queue and classify vertices;
side-heap = empty;
while vertex-queue is not empty do begin
   v = the bottom vertex in vertex-queue;
   if v.type is OBJ-MIN, HOLE-MIN, or ISO-MIN
      then LocalMin(v);
      else LocalOther(v);
   Delete the bottom vertex from vertex-queue; end
end
```

3.2 Intersecting Two Sides

The procedure "IntersectSides" is used in most of the procedures in §3.1. Its main role is intersecting two sides but not limited to this. Each new event vertex generated from an intersection must be inserted into a correct location of *vertex-queue*, and the top coordinates of certain sides must be modified if necessary. One intersection induces three different vertices with the same coordinate; one *connected-type* vertex and two *isolated-type* vertices, see Figure 8.

For a consistent processing of the algorithm, by assuming as if the two *isolated-type* vertices were kicked out of ϵ-bound by cutting the ends of the corresponding subedges (see Figure 7). Thus, we can uniquely determine the processing order of these three coincident vertices. For a *left* side, the first (highest) edge in the edge list of this side has top-to-bottom direction, and for a *right* side the first edge has bottom-to-top direction. So, we can classify the intersections into four types as in Figure 8. Upper (resp. lower) subedge of an edge generated from the intersection is represented with a superscript '+' (resp. '−'), e.g., $left^+$ and $left^-$, see Figure 8. The procedure for intersecting two sides is described below. Common routines [4, 6, 9] can be used to compute intersections between two monotone curve segments for procedure "IntersectTwoEdges".

```
procedure IntersectSides (side1, side2 )
  (* Intersecting the two side side1 and side2 *)
begin
edge1 = first edge in the list side1.edges;
edge2 = first edge in the list side2.edges;
p = IntersectTwoEdges(edge1, edge2);
Subdivide edge1 and edge2 at p;
Create and order three new vertices;
Insert the three vertices into vertex-queue with sorted order;
end
```

As a final remarks on the description of *SPS* algorithm, we mention some topological characteristics of the intersections computed in this algorithm.

Property 1 *The connected type vertex generated by the intersection of two edges is determined by the two subedges which have their out normals facing each other. The intersection end points of the other subedges become isolated type vertices.*

Property 2 *The connected type vertex generated by the intersection of two edges is determined by one of the next four pairs of subedges which are two consecutive edges in the counter clockwise angular order around the vertex.*
• $left^+, left^-$ • $right^-, right^+$ • $left^+, right^+$ • $right^-, left^-$

We can easily see that the above two properties are satisfied in Figure 8. The four conditions of Property 2 are called as *connection conditions*.

4 Modeling Uncertanties

Developing general mechanisms to produce robust implementations of geometric algorithms is an immediate research goal of geometric and solid modeling. As an effort towards this direction, Milenkovic suggested two techniques called *data normalization* and *hidden variable method* [7] to solve the problem of determining the topological arrangement of polygonal regions in a plane. In §5, we develop two techniques called *multiple intersection merging* and *intersection splitting* which are similar to *vertex shifting* and *edge cracking* in Milenkovic's data normalization technique. First, we modify Milenkovic's robustness conditions [7] to the conditions needed for the robustness of our algorithm.

Definition 4.1 Robustness Conditions
If the SPS algorithm satisfies the following three conditions, the algorithm is robust during the execution of the algorithm.

1. *No two non-coincident event vertices are closer than ϵ.*
2. *No event vertex is closer than ϵ to other edge.*
3. *No two edges are closer than ϵ except at their vertices.*

We say a plane sweep algorithm is *robust within uncertainty bound* ϵ, if the algorithm satisfies the above three conditions. The algorithm is in *normal state* when it satisfies these robustness conditions, and in *abnormal state* otherwise. To determine the uncertainty bound ϵ good for the data normalization of curve segments, we approximate each curve segment by a polygonal chain of line segments. Each curve segment is approximated within an approximation error ϵ_0, and the corresponding uncertainty region looks like a band (called ϵ_0*-band*) as in Figure 9. The uncertainty bound ϵ_0 depends on several factors such as the precision of arithmetic operations, the acceptable error in each application problem, etc. It is known that the uncertainty bound can be calculated from these factors [3, 7].

We can also determine other uncertainty bounds ϵ_a, ϵ_b and ϵ_c. ϵ_a represents the uncertainty bound for the input curve segments when other preprocessing operations are applied to the curve segments. ϵ_b defines an uncertainty bound for the intersection operation itself. When two curve segments with ϵ_a*-band* are intersected, the uncertainty region for the intersection point has a diamond shape as in Figure 10-(a), and we can take the radius of a circle enclosing the region as an uncertainty bound. The size of ϵ_b depends on both ϵ_a and the angle between the two curve segments at the intersection point. In Figure 10-(b), the diamond shape region may become thin and long when two curves intersect with a very small angle (a *long intersection* case). Thus, we fix an appropriate ϵ_b as ϵ_b^* and treat the long intersection case (where $\epsilon_b \geq \epsilon_b^*$) quite differently from the other normal case. ϵ_c is an error bound which counts for the errors involved in the various arithmetics used in testing whether the curve segments intersect. Before continuing the description of algorithm, we state the basic assumptions for the input curve segments.

Assumptions

1. *No two different vertices are closer than ϵ_a. Any points within*

ϵ_a distance are identified as a single coincident vertex.

2. *No vertex is closer than ϵ_a to any other edge.*
3. *The number of edges passing through the circlular region of radius $\epsilon_b^* + k\epsilon_c$ is limited to a certain fixed number k.*

The above assumptions 1 and 2 guarantee that the algorithm is in a normal state at the start. Further, we can make these assumptions hold and make the algorithm robust by removing anomalies arising in the middle of algorithm execution according to the next lemma.

Lemma 4.1 *The SPS algorithm may fall into an abnormal state only when either of the next two cases occurs.*
• *Multiple Intersection : Multiple interesections occur in a very small region.*
• *Long Intersection : Two edges are within a very small distance over an interval, or they intersect with a very small intersection angle.*

5 Correcting Intersection Anomalies

A common error encounted in the plane sweep is the case where more than two edges intersect at the same point or more feasibly in a very small region. Due to the numerical errors resulting from finite precision arithmetic, the *SPS* algorithm may not determine a correct planar topology for an arrangement of planar curve segments in this case. Further, the subedges resulting from multiple intersection may be very tiny and thus may break the normal state. In order to make right decisions for a correct topology, we create a data structure which enables an intersection point to collect all the edges closer than ϵ_b to it. A second problem is about the *long intersection* case which we mentioned in §4. We convert each *long intersection* to a single edge.

5.1 Multiple Intersection Merging

Multiple intersection occurs when an edge intersects at the point which had been already determined as an intersection point of other edges. Of course, a multiple intersection implies that the uncertainty ϵ_a–band of new edge and the uncertainty region for an intersection point have a non–empty intersection. The problem is how to solve the following two questions in a multiple intersection.
• The algorithm has to make the correct topological decisions at each multiple intersection point to construct a topologically correct final object.
• The algorithm must satisfy the *robustness conditions* at each multiple intersections.

The new structure `IntersectionVertex` may have an unlimited number of adjacent edges sorted according to the angular order around the vertex. Thus, we can preserve a correct topology by proceeding the following steps when a multiple intersection is detected.

1. Subdivide the edge into two subedge at the multiple intersection point.
2. Insert the new subedges into the correct location in the angular order.
3. Make new vertex connections if an inserted subedge and its neighboring subedges satisfy the *connection conditions* of Property 2 in §3.2.
4. Update the vertex structure if necessary.
5. Update the uncertainty bound ϵ_b for this intersection.

Figure 11 shows an example of *multiple intersection merging*. In (a), at the event vertex V_3, a new edge E_3 crosses through the uncertainty region for the intersection point of the two edges E_1 and E_2. Before the new edge E_3 appears, E_1 and E_2 construct a new vertex of type *HOLE–MIN* connected to the two subedges E_1^+ and E_2^+. After E_3 penetrates the intersection uncertainty region, two pairs of subedges (E_3^+, E_2^-) and (E_1^-, E_3^-) make the new *connected type* vertices. Note that the two *ISO–MAX* vertices in *vertex–queue* are now replaced with these new vertices, and the uncertainty bound for this intersection is now increased from ϵ_b^1 to ϵ_b^2, where ϵ_b^2 includes the error generated from the calculation of a new intersection, i.e., $\epsilon_b^2 = \epsilon_b^1 + \epsilon_c$. Consider the case where the error propagation occur, see Figure 12. The intersection uncertainty bound is growing up larger and larger in this situation. Since we assume the maximum number of edges passing through a very small region is limited to k in the assumption 3 of §4, the number of error propagation is also limited to k.

New data structures for this scheme are as follows. At first, the structure `point` has two pointers to the structure `IntersectionVertex` (this pointer called as `intersect` field) and to the structure `vertex` which is incident to the point. The coincident points arising from an intersection must point to the corresponding `IntersectionVertex` structure. The structure `IntersectionVertex` has a pointer `subedge` to the list of all the subedges which are emanating from this intersection. These subedges are sorted according to the angular order about this vertex. The other fields of the structure are the `x,y` coordinates of the intersection point (it is the center of uncertainty circle), and an uncertainty radius `epsilon` which `epsilon` contains the value of ϵ_b. Now, we describe the procedure "MultipleIntersectionMerging" based on the above data structures.

```
procedure MultipleIntersectionMerging (edge, intersect)
(* execute intersection collection with one edge edge and
  an IntersectionVertex structure - intersect *)
begin
edge1, edge2 = subedges of edge cut at the intersection;
Insert edge1 into the sorted list of intersect.subedges;
pre-edge = the edge previous to edge1 in intersect.subedges;
next-edge = the edge next to edge1 in intersect.subedges;
if (pre-edge, edge1) pair satisfies the connection conditions
  then begin
  old-vertex = the existing event vertex adjacent to pre-edge;
  new-vertex = new connected type vertex (pre-edge and edge1);
  if old-vertex is not isolated-type then begin
    mate-edge = the edge connected to pre-edge via old-vertex;
    iso-vertex = new isolated-type vertex with mate-edge); end
  Delete old-vertex from vertex-queue;
  Insert new-vertex and iso-vertex into vertex-queue; end
if (edge1, next-edge) pair satisfy the connection conditions then
  repeat the steps similar to the case of (pre-edge, edge1) pair;
repeat the above steps to the subedge edge2;
intersect.epsilon = intersect.epsilon + \epsilon_c;
end
```

By this scheme, we can achieve the following lemmas and solve the multiple intersection anomalies.

Lemma 5.1 *Multiple intersection merging maintains a correct topology to construct the final object.*

Lemma 5.2 *After each multiple intersection merging, the plane sweep algorithm is in a normal state.*

5.2 Intersection Splitting

The *long intersection* case occurs when two edges intersect with a very small angle, see Figure 10. We apply an *intersection splitting* technique to this *long intersection* case. The technique is classified to two types. We consider the long intersection of type *left-left* and *right-right* intersection. When the intersection error bound ϵ_b is larger than ϵ_b^*, we split the intersection into two intersections (denoted *splitting intersections*) and merge the two edges to one of these two edges (denoted *splitting edge*) which connects the two *splitting intersections*. We can merge the two edges and give the same direction to the combined splitting edge as in Figure 13–(a). However, in *left-right* or *right-left* intersection, we simply delete the corresponding *splitting-edge* and only make two *splitting intersections* since the *splitting-edge* is in the interior of the final object, see Figure 13–(b). In the case of *left-left* (resp. *right-right*) intersection, the intersected edges are merged to the *leftmost* (resp. *right*) edge. So, we always merge an inner edge to an outer edge to make the *splitting-edge*. Now, we describe the *intersection splitting* algorithm. We only present the case of Figure 13 in the next procedure. Other cases are similar.

```
procedure IntersectionSplitting(edge1,edge2,type1,type2,ε_b)
(* intersection splitting for two long intersection edges
edge1 and edge2 with type1 and type2 respectively *)
begin
(* Assume edge2 is to be merged to edge1 *)
split-edge = subedge of edge1 within ε_b-uncertanty region;
ignore-edge = subedge of edge2 within ε_b-uncertanty region;
divide edge1 and edge2 to edge1-up, edge1-down and
edge2-up, edge2-down respectively;
if type1 = type2 then begin
  (* Assume edge1 and edge2 are left type edges and
     edge1 is to the left of edge2 below the intersection *)
  make LEFT-SIDE vertex with edge1-down and split-edge;
  make LEFT-SIDE vertex with edge2-up and split-edge;
  make ISO-MAX vertex with edge2-down;
  make ISO-MIN vertex with edge1-down;
  make two IntersectionVertex for splitting intersections;
else begin
  (* Assume edge1 and edge2 are left and right type respectively*)
  make HOLE-MAX vertex with edge1-down and edge2-down;
  make two ISO-MIN vertex with edge1-up and edge2-up;
  make two IntersectionVertex for splitting intersections;
  end
end
```

The topological correctness of this scheme is shown in the following lemma.

Lemma 5.3 *Intersection splitting maintains a correct topology to construct the final object.*

Further, the following lemma shows that the intersection splitting satisfies the three robustness conditions.

Lemma 5.4 *After each intersection splitting, the plane sweep algorithm is in a normal state.*

5.3 Robust Plane Sweep (*RPS*) Algorithm

Summarizing all the above discussions, we now describe the *RPS* algorithm. We omit other descriptions and present only a modified version of the procedure "IntersectSides" which contains the solutions to resolve the intersection anomalies.

```
procedure IntersectSides (side1, side2 )
(* Intersecting two sides side1 and side2 *)
begin
edge1 = the first edge in the list side1.edges;
edge2 = the first edge in the list side2.edges;
if the top or bottom end points of edge1 and edge2 are incident
  then RETURN(); (* Intersection is already checked *)
(p,ε_b) = IntersectTwoEdges(edge1, edge2);
if p does not exist then return();
end1 = the top end point of edge1;
end2 = the top end point of edge2;
if ε_b ≥ ε_b* then
  IntersectionSplitting(edge1,edge2,side1.type,side2.type,ε_b);
else begin
  if p is in the uncertainty region of end1.intersect.epsilon
    then MultipleIntersectionMerging(edge2, end1.intersect);
    else if p is in the uncertainty region of end2.intersect.epsilon
      then MultipleIntersectionMerging(edge1, end2.intersect);
      else begin (* normal intersection *)
        Make three vertices and insert them into vertex-queue;
        Make p.intersect;
      end end
end
```

Finally, we conclude with the final theorem.

Theorem 5.1 RPS *algorithm is robust within* $\epsilon(=\epsilon_a+\epsilon_b^*+k\epsilon_c)$ *and they construct a topologically correct final object.*

6 Conclusion

In this paper, we presented the data structures and implementation details for plane sweep algorithm to construct a planar object boundary from a given input set of plane algebraic curve segments. First, assuming the robustness conditions for the input curve segments, we presented a simple plane sweep (*SPS*) algorithm. Then, using two techniques to enforce robustness conditions to the given input by applying slight modifications to the input geometry if necessary, we presented a robust plane sweep (*RPS*) algorithm which can resolve topological inconsistencies resulting from numerical errors in geometric computations using finite precision arithmetic.

References

[1] Bently, J.L., and Ottmann, T.L., (1979), "Algorithms for Reporting and Counting Geometric Intersections," *IEEE Transactions on Computers*, Vol. 28, pp. 643–647.

[2] Chazelle, B.M., and Edelsbrunner, H., (1990), Private Communications.

[3] Hoffmann, C.M., (1989), *Geometric and Solid Modeling: An Introduction*, Morgan Kaufmann, San Mateo, California.

[4] Johnstone, J.K., and Goodrich, M.T., (1991), "A Localized Method for Intersecting Plane Algebraic Curve Segments," *Visual Computer*, Vol. 7, No. 2, pp. 60–71.

[5] Kim, M.-S., (1988), "Motion Planning with Geometric Models," Ph.D Thesis, Dept. of Computer Science, Purdue University, December, 1988.

[6] Kim, M.-S. and Lee, I.K., (1990), "Gaussian Approximations of Objects Bounded by Algebraic Curves", *Proc. of 1990 IEEE Int'l, Conf. on Robotics and Automation*, May 13–18, Cincinnati, pp. 322–326.

[7] Milenkovic, V., (1988), "Verifiable Implementations of Geometric Algorithms Using Finite Precision Arithmetic," *Artificial Intelligence*, Vol. 37, pp. 377–401.

[8] Preparata, F.P., and Shamos, M.I., (1985), *Computational Geometry: An Introduction*, Springer-Verlag, New York.

[9] Sederberg, T.W., and Scott, R.P., (1986), "Comparison of three curve intersection algorithms," *Computer Aided Design*, Vol. 18, No. 1, pp. 58–63.

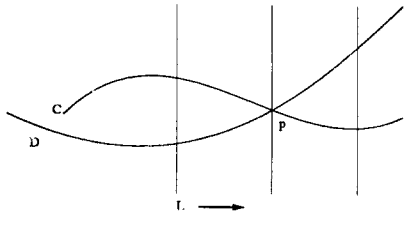

Figure 1: Plane Sweep Algorithm

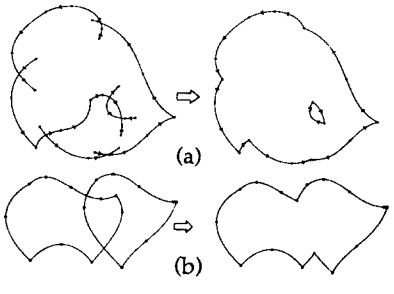

Figure 2: Input Curve Segments and Output Object

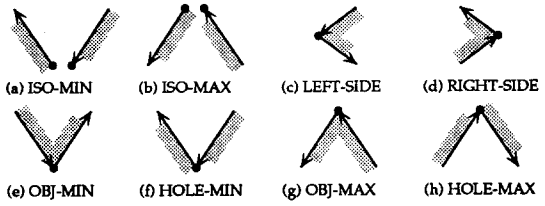

Figure 3: Vertex Classes

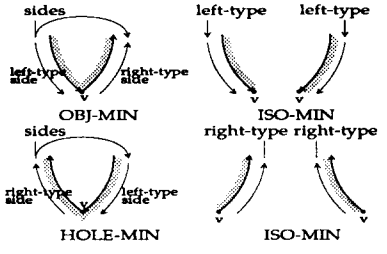

Figure 4: Making New Sides

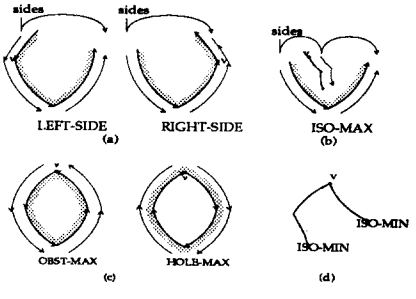

Figure 5: Other Local Processings

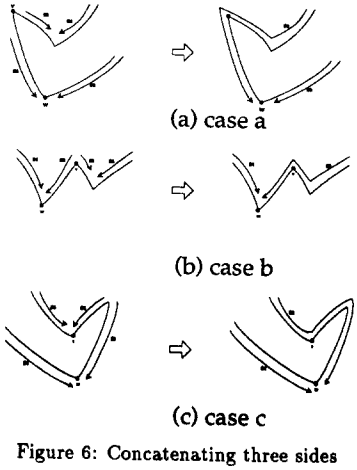

Figure 6: Concatenating three sides

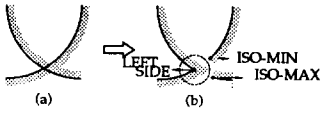

Figure 7: Cutting Subedges

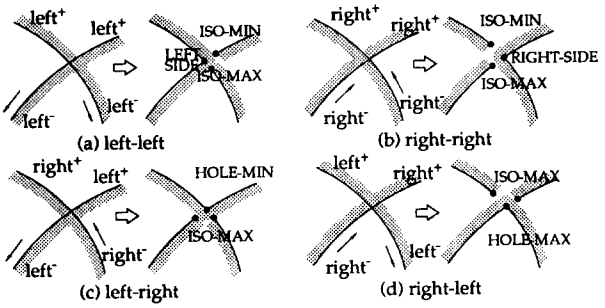

Figure 8: Four Types of Intersection

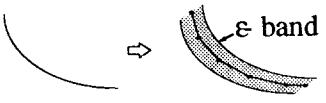

Figure 9: Polygonal Approximation of Curve Segment

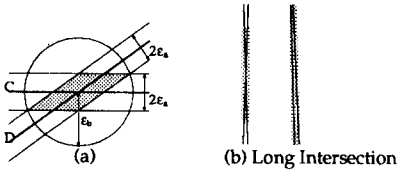

Figure 10: Uncertainty Bounds of Intersection

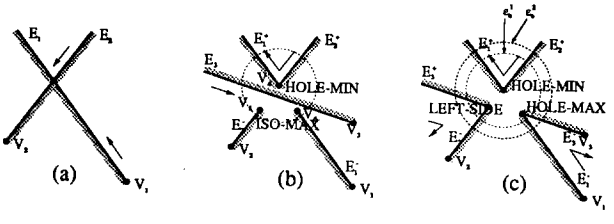

Figure 11: Merging Multiple Intersection

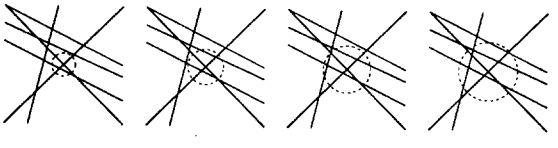

Figure 12: Error Propagation of Multiple Intersection

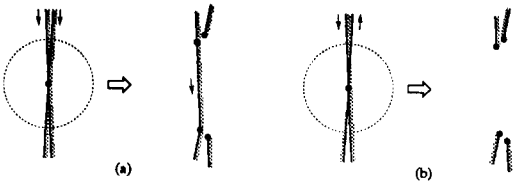

Figure 13: Intersection Splitting