1-1-1986

The Pairwise Intersection Problem for Monotone Polygons

David B. Levine
Dartmouth College

THE PAIRWISE INTERSECTION PROBLEM FOR MONOTONE POLYGONS

David B. Levine

Technical Report PCS-TR86-131

Abstract

Geometric intersection problems arise in a number of areas of computer science including graphics and VLSI design rule checking. Previous work has concentrated on solving the pairwise intersection problem for line segments and iso-oriented rectangles. This thesis extends that work by presenting efficient algorithms to solve the pairwise intersection problem for monotone polygons.

For general line segments, the problem has been solved in O((N+I)*logN) time using a sweeping line technique, where N is the number of segments and I is the number of intersections reported. We combine this technique with approaches taken to solve the iso-oriented rectangle problem to yield an algorithm which solves the pairwise intersection problem for monotone polygons in the same asymptotic time. In addition, there are certain classes of line segments for which the pairwise intersection problem may be solved in O(N*logN + I) time, the best possible. We generalize each such class of line segments to a class of polygons and present algorithms to solve the associated polygon problem. Finally, we discuss the impacts which possible improvements to the line segment problem would have on our results.

Preface

No thesis is ever completed in isolation. In my case, there were many people who have helped me along the way and I would like to take this opportunity to thank them.

First of all, there is my family. Although their knowledge of computer science is somewhat limited, without their support, it is doubtful that this work would have been completed. They were always there when I needed them and offered to help in any way possible. Their contributions cannot be measured.

As far as the body of this work goes, I owe a great deal of thanks to my advisor, Scot Drysdale. It was he who took the time and energy to help me get up to speed in computational geometry, and in computer science as a field. He suggested that I work on this problem and patiently listened to many solutions (good and bad) to this and other problems. He was always very accessible and taught me a good deal about what it means to be a computer scientist. The time and energy which he put into my graduate career is greatly appreciated.

I would also like to thank my thesis committee for their careful readings of several drafts of this work. I know how much time some of them spent on this when they had other, more important concerns.

Finally, I would like to thank my fellow students who dwelled with me in the basement of Bradley Hall while most of this work was completed. The diversions, academic and otherwise, which were to be found there made life as a graduate student much more pleasant for me than it was for some of my peers at other schools.

Table of Contents

List of Illustrations

Chapter 1

Introduction

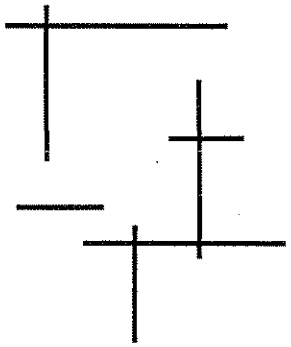
This thesis is devoted to trying to learn faster ways to detect and list pairwise intersections of a set of objects in the plane. Such problems arise naturally in several different fields. For example, when laying out any type of circuit, the wires and components may be modeled by polygons, and then one aspect of design rule checking would be to assure that only certain pairs of the objects intersected each other. As another example, graphics routines that do hidden surface removal often model the surface of a solid by polygons and then remove portions of polygons which overlap.

When dealing with sets of objects such as these, three questions naturally arise concerning intersections among them: a) Do any of the objects intersect?, b) How many of the objects intersect?, and c) Which of the objects intersect? Each successive question is obviously more difficult than its predecessors. In this thesis we will concern ourselves only with the last of these questions, realizing that the algorithms which we present will apply to the other two questions as well, substituting either an increment of a counter for each listing, or just causing a premature stop to be executed when the first intersection is found.

When discussing asymptotic running times for these algorithms, we will generally take N to be the number of objects in the input set, and I to be the number of intersecting pairs. Since the algorithms presented can only compute the number of intersecting pairs by actually finding all such pairs and counting them, the asymptotic running time of any of these algorithms must be the same

as for a similarly based algorithm which answers questions of the third type. If we are only interested in detecting "any intersection", then the associated algorithm, obtained by inserting the premature stop into the reporting phase, will have the same asymptotic behavior as would the original algorithm if there were no intersections (i.e. I=0).

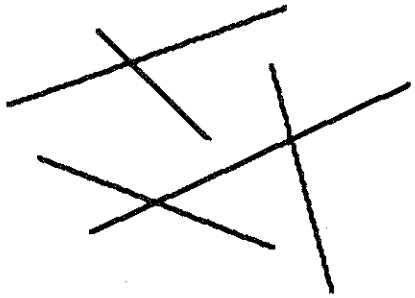
The first class of objects with which we will concern ourselves is line segments. There are two cases of the segment intersection problem. In the first case, we assume that all of the line segments are *isothetic*, meaning that they are all either horizontal or vertical with respect to a given coordinate system (see Figure 1a).



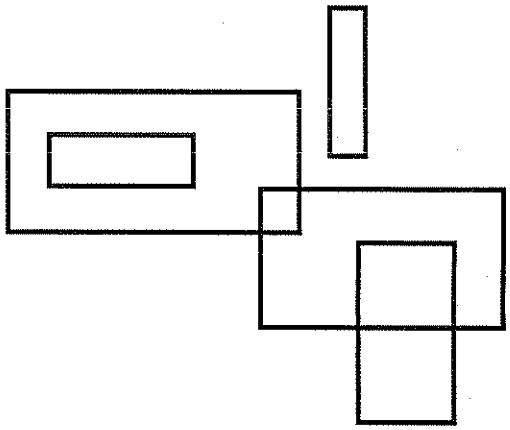
Isothetic Line Segments
Figure 1a

In the second case, the line segments may assume any orientation (see Figure 1b).
In both cases, we are given, as input, the endpoints of each line segment and asked to output a list of those pairs of segments which intersect.

The second class of objects consists of iso-oriented rectangles (see Figure 2). These rectangles are usually given in terms of their lower left and upper right corners, and once again, we output a list of pairs which intersect.



General line segments
Figure 1b



Iso-Oriented Rectangles
Figure 2

Finally, we will concern ourselves with more general polygons, asking the same questions. In this case, each vertex of the polygon is given and a list of pairwise intersections is once again produced as output.

In 1976, Dobkin, Lipton, and Reiss [DobLip] showed that the Element Uniqueness problem takes at least O(N*logN) time to solve. This problem takes as input a multiset of N values

and states whether or not it is also a set, i.e. whether no two of the values are equal. Since a point can be thought of as a line segment of length zero (or a polygon of area zero), it is clear that our first question is at least as difficult as the Element Uniqueness problem and hence must take at least O(NlogN) time. In addition, if we wish to answer the third question, we must take at least O(I) time to list the I pairs of data. Thus, we will consider O(N*logN + I) time to be optimal for solving the pairwise intersection problem.

For purposes of this thesis, we include the following definitions:

Polygon - A polygon is an ordered list of vertices in the plane with straight-line edges connecting them (in the given order). In addition, there is assumed to be a straight-line edge from the last vertex to the first.

Simple Polygon - A polygon is simple if no two vertices are the same and no two edges intersect except at a vertex. This thesis deals only with simple polygons.

Monotone Polygon - A polygon is considered to be monotone with respect to a given line if any line perpendicular to the reference line passes through the polygon at most twice (once in and once out). The term **vertically convex** is sometimes used in the literature to mean "monotone with respect to the x-axis".

Iso-Oriented Polygon - A polygon is iso-oriented if all of its edges are isothetic.

Quadrilateral - A quadrilateral is a simple polygon with four vertices (and hence four edges).

Trapezoid - A trapezoid is a quadrilateral with two parallel edges (usually assumed to be horizontal).

Parallelogram - A parallelogram is a quadrilateral with two sets of parallel edges.

Rectangle - A rectangle is a parallelogram with four right angles.

Iso-Oriented Rectangle - A rectangle is iso-oriented if it has a horizontal edge.

Triangle - A triangle is a polygon with three vertices.

Right Triangle - A right triangle is a triangle two of whose edges (called **legs**) form a right angle.

Iso-Oriented Right Triangle - A right triangle is iso-oriented if it has a horizontal leg. Note that there are four different alignments for iso-oriented right triangles. Note also that an "iso-oriented right triangle" is not an "iso-oriented polygon".

Similar Triangles - Two triangles are similar if they have three pairs of parallel edges. Note that this condition is slightly stronger than the one found in most high school geometry books since it doesn't account for rotations.

Additional definitions will be introduced when needed. Throughout the thesis an attempt has been made to list point names in **boid** type and segment names is normal type.

Two tranformation matrices are commonly refered to in the thesis. Their definitions are given below:

$$\text{rot}(\theta) = \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix}$$

and

$$\text{shear}(\theta) = \begin{pmatrix} 1 & 0 \\ \tan(\theta) & 1 \end{pmatrix}$$

The matrix **rot** merely rotates the coordinate system counter-clockwise through an angle θ thereby giving the illusion that the object is rotated clockwise by the same amount. The matrix **shear** causes vertical lines to appear to be rotated clockwise by θ while fixing horizontal lines.

To apply a transformation matrix to a vector, we will multiply the vector by the matrix (i.e. the matrix is on the right). For example, the transformation **shear**(45) sends (x,y) to (x+y,y).

Chapter 2

Previous Work

Section 2.1 - Intersections of General Line Segments

The segment intersection problem was first solved efficiently by Michael Shamos in his doctoral dissertation [Sha]. He wished only to answer the question about whether or not an intersection occurred:

Problem P1.1 Given N line segments in the plane, determine if any two of them intersect each other.

This problem has immediate application in determining whether or not a polygon is simple, and is a useful step in determining if two polygons intersect. If the input set is restricted to a straight line, say the **x**-axis, this problem is called the Interval Overlap problem.

Shamos and Hoey [ShaHoe] have solved this problem by sorting the endpoints of the segments by x-coordinates, breaking ties based first by placing left endpoints before right ones, and secondly by y-coordinate. They then using a sweeping line to scan through the list, checking for intersections. Intersections are detected within a list of "active" segments, ordered by the **y**-coordinate of the respective segments. (This list is generally stored as a balanced, threaded binary search tree to facilitate faster access.) An "active" segment is one whose left endpoint has been passed by the sweeping line, but whose right endpoint has not yet been passed. Thus, at any given moment in time, the tree contains those segments which intersect the sweeping line and they are ordered according to the **y**-coordinates of these intersections. Shamos and Hoey note

that if two segments intersect each other, then at some point they must be adjacent in this list.

They assume the existence of the following five routines which they use in their algorithm:

Insert(A,T) - Inserts segment A into the tree, T. [This can be performed in O(logN) time.]

Delete(A,T) - Deletes segment A from T. [This, too, is O(logN).]

Above(A,T) - Returns the segment immediately above A in T. [This can be done in O(1) time if a pointer to the segment itself is provided.]

Below(A,T) - Returns the segment immediately below A in T. [This has the same running time as **Above**.]

Intersects(A,B,**p**) - Returns **true** if A and B intersect to the right of **p**, and **false** otherwise. [This can be done in constant time.]

The algorithm is thus:

```
1.  initialize the tree, T;
2.  sort the 2N endpoints by the x-coordinates into List1;
3.  repeat
4.      let p be the next point in List1;
5.      let S be the segment associated with p;
6.      case p:
7.          left endpoint:
8.              Insert(S,T);
9.              let A be Above(S,T);
10.             let B be Below(S,T);
11.             if Intersect(A,S,p) then report it and stop;
12.             if Intersect(B,S,p) then report it and stop;
13.         right endpoint:
14.             let A be Above(S,T);
15.             let B be Below(S,T);
16.             if Intersect(A,B,p) then report it and stop;
17.             Delete(S,T);
18.     end case;
19. until List1 is empty;
20. end.
```

Since the algorithm names the appropriate segments only after it has determined that they intersect, we need only be concerned that it would miss some intersection. But, as explained

in what follows, this cannot happen.

Suppose that E and F are the two segments which form the leftmost intersection. At some point in time E and F must be next to each other in T. (This is certainly true when the sweeping line crosses the point of intersection.) The degenerate case of three segments intersecting in a common point may be handled through proper ordering in **Insert** and **Delete**, so we will ignore it here. There are five ways in which the projections of E and F might become adjacent on the sweeping line: either E is inserted next to F, F is inserted next to E, some segment between them in the tree is deleted, some segment has just crossed E, or some segment has just crossed F. Both of the first two cases are detected by the "left endpoint" clause of the case statement; the third case is detected by the "right endpoint" clause; and the last two cases cannot occur since the E-F pair is the leftmost intersection. Thus, the algorithm must either find this intersection or terminate before this point is reached. However, the only way which the algorithm could terminate earlier would be to find a different intersection. Thus, if there are any intersections, the algorithm must terminate before reaching the leftmost of them.

To finish our proof that the algorithm is correct, we must show that the order maintained in our tree (and used by **Above** and **Below** to check adjacent segments) is the same as the order imposed by the projections onto the sweeping line. The order imposed in the latter case may change (as the sweeping line moves) in three ways: 1) a segment is added to the order, 2) a segment is removed from the order, or 3) two (or more) segments switch positions within the order. The first two cases are handled in the algorithm through **Insert** and **Delete**. The last case arises only if two (or more) segments intersect. But this will never happen since, as shown above, the algorithm terminates before reaching the leftmost intersection. Hence, the algorithm is correct.

To see that the algorithm runs in O(N*logN) time, we merely multiply the cost of each

step by the number of times that that step is executed. Steps 1 and 2 are executed once and take O(N*logN) time. The loop between Steps 3 and 19 is executed N times at a cost of O(logN) per execution. Thus, we see that Shamos and Hoey's algorithm runs in O(N*logN) time and we have proven the following theorem.

Theorem. There exists an O(N*logN) algorithm which determines if there are any pairwise intersections among a list of segments.

Although this algorithm detects intersections, it lists at most one intersection (not necessarily the leftmost) and stops. If we wish either to count the number of intersections, or to list all of them, then a more complicated procedure is necessary. Such a procedure was first presented by Jon Bentley and T.A. Ottman [BenOtt] and is given in detail below. In Shamos and Hoey's algorithm, we stop when we discover that a pair of segments intersect (at some point to the right of our sweeping line). If the sweeping line continued its sweep, then immediately after it passed a point of intersection the order of the segments would be incorrect. Bentley and Ottman's idea is to maintain the order in the tree by actually swapping the nodes corresponding to the intersecting segments that we encounter. Since all of the other updating operations (insertions and deletions) are handled by the original algorithm, our tree will always be in order and we can continue to (try to) detect more intersections. To facilitate this, we will keep a priority queue of "events" similar to Shamos and Hoey's sorted list of endpoints. We will use the term "event" to signify either the insertion of a segment into the tree, indicated by its left endpoint in the list, the removal of a segment from the tree, indicated by its right endpoint in the list, or the swapping of two segments in the tree, indicated by their intersection. Events of the first two types are inserted into the queue at the beginning of the algorithm with a sort, and intersections are inserted as they are discovered at a cost of O(logQ) per intersection where Q is the size of the queue. Since Q is always less than

N^2+2*N (there are at most N^2 intersections and 2*N endpoints), this cost is simply O(logN). If proper record is kept of the pointers to the segments, then the swapping operation will take only constant time. The size of the tree is bounded by N, and hence the insertions and deletions can be done in O(logN) time.

We now present Bentley and Ottman's algorithm for segment intersections. (A worked example may be found in the first section of the appendix.)

```
1.  initialize the tree, T;
2.  sort the 2N endpoints by the x-coordinates into List1;
3.  repeat
4.      let p be the point associated with the next event in List1;
5.      let S be the segment associated with p;
6.      case p:
7.          left endpoint:
8.              Insert(S,T);
9.              let A be Above(S,T);
10.             let B be Below(S,T);
11.             if Intersect(A,S,p)
                then insert the appropriate event into List1;
12.             if Intersect(B,S,p)
                then insert the appropriate event into List1;
13.         intersection:
14.             let S be the lower of the two segments;
15.             let R be the higher of the two segments;
16.             report(S,R);
17.             swap S and R within the tree;
18.             let A be Above(S,T);
19.             if Intersect(A,S,p)
                then insert the appropriate event into List1;
20.             let B be Below(R,T);
21.             if Intersect(B,R,p)
                then insert the appropriate event into List1;
22.         right endpoint:
23.             let A be Above(S,T);
24.             let B be Below(S,T);
25.             if Intersect(A,B,p)
                then insert the appropriate event into List1;
26.             Delete(S,T);
27.     end case;
28. until List1 is empty;
29. end.
```

Once again it is clear that this algorithm only reports pairs of segments that do intersect. Thus, we need only be concerned that it might not find some intersection. The analysis of the previous algorithm showed that the tree could change only at three times: the insertion of a segment, the deletion of a segment, and the intersection of two segments. The same analysis as before shows that insertion and deletion are handled correctly. Swapping two segments at their point of common intersection can be viewed as interchanging them with respect to the order imposed by projection onto the sweeping line. As long as all intersections are detected before the sweeping line reaches the point of intersection, the modifications to the tree will maintain our order. A simple adaptation of the proof of correctness for the previous algorithm gives this result. Thus, the order is always maintained and since all of the necessary checks are made in Steps 7-26 of the program, the algorithm is correct.

It is straightforward to see that all of the steps in this algorithm are basically the same as before except Steps 11, 12, 19, 21, and 25 which can take O(logN) time and which are executed I times, where I is the number of intersections detected. Thus, O(I*logN) time may be spent on these steps. Steps 14-18 and 20 are executed I times taking constant time each, thus giving the overall algorithm an asymptotic running time of O((N+I)*logN).

It is natural to be concerned about the space requirements of this algorithm. The tree of segments has at most N nodes, but the queue of events may have up to N*(N-1)/2 intersections in it. Thus, it would appear that the overall space requirements are $O(N^2)$. However, Kevin Brown has shown [Bro] that it is only necessary to store the leftmost intersection (of those to the right of the sweep line) encountered for any given segment, since all other intersections will be "rediscovered" later. Since two segments are involved in any intersection, the number of intersections in the queue at any given moment is reduced to at most N/2 and hence the space bound is O(N). This

result is summed up in the following theorem:

Theorem. Given a set of line segments in the plane, it is possible to list all of the pairwise intersections among them in O((N+I)*logN) time and O(N) space.

Bentley and Ottman also examine a special case of this algorithm. If all of the segments are isothetic, then the insertions in Steps 11, 12, 19, 21, and 25 will all occur at the beginning of the list and hence we could code this in such a way that each of these insertions takes O(1) time. Thus, the factor of logN in the I*logN disappears from the running time and the algorithm runs in O(N*logN + I) time. Other coding simplifications may be made and the resulting algorithm is given below.

```
1.   sort the appropriate endpoints by x-coordinate into List1;
2.   repeat
3.        let p be the next point in List1;
4.        let S be the segment associated with p;
5.        case S:
6.             horizontal:
7.                  if p is a left endpoint
8.                  then Insert(S,T)
9.                  else Delete(S,T);
10.            vertical:
11.                 let A be Above(S,T);
12.                 while A<upper y-value of p do
13.                      Report(A,S);
14.                      let A be Above(A,T);
15.                 loop;
16.       end case;
17.  until List1 is empty;
18.  end.
```

Simple inspection should show that this algorithm is the same as the previous one except that it is never necessary to insert the vertical segments. Since insertions are never actually put into the queue, the space requirements are O(N) and Brown's improvement is not needed. (It is assumed that all horizontal-horizontal and vertical-vertical intersections are found and reported in

the sorting step.) Thus, we have the following theorem:

Theorem. Given a set of isothetic line segments in the plane, it is possible to list all of the pairwise intersections among them in O(N*logN + I) time and O(N) space.

Using linear algebra, we may extend this input set for this algorithm to include segments of any two arbitrary slopes. This follows from the following lemma.

Lemma. Any set of planar segments which are constrained to two arbitrary slopes may be transformed in linear time to a set of planar segments which are isothetic, and this transformation may be done in such a way that no intersections are created or destroyed.

Proof. By construction. If the two slopes form angles θ_1 and θ_2 with the horizontal, then the transformation **rot**$(-\theta_1)$***shear**$(90-(\theta_1-\theta_2))$ will correctly orient the segments. Since this transformation is 1-1 and onto, no intersections will be created or destroyed and the Lemma is proved.

Thus, by calculating the matrix above (in constant time) and applying it to each segment (in linear time for the entire input set) we may transform an input set containing segments constrained to any two slopes into an isothetic input set, and thus we have the following corollary to our previous theorem.

Corollary. Given a set of line segments in the plane constrained to two arbitrary slopes, it is possible to list all of the pairwise intersections among them in O(N*logN + I) time and O(N) space.

Section 2.2 - Intersections of Sets of Disjoint Segments

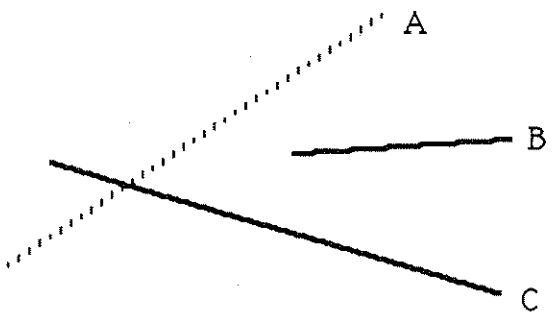
Bentley and Ottman's algorithm is not optimal (including as it does an extra factor of logN in the I*logN term), though it does run in optimal time on isothetic line segments. Knowing one case for which we can optimally solve the problem might lead us to ask whether or not there are other cases which arise naturally and for which we can solve the intersection problem optimally. In a 1983 manuscript, Harry Mairson and Jorge Stolfi [MaiSto] did indeed present such a class of segments. If the segments can be partitioned into two lists, S and T such that there are no intersections among segments in the same list, then they showed how to find all of the intersections in optimal time. We now present their algorithm.

We will keep two trees, SL (holding segments from the S list) and TL (holding segments from the T list), which will contain all of the "active" segments from our two lists. In addition, we introduce the notation $A \cap_{\mathbf{p}} B$ to mean "A intersects B to the left of **p**'s **x**-coordinate". Finally, we introduce five subroutines:

Locate(**p**,L) -	Locates the pair (A,B) of segments in the tree L which are just above and below point **p**.
Delete(A,L) -	Deletes segment A from tree L.
Insert(C,A,B) -	Inserts segment C between segments A and B in the tree which contains A and B.
Above(A) -	Returns the segment above A in the tree containing A.
Below(B) -	Returns the segment below B in the tree containing B.

The first three of these operations can be done in O(logN) time if the trees are kept balanced. The last two can be done in constant time if the tree is threaded.

Ideally, we would process our endpoints in a manner similar that used by Bentley and Ottman. Since neither of the trees will ever need two segments swapped (no intersections exist within a given tree), we could just scan the other tree upon deletion to learn which of its segments intersected the segment we were deleting. (This would cause all intersections to be detected after the sweeping line passed the point of intersection, but since the order within a given tree doesn't change, this poses no problem in and of itself.) Unfortunately, the segments may be situated in such a way that one of them **shields** the intersection of another (see Figure 3).



B shields the A-C intersection
Figure 3

By this we mean that at the time of deletion of segment A, B is the closest segment to A (either above or below) in the other tree, B does not intersect A, and B's projection onto the sweeping line lies between A's projection and the projection of some other segment which does intersect A (in this case, C). Many segments could have their projections lie between A and C without intersecting A. (In this case, each of those segments would shield the A-C intersection.) Thus, at the time A is removed from its list, it might be necessary to search the entire opposite tree to detect

intersections. Mairson and Stolfi solve this problem by not allowing shielding to occur. To prevent shielding, they detect it as soon as they insert the segment which causes the shielding. They then break one of the shielded segments into two segments (one of which they delete immediately according to the deletion procedure) and check to make sure that no more shielding exists. Thus at no time, other than the actual insertion of a segment, does shielding exist.

We may now present an informal description of Mairson and Stolfi's algorithm. We develop a sorted list of endpoints as before, and scan through it processing the respective segments. At the time of insertion, we insert the segment into the appropriate tree, guarding against shielding as described above. At the time of deletion, we remove the segment from the appropriate tree, and scan in the other tree to find segments which intersect the one we are removing.

A more formal description of this algorithm is given below and a worked example may be found in the second section of the appendix.

```
1.    initialize SL and TL to be empty trees;
2.    sort the endpoints of the segments by x-cooridinate into List1;
3.    repeat
4.          let p be the next point in List1;
5.          let C be the segment corresponding to p;
6.          case p:
7.                left endpoint:
8.                      let (Ta,Tb) be Locate(p,TL);
9.                      let (Sa,Sb) be Locate(p,SL);
10.                     let A be a horizontal segment at y = maxvalue;
11.                     let B be a horizontal segment at y = minvalue;
12.                     if Ta∩pSb
                        then let (A,B) be (Ta,Sb);
13.                     if Sa∩pTb
                        then let (A,B) be (Sa,Tb);
14.                     while A∩pB do
15.                           let q be the intersection of A and B;
16.                           let A' be A;
```

```
17.                         while A'∩pB do
18.                             Report(A',B);
19.                             let A' be Above(A');
20.                         loop;
21.                         change the left endpoint of B (as stored in
                            the appropriate tree) to q;
22.                         let B be Below(B);
23.                     loop;
24.                     { at this point we know that the segment does not
                          shield any intersections }
25.                     if C∈S
26.                     then Insert(C,Sa,Sb)
27.                     else Insert(C,Ta,Tb);
28.                 right endpoint:
29.                     if C ∈S
30.                     then begin
31.                         Delete(C,Sa,Sb);
32.                         let (A,B) be Locate(C,TL)
33.                     end
34.                     else begin
35.                         Delete(C,Ta,Tb);
36.                         let (A,B) be Locate(C,SL)
37.                     end;
38.                     while C∩pA do
39.                         report(C,A);
40.                         let A be Above(A)
41.                     loop;
42.                     while C∩pB do
43.                         report(C,B);
44.                         let B be Below(B)
45.                     loop;
46.         end case;
47.     until List1 is empty;
48.     end.
```

To see that the algorithm performs correctly, consider the following invariant: "Every time a new segment is inserted, it shields no intersections." If this is true at all times, then by comparison to the general algorithm, it should be possible to convince ourselves that Mairson's algorithm is correct. We note that if the invariant is true when Step 8 is begun, then it is also true when Step 27 is completed, as the intermediate steps guarantee that the new segment does not shield

any intersections. As removal of a segment cannot cause shielding to occur, the invariant cannot be violated by actions of Steps 28-45. It is trivially true after Step 2 since there is nothing in either tree, and since Steps 3-5 do not affect the trees, they cannot affect the invariant. Thus, the invariant starts true and its truth value is never changed, so it must always be true. However, we previously argued that if shielding did not occur, then the checks in Steps 28-45 would be sufficient to guarantee that the algorithm performed correctly. Thus, we have proven the following theorem:

Theorem. There exists an O(N*logN + I) algorithm to detect pairwise intersections among lists of disjoint segments.

The input set here can be expanded from two disjoint lists to K disjoint lists. The only change necessary is that in Steps 8-45, each ot the K-1 trees which do not contain our segment must be searched upon an insertion or a deletion. (Note that we do not need to check all pairs of trees for possible shielding or possible intersections, only the pairs involving the segment which we are inserting or deleting.) This change adds a factor of K to the running time, yielding an overall asymptotic running time of O(K*N*logN + I) for the new algorithm.

Section 2.3 - Rectangle Intersections Using Segment Trees

Having learned how to detect intersecting line segments we now turn our attention to the next most simple set of objects for which we may solve our problem, namely iso-oriented rectangles. Two sets of authors have presented algorithms for this problem, using two distinct data structures. The algorithms of this thesis follow the approach of the first pair of authors while using the data structure provided by the second. Thus, we present both algorithms for the iso-oriented rectangle problem.

In 1980, Bentley and Wood [BenWoo] solved this problem by observing that if two rectangles intersect, then one of two conditions is met; either one of the rectangles contains the other or some of the edges of the rectangles must intersect. Their algorithm works in two passes, one to detect each type of intersection.

The case of edges intersecting is handled first. Each of the edges may be viewed as an isothetic line segment in its own right, and thus we may simply apply our algorithm for detection of intersections among isothetic line segments. This algorithm may detect more than one segment intersection for each rectangle intersection, but in fact it may never detect more than four per rectangle intersection, so the "I" portion of the asymptotic run time is the same in either case. (In fact, the algorithm will always detect either two or four intersections, depending upone whether or not some corner of one rectangle is contained in the other rectangle.) The time for this step of the algorithm is $O(N*\log N + I)$.

The "containment" intersections are detected through the use of interval trees (also sometimes called segment trees). Interval trees are a form of tree where each leaf represents a

value which is set in advance (usually an integer) and where these values are ordered within the tree. The interval (segment) is placed in the tree by marking each node at which the following two conditions apply: 1) all of the leaves beneath it hold values contained in the interval, and 2) none of its ancestors are marked. In this manner, an interval is marked at most once along each path in the tree. It is straightforward to construct routines which pass down the tree marking (or unmarking) the nodes for an interval so that the interval may be inserted (or deleted) in O(logN) time.

Interval trees are designed to answer the query, "Which intervals in the tree contain a given value?" To answer this query, we merely search for the leaf which contains the value and then report all of the intervals that we find on the path to that leaf. This query may be done in O(logN + I) time where "I" is the number of intervals reported.

(Bentley and Ottman present this phase of the algorithm using a sweeping line which moves "bottom to top". So as to be consistent with the rest of this thesis, we describe an analogous algorithm which uses a "left to right" sweep.)

We use interval trees to solve the "containment" problem for iso-oriented rectangles as follows. For each rectangle, we construct an interval (segment) corresponding to the interval derived by projecting the rectangle onto the **y**-axis. We set up an interval tree for the endpoints of these projections in advance. (This takes linear time.)

For each rectangle we then put the following points into a list: a point on the right edge, a point on the left edge, and any interior point (by convention, the centroid). We sort this list based upon **x**-coordinate and then scan through with a sweeping line, as we did in the other algorithms.

When we reach a left edge point, we insert the interval (derived by projection of this rectangle onto the **y**-axis) into the tree. When we reach a right edge, we delete the corresponding interval. (The property that only one node along a given path is marked and the fact that any two

"marks" placed on the same node correspond to an intersecting pair combine to allow the insert/delete operations to be done quickly enough for our purposes.)

When we reach an interior point, we perform a query on the interval tree asking for intervals containing the **y**-value of our interior point. Since all rectangles which have **x**-value corresponding to this point are in the tree, this query is actually asking for rectangles which contain our interior point. Any rectangle which contains the one which gave us the interior point will necessarily be reported. It is possible (likely even) that "false containments" will be reported as well. (By "false containments", we mean that the algorithm may report that one rectangle contains another when the "larger rectangle" does not entirely contain the "smaller" one; however, if this report is made, then it is true that the two rectangles intersect and there is no additional time penalty.) But each time a rectangle is reported which does not contain the entire rectangle for our interior point, we can be sure that it does indeed intersect our target rectangle since the interiror point just processed is a point of intersection. By the nature of our queries, we can also be sure that this happens at most twice for each pair of intersecting rectangles. The running time for this pass of the algorithm is O(N*logN + J) where J is the number of "containments" reported.

By combining these two parts, and noting that every reporting actually does correspond to an intersecting pair, we see that we have the following result.

Theorem. Given a list of iso-oriented rectangles, the pairwise intersections among them may be listed in O(N*logN + I) time.

Section 2.4 - Rectangle Intersections Using Treaps

A slightly different approach to this problem has been taken by Edward McCrieght [McCr1, McCr2]. He uses a "one pass" approach more similar to that employed by Bentley and Ottman. He does not differentiate between edge intersections and containments. Instead he uses the sweeping line to reduce the problem of rectangle intersection to the problem of interval overlap. He then solves the latter in a manner which yields the same solution(s) as the original problem. (This is analogous to Bentley and Wood's using the interval tree query to detect rectangle containments.)

McCreight solves these two problems (rectange intersection and interval overlap) with a data structure which he calls a **priority search tree**. For purposes of this thesis, we will call it a **treap** as it embodies the concepts of both trees and heaps. A treap is organized like a binary tree in that each node has at most two children and that the nodes in the left subtree have keys less than nodes in the right subtree. Unlike a binary search tree, however, the root of any subtreap is not meant to be a partitioning value, but the point of maximum value. In addition, treaps store a pair of data values which are used as keys rather than a single key. In the case of the interval overlap problem, two coordinates, **b** and **t**, are used. We will insert projections of teh rectangles into this structure. The **t**-value will be the projection of the top edge of the rectangle onto the sweeping line, and the **b**-value will be the projection of the bottom edge, (see Figure 4). The treap is organized so that the interval with the largest **t**-value is at the root and so that the remaining intervals are split among the subtrees based on their **b**-values. McCreight claims [McCr1] that the treap can be kept balanced with respect to height so that searchs in it will have O(logN) running time.

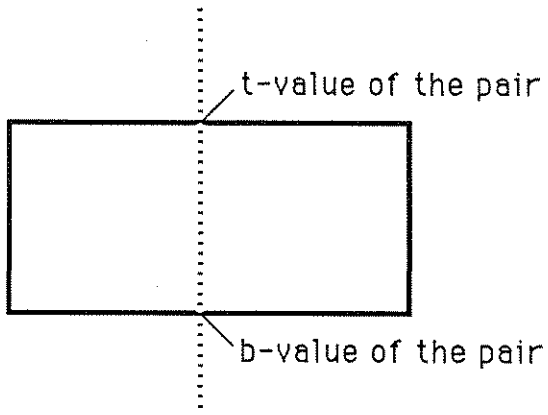


Figure 4

It turns out, however, that despite McCreight's claims that "standard techniques may be used to balance this structure" [McCr1], significant changes must be made. The solution is to more accurately implement the claim that the treap is "half tree, half heap". McCreight describes the details [McCr2]. We will keep the treap ordered and balanced by **b**-value as is the case in a normal search tree. Thus, our root will have near median **b**-value, and we will not be concerned with the **t**-values for purposes of balancing. Thus, we may balance the treap as we would any search tree.

Several of the access functions, however, need to know the maximum **t**-value of any given subtreap. To avoid any problems with these, we must still have each node maintain the property that we may easily access the maximum **t**-value in the subtreap. We do this by storing an additional pair at each node. This secondary pair will be that pair in the subtreap which a) is not the root, b) has maximum **t**-value, and c) has not already been used as a secondary pair further up in the treap. Thus, the root of the treap has the overall maximum **t**-value (either in its primary or secondary pair) and one of its children has this property (relative to the subtreap) as well. The root of the

subtreap which contains secondary pair will not store the actual maximum **t**-value in its secondary field; rather, it will store the maximum pair among those remaining. Thus, whenever a scan reaches a node, it will already have seen (and processed) all nodes in the subtreap which have **t**-value greater than the one in the secondary pair. Hence, the secondary pair can be thought of as holding the absolute maximum **t**-value not yet seen.

The problem with this is that the rotation operations will not (in general) preserve the integrity of the secondary fields. However, we can easily design two procedures which can be used to preserve the desired properties. **DisposeSecondaryPair**(Node) will move Node's secondary pair down the tree (violating the maximality condition at Node), cascading other values down the treap so that the condition is maintained at all nodes in the subtreap. **ExtractSecondaryPair**(Node) is the inverse operation. It will examine the two children of Node, bring up the appropriate one to be the secondary pair, and travel down that branch of the treap, "fixing" each node in turn. These procedures perform functions similar to **UpHeap** and **DownHeap** in more conventional heap implementations [Sed]. Each of them may be performed in O(D) time where D is the depth of the treap. Since we keep the treap balanced, this will be O(logN) time.

To perform a rotation, we must first "**Dispose**" of the secondary pairs associated with each of the nodes involved in the rotation (in increasing order of their depth in the treap). We may then rotate the nodes as we would for any binary search tree and then re"**Extract**" the secondary pairs for each of the nodes (in decreasing order of depth in the treap). Since the nodes being moved have no secondary pairs at the time they are moved, we do not invalidate the heap property. Unfortunately, most tree balancing schemes can create O(logN) rotations per update. Since a single rotation involves three calls to **DisposeSecondaryPair** and **ExtractSecondaryPair**

(one for each node involved in the rotation), our updates may take $O(\log^2 N)$ time. In practice, this might not be so bad, but it is an unnecessary cost. Tarjan has shown [Tar] how to maintain a search tree with at most a constant number of rotations per update using a red-black, 2-3-4 tree as an underlying data structure. This will allow our access functions to run in O(logN) time, the desired result.

We now have a data structure which contains zero or more intervals and we are able to perform the following operations on it:

InsertInterval[b,t,Tr] - Inserts the interval [b,t] into the treap, Tr.

DeleteInterval[b,t,Tr] - Removes the interval [b,t] from the treap, Tr.

ListIntervalsContaining[u,Tr] - Given a test point, u, list those intervals in the treap Tr which contain u.

ListIntervalsOverlapping[u,v,Tr] - Given a test interval, (u,v], list those intervals in the treap Tr which overlap (u,v].

In addition, bookkeeping information is maintained for each node. This information consists of two boolean values, one indicating whther this node's primary pair has appeared as a secondary pair somewhere else (i.e. above) in the treap, and one indicating if the data in this secondary pair is valid. (For instance the root's primary pair does not appear above and a leaf will not have valid data in its secondary pair. Other cases exist for both of these conditions.) Both of these values are easily updated by the access functions. McCreight uses all four of the access functions listed above to solve the rectangle intersection problem. The later results in the thesis use only the first three, so only they are given below in detail. We merely note that **ListIntervalsOverlapping**(u,v,Tr) runs in O(logN + I) time where I is the number of intervals listed.

InsertInterval(b,t,Tr)

```
1.    let [u,v] be the root of Tr;
2.    with [u,v] begin
3.          let Done be false;
4.          let Above be false;
5.          while not Done do
6.                if (not ValidSecondaryPair) or (t>SecondaryPair.upper)
7.                then begin
8.                      DisposeSecondaryPair([u,v]);
9.                      let ValidSecondaryPair be true;
10.                     let Above be true;
11.                     let SecondaryPair be [b,t]
12.               end;
13.               if b>u
14.               then if [u,v] has a right child
15.                     then let [u,v] be [u,v]'s right child
16.                     else begin
17.                           let Done be true;
18.                           let [u,v]'s right child be [b,t];
19.                           let [u,v]'s right child's ValidSecondaryPair be
                              false;
20.                           let [u,v]'s right child's UsedAbove be Above
21.                     end
22.               else if [u,v] has a left child
23.                     then let [u,v] be [u,v]'s left child
24.                     else begin
25.                           let Done be true;
26.                           let [u,v]'s left child be [b,t];
27.                           let [u,v]'s left child's ValidSecondaryPair be
                              false;
28.                           let [u,v]'s left child's UsedAbove be Above
29.                     end
30.         loop
31.   end { with }
32.   end.
```

DeleteInterval(b,t,Tr)

```
1.    let [u,v] be the root of Tr;
2.    with [u,v] begin
3.          while [u,v]<>[b,t] do
4.                if ValidSecondaryPair and SecondaryPair = [b,t]
5.                then ExtractSecondaryPair([u,v]);
6.                if b>u
7.                then let [u,v] be [u,v]'s right child
```

```
8.                  else let [u,v] be [u,v]'s left child
9.          loop
10.         if ValidSecondaryPair
11.         then DisposeSecondaryPair([u,v]);
12.         remove [u,v] from the treap  (i.e. hook up the pointers);
13.   end  { with }
12.   end.
```

Note: In the case of both **InsertInterval** and **DeleteInterval**, balancing should be done at whatever point is appropriate for the tree structure which holds the primary keys. In the case of **DeleteInterval**, Step 12 may involve additional calls to **DisposeSecondaryPair** and **ExtractSecondaryPair**, if it is necessary to "copy" up a node with at most one child.

ListIntervalsContaining(u,Tr)

```
1.    if Tr is empty, then stop;
2.    with [b,t] begin
3.          let [b,t] be the root of Tr;
4.          if not UsedAbove and b<=u<=t
5.          then report [b,t];
6.          if ValidSecondaryPair and SecondaryPair.upper >= u
7.          then begin
8.                if SecondaryPair.lower <= u <= SecondaryPair.upper
9.                then report[SecondaryPair.lower,SecondaryPair.upper];
10.               if u>=b and Tr^.right is not empty
11.               then call ListIntervalsContaining(u,Tr^.right);
12.               if Tr^.left is not empty
13.               then call ListIntervalsContaining(u,Tr^.Left);
14.         end
15.   end  { with }
16.   end.
```

The insertion and deletion procedures have a bounded number of calls to **DisposeSecondaryPair** and **ExtractSecondaryPair** (other than for balancing). Thus, the total cost of each of these routines is O(insertion/deletion + dispose/extract + balancing). Using Tarjan's result, this comes out to O(logN). The correctness of these routines is most easily seen by analogy to normal binary search tree insertion and deletion.

The third algorithm is basically a standard pre-order walk of the tree. It aborts only when it can be shown that no interval in the subtreap will contain u. (The purpose of the first check in

Step 5 is to ensure that no interval is reported more than once.) McCreight shows [McCr2] that the running time of this algorithm is O(logN + I) where I is the number of intervals reported. We present a slightly different proof of this fact which is due to Sam Bent [Bent].

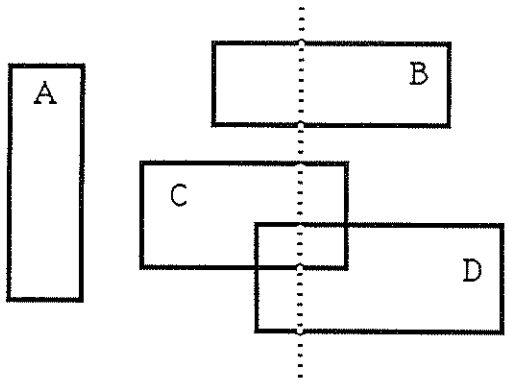
Define the **principal search path** to be the path traced by a traditional search for u, using the primary **b**-values as search keys. This path has length O(logN), and divides the treap into two parts. To the left, all **b**-values (primary and secondary) are less than or equal to u; to the right, all **b**-values are greater than u. (On the path itself, the **b**-values can lie on either side of u.)

Step 9 ensures that the algorithm doesn't visit any nodes at all to the right of the principal search path. The nodes to the left are naturally paritioned into disjoint trees whose roots are left children of distinct nodes on the principal path. Hence there are at most O(logN) of them. The algorithm visits an "upper subtree" of each of these trees. In each case, all of the internal nodes pass Step 7, because they pass Step 5 and <u>all</u> **b**-values are less than or equal to u. Thus, if the algorithm reports k containments in one subtree, it visits at most 2*k+1 nodes (k internal, k+1 external) overall.

Summing over all subtrees, the algorithm visits at most 2*I+O(logN) nodes to the left of the principal path. It visits O(logN) nodes along the principal path, and it visits no nodes to the right of the principal path. Hence the algorithm visits O(logN + I) nodes, doing constant work at each. This proves the result.

We now present the reduction of the segment problem to the interval overlap problem. As before, we use a sweeping line construction. At any given point in time, those rectangles which intersect the sweeping line will be in the treap (see Figure 5).

By checking a rectangle against the treap before we insert it, we will detect each intersecting pair as soon as the sweep line reaches the rectangle in that pair with the rightmost left



Rectangles B, C, and D
are in the treap
Figure 5

edge. The complete algorithm is given below.

```
1.    sort all of the lower left and lower right corners of the
      rectangles by x-coordinate into List1;
2.    repeat
3.          let p be the next point in List1;
4.          let [u,v] be ther vertical edge corresponding to p;
5.          case p:
6.                left endpoint:
7.                      ListIntervalsOverlapping(u,v,Tr);
8.                      ListIntervalsContaining(u,Tr);
9.                      InsertInterval(u,v,Tr);
10.               right endpoint:
11.                     DeleteInterval(u,v,Tr);
12.         end case;
13.   until List1 is empty
14.   end.
```

Making use of our previous arguments, we note that Step 1 is executed once at a cost of O(N*logN) time; Steps 7-9 is executed N times for a total cost of O(N*logN + I) time; and Step 11 is executed N times at a cost of O(logN) time each. Combining these results, we get another O(N*logN + I) algorithm for the rectangle intersection problem.

Since only endpoints are inserted into the lists of "events" and since both segment trees and treaps use only a constant amount of storage per interval, we see that both Bentley and Wood's approach and McCreight's algorithm use O(N) space.

Section 2.5 - Intersections of Other Objects

If the rectangles are not iso-oriented, but are aligned with respect to some other coordinate axes, then a change of basis may be applied to align the rectangles, and then one of the two algorithms just described may be run. The calculation of this change of basis takes only constant time, and it may be applied to the input set in linear time. As before no intersections will be created or destroyed, so the correct answers will be obtained.

If the rectangles are not aligned with respect to a given set of coordinate axes, or if the input set consists of other types of polygons, then the best known algorithm is the naive one; namely check each polygon against every other and report those that intersect. Clearly this takes $O(N^2)$ time regardless of the number of intersections detected. There are at least two ways to improve this time. One was developed by G.F. Swart at the University of Washington [Swa] and will be discussed in Chapter 5; the other is the main result of this thesis and is the subject of Chapter 3.

Nievergalt and Preperata [NiePre] have done work on the associated (and more difficult) problem of actually computing the region of intersection, but they are concerned with two polygons only. They assume that the intersection exists and thus would need to apply some other algorithm, either the naive one or one discussed later, to actually detect the intersections.

Chazelle has shown [Cha] that Bentley and Ottman's algorithm is not always optimal for the segment problem, but we do not make use of his results and therefore defer their discussion to Chapter 5.

Chapter 3

Primary Results

Section 3.1 - Introduction

In this chapter we will expand the set of inputs for which we detect pairwise intersections to include all polygons of bounded edge size which are monotone with respect to the **x**-axis. (Actually, they can all be monotone with respect to any given axis and we will apply the appropriate transformation in linear time to reduce to this case.)

As in the case of intersecting segments, we present several classes of inputs and several algorithms to use for them. The most significant class is the general case, for which we will use a modified treap structure which will yield a running time which is the same as Bentley and Ottman's. We will also study more complicated iso-oriented polygons as well as lists of disjoint polygons. In each of these cases we will be able to solve the problem in the optimal amount of time.

Section 3.2 - Polygon Intersections Using Treaps

Our first algorithm is based upon McCreight's solution to the rectangle intersection problem. We reduce the polygon intersection problem to the interval overlap problem in much the same way that McCreight did. We envision a sweeping line passing left to right through our input set, and at any given time we will have a list of "active" intervals which will be in our treap. By examining this list of intervals, we will determine the polygon intersections.

Although in the case of the rectangle intersection problem the intervals along the sweeping line never changed for a given rectangle (i.e. the top and bottom of the rectangle are at constant **y**-value), in the case of the polygon intersection problem these values will vary. To help keep track of these variations we use the notions of upper and lower hulls. The upper (lower) hull of a monotone polygon is the ordered set of vertices (and their associated edges) lying clockwise (counterclockwise) between v_0 and v_1 where v_0 is the vertex of minimum **x**-coordinate (lowest in case of ties) and v_1 is the vertex of maximum **x**-coordinate (highest in case of ties).

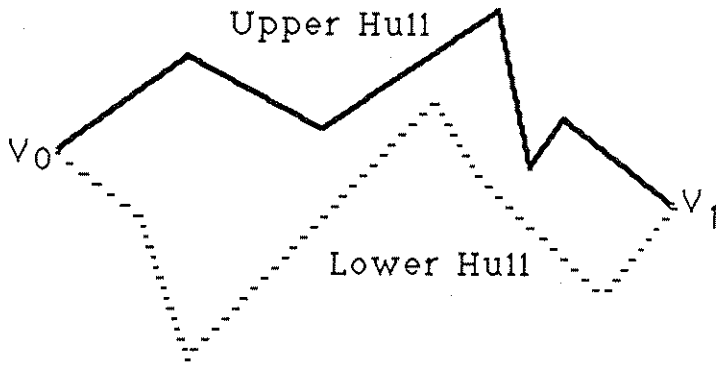
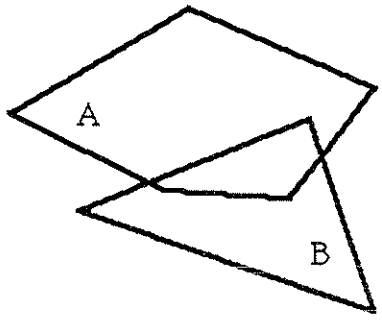


Figure 6

Since the polygons are monotone with respect to the **x**-axis, the upper and lower hulls are well defined and can be thought of as piecewise linear functions. They will correspond to the top and bottom intersections, respectively, of our sweeping line with the polygon.

At this point we note that it no longer suffices just to maintain the upper and lower hulls to solve the problem. In McCreight's algorithm we detect the intersection of two rectangles when the one with the rightmost left edge is inserted. However, at the time the corresponding polygon is inserted, it may not be possible to detect this. The example in Figure 7 makes this clear.



The intersection can't be detected
at the time of insertion of either
of the polygons
Figure 7

If, however, it were only necessary to detect containments (rather than overlaps), then modifications to McCreight's algorithm to handle hulls would be sufficient to solve the problem, since if the second polygon is contained in the first, we may detect that at any time when the second polygon is "active". In order to solve the problem using this type of approach, we would, however, have to find all "non-containment" intersections in some other manner.

This was exactly the approach taken by Bentley and Wood when they solved the iso-oriented rectangle intersection problem. If, however, we try to modify segment trees to handle

the concept of upper and lower hulls, we run into difficulties choosing values for the leaves. Thus, we will use the approach of Bentley and Wood combined with the treap data structure. As was the case when we used this approach for the rectangle intersection problem, we may detect "false containments", but we note once again there will be no time penalty for these repeated discoveries.

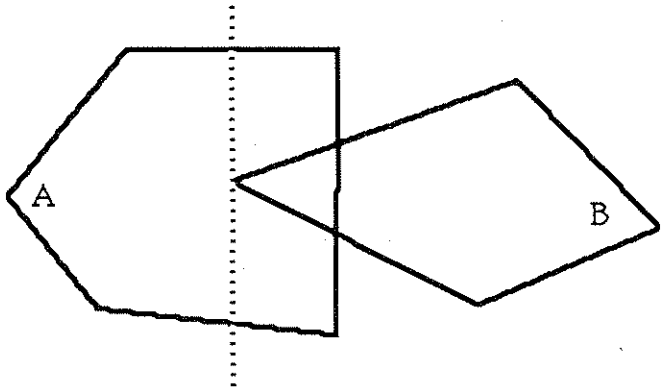
For the rectangle intersection problem, we construct a list of left and right endpoints for the rectangles. Whenever a left endpoint is encountered, we search the treap looking for rectangles which overlap or contain the current one. Then, after reporting the results, if any, of the search, we insert the corresponding interval into the treap. When a right endpoint is encountered, the corresponding interval is deleted from the treap.

Our list for the general problem consists of three types of "events": left endpoints, right endpoints, and edge intersection points. We will find all of the edge intersection points in advance by running Bentley and Ottman's algorithm on the edges of the polygons (as was the case with Bentley and Wood's algorithm, though the asymptotic running time is greater in this case).

At each node of the treap, rather than storing a pair of constants representing the values of the top and bottom edges, we will store a pair of pointers to linked lists containing the upper and lower hulls. We will represent each hull by a linked list of its vertices and we may easily evaluate its value at any given **x**-value by simply scanning through the list until we find the appropriate edge and then doing a simple interpolation. Since our sweeping line moves left to right, we may discard any nodes we pass in the list. Thus, for each polygon we discard O(# of sides) edges, but the total cost of this is O(N) and may be ignored in run time analyses. The interpolations take O(1) time apiece, just as it took O(1) time to look up the value of the top and bottom edges before.

When we encounter a left endpoint of our polygon, (**lx**,**ly**), we search the current treap and list all intervals which contain **ly**. As before, not all of these will be true containments (see

Figure 8), but they all represent intersections and no pair is found more than one extra time as a result of this search; this adds nothing to the overall cost.



B is reported as "contained in A". Note that although this is obviously false, B does intersect A
Figure 8

We then insert the pointers to the upper and lower hulls into the treap.

When we encounter a right endpoint, (**rx,ry**), we merely delete the appropriate pair of pointers from the treap.

Intersection points are the most difficult. Each intersection point refers to two polygons. Beyond the intersection point, if no updating were done, then it is possible that the ordering properties of the treap would be violated. (We will look at this a little more later.) We must devise a way to update the treap so as to preserve the ordering properties to such a degree that future searches work as expected. The simplest, but not necessarily the fastest, way to do this is to delete one of the polygons and then reinsert it just after the intersection. After the reinsertion is done, the ordering will once again be correct. (In practice we may do this at the point of intersection by checking the slopes of various edges to break the ties.)

The question remains about how insertions, deletions, and searches are actually

performed. The answer is quite simple. For insertion and deletion we use McCreight's algorithms except that when they say to check the top or bottom value, we check the value of the upper or lower hull at the **x**-value of the sweeping line. Searching is even easier. Since we search at the moment of insertion, and since at that point our upper and lower hulls have the same value, we need only check to see if the intervals in the treap contain our target point. Thus, we need only call **ListIntervalsContaining**, with the **y**-value of the left endpoint of our hull. As with insertion and deletion, we replace the checks for top and bottom values with the appropriate "hull function" calls.

Thus, insertion and deletion take O(logN) time and the searching/listing takes O(logN + S) time where S is the number of intervals containing our test point. The swaps at the intersection points involve one insertion and one deletion and thus take O(logN) time.

Thus, the entire containment pass of the algorithm runs in time O((N+I)*logN + S) where I is the number of edge intersections and S is the number of containments. The first pass, detecting edge intersections, takes O((N+I)*logN) time and thus the algorithms overall running time is determined by the second step.

Other than replacing value checks by function calls as described above, there are no changes to the subroutines and we now present the general algorithm for purposes of completeness. (A worked example may be found in the third section of the appendix.)

```
1.    let List1 hold all of the pairs found by running Bentley and
      Ottmann's segment intersection algorithm on the edges of
      the polygons, but not including the intra-polygon intersections;
      which will occur at the vertices; { This list is already sorted }
2.    let List2 hold all of the extremal hull vertices; { by this we
      mean the first and last vertices of upper and lower hulls }
3.    sort List2 by x-cooridinate;
4.    merge List1 and List2 into List3;
5.    repeat
6.          let p be the next point/pair in List3;
7.          case p:
```

```
8.                  left endpoint:
9.                      let u be the y-value of p;
10.                     ListIntervalsContaining(u);
11.                     Insert(the hull functions associated with p);
12.                 intersection:
13.                     UpdateTreap(p); { This will generally be a call
                        to Delete followed by a call to Insert }
14.                 right endpoint:
15.                     Delete(the hull functions associated with p);
16.         end case
17.     until List3 is empty
18.     end.
```

Steps 5-10 and 13-18 run as before. Step 4 takes time linear in the number of events, Step 2 takes O(N*logN) time, and Step 1 takes O((N+I)*logN) time. We note that Steps 12-13 are executed I times and that if they could be done in constant time, then the containment pass of the algorithm would run in optimal time. This would imply that any improvements to the segment intersection problem would be reflected immediately in the polygon problem. Unfortunately, we are not able to prove this result at this time. (Some improvements to the segment algorithm will be discussed in Chapter 5 however.) Steps 12-13 still require O(logN) time per execution and thus the overall running time is O((N+I)*logN + S) time where I is the number of edge intersections and S is the number of "containments".

Before we turn our attention to the sizes of I and S, we should investigate the issue of treap updates more closely. In general, the edge intersections will be of three types. Type 1 intersections are between two lower hulls. Type 2 intersections are between upper hulls. Type 3 intersections are between one lower and one upper hull.

It is a consequence of the treap structure that no update need be done for Type 3 intersections. The proof of this fact follows from the fact that there are no upper value/lower value interactions in the treap. Both before and after the sweeping line passes the intersection point, the lower values (primary pairs) have the same relationships with respect to each other; thus there is no

change in the "shape" of the treap. Similarly, both before and after the sweeping line passes the intersection point, the upper values (secondary pairs) have the same relationships with respect to each other; hence, no changes need be made to the secondary pairs. The combination of these two observations substantiates the claim.

Type 2 intersections may be fairly easy to deal with. Since two upper hulls are intersecting the functions must have the same **y**-values. Thus, if the secondary pairs are on the same path within the treap, one must be the parent of the other in some subtreap (since the **y**-value is used as in a heap). To update the treap is now merely a matter of "promoting" the son. If the parent belongs in the same subtreap from which the son came, then only a swap need be done, but if the parent belongs in the other subtreap, then problems may result. If the two nodes are in different subtreaps, then no update need be done.

A similar problem arises for Type 1 intersections. If the pair is a leaf and its parent, then only a swap need be done, but otherwise problems may result.

Unfortunately, at this time no update strategies are known which run quickly on Type 1 and Type 2 intersections. The best general strategies examined have cases which cause O(logN) data movements and are thus no better than the "**Delete** and re**Insert**" method. It remains to speed up these updates or to prove that this can't be done.

There are two remaining issues to resolve with respect to this algorithm. The first is the space requirements. If Brown's improvement to Bentley and Ottman's algorithm is used in Step 1, then the entire algorithm would seem to need only O(N) space. The treap needs linear space (assuming that the number of edges per polygon is bound by a constant, say K), but the list initially contains 2N+I events. The space requirements may be reduced to O(N) as follows. Rather than using the "two pass" approach where we find all of the edge intersections and then restart our

sweeping line for containments, we solve both problems simultaneously. Since all edge intersections are detected before the sweeping line gets to them we may simultaneously maintain the binary tree of segments and the treap, updating both of them when the sweeping line reaches the point at which the segments (hull edges) cross. In this case, use of Brown's improvement guarantees that only O(N) intersection events are in the list at any given moment and thus the space bounds are met.

The second issue is the size of I and S. S may obviously be $O(N^2)$ as can be seen by considering concentric triangles. The bounds on I are also quadratic, but in this case we should also pay attention to how K, the maximum number of edges per polygon, affects these requirements. If we have two polygons with K edges, how many edge intersections may they cause? The naive answer is $O(K^2)$, but since our polygons are monotone with respect to the **x**-axis, we may further reduce this to O(K). To see this we examine edge intersections by hull (i.e. upper hull-upper hull, lower hull-upper hull, upper hull-lower hull, and lower hull-lower hull). Each hull is a piecewise linear function, so whenever an edge intersects another we will charge that intersection to the edge whose right endpoint has smaller **x**-coordinate. In this manner we see that at most one intersection is charged to each edge per match-up. Thus, at most K intersections are charged per match-up and thus at most 4K intersections occur. (The number 4K is not a tight bound, but it suffices for our purposes.)

Thus, we have proven the following theorem:

Theorem: Given N polygons with at most K edges each, there exists an algorithm to list the pairwise intersections which runs in O((N+I)*logN + S) time and O(K*N) space.

To close, we note that if the polygons are not monotone with respect to the **x**-axis, but are monotone with respect to some other line, then we may rotate our coordinate system to allow

the algorithms to perform correctly. If the polygons are not monotone with respect to the **x**-axis even after rotations, then at some point it will be necessary to have two (or more) intervals in the treap for a given polygon. One solution to this is dilemna is to subdivide the polygon into monotone pieces, but this can get expensive. Swart [Swa] discusses solutions to this problem, but all involve additional time factor related to the number of intervals which must be inserted.

Section 3.3 - Intersections of Sets of Disjoint Polygons

When we studied the segment intersection problem, we examined a general algorithm, Bentley & Ottman's, which ran in O((N+I)*logN) time, and then studied special cases for which we could devise algorithms which ran in optimal time. The most interesting of those special cases was when there were two lists of segments such that no segment intersected any other segment in the same list. We have obtained an O((N+I)*logN) time algorithm for the general case of the montone polygon intersection problem, and we will now study special cases of the input for this problem which admit optimal solutions. Similar to the situation we had before, a very interesting special case is when the polygons may be split into two lists such that no two polygons in the same list intersect each other.

Our previous algorithm was obtained by generalizing McCreight's data structure to allow it to handle upper and lower hulls rather than fixed values. This algorithm is obtained in an even more straightforward manner, by generalizing Mairson and Stolfi's algorithm. An interesting situation arises here, however. Whereas with general polygons we could have up to O(N) containments per polygon (if they were all nested), with this type of input we may have only one containment per polygon. To see this, consider what would happen if one polygon were contained in more than one other. Then an interior point of that polygon would belong to at least three polygons. However, a given point may appear at most once per list and we have only two lists, so a contradiction results. Thus, each polygon is contained in at most one other.

We may combine this fact with our general "find edge crossings, then find containments" approach to yield an optimal algorithm for the pairwise intersection problem when the

input consists of lists of disjoint polygons. We begin by noting that Mairson and Stolfi's original algorithm actually works on a slightly more general input set than claimed. The segments in a given list can intersect each other as long as each intersection point is at the endpoint of one of the segments involved. In this case, a test of the slopes involved could ensure proper insertion and deletion and the two (or more) trees would still have the property that they never needed to have two segments swap places. Thus, all of the edges from a given polygon may go into the same list of segments.

When we are presented with two lists of disjoint polygons, we now create two lists of almost disjoint segments. We may use Mairson and Stolfi's algorithm to find all of the edge crossings and it will remain only to detect containments. However, it is easy to determine if a point, **p**, from an S polygon is contained in a T polygon. We merely call **Locate**(**p**,TL) and check if the "above" and "below" segments belong to the same polygon. This test may be done at any time that the polygon S is "active" and we will choose to do it when S is first encountered. The complete algorithm it given below. (A worked example may be found in the fourth section of the appendix.)

```
1.  initialize SL and TL to be empty trees;
2.  sort the endpoints of the edges of the polygons by x-coordinate
    into List1;
3.  repeat
4.      let p be the first endpoint in List1;
5.      let C be the edge (segment) corresponding to p;
6.      case p:
7.          left endpoint:
8.              let (Ta,Tb) be Locate(p,TL);
9.              let (Sa,Sb) be Locate(p,SL);
10.             if C ∈ S and Ta and Tb are from the same polygon
11.             then report C's polygon and Ta's polygon as an
                intersecting pair;
12.             if C∈ T and Sa and Sb are from the same polygon
13.             then report C's polygon and Sa's polygon as an
                intersecting pair;
14.             let A be a horizontal segment at y = maxvalue;
```

```
15.                     let B be a horizontal segment at y = minvalue;
16.                     if T_a ∩_p S_b
17.                     then let (A,B) be (T_a,S_b);
18.                     if S_a ∩_p T_b
19.                     then let (A,B) be (S_a,T_b);
20.                     while A ∩_p B do
21.                             let q be the intersection of A and B;
22.                             let A' be A;
23.                             while A ∩_p B do
24.                                     report their respective polygons as an
                                        intersecting pair;
25.                                     let A' be Above(A');
26.                             loop;
27.                             change the left endpoint of B (as stored in
                                the appropriate tree) to q;
28.                             let B be Below(B);
29.                     loop;
30.                     { at this point we know that the segment C does
                          not shield any intersections }
31.                     if C ∈ S
32.                     then Insert(C,S_a,S_b)
33.                     else Insert(C,T_a,T_b);
34.             right endpoint:
35.                     if C ∈ S
36.                     then begin
37.                             Delete(C,S_a,S_b);
38.                             let (A,B) be Locate(C,SL)
39.                     end
40.                     else begin
41.                             Delete(C,T_a,T_b);
42.                             let (A,B) be Locate(C,TL)
43.                     end;
44.                     while C ∩_p A do
45.                             report C's polygon and A's polygon as an
                                intersecting pair;
46.                             let A be Above(A)
47.                     loop;
48.                     while C ∩_p B do
49.                             report C's polygon and B's polygon as an
                                intersecting pair;
50.                             let B be Below(B)
51.                     loop;
52.         end case;
53.     until there are no more points in List1;
54.     end.
```

The correctness of this algorithm follows from the correctness of Mairson's original algorithm and from the additional tests in Steps 10-13. That these tests detect only containments is obvious. That all containments are detected follows from the argument which proved that there is at most one containment per polygon.

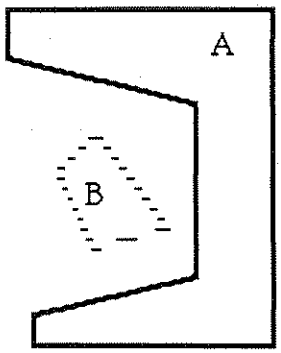
All that we have added at Steps 10-13 is a constant cost (two conditional tests with at most one additional statement, the report, executed), so we have not affected the overall running time of Mairson's algorithm. Thus, we have the following theorem.

Theorem. There exists an O(N*logN + I) algorithm to solve the pairwise intersection problem for lists of disjoint monotone polygons (of bounded edge size).

There are two generalizations which follow immediately from the design of this algorithm. To obtain the first, we observe that this algorithm does not talk about upper and lower hulls. Thus, it is not necessary to restrict the polygons to be monotone with respect to the **x**-axis. Rather, as long as they are simple (and hold the non-intersecting property), we can solve this problem. The only change, which is necessary, is to modify the code in Steps 10-13 to additionally verfify that our test point lies within the appropriate polygon, since that need not be the case if the polygon has a "dent" in it as in Figure 9.

To do this, we simply pre-process each polygon as follows. We find a leftmost vertex. For both of the edges incident to this vertex, we know which side of the edge (segment) represents the interior of the polygon. We may now scan the edges in order, determining which side of a given edge represents the interior of the polygon by checking its predecessor. This can clearly be done in O(N) time overall, so the algorithm doesn't take any longer (asymptotically) to complete. Thus, we have the following theorem.

Theorem. There exists an O(N*logN + I) algorithm to solve the pairwise intersection problem for lists of disjoint simple polygons (of bounded edge size).



B is not contained in A even though A
is immediately above and below B at
the time of B's insertion
Figure 9

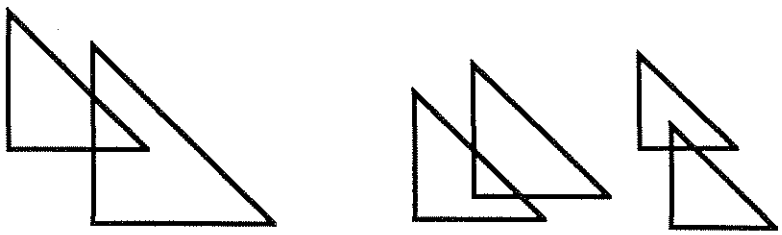
The second generalization concerns input sets which are composed of more than two disjoint lists. The comments made in Section 2.2 regarding line segments apply directly here (substituting the word "polygon" for the phrase "line segment") and hence the details are omitted.

Section 3.4 - Intersections of Similar Right Triangles

In addition to disjoint lists of polygons, there are other special cases of the input sets for which we may solve the intersection problem in optimal time. The remaining such sets have the property that they may all be decomposed into sets of triangles and parallelograms which are *similar*, i.e. which have the same angles and orientation, but which vary in size and placement. We shall begin our study of these by examining the smallest such case, namely iso-oriented right triangles with the hypoteneuse "in the first quadrant relative to the legs".

Let us carefully examine the actions of the algorithm from Section 3.1 upon data composed entirely of such triangles. The first step, running Bentley and Ottman's algorithm, may be improved to optimal time since we may detect edge intersections optimally by examining each of the three pairs of slopes as described at the end of section 2.1. As we noted in Section 3.1, if only intersections of Type 3 (upper hull-lower hull) take place, then we needn't update the treap structure and the remainder of the algorithm will also run in optimal time. But this is actually the case here! We may ignore vertical edges as long as the hypotenueses are inserted before containments are checked. Thus, there are only three types of intersections to consider: leg-leg, hypoteneuse-hypoteneuse, and leg-hypoteneuse. Since the triangles are similar, all of the hypoteneuses are parallel and hence cannot cross one another. The same is true of the legs (which are all horizontal). The only remaining intersections are between a hypoteneuse and a leg, i.e. between an upper and a lower hull. Thus, the treap does not need to be updated for edge intersections and our algorithm will run in optimal time. Of course, appropriate changes must be made to the code to ensure that the unnecessary treap updates are not done. In fact, we may save

even more work (though no asymptotic gain is made) by ignoring the vertical edges entirely. If we detect containments when the polygon is deleted rather than inserted, we may ignore the vertical edges since the intersection will be detected at some other step (see Figure 10).



Either "containment" (left) or
Hypotenuse-Leg intersection (right)
always occurs
Figure 10

The modified algorithm is thus given below.

```
1.    run Bentley and Ottman's algorithm for isothetic segments on
      the horizontal legs and the hypoteneuses; { Apply
      transformation matrices as necessary. }
2.    sort the endpoints of the horizontal legs by x-coordinate
      into List1;
3.    repeat
4.          let p be the next point in List1;
5.          case p:
6.                left endpoint:
7.                      Insert(the hull functions associated with p);
8.                right endpoint:
9.                      Delete(the hull functions associated with p);
10.                     ListIntervalsContaining(p);
11.         end case;
12.   until List1 is empty;
13.   end.
```

Step 1 runs once in O(N*logN + I) time; Step 2 runs once in O(N*logN) time; Step 10 runs in O(N*logN + S) time overall; Steps 7 and 9 run N times each in O(logN) time; and the other steps are executed at a total time cost of O(N). Thus, the overall running time of the algorithm can be seen to be O(N*logN + S + I).

An interesting consequence of this approach is that since the treap never needs to be updated, we did not need to execute Step 1. We could merely have checked (after Step 7) for segments overlapping the vertical leg instead of just checking for intervals containing **p** at Step 10. This, however, is McCreight's original algorithm (on the modified data structure, of course). Thus we have proven that the original algorithm was much more versatile than we expected.

Finally, we note that none of the operations performed took advantage of the fact that the hypotneuse was "in the first quadrant with respect to the legs". As long as all of the upper hulls were parallel and the same was true of the lower hulls, both algorithms (the one presented here and the original) would work correctly. Thus, we may generalize the description of the input to read "any similar triangles containing a vertical edge".

Section 3.5 - Intersections of Other Almost Iso-oriented Polygons

When we studied line segments, we noted that in addition to lists of disjoint segments, there was another class of input for which we could solve the intersection problem in optimal time. This was the class of segments where each segment has one of a small number of slopes. We solved this problem first for two slopes. If we generalize it to more slopes, then the time bound grows as the square of the number of slopes (since there are that many pairs upon which we need to run the algorithm). The same generalizations may be applied to the polygonal intersection problem as well. We will use two techniques to do so. The first will effectively reduce the number of slopes by one (no real savings asymptotically, but potentially sizable in practice since the number of slopes is assumed to be small). The second is a generalization to any number of slopes, as before, at a time penalty proportional to the square of the number of slopes admitted.

To demonstrate the first technique, let us consider trapezoids of the form illustrated in Figure 11, i.e. trapezoids with two horizontal edges, one of slope 1, and one of slope -1, with the slope 1 edge being the leftmost.

Figure 11

When we were dealing with right triangles, we noted that there was no need to consider the vertical edge. Here, however, there is no vertical edge, but we can create one by applying an appropriate

transformation, **shear**(-45). This turns all lines of slope -1 into vertical lines (see Figure 12), effectively reducing the number of slopes under consideration by one.

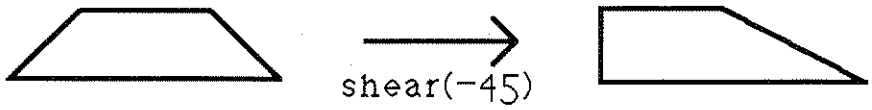


Figure 12

To demonstrate the second technique we will continue to examine this data set. We now subdivide the sheared trapezoids into a rectangle and an iso-oriented right triangle. We know how to solve each of these subproblems in optimal time and we can "merge" the two solutions into one which will also run in optimal time. As before, we will detect the edge intersections first, using three applications of Bentley and Ottman's algorithm - one for each pair of slopes. Then we will solve the "containment" problem. However, if we keep all of the hulls in one treap, then it is possible that we will have upper hull-upper hull intersections and will thus have to update the treap at a cost of O(logN) per edge crossing. To avoid paying this cost, we will keep two treaps - one for the rectangles and one for the triangles. Whenever a left endpoint of a rectangle is encountered, we will search both the "active rectangle" treap and the "active triangle" treap for containments. (We could do this for the left edges of triangles instead, but it need only be done once.) As before we may find false containments, but we find at most one of them per intersecting pair and any that we do find are indeed indicative of in intersecting pair.

This technique may be generalized to cases where decomposition is into more pieces. As mentioned above, however, the cost is relatively high. If we admit just one more slope, e.g. allow horizontal and vertical lines as well as those at forty-five degree angles, and if we further allow any

combinations of these lines which form simple polygons, then we must run Bentley and Ottman's algorithm six times (once for each pair of slopes) and there will be nine, the number of non-vertical slopes squared, treaps to check for containment since any non-vertical line may form the upper or the lower hull or both. The additional cost in each case is proportional to the square of the number of slopes (less one if we can use the first technique) since each pair of slopes is handled separately. Whether or not this cost is too prohibitive depends upon the expected number of intersections. If we let D be the number of slopes allowed, then simple algebra shows that the decomposition should be used if I is expected to be larger than $O(N*D^2)$. This analysis ignores the number of pieces into which we must decompose the polygons since this number less than the size of the largest one, which we have assumed is bounded.

Chapter 4

Implementations

Section 4.1 - Introduction

As with any algorithms, theoretical descriptions of solutions such as those presented in the previous chapter have specific details which must be addressed before a running implementation of the algorithms is possible. This chapter is devoted to providing some of those details. It is divided into three parts, corresponding to the three versions of the algorithms presented in Chapter 3. The first and longest part discusses the overall approach as well as the solution to the general intersection problem. It should provide virtually all of the details (except the code) necessary to implement the algorithm described in Section 3.2. The second part discusses the details which are necessary to implement the algorithm for lists of disjoint polygons. The last section is devoted to the generalizations presented at the end of the third chapter.

Section 4.2 - Implementing the General Algorithm

When we introduced the general algorithm for detecting intersections among monotone polygons, we first described it as a two pass algorithm. The first pass was dedicated to finding all of the edge-edge intersections, and the second pass was dedicated to finding all of the containments. We then discussed how to merge these two passes into one.

From the point of view of the implementor, this is still a good strategy. Subdividing the problem will allow him to test smaller routines, develop separate portions of the program separately and obey the rules of structured design. Most importantly, since the first step simply involves running another known algorithm (Bentley and Ottman's), it becomes possible to use someone else's package. We will finish this section with a discussion about how to re-merge the passes, and it will be clear that the cost to the implementor is minimal.

We begin our study of the implementation by examining the components used. For the edge intersection pass, it is easiest to actually store the endpoints for the segments, yielding a data structure similar to the one below (described in the C programming language):

```
struct segment {
        int             sid;            /* Segment ID */
        float           XLeft;          /* Left endpoint  (X)  */
        float           YLeft;          /* Left endpoint  (Y)  */
        float           XRight;         /* Right endpoint (X)  */
        float           YRight;         /* Right endpoint (Y)  */
        struct plgn     *EdgeOf;        /* Pointer to polygon  */
};
```

The containment pass examines the hulls themselves and thus a slightly different

structure is used. We are primarily concerned with the upper and lower hulls as entities, but due to the nature of the algorithm, we will scan each hull left-to-right exactly once. This leads us to conclude that a linked list of the vertices of each half-hull is the appropriate data structure. Once again, a sample declaration is given below:

```
struct hullpoint {
        float           X;      /* X-coordinate of the point */
        float           Y;      /* Y-cooridnate of the point */
        struct hullpoint
                        *next;  /* Link to the next hull point */
};
```

To save space in the final implementation, the edge intersection pass may later be modified to examine segments which are pairs of hullpoints. There is only a small additional cost in this.

Regardless of the details of our component design, the most difficult coding problem is posed by the larger data structures. Since all of the sweeping-line algorithms discussed are event driven, it will be necessary for us to have some way to maintain a list which will a) allow insertions in O(logN) time, and b) return the minimum element quickly (O(logN) time is sufficient). Any standard priority queue algorithm will suffice here, though if the implementation is going to be using large amounts of virtual memory, then it is best to choose one which causes relatively few page faults. Heaps perform well in this repsect, and hence are a logical choice. Heap implementations may be found in many standard texts [Knu, Sed].

All of the algorithms discussed dynamically build up and break down large tree-like data structures. Bentley and Ottman use threaded binary trees and McCreight makes use of treaps. In both cases, if the structures are kept balanced, then it is easy to see that the depth of the tree (and

hence the running times of most of the operations) is O(logN). Unfortunately, there is no guarantee that this will be the case with random data. For both data structures, it is easy to construct a sequence of insertions which will cause the tree to have depth equal to the number of elements inserted (i.e. to make the tree into a chain). This is unacceptable from any point of view since it would cause the algorithm to run more slowly than even the naive solution. If the expected depth were O(logN) then perhaps this would be acceptable, but there is no reason to believe that typical polygonal inputs will have this property. Thus, we must balance our data structures.

Bentley and Ottman use simple binary search trees to hold the segments. Thus any balancing scheme which guarantees that the depth is O(logN) on search trees may be implemented. AVL trees have this property [Knu], and coding implementations are well known.

Since the treap will be based on a red-black 2-3-4 tree, it seems sensible to base the original tree of segments on such a structure as well. Sedgewick and Guibas [SedGui] have presented the actual code to do this, although the implementor should pay special attention to the parameter passing mechanisms as their code seems to have some errors. To modify their code to handle treaps one need only add a constant time check at each node to ensure that the secondary pairs are properly updated. This may involve up to one call to each of **DisposeSecondaryPair** or **ExtractSecondaryPair** as described in the algorithms presented in Section 2.4. This adds nothing to the asymptotic cost of either insertions or deletions. The search operation is the same except that both pairs must be checked at each node. This, too, causes no time penalty. The code for **DisposeSecondaryPair** and **ExtractSecondaryPair** may be found in McCreight's paper [McCr2], though the names there are **DisposeP** and **ExtractP**. The only changes necessary to these routines involve changing the "look-up"s of pair values to calls to the **EvaluateHullValue** routine. Also found in McCreight's paper are sample declarations for nodes and some discussion of

how to easily handle the bookkeeping problems which result from the inclusion of the secondary pairs.

To complete our discussion of the implementation we must describe how to merge the two passes of the algorithm. We note that the containment pass of the algorithm needs to maintain a list of events which includes one insertion and one deletion per polygon as well as all of the swapping events. Since there may be $O(N^2)$ of the latter, it might be necessary to have $O(N^2)$ storage. However, we may perform both passes at once. If we store all of the segment insertion, segment deletion, polygon insertion, and polygon deletion events in the list to begin with, then we may proceed in the obvious way, performing the requisite operations on each data structure as dictated by the events. We use Brown's improvement to guarantee that no more than O(N) segment swap events are inserted into the queue. Now the only change necessary is to the **HandleSegmentSwapEvent** routine. In addition to swapping the two segments, it must update the treap appropriately. The actions taken will depend upon the Type of the intersection, as described in Section 3.2. If, in addition, this routine is responsible for reporting the segment intersection, then we will have a list of the polygonal intersections in left to right order.

We conclude our discussion of the implementation of the general algorithm by noting that there are a few details which the implementor must still resolve. Depending upon whether the input polygons are assumed to be open or closed, the order for tiebreaking among insertion and deletion events must change. (If boundaries are to be included, then insertions should be done before deletions with swaps/searches intermediate; otherwise, things should be done in the opposite order.) In addition, care must be taken to include or exclude the endpoints of the segments depending upon the desired results.

Section 4.3 - Implementation for Sets of Disjoint Polygons

Whereas the description of the general algorithm hid all sorts of details from the implementor, the coding of the algorithm presented in section 3.3 could not be more straightforward. In this case, we need only keep track of segments, and the data structure involved is a simple binary search tree. The segment structure given in the previous section is more than sufficient for our needs here, and any tree implementation may be used. In this case, the search key will be the **y**-value of the segment at the time of insertion, and it will be compared against the current **y**-values of segments already in the tree. As before, it is possible to create very unbalanced trees and, thus, to guarantee optimal performance, the trees must be balanced. An AVL tree implementation will be sufficient for this purpose.

Unlike the previous algorithm, this one need not maintain hulls. Each segment has a field for the ID of the polygon to which it points, and "searching" is done only when the first segment of a polygon is encountered. If the two segments returned by **Locate** are from the same polygon then a containment is reported. Thus there are no changes to Mairson's original algorithm other than insertion of this check. (If we expand the input to include non-monotone polygons, then it is not sufficient to check that the segments are from the same polygon; one must also check for inclusion. However, if an additional bit is stored with each segment indicating whether the interior of the polygon is above or below that segment, then the check is still quite simple.)

Mairson's algorithm itself is straightforward to code, and thus, this one is as well.

Section 4.4 - Implementation for Almost-Isothetic Polygons

When we studied the intersections of similar right triangles, trapezoids and other objects whose edges were constrained to only a few slopes, we didn't really design a new algorithm. Rather, we noted that some of the work done by the general algorithm didn't need to be done in these special cases and that the time savings could be derived simply by not doing this work. We presented two techniques for extending the class of inputs for which we could construct optimal solutions to the intersection problem.

The first technique consisted of applying a linear transformation to the inputs. The implementation of this is quite straightforward and only involves multiplying segment endpoints by a 2x2 matrix.

The second technique involved using some larger number of treaps to hold the hull data. Once again, the adaptation of our original algorithm is straightforward. One need only add a "which treap" parameter to each of the access functions and then change the calling routines to cycle through the various treaps making the appropriate calls for each one.

In each of these cases, we then remove the code concerned with updating the treaps from **HandleSegmentSwapEvent** (probably only a call to **DeleteTreap** followed by a call to **InsertTreap**) and we will have a program which runs in optimal time.

Other researchers have devised ways to potentially speed up certain parts of most of these algorithms and these comprise the next chapter.

Chapter 5

Other Relevant Work

There are basically two types of results which affect this work: techniques to speed up either one or both of the passes of the algorithm, and algorithms for similar types of problems. We begin our study of other results with the former.

One of the embarassing problems with Bentley and Ottman's algorithm is that if the number of intersections is greater than $O(N^2/\log N)$, then the naive algorithm performs better than the sophisticated one. Thus, various researchers have attempted to improve this result. Chazelle [Cha] has reduced the time bound to $O(N*\log^2 N/\log\log N + I)$ using a data structure which he calls a **hammock**. This approach is useful for the edge intersection pass of the algorithm, but does not help to speed up the containment pass, and has the disadvantage that it is not possible to directly integrate it into the sweeping line method, thereby forcing us to increase the space requirements. Myers [Mye] has produced an algorithm which runs in optimal time for the average case, but is not an improvement in the worst case. Although it is easier to incorporate this result into ours than was the case with Chazelle's algorithm (simply due to the relative complexity of the data structures involved), it is still necessary to use the two pass approach with the incumbent time penalty. Myers' approach, however, would seem to lend itself more readily to generalization. Little work has been published on speeding up the containment pass of the algorithm.

Of the work which has been published on problems similar to this one, Swart's dissertation [Swa], is certainly the closest to this one with respect to the problem attacked and the solution techniques. Published as a technical report just after the algorithms of Chapter 3 were

developed, Swart's work makes slightly different assumptions about the input polygons. He admits a larger class of polygons by allowing all simple polygons. His first step, however, is to decompose these polygons into "chains" which effectively break them up into separate monotone polygons (in a manner not unlike the one we used to split trapezoids into rectangles and triangles). He defines a data structure which he calls a **multiple version tree** which he searches in much the same way that we search the treap. His overall time bounds are equivalent to ours as are his space bounds.

His approach is based more loosely on the sweeping line technique than ours is, but this is due to the additional constraints which he has put on the problem. In addition, he reports much more information about the structure of the intersections thereby adding a "hidden" cost to his complexity similar to the one which we sometimes incur by reporting intersections more than once.

Other results involve different generalizations of the initial problems. Guting and Wood [GutWoo] have presented a divide and conquer approach to the rectangle intersection problem, and Edelsbrunner [Ede1,Ede2] has shown how to generalize that problem to arbitrary dimensions (at a cost corresponding of $O(\log^d N)$ where d is the number of dimensions). Edelsbrunner [Ede3] has also studied the problem for **k-oriented objects** (objects which are the intersections of oriented half-planes) in higher dimension, but his results yield no direct improvement for the problem in two dimensions since there is an extra O(logN) factor involved.

The approach which our third set of algorithms takes, namely splitting the input set into subsets of easy-to-solve problems seems not to have been generalized, probably due to the fact that the costs quickly multiply to the point where even naive algorithms will run faster.

No work has been published related to our generalization of Mairson's algorithm.

Finally, an as yet unpublished paper by Edelsbrunner and Guibas [EdeGui] describes a way to sweep a curved line across a set of objects which may intersect. They report an $O(N^2)$ algorithm which may report $O(N^2)$ intersections, but not in sorted order.

Chapter 6

Conclusions

Intersection of geometric objects in the plane has applications in many fields including graphics and VLSI design. Previous algorithms worked on line segments and isothetic rectangles. This thesis includes algorithms which generalize the input set to monotone polygons or, under certain additional restrictions, to simple polygons.

The problem for line segments has been solved in general in O((N+I)*logN) time and in O(N*logN + I) time for certain restricted classes of input. Our algorithms maintain these time bounds. In addition, we have generalized each of the classes of line segments to a corresponding class of polygons and shown that most of the algorithms generalize as well. Some data structures (e.g. interval trees) had to be discarded, and some had to be modified (e.g. treaps), but the algorithms themselves use the same general techniques as their predecessors and run in the same amount of time. These results are summarized in the table below:

Class of Input	Time Bound
Vertically Convex Polygons of Bounded Edge Size	O((N+I)*logN)
2 Lists of Disjoint Polygons	O(N*logN + I)
K Lists of Disjoint Polygons	O(K*N*logN + I)
Similar Right Triangles	O(N*logN + I)
Polygons Whose Edges Are Constrained to One of D Slopes	$O(D^2*N*logN + I)$

Several problems remain open. The first of these relates to Chazelle's algorithm [Cha]. One of the many contributions of his result is that it presented an algorithm which detects intersections but does not necessarily have to "sort" the intersections. No such result has been proven for polygons although it would seem that one should exist. (Note that Edelsbrunner and Guibas' work [EdeGui] will not yield this result as their algorithms do not report the intersections in sorted order.)

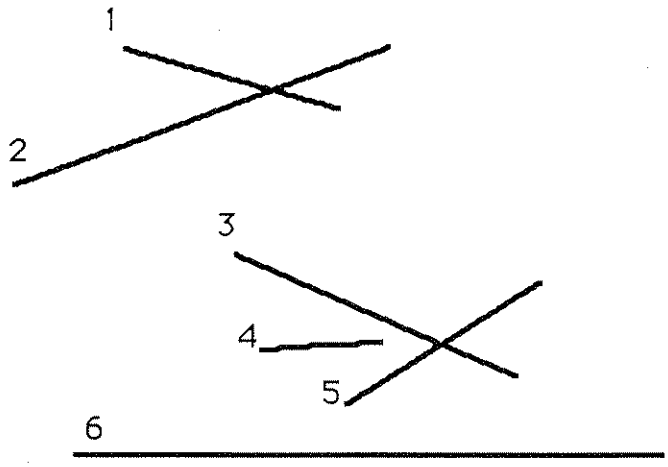
We have also modified McCreight's data structure in significant ways. Since the bounds on the pairs may now vary, questions arise concerning how to update the pairs. We showed in Chapter 4 that some types of "updates" do not change the treap, but others do. It remains an open problem to discover ways of performing these updates in better than O(logN) time.

Other, more obvious generalizations exist as well. One can ask for a list of intersecting polytopes in three (or more) dimensions, or perhaps just for a count of the number of intersecting pairs (in two or more dimensions). Lastly, one could ask for an overall time improvement to the general algorithm, or an enlargement of the class of problems for which we have optimal solutions.

Appendix

Worked Example #1 - Bentley and Ottman's Algorithm

This example is meant to highlight some of the features of Bentley and Ottman's algorithm for segment intersection. We note that Shamos' detection algorithm performs identically until it terminates part way through. We note this point of termination as well. The algorithm is being run on the input set below. At each step of the algorithm, we present the tree of segments intersecting the sweeping line (as a linear list). (Note that since no segment in our list has more than one intersection, we will not see any cases where Brown's improvement is used. Worked Example #3 has some such cases.)



Input Set for Bentley and Ottman's algorithm
Figure WE - 1

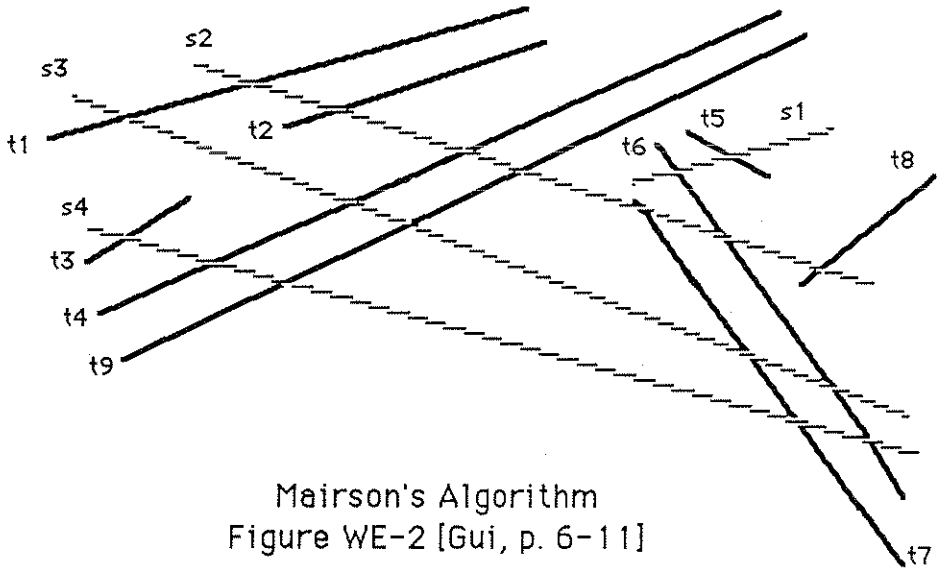
Action taken	Tree after action
Segment 2 inserted.	2
Segment 6 inserted.	
2 vs. 6 checked.	6-2
Segment 1 inserted.	
1 vs. 2 checked.	
(Intersection detected.)	6-2-1
(Shamos' algorithm stops here.)	
Segment 3 inserted.	
2 vs. 3 checked.	
3 vs. 6 checked.	6-3-2-1
Segment 4 inserted.	
3 vs. 4 checked.	
4 vs. 6 checked.	6-4-3-2-1
Segments 1, 2 swapped/reported.	
1 vs. 3 checked.	6-4-3-1-2
Segment 5 inserted.	
4 vs. 5 checked.	
5 vs. 6 checked.	6-5-4-3-1-2
Segment 1 deleted.	
2 vs. 3 checked.	6-5-4-3-2
Segment 4 deleted.	
3 vs. 5 checked.	
(Intersection detected.)	6-5-3-2
Segments 5, 3 swapped/reported.	
2 vs. 5 checked.	
3 vs. 6 checked.	6-3-5-2
Segment 2 deleted.	6-3-5
Segment 3 deleted.	

5 vs. 6 checked.	6-5
Segment 5 deleted.	6
Segment 6 deleted.	(empty tree)

Algorithm terminates.

Worked Example #2 - Mairson's Algorithm

Mairson's algorithm is a bit more complex than Bentley and Ottman's, so a more complicated input set is used. Once again, we will present the actions taken by the algorithm, but this time we will not present the two trees at each step of the computation. The sample data given below is taken from Guibas' lecture notes [Gui], but the explanation there contains many errors. They are fixed here.



Mairson's Algorithm
Figure WE-2 [Gui, p. 6-11]

Segments t_1, t_3, t_4, t_9, s_3, s_4 inserted.
Intersection t_3s_4 reported; Segment t_3 deleted.
Segment s_2 inserted.
Segment t_2 found in t_1-s_3 cone.

Intersection t_1s_3 reported; Segment s_3 split.

Segment t_2 inserted.

Segment s_1 found in s_2-t_4 cone.

Intersections s_2t_9, s_2t_4, s_2t_2, s_2t_1 reported; Segment s_2 split.

Intersections s_3t_9, s_3t_4 reported; Segment s_3 split.

Intersections s_4t_9, s_4t_4 reported; Segment s_4 split.

Segment s_1 inserted.

Segments t_1, t_2 deleted.

Segments t_6, t_7, t_5 inserted.

Intersections t_5s_1 reported; Segment t_5 deleted.

Segment t_8 found in t_6-s_2 cone.

Intersections t_6s_2, t_6s_1 reported; segment t_6 split.

Intersection t_7s_2 reported; segment t_7 split.

Segment t_8 inserted.

Segments t_4, t_9, s_1 deleted.

Intersection s_2t_8 reported; segment s_2 deleted.

Intersections s_3t_6, s_3t_7 reported; segment s_3 deleted.

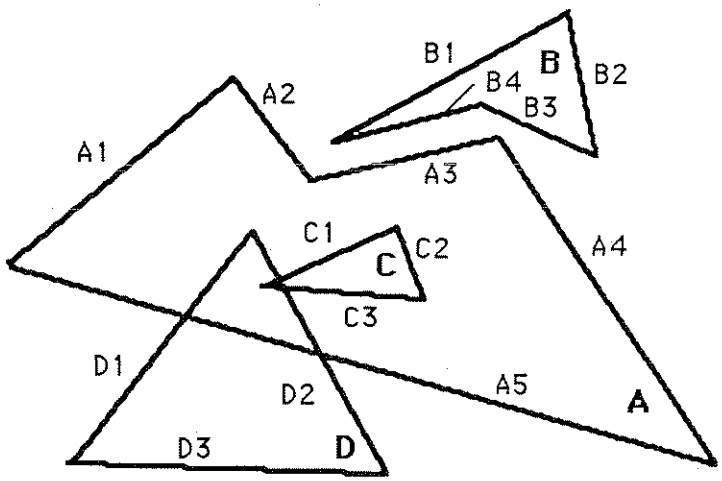
Intersections s_4t_6, s_4t_7 reported; segment s_4 deleted.

Segments t_6, t_7, t_8 deleted.

Algorithm terminates.

Worked Example #3 - Polygon Intersection Using Treaps

As was the case with Mairson's algorithm, we will not show the actual data structures at any given time, but rather we list the "events" that occur as we pass the sweeping line through our input set (given below). In addition, for this implementation, we will use Brown's improvement to the segment intersection algorithm which allows only one "swap event" to be stored for each edge. (Note: the term "segment tree" is used here to mean the tree of segments and should not be confused with the "interval/segment tree" of Section 2.4.)



General Polygonal Intersection Algorithm
Figure WE - 3

Determine that Polygon A doesn't overlap with anything already in the treap.
Insert Polygon A into the treap.
Insert Segment A5 into the segment tree.
Insert Segment A1 into the segment tree.
Determine that Polygon D doesn't overlap with anything already in the treap.

Insert Polygon D into the treap.

Insert Segment D3 into the segment tree.

Insert Segment D1 into the segment tree.

Insert Swap Event D1-A5 into queue.

Update the treap for the D1-A5 swap.

Delete Segment A1 from the segment tree.

Insert Segment A2 into the segment tree.

Swap Segments D1 and A5 in the segment tree.

Delete Segment D1 from the segment tree.

Insert Segment D2 into the segment tree.

Insert Swap Event D2-A5 into the queue.

Determine that Polygon C is contained in Polygons A and D. (Note that C/D is actually a false containment.)

Insert Polygon C into the treap.

Insert Segment C3 into the segment tree.

Remove Swap Event D2-A5 from the queue and insert Swap Event D2-C3 into the queue.

Insert Segment C1 into the segment tree.

Remove Swap Event D2-C3 from the queue and insert Swap Event D2-C1 into the queue.

Update the treap for the D2-C1 swap.

Swap Segments D2 and C1 in the segment tree.

Insert Swap Event D2-C3 into the segment tree.

Update the treap for the D2-C3 swap.

Swap Segments D2 and C3 in the segment tree.

Insert Swap Event D2-A5 into the queue.

Update the treap for the D2-A5 swap.

Swap Segments D2 and A5 in the treap.

Delete Segment A2 from the segment tree.

Insert Segment A3 into the segment tree.

Determine that Polygon B doesn't overlap with anything already in the treap.

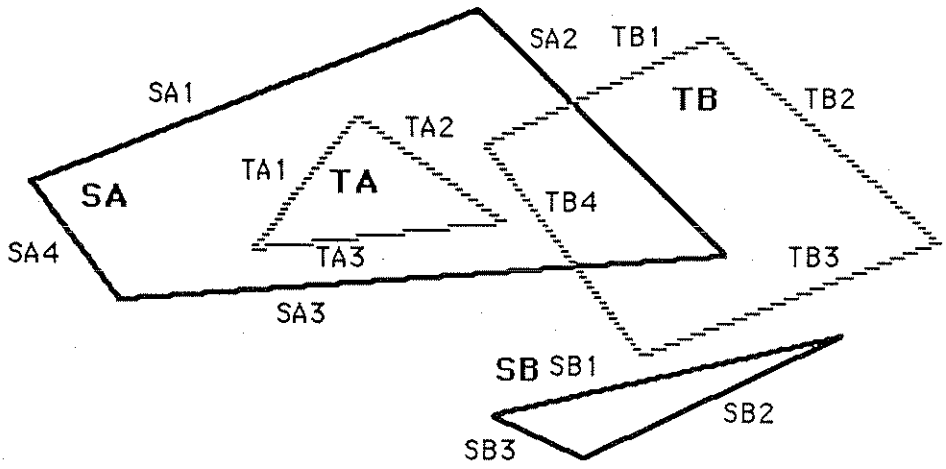
Insert Polygon B into the treap.

Insert Segment B4 into the segment tree.
Insert Segment B1 into the segment tree.
Delete Segment D3 from the segment tree.
Delete Segment D2 from the segment tree.
Delete Polygon D from the treap.
Delete Segment C1 from the segment tree.
Insert Segment C2 into the segment tree.
Delete Segment C3 from the segment tree.
Delete Segment C2 from the segment tree.
Delete Polygon C from the treap.
Delete Segment B4 from the segment tree.
Insert Segment B3 into the segment tree.
Delete Segment A3 from the segment tree.
Insert Segment A4 into the segment tree.
Delete Segment B1 from the segment tree.
Insert Segment B2 into the segment tree.
Delete Segment B3 from the segment tree.
Delete Segment B2 from the segment tree.
Delete Polygon B from the treap.
Delete Segment A5 from the segment tree.
Delete Segment A4 from the segment tree.
Delete Polygon A from the treap.

Algorithm terminates.

Worked Example #4 - Disjoint Lists of Polygons

Our fourth worked example shows the algorithm described in Section 3.3. Since shielding is broken in the same way as in Mairson's normal algorithm, the test data this time has been chosen in such a way as to avoid that. We simply list the actions of the algorithm on the data set below.



Intersection of Disjoint Lists of Polygons
Figure WE - 4

Insert Segment SA1 into the S tree.
Determine that the left endpoint of SA is not in any T-Polygon.
Insert Segment SA4 into the S tree.
Delete Segment SA4 from the S tree.
Insert Segment SA3 into the S tree.
Insert Segment TA1 into the T tree.
Determine that the left endpoint of TA is inside SA and report the intersection.

Insert Segment TA3 into the T tree.
Delete Segment TA1 from the T tree.
Insert Segment TA2 into the T tree.
Insert Segment SB1 into the S tree.
Determine that the left endpoint of SB is not in any T-Polygon.
Insert Segment SB3 into the S tree.
Delete Segment SA1 from the S tree.
Insert Segment SA2 into the S tree.
Insert Segment TB1 into the T tree.
Determine that the left endpoint of TB is inside SA and report the intersection. (Note that this is a "false containment".)
Insert Segment TB4 into the T tree.
Delete Segment TA2 from the T tree.
Delete Segment TA3 from the T tree.
Delete Segment SB3 from the S tree.
Insert Segment SB2 into the S tree.
Delete Segment TB4 from the T tree.
Discover the TB4-SA3 intersection.
Insert Segment TB3 into the T tree.
Delete Segment TB1 from the T tree.
Discover the TB1-SA2 intersection.
Insert Segment TB2 into the T tree.
Delete Segment SA2 from the S tree.
Delete Segment SA3 from the S tree.
Delete Segment SB1 from the S tree.
Delete Segment SB2 from the S tree.
Delete Segment TB2 from the T tree.
Delete Segment TB3 from the T tree.

Algorithm terminates.

Bibliography

[Bent] Bent, S., private communication.

[BenOtt] Bentley, J.L., and Ottman, T., Algorithms for Reporting and Counting Geometric Intersections, IEEE Transactions on Computers, C-28: 643-647, September1979.

[BenWoo] Bentley, J.L., and Wood, D., An Optimal Worst Case Algorithm for Computing Intersections of Rectangles, IEEE Transactions on Computers, C-29: 571-576, July 1980.

[Bro] Brown, K. Q., Comments on "Algorithms for Reporting and Counting Geometric Intersections, IEEE Transactions on Computers, C-30(2): 147-148, February 1981.

[Cha] Chazelle, B., Intersection is Easier than Sorting, Proceedings of the 16th Annual Symposium on the Theory of Computing, ACM SIGACT, pp. 125-134, 1984.

[DobLip] Dobkin, D.P., Lipton, R.J., and Reiss, S.P., Excursions into Geometry, Technical Report 71, Department of Computer Science, Yale University, 1976.

[Ede1] Edelsbrunner, H., A New Approach to Rectangle Intersections: Part 1, International Journal of Computer Math, 13: 209-219, May 1983.

[Ede2] Edelsbrunner, H., A New Approach to Rectangle Intersections: Part 2, International Journal of Computer Math, 13: 220-229, May 1983.

[Ede3] Edelsbrunner, H., Intersection Problems in Computational Geometry, Bericht F 93, June 1982.

[EdeGui] Edelsbrunner, H., and Guibas, L.J., Topologically Sweeping an Arrangement, Proceedings of the 18th Annual Symposium on the Theory of Computing, ACM SIGACT, to appear, 1986.

[Gui] Guibas, L.J., Lecture Notes for Stanford Course CS 445, 1982.

[GuiSed] Guibas, L.J., and Sedgewick, R., A Dichromatic Framework for Balanced Trees, Proceedings of the 19th Annual Symposium on the Foundations of Computer Science, IEEE Computer Society, pp. 8-21, 1978.

[GutWoo] Guting, R.H., and Wood, D., Finding Rectangle Intersections by Divide and Conquer, IEEE Transactions on Computers, C-33: 671-657, July, 1984.

[Knu] Knuth, D.E., The Art of Computer Programming, Volume 3: Sorting and Searching, Addison-Wesley, Reading, Mass., 1973.

[MaiSto] Mairson, H.G., and Stolfi, J., Reporting and Counting Line Segment Intersections, DEC Systems Research Center, Manuscript, 1983.

[McCr1] McCreight, E.M., Efficient Algorithms for Enumerating Intersecting Intervals and Rectangles, Technical Report CSL-80-9, Xerox PARC, June 1980.

[McCr2] McCreight, E.M., Priority Search Trees, SIAM Journal of Computing, Vol. 14, No. 2, pp. 257-274, May 1985.

[Mye] Myers, E.W., An O(E log E + I) Expected-Time Algorithm for the Planar Segment Intersection Problem, Technical Report TR 82-3, University of Arizona, June 1982.

[NiePre] Nievergelt, J., and Preperata, F.P., Plane-Sweep Algorithms for Intersecting Geometric Figures, Communications of the ACM, 25: 739-747, October1982.

[Sed] Sedgewick, R., Algorithms, Addison-Wesley, Reading, Mass., 1983.

[Sha] Shamos, M.I., Computational Geometry, Ph.D. Dissertation, Yale University, 1978.

[ShaHoe] Shamos, M.I., and Hoey, D., Geometric Intersection Problems, Proceedings of the 16th Annual Symposium on the Foundations of Computer Science, pp. 151-162, IEEE Computer Society, 1975.

[Swa] Swart, G.F., Efficient Algorithms for Computing Geometric Intersections, Ph.D. Dissertation, University of Washington, 1985.

[Tar] Tarjan, R.E., Updating a Balanced Search Tree in O(1) Rotations, Information Processing Letters, 16: 253-257, June 1983.