1-1-1989

Visualization of a plane sweep algorithm for construction of the visibility graph for robot path planning

Vikas B Patel
University of Nevada, Las Vegas

Order Number 1340593

Visualization of a plane sweep algorithm for construction of the visibility graph for robot path planning

Patel, Vikas B., M.S.

University of Nevada, Las Vegas, 1990

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

Visualization of a Plane Sweep Algorithm for Construction of the Visibility Graph for Robot Path Planning

by

Vikas B. Patel

A thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science

in

Computer Science

Computer Science Department
University of Nevada, Las Vegas
May 1990

Approval

The thesis of Vikas B. Patel for the degree of Master of Science in Computer Science is approved.

Chairperson: Yonina Cooper, Ph.D.

Examining Committee Member: Laxmi Gewali, Ph.D.

Examining Committee Member: Evangelos Yfantis, Ph.D.

Graduate Faculty Representative: Lon Spight, Ph.D.

Graduate Dean: Ronald Smith, Ph.D.

University of Nevada, Las Vegas

May 1990

Acknowledgements

I would like to thank my advisor, Dr. Yonina Cooper, for overseeing my work during the course of my graduate years. I am very fortunate to be working with someone of her caliber.

A special thanks to Dr. Laxmi Gewali for many interesting discussions during the implementation phase of the project. Other members of the faculty to whom I would like to extend this appreciation to include Dr. Kazem Taghva, Dr. Roy Ogawa, Dr. Evangelos Yfantis, and Dr. Tom Nartker. I am lucky to have been in contact with such an outstanding group of professionals.

Lastly, I would like to thank my family for their financial and emotional support; particularly my father, to whom I dedicate this thesis. He has been my sole inspiration throughout the course of my education.

This research has been supported by the Army Research Office under Contract # 330-4518-130.

Abstract

Research and development work in robotics and industrial automation has prompted a need for efficient motion planning algorithms for collision avoidance. To build fully autonomous robots, these algorithms must also model the environment correctly and accurately to safely maneuver the robot around obstacles. The main focus of this thesis is on the following problem and its solution: Given a set of obstacles represented as polygons in two-dimensional space, determine the shortest, collision-free path from the source point of the robot to some destination point. A fast and efficient algorithm for solving this problem is based on a plane-sweeping technique and runs in $O(N^2 \log N)$ time.

Since this solution has been studied very briefly in its theoretical form by [SS84], we present an in-depth analysis of the plane-sweep algorithm along with a full-scale implementation as well as an animation of the plane-sweeping technique.

Table of Contents

1. Introduction

One of the most fundamental problems in robotics and motion planning is that of finding optimal paths for robot navigation in a known environment. To design fully autonomous robots, very efficient algorithms are required to determine the shortest, collision-free path around obstacles. The two-dimensional version of this problem is easily solved in polynomial time ([LW79], [LP84], [SS84]); however, in three dimensions the problem becomes very difficult and no practical solution has been found that works in all cases. However, considerable effort has been made to solve many special cases of the problem when the environment is three-dimensional [WIRJ81], [Pap84], [ROH88], [ROIK87], [ROIK88].

"In the general 3-D case the problem is quite hard to solve, and is not even discrete; ..." [SS84]. The shortest path in this case between any two given points is a polygonal line whose vertices lie on the edges of the given polyhedral obstacles. Unfortunately, this is all we can deduce about the solution path. "Even if we knew the sequence of obstacle edges through which the desired shortest path passes, the calculation of these points of contact of the path with these edges requires solution of high-degree algebraic equations, which must be accomplished either by numerical approximate methods, or by precise, but very inefficient, symbolic algebraic calculations" [SS84]. Furthermore, the fastest known algorithms that solve these equations to produce the exact path are

doubly-exponential; making the implementation of such a procedure very impractical.

Papadimitriou [Pap84] presents an algorithm which uses a polynomial approximation scheme to find the shortest distance between two points in the presence of polyhedral obstacles. But in the same paper he also states: "Our algorithm is probably not fast enough to be immediately practical..." [Pap84]. Papadimitriou [Pap84] even questions whether the procedure to find the exact solution is NP-complete.

For the generalized two-dimensional version of this problem, the problem can be easily solved in polynomial time. Sharir and Schorr [SS84] very briefly outline the general approach for determining the solution path. The algorithm is based on a plane sweep of the polygonal obstacles in the environment. Although their paper was intended to present a solution for a special case in 3-D, they describe this algorithm in very little detail which makes the implementation of such an algorithm difficult.

The theoretical aspects for many special cases of the path planning problem in both two- and three-dimensions have been studied extensively in literature with little regard to practicality. Before we can devise an efficient (and practical) algorithm to solve the problem in three-dimensional space, the implementation details for the two-dimensional problem need to be fully understood. Studying the two-dimensional problem solution will provide insight into the three dimensional solution.

The goal of this thesis is to provide the reader with a sound understanding of the problem and the solution (for the two dimensional case) in terms of theory **and** practice. For this reason, the paper

contains an in-depth analysis of the plane-sweeping algorithm used to solve the problem and a full-scale implementation. Some of the details that were overlooked previously are also resolved [SS84]. To provide an in-depth understanding of the plane-sweeping technique, we have "animated" the algorithm to allow the user to visualize the method by which the solution is obtained.

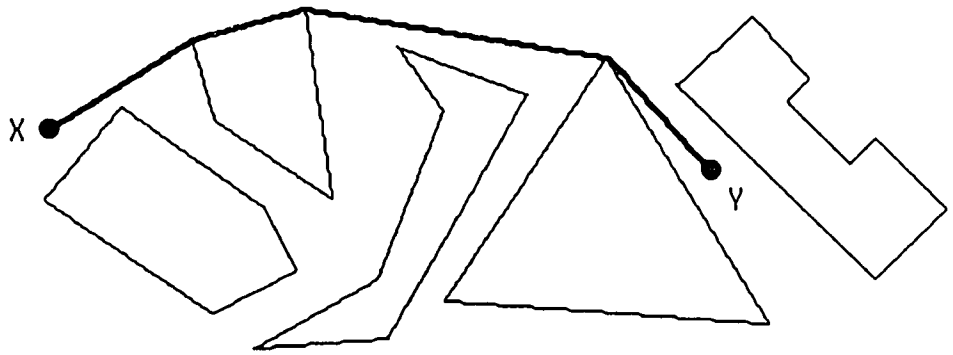


Figure 1.1. The shortest path from source X to destination Y

A more precise statement of the problem is now presented. We represent the solid obstacles in the environment as disjoint, simple polygons. Assume that the environment is predefined and static. Treating the robot as a single moving point, the problem is to find the shortest path from the current position, X, of the robot to some pre-determined destination point, Y. (See Figure 1.1)

Observe that the path is always a polygonal line whose vertices are the corners of the polygonal obstacles in the environment. Another, more interesting, property of the path is that the adjacent vertices of

this polyline are corners, including X and Y, which are visible to one another. These two properties are encapsulated in a data structure known as the *visibility graph* introduced by Lozano-Perez and Wesley [LW79]. The nodes in the visibility graph represent the corners (or points) of all the polygons in the environment while the edges represent the visibility relation which essentially connects nodes that are visible to one another. Each edge also has a weight, the Euclidean distance between the two corners.

Once the visibility graph has been generated, the shortest path from X to Y is obtained via Dijkstra's shortest path algorithm [AHU83]. Thus, the problem basically reduces to the construction of the visibility graph.

The organization of this paper is as follows. Chapter 2 describes basic path planning strategies in two dimensions. The plane-sweep algorithm is then presented in Chapter 3. The chapter includes a sample trace of the algorithm over a small environment. In-depth explanations of the algorithm are also presented in terms of its implementation and data structures. Chapter 4 describes how the algorithm is animated and what is required of the machine to perform such an animation. In Chapter 5 we analyze the time and space complexity of the plane-sweep algorithm. The paper then conclude in Chapter 6 with a discussion on our accomplishments and future work.

2. Path planning in two dimensions

2.1. Visibility

As noted earlier, the path from X to Y is always a polyline whose adjacent points are visible to one another. Visibility is defined:

Definition 2.1.1: Two points, u and v, are *visible* with respect to each other if and only if the line segment joining them does not lie completely within any polygon, nor intersects any other line segment belonging to the obstacles in the environment.

The word "intersect" in this definition is subject to a constraint which deserves some attention. (Refer to Figure 2.1.1.) When testing all line segments of the obstacles for intersection with the segment joining points u and v, line segments that are incident to either point u or v will be treated as non-intersecting. For example, in Figure 2.1.1(a) we know that points u and v would be visible to each other in such an environment. Although, line segments r and s are both incident to point v, we treat them as non-intersecting. That is, the intersection test is exclusive of the endpoints of the line segments. This observation is crucial in properly implementing the plane sweep algorithm.

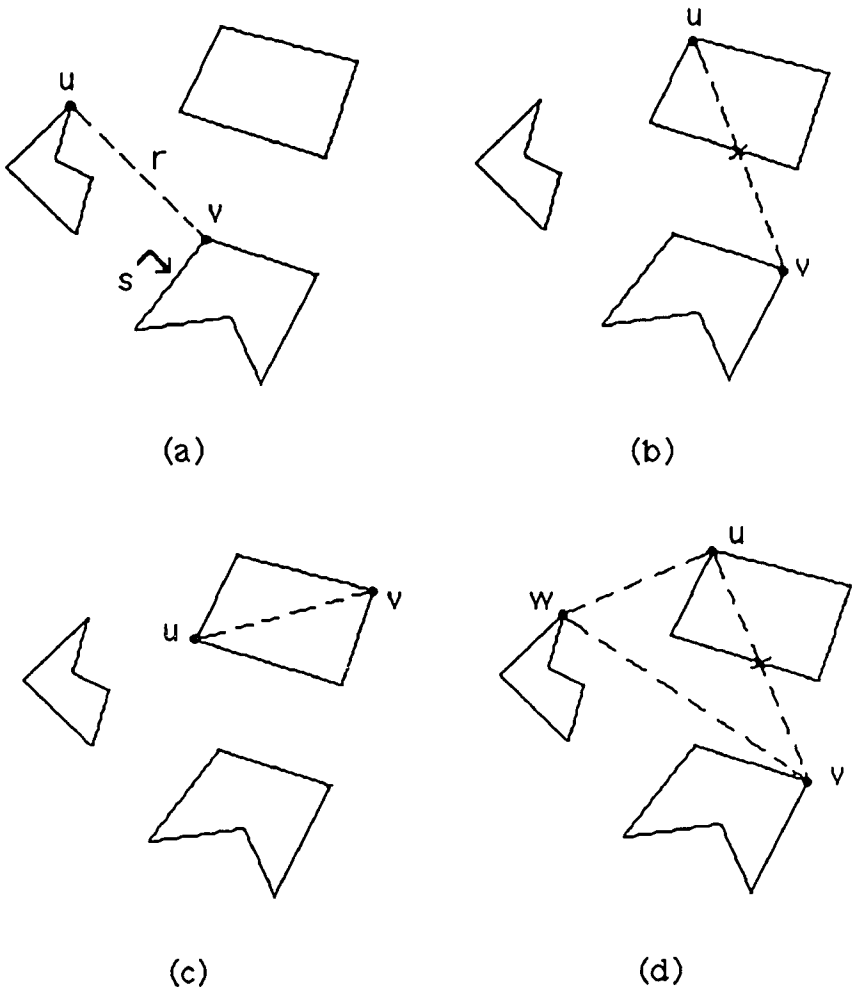


Figure 2.1.1. Visibility of points.

Figure 2.1.1(b) shows a case where two points, u and v, are not visible since one of the line segments in the environment clearly intersects the segment joining u and v. In Figure 2.1.1(c) these two points are not visible to one another since the line segment connecting them lies within a polygon. This can be easily detected by checking if u and v are vertices of the same polygon.

To aid the development of the algorithm, we define the visibility relation, Θ, for a set of points as follows:

Definition 2.1.2 : Two points, p and q, are related through the *visibility relation*, Θ, if and only if p and q are visible to one another. We denote this as $p\Theta q$.

It is interesting to note that Θ is a compatibility relation (one that is both reflexive and symmetric) which falls short of an equivalence relation (one that includes transitivity also). This observation can easily be deduced from Figure 2.1.1(d). In this example, $u\Theta w$ and $w\Theta v$ do not imply $u\Theta v$. There may be some instances where transitivity holds but not in general. This basically says that we cannot deduce all the information of the set of points in the environment by knowing how a smaller subset of points are related through the Θ relation. Each point must undergo some type of processing to determine which ones are visible to one another. To collect all the information concerning visibility in the environment, we use an undirected graph to model the visibility relation, Θ.

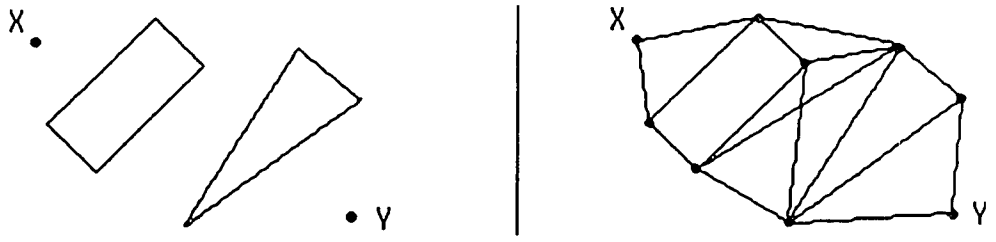
2.2. The Visibility Graph

Since the solution path is always along the visible corners (points), we use a graph to hold all the information concerning visibility of points in the environment.

Definition 2.2.1 : Let $V = C \cup \{X, Y\}$ be the node set where C is the set of all corners of the obstacles and X, Y are the source and destination points, respectively. Let the edge set be defined as $E = \{ \langle u, v \rangle \mid u \Theta v$, where $u,v \in V\}$. Each of the edges is weighted with the Euclidean distance between its endpoints. The corresponding undirected graph comprised of these two sets is called the *visibility graph* of the environment and is denoted VG(V, E).

VG(V, E) is a weighted, undirected graph. The nodes, V, represent the entire collection of points in the environment, including X and Y. The edges, E, are those connecting nodes in V which are visible to one another. Let S be the set of line segments belonging exclusively to the obstacles in the given environment. Note that S is a subset of E.

Figure 2.2.1 shows an example of a visibility graph for a simple environment consisting of two obstacles. In this example, the set S is comprised of all of the line segments appearing in Figure 2.2.1(a) while E contains all the line segments in the corresponding visibility graph of Figure 2.2.1(b). The edge weights of VG have been omitted for clarity.



(a) (b)

Figure 2.2.1. A sample environment (a) and its corresponding visibility graph (b).

The path planning problem is reduced to finding an algorithm for constructing the visibility graph, VG. After constructing VG, we apply Dijkstra's shortest path algorithm [AHU83] to find the shortest path from X to Y.

2.3. The Direct Solution (Brute-Force Algorithm)

Assume that the there are N elements in the segment array S, which implies that there are N+2 points in the array of points V, including the source, X, and destination, Y, points. To determine the set of points $\{q \in V \mid p\Theta q\}$ which are visible to any single point $p \in V$, we perform the following steps. For each of the remaining points, $q \in V$, draw a line segment from p to q and check to see if it intersects any of the line segments $s_k \in S$, for k = 1, 2, ..., N. The set S consists of all of the edges belonging to the obstacles. If no intersection is found, then $p\Theta q$

and edge <p, q> is added to VG along with its calculated weight. Edges are inserted into the graph only when this test holds. The graph is constructed by repeating this process for each of the N+2 points. (Refer to the pseudo-code in Figure 2.3.1)

The procedure is quite simple but very time consuming. Figure 2.3.1 allows us to determine the time complexity of this algorithm.

```
1.   for each p_i ∈ V do                                  {i = 0,1,...,N+1}
2.     for each q_j ∈ V and q_j ≠ p_i do begin            {j = 0,1,...,N+1}
3.         k := 1;
4.         visible := test_and_set(p_i, q_j);
5.         while (visible and (k <= N)) do begin
6.               if line segment s_k ∈ S intersects <p_i, q_j> then
7.                     visible := false;
8.               k := k + 1;
9.         end
10.        if (visible) then add_edge <p_i, q_j>;
11.    end
12.  end
```

Figure 2.3.1. A brute-force algorithm for constructing VG.

The two outermost loops (lines 1 and 2) are performed once for each of N+2 points in the node set. At line 4, a call is made to a boolean function 'test_and_set' with the current points, p_i and q_j, to initialize the variable 'visible'. The function returns FALSE if both the points lie on the same polygon and are not adjacent to one another. It returns TRUE otherwise. This is to detect the case when the line segment joining the two points lies within a polygon. (Recall Figure 2.1.1(c) in section 2.1.) The innermost while-loop (line 5) which performs the intersection test

of the segment joining p_i and q_j with all $s_k \in S$ is obviously the most time consuming, not only because of the nesting, but because of the floating point calculations required by the intersection test at line 6. Since the intersection test is made against each line segment $s_k \in S$, the time complexity of the algorithm is $O(n^3)$.

The complexity of the algorithm warrants the need for a faster algorithm which can minimize the number of intersection tests.

3. The Plane-sweep Algorithm

The next few sections present a more efficient algorithm which is based on a plane-sweeping technique. The algorithm significantly reduces the number of intersection tests required at each point.

Section 3.1 describes how the algorithm works in theory along with some related sub-problems. Section 3.2 traces a sweep-cycle of the algorithm on a sample environment. Section 3.3 presents the practical issues for implementing such an algorithm along with the pseudo-code outlining the key procedures. Finally, section 3.4 analyzes the time and space requirements.

3.1. General Idea

Recall that S is the set of all line segments comprising the polygons in the environment and V is the set of all distinct vertices derived from these polygons, including the points X and Y. Assume the cardinality of each of these sets to be $|S| = N$, $|V| = N+2$.

Now, for each point $v \in V$, we find all the points visible to v in the following manner. First, treat v as the origin of the coordinate system (refer to Figure 3.1.1). (No translation of axes is actually required in the algorithm.) Now sort the remaining N+1 points of V in increasing angular order in a counter-clockwise manner with respect to the positive x-axis and the origin v. This produces the sorted list, SP, with $|SP| = N+1$.

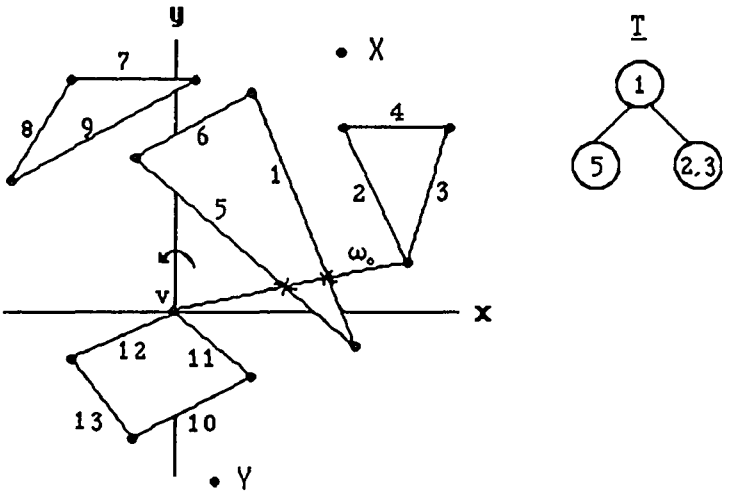


Figure 3.1.1. The initial instance of the sweep-line, ω_0, and the corresponding binary tree, T.

Next, we extend a ray from v to the first point $p_0 \in SP$. This ray defines the sweep-line rooted at v. The sweep-line will rotate counter-clockwise from the positive x-axis (Figure 3.1.1). For the initial instance of the sweep-line, ω_0, test all line segments $s_k \in S$ (k=1, 2,..., N) for intersection with ω_0. The test is exclusive of the endpoints of ω_0. For each s_k intersecting ω_0, we calculate the distance, d_k, to the point of intersection from v. We insert s_k into a binary tree, T, using this distance. (For a description of binary trees and their use, see [AHU83] and [Wir86].) This binary tree will either be empty or contain line segments that are sorted in increasing order with respect to the distances. Once these line segments have been inserted into T, we determine if p_0 is visible to v by checking if T is empty. If it is, then $p_0 \Theta v$ holds, and edge $<p_0,v>$ is added to the visibility graph, VG. Next add to T the segment(s) that are incident to p_0 and lie in the same

counter-clockwise, sweep direction, relative to the positive x-axis. The distance associated with these particular line segment(s) is from v to p_0.

Figure 3.1.1 shows the initial instance, ω_0, of the sweep-line and the corresponding binary tree, T. In this diagram, the nodes of T contain only line segment numbers with their distances omitted for clarity. Note that line segments 2 and 3 are both incident to p_0 and lie in the same sweep direction. They are both inserted into T with the same distance and therefore, occupy a single node. Segments 1 and 5 are also inserted into the tree since they intersect ω_0. The distance to the point of intersection for segment 5 is less than that for segment 1 which, in turn, is less than the distance to segments 2 and 3. Although the diagram has the distances omitted, recall that the ordering of these line segments is induced by their respective distances. The line segment with the smallest distance in T can always be found by traversing the tree recursively until the leftmost node is reached. We denote the line segment residing in this node as **minseg(T)**. In Figure 3.1.1, minseg(T) is the node containing segment 5.

The test for intersection against all N line segments of S is performed exactly once for the initial instance, ω_0, of the sweep-line. This is to initialize the binary tree so that it will properly reflect the status of the sweep at any given instance, ω_i.

To simplify the presentation of the plane-sweep algorithm, the following is needed:

Definition 3.1.1: At any instance of the sweep-line, ω_i, edges that own p_i and have their other endpoint yet to be encountered by the sweep-line are referred to as *active* edges. If both endpoints of an edge have been encountered during the sweep, then that edge becomes *inactive*.

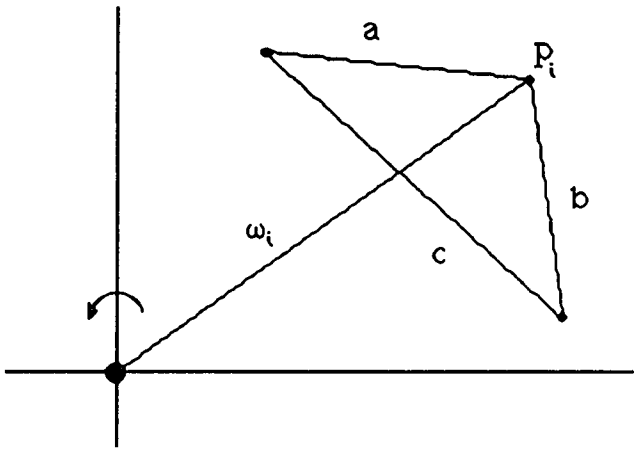


Figure 3.1.2. Active edge a and inactive edge b.

Figure 3.1.2 can be used to help clarify the concepts in definition 3.1.1. At this particular instance, ω_i, edge a becomes active. Edge b changes from active to inactive since it has been scanned completely by the sweep line.

Returning to the algorithm, the sweep cycle continues by rotating the sweep-line in a counter-clockwise fashion. This results from varying the endpoint (not the center) of the sweep-line by traversing the sorted list, SP, of points. At each instance ω_i (i=1,2,...,N-1), we insert those edges that become active into T and remove from T those edges

that become inactive. Edges that lie on the sweep-line itself are ignored in this updating process.

Next, we update the distance associated with minseg(T) by replacing it's current distance with the distance from the origin to the point of intersection between ω_i and minseg(T). Recall that minseg(T) is the line segment with the smallest distance in T. The updating of this particular line segment at each instance of the sweep-line is a very critical (and necessary) operation for the algorithm to work properly. This detail has also been overlooked in previous works [SS84], [LP84]. Before moving to the next instance, ω_{i+1}, test to see if minseg(T) has changed from the previous instance of the sweep-line, ω_{i-1}. Since minseg(T) is an element of S or undefined (if T is empty), this test can be performed quite easily by simply saving the previous minseg(T) and comparing it to the current one. If there is a change, then $p_i \Theta v$ holds and we add edge $<p_i,v>$ to VG. The only time edges are added to VG is when this test holds. We repeat this process of inserting, deleting, updating, and then testing at each of the remaining instances of the sweep cycle to find all points visible to v.

This entire procedure is repeated for each $v \in V$ to complete the construction of VG. To fully understand how this sweep mechanism yields a simple test for determining visibility, the next section traces this procedure on a sample environment.

3.2. Trace of the Sweep Cycle

The details of the plane-sweep algorithm can be clarified by tracing the algorithm on a sample environment. This section is devoted to a trace using the environment shown in Figure 3.1.1. One cycle of the sweep line will show the mechanics of the algorithm. A "snapshot" of the sweep-line at each point in SP during the cycle and the results of the corresponding binary tree, T, are shown.

In each of the diagrams, the lengths associated with the line segments in T are not exact and have been reasonably guessed for the sake of the trace. Since we are also interested in detecting a change in the minimal element, minseg(T), we have designated this element with a circle next to it in the tree.

Each diagram displays the status of T **after** all insertions and deletions of edges have taken place at that particular instance of the sweep-line. Furthermore, the updating of the length associated with minseg(T) has also been performed. So you may notice this length varying slightly from one snapshot to the next. Again, this length is only an approximation.

Figure 3.2.1 shows the initial instance of the sweep-line and illustrates the format for the rest of the snapshots in the trace.

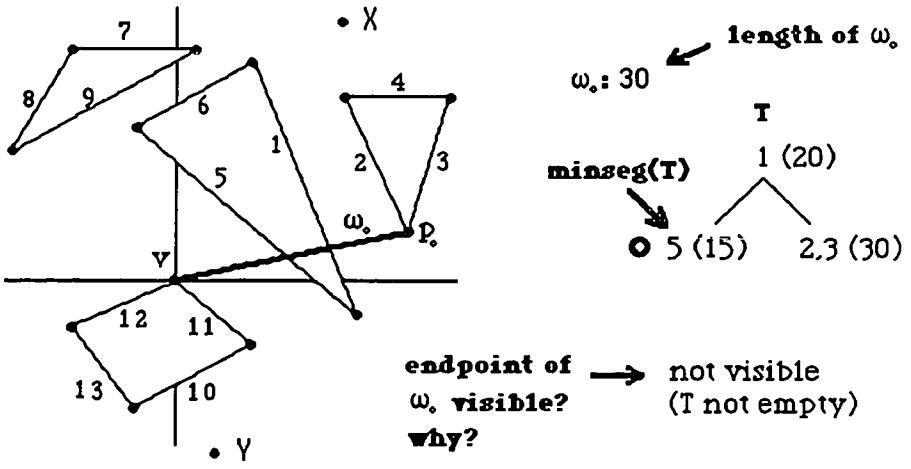


Figure 3.2.1. Initialization of T during instance, ω_0.

In Figure 3.2.1, the initial configuration of T is obtained by testing all $s \in S$ with ω_0 for intersections. This is the reason segments 1 and 5 were inserted into the tree. (Segment 5 enters with distance 15, and segment 1 enters with distance 20.) Since T is not empty, p_0 is not visible to v and vice-versa. Next, segments 2 and 3 are inserted (with distance 30) into T since they are active at this instance.

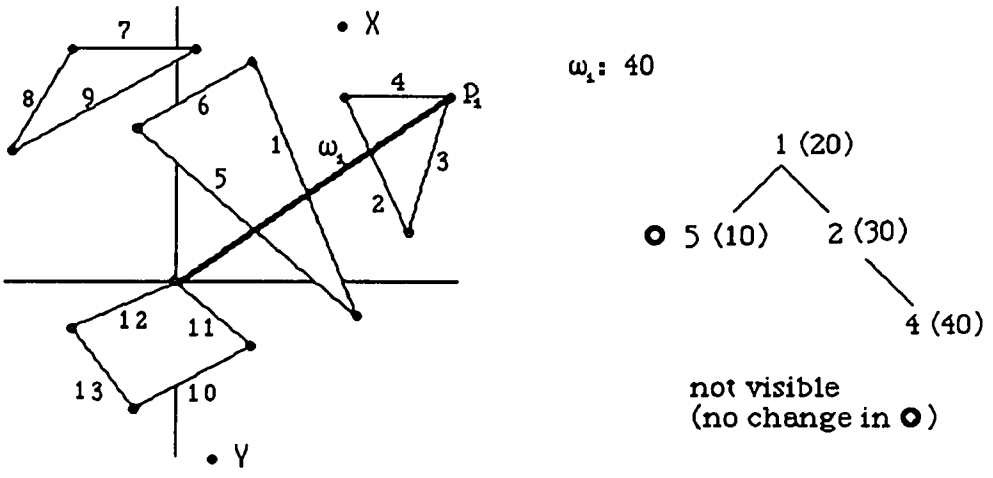


Figure 3.2.2. Instance ω_1.

The instance, ω_1, shown in Figure 3.2.2 shows that edge 3 becomes inactive and leaves T. Edge 4 becomes active and is inserted into T. The length of minseg(T), edge 5, is updated by finding the intersection point of edge 5 with ω_1. Since the ordering in the tree does not induce a change in minseg(T) with respect to the previous minseg(T), p_1 is not visible and we test the next point in SP.

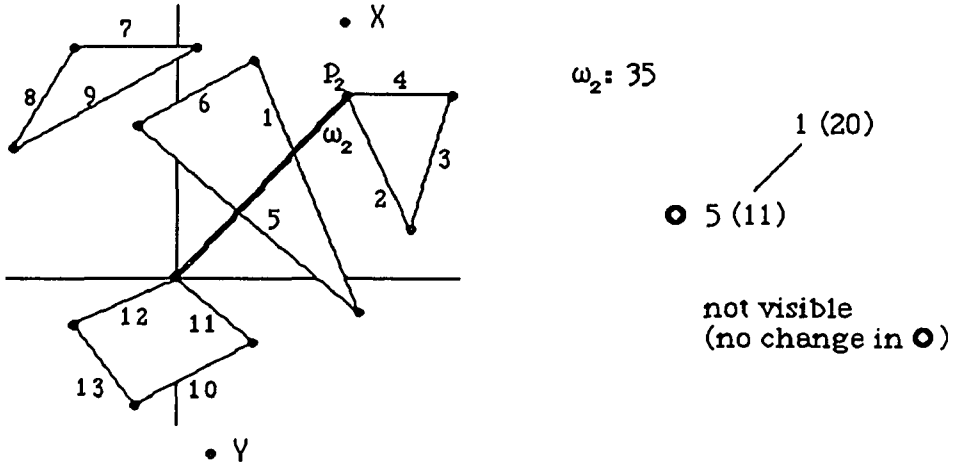


Figure 3.2.3. Instance ω_2.

In Figure 3.2.3, the tree changes slightly with the removal of edges 2 and 4 which have both become inactive. Minseg(T) is the same as for the previous point and therefore point p_2 is not visible.

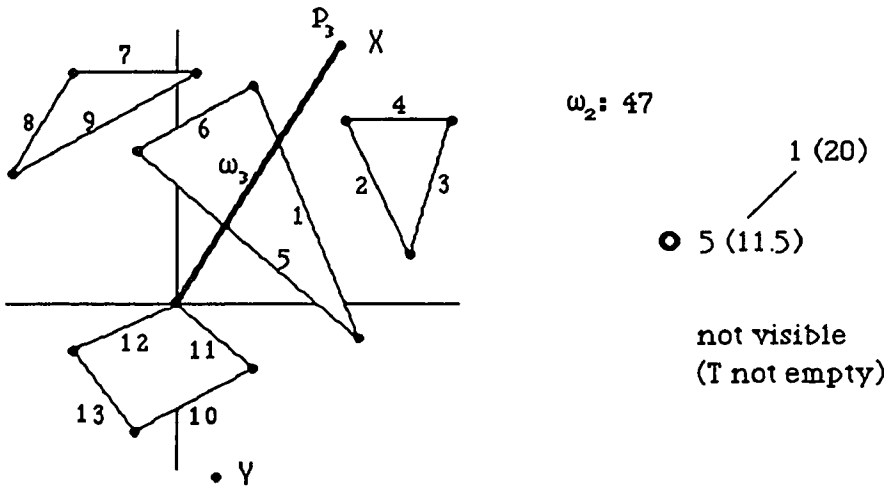


Figure 3.2.4. Instance ω_3.

The next instance, ω_3, in Figure 3.2.4, determines if point X will be visible to the origin. Since there are no incident edges to X, no insertions or deletions to T take place. The only way this point could be visible is if T were empty. In this case T is not, so the point is not visible. Minseg(T) remains the same and its distance is updated in the tree.

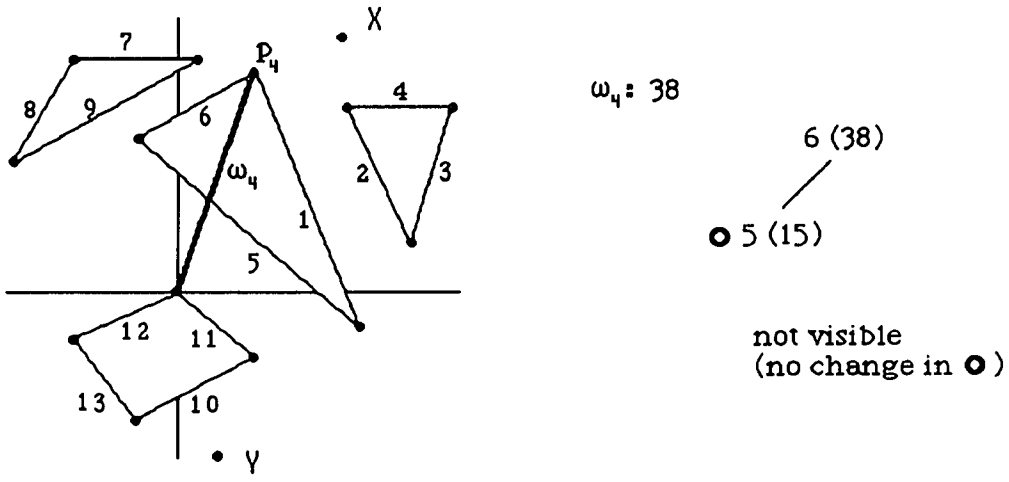


Figure 3.2.5. Instance ω_4.

Referring to instance ω_4 in Figure 3.2.5, edge 1 becomes inactive and is removed from T. Edge 6 is now active and inserted into T. There is no change in minseg(T) from the previous instance so point p_4 is not visible.

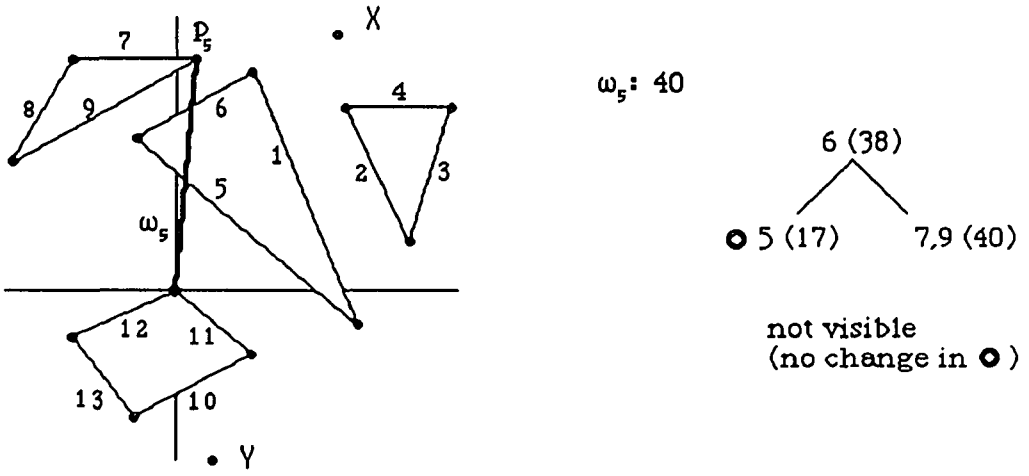


Figure 3.2.6. Instance ω_5.

Next, edges 7 and 9 become active (Figure 3.2.6) and enter the ordering when the sweep-line is passing p_5. This point is also not visible since line segment 5 has not left the ordering. The next transition is more enlightening as to how the algorithm determines visibility.

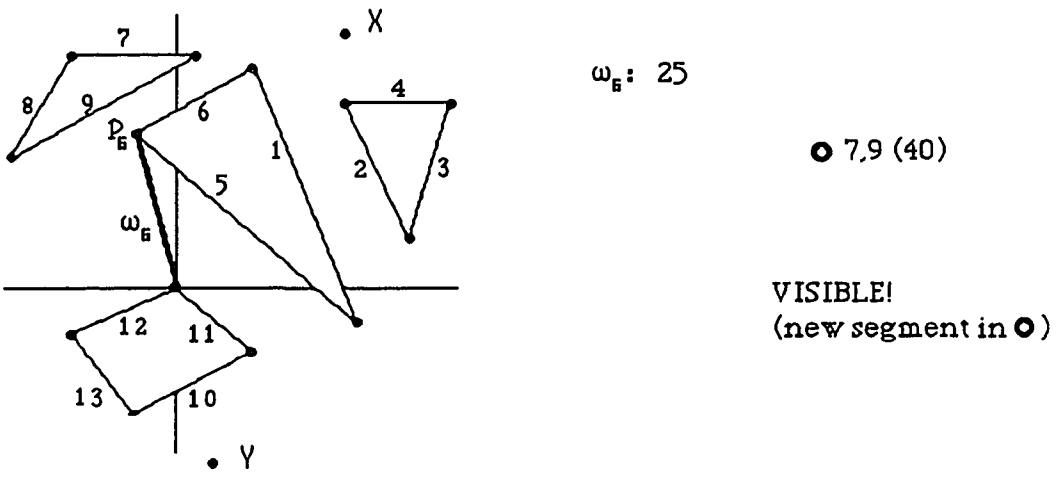


Figure 3.2.7. Instance ω_6.

In figure 3.2.7, edges 5 and 6 become inactive and are removed from T. This gives us a new minseg(T), edges 7 and 9, which means that point p_4 is visible. So the ordering of the line segments maintained by T essentially allows the algorithm to determine which points lie "behind" the closest edge to the origin. Since the edge that is closest to the origin always blocks the view of those points that lie on the opposite side of this edge, the algorithm can quickly determine if the next point in the sweep is visible or not. For our example, line segment 5 was the closest to the origin since the beginning of the sweep-cycle and thus, the points belonging to edges 1, 2, 3, and 4 were all hidden from this center (including X) because they were being obstructed by edge 5.. This was the reason for inspecting minseg(T) for changes. As soon as edge 5 left the ordering, point p_6 became visible (Figure 3.2.7).

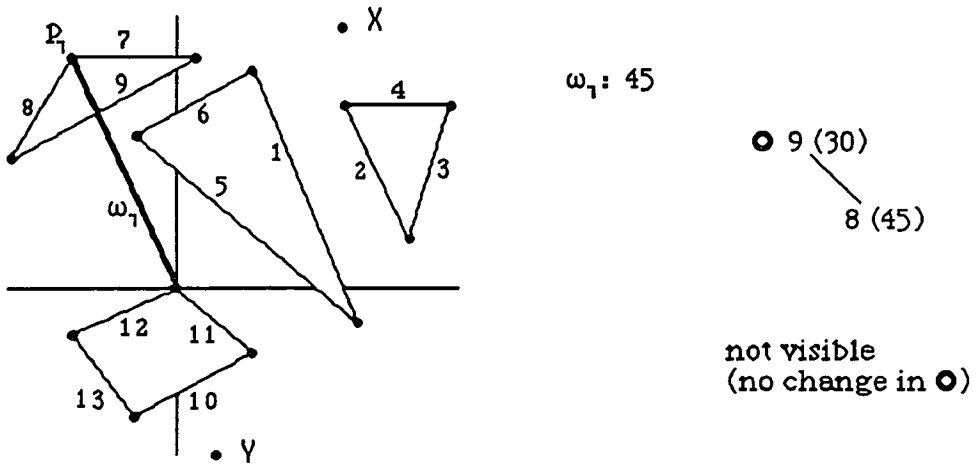


Figure 3.2.8. Instance ω_7.

For the sweep-line, ω_7 in Figure 3.2.8, edge 7 becomes inactive and leaves the ordering while 8 becomes active and is inserted into T. Point p_7 is not visible since segment 9 is still minseg(T).

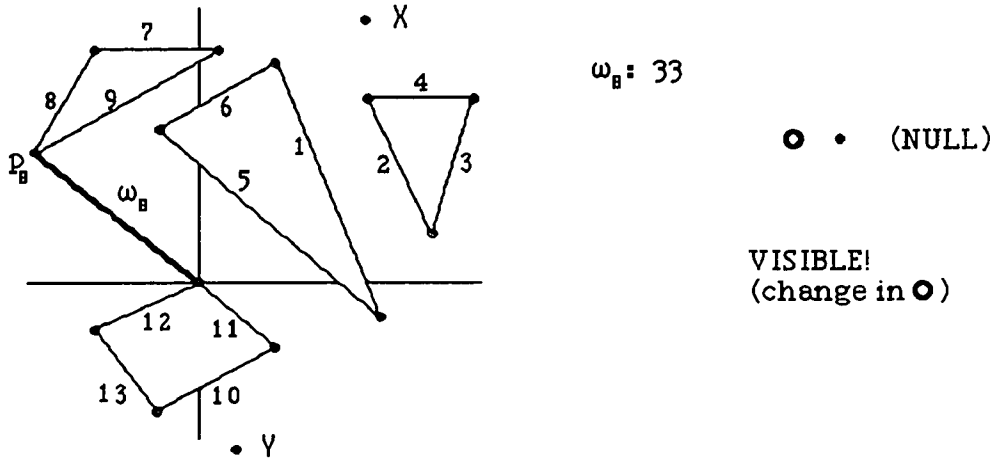


Figure 3.2.9. Instance ω_8.

When the sweep-line reaches point p_8 in Figure 3.2.9, T becomes empty as a result of edges 8 and 9 becoming inactive. This constitutes a change in minseg(T) from 9 in the previous diagram (Figure 3.2.8) to being undefined. Hence, point p_8 is visible to the origin.

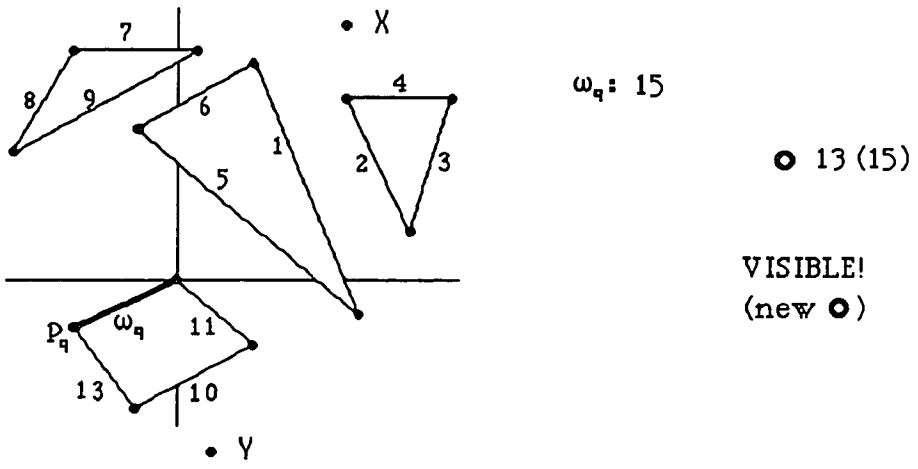


Figure 3.2.10. Instance ω_9.

For the instance in Figure 3.2.10, recall that edges that lie on the sweep line are ignored in this updating process of T. Here, edge 13 enters T and establishes a new minseg(T) compared to the undefined value in the previous instance (ω_8) when T was empty. Consequently, the point p_9 is visible.

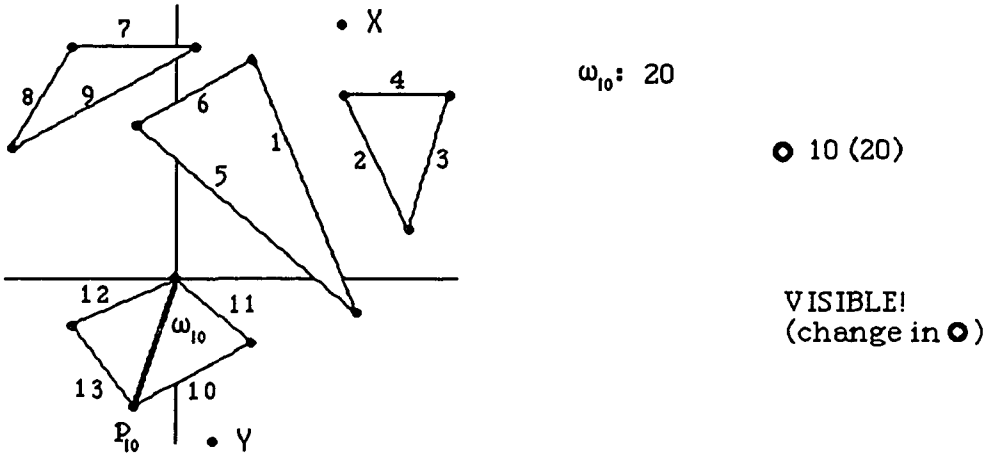


Figure 3.2.11. Instance ω_{10}.

This is an interesting case (Figure 3.2.11). Edge 13 is inactive and leaves T, while edge 10 becomes active and enters. This produces a new minseg(T) which says point p_{10} is visible. But at this instance, ω_{10}, the sweep-line is within a solid obstacle and point p_{10} cannot possibly be visible to the center! The implementation in the next section will account for this problem by setting a flag when the sweep-line is within an obstacle. Then, before we label an endpoint as being visible, we check this flag to see if it is set. Only then do we label the point as being visible.

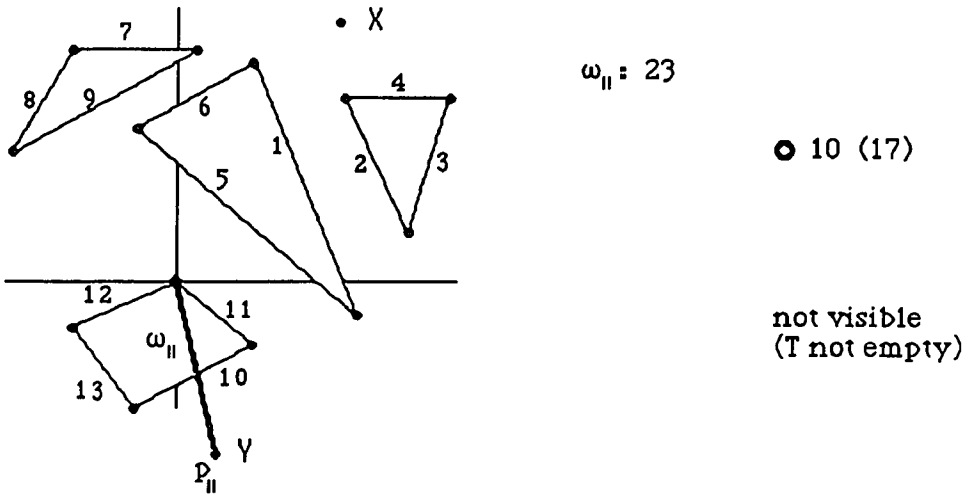


Figure 3.2.12. Instance ω_{11}.

The next instance of the sweep-line is ω_{11} (Figure 3.2.12). Since Y has no incident edges, no insertion or deletion takes place in T. For this point to be visible to the center, the binary tree, T, must be empty. In figure 3.2.12, Y is not visible since the tree is not empty. We update the distance of minseg(T) by calculating the point of intersection of ω_{11} with itself (edge 10).

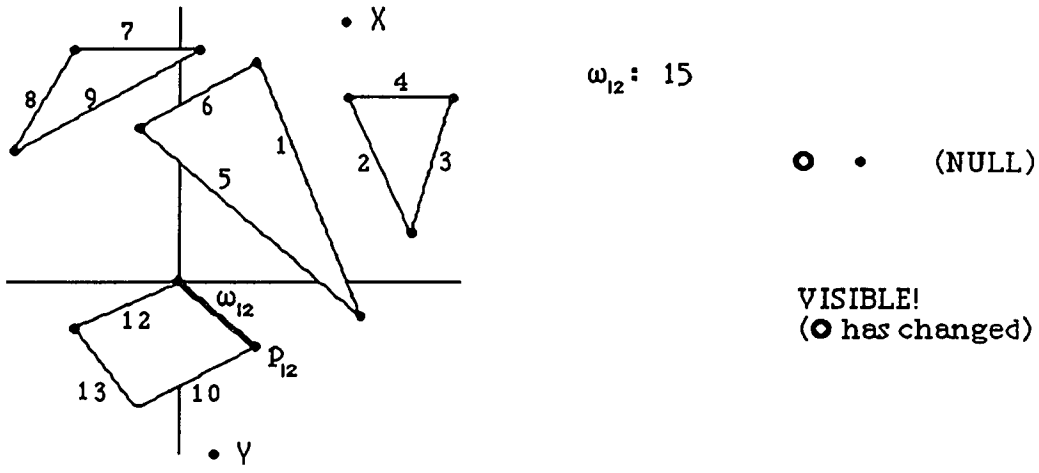


Figure 3.2.13. Instance ω_{12}.

In Figure 3.2.13, edge 10 becomes inactive and leaves the ordering. The tree becomes empty once again which constitutes a change in minseg(T) from the previous instance. Hence, point p_{12} is visible. And finally, we have the last transition:

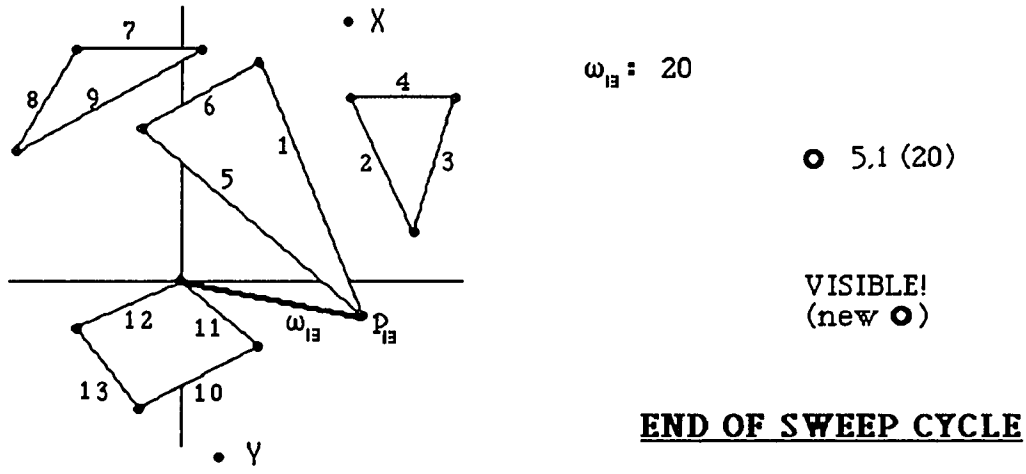


Figure 3.2.14. Instance ω_{13}.

In this last instance, edges 1 and 5 become active once again and enter T (Figure 3.2.14). This gives a new minseg(T) by which the point p_{13} becomes visible. Since this endpoint is the last one in the sorted list SP, the sweep cycle ends. All points visible to the origin have been found. We repeat this entire process for each of the remaining points by making them origins and applying a sweep-cycle. Recall that each time a visible point is found, an edge is added to VG. The visibility graph will be complete after a sweep-cycle is performed for each point $v \in V$. We then apply Dijkstra's shortest path algorithm [AHU83] on the visibility graph to obtain the solution.

3.3. Implementation

This section contains the actual algorithm and how it can be implemented. In Section 3.3.1 we describe how the environment can be modeled internally. Section 3.3.2 presents the required data structures and the procedure that initializes them. The support routines that are required by the plane-sweep algorithm appear in pseudo-code form in Section 3.3.3. The section concludes with the presentation of the plane-sweep algorithm itself in Section 3.3.4.

3.3.1. Internal Representation of the Environment

The first requirement for implementing this algorithm is to devise a format for describing the information contained in the environment. For this implementation, we placed this information in an input file with the format shown in Figure 3.3.1.1.

line 1:	a_1	b_1	*coordinates for point X*
line 2:	a_2	b_2	*coordinates for point Y*
line 3:	i_1		*number of points in polygon 1*
line 4:	x 1	y 1	*point 1 of polygon 1*
line 5:	x 2	y 2	*point 2 of polygon1*
	.		.
	.		.
	.		.
line $3+i_1$:	x_{i_1}	y_{i_1}	*point i_1 of polygon 1*
line $4+i_1$:	i_2		*number of points in polygon 2*
line $5+i_1$:	x 1	y 1	*point 1 of polygon 2*
line $6+i_1$:	x 2	y 2	*point 2 of polygon 2*
	.		.
	.		.
	.		.
line t:	x_{i_2}	y_{i_2}	*point i_2 of polygon 2*
line t+1:	i_3		*number of points in polygon 3*
line t+2	x 1	y 1	*point 1 of polygon 3*
	.		.
	.		.
	.		.

Figure 3.3.1.1. Format for the input file.

Lines 1 and 2 of this file contain the coordinates of source point, X, and destination point, Y, respectively. Beginning at line 3, each of the

polygons representing the obstacles in the environment are described as follows. First, the number of points in the polygon appears as a single integer on a line (e.g. line 3 in Figure 3.3.1.1). Next, if there are k points in the polygon, then on the next k lines contain the coordinates of these points. These k points must be listed in the order of occurrence as the polygon is traversed in a circular fashion. This also provides a means of representing line segments since adjacent points in the listing define these segments. The first and last point also constitute a line segment (similar to a circular list).

For example, in Figure 3.3.1.1 polygon 1 is defined beginning at line 3, which indicates the number of points in this polygon. All points belonging to this polygon are listed starting on the next line in the order of appearance by performing a circular traversal of the edges of polygon 1. As a result of doing this, points on lines 4 and 5 define a line segment of polygon 1. Points on lines 5 and 6 define another, and so on. The last point of this polygon (on line $3+i_1$) along with the first point (line 4) also constitute a line segment of this polygon.

Note that the format chosen for the input file will not contain any redundant points, except possibly if X or Y is a vertex of one of the polygons. If another format is chosen which allows redundant points, then the data structures presented in this section may not be applicable for that implementation. This implementation requires that all of the points be unique, with the exception of X and Y.

3.3.2 Data Structures

Once the input file has been created according to the format outlined in the previous section, the information contained in the file must be transferred into the computer using data structures that will preserve the relationship between points to segments and segments to polygons. For this reason, we separate the points and segments by using two structures which are linked together through pointers. All data structures presented in this section are global.

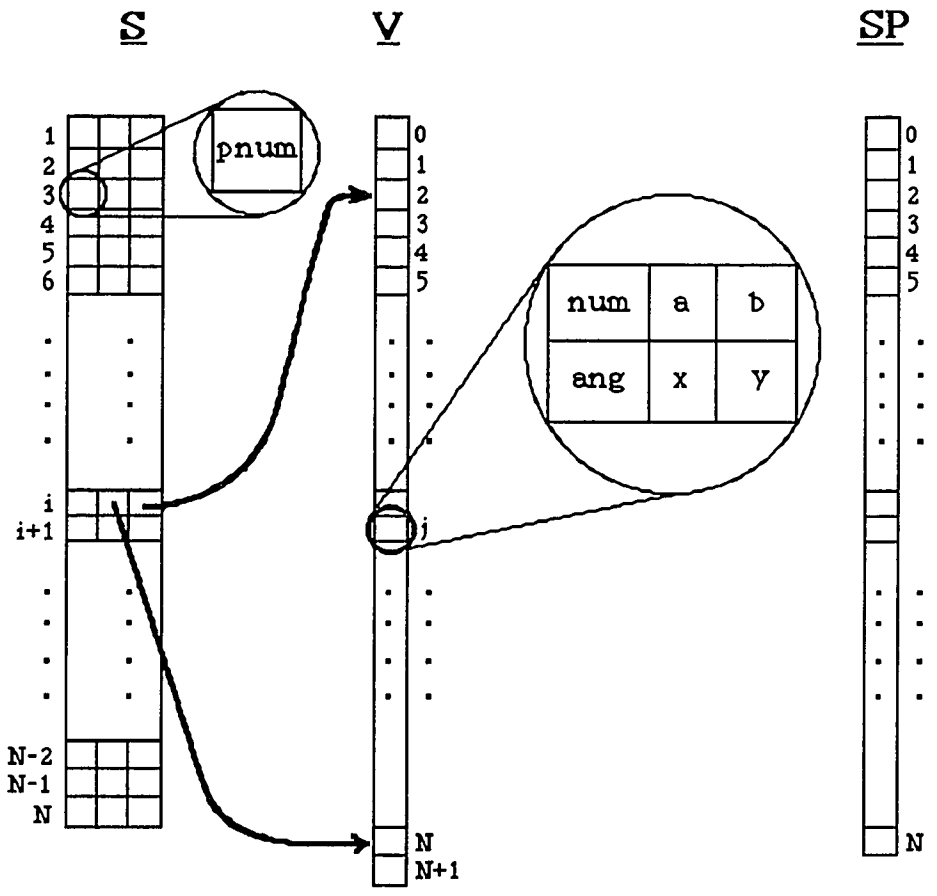


Figure 3.3.2.1. Data structures S, V, and SP.

The first data structure is an array of records, V, which is used to hold information concerning each of the points in the environment. (Refer to Figure 3.3.2.1.) Each element of the point array, V, is a record with 6 fields representing a single point. The *num* field in each element contains the array index of that element in V (e.g. the *num* field of the fourth element in V will contain 4). This labels each of the points in V. These labels will be used each time an edge is added to the visibility graph.

The *x* and *y* fields are the co-ordinates of the point. Recall that the coordinates are given in the input file and remain static throughout the entire construction phase of the visibility graph, VG.

The *ang* field is used to hold the measurement of the angle which results from the polar coordinates of the point. We refer to this angle measure as the *polar angle*. This field is used when the points need to be sorted in angular order as described Section 3.1.

Since each point of the polygons has exactly two incident line segments, fields *a* and *b* contain pointers (indices) to these line segments in the line segment array, S. This provides a means of determining which two line segments own a particular point. For the source and destination points, X and Y, respectively, these two fields are set to -1 if there are no adjacent segments, i.e. the point does not lie on any vertex of the polygons.

Before the beginning of each sweep cycle for the origin $o \in V$, we will need to sort the remaining points of V by their polar angles. For this purpose, we use a data structure, SP, which is identical to V except with one less element, the current origin. See Figure 3.3.2.1. Note that

both the array of points, V, and sorted array of points, SP, begin with index 0.

To represent the line segments belonging to the obstacles we use an array of records, S (Figure 3.3.2.1). This data structure, like V and SP, is also global. Each record of the segment array, S, consists of two pointers, each pointing to an element of V. These constitute the endpoints of the segment. An extra field, *pnum*, is used to store the number of the polygon to which this segment belongs. This field is set by keeping track of the current number of polygons that have been read from the input file.

Next, we require a binary tree, T, such as an AVL-tree or a 2-3 tree. Since binary trees are very common data structures in computer science (see [AHU83], [Wir86]), we use the standard implementation of an AVL tree presented in [Wir86]. Recall that the function of T is to maintain a sorted list of line segments with respect to their distances as the key. For this reason, each of the nodes for such a tree should also include the following fields:

dis	:	distance associated with this edge
seg_num	:	segment number (actually an index of S)

The *dis* field contains a real number which is the distance from the current origin to the point of intersection of the sweep-line and the line segment being scanned [Sections 3.1 and 3.2].

Recall from Sections 3.1 and 3.2 that each node contains a line segment. In each node of the binary tree, the field *seg_num* holds an

index to the segment array, S, which is the line segment contained in the node.

The last, major data structure required for the plane-sweep algorithm is the visibility graph, VG. This is an undirected graph and can be represented using either an adjacency list or an adjacency matrix. The implementation details can be found in any standard text on data structures ([AHU83], [Wir86]). Our implementation uses the adjacency matrix.

3.3.3 Support Routines

Before proceeding to the plane-sweep algorithm, we describe the support routines which are used by the algorithm.

First, the implementation for the intersection test is presented in Figures 3.3.3.1 and 3.3.3.2. This is the procedure given in [Sed88] and also described briefly in [PS85]. We have modified it slightly for our implementation to make the intersection test exclusive of the endpoints as stated in Section 2.1. The two segments, s1 and s2, to be tested for intersection are passed to the boolean function *intersect* (Figure 3.3.3.2), which in turn calls function *ccw* (Figure 3.3.3.1), to determine whether s1 and s2 intersect. The name of this function is an abbreviation for the word "counter-clockwise". The function *ccw* determines if the journey from point p0 to p2 via point p1 is a counter-clockwise turn or not. The function is three-valued (1, 0, -1). The value 0 indicates the three points are collinear. The other two values, 1 and -1, signify clockwise and counter-clockwise turns, respectively. See [Sed88] for additional details.

```
1.   function ccw(p0, p1, p2: point) : integer;
2.   var dx1, dx2, dy1, dy2;
3.   begin
4.        dx1 := p1.x-p0.x;  dy1 := p1.y-p0.y;
5.        dx2 := p2.x-p0.x;  dy2 := p2.y-p0.y;
6.        if (dx1*dy2 > dy1*dx2) then ccw := 1;
7.        if (dx1*dy2 < dy1*dx2) then ccw := -1;
8.        if (dx1*dy2 = dy1*dx2) then  begin
9.          if (dx1*dx2 < 0) or (dy1*dy2 < 0) then ccw := -1
10.         else if (dx1*dx1+dy1*dy1) > (dx2*dx2+dy2*dy2)
11.                 then ccw := 0
12.         else ccw := 1;
13.       end;
14.  end;
```

Figure 3.3.3.1. Function ccw.

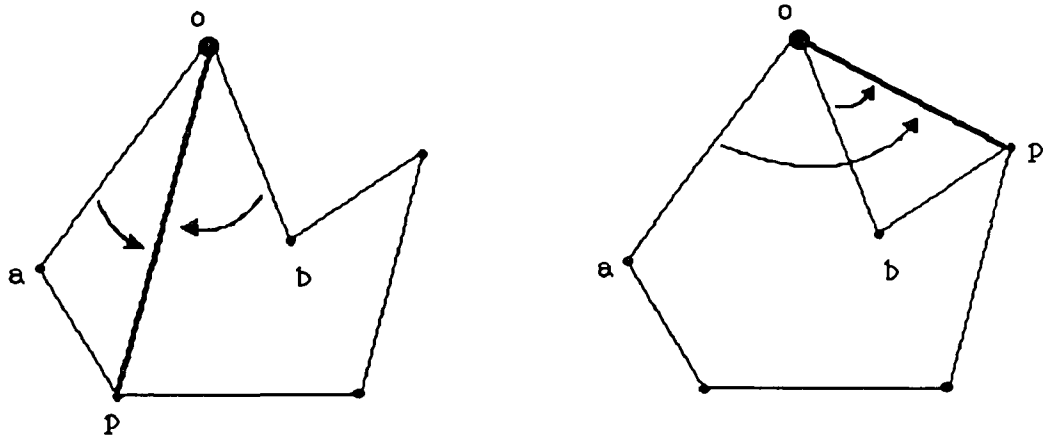
```
1.   function intersect (s1, s2 : segment) : boolean;
2.   var a, b, c, d : integer;
3.   begin
4.        a := ccw(s1.p1, s1.p2, s2.p1);
5.        b := ccw(s1.p1, s1.p2, s2.p2);
6.        c := ccw(s2.p1, s2.p2, s1.p1);
7.        d := ccw(s2.p1, s2.p2, s1.p2);
8.        intersect := (a*b <= 0) and (c*d <= 0);
9.   end;
```

Figure 3.3.3.2. The test for intersection.

The line segments, s1 and s2 intersect if both endpoints of each segment are on different "sides" (have different *ccw* values) of the

other. The function *ccw* also is also very useful in many other geometric intersection problems [Sed88].

The next major routine is a test for detecting when the initial instance, ω_0, of the sweep-line lies within the adjacent segments of its origin. (Recall cases a and b in Figure 2.1.1. of Section 2.1.) A global, boolean variable, *viewable*, is used by the plane-sweep algorithm to detect this event. We set *viewable* to TRUE if the sweep-line is found to be outside the adjacent segments of the origin and FALSE otherwise. Before explaining how the test is performed, Figure 3.3.3.3 illustrates why this test will work.



(a) (b)

Figure 3.3.3.3. Detecting the case when sweep-line is within a polygon.

Figure 3.3.3.3(a) shows the case where the sweep-line lies within the adjacent segments, *a* and *b*, of the origin, *o*. Similarly, Figure 3.3.3.3(b) illustrates the case when the sweep-line is outside these

segments. In both diagrams, the dark line represents the instance of the sweep-line. Note that in Figure 3.3.3.3(a) the directions (indicated by arrows) of ∠aop and ∠bop are opposite one another, while in Figure 3.3.3.3(b) both directions are counter-clockwise. If both of these angles open in the same direction, then we set the boolean variable *viewable* to TRUE since the sweep-line occurs outside the adjacent segments of the origin. Otherwise, we set viewable to FALSE. This test is performed by function *set_view* (Figure 3.3.3.4) via two calls to the function *ccw* . The function makes the test only for the initial instance of the sweep-line.

```
1.   function set_view (a, o, b: point) : boolean;
2.   begin
3.        if (ccw(a, o, SP[0]) == ccw(b, o, SP[0])) set_view := true
4.        else set_view := false;
5.   end;
```

Figure 3.3.3.4. Function *set_view*.

Finally, we require a procedure to create the list of sorted points of V in angular order. Recall that each instance of the sweep-line is produced by traversing this sorted list of points. In the previous section, we defined a global data structure, SP, where these points are to be stored in increasing order of the polar angles. These polar angles are determined by the standard mathematical transformation of the given Euclidean coordinates of each point. This is how the value of each *ang*

field is calculated. Once this field has been calculated for each point, we use the standard quicksort routine to sort the array of points, V. This procedure effectively sorts the points by using the *ang* field as the key. If more than one point has the same angle, then those points are further sorted by increasing distance from the origin.

At this point, we have the necessary tools to present the plane-sweep algorithm for constructing the visibility graph, VG.

3.3.4. Algorithm for Constructing VG

As explained in Section 3.1, the first step in the plane-sweep algorithm is to initialize the binary tree, T, for the initial instance, ω_0. This is performed by the *initial_set_up* procedure (Figure 3.3.4.1). The actual parameter for *s_line* is the initial instance, ω_0, of the sweep-line.

```
1.   procedure initial_set_up (s_line : segment);
2.   var  j : integer;
3.   begin
4.        T := nil;
5.        get_adj_pts  (s_line.origin);
6.        viewable := set_view(A, s_line.origin, B);
7.        for j := 1  to N do
8.             if (intersected(s_line, S[j])) then
9.                  insert segment S[j] into binary tree T;
10.       if (viewable) and (T = NIL) then
11.            add_to_VG (s_line.origin, SP[0]);
12.  end;
```

Figure 3.3.4.1. Initialization of T before a sweep cycle.

The sweep-line (*s_line*) is a record consisting of two points, the origin and the point SP[0], the first point in the sorted list SP. The two global variables, A and B, are initialized by the procedure *get_adj_pts* (line 5) [Appendix C, p 15]. These constitute the two points lying on the adjacent segments of the origin. Next, the function *set_view* initializes the global, boolean flag *viewable* (line 6). Then all line segments that intersect this initial instance of the sweep-line are determined (lines 7-8). These segments are then inserted into T (line 9). Next, if T is empty, and the sweep-line is not within the adjacent segments of the origin (e.g. *viewable* = TRUE), then the first point, SP[0], is visible to the current origin (line 10). In which case the edge is inserted into visibility graph, VG, (line 11). The initial set_up procedure (Figure 3.3.4.1) is used by the *build_VG* procedure (Figure 3.3.4.2).

```
1.    procedure build_VG ();
2.    var  u, v, prev_seg, curr_seg   : integer;
3.         sweepline : segment;
4.    begin
5.         for u := 0 to N+1 do begin
6.              angular_sort_around(V[u]);
7.              set sweepline to be (V[u], SP[0]);
8.              initial_set_up  (sweepline);
9.              for v := 1 to N do begin
10.                  prev_seg := minseg(T);
11.                  sweepline.endpt := SP[v];
12.                  if (SP[v] = A) or (SP[v] = B) then begin
13.                       add_to_VG (V[u], SP[v]);
14.                       viewable := not viewable;
15.                  end
16.                  else  begin
17.                       insert active edges at SP[v] into T;
18.                       delete inactive edges at SP[v] from T;
19.                  end;
20.                  update_minseg(sweepline,  minseg(T));
21.                  if (minseg(T) <> prev_seg) and (viewable)
22.                    then add_to_VG(V[u], SP[v]);
23.             end;
24.             free all nodes of T;
25.        end;
26.   end;
```

Figure 3.3.4.2. The plane-sweep algorithm, *build_VG*.

All structures used by procedure *build_VG* (Figure 3.3.4.2) are global. The procedure performs a sweep-cycle for each point in the array of points, V (line 5). For each of these points, the angular sorting

routine (line 6) creates the array of sorted points, SP. The list SP is sorted with respect to the current origin, V[u]. Then (line 7) the sweep-line is set to be the segment with endpoints SP[0] and V[u], which define the initial instance.. The sweep cycle for the instance is then initialized (line 8). This determines if the first point in the sorted list, SP, is visible to the origin, V[u], and initialize the binary tree, T, to contain all active edges at this initial instance.

The loop beginning at line 9 performs the sweep-cycle to determine the visibility of points in array SP with origin, V[u]. This inner-loop is used to traverse the sorted list of points while the outer-loop (line 5) varies the origin. Recall that the origin does not appear in SP and, since the visibility of the first point in this list has been determined by procedure *initial_set_up*, this inner loop is executed for the remaining N points, i.e., there are N remaining instances of the sweep-line in each cycle.

The current, minimal element of T is computed (line 10) and saved in the variable *prev_seg*. Recall from Section 3.3.2 that each node in T contains a field *seg_no* which holds the index to a segment in the segment array S. The *minseg* function [Appendix C, p. 13] returns that index of the node with the smallest value in the *dis* field.

Next (line 11) the sweep-line is advanced to the v^{th} instance by changing the endpoint field of *sweepline*. Then, this instance (lines 12-15) is checked to see if it is one of the adjacent segments of the origin, V[u]. If this is the case (line 12) the point SP[v] is visible to the origin. Also at this instance, we negate the global, boolean variable *viewable* to indicate that the sweep-line has scanned an adjacent segment of the

origin. This is to indicate when the sweep-line has changed states from being within a polygon to moving outside the polygon and visa-versa.

If the test for adjacency (line 12) fails, then the active segments are inserted (line 17) into the binary tree, T, and inactive segments are removed (line 18) from T. The insert and delete routines for T are dependent on which type of binary tree is being implemented, either AVL or 2-3 tree [Appendix C, pps. 9-13].

The *dis* field of the smallest element in T is updated (line 20). Recall from Section 3.1 that this field is to be replaced with the distance from the origin to the point of intersection of the sweep-line with segment minseg(T).

After updating this element in T, we test to see if the smallest element in T has changed (line 21). If there is a change during this instance of the sweep-line, then point SP[v] is visible to the origin, V[u], and the appropriate edge is added to the visibility graph, VG (line 22).

After execution of the inner loop (lines 9-23), the sweep cycle for origin V[u] has been performed and all points visible to the origin in the sorted list of points, SP, have been found and inserted into the visibility graph. All nodes of the binary tree are freed (line 24) before the outer-loop (lines 5-25) performs the next sweep-cycle for the next origin in V.

When the execution of procedure *build_VG* terminates, we have constructed the visibility graph, VG, and we can then obtain the shortest path via Dijkstra's algorithm. Since we wish to find the shortest distance between the source point, X, and destination point, Y, we use the implementation provided in [RND77] to obtain, not only the shortest distance between X and Y, but the actual sequence of vertices of the polygonal line of which the path is composed [Appendix C, p 15].

3.4. Time/Space Requirements

The time complexity for this algorithm is $O(N^2 \log N)$ which can be accounted for as follows. First, all of the support routines presented in Section 3.3.3 are performed in constant time, except for the angular sorting procedure. Since the fastest sorting time for any list of size N is O(Nlog N), we use a quicksort in the implementation.

Analysis of procedure *build_VG* in Section 3.3.4 (Figure 3.3.4.2) begins by observing that the outer-most loop beginning at line 5 takes O(N) time. Line 6 in this loop requires O(Nlog N) bringing the complexity of the algorithm up to $O(N^2 \log N)$. Line 7 is performed in constant time. At line 8 we call the *initial_set_up* procedure which requires O(N) time since N line segments are being tested for intersection in this procedure. See Figure 3.3.3.1. Since this occurs in the outer loop, the $O(N^2 \log N)$ time computed for this algorithm so far remains the same. The inner-most loop beginning at line 9 is performed O(N) times and also does not increase the complexity. Everything in this loop except the insert and delete calls to the binary tree (lines 17 and 18) is performed in constant time. Lines 17 and 18 each require O(log N) time since this is the time it takes to insert and delete nodes in a binary tree [AHU83]. Hence, lines 9-23 are performed in time O(Nlog N). Since these lines occur in the outer-loop along with the call to the sort routine (line 6), the algorithm requires $O(N^2 \log N)$ time. The 'log N' term is contributed to the complexity because of the need to sort the points in angular order (line 6) and the execution of the sweep-cycle (lines 9-23).

Upon completion of the procedure *build_VG*, we require calling Dijkstra's shortest path algorithm on the visibility graph to obtain the solution. This is performed in $O(N^2)$ time if an adjacency matrix is used to model the undirected graph, VG [AHU83].

The space requirements for this algorithm are as follows. Recall that the four major structures needed for the algorithm are: the array of segments S, the array of points V, the array of sorted points SP, a binary tree T, and an undirected graph VG (the visibility graph). Structures S, V, and SP all require O(N) space as shown in Figure 3.3.2.1 of Section 3.3.2. The binary tree, T, requires O(N) nodes since this is also the number of segments in the environment. The space required for VG is O(N) if an adjacency list is used to model the undirected graph and $O(N^2)$ space if it is modeled with an adjacency matrix [AHU83]. However, if one tries to save space by using the adjacency list, then the time complexity of the plane-sweep algorithm will be increased to $O(N^3)$ which is no better than the brute-force algorithm. This is due to the fact that Dijkstra's shortest path algorithm will have to be modified to retrieve elements from this list. The $O(N^2)$ time complexity for Dijkstra's algorithm is for graphs that are represented by an adjacency matrix which allows access to its elements in constant time. Since accessing a particular elements in the adjacency list requires O(N) time, the shortest path algorithm will require $O(N^3)$ time. For this reason, our implementation uses the adjacency matrix. The adjacency matrix is the largest structure required for the plane-sweep algorithm and hence, the space complexity for the overall solution is $O(N^2)$.

4. Animation

In order to better understand the plane sweep algorithm, the algorithm was animated so that the user could observe the execution of the algorithm. The environment is displayed as the execution begins. The initial instance of the sweep-line is also sketched into the environment. As the sweep-line moves through its cycle, active edges are highlighted through the use of color. The user observes the motion of the sweep-line through the sweep cycle for each point.

This section is devoted entirely to the visualization of the plane-sweep algorithm. Sections 4.1 and 4.2 explain the *animation* of this algorithm and how it is implemented. The code appearing in appendices include all of the animation features described herein. Our graphics-based implementation has been tested successfully on an IBM PS/2 (using TurboC) and the Silicon Graphics IRIS-4D workstation.

4.1 Description of the Visual Interface

The plane-sweep algorithm described in Sections 3.1 and 3.3 can be animated to provide a better understanding of its behavior. In other words, it is possible to modify the implementation so that we can observe the mechanics of the algorithm at each instance during execution, similar to the snapshots of the trace presented in Section 3.2. By treating each snapshot as a frame and displaying it on the screen in sequence, the process by which the solution is obtained can easily be understood. Another advantage of this type of animation is that it also

illustrates the basic idea behind plane-sweep algorithms in general, since they are used to solve many problems in computational geometry [HNS88], [NP82].

In animating the sequence of instances on the screen, we use color as the medium for identifying the various sets of information such as: the set of obstacles represented as polygons, the instance of the sweep-line, active and inactive edges, and the visible points that have been found during the current sweep cycle. All of these items must be accounted for during each animation frame. Color coding them is a means of identification, particularly for the active and inactive edges during the sweep cycle. If the polygons of the environment are displayed in a color, say GREEN, and the background of the display screen is BLACK, then we highlight the active edges in a different color, say RED, which indicates that those edges have been inserted in the binary tree. Since the line segments belonging to the polygons have been colored in GREEN, we need to distinguish the sweep line by yet a another color, say WHITE. When edges become inactive as a result of moving the sweep-line to the next instance, we restore their original color (GREEN) to signal that the edge is no longer in the tree. Since the background color is BLACK, we can erase the sweep line by redrawing it in BLACK for the same instance. In essence, the changes that take place from one snapshot to another are indicated by drawing in one color and redrawing (erasing) in another.

Although this visual interface is not needed by the robot executing the plane-sweep algorithm, it is of great use to humans by allowing us to monitor this execution from a remote site. We can

visualize not only the obstacles in the environment that surround the robot, but also the process by which it determines how to manuever around them.

4.2. Implementation of the Visual Interface

Before describing the routines required for the implementation, we analyze which parts of the plane-sweep algorithm are to be visualized on the display during the course of its execution. First, the static environment consisting of simple, disjoint polygons is drawn on the screen via a line segment drawing routine. The source and destination points (X, Y respectively) are also displayed. Next, each instance of the sweep-line is displayed temporarily as a line segment on the screen. At each instance we also highlight those segments that become active in a different color. (Recall that each of these items is color coded.) The original color of these segment is restored when it becomes inactive at some later instance. This will give the user an idea of which segments are currently in the binary tree, T. Since visibility of a single point is determined at each instance of the sweep-line also, we place large dot on that point if it is deemed visible to the origin. Finally, when the visibility graph has been constructed and the shortest path has been determined, we draw the solution path on the screen in yet another color.

Ultimately, the visualization of this algorithm produces a trace (similar to the one in Section 3.2) of the sweep-line for each origin with one exeception. Our implementation of the visual interface does not

support visualization of the binary tree due to its dynamic nature. Although the animation of the binary tree is not an issue in the application of the plane-sweep algorithm, it would be an excellent extension to this implementation in providing a more complete understanding of the plane-sweeping technique.

Since graphics routines vary from machine to machine, we need to make the code for the implementation as portable as possible. This is done by using a header file which contains the following procedures along with their machine dependent code:

```
procedure initialize_display ();
      { code to initialize the display to graphics mode
        and set up the screen coordinates appropriately
        so that all of the polygons can be seen on the
        screen. }

procedure color_x (c : color);
      { code to set the current screen color to be c  }


procedure line_x (p1, p2 : point);
      { code to draw a line segment between points
        p1 and p2 in the current screen color.  }


procedure text_x (x, y : real;  s : string);
      { code to print the string s on the graphics screen
        beginning at the coordinate (x, y).  }


procedure circle_x (x, y, r : real);

      { code to draw a filled circle on the screen in the
        current color at position (x, y) with radius r.  }
```

```
procedure clear_x ();
        {  code to clear the graphics screen to the current
           color.  }
```

Note that the machine-dependent code for each graphics primitive are basic routines that can be found in almost any machine that supports graphics. See header files for IBM PS/2 and IRIS-4D machines in the Appendices A and B.

Procedure *clear_x* is used to clear the graphics display before the environment is drawn. Once all of the polygons have been drawn on the screen, the animation sequence begins for each cycle as follows. First, the initial instance of the sweep-line needs to be drawn. We call procedure *line_x* from routine *intial_set_up* described earlier to draw the first instance on the screen. *Line_x* is also used to highlight the active edges. These active edges will already have been drawn on the screen in some given color; we redraw them in another color via *color_x* to indicate to the user that those segments have been inserted into the binary tree. Before going to the next instance, it may be necessary to include a delay at this point to allow the user to observe the situation being displayed. The need for the delay is machine dependent. If the machine is very slow in performing graphics-related I/O, then there is no need for the delay. However, if the machine is very fast, a delay will be needed from one instance to another. In our implementation, instead of using a delay, we wait for a keypress from the user before advancing to the next instance of the sweep-cycle.

When the sweep-line is to be advanced to the next instance, we erase the current instance which appears as a line segment on the

display, and then draw the next instance. The erasing of line segments can be performed by redrawing the segment in the color chosen for the background. This may distort the line segments that intersect that particular instance of the sweep-line. We simply redraw these segments since they can be determined by the array of segment, S, and the array of sorted points, SP.

At each instance when edges become inactive, we remove the highlighted color of these edges by redrawing them in their normal color via procedures *color_x* and *line_x*. Recall that edges that become active were highlighted in a different color to indicate to the user that they are currently in the binary tree. When the algorithm restores the inactive edges to their original color, this indicates that those edges have been removed from the tree during that instance. Since we determine visibility of a point at each instance also, we place a dot (filled circle) on that point if it is visible to the origin.

Then, before moving to the next sweep-cycle, we erase the screen and redraw the environment. The method outlined in the preceding paragraphs is performed for each instance of the sweep-line during each cycle so that the user can visualize the mechanics of this algorithm.

The animation is performed primarily in procedure *build_VG* described in section 3.3.4. Refer to the source code in our implementation for the placement of the actual calls to the graphics primitives are inserted [Appendix C, pps. 16-19].

5. Conclusion

An algorithm which runs in $O(N^2 \log N)$ time for solving the shortest path problem in two dimensional space has been presented. Although this may seem like only a slight improvement over the brute-force algorithm which runs in $O(N^3)$ time, the major advantage of this algorithm is that it eliminates a majority of floating point operations required in the brute-force method. Recall the support routine to perform the intersection test in Section 3.3.3 (Figure 3.3.3.1). Each call to this function executes anywhere between 8 and 12 multiplications of floating point variables. In the brute-force algorithm, this function is called N^3 times, whereas the plane-sweep algorithm makes N^2 calls. Practically, this turns out to be a tremendous improvement since no time is spent on unnecessary calculations. Hence, this algorithm will be much faster than the brute-force algorithm.

Another approach to increasing the speed of this particular algorithm is to exploit the parallelism inherent in its design. This is not surprising since many problems centered around geometric intersections possess this property [Mer86], [Qui87], [Goo87] The brute-force algorithm presented in Section 2.3 is no exception. The implementation of the plane sweep algorithm for both distributed and shared-memory systems is yet to be explored. We plan to extend this work for such an implementation.

Further work on this algorithm includes extending the representation of the environment to contain circles and not just polygons. One step further would be to devise a scheme using the

plane-sweep technique for solving the problem when the obstacles are represented as closed curves.

The problem of finding a faster algorithm than the plane-sweep algorithm is still open. In addition to being faster, it should also be feasible in terms of implementation.

The applications for the plane-sweep technique are numerous. For example, the same technique could be applied to the wire routing problem for designing circuit boards. Another application would be to incorporate it into a larger algorithm for determining the intersection of polyhedra in 3-space. The real challenge that remains is to extend this solution for the three-dimensional shortest path problem. The reason being that collision avoidance problems generally require the solution of standard computational geometry problems in high dimensional C-spaces whose vectors represent position and orientation information and perhaps other types of information about objects as well [Whi85]. There is no doubt that the plane-sweep algorithm is quite suitable for building a strong foundation for the generalized three-dimensional version of the problem. Although that problem remains unsolved, we can begin to formulate ideas on how to solve the problem from a bottom-up approach by finding efficient solutions to various sub-problems related to it, such as the plane-sweep algorithm.

Appendix A

(Source Code for IBM PS/2 Graphics)

```
/*
    file:  ps2driver.h

        This header file contains the graphics primitives for the IBM PS/2 Model 80
        running Borland TurboC 5.0.  These are the same routines described in the
        animation section of the thesis.
*/

#include <graphics.h>                   /*  include graphic's library       */
#include <conio.h>                      /*  include console i/o library     */

float rx, ry;

/*
        sc :  convert the real world coordinates (x, y) to screen coordinates
*/

sc (x, y)                                                                   sc
float *x, *y;
{
        *x = 320.0 + rx * (*x);
        *y = 240.0 - ry * (*y);
}


/*
        initialize_display: Set up the display screen for graphics mode.
*/

initialize_display ()                                                       initialize_display
{
        int g_driver, g_mode, g_error;

        detectgraph(&g_driver, &g_mode);
        if (g_driver < 0) {
                printf ("No graphics hardware detected!\n");
                exit(1);
        }
        initgraph(&g_driver, &g_mode, "");
        g_error = graphresult();
        if (g_error < 0) {
                printf ("initgraph error: %s.\n",grapherrormsg(g_error));
                exit(1);
        }
        setbkcolor (BLACK);
        clearviewport ();
        setlinestyle(SOLID_LINE, 0, NORM_WIDTH);
        rx = 320.0 / RIGHT;  ry = 240.0 / TOP;
}


clear_x ()              /*  simply erase the screen in the current color  */     clear_x
{
```

```
        clearviewport();
}

restore_graphics_display ()                    /* restore screen to normal */
{                                                                    restore_graphics_display
        getch();
        closegraph();                                                                    60
}


/*  color_x : Set the current color to i  */

color_x (i)                                                                      color_x
int;
{
        setcolor (i);
}                                                                                70


/*   line_x :  Draw a line from point (x1, y1) to (x2, y2) in the current color */

line_x (x1, y1, x2, y2)                                                          line_x
float x1, y1, x2, y2;
{
        sc (&x1, &y1);  sc (&x2, &y2);
        line (x1, y1, x2, y2);
}                                                                                80


/*   circle_x :  Draw a filled circle at position (x, y) with radius r in the current color */

circle_x (x, y, r)                                                               circle_x
float x, y, r;
{
        sc (&x, &y);
        r = r * 3;                                    /* scale the radius */
        fillellipse ((int)x, (int)y, (int)r, (int)r);                            90
}


/*  text_x : Print the text contained in string s begining at position (x, y)  */

text_x (x, y, s)                                                                 text_x
float x, y;
char *s;
{
        sc (&x, &y);                                                             100
        moveto ((int) x, (int) y);
        outtext (s);
}
```

Appendix B

(Source Code for IRIS-4D Graphics)

```
/*
    file:  iris_driver.h

        This header file contains the graphics primitives for the Silicon Graphics'
        IRIS-4D workstation.  These are the same routines described in the animation
        section of the thesis.
*/

                                                                              10
#include <gl.h>
#include <device.h>


/*
        initialize_display:     Set up the display screen to have dimensions
                                LEFT, RIGHT, TOP, and BOTTOM.  Initialize the
                                display for graphics mode.
*/
                                                                              20
initialize_display ()                                          initialize_display
{
        keepaspect (1, 1);
        winopen ("Plane Sweep");
        ortho2 (LEFT, RIGHT, BOTTOM, TOP);
        color (BLACK);
        clear ();
}

                                                                              30
clear_x ()                      /*  clear the graphics display  */             clear_x
{
        clear();
}

color_x (i)                     /*  set the current color to i  */             color_x
long i;
{
        color (i);
}                                                                             40


line_x (x1, y1, x2, y2)         /*  draw a line from (x1,y1) to (x2,y2)  */     line_x
float x1, y1, x2, y2;
{
        move2 (x1, y1);
        draw2 (x2, y2);
}

                                                                              50
circle_x (x, y, r)              /*  draw a filled circle at (x, y) with radius r  */   circle_x
float x, y, r;
```

```
{
        circf (x, y, r/2.0);
}


text_x (x, y, str)              /*  print text pointed to by 'str' at (x, y)  */        text_x
float x, y;
char *str;                                                                              60
{
        cmov2 (x, y);
        charstr(str);
}


getch()                         /*  get a click from left button on the mouse  */       getch
{
        while (!getbutton(LEFTMOUSE)) ;
        while (getbutton(LEFTMOUSE)) ;                                                  70
}
```

Appendix C

(C Source Code for Plane-Sweep Algorithm)

```
/*
        VISUALIZATION OF A PLANE SWEEP ALGORITHM FOR CONSTRUCTION OF THE
                VISIBILITY GRAPH FOR ROBOT PATH PLANNING

        This implementation is the plane-sweeping algorithm described in
        the thesis.  The program has been tested successfully on an IBM PS/2                10
        Model 80 and the Silicon Graphics IRIS-4D graphics workstation.

        Programmer      :  Vikas B. Patel
        Date            :  April 6, 1990
        Institution     :  University of Nevada, Las Vegas (Computer Science Dept.)

*/


#define MACHINE     0                       /*  0 = IRIS,  1 = IBM  */              20

/*
   The following constants are the paths to the
   header files containing the graphics routines.
*/

#define IBM_driver      "\tc\ps2drivr.h"
#define IRIS_driver     "/users/grad/vikas/path/iris_driver.h"

#include <ctype.h>                                                                  30
#include <math.h>
#include <stdio.h>

#define TRUE            1
#define FALSE           0

/*
   constants LEFT, RIGHT, TOP & BOTTOM are the
   maximum screen coordinates for drawing objects
*/                                                                                  40

#define LEFT            -50.0
#define RIGHT            50.0
#define TOP              50.0
#define BOTTOM          -50.0

#define MAXSEGMENTS 100                 /* dimension of segment list       */
#define MAXPTS      80                  /* dimension of point list         */
#define INFINITY    1000000
                                                                                    50

struct POINT {                  /*  record for each point in the environment */
        int num, a, b;
```

```
        float x, y, ang;
};


struct SEG_STRUCT {             /*  record for each line segment in the env. */
        int poly_num;
        struct POINT *l, *r;                                                    60
};


typedef struct NODE {
        float dis;              /*  distance to point of intersection       */
        int count;              /*  number of times entry occur             */
        int seg_no[3], bal;     /*  segment number, balance factor          */
        struct POINT p;         /*  point of intersection                   */
        struct NODE *l, *r;     /*  links to left and right subtrees        */
} TNODE, *TPTR;                                                                 70


/*
    GLOBAL VARIABLES:

        flag     :  boolean flag in inserting and deleting elements of the tree
        NPOLYS :  indicates the number of polygons in the environment
        NSEGS :        "     "    "    "  segments  "  "       "
        NPTS  :        "     "    "    "  points    "  "       "
        viewable:  boolean flag used to indicate when sweep-line is within polygon  80
        A, B     :  indices to segments that are adjacent to the current origin
        final    :  array of integers used in Dijkstra's shortest path algorithm
        label    :  array of reals used in Dijkstra's shortest path algorithm
        s_dist   :  array of distances for the edges that are currently in the binary tree
        VG       :  two-dimensional array (adjacency matrix) to model visibility graph
        origin   :  the current origin of the sweep cycle
        X, Y     :  the source and destination points, respectively
        I_PNT    :  the point of intersection of two line segments
        V        :  array of points in the environment
        SP       :  array of sorted points of a sweep cycle                     90
        S        :  array of line segments belonging to the obstacles
        T        :  the root of the binary tree  (AVL-tree)
      sweepline:  line segment representing the sweep-line
*/

int     flag, NPOLYS, NSEGS, NPTS, viewable, A, B, final[MAXPTS];
char    env_file[30];
float   s_dist[MAXSEGMENTS], VG[MAXPTS][MAXPTS], label[MAXPTS];
struct POINT origin, X, Y, I_PNT, V[2*MAXSEGMENTS], SP[2*MAXSEGMENTS];
struct SEG_STRUCT S[MAXSEGMENTS], sweepline;                                    100
struct NODE *T;

/*
   Include the appropriate graphics header file
*/
```

```
#if (MACHINE == 0)
#include IRIS_driver
#endif
#if (MACHINE == 1)                                                    110
#include IBM_driver
#endif

/*
   theta:        This function returns the angle made by the line segment
                 consisting of endpoints p1 and p2 with the x-axis.  The
                 angle begins from the positive x-axis.  Hence, the
                 range of this function is [0, 360).
*/
                                                                      120
float theta (p1,p2)
struct POINT *p1,*p2;
{
        float t, val, dx, dy;

        dx = p2->x - p1->x;  dy = p2->y - p1->y;
        if (dx == 0.0) {
           if (dy > 0.0) val = 90.0;
           else if (dy == 0.0) val = -1.0;
           else val = 270.0;                                          130
        }
        else if (dy == 0.0) {
           if (dx > 0.0) val = 0.0;
           else if (dx == 0.0) val = -1.0;
           else val = 180.0;
        }
        else {
           t = (180.0 / M_PI) * atan(dy / dx);
           if (t < 0.0) t = -t;
           if ((dy < 0.0) && (dx > 0.0)) val = 360.0 - t;             140
           if ((dy < 0.0) && (dx < 0.0)) val = 180.0 + t;
           if ((dy > 0.0) && (dx < 0.0)) val = 180.0 - t;
           if ((dy > 0.0) && (dx > 0.0)) val = t;
        }
        return (val);
}


/*
   min, max :  These two functions simply return the minimum          150
               or maximum value of its arguments.
*/

float min (a, b)          /* return the minimum of a and b */
float a,b;
{
        if (a < b) return (a); else return (b);
}
```

```
                                                                        160
float max (a,b)          /* return the maximum of a and b */
float a,b;
{
        if (a > b) return (a); else return (b);
}


/*
   len :  Function to return the Euclidean distance between
          the two points, p1 and p2.                                    170
*/

float len (p1, p2)
struct POINT p1, p2;
{
        return (sqrt((p1.x-p2.x)*(p1.x-p2.x) + (p1.y-p2.y)*(p1.y-p2.y)));
}


/*                                                                      180
   draw_env :    Procedure to draw the environment on the screen
                 based on the information contained in the array
                 of points, V, and the array of line segments, S.
*/

draw_env ()                                                  draw_env
{
        int i;
        char str[5];
                                                                        190
        color_x (BLACK);                        /*  background color        */
        clear_x ();
        color_x (BLUE);                         /*  color for coordinate axes */
        line_x (0.0, TOP-5.0, 0.0, BOTTOM+5.0); /*  draw Y-axis on screen    */
        line_x (LEFT+5.0, 0.0, RIGHT-5.0, 0.0); /*  draw X-axis on screen    */

        /*
           now draw all of the line segments belonging
           to the obstacles of the environment
        */                                                              200

        color_x (GREEN);
        for (i=0; i <= NSEGS; i++)
                line_x (S[i].l->x,S[i].l->y,S[i].r->x,S[i].r->y);

        color_x (YELLOW);                       /*  draw the source and     */
        circle_x (X.x, X.y, 2.0);               /*  destination points on the */
        circle_x (Y.x, Y.y, 2.0);               /*  screen and then label   */
        color_x (WHITE);                        /*  them appropriately      */
        text_x (X.x-1.5, X.y, "X");                                     210
        text_x (Y.x-1.5, Y.y, "Y");
}
```

```
/*
   set_up_environment:   This procedure reads in the information contained in
                         the input file and sets up the internal structures, S
                         and V, via calls to procedure 'add_segment'.  It then
                         calls 'draw_env' to display the information.
*/                                                                          220

set_up_environment ()                                          set_up_environment
{
        int i, j, first;
        FILE *infile;

        printf ("Enter environment filename: ");
        scanf ("%s",env_file);
        printf ("Reading %s...\n",env_file);
        infile = fopen (env_file, "r");                                     230
        rewind (infile);
        NSEGS = 0;  NPTS = -1; NPOLYS = -1;
        fscanf (infile, "%f %f", &X.x, &X.y);                  /* read point X */
        fscanf (infile, "%f %f", &Y.x, &Y.y);                  /* read point Y */

        while (fscanf(infile, "%d", &j) != EOF) {
          NPOLYS++;
          for (i=0; i < j; i++) {
            NPTS++;
            fscanf(infile, "%f %f", &V[NPTS].x, &V[NPTS].y);                240
            V[NPTS].num = NPTS;  V[NPTS].a = NSEGS;  V[NPTS].b = NSEGS+1;
            if (i==0) {
                first = NPTS;
                V[NPTS].b = NSEGS+j-1;
            }
            if (i>0) {
                S[NSEGS].l = &V[NPTS-1];  S[NSEGS].r = &V[NPTS];
                S[NSEGS].poly_num = NPOLYS;
                NSEGS++;
            }                                                               250
          }
          S[NSEGS].l = &V[first];  S[NSEGS].r = &V[NPTS];
          S[NSEGS].poly_num = NPOLYS;
          NSEGS++;
        }
        NSEGS--;
        fclose (infile);

        V[++NPTS] = X;  V[NPTS].num = NPTS;  V[NPTS].a = -1;  V[NPTS].b = -1;
        X = V[NPTS];                                                        260
        V[++NPTS] = Y;  V[NPTS].num = NPTS;  V[NPTS].a = -1;  V[NPTS].b = -1;
        Y = V[NPTS];
}
```

```
swap (a, b)                          /*  swap points a and b  */                    swap
struct POINT *a, *b;
{
        struct POINT t;
                                                                                    270
        t = *a;  *a = *b;  *b = t;
}


/*
   quicksort :  This is a standard quicksort which sorts the list l from index
                'low' to 'high'.  If the flag 'dis_key' is FALSE, then the sort-
                ing takes place with respect to the angles associated with each
                point.  Otherwise, the sorting takes place with respect to the
                distance to that point from the origin.                          280

*/

quicksort (l, low, high, dis_key)                                                quicksort
struct POINT l[];
int low, high, dis_key;
{
        int i, lastlow;
        float t;
                                                                                    290
        if (low < high) {
                swap (&l[low], &l[(int)((low+high)/2)]);
                if (dis_key==TRUE) t = len(origin, l[low]);
                else t = l[low].ang;
                lastlow = low;
                for (i=low+1; i<=high; i++) {
                   if (dis_key==TRUE) {
                        if (len(origin,l[i]) < t) {
                                lastlow++;
                                swap(&l[lastlow], &l[i]);                           300
                        }
                   }
                   else if (l[i].ang < t) {
                        lastlow++;
                        swap(&l[lastlow], &l[i]);
                   }
                }
                swap(&l[low], &l[lastlow]);
                quicksort(l, low, lastlow-1, dis_key);
                quicksort(l, lastlow+1, high, dis_key);                             310
        }
}


/*
   angular_sort:        This procedure is the driver to the quicksort
                        routine (selection_sort).  It immediately calls the
```

```
                    selection sort routine to produce a sorted list of
                    points, SP, in order of increasing angles with respect
                    to the current origin.  After this call is made, the list
                    SP is checked for points with the same angle.  If a
                    sequence of points is found with this property, then they
                    are further sorted by order of increasing distance to the
                    current origin.  For this we make calls to the same
                    quicksort routine but with the third parameter set
                    to TRUE to indicate that the sorting is to take place with
                    respect to distance.
*/
                                                                          330
angular_sort ()                                                  angular_sort
{
        int i,j;

        quicksort(SP, 0, NPTS-1, FALSE);                  /* sort by angles      */
        i = 0;
        do {
          while ((i < NPTS) && (SP[i].ang != SP[i+1].ang)) i++;
          if (i < NPTS) {
             j = i;                                                       340
             while ((j < NPTS) && (SP[j].ang == SP[j+1].ang)) j++;
             quicksort(SP, i, j, TRUE);          /* sort by dist if necess. */
             i = j + 1;
          }
        } while (i < NPTS);
}


/*
   eq :  Boolean function to test whether the two record of points,          350
         p and q are equal or not.
*/

int eq(p,q)
struct POINT p,q;
{
        if ((p.x==q.x) && (p.y==q.y)) return (1);  else return (0);
}


                                                                          360
/*
   set_ipoint:    This procedure calculates the point of intersection of
                  line segments s1 and s2 and stores the result in the
                  global variable I_PNT.  This is done by solving the
                  equations of both lines of s1 and s2 simultaneously.
                  The case where the two segments meet at a common end-
                  point is also taken into consideration.
*/

set_ipoint (s1, s2)                                              set_ipoint
struct SEG_STRUCT s1, s2;                                                 371
```

```
{
        float m1, m2, b1, b2, dy1, dy2, dx1, dx2;

        if ((eq(*s1.r, *s2.r)==TRUE) || (eq(*s1.r, *s2.l)==TRUE)) I_PNT = *s1.r;
        else if ((eq(*s1.l,*s2.r)==TRUE)||(eq(*s1.l,*s2.l)==TRUE)) I_PNT = *s2.l;
        else {
                m1 = INFINITY;  m2 = -m1;
                dx1 = s1.l->x - s1.r->x;  dy1 = s1.l->y - s1.r->y;
                dx2 = s2.l->x - s2.r->x;  dy2 = s2.l->y - s2.r->y;                    380
                if (dx1 != 0) m1 = dy1 / dx1;
                if (dx2 != 0) m2 = dy2 / dx2;
                b1 = s1.l->y - m1 * s1.l->x;
                b2 = s2.l->y - m2 * s2.l->x;
                if ((dx1 == 0) || (dx2 == 0)) {
                        I_PNT.x = s1.l->x;  I_PNT.y = m2 * I_PNT.x + b2; }
                else {
                        I_PNT.x = (b2 - b1) / (m1 - m2);
                        I_PNT.y = m1 * I_PNT.x + b1;
                }                                                                     390
        }
}


/*
   ccw :  A three-valued (0, 1, -1) function which is used by yet another
          function (intersect) to determine the direction (clockwise or
          counter-clockwise) in traveling from point p0 to p2 via p1.  This
          is the standard implementation provided by [Sed88].
*/                                                                                    400

int ccw (p0, p1, p2)
struct POINT *p0,*p1,*p2;
{
        int ret_value;
        float dx1,dx2,dy1,dy2;

        dx1 = p1->x - p0->x;  dy1 = p1->y - p0->y;
        dx2 = p2->x - p0->x;  dy2 = p2->y - p0->y;
        if (dx1*dy2 > dy1*dx2) ret_value = 1;                                         410
        if (dx1*dy2 < dy1*dx2) ret_value = -1;
        if (dx1*dy2 == dy1*dx2) {
                if ((dx1*dx2 < 0) || (dy1*dy2 < 0)) ret_value = -1;
                else if ((dx1*dx1+dy1*dy1) >= (dx2*dx2+dy2*dy2)) ret_value = 0;
                else ret_value = 1;
        }
        return (ret_value);
}

                                                                                      420
/*
   intersect:     Boolean function to determine if the two line segments, s1
                  and s2, intersect.  If they do, then the point of inter-
                  section is stored in the global variable I_PNT.
```

```
*/

int intersect(s1,s2)
struct SEG_STRUCT s1, s2;
{
        if (s1.l->num == s2.l->num || s1.l->num == s2.r->num) return(FALSE);
        if ((((ccw(s1.l,s1.r,s2.l)*ccw(s1.l,s1.r,s2.r)) <= 0) &&
             ((ccw(s2.l,s2.r,s1.l)*ccw(s2.l,s2.r,s1.r)) <= 0)) == TRUE) {
                set_ipoint(s1, s2);
                return (TRUE);
        }
        else return (FALSE);
}

/*
   balance_L, balance_R, del, delete, insert :

        These are all routines to manipulate the binary tree, T.  It is the
        standard implementation of an AVL tree presented in [Wir86].
*/

struct NODE *balance_L (p, h)           /*  balance tree by rotating left at node p */
struct NODE *p;
int *h;
{
        struct NODE *p1, *p2;
        int b1, b2;

        switch (p->bal) {
           case -1 : p->bal = 0;  break;
           case  0 : p->bal = 1;  *h = FALSE;  break;
           case  1 : p1 = p->r;  b1 = p1->bal;
                     if (b1 >= 0) {                    /* single RR rotation */
                       p->r = p1->l;  p1->l = p;
                       if (b1 == 0) {
                               p->bal = 1;  p1->bal = -1;  *h = FALSE; }
                       else { p->bal = 0;  p1->bal = 0; }
                       p = p1;
                     }
                     else {                            /* double RL rotation */
                       p2 = p1->l;  b2 = p2->bal;
                       p1->l = p2->r;  p2->r = p1;
                       p->r = p2->l;  p2->l = p;
                       if (b2 == 1) p->bal = -1;
                       else p->bal = 0;
                       if (b2 == -1) p1->bal = 1;
                       else p1->bal = 0;
                       p = p2;  p2->bal = 0;
                     }
                     break;
        }
        return (p);
}
```

```
struct NODE *balance_R (p, h)          /* balance tree by rotating right at node p */
struct NODE *p;                                                                    480
int *h;
{
        struct NODE *p1, *p2;
        int b1, b2;

        switch (p->bal) {
           case  1 : p->bal = 0;  break;
           case  0 : p->bal = -1;  *h = FALSE;  break;
           case -1 : p1 = p->l;  b1 = p1->bal;
                     if (b1 <= 0) {                        /* single LL rotation */   490
                       p->l = p1->r;  p1->r = p;
                       if (b1 == 0) {
                               p->bal = -1;  p1->bal = 1;  *h = FALSE; }
                       else { p->bal = 0;  p1->bal = 0; }
                       p = p1;
                     }
                     else {                                /* double LR rotation */
                       p2 = p1->r;  b2 = p2->bal;
                       p1->r = p2->l;  p2->l = p1;
                       p->l = p2->r;  p2->r = p;                                   500
                       if (b2 == -1) p->bal = 1;
                       else p->bal = 0;
                       if (b2 == 1) p1->bal = -1;
                       else p1->bal = 0;
                       p = p2;  p2->bal = 0;
                     }
                     break;
        }
        return (p);
}                                                                                  510

struct NODE *y, *z;                                    /* temporary nodes */

struct NODE *del (r, h)
int *h;
struct NODE *r;
{
        if (r->r != NULL) {
                r->r = del (r->r, h);
                if (*h == TRUE) r = balance_R(r, h);                               520
        }
        else {
                z->dis = r->dis;  z = r;  r = r->l;  *h = TRUE;
        }
        return (r);
}

/*
   delete :  Function that returns a pointer to the root of the
             AVL tree after node p with distance dis has been                      530
```

```
            removed.  See [Wir86] for more details.
*/

struct NODE *delete (s, dis, p, h)
int s,*h;
float dis;
struct NODE *p;
{
        struct NODE *q;
                                                                        540
        if (p == NULL) {
                printf ("Cannot find node to delete!\n"); exit();  }
        else if (p->dis > dis) {
                p->l = delete (s, dis, p->l, h);
                if (*h == TRUE) p = balance_L(p,h);
        }
        else if (p->dis < dis) {
                p->r = delete (s, dis, p->r, h);
                if (*h == TRUE) p = balance_R(p,h);
        }                                                               550
        else {                                          /* node found!  delete it! */
           if (p->count == 2) {
                if (s == p->seg_no[1]) p->seg_no[1] = p->seg_no[2];
                p->count = 1;
           }
           else {
                q = p;
                if (q->r == NULL) {
                        p = q->l;  *h = TRUE; }
                else if (q->l == NULL) {                                560
                        p = q->r;  *h = TRUE; }
                else {
                        z = q;
                        q->l = del (q->l, h);
                        q = z;
                        if (*h == TRUE) p = balance_L(p, h);
                }
                free (q);
             }
        }                                                               570
        return (p);
}

/*
   insert:      Function that returns a pointer to the root of the
                AVL tree after node p with distance dis has been
                inserted.  See [Wir86] for more details.
*/

struct NODE *insert (s, dis, p, h)                                      580
int s,*h;
float dis;
struct NODE *p;
```

```
{
        struct NODE *p1, *p2;

        if (p == NULL) {
                p = (struct NODE *) (malloc(sizeof(TNODE)));  *h = TRUE;
                if (p == NULL) {
                    printf ("Failed to create node\n");  exit(1); }                    590
                else {
                    p->dis = dis;  p->l = NULL;  p->r = NULL;  p->bal = 0;
                    p->count = 1;  p->seg_no[1] = s;
                }
        }
        else if (p->dis > dis) {
                p->l = insert (s, dis, p->l, h);
                if (*h == TRUE)
                   switch (p->bal) {
                      case  1 : p->bal = 0;   *h = FALSE;  break;                      600
                      case  0 : p->bal = -1;  break;
                      case -1 : p1 = p->l;
                                if (p1->bal == -1) {          /* single LL rotation */
                                        p->l = p1->r;  p1->r = p;
                                        p->bal = 0;  p = p1;
                                }
                                else {                        /* double LR rotation */
                                        p2 = p1->r;  p1->r = p2->l;
                                        p2->l = p1;  p->l = p2->r;
                                        p2->r = p;                                     610
                                        if (p2->bal == -1) p->bal = 1;
                                        else p->bal = 0;
                                        if (p2->bal == 1) p1->bal = -1;
                                        else p1->bal = 0;
                                        p = p2;
                                };
                                p->bal = 0;  *h = FALSE;
                                break;
                      default : break;
                   };                                                                  620
        }
        else if (p->dis < dis) {
                p->r = insert (s, dis, p->r, h);
                if (*h == TRUE)
                   switch (p->bal) {
                      case -1 : p->bal = 0;   *h = FALSE;  break;
                      case  0 : p->bal = 1;  break;
                      case  1 : p1 = p->r;
                                if (p1->bal == 1) {           /* single RR rotation */
                                        p->r = p1->l;  p1->l = p;                      630
                                        p->bal = 0;  p = p1;
                                }
                                else {                        /* double RL rotation */
                                        p2 = p1->l;  p1->l = p2->r;
                                        p2->r = p1;  p->r = p2->l;
                                        p2->l = p;
```

```
                                        if (p2->bal == 1) p->bal = -1;
                                        else p->bal = 0;
                                        if (p2->bal == -1) p1->bal = 1;
                                        else p1->bal = 0;                    640
                                        p = p2;
                                }
                                p->bal = 0;  *h = FALSE;
                                break;
                        default : break;
                        };
        }
        else {
                p->count = 2;
                p->seg_no[2] = s;                                            650
                *h = FALSE;
        }
        return (p);
}

/*
    min_seg :  This function returns an index to a segment residing in
               the array of segments S.  It determines the index by
               traversing the binary tree, T, to the leftmost node and
               retrieving the segment number stored in the 'seg_no' field    660
               of that node.
*/

int min_seg(p)
struct NODE *p;
{
        struct POINT t1, t2;
        struct NODE *q;

        q = p;                                                               670
        if (q == NULL) return (-1);
        while (q->l != NULL) q = q->l;
        if (q->count == 2) {
                set_ipoint(sweepline, S[q->seg_no[1]]);  t1 = I_PNT;
                set_ipoint(sweepline, S[q->seg_no[2]]);  t2 = I_PNT;
                if (len(origin,t1) < len(origin, t2)) return (q->seg_no[1]);
                else return (q->seg_no[2]);
        }
        else return (q->seg_no[1]);
}                                                                            680


/*
   add_arc:      This procedure adds an arc from point number i to point
                 number j in the visibility graph VG, represented as an
                 an adjacency matrix.  It essentially places the value
                 'arc_weight' into the matrix at positions (i, j) and (j, i)
                 since the graph is undirected.  It then places a dot on the
                 screen at point number j to indicate that it is visible to
```

```
                the origin (point number i).                                    690
*/

void add_arc (i, j, arc_len)
int i, j;
float arc_len;
{

        color_x(BLUE);
        circle_x(V[j].x,V[j].y,1.5);
                                                                                700
        VG[i][j] = arc_len;
        VG[j][i] = arc_len;

}


/*
   seg_on_line:   Boolean function to determine if line segment i in S lies on
                  the sweep-line s.  If it is, the function returns
                  TRUE.  It returns FALSE otherwise.                            710
*/

int seg_on_line(s, i)
struct SEG_STRUCT s;
int i;
{

        if (ccw(s.l,s.r,S[i].l)==0 && ccw(s.l,s.r,S[i].r)==0) return (1);
        else return(0);
}                                                                               720


/*
   angle :  This function returns the angle made by points a, b, and c (in
            that order).  Hence, its range is [0, 360).
*/


float angle (a, b, c)                           /* return the angle made by a,b,c */
struct POINT a, b, c;                                                           730
{
        double t1, t2, ang;

        t1 = theta(&b,&a);  t2 = theta(&b,&c);
        ang = max(t1,t2) - min(t1,t2);
        if (ang > 180.0) ang = 360 - ang;
        return ((float) ang);
}

                                                                                740
/*
   set_view :     This function returns TRUE if the initial instance of the sweep
```

```
                line lies outside the adjacent segments belonging to the current
                origin.  It returns FALSE otherwise.
*/

int set_view(a,o,b)
struct POINT a, o, b;
{
        if (o.a == -1) return (FALSE);          /*  point is either X or Y  */
        if (ccw(&a, &o, &SP[0]) == ccw(&b, &o, &SP[0]))
                return (TRUE);
        else return (FALSE);
}


/*
   get_adj_pts :        This procedure returns the adjacent segments of point p
                        in the line segments pointed to by s1 and s2.
*/

get_adj_pts(p, s1, s2)                  /* return the adjacent segments to point p */
struct POINT p;
int *s1, *s2;
{
        *s1 = (S[p.a].l->num == p.num) ? S[p.a].r->num : S[p.a].l->num;
        *s2 = (S[p.b].l->num == p.num) ? S[p.b].r->num : S[p.b].l->num;
}


/*
    shortest_path :  This procedure determines the shortest path from point V[s] to V[f]
                       in the visibility graph, VG.  It uses Dijkstra's greedy algorithm on
                       the adjacency matrix representing the undirected graph.
*/


shortest_path (s, f)
int s, f;
{
        int t, k, x, last, pre[MAXPTS];

        for (x=0; x<=NPTS; x++) {
                label[x] = INFINITY;
                final[x] = FALSE;
        }
        label[s] = 0;
        final[s] = TRUE;
        last = s;
        while (final[f] == FALSE) {
            for (x=0; x<=NPTS; x++)
                if (final[x] == FALSE && (label[x] > label[last]+VG[last][x])) {
                        label[x] = label[last] + VG[last][x];
                        pre[x] = last;
                }
```

get_adj_pts

shortest_path

```
            t = INFINITY;
            for (x=0; x<=NPTS; x++)
                if (final[x] == FALSE && label[x] < t) k = x;
            final[k] = TRUE;
            last = k;                                                    800
        }
        color_x (CYAN);
        t = f;
        do {
            line_x(V[pre[t]].x,V[pre[t]].y,V[t].x,V[t].y);
            t = pre[t];
        } while (t != s);
}

                                                                         810
/*
   initial_set_up :        This is the procedure described in the implementation section
                           of the thesis.  It will initialize the AVL tree to the
                           configuration obtained when the sweep–line is at the first
                           instance.  All N line segments are tested for intersectoin
                           in this procedure and the visibility of the first point
                           in the sorted array of points, SP[0], is determined.
*/

initial_set_up(sline)                                          initial_set_up
struct SEG_STRUCT sline;                                                 821
{
        int i, j, k;
        struct SEG_STRUCT s1;

        T = NULL;
        color_x(WHITE);  line_x (origin.x, origin.y, SP[0].x, SP[0].y);
        if (origin.a != -1) get_adj_pts(origin, &A, &B);
        if (SP[0].a != -1) get_adj_pts(SP[0], &i, &k);
        viewable = set_view(V[A],origin,V[B]);                           830
        for (j=0; j<=NSEGS; j++) {
           s_dist[j] = -1.0;
           if ((intersect(sline,S[j])==TRUE) && (seg_on_line(sline,j)==FALSE)) {
                s_dist[j] = len(origin, I_PNT);
                T = insert(j, s_dist[j], T, &flag);
                color_x (RED);
                line_x(S[j].l->x,S[j].l->y,S[j].r->x,S[j].r->y);
           }
        }
        if (viewable) {                                                  840
          k = min_seg(T);
          if (SP[0].a == k || SP[0].b == k || T==NULL)
                add_arc (origin.num, SP[0].num, len(origin, SP[0]));
        }
        if (SP[0].num == A || SP[0].num == B)
                add_arc (origin.num, SP[0].num, len(origin, SP[0]));
        getch();
        color_x(BLACK);  line_x (origin.x, origin.y, SP[0].x, SP[0].y);
```

```
}
                                                                        850

/*
   calculate_angles :      This procedure will recalculate the angles in the 'ang' field
                           of each point in the point arrays V and SP with respect
                           to the origin which is the argument.
*/

calc" te_angles (center)                                    calculate_angles
stru . POINT *center;
{                                                                       860
        int i, j;

        j = 0;
        for (i=0; i<=NPTS; i++) {
                V[i].ang = theta (center, &V[i]);
                if (V[i].num != center->num) SP[j++] = V[i];
        }
}

                                                                        870
/*
   build_VG :    Here is the main procedure for generating VG. The sweep cycle is executed
                 for each of the N points. In each cycle, the updating procedure on the
                 edges of the obstacles is performed and visibility of points to the center
                 is determined. The procedure executes in time O(n^2 log n) as
                 described in the paper. The animation routines are also embedded within
                 this procedure to allow visualization of the algorithm as described in the
                 animation section of the thesis.
*/
                                                                        880
build_VG ()                                                 build_VG
{
        int u, v, i, r[2], no_ints, curr, prev;
        float angular_pos;
        struct SEG_STRUCT ds;

        for (u=0; u<=NPTS; u++) {
            draw_env();
            origin = V[u];
            flag = TRUE;                                            890
            calculate_angles(&V[u]);
            angular_sort ();
            sweepline.l = &V[u];  sweepline.r = &SP[0];
            initial_set_up(sweepline);
            prev = min_seg(T);
            for (v=1; v<NPTS; v++) {
              sweepline.r = &SP[v];
              color_x(WHITE);  line_x (V[u].x, V[u].y, SP[v].x, SP[v].y);

              /* check if we need to toggle the VIEWABLE flag  */       900
```

```
if ((SP[v].num == A) || (SP[v].num == B)) {
    add_arc(V[u].num, SP[v].num, len(V[u],SP[v]));
    viewable = (viewable == TRUE) ? FALSE : TRUE;
}

/* Next, insert & delete active and inactive edges to T */

r[0] = SP[v].a;  r[1] = SP[v].b;
for (i=0; i < 2; i++) {                                          910

    /*
          If the segment is in the binary tree,
          then remove it at this instance since it has
          become inactive. The original color of this
          segment is also restored. */

    if (s_dist[r[i]] != -1.0) {
          T = delete(r[i], s_dist[r[i]], T, &flag);
          s_dist[r[i]] = -1.0;                                   920
          color_x (GREEN);  ds = S[r[i]];
          line_x(ds.l->x,ds.l->y,ds.r->x,ds.r->y);
    }

    /*
          Otherwise, the segment was not in the binary tree.
          We ignore this segment if it lies on the sweep-line.
          If this is not the case, then the segment has become
          active and is inserted into the tree. The segment is
          also highlighted on the screen to indicate that it is  930
          active.  */

    else if (seg_on_line(sweepline,r[i]) == FALSE) {
          s_dist[r[i]] = len(V[u],SP[v]);
          T = insert(r[i], s_dist[r[i]], T, &flag);
          color_x (RED);  ds = S[r[i]];
          line_x(ds.l->x,ds.l->y,ds.r->x,ds.r->y);
    }
}
                                                                 940
/* update minseg(T) to reflect current orientation */

curr = min_seg(T);
if (curr != -1) {
          set_ipoint(sweepline, S[curr]);
          color_x(WHITE);  circle_x(I_PNT.x,I_PNT.y,1.0);
          T = delete (curr, s_dist[curr], T, &flag);
          s_dist[curr] = len(I_PNT, V[u]);
          T = insert(curr, s_dist[curr], T, &flag);
}                                                                950

/* Check to see if minseg(T) changed from previous orientation  */

curr = min_seg(T);
```

```
                if ((curr != prev) && (viewable == TRUE))
                        add_arc(V[u].num, SP[v].num, len(V[u],SP[v]));
                        prev = curr;
                color_x(BLACK);
                line_x (V[u].x, V[u].y, SP[v].x, SP[v].y);
        }                                                                    960

        /* Free all nodes of the binary tree for the next sweep cycle.  */

        while (T != NULL)
                T = delete (T->seg_no[1], T->dis, T, &flag);
    }
}


main ()                                                                      main
{                                                                            971
        int i, j;


        set_up_environment ();
        initialize_display ();
        for (i=0; i<=NPTS; i++)
                for (j=0; j<=NPTS; j++) VG[i][j] = INFINITY;
        build_VG();
        draw_env();                                                          980
        printf ("shortest path...");
        shortest_path(X.num, Y.num);
        getch();

        restore_graphics_display ();

}
```

Bibliography

[AHU83] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. Data Structures and Algorithms. Addison-Wesley Publishing Company, 1983.

[BO79] Jon L. Bentley and Thomas A. Ottmann. Algorithms for Reporting and Counting Geometric Intersections. IEEE Transactions on Computers, c-28(9): 643-647, 1979.

[Boy79] John W. Boyse. Interference Detection Among Solids and Surfaces. Communications of the ACM, 22(1): 3-9, 1979.

[CW83] Francis Chin and Cao An Wang. Optimal Algorithms for the Intersection and Minimum Distance Problems Between Planar Polygons. IEEE Transactions on Computers, c-32(12): 1203-1207, 1983.

[FF88] Leila De Floriani and Bianca Falcidieno. A Hierarchical Boundary Model for Solid Object Representation. ACM Transactions on Graphics, 7(1): 42-60, 1988.

[Goo87] Michael T. Goodrich. Finding the Convex Hull of a Sorted Point Set in Parallel. Information Processing Letters, 26: 173-179, 1987.

[Gun88] Oliver Gunther. Efficient Structures for Geometric Data Management. Lecture Notes in Computer Science 337, Springer-Verlag Publishing, 1988.

[HMMN84] Stefan Hertel, Martti Mantyla, Kurt Mehlhorn, and Jurg Nievergelt. Space Sweep Solves Intersection of Convex Polyhedra. Acta Informatica, 21:501-519, 1984.

[HNS88] K. Hinrichs, J. Nievergelt, and P. Schorn. A Sweep Algorithm for the All-Nearest-Neighbors Problem. Computational Geometry and its Applications, Springer-Verlag Publishing, 43-54, March 1988.

[Kha86] Oussama Khatib. Real-Time Obstacle Avoidance for Manipulators and Mobile Robots. The International Journal of Robotics Research, 5(1): 90-98, 1986.

[LP84] D. T. Lee and F. P. Preparata. Euclidean Shortest Paths in the Presence of Recilinear Barriers. Networks, 14:393-410, 1984.

[LW79] T. Lozano-Perez and M.A. Wesley. An Algorithm for Planning Collision-free Paths Among Polyhedral Obstacles. Communcations of ACM, 22(10): 560-570, October 1979.

[Mer86] Ed Merks. An Optimal Parallel Algorithm for Triangulating a Set of Points in the Plane. International Journal of Parallel Programming, 15(5): 399-411, 1986.

[NP82] J. Nievergelt and F. P. Preparata. Plane-Sweep Algorithms for Intersecting Geometric Figures. Communications of ACM, 25(10):739-747, October 1982.

[Oom87] B. John Oommen. An Efficient Geometric Solution to the Minimum Spanning Circle Problem. Operations Research, 35(1): 80-86, 1987.

[Pap84] C. H. Papadimitriou, An algorithm for Shortest-path Motion in Three Dimensions. Manuscript. Computer Science Department, Stanford University, July 1984

[PS85] Franco P. Preparata and Michael Ian Shamos. Computational Geometry: An Introduction. Springer-Verlag Publishing, 1985.

[PY73] Franco P. Preparata and Raymond T. Yeh. Introduction to Discrete Structures for Computer Science and Engineering. Addison-Wesley Publishing Co., 1973.

[Qui87] Michael J. Quinn. Designing Efficient Algorithms for Parallel Computers. McGraw-Hill Book Company, 1987.

[ROH88] Hans Rohnert. Time and Space Efficient Algorithms for Shortest Paths Between Convex Polygons. Information Processing Letters, 27:175-179, April 1988.

[ROIK87] N. Rao, B. Oommen, S. Iyengar, and L. Kashyap. Robot Navigation in Unknown Terrains Using Learned Visibility Graphs. Part I: The Disjoint Convex Obstacle Case. IEEE Journal of Robotics and Automation, 3(6):672-681, December 1987.

[ROIK88] N. Rao, B. Oommen, S. Iyengar, and L. Kashyap. On Terrain Model Acquisition by a Point Robot Amidst Polyhedral Obstacles. IEEE Journal of Robotics and Automation, 4(4):450-455, August 1988.

[RND77] Edward M. Reingold, Jurg Nievergelt, and Narsingh Deo. Combinatorial Algorithms: Theory and Practice. Prentice-Hall Inc., 1977.

[Sed88] Robert Sedgewick. Algorithms. Addison-Wesley Publishing Co., 1988.

[SS84] M. Sharir and A. Schorr. On Shortest Paths in Polyhedral Spaces. Technical Report 138, Robotics Report 31, New York University, October 1984.

[Til81] Robert B. Tilove. Line/Polygon Classification: A Study of the Complexity of Geometric Computation. IEEE Computer Graphics and Applications, pp. 75-83, April 1981.

[Whi85] S. H. Whitesides. Computational Geometry and Motion Planning. Computational Geometry. Elsevier Science Publishers, pp. 377-427, 1986.

[Wir86] Nicklaus Wirth. Algorithms and Data Structures. Prentice-Hall Inc., 1986.

[WIRJ81] C.R. Weisbin, S.S. Iyengar, S.V.N. Rao, and C.C. Jorgensen. Robot Navigation Algorithms Using Learned Spatial Graphs. Robotica, 4:93-100, 1986.