1-1-1992

Preliminary Investigations Into Efficient Line Of Sight Determination In Polygonal Terrain

Mikel D. Petty

INSTITUTE FOR SIMULATION AND TRAINING

Contract Number N61339-89-C-0044
February 28, 1992

Preliminary Investigations into Efficient Line of Sight Determination in Polygonal Terrain

Institute for Simulation and Training
12424 Research Parkway, Suite 300
Orlando FL 32826

University of Central Florida
Division of Sponsored Research

IST-TR-92-5

INSTITUTE FOR SIMULATION AND TRAINING

Preliminary Investigations into Efficient Line of Sight Determination in Polygonal Terrain

Contract N61339-89-C-0044
February 28, 1992

IST-TR-92-5

Prepared by
Mikel D. Petty, Charles E. Campbell, Robert W. Franceschini,
Micheline H. Provost, Clark R. Karr

Mikel D Petty

Reviewed by

Richard R. Dunn-Roberts

[illegible]

Institute for Simulation and Training • 12424 Research Parkway, Suite 300 • Orlando, Florida 32826

University of Central Florida • Division of Sponsored Research

PRELIMINARY INVESTIGATIONS INTO EFFICIENT LINE OF SIGHT DETERMINATION IN POLYGONAL TERRAIN

Mikel D. Petty
Charles E. Campbell
Robert W. Franceschini
Micheline H. Provost
Clark R. Karr

February 28, 1992

IST-TR-92-5

Institute for Simulation and Training
12424 Research Parkway, Suite 300
Orlando FL 32826

Table of Contents

1. Introduction

1.1 Purpose

This document is the final technical report required as a deliverable under University of Central Florida Division of Sponsored Research grant #90370, titled "Efficient Line of Sight Determination in Polygonal Terrain". It is also a non-required deliverable under DARPA contract N61339-89-C-004, titled "Intelligent Simulated Forces: Evaluation and Exploration of Computational and Hardware Strategies". The research described herein was supported by those two sources.

1.2 Structure of this document

Following this subsection, the remainder of section 1 introduces the Line of Sight problem in the context of real-time battlefield simulation, describes polygonal terrain, and lays out the objectives of this research. Section 2 explains in detail the Line of Sight algorithms developed during this project, and the existing algorithm used as a standard for comparison. Section 3 presents the comparison experiment performed on those algorithms, and the experimental results obtained. Section 4 summarizes the results and identifies potential areas of future related work. Sections 5 and 6 contain references and appendices related to the project.

This document assumes that the reader is familiar with computer algorithm design in general, but not with the specific algorithms or data structures used for Line of Sight determination. It further assumes that the reader has some familiarity with real-time battlefield simulation, as exemplified by the SIMNET system.

(This document shares some text excerpts with IST Technical Report IST-TR-92-6, titled "The IST Semi-Automated Forces Dismounted Infantry System: Capabilities, Implementation, and Operation". Most of the shared text is contained in section 2.3 of this document. The text is repeated so as to allow each report to be read without requiring the reader to have access to the other.)

1.3 Line of Sight in battlefield simulation

In an interactive real-time battlefield simulation (such as SIMNET), a question of paramount concern is this: can two hostile entities, such as tanks, see each other? More formally, does there exist a Line of Sight (LOS) between them? The existence of a LOS between a pair of entities in the simulation substantially affects their behavior; for example, a LOS is a prerequisite of direct fire. It is the simulated terrain that can block the LOS; two tanks on opposite sides of a hill cannot see each other.

A LOS determination must be made for each pair of entities whose behavior may be affected by the existence of a LOS. The potential number of LOS determinations that must be made in a simulation is very large; in the worst case, for a scenario with n entities, on the order of n^2 individual LOS checks must be made during each LOS check cycle. More precisely, the worst case is approximately

$$((n/2)^2)/2 \in O(n^2)$$

Heuristics exist to reduce this in the average case to O(n), but the worst case remains $O(n^2)$. (See section 1.6 for a definition of order notation.)

Furthermore, those LOS determinations must be made often to ensure realistic behavior. For a typical real-time simulation, each entity should check for a LOS to each hostile entity at least once each second, and preferably more often, to maintain realism.

The point is that in a simulation of a size typical for interactive battlefield simulation, a great many LOS determinations must be made in a short time. The required LOS calculations can consume a significant portion of the computational power of the simulation computer, reducing the resources available for more interesting processing, such as realistic vehicle dynamics or intelligent behavior. An improvement in LOS algorithm run-time efficiency could have a significant impact on simulation system performance.

By now, a reader who is familiar with distributed battlefield simulation systems such as SIMNET will be raising the issue of SIMNET's inherent parallelism. In SIMNET, it is often the case that each simulated entity is supported by a separate simulation computer. Thus, the $O(n^2)$ LOS determinations are distributed onto n simulation computers. The situation can be described as a parallel algorithm requiring O(n) LOS determinations on each of n processors. Furthermore, there are no data dependencies, given the usual SIMNET situation of a redundant static terrain database on each simulation computer.

There are two points to consider in relation to the preceding observation. First, reducing the number of LOS determinations for each simulation computer from $O(n^2)$ to O(n) does not change the fact that an improvement in the efficiency of each LOS determination makes more computational power available to other tasks. Second, SIMNET simulations often include simulators that control multiple, possibly many, entities. Semi-automated forces systems are a prime example of this. Those systems must perform LOS determinations for each of the entities they support, between each pair of those entities as well as the other entities in the simulation. For semi-automated forces systems, the processing of LOS determinations has been estimated to consume more than half of a system's computational power [Companion,1989]. Clearly, a more efficient LOS algorithm is of considerable importance to semi-automated forces systems.

1.4 Polygonal terrain

In nearly all interactive real-time battlefield simulations, the terrain over which the simulation takes place is constructed from a large set of planar polygons. Adjacent polygons share edges, but are not necessarily coplanar; the ridges, valleys and rolls in the terrain are determined by the 3D spatial coordinates of the polygons. See figure 1.1 for an example.

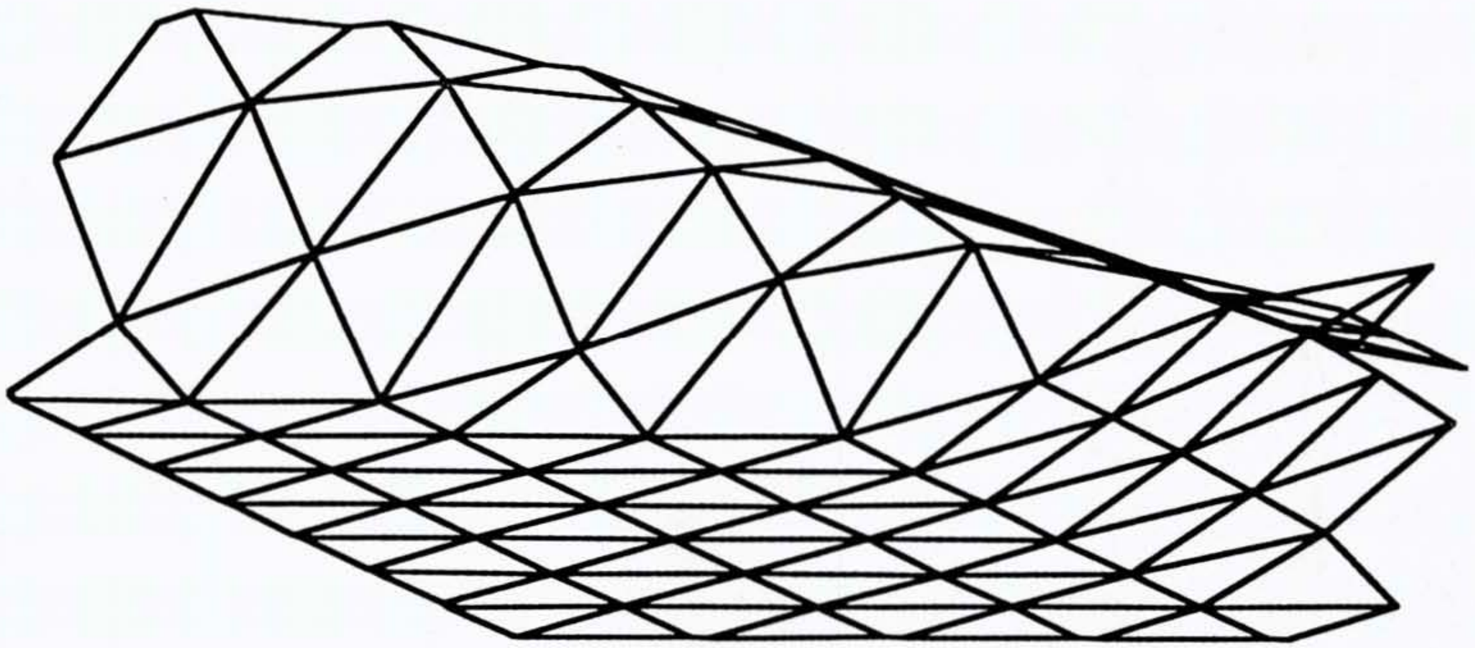


Figure 1.1. Polygonal terrain representing a hillside (seen from an oblique viewpoint).

1.5 Research goals

Of course, real-time battlefield simulations exist (e.g. SIMNET), so LOS algorithms have been produced. They are often ad hoc algorithms whose performance is adequate but not optimized [Stanzione,1989], or approximation algorithms that achieve good performance at the expense of occasionally incorrect results [Gonzalez,1990].

Some interesting theoretical results in the computer science subdiscipline of computational geometry have been derived that are potentially relevant to the problem of efficient LOS determination; see any of [Guibas,1987], [Ghosh,1987], [Chazelle,1988], [Hershberger,1989], or especially [Cole,1989].

Unfortunately, there are two features of the theoretical results that make them something less than a definitive answer to the LOS problem. The first shortcoming is that many of the results are for restricted domains. The polygonal terrain of most battlefield simulations is made up of many polygons, located in three dimensions (3D), and the polygons cannot be assumed to be triangles. The theoretical results are often for one polygon, for coplanar polygons, and/or for triangles or triangulated polygons.

The second problem with the theoretical results is that the analysis of the time required for these algorithms is usually taken only to the point of showing that the algorithm requires time linear in the size of the problem. (The "size of the problem" may be the number of entities performing line of sight checks, as in section 1.2, or the number of polygons involved in the LOS check, as in most LOS algorithm analysis.) In other words, linear algorithms require a time proportional to the number of polygons or polygon vertices to be considered times a multiplicative constant. Such algorithms are called "optimal". However, as is the practice in theoretical algorithmic analysis, the magnitude of the multiplicative constant is neither given nor even known. It should be clear that an algorithm requiring time proportional to 2 times the number of polygons is much more suitable than one requiring time proportional to 5 times the number of polygons.

The objective of this project was to develop more efficient algorithms for LOS determination in polygonal terrain, and to measure their efficiency empirically. Three new LOS algorithms were developed at IST under this project; they are herein referred to as algorithms F, C, and P, and are described in this report. Their performance was compared to the SIMNET Planview Display LOS algorithm, which is also briefly described.

1.6 Order notation

This document will often give the computational time or space requirements of a particular LOS algorithm or sub-algorithm. The standard order notation, as used in algorithmic analysis, will be employed for that purpose. This section defines the order notation for those readers who are not familiar with it. The definition is adapted from [Preparata,1988] and [Brassard,1988].

$O(f(n))$ denotes the set of all functions $g(n)$ such that there exist positive constants C and n_0 with $|g(n)| \leq Cf(n)$ for all $n \geq n_0$.

In other words, $O(f(n))$ is the set of functions that are at most as large as some constant times $f(n)$. This notation is used to specify the upper-bound or worst-case performance of the algorithm. Informally, if an algorithm's time complexity is described as $O(\log n)$, for example, that means that the time required by that algorithm to run will increase by an amount proportional to the logarithm (usually base 2) of the size of the increase in the problem.

2. LOS algorithms

2.1 General algorithm structure

Although they differ substantially in detail, the three LOS algorithms developed for this project share a common abstract structure. This section will describe that structure; the following sections will explain the algorithms in depth.

In its simplest form, the LOS problem can be stated in this way: given a finite set of polygons in 3-space, and a line segment in 3-space specified by its endpoints, does the line segment intersect any of the polygons? In this formulation, the set of polygons corresponds to the polygonal terrain database, where the terrain itself, the treelines, the buildings, and all of the other features of the terrain database can be constructed from or considered as polygons, and the endpoints of the line segment correspond to the sighting and target entities. Thus a naive brute-force algorithm would simply check every polygon in the finite set to determine if that polygon and the line segment intersected; if none did, the LOS is unblocked.

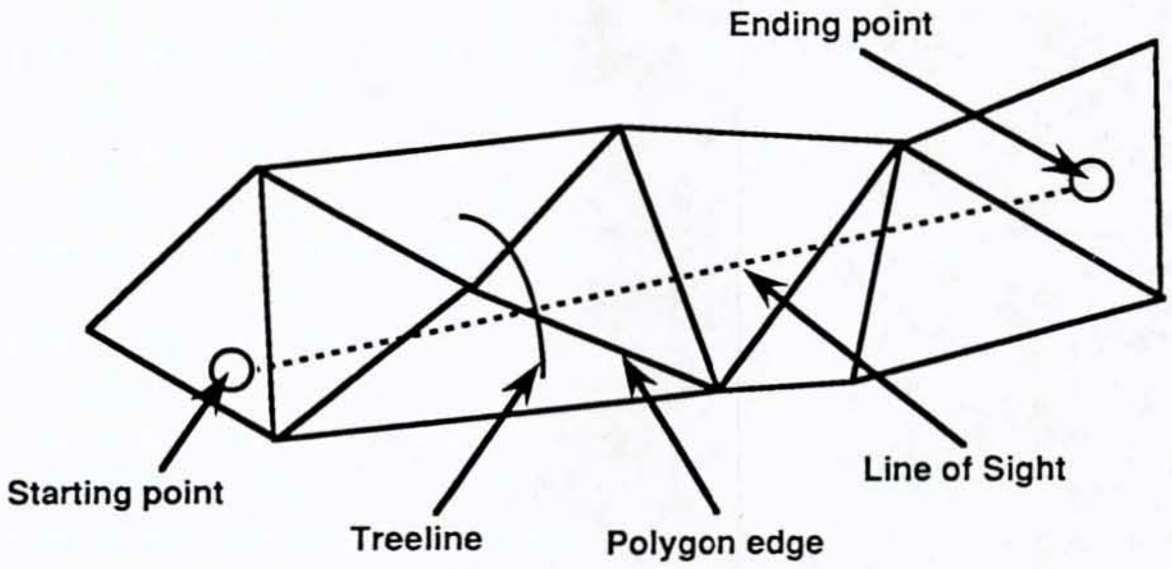


Figure 2.1. A simple LOS example, seen from directly above.

Of course, the polygons of a polygonal terrain database have considerable organization beyond that of a set, assuming the database is representing the surface of a piece of normal terrestrial terrain. For example, the base polygons (the polygons that represent the surface of the terrain) of a polygonal terrain database do not overlap, and adjacent base polygons share common edges. The LOS algorithms to be explained take advantage of the organization inherent in the terrain database polygon set as a means of reducing the execution time of a LOS determination. In order to use that organization, the polygons must be converted into some data structure that embodies or represents the organizational structure of the terrain polygons. (Terminological note: hereinafter, terrain database refers to the polygon set that represents the terrain; these terms will be used interchangeably.)

Constructing the data structure representing the polygons is referred to as *preprocessing*; during preprocessing, the polygons of the terrain database are converted into a data structure specific to the particular LOS algorithm. One possible data structure, but by no means the only one, is a SIMNET format terrain database. Preprocessing is done only once for a given terrain database and LOS algorithm.

Once the terrain data structure has been constructed, it is used to determine if a LOS exists between the sighting and target points. This LOS determination has two basic steps. In the first step, referred to as *point location*, the terrain data structure is used to identify the base terrain polygons which include the two points. Of course, both points may be in the same polygon. Point location is alwoays performed in two dimensions (2D); the sighting and target points, as well as the base polygons, are projected onto the xy plane by setting all the z coordinates to 0. (Point location, taken as the general problem of determining the 2D region containing a given point, is an entire problem category in computational geometry; see [Preparata,1988].)

The second step of the LOS determination process is herein referred to as *LOS traversal*. During LOS traversal, the terrain data structure is used to identify all the base terrain polygons through which the line segment joining the sighting and target points passes. As in point location, the LOS traversal is considered in 2D; the line segment and the base polygons are projected into the xy plane.

As each polygon through which the LOS passes is identified, either during point location or LOS traversal, it is checked to determine if the LOS intersects it, i.e. if it blocks the LOS. Note that this check considers the LOS and the polygon in 3D. The mechanisms for performing the check are different for each algorithm, but they all take advantage of the easily proven fact that if the LOS intersects a planar polygon, it must pass through or below one of the edges of the polygon. (Here below refers to having a smaller z coordinate, which is interpreted as under the surface of the terrain.)

Each LOS algorithm consists of these three components, or sub-algorithms: preprocessing, point location, and LOS traversal. Figure 2.2 (on the next page) shows the algorithms explained in this section and identifies for each the sub-algorithm used for each of the three components.

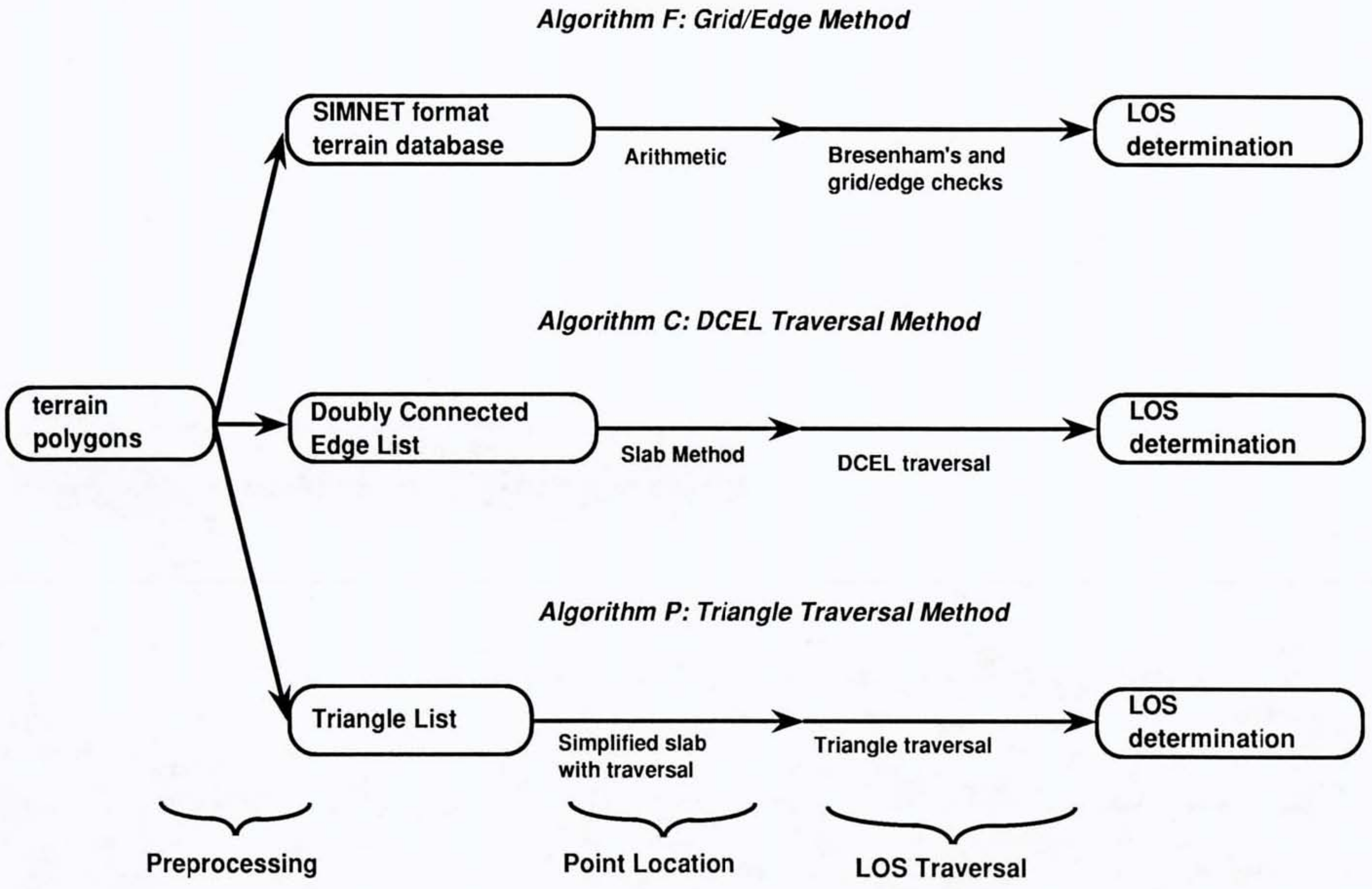

Algorithm F: Grid/Edge Method
SIMNET format terrain database
Arithmetic
Bresenham's and grid/edge checks
LOS determination
Algorithm C: DCEL Traversal Method
terrain polygons
Doubly Connected Edge List
Slab Method
DCEL traversal
LOS determination
Algorithm P: Triangle Traversal Method
Triangle List
Simplified slab with traversal
Triangle traversal
LOS determination
Preprocessing
Point Location
LOS Traversal

Figure 2.2. The three LOS algorithms in abstract.

All of the LOS algorithms described in this document accept starting and ending sighting points as input, and return 1 if a LOS can be established between the two points, and return 0 if not.

Typically, a single terrain database is involved in many LOS determinations. For each LOS determination, the data structure constructed during preprocessing is used for point location and LOS traversal. Thus, it is appropriate to expend additional computational effort during preprocessing, which must be done only once for each terrain database, in order to reduce the time required to perform point location and/or LOS traversal, which must be done for every LOS determination. Computational time spent during preprocessing is not considered in the computational cost of the LOS determinations.

2.2 Polygonal terrain database

As previously mentioned, a *base polygon* is one that forms the ground surface of the terrain being represented. Hills, valleys, and flat areas are constructed from base polygons. In a properly constructed terrain database, projecting the complete set of base polygons into the xy plane produces a complete coverage of the plane, with no overlaps or gaps between the base polygons. (Note that this assumption rules out the possibility of caves, overhanging cliffs, and similar terrain structures. We do not consider this to be a serious limitation; SIMNET uses no such terrain.)

Features are defined to mean terrain features, also constructed from polygons, which are not base polygons, and may block a LOS. Two types of features are considered in the terrain database used for this project. *Treelines* are fencelike sequences of vertical polygons, which extend up from base polygons to a fixed distance above them. A treeline may consist of up to 256 continuous polygons. They are intended to represent rows or narrow strips of trees. A *canopy* is intended to represent a group or clump of trees. A canopy is constructed from a treeline which begins and ends from the same point, together with a set of LOS-blocking polygons that form a "roof" or "cover" extending across from one side to the other of the surrounding treelines.

Polygonal terrain databases may contain other features or objects on the terrain, such as individual trees, buildings, utility poles, or water towers. For purposes of this project, such objects are not considered to block a LOS and are thus not processed by the LOS algorithms. In general, such objects would be handled by the algorithms in a manner similar to the treeline or canopy features.

The terrain database used in this project as input to the preprocessing routines consists of records, each containing a single polygon. The polygon records identify the type of the polygon and list the polygon's vertices. The polygons are either triangles or quadrilaterals for base polygons and canopy polygons, and can have up to 256 vertices for treelines (canopy and treelines are defined below). Road, river, and object polygons are not processed, as they are not considered to block a LOS.

2.3 IST algorithm F: Grid/edge method

2.3.1 Overview

Algorithm F, the Grid/edge method, determines point-to-point LOS using the SIMNET terrain database; i.e. the data structure produced during the preprocessing step is a SIMNET format terrain database. The LOS calculation involves deciding whether the line between two points is blocked by a feature of the terrain database. For this algorithm, as for the others to follow, there are three types of terrain which can block the LOS: elevation of the terrain (e.g. mountains, hills, and valleys) in the form of base polygons, treelines, and canopies.

To check whether the LOS is blocked by the elevation of the terrain, the algorithm must determine if the LOS intersects a polygon along the LOS. The check for intersection is done by determining if the LOS has a lower height (i.e. z coordinate) than the height of any polygon edge the LOS crosses, at the point the LOS crosses the edge.

Section 2.3.2 discusses the preprocessing required by this algorithm. Sections 2.3.3 and 2.3.4 present the point location and LOS traversal steps, which are part of the run-time LOS determination.

2.3.2 Preprocessing

The preprocessing step builds from the terrain polygons a SIMNET format terrain database. The process of constructing a SIMNET format terrain database is involved and tedious, and the original development of that preprocessing algorithm was not part of the research performed for this project (although a new program to perform that preprocessing was written by a member of the project team). Consequently, the preprocessing itself will not be detailed; instead, the structure of the SIMNET format terrain database will be described.

The terrain database is divided into blocks which represent 500 meter x 500 meter squares called *patches*. Each patch is composed of 16 125 meter x 125 meter squares called *grids*. Both patches and grids can be referenced by computing indices based on the coordinates of a point inside them.

The surface of the terrain is represented in the database by polygons; the 3D vertices of each polygon are used to compute the height of any point within the polygon. Polygons are organized by their patch and grid locations. Each edge within a patch structure has a code that tells which grids within the patch contain that particular edge. Polygons do not span patch boundaries; that is, the polygons are defined so that patch boundaries and polygon edges are always collinear, although the reverse is not true. Polygons may span grid boundaries.

The terrain for each patch is represented in the database as vertices, edges, and polygons in the following manner. Each vertex is an entry in a point array; the point array holds the coordinates of all vertices. Each edge is an entry in an edge array; the edge array holds the indices into the vertex array of the edges' vertices. Each polygon is an entry in a polygon array; the polygon array holds the indices into the edge array of the polygons' edges.

In short, the database is defined in such a fashion that an algorithm can move among patches, grids, polygons, edges, and vertices with ease.

2.3.3 Point location

The patch and grid structure of the SIMNET terrain database makes point location quite simple. Because the patches and grids are square and of fixed size, it is possible to determine the patch and grid containing a given sighting point using arithmetic operations in O(1) time. Although a grid may contain multiple terrain polygons, it is not necessary to determine which polygon in the grid contains the point; identifying the grid is sufficient for this algorithm, for reasons the next section will make clear.

2.3.4 LOS traversal

As mentioned in section 2.1, if a LOS intersects a polygon, it will intersect or pass under (in 3D) one of the edges of that polygon. Thus one way to determine if a line intersects a polygon is to find the point of intersection, if any, of the line with each edge of the polygon. Because the terrain database contains a list of edges of polygons indexed by grid location, it is not necessary to make reference to the list of polygons in order to compute LOS. Instead, this algorithm computes the intersections with the LOS of all edges of polygons which are within the grids containing the LOS. This removes a level of indirection and speeds the process of determining LOS.

With that in mind, the first task in determining LOS within the SIMNET terrain database is to determine which patches and grids the LOS passes through; those patches and grids must be searched for the possible LOS blockages by base, treeline, or canopy polygons. In Algorithm F, the patch and grids through which the LOS passes are found using Bresenham's algorithm [Bresenham,1965] (see below). First, the patch and grid indices are computed for the endpoints of the LOS. Then Bresenham's algorithm is used to determine all (patch, grid) pairs which lie along the LOS. The result is a list of (patch, grid) pairs which are then checked for intersections with the LOS.

Bresenham's algorithm, as presented in [Foley,1982], is a two dimensional (2D) scan-conversion algorithm used in computer graphics. Given the endpoints of a segment to be drawn on a 2D raster graphics display, Bresenham's algorithm computes the coordinates of the pixels which lie near the line segment. To adapt this algorithm to the problem of identifying the patches and grids through which a LOS passes in 2D, one need only think of the corners of the grids as pixels.

2.3.4.1 Intersecting LOS with base polygons

The next step in determining LOS is to test whether any of the base polygons in the grids through which the LOS passes block the LOS. This is accomplished by testing each of the polygon edges within the current grid for intersection with the LOS (the line intersection algorithm used is listed in the section 2.3.4.4). The line intersection is computed using (x, y) coordinates only. Then, if the polygon edge and the LOS are found to intersect in 2D, the z coordinates are calculated for the edge and the LOS at the intersection point. These z coordinates determine whether the LOS is blocked. If the z value for the polygon edge is higher than the z value for the LOS, the LOS function immediately decides that the LOS is blocked, and returns 0 without further processing.

2.3.4.2 Intersecting LOS with treelines

After the base polygons are checked, the treelines which are within the (patch, grid) pairs are checked to see whether they block the LOS. The treeline check proceeds in a manner very similar to the polygon edge check. Treelines are represented by a linked list of 3D coordinates; the treeline extends between these points. Further, the treelines have a constant height above the ground. The approach used to determine whether the treelines block the LOS involves first computing the (x, y) intersection point of each treeline edge with the LOS. If the treeline edge and the LOS intersect, the height of their intersection point in the treeline is computed as the height of the treeline plus the height of the terrain. This is compared to the height of the point along the LOS to determine whether the LOS is blocked. If the LOS is blocked, the LOS function returns 0, which indicates that the LOS is blocked, without further processing.

2.3.4.3 Intersecting LOS with canopies

Canopies are represented in the terrain database as a combination of one or more treelines (the border) and polygons (the cover). Thus the LOS test for a canopy involves combining the steps outlined above for base polygons and treelines. Canopies are organized by patches rather than grids. Therefore, all canopies located within a patch through which the LOS passes are checked.

2.3.4.4 Line intersection algorithm

The processes of intersecting the LOS with base polygons and of intersecting the LOS with treelines both depend on determining if and where two line segments intersect. This algorithm determines whether two line segments l and m intersect. Segments l and m are defined by their coordinates $(lx_1, ly_1) \dashrightarrow (lx_2, ly_2)$ and $(mx_1, my_1) \dashrightarrow (mx_2, my_2)$. If the segments do intersect, the intersection point can be determined based on the values found as the result of computing parameters t1 and t2, as well as determinant D. If the lines are coincident, this fact is returned to be dealt with by the LOS traversal algorithm.

The intersection algorithm is as follows:

Parameterize the line segments.

$$l: \langle x, y\rangle = \langle lx_1, ly_1\rangle + t1 * \langle ldx, ldy\rangle$$
$$m: \langle x, y\rangle = \langle mx_1, my_1\rangle + t2 * \langle mdx, mdy\rangle$$

where $ldx = lx_2 - lx_1$,
$ldy = ly_2 - ly_1$,
$mdx = mx_2 - mx_1$,
$mdy = my_2 - my_1$,
$0 <= t1 <= 1$,
and $0 <= t2 <= 1$.

Rewrite these equations into 2 equations with (t1, t2) as the solution by setting the x and y values equal:

$$ldx * t1 - mdx * t2 = mx_1 - lx_1$$
$$ldy * t1 - mdy * t2 = my_1 - ly_1$$

Using Cramer's rule [Munem,1984], define D, t1, and t2 as follows:

$$D = \begin{vmatrix} ldx & -mdx \\ ldy & -mdy \end{vmatrix}$$

$$T1 = \begin{vmatrix} dx & -mdx \\ dy & -mdy \end{vmatrix}$$

$$T2 = \begin{vmatrix} ldx & dx \\ ldy & dy \end{vmatrix}$$

where $dx = mx_1 - lx_1$,
and $dy = my_1 - ly_1$.

Then, the two line segments intersect if and only if

$0 <= t1 <= D$ and $0 <= t2 <= D$ (if $D > 0$)
$D <= t1 <= 0$ and $D <= t2 <= 0$ (if $D < 0$)

The point of intersection can be calculated by using the original equations for l and m. However, it should be noted that the t1 and t2 values calculated above need to be scaled by D (t1 = T1/D and t2 = T2/D) in order to use them in the equations for l and/or m.

The lines are parallel if D = 0. They are coincident if D = 0 and t1 = 0 (t2 will also be 0).

2.3.5 Order analysis

The time and space complexity of several components of the overall algorithm are given in this section.

Point location:
  O(1) time (simple arithmetic on the point's coordinates suffices).
Line intersection:
  O(n) time per grid, where n is the number of polygon edges in the grid.
Features:
  O(m) checks per grid, where m = number of feature segments in the grid.

Note that the order analysis is not very informative for this algorithm. For example, although each line intersection test can be done in O(1) time, the LOS algorithm does many more line intersection tests than are strictly necessary, because it tests every edge in a grid, instead of only those edges through which the LOS passes.

2.4 IST algorithm C: DCEL Traversal method

2.4.1 Overview

The basic idea of the DCEL Traversal method is to begin at the start point and check each base polygon traversed (in 2D) by the LOS to determine if it blocks visibility. The polygons traversed by the LOS are identified quickly using a DCEL data structure, which will be explained below. The features (treelines and canopies) which lie on these base polygons must also be checked. As soon as a base polygon or feature is found that blocks the LOS, the determination is halted; if no blocking polygon is found, the LOS is clear.

Section 2.4.2 discusses the preprocessing required by this algorithm. Sections 2.4.3 and 2.4.4 present the point location and LOS traversal steps, which are part of the run-time LOS determination.

2.4.2 Preprocessing

The preprocessing step constructs the two main data structures used for the point location and LOS traversal steps. The LOS traversal uses a doubly-connected-edge-list (DCEL) data structure, as described in [Muller,1978] and [Preparata,1988]. A DCEL is well suited to represent a planar embedding of a connected planar graph. Under reasonable assumptions, such as no overlapping base polygons and no gaps in the terrain, a polygonal terrain database projected into 2D by removing the z coordinate is such a graph. The preprocessing step constructs the DCEL by first constructing an intermediate data structure, herein referred to as a sorted edge list, from the terrain database and then building the DCEL from the sorted edge list. The sorted edge list has the following form: each vertex in the database has a doubly linked list associated with it, which holds every edge which has that vertex as an endpoint, sorted by their radial angles.

The sorted edge list is somewhat difficult to build due to the nature of the polygon set used for testing (see sections 2.2 and 3.2). In particular, the polygon set contains duplicate polygons. Furthermore, some of the edges of different polygons overlap, yet are not the same edge (i.e. they partially overlap; see Figure 2.3). The DCEL structure requires that each edge in the connected planar graph it represents, and thus the terrain database, be unique and non-overlapping. Also, the terrain database consists of both triangles and quadrilaterals. In order to avoid special cases during the LOS determinations, all quadrilaterals must be split into triangles.

Sections 2.4.2.1 through 2.4.2.5 describe in detail how the DCEL is constructed from the terrain database polygons.

A second phase of the preprocessing is to set up the data structure which will be used for the point location step. That structure is a set of sorted lists of vertices. Section 2.4.2.6 describes those lists in more detail, and explains how they are built.

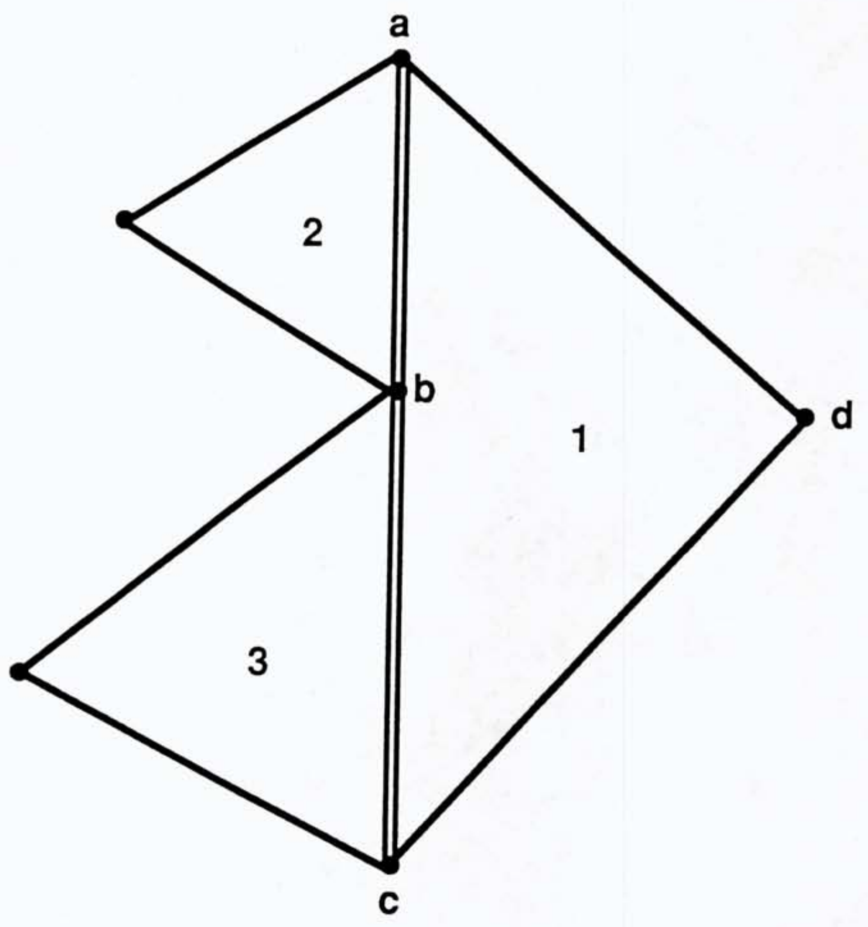


Figure 2.3 Edge ac of triangle 1 overlaps edges ab of triangle 2 and edge bc of triangle 3.

2.4.2.1 Read and prepare polygons

The terrain polygons are read in, one at a time, from the terrain database file. Each polygon is handled according to its type. Treeline and canopy polygons are stored as edges in a list for later use (see section 2.4.2.5). Base polygons are immediately processed.

Processing of base polygons is performed in several steps. First, any base polygons that are quadrilaterals are split into two triangles. Splitting a quadrilateral into triangles requires that the vertices of the quadrilateral be ordered. Because there is no guarantee the vertices are given in order, they are ordered by sorting them by radial angle around a point inside the polygon. Such an interior point can be computed by averaging the coordinates of any three of the four vertices. Once the vertices have been ordered, the quadrilateral is split along one of its diagonals, producing two triangles.

Next, the vertices and edges of the polygon are stored in two AVL trees, a vertex AVL tree and an edge AVL tree. (An AVL tree is height balanced binary tree, where the difference in height between two of its subtrees is at most 1. Once the AVL tree is constructed, an element of an AVL tree may be accessed in O(log n) time. See [Knuth,1973] or [Stubbs,1985] for more information on AVL trees.) The vertices are stored by ascending y and x coordinates, respectively. As each vertex is stored in the vertex AVL tree, it is given an unique identifier, which is returned for use with edge insertion. If a given vertex already exists, it is not stored, but the existing identifier is returned. At the end of this process, the total number of vertices in the terrain is known.

For each edge, its endpoints (which are vertices) are first ordered by ascending y and x coordinates. The edge is then stored in the edge AVL tree by the y and x coordinates of its first endpoint. If the first endpoint is identical to the endpoint of an edge already in the tree, the second endpoints are used. If both endpoints are identical to those of an edge already in the tree, the edge is not stored. As with the vertices, each edge is assigned an unique identifier. An edge is identified by the two vertex identifiers (assigned above) which correspond to its endpoints.

2.4.2.2 Build edge and vertex lists

At this point, the vertices and edges have been numbered (assigned unique identifiers) and duplicate vertices and edges have been removed. The two AVL trees are then transformed into lists which provide direct access to the information.

Two vertex lists will be constructed in a single pass through the vertex AVL tree. The first list will store the vertices sequentially into an array using the ID number they were assigned. Using this array the coordinates for a vertex can be easily retrieved given its identifier, or number. The second list will store the vertices in sorted order for use in building the point location structures (see section 2.4.2.6). Using a recursive depth-first traversal of the vertex AVL tree, the vertices are removed and placed into the sequential list according to their number as well as being placed in the second list in sorted order.

Building the sorted edge list mentioned in the previous section is a bit more difficult. The sorted edge list has one slot for each existing vertex. These slots will point to a list of edges which have the corresponding numbered vertex as an endpoint. Thus, every edge will appear in the edge list exactly twice, once for each endpoint. The edges are inserted by ascending radial angle, and are doubly linked, so that the next and previous edges can be easily accessed. The process of building the edge list is as follows:

```
for each edge in the edge AVL tree:
    remove an edge from the edge AVL tree
    compute the radial angle with respect to the x-axis using the
        first endpoint as the origin of the edge
    store the edge according by ascending radial angle in the
        list pointed at by the slot at the index which
        corresponds to the first endpoint's identifier
    compute the radial angle with respect to the x-axis using
        the second endpoint as the origin of the edge
    store the edge according by ascending radial angle in the
        list pointed at by the slot at the index which
        corresponds to the second endpoint's identifier
endfor
```

2.4.2.3 Remove overlapping edges

As previously stated, the terrain database contains overlapping edges. These must be removed for the DCEL structure (see next section) to operate properly. The edge list built in the previous section provides easy identification of the overlapping edges, as well as a means for removing them.

Unfortunately, simply removing the overlapping edges will not necessarily preserve triangulation. Figure 2.4 (a) shows a situation where a problem may arise; in figure 2.4 (a), edge ac overlaps with edges ab and bc. Removing edge ac produces a quadrilateral abcd and so requires that edge bd be added to preserve the triangulation. The final product is shown in figure 2.4 (b) on the next page.

Note that both occurrences of edge ac must be removed from the edge list (there is one in the list for vertex a, and another in the list for vertex c). Similarly, edge bd must be inserted into the list for each of its endpoints, b and d.

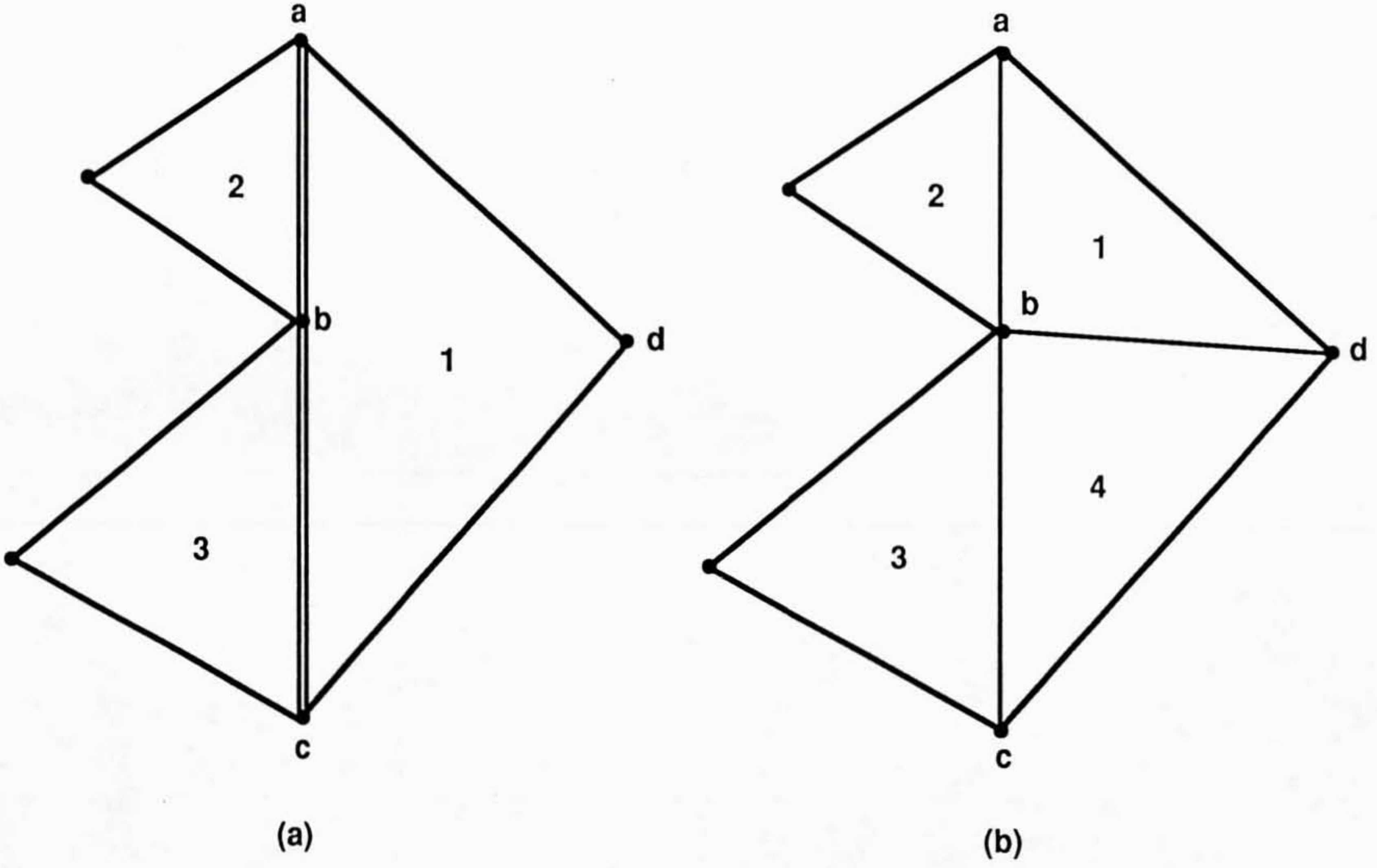


Figure 2.4. Remove edge ac and add edge bd to preserve triangulation.

With this in mind, the process of removing overlapping edges can be described. Overlapping edges have the same radial angle from a common vertex, and thus will be adjacent in that vertex's list of edges. Using the longer of the two overlapping edges, the next clockwise edge of one endpoint is compared with the next counterclockwise edge of the other endpoint, and vice versa. One of these comparisons produces a common vertex which defines an existing triangle with the endpoints of the longer overlapping edge. This common vertex will be used to divide the triangle into two smaller ones by adding an edge from the common vertex to the vertex of the shorter overlapping edge which is not an endpoint of the longer edge. The latter vertex will lie on the longer overlapping edge; it shall be called the overlapping vertex. The new edge must be inserted into the edge lists for each of its endpoints.

The longer overlapping edge cannot simply be removed, since it may still overlap one or more edges. Instead it is shortened to exclude the shorter overlapping edge, taking the overlapping vertex as its new endpoint. The original longer overlapping edge is then removed from the list of the vertex it originally shared with the shorter overlapping edge, and the modified edge is added to the list of its new endpoint, the overlapping vertex. The other vertex of the modified edge is unchanged, and thus no adjustment to its list is necessary. The modified edge can now be compared to the edges adjacent to it in the edge list to determine if it exactly overlaps (has the same endpoints as) another edge and thus can be removed, or if the process needs to be repeated for another overlapping edge.

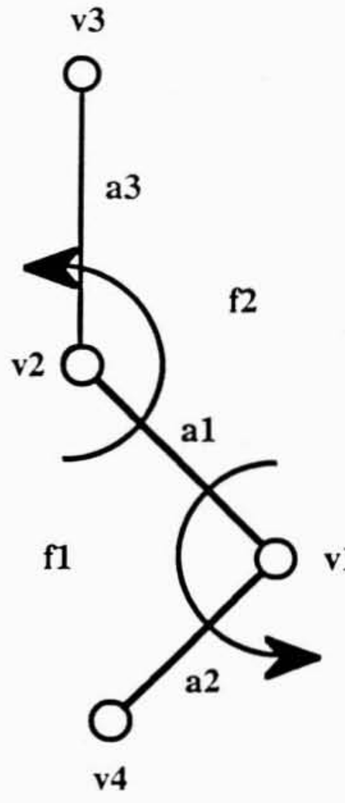
This process must be repeated until no overlapping edges remain in the edge list.

2.4.2.4 Build DCEL

With the overlapping edges removed, everything is in place to build the DCEL structure. The DCEL contains six fields for each edge in the edge list. The first field (V1 in the figure) contains the first endpoint of the edge. Likewise, the second field (V2) holds the second endpoint. (First and second endpoints are determined arbitrarily.) The third and fourth fields (F1 and F2) hold the identification numbers of the polygons which lie on the left and right side of the edge, with left and right determined by orienting the edge from the first endpoint to the second. (In a DCEL, polygons are usually referred to as faces; in this presentation, face and polygon are equivalent.) Similarly, the fifth and sixth fields (P1 and P2) hold the numbers of the first edges encountered when moving counterclockwise from the first and second endpoints, respectively. Figure 2.5 is an example of a DCEL.

The process of building a DCEL from an edge list begins with filling the vertex fields. This can be accomplished in a single pass through the edge list. Each edge has a number associated with it, which is used as an index into the DCEL. As an edge is encountered, its vertices are placed into the vertex fields V1 and V2 at that edge's index. Recall that the vertices were ordered as they were stored into the vertex AVL tree, so no conflict will arise from edges being in the edge list twice.

After the vertex fields have been filled, the edge pointer fields P1 and P2 can be entered. This is performed with a single pass over the DCEL. An edge is located in the edge list using the first vertex for that edge. The index of the next edge in the list will fill field P1, since it is the index of the first edge encountered when moving in a counterclockwise direction (the edges are sorted according to radial angle). Field P2 is similarly filled using vertex 2.



	V1	V2	F1	F2	P1	P2
1						
2						
⋮						
a1	v1	v2	f1	f2	a2	a3
⋮						
a2	v4	v1	f1			
⋮						
a3	v2	v3		f2		
⋮						

Figure 2.5 Example DCEL.

Filling the face (or polygon) fields F1 and F2 is slightly more complicated. The general procedure is to start with an edge, select an unnumbered face adjacent to it, assign that face a face number, and follow the edge pointers to assign the face number to all of the edges surrounding that face. The algorithm proceeds as follows:

```
For each edge in the DCEL:
    if  F1 of DCEL[edge] has not been assigned a number
        fill F1 of DCEL[edge] with currentFaceNumber
        vertex = V1 of DCEL[edge]
        nextEdge = P1 of DCEL[edge]
        while (nextEdge != edge)
            if  (V1 of DCEL[nextEdge] = vertex)
                fill F2 of DCEL[nextEdge] with currentFaceNumber
                vertex = V2 of DCEL[nextEdge]
                nextEdge = P2 of DCEL[nextEdge]
            else ( V2 of DCEL[nextEdge] = vertex )
                fill F1 of DCEL[nextEdge] with currentFaceNumber
                vertex = V1 of DCEL[nextEdge]
                nextEdge = P1 of DCEL[nextEdge]
            endif
        endwhile
        currentFaceNumber = currentFaceNumber + 1
    endif
endfor
```

It can be shown that if the F1 field of every edge is filled, then the F2 field of every edge must also be filled, so checking the F1 fields only will suffice.

2.4.2.5 Add feature segments

Section 2.4.2.1 stated that the polygons for features (treelines and canopies) are stored as edges in a list. For the purposes of LOS determination, only those features which lie on polygons traversed by the LOS need to be checked for obstructing the LOS.

The approach taken by this algorithm is to associate the feature edges with the terrain polygons they lie in. For this, a list of pointers is created with a slot for each face (or terrain polygon) in the DCEL. Each pointer points to a list of feature edges which lie on its associated face. The feature edges are stored as follows:

```
for each edge in the feature edge list
    determine the face containing the endpoints of the edge
    if  the endpoints lie in the same face
        store the edge in that face's feature list
    else
        find the intersection of the edge and the face
        divide the edge into two parts using the intersection
            point
        store the part of the edge which lies entirely with the
            face in that face's edge list
        repeat the process on the part of the edge which lies
            outside the face using the adjacent face
    endif
endfor
```

The adjacent face mentioned in the algorithm can be found using the DCEL, since the intersecting edge of the current face is known. The current face is one of the faces associated with the intersecting edge, the adjacent edge is the other.

2.4.2.6 Initialize slabs

The slab method for point location [Preparata,1988] was chosen for this project on the basis of execution speed; it provides a worst case point location time of O(log n). The slab method can be used on any planar straight-line graph (PSLG), which, as previously observed, includes reasonable polygonal terrain databases projected into the xy plane.

The basic idea of the slab method is to divide the graph into "slabs" by drawing a horizontal line through each vertex (see Figure 2.6). By sorting the slabs by y coordinate, the slab in which a query point lies can be found quickly with a binary search.

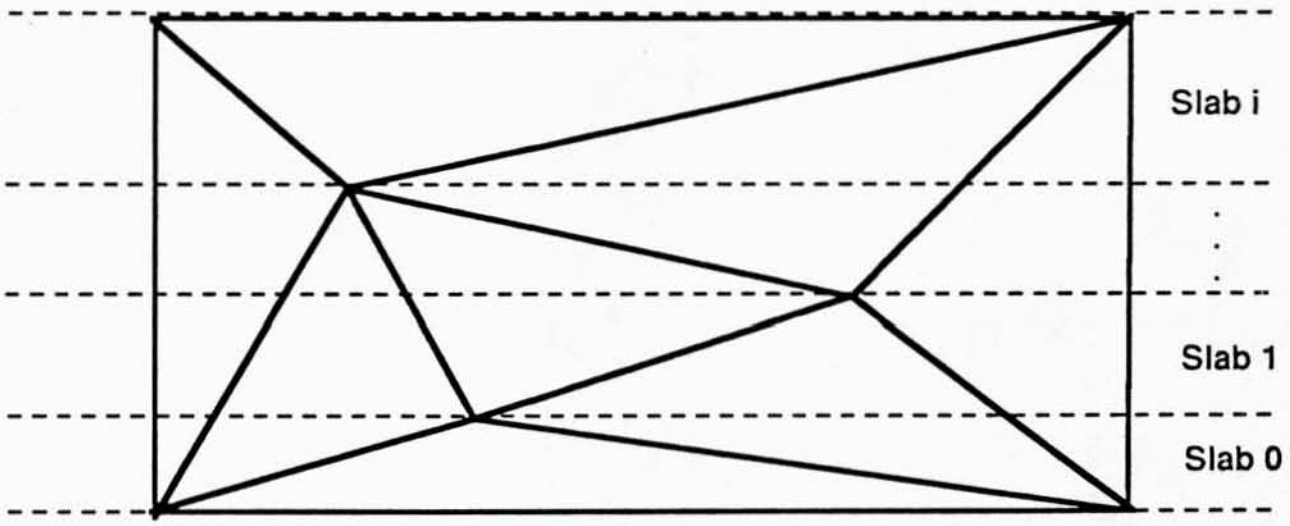


Figure 2.6 The vertices of the terrain polygons determine the horizontal slabs.

Each slab is partitioned by edges of the graph, or in this case, edges of the terrain polygons. These edges define trapezoids and triangles within the slab. Note that within a slab the edges cannot intersect, since each vertex defines a slab boundary. By sorting the segments within a slab by x value along one y boundary of the slab, the trapezoid or triangle within a slab in which a query point lies can again be found with a binary search.

The process of building the slabs is facilitated by structures and routines already in place. A planar scan process is used to build the slab data structure, where the scan's event points are the vertices. The sorted vertex list built from the vertex AVL tree (see section 2.4.2.2) holds the event points (vertices) for building the slabs. Another AVL tree is used to store the edges within the slabs in sorted order. At each event point (vertex), the incoming edges (radial angle > 180) are deleted from the tree and the outgoing edges (radial angle < 180 and not equal to zero) are inserted into the tree.

Horizontal edges are not used. For this process, the edge list described in section 2.4.2.2 is be used, since for each vertex, the edges which have that vertex for an endpoint are stored sorted by their radial angles. After a slab has been updated, the tree is output, producing the next slab. It is important to note that more than one vertex may have the same y-value, so all vertices with the same y-value must be processed before the slab is output.

2.4.3 Point location

To begin the LOS determination, it is necessary to identify the base polygons which contain the starting and ending sighting points, when they are projected into the terrain.

The point location data structures which will be used have been initialized by the preprocessing routines. To determine which terrain polygon a sighting point lies within requires two binary searches. The first search is on y values to determine which slab contains the point. The second search is on x values to find the edge within the slab the point lies just beyond (the slope of the edge within the slab is, of course, considered). Once the edge has been identified, a simple lookup in the DCEL will provide the number of the terrain polygon (face) in which the sighting point lies.

Thus locating the polygon in which a point lies requires two binary searches, one on the slabs' y coordinates, and one on the x coordinates of a single slab's partitioning edges; hence the O(log n) point location time.

2.4.4 LOS traversal

Having identified the start and finish polygons for the LOS, the DCEL is now used to identify all the base polygons through which the LOS passes (as described in section 2.1, this traversal is considered in 2D).

The LOS enters each traversed terrain polygon across one of its edges. Since terrain polygons are all triangles, and only straight lines are present, the LOS must leave through one of the remaining two edges of the triangle. The edge pointers P1 and P2 in the DCEL are used to determine the other two edges; the other two edges must share the same face as the edge through which the LOS entered. The vertical plane containing the LOS is tested for intersection with the other two edges using one of the intersection tests described in the following sections.

When an intersection is found between the LOS's vertical plane and the edge, the height values of the terrain edge and the LOS at the point of intersection are compared to determine if the LOS is blocked. If so, 0 is returned to indicate failure. Otherwise, the intersected edge is used to determine the next face to be entered by the LOS, and the traversal continues.

2.4.4.1 LOS and edge intersections

Intersections between the LOS and edges in the terrain database are determined by two routines. One routine calculates the intersection between a plane and a line segment, while the other calculates the intersection between two line segments. The plane-line intersection routine is used for the bulk of the intersection tests because it requires fewer arithmetic operations than the line-line intersection routine. In the plane-line intersection routine the vertical plane in which the LOS lies is tested for intersection with edges from the terrain database.

However, the plane-line intersection routine is not appropriate for situations where it may find an intersection beyond the LOS. This situation occurs in the start and finish polygons, where the plane-line intersection routine will find two intersections, one along the LOS, and one in the opposite direction which does not intersect the LOS. The same situation can occur with feature segments on the start and finish polygons. It is in these situations that the line-line intersection routine is used. Both of these routines are more fully discussed below.

2.4.4.1.1 Plane-line segment intersection

The plane-line segment intersection routine uses the normal to the vertical plane in which the LOS lies, which only need be computed once, and a point on the plane, which can be an endpoint of the LOS, to determine the intersection between the LOS and an edge of the terrain database. The method is as follows:

Parametric equations:

plane: $N_x(x-x_0) + N_y(y-y_0) + N_z(z-z_0)$

where N is the normal to the plane and $<x_0,y_0,z_0>$ is a point on the plane

line:
$$x = x_1 + (x_2 - x_1)t$$
$$y = y_1 + (y_2 - y_1)t$$
$$z = z_1 + (z_2 - z_1)t$$

where $<x_1,y_1,z_1>$ and $<x_2,y_2,z_2>$ are endpoints of the line segment.

Using the parametric equations for the plane and for the line determined by the endpoints of the line segment, solve for t. If

$0 <= t <= 1$, the line segment intersects the plane, and the parametric equations can be used to compute the point of intersection.

2.4.4.1.2 Line segment-line segment intersection

Using the parametric equations for the LOS and an edge in the terrain database, the following method is used to determine their intersection:

Parametric equations:

$$x_1 = a_x + (b_x - a_x)s$$
$$y_1 = a_y + (b_y - a_y)s$$
$$z_1 = a_z + (b_z - a_z)s$$

where $<a_x,a_y,a_z>$ and $<b_x,b_y,b_z>$ are endpoints of a line segment.

$$x_2 = u_x + (v_x - u_x)t$$
$$y_2 = u_y + (v_y - u_y)t$$
$$z_2 = u_z + (v_z - u_z)t$$

where $<u_x,u_y,u_z>$ and $<v_x,v_y,v_z>$ are endpoints of a line segment.

Solve for s and t by setting $<x_1,y_1,z_1>$ equal to $<x_2,y_2,z_2>$. If $0.0 <= s,t <= 1.0$ the line segments intersect. The point of intersection can then be computed by substituting either s or t into the corresponding parametric equations.

2.4.4.2 Check features of polygon

If the LOS is established across a base polygon, the features (treelines and canopy polygons) which are on that polygon are checked. The feature polygons for a given terrain polygon, if any, have been stored as edges in a list for that terrain polygon. The LOS is then checked for intersection with each of these edges, using the intersection routines (see the previous section). The actual intersection location is computed and compared with the LOS coordinates at that location only if an intersection occurs. Again, if the LOS is found to be blocked, the LOS determination process immediately terminates, indicating a failure to establish a LOS by returning 0.

2.4.4.3 Terminate when finish polygon is reached

When the current polygon location is found to be the finish polygon, and no features of that polygon block the LOS, the LOS has been established. In this case, the algorithm indicates that the LOS exists, i.e. is not blocked, by returning 1.

2.4.5 Order analysis

The time and space complexity of several components of the overall algorithm are given in this section.

Point location:
  $O(\log n)$ time, $O(n^2)$ space where n = number of vertices.
Line Intersection:
  $O(1)$ time per polygon.
Determine next polygon:
  $O(1)$ time per polygon.
Features:
  $O(m)$ checks per polygon, where m = number of feature segments in the polygon.

2.5 IST algorithm P: Triangle Traversal method

2.5.1 Overview

This section describes the Triangle Traversal method and its implementation. The basic idea of this method is to triangulate the polygonal terrain, and then construct a simple data structure which stores the three neighbors for each terrain triangle. This triangle list can be used to quickly trace the LOS from one triangle to the next.

Section 2.5.2 discusses the preprocessing required by this algorithm. Sections 2.5.3 and 2.5.4 present the point location and LOS traversal steps, which are part of the run-time LOS determination.

2.5.2 Preprocessing

The primary goal of the preprocessing step is to create the data structures used to facilitate the point location and LOS traversal algorithms. Point location is used to determine the triangles within the terrain which contain the starting and ending points of the LOS. The LOS traversal algorithm is used to determine if, given two endpoints in 3-space, there is an obstruction between them. This obstruction could be terrain, or some feature on the terrain.

The main data structure created during preprocessing consists of a list of triangles, referred to as the Triangle list. A triangle on the Triangle list stores the following information:

```
Vertex 1, 2, 3;
Neighbor 1, 2, 3;
Features;
```

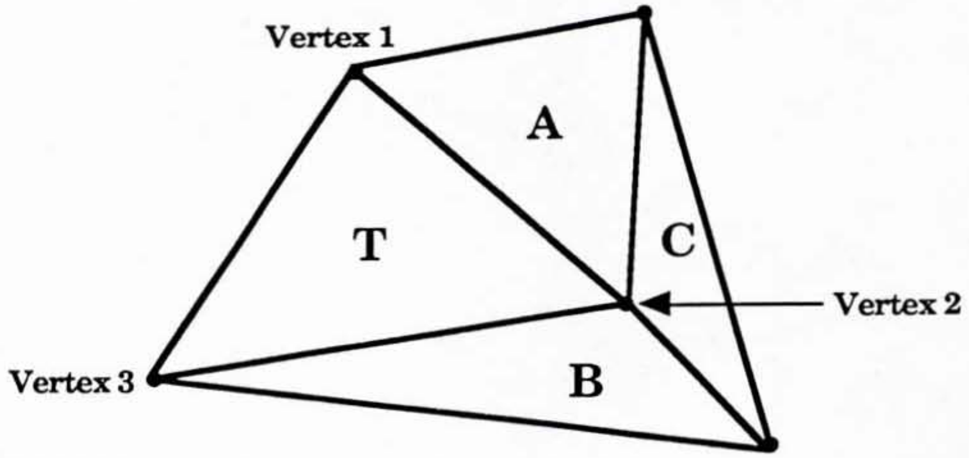


Figure 2.7. Triangle T and its neighbors.

Vertex 1, 2, and 3 are the triangle's associated vertices in clockwise order. Neighbor 1 is a pointer to the triangle's neighbor along its 1-2 edge; similarly, Neighbor 2 is a pointer to the triangle's neighbor along its 2-3 edge, and Neighbor 3 is a pointer to the neighbor along the 3-1 edge. Vertex 1, 2, and 3 are stored as indices into a master list of vertices, referred to as the Vertex list, and Neighbor 1, 2, and 3 are stored as indices into the Triangle list. Figure 2.7 shows triangle T, where T's Neighbor 1 is triangle A, T's Neighbor 2 is triangle B, and T's Neighbor 3 is non-existent (indicated by storing '-1' as Neighbor 3's index).

In order for there to exist exactly one Neighbor along each edge of each triangle, it is imperative that any two triangles which have an overlapping edge share exactly the same edge. Consequently, given the four polygons shown in Figure 2.8 to be triangulated, Triangulation 1 is incorrect, while Triangulation 2 is correct. Triangulation 1 is incorrect since triangle F shares its right edge with triangles H, J, and L.

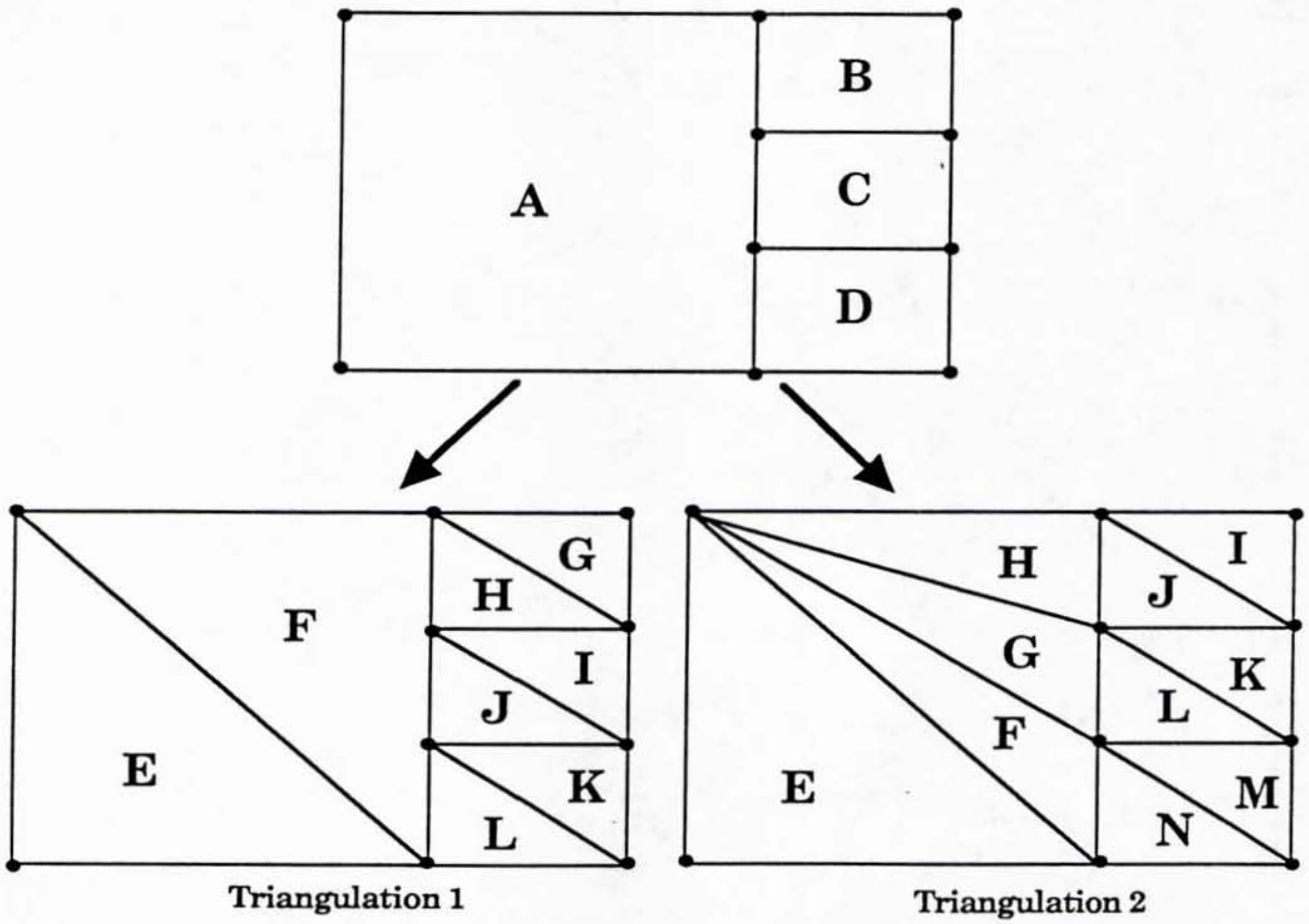


Figure 2.8. Triangulation 1 is incorrect, Triangulation 2 is correct.

The last piece of information stored in a triangle, Features, is a pointer to a list of feature edges which reside on the given triangle. Given SIMNET terrain, such features consist of treelines and canopies.

The Triangle list is used during both the point location and the LOS traversal algorithms. Additional data structures created include the Vertex list and the Y-Slab list as described in section 2.5.2.1. The sorted Vertex list is used both for point location and to reduce the amount of storage used. The Y-Slab list is used for point location and to facilitate the triangulation of the terrain base polygons.

The following sections describe the process of reading in the terrain polygons and how the information is stored.

2.5.2.1 Create Vertex list and Y-Slab list

The polygons are read and processed one at a time, in multiple passes. Each polygon is one of the following types: base, road, river, object, treeline, canopy cover, and canopy treeline. The base polygons store the actual terrain, while the other polygons are features on the terrain. The purpose of the first pass over the input polygons is to create the Vertex list, where vertices are stored in ascending y and x coordinates, respectively. This is accomplished by inserting each base polygon vertex into an AVL tree. Once the AVL tree is created, the sorted vertices are read from the tree and stored in a sorted array, the Vertex list, for direct access. The AVL tree is then deleted.
As the Vertex list is being created from the AVL tree, an additional structure is also created for the point location process, called the Y-Slab list. The Y-Slab list serves to partition the terrain into horizontal slabs. Each entry in the Y-Slab list stores two items, beginV and endV, which are indices into the sorted Vertex list. For each Y-Slab in the Y-Slab list, the vertices from beginV to endV all have the same y coordinate and ascending x coordinates. The identifying y coordinate of each Y-Slab in the Y-Slab list is unique and the Y-Slabs in the list are ordered in ascending y coordinates. Figure 2.9 shows a small set of base polygons of terrain and its associated Y-Slab list.

In addition, each vertex in the Vertex list stores an index into the Y-Slab list indicating which Y-Slab it belongs to.

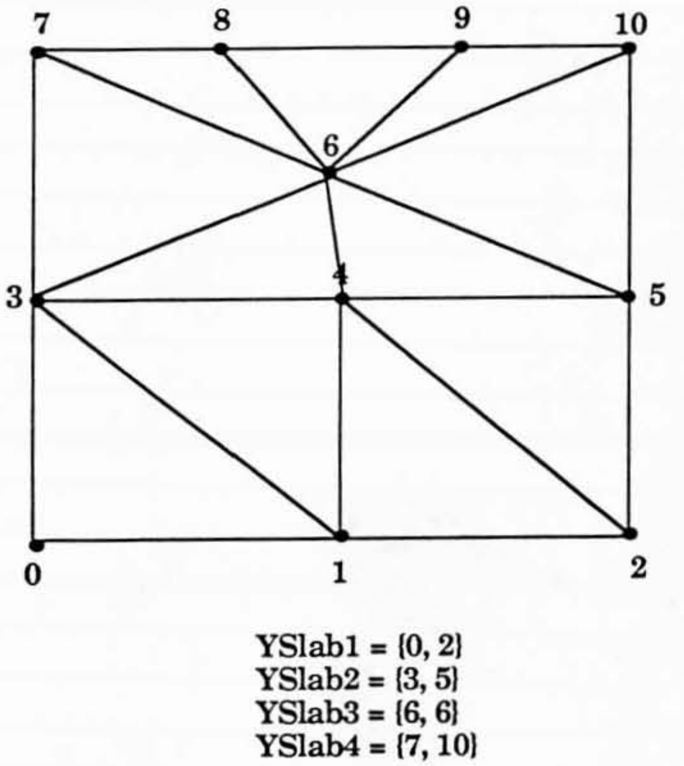


Figure 2.9. Example Y-Slab lists.

2.5.2.2 Create Triangle list

The primary purpose of the second pass over the input polygons is to create the Triangle list, where the triangles are constrained as described in section 2.5.2. The polygons are again read in, one at a time, from the terrain database file. Road, river, and object polygons are discarded. Treelines and canopy polygons are stored as edges in a list to be handled after the Triangle list is created, as described in section 2.5.2.4. The base polygons are processed to create the Triangle list, where a SIMNET base polygon is known to be have either 3 or 4 sides. A 3-sided base polygon is input to the triangulation routine as is; a 4-sided base polygon is first split into two 3-sided base polygons, and each 3-sided base polygon is then input to the triangulation routine. The triangulation routine can be summarized as follows:

```
1. Sort the base polygon's 3 vertices in clockwise order.

2. Set edgeVertices = set of all vertices which lie on polygon's
   edges, other than its original three vertices.
   (This is performed using the ordered Vertex list, the Y-Slab
   list, and each vertex's pointer to its associated Y-Slab.)

3. If  edgeVertices = the empty set then
        Create a new triangle T from the base polygon
        Add T to the Triangle list
        Return
   else
        Continue with Step 4
   endif

4. Sort the polygon vertices and edgeVertices in clockwise order.

5. Using the ordered vertices, triangulate polygon, adding each
   new triangle to the Triangle list.
```

Figure 2.10 shows the triangulation of 3-sided base polygon P (P1, P2, P3), as would occur in Step 5 of the above routine. The triangles T_j, T_{j+1}, T_{j+2}, and T_{j+3} would be added to the Triangle list.

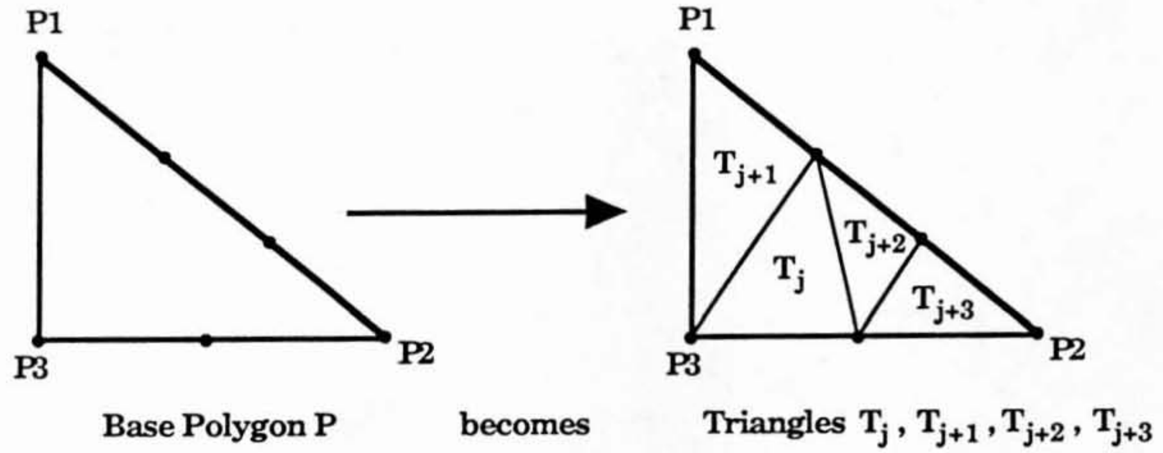


Figure 2.10. Triangulation of polygon P.

A vertex triangle list is associated with each vertex, and contains the index of each of the triangles which use the vertex. As each new triangle T_i is added to the Triangle list, each of its vertices is updated with the addition of T_i's index i to the vertex's vertex triangle list. The vertex triangle lists are used to help set each triangle's neighbors, as described below, and during the point location process.

2.5.2.3 Set triangles' neighbors

At this point, the entire Triangle list has been created; however, many of the triangles' neighbors have not yet been set. This is because new triangles are often added to the Triangle list with either some or all of their neighbors not yet determined. In fact, the only time a triangle is added with its neighbors predetermined is when it is created in Step 5 of the above triangulation routine. Consequently, once the Triangle list is created, a pass is made over this list to set the neighbors of each triangle as required. This can be accomplished quite simply, since each vertex stores a list of all triangles which use that vertex, its vertex triangle list. For instance, to set Neighbor 1, the neighbor along the 1-2 edge of triangle T, the algorithm simply looks through vertex 1's vertex triangle list searching for a triangle which uses vertex 2.

2.5.2.4 Add feature segments

Section 2.5.2.2 stated that features are stored as edges in a list to be handled after the Triangle list was created. For the purpose of the LOS traversal, only those features which lie on triangles crossed by the LOS need to be checked for obstructing the LOS.

The approach taken in this algorithm is identical to the approach used in algorithm C (see section 2.4) wherein feature edges are associated with the terrain triangles they lie in. The following is the method used for incorporating the feature edges into the Triangle list.

```
for each edge in the feature edge list
    point locate the endpoints of the edge
    if  the endpoints lie in the same triangle
        store the edge in that triangle's Feature list
    else
        find the intersection of the edge and the triangle
        divide the edge into 2 parts using the intersection point
        store the part of the edge which lies entirely in the
            triangle in the triangle's Features list
        repeat the above process on the part of the edge which
            lies outside of the triangle, using its neighbor
            along the given edge
    endif
endfor
```

This completes the preprocessing for the Triangle Traversal method.

2.5.3 Point location

Point location is required to determine the triangles which contain the starting and ending points of the LOS. Given point p to be located, the method used for point location is as follows:

1. Locate the bounding upper and lower Y-Slab for point p. (Requires one binary search.)
2. Let v1 = the vertex closest to p on the lower Y-Slab.
   Let v2 = the vertex closest to p on the upper Y-Slab.
   Let v = the closer of (v1, v2) to p.
   (Requires two binary searches.)
3. Test each triangle T in v's vertex triangle list to determine if p is included in T. If found, return triangle T's index, otherwise continue to Step 4.
4. Follow line from vertex v to point p, using the Triangle list to determine each next triangle along the line vp to be tested for point inclusion of p.

The method used in Step 4 to go from triangle to triangle is almost identical to the LOS traversal algorithm's method for stepping through the triangles with the pleasant exception that there is no "special case" as described in section 2.5.4.1.

2.5.4 LOS traversal

The LOS traversal algorithm accepts as input the LOS starting and ending points, along with the triangles T_{start} and T_{end} which contain them, and returns 1 (not blocked) if a LOS can be established between the two points or 0 (blocked) otherwise. The general idea is to start at triangle T_{start} and check every triangle along the LOS to determine if the LOS is blocked, either by the triangle itself, or by a Feature edge on the triangle. This process is continued until either a blockage is found or T_{end} is reached.

To determine if the triangle itself blocks the LOS, the algorithm projects the LOS and the triangle to 2D, ignoring z coordinates. It then determines the location where the LOS intersects an edge of the triangle in 2D. Then, given the (x, y) location of the intersection point, it computes and compare the z-values for the LOS and the intersected edge to determine if the edge is above the LOS, indicating an obstruction. If the triangle itself does not create an obstruction, then each of its associated Feature edges is checked to determine if any of them create an obstruction. If no obstruction is found, then the "next" triangle that the LOS crosses is examined, and so on, until either an obstruction is found or the algorithm reaches the last triangle, T_{end}.

The "next" triangle along with which of its edges is intersected by the LOS is determined using the current triangle's intersected edge, along with its third vertex as described in the next section.

2.5.4.1 Stepping from triangle to triangle along the LOS

Assume that the algorithm is currently examining triangle T_i which is intersected by the LOS L = QR (Q and R are the endpoints, i.e. the sighting and target points). Call the vertex above the LOS on the intersecting edge A, and the vertex below the LOS on the intersecting edge B, and the remaining vertex on the current triangle C. (The LOS and the triangles have been projected into the xy plane, so in this section *above* means have a larger y coordinate, and *below* means having a smaller y coordinate. Above and below indicate relative postions when drawn in a conventional 2D coordinate system.) The algorithm will determine the next intersected edge and its intersection point, P_{n+1}, and the next triangle T_{i+1}.

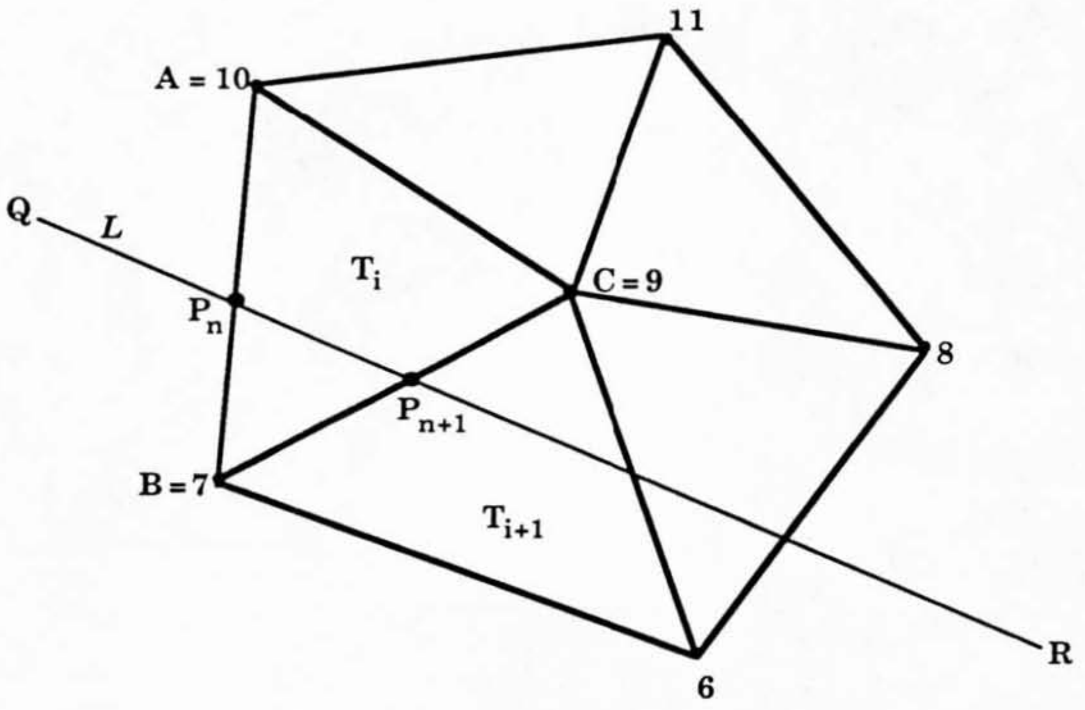


Figure 2.11. Finding the next triangle (the values on the vertices are their y coordinates).

These determinations can be made as follows, where L is the LOS:

```
case C above (in 2D) L: (as in Figure 2.11)
        $P_{n+1}$ = L intersect Edge BC (in 2D)
        check $P_{n+1}$ to see if LOS obstructed (in 3D)
        $T_{i+1}$ = $T_i$'s Neighbor along edge BC
        A = C B = B, C = $T_{i+1}$'s vertex other than A and B
case C below L:
        $P_{n+1}$ = L intersect Edge AC (in 2D)
        check $P_{n+1}$ to see if LOS obstructed (in 3D)
        $T_{i+1}$ = $T_i$'s Neighbor along edge AC
        A = A, B = C, C = $T_{i+1}$'s vertex other than A and B
case C on L:
        $P_{n+1}$ = C
        check $P_{n+1}$ to see if LOS obstructed (in 3D)
        handle "special case" for determining A, B, C, and $T_{i+1}$
```

Once T_{i+1} has been identified, check T_{i+1}'s Features to determine if any features constitute an obstruction.

As each of the LOS checks is performed, if an obstruction is found, then the LOS traversal algorithm returns a 0 (blocked) and terminates. The above sequence is repeated until either an obstruction is found or $T_{i+1} = T_{end}$.

Note that P_{n+1} is calculated in 2D, that is, its (x, y) value is determined. The check on P_{n+1} involves calculating the z-value of that (x, y) point on the LOS, call this LOS_z, and calculating the z-value of the same (x, y) point on the triangle's intersected edge (BC or AC), call this T_z. If T_z is greater than LOS_z, the LOS is blocked.

The following shows how the "special case", where vertex C on Line L=QR, is handled.

```
while (edge E = CC' is collinear with L and C' between C and R)
    check C' to see if LOS obstructed
    check Features for triangles on both sides of CC' for
        obstruction
    C=C'
endwhile
```

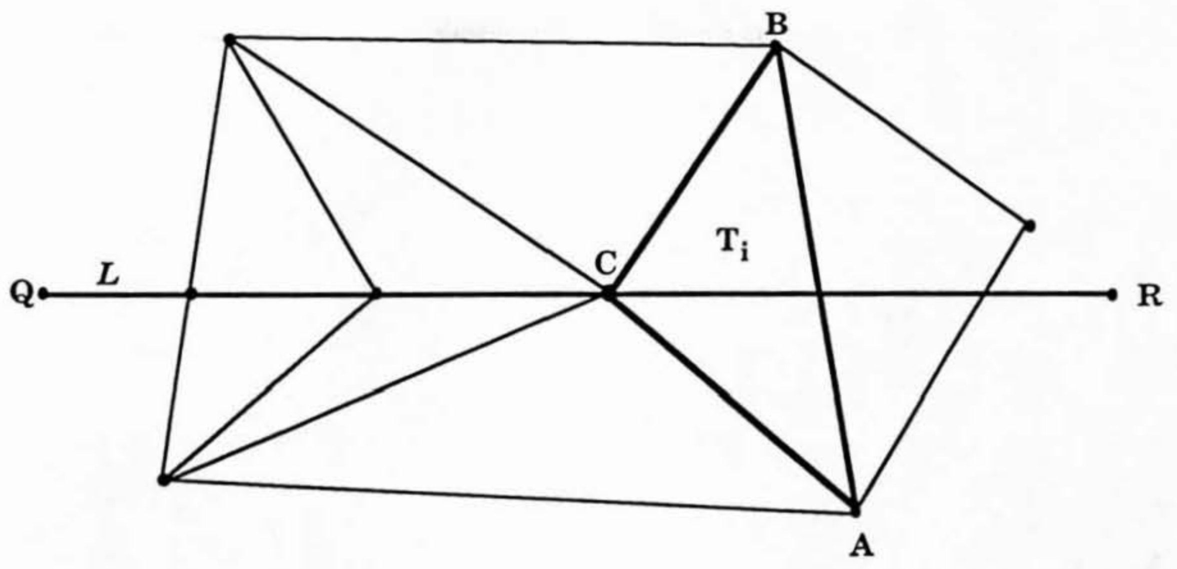
At this point it is known that no edge emanating from C is collinear with L=QR, and so one of following two cases is true:

(i) One of the triangles of which C is a vertex is the last triangle T_{end}.

or

(ii) One of the triangles of which C is a vertex intersects L, both at the point C and somewhere along the edge AB where A and B are the two vertices, other than C, on the triangle.

If case (i) is true, then the LOS traversal is complete. Otherwise, it is necessary to search through vertex C's triangles, found in its vertex triangle list, to find the triangle T which intersects L with its edge AB, where A and B are the two vertices, other than C, on triangle T. The algorithm then tests the newly found intersection point on segment AB and also tests the Feature edges of triangle ABC for an obstruction. Since identifying one vertex as A and one as B is determined arbitrarily, once the edge AB which intersects L is found, A is set to be the vertex above L and B to be the vertex below L. The next triangle, T_{i+1}, is set to the neighbor of the triangle ABC along the AB edge. The new C is set to be T_{i+1}'s vertex other than A and B. Figure 2.12 illustrates this process.



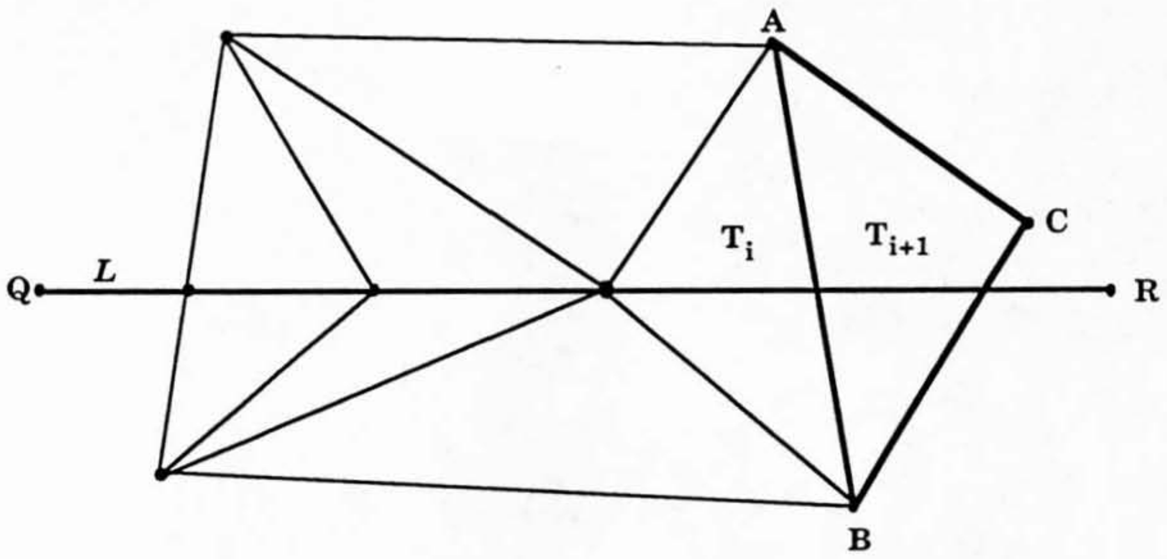


Figure 2.12. C lies on the LOS.

At this point the handling of the special case is complete; the algorithm is ready to test if the current vertex C is above, below, or on the line L=QR and continue normally.

As mentioned above, the algorithm traces from triangle to triangle until either an obstruction is found or triangle T_{end} is reached. If T_{end} is reached, the algorithm returns a 1, indicating that the LOS is unblocked.

2.5.5 Geometric support routines

The point location and LOS determination algorithms are highly dependent on a core set of 2D and 3D geometry routines. In order for the top level algorithms to be as efficient as possible, it is very important that these core routines be extremely fast. Although the LOS traversal is intrinsically a 3D problem, almost all calculations can be made in 2D, increasing efficiency. In fact, the only time a 3D calculation is made is to determine the z-value of an (x, y) point of intersection of the LOS and a triangle edge as explained in section 2.5.4.1.

The following gives a brief description of the core set of geometry routines.

2.5.5.1 Left turn (2D)

Given three points P, Q, R,

```
return  1 if P, Q, R form a left turn
        0 otherwise.
```

```
if ((($Q_x$ - $P_x$) * ($R_y$ - $P_y$) - ($R_x$ - $P_x$) * ($Q_y$ - $P_y$)) > 0)
        return [1];
else
        return [0];
endif
```

The above routine is used to determine if a point is located within a specified triangle in 2D. This is based on the fact that point P is contained in triangle QRS iff QRP, RSP, and SQP each form a left turn where Q, R, and S are counter-clockwise. The routine's mathematical equation is determined using the fact that given the points P, Q, and R, then they form a left-hand turn if and only if the following determinant is greater than zero [Preparata,1988]:

$$\begin{vmatrix} P_x & P_y & 1 \\ Q_x & Q_y & 1 \\ R_x & R_y & 1 \end{vmatrix}$$

2.5.5.2 Point on line (2D)

Given two points P and Q and a test point T

```
return  0 if T is not on the (infinite) line PQ
        1 if T is on the open ray P
        2 if T is within the line segment PQ
        3 if T is on the open ray Q
```

```
if  ABS(($Q_y$ - $P_y$) * ($T_x$ - $P_x$) - ($T_y$ - $P_y$) * ($Q_x$ - $P_x$)) >=
    MAX(ABS($Q_x$ - $P_x$), ABS($Q_y$ - $P_y$))
    return[0];
if ($Q_x$ < $P_x$ and $P_x$ < $T_x$) or ($Q_y$ < $P_y$ and $P_y$ < $T_y$) return [1];
if ($T_x$ < $P_x$ and $P_x$ < $Q_x$) or ($T_y$ < $P_y$ and $P_y$ < $Q_y$) return [1];
if ($P_x$ < $Q_x$ and $Q_x$ < $T_x$) or ($P_y$ < $Q_y$ and $Q_y$ < $T_y$) return [3];
if ($T_x$ < $Q_x$ and $Q_x$ < $P_x$) or ($T_y$ < $Q_y$ and $Q_y$ < $P_y$) return [3];
return [2];
```

The above routine is used during preprocessing to aid in the triangulation process. Specifically, it is used in Step 2 of the triangulation routine in section 2.5.2.2.

This routine was written by Alan W. Paeth of the University of Waterloo and can be found in [Glassner,1990].

2.5.5.3 Point above/below/on line segment (2D)

Given the point P and line segment LS with slope m, intercept b, and lineType equal to VERTICAL or REGULAR,

```
return  -1 if P below LS
         1 if P above LS
         0 if P on LS.
```

(Note: 'b' is a y-intercept, unless LS is of lineType VERTICAL, in which case 'b' is an x-intercept.)

```
if  (lineType is VERTICAL) then
    if  (the real numbers Px and b are "equal")
        return [0];
    else
        if  ( Px > b )
                return [-1];
        else
                return [1];
        endif
    endif
else
    onLineValue = m * Px + b;
    if  (the real numbers Py and onLineValue are "equal")
            return [0];
    else
        if  (Py > onLineValue)
                return [1];
            else
                return [-1];
        endif
    endif
endif
```

The above routine is used repeatedly to determine if a particular point is above, below, or on the LOS. Specifically, these are the three cases tested for as described in section 2.5.4.1. Since the LOS does not change, the values of m, b, and lineType are computed once and each call to the above routine requires at most one multiplication and one addition.

2.5.5.4 Point of intersection between line segments (2D)

Given the parametric equation of one line segment LS1:

$$\langle R_x, R_y \rangle = \langle U_x, U_y \rangle + t * \langle V_x, V_y \rangle$$

and endpoints A, B of the second line segment LS2, where LS1 and LS2 are known to intersect, set the intersection point P of LS1 and LS2.

$$P_x = A_x + (B_x - A_x)((A_y - U_y)*V_x - (A_x - U_x)*V_y)/((B_x - A_x)*V_y - (B_y - A_y)*V_x);$$

$$P_y = A_y + (B_y - A_y)((A_y - U_y)*V_x - (A_x - U_x)*V_y)/((B_x - A_x)*V_y - (B_y - A_y)*V_x);$$

The above line segment intersection routine is used repeatedly to determine the point of intersection between the LOS and a crossed triangle edge. Again, since the LOS does not change, its parametric equation is computed once and repeatedly input as LS1. LS2 is the triangle's edge that is known to intersect the LOS as described in section 2.5.4.1.

2.5.5.5 Calculate z-value of (x, y) point on line segment (3D)

Given the parametric equation of line segment L

$$\langle R_x, R_y, R_z \rangle = \langle Q_x, Q_y, Q_z \rangle + t * \langle M_x, M_y, M_z \rangle$$

where L is known to be NOT perpendicular to the xy plane, and Point P, where P_x and P_y are already determined,

set P_z such that P is on the line L.

```
if  ($M_x$ does not equal 0.0)
    $P_z = Q_z + (P_x - Q_x) / M_x * M_z;$
else
    $P_z = Q_z + (P_y - Q_y) / M_y * M_z;$
endif
```

This is the only 3D geometry routine. It is used repeatedly to determine the z-value of an (x, y) location on the LOS during the LOS traversal algorithm. One test is made to determine if the LOS is perpendicular to the xy plane before a call to this routine is ever made. Thus some operations are saved since it is known that if M_x is zero and L is NOT perpendicular to the xy plane, then M_y is non-zero.

A separate, almost identical, routine is used to calculate the z-value of an (x, y) point on an arbitrary line segment, and is used to calculate the z-value of an (x, y) point on a triangle edge during the LOS traversal algorithm. The body of this slightly modified routine is:

```
if ($M_X$ does not equal 0.0)
    $P_Z = Q_Z + (P_X - Q_X) / M_X * M_Z$;
else
    if ($M_Y$ does not equal 0.)
        $P_Z = Q_Z + (P_Y - Q_Y) / M_Y * M_Z$;
    else
        if ($M_Z > 0.0$)
            $P_Z = Q_Z + M_Z$;
        else
            $P_Z = Q_Z$;
        endif
    endif
endif
```

The third branch occurs if the line segment is perpendicular to the xy-plane, which could theoretically occur for a given triangle edge. In this case, the z-value is assigned the greater z-value of the two endpoints on the line segment.

This concludes the set of support geometry routines.

2.5.6 Order analysis

The following summarizes the time and space order analysis for the point location, and the LOS traversal.

2.5.6.1 Point location

Given terrain with mostly uniformly spaced vertices, as is true for SIMNET terrain, the point location algorithm requires O(log n) time for the average case, where n is the number of vertices. It does not require $O(n^2)$ storage as does the more popular slab method [Preparata,1988], but instead requires O(n) storage. The downfall of this method is that a worst case could require O(n) time, where n is the number of polygons in the terrain database, which in some applications may be intolerable.

2.5.6.2 Line of sight determination

O(1) time is required to process each triangle along the LOS, since constant time is required to determine which edge is intersected, the intersection point, and the next triangle to be examined. Even when handling the "special case" where the vertex C is on line L, if it can be assumed that there is some maximum number of triangles emanating from any one vertex, then the special case still requires constant time. However, the coefficient of this constant will probably be higher when handling the special case.

Since each Feature segment requires O(1) time to process, then given the number of Feature segments for a particular triangle is r, the time to check Feature segments for that triangle is O(r).

The space required is O(n) for each of the vertex list, Triangle list, and Y-Slab list, where n is the number of vertices.

2.6 BBN SIMNET PVD algorithm

2.6.1 Overview

The BBN SIMNET Planview Display LOS algorithm (hereinafter called the PVD algorithm) determines point to point LOS on the SIMNET terrain database. This algorithm determines if a LOS is blocked by:

1. base polygons, which may form mountains, hills, and valleys;
2. objects, such as towers, buildings, telephone poles;
3. trees;
4. treelines;
5. canopies; and
6. vehicles along the LOS.

The PVD algorithm also determines if there are any terrain features behind the target location along an extended LOS. These background features are termed "clutter" and while clutter does not block LOS other algorithms may use clutter to obscure a target.

This algorithm was not developed by IST as part of this research project; consequently, it will not be explained in as much detail as the other LOS algorithms. More information on this algorithm is available in [BBN,1991].

2.6.2 Preprocessing

Because this algorithm uses the SIMNET terrain database, the preprocessing step consists of assembling that database from the polygon set. That process has been previously discussed, as well as the format of that database, in section 2.3.

2.6.3 LOS determination

The PVD algorithm consists of 5 major steps. First, the necessary data elements and structures are initialized. This involves determining the starting location's patch, determining the LOS's slope, determining vertical and horizontal distances for stepping through the patches, and a variety of support data.

The second step in the algorithm determines if the minimum elevation of any patch along the LOS is greater than the LOS in that patch. This constitutes a relatively quick check for a relatively large LOS blockage by the terrain. If the minimum elevation within a patch is greater than the LOS in the patch, then all terrain feature in that patch is higher than the LOS and something in the patch must block the LOS. If such a patch is found, the algorithm terminates with the LOS blocked.

If the second step does not find the LOS blocked, the third step in the algorithm is performed. The PVD algorithm checks all vehicles to determine if any vehicle intervenes between the starting location and the target location. Each vehicle is modeled as a sphere rather than a silhouette or cutout of its actual shape. This step excludes the vehicles at the starting and target locations so that the sighting vehicle and target vehicles do not block the LOS. If a vehicle blocks the LOS, the algorithm terminates with the LOS blocked.

If LOS is still open, the fourth step in the algorithm is performed. This step does the most exhaustive analysis of the terrain. The PVD algorithm moves through each patch along the LOS. For each patch, the algorithm checks for five types of LOS blocks.

1. All the edges in the patch are checked to determine if any edge intersects the LOS in 2D (x and y, not elevation) and is higher than the LOS. If so, the LOS is blocked.

2. Each object within the patch is checked to determine if the LOS traverses the "box" defining the outline of the object.

3. The algorithm checks all trees within the polygon to determine if any individual tree blocks LOS.

4. The algorithm checks each treeline in the polygon. Each treeline is checked in two ways. First, the treeline as a vertical polygon is checked and second, the trees at each end of the treeline are checked. A LOS may, from certain angles, intersect both a tree and the treeline.

5. The algorithm checks each canopy within the patch to determine if any canopies obscure the LOS.

The fifth step in the PVD algorithm checks for "clutter" by extending the LOS beyond the target location and checking the minimum elevation of each patch beyond the target against the LOS at that patch. If the minimum elevation within a patch is greater than the LOS, the algorithm terminates, indicating that the LOS has "clutter". This check is similar to the second step except that patches beyond the target rather than between the starting location and target are checked.

The PVD algorithm has three broad classes for LOS determination. A LOS may be unblocked and uncluttered, unblocked and cluttered, or blocked. Blocked LOSs are further divided into a number of categories indicating what kind of blockage occurred, e.g. by terrain, tree, treeline, etc and whether the blockage is complete or partial.

2.6.5 Experimental modifications

The IST LOS algorithms explained in the previous sections consider only terrain, treelines, and canopies. To provide a suitable comparison to the IST algorithms, the PVD algorithm was modified by IST to not perform the checks for:

1. objects,
2. intervening vehicles, and
3. clutter.

3. LOS comparison experiment

3.1 Discussion

The LOS comparison experiment was simple in concept. All four LOS algorithms (the three developed at IST and the BBN PVD algorithm) were used to preprocess the same set of input terrain polygons, and then to process the same set of LOS test cases. A common shell program was written to call the preprocessing routine for each algorithm, to call the LOS determination routine for each LOS test case, and to track the time required for each step of each algorithm.

For the experiment, all four algorithms were compiled using the Borland C++ 2.0 compiler, under the large memory model. Although a C++ compiler was used, all algorithms were written in ANSI C, without using any C++ features. The timing runs were executed on a Hewlett Packard Vectra RS/25C, equipped with an Intel 80386 CPU and an Intel 80387 math coprocessor. For this experiment, the CPU clock speed of the Vectra was set to 4.77 Mhz; this was done because we were interested not in absolute speed but in relative speed of the algorithms and a slower CPU clock speed would magnify efficiency differences between the algorithms.

For the preprocessing step, the entire input terrain polygon file (see section 3.2) was read from disk and saved in internal memory before the preprocessing routine was called and the timer started. By doing so, the preprocessing times do not include any disk I/O time. A similar approach was used for the LOS test cases (see section 3.3); all were read into internal memory before processing to avoid disk I/O during the timed portion of the runs.

Section 3.4 gives the actual run times for the various algorithms during the experiment. The times given in section 3.4 are for the processing of the entire LOS test case set, not a single LOS determination. Section 3.3 gives more detail on the LOS test case set.

3.2 Test polygonal terrain

The test polygonal terrain database used for this project is a subset of the standard Ft. Knox KY SIMNET terrain database. All of the base, treeline, and canopy polygons from a 1500 meter square portion (that is, a 3x3 set of terrain patches) of the database were extracted and converted into the generic polygon file which is the input to the preprocessing for the LOS algorithms.

Note that in the case of Algorithm F, which uses the SIMNET terrain database as its LOS data structure, the generic polygon file was converted back into SIMNET format by its preprocessing step. This was done so as to measure the time required for that preprocessing, i.e. the time required to build a SIMNET terrain database.

The polygon file contains 281 polygons, including 270 base polygons, 7 treelines, 2 canopy treelines, and 2 canopy polygons. The southwest corner of the 9 patch square used is located at point (39500, 39500) from the southwest corner of the standard Ft. Knox KY SIMNET terrain database.

3.3 Test LOS cases

A thorough set of test cases was developed for use in testing the LOS algorithms. The individual test cases (pairs of entities on the terrain) were be selected to exercise all aspects of LOS determination, according to the principles of systematic test case design [Myers,1979].

The set of test cases included the following situations, as well as others:

1. unblocked LOS
2. LOS blocked by base polygons
3. LOS blocked by treelines
4. LOS blocked by canopies
5. blocked and unblocked LOSs which were collinear with one or more polygon edges.

Each record in the test case file contained, in addition to the LOS endpoints, the test case number, a brief description of the test case, and the expected result of the test case (either 1 not blocked or 0 blocked). The LOS test shell program compared the answer returned by the LOS algorithms with the expected result in the test case and aborted the run if they did not match.

During the LOS determinations, each test case was considered in both directions; in other words, given a test case with endpoints p_1 and p_2, the LOS from p_1 to p_2 and the LOS from p_2 to p_1 were both checked.

3.4 Experimental results

The following execution times were experimentally obtained for each algorithm. For these timed runs, each of 20 distinct LOS test cases were run, in both directions as described above, and each was repeated 100 times, for a total of 4000 LOS determinations. The times were tracked by the test shell program. All times are in seconds.

Algorithm	Preproc	Pt Loc	Travers	LOS Tot
F: Grid/edge method	1200.0	n.a.	n.a.	258.0
C: DCEL Traversal	10.9	**4.0**	111.9	**115.9**
P: Triangle Traversal	17.0	50.9	**96.7**	147.6
BBN SIMNET PVD	n.a.	n.a.	n.a.	154.2

Preproc:	Preprocessing; time spent converting the input polygons into the LOS data structure.
Pt Loc:	Point location; portion of the time spent performing the LOS determinations that was used in point location.
Travers:	LOS Traversal; portion of the time spend performing the LOS determinations that was used traversing the LOS from one polygon (or grid) to the next).
LOS Tot:	LOS Total; total time for the LOS determinations. Recall that these times are for the entire LOS test case set, not a single LOS determination.

The point location and LOS traversal times for Algorithm F and the BBN SIMNET PVD algorithm could not be tracked separately due to the structure of those algorithms; LOS Total time only is given for them.

4. Conclusions and future work

As can be seen from the timing results in section 3.4, two of the three IST LOS algorithms were faster than the SIMNET PVD LOS algorithm; Algorithm C was just over 24.8% faster. This result was quite gratifying because essentially no optimization was performed on the IST algorithms. It had been intended to use a run-time profiler to identify heavily-used portions of the code and optimize those portions for maximum performance, but project time was not available for that step. The algorithms were simply designed, implemented, debugged, and immediately timed.

IST's Intelligent Simulated Forces project is currently implementing a Semi-Automated Forces Testbed for the SIMNET battlefield simulation environment [Smith,1992], under DARPA contract N61339-89-C-0044. Ironically, the LOS algorithm used in that Testbed is Algorithm F, the slowest algorithm by far in this experiment. Future versions of the Testbed will likely use an enhanced version of one of the other LOS algorithms. A faster LOS algorithm would free additional computational resources to the Testbed's other processing.

This experimental study is just a beginning; there is clearly much additional work to be done in this area. One idea is obvious from an examination of the timing results table. The table shows that Algorithm C had the fastest point location process, whereas Algorithm P performed the LOS traversal more efficiently. An obvious step to take would be to combine Algorithm C's point location with Algorithm P's LOS traversal to produce an algorithm that could probably complete the test cases in approximately 100 seconds, for a projected 50% speedup over the SIMNET PVD algorithm. The price of such a combination would be increased preprocessing time and storage requirements.

The SIMNET PVD algorithm contains a number of interesting and clever "tricks" which very quickly check simple situations that could block the LOS; if one of those is found, the entire LOS traversal process is rendered unnecessary and is not executed (see section 2.6). Those checks are not present in any of the IST algorithms; integrating them into the IST algorithms would certainly increase their speed. Doing so may or may not reduce the generality of the LOS algorithm, depending on how the special checks are built into the algorithm.

Both the SIMNET PVD algorithm and IST Algorithm F take advantage of the patch/grid structure of the SIMNET terrain database to perform point location in O(1) (i.e. constant) time. Algorithm C and Algorithm P (the two fastest algorithms), because they do not use the SIMNET terrain database, must perform searches of the terrain polygons for point location. They manage to overcome that disadvantage and still run faster than the SIMNET PVD algorithm and Algorithm F by virtue of their highly efficient LOS traversals. One member of the project team has already designed a point location algorithm, inspired by the patch/grid structure, which can be used by those algorithms and may provide constant time point location in the average case.

In terms of future work, the final point to be made is that there remains considerable theoretical research in this area that may be applicable to the pragmatic problem of LOS determination. A number of research publications, especially [Cole,1989], contain ideas that seem very promising but were not applied in this preliminary experiment. A substantial increase in efficiency in LOS algorithms, beyond what was achieved here, appears to be very possible. As this is written, preliminary design of improved algorithms has already begun.

5. References

BBN (1991). *Software Design Document, PVD CSCI (3)*, BBN Systems and Technologies, June 1991, 388 pages.

Brassard, G., and Bratley, P. (1988). *Algorithmics: Theory and Practice*, Prentice-Hall, Englewood Cliffs NJ, 361 pages.

Bresenham, J. E. (1965). "Algorithm for Computer Control of Digital Plotter", *IBM Systems Journal*, Vol. 4, No. 1, pp. 25-30.

Chazelle, B., and Guibas, L. J. (1988). "Visibility and Intersection Problems in Plane Geometry", *Technical Report CS-TR-167-88*, Princeton University, June 1988, 40 pages.

Cole, R., and Sharir, M. (1989). "Visibility Problems for Polyhedral Terrains", *Journal of Symbolic Computation 7*, pp. 11-30.

Companion, M. A. (1989). "Intelligent Simulated Forces: Evaluation and Exploration of Computational and Hardware Strategies" *Quarterly Progress Review #1*, Institute for Simulation and Training, University of Central Florida, 20 July 1989, 17 pages.

El Gindy, H., and Avis, D. (1981). "A Linear Algorithm for Computing the Visibility Polygon from a Point", *Journal of Algorithms 2*, pp. 186-197.

Foley, J. D., and Van Dam, A. (1982). *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, 664 pages.

Ghosh, S. K., and Mount, D. M. (1987). "An Output Sensitive Algorithm for Computing Visibility Graphs", *Proceedings of the 28th IEEE Symposium on Foundations of Computer Science*, Los Angeles 1987, pp. 11-19.

Glassner, A. S. (Editor) (1988). *Graphic Gems*, Academic Press, New York NY, 833 pages.

Gonzalez, G., Mullally, D., Smith, S., Vanzant-Hodge, A., Watkins, J., and Wood, D. (1990). "A Testbed for Automated Entity Generation in Distributed Interactive Simulation", *Technical Report IST-TR-90-15*, Institute for Simulation and Training, University of Central Florida, 15 August 1990, 37 pages.

Guibas, L. J., Hershberger, J., Leven, D., Sharir, M., and Tarjan, R. E. (1987). "Linear Time Algorithms for Visibility and Shortest Path Problems inside Simple Polygons", *Algorithmica 2*, pp. 209-233.

Guibas, L. J., and Hershberger, J. (1989). "Optimal Shortest Path Queries in a Simple Polygon", *Journal of Computer and System Sciences 39*, pp. 126-152.

Hershberger, J. (1989). "An Optimal Visibility Graph Algorithm for Triangulated Simple Polygons", *Algorithmica 4*, pp. 141-155.

Knuth, D. E. (1973). *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley, Reading MA.

Lee, D. T., and Preparata, F. P. (1979). "An Optimal Algorithm for Finding the Kernel of a Polygon", *Journal of the ACM*, Vol. 26, No. 3, July 1979, pp. 415-421.

Muller, D. E., and Preparata, F. P. (1978). "Finding the intersection of two convex polyhedra", *Theoretical Computer Science 7 (2)*, pp. 217-236, October 1978.

Munem, M. A., Tschirhart, W., and Yizze, J. P. (1974). *College Algebra*, Worth Publishers, 518 pages.

Myers, G. J. (1979). *The Art of Software Testing*, John Wiley & Sons, 177 pages.

Nagy, G. and Wagle, S. G. (1979). "Approximation of Polygonal Maps by Cellular Maps", *Communications of the ACM*, Vol. 22, No. 9, September 1979, pp. 518-525.

Preparata, F. P., and Shamos, M. I. (1988). *Computational Geometry: An Introduction, 2nd Edition*, Springer-Verlag, New York NY, 398 pages.

Smith, S. H., Karr, C. K., Petty, M. D., Franceschini, R. W., and Watkins, J. E. (1992). "The IST Semi-Automated Forces Testbed", *Technical Report IST-TR-92-7*, Institute for Simulation and Training, University of Central Florida, 28 February 1992.

Stanzione, T. (1989). "Terrain Reasoning in the SIMNET Semi-Automated Forces System", *Geo'89 Symposium on Geographical Information Systems for Command and Control*, SHAPE Technical Centre, The Hague, Netherlands, October 1989.

Stubbs, D. F., and Webre, N. W. (1985). *Data Structures with Abstract Data Types and Pascal*, Brooks/Cole, Monterey CA, 459 pages.

6. Appendices

6.1 List of acronyms

A number of acronyms are used in this document. While each is defined the first time it is used, all acronyms are defined again in this section in alphabetical order for ease of reference.

BBN	Bolt, Beranek, and Newman Systems and Technologies
DCEL	Doubly Connected Edge List
DSR	Division of Sponsored Research
I/O	Input/Output
IST	Institute for Simulation and Training
LOS	Line of Sight
Mhz	Megahertz
PSLG	Planar Straight Line Graph
PVD	Planview Display
SIMNET	Simulator Network
UCF	University of Central Florida
2D	Two dimensions, or two dimensional
3D	Three dimensions, or three dimensional

6.2 Authors' biographies

Mikel D. Petty is Principal Investigator of the Intelligent Simulated Forces project at the Institute for Simulation and Training. He has earned a M.S. in Computer Science from the University of Central Florida and a B.S. in Computer Science from the California State University, Sacramento. He is currently a Ph.D. student in Computer Science at UCF, with research interests in simulation, artificial intelligence, and computational geometry. He has over ten years experience participating in and leading large software development projects.

Charles E. Campbell is a Graduate Research Assistant for the Intelligent Simulated Forces project at the Institute for Simulation and Training. He has earned a M.S. in Computer Science from the University of Central Florida and a B.S. in Computer Science from Indiana University. His career interests include computer graphics, virtual reality, and object-oriented programming.

Robert W. Franceschini is an Undergraduate Research Assistant for the Intelligent Simulated Forces project at the Institute for Simulation and Training. He will earn a B.S. degree in Computer Science from the University of Central Florida in May, 1992. He is currently an undergraduate Mathematics student at UCF, and will begin work on an M.S. in Computer Science at UCF in Fall, 1992. His research interests include artificial intelligence, object-oriented programming, and graph theory.

Micheline H. Provost is a Graduate Research Assistant at the Institute for Simulation and Training. She has earned a M.S. in Computer Science from the University of Central Florida; the title of her thesis is "ExploreNet: A Networked Simulation Environment for Cooperative Problem Solving." In addition, she has earned a B.S. in Computer Science and a B.S. in Mathematics, also from the University of Central Florida. Her career interests include computer simulation and object oriented methodologies.

Clark R. Karr is a Software Engineer in IST's Intelligent Simulated Forces project. He served as the Semi-Automated Forces Dismounted Infantry (SAFDI) Team Leader. He has earned a M.S. in Computer Science from the University of Central Florida and a B.S. in Biology from the University of Denver. He is currently a Ph.D. student in Computer Science at UCF. His research interests are in the field of artificial intelligence, specifically intelligent behavior in simulated environments and natural language understanding.

6.3 Credits

Algorithm F, the Grid/edge method, was designed by Robert W. Franceschini and Mikel D. Petty, and implemented by Franceschini. His implementation included a a program to build a SIMNET format terrain database from a set of 3D polygons. Algorithm C, the DCEL Traversal method, was designed and implemented by Charles E. Campbell. Algorithm P, the Triangle Traversal method, was designed by Micheline H. Provost and Mikel D. Petty, and implemented by Provost. Clark R. Karr adapted the BBN PVD LOS algorithm to this project's test environment, assisted with the algorithm timing tests, and prepared many of the figures for this technical report. Richard Dunn-Roberts performed the IST review of this technical report, and is responsible for a number of valuable improvements to it. Mikel D. Petty conceived, proposed, and led the Line of Sight project.

6.4 Test data files and algorithm source code

The test polygon data file, the LOS test case file, and the source code for the implemented algorithms are all available upon request from IST. Their length precludes their inclusion in this document.

0000037