AN EFFICIENT LOOP DETECTION AND REMOVAL ALGORITHM FOR 3D SURFACE-BASED LITHOGRAPHY SIMULATION

by

John J. Helmsen

Memorandum No. UCB/ERL M92/125

10 November 1992

AN EFFICIENT LOOP DETECTION AND REMOVAL ALGORITHM FOR 3D SURFACE-BASED LITHOGRAPHY SIMULATION

by

John J. Helmsen

Memorandum No. UCB/ERL M92/125

10 November 1992

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

AN EFFICIENT LOOP DETECTION AND REMOVAL ALGORITHM FOR 3D SURFACE-BASED LITHOGRAPHY SIMULATION

by

John J. Helmsen

Memorandum No. UCB/ERL M92/125

10 November 1992

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Chapter 1

Introduction

1.1 Surface Based Lithography Simulation using SAMPLE-3D

Simulation of photolithography processes, with tools like SAMPLE-3D [1][2][3], is helpful in the development and design of these processes. Once theoretical models have been developed and validated, process simulation tools allow designers and production engineers to predict outcomes of proposed and actual processes reducing the number of these experiments. Three-dimensional simulators are beginning to emerge due to the availability of computational power and investments in implementing algorithms suitable for 3D. Making 3D simulations fast, memory efficient, physically accurate and robust requires applying and even extending concepts from computational geometry. The questions of interest in this study are how, and to what extent, a computational geometry approach could improve the SAMPLE-3D lithography and pattern transfer simulator.

The SAMPLE-3D simulator is based on the use of surface advancement algorithms [4][5]. The surface is represented as a mesh of nodes, segments and triangular faces, organized in a winged edge data structure. In simulation, the continuous photolithography process is partitioned into discrete time steps. SAMPLE-3D maintains a representation of the surface at each time step. The location and geometry of the mesh surface at the next time step is determined using the present mesh and the lithography model. At each step, the nodes of the mesh are advanced along trajectories which are derived from this information. The path of each node may therefore be likened to a ray of light passing through an inhomogeneous medium. Because of this physical interpretation, this method is described as a *ray-trace* [1][3][8][9][10] algorithm. To reduce resource consumption and maintain accuracy, the length of the mesh segments are kept within a range determined by the segment length parameter. If during simulation, a segment shrinks or grows beyond specified limits, the regularization process deletes or adds segments, and the neighboring triangles are updated.

When SAMPLE-3D is used to simulate structures with standing wave effects, or processes that cause structures that change their topological status during simulation, some mesh nodes may cross through the surface and form regions of negative area. The surface of this space is called a loop. (Figure 1) To perform simulation accurately and efficiently, these loops must be removed. Failure to do so results in unnecessary consumption of computer resources and possible inaccuracies. Resource consumption by loops results from the extra memory that they occupy and the extra time that is added for computation of the next time step. The simulation may also be made inaccurate if the process is dependent on geometry, such as shading or reflection effects. The loop, since it represents a nonexistent piece of surface, may falsely shade a valid surface piece. Since the loops represent nonphysical structures, visualization of results is also difficult. Loop identification and removal are important in practice and some methods have been reported [1][2][3]. Using a minimal amount of CPU and memory resources is also necessary. For example, when the SAMPLE-3D code was performance tested, with a simple loop removal method included, the method was found to consume half of the CPU cycles and be the major cause of failure on complex examples. A more acceptable method would consume a small number of CPU resources compared to surface advancement and would fail only in the most degenerate cases.

1.2 First Pass Loop Removal in SAMPLE-3D

In the original investigation of lithography simulation using SAMPLE-3D, a straightforward extension of 2D loop removal was proposed. An intersection test was performed on all the triangles of the mesh, so that pairs of intersecting triangles could be identified. The line segment that is common to both triangles in the pair represents the boundary between the valid surface and a loop. Each triangle was split along its segments to form smaller triangles. The orientation vectors of the triangles were used to determine which triangles were valid surface and which were loops. Because a surface of a three dimensional object has a consistent orientation, and because the motion of the surface under the process models was either consistently in the direction or consistently in the opposite of the surface normal for the whole simulation, the loops could be identified as parts of the surface that were inconsistent with these invariants. (Figure 2) The loop parts were

then removed by deleting the inconsistent triangles and reestablishing the connectivity such that the surface had consistent orientation and was not self-intersecting. This method was easy to implement and worked for application studies, however it was limited in speed and robustness. Examination of CPU resources and analysis of simulator crashes identified three issues to be addressed. First, to identify the pairs of intersecting triangles in the mesh, each pair of triangles was examined to determine intersection. This required $O(N^2)$ time (where N is the number of triangles.) This was found to dominate the CPU time and is a fundamental consequence of any approach which does not include auxiliary mechanisms to spatially localize intersection testing. Second, a robust method to handle degenerate triangle intersection cases is needed. The intersection of two triangles was assumed to be a line segment. Parallel triangles and triangles that had a single point of intersection gave ill-determined results. Third, certain loop structures were not considered in the original algorithm. Triangle that have a single point in common were assumed to always be non-intersecting. As we will show later, this is not the case. These difficulties have led to this work.

1.3 Work in This Report

The three issues of the original implementation are addressed in this thesis. A method of loop removal based on a hierarchial spatial subdivision structure known as an octtree has been proposed and implemented. This method finds the intersecting triangle pairs in $O(N\log N)$ time by utilizing an auxiliary octtree structure which allows a quick mapping of any region of space to the set of triangles intersecting it. By mapping each triangle as a region, the octtree is used to find those triangles which are nearby to a given triangle. In this manner only nearby triangles need to be checked for intersections. A description of the method and a discussion of its theoretical aspects is given in Chapter 2. In Chapter 3, the triangle intersection and loop identification routines of the algorithm are discussed. The $O(N\log N)$ property for the algorithm is shown to be preservable throughout the entire algorithm, at least for 'typical' surfaces as encountered in actual IC processing. A more robust version of subdividing intersecting triangles is presented. An improved method of determining the loop sections of the mesh is also discussed. The method is shown to be

useful not only for removing negative volume loops, but for handling all topological deformations that may occur during simulation such as air bridges, which are of interest in fabricating sensors. Time and memory consumption results of the implementation of this algorithm in SAMPLE-3D are given in Chapter 4.

1.4 Overview of the Algorithm

The algorithm operates in three basic parts. First, a method of localizing triangles spatially is used to reduce computational overhead in finding triangle intersections. The particular method used is called an octtree, which is based on hierarchically subdividing the simulation space. Other methods such as multi-dimensional segment trees may be used, but the octtree has further uses in visibility and reflection computations, which make it more useful than the alternatives.

Once the triangle intersection lines have been found, they are used to partition the mesh so that they form the boundaries between new adjusted triangles. This is done by placing the new nodes first, then the segments and finally the triangles. This minimizes round off error and maximizes robustness.

Once the mesh is subdivided into parts separated by intersection lines, the loops are identified and removed, and the mesh is reconnected so that it is continuous and loop-free.

Chapter 2

Octtree Algorithm

2.1 Why an Octtree?

Many algorithms in computational geometry operate in O(Nlog$N$) time, where N is the number of objects that the algorithm must take as input. For the loop removal problem, N is the number of triangles that describe the surface. The O(Nlog$N$) limit exists because there is a necessary O(N) step of reading the input implicit in the algorithm, and because an intermediate data structure requiring O(log$N$) time for insertion and deletion of a single object is often mandatory.

To allow loops to be removed in O(Nlog$N$) time, the necessary first step of finding all loops must be performed in O(Nlog$N$)time. Since the mesh is represented by triangles connecting all points on the surface, detecting if there is a self intersecting point of the mesh is equivalent to checking if a pair of intersecting triangles exists. The entire region of self intersection is then the union of the intersection regions of all triangle pairs. In the previous delooping implementation, an intersection test was performed on all pairs of triangles in the mesh. This takes O(N^2) time and is highly inefficient. Since most pairs concern two rather distant triangles, the method wasted much effort. If each triangle is checked only against other nearby ones, then much time can be saved. Therefore, to improve the efficiency of the routine, a method must be implemented whereby triangles can be implicitly ordered in space, so that the algorithm only examines likely intersection candidates.

2.2 What is an Octtree?

To perform a spatial sort upon the triangles, we use an octtree[6]. The octtree may be considered as a three-dimensional equivalent of a binary tree. The root node of the octtree represents the entire simulation space. This node has eight children, or subnodes. The subnodes of the root node represent subspaces. Typically the octants are formed by slicing the complete domain by three mutually intersecting perpendicular planes. Although their dimensions are halved, these subspaces are otherwise geometrically similar to the full simulation space. The subspaces are

arranged in a 2x2x2 formation. Each subnode in turn has 8 octants which represent further divisions of the simulation region. This division continues recursively to a preset depth, or until further subdivision is deemed unnecessary. Subnodes that have no children are called leaves (Figures3 & 4).

2.3 Sorting Triangles with an Octtree.

Determination of pairs of intersecting triangles is made efficient by having all triangles located in this octtree. Starting with an initially empty octtree, each triangle is inserted into it in turn. Before the triangle is inserted, it is converted to a polygonal representation. This representation is an ordered list of points which define a polygon on a plane in three-dimensional space. The plane is the plane of the triangle. This represents the intersection of the set of points contained in the triangle with the region of space associated with a node of the octtree. The triangle is first inserted into the root node, and, using the polygon, is checked to see which subnodes it intersects. Subnode intersection is performed by dividing the polygon into eight smaller polygons through plane divisions. This generates eight residue polygons (some of which may be non-existent, i.e. there are no points of the triangle, which intersect the region of space associated with the subnode.) which represent the intersection of the triangle with the eight regions of space associated with the eight subnodes. The division into eight polygons is performed in three steps. The original polygon is divided into two parts by one of the three bisecting coordinate planes. These two polygons are divided by the second plane so that four polygons are formed, and finally the third plane is employed to form eight polygons. If a polygon formed for a subnode is non-existent, then the triangle does not intersect it. If a subnode has a defined residue polygon, the intersection test can be applied recursively using the new polygon associated with the subnode as the input to recursion. Recursion halts at a predefined depth.

Although the insertion test for a triangle can be applied recursively at any subnode, it is not always desirable to do so. Often, to save memory storage, recursion down to the preset maximum depth is not performed. In the octtree, if a subnode contains only one triangle, recursive subdivision is not performed until another triangle is inserted into the subnode. The space that would be

consumed by recursion is saved. When the limit of recursion is reached, a linked list is formed of all the triangles that intersect that leaf. A new list of triangles is formed, which is the union of all the lists of triangles in the leaves which contain the inserted triangle. This list of triangles is the list of candidates for intersection with the inserted triangle. An explicit intersection test is performed between the inserted triangle and the candidates to find the actual intersections.

The time that each triangle requires to be inserted into the octtree is a function of the number of intersection tests with subnodes and of the number of triangle-triangle intersection tests that must be performed. The number of triangle-triangle intersection tests is the number of leaves that the triangle intersects times the average occupancy of the leaves. As the height of the tree increases, the number of leaves that the triangle may intersect also increases, since each leaf represents a smaller amount of the simulation region. If the height is increased too far, then the triangle will occupy very many leaves. This is undesirable, since the increased accuracy in localizing the triangle, and thereby ruling out comparison candidates, does not make up for the extra time and memory consumed in forming such a small subdivision. If the tree is too shallow, then the larger leaves may contain too many comparison candidates. The extreme case, where the height of the tree is one, is equivalent to the old method. Therefore, to utilize the octtree to its fullest potential, a method of determining the optimal level of recursion must be found. A sensible trade-off is to make the size of the leaves similar to the size of the triangles, which are constrained in their dimension by the feature size parameter. Under this condition, the number of leaves that a triangle may intersect has an average constant value, and the number of leaves in the tree in total is $O(N)$ where N is the number of triangles. The height of this octtree is $O(\log N)$, assuming that typically encountered meshes have a relatively even distribution of leaves. Therefore, the total time for triangle insertion is $O(N \log N)$. Since the time of an algorithm is also related to the size of its output, the number of triangle intersection pairs is also a contributing factor, so the real time required is $O(N \log N + N_i)$ where N_i is the number of intersection pairs although the latter is never expected to dominate, since most triangles do not take part in any intersection. Likewise, since the number of leaves in the tree is $O(N)$, the memory that must be consumed to form the octtree is expected to

be O(N). Experimental results from our implementation of such an octtree structure confirm these expectations (see Chapter 4).

2.4 Getting Good Intersections.

Sometimes a triangle-triangle pair will have a coincidence between the vertex of one triangle and the plane of another triangle. A coincidence is defined as when the vertex is within a distance of 10^{-8} of the size of a triangle. Since a triangle in a pair may have a node that is coincident with the plane of the other triangle, seven new intersection types arise that are distinct from the basic Face & Face = Line Segment type. (Figure 5) To simplify the code and to reduce the number of cases that must be analyzed and handled, we remove undesirable coincidences by small randomized movements of the offending vertices. This causes the intersections that may occur to reduce to the desired generic case. The only type of node-plane coincidence that is maintained is when the node is an actual node of both triangles and when the node is not the entire intersection region of the pair. In an iterative procedure, each such vertex is moved normal to the plane of the intersecting triangle, a distance of plus or minus 10^{-8} to 10^{-7} of the size of a triangle. The additional time that is required by this preprocessing step is O($N_n \log N$), where N_n is the number of triangles that contain nodes to be moved. This may be performed in this time by moving the node, removing all triangles which neighbor the node from the octtree, replacing the triangles in their new positions and recomputing the intersections. Presently when any nodes are moved, the entire set of intersections is recomputed. Although it is only necessary to remove and replace the triangles adjacent to a pushed node, a complete recomputation is performed to simplify programming complexity. Since coincidences are expected to occur infrequently once the symmetry in the initial conditions is broken, this term is omitted from the complexity analysis. Experiment has confirmed this assumption (see Chapter 4). After the first time step, on examples that represent real structures, the octtree needs to be recomputed at most twice.

Chapter 3

Deloop

3.1 Identifying the Intersection Line.

Each intersecting triangle pair now has associated with it a well-defined intersection line segment. Taken together, these segments form a set of piecewise linear curves in the domain space of the surface mesh. Figure 6 shows the line segments for two intersecting surfaces. These lines define the boundaries between sections of the mesh that represent the actual surface and sections that are loops. Because of the continuity of the surface, i.e. each segment of a triangle is contained by another triangle or is on the simulation region boundary, the intersection lines either form closed curves in the domain space, or have both ends terminate at the boundary. In normal space, where each intersection line is the superposition of either two segments of the same or of different intersection lines in the domain space, this means that the intersection lines may have three types of terminations. First, the termination may occur at a Node & Node = Node type intersection. In this case, two triangles which share a common node, intersect one another at more points than the common node. Although the intersection line passes through the node and does not terminate in the domain space, it often does terminate in normal space (it is possible that the line may continue onwards, but it is not guaranteed.) Loops which contain this type of intersection are called "banana" loops (Figure 9) due to their typical shape. Second, the intersection line may terminate at the simulation boundary. In the domain space, this can be represented as two separate lines that intersect the simulation boundary. Because the intersection line separates the plane into two disjoint parts, it may be considered a closed curve. Lastly, the curve may be closed in three dimensional space, and have no endpoints. This is represented as two separate closed curves in the domain space. Although each of these three types reduces to closed curves in the domain space, it is important to identify them, since intersection tracking operations are performed on them in three-dimensional space.

3.2 Tracking the Intersection Line.

Because the intersection lines form the boundaries between parts of the mesh which are actual surface and the parts of the mesh which are loops, it is necessary to subdivide intersecting triangles along the intersection line. This is performed by picking an appropriate starting point and following the intersection line along the mesh. The connectivity information of the winged edge data structure is particularly important for this part of the algorithm. The intersection pairs which were generated by the octtree have no inherent order. Therefore, in order to know which intersection pair is the next on the line, the edges by which the intersection line leaves the old pair must be examined. Given a present pair, when the intersection line crosses an edge, the triangle whose edge has been crossed is replaced in the pair. This new pair of triangles is then looked up from the old list.

Given the entire list of triangle intersection pairs, an initial pair is picked. Associated with this pair of triangles is an intersection line segment, which may be connected to zero, one or two other intersection line segments at its endpoints. Having zero, one, or two segments attached means having two, one or zero intersection line termination conditions attached to that segment, respectively. If there are no connected segments, then the intersection line is considered to be completely tracked. If there is one, then the line is followed to the next triangle pair using the connectivity of the mesh as a guide. This continues until a termination condition is reached. If there are two attached segments, then an arbitrary direction is chosen and the line is followed until either a termination condition is reached or it returns to the original pair of triangles. If a termination condition is reached, the line is then tracked from the starting point in the other direction until the second termination condition has been reached. The pairs of triangles are removed from the list of pairs after they are processed. This can be done in $O(N_i \log N_i)$ time where N_i is the number of pairs. When the intersection line under consideration is completely tracked, a pair is taken out of the list. This is repeated until all pairs have been processed. Such a linearly ordered processing of all the intersection line segments reduces code complexity and enhances robustness. It insures that if some triangles are intersected more than once, either by the same or different inter-

section lines, the algorithm need only concern itself with a single intersection line at any time entering and leaving triangles.

3.3 Splitting the Triangles.

As the intersection line is tracked, new mesh nodes and edges are being created. A new mesh edge is formed for each intersection line segment, and a new node is created at the end of each segment. Each new node that is created represents the intersection of the face of a triangle with a segment that borders two other triangles. Therefore, two new edges are created, between the new node and the nodes of the old edge. In this manner, a framework of edges and nodes is created which represents the intersection line. Once this has been performed, each triangle has associated with it a sequence of intersection lines that traverse it. If the original border of the triangle is traversed in an ordered, non-repeating manner, which can be done in $O(N)$ time, a set of polygons is derived which partition the surface of the triangle (Figure 10). Each of these polygons is then triangulated (Figure 11 local & Figure 7 global). This, theoretically, takes $O(N_l \log N_l)$ time [7], where N_l is the number of intersection line segments that were contained in that triangle. The old triangle is then removed from the mesh, and the new triangles are inserted.

Since each intersection pair contributes one line segment to each triangle, the total time for this algorithm is $O(N_i \log N_i)$. Three, four or n-way mutual triangle intersections may be handled in this manner by performing a planar sweep across the plane of the triangle and checking for intersections of intersection line segments. New nodes would be formed at the points where the segments cross, and the segments would be divided into smaller segments with the new nodes as endpoints. In some cases, such as when the N triangles in the mesh are mutually intersecting, the time complexity would be $O(N^2)$, however, for most geometries encountered in practical lithography tasks, these higher order intersections are so rare that they can be neglected in the complexity analysis.

Once all of the polygons that were formed by the intersection line have been triangulated, the intersection line now lies along the edges of the new triangles (Figures 7 & 11). Each intersec-

tion line segment now has four triangles connected to it, two triangles in each of two planes.

Another method was previously tried for triangle subdivision. This method subdivided each triangle into smaller triangles simultaneously with intersection line traversal. As the intersection line was observed to leave a triangle, the triangle would be divided into smaller triangles which had the intersection line along their edges. The triangles were reinserted into the octtree and rechecked for intersections with other triangles. The motivation for this method was that divisions of triangles could be simplified to a single line crossing from one side of a triangle to another. This would remove the difficulties that are inherent when many intersection lines cross a single triangle. After some simple examples, an attempt was made to extend this approach for more general cases. The extra code required was found to be excessive in length and hard to maintain. In many cases, the need to recompute intersections with extremely small triangles also led to significant numerical inaccuracy. The approach in the above paragraphs did away with many of these complications. The original reason for handling complicated examples was also found to be inappropriate. The intersections encountered in typical meshes are not complex enough to require such a treatment.

3.4 Loop Identification and Removal

Now that the mesh has been tessellated so that the intersection line only appears on mesh segments, it is still necessary to determine which parts of the mesh are surface and which are loops. Considering the four triangles meeting at each intersection line segment, we see that no two surface triangles or two loop triangles can both exist in the same local plane on opposite sides of the intersection line. We also see that each of the intersection lines forms a closed loop in the domain space (possibly closed outside the simulation boundary). Therefore the mesh is divided into two distinct groups of triangles by the intersection lines. The Jordan Curve Theorem [7] can be used to identify the triangles forming the loops. This theorem states that, given a closed curve in Euclidean space, an escape ray from a point in the region bounded by the curve will cross the curve an odd number of times. An escape ray from a point external to the curve will cross it an

even number of times. If the curve is considered to be the surface mesh, and the Euclidean space is the simulation space, then any path on the surface can be considered to be part of an escape ray. A path connecting any two triangles through the domain space that crosses the intersection line an even number of times is therefore equivalent to saying that the two triangles are of the same type. A connecting path between two triangles that crosses an odd number of times means that the triangles are of a different type. Any path between two triangles will give the same results since the intersection curves are closed. Different paths will have the same parity of their number of crossings, although they may have a different number of absolute crossings. The mesh triangles can therefore be labeled in $O(N)$ time. Each triangle is selected and labeled either "surface" or "loop". It is then put into a list of triangles that possibly have adjacent triangles that are not yet labeled. This triangle is then removed from the list, its adjacent triangles are labeled and these are placed into the list if they are not yet labeled. This process repeats itself until all triangles are labeled. The parts that are labeled as loop triangles are then removed from the mesh. After all triangles of type "loop" have been removed from the mesh, the data structure needs to be restored to its canonical form. Since at each intersection line segment, there are two "loop" triangles and two "surface" triangles that contain it as an edge, mesh integrity is restored by replacing the intersection line segment with a normal edge that connects the two "surface" triangles. A cross sectional view of this example on the two basic types of loops is shown (Figure 12).

In the preceding paragraph, the algorithm assumed that we have reliable knowledge of a triangle that was a surface triangle. For lithography tasks that are primarily two-dimensional in nature, such as IC wafer processing, we can typically start in a corner of the simulation area, sufficiently far away from areas where loops might be generated. For more complicated three-dimensional tasks, such as pieces of silicon that are floating free in a wet etch process, the type of a starting triangle can be determined by counting the number of surface intersections on an escape ray leaving the triangle in the direction of its outward surface normal (Figure 13). If the escape ray crosses an equal number of surface pieces in the direction of their outward surface normals as in the direction of their inward surface normals, then the ray has emanated from a valid surface trian-

gle. This triangle may now be used in the loop labeling algorithm as a valid starting point.

3.5 Identification of Detached Parts.

Parts of the surface may become separated during the loop removal; they represent pieces of the bulk that have become detached during processing. These pieces can be detected with the same marking algorithm that marks triangles as "loop" or as "surface". If we start from a triangle that clearly belongs to the bulk of our device, all other "bulk" triangles will be connected to it through a path along other "surface" triangles. "Surface" triangles that cannot be reached in this manner are on pieces that have become detached from the surface. To identify the discrete pieces that have become detached, a list of all the "non-bulk surface" triangles is made. One of the triangles in the list is chosen, and all triangles that can be connected to it along "surface" triangles are located. These triangles represent one piece. If the "non-bulk surface" triangle list is non-empty, repeat the process to locate other discrete pieces until the list is empty.

Nested loops, i.e. areas where two loops intersect, have the ability to create a 'false' surface (Figure 13). We can detect these structures by a variant of the loop marking scheme which is dependent on orientation. This method is a three-dimensional variant on the two dimensional winding number. It has not yet been implemented, however. Instead of using a two-state value to distinguish between "loop" and "surface" regions, an integer is used. At the true surface, the integer flag is set to 1. The mesh is marked in the same manner as the original scheme with one exception. As the intersection segments are crossed, the plane that is being passed through is examined. If the marking path is proceeding in a direction opposite to that of the surface normal, then the counter is decremented by 1. If the marking path is proceeding in the direction of the surface normal, then the counter is incremented by 1. If no intersection line was crossed, no change is made to the counter. After labeling, all 1's are actual surface. Both the "bulk" and detached pieces that represent actual objects which have become removed from the bulk during processing will be labeled with 1's. Pieces which have been labeled with other numbers are "non-surface".

3.6 Cleaning the Mesh

In the introduction, it was stated that the mesh had both upper and lower limits on the size of its segments. This condition must now be restored. Since it is considered desirable to leave the intersection line as sharply defined as possible, the first segments to be restored to the proper value are the intersection line segments. This is done by shrinking to zero the length of the small ones and by dividing the large ones into smaller parts. While this is occurring, it is possible to shrink the intersection line too far and initiate a topological change in the surface by closing up a hole or pinching off a created neck of surface. Therefore, the routine must be modified, so that any cleaning up procedure that would change the topological state of the mesh is not carried out. (This has not yet been implemented.) Next, to maintain the intersection line, any other edges that have endpoints on the line, but are not themselves on the line are modified, so that the proper range of segment lengths is restored. When this is done, the points on the intersection line are considered to be fixed in place. Finally, the rest of the mesh is cleaned in the normal manner. (Figure 8)

3.7 Preparing for the Next Time Step in Isotropic Etching

Most etching and deposition methods only require geometrical information to determine the advancement vectors for the next time step. Isotropic etching, via the ray trace method, requires that the directional component of the velocity vector be maintained from one time step to another. This requirement must be satisfied because computing the new direction of advancement with only geometrical information gives inferior results to the ray-trace method. Whenever a new node is created, the direction vector, which would otherwise be computed from information from the previous time step, is interpolated from the neighboring nodes. Therefore, when a new node for an intersection line segment must be created, the direction vector has to be interpolated from the interaction of the two planes.

One method which was attempted, was to apply an interpolated variant of the facet motion algorithm[11]. Each node on the intersection line arises from the intersection of an edge between two triangles, in one plane, and the face of a third in the other plane (Figure 14). The two triangles

ABC and BCD are connected to the segment BC, and the third triangle EFG which is the face that the segment intersects, are considered to be moving facets. Their component of motion in the direction of their normals represents the motion of the triangles at the intersection point. Only the normal component is required because the instantaneous direction of the new node is the important component of the ray trace algorithm. Barycentric interpolation of the normal components of motion is used to find the contribution of each triangle. Once this operation is performed, the vectors V_{ABC}, V_{BCD} and V_{EFG} which represent the motion of each triangle, are transformed to the slowness space. In the slowness space, the direction of the vectors is maintained, but the magnitude is the inverse of each vector's magnitude in the simulation space. The slowness vector which determines the motion of the node is then found (according to the facet motion algorithm,) as the normal vector from the origin onto the plane defined by the endpoints of the three facet slowness vectors. The direction of the new slowness vector is the desired direction and the etch rate is the inverse of the magnitude. (Figure 15)

This method has been shown to be inappropriate for isotropic etching (although it might be suitable for other etching methods.) In isotropic resist etching, the intersection line segments and nodes often lie on portions of the surface which do not have a continuous surface derivative (Figure 16). For this reason, the vector, while possibly accurate for the first time step, represents the motion of the point of intersection between two etch wavefronts, and cannot be assumed to be consistent with the normal manner of surface evolution. A second difficulty which arises is the difficulty in interpolating new nodes properly for split segments. When a segment has grown long enough that it needs to be divided, the new node that is to be created must have a direction vector associated with it. As can be seen in Figure 17, a simple interpolation is not satisfactory in giving a good guess of the proper direction. To get a proper interpolation, there must be information retained about the location of the surface before delooping was performed. This difficulty was also encountered in the two-dimensional version of SAMPLE (Figure 18). The following method is its extension into three dimensions.

Instead of creating only one intersection segment at the intersection of two triangles, two are

created. Two nodes now exist for every one original intersection segment node. The direction vectors associated with these nodes are the ones derived for the originally intersecting planes.above. (The two vectors from the two triangles on the segment are combined into one.) The surface is then advanced at the next time step as per the ray-trace algorithm. Two effects are noted. First, the intersection between the two etch wavefronts does not need to be controlled explicitly, thereby maintaining accuracy. Second, a piece of the mesh is maintained which can be used for accurate interpolation. The drawback is that the loop removal program must be modified for removing discontinuous surfaces. This is not difficult, however, if the winding number paradigm is implemented. Instead of marking the whole surface during the loop identification phase, as each intersection line is crossed, the part on the other plane that will receive a "1" is determined. Further labeling proceeds from that point. In this manner, the discontinuities will not be encountered, and the surface will be identified properly.

Chapter 4

Results

4.1 Octtree Results.

The algorithm was compared to an older deloop algorithm previously implemented for SAMPLE-3D [1]. Both methods were run on the test case shown in Figure 19, which contains 1500 triangles. The result after loop removal is shown in Figure 20. The execution time was 8.8 seconds on a DECstation 5000/125. The octtree had to be recomputed only one extra time to remove possible roundoff difficulties (see Chapter 2, Section 4). The older deloop method required 100 seconds for this test. The new algorithm also removed the loop successfully while the old one failed.

Times required to find all triangle-triangle intersection pairs are shown graphically (Figure 21 & 22) for test on three structures of 680, 3201 and 7421 triangles respectively. A comparison was made between finding the intersections using the octtree method and the brute force method used in the old algorithm (Figure 21). The octtree method has a slope of approximately 1 on the log-log plot, demonstrating an apparent linear dependence of time on the number of triangles. The old method has a slope of almost exactly 2, demonstrating the expected quadratic dependence on the number of triangles.

The variation in time to find the intersections, based on the maximum depth of octtree recursion has also been found (Figure 22). The memory shows an O(N) dependence at all points as expected (Figure 22). Memory consumption for a constant number of triangles with a tree of varying height also behaved as expected. For low feature size to subcell size ratios, the major component in memory storage is the list of triangles on the surface. There is little overhead present due to the octtree hierarchy. As the depth of the tree increases, the number of triangles stays constant, but because of the need to maintain more hierarchial information in the presence of greater recursion, memory needs increase. This causes the memory consumption curve to assume the shape in Figure 22. The minimum of the time curves is approximately where the edge length of the small-

est octtree cells is the same as the predefined size parameter of the mesh segments. The time increases for smaller heights due to inadequate subdivision. The original $O(N^2)$ dependence exists within each leaf, so if the number of triangles in each leaf is large, this is less efficient than further subdivision. Time consumption increases for excessive subdivision due to the need to create extra hierarchial information, but because the number of triangles in each leaf was already small, this information is useless. Decisions on how to trade memory consumption for speed can be made based on these curves. If memory is important, then the octtree may be set to a lower height. This will cause the simulation to take slightly longer, but less memory will be consumed. The intersection runs were performed on a DECstation 3000.

Because both the old and new methods show predictable time and memory usage as a function of the number of triangles, the results can be extrapolated to predict behavior for larger meshes.

4.2 Deloop Results.

The example given in Figures 23-28 is the surface model of exposed resist that has been etched for 5 seconds of simulation time. This example has 6100 triangles. 640 intersection line segments are generated during execution. Deloop was performed after surface advancement was completed. Figure 23 is the original surface shown from a top view. Figure 24 is the delooped surface. The delooped surface consists of two parts. The first part, shown in Figure 25 is a top view of the bulk of the resist. Figure 26 is the same picture shown from the side. The second part in Figure 27 shows the four pieces that have detached themselves from the surface. Figure 28 shows the intersection lines which represent the self-intersection of the mesh.

The resist has been exposed using a contact cut mask. The bottom of the resist is reflective, which causes the electrical field to form standing waves. Higher electrical field intensities result in higher etch rates in those regions, thereby causing layers of resist where the etch rate is alternatively high and low.

Although this example has 640 intersection line segments, the triangle intersection location

part of the algorithm is still the most time intensive part. This suggests that the dominating term in loop removal is not in the removal of the loops, but in their location and identification.

4.3 Conclusions

The introduction of an auxiliary octtree structure to quickly find the set of triangles which are "near" to a given coordinate has reduced the deloop time in SAMPLE-3D by an order of magnitude. Since the deloop time is now O(Nlog$N$) versus O(N^2), this improvement will be even larger for cases involving in excess of 6,000 triangles. A case of intersecting triangles with parts detaching from the surface has been identified as causing the previous version to crash. In the new version, this case has been accounted for in the theory and handled appropriately in examples. This has been achieved by a more rigorous geometrical approach to the deloop problem. The new method can also readily deal with topological changes in the surface, as verified by the removal of the outermost intersection line in Figure 28. This intersection line removal is topologically equivalent to the formation of an air bridge or tunnel. This has also been verified in the same example where parts of the surface were detached from the bulk.

This work has shown that improvements in the time and memory performance of algorithms employed in process simulation are valuable. The increase in performance can be significant. When designing useful and robust simulators, a solid knowledge of computer science and software engineering is just as important as experience in the area to which simulation is being applied.

References

[1] K.K.H. Toh, *Ph.D. Dissertation*, University of California, Berkeley, Dec. 1990.

[2] E.W. Scheckler, K.K.H. Toh, D.M. Hoffstetter and A.R. Neureuther, "3D Lithography, Etching and Deposition Simulation," *Symposium on VLSI Technology*, pp. 97-98, (Oiso, Japan), May 28-30, 1991.

[3] E.W. Scheckler, *Ph.D. Dissertation*, University of California, Berkeley, Nov. 1991.

[4] A. Moniwa, T. Matsuzawa, N. Hasegawa and H. Sunami, "Three-Dimensional Photoresist Imaging Process Simulator for Strong Standing Wave Effect Environment", *IEEE Transactions on CAD*, pp. 431-437, 1987.

[5] K.K.H. Toh and A.R. Neureuther, "Three-Dimensional Simulation of Optical Lithography", *Proceedings SPIE, Optical/Laser Microlithography IV*, vol. 1463, pp. 356-367,

March 1991.

[6] D. Meagher, "Geometric Modeling Octtree Using Encoding", *Computer Graphics and Image Processing*, p. 192, June 1982.

[7] F.P. Preparata and M.I. Shamos, *Computational Geometry*, Springer Verlag, New York, 1985.

[8] P.I. Hagouel, *Ph.D. Dissertation*, University of California, Berkeley, 1976.

[9] L. Jia, W. Jian-kun, and W. Shao-jun, "Three-Dimensional Development of Electron Beam Exposed Resist Patterns Simulated by Using Ray Tracing Model," *Microelectronic Engineering*, vol. 6, pp. 147-151, 1987.

[10] E. Barouch, B. Bradie, H. Fowler, and S.V. Babu, "Three-Dimensional Modeling of Optical Lithography for Positive Photoresists," *Interface' 89: Proceedings of KTI Microelectronics Seminar*, pp. 123-136, Nov. 1989.

[11] B. Foote, *M.S. Thesis*, University of California, Berkeley, Sept. 1990

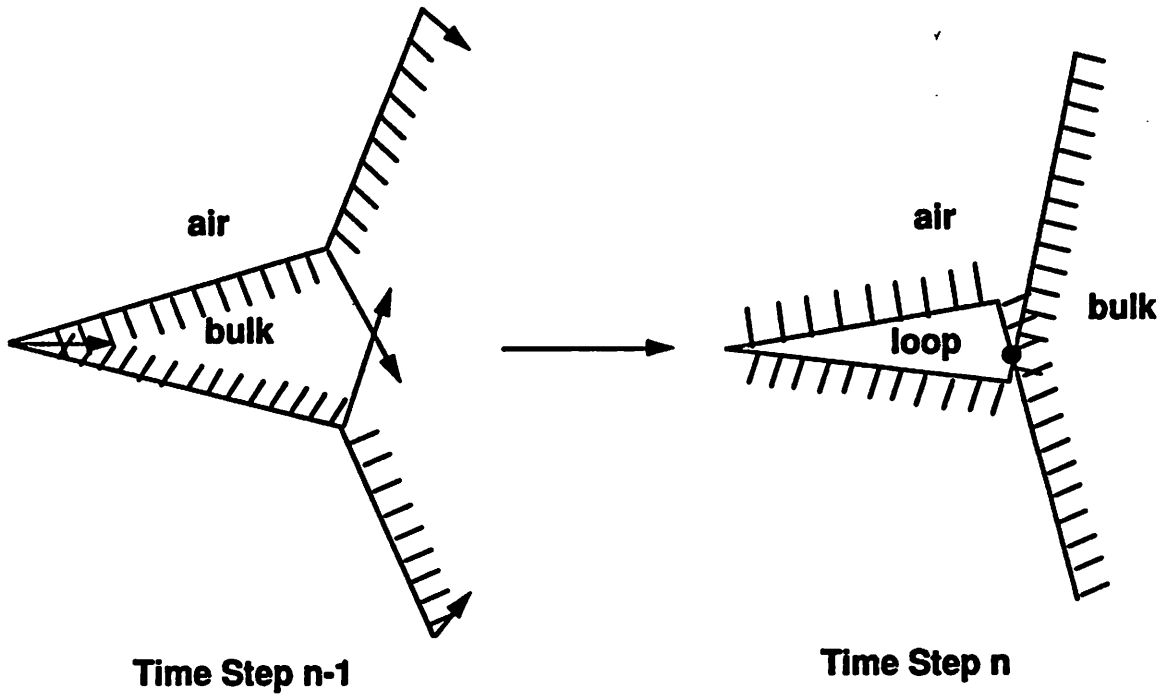


Figure 1

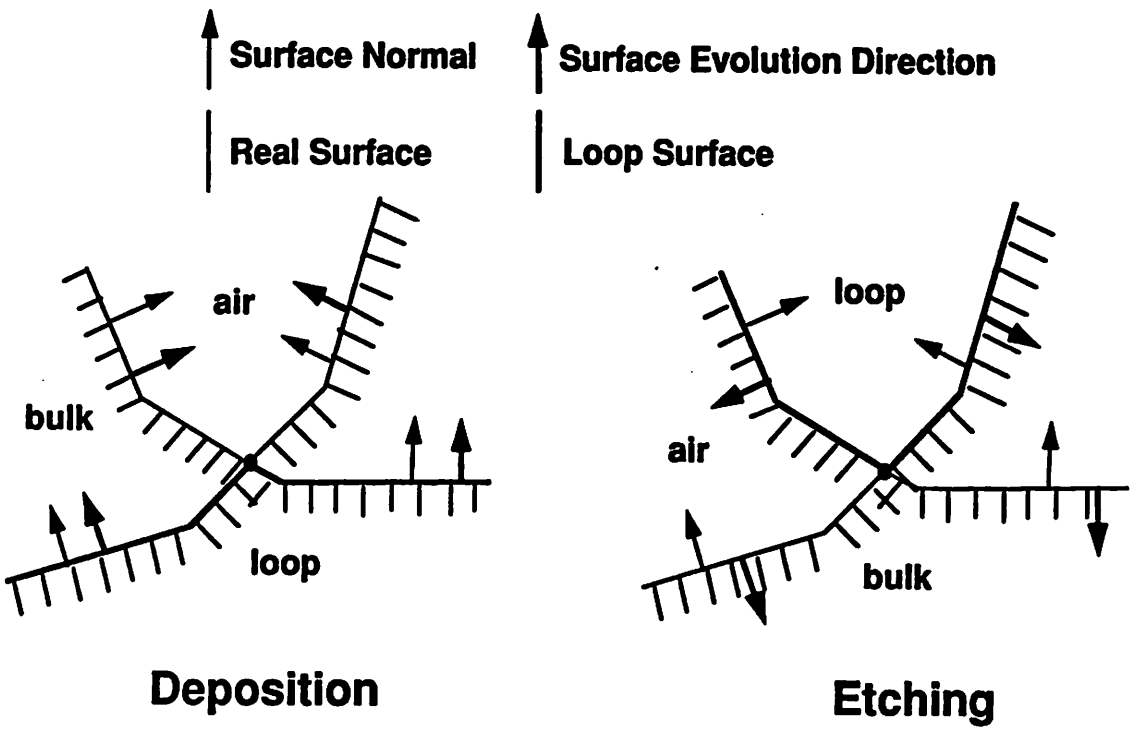
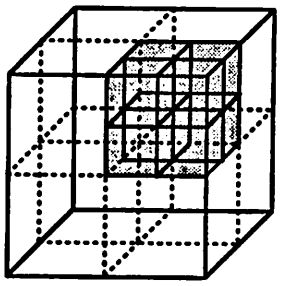
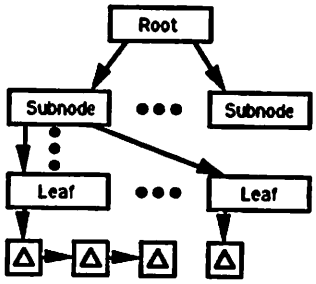


Figure 2

Octtree Triangle Sorting



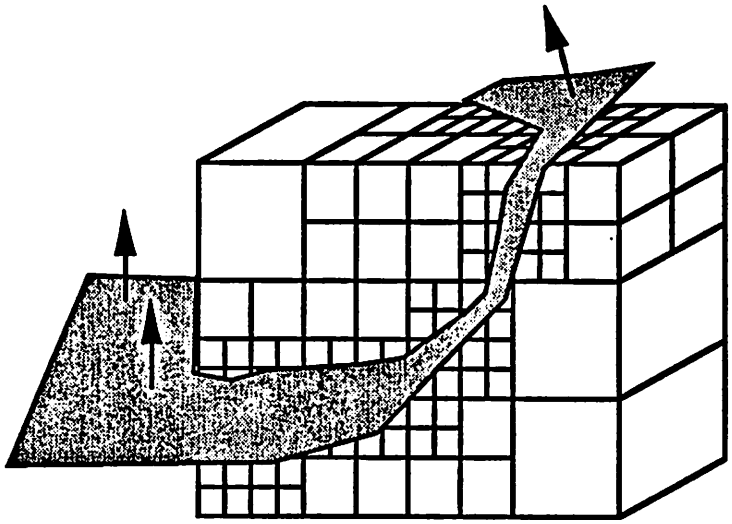
Octtree Spatial Division



Triangle Location Storage

Figure 3

Octtree Spatial Division



The simulation region is tessellated more finely near the surface than away from it.

Figure 4

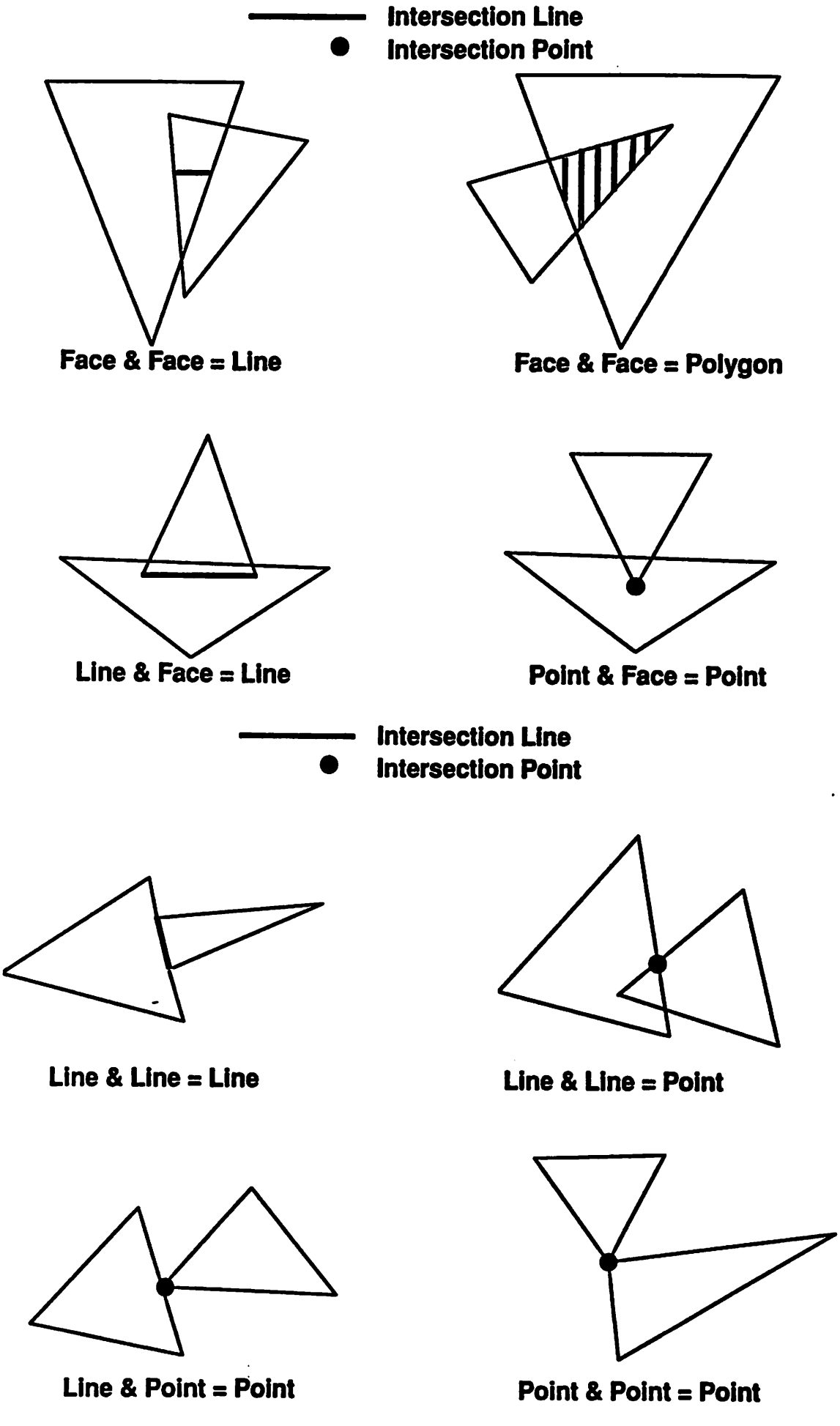

Intersection Line
Intersection Point
Face & Face = Line
Face & Face = Polygon
Line & Face = Line
Point & Face = Point
Intersection Line
Intersection Point
Line & Line = Line
Line & Line = Point
Line & Point = Point
Point & Point = Point

Figure 5

Triangular Mesh with Intersection Line

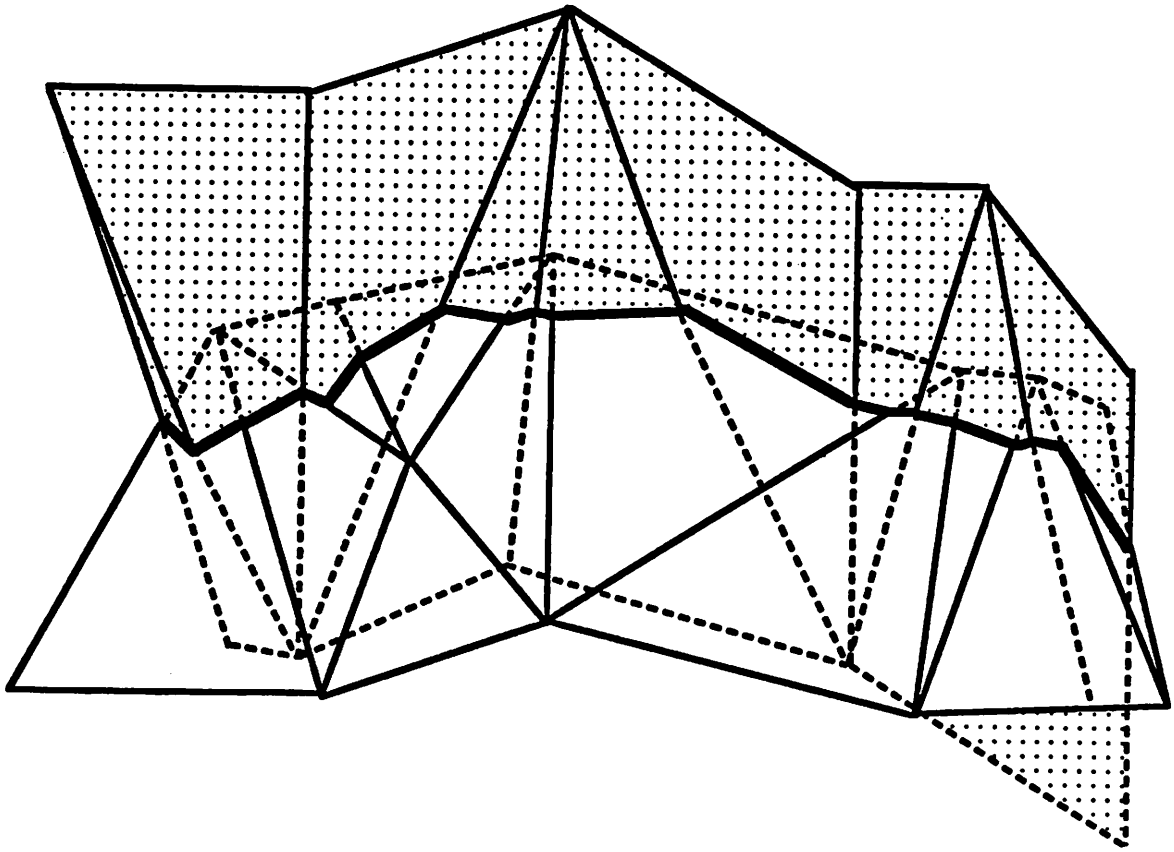


Figure 6

Triangular Mesh after Triangle Division

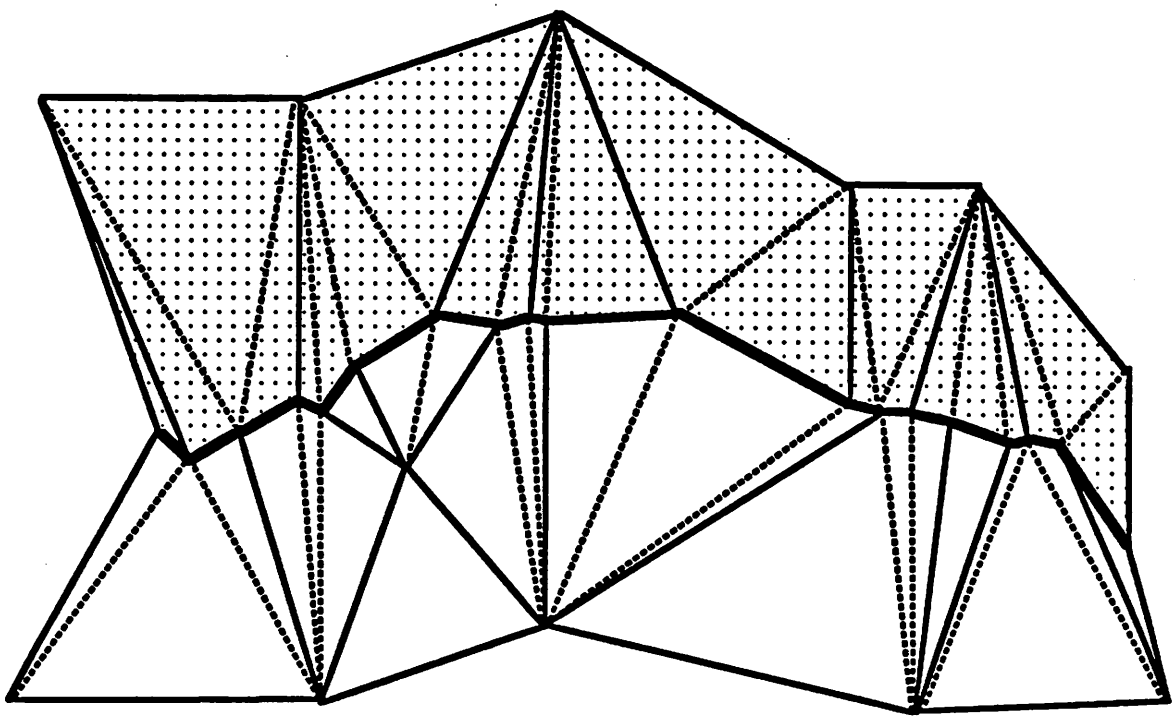


Figure 7

Triangular Mesh After Small Segments Removed

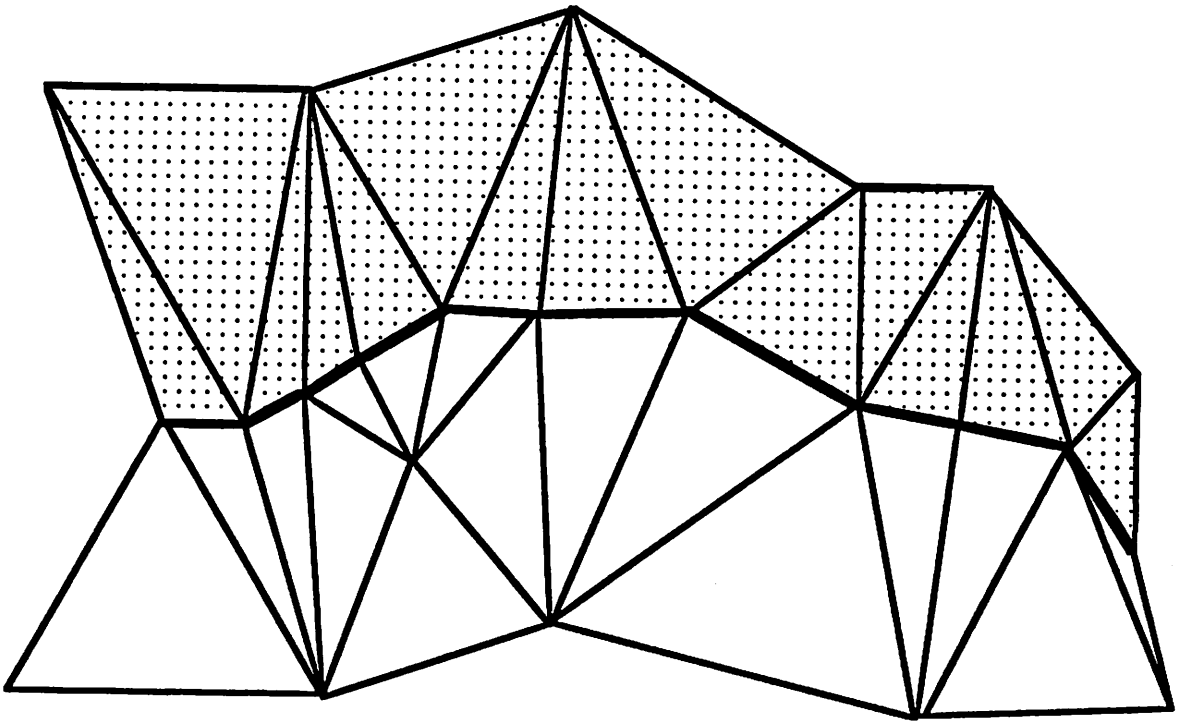
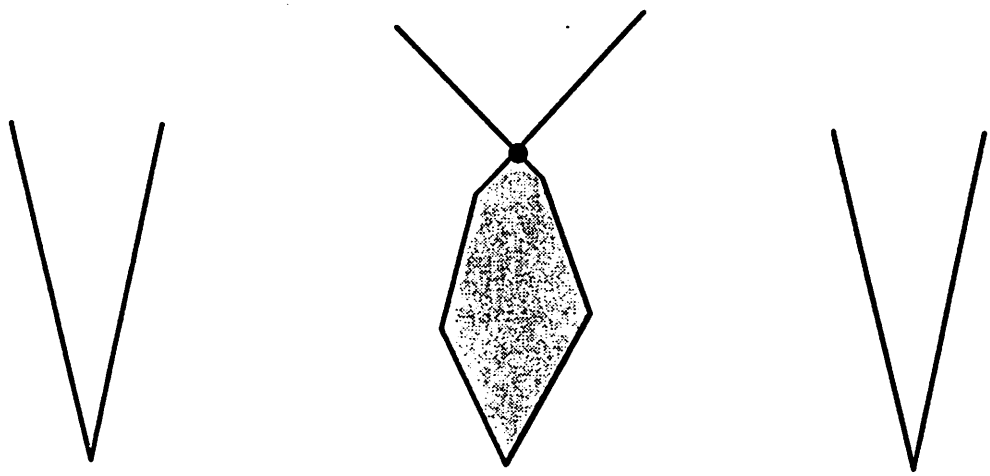
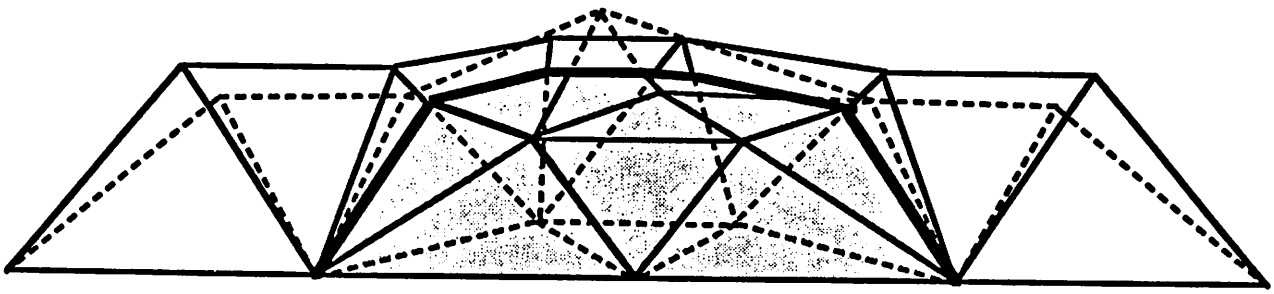


Figure 8



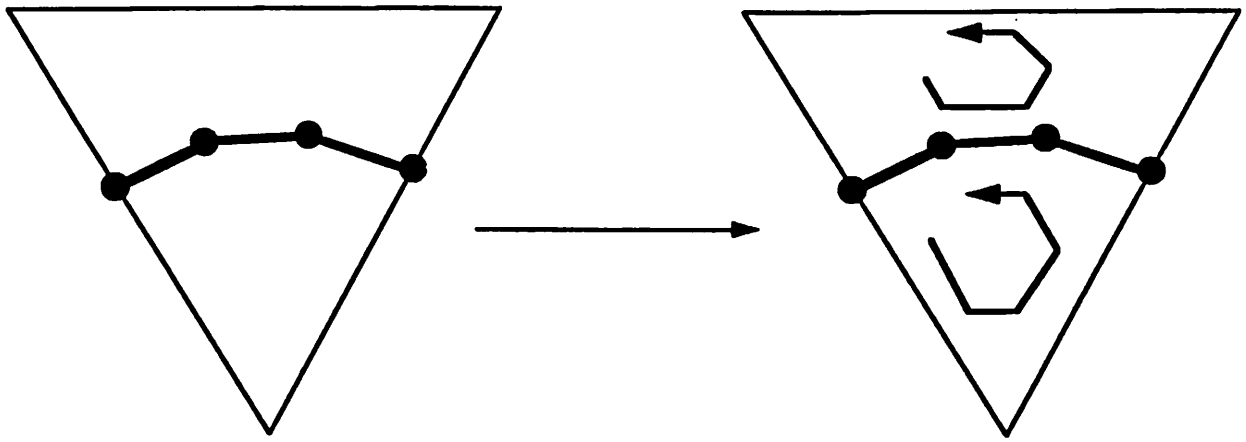
Banana Loop Cross Sections



Banana Loop Front View

Figure 9

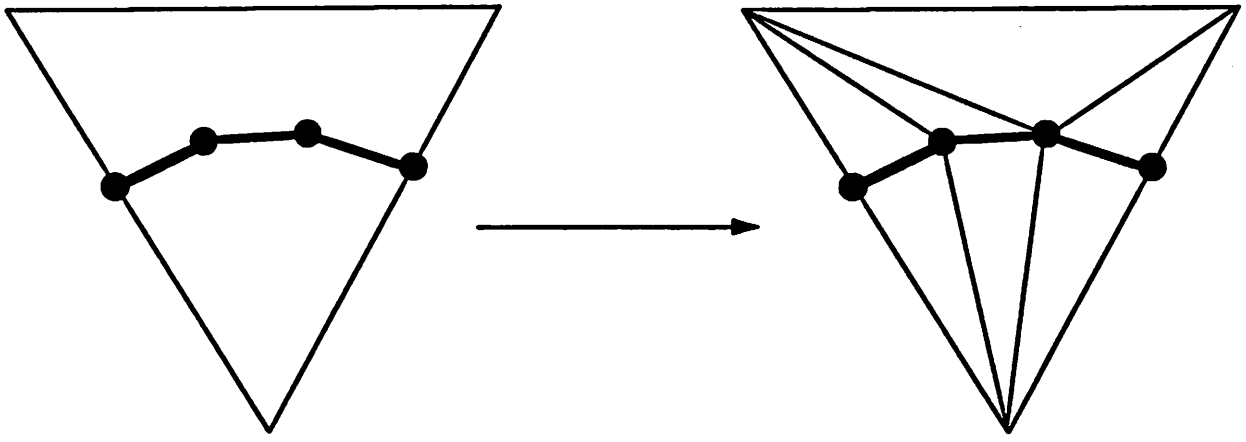
Splitting Triangles



Once the intersection line is determined, each triangle is decomposed into polygons.

Figure 10

Splitting Triangles



Once the polygons are determined, each is triangulated.

Figure 11

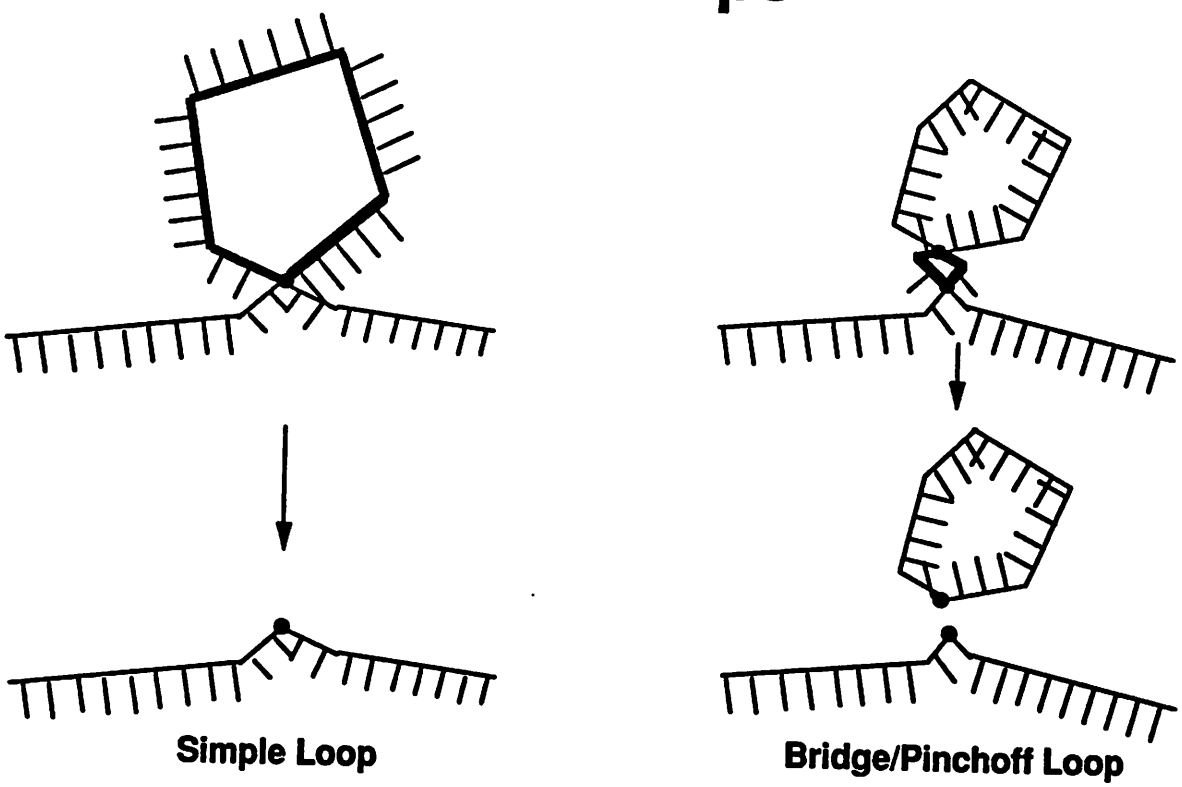


Figure 12

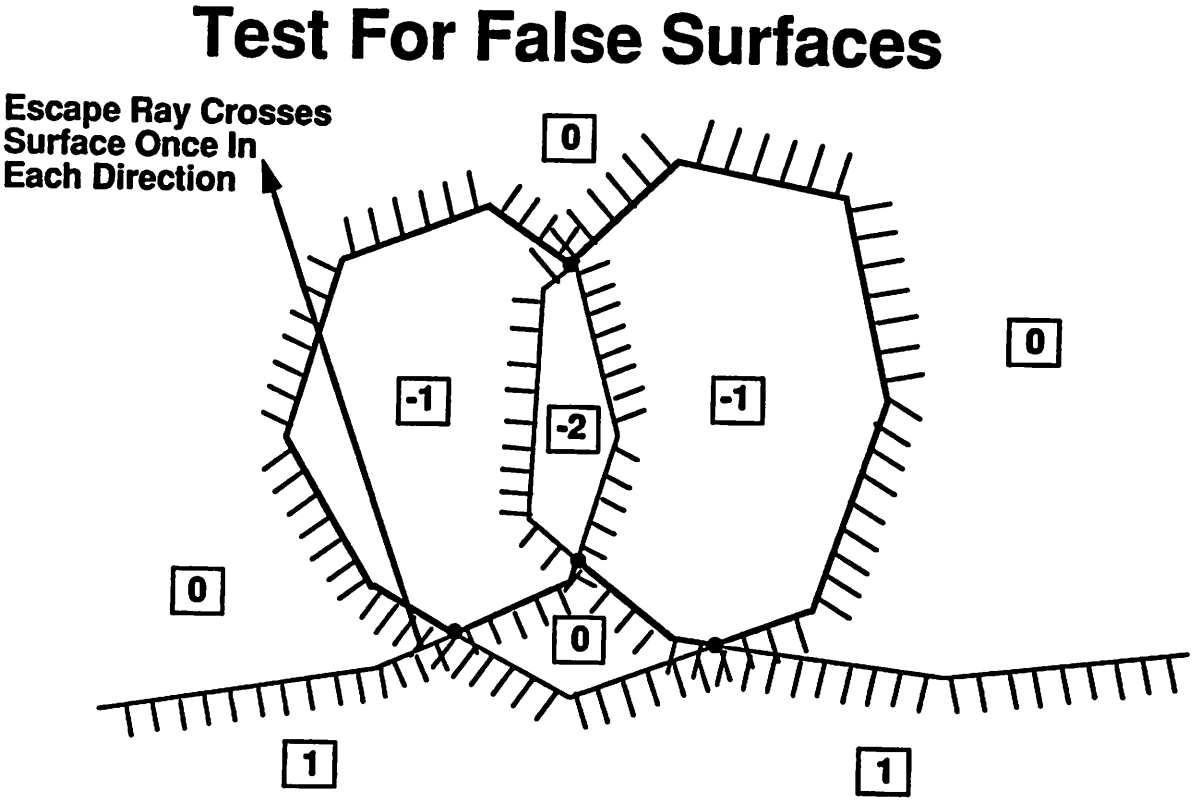


Figure 13

Node Direction Interpolation

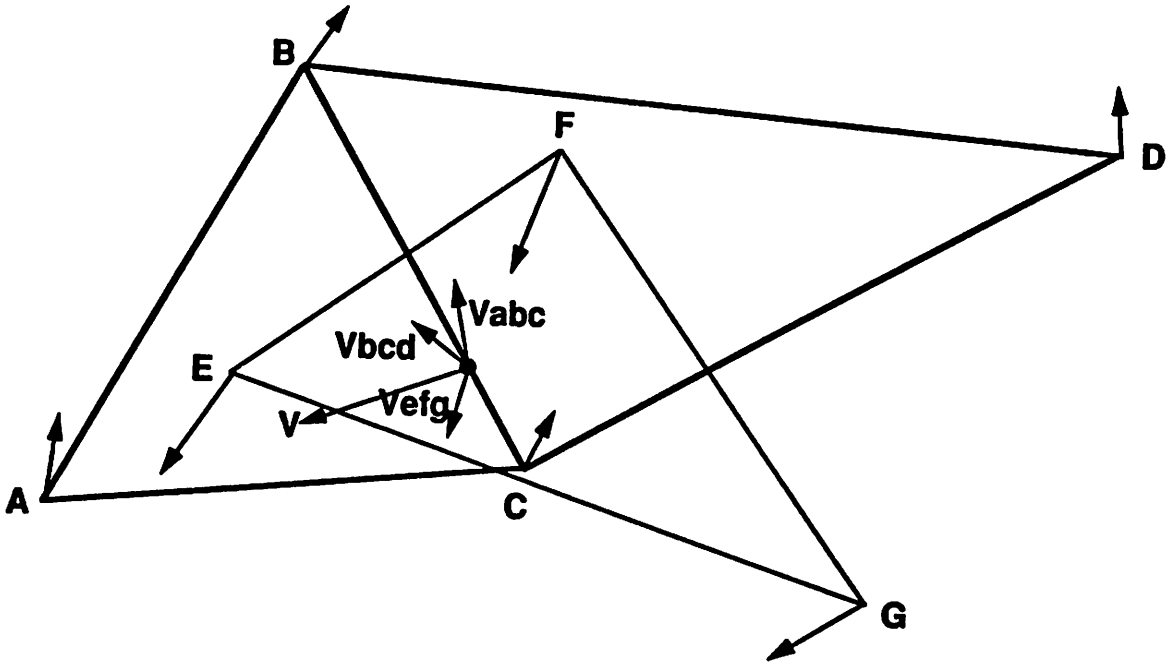


Figure 14

Slowness Diagram

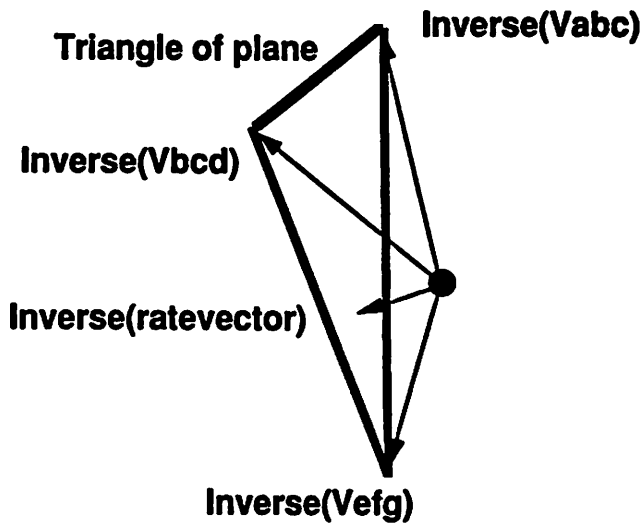
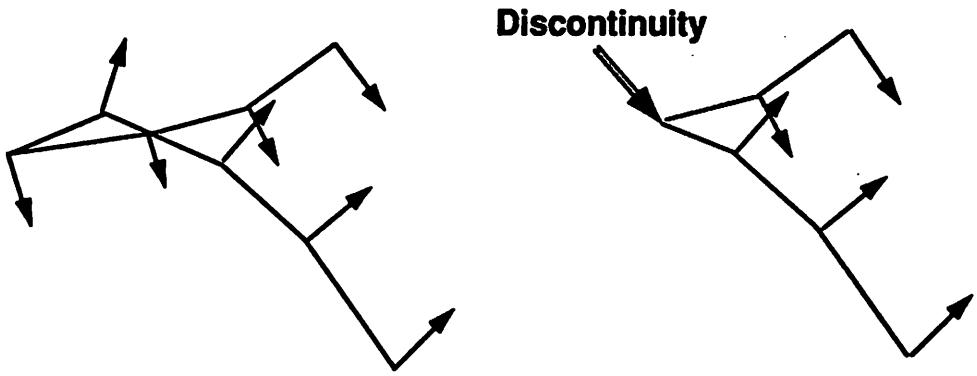


Figure 15

Discontinuous Surface Derivatives



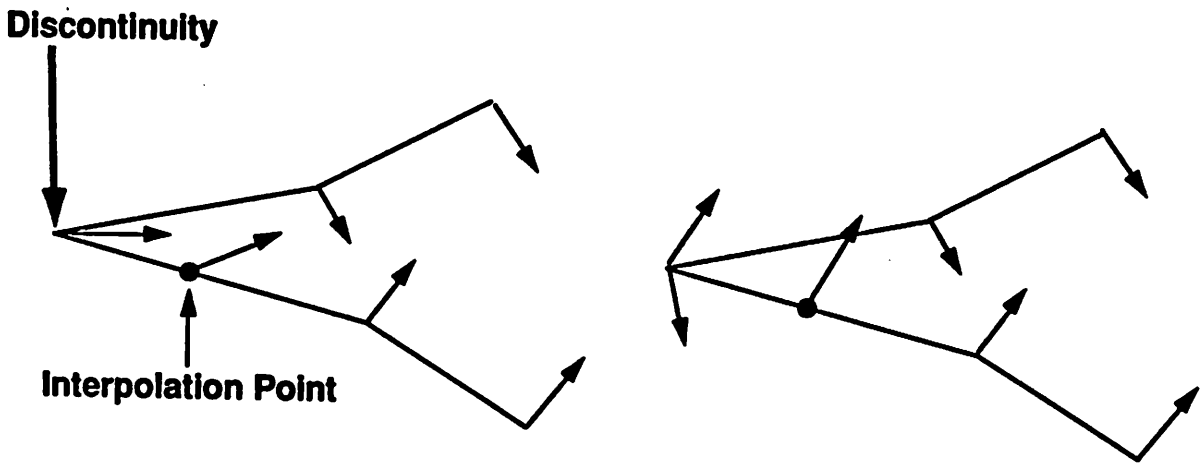
Before Formation

After Formation

An originally continuous surface forms a discontinuity when two wavefronts collide.

Figure 16

Interpolation Difficulty



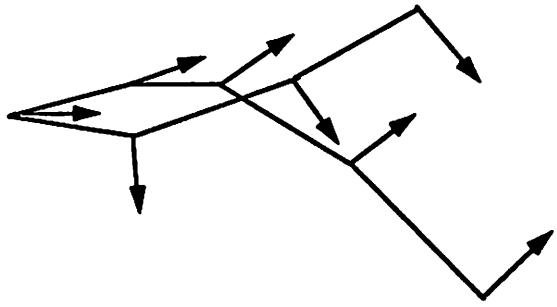
Interpolation using calculated vector for discontinuity

Interpolation using double node method. (The two nodes have same spatial coordinates but different vectors)

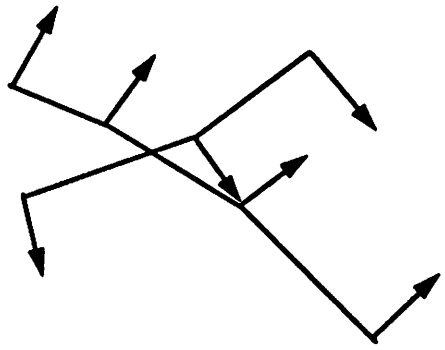
Figure 17

Discontinuity Time Evolution

(Advanced From the Previous Figure)



Representing the discontinuity as a ray causes inaccurate simulation



Advancing wavefronts with the discontinuity as two border points gives far more accurate answers

Figure 18

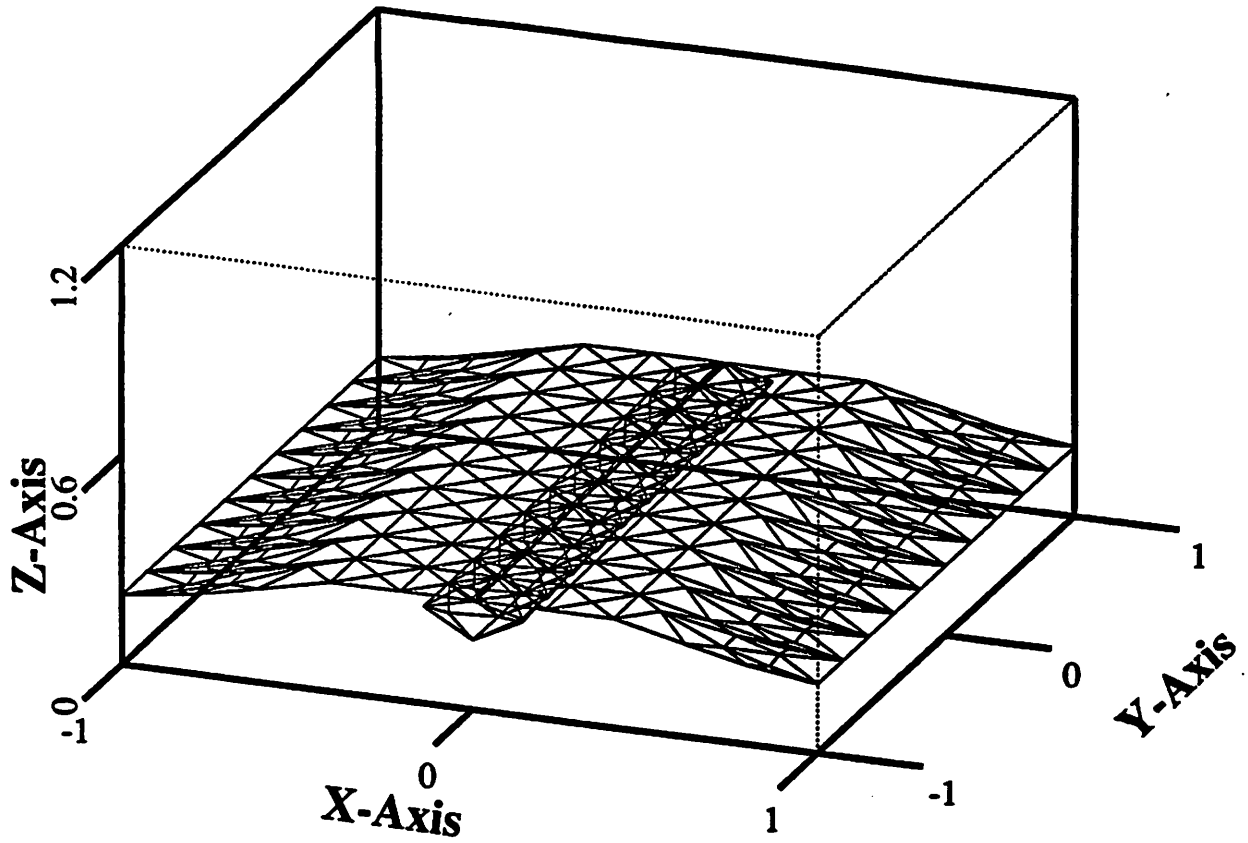


Figure 19

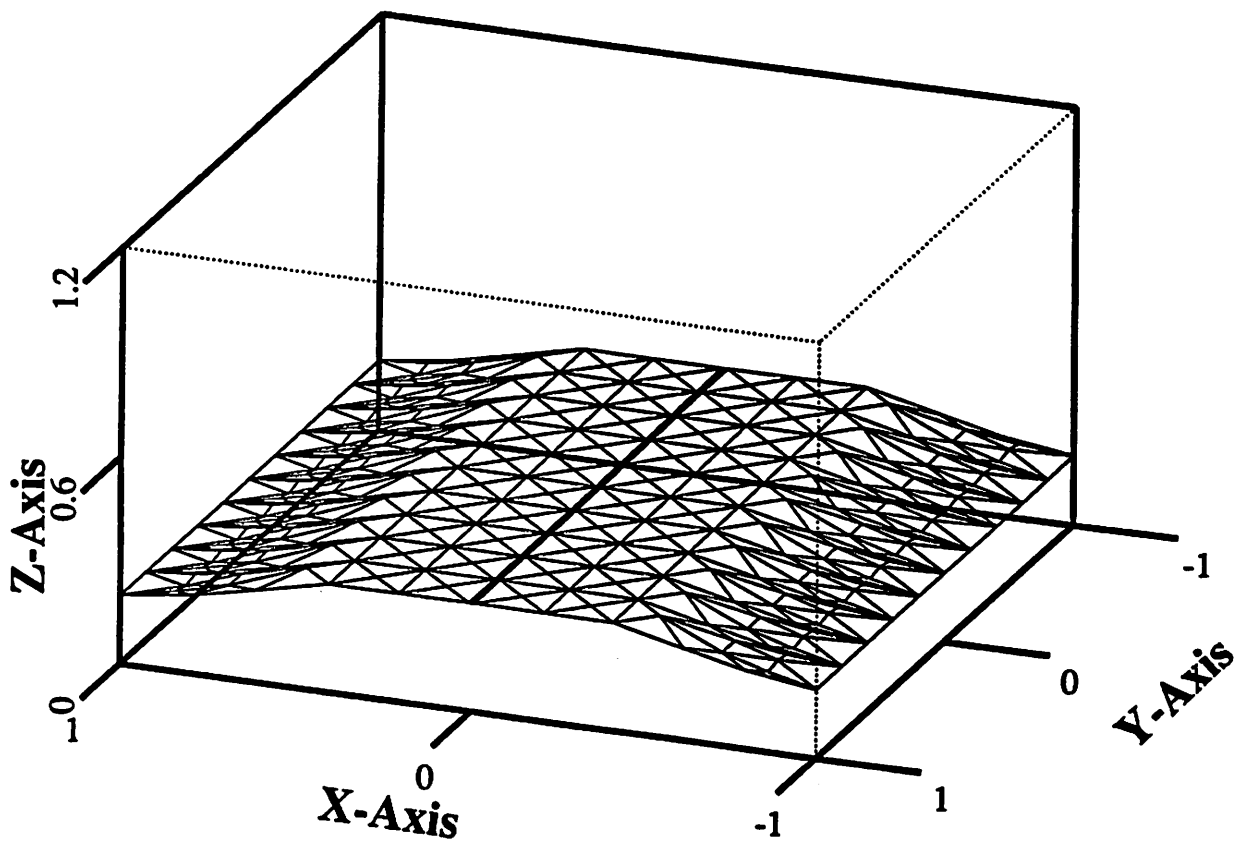


Figure 20

Octtree Time Results

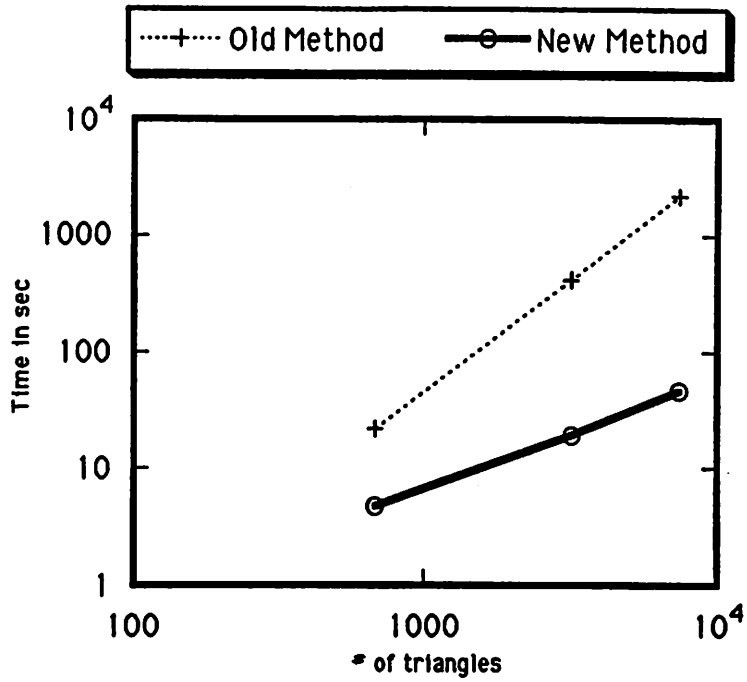


Figure 21

Octtree Time & Memory

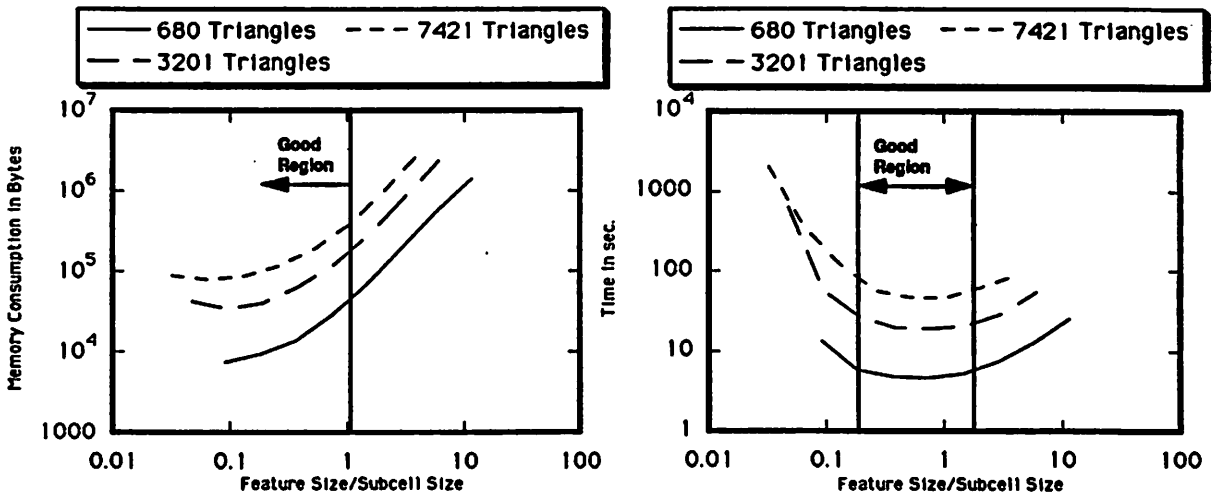


Figure 22

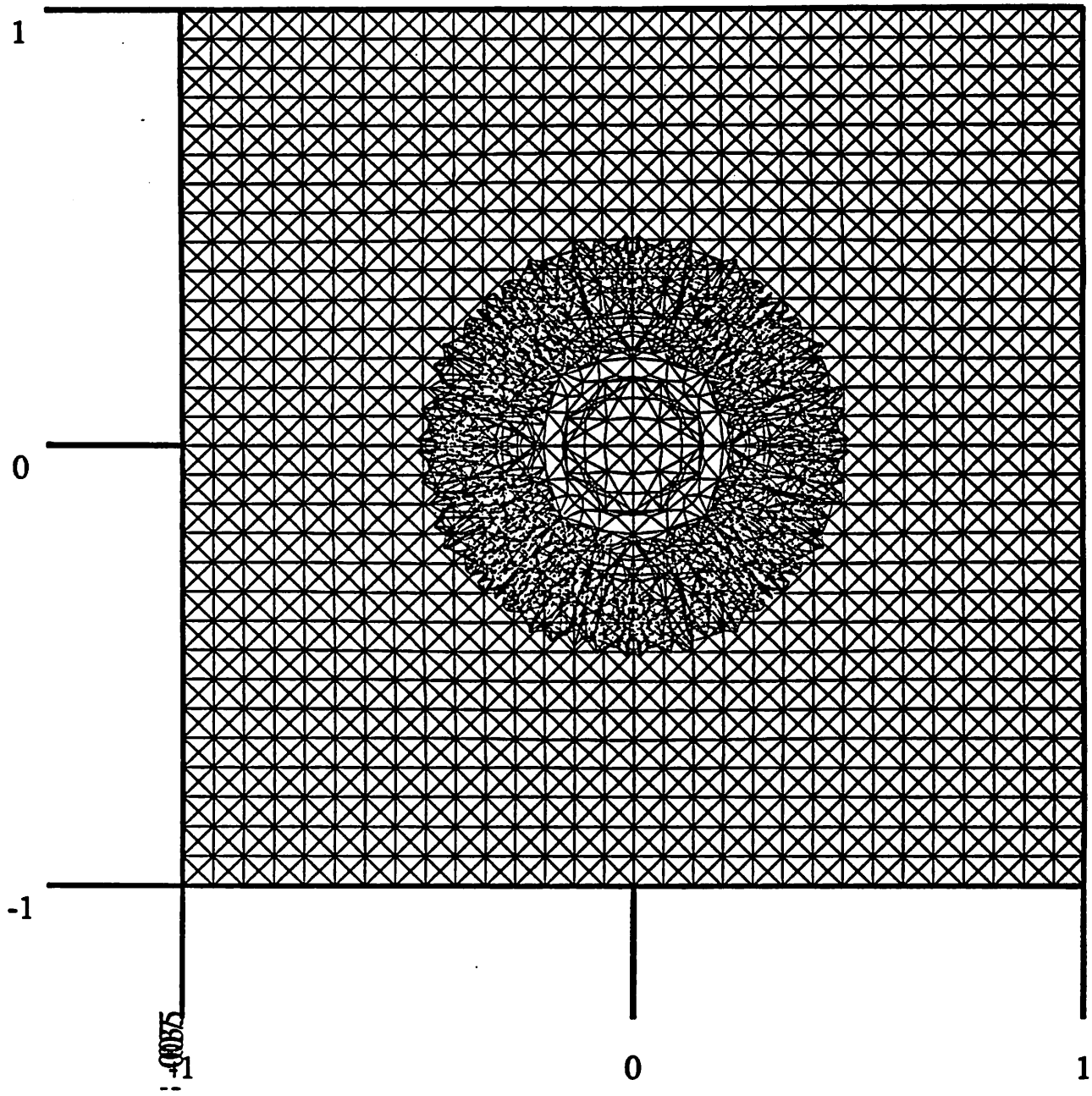


Figure 23

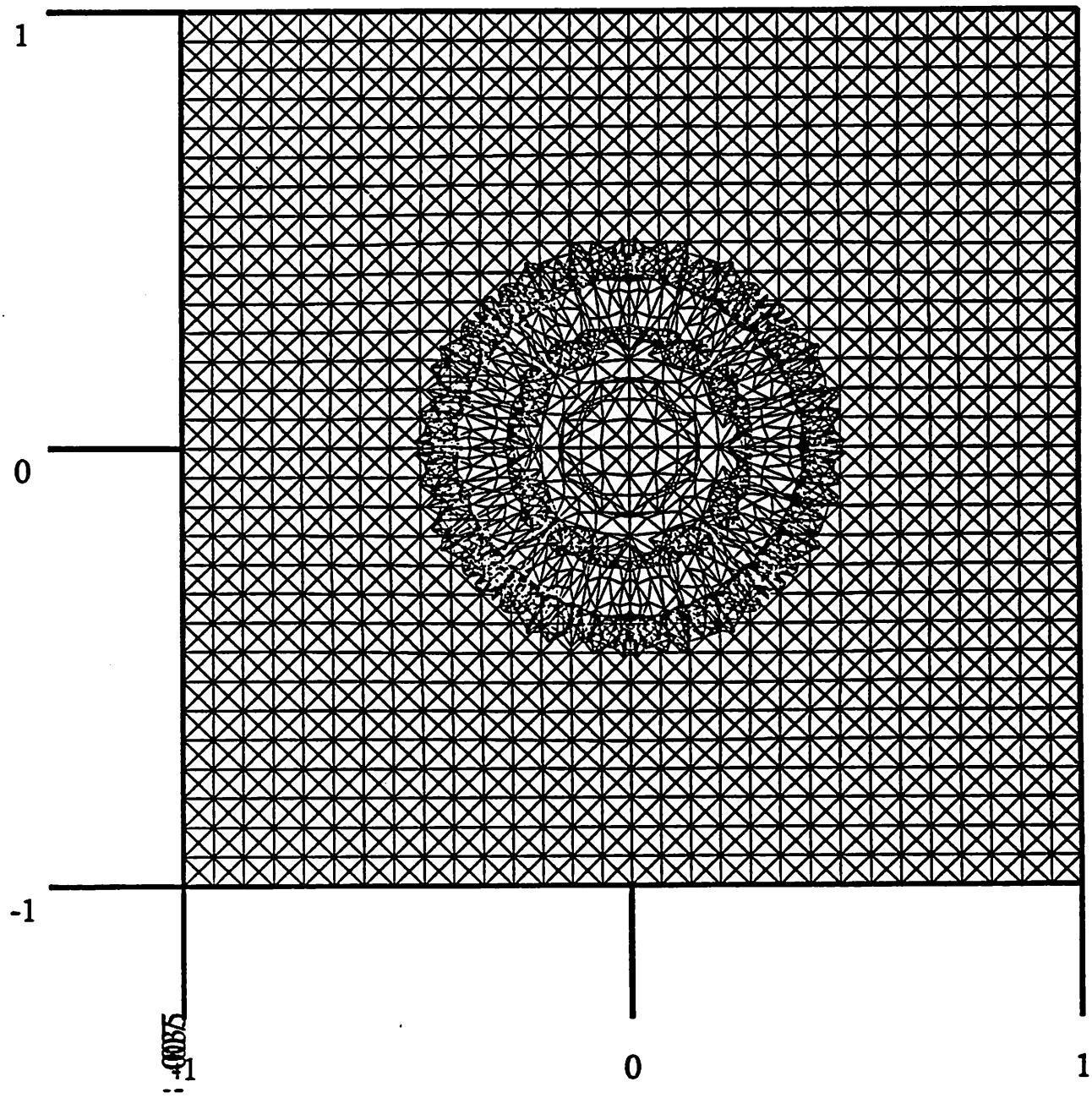


Figure 24

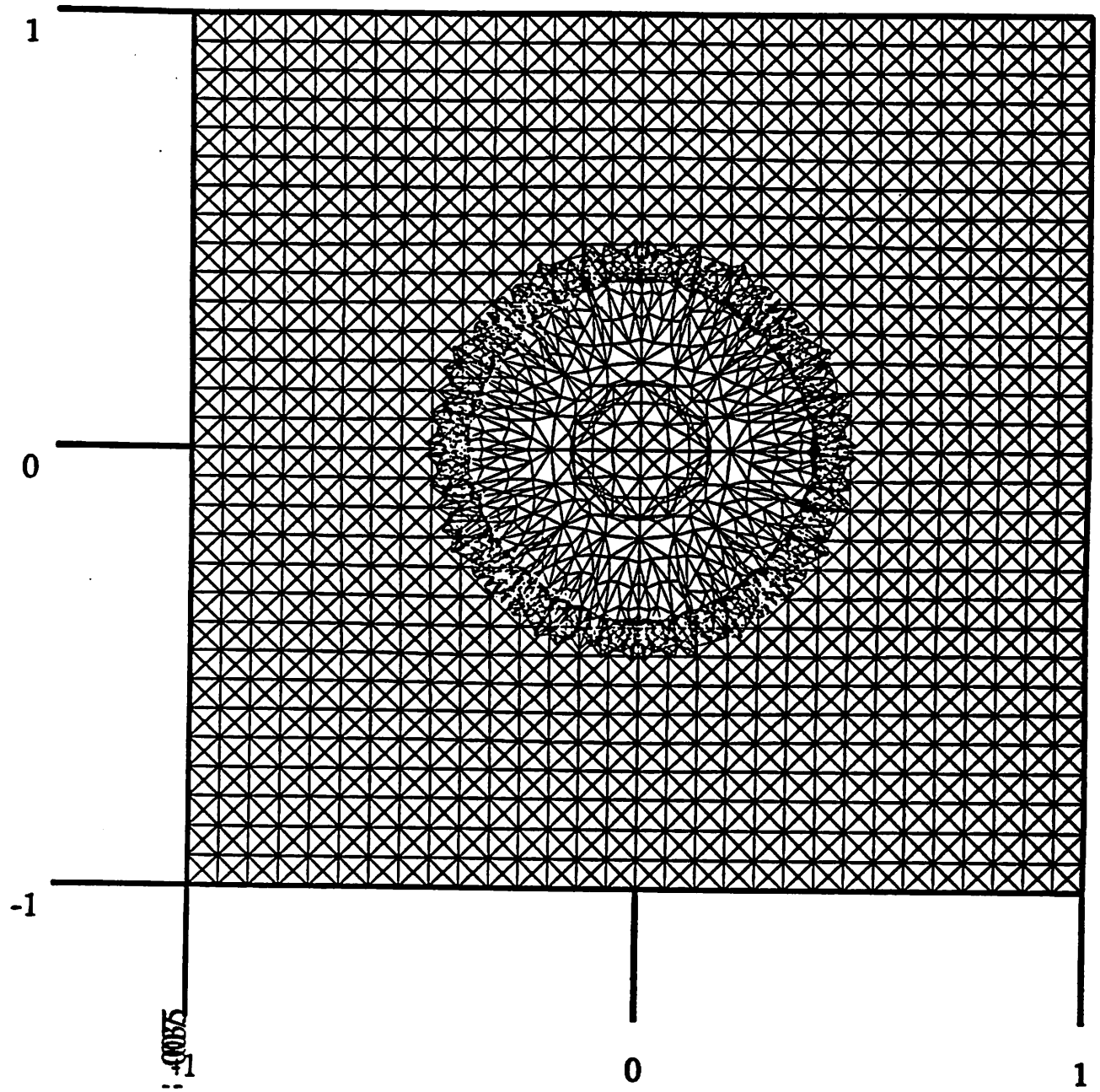


Figure 25

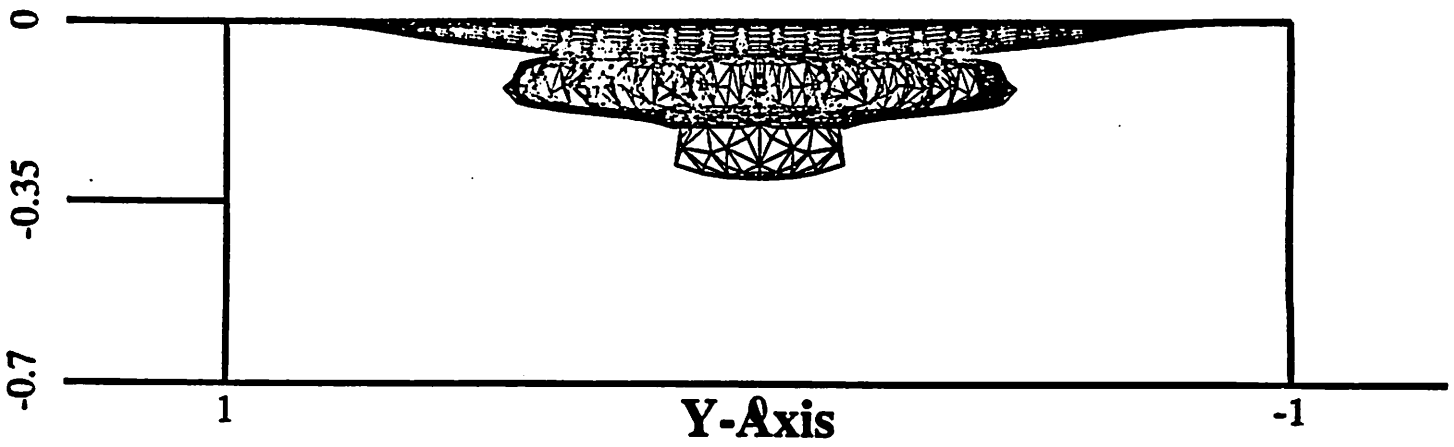


Figure 26

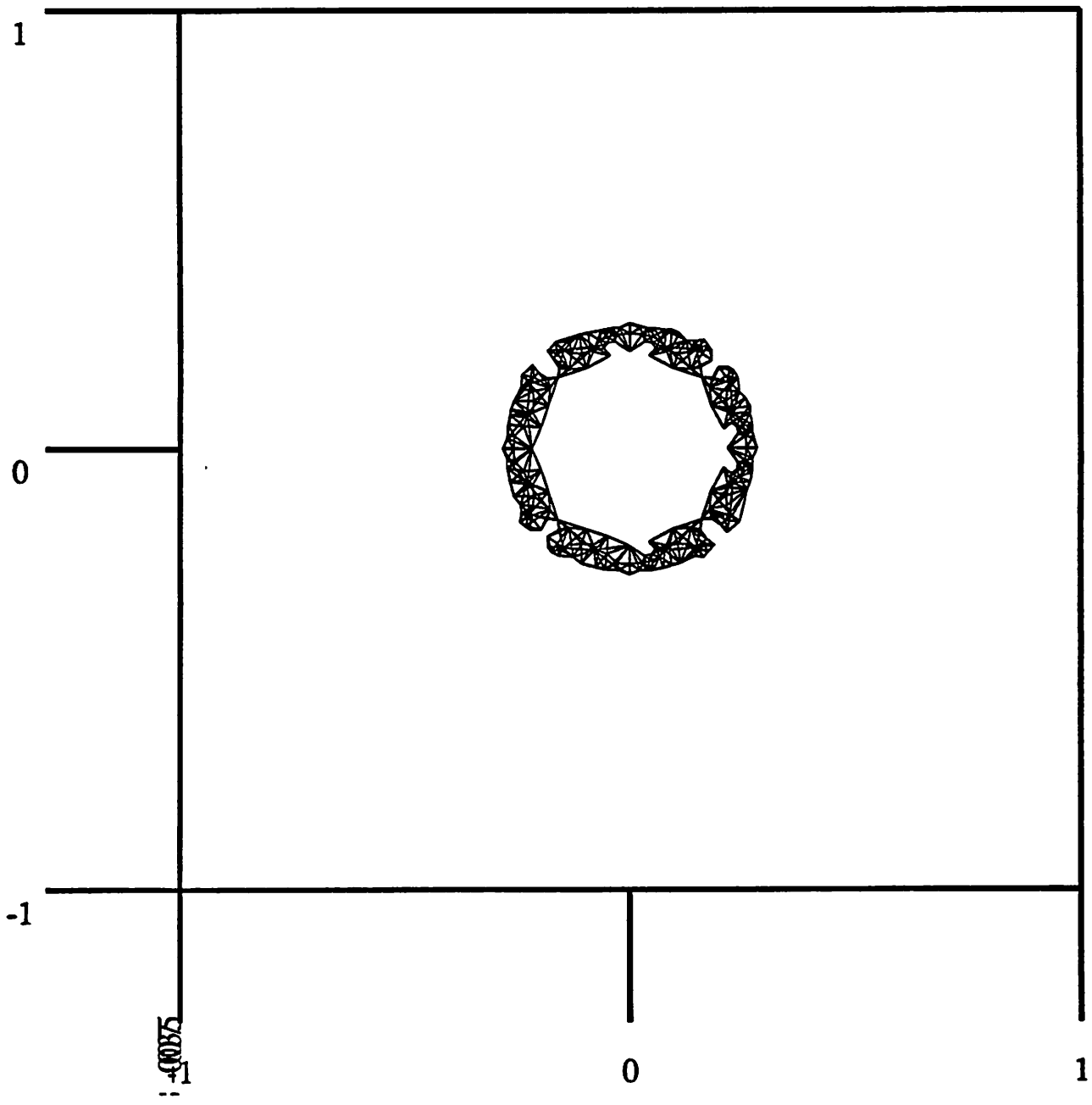


Figure 27

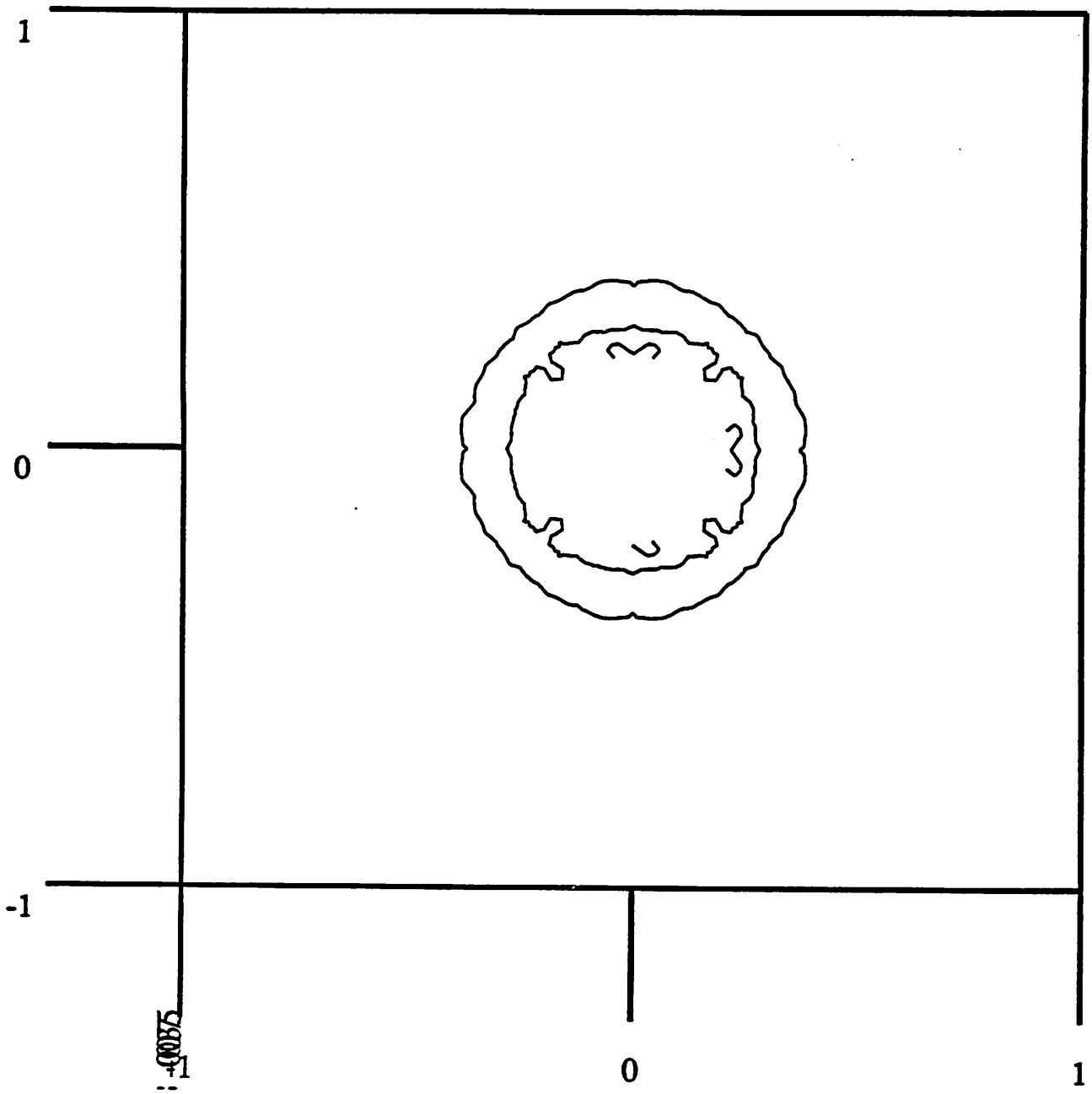


Figure 28