1990

Towards Automatic Grid Generation using Binary Space Partition Trees

George Vanecek

Report Number:
90-948

TOWARDS AUTOMATIC GRID GENERATION USING BINARY SPACE PARTITION TREES

George Vanecek

CSD-TR-948
January 1990

TOWARDS AUTOMATIC GRID GENERATION USING BINARY SPACE PARTITION TREES*

George Vaněček, Jr.*

Computer Sciences Department
Purdue University
Technical Report CSD-TR-948
CAPO Report CER-90-6
January, 1990

Abstract

The problem of discretizing three-dimensional solid objects is considered. The objects may be presented in standard boundary representation. We discuss constructing from this representation a binary space partition tree, a data structure especially well-suited to the geometric processing needed for grid generation. We also give algorithms for generating fixed-mesh grids and variable-mesh grids adaptively. The method has been implemented on top of a solid modeling system.

* This work has been supported in part by NSF grant CCR-86-19817.

1 Introduction

Solid modeling techniques are applied in mechanical simulation system for representing objects and detecting collisions in dynamic simulation systems [2, 9, 10, 26], and they are also applied in systems for solving partial differential equations (PDE) to represent and discretize 3D domains [19]. In such applications there is no single "best" representation schema for solids. Therefore, solids are modeled in a multiplicity of ways, including boundary representations, constructive solid geometry [18], and volume-based representations such as octrees [22] and binary space partition trees(BSP) [6]. Once a solid has been constructed, a basic operation is to determine the relative position of a point or a line-segment in relation to the solid. This is known as the point/solid or line/solid classification problem, and determines whether the entity to be classified is outside, inside, or on the boundary of the solid. Since a line may be partially inside and outside, the operation should also determine a segmentation of the line such that each segment is entirely either inside, or outside, or on the solid's boundary.

This problems is similar to the point location problem in computational geometry [13, 17], and to the clipping problem in computer graphics [23]. The line/polygon problem has been investigated for the case of constructive solid geometry; e.g., [25, 27]. A solution for octrees appears in Samet's books [21, 22]. Naylor solves the point/solid classification using binary space partition trees in [24].

The point/solid and the line/solid classification methods have uses in solving a wide range of diverse problems. One such problem is the construction of a grid that discretizes a given three-dimensional domain. That is, given a grid and a domain, the problem calls for the classification of each grid point with respect to the domain, and for the location of all intersections between the domain boundary and the grid lines.

This paper presents basic techniques for performing point and line segment classification in solids using binary space partition (BSP) trees. In Section 2 BSP trees are reviewed, and the data structures used to implement them is defined. Section 3 reviews various boundary representation, and their conversion to the BSP tree structure. In Section 4 the point/solid classification algorithm is given and its efficiency is considered. A new line/solid classification algorithm is presented in Section 5. In the remaining sections we solve the fixed grid generation problem using point and line classification with BSP trees, and we discuss our experience with an implementation.

2 Binary Space Partition Trees

A *binary space partition tree* is a binary tree data structure that represents the volume occupied by a solid. The solids considered have a distinct interior and exterior and have a boundary consisting of planar faces. The interior nodes of the BSP tree correspond to oriented hyperplanes; that is, to lines in 2D, and to planes in 3D. The two subtrees of an interior node correspond to the regions above and below the hyperplane. Leaf nodes correspond to regions that are either inside or outside the solid.

A two-dimensional solid consisting of six edges, and its BSP tree is shown graphically in Figure 1. The space containing the solid is partitioned by six splitting lines labeled a through f into eight regions labeled 1 through 8. Regions 5 and 6 are represented by leaf nodes labeled INSIDE, and the remaining six regions by leaf nodes labeled OUTSIDE. A left branches of an interior node accesses a subtree that represents regions above the hyperplane labeling the node. Similarly, the right subtree represents regions that are below the hyperlplane. We use the tree of Figure 1 throughout this paper.

An oriented hyperplane labeling an interior node is given by the equation $H = Ax+By+Cz+D$. A point p is considered above the plane H if $H(p) > \epsilon$, on the plane if $|H(p)| \leq \epsilon$ and below the plane if $H(p) < \epsilon$, for some small $\epsilon > 0$.

BSP trees were first used by Fuchs, Kedem and Naylor to solve the hidden surface(line) problem [6]. More important for our problem, BSP trees are efficient data structures for the point or line solid classification problem.

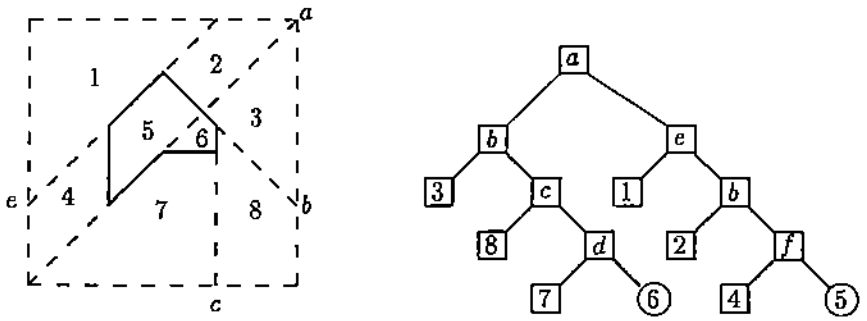

Figure 1: A 2D example of a solid and its BSP tree. Regions 5 and 6 are inside.

3 BRep to BSP Tree

We summarize the construction of a Binary Space Partition tree from a boundary representation. First we review boundary representation (BRep) data structures. Then we describe an algorithm for converting a BRep to a BSP Tree.

Solids with a manifold boundary can be represented by the winged-edge data structure developed by Baumgart [3]. The winged-edge data structure was extended by Braid to include multi-connected faces, by introducing the topological entity *loop*. In Braid's data structure, each face consists of one or more loops of edges, one for the outer edge loop, and one for each hole in the face [4]. Yamaguchi and Tokieda, taking a different approach, allow multi-connected faces by introducing of bridge edges [32]. Many variations on these data structures have been used since, including as Mäntylä's half-edge data structure [14], Guibas and Stolfi's quad-edge representation [7], and Hanrahan's face-edge representation [8]. Non-manifold representations, such as Weiler's radial-edge data structure, Karasick's star-edge data structure, and Vaněček's fedge-based data structure, have been introduced in [31, 11, 30].

Any of these boundary data structures can be converted to a BSP tree by a recursive function ComputeBSP. We are given a set F of faces bounding the solid. F is processed recursively as follows:

1. If F is a singleton then create an interior node with two leaf nodes as descendants. The left leaf is labeled INSIDE and the right leaf is labeled OUTSIDE. Skip all remaining steps.
2. If F is not a singleton, select a face f in F, and create an internal tree node r with descendants T_a and T_b. r is labeled with the hyperplane H in which f lies.
3. Delete from F the face f and all other faces coplanar to f.
4. Split the remaining faces in F into two sets F_a and F_b as follows:
   (a) F_a contains all faces of F that are above H.
   (b) F_b contains all faces of F that are below H.
   (c) The remaining faces are partitioned by their intersection with H. Each resulting face is added to F_a or F_b according to whether it is above or below H.
5. Recursively process the face sets F_a and F_b resulting in the subtrees T_a and T_b, respectively.

COMPUTEBSP depends on the ability to split a face by a plane into subfaces that do not cross the plane. Figure 2 shows graphically an example of splitting a face. In the example, a single face splits into five faces, three above, and two below the splitting plane. This fundamental operation is

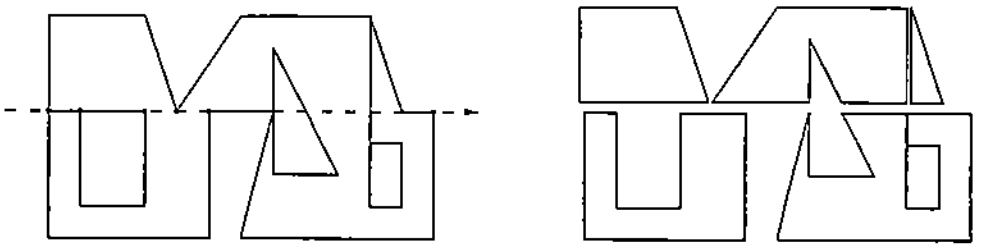

Figure 2: A single face shown before and after it is split by Function SPLITFACE into five faces. The face has three holes connected by bridge edges.

used by most solid modelers to perform set operations, and can be found described in [30, Section 5.4], [12], and elsewhere. The implementation varies only according to the particular data structure used. As an example, the data structure of the face shown in Figure 2 uses bridge edges which simplify the face splitting operation (refer to Section 6.3).

The shape of the resulting BSP tree depends on the order of choosing the faces. Although some have smaller height than others and therefore are more desirable, all possible trees represent the same volume of space occupied by the solid. More details on the construction can be obtained from Thibault and Naylor's work in [24, Page 155], or the original work by Fuchs in [6].

For a long time, the tightest upper-bound on the tree size has been $O(n^d)$ for n faces in a d-dimensional space. Recently, Paterson and Yao have shown that trees of size $O(n \log n)$ for n edges in 2D and $O(n)$ when the edges are orthogonal can be constructed [16]. In 3D, they have shown that trees of size $O(n^2)$ for n faces can be constructed, and prove a lower bound of $\Omega(n^{1.5})$. This result is based on the assumption that each hyperplane contains at least one face of the solid (what Paterson and Yao call an autopartition). They suggest that if this restriction is relaxed and hyperplanes are selected that do not contain any faces, the tree depth can be reduced. As an example where this is necessary, consider a convex solid with n faces. All autopartitions will result in a degenerate tree of depth n. In [30, Section 4.6.2], Vaněček proposes a hybrid decomposition technique which simply uses a regular decomposition method for the first few cuts depending on the number of given faces. That is, instead of choosing the first splitting plane to contain one of the faces of the solid, it cuts the solid in half by a plane orthogonal to one of the principal axes. As an example, the BSP tree of a spherical polyhedron with 144 faces can be reduced from a degenerate tree of depth 144 to a tree with depth 19 and average depth 11.

4 Point/Solid Classification

A point/solid classifier is a function that determines whether a point is inside, outside or on the boundary of a solid. Given a BSP tree representing the solid S, the function proceeds by "passing" the point starting from the root down to a leaf of the tree. The specific leaf node reached indicates whether the point is inside or outside the solid. If the point lies on the splitting plane of one of the interior nodes, the classification depends on a classification determined by evaluating both subtrees. Here the results could be one of INSIDE, OUTSIDE, or ON the boundary of the solid.

Consider the situation in which the point lies on the hyperplane of an interior node r. Here the point is passed into both subtrees, and is ultimately classified at several leaves below r. These classifications are percolated up to r and combined. The two classifications reaching r from the left and the right subtree are combined as indicated in Table 1. For details see [24, page 154]. Examples of the three possible outcomes are shown in Figure 3.

Above	Below	Result
in	in	in
in	out	on
out	in	on
out	out	out
in/out	on	on
on	in/out	on

Table 1: Result of a point/solid classification when the point lands on a splitting plane. See Figure 4.

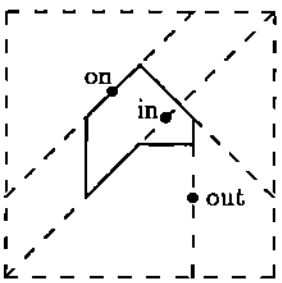

Figure 3: Example points lying on splitting lines. Refer to Table 1.

The average cost of the point/solid classification is linear in the height of the tree. The analysis depends on an argument counting the number of splitting planes that contain any given point [15]. In the worst case, the height of the tree is proportional to the number of faces. This occurs for convex bodies. In such situations, the efficiency can be improved by considering also splitting planes that do not contain faces [16, 30].

5 Line/Solid Classification

The line/solid classifier is a function that takes a line segment and returns what portions of the line segment are outside, on, and inside the solid. The function again uses the BSP tree of a solid and is an extension of the point/solid classification function described above. The line segment is filtered through the BSP tree. At each internal node the line segment is checked against the splitting plane. The segment is split, if it crosses the plane. The segments above and below the splitting plane are considered recursively, and their corresponding classifications are combined and returned.

The recursive procedure LSEGBSPCLASS for doing this follows. It accepts the two endpoints of a line segment in a list, and a BSP tree T, and returns the same list with additional points inserted along with the segment classifications IN, OUT or ON. The segment is represented as a singly-linked list of nodes. Each node n has three fields, POINT(n), CLASS(n), and NEXT(n). The class field, initially set to UNKNOWN, can be one of $\{\text{UNKNOWN}, \text{IN}, \text{OUT}, \text{ON}\}$.

```
recursive procedure LSegBSPClass(L : list; T : BSPTree)
begin
    if T = INSIDE or T = OUTSIDE then
      Class(L) ← LeafClass(Class(L), T)
    else begin
```

		c_2	
		INSIDE	OUTSIDE
	UNKNOWN	INSIDE	OUTSIDE
c_1	INSIDE	INSIDE	ON
	OUTSIDE	ON	OUTSIDE

Table 2: Function LEAFCLASS(c_1, c_2).

		c_2		
		ABOVE	ON	ABOVE
	ABOVE	ABOVE	ABOVE	CROSS
c_1	ON	ABOVE	ON	BELOW
	BELOW	CROSS	BELOW	BELOW

Table 3: Function LSEGPLCLASS(c_1, c_2).

```
  c ← PoPlClassify(Point(L), BSPPlane(T))
  c' ← PoPlClassify(Point(Next(L)), BSPPlane(T))
  case LSegPlClass(c, c') do
     ABOVE : LSegBSPClass(L, BSPAbove(T));
     BELOW: LSegBSPClass(L, BSPBelow(T));
     CROSS  : p ← CrossPoint(Point(L), Point(Next(L)), BSPPlane(T))
              Next(L) ← NewNode(p, Next(L))
              LSegBSPClass((if c = ABOVE then L else Next(L)), BSPAbove(T))
              LSegBSPClass((if c = BELOW then L else Next(L)), BSPBelow(T));
     ON       : L' ← Next(L)
              LSegBSPClass(L, BSPABove(T))
              while L ≠ L' do begin
                 N ← Next(L)
                 if Class(L) ≠ ON then LSegBSPClass(L, BSPBelow(T))
                 L ← N
              end;
     end case
  end
end /* LSegBSPClass */
```

When LSEGBSPCLASS reaches a leaf node, the classification of the line segment is changed according to the value of Function LEAFCLASS given by Table 2.

When LSEGBSPCLASS is given an internal node of a tree, it determines the relationship of the line segment with respect to the splitting line and performs one of four possible cases. The case is determined by Function LSEGPLCLASS given by Table 3. The line segment can be ABOVE, BELOW, CROSS, or ON the splitting plane. For the two cases of the line segment lying entirely above or below the splitting plane, LSEGBSPCLASS simply passes the segment down into the appropriate subtree.

When the line segment crosses the splitting plane, the intersection point is computed by CROSSPOINT and inserted into the list between the two endpoints of the line segment. The new segments are then passes down the appropriate subtrees, and their results are combined.

When the line segment lies in the splitting plane, some difficulties arise from the fact that the

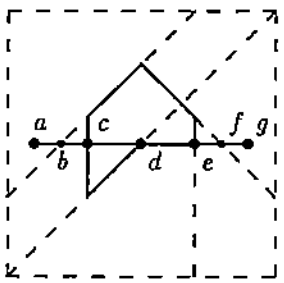

Figure 4: Example of classifying the line segment $[a, g]$.

classification depends on both subtrees and that the new segments resulting from classifying the segment in the left subtree need not be the same as the segments resulting from the right subtree. The solution to this case lies in first obtaining the INSIDE/OUTSIDE segments from the left subtree, and then classifying each one individually in the right subtree (refer to Table 2).

Figure 4 show graphically an example of a line segment $[a, g]$ that is partitioned into six segments. LSEGBSPCLASS is given the list

$$[(a, \text{UNKNOWN}), (g, \text{UNKNOWN})],$$

and returns the list with five additional points, namely,

$$[(a, \text{OUT}), (b, \text{OUT}), (c, \text{IN}), (d, \text{ON}), (e, \text{OUT}), (f, \text{OUT}), (g, \text{UNKNOWN})].$$

This example illustrates that the BSP tree partitions the line segment into more than the minimum number of segments required. After classification, adjacent segments with identical classification need to be merged. Merging the above list would produce

$$[(a, \text{OUT}), (c, \text{IN}), (d, \text{ON}), (e, \text{OUT}), (g, \text{UNKNOWN})].$$

The cost of the classification is proportional to the number of regions that the line segment intersects with, and the cost of merging the list is linear in the length of the list.

6 Grid Generation

We now consider the problem of constructing a grid discretizing a given three-dimensional domain. This is the task of the *domain processor* [20] in ELLPACK, a system for solving elliptical partial differential equations [20].

Each grid point must be classified with respect to the domain. Moreover, all intersections between the domain boundary and the grid lines need to be determined. Finally, the grid may have to satisfy additional constraints. For example, we may wish to refine the grid so that each domain vertex is on a grid line. For simplicity, we assume here that all grid lines are parallel to the principal axes. Note that this is not an essential assumption.

Consider a three-dimensional grid G, not necessarily uniform, occupying a box with extreme points $p_{\min} = (x_{\min}, y_{\min}, z_{\min})$ and $p_{\max} = (x_{\max}, y_{\max}, z_{\max})$. Each grid point can be indexed as (i, j, k), where $0 \leq i < n_x$, $0 \leq j < n_y$ and $0 \leq k < n_z$, corresponding to the coordinates (x_i, y_j, z_k). We find all intersections of the grid lines with the domain boundary. Such intersections are *boundary points*. Moreover, we determine for each grid point that is not a boundary point whether it is

- inside the solid but not adjacent to a boundary point,

- inside the solid and adjacent to a boundary point,
- outside the solid but not adjacent to a boundary point,
- outside the solid and adjacent to a boundary point.

We could construct a BSP tree for s, choose a grid G, and classify all of its $n_x n_y n_z$ grid points using the point/solid classification function described before. However, this does not determine the boundary points. Instead, we classify the grid lines with the line/solid classification function, and so determine all boundary points. The interior grid points are then readily obtained from the line segmentation. This approach has the advantage that the local boundary topology between adjacent grid points can be analyzed. For example, if the boundary crosses between adjacent grid points more than once, a situation that may be undesirable, then additional grid lines could be inserted.

6.1 Fixed Grid Generation

Given the specification of a grid, the X grid lines are the grid lines parallel to the x axis through grid point. Similarly, the Y grid lines and the Z grid lines are parallel to the y and z axis, respectively. We are given a solid s, in BRep. We are also given a grid specification by the extreme points $p_{\min} = (x_{\min}, y_{\min}, z_{\min})$ and $p_{\max} = (x_{\max}, y_{\max}, z_{\max})$, and the index bounds n_x, n_y, n_z. We obtain the set B of boundary points, and the set P of grid points that are inside or on the boundary of s. Fixed grid generation proceeds as follows:

1. From s construct the BSP tree T.
2. Generate and classify in T the Z grid line segments

$$\left\{\overline{((x_i, y_j, z_{\min}), (x_i, y_j, z_{\max}))}\right\},$$

   where $0 \leq i < n_x$, and $0 \leq j < n_y$. Replace each segment in Z with its partition. Note that all partition segments classified as OUTSIDE can be discarded. Moreover, process the X and Y sets the same way.
3. Construct the boundary point set B from the endpoints of the line segments in the X, Y and the Z sets:

$$B = \{p, q \mid (p, q) \in X \cup Y \cup Z\}.$$

   Note that all duplicates are eliminated.
4. Construct the grid point set P from the set X, or Y, or Z, choosing the set with the smallest cardinality: from each line segment in the set, obtain all the points that are also grid points, and add them to P.

Note that the lines of all three sets, X, Y, and Z, must be classified with respect to s in order that no boundary point is missed. For finding all grid points, however, only one of the sets is needed.

The cost of classifying the grid lines is $O\big((n_x n_y + n_y n_z + n_z n_x) L(m)\big)$, where $L(m)$ is the cost of the line/solid classification in a BSP tree with m nodes. The cost of obtaining the boundary points is $O(l \log l)$, with l line segments, resulting in at most $2l$ boundary points. The $\log l$ factor is the cost of eliminating duplicates from B. The grid points are obtained from one of the three grid line sets X, Y or Z and the cost of obtaining them is linear in the number of grid points in or on the boundary of the solid.

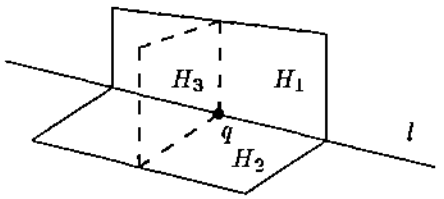

Figure 5: Placing a boundary point q on a grid point by introducing a plane H_3.

6.2 Adaptive Grid Generation

The conceptual simplicity of fixed grid generation suggests to explore adaptive versions of the algorithm that discretize the domain satisfying additional requirements. Among the possible constraints, we consider the following:

1. The spacing of the grid is such that the boundary of the domain does not intersect the edge of a grid element more than once.
2. The domain's boundary is fairly smooth in each grid element.
3. Domain vertices lie on grid lines.

These conditions are analogous to the constraints imposed in the 2D version of the problem in [20, page 229]. Note, however, that condition 3 can and should be relaxed for polyhedral models that approximate curved solids, for in such approximations many facets are generated and with it many vertices, so that condition 3 might generate a mesh that is too dense. We will comment on a relaxation of condition 3 later.

An automatic grid generation proceeds by creating an initial grid that contains grid lines passing in at least one principal direction through every vertex of the solid. To find these lines, construct planes parallel to the coordinate axes containing the vertices of s. Each vertex must lie in at least two perpendicular planes, whose directions could be chosen at random, say. Alternatively, all three planes may be chosen thereby placing each vertex at the corner of a grid element. This initial grid is then iteratively refined with planes defining additional grid lines as necessary.

More precisely, let q be a boundary point on grid line l at which the domain boundary intersects a grid line a second time. Note that l is defined by two perpendicular planes H_1 and H_2. We pass through q a plane H_3 perpendicular to l, thereby refining the grid and eliminating the excess boundary point in the adjacent regions. In particular, the intersections of H_3 with H_1 and H_2 define two new grid lines through q, as shown in Figure 6.2. Although this refinement of the grid eliminates multiple boundary points between two adjacent grid points under consideration, it has the same problem as the extended quadtree [1] construction and polytree [5] construction. For many solids, the regular decomposition approach (i.e., orthogonal cuts) cannot yield a finite grid satisfying certain criteria. For instance, criteria requiring that grid regions intersect with only a single face, an edge (i.e., two faces), or a vertex cannot always be satisfied. The usual means of dealing with the existence of regions that do not satisfy the criteria is to impose a minimum size requirement on a region, and defer its processing until needed. In our problem, the effect of resolving multiple boundary points between grid points by placing one of the boundary points on a grid plane is that it introduces new boundary points that may have to be moved to grid points in turn, and this cannot always be done. The only viable solution to this problem is to generalize the orthogonal grid and allow local non-orthogonal meshes.

We suggested that for polyhedral models that approximate curved solids not all vertices need lie on a grid line. Certain vertices in the BRep are the result of approximating a nonlinear surface, or curve, and it is these vertices that should be ignored. We suggest two methods for recognizing such vertices:

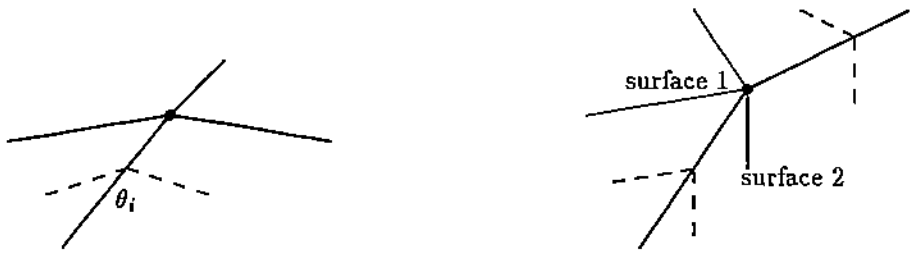

Figure 6: A vertex on one surface (a), and on two surfaces (b).

1. Consider the construction of a domain from known curved-surface primitives, using for example, regular Boolean operations [28]. The curved boundaries are polygonized, and the planar facets can be annotated with the information about the surface they approximate. A vertex in the BRep that is incident only to faces approximating the same surface, or the intersection of two surfaces, is present as a consequence of curved-surface faceting, and such vertices need not be placed on a grid line.

2. If no additional information about the faces is available, then the local topological and geometric information of the vertex can be used to infer if the vertex should be placed on a grid line or be considered present because of faceting. Given that a vertex has $n \geq 3$ adjacent faces (and edges), and the edge sectors have angles θ_j, two cases are recognized:

   (a) All incident edges have faces that are close to being coplanar. That is, $\theta_i \approx 180°$, for all i.

   (b) All but two incident edges satisfy $\theta_i \approx 180°$.

The two cases are illustrated by Figure 6.

The algorithm for the adaptive grid generation is now as follows:

1. From s construct the BSP tree T.

2. Let G_x, G_y and G_z be the *grid-plane sets* containing the grid planes in the x, y and z directions, respectively, initially empty.

3. For each vertex that is incident to three or more surfaces in the BRep of s, a plane passing through each of the three coordinates is inserted into the corresponding grid-plane list, unless already present.

4. For each $x_i \in G_x$ and each $y_j \in G_y$, generate and classify in T the grid line segment

$$\left\{\overline{((x_i, y_j, z_{\min}), (x_i, y_j, z_{\max}))}\right\}.$$

   Scan over the classified line segment and for each adjacent pair of grid points with $z_{k+1} - z_k > \delta$ that contains three or more boundary points, insert a new grid plane into G_z at every alternate boundary point. Here, δ is the minimum grid point separation.

5. Perform the analogous operations for the (G_y, G_z) and the (G_z, G_x) pairing.

6. Repeat steps 4 and 5 for all newly inserted grid planes. When no new grid planes have been inserted, stop.

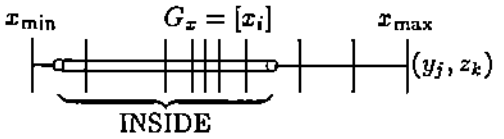

Figure 7: An example of checking a grid line by the adaptive algorithm. Classifying the grid line reveals the relationship between boundary points (circles) and grid points, G_x.

	Example 1 *block*	Example 2 *handle*	Example 3 *torus*
Number of Faces	6	209	1024
Number of Edges	12	357	2048
Number of Vertices	8	146	1024
Number of BSP Nodes	7	451	3490
BSP Tree Depth	6	19	29
Avr. BSP Tree Depth	4	11	14
BSP Creation Time(sec)	0.05	15	170

Table 4: BSP tree statistics for the three examples of Section 6.

The cost of the adaptive grid generation algorithm is $O(n_x n_y n_z L(m))$, where n_x, n_y, and n_z are the total number of grid planes generated. In most instances, one or two repeats of sweeping the x, y and z planes should be sufficient. The repetitions terminate when either all grid edges of length more than δ contain at most one boundary point, or some maximum limit of repetitions is reached. The complexity can be reduced by maintaining a set of grid elements (i.e., regions) with complex interiors and only processing those regions in the next pass. This suggests, once again, the creation of a local mesh that could be adapted recursively.

6.3 Experiments

The BSP tree algorithm and the grid generation methods presented in this paper have been implemented on a Symbolics 3620, on top of the solid modeler ProtoSolid [29]. When required, the data structure used by ProtoSolid for representing the boundary of a solid is converted to a BSP tree by the Function COMPUTEBSP of Section 3. Fixed grids are then generated as described before. To illustrate the method, three examples are presented.

Example 1 A unit cube is rotated 45° around the z axis.

Example 2 A luggage handle is created that contains two holes, and two notches on the bottom (see Figure 8). The handle was constructed from a torus, three cylinders and a block with Boolean set operations. The boundary of the handle contains 209 faces, 357 edges and 146 vertices. A fixed grid of size (10,10,20) is shown in Figure 9.

Example 3 A torus is positioned on the $x - y$ plane and rotated 45° around the x axis, and $-45°$ around the z axis, as shown in Figure 10. This example illustrates a solid with a large number of faces. The object's boundary contains 1024 faces, 2048 edge and 1024 vertices. It took 156 seconds to create the BRep alone. The BSP tree has 3490 nodes consisting of 924 INSIDE, and 2333 OUTSIDE nodes. A fixed grid of size (20,20,20) is shown in Figure 11.

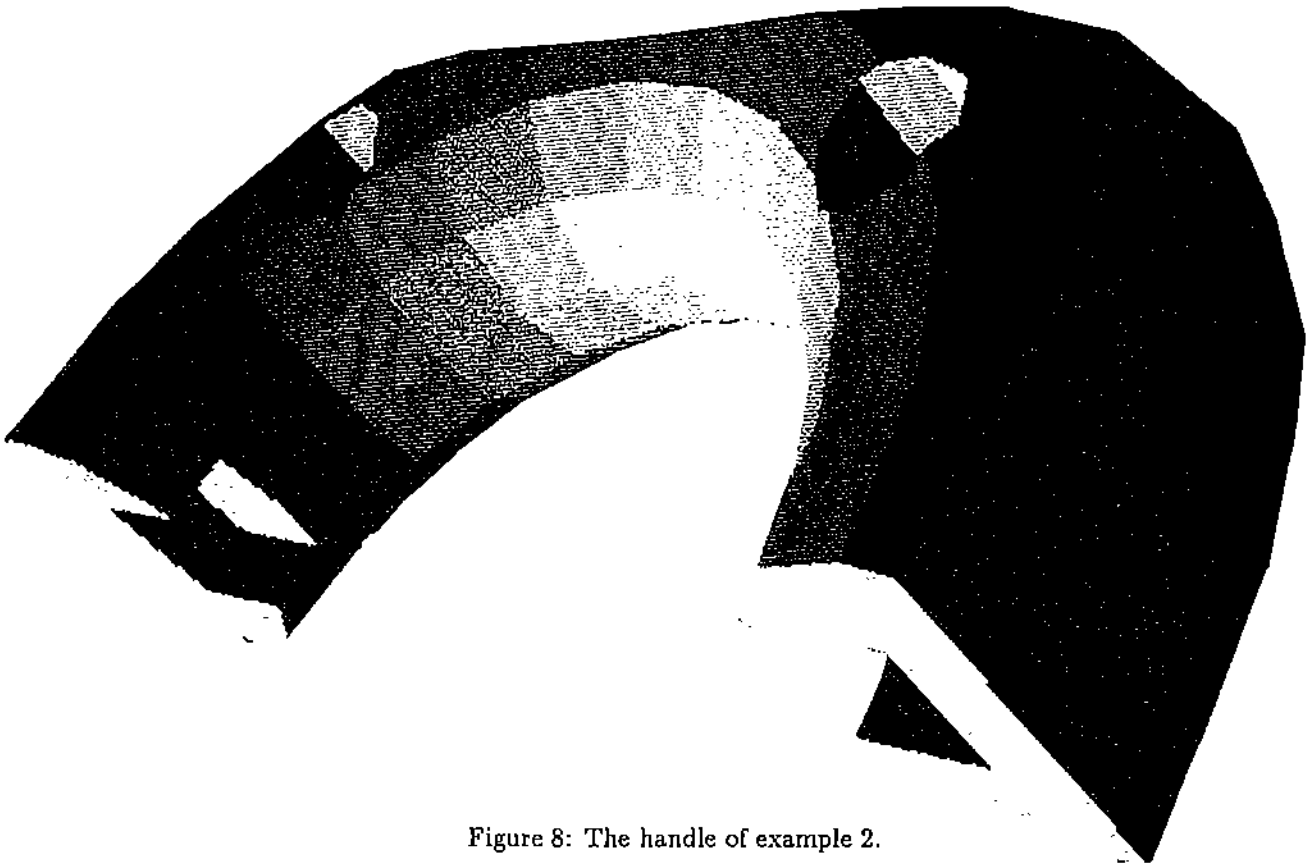

Figure 8: The handle of example 2.

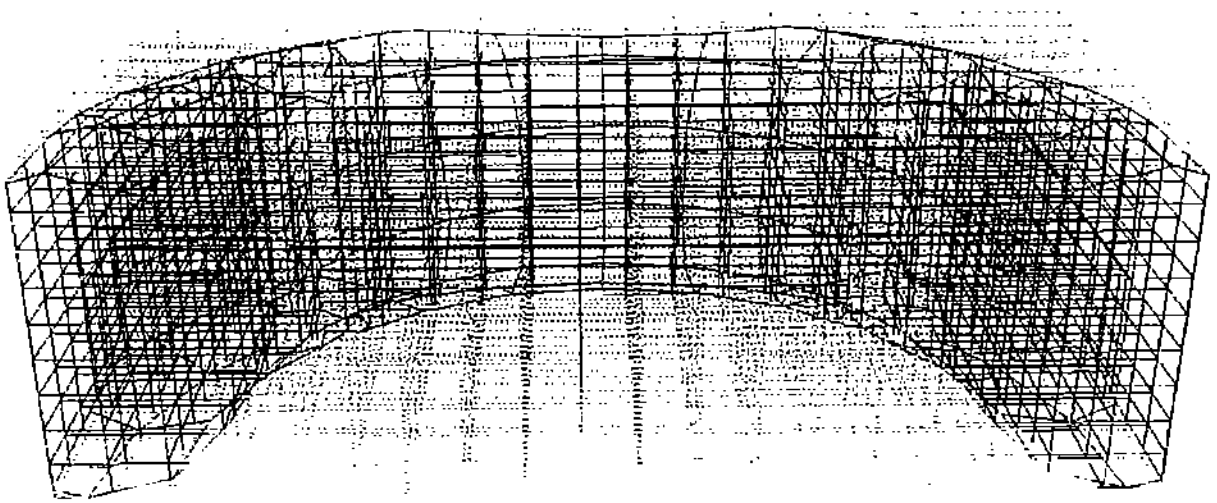

Figure 9: A (10,10,20) grid of example 2.

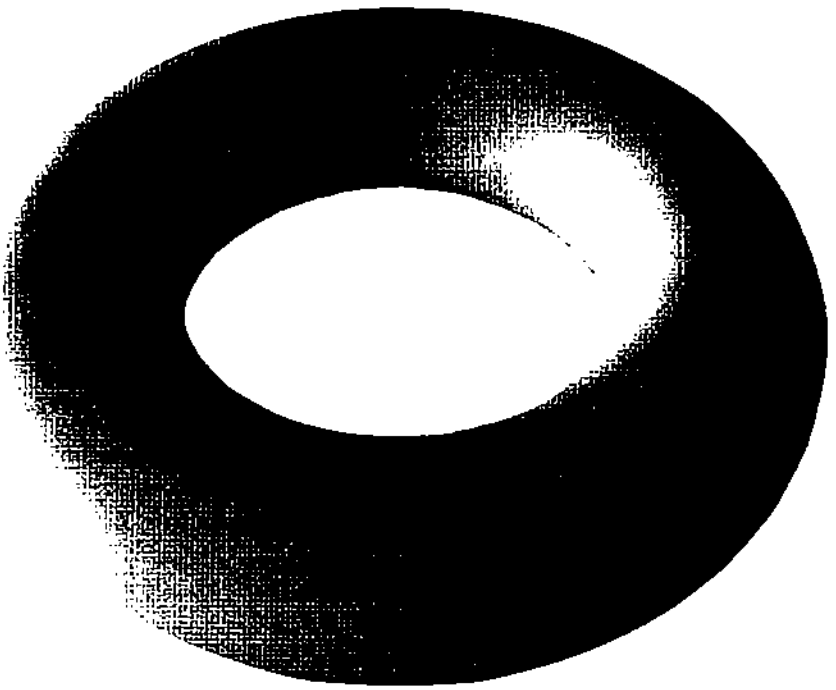

Figure 10: The torus of example 3.

	Example 1 *block*	Example 2 *handle*	Example 3 *torus*
Grid Size (n_x, n_y, n_z)	(5,5,5)	(10,10,20)	(20,20,20)
Number of In/On/Out Lines	32/26/87	472/43/834	493/0/1655
Number of In/On/Out Points	15/60/50	647/98/1255	587/0/7413
Grid-Point Class. Time(sec)	0.4	8.8	82
Grid-Line Class. Time(sec)	0.6	27	179

Table 5: Fixed grid generation summary for the three examples.

Table 4 presents the size of the BReps and BSP trees of the three examples. For the BReps, the number of faces, edges and vertices are given. For the BSP trees, the total number of nodes in the trees are listed, along with the maximum tree depth, the average tree depth and the time it took to create the BSP trees.

In Table 5 a single fixed grid is generated for each example, and the inside/outside points and line segments are counted. The times it took to perform point/solid classification of the grid points, and the line/solid classification of the grid lines are also given.

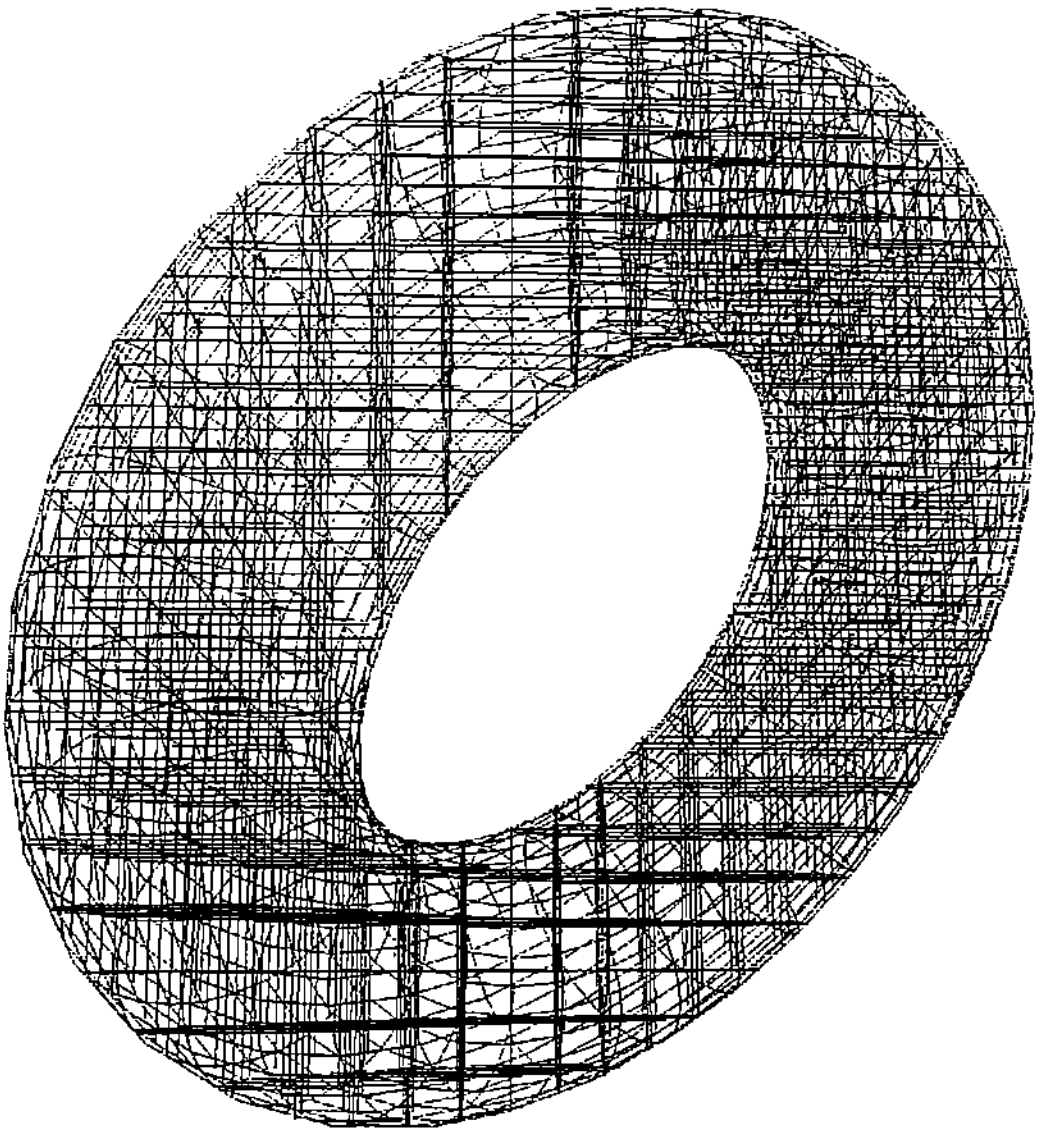

Figure 11: A (20,20,20) grid of example 3.

References

[1] D. Ayala, P. Brunet, and I. Navazo. Object representation by means of nonminimal division quadtrees and octrees. *ACM Transactions on Graphics*, 4(1):41–59, January 1985.

[2] C. Bajaj, W. R. Dyksen, C. M. Hoffmann, Elias N. Houstis, J. T. Korb, and J. R. Rice. Computing about physical objects. Technical Report CSD-TR-696, Purdue Unviersity, July 1987. CAPO Report CER-87-1.

[3] B. Baumgart. Winged–edge polyhedron representation. Technical Report CS–320, Stanford University, Stanford, CA, 1972.

[4] I. C. Braid, R. C. Hillyard, and I. A. Stroud. *Stepwise construction of polyhedra in geometric modeling*, pages 123–141. Academic Press, New York/London, 1980.

[5] I. Carlbom, I. Chakravarty, and D. Vanderschel. A hierarchical data structure for representing the spacial decomposition of 3d objects. *IEEE Computer Graphics & Applications*, 5(4):24–31, 1985.

[6] H. Fuchs, Z. M. Kedem, and B. F. Naylor. On visible surface generation by a priori tree structures. *Conf. Proc. of SIGGRAPH '80*, 14(3):124–133, July 1980.

[7] L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computations of voronoi diagrams. *ACM Transactions on Graphics*, pages 74–123, April 1985.

[8] P. M. Hanrahan. Creating volume models from edge-vertex graphs. *Computer Graphics*, 16(3):77–84, 1982.

[9] C. M. Hoffmann and J. E. Hopcroft. Simulation of physical systems from geometric models. *IEEE Journal of Robotics and Automation*, RA–3(3):194–206, June 1987.

[10] C. M. Hoffmann and J. E. Hopcroft. Model generation and modification for dynamic systems from geometric data. *CAD Based Programming for Sensory Robots*, F50:481–492, 1988.

[11] M. Karasick. *On the Representation and Manipulation of Rigid Solids.* PhD thesis, McGill University, 1988.

[12] Jiři Kripac. Algorithm for splitting planar faces. *Computer Aided Design*, 19(6):293–297, August 1987.

[13] D. T. Lee and F. P. Preparata. Location of a point in a planar subdivision and its applications. *Proc. 8th ACM Symp. Theory of Computing*, pages 231–235, May 1976.

[14] M. Mäntylä. *An introduction to SOLID MODELING.* Computer Science Press, 1988.

[15] B. F. Naylor. *A Priori Based Techniques for Determining Visibility Priority for 3-D Scenes.* PhD thesis, University of Texas at Dallas, May 1981.

[16] M. S. Paterson and F. F. Yao. Binary partitions with applications to hidden-surface removal and solid modeling. In *Proceedings of ACM on Computational Geometry*, pages 23–32. ACM, 1989.

[17] F. P. Preparata and M. Ian Shamos. *COMPUTATIONAL GEOMETRY, An Introduction.* Springer-Verlag, 1985.

[18] A. A. G. Requicha. Representations for rigid solids: Theory, methods, and systems. *ACM Computing Surveys*, 12(4), December 1980.

[19] J. R. Rice. Supercomputing about physical objects. Technical Report CAPO Report CER–87–6, Computer Science Department, Purdue University, West Lafayete, IN 47906, September 1987.

[20] J. R. Rice and R. F. Boisvert. *Solving Elliptical Problems Using ELLPACK*. Springer-Verlag, New York, 1985.

[21] H. J. Samet. *Applications of Spatial Data structures: Computer Graphics, Image Processing, and GIS*. Addison–Wesley, Redding, MA, 1989.

[22] H. J. Samet. *Design and analysis of Spatial Data Structures: Quadtrees, Octrees, and other Hierarchical Methods*. Addison–Wesley, Redding, MA, 1989.

[23] I. E. Sutherland and G. W. Hodgman. Reentrant polygon clipping. *Communications of the ACM*, 17(1):32–42, January 1974.

[24] W. C. Thibault and B. F. Naylor. Set operations on polyhedra using binary space partitioning trees. *ACM Computer Graphics SIGGRAPH '87*, 21(4):153–162, July 1987.

[25] R. B. Tilove. A study of geometric set-membership classification. Master's thesis, University of Rochester, Rochester, N.Y., November 1977.

[26] R. B. Tilove. Extending solid modeling systems for mechanism design and kinematic simulation. *IEEE Computer Graphics & Applications*, pages 9–19, June 1983.

[27] R. B. Tilove and A. A. G. Requicha. Closure of boolean operations on geometric entities. *Computer Aided Design*, pages 219–220, September 1980.

[28] G. Vaněček Jr. Obtaining boundaries with respect: A simple approach to performing set operations on polyhedra. Technical Report CSD-TR-920, Purdue University, Department of Computer Science, West Lafayette, IN 47907, October 1989.

[29] G. Vaněček Jr. Protosolid: An inside look. Technical Report CSD-TR-921, Purdue University, Department of Computer Science, West Lafayette, IN 47907, October 1989.

[30] G. Vaněček Jr. *Set Operations on Polyhedra using Decomposition Methods*. PhD thesis, University of Maryland, College Park, Maryland, June 1989.

[31] K. Weiler. *Topological Structures for Geometrical Modeling*. PhD thesis, Rensselaer Polytechnic Institute, Troy, N.Y., August 1986. Technical Report TR-86032.

[32] F. Yamaguchi and T. Tokieda. Bridge edge and triangulation approach in solid modeling. In Tosiyasu L. Kunii, editor, *Frontiers in Computer Graphics '84*, pages 44–65. Springer-Verlag, 1985.