EFFICIENT POLYHEDRON INTERSECTION AND UNION

Wm. Randolph Franklin

Electrical, Computer, and Systems Engineering Dept.
Rensselaer Polytechnic Institute
Troy, NY, 12181
518-270-6324

ABSTRACT

A algorithm for calculating the set theoretic combinations, such as union, intersection and difference, of two polyhedra is described. The polyhedra are in the Eulerian surface description format, specified by their edges and faces. They can have multiple components and nested holes. This algorithm uses an adaptive grid to handle complex objects with thousands of faces without the overhead of comparing every pair of faces to find which pairs intersect. It produces a list of new faces with tags indicating which of these faces are included in each of the set combinations, so that all the combinations of two polyhedra are produced at once with no extra cost.

KEYWORDS: polyhedron intersection, polyhedron, intersection, union, set operation, Eulerian operation graphics, geometry, computer aided design.

INTRODUCTION

As computer aided design progresses, it is becoming apparent that there are various low level operations that are common to several applications and whose execution time is an important factor in these applications. One such low level operation is combining two polyhedra to obtain the intersection, union, or difference. Polyhedron intersection is used in places such as:

1) Solid modelling, where the user is forming complex objects from a small set of primitives,

2) NC tool verification, where we wish to know the volume cut out of an object by a drill,

3) Interference detection, such as fitting parts together, where we wish to know whether two parts are trying to occupy the same space.

There are several solutions to this problem, each with various advantages and disadvantages. Some of them are:

Eastman and Yessios [6] give a general algorithm for combining 2-D polygons. It handles polygons with nested subpolygons, and also cases where two polygons are partially coincident, although this is messy. It works by finding all the intersections between the polygons' edges, and then "threading" or traversing around the pieces of edges to determine the resulting algorithm. This method is linear in the complexity of the input and the output, provided that an efficient edge intersection routine is used. However, it is inherently sequential, and would be difficult to execute in parallel. A more critical problem is that if this method is extended to 3-D, the threading becomes much more difficult and error-prone. The nature of threading is such that one wrong decision renders the rest of that thread of edges (or faces in 3-D) completely wrong, so special cases must be handled carefully.

Maruyama [15] gives a procedure for determining whether two polyhedra intersect by comparing the faces pair by pair. However, he does not determine the intersection, only whether it is null. We use an extension of that method in this paper.

Baer, Eastman, and Henrion [1] give a good summary of geometric modelling systems, which "shape operations" (i.e. intersection etc.) they perform, and how they do it.

Lozano-Pérez and Wesley [12] describe collision avoidance of moving objects, which is an operation where this sort of algorithm can be useful.

Tilove [19] considers important questions of what intersection and so on mean in the abstract, and introduces "regularized set operators" to answer them. He also gives recursive methods of intersection and union of objects defined as as combinations from a small family of primitives. These methods are used in P.A.D.L., one of the best known geometric processors. However, this method of constructing objects can require many primitives to be added and subtracted to form some quite simple objects. Each face of the object might be the result of a complete primitive component so the database can need a complete object for every original face. In addition, some quite simple operations, such as determining the volume of the object, are impossible to do, except by Monte Carlo techniques. Finally, this method has not yet been extended to intersecting objects with thousands of faces.

Boyse [4] gives an algorithm for determining whether two objects interfere, where one of the objects can be moving along a straight or circular trajectory. His objects are composed of vertices, straight edges, and flat faces, so for two objects to interfere it is necessary and sufficient for an edge of one object to pass through a face of the other object. This he tests for. However, this does not extend to producing the intersection.

If one of the objects is a convex polyhedron, then we can use the fact that it is the intersection of a number of semi-infinite half-spaces, one for each face, by intersecting them against the other polyhedron, one by one. This is easier if the other polyhedron is also convex. However, the only way that this method generalizes to non-convex objects is to partition them into convex pieces, which increases the complexity.

Another, completely different method is that of Meagher [16] who approximates the object by an octree of different sized cubes. This has the advantage that no floating point operations are needed to combine the objects, but also the disadvantage that the surface of the object is not represented exactly so that shading is complicated and poorer.

Baumgart [2, 3] has a good geometric manipulation system with polyhedron combination operators, using a different algorithm. Braid [5] has another polyhedron combination algorithm. Parent [17] describes another one.

An independently developed algorithm similar to the one presented here, except that it does not use an adaptive grid, has been used for some time in P.A.D.L. [21]. It has been presented at several short courses, but has not been described in the open literature.

Another independent similar algorithm is described in [20], although Franklin has been unable to obtain a copy. Again, it does not include a means of handling complex objects. It has been implemented, [18, 22], but cannot handle two objects with a common face, or an edge of one lying in a face of the other.

The principal contribution of the algorithm presented here is that, through its use of an adaptive grid, it can process scenes with thousands of faces. In addition, unlike some other algorithms, it produces all the boolean combinations at little more than the cost of producing one. Since there are no complicated data structures such as pointers, it is easier to implement in low level languages such as fortran. Of course it can handle nested holes and multiple components.

This algorithm is part of the Kepler geometric manipulation system described in [11]. It is now being implemented. An earlier algorithm for polygons was implemented in 1973.

POLYGON COMBINATION ALGORITHM

In order to better understand the polyhedron intersection algorithm, we will first present a polygon intersection algorithm that uses the same principles. It is:

1. Overlay an adaptive grid on the scene, whose fineness is a function of the number and length of the edges in the input polygons, P_α and P_β. Adaptive, or variable, grids are described in [8, 9, 10].

2. Initialize an empty set, S_1, that will hold ordered pairs of the form (Cell, Edge).

3. For each edge in either polygon, determine which cells it passes through, and add ordered pairs to S_1. We are not cutting the edges into pieces where they cross a cell boundary, as in Warnock's algorithm; we are merely noting which cells each edge passes through and adding an element to S_1 for each one.

4. Sort S_1 by Cell so that we have in one place all the edges passing through each cell.

5. Initialize an empty set, S_2, that will hold ordered pairs of intersecting edges.

6. For each cell, C:

a) Intersect all the edges of P_α in C with all the edges of P_β in C.
b) For each pair of edges, E_1 and E_2, found to intersect, write two ordered pairs, (E_1, E_2) and (E_2, E_1) to S_2. Overlapping collinear edges are considered to intersect.

7. Sort S_2 by the first edge of each pair, so that we have in one place all the edges intersecting each edge.

8. Initialize an empty set, S_3, that will hold segments. A segment is a whole edge or a piece of an edge of P_α or P_β that will not be further subdivided, and that may be used in the resulting polygon. There are two types of segments: those that come from an edge of only one polygon, and those that are common to an edge of both polygons because those edges overlapped. For the former type, store not only the segment's endpoints with it, but also which polygon it came from. For the latter type, store not only the endpoints, but also whether or not the two edges it came from are oriented in the same or in opposite directions.

9. For each edge, E, find the other edges that intersect it, from S_2. Find the points at which they intersect E, sort those points along E, and use them to split E into segments. If another edge is collinear with E, then it is considered to intersect E at each of their endpoints. If the segment is not the result of collinear edges, then if one endpoint of S is an intersection, then we can determine whether S is in the other polygon by examining the direction of the other edge in the intersection. Else we must do a point inclusion in polygon test. Add each segment to S_3.

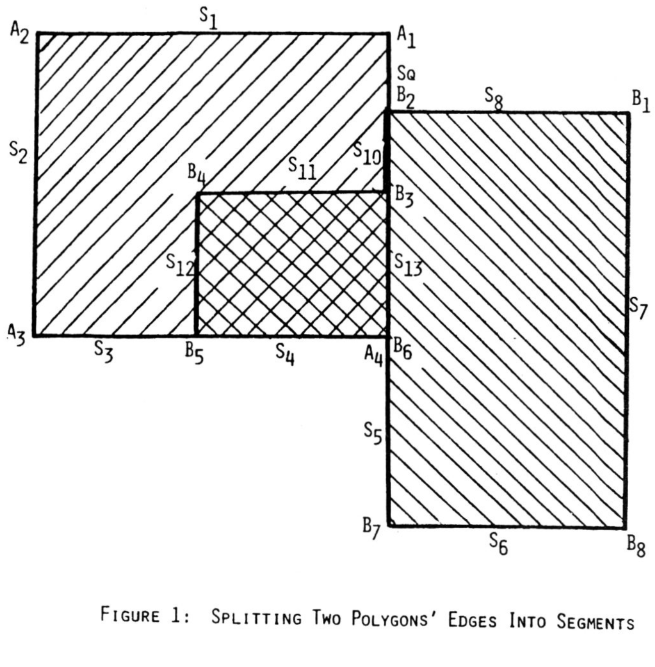

FIGURE 1: SPLITTING TWO POLYGONS' EDGES INTO SEGMENTS

OPERATION DESIRED					
P_α or P_β	*				
P_α and P_β		*			
P_α and not P_β			*		
P_β and not P_α				*	
P_α exclusive or P_β					*
EDGE SEGMENTS TO INCLUDE					
On P_α inside P_β		+		-	-
On P_α outside P_β	+		+		+
On P_β inside P_α		+	-		-
On P_β outside P_α	+			+	+
On both, in same direct.	+	+			
On both, in opposite directions			+	-	

Table 1

Which Segments To Include For Each Result

For example, see figure 1, which shows the segments caused by combining two simple polygons. In it, P_α has four vertices and P_β eight. Together, their 12 edges form 13 segments, of which two are common to both polygons. S_4 includes edges of P_α and P_β in the same direction, while S_{10} includes edges of P_α and P_β in opposite directions.

10. Depending on the particular result desired, select the appropriate subset of the edge segments in S_3, as shown in table 1. To use the table, select the operation desired, read across that row to the column marked "*", and then read down that column. For each row below with a "+" or "-", read to the left to find the type of segment to include. A "+" means to include that type of segment in the same direction as it appeared in the input polygon, while a "-" means to include it, but reverse its direction. If the segment is on both polygons in opposite directions, then "+" means to include it in the direction in which it is on P_α, and "-" means the other direction.

For example, if we want P_α and-not P_β, we will include all segments on P_α outside P_β, all segments on P_β inside P_α, but reversed in direction, and all segments on both P_α and P_β in opposite directions, included in the direction that they are on P_α.

If we desire to have the edges of the resulting polygon in order, then we can do the following steps:

11. Initialize an empty set, S_4, that will contain ordered pairs of a vertex and a segment ending on it.

12. For each segment that was selected, add two ordered pairs, for its two endpoints, to S_4.

13. Sort S_4 by the vertex to get all the segments on each vertex together.

14. Further sort all the segments on each vertex in clockwise order around the vertex.

15. Finally traverse the resulting planar graph to pull out all the regions in order. These are the components of the resulting polygon.

To better understand this algorithm, it is fruitful to consider the successive sets of data that are created. They are:

1. {Input edges}
2. {Cell, Edge in it}
3. {(Edge, Intersecting edge)}
4. {(Edge, {All intersecting edges})}
5. {Segments}
6. {Selected segments}
7. {(Vertex, Segment at it)}
8. {(Vertex, {All segments at it})}
9. Edges of resulting polygon in order.

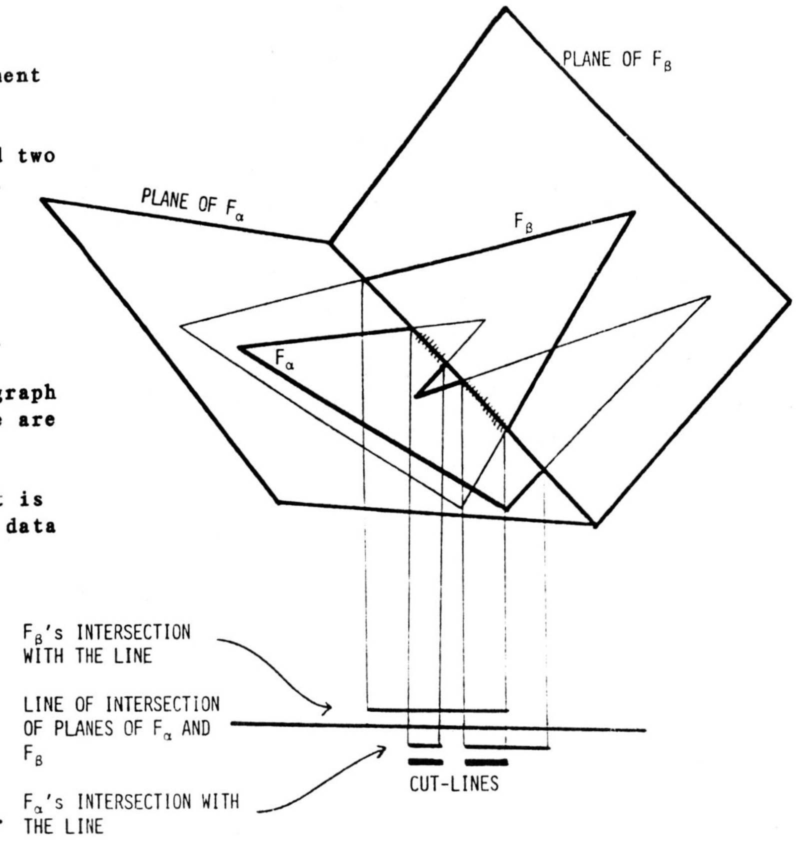
PLANE OF F_β
PLANE OF F_α
F_β
F_α
F_β's INTERSECTION WITH THE LINE
LINE OF INTERSECTION OF PLANES OF F_α AND F_β
F_α's INTERSECTION WITH THE LINE
CUT-LINES

FIGURE 2: FINDING THE CUT-LINES FROM THE INTERSECTION OF F_α AND F_β

POLYHEDRON COMBINATION ALGORITHM

The 3-D version of the algorithm is similar to the 2-D version, albeit more complicated.

1. Establish a 2-D coordinate system for each face plane. This will allow us to refer to points in the plane of a face with coordinate pairs and to use all our 2-D graphic routines such as point-in-polygon testing. Also calculate the transformation matrices between 2-D and 3-D.

2. From the number and area of the faces, determine the expected number of pairs of faces that intersect, and from this calculate the resolution, B, of a 3-D grid to lay over the polyhedra, P_α and P_β.

3. Initialize an empty set, S_1, that will hold (Cell, Face) ordered pairs. "Face" means not only the corner vertices of the face, but also which polyhedron the face belongs to.

4. For each face in either polyhedron, determine which cells it passes through, and add elements to S_1. For speed, we can enclose the face in a box and use all the cells that the box passes through, since a few extra cells don't matter. Alternatively, a form of Bresenham's algorithm can be used to find the exact cells.

5. Sort S_1 by cell number to get in one place all the faces in each cell.

6. Initialize a set, S_2, that will contain ordered triples of the form (face, cut-line, other face). A CUT-LINE is a line in the plane of the face across part (or all) of it, but not extending outside the face. The cut-lines will later be used to cut the face into facets. The other face is the face that intersected this face to create the cut-line.

7. Initialize a set, S_3, that will contain triples of the form (face, edge, cut-point). The edge is an edge of the face. The CUT-POINT is a point on the edge where a cut-line meets the edge. Cut-lines don't have to end at edges

of the face (they can stop at a point inside the face), but many do end at an edge.

8. Initialize a set, S_4, that will hold facets. A facet is a part (or all) of a face, and itself has straight edges and is planar. The sets of faces of all the resulting polyhedra are subsets of this set of facets; that is, the facets will not have to be further subdivided. Just as there were two types of segments in the 2-D case, there are two types of facets: those that come from a face of only one of the input polyhedra, and those that are common to two overlapping faces. For the former type, each facet will know which polyhedron it came from, and whether it is inside or outside the other polyhedron. For the latter type, each facet will know whether the two faces it is on face in the same or in opposite directions. The facet will be oriented in the same way as the face of P_α.

9. For each cell, test all the faces, F_α, of P_α in it against all the faces, F_β, of P_β to find which pairs intersect:

a) Assume that the faces are not coplanar. (If they are, go to step (k).) Calculate the line of intersection of the planes of F_α and F_β.

b) Establish a 1-D coordinate system on it, and calculate the matrices needed to transform between it and the 2-D systems on F_α and F_β.

c) Calculate the equation of the intersection line in the 2-D systems of F_α and F_β.

d) For each face, determine the parts of that line that fall within the face. This is similar to point-in-polygon testing, and involves intersecting the line against all the edges of the face.

e) Convert the two included ranges of the intersection line back to the line's 1-D system.

f) Intersect the ranges of the two faces to get the common range. This is done by sorting all the endpoints together along the line, and then sequentially passing along them, while keeping track of which ranges we are currently in. This common range has zero or more intervals. If it has zero, then the two faces don't intersect so we can proceed to the next pair of faces.

g) Convert the common range back to the coordinate system of each face. Each interval is a CUT-LINE of the face.

h) Add this cut-line to S_2.

i) Each endpoint of an interval of the common range is on an edge of either F_α or F_β (or both if the edges happen to intersect). Add the appropriate record or records to S_3.

j) Go back and process the next pair of faces.

Figure 2 shows how two faces are intersected to create cut-lines in both of them. P_α with 5 vertices is intersecting P_β with 3. The line of intersection of their planes is shown at the bottom of figure 2. It cuts P_α in two places and P_β in one place. The intersection of these two regions gives two cut-lines, which are shown heavier at the bottom of figure 2. Figure 3 shows those two cut-lines drawn on each of F_α and F_β separately. After other cut-lines are found on F_α and F_β, the faces will be partitioned into facets.

k) If F_α and F_β are coplanar, then use a simple extension of the polygon combination algorithm presented in the previous section to find all the regions into which F_α and F_β partition the plane. This is the polygons F_α-or-F_β, F_α-and-F_β, F_α-and-not-F_β, and F_β-and-not-F_α. Some of them may be null,and some may have multiple disjoint components. If F_α and F_β do not intersect even though they are coplanar, there will be no components. Add each component as a separate facet to S_4.

10. Sort S_2 by face so that we have in one place all the cut-lines on each face, and which other face caused each one.

11. Sort S_3 by face (major key) and edge (minor key) so that we have in one place all the cut-points on each edge of each face.

12. Repeat for each face, F, to determine the facets that partition it:

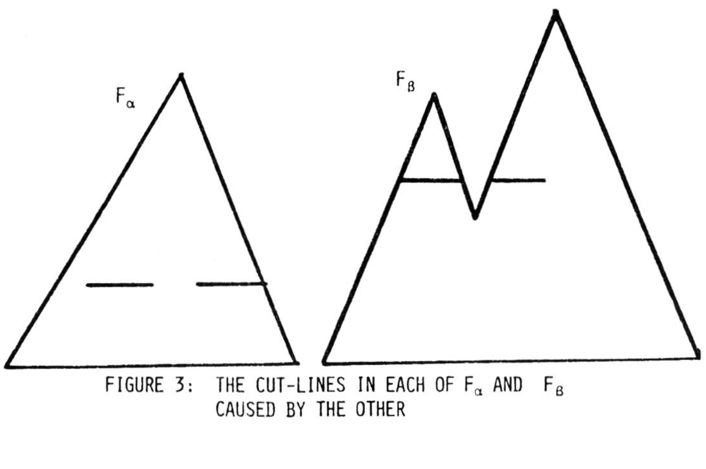

FIGURE 3: THE CUT-LINES IN EACH OF F_α AND F_β CAUSED BY THE OTHER

a) Initialize an empty set, S_5, of segments in the plane of F.

b) For each edge, E, of F, read the cut-points on it from S_3, sort them in order along E, and use them to partition E into segments.

c) Add these segments to S_5.

d) Read cut-lines on F from S_2 and add them to S_5.

e) S_5 is now a set of segments defining a planar graph in the plane of F. Its regions are the facets of F. Determine them; that is, determine the vertices in order around the perimeter of each facet. This can be done in linear time by a traversal algorithm similar to that used for finding the vertices of the intersection polygon in order. Each edge of each facet results from either a part of an edge of the original face, or from a cut-line on the face.

Figure 4 shows the three segments that partition F_α after four faces in all have intersected it.

f) Determine whether each facet that is on only one polyhedron is inside or outside the other polyhedron. If any edge of the facet results from a cut-line, then we can determine this by comparing the face against the face of the other polyhedron that caused the cut-line. Otherwise, we must select a point inside the facet and test it against the other polyhedron using a point inclusion in polyhedron test. If we preprocess each polyhedron by projecting it onto a 2-D grid, then this test can be done in time proportional to the depth complexity of the polyhedron (i.e. 2 for convex polyhedra).

g) For each facet, add an appropriate element to S_4.

13. With S_4, we can determine any of the boolean combinations of P_α and P_β: Use table 1, as in the 2-D case, to select the appropriate facets.

TIMING

Since this algorithm makes successive sequential passes through the data, the execution time is proportional to the number of elements in the various sets being processed. For many input scenes, this will be linear in the number of edges, although some inputs will cause it to take quadratic time. In many of these cases, a hierarchical grid will speed it up again. Nevertheless, since it uses an adaptive grid which has been demonstrated to handle scenes with up to 10,000 objects in linear time, the polyhedron combination algorithm can be expected to execute quite efficiently also.

SUMMARY

The techniques shown here can be extended in several ways. For example, the only conceptual difference with intersecting objects that have curved faces, such as bicubic patches [7], is that two curved faces can intersect even though none of the edges of either face pass

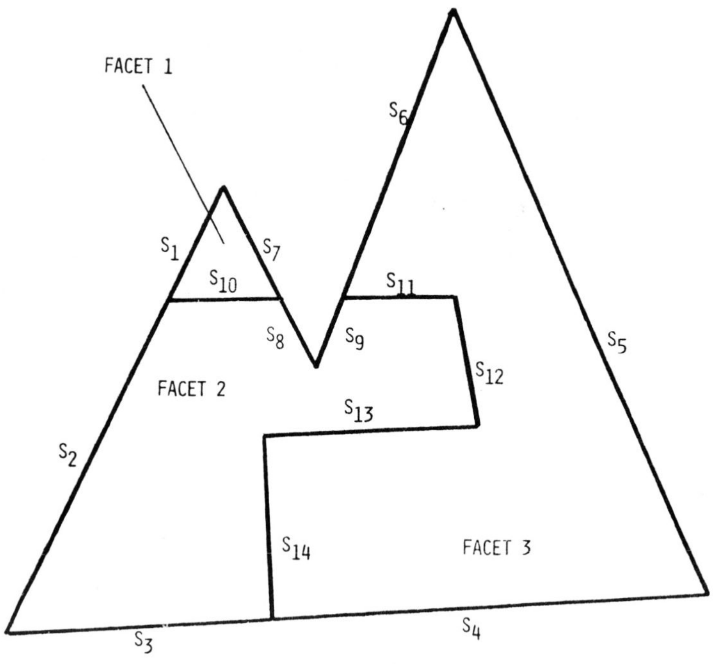

FIGURE 4: F_α AFTER BEING CUT BY 4 FACES, CAUSING 14 SEGMENTS AND 3 FACETS

through the other face. There are of course some non-trivial numerical analysis problems involved.

Another extension is to higher dimensions, since with the configuration space approach [13, 14], an object that can be oriented in different directions in 3-space is equivalent to an object in 6-space whose orientation cannot be changed. Therefore, determining interference requires polytope combination algorithms in higher dimensions.

ACKNOWLEDGEMENT

This material is based upon work supported by the National Science Foundation under grants no. ENG79-08139 and ISP79-20240.

REFERENCES

[1] A. Baer, C. Eastman, and M. Henrion, "Geometric Modelling: A Survey", CAD 11 (5), Sept. 1979, pp. 253-272.

[2] B.G. Baumgart, GEOMED: Geometric Editor, Stanford University STAN-CS-74-414, May 1974. Also available as NTIS AD-780 452.

[3] B.G. Baumgart, Geometric Modelling for Computer Vision, Oct. 1974, Stanford University Artificial Intelligence Memo AIM-249, Oct. 1974.

[4] J.W. Boyse, "Interference Detection Among Solids and Surfaces", Comm. ACM 22 (1), Jan. 1979, pp. 3-9.

[5] I.C. Braid, "The Synthesis of Solids Bounded by Many Faces", Comm. ACM 18 (4), April 1975, pp. 209-216.

[6] C.M. Eastman and C.I. Yessios, An Efficient Algorithm for Finding the Union, Intersection and Differences of Spatial Domains, Carnegie-Mellon University, Dept. of Computer Science, Sept. 1972.

[7] I.D. Faux and M.J. Pratt, Computational Geometry for Design and Manufacture, John Wiley, 1979.

[8] W.R. Franklin, Combinatorics of Hidden Surface Algorithms, Harvard University Center for Research in Computing Technology, TR-12-78, May 1978.

[9] W.R. Franklin, "A Linear Time Exact Hidden Surface Algorithm", Computer Graphics (ACM-SIGGRAPH) 14 (3), July 1980, pp. 117-123.

[10] W.R. Franklin, "An Exact Hidden Sphere Algorithm That Operates in Linear Time", Computer Graphics and Image Processing 15 (4), April 1981, pp. 264-379.

[11] W.R. Franklin, "3-D Geometric Databases Using Hierarchies of Inscribing Boxes", Proceedings of the 7th Canadian Man-Computer Conference, 10-12 June 1981, Waterloo, Ontario, pp. 173-180.

[12] T. Lozano-Pérez and M.A. Wesley, "An Algorithm for Planning Collision-Free Paths Among Polyhedral Obstacles", Comm. ACM 22 (10), Oct. 1979, pp. 560-570.

[13] T. Lozano-Pérez, Spatial Planning: A Configuration Space Approach, MIT Artificial Intelligence Laboratory, A.I. Memo 605, Dec. 1980.

[14] T. Lozano-Pérez, Automatic Planning of Manipulator Transfer Movements, MIT Artificial Intelligence Laboratory, A.I. Memo 606, Dec. 1980.

[15] K. Maruyama, "A Procedure to Determine Intersections Between Polyhedral Objects", Int. J. Comput. Infor. Sc. 1 (3), 1972, pp. 255-266.

[16] D.J.R. Meagher, Octree Encoding: A New Technique for Representation, Manipulation, and Display of Arbitrary 3-D Objects by Computer, Rensselaer Polytechnic Institute, Image Processing Laboratory, Technical Report IPL-TR-80-111, Oct. 1980.

[17] R.E. Parent, "A System for Sculpting 3-D Data", Computer Graphics (ACM) 11 (2), Summer 1977, pp. 138-147.

[18] S.A. Ricketts, and W.R. Winfrey, MODEL: An Interactive Computer Graphics System for Three-Dimensional Geometric Modelling and Computation, Babcock and Wilcox, Power Generation Group, Nuclear Power Generation Division, NPGD-TM-550, May 1980.

[19] R.B. Tilove, "Set Membership Classification: A Unified Approach to Geometric Intersection Problems", IEEE T. Comp. C-29 (10), Oct. 1980, pp. 874-883.

[20] J.A. Turner, An Efficient Algorithm for Doing Set Operations on Two- and Three-Dimensional Spatial Objects, Architectural Research Laboratory, University of Michigan.

[21] H. Voelcker, personal communication.

[22] W.R. Winfrey, IMAGES: A Computer Graphics System for Jet Impingement Analysis, Bobcock and Wilcox, Power Generation Group, Nuclear Power Generation Division, BAW-1581, Sept. 1979.