COMPUTER VISION, GRAPHICS, AND IMAGE PROCESSING 22, 327–338 (1983)

Rays — New Representation for Polygons and Polyhedra[1]

W. RANDOLPH FRANKLIN

Electrical, Computer, and Systems Engineering Department, Rensselaer Polytechnic Institute, Troy, New York 12181

Received April 23, 1982

A new data structure for storing polygons and polyhedra is presented. It can handle general cases with multiple disconnected components and nested holes with islands to an arbitrary depth. The advantage of this method is that, as in constructive solid geometry, it can support Boolean operations such as union and intersection on nested hierarchical objects. However, unlike CSG, properties such as area, perimeter, and moment of inertia can also be determined exactly. They can be determined by making one sequential pass through the object (which can be on secondary storage), calculating a scalar function of each ray, and adding the numbers. It is never necessary to determine either the global connectivity or the edges themselves. For polygons, the data structure is a set of semiinfinite rays where each ray also has an associated value, either + or −. Each edge of the polygon is represented by two rays, although these two rays are nowhere tagged as belonging to the same edge, but are interspersed with all the other rays.

1. INTRODUCTION

In computer graphics and CAD, various methods are used to represent objects such as surface and volume representations, and these are manipulated by classes of operators ranging from Eulerian surface representations [1, 2, 5] to constructive solid geometry (CSG) [3, 4, 5, 6]. Although the different representations for an object contain the same information in the sense that one form can be converted to another, certain operations can be performed more easily in one representation than another. Thus, with CSG, nested intersections and unions can be made formally rigorous, while with the Eulerian form, surface areas and volumes can be calculated quickly and exactly.

In this paper, we present a new method of representing polygons that combines some advantages of both Euclidean and CSG forms, called a *ray* representation. It can handle nested polygons with multiple components, as shown in Fig. 1. The polygons can have curved sides, as shown in Fig. 2; the only restriction on the curves is that "reasonable" operations, such as intersections with each other and integration of the area below them, are possible either analytically or numerically. The representation can be extended to higher dimensions.

This paper is divided into several sections. Section 2 describes the data structure for straight-sided polygons, and gives several examples. After that we see how to convert to a conventional edge representation, although that is only necessary when converting polygons for input to other systems. Section 4 presents algorithms for operations such as finding perimeter length, area, and moment of inertia, and testing whether a point is in the object. Next, we see how to compute intersection, union,

[1]This material is based upon work supported by the National Science Foundation under Grants ENG 79-08139 and ECS 80-21504.

0734-189X/83 $3.00

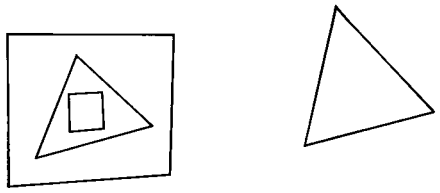

FIG. 1. Polygon with nested holes and disconnected components.

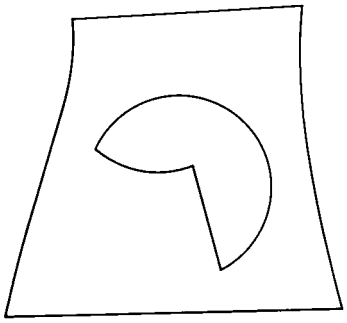

FIG. 2. Polygon with curved edges.

and difference efficiently. In Section 6, the extension to curved sides is explained, and finally an extension to 3-D is given.

2. DATA STRUCTURE

The data structure for the ray representation of a polygon is a set of rays or semiinfinite line segments, one for edge incident on each vertex, i.e.,

$$R = \{R_i\} = \{(P_i, U_i, B_i)\}$$

where $P_i = (P_{i1}, P_{i2})$ is the Cartesian location of a vertex, and $U_i = (U_{i1}, U_{i2})$ is a unit vector, in the direction of an edge incident on P_i. By convention, we traverse the polygon's perimeter in the positive (counterclockwise) direction so that the outside is on the right. U_i follows this convention. B_i is a flag bit, either 1 or 0, that indicates whether U_i is pointing towards or away from the vertex at the other end of the edge.

Since there are two edges incident on each vertex, R has twice as many elements as there are vertices, the P_i include each vertex exactly twice, and each edge is represented as two rays. For example, see Fig. 3, which shows a rectangle. We have the vertices

$$V_1 = (0,0)$$
$$V_2 = (2,0)$$
$$V_3 = (2,1)$$
$$V_4 = (0,1).$$

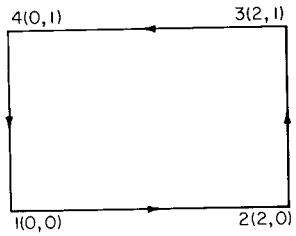

FIG. 3. Sample rectangle.

The two unit vectors on V_1 go to V_2 and from V_4. They are

$$U_{11} = (1,0)$$
$$U_{12} = (0,-1).$$

Note that if we rooted U_{11} and U_{12} at V_1, U_{12} would point away from V_4. Therefore

$$B_{11} = 1$$

but

$$B_{12} = 0.$$

If we continue around the polygon, we determine its complete data structure:

$$\begin{aligned} R = \{ & \\ & ((0,0), \quad (1,0), \quad 1), \\ & ((0,0), \quad (0,-1), \quad 0), \\ & ((2,0), \quad (1,0), \quad 0), \\ & ((2,0), \quad (0,1), \quad 1), \\ & ((2,1), \quad (0,1), \quad 0), \\ & ((2,1), \quad (-1,0), \quad 1), \\ & ((0,1), \quad (-1,0), \quad 0), \\ & ((0,1), \quad (0,-1), \quad 1) \\ & \}. \end{aligned}$$

The R has been listed here in the order corresponding to the edge representation of the polygon, but since it is a set and not an ordered tuple, there is, in fact, no order. This is a key point of the ray representation. Since the rays are not stored in any order, not only are the edges nowhere explicitly stored, but we do not even know (without sorting) the two rays that are incident on a given vertex. Indeed we have a unit vector at V_1 pointing toward V_2, and vice versa at V_2 pointing at V_1. But since these are unit vectors, we do not know the edge, and don't know that V_1 is even connected to V_2.

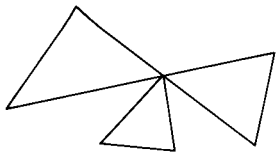

FIG. 4. Degenerate polygon with six edges on the same vertex.

The ray representation allows for degenerate cases such as Fig. 4, where the central vertex has six edges incident on it. Here there are six rays with the same endpoint. Cases such as shown in Fig. 7, where two edges are collinear, are also handled without the algorithms being aware of them.

3. CONVERSION TO AN EDGE REPRESENTATION

The ray representation is a closed system in that the intersection, union, and difference operators return polygons in the same representation. Nevertheless, if it is to be useful, it must be possible to convert polygons from the ray representation to a more conventional edge representation. The existence of this conversion also shows the sufficiency of the ray representation, i.e., that it contains enough information to fully represent the object.

The key point is to consider the rays as order-2 elements of a group, in the sense that if two rays are coincident, then they cancel. If two rays with the same slope coincide for some finite distance, then they cancel for that distance only. Thus two semiinfinite rays can be combined to form a finite line segment, as shown in Fig. 5. The process generalizes, so that an even number of coincident rays completely cancel, while an odd number cancel all but one. See Fig. 6, where of four rays at different points, two, three, or four coincide to leave two finite lines.

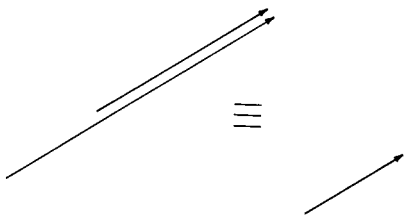

FIG. 5. How two semiinfinite rays form a finite line segment.

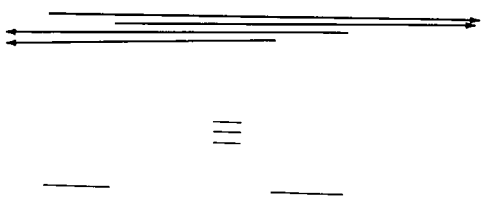

FIG. 6. How four rays form two collinear edges.

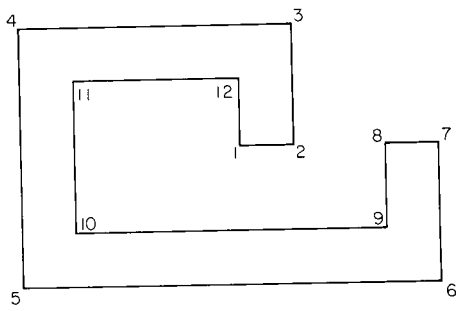

FIG. 7. Polygon with two collinear edges (V_1V_2 and V_7V_8).

It is now clear how to convert from a ray to an edge representation. The process is, briefly:

(1) partition R into classes of rays which fall on the same infinite line. This is equivalent to partitioning a set of points with homogeneous coordinates in 2-D, since the lines can be converted to points by a projective inversion. Thus, this can be done by normalizing and sorting.

(2) in each class, sort the endpoints of the rays along the line. The sorted list, together with the direction each ray leaves its endpoint, will produce the finite line segments.

This process will handle cases of two or more collinear edges. For example, see Fig. 7 where E_{12} and E_{78} are collinear. Here the rays from V_1, V_2, V_7, and V_8 form the pattern shown in Fig. 6, which produces two line segments, which is the goal.

4. SIMPLE OPERATIONS WITH THE RAY DATA STRUCTURE

Unless we can efficiently perform the common operations with this ray data structure, it is useless. This section shows how to test if a point is in a polygon, find area and moment of inertia, and calculate perimeter length, all without determining the edges.

These operations all have the same form:

(1) Calculate a function of each ray in the set R. Each ray is processed independently.

(2) Combine the returned values to give the desired result.

Point-in-Polygon Testing

This algorithm is a derivative of the edge data structure method, where we run a ray up from the point in question, X, and count how many of the edges of the polygon P it crosses. Iff it is an odd number, then X is inside P. Here, we also run a ray up from X, but in the first step we count how many rays from R it crosses. See Fig. 8, where the ray up from X crosses R_1, but not R_2 or R_3. The X is inside P if and only if there are an odd number of crossings.

If we are to test many points against the same polygon, then we can preprocess the polygon with an adaptive grid [3]. This will cut the time required to test each point.

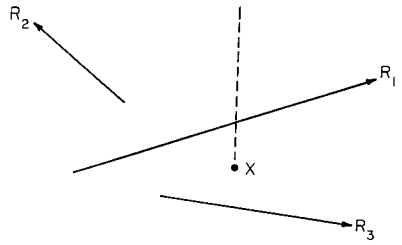

FIG. 8. Part of point-in-polygon testing: X is below ray R_1 only.

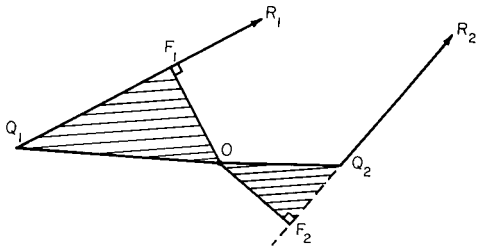

FIG. 9. Dropping perpendiculars from 0 to (extended) rays to calculate areas.

Polygon Area

In the usual method, we run lines from the origin O to each vertex of P to partition it into triangles whose area we sum. Here, we drop a perpendicular from O to each R_i. Let its foot be at F_i, and let the end of R_i be Q_i (see Fig. 9). Find the signed area of triangle OQ_iF_i. Then if B_i (the flag bit associated with R_i) is 0, negate the area. The sum of the resulting values will be the area of the polygon. Now, the foot of the perpendicular may fall on the ray, as for R_1 in Fig. 9, or on the ray extended, as for R_2. Here OQ_2F_2 is negative since those points are clockwise.

Figure 10 shows a triangle composed of rays R_1 through R_6. The $B_1 = B_3 = B_5 = 1$ while $B_2 = B_4 = B_6 = 0$. Of the six feet of the perpendiculars, F_1, F_2, and F_6 fall on their extended rays, while the others fall on the rays proper. Of the six triangles, only

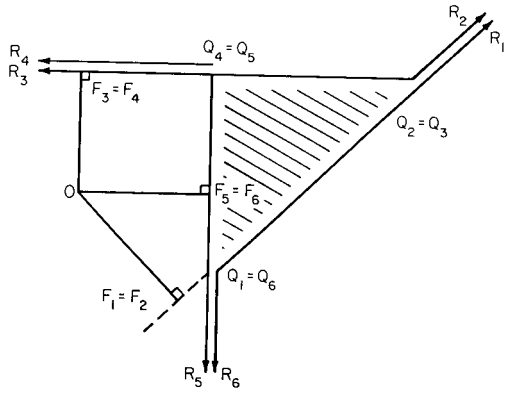

FIG. 10. Determining the area of a triangle.

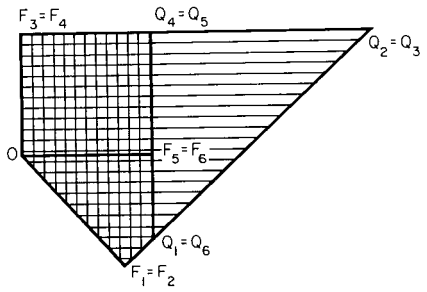

FIG. 11. Subtracting negative area (vertical hatches) from positive area (horizontal).

OQ_3F_3, OQ_4F_4, and OQ_6F_6 are positive. After the B_i are applied, only OQ_2F_2 and OQ_3F_3 are positive. The resulting effect is shown in Fig. 11, where the positive area has horizontal crosshatches and the negative area vertical crosshatches. The net area is the triangle.

Polygon Moment of Inertia

This is determined in the same manner as the area, except that instead of adding up signed areas of triangles, we add up signed moments of inertia.

Perimeter Length

To calculate the total length of all the edges without finding the edges, drop a perpendicular to each ray, as shown in Fig. 9. Calculate the length of Q_iF_i, making its sign positive if F_i is on the ray proper, and negative if F_i is on the ray extended. Thus in Fig. 9, Q_1F_1 is positive, but Q_2F_2 is negative. The perimeter length is half the sum of these quantities.

5. SET OPERATIONS

The ray data structure also admits explicit set operations on polygons in its format.

Since the output is in the same form as the input, i.e., rays, we have all the vertices of the resulting polygon explicitly, and if we apply the algorithm given in an earlier section, we can obtain the explicit edges. Thus we also have an efficient set combination algorithm for the edge representation.

To see how the set operations work, consider, without loss of generality, determining the union of polygons A and B to form polygon C. The vertices of C are of two origins:

(i) some of the vertices of A and B themselves, and

(ii) all of the intersections of edges of A with edges of B.

We determine which vertices of A and B to include by testing the vertices of each polygon against the other with the point-in-polygon test described above. Since many points will be tested against the same polygon, use the preprocessing version of the algorithm. For union, use those vertices of A that are outside of B, and those of B outside of A. More precisely, include in C all rays of A whose endpoint is in B, and so on. For some of the set operations, such as $A - B$, the ray's direction and flag bit must be negated. For example, see Fig. 12, where when point X is a vertex of

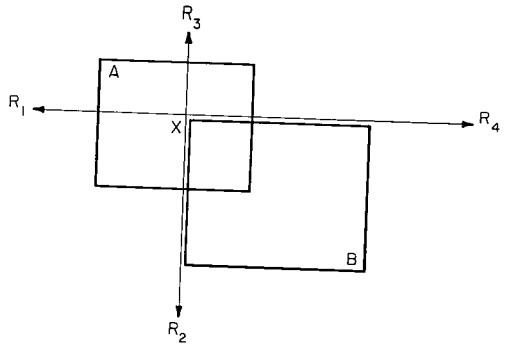

FIG. 12. Rays on vertex X when combining polygons A and B: R_1 and R_2 for $A \cap B$, but R_3 and R_4 for $A - B$.

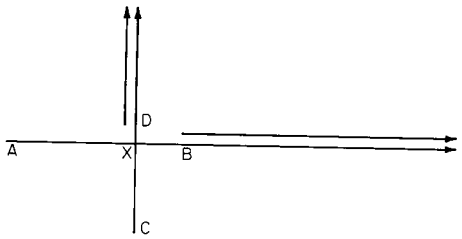

FIG. 13. Intersection of two line segments defined by rays.

$A \cap B$, its rays are R_1 and R_2. When it is a vertex of $A - B$, its rays are R_3 and R_4. As for the other set operations, for $A \cap B$, include those rays of A and rays of B whose endpoints are inside the other polygon. For $A - B$, include rays of A outside B, but rays of B inside A, and reverse the latter. For $B - A$, include rays of A inside B and reverse them, and rays of B outside A. For the exclusive-or of A and B, calculate $(A - B) \cup (B - A)$.

The second class of rays of C is rooted at the intersections of edges of A with B. The problem is to determine these intersections without actually determining the edges themselves. We will present a slow but simple way and then indicate how it can be made more efficient. The key observation is that if we consider a point P on at least one ray, then if and only if it is on an edge, P is on an odd number of rays. This relates back to Figs. 5 and 6. Thus, if two bundles of coincident rays intersect at P, then the corresponding edges proper will intersect at P if and only if both bundles have an odd number of rays in them. Thus, in Fig. 13, edges AB and CD intersect at X since one single ray intersects one ray there. Now, we do not in fact have bundles of coincident rays, but only two sets of individual rays, so the algorithm is

(i) Find all intersections of rays of A with rays of B, and call the points P_i. Each P_i has pointers to the rays whose intersection caused it.

(ii) Sort the P_i to collect sets of P_i with the same coordinates.

(iii) Discard those sets of P_i with an even number of points in them. Each remaining set gives one vertex of C, so the only remaining problem is to find the rays associated with the vertices.

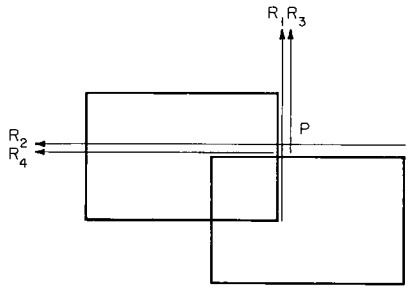

FIG. 14. Intersection of two polygon edges.

(iv) The rays of C that are rooted at P_i are parallel to the rays that intersected to form P_i. See Fig. 14, where R_1 of polygon A crossed R_2 of polygon B at P. The rays of C rooted at P are R_3 and R_4, which are parallel to R_1 and R_2, respectively. For the cases of $A - B$ and $B - A$, as before, some of the rays must be reversed.

This algorithm would be fine if it did not require intersecting every ray of A with every ray of B, which has quadratic time complexity. The key to improving it is the observation that we do not want every ray intersection, but only those duplicated an odd number of times. Further, so long as two rays are coincident, they do not represent an edge and so can be ignored. The algorithm is a recursive one that is related to Warnock's hidden surface algorithm:

Divide the screen into a grid of, say, 10 by 10 squares. For each ray R_i, determine which squares it passes through. Sort this data by square so that we have all the rays passing through each square. Within each square, further sort those rays that are not rooted in it, i.e., those that pass completely through it. Eliminate any duplicates, and the remainder are the candidates for intersections within the square. If there are too many of them, recursively subdivide the square and repeat. Unlike the adaptive grid algorithms of [3], one level of this grid should not become finer as the number of rays increases. Since the rays do not get shorter as they become more numerous, we must use recursion instead, which carries some overhead.

If the resulting polygon is ill formed in a set-theoretic CSG sense, as shown in Fig. 15, this can be corrected by noting that two opposite rays are rooted at the same vertex and deleting both of them. The remaining rays will form a correct triangle. Problems such as shown in Fig. 16 can be corrected with the same techniques as used in the conversion to the edge representation.

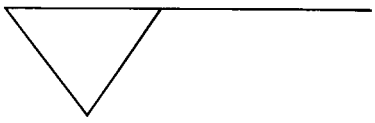

FIG. 15. Ill-formed polygon with spike.

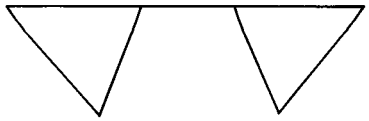

FIG. 16. Ill-formed polygon with section with no interior.

6. POLYGONS WITH CURVED EDGES

The ray data structure and associated algorithms have a natural extension to polygons with curved sides as shown in Fig. 2. The extension is to consider a ray to be a semiinfinite half of an infinite curved line. For straight rays, the area algorithm requires dropping a perpendicular from the origin to the ray, possibly extended. For a curved ray, there might be no unique perpendicular. This is immaterial since the only required characteristic of the foot of the perpendicular is that it be a point on the infinite extension of the ray, uniquely determined by that infinite curved line and independent of the endpoint of the ray. See, for example, Fig. 17, where the two rays are parts of the parabola $Y = X^2 - 1$, and the unique point for both is $(0, -1)$. This means that the area of the two-sided polygon, with sides consisting of the straight line and the parabola between $A(1, 0)$ and $B(2, 3)$ in Fig. 18, is determined by adding the horizontally crosshatched areas and subtracting the vertically crosshatched ones.

Minor modifications are needed for curves such as circles that close in on themselves and are of finite total length. They involve cutting the curve at some fixed point, and then watching signs carefully. Curves that loop through themselves can be handled provided they do not do so in the region which is used as an edge.

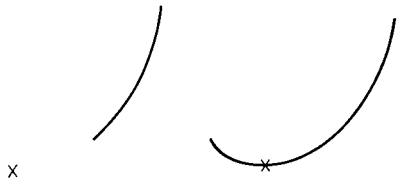

FIG. 17. Picking a unique point (0–1) on a section of the parabola $Y = X^2 - 1$ independent of the section's endpoints.

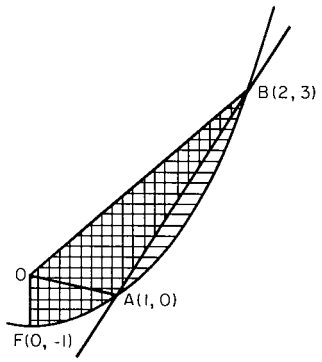

FIG. 18. Finding area of a curved polygon.

It can be seen that the only limitation on the curves is that the operations required by the various algorithms, such as determining intersections of the curves with each other and with straight lines, and integrating areas of regions bounded by the curves, are possible.

7. THE RAY DATA STRUCTURE IN 3-D

There are extensions into three dimensions of the 2-D data structures. We will now see one example designed to determine mass properties such as volume and surface area. As before, we do not know either the global topology or even the edges of the polyhedron A, but only the location and a local epsilon neighborhood of each vertex. More precisely, the data structure for A is a set S, with one element for each face incident on each vertex. Thus a cube with eight vertices, each with three faces incident on it, would have 24 elements in S. Each S_i is an ordered tuple of

(i) the coordinates of the vertex V, and

(ii) the unit direction vectors P and Q of the two edges incident on V that border the particular face incident on V. Each vector points along its edge from V towards the other end of that edge. The two vectors are ordered so that their cross product, which is normal to the face, points towards the outside of A. (This requirement replaces the need for flag bits and pointing the edges in a positive direction.)

Thus, instead of having the elementary data structure be a ray which is half of an infinite line, in 3-D it is a *wedge* or a slice of an infinite plane. For example, the three elements of S for the vertex $(-1, -1, -1)$ of the cube with vertices at $(\pm 1, \pm 1, \pm 1)$ are

$$\begin{aligned}&((-1,-1,-1),\quad (0,1,0),\quad (1,0,0))\\&((-1,-1,-1),\quad (0,0,1),\quad (0,1,0))\\&((-1,-1,-1),\quad (1,0,0),\quad (0,0,1)).\end{aligned}$$

The formula for volume is the sum of the following expression for each ordered triple (V, P, Q):

$$\frac{1}{6}\frac{V\cdot P\times Q}{\left(1-(P\cdot Q)^2\right)}\left(2V\cdot P\,V\cdot Q-P\cdot Q\left((V\cdot P)^2+(V\cdot Q)^2\right)\right).$$

8. SUMMARY

It is possible both to perform set operations and to determine mass properties of polyhedra and polygons without knowing the explicit edges. The set of the locations and local neighborhoods of the vertices is sufficient.

REFERENCES

1. A. Baer, C. Eastman, and M. Henrion, Geometric modelling: A survey, *Comput. Aided Des.* **11**, 1979, 253–272.
2. C. M. Eastman and C. I. Yessios, An Efficient Algorithm for Finding the Union, Intersection, and Differences of Spatial Domains, Dept. Computer Science, Carnegie–Mellon Univ., Sept. 1972.

3. W. R. Franklin, An exact hidden sphere algorithm that operates in linear time, *Computer Graphics and Image Processing* **15**, 1981, 364–379.
4. Y. T. Lee and A. A. G. Requicha, Algorithms for computing the volume and other integral properties of solid objects, 1: Known methods and open issues, 2: A family of algorithms based on representation conversion and cellular approximation, *Assoc. Comput. Mach.* **25**, 1982, 635–650.
5. A. A. G. Requicha, Representations for rigid solids: Theory, methods, and systems, *Assoc. Comput. Mach. Computing Surveys* **12**, 1980, 437–464.
6. R. B. Tilove, Set membership classification: A unified approach to geometric intersection problems, *IEEE Trans. Comput.* **C-29**, 1980, 874–883.