Constructive Solid Geometry for Polyhedral Objects

David H. Laidlaw
W. Benjamin Trumbore
Department of Computer Science

John F. Hughes
Department of Mathematics

Brown University
Providence, RI 02912

Abstract

Constructive Solid Geometry (CSG) is a powerful way of describing solid objects for computer graphics and modeling. The surfaces of any primitive object (such as a cube, sphere or cylinder) can be approximated by polygons. Being abile to find the union, intersection or difference of these objects allows more interesting and complicated polygonal objects to be created. The algorithm presented here performs these set operations on objects constructed from convex polygons. These objects must bound a finite volume, but need not be convex. An object that results from one of these operations also contains only convex polygons, and bounds a finite volume; thus, it can be used in later combinations, allowing the generation of quite complicated objects. Our algorithm is robust and is presented in enough detail to be implemented.

1. Introduction

The algorithm presented finds the polygonal boundaries of the union, intersection, or difference of two polyhedral solids. Our presentation differs from others in the literature [REQ85, TUR84] in several ways. We differ from [REQ85] by presenting an exhaustive analysis of all types of intersections, rather than discussing only generic cases, and by efficiently addressing the difficulties which arise when dealing with coplanar polygons. We differ from [TUR84] by restricting object boundaries to be convex polygons, by subdividing polygons without introducing non-essential vertices, and by allowing objects that are not manifolds. We also sketch an argument showing that the algorithm terminates.

"Constructive Solid Geometry" [REQ80a] operations are defined on surfaces that bound a volume of finite extent. These surfaces may be constructed from several pieces, with very weak constraints on how these pieces touch one another. As a result these objects can be more general than the standard polyhedral surfaces found in mathematics. For example, a single object can consist of two cubes joined along an edge and a third cube that is not connected to the first two (Figure 3.1a). The shared edge touches four faces and cannot be an edge of a polyhedral surface, but the object is still valid. Many other CSG systems will not allow this type of object. The union, intersection, and difference operations on the solids bounded by each object give rise to corresponding operations on the boundaries. We identify these boundaries with the same names as the objects. The *union* of two objects is defined as the boundary of the volume contained in either or both of the objects, while the *intersection* is defined as the boundary of the volume that they have in common. The *difference* of two objects is defined as the boundary of the volume contained in the first object but outside of the second. Using primitive objects like cubes, spheres, and cylinders, this algorithm can construct more complicated objects that in turn can be used to construct even more complex objects.

Our algorithm is part of a set of rendering and solid modeling programs developed at Brown. SCEFO, a language developed at Brown to describe animations and static scenes, describes CSG combinations of objects using a binary tree, with a primitive object at each leaf and a set operation at each internal node [STR84]. From this description renderers create images of the objects. Figure 1.1 (after section 10) shows the CSG construction of a spoon using primitive objects. The images are rendered polygonally, using this algorithm.

A ray-tracer and a polygonal renderer can render an object described as a binary tree of CSG combinations of primitives. Ray-tracers intersect a ray with each primitive object and perform the CSG operation along the ray [ROT82], while polygonal renderers use our algorithm to produce polygonal versions of CSG combinations and then render them as polygons. Using renderers that understand SCEFO, we can quickly render polygonal versions of objects to preview an image and later ray-trace them to produce a more polished image (see Figure 10.1).

The basic ideas of the algorithm have been suggested in other sources [REQ80a, REQ80b], but were not described in enough detail to be implemented. We will present the algorithm so that it may not only be implemented but also verified. As long as the polygons of an object bound a volume of finite extent and each polygon is convex, the algorithm will work and will produce a new object that satisfies the same restrictions.

The paper is organized as follows: first, we present an overview of the algorithm. Then the data structure used by the algorithm is described, followed by a detailed description of the algorithm. A discussion of results, problems and extensions concludes the paper.

2. Overview

Our algorithm operates on two objects at a time. The routines can be called successively using the results of earlier operations to create more complicated objects. Each object is represented as a collection of polygons and vertices; each spatially distinct vertex is represented exactly once, and each polygon contains a list of references to vertices. Each polygon also contains a normal that points outwards from the object. Each vertex contains a list of references to other vertices connected to it by an edge.

This research was sponsored in part by the Digital Equipment Corporation and the Office of Naval Research under contract N00014–78–C–0396, Andries van Dam, principal investigator.

The algorithm first subdivides all polygons in each of the objects so that no two polygons intersect. Two non-coplanar polygons *intersect* if a vertex of one lies in the interior of a face or edge of the other, or if an edge of one crosses an edge or face of the other. Polygons that share a vertex or an edge, or that are coplanar, do not intersect. The logic for not intersecting coplanar polygons is described in the last paragraph of section 4.

Once the polygons of both objects have been subdivided, the polygons of each object are classified with respect to the surface of the other object. A polygon in one object may lie inside, outside, or on the boundary of the other object. Vertex adjacency information is used here so that the same classification can be assigned to polygons that are adjacent and do not cross the surface of the other object. This avoids comparing all polygons in one object with all polygons in the other object. The boundary of the combination of the objects will be a subset of all the polygons in both objects. Each polygon's classification is determined by casting a ray from the polygon through the other object and testing the intersection point and surface normal of the nearest intersecting polygon in the other object. The algorithm uses the classification of each polygon to retain or delete it according to the set operation being performed.

As the algorithm proceeds it modifies the objects until, finally, the set of all polygons in both modified objects forms the resulting object.

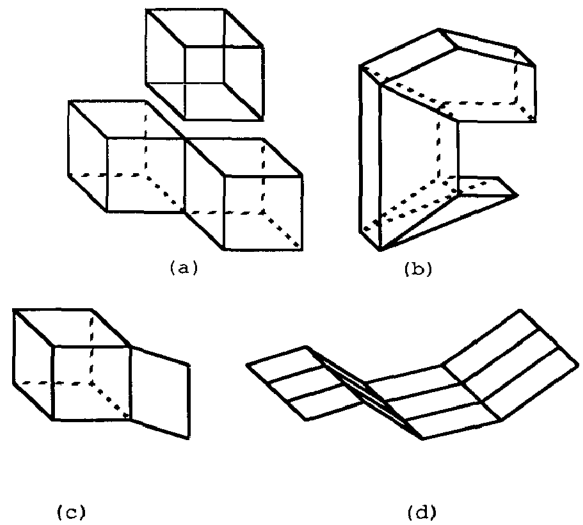


Fig. 3.1: Objects (a) and (b) are valid; objects (c) and (d) are not

3. Object Data Structure

While these CSG routines are flexible enough to operate on many different types of objects, the objects must satisfy certain restrictions. Because we are performing operations on the boundaries of volumes, each object must be constructed of polygons that form the topological boundary of the closure of an open set of finite extent in R^3 [REQ80]. More simply, an object must be the surface of a volume and must not have dangling faces or edges. This restriction enables us to efficiently distinguish the interior of the object from the exterior. Planes and other surfaces that do not enclose a volume are not valid objects, but can often be modified to bound some volume. Figure 3.1 shows examples of two valid and two invalid objects.

Each polygon in an object must satisfy five restrictions. (1) It must be planar and convex. (2) No three vertices in the polygon may be collinear. (3) It may not contain the same vertex twice. (4) The vertices must be ordered clockwise when viewed from outside the object, so that cross-products using the directed edges of the polygon may be used to determine the interior of the object. (5) No polygon may intersect any other polygon in the object. A simple verification program can check that the order of the vertices in each polygon agrees with the direction of the normal, and verify that all polygons are convex and planar. Combining two valid objects always produces a new valid object which therefore does not need to be verified.

The vertex data structure contains the spatial location of the vertex as well as a list of pointers to adjacent vertices and a "status" field. Initially, the status field is set to UNKNOWN, but as the algorithm proceeds, this field changes to indicate whether the vertex is INSIDE, OUTSIDE, or on the BOUNDARY of the other object. The list of adjacencies is used for traversing the edges of the object to find connected regions of vertices with identical status. This adjacency informations calculated after the objects have been intersected with each other.

The polygon structure includes a list of pointers to the vertices of the polygon, the plane equation, and the extent of the polygon. The plane equation is used as the polygon normal, and is also used when intersecting polygons. The extent is used to determine quickly if two polygons do not intersect.

The object structure consists of the extent of the object, an array of vertices, and an array of polygons. Again, the extent is used to determine quickly when a polygon does not intersect the object. The data structures for objects, vertices, and polygons are shown in Figure 3.2.

Object Structure
array of vertices
array of polygons
object extent (minimum and maximum x,y,z)

Vertex Structure
spatial location (x, y, z)
array of pointers to adjacent vertices
status (inside, outside, boundary, or unknown)

Polygon Structure
array of pointers to vertices
polygon extent (minimum and maximum x,y,z)
polygon plane equation (x,y,z,d)

Fig 3.2: Data structures

4. Intersecting the Objects

The first step in the algorithm is splitting both objects so that the polygons in each do not intersect. In this discussion, we will refer to the object which is to be split as objectA and to a polygon in that object as polygonA. Similarly, polygonB is a polygon in objectB, the other object.

The first part of Figure 4.1 explains how pairs of objects are subdivided. When polygonA is split, new edges will be introduced into objectA, and a face that is split will become two or more new faces. The new edges in objectA may intersect the interiors of faces of objectB, possibly requiring further subdivision of polygons in objectB.

When the splitting routine is initially called, the first object is objectA and the second is objectB. After this initial splitting, no face of the second object intersects the interior of any face of the first object. So on the second pass, the new faces that are generated by splitting faces of the second object create no further intersections with interiors of faces in the first object; only new edge intersections are created.

Consequently on the third pass (the first object is once again objectA), polygons in objectA will only be changed by splitting edges at points where these edges intersect new edges of the second object. This will, as before, introduce no new edges that intersect faces in objectB. It also will not introduce any new edge-edge intersections, since the only new edges that are added come in the interiors of polygonAs, and these never intersect the faces or edges of polygonBs. Thus there is no need to make a fourth pass; the algorithm is finished.

Subdividing Objects
split the first object so that it doesn't intersect the second object
split the second object so that it doesn't intersect the first object
split the first object again, resolving newly introduced intersections

Splitting ObjectA by ObjectB
if extent of objectA overlaps extent of objectB
  for each polygonA in objectA
    if the extent of polygonA overlaps the extent of objectB
      for each polygonB in objectB
        if the extents of polygonA and polygonB overlap
          analyze them as in "5. Do Two Polygons Intersect?"
          if they are not COPLANAR and do INTERSECT
            subdivide polygonA as in
              "6. Subdividing Non-Coplanar Polygons"
          else if they do NOT_INTERSECT
            or if they are COPLANAR
            (do nothing)

Fig. 4.1: Splitting objects

The second part of Figure 4.1 explains how all the polygons in one object are split so that they do not intersect a second object. For each pair of polygons with overlapping extents, the routine described in the section "Do Two Polygons Intersect?" determines whether the polygons are COPLANAR, INTERSECT in a line (or possibly a point), or do NOT_INTERSECT. Pairs of polygons that INTERSECT are subdivided as described in the section "Subdividing Polygons." New polygons are added to the end of the the list of polygons in objectA, and are checked against objectB after all original polygons in objectA have been checked. Those that are COPLANAR or do NOT_INTERSECT are not subdivided.

Although COPLANAR pairs of polygons are not subdivided, after the first two subdivisions all groups of adjacent coplanar polygons in one object will have corresponding groups of coplanar polygons in the other object. While the polygons in these groups may not be identical, the regions they cover will be the same. Each edge of a polygon is shared by at least one other polygon. If an edge of polygonB crosses polygonA, then there must be another polygonB which also has that edge. If this polygonB is not coplanar with polygonA, then polygonA will be subdivided when compared to this second polygonB. If the adjacent polygonB is coplanar with polygonA, it either extends beyond polygonA (and will eventually be subdivided by some polygonA), or is contained within polygonA and, again, will not be used to subdivide polygonA.

5. Do Two Polygons Intersect?

This section describes how to determine whether two polygons are coplanar, intersect in a line (or possibly a point), or do not intersect. *The first step in determining whether the two polygons* intersect is finding the signed distance from each of the vertices in polygonA to the plane of polygonB. The distance is positive if the normal vector points from the plane of the polygon towards the point. If these distances are all zero, then the polygons are coplanar. If they are all positive or all negative, then polygonA lies entirely to one side of the plane of polygonB, and thus the two polygons do not intersect; otherwise they may intersect, and the signed distance from each vertex in polygonB to the plane of polygonA is computed. Again, if the distances are all positive or all negative, then polygonB lies entirely to one side of polygonA, and the two polygons do not intersect. Coplanar polygons would have been discovered by the first test, so the distances cannot all be zero.

If the preceding tests are inconclusive, then we calculate the *line of intersection of the two planes. The line of intersection L is* determined by a point P and a direction D. Some segment of this line is interior to or on the perimeter of polygonA, and some segment is interior to or on the perimeter of polygonB. If these two segments overlap, then the polygons intersect. If the segments do not overlap, then the polygons do not intersect.

Data structures for each of the two segments store information that is used to subdivide polygonA and polygonB, if they intersect. This information includes the distance from P to the starting and ending points of the segment, as well as descriptors that record whether each point of the segment corresponds to a vertex of the polygon it spans, a point on its edge, or a point on its face. *Because all polygons are convex and contain no collinear vertices,* it follows that the intersection is a single line segment and that three descriptors are sufficient to describe the entire segment: one *for the starting point, a second for the interior of the segment,* and a third for the ending point. A segment that starts at a vertex, crosses a face, and ends at an edge can be represented by the *mnemonic vertex-face-edge. Similarly, any type of segment can be* represented by a three-word mnemonic.

Only the distances from P to the start and end of the segment are necessary to determine if the polygons intersect. If the segments overlap, then the additional information is used later to subdivide the polygons.

The intersection of L with either polygon must both start and end at a vertex or an edge. In addition to the beginning and ending points on L and the three type descriptors for the segment, the segment structure stores the indices of the vertices preceding the endpoints of the segment (for example, B and E in Figure 6.3a-q). The segment data structure is shown in Figure 5.1.

distance of start of segment from P
distance of end of segment from P
descriptors for starting, middle, and ending points
index of polygon vertex near start point
index of polygon vertex near end point

Fig. 5.1: Segment data structure

The remainder of this section and the following sections discuss operations on polygonA; these same operations are also performed on polygonB.

The segment structure is filled in as follows. There are six different ways in which L can intersect polygonA. They are characterized by the types of the starting, middle, and ending points of the intersection segment. Because the polygons are convex, the segment starts at a vertex or edge, continues through a vertex, an edge, or the face, and ends at a vertex or edge. Of the twelve combinations, six are not possible. Vertex-edge-edge, edge-edge-vertex, and edge-edge-edge are impossible because any segment that contains edge points in the middle must begin and end at a vertex; if a segment contains some points on an edge, it must contain all points on the edge, including both endpoints. Similarly, edge-vertex-edge, vertex-vertex-edge and edge-vertex-vertex are impossible. Figure 5.2 gives examples of the six possibilities.

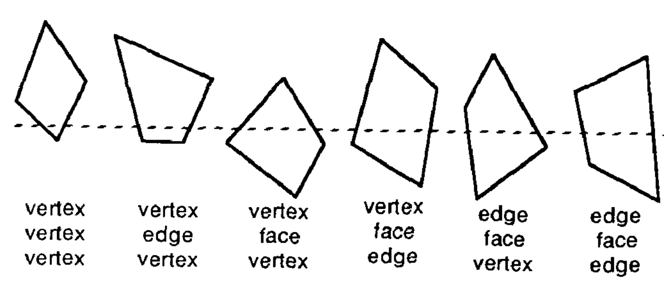


Fig. 5.2: Intersection possibilities of a polygon and a line in a plane

The distances of all the vertices of polygonA from the plane of polygonB were calculated for an earlier test; they are now used to find where L crosses polygonA, since the distance from each vertex to the plane of polygonB is proportional to the distance from the vertex to L. Vertices with distance zero lie on L, while adjacent vertices with distances that differ in sign lie on opposite sides of L and thus are endpoints of an edge that crosses L. For a vertex intersection the index of the vertex is saved in the segment structure and the endpoint type is set to VERTEX (B and E in Figure 6.3b). For an edge intersection, the ratio of the calculated vertex distances is used to find the intersection point, and the intersection point is used to find the distance between the intersection point and P along L. The index of the first vertex of the edge is saved and the endpoint type is set to EDGE (B and E in Figure 6.3k).

The midpoint descriptor is determined from the endpoints of the segment. It is set to EDGE if the endpoints of the segment are adjacent vertices in polygonA, and to VERTEX if the endpoints are the same vertex. Otherwise, the middle points must lie in the FACE of the polygon.

6. Subdividing Non-coplanar Polygons

Given two polygons, polygonA and polygonB, that intersect and are not coplanar, we must subdivide them so that the resulting smaller polygons do not intersect and are still legal polygons. We are also given two segment structures, one representing the intersection of polygonA with L, the other representing the intersection of polygonB with L.

To split polygonA so that none of the resulting smaller polygons intersect polygonB, we need to find the intersection of segmentA and segmentB and determine the type of that intersection segment with respect to polygonA. If either end of segmentA is changed, then the type of that end becomes the same type as the middle points (Figure 6.1).

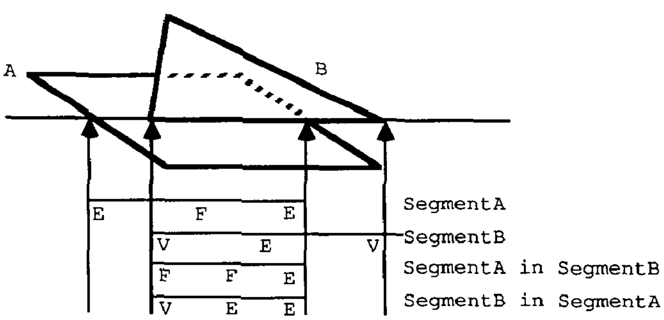


Fig. 6.1: Intersecting two segments

To subdivide polygonA so that the new polygons do not intersect polygonB, the intersection segment must become an edge in the decomposition. The splitting of polygonA is dependent on how the intersection segment cuts across it. Since the starting point can be a vertex, an edge, or a face, as can the midpoints and the endpoint, there are at most 3×3×3 = 27 different kinds of intersection segments. Thirteen of these segment types are impossible, because they have middle point types that are of lower dimension than one of the end types, and in convex polygons that is not possible. Of the remaining fourteen types, four are symmetric to other types with their endpoints swapped. We then need discuss only ten.

In the list of 27 segment types in Figure 6.2, the thirteen impossible types are marked with an "X," and the four symmetric cases are marked with an "S." The remaining ten are numbered to correspond with the discussion that follows. A description of the geometry of the intersection for each of these ten types follows, as does a discussion of the method of splitting polygonA into smaller polygons for each type.

(1)	vertex-vertex-vertex	(X)	edge-edge-face
(X)	vertex-vertex-edge	(S)	edge-face-vertex
(X)	vertex-vertex-face	(8)	edge-face-edge
(2)	vertex-edge-vertex	(9)	edge-face-face
(3)	vertex-edge-edge	(X)	face-vertex-vertex
(X)	vertex-edge-face	(X)	face-vertex-edge
(4)	vertex-face-vertex	(X)	face-vertex-face
(5)	vertex-face-edge	(X)	face-edge-vertex
(6)	vertex-face-face	(X)	face-edge-edge
(X)	edge-vertex-vertex	(X)	face-edge-face
(X)	edge-vertex-edge	(S)	face-face-vertex
(X)	edge-vertex-face	(S)	face-face-edge
(S)	edge-edge-vertex	(10)	face-face-face
(7)	edge-edge-edge		

Fig. 6.2: Identification of valid segment types

The diagrams for each type of segment (Figure 6.3) illustrate the intersection segment and how a polygon with that type of intersection is subdivided. The vertex in the segment structure associated with the beginning of the intersection segment is marked with a "B" and the vertex of the end is marked with an "E." Vertices that are added so that the polygon can be split are marked "M" and "N." The vertices are ordered clockwise in the diagrams.

Note that all subdivisions produce only legal new polygons; no collinear vertices or non-convex polygons are introduced. All vertices that are added must lie on the boundary of objectB, and are thus marked as boundary vertices. These boundary vertices play an important role in selecting polygons for the resultant object (section 8).

(1) Vertex-vertex-vertex – The polygon is intersected at a single vertex and does not need to be subdivided. The vertex is marked as a boundary vertex (Figure 6.3a).

(2) Vertex-edge-vertex – The polygon is intersected along an entire edge and does not need to be subdivided. Both vertices are marked as boundary vertices (Figure 6.3b).

(3) Vertex-edge-edge – The segment intersects the polygon along part of an edge, starting at a vertex and ending in the interior of the edge. The vertex is marked as a boundary vertex. A new vertex is added in the interior of the edge and the polygon is subdivided so that it forms two new polygons (Figures 6.3c and 6.3d).

(4) Vertex-face-vertex – The segment cuts across the polygon starting at a vertex and ending at a vertex. The polygon is cut into two polygons along the line between the two vertices and both vertices are marked as boundary vertices (Figure 6.3e).

(5) Vertex-face-edge – The segment cuts the polygon starting at a vertex, crossing a face, and ending at an edge. The vertex is marked as a boundary vertex. A new vertex is added along the edge and the polygon is divided into two polygons (Figure 6.3f).

(6) Vertex-face-face – The segment crosses part of the polygon, starting at a vertex and ending in the interior of the face. The vertex is marked as a boundary vertex. A new vertex is added in the face. If the segment continued, it would either pass through one of the vertices on the other side of the polygon or miss all of them. If the extended segment passes through a vertex, the polygon is divided into four new polygons to avoid introducing collinear edges or non-convex polygons in the decomposition (Figure 6.3g). If the segment misses the vertices, then the polygon is divided into three new polygons (Figure 6.3h).

(7) Edge-edge-edge – The intersection starts at a point in the interior of an edge and ends at a point in the interior of the same edge, possibly the same point. If the points are not the same, then two new vertices are added along the edge and the polygon is divided into three new polygons (Figure 6.3i). Otherwise, if the intersection is a single point on the edge, then a single new vertex is added along the edge and the polygon is divided into two polygons (Figure 6.3j).

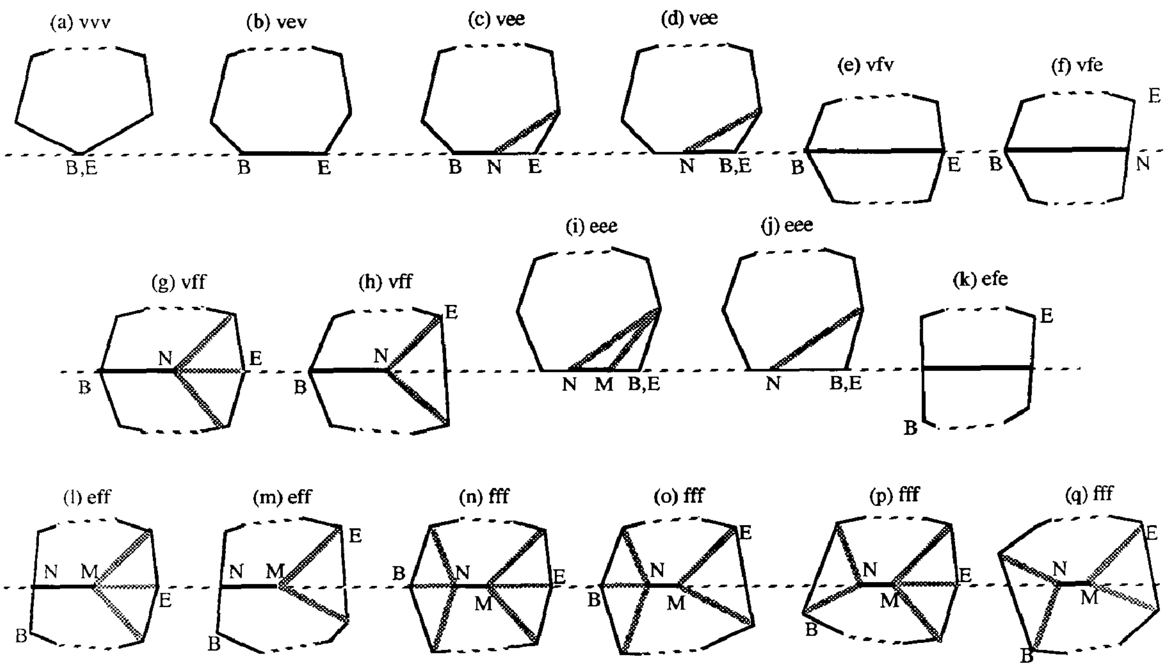


Fig. 6.3: Polygon subdivisions for different segment types

(8) Edge-face-edge – The polygon is cut across its face starting and ending at two different edges. Two new vertices are added along the edges and the polygon is divided into two polygons along the intersection line (Figure 6.3k).

(9) Edge-face-face – The segment cuts across part of the polygon starting at an edge and ending in the interior of the face. Two new vertices are added, one in the face and one along the edge. If the extension of the intersection segment would pass through a vertex, then the polygon is divided into four new polygons, just as with (6) vertex-face-face (Figure 6.3l). Otherwise, the polygon becomes three new polygons (Figure 6.3m).

(10) Face-face-face – In this final case the intersection segment both starts and ends in the interior of the polygon, possibly at the same point. If the intersection is a single point, then one new vertex is added, otherwise two new vertices are added. As with (9) edge-face-face and (6) vertex-face-face, the continuation of the segment will hit either a vertex or an edge of the polygon, this time in both directions. Figures 6.3n-q illustrate how the polygon is divided into four, five, or six new polygons depending on where the segment crosses the perimeter of the polygon.

These descriptions mention several operations that have not yet been explained. Some add a vertex in the interior of an edge, some add a vertex in the interior of a face, and most replace a polygon with several new polygons.

When a vertex must be added in the interior of an edge, the intersection segment structure contains the distance of the new vertex from P on L. By using that distance and the equation for the line of intersection, we can find the coordinates of the point. The intersection segment structure also contains the index of the first vertex of the edge that is intersected. The calculated point, which may have suffered from some floating-point error, is projected onto this edge. If the vertex has coordinates different from all existing vertices, then a new vertex is added to the object; otherwise, nothing is added.

Adding a vertex that lies in the interior of a face is more complicated. Again, the approximate coordinates are found by substituting the distance of the new point along L into the equation of L. The new point is then projected onto the plane of the polygon and the vertex is added just as a new vertex along an edge is added.

In addition to updating edges and adding new vertices, polygons must be replaced with smaller polygons. Figure 6.3 shows how a polygon is subdivided in each case, but if the original polygon has few vertices, the decomposition may produce degenerate polygons containing only two vertices. These polygons should not be added to the object, and are ignored. Figure 6.4 (after section 10) shows wireframe renderings of a pair of overlapping cubes and their state after having been intersected with each other.

7. Classifying Polygons

A routine that determines the position of polygonA relative to objectB is used several times by the algorithm. It is given objectB and a polygonA and returns the position of polygonA with respect to objectB: INSIDE, OUTSIDE, on the boundary of objectB with the normal vector facing in the SAME direction as the normal vector to objectB at that point, or on the boundary of objectB with the normal vector facing in the OPPOSITE direction.

The average of the vertices of a polygon is called the barycenter. A ray is cast from the barycenter of polygonA in the direction of the normal vector to polygonA, and is intersected with every polygonB in objectB. The polygonB that intersects the ray closest to the barycenter is found. If the barycenter does not lie in the plane of the nearest polygonB, then the direction of the normal vector to polygonB determines whether polygonA is inside or outside objectB. If the normal to polygonB points toward polygonA, then polygonA is OUTSIDE objectB; otherwise, polygonA is INSIDE objectB. If no polygons were intersected, then polygonA is OUTSIDE objectB. If the origin of the ray lies in the plane of the nearest polygonB, then polygonA lies in the boundary of objectB. In this case, if the normal vectors of polygonA and polygonB point in the same direction polygonA is classified as SAME; otherwise, it is classified as OPPOSITE. These two classifications are used in the next section.

The ray can intersect each polygonB in objectB in several different ways. To determine an intersection type, we need to know the dot product of the ray being cast with the normal vector of the polygonB being checked, and the signed distance from the barycenter to the plane of polygonB in the direction of the normal vector. Figure 7.1 shows the five possible intersection types.

First, if the signed distance is negative, then polygonB is behind the barycenter and can be ignored.

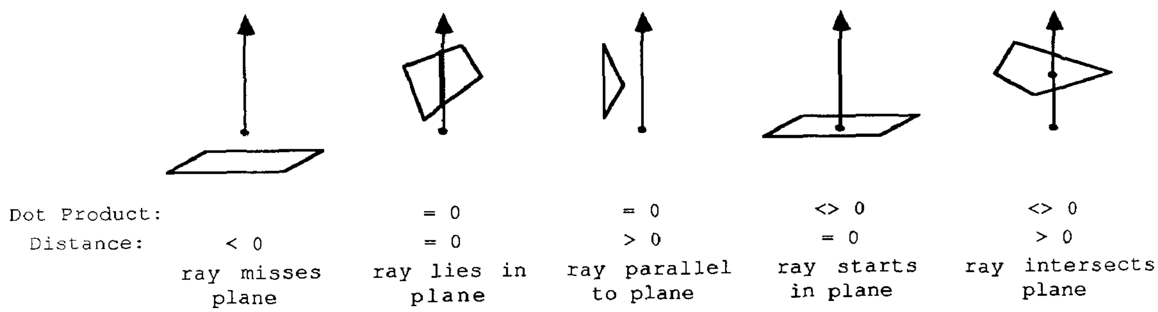


Fig. 7.1: Intersections of a ray and polygonB

Second, if the dot product and the distance are both zero, then the ray lies in the plane of polygonB. Without complicated analysis of all the polygons that the ray intersects, it is impossible to determine the status of polygonA in this case, so the direction of the ray must be perturbed by some small random value and the classification retried for the new direction. Although it is theoretically possible that an infinite number of random perturbations will all lead to invalid directions, in our implementation we have never needed to perturb the direction more than once to find a valid direction.

Third, if the dot product is zero and the distance is positive, the ray is parallel to the plane but never intersects it and therefore does not intersect polygonB.

Fourth, when the dot product is non-zero and the distance is zero, the barycenter lies in the plane of polygonB. If the barycenter lies outside polygonB in that plane, then the ray does not intersect polygonB; otherwise, polygonA lies on the boundary of objectB and this must be the closest intersection.

The fifth case occurs when the dot product is non-zero and the distance is positive. The point of intersection of the ray and the plane of polygonB must be inside polygonB, outside polygonB, or on an edge of polygonB. If an edge is hit, the ray must be perturbed and recast for all polygonBs in objectB. If the ray misses the interior of polygonB, then there is no intersection. Otherwise, the ray intersects polygonB, and this intersection is saved if it is closer than any intersection yet found. Figure 7.2 shows pseudocode for the polygon classification routine.

8. Marking Vertices

Once each object has been split so that none of the polygons in either object intersects any of the polygons in the other object, all the vertices in each object that lie on the boundary of the other object will have been marked as BOUNDARY vertices by the routines that subdivided each polygon. This section describes how the remaining vertices, still marked as UNKNOWN, are classified as lying INSIDE or OUTSIDE the other object so that the set of polygons that make up the resulting object may be found. This resulting set of polygons is a subset of all the polygons in both of the objects. Whether or not each polygon is in this subset depends on whether it lies INSIDE, OUTSIDE, or on the BOUNDARY of the other object. The polygon classification routine could be called to classify each polygon in both objects, but this would be time-consuming. Instead, all the vertices of the object are classified by classifying just a few of the polygons. Once all the vertices have been classified, all of the polygons that have at least one vertex not in the boundary of the other object can be classified, and only the polygons that have exclusively boundary vertices need to make extensive use of the ray-casting routine. This procedure must be executed for both objects.

```
create a RAY starting at the barycenter of polygonA
  in the direction of the normal of polygonA
while no successful cast has been made
  for each polygonB in objectB
    find the DOT PRODUCT of RAY direction
        with the normal of polygonB
    find the DISTANCE from barycenter to the plane of polygonB
    if (DOT PRODUCT = 0) and (DISTANCE = 0)
      cast is unsuccessful – leave loop and perturb
    else if (DOT PRODUCT = 0) and (DISTANCE > 0)
      no intersection
    else if (DOT PRODUCT <> 0) and (DISTANCE = 0)
      if RAY passes through interior or edge of polygonB
        save polygonB – this is closest possible intersection
      else
        no intersection
    else if (DOT PRODUCT <> 0) and (DISTANCE > 0)
        find intersection point of ray with plane of polygonB
      if intersection is closest yet
        if RAY passes through interior of polygonB
          (first check if point is within extent of polygonB)
          save polygonB
        else if RAY hits an edge of polygonB
          cast is unsuccessful – leave loop and perturb
        else
          no intersection
  if cast is unsuccessful
    perturb RAY by a small random value
end while
if there were no intersections
  return OUTSIDE
find the polygonB closest to POINT
find the DOT PRODUCT of closest polygonB normal and RAY
find the DISTANCE to closest polygonB
if (DISTANCE == 0)
  if (DOT PRODUCT > 0)
    return SAME
  else if (DOT PRODUCT < 0)
    return OPPOSITE
else if (DOT PRODUCT > 0)
  return INSIDE
else if (DOT PRODUCT < 0)
  return OUTSIDE
```

Fig. 7.2: Polygon Classification Routine

We first use the edges of the subdivided polygons to calculate the adjacency information for each object. Then begin at the first polygon in the object structure that contains a vertex marked UNKNOWN. The polygon cannot lie in the boundary of the other object, since it contains at least one vertex that does not lie on the boundary; the polygon classification routine determines if the polygon is INSIDE or OUTSIDE the other object. The vertex is marked appropriately, and all UNKNOWN vertices connected by edges to this vertex are marked identically. Since all BOUNDARY vertices were detected when the polygons were split, the vertices of each object have been divided into connected regions: each connected region is separated from other regions by boundary vertices, and all the vertices in a connected region of one object lie on the same side of the other object. Once the entire region has been marked, another polygon with vertices marked UNKNOWN is found. The operation is repeated until all polygons have been checked and all vertices classified. Figure 8.1 shows pseudocode for the region-marking routine.

```
Region-Marking Routine:
calculate adjacency information for all vertices of objectA
for each polygonA in objectA
  if any vertices are marked UNKNOWN
    call Polygon Classification Routine
        to determine if polygonA INSIDE/OUTSIDE objectB
    for each UNKNOWN vertex in polygonA
        call Vertex Marking Routine

Vertex-Marking Routine:
mark the specified UNKNOWN vertex as INSIDE/OUTSIDE
for each vertexA' adjacent to vertexA
  if vertexA' is marked UNKNOWN
    call this routine recursively for vertexA
```

Fig. 8.1: Region- and vertex-marking routines

9. Selecting Polygons for Output

Once the two objects have been intersected and all vertices have been classified as INSIDE, OUTSIDE, or BOUNDARY, the polygons that comprise the resulting object must be selected. Figure 9.1 shows which polygons are in the set of polygons that comprise the CSG combination of the two objects.

	polygons in A			
	inside	outside	same	opposite
$A \cup B$	no	yes	yes	no
$A \cap B$	yes	no	yes	no
$A - B$	no	yes	no	yes

	polygons in B			
	inside	outside	same	opposite
$A \cup B$	no	yes	no	no
$A \cap B$	yes	no	no	no
$A - B$	yes	no	no	no

Fig. 9.1: Selecting polygons for output

When a difference is performed, each polygonB inside objectA must have the order of its vertices reversed, and its normal vector must be inverted, since the interior of objectB becomes the exterior of the resulting object. Faces classified as SAME or OPPOSITE in one object exactly match faces in the other object. In the combination, at most one face needs to be added. We have chosen to always take that face from objectA, so polygons in objectB classified as SAME or OPPOSITE are never retained. Once the appropriate polygons have been deleted from both objects, the objects are combined to form the resulting object.

Most polygons are classified by examining the classifications of their vertices. Polygons that have only boundary vertices are classified by the ray-casting routine described earlier. Vertices classified as INSIDE or OUTSIDE are deleted or kept according to the table in Figure 9.1, although BOUNDARY vertices are never deleted. Once the polygons have been deleted, the normals and vertices reversed if necessary, and the object parts linked together, the CSG operation is complete.

The pseudocode in Figure 9.2 for the polygon selection routine makes use of the polygon classification routine described previously. Figure 9.3 (after section 10) shows wireframe and raster renderings of two cubes which have been unioned, intersected, and differenced.

```
(called by the union, intersection, and difference control routines)
(deletes polygons in objectA that are STATUS relative to objectB)

for each polygonA in objectA
  for each vertexA in polygonA
    if the status of vertexA is not BOUNDARY
      the status of the polygonA is the status of vertexA
  if no status for polygonA was found
    determine status of polygonA
    using the polygon classification routine
  if polygons of this status should be deleted for this operation
    delete polygonA from objectA

for each vertexA in objectA
  if vertices with this status should be deleted for this operation
    delete vertexA
```

Fig 9.2: Selecting polygons for output

10. Conclusions

We have presented a straightforward yet robust algorithm for performing CSG operations on polygonal objects. The algorithm runs in $O(V^2+P^2)$ where V is the total number of vertices and P the total number of polygons in both objects after subdividing. The time can probably be reduced to $O(V\log V+P\log P)$ with suitable sorting of polygons and vertices.

Floating-point granularity must be considered when implementing this algorithm. A CSG combination that strains many commercial solid modellers combines two unit cubes, one rotated N degrees first around the x-axis, then around the y-axis, and finally around the z-axis [JOH86] Most commercial solid modelers fail when $0.5\,degree < N < 1\,degree$. Our implementation is successful for $N > 0.1\,degree$. Rather than failing catastrophically on the test case for smaller values of N, the algorithm detects a potential error and prints an error message. The error is detected when the signed distances from the vertices of one polygon to the plane of another are calculated. A consistency check signals that the calculated distances are impossible.

There are several places where we attempt to correct possible floating point problems. All floating point comparisons are made so that numbers that differ less than a small predefined value are considered equal. For example, when a vertex is added to an object, the list of existing vertices is checked for an equivalent vertex using the approximate comparison above. If a match is found, then the coordinates of the new vertex are set to be identical to the coordinates of the vertex that was found. In addition, when the coordinates of a new vertex that lies on the edge or face of a polygon are calculated, the calculated value is projected onto the edge or face to ensure that small errors will not propogate.

We are currently continuing work on this algorithm in several directions. This algorithm divides polygons up more than is strictly necessary. After several operations, what might have been a single polygon in the resulting object may have instead become 10 or 20. We would like to combine these coplanar faces to reduce the number of polygons in a resulting object. Also, when several CSG operations must be performed to generate an object, the intermediate results are often not of interest. A modification of this algorithm might subdivide all of the sub-objects at once, classifying each polygon with respect to all of the other objects. If the entire operation were performed at one time, a tremendous amount of overhead from individual operations might be saved.

Figure 10.1 shows a spoon described in SCEFO using CSG. The ray-traced image was rendered in 1300 CPU seconds on a VAX 11/780 running 4.2bsd UNIX. The polygonal image was rendered using a Z-buffer algorithm and this CSG algorithm, taking 76 seconds on the same machine. Both images were rendered at a resolution of 640 $\times$ 512 pixels, and the ray-traced image is antialiased. In addition to making quick polygonal renderings possible, this algorithm is used to generate wireframe representations of objects for interactive modeling and animation previewing.

11. Acknowledgements

We would like to thank Trina Avery, Matthew Kaplan, Barbara Meier, Joseph Pato, and Andries van Dam for reading early versions of the paper. Thanks also goes to Andries van Dam and the Brown University Computer Graphics Group for their assistance and support.

12. References

FOL82 Foley, J. D. and A. van Dam, *Fundamentals of Interactive Computer Graphics,* Addison-Wesley, 1982.

REQ80a
Requicha, A. A. G. and H. B. Voelcker, "Constructive Solid Geometry," Production Automation Project Technical Memorandum TM-25, April 1980.

REQ80b
Requicha, A. A. G. and H. B. Voelcker, "Mathematical Foundations of Constructive Solid Geometry: General Topology of Closed Regular Sets", Production Automation Project Technical Memorandum TM-27a, November 1980.

REQ83 Requicha, A. A. G. and H. B. Voelcker, "Solid Modeling: Current Status and Research Directions," *IEEE Computer Graphics and Applications,* 3 (7), October 1983.

REQ85 Requicha, A. A. G. and H. B. Voelcker, "Boolean Operations in Solid Modeling: Boundary Evaluation and Merging Algorithms," *Proceedings of the IEEE* January 1985, pp. 30-44.

ROT82 Roth, Scott, "Ray Casting for Modeling Solids," *Computer Graphics and Image Processing* 18 (1982), pp. 109-144.

STR84 Strauss, P., M. Shantzis and D. Laidlaw, "SCEFO: A Standard Scene Format for Image Creation and Animation," Brown University Graphics Group Memo, Providence, R.I., 1984, 32 pp.

TUR84 Turner, James A., "A Set-Operation Algorithm for Two- and Three-Dimensional Geometric Objects," Architecture and Planning Research Laboratory, College of Architecture, University of Michigan, Ann Arbor, MI, August, 1984.

JOH86 Johnson, Robert H., *Solid Modeling: A State of the Art Report (Second Edition),* CAD/CIM Alert, Management Roundtable, Inc., 1986.

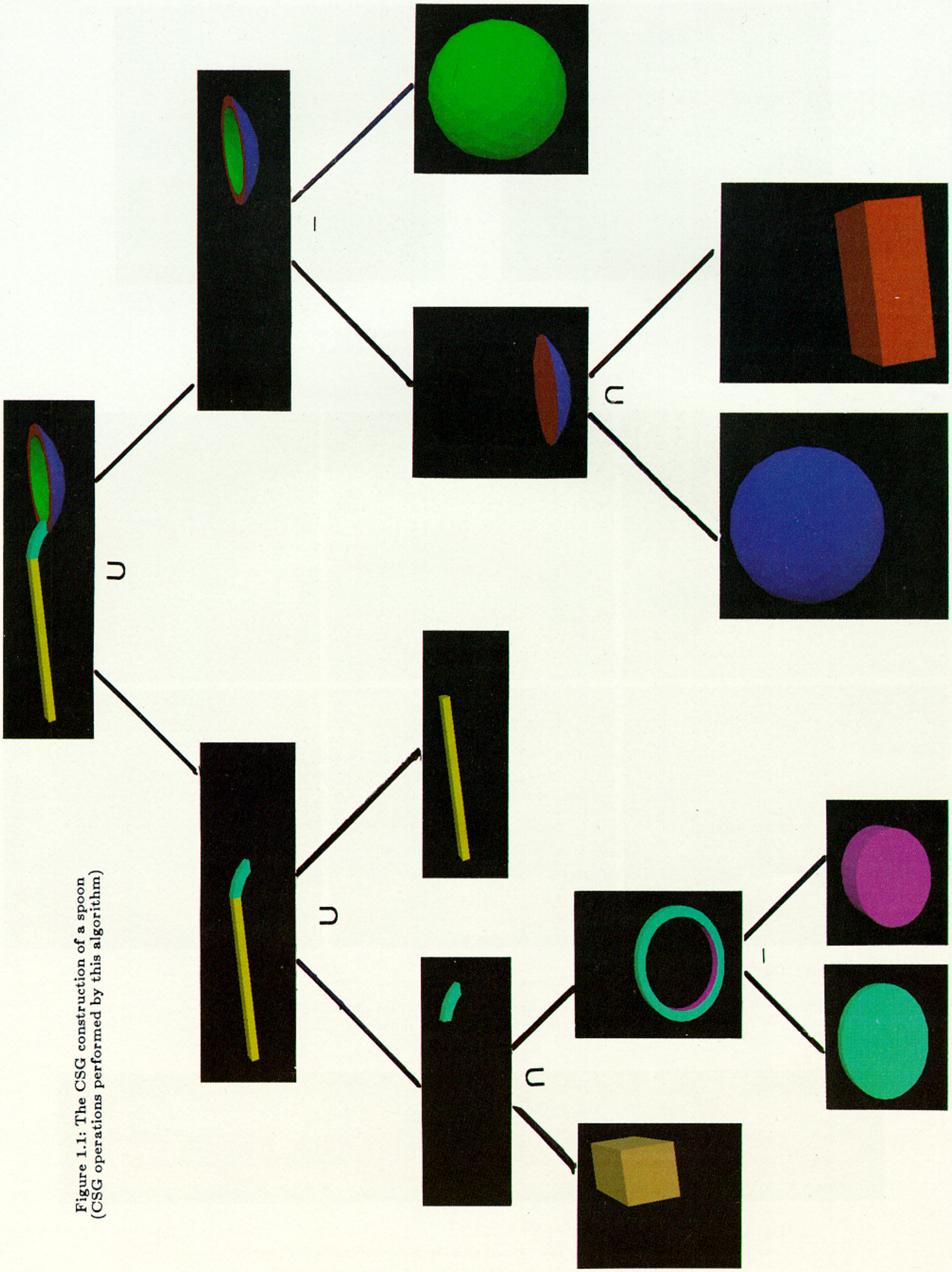
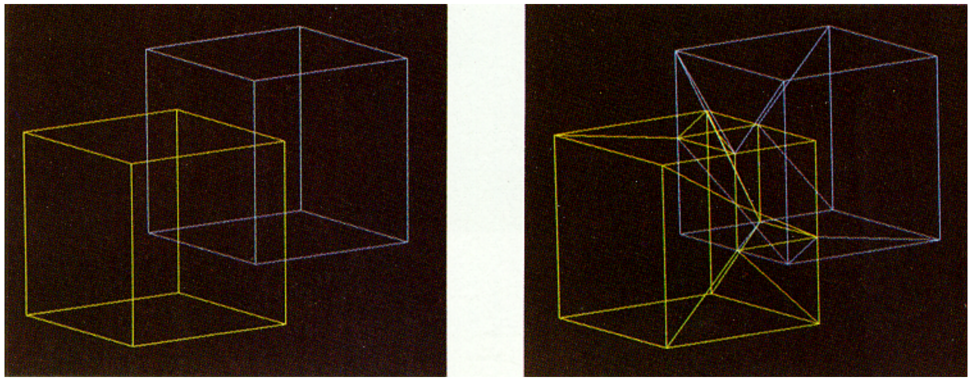
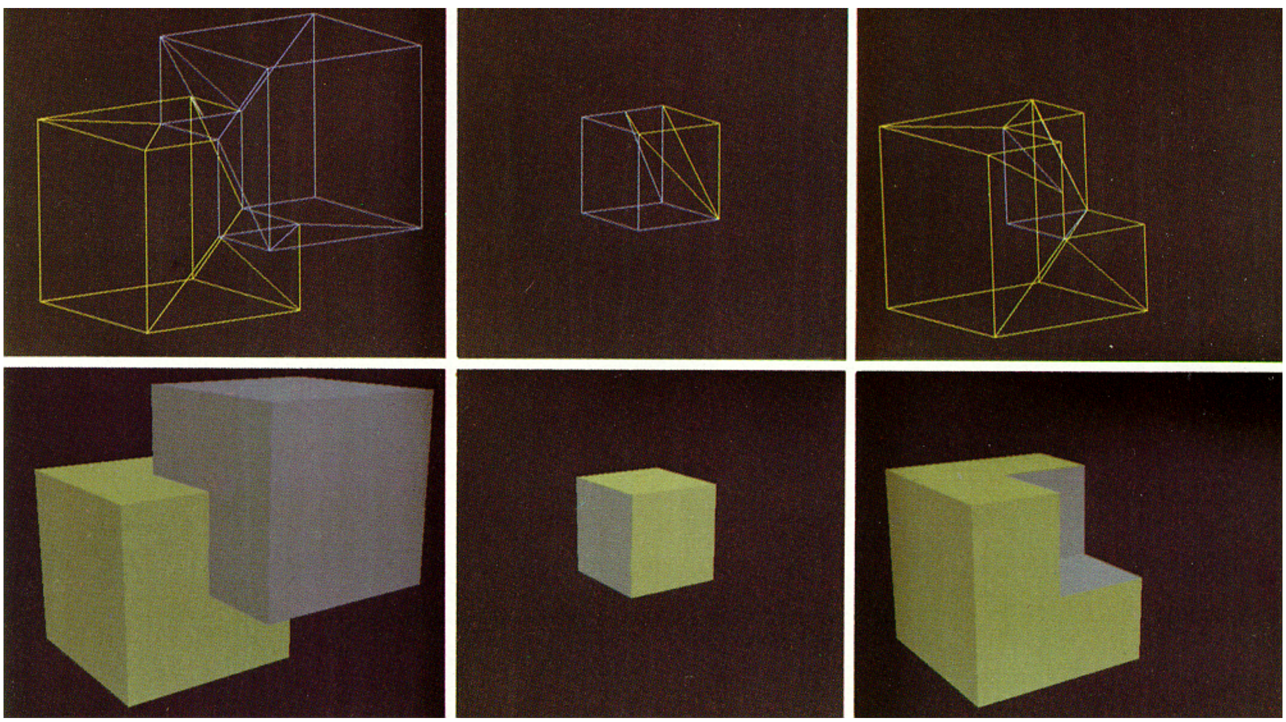


Figure 1.1: The CSG construction of a spoon
(CSG operations performed by this algorithm)



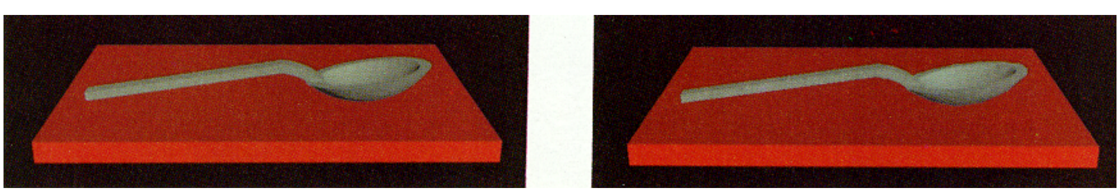
(a) (b)

Figure 6.4: (a) Two cubes positioned so that they overlap
(b) The result of splitting the two objects against each other



(a) (b) (c)

Figure 9.3: Wireframe and raster renderings of two cubes:
(a) union, (b) intersection, (c) difference



(a) (b)

Figure 10.1: (a) was ray-traced and (b) was rendered polygonally
(both images were generated from the same description)