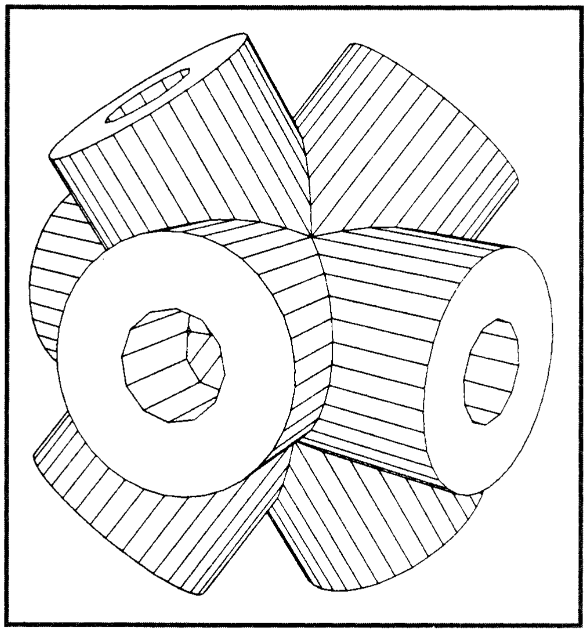
Partitioning Polyhedral Objects into Nonintersecting Parts

Mark Segal and Carlo H. Sequin
University of California, Berkeley

We describe an algorithm for partitioning intersecting polyhedrons into disjoint pieces and, more generally, removing intersections from sets of planar polygons embedded in three space. Polygons, or faces, need not be convex and may contain multiple holes. Intersections are removed by considering pairs of faces and slicing the faces apart along their regions of intersection. To reduce the number of face pairs examined, bounding boxes around groups of faces are checked for overlap.

The intersection algorithm also computes set-theoretic operations on polyhedrons. Information gathered during face cutting is used to determine which portions of the original boundaries may be present in the result of an intersection, a union, or a difference of solids.

The method includes provisions to detect, and in some cases overcome, the effects of numerical inaccuracy on the topological decisions that the algorithm must make. The regions in which ambiguous results are possible are flagged so that the user can take appropriate action.



Constructive solid geometry (CSG) has become an essential feature of almost every solid modeling system. With CSG, complex solid objects are built from intersections, unions, and differences among primitive objects. The resultant objects can then be used as building blocks for more complex solids.

A polyhedral CSG object can be represented as a tree with primitive solids having planar polygonal boundaries at the leaves and CSG operations at the nodes. This representation is sufficient for hidden-surface removal with suitable renderers.[1,2] The CSG tree does not, however, provide a direct description of the object's boundary.

Many applications, especially hidden-surface algorithms exploiting object coherence, require an explicit boundary representation consisting of noninter-

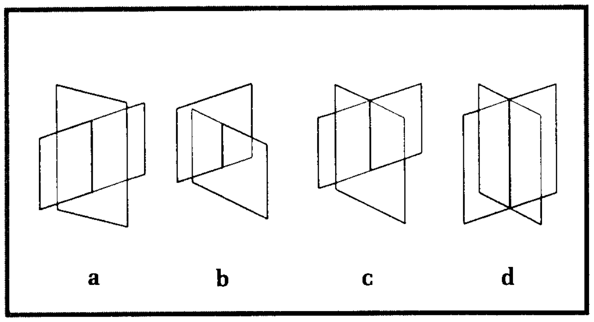


Figure 1. Four cases of intersection: (a) a cut with a slit, (b) two notches, (c) a cut with a notch, (d) two cuts.

secting planar polygonal faces.[3,4] The physical properties of an object (center of mass or moments of inertia, for instance) are also most conveniently and efficiently computed with such a description.[5,6] Finally, to determine topological (coordinate invariant) properties of an object, an explicit description of its boundary is essential.

The algorithm we present can produce a polyhedral object's boundary from its CSG tree. We partition the boundaries of indicated polyhedral objects into nonintersecting parts so that the regularized boundary of the intersection, union, or difference of the objects can be produced.[7] Since the algorithm bases its operation on removing intersections from pairs of faces, it can be used to partition a list of intersecting faces into nonintersecting parts. The approach works on faces with an arbitrary number of contours of arbitrary complexity, without first breaking them into simpler pieces.[8,9]

Explicit tolerance values are maintained with all vertices, edges, and faces, and are used to locate regions in which numerical inaccuracy may lead to ambiguous topology. Various heuristics overcome most situations of this type, but some require user intervention.

Overview of the algorithm

The algorithm operates by looking for and removing intersections among pairs of candidate objects. An object is either a single face or a collection of other objects. Intersections are removed from a collection by removing pairwise intersections between its component objects. Bounding boxes are used to determine whether two collections within an object interfere with each other.[10] If they do interfere, each collection is expanded into a list of the objects that constitute it. The items in the lists are individually checked and processed for internal intersections, if necessary. Then, each object from one list is checked against each object from the other, removing any pairwise intersections. This process is applied to the collection of objects supplied as input, eventually removing all intersections. The recursion can be summarized as follows:

```
procedure intersect($object_1$, $object_2$)
        if $object_1$'s bounding box does not overlap
            $object_2$'s bounding box,
        then return;
        if $object_1$ and $object_2$ are faces
        then call faceIntersect($object_1$, $object_2$); (see text)
        else  if $object_1$ is a collection
              then for each object $O \in object_1$
                        call removeIntersect($O$);
                        call intersect($O$, $object_2$);
                   end
              else  for each object $O \in object_2$
                        call removeIntersect($O$);
                        call intersect($object_1$, $O$);
                    end

end intersect

procedure removeIntersect($O$);
        if intersections already removed from $O$
        then return;
        else for each pair of objects $object_1, object_2 \in O$
                call intersect($object_1$, $object_2$) ;
             end
end removeIntersect


Main program:
        call removeIntersect(input);
```

The lowest level of the recursion detects and removes intersections from a particular pair of faces (the call to faceIntersect). The intersection of a pair of faces may be empty, or may consist of either a set of collinear line segments (for transversal faces) or a set of coplanar polygonal regions (for coplanar faces).

A face pair is classified by determining if at least one edge of each face crosses the other's plane (see section below entitled "Intersection classification"). If this is the case, all intersection points of each face's edges with the other's plane are computed, combined into a list, and ordered along the intersection line. The sorted list determines the intersection segments along which each face is topologically partitioned.[11]

If all edges of one face lie in the other's plane, the faces are deemed coplanar.[12] General partitioning of disconnected faces requires a procedure to partition overlapping polygons, but our algorithm takes advantage of connectedness among a solid's boundary faces, partitioning coplanar faces with no additional work.

If the faces are not coplanar, the list of sorted intersection points indicates where the faces intersect, if they do at all. In cases of intersection, both faces are sliced apart (see "Intersecting transversal face pairs" below). There are four cases (see Figures 1a-1d). In Figure 1a, one face is cut apart along the line, while a slit (a hole of zero width) is inserted into the other face. Figure 1b shows how both faces are notched to accommodate an intersection not bounded by edges of the same face. The two other cases arise if intersection points from the two faces coincide. In Figure 1c, one face must be cut and the other notched. In Figure 1d, both faces must be cut.

These diagrams illustrate simple cases; typically a face must be sliced several times. The slits, notches, and cuts induced in a face by other faces link up, eventually partitioning the face.

When the intersection removal computes a CSG operation on two or more solids, membership information is gathered during face cutting. After all intersections have been removed, the information determines which face portions belong in the desired result and which do not (see "Solid modeling" below). The latter portions are thrown away, while the former are outputted. The results of several distinct Boolean operations can thus be produced simultaneously.[13]

An important feature of the algorithm is its immunity to moderate numerical inaccuracies in the input description (see "Numerical accuracy" below). Difficulties arising from slightly "warped" faces and grazing intersections are overcome, ensuring, under appropriate conditions, that a CSG operation applied to the boundaries of several solids results in the boundary of another solid.

Geometric items and data structures

There are five geometric items on which the algorithm operates: vertices, edges, contours, faces, and solids. The corresponding data structures that the algorithm builds from an ASCII description have been described in detail elsewhere;[4] they are reviewed here briefly.

Vertices

A vertex specifies a point in space given by three coordinates. Its data structure contains a triple of floating-point coordinates and a tolerance indicating their accuracy. In addition, there is a list of all edges to which the vertex is connected.

Edges

An edge joins a pair of vertices defining its endpoints. Each edge structure has pointers to its starting vertex and to its ending vertex. In addition, a list indicates in which contour(s) of which face(s) the edge is used and the corresponding location within those contours.[14]

Contours

A contour represents a single closed planar polygonal curve. Self-intersecting contours are not allowed. A contour may assume one of two orientations; a component edge may be used in either direction. Each contour structure is a singly linked list of pointers to edges. Each list element also shows whether the edge is used in the defined or reversed direction. The list elements are circularly linked, facilitating the splitting and merging of contours.

Faces

A face specifies by one or more contours a set of finite two-dimensional patches in the same plane. A normal and a distance from the origin define the plane in which the patches lie. Each contour divides the face's plane into two areas: a finite area "inside" the contour and an infinite area "outside" the contour.[15] The cross product of the face normal with the direction of a contour edge points from the edge into the region enclosed by the contour. A contour enclosing its outside is called a hole.

Each contour must bound an open planar set so that local topological determinations are possible.[16] This restriction ensures that a contour does not contain zero-width "spikes," although zero-width "notches" are allowed (see Figure 2).

The representation of a face consists of a list of contours and four real numbers describing its plane equation. A tolerance indicates the accuracy of the plane equation's coefficients. If appropriate, each face also points to the solid to which it belongs.

Solids

Finally, any face collection may be flagged as forming the boundary of a solid in three space. CSG operations may be specified on such collections. The normals to the boundary faces point away from the solid's interior, inducing a boundary orientation that distinguishes inside from outside for solid modeling operations.

We recognize two types of boundary: complete and partial. A complete boundary encloses a solid, dividing space into the solid's inside and outside. A partial boundary divides space only locally into an inside and an outside. Sometimes it is useful to work with partial boundaries; for instance, a large square face can be used as a "knife" to cut a solid into two parts or sets of parts. Since boundaries are treated essentially as lists of faces, boundaries may be either partial or complete. However, as Figure 3 demonstrates, careless use of partial boundaries can lead to ambiguous CSG results.

No checks are made to ensure that faces in a collection do not intersect and are connected so as to define an oriented polyhedral boundary.[17] It is the responsibility of the calling environment to make sure that any face col-

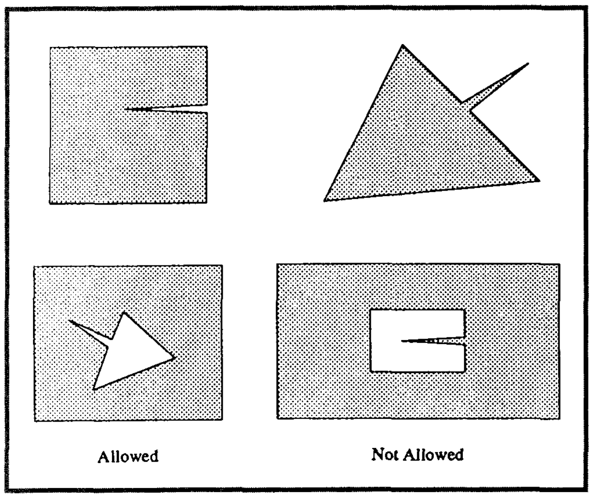


Figure 2. Notches and spikes.

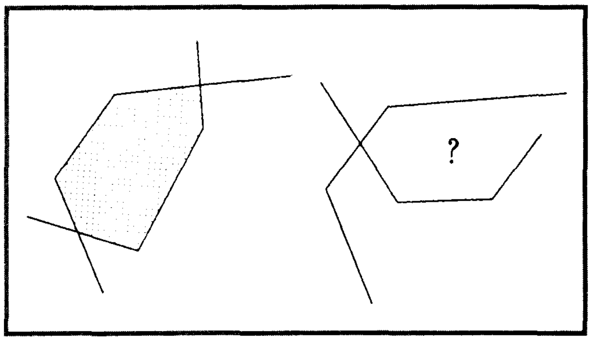


Figure 3. Partial boundaries in two dimensions. The first intersection region is well defined; the second is not.

lection so used is correctly defined.

The data structure of a solid contains—in addition to a list of faces defining its boundary—the dimensions of its enclosing bounding box.

Intersection classification

The first step in removing intersections from a pair of faces is to obtain the plane equation for each face. Newell's algorithm[18] is used to compute the plane equation if one is not provided in the input.

Next, the coordinates of every vertex belonging to a particular face are substituted into that face's plane equation. The magnitude of the largest deviation from zero added to the largest vertex tolerance is recorded as the face's tolerance.[12] This tolerance defines a slab that the plane, imperfectly specified by the vertices, is known to lie within. Vertex tolerances, if not supplied, are assigned a small value (see "Numerical accuracy" for more about tolerances and their use). The face tolerance is used to determine whether a vertex lies within the face or to one side of it.

The faces are then checked to see whether they intersect transversally, do not intersect, or are coplanar. Each contour of one of the faces is traversed and the oriented (signed) distances of its vertices from the other face are found. Traversal continues until either two vertices lie on opposite sides of the other (transversal) face, or all edges of all contours in the first face have been traversed.

If the first situation arises, the process is repeated with the faces' roles interchanged. If, during the second traversal, vertices of the second face are found on opposite sides of the first, the faces may intersect and transversal intersection processing continues. The vertex giving rise to the first sign change is saved to avoid reexamination of those edges known not to cross the other face's plane.

If, on the other hand, the second situation arises during the first or second traversal, the faces either do not intersect or are coplanar. The faces are deemed coplanar if all the vertices of one face lie within tolerance of the other's plane. If coplanar, the pair is recorded for possible use in boundary classification. If not, the faces do not intersect and are returned untouched.

Intersecting transversal face pairs

The core of the intersection algorithm modifies the contours of two transversal faces so that they no longer intersect. After a pair of transversal faces is processed, the only regions that the two share will lie on common contour edges or vertices.

Definitions

If they meet at all, two transversal face planes meet in an intersection line. The intervals that a face patch (as defined by contours) has in common with the intersection line are *segments*. Each segment is defined by its two *endpoints*. An endpoint is an *entry* if it lies at the end of the segment with smaller parameter value along the intersection line $\mathbf{v} + t\mathbf{u}$; otherwise, it is an *exit*. The regions formed by the intersection of segments from a pair of faces are called *cutting intervals*. These intervals are the portions of the intersection line in which a pair of faces intersect each other; they are therefore the regions along which face patches must be partitioned. Cutting intervals are found by determining their starting and ending endpoints.

Overview

First, one of the faces is deemed the main face, and the other, the transversal face. Each contour of the main face is traversed for edge intersections with the transversal face's plane (and thus, the intersection line). These points determine intersection segments for the main face's patches. Then the roles of the faces are interchanged and the process repeated. Next, segment information from both faces (kept as a set of endpoints) is combined and sorted to find the cutting intervals. Finally, cuts, notches, and slits are inserted into the two faces as indicated by the sorted endpoints.

Intersection line calculation

Since the planes of the faces have been determined not to be parallel, we can compute their intersection line described by a direction vector, **u**, and a point on the line, **v**. Using the two normalized plane equations $\mathbf{n}_1 \cdot \mathbf{x} = d_1$ and $\mathbf{n}_2 \cdot \mathbf{x} = d_2$ with respective unit normal vectors $\mathbf{n}_i$, we find the direction **u** by computing $\mathbf{n}_1 \times \mathbf{n}_2$ and normalizing. To obtain a point on the line, we introduce a third arbitrary plane equation, $\mathbf{u} \cdot \mathbf{x} = d_3$, constructed to have normal **u** and to pass through some vertex in one of the two faces, yielding

$$\begin{bmatrix} \mathbf{n}_1 \\ \mathbf{n}_2 \\ \mathbf{u} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \end{bmatrix}$$

or $A\mathbf{x} = \mathbf{d}$. This system is nearly orthogonal, since each of the rows $\mathbf{r}_i$ of A is a unit vector, and $\mathbf{r}_1 \cdot \mathbf{r}_3$ and $\mathbf{r}_2 \cdot \mathbf{r}_3$ are both zero by construction. After altering $\mathbf{r}_1$ so that it is orthogonal to $\mathbf{r}_2$, the system becomes $Q\mathbf{x} = \mathbf{d}'$ with $Q^TQ = I$ so that $\mathbf{v} = Q^T\mathbf{d}'$.

Endpoint determination

Endpoints of segments can arise in one of two ways. If an edge of the main face penetrates the transversal face (its end vertices lie on opposite sides of the transversal face's plane), then the intersection point is an endpoint (see Figure 4). If a vertex at the end of an edge is found to lie within the transversal face, it sometimes is an endpoint, depending on the local orientation of the main face's contour at the vertex.

Whether the endpoint is an entry or an exit is found by comparing the direction of a vector pointing from the contour into the enclosing patch with the direction of the intersection line:

$$\mathbf{sign}((\mathbf{n}_m \times \mathbf{e}) \cdot \mathbf{u}) = \begin{cases} +1 & : \mathbf{entry} \\ -1 & : \mathbf{exit} \end{cases} \quad (1)$$

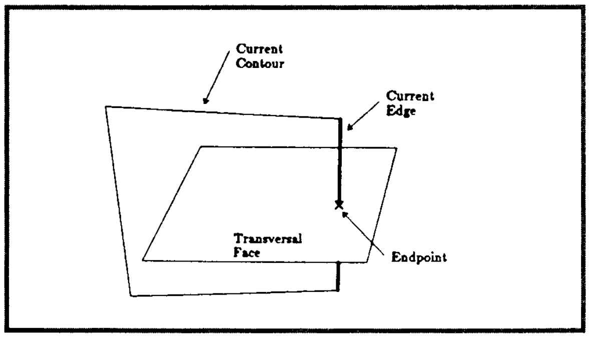


Figure 4. An endpoint occurring in the middle of an edge.

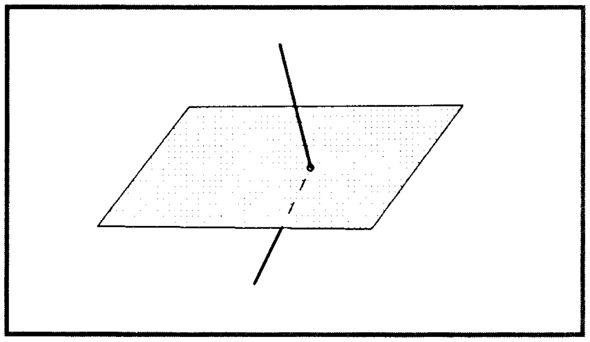


Figure 5. Vertex in the transversal face.

where $\mathbf{n}_m$ is the normal to the main face. Intersection points are not permanently entered into the data structure, because they may not represent cutting interval endpoints. Later, sorting will determine which new points should become incorporated vertices.

Sometimes an endpoint occurs at the end of an edge; a vertex of the main face may be found to lie in the transversal face. Whether or not such a central vertex $\mathbf{v}_c$ is an endpoint is determined by the orientation of the edges incident on it (with directions $\mathbf{e}_1$ and $\mathbf{e}_2$) relative to the intersection line. The various possibilities are illustrated in Figures 5 through 8. There are essentially four cases determined by the locations of the vertices adjacent to $\mathbf{v}_c$. Three of these four cases split into two subcases, depending on the orientation of the contour at $\mathbf{v}_c$.

The simplest case arises if both adjacent vertices lie on opposite sides of the transversal face (see Figure 5); the central vertex always defines an endpoint. Whether it is an entry or an exit is determined by choosing an incident edge and using Equation 1. Since neither adjacent vertex lies on the intersection line, this determinant cannot be zero.

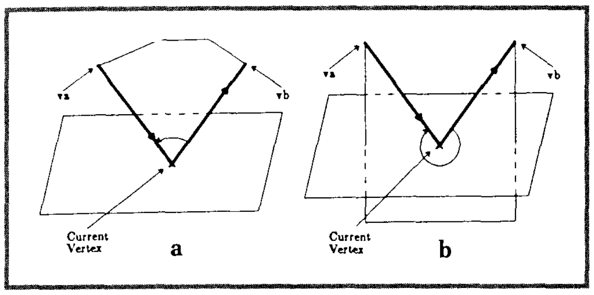


Figure 6. Neither adjacent vertex lies in the transversal plane: (a) No endpoint is indicated; (b) two endpoints are indicated at the marked vertex.

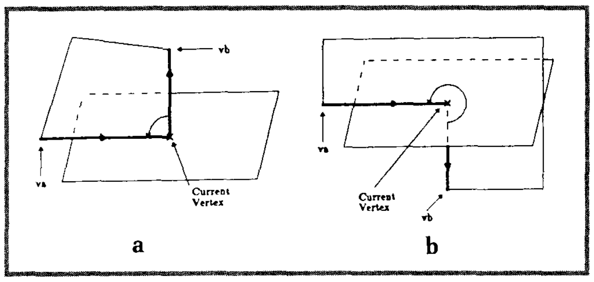


Figure 7. One adjacent vertex in the transversal plane: (a) The marked vertex does not define an endpoint; (b) the marked vertex defines an endpoint.

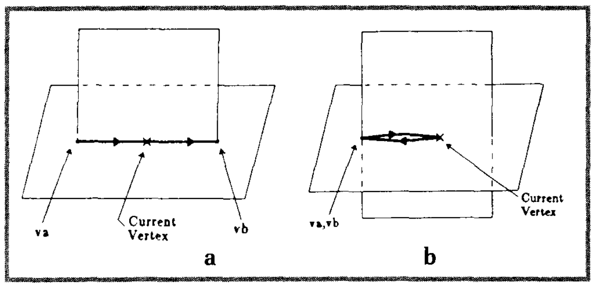


Figure 8. Both adjacent vertices in the transversal plane: (a) The vertex at × does not represent an endpoint; (b) the vertex at × represents an endpoint.

The other three cases are distinguished by whether (1) neither adjacent vertex lies in the transversal plane (but both lie on the same side of it), (2) one adjacent vertex lies in the transversal plane, or (3) both adjacent vertices lie in the transversal plane.

In the first case, the main vertex does not indicate an endpoint if the angle made by the two incident edges is less than 180 degrees (see Figure 6a). It indicates two endpoints (one entry and one exit) if the angle is greater than 180 degrees (Figure 6b). If sign($(\mathbf{n}_m \times \mathbf{e}_1) \cdot \mathbf{e}_2$) is positive, the angle is less than 180 degrees; if negative, the angle is greater than 180 degrees. A value close to zero indicates an angle close to 360 degrees (since "spikes" are not allowed).

If one adjacent vertex lies in the transversal plane and one does not, there are again two subcases, depending on contour orientation. The angle on the inside of the contour is compared with 180 degrees as in the previous case, except that this time the determinant cannot be near zero. If the determinant is positive, no endpoint occurs at the vertex (see Figure 7a); if negative, a single endpoint occurs there (Figure 7b). Whether an endpoint is an entry or an exit is found by setting $\mathbf{e}$ to $\mathbf{e}_1$ or $\mathbf{e}_2$, whichever does not lie along the intersection line, and computing $(\mathbf{n}_m \times \mathbf{e}) \cdot \mathbf{u}$ (Equation 1).

Finally, both adjacent vertices may lie in the transversal plane. If the edges go in the same direction, there is no endpoint at the vertex (see Figure 8a). If they go in opposite directions, the vertex is at the end of a notch and represents an endpoint (Figure 8b). The determination is made by examining sign($\mathbf{e}_1 \cdot \mathbf{e}_2$). Whether such an endpoint is an entry or an exit is determined by sign($\mathbf{e}_1 \cdot \mathbf{u}$).

Even if a vertex lying in the transversal plane is not an endpoint, it is placed in the endpoint list anyway. Sorting will determine if it lies within tolerance of a vertex in the transversal face, requiring a merge. These special vertices are otherwise ignored during intersection line processing.

Whenever an endpoint is found, its relative position on the intersection line is determined for sorting by finding the corresponding parameter value: $t = \mathbf{u} \cdot (\mathbf{p} - \mathbf{v})$, where $\mathbf{p}$ represents the endpoint. Also recorded with each endpoint is the edge incident on or containing it and the contour on which the endpoint is located.

Endpoints are also linked on circular lists, mimicking the structure of the contours from which they arose. This linking allows retention of the original endpoint ordering along each contour after sorting is completed. The original structure is required during cutting so that endpoint information can be updated when contours are rearranged.

Edge-induced endpoints

The determination of endpoints as described so far is sufficient for partitioning disconnected faces. However, a solid's boundary is connected from contour to contour along edges. Therefore, an edge that lands entirely within a face must induce a partition in that face (see Figure 9). Such an edge is recognized when both its end vertices lie in the transversal face. As shown in Figures 7 and 8, either zero or one endpoint may be produced at either vertex. But in all cases two "phantom" endpoints are entered into the endpoint list. These special endpoints may induce a cut in the transversal face. However, their phantom status is noted during intersection line traversal, so they do not induce cuts in the face that contributed them.

If two contours, one in the main face and one in the transversal face, share a common edge, each will contribute a pair of phantom endpoints, with no net effect after partitioning.

Sorting the endpoints

The calculated endpoints are sorted and grouped along the intersection line so that cutting intervals can be determined. Sorting is done first on the value of t associated with each endpoint. Since a single endpoint may be repeated up to two times, secondary sorting is performed first on the basis of the main face that contributed the endpoint, and second on the basis of whether the endpoint is an exit or an entry (see Figure 10).

Traversing the intersection line

The cutting intervals are found by traversing the sorted list of coalesced endpoints in the direction of increasing t. Two flags indicate whether the current position of the traversal lies inside or outside either of the two faces; both flags start indicating "outside." Each endpoint, encountered in turn, flips the state of the flag corresponding to the face whose edge spawned the endpoint. If, after encountering an endpoint, both flags indicate "inside," that endpoint must have been an entry. Each face is partitioned from this starting endpoint to the next endpoint in the sorted list (which must be an exit).

As many as two entry endpoints may appear at the start of a cutting interval: One from each face is possible. Sorting ensures that if there are two such endpoints, the second one lies just before the first in the endpoint list, making the situation easy to detect. Similarly, either one or two exit endpoints may be present at the end of the cutting interval.

The multiplicities of the cutting interval starting and ending points determine how each of the two faces is partitioned. For each face there are three basic possibilities: a cut, a notch, or a slit.

Cuts

A face is cut if endpoints from it are present at both the start and end of the cutting interval. There are two possible contour arrangements that can require a cut: The cut may split a single contour into two or merge a pair of contours into one (see Figure 11). In either case, the patch between the two endpoints is partitioned at the intersection line.

A cut begins with examination of the two supplied endpoints. If an endpoint represents a newly created vertex, the affected edge(s) must be split in two (see Figure 12). All uses of the same edge (possibly in other faces) are split at once.

Next, a new edge is created (if it is not already present) between the two vertices at the endpoints. The contours indicated by the endpoints are relinked to include the new edge twice, once in each direction (see Figure 13).

Whether a cut splits a contour or joins a pair of contours, portions of those contours may yet lie ahead in the

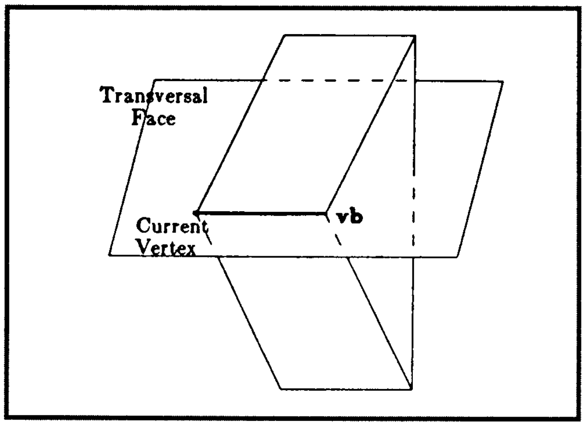


Figure 9. An edge lying entirely in the transversal face. Such an edge must induce a partition of that face.

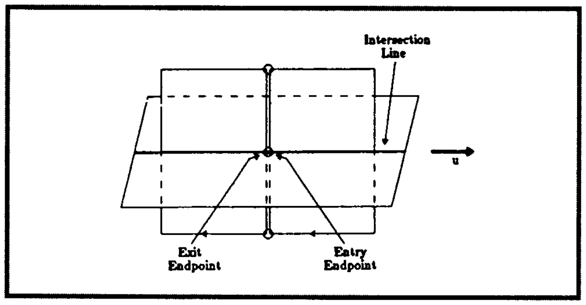


Figure 10. The use of entry and exit attributes to disambiguate endpoints during sorting.

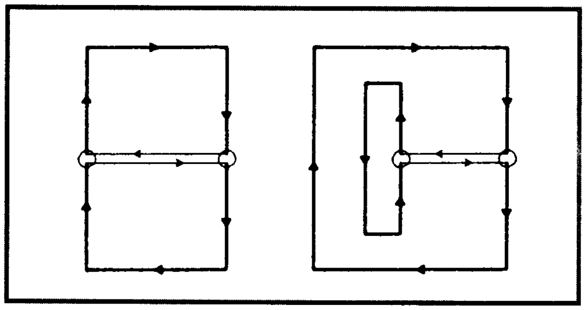


Figure 11. Two forms of a cut.

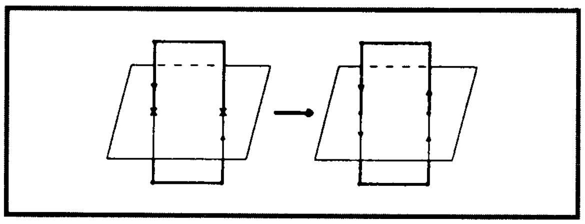


Figure 12. Subdividing edges.

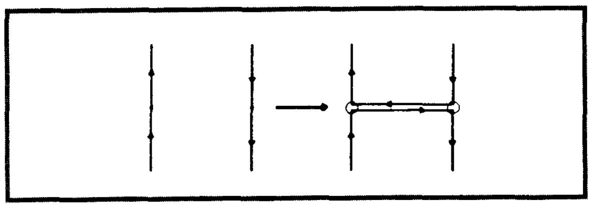


Figure 13. Reforming the contour.

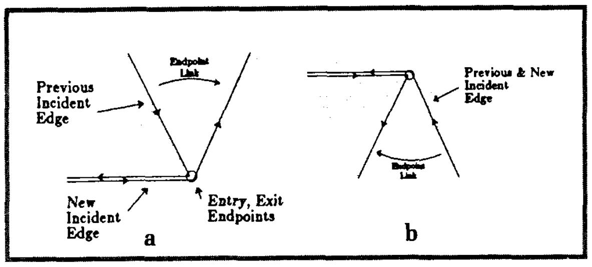


Figure 14. The situation at a double endpoint after a cut: (a) The incident edge must be updated; (b) the incident edge remains correct.

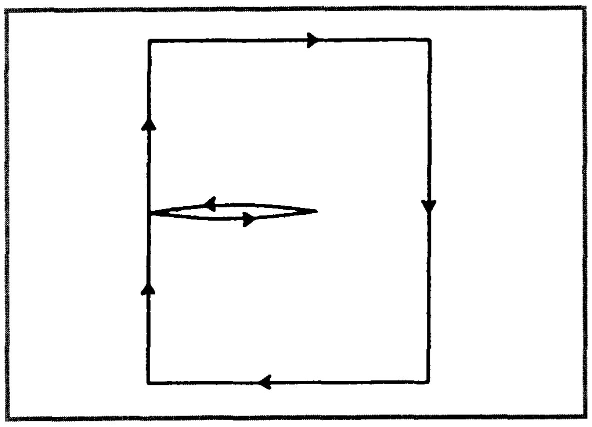


Figure 15. A notch.

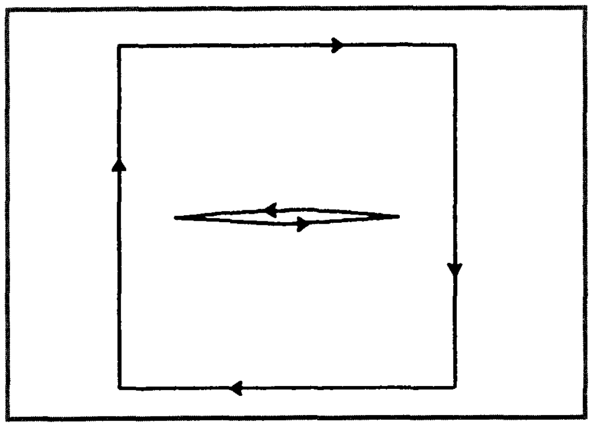


Figure 16. A slit.

list of sorted endpoints. The circularly linked list or lists of endpoints are relinked to correspond to the edge relinking and then used to propagate the new contour information to indicated unencountered endpoints.

A final check, made using the circular endpoint lists, determines if the exit endpoint is simultaneously an entry endpoint (see Figure 14). If so, the cut just completed may have replaced the edge incident on the as yet unprocessed entry endpoint (Figure 14a), requiring that its incident edge field be modified.

If the cut splits a contour, a flag is set indicating on which side of the transversal face each new contour lies. Recall that the edge along the intersection line has direction $\mathbf{u}$; let $\mathbf{n}_m$ be the normal to the face containing the partitioned contours. Then $\mathbf{s} = \mathbf{n}_m \times \mathbf{u}$ points from the edge into the contour that uses the edge in the forward direction. If $\mathbf{n}_t$ is the normal of the face transversal to the contours, then $\mathbf{s} \cdot \mathbf{n}_t > 0$ indicates that the contour is outside the transversal object; $\mathbf{s} \cdot \mathbf{n}_t < 0$ indicates that it is inside (since the faces are transversal, the dot product cannot be zero). The opposite is true for the other contour using the edge along the intersection line in the reverse direction.

Notches and slits

A notch (see Figure 15) is created in a face if one of the cutting interval endpoints arose from a contour of that face and the other arose from a contour of the transversal face. As with a cut, the edge in which the notch is made may have to be reformed into two edges. Then the notch's edge is created and inserted twice (once in each direction) into the contour.

A slit (see Figure 16) is placed in a face if both defining endpoints arose from a contour or contours of the transversal face. No edges need be split in this case. Again, a new edge is created and included once in each direction in a new contour.

Notch creation and slit creation are simpler operations than cutting because they do not require contour splitting or merging, or contour classification. However, either operation may create a new endpoint that will have to be considered if the traversal of the intersection line is not yet complete. A slit creates two new endpoints, one at each end of the slit. Only the one with the largest implied value of t need be considered, because only the portion of the intersection line corresponding to larger t has not yet been processed. Similarly, a notch starting at an edge and ending on an isolated vertex farther along the intersection line creates an entry endpoint that may enter into further partitioning.

Coplanar faces

If one face's vertices all lie within tolerance of another face's plane, the two faces are considered coplanar. Then overlapping face regions must be partitioned into disjoint regions bounded by nonintersecting contours in the faces' common plane. Such a coplanar partitioner[19] must be available for partitioning an arbitrary set of faces.

However, if all faces considered belong to boundaries of one or more solids, we need not implement this procedure. A face in a solid's boundary is connected on each edge (perhaps through other faces) to another face not coplanar with it. Through the use of edge-induced endpoints, the necessary partitioning of coplanar faces is effected automatically by the transversal faces attached to the coplanar contours. Our algorithm simply records coplanar face pairs on a list so that their contours can be combined when partitioning is complete. Coplanar faces must also be known for contour classification in solid modeling.

Solid modeling

A regularized solid modeling (CSG) operation on boundary representations of two (or more) solids is a set of contours that jointly bound the desired combination. To determine which of the original or cut contours are present in a solid modeling result, each contour of a boundary must be classified as inside or outside the other solid(s). Cutting returns this information by labeling contours. However, contours in a given boundary may remain uncut, so some method is needed to classify them with respect to the other boundary (or boundaries).

Propagation of membership information

The first step propagates inclusion information from classified contours to unclassified contours across shared edges.[20] Propagation begins by filling in a field for each use of an edge within a contour to point back at the contour in which it is used. The edges are then used to locate contours connected to a classified contour. The propagation takes the following form:

```
for each face do
    for each contour do
        if contour is classified do
            propagateMark(contour);

procedure propagateMark(contour)
    for each edge in contour do
        if contour on other side of edge is
            unmarked, then
        begin
            mark other contour;
            propagateMark(other contour);
        end
end propagateMark
```

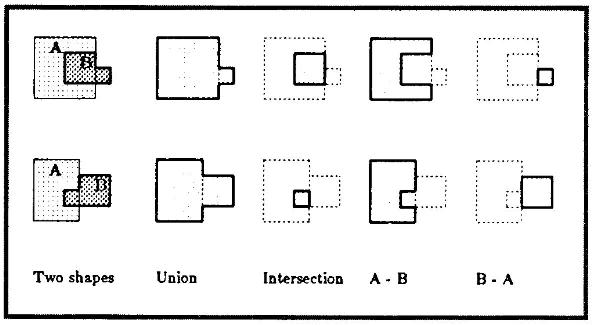


Figure 17. Boolean set operations with coplanar contours. Each edge represents a face seen edge on.

Table 1. Default case selection for coplanar contours in solid modeling.

CSG Operation	What to do with matching coplanar contours when normal orientation is: Same	Opposite
$A \cup B$	keep one	keep neither
$A \cap B$	keep one	keep neither
$A - B$	keep neither	keep one

Note that the propagation does not alter information about contours already classified.

Propagation lessens the number of unclassified contours, but some contours may not be connected to contours that were cut (i.e., the enclosed solid may have holes). After propagation, the face list is traversed once more to find all unclassified contours. For each contour remaining unclassified, a vertex on it is picked, and a point-in-polyhedron test[21] is used to classify it with respect to the transversal solid.

Coplanar contours

There is one final case in which the various attempts at contour classification fail: Two contours from distinct solids' boundaries may coincide after partitioning (except possibly for their orientation) as a consequence of coplanar faces. Because the faces have already been partitioned, regions enclosed by coplanar contours are either identical or disjoint. The algorithm must locate any matching contour pairs so that the correct one(s) can be eliminated from the CSG result. (Examples of operations on objects with coplanar faces are shown in Figure 26 below.)

The relative orientation of a pair of coinciding contours is either equal or opposite (see Figure 17), depending on the sign of the dot product of the associated face normals. The requested CSG operation then determines whether one or neither of the coinciding contours is kept in the output (see Table 1).

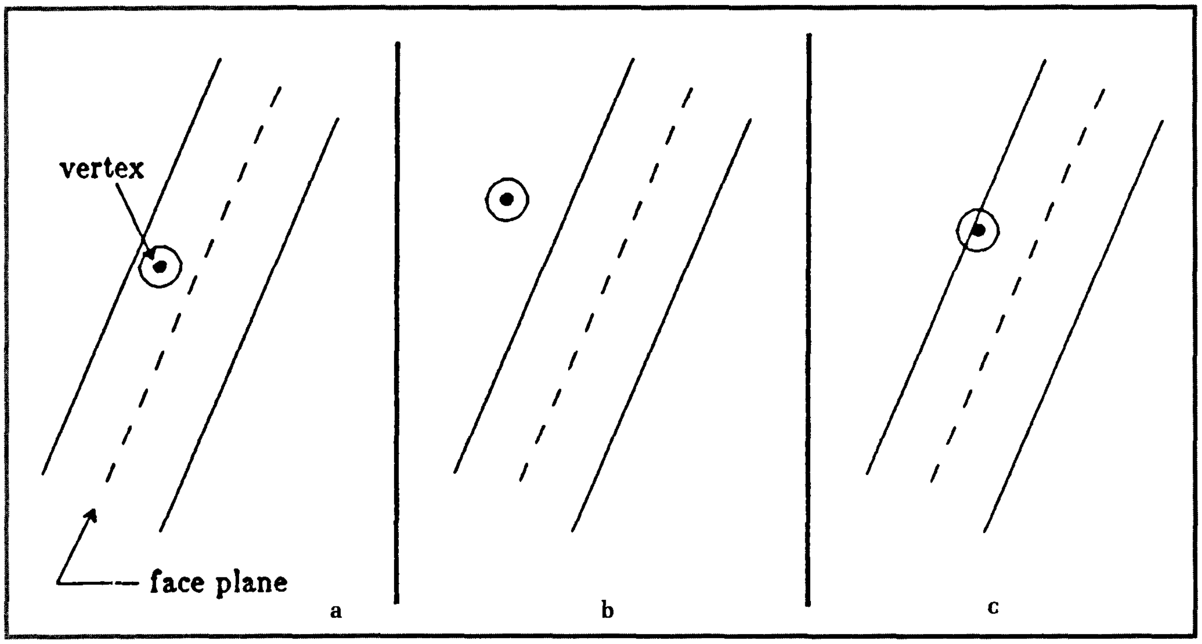


Figure 18. The three vertex-face configurations: (a) vertex in face, (b) vertex not in face, (c) ambiguous.

After cutting and propagation, the list of coplanar faces (if non-null) is traversed. Each contour of one of these faces is tested for a match in the other by examining the face list of the first edge in the contour. This list records all uses of the edge in any contour in which it is included. If two contours from distinct coplanar faces share an edge, and the enclosed regions of each contour lie on the same side of the edge, the contours must match. Coplanar contours with no matching parts are not affected; these will have been given correct tags during propagation.

Numerical accuracy

Partitioning intersecting faces requires changing the topology of the input: Edges and vertices are created; contours are split and merged. The algorithm relies on numerical data to indicate which operations are required to induce these changes. Because topological information is discrete, its derivation from numerical data may be sensitive to small perturbations in the input.

To appreciate the problem, consider two congruent cubes with their centers aligned, but with one cube rotated by some angle $0 \leq \alpha \leq \pi$ about its center with respect to the other cube, and suppose the algorithm is used to compute the intersection of these cubes.[9] For very small values of α we would expect the output to be a cube. For larger values of α a variety of other objects is possible.

There must be some value of α at which a transition occurs. For α near this value, some of the algorithm's topological determinations will be subject to small perturbations of the input. In general, some decisions may fall on the side of small α, while others fall on the side of large α. Therefore, an algorithm making only local determinations cannot guarantee a globally consistent result.

To avoid such difficulties, the necessary topological operations must be derived from well-conditioned numerical computations. As the above example shows, there may be situations in which this is not possible. A reliable modeling system must flag such situations to notify the user that ambiguous topology may be produced.

We achieve this goal by assigning a tolerance to each vertex and face reflecting the accuracy of the coordinates that describe its position. For a vertex, this tolerance defines a sphere called the tolerance region, centered on the vertex. The sphere is large enough to contain the actual (but uncertain) vertex location. The tolerance region for an edge is a truncated cone derived from the tolerances of its end vertices. For a face, the tolerance gives the half-width of an infinite slab centered about the face's plane. These tolerances are typically 100 to 1000 times greater than the machine round-off error, so they will not be affected by reasonably well conditioned computations.[22] In turn, we expect a vertex or face tolerance to be 100 to 1000 times smaller than the smallest geometrical features (i.e., vertex separation) in an individual boundary description.

At the start of the program the initial tolerance for any vertex read from the input description is set to a small arbitrary value. Face tolerances are set so that each face's tolerance region contains the tolerance regions of all its vertices. In addition, a maximum vertex tolerance and face tolerance are also specified, giving bounds on how large a vertex's or face's tolerance region may grow during the algorithm's operation.

The fundamental topological datum computed by the intersection algorithm is the position of a vertex relative to a face's plane. A vertex lies in the face's plane if and only if its tolerance region lies entirely within that of the face (see Figure 18a). If their tolerance regions are entirely disjoint, the vertex clearly lies to one side or the other of the plane (Figure 18b). There is a third possibility: The tolerance regions may partly overlap (Figure 18c). Because a small perturbation could transform this situation into either of the others, this possibility is ill conditioned. Therefore, according to the position of the center of the vertex's tolerance sphere, an arbitrary decision is made as to whether the vertex should be included in the face. A tolerance error is signaled to notify the user of the difficulty.

A new vertex, created when an edge penetrates a face, is placed on the ideal center plane of the slab representing a face. The tolerance regions of the penetrating edge intersect this plane in an ellipse; the new vertex is assigned a tolerance large enough to contain this ellipse. If the edge and face intersect with a small angle, the tolerance of the new vertex may be larger than either the preset maximum vertex tolerance or the face tolerance (see Figure 19); in either case, a tolerance error is signaled before the process continues.

Two vertices (from distinct boundaries) may come to lie within a tolerance of one another. Since the situation occurs only on or near the intersection line of two faces, such vertex pairs are detected during processing of the intersection line, and the two vertices are then merged into a single vertex with an appropriate larger tolerance. If this tolerance exceeds the maximum vertex tolerance, a tolerance error is signaled.

Similarly, faces determined to be coplanar must be merged. The merging of several faces may create a new face whose tolerance region is too large. The maximum face tolerance value is used to signal such cases as tolerance errors.

Returning to the example of two cubes, by adjusting the values of the input and maximum tolerances, we can adjust the point where the critical value of α occurs. Near α, the output may be incorrect, but an error will have been signaled. On either side of a zone around this value, the algorithm's output will be correct (see Figure 20). Whether an algorithm can be found that always produces consistent and reasonable boundaries, even for such borderline cases, is a topic for future research.

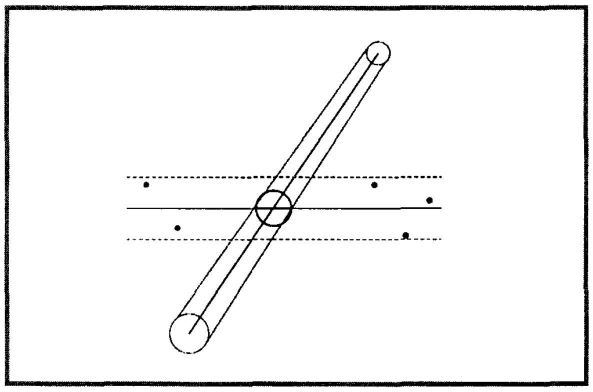


Figure 19. The intersection of an edge tolerance region with a face tolerance region, defining a new vertex's tolerance region.

Summary and results

We have described an algorithm for partitioning intersecting polygonal faces of arbitrary complexity into nonintersecting parts. With some simple extensions, the algorithm can compute the boundary representation of a CSG operation on two or more solids, each specified as a boundary representation. It is therefore useful as a solid modeling tool.

The algorithm has been implemented in about 2600 lines of C code. These are used with about 5000 lines of previously existing subroutines designed to manipulate the UNIGRAFIX[23] data structures and their ASCII descriptions. The program takes approximately 200 CPU seconds on a VAX 11/750 for objects in which about 10,000 face pairs must be checked for possible interaction. The execution time grows linearly with the number of face pairs, so hierarchical organization is essential for complicated scenes. Most of the time is spent computing distances of vertices to faces. Some results of intersection removal and solid modeling operations are shown in Figures 21 through 26. All examples were run on a VAX 11/750.

Special care has been taken in the design of the algorithm to ameliorate the effects of inconsistencies arising from inaccuracy in the input specification and to detect those arising from geometrical uncertainties. The algorithm handles designed or incidental alignments between two objects by using tolerances to classify pairs

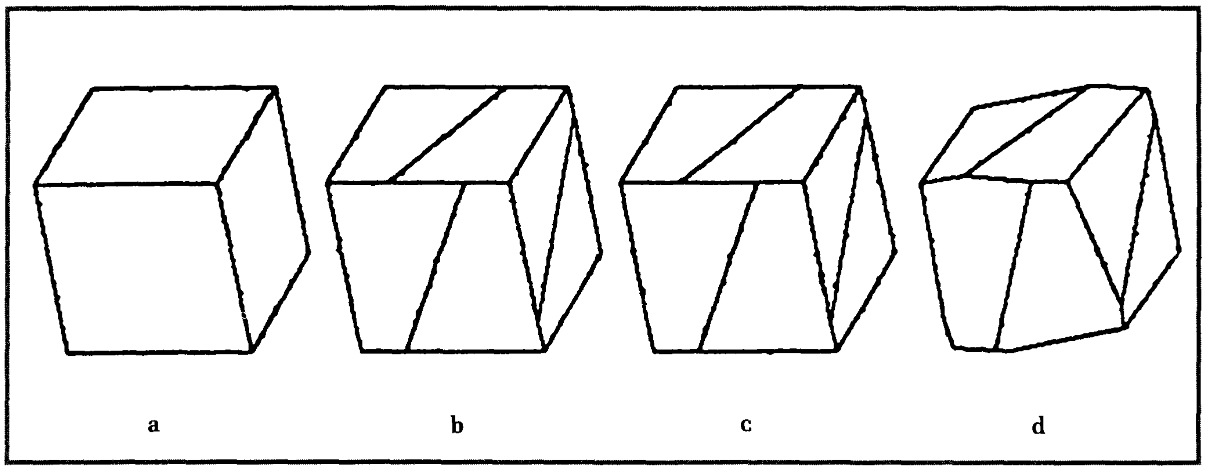


Figure 20. Topological ambiguity. Each scene is the intersection of two unit cubes, one rotated slightly from the other. In the first scene (a), the angle is 0.0001° and the algorithm produces a single cube as a result. The third scene (c) shows an angle of 0.05°. The second scene (b) shows an angle of 0.001°; with the nominal tolerances, the algorithm produced tolerance errors and failed to produce the boundary of an object as output. The tolerances were decreased to produce the scene pictured here. For comparison, the fourth scene (d) shows an angle of 15°. (The nominal tolerances are as follows: vertex tolerance = 10^{-9}, face tolerance = 10^{-5}, maximum vertex tolerance = 10^{-6}, maximum face tolerance = 2×10^{-4}.)

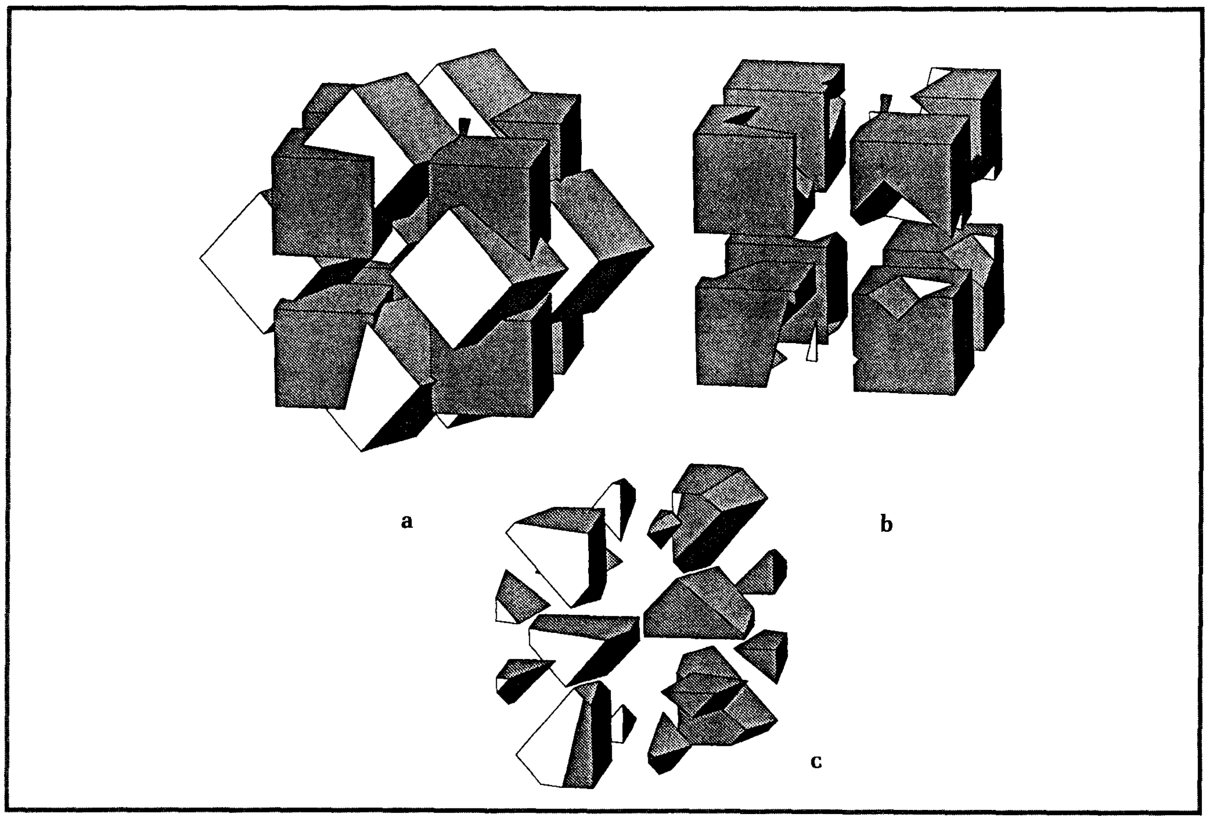


Figure 21. These images show the algorithm's operation on simple hierarchical objects. Two objects consisting of eight distinct cubes were superimposed, and the union (a), difference (b), and intersection (c) simultaneously outputted. From a total of 735 pairs considered, 114 face pairs were partitioned; total CPU time on a VAX 11/750 was 28 seconds.

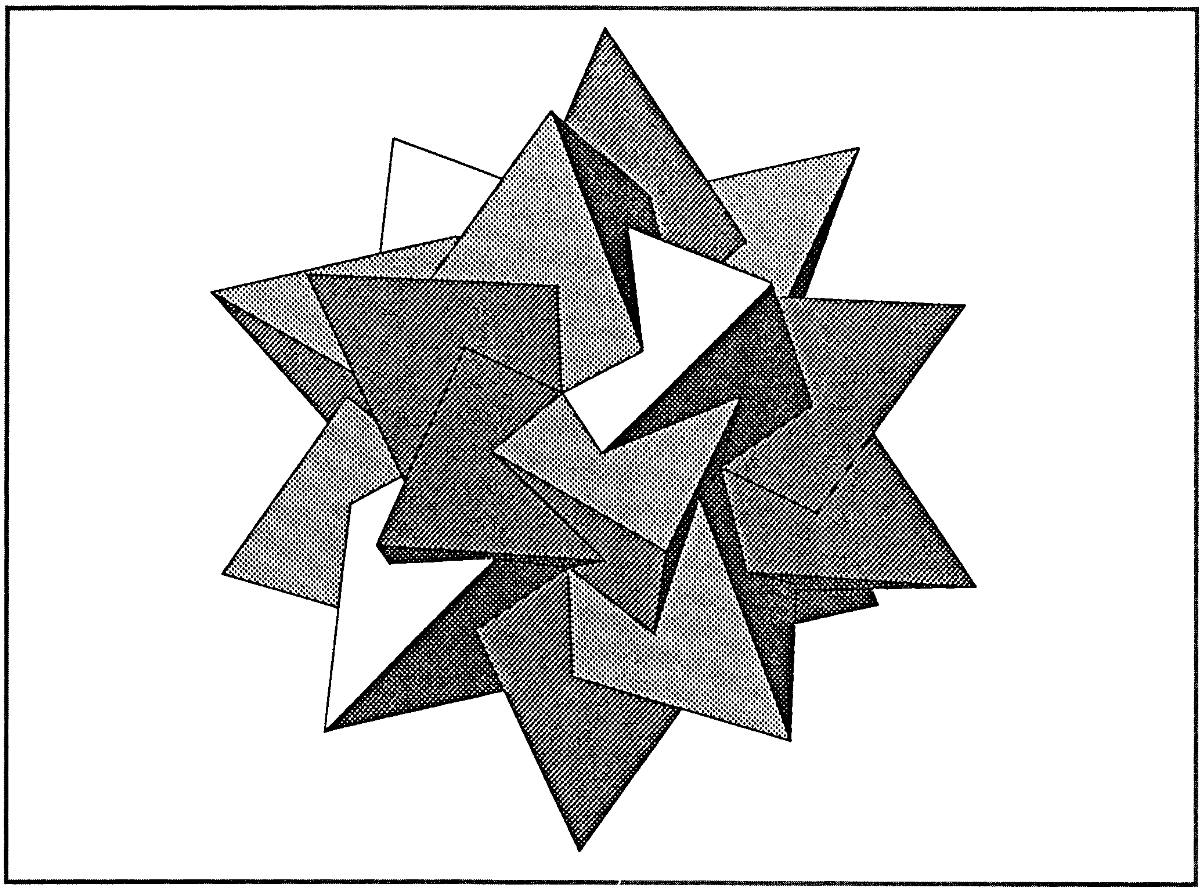


Figure 22. Union of five interlocking tetrahedrons. The algorithm also correctly produces an icosahedron (with no duplicated vertices) for the intersection of these tetrahedrons.

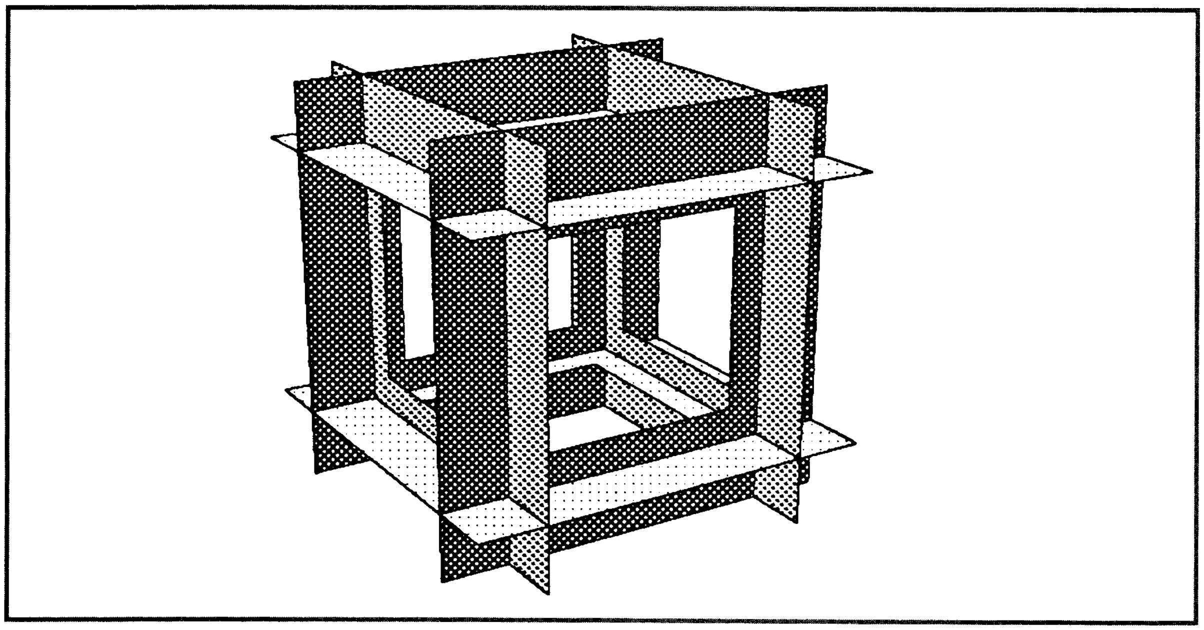


Figure 23. An example illustrating the algorithm's operation on disconnected faces.

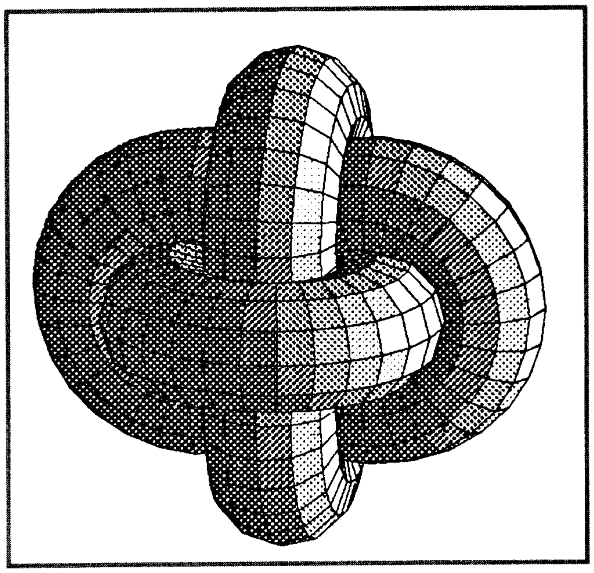


Figure 24. A larger example with 3 x 40 x 12 faces. Here 836,352 pairs were examined, but 530,912 were rejected on the basis of bounding box tests. This approach speeds up running time by about 30 percent. Only 504 face pairs (0.1 percent) were actually intersecting. This produced 1072 cuts and 520 new vertices. Total running time was 2034 seconds.

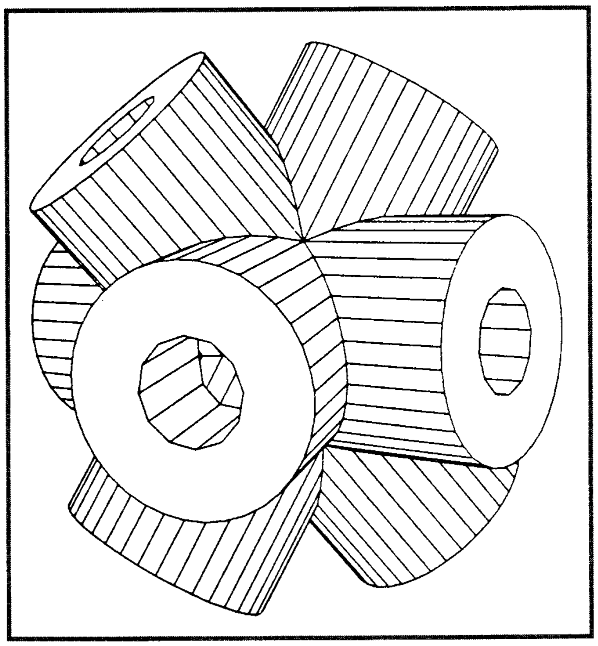


Figure 25. This example represents 15,000 face pairs, and none can be eliminated using bounding boxes. Of the face pairs, 1056 were actually intersecting, producing 2176 cuts and 892 new vertices. The running time was 487 seconds. For comparison, a scanline renderer designed to accommodate intersecting faces produced a display in 251 seconds on the unpartitioned version.

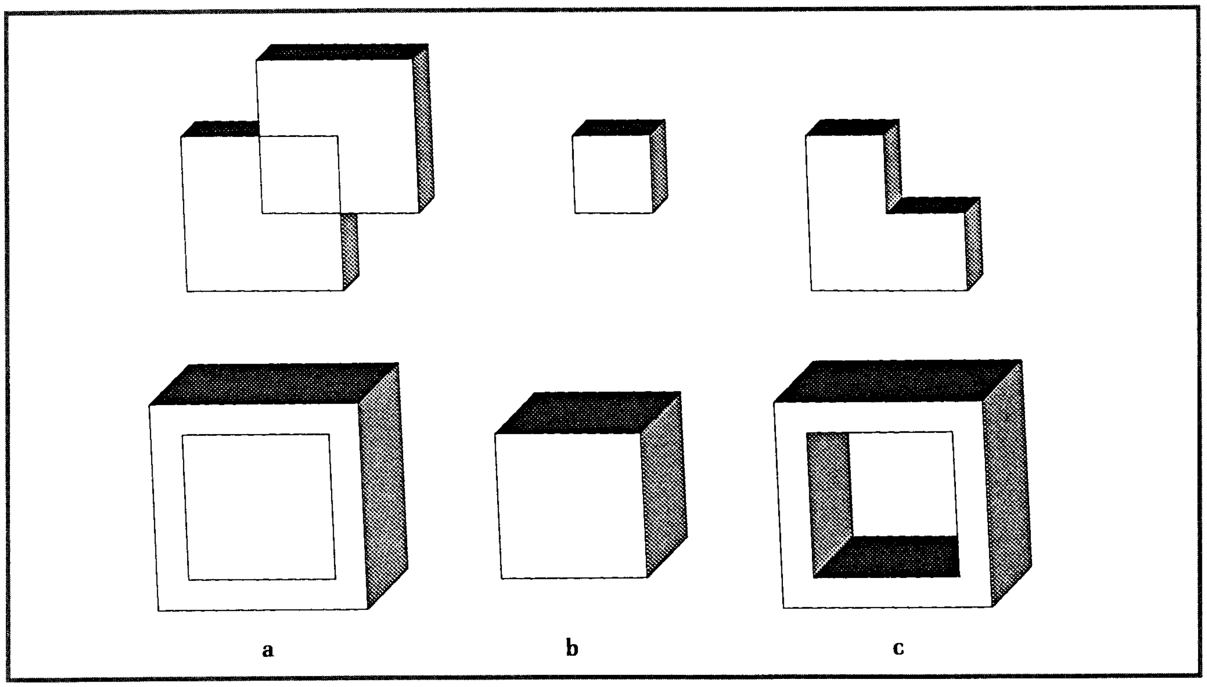


Figure 26. Solid modeling with coplanar faces: (a) union, (b) intersection, (c) difference.

of vertices, edges, or faces from distinct objects as disjoint or coincident. However, the use of tolerances cannot resolve all possible ambiguities. In some cases, the user must provide more sophisticated procedures or explicit data to guarantee a consistent output description. Our algorithm detects regions in which inconsistencies may arise and alerts the user. ■

Acknowledgment

This work was supported in part by the Semiconductor Research Corporation and Tektronix, Inc.

References

1. P.R. Atherton, "A Scan-Line Hidden Surface Removal Procedure for Constructive Solid Geometry," *Computer Graphics* (Proc. SIGGRAPH 83), July 1983, pp. 73-82.
2. S.D. Roth, "Ray Casting for Modeling Solids," *Computer Graphics and Image Processing,* Feb. 1982, pp. 109-144.
3. G. Hamlin and C.W. Gear, "Raster-Scan Hidden Surface Algorithm Techniques," *Computer Graphics* (Proc. SIGGRAPH 77), July 1977, pp. 206-213.
4. C.H. Sequin and P.R. Wensley, "Visible Feature Return at Object Resolution," *CG&A,* May 1985, pp. 37-50.
5. Y.T. Lee and A.A.G. Requicha, "Algorithms for Computing the Volume and Other Integral Properties of Solid Objects," *Comm. ACM,* Sept. 1982, pp. 635-650.
6. S. Lien and J.T. Kajiya, "A Symbolic Method for Calculating the Integral Properties of Arbitrary Nonconvex Polyhedra," *CG&A,* Oct. 1984, pp. 35-41.
7. M. Mantyla, "Boolean Operations of 2-Manifolds Through Vertex Neighborhood Classification," *ACM Trans. Graphics,* Jan. 1986, pp. 1-29.
8. F. Yamaguchi and T. Tokieda, "A Solid Modeler with a 4×4 Determinant Processor," *CG&A,* Apr. 1985, pp. 51-59.
9. D.H. Laidlaw, W.B. Trumbore, and J.F. Hughes, "Constructive Solid Geometry for Polyhedral Objects," *Computer Graphics* (Proc. SIGGRAPH 86), Aug. 1986, pp. 161-170.
10. S.M. Rubin and J.T. Whitted, "A 3-Dimensional Representation for Fast Rendering of Complex Scenes," *Computer Graphics* (Proc. SIGGRAPH 80), July 1980, pp. 110-116.
11. K. Maruyama, "A Procedure to Determine Intersections Between Polyhedral Objects," *Int'l J. Computer Information Science,* Sept. 1972, pp. 255-265.
12. M. Segal, "Consistent Calculations for Solid Modeling," *Proc. First Symp. Computational Geometry,* ACM, New York, 1985, pp. 29-38.
13. W.R. Franklin, "Efficient Polyhedron Intersection and Union," *Proc. Graphics Interface 82,* Canadian Information Processing Soc., Toronto, 1982, pp. 73-80.
14. B.G. Baumgart, "Geometric Modelling for Computer Vision," Tech. Report STAN-CS-74-463, Stanford Univ. AI Lab, Stanford, Calif., 1974.
15. V. Guilleman and A. Pollack, *Differential Topology,* Prentice-Hall, Englewood Cliffs, N.J., 1974.
16. R.B. Tilove, "Set Membership Classification: A Unified Approach to Geometric Intersection Problems," *IEEE Trans. Computers,* Oct. 1980, pp. 874-883.
17. C.M. Eastman and K. Weiler, "Geometric Modelling Using the Euler Operators," *Proc. First Ann. Conf. Computer Graphics in CAD/CAM Systems,* MIT Press, Cambridge, Mass., 1979, pp. 248-254.
18. W.M. Newman and R.F. Sproull, *Principles of Interactive Computer Graphics,* McGraw-Hill, New York, 1979.
19. K.J. Weiler, "Polygon Comparison Using a Graph Representation," *Computer Graphics* (Proc. SIGGRAPH 80), July 1980, pp. 10-18.
20. S.W. Thomas, *Modelling Volumes Bounded by B-Spline Surfaces,* doctoral dissertation, Univ. of Utah, Salt Lake City, 1984.
21. J. Lane, B. Magedon, and M. Rarick, "An Efficient Point in Polyhedron Algorithm," *Computer Vision, Graphics and Image Processing,* Apr. 1984, pp. 118-125.
22. G.H. Golub and C.F. Van Loan, *Matrix Computations,* Johns Hopkins Press, Baltimore, 1983.
23. C.H. Sequin, M. Segal, and P. Wensley, "UNIGRAFIX 2.0 User's Manual and Tutorial," tech. report, Computer Science Division, Univ. of California, Berkeley, 1983.

Mark Segal is a doctoral student in computer science at the University of California, Berkeley. His research interests include geometric modeling and computational geometry. He received a BA in applied mathematics from Harvard University in 1982 and an MS in computer science from UC Berkeley in 1985. Segal is a member of IEEE and ACM.

Carlo H. Sequin is a professor of computer science at the University of California, Berkeley, and has been a member of the faculty since 1977. His research interests are in computer graphics, solids modeling, and CAD tools for mechanical and electrical design.

Sequin has worked at the Institute of Applied Physics in Basel, Switzerland, where he was concerned with interface physics of MOS transistors and problems of applied electronics in the field of cybernetic models. From 1970 to 1976 he worked at Bell Telephone Laboratories, Murray Hill, N.J., on the design and investigation of charge-coupled devices for imaging and signal processing applications.

Sequin received his PhD in experimental physics from the University of Basel in 1969. He is a member of ACM and a fellow of IEEE.

The authors can be contacted at the Department of Electrical Engineering, Computer Science Division, University of California, Berkeley, CA 94720.