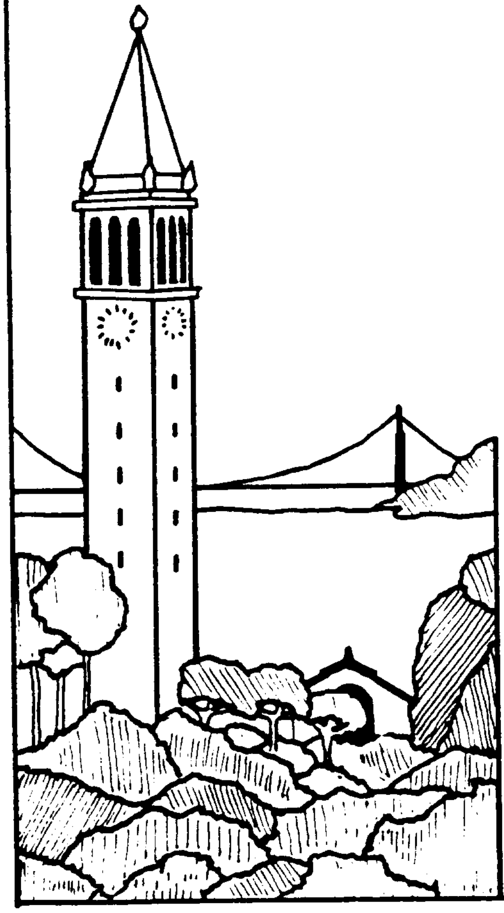
Partitioning Polyhedral Objects into Non-Intersecting Parts

Mark Segal



Report No. UCB/CSD 86/284

February 1986

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Partitioning Polyhedral Objects into Non-Intersecting Parts

Mark Segal

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California
Berkeley, CA 94720

ABSTRACT

An algorithm for partitioning intersecting polygons embedded in three-space into disjoint parts is described. Polygons, or *faces*, need not be convex and may contain multiple holes. Pairwise intersections between faces are removed by slicing faces apart along their regions of intersection. To help reduce the number of face pairs examined, bounding boxes are found for objects consisting of groups of faces, and these boxes are checked for overlap.

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

Tolerances are calculated for computed vertices, edges and faces and are used to locate regions in which numerical inaccuracy may lead to erroneous results. Various heuristics overcome most such situations, but some require further information from the user.

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

1. INTRODUCTION

Polyhedra in three dimensions are often represented by specifying their boundaries as a set of polygonal faces. Each face, in turn, is defined by a set of bounding edges whose endpoints are specified by vertices. We sometimes allow more general groups of polygons that do not necessarily form polyhedron boundaries. In any case, a specification may allow faces to intersect one another.

We wish to determine and remove these intersections for two reasons. First, many rendering algorithms that exploit object coherence for efficiency cannot tolerate intersecting faces in the geometric descriptions on which they operate.[1,2] Renderers that do accept intersecting faces do so at added computational expense.[3]

Second, solids modeling systems usually allow the specification of complex solids as the intersection, union, or difference of simpler objects. While some renderers (notably ray-tracers) are capable of displaying such combinations directly,[4,5] those that cannot require a simple list of faces. Further, if one wishes to compute physical properties of an object (center of mass, moments of inertia, etc.), it is simplest to work with an explicit description of the object's boundary.[6,7] Finally, we may wish to determine a topological (coordinate-free) property of an object. A ray-tracer, while capable of displaying a complex boolean operation result, does not reveal the topological structure of the displayed object.

The algorithm we present partitions arbitrary polygons embedded in three-space into non-intersecting parts. If the polygons represent boundaries of solids, the algorithm produces the results of set-theoretic operations on those solids as a byproduct of its operation. The approach is simple and general, operating on faces directly without first breaking them into simpler pieces.[8] The method includes provisions for detecting and possibly overcoming the effects of numerical error on the resulting faces and boundaries.

1.1. Overview of the Algorithm

The algorithm operates by looking for and, if found, removing intersections from 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 the component objects. Bounding boxes are used to determine when a pair of collections within an object interfere.[9] If they do interfere, each collection is expanded into a list of the objects comprising it. The items in the lists are individually checked and processed for intersections, if necessary. Then, each object from one list is checked against each object from the other, removing any pairwise intersections. This recursive process is applied to the object representing the total input to the algorithm, eventually removing all intersections.

The lowest level of the recursion is to detect and remove intersections from a particular pair of faces. The intersection of a pair of faces may consist of either a set of line segments (for transversal faces) or a set of planar regions (for faces that are coplanar), or be empty.

A face pair is first passed to a procedure that determines if at least one edge of each face crosses the other's plane. If this occurs, all intersection points of each face's edges with the other's plane are found and then sorted into order along the intersection line. This allows determination of the intersection segments along which each face is topologically partitioned.[10] Sorting the intersection points also allows finding vertices that are near enough to one another that they should be considered a single vertex.

If all of one face's edges lie in the other's plane, the faces are coplanar,[11] requiring, for generality, the use of a procedure to partition overlapping regions in the plane. However, our algorithm takes advantage of connectedness among a solid's boundary faces. Partitioning of coplanar faces, when both faces belong to solid boundaries, is achieved with no extra work.

If the intersection removal is being done to compute the boundary of a set-theoretic operation on two or more solids, membership information is gathered during face partitioning. After all intersections have been removed, this information is used to determine which face portions belong in the desired result and which do not. The appropriate portions are simply thrown away, and the result output. The results of several distinct boolean operations can thus be output simultaneously.[12]

An important feature of the algorithm is its immunity to moderate numerical inaccuracies in the input description. Methods are employed to overcome difficulties arising from "warped" faces and grazing intersections. These measures, in the presence of some restrictions on the solids' interactions, insure that the result of a boolean operation applied to the boundary representations of two or more solids is the boundary representation of another solid.

2. GEOMETRIC ITEMS AND DATA STRUCTURES

2.1. Geometric Items

There are four basic geometric items with which the algorithm is concerned: vertices, edges, contours and faces. A *vertex* includes a point in space specified by three coordinates. An *edge* joins a pair of vertices defining its endpoints; each edge has a *starting vertex* and an *ending vertex*. A *contour* represents a single, closed 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.

A *face* specifies a set of finite two-dimensional *patches* in the same plane; contours specify non-intersecting boundaries of the planar patches. The face also specifies a normal and a distance from the origin defining 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.[13] The contour is said to *enclose* one of these areas. If the ordering of edges in a contour defines a clockwise traversal with the face normal pointing up, then the contour encloses its inside; otherwise, it encloses its outside. A contour enclosing its outside is called a *hole*.

Each contour must bound an open planar set so that local topological determinations are possible.[14] This restriction insures that a contour may not contain zero-width "spikes" (although zero-width "notches" are allowed).

Finally, the algorithm may consider a fifth type of object: a *solid*. A flag may be attached to any face collection indicating that it forms the boundary of a solid in $\mathbf{R}^3$. Set-theoretic operations may be specified on such collections. The normals to the boundary faces point away from the solid's interior, inducing a boundary orientation. The orientation (face normal) is used to distinguish inside from outside for solid modeling operations.

No checks are made to insure that faces in the collection do not intersect and are connected in such a way as to define an oriented polyhedral boundary.[15] It is up to the calling procedure to make sure that any face collection so used is correctly defined.

2.2. Data structures

The data structures that the algorithm builds from an ASCII description of vertices, contours, and faces have been described in detail elsewhere.[1] Briefly, the essential structures are the following:

2.2.1. Vertices

A vertex's main feature is the triple of floating point numbers describing its coordinates. A tolerance indicates the accuracy of the coordinates. In addition, there is a pointer to a list of edges to which the vertex is connected, and a pointer to the vertex's ASCII name.

2.2.2. Edges

Each edge has two pointers to the vertices that define it. In addition, there is a pointer to a list that indicates in which faces the edge is used and the location within a particular contour of that use.[16]

2.2.3. Contours

Each contour is a list of pointers to edges. Each list item also contains a flag showing whether the edge is used in the defined direction or reversed direction. The list elements are circularly linked, facilitating the splitting and merging of contours. There is also a field for flags that are used to assign contours as "inside" or "outside" one or more solids. These flags are set during cutting and are used to determine which contours should be present in the boundary of the result of a solid modeling operation.

2.2.4. Faces

A face consists of a list of contours and four real numbers describing its plane equation. As with a vertex's coordinates, a tolerance indicates the accuracy of the plane equation's coefficients. If appropriate, each face also points to the solid to which it belongs. For a face assigned an ASCII name, there is a non-null pointer to the name string.

2.2.5. Solids

A solid is a list of faces defining its boundary and a bounding box.

3. TOLERANCES

We associate tolerances with the positions of vertices and the equations of planes to control the effects of limited precision in the input description and the algorithm's calculations. For example, in a face with more than three edges, numerical inaccuracies in the edges' construction and positioning may cause them to deviate from the plane in which they should lie. Since a face's edges define its plane, we must assume that faces may not be precisely specified; the plane equation is subject to some error which we record as a face tolerance.[11]

Further, during its operation, the algorithm must decide whether certain face pairs are transversal or coplanar. Because of the face tolerances, as well as possible roundoff errors in positioning two distinct objects relative to each other, exact equality of plane normals is an inappropriate test for coplanar faces. Similarly, some vertices may be close enough to be considered copositional.

These difficulties arise when a vertex, edge, or face from one object lies close to a vertex, edge, or face in another object. Without some provision to overcome them, running the algorithm on a pair of faces could produce spikes or other illegal contour constructs. A solid modeling operation might produce output that was not the boundary of a solid in $\mathbf{R}^3$.

To resolve any ambiguities in determining what constitutes close, we introduce a tolerance model. If the input objects adhere to this model, and we make some assumptions about the way distinct objects interact, we can be sure that the algorithm's output will adhere to the model as well. Further, the algorithm can determine as it operates if the assumptions are invalid, halting its operation or notifying the user when it recognizes a region in which it may not produce correct output.

The model is vertex based. Each input vertex has or is assigned a tolerance, ϵ_v, that specifies how much in error the least accurate coordinate may be. Thus, a vertex's tolerance is a half-width that defines a small cube, or tolerance *region* in which the actual vertex is known to lie. Each of the input object's edges must be much longer than any vertex tolerance, so that every edge's direction is well-defined.

Each face's tolerance is constructed from its vertices' coordinates and tolerances. A face tolerance, ϵ_f, also defines a region around the plane specified by the face's plane equation. When intersection removal begins, the tolerance must be large enough so that a face's tolerance region contains the tolerance regions of all vertices that lie in the face.

As the algorithm proceeds, new vertices appear at intersections of edges with a face. A new vertex's tolerance is the maximum of three quantities: the tolerances of the edge's two defining vertices, and half the face's tolerance divided by the sine of the angle made by the edge with the face's plane.* This last number measures the half-length of the edge's intersection with the face's

*Since we assume vertex separation large, we assume the end tolerances define the edge tolerance region. The creation of an edge whose length is comparable to the tolerances of its defining vertices is considered an error.

tolerance region. Because the face's plane equation is specified with a tolerance, the computed vertex can lie anywhere along the transversal edge inside the face's tolerance region.

After the edge intersection points for a pair of faces have been sorted along the intersection line, vertices are merged if their respective tolerance regions overlap. A tolerance is created for the vertex resulting from the merge by adding together the tolerances of the component vertices and subtracting any overlap. One assumption is that the tolerance of any vertex, after computation and possible merging, does not exceed a supplied maximum vertex tolerance, ϵ_V. If this assumption proves false, the algorithm cannot continue as it cannot insure that resulting face tolerance regions will contain their component vertex tolerance regions. We term such an occurrence a *tolerance error*. In case of a tolerance error, the algorithm must consult a set of default rules or ask the user for advice on how to proceed. Once such an error occurs, the algorithm cannot insure the validity of its output. If there is no source of further information about the numerically ambiguous situation, the algorithm may produce topologically viable results if run again with revised tolerance values.

During intersection processing, decisions must be made as to whether a vertex lies within a face's tolerance region. The face tolerance may grow as new vertices are added, so a maximum value must be known so that consistent decisions are possible. As for vertices, a maximum face tolerance, ϵ_F, must be specified. This value must be at least as large as the largest face tolerance before intersection processing plus ϵ_V. This insures that each face's tolerance can be made large enough to include any vertices created as part of the face. ϵ_F is used in determining when a vertex from one object lies in a face of another object, while ϵ_f is used in computing new vertex tolerances.

This arrangement implies that a vertex in one object be considered to lie in a face of a second object if the tolerance region specified by ϵ_V about the vertex lies completely within the tolerance region around the face defined by ϵ_F (Figure 1). Thus, even if the vertex's tolerance were to grow to ϵ_V, the associated tolerance region would still lie acceptably close to the face's plane. A vertex is considered *not* to lie in a face if the largest possible tolerance region about the

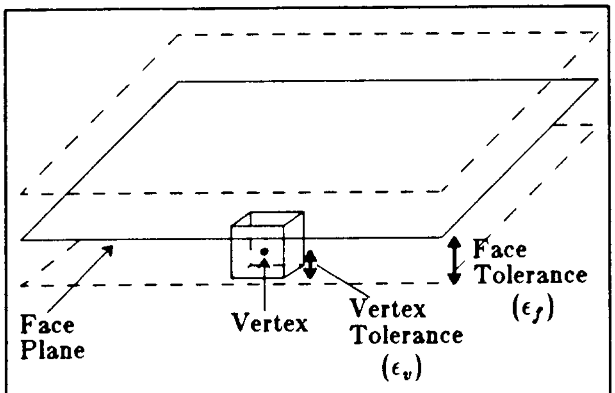


Figure 1. A vertex whose tolerance region lies within a face's tolerance region.

vertex lies *entirely outside* the face's largest possible tolerance region.

Unfortunately, some vertices may not fall into either category; their maximum tolerance regions partially overlap the maximum face tolerance region. The detection of such a vertex produces a tolerance error, indicating that the algorithm has insufficient information to proceed.

If all of one face's vertices lie in another face, the two are considered coplanar. Such face pairs are recorded, and are merged after transversal intersection processing is complete. A new plane equation is found using Newell's algorithm[17] applied to all contours of the original faces. A single face created from the merging of two faces may have a tolerance larger than either of the two single face tolerances; such a face must be re-checked for intersection against all the other faces in the input. Since coplanar faces rarely occur, the required extra work is small. Further, a merged face cannot engulf too many coplanar faces, for eventually ϵ_F will be exceeded.

Finally, two faces that intersect only grazingly may make it impossible to accurately compute an intersection line. This may occur in spite of some vertices from each face lying far enough from the other's tolerance region so that the faces are not considered coplanar. Such a face pair would introduce vertices with tolerances well above any reasonable ϵ_V, once again producing a tolerance error.

The tolerance model we have adopted is a simple one. However, despite its drawbacks, we have found our simple model adequate; ambiguous cases arise only rarely. When they do arise, re-running the algorithm with slightly larger tolerances nearly always produces the desired result.

4. INTERSECTION CLASSIFICATION

The first step in removing intersections from a pair of faces is to determine a plane equation for each face. Each face fed to the algorithm normally has a plane equation attached, but if the equation is missing, one is computed using Newell's algorithm. To save computation, only the first face contour is used.

Next, every vertex belonging to a particular face has it coordinates 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.[11] Vertex tolerances, if not supplied in the input, are assigned an arbitrary small value. For a vertex not belonging to a particular face, the sign of the oriented distance from the vertex to the face's plane determines on which side of the plane the vertex lies. This determination is valid only if the vertex lies entirely outside the face's tolerance region.

Once plane equations and tolerances have been found for a pair of faces, a determination is made as to whether the faces might cross, do not cross, or are coplanar. The edges in each contour of one face are traversed and the oriented distance of each vertex from the other face is found. Traversal continues until either: (1) two vertices produce different signs (that is, they are on opposite sides of the other face), or, (2) all edges of all contours in the first face have been traversed.

If (1) occurs, the first face has vertices on both sides of the second face's plane, and 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 proceeds. The point at which the first sign change is detected is saved so that later traversal skips the portions of each face that do not cross the other's plane.

On the other hand, if (2) occurs 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 $2\epsilon_V$ of the second face's tolerance region. If the faces are coplanar, they are processed by the coplanar intersection removal routine. If not, the faces do not intersect and are returned untouched.

5. INTERSECTING TRANSVERSAL FACE PAIRS

First, we introduce some definitions to permit easy description of the regions where faces intersect. If they do so at all, two transversal face planes meet in an *intersection line*. We call the intervals that a face patch (as defined by contours) has in common with the intersection line intersection *segments*. Each segment is defined by its two *endpoints*. 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. Partitioning intervals are found by determining their endpoints.

At first, each face is treated separately. By determining the points of intersection of each contour with the intersection line, intersection segments for each face patch are determined. Next, segment information from both faces (kept as a set of endpoints) is sorted to find the cutting intervals.

Cutting operations are then applied to the regions of interaction. There are four cases (Figures 2-5). Figure 2 shows the case if Face 1's intersection segment is a proper subset of Face 2's intersection segment. In this case, Face 1 is *cut* apart along the line, while a *slit* (a hole of zero width) is inserted into Face 2. The other case arises when neither face's segment contains the other's. Figure 3 shows how both faces are *notch*ed to accommodate the intersection. The two other cases arise if segment endpoints from the two faces coincide. In Figure 4, one face must be *cut* and the other *notch*ed. In Figure 5, both faces must be *cut*. These diagrams illustrate simple cases; typically, these operations must be applied to a face more than once.

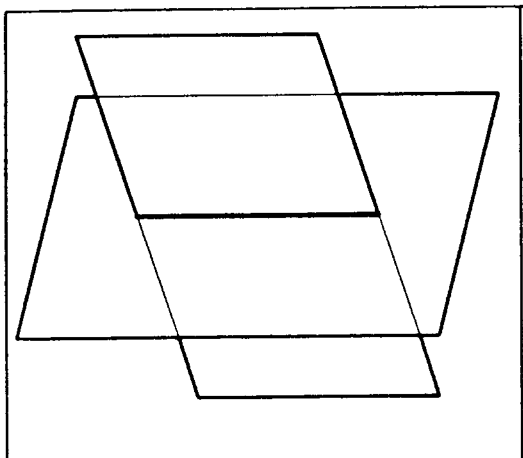


Figure 2. A *cut* with a *slit*.

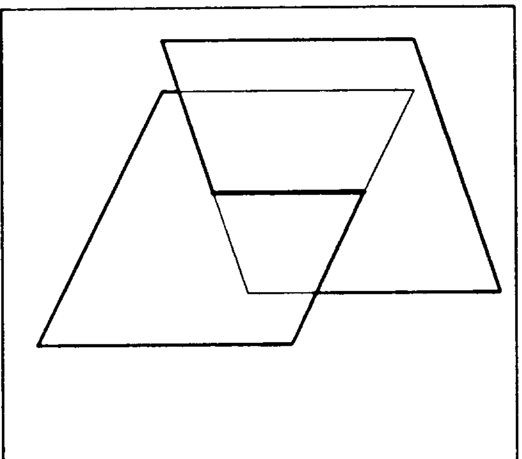


Figure 3. Two *notches*.

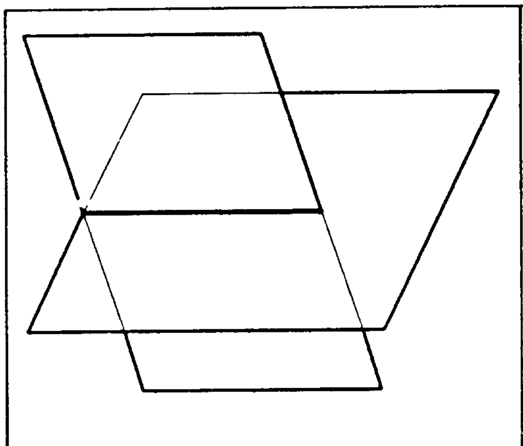


Figure 4. A *cut* with a *notch*.

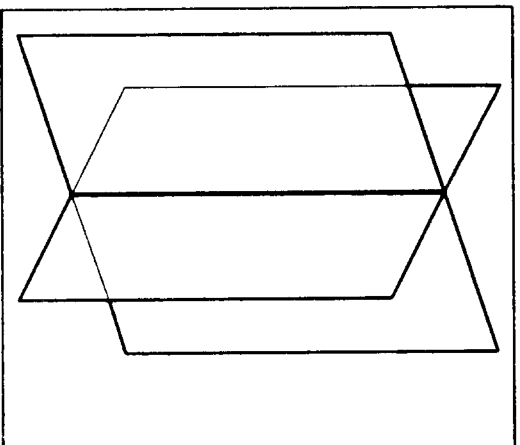


Figure 5. Two *cuts*.

5.1. Intersection Line Calculation

Since the face planes are not parallel, we can compute the intersection line. The calculation involves finding a direction vector, **u**, and a point on the line, **v**, from the two normalized plane equations:

$$a_1x + b_1y + c_1z = d_1$$

$$a_2x + b_2y + c_2z = d_2$$

Recall that the vector (a_i, b_i, c_i) is a normal to the appropriate face.

A direction vector **u** is found by computing the cross-product of the two plane normal vectors, and normalizing the result. To obtain a point on the line, we introduce a third arbitrary plane equation, constructed to have normal **u** and to pass through some vertex in one of the faces, yielding

$$\begin{bmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \end{bmatrix},$$

or,

$$\mathbf{A}\mathbf{x} = \mathbf{d}$$

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. We need only alter $\mathbf{r}_1$ so that it is orthogonal to $\mathbf{r}_2$. We apply the elementary row operation

$$\mathbf{R}_1 \leftarrow \mathbf{R}_1 - (\mathbf{r}_1 \cdot \mathbf{r}_2)\mathbf{R}_2$$

and renormalize

$$\mathbf{R}_1 \leftarrow \frac{1}{|\mathbf{r}_1{}'|}\mathbf{R}_1$$

with $\mathbf{R}_i = (a_i, b_i, c_i, d_i)$. $\mathbf{r}_1{}'$ is a three-vector whose components are the first three components of $\mathbf{R}_1$ after the first row operation. The resulting system is orthogonal, and we obtain

$$\mathbf{v} = \mathrm{A}'^T \mathbf{d}',$$

where the primed quantities represent the results of the orthogonalizing row operations.

5.2. Endpoint Determination

Endpoints are the points of intersection of contours with the intersection line. To find these points, each edge of each contour is examined. If the endpoints of an edge belonging to one face lie on opposite sides of the transversal face's plane, then the edge penetrates the transversal face. The point of intersection is a segment endpoint, and so must be calculated.

If the vertices of the edge are given by $\mathbf{v}_1$ and $\mathbf{v}_2$, we set $\mathbf{b} = \mathbf{v}_2 - \mathbf{v}_1$. The edge is then parameterized by $\mathbf{v}_1 + s\mathbf{b}$, with $0 \leq s \leq 1$. The plane equation of the transversal face is $\mathbf{n} \cdot \mathbf{x} = d$, where $\mathbf{x}$ represents any point in the plane. We solve these two equations for s:

$\mathbf{n}\cdot(\mathbf{v}_1 + s\mathbf{b}) + d = 0$ gives $s = -\dfrac{d - \mathbf{n}\cdot\mathbf{v}_1}{\mathbf{n}\cdot\mathbf{b}}$. The desired point is then $\mathbf{p} = \mathbf{v}_1 + s\mathbf{b}$.

Actually, since the algorithm substitutes vertex coordinates into the transversal face's plane equation to determine the side on which a vertex lies, the distances from the plane are already known. That is, $d_1 = \mathbf{n}\cdot\mathbf{v}_1 + d$ and $d_2 = \mathbf{n}\cdot\mathbf{v}_2 + d$ have already been found. Since $\mathbf{n}\cdot\mathbf{b} = d_1 - d_2$, it saves work to set $s = \dfrac{d_1}{d_1 - d_2}$ (this quantity is always positive since d_1 and d_2 have opposite signs).

Intersection points are provisionally added to the data structure as new vertices. They are not permanently entered because they may not represent cutting interval endpoints. Later, sorting will determine which new vertices should become incorporated into the full data structure.

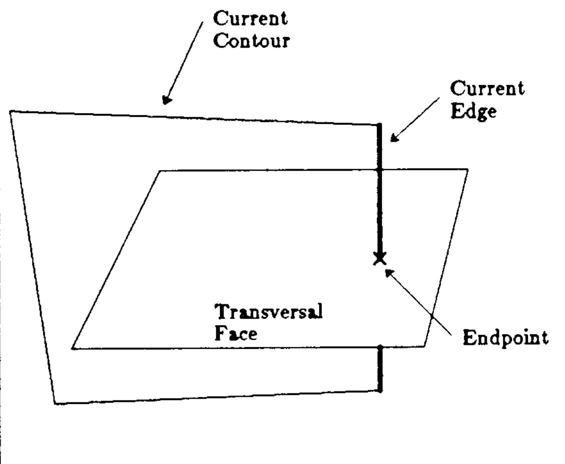


Figure 6. An endpoint occurring in the middle of an edge.

Vertices created during this step are assigned a tolerance. The tolerance value is found using the tolerances of the two end vertices and that of the intersecting plane (see section 3).

Not all segment endpoints occur inside an edge. To determine whether an already present vertex represents an endpoint, the two edges adjacent to it are checked to see if they lie in the transversal face. The check is made by computing the distance of each of the edges extreme vertices from the transversal face's plane. For each vertex, a distance of zero indicates the associated edge lies in the transversal face's plane and therefore on the intersection line.

One possibility is that both edges lie in the transversal face. If the two edges have the same direction (as used in the particular contour), then the central vertex does not define an endpoint. If they have opposite directions (as results from an occurrence of a notch or a slit), there is a single intersection segment incident on the central vertex (Figure 7a & b).

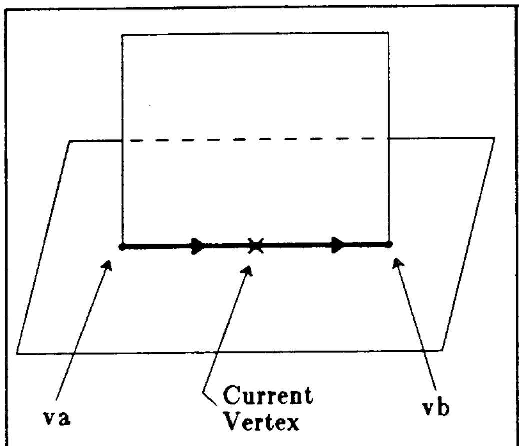


Figure 7a. The vertex at X does not represent an endpoint.

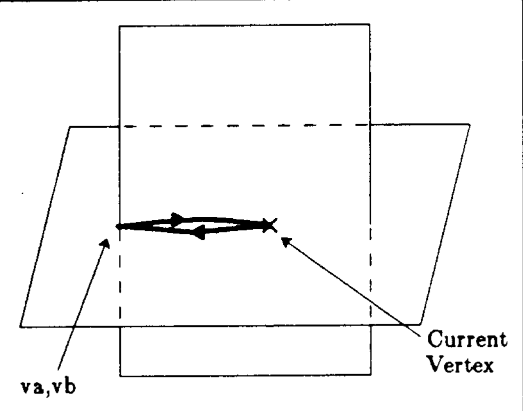


Figure 7b. The marked vertex determines an endpoint.

Another possibility is that only one of the edges connected to the central vertex lies in the transversal face. In this case, the central vertex defines an intersection point if the angle made by the contour there is greater than 180 degrees. To determine if the angle is less than or greater than 180 degrees, the cross product of the directions indicated by the two incident edges is compared to the face normal of the contour under consideration. If the cross product points in the direction opposite to the face normal, the angle is greater than 180 degrees, and the central vertex defines a single endpoint (Figures 8a & b).

A third possibility is that neither edge lies in the transversal face's plane. If the vertices lie on opposite sides of the intersection line, the central vertex defines a single endpoint. If both vertices lie on the same side of the intersection line, the central vertex either defines two endpoints or none. As in the case of a single extreme vertex on the intersection line, the cross product of the edge directions is compared to the face normal to determine if the angle formed by the contour is greater than 180 degrees. If so, a double endpoint is indicated; otherwise, the central vertex is not an endpoint (Figures 9a & 9b).

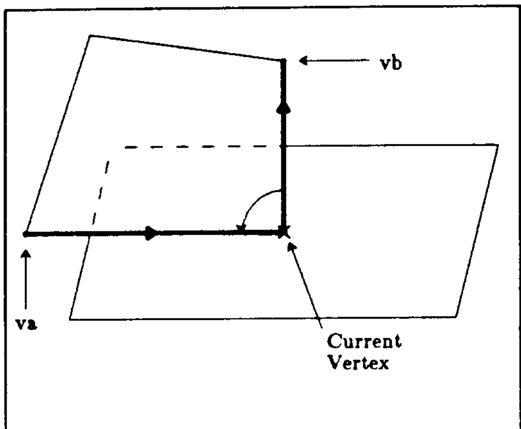


Figure 8a. One adjacent vertex in the transversal plane. The marked vertex does not define an endpoint.

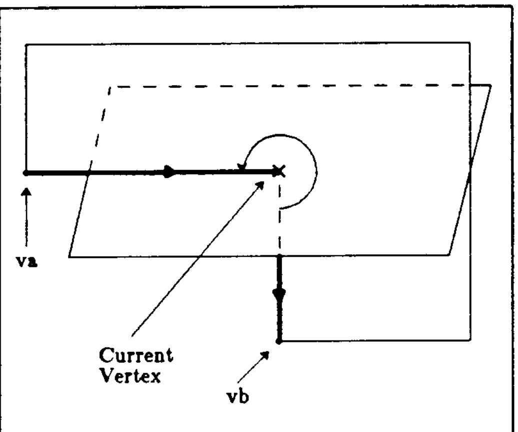


Figure 8b. The marked vertex defines an endpoint.

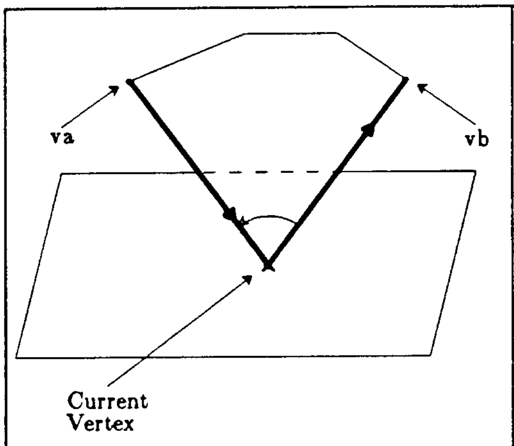


Figure 9a. Neither adjacent vertex lies in the transversal face. No endpoint is indicated.

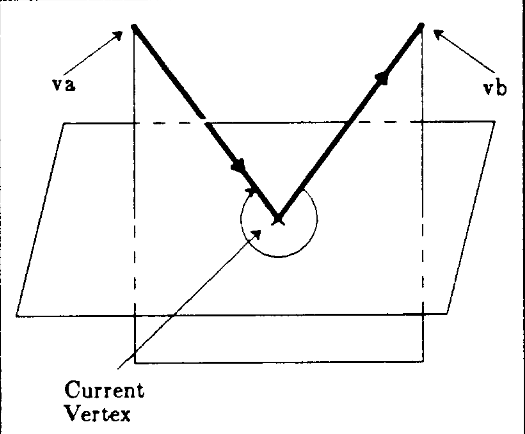


Figure 9b. Two endpoints are indicated at the marked vertex.

Even if a vertex lying in the transversal face is not an endpoint, it is placed in the endpoint list anyway. Sorting will determine if it lies within ϵ_V of a vertex in the transversal face, requiring a merge. These special vertices are essentially ignored during intersection line processing. However, each face has a (possibly empty) list of vertices that impinge on it but do not lie on any of its contours. If such a vertex is found inside the transversal face, that face's list is updated. The list can be thought of as giving the locations of "pinholes" in the face.

Whenever an endpoint is found, its relative position on the intersection line is determined for sorting. The line is parameterized by $\mathbf{v} + t\mathbf{u}$; though the endpoint should already be on the line, the form of the equation makes projection onto the line a simple means of finding t. If $\mathbf{p}$

represents the endpoint, then $t = \mathbf{u}^T(\mathbf{p}-\mathbf{v})$.

Each endpoint is classified as an *entry* or an *exit*. The decision depends on the orientation of the contour at the endpoint. If **u** points from the endpoint into the indicated face patch, then the endpoint represents an entry. If **u** points away from the indicated patch, the endpoint is classified as an exit. Two endpoints at the same vertex produce one entry and one exit.

To make the classification, let $\mathbf{e}_1$ represent the edge incident on the current vertex and $\mathbf{e}_2$ represent the edge emanating from it. If both $\mathbf{e}_1 \mathrm{x} \mathbf{u}$ and $\mathbf{u} \mathrm{x} \mathbf{e}_2$ are zero (both edges lie on the intersection line, Figure 7), the endpoint is an entry if $\mathbf{e}_1 \bullet \mathbf{u}$ is positive, and an exit if this quantity is negative. Otherwise (Figure 6 or 8), let **x** be a cross-product result which is non-zero (either $\mathbf{e}_1 \mathrm{x} \mathbf{u}$ or $\mathbf{u} \mathrm{x} \mathbf{e}_2$), and let **n** be the normal to the current face. Then the endpoint is an entry if $\mathbf{n} \bullet \mathbf{x} > 0$.

For a pair of endpoints at the same vertex (Figure 9b), one must be an entry and the other an exit. The first endpoint's classification is found as for a single endpoint, using the first incident edge in the cross product. The dot product result cannot be zero in this case.

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 in some cases during cutting to determine the resultant contour topology.

5.3. Edge Induced Endpoints

Unlike a simple set of faces, a solid's boundary is connected from contour to contour along edges. Therefore, an edge (which must lie between two contours in a solid) that lands entirely within a face must induce a partition in that face (Figure 10).

Determining endpoints as described so far would not create the desired partition in such a case. To remedy this situation, we must allow an edge lying entirely in the transversal face to determine an intersection segment in that face. When such an edge is encountered, both the current and next vertices on the contour under traversal lie in the transversal face. This arrangement may produce no endpoint at either the current or next vertices. But the edge is recognized and two *phantom* endpoints entered into the endpoint list. These special endpoints may induce a cut in the transversal face. But their phantom status is noted during intersection line traversal, and no cut is made across the indicated face (where there is an edge already).

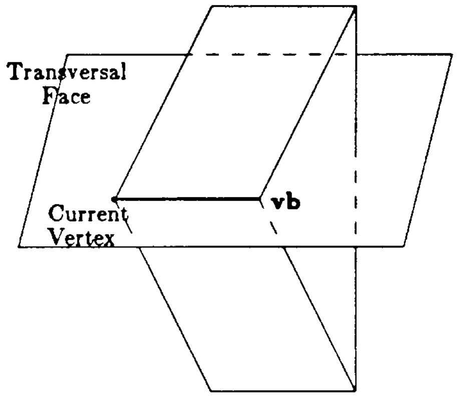


Figure 10. An edge lying entirely in the transversal face. Such an edge must induce a cut in that face.

The edge giving rise to the phantom endpoints is marked, and a list of such edges is kept through the endpoint determination in one face. If the edge is encountered again in the traversal of the current face (two contours may share the edge), the mark prevents the addition of more phantom vertices. When the traversal of the current face is complete, the edge list is examined and indicated edges unmarked. The list structure used to keep track of the edges is then deallocated.

If two contours in different faces share a common edge, each will contribute a pair of phantom endpoints, with no net effect after partitioning.

5.4. Sorting the Endpoints

Sorting orders the calculated endpoints along the intersection line and allows grouping them so that cutting intervals can be determined. Sorting is done first on the value of t associated with each endpoint. A single endpoint may be repeated up to two times per face. Therefore, secondary sorting is performed, based firstly on the face in which the endpoint was found, and secondly on whether the endpoint is an entry or an exit (exiting endpoints place first).

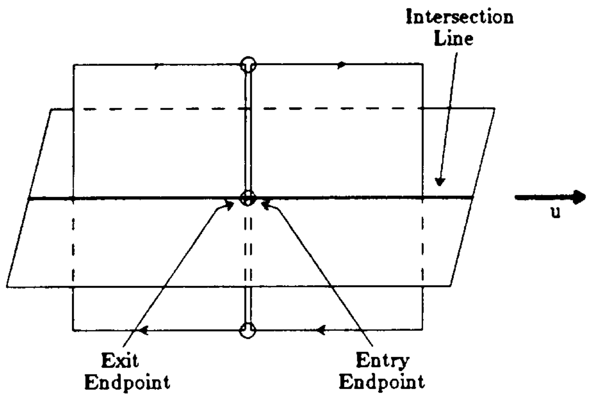


Figure 11. The use of entry and exit attributes to disambiguate endpoints during sorting.

For convenience, UNIX's *qsort* quicksort routine is used for the sorting. The number of endpoints for a pair of faces is typically less than twenty; a pair of convex faces will create 4 enpoints.

Once the endpoints are sorted, a check is made for vertices or provisional vertices whose tolerance regions overlap, requiring a merge. Each vertex is checked against its neighbor. If a merge is indicated, the pair of vertices is replaced by a single vertex with a new tolerance. The merging process may eventually engulf several vertices. Such coalescing poses no problems unless the maximum vertex tolerance is exceeded in the process. A single vertex with a larger than acceptable tolerance also produces an error.

Vertex merging may require the elimination of edges. If an edge appears between two merged vertices, that edge no longer has any meaning, and so is removed. Further, two edges emanating from each original vertex may possess a common endpoint. If both edges belong to the same contour which encloses an angle of less than 180 degrees at the endpoint, then the edges must be eliminated from that contour. Otherwise, after the merge, the edges would define a contour portion enclosing zero-area (a spike), which is disallowed by the regularity condition on face patch boundaries.

5.5. 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 each face. Each endpoint, encountered in turn, flips the state of the flag corresponding to the face that spawned the endpoint. When both flags indicate "inside" the affected faces are appropriately modified, partitioning them along the region of intersection.

There may be as many as two entry endpoints at the start of a cutting interval: one from each face is possible. Sorting insures that if there are two such endpoints, they are adjacent in the endpoint list. Two adjacent entry endpoints indicate two starting points: one for each face. Similarly, a single exit endpoint indicates the end of a segment for only one face; two exit endpoints indicate segment termination for both faces. In either case, the cutting interval ends.

The cutting interval starting and ending point multiplicities are used with face membership information to determine how each of the two faces must be altered to partition them along the detected portion of their intersection. For each face there are three basic possibilities: a cut, a notch, or a slit.

5.5.1. Cuts

A face is *cut* if both endpoints of the cutting interval lie on contours of the same face. 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. In either case the patch between the two endpoints is partitioned at the intersection line.

The cutting process begins by examining the two supplied endpoints. Either endpoint may represent a newly created vertex. If so, the affected edge(s) must be split in two (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 representing the endpoints. The contours indicated by the endpoints are relinked to include the new edge twice, once in each direction. (Figure 13).

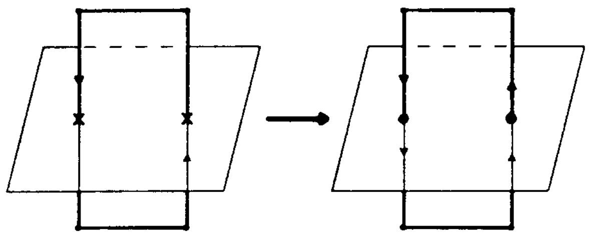


Figure 12. Subdividing edges at segment endpoints.

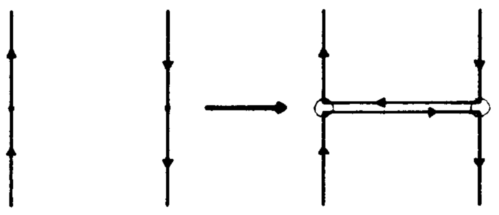


Figure 13. Reforming the contour(s) incident on the endpoints to induce a cut.

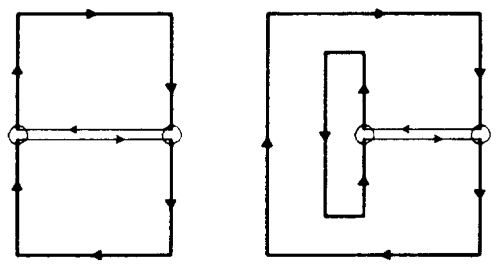


Figure 14. The two forms of a cut.

If the two incident contours are instances of the same one, that contour is split to represent two disjoint regions. If the contours are different, the cut merges the two contours into one (Figure 14).

In either case, these changes must be reflected in the list of the face's contours. Portions of the affected contours may yet lie ahead in the list of sorted endpoints. The circularly linked list of endpoints is used to propagate the new contour information to indicated unencountered endpoints. The list(s) are first relinked to correspond to the edge relinking. The resulting pair of lists (or single list, if the two contours were different) are traversed, and the new contour information is updated for any endpoints lying on the modified contour(s).

A final check is required to determine if the exit endpoint is simultaneusly an entry endpoint (Figure 15). If so, the cut just completed may have altered the edge incident on the exit endpoint's entry double. Since this second endpoint has not yet been encountered, its incident edge field must be appropriately modified.

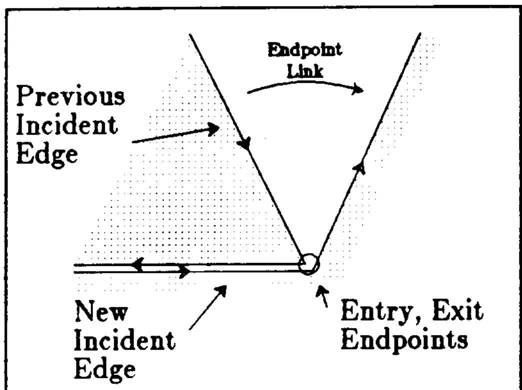


Figure 15a. The situation at a double endpoint after a cut. The incident edge must be updated.

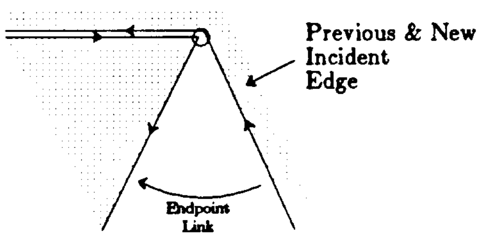


Figure 15b. The incident edge remains correct.

The new incident edge is determined by the direction of the contour's edges at the double endpoint. If the circular links among endpoints are arranged as in Figure 15a, with the exit endpoint's link pointing to its entry double, then the contour orientation is such that the completed cut has replaced the edge incident on the exit endpoint with a new edge. Thus, the incident edge must be changed to the edge created by the cut. Figure 15b shows the other possible arrangement, where the incident edge does not change.

If the cut splits a contour, a flag is set indicating on which side of the transversal face each new contour lies. For each contour, a vector is constructed normal to the intersection line and pointing into the patch enclosed by the contour. The dot product of this vector with the transversal face's normal is formed. If the result is positive, the corresponding contour lies outside the transversal boundary; if negative, it lies inside.

5.5.2. Notches and Slits

A notch (Figure 16) is created in a face if one of the cutting interval endpoints arose from a contour of that face and the other arose from 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 (Figure 17) is placed in a face if both defining endpoints arose from a contour or contours of the transversal face. No edges need be split for a slit; the new edge is created and included once in each direction in a new contour.

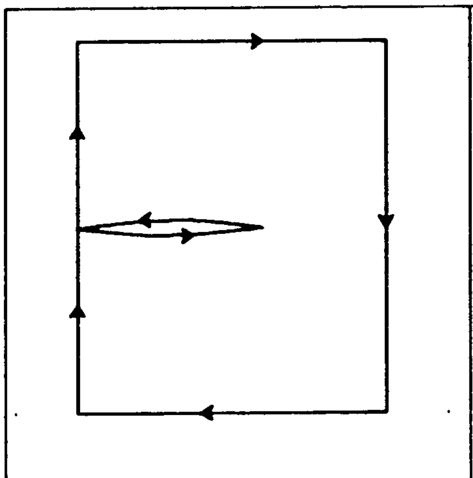


Figure 16. A notch.

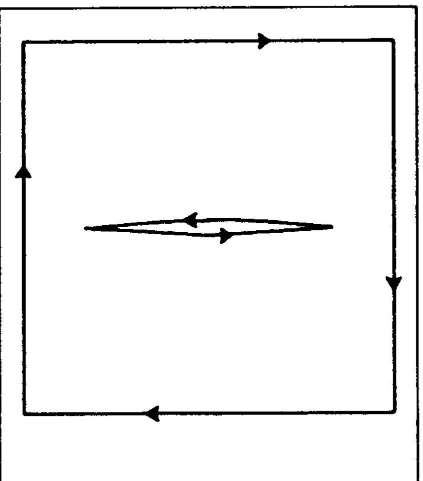


Figure 17. A slit.

Notch and slit creation are simpler operations than cutting because they do not require contour splitting or merging. However, either operation may create a new endpoint that will have to be considered later if partitioning 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 creates an entry endpoint that may enter into further partitioning.

For a pair of faces, only one such isolated endpoint can occur as the result of a single notch or slit operation. Therefore, instead of inserting a new endpoint into the sorted list (which would require shuffling the list to accomodate the insertion), one of the endpoints giving rise to the notch or slit is re-used and filled with the new information. A flag is set indicating that such an endpoint has been created. During the next partitioning step (if any), the entry endpoint may coincide with this immediately preceeding slit or notch induced endpoint. If so, a second entry endpoint, corresponding to the previously isolated one, is added before partitioning proceeds.

6. COPLANAR FACES

If one face's vertices all lie within another face's tolerance region, the two faces are considered coplanar. In this case the partitioning procedure must be quite different from that for transversal faces. Overlapping face regions must be partitioned into disjoint regions bounded by non-intersecting contours in the faces' plane.

One way to achieve this partition is to employ a modified hidden surface removal algorithm that returns visible and invisible face portions in object space.[2] One face is shifted along the other's normal, and the pair is then "viewed" along the normal. A new face is constructed using the resulting visible and invisible contours. A selection among duplicated contours (those occurring as both visible and invisible) is usually made depending on the application.

For fully general face partitioning, a coplanar partitioner must be available; however, if all considered faces belong to boundaries of one or more solids, we need not implement a separate procedure. The algorithm simply records coplanar face pairs on a list for later merging and possible use during contour classification.

The use of edge induced endpoints (section 5.3) automatically partitions any coplanar face pairs. A face in a solid's boundary must be connected on each edge (perhaps indirectly) to another face not coplanar with it. The edges joining the coplanar portion to transversal faces will induce the required cuts when the adjoining faces are processed.

7. SOLID MODELING

A solid modeling operation on boundary representations of two (or more) solids requires producing a set of contours that bounds the desired combination. Portions of the original solids' boundaries will be present in the resultant boundary; these resultant portions are a subset of the partitioned boundaries. Although the essential information for selecting among the boundary portions is gathered during face cutting, some enhancements must be added to the basic partitioning algorithm to support solid modeling operations.

7.1. Propagation of Membership Information

The creation of a solid modeling result from partitioned contours requires that every contour of one solid be classified as inside or outside the other solid. Cutting returns this information by labeling contours. However, many contours in a given boundary remain uncut when two or more boundaries are superimposed. Therefore, some method is needed to classify uncut faces with respect to the other boundary.

The first step is to propagate inclusion information from classified contour to unclassified contour across shared edges.[18] 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 propagation then takes the following form:

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

procedure propagateMark(contour)
    for each edge in contour do
        if contour on other side of edge is unmarked, propagate mark from current to
        unmarked contour.
        propagateMark(other contour);
    end
end propagateMark
```

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

Propagation lessens the number of unclassified contours, but may not eliminate them all; some contours may not be connected to contours that were cut (i.e. the enclosed solid may have holes). To classify these unconnected contours, the face list is traversed once more to find all contours.

For each unconnected contour, a large triangle is constructed. One edge of the triangle is formed by an edge of the unclassified contour. The third vertex is placed just beyond a corner of the bounding box of the transversal solid. The idea is to cut this large triangle against the transversal object, find the the resulting contour containing the original edge, and use the cut contour's classification to classify the original contour.* If no cut occurs, the original edge is outside the transversal solid. The result is then propagated from the original contour to any others that are connected to it across edges or other contours. While a more efficient point-in-polyhedron test could be used,[10] this method employs the already present partitioning machinery, obviating the need for a special algorithm.

7.2. 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. This situation represents a non-transversal intersection of the two boundaries. A small change in the position of either boundary would resolve any ambiguity. (See Figure 18.)

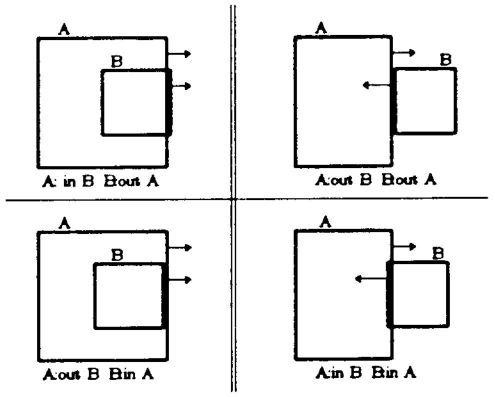


Figure 18. Two dimensional version of coplanar contours problem. Each edge is analogous to a contour. A decision must be made as to which row represents the desired situation. The column is selected by relative normal orientation. Arrows indicate edge normals.

There are two basic situations for a pair of coinciding contours: their relative orientation is either equal or opposite. The algorithm distinguishes between these cases by examining the sign of the dot product of the associated face normals. The requested set-theoretic operation then

*To prevent the transversal object from being cut, a global flag is set which disables cutting for any face not bearing a special mark. The triangle is the only face with the mark, and so is the only face cut.

determines whether one or both of the coinciding contours is kept in the output (Table 1).

Operation	Normal Orientation	
	Same	Opposite
A ∪ B	keep one	keep neither
A ∩ B	keep one	keep neither
A − B	keep neither	keep one

Table 1. Default case selection for coplanar contours in solid modeling. The entries describe what should be done with members of a matching pair of coplanar contours to achieve a solid modeling result.

The algorithm must locate any matching contour pairs so that the correct one(s) can be eliminated from the boolean operation result. After cutting and propagation are complete, 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.

Because the faces have already been partitioned, regions enclosed by coplanar contours are either identical or disjoint. Therefore, 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.

The test for a match is made by examining the face list of the first edge in the contour. This list displays all uses of the edge; each item points to a face and contour in which it is included. The contour's orientation is used along with the face normal to test if a use of an edge in the second face represents a matching contour. If the face normals point in the same direction, the contours match if and only if the edge appears in the same direction in each. If the face normals oppose, the contours match if and only if the edge appears in opposing directions. If a match is found, the default is applied or the user is queried. Coplanar contours with no matching contours are not affected; these will have been given correct tags during propagation.

8. SUMMARY

We have described an algorithm for partitioning intersecting faces into non-intersecting parts. One application of this algorithm has been the pre-processing of polyhedral scenes with intersecting elements so that these can be displayed by renderers that do not tolerate intersections. By virtue of some simple extensions, the algorithm can be used to compute the boundary representation of a set-theoretic operation on two or more solids, each specified as a boundary representation. It is therefore useful as a solid modeling tool.

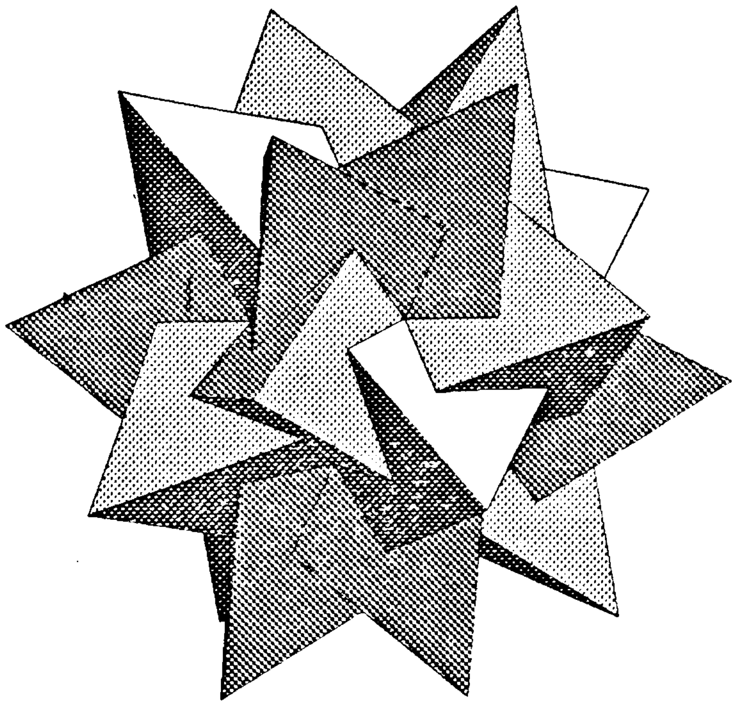
Special care has been taken in the design of the algorithm to ameliorate the effects of inconsistencies arising from inaccuracy in the input specification. It handles designed or incidental alignments between two objects by employing tolerances to indicate when any two vertices, edges, or faces from distinct objects should be considered connected. However, the use of tolerances cannot resolve all possible ambiguities. The algorithm can detect where the use of specified tolerances may fail, but cannot overcome the possible failure to produce topologically viable output.

Our tolerance model is not the only one possible. One way to add some accuracy and make ambiguous cases less likely (at the cost of increased complexity) is to localize a computed vertex along the edge and within the face whose intersection it defines. Instead of a crude tolerance cube around each vertex, which may extend beyond the tolerance region of the indicated face, the vertex tolerance region would be a thin tube lying along the edge and entirely within the face. Thus, face tolerances would not have to increase to accomadate computed vertices.

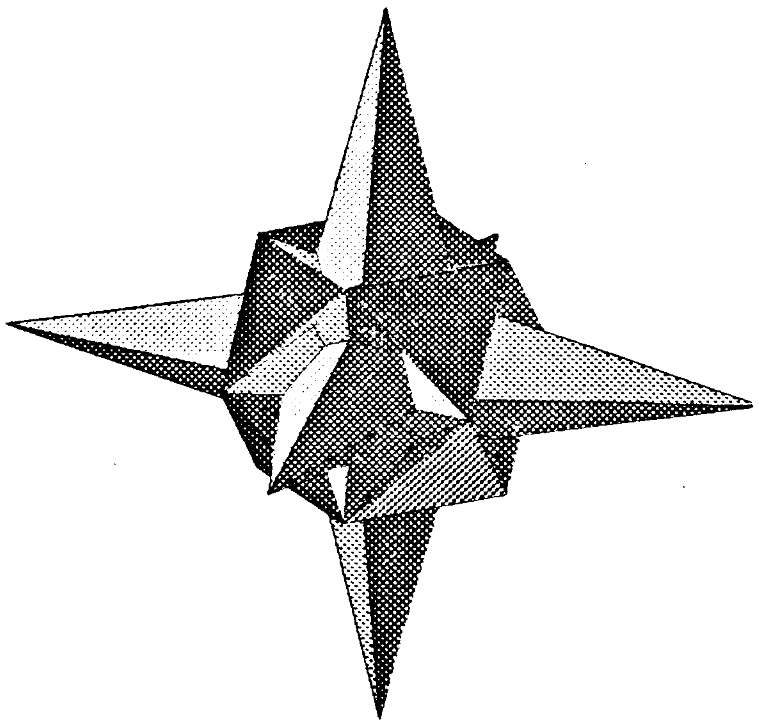
While this model could provide greater accuracy, situations could still arise in which the available numerical information would be insufficient to produce an unambiguous partitioning. Such situations require more extensive description in the affected regions. It would be interesting to examine ways in which such extra information could be derived, either from the user or from some more general description of the objects under consideration.

The algorithm has been implemented in about 2600 lines of C code. This is in addition to about 5000 lines of previously existing subroutines designed to manipulate the UNIGRAFIX[20] data structures and their ASCII descriptions. The program runs in reasonable time (less than 100

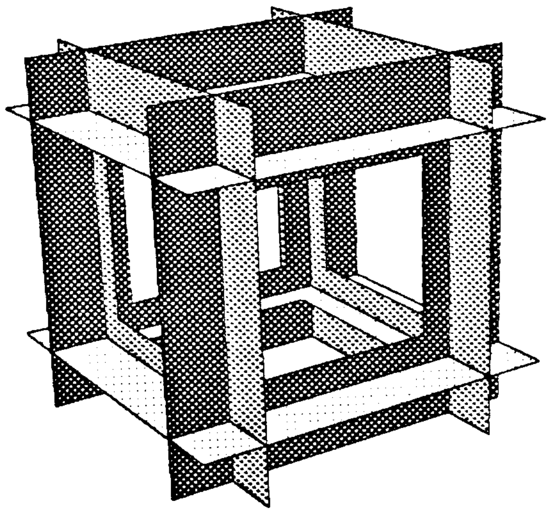
CPU seconds on a VAX 11/750) for objects in which a few hundred faces (about 10000 face pairs) must be checked for possible interaction, but the $O(n^2)$ behavior of the algorithm becomes prohibitive with larger scenes. Of course, if a scene can be grouped into parts whose bounding boxes do not mutually intersect, the program runs in acceptable time by considering each part in turn. Some results of intersection removal and solid modeling operations are reproduced below.



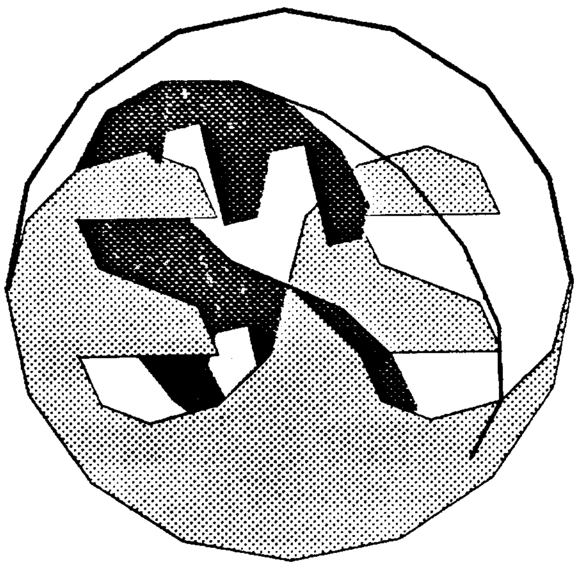
Union of five interlocking tetrahedrons. The algorithm also correctly produces an icosahedron (with no duplicated vertices) for the intersection of these tetrahedrons.



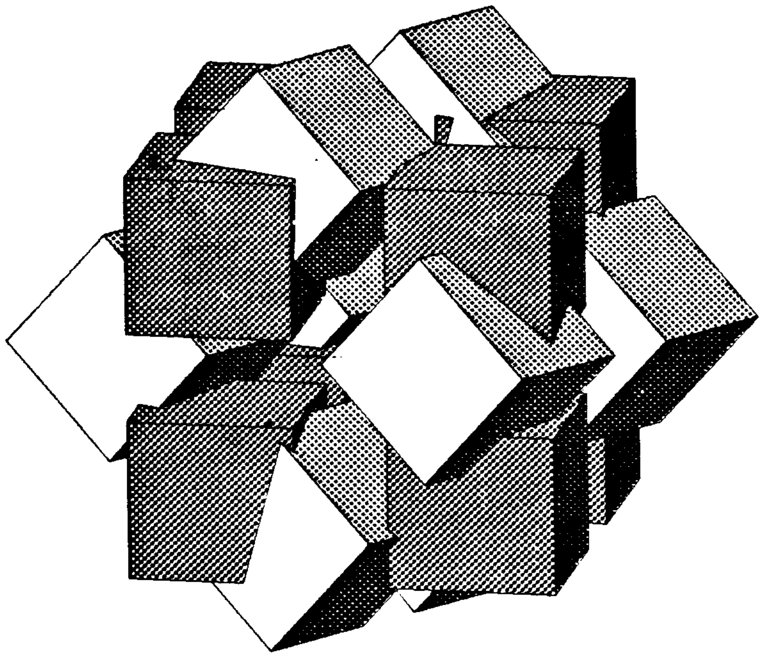
This object was created by starting with a cubeoctahedron with each square face divided into four equal triangles meeting at the square's center. Each square's center point was then moved to a position outside the face on the opposite side, pulling the attached triangular faces with it. The algorithm was run so that the object could be displayed.

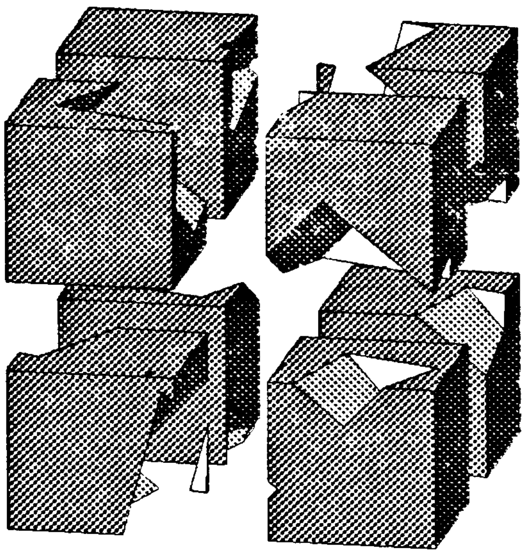


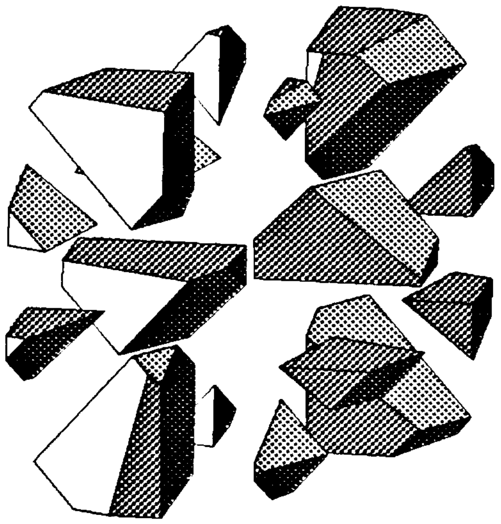
An example illustrating the algorithm's operation on disconnected faces.



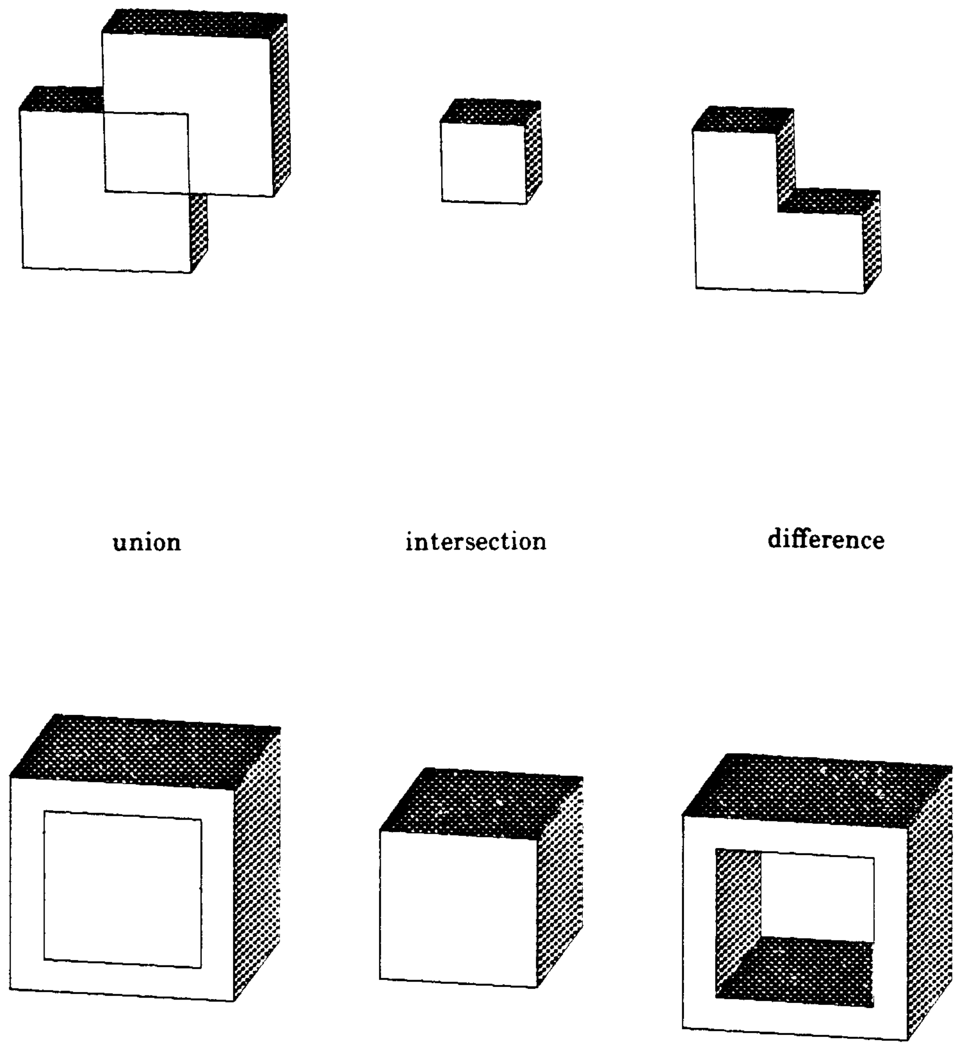
A pair of complex, non-convex faces.





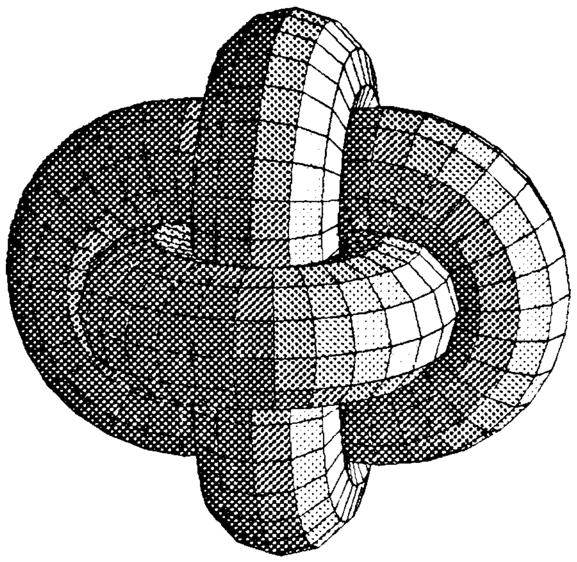


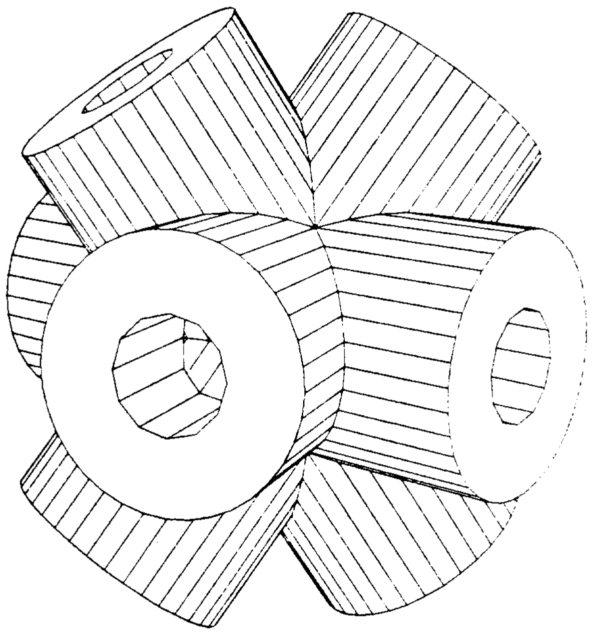
These three examples show the algorithm's operation on simple hierarchical objects. Two objects consisting of eight distinct cubes were superimposed, and the union, difference and intersection simultaneously output. 114 face pairs were paritioned out of a total of 735 pairs considered; total CPU time on a VAX 11/750 was 28 seconds.



Solid modeling with coplanar faces.

```
UNIGRAFIX: User: 0; System: 0; isect started.
UNIGRAFIX: User: 73; System: 0; Readscene terminated.
ugisect: Number of face pairs examined: 836352
ugisect: Number of face pairs rejected using bounding boxes: 530912
ugisect: Number of face pairs partitioned: 504 (= 0.1%)
ugisect: Total cuts: 1072, New Vertices: 520
UNIGRAFIX: User: 2034; System: 14; isect finished.
```





```
UNIGRAFIX: User: 0; System: 0; isect started.
UNIGRAFIX: User: 1; System: 0; Readscene terminated.
ugisect: Number of face pairs examined: 15000
ugisect: Number of face pairs rejected using bounding boxes: 0
ugisect: Number of face pairs partitioned: 1056 (= 7.0%)
ugisect: Total cuts: 2176, New Vertices: 892
UNIGRAFIX: User: 487; System: 8; isect finished.
```

Some larger examples. The program's diagnostic output is reproduced to give an idea of its performance. Both examples were run on a VAX 11/750. In the first example, the use of bounding boxes around faces reduces running time about 30%.

The second example was rendered after intersection processing in 91 seconds on a VAX 11/750. For comparison, a scanline renderer designed to accomadate intersecting faces ran in 251 seconds on the unprocessed version.

Acknowledgements

I would like to thank Ziv Gigus for his pestering me about details of the algorithm's operation in tricky cases. His questions forced me to think hard about some aspects of the intersection removal process, which often led to clearer and simpler methods.

I must also thank my research advisor, Prof. Carlo H. Séquin. Our early discussions played an important role in shaping the underlying philosophy of the intersection removal process. His patience in allowing me to spend needed time to work out many of the numerical problems is much appreciated, as is his careful reading of many earlier versions of this report.

Thanks also go to the other members of the Unigrafix group: Eric Bier, Nachshon Gal, H.B. Siegel, and Paul Wensley, for listening to my ideas and providing suggestions for improved software.

References

1. C.H. Séquin and P.R. Wensley, "Visible Feature Return at Object Resolution," *Computer Graphics & Applications*, vol. 5, no. 6, May 1985.
2. G. Hamlin and C.W. Gear, "Raster-Scan Hidden Surface Algorithm Techniques," *Computer Graphics*, vol. 11, no. 2, pp. 206-213, Summer 1977.
3. Nachshon Gal, "Hidden Feature Removal and Display of Intersecting Objects in UNIGRAFIX," Master's Report, UC Berkeley, January 1986.
4. Scott D. Roth, "Ray Casting for Modeling Solids," *Computer Graphics and Image Processing*, vol. 18, pp. 109-144, 1982.
5. Peter R. Atherton, "A Scan-Line Hidden Surface Removal Procedure for Constructive Solid Geometry," *SIGGRAPH '83 Proceedings, Computer Graphics*, vol. 17, no. 3, pp. 73-82, July 1983.
6. Y. T. Lee and A. A. G. Requicha, "Algorithms for Computing the Volume and Other Integral Properties of Solid Objects," *CACM*, vol. 25, no. 9, pp. 635-650, September 1982.
7. Sheue-lig Lien and James T. Kajiya, "A Symbolic Method for Calculating the Integral Properties of Arbitrary Nonconvex Polyhedra," *IEEE CG & A*, vol. 4, no. 10, pp. 35-41, October 1984.
8. Fujio Yamaguchi and Toshiya Tokeida, "A Solid Modeler with a 4x4 Determinant Processor," *IEEE CG & A*, vol. 5, no. 4, pp. 51-59, April 1985.
9. Steven M. Rubin and J. Turner Whitted, "A 3-Dimensional Representation for Fast Rendering of Complex Scenes," *SIGGRAPH '80 Proceedings, Computer Graphics*, vol. 14, no. 3, pp. 110-116, July 1980.
10. Kevin J. Weiler, "Polygon Comparsion Using a Graph Representation," *SIGGRAPH '80*, pp. 10-18, July 1980.
11. M. Segal, "Maintaining Topology in the Face of Numerical Inaccuracy," In preparation.
12. W.R. Franklin, "Effecient Polyhedron Intersection and Union," *Graphics Interface*, pp. 73-80, May, 1982.
13. V. Guilleman and A. Pollack, *Differential Topology*, Prentice-Hall, Englewood Cliffs, NJ, 1974.
14. R.B. Tilove, "Set Membership Classification: A Unified Approach to Geometric Intersection Problems," *IEEE Transactions on Computers*, vol. C-29, pp. 874-883, Oct. 1980.
15. C. M. Eastman and K. Weiler, "Geometric Modelling using the Euler Operators," *Proc. First Annual Conference on Computer Graphics in CAD/CAM Systems*, pp. 248-254, April 1979.
16. B. G. Baumgart, "Geometric Modelling for Computer Vision," Report STAN-CS-74-463, Stanford AI Lab, 1974.
17. W.M. Newman and R.F Sproull, "Plane Equations," in *Principles of Interactive Computer Graphics, 2nd Edition*, p. 499, McGraw-Hill, New York, 1979.
18. Spencer W. Thomas,, "Modelling Volumes Bounded by B-Spline Surfaces," PHD Thesis, University of Utah, June 1984.
19. Jeff Lane, Bob Magedon, and Mick Rarick, "An Efficient Point in Polyhedron Algorithm," *Computer Vision, Graphics, and Image Processing*, vol. 26, pp. 118-125, April 1984.
20. Carlo H. Séquin, Mark Segal, and Paul Wensley, "UNIGRAFIX 2.0 User's Manual and Tutorial," UC Berkeley Technical Report UCB/CSD 83/161, December 1983.

NAME

ugisect - convert intersecting faces and wires into non-intersecting objects

SYNOPSIS

ugisect [**-I**] [**-D**] [**-U**] [**-A**] [**-vVfF** *eps*] [**-r**] < inputfile > outputfile

DESCRIPTION

Ugisect reads a UNIGRAFIX file and cuts up any intersecting faces to produce a scene description with no intersecting elements. Each existing intersecting element is partitioned into several pieces. The default is to keep all these pieces together in a single statement with multiple contour groups.

Instances of definitions that are intersecting are expanded to the next lower hierarchical level, where all components are again checked for intersection.

Command line flags cause *ugisect* to compute the boundaries of boolean combinations of two solids. To use these options, the input file must consist of exactly two instances at the top level. Faces and wires may occur only in definitions. Each instance must define the boundary of a solid object for the output to be meaningful (*ugisect* does not check that the boundaries are well defined). The instances may contain arbitrary hierarchy.

-I Output the boundary of the intersection of the two solids specified by the two input instances.

-D Output the boundary of the difference of the first specified solid minus the second specified solid.

-U Output the boundary of the union of the two solids specified by the two input instances.

-A <name>
Simultaneously output the intersection, difference and union of the two specified solids. With this option, nothing is placed on standard output; instead, three files "*name*.inter", "*name*.diff" and "*name*.union" are created and the three separate results placed in them. If no name is specified, "inter" is used.

The other flags are:

-v *eps*

-V *eps* Change vertex tolerances. The first form sets the nominal tolerance assigned to vertices as they are read in. The default value is 1e-9. The second form sets the maximum vertex tolerance, which determines how near vertices must lie to be merged into the same vertex. The default is 1e-7.

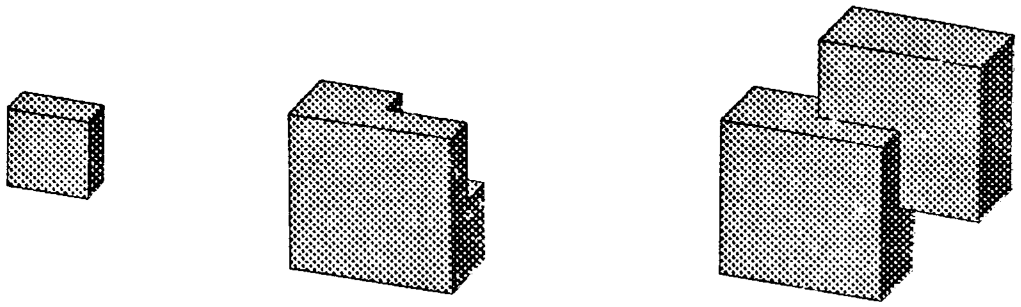
-f *eps*

-F *eps* Change face tolerances. The first form sets the minimum nominal tolerance for input faces; a face may have a computed input tolerance larger than this value if at least one of its vertices lies sufficiently far from its computed plane equation. The default is 1e-6. The second form sets the maximum face tolerance that no face may exceed. It also determines how close together two faces must lie to be considered coplanar. The defualt is 1e-4. If a maximum vertex or face tolerance is exceeded as intersection processing proceeds (as the result of merging), an error message is printed, and the final output may be topologically inconsistent.

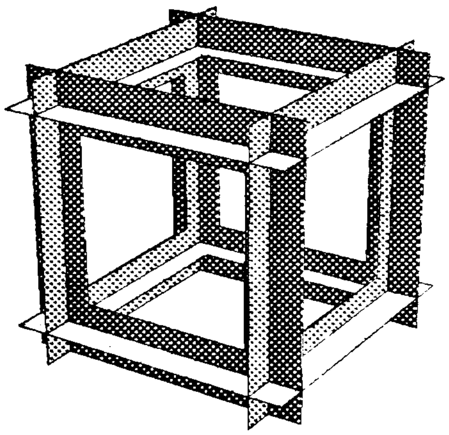
-r Recover vertices. Normally, in the interests of eliminating redundant calculation, computed intersection points are saved even if not immediately incorporated into a cut face. Such a point may arise again in the consideration of other face pairs. However, for large scenes, the memory cost may be prohibitive. The **-r** option prevents intermediate intersection points from being saved, considerably reducing storage requirements.

EXAMPLES

```
ugisect -A slabs < ~ug/lib/slabs
  ugplot -ed .4 .5 -1 -sa -dw -sy 3 < slabs.inter
  ugplot -ed .4 .5 -1 -sa -dw -sy 3 < slabs.diff
  ugplot -ed .4 .5 -1 -sa -dw -sy 3 < slabs.union
```



```
ugshrink -f 1.3 < ~ug/lib/cube | ugshrink -H -f 0.6 | ugisect
   | ugplot -ep -6 5 -10 -ab -sa -dw -sy 2.5 -sx 2.5
```



FILES

~ug/bin/ugisect, ~ug/src/ugc
name.inter, name.diff, name.union

SEE ALSO

ugexpand (UG), ugxform (UG), ugshow (UG), ugplot (UG) ugdisp, (UG)

DIAGNOSTICS

Upon termination *ugisect* will print out statistics on the number of intersecting elements.

BUGS

Does nothing about wires; currently they always pass through uncut.

Will produce a warning and possibly incorrect results if coplanar faces are detected, unless each member of the coplanar pair belongs to a distinct solid boundary.

Instances are expanded whenever their bounding boxes intersect; if it turns out that the instances do not actually intersect the instances are left expanded.

Occasionally a hole is output as the first contour of a face when using boolean operations.

AUTHOR

Mark Segal