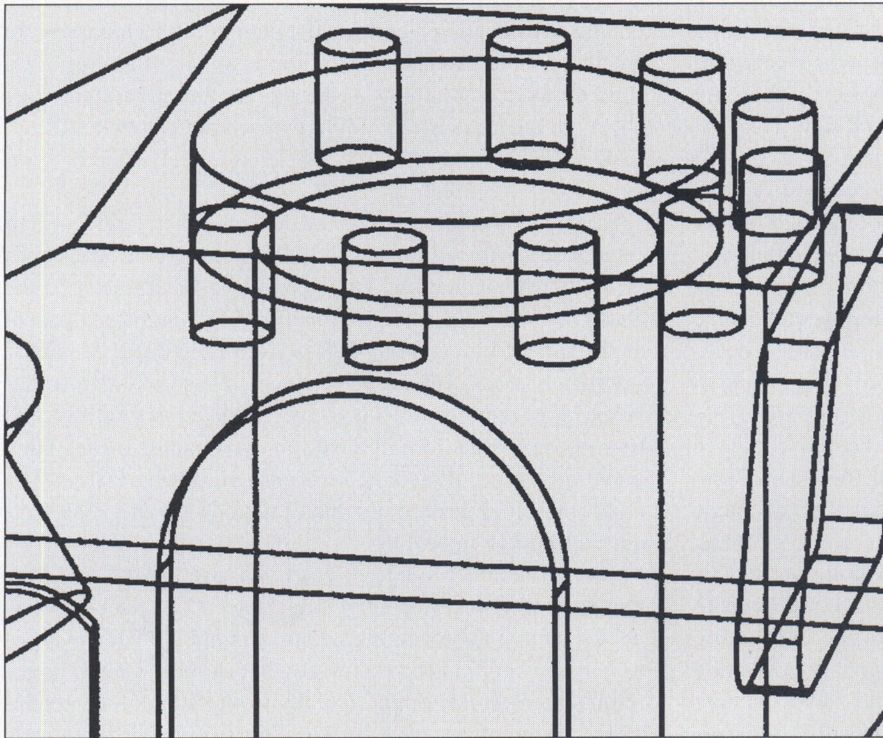
Incremental Boundary Evaluation Using Inference of Edge Classifications

James R. Miller
University of Kansas



An incremental algorithm exploits the simultaneous availability of CSG and boundary representations of solids to avoid most explicit edge classifications.

With the many different representations for solid models,[1] placing the representations into simple categories is difficult. Nevertheless, two basic types of representation are most prevalent in systems: an implicit constructive scheme called *constructive solid geometry* and an explicit boundary-based scheme called *boundary representation*. CSG defines solids in terms of Boolean combinations of primitive solid volumes. The CSG representation of a solid is often called a *CSG tree*, because this representation can use a binary tree whose terminal nodes are the primitive solids and whose nonterminal nodes are the regularized Boolean set operations.[1]

The data structure in the explicit scheme is a graph describing the subsets of surfaces and curves that form the outer boundary of the solid. The elements of this representation—called a boundary representation, or B-rep—form a hierarchy of higher dimensional to lower dimensional forms. The boundary of a solid object consists of one or more *shells* (a single connected "skin"). Each shell is defined by an enumeration of one or more *faces*. A face is a portion of a surface bounded by one or more *loops* on the surface. A loop is a closed, connected set of *edges*, and an edge is a portion of a curve. (A loop may instead be defined by a single vertex. For example, a solid defined by a bounded portion of one half of a right circular cone would have a single vertex loop at the cone's vertex.) Finally, an edge is bounded by two *vertices*. In addition to these lists, B-reps record specific adjacency relationships. Adjacency relationships describe how the elements of an object's boundary are connected.[2]

Solid modeling systems commonly support construction of the solid that results from applying a Boolean operation between two other solids. The computation of the B-rep describing the result is called *boundary merging* if the input solids are described only by B-reps; it is termed *boundary evaluation* if there are CSG descriptions of the given solids.[3] Boundary evaluation is *nonincremental* if we have only CSG descriptions; it is *incremental* if we have redundant CSG and B-rep descriptions of the solids and appeal to both in the algorithm.[3] It is also common to insist that the B-rep describe a *regular* solid: No dangling faces, edges, or vertices are allowed in the B-rep.[1]

In this article I focus on incremental boundary evaluation, although much of the presentation applies as well to boundary merging. The algorithm I describe has been implemented and evaluated in cryph, a geometric modeling system with both CSG and B-rep descriptions of solids, which is being developed and used as a research and teaching tool at the University of Kansas. The B-rep in cryph is based on Weiler's radial edge data structure.[4] Among other things, this data structure explicitly separates the concept of a *use* of a topological element from the element itself. As we shall see, this separation is a natural match to the way we think about boundary evalua-

0272-17-16/93/0100-0071$03.00 © 1993 IEEE

tion, and it lets us simplify the logic in critical sections of the algorithm. The data structure maintains ownership relations between different types of uses. For example, a vertex use knows the unique edge use on which it lies, and conversely the edge use knows the vertex use at which it starts.

For the boundary-evaluation algorithm to generate the B-rep corresponding to a CSG representation of a solid, a necessary condition is that the set of objects representable in CSG be a subset of the set representable with a B-rep. In particular, I call attention to the distinction between two possible domains: manifold objects and regular sets.[1,5] Loosely speaking, regular sets are objects, all of whose faces separate solid material from air and all of whose edges form part of the boundary of a face. Manifolds are a more restrictive class in that, for example, an edge can belong to only two faces. Nonmanifold solid objects are easily represented in CSG and arise quite naturally in modeling operations. Therefore, our data structures and algorithms should support nonmanifold B-reps. Weiler designed his radial edge data structure for nonmanifold models.[4] However, the algorithmic processing required for nonmanifold objects in the boundary-evaluation algorithm is beyond the scope of this article. Here I simply point out those portions of the algorithm affected by the existence of nonmanifold input or output. I tackle the former in another article (submitted for publication under the title "Incremental Boundary Evaluation for Nonmanifold Partially Bounded Solids").

An important part of my approach involves exploiting adjacency information in the B-reps to simplify and accelerate critical parts of the algorithm. While the basic idea is not new (Requicha and Voelcker mentioned the possibility of inferring information in this fashion[3]), it has not been described in the literature, and I am not aware of systems that operate in this fashion. The concept is simple, but subtle issues arise.

The boundary-evaluation algorithm depends on standard curve and surface utilities:

- The computation of intersections between curves and surfaces.
- Determination of coincidence relationships between points, curves, and surfaces.
- The computation of differential quantities at a point on a curve or surface.

An appropriate interface to a set of such utilities lets the boundary-evaluation algorithm operate without knowledge of internal representations of curves and surfaces or of the specific algorithms used to support these operations. We can therefore push all uncertainty due to numerical inaccuracies into black-box geometry-specific routines. I constructed the boundary-evaluation algorithm in cryph in this fashion, so I do not discuss the extensive literature describing these utilities. References to descriptions of the representations and many of the cryph intersection algorithms can be found elsewhere.[6,7]

Previous work

Requicha and Voelcker[3] provided one of the earliest and clearest descriptions of theoretical and computational issues in boundary evaluation. They established a framework for classifying algorithms and surveyed several algorithms known at the time. My approach differs from theirs in a couple of significant areas. I generate connected instead of maximal faces. More significantly, I actively use the adjacency information in the B-reps to minimize required computation. They make no use of adjacency information, noting instead the point in their algorithm at which it could be generated if required by other algorithms.

Several authors provide extensive descriptions of boundary merging algorithms for polyhedral objects.[5,8-11] Details vary, but a common approach is based on computation and analysis of vertices and explicit vertex neighborhoods.[10] These algorithms often do not extend directly to the curved-surface domain.[5] By comparison, I designed my approach for models with nonlinear boundaries, and it uses edge and face neighborhoods instead of vertex neighborhoods, which are very difficult to represent and manipulate in curved-surface domains.[3]

Requicha and Voelcker[3] and Mäntylä[5,10] established some mathematical characterizations and theoretical concepts that space does not allow me to review. Here I include only what is necessary for my description.

Crocker and Reinke[12] describe a nonincremental boundary-evaluation algorithm for solids whose boundaries can be represented or approximated with piecewise polygons and quadrics. Their algorithm is built on the PADL-2 boundary evaluator.[3] More recently, they developed methods enabled by advances in nonmanifold data structure representations, which dramatically improve the performance of editing operations.[13]

High-level approach

Classification of a candidate set E with respect to a subject set S is the process of partitioning E into subsets that lie entirely inside, outside, or on the boundary of S.[14] It is common to use the notation in$S$, out$S$, or on$S$ to denote the subsets of E that are respectively inside, outside, or on the boundary of S.

We say a face on the boundary of a solid is *connected* if between any two points in the face interior there is a path that lies entirely in the face interior. If a face on a B-rep consists of all subsets of a surface that lie on the solid's boundary, the face is said to be *maximal.* My algorithm generates connected faces. Connected faces make the detection of multiple shell

An important part of my approach involves exploiting adjacency information in the B-reps to simplify and accelerate critical parts of the algorithm.

results easier, and it is often necessary to attach different attributes to different connected faces on the same surface.

We say two loops on a surface are *compatible* if, on the surface, there is a path from an edge on one loop into its local neighborhood that arrives at an edge of the other loop on the side of its local neighborhood, and the path does not cross any edge in either loop. On surfaces of topological genus zero, two compatible loops will always form a well-defined (but possibly unbounded) connected face. Though symmetrical, compatibility is not a transitive relationship.

An edge lying on one of the solids input to the boundary-evaluation algorithm is called a *self edge*. A new edge arising from the intersection of a face on one operand with a face of the other is called a *cross edge*. If a cross edge is also a self edge, we call it a *CESE* (cross edge self edge).

Consider evaluating the boundary of a solid C defined as the result of applying a Boolean operation $\circ$ between solids A and B ($C \leftarrow A \circ B$). We can greatly simplify an implementation of the boundary-evaluation algorithm by using the identities

$$A - B \equiv A \cap {\sim}B$$
$$A \cup B \equiv {\sim}({\sim}A \cap {\sim}B)$$

where $\sim$ denotes the unary complement operator. That is, with appropriate preprocessing and postprocessing, we need to implement only the intersection operation. Since we use the radial edge data structure,[4] the complement operation requires only pointer modification; hence this adds little overhead to the overall algorithm. I therefore present the algorithm assuming only intersections are performed, but for clarity I use all three operations when showing sample geometry in figures.

The standard approach is based on the generate-and-test paradigm: Generate sets of faces, edges, and vertices known to contain all those of C, then test each member of the three sets to determine which ones belong to the boundary of C.[4] The faces of C are a subset of the faces of A and B. To determine the subset, faces generally must be partitioned. That is, for each face f from either A or B, f will survive in its entirety on C, f will not exist at all on C, or some subset of f will exist on C.

Similar remarks apply for edges. The edges of C are a subset of the self edges of A, the self edges of B, and the cross edges arising from intersections between the faces of A and B. Again, to determine the subset, we must first partition the edges. Aside from the one exception discussed in the next section, we partition the self edges of one operand at their discrete points of contact with faces on the other operand.

The vertices of C are a subset of the vertices of A, the vertices of B, and new vertices created during the edge partitioning and cross-edge generation steps. We can base the subset determination on point classification.

The interior of the partitioned faces and edges will have a constant classification with respect to C. Therefore, we can determine which partitions comprise the boundary of C by classifying an arbitrary point in the interior of each partition. However, it suffices to compute the edge classifications, from which we can deduce the faces of C. The vertices of C are simply those on the surviving edges (and surviving single vertex loops).

Recalling our earlier characterization of regular sets, we can decide whether an edge lies on the boundary of C (that is, classify the edge with respect to C) by studying its neighborhood. If an edge is completely surrounded either by solid material or by air, then it is not on the boundary; otherwise it is. Different classification approaches are appropriate for the three types of edges mentioned earlier: cross edges, self edges, and CESEs.

We first consider cross edges. Any cross edge arising from the transverse intersection of a face of A with one from B will be partly surrounded by material and partly surrounded by air (and hence will lie on C), regardless of the orientations of the two faces involved. Cross edges surviving on C that lie on tangential curves of intersection between surfaces of A and B must have been CESEs. (If either the input or output is nonmanifold, this assertion might not hold.)

We classify self edges that do not lie on the boundary of the other solid by simply classifying a point in the interior of the edge with respect to the other solid. This classification will be either in$X$ or out$X$ (X being the other solid—either A or B). If the classification is in$X$, then the neighborhood of the edge on C will be the same as it was on the input solid (that is, partly surrounded by air, partly surrounded by material). Hence the classification must be on$C$. If instead the classification is out$X$, then the edge is completely surrounded by air; hence its classification with respect to C must be out$C$.

Summarizing, we can conclude without any explicit computation that pure cross edges must lie on C, and we can determine whether a pure self edge lies on C simply by classifying a point with respect to the other solid. It is only when classifying CESEs that we must resort to some explicit determination and analysis of edge neighborhoods. My approach is similar to that described by Requicha and Voelcker[3] for all edges, and I describe it in the next section.

My algorithm actually performs far fewer explicit edge classifications than has been implied. By taking direct advantage

```
if p is a vertex of the edge then
    if p is a vertex of the face then
        merge the vertices
    endif
else if p is in the interior of the edge then
    if p is a vertex of the face then
        split the edge at the vertex
    else if p is on or in the face then
        create a vertex at p and use it to split the edge
endif
a

for each curve, c, on A do
  for each surface, s, on B do
    intersect(c,s) to get a list of points
      for each point, p, in the list do
          <logic shown above using an edge on c and a face on s>
b
```

Figure 1. Self-edge partitioning: (a) classifications, (b) logic.

of adjacency information among the self edges, it can infer virtually all self-edge classifications without point classification or other numerical computation. Moreover, it can determine the resulting connected faces simply and directly.

The algorithm

The algorithm proceeds as follows:

1. If A or B is the empty solid, then generate C trivially and exit.
2. Partition self edges of A at their points of intersection with faces of B.
3. Partition self edges of B at their points of intersection with faces of A.
4. Compute and partition cross edges, associating uses with surfaces on A and B.
5. Classify CESE uses discovered during step 4.
6. Infer self-edge classifications, splitting and merging faces as appropriate.
7. Check for split and merged shells.

Self-edge partitioning (steps 2 and 3)

Each edge on one operand must be compared with each face on the other. We first intersect the curve underlying the edge with the surface underlying the face. If the curve lies on the surface, we do nothing at this stage. The edge may be a CESE, but this will be discovered in step 4. We classify each discrete intersection point p with respect to both the edge and the face. Figure 1a describes the use of the classifications. Frequently, multiple edges lie on a curve, and multiple faces lie on a surface. To minimize computations, we organize the partitioning logic as shown in Figure 1b.

Cross-edge generation (step 4)

Each face of A must be intersected with each face of B. Again, because multiple faces frequently lie on a surface, we organize the process by looping over all surfaces s_A in A and s_B in B. Cross edges are then those portions of a curve of intersection between s_A and s_B that lie in the interiors of a face on s_A and a face on s_B. If we find an s_A and s_B to be identical, then we throw away one surface, forcing all faces that formerly referenced it to refer to the other surface. We haven't sufficient information at this stage to do anything further; we will discover any faces that must get merged during step 6.

We have an additional motivation for proceeding surface by surface. Distinct connected faces on the input may be merged into a single connected face on the result. This affects the classification inference process of step 6, since we need to consider at once all faces lying on a given surface. As a result, it is convenient to organize cross-edge uses as well as the classification inference process of step 6 on a surface-by-surface basis.

Assuming s_A and s_B intersect, our first task is to determine the proper orientation of a cross edge on each face. We define the orientation of an edge use on a face so that the face interior is locally to the left as we look in the direction determined by the edge use. Curves on which edges lie are oriented. Hence uses of edges need only a flag that states whether the orientation of the use is the same as or opposite that of the underlying edge. Uses of edges on adjacent faces have opposite orientations, so we need only establish the orientation of one use. We use an arbitrary point P on the curve to test whether the orientation of a use on a face of s_A should agree with the curve orientation. We compute the unit tangent vector $\mathbf{u}$ to the intersection curve at P, and the unit outward-pointing normal vectors $\mathbf{n}_A$ and $\mathbf{n}_B$ to faces on surfaces s_A and s_B also at P. If the orientation should be recorded as "agree," then the vector from P into the face on s_A should point away from $\mathbf{n}_B$. That is,

```
if ((nA × u) · nB) < 0.0 then
    orientationFlag = AGREE
else
    orientationFlag = DISAGREE
```

If $\mathbf{n}_A$ is parallel to $\mathbf{n}_B$, then $((\mathbf{n}_A \times \mathbf{u}) \cdot \mathbf{n}_B)$ will be precisely zero. This can occur at one or more isolated points on the curve. If P is such a point, we can simply choose another and proceed. If the curve is a tangent curve of intersection, however, then $\mathbf{n}_A$ is parallel to $\mathbf{n}_B$ throughout. Assuming the input and output are manifold, there can be no new edges on such a curve. We proceed with the remainder of the logic here, however, because there may be CESEs lying on this curve (for example, when creating a fillet).

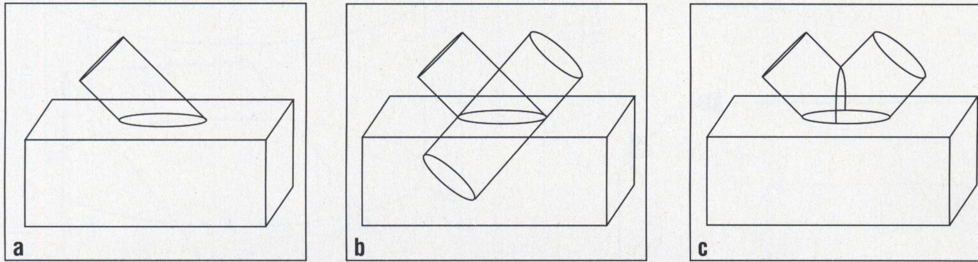


Figure 2. (a) Union of a block and a bounded cylinder. (b) Positioning another bounded cylinder. (c) The original elliptical self edge is partitioned while generating and partitioning cross edges arising between the solid and the second cylinder.

Next we partition the curve. The naive approach for partitioning the cross curves is to do an edge partitioning operation as outlined above, using all faces adjacent to a face lying on s_A or s_B. This is undesirable for a number of reasons. It requires unnecessary curve-surface intersection operations and point-face classifications. Furthermore, it usually creates more partitions than necessary. Because of the self-edge partitioning of steps 2 and 3, all points at which the curve must be partitioned are either vertices on the boundary of faces lying on s_A and s_B or isolated points of tangency between s_A and s_B. This means we can find partitioning points simply by performing point-on-curve tests using the vertices adjacent to the faces.

Because cross edges must be partitioned at points of tangency between s_A and s_B, four cross edges are generated, for example, if two bounded cylinders intersect in a pair of ellipses. Similar situations force the additional self-edge partitioning mentioned in the previous section. Consider the geometry of Figure 2a. If we add the second cylindrical primitive as shown in Figure 2b, we need to partition the original self edge lying on the ellipse to connect it properly with the new cross edge lying on the other ellipse, but we do not discover this until generating the cross edges.

If the two surfaces have one or more isolated points of intersection that lie in the interior of a face on *A* and one on *B*, then we must classify the isolated points in addition to intersection curve partitions. Points that classify as on*C* are added as single vertex loops on the containing faces. The presence of single vertex loops usually indicates the existence of a nonmanifold condition, but it will occasionally arise in manifold situations, such as when the vertex of a cone is a part of the boundary of a model.

We search for the vertices that partition a cross edge by examining the lead vertex on each self edge on each face lying on one of the surfaces. We could query the face vertices directly, but we proceed in this fashion so that we can look for CESEs in the process. At each self edge, we check to see whether the underlying curve is coincident with the current cross edge. If so and if the edge is also on or in a face on the other operand, then we record this cross-edge partition as a CESE. We must tag each partition since some cross-curve partitions may be CESEs, while others are pure cross edges (Figure 3).

CESE classification (step 5)

During step 4, we associated cross-edge uses with surfaces. In step 6 we look at these uses to infer classifications of the self edges on faces lying on those surfaces. We also detected CESEs during step 4. As we explicitly classify these CESEs, we decide face by face which uses we need and which we must delete. The low-level Euler operators delete an edge if and when all its uses are deleted.

The geometric computations are basically the same as those that Requicha and Voelcker describe.[3] We slice both solids with a plane perpendicular to and passing through the middle of the CESE. This generates two curvilinear wedges on the plane, one from each solid. If the wedge interiors are disjoint, then the neighborhood of the edge with respect to the resulting solid is empty, and all uses of the CESE are deleted. If the interiors overlap and the boundaries are all distinct, then we decide the fate of CESE uses on a straightforward face-by-face basis. If the sectors share a boundary that is a boundary of the result, then we have overlapping faces that must get merged. We arbitrarily choose one face to survive, and all the self edges and CESEs of the other that lie on the boundary of the result are transferred to the surviving face as if they were cross edges.

If the conclusion of this analysis is that a use of the CESE must survive on a face that did not already have a use (that is, if the CESE was a self edge only on the other operand), then we create a new use and add it to the face as if it were a pure

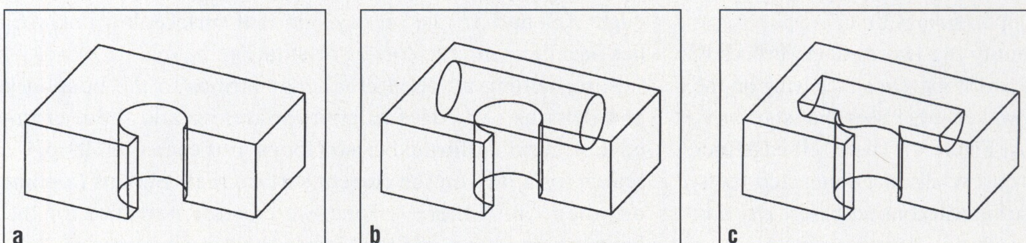
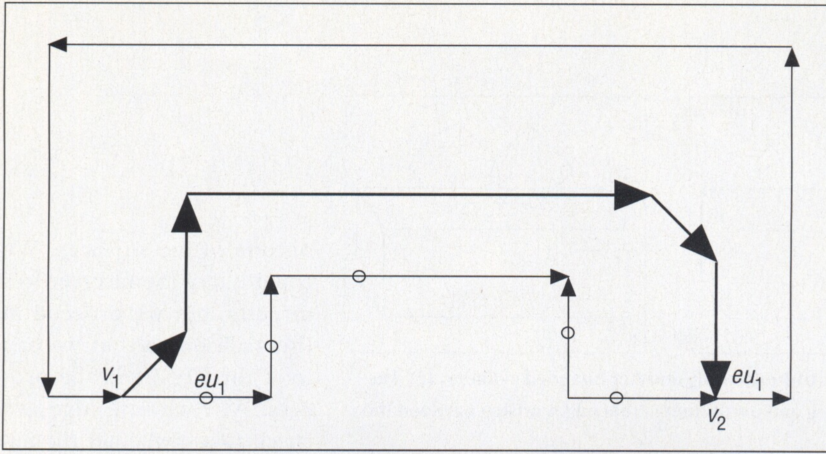


Figure 3. (a) A simple solid. (b) A bounded cylinder induces a CESE and a pure cross edge on the same curve of intersection. (c) The solid that results from subtracting the bounded cylinder.



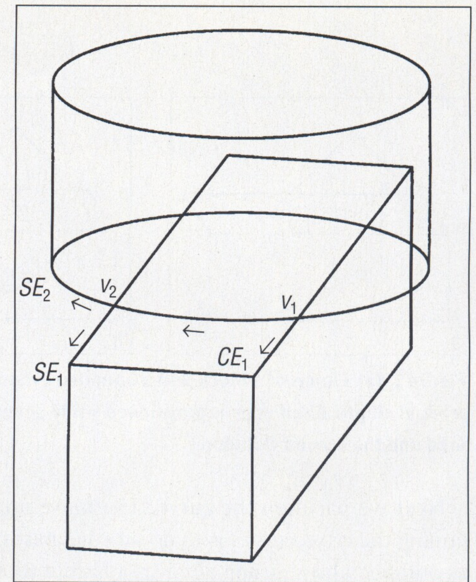


Figure 4. (Above) When stitching in the open cross-edge string (bold), we infer that the self-edge uses marked with a circle are to be deleted and the others retained.

Figure 5. (Right) Satisfying the second requirement results in CE_1 being connected to SE_1 instead of SE_2.

cross edge. If instead the CESE was a self edge on both operands, then we delete one edge at this stage and transfer ownership of its uses to the other.

CESEs must satisfy one further requirement before being allowed to survive on the result: They must delimit faces lying on different surfaces.

Inference of self-edge classifications (step 6)

There are two distinct stages of self-edge classification inference. The first and more complex is the classification inference for self edges in loops touched by cross edges. The second stage handles those loops (and faces and shells) not touched by cross edges.

By the time we reach this step of the algorithm, those CESE uses that should survive on the result have been "promoted" to pure cross-edge status on the appropriate surfaces. Therefore, with one very minor exception during the second stage, we no longer need to distinguish CESEs from cross edges. For the remainder of this section (unless explicitly stated otherwise), the term cross edge includes CESEs added to a surface as a cross edge during step 5.

We first form connected strings of cross-edge uses on each surface. Because of our earlier processing, this requires no numerical computations or comparisons, rather only pointer-based logic. That is, we simply connect cross edge i to cross edge j if i stops at the same vertex at which j starts. For manifold faces, any vertex will determine exactly one such (i, j) pair. After we have connected all cross-edge uses in this fashion, some strings will form closed connected loops, and others will be open. We consider the open strings first.

For each open string we identify the two distinct vertices v_1 and v_2 at which the string starts and stops. On a face lying on the current surface, there will be two self-edge uses that start at v_1 and v_2, which we call eu_1 and eu_2 (Figure 4). These self-edge uses may lie on a common face or on two distinct connected faces. The last cross-edge use in the string will connect before eu_2, and the first in the string will connect after eu_1's predecessor.

We deduce self-edge classifications from how these open strings connect into the boundary. For example, if there are only one face and one such string of cross-edge uses on the surface, then we must delete all self-edge uses starting with eu_1 and stopping with eu_2's predecessor. All other self-edge uses on the face survive (Figure 4). In general, there will be multiple strings of cross-edge uses and multiple connected faces on a surface. Therefore, when processing a given string, we don't have sufficient information to delete self-edge uses immediately. Instead we mark the vertex use of the first cross-edge use in each string, mark the vertex use of each eu_2, and remember each eu_1. After all open cross-edge strings on a surface have been processed, we take each remembered eu_1 in turn and delete it and its successors until we reach an edge use whose vertex use has been marked.

Determining the proper eu_1 and eu_2 is crucial. To find eu_1 and eu_2, we search for uses of the vertices v_1 and v_2 instead of looking for all edges on all faces on the surface that start at the vertices. The number of vertex uses is generally much smaller than the number of edges, and the desired vertex use vu must satisfy only two simple conditions:

1. Face(vu) lies on the current surface.
2. Face(vu) has the same orientation as the face on which the current cross edge was discovered.

Additional conditions must be satisfied if faces may be nonmanifold.

Figure 5 illustrates why we need the second test. CE_1 is a cross edge lying on the top surface of the block. Both self edges SE_1 and SE_2 lie on faces on that surface, but only SE_1 lies on a face with the correct orientation.

After adding all open cross-edge strings to the boundary and deleting self edges as appropriate, we add those cross-edge strings that formed closed loops. For each such loop, we search for a face on the current surface that contains (a point on) the loop. If there is none, we create a new face for the loop; otherwise, we add the loop to the containing face.

Figure 6. Recursive procedures for detecting split and merged shells: (a) test_shell, (b) mark_adjacent_faces.

```
test_shell(s):
f = first face in the unordered list of faces bounding s
if f is marked then
    {two shells have merged; this one can be deleted}
    delete_shell(s)
else
    mark_adjacent_faces(f)
    if any face in s's list of faces is still unmarked then
        {s has been split}
        create a new shell, s', containing the unmarked face
        test_shell(s')
endif
a

mark_adjacent_faces(f):
mark f
for each edge use, eu, on f do
    raf = radially_adjacent_face(eu)
    if raf is unmarked then
            mark_adjacent_faces(raf)
b
```

As we add cross-edge strings (open or closed) to the boundary, faces may get split or merged. Two connected faces are merged if a loop from each is bridged by an open cross-edge string or if they share a CESE. Faces can get split as we stitch cross edges into the boundary or add new closed cross-edge loops to faces. If attaching a cross-edge loop into the boundary and the subsequent deletion of intervening self edges cause a loop to split in two, we create a new loop from one piece and add it to the face as if it were a new cross-edge loop. We detect this condition while deleting the self edges between marked vertices, so we know all newly formed loops. When we add a new closed cross-edge loop to a face, we determine whether it splits the face, using the notion of compatible loops described in the section entitled "High-level approach." We assume that the face is initially defined by a set of compatible loops. If the newly added loop is compatible with all the existing ones, we can simply add it to the face. Otherwise, we must take the new loop and all the existing ones, and determine the connected faces by partitioning the set of loops into the smallest number of sets, so all loops within a set are compatible with each other.

The second and final stage of self-edge classification inference is straightforward. While we cannot infer directly the classifications of self edges in loops untouched by cross edges, all self edges in such a loop will have the same classification. Therefore, it suffices to classify a single point on the loop. If the loop contains a CESE use, then the classification of the CESE use determined during step 5 is the entire loop's classification. Otherwise, we generate an arbitrary point in the interior of a self edge in the loop, classify it with respect to the other solid, use the classification as described in the section entitled "High-level approach," and propagate the result to all self-edge uses in the loop. If an entire face or shell was untouched by cross edges, then the loop result propagates throughout the face or shell.

Checking for split and merged shells (step 7)

Our approach for detecting split and merged shells assumes connected faces were generated during step 6. It is based on the ability to traverse the elements of a B-rep in two fundamental ways: by querying the lower dimensional boundary of a higher dimensional form (in this case, the faces bounding a shell), and by following adjacencies in edge uses. In particular, we traverse successive edge uses around a face and query the faces radially adjacent to a given edge use. We begin by assuming all faces are unmarked and proceed as follows:

for each input shell, *s*, on each operand **do**
  test_shell(*s*)

Figure 6a shows the recursive procedure test_shell. Figure 6b shows the final recursive procedure mark_adjacent_faces.

Implementation

The algorithm, implemented in cryph, runs on a variety of systems, including Silicon Graphics Iris workstations and DEC servers. I have implemented no global optimization techniques of any sort. When determining cross edges, for example, the algorithm intersects all surfaces of *A* with all surfaces of *B*. It uses no initial culling based on, for example, spatial extents. I plan to evaluate some appropriate methods, but their absence now should be considered when examining the execution times in Table 1.

Table 1. The algorithm's performance in evaluating the parts shown in Figures 7 and 8.

Model	Number of Primitives	Time to Evaluate from Scratch (seconds)	Number of Individual Boolean Ops Requested	Number of Individual Boolean Ops with Subsecond Response	Longest Response Time (seconds)	Average Response Time (seconds)
L-bracket (Fig. 7)	11	2.33	17	17	0.48	0.14
ANC101 (Fig. 8)	72	37	67	59	10	0.55

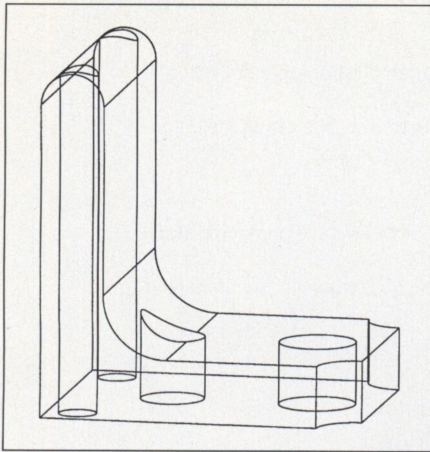
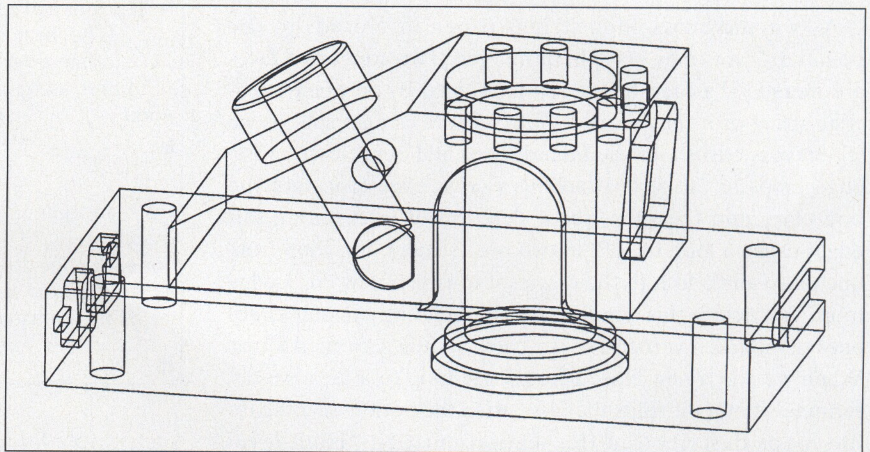


Figure 7. An L bracket.

Figure 8. CAM-I testbed part ANC101.



The algorithm evaluated the B-reps for the two parts shown in Figures 7 and 8. Table 1 shows the time required to evaluate the B-rep from the CSG tree. It also gives the incremental response time for individual Boolean operations. Since this is what a designer actually sees while designing the part, it is a more meaningful measure. The table gives the total number of such requests (cryph allows Boolean operations to be specified between *n* operands) and the number satisfied in less than a second. For the ANC101, six of the eight operations requiring more than one second were satisfied in less than four seconds. The one requiring 10 seconds was a Boolean applied to five children; hence it was like four traditional Boolean operations. The other operation requiring more than four seconds was a Boolean applied to three children. All times were measured on a Silicon Graphics Iris 4D/220 workstation.

Summary

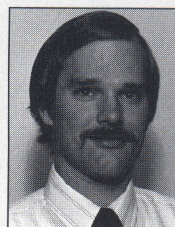
An incremental boundary-evaluation algorithm exploits adjacency information in B-reps to minimize the number of explicit edge classifications required. Evaluations of the implemented algorithm show that it performs reliably and well. Global optimization schemes could increase performance significantly. I have submitted for publication another article that describes extensions to this algorithm supporting nonmanifold and semi-infinite solids. ❑

Acknowledgments

I am indebted to Kevin Weiler for his willingness during the early stages of this work to discuss at length various aspects of his radial edge data structure. This work was supported in part by the University of Kansas general research allocation #3760-X0-0038.

References

1. A.A.G. Requicha, "Representations for Rigid Solids: Theory, Methods, and Systems," *ACM Computing Surveys*, Vol. 12, No. 4, Dec. 1980, pp. 437-464.
2. K.J. Weiler, "Edge-Based Data Structures for Solid Modeling in Curved-Surface Environments," *IEEE CG&A*, Vol. 5, No. 1, Jan. 1985, pp. 21-40.
3. A.A.G. Requicha and H.B. Voelcker, "Boolean Operations in Solid Modeling: Boundary Evaluation and Merging Algorithms," *Proc. IEEE*, Vol. 73, No. 1, Jan. 1985, pp. 30-44.
4. K.J. Weiler, *Topological Structures for Geometric Modeling*, doctoral dissertation, Rensselaer Polytechnic Inst., Troy, N.Y., 1986.
5. M. Mäntylä, *An Introduction to Solid Modeling*, Computer Science Press, Rockville, Md., 1988.
6. J.R. Miller, "Geometric Approaches to Nonplanar Quadric Surface Intersection Curves," *ACM Trans. Graphics*, Vol. 6, No. 4, Oct. 1987, pp. 274-307.
7. J.R. Miller and R.N. Goldman, "Using Tangent Balls to Find Plane Sections of Natural Quadrics," *IEEE CG&A*, Vol. 12, No. 2, Mar. 1992, pp. 68-82.
8. H. Chiyokura, *Solid Modelling with DESIGNBASE: Theory and Implementation*, Addison-Wesley, Reading, Mass., 1988.
9. C.M. Hoffmann, *Geometric and Solid Modeling: An Introduction*, Morgan Kaufmann, San Mateo, Calif., 1989.
10. M. Mäntylä, "Boolean Operations of 2-Manifolds Through Vertex Neighborhood Classification," *ACM Trans. Graphics*, Vol. 5, No. 1, Jan. 1986, pp. 1-29.
11. C.M. Hoffmann, J.E. Hopcroft, and M.S. Karasick, "Robust Set Operations on Polyhedral Solids," *IEEE CG&A*, Vol. 9, No. 6, Nov. 1989, pp. 50-59.
12. G.A. Crocker and W.F. Reinke, "Boundary Evaluation of Non-Convex Primitives to Produce Parametric Trimmed Surfaces," *Computer Graphics* (Proc. Siggraph), Vol. 21, No. 4, July 1987, pp. 129-136.
13. G.A. Crocker and W.F. Reinke, "An Editable Nonmanifold Boundary Representation," *IEEE CG&A*, Vol. 11, No. 2, Mar. 1991, pp. 39-51.
14. R.B. Tilove, "Set Membership Classification: A Unified Approach to Geometric Intersection Problems," *IEEE Trans. Computers*, Vol. C-29, No. 10, Oct. 1980, pp. 874-883.



James R. Miller is an associate professor of computer science at the University of Kansas. His research interests are computer graphics and geometrical modeling for mechanical CAD/CAM.

Miller received his BS in computer science from Iowa State University and his MS and PhD in computer science from Purdue University. He is a member of ACM Siggraph.

Miller can be reached at the University of Kansas, Dept. of Computer Science, 415 Snow Hall, Lawrence, KS 66045-2192.