GEOMETRIC MODELING: A First Course

Copyright © 1995-1999 by Aristides A. G. Requicha
Permission is hereby granted to copy this document for individual student use at USC, provided that this notice is included in each copy.

6.5 Boolean Operations

This section discusses several problems related to computing BReps of solids that are combined by regularized set operations, usually called simply *Boolean operations*.

6.5.1 Boundary Evaluation and Merging

Regularized set operations have precise mathematical definitions, and therefore the evaluation of $S = A \quad B$ is a well-defined mathematical problem. Here A, B, and hence S are *r*-sets, and is a regularized set operator (union, difference or intersection). We saw in Section 6.1 that a mathematical problem may have many computational versions, depending on the representations selected for the input and output. If the input and output representations are CSG trees, the problem is trivial: the output CSG tree consists simply of the input subtrees joined by a new operator node. As a more interesting example, there are efficient algorithms for computing Boolean operations when the input and output are represented by octrees. In this text, we focus on Boolean opeartion calculations that involve BReps, because these are the most common and important versions of the general problem.

The algorithms to be discussed are all based on a fundamental fact: boundaries may be destroyed, but they cannot be created by Boolean operations. In precise terms, this can be expressed as follows.

Boundary Inclusion Relationship:

$$(A \quad B) \quad A \quad B.$$

This is easy to prove rigorously. It implies that the BRep of a solid S that results from one or more Boolean operations can be computed by first generating a superset of its boundary as the union of the boundaries of the sets being combined, and then discarding those portions of the superset that are not *on* S. This is an instance of a commonly-used algorithmic paradigm called *generate-and-test*.. In pseudo-code the algorithm is as follows.

Algorithm 6.5.1.1

```
Generate a sufficient set of tentative faces F
for each F do {
FwrtS = ClassFaceSol(F,S);
AddToBRep(Fons); }
```

Generation of tentative faces is guided by the boundary inclusion relationship discussed above. Testing is done by set membership classification. Not shown in the pseudo-code

above (and in the algorithms to be discussed in the next subsections) is a clean-up phase, in which adjacent faces lying on the same surface are merged, and links required by the specific BRep scheme being used are established. (It is possible to clean up as you go, by using a sophisticated `AddToBRep` routine. This makes the code harder to understand, and will be ignored in this text.)

A sufficient set of faces is any set that includes S. The boundary inclusion relationship immediately suggests that we use the faces of A and the faces of B to compute the BRep of $S = A \quad B$. If we do that, we have a boundary merging problem, which can be defined as follows.

Boundary Merging:

Given: BRep of A and BRep of B
Find: BRep of $S = A \quad B$

Suppose now that A and B have CSG representations. Applying the inclusion relationship recursively down each subtree, we conclude that the set of all the faces of all the primitives in the CSG representation of S also is a sufficient set. If we use this set we have a non-incremental boundary evaluation problem.

Non-Incremental Boundary Evaluation:

Given: CSG representation of S
Find: BRep of S

This is a pure representation-conversion problem, in which we convert a CSG tree into a BRep. A variation on this problem consists of converting a hybrid CSG/BRep into a BRep. Suppose that we represent S by a CSG tree in which the leaves are not primitive. Instead, they are solids represented by BReps. Now we use the faces of the leaf BReps as the set of tentative faces. (This approach was used in the boundary evaluator of the PADL-2 system.) Boundary merging is a special case of this problem in which the right and left subtrees are leaves, represented by BReps.

Non-incremental boundary evaluation can be useful, for example, to de-archive a stored object defined in CSG. However, it is inadequate to support interactive design. Suppose that a user is constructing an object by applying successive Boolean operations. A good user interface must show the user the result of each operation. This is normally done by computing a BRep and generating a graphic display from the BRep. If boundary evaluation were non-incremental, each small change in the object would cause a complete re-evaluation of the boundary—a slow and inefficient approach. What is needed is a method for updating the boundary incrementally as the user constructs the object.

Incremental Boundary Evaluation:

Given: CSG representation of S, and BReps of its subtrees A and B
Find : BRep of S

Note that we could solve the incremental boundary evaluation problem by recursive application of a boundary merging algorithm, or we could ignore intermediate BReps and compute the final BRep for S non-incrementally (and slowly). But there are non-incremental evaluation methods that exploit both boundary information and CSG classification algorithms, and are worth studying.

Algorithm 6.5.1.1 could be implemented by writing a suitable face classifier. However, it is more profitable to look directly for the edges of the resulting solid. These edges are the primary components of the face representations that appear in the BRep of S. The next subsections discuss edge-based generate-and-classify algorithms for non-incremental boundary evaluation anf for boundary merging.

6.5.2 The Generate-and-Test Paradigm for Edges

The basic algorithm is similar to the face-based generate-and-test Algorithm 6.5.1.1. We generate a sufficient set of tentative edges and then classify them. There is a subtlety. All we need to do in the face-based algorithm is to extract the *on* subset of each face, but now there may be portions of curves that are *on* S and yet they are not edges of the BRep of S. The example in Figure 6.5.2.1 illustrates the issue. We assume that our BRep's faces are c-faces, i.e., connected, maximal subsets lying on primitive surfaces. In the figure we subtract the block B from the L-shaped object A. Edge E is *on* S but it is not an edge of the face F of S. It lies in the interior of th F. These edges can be immediately discarded by checking their neighborhoods. If the neighborhood indicates that the sector filled with material is bounded on both extremes by the same surface, then the edge is on the boundary of S but is not the boundary of a face of S. The generate-and-test algorithm for edges follows.

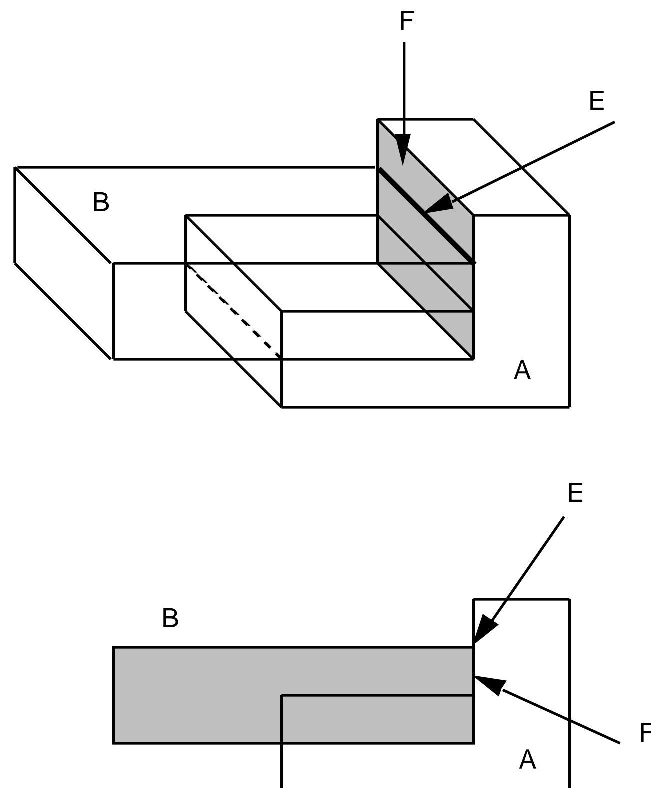

Figure 6.5.2.1 – The tentative edge E is *on* $S = A -^* B$ but is not an edge in its BRep. (3-D view at the top, and section view at the bottom.)

Algorithm 6.5.2.1

```
Generate a sufficient set of tentative edges E
for each tentative edge E do {
 EwrtS = ClassCurveSol(E,S);
 if not SingleSurf(NbhdSol(EonS,S)) then AddToBRep(EonS); }
 // SingleSurf is true if the neighborhood is bounded
 // by the same surface on both extremes.
```

Here we are again ignoring the clean-up phase, in which the resulting edges are properly linked to the BRep structure, and so on. Note that it is very easy to group the edges into potential faces simply by remembering on which surfaces they lie. This information is available from the edge generation phase, as we will see shortly.

Two issues remain to be addressed: how to generate tentative edges, and how to classify them. The second was discussed earlier, in the section on curve/solid classification. We can use edge classification procedures based either on CSG or BReps, depending on which of these representations is available.

Unlike faces, new edges can be generated by Boolean operations. Because of the boundary inclusion relationship, however, any edge of the resulting solid must be a subset of one or more of the faces in a sufficient set of tentative faces. Figure 6.5.2.2 shows that the edges of a Boolean composition are either (i) subsets of the edges of the objects being combined or (ii) the intersection of two faces from different objects. We call the first *self-edges*, and the second *cross-edges.* Therefore we generate tentative edges by collecting all the self-edges of the objects being combined, and intersecting pairs of their faces to produce cross-edges. In practice, intersecting faces is difficult, because the faces can be of arbitrary complexity. Instead, we can generate an over-sized cross-edge, for example, by intersecting two supersets of the faces. For planar faces the simplest supersets are rectangles. These are very easy to intersect and, provided that each encloses a face, produce a tentative edge that is larger than necessary. This does not affect the correctness of the algorithms because tentative edges are classified and only their *on* subsets are output.

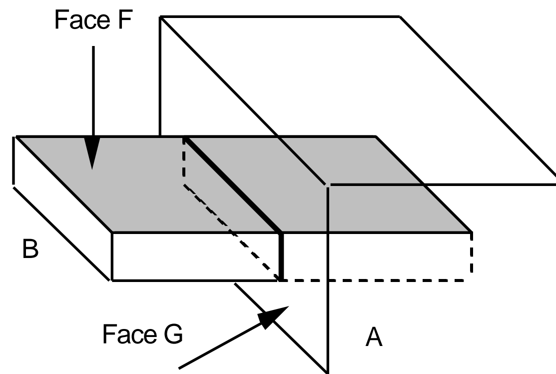

Figure 6.5.2.2 – Tentative edge generation

We are now ready to present simple but complete algorithms for Boolean operations. In the sequel, we assume that BReps contain face and edge neighborhoods.

6.5.3 Incremental Boundary Evaluation

We are given both a BRep and a CSG tree for two solids A and B and we want to compute the BRep of $S = A \quad B$. We consider as tentative faces the faces of A and the faces of B. Thus, the edges of A and B are the self-edges in the tentative edge generation step. Cross-edges are produced by intersecting supersets of the faces of A with supersets of the faces of B. These supersets can be, for example, faces of primitives, which normally are simple shapes.

After the tentative edges are generated, they must be classified with respect to S. We do this by using the CSG representation of S and the classification algorithms for CSG. Note that the classification of the self-edges of A with respect to A is known: the edges are *on* A and their neighborhoods are available in the BRep of A. Therefore we simply classify them with respect to B and combine the results to get the classification with respect to S. Similarly, the self-edges of B can be classified only with respect to A and the results combined with the known classifications with respect to B.

A complete algorithm in pseudo-code follows. (This algorithm is a simplified version of PADL-2's boundary evaluator.)

Algorithm 6.5.3.1

```
for each face F of A do { // Self-edges of A
 for each self-edge E of F do {
  EwrtA = (E, NbhdSol(E,A)); // Read from the BRep of A
  EwrtB = ClassEdgeSolC(E,B); // Use CSG classification
  EwrtS = CombClassEdgeSolC(EwrtA, EwrtB,  );
  if not SingleSurf(NbhdSol(EonS,S)) then AddToBRep(EonS);
  }
 }

for each face F of B do { // Self-edges of B
 for each self-edge E of F do {
  EwrtB = (E, NbhdSol(E,B)); // Read from the BRep of B
  EwrtA = ClassEdgeSolC(E,A); // Use CSG classification
  EwrtS = CombClassEdgeSolC(EwrtA, EwrtB,  );
  if not SingleSurf(NbhdSol(EonS,S)) then AddToBRep(EonS);
  }
 }

for each face F of A do { // Cross-edges
 for each face G of B do {
  if not Surf(F) == Surf(G) then {
   E = F'  G'; // Use supersets: F'  F, G'  G
   EwrtS = ClassEdgeSolC(E,S)
   if not SingleSurf(NbhdSol(EonS,S)) then AddToBRep(EonS);
   }
  } // B face loop
 } // A face loop
```

This algorithm has obvious inefficiencies, which are easy to fix. For example, cross-edges are computed twice.

6.5.4 Boundary Merging

Now we are given the BReps of solids A and B and we want to compute the BRep of $S = A \otimes B$. We can use Algorithm 6.5.3.1 provided that we replace the edge/solid classification routines with their BRep counterparts. These were discussed earlier, in Section 6.3.3.

Instead of classifying the cross-edges with respect to the two solids A and B by using BRep algorithms, we can use face/edge operations, as shown in Figure 6.5.4.1. Observe that the intersection of the two faces F and G in the figure consists of two segments. One is a self-edge of G and the other is the intersection of the 2-D *interiors* of F and G, regularized in 1-D. The self-edge need not be considered here because it is processed within the self-edge loops of the algorithm. Therefore, what we need to compute is

$$E = r_1(i_2F \cap i_2G),$$

where the interiors are in the 2-D topologies of the host surfaces of the faces, and the regularization is in the 1-D topology of the host line of the cross-edge C. Now

$$i_2F \cap i_2G \subset C \Rightarrow (i_2F \cap C) \cap (i_2G \cap C) = i_2F \cap i_2G.$$

Regularizing and using the standard notation for classification results yields

$$E = r_1((i_2F \cap C) \cap (i_2G \cap C)) = CinF \cap_1^* CinG,$$

where the regularized intersection is in 1-D.

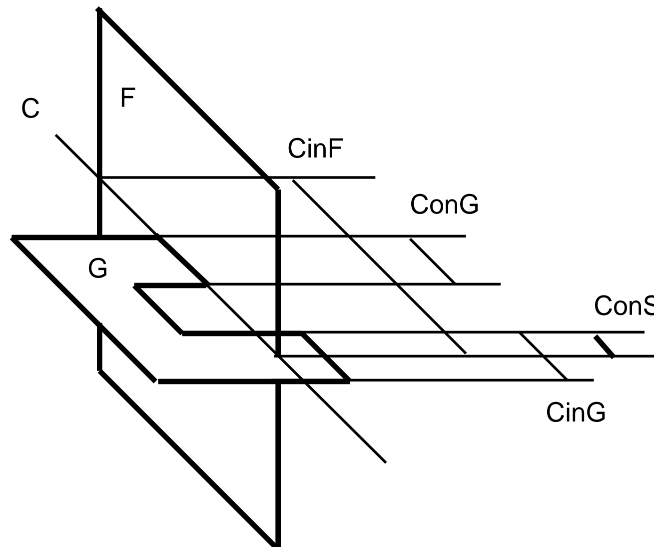

Figure 6.5.4.1 – Edge/face operations

Therefore the procedure is a follows. First we generate an oversized cross-edge C as before, *i.e.*, by intersecting supersets of the two faces F and G under consideration. Then we classify the edge with respect to each of the faces, and combine the results by

intersecting the two *in* subsets. We classify the cross-edge C with respect to faces F and G by using a 2-D edge/face classifier for faces represented by their boundaries. Note that the edge E thus computed always has a partially full neighborhood with respect to$S$, regardless of the Booelan operator used to combine A and B. In other words, E is an edge of S. Therefore $E = ConS$, and we don't need to further classify it.

A complete algorithm for boundary merging that uses the face/edge procedures just described follows.

Algorithm 6.5.4.1

```
for each face F of A do { // Self-edges of A
 for each self-edge E of F do {
  EwrtA = (E, NbhdSol(E,A)); // Read from the BRep of A
  EwrtB = ClassEdgeSolB(E,B); // Use BRep classification
  EwrtS = CombClassEdgeSol(EwrtA, EwrtB,  );
  if not SingleSurf(NbhdSol(EonS,S)) then AddToBRep(EonS);
  }
 }

for each face F of B do { // Self-edges of B
 for each self-edge E of F do {
  EwrtB = (E, NbhdSol(E,B)); // Read from the BRep of B
  EwrtA = ClassEdgeSolB(E,A); // Use BRep classification
  EwrtS = CombClassEdgeSol(EwrtA, EwrtB,  );
  if not SingleSurf(NbhdSol(EonS,S)) then AddToBRep(EonS);
  }
 }

for each face F of A do { // Cross-edges
 for each face G of B do {
  if not Surf(F) == Surf(G) then {
   C = F'  G'; // Use supersets: F'  F, G'  G
   CwrtF = ClassEdgeFaceB(C,F);
   CwrtG = ClassEdgeFaceB(C,G);
   E = RegInt1(CinF,CinG);
   // Regularized intersection in 1-D. E = ConS
   NbhdSol(E,S) = CombNbhdSol(NbhdSol(E,A),NbhdSol(E,B),  );
   // The Nbhds of E with respect to A and B are copied from
   // the BReps of A and B
   if not SingleSurf(NbhdSol(E,S)) then AddToBRep(E);
   }
  } // B face loop
 } // A face loop
```

Cross-edge processing requires a `ClassEdgeFaceB` procedure, which is the 2-D analog of the `ClassEdgeSolB` procedure discussed in Section 6.3.3. Recall that `ClassEdgeSolB` invoked a transition evaluation routine, which used the neighborhoods with respect to the solid for the points in the intersection list. These neighborhoods were assumed to be stored in the BRep with the face representations. Similarly, `ClassEdgeFaceB` will need access to neighborhoods *with respect to faces* for the points of intersection between C and the edges of each face. These neighborhoods are not

normally stored in the BReps, but they can be inferred from their 3-D counterparts as follows—see Figure 6.5.4.2.

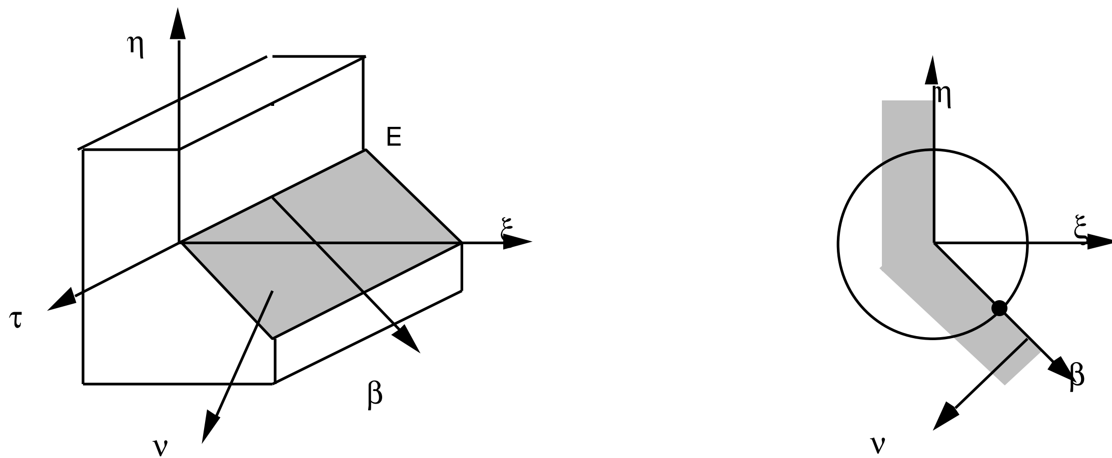

Figure 6.5.4.2 – Inferring a 2-D neighborhood from a 3-D neighborhood

We represent the neighborhood of the edge E with respect to the solid as explained in Section 6.4.1—see Figure 6.4.1.3—by an angular sector (or, equivalently, an arc of the unit circle) on the plane normal to the edge. The neighborhood of E with respect to the shaded face F is represented by a side bit relative to the binormal β, also as explained in Section 6.4.1, Figure 6.4.1.6. To compute this side bit we intersect the binormal with the unit circle. If the intersection is on the boundary of the angular sector that corresponds to the 3-D neighborhood of E, as shown in 2-D on the right of the figure, then β points towards the face F. Therefore the 2-D neighborhood of E with respect to F is 1. If β intersected the unit circle at an interior point of the angular sector, then the side bit would be 0.

Cross-edges can be inferred from self-edges at considerable computational savings, *for polyhedral solids*. For example, in Figure 6.5.4.1 the vertices that bound the cross-edge E = *ConS* are intersections of either self-edges of F with face G or of self-edges of G with face F. It is easy to see that any vertex of a Boolean combination of two polyhedra (and hence any cross-edge vertex) must lie in a self-edge of one of them. The proof is as follows. A vertex of a polyhedron $S = A \quad B$ must lie in at least three faces of S. Because of the boundary inclusion relationship, these faces are subsets of the faces of A and B. Since there are three faces and only two solids, a vertex must lie in at least two faces of one of the solids A or B, and therefore it must lie in a self edge of that solid.

6.5.5 Low-Level Routines

Let us summarize here the most important computational utilities required by the Boolean operation algorithms discussed in the previous sections for general, curved solids.

- Surface/surface intersections for computing cross-edges.

- Curve/surface intersections. These are needed by edge/solid classifiers, regardless of whether they are based on boundaries or CSG. Edge/solid classification for BRep solids starts by intersecting the tentative edges with the host surfaces of the faces. For CSG, the edge/primitive classifiers must likewise intersect the edge with the surfaces of the primitives.

To be continued…