A REPRESENTATION OF SOLID OBJECTS FOR PERFORMING BOOLEAN OPERATIONS

Technical Report 86-006

Sandra H. Bloomberg

The University of North Carolina at Chapel Hill
Department of Computer Science
New West Hall 035 A
Chapel Hill, N.C. 27514

A polyhedron can be represented as a binary tree whose nodes contain oriented planes containing the faces of the polyhedron. The set of null sons represents a partition of three-space into convex components, distinguishing the inside and outside of the polyhedron. This representation allows simple algorithms for performing regularized union, intersection, and difference of polyhedra, and for determining if a point is contained in a polyhedron.

A Representation of Solid Objects for Performing Boolean Operations

Sandra Bloomberg

University of North Carolina at Chapel Hill

The Boolean operations union, intersection, and difference are encountered frequently in the computer design and manipulation of solid objects. These operations are necessary for obtaining a boundary representation from a Constructive Solid Geometry (CSG) representation, and are useful for simulating manufacturing processes such as milling and for detection of spatial interference in applications like VLSI and CAD/CAM. The procedures for performing Boolean operations tend to be quite complicated, and have been called the "... most complex and delicate software modules in a solid modeler" [Requicha and Voelcker, 1985] . This paper presents a representation for polyhedra with which Boolean operations are simple. This representation also allows simple algorithms for classifying points and lines (as inside or outside of an object). The representation may be extended to solid objects whose faces are more general than polygons but meet criteria given in Section 4.

There are many algorithms in use for performing Boolean operations. It is not too difficult to come up with an algorithm that works most of the time; the hard part is dealing with exceptional cases such as when two solids have overlapping faces or when a vertex or edge of one polyhedron lies within a face of the other polyhedron. The most attractive aspect of the representation and algorithms presented here is that they eliminate the need for special consideration of these cases.

Section 1 of this paper describes the tree representation, called a planetree. Section 2 tells how to determine if a point, line, or polygon is contained in a solid object. Section 3 gives algorithms for finding regularized intersection, union, and difference, and

indicates how to deal with overlapping faces. Section 4 discusses general concerns about the representation, including the size of a planetree, converting a planetree to a boundary representation (getting faces of a subdivision of the object and its complement into convex subparts along the way), performing sequential Boolean operations, and extending the representation to more general solids. Section 5 gives results.

In this paper, a *polyhedron* is a solid three–dimensional object whose boundary is an (orientable) surface made up of planar polygonal faces which intersect only at edges and vertices. It is not necessary that the polygonal faces be convex, but I will assume the faces are convex to simplify descriptions.

1. The Planetree

A planetree based on a polyhedron is a binary tree whose nodes contain the oriented planes containing the faces of the polyhedron. The orientation can be chosen so that the normal vector to the plane points away from the polyhedron. Call the open half–space pointed to by the normal vector the *outside* of the plane; call the other open half space the *inside* of the plane. For a given node in a planetree, the right subtree represents a subdivision of a certain subspace on the inside of that node's plane, and the left subtree represents a subdivision of a certain subspace on the outside of the plane. We shall see that the set of null sons represents a partition of three-space into convex polyhedral parts (not all bounded). The null right sons are associated with regions inside the polyhedron, while the null left sons correspond to the regions outside of the polyhedron.

A convex polyhedron could be represented by a list of the oriented planes containing faces of the polyhedron, so that the interior of the polyhedron is the space that lies on the inside of all of these planes. A planetree for a convex polyhedron with n faces is a one dimensional tree of height n, the maximum height of a planetree for a polyhedron with n faces, as in Figure 1. (Figure 1 really shows a tree for a polygon and a partition of two–space. I will continue to refer to "planes containing faces of a polyhedron" as in 3–d, while the figures will actually show the 2–d analog, "lines containing edges of a polygon". In the figures, lower case letters are faces, upper case are the corresponding planes. So, A is the plane containing face a, for example.) The space partition for a convex polyhedron with n faces will have n+1 components: one will be the interior of the polyhedron, and the rest of space will be divided into n parts. Each node of the planetree will have a null left son, and all but the single leaf node will have a non-empty right son. In the example, the root node has a null left son because none of the polyhedron's interior is outside of plane

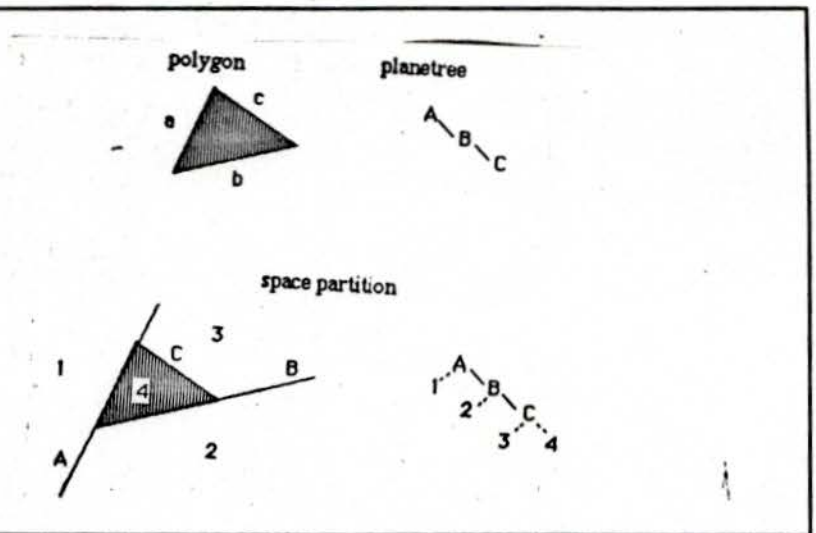

Figure 1: A convex polygon, its tree, and the corresponding partition of space. (A is the line containing edge a. In 3–d, A would be the plane containing face a.)

A. The null left son of the node labelled A corresponds to the half–space outside of plane A, region 1 in the figure. The right subtree of the root node represents the half–space inside of plane A. The null left son of the node labelled B corresponds to the space that is both inside of plane A and outside of plane B, region 2. The interior of the polyhedron is region 4, which is inside planes A, B, and C.

The subspace associated with a null son is the intersection of half–spaces determined by all the planes in nodes between the root and the null son, and so must be convex. At each node between the root and the null son, if you travel down a right son to go from the node toward the null son, use the half–space on the inside of that node's plane. If you travel down a left son to go toward the null son, use the outside of that node's plane. The union of all the subspaces associated with null sons is all of three–space.

When the polyhedron is not convex, its interior may not lie entirely on a single side of a face–containing plane. Thus, the resulting planetree is no longer linear. Figure 2 shows one possible planetree for a nonconvex polyhedron and the corresponding space partition. The left subtree of the node labelled D represents a division of the part of space that is both inside planes A, B, and C, and outside plane D (regions 4 and 5 in the space partition of Figure 2), while the null right son of the node labelled D, region 6, is associated with the region of space inside planes A, B, C, and D.

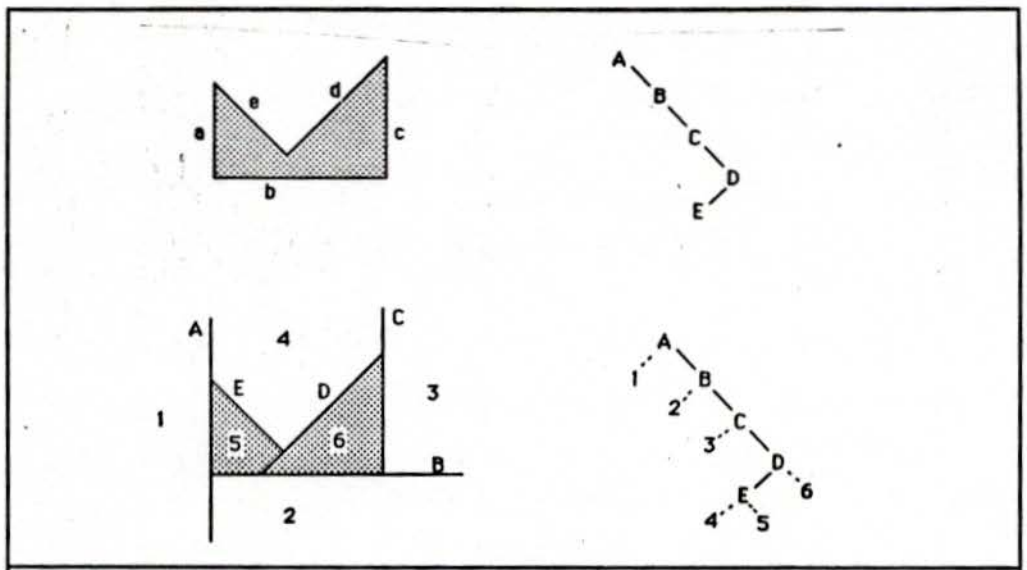

Figure 2: A nonconvex polygon, its tree, and the corresponding partition of space when the edges are ordered **a**–*b*–*c*–*d*–*e*.

The procedure for building a planetree for a polyhedron uses the polyhedron's polygonal faces for comparison, but planetree nodes contain only the oriented plane containing a face. Each node of a planetree contains the coordinates c_1, c_2, c_3, c_4 of the plane equation $c_1x + c_2y + c_3z + c_4 = 0$ where the vector (c_1, c_2, c_3) points toward the outside of the plane.

The tree–building procedure, and other operations in the rest of this paper, repeatedly call for finding the part of a polygon on a given side of a plane. Clipping a polygon against a plane can be done easily and efficiently using the Sutherland/Hodgman reentrant clipping algorithm [Sutherland and Hodgman, 1974]. If the polygon is convex, the procedure is particularly straightforward. The result will be a single convex polygon. If an edge of the polygon lies within the clipping plane, the result will be either the entire original polygon, or will have fewer than three vertices and so can be discarded.

To build a tree for a polyhedron, list the faces in any order and, for each face, insert the plane containing that face into the tree. Let P be the plane containing face p. P is inserted into an empty tree by placing it into the root node. To insert P into a nonempty tree, use face p by using the plane in the root node to divide p into two pieces: the part outside of the root's plane, $p_{outside}$, and the part inside of the root's plane, p_{inside}. If p_{inside} is nonempty, use p_{inside} to insert P into the right subtree. If $p_{outside}$ is nonempty,

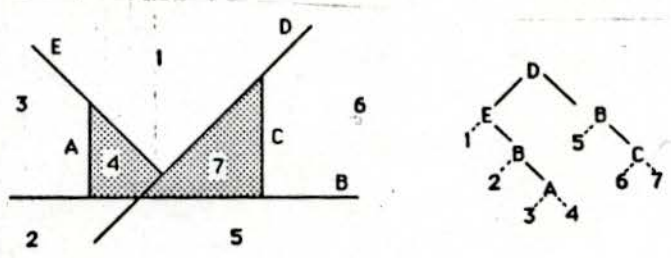

Figure 3: Regions formed by changing the order of the faces to *d*–*e*–*b*–*a*–*c*.

use $p_{outside}$ to insert P into the left subtree. If the face, p, is contained in the root's plane, P need not be added to either subtree.

This algorithm for building a planetree is given in the following routine, **buildtree**. Assumed available is a subroutine, divide($R, p, p_{inside}, p_{outside}$), which takes a plane, R, and a polygon, p, and finds the parts of p on the inside, p_{inside}, and on the outside, $p_{outside}$, of R.

```
buildtree(polyhedron, tree)
    let tree be an empty tree
    for (each face, p, of polyhedron)
        insert(p, tree)

insert(p, tree)
    if (tree is empty)
        put the plane containing p into tree
        return
    R = plane in root node of tree
    divide(R, p, p_inside, p_outside)
    if (p_inside not empty)
        if (tree → rightson is NULL) create a rightson containing P
        else insert(p_inside, tree → rightson)
    if (p_outside not empty)
        if (tree → leftson is NULL) create a leftson containing P
        else insert(p_outside, tree → leftson)
```

For a non-convex polyhedron, the size and shape of the tree depends upon the order in which face-containing planes are added to the tree. Figure 3 shows the resulting planetree and subregions when the faces of the polyhedron of Figure 2 are ordered *d*–*e*–*b*–*a*–*c*. Notice that the left subtree of the node containing E is empty, even though a part of face b is outside of E, and B is added after E. That is because only the part of b that is outside of D is used for comparison with E in building the tree.

2. Classifying Points, Lines, and Polygons

Classifying points, line segments, and polygons with respect to a polyhedral region can be done using the planetree representation of the polyhedron. To compare a point to a planetree, compare the point with the plane at the root. If the point is inside the root's plane, it must lie in the region represented by the right subtree, so pass the point down to the right subtree; if the point is outside, pass the point down to the left subtree. Now compare the point with the appropriate subtree. If the point is passed down to a null left subtree, the point is outside the polyhedral region; if a null right subtree is encountered, the point is inside the region. (Choose a point and try it for the trees in the figures.)

For a point that is coplanar with a face, the situation is slightly messier because that point could be inside, outside, or on the surface of the polyhedron. When a point lies within a node's plane, pass the point down to both left and right subtrees. A point lying on the surface of the polyhedron will be eventually passed to both a null left son and a null right son. Even if a point does not lie on the surface, but is contained in a plane of the tree, the point may be passed down to more than one null son. In this case, if the point is inside the polyhedron, it will always be passed to null right sons, and if it is outside, it will always be passed to null left sons.

The point classification procedure described above is given in the following code:

```
classifypoint(point, tree)
    R = oriented plane in root node of tree
    if (point is inside or on R)
        if (tree → rightson is NULL) print("point is inside")
        else classifypoint(point, tree → rightson)
    if (point is outside or on R)
        if (tree → leftson is NULL) print("point is outside")
        else classifypoint(point, tree → leftson)
```

Comparing a line segment with a planetree determines which parts of the line segment are inside a polyhedral region. At each node, compare the line segment to the plane at that node. Pass the *part* of the line segment that is inside the plane down to the right subtree, and the *part* of the line segment that is outside the plane down to the left subtree. If only one endpoint of the line segment is within a plane, the line is considered to be completely inside or completely outside of that plane, depending upon where the other endpoint lies. If the whole line segment lies within a plane, it is considered to be both inside and outside that plane. Pieces of the line inside the polyhedron will eventually be passed to a null right son, while pieces of the line that are outside the polyhedron will eventually be passed to

a null left son. The results are not guaranteed to be minimal; that is, a line segment that is completely within a polyhedron could, for example, be divided into multiple pieces, but all of these pieces would be labelled as being inside the polyhedron by being passed to null right sons. When a line segment lies within a plane of the tree, there may be overlapping pieces in the result, and sections on the boundary will be both inside and outside the polyhedron.

A similar algorithm is used to compare a polygon with a planetree. At each node, divide[1] the polygon into pieces inside and outside of the node's plane. If the right subtree is empty, this inside part of the polygon (if any) is inside the polyhedron. Otherwise, pass the inside part of the polygon (if any) down to the right subtree. If the left subtree is empty, the outside part of the polygon (if any) is outside the polyhedron. Otherwise, pass the outside part of the polygon (if any) down to the left subtree. As in the case of a line segment, the answer may be in pieces. This is illustrated in Figure 4. Again, when a polygon is contained in a plane of the planetree, the polygon needs to be considered to be both inside and outside of a plane it lies within. Thus, there may be overlapping pieces in the output, and sections that are on the boundary will be indicated as both inside and outside the polyhedron. The algorithms for Boolean operations will be able to avoid generating overlapping faces.

3. Intersection, Union, and Difference

Consider the intersection of two solid polyhedra, A and B, where each polyhedron is defined by a list of oriented planar polygonal faces. The problem is to find the polygonal faces of the surface of the intersection. Each face of the intersection, $A \cap B$, is contained in one of the faces of A or B. Thus, the faces of the intersection are just the parts of the faces of A that are inside of B and the parts of the faces of B, that are inside of A. So, the problem of intersection can be done by comparing each polygonal face of A to the planetree for B, and comparing each face of A to the planetree for B. To find the parts of a polygon inside the polyhedron represented by a planetree, do a comparison similar to that described in Section 2, keeping pieces passed to a null right son, and discarding pieces passed to a null left son[2].

This same idea works for other Boolean operations. To find the union of two polyhedra, $C = A \cup B$, it is still true that each face of C is contained in a face of A or one of B. In

[1] Again, the Sutherland/Hodgman clipping algorithm works well here.
[2] I am ignoring overlapping faces temporarily.

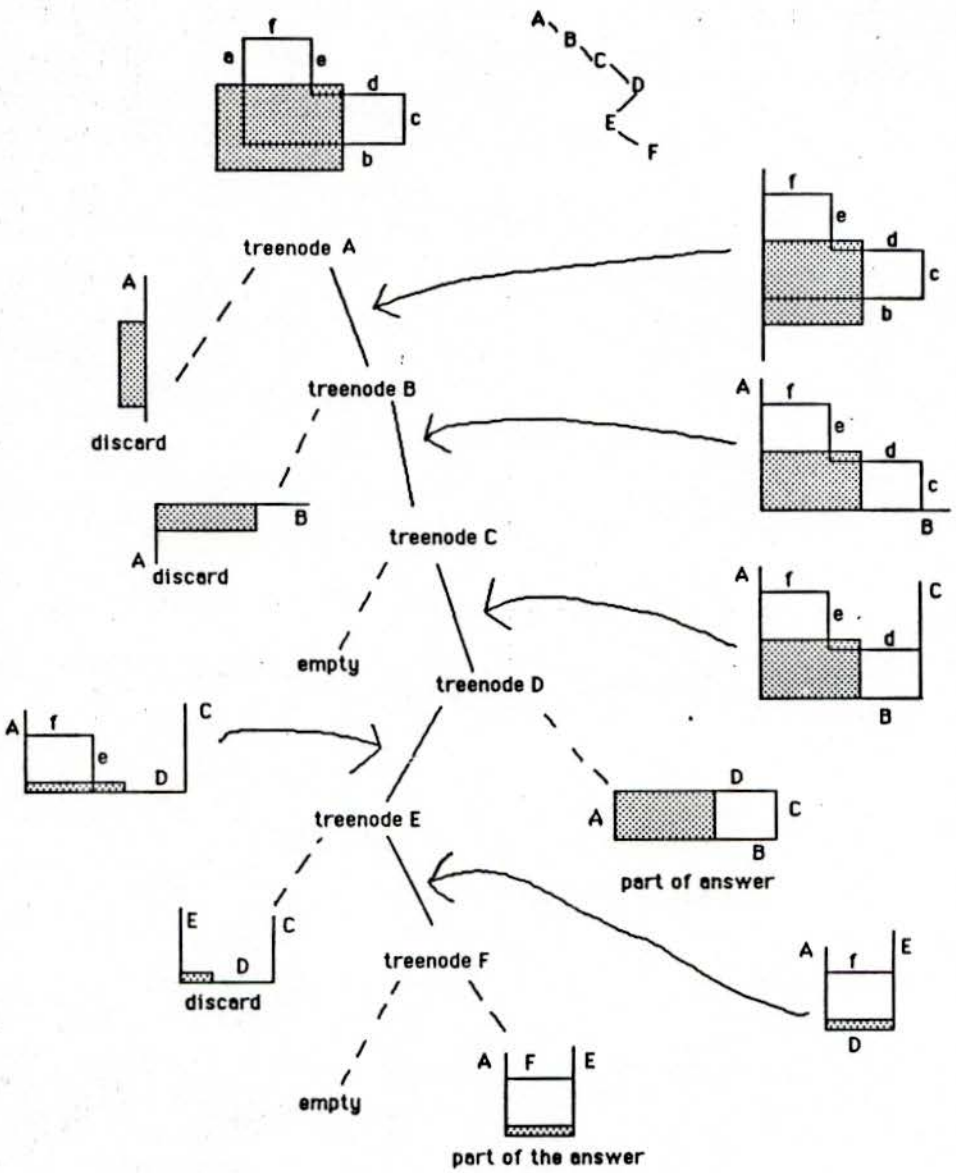

Figure 4: These are the steps in finding the parts of the shaded rectangle that are inside of the L-shaped region. The planetree for the L-shaped region was built from faces ordered *a-b-c-d-e-f*. To get an idea of the 3-d situation, think of the shaded rectangle as a polygon which lies within the plane of this page. Think of the L-shaped region as if it were really a polyhedron in 3-d extending above and below the page. *A* is then a plane that intersects the page along the line containing *a*.

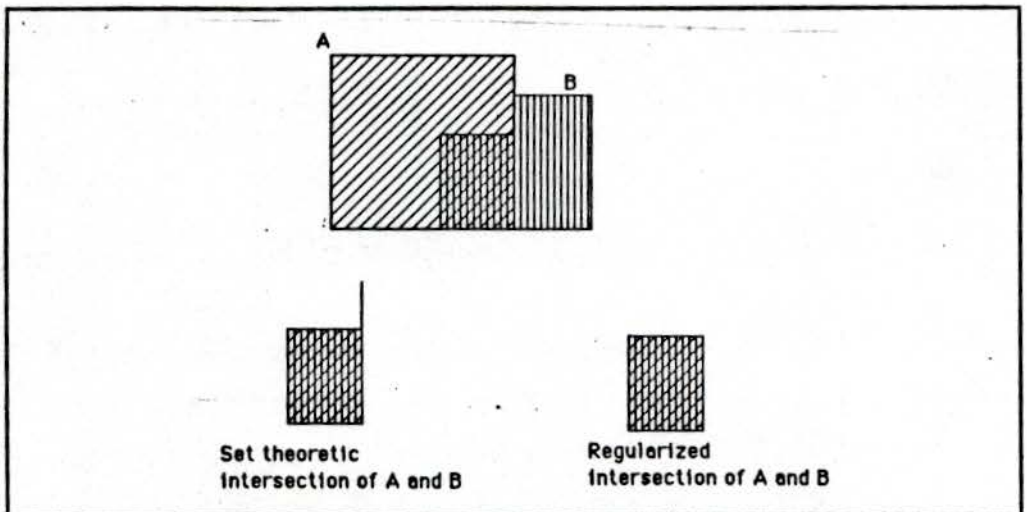

Figure 5: (a) Two closed sets, A and B. (b) The intersection of A and B. (c) The "regularized" intersection of A and B.

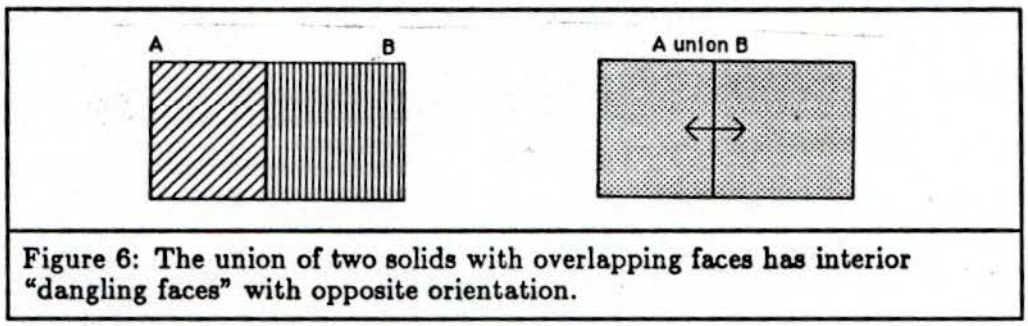

Figure 6: The union of two solids with overlapping faces has interior "dangling faces" with opposite orientation.

the union case, however, the faces of C are the parts of the faces of A that lie outside of B and the parts of the faces of B that lie outside of A. When doing the polygon to planetree comparison to find the parts of the polygon outside of a polyhedron, discard parts passed to a null right son and keep parts passed to a null left son. To find the difference of two polyhedra, $C = A - B$, we need the faces of A outside of B and the faces of B that are inside A. Again, this can be done by polygon/planetree comparisons.

The hard part of doing Boolean operations cleanly is dealing with exceptional cases. The special cases of two polyhedra intersecting at a vertex or an edge disappear with the Sutherland/Hodgman clipping, so the only concern is for coplanar overlapping faces. In the previous section, a polygon is assumed to be both inside and outside of a plane it lies within. When doing Boolean operations, this assumption includes unnecessary faces which are undesirable for many applications. For example, faces of the intersection of two

polyhedra resulting from overlapping faces of the original polyhedra will be contributed twice. This assumption also yields the "dangling faces" resulting from nonregularized[3] intersection, as illustrated in Figure 5. The result of the union of two adjacent polyhedra will include faces in its interior, as in Figure 6.

To avoid generating overlapping faces, a polygon being compared to a planetree cannot be considered to be both inside and outside a plane it lies within. A polygon that is a face of a polyhedron is oriented like the plane it is contained in; the *inside* normal points into the polyhedron, and the *outside* normal points away from the polyhedron. This orientation is unique because faces are only allowed to intersect at edges and vertices. When comparing an oriented polygon to a planetree, whether the polygon should be considered inside or outside of a plane it lies within depends upon the operation being done and whether or not both the plane and the polygon have the same orientation. When finding the part of a polygon that is inside or outside of a polyhedron for performing a Boolean operation, the classifying routine needs to know how to classify a polygon that lies within a plane. The remainder of this section presents appropriate classifying routines, and programs for performing Boolean operations.

The routine **findinside**(p, *tree*, *insideorientation*) finds the part of a polygon, p, inside of a polyhedron represented by a planetree, *tree*. The subroutine **sameplane**(R, p) returns TRUE if the plane, R, contains the polygon, p; returns FALSE otherwise. The subroutine **orientation**(R, p) returns SAME if the plane, R, and the coplanar polygon, p, have the same orientation; returns OPPOSITE otherwise. As before, the subroutine **divide**($R, p, p_{inside}, p_{outside}$) takes a plane, R, and a polygon, p, and finds the parts of p on the inside, p_{inside}, and on the outside, $p_{outside}$, of R. The **findinside** routine determines what parts of a polygon are inside an object by repeatedly determining which part is inside the plane in a node of the tree. In **findinside**, the argument *insideorientation* indicates when a polygon should be considered to be inside of a plane it lies within (= is coplanar with). Its values may be SAME, OPPOSITE, or NONE. If *insideorientation* is SAME, a polygon is considered to be inside a plane with the same orientation, and outside a plane with opposite orientation. If *insideorientation* is NONE, a polygon always outside the plane it lies within. If *insideorientation* is OPPOSITE, a polygon is inside a plane with the opposite orientation, and outside a plane with the same orientation.

[3] A set resulting from a regularized operation is equal to the closure of its interior. (See [Requicha and Voelcker, 1984].)

```
findinside(p, tree, insideorientation)
    R = plane in root node of tree
    if (sameplane(R, p))
        if (orientation(R, p) = insideorientation)
            p_inside = p
            p_outside = empty
        else
            p_outside = p
            p_inside = empty
    else divide(R, p, p_inside, p_outside)
    if (p_inside not empty)
        if (tree → rightson is NULL) add p_inside to the result
        else findinside(p_inside, tree → rightson, insideorientation)
    if (p_outside not empty)
        if (tree → leftson is NULL) do nothing
        else findinside(p_outside, tree → leftson, insideorientation)
```

The next routine, **findoutside**, finds the part of a polygon outside of the polyhedron represented by a planetree. It is very much like **findinside**, except in this case, since we are looking for the part of the polygon outside of the planetree, the argument *outsideorientation* indicates when a polygon should be considered to be outside of a plane it lies within.

```
findoutside(p, tree, outsideorientation)
    R = plane in root node of tree
    if (sameplane(R, p))
        if (orientation(R, p) = outsideorientation)
            p_outside = p
            p_inside = empty
        else
            p_inside = p
            p_outside = empty
    else divide(R, p, p_inside, p_outside)
    if (p_outside is not empty)
        if (tree → leftson is NULL) add p_outside to the result
        else findoutside(p_outside, tree → leftson, outsideorientation)
    if (p_inside is not empty)
        if (tree → rightson is NULL) do nothing
        else findoutside(p_inside, tree → rightson, outsideorientation)
```

Let A and B be polyhedra. To find the intersection, we need to find the faces of A that are inside B, and the faces of B that are inside A. This amounts to two calls to **findinside**, and the only question is how to set *insideorientation*. Consider Figure 7. The inside of $A \cap B$ was originally inside of both A and B, so we want parts of original overlapping faces that had the same orientation. Thus, we set *insideorientation* to SAME for finding

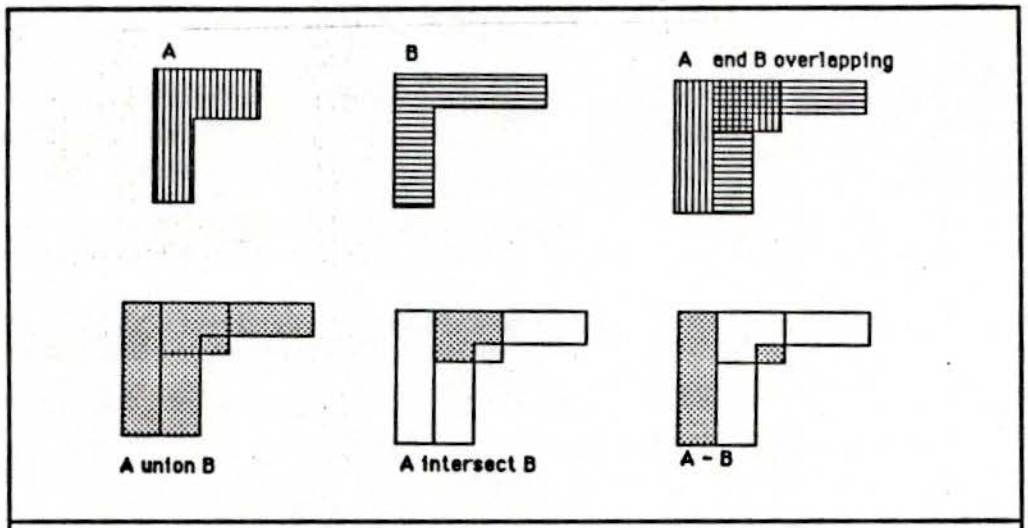

Figure 7: Union, intersection, and difference of objects with overlapping boundaries.

the faces of A that are inside of B. Since we will have all the parts of overlapping faces we need from A, we set *insideorientation* to NONE when finding faces of B inside of A.

```
intersect(A, B)
    buildtree(B, Btree)
    for each face, a, of A
        findinside(a, Btree,SAME)
    buildtree(A, Atree)
    for each face, b, of B
        findinside(b, Atree,NONE)
```

Union is very similar to intersection. This time we want faces of A outside of B, and faces of B outside of A. Overlapping faces with the same orientation (= same "inside") will be part of the result; overlapping faces with opposite orientation would be internal in the result, so they should not be included. As before, we only keep overlapping faces from one of the original figures. So, here we set *outsideorientation* to SAME for one set of faces, and set it to NONE for the other set of faces.

```
union(A, B)
    buildtree(B, Btree)
    for each face, a, of A
        findoutside(a, Btree,SAME)
    buildtree(A, Atree)
    for each face, b, of B
        findoutside(b, Atree,NONE)
```

The difference, $A - B$, is the set inside of A, but outside of B. (This is not the same thing as $B - A$.) So this time, we want to keep overlapping parts of faces with opposite orientation. Intersection and union are symmetric operators, so in those cases it did not matter whether part of an original face of A or B was kept; we were always keeping faces with same orientation. In the difference case, for $A - B$, we need to keep the overlapping face from A, not B, so that the face in the result will have the correct orientation. If we were finding $B - A$, we would keep the faces of B instead. The next routine finds $A - B$.

```
difference(A, B)
    buildtree(B, Btree)
    for each face, a, of A
        findoutside(a, Btree,OPPOSITE)
    buildtree(A, Atree)
    for each face, b, of B
        findinside(b, Atree,NONE)
```

4. Other Properties of the Planetree Representation

This section deals with properties of the planetree representation and the algorithms for Boolean operations. Methods are given for determining when two planetrees represent the same solid, and for converting a planetree to a boundary representation of the solid or to a convex subdivision of the solid. Sequential Boolean operations, the size of the trees, and extensions to nonpolyhedral solids are discussed.

In general, a boundary representation for a polyhedron is not unique. Even the same set of faces can yield different planetree representations when the order of the faces is changed. If, for two given planetrees, there exists for each a clean boundary representation (= list of nonoverlapping faces), the difference operator can be used to test for equality. That is, A and B are the same solid if and only if $A - B$ is empty.

If a list of faces is not available, it is possible to convert a planetree representation into a boundary representation. The faces will all be contained in planes in the planetree, so first make a list of all the planes in the tree, including each plane twice, once for its own orientation and once for the opposite orientation. (Recall that the building procedure discards a plane that is already in the tree, even if it has the opposite orientation. Thus, we have to include both orientations to be sure to get all the faces.) For each oriented plane, first find the parts that are inside the polyhedron represented by the planetree using findinside, considering a face to be inside a plane of the same orientation (*insideorientation* = SAME). This yields all parts of planes that are inside or on the

boundary of the object. Since we do not want to include the parts of these faces that are strictly inside the object, next apply the routine **findoutside** to this new set of faces, considering a face to be outside a plane it is in (*outsideorientation* = SAME). The result will be a set of nonoverlapping faces that are the boundary of the polyhedron. It will not necessarily be the same set of faces that was used to build the tree.

The set of faces obtained halfway through the boundary finding process is a set of faces that are on or inside the original solid. These faces are the faces of a subdivision of the original object into convex subobjects. It is still necessary to divide this set of faces into subsets corresponding to subobjects in order to have a subdivision of the original object.

Lists of faces obtained from planetree operations have no adjacency information for faces. The 2D analogs of the polyhedron comparison algorithms thus have the disadvantage that the list of edges of the resulting polygon is no longer in order around the polygon, so the process of comparing boundary elements to trees must be followed with a tracking step if adjacency information is desired. Converting a CSG representation to a boundary representation can involve a long sequence of Boolean operations. Planetrees are built from a list of faces, so no adjacency information needs to be reconstructed in the intermediate steps. If repeated operations are being done on objects with many faces, the result may have many more faces.

It is not always necessary to do multiple Boolean operations sequentially. For example, $A \cup B \cup C$ can be found from the faces and planetrees of A, B, and C without computing an intermediate tree by finding the parts of faces of each outside of both of the other two. To find the part[s] of a face outside of both of two trees, first find the parts outside of one tree, then find the parts of that result that are outside of the second tree. To find $A \cup B \cup C$, find the parts of faces of A on the outside of B's tree and C's tree (use *outsideorientation* = SAME for both). Next, find the parts of faces of B that are outside of A's tree (*outsideorientation* = NONE) and outside of C's tree (*outsideorientation* = SAME). Finally, find the parts of faces of C that are outside of A's tree and B's tree (use *outsideorientation* = NONE for both).

The size of a planetree is similar to the size of the BSP–tree, introduced as part of a visible–surface algorithm [Fuchs *et al.*, 1980]. The size of the BSP–tree was addressed in an article by Fuchs, Abram, and Grant [Fuchs *et al.*, 1983], where they present evidence that the number of nodes need not be much greater than the number of faces. To minimize the number of nodes in the tree, they first add faces that do not "split" many other faces.

Nodes of a BSP–tree contain a list of vertices of a polygon, while nodes of a planetree contain the coefficients of a plane equation. Coplanar faces need only be represented by one plane in a planetree.

If the tree representation is to be used for objects whose faces are not planar polygons, the form of the faces must allow three computations. You need to be able to find the oriented surface containing a face, decide which side of such a surface another face is on, and clip a face against a side of a surface. (For non–convex faces, this could cause more than one piece of a face to be passed down to a subtree for clipping or tree building.) The ease of performing these operations would determine the usefulness of the tree representation. Obviously, the algorithm for finding convex subpieces is not valid for all such extensions.

5. Results

The intersection, union, and difference algorithms, and the algorithm for converting a planetree to a boundary representation were implemented in C under UNIX. [4] The objects used for testing were tiled from contours. The implementation was straightforward; it did not take advantage of locality.

6. Acknowledgements

This work was supported in part by NIA grant R01-CA39060. I thank Steve Pizer and Henry Fuchs for useful conversations and encouragement, Steve Pizer and Ed Bloomberg for reading many drafts, and Ari Requicha and Lee Nackman for encouragement to write up these results.

References

Aristides A. G. Requicha and Herbert B. Voelcker. January, 1985. "Boolean Operations in Solid Modeling: Boundary Evaluation and Merging Algorithms," *Proceedings of the IEEE*, **73**, 30 – 44.

Ivan E. Sutherland and Gary W. Hodgman. January, 1974. "Reentrant Polygon Clipping," *Communications of the ACM*, **17**(1), 32–42.

Aristides A. G. Requicha and Herbert B. Voelcker. January, 1984. Boolean Operations in Solid Modelling: Boundary Evaluation and Merging Algorithms, TM-26, Production Automation Project, University of Rochester, Rochester, New York.

Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. July, 1980. "On Visible Surface Generation by A Priori Tree Structures," *Computer Graphics*, **14**(3), 124–133.

Henry Fuchs, Gregory Abram, and Eric Grant. July, 1983. "Near Real-Time Shaded Display of Rigid Objects," *Computer Graphics*, **17**(3), 65-72.

[4] UNIX is a trademark of Bell Laboratories.