Set Operations on Polyhedra Using Binary Space Partitioning Trees

William C. Thibault
Georgia Institute of Technology
Atlanta, GA 30332
and
Bruce F. Naylor
AT&T Bell Laboratories
Murray Hill, NJ 07974

Abstract
We introduce a new representation for polyhedra by showing how Binary Space Partitioning Trees (BSP trees) can be used to represent regular sets. We then show how they may be used in evaluating set operations on polyhedra. The BSP tree is a binary tree representing a recursive partitioning of d-space by (sub-)hyperplanes, for any dimension d. Their previous application to computer graphics has been to organize an arbitrary set of polygons so that a fast solution to the visible surface problem could be obtained. We retain this property (in 3D) and show how BSP trees can also provide an exact representation of arbitrary polyhedra of any dimension. Conversion from a boundary representation (B-reps) of polyhedra to a BSP tree representation is described. This technique leads to a new method for evaluating arbitrary set theoretic (boolean) expressions on B-reps, represented as a CSG tree, producing a BSP tree as the result. Results from our language-driven implementation of this CSG evaluator are discussed. Finally, we show how to modify a BSP tree to represent the result of a set operation between the BSP tree and a B-rep. We describe the embodiment of this approach in an interactive 3D object design program that allows incremental modification of an object with a tool. Placement of the tool, selection of views, and performance of the set operation are all performed at interactive speeds for modestly complex objects.

CR Categories I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling - object representation and geometric algorithms.

Keywords - polyhedra, set operations, geometric modeling, geometric search, point location.

1. Introduction

While the study of polyhedra has an ancient history, computer science has given it renewed attention in the various sub-disciplines of computational geometry, geometric modeling, computer graphics, robotics, and computer vision. Its attractiveness stems from the relative simplicity of linear computations when compared to non-linear, coupled with the fact that linear approximations of non-linear sets can often be quite satisfactory. An important example of this comparative simplicity is set operations: union, intersection, difference and exclusive-or (and their complements). The algebra of set operations defined on the collection of linear sets of any dimension $\leq$ some d is closed (assuming a countable number of operations). This is not true of non-linear sets; for example, the intersection of two quadrics (second degree) can be a fourth degree curve. When computational speed is important, such as in interactive object design, using polyhedral approximations of non-linear solids can provide a very effective alternative to non-linear computations. On the other hand, for operations which are not speed-critical, a second unapproximated non-linear representation can be used, if the greater accuracy is needed.

 ACM-0-89791-227-6/87/007/0153 $00.75

The most prominent method of representing polyhedra at this time would appear to be boundary representations (B-reps): in a d dimensional space, a d-polyhedron (also called a d-polytope) is represented by a set of $(d-1)$-polyhedra, called faces, which are in turn represented by $(d-2)$-polyhedra, and so on until d equals 0, at which point the d coordinates of a vertex are used. An alternative suitable for representing convex polyhedra is provided by the volumetric approach, where the intersection of a set of halfspaces determines a polyhedron.

In this paper, we develop a new approach first presented in [Nayl86] and describe in greater detail in [Thib87]. It is based on the dimension independent concept embodied in the Binary Space Partitioning Tree, abbreviated BSP tree, which, at its simplest, is a binary tree whose non-leaf nodes are labeled with hyperplanes and whose leaves correspond to cells of a convex polyhedral tessellation (partitioning) of d-space. The approach provides what is essentially a volumetric representation of general linear polyhedra. What we mean by general is that any genus (handles/holes) is permissible, any number of connected components (separate objects), and regions of connectivity with no interior, such as two parts connected only by a vertex. More generality is available in that the interior of the polyhedra need not be completely bounded, i.e. it may be (semi-)infinite.

Previous work has established the BSP tree as an effective representation of polyhedra for efficient visible surface determination, both in polygon tiling environments [Schu69] [Fuch80] [Nayl81] [Fuch83] and for ray-tracing [Nayl86] (Figure RAY-TRACING). In this paper, we concentrate on the problem of evaluating set operations, the set theoretic analog of boolean operations, defined on 3D polyhedra. This takes two forms. One begins with a set (theoretic) expression represented as a tree (i.e. a CSG tree) defined on a set of polyhedra represented by B-reps. The method produces the polyhedron defined by the CSG tree by constructing its (non-unique) BSP tree representation. The resulting tree can then be used for rendering by the techniques referred to above or as input to the second approach. The second approach takes a BSP tree as one operand and a B-rep as the other and produces a new BSP tree determined by the set operation via modification of the original tree. We have used this technique as the basis for an interactive program that supports modification of a work piece, represented by a BSP tree, through the adding, subtracting or

intersecting of a tool, represented by a B-rep.

2. Representation of Polyhedra by BSP Trees

2.1 Generic BSP Trees[1]

A BSP tree represents a recursive, hierarchical partitioning, or subdivision, of d-dimensional space. It is most easily understood as a process which takes a subspace and partitions it by any hyperplane that intersects the interior of that subspace. This produces two new subspaces that can be further partitioned. Figure BSPT illustrates the relationship between the partitioning of space and the corresponding BSP tree. In (a), we see a recursive partitioning of the plane. Note how partitioning first by u produces two subspaces whose subsequent partitionings proceed independently of each other. The distinction between the two halfplanes formed by a line is indicated by the orientation of the normal vector to each line (indicated by arrows). Which of the two possible orientations is used is typically arbitrary. Now referring to (b), we see that in the corresponding BSP tree, each (sub-)line is associated with an internal node of the tree. The right subtree of each internal node represents the region of the plane lying to the side of the line pointed to by the normal. The left subtree represents the other side. The resulting partitioning produces a set of unpartitioned subspaces that correspond to leaves of the tree (labeled with digits).

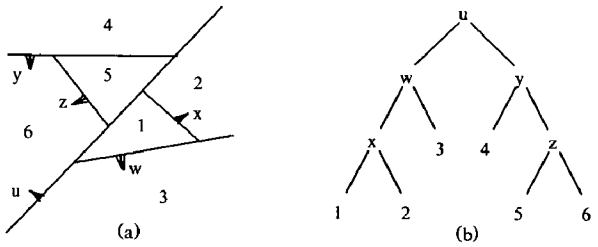


Figure BSPT. Geometry of a 2D partitioning (a) and its BSP tree (b).

More formally, for a hyperplane

$$H = \{(x_1, ..., x_d) | a_1x_1 + \cdots + a_dx_d + a_{d+1} = 0\},$$

the *right* (or in B-rep parlance, the "front") halfspace of H is

$$H^+ = \{(x_1, ..., x_d) | a_1x_1 + \cdots + a_dx_d + a_{d+1} > 0\},$$

and the *left* (or "back") halfspace of H is

$$H^- = \{(x_1, ..., x_d) | a_1x_1 + \cdots + a_dx_d + a_{d+1} < 0\}).$$

The right side of H lies to the side of H in the direction of the hyperplane's normal, $(a_1,...,a_d)$.

Each node v represents a region of space $R(v)$ (to be defined below). Leaves correspond to un-partitioned polyhedral regions, which we call *cells*. Each internal node v of the tree is associated with a *partitioning hyperplane*, H_v, which intersects the interior of $R(v)$. The hyperplane partitions $R(v)$ into three sets: $R(v) \cap H_v^+$, $R(v) \cap H_v^-$, and $R(v) \cap H_v$. The d-dimensional region in H_v^+ is represented by the right child of v, *v.right*, and the region in H_v^- is represented by the left child, *v.left*. The intersection of H_v and $R(v)$ is called the *sub-hyperplane* of H_v, indicated by $SHp(v)$, and is of dimension $d-1$.

$R(v)$ is the intersection of open halfspaces on the path from the root to v. More precisely, for each edge (v_1,v_2) in the tree we associate a halfspace $HS(v_1,v_2)$ defined as follows: for any node v, let $HS(v,v.left)$ denote H_v^-, and $HS(v,v.right)$ denote H_v^+. Let $E(v)$ denote the set of edges on the path from the root to v. Then $R(v) = \bigcap_{e \in E(v)} HS(e)$. For the root node, whose $E(v)$ is empty, $R(v)$ is defined to represent all of d-space. Thus, $R(v)$ is convex, non-empty, may not be completely bounded, and is topologically open. It also follows that sub-hyperplanes have the same properties. An important relationship between sub-hyperplanes and regions is that the sub-hyperplanes associated with nodes on the path from the root to v contain the boundary of $R(v)$. Finally, a *trivial* BSP tree consists of only a single node (a cell).

2.2 Representation of Regular Sets

A regular set S has an interior, an exterior, and a boundary denoted by *int* S, *ext* S, and *bd* S, respectively. A set is *regular* if it is the closure of its interior [Requ78], i.e $S = cl(\,int\ S\,)$, where *cl* denotes closure. (The *closure* of a set consists of the set together with its boundary.) Given a BSP tree, we can use it to represent linear regular sets, and polyhedra in particular. We need to simply classify each cell as either *in* the set or *out* of the set. Each leaf then has at least one attribute, membership, with values $\in \{ in, out \}$. For example, in Figure BSPT, consider the set defined when cells 1 and 5 are assigned the value *in* and the rest are assigned *out*. Since each cell is open (and therefore, has an interior) and is non-empty by construction, we can take the union of all in-cells and then form the closure of this union, to produce a regular set.

$$S = cl(\bigcup_i C_i), \text{ for all } C_i = in$$

Note that points lying between two in-cells are included in S and are in *int* S. The boundary of the set consists of points between in-cells and out-cells, and all such points lie in sub-hyperplanes of the tree.

$$bd\ S = \bigcup_{i,j} cl(C_i) \cap cl(C_j), \text{ for all } C_i = in, C_j = out$$

Methods of constructing such representations will be described in subsequent sections.

2.3 Point Classification

We can show the sufficiency of the above representation by solving a problem studied in computational geometry [Prep85]. The *point classification* problem can be stated: Given a set S and a point p, determine if p lies in *int* S, *ext* S, or *bd* S. We assume S is regular and we have a BSP tree representing S. Figure POINT-CLASSIFY gives an algorithm for solving this problem in d-space. The recursive process begins at the root of the tree and uses location of a point with respect to a hyperplane to control the search. To solve the problem, we must know whether the neighborhood of p is homogeneous, and therefore *in* or *out*, or non-homogeneous, and therefore *on*. If p lies in a cell, its neighborhood is known to be homogeneous. When p lies on some H_v, the search must be performed on both subtrees to determine all cells in whose closure p lies. If the value of all such cells are not the same, p is known to be *on*, otherwise it is known to be *in* or *out*, depending upon the value. (Note that the search could terminate whenever the first *on* value is encountered). While *bd* S has measure zero, it is given non-zero measure numerically by specifying an interval about zero which is mapped to "on the hyperplane", thus giving thickness to the hyperplanes. Machine precision determines a lower bound on this interval.

In [Kala82], this problem is solved for 3D in $O(n)$ for a B-rep with n faces. A result in [Nayl81] shows that this could be at most $O(n)$ for any BSP tree constructed from n faces (the tightest known upper-bound on tree size is $O(n^d)$). However, when a balanced BSP tree is of size $O(n)$ (which may or may not be possible for a given set of faces), this can be solved in $O(log\ n)$.

1. This section is an adaptation of work presented in [Nayl81].

```
procedure point_classify (p : point;  v : BSPTreeNode)
                                    returns (in, out, on)
  if v is a leaf
    return the leaf's value (in or out)
  else
    let d = dot_product(p, H_v).
    if d < 0 then
      return point_classify (p, v.left)
    else if d > 0 then
      return point_classify (p, v.right)
    else (* p lies on the partitioning hyperplane *)
      l := point_classify (p, v.left)
      r := point_classify (p, v.right)
      if l = r then
        return r
      else
        return "on"
end point_classify ;
```

Figure POINT-CLASSIFY.

2.4 Augmented BSP Trees

A common means of augmenting the generic BSP tree is to include other sets within the BSP tree structure. In particular, leaves can each include a collection of sets (objects) contained completely within the corresponding cell, e.g. [Schu69], and similarly, internal nodes can include sets lying in the corresponding sub-hyperplane, e.g. [Fuch80]. Traditionally, the motivation for this has been the visible surface problem in 3D. Given an arbitrary viewing position, a traversal of the tree can induce a visibility priority ordering on the contents of the various subspaces (cells and sub-hyperplanes). Because of the usefulness of boundary representations for polygon tiling, polygons have been stored at the various nodes. We retain this visibility property by associating sets of polygons with internal nodes, where each set lies on the node's sub-hyperplane, and are in the boundary of the represented polyhedron. At each node v, these faces are separated into those whose normals have the *same* orientation as the normal of H_v and those whose orientation is *opposite*.

3. B-rep → BSP tree

We now examine converting a B-rep into an equivalent BSP tree. Essentially any of the many varieties of B-reps can be used, as long as they are sufficient and form a valid representation of a polyhedron. We use the term *face* to refer to the (d-1)-dimensional boundaries of a d-polyhedron, H_f to denote the hyperplane containing a face f, and we assume that face normals point to the exterior.

The approach is essentially one that first appeared in [Fuch80] with one significant extension : assignment of values to leaves. The algorithm begins with a set of faces forming one or more disjoint polyhedra. At each stage, the recursive process selects a hyperplane H and partitions the current set of faces F into three sets of faces, $F(H^+)$, $F(H^-)$, $F(H)$, corresponding respectively to the three subspaces H^+, H^-, H. The *partitioning of a face*, $f \in F$, is defined as the result of forming the following three sets, one or two of which will be empty:

$$f^+ = cl(H^+ \cap int\, f),\quad f^- = cl(H^- \cap int\, f),\quad f^0 = cl(int(H \cap int\, f)),$$

where *int* is with respect to H_f. Partitioning all faces of F produces $F(H^+)$, $F(H^-)$, $F(H)$, respectively. The set $F(H)$ is retained at a new BSP tree node v (separated into same and opposite lists). The process then proceeds recursively on the other two sets until the current list of faces is empty (Figure BUILD-BSPT).

Figure CONCAVE shows how the algorithm can create a BSP tree from a concave polygon. One note worthy consequence of this process is that each polyhedron is decomposed into a set of convex regions (in-

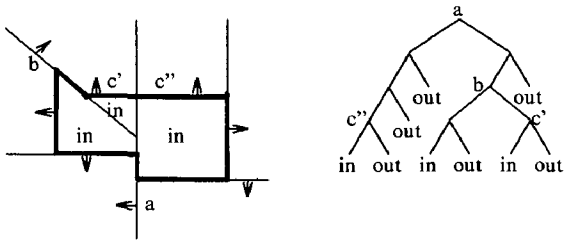


Figure CONCAVE. A concave set and its BSP tree.

```
procedure Build_BSPT ( F : set of faces ) returns BSPTreeNode

  Choose a hyperplane H that embeds a face of F;
  new_BSP := a new BSP tree node with H as its
          partitioning plane;
  <F_right, F_left, F_coincident, > := partition faces of F with H;
  Append each face of F_coincident to the appropriate face list
    of new_BSP;

  if (F_left is empty) then
    if (F_coincident has the same orientation as H) then
      (* faces point "outward" *)
      new_BSP.left := "in";
    else  new_BSP.left := "out";
  else
    new_BSP.left := build_BSPT( F_left );

  if (F_right is empty) then
    if (F_coincident has the same orientation as H) then
      new_BSP.right := "out";
    else  new_BSP.right := "in";
  else
    new_BSP.right := build_BSPT( F_right );

  return new_BSP;
end;  (* Build_BSPT *)
```

Figure BUILD-BSPT. Algorithm to build a BSP tree from a boundary representation.

cells). Note that the only aspect of this algorithm dependent upon the particular B-rep variant is the splitting of a face by a hyperplane. While any order of selection will produce a BSP tree representing the same set, some orders produce more desirable trees. The issue of selecting partitioning hyperplanes can be somewhat complicated, and is discussed briefly in section 5.1.

It is necessary, however, that all points on the boundary of the polyhedra lie in sub-hyperplanes of the resulting BSP tree (section 2.2 above). This is accomplished most simply by always choosing a hyperplane that embeds some face among the current set of faces. Eventually, all points on the original faces will lie in sub-hyperplanes. The second requirement is the correct classification of cells. Assignment of values to leaves occurs when the partitioning of a set of faces finds no faces on one side of the partitioning hyperplane. That region is then known to be *homogeneous*, i.e. the region lies either entirely within the interior of one of the polyhedra or entirely in the exterior of all the polyhedra. We know this because for it to be non-homogeneous, there would be some part of a boundary to make it so, i.e. to mark the transition between inside and outside. Therefore, the region forms a cell and can be classified as *in* or *out*. In this algorithm, differentiating between the two cases is simple. When hyperplanes are chosen from the (hyper)plane equation of some face, we use the fact that normals point to the exterior to deduce the fact that a left leaf must be *in* (in the back-halfspace of the face) and a right leaf *out* (in the front-

halfspace). Also, it is not difficult to show that when one subtree of v is a cell, any faces coincident with H_v will all have the same orientation.

Another quite similar approach involves an idea that we will need later: the concept of *inserting a face into a BSP tree.* Let us say that we had used the above algorithm to build a BSP tree out of only $n-1$ of the n faces. We could "add" the last face f to the tree in the following way. Let v be some node in the tree, initially equal to the root. Partition f by H_v. If it is coincident, add it to the appropriate face list of v. Otherwise, pass any part of f lying in H_v^- to *v.left*, and similarly any part in H_v^+ to *v.right*. Now repeat the process recursively on the subtrees. If and when a part of f reaches a leaf, create a new node. Now, if one begins with a trivial BSP tree, and inserts each face one-at-a-time, a BSP tree representing the polyhedra will be constructed.

Before leaving this discussion, we should point out that a much simpler case occurs when the input is a single convex polyhedron P of n faces. The above algorithm, when restricted to partitioning hyperplanes that embed a face, will always produce the same tree structure with n nodes independent of the order in which faces/hyperplanes are selected (Figure CONVEX). Each right child is a leaf with value *out* and the only left leaf has a value *in* representing *int P*. This structure is very similar to a list of the minimal set of (closed) halfspaces whose intersection equals P.

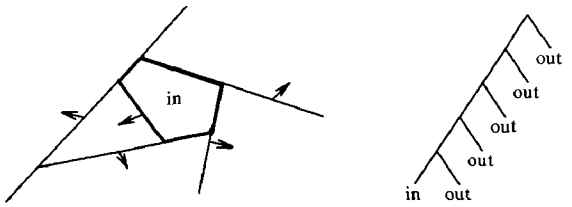


Figure CONVEX. A convex set and its BSP tree.

4. Evaluation of Set Operations Using BSP trees

Since we are concerned with regular sets, we are interested only in the regularized set operations [Requ78], which are denoted as such by an asterisk: $\cap^*$, $\cup^*$, $-^*$, and $\sim^*$. First, consider the unary complementation operator. Given a BSP tree representing a set S, a BSP tree representing its complement, $\sim^* S$, can be formed by simply complementing the cell values: all in-cells are changed to out-cells and all out-cells to in-cells. Any boundary polygons at internal nodes must have their orientations reversed as well. A boundary representation can be similarly complemented by reversing the orientation of every face.

To evaluate binary operators, we will use expression simplification in a geometric setting. Consider for example the expression $S_1 \cap^* S_2$. If we have determined that, for some region R, that $R \subseteq ext\ S_2$, then the expression in R may be simplified to $S_1 \cap^* \varnothing = \varnothing$, where $\varnothing$ denotes the empty set. If instead $R \subseteq int\ S_2$, the expression reduces to $S_1 \cap^* U_R = S_1$, where U_R is the universal set restricted to R. In either case, we can perform the simplification without any knowledge of the structure in R of S_1, which could be an arbitrarily complex sub-expression on arbitrarily complex objects. Analogous cases exist for the other operations (Figure SIMPLIFY). This has been called "pruning" in the context of CSG trees.

To utilize this technique we must partition the space into regions such that at least one operand is homogeneous in each region. That is, given the expression $S_1\ op\ S_2$ defined on some space, one must find a partitioning of that space such that for each region R_i of the partitioning, $R_i \subseteq int\ S_j$ or $R_i \subseteq ext\ S_j$, j = 1 or 2. For an expression of n operands, this property may need to hold in each R_i for up to $n-1$ of the operands, depending on the expression. This technique appears in a number of places, e.g. [Wood82] [Tilo84], and seems fundamental to the problem. We use a BSP tree to partition space to achieve these conditions.

op	left operand	right operand	result
$\cup^*$	S	in	in
	S	out	S
	in	S	in
	out	S	S
$\cap^*$	S	in	S
	S	out	out
	in	S	S
	out	S	out
$-^*$	S	in	out
	S	out	S
	in	S	$\sim^*$ S
	out	S	out

Figure SIMPLIFY. Expression simplification rules. S is an arbitrary regular set.

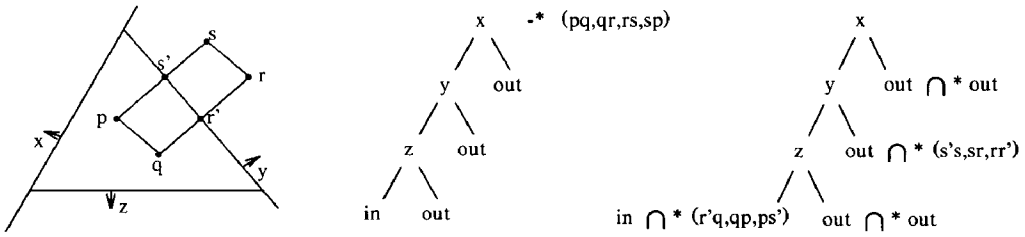
4.1 BSP Tree <op> B-rep → BSP Tree

Given a BSP tree $\hat{T}$ representing a polyhedron T, and a B-rep $\hat{B}$ representing a polyhedron B, we wish to evaluate $T\ op\ B$ or $B\ op\ T$, where *op* is a regularized set operation. In the case of the difference operator $S_1 -^* S_2$, we choose to complement the right operand and evaluate the equivalent $S_1 \cap^* \sim^* S_2$ [3]. Now, the approach is to perform the set operations on open sets only, since these are closed under standard (non-regularized) union and intersection. If the boundary of the result is needed, it is explicitly computed (see section 4.3 below). We will need to classify T and B with respect to each other. This is achieved by discovering parts of one that lie in the interior or exterior of the other. We refer parenthetically to Figure SET-OP, which illustrates $T -^* B$.

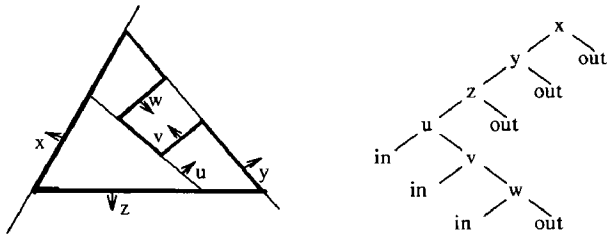
We begin by inserting collectively into $\hat{T}$ all of the faces of $\hat{B}$. As the faces filter down into $\hat{T}$ we can discover which if any of the subtrees of $\hat{T}$ lie entirely in *int B* or *ext B*. When at some node v, no part of $\hat{B}$ is found to lie on one side of H_v, say, the left side, then R (*v.left*) must be homogeneous with respect to B (e.g., x.right and z.right in the figure), as explained in section 3. A general method for determining whether the region is in *int B* or *ext B* is given below in Section 4.5. When this occurs, the subtree rooted at *v.left* is either left untouched, or is replaced by a leaf, depending upon the simplification rules (in our example, both x.right and z.right are not modified). If it is also the case that no face of B is coincident with H_v, then the sub-hyperplane of v has also been classified with respect to B. The faces of v are kept or deleted according to the same simplification rules (e.g, x's sub-hyperplane is in *ext B* and its face is kept). Deletion of v may also be possible (see section 4.4).

The insertion process results in $\hat{B}$ being distributed among some subset of the subspaces of $\hat{T}$, i.e. cells or sub-hyperplanes. The reaching of a leaf l by some subset of the boundary, $\hat{B}_l$, means that $\hat{B}_l$ has been classified with respect to T (e.g. the faces in y.right and z.left in (3)). The operation can then be evaluated since we have a region in which one operand, T, is homogeneous. The result is either T's value (e.g. y.right) or B's value (e.g. z.left) in the region represented by the leaf, depending upon the particular operation and the value of the leaf, as given in Figure SIMPLIFY. If the value is T's, then the faces of $\hat{B}_l$

3. It is possible to gain a little in efficiency by performing the complementation as part of the evaluation so that only the parts included in the result are actually complemented.



(1) Initial geometry. (2) Initial representations. (3) BSP tree after classifying (qp,rq,sr,ps).



(4) Resulting partitioning. (5) Final BSP tree.

Figure SET-OP. BSP tree −* B-rep → BSP tree.

are discarded (as in y.right); otherwise $\hat{T}$ is "extended" by replacing the leaf with a subtree built from the faces of $\hat{B}_l$ (as in z.left). This can be performed by the procedure Build-BSPT, given earlier. Thus, the cell is "refined" to reflect B's value in the region. The tree now represents the desired set. We refer to this algorithm as the *incremental* set-op evaluation algorithm because it can be used to create a polyhedron by a series of "incremental" modifications to an initial polyhedron. The algorithm is summarized in Figure INCREMENTAL SET-OP.

4.2 CSG on B-reps → BSP Tree

A Constructive Solid Geometry representation (CSG) of a set S is a binary tree in which the internal nodes represent (regularized) set operations and leaves are instanced primitives (such as blocks, cones, etc.) [Requ80]. One can classify some arbitrary set s with respect to S by first classifying s with respect to each primitive, and then combining the classifications according to the set theoretic expression represented by the CSG tree [Tilo80][Roth82]. An alternative is to convert the CSG representation to a more explicit form, such as a B-rep or BSP tree, and classify with respect to that representation. The algorithm we now present provides this latter approach.

We define a CSG evaluation problem π as a pair (τ, R), where τ is a CSG tree with polyhedral primitives represented as B-reps or as trivial BSP trees (representing $\varnothing$ or U), and where R is a convex region of d-space on which τ is defined. The algorithm returns a BSP tree which represents the same set in R as τ. Starting with the problem $\pi = (\tau, R)$, the algorithm chooses a hyperplane H to partition the problem into two sub-problems, $\pi_{left} = (\tau_{left}, R \cap H^-)$, and $\pi_{right} = (\tau_{right}, R \cap H^+)$. The root of the tree returned has H as its partitioning hyperplane, and its left and right subtrees are the results of the recursive evaluation of π_{left} and π_{right}, respectively. The recursion is terminated when the current CSG tree reduces to a trivial BSP tree (a cell).

The algorithm is quite similar to Build-BSPT of section 3, with one important difference: rather than having just a simple list of faces to partition, we have a CSG tree with faces at its leaves. Figure CSG-EVALUTATION describes the algorithm. As before, a hyperplane H is chosen at each stage that embeds a face using a heuristic (Section 5.1). Two copies of the CSG tree are generated and modified to represent the set in each of the two halfspaces of H. This entails for each primitive replacing the faces of that primitive in the respective CSG trees with the subset of the faces that lies in each halfspace.

```
if op = −* then
  B := Negate_B-rep( B )
  op := ∩*

procedure Incremental_Set_op
                ( op : set_operation ; v : BSPTreeNode ;
                  B : set of Face ) returns BSPTreeNode
  if v is a leaf then
    case op of
      ∪* : case v.value of
        in  : return v
        out : return Build_BSPT( B )
      ∩* : case v.value of
        in  : return Build_BSPT( B )
        out : return v
  else
    <B_left, B_right, B_coincident> := partition B with H_v
    if B_left has no faces then
      status := Test_in/out(H_v, B_coincident, B_right)
      case op of
        ∪* : case status of
          in  : discard_BSPT( v.left )
                v.left := new "in" leaf
          out : do nothing
        ∩* : case status of
          in  : do nothing
          out : discard_BSPT( v.left )
                v.left := new "out" leaf
    else
      v.left := Incremental_Set_op( op, v.left, B_left )

    if B_right has no faces then
      (* similar to above *)
    else
      v.right := Incremental_Set_op(op, v.right, B_right)
    return v

end Incremental_Set_op ;
```

Figure INCREMENTAL SET-OP. Psuedo code for the incremental set evaluation algorithm.

Faces coincident with H are retained at the new node. Detection of homogeneous regions allows CSG tree simplification using the rules in Figure SIMPLIFY. If the CSG tree is reduced to, in effect, a single value (in/out), the problem in that region has been solved and is represented by a cell of the BSP tree. The entire problem, then, is solved through the discovery/creation of regions which are homogeneous with respect to the defined set, where each region is represented by a different cell of the resulting BSP tree.

```
procedure Evaluate_CSG ( τ : CSGTree ) returns BSPTree

   choose a face f of a primitive of τ
   v := new BSPTreeNode; H_v := H_f
   <τ_left , τ_right> := Split_CSG ( τ, H_v )

   τ_left := Simplify_CSG ( τ_left )
   if τ_left represents ∅ then
      v.left := new "out" leaf
   else if τ_left represents U then
      v.left := new "in" leaf
   else
      v.left := Evaluate_CSG ( τ_left )
   (* similar code for τ_right *)
   return v
end; (* Evaluate_CSG *)

procedure Split_CSG ( τ : CSGTree; H : plane_equation )
                                 returns <CSGTree, CSGTree>
   if τ is not a primitive then
      τ_left := copy ( τ.root )
      τ_right := copy ( τ.root )
      <τ_left.left, τ_right.left> := Split_CSG ( τ.left, H )
      <τ_left.right, τ_right.right> := Split_CSG ( τ.right, H )
   else
      <τ_left, τ_right, τ_coincident> := partition τ with H
      if τ_left = ∅ then
         τ_left = Test_in/out ( H, τ_coincident, τ_right )
      elseif τ_right = ∅ then
         τ_right = Test_in/out ( H, τ_coincident, τ_left )
      Add τ_coincident to v's face lists

   return <τ_left, τ_right>
end; (* Split_CSG *)
```

Figure CSG-EVALUATION. Algorithm for converting from a CSG tree to a BSP tree.

4.3 Boundaries

The two algorithms described above produce, in effect, a generic BSP tree which is sufficient for point classification and ray-tracing. While certain faces were retained at internal nodes, these no longer correspond necessarily to the boundary of the set S represented by the tree $\hat{S}$. Since B-reps are useful for rendering via polygon tiling, and the BSP tree can induce a priority ordering on the faces, we may wish to generate the boundary faces of S. This requires that for each node v of $\hat{S}$, we find and store at v, $bd\ S \cap SHp(v)$ (where $SHp(v)$ is the sub-hyperplane of v). There are two alternatives. One is to discard the old faces entirely and generate the boundary faces directly from the generic BSP tree using a technique described in [Thib87]. The second, which we will describe here, constructs the new faces from the faces of the operands.

The boundary of the result of any set operation is known to be a subset of the boundaries of the operands. Now, since $bd\ S$ is known to lie entirely within the sub-hyperplanes of $\hat{S}$, only the parts of the original faces which lie in these sub-hyperplanes can possibly be in $bd\ S$. These two facts imply that the faces retained at $\hat{S}$'s nodes form a superset of $bd\ S$, i.e. their union contains $bd\ S$, and the discarded faces do not contain any subset of $bd\ S$. It also immediately follows that for a given node v, any part of $bd\ S$ lying in the $SHp(v)$ must be a contained in the region covered by the faces retained at v. However, parts of these faces may lie in either $int\ S$ or $ext\ S$. To find the *on* parts of these faces, we can insert them into the subtrees of v, analogous to the technique used in point classification for points lying in sub-hyperplanes. This produces a set of new faces, a subset of which form $bd\ S \cap SHp(v)$, and this subset is retained at v (as opposed to extending the tree as in sections 3 and 4.1).

4.3.1 Classifying Faces. Consider for the moment the case where *v.right* is a cell with value *out*, as at node y in Figure SET-OP. Then the boundary contained in the $SHp(v)$ is precisely the points lying between this out-cell and those in-cells in *v.left* whose closure intersects H_v. Moreover, the orientation of the boundary faces must be that of H_v, since they are to point to the exterior, which by construction lies in *v.right*. Therefore, faces in the opposite-face list cannot be in $bd\ S$. Now, if we classify the same-faces by inserting them into *v.left*, the resulting faces which are classified as *in* with respect to *v.left*, i.e those which reach in-cells, must lie in $bd\ S$. Those in out-cells would be between two out-cells and thus known to lie in $ext\ S$. These can be discarded. As an example, in Figure SET-O, a face of the original tree at node y, when inserted into *y.left* would be split into three pieces, two of which are *in* and the third (middle piece) is *out*.

Now, to extend this for an arbitrary *v.right*, we first take the in-faces from the *v.left* insertion/classification above and insert/classify then with respect to *v.right*. The faces resulting from this insertion that are classified as *out* are then known to lie between an in-cell and an out-cell, and therefore in $bd\ S$. Now, the same process applied to the opposite-faces, but with the insertion sequence reversed (*v.right* then *v.left*), produces faces in $bd\ S$ whose orientation is opposite of H_v.

In the case of the incremental algorithm, we can exploit the fact that a single set operation is being evaluated, and use its semantics to avoid inserting faces into both subtrees. Consider union. We know that the neighborhood in the back-halfspace of a face of either operand is in the interior of the result. Therefore, we know *a priori* that same-faces inserted into *v.left* will all land in in-cells, and similarly for opposite-faces inserted into *v.right*. Thus, each face needs only to be classified with respect to one subtree: same-faces with respect to *v.right*, and opposite-faces with respect to *v.left*. The resulting faces that land in out-cells lie on the boundary, since the other side is known to be in-cells. For intersection, a similar analysis indicates that same-faces should be inserted into *v.left*, opposite-faces into *v.right*, and that resulting faces lying in in-cells should be kept.

While the above technique guarantees that the union of the remaining faces is exactly $bd\ S$, it does not guarantee that the set of faces at each node are disjoint. If the faces are given the same attributes, such as color, this redundancy will not affect renderings of the object, other than to possibly increase time and space requirements. However, this redundancy can be eliminated by merging the faces, i.e. by forming for each node independently the union of the same-faces and separately the union of the opposite-faces.

4.3.2 Face Merging. Merging of faces can be performed by the CSG evaluation algorithm in the dimension of the faces, optimizing for the fact that there is only one type of operator: union. Conceptually, we have a CSG tree representing $f_1 \cup f_2 \cup \cdots \cup f_n$, for n faces. The result is a BSP tree, in $(d-1)$-space, where the $(d-1)$ value of "in" corresponds to the d-value of "on", and similarly "out" corresponds to "not-on". Faces lying in a hyperplane H are orthogonally projected into a coordinate hyperplane by dropping the coordinate corresponding

to the largest coordinate of H's normal. The tree building process proceeds as before, but in the lower dimension. The recursion terminates when regions are discovered that are either completely covered by some face or contain no faces.

Let us consider the case where $d = 3$. If convex polygons are the desired output, it is relatively straightforward to maintain a vertex-list representation of the regions of the 2D tree. All in-regions yield polygons whose vertices are projected back into H. Now, for concave polygons, we must find the $(d\text{-}2)$ boundary of the in-regions. This means that finding the 2D boundary of a 3D set requires recursing in dimension and finding the 1D boundary of 2D set and subsequently the 0D boundaries of 1D sets. Thus, to perform the complete boundary evaluation requires that we apply our algorithm recursively in dimension. The recursion forms 1D BSP trees for each internal node of a 2D BSP tree. The in-cells of these trees lie on polygon edges. In this 1D-space, hyperplanes are forced to the form $[\ 1\ \ -x\ ]$. Vertices lie on these hyperplanes and have value x, and the left subtree of a node contains values $< x$ while the right subtree values are $> x$, i.e. they are binary search trees. To find the minimum boundary of these 1D in-regions, i.e. the pairs of vertices bounding each edge, we can traverse each 1D tree using the procedure in Figure GENERATE-EDGES. The vertices are projected back from 1D to 2D which are then projected back into H defined in 3D. This then produces a merged set of edges bounding the on-regions (with respect to the 3D polyhedron) lying in a given sub-hyperplane[4].

```
Global variables
  v1,v2 : scalar,  last_value : { in,out } := out
  edge_list : LIST OF ( v1,v2 )

Generate_Edges( root, [ 1  −−∞] )

procedure Generate_Edges( v : BSPTreeNode,
                          min : 1D-Hyperplane )
  if v = leaf then
    case ( last_value, v.value )
      ( out,in ) -> v1 := min.x
      ( in,out ) -> v2 := min.x
                    edge_list += NewEdge( v1, v2 )
    last_value := v.value
  else
    Generate_Edges( v.left, min )
    Generate_Edges( v.right, H_v )

end Generate_Edges
```

Figure GENERATE-EDGES

Another alternative for boundary generation from the CSG evaluator, described in [Thib87], uses a technique where each same-face is inserted into *v.left* and a copy, but with orientation reversed, is inserted into *v.right*. The complementary operation is performed for the opposite-faces. The resulting in-cell faces are retained and merged together as above, but with the following "glue" operator in place of union:

(same, same) -> same
(same, opposite) -> not-on
(opposite, same)-> not-on
(opposite, opposite) -> opposite

The 1D boundaries of same and opposite regions are constructed independently. This kind of operation has appeared elsewhere, e.g. [Putn86], to "regularize" the set.

4. Representing a set of arbitrary non-overlapping polygons by a set of edges is sufficient for many polygon processing algorithms.

4.4 BSP Tree Reduction

Once a BSP tree has been constructed as the result of the evaluation of set operations, it may be possible to *reduce* the tree by eliminating certain nodes without changing the represented set. We identify two cases in which this reduction is possible. The first case occurs when both subtrees of a node v are cells with identical values (Node z in Figure REDUCE). Since $R(v)$ is homogeneous, the subtree rooted at v can be replaced by a cell with the same value. Note that no boundary faces could lie in the sub-hyperplane of such a node. This case arises naturally from expression simplification during which a formerly non-homogeneous region is simplified to a homogeneous one, and is analogous to the "condensation" of quad/oct-trees. It can be performed as part of the tree construction.

We may also remove a node that has as one child a cell and, in addition, has no part of the boundary in its sub-hyperplane (node u in Figure REDUCE). This means that all cells in the other subtree bounded by this node's hyperplane have the same value as the cell. Since the sub-hyperplane does not contribute to the differentiation of space, the tree rooted at this node can be replaced by the node's non-trivial subtree (Node w). This reduction can be performed during the phase that generates the boundary faces. (With the incremental algorithm, this can be detected and effected during set-op evaluation.)

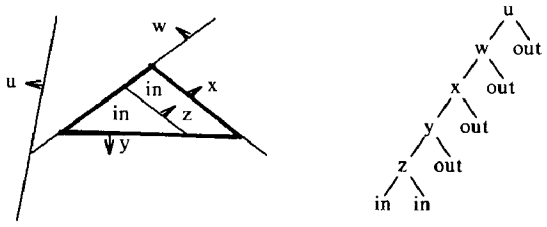


Figure REDUCE. Nodes u and z can be eliminated.

4.5 The In/Out Test

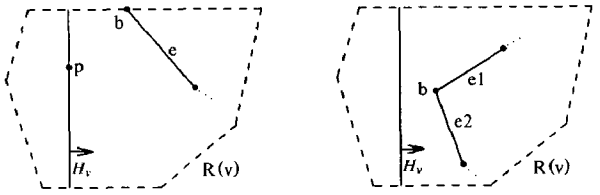
In all three of our algorithms that produce BSP tree representations of polyhedra, we discover regions that do not contain any faces of a polyhedron B represented by a B-rep $\hat{B}$. In these cases, we must determine whether that region lies in *int B* or *ext B*. In procedure Build_BSP, we saw that we could use the normal of a face, coincident with H, to answer this question. However, in the set operation algorithms, no such face may exist. We must then decide the status of this region based upon the subset of *bd B* lying on the non-homogeneous side of H. We solve this for dimensions 1, 2 and 3.

Let $\hat{B}_v = \hat{B} \cap R(v)$. (Note that since $R(v)$ is open, $\hat{B} \cap bd\ R(v)$ is not included in $\hat{B}_v$). We assume, without loss of generality, that B_v lies entirely in H_v^+, and therefore in $R(v.right)$. We are then interested in determining the status of $R(v.left)$ with respect to B. In the case where $B \cap R(v)$ is convex, this is relatively simple. We can test some point lying in $SHp(v)$ for inclusion in the back half-spaces of all faces of $\hat{B}_v$. If the point is "behind" all of these faces, then $R(v.left) \subset int\ B$, otherwise $R(v.left) \subset ext\ B$. Such a point can be easily produced if each sub-hyperplane embeds some face: we use the centroid of three non-collinear vertices of this face.

We now address the problem for (sets of) arbitrary polyhedra. One alternative is the ray casting test [Laid86]. This method would intersect a ray emanating from a point on the sub-hyperplane with $\hat{B}_v$ to find the closest face, from whose orientation the classification can be obtained. If the closest intersection point lies on more than one face, the process is repeated with a randomly perturbed ray. We have,

however, discovered a simpler method which uses the closest vertex b of $\hat{B}_v$ to H. This b can be found trivially during the partitioning of $\hat{B}_v$ by H_v. In the following, let p be a point in $SHp(v)$.

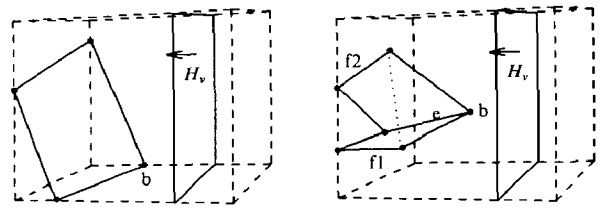
In 1D, the problem is solved exactly as in Build-BSPT, i.e from the orientation of the single face (a point). For 2D, the problem is illustrated in Figure 2D-IN/OUT. Vertex b is either in $bd\ R(v)$ or in $int\ R(v)$. If b lies in $bd\ R(v)$, then there is a single edge e in $\hat{B}_v$ incident with b. (A second edge could only lie in $bd\ R(v)$ or $ext\ R(v)$). If p lies in H_e^+, then $R(v.left)$ lies in $ext\ B$. Otherwise, $R(v.left)$ is in $int\ B$. Now, if b is in $int\ R(v)$, b is incident with two edges, e_1 and e_2. The region $R(v.left)$ is in $ext\ B$ if e_1 and e_2 lie in each other's back halfspace, i.e., if $e_1 \subset H_{e_2}^-$ and $e_2 \subset H_{e_1}^-$. This means that b is a point of "local convexity" of B. Otherwise $R(v.left)$ is in $int\ B$ (and b is a point of "local concavity").



(a) closest vertex in $bdR(v)$. (b) closest vertex in $'int'R(v)$.

Figure 2D-IN/OUT.

In 3D, the situation is somewhat similar: either b lies in $bd\ R(v)$ and is not shared by any other face of $\hat{B}_v$, or b is shared by more than one face of $\hat{B}_v$ (and may lie in either $bd\ R(v)$ or $int\ R(v)$). The test for the first case is the same as above: p is tested against the hyperplane of the single face containing b. When b is shared by more than one face of $\hat{B}_v$, we select the edge which forms the smallest angle with the plane H_v (think of b lying on H_v). In the neighborhood of b, this is the closest edge of $\hat{B}_v$ to H_v (Figure 3D-IN/OUT). If f_1 and f_2 are the faces that share this edge, then $R(v.left)$ is in $ext\ B$ if, in a local region of b, f_1 and f_2 lie in each other's back halfspace; otherwise, $R(v.left)$ lies in $int\ B$. To determine this we first find a vertex of f_1 adjacent (connected by an edge) to b but not lying in f_2. The location with respect to the plane of f_2 of this vertex provides the same answer as in the 2D case above. If the faces are convex, any vertex of f_1 not lying in f_2 will do. If there is a tie for the closest vertex, we can choose the one that allows the simplest test.



(a) vertex b on only one face. (b) vertex b on more than one face.

Figure 3D-IN/OUT.

5. Experience

5.1 Selection of Partitioning Hyperplanes

While a thorough discussion of methods by which to select partitioning hyperplanes is beyond the scope of this paper, we will at least describe the primary ones we have been using. The two principal properties of BSP trees that we are wanting to optimize are size and balance. Because finding the optimal is considered to be computationally hard, heuristics are employed. Most of our work has been with heuristics that select a hyperplane from among those that embed faces. For a set of faces, we define the *candidate set* to be those faces that are to be considered for generating partitioning hyperplanes. The *test set* consists of the faces against which each candidate hyperplane is tested, with possible outcomes being "in front of", "in back of", and "intersected by". The heuristic is a function of the number of outcomes of each type that occurred when a candidate was tested against the test set. The candidate chosen is that member of the candidate set that maximizes the heuristic. We investigated three heuristic functions:

$$Heur_1\ (front,back,split) = (-|back - front|) - w_{split} * split$$
$$Heur_2\ (front,back,split) = (front * back) - w_{split} * split$$
$$Heur_3\ (front,back,split) = front - w_{split} * split$$

The weight w_{split} allows "tuning" of the heuristics. The reason for applying a negative weight to intersected candidates is that splitting of faces tends to increase tree size and total computation time. The first two heuristics try to balance the number of faces on each side of the candidate. The third is motivated by CSG trees with convex primitives and attempts to maximize the number of faces in the exterior of some primitive. This can facilitate CSG tree simplification, since in one of the two subspaces, the value of the primitive will be *out*.

5.2 Implementation of the CSG evaluation algorithm

The CSG evaluation algorithm has been implemented in a dialect of Pascal running under Unix BSD 4.3. The CSG tree is described in a simple language of our design, translated using *lex* and *yacc*. Statistics obtained for various test objects are given in Figure STATS. Objects "stand" and "holed head" are depicted in Figure RAY-TRACING. In Figure CLUTCHPLATE, the edges (highlighted) reveal the spatial partitioning of that object. Tests were run on a VAX 8650. For each heuristic, $w_{split} = 8$, the candidate set consisted of 5 polygons chosen at random from each primitive in the current sub-problem, and the test set consisted of all polygons in the sub-problem. Early experience with various candidate set sizes shows that heuristics $Heur_1$ and $Heur_3$ are comparable. $Heur_2$ produces trees with a larger number of nodes, but with less CPU time than is required by the other heuristics for the same objects.

5.3 Implementation of the Incremental Algorithm

We have implemented the algorithm for incremental set operations in C on a Silicon Graphics IRIS workstation. The user modifies a "work piece", represented by a BSP tree, with a "tool", represented by a B-rep. The user can interactively control the view and the tool's position. Moving the tool results in a temporary union of the current work piece with the tool at its current position. Visibility is accomplished by transmitting the polygons in back to front order, using the visibility priority ordering produced from the BSP tree. The union we use for this is a "lazy" union because we do not re-evaluate boundary polygons in nodes of the resulting tree. We can do this because the visible surface of two objects that interpenetrate is the same as the visible surface of their union. Re-evaluation would only serve to eliminate invisible faces in the interior or overlapping faces on the boundary. Our impression is that the time required to draw the extra polygons is less than that needed to update the boundaries. In addition, subtrees lying inside the tool are saved and former cells that were refined by the tool's faces are noted. Thus to restore the original tree for use in the next frame requires reinstating the "removed" subtrees and cells, and removing any of the tool's faces lying in internal nodes of the tree. Finally, once the tool is positioned, the user chooses a set operation and the BSP tree is modified to reflect the result.

The initial work piece is obtained by either converting some B-rep to a BSP tree or using the result of the CSG evaluator. We have restricted the tools to be convex polyhedra so that we can take advantage of the

object	number of primitives	number of polygons	heuristic used	cpu (seconds)	tree size (nodes)	tree height	number of polygons
clutchplate	8	158	1	8.3	368	43	353
			2	7.2	408	27	362
			3	8.3	369	47	353
stand	31	623	1	41.9	713	31	1781
			2	41.1	814	31	1825
			3	152.9	896	93	1850
holed head	3	955	1	30.6	1536	104	1982
			2	24.5	1811	61	2167
			3	32.8	1532	90	2067

Figure STATS. Statistics for some test cases.

simpler algorithms for tree building and in/out testing. We have not found this to be an unnatural limitation for the user. Also, the IRIS workstation requires polygons to be convex. In forming the boundary during set operations, we take advantage of the convex decomposition generated by the BSP tree to provide us with convex polygons.

6. Concluding Comments

6.1 Comparison to Alternative Approaches

Space limitations prevent any but a limited discussion of the relationship of our work to others. The octree [Meag82] is similar in ways to the BSP tree. Both are tree structures that recursively subdivides space and assigns values to leaves, and both are dimension-independent. The most obvious difference is that octrees require the partitioning to be axis-aligned and the subdivision to be uniform. Of course, any partitioning of space by an octree can be modeled by a BSP tree[5]. The simplicity of octrees is attractive, and this can lead to certain advantages. But, for representation of polyhedra, the octree in general provides only an approximation, and it is typically a very verbose one. However, work described in [Carl85] [Ayal85] attempts to addresses these problems. While the verbosity is reduced, it still remains a problem. Set operations (in [Ayal85] and described for 2D only) require identifying and handling a number of cases, an aspect that tends to complicate implementations and makes extension to higher dimensions difficult. More importantly, axis-aligned partitioning schemes do not transform. To transform an octree it must be rebuilt. BSP trees do transform: simply apply the transformation to each hyperplane (the inverse of what would be applied to points). Also, we expect the generality of orientation to lead to smaller representations.

In B-rep algorithms, e.g. [Mant83] [Requ85] [Laid86] [Putn86], the geometric search structure, the set operations, and the visible surface determination are independent. In the BSP tree, they are all unified in a single structure (also true of octrees). While boundary representations transform, the search structures are typically axis-aligned. With one exception [Putn86], the algorithms for set operations are not dimension-independent and are somewhat complex with, once again, considerable case analysis. The principal "case analysis" per se for the BSP tree is the partitioning of a face by a hyperplane. On the other hand, B-reps are typically more concise (although not always).

6.2 Future Work

Other operations that we have examined include the calculation of metric properties such as volume, surface area, center of mass, etc. (see [Thib87]). We have also made a potentially important step by devising a closed set theoretic (boolean) algebra on BSP trees, thus dispensing with B-reps per se. In addition, the original ray-tracing techniques have been extended considerably, now exploiting non-linear hyperplanes. Utilization of non-linear hyperplanes is also possible with the fundamental techniques presented in this paper. However, the simplicity of linear computations would be lost in doing so. Nonetheless, we intend to explore this option. Heuristics are another area requiring greater study. All partitioning hyperplanes do not need to embed faces. One technique we have begun investigating is the use of a "median cut" algorithm similar to that used to build k-d trees [Bent79]. This can result in more well-balanced trees, especially for convex regions bounded by many faces.

6.3 Conclusions

A new representation for something as fundamental as polyhedra introduces a new "algorithm space" to explore. Divide-and-conquer algorithms are often simple and efficient and we believe this is reflected in the BSP tree algorithms. Also, the dimension independent aspect allowed a solution to the boundary problem without introducing a different methodology. The unified framework provided for geometric searching, set operations, and visible surface rendering reduces the conceptual complexity as well as the complexity of implementations.

The representation can be viewed as something of a cross between octrees and boundary representations. It has the unifying quality of octrees, but is not as simple. It has the exactness, transformability and conciseness of boundary representations, although not generally as concise. In fact, one might view the greater verbosity as the cost of the unity, something which must be weighed against the other gains.

5. To make the cost of determining the location of a point in a BSP tree more comparable to an octree, we use plane-equation-type = (x-axis, y-axis, z-axis, arbitrary) and optimize when not "arbitrary".

References

[Ayal85] D. Ayala, P. Brunet, R. Juan, and I. Navazo, "Object Representation by Means of Nonminimal Division Quad trees and Octrees," *ACM Transactions on Graphics Vol. 4(1)* pp. 41-59 (January 1985).

[Bent79] Jon Louis Bentley and Jerome H. Friedman, "Data Structures for Range Searching," *Computing Surveys Vol. 11(4)*, pp. 397-409 (December 1979).

[Carl85] Ingrid Carlbom, Indranil Chakravarty, and David Vanderschel, "A Hierarchical Data Structure for Representing the Spatial Decomposition of 3-D Objects," *IEEE Computer Graphics and Applications*, pp. 24-31 (April 1985).

[Fuch80] H. Fuchs, Z. Kedem, and B. Naylor, "On Visible Surface Generation by a Priori Tree Structures," *Computer Graphics Vol. 14(3)*, (June 1980).

[Fuch83] Henry Fuchs, Gregory D. Abram, and Eric D. Grant, "Near Real-Time Shaded Display of Rigid Objects," *Computer Graphics Vol. 17(3)* pp. 65-72 (July 1983).

[Kala82] Yehuda E. Kalay, "Determining the Spatial Containment of

a Point in General Polyhedra," *Computer Graphics and Image Processing Vol. 19* pp. 303-334 (1982).

[Laid86] David H. Laidlaw, W. Benjamin Trumbore, and John F. Hughes, "Constructive Solid Geometry for Polyhedral Objects," *Computer Graphics Vol. 20(4)* pp. 161-170 (August 1986).

[Mant83] Martii Mantyla and Markku Tamminen, "Localized Set Operations for Solid Modeling," *Computer Graphics Vol. 17(3)* pp. 279-288 (July 1983).

[Meag82] D. Meagher, "Geometric Modeling using Octree Encoding," *Computer Graphics and Image Processing Vol. 19* (June 1982).

[Nayl81] Bruce F. Naylor, "A Priori Based Techniques for Determining Visibility Priority for 3-D Scenes," Ph.D. Thesis, University of Texas at Dallas (May 1981).

[Nayl86] Bruce F. Naylor and William C. Thibault, "Application of BSP Trees to Ray-Tracing and CSG Evaluation," Technical Report GIT-ICS 86/03, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia 30332 (February 1986).

[Prep85] Franco P. Preparata and Michael Ian Shamos, *Computational Geometry: An Introduction,* Springer-Verlag, New York (1985).

[Putn86] L. K. Putnam and P. A. Subrahmanyam, "Boolean Operations on n-Dimensional Objects," IEEE Computer Graphics and Applications, pp. 43-51 (June 1986).

[Requ78] Aristides A. G. Requicha and Robert B. Tilove, "Mathematical Foundations of Constructive Solid Geometry: General Topology of Closed Regular Sets," TM-27a, Production Automation Project, University of Rochester, Rochester, New York 14627 (June 1978).

[Requ80] Aristides A. G. Requicha, "Representations for Rigid Solids: Theory, Methods, and Systems," *Computing Surveys Vol. 12(4)* pp. 437-464 (December 1980).

[Requ85] Aristides A. G. Requicha and Herbert B. Voelcker, "Boolean Operations in Solid Modeling: Boundary Evaluation and Merging Algorithms," Proceedings of the IEEE Vol. 73(1) pp. 30-44 (January 1985).

[Roth82] Scott D. Roth, "Ray Casting for Modeling Solids," *Computer Graphics and Image Processing Vol. 18* pp. 109-144 (1982).

[Schu69] R. A. Schumacker, R. Brand, M. Gilliland, and W. Sharp, "Study for Applying Computer-Generated Images to Visual Simulation," AFHRL-TR-69-14, U.S. Air Force Human Resources Laboratory (1969).

[Thib87] William C. Thibault, "Application of Binary Space Partitioning Trees to Geometric Modeling and Ray-Tracing", Ph.D. Dissertation, Georgia Institute of Technology, Atlanta, Georgia, (1987).

[Tilo80] Robert B. Tilove, "Set Membership Classification: A Unified Approach to Geometric Intersection Problems," *IEEE Transactions on Computers Vol. C-29(10)* pp. 874-883 (October 1980).

[Tilo84] Robert Tilove, "A Null-Object Algorithm for Constructive Solid Geometry," *Communications of the ACM Vol. 27(7)* (July 1984).

[Wood82] J. R. Woodwark and K. M. Quinlan, "Reducing the effect of complexity on volume model evaluation," *Computer Aided Design Vol. 14(2)* (1982).

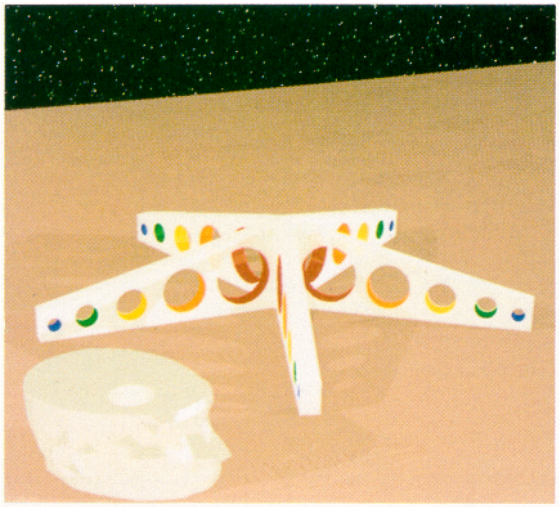


Figure RAY-TRACING. Two objects defined with BSP trees and rendered by ray-tracing.

Figure CLUTCHPLATE. Edges reveal the partitioning.