Merging BSP Trees Yields Polyhedral Set Operations

Bruce Naylor*, John Amanatides† and William Thibault‡

*AT& T Bell Laboratories

†York University

‡California State University at Hayward

Abstract

BSP trees have been shown to provide an effective representation of polyhedra through the use of spatial subdivision, and are an alternative to the topologically based b-reps. While bsp tree algorithms are known for a number of important operations, such as rendering, no previous work on bsp trees has provided the capability of performing boolean set operations between two objects represented by bsp trees, i.e. there has been no closed boolean algebra when using bsp trees. This paper presents the algorithms required to perform such operations. In doing so, a distinction is made between the semantics of polyhedra and the more fundamental mechanism of spatial partitioning. Given a partitioning of a space, a particular semantics is induced on the space by associating attributes required by the desired semantics with the cells of the partitioning. So, for example, polyhedra are obtained simply by associating a boolean attribute with each cell. Set operations on polyhedra are then constructed on top of the operation of merging spatial partitionings. *We present then the algorithm for merging two bsp trees independent of any attributes/semantics, and then follow this by the additional algorithmic considerations needed to provide set operations on polyhedra. The result is a simple and numerically robust algorithm for set operations.*

Introduction

Methods for representing geometric objects is an issue of considerable importance to disciplines dealing with geometric computation. Several different representations, such as boundary-representations (b-reps), octrees, and csg trees, are currently in use, and a number of new approaches are being explored by various researchers. As in all computation, the data representation/structure determines the algorithms that are needed to provide the operations associated with any semantic domain. And it is the efficiency and simplicity of the algorithms operating on the data structures that determines the attractiveness of a particular representation.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

Constructive solid geometry introduced the explicit use of the paradigm of constructing complex objects from combinations of other usually simpler objects. This methodology is built upon the mathematics of set theoretic expressions. These expressions are analogous to parenthesized boolean expressions, but the variables are instead subsets of a Euclidean D-dimensional space and the operations include, in addition to the analogous regularized boolean set operations, affine transformations. Instancing, i.e. the utilization of named sub-expressions, is also a part of this method.

These expressions define a value and they can, at least in principal, be evaluated to produce this value. For example, ray-casting evaluates the expressions in a 1D sub-domain of the typically 3D domain, and so solves a simpler problem: classify a line with respect to the expression. When the operands are restricted to polyhedra and are represented by b-reps, then any number of algorithms are known for evaluating such an expression (see for example [Mantyla 88] or [Hoffman 89]).

The methodology underlying b-reps is that of the direct representation of the topology of a polyhedral surface/boundary. The topological approach requires the decomposition of a D-space polytope into connected components of all dimensions d, $0 \le d \le$ D, and explicitly encodes the connectivity/incidence among these components. Thus, the methodology distinguishes for every d, $0 < d \le$ D, affine subspaces containing sets of d-manifolds (shells), preferably with their relative containment (which shells are inside which other shells), along with their connected set of d-1 dimensional boundary elements and the connectivity of these to other elements outside of their affine subspace, and so on recursively in dimension.

B-reps, while widely used, possess a number of inherent difficulties in terms of their representational power. The reliance on the concept of manifolds is at odds with the need for permitting non-manifold boundaries, i.e. the presence of regions on the boundary whose neighborhood is not homeomorphic to an ε-ball of some affine subspace. (However, this problem is fixable.) A second is the inability to represent sets whose boundary is unbounded, such as a linear halfspace.

On the algorithmic side, performing set operations with b-reps requires explicit detection of the co-incidence of all C(D, 2) combinations of the variously dimensioned elements (e.g. face-face, face-edge, edge-vertex) along with some appropriate action for each. And the fundamental importance of incidence to the topological methodology exacerbates the already difficult problem of numerical robustness. Additionally, efficiency considerations necessitate some kind of spatial search structure, one that is extrinsic to the representation and is typically an axis-aligned spatial decomposition. Therefore, it does

ACM-0-89791-344-2/90/008/0115 $00.75

transform with the representation and so must be reconstructed after each transformation.

An alternative that has been evolving throughout the decade of the 80's is the *binary space partitioning tree*. The fundamental methodology underlying bsp trees is spatial partitioning. Hyperplanes are used to recursively subdivide D-space to create a disjoint set of D-dimensional cells. Each cell is then designated as either in the interior of the set or in the exterior. The boundary of the set need not be represented explicitly as it is derivable from the cells. The representational power of linear bsp trees is the class of linear sets[1], which includes linear polytopes. The methodology of spatial partitioning ignores all topological properties of the set, and so bsp tree algorithms treat all topologically distinct sets identically, nor is any distinction made between convex and non-convex sets. Thus the entire representational domain is treated uniformly, providing a considerable improvement in the simplicity of the algorithms. In addition, the spatial search structure is intrinsic to the representation and so transforms with it.

I. BSP Trees

The most intuitive way to understand bsp trees is through the process that constructs them, and so we begin our introduction to bsp trees with an example. Figure 1 illustrates the construction of a bsp tree. One begins with a region of space r, chooses some hyperplane h that intersects r, and then uses h to induce a binary partitioning on r that yields two new regions: $r.child^- = r \cap h^-$ and $r.child^+ = r \cap h^+$, where h^- and h^+ are the negative and positive *open* halfspaces of h respectively. Each of these unpartitioned children can in turn be partitioned, and so on, to produce a binary tree of regions.

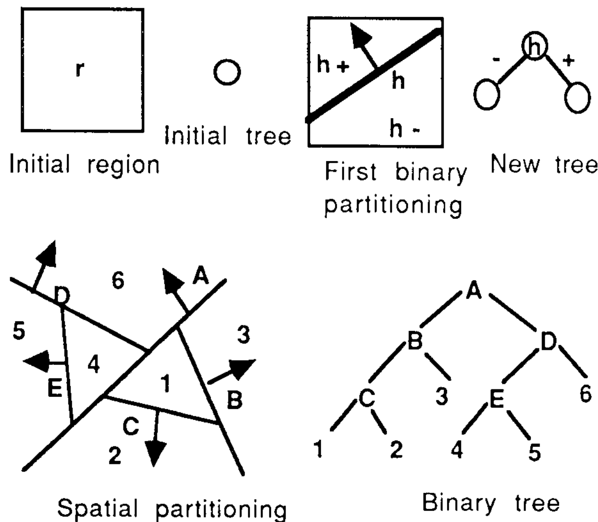

Figure 1 Constructing a bsp tree

A bsp tree is then a hierarchical set of regions of a D-dimension Euclidean space with a relation of parent-child defined on the set corresponding to "child formed by a binary partitioning of parent". The graph of the relation on the set is a binary tree. The process that builds bsp trees uses a single local operator, viz. *binary partitioning*, which provides the construction: (region, hyperplane) -> ($region^-$, $region^+$, binary partitioner). A *binary partitioner* of a d-region R is any d-1 subset of R which partitions R into two disjoint subsets, R^- and R^+, such that any path between two points, $p^- \in R^-$ and $p^+ \in R^+$, must intersect the binary partitioner. Recursively applying this operator produces a bsp tree.

While the bsp tree is a geometric entity whereas its binary tree is combinatorial, the language of binary trees is often useful for describing certain aspects of the bsp tree. By definition, there is an isomorphism between bsp tree regions and binary tree nodes, and we denote the region of a node V as V.region and conversely the node of a region R as R.node. Each internal node V has an associated binary partitioner that partitions V.region, while each leaf node corresponds to unpartitioned region. These unpartitioned regions are called *cells*. (In figure 1, the cells are labeled with numbers.) Each edge of the binary tree corresponds to a halfspace: a left edge to the negative halfspace of the parent node's hyperplane and a right edge to the positive halfspace. We can then define any region R as the intersection of open halfspaces corresponding to edges on the path from the root to R.node. (In figure 1, cell 3 = $A^- \cap B^+$.) Thus, if the initial region, typically all of D-space, is a convex and open set, it follows that all of the regions of the tree are convex and open sets.

The binary partitioner of a partitioned region R, denoted as R.bp, is comprised of a hyperplane, bp.hp, a sub-hyperplane (or sub-hp), bp.shp, which is the intersection of R.bp.hp with R, and its two halfspaces $bp.hs^-$ and $bp.hs^+$. Every region R is the root region of some bsp tree T, denoted as R.tree, and the symmetrical relation is denoted as T.root_region (to unambivalently denote the set of points corresponding to T.root_region, as opposed to the data structure, we may also use T.root_region.domain). The two subtrees are denoted as T.neg_subtree and T.pos_subtree lying in $T.root_region.bp.hs^-$ and $T.root_region.bp.hs^+$ respectively. The set of cells corresponding to the leaves of T together with the sub-hyperplanes of its internal nodes forms a partitioning of T.root_region, and is denoted as T.partitioning.

Review of previous work

The original context in which the bsp tree was developed is that of rendering. The linearity of both planes and viewing rays means that if a ray intersects a plane it does so at only one point. And so the plane divides the ray into near and far sections. This permits inducing a visibility priority ordering on the three subspaces formed by the plane: near halfspace -> plane -> far halfspace. Given a bsp tree T, determining this ordering at every node of the tree in a recursive manner provides a total ordering of the elements of T.partitioning (see [Schumaker et al 69] or [Sutherland, Sproull, Schumaker 74], and [Fuchs, Kedem, Naylor 80] or [Naylor 81]).

These techniques were extended to ray-tracing polyhedra and non-linear csg-dags in [Naylor and Thibault 86]. This work led to the association of attributes at the cells and the overt idea of bsp trees as a representation of polytopes. In [Thibault and Naylor 87] and [Thibault 87], several new algorithms were introduced. Conversation from a b-rep to a bsp tree and point classification algorithms were derived by extending earlier very similar algorithms. The work with csg-dags led to an algo-

---

[1] We have only studied bsp trees of finite size (as in number of nodes); but the concept can be extended to trees that are countably infinite.

rithm for evaluating a csg expression in which the primitive objects are polyhedra each represented by a b-rep, to yield a single bsp tree corresponding to the expression's value. An earlier idea of inserting moving objects into a bsp tree led to an algorithm for evaluating a polyhedral set operation between a bsp tree and a b-rep to yield a bsp tree, i.e. bspt <op> b-rep -> bspt. Finally, algorithms were given for generating the polyhedral boundary as either a set of convex polygons represented by a list of vertices or as a set of edges.

In [Bloomberg 86], very similar ideas are developed, and an algorithm for bspt <op> bspt -> brep is given which classifies faces of one tree with respect to the other; but no subtrees are classified atomically. Even more recent work on bsp trees has provided a means of generating shadows for polyhedral models [Chin and Feiner 89] , interactive object design and view volume clipping [Naylor 90a], radiosity [Fussell and Campbell 90], as well as algorithms with asymptotically improved bounds for constructing bsp trees from a set of faces in 3D and edges in 2D (i.e. conversion from b-rep to bsp tree) [Paterson and Yao 89] and [Paterson and Yao 90]. In [Torres 90], a new treatment is given of the original problems addressed in [Schumaker et al 69] of constructing an inter-object bsp tree of moving objects, where the individual objects are represented as bsp trees.

Geometric model as attributes on a space

The motivation for inducing a partitioning on a space S is to provide a means of distinguishing points in S through the association of arbitrary attributes with any of its points; that is to provide the mapping *Model(X ∈ S) -> Attributes*. We use the bsp tree to implement this general function. We associate with each element of our partitionings (cells and sub-hps) a set of C^0 or higher continuous functions whose domain is considered to be restricted to that element. This provides a quite general mechanism for constructing complex discontinuous functions on S that are piecewise C^0. However, we will restrict our attention in this paper to the problem of represent regular sets, which requires the simplest possible set of attributes, *Membership : { In, Out }*. Nonetheless, the principal result of this paper is the merging of two independent bsp tree spatial partitionings both defined on S. This merging operation is completely independent of the semantics of any attribute space, and requires only the ability to determine whether the attribute space of two elements can be represented by a single attribute space. Set operations are then constructed on top of this merging operation.

II. Merging Trees

The most primitive operation then is merging two spatial partitionings : given partitionings of the same space, P1 and P2, form a new partitioning P3 = P1 + P2 from the pairwise intersection of the cells of P1 and P2, i.e. a cell $c3 \in P3 \Leftrightarrow \exists\ c_1 \in P1,\ c_2 \in P2,\ s.t.\ c3 = c_1 \cap c_2,\ c3 \neq \emptyset$. Merging can be illustrated by simply overlaying the two partitionings on top of each other, as shown in figure 2.1.

We will then merge two trees T1 + T2 -> T3, s.t. T3.partitioning = T1.partitioning + T2.partitioning. However, since bsp trees are a hierarchy of regions, we will need to do somewhat more than merely merge their partitionings. Nonetheless, the algorithm to perform merging of bsp trees is fairly simple and recursive.

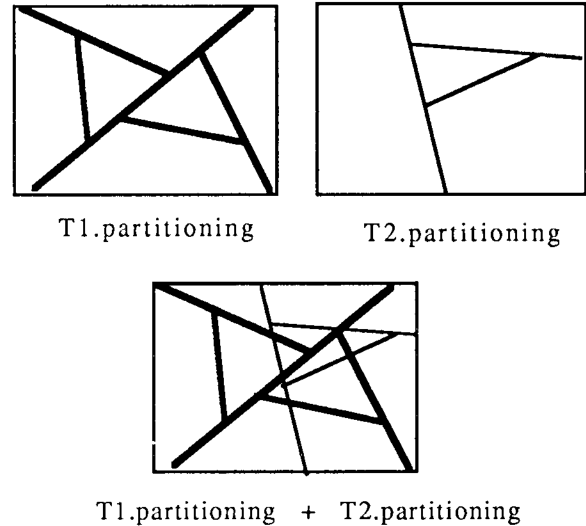

Figure 2.1 Merging partitionings

As with most bsp tree algorithms, we can understanding tree merging in terms of the paradigm of inserting an object into a tree; in this case, the object is a tree as well. (Below, we will relax this asymmetrical view). As always, we need the two basic bsp tree operations: performing a binary partitioning of the object if at a partitioning node and executing a cell <op> object when at a leaf.

Performing a binary partitioning of a bsp tree by the binary partitioner of a node provides (Bspt, Bp) -> (inNegHs, inPosHs : Bspt); that is, a tree is split by a binary partitioner to yield two trees $T^- = T \cap bp.hs^-$ and $T^+ = T \cap bp.hs^+$. A Cell <op> Tree routine is imported by the tree merging routine, and it is this routine that embodies the semantics of the application. Its function is to merge the single set of attributes of a cell with the attributes of a tree. When the semantics is that of set operations on polyhedra, the spatial structure of the result will be either that of the cell or the tree (the specifics are discussed below in section V).

Given these two operations, the algorithm partitions one tree, say T2, by the binary partitioner at the root of the other, T1. The two resulting trees, $T2^-$ and $T2^+$, are defined on exactly the same region (domain) as T1.neg_subtree and T1.pos_subtree respectively. Thus, we have created two new sub-problems, each identical in form to the original problem: merge two trees each of which partition the same subspace. When a cell is reached, the semantics-dependent Cell <op> Tree routine is called. The basic algorithm is given in Figure 2.2. An illustration of tree merging appears in Figure 2.3. As one can see, each cell of T1 is replaced with that subset of T2 that lies in that cell.

While figure 2.2 provides the essentials of the merging algorithm, there remain a number of secondary issues. The first of these arises from the fact that the algorithm is completely symmetric with respect to its two operands, so one has the option of choosing at each recursive invocation of Merge_Bspt(), whether to partition the first tree by the second or the second by the first. A method Choose_Partitioner() can be provided to Merge_Bspt() for this purpose, and may enforce whatever policy is appropriate for the current usage. (Note that since the merge operations may be used to provide a non-commutative operator, the order of the operands must be preserved by having two distinct CASEs, one with T1 as the partitioner and one for T2.)

```
Merge_Bspts : ( T1, T2 : Bspt ) -> Bspt
 ::=
 Types
  PartitionedBspt : ( inNegHs, inPosHs : Bspt )

 Imports
  Merge_Tree_With_Cell : ( T1, T2 : Bspt ) ->  Bspt        User defined semantics.
  Partition_Bspt : ( Bspt, Bp ) ->  PartitionedBspt

 Definition
   IF T1.is_a_cell  OR  T2.is_a_cell
     THEN
       VAL := Merge_Tree_With_Cell( T1, T2 )
     ELSE
       Partition_Bspt( T2,  T1.root_region.bp )  ->  T2_partitioned
       VAL.neg_subtree  :=
                   Merge_Bspts(  T1.neg_subtree,  T2_partitioned.inNegHs  )
       VAL.pos_subtree:=
                   Merge_Bspts(  T1.pos_subtree,  T2_partitioned.inPosHs  )
       VAL.root_region  :=  T1.root_region
     END If
  RETURN VAL
END  Merge_Bspts
```

Figure 2.2 Merging BSP Trees Algorithm

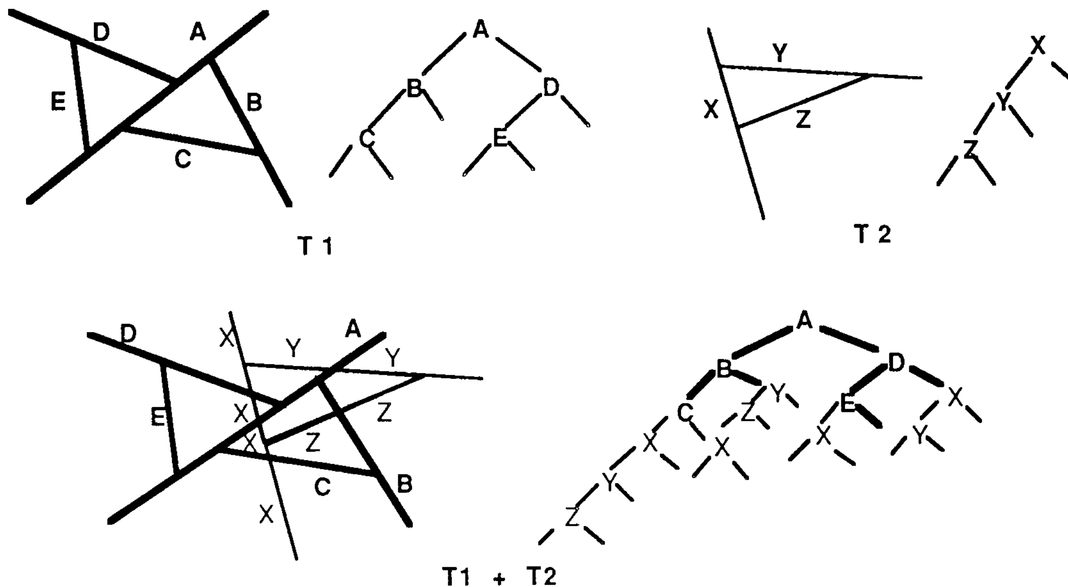

Figure 2.3 Merging two trees

Secondly, it may be necessary to perform merging of attributes in the sub-hp of the bp that is used as the partitioner. This can be handled by a Merge_Bp_Attributes() method. For representing polyhedra, these attributes are the faces of the polyhedra and the requisite routines are discussed below in section VI.

Finally, it is desirable to perform *condensation*. When the attributes defined on Tree.root_region.domain are homogeneous, there is no reason to maintain a partitioning of the domain, and so we will condense the tree into a single cell. Under the recursive assumption that the two subtrees are already condensed, determining homogeneity requires first that they both be singular, i.e. comprised of a single node, and then that their attributes be identical. If attributes defined on the domain of the binary partitioner are independent of those of the subtree, then this subspace must be taken into account as well when determining homogeneity. Note that in the case of polyhedra, binary partitioner attributes, i.e. the faces of the polyhedra, are *not* independent; they are a function of their neighborhood of cells.

III. Binary Partitioning of a BSP Tree

We now address the problem of partitioning a bsp tree. Given a bsp tree T and a binary partitioner P defined on the same region of space, we want to form two trees, T^- and T^+ such that $T^- = T$

and $T^+ = T \cap P.hs^+$, where Regions(T^-) ≡ { $r^- \mid r^- = r$ ... $r \in$ Regions(T), and $r^- \neq \emptyset$ } and similarly for ... T^+).

...ompute the two trees resulting from this operation, we ...e again use the notion of inserting a geometric entity ... tree; in this case, the entity is a binary partitioner. ...sertion process will identify which regions of T lie en... P.hs$^-$, or entirely in P.hs$^+$, or are intersected by P. ...hat insertion visits exactly those regions that are inter... by P). Accomplishing this requires determining the rela...tial relationships of two bp's and, when they intersect, ...g each bp by the hyperplane of the other. This opera... well as the representation and generation of sub-hp's is ...sed below in section IV.

...e have the usual form of first distinguishing between cell ...rtitioning nodes (singular and non-singular trees), and in ...ase of a partitioning node, performing a binary partition... the inserted entity, i.e. the partitioning bp. Partitioning ...s trivial: one needs only to return two copies of that cell. ...rtitioning node, however, the issue is more involved. ...first step is to perform a *bi-partitioning* between P and ...hat is, each bp is classified with respect to the other ... standard binary partitioning cases:

...cation : { **InNegHs, InPosHs, InBoth, On** }.

... 3.1 shows the four possible geometric configurations. ...shown are InNegHs/InPosHs, InPosHs/InPosHs, and On-...rallel, since they have the same geometry but with one ...rmal flipped.) The routine to perform this operation, Bi-...rtition_Bps() is discussed in section IV.

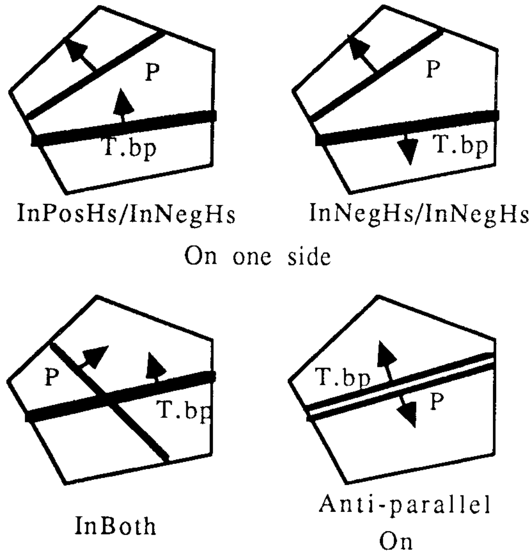

Figure 3.1 Spatial relationships between two binary partitioners

While each of the seven cases are treated separately, they all share the basic premise that any subtree containing the partitioner will need to be partitioned, and any that does not will need no modification. So, the case where P's location = InNegHs results in T.neg_subtree being partitioned but not T.pos_subtree, and InPosHs requires the opposite action, InBoth entails partitioning both, and On neither. The parts of subtrees resulting from this recursive partitioning are then pieced together to form the two trees which are the return values of this operation.

To see this more clearly, figure 3.2 attempts to illustrate what is taking place for the InBoth case in which four subtrees are generated, two from each subtree of T. During the process of inserting the bp P into the tree, one views the activity primarily in terms of the two halfspaces of T.root region: we construct $P^- = P \cap T.hs^-$ and $P^+ = P \cap T.hs^+$. In contrast, the result, which is formed after any required subtree partitioning, is instead in terms of the halfspaces of P: $T^+ = T \cap P.hs^+$ and $T^- = T \cap P.hs^-$, which also entails computing $T.bp^- = T.bp \cap P.hs^-$ and $T.bp^+ = T.bp \cap P.hs^+$. So we have T^- being formed out of pieces from both of T's two subtrees :

T⁻.neg_subtree := T's_neg_subtree.inNegHs
T⁻.pos_subtree := T's_pos_subtree.inNegHs
T⁻.root_region.bp := T's_bp.inNegHs

and similarly for T^+.

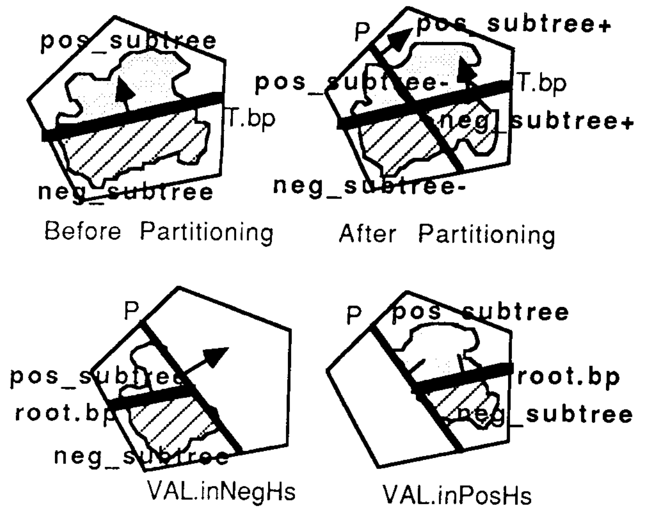

Figure 3.2 Partitioning a tree for InBoth case

The cases in which P is entirely to one side of T.bp is illustrated in figure 3.3. There are four instances of this case obtained by flipping normals; only one is shown here. For this case T.bp and T.neg_subtree remain intact; only T.pos_subtree is partitioned. The return values are:

T⁻.neg_subtree := T.neg_subtree
T⁻.pos_subtree := T's_pos_subtree.inNegHs
T⁻.root_region.bp := T.bp
and
T⁺ := T's_pos_subtree.inPosHs.

Analogous assignments yield the other three instances.

And finally, the third case of On requires no further partitioning and is given simply by selecting the appropriate subtrees:

```
IF normals are parallel
 THEN
       T- := T.neg_subtree
       T+ := T.pos_subtree
 ELSE
       T+ := T.neg_subtree
       T- := T.pos_subtree
 END
```

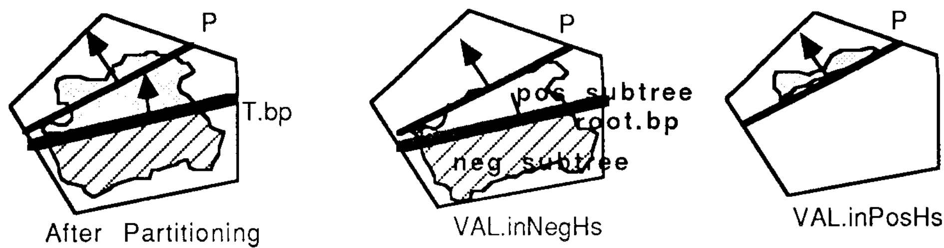

Figure 3.3 Partitioning a tree for InPosHs case

It is important to note that any newly formed tree should have the condensation operation applied to it. While not needed for correctness, this can have a significant impact on performance. Consider figure 3.4 in which two complex objects are each contained inside their bounding simplex. If T2 is inserted into T1, then T2 will be partitioned by X then Y and then Z. At this point, the fragment of T2 inside T1's bounding simplex will be condensed to a single out-cell, and so the merging operation will be complete. Neither of the subtrees inside the bounding simplices will be visited during this process.

With the description of Partition_Bspt complete, we make the following observation : to merge T1 with T2, we can insert T2 into T1, which entails the apparent paradox of inserting T1 into T2 (actually, T1's bp's), but one piece at a time.

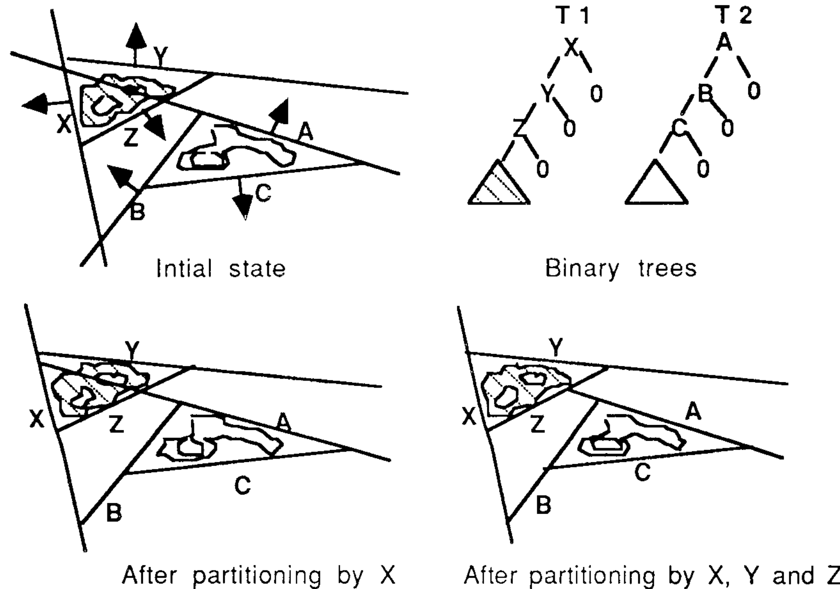

Figure 3.4 Effect of condensing during partitioning

IV. Representation and partitioning of binary partitioners

Partition_Bspt relies upon Bi-Partition_Bps as the basic operation for determining the relative location of two Bps and for splitting them when the location is InBoth halfspaces. To provide this, we will need an explicit representation of the domain of a bp, i.e. of its sub-hp. This is unlike all previous operations on bsp trees, which require only hyperplane equations and possibly a single "representative point" in the interior of a sub-hp (as in the set operations in [Thibault and Naylor 87]). Determining the respective locations of two binary partitioners that partition the same region R can be based on computing their intersection :

$$\mathbf{P1.shp} \cap \mathbf{P2.shp}$$
$$= (\mathbf{R} \cap \mathbf{P1.hp}) \cap (\mathbf{R} \cap \mathbf{P2.hp})$$
$$= \mathbf{P1.shp} \cap \mathbf{P2.hp}.$$

As the value of R must appear in the expression, we "encode" it into a sub-hp. However, when there is no intersection, we need to know in which halfspace the sub-hp lies, and in this case the routine that computes P1.shp $\cap$ P2.hp can tell us only the location of P1 with respect to P2.hp. Therefore, we will need to either use the representative point method or perform the complementary operation P2.shp $\cap$ P1.hp.

In the current implementation, a sub-hp is represented by a b-rep, and for the sake of simplicity we restrict the embedding space to be 3-dimensional so that the sub-hps are polygons. Since sub-hps are convex, we can represent them using the simplest representation: a list of vertices.

Given explicit sub-hps, the bi-partitioning operation is comprised of two applications of the same operation: partition a polygon by a plane. First P1.shp is partitioned by P2.hp. We then performing the same operation for P2.shp with respect to P1.hp

There is a one problem with using b-reps: sub-hps may be unbounded sets and b-reps can not represent unbounded sets (e.g. the sub-hp of T.root_region.bp when T.root_region.domain = 3-space is a hyperplane). Our solution to this is to represent 3-space as a bounded set, in particular as a box centered at the origin whose size is sufficiently large to accommodate the geometric model. We call this the *universe-box*. Since geometric models typically require no more than 7 orders of magnitude (e.g. from 1 mm. to 1 km), constructing a sufficiently large box is easily done without compromising numerical robustness appreciably, especially when using double precision.

To generate a representation of the sub-hp for the bp at some tree node v, we need to first construct a polygonal representation of the bp's hp [Thibault 87]. This is done by projecting one of the sides of the universe-box onto the hyperplane. The side chosen is the one whose the ratio of its area to that of its projection is closest to unity. To achieve this, we choosing the side whose normal makes the smallest angle with that of the hyperplane.

Given this "hyperplane as bounded polygon", we insert it into the tree, partitioning it at each node as usual, but following only the path that leads to the target node v. Thus, when our incipient sub-hp is InBoth, we retain only that half which is in the region of the next node on the path. When we reach v, we have the desired sub-hp. Given any bsp tree containing no explicit sub-hp's, we will perform this operation for every node in the tree. (Rather than following a path from the root to the "current" node, we actually follow the path in the opposite direction, from the node to the root via parent links. This of course is a unique path and avoids the issue in the root-to-node order of knowing whether to follow the left or right child.)

Anytime an affine transformation is applied to a tree, those sub-hp's which are "unbounded" will need to be recomputed since they will no longer correspond to the intersection of their hyperplane with the universe-box. To facilitate this, each sub-hp is tagged to indicate whether it is unbounded, and if so, then regenerated when transformed. (If one uses the first D+1 nodes to construct a bounding simplex, then only the first D sub-hps of this simplex are unbounded, i.e intersect the universe box.) All other sub-hp's can be transformed normally (by their vertices), since they will remain bounded (under the assumption that the universe-box is sufficiently large).

V. Set Operations on Polyhedra

Once the mechanism for merging spatial partitionings is in place, performing set operations on polyhedra is a relatively simple matter. The merging process recurses until one of the two operands is homogeneous, i.e. is a cell, at which point we use a routine for merging cell attributes with those of some arbitrary tree (which may also be a cell). For set operations, this amounts to simply selecting either the cell or the tree, possibly complemented, as a function of the membership attribute (Figure 5.1). Complementation of a tree involves simply the boolean complementation of the membership attribute. Figure 5.2 illustrates the union of two bsp tree objects.

In general, there is more to do than this. If there are other attributes, such as color, index of refraction, density, or whatever, these will need to be merged in some appropriate way as well. And so the above routine will need to be augmented to handle these. (Exactly how a particular attribute, such as color or transparency, should be merged in a union for instance, is currently an unsettled issue). Note that this additional merging of attributes may generate condensable subtrees.

```
Cell_SetOp_Tree: ( T1, T2 : Bspt ) -> Bspt
 ::=
  VAL :=
    IF  T1.is_an_InCell
     THEN
      CASE  operation
         Union -> T1
         Intersection  -> T2
         Difference  -> Complement_Bspt( T2 )

         Symmetric_Difference  ->
                Complement_Bspt( T2 )
      END
    ELSEIF  T1.is_an_OutCell
     THEN
      CASE  operation
         Union -> T2
         Intersection  -> T1
         Difference  -> T1
         Symmetric_Difference  ->  T2
      END

    ELSE Repeat the above with T1 and T2 swapped.

  END  Cell_SetOp_Tree
```

Figure 5.1 Cell_SetOp_Tree for Set Operations

VI. Polyhedral Faces

While the above routine covers "attribute maintenance" for D-dimensional cells, there remains the same issue for the D-1 domains of the binary partitioners. For polyhedra, the entire boundary of the set lies in the sub-hps, and while the boundary is wholly derivable from the D-cells, one may wish to explicitly represent the set of boundary faces. The primary motivation for doing so is to provide the input required by rendering systems that are based on polygon drawing. Using the bsp tree, one can provide a visibility priority ordering of the faces to such a system. (Note that if instead one uses ray-tracing, the explicit representation of the boundary is unnecessary.)

The boundary of a set is precisely those points whose ε-neighborhood contains both interior and exterior points for all ε. Thus, any subset of a sub-hp which has an in-cell on one side and an out-cell on the other will be on the boundary of the polyhedra. In figure 6, we illustrate this idea by showing the boundary along with normals to the faces oriented to "point" to the exterior. Note that a sub-hp may contain more than one face and that the face orientations may be either parallel to the hyperplane orientation or anti-parallel.

There are two possible approaches to face generation: either in every tree maintain faces as an intrinsic component of the representation, or delay creation of any faces until after an entire expression has been evaluated. Both approaches utilize a *neighborhood operation* that finds those cells in the neighborhood of a sub-hp, or equivalently, those cells whose boundary intersects a sub-hp. Consider any subtree T. If we insert the sub-hp at T.root_region into T.neg_subtree and then into T.pos_subtree, the cells that are reached are precisely those in the sub-hp's neighborhood. These cells can be naturally grouped into those in the positive subtree and those in the negative subtree. In addition, the search of a subtree will partition the sub-hp into subsets that bound a single cell, and so will classify the sub-hp into "in" and "out" subsets.

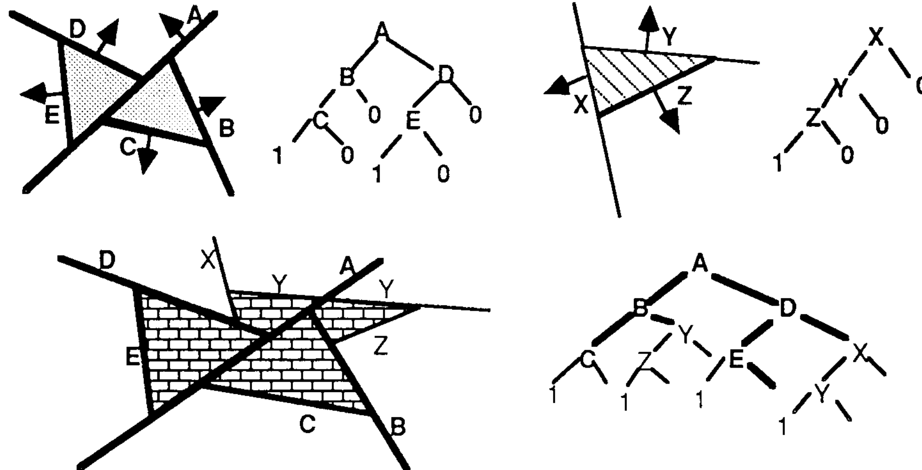

Figure 5.2 Union of two objects

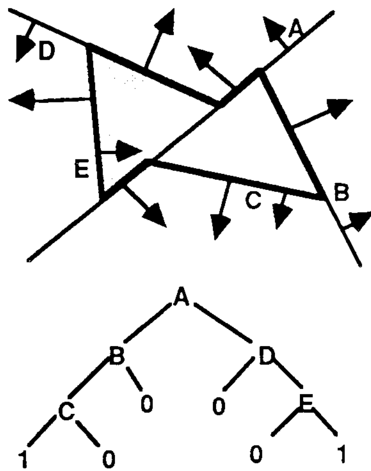

Figure 6 Faces of a polytope

So with this, generation of faces from a sub-hp is straightforward [Thibault 87]. First, classify the sub-hp with respect to one of T's subtrees, say the negative subtree, and then classify the resulting fragments with respect to the positive subtree. This yields fragments whose neighborhood is "homogeneous", i.e. is the same for all points in the fragment. With this information, we can generate the following classifications:

neg_subtree	pos_subtree	classification
in	in	in
in	out	on (parallel)
out	in	on (anti-parallel)
out	out	out

Retaining the *on* fragments, separated into parallel and anti-parallel lists, as part of each binary partitioner provides the polyhedral faces. These faces, in fact, provide only a partial classification of the partitioner's domain. There are two kinds of on-regions, parallel and anti-parallel, but in-regions and out-regions are lumped together implicitly as *not-on*. Note that these face fragments are convex, since they are generated by the intersection of halfspaces with their supporting hyperplane. To generate all of the faces of a tree that contains explicit sub-hps but not explicit faces, one simply performs this neighborhood operation for every binary partitioner in the tree.

The alternative is to maintain explicit faces at all times. So the result of a single set operation, T3 := T1 <op> T2, will entail generating all of the polyhedral faces of T3 before T3 is used in any subsequent set operations. The objective is the same as before, to know the classification of all sub-hps, but the approach is less direct. Instead of classifying sub-hps, we employ the fact that the faces of T3 are a subset of those of T1 and T2 combined, and in effect classify only the subsets of sub-hps "covered" by the faces. Note that the addition of faces to the representation will require extending both Complement_Bspt() to swap parallel and anti-parallel face lists and Bi-Partition_Bp() to partition any faces lying in a binary partitioner (this will be needed only if its sub-hp is InBoth, and thus the sub-hp provides a "convex hull" test for the faces).

We will use a combination of two techniques already introduced: cell <op> tree and the neighborhood operation. The first is employed under two circumstances. When, during the recursive process of tree merging, one of the two arguments is a cell, the routine Cell_SetOp_Tree() is called. This executes"the set operation by returning one of the two arguments (possibly complemented), and so implicitly classifies any faces in the process. The second circumstance occurs during Partition_Bspt() whenever the partitioning bp reaches a cell. Its face fragments can then have the same rules applied to them. The only remaining case is that of on-faces. Whenever the On case in Partition_Bspt() occurs we will need to use the neighborhood operation in leu of direct classification by cells. On-faces from T1 (T2) will need to be classified with respect to the subtrees of T2 (T1) (see [Thibault and Naylor 87] and [Naylor 90a]).

VII. Numerical Robustness

The occurrence of numerical errors due to finite arithmetic has been a nemesis of geometric computing since its inception. Its negative impact is greatest when the result of a numerical computation is used to discriminate logical alternatives in an algorithm. This can lead to arbitrarily "discontinuous" behavior; that is, the output of the algorithm can be highly sensitive to