

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.

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 , $0 \leq d \leq D$, and explicitly encodes the connectivity/incidence among these components. Thus, the methodology distinguishes for every d , $0 < d \leq D$, 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 not

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.



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 sets¹, 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.

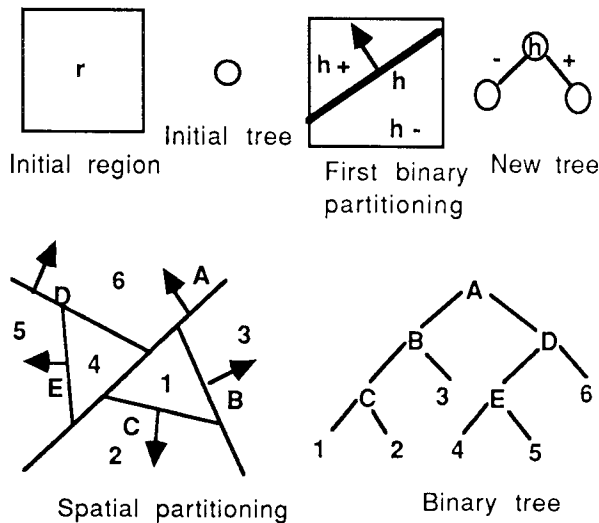


Figure 1 Constructing a bsp tree

A bsp tree is then a hierarchical set of regions of a D-dimension Euclidean space with a relation of parent-child defined on

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.

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) \rightarrow (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 \rightarrow plane \rightarrow 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. Conversion 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-

algorithm 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. $\text{bspt} \langle \text{op} \rangle \text{b-rep} \rightarrow \text{bspt}$. 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 $\text{bspt} \langle \text{op} \rangle \text{bspt} \rightarrow \text{brep}$ 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 $\text{Model}(X \in S) \rightarrow \text{Attributes}$. 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 representing regular sets, which requires the simplest possible set of attributes, $\text{Membership} : \{ \text{In}, \text{Out} \}$. 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 $c3 \in P3 \Leftrightarrow \exists c1 \in P1, c2 \in P2, \text{ s.t. } c3 = c1 \cap c2, c3 \neq \emptyset$. 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 $T1 + T2 \rightarrow T3$, s.t. $T3.\text{partitioning} = T1.\text{partitioning} + T2.\text{partitioning}$. 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.

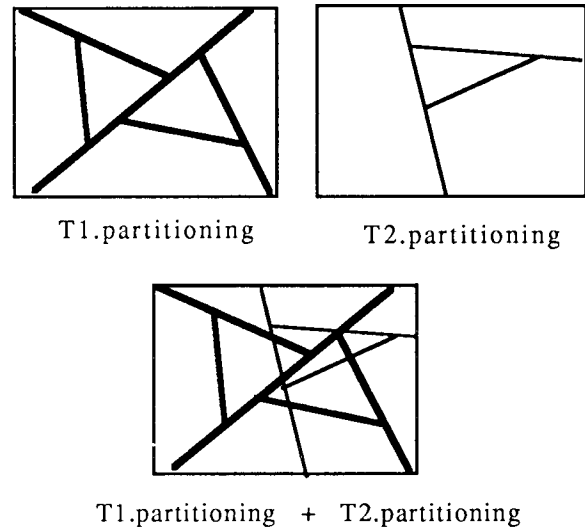


Figure 2.1 Merging partitionings

As with most bsp tree algorithms, we can understand 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 $\langle \text{op} \rangle$ object when at a leaf.

Performing a binary partitioning of a bsp tree by the binary partitioner of a node provides $(\text{Bspt}, \text{Bp}) \rightarrow (\text{inNegHs}, \text{inPosHs} : \text{Bspt})$; that is, a tree is split by a binary partitioner to yield two trees $T^- = T \cap \text{bp.hs}^-$ and $T^+ = T \cap \text{bp.hs}^+$. A Cell $\langle \text{op} \rangle$ 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 $T1.\text{neg_subtree}$ and $T1.\text{pos_subtree}$ 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 $\langle \text{op} \rangle$ 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

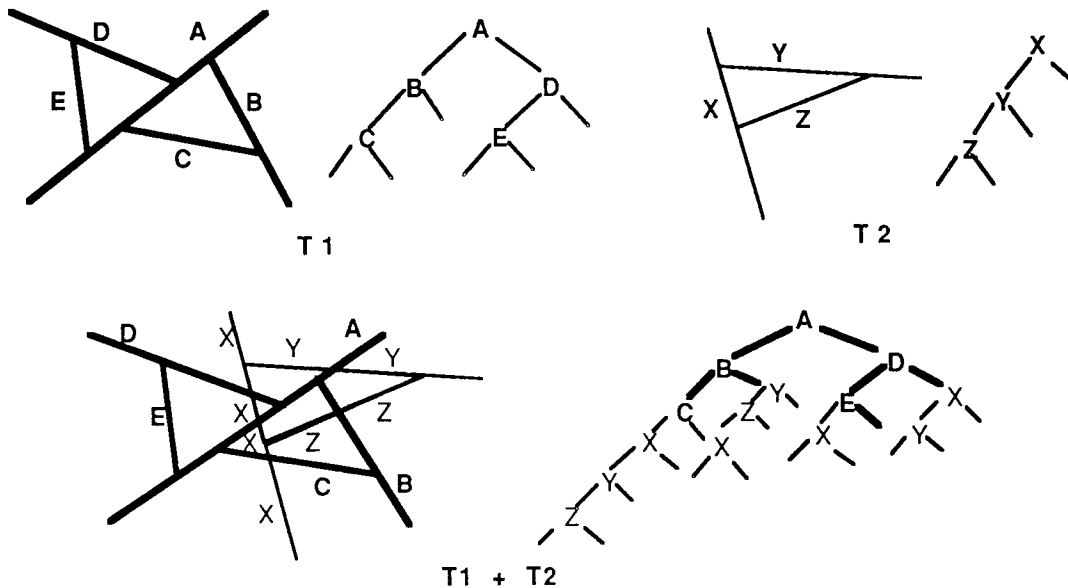


Figure 2.3 Merging two trees

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 $T^- = T$