1994

Incremental Construction of Multi-Dimen- sional Space Partitioning Trees

George Vanecek

Shankara Shastry M.C.

Report Number:
94-077

INCREMENTAL CONSTRUCTION OF MULTI-DIMENSIONAL SPACE PARTITIONING TREES

George Vanecek, Jr.
Shankara Shastry M.C.

CSD-TR-94-077
November 1994

Incremental Construction of Multi-dimensional Space Partitioning Trees[1]

George Vanecek, Jr.[2]
Shankara Shastry M. C.

Purdue University
Department of Computer Sciences
West Lafayette, IN 47907-1398

A multi-dimensional space partitioning (MSP) tree is a data structure that provides a spatial index to the vertices, edges and faces of a boundary representation (B-rep) representing solids. Currently, to build an MSP tree, an algorithm recursively partitions an entire B-rep and results in a unified MSP-tree/ B-rep structure called the Brep-index. There are applications, however, for which it is desirable to insert several B-reps one entity at a time rather than globally as a set of objects.

A solution to the global problem is to use an incremental algorithm to create the MSP tree. The incremental approach has the appealing property that multiple objects can be inserted into the MSP tree by incrementally inserting the entities of all the objects in any order. Of course, the order affects the size and the shape of the resulting tree. In effect, the resulting partition yields the regularized union.

In this paper, we summarize the MSP tree, provide an incremental algorithm that inserts the faces, edges and vertices of the B-reps and review the rules for compressing the tree.

1. Introduction

In the paper on Brep-index [Van91bi], we introduced the multi-dimensional space partitioning (MSP) tree data structure and gave an algorithm for its creation. The algorithm partitions space induced by a boundary representation (B-rep) by recursively subdividing the entire B-rep. In that paper we suggested the possibility of an incremental algorithm for creating the MSP tree but did not give details.

Since then, research on the Brep-index has been continuing with emphasis on its application of supporting contact-region detection in the simulations of virtual environments

1. ONR Grant N00014--94--1--0576.

2. Email: vanecek@cs.purdue.edu,
http://www.cs.purdue.edu/people/vanecek

[CV93]. One aspect that was not considered initially has since then been addressed, namely, the size of the underlying B-rep. Suppose that we wish to do a physical-based simulation of a virtual robot moving inside a very large and detailed model of a building. With the original algorithm, it is not practical to load the entire B-rep into primary memory and to construct the Brep-index.

In this paper, we present the incremental algorithm for creating the MSP tree. We also show that this algorithm generalizes to an n-body MSP tree that indirectly supports the Boolean set operations of those n-bodies. This further simplifies the construction of large models by allowing the complete model to be constructed from smaller components while preserving unique spatial volumes. In the remainder of this section, we present the background and motivation of MSP trees and the Brep-index. We then present the incremental creation algorithm in Section 2. We summarize the work in Section 3 on page 18.

1.1 Motivation and Background

This work is driven by the desire to better understand then to implement an efficient collision and contact analysis support for the simulations of virtual environments. A computer application involving virtual environments must deal with spatial issues concerning objects. Consider a geometric model of a building, a ship, or a village, and populated with movable objects such as chairs, levers, or robots. In dealing with spatial issues in such environments, these applications spend majority of their time processing spatial queries, such as determining the distance to a given object, classifying a line against an object, or determining the contact areas of two touching objects. To make the applications practical, the spatial queries must be made sufficiently efficient and robust.

Although there are many different representation schemes for objects, the boundary representations (B-reps) are used most frequently in applications where the visualization of objects is required. With the B-reps alone, it is computationally difficult and time consuming to support spatial operations. The computational efficiency of spatial queries on a polyhedral objects can, however, be greatly enhanced by appropriately preprocessing the objects in terms of spatial proximities of their components. One approach is to combine an MSP tree with a B-rep and to use it to improve spatial access to the B-reps. We call such a unified representation the Brep-index. Another approach is to incorporate multiple resolutions into the representation. Finally, a third approach is to reduce the number of B-rep entities that have to be processed by considering relative velocities of movable objects and using a new algorithm based on the computer-graphics technique of back-face culling.

The MSP trees and the Brep-index grew out of work with the Newton physical-based simulation system, and the desired support for near real-time collision and contact-region detection [Van91a]. Given arbitrary nonconvex polyhedra, the early use of binary space partitioning (BSP) trees revealed that while collision detection was possible, the precise detection and analysis of contact regions was not. The problem with the BSP trees is that no information is retained about the content of the partitioning planes. This recognition lead directly to the generalization of the BSP trees leading to the MSP trees.

Once the existence of MSP trees was known, the direct relationship between the partitioning induced by the MSP trees and the corresponding B-rep was recognized. This lead to the merging of the MSP tree and the B-rep to yield the Brep-index. In the Brep-index, the MSP tree serves as a spatial index to the B-rep. Its use directly supports the classification of points, line segments, and polygons, and indirectly supports higher-level operations such as collision and contact-region detection.

Although the MSP trees are ideal for supporting classification of entities, they cannot support all the needed operations. For one, they cannot support distance computations. Even though the MSP trees are spatial structures, the recursive convexification of space has no relationship to the Voronoi regions induced by the objects; and the Voronoi regions have to be known, even if only implicitly, to compute the shortest distance to the boundary. Another deficiency with the MSP trees is the lack of region adjacency information. The idea of walking from one region to another by skipping to an adjacent region in a given direction is not supported by the tree. Furthmore, the tree-traversal methods as employed by octrees cannot be used do to the nonregular nature of the MSP trees.

1.2 Multidimensional Space Partitioning Trees

An MSP tree represents a recursive partitioning of three-dimensional Euclidean space by a set of oriented planes resulting in a hierarchy of one, two and three dimensional convex regions. A convex three-dimensional region is partitioned by a cut plane into three subregions consisting of two open half-spaces and a plane. We refer to the three regions as lying above, below, or on the cut plane, where above is in the direction of the plane's normal. Similarly, a convex two-dimensional region is partitioned by a transversal plane into two open half-planes and a line; a convex one-dimensional region is partitioned by a transversal plane into two open half-lines and a point.

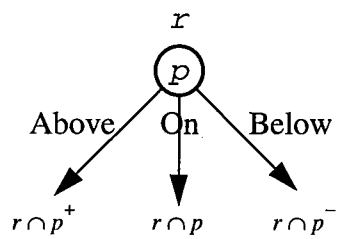

FIGURE 1. Cutting a Region *r* into three subregions by the Cut-plane *p*

This partitioning can be represented by a ternary tree data structure (as shown in Figure 1). Let us consider an example of an MSP tree for a unit cube. Given that the B-rep for the cube labels the Vertices V1 through V8, the Edges E1 through E12, and the Faces F1 through F6, the Brep-index for the cube is shown in Figure 2.

The cube has a total of 26 vertices, edges and faces, and 26 internal nodes. Vanecek has shown that for any Brep-index, the total number of B-rep entities is a lower bound on the number of internal nodes of the MSP tree [Van91bi]. The size of the tree grows as global fragmentation increases.

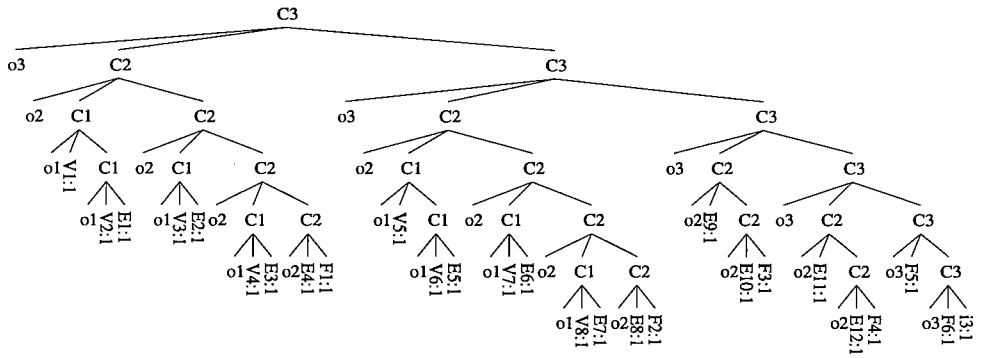

FIGURE 2. Brep-index for a unit cube. C*n* and o*n* and i*n* are nodes representing regions of dimensionality *n*, and *Xi:j* represents the *i*th entity of solid *j* with *X* denoting either a Vertex, an Edge, or a Face. o and i represent outside and inside regions respectively.

Some explanation regarding the notation used in this paper is in order. Since an MSP tree is a spatial index into the B-rep of one or more solids resulting in the Brep-index, inserting a solid into an MSP tree essentially means inserting its B-rep entities into the MSP tree. Our reference to the boundary of a solid itself is at a high level of abstraction (independent of any specific representation) so that we can treat all regions, solids, and boundary entities as point sets.

Since the MSP tree represents a partitioning of space, there is no restriction on the number of solids that induce the tree. When the solids are mutually non-penetrating and do not touch, the insertion algorithm for multiple solids is identical to the algorithm that assumes only one solid. However, when the solids touch or interpenetrate, the insertion algorithm for multiple solids gets more complicated. In this paper, we assume that the solids may touch, but may not interpenetrate. This is due to the asymmetry of the classification relationship caused by the order of insertion. This restriction is explained further in the next section. As an example, consider two unit cubes with one translated by one unit in the x direction so that the two cubes touch, as shown in Figure 3. The MSP tree encodes the regularized union of the two cubes and contains 38 internal nodes, 10 face nodes, 20 edge nodes and 12 vertex nodes (as shown in Figure 4).

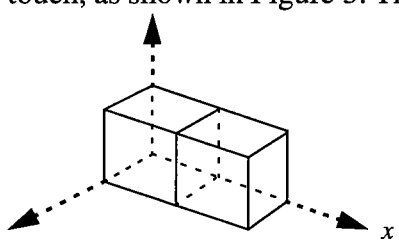

FIGURE 3. Two touching cubes.

Given a region, r, represented by a node in the tree along with an oriented plane, p, p^+ represents the open halfspace above p, and p^- represents the open halfspace below p. The three subregions are formed by intersecting the region with the two halfspaces and the plane (as shown in Figure 1). More formally, for an oriented plane

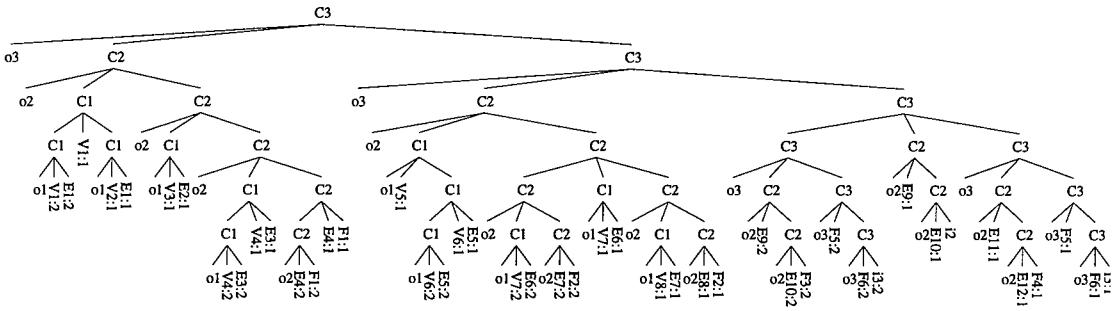

FIGURE 4. The Brep-index for the two cubes of Figure 3.

$$p = \{ (x_1, x_2, x_3) \mid a_1x_1 + a_2x_2 + a_3x_3 + a_4 = 0) \} \quad (1)$$

where (a_1, a_2, a_3) is the outward pointing normal,

the open halfspaces p^+ and p^- are defined as

$$p^+ = \{ (x_1, x_2, x_3) \mid a_1x_1 + a_2x_2 + a_3x_3 + a_4 > 0) \} \quad (2)$$

$$p^- = \{ (x_1, x_2, x_3) \mid a_1x_1 + a_2x_2 + a_3x_3 + a_4 < 0) \} \quad (3)$$

We refer symbolically to an internal tree node, n, given its convex region r, its cut plane p, and the above, on, and below subtrees Ta, To, and Tb respectively, as

$$\text{cut}(\, r, p, Ta, To, Tb\,).$$

Similarly, we refer to a leaf node n symbolically as

$$\text{leaf}(\, r, x\,)$$

where x is either the label out, or in, or a reference to a face, an edge, or a vertex of some boundary representation. Given a node n of the MSP tree that is defined either by the cut or the leaf functions, we use the access functions region(n), plane(n) etc., to retrieve the various components of the node, the predicates isInternal(n), isOutside(n), etc., to make queries on the type of the node and the topological adjacency functions adjV(e), adjE(f), etc., to get the adjacent boundary entities. These functions are defined in the Appendix.

Given a solid S, space can be partitioned into the interior of the solid, the boundary of the solid and the complement of the solid,

$$E^3 = iS \cup bS \cup cS$$

where the boundary is itself partitioned into a set of vertices, edges, and faces,

$$bS = V(S) \cup \cup E(S) \cup \cup F(S).$$

In terms of the point sets, the vertices, the edges and the faces forming the boundary of the solid as well as the interior of the solid are considered open point sets. Thus any two are

pairwise disjoint and the union of all the sets gives the complete solid. In general, an MSP tree T induces a partitioning of space

$$\left(E^3 = \bigcup_{n \in \text{leaves}(T)} \text{region}(n) \right) \wedge$$

$$\forall (n_1, n_2 \in \text{leaves}(T))\ (\text{region}(n_1) \cap \text{region}(n_2) = \varnothing) \tag{4}$$

where leaves(T) is the set of leaf nodes of T. A Region r representing a leaf n of a leaf in leaves(T) is a convex, open subregion of space defined by the intersection of the planes and half-spaces along the path from the root to the leaf. If we let path(n) be the set of nodes along the path to n, we get

$$r = \bigcap \text{region}(\text{path}(n)) . \tag{5}$$

Thus the tree represents the entire three-dimensional space and a partition that encodes a volume, V_n, created from the regularized union of solids S_0, S_1,..., S_n as given by

$$V_n = ((S_0 \cup^* S_1) \cup^* S_2) \ldots \cup^* S_n . \tag{6}$$

2. Incrementally Creating MSP Trees

The MSP tree can be grown incrementally by inserting all the faces, followed by all the edges and finally by all the vertices, one solid at a time. The order of insertion starting with the faces then the edges and finally the vertices is important. During the insertion of a face, the face may be split into two or more faces with new edges added to the B-rep. Similarly, inserting an edge may result in the edge being split into two or more edges with new vertices added. The insertion of a B-rep into the MSP tree may thus fragment the B-rep, and this fragmentation is consistent with the partitioning of space induced by the MSP tree. That is, no face, or edge crosses the boundary of any region. More specifically, there exist regions that have shapes identical to the faces, edges and vertices of the fragmented B-rep.

Given a valid MSP tree T, and a B-rep B, the B-rep can be inserted into the tree with the regularized union set operation by

insertSolid(B, T) =insertSet(vertices(B''),
insertSet(edges(B'), insertSet(faces(B), T))).

In terms of the topological entities of a boundary representation, the original faces of B are the first to be inserted into the tree. These may fragment into subfaces thus modifying the original B-rep, B, resulting in a modified B-rep, B'. This modified B-rep may contain additional edges and vertices. The entities that are inserted next are the edges of this modified B-rep and may themselves be split and introduce new vertices giving rise to a new B-rep B''. The vertices of this B-rep are inserted last. This insertion may also fragment the B-rep further, giving rise to a modified B-rep B'''.

The sets of faces, edges, and vertices are inserted by

insertSet($\{x_1,...x_k\}$, n) = **if** $k=1$ **then**
insert(x_1, n)
else insertSet($\{x_2,...,x_k\}$, insert(x_1, n)).

Consider the example using a tetrahedron shown in Figure 5. Figure 5(a) shows the MSP tree after the four faces of a tetrahedron have been inserted; Figure 5(b) shows the tree after the six edges have been inserted; and the final tree after the four vertices have been inserted is shown in Figure 5(c).

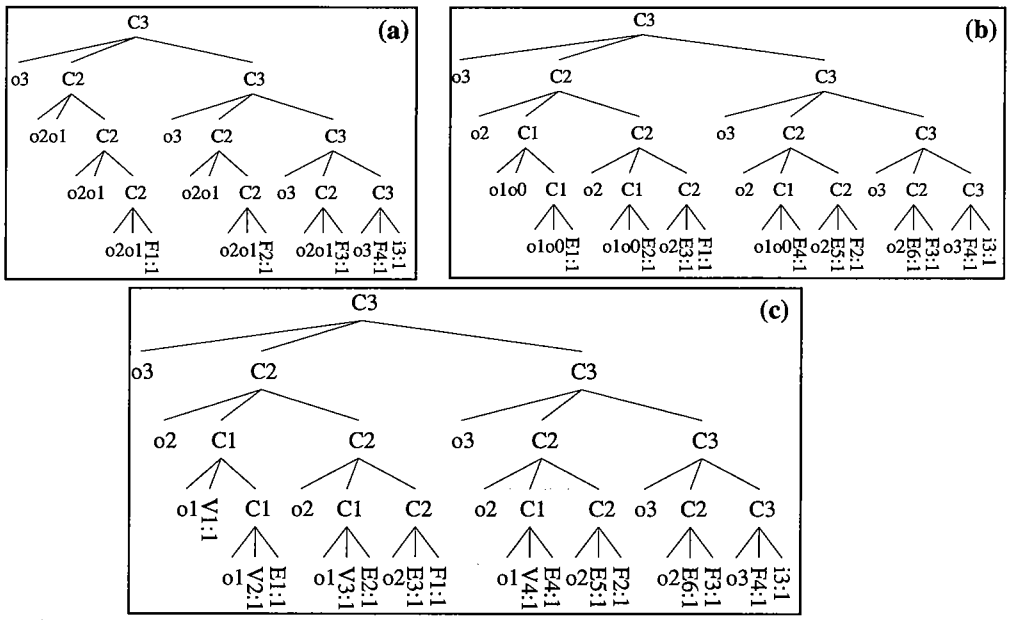

FIGURE 5. The MSP tree shown after the faces (a), edges (b), and vertices(c) of a tetrahedron have been inserted into an empty tree.

In effect, the insertion performs a classification of the entities, grouping some parts of the tree and pruning other parts. For the first B-rep inserted, all the entities are classified as lying outside another solid, and so all the entities are retained by the MSP tree. Subsequently inserted B-reps may be only partially retained if they interpenetrate or touch other B-reps previously inserted. This process can be thought of as performing the following steps:

1. Using the given MSP tree T_1 as a carrier, build another tree T_2 for the new solid.
2. Merge T_1 and T_2 into a single tree and prune branches based on the regularized union set operation.
3. Prune the tree to reduce redundant cuts (see Section 2.4 on page 15).

Of course we modify T_1 directly and we do not build T_2 and merge it.

To do this, we need to reformulate the regularized union set operation into an incremental operation. The set operator requires the classification of faces, edges, and vertices. Con-

sider the case of faces. The classification of the faces of a of Solid A in relation to a Solid B fragments and partitions the faces of A into four sets:

$A\mathbf{in}B$: faces of A that are inside B,

$A\mathbf{out}B$: faces of A that are outside B,

$A\mathbf{with}B$: faces of A that are on B with the same orientation, and

$A\mathbf{anti}B$: faces of A that are on B with opposite orientation.

That is,

$$\text{faces}(A) = A\mathbf{in}B \cup A\mathbf{out}B \cup A\mathbf{with}B \cup A\mathbf{anti}B. \quad (7)$$

Of the eight sets of faces from A and B, the regularized union of the two solids consists of the faces

$$\text{faces}(A \cup^* B) = A\mathbf{out}B \cup A\mathbf{with}B \cup B\mathbf{out}A. \quad (8)$$

The first solid inserted, say A_1, is equivalent to the operations $\varnothing \cup^* A_1$ which from Equation (8) show that $\text{faces}(\varnothing \cup^* A_1) = A_1\mathbf{out}\varnothing = \text{faces}(A_1)$. Thus all the faces are retained. The same holds true for the edges and the vertices. Equation (8) gives a constructive method that forces an a priori classification of all the faces of which some are used to construct the result. The incremental variant of the equation should instead specify what faces of A we need to remove and what faces of B we need to add. In general, a subsequently inserted solid, A_n, causes some faces of the previously inserted solids to be removed, and some of the faces of A_n to be added. By combining the eight face partitions given by faces(A) and faces(B) in Equation (7), and noting that $A\mathbf{anti}B = B\mathbf{anti}A$, the incremental regularized union equation follows directly from Equation (8):

$$\text{faces}(A \cup^* B) = \text{faces}(A) - A\mathbf{in}B - B\mathbf{anti}A \cup B\mathbf{out}A. \quad (9)$$

This means that from the MSP tree representing solid A, faces that correspond to the sets $A\mathbf{in}B$ and $B\mathbf{anti}A$ are deleted and the faces that correspond to the set $B\mathbf{out}A$ are added. In practice it turns out that the deletion of the faces corresponding to $B\mathbf{anti}A$ and the insertion of the faces that correspond to $B\mathbf{out}A$ can be done locally, but the deletion of the faces that correspond to the set $A\mathbf{in}B$ is not possible locally, because the faces are inserted one by one and the classification of the faces of A with respect to the solid B is not known until the entire solid B is inserted. This necessitates an entire search through the MSP tree after B is inserted to replace the set $A\mathbf{in}B$ with the in classification. Now, such a situation occurs only when the solids are interpenetrating. For the addressed problem of physical-based simulations for virtual environments, the solids can not interpenetrate and so the assumption of non-interpenetrability is a reasonable one.

With the assumption of non-interpenetrability, the following holds for any two solids A and B:

$$A\mathbf{in}B = B\mathbf{in}A = A\mathbf{with}B = B\mathbf{with}A = \varnothing. \quad (10)$$

Now the incremental equation for the regularized union further simplifies from Equation (9) and Equation (10):

$$\text{faces}(A \cup^* B) = \text{faces}(A) - A\text{anti}B \cup B\text{out}A. \quad (11)$$

The details of inserting the faces, edges, and vertices of subsequent solids are described in the next section.

2.1 Inserting a Face

Given a Face f, it must induce a local partitioning of space so that the area of the face is strictly covered by one or more leaf regions of the MSP tree:

$$\exists N \subset \text{leaves}(T)\left(f = \bigcup_{n \in N} \text{region}(n) \right). \quad (12)$$

If the face is convex and does not undergo any fragmentation, this is equivalent to saying that the face is equal to exactly one leaf region:

$$\exists n \in \text{leaves}(T)\,(f = \text{region}(n)). \quad (13)$$

Typically, the fragmentation occurs when a face is split by a cut plane resulting in convex 2D and 1D regions as given by the split function:

$$\text{split}(f, P) = \text{separate}(f \cap P^+) \cup \text{separate}(f \cap P^-) \text{ where}$$
$$\text{separate}(r) = \{ f \mid f \subseteq r \textbf{ and } f \text{ is connected } \textbf{and } f \text{ is maximal} \}$$

As an example of inserting a face, inserting face F_1 of the tetrahedron into an empty tree results in the subtree shown in Figure 6.

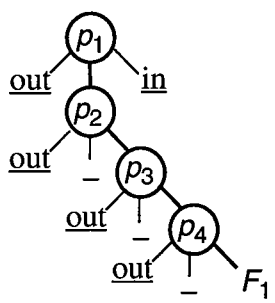

FIGURE 6. Partial tree created by the insertion of face F_1.
Here, $F_1 = p_1 \cap p_2^- \cap p_3^- \cap p_4^-$

The syntax of the face insertion function is:

insert: Face × MSPTree → MSPTree

The algorithm to insert Face f of Solid S_{i+1} into the MSP tree starting at Node n is given by the recursive function insert(f, n) shown in Figure 8. It has three major steps:

1. Propagate the face through the tree starting at the root.

   At each internal node if the face lies completely above, on, or below the cut plane of that node descend down either the *Above* (S1), the *On* (S2), or the *Below* (S3) subtree respectively. If the face crosses the cut plane, split the face into two or more faces lying completely above or below, and starting at that node recursively insert each one in turn (S4).

   Once a leaf node of the MSP tree is reached, perform either step 2 if the leaf represents a face belonging to another solid or step 3 if the leaf represents an *outside* or an *inside* region.

2. Once a leaf node of the MSP tree representing a face g of another solid has been reached, one of the following is performed:

   a. If f and g have the same contain the same regions and have opposite orientation, ignore f and replace the face-node n with an *inside* region (S5).

   b. If one of the adjacent edges e of the face f does not lie on a plane transversal to the support plane of the face g,
   $(\exists (e \in \text{adjE}(f)))\ (\forall m \in \text{path}(n))$

   $$(\text{plane}(g) \neq \text{plane}(m) \wedge e \not\subset \text{plane}(m))$$

   split Face g into (f_a, e_o, f_b) replace n with a new internal node
   cut(_, p, leaf(_, f_a), leaf(_, e_o), leaf(_, f_b)),
   with a cut plane p that is transversal to the face f at the edge e and insert f at node n (S6). This transversal plane is either the support plane of the adjacent face of the edge e or the perpendicular bisector of the two adjacent faces of the edge e, if the two faces are almost coplanar. For example, consider Figure 7. In Figure 7(a), the perpendicular bisector of the two adjacent faces of the edge e is chosen as the transversal plane p since the faces are almost coplanar. In Figure 7(b), the support plane of the adjacent face itself is the transverse plane. The definitions of transversal plane and support plane functions, trPlane and SupPlane are given in Appendix.

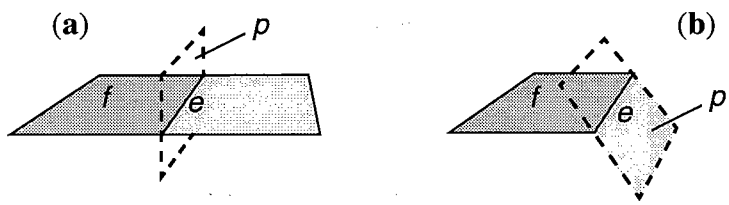

FIGURE 7. Choosing a transversal cut through Face f and an adjacent Edge e of f depending on the angle between its adjacent faces.

3. Once a leaf node of the MSP tree representing an *outside* or *inside* region has been reached, one of the following is performed:

```
insert( f, n ) =
    if isInternal(n) then
        if f ⊆ plane(n)+ then
            cut( region(n), plane(n),
                insert(f, above(n)), on(n), below(n) )                    (S1)
        else if f ⊆ plane(n) then
            cut( region(n), plane(n),
                above(n), insert(f, on(n)), below(n) )                    (S2)
        else if f ⊆ plane(n)− then
            cut( region(n), plane(n),
                above(n), on(n), insert(f, below(n)) )                    (S3)
        else
            insertSet( split( f, plane( n )), n )                         (S4)
    else if isFace(n) then
        if f = face(n) and supPlane( f ) = – supPlane( face(n) ) then
            leaf( region(n), in )                                         (S5)
        else if ∃e ∈ adjE( f ) (e ⊂ region(n) ) then
            let p = trPlane( f, e )
                {g+, g−} = split(f, p)
                e' = edge(n) ∩ p
            in
                insert( f, cut( region( n ), p,
                                leaf( region(n) ∩ p+, g+ ),
                                leaf( region(n) ∩ p, e'),
                                leaf( region(n) ∩ p−, g− )))              (S6)
        if dim( n ) = 3 then
            cut( region(n), plane( f ),
                leaf( region(n) ∩ plane(f)+, out ) ,
                insert( f, leaf( region(n) ∩ plane(f), _ )),
                leaf( region(n) ∩ plane(f)−, in ))                        (S7)
        else if ∃e ∈ adjE( f ) (e ⊂ region(n) ) then                      (S8)
            let p = trPlane( f, e )
            in
                insert( f, cut( region(n), trPlane( f, e ),
                                leaf( region(n) ∩ p+, out ),
                                leaf( region(n) ∩ p , _ ),
                                leaf( region(n) ∩ p−, in )))
        else
            leaf( region(n), f )                                          (S9)
```

FIGURE 8. Function to insert a Face

a. If there does not exist a node m along the path whose cut plane contains f,

$$\neg\exists\,(m \in \text{path}\,(n))\, f \subset \text{plane}\,(m)\,,$$

replace n with a new internal node

$$\text{cut}(\_, \text{plane}(f), \underline{\text{out}}, \_, \underline{\text{in}}).$$

Afterwards insert f starting at node n (S7) .

b. Otherwise, if one of the adjacent edges of the face does not lie on a previous cut plane,

$$\exists\,(e \in \text{adjE}\,(f)\ \forall m \in \text{path}\,(n))$$
$$\text{plane}(f) \neq \textit{plane}\,(m)\ \wedge\ e \not\subset \text{plane}\,(m)$$

replace n with a new internal node

$$\text{cut}(\_, p, \underline{\text{out}}, \_, \underline{\text{in}})$$

where the cut plane p is transversal to the face f at the edge e (S8).

c. Otherwise, create a new leaf node pointing to Face f (S9).

Step 2a needs an elaboration. Consider the Face g of some previously inserted solid lying in a leaf region. If an edge adjacent to the new face being inserted falls into the region, the 2D region must be slit by a transversal plane containing the edge. The splitting of the face may force the splitting its bordering edges. But if one of the edges is split into two sub-edges and a vertex, the original 1D region referencing that edge somewhere else in the tree becomes inconsistent. To reestablish consistency, when an edge or a face referenced by the tree is split, the leaf node referencing those entities has to be split as well. For this reason the entities of the B-reps have backpointers into the leaf nodes of the tree. As an example, consider Face f and two bordering edges e_1 and e_2 which are split by the cut plane p as shown in Figure 9. The modified tree is shown in which the leaf regions for f, e_1

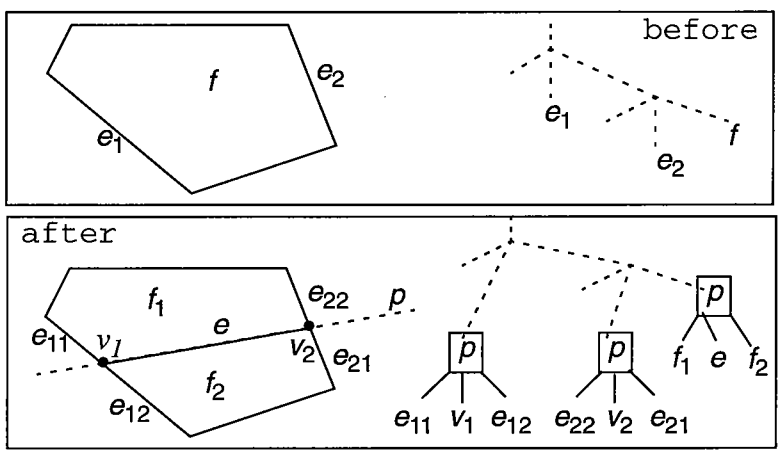

FIGURE 9. Effect on the tree caused by splitting a convex Face *f* by Plane *P*.

and e_2 are replaced by internal nodes.

2.2 Inserting the Edges

After all the faces of a B-rep (i.e., A_n, of the possibly fragmented B-rep) are inserted into the tree, the following conditions hold:

- All faces of A_n are convex.
- No face of A_n intersects a face of any B-rep A_i, for 1,...,n-1.

The syntax of the edge insertion function is:

insert: Edge × MSPTree → MSPTree

The algorithm to insert an Edge e of Solid S_{i+1} into the MSP tree starting at Node n is given by the recursive function insert(e, n) shown in Figure 10. The major steps of the algorithm are:

1. Propogate the edge through the tree starting at the root as in the case of face insertion algorithm ((S10) - (S13)).

   Once a leaf node is reached, perform either step 2 if the leaf region represents a face or step 3 if it represents an edge region or step 4 if it represents an inside or outside region.

2. Once a leaf node of the MSP tree representing a face g of a another reached, split the face g into (g^+, e_g, g^-) and replace n with an internal node

   cut (_, p, leaf(_, g^+), leaf(_, e_g), leaf(_, g^-))

   with a cut plane p that is the support plane of one of the adjacent faces of e and insert e at node n (S14).

3. Once a leaf node of the MSP tree representing an edge e_g of another solid has been reached, one of the following is performed.

   a. If e and e_g contain the same regions and all the adjacent faces of the edge e have been classified as inside, ignore e and replace edge node n with an inside region (S15), otherwise, simply ignore e (S16).

   b. If one of the adjacent vertices v of the edge e does not lie on the end planes of the edge e_g,

      $(\exists (v \in \text{adjV}(e)))\ (\forall m \in \text{path}(n))\ (\text{plane}(m) \neq \text{endPlane}(e))$

      split edge e by the end plane of e at the vertex v and insert e at the new internal node n (S17). The end plane of the edge e at the vertex v is the plane that is adjacent to the edge e and vertex v not containing the edge e. The endPlane function is defined in Appendix.

4. Once a leaf node of the MSP tree representing an outside or inside region has been reached, one of the following is performed.

```
insert( e, n ) =
    if isInternal( n ) then
        if e ⊆plane(n)+ then
            cut( region(n), plane(n),
                insert( e, above(n) ), on(n), below(n) )                    (S10)
        else if e ⊆ plane(n) then
            cut( region(n), plane(n),
                above(n), insert( e, on(n) ), below(n) )                    (S11)
        else if e ⊆ plane(n)− then
            cut( region(n), plane(n),
                above(n), on(n), insert( e, below(n) ))                     (S12)
        else
            insert( e ∩ plane(n)−, insert( e ∩ plane(n)+, n ) )             (S13)
    else if isFace( n ) then
        let p = supPlane(f) | f ∈adjF(e)
            {g+, g−} = split( f, p )
                e_g = edge(region(n) ∩ p)
        in
            cut( region( n ), p,
                leaf( region(n) ∩ p+, g+ ),
                insert( e, leaf( region(n) ∩ p, e_g) ),
                leaf( region(n) ∩ p−, g− )))                                (S14)
    else if isEdge( n ) then
        if e = region( n ) then
            if ∀f ∈adjF(edge(n)) hasClass( f, in ) then
                leaf( region(n), in )                                       (S15)
        else
            n                                                               (S16)
    else if ∃v ∈adjV( e ) (v ⊂ region(n) ) then
        let p = endPlane( e, v )
        in
            insert( e, cut( region(n), p,
                    leaf( region(n) ∩ p+, region(n) ∩ p+ ) ),
                    leaf( region(n) ∩ p, region(n) ∩ p ) ),
                    leaf( region(n) ∩ p−, region(n) ∩ p−) ))                (S17)
    else assert( isOutside(n) or isInside( n ) )
        if dim( n ) = 2 then
            let n = leaf(r, x)
                p = supPlane(f) | (f ∈adjF(e))
            in
                insert( e, cut( region(n), p,
                        leaf( region(n) ∩ p+, x),
                        leaf( region(n) ∩ p, _ ),
                        leaf( region(n) ∩ p−, x) )                          (S18)
```

```
else if ∃v ∈ adjV( e ) ( v ⊂ region(n) ) then
    let p = endPlane( v, e )
    in
        insert( e, cut( region(n), p,
                        leaf( region(n) ∩ p+, out ),
                        leaf( region(n) ∩ p, _ ),
                        leaf( region(n) ∩ p−, in ))))                    (S19)
else
    leaf( region(n), e )                                                 (S20)
```

FIGURE 10. Algorithm to insert an edge

a. If the leaf region is a two dimensional region representing an entity x (outside or inside), replace n with a new internal node
cut (_, p, x, _, x)
where p is support plane of the adjacent faces of the edge e and insert e starting at node n. This case occurs when the edge being inserted lies on one of the cut planes representing either an outside or an inside region (S18).

b. Otherwise, if one of the adjacent vertices of the edge does not lie on the endplanes of the region n, replace n with an internal node
cut(_, p, out, _, in)
where p is the endplane of the edge e at the vertex v and insert e starting at node n (S).

c. Otherwise, create a new leaf node pointing to edge e (S).

2.3 Inserting the Vertices

The syntax of the vertex insertion function is:

insert: Vertex × MSPTree → MSPTree

The algorithm to insert a vertex v of Solid S_{i+1} into the MSP tree starting at Node n is given by the recursive function insert (v, n) as shown in Figure 11. The major steps in this algorithm are similar to those in the edge insertion algorithm.

2.4 MSP Tree Compression

As Equation (9) for the incremental regularized union shows, the insertion of a face (and similarly an edge, or a Vertex) can cause several entities of a previously inserted B-reps referenced from the leaves to be removed and replaced by interior nodes (e.g., (S14)). This can leave empty regions and redundant cuts that needlessly increase the size of the tree. These regions can, however, be merged and redundant internal nodes removed. In [BV93], Vanecek introduced rules to manipulate the structure of the MSP tree. Here, we adopt these rules to the incremental construction algorithm.

```
insert( v, n ) =
if isInternal( n ) then
    if point( v ) ∈ plane( n )+ then
        cut( region( n ), plane( n ),
            insert( v, above( n ) ), on( n ), below( n ) )
    else if point( v ) ∈ plane( n ) then
        cut( region( n ), plane( n ),
            above( n ), insert( v, on( n ) ), below( n ) )
    else
        cut( region( n ), plane( n ),
            above( n ), on( n ), insert( v, below( n ) ) )
else if isFace( n ) then
    let p = supPlane(f) | f ∈ adjF(v)
        {g+, g-} = split( f, p )
        eg = edge(region(n) ∩ p)
    in
        cut( region( n ), p,
            leaf( region(n) ∩ p+, g+ ),
            insert( v, leaf( region(n) ∩ p, eg) ),
            leaf( region(n) ∩ p-, g- )))
else if isEdge( n ) then
    let p = supPlane(f) | f ∈ adjF(v)
    in
        cut( region(n), p,
            leaf( region(n) ∩ p+, edge(n) ∩ p+ ),
            leaf( region(n) ∩ p, v),
            leaf( region(n) ∩ p-, edge(n) ∩ p-))
else if dim( n ) = 2 or dim( n ) = 1
    let n = leaf( r, x )
        p = supPlane( f ) | f ∈ adjF( v )
    in
        insert( v, cut( region(n), p,
                    leaf( region(n) ∩ plane(n)+, x)
                    leaf( region(n) ∩ p, _),
                    leaf( region(n) ∩ p-, x) )
else assert dim( n ) = 0
    if isOutside( n ) then
        leaf( region( n ), v)
    else if ∀ f ∈ adjF( v ) hasClass( f, in )
        leaf( region( n ), in )
    else
        n
```

FIGURE 11. Algorithm to insert a vertex

The first rule states that any non-zero dimensional region r and Entity x can be cut by a transversal plane p, and conversely a redundant cut through a homogeneous region can be removed,

$$\text{leaf}(\, r, x\,) \Leftrightarrow \text{cut}(\, r, p, \text{leaf}(\, r \cap p^{+}, x\,), \text{leaf}(\, r \cap p, x\,), \text{leaf}(\, r \cap p^{-}, x\,)\,) \qquad \text{(R1)}$$

The orientation of a cut plane in an internal node can be flipped, if the children are also flipped,

$$\text{cut}(\, r, p, x, y, z\,) \Leftrightarrow \text{cut}(\, r, -p, z, y, x\,) \qquad \text{(R2)}$$

Two transversal cuts, p_1 and p_2, passing through a given region can be ordered in two ways, depending on which cut is considered first,

$$\begin{aligned} &\text{cut}(\, r, p_1, \text{cut}(\, r \cap p_1^{+}, p_2, a, b, c\,), \text{cut}(\, r \cap p_1, p_2, d, e, f\,), \text{cut}(\, r \cap p_1^{-}, p_2, g, h, i\,)\,) \\ &\qquad\qquad \Leftrightarrow \\ &\text{cut}(\, r, p_2, \text{cut}(\, r \cap p_2^{+}, p_1, a, d, g\,), \text{cut}(\, r \cap p_2, \; p_1, b, e, h\,), \text{cut}(\, r \cap p_2^{-}, p_1, c, f, i\,)\,) \end{aligned} \qquad \text{(R3)}$$

The above three rules can be applied using simple pattern matching. However, they alone are not sufficient for the compression. An additional rule is needed that requires taking the orientation of the cut planes into account. Given the cut plane of a child node does not intersect the cut plane of the parent node within the parent region, the order of the cuts can be exchanged provided the relative positions of the subregions are known. This rule has two variations, one for the above and one for the below subregions.

Given that $r \cap p_1 \cap p_2 = \varnothing$, and $\text{region}(e) \subset p_2^{+}$,

$$\begin{aligned} &\text{cut}(\, r, p_1, \text{cut}(\, r \cap p_1^{+}, p_2, a, b, c\,), d, e\,) \\ &\qquad \Leftrightarrow \\ &\text{cut}(\, r, p_2, a, b, \text{cut}(\, r \cap p_2^{-}, p_1, c, d, e\,)\,). \end{aligned} \qquad \text{(R4)}$$

Similarly, given that $r \cap p_1 \cap p_2 = \varnothing$, and $\text{region}(e) \subset p_2^{-}$

$$\begin{aligned} &\text{cut}(\, r, p_1, \text{cut}(\, r \cap p_1^{+}, p_2, a, b, c\,), d, e\,) \\ &\qquad \Leftrightarrow \\ &\text{cut}(\, r, p_2, \text{cut}(\, r \cap p_2^{+}, p_1, a, d, e\,), b, c\,). \end{aligned} \qquad \text{(R5)}$$

From symmetry, given that $r \cap p_1 \cap p_2 = \varnothing$, and $\text{region}(e) \subset p_2^{+}$

$$\begin{aligned} &\text{cut}(\, r, p_1, \text{a}, b, \text{cut}(\, r \cap p_1^{+}, p_2, c, d, e\,)) \\ &\qquad \Leftrightarrow \\ &\text{cut}(\, r, p_2, c, d, \text{cut}(\, r \cap p_2^{-}, p_1, a, b, e\,)\,). \end{aligned} \qquad \text{(R6)}$$

Similarly, given that $r \cap p_1 \cap p_2 = \varnothing$, and $\text{region}(e) \subset p_2^{-}$

$$\text{cut}(\, r, p_1, \text{a}, b, \text{cut}(\, r \cap p_1^{+}, p_2, c, d, e\,)) \Leftrightarrow \text{cut}(\, r, p_2, \text{cut}(\, r \cap p_2^{+}, p_1, a, b, c\,), d, e\,). \quad \text{(R7)}$$

From the above four rules, three more useful rules can easily be derived. These are

$$\text{cut}(\, r, p_1, \text{cut}(\, r \cap p_1^{+}, p_2, a, b, c\,), c, c\,) \Leftrightarrow \text{cut}(\, r, p_2, a, b, c\,), \quad \text{(R8)}$$

$$\text{cut}(\, r, p_1, \text{cut}(\, r \cap p_1^{+}, p_2, a, b, c\,), c, \text{cut}(\, r \cap p_1^{+}, p_2, c, d, e\,)\,) \Leftrightarrow \text{cut}(\, r, p_2, a, b, \text{cut}(\, r \cap p_1^{+}, p_2, c, d, e\,)\,), \quad \text{(R9)}$$

and

$$\text{cut}(\, r, p_1, \text{cut}(\, r \cap p_1^{+}, p_2, a, b, c\,), c, \text{cut}(\, r \cap p_1^{+}, p_2, c, d, e\,)\,) \Leftrightarrow \text{cut}(\, r, p_2, \text{cut}(\, r \cap p_1^{+}, p_2, a, b, c\,), d, e\,) \quad \text{(R10)}$$

where *a, b,* and *c* are any subtrees.

With these rules we can modify the insertion algorithms of Section 2.1, Section 2.2, and Section 2.3 to automatically compress the tree. Specifically, we replace the call to insert(*x,n*) with simplify(insert(*x,n*)), where

simplify: MSPTree → MSPTree

The function simplify(*n*) is:

```
if n matches rhs(R1) then
    leaf(region(n), entity(above(n)))
else if n matches rhs(R2) then
    ...
else
    n
```

3. Conclusion

The use of this incremental algorithm for creating an MSP tree enables the spatial preprocessing of multiple nonself-interpenetrating B-reps of arbitrary complexity.

There are several related issues that are currently being studied:

- Tree balance control.
- Very large environments.

Appendix

We use the following (partial definitions of) data types:

Boolean = { false, true },
Region = { r | $r \subseteq E^3 \wedge r$ is convex, compact, bounded and with piecewise-linear boundary }
Face = { f | $f \in$ Region $\wedge$ dim(f) = 2}
Edge = { e | $e \in$ Region $\wedge$ dim(e) = 1 }
Vertex = { v | $v \in$ Region $\wedge$ dim(v) = 0 }
Entity = { out, in } $\cup$ Face $\cup$ Edge $\cup$ Vertex

In addition to the boundary entities Face, Edge, and Vertex we assume the boundary representation provides the edge/vertex, the face/edge, and the edge/face topological adjacency relationships on the boundary entities:

adjV: Edge $\rightarrow$ { Vertex }
adjE: Face $\rightarrow$ { Edge }
adjF: Edge $\rightarrow$ { Face }
adjF: Vertex $\rightarrow$ { Face }

With these definitions, the algebraic specification for the multi-dimensional space partitioning tree abstract data type is:

type MSPTree **imports** Boolean, Plane, Region, Entity

operations:

leaf: Region $\times$ Entity $\rightarrow$ MSPTree
cut: Region $\times$ Plane $\times$ MSPTree $\times$ MSPTree $\times$ MSPTree $\rightarrow$ MSPTree
plane: MSPTree $\rightarrow$ Plane
region: MSPTree $\rightarrow$ Region
entity: MSPTree $\rightarrow$ Entity
vertex: MSPTree $\rightarrow$ Vertex
edge: MSPTree $\rightarrow$ Edge
face: MSPTree $\rightarrow$ Face
above: MSPTree $\rightarrow$ MSPTree
on: MSPTree $\rightarrow$ MSPTree
below: MSPTree $\rightarrow$ MSPTree
isInternal: MSPTree $\rightarrow$ Boolean
isOutside: MSPTree $\rightarrow$ Boolean
isInside: MSPTree $\rightarrow$ Boolean
isVertex: MSPTree $\rightarrow$ Boolean
isEdge: MSPTree $\rightarrow$ Boolean
isFace: MSPTree $\rightarrow$ Boolean

for all $r \in$ Region; $x \in$ Region; *Ta, To, Tb* $\in$ MSPTree; **and** $p \in$ Plane, **axioms:**

plane(cut(_,p,_,_,_)) = p (A21)
region(leaf(r,_)) = r (A22)
region(cut(r,_,_,_,_)) = r (A23)
entity(leaf(_,x)) = x (A24)

vertex(leaf(_,*x*))	= *x* **if** *x* ∈ Vertex	(A25)
edge(leaf(_,*x*))	= *x* **if** *x* ∈ Edge	(A26)
face(leaf(_,*x*))	= *x* **if** *x* ∈ Face	(A27)
above(cut(_,_,*Ta*,_,_))	= *Ta*	(A28)
on(cut(_,_,_,*To*,_))	= *To*	(A29)
below(cut(_,_,_,_,*Tb*))	= *Tb*	(A30)
isInteranl(leaf(_,_))	= false	(A31)
isInternal(cut(_,_,_,_,_))	= true	(A32)
isOutside(leaf(_,*x*))	= (*x* == out)	(A33)
isOutside(cut(_,_,_,_,_))	= false	(A34)
isInside(leaf(_,*x*))	= (*x* == in)	(A35)
isInside(cut(_,_,_,_,_))	= false	(A36)
isVertex(leaf(_,*x*))	= (*x* ∈ Vertex)	(A37)
isVertex(cut(_,_,_,_,_))	= false	(A38)
isEdge(leaf(_,*x*))	= (*x* ∈ Edge)	(A39)
isEdge(cut(_,_,_,_,_))	= false	(A40)
isFace(leaf(_,*x*))	= (*x* ∈ Face)	(A41)
isFace(cut(_,_,_,_,_))	= false	(A42)

Any specification not given above results in an error.

Functions:

supPlane(f) = { $p = (x_1, x_2, x_3) \mid (a_1x_1 + a_2x_2 + a_3x_3 + a_4 = 0)$ **and** ($f \subset p$) }
where (a_1, a_2, a_3) is the outward pointing normal.

trPlane(f, e) = **let** = (adjF(e) | $g \neq f$)
in
if (supPlane(g) = supPlane(f))
p | ($p \perp$ supPlane(g) **and** $p \perp$ supPlane(f))
else
supPlane(g)

endplane(e, v) = (supPlane(f) | ($f \in$ adjF(v) **and** $e \notin$ adjE(f)))

References

[VAN91bi] G. Vanecek Jr. , 'Brep-index: A Multidimensional Space Partitioning Tree (revised)', *International Journal of Computational Geometry and Applications* Vol 1 No. 3 (1991) pp 243-262

[CV93] G. Vanecek Jr., and J. Cremer, 'Project Isaac: Building Siimulations for Virtual Environments',*Tech. report CSD-93-68* Dep. Computer Science, Purdue University, USA (1992)

[VAN91a] G. Vanecek Jr., 'A Data Structure for Analyzing Collisions of Moving Objects', *Hawaii International Conference on Systems Science, HICCS-24* pp 671-180 (1991)

[BV93] William J. Bouma and G. Vanecek Jr., 'Contact Analysis in a Physically Based Simulation', *2nd ACM/SIGGRAPH Symposium on Solid Modeling and Applications*, May 1993