Conversion of Binary Space Partitioning Trees to Boundary Representation

João Comba[1] and Bruce Naylor[2]

[1] Computer Science Department, Stanford University *
[2] Spatial Labs, Inc. **

Summary. Binary Space Partitioning Trees (BSP-Trees) have been proposed as an alternative way to represent polytopes based on the spatial subdivision paradigm. Algorithms that convert from Boundary Representation (BRep) to BSP-Trees have been proposed, but none is known to perform the opposite conversion. In this paper we present such an algorithm, that takes as input a BSP-Tree representation for a polytope and produces a BRep as output. The difficulty in designing such algorithm comes from the fact that the information about the boundary is not explicitly represented in the BSP-Tree. The solution we present involves a recursive traversal of the tree to compute lower dimensional information, along with a gluing algorithm that combine the convex regions defined by the BSP-Tree, removing internal features. A new data structure is proposed (a Topological BSP-Tree), that augments the traditional BSP-tree with topological pointers and is used to store intermediate results used in the reconstruction of the BRep.

1. Introduction

Boundary representation (BRep) is a widely used representation of solid geometry, based on the description of an object by its boundary, as a collection of faces, edges and vertices[3]. On the other hand, Binary Space Partitioning Trees (BSP-Trees) consist of convex hierarchical decompositions of the space, where the object is represented by the union of convex regions. The basic operation for the construction of this decomposition consists of a partition of the underlying space by a given hyperplane. This partition is represented as a binary tree, where each node is identified by a hyperplane, and the left and right subtrees of the node represent the two halfspaces obtained in the partition. The recursive application of this operation creates convex hierarchical decompositions of the space. In Solid Modeling, BSP-Trees have been used along with BReps, and many algorithms have been proposed, like the conversion from BRep to BSP-Tree [11], or the one that computes boolean operation with BSP-Trees [7].

In this paper we consider the problem of converting a BSP-Tree representation of a polytope into a BRep. This problem is similar to the conversion of CSG to BRep (also called the *Boundary Evaluation*), because the BSP-Tree can be first converted into a CSG by computing the union of all the convex

* e-mail: comba@cs.stanford.edu, web: http://www-graphics.stanford.edu/
** e-mail: naylor@spatial-labs.com

regions that the BSP-Tree defines, and applying over the result any of the Boundary Evaluation algorithms proposed in the literature([9], [10]).

However, this approach does not exploit the spatial subdivision information that the BSP-Tree encodes, which may lead to more efficient algorithms. In order to use this information, we propose an algorithm that converts the BSP-Tree to a BRep by working directly in the structure of the BSP-Tree. In

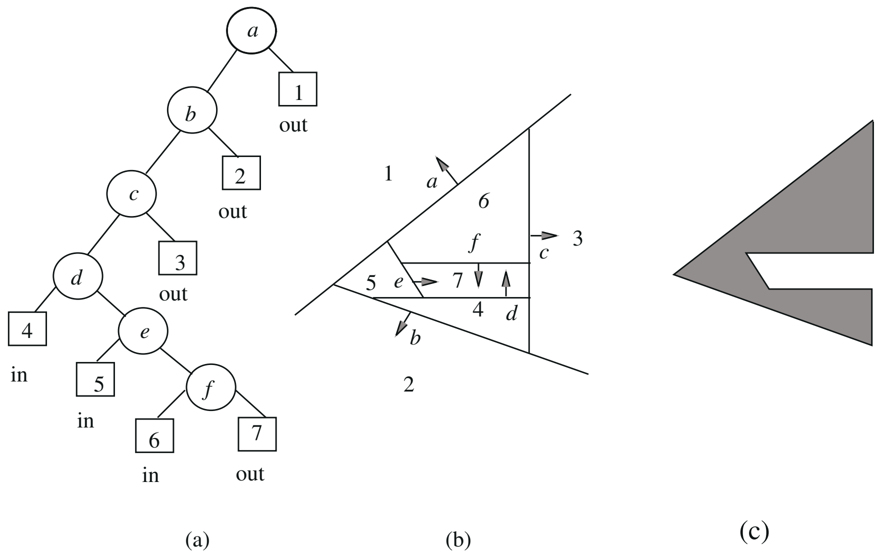

Fig. 1. (a) BSP-Tree, (b) Partition induced by the BSP-Tree. The resulting object is composed of 3 different regions (4, 5, 6), corresponding to the IN nodes (c) Computing the boundary requires gluing different regions and removing internal elements

Fig. 1 we illustrate the problem of reconstructing the boundary of an object obtained by a partition induced by the BSP-Tree. In this example the union of regions 4, 5 and 6 (Fig. 1c) form the object. Looking at the example we identify some difficulties in the design of the conversion algorithm:

- The BSP-Tree does not have lower dimensional information: Intersections need to be computed to obtain the faces, edges and vertices on the boundary.
- The BSP-Tree does not have adjacent-topological information.
- The BSP-Tree represents a convex decomposition of solids as convex regions, and the boundary may be partitioned in multiple independent components that are not minimal.

In the following sections we present how we solved each of the above problems, and give the algorithm to compute a BRep from a BSP-Tree. Initially we review some basic concepts about BSP-Trees. The algorithm that computes the BRep from a BSP-Tree is discussed next, where we describe the Topological BSP-Tree (TBSP-Tree), an auxiliary data structure that stores intermediate results and topological relations among them.

2. Review of BSP-Trees

2.1 Basic concepts

Binary Space Partitioning trees (BSP-Trees) are spatial search structures used in many different aspects of Computer Graphics and Geometric Modeling. Applications in Solid Modeling [6] [7] [11] [13] , Visibility Orderings [1] [2] [12] and Image Representations [8] , among others, can be found in the literature. In order to describe the concepts of BSP-Trees, it is always nice to first explain the relation it has with Binary Search Trees.

Binary Search Trees have been used in Computer Science in many ways, but mostly as a data structure to accelerate search queries based in symbolic values. A geometric interpretation of this data structure is a hierarchy of binary partitions of the real line, where the partitioner is a point and each partition obtained represents an interval. The problem with this interpretation is that it does not directly generalize to higher dimensions, as points do not partition such spaces. The misleading information with this interpretation is that the partitioner that was supposed to be a point is in fact a hyperplane. In general, for a D-dimensional space the partitioner corresponds to the hyperplane to that space (a (D-1)-D element), and the partition has the same dimension of the underlying space. BSP-Trees and Partition Trees [4] use this analogy to extend the concepts of Binary Search Trees to higher dimensional spaces, but we choose to work here with BSP-Trees as they have a more natural correspondence with the representation of solids than Partition Trees.

One advantage of BSP-Trees is the ability to combine a search structure with a representation scheme in one unique data structure. The use of BSP-Trees as a solid representation scheme reveals one application where this combination is well exploited. Solids are represented using BSP-Trees by a union of convex regions, which are identified by associating attributes to the leaves of the tree. These regions may be either inside or outside the solid, and are defined by the hyperplanes that are in the path from the root to the leaf node. In order to precisely define the region, the leaves have associated an attribute that indicates that the region is inside or outside the solid.

2.2 Formal definitions

Many algorithms in BSP-Trees are better explained if we represent the main concepts formally. In this section we present some definitions and properties that are going to be used in the paper.

Definition 2.1. Hyperplanes and Halfspaces: The hyperplane is the basic element to recursively partition the space, and it is described by the following equation:

$$h \;=\; \{(x_1,\; ...,\; x_d) \mid a_1x_1 \;+\; ... \;+a_dx_d \;+\; a_{d+1} \;=\; 0\}$$

The hyperplane separates the space in two halfspaces, the positive and the negative halfspace. Each of them is expressed as:

$$h^+ \;=\; \{(x_1,\; ...,\; x_d) \mid a_1x_1 \;+\; ... \;+a_dx_d \;+\; a_{d+1} \;>\; 0\}$$
$$h^- \;=\; \{(x_1,\; ...,\; x_d) \mid a_1x_1 \;+\; ... \;+a_dx_d \;+\; a_{d+1} \;<\; 0\}$$

The normal of the hyperplane is defined by the vector $(a_1,\; a_2, ...,\; a_d)$. The positive halfspace corresponds to the one that lies in the direction of the normal.

Definition 2.2. BSP-Tree Nodes and Leaves: A BSP-Tree node represents the information of the binary partition being performed on the space. It consists of a partitioning hyperplane, and left and right children that point to BSP-Tree representations of the positive and negatives halfspaces. The hyperplane that defines the node n is denoted with a *H(n)*.

A BSP-Tree leaf contains attributes associated to a given region. It contains the labels *IN* (if the region is inside the solid), and OUT (if the region is outside the solid), but may also contain additional attributes, like color or density.

Definition 2.3. Region Path: A Region Path corresponds to the path that leads from the root of the tree to another node of the BSP-Tree. This path represents one given partition of the space, and is represented by an ordered list of nodes L.

$$RP(n) \;= L \;=\; \{root, ...,\; parent(parent(n)),\; parent(n),\; n\}$$

Definition 2.4. Region: A region of a given node represents the geometric interpretation of the partition defined by the region path RP(n). It corresponds to the intersection of all positive halfspaces in the region path, and can be formulated as:

$$R(n) \;=\; \{\; \cap\; H^\diamond(\nu) \mid H^\diamond(\nu) \;\in\; RP(n),\; \diamond = +, -\}$$

Definition 2.5. Sub-hyperplane: A sub-hyperplane s of a hyperplane h at a given node n is defined by the intersection of the hyperplane h with the region *R(n)* defined in the node.

$$s = SUB(h, n) \;=\; \{h \;\cap\; R(n)\}$$

Definition 2.6. Projected Hyperplane: A projected hyperplane p is a one dimension lower hyperplane that is obtained by projecting the intersection of two hyperplanes *h1* and *h2* orthogonally to one of the coordinate axis.

Definition 2.7. Path Partial Ordering $\prec_p$ *:* Let h_1 and h_2 be two hyperplanes in a common region path p from the root of the tree. We define $h_1 \prec_p h_2$ if h_1 is in the left subtree of h_2. Otherwise, $h_2 \prec_p h_1$.

3. Computing the BRep from a BSP-Tree

The BSP-Tree represents solids as the union of convex regions, and the boundary associated with each of these regions is defined by the intersection of halfspaces in the tree. Each convex region is identified by two attributes: a leaf node l labeled as *IN* (i.e. not empty), and a region path *RP(l)* associated with l. The boundary of this region is obtained by the intersection of the hyperplanes in its corresponding region path.

In the simple case where the BSP-Tree consists of a single convex region we need to consider only one region path, and to obtain the boundary we compute the intersection of all hyperplanes in a the region path (Fig. 2).

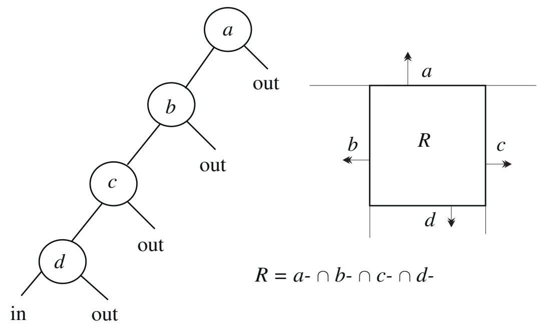

Fig. 2. Simple BSP-Tree generating only one convex region

In a more complicated BSP-Tree, with more than one convex region, it is likely that some of the region paths associated with these regions will share nodes in the tree. In Fig. 3, region paths of the regions R1, R2, R3 and R4 share the nodes a and b, and the intersection of $H(a)$ and $H(b)$ is on the boundary of all these regions. In fact, any node in the tree may contribute to the boundary of all regions that contains the node in its region path. A related consequence of this fact is that when computing a given region we may not need to compute the intersections of all hyperplanes in the region path, as some may not contribute to the region (redundants to the region),

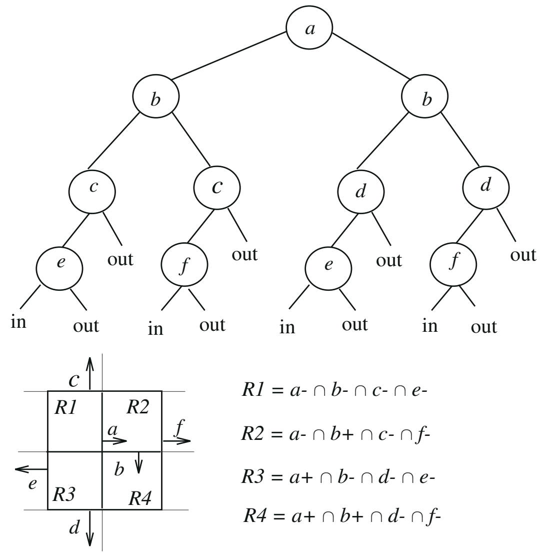

Fig. 3. Complex BSP-Tree generating more than one convex region

The sharing of nodes among different regions implies that the computation of intersections to find the boundary of regions may require repeated computations. This fact reveals a structure common in Dynamic Programming (DP) problems. In each node of the tree we can partition the problem of computing the boundary into smaller problems, corresponding to the boundary in the positive and negative halfspaces of the node. In order to obtain such boundaries, we need to compute intersections among hyperplanes in each of the subtrees (separated subproblems) and ancestor hyperplanes (shared subproblems), which identifies the DP structure.

One way to avoid re-computation of common intersections shared by subproblems is to store them in intermediary data structures. The way of representing these intersections, here called lower dimensional information, is described in details in the next section.

3.1 Storing lower dimensional elements

One method of representing lower dimensional elements in a BSP-Tree relies on using BSP-Trees of lower dimension, which results in a structure called a multi dimensional BSP-Tree. In 1990, Naylor [5] proposed but did not elaborate the use of a pure BSP-Tree model to represent solids. The proposal was based in the extension of the standard BSP-Tree model to represent explicitly the multi-dimensional information defined by the structure of the tree. In 1991, Vanecek [13] used similar ideas and proposed the BRep-Index, which consisted of a multi-dimensional BSP-Tree (called MSP) attached to

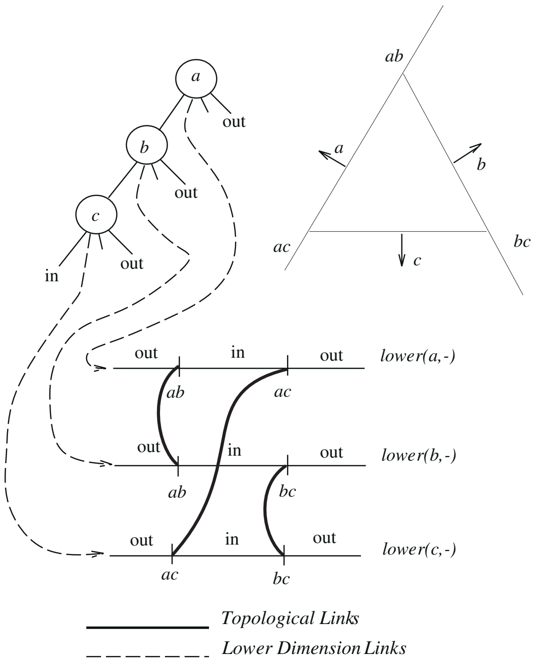

Fig. 4. Topological BSP-Tree (TBSP-Tree) example in 2D

a BRep, with the goal of providing efficient access to the BRep structures. The BRep-Index is constructed in such a way that there is a correspondence between (0,1,2)-d nodes of the MSP with the vertices, edges and faces of the BRep.

In this paper, we represent the lower dimensional information obtained in a different kind of multi-dimensional BSP-Tree. In the BRep Index of Vanecek the topological information is stored in the BRep structure and not in the MSP. In the data structure proposed in this paper, the topological information is stored in a multi dimensional BSP-Tree, augmented with additional topological pointers connecting elements topologically adjacent. We call this data structure a *Topological BSP-Tree* (TBSP-Tree).

The motivation for creating such a structure comes from the fact that in order to obtain lower dimensional information we must compute the intersection of hyperplanes, which gives certain information about the way that elements are topologically related. In the example of Fig. 4, when computing the intersection of the lines a and b we obtain a point ab that we need to insert into the lower dimensional BSP-Trees associated with a and b. In other words, a is topologically adjacent to b by ab. In order to preserve this information, we connect the nodes we obtain in this intersection with topological pointers.

Besides the addition of these topological pointers, the TBSP-tree is different from the MSP of Vanecek because we keep not only one, but two pointers to lower dimensional dimensional BSP-Trees, corresponding to the subdivisions formed over both sides of the hyperplane. One reason for this choice includes the simplification of the incremental algorithm to build the topological BSP-Tree, which requires a partial ordering among the hyperplanes, that can be simplified if we process the subdivisions in both sides separately.

3.2 Navigating the TBSP-Tree

The TBSP-Tree stores topological information about the intersections computed in a pre-order traversal of the tree. This is achieved by creating copies of the same intersection and connecting them with topological pointers. In general, for each intersection of two hyperplanes h_1 and h_2 we keep three copies in the TBSP-Tree. The node higher located in the tree receives one copy, stored in the lower dimensional tree being used in the current region path. The other two copies are stored in the two lower dimensional trees associated with the other node. These last two copies are connected to the first copy created by a topological pointer. In the case where $h_1 \prec_p h_2$, the first copy is stored in the negative lower dimensional tree of h_2, because h_1 is in the left subtree of h_2. The other two copies are stored in the positive and negative lower dimensional trees of $h1$.

Using the information stored in the TBSP-Tree we are able to recover the basic elements of the Boundary Representation. This can be illustrated by looking at the partial TBSP-Tree representation for a cube in Fig. 5. The faces of the cube are defined by the hyperplanes $H(a)$, $H(b)$ and $H(c)$, which are represented in a 3D BSP-Tree by the nodes a, b and c. The intersection $H(a) \cap H(b)$ is represented in the lower dimensional trees associated with a and b as ab and ba respectively. The edges that belong to the face $H(a)$ are defined in the lower dimensional trees associated with node a, which in this case has one empty tree, as the partition occurs in only one of the sides of a.

In order to traverse the faces of the cube directly from the TBSP-Tree we make use of the following operators:

- **TreeParent(node) and TreeSon(node,side):** Traditional pointers in trees for parents and sons.
- **DimensionParent(node) and DimensionSon(node,side):** Pointers that reflect an incidence relation in the dimension of the space. DimensionSon(node,side) returns a pointer to the lower dimensional BSP-Tree associated with the node in the specific side, and DimensionParent(Node) returns an upper-dimensional BSP-Tree associated with the node.

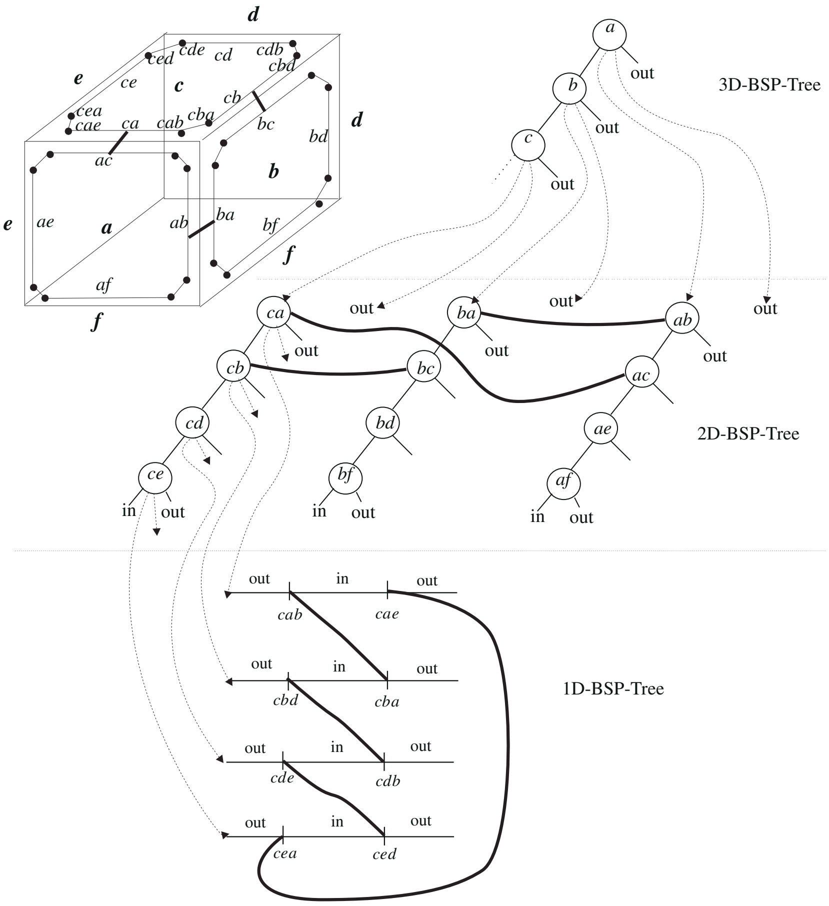

Fig. 5. Topological BSP-Tree (TBSP-Tree) example in 3D

- **TopologicalCopy(node,side):** The application of this operators returns the topological copy of a node at a given side. In Fig. 5, TopologicalCopy (ab,-) returns the copy ba^-, which corresponds to the same intersection ab, but stored in the negative lower dimensional BSP-Tree of b.
- **TopologicalNeighbor(node):** In 1D it is possible to define an adjacency relation. The Topological neighbor of a node corresponds to the next node in ascending order in the real line. This order reflects our convention for the orientation of loops in the representation of the boundary.

These operators allow us to access the boundary elements for the cube. In Fig. 6 we show the procedure to visit the elements of a face (**VisitFace**). The application of this procedure to the leaf node IN (next to node cea) will visit edges (cea, ced), (cde, cdb), (cbd, cba) and (cab, cae), which correspond to the edges of the face a.

```
procedure VisitFace(1D-BSPTree node1D)
  // The starting node corresponds to a leaf containing
  // an IN attribute in the 1D-BSP-Tree
  currentNode1D = startNode1D = DimensionParent(node1D);
  do
    nextNode1D = TopologicalNeighbor (startNode1D);
    VisitEdge (startNode1D, nextNode1D);
    currentNode1D = TopologicalCopy (currentNode1D);
  while (currentNode1D != startNode1D)
end VisitFace;
```

Fig. 6. Procedure to visit a face using the TBSP-Tree

3.3 Incrementally Computing the TBSP-Tree

For every node visited in the traversal of the tree, we discover more information about how the BSP-Tree partitions the space. When a new node is reached, we may compute the intersection of the hyperplane that defines the node against all hyperplanes in the region path of the node. These intersections are used to update the lower dimensional BSP-Trees of all the nodes in this region path.

Every intersection obtained is projected onto one of the coordinate axis before insertion into the lower dimensional trees, corresponding to the concept defined before of a projected hyperplane. One way to understand this operation is to remember the way that Gaussian Elimination (GE) solves linear systems. In GE, the solution of a linear system involves first a triangulation step, where each of the columns under the pivot (the diagonal element) is eliminated (replaced by zeros). In fact, at every elimination step the columns of the matrix, which can be interpreted as the coefficients that define a hyperplane, are projected into one of its dimensions. This is exactly the same operation we are performing here to obtain the projected hyperplanes. In order to compute all lower dimensional information induced by the BSP-Tree on a specific hyperplane we compute the intersections with the other hyperplanes, and project these intersections in a direction orthogonal to one of the coordinate axis. This has the same effect of sweeping with zeros a column in GE. For simplicity, from now on when we refer to intersections we are in fact referring to the projected hyperplanes of the intersections.

The update step of the lower dimensional trees is performed after an intersection is found, consisting of the insertion of projected hyperplanes in lower dimensional trees. This insertion operation is guided by a partitioning operation, which involves the classification of a hyperplane against a partitioning hyperplane. Depending on the result of this classification, the intersection between the hyperplanes is computed and the hyperplane being inserted is partitioned into two sub-hyperplanes. The sub-hyperplanes are recursively inserted into the left and right subtrees of the partitioner node. However, we exploit one unrecognized property of the BSP-Trees that guarantees that we need to perform this insertion into only subtree of the node.

Suppose that in the BSP-Tree described in Fig. 7 we want to compute the lower dimensional information associated with node a. Performing a pre-

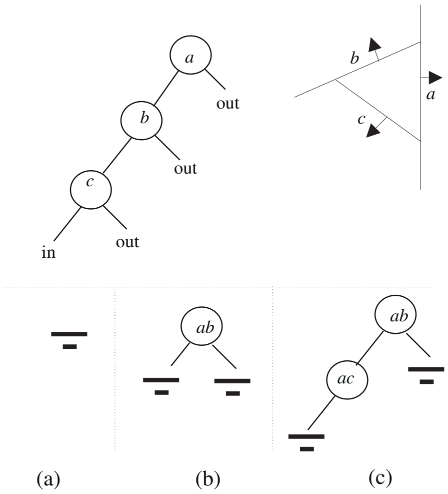

Fig. 7. Simple case in the incremental construction of the TBSP-Tree. The status of the lower dimensional BSP-Tree associated with a after the visit of nodes a, b and c is illustrated in (a), (b) and (c)

order traversal of the tree we visit in this order nodes a and b in the tree. In this case b intersects a, therefore we compute the intersection ab and insert as a node into the negative lower dimensional BSP-Tree of a, as $b \prec_p a$. The next node visited is c, and the insertion of ac into the lower dimensional tree of a involves a partition against ab, to decide in which side of ab we need to insert ac. Using the the path partial ordering defined by the tree, we observe that $c \prec_p b$, and therefore we must have that $ac \prec_p ab$, which means that ac needs to be inserted into the left subtree of ab.

But it is easy to formulate a case where the insertion operation can not be guided by such partial ordering. In Fig. 8, b does not intersect a. This is only possible because b is parallel to a, as b is a child of a. Clearly, one of the subtrees of b (left or right) does not need to be tested against a (in this case the right subtree). Therefore, even if nodes c and d intersect a we do not need to partition one against the other, as b is *shielding* one of them from intersecting a.

A more complicated example involves the case where some node (not a child) does not intersect a. Consider the case in Fig. 9 where b and e intersect a, but c does not intersect a. The insertion of af into the lower dimensional BSP-Tree associated with a requires first a partition against ab. Using as

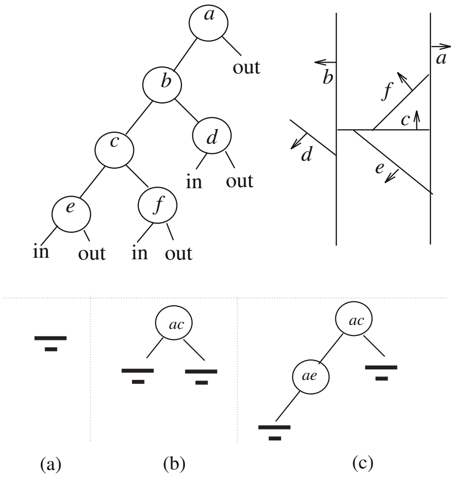

Fig. 8. First case of Partial Ordering not defined. The status of the lower dimensional BSP-Tree associated with a after the visit of nodes a, c and e is illustrated in (a), (b) and (c)

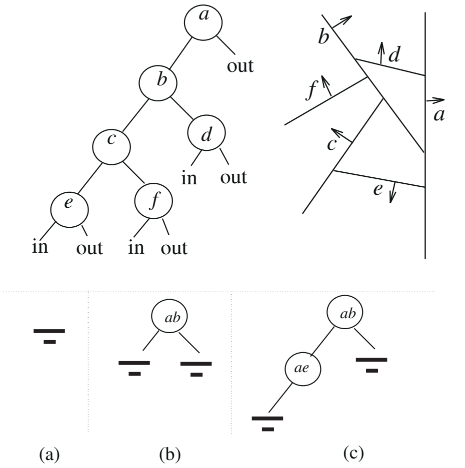

Fig. 9. Second case of Partial Ordering not defined. The status of the lower dimensional BSP-Tree associated with a after the visit of nodes a, b and e is illustrated in (a), (b) and (c)

above the partial order $f \prec_p b$ we propagate the result to the left subtree of ab. Now we need to partition af against ae, but the path partial ordering is not defined between e and f. Note that the fact that c does not intersect a tells us that one of its subtrees does not need to be tested for intersection against a, because as before c and its ancestors shield one of its subtrees from intersecting a. In this case, c and b shield f from intersecting a

Using such results, we may decide in which side to insert a given hyperplane when we perform the partitioning operation. We partition the hyperplane by computing the intersection with the partitioner, and propagate the result to the side consistent with the partial ordering defined by the current path. When a leaf node is reached, we evaluate the sub-hyperplane partitioned, and if it is not empty we create a new node with undefined left and right attributes, and update all the 1D-BSP-Trees of the most recent partitioner hyperplanes found along the way.

One difficulty in this incremental construction is that we only discover the attributes (*IN* or *OUT*) when we visit a leaf node. At this point, we need to propagate this attribute to all lower dimensional BSP-Trees in the current region path and create leaf nodes with the attribute just discovered. In fact this difficulty is a particular case of the insertion that we are performing in the other cases, only that in this case we are not inserting a hyperplane but an attribute, that can be localized in the tree using the partial ordering as above.

Another important operation that needs to be performed when computing intersections and inserting them into lower dimensional BSP-Trees is to keep track of the place where we insert the different copies of an intersection. For instance, the same intersection h_{12} of h_1 and h_2 needs to be inserted into the lower dimensional BSP-Tree associated with h_1 and with h_2. Topological pointers are created to connect the copies of this intersection, which will allow the reconstruction of the BRep in a next step. The procedure to compute the lower dimensional information is described in Fig. 10.

```
procedure ComputeLowerDimension(BSPTree *current, BSPTree *path)
  for each hyperplane hp in the current path
    if current hyperplane hc is a leaf
      // We update the lower dimensional trees with the attribute
      Insert(DimensionSon(hp, side(hp)), current.attribute, path);
    else
      Compute intersection hpc between hp and current hc
      if hpc is not empty
        // We insert the intersection into the lower BSP-Trees
        // of hc and hp and link them with topological links
        Insert(DimensionSon(hc, +), hpc, path);
        Insert(DimensionSon(hc, -), hpc, path);
        Insert(DimensionSon(hp, side(hp)), hpc, path);
      endif
    endif
  endfor
  ComputeLowerDimension(current.left, path ∪ current+)
  ComputeLowerDimension(current.left, path ∪ current-);
end ComputeLowerDimension;
```

Fig. 10. Procedure to Compute Lower Dimensional Information

3.4 Using the TBSP-Tree to reconstruct the Boundary

The incremental construction of the TBSP-tree built information about topological relations among the elements in all dimensions. A node in the tree has all the information necessary to reconstruct the boundary when both lower dimensional trees have been completely computed. In order to extract the boundary we perform a gluing operation in the dimension of the embedding space, which remove edges internal to faces and faces internal to solids.

In Fig. 11 we have an example where the partitions in both sides of the hyperplane h need to be glued together to remove internal elements. The gluing operation is responsible for combining the results from the two subtrees in such way that the result is a valid representation for the boundary defined in the subtrees of the node. In order to glue the information in the positive and negative halfspaces of a node, we perform the symmetric difference of the lower dimensional BSP-Trees in both sides of the hyperplane, which is a boolean operation that can be executed by a tree merging algorithm for BSP-Trees[7]). The result of the symmetric difference operation is a tree whose *IN* regions are elements of the boundary of the object.

Note that the gluing process to remove internal features entails a recursion on dimension that first glues lower dimensional features. In other words, in order to glue two faces we first glue the elements internal to each of these

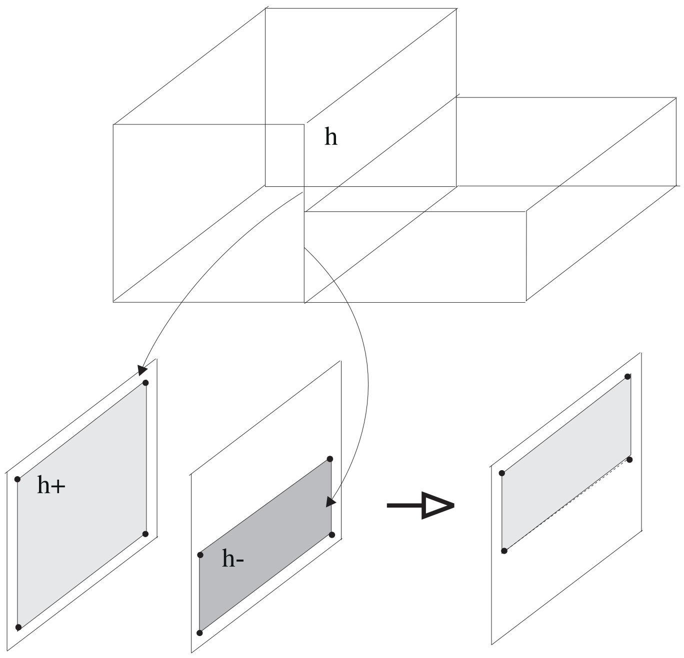

Fig. 11. Gluing opposite faces to obtain the boundary

faces separately, which will correspond to the removal of internal edges of a face. The algorithm initially performs two gluing operations in the positive and negative lower dimensional trees, and a symmetric difference operation that combines the results obtained. The algorithm for gluing TBSP-Trees is dimension-independent, and the pseudo-code is described in Fig. 12.

```
procedure GlueBSPTree(BSPTree node, Dimension dim)
  if dimension > 1
    tree1 = GlueBSPTree(DimensionSon(node,+), dim-1);
    tree2 = GlueBSPTree(DimensionSon(node,-), dim-1);
  endif
  return SymmetricDifference(tree1, tree2);
end GlueBSPTree;
```

Fig. 12. Procedure to glue TBSP-Trees

The gluing process can be implemented in such a way that the boundary relations of the different dimensioned elements is preversed. In Fig. 13 we illustrate the gluing operation for an example in 2D. The complicated cases arise in nodes f and c, where we have partitions induced by the BSP-Tree in both sides of the hyperplanes. In Fig. 13a we show the partition induced by the tree, and the information stored in the TBSP-Tree is illustrated by the cycles of edges and vertices in each cell. When performing the symmetric difference we not only remove internal features, but we also join the cycles in both sides of the hyperplane. In Fig. 13b we show the result of the application of the gluing operation to the node f. The corresponding features are identified by performing the symmetric difference operation for both lower dimensional trees in one dimension lower, and the cycles are joined together when we have mutually incident relations. In this case, the edge generated by the hyperplane f is removed, and the cycle of edges of both copies of f are joined together. In a similar way Fig. 13c shows the result of the gluing operation for node c.

It is important to demonstrate that the boundary information obtained is in fact minimal. This is achieved due to the fact that all internal features are removed by the recursion in dimension performed by the gluing algorithm. This is an important consequence, because the convex decomposition created by the BSP-Tree decomposition may generate a fragmentation of the boundary when non-convex objects are represented. The gluing operation, as proposed, provides an elegant solution to the problem of reconstructing the minimal BRep from a BSP-Tree representation of a polytope.

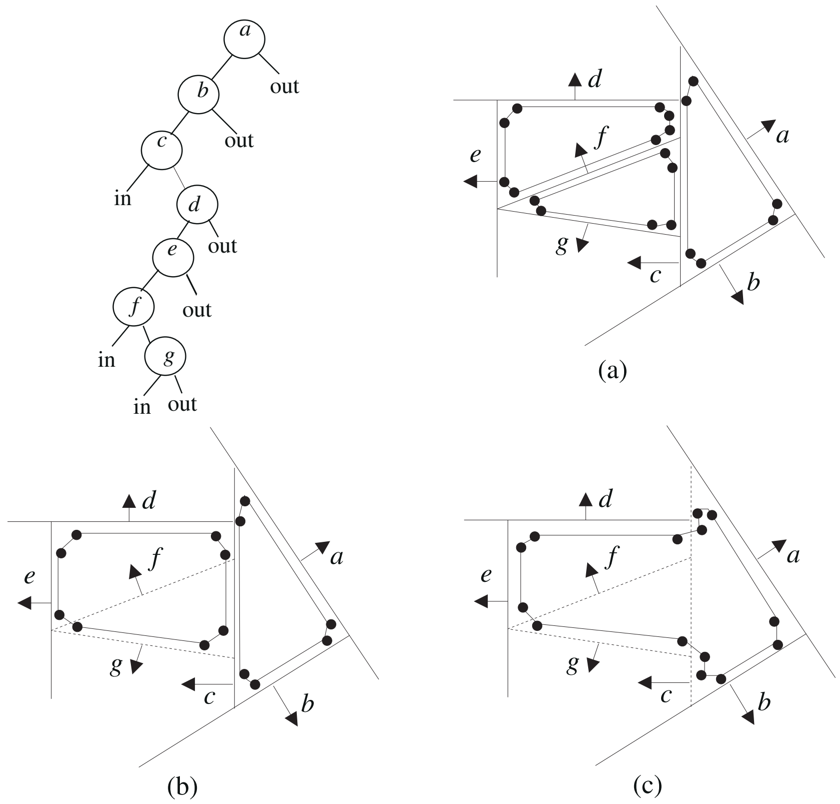

Fig. 13. Example of the gluing Procedure in 2D

4. Conclusions

In this paper, we have presented an algorithm to convert a BSP-Tree representing a polytope to a BRep representation. The storage of lower dimensional information in a TBSP-tree allowed us to reconstruct the information necessary to recover the boundary of the object. The TBSP-Tree is incrementally computed during a pre-order visit of the tree, which computes all lower dimension information in both sides of each node. A gluing step is performed when all information about a node is discovered. which involves a recursion in the dimension of the space to remove internal features and glue the boundary representation in each side of the node. As a consequence of this process, the boundary representation obtained at the end corresponds to a minimal BRep.

We believe that the importance of the BSP-Tree representation can be extended by having such algorithm available. BSP-Trees and BReps are both representation of polytopes used in Solid Modeling, each one with its own advantages. For example, BSP-Trees are more efficient where visibility orderings or boolean operations need to be computed, whereas in some other applications, like topological deformations, we would prefer to use the BRep. By having both conversion algorithms available we can exploit the advantages of each model more efficiently.

Another application where the conversion algorithm proposed in this paper can be used refers to the problem of finding a near-optimal BSP-Tree representation of a polytope. The BSP-Tree representation is not unique, and many different trees may represent a given polytope. The problem of finding a minimal tree is even more important when we perform successive boolean operations by applying tree merging algorithms, which may generate trees that need to be re-structured. Unlike Binary Search Trees, where we have algorithms to keep a tree balanced, the balancing of a tree in higher dimensions is a much more difficult problem. By applying the conversion described here we obtain a minimal BRep, which aggregates the geometric information expressed by the tree. The further conversion from BRep to BSP-Tree may generate a much more balanced tree, as the topological information of the BRep gives better heuristics in the construction of the tree.

Finally, the combination of topological information in the BSP-Tree, which resulted in the TBSP-Tree, shows promissing applications in Solid Modeling. In future work we plan to extend BRep algorithms, like topological deformations, to work directly with TBSP-Trees.

References

1. H. Fuchs, Z. M. Kedem, and B. F. Naylor. On visible surface generation by a priori tree structures. *Computer Graphics (SIGGRAPH '80 Proceedings)*, 14(3):124–133, July 1980.
2. Dan Gordon and Shuhong Chen. Front-to-back display of BSP trees. *IEEE Computer Graphics and Applications*, 11(5):79–85, September 1991.
3. M. Mantyla. *An Introduction to Solid Modeling*. Computer Science Press, Rockville, Md, 1988.
4. J. Matoušek. Efficient partition trees. *Discrete Comput. Geom.*, 8:315–334, 1992.
5. Bruce Naylor. Binary space partitioning trees as an alternative representation of polytopes. *Computer-Aided Design*, 22(4):250–252, May 1990.
6. Bruce Naylor. SCULPT an interactive solid modeling tool. In *Proceedings of Graphics Interface '90*, pages 138–148, May 1990.
7. Bruce Naylor, John Amanatides, and William Thibault. Merging BSP trees yields polyhedral set operations. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 115–124, August 1990.
8. Bruce F. Naylor. Partitioning tree image representation and generation from 3D geometric models. In *Proceedings of Graphics Interface '92*, pages 201–212, May 1992.
9. A. A. G. Requicha and H. B. Voelcker. Boolean operations in solid modeling: Boundary evaluation and merging algorithms. *Proc. IEEE*, 73(1):30–44, January 1985.

10. Jaroslaw R. Rossignac and Herbert B. Voelcker. Active zones in CSG for accelerating boundary evaluation, redundancy elimination, interference detection, and shading algorithms. *ACM Transactions on Graphics*, 8(1):51–87, 1989.
11. William C. Thibault and Bruce F. Naylor. Set operations on polyhedra using binary space partitioning trees. In Maureen C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 153–162, July 1987.
12. Enric Torres. Optimization of the binary space partition algorithm (BSP) for the visualization of dynamic scenes. In C. E. Vandoni and D. A. Duce, editors, *Eurographics '90*, pages 507–518. North-Holland, September 1990.
13. G. Vanecek, Jr. Brep-index: a multidimensional space partitioning tree. *Internat. J. Comput. Geom. Appl.*, 1(3):243–261, 1991.