EG UK Theory and Practice of Computer Graphics (2006)
M. McDerby, L. Lever (Editors)

Out of core Polyhedral Union and its Application to Interactive Shadow Rendering

J. Fedorkiw, C. Smith, and S. Ghali

Department of Computing Science
University of Alberta
Canada

Abstract

Four methods for storing a set of points using a computer are currently known: Boundary representations, Constructive Solid Geometry, Binary Space Partitioning trees, and Nef polyhedra. We describe a time and space-efficient BSP-based algorithm for computing the union of a set of solids and compare it with the other solid representations. The algorithm does not require that the entire tree fit in memory; it only needs to maintain the path from the root to one node in the tree at a time. We show that the algorithm is practical by providing time and space statistics. We also show the benefit of using the resulting union solid for computing interactive shadows.

1. Introduction

Modeling systems frequently need to store point sets using more than one representation, and so algorithms that convert from one representation to another are essential components of such systems. Four methods are currently known for storing solids: Boundary representations (b-reps), Constructive Solid Geometry (CSG) trees, Binary Space Partitioning (BSP) trees, and Nef polyhedra (ignoring spatial occupancy representations as they do not transform). The first two methods, b-reps and CSG, have been widely studied. A significant amount of work has recently been done on the last, Nef polyhedra, particularly in the context of the CGAL library [Bie95, See01, GHH*03]. The third method, solid modeling BSP trees, has generally been considered to be a special type of BSP trees as used for depth sorting computation, which has perhaps resulted in too little work done on the set of solids it captures [Nay90] and on its conversion to alternative representation schemes [CN96]. We describe in this paper an algorithm that converts a BSP tree representing a solid to its boundary representation without requiring that the BSP tree entirely fit in core memory at any one time. The most important application to-date of BSP trees to model solids is their use for Shadow Volume BSP trees [CF89]. To remain well-grounded in a practical application, we choose Shadow Volume BSP (SVBSP) trees as the application domain, but the details are applicable for an arbitrary BSP tree modeling a solid. Since shadow volumes arise individually for each solid in a scene, and since computing the boolean union operation is easy on a BSP tree [TN87], but difficult on a b-rep [Män88], it is advantageous to compute the union of shadow volumes, which results in significant speedups during runtime resulting from a reduction in shadow polygon fillrate [GS05].

BSP trees have the advantage over modeling with either CSG trees or Nef polyhedra of not requiring the computation of neighborhoods, which pose a significant implementation challenge. BSP trees have the mild constraint that they can only capture regular sets, but that is an acceptable, and sometimes desirable, constraint in many practical system.

Compared to CSG and Nef representations, a BSP tree reverses the roles of interior and leaf nodes. Whereas a Nef polyhedron can be considered a special type of CSG tree in which only (open) halfspaces are allowed at the leaves, in a BSP tree each interior node stores an oriented plane that partitions space into two sets corresponding to its two subtrees and each leaf node stores a boolean flag identifying whether the convex set represented at the leaf is included in the solid being modeled.

In what follows we describe SVBSP trees, how the boundary of the union of shadow volumes can be extracted, how the computation can be carried out while storing only one path from the root to a leaf, and how to handle robustness issues that arise during implementation. We conclude by providing time and space statistics.

2. Extracting the Boundary of a shadow volume BSP Tree

A shadow volume BSP Tree partitions a scene into shadowed and non-shadowed cells [CF89]. The scene is recursively partitioned by oriented planes that pass through the light source and silhouette edges. Figure 1 illustrates a scene and its SVBSP tree.

Each path from the root node to a leaf node in the SVBSP tree describes a unique cell that bounds points that are either entirely inside shadow or outside shadow. For example the path $Oa - Oh - Oi - Oj - Ok - in$ in Figure 1 describes the cell containing the points occluded by quadrangle $hijk$. This cell is defined by the intersection of the positive halfspace of Oa and the negative halfspaces of Oh, Oi, Oj and Ok. Each leaf node contains points that are either entirely *in* shadow or *out* of shadow, as defined by the node labels. Once the tree is constructed, it is possible to query the tree to determine whether a point lies in shadow, or to generate lists of illuminated and non illuminated polygons [CF89].

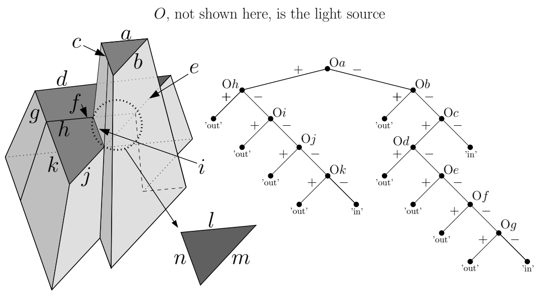

Figure 1: *Two triangles and the shadow volume they produce are shown on the left. The triangle closer to the light, abc, remains intact; the triangle below it is split into the two visible quadrangles $defg$ and $hijk$, as well as the occluded triangle lnm. The SVBSP tree is shown on the right.*

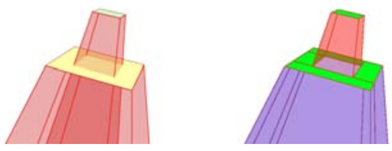

Figure 2: *In traditional shadow volumes, shown on the left, all polygons are quadrangles with two affine and two ideal (w=0) vertices. By contrast in the set union of the shadow volumes, shown on the right, some polygons are defined by four affine vertices.*

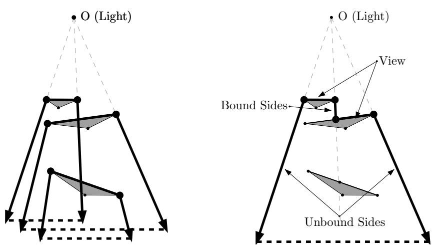

Figure 3: *Sections in the shadow volumes with and without the computation of the set union are shown.*

3. Out-of-Core Construction of BSP Tree

The new BSP tree consists of 3 node types: 'interior', 'in' and 'out' nodes. The 'interior' nodes are the nodes defining a splitter plane and the 'in' and 'out' nodes are leaf nodes indicating volumes in and out of shadow, respectively. The tree is constructed such that every interior node is guaranteed to have 2 child nodes.

Interior nodes store a splitting plane, the edge e defining the splitting plane, a flag indicating whether e is a silhouette edge, and a set of point intervals along e that identify the set of points that remain on the (topological) boundary of the set captured by the BSP tree. Leaf nodes labeled 'in' store the polygon visible from the observer and, optionally, a list of the polygons thus occluded.

Given a three dimensional scene consisting of convex polygons a BSP tree is constructed by processing the polygons in front to back order from the observer (using a secondary BSP tree). A polygon is incrementally inserted in the initially empty tree. At each 'interior' node the polygon is split; the up to two fragments that result are recursively inserted in child nodes. The polygon is thus fragmented into pieces that are either wholly lit or wholly in shadow. Each fragment that arrives at a leaf node labeled 'out' is illuminated and is used to extend the tree to describe the shadow volume produced by the fragment. The tree is initialized to one node labeled 'out'.

We show that it is not necessary to compute and store the entire tree before extracting the boundary of the solid defined by the BSP tree. It is sufficient to traverse the tree in a manner reminiscent of depth-first tree traversal. The solution can be extracted on the fly and the tree trimmed such that only a single path from the root node is stored at any given moment.

Because we keep only one path of the BSP tree in memory at any given moment the space is now bound by the depth of the tree, which is at worst linear in the number of edges in the scene (Figure 18). The *Percolate* method listed next performs the depth-first traversal and pruning of the BSP tree.

At each interior node the polygon fragment is first split by the splitter plane (*Percolate*, line 9). This operation is per-

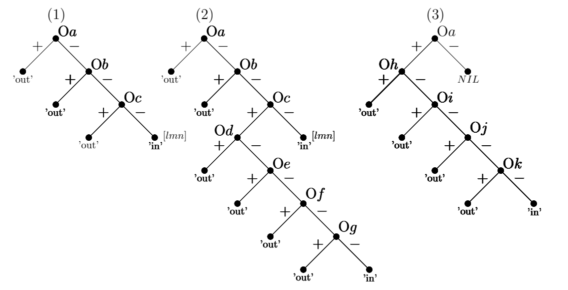

Figure 4: *Depth first creation of the BSP Tree. (1) The polygons are inserted into the tree, sorted in front to back order. The first polygon, abc, is visible so the method buildSubtree is called to construct a subtree describing its shadow volume. The rest of the polygons are first split by Oa, then Ob, then Oc with the resulting fragments filtering to the appropriate subtrees (while maintaining their relative order). (2) After processing the negative child of Oc, the positive child is processed. The quadrangle ghij is the first (and, in this case, only) polygon at a node labeled 'out', thus it is visible and again the buildSubtree method is called to replace the leaf node. This subtree is processed then pruned. (3) After processing and pruning Oa's negative subtree the positive subtree is processed, where triangle def creates another subtree.*

```
Percolate(list sortedPolys)
1. if NodeLabel == 'interior' OR NodeLabel == 'out'
2.     list PosPolygons, NegPolygons;
3.     if NodeLabel == 'out'
4.         BuildSubtree(sortedPolys.front())
5.         OutputVisiblePolygon(sortedPolys.front());
6.         sortedPolys.popFront();
7.
8.     For each Polygon in sortedPolys
9.         (PosPoly, NegPoly) = Split(currentPoly, SplittingPlane);
10.          if PosPoly != nil and NegPoly != nil
11.              Intercept(currentPolygon)
12.
13.          PosPolygons.append(PosPoly);
14.          NegPolygons.append(NegPoly);
15.
16.      OutputSideShadowVolumes();
17.
18.      if(positiveChild)
19.          positiveChild.percolate(PosPolygons);
20.          delete positiveChild;
21.      if(negativeChild)
22.          negativeChild.percolate(NegPolygons);
23.          delete negativeChild
24.
25. if NodeLabel == 'in' //Polygon is not visible from the light
26.      OutputInvisiblePolygons(sortedPolys);
```

formed by the *Split* method, which in addition to producing the two fragments on either side of the splitter plane, must also ensure that the fragments have appropriate silhouette edge status. See Figure 5.

```
split(Polygon P)
1. PositiveFragment = fragment of P on the positive side of the splitter
     plane, nil if no such fragment exists
2. NegativeFragment = fragment of P on the negative side of the splitter
     plane, nil if no such fragment exists
3. If P is co-planar with the splitter plane
4.     NegativeFragment = P
5. Update Silhouette status of edges in each fragment so they match status of edges in P.
New edges get a non-silhouette status as shown in Figure 5.
6. return(PositiveFragment, NegativeFragment)
```

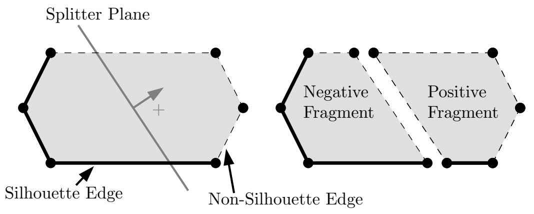

Figure 5: *Splitting a polygon by a splitter plane.*

Whenever a polygon fragment passes through an interior node containing a silhouette edge, the intercept method is called (*Percolate*, line 11). The *Intercept* method takes the polygon being processed and uses it to update the interval set for the node, such that it accurately reflects the segments along *e* that produce bound shadow polygons and those that produce unbound shadow polygons. The operation of this function is illustrated in Figure 6. The one-dimensional set operations ($-$ and $\cap$) must be implemented as regularized operations, which in practice simply means that no interval can have coincident extremities. Each node in the tree stores an interval I initialized to the two endpoints of the casting segment. In Figure 6, for example, I is initialized to $[a,b]$. The initially empty set S incrementally stores the shadow polygons, such as polygon $(L1,s,t,L2)$ in Figure 6.

```
Intercept(Polygon P)
1. Edge $E$ = Intersection(P, SplitterSegment);
2. if $E$ is a silhouette edge of P
3.     return;
4.
5. Point I1 = Projection of $E$.source through O onto $e$
6. Point I2 = Projection of $E$.target through O onto $e$
7.
8. if O is between I1 and $E$.source
9.     I1 = Intersection(P, SplitterSegment)
10. if O is between I2 and $E$.target
11.     I2 = Intersection(P, SplitterSegment)
12.
13. Interval $\beta$ = [Min(I1, I2), Max(I1,I2)]
14. Clamp $\beta$ to be within $e$'s endpoints
15.
16. $I = I - \beta$
17. $K = I \cap \beta$
18.
19. For each Interval [P1, P2] in $K$
20.     P3 = Intersection(Line(O, P1), P);
21.     P4 = Intersection(Line(O, P2), P);
22.
23.     S.append(Polygon(P1,P3,P4,P2);
```

It should be noted that on line 13 of the *Intercept* method it is indeed possible to compute the minimum and maximum values of I1 and I2. Since both values lie on the line *e* they can be compared using the one dimensional operations greater than and less than. To execute these operations we consider the dominant component of the three defining the points: x, y and z, the component of largest (absolute) line slope.

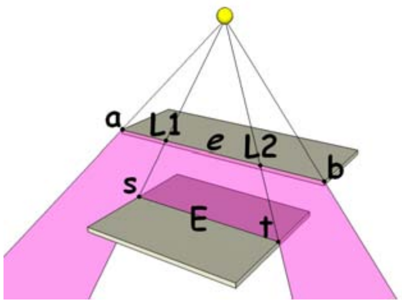

Figure 6: *When the intercept method is called for edge e the interval set is* $\{[a,b]\}$*. After E's endpoints, s and t, are projected onto e the interval set becomes* $\{[a,L1],[L2,b]\}$

Finally, each time a polygon fragment reaches a node labeled 'out', it is visible from the observer, which means a subtree describing its shadow volume must be built (*Percolate*, line 4). This is accomplished by using the *BuildSubtree* method:

```
BuildSubtree(Polygon P)
Each edge in P is combined with the light source to construct a new set of
splitting planes. These splitting planes are used to build a new subtree, the
root of which (now labeled 'interior') replaces the current node labeled 'out'.
```

To extract the boundary of the union of shadow volumes we require three pieces of information from the tree: the view, the bounded side polygons, and the unbounded side polygons, as shown in Figures 2 and 3. The view forms both the near and far caps (by projecting the view to infinity) of the shadow volume, while the bounded and unbounded polygons provide the sides of the shadow volume.

The interval set at each node describes the set of points P on *e* for which the ray OP does not intersect any geometry in the scene, thus each interval in the interval set defines a segment on *e* from which an unbound shadow polygon emerges. The unbound shadow polygon is described by four points: the two endpoints of the interval, and two corresponding ideal points. Each ideal point is the intersection of the corresponding ray OP with the ideal sphere.

4. Robustness Issues

4.1. Robust Projection

One particular intricacy of this algorithm is the projection performed to update *e*'s interval in the intercept method. The complication is illustrated in Figure 7.

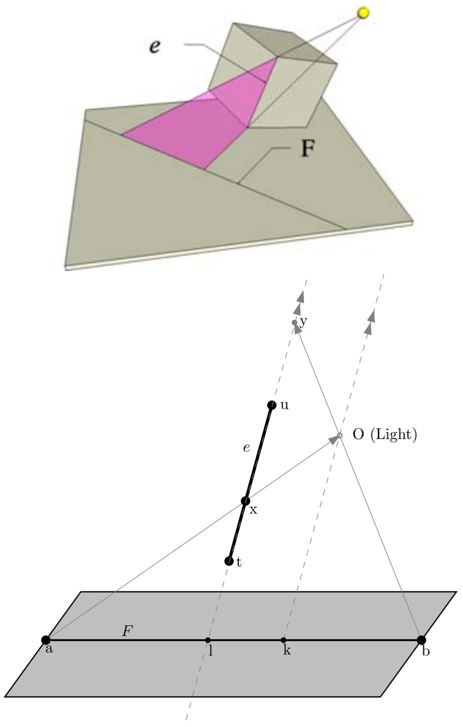

Figure 7: *The interval resulting from the projection (after clamping to e's endpoints) of a and b onto e through O is* $[x,u]$*, which incorrectly predicts that for any point P in the interval* $[x,u]$ *the ray OP intersects* F*. The correct interval is instead* $[x,t]$*.*

The intercept method computes the edge F, the intersection of *e*'s splitter plane and the polygon, and projects it back onto *e* to compute an interval. This interval describes the set of points P on *e* for which the ray OP intersects F; the interval describes the segment on *e* from which a shadow volume emerges and is then intercepted by F (see Figure 7). But the projection has to be implemented carefully.

If k is the intersection of the line passing by O and parallel to *e* with the polygon, and if one of F's endpoints, b, lies to the right of k, the ray Oy will not intersect F – even though when we perform the backward projection we get the point y on *e*. Effectively the point k projects to the ideal point on *e* and the edge ab does not map to the affine edge xy, but to its complement, which includes the ideal point. A simple solution, once this case is detected, is to clip the projection of any endpoint to the right of k to the point t on *e*. If the endpoints of F straddle k, this will give the desired outcome of ignoring points to the right of k. If both points project to t, the resulting interval would be a point, which subsequently vanishes anyway since we use regularized set operations.

This solution is implemented on lines 8—11 of the *Intercept* method. If O lies between a point (b) and its projection

(y), we use l, the point of intersection of e and the polygon as the projection point. This intersection point is guaranteed to be outside the line segment e, thus simulating the ideal point. If e passes through the polygon containing F in the original scene, e would be split by P in the secondary BSP tree. So at worst e intersects P at e's endpoint. Also, because the interval is clamped to $[t,u]$, the intersection l will be correctly clamped to t.

In effect this is a classical projective geometry problem that is well-known in computer graphics. If we project a segment through a center of perspectivity O onto a plane π, but the endpoints of the segment lie on opposite sides of the plane passing by O and parallel to π, it would be incorrect to assume that the projection is the affine segment connecting the projections of the endpoints. The projection is instead the "other" segment joining the two points, which includes one ideal point. This is the reason it is necessary to clip to a view volume before performing the projection, which is then guaranteed to consist entirely of affine points.

4.2. Robust Rendering

A second problem arises when rendering shadows using the resulting shadow volumes whereby some minor cracks appear as dots of shadow or of light between shadow polygons. Figure 8 illustrates the problem, which is, once again, a classical problem in computer graphics: T-vertices. Since adjacent shadow polygons are rasterized independently, their common edge will not always be pixel-exact. T-vertices can be eliminated using two methods depending whether they are infinite or finite shadow polygons. For infinite shadow polygons one does not cast the segment provided to infinity, but builds a segment from each neighbor's finite points (or just the segments point if one neighbor is infinite) and casts this segment to infinity, which guarantees that the points lie on the same line. In the case of finite polygons, one simply triangulates the side mesh such that the problem no longer arises.

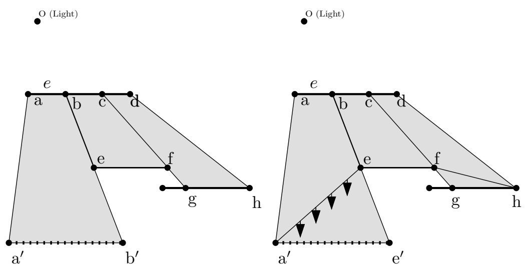

Figure 8: *The two adjacent polygons a-b-b′-a′ and b-c-f-e may have a crack between them. This results because when e is scan-converted it may no longer be on the line formed by b and b′. Similarly f may not be on the line formed by c and g.*

5. Implementation

The algorithm has been implemented in C++ with the help of the Computational Geometry Algorithms Library. CGAL made it possible to experiment with various number types. Using floating point arithmetic makes it possible to process only small scenes. Numerical robustness issues quickly arise when processing other than toy scenes and it is difficult to solve numerical issues by using epsilons, a popular technique in computer graphics. We have opted instead to use the certain and more elegant, but also more expensive, combination of exact number types (rational numbers) with floating point filters. Only if a floating point operation is unreliable are rational numbers used to evaluate the expression. As one would expect, maintaining rational numbers defined with varying-length integers consumes significant space, which was the original motivation for seeking a modification to BSP trees that does not require the storage of the entire tree in memory at the same time.

In addition to implementing the BSP tree we also implemented an application that reads the boundary shadow volume produced and that renders the scene with shadows. The results observed are explored in the next section.

6. Regular vs. Concise Shadow Volumes

The recently popular stencil shadow volumes render pixel-perfect shadows and also produce accurate shadows on dynamic objects. In traditional shadow volume rendering [Cro77] each object in the world casts shadow volumes.

Rendering shadows using shadow volumes involves three rendering passes: The scene is first rendered using ambient lighting, then a mask that identifies shadowed pixels is created using the stencil buffer. Finally, pixels not masked by the stencil buffer are re-rendered using full lighting.

To produce the appropriate shadow mask in the stencil buffer the shadow volumes that fail the depth test are rendered into the stencil buffer. Back-facing shadow polygons increment the pixel value, while front-facing shadow polygons decrement the value. The number at each pixel indicates how many shadow volumes the pixel is in. If the value is zero, the pixel is in no shadow volumes, and if it is one or more, it lies in shadow. See Figure 9.

By using the concise shadow volumes to produce shadows for a scene we minimize the surface area of the shadow volumes that need to be rendered. As shown in Figure 10 this results in significant improvements in frame-rates for complex scenes in which shadow volumes interact. In other scenes where the shadow volume exhibits little interaction, using concise shadows would be unnecessary (Figure 11).

7. Analysis

We empirically analyzed the time and space requirements for constructing the BSP tree on two scenes of various complex-

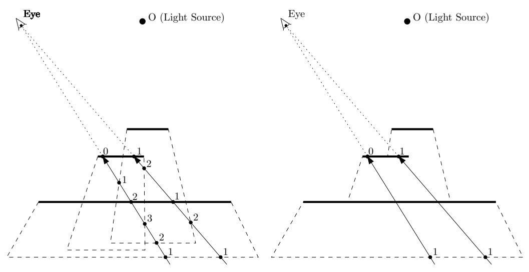

Figure 9: *Rendering shadow polygons into the stencil buffer is tantamount to casting a ray from infinity through each pixel and the viewer. As the ray passes through back and front facing shadow polygons (that fail the Z-test) the value for that pixel is incremented or decremented. The concise shadow volumes (right) generate fewer shadow polygons than the traditional shadow volumes (left).*

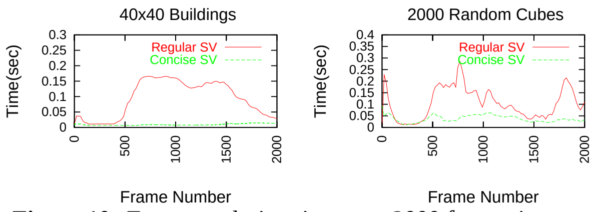

Figure 10: *Frame rendering time over 2000 frames is compared for the Buildings-40 and Random-1000 scenes using a fixed camera path.*

ity. The first scene consists of two layers of *n* parallel scaled cubes as shown in Figure 12. The second scene consists of *n* random cubes as shown in Figure 14. The time diagrams shown below are determined under filtering and the Gmpq number type with CGAL.

It is interesting to know the effect of storing a single path in the tree on the time and so we compared the time for constructing both the full and the pruned trees. As can be seen by comparing Figures 15 and 16, pruning the tree in a depth-first manner has no significant impact on the runtime.

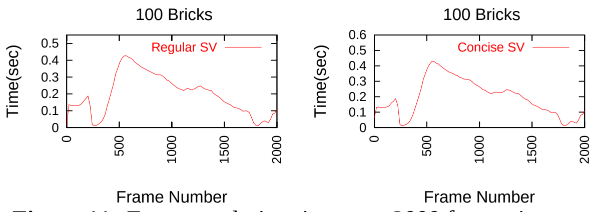

Figure 11: *Frame rendering time over 2000 frames is compared for the Bricks-100 scene. The two graphs have been separated because the rendering times are almost identical.*

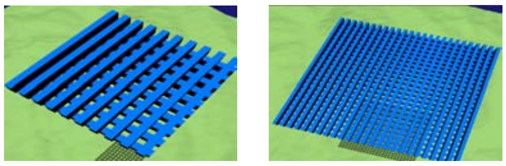

Figure 12: *The "Bricks" scene.*

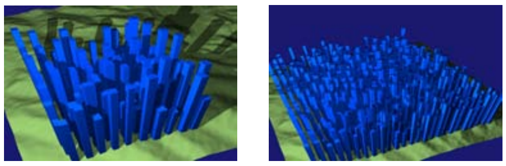

Figure 13: *The "Buildings" scene.*

7.1. Comparison with Nef Polyhedra

An alternative method for computing the boundary of shadow volumes is possible. CGAL offers an implementation of Nef Polyhedra that performs boolean operations between two Nef polyhedra. By constructing a Nef polyhedron for each shadow volume the boundary of shadow volumes can be extracted.

The construction of the boundary of shadow volumes using Nef polyhedra was timed for the Bricks and Random scene and the results are shown in Figure 19. Figure 20 compares the use of Nef polyhedra and BSP trees, which suggests the superiority of the latter.

8. Conclusion and Future Work

We have described a time and space efficient algorithm that generates the boundary of a BSP tree without computing the boundary of individual leaf nodes of the tree. The particular set of BSP trees we capture is the set of solids representing shadow volumes resulting from illuminating a scene from a point source. Many intriguing questions remain open. The visualization of a Nef or of a BSP solid without computing their boundary are intriguing questions.

Traditional thinking had suggested that the visualization of CSG trees can only be performed after extracting their boundary, but recent work has shown that the visualization is effectively independent of boundary extraction [HR05].

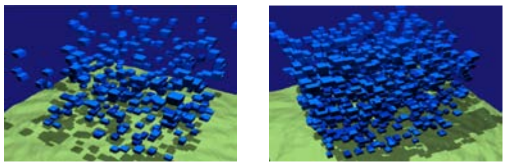

Figure 14: *The "Random" scene.*

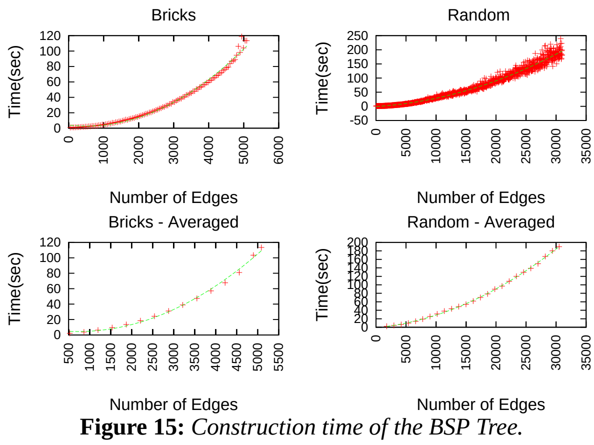

Figure 15: *Construction time of the BSP Tree.*

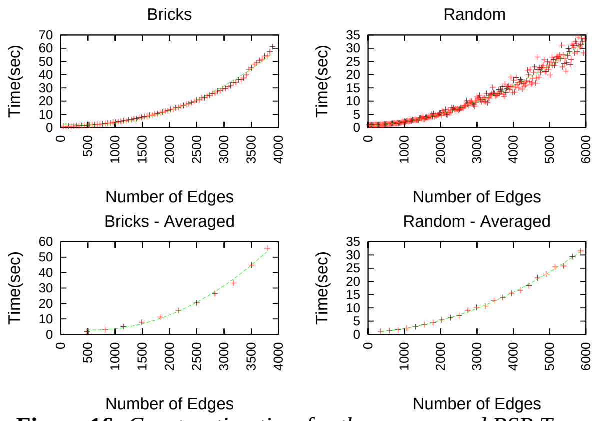

Figure 16: *Construction time for the non-pruned BSP Tree.*

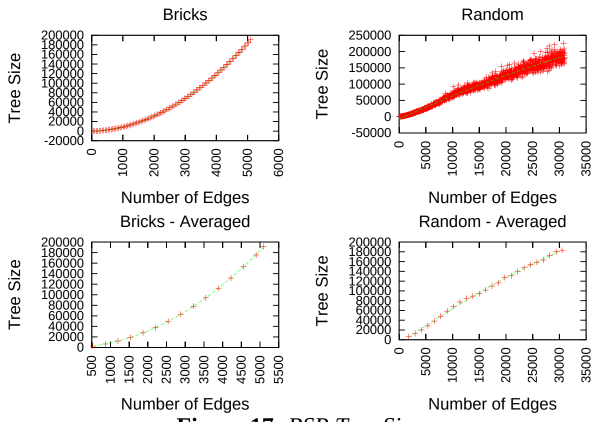

Figure 17: *BSP Tree Size.*

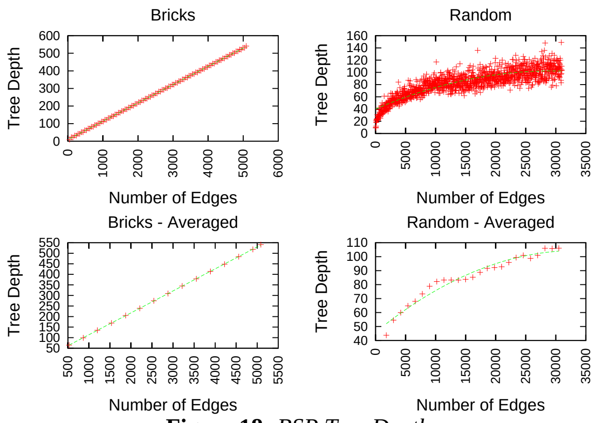

Figure 18: *BSP Tree Depth.*

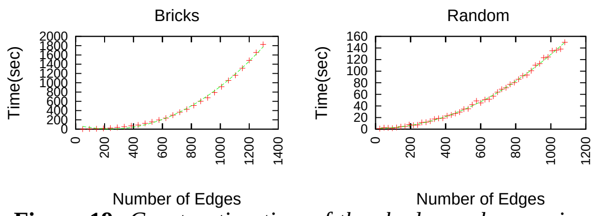

Figure 19: *Construction time of the shadow volume union using Nef polyhedra.*

At this time it is not known whether Nef or BSP representations can be visualized without computing the boundary. Such visualization of a BSP tree could potentially mean that shadow volumes can be determined using the stencil buffer without explicitly knowing the boundary of the solid.

Even though BSP trees are one of the more elegant methods for representing solids and the only one that makes the implementation of boolean operations a relatively simple task, they have not been widely adopted, which is perhaps due to a lack of a general theory of BSP solids. When implementing BSP algorithms, it is necessary to be careful about predicates such as the sidedness of a point with respect to a

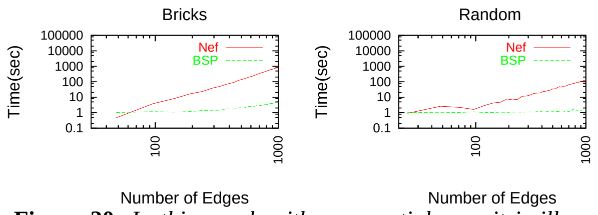

Figure 20: *In this graph with exponential axes it is illustrated that the time complexity of constructing shadow volumes using Nef polyhedra grows significantly faster than that of the BSP tree.*

partitioning plane, but similar robustness issues arise in any solid modeling algorithm. Two questions need to answered for the development of such a BSP theory.

The first problem is how one can capture the set of points lying on the partitioning hyperplane. An intuitive idea is to store a BSP tree of dimension $n-1$ at every interior node of an n-dimensional BSP tree, but the details have so far not been worked out. The second problem is the computation of the boundary of an arbitrary BSP tree. It has been shown that the boundary of individual leaves can be coallesced to produce the final result [CN96], but an algorithm for this problem is useful only if, crucially, the boundary is computed without first computing the boundary of individual leaves. That algorithm has also not been so far shown to be simple enough to be implementable.

The method presented to compute the boundary of a BSP tree requires that the tree in question represent a shadow volume. It remains to be investigated whether this method can be elegantly extended to compute the boundary of an arbitrary solid represented by a BSP tree. To extract the boundary of a BSP representation of a shadow volume we require both the BSP tree representing the solid as well as a 1D BSP tree capturing visibility along each splitter edge. To extend this method to arbitrary solids, we would require a 2D BSP tree in addition to the 1D BSP tree. For each splitting plane in the 3D BSP tree a 2D BSP tree would describe the facets of the solids on that plane. Each line in the 2D BSP tree would also require a 1D BSP tree.

Another intriguing problem does not arise in our case because all partitioning planes are defined by points in the input. But suppose that we are given an arbitrary line in 2D or plane in 3D defined by second or third generation points (a line whose extremities are themselves given by line intersection). How can one reduce the number of bits required for defining the line or the plane by finding points incident to the line or the plane, but ones that have a brief description?

References

[Bie95] BIERI H.-P.: Nef polyhedra: A brief introduction. In *Geometric modelling, Dagstuhl, Germany 1993* (Wien / New York, 1995), Hagen H. H., Farin G. E., Noltemeier H.„ Albrecht R., (Eds.), vol. 10 of *Computing. Supplementum*, Springer, pp. 43–60.

[CF89] CHIN N., FEINER S.: Near real–time shadow generation using BSP trees. In *SIGGRAPH '89* (Aug. 1989), pp. 99–106.

[CN96] COMBA J., NAYLOR B.: Conversion of binary space partitioning trees to boundary representation. In *Proceedings of Theory and Practice of Geometric Modeling* (Tuebingen, Germany, Oct 1996).

[Cro77] CROW F. C.: Shadow algorithms for computer graphics. *Computer Graphics 11*, 2 (1977), 242–248.

[GHH*03] GRANADOS M., HACHENBERGER P., HERT S., KETTNER L., MEHLHORN K., SEEL M.: Boolean operations on 3D selective nef complexes: Data structures, algorithms, and implementations. In *European Symposium on Algorithms* (2003), pp. 654–66.

[GS05] GHALI S., SMITH C.: Computing the boundary of a class of labeled-leaf BSP solids. In *17th Canadian Conference on Computational Geometry* (2005), pp. 159–162.

[HR05] HABLE J., ROSSIGNAC J.: Blister: GPU-based rendering of boolean combinations of free-form triangulated shapes. *ACM Trans. Graph. 24*, 3 (2005), 1024–1031.

[Män88] MÄNTYLÄ M.: *An Introduction to Solid Modeling*. Computer Science Press, Rockville, MD, 1988.

[Nay90] NAYLOR B.: Binary space partitioning trees as an alternative representation of polytopes. *Computer–Aided Design 22*, 4 (1990), 250–252.

[See01] SEEL M.: *Implementation of Planar Nef Polyhedra*. Research Report MPI-I-2001-1-003, Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany, August 2001.

[TN87] THIBAULT W. C., NAYLOR B. F.: Set operations on polyhedra using binary space partitioning trees. *Computer Graphics 21*, 4 (1987), 153–162.