A Tutorial on Binary Space Partitioning Trees (part of EG tutorial on visibility*)

Yiorgos L. Chrysanthou

Department of Computer Science,
UCL University of London,
Gower Street, London WC1E 6BT, UK.

The Binary Space Partitioning (BSP) tree algorithm was initially developed as an efficient method for ordering a set of polygons in order to solve the visible surface determination problem [12]. Based on Schumacher's work [24] on ordering linearly separable static sets, it uses an initial pre-processing step to give a linear display algorithm. It was later shown that BSP trees can be used to represent polyhedra and perform set operations upon them [25, 19]. Currently they are widely used for applications ranging from motion planning [26] to image representation [22] and shadow generation [6]. In this short tutorial we will just concentrate on their use for solving visibility problems. We start with a description of what is a BSP tree, how to build one from a set of polygons and how to use it for visibility ordering, for example to use with the painters algorithm. In *Section 4*, we describe how two BSP trees can be merged into one. Tree merging in an ingenious and elegant technique which can be put to a variety of uses such as view volume culling, described in *Section 5*, and occlusion culling described in *Section 6*. *Section 7* touches on the issue of building good trees, while in *Section 8* we give a short review of proposed solutions to the problem of using BSP trees with non-static environments. Finally as an appendix we attach a recent paper by Bittner et al that proposes a different way of doing occlusion culling again using BSP trees.

1 Definitions

The BSP tree is a hierarchical subdivision of n-dimensional space into homogeneous regions, using (n-1)-dimensional hyperplanes.

A hyperplane h (line in 2-D and plane in 3-D) is defined by an expression of the form $f_n = a_1x_1 + a_2x_2 + a_3x_3 + \cdots + a_nx_n + a_{n+1}$. The set of points in space

*An updated version of this document can be found on the authors web page http://www.cs.ucl.ac.uk/staff/Y.Chrysanthou/

that make $f_n > 0$ define the front (or positive) half-space of h (h^+), while those that make $f_n < 0$ define the back (or negative) half-space of h (h^-). The points $f_n = 0$ are on the hyperplane.

The BSP tree is stored in a binary tree structure. Each node t on the tree corresponds to a region in space (subspace), denoted by $r(t)$. Each internal node holds a hyperplane h_t that partitions the region at that node into front and back subspaces. The two subspaces are represented by the left ($t.front$) and right ($t.back$) "children" of the node. A leaf node, in contrast to an internal node, holds no hyperplane and corresponds to an unpartitioned region of space, which is called a *cell*.

The root node of a tree corresponds to the whole of the n-dimensional space. The region ($r(t)$) of each other node is defined by the intersection of the open half-spaces determined by the hyperplanes associated with the previous nodes on the path from the root to the node t. To be more precise, given a node t_i at depth i which lies along the path $\{t_0,...,t_i\}$, where the subscripts denote depth, if $i = 0$ then $r(t_0) = \Re^n$, otherwise if t_i is the front child of t_{i-1} then $r(t_i) = h^+_{t_{i-1}} \cap r(t_{i-1})$ else $r(t_i) = h^-_{t_{i-1}} \cap r(t_{i-1})$.

The intersection of the hyperplane h_t of a node t with its region $r(t)$, is called the sub-hyperplane ($shp(t)$). Notice that since $r(t_0)$ is unbounded, so is $shp(t_0)$. Also any other $r(t_i)$ and $shp(t_i)$ may only be partly bounded.

In most practical applications we use BSP trees in 2-D and 3-D spaces. The following description on building and using the tree for various tasks is given for 3-D environments (using polygons and planes) but as the concepts behind the BSP trees are dimension independent the same ideas are also valid in any other dimantion. (Note however that for simplicity in the figures, polygons are represented by line segments and planes by lines.)

2 Building a Tree

Given a set of polygons $S = \{s_1, .., s_n\}$, we can construct a BSP tree and partition 3-D space using the following simple recursive algorithm. A polygon is selected from S; the plane defined by this polygon is used to make the root of the tree and to partition space into front and back half-spaces. A reference to the polygon is also stored on the node. The rest of the polygons are compared against this plane and depending on which side they lie they are placed into two sets, *front* and *back*.

Any polygon lying partly in both subspaces is split along the intersection with the plane and the two fragments are placed in the corresponding sets. Any polygon found to be coplanar with the root plane is stored at the root along with the root polygon, Figure 1.

This procedure is repeated again recursively. A polygon is selected from each

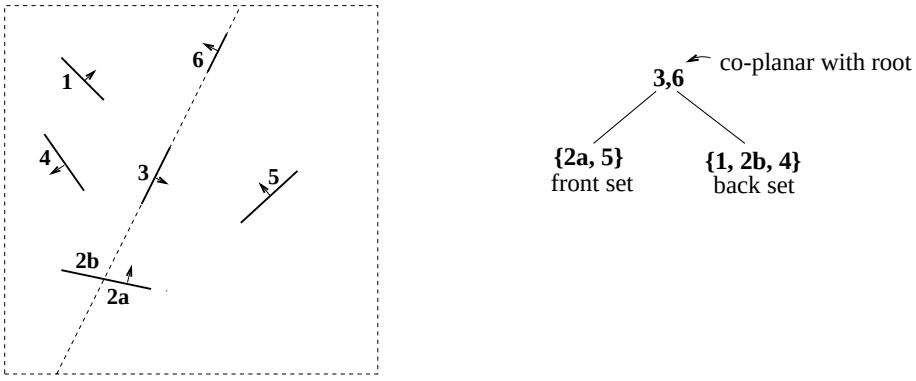

Figure 1: Partitioning space and the polygons with a polygon-plane

of the two sets and its plane forms the root of the corresponding subtree that further subdivides space and the rest of the polygons. This is repeated until all polygons have been used and the original space has been subdivided into homogeneous regions (cells). In Figure 2 the cells are labelled (a to f) and shown as leaves on the tree. In general we will not show the cells on the tree and we may refer to the last internal nodes as leaves. Whether a 'leaf' is a cell or the last internal node splitting into empty subspaces, will always be clear by the context.

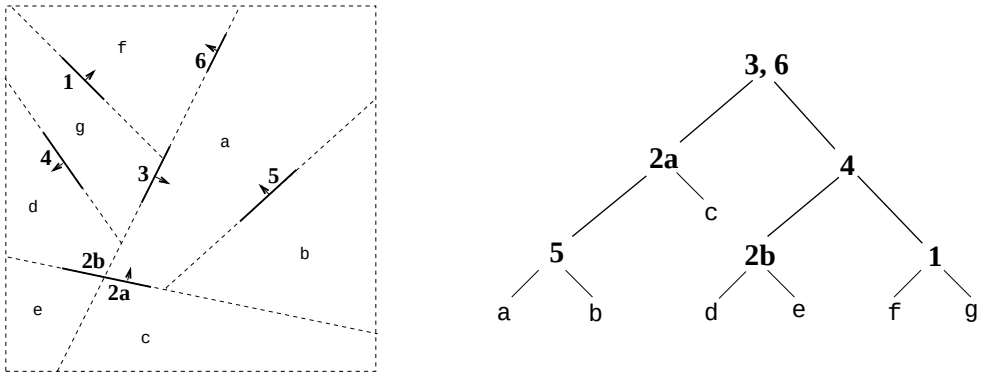

Figure 2: A complete subdivision

An alternative (incremental) approach for building a BSP tree from a set of polygons was suggested by Thibault and Naylor [25]. Each polygon is inserted in turn into the, initially empty, tree until it reaches a cell. The plane of the polygon then defines a node that splits the cell in two. Inserting a polygon into the tree is again a recursive procedure that compares the polygon against the root plane and sends it into the appropriate subtree. As before polygons might be split by the root plane or stored at the root if they are coplanar with it.

Both ways of building the tree have the same complexity. The advantage of the latter is that it can be used to add polygons to the tree after it has been built. This is useful for performing incremental changes to the tree (see [10]).

Selecting the appropriate root polygon at each iteration can be crucial for the efficiency of the resulting tree. Some efficiency issues are discussed in *Section 7.*

3 Visible Surface Determination

Given a BSP tree such as the one built above in Figures 1 and 2, we can use it to solve the visible surface determination problem by traversing it from any given viewpoint to get the back-to-front order of the polygons stored at the nodes. The polygons can then be displayed in that order using over-painting to cover hidden surfaces.

```
void displayTree(Tree node, 3DPoint viewpoint)
{
        if (viewpoint in-front of node plane)
              displayTree(back subtree, viewpoint);
              draw polygons on the node;
              displayTree(front subtree, viewpoint);
        else
              displayTree(front subtree, viewpoint);
              draw polygons on the node;
              displayTree(back subtree, viewpoint);
        endif
}
```

Figure 3: Traversing the tree to get a back-to-front order

The traversal is based on the fact that given a viewpoint and two sets of polygons separated by a plane, the polygons on the same (*near*) side as the viewpoint can obstruct but cannot be obstructed by polygons on the other (*far*) side. So to get the reverse, back-to-front, order from the tree the simple recursive algorithm of Figure 3 can be used: compare the viewpoint against the root plane, traverse the far subtree first then display the root polygon(s) and then traverse near the subtree.

When the shading of the polygons is defined by a complex function, the multiple over-painting of each pixel performed by the above algorithm could be costly. This can be avoided by using the scan-line algorithm suggested by Gordon [13]. The tree in this method is traversed in front-to-back order. As the polygons are displayed the filled segments at each scan-line are recorded. These segments are never overwritten and the algorithm terminates when the whole screen has been covered, or when the tree is fully traversed. Similar techniques are sometimes used in computer games [1].

4 Merging BSP Trees

The initial motivation of tree merging was for combining polyhedra in applications such as constructive solid geometry (CSG). In CSG this is usually done by means of boolean set operations (*union, intersection and difference*) to form more complex objects. The description of the polyhedra and objects is usually by boundary representations (B-reps). The use of this representation has many drawbacks such as being able to deal only with closed sets and requiring different

data structures or complex algorithms to perform the different operations (spatial search, model modifications, rendering).

Naylor and Thibault [25, 19] presented an alternative way of representing and combining polyhedra that is simple and efficient and also allows for open sets.

Thibault described how a BSP tree can be used to represent any arbitrary polyhedron [25]. This is done by giving an IN or OUT value to each leaf node depending on whether the corresponding cell is inside or outside the polyhedron. A simple example can be seen in Figure 4. The grey area corresponds to the inside of the polyhedron which is denoted by the IN cells of the tree on the right.

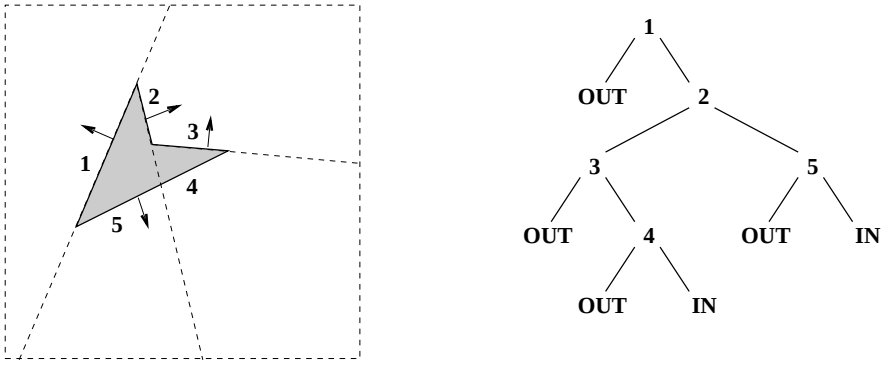

Figure 4: Representing a polyhedron by a BSP tree

Given two polyhedra P_1 and P_2 represented as BSP trees, T_1 and T_2, any boolean set operation can be performed on them by merging their trees [19]. Merging the partitionings of space, induced by T_1 and T_2 produces a third partitioning, T_3, that includes the two first. The values of the cells of the new partitioning depend on the operation used. The merging process however is always the same.

Merging T_1 and T_2 can be seen as inserting T_2 into T_1. In principle this is similar to the way we inserted the polygons in the tree during the incremental construction: starting at the root of T_1, T_2 is inserted recursively into T_1 until it reaches the leaves (cells) of T_1. At each step of the recursion, T_2 is compared against the plane h_{t_1} of each node t_1 of T_1 and is split into T_2^+ and T_2^-, where T_2^+ is the intersection of T_2 with the front half-space of h_{t_1} and T_2^- with the back half-space. Then T_2^+ is inserted in $t_1.front$ and T_2^- in $t_1.back$. Once it reaches a cell, an external routine is called that combines T_2 and the cell.

The pseudo-code for the merging is given in Figure 5. Comparing and splitting the tree by the plane of a node is performed by the function *partitionTree* which we will explain shortly. Combining a tree and a cell is done by *treeOpCell*. This depends on the set operation being performed. In general this routine will return either the cell or the tree or the complement of the tree (denoted by ∼t). The complement of the tree is found by reversing the attributes of its leaves, IN becomes OUT and OUT becomes IN. In this thesis only the *union* operation is used but the full table of the resulting values for all operations is given in Table

```
Tree merge(Tree t1, Tree t2)
{
      if (leaf(t1) or leaf(t2))
            return treeOpCell(t1, t2);
      endif

      {t2+, t2-} = partitionTree(t2, shp(t1));
      t1.front = merge(t1.front, t2+);
      t1.back = merge(t1.back, t2-);
      return t1;
}
```

Figure 5: Merging two trees

op	Cell	Tree	Cell <op> Tree
$\bigcap$	IN	t	IN
	OUT	t	t
$\bigcup$	IN	t	t
	OUT	t	OUT
—	IN	t	∼t
	OUT	t	OUT

Table 2.1: Combining a cell and a tree

2.1.

If there are other attributes involved such as colour, texture etc, then these also have to be merged. How this is done depends on the application. An example of trees augmented with additional values at the cells can be seen in [9]. There each cell holds a list of the polygons that occlude it from the source. In the *treeOpCell* function when the tree is retained, the list of occluders from the cell is added to the cells of the tree.

4.1 Partitioning a Tree with a Plane

As T_2 is inserted into T_1, at each node t_1 it is partitioned by the plane h_{t_1}, into T_2^+ and T_2^-. This partitioning is a recursive procedure that involves inserting h_{t_1} into T_2. Here again we differentiate between leaf nodes and internal nodes. When inserting h_{t_1} into T_2, if T_2 is a cell then T_2^+ and T_2^- are just copies of that cell. If not then three steps are performed:

1. h_{t_1} is compared against h_{T_2} to find their relative positions,
2. the subtrees of T_2 in which h_{t_1} lies are partitioned,
3. the resulting subtrees of the above partition are combined to form T_2^+ and T_2^-.

For step 1 the important thing to notice is that h_{t_1} and h_{T_2} are not defined over the whole of 3-D space but rather in the subspace formed by the intersection of $r(t_1)$ and $r(T_2)$. So the relation of h_{t_1} and h_{T_2} that we need is with respect to this region. Since h_{t_1} and h_{T_2} are infinite planes we need to find their intersections with the region in consideration. This intersection is what we earlier called the sub-hyperplane (*shp(t)*). The sub-hyperplanes are represented as polygons and their relative position is determined by comparing one against the plane of the other.

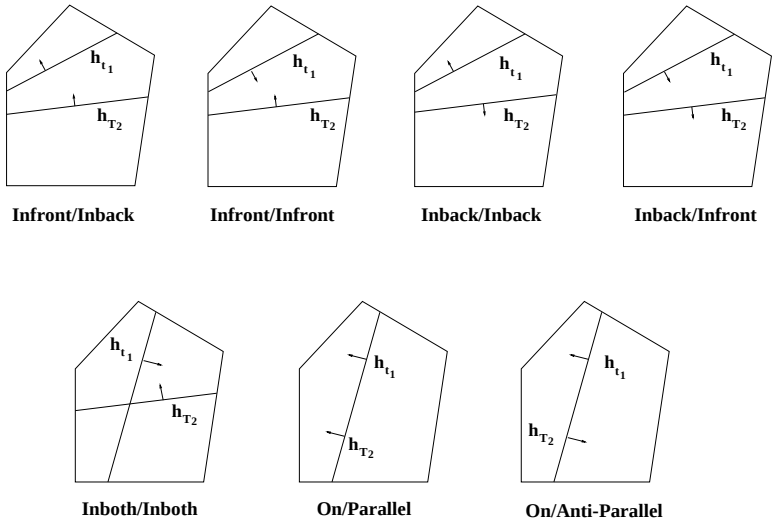

Figure 6: The sub-hyperplane of t_1 in respect to the sub-hyperplane of T_2

One problem with this approach is that $r(t_1) \bigcap r(T_2)$ may be open (if for example t_1 and T_2 are near the root), then the sub-hyperplanes will be open. The solution, as suggested by Thibault in [25], is to represent 3-D space as a bounded set, for example, as a large enough bounding box containing the model. Any hyperplane is first clipped against this box to form a polygon and then is intersected against the node planes.

There are 7 possible classifications between the two sub-hyperplanes, which can be grouped into three sets (in one subtree, in both, coplanar), shown in Figure 6.

Each classification has two parts, the first is found by comparing $shp(t_1)$ against h_{T_2} and it shows the subtree of T_2 in which $shp(t_1)$ lies, possibly in both subtrees. The general idea is that only the subtrees in which $shp(t_1)$ lies will be partitioned while the others will be left unchanged.

For example in the case where we have *Infront/Inback* (Figure 7(a)) the front subtree of T_2 is partitioned to give $T_2.front^+$ and $T_2.front^-$ while $T_2.back$ remains unpartitioned. In the *Inboth/Inboth* case of Figure 8, both subtrees of T_2 are partitioned. In the case where h_{t_1} and h_{T_2} are coplanar no subtree is partitioned.

The third step is to put the pieces resulting from the partitioning of T_2 together to make T_2^+ and T_2^-. For each of the three sets of classifications we proceed as follows:

- $shp(t_1)$ falls entirely in one side h_{T_2}. For this we also need the second part of the classification, $shp(t_2)$ against h_{t_1}. For the case of *Infront/Inback* the resulting trees are (Figure 7(b)):

  $T_2^-.front = (T_2.front)^-$

  $T_2^-.back = T_2.back$

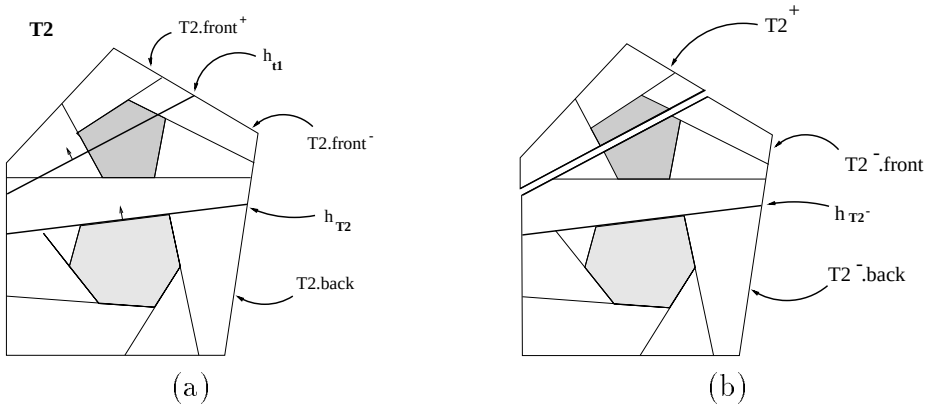

Figure 7: Infront/Inback (a) h_{t_1} partitions $T_2.front$ into $T_2.front^+$ and $T_2.front^-$ and (b) T_2^- and T_2^+ after partitioning

and

$$T_2^+ = (T_2.front)^+$$

the other three cases are analogous.

- for the *Inboth/Inboth* case the two new trees are (Figure 8(b)):

$$T_2^+.front = (T_2.front)^+$$
$$T_2^+.back = (T_2.back)^+$$

and

$$T_2^-.front = (T_2.front)^-$$
$$T_2^-.back = (T_2.back)^-$$

- when the two hyperplanes are coplanar:
if they are parallel and facing the same direction then

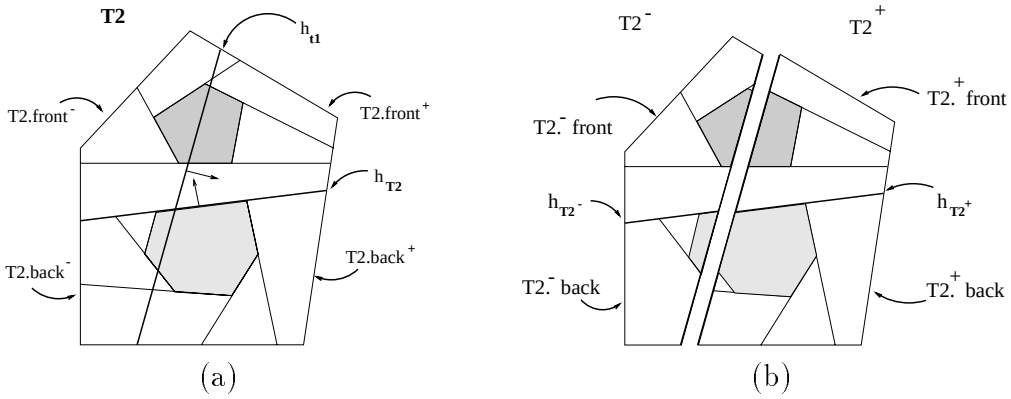

Figure 8: Inboth/Inboth (a) h_{t_1} partitions both $T_2.front$ and $T_2.back$ and (b) T_2^- and T_2^+ after partitioning

$T_2^+ = T_2.front$

$T_2^- = T_2.back$

otherwise (anti-parallel)

$T_2^+ = T_2.back$

$T_2^- = T_2.front$

As we said at the beginning of the section, BSP merging provides a fast and simple way of combining polyhedra, open or closed. Some examples where this could be employed are given in the next 2 sections. Merging has also been used for shadow generation [4, 7, 9] and collision detection. Naylor in [17] describes how a variety of other visual effects can be achieved using merging, such as blasting holes in walls and using a transparent force field to slice through the environment.

5 View Volume Culling

Once we have the merging procedure and a BSP representation of the scene, it is fairly straight forward to implement view volume culling [17]. First we need to construct a BSP tree for the view volume, labelling the leaves as *IN* and *OUT* depending on whether they correspond to the inside or outside regions. Then we can use merging to perform an intersection operation with the scene tree. To achieve that we need to set the *treeOpCell* function so that when we have a tree and an *OUT* region it returns *OUT* and for a tree + an *IN* region it returns the tree. All the polygons that are visible will be on parts of the scene tree that fall in the *IN* regions. Note that since we use the general merging technique, the view volume can have any polyhedral shape, it does not have to be rectangular.

For added performance the merging of the tree and the view volume can be combined with a back to front traversal and instead of creating a new merge tree we can just display the *IN* polygons as we find them.

6 Occlusion Culling

There are several ways in which BSP trees can be used to perform occlusion culling. In this section we will briefly describe the beam tracing method presented by Naylor [**?**]. Another method can be found in the paper by Bittner appended at the back of this document.

6.1 Beam Tracing

The idea resembles that used in the Shadow Volume BSP (SVBSP) trees, which we describe later in the shadow section of the tutorial, but here the use of merging

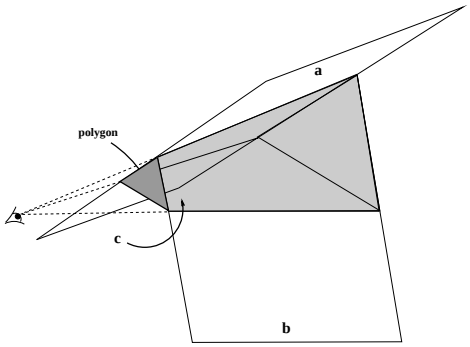

Figure 9: Beam from a single triangle

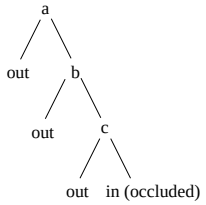

Figure 10: tree structure

makes it much more scalable. As for the SVBSP case, here the beams are also defined by the edges of scene polygons. See for example Figure 10, where we have a beam defined by one triangle and the eye.

Given a set of 3-D polygons represented as a BSP tree and a particular viewpoint we want to build a beam that will represent all the occluded parts of the image.

Initially we start with an empty beam, corresponding to no occlusion, and we traverse the scene BSP tree in front-to-back order. We build the beam of the nearest polygon and as we go further away, at each step we perform a union operation between the beam and the model tree. Any subtree that is occluded by the current beam will be discarded while the beam will be augmented with the planes of the next nearest polygon. This process will stop if we have a beam that covers the whole view or if it occludes all remaining polygons. Or of course until we have processed the whole model.

7 Efficiency Considerations

The issue of creating trees that can be used efficiently is a problematic one, this is because there is no single notion of "efficiency". There are two attributes of the tree that need to be adjusted, size and shape, but they tend to have different importance depending on the application.

As shown by Paterson and Yao [20, 21] for a set of n initial polygons the upper bound for space and time complexity for building a BSP tree is $O(n^2)$, although the expected case is closer to $O(nlogn)$. There can be great variation depending on the partitioning polygon selected as the root at each iteration.

One method that is often used for controlling the size and shape of a tree is to select a few candidate polygons at each iteration and find the best of these to use as root. The evaluation is done by comparing them against the rest of the polygons in the subspace and computing the weighted sum of two quantities, size (number of resulting splits) and distribution (difference in the number of

polygons in each of the resulting subsets).

The weights used depend on the application. For visible surface determination the balance of the tree is not important, since every node is only visited once, but the size is very important. On the other hand for ray tracing or algorithms involving classifications (eg tree merging), balance is more important than size. Also balanced trees are generally faster to build (if the number of splits created is not overwhelming) even though this doesn't reflect the run-time performance.

A different measurement of efficiency, based on expected cost of various operations given by probability models, is presented by Naylor [18]. In simple terms, the idea is to keep the larger cells (with a great probability of being visited) on shorter paths and the smaller ones on longer paths. In a sense this is a sequence of approximations similar to bounding volumes. This method builds trees well suited for merging since in effect the objects are wrapped with the minimal number of sub-hyperplanes extending outside.

Another problem with the existing algorithms for building efficient trees is that they are static in nature. For example, the method of sampling to select a suitable root for each subtree assumes that we have all polygons already at our disposal and also that once a selection is made it is *permanent* in the sense that it cannot be changed when better knowledge is acquired, without rebuilding the subtree involved [15].

When using BSP trees for visible surface determination a dot product operation needs to perfomed with the viewpoint and each partitioning plane of the tree. This could potentially be a costly operation. In recent years a number of "variations" of the BSP trees have been proposed which aim at minimizing this cost [5, 14, 23]. In these methods there is always a sacrifice to be made for the added speed. This is usually in terms of memory and in the functionality of the resulting tree (none of them can be used for merging for example).

8 BSP Trees in Dynamic Scenes

The problem of using BSP trees in dynamic scenes was something that researchers from the very infancy of the algorithm have tried to solve. The earliest work, even though not as yet on BSP trees, was that of Schumacker [24]. In his algorithm, which is considered to be the predecessor of the modern BSP trees, the tree was built using manually defined separating planes between objects. Each of these objects would have its faces sorted into a visibility order valid from any viewpoint (after back-face elimination). The ordering between the objects as seen from each tree cell was pre-calculated and stored and so at run time locating the position of the viewpoint was all that was needed, to get the priority order. In this structure the objects were allowed to move, without any recalculation of the tree, as long as they did not cross any of the separating planes.

The first partial solution for dynamic changes of BSP trees was given by Fuchs

[11] and it was based on the same principle as that of Schumacker. If we know in advance the objects that will be moving and the region in which they will do so then a tree can be constructed such that the relevant region is enclosed in a tree cell. Then the objects can move in that region independently with regard to the rest of the tree.

Sudarsky and Gotsman [**?**] also place bounding volumes around the areas where object might move but they use them even more efficiently since they don't add a moving object at all if it's corresponding movement volume is not in view. However, this is probably more important for minimising message passing in a distributed situation rather than for speeding up the tree modification.

A different method that again involves knowing the objects that will be moving in advance but not their path, was used by Naylor [16, 17]. First a tree the static objects are built into one tree, with this we merged the tree of the moving object at each frame to produce a complete scene tree. No removal is ever necessary since the original copy of the static tree is used for merging at each frame.

Torres [27] presented a BSP tree with several optimisations over the standard structure. Each object has its own single BSP tree which is built by considering the polygons of the object alone. These single trees form the leaves of the scene BSP tree of which the internal nodes are separating planes between the objects. If such separating planes cannot be found then user defined partitioning planes are required. To make the single trees of convex objects more balanced and speed up their construction, *halving planes* are sometimes used, which are planes chosen to be parallel to as many object polygons as possible to minimise splits and positioned so as to have an almost equal number of the polygons on each side.

Also wrapping planes are sometimes used over complex objects in order to minimise the splitting of objects that are not linearly separable. Speed ups are obtained over the initial building of the tree and the moving of certain objects but the general idea as far as interaction is concerned is not much different from that of Schumacker.

None of these algorithms is general enough. Chrysanthou and Slater in [10] suggest a simple method which requires no prior knowledge of the moving objects. Here the polygons of a moving object, and the nodes they define, are removed from the tree and re-inserted at their new position at each frame. This seems to work well for small moving objects in the limited experiments they performed. However the theoretical question of efficiently removing nodes with no empty sub-trees still remains. Some ideas on this are given in [8] but they haven't been properly tested.

There has also been some work in Computational Geometry on dynamic BSP trees [2, 3].

References

[1] Michael Abrash. *Zen of Graphics Programming.* Coriolis Group Books, second edition, 1996.

[2] P. Agarwal, L. Guibas, T. Murali, and J. Vitter. Cylindrical static and kinetic binary space partitions. In *Proceedings of the 13th International Annual Symposium on Computational Geometry (SCG-97)*, pages 39–48, New York, June 1997. ACM Press.

[3] Pankaj K. Agarwal, Jeff Erickson, and Leonidas J. Guibas. Kinetic binary space partitions for intersecting segments and disjoint triangles (extended abstract). In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 107–116, San Francisco, California, 25–27 January 1998.

[4] A. T. Campbell. *Modelling Global Diffuse Illumination for Image Synthesis.* PhD thesis, Department of Computer Science, University of Texas at Austin, December 1991.

[5] Han-Ming Chen and Wen-Teng Wang. The feudal priority algorithm on hidden-surface removal. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 55–64. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[6] N. Chin and S. Feiner. Near real-time shadow generation using BSP trees. *ACM Computer Graphics*, 23(3):99–106, 1989.

[7] N. Chin and S. Feiner. Fast object-precision shadow generation for area light sources using BSP trees. In *ACM Computer Graphics (Symp. on Interactive 3D Graphics)*, pages 21–30, 1992.

[8] Y. Chrysanthou. *Shadow Computation for 3D Interaction and Animation.* PhD thesis, Queen Mary and Westfield College, University of London, February 1996.

[9] Y. Chrysanthou and Slater M. Incremental updates to scenes illuminated by area light sources. In J. Dorsey and Ph. Slusallek, editors, *Rendering Techniques '97*, pages 103–114. Springer Computer Science, 1997.

[10] Y. Chrysanthou and M. Slater. Dynamic changes to scenes represented as BSP trees. *Computer graphics Forum, (Eurographics 92)*, 11(3):321–332, 1992.

[11] H. Fuchs, G. D. Abram, and E. D. Grant. Near real-time shaded display of rigid objects. *ACM Computer Graphics*, 17(3):65–72, 1983.

[12] H. Fuchs, Z. M. Kedem, and B. F. Naylor. On visible surface generation by a priori tree structures. *ACM Computer Graphics*, 14(3):124–133, 1980.

[13] D. Gordon and S. Chen. Front-to-back display of BSP trees. *IEEE Computer Graphics & Applications*, 11(5):79 –85, 1991.

[14] A. James and A.M. Day. The priority face determination tree on hidden surface removal. *Computer Graphics Forum*, 17(1):55–72, 1998.

[15] S. McPartlin. A foundation for BSP trees. Second year progress report. Technical report, University of Edinborough, October 1994.

[16] B. F. Naylor. Sculpt: An interactive modeling tool. In *Proceedings of the Graphics Interface '90*, pages 138–148, 1990.

[17] B. F. Naylor. Interactive solid geometry via partitioning trees. In *Proceedings of the Graphics Interface '92*, pages 11–18, 1992.

[18] B. F. Naylor. Constructing good partitioning trees. In *Proceedings of the Graphics Interface '92*, pages 181–191, 1993.

[19] B. F. Naylor, J. Amandatides, and W. Thibault. Merging BSP trees yields polyhedral set operations. *ACM Computer Graphics*, 24(4):115–124, 1990.

[20] M. S. Paterson and F. F. Yao. Binary partitions with applications to hidden surface removal and solid modeling. In *Proceedings of the 5th Annual ACM Symposium on Computational Geometry*, pages 23–32, 1989.

[21] M. S. Paterson and F. F. Yao. Optimal binary space partitions for orthogonal objects. *Discrete Computational Geometry*, 5:485–503, 1990.

[22] H. Radha, R. Leonardi, M. Vetterli, and B. Naylor. Binary space partitioning tree representation of images. *Journal of Visual Communication and Image Representation*, 2(3):201–221, 1991.

[23] Amela Sadagic. *Efficient image display for head-slaved viewing of virtual environments.* PhD thesis, Department of Computer Science, University College London, March 1999.

[24] R. Schumacker, B. Brand, M. Gilliland, and W. Sharp. Study for applying computer-generated images to visual simulation. Technical Report AFHRL-TR-69-14, NTIS AD700375, U.S. Air Force Human Resources Lab., Air Force Systems Command, Brooks AFB, TX,, September 1969.

[25] W. C. Thibault and B. F. Naylor. Set operations on polyhedra using binary space partition trees. *ACM Computer Graphics*, 21(4):153–162, 1987.

[26] A. O. Tokuta. Motion planning using Binary Space Partitioning. In *IEEE International Workshop on Intelligent Robots and Systems IROS '91*, pages 86–90, Osaka, Japan, November 1994.

[27] E. Torres. Optimization of the binary space partition algorithm (BSP) for visualization of dynamic scenes. *Computer graphics Forum, (Eurographics 90)*, 9(3):507–518, 1990. C.E. Vandoni and D.A. Duce (eds.), Elsevier Science Publishers B.V. North-Holland.