BSP (=Binary Space Partition) Trees

BSP trees are a geometric tool that can be used for a variety of tasks, including resolving visibility, computing shadows and also to reject polygons that are outside the view volume. BSPs are too complex to implement in hardware (at least today), so those operations will be performed by the CPU rather than the GPU. BSPs are particularly useful for walkthrough/flythrough applications where the viewpoint is allowed to move but (most of) the scene and the lights are fixed (however, extending them to handle scene is, to some extent, possible). Historically, they have been used with great success in flight simulators. Variants of BSP trees are commonly used in computer games too.

1 Idea

BSP trees come in a number of varieties, but the basic idea is always the same: to recursively partition the space using planes (in our 2D examples, lines). The partitioning can be represented as a tree, which has the partitioning planes at the internal nodes and objects of interest at the leaves. The edges of the BSP tree can be thought of as corresponding to 'sides' of the separating planes. For example, the left edge of the root in Figure 1 corresponds to the half-space bounded by p_1 that contains the objects A, F and E (notice that these are the objects at the leaves of the subtree rooted at p_2, the other end of that edge).

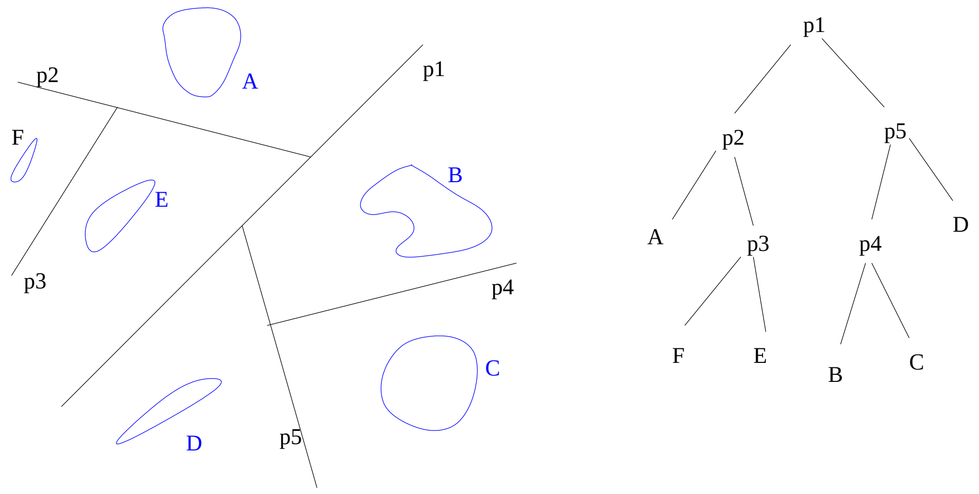

Figure 1: A BSP Tree. The separating planes are shown in black and the objects are shown in blue.

A BSP Tree can be constructed with a recursive procedure that first selects a separating plane (here, we were trying to find one which has roughly the same number of objects on the left and on the right, but it is not absolutely necessary), puts it at the root and divides the objects into two groups, each of them consisting of objects on one side of the plane. Then, it recursively constructs the subtrees rooted at the children of the node corresponding to the separating plane. Of course, if one of the groups consists of just one object, then is is not split any further and a leaf node in the tree, corresponding to that object, is created.

In our example, the first plane that is selected is p_1, splitting the objects into two groups $\{A, F, E\}$ and $\{B, C, D\}$. The first group is put into the left subtree, the second one - into the right subtree. Then, p_2 is selected as the splitting plane for the first group and made the root of the left subtree. On one side of p_2, there is only one object (A), and it forms a leaf node-child of p_2. On the other side, we have two objects and therefore we select a plane separating them and make it a child of p_3. Now, p_3 has only one object at each side and we make them leaves and children of p_3. The same procedure is applied to build the right subtree of p_1.

2 Painter's algorithm

Painter's algorithm allows to order the objects stored at the leaves of a BSP tree in back-to-front order from the viewpoint (located in any place). Back to front order means that occluders appear later in the order than ocludees (the objects that occlude them, either partially or completely). Notice that, in particular, this means that all the objects can be drawn in this order, even without using the z-buffer, and the final scene will have the correct visibility (the objects that are closer to the eye will overwrite those which are further away).

How to obtain the back-to-front ordering using a BSP tree? The recursive procedure is like this:

```
Painter ( $n$: node of the BSP tree, $V$ (=viewpoint) )
if $n$ is a leaf node, output the object it contains and return
Here we know that $n$ is an internal node. It corresponds to some partitioning plane. Let $n_V$ and $n'_V$ be its children, corresponding to the side of that plane that contains the viewpoint and which does not contain the viewpoint ( respectively).
Recursively call Painter($n'_V$,$V$)
Recursively call Painter($n_V$,$V$)
```

3 Splitting

The problem with separating planes is that they don't always exist. For example, they don't for the four rectangles shown in Figure 2 (if they did, we would be able to obtain a back to front ordering of the rectangles by running the painter's algorithm; in this ordering red would appear after blue, which would appear after black, which would appear after green which would appear after

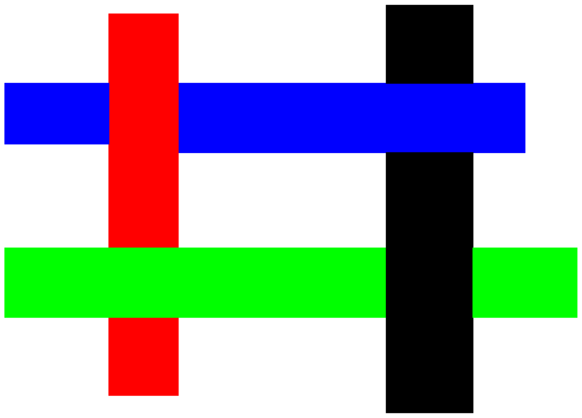

Figure 2: A BSP Tree.

red... contradiction!).

In spite of this, it is possible to construct a BSP tree. However, in general, this requires splitting (here, cutting using planes) the objects into smaller ones. Basically, the procedure is to select randomly a polygon, make it the root of the tree and use it's plane as a splitting plane: cut all polygons intersecting the plane of that polygon into two. Then, the same procedure is applied recursively to construct the two subtrees of our tree. Each of them is built from polygons on one side of the splitting plane.

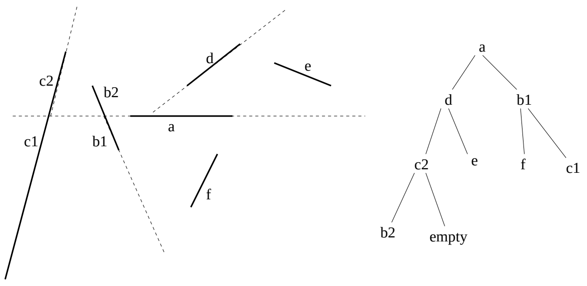

Figure 3: A BSP tree for any collection of polygons

In the example in Figure 3, the first polygon we select is a. It splits the remaining polygons into two groups: $\{b2, c2, d, e\}$ and $\{b1, c1, f\}$ (notice that polygons in each group are in the same subtree rooted at a chld of a). To build the left subtree, we select a polygon (here, d) and again use its plane as the splitting plane. In this case, that plane does not cut any of the polygons. We put d at the root of the subtree and proceed to construct subtrees rooted at its children from the two groups of polygons $\{e\}$ and $\{b2, c2\}$... By convention, when we encounter a polygon whose plane has nothing on one side, we put an 'empty' node into our tree. In this way, all non-leaf nodes have two children.

The children can be thought of as corresponding to two sides of the plane of the polygon at the parent node.

The Painter's algorithm can also work for this kind of a BSP tree. To output all polygons in back-to-front order (some of them resulting from splitting operations), it first outputs all polygons in the subtree that corresponds to the side of the splitting plane at the root that does not contain the viewpoint (by recursively calling itself for the corresponding subtree). Then, it outputs the polygon at the root. Finally, it outputs all polygons in the other subtree (this is also a recursive call).

4 SVBSP trees

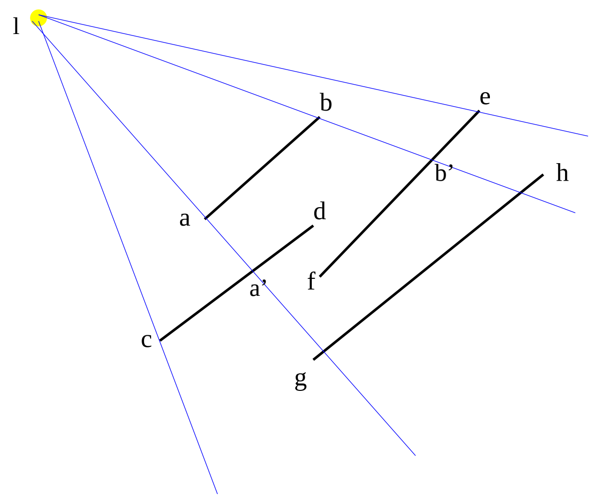

Figure 4: Example with a light source and four lines. The nodes of the corresponding SVBSP tree will contain the blue lines and some of the intervals.

One can use the ideas provided by BSP trees to compute shadows. The trees are called shadow volume BSP trees (SVBSP for short). They can be thought as decision trees that allow answering shadow queries in a simple manner. A shadow query asks whether a point is in shadow or not. As usual, let's do a 2D example (Figure 4). The internal nodes of an SVBSP tree will be lines. Two types of lines will be of importance: (A) the lines that pass through the light source and an endpoint of an interval and (B) the lines that contain one of the intervals. Each of the leaves will simply be a query result: 'lit' or 'in shadow'. When resolving a query, we will descent down the tree. At each internal node, we will select a child to depending on which side of the line associated with that node we are.

For nodes of type (A) (blue lines in 4) we will test whether the interval is on the same side as the query point or not. If it is, we'll descend to the left child.

Otherwise, we'll descend to the righ child.

For nodes of type (B) (lines containing the intervals) we will test whether the query point is on the same side of the line as the light source. If it is, we'll descend to the right. Otherwise, we'll descend to the left.

We'll find the result of the query at the leaf we reach making the above decisions on the way.

4.1 Construction of SVBSP

We'll construct the tree in an incremental fashion, by inserting to it the intervals in the front-to-back order from the light source. One can obtain such ordering e.g. by building a BSP tree and running the 'reverse' painter's algorithm (in which the traversal order for children is reversed).

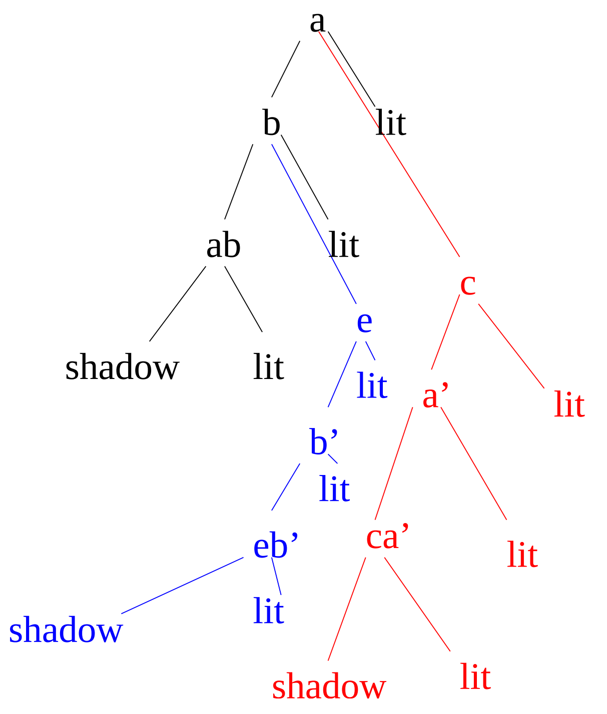

Figure 5: SVBSP for intervals shown in Figure 4.

We start from the tree for an empty set of intervals. This is a trivial tree which has just one node that says 'lit' (no lines=no shadows).

The tree after the first interval $\bar{ab}$ is inserted is shown in black in Figure 5 (check that queries really work!). Adding an interval is a lot like executing a query for the entire interval. Here is what happens when we insert $\bar{cd}$ to the black tree. At the root, we ask whether we are on the same side of line(l, a) as the interval $\bar{ab}$. For the entire interval $\bar{cd}$, this is a question that cannot be answered in a definite way, sine it is different for points in the subinterval $\bar{ca'}$ and $\bar{a'd}$ (to get a', split $\bar{cd}$ with the line line(l, a)). Therefore, we split the interval and seek the answer for each resulting subinterval separately. For $\bar{da'}$, this means descending to the left child of a, then to the left child of b and finally to the left child of ab, a leaf saying 'shadow'. This means that the subinterval $\bar{da'}$ is entirely in shadow and therefore inserting it does not really increase the shadow (in this particular case, points in shadow of $\bar{da'}$ are already in shadow because of $\bar{ab}$). Now we return to $\bar{ca'}$. Since it's not on the same side of line(la), we descrnt from a to its right child, which says 'lit'. There, we extend our SVBSP tree by inserting an SVBSP tree for the interval $\bar{ca'}$ (shown in red).

Similarly, inserting the interval $\bar{ef}$ leads to adding the blue nodes and edges to the SVBSP tree. When the last interval, $\bar{gh}$, is being inserted into the tree, it is split into two at the root node 'a'. One its subinterval descends to the left, the other- the right. One of them is further split into two at the 'b' node. For each of the subintervals, the algorithm will eventually find out that it is in shadow (try yourself!). Therefore, the tree ned not be extended. The final tree for the arrangement in Figure 4 is shown in Figure 5 (consists of all edges and nodes regardless of their colors).