Chapter 8

BSP Trees

8.1 Introduction

In this document, we assume that the objects we are dealing with are represented by polygons. In fact, the algorithms we develop actually assume the polygons are triangles, though they can be generalized to other polygons. In previous lectures, we learned how to get a triangle from the 3D world in which our objects live onto the 2D screen on which the objects will be viewed using projection and other techniques. When one looks at a 3D object from a certain view point, it is never possible to see all the sides of this object. This means that when this object is projected onto a 2D surface, some of the triangles which make up the object should not be seen because they are hidden from the viewer. In other words, they should be eliminated. This is known as hidden surface elimination. There are many techniques to achieve this. In this document, we develop a technique to achieve this which uses BSP trees. BSP stands for Binary Space Partitioning. The reason for this name will become obvious as we describe the process. The technique we are about to describe works well when we are making many images from different view points of the same geometry.

8.2 Overview

The BSP tree algorithm is an example of a painter's algorithm. Such an algorithm draws objects from back to front. Polygons which are behind other polygons and should not be seen will be drawn over and will not be seen. Thus, this will perform hidden surface elimination automatically. This is illustrated on figure 8.1. The algorithm is very simple. It can be implemented as follows:

Algorithm 93 (Painter's Algorithm) *Follow these steps:*

1. *Sort objects from back to front.*

2. *For each object do*

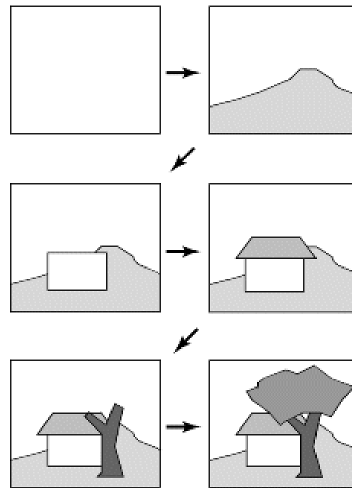

Figure 8.1: An illustration of the painter's algorithm

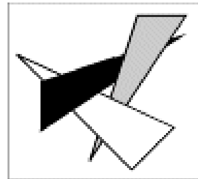

Figure 8.2: A cycle

- *Draw object on the screen*

In reality, it is not as simple. There is one major difficulty, the relative order of multiple objects is not always well defined. This makes the first step of our algorithm not easy to implement. Figure 8.2 shows an example of what is called a cycle. The relative order of the three triangles is not clear. To avoid a situation like the one of figure 8.2, we must make sure that no triangle crosses the plane defined by any other triangle. This is a very restrictive assumption. First, we will develop an algorithm with this assumption. Then, we will see how to relax this condition and still have a painter's algorithm. Before we do this, let us review some mathematical concepts which will be very useful here.

8.3 Quick Math Review

Let $a = (x_a, y_a, z_a)$ be a point and $\vec{n} = (A, B, C)$ be a vector. You will recall that the equation of the plane containing a with normal $\vec{n}$ is given by

$$(p - a) \cdot \vec{n} = 0 \tag{8.1}$$

In this equation, $p = (x, y, z)$ represents a point and equation 8.1 is simply the condition the point must satisfy to be on the plane.

If we plug in the coordinates of the points and vector involved, equation 8.1 becomes

$$(x - x_a) A + (y - y_a) B + (z - z_a) C = 0$$

which can be written as

$$Ax + By + Cz + D = 0 \tag{8.2}$$

where

$$\begin{aligned} D &= -Ax_a - By_a - Cz_a \\ &= -\vec{n} \cdot a \end{aligned}$$

This can also be written as

$$\vec{n} \cdot p + D = 0 \tag{8.3}$$

Given a point p, we can define a function f by

$$f(p) = (p - a) \cdot \vec{n} \tag{8.4}$$

Thus, a point p is on the plane if

$$f(p) = 0$$

What about if the point is not on the plane? Either we will have $f(p) > 0$ or $f(p) < 0$. Remembering that

$$(p - a) \cdot \vec{n} = \|(p - a)\| \|\vec{n}\| \cos\theta$$

where θ is the smallest angle between $(p - a)$ and $\vec{n}$ taken between 0 and π. We see that if p is on the side of the plane where $\vec{n}$ is pointing, the angle between $(p - a)$ and $\vec{n}$ will be between 0 and $\frac{\pi}{2}$ and will therefore be positive. Hence, we will have $f(p) > 0$. Similarly, if p is on the side of the plane opposite the direction where $\vec{n}$ is pointing, we will have $f(p) < 0$. Thus, the function f which defines the plane gives us very important information. It tell us the following:

Proposition 94 *Let f be the function defining the plane containing a with normal $\vec{n}$ (as in equation 8.2, and let p be a point. Then, the following is true:*

1. *If $f(p) = 0$, then p is on the plane.*

2. *If $f(p) > 0$, then p is on the side of the plane where $\vec{n}$ is pointing.*
3. *If $f(p) < 0$, then p is on the side of the plane opposite the direction where $\vec{n}$ is pointing.*

Given a triangle T with vertices $a = (x_a, y_a, z_a)$, $b = (x_b, y_b, z_b)$ and $c = (x_c, y_c, z_c)$, we can write the equation of the plane containing this triangle. We can select any of the vertices as the point the plane will contain (since it will contain all the vertices). Let us select a. The normal is given by $(b-a)\times(c-a)$. Thus, the function f defining the plane which contains T is

$$f(p) = (p-a)\cdot((b-a)\times(c-a)) \tag{8.5}$$

This is called the scalar triple product. Geometrically, $|f(p)|$ represents the volume of the parallelepiped determined by the vectors $p-a$, $b-a$ and $c-a$. It can be proven (see a multivariable calculus book) that

$$f(p) = \begin{vmatrix} x - x_a & y - y_a & z - z_a \\ x_b - x_a & y_b - y_a & z_b - z_a \\ x_c - x_a & y_c - y_a & z_c - z_a \end{vmatrix} \tag{8.6}$$

Expanding this determinant gives

$$\begin{aligned} f(p) \quad = \quad & (z_c(y_b - y_a) - y_c(z_b - z_a))\,x \\ & + (x_c(z_b - z_a) - z_c(x_b - x_a))\,y \\ & + (y_c(x_b - x_a) - x_c(y_b - y_a))\,z \\ & + (x_c(y_a z_b - y_b z_a) + y_c(x_a z_b - x_b z_a) + z_c(x_b y_a - x_a y_b)) \end{aligned} \tag{8.7}$$

A point p will be on the triangle with vertices a, b, c if and only if $f(p) = 0$ where f is as in equation 8.7. It is important that we be consistent when we form the cross product to find the normal of a triangle so that all the normals are oriented the same way.

8.4 The Simple Case of Two Triangles

We begin by illustrating our algorithm with two triangles. Let T_1 and T_2 be two triangles. Let the plane containing T_1 be defined by the function f_1. Though this is not correct, we will use the notation $f(T)$ to mean $f(p)$ for every vertex of the triangle T. Let e denote the view point. Our goal is to sort these two triangles from the furthest to the closest of e. We know that $f_1(T_1) = 0$. If $f_1(e)$ and $f_1(T_2)$ have the same sign, then e and T_2 are on the same side of T_1. Thus T_1 is the furthest. If $f_1(e)$ and $f_1(T_2)$ have opposite signs, then e and $T_2$are on opposite sides of T_1. Thus T_2 is the furthest. Thus, we have the following algorithm:

Algorithm 95 *To sort triangles T_1 and T_2 with respect to the viewpoint e, follow the steps below:*

- *If* ($f_1(e)$ *and* $f_1(T_2)$ *have the same sign*) *then*
  - *Draw* T_1
  - *Draw* T_2
- *else*
  - *Draw* T_2
  - *Draw* T_1

8.5 A First Algorithm

We now generalize the algorithm described above to the case of many objects. We still have our restriction that no triangle crosses the plane defined by any other triangle. To derive our algorithm, we will create a binary tree data structure. This means that at each node, there can only be at most two edges (or branches). In our case, the two branches will be the positive branch and the negative branch. The positive branch will contain those triangles for which $f(p) > 0$ where f is the function defining the plane containing the triangle at the root of the tree.. We will present two procedures. One which creates the tree, the other one which traverses the tree to draw the triangles in the correct order. Suppose that we have triangles labeled $T_1, T_2, ..., T_n$. The procedure to create the tree will take the triangles one at a time and figure out where to put it in the tree. Since T_1 is the first triangle in the list, we will make it the root of the tree. As above, let e denote the viewpoint. Let f_i be the function which defines the planes containing T_i. We will use the subscripts when we mean to talk about a specific triangle. Otherwise, we will use T to denote a triangle, and f to denote the function defining the plane containing it.

The algorithms we are about to write are recursive. It means that they can call themselves. So, to implement them the chosen programming language has to support recursion, which is the case for the most commonly used programming languages such as C, C++, java, pascal, ... Let us assume further that our tree is implemented by a data structure called **bsptree**.

The function (method) to draw the tree is shown below. It assumes that at each node, the branch on one side should be drawn first, then the node should be drawn, then the branch on the other side should be drawn last. Which side is drawn first depends on where the view point is with respect to the current node.

- **function** draw(bsptree tree, point e)
- If (tree.empty) then
  - return
- if ($f_{tree.root}(e) < 0$) then

- draw(tree.plus,e)
- plot tree.triangle
- draw(tree.minus,e)

- else
  - draw(tree.minus,e)
  - plot tree.triangle
  - draw(tree.plus,e)

The code which builds the tree which can be traversed by the code above is:

- tree-root = node(T_1)
- for ($i = 2$ to n) do
  - tree-root.add(T_i)

The add function is defined as follows:

- function add(Triangle T)
- if ($f(a) < 0$ and $f(b) < 0$ and $f(c) < 0$) then
  - if(negative-subtree is empty) then
    - negative-subtree = node(T)
  - else
    - negative-subtree.add(T)
- if ($f(a) > 0$ and $f(b) > 0$ and $f(c) > 0$) then
  - if(positive-subtree is empty) then
    - positive-subtree = node(T)
  - else
    - positive-subtree.add(T)
- else
  - for now, we assumed this was not possible

Practice building a tree and then drawing the rectangles from it assuming we have the following rectangles: $T_1, T_2, ..., T_{10}$ arranged in the following order: $T_5\ T_8\ T_4\ T_3\ T_1\ T_2\ T_6\ T_{10}\ T_7\ T_9\ e$

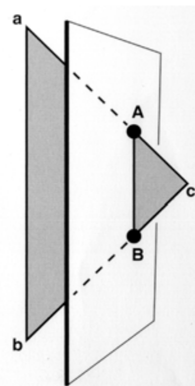

Figure 8.3: Triangle crossing the plane of another triangle

8.6 Triangle Cutting

The only case left is when a triangle crosses the dividing plane as shown in figure 8.3When it happens, two vertices of the triangle are on one side, the third is on the other. We are showing the case when c is on the other side. There are two other cases. We will write an algorithm for this case. If it is not c but one of the other vertices, we will do some swapping first. Remember, we always keep the vertices in the same order (counterclockwise) so that the normal point in the same direction. When we are in this case, we create new triangles. We create $T_1 = (a, b, A)$, $T_2 = (b, B, A)$ and $T_3 = (A, B, c)$. We also need to worry about another issue. If the vertex which is by itself happens to be very close to the plane, then T_2 would be almost flat, and T_3 would have an area almost equal to 0. Furthermore, because of errors introduced in every computations, 0 is never 0 in computer science. It is a very bad habit to check if something is equal to 0. We should never have a statement like if $x == 0$. Instead, we should check if $abs\,(x) < \varepsilon$, where ε can be considered to be 0. With this in mind, we can rewrite the above algorithm as follows:

- function add(Triangle T)
- $fa = f\,(a)$
- $fb = f\,(b)$
- $fc = f\,(c)$
- if $(abs\,(fa) < \varepsilon)$ then
  - $fa = 0$
- if $(abs\,(fb) < \varepsilon)$ then

  - $fb = 0$

- if $(abs\,(fc) < \varepsilon)$ then
  - $fc = 0$
- if $(fa \leq 0$ and $fb \leq 0$ and $fc \leq 0)$ then
  - if(negative-subtree is empty) then
    * negative-subtree = node(T)
  - else
    * negative-subtree.add(T)
- if $(fa \geq 0$ and $fb \geq 0$ and $fc \geq 0)$ then
  - if(positive-subtree is empty) then
    * positive-subtree = node(T)
  - else
    * positive-subtree.add(T)
- else
  - cut triangle into three triangles and insert the new triangles into the tree.

The algorithm to cut the triangles is:

- If $(fa * fc \geq 0)$ then
  - swap(fa, fc)
  - swap(a, c)
  - swap(fb, fc)
  - swap(b, c)
- else if $(fb * fc \geq 0)$ then
  - swap(fb, fc)
  - swap(b, c)
  - swap(fa, fc)
  - swap(a, c)
- Compute A
- Compute B

- if ($fc \geq 0$) then
  - negative-subtree.add(T_1)
  - negative-subtree.add(T_2)
  - positive-subtree.add(T_3)
- else
  - positive-subtree.add(T_1)
  - positive-subtree.add(T_2)
  - negative-subtree.add(T_3)

We can find A as follows. A is at the intersection of the dividing plane and the line through a and c. The line through a and c has equation

$$p(t) = a + t(c - a)$$

We recall from equation 8.3 that the dividing plane is given by $\vec{n} \cdot p + D = 0$. Combining the two gives

$$\begin{aligned} \vec{n} \cdot a + t\vec{n} \cdot c - t\vec{n} \cdot a + D &= 0 \\ t &= -\frac{\vec{n} \cdot a + D}{\vec{n} \cdot (c - a)} \end{aligned}$$

Therefore,

$$A = a - \frac{\vec{n} \cdot a + D}{\vec{n} \cdot (c - a)}(c - a)$$

We obtain a similar expression for B.

8.7 Assignment

1. Prove equation 8.7
2. Derive the expression for B at the end of the last section (just above).
3. Given N triangles, what is the minimum number of triangles that could be added to a resulting BSP tree? What is the maximum?
4. Think about and be ready to discuss in class the implementation of a simple $3D$ graphics pipeline which would only involve projection. It would take as input a set of triangles representing the various objects in the scene. The output would be pixel coordinates. Be very specific as to how you would implement this.