Computer Graphics – Lab Assignment
Transparency and Depth Sorting

November 30, 2010

1 Introduction

Until now we have only worked with completely opaque geometry. And although last week's leaf assignment used a texture with transparent areas, these areas were *fully* transparent and didn't need to be rendered, while the non-transparent parts of the leaf formed more-or-less a normal piece of opaque geometry.

In this final lab assignment we're going to look at how to handle *semi*-transparent geometry, such as a window's glass pane modelled with a 95% transparent quad with a slight blueish color.

The common OpenGL method for drawing such geometry is to use "blending". With blending enabled, OpenGL uses a *blending function* to compute updated pixel colors. For each pixel this function takes the color of the pixel as currently stored in the frame buffer together with the color of the next piece of geometry to be drawn and computes a new pixel color[1] . See Figure 1 for the kind of transparency effects this can produce.

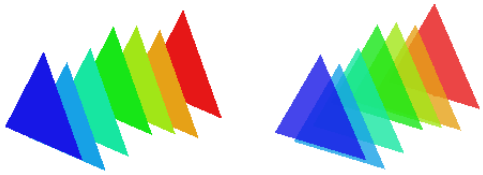

Figure 1: Without blending (left), with blending (right)

As can be seen, by enabling blending the different semi-transparent layers augment the color of the geometry seen through them. But the "problem" with this kind of geometry is that it introduces a restriction on the way the geometry is drawn. As the frame buffer (and depth buffer as well) only stores a single (color) value we need to draw the geometry in a specific order to get a correct rendering. When, for example, we first draw a piece of opaque geometry followed by a transparent piece that is in front of the opaque piece (as seen from the viewpoint used) we can no longer correctly draw a piece of transparent

[1] Or, more correctly, the blending uses the color of the geometry as seen through the current pixel, which might vary across its surface because of texturing and shading.

geometry that is in between the two pieces already drawn. The final pixel colors in that case will almost certainly be incorrect.

So for the case of having one or more pieces of transparent geometry the construction of the correct image requires that we draw from back to front with respect to the viewpoint used. Unfortunately, there is no support in OpenGL to determine this order for a given 3D scene. There are even situations in which there simply is no correct order (think of two transparent polygons that intersect).

In this assignment we will look at a general way to do correct back-to-front rendering for a scene containing transparent geometry, including intersecting geometry.

2 BSP Trees

We are going to use a Binary Space Partitioning (BSP) tree to perform correct depth-sorted rendering of transparent geometry. What follows is a quick overview of BSP trees, for an extensive introduction see Shirley et.al, section 8.1.

A BSP tree is a data structure that allows organising and subdividing a polygonal model in such a way that a depth-sorted traversal for correct transparency rendering can be done efficiently[2]. Building a BSP tree can be seen as a preprocess to rendering. A bit of work is needed for constructing a tree, while actually traversing the tree during rendering is relatively easy and fast. As long as the geometry in a 3D scene doesn't change the corresponding BSP tree stays valid. Here, we will only deal with static geometry.

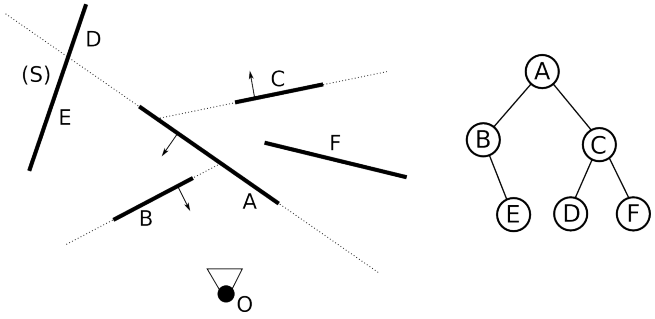

Figure 2: Polygons and the split planes they define (left); The corresponding BSP tree (right)

See Figure 2. A node in a BSP tree contains a (planar) polygon which defines a split plane, dividing space into two sub-spaces[3]. For each sub-space polygons

[2]There are actually more uses for BSP trees, but we are only interested in depth-sorting for transparency here. See `http://en.wikipedia.org/wiki/Binary_space_partitioning` and `http://en.wikipedia.org/wiki/Doom_engine` for more examples

[3]In principal we can use arbitrary split planes, but as we need to store scene triangles anyway we can simply use one vertex of a triangle and its normal vector to define a BSP split plane. This also overcomes the problem that there are an infinite number of arbitrary split planes that could possibly be used, while there is only a finite number of scene triangles and corresponding planes. This saves us from having to pick planes from an infinite set.

that occupy that sub-space are stored as a BSP sub-tree. For example, polygon A is the root node of the tree. Its left child is the subtree defined by B and its children, its right child the subtree defined by C and its children.

An invariant we use throughout this assignment is that the left child of a BSP node contains only polygons that are on the side of the plane that the node's plane normal points to. And subsequently that the right child has the polygons NOT on the side the normal points to. This is important to remember, as both the tree construction as well as the traversal will need to consistently use this invariant for correct results.

In our case only the transparent polygons of the scene will be present in the tree, as the opaque geometry doesn't need depth-sorting for rendering. As long as we draw all opaque geometry before drawing any semi-transparent geometry this should produce a correct rendering.

The way we can use a BSP for depth-sorted rendering is to take the observer's position O (the viewpoint) and check the root node of the tree. As you can see O is on the "left" side of A (the side where A's normal vector points to). The correct back-to-front order is then 1) Draw A's right subtree, 2) Draw A itself, 3) Draw A's left subtree. For each subtree the same test of observer position needs to be done so the correct sub-subtree rendering order is used. By recursively traversing the tree in this way all the polygons will get drawn in the correct order.

Construction of a BSP tree is fairly straightforward. 1) Start with all the polygons in a scene and pick one arbitrarily to define the split plane. 2) Sort the remaining polygons into groups that are on the left and right sides of the plane. 3) Recursively subdivide the two groups obtained this way by applying step 1 and further. 4) Create a node which has the two subtrees resulting from step 3 as children and return it.

But there's a catch in step 2. As can be seen in Figure 2, polygon S straddles the plane defined by A. So we can't classify it uniquely to be on either left or the right side of A's plane. During tree construction we will therefore need to split such polygons, e.g. we need to split S along A's plane into D and E. The latter two polygons can be unambiguously classified to be in the left or right groups.

In the code you need to write you will have to test how a point is positioned relative to a split plane. Given a plane defined by a point $\mathbf{p}$ and normal vector $\mathbf{n}$, and a point $\mathbf{x}$ that you want to test, you can compute $(\mathbf{x} - \mathbf{p}) \cdot \mathbf{n}$. For the in-product of two vectors the following holds:

$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}||\mathbf{b}| \cos \alpha$$

where α is the angle between the vectors $\mathbf{a}$ and $\mathbf{b}$. As the two vector lengths in the above equation are always positive the sign of the in-product (positive, negative or zero) is determined by the sign of the cosine, and therefore depends on the angle between the vectors. Therefore, the sign of the in-product is all you need to determine where $\mathbf{x}$ lies with respect to the plane.

3 Framework

The framework for this week's assignment is very similar to the one used in the texture mapping assignment. The scene once again contains a house, but the

walls of the house are different this time. Two of the walls (one short one, one long one) are made of multiple polygons, where the center polygons of the wall represent a window. This window polygon is untextured and 50% transparent. The other two walls consist of two triangles (forming a quad) and use a texture of which the center part has an alpha value of 0.5. These walls therefore use different methods for creating models with transparency.

The use of texturing can be toggled with **t**. Toggling the display of opaque geometry, to inspect only the transparent geometry, can be done with **o**. You can switch to line drawing mode with the **l** key, which will show all the edges in the scene (and back to polygon mode with **p**). This can help when working on polygon splitting later on.

4 Assignment - Depth-sorting using a BSP tree

We have a bit of a chicken-and-egg problem: you can't verify that a BSP tree traversal routine is correct when the BSP tree used is incorrect with respect to the invariant mentioned above. And you won't know you have a well-built tree until the traversal shows that the tree is correct. So, the whole assignment involves iterating a bit over the different tasks, until everything works correctly.

The framework code contains an initial BSP-construction algorithm. However, this routine does not build a correct tree, it merely assigns polygons arbitrarily to the left and right child nodes for each split, without looking at their position relative to the split plane. See routine **BuildBSPTreeIncorrect** in *bsp.c*.

4.1 Task - BSP traversal

The first task is now to add correctly ordered drawing of the polygons in the tree, based on the current viewpoint. Again, you will not be able to verify with the BSP tree provided by the framework that you implemented this part correctly, but we will need to start somewhere.

Fill in **DrawBSPTree** in *main.c* with a first version of BSP traversal. Use recursion to traverse the tree. Use **DrawPolygon** to draw polygons.

Run the program to make sure you don't have weird crashes related to incorrect pointers, etc.

4.2 Task - Improved BSP construction

The second task is to write a correct version of the BSP construction algorithm.

At the end of **InitializePolygonlists** in *main.c* replace the call to **BuildBSPTreeIncorrect** with one to **BuildBSPTree**. Then, implement **BuildBSPTree** in *bsp.c*. Copying the body of **BuildBSPTreeIncorrect** is a good start, *but remember to update the recursive calls!* Then add the following:

- Classify polygon vertices as either "left of", "right of" or "on" the split plane being used. For the "on" classification use a threshold of 10^{-3} for the in-product test. The reason we use a safety margin is that polygons might share vertices with each other and chances are high those vertices will not get classified as exactly "on" the plane (due to floating-point errors), leading to unnecessary splitting.

- Based on the vertex classifications you can classify the polygon itself. For now, you can ignore polygons that straddle the split plane. Simply assign these problem polygons to either left or right child.

Test the resulting program. The walls and windows of the house should now be correctly rendered w.r.t. transparency, as a BSP tree can be constructed for the transparent polygons in the scene without needing any splits.

4.3 Task - Adding polygon splitting

In **InitializePolygonlists** in *main.c* uncomment the loading of the "tree" object, then compile and run the program. Note that this object consists of intersecting geometry and so can't be correctly handled during BSP tree construction as you have currently implemented. We will need to add splitting of geometry to the construction algorithm.

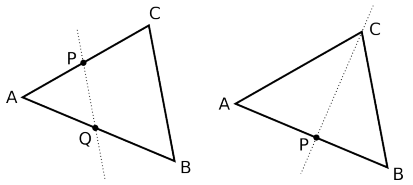

Figure 3: The two ways in which a triangle can get split by a plane.

The framework already makes sure that all (semi-)transparent geometry loaded from the .obj files used is converted to triangles. This makes splitting during BSP construction much much easier, as a triangle intersected by a plane can only get split in two ways. For general polygons there are many more situations, which we don't want to have to handle. Note that any geometry created as a result of splitting during BSP construction will need to consist of triangles only!

See Figure 3 for the two ways in which a triangle can get split by a plane. Implement function **split_triangle** in *bsp.c*, first determining which case applies for the given triangle, followed by calling the right split function. The left case in Figure 3 should be handled by **split_triangle_two_edge_intersections** in *bsp.c*, the right case by **split_triangle_one_edge_intersection**. Implement both these functions as well. Rework your implementation of **BuildBSPTree** so that it calls **split_triangle** to handle triangle splitting. Note that the *side* argument the latter function takes can also be of use in **BuildBSPTree**, when classifying triangles based on the split plane.

You can use **check_for_intersection** to determine if and where a line segment (e.g. an edge) intersects a split plane. As the geometry being split might be textured you will need to set correct texture coordinates on new triangles (using linear interpolation for new edge points such as P and Q).

Any new triangles that are created should receive the correct normal vector.

You can make a copy of a polygon by simply assigning it to a local variable, e.g. **poly p_copy = p;**.

4.4 Task - The final test

Finally, uncomment the fish bowl (+ gold fish) and fence objects in **InitializePolygonlists**. These objects will really test your construction and traversal algorithm for correctness[4]. If you implemented the BSP algorithms correctly the scene will be rendered with correct transparency from all viewpoints. If this is not the case go back over the previous steps to debug them and correct any mistakes left.

5 Grading

A correct BSP traversal routine is worth 2 points. Correct classification and sorting of polygons during BSP construction (including maintaining the invariant) is worth 2 points. The two routines handling triangle split cases are worth 2 points each, when implemented correctly. A correct implementation of **split_triangle** is worth 1 point.

You can get one additional point for writing clean, well-structured and well-commented code (on the opposite side, unreadable or overly complex code might cost you a point).

[4] Actually, there's a semi-transparent cow as well, if you're brave...