Growing a Forest

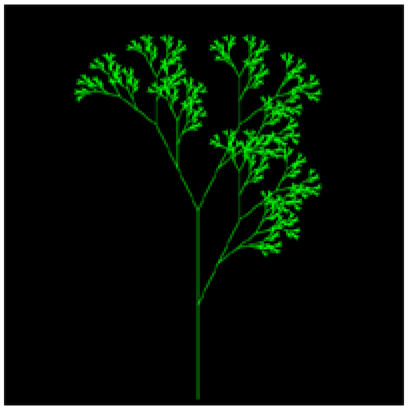
Author: Nathan Thompson

The idea

When I first starting thinking about the subject for my final project I wanted to create something that was simple in concept, but that employed a number of the more advanced OpenGL concepts that I had heard about. I followed this format because there were many topics that interested me, but I didn't know how many I would have time to implement. First I wanted to try using procedural geometry using L-systems, so I started out by trying to think of something interesting that L-systems would create. This is where I came up with the idea to create a simple forest of trees.

I set out designing many different tree-like L-systems. I needed the L-system for my tree to be fairly simple so that when I drew many of them, my performance didn't suffer. I discovered during my experimentation that many L-systems that actually resembled what I considered normal looking trees were too complex for my needs. I considered many simple L-systems, I found many that looked very sparse like figure 1.

Figure 1

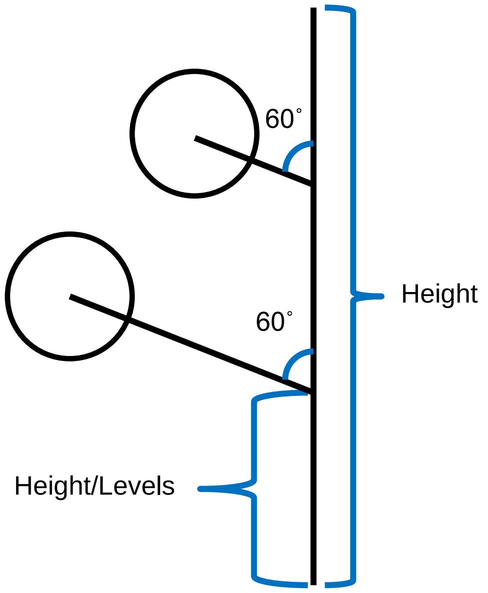


I thought I could add bushels of leaves to the endpoints of the branches in the form of a textured sphere to make the tree look more full, but what that produces is a lot of overlapped geometry which takes a lot of performance to render while not added any visual enhancement to the scene. Secondly I didn't like a lot of the abnormal looking trees that were produced so I decided against using L-systems to create my trees.

The Tree

I went back to the drawing board and decided to geometrically define a simple looking tree. I wanted the tree to have varying levels of branches that were equidistant from each other and I wanted the branching factor for any level of branches to be linearly proportional to its level number. The diagrams below show my final design (note that the angles in the images may not be accurately drawn).

Figure 2



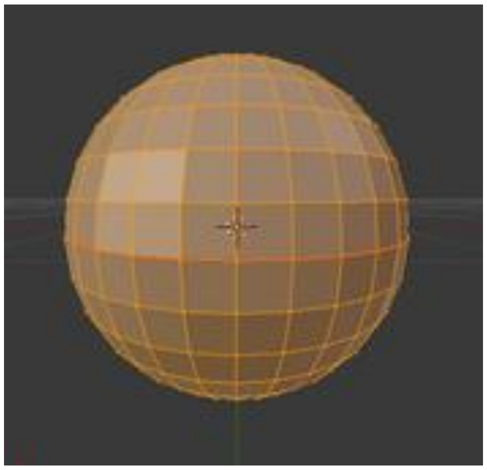
I experimented with a few different branching factors and decided that the number of branches at any level should be three times its level number. The resulting design created an aesthetically pleasing tree for levels two through five, but for higher levels of trees it produces somewhat strange looking trees. In order to reduce my final polygon count I wanted to limit the maximum number of levels to about five, but wouldn't determine my actually maximum until I was nearly finished with the project. In the end, each of the trees in my forest would have a random number of levels between two and five inclusive.

Once I had this basic design I needed to decide how to define the final geometry. I knew that the bushels of leaves would be spheres. These bushels are represented as circles in

Figure 2. I decided that to simplify my work I would make the branches as hemispheres that were scaled in the Y direction.

I started by creating an approximation of a sphere by subdividing a tetrahedron but the result would be difficult to divide in half for my hemisphere and it seemed like it might be complicated to apply textures to it so I did some research and adopted the UV Sphere. To create a UV Sphere you start with a vector of length R along a chosen axis and create a vertex, I used the x-axis. You then rotate the vector along the z-axis by angle theta which is some integer that is a divisor of 360 degrees. You create a new vertex at the end of this vector. Continue doing this until you have the approximation of an equally divided circle. Then take this circle and rotate it around the y-axis by the same theta degrees and create a new circle approximation. Continue doing this until you have an approximation of an equally divided sphere. An example of the resulting sphere approximation is shown in Figure 3.

Figure 3



The faces at the top and bottom are triangles and admittedly, adding a good looking texture to that area is difficult. I did the best I could, but counted on the fact that most people would not be focused on the top or bottom of my trees. Because I was careful in deciding the number of angular subdivisions, the rest of my spheres can be unwrapped to form a rectangular grid of square faces that has a width that is twice its height. This made texture mapping that part of the sphere extremely simple.

Giving the illusion of texture

I wanted to experiment with bump mapping, but upon having researched it decided that it seemed easier to employ normal mapping. Normal mapping takes a color image and for any given point on that image the XYZ values of the normal will be in proportional to the RGB values of the image. Generally any value in the RGB values of a normal map range from 0.0 to 1.0 but the blue values tend to be in the range 0.5 to 1.0 so that all normals continue to point away from the face. This is why most normal maps tend to have a blue tint. My spheres used for leaf bushels use a leaf texture and a leaf normal map, but I found that these spheres didn't portray the effects of normal mapping very well so for my hemispheres I used a solid color for the texture and a normal map that resembled tree bark. Figure 4 shows the texture and normal map I used and the resulting hemisphere is shown in Figure 5.

Figure 4

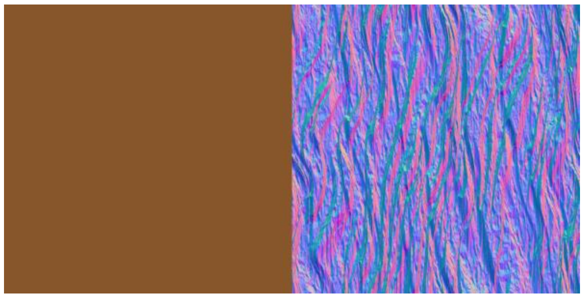
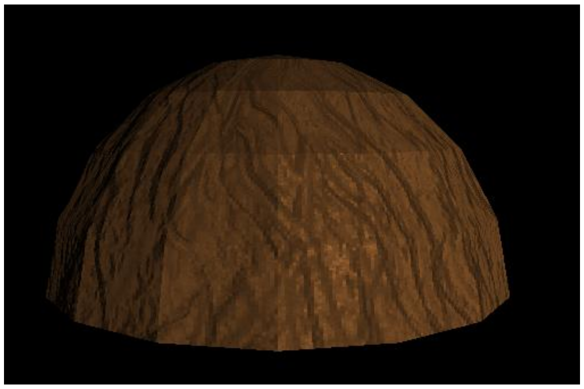


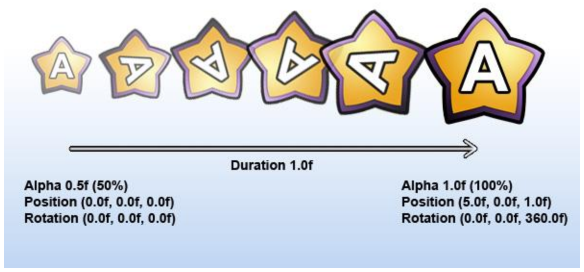
Figure 5



Making it grow

I have always been interested in 3D animation and wanted to employ keyframe animation in my project. The concept of keyframe animation is fairly simple to understand. You can create an animated scene by rapidly taking pictures of an event at regular intervals. You can then play the animation by displaying those pictures on a device at those same intervals. For the purposes of my project, the pictures in my example are called frames and the interval is the framerate. In 3D animation you have a special subset of frames called keyframes. At the keyframe at time x you define a specific set of translations, rotations, and scaling for each object in your scene. At the keyframe at time y where y > x and y-x is greater than the time for one interval of your animation you specify another set of translations, rotations, and scaling for the objects in your scene. For each frame in your animation between the keyframes at times x and y you interpolate the values of your translations, rotations, and scaling proportional to the amount of time that has passed since time x. The result is illustrated in Figure 6

Figure 6


Duration 1.0f
Alpha 0.5f (50%)
Position (0.0f, 0.0f, 0.0f)
Rotation (0.0f, 0.0f, 0.0f)
Alpha 1.0f (100%)
Position (5.0f, 0.0f, 1.0f)
Rotation (0.0f, 0.0f, 360.0f)

My trees begin as a hemisphere that represents the tree's trunk. The hemisphere is stretched in the y direction until it reaches its full height. When the trunk reaches height full_height/levels it spawns a new level of branches. The branches will reach their full length at the same time that the trunk reaches its full height which I did for simplicity. When the trunk and branches reach their full length the tree grows a bushel of leaves at the end of each branch and the end of the trunk.

Complications

While I succeeded in creating a growing forest with textures and normal mapping, there are a number of problems that I faced in the process. The first was in my general code organization. I was still in the process of learning OpenGL when I started this project and my knowledge of C++ was a bit rusty when I first began this project. This project started as a simple codebase that has been added upon like adding body parts to Frankenstein's monster. Unlike the monster we are familiar with, as I continued adding more body parts, I ended up with a monstrous mass of metaphorical limbs that were sewn together into a beast that no longer resembled any organized creature. This wasn't an unforeseen complication. Many students of my Computer Graphics course at USU focused their projects on creating an organized graphics programming framework. I was more interested in putting Computer Graphics theory into practice than in creating an organized framework for graphics programming and it shows. My resulting program shows a very beautiful scene and implements some very interesting computer graphics concepts. My resulting code, on the other hand, is somewhat unorganized.

The second complication arose when I was implementing normal mapping for the hemispheres. As the hemispheres are scaled mostly in the local y direction, so are their textures. Because the final y scale is so much greater than the x and z scales the final normal map looks very skewed and no longer resembles tree bark accurately. Sadly, this is just the nature of such radical scaling and while I could have compensated by altering my textures to have a height that was proportional to the scaling, but this would have increased my texture size and the textures would have looked squished when the hemisphere was at its original state at the beginning of the growth animation's scaling. As a result I decided against it. What I wanted was a way to preserve the look of the texture as the sphere was scaled, but I found no way to do this.

The third complication I found was in the processing time for my program. When I ran my program on my development machine I could create 100 simultaneously animating trees without a problem. If I tried to do more than that then my performance suffered. This was partially due to my lack of optimization. Each tree has a set of world translation, rotation and scaling that is randomly generated. Each branch and bushel of a tree has a set of translations, rotations, and scales relative to the tree's position. Also defined for each branch and bushel is the interpolated animation transformation matrix. A translation and scale each require one matrix and a rotation requires three matrices, one for a rotation in each of the x, y, and z directions.

Consequently for any object I needed to multiply 15 4x4 matrices before applying any camera and projection transformations. These multiplications were done on the CPU for each frame because passing that many matrices to the shader seemed unwise. A tree has 28 objects on average which means that a total of 420 matrix multiplications must be done for each tree on the CPU each frame. This gets very costly with a large number of trees.

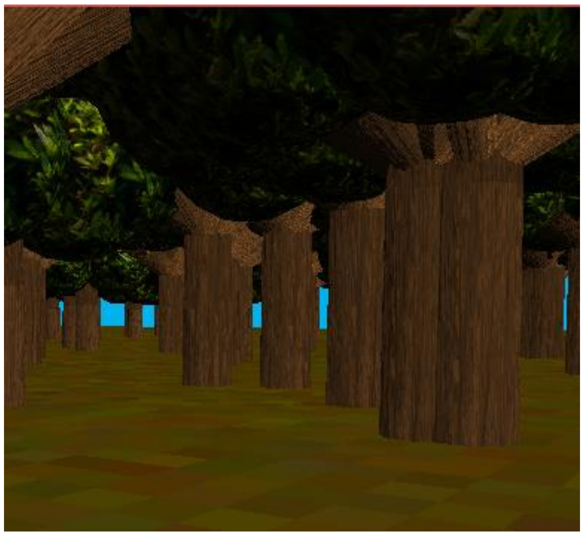
I could have greatly lowered this number by observing common behaviors in my tree. The transformations not involved in animation should have been stored for each object when I actually stored them as two separate matrices and multiplied them each frame. The first matrix is the tree's global transformation matrix. The second matrix for any object in that tree is the local transformation matrix for that object. If I multiplied those together and saved them in the original object I could have severely reduced the number of matrix multiplications I needed to do each frame.

This is an unfortunate drawback to animation. Even with the best optimization, you still need to do a lot of calculation to animate an object. Because these optimizations would only increase the number of trees I can draw and because I found that my program could make a good forest without the optimization and because the amount of work it would take to apply this optimization was not trivial, I didn't implement this optimization.

While not really an issue for me, I could have significantly reduced the amount of computer memory required for my project. There are many animations that are the same for objects in a tree. Each branch in a level of branches has the same animation: scale in the y-direction to scale j. Each bushel of leaves has the same animation. If I consolidated their animation matrices into one, I could have saved a considerable amount of memory, but because these animation matrices still need to be multiplied by each object's other transformation matrices, I do not save much performance by employing this optimization.

The last complication I faced was in my tree placement. While I don't mind if branches intertwine between trees, it is unrealistic to have two trees so close together that their trunks are on top of each other. The result is a "Siamese twin tree" as shown on the rights side of Figure 7.

Figure 7



I could have fixed this problem by implementing a collision detection algorithm between tree trunks, but that would have taken no small amount of work and would have increased my processing time. In a production environment, I would have made this optimization, but the purpose of this project was to test my knowledge of OpenGL and 3D graphics concepts. Collision detection is not directly related to graphics programming so I didn't find it worth the effort.

Conclusion

I am proud of the results of my forest program. To tell the truth I didn't think I would have time to implement keyframe animation and normal mapping in the given time period. The resulting scene as shown below is a good approximation of a forest in my opinion. The animations worked as perfectly as I could have hoped. The normal mapping works just fine, but is somewhat wasted on this scene. Between the skewing caused by the animation transformations and the fact that the normal maps do very little for the forest when you are looking at the entire forest from a distance, it is hard to visually understand the power of normal mapping.

To tell the truth I was worried that my computer wouldn't be able to animate an entire forest efficiently, but I am happy to say that it does an excellent job. I am particularly proud of how good the scene looks despite the fact that the only models that I define for this scene are a sphere, a hemisphere and a plane (used for the ground). I could go on, but in this case I believe a picture does say a thousand words.

Figure 8