CSCI 441 - Lab 09
Friday, October 30, 2015

LAB IS DUE BY **FRIDAY NOVEMBER 06 11:59 PM**!!

Today, we'll be using sprites as a static particle-type system. These sprites will be represented by a point but actually be a fully textured quad! We'll then apply billboarding to make our scene appear to be more in-depth than it really is.

Please answer the questions as you go inside your README.txt file.

Step 1 – Growing A Single Tree

You are able to type make right away and see a (pretty) scene. The loading of textures is provided for you. Be sure to take a look out how the transparent texture is loaded and registered into OpenGL.

Our trees will be sprites. The sprite is represented by a single point, which can think of like being a particle (but there is no particle system or rules to follow). We will then render a textured quad at this point location.

There is already a `vector` of `Points` available for you to use to store all these sprite locations. `TODO #1` will generate semi-random locations for all of our sprites. The location of the sprite will correspond to a grid location. Our grid lies in the XZ plane so our sprite location will follow the format (*x*, 0, *z*). Use the following equations/formulae to determine appropriate values for x and z:

- X: random value between (–`n`/2, `n`/2) where `n` is the total number of sprites
- Z: `i` – `n`/2 where `i` is the sprite index and `n` is the total number of sprites

Initially, `n` will equal 1 for our case but later we'll increase 1 to be `n`.

Next, we need to actually draw our sprite. TODO #2 is already inside a loop to go through all our sprites. For each sprite location, we want to draw a textured quad at that location. Use the method `drawTexturedSprite( width, height )` to handle the drawing of the quad. Use a width and height of both 20. Be sure to look at the `drawTexturedSprite()` method as you will need information from it later.

Compile and run.

Q1: You should now see a lone tree that you can rotate around and see through the lies that it's really a flat object. Ya?

Well let's fool the eyes and rotate our sprite so it appears to be much fuller than it is.

Step 2 – Rotate the Tree (i.e. Billboarding)

And now for the magic to happen through the power of math! Billboarding simply rotates our quad so that it faces the camera. In one sentence, billboarding must:

> Rotate a quad so that its surface normal is aligned with the camera vector.

This leads to a series of questions:

1. What is the surface normal (vector) of our quad?
2. What is the camera vector?
3. How do we rotate one vector so that it is aligned with another vector?

For #1, look at how our textured quad is drawn. What is the surface normal for these four points?

For #2, the camera vector is the opposite of our view vector. Question 2a: how do we compute the view vector? This is the vector the camera is looking along. Thinking about how camera matrix is generated from the parameters of gluLookAt(), how can we compute the view vector?

Well we've seen #3 before, it was a question on Exam I. We now need to rotate our surface normal some angle around some axis so that it is pointing towards the camera.

At `TODO #3`, compute the angle to rotate through and axis to rotate around. Don't forget to work with normalized vectors, it makes the math fairies happy (and these particular math fairies happen to live in these trees we are drawing). Then before drawing each quad, make sure you rotate using the values you just computed (using a single `glRotatef()` call). (Be aware of order of operations. OOO!)

Compile and run.

Q2: How does your tree look as you rotate around it?

Hmm, not quite right is it. Well, we don't want our surface normal to actually point ***at*** the camera. A tree should always be rooted to the ground and should therefore rotate around its local Y axis. We can accomplish this by aligning our surface normal with the XZ-components of the view vector. After you have computed the view vector, set the Y-component to zero and renormalize.

Compile and run again.

Q3: Does your tree properly rotate now?

Fantastic, let's add some more trees.

Step 3 – Grow & Rotate a Forest

Back near `TODO #1`, change the number of sprites to 50.

Compile and run.

Q4: Do you now see a forest of trees all spinning around their local axis?
Q5: Have you lost the transparency?

Recall that OpenGL places fragments into the depth buffer based on the order they are drawn. Blending for transparency blends the current fragment with the value in the frame buffer currently. So if a fragment fails the depth test, it doesn't blend in. We therefore need to draw all of our objects back to front so that blending takes place properly.

Step 4 – Sort the Forest

At `TODO #4`, you will need to sort the vector of tree sprites. Use your favorite sorting algorithm. Sort in place, sort to another vector. As long as you have a sorted vector, you are good to go. But what is our sorting criteria?

We want to sort them by distance to the camera. But like our rotation this is not the straight line distance to our camera. It is the distance to our camera projected onto the view vector.

We get this value by using a Point equal to our Camera position with the Y-coordinate set to 0. We'll call this point XZ-Camera and use it to compute our view vector projected onto the XZ plane (Origin – XZ-Camera). Now we can compute a vector from each sprite point to this XZ-Camera point, we'll call this spriteVector. Finally, we compute the projection of our spriteVector onto the view vector we just computed. Luckily for us, the dot product does this calculation for us. To simplify this paragraph, the distance for each sprite is the dot product of our spriteVector with the viewVector. Now sort by distance, greatest to smallest.

If you didn't sort in place, be sure to update the `for` loop for drawing all the sprites.

Compile and run.

Q6: Do you have a happy fairy filled forest now as your camera rotates?

Great work. Now transfer this knowledge to the Assignment. Go get them Particle Man! And answer the questions on the next page.

Q7: Was this lab fun? 1-10 (1 least fun, 10 most fun)
Q8: How was the write-up for the lab? Too much hand holding? Too thorough? Too vague? Just right?
Q9: How long did this lab take you?
Q10: Any other comments?

To submit this lab, zip together your source code, Makefile, screenshot, and README.txt with questions. Name the zip file <HeroName>_L9.zip. Upload this on to Blackboard under the Lab09 section.

LAB IS DUE BY **FRIDAY NOVEMBER 06 11:59 PM**!!