CSCI 441 - Lab 02

Friday, September 8, 2017

LAB IS DUE BY **FRIDAY SEPTEMBER 15 11:59 PM**!! THIS IS AN INDIVIDUAL LAB!

In today's lab, we will jump into a 3D world and fly around. First, copy the contents of the include folder into your Z:/CSCI441/include folder. This contains the class library files to draw 3D Objects. These are header only libraries so they will work on any operating system. Hooray!

Part 1 – Our New Code Structure

Now that we've moved to 3D, there are some new functions we need to check out.

First find our draw loop. After clearing the buffers we set up our viewport (how much of the window we draw to) and our projection matrix. Notice we need to switch matrix modes, load the identity, and then set up our perspective projection.

Next we need to do is make sure we are working with the Modelview Matrix. `glm::lookAt()` handles positioning our camera.

In our `setupOpenGL()` function, we enable depth testing and set up our shading model. There is some additional code to add a default light to our scene. You do not need to worry about what this does or how it does it (yet). We'll get to lighting in a couple weeks but for now, be aware that it exists.

Let's start drawing!

Part 2 – Creating a Static Scene with a Display List

Before we do anything, compile and run our code. It should launch with no problem.

What do you currently see? Well, let's create a more complex 3D scene.

You may have seen in the `setupScene()` function, a call to `generateEnvironmentDL()`. Previously, we have been operating OpenGL in **immediate mode** where primitives are passed through the pipeline as soon as they are specified. For this lab, we are creating a static scene and our objects will never move. We can therefore improve our rendering performance by telling the GPU what to draw and how to draw it only once. The GPU will store the final result of what is drawn and we can then reference our scene from memory when it is needed. This rendering mode is called **retained mode**. We do this through use of display lists. Let's create a display list.

One of our global variables is

```
GLuint environmentDL
```

We will use this to store the handle, or identifier, to our display list. All of drawing code will go inside the `generateEnvironmentDL()` function at `TODO #1`. We need to first request a valid display list handle from OpenGL and assign this value to our global variable. The command to do so is:

```
glGenLists( GLsizei n )
```

where n is the number of handles to request and it returns the first handle – in this case n should be 1. Now we're ready to start drawing.

When we are working with 2D primitives, we needed to tell OpenGL to begin and end. We do the same with display lists. To tell OpenGL to begin a display list, add the line

```
glNewList( environmentDL, GL_COMPILE );
```

This tells OpenGL to begin storing all upcoming draw commands and "compile" them on the GPU. It does not display them yet, just passes them through all the necessary transformations needed. When we are done drawing, we'll add the line

```
glEndList();
```

Find where our teapot is drawn in the `renderScene()` function. Cut and paste those lines of code in between our new list and end list calls.

Compile and run the code. How's our teapot look now?

Well, there's nothing to see because we haven't actually told OpenGL to display anything yet. We've only told it to how to render our scene, not where to render it. At `TODO #2` in `renderScene()`, let's tell OpenGL to execute our display list. We do this by calling

```
glCallList( environmentDL );
```

By passing our display list handle to the function `glCallList()`.

Compile, run, voila!

Is the teapot back? Great, let's get rid of it. Delete the lines of code in our display list where the teapot is drawn. Instead, let's call two functions we have the shells of: `drawCity()` and `drawGrid()`. Add calls to those between our new list and end list lines. Let's start by drawing a grid first.

Find `TODO #3`. You need to draw a grid of lines using 2D primitives. Take note of the comments, again we'll discuss why later, but whenever we use OpenGL line primitives we need to turn the lighting off. We must then reenable it for drawing 3D objects. Ok, so at TODO #3 create a series of white lines in the XZ-Plane to create a grid. HINT: You'll want to use nested for loops and they each should range from -50 to 50. And it should only take 11 lines of code so shouldn't be overly complex.

(Though if you are particularly savvy, you can do it with one for loop and 9 lines). One set of lines will be running parallel to the x-axis while another set will be running parallel to the z-axis.

Compile and run.

If you are getting hung up on this step, do not spend too much time on it. You will need to add a grid for your Assignment3 so can focus on it later.

Let's continue on and create our city. We'll add to `TODO #4` in the `drawCity()` function. We want to iterate over our grid locations (hint hint, double for loop again with same parameter range) and randomly place buildings. We have the function `getRand()` which was written to return a random value between 0.0f and 1.0f. Each time through the loop, check if both our indices are even and if a random value is below a threshold of 0.4. If this is true, then draw a solid unit cube with a random color at the current grid location.

Compile, run.

This is a start, but these look like buildings for ants! We need them at least three times as large. Assign each building a random height between 1 and 10. Once you computed the height, scale our cube along Y to our new height.

Compile & run.

The cube gets drawn through its COM (center of mass). Before we scale, we need to translate our cube vertically by half its future height. Now when we scale, it will extend back down to the grid.

Great! Now we have a random colorful city on a network of roads. Let's get flying!

Part 3 – Creating the Free Cam

To create our flight simulator, we need to make a free cam that can fly through our city. We have a number of global variables already declared that we'll use for our camera (they should be clearly labeled). First, we need to do a spherical to Cartesian conversion. At TODO #5 is our `recomputeOrientation()` method. Set the Cartesian direction vector based on spherical coordinates (the two angles). Don't forget to normalize your vector!

Next, let's get our camera into position. `TODO #6` is at our `glm::lookAt()` call. Change the parameters so our camera is now positioned at its proper XYZ location and is oriented to the look at point correctly. Recall: a Free Cam is oriented along its direction of view vector. We just need some simple vector math here.

Compile & run.

Let's start flying now! We'll have the 'w' and 's' keys control our flight. When the 'w' key is pressed, we need to move our camera (or change its position) one step along our direction vector – again basic vector math. Likewise, when the 's' key is pressed we'll take one step backwards along our direction. (OH NO! No `TODO` hint, where does this go?) Be sure to choose an appropriate step size.

Compile, run, start pressing 'w' and 's'.

Well, we can sort of move if we continually press 'w' or 's' over and over again. But it'd be nicer if the user can press and hold 'w' and the plane flies forward. When we check if 'w' or 's' is pressed, our keys can be in a third state beyond press or release. This third state is repeat. Check if the key is pressed or GLFW_REPEAT.

Compile, run, and hold 'w' or 's'.

(How does it feel to zoom through space? What step size did you choose to not have all passengers take out their barf bags?)

The only thing left is to be able to turn our plane along a new heading by changing its pitch and yaw (or phi and theta). This will be tied to the mouse by Left clicking and dragging. We're already set up to check if the left mouse button is held down while we're dragging the mouse. To change our angle, we need to measure the change in our mouse position. Hooray! More global variables.

Let's change our yaw, or theta, first. This will be tied to any change in the mouse's X position. In our active motion function, we can take the difference between the current mouse position and our last stored position. This will measure the change in pixels. Since we want to change our angle in radians, we'll probably want to scale this change by some value (such as 0.005). Now that we've added or subtracted some value to theta, we can recompute the orientation and redraw. Those two steps are already there.

Compile & Run.

Ok, one final step. We need to do what we just did for X and theta with Y and phi. We don't want "inverted flight controls" (i.e. pulling back on the yoke makes the plane go up). When the mouse goes towards the top of the screen, we want the plane to pitch upwards. Finally, we don't want our plane flying upside-down, so we need to check the limits on phi and make sure it stays within the range of (0, M_PI).

Compile, run, and fly around!

Q1: Can you navigate in between two buildings? Submit a screenshot of your aviation expertise.

Congrats on implementing the free cam! For the next assignment, you will need to implement an arcball cam. You should be able to use this lab as the starting point for your assignment. You will need to make some modifications to how your camera's position and look at point are computed. Don't forget you can now zoom in and out with an arcball cam so will need to incorporate a camera radius.

Q2: Was this lab fun? 1-10 (1 least fun, 10 most fun)
Q3: How was the write-up for the lab? Too much hand holding? Too thorough? Too vague? Just right?
Q4: How long did this lab take you?
Q5: Any other comments?

To submit this lab, zip together your main.cpp, Makefile, screenshot, and README.txt with questions. Name the zip file <HeroName>_L02.zip. Upload this on to Blackboard under the L02 section.

LAB IS DUE BY **FRIDAY SEPTEMBER 15 11:59 PM**!! THIS IS AN INDIVIDUAL LAB!