CSCI 441 - Lab 02
Friday, September 06, 2024

LAB IS DUE BY **FRIDAY SEPTEMBER 13 11:59 PM**!! THIS IS AN INDIVIDUAL LAB!

In today's lab, we will jump into a 3D world and fly around.

Part 0 – Our New Code Structure

Our class library is beginning to expand and using an object-oriented approach. Copy the `include/CSCI441/Camera.hpp` file with the remainder of the `include/CSCI441` library on your machine. This is an abstract Camera class that we will make concreate implementations of to implement a FreeCam and ArcBall model.

Now that we've moved to 3D, there are some new functions we need to check out. Our class library is now making use of the `CSCI441::SimpleShader3` namespace to be working in 3-dimensions. This has the same interface as `CSCI441::SimpleShader2` but with expanded functionality.

In our `mSetupOpenGL()` function, we enable depth testing.

In our `mSetupScene()` function, 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.

Next find our draw loop in `run()`. After clearing both buffers we set up our viewport (how much of the window we draw to) then we set up our projection matrix. NOTE: the Field of View is in degrees.

Next we position our camera by creating a view matrix. In class Monday we will look at the composition of the view matrix.

Great! We're ready. Let's start drawing!

Part 1 – We Built This City

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

What do you currently see? Just a 2D grid that is in the XZ-plane. Oh, and there's a teapot in the center of our world. Well, let's create a more complex 3D scene. Begin by deleting the teapot out of `_renderScene()`, it'll just get in our way as we move forward (figuratively and literally).

You may have seen in the `mSetupBuffers()` function, a call to `_generateEnvironment()`. This is where the grid is being created and where we will generate all of the building information.

At `TODO #1` we will create our city. We want to iterate over our grid locations. Use a nested double for loop ranging from the `LEFT_END_POINT` to `RIGHT_END_POINT` and `BOTTOM_END_POINT` to `TOP_END_POINT`, stepping by the `GRID_SPACING WIDTH` and `LENGTH`. (While the constants are stored as floats for flexibility drawing our grid, make the looping parameters integers to simplify the next step). We have the function `gen_rand_GLfloat()` which was written to return a random value between 0.0f and 1.0f. Each time through the innermost loop, check if both our indices are even and if a random value is below a threshold of 0.4f. If this is all true, then we'll create a building.

We are not actually creating the building here, but rather the attributes that make up a building. The attributes that will describe a building are its location and color. See the `BuildingData` struct definition inside the header file. Go to the comment labeled `LOOK HERE #1`. This is where our buildings are being drawn. We are setting its color and then pushing a transformation to position the building. Our task inside `_generateEnvironment` is to set the `color` and `modelMatrix` that positions each building.

First create a translation matrix that will position the building on the grid at the corresponding `i j` location. Remember that the grid is in the XZ-plane, so pass the location accordingly to our translation matrix.

Next, generate a `vec3` to represent the color with each RGB component set to a random value between `0.0` and `1.0`.

Finally make an instance of our `BuildingData` struct, assign each component, and push it on to the `_buildings` vector that was already created for you.

Compile, run.

This is a start, but these look like buildings for ants! We need them at least three times as large. Compute a random height between 1.0f and 10.0f after making the translation matrix. Once you have computed the height, create a scale matrix to scale our cube along Y by our new height. In previous labs/assignments, we pushed each transformation individually. For these buildings, we're only able to apply a single transformation. Therefore, we'll need to create the final `modelMatrix` that has all the transformations concatenated together. Since matrix multiplication occurs from right to left, we need to make sure our order of operations is correct.

```
glm::mat4 modelMatrix = scaleMatrix * translateMatrix;
```

Compile & run.

The cube gets drawn through its COM (center of mass). We need to translate our cube vertically by half its height. When we apply the scale, it will extend back down to the grid. Create a second translation matrix to accomplish this transformation and be sure to apply it in the correct order of operations.

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

Part 2 – Creating the Free Cam

To create our flight simulator, we need to make a free cam that can fly through our city. We have the beginning structure to put everything in place. Begin by looking at what the abstract `CSCI441::Camera` class provides. We have the necessary camera parameters created as data members (theta, phi, radius + position, direction, look at point, up vector). First, we need to do a Spherical to Cartesian conversion as laid out in the video slides. At `TODO #2` 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. `CSCI441::Camera::computeViewMatrix()` handles our `glm::lookAt()` call. At `TODO #3` we need to update the look at point so our camera is now oriented at the proper point correctly and then recompute the view matrix (use the function call mentioned at the

start of this paragraph). *Recall: a Free Cam is oriented along its direction of view vector. We just need some vector math here.*

Compile & run. The view should look the same as before. If yes, then delete the line referenced by `TODO #4`. That was present to do some initial setup before you had the implementation in place.

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 vector math. Likewise, when the `'s'` key is pressed we'll take one step backwards along our direction. Begin by putting in the implementations for `moveForward()` and `moveBackward()`. After updating the camera position, don't forget to update the look at point and recompute the view matrix. Once these functions are implemented, be sure to call them at the appropriate places. (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 held (`GLFW_REPEAT`).

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

Note & Hint: depending on your OS/Hardware you may notice an initial delay between the first movement step and then the continuous movement. There is a delay in which the repeated events are called when the key is held down. This implementation only allows a single key to be held down. For the assignment, you will want the ability for multiple keys to be held down. To perform that task, you'll only want to check for press/release actions in your event handler and set individual key states in your engine. The update of the movement will not occur in the event handler, but rather in your respective update scene function.

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.

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 denoted by `TODO #5`, we can take the difference between the current mouse position and our last stored mouse 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 by rotating our camera. This step is already there.

Compile & Run.

Ok, one final step at `TODO #6`. 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.

Compile, run, and fly around!

Q1: Submit a screenshot of your aviation expertise to your next website update.

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 a new concreate Camera class and 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 as well.

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 source code and `README.txt` with answers. Name the zip file <HeroName>_L02.zip. Upload this on to Canvas under the L02 section.

LAB IS DUE BY **FRIDAY SEPTEMBER 13 11:59 PM**!! THIS IS AN INDIVIDUAL LAB!