CSCI 441 - Lab 00
Friday, August 28, 2020

LAB IS DUE BY **THURSDAY SEPTEMBER 03 11:59 PM**!! THIS IS AN INDIVIDUAL LAB!

In today's lab, we will learn how to do basic 2D drawing using OpenGL and GLFW. If you're reading this, then you have already downloaded the lab00.zip file from the course schedule page. Congratulations! You completed the first step.

I must start with a warning. Today may be slightly frustrating as we get everything going for the first time. The computers in the campus lab (and your personal machines) are different – they have different CPUs, GPUs, and different versions of OpenGL installed. Additionally, each of you have different PATHs set on your profile – some will point to MinGW and some will point to Cygwin. Have patience and we'll get through it.

In order for anything to appear in our window, we need perform the three steps necessary to draw some primitives. You should be able to easily find where these sections of the code are since they each have a big TODO comment.

You'll quickly realize one of the ugly aspects of OpenGL development and its state-machine-ness: we're going to need lots of global variables. You'll see why next week.

Let's jump right in!

Step -1 (This is only for the lab machines. If your personal machine is already set up, then continue on to the next section)

The first step is to copy some needed files onto your `Z:/` drive. Begin by creating a folder at the root of the `Z:/` drive called "`CSCI441`". Copy the `include/` and `lib/` folders from this lab00 package into your new `Z:/CSCI441` folder. This copies over the needed OpenGL, GLFW, GLEW, and glm library files to run your program.

Step 0 (This is for all machines)

Be sure to copy the `include/CSCI441` folder to either the `Z:/` drive folder or to the same place you installed all of the other libraries. This is our class library and we'll expand it with future labs to provide additional functionality.

Lab00A – Raise the Triangles

Open CLion and then open the folder named "lab00a." This folder contains the framework for your first OpenGL program. There are two files "main.cpp" and "CMakeLists.txt" as well as another cmake folder that CLion uses.

The CMakeLists.txt file contains the information necessary to compile and link your application. It is currently configured to work in the lab environment. If you are anywhere else, then read through the comments to make the following potential adjustments:

- If you installed the OpenGL/GLFW/GLEW/glm libraries somewhere that are not on your path, then update the `include_directories` and `target_link_directories` folders to point to the locations on your machine.
- If you are running on OS X, then comment out the Windows specific line and then uncomment the OS X specific line.
- If you are running Linux, then use the Windows line but you will likely need to change the library names to match your installation. Please share on Piazza your distro and the corresponding libraries to build. This will vary.

For all of our projects (regardless of OS), we will need to edit the working directory within CLion. Go to Run > Edit Configurations and set the working directory to be the parent directory `..` This will be necessary when we get to File I/O very soon, but also for the Windows lab machines to have `glew32.dll` on the execution path. *Note: If you are on Windows on your personal machine, then you likely need to delete the `glew32.dll` included with this distro since it may conflict with the version you have installed.*

main.cpp will contain all the code for this exercise. Press the green arrow to run your program, you'll see two g++ calls (one that compiles and one that links). If everything is set up properly, you will see an empty window with the title "Lab00A". Right now, we have to click the X to close our window. We will learn better ways to close our window next week.

Go ahead and open main.cpp. Update the comments at the top of the file with your name and info. Find the `setupBuffers()` function. We'll begin here.

Step 1 – Hard Code Everything In

Step 1A

Let's draw our very first (of many many many) triangles. Recall the three steps to draw a triangle:

1. Specify vertex attribute data on CPU RAM
2. Transfer vertex data to GPU
3. Tell GPU which data to draw

These three steps will correlate to the three `TODO`s `1A`, `1B`, and `1C`.

Begin at `TODO 1A` to add the vertex attributes corresponding to a single triangle. A vector of type `glm::vec2` has been created called `triforcePoints`. In our current setup, (0, 0) is located in the lower left corner. Add the following three vertices:

- (100, 100) (200, 100) (150, 180)

(You can use the constructor `glm::vec2(x, y)` to create a two-dimensional point).

Additionally, we will need a specify a color for each vertex as well. A vector of type `glm::vec3` has been created called `triforceColors`. Since the RGB color components are floats, that means they will range from 0.0 to 1.0. Add a nice gold color (hint: try R=0.9, G=0.8, B=0.1) for each vertex.

Step 1B

Now that we have entered all of our data and stored it on the CPU, it's now time to move to TODO 1B and transfer the data to the GPU.

As mentioned, this is actually a multistep process, but our class library handles the process for us (for now – we'll eventually need to do it on our own). The function `CSCI441::SimpleShader2::registerVertexArray()` will handle the process for us.

Pass the two vectors – points first, colors second – to the function. The function then returns the VAOD (vertex array object descriptor) which is the handle to the location on the GPU. There is already a global variable declared for your use. Assign the result of the function call to this variable, `triforceVAO`.

There is a second global variable ready for your use, `numTriforcePoints`, that needs to be set to the number of vertices that were registered on the GPU.

Step 1C

We are now ready to do the actual draw step! This step will go in to our `renderScene()` function. Again, the drawing process is multistep but your class library handles it for us. We'll need to call the `CSCI441::SimpleShader2::draw()` function. This function expects three arguments:

1. The primitive type to display – we want to draw triangles
2. The VAO handle to load – the one we set in Step 1B above
3. The number of points to render – the value we set in Step 1B above

We're now ready to compile and run. You will see a happy little gold triangle staring right back at you. Congrats! You just completed your very first OpenGL program and rendered a triangle.

Alright, let's add two more triangles. Go back and modify the values in Steps 1A, 1B, and 1C. Add the following vertex locations to the three you already have and apply the same gold color for each of them:

- (200, 100) (300, 100) (250, 180)
- (150, 180) (250, 180) (200, 260)

Compile and run. Voila! (Hmm, where have I seen that before…)

(By the way, don't feel bad about your artistic abilities. This is about the extent of mine!)

Step 2 – Encapsulate and Transform

Our approach so far has worked, but needing to know the exact coordinate of every vertex can be cumbersome. To best complete the Step 1 (and for me to put it together) may require graph paper, lots of plotting, and trial and error. A better approach would be to draw an object how we want it to look, and then place it where we want. Previously, the pyramid will always be between the coordinates (100,100) and (300, 260). What if we wanted the pyramid in a different place? We need to compute new coordinates. What if we wanted the pyramid smaller? We need to compute new coordinates. What if we wanted the pyramid rotated? We need to compute new coordinates.

Luckily, we don't need to do that. We can apply **transformations** to manipulate the vertex locations we are providing. We will go into much more detail about this on Monday. But for now, the position, size, and orientation of what we are drawing is stored in a matrix.

We will create all of our objects separately. Each will exist in its own **object space**. The transformation matrix then gets applied to the individual local object and the newly positioned object now exists in a common **world space**. This allows us to create multiple instances of the same object and position them as we like around our world. Again, we will talk about this much more next week but become familiar with this process and these bold terms.

What exactly is a transformation? Anything that manipulates where our objects are located is a transformation. We have three main types of transformations: translations, rotations, and scales. We'll only look at translation in this lab, but you can deduce how the others would work.

Step 2A

In Step 1 we used vectors to store our data since we added more points as we went along. This time, we want to draw one, and only one, triangle. Let's use an array to store the vertex attributes.

Step 2A will mirror Step 1A. Create three vertices with the following location and the same gold color:

- (-50, -50) (50, -50) (0, 30)

Step 2B

Before we can register the data on the GPU, we need to create global variables to track the VAO handle and the number of points corresponding to the VAO. At `TODO 2B`, create complementary variables that existed for Step 1.

Step 2C

Again, mirroring Step 1B, register your triangle data on the GPU. This time the library function will want three arguments:

1. The number of points
2. The position data
3. The color data

Step 2D

Before we get to drawing, let's start thinking ahead. Create a function called `drawMyTriangle` that returns nothing. Inside this function, place the library draw call to draw your single triangle.

Step 2E

In the `renderScene()` method, call your triangle function (after the previous triangles you had drawn). Compile and run the program. You should see a total of 4 triangles.

Can you see the whole triangle? Ok, so we can see 3 triangles and part of a fourth. Well it seems like our initial position was not properly positioned in our window. We could go back and change the triangle coordinates in the function, but we like how the triangle looks as is, in its own object space. Instead, let's just translate where the triangle gets drawn too.

Step 2F

The way OpenGL maintains transformation information is through a matrix. In order to translate our triangle to some position in the window, we first need to compute the transformation matrix that corresponds to this translation. This is accomplished through the GLM library call

```
glm::mat4 tMtx = glm::translate( glm::mat4( 1.0f ),
                                 glm::vec3( float x, float y, float z ) );
```

GLM (OpenGL Mathematics) is a library that performs and simplifies the necessary matrix manipulation. It handles a bunch of the nasty math that we as computer programmers want to avoid.

The above function returns a matrix that represents the calculation to move our object by (x, y, z). The first parameter we will pass is the identity matrix in (using a different matrix allows us to use a previous transformation as a starting point to stack transformations if desired). The next parameter is a vector corresponding to the XYZ location we want to move to.

Wait, Z? We don't have a Z! Well we do, and we'll talk about where it is next week. For now, we can just set Z to zero in our call. This transformation essentially moves the origin in object space to a new position in our world space. Let's move the triangle so we can fully see it in the window. Let's translate the triangle to (300, 300, 0). Create a translation matrix called `transMtx` with the following line, place it in `renderScene()` just before your `drawMyTriangle()` call:

```
glm::mat4 transMtx = glm::translate(glm::mat4(1.0f),
                                    glm::vec3(300.0f, 300.0f, 0.0f));
```

`transMtx` now contains the matrix to move the triangle. We need to tell OpenGL to apply this transformation to future draws. Luckily, our class library abstracts this process for us once again. On the next line after the one just added above, enter:

```
CSCI441::SimpleShader2::pushTransformation( transMtx );
```

What's this doing? We are pushing our transformation on to a stack and applying the product of the stack to every vertex when rendering. Yes this seems a little complex, but again we'll go into the math behind all this on Monday and it will make sense then. Compile and run to test.

As a sanity check, your TODOs 2E and 2F should look like:

```
glm::mat4 transMtx = glm::translate(glm::mat4(1.0f),
                                    glm::vec3(300.0f, 300.0f, 0.0f));
CSCI441::SimpleShader2::pushTransformation( transMtx );
drawMyTriangle();
```

Great, we now see our triangle. But we want something cooler.

Step 3

Create another function called `drawMyPyramid()` which also has a return type of `void` and no parameters. Inside of this function, call your triangle method. In `renderScene()`, replace the call to draw your single triangle to your new pyramid method. Compile and run. You should still see your triangles right where they were.

But this new method should be drawing a pyramid, not a single triangle. We will need to call our triangle function three times to create a pyramid and place each triangle in its proper place. Using the steps above, create a new translation matrix for each of the following (x,y,z) values and apply it to your triangle function call:

- (-25, -25, 0)
- (75, -25, 0)
- (25, 55, 0)

Compile and run. You should three new triangles, briefly then it all goes black. What's happening? Let's think about what happens every time we tell OpenGL to push our translation matrix on to the stack. OpenGL is going to add the translation to any previous translations it has stored. So before we called our method, we told OpenGL to translate to (300, 300). Then we tell it to translate to (-25, -25) which when applied together moves to (275, 275). Then we translate again to (350, 250). This is not what we want. Inside our pyramid method, we want all of our translate calls to be relative to wherever the pyramid is being drawn. We have two choices:

(1) We can compute new translations to calculate the relative position to the previously drawn triangle. But then if we need to move the second triangle, the third gets moved in turn and we'd have to recalculate the third triangle.

(2) We can position each triangle independently and not have them rely on any previous transformations.

We want to do option #2 so that we can place each instance where we want and no two instances are dependent upon the other.

In order to accomplish this, we need to apply our translation and then undo it to move back to where we came from. We could make a new translation that contains the opposite movement (i.e. translate (25, 25)) and push it on to the stack. Or more simply we can apply the inverse of our translation matrix and

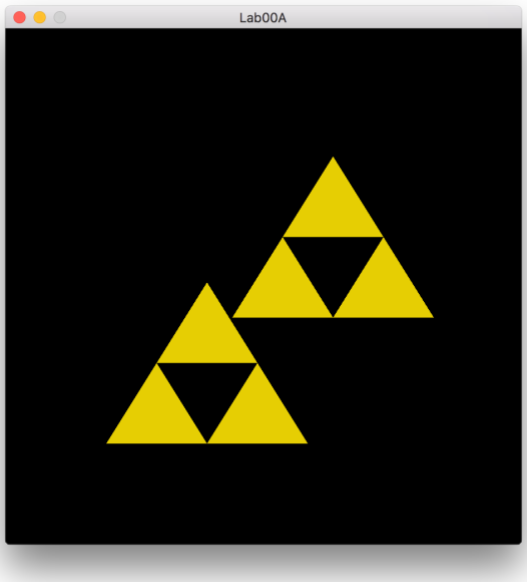
pop it off the stack. After your first triangle call use the library function to pop the last transformation off the stack.

```
CSCI441::SimpleShader2::popTransformation();
```

Your first triangle draw call should now look like:

```
glm::mat4 transMtx1 = glm::translate(glm::mat4(1.0f),
                                     glm::vec3(-25.0f, -25.0f, 0.0f));
CSCI441::SimpleShader2::pushTransformation( transMtx1 );
     drawMyTriangle();
CSCI441::SimpleShader2::popTransformation();
```

This handles making sure that only the triangle sandwiched between the two transformation calls gets moved. Make sure the next two triangles follow the same template. And don't forget about the original call to the drawMyPyramid! Compile, run, and you should have two pyramids that remain static, like below.



Well yea, that's cool. Congratulations! You've finished your first OpenGL program. Let's move on.

Lab00B – Bob Ross Time

Open the folder named "lab00b" and main.cpp. This looks very familiar to lab00a at this point. Every OpenGL program will have the same starting boilerplate (don't forget to edit CMakeLists.txt if necessary). The difference is what we draw and how we draw it.

For this lab you have complete freedom. Try out different primitives, different colors. Experiment with scale and rotate in addition to translate. (Hint: rotate wants the angle in degrees) Scale and rotate will work just like translate does above, just a different glm function call. Now for your task:

> You have a 512x512 window to work with. (0, 0) is in the lower left corner and (511, 511) is in the upper right corner if you had not figured that out just yet. Create what your track will look like. Are there trees and bushes scattered around? A river running through it with a log flume? Mountains that a mine car travels through?

Next week for A2, you will see all of the rides connected together to make a complete Onogoro Island map and you will be able to walk between park sections (a la our popular hero Link). Other students will see your work. Show off your talent and make your hero proud.

SUBMISSION
Be sure to answer these questions in your README when submitting this lab.

Q1: Was this lab fun? 1-10 (1 least fun, 10 most fun)
Q2: How was the writeup for the lab? Too much hand holding? Too thorough? Just right?
Q3: How long did this lab take you?

Be sure to take a screen shot of your final lab00a and lab00b – you'll want these for your webpage. Don't forget a README! This will be much more important later on. See A1 for README details.

Zip up your completed lab00a, lab00b folders and README. Name your zip file - <HeroName>_L00.zip. Submit this zip file through Canvas to L00.

LAB IS DUE BY **THURSDAY SEPTEMBER 03 11:59 PM**!! THIS IS AN INDIVIDUAL LAB!