CSCI 441 - Lab 00
Friday, August 25, 2017

LAB IS DUE BY **WEDNESDAY AUGUST 30 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 will be slightly frustrating as we get everything going for the first time. The computers in this lab 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.

We have already gone over the basic flow of a GLFW program. In order for anything to appear in our window, we must tell OpenGL to draw some primitives. As previously mentioned, any commands in our Display Function (e.g. `renderScene()`) will get drawn to the screen. You should be able to easily find this section of the code as it says "YOUR CODE WILL GO HERE" in a big comment.

For now, this is the only section we will modify. Let's jump right in!

Lab00A

I recommend using Windows PowerShell instead of the standard command prompt. Press the windows key, search for PowerShell, and open it up. It will look like a standard command prompt. The benefit of using PowerShell is that we have many of the same commands and capabilities that we would from a Unix prompt. For instance, you can use "ls" instead of "dir".

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 and GLFW library files to run your program.

Open the folder named "lab00a." This folder contains the framework for your first OpenGL program. There are two files "main.cpp" and "Makefile."

main.cpp will contain all the code for this exercise. The Makefile is what will handle building our application. If you are unfamiliar with Makefiles, then there are plenty of tutorials available online (here is one to read through: http://www.cs.umd.edu/class/fall2002/cmsc214/Tutorial/makefile.html). In a nutshell, there is a special program called "make" that will run whatever is in your Makefile. The contents of the Makefile specify how to compile the program. Type "make" on the command line, you'll see two g++ calls (one that compiles and one that links). You should now have a program called "lab00a." Run this program.

You should 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 using your favorite IDE or text editor (Visual Studio, Eclipse, Notepad++, Sublime, what have you). Update the comments at the top of the file with your name and info. Find the `renderScene()` function and the comment stating "YOUR CODE GOES HERE."

Let's draw our very first (of many many many) triangles. OpenGL is a giant state machine, so to begin drawing we need to tell it to change to triangle drawing mode. To draw any primitive, we first call:

```
glBegin( GL_PRIMITIVE );
```

In our case, the *GL_PRIMITIVE* we are interested in is GL_TRIANGLES. Replace *GL_PRIMITIVE* with GL_TRIANGLES. Next, we need to pass the three vertices of our triangle to OpenGL. We can specify a 2D vertex point with the following call:

```
glVertex2f( float x, float y );
```

In our current setup, (0, 0) is located in the lower left corner. Add the following three vertices:

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

Now we need to tell OpenGL we are done drawing triangles. To stop drawing any primitive, we finally call:

```
glEnd();
```

Go back to the terminal, type make to build our program, and run the executable. Awesome! You should now see a white triangle on your screen. Congratulations! You've just completed your very first graphics program drawing the simplest possible object – a triangle.

Now it would be nice if it wasn't white. Well, we can add color to our primitives! The function call follows the OpenGL style and is:

```
glColor3f( float R, float G, float B );
```

Since the RGB color components are floats, that means they will range from 0.0 to 1.0. Before your triangle call, prior to the `glBegin()`, add a call to set the color to a nice gold color (hint: try R=0.9, G=0.8, B=0.1). This will now change the OpenGL state to a different color, so everything we draw from this point on will have a new color.

Once again, compile and run.

Alright, let's add two more triangles. Use the following vertices:

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

Compile and run. Voila! The first pyramid of Aaru! (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!)

Our approach so far has worked, but needing to know the exact coordinate of every vertex can be cumbersome. To best complete the last part (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. OpenGL maintains **transformation** information as part of its state. 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 object and the object now exists in **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.

Somewhere in your program, perhaps just above `renderScene()`, create a function called `drawMyTriangle()`. It should have a return type of `void` and accept no parameters.

We'll now draw a triangle with the following vertices. Give it the same nice gold color.

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

In the `renderScene()` method, call your triangle function (in addition to 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.

The way OpenGL maintains transformation information is through a stored matrix. We can manipulate this matrix and load new matrices into its place. When we manipulate the matrix, we would multiply the stored matrix with our transformation matrix. When we load a new matrix in, we would replace the stored matrix with our transformation matrix. Both methods are valid but require different implementations. We will use the first method, modifying the stored matrix through multiplication, as it is simpler to implement.

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 mMtx,
                                 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. The first parameter we will pass the identity matrix in (this would allow 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(), glm::vec3(300, 300, 0));
```

transMtx now contains the matrix to move the triangle. We need to tell OpenGL to apply this transformation to future draws. This is done by telling OpenGL to multiply its current transformation matrix by our translation matrix. On the next line after the one just added above, enter:

```
glMultMatrixf( &transMtx[0][0] );
```

What's this doing? Well we are telling OpenGL to multiply its matrix stored in memory with our matrix that starts at the memory address of the first element in our matrix. 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.

Great, we now see our triangle. But we want something cooler. 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 triangle to your new pyramid method. Compile and run. You should still see your triangle right where it was.

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, translate each triangle instance to the following locations:

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

Compile and run. You should three new triangles. But hmm, none are in the right place to make a pyramid, they are all over the place. What's happening? Let's think about what happens every time we tell OpenGL to apply our translation matrix using `glMultMatrixf()`. 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,

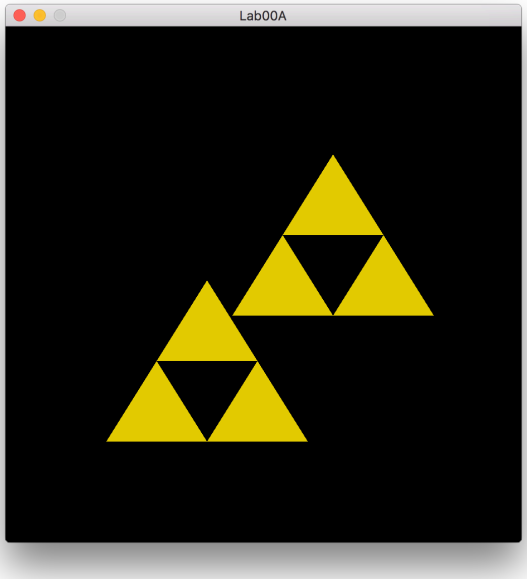
25)) or more simply we can apply the inverse of our translation matrix. After your first triangle call, multiply the matrix by the inverse. You can get the inverse of a matrix using the glm call

```
glm::mat4 glm::inverse( glm::mat4 mtx );
```

Your first triangle draw call should look like:

```
glm::mat4 transMtx1 = glm::translate(glm::mat4(), glm::vec3(-25, -25, 0));
glMultMatrixf( &transMtx1[0][0] ); {
     drawMyTriangle();
}; glMultMatrixf( &( glm::inverse( transMtx1 ) )[0][0] );
```

This handles making sure that only the triangle sandwiched between the two `glMultMatrixf()` calls gets moved. Make sure the next two triangles follow the same template. Compile, run, and you should have two pyramids like below.



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

Lab00B

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. 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 ride 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 Aaru Park 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 **WEDNESDAY AUGUST 30 11:59 PM**!! THIS IS AN INDIVIDUAL LAB!