CSCI 441 - Lab 00
Friday, August 26, 2016

LAB IS DUE BY **WEDNESDAY AUGUST 31 11:59 PM**!! THIS IS AN INDIVIDUAL LAB!

In today's lab, we will learn how to do basic 2D drawing using OpenGL and GLUT. 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 graphics cards 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 GLUT 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."

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".

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 a special program "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 say two g++ calls. You should now have a program called "lab00a." Run this program.

If you are getting an error "make cannot be found" it may be called gmake on your install. Additionally, try the next solution too.

If you are getting an error "<GL/glut.h> cannot be found", this was the solution that worked last semester. In PowerShell, you can type "which g++" and it should say either "C:/Strawberry/…" or "…/mingw/…". Some students last year said when they switched to a different computer, this error went away. Others used the following command to change their PATH. You must instead use the regular command prompt (search for 'cmd') and type this command to edit your path:

```
set PATH=C:\Strawberry\c\bin;%PATH%
```

Right now, we don't have any way to close our window. You will need to go back to the terminal and hit Ctrl+C to close the 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++, 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. 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.

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!

(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 triforce will always be between the coordinates (100,100)

and (300, 260). What if we wanted the triforce in a different place? We need to compute new coordinates. What if we wanted the triforce smaller? We need to compute new coordinates. What if we wanted the triforce 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`.

The way OpenGL maintains transformation information is through a stack. We can push and pop to this stack. When we push to the stack, a copy of the current transformation is added. Inside your triangle function, we'll first push a new copy to the stack. This is done through the call:

```
glPushMatrix();
```

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

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

And since we pushed a matrix to the stack, we need to pop a matrix off to return to the state we were before we started drawing the triangle.

```
glPopMatrix();
```

In the `renderScene()` method, call your triangle function (in addition to the previous triangles you had drawn). Compile and run the program.

Can you see the whole triangle? We'll get to that in a moment. First, play around with the push/pop commands. **How many times are you able to push to the stack? How many times can you pop?** Put the answers to these two questions in your README file. Now back to the drawing.

Well it seems 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. Since we will be changing the transformation information, we will want to push and pop the stack before and after our triangle function in `renderScene()`.

Push to the stack, then translate our triangle to some position in the window. This is accomplished through the call

```
glTranslatef( float x, float y, float z );
```

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 call essentially moves the origin in object space to a new position in our world space. Move the triangle so you can fully see it in the window. (Perhaps use (300, 300, 0)); Compile and build ot test.

Great, we now see our triangle. But we want something cooler. Create another function called `drawMyTriforce()` which also has a return type of `void`.

We will need to call our triangle function three times. Translate each instance to the following locations, being sure to use push and pop appropriately.

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

In `renderScene()`, now call your triforce method instead of the triangle method. Compile and run.

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 "shell." 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) 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 hero's home world looks like. Are there trees and bushes scattered around? A river running through? Mountains?

Next week for A2, you will see all of the home worlds connected together and you will be able to travel between worlds (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.

Q3: Was this lab fun? 1-10 (1 least fun, 10 most fun)
Q4: How was the writeup for the lab? Too much hand holding? Too thorough? Just right?
Q5: 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 Blackboard. You will see under the content section a Lab00 folder. Upload your zip file here.

LAB IS DUE BY **WEDNESDAY AUGUST 31 11:59 PM**!! THIS IS AN INDIVIDUAL LAB!