CSCI 441 - Lab 01
Friday, September 2, 2016

LAB IS DUE BY **FRIDAY SEPTEMBER 9 11:59 PM**!! THIS IS AN INDIVIDUAL LAB!

In today's lab, we will add keyboard & mouse interaction. Then we will add some animation. Finally, we'll put them together. We'll pick up with where we left off in Lab00A and work with our triforce again.

Please answer the questions as you go inside your README.txt file.

Part 1 – The Keyboard

We have already seen two callbacks within GLUT, the display callback and the reshape callback. Those are the only two callbacks required. But there are many more callbacks available, in this section we'll add the keyboard callback.

As mentioned, a callback accepts a function as a parameter and runs that function when an event occurs. (If you want a longer description about callbacks, here is a website to browse http://www.tutok.sk/fastgl/callback.html). We can specify our own function to use, as long as it follows the expected format with the proper parameters. For instance, any keyboard callback function must be of the following form:

```
void functionName( unsigned char key, int mouseX, int mouseY );
```

We can set any functionName that we want (let's use myKeyboard), but we need to be sure the function accepts those three parameters. The parameters are the key that was pressed, and the location of the mouse within the window when the key was pressed. For now (and generally) we will only be concerned with what key was pressed.

Begin by declaring a creating a function for the keyboard callback specified above. Choose an appropriate name. We now need to register our callback with GLUT. In our main() function, we saw where the other callbacks for display and reshape were set, so we will add the keyboard callback there as well. The GLUT command for this is:

```
glutKeyboardFunc( myKeyboard );
```

This callback will only handle the regular character keyboard keys, or anything that can be represented in ASCII (hence the unsigned char). There are other callbacks if we want to handle the arrow keys, page up, page down, etc.

At this point, we should be able to build and run our program.

Well nothing happens because our keyboard function isn't doing anything! Let's do something incredibly useful. Until now, we have had to hit Ctrl+c in the terminal exit our program. Let's set up our program to close itself. In your keyboard function that you defined, you will need to check if the ESC (escape) key was pressed. If it is, then we can call `exit()` to close down our program. (GLUT is not very elegant and does not provide a glutExitMainLoop() function. It feels that is the windowing system's job.)

Build and run your program. Press the escape key. Did it exit?

Congrats! We have keyboard interaction. We'll come to our keyboard function later on.

Part II – Animation

Now that we can easily exit, let's start adding some animation. We will want to use double buffering to smoother animation. As discussed in class on Wednesday, we need to request double buffering from GLUT by passing GLUT_DOUBLE to the display initialization mode. We want to keep the RGB mode as well.

Next, in our display function we need to begin by telling OpenGL to draw to the back buffer. Now we will clear what is in the buffer and draw what we have currently. Finally, instead of calling `glFlush()` directly, we want to tell GLUT to swap the back and front buffers with `glutSwapBuffers()`.

Voila! We are set up for animation. So let's get to it. It'd be cool if our triforce spun continually and we could just stare at it going round and round and round and round. We will want to use a timer function instead of a idle function.

Create a timer function matching the following format:

```
void myTimer( int value );
```

Now when we go to register our callback, it is slightly different. The GLUT call is:

```
glutTimerFunc( unsigned int milliseconds, myTimer, int value );
```

Basically, we are telling GLUT how long to set the timer, what function to call when the timer expires, and what value to pass to the function. Almost always, we will pass 0 as the value since we will not use the value often.

Ok our timer is set, let's do something. Add a global variable that will store the current rotation for our triforce. Name it appropriately (such as triforceAngle) and give it some starting value. Inside our timer function, we'll now change our triforce angle by some amount. Let's add 20 degrees to the triforceAngle. Lastly, in our display function, we will need to make sure we are rotating the triforce by our variable. Change the rotation value "45" to your variable.

Build and run.

The timer function does not signal a redisplay event so our display never changed. Inside the timer function AFTER we changed the angle value, let's tell GLUT to redraw by simply calling

```
glutPostRedisplay()
```

Build and run again.

We're almost there! It should have moved, but it's not movING. When we tell GLUT to register a timer, it only registers a single timer. The final step of our timer function should be to register another timer callback. Just as we registered the call back in our main function, we need to add the same line again at the end of our timer function.

Add that, build, and run.

Great! It's moving, but a little hard to see. Experiment with different values to change the angle by. Choose one that looks visually appealing and has smooth animation.

Part IIIA – Mouse Buttons

We're now going to add in the final piece of our interaction – the mouse. The mouse callbacks behave just as the keyboard callback does. Let's create a function for the mouse buttons that follows the form:

```
void myMouse( int button, int state, int mouseX, int mouseY );
```

And we'll register that callback with our keyboard by calling GLUT's:

```
glutMouseFunc( myMouse );
```

When a mouse button is pressed or released, our mouseName function will be called. The button will be set to the corresponding mouse button, the state will signify is the button was pressed (down) or released (up), and the location of the mouse at the time of the click is returned as well.

Let's add another visual effect. Previously, we set our triforce to be a nice gold color. Let's allow the user to change the color of the triforce. To get everything in place, let's add another global variable that's a boolean to act as a flag for our triforce color. Name it something appropriate (EVIL_TRIFORCE could work) and set it to be false initially. When we draw our triforce, we'll check the value of this boolean. If we do not have an evil triforce (a good triforce), then we'll keep our gold color. Otherwise, if we do have an evil triforce, let's tell OpenGL to draw the triforce in red instead.

Now in our mouse function, we'll allow the left mouse button to change this flag. We need to check both the button and the state of the button. First, check if the button passed to our function is the left button, GLUT_LEFT_BUTTON. If it is, then now let's check what state our button is in. If the button has been pressed, GLUT_DOWN, then let's set our evil triforce flag to true. Otherwise if the button has been released, GLUT_UP, then we'll set our flag back to false.

Compile and run.

Now stuff is happening! Let's add another way for the user to make an evil triforce. Let's go back to our keyboard function. Inside here, if the user hits the 'c' key (to signify a [c]hange in our triforce's allegiance) we will set the evil triforce flag to the opposite of what it's current value is.

Great, let's do the final piece now.

Part IIIB – Mouse Movement

When the mouse button is clicked, that is considered an "active" event. However, if you are just moving the mouse around the screen not clicking, then this is considered a "passive" event. And there's an app for that! Sorry, wrong class, I mean, there's a callback for that! Let's create our final function:

```
void myMotion( int mouseX, int mouseY );
```

And be sure to register that callback!

```
glutPassiveMotionFunc( myMotion );
```

We are now hooked up. It'd be real cool if the triforce moved around the screen following our mouse. You may have guessed – two more global variables! One for the x position and one for the y position. Create them and set them to some initial value. When we draw the triforce, instead of it being stuck at (200, 300) change the translate call to these two global variables.

Build, run, and verify the triforce is where you set the initial value to.

Ok great, let's now tie it to the mouse. Inside our motion function, all we need to do is set our global `triforceXPosition` to the mouseX value and the `triforceYPosition` to the mouseY value.

Compile and run. Ta Da!

Well, OpenGL and GLUT have a different idea of where (0, 0) should be. OpenGL uses the lower left corner for (0, 0) while GLUT considers the upper left corner to be (0, 0). Change your `triforceYPosition = mouseY` assignment to correct for this disagreement between OpenGL and GLUT.

And there we have it! Animation? Check. Interaction? Check. We could certainly do a whole lot more (and we will in A2)! For instance, you could use the arrow keys to make the scale larger and smaller. But that's for another day and adventure.

Q1: To do all this interaction, we are making a lot of global variables. Is this the best solution? Is there any way around this? Is it possible to not use global variables? (Note: There's no right answer to this, what is your opinion to best approach this problem.)

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

To submit this lab, zip together your main.cpp, Makefile, and README.txt. Name the zip file <HeroName>_L1.zip. Upload this on to Blackboard under the Lab01 section.

LAB IS DUE BY **FRIDAY SEPTEMBER 9 11:59 PM**!! THIS IS AN INDIVIDUAL LAB!