CSCI 441 - Lab 01
Friday, September 04, 2020

LAB IS DUE BY **FRIDAY SEPTEMBER 11 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

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 MUCH 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( GLFWwindow *win, int key, int scancode, int action, int mods );
```

We can set any `functionName` that we want (let's use `keyboard_callback`), but we need to be sure the function accepts those five parameters in that order. The parameters are the window the key was pressed in, the key that was pressed, a unique scancode, whether the key was pressed or released, and which modifier keys were pressed. For now (and generally) we will only be concerned with what key was pressed and what action occurred.

Begin by declaring and creating a function for the keyboard callback specified above. Choose an appropriate name and place it with our other callbacks (Hint: There's a section marked `TODO #1`). We now need to register our callback with GLFW. In our `setupGLFW()` function find `TODO #2`, we will add the keyboard callback there. The GLFW command for this is:

```
glfwSetKeyCallback( window, functionName );
```

where *`functionName`* is the name of your function. This callback will only handle any of our keyboard keys.

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 to exit our program or click the X on the window. 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 `glfwSetWindowShouldClose(win, GLFW_TRUE)` to tell the window to close and then halt our rendering loop.

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

Congrats! We have keyboard interaction. We'll come back 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 for smoother animation. As discussed in class on Wednesday, we need to request double buffering from GLFW by hinting to our window to open with double buffering. Find the other `glfwWindowHint()` calls and add one that sets `GLFW_DOUBLEBUFFER` to `GLFW_TRUE`.

Next, in our draw loop we need to begin by telling OpenGL to draw to the back buffer. Find `TODO #3` and add the call to draw to the back buffer.

Next we will tell OpenGL to clear what is in the buffer and draw what we have currently. The call to tell GLFW to swap the back and front buffers with `glfwSwapBuffers()` is already present for you.

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.

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. After we've swapped buffers and checked for events, we'll now change our triforce angle by some amount. Let's add about 60 degrees (or 0.05 radians since glm wants radians) to the triforce angle. Lastly, in our render function, we will need to make sure we are rotating the triforce by our variable. Change the hard coded rotation value "0.785" to your variable.

Build and run.

Great! It's moving, but may be 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 mouse_button_callback( GLFWwindow *window, int button, int action, int mods );
```

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

```
glfwSetMouseButtonCallback( window, mouse_button_callback );
```

When a mouse button is pressed or released, our mouse function will be called. The button will be set to the corresponding mouse button, the action will signify if the button was pressed or released, and any modifiers at the time of the click are 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.

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 action of the button. First, check if the button passed to our function is the left button, `GLFW_MOUSE_BUTTON_LEFT`. If it is, then now let's check what action our button is performing. If the

button has been pressed, `GLFW_PRESS`, then let's set our evil triforce flag to true. Otherwise if the button has been released, `GLUT_RELEASE`, then we'll set our flag back to false.

Regardless of the action that took place, we now need to change the colors. Since we already sent the vertex attributes to the GPU and received a VAO handle back to reference. Find `TODO #4`. In this update function, we need to send the GPU the new color information to replace what it already has stored in memory. We will use the class library function

```
CSCI441::SimpleShader2::updateVertexArray( GLuint VAO,
                                           std::vector<glm::vec2> points,
                                           std::vector<glm::vec3> colors
                                         );
```

If our flag is true, we need to pass the red color vector to the function to send to the GPU. If our flag is false, we need to pass the gold color vector to the function to send to the GPU. Regardless of our flag, we need to send the `triangleVAO` (to reference the correct location in GPU memory) and the existing `trianglePoints` to draw the same shape.

Compile, run, and start clicking.

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 and call our same `updateTriangleColors()` function.

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 cursor_callback( GLFWwindow *window, double x, double y );
```

And be sure to register that callback!

```
glfwSetCursorPosCallback( window, cursor_callback );
```

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 cursor function, all we need to do is set our global `triforceXPosition` to the cursor x value and the `triforceYPosition` to the cursor y value.

Compile and run. Ta Da!

Wait, something's not quite right. Our drawing window places the origin in the lower left corner of the window. However, the system returns our mouse position with the origin at the upper left corner of the window. No problem, we just need to subtract the returned y location from our windowHeight before assigning it to the triforceYPosition.

Compile and rerun.

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: 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?
Q4: Any other comments?

To submit this lab, zip together your main.cpp, CMakeLists.txt, and README.txt. Name the zip file <HeroName>_L01.zip. Upload this to Canvas under the L01 section.

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