CSCI 441 - Lab 03
Friday, September 18, 2015
LAB IS DUE BY **FRIDAY SEPTEMBER 25 11:59 PM**!!

Today, we'll add some new features to our OpenGL Toolbox and display some nifty looking curves.

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

Step 1 – Getting Our Code To Compile

THIS IS AN INDIVIDUAL TASK.

If you try to build this lab straight out of the box, it won't work. Type `make`. Initially you'll get an error along the lines of "Can't find GL/glui.h".

We need to do a little setup first. You must follow these instructions exactly for the latest Makefile to work. We'll be performing similar steps like this in future labs so be sure to follow directions exactly.

1. Navigate to the root of your Z: drive
2. Create a folder called **CSCI441GFx** (it should look like: Z:/CSCI441GFx)
3. Change into this new folder you just made
4. Create a new folder called **include** (Z:/CSCI441GFx/include)
5. Create another new folder called **lib** (Z:/CSCI441GFx/lib)
6. From this lab zip file, copy the contents of the include and lib folders into the respective folders you just made.

You should now have the following file structure:

- Z:
    - CSCI441GFx
        - include
            - GL
                - glui.h
        - lib
            - libglui.a

Alright cool, we've got GLUI set up. If you want to make sure you set everything up correctly, switch into the gluiTest folder and type `make` to build this example program.

Ok, back to our lab. Type `make` again. Bah another error! Can't find Point.h? What's that? Well it's our first task.

Step 2 – Creating a Point Class

From this point forward in this lab, you may work with another student. HOWEVER, you each MUST submit your OWN submission. IF you work with another student, you BOTH MUST make note in your README who you had worked with.

We're getting a jump moving towards our OOP-tastic goals. We need to create a new file Point.h that will contain our Point class. We'll put the class declaration into Point.h and the implementation into Point.cpp, because it's good practice and that's what our Makefile is expecting.

Our Point class will be very basic. It needs three member variables (x, y, z) of an appropriate type. We then have two constructors, the default constructor with no parameters and an overloaded constructor that takes three parameters and assigns them to x, y, z respectively.

In our header file, outside the class definition, let's add three overloaded operators:

```
1. Point operator*( Point p, float f ); // multiplies a Point by a float
2. Point operator*( float f, Point p ); // multiplies a float by a Point
3. Point operator+( Point a, Point b ); // adds two Points together
```

All the operations should operate on each member variable of the Point. For example, the first implementation would look like:

```
Point operator*( Point p, float f ) {
     return Point( p.getX() * f, p.getY() * f, p.getZ() * f );
}
```

Finish the other two in our Point.cpp file. This will make our math much simpler later on.

Now that Point.h and Point.cpp are completed, we can finally compile! Type `make` and run the program.

Q1: Does it work? What are we looking at? What type of camera model is implemented?

Step 3 – Adding a Basic Menu

In our main() function, we have everything set up and the shells in place. We need to fill in the missing pieces. You should see a function `createMenus()` being called. Find `TODO #01` inside the `createMenus()` function. We'll do all of our

menu setup here. Reference Slide 39 from Wednesday's class and use your Primer book for the syntax.

On Slide 39, there are four steps. The first step requires us to register a callback with GLUT. We'll use the `myMenu()` function that we'll define momentarily. Now with step two, we can begin adding options to our menu. When we add an option to the menu, we need to provide a string that gets displayed on the menu and a value that gets passed to our callback. For now, add an option that prompts the user to quit the program (so our text should probably be "Quit") and passes some value to the callback (let's just use 0). Finally, we need to attach the menu to a mouse button so as decided we'll use the Right mouse button.

Compile and run.

Q2: If you right click, do you see a menu? What happens when you click the Quit option?

Clicking the menu option creates a menu event, so we now need to handle it. `TODO #02` is inside our `myMenu()` callback we referenced. Inside our callback, we need to check the value being passed in. If it equals our exit option's value, then we'll quit our program just as we currently do from the `ESC` key.

Compile and run.

Q3: Now does the program exit?

Fantastic, moving on.

Step 4 – Loading Our Control Points

We are going to eventually be drawing a Bezier curve. As discussed, Bezier curves are defined by a series of control points. We will read the control points in from a file to allow greater flexibility in our Bezier Curve Viewer (the BCV for you folk in the know). `TODO #03` will get us in the right place. Make sure that a single command line argument has been provided by the user. Perhaps provide a helpful usage message if the user does not run your program as expected.

Test this step and make sure the program starts as desired.

Now we need to actually read in our data, `TODO #04` will accomplish this for us. Hopefully you figured out already that you need to pass the command line argument to `loadControlPoints()`. This function will read through the file and create a Point for each line in our file. The control file has the following structure

```
n-points
x1, y1, z1
x2, y2, z2
…
xn, yn, zn
```

The first line is the number of points in the file and then there are n lines that follow with one point per line. Read in each point and store it in our controlPoints vector. You may assume that the control file will be properly formed, meaning it will contain the proper number of points to draw a complete Bezier Curve (4, 7, 11, etc.)

There are two control files included with this lab, controlPoints4.csv and controlPoints7.csv. Compile, run, and test running/loading with each file.

Step 5 – Draw our Control Point Cage

We want to be able to visualize where our control points are in relation to the final curve. For this step we'll fill in `TODO #05` and `TODO #06`. First for `TODO #05`, draw a green sphere at the location of each control point. Make the sphere an appropriate size relative to our grid size.

Compile and run.

Q4: Run with the controlPoints4.csv file. How many spheres do you see?

Now for TODO #06, connect each pair of control points with a yellow line. Refer to how the grid gets drawn as a reminder of how to work with 2D OpenGL Primitives. You'll probably want to make the control cage lines a thickness of 3.0f so they are easier to see.

Compile and run.

Q5: Run with the controlPoints4.csv file. How many yellow lines do you see?

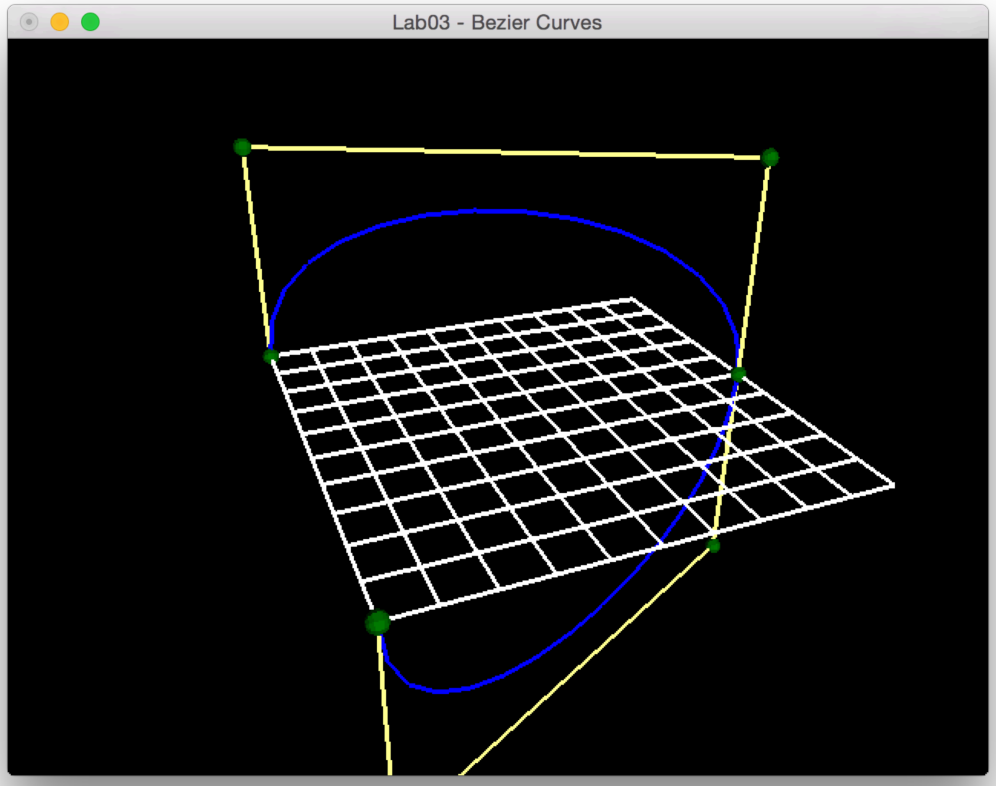
You should have not connected the last point back to the first point, so now we should see our full control point cage. Let's draw the curve!!

Step 6 – Drawing the Bezier Curve

Now we are ready to draw the curve. Use a blue line with thickness of 3.0f to draw the curve. You will need to fill in `TODO #07` and `TODO #08` to complete this task.

When testing, start with controlPoints4.csv. Once you have a single curve drawn, then test with controlPoints7.csv. Your program should be able to handle any number of Bezier Curves as specified from a file.

Once your view for controlPoints7.csv looks like below….you did it! Hooray!



EXTRA EXTRA! Extra Credit Achievement 

The follow section is not required for the lab. If you want to earn an extra credit achievement, then continue. Otherwise, skip to the end to zip up your submission. ☺

If you are feeling extra ambitious and want to earn an Extra Credit Achievement, then complete the Picking Made Easy tutorial included with this lab (the tutorial was originally put together by Oregon State University. Thank you to them for doing such a great job that no modifications are necessary). We want to be able to click on a control point and move its location. To earn the extra credit achievement, you must allow the user to Shift + Left Click on a control point. If the user clicked a control point, then highlight that control point red. Once the control point is selected, the user can move the control point's location along the X-, Y-, and Z-axis

through keyboard presses. If the user Shift + Left Click's and does not select a control point, then deactivate any previously selected control points. Only one control point can be active at a time.

Q6: Was this lab fun? 1-10 (1 least fun, 10 most fun)
Q7: How was the write-up for the lab? Too much hand holding? Too thorough? Too vague? Just right?
Q8: How long did this lab take you?
Q9: Any other comments?

To submit this lab, zip together your main.cpp, Point.h, Point.cpp, Makefile, screenshot, and README.txt with questions. Name the zip file <HeroName>_L3.zip. Upload this on to Blackboard under the Lab03 section.

LAB IS DUE BY **FRIDAY SEPTEMBER 25 11:59 PM**!!