CSCI 441 - Lab 08
Friday, October 23, 2020
LAB IS DUE BY **FRIDAY OCTOBER 30 11:59 PM**!!

Today, we'll be rendering a Bézier Curve and animating a sphere along the curve..

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

Step 0 – New Library Files

You'll notice an include folder with this lab. To make sure everyone got all the patches from before break, please copy the files into your Z:/CSCI441/include/CSCI441 folder.

Also, in CLion be sure to update the working directory to be the parent (Run > Edit Configurations > Working Directory = ..).

Begin by running the starter code. It will prompt you to enter the filename for a Bézier control file. There are two files included with this lab:

- data/controlPoints4.csv
- data/controlPoints7.csv

Run the program with each file. There are a number of starter pieces of code given to you:

- the control points are read in from the file
- the control points are plotted
- the control cage connecting the points are plotted
- a ground plane is rendered
- pressing '1', '2', '3' toggles between a point light, directional light, and spot light
- a working gourad shader using Phong illumination is included to render the triangles
- a working flat shader is included to render the lines

Next, take a look at the format of each of the data control point files. We will begin using external files to represent information from our scene. By loading a different file, we can generate a new scene.

The first line, `n`, of the file states how many control points are in our curve system. The next `n` lines then contain the `x,y,z` location of each control point in the local object space of the curve system. Be familiar with this file format because you will need to generate new control point files for your next assignment.

Step 1 – Draw The Curve

The first step we will want to do is actually draw the static curve. We'll do this in two steps.

Part I – Evaluate A Curve At A Given Point

At TODO #01, we have the function stub to solve the Bézier Curve equation. This function takes in the four control points to represent a cubic Bézier Curve and the parameter t at which to evaluate the curve at. Solve the curve equation (it's given in the lecture slides from Wednesday – make sure you copy it correctly) and return the point that results.

Great, that part was rather straightforward. Let's use it now!

Part II – Generate The Curve

TODO #02 is down in setupBuffers(). It is here because we want to generate an array of points to represent the curve. We'll then use these points to draw a line connecting all the points. You have a number of global variables at your disposable that you will want to use:

- bezierCurveVAO – the VAO for the curve
- bezierCurveVBO – the VBO for the curve
- numEvalPoints – the number of points generated that make up the curve

Be sure to be using these variables as they are used when rendering the curve. Look at LOOKHERE #1 to see them in use.

The first thing you'll need to do is ask the user what resolution they wish to render the curve at. The resolution corresponds to the number of steps we'll take when evaluating the curve. This also equals the number of lines drawn. Since we are using a line strip to draw the curve, this means that each curve will be made up of `resolution+1` points. Across our curve system, we'll need to generate the number of evaluation points equal to the number of curves times our (resolution+1). Set the `numEvalPoints` variable equal to this value. Look above to where the `loadControlPointsFromFile()` function is called for some additional variables that can help you in this calculation.

We're now ready to begin evaluating. We'll need to loop over every curve in our system. The controlPoints array holds all of the control points that make up our piece-wise system. Control Points 0-3 make up the first curve, 3-6 the second curve, 6-9 the third, and so forth. Set up a for loop for each curve in our system.

Inside the loop for efficiency, let's retrieve each of the four control points that make up the current curve. Make four variables `p0, p1, p2, p3` and set them equal to the successive position within the array (`controlPoints[0], controlPoints[1]`, and so forth). Remember that our system can be made up of multiple curves so generalize the positions off of your looping parameter `i`.

Great, we now have the four points that make up the current curve. We'll need another loop to step through our resolution. It should range from 0 to resolution inclusive. This corresponds to each step along the way.

Inside this inner for loop, we need to convert the current resolution step to a fractional parameter. If our looping parameter is `j`, then we would divide by the resolution to determine what fraction of the way we

are – be sure to cast to float! We're now ready to evaluate our current curve at the current parameter point.

We can now call the function we created in Part I above. We'll store the resulting point in to an array (or vector).

Once we've generated all of the points, we can create the VAO, the VBO, and connect the attribute pointer. You should be able to model these steps after the control cage VAO above.

You should now be able to run your program, enter the control file and resolution to see the curve.

Q1: What is an appropriate resolution to render the curve?

Step 2 – Animate A Sphere Along The Curve

The last piece we want to do is to now animate an object along the curve so our models can move in more interesting patterns.

The bulk of the code will go at TODO #03, but we'll need to make a global variable and update it in `updateScene()` for this all to work.

Part I – Setup the Animation

Begin by creating a global variable to track our current position along the entire curve system. It should start at 0.

Then, in `updateScene()` we'll want to increase this variable by a small amount every frame (say 0.005f for starters). This value will span our entire curve system, so it can range from 0 to `numCurves`. Recall that when we have a system parameter, such as 1.75, then the integral part represents which curve we are currently evaluating and the fractional part represents where along the corresponding curve we are.

Therefore, add a check to see if our global evaluation parameter has gone beyond the size of our curve system (`numCurves`). If it has, then reset it back to the beginning (0).

Perfect. Our global parameter should now be continually increasing from 0 to `numCurves` and back to 0 as our program runs.

Part II – Evaluate the Curve and Draw A Sphere

Now we're actually ready to enter the code at TODO #03. We need to determine the two pieces of information based on our global parameter we just created:

1. Which curve we are on (this is the integral part of the parameter value)
   Once we know which curve we are on, once again grab the four control points from the array that correspond to this curve.
2. What fraction along the curve we are (this is the fractional part of the parameter value)

With this information, we have the four control points and the parameter value now between [0, 1]. We can again call our evalBezierCurve() function to calculate the point in space along the curve we should be at.

We will then create a model matrix to translate to this position, send the matrices to the shader, and draw a sphere at this location.

Run your program and you should see a sphere moving along the curve and then jumping back to the beginning when it reaches the end.

Q2: Should the sphere animation step size equal the resolution the curve is drawn at? Why or why not?

We're now ready to draw any Bézier Curve system and animate any object in place of the sphere! Go forth and be curvy.

Q3: Was this lab fun? 1-10 (1 least fun, 10 most fun)
Q4: How was the write-up for the lab? Too much hand holding? Too thorough? Too vague? Just ri4ht?
Q5: How long did this lab take you?
Q6: Any other comments?

To submit this lab, zip together your source code and README.txt with questions. Name the zip file <HeroName>_L08.zip. Upload this on to Canvas under the L08 section.

LAB IS DUE BY **FRIDAY OCTOBER 30 11:59 PM**!!