CSCI 441 - Lab 09
Friday, October 30, 2020
LAB IS DUE BY **FRIDAY NOVEMBER 06 11:59 PM**!!

This lab will get you plotting a Bezier Surface with Tessellation Shaders.

Step 0 – Includes and Existing Code Structure

There is are two files in the include/CSCI441/ folder – a new materials.hpp and a patched OpenGLUtils.hpp. This adds new some predefined material properties for your use. Copy them to your Z:/CSCI441/include.

Going down the main.cpp, here is an overview of the code structure for this lab:

- We're storing all of our VAO descriptors in an array and defined constants to correspond the object of interest.
- We will have three different shader programs
    1. The Gourad Shader from Lab08 to illuminate our control point spheres
    2. The Flat Shader from Lab08 to render our lines
    3. The Bezier Shader to render our surface. This is where the bulk of today's lab will take place.
- sendMaterialProperties() now accepts a CSCI441::Materials::Material object as a parameter. This is from the materials.hpp library file that has a number of predefined material property sets for you to use.
- loadControlPointsFromFile() reads in a Bezier Patch file. It reads in n surfaces comprised of m control points.
- In setupBuffers(), we are calling loadControlPointsFromFile(). Upon completion of the function call, the following variables hold the necessary information:
    1. numControlPoints – this global variable is the number of unique control points
    2. numSurfaces – this global variable is the number of surfaces to be drawn
    3. controlPoints – this global variable is the glm::vec3 array of (x,y,z) locations
    4. patchIndices – this local variable is the order to connect the control points to make each surface
- After loading in the control points, the control cage is setup to connect each set of four control points in order. Additionally, the VBO and IBO for the patch are sent over to the GPU.

Great! Run the program. You will see the control points plotted as red spheres and white lines forming a grid of the control cage. You can press 'p' and 'c' to toggle displaying the control points and cage.

Step 1 – Tessellate It

There are a series of steps we'll need to do – both on the OpenGL side and the GLSL side. Let's handle the OpenGL side first since it is simpler.

Part I - OpenGL

There are three things we need to do on the OpenGL side:

1) In setupOpenGL(), we need to set the patch parameter property for the number of vertices per patch. This is accomplished with the call

```
glPatchParameteri( GL_PATCH_VERTICES, n );
```

where n is the number of control points that make up a single patch. Our global constant POINTS_PER_PATCH holds the number that we need. Pass this in place of n.

2) In setupShaders() we need to include compile our full tessellation shader program. If you look in the shaders/ folder, we have four shaders that we'll need to use for the Bezier calculation. The existing shader program is only using the Vertex and Fragment shaders. Add the other TCS and TES filenames to compile and link within the shader program. The constructor will take four arguments in order: Vertex, Tessellation Control, Tessellation Evaluation, Fragment.

3) In renderScene() we need to draw the patches after binding the VAO for the PATCH. We'll use our same glDrawElements() call but give it these four arguments:
   a. The primitive type – GL_PATCHES
   b. The number of indices to use – the number of surfaces times our points per patch
   c. The data type of each index – GL_UNSIGNED_SHORT
   d. The pointer to the start of our data – (void*)0

That's all! On to the GLSL side.

Part IIA – GLSL – Tessellation Control Shader

We need the Tessellation Control Shader to accomplish its three tasks and these correspond to our three TODOs.

A) First, specify the number of vertices that make up the control points for a patch. This value must match the value entered in #1 from the OpenGL section above.
B) We will pass the control point for our current invocation through from the input to the output.
C) We need to set the tessellation levels. Start by setting the four outer levels and the two inner levels to 2.

The above steps are pulled directly from Wednesday's slides. As a sanity check, the above steps should take 8 total lines of code.

The control shader is the simpler of the two. Let's finish the process out.

Part IIB – GLSL – Tessellation Evaluation Shader

The first thing the evaluation shader has to do is to specify the primitive information. At TODO A, we'll want to use quads as our primitive. Then choose a spacing type (equal) and a winding order (ccw) .

The next two steps is to get all of our data that we'll need to compute the Bézier Patch equation. First at TODO B, get our u and v values from the gl_TessCoord variable. These correspond to the x and y components of gl_TessCoord – store them in variables u and v respectively.

Now at TODO C we need to get our 16 control points from the gl_in[].gl_Position array. They go in order 0-15. I recommend storing them in variables named p00, p01, p02, p03, p10, p11, etc.

We're now ready to perform our Bézier Curve equation. Create a helper function at TODO D to perform this equation. The function should take in the four control points and the floating point parameter value. This is the same that we had done in Lab08, but now the GPU will compute the equation. Note that our control points are vec4.

Now at TODO E, call the function you just created for the sixteen control points and two parameter values. Refer to the slides for the proper ordering to call the points. Store the result of the patch equation in a variable.

Finally at TODO F, we need to set the output position equal to the point we just computed. As with the vertex shader, assign the built in gl_Position variable.

We can now rebuild and run our program. You should see a green (or magenta depending on your chosen winding order) pyramid.

We'd like it to appear a bit smoother. Increase the tessellation levels to be 20 for all levels. Rerun and the surface should flow much more gently.

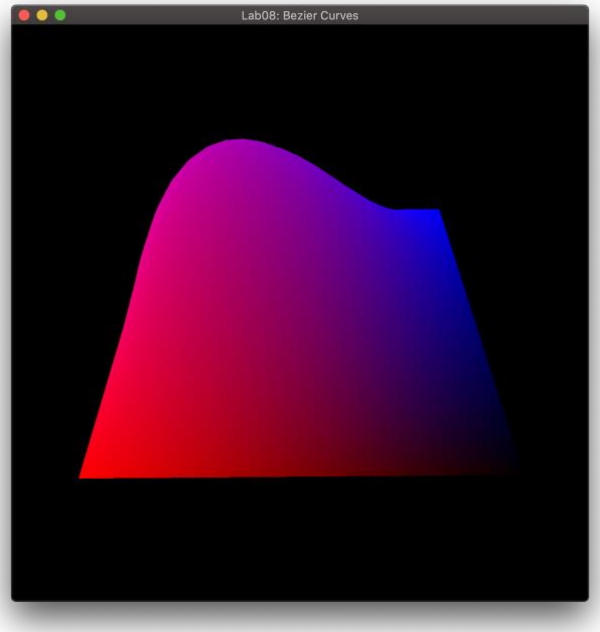
Step 2 – Color It

Let's add some color to the fragments so we can better determine the shape of the surface. First in the TES create a varying output of type vec3 that will be sent to the fragment shader.

Now, after setting the gl_Position variable, set this varying to have its R component equal to u and B component equal to v. Set G to 0. The (u, v) corresponds to the coordinate within the patch.

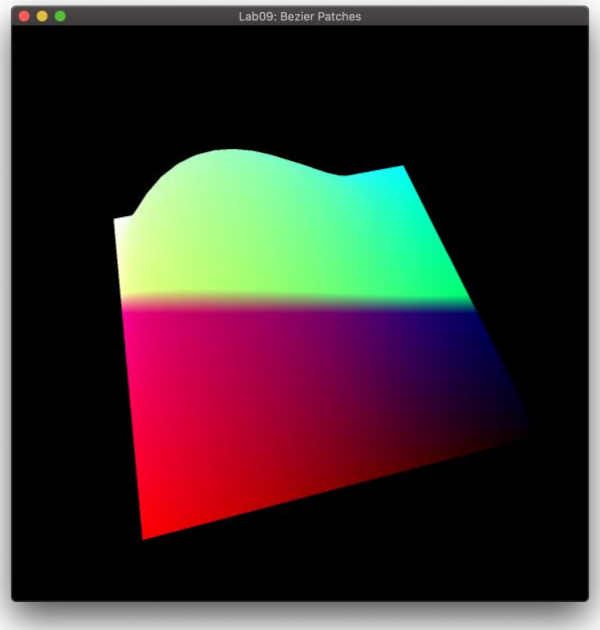
Switch to the fragment shader, receive the varying input and set the output to be the varying we just received (instead of the hard coded green).

Rerun and you should see an image similar to below:



Let's make one final adjustment to our color. Change the G component to be the y-value of our Bezier point that we computed in Step 2.

Rerun and rotate around the surface. Is the color staying constant as the patch moves?



No, the color for a corresponding part of the surface changes as the object rotates. Let's think about what space our Bezier point is in when we set the height to use for color.

Let's trace through the transformation pipeline. Look in our Vertex Shader - we receive the control point location in object space and transform it to clip space. Look in the TCS – we pass the value through, still in clip space. Look in the TES - we then processes and set the final point – in clip space.

We need to change which space we set our patch point for coloring in. This is accomplished by changing when the transformation happens. Begin by removing the mvpMatrix from the vertex shader. Now the Vertex Shader will pass the control point through still in object space.

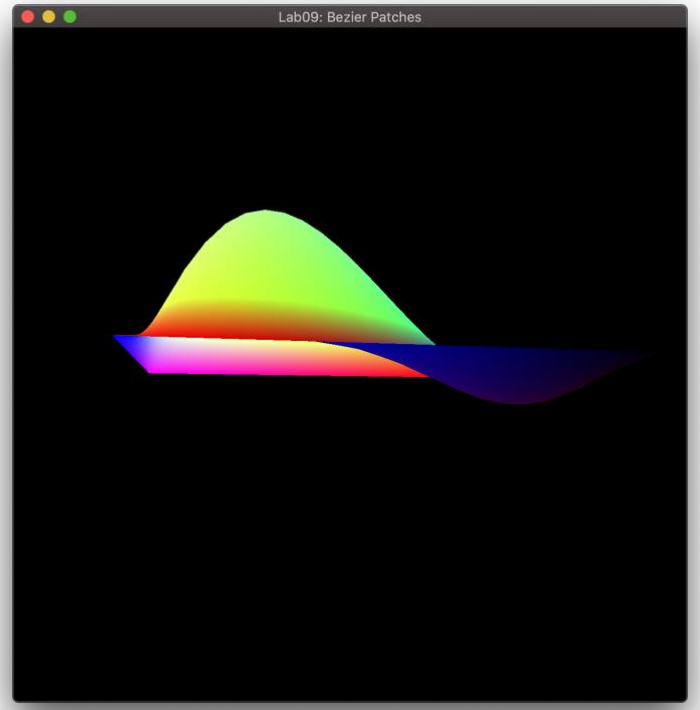
The TCS will also pass the control point through still in object space.

Now in the TES, add the mvpMatrix back as a uniform. We purposely stored the result of the Bezier curve calculation in a separate variable. When we set the output gl_Position, multiply our Bezier point by the MVP matrix to set the output in clip space.

Q1: What is the negative effect of multiplying in the TES instead of the VS?

The patch point we are using as a varying to the Fragment Shader will still be in object space.

Rerun. Now our patch should look similar to below (with the height affecting color)



Rotate the camera around and the color will remain constant as the object rotates.

Step 3 – What's Next?

As far as this lab, we're done.

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

To submit this lab, zip together your source code and README.txt with questions. Name the zip file <HeroName>_L09.zip. Upload this on to Canvas under the L08 section.

LAB IS DUE BY **FRIDAY NOVEMBER 06 11:59 PM**!!