CSCI 444/544 - Lab 3
Friday, February 5, 2021
LAB IS DUE BY **FRIDAY FEBRUARY 12 11:59 PM**!!

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

Step 0 – Patch Files

There are updated files in the include/CSCI441/ folder. This adds some new functionality:
- ShaderProgram has functionality for setting uniforms
- Shader debug info only displays if values are non-zero
- Shader Pipeline class added

Great! Run the program, you should see the FPS information and some trianagles. If you receive compiler errors, then the files weren't patched properly.

Step 1 – Plot the Control Points

Inside of `setupBuffers()`, we've hard coded in some control points and the order they make up our patch controls.

Previously our chosen primitive type was a triangle. Well now we just want to plot points. Here is the call we used to draw the cube previously

```
glDrawElements( GL_TRIANGLES,
                iboCounts.cube,
                GL_UNSIGNED_SHORT,
                (void*)0 );
```

The four arguments to this function are:
1. The primitive type to use
2. The number of indices we are passing
3. The type of each index
4. The offset to start in memory

We want to modify the above call to work with our control points. What is the new primitive we should use to draw points? How many are there?

Once you add in the new line at TODO #1 and run the program – you should see some red points floating in space.

Great! Let's make them disappear and put a patch in its place.

Step 2 – Tessellate It

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

Part I – CPU + OpenGL

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

1) First we need to include our two tessellation shaders in our shader program. If you look in the shaders/ folder, we have four shaders that we'll need to use. Be sure to list them in order: Vertex, Tessellation Control, Tessellation Evaluation, Fragment.
2) Next, we need to add a new line above in setupOpenGL() that sets the number of vertices that make up a patch. (This command is in the slides and goes at TODO #2).
3) Finally, we need to change from drawing points to drawing patches. At TODO #3, change the old line that drew points to use the patch primitive instead.

That's all! On to the GLSL side.

Part IIA – GPU + 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 #2 from the OpenGL section above.
B) We will pass the control point for our current invocation through from the input to the output.
C) If we are currently on invocation 0, then we'll set the tessellation levels. Start by setting the four outer levels and the two inner levels to 2.

As a sanity check, the above steps should take 10 total lines of code.

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

Part IIB – GGPU + LSL – Tessellation Evaluation Shader

(Hint: OpenGL 4 Shading Language Cookbook Chapter 7 will help.)

The first thing the evaluation shader is to specify the primitive information. At TODO D, we'll want to use quads as our primitive. Then choose a spacing type and a winding order.

The next two steps is to get all of our data that we'll need to compute the Bézier Patch equation. First at TODO E, 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 F we need to get our 16 control points. 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 G to perform this equation. The function should take in the four control points and the floating point parameter value.

Now at TODO H, 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 I, we need to set the output position equal to the point we just computer.

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

We'd like it to appear a bit smoother. Increase the tessellation levels to be 15 for all levels. Rerun and the surface should flow much more gently.

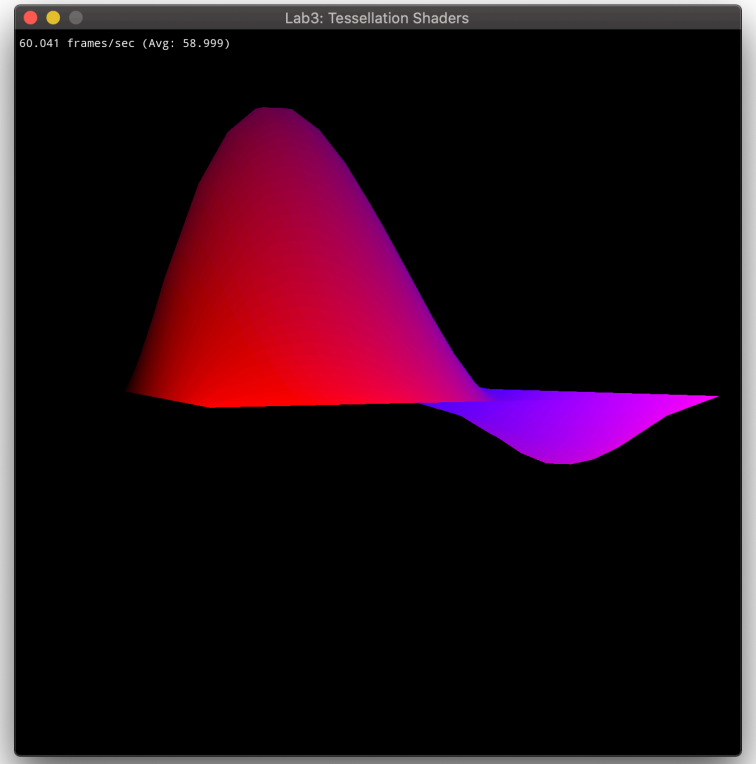
Step 3 – Color It

Let's add some color to the fragments so we can better determine the shape of the surface. First at TODO J create an output of type vec3 that will be sent to the fragment shader.

Now, at TODO K set this varying to have its R component equal to u and B component equal to v. Set G to 0. The (u, v) correspond to the coordinate within the patch.

Switch to the fragment shader, at TODO L receive the varying input. Finally, at TODO M set the output to be the varying we just received.

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. In our Vertex Shader we receive the control point location in object space and transform it to clip space. The Control Shader passes the value through, still in clip space. The Evaluation Shader then processes and sets the final point – in clip space.

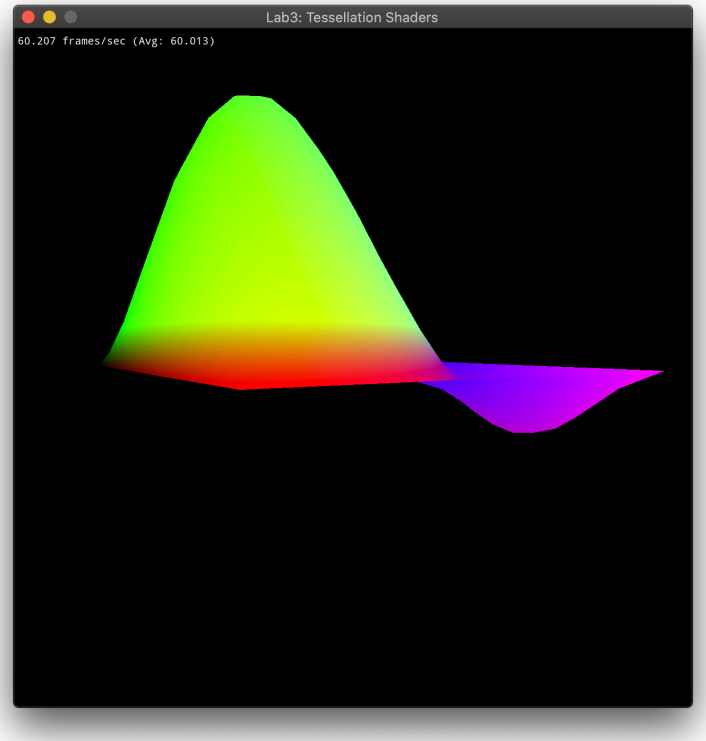
We need to change which space we set our patch point for coloring in. This is accomplished by changing when the transformation happens. Begin by removing the mvpMtx from the vertex shader. Now the Vertex Shader passes the control point through still in object space.

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

Now in the Evaluation Shader, add the mvpMtx back. We purposely stored the result of the Bezier curve calculation in a separate variable. When we set the output gl_Position, multiple our Bezier point by the MVP matrix to set the output in clip space.

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 4 – What's Next?

As far as this lab, we're done. Submit your project source code to Canvas as a single zip file.

For the next assignment, we don't want to hardcode in the control points – instead we will read them from a file. We will also have multiple patches now. We will want to display both the control points AND the surface. (How many programs needed?) Since we will then be applying the Phong Shading in the Fragment Shader, we also need vertex normals for each of our points.

LAB IS DUE BY **FRIDAY FEBRUARY 12 11:59 PM**!!