CSCI 441 - Lab 10

Friday, November 2, 2018

LAB IS DUE BY **FRIDAY NOVEMBER 9 11:59 PM**!!

Today, we'll look at how to properly use geometry shaders and apply texturing. We will begin with a pass-through geometry shader and then expand a point into a textured quad.

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

Step 0 – Get the Libraries copied

There are some new CSCI441 library files included in this lab. Copy them into your existing `Z:/CSCI441/include/CSCI441` folder.

Step 1 – Set up OpenGL Code

Type make. We can run the lab by simply typing

```
./lab10
```

Hrmma a black void. There will be three files you need to modify for this lab:

- `main.cpp`
- `shaders/billboardQuadShader.g.glsl`
- `shaders/billboardQuadShader.f.glsl`

Let's look at some things going on. Go to `LOOKHERE #1` in `setupShaders()` of `main.cpp`. Hooray, a Shader Program class to make our lives easier! This code is in `include/CSCI441/ShaderProgram3.hpp`. Note how our uniforms correspond to the ModelView matrix and the Projection matrix separately and not as the MVP like prior. This will be important very soon.

Scroll down to `LOOKHERE #2` in `setupBuffers()`. We are generating `NUM_POINTS` (which is set to 20) points set to random locations in the range [-5, 5] in all dimensions. These points are then put into a VBO and wrapped in a VAO.

And now in `LOOKHERE #3` we use our program, set our uniforms, bind our VAO, and render the points. Let's get everything hooked up so we can see stuff. Open up `shaders/billboardQuad.g.glsl`. This file corresponds to our Geometry Shader.

We'll begin with `TODO #A` where we need to set the primitive input type. This takes the form of

```
layout( primitiveType ) in;
```

Since in `main.cpp` we called the function `glDrawArrays()` with the argument `GL_POINTS`, that means our geometry shader must be ready to accept points as input. Therefore *`primitiveType`* should simply be `points`.

The next step, at `TODO #B`, is to set the primitive output type and number of vertices the geometry shader could output. This takes the form of

```
layout( primitiveType, max_vertices = n ) out;
```

Since we are creating a pass-through shader, the primitiveType should again be `points` and the max_vertices will be `1`.

Now we see the `projMatrix` uniform. Where is the ModelView matrix? That's in the vertex shader. Our vertex shader is performing the transformation into eye space and our geometry shader will be responsible for completing the transformation into clip space.

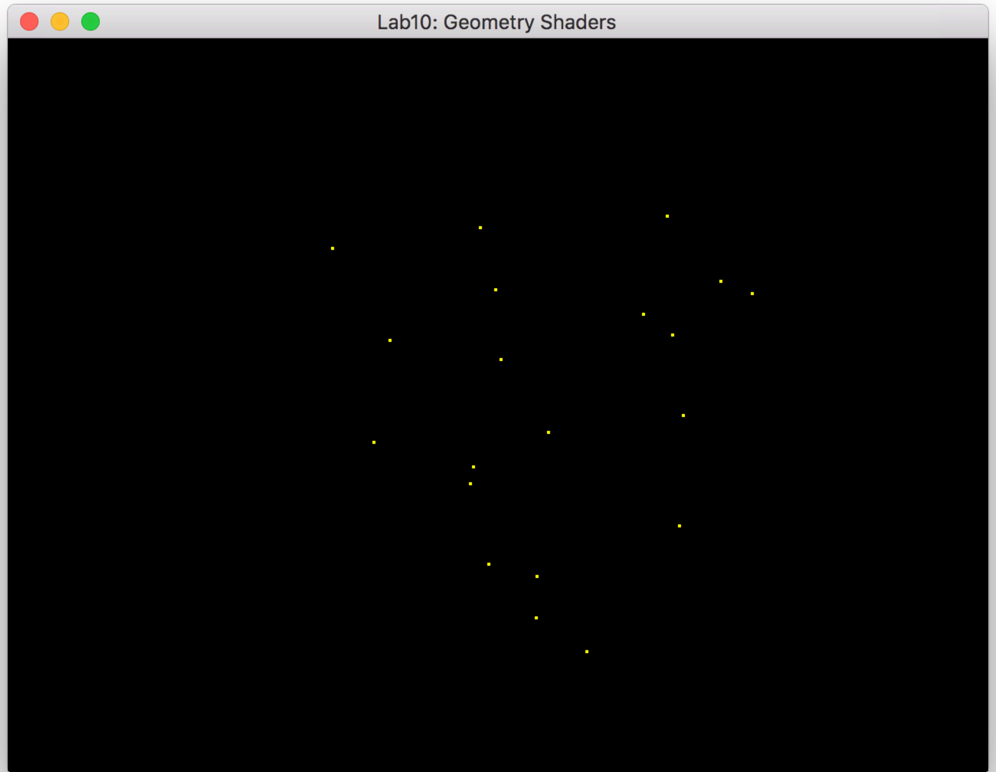
Let's now make the geometry shader do something! The geometry shader needs to set the value of `gl_Position` for every vertex it emits. And this needs to be in clip space. The input to the geometry shader comes in the form of an array named `gl_in`. This is actually an array of structs so we will want the `gl_Position` member of `gl_in`. This variable is equal to the output of our vertex shader, so the vertex in eye space. We'll need to transform this into clip space. The line we'll need to add at `TODO #C` is

```
gl_Position = projMatrix * gl_in[0].gl_Position;
```

Don't simply just copy and paste this line. Retype it yourself and understand what each part of the expression is contributing.

Now we're ready to send this vertex on its way. At `TODO #D`, tell the geometry shader to ship the vertex down the pipe with the command `EmitVertex()`. Finally, at `TODO #E` tell the geometry shader to wrap up this primitive with the command `EndPrimitive()`.

At this point, we can rerun our program and we should now see some random dots floating around.



Hooray! Our first working geometry shader! It simply passes the points through. Let's start to manipulate our geometry.

Begin by changing `TODO #B` to output a max of 4 vertices. We're going to use our geometry shader to create geometry. Next begin by copying the line after `TODO #C` where you set `gl_Position` to sections `TODO #F`, `G`, `H` to set `gl_Position` a total of four times. Be sure to also call `EmitVertex()` after every setting of `gl_Position`. Let's rerun the program now. You should see the same thing you did before.

All four points are being drawn at the same location. We want to move them all slightly. Prior to the multiplication by the projection matrix, we need to translate each point slightly. We are working in eye space currently and this will give us instant billboarding (when we get to it). We want to shift each of our points in X and Y. We'll move the point by adding a vector (yay vector math) and then multiply by the matrix. The general form will look like

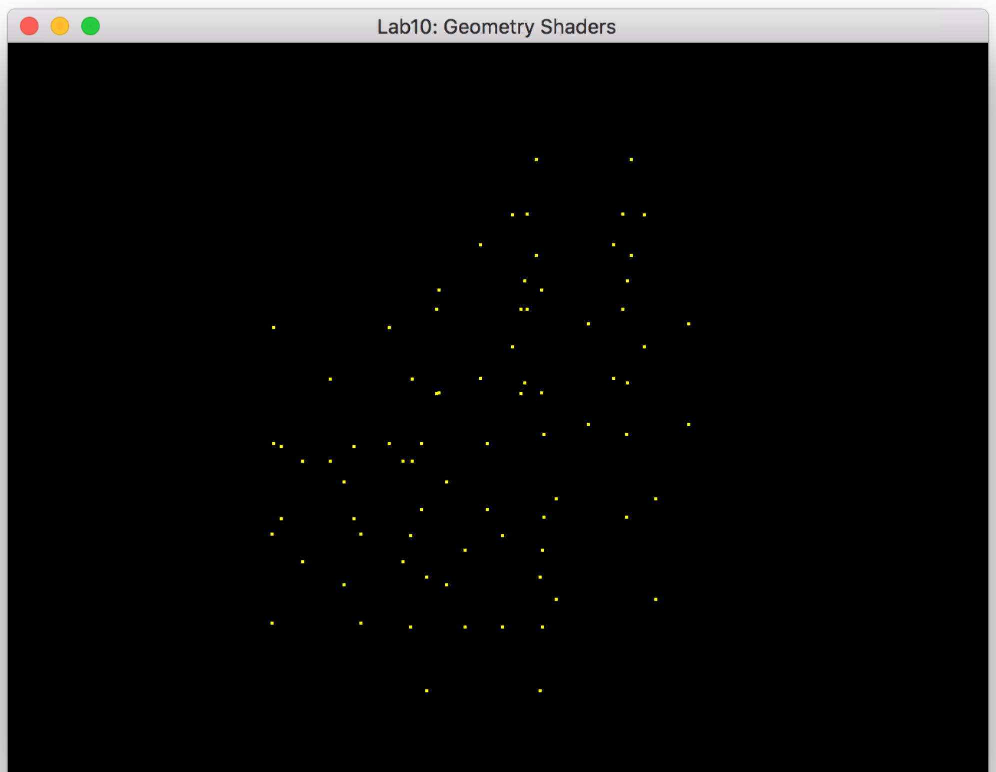
```
gl_Position = projMatrix * (gl_in[0].gl_Position + vector);
```

We'll need to replace the vector with the appropriate translation. We ultimately want to make a quad, so we'll want to have a quad centered at our current location. Therefore, prior to emitting each vertex we'll move it to a corner of our quad. The four corners will correspond to

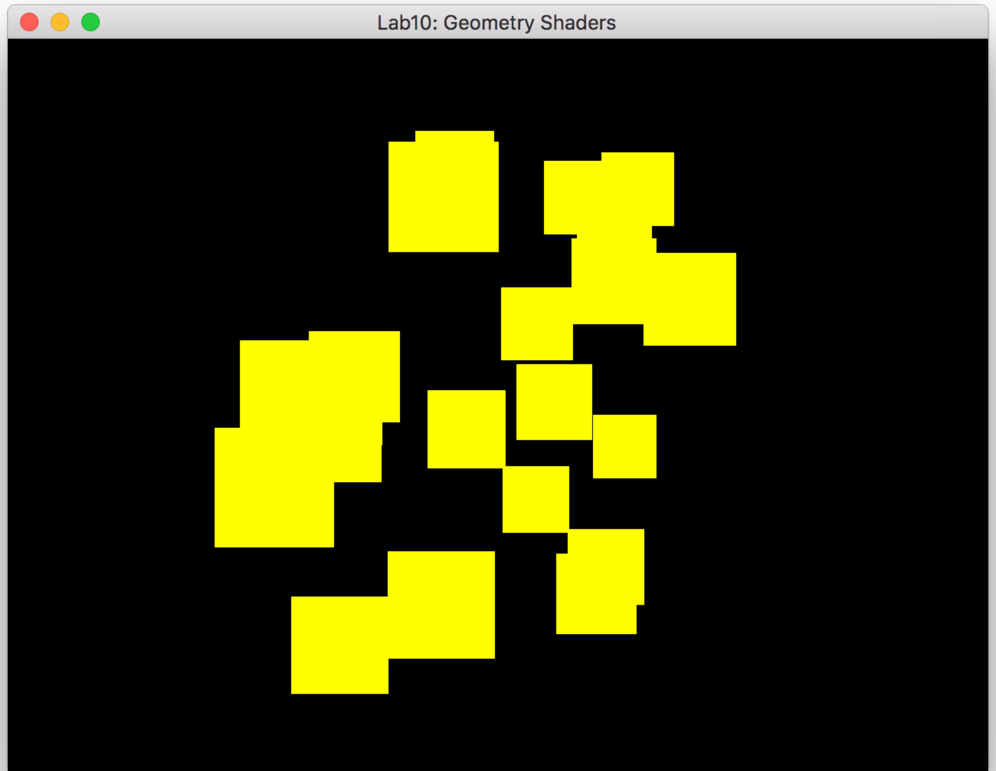
(-1, -1)
(-1, 1)
(1, -1)
(1, 1)

and we need to replicate that with each vector. Recall that `gl_Position` is a `vec4` so we need to add a `vec4`. X and Y will be set to above. We don't want to modify Z, so what should it's value be? This is also representing a vector, so what should W be?

When all four vertices are moved, rerun the program and now we should see 4X as many points as we did before!



And now for the Geometry Shader magik!! All we need to do is to change our output primitive from `points` to a `triangle_strip` and we'll begin making quads! If you emit your vertices in the correct order then you will see some beautiful looking quads, as below:



If you are seeing bowties or just triangles, then reorder the points you are emitting to follow proper CCW winding order for a triangle strip.

If we rotate our camera, then the quads rotate to always face us.

Q1: Why are they billboarding?

Let's make them look a little prettier. Switch back to `main.cpp` briefly. At `LOOKHERE #4`, we are binding a texture. You can follow it through from `setupTextures()` where we load in the image and assign the texture handle. We are binding a texture to make it active.

To render the texture we need to keep working in our shaders. The first step will be in the Geometry Shader. We are emitting vertex positions but no other attributes. We need to have texture coordinates specified somewhere so we know which texel to load.

Let's get the texture coordinate variable set up first. It will be created and output from the geometry shader, then read and used in the fragment shader. A varying! At `TODO #I`, add the output varying variable to the geometry shader. A texture coordinate is simply a two dimensional vector. Give it an appropriate name.

Now in the fragment shader at `TODO #J`, add the input varying variable ensuring the name and type match the output.

We're ready to go. Now in the geometry shader, prior to emitting each vertex we need to set a texture coordinate that corresponds to the vertex. The texture coordinates should range over

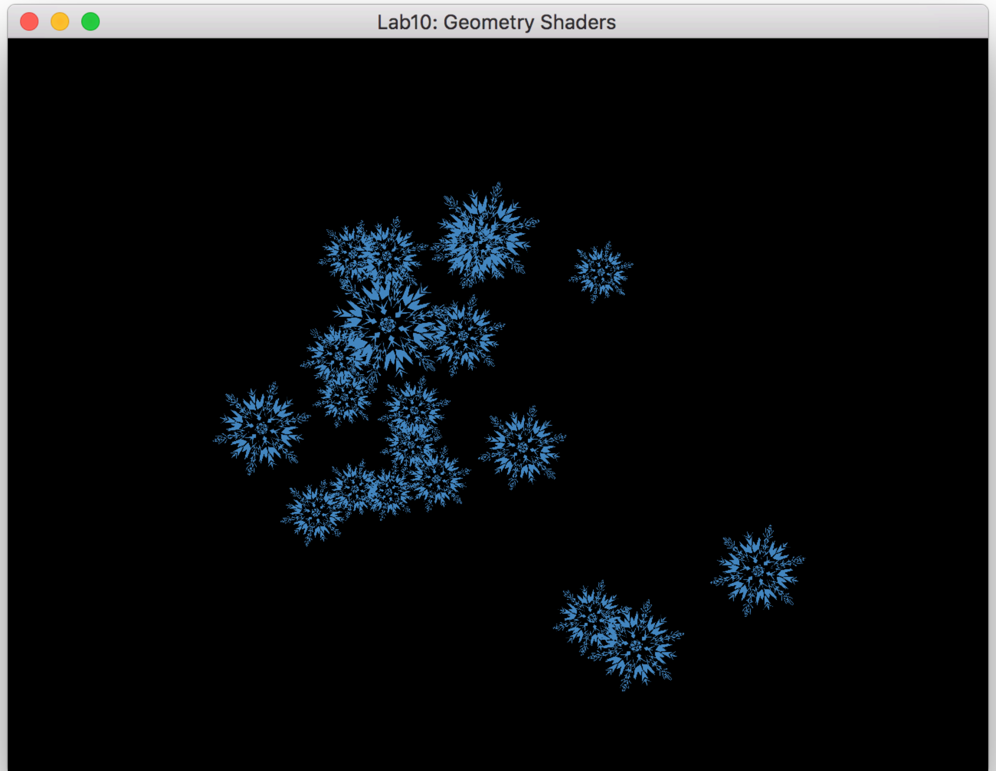
(0,0)
(1,0)
(0,1)
(1,1)

Make sure you are emitting the texture coordinate with the corresponding vertex so they span the correct order.

Now in the fragment shader we need to look up the texel. The first step, at `TODO #K`, is to get access to the bound texture. Shaders have a special type for 2D textures and that is `sampler2D`. Create a `uniform` of this type and give it an appropriate name (except for texture since that keyword is reserved as we'll see in a second).

Finally at `TODO #L`, we need to load the texel. GLSL has a special function called `texture()` that takes two arguments. The first is the `sampler2D` variable and the second is the texture coordinate. Set `fragColorOut` equal to the result of this function call.

And barring no typos, when we run our program we'll see the following!!



Hip hip hooray! Rotate around and enjoy the winter wonderland. Wait, something's not right? The snowflakes are occluding each other!

Q2: Why do we see the black of the quad blocking snowflakes that are behind the current one?

Until now, everything for this lab was dealing exclusively with shader programming and we didn't need to touch anything with OpenGL.

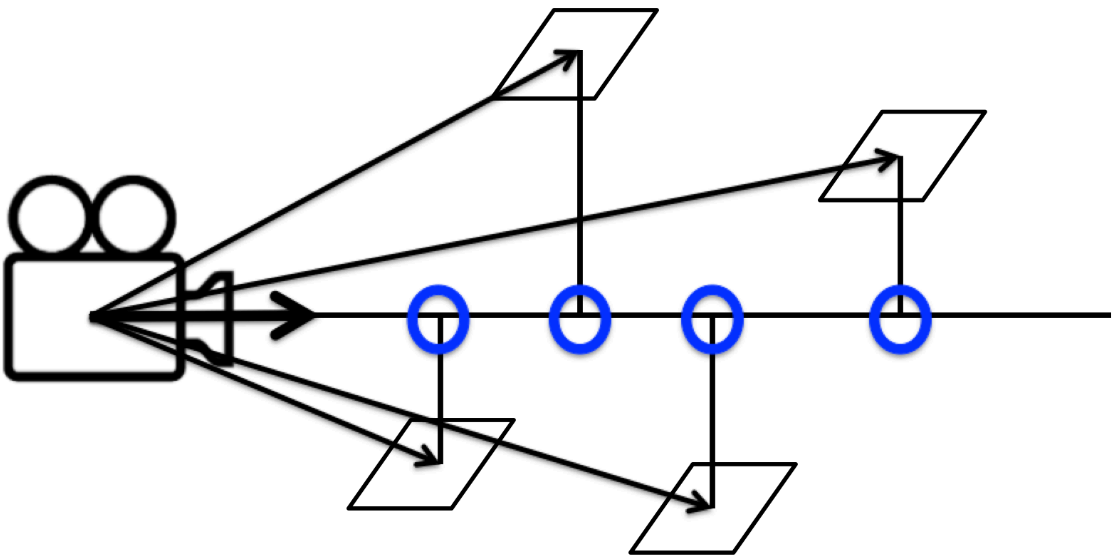
Q3: Why?

In order to properly render our transparent objects, we need to draw the back to front.

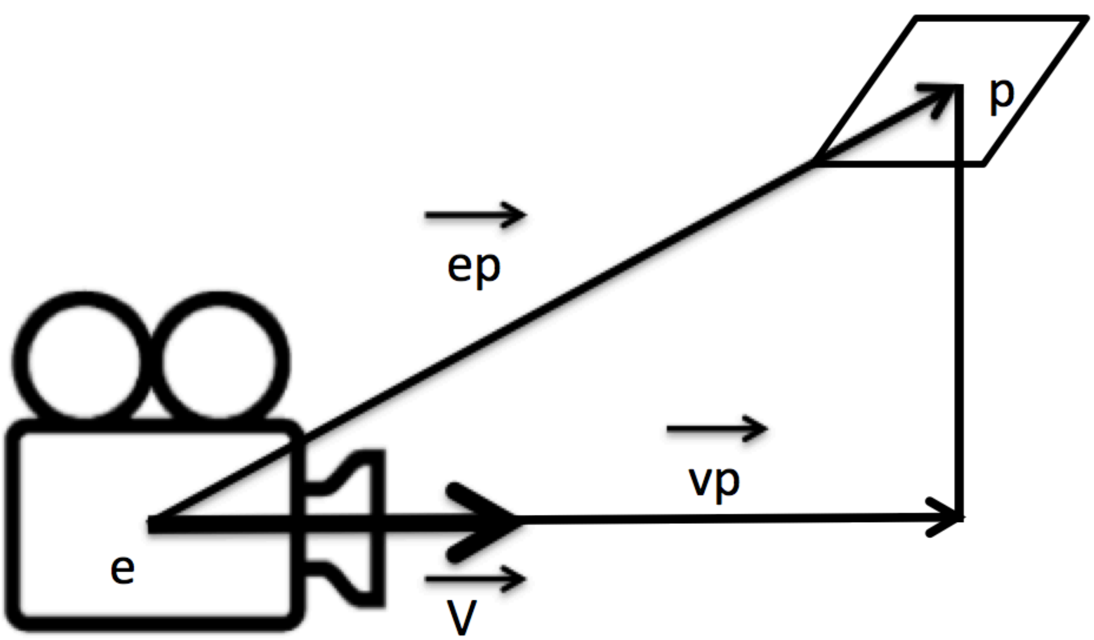
Q4: Why?

This means we need to reorder the vertex data before we send it to the GPU to draw. We will do this in two steps. First we will compute the distance to the camera for each point. Then we will sort the points by this distance.

In the image below, the circled points are the distances we need to sort by.



For an individual triangle, we have the following information:



`v` is our normalized view vector. At `TODO #1`, we will begin. Compute the normalized view vector based off of our `lookAtPoint` and our `eyePoint`.

Create two arrays, each with the size of NUM_POINTS. The first array will be of type int and will store our ordered indices (name it as such). The second array will store the corresponding distances and be of type double. Name it appropriately as well.

Now we are going to loop through each of our points stored in the points array. For each point, we first need to convert the point into world space. This means we will create a vec4 out of the ith point and multiply it by the model matrix. This will get us point p in world space. Our camera point, eyePoint, is already in world space. From these two points we can create the vector ep. As discussed in class, the length of vector vp gives us the distance to the point projected onto the view direction. This distance corresponds to the dot product between v and ep. Compute this distance aka dot product. (Hint: glm has a function called dot). We will now put into the ith position of the ordered indices array the value of i (our looping parameter) to correspond to the index of the point we just computed. Then put the computed distance into the ith position of the distance array.

Next we need to sort the distances and move the ordered indices as well. For simplicity, I'd recommend doing a bubble sort. Compare the ith and jth distances. If the ith distance is less than the jth distance, then swap the indices and distances for position i and j in each array.

You may want to verify that these two arrays are sorted in the proper order.

Once you have these properly sorted, create a new array of type Vertex (our special vertex struct) with NUM_POINTS elements. We'll now loop through our ordered indices and copy the correspond position into our new array. It should look like so

```
for( unsigned int i = 0; i < NUM_POINTS; i++ ) {
     orderedPoints[i] = points[ orderedIndices[i] ];
}
```

Perfect. Now at TODO #2, we need to bind our pointsVBO to the GL_ARRAY_BUFFER. And now we send over the ordered vertex data to the GPU. We will use the glBufferSubData() command for this. Recall the four arguments:

1. Target (GL_ARRAY_BUFFER)
2. Offset (0 – we want to start at the beginning)
3. Size (sizeof(orderedPoints))
4. The actual data (orderedPoints)

Compile. Run. Move the camera around. Voila! Magical transparent snowflakes.

Q5: Did this lab help clear up the confusion involving GLSL/GLEW and shaders? If not, what confusion remains?
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 source code, Makefile, screenshot, and README.txt with questions. Name the zip file <HeroName>_L10.zip. Upload this on to Canvas under the L10 section.

LAB IS DUE BY **FRIDAY NOVEMBER 9 11:59 PM**!!