CSCI 441 - Lab 11
Friday, November 13, 2015

LAB IS DUE BY **FRIDAY NOVEMBER 20 11:59 PM**!!

Today, we'll finally look at how to properly use shaders. We will look at the multitexturing example with the dirt pathway in a field of grass. This multitexturing technique is very useful for normal mapping, specular mapping, displacement mapping, etc.

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

Step 1 – The Old Boring Field

Type make and then run the lab. Everything will start. You should see a skybox and a boring plain grassy ground plane. Our goal is to add a dirt path through the grass. One option would be to increase the geometry of our ground and make the dirt section individual quads. But that is then rigid and if we want to reroute our path, we need to come up with a new set of vertices. Instead we will use a shader program to allow a grayscale texture dictate where our grass and dirt should blend together.

Let's look at how a shader program gets properly compiled. The existing code is there for you. Go through it step by step. Open the file `include/Shader_Utils.h`. At the bottom of the file is the function `createShader()` that handles the compiling and linking of our program. Here are the important steps:

1. `compileShader()` – takes a text file and a shader type. Sends the contents of the GLSL file to the GPU, staging it for compiling.

Q1: Inside the function `compileShader()`, we have the function `glShaderSource()`. What is the value of the third argument, `shaderString`, that gets passed to this function?

2. Still inside `compileShader()`, we actually call `glCompileShader()` which tells the GPU to go ahead and compile our shader. Note that at this point we check the log for an error during compilation and print it out if there is one. We will know at this point if there are any syntax errors within our shader. We then return the handle to our shader.

3. After we compile the vertex and fragment shaders, we now create our actual shader program. We need to attach the all the components of our shader program together (using `glAttachShader()`) and then we link the program. At this point, we now print the log for our entire shader program

to see if there were any linking errors. Errors you can expect at this stage are either:

a. An error if one of the shaders did not compile so the program cannot be linked
b. An error if the fragment shader reads a varying variable that the vertex shader did not right
c. A warning if the vertex shader writes to a varying variable and the fragment shader does not read it

4. At this point, if there are no errors your shader program is ready to go! The amount of debug information will vary from computer to computer, driver to driver, graphics card to graphics card. It seems the lab machines give nice output and verify a successful compilation and linkage. Now whenever we want to use our shader program, we only need the handle to the entire program – not the handles to our individual vertex or fragment shaders. If we have more than one shader program, then we would have more than one handle and we would choose which one to use. We can only run a single shader program at a time, so if we want two accomplish multiple effects we need to have a single program that encompasses all the effects.

Back in main.cpp, before we do any of our shader compilation we need to make sure we initialize GLEW. This is done through `glewInit()`. Two things can go wrong when you try to compile the shaders if you had not done the proper set up.

First, if you do not include `<GL/glew.h>` then at compile time, you will get an error that `glShaderSource()`, et al. cannot be found. `glew.h` redefines `gl.h` internally, hence the reason it needs to be included before `gl.h` This redefinition includes all these shader specific functions.

Second, if you include `glew.h` but do not call `glewInit()` then you will get a Seg Fault at runtime when you try to call any of the shader functions. `glewInit()` sets up your OpenGL context to include the necessary extensions to work with shaders and the GPU.

Let's now start with our GLSL code to create our shaders.

Step 2 – The Vertex Shader

Open `shaders/multitexturing.v.glsl`. Note one of our preprocessor directives

```
#version 120
```

This states what GLSL version the shader is written in. (As an aside: The most recent version is 430 (for 4.30) but we are sticking with version 1.20 and OpenGL 2.1 since it is the most accepted version. And future versions are backward compatible. GLSL 1.50 forces use to rewrite our entire OpenGL code and start using OpenGL 3.3. This rewrite involves passing every piece of information manually, including the ModelView matrix, Projection matrix, light information, etc. That doesn't provide us any real advantage at this stage. For the advanced graphics course, we would focus on OpenGL 3.3+ and ensure our hardware is up to date. Continuing on.)

The vertex shader has one main purpose that it must accomplish – we need to set gl_Position to our vertex in clip space. That means we need to do the transformation ourself. We have built in variables that we can use to do this calculation.

gl_ModelViewProjectionMatrix is a built-in uniform variable that contains our MVP matrix.

Q2: What is a uniform variable for GLSL?

gl_Vertex is a built-in attribute variable that contains our vertex location.

Q3: What is an attribute variable for GLSL?

Under Vertex Calculations, set gl_Position equal to the MVP matrix times the vertex. Order matters! The order in the sentence is the order you should set up your equation.

Now for the Texture Calculations. We eventually will have three textures tied to this quad. Texturing actually takes place in the Fragment Shader, so we need to pass through the texture coordinates and allow them to be interpolated for the Fragment Shader. Again, we will make use of the built-in variables.

gl_MultiTexCoord# (where # equals 0-7 or more depending on your system) is the built-in texture coordinate attribute for the # texture bound to this vertex.

gl_TexCoord[#] is the built-in texture coordinate varying variable for the # texture bound to this vertex.

Q4: What is a varying variable for GLSL?

We simply need to set gl_TexCoord[#] = gl_MultiTexCoord# for # equal to 0, 1, and 2.

Great our vertex shader is done. Onto the fragment shader!

Step 3 – The Fragment Shader

Open shaders/multitexturing.f.glsl. The Fragment Shader has one goal – to set the final RGBA value for this fragment. (Aside, it can also modify the fragment's depth because we have not gotten to the depth test yet, but there's no reason we'll do that.) For this Fragment Shader, the RGBA value will come from a combination of textures. So first let's get our textures.

The first thing we need to do is say what three textures we are using. In TODO Part I, we need to create three uniform variables – one for each of our textures. These uniform variables will have the special type sampler2D. Name each variable something appropriate to represent the grass texture, the dirt texture, and our dryness texture. The latter will dictate where is grass and where is dirt.

Now, let's get our texels from each texture. In the Texture Calculations section, we need to get a texel for each texture. This is accomplished through the function texture2D(sampler2D textureHandle, vec2 textureCoordinate). We need to pass in our texture handle (one of the sampler2D uniforms we created) and the texture coordinate (which comes interpolated from the vertex shader). This function then returns a vec4 variable that contains the RGBA information of the texel.

The texture coordinates are the varying variables we passed from the vertex shader. Recall: gl_TexCoord[0], 1, and 2. We will pass as the argument the s and t values. The texture handle is the name of the variable we set as a uniform earlier. So the full call will have the form:

```
vec4 texel = texture2D( texHandle, gl_TexCoord[0].st );
```

Get the texel for each texture. The texture coordinates map to the handles as follows:

TexCoord[0] – grass
TexCoord[1] – dirt
TexCoord[2] – dryness

Lastly, we will set gl_FragColor to the final RGBA value of this fragment. We are going to use the dryness texel value as our blending parameter, t. This t will control our lerping between the grass and dirt texels. From the slides, we can use the mix() function to lerp for us. The function has the form:

vec4 = mix(vec4 a, vec4 b, float t);

which then lerps some value t from a to b. We want to lerp from grass (a) to dirt (b) and use the dryness's red channel for t. Access the red channel by calling texel.r. Set the result of this mix() call to gl_FragColor and we're done! Let's head back to the comfort of C++ land to see if this compiles (it should).

Step 4 – Compiling the Shader

At TODO #1 in setupShaders(), we need to create our shader program. Use the function createShaderProgram() and pass in the filename of our vertex shader and fragment shader. Use the relative path from the root of the lab11 directory, so be sure to include shaders/ before the filename. This function returns a handle to our shader so set it to shaderProgramHandle.

Compile and run. If anything went wrong, you'll see output when the shader code gets compiled. If there is an error, modify the GLSL code and rerun the program.

Q5: Why do we not need to recompile the C++ code after we modify the GLSL code? Why are we allowed to rerun our C++ program to test new GLSL code?

Now we need to set up our uniform variables for this shader. This will go after TODO #2. First, let's tell OpenGL to use our shader. With the glUseProgram() function, pass our shaderProgramHandle. Now we need to find the location in memory of our uniform. We'll use the function glGetUniformLocation(). It takes two parameters, the handle of our shader program and the name of our uniform variable. This name must match exactly the name of the uniform variable we set up in our shader program, the fragment shader to be specific. This function returns a GLuint which is then the location of that uniform. Get the uniform location for the grass texture handle you created. Store this in some variable.

We will now set the uniform value. To set a value for a uniform, the function in this case is glUniform1i(). There are other forms of this function, just as there is with glVertex*(). This specific function takes one parameter of type int. We will pass to this function, the location of the grass texture uniform and then the numeric literal 0. This means our grass texture is the texture id 0.

Repeat the above two steps for the dirt and dryness uniform variables. The values for each uniform follow our mapping from before:

Grass – 0
Dirt – 1
Dryness – 2

Lastly, tell OpenGL not to use any shader program, so set glUseProgram() back to 0.

Step 5 – Bind Multiple Textures

Until this point, we could only have one texture bound at a time. We now want to have three textures applied to the same quad simultaneously. Luckily, OpenGL allows us to do this. We just need to turn on the other textures.

At TODO 3, we will go through and bind each texture. First, make sure we enable texturing (this line is there for you). Next, we will tell OpenGL which texture is the current active texture. OpenGL is a state machine, we can only work with one texture at a time, but we can set which texture we want to use. We will set the active texture with the command glActiveTexture(). We can pass a special parameter GL_TEXTURE# where # is the number texture we want to be the active one. We're seeing this a lot in all of our code. So we will set GL_TEXTURE0 to be active. NOW we'll bind out texture as we have done before. Bind the grass texture to Texture0.

Repeat this for the next two textures by making each texture number active and binding the appropriate texture. AGAIN, here is the mapping to use. We MUST be consistent.

0 – Grass
1 – Dirt
2 – Dryness

Go ahead and compile and run at this point.

Q6: Woah! What happened to our skybox?!

Well if we follow our chain of command, we last set Texture2 to be the active texture. In our skybox when we bind our skybox texture, we're binding it to Texture2. But our texture coordinates refer to Texture0, so we're seeing Texture0 every where! At TODO 3b, set Texture0 to be active again so our skybox uses the correct texture.

Compile, run.

Q7: Phew, better?

Ok, two more steps to go.

Step 6 – Using Each Texture

Currently when we draw our ground quad, we are passing the Texture Coordinate and by default this refers to Texture0. We need to pass in a texture coordinate for each of our three textures. We'll need to replace all of the glTexCoord2f() function calls, with a new method. This method is glMultiTexCoord2f() – hey this looks like what we're using in our vertex shader!

First at TODO 4, replace each glTexCoord2f() call with a glMultiTexCoord2f() call. This second function takes 3 parameters: the texture number to set the coordinate for and then the S & T values. Convert the glTexCoord2f() calls to be Texture0 (our grass texture) ranging from 0 to 1.

Now ADD before each vertex, texture coordinates for the other two textures, which also will range from 0 to 1.

Compile and run.

Q8: What do we see?

D'oh! We forgot to actually use the three textures in our rendering. At TODO 5, tell OpenGL to use our custom shader when rendering the quad.

Compile and run.

Q9: Does the ground look cool now? How about our skybox?

Finally, at TODO 5b tell OpenGL to not use any shader when we render the skybox.

Compile and run.

Q10: Everything hunky-dory?

There we have it! Let's do one last thing. Change the texture coordinates for our dirt texture to range from 0 to 5 now.

Q11: What should this do?

Compile and run.

Q12: What changed?

And we're done! Lots of reading, but actually little coding. I hope this helped. Use the remaining questions below to provide any additional feedback. It'll be very helpful for me when planning next year's course. Thanks for being guinea pigs and I

hope you got something positive out of this experience in trying to teach yourself the shader process.

Now that we can multitexture quads, we can do normal mapping, specular mapping, displacement mapping, etc. We'll see some more options coming up too.

Q13: Should that lab have been done sooner? (I think I know the consensus)
Q14: Did this lab clear up the confusion involving GLSL/GLEW and shaders? If not, what confusion remains?
Q15: Was this lab fun? 1-10 (1 least fun, 10 most fun)
Q16: How was the write-up for the lab? Too much hand holding? Too thorough? Too vague? Just right?
Q17: How long did this lab take you?
Q18: Any other comments?

To submit this lab, zip together your source code, Makefile, screenshot, and README.txt with questions. Name the zip file <HeroName>_L11.zip. Upload this on to Blackboard under the Lab11 section.

LAB IS DUE BY **FRIDAY NOVEMBER 20 11:59 PM**!!