Lab 10: Multiple Shader Programs

In this lab you will learn how to use multiple shader programs. To begin, download the Lab10.zip file from the Canvas site, and unzip it. This lab uses texture mapping (like Lab 9), so you will need to run a local webserver from the Lab 10 folder as you learned to do for Lab 9. As a quick refresher, you need to open a command prompt window, change to the Lab 10 folder, and then start the webserver. For me, I typed these commands in the command prompt window to start the Python webserver:

```
> cd C:\Users\dcliburn\Desktop\Teaching\COMP153\Labs\Fall2020Labs\Lab9
> python -m http.server
```

Next, you should open your web browser and type `http://localhost:8000/` in the address bar. This will show a list of contents in the directory from which the webserver is running. Click on Lab10.html. The program should now run now, and you should see many familiar objects and images from previous labs. You can move around in the environment using the arrow keys (up/down to move forward/back and left/right to turn left/right). Explore the environment and get a good look at all of its features.

Three different vertex and fragment shader pairs are being used in this program. All three are defined in Lab10.html. The first, the "phong tex shaders", implements the Phong Reflection Model and multiplies a texture by each fragment's color to determine the ambient and diffuse color before lighting calculations are applied (similar to the shader used in Lab 9). If you travel around the environment you should notice that a specular highlight is clearly visible on the glacier, Yosemite, and tiger images from some locations in the scene. The second shader is the "toon shader" and it is used to illuminate the ground, walls, cube, and pyramid. Toon shading is an alternative lighting technique that applies a light's effects to an object in discrete intervals based on the angle between the surface normal and the light, and it does not produce any specular highlights. The image below shows the effects of toon shading applied to a teapot model to give it a "cartoonish" look.

The final shader implements “point sprites”. A **point sprite** has an image applied to a point in the scene rather than a color. If you increase the size of the point (using `gl_PointSize` in the vertex shader) then the size of the image will increase as well. The advantage of using point sprites over creating a polygon with a texture applied to it (as is done by the “phong tex shader”) is that point sprites will always appear to face the viewer generating the illusion that an object is three dimensional when it is only two dimensional. An entity that always turns to face the user is known as a **billboard** in computer graphics and can be implemented in multiple ways (one of which is through point sprites). There are two point sprites in the scene, the star in the middle of the environment and the tree on top of the cube. You have probably noticed that the star point sprite seems to move about the scene with you. However, it is in a fixed position, it just does not get larger or smaller as you move closer or get further away. Find the `//TODO:` comment in the point sprite vertex shader (near the bottom of Lab10.html) and follow the instructions there to scale the size of the point sprite by the distance from the viewer to create a more realistic look. Note that this code comes from page 347 of the OpenGL Programming Guide (8th edition). After making this change the point sprite looks better, but it is still very noticeable that the images are squares; the star and tree are both surrounded by black pixels. Wouldn’t it be nice if the black pixels were transparent to make the images look more like a real star and tree? Find the `//TODO:` comment in the point sprite fragment shader and follow the instructions there to `discard` fragments where the texture is black (or very close to black). After making these changes it should look a lot more like a star is floating in the middle of the environment, and a small tree is on top of the cube. Remember that if your browser does not seem to be loading the latest versions of the code files, make sure you have saved them and then, at least in Chrome, you can press **CTRL-F5** to force the page to reload the latest versions of the files.

Now we are going to take a closer look at the JavaScript files used in this example. As mentioned earlier there are three vertex and fragment shader pairs defined in Lab10.html. All shaders are loaded, compiled, and linked using separate program variables in `initPrograms` of model.js, however, only one shader program can be active at a time in WebGL. A shader is made active by passing its program variable to `gl.useProgram`. The most challenging thing about using multiple shader programs is making sure that necessary uniform variables are correctly bound to the current active shader. For example, consider the call `modelMatrixLoc = gl.getUniformLocation(program1, "modelMatrix");`. This binds `modelMatrixLoc` to the correct location of the variable `modelMatrix` in `program1`. Calls to the function `gl.uniformMatrix4fv(modelMatrixLoc, false, model_matrix);` will correctly assign a value to `modelMatrix` in `program1`. However, consider what happens if you later call `gl.useProgram(program2);`. Subsequent calls to `gl.uniformMatrix4fv` will continue to assign values to the `modelMatrix` variable in `program1`, not the one in `program2`, unless you bind `modelMatrixLoc` to the `modelMatrix` variable in `program2` using the function `gl.getUniformLocation`. This can get very complicated if you have lots of uniform variables in each shader, and you have to rebind and reassign values to the uniforms in each shader each time you make a new shader active. The correct solution for this is to use **uniform buffer objects** which allow you to create a buffer of uniform variables that can be assigned to once by your WebGL program, and then read by each shader. However, while our textbook has a section on uniform buffer objects, it contains no code examples illustrating their usage. Furthermore, some of the webpages I found describing functions used with WebGL uniform buffer objects list them as “experimental technology” (see https://developer.mozilla.org/en-US/docs/Web/API/WebGL2RenderingContext/bindBufferRange and https://developer.mozilla.org/en-US/docs/Web/API/WebGL2RenderingContext/getUniformBlockIndex as examples). Thus, I decided not to

use uniform buffer objects in this lab. Instead, I created my own function, `changeShaderProgram`, to help you quickly set the active shader program, bind uniform variables to it, and assign appropriate values to the uniforms. If you look at the `drawModel` function of model.js you should notice that the generalized process is to call `changeShaderProgram` to activate a shader, then position and draw all of the objects that should be rendered using that shader, and then call `changeShaderProgram` again to activate a different shader.

You have probably noticed that there are a lot more JavaScript files in this lab than in the past. Since I have many different types of objects in this lab, I tried to modularize the code somewhat to keep model.js from becoming hundreds of lines of code long. There are separate JavaScript files for many of the types of entities rendered in the scene (cube.js, ground.js, pyramid.js, and texsquare.js). Each of these files is loaded near the top of Lab10.html. If you open each file in your code editor, you will notice that each file defines functions to *init* (initialize/create buffers for) the entity and to *draw* the entity. The *init* functions are called in `initBuffers` of model.js and the *draw* functions are all called in `drawModel` of model.js. You can easily create your own JavaScript files to define additional entities to add to the environment by following this format. Try to add your own model to the environment to be rendered and shaded using the toon shader. You should create your own JavaScript file with appropriate variables and *init* and *draw* functions. Perhaps you could use the model someone in your team created for Assignment 3 (which should already have surface normals defined). Make sure you load the JavaScript file in Lab10.html, call your model's *init* function in `initBuffers` of model.js, and call the *draw* function in `drawModel` of model.js (find `//TODO 3:`).

Next, add your own texsquare entity to the environment. To do this you will need to load an image to use as the texture at the top of Lab10.html. You will then need to create a variable to use to reference the texture at the top of model.js and initialize the texture so that WebGL can use it in `initTextures` of model.js. Finally, you will need to position and draw the texsquare in `drawModel` of model.js (find `//TODO 4:`).

Finally, add your own point sprite to the environment. To do this you will need to first create your own image that has a black background that will be made transparent by the fragment shader. Here are some steps you can follow to create an appropriate image if you are running Windows.

1) Open MS Paint.
2) Click on the Edit Colors icon (right of the screen) and define a custom color with all values of zero (completely black) – click OK.
3) Click on the "Fill with Color" icon (looks like a can of paint with red paint pouring out). Now, click in the image and the background will be set to completely black.
4) Draw your image using colors other than completely black. NOTE: If you want to use black, then you can define a custom color with values of 40,40,40 for Read, Green and Blue. This will appear to be very dark gray (almost black) but will not cause the fragment to make it transparent.
5) When you are done drawing your image, choose "Save as", give your image a name, make sure you put the image in the Lab 10 folder, and then click Save. You have now made an image that the program can load and use for a point sprite.

Once you have your image created, you will need to load it at the top of Lab10.html, create a variable to use to reference the texture at the top of model.js, and initialize the texture so that WebGL can use it in

`initTextures` of model.js. Finally, near the bottom of `drawModel` in model.js, bind your texture, and then position and draw a point sprite using your own image as the texture (find `//TODO 5:`).

Get started on Assignment 7 when you finish this lab. **Remember that you can type `Ctrl-c` in the command prompt window to terminate the webserver when you are done.**