Follow the instructions carefully. If you encounter any problems in the setup, please do not hesitate to reach out to CA’s on Piazza or come to office hours.

1 Background

In assignment, you will familiarize yourself with the OpenGL API used by the framework. Instead of using the framework classes directly, you will learn and implement some of what they are doing under-the-hood. This assignment will aim to get you comfortable with sending geometry to the OpenGL pipeline as well as using shaders to render and animate that geometry on the screen. It is recommended, though not required, that you also try to find the classes that use these functions so that you also get an understanding of what some of the framework classes do. However, try to avoid copy and pasting from the framework code directly as writing it yourself will greatly help you in learning the OpenGL API! If at any point you want to learn more about any of these steps, check out the OpenGL Programming Guide (the red book).

2 Update Your Code

Assuming you followed the instructions to download the code from Assignment 1, to get the code for assignment 2 all you have to do is run:

```
git fetch source
git merge source/master
```

Afterwards, go to `main.cpp` and change the line that says:

```
#define APPLICATION Assignment1
```

to

```
#define APPLICATION Assignment2
```

3 Basic Shader Setup

3.1 Create Shaders

Initial shaders are given to you in the “shaders/hw2” directory: “hw2.vert” and “hw2.frag.” The only requirement is that the shaders be located under the “shader” directory.
3.2 Load and Compile Shaders

We will load and compile the shaders inside the “Assignment2::SetupExample1” function. First, we will load the shader data into memory. Make sure you do the next step for BOTH the vertex and the fragment shader (store the data in two separate strings).

1. Construct the full filepath to the shader files. We provide a macro “SHADER_PATH” that contains the full path of the “shaders” folder. To get the full path of “shaders/hw2/hw2.vert” in C++, you can just create a string as follows:

```cpp
const std::string completeFilename = std::string(STRINGIFY(SHADER_PATH)) + "/hw2/hw2.vert";
```

2. Read the file data into a string. For example, you could use a std::ifstream to read the file data into a string.

Next, we will compile the vertex and fragment shaders. Do the following step for EACH shader:

1. Use glCreateShader to create the appropriate typed shader (vertex or fragment). To be consistent with the rest of the framework, use the “*.vert” extension to denote vertex shaders and the “*.frag” extension to denote fragment shaders. This function will return a GLuint—make sure you store the result for the vertex and fragment shader separately (these values will be used in the checkpoint).
2. Use glShaderSource to tell OpenGL what the contents of each shader is. This needs to be done separately for both the vertex and fragment shaders.
3. Use glCompileShader to compile both your vertex and fragment shader.

At this point you should have two GLuints: one for the vertex shader and one for the fragment shader. Next you will need to tell OpenGL to combine these two shaders together to create a shader program. A shader program is what is used to render a piece of geometry. You will need to do the following steps:

1. Create a shader program using glCreateProgram. This function returns a GLuint—**make sure you store the return value as a member variable in the Assignment2 class**.
2. Now attach both your vertex and fragment shader to the shader program using the glAttachShader function. The program id is the value returned by glCreateProgram and the shader id is the vertex/fragment values returned by glCreateShader.
3. Finally, link the shader program using glLinkProgram.

Desperate for help? Check out “common/Rendering/Shaders/ShaderProgram.cpp.”

3.3 Checkpoint

The following piece of code should be UNDER (but in the same function as) the code you wrote for this section.

```cpp
const GLuint vertexShaderId = 0;
const GLuint fragmentShaderId = 0;
const GLuint shaderProgramId = 0;
```

Replace the zeroes with the proper IDs. For the vertex/fragment shader ID, replace them with the values returned by glCreateShader. For the shader program ID, replace it with the value returned by glCreateProgram.

If you see “SUCCESS: Checkpoint 1 completed.” then you have finished this checkpoint correctly.

4 Geometry

4.1 Setup Buffers

Modern OpenGL beginning with OpenGL 3.0 began using buffers to send vertex data to the GPU. This reduces the amount of overhead on the CPU and allows for faster performance (to put it simply). These
buffers are stored within a vertex array object which are necessary so that the OpenGL shader knows what
data is what. In this section we will create the vertex array object as well as the buffer to store the vertex
position data for the given triangles. We will setup the geometry inside the “Assignment2::SetupExample1”
function.

1. Generate ONE vertex array object ID using glGenVertexArrays. You can pass a pointer to a GLuint to
the arrays parameter. For example:

```c
GLuint vao;
glGenVertexArrays(1, &vao);
```

Make sure you store the vertex array object ID in the Assignment2 class as a member variable.

2. Bind the vertex array object using glBindVertexArray.

3. Generate a buffer ID for the vertex positions using glGenBuffers.

4. Bind the buffer using glBindBuffer. For target use GL_ARRAY_BUFFER.

5. Pass the vertex position data to the buffer using glBufferData. For target use GL_ARRAY_BUFFER.
For data, use &vertexPositions[0]. For usage, use GL_STATIC_DRAW. What should be used for size
is left as an exercise for the reader.

6. Use glVertexAttribPointer to let OpenGL how the currently bound buffer’s data should be used. index
for now should be 0. This corresponds to the “layout (location = 0)” inside the vertex shader. size
should be 4 (we are using vec4’s for position). type should be GL_FLOAT. normalized should be
GL_FALSE. stride and pointer should both be 0.

7. Use glEnableVertexAttribArray to make the current buffer object be passed to OpenGL along with the
current vertex array object. index should be the same as in the previous step.

Need help? Check out “common/Rendering/RenderingObject.cpp.”

4.2 Send Buffers to the GPU

In every frame, one needs to tell OpenGL what to draw and how to draw it. We will do this in the
“Assignment2::Tick” function. This is fairly simple. All we have to do is tell OpenGL to use the right shader
program and to bind the right vertex array object. Afterwards, we just call the right draw command and
voilá! Stuff is on the screen!

1. Use the shader program using glUseProgram. program should be the shader program ID you generated
earlier using glCreateProgram.

2. Bind the vertex array object using glBindVertexArray. The array should be the vertex array object
ID you generated earlier using glGenVertexArrays. For this assignment, this part is technically
unnecessary, but in real code you would want to bind the vertex array object again because there will be
more than one vertex array object that you need to bind.

3. Send the draw command to OpenGL using glDrawArrays. mode should be GL_TRIANGLES. first
should be 0. count is the number of vertices you are sending.

4.3 Checkpoint

At this point, you should see the triangles from assignment 1 but they should be all white!

5 Slightly-More Advanced Shaders

In this section, if you want to find a GLSL function, you can look at the documentation here. The GLSL
functions are those that are not prefixed by gl. If you want to learn more about shader programming with
GLSL this tutorial is nice and the OpenGL Programming Guide is also a nice reference. There are also
shaders that are provided along with the in-class framework in the shaders directory. Feel free to look at
those as well.
5.1 Vertex Shader

The vertex shader is used to transform your 3-D geometry into normalized device coordinates (NDC). We set vertex positions in normalized coordinates so you do not have to worry about performing transformations in the vertex shader. For this assignment, we will move the triangles around the screen over time. This part of the assignment will introduce you to the concept of shader uniforms. Let us make the triangles move up and down the screen.

1. First, introduce a “time” floating-point member variable into the Assignment2 class. Be sure to initialize this to 0 in the constructor.
2. Inside the “Assignment2::Tick” function, increase the “time” by “deltaTime.”
3. Now we want to let the shader know about the “time” variable so inside “hw2.vert” create a uniform variable like so:

   ```
   uniform float inputTime;
   ```

4. Now use this “inputTime” variable to modify the position of the vertex so that the triangles go up and down the screen (note: the position of the vertex in NDC coordinates gets written to “gl_Position”). Hint: going down is going in the negative direction along the Y-axis. Be aware that you are only changing the position of the vertex as seen by the rest of the OpenGL pipeline and NOT the actual data. The next frame will still receive the original vertex location data.
5. Now you need to somehow connect the CPU “time” to the GPU “inputTime.” Back in the “Assignment2::Tick” function, you need to figure out the location of the shader uniform using the glGetUniformLocation function. Note that this should go after you call glUseProgram. name should be the name of your uniform variable—in this case “inputTime”.
6. Next, you need to take the location returned by glGetUniformLocation and call glUniform1f to set the “inputTime” uniform variable in the shader.

At the end of this section you should see your triangles move up and down the screen. If you need help setting shader uniforms see “common/Rendering/Shaders/ShaderProgram.cpp.”

5.2 Fragment Shader

In this section, you will make the color of the triangles change over time in the fragment shader. This will be left for you as an exercise. Feel free to do whatever you want on the CPU or on the GPU to accomplish this. OpenGL shaders can also pass interpolated data from the vertex shader to the fragment shader—look at “shaders/basicColor/basicColor.vert” and “shaders/basicColor/basicColor.frag” for how to do this. As a hint, note that setting a uniform on a shader program makes that uniform available on both the vertex and the fragment shader. At the end of this section, you should see your triangles change color over time.

6 Grading

This assignment will be graded on the following tasks:

- Complete section 3. Output that the shader program compiles fine.
- Complete section 4. Make sure the triangles are visible on screen.
- Complete section 5.1. Make sure the geometry smoothly transforms over time.
- Complete section 5.2. Make sure the color of the surface changes smoothly transforms over time.

According to the following rubric:

- 4 – Complete all 4 tasks.
- 3 – Complete 3 of the 4 tasks.
- 2 – Complete 2 of the 4 tasks.
• 1 – Complete 1 of the 4 tasks.
• 0 – Nothing to show.