Follow the instructions carefully. If you encounter any problems in the setup, please do not hesitate to reach out to CAs 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 highly 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). This homework requires you to spend time reading the linked OpenGL documentation. Read the provided links (links are blue) carefully before asking questions on Piazza.

2 The Graphics Pipeline (Optional Reading)

![3D Graphics Rendering Pipeline](image)

3D Graphics Rendering Pipeline: Output of one stage is fed as input of the next stage. A vertex has attributes such as (x, y, z) position, color (RGB or RGBA), vertex-normal (nv, nv, nv), and texture. A primitive is made up of one or more vertices. The rasterizer raster-scans each primitive to produce a set of grid-aligned fragments, by interpolating the vertices.

Figure 1: A visual description of the graphics pipeline. Source.
“graphics pipeline.” How you interact with and the amount of control you have over the pipeline has changed since OpenGL’s inception. In this class, we will focus on the modern programmable graphics pipeline. There still exist some fixed-function parts of the pipeline (e.g. parts of the pipeline where you can specify which function to use but not write your own function); however, we will gloss over them in this document. If you are interested, you can read more about how/why things like depth testing and stencil testing are used.

A brief overview of the graphics pipeline can seen in Figure 1. While it is somewhat simplified, it is sufficient to cover the parts of the graphics pipeline you will be seeing in this class.

The “raw vertices & primitives” are generally triangles that you wish to display on the screen. You pass all this data to OpenGL and the underlying driver will transfer all this data to the GPU for you (at some point). However, you can’t just pass a raw pointer pointing to whatever data you want and expect OpenGL to know what it means and how to use it. As a result, you will have to use a “vertex array object” (VAO) to keep track of what is what. However, by themselves, VAOs do not contain data. This is why you then need to create buffers to store your vertex data. When storing vertex data, these buffers are referred to as vertex buffer objects (VBOs). You then connect the VBOs to the VAO so that OpenGL knows what is what. You can use these VBOs to pass per-vertex information down the pipeline. While in general you use the VBOs to store vertex positions, there is technically no strict requirement to actually do so. You can also store things like vertex normals and vertex colors in VBOs as well. You will do these steps in Section 5.

Next, OpenGL will run a “vertex shader” (vertex processor in the figure) on every vertex that you passed in. You generally use the vertex shader to transform the vertex from world space to normalized device coordinates which is the format that the rest of the OpenGL pipeline is expecting it to be in. You can also do more general per-vertex computations here as well and OpenGL will automatically interpolate these values across the input triangles before making it available to the fragment shader (more on that later). This is great! However, the one caveat is that since you are able to program these vertex shaders to perform arbitrary computations, you must compile the vertex shader to run on your GPU. You will compile the vertex shader in Section 4 and actually use/modify a vertex shader in Section 6.1.

At this point, OpenGL will “rasterize” your triangles onto the screen and for each triangle a set of “fragments” are generated. While not technically accurate, it may be helpful to think of “fragments” as pixels (in DirectX, fragment shaders are referred to as pixel shaders). Each fragment then gets processed by a “fragment shader” which, like the vertex shader, can be programmed to perform arbitrary calculations. It is here that you generally perform shading/lighting/texturing computations. Much like before, you have to compile your fragment shader to run on your GPU. You will compile the vertex shader in Section 4 and actually use/modify a fragment shader in Section 6.2. At this point, it is important to note that while you write/compile the vertex and fragment shaders separately, you actually have to “link” the two together into a “shader program” before OpenGL can use them. Any time that you want to use a vertex shader with a different fragment shader (or vice versa) you have to create a new shader program. In general, you want to reduce the number of shader programs you have and use as switching between OpenGL shader programs does incur some non-trivial overhead. This overhead won’t matter for this class but something to keep in mind if decide to pursue real-time rendering any further!

Finally, OpenGL will take all the fragments and combine them into the pixel values that get displayed on your screen. This is where some fixed-function pipeline operations such as the depth test and color blending come into play. You can see where these operations get set up in “MediaLayer::InitializeOpenGL.”

## 3 Update Your Code

Go to `main.cpp` and change the line that says:

```cpp
#define APPLICATION Assignment1
```

to

```cpp
#define APPLICATION Assignment2
```
4 Basic Shader Setup

4.1 Create Shaders

Initial shaders are given to you in the “shaders/hw2” directory: “hw2.vert” and “hw2.frag.” All shaders must be located under the “shader” directory.

4.2 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 (“hw2.vert”) and fragment shaders (“hw2.frag”). Do the following step for EACH shader:

1. Use glCreateShader to create the appropriate typed shader (vertex or fragment). This function will return a GLuint—store the returned values for the vertex shader and fragment shader in the provided “vertexShaderId” and “fragmentShaderId” respectively. Note: the framework uses the “*.vert” extension to denote vertex shaders and the “*.frag” extension to denote fragment shaders.
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, very simply, 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 (you need to create the variable).
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.”

4.3 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.

5 Geometry

5.1 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 is 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 set up 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`.


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 you to figure out.

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.”

5.2 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 voila! 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.
5.3 Checkpoint

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

6 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.

6.1 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:

```glsl
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.”

6.2 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.
7 Grading

This assignment will be graded on the following tasks:

- Complete section 4. Output that the shader program compiles fine.
- Complete section 5. Make sure the triangles are visible on screen.
- Complete section 6.1. Make sure the geometry smoothly transforms over time.
- Complete section 6.2. Make sure the color of the surface changes smoothly 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.