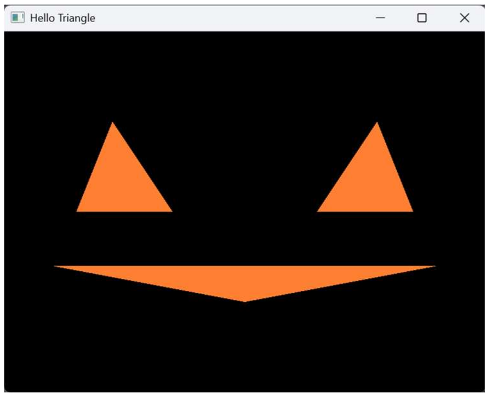
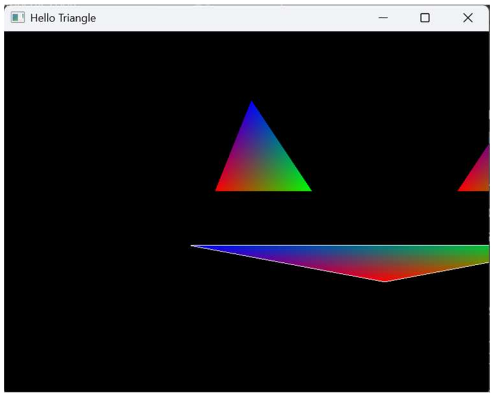
ICE 4.1: Shader Class, Uniforms, and Vertex Attributes

This ICE extends from ICE 3.1 by incorporating a Shader class manages shader program compilation and linking.

This ICE assumes you have worked through the Hello Triangle and Shaders tutorials available at www.learnopengl.com.





1. Download the Shader.hpp and Shader.cpp files from the Lesson Materials and add them to a copy of your ICE 3.1 solution (the ICE 3.1 solution is in the Lesson 3 module).
2. Refactor your code to make use of the new shader class.
    a. Extract the vertex shader source and fragment shader source from your main program and save them as individual glsl files (see reading for the formatting…hint…ensure you remove all the quotes and newline characters). The file names may be vertex.glsl and fragment.glsl (you can name them whatever you want). Place them in their own shader folder to keep your project neat.
    b. Examine the Shader.hpp and Shader.cpp files. Note that the constructor takes the path to your two files (e.g., "./Shaders/vertex.glsl", "./Shaders/fragment.glsl") as input, reads in the files, compiles the shaders, and builds the shader program. The identity of the shader program is stored as the shader's ID data member.
    c. Remove the original shader program code from your main program (shader source, shader compilation, shader program creation/linkage).
    d. Create an object of the Shader class using the constructor.
    e. Inside the render loop, you will need to pass the shader object's ID to your shape's draw member function.
3. Confirm that your code produces the original image using the new shader class. Now you can easily create multiple shader programs (to handle different types of data and shapes).
4. Use a uniform to change the color of one of your shapes (note the useful Shader::setVec4 member function). You will need to modify the fragment shader to do this. To turn the color on and off easily, you can add a second Boolean uniform to use, or not use, the color uniform.
5. Use another uniform to manipulate the location of all your vertices. Create an offset vector in your main program, set it using the Shader::setVec4 member function, then in your vertex Shader add the x and y values to your vertices (for example adding a certain amount to the x and y value and changing it in the render loop). This is an inefficient method of moving objects around the screen, but effective. Try altering this using the keyboard input for up, down, left,

and/or right (your computer is fast…move it by 0.01 each time you press the key…you want it to keep moving while you hold the key down).

6. Use per-vertex colors. Add three float values to each vertex in your shapes for new RGB values (e.g., 1.0,0.0,0.0 for bright red).
   - **a.** You will need to modify your VAO attributes so that it can interpret 6 values per vertex with the first three being position coordinates and the next three being color components. With our implementation of the VAO structure, you will need to push two AttributePointer structures now onto the VAO structure's vector of AttributePointers. If you define different colors for each vertex, the colors will be mixed (remember, the offset and stride are both in BYTES).
   - **b.** You will need to modify the vertex shader to accept 3 more values (location=1) and provide an output (e.g., out vec4 vertex_color), then set your output value to the color you take in for each vertex.
   - **c.** You will also need to modify the fragment shader to input the output from the vertex shader (MUST be the SAME NAME) and then set that to the FragColor.
7. **CHALLENGE:** You can use the EBO to draw an outline on a shape (draw the shape first as normal, then draw the shape again using a line loop. You may want to alter the line thickness with a call to glLineWidth(3.0) (just make sure you change it back to 1.0 after you draw the outline).