# Rendering with the graphics pipeline

## Introduction

By the end of this session a student will be able to:

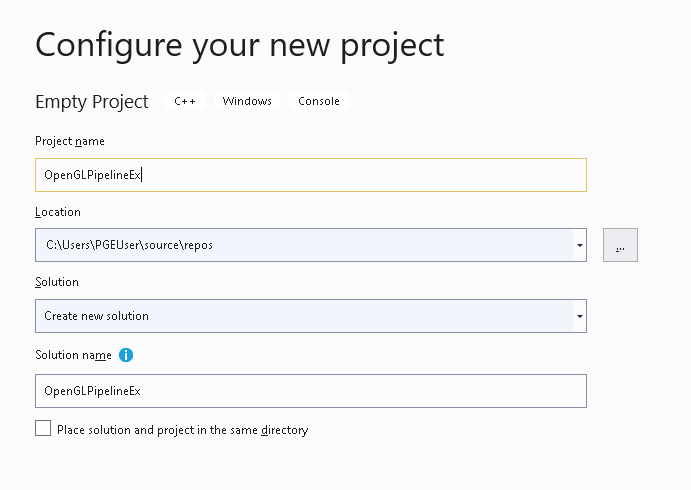
1. Define shader code, create and compile shader code, and create a program for shaders.
2. Define vertices that make up a shape in C++ code.
3. Create a vertex buffer object into which the vertices defined in C++ code are transferred to graphics memory to be rendered.
4. Render the shape using the vertex buffer object and program.

This tutorial assumes you are using Visual Studio 2019.

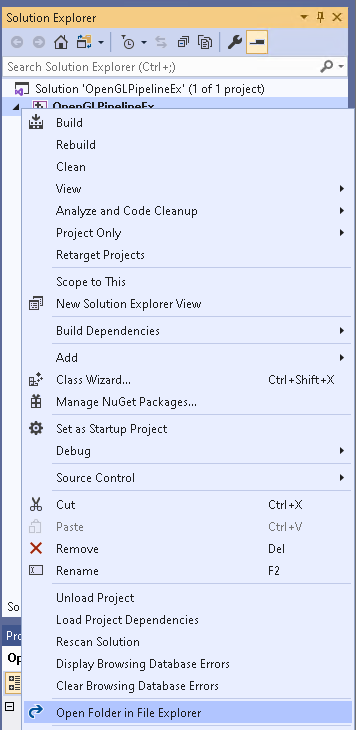
## Task – Setup project

For this task:

1. Create a new Empty C++ project, call it OpenGLPipelineEx..

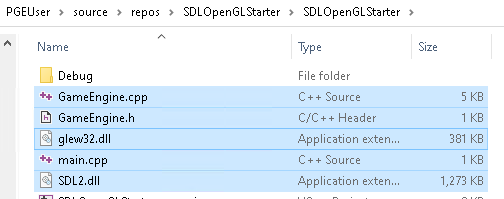


1. Add the SDL and GLEW libraries to the VC++ Directories and the .lib files to the Linker > Input > Additional Dependencies option. This is described on pages 6 to 10 of the “Getting Started” exericse.
2. Copy the GameEngine.h, GameEngine.cpp and main.cpp files from the “Getting Started” exercise to the project folder, and the SDL2.dll and GLEW32.dll files from the same folder. To do this, first, right click on the project name in the Solution Explorer (project name is the second item in the Solution Explorer) and select “Open Folder in File Explorer”.

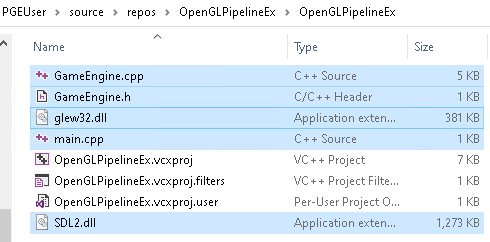


Right click here

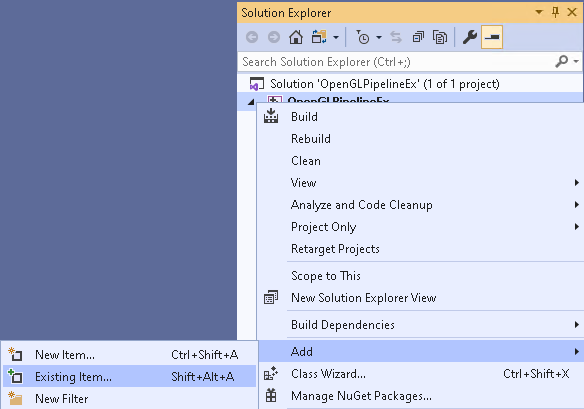
1. A file explorer will open. Now copy the GameEngine.h, GameEngine.cpp and main.cpp from the existing project folder to the new project folder. Source:



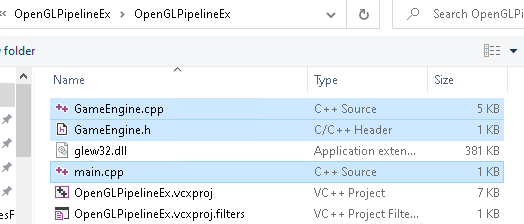
Destination:



1. Next, we need to add the source code files to the project. Right click on the project name and this time select Add, then existing item:



1. A file explorer window will open. Press the CTRL key and click on GameEngine.cpp, GameEngine.h and main.cpp to select them. Click OK when you’ve done this.

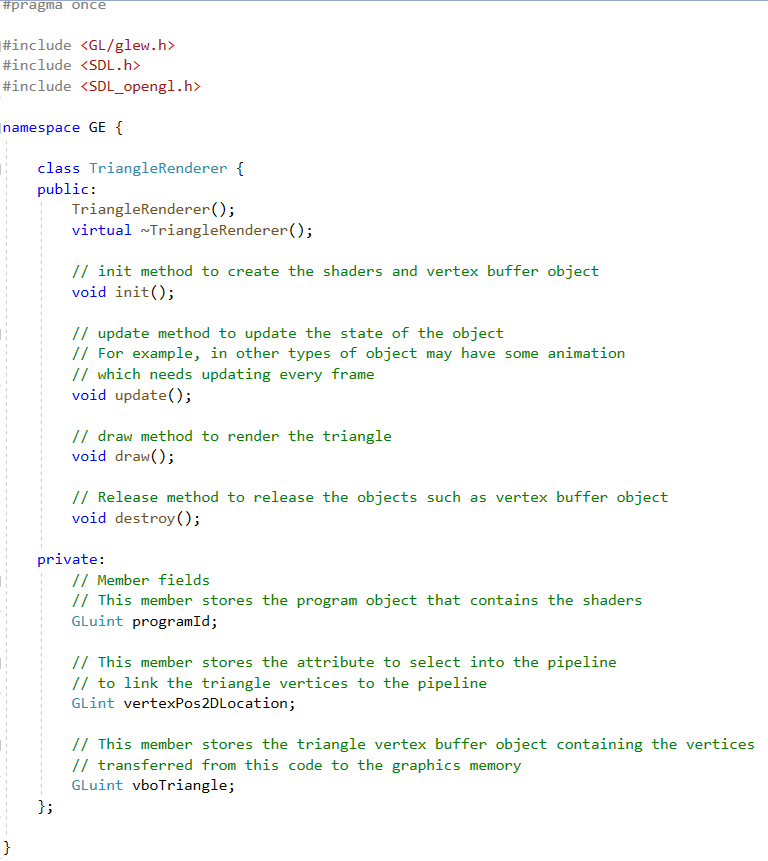


1. You should see the files added to the project. To confirm the code has transferred okay, build and run the program.

## Task – Create the TriangleRenderer class

In order to encapsulate the triangle creation and rendering code, we will create a class.

1. Create new files, TriangleRenderer.cpp and TriangleRenderer.h. You do this by right clicking on the project name in Solution explorer, then select Add… and finally New Item. Make sure you select the correct type of file, C++ File (.cpp) and Header File (.h), respectively. If you are unsure how to do this, then re-read pages 10 to 11 on how to create source code files.
2. Make sure the TriangleRenderer.h file is open and type in the following to declare the class.



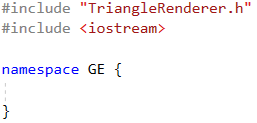
Note that you should see the #pragma once is already in the new file.

The TriangleRenderer class shares many similarities with the GameEngine class in the last exercise. There is a constructor and destruction but these aren’t implemented. The init method will setup the shaders, program and vertex buffer object to store the triangle vertices. The update method is used to update the state of the triangle except this method won’t have any code and doesn’t do anything as a result. We will create other classes that also have an update method that updates the state of the object, e.g. implements a basic motion. The draw method will use the program and vertex buffer object to render the triangle. Finally, the destroy method destroys the OpenGL objects, program and vertex buffer object, releasing the memory for other programs.

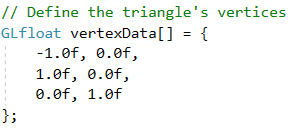
1. Open the TriangleRenderer.cpp file. Add the following code to add the header files necessary for the implementation of the TriangleRenderer class.



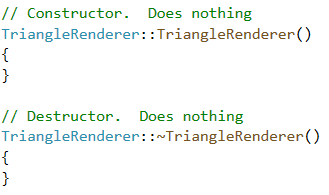
1. The TriangleRenderer will be part of the GE namespace. Type in the following where after you have typed the { the } is added. Make sure you add new lines as you will be adding code between them. Your code should look something like the following after typing in the namespace.



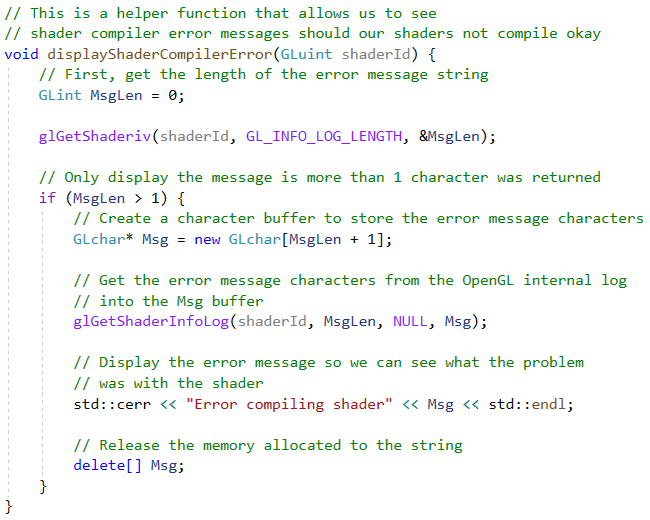
1. With the cursor inside the { and }, type in the following to define the triangle vertices. It is an array of six floating point numbers that will be handled in pairs by the vertex shader.



1. Type in the following to define the constructor and destructor.

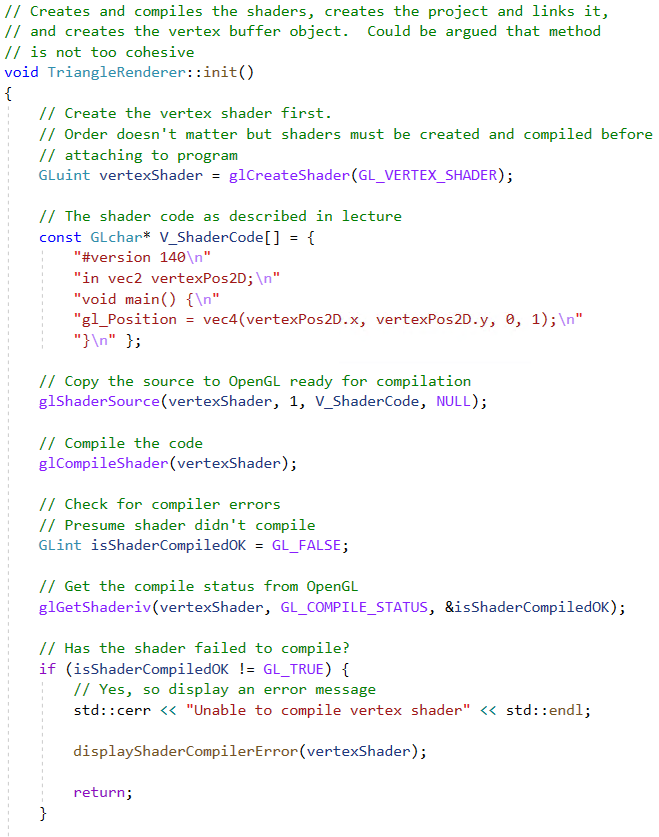


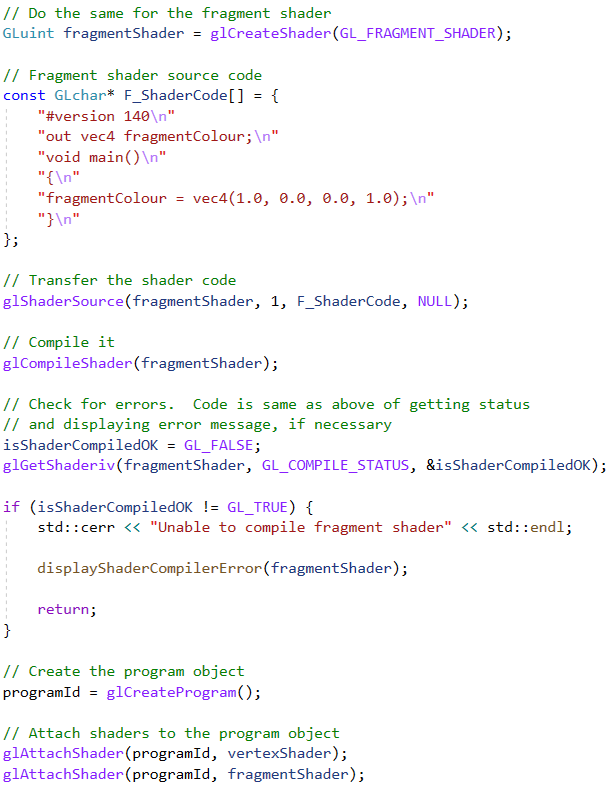
1. Type in the following to define a function to display shader compilation error message text.

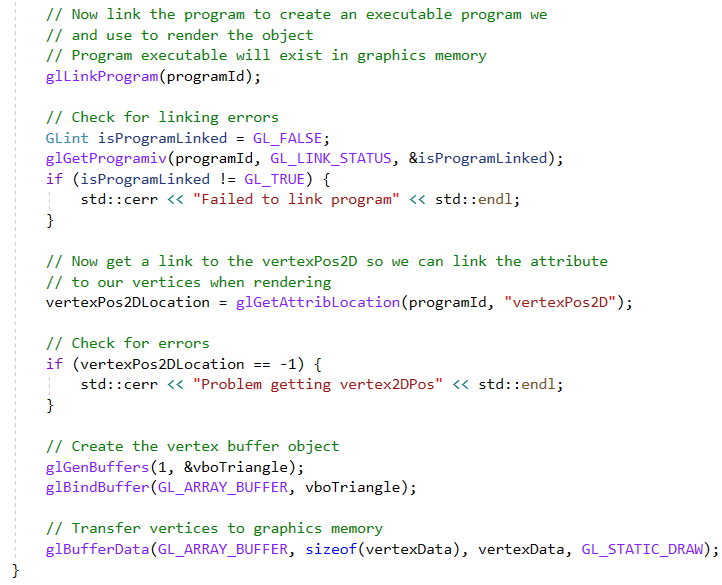


Read the comments in the code to understand how it works. You should consider how to make this code available to other classes that use shaders so compilation errors can be displayed on screen to help debug them.

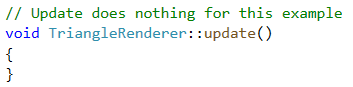
1. Type in the following to define the init method. This method uses the OpenGL code described in the lecture to create and compile shaders, create the program object, attach shaders to them and finally link the program. The method also creates the vertex buffer object and transfers the vertices shown is step 12 to graphics memory.



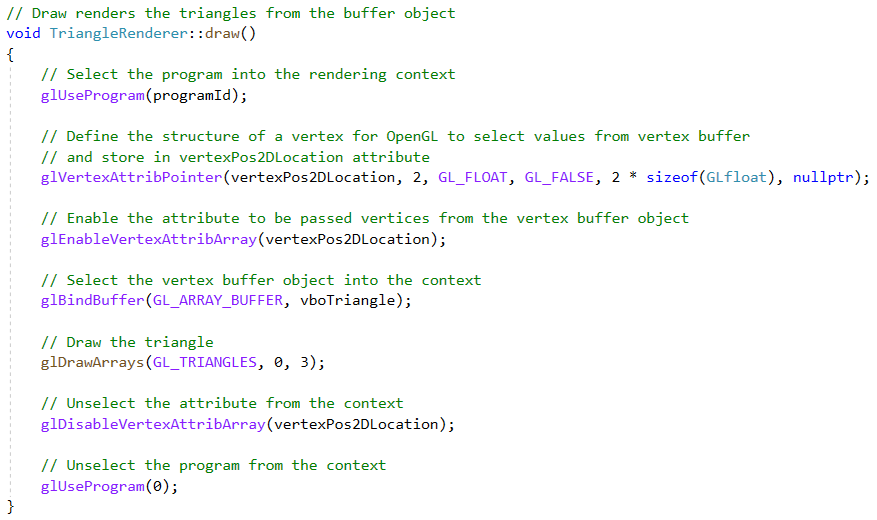




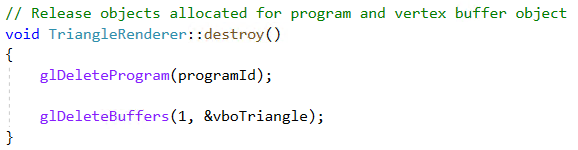
1. Type in the following to define the update method. As described previously, for this class update does nothing.



1. Type in the following to define the draw method. It selects the program created in the init method into the context. It also sets the vertexPos2D attribute as source for vertices and selects the vertex buffer object containing the triangles into the context. It draws the triangle and, finally, unselects the program and attribute from the context.



1. Type in the following to define the destroy method that releases the OpenGL objects created to store the program and vertex buffer object.



## Task – Using the TriangleRenderer class

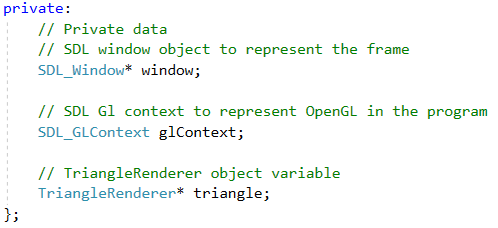
Now the TriangleRenderer class is complete you need to instantiate an instance (object) from the class and use it in the GameEngine class to render a triangle on the screen. You should have noticed the methods (which form the interface of the class) are similar to the methods of the GameEngine class such as there’s an init, update, draw and so on. These to be invoked in a certain order, e.g. we can’t draw until the object has been initialised with init method.

To use the class in the GameEngine we:

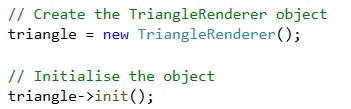
1. Create a data member field to store the TriangleRenderer object.
2. In the GameEngine init method create an instance of the TriangleRenderer object and invoke its init method to create the shaders, program and vertex buffer object.
3. In the GameEngine draw method invoke the TriangleRenderer object’s draw method.
4. In the GameEngine shutdown method invoke the TriangleRenderer’s object’s destroy method.

To achieve the above you need to first open the GameEngine.h file and complete the following steps:

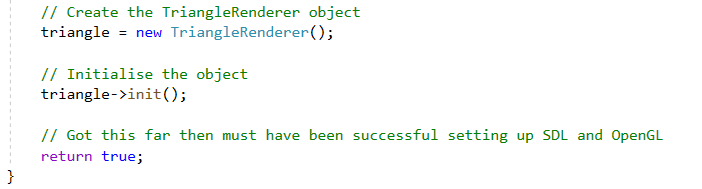
1. Modify the private section to the following in order to add a data member for the TriangleRenderer.



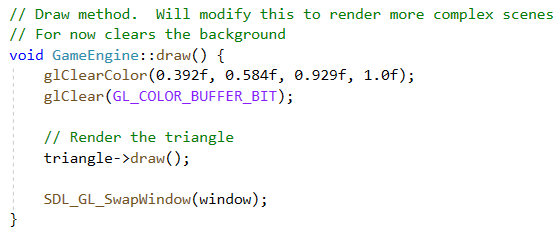
1. Open the GameEngine.cpp file. Find the init method and put the following code just before the return true; statement.



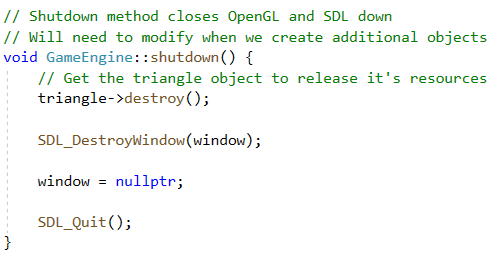
the end of the init method should look like the following when complete.



1. In the draw method, add the highlighted code to invoke the TriangleRenderer object’s draw method to render the triangle in the current context.



1. In the shutdown method add the highlighted code to invoke the TriangleRenderer’s destroy method to release the OpenGL objects used by the TriangleRenderer.

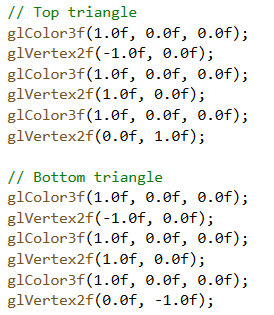


Now build and run the program removing any errors in the source. You should see red triangle on the screen like in the previous exercise.

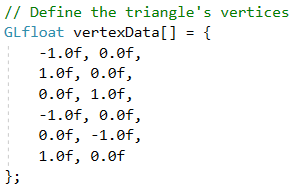
## Open tasks

1. If you created you own shapes using immediate mode, then try to reproduce then using the pipeline. To do this you need the vertex values only for this week, we will look at how we can send colour data to the pipeline in the next exercise.

To change the shape of the object, you need to modify the vertices in the array defined in step 12 on page 6. For example, if you created two triangle (irrespective of vertex colours) to create a diamond shape you might have the following code using immediate mode.



The vertices are (-1.0f, 0.0f), (1.0f, 0.0f), (0.0f, 1.0f), (-1.0f, 0.0f), (1.0f, 0.0f), (0.0f, -1.0f). To use them in the TriangleRenderer we change the array in step 12, page 6 to:



You will also have to change the code in the TriangleRenderer’s draw method to reflect the correct number of vertices, e.g. change:

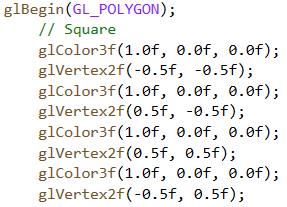


to

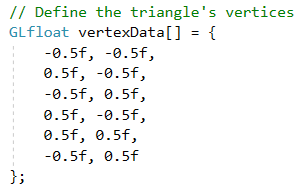


And run the program. Try with your own shapes.

Issue: you may have used one of the other geometry types such as GL\_POLYGON to draw an object. For this program, you need to convert the shape into a number of triangles where there will be more vertices. For example, you may have drawn a square using code like this.

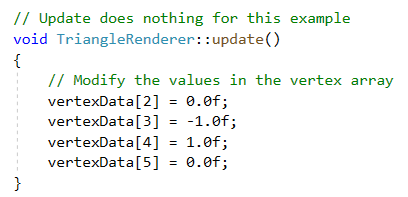


This will not directly work with pipeline mode because it is expect triangles, not a quad. You need to break down the quad into two triangles in this example. One triangle is (-0.5f, -0.5f), (0.5f, -0.5f), (-0.5f, 0.5f) and the other is (0.5f, -0.5f), (0.5f, 0.5f), (-0.5f, 0.5f). The vertex array would look like the following:

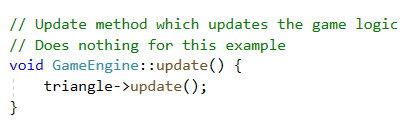


You also need to change the glDrawArrays to draw six vertices as shown above.

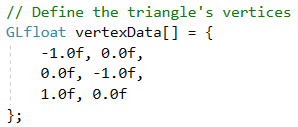
1. Whilst it was said the update method wouldn’t be used for this exercise, we’ll make a change to show an interesting phenomena. Put the vertexData array back to the original code and change the update method to the following.



Modify the GameEngine update method to the following also:



What these statements do is change a vertex so the triangle is upside down and then rendered upside down. Build and run the program, what happens? Is it what you expect give what was just said. What should happen is the triangle should be display upside down because the vertices have been change, was it? If it wasn’t, then why not? If you are unsure the values , then change the vertexData array to the following:



1. Using the TriangleRenderer as a template, create other types of renderer objects, e.g. QuadRenderer.