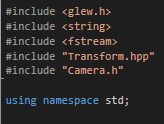
# Shaders

We know what shaders are, how to use them but our code for them is …. Messy….

So, lets clean it up by creating a class to hold all our shader information.

First, create a new class (.h and .cpp) called Shader.

In this class, add the following includes and using statements

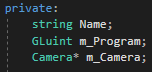


Create a basic class called Shader, with a public constructor and destructor.

Now create a constructor that takes 2 arguments, a const string: FileLocation and a camera reference: camera.



Create 3 new private variables to hold the name of our shader, a reference to the GL shader program and a pointer to a camera.



## Loading Shaders

Ok, lets load a shader!

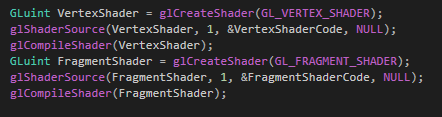
Loading shaders is easy. So far we can be using hard coded string in our main function, but that’s not very flexible or convenient. So, we are going to shift over to loading our shaders from text files.

In the Cpp, create the following function definition:



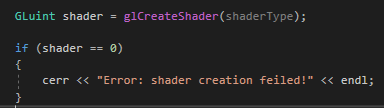
We don’t need this to be a class method as it is not class specific, so iv made mine a static function, so all the code can see it!... possible not the best practice, but it will do for now.

Let’s stop for a moment and look at our old shader creation code.

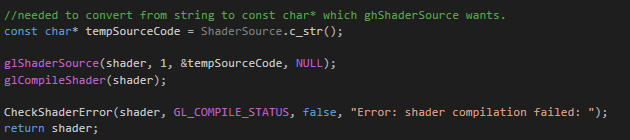


Notice that these to shadres, the vert and the frag shader, are loaded in the same way, but with different parameter, our CreateShadrer() function is going to remove this duplication of code and allow us to specify the shader type and the code we want to load. The return of our function will be the Guint for our shader reference.

So, the first thing to do is call create shader and pass it our shader type. At this point its also a good idea to check the shader has been created properly and generate an error if not.

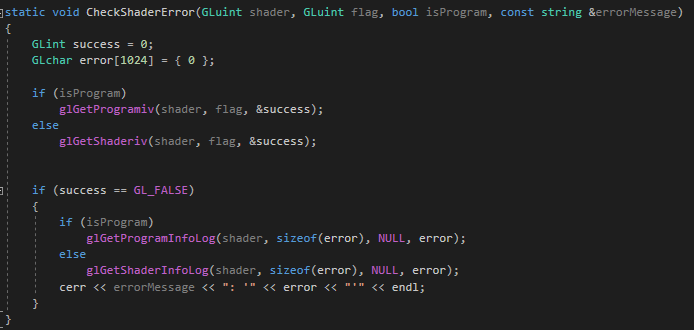


Now were going to link our shader source compile it and check for errors. Finally we will return the shader GLuint to the calling function as this will be our reference to the shader we have just loaded.



Our createShader fuction is now complete.

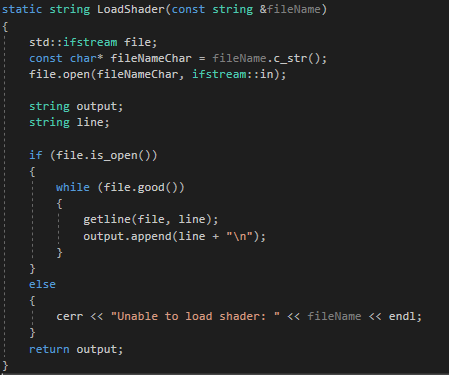
If you are getting any errors from CheckShaderError(), like “function not defined”, just cut the function from our main.cpp in to the shader cpp and make it static, like so:



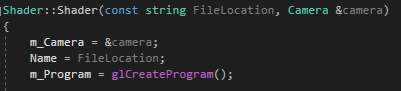
Since this is a shader specific function, it makes sense that it should live in the shader code file.

Ok, so we can create, link and compile and shader type we want, from a string of source code. Now we need to load our shader code from a text file.

So, I’m going to make a new static function called, LoadShader that will take singe: filename as a parameter and return a string, which will be the condense of the file, IE our shader code.



Ok, time to use all this. Go back to the constructor, in the cpp, and set up you m\_camera, Name and m\_Program veriables like so.

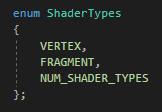


We will need a reference to our camera later, so we can pass its view information to our shade, the name will just be the fileLocation, for debug purposes and m\_program will be a reference to our shader program.

Now we need a place to store the shaders. Head back to the h file and add a private GLuint array, m\_Shaders[];



Num shaders is an enum that specifies which shader type we want and the number of shader types we can have. The enum looks like this:

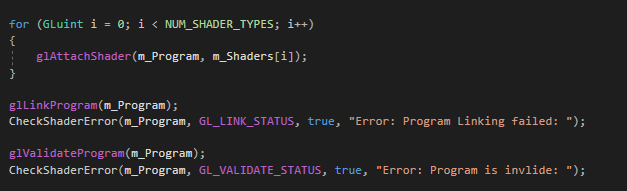


Over in the cpp, we can use this new array and the ShaderTypes enum to load our shaders in to a known place in our program.



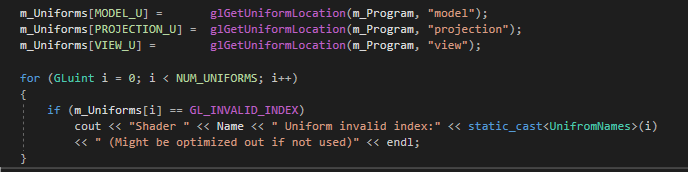
Now if we wanted to, we could add a geometry shader with ease, or any other type of shader.

Ow we need to attach our shaders, link the program, validate and check for errors.



Nothing new here, I’m just looping over our m\_Shadres array to reduce how hard coding each shader to be attached.

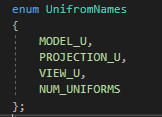
Now we come to the uniforms.



Uniforms are variables n our shaders that are updated every frame from the CPU to the GPU but their values persist for all instances of the vertex shader (we would have to pass them on to the fragment shader in order to use them there, more on this in the lighting tutorial).

So, we fine the uniforms by name, from the program and store a reference to it in our m\_Uniforms array. We’re using another enum to hold our enum references in known locations in memory.

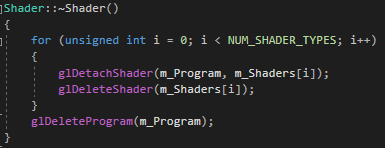
The m\_Uniforms array and the uniforms enum look like this:





And they are defined in the h file.

In the shader destructor, we need to delete the shaders and program.



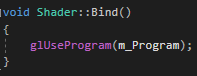
Now we only need two more functions, Update and Bind.

So define them in the h file.

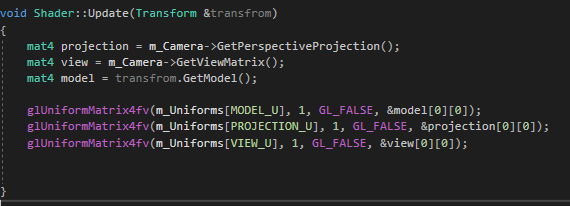


Update takes a transform, as we will need to pass the model to the shader so we can use it int the shader.

The implementation of bind is simple



And the update is almost as easy.



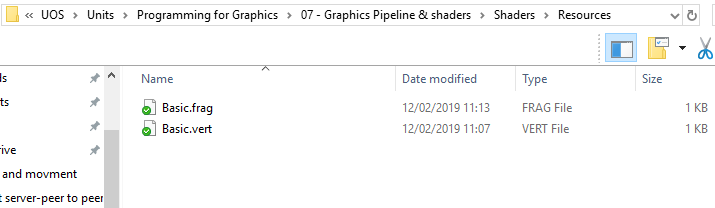
This is the same code we had in our main, while loop.

Now we need to remove the old shader code from the main function.

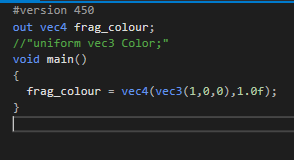
So, go through the main code and comment it out, or delete it, which ever.

Now we need to make a folder and some text files which will hold our vertex code.

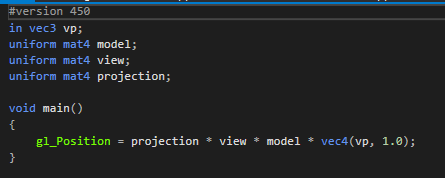
In the solution folder, make a new folder called “resources and inside that make a 2 text files, called Basic.frag and Basic.vert.



Add these files to the project and add the following code to the basic.frag file:



And add the following code to the vert file:



Only thing left is to load the shaders and call them.

Include the shader.h file in the main.cpp and create a new shader object.



The first parameter is the location of our basic shader files, the second is our camera.

And above the square.draw function add the following code:



Don’t forget to delete the object and clean up the pointer at the end.