Lab 3 – Introduction to Shaders

In this lab we are going to start of the process of moving towards using the Programmable Pipeline.

The tasks we are going to complete in this lab includes

- Loading Vertex and Fragment Shaders from file
- Compiling and Linking Shaders
- Rendering with Shaders
- Sending Uniforms
- Adding more input variables for the vertices

1. Initialisation Changes

We are now going to be using the programmable pipeline, we need to make two changes to how we initialise OpenGL. We are also going to remove all code that deals with the old Fixed Function Pipeline.

1.1 GLEW

We need to turn on some experimental features for GLEW, while not really needed for this exercise it may be needed in the future. Add the following code to the **initOpenGL** function **before** the call to **glewInit**

```
glewExperimental = GL_TRUE;
```

This call will turn on the experimental version of GLEW, this queries all supported extensions by the Graphics Driver when the **glewInit** function is called.

1.2 SDL

Since we are using the Programmable Pipeline we need to tell SDL to create a context which supports OpenGL 3.2. To do this we need to set some attreibutes after we initialise SDL but before we create the OpenGL Context. Add the following code snippet to the top of **initOpenGL** function before we create the **SDL_GL_Context**.

```
//Ask for version 3.2 of OpenGL
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 3);
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, 2);
SDL_GL_SetAttribute(SDL_GL_CONTEXT_PROFILE_MASK, SDL_GL_CONTEXT_PROFILE_CORE);
```

This will ask for version 3.2 of OpenGL with a Core Profile, read more about profiles at the following URL
https://www.opengl.org/wiki/Core_And_Compatibility_in_Contexts

1.3 Remove old fixed instructions

Now that we have created 3.2 Core Profile version of an OpenGL Context we should remove this old fixed function functionality.

Navigate to the **setViewport** function and remove the following lines in bold

```
glMatrixMode( GL_PROJECTION );
glLoadIdentity( );

//Calculate perspective matrix, using glu library functions
gluPerspective( 45.0f, ratio, 0.1f, 100.0f );

//Swith to ModelView
glMatrixMode( GL_MODELVIEW );

//Reset using the Indentity Matrix
glLoadIdentity( );
```

The reason we are removing these lines is that they all deal with transformations which functionality has been taken over by the Vertex Shader. The only one that remains is **glViewport**, this is used in the non-programmable stages.

We can also remove pretty much everything from the render function apart from the Clear Calls, the binding of the buffer, the call to draw and the swap.

Your **render** function should look like the following

```
glClearColor( 0.0f, 0.0f, 0.0f, 0.0f );
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

glBindBuffer(GL_ARRAY_BUFFER, triangleVBO);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, triangleEBO);




glDrawElements(GL_TRIANGLES,36,GL_UNSIGNED_INT,0);
SDL_GL_SwapWindow(window);
```

The above assumes that you have completed the Element Array Buffer Exercise with the cube drawn to the screen.

2. Loading, Compile & Linking Shaders

In order to use the Programmable Pipeline we need to first load, compile and link the shaders we are going to bind to the pipeline.

2.1 Compile Shader from string

We are going to create our own compile function, which will be used to simplify our Game Code.

Add a new header file called **Shader.h** to the project(right click, add new file). In the empty file, add the following header guard.

```
#ifndef Shader_h
#define Shader_h


#endif
```

We are going to add the include statements to support the loading and compilation of the shaders. Add the following just inside the header guard

```
#include <GL/glew.h>
#include <SDL_opengl.h>
#include <iostream>
#include <fstream>
#include <string>
```

We are now going to add an enum which will be used to define the shader type we are going to compile, this is just something we are making up so that we can use the same function to compile different types of shaders. Add the following code to **Shader.h** just after the include statements.

```
enum SHADER_TYPE
{
        VERTEX_SHADER=GL_VERTEX_SHADER,
        FRAGMENT_SHADER=GL_FRAGMENT_SHADER
};
```

This just simply maps our own enum onto OpenGL equivalents, the only reason to do this if we want to eventually move to cross platform development.

Now add the following function declaration to load a shader from a char array, this should just going below the above **enum** statement in **Shader.h**

```
GLuint loadShaderFromMemory(const char * pMem, SHADER_TYPE shaderType);
```

This function will take a character array as the first parameter, this will hold our shader source code(loaded from file or just defined as global character array). The second parameter is the type of shader we are attempting to load(see above enum). This function will return **GLuint** which is the ID of the loaded shader, if the function was unsuccessful then the value return will be 0.

Now we are going to create a Source file(cpp) file to hold our definition of the above function. Add a new source file called **Shader.cpp** to the project(right click, add new file).

Open this new file and add the following code to include the Shader header file and define the function.

```
#include "Shader.h"

//Load it from a memory buffer
GLuint loadShaderFromMemory(const char * pMem, SHADER_TYPE shaderType)
{
	GLuint program = glCreateShader(shaderType);
	glShaderSource(program, 1, &pMem, NULL);
	glCompileShader(program);
	return program;
}
```

The first line creates the program with specified shader type, the 2nd line copies the character array containing the shader source into the shader program, we then compile the shader and finally return the compiled program.

2.2 Asset Path for Shaders

Create a folder under the solution directory(right click on solution, select **Open Folder in File Explorer**), name this folder **assets**. Inside this assets folder create a new folder called **shaders.**

Switch back to Visual Studio and open up **main.cpp**, add the following after the include files

```
#ifdef _DEBUG && WIN32
const std::string ASSET_PATH = "../assets";
#else
const std::string ASSET_PATH = "assets";
#endif

const std::string SHADER_PATH = "/shaders";
```

This will be used to build a directory we can load shaders dependent on what version of the application we are building. For example if we are in debug mode(development) then we build from a directory which is just up a level from the debug directory. If we are any other state, we will load relative from the executable

2.3 Load Shader file into string

We already have a load shader function which takes in a char array, we now need a function which takes in a file, reads the contents of the file into a char array and then calls our **loadShaderFromMemory** function

Open up **Shader.h** file and add the following function declertation to the file, you can add this file to any part of the file.

```
GLuint loadShaderFromFile(const std::string& filename, SHADER_TYPE shaderType);
```

This is very much like our loadShaderFromMemory function but instead of receiving a char array it takes in a string representing the path and filename.

We are now ready to define the function, switch to **Shader.cpp** and the following code to load a file.

```
//Load Shader from File
GLuint loadShaderFromFile(const std::string& filename, SHADER_TYPE shaderType)
{
	std::string fileContents;
	std::ifstream file;
	file.open(filename.c_str(), std::ios::in);
	if (!file)
	{
		return 0;
	}
	//calculate file size
	if (file.good())
	{
		file.seekg(0, std::ios::end);
		unsigned long len = file.tellg();
		file.seekg(std::ios::beg);

		if (len == 0)
		{
			std::cout << "File has no contents " << std::endl;
			return 0;
		}

		fileContents.resize(len);
		file.read(&fileContents[0], len);
		file.close();
		GLuint program = loadShaderFromMemory(fileContents.c_str(), shaderType);
		return program;
	}

	return 0;
}
```

This function is fairly long, but should be easy to follow. We use **ifstream** in order to open the file, if it is successful(**file.good()**) then we seek to the end(**file.seekg**) and this enables to get the length of the file(**file.tellg()**). We then reset the stream to the beginning of the file. We then resize the string which will contain the contents of the file so it is the size of the file(**fileContents.resize**), then read(**file.read**) and close the file(**file.close**). We now have the contents of the file in a string, we can use this to call our **loadShaderFromMemory** function.

NB. If for some reason you don't understand any of this, please checkout the C++ docs online or ask your lab demonstrator

2.4 Checking for Compile Errors

While the above functions will load our shader there is no error checking to see if our shader has been compiled properly. We are going to create a function which will check for any errors that has been generated by a compiled shader.

Add the following code snippet to **Shader.h**, add this function decleration to the bottom of the file.

```
bool checkForCompilerErrors(GLuint shaderProgram);
```

Now switch to **Shader.cpp** and add the definition for the above function

```
bool checkForCompilerErrors(GLuint shaderProgram)
{
	GLint isCompiled = 0;
	glGetShaderiv(shaderProgram, GL_COMPILE_STATUS, &isCompiled);
	if (isCompiled == GL_FALSE)
	{
		GLint maxLength = 0;
		glGetShaderiv(shaderProgram, GL_INFO_LOG_LENGTH, &maxLength);

		//The maxLength includes the NULL character
		std::string infoLog;
		infoLog.resize(maxLength);
		glGetShaderInfoLog(shaderProgram, maxLength, &maxLength, &infoLog[0]);

		std::cout << "Shader not compiled " << infoLog << std::endl;

		//We don't need the shader anymore.
		glDeleteShader(shaderProgram);
		return true;

	}
	return false;
}
```

This function might seem complex but lets break it down a little bit, first point to realise that the **glGetShaderiv** function is used to get parameters from a shader object, the middle parameter defines what you attempting to retrieve from the shader object and what you should pass into the last parameter

- **GL_COMPILE_STATUS** – returns GL_TRUE if compile was successful, or GL_FALSE if not
- **GL_INFO_LOG_LENGTH** – returns the number of characters in the log that is generated

This means that the function checks to see if we compiled the file, if not then we get the length of the log. This is then used to resize a string to hold the error message. We call **glGetShaderInfoLog** to fill out this string with the error message, which we then display to the outputstream(**cout**). The last thing we do is delete the shader, this ensures that we don't leak any memory allocated to the shader object.

We now need to call this error checking function, open up Shader.cpp and in the **loadShaderFromMemory** and add the following snippet just before the **return statement** at the end of the function

```
if (checkForCompilerErrors(program))
{
        return 0;
}
```

3. Writing & Compiling Shaders

We now have the utility functions ready which will be used to load, compile and check for any errors. This means that we can start writing shaders.

At the very least we are going to create two shaders

- Vertex Shader – simpleVS.glsl
- Fragment Shader – simpleFS.glsl

Both of these files will be created in the shader folder we created in **step 2.2**

3.1 Vertex Shader

First off we are going to create the Vertex Shader , **right click** on the project**,** select **Add->New Item**. In the dialog that pops up, select the **Utility** option from the left hand side and then select **text file**, name the file **simpleVS.glsl** and make sure the location points to the **Shader** folder

Open this file in visual studio and add the following code

```
#version 150

in vec3 vertexPosition;

uniform mat4 MVP;

void main()
{
	gl_Position = MVP * vec4(vertexPosition, 1.0);
}
```

This contains a very basic shader which simply transforms an input vertex by a matrix.

3.2 Fragment Shader

Follow similar steps as part 3.1 to create the Fragment shader, the major difference is that the name should be **simpleFS.glsl**. Add the following code to the file in Visual Studio.

```
#version 150

out vec4 FragColor;

void main()
{
	FragColor = vec4(1.0f,1.0f,1.0f,1.0f);
}
```

This simply outputs a white colour for every pixel fragment

3.3 Compile Shaders

Now that we have the shaders ready, we need to load and compile the shaders. Add the following function to **main.cpp**, make sure it is above **main** function in the file.

```
void createShader()
{

}
```

Add the following code to the above function to load and compile each of the shader files above.

```
GLuint vertexShaderProgram=0;
std::string vsPath = ASSET_PATH + SHADER_PATH+"/simpleVS.glsl";
vertexShaderProgram = loadShaderFromFile(vsPath, VERTEX_SHADER);

GLuint fragmentShaderProgram=0;
std::string fsPath = ASSET_PATH + SHADER_PATH + "/simpleFS.glsl";
fragmentShaderProgram = loadShaderFromFile(fsPath, FRAGMENT_SHADER);
```

This simple builds a path to the files and then calls our own **loadShaders** function.

4. Linking Shaders

We now have compiled shaders, we require that these shaders are linked together into a Shader Program. The first we are going to do is add a integer which will be used to track the shader program, we are then going to link the shaders together and finally delete the individual vertex and fragment shader programs.

4.1 Shader Program Global Value

Add the following code snippet in **main.cpp** to declare a global integer to hold our shader program.

```
GLuint shaderProgram=0;
```

4.2 Creating and Linking the Shader Program

First off we need to create another utility function which will be used to check for linking errors. Navigate to **Shader.h** and add the following deceleration to the file.

```
bool checkForLinkErrors(GLuint program);
```

Then move to **Shader.cpp** and add the following function definition

```
bool checkForLinkErrors(GLuint program)
{
	GLint isLinked = 0;
	glGetProgramiv(program, GL_LINK_STATUS, &isLinked);
	if (isLinked == GL_FALSE) {

		GLint maxLength = 0;
		glGetProgramiv(program, GL_INFO_LOG_LENGTH, &maxLength);

		//The maxLength includes the NULL character
		std::string infoLog;
		glGetShaderInfoLog(program, maxLength, &maxLength, &infoLog[0]);

		std::cout << "Shader not linked " << infoLog << std::endl;

		//We don't need the shader anymore.
		glDeleteProgram(program);
		return true;
	}

	return false;
}
```

The above function is pretty much equivalent to the compile function, the only real difference is the call to **glGetProgramiv** rather than **glGetShaderiv**.

We are now at the stage where we can link and attach our shaders Navigate to the **createShader** function in main.cpp and add the following code snippet after we have loaded both shaders.

```
shaderProgram = glCreateProgram();
glAttachShader(shaderProgram, vertexShaderProgram);
glAttachShader(shaderProgram, fragmentShaderProgram);
glLinkProgram(shaderProgram);
checkForLinkErrors(shaderProgram);

//now we can delete the VS & FS Programs
glDeleteShader(vertexShaderProgram);
glDeleteShader(fragmentShaderProgram);
```

This snippet starts off by creating the program, then it attaches the Vertex & Fragment shader, links the shader & checks for errors and finally it deletes the Vertex and Fragment shader programs.

4.3 Calling our createShader function

We should now call **createShader** function**,** add the following code to the **main** function in **main.cpp**. This should be the last thing done before you enter the Game While loop.

```
createShader();
```

4.4 Cleanup

The act of creating a shader program allocates some memory, ensure you add the following code snippet to the **Cleanup** function in **main.pp**. This should be the first line in the function.

```
glDeleteProgram(shaderProgram);
```

5. Rendering with Shaders

We are almost ready to draw some geometry using our shaders but we still require a few bits of 'glue' code to map our Vertices on the Application Stage to the Vertices in the shader.

5.1 Input Vertices

We as graphic programmers need to provide information about the layout of the vertices. The first stage of this will occur when the Shader is linked. Add the following code snippet to the **createShader** function of the **main.cpp** file. This should line should be added after we link the program.

```
glBindAttribLocation(shaderProgram, 0, "vertexPosition");
```

This line basically binds the location 0 to vertexPosition(shader input variable) in the Shader Program(the 1st parameter)

5.2 Rendering

Since we are using Shaders our render function will change a fair bit. Navigate to the **render** function and add the following line to bind the shader program to the pipeline. This line should be added just after we bind the vertex buffer.

```
glUseProgram(shaderProgram);
```

The **glUseProgram** will bind the shader program to the pipeline and will be the active program until we call the function again.

Immediately after the useProgram line, add the following snippet in order to finalise the bindings of the C++ Vertex Structure to the Vertex in the Shader.

```
  //Tell the shader that 0 is the position element
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
```

These lines are very similar to what we have seen before(**glVertexPointer**), the first parameter is the index value, which is linked to the **glBindAttribLocation**.

6. Sending Uniforms

In step 5 we have seen how we send vertices across to the shader, in this section we are going to see how we send Uniforms(constants). Usually when it comes to uniforms we have to setup these values in the application stage, then we find the location in the program and finally send the Uniform across.

6.1 GLM Library

Many of uniforms we send will be matrices or vectors, we can define these as structures in our C++ code or we can use a library such as GLM. This library comes with support for most major geometrical constructs(vectors & matrices) and many of operations we required for 3D Graphics.

The first thing you will need to change the project settings so that the **Additional Include Directory** has an entry for **C:\Specliasied Applications\glm**. There is no need to setup libraries as GLM is a header only library.

Add the following header files to **main.cpp**

```
//maths headers
#include <glm/glm.hpp>
using glm::mat4;
using glm::vec3;

#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
```

The above statements include the header file for the library, transformations and a header which is used to return pointers of maths of types.

6.2 Matrices

We are going to add some global values which will be used to hold our Application stage matrices. Add the following to the top of the **main.cpp** file, next to the other global values.

```
//matrices
mat4 viewMatrix;
mat4 projMatrix;
mat4 worldMatrix;
```

Next more down to the **update** function and add the following to calculate all the matrices.

```
projMatrix = glm::perspective(45.0f, (float)WINDOW_WIDTH /
(float)WINDOW_HEIGHT, 0.1f, 100.0f);

viewMatrix = glm::lookAt(vec3(0.0f, 0.0f, 10.0f), vec3(0.0f, 0.0f, 0.0f), vec3(0.0f, 1.0f,
0.0f));

worldMatrix= glm::translate(mat4(1.0f), vec3(0.0f,0.0f,0.0f));
```

The reason we do this in the update function is because we may update these during the course of the game. These lines are the GLM equivalent of the transformations we used in Lab 2.

6.3 Sending Values to Uniform Locations

We have some values you want to send, you have first grab the location using **glGetUniformLocation,** once you have the location you can send the value across using on the **glUniform*** functions. Add the following code snippet after we call **glUseProgram** in the render function.

```
GLint MVPLocation = glGetUniformLocation(shaderProgram, "MVP");
mat4 MVP = projMatrix*viewMatrix*worldMatrix;
glUniformMatrix4fv(MVPLocation, 1, GL_FALSE, glm::value_ptr(MVP));
```

The above code, grabs the location **MVP** for the shader program, then calculates a combined Model View Projection matrix and finally sends it across to the shader.

7. Additional Exercises

1. Move the cube to different positions
2. Move the camera using keyboard controls
3. Add a uniform value to the fragment shader to control the colour of the fragment. You should then send a value to the shader from the application stage.