Basic Project

COMP3015 Lab 1

Overview

In this lab session, we'll create a basic project in OpenGL with all the dependencies linked. At the end of this lab you'll be able to create an OpenGL project from scratch with all the dependencies in place. Part1 – Basic OpenGL project.

Part 1 - Installation

Requirements

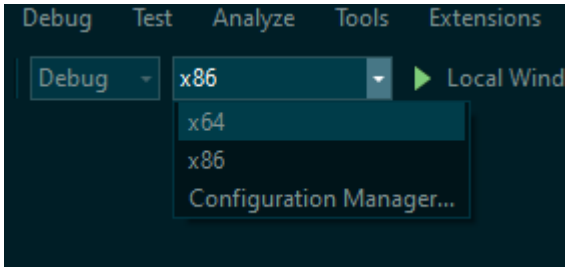
- GLFW (multi-platform library for OpenGL)
- CMake (tool for building, testing and packaging software)
- GLAD (web service)
- GLM (C++ mathematics library for graphics)
- Visual studio 2022 (if you don't have it yet)

Building GLFW with CMake

1. First, we add a folder called "OpenGL" in C drive's public user folder (C:\users\public\OpenGL\). Inside this folder we create 2 folders "lib" and "include".
2. GLFW – we need to create a custom library for our system.
3. Download GLFW (source package) and unpack the folder.
4. Download CMake binary and install it on your machine. Run CMake once installed.
5. We use CMake to generate a project that will build the library for us, by running CMake on the root folder of the downloaded GLFW.
6. Choose the "Where to build the binaries" – Create a new folder called "build" inside the root folder GLFW.
7. Hit configure in CMake and let it compile. Keep all the settings as they are and once finished compiling hit Generate. This should generate a visual studio solution (GLFW.sln) in the new build folder.
8. Open Visual studio project by clicking on the solution of the newly generated project.
9. We run the generated project and create the library; you can find the glfw3.lib in build/src/Debug
10. We add the GLFW library that we created to the "lib" folder in Dependencies.
11. We add the content of the "include" folder (a folder called GLFW) from the downloaded GLFW to the C:/users/public/OpenGL/include.

Visual Studio Project

Create a new Empty C++ project in Visual Studio. Make sure you set the project to 64-bit, so change the dropdown next to the Debug from x86 to x64:



Add a new .cpp file to the project called main.cpp & add the following code:

```
int main()
{
	return 0;
}
```

Add before “return = 0;” line the following code

```
std::cout << "Hello World" << std::endl;
```

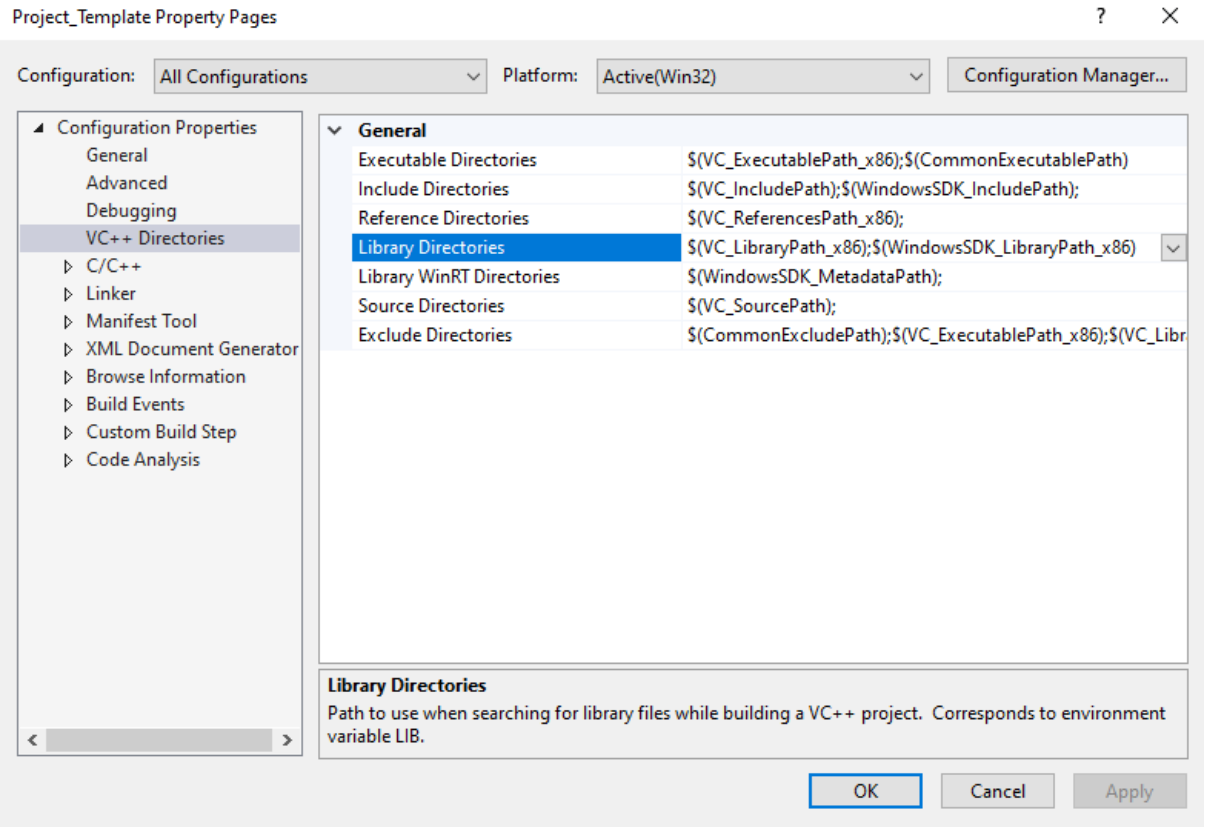
If you run the code now, you should get an error. It’s missing the output stream library. In order to fix it, add the following code at the top of your file:

```
#include <iostream>
```

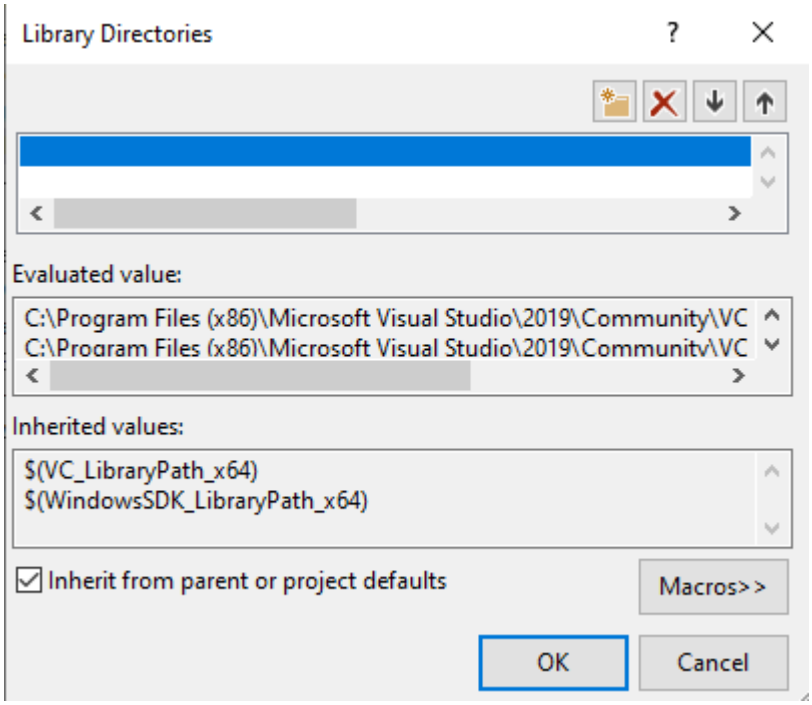
Linking

Next we need to link the GLFW library with your project. In order to link a library to a project, first we need to tell the project where it can find the library and second we link it.

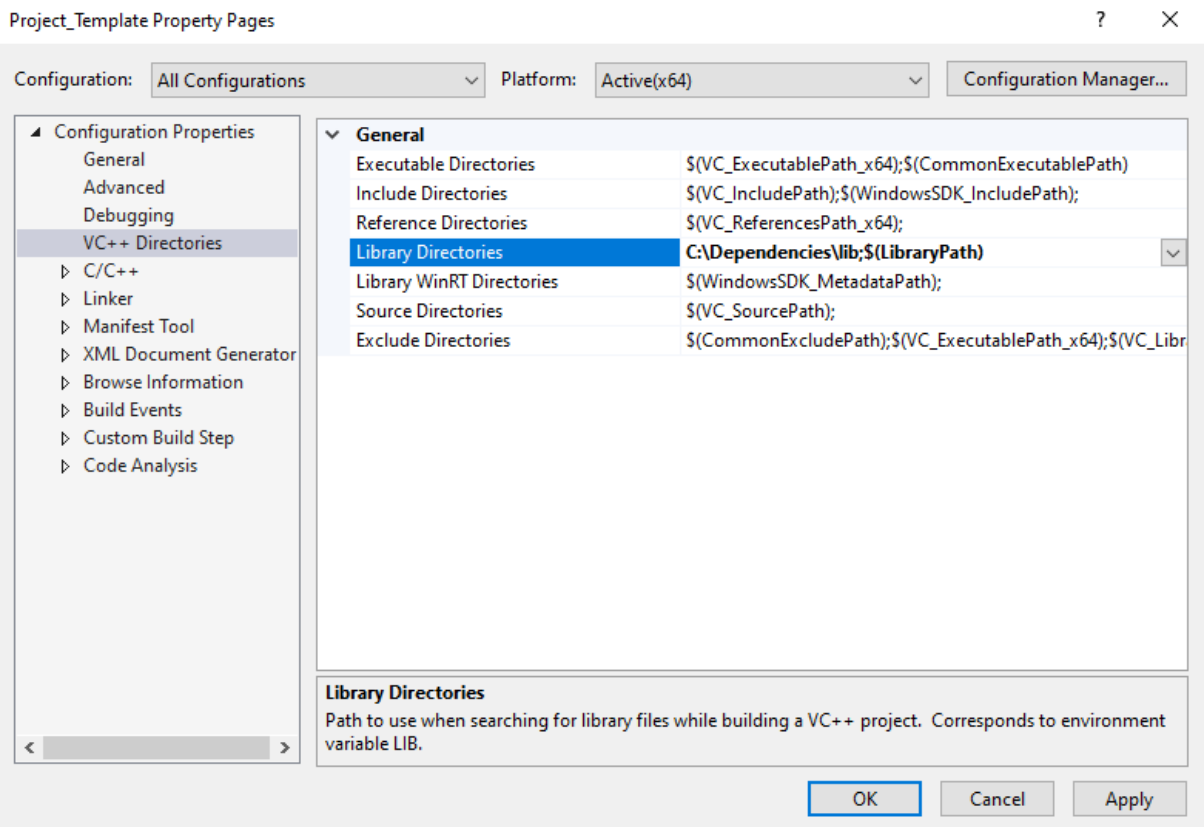
1. First let’s tell the project where the library is.
2. Right click on the project name in the solution explorer and go to VC++ Directories. Make sure you select the ”Library Directories”:



Here, we specify the path for our dependencies (both “lib” and “include”). Click on the arrow on the right and a new window will open:

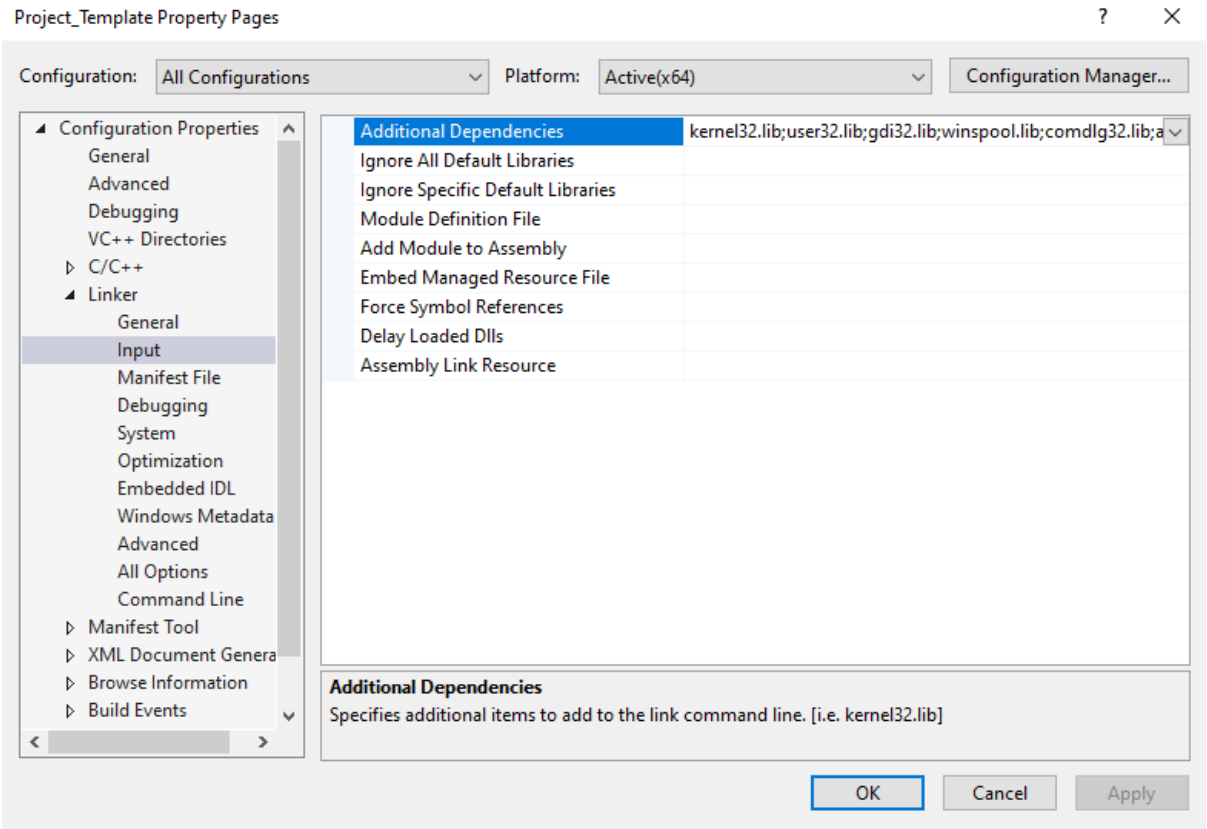


In the highlighted section (the blue bit) add the path to your “Dependency” folder that you can find in C:/users/public/OpenGL/lib. After you added the path, you should have something similar to this screen:

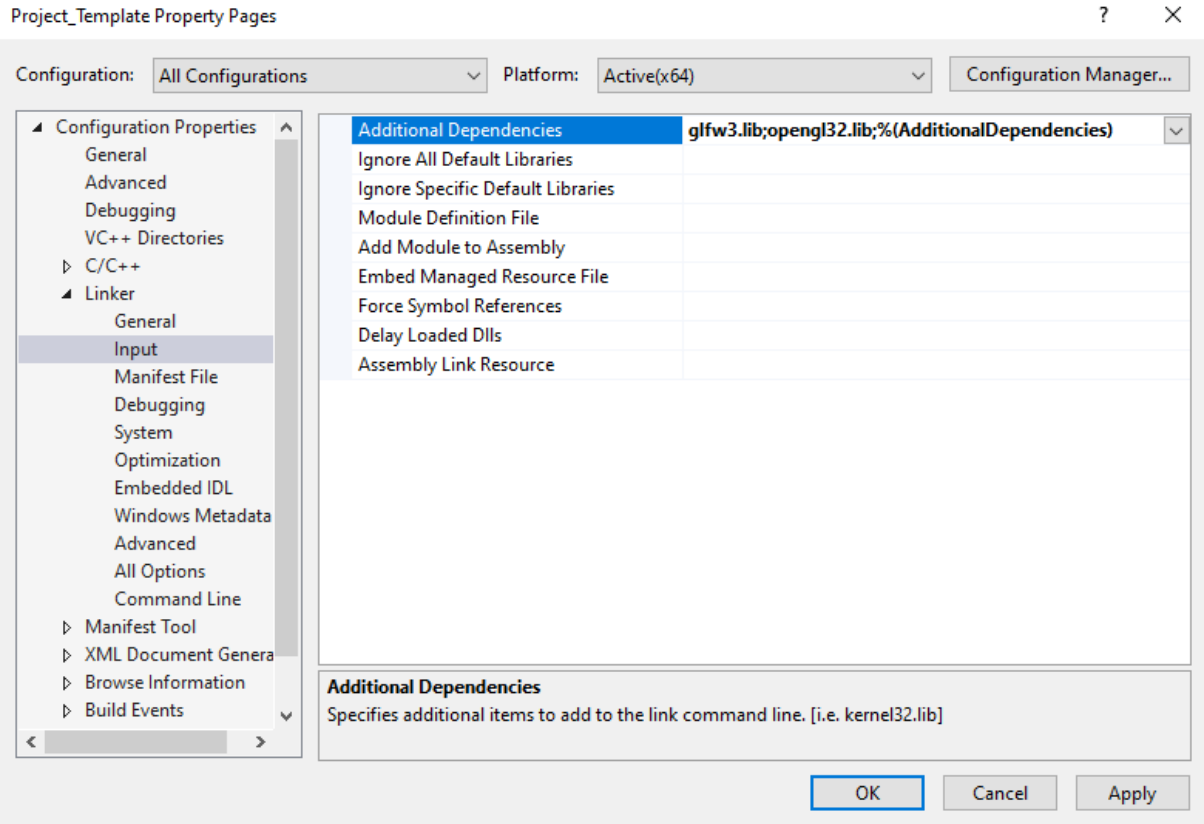


Now highlight the “Include Directories” and do the same steps from points 1 & 2 under Linking, for the “include” folder.

Now that we have told the project where the library is, we need to link the two. If the project properties are not open, please open it. Please select Linker/Input/Additional Dependencies



Just like before, add the GLFW library (make sure you get the name right) “glfw3.lib”. And on a new line add the OpenGl library that already exists on your machine “opengl32.lib”. Now you should have something like this:



Hit Ok and close the project properties window. Last thing is to test if the path is correct for our library and if we can use it. Add the following code at the top of your main.cpp file.

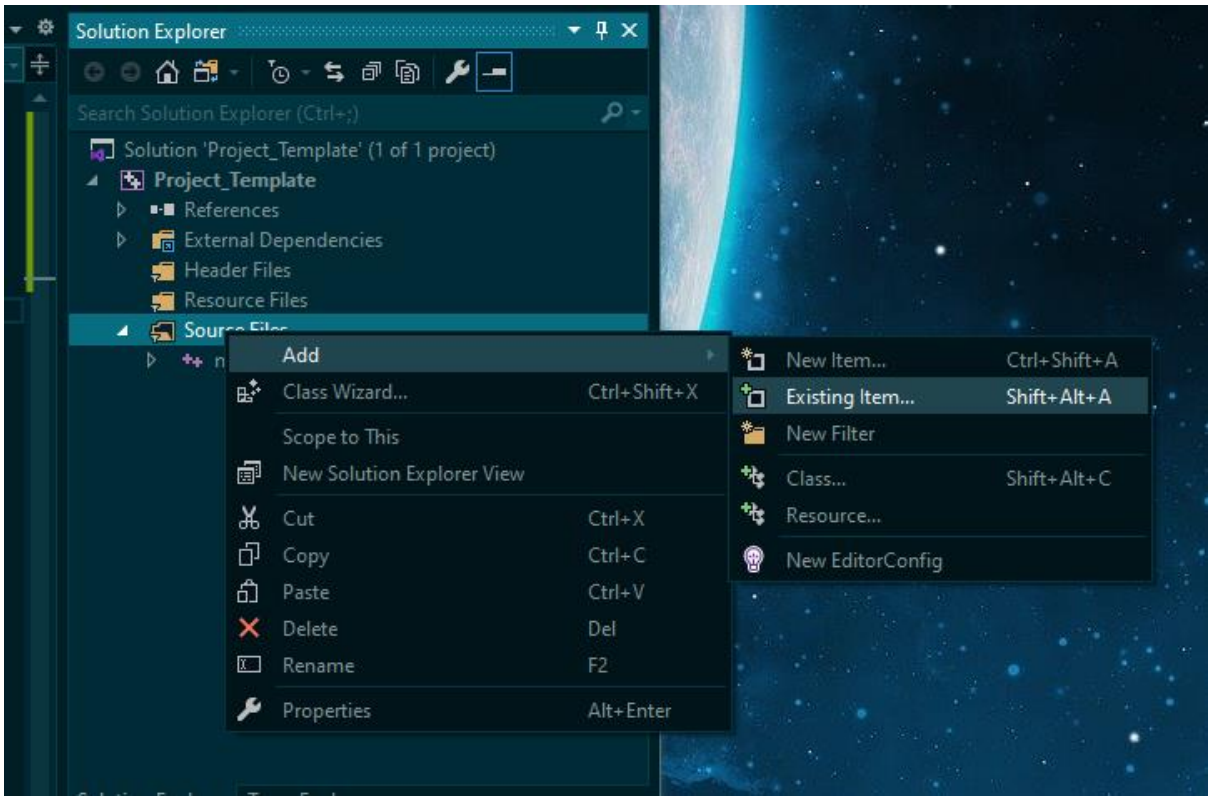
```
#include <GLFW/glfw3.h>
```

Now build the solution. If you’ve done it right, you shouldn’t have any errors. If you still have errors, check all the dependencies. Do the next steps (adding GLAD), GLFW needs Glad to run before it.

GLAD

Next step is to add GLAD to our project. Generate GLAD dependencies for our system

1. Open the web browser tool and generate the dependencies. Make sure you pick C++ and OpenGL 4.6 and make sure the profile is set to “Core”. Tick the option “Generate a loader”
2. Unpack the folder and copy the content of the GLAD “include” folder and paste it into C:/users/public/OpenGL/include. (The content of GLAD “include” should have 2 folders “glad” and “KHR”).
3. Add the “glad.c” file to your project. Copy the file in your project (put it in the same folder where your main.cpp file is). As you can see, even though you copied the file in the project folder it doesn’t show in the project. You need to add it to the project, right click on the “Source Files” and add an existing item. Pick the glad.c file in the project folder and add it to the Project.



Now you should see the file next to your main.cpp file.

Let's test the project so far. Add the following line at the top of the main.cpp file. Make sure you have the correct order. Glad needs to be before GLFW:

```
#include <glad/glad.h>

#include <GLFW/glfw3.h>

#include <iostream>
```

If you've done everything ok. You shouldn't get any errors when building the project.

GLM

Next step is to add the GLM mathematics library. Download GLM from the link provided and unzip the folder. Copy the root directory ("glm") of the header files in the "include" folder (C:/users/public/OpenGL/include).

1. If you want to test the link, add the following line to the top of your main.cpp file and re-build the project. If done correctly, you shouldn't get any errors.

```
#include <glm/glm.hpp>
```

2. Once you test it, you can delete it. We don't need it at the moment. We'll add in any header files required by the code.

Part 2 - Shaders

Overview

In this part we'll open a basic shader project template and we'll use that template to work through various exercises.

Project

1. Download from DLE under the "Lab 1 Template" and get the Basic_Project_Template zipped folder and unpack it.
2. Open the project in Visual Studio and run it once, you should see a coloured triangle. Make sure everything works as expected. If there are any issues, please try and solve them. If you created your Dependency folder in C:, you shouldn't have any issues with the project. (If you have lots of errors in the glad.c file that means my version of glad.c is older than yours. Replace my glad.c file with the glad.c file that has been generated for you. Look at part1 how to add glad.c to your project).
3. In the Project solution explorer we have a folder called "helper". This folder contains all the necessary files to make this project work. The goal of this module is to focus on the shader, so I won't insist on the OpenGL side. Typically any changes you need to make are to the shaders and the files in the "Source Files" folder (except glad.c).
4. A media folder will be used throughout the module to add media files that we're going to use.
5. This is the format we're going to use throughout the module and for submissions.

Uniforms

In this part we'll rotate the triangle on the screen using a uniform variable, namely a rotation matrix. Uniform variables allow for data to be sent to all stages of the graphics pipeline at once. For this reason, they do not need to be passed from one stage to the next within the pipeline. In order to make use of them, the following information regarding them is important:

1. Are intended to be used for data that doesn't change very often.
2. Well suited for matrices used for modelling, viewing and projective transformations.
3. Uniform variables are read only. The value can be changed only from outside the shader.
4. They can be initialised within the shader by assigning them to a constant value along with the declaration
5. They can appear in any shader within a shader program and they are always as input variables.
6. Can be declared in more than one shader but it must be the same in all shaders.
7. The uniform variables are held in a shared uniform namespace for the entire shared program.

Vertex Shader

Let's start with the Vertex Shader:
Declare the uniform variable using the key word "uniform". Do this after the following line "out vec3 Color;":

```
uniform mat4 RotationMatrix;
```

Next, modify the following line "gl_Position = vec4(VertexPosition,1.0);" by multiplying the vec4 with the "RotationMatrix" variable that we just declared. Note, there is no need to modify the fragment shader as we only pass the colour for the output.

OpenGL

Header

We are done with the shaders, we now move to the code in OpenGL as we need to modify it a bit in order to create the matrix and pass it on to the shader. The focus will be in the "Scenebasic_uniform" file. Let's start with scenebasic_uniform.h:

Let's create a private variable called "rotationMatrix" of type glm::mat4. In order for this to work, we need a couple of includes at the top of the file under #include "helper/glslprogram.h":

```
#include <glm/glm.hpp>

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

CPP

We are done with the scenebasic_uniform.h file. Let's move on to the implementation in scenebasic_uniform.cpp:

1. We add four lines of code to the render() function. Make sure you add the code starting right under this line `glClear(GL_COLOR_BUFFER_BIT);`
2. The first line creates a rotation matrix using the glm library. The rotation will rotate with a value given by the variable called "angle".

```
rotationMatrix = glm::rotate(glm::mat4(1.0f), angle, vec3(0.0f,0.0f,1.0f));
glUniformMatrix4fv(location, 1, GL_FALSE, &rotationMatrix[0][0]);
```

3. Next we query for the location of the uniform variable. For that we need to get the programHandle in order to access the location. Make sure you use the correct name for the variable "RotationMatrix". If the uniform variable is not an active uniform, the function returns -1:

```
GLuint programHandle = prog.getHandle();
GLuint location = glGetUniformLocation(programHandle, "RotationMatrix");
```

4. Next we assign a value to the uniform's variable location using this line:

```
glUniformMatrix4fv(location, 1, GL_FALSE, &rotationMatrix[0][0]);
```

5. Please do a bit of investigative research and identify the arguments for `glUniformMatrix4fv`.
6. One last thing to do, make sure we update the value of the angle over the duration of the application. In the update() function add the following code:

```
if( m_animate ) {
        angle += 0.1f;
if( angle >= 360.0f)
angle -= 360.0f;
}
```

The code above is self-explanatory but if something is not clear, do ask. That's it!!!

Extras

For a better understanding, I would suggest reading the first 2 chapters of the OpenGL Superbible book.

Additionally, check out the Shaders section on the Learn OpenGL website (or from the book) - https://learnopengl.com/Getting-started/Shaders

Consider exploring more advanced topics on uniforms in the OpenGL4 Shading Language Cookbook; see the following recipe from “Working with GLSL Programs” - “Using uniform blocks and uniform buffer objects” and “Using Program pipelines”.