Computer Graphics 4052
Lab 1A

Binh-Son Hua
Trinity College Dublin

In the first lab, our objective is to get familiar with a simple OpenGL application that draws a triangle onto the screen. This tutorial walks you through a simple C++ program for OpenGL programming. Let us now get started.

1. Launching the OpenGL window

Download lab1.zip from Blackboard. Unzip and you will find a lab1 folder.
This folder contains the starting code in C++ for your OpenGL tutorial. You will work on lab1_window.cpp. This file contains the boilerplate code of setting up an OpenGL application in C++ for you to get started.

First, let’s compile our program and execute lab1_window to launch our OpenGL window. On Linux and Mac, we will use CMake 3.1 to automatically generate a Makefile, and then call make on the Makefile, which will subsequently call the gcc compiler to compile lab1_*.cpp into executables.

Navigate to lab1 root folder and try the following steps:

- mkdir build
- cd build
- cmake ..
- make

The command “cmake ..” will pass the CMakeLists.txt in the root of lab1 folder to cmake, which generates a Makefile. The “make” command performs the compilation based on this Makefile. The compilation involves compiling the GLFW library and a few other libraries necessary for our program. Note that everything are compiled from source; you do not need to download any dependencies.

On Windows, please see the compilation guide on the last page of this document.
If the compilation is successful, you will see an executable file named ‘lab1_window’ in the build folder. Execute lab1_window, you will see a blank window with a dark blue background, as follows.

Figure 1: Lab 1 window

Let us now examine the source code of lab1_window.cpp to understand what is going on. In lab1_window.cpp, we use GLFW, an open-source multi-platform library for OpenGL to create a simple window and receive mouse and keyboard input events. There is an alternative library named GLUT (and FreeGLUT) that can also be used for creating such a window and receiving events. Both GLFW and GLUT work fine in the context of our tutorial, but GLFW is preferred here as a modern option.

After GLFW initialization, we enter a do..while loop that repeatedly performs some drawing calls to render some content to the screen. This is also known as the game loop or the rendering loop. In lab1_window, we simply perform glClear, which sets the background of the window to a pre-defined color specified by the following line:

```
glClearColor(0.2f, 0.2f, 0.25f, 0.0f);
```

This makes the window display a blank screen in a dark blue color. This function accepts four values to represent a color in OpenGL including red, green, blue, and alpha channel. The alpha channel is for blending which we won't be using in this tutorial.

Task 1: Try to change the values of the red, green, blue channel and recompile lab1_window. To recompile, go the build folder and run

- make

Re-run lab1_window to see if the window background is changed to a new color.

2. Draw a triangle

Let us now proceed to draw a 3D triangle into the OpenGL window.
You can start with the code provided in lab1_triangle.cpp.
We will follow the programmable graphics pipeline in modern OpenGL (instead of the fixed graphics pipeline). There are two shaders we will use: a vertex shader and a fragment shader. The shaders are written in GLSL, the OpenGL shading language.

We will define the triangle by its three vertices. Each vertex is represented by a tuple of 3D coordinates of x, y, and z. We will load the triangle into the vertex buffer, which can be accessed by a vertex shader that transforms each vertex and projects them on the screen. The fragment shader will then colour each pixel belonging to the triangle and output the final pixel value to the screen for display.

The shaders are stored as text files. Our program will load these files and call the OpenGL API to compile and link the shaders for our rendering. For simplicity, you are provided with a LoadShaders function that will load the vertex and fragment shader, compile, link, and attach the shaders to our OpenGL window. The code is provided in lab1_triangle.cpp.

(As an alternative, we can also store shaders as raw literal strings from C++ 11. This is demonstrated in Lab 1B, if you find loading shaders from files is not convenient enough.)

Let us now add the code to define the triangle (at TODO 1 location).

```
// Define a triangle with three (x, y, z)s
static const GLfloat vertex_buffer_data[] = {
    -0.5f,
    -0.5f,
    0.0f,
    0.5f,
    -0.5f,
    0.0f,
    0.0f,
    0.5f,
    0.0f,
};
```

This array contains the x, y, z coordinates of each vertex in the triangle.
To render the triangle, let us load this array into the GPU memory and make it accessible by OpenGL. Particularly, we will load this data to a vertex array object in OpenGL. The vertex array object can have a vertex buffer object that contains this data. Continue to add the following code after the triangle definition code.

```
// Create a vertex array object
GLuint VertexArrayID;
glGenVertexArrays(1, &VertexArrayID);
glBindVertexArray(VertexArrayID);


// Create a vertex buffer object to store the vertex data
GLuint VertexBufferID;
```

```
    glGenBuffers(1, &VertexBufferID);
    glBindBuffer(GL_ARRAY_BUFFER, VertexBufferID);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertex_buffer_data), vertex_buffer_data,
GL_STATIC_DRAW);
```

Finally, let's compile our GLSL program before getting into the do-while loop for rendering the triangle.

```
    // Create and compile our GLSL program from the shaders
    GLuint programID = LoadShaders("../lab1/lab1_triangle.vert", "../lab1/lab1_triangle.frag");
    if (programID == 0)
    {
        std::cerr << "Failed to load shaders." << std::endl;
        return -1;
    }
```

In the do-while loop, after clearing the background with glClear function, let's draw the triangle as follows at the TODO 2 location.

```
        // Enable our GLSL program
        glUseProgram(programID);

        // This function lets OpenGL know about the format of our vertex buffer, which is a float
array of attributes, each attribute has three values of x, y, z
        glEnableVertexAttribArray(0);
        glBindBuffer(GL_ARRAY_BUFFER, VertexBufferID);
        glVertexAttribPointer(
            0,      // location of the buffer
            3,      // number of components per attribute
            GL_FLOAT, // type
            GL_FALSE, // normalized
            0,      // stride
            (void *)0 // array buffer offset
        );

        // Draw the triangle starting from vertex 0 for 3 vertices
        glDrawArrays(GL_TRIANGLES, 0, 3);
        glDisableVertexAttribArray(0);
```

The above code snippet activates our GLSL program so that the vertex and fragment shader can be called to process our triangle vertices and project them to the screen. We then make a draw call to draw the triangle. Note that we need to let OpenGL understand the values we stored in our vertex buffer. From OpenGL side, it is simply an array of floats, so we have to indicate the format of the buffer, i.e., we store the x, y, z coordinates consecutively. The function glVertexAttribPointer performs such an indication to OpenGL, before glDrawArrays performs the rendering.

Finally, let's do a bit of housekeeping by cleaning up the buffers after the loop at TODO 3:

```
    glDeleteBuffers(1, &VertexBufferID);
    glDeleteVertexArrays(1, &VertexArrayID);
    glDeleteProgram(programID);
```

Recompile and execute lab1_triangle. If you do it correctly, you will see the following screen.

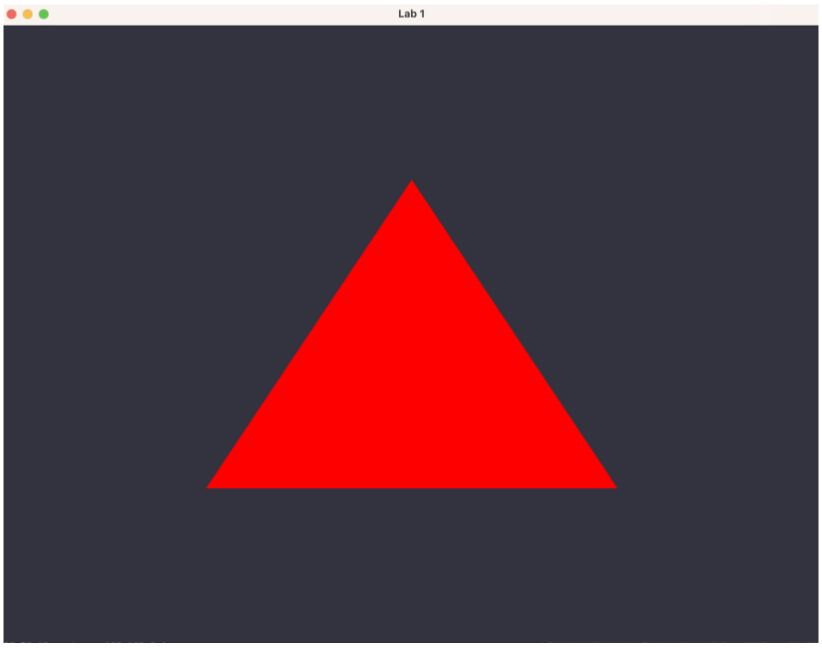


Figure 2: Draw a triangle

Task 2: So far we have not discussed what is inside the vertex and fragment shader. If you open lab1_triangle.vert and lab1_triangle.frag in a text editor, you will see their GLSL code, which looks very similar to C code.

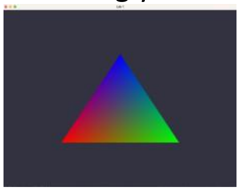
The vertex shader simply passes the vertex position on to the graphics pipeline.
The fragment shader simply outputs a red color (vector 1, 0, 0) to the screen.
Therefore, the triangle color is red.

Based on the current code, could you extend the current implementation to assign a different color to each vertex of the triangle?

Hint: you might consider defining the colors in a new array, load it into a vertex buffer, and use the following code for rendering:

```
glEnableVertexAttribArray(1);
glBindBuffer(GL_ARRAY_BUFFER, ColorBufferID);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, 0);
glDrawArrays(GL_TRIANGLES, 0, 3);
```

Note that this won't work as is. You will need to modify the vertex and fragment shader accordingly. The final result will look like below.



Finally, consider adding some simple movement to the triangle by changing the vertex positions in the vertex buffer data array while in the loop, and reload the vertex buffer.

Submission

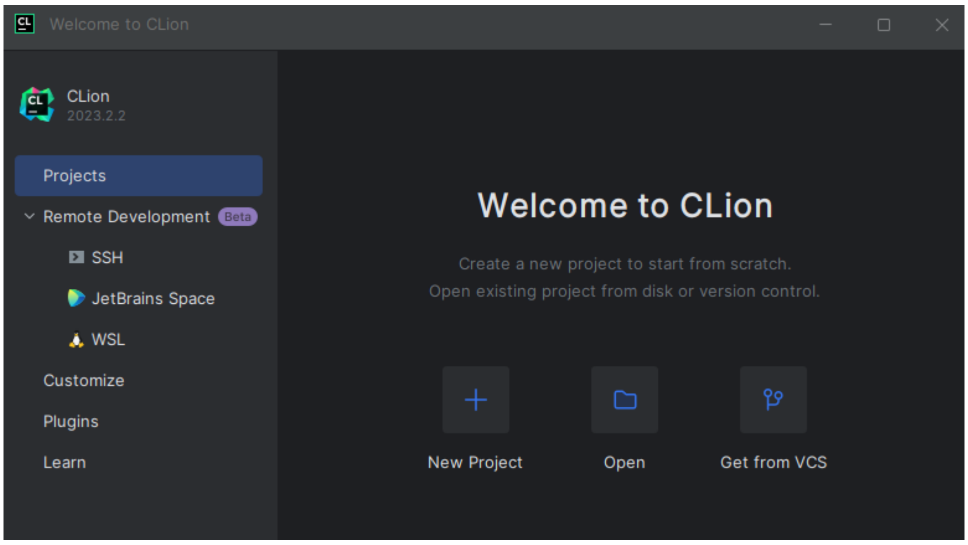
You do not need to submit Lab 1A.

Appendix

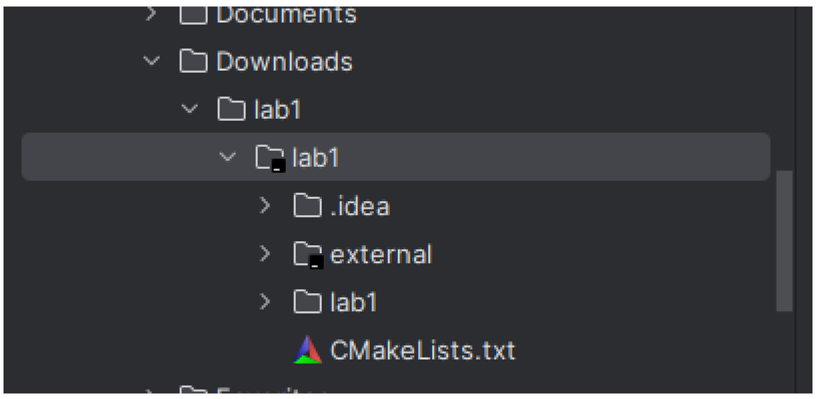
Window compilation

This guide is for compiling the code of this lab on an SCSS lab machine that runs a Windows operating system. We will use CLion to compile, but Visual Studio 2022 works well, too.

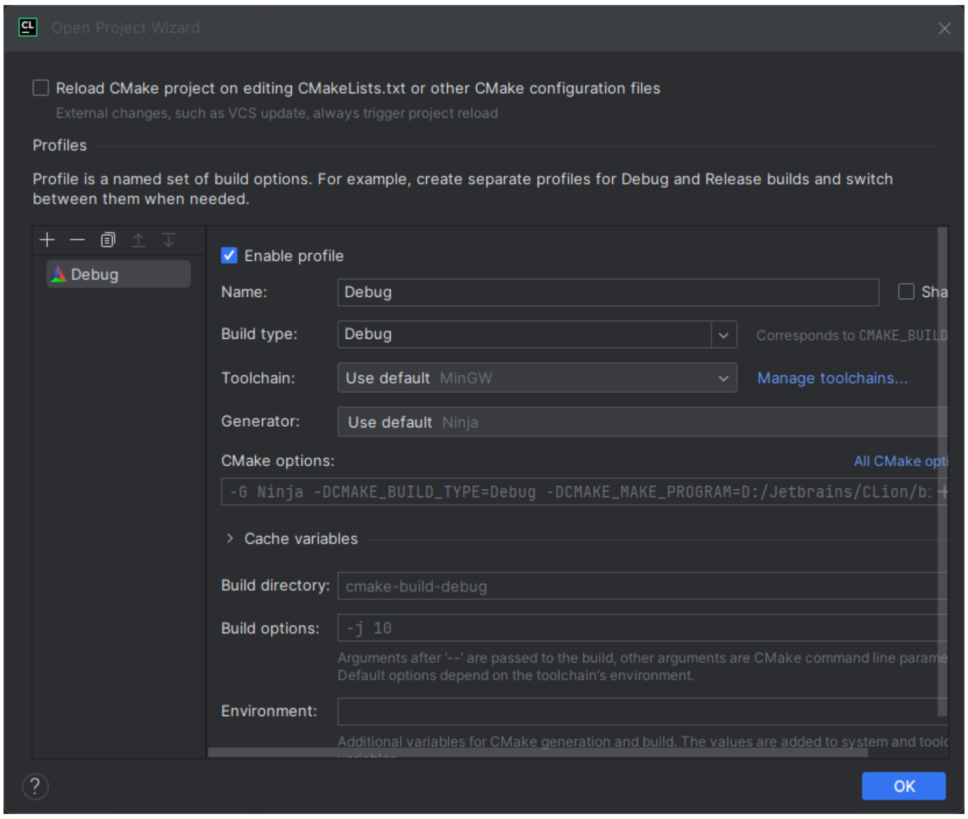
Open CLion. In the main screen, you will see:



Click on Open. Navigate to the lab1 folder. Note that this is the folder that contains the CMakeLists.txt.



After choosing the folder, the Open Project wizard will appear. Let us use the Debug mode to compile our program for now. Press OK.


Open Project Wizard

☐ Reload CMake project on editing CMakeLists.txt or other CMake configuration files

External changes, such as VCS update, always trigger project reload

Profiles

Profile is a named set of build options. For example, create separate profiles for Debug and Release builds and switch between them when needed.

Debug

☑ Enable profile

Name: Debug ☐ Sha

Build type: Debug Corresponds to CMAKE_BUILD

Toolchain: Use default MinGW Manage toolchains...

Generator: Use default Ninja

CMake options: All CMake opti

```
-G Ninja -DCMAKE_BUILD_TYPE=Debug -DCMAKE_MAKE_PROGRAM=D:/Jetbrains/CLion/b:
```

Cache variables

Build directory: cmake-build-debug

Build options: -j 10

Arguments after '--' are passed to the build, other arguments are CMake command line parame
Default options depend on the toolchain's environment.

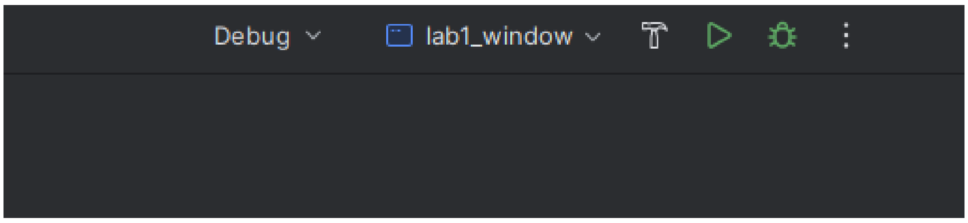
Environment:

Additional variables for CMake generation and build. The values are added to system and toolc

OK

CLion C++ editor will now open. On the top center of your window, you will see the current compilation setting already set to Debug, and the application set to lab1_triangle. Choose the application you want to compile from the dropdown menu, in our case, choose lab1_window.

Press the hammer button to build. CLion will then call cmake on CMakeLists.txt in lab1 folder, generates the Makefile and create an executable of our program, in this case lab1_window.


Debug lab1_window

Press the play button to launch lab1_window. You will see a blank screen with a dark blue background.