SCIT

School of Computing & Information Technology

CSCI336 – Interactive Computer Graphics

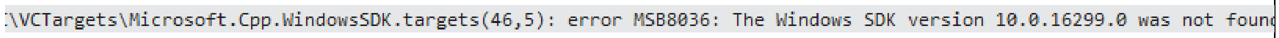
Getting Started

The aim of this lab is to introduce you to the GLFW library, which is a simple API for creating windows and contexts, receiving input and handling events.

Before you start this lab, you should go through the "Setting up the Environment" document, which shows how to set up a project. While the project for this lab has already been set up for you, it is a good idea to know how to set it up yourself.

Visual Studio only:

If you get an error that looks something like this:



```
\VCTargets\Microsoft.Cpp.WindowsSDK.targets(46,5): error MSB8036: The Windows SDK version 10.0.16299.0 was not foun
```

This means the Windows SDK version on your system is different from the one used in the project. To fix this, please refer to the Appendix at the end of this document.

Header Files

Have a look at the code in lab1.cpp. The code starts by including the relevant C++ header files.

```
#include <cstdio>
#include <iostream>
```

These are included so that we can have C++ input and output, e.g., for displaying messages to the console.

If you do not want to keep having to type `std::`, you can uncomment the following:

```
//using namespace std;
```

Next, the GLEW header file is included.

```
#include <GLEW/glew.h>
```

The OpenGL Extension Wrangler Library (GLEW) is a cross-platform open-source C/C++ extension loading library. GLEW provides efficient run-time mechanisms for determining which OpenGL extensions are supported on the target platform. The purpose of using GLEW is to access and load the OpenGL extensions that are available on the current platform. Without this, some platforms will only have OpenGL 1.2 functionality made available by default.

This is followed by including the GLFW version 3 header.

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

This header already includes the OpenGL header (i.e. opengl.h), so it should not be included again. Note that if you are including platform-specific headers (e.g., windows.h), these must be included before the GLFW header. Otherwise, they may conflict with GLFW.

Windows only:

The corresponding dynamic linked libraries (glew32.dll and glfw3.dll) must be located in the same directory.

Then, a couple of global variables are declared for the window width and height.

```
unsigned int gWindowWidth = 800;
unsigned int gWindowHeight = 600;
```

Creating a Window

Skip down the code to the main function. The first statement declares a pointer to the GLFW window handle, which will be used when dealing with the window.

```
GLFWwindow* window = nullptr;     // GLFW window handle
```

Next, you will see an error callback function being set. This is so that GLFW knows what function to call when an error event occurs.

```
glfwSetErrorCallback(error_callback);    // set GLFW error callback function
```

Most GLFW events are reported through callbacks, whether it is a key being pressed, a GLFW window being moved, an error occurring, etc. Callbacks are simply C functions (or C++ static methods) that are called by GLFW with arguments describing the event.

When a GLFW function fails, an error is reported to the GLFW error callback. You can receive these reports with an error callback. This function must have the following signature:

```
void error_callback(int error, const char* description)
```

The content in the callback function simply displays the error message if a GLFW function fails:

```
std::cerr << description << std::endl;   // output error description
```

Before you can use most GLFW functions, the library must be initialised. On successful initialisation, GLFW_TRUE (i.e. 1) is returned. If an error occurred, GLFW_FALSE (i.e. 0) is returned.

```
if (!glfwInit())
{
       // if failed to initialise GLFW
       exit(EXIT_FAILURE);
}
```

When you are done using GLFW, it should be terminated with:

```
glfwTerminate();
```

This destroys any remaining windows and releases any other resources allocated by GLFW.

Before creating a window, we can specify the minimum OpenGL version (if not supported, window creation will fail). By default, the OpenGL context that GLFW creates may be of any version. The following specifies OpenGL 3.3 core profile as the minimum:

```
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
```

The window and its OpenGL context are created with a single call to **glfwCreateWindow()**, which returns a handle to the created combined window and context object. The following creates a window with the dimensions that were previously defined and a window title "Lab 1":

```
window = glfwCreateWindow(gWindowWidth, gWindowHeight, "Lab 1", nullptr, nullptr);
```

Note that this returns the window handle (or NULL if it failed). The window handle is passed to all GLFW window related functions and callbacks, so that they know which window received an event.

When a window and context is no longer needed, destroy it using:

```
glfwDestroyWindow(window);
```

Before you can use the OpenGL API, you must have a current OpenGL context:

```
glfwMakeContextCurrent(window);    // set window context as the current context
```

The next function sets how many frames to wait until swapping buffers[1]. By default, this is set to zero, which means that buffer swapping occurs immediately[2]. On fast machines, this potentially wastes CPU and/or GPU cycles as the majority of monitors refresh rate at ~60Hz (monitors specialised for gaming may have higher refresh rates).

```
glfwSwapInterval(1);               // swap buffer interval
```

Next, GLEW is initialised (its purpose was described above):

```
if (glewInit() != GLEW_OK)
{
       // if failed to initialise GLEW
       std::cerr << "GLEW initialisation failed" << std::endl;
       exit(EXIT_FAILURE);
}
```

The init() function is used for initialising the scene and render setting. Note that it is only called once before the rendering loop. In this example, it only contains a single OpenGL function which sets the clear background colour:

[1] Some GPUs ignore this setting.
[2] Depends on the system's settings.

```
glClearColor(0.2f, 0.2f, 0.2f, 1.0f);
```

The next section is the rendering loop:

```
while (!glfwWindowShouldClose(window))
{
	render_scene();			// render the scene

	glfwSwapBuffers(window);	// swap buffers
	glfwPollEvents();		// poll for events
}
```

Each time the body of the loop is run, it generates a single frame, makes it available for display (by swapping buffers) and checks for events.

Note the condition that is set for exiting the while loop. Each window has a flag indicating whether the window should be closed. When the user attempts to close the window, by either pressing the close widget in the title bar or using a key combination (like Alt+F4 on MS Windows or Command-W on macOS), this flag will be set to 1. Note that the window is not actually closed, so you are expected to monitor this flag and either destroy the window or give some kind of feedback to the user.

This calls the render scene function to update the scene every update cycle:

```
	render_scene();			// render the scene
```

All that the render scene function does is clear the colour buffer to the colour that was previously set using the **glClearColor** function:

```
static void render_scene()
{
	glClear(GL_COLOR_BUFFER_BIT);

	glFlush();
}
```

GLFW windows use double buffering by default. That means that each window has two rendering buffers; a front buffer and a back buffer. The contents in the front buffer are sent to the display device whenever it refreshes its display, whereas the back buffer is typically the buffer that you render to. When the rendering of a frame is complete, the buffers need to be swapped with one another, so the back buffer becomes the front buffer and vice versa using the following function:

```
	glfwSwapBuffers(window);	// swap buffers
```

The swap interval was previously set using the **glfwSwapInterval()** function.

GLFW needs to communicate regularly with the window system in order to receive events. The following checks for events every update cycle:

```
	glfwPollEvents();		// poll for events
```

Before exiting the program, the window is closed and GLFW is terminated:

```
glfwDestroyWindow(window);
glfwTerminate();
```

Compiling and running the program will create a window.

Keyboard and Mouse Input (You will have to add some of this code yourself)

In GLFW, callback functions can be set to receive various events.

To receive key press and release events, create a key callback function and set the callback function. Put this before the rendering loop:

```
glfwSetKeyCallback(window, key_callback);
```

The key press or release callback function receives the keyboard key, platform-specific scancode, key action (i.e. GLFW_PRESS, GLFW_REPEAT or GLFW_RELEASE) and modifier bits (e.g., for whether the SHIFT, ALT, CTRL keys are held):

```
static void key_callback(GLFWwindow* window, int key, int scancode, int action, int mods)
{
	// close the window when the ESCAPE key is pressed
	if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
	{
		// set flag to close the window
		glfwSetWindowShouldClose(window, GL_TRUE);
		return;
	}
}
```

Instead of using a callback function, key states for named keys are also saved in per window state arrays that can be checked using **glfwGetKey()**. For example:

```
if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
	glfwSetWindowShouldClose(window, GL_TRUE);
```

This function only returns a cached key event state. It does not poll the system for the current state of the key. Whenever you check the state, you risk missing the state change that you are looking for. If a pressed key is released again before you check its state, you will have missed the key press. The recommended solution for this is to use a key callback, but the GLFW_STICKY_KEYS input mode is also available.

```
glfwSetInputMode(window, GLFW_STICKY_KEYS, GL_TRUE);
```

When sticky keys mode is enabled, the state of a key will remain GLFW_PRESS until the state of that key is checked with **glfwGetKey()**. Once it has been checked, if a key release event had been processed in the meantime, the state will reset to GLFW_RELEASE, otherwise it will remain as GLFW_PRESS.

Mouse input comes in many forms, including cursor motion, button presses and scrolling offsets. We will look at mouse button input and obtaining the cursor position. Mouse button input is very similar to key press/release input.

Set the mouse callback function:

```
glfwSetMouseButtonCallback(window, mouse_button_callback);
```

Define the mouse callback function:

```
static void mouse_button_callback(GLFWwindow* window, int button, int action, int mods)
{
	// quit if the right mouse button is pressed
	if (button == GLFW_MOUSE_BUTTON_RIGHT && action == GLFW_PRESS)
	{
		// set flag to close the window
		glfwSetWindowShouldClose(window, GL_TRUE);
		return;
	}
}
```

Similar to keyboard input, a mouse button's cached state can be checked using **glfwGetMouseButton()**:

```
if (glfwGetMouseButton(window, GLFW_MOUSE_BUTTON_RIGHT) == GLFW_PRESS)
	glfwSetWindowShouldClose(window, GL_TRUE);
```

The mouse cursor position can be checked using **glfwGetCursorPos()**. The cursor position is measured in screen coordinates relative to the top-left corner of the window. For example:

```
double xpos, ypos;
glfwGetCursorPos(window, &xpos, &ypos);
```

Framebuffer Size

The framebuffer size of a window can be obtained using the **glfwGetFramebufferSize()** function. For example:

```
int width, height;
glfwGetFramebufferSize(window, &width, &height);
```

You can also set a framebuffer size change callback function that will be called when a window is resized. Its signature is as follows:

```
static void framebuffer_size_callback(GLFWwindow* window, int width, int height)
```

Set the window resize callback function using:

```
glfwSetWindowSizeCallback(window, framebuffer_size_callback);
```

Window Title

The window title can be set, using:

```
glfwSetWindowTitle(window, "MyWindow");
```

Note that the second argument takes a C-style string. So if using the C++ string class, the **c_str()** method will return a C-style string.

```
#include <string>

...

int a = 123;
std::string str = "a = " + std::to_string(a);
```

```
glfwSetWindowTitle(window, str.c_str());
```

You can find more information from the GLFW documentation. This can be found on the GLFW website: http://www.glfw.org/docs/latest/

Exercises

By now, you should have a basic grasp of the GLFW library. Try modifying the program to do the following:

- Change the background colour based on keyboard input.
- Quit the application when the right mouse button is pressed.
- Output mouse cursor coordinates to the console whenever the left mouse button is pressed within the window. Note how the mouse cursor coordinates change with respect to the top left corner of screen.
- Set the window title to the new framebuffer size when the window is resized.

References

Among others, much of the material in this lab was sourced from:

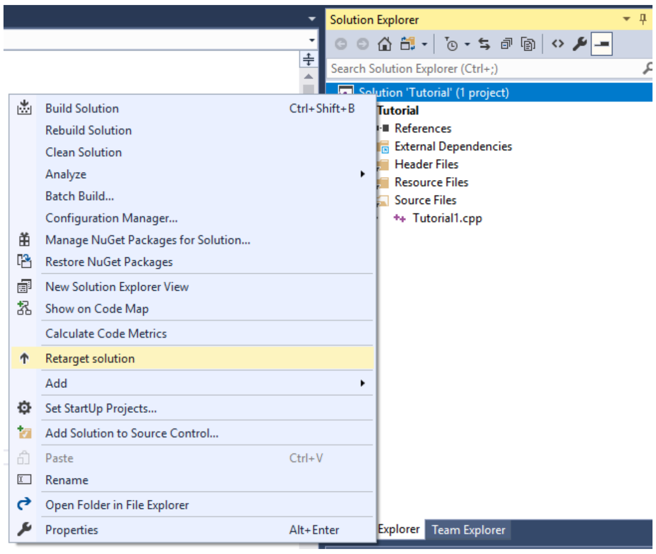
- http://www.glfw.org/
- http://glew.sourceforge.net/

Appendix

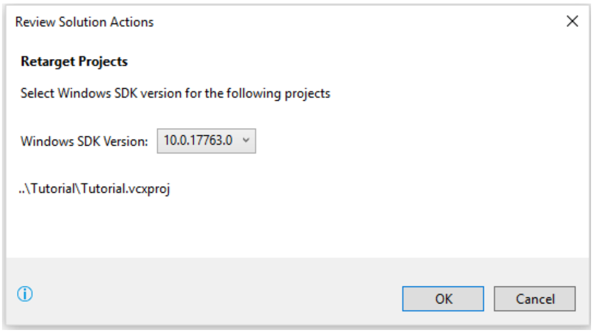
To fix the following error:

```
:\VCTargets\Microsoft.Cpp.WindowsSDK.targets(46,5): error MSB8036: The Windows SDK version 10.0.16299.0 was not found.
```

Right-click on the solution and select 'Retarget solution':



This will display the Windows SDK versions on your system, select an appropriate one and click OK:


Review Solution Actions

Retarget Projects

Select Windows SDK version for the following projects

Windows SDK Version: 10.0.17763.0

..\Tutorial\Tutorial.vcxproj

Done.