

# Lecture 3

## Basic Drawing in OpenGL

**CS104: Fundamentals of Computer Graphics**

**Hon-Cheng WONG**

**Faculty of Information Technology**

Slides are mainly based on the textbook



**Macau University of Science and Technology**  
**Macao, China**

Core and supporting libraries  
used in this course



## ⌘ Core library: **OpenGL**

☑ Provided by Visual Studio 2019 SDK

## ⌘ Supporting libraries:

☑ Window management: **GLFW**

☑ Extension library: **GLEW**

☑ Math library: **GLM**

☑ Texture management: **SOIL2**

## Window management: GLFW



- ⌘ OpenGL doesn't actually draw to a computer screen. Rather, it renders to a frame buffer, and it is the job of the individual machine to then draw the contents of the frame buffer onto a window on the screen.
- ⌘ Many libraries can do this job: **GLUT**, **freeglut**, **CPW**, **GLOW**, and **GLUI**.
- ⌘ One of the most popular options, and the one used in this book, is **GLFW**, which has built-in support for Windows, Macintosh, Linux, and other systems. It must be compiled on the machine where it is to be used.

## Extension library: GLEW



- ⌘ OpenGL is organized around a set of base functions and an extension mechanism used to support new functionality as technologies advance.
- ⌘ Modern versions of OpenGL require identifying the extensions available on the GPU. Identifying them involves several rather convoluted lines of code that would need to be performed for each modern command used. It has become standard practice to use an extension library to take care of these details.
- ⌘ A commonly used library among those listed is **GLEW**.

## Math library: GLM



- ⌘ 3D graphics programming makes heavy use of vector and matrix algebra. For this reason, use of OpenGL is greatly facilitated by accompanying it with a function library or class package to support common mathematical tasks.
- ⌘ The most popular one is OpenGL Mathematics, usually called **GLM**, which is a header-only C++ library compatible with Windows, Macintosh, and Linux.
- ⌘ **GLM** provides classes and basic math functions related to graphics concepts, such as vectors and matrix.

## Texture management: SOIL2



- ⌘ When we need to add “texture” to the objects in our graphics scenes, we frequently need to load such image files into our C++/OpenGL code.
- ⌘ It is generally preferable to use texture loading library. Some examples are **FreeImage**, **DevIL**, **GLI**, and **Glow**.
- ⌘ The most commonly used OpenGL image loading library is Simple OpenGL Image Loader (**SOIL**).
- ⌘ **SOIL2** is an updated fork of SOIL and it is compatible with a wide variety of platforms.

Related files of  
the core and supporting libraries



⌘ OpenGL: **opengl32.lib** (in VS 2019 SDK)

⌘ GLFW: **glfw3.lib**

(built with the source code)

⌘ GLEW: **glew32.lib, glew32.dll**

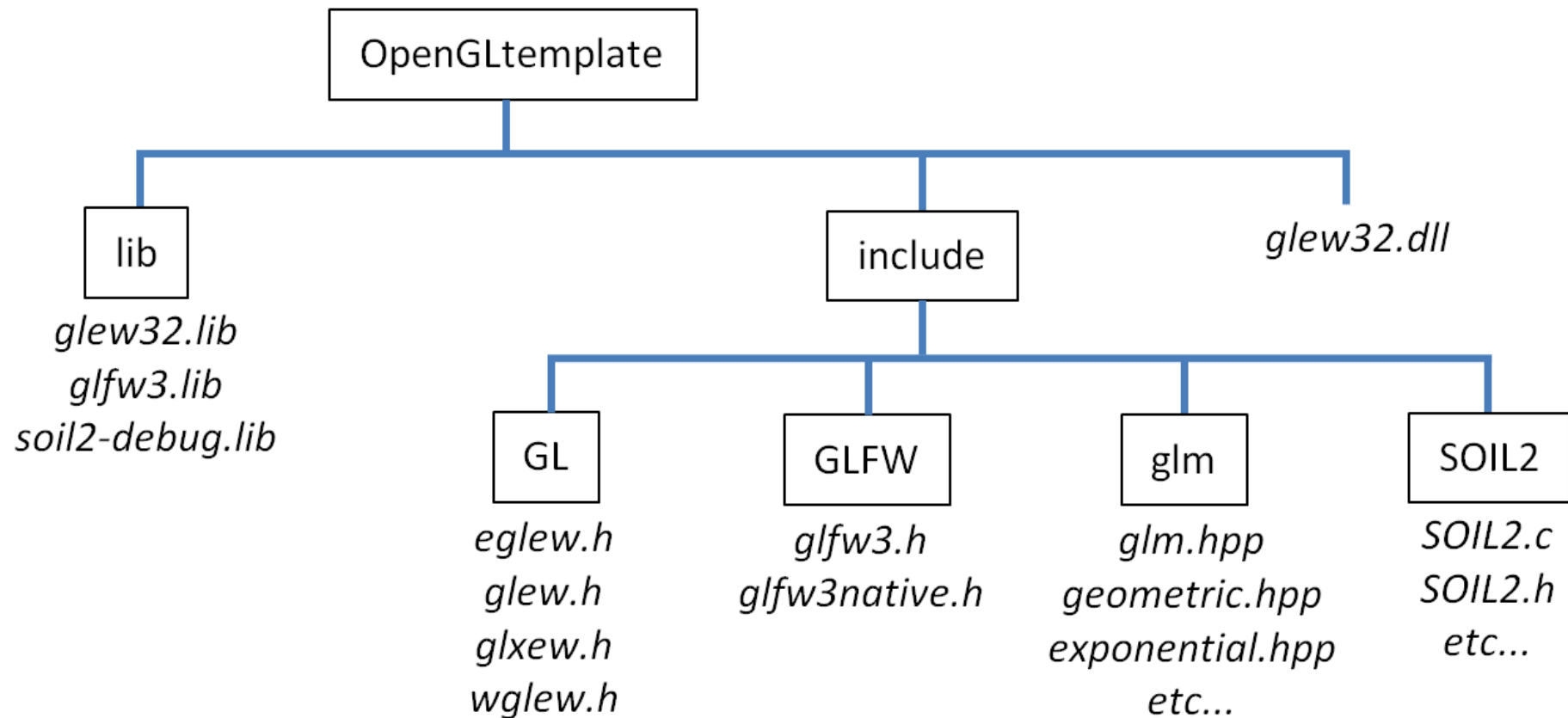
(32-bit binaries)

⌘ GLM: source codes in the folder “**glm**”

⌘ SOIL2: **soil2-debug.lib**

(built with the source code)

## Suggested library folder structure



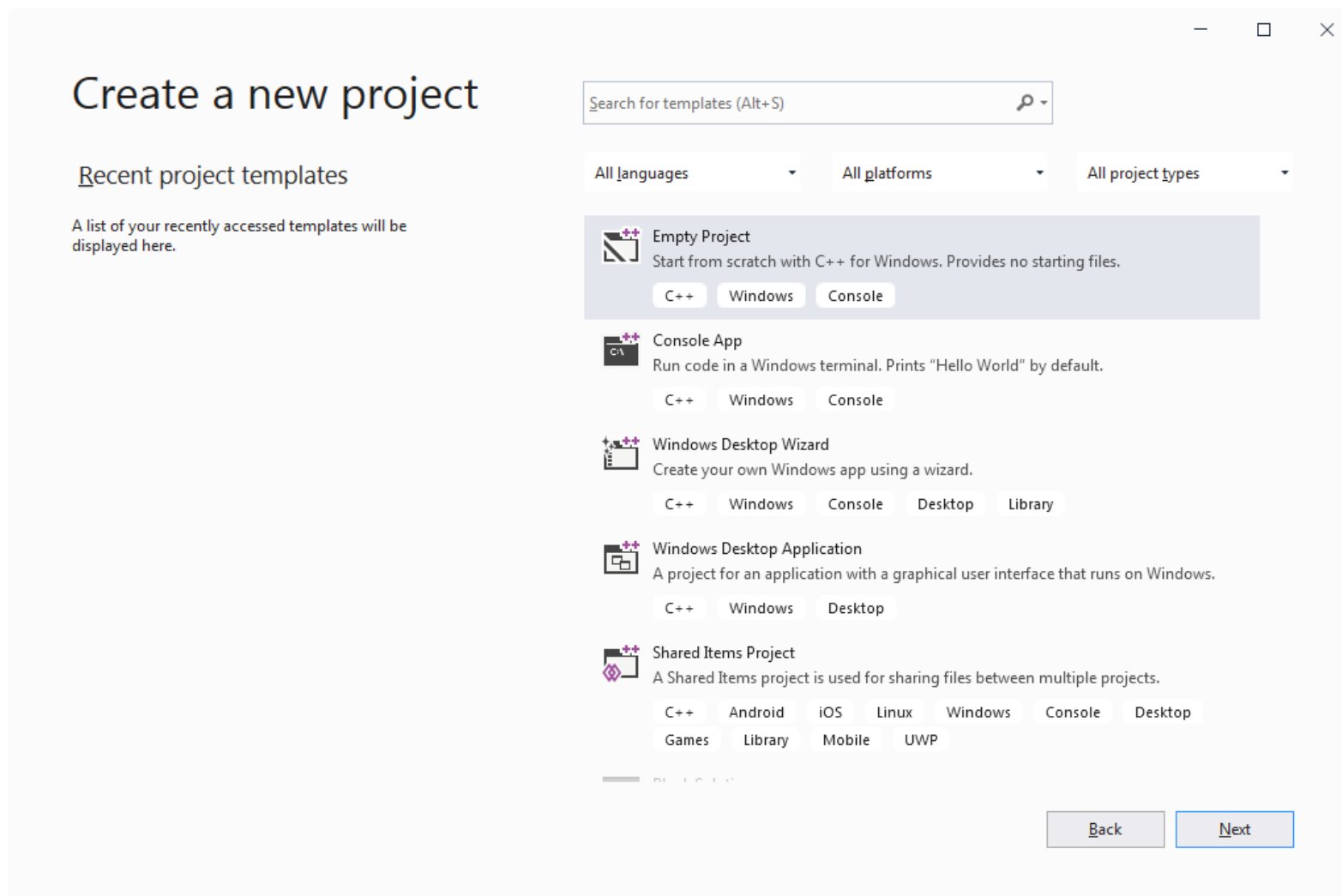


## Using Visual Studio 2019 for OpenGL



- ⌘ Create a new project;
- ⌘ Choose **Empty Project** as template;
- ⌘ Configure your new project;
- ⌘ **Add New Item** or **Add Existing Item** to **Source Files**;
- ⌘ **Add New Item** or **Add Existing Item** to **Header Files**;
- ⌘ **Add paths** to “include” and “lib” folders under project properties;
- ⌘ **Add libraries** (four libraries) under project properties;
- ⌘ **Build->Compile**;
- ⌘ **Build->Build Solution**;
- ⌘ Run it within **Visual Studio** or in **command prompt**.

# Using Visual Studio 2019 for OpenGL



# Using Visual Studio 2019 for OpenGL

Configure your new project

Empty Project C++ Windows Console

Project name

Program2\_1

Location

E:\OpenGLtemplate\

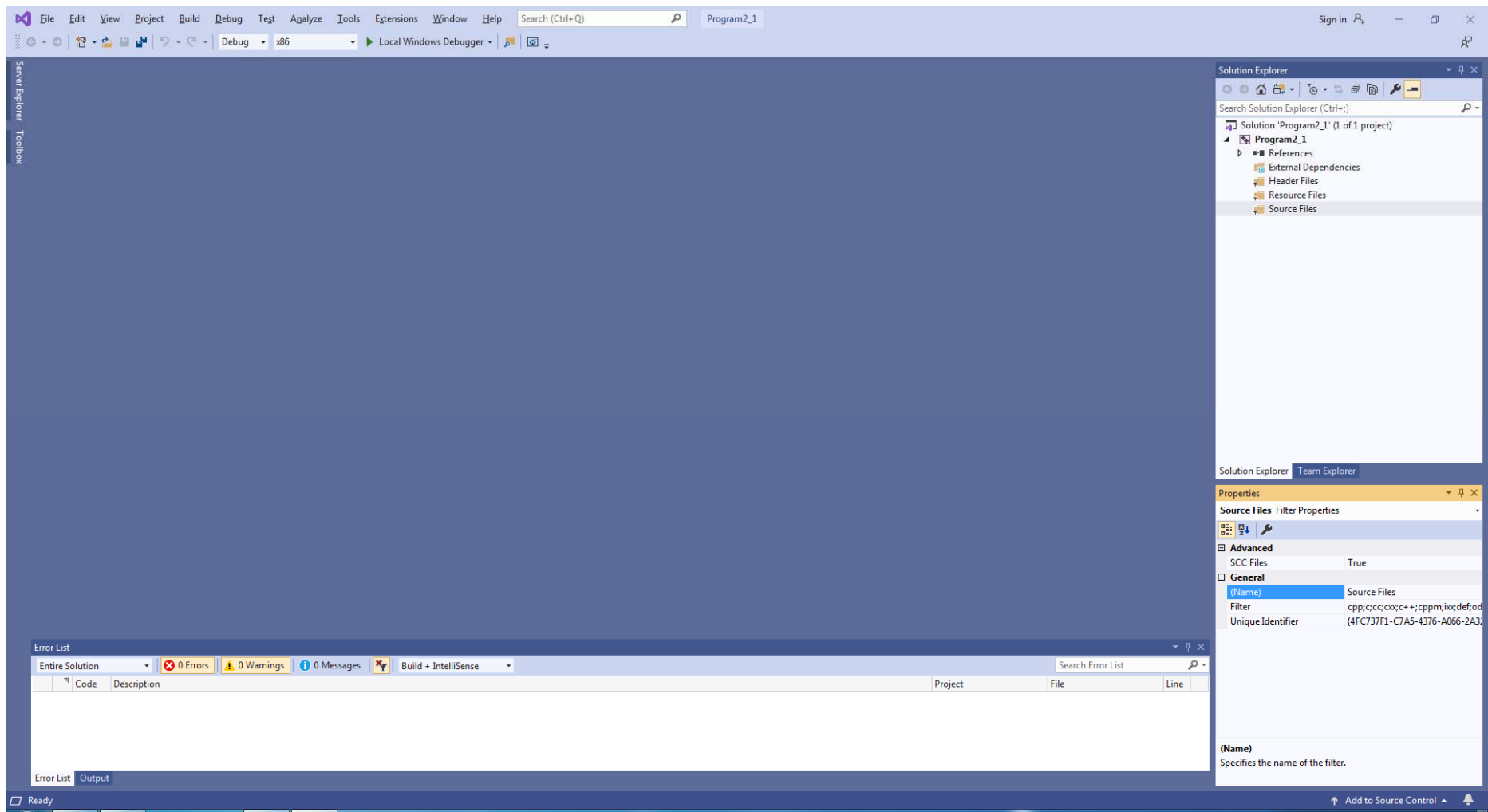
Solution name ⓘ

Program2\_1

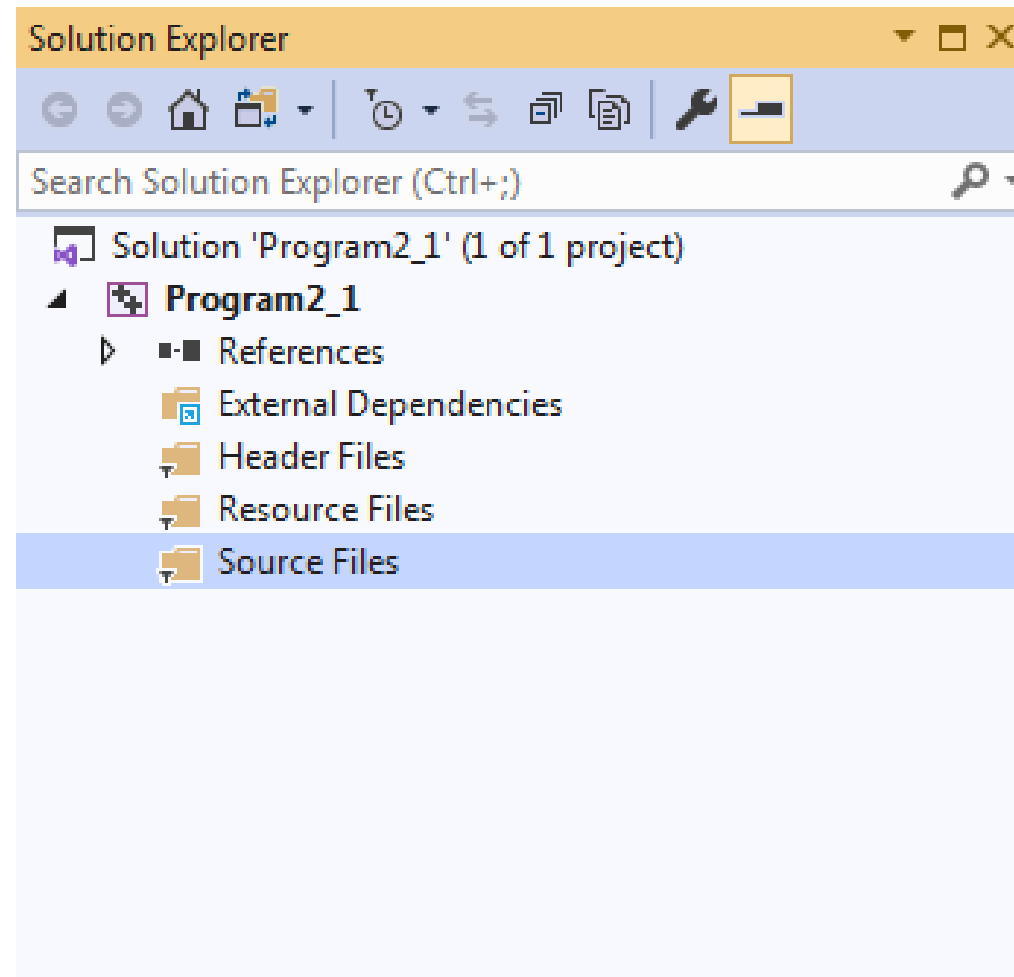
☐ Place solution and project in the same directory

Back Create

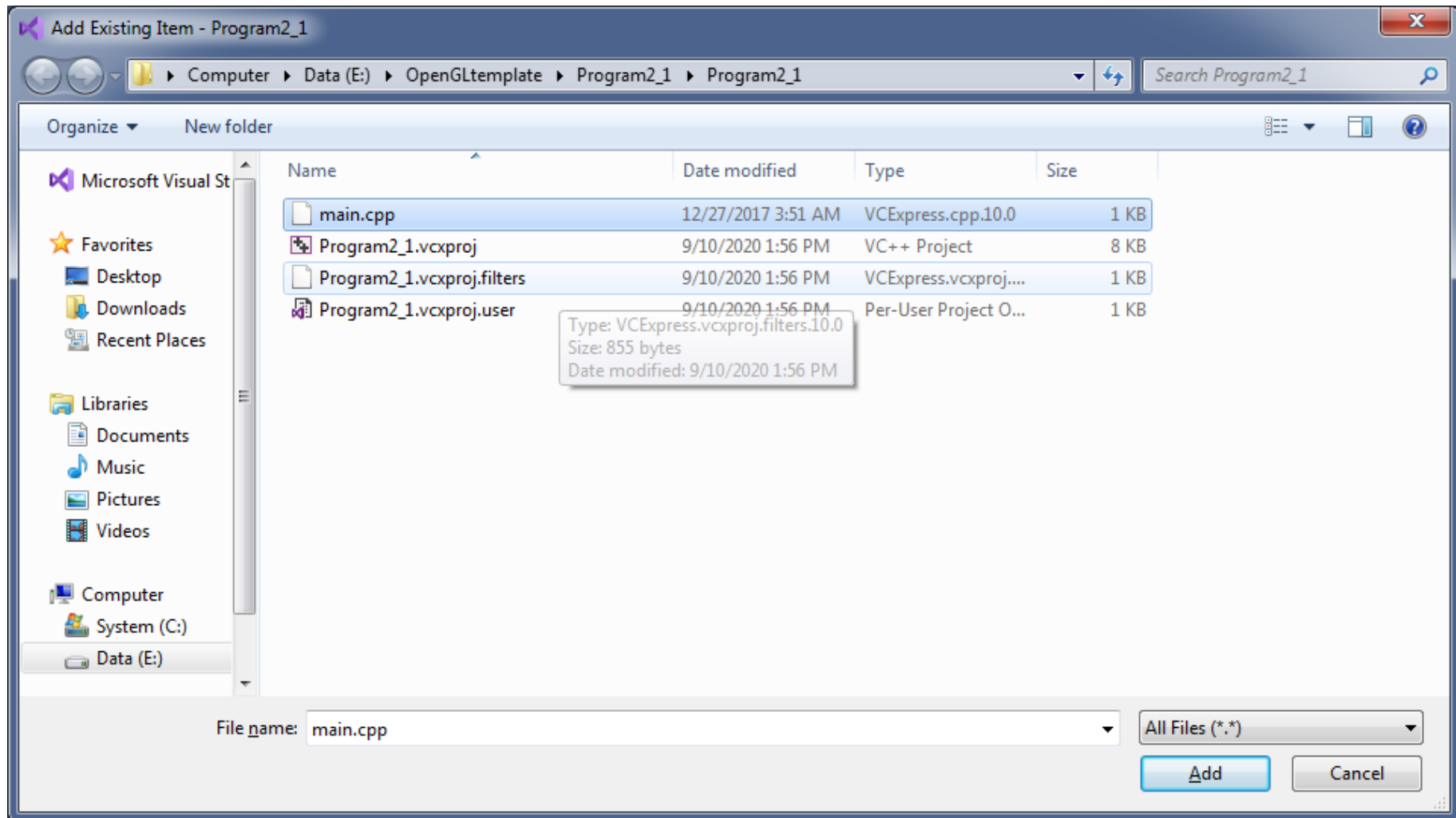
# Using Visual Studio 2019 for OpenGL



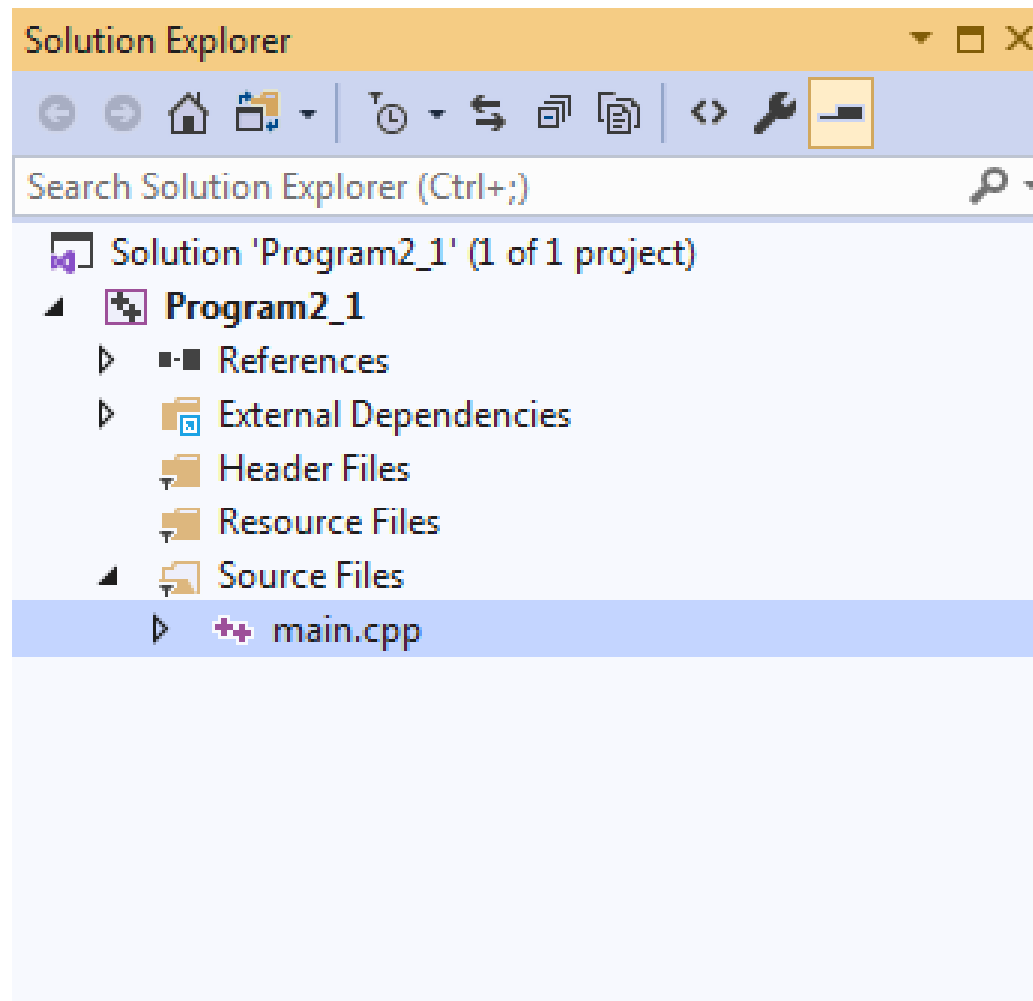
# Using Visual Studio 2019 for OpenGL



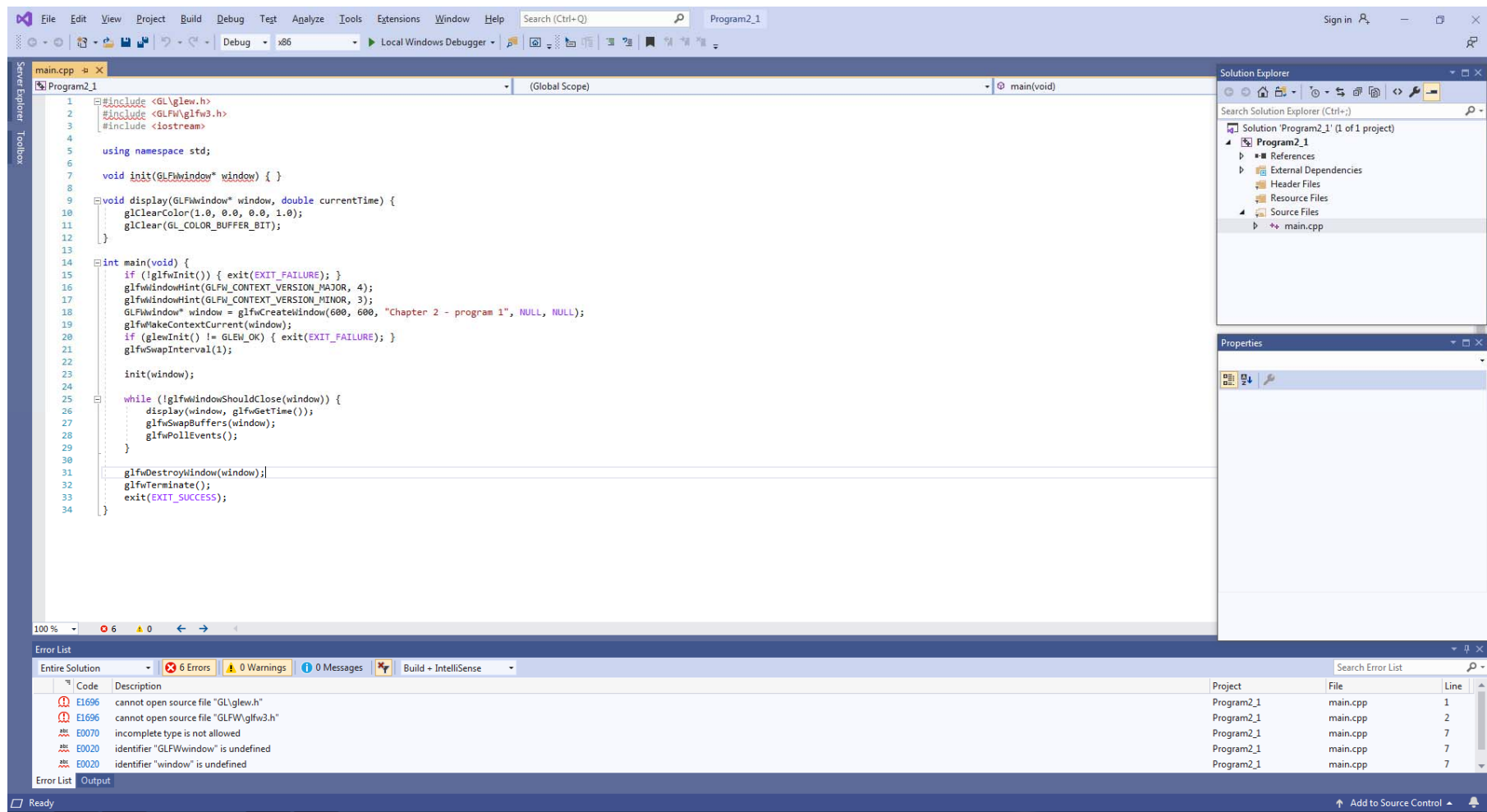
# Using Visual Studio 2019 for OpenGL



# Using Visual Studio 2019 for OpenGL

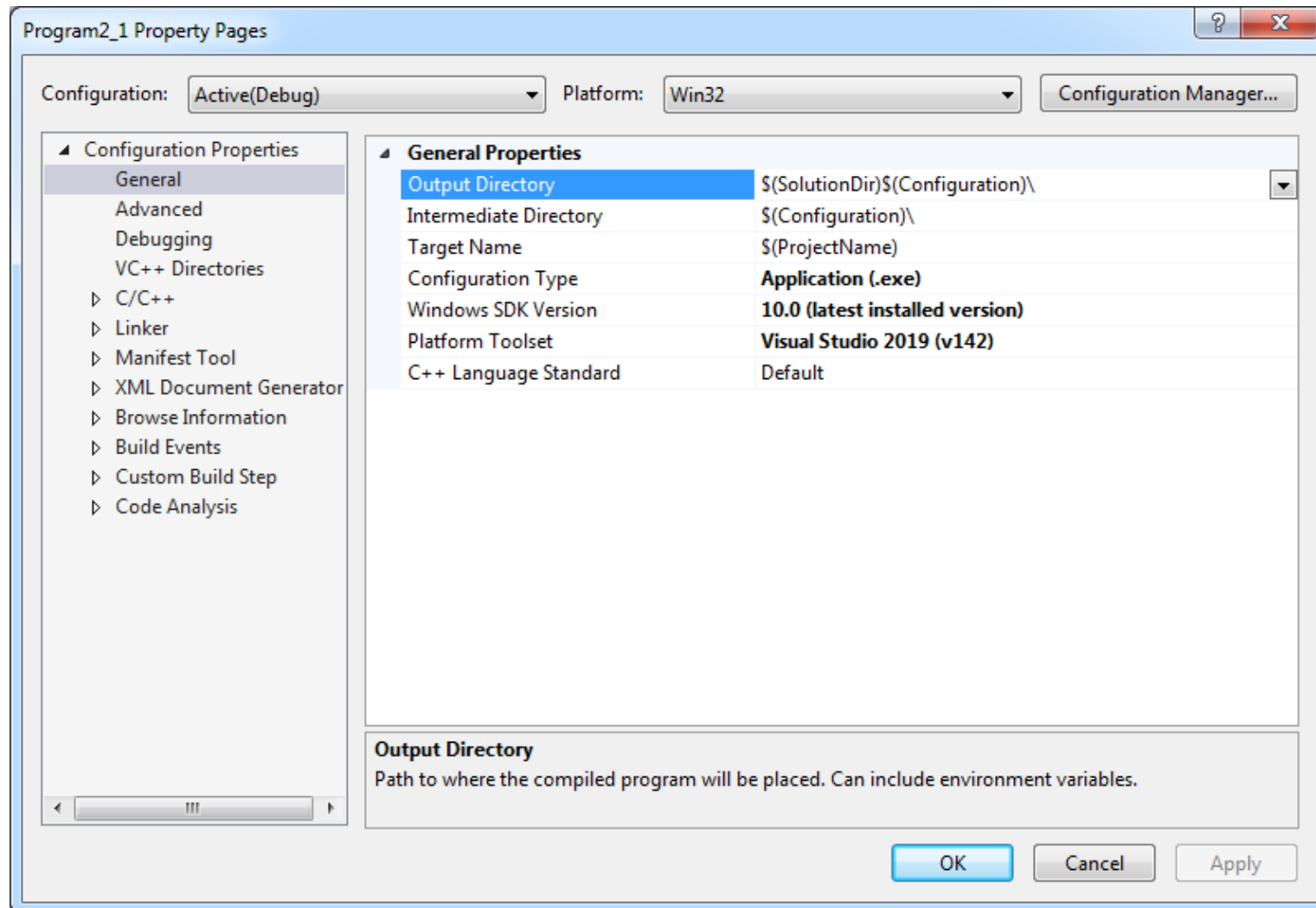


# Using Visual Studio 2019 for OpenGL

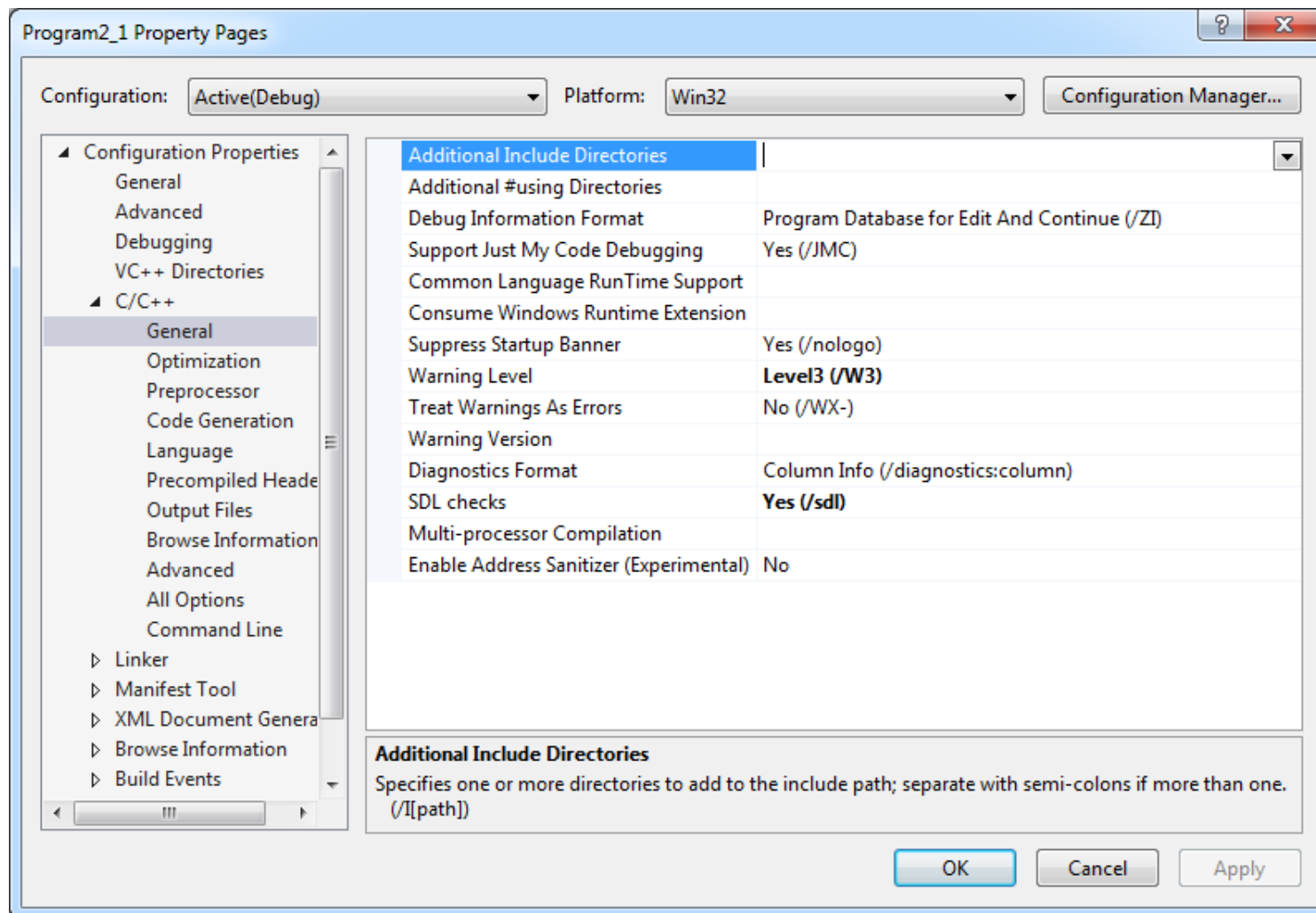




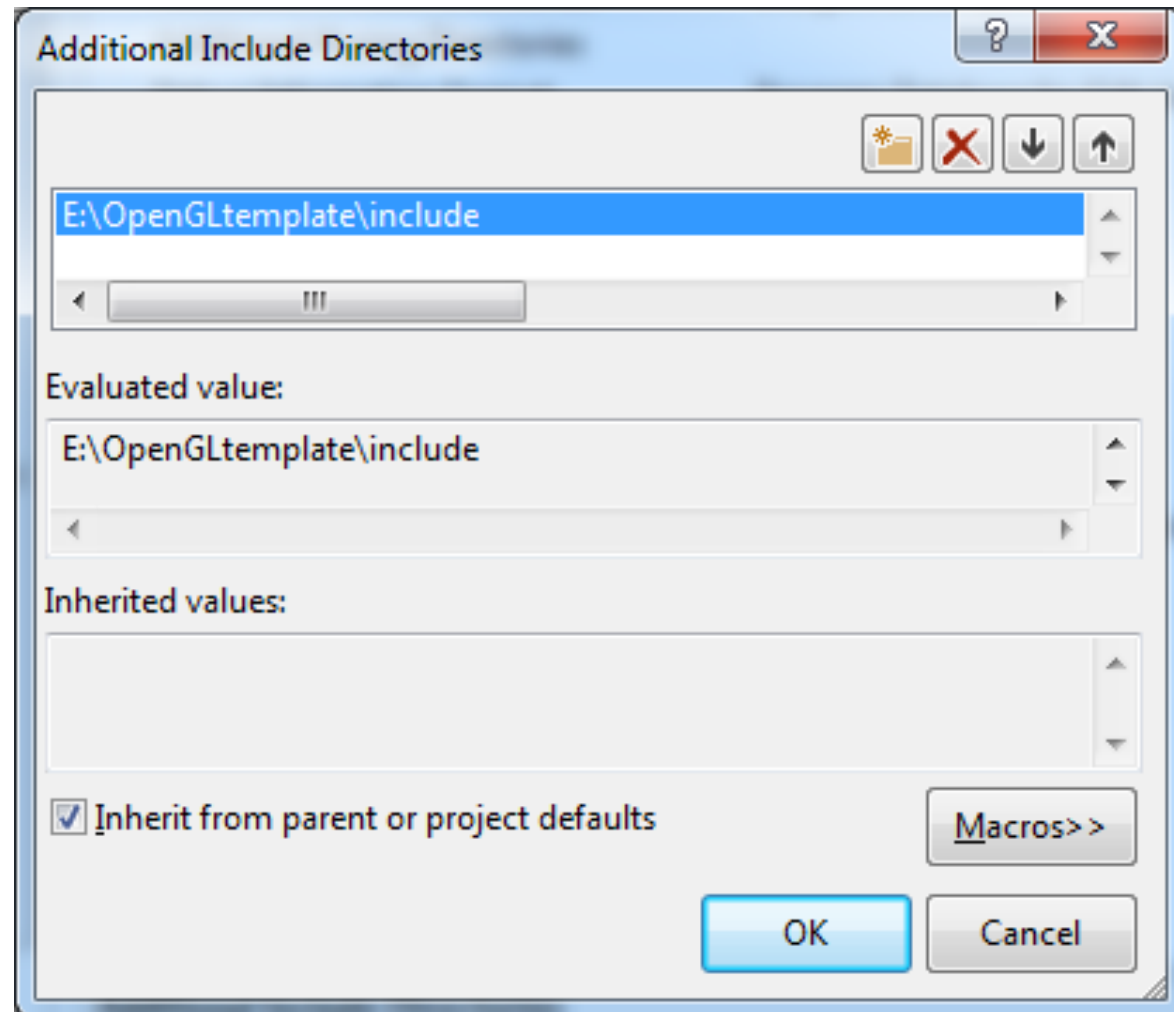
# Using Visual Studio 2019 for OpenGL



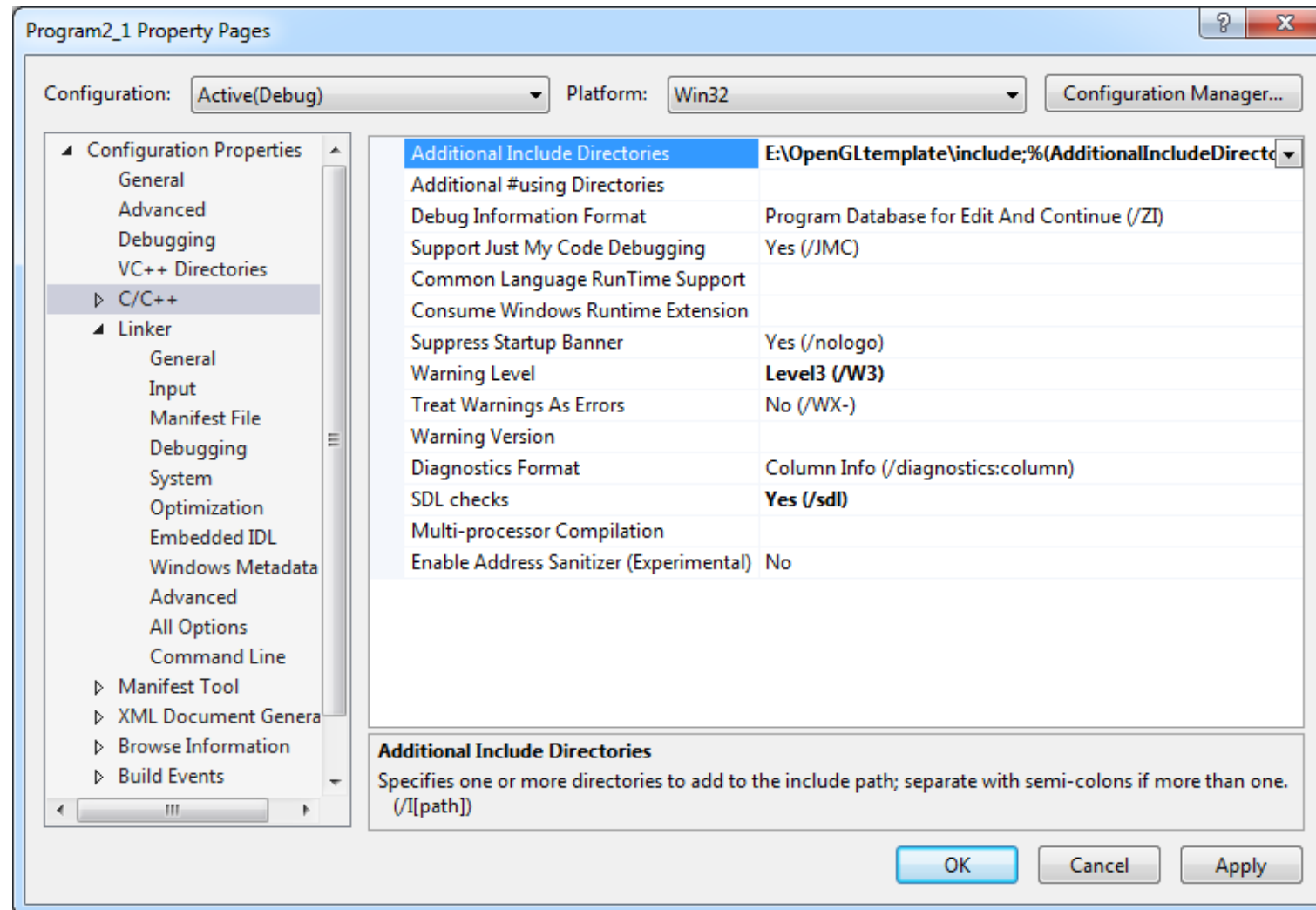
# Using Visual Studio 2019 for OpenGL



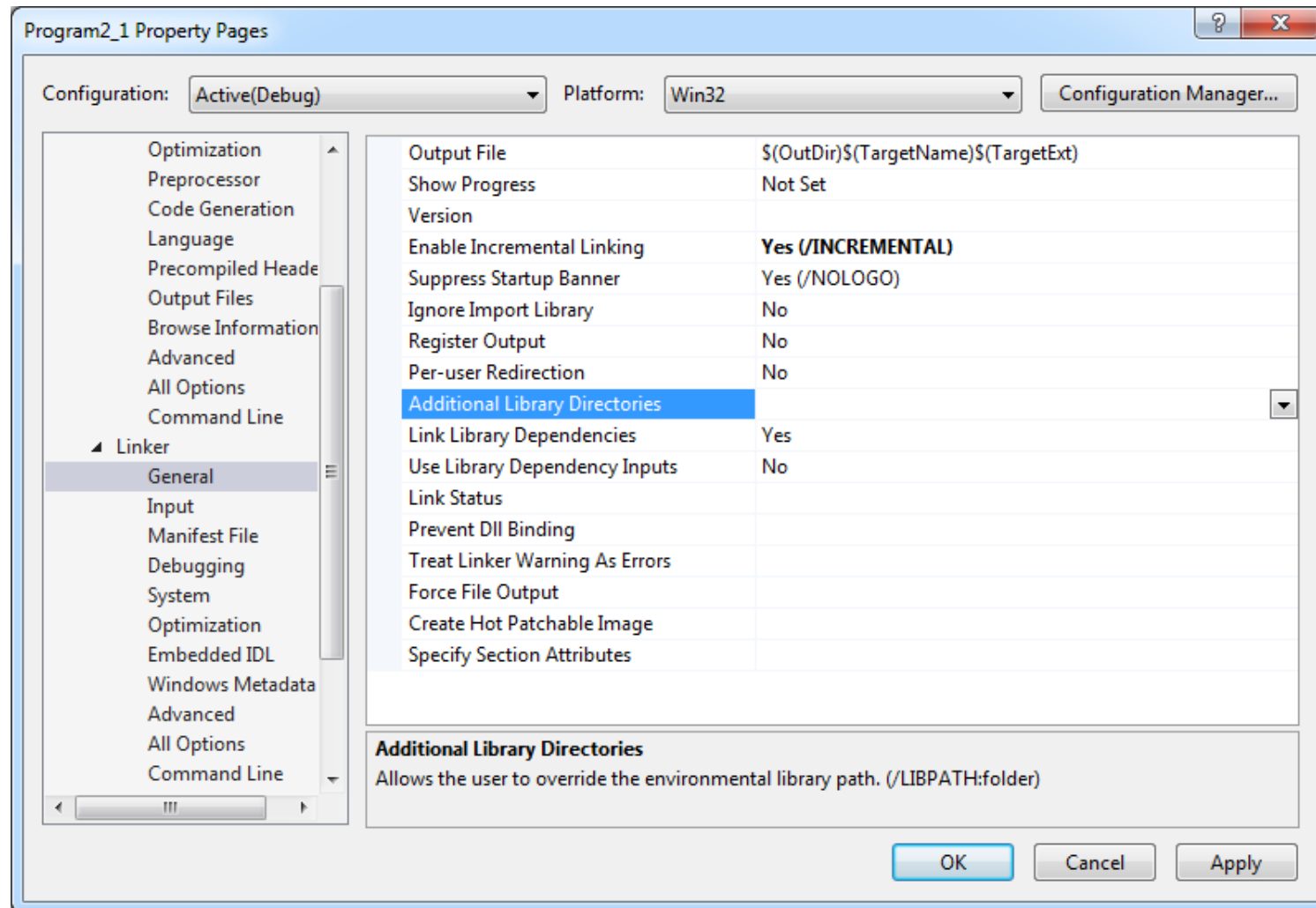
## Using Visual Studio 2019 for OpenGL



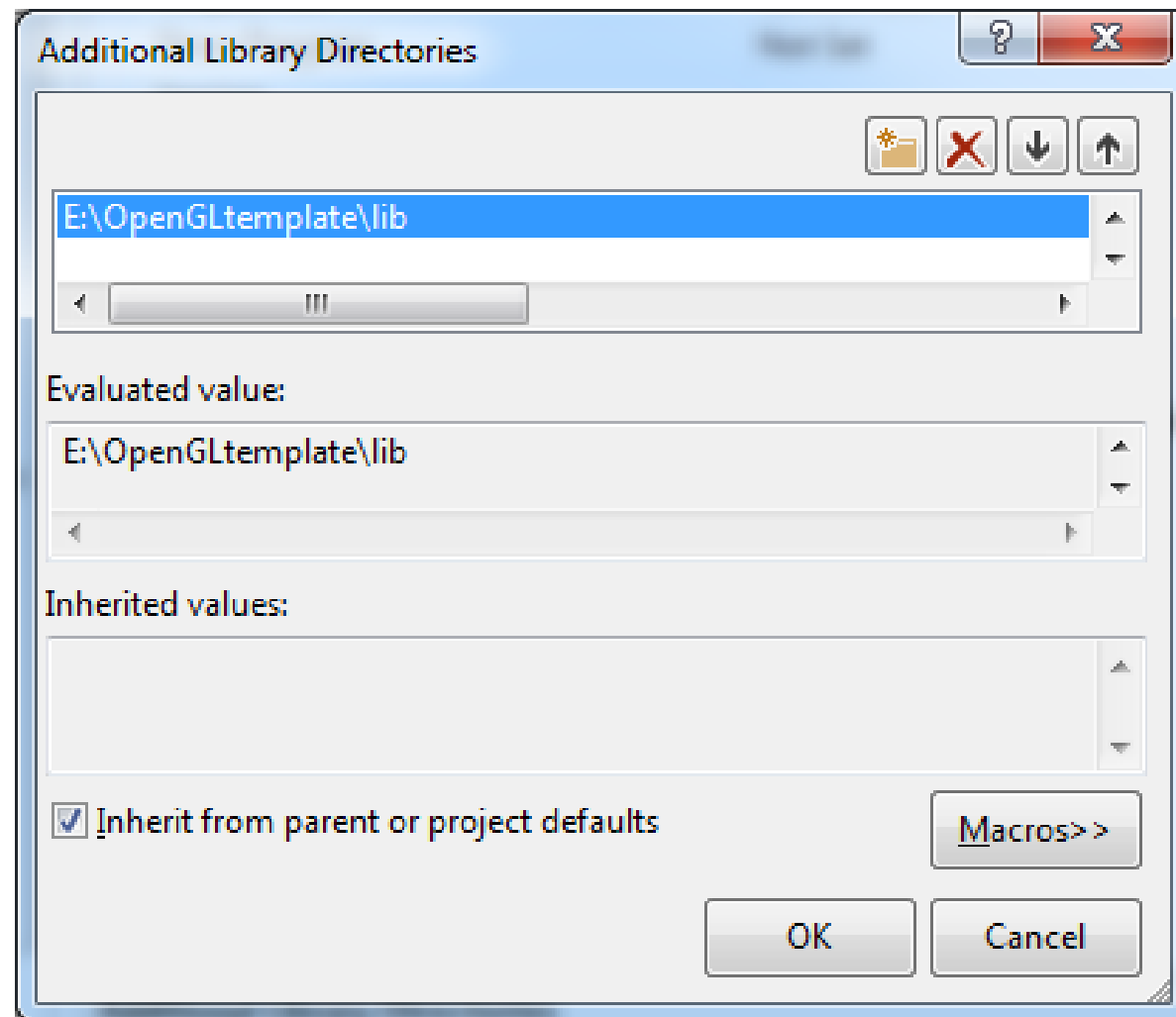
# Using Visual Studio 2019 for OpenGL



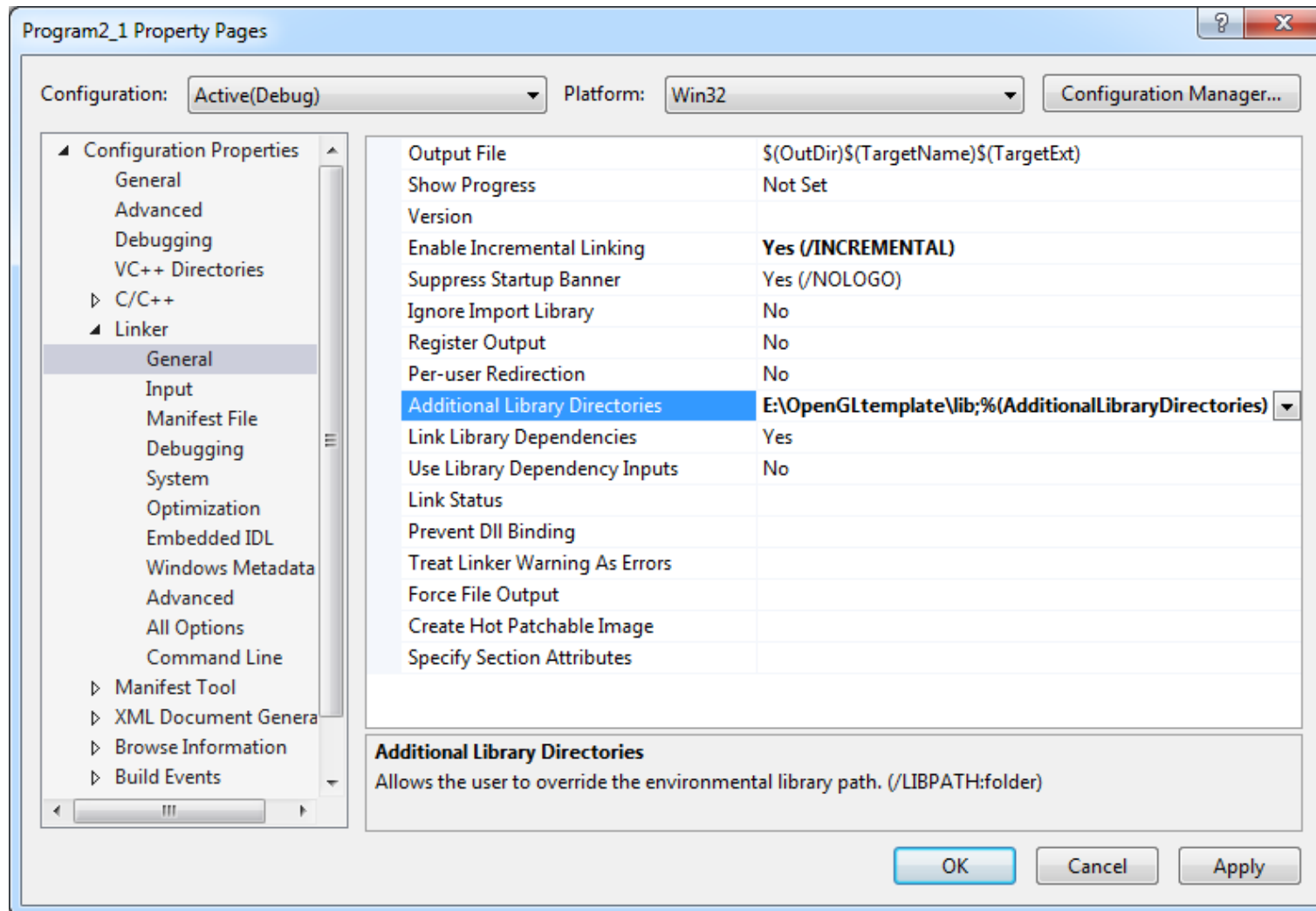
# Using Visual Studio 2019 for OpenGL



## Using Visual Studio 2019 for OpenGL



# Using Visual Studio 2019 for OpenGL

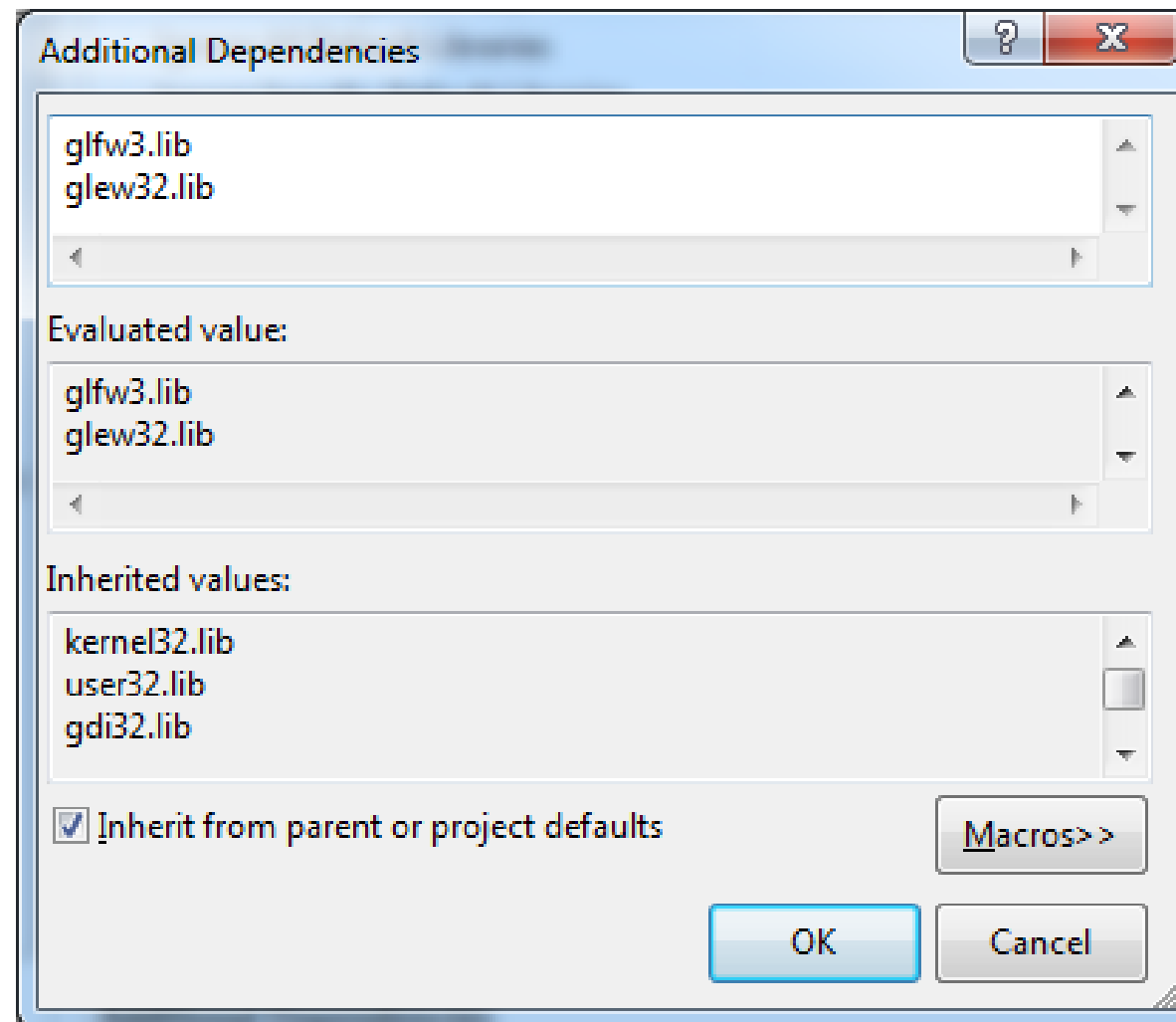


1. *Journal of the American Medical Association*, 2000; 284: 2689-2695.

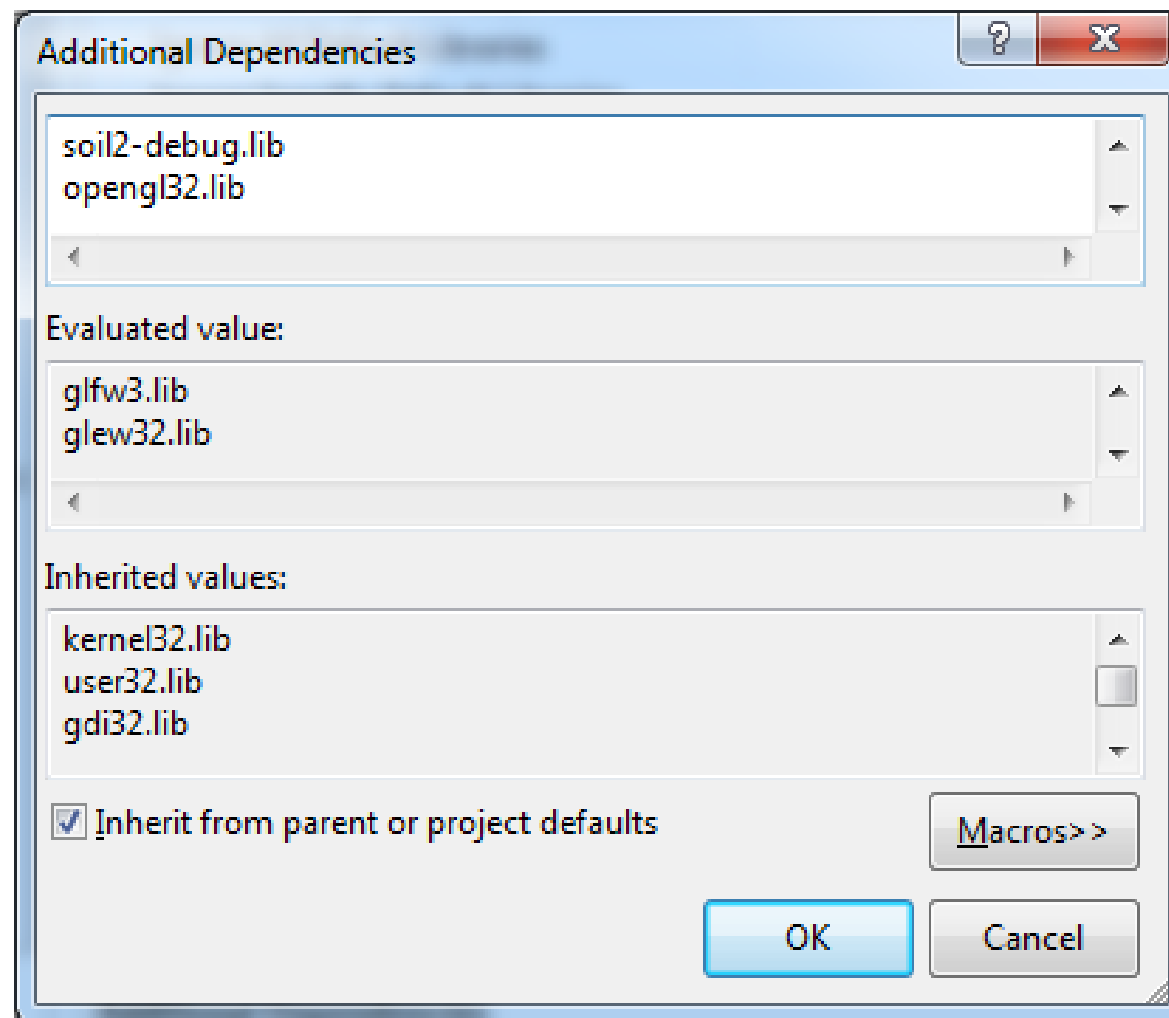




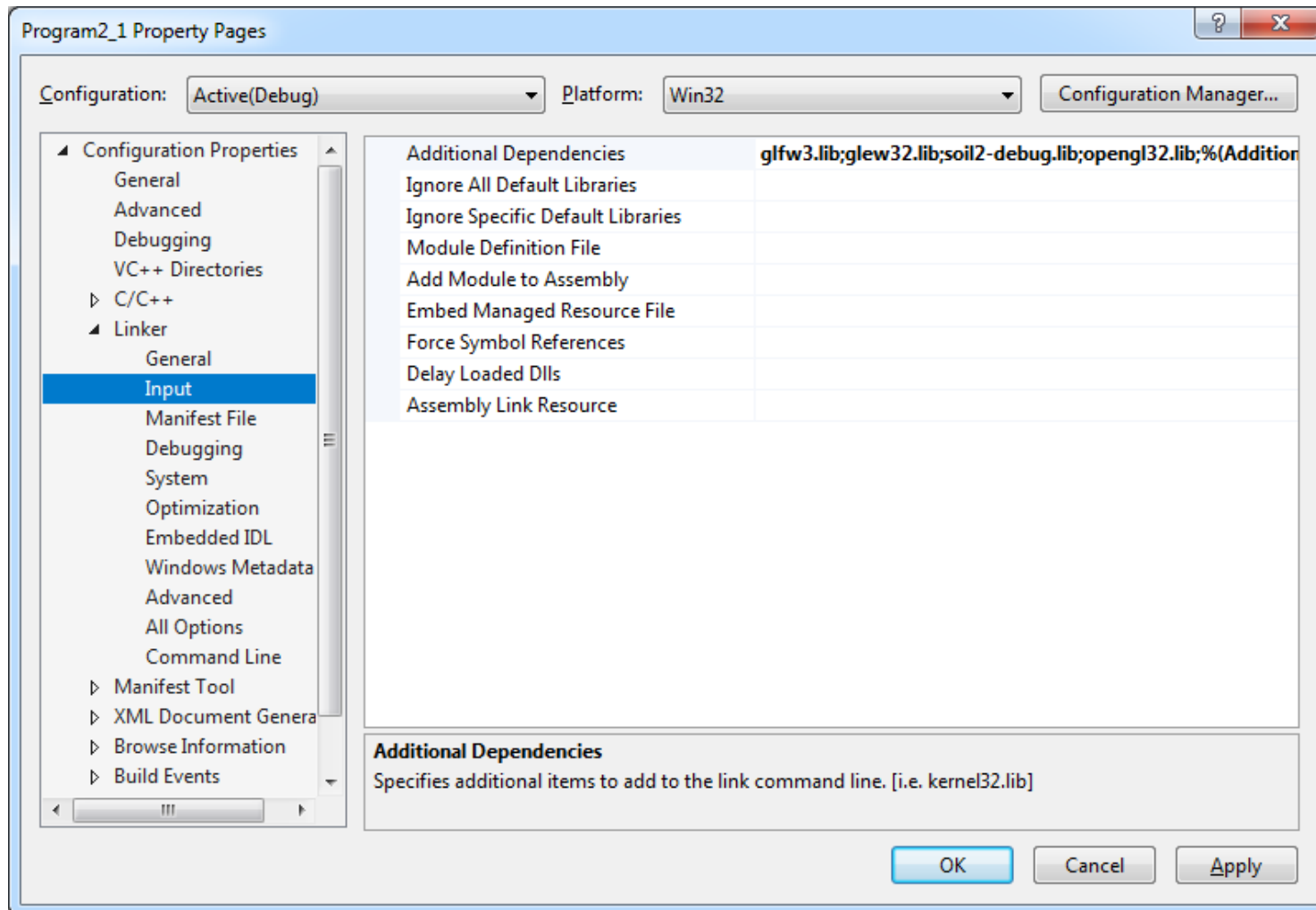
## Using Visual Studio 2019 for OpenGL



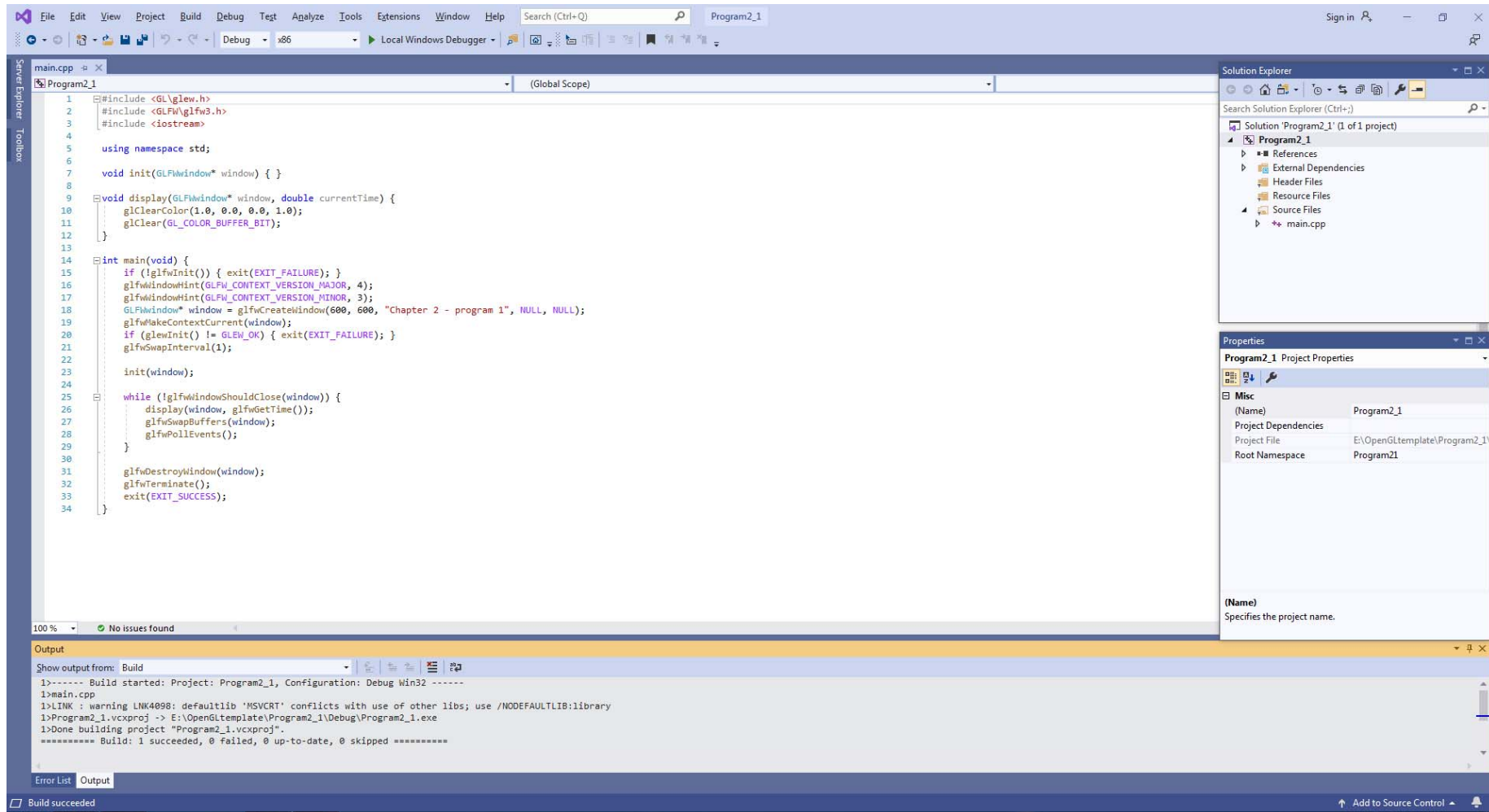
## Using Visual Studio 2019 for OpenGL



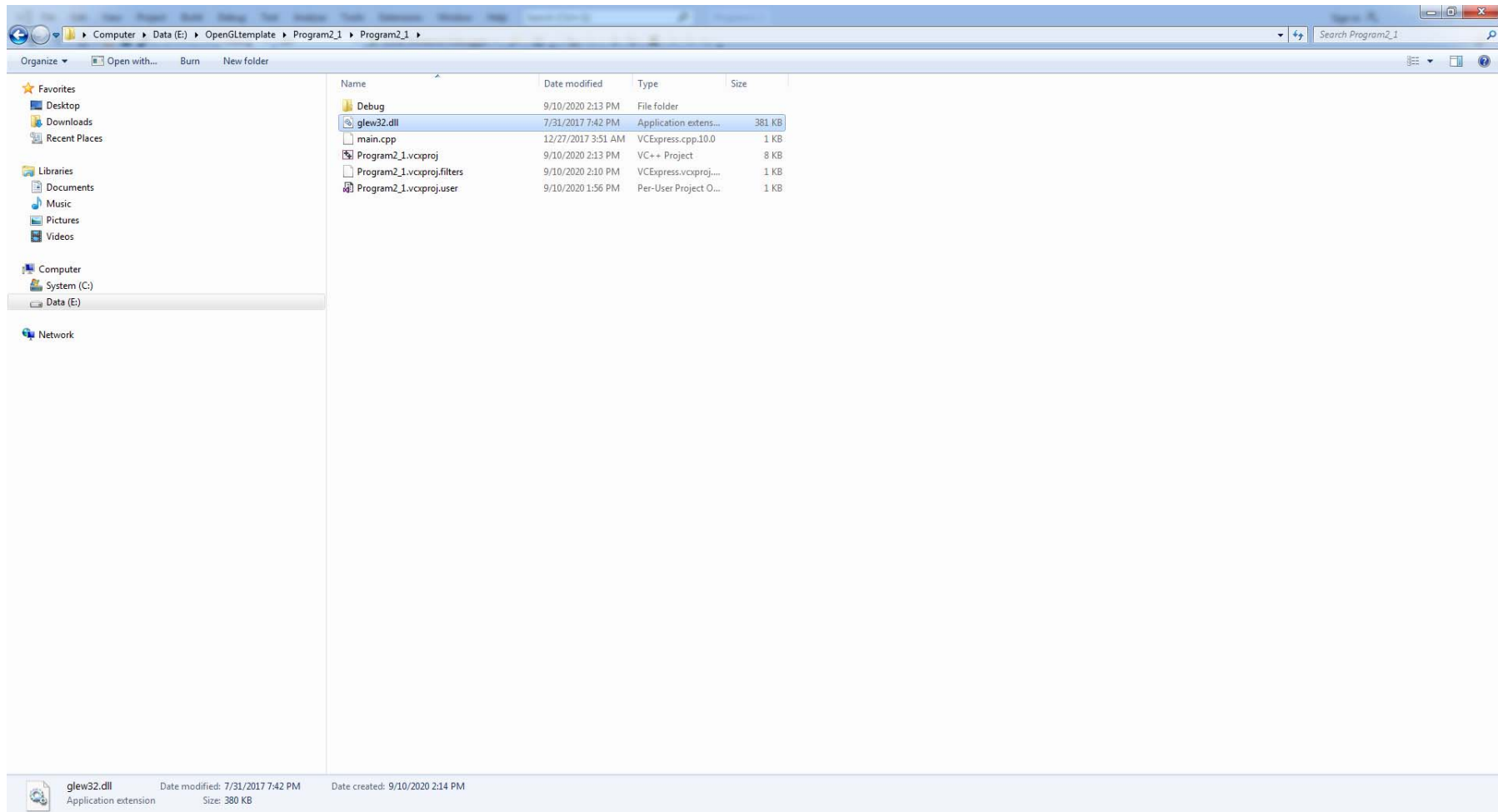
# Using Visual Studio 2019 for OpenGL



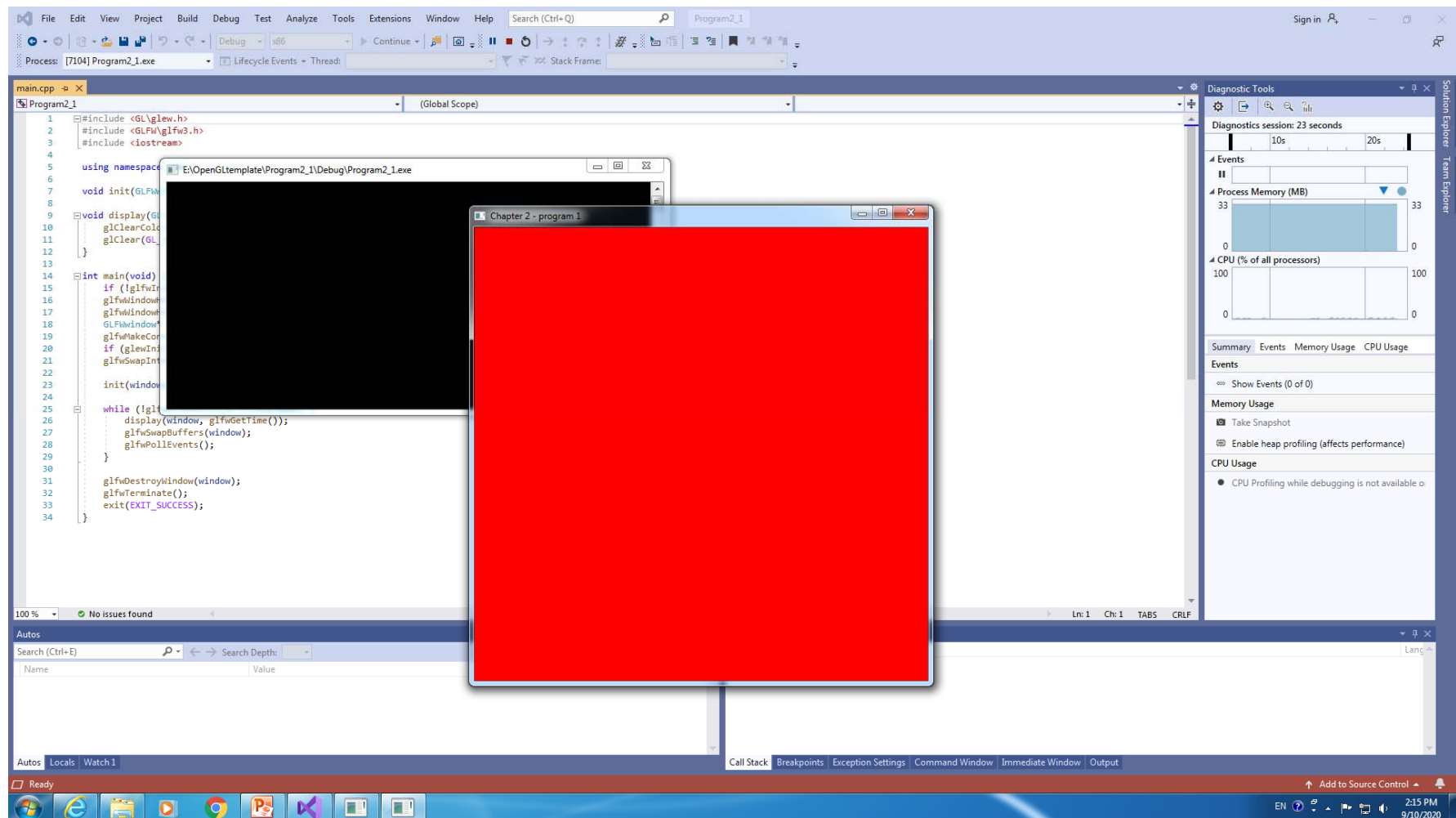
# Using Visual Studio 2019 for OpenGL



# Using Visual Studio 2019 for OpenGL



# Using Visual Studio 2019 for OpenGL



## OpenGL command formats

### glVertex3fv(...)

*Number of  
components*

2 - (x, y)  
3 - (x, y, z)  
4 - (x, y, z, w)

*Data Type*

b - byte  
ub - unsigned byte  
s - short  
us - unsigned short  
i - int  
ui - unsigned int  
f - float  
d - double

*Vector*

omit "v" for  
scalar form  
glVertex3f(x, y, z)

## OpenGL command formats

Suffix	Data Type	Typical Corresponding C-Language Type	OpenGL Type Definition
b	8-bit integer	signed char	GLbyte
s	16-bit integer	short	GLshort
i	32-bit integer	int or long	GLint, GLsizei
f	32-bit floating-point	float	GLfloat, GLclampf
d	64-bit floating-point	double	GLdouble, GLclampd
ub	8-bit unsigned integer	unsigned char	GLubyte, GLboolean
us	16-bit unsigned integer	unsigned short	GLushort
ui	32-bit unsigned integer	unsigned int or unsigned long	GLuint, GLenum, GLbitfield



## OpenGL color models

- ⌘ On a color computer screen, the hardware causes each pixel on the screen to emit different amounts of red, green, and blue light. These are called the **R**, **G**, **B** values. They're often packed together (sometimes with a fourth value, called **alpha**, or **A**), and the packed value is called the **RGB** (or **RGBA**) value.
- ⌘ In either color-index or RGBA mode, a certain amount of color data is stored at each pixel. This amount is determined by the number of bitplanes in the framebuffer. A **bitplane** contains 1 bit of data for each pixel.

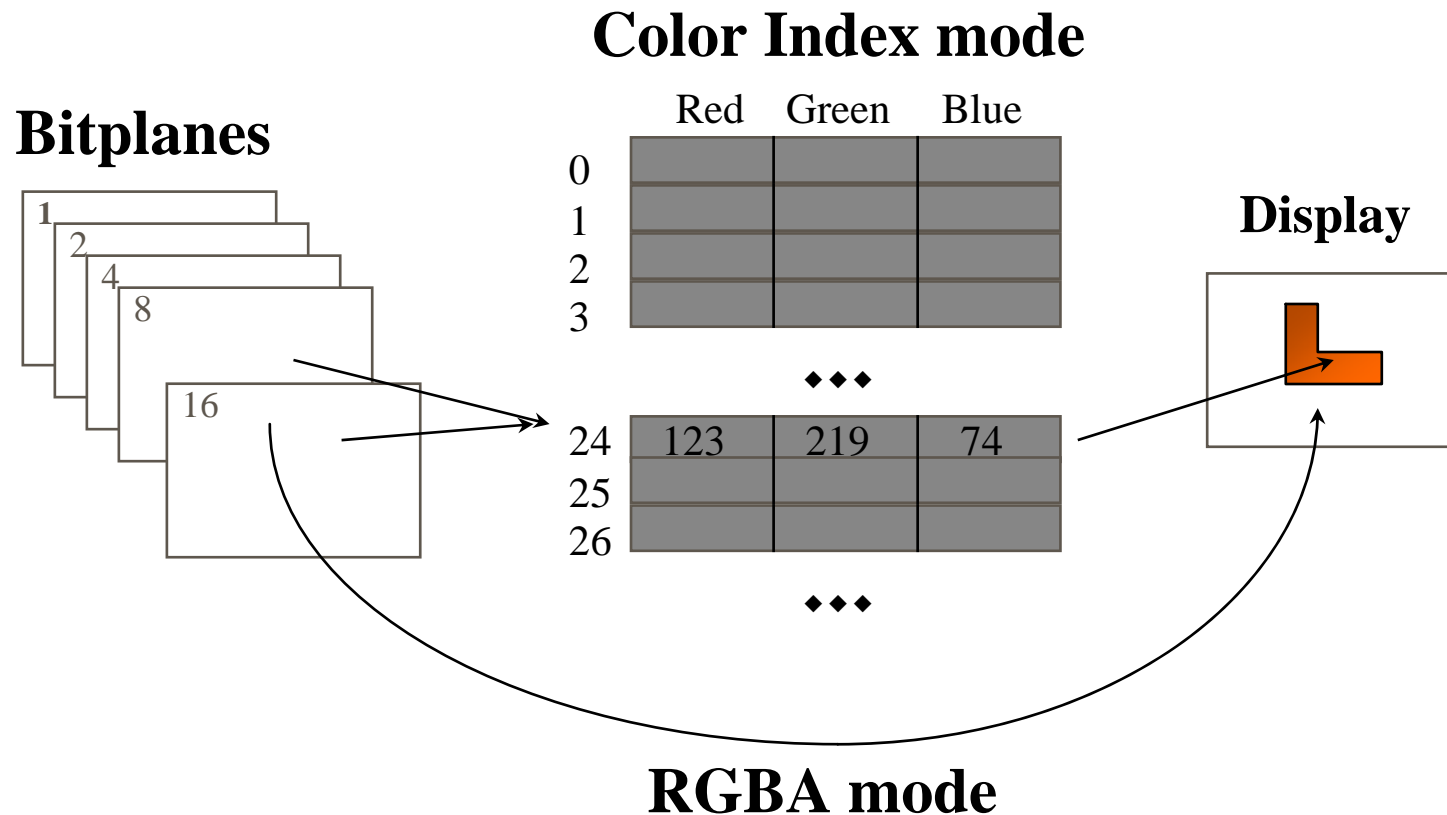
## OpenGL color models



- ⌘ The color information at each pixel can be stored either in RGBA mode, in which the R, G, B, and possibly A values are kept for each pixel, or in color-index mode, in which a single number (called the **color index**) is stored for each pixel. Each color index indicates an entry in a table that defines a particular set of R, G, B values. Such a table is called a **color map**.

## RGBA versus color-index mode

⌘ A **framebuffer** is a stack of bitplanes.



## OpenGL color models



- ⌘ On any graphics system, each pixel has the same amount of memory for storing its color, and all the memory for all the pixels is called the **color buffer**.
- ⌘ Color-index mode is not the common one used at present and also requires more detailed interaction with the windowing system than does RGBA mode. Hence, we shall always use RGB or RGBA color.
- ⌘ The R, G, and B values can range from 0.0 (none) to 1.0 (full intensity).

## Program 2.1

### First C++/OpenGL application



```
#include <GL\glew.h>
#include <GLFW\glfw3.h>
#include <iostream>
using namespace std;
void init(GLFWwindow* window) { }
void display(GLFWwindow* window, double currentTime) {
    glClearColor(1.0, 0.0, 0.0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT);
}
```

## Program 2.1

### First C++/OpenGL application

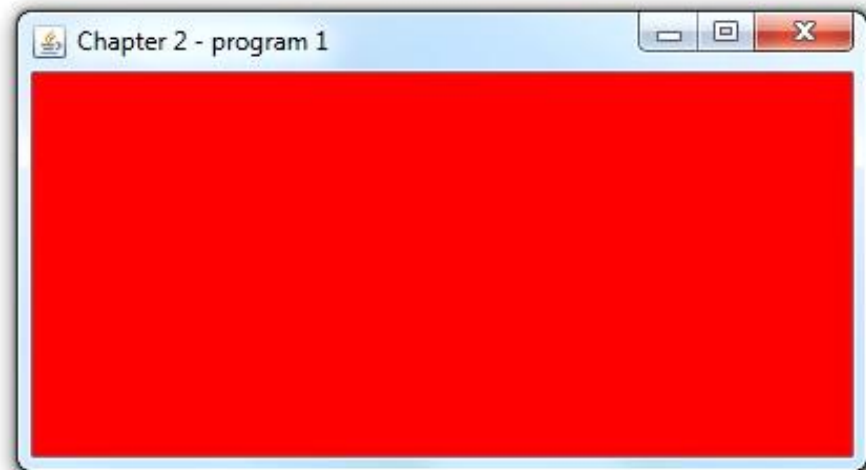


```
int main(void) {  
    if (!glfwInit()) { exit(EXIT_FAILURE); }  
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);  
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);  
    GLFWwindow* window = glfwCreateWindow(600, 600, "Chapter2 –  
                                     program1", NULL, NULL);  
    glfwMakeContextCurrent(window);  
    if (glewInit() != GLEW_OK) { exit(EXIT_FAILURE); }  
    glfwSwapInterval(1);  
}
```

## Program 2.1

### First C++/OpenGL application

```
init(window);  
while (!glfwWindowShouldClose(window)) {  
    display(window, glfwGetTime());  
    glfwSwapBuffers(window);  
    glfwPollEvents();  
}  
glfwDestroyWindow(window);  
glfwTerminate();  
exit(EXIT_SUCCESS);  
}
```



## Program 2.1

### First C++/OpenGL application



⌘ The `main()` function in **Program 2.1** is the same one that we will use throughout this course.

Among the significant operations in `main()` are:

- ☑ Initializes the GLFW library;
- ☑ Instantiates a `GLFWWindow`;
- ☑ Initializes the GLEW library
- ☑ Calls the function “`init()`” once;
- ☑ Calls the function “`display()`” repeatedly.



## Program 2.2

### Shaders, Drawing a POINT



*(.....#includes are the same as before)*

```
#define numVAOs 1
```

```
GLuint renderingProgram;
```

```
GLuint vao[numVAOs];
```

```
GLuint createShaderProgram() {
```

```
    const char *vshaderSource =
```

```
        "#version 430  \n"
```

```
        "void main(void) \n"
```

```
        "{ gl_Position = vec4(0.0, 0.0, 0.0, 1.0); }";
```

```
    const char *fshaderSource =
```

```
        "#version 430  \n"
```

```
        "out vec4 color; \n"
```

```
        "void main(void) \n"
```

```
        "{ color = vec4(0.0, 0.0, 1.0, 1.0); }";
```

## Program 2.2

### Shaders, Drawing a POINT



```
GLuint vShader = glCreateShader(GL_VERTEX_SHADER);
GLuint fShader = glCreateShader(GL_FRAGMENT_SHADER);

glShaderSource(vShader, 1, &vshaderSource, NULL);
glShaderSource(fShader, 1, &fshaderSource, NULL);
glCompileShader(vShader);
glCompileShader(fShader);

GLuint vfProgram = glCreateProgram();
glAttachShader(vfProgram, vShader);
glAttachShader(vfProgram, fShader);
glLinkProgram(vfProgram);

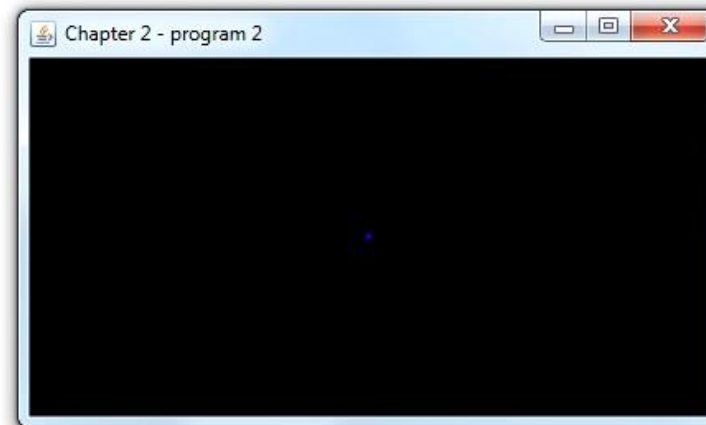
return vfProgram;
}
```

## Program 2.2

### Shaders, Drawing a POINT

```
void init(GLFWwindow* window) {  
    renderingProgram = createShaderProgram();  
    glGenVertexArrays(numVAOs, vao);  
    glBindVertexArray(vao[0]);  
}  
  
void display(GLFWwindow* window, double currentTime) {  
    glUseProgram(renderingProgram);  
    glDrawArrays(GL_POINTS, 0, 1);  
}
```

*...main() is the same as before*

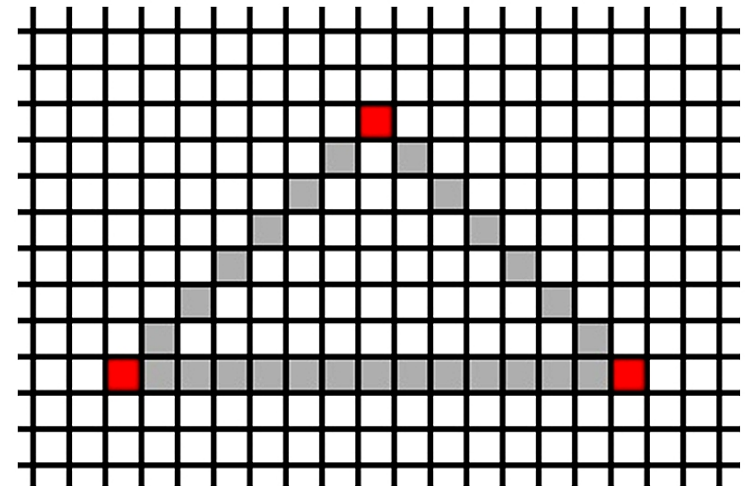


## Rasterization

⌘ Rasterization starts by interpolating, pairwise, between the three vertices of the triangle. For now it is sufficient to consider simple linear interpolation as shown in the figure. The original three vertices are shown in red.

⌘ If rasterization were to stop here, the resulting image would appear as wireframe. There is an option in OpenGL, by adding the following command in the `display()` function, before the call to `glDrawArrays()`;

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
```



## Fragment shader

⌘ Fragment shader offers us many ways to calculate colors. One simple example would be to base the output color of a pixel on its location. There is a variable for accessing the coordinates of an incoming fragment, called `gl_FragCoord`.

```
#version 430
```

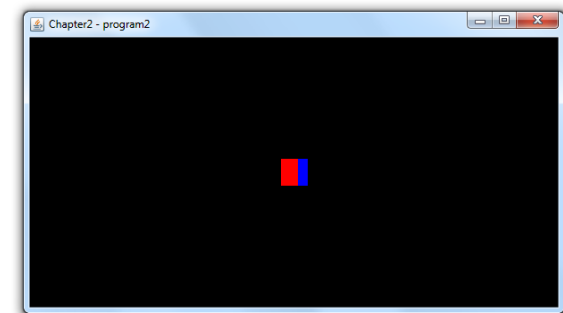
```
out vec4 color;
```

```
void main(void)
```

```
{ if (gl_FragCoord.x < 200) color = vec4(1.0, 0.0, 0.0, 1.0);
```

```
  else color = vec4(0.0, 0.0, 1.0, 1.0);
```

```
}
```



## Detecting OpenGL and GLSL errors

- ⌘ The workflow for compiling and running GLSL code differs from standard coding, in that GLSL compilation happens at C++ runtime.
- ⌘ Another complication is that GLSL code doesn't run on the CPU (it runs on the GPU), so the operating system cannot always catch OpenGL runtime errors. This makes debugging difficult.
- ⌘ There are three utilities for catching and displaying GLSL errors
  - ⏏ `checkOpenGLError` – checks the OpenGL error flag for the occurrence of an OpenGL error
  - ⏏ `printShaderLog` – displays the contents of OpenGL's log when GLSL compilation failed
  - ⏏ `printProgramLog` – displays the contents of OpenGL's log when GLSL linking failed

## Example of checking for OpenGL errors

```
GLuint createShaderProgram() {  
    GLint vertCompiled;  
    GLint fragCompiled;  
    GLint linked;  
  
    ...  
  
    // catch errors while compiling shaders  
    glCompileShader(vShader);  
    checkOpenGLError();  
    glGetShaderiv(vShader, GL_COMPILE_STATUS, &vertCompiled);  
    if (vertCompiled != 1) {  
        cout << "vertex compilation failed" << endl;  
        printShaderLog(vShader);  
    }  
}
```

## Example of checking for OpenGL errors

```
// catch errors while linking shaders
glAttachShader(vfProgram, vShader);
glAttachShader(vfProgram, fShader);
glLinkProgram(vfProgram);
checkOpenGLError();
glGetProgramiv(vfProgram, GL_LINK_STATUS, &linked);
if (linked != 1) {
    cont << "linking failed" << endl;
    printProgramLog(vfProgram);
}
return vfProgram;
}
```



## Detecting OpenGL and GLSL errors

- ⌘ A common result of shader runtime errors is for the output screen to be completely blank, essentially with no output at all. This can happen even if the error is very small typo in a shader, yet it can be difficult to tell at which stage of the pipeline the error occurred.
- ⌘ One useful trick in such cases is to temporarily replace the fragment shader with the one in **Program 2.2**. Recall that in that example, the fragment shader simply output a solid blue.
- ⌘ If the subsequent output is of the correct geometric form (but solid blue), the vertex shader is probably correct, and there is an error in the original fragment shader. If the output is still a blank screen, the error is more likely earlier in the pipeline, such as in the vertex shader.

## Reading GLSL source code from files

- ⌘ So far, our GLSL shader code has been stored inline in strings. As our programs grow in complexity, this will become impractical.
- ⌘ For practicality, code to read shaders is provided in `readShaderSource()`, shown in **Program 2.4**. It reads the shader text file and returns an array of strings, where each string is one line of text from the file. It then determines the size of that array based on how many lines were read in.
- ⌘ Note that here, `createShaderProgram()` replaces the version from **Program 2.2**.
- ⌘ In this example, the vertex and fragment shader code is now placed in the text files “`vertShader.glsl`” and “`fragShder.glsl`” respectively.

## Program 2.4

### Reading GLSL source from files

- ⌘ Add `readShaderSource()` and modify the `createShaderProgram()` in **Program 2.2** as follows.

```
GLuint createShaderProgram() {  
    (..... as before plus...)  
    string vertShaderStr = readShaderSource("vertShader.glsl");  
    string fragShaderStr = readShaderSource("fragShader.glsl");  
    const char *vertShaderSrc = vertShaderStr.c_str();  
    const char *fragShaderSrc = fragShaderStr.c_str();  
    glShaderSource(vShader, 1, &vertShaderSrc, NULL);  
    glShaderSource(fShader, 1, &fragShaderSrc, NULL);  
    (...etc., building rendering program as before)  
}
```

## Building objects from vertices



- ⌘ Ultimately, we would like to draw objects that are constructed of many vertices. Large parts of this course will be devoted to this topic.
- ⌘ Now we will define three vertices and use them to draw a triangle. We can do this by making two small changes to **Program 2.2/2.4**:
  - ☒ Modify the vertex shader so that three different vertices are output to the subsequent stages of the pipeline
  - ☒ Modify the `glDrawArrays()` call to specify that we are using three vertices
- ⌘ In the C++/OpenGL application we specify `GL_TRIANGLES` (rather than `GL_POINTS`), and also specify that there are three vertices sent through the pipeline.

## Program 2.5 Drawing a triangle

### ⌘ Vertex shader

```
#version 430
```

```
void main(void)
```

```
{
```

```
    if (gl_VertexID == 0) gl_Position = vec4(0.25, -0.25, 0.0, 1.0);
```

```
    else if (gl_VertexID == 1) gl_Position = vec4(-0.25, -0.25, 0.0, 1.0);
```

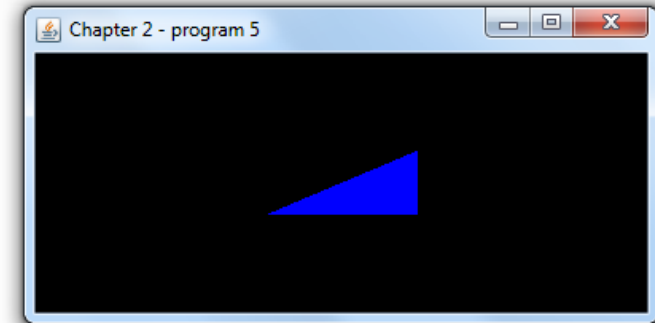
```
    else gl_Position = vec4(0.25, 0.25, 0.0, 1.0);
```

```
}
```

### ⌘ C++/OpenGL application – in display()

```
...
```

```
glDrawArrays(GL_TRIANGLES, 0, 3);
```



## Animating a scene



- ⌘ Many of the techniques in this course can be animated. This is when things in the scene are moving or changing, and the scene is rendered repeatedly to reflect these changes in real time.
- ⌘ Recall from **Program 2.1** or **Program 2.2** that we have constructed our `main()` to make a single call to `init()`, and then to call `display()` repeatedly. Thus, the loop in the `main()` was causing it to be drawn over and over again.
- ⌘ For this reason, our `main()` is already structured to support animation. We simply design our `display()` function to alter what it draws over time. Each rendering of our scene is then called a *frame*, and the frequency of the calls to `display()` is the *frame rate*.

## Program 2.6 Simple animation example

⌘ In C++/OpenGL application

// same #includes and declarations as before, plus the following:

**float x = 0.0f;** *// location of triangle on x axis*

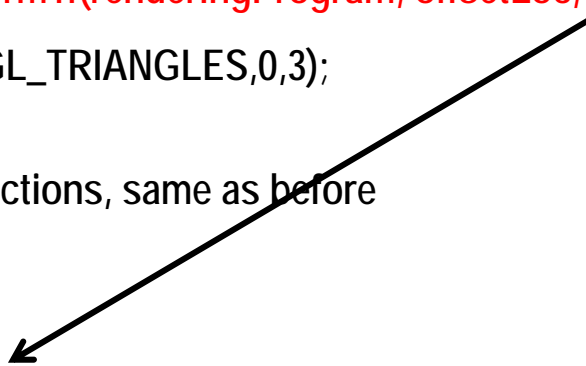
**float inc = 0.01f;** *// offset for moving the triangle*

```
void display(GLFWwindow* window, double currentTime) {  
    glClear(GL_DEPTH_BUFFER_BIT);  
    glClearColor(0.0, 0.0, 0.0, 1.0);  
    glClear(GL_COLOR_BUFFER_BIT); // clear the background to black, each time  
    glUseProgram(renderingProgram);
```

*(continued)*

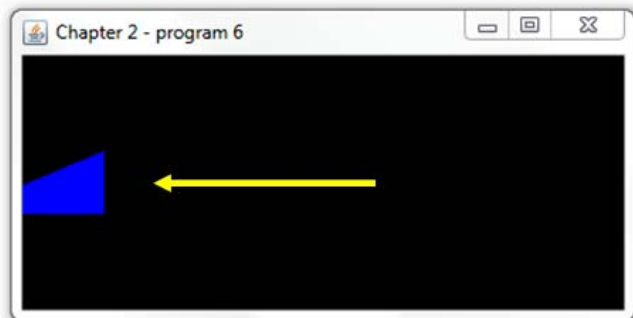
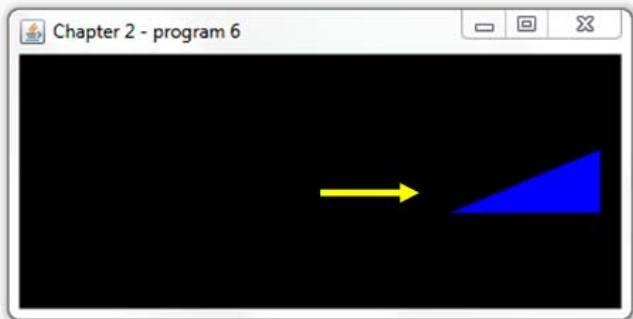
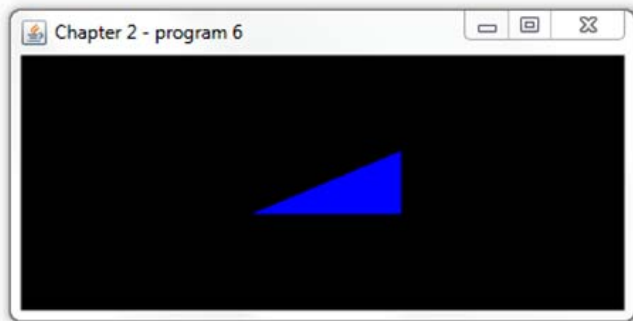
## Program 2.6 Simple animation example

```
x += inc;      // move the triangle along x axis
if (x > 1.0f) inc = -0.01f;      // switch to moving the triangle to the left
if (x < -1.0f) inc = 0.01f;      // switch to moving the triangle to the right
// retrieve pointer to "offset"
GLuint offsetLoc = glGetUniformLocation(renderingProgram, "offset");
glProgramUniform1f(renderingProgram, offsetLoc, x); // send value in "x" to "offset"
glDrawArrays(GL_TRIANGLES, 0, 3);
}
... // remaining functions, same as before
}
#version 430
uniform float offset;
void main(void)
{
    if (gl_VertexID == 0) gl_Position = vec4( 0.25 + offset, -0.25, 0.0, 1.0);
    else if (gl_VertexID == 1) gl_Position = vec4( -0.25 + offset, -0.25, 0.0, 1.0);
    else gl_Position = vec4( 0.25 + offset, 0.25, 0.0, 1.0);
}
```





## Program 2.6 Simple animation example



⌘ Note that in addition to adding code to animate the triangle, we have also added the following line at the beginning of the display function:

```
glClear(GL_DEPTH_BUFFER_BIT);
```

⌘ It is for *hidden surface removal*, which requires both a color buffer and depth buffer.

⌘ It will continue to appear in most of our applications.

## Organizing the C++ code files

- ⌘ So far, we have been placing all of the C++/OpenGL application code in a single file called “main.cpp”, and the GLSL shaders into files called “vertShader.glsl” and “fragShader.glsl”.
- ⌘ It is not a best practice when we have a lot of application code into main.cpp.
- ⌘ We define a file called “Utils.cpp” to contain the functions that we wish to reuse. These functions are: the error-detecting modules, the functions for reading in GLSL shader programs, etc. A “createShaderProgram()” function can be defined for each possible combination of pipeline shaders assembled in a given application, for examples, vertex and fragment shaders:

`Guint Utils::createShaderProgram(const char *vp, const char *fp)`

- ⌘ **Program 2.5** is an example of using modules.

## Exercises



- ⌘ Modify **Program 2.2** to add animation that causes the drawn point to grow and shrink, in a cycle. Hint: use the `glPointSize()` function, with a variable as the parameter.
- ⌘ Modify **Program 2.5** so that it draws an isosceles triangle (rather than the right triangle shown in Figure 2.15).

## Summary



- ⌘ Core and supporting libraries used in this course;
- ⌘ Using Visual Studio 2019 for OpenGL;
- ⌘ OpenGL command formats;
- ⌘ OpenGL color models;
- ⌘ First C++/OpenGL application;
- ⌘ Shaders for drawing a point;
- ⌘ Detecting OpenGL and GLSL errors;
- ⌘ Reading GLSL source code from files;
- ⌘ Building objects from vertices;
- ⌘ Animating scene;
- ⌘ Organizing the C++ code files.

**End of Lecture 3**