Tutorial 0

title: "Configuring Visual Studio forOpenGL" author: CS200 Computer Graphic I geometry: margin=1.5cm

Assignment 0 Configuring Visual Studio forOpenGL

Introduction

OpenGL is a specification that describes a low-level library or application programming interface for accessing features in graphics hardware. It contains about 500 functions to specify the objects, images, and operations required to produce three-dimensional computer graphics applications. The specification is managed by the not for profit, member funded consortium called Khronos. The key characteristics of OpenGL are:

- **Platform (graphics hardware and operating system) independence**: The OpenGL specification can be implemented completely in software (such as the Mesa 3D graphics library) or on a variety of graphics hardware systems. OpenGL is designed to complement but not duplicate the unique windowing systems implemented by each operating system. This means that OpenGL doesn't manage windows or handle input systems. Because every operating system implements its windowing system in different ways, each operating system has a different mechanism for applications to interface with OpenGL. This design for platform independence makes it easier for programmers to port OpenGL programs from one system to another. Much of the OpenGL code in programs can be used as is except for the portion relating to windowing and input systems.
- **Extension mechanism**: A major advantage of OpenGL is that it can continuously evolve. While the consortium manages major periodic updates to the specification, it is possible for graphics hardware manufacturers, operating system vendors, and publishers of graphics software (such as Autodesk, Adobe, Unity) to enhance and extend OpenGL's functionality using the extension mechanism. Extensions are features that are not part of the official OpenGL specification but are considered useful. The consortium integrates many of the extensions into the next major update of the specification.
- **Low-level specification**: Since OpenGL is a low-level graphics specification, it doesn't provide functionality for describing three-dimensional objects or operations for reading images described in popular file formats such as JPEG. Similarly, OpenGL doesn't provide functionality for implementing mathematics for graphics programming. The OpenGL shading language (GLSL) is designed for efficient vector and matrix processing and therefore incorporates built-in operators for implementing these operations.

OpenGL and Microsoft Windows

We'll be using computers running Microsoft Windows. OpenGL is supported on Windows, but because of certain decisions taken by Microsoft, two issues must be resolved for Windows applications to interface with OpenGL.

- Since OpenGL is designed to be independent of a computer's operating or windowing system, how do applications create and manage windows with specific types of buffers, color formats, and other characteristics as well as handle event-driven messages from input devices and menu controls?
- Graphics hardware vendors (such as Nvidia, AMD, Intel) implement the latest OpenGL version as well as extensions in the form of graphics drivers. How are OpenGL calls made by application programs executed by graphics hardware?

OpenGL Contexts and Windows Device Contexts

Windows has many methods of drawing into a window including Graphics Device Interface (GDI), GDI+, and Windows Presentation Foundation. GDI is common to all versions of Windows and is commonly used by OpenGL application programmers to render to windows. Every window has a data structure called ***device context*** which defines a set of graphic objects and their associated attributes, and the graphic modes that affect output. Graphic objects include a pen for line drawing, a brush for painting and filling, a bitmap for copying or scrolling parts of the screen, a palette for defining the set of available colors, a region for clipping and other operations, and a path for painting and drawing operations. An application never has direct access to the device context; instead, it operates on the structure indirectly by calling various functions.

OpenGL defines an internal data structure called a ***context*** to keep track of state settings and operations to be applied on input data. In Windows, an OpenGL context is referred to as a ***rendering context*** and it serves as a link between OpenGL and the windowing system. This means that for an application to render anything using OpenGL, it must first initialize an OpenGL rendering context, and before the rendering context can be created, a GDI device context is required, and to get the device context a window must first be created. Windows provides a specific set of features and functions labeled WGL (**W**indows **GL**) that provide linkage between rendering contexts and device contexts. Working backwards, we use GDI API to create a window and get access to the device context associated with this window. Next, the device context must be configured to the rendering needs of the OpenGL application. Specific details include whether the color buffers are single or double buffered, the depth of these color buffers, whether a depth buffer, stencil buffer, accumulation buffer is required, and the number of bits for each of these buffers. OpenGL on Windows uses the term pixel formats to encapsulate this rendering related information. After setting the pixel format of the device context associated with a window, the final step is to create an OpenGL rendering context which is a structure that allows OpenGL rendering to be applied on the device context. Once the application has a rendering context, it can enter the main loop and query the message queue for any mouse clicks or keyboard input messages.

Using OpenGL Extension Mechanism on Microsoft Windows

Since OpenGL is designed to be platform-independent, its specific implementation is tied to a particular combination of graphics driver and operating system. Graphics drivers are implementations by hardware vendors of the core OpenGL specification and extensions to manage graphics hardware. At runtime, drivers translate calls to OpenGL API and extensions into native machine language instructions for execution by the graphics hardware. To compile successfully, source files making calls to OpenGL functions must include the following header files:

header	description
`gl.h`	contains declarations of functions defined in OpenGL version 1.1. This file ships with Windows
`glcorearb.h`	contains declarations of functions defined in higher versions of OpenGL and extensions. This file is maintained by the consortium and the latest version is available here. Extensions are constantly augmented by vendors and graphics drivers may not implement all of the extensions declared in this file.
`wglext.h`	exposes extensions specific to Windows. This file is also maintained by the consortium and the latest version is available here.

Windows ships with a software implementation of OpenGL compatible with version 1.1 (circa 1997) comprising of header file `gl.h` , a dynamic-linked library (DLL) `opengl32.dll` and an import library `opengl32.lib` . As explained earlier, `gl.h` contains declarations of functions defined in OpenGL 1.1 specification. `opengl32.dll` is a dynamic linked library module containing the software implementation of OpenGL 1.1 that can be shared by multiple OpenGL applications being executed simultaneously. The import library `opengl32.lib` allows the linker to resolves references made to functions exported by `opengl32.dll` and it supplies the runtime system with information needed to load `opengl32.dll` from disk to memory and locate the OpenGL functions exported by the DLL when the application is loaded. To compile successfully, source files making calls to functions from OpenGL 1.1 include header file `gl.h` . Next, these object files are linked with import library `opengl32.lib` to create an executable. When the user program is loaded into memory for execution, the Windows runtime environment will load `opengl32.dl` l into memory. At runtime, calls made by the user program to OpenGL API functions will be executed by their (software) implementations in `opengl32.dll` . Microsoft has no plans to update `gl.h` , `opengl32.lib` , and `opengl32.dll` that are shipped with Windows. Because `opengl32.dll` is part of Windows, it cannot be altered by graphics drivers to add new features for higher OpenGL versions. Thus, it seems that programmers can only create OpenGL applications in Windows that conform to OpenGL 1.1. How can programmers build modern graphics applications that take advantage of the latest OpenGL versions, extensions, and graphics hardware?

To access functions from higher versions, Windows employs a trick. First, Windows permits graphics hardware vendors to implement the latest versions of OpenGL (along with extensions) in the form of an installable client driver (ICD). The ICD implements the entire OpenGL specification using a combination of software and the specific graphics hardware for which it is written. In addition, Windows allows the OpenGL runtime `opengl32.dll` (implementing OpenGL 1.1) to access the Windows registry and load the appropriate ICD (for example, your computer may have three different graphics hardware, one each from Intel, AMD, and Nvidia and therefore will have three different ICDs). So, how does the ICD expose the new functions to applications? Windows provides a specific function called wglGetProcAddress to return the address of an OpenGL function that is defined in higher OpenGL versions. So, at runtime, the OpenGL program can manually obtain pointers to OpenGL functions from higher OpenGL versions by calling **`wglGetProcAddress`**. The OpenGL program can then dereference these pointers and have the ICD execute the functions either in software or on the specific hardware for which the ICD has been implemented.

Writing Platform-Independent OpenGL Applications

We'd like to concentrate our discussions solely on computer graphics rather on the different mechanisms that each operating system uses to help applications interface with OpenGL. We'll use helper libraries to abstract away platform-specific drudgery required to interface our applications to OpenGL. This will come at the cost of flexibility and control but will greatly aid in quickly getting our OpenGL applications up and running.

- Since OpenGL is designed for compatibility across operating systems, it is relatively easy for programmers to port OpenGL programs from one operating system to another. Programmers can encapsulate the portion of code specific to a particular operating system and decouple it from the code specific to OpenGL. However, programmers do have to deal with the low-level operating system mechanisms to interface with OpenGL. Instead, it would be more convenient for computer graphics practitioners and students to abstract away operating system specific code. Such an abstraction would alleviate the need for programmers to learn specific details of one or more operating systems and instead concentrate on the implementation of graphics functionalities. A list of toolkits that implement an abstraction layer to create and manage windows, create OpenGL contexts, and deal with user input in a platform independent manner is listed here. This course will use a popular toolkit called GLFWto implement tutorials and assignments.

- Recall that since Windows ships with OpenGL 1.1, programmers must download header files glcorearb.h to obtain interfaces to higher versions of core OpenGL specification andextensions and wglext.h for Windows-specific extension interfaces. In addition, applications must call WGL function `wglGetProcAddress` to manually obtain pointers to functions defined in higher versions of OpenGL and extensions. There are many varying implementations of core OpenGL and extensions and both downloading the appropriate header files and getting access to these functions through function pointers is tedious work for application programmers. Thankfully, many librarieshave been implemented to abstract away the header files and function pointer inconveniences. We'll use one such platform-independent library called GLEWto provide the necessary header files and expose the functionality of core OpenGL and extensions.

- Recall that OpenGL doesn't provide functionality for implementing mathematics for graphics programming while GLSL contains built-in operators for efficient vector and matrix processing. In some cases, we'll use a C++ mathematics library designed for graphics programming called OpenGL Mathematics(GLM). This library was picked because it is a light-weight header-only library and more importantly because it is based on GLSL specifications so that anyone who knows GLSL can use GLM. In other cases, we'll implement a two- and three-dimensional math library from scratch in C++.

- Since OpenGL is a low-level graphics library, it doesn't provide higher level abstractions for loading images stored in file formats such as JPEG, BMP, PNG, and so on. In some cases, we'll use an image library implemented as a single-file C header file `stb_image.h` to load images from file, format image data to raw data, and store the raw data in memory. In other cases, we'll have to write code to manually parse image files using the file format's specification.

- Certain applications might require graphical user interfaces. Rather than building them from scratch, we use a C++ library called dear imgui.

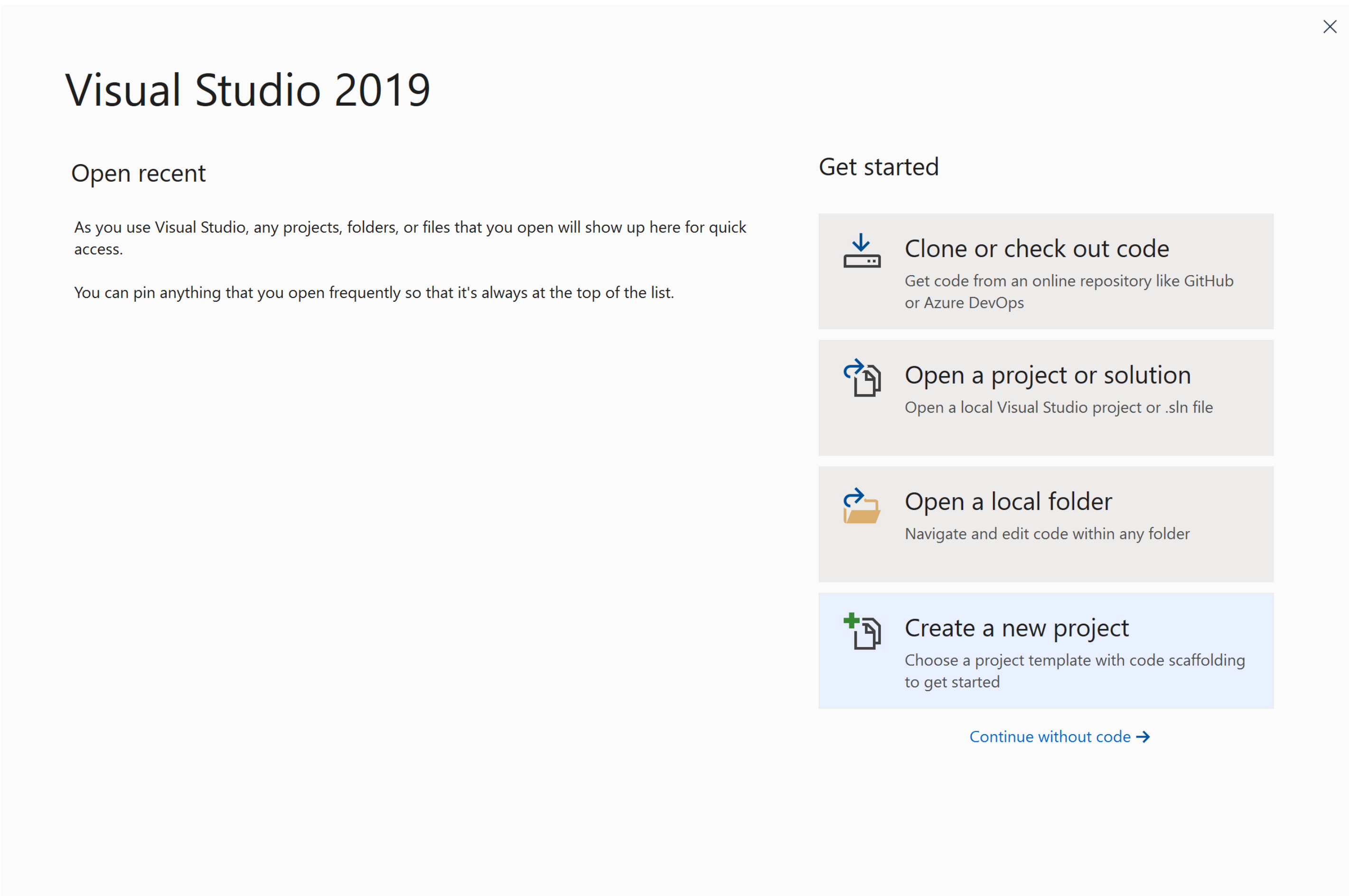
Visual Studio Project Settings

Because we're using many helper libraries, many properties such as compiler options, linker options, and custom build steps must be specified for each individual Visual Studio project. Manually setting these properties for every new project can quickly become repetitive, tedious, and error-prone. Instead, Visual Studio allows these properties to be saved in a property page. When a new project is created, the properties from the saved property page can be directly loaded into it. The following steps describe how to create and save a property sheet, attach this property sheet to a project, and finally build and execute an OpenGL application.

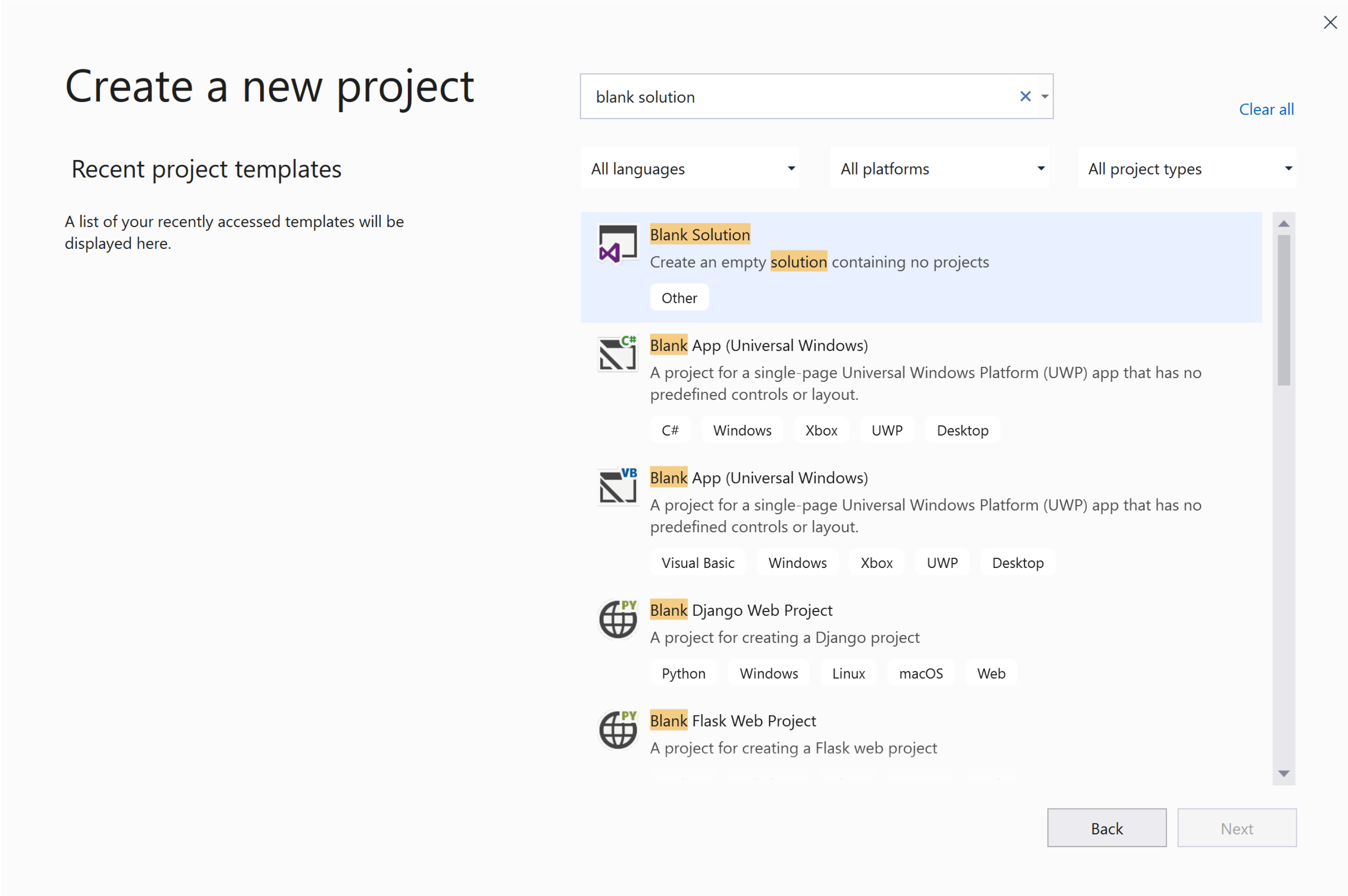
The newest version 4.6 of the OpenGL specification was released in 2017. To accommodate your personal machines for implementing tutorials and assignments, this course will be using OpenGL version 3.3 which was released in 2010. Most machines purchased in the past ten years should have graphics hardware compatible with version 3.3. However, it is possible that all though your graphics hardware is compatible, their graphics drivers have not been recently upgraded. To avoid compatibility issues, it is a good idea to update your graphics driver now before continuing with this tutorial. If you don't know how to update your graphics drivers (your machine may have more than one graphics hardware) or which OpenGL version your graphics driver is compatible with, download, install, and run this tool.

Visual Studio 2019: Creating Solutions and Projects

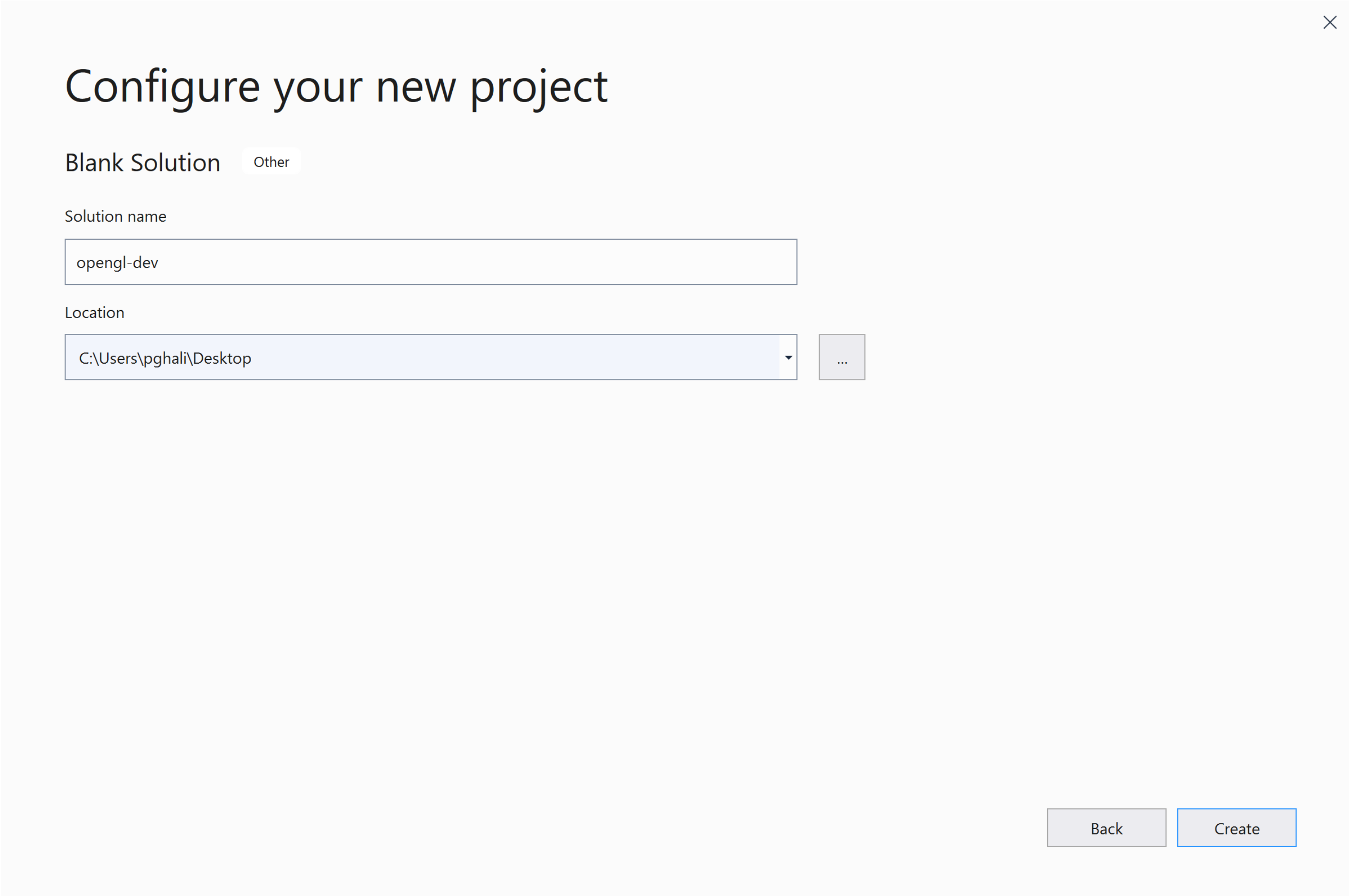
1. Create an (empty) Visual Studio solution as shown in the following screenshot:
   - Open Visual Studio 2019. On the start window, choose **Create a new project**.



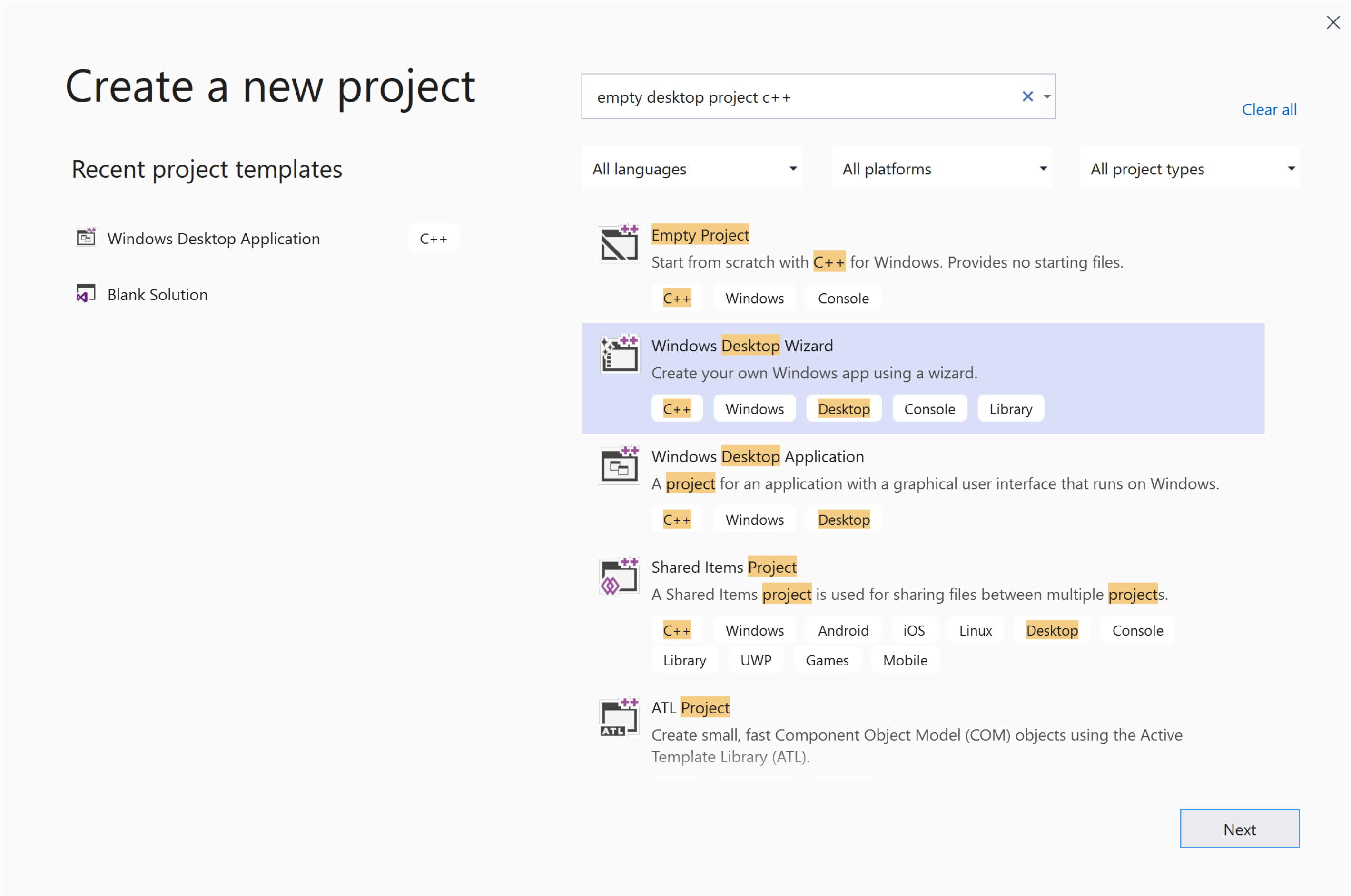
   - On the **Create a new project** page, enter **blank solution** into the search box, select the **Blank Solution** template, and then choose **Next**.



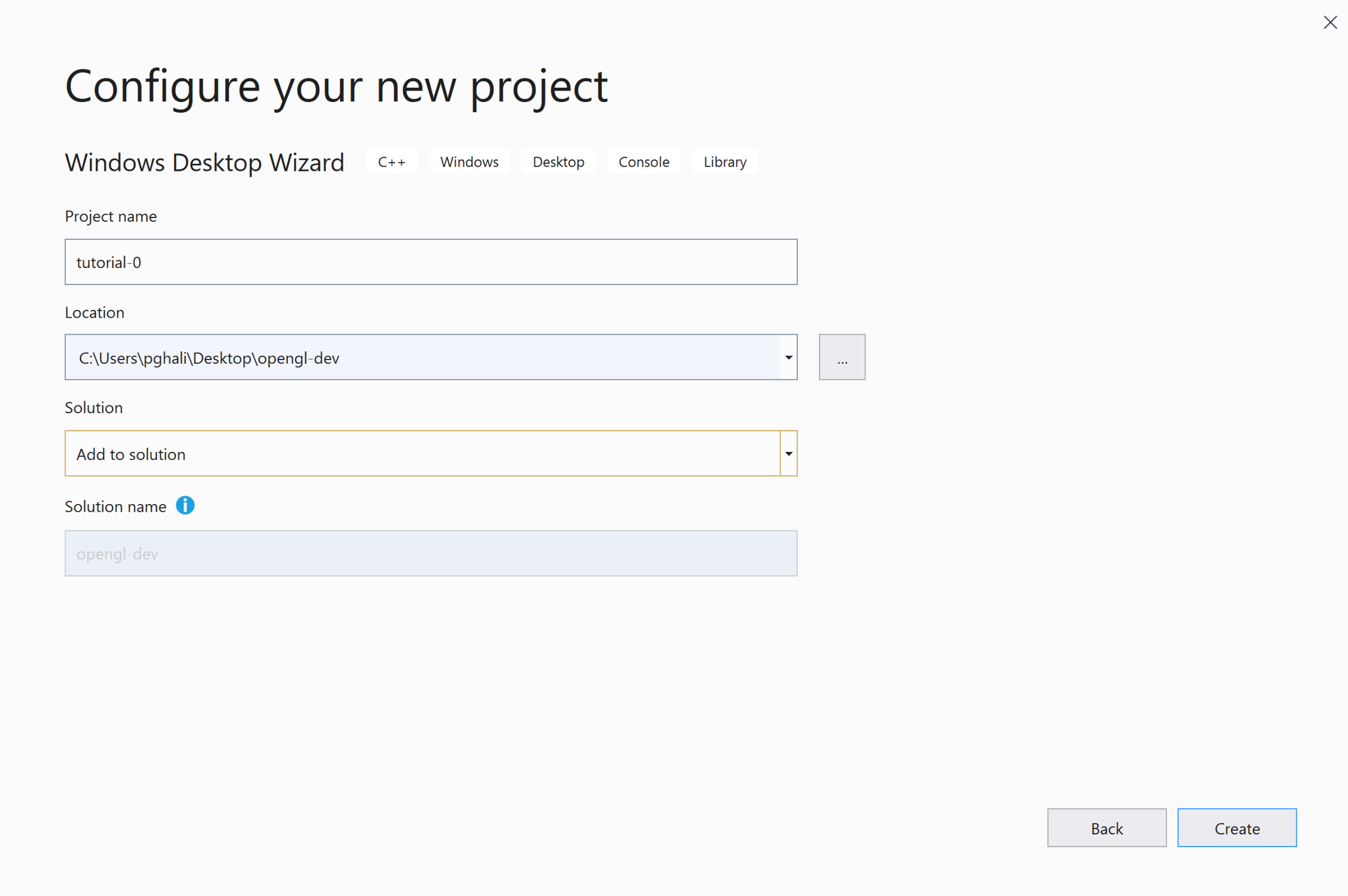
- As shown in the following image, provide **opengl-dev** as the solution's name and a path that specifies a convenient location on your computer, and then choose **Create** :



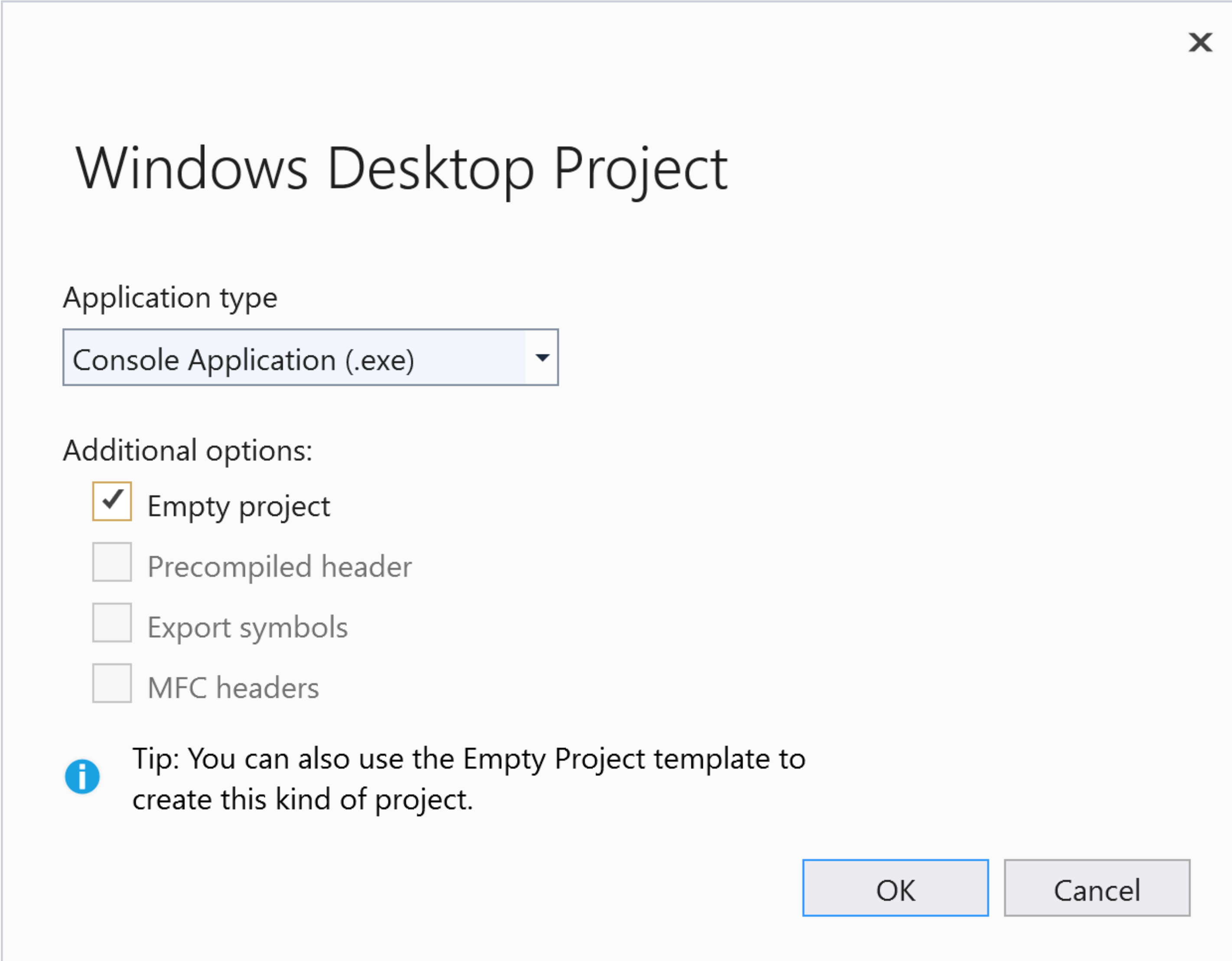
2. Now, add the first project to the solution. We'll start with an empty project and add the required items to the project.
   - From the main menu, choose **File** > **New** > **Project**. A dialog box opens that says **Create a new project**.
   - Enter the text **empty desktop project c++** into the search box at the top.



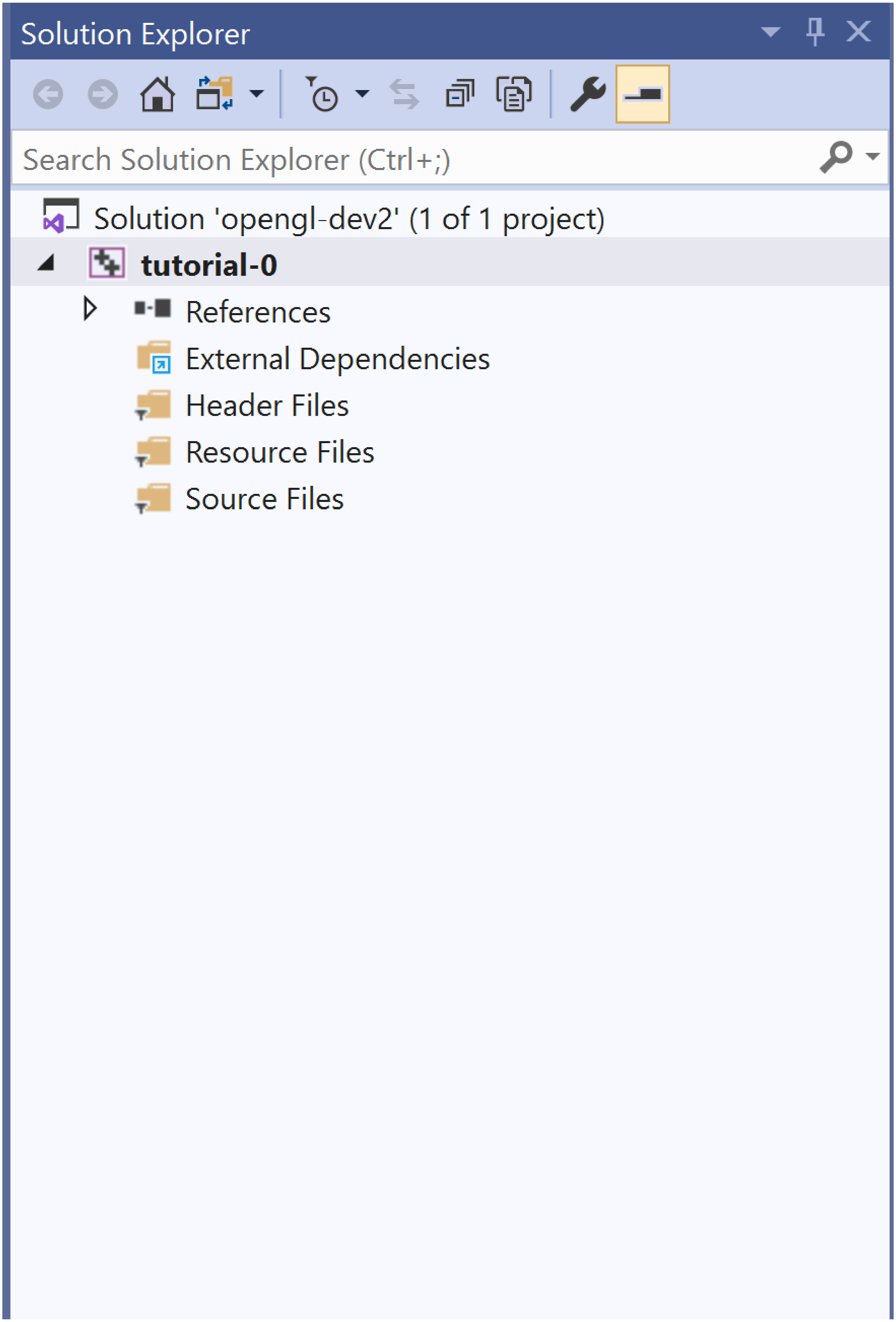
- Select the **Windows Desktop Wizard** template, and then choose **Next**.
- Name the project **tutorial-0** , toggle **Solution** to **Add to solution** , and then choose **Create**.

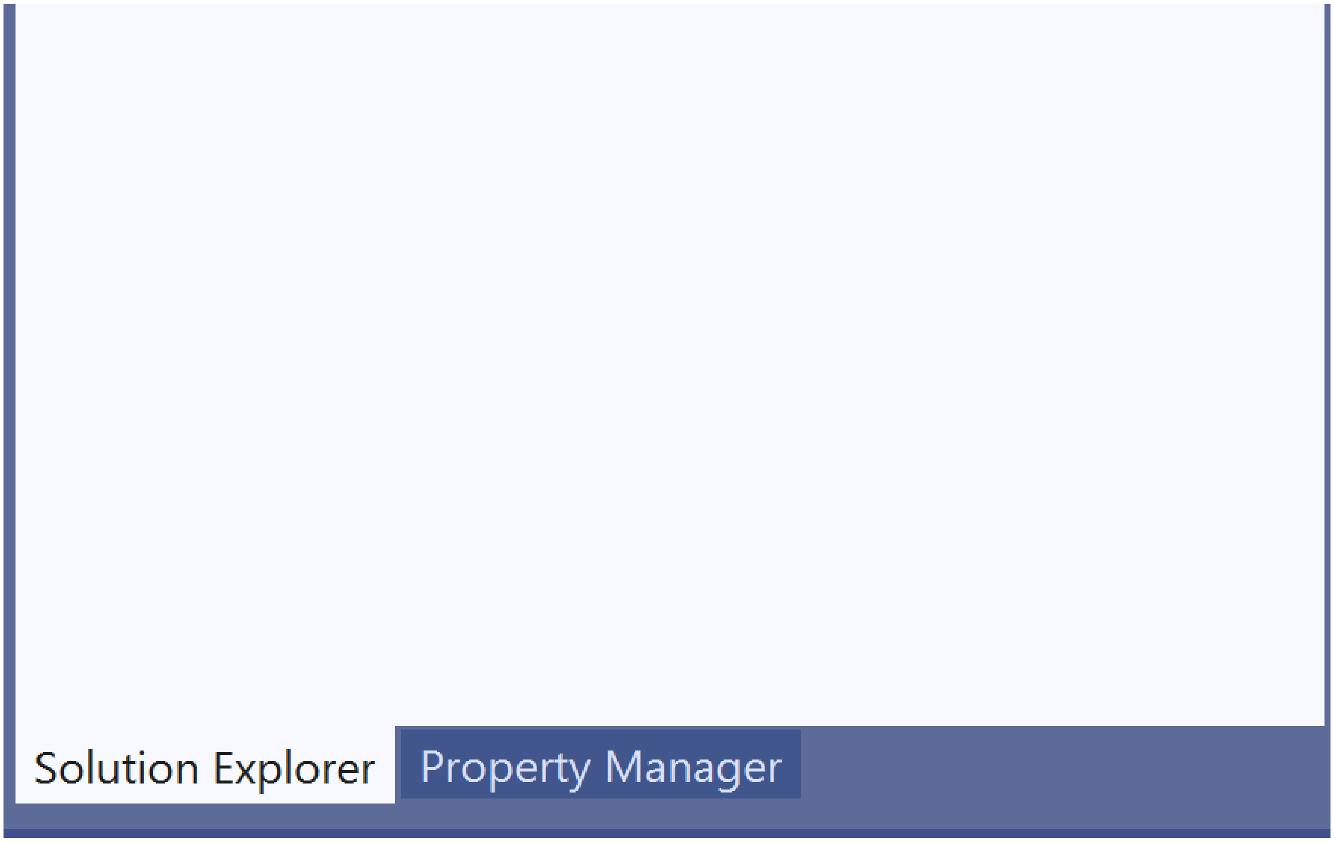


- A **Windows Desktop Project** dialog opens up. Choose **Application Type** as **Console**** Application (.exe) **, check the** Empty Project **box, and then choose** OK**.



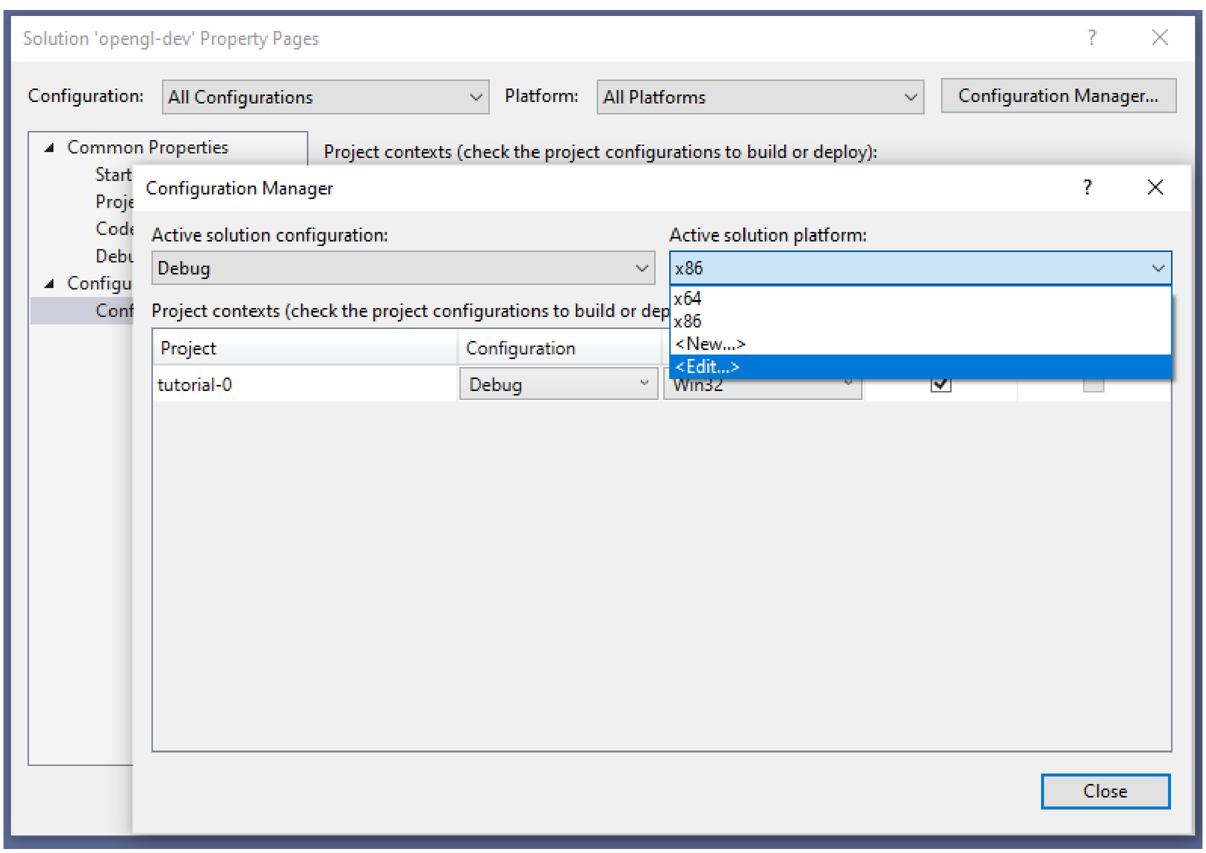
- Your **Solution Explorer** view window will now look like this:


Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'opengl-dev2' (1 of 1 project)
tutorial-0
References
External Dependencies
Header Files
Resource Files
Source Files

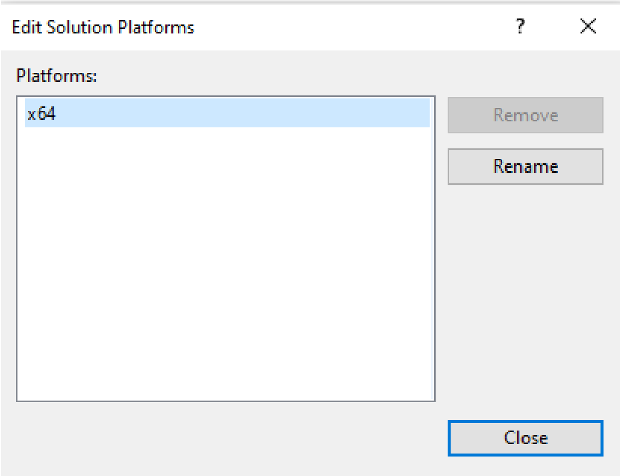


3. No we are going to remove **Win32/x86** configurations. Desktop PC's are no longer made with 32 bit CPU's. In this class we are focusing on learning about graphics and not supporting old platforms. By removing the 32 bit configurations we reduce the complexity of configuring our project.

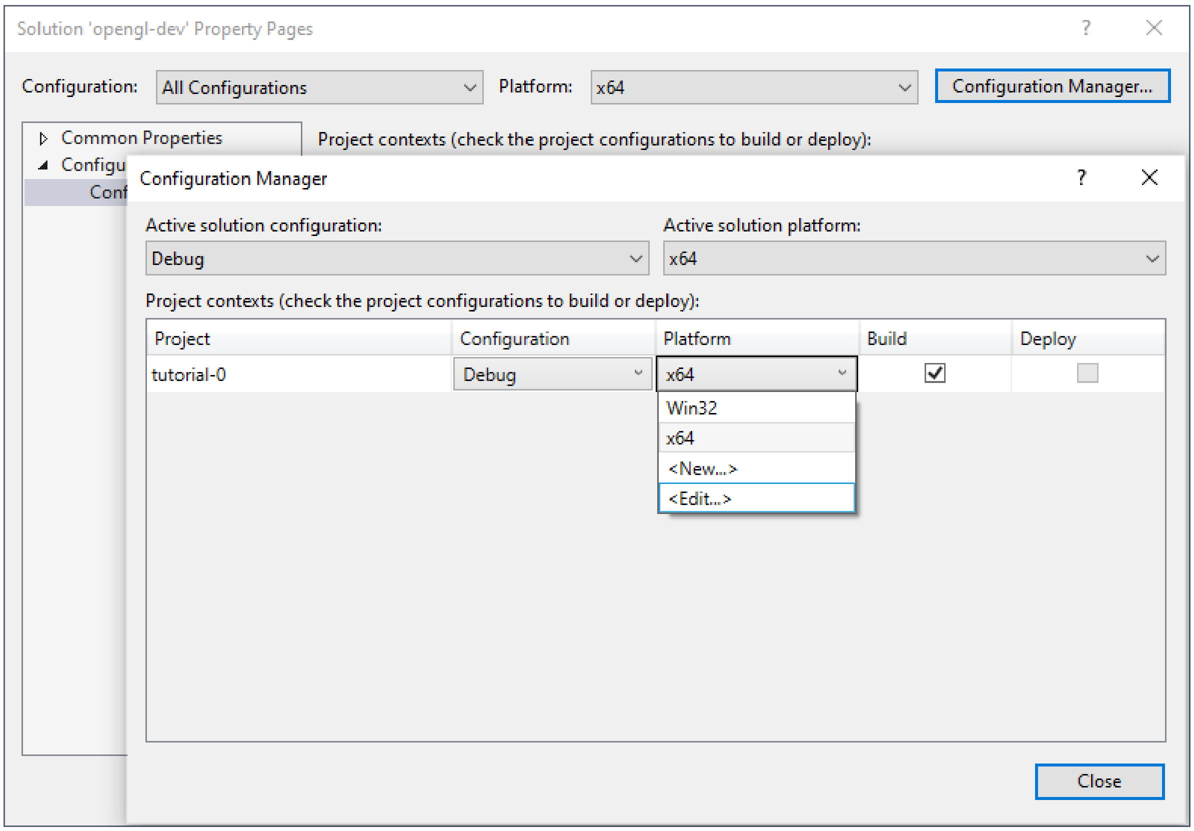
    - In the main menu, click **Build** > **Configuration Manager**. Click on the dropdown for **Active solution platforms** and select **Edit**



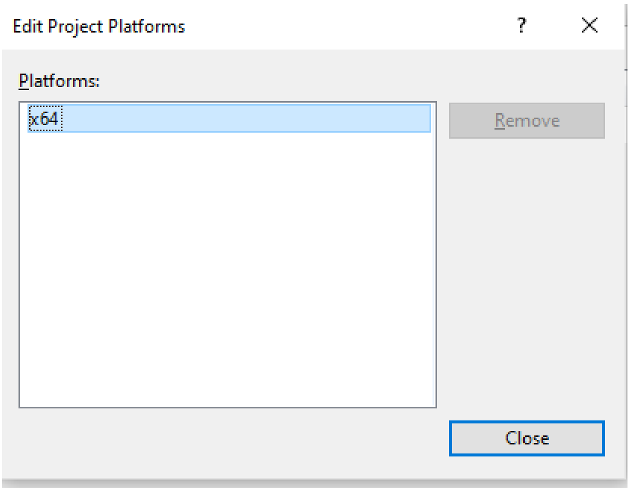
- Remove **x86** so that only x64 is left



- Click on the dropdown for the Project **Platform** and select **Edit**



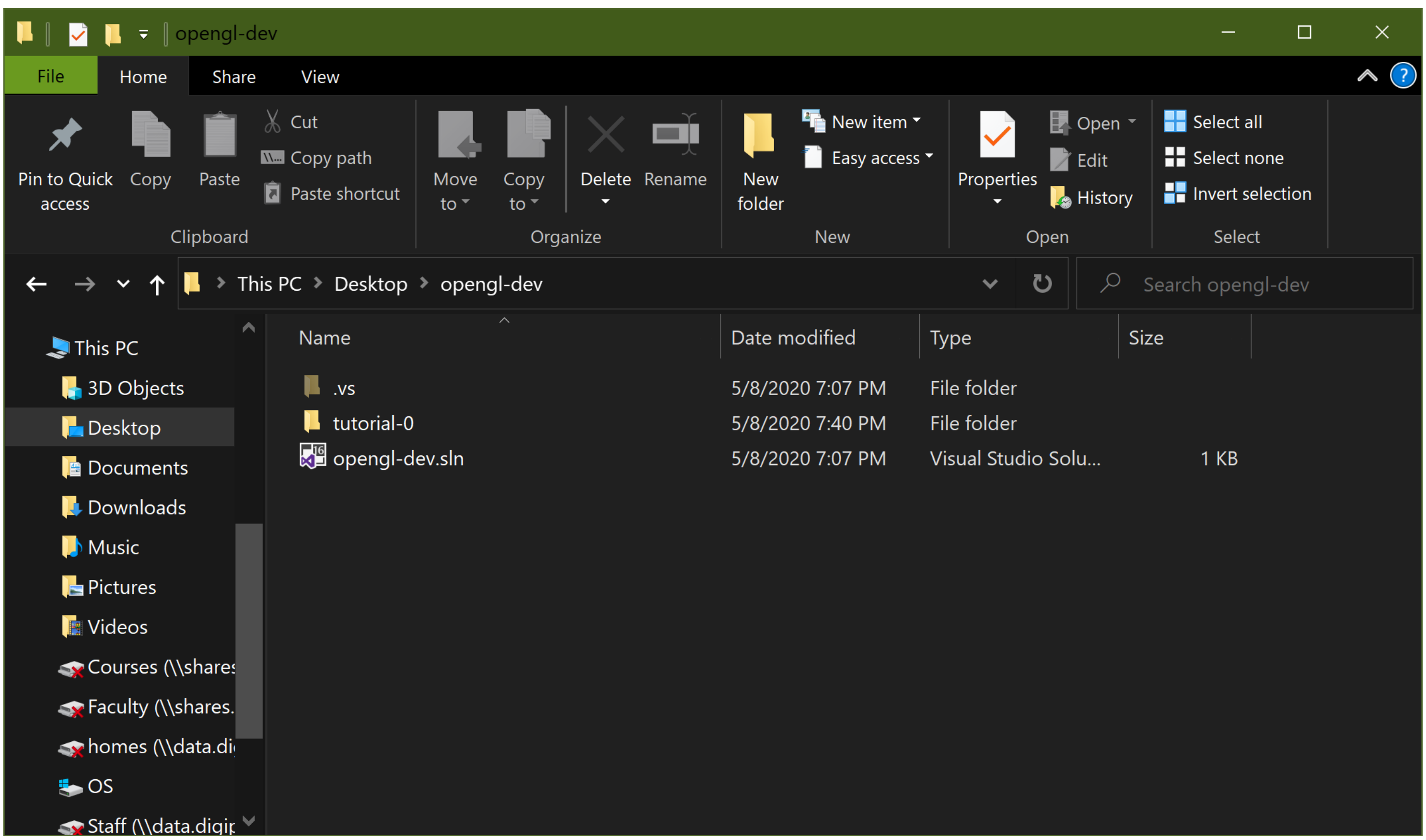
- Remove **Win32** so that only x64 is left



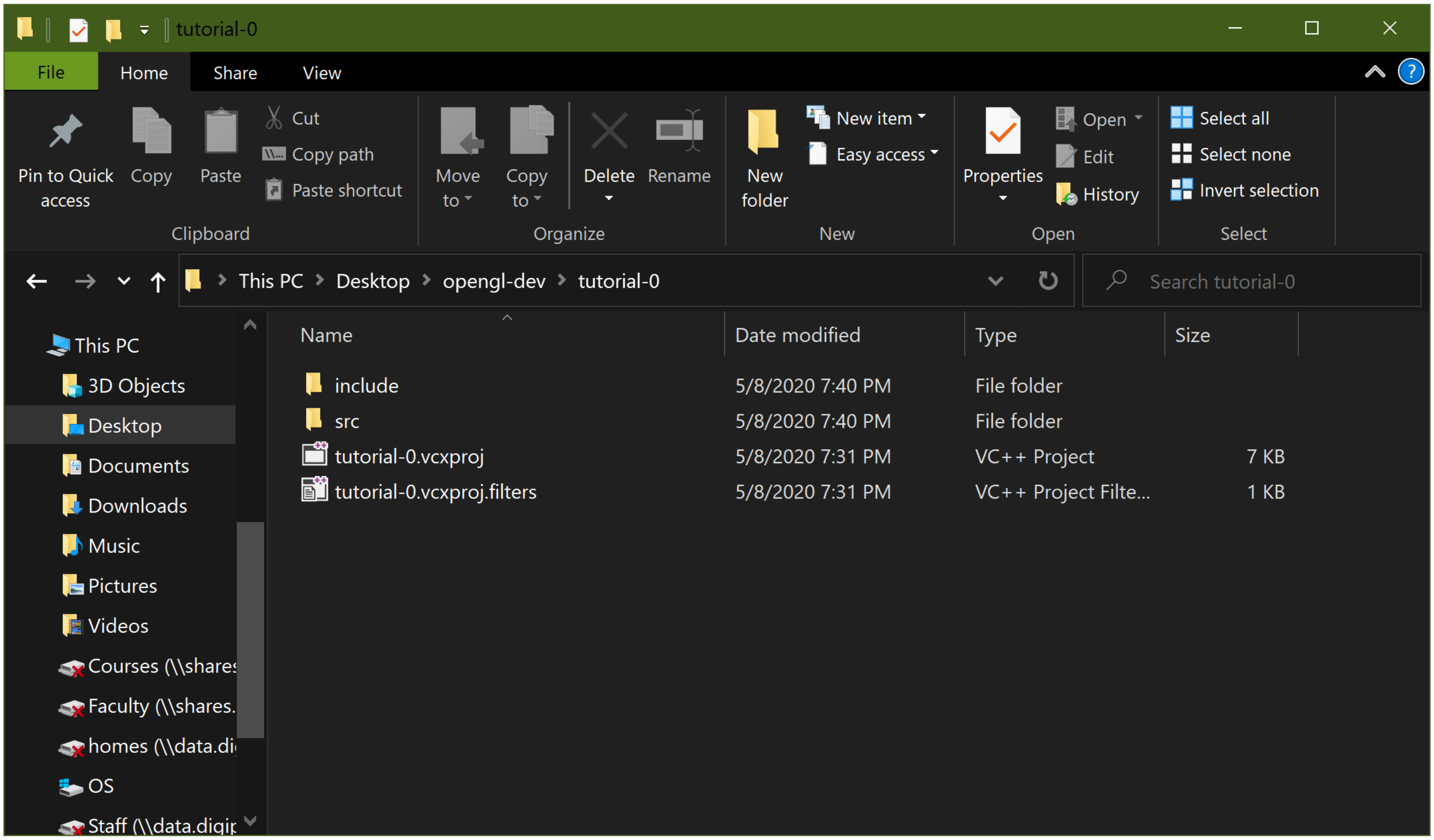
- Moving forward we now only need to worry about 64bit binaries

Adding source code and header files to project

If you've correctly followed the process of creating solution **opengl-dev** and adding project **tutorial-0** to the solution, then you should have a hierarchy consisting of a *solution* directory `opengl-dev` within which there is a *project* subdirectory `tutorial-0` . This hierarchy is important for submissions and grading. The picture directly below illustrates this hierarchy. Make sure to confirm that you have created a similar hierarchy. Otherwise, you've not correctly interpreted instructions for creating a solution and adding a project to that solution.



As part of the tutorial, starter code is provided as part of the tutorial in the directory labeled `./starter-code` . Source files `glapp.cpp` and `main.cpp` are located in directory `src` while the header file `glapp.h` is located in directory `include` . Copy directories `src` and `include` into directory `tutorial-0` . Retain this organizational structure of storing source files in `src` and header files in directory `include` in every new project that you create. The directory structure in directory `tutorial-0` is illustrated in the following picture:



This directory structure imposes the rule that source files of a project located in directory src can only include header files with a relative path `../include` , as in this code snippet from `main.cpp` :

```
#include "../include/glapp.h"
```

Add source files `glapp.cpp` and `main.cpp` to the **Source Files** filter of the current project `tutorial-0` and add header file `glapp.h` to the **Header Files** filter.

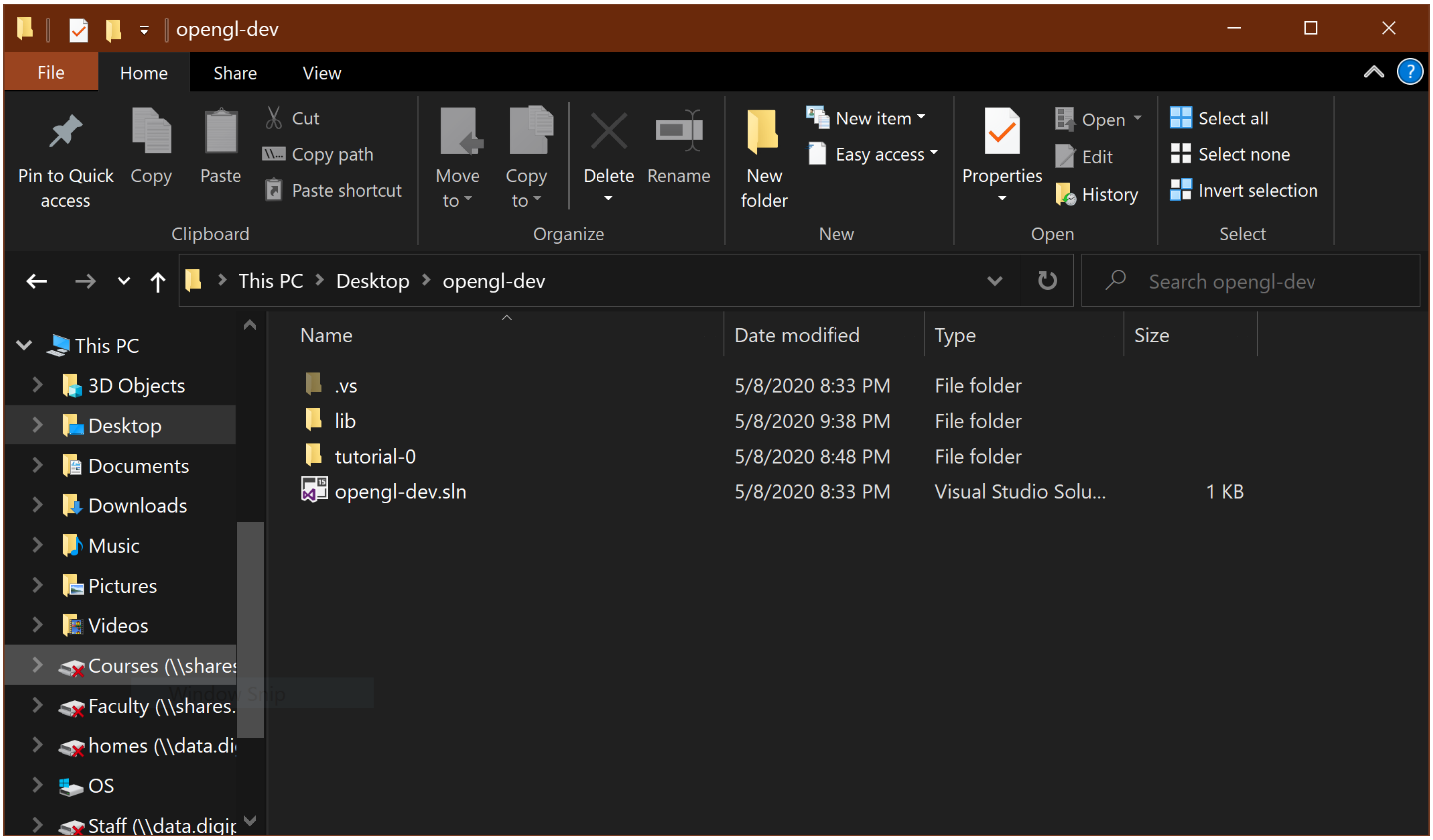
Compiling the source files will cause errors since the compiler is unable to locate header files such as `glew.h` , `glm.h` , or `glfw3.h` .

Visual Studio Property Pages

Because we're using many helper libraries, many properties such as compiler options, linker options, and custom build steps must be specified for each individual project. Manually setting these properties for every new project can quickly become repetitive, tedious, and error-prone. Instead, Visual Studio allows these properties to be saved in a property page. Property pages allow users to specify and customize configuration properties and parameters to individual projects within a solution. They collect a project's configuration properties and parameters in a single file. When a new project is created, the properties from the saved property page can be directly loaded into it rather than tediously updating parameters one at a time.

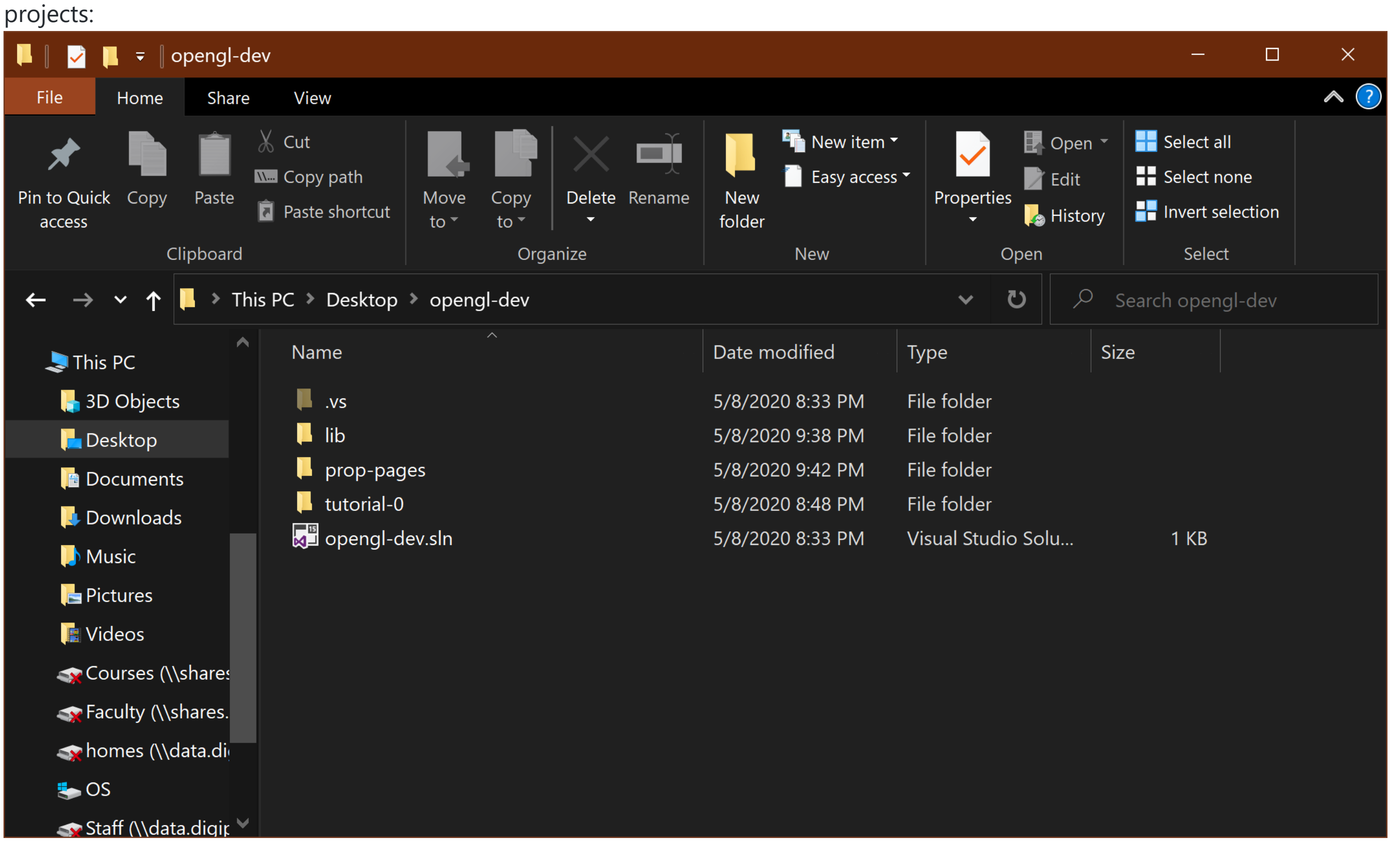
The following steps describe how to create and save a property sheet, and attach this property sheet to a project. Later the parameters specified in this property page will help us build and execute our first OpenGL application.

1. Begin by moving the previously created `lib` folder into the folder containing the newly created solution folder `opengl-dev` .

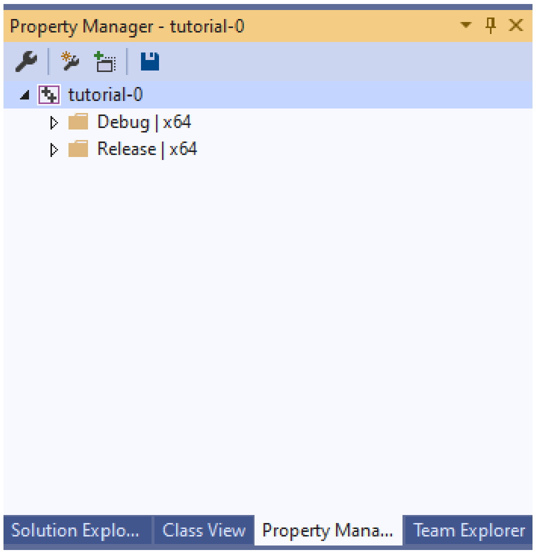


2. Create a single property sheet that can be used by all projects in the solution **opengl-dev** that interface with OpenGL:

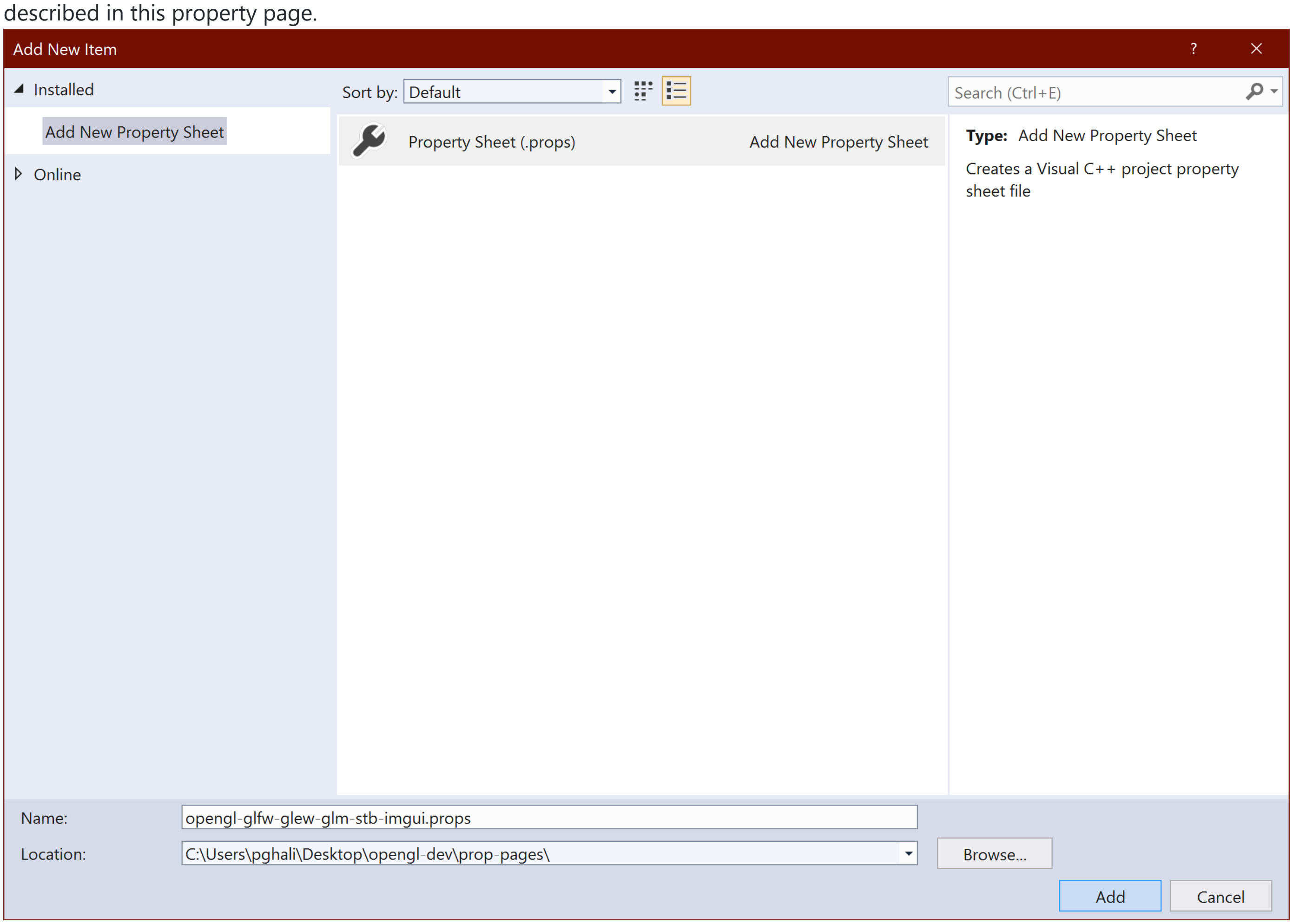
   - Create a subdirectory `prop-pages` in the solution directory `opengl-dev` to store the various property pages used by different projects:



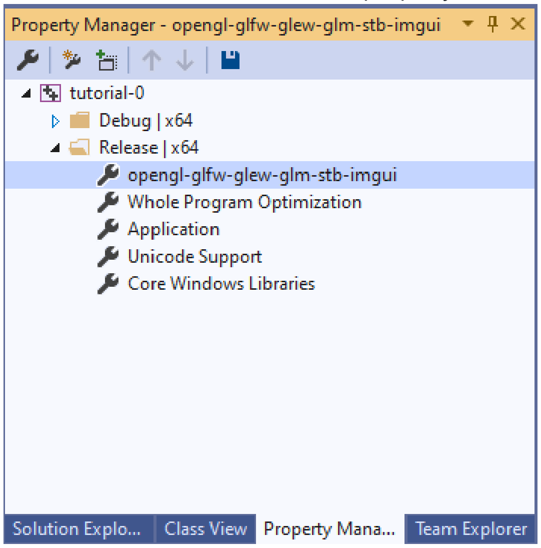
   - In the main menu, click **View** > **Other Windows** > **Property Manager**. The **Property Manager** appears as a tab along with the **Solution Explorer** in the **Solution Explorer** view window:



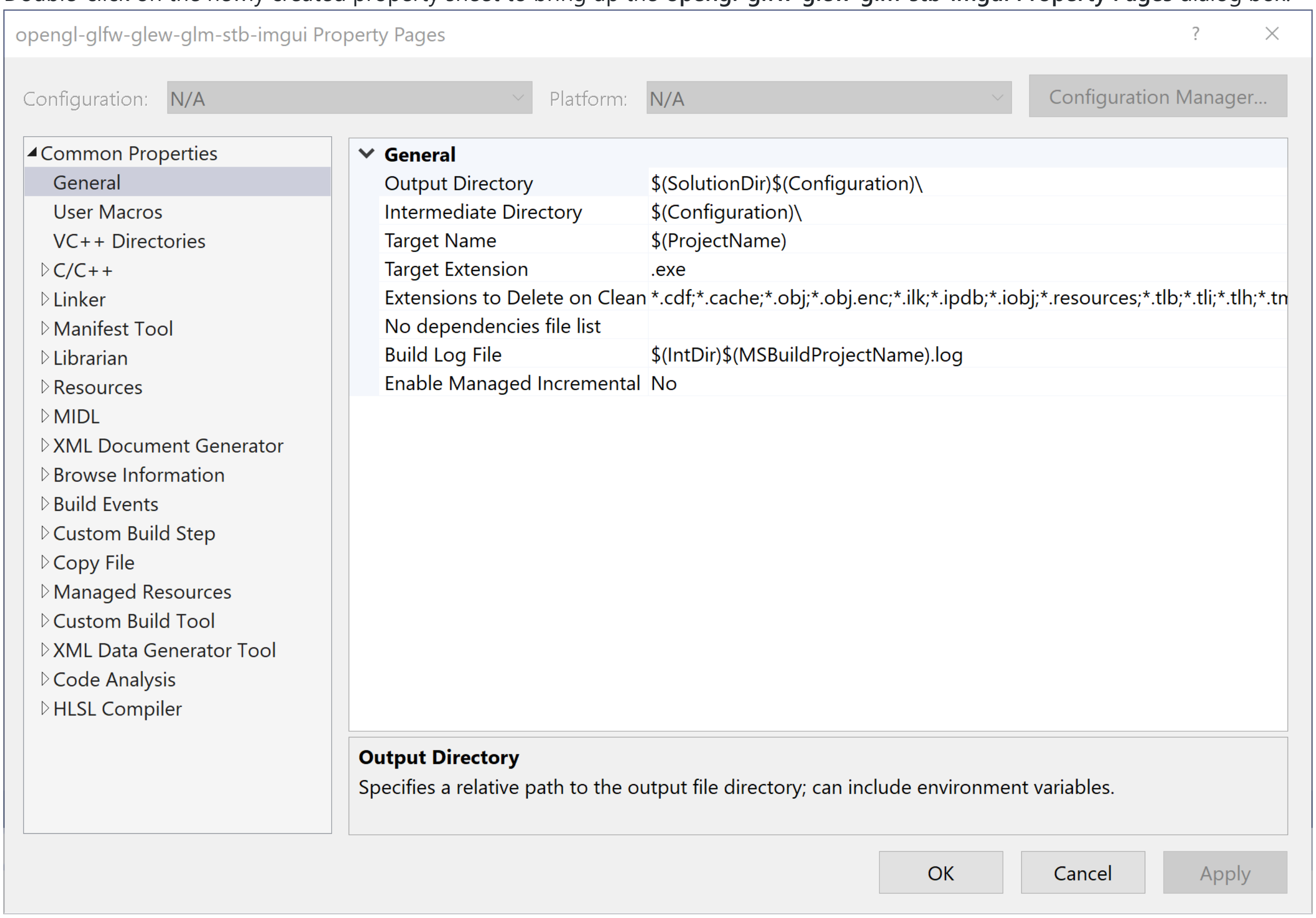
- Expand **tutorial-0** tree, right-click on **Release | x64** and select **Add New Project Property Sheet ...**. Next, enter **opengl-glfw-glew-glm-stb-imgui.props** in the **Name** field and a path in **Location** field to store the property page in directory `prop-pages` . The property page's name is lengthy and descriptive to clearly indicate the helper libraries whose properties and parameters are described in this property page.



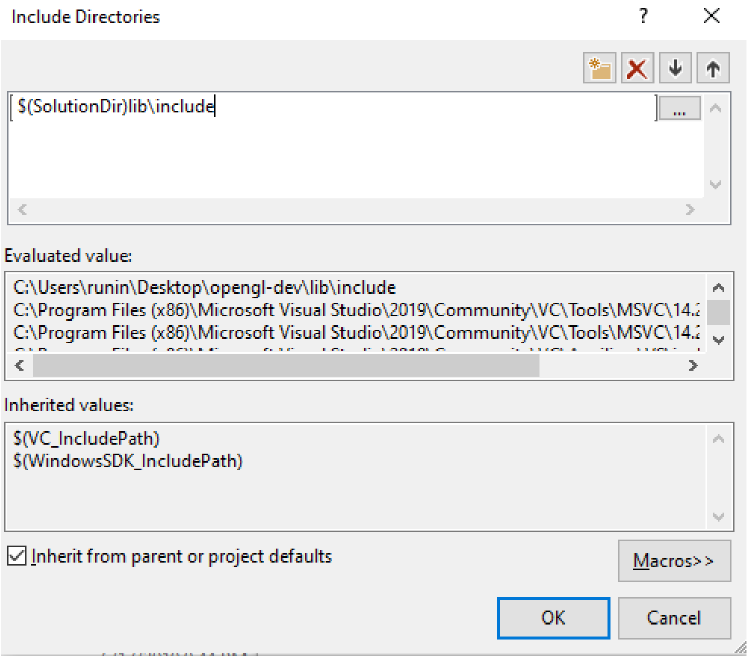
- Choose the **Add** button. The new property sheet is added to the **Release** > **x64** subtree:



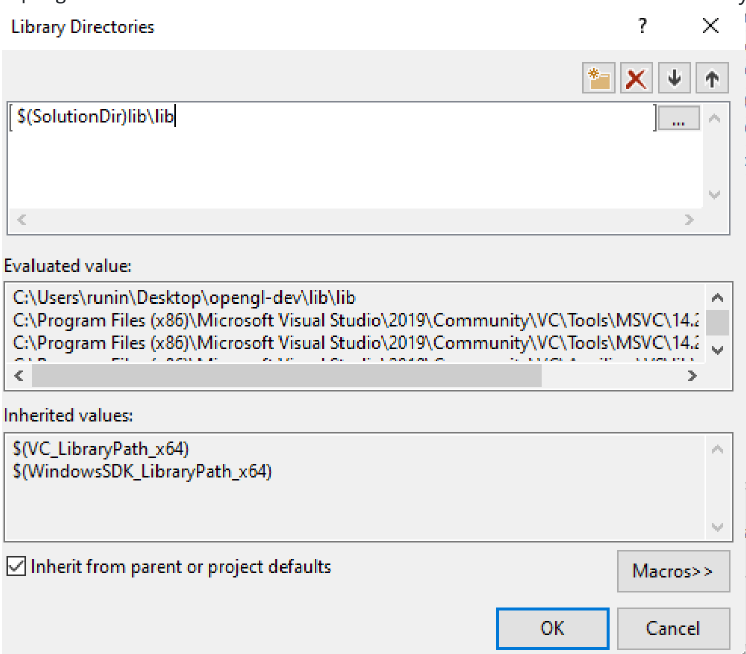
- Double-click on the newly created property sheet to bring up the **opengl-glfw-glew-glm-stb-imgui Property Pages** dialog box:



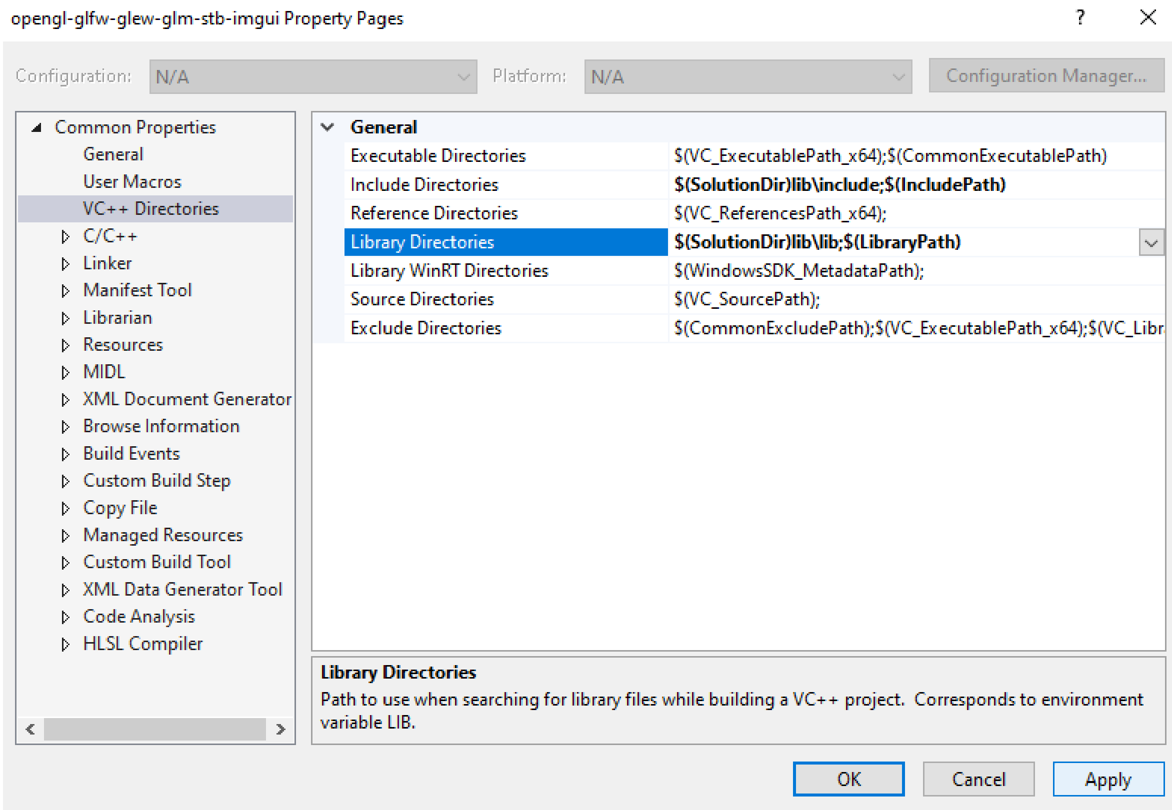
- In the left pane, click on **Common Properties** > **VC++ Directories**. In the right pane, choose **Include Directories** field to add paths to header files of GLFW, GLEW, glm, stb, and ImGUI packages as shown in the following picture. After entering these paths, choose **OK** followed by **Apply** to ensure paths to header files are saved.



- Choose **Library Directories** field to add paths to GLFW and GLFW import library files as shown in the following picture. GLM and stb are header-only libraries while ImGui provides both header and source files and will therefore not require linkage. The path to `opengl32.lib` is known to Visual Studio and doesn't need to be set by the programmer.

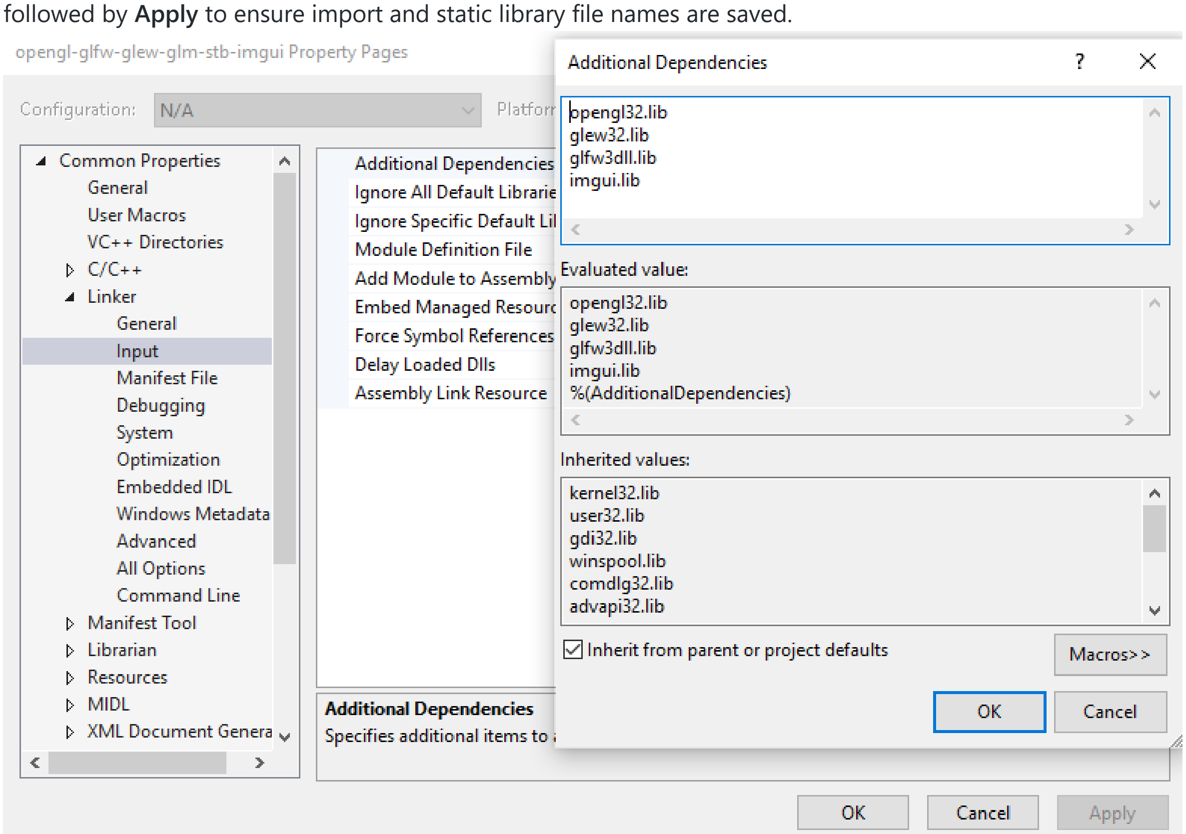


- After entering these paths, choose **OK** followed by **Apply** to ensure paths to binary library files are saved.



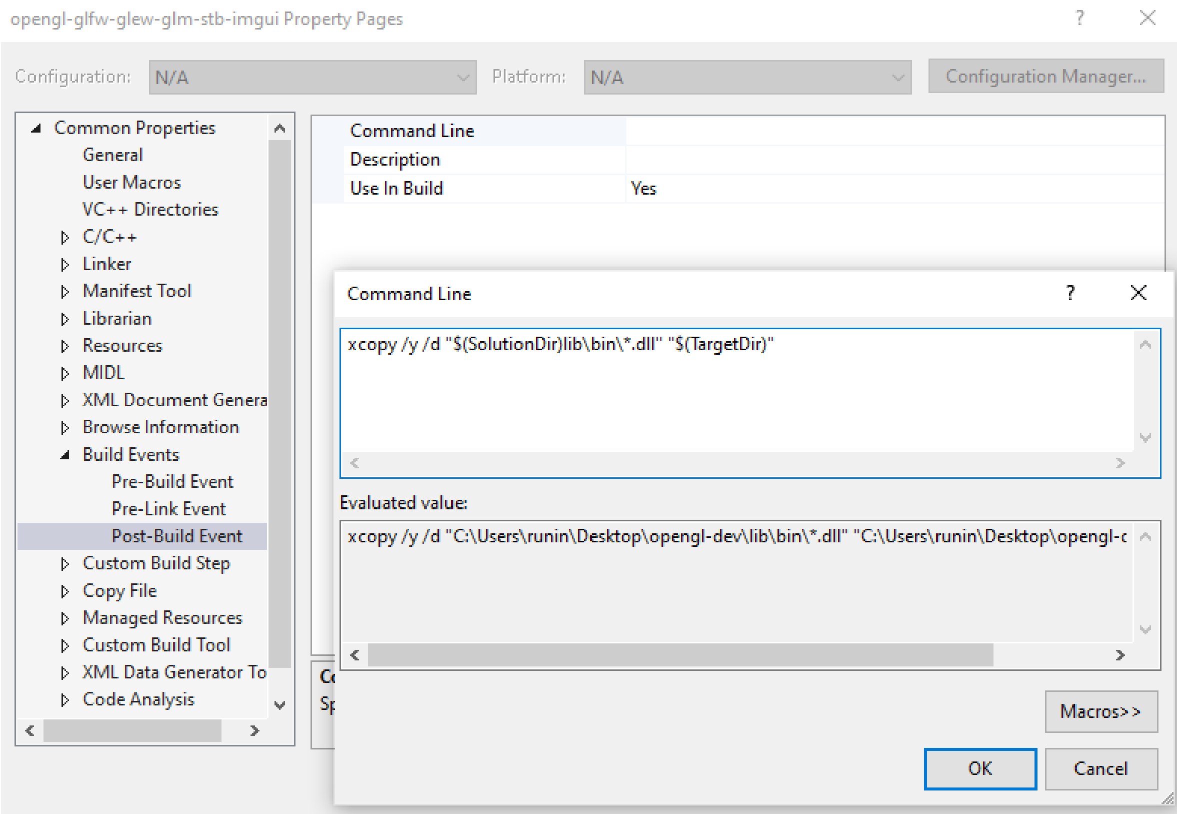
3. Providing names of library files to the linker:

   - While the previous setting provides paths to where import and linkage files are located, the linker will require names of specific libraries to use for OpenGL, GLFW, and GLEW. In the left pane of the **openGL-glfw-glew-glm-stb-imgui Property Pages** dialog box, expand the **Linker** subtree and click on **Input** item. Each library file name is added - one line at a time - to the **Additional Dependencies** field of the right pane as shown in the following screenshot. After entering these library names, choose **OK** followed by **Apply** to ensure import and static library file names are saved.



4. Providing DLLs to the runtime executable:

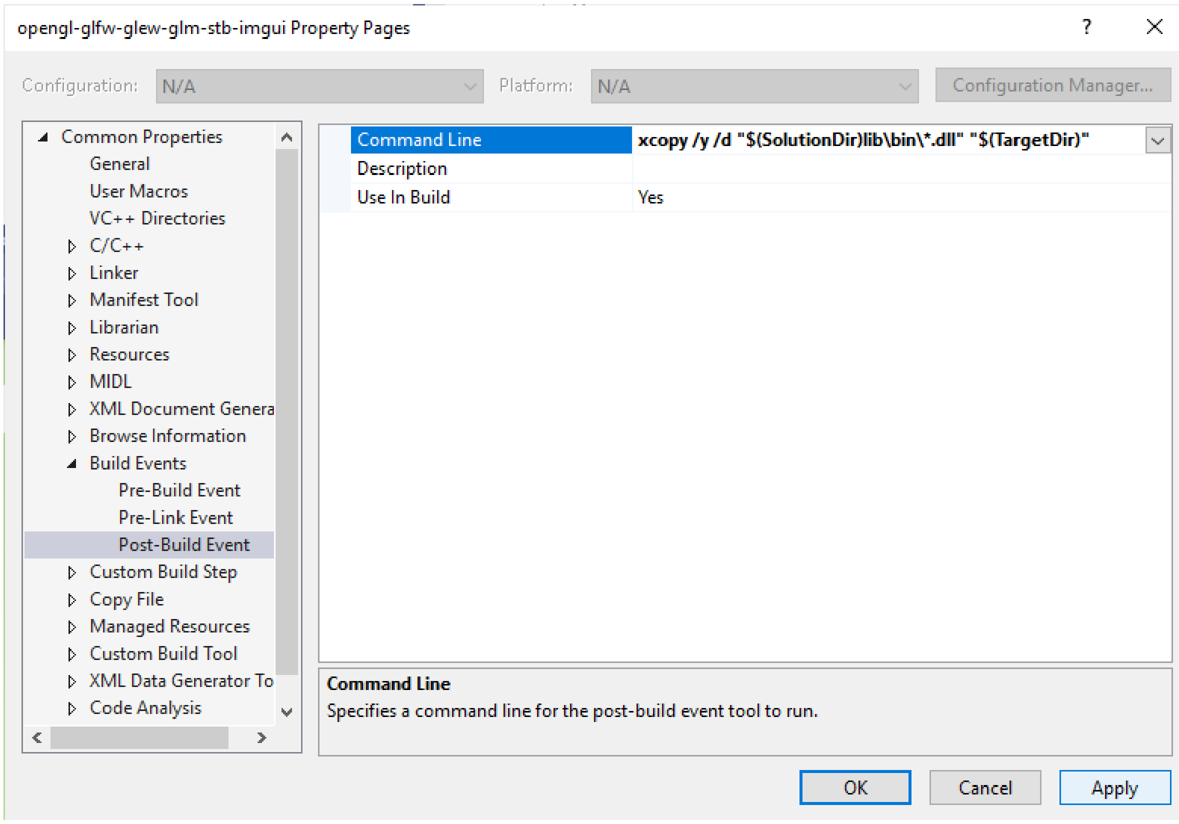
   - The runtime executable must be able to load dynamic-linked libraries `opengl32.dll` , `glew32.dll` , and `glfw3.dll` . Visual Studio knows the location of `opengl32.dll` and can directly provide this file to the runtime executable. However, the other two DLLs must be copied to the folder containing the project executable which is `x64\Release` folder for Release configuration or `x64\Debug` folder for Debug configuration. Instead of manually copying these DLLs, we can use a **Post-Build Event** to have Visual Studio automatically copy these DLLs to the folder containing the project executable. In the left pane of the **openGL-glfw-glew-glm-stb-imgui Property Pages** dialog box, expand **Build Events** tree and choose **Post-Build Event**. In the right pane, edit the **Command Line** field to add the commands listed in the following figure.

     `xcopy` is a windows program to copy files/directories on the command line. The `*` in `*.dll` is a wildcard that tells `xcopy` to copy any file that ends with the `.dll` extension.

     Note the use of quote characters `"` , this prevents issues from file paths that have whitespace characters in them.

   - After entering the commands, choose **OK** followed by **Apply** to ensure this setting is saved.

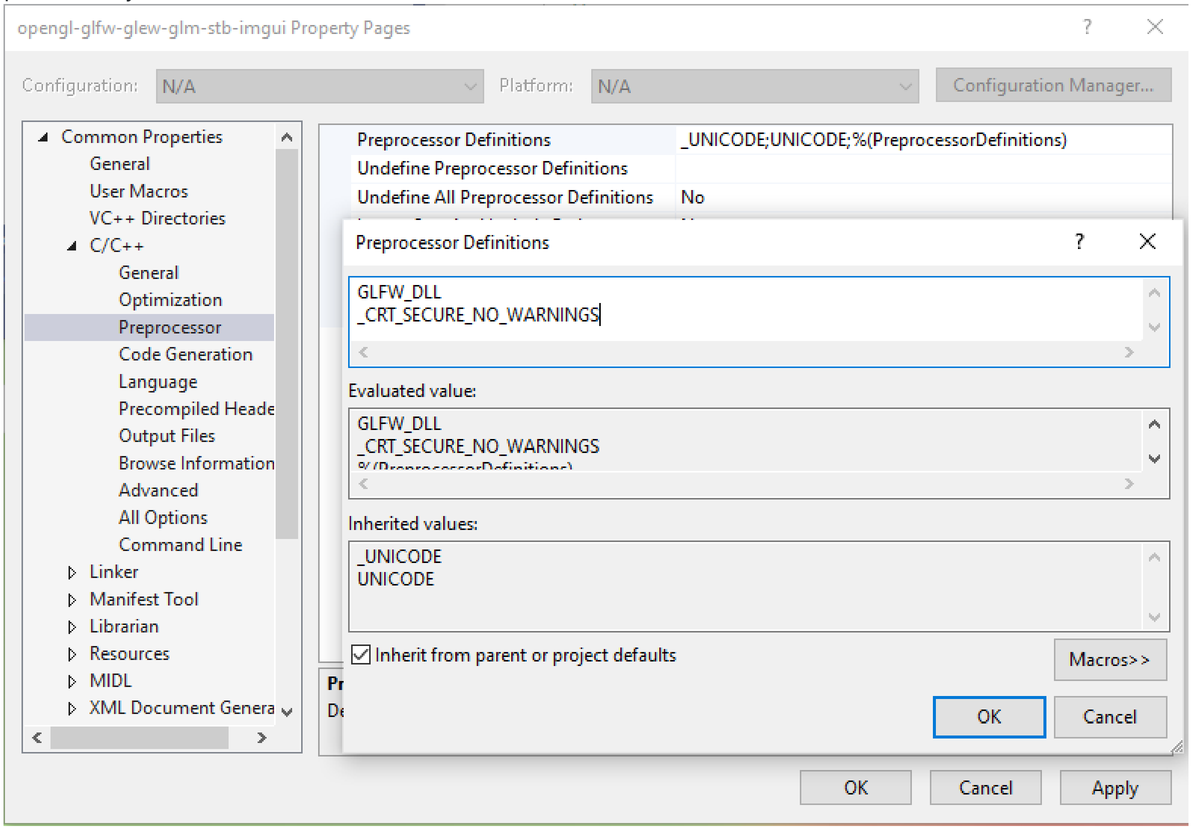


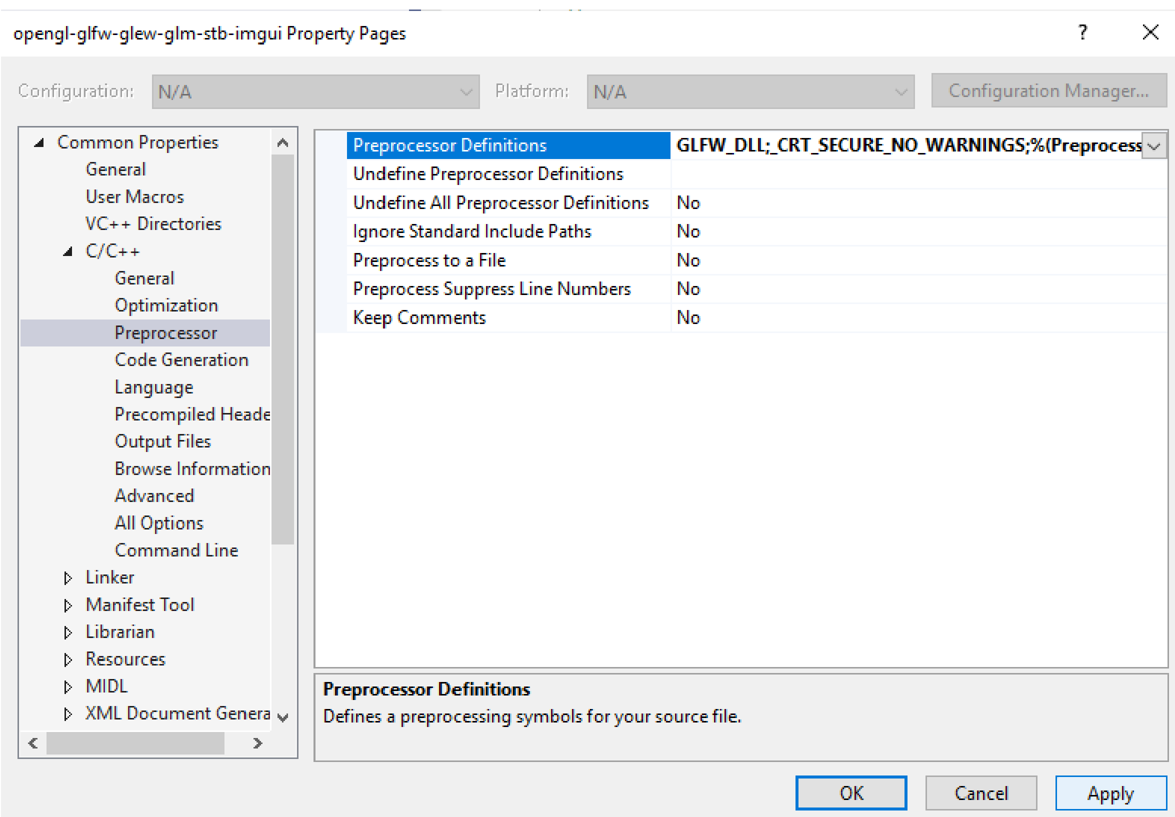
5. Adding commonly used preprocessor definitions

   - Using DLL version of GLFW: When compiling an application that uses the DLL version of GLFW, we're required to define the `GLFW_DLL` macro before the GLFW header `glfw3.h` is included. Rather than adding this macro to source files, we add this once to the property page. In the left pane of the **openGL-glfw-glew-glm-stb-imgui Property Pages** dialog box, expand **C/C++** tree and choose **Preprocessor**. In the right pane, edit the **Preprocessor Definitions** field and add `GLFW_DLL`.

   - Eliminating security features in CRT (C runtime library): Microsoft's C/C++ standard libraries have incorporated secure versions for many functions. If a secure version exists, it has a `_s` ("secure") suffix. These secure functions are non-portable to other platforms. By default, Visual Studio generates warnings when older, less secure functions are called. These warnings can be eliminated by using `warning` pragma or by defining `_CRT_SECURE_NO_WARNINGS` in source code or by incorporating into the **openGL-glfw-glew-glm-stb-imgui** Property Pages by adding `_CRT_SECURE_NO_WARNINGS`.

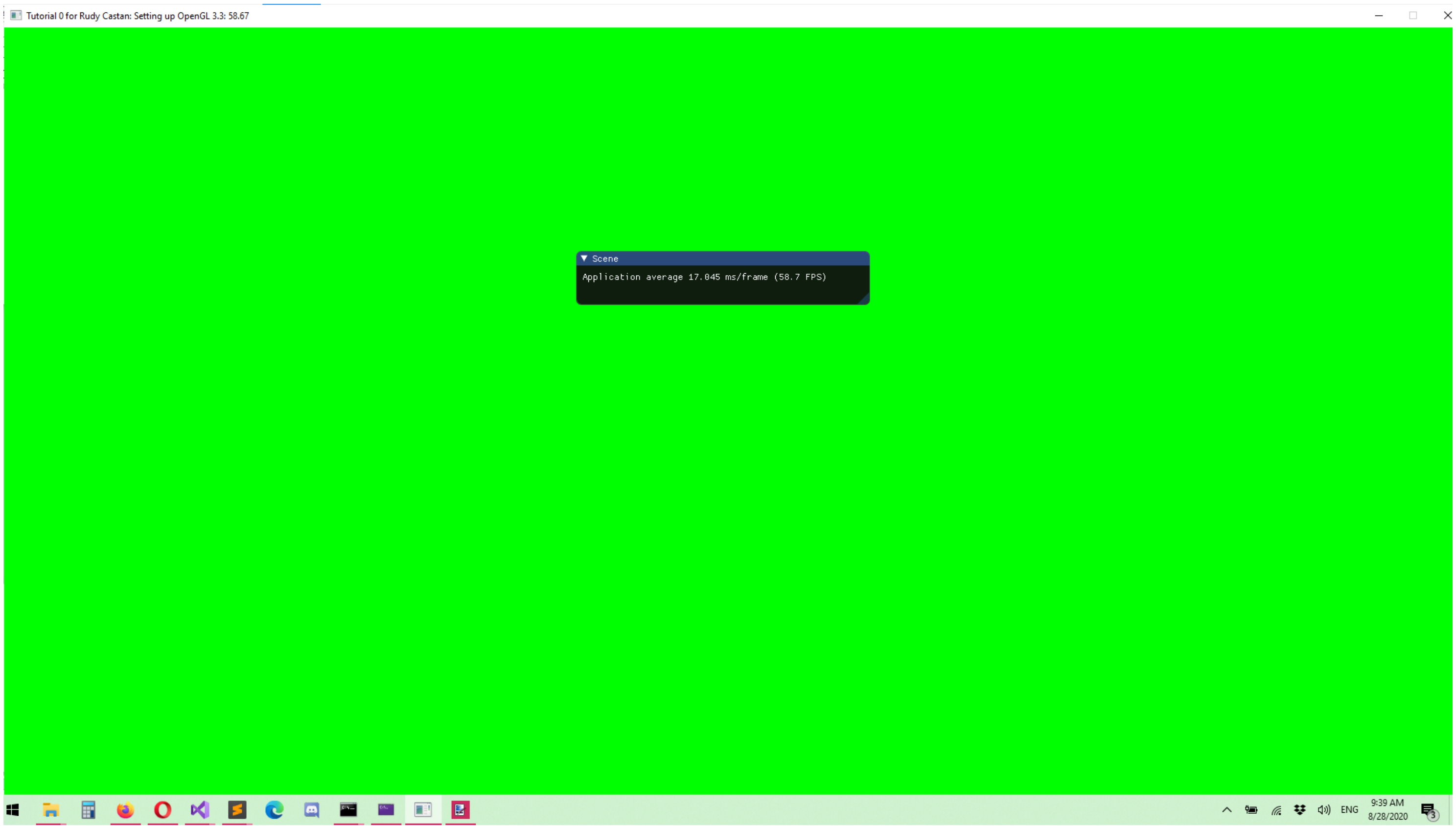
- Use the following screenshot to enter the preprocessor definitions and choose **OK** followed by **Apply** to ensure the setting is permanently saved.





Testing environment settings:

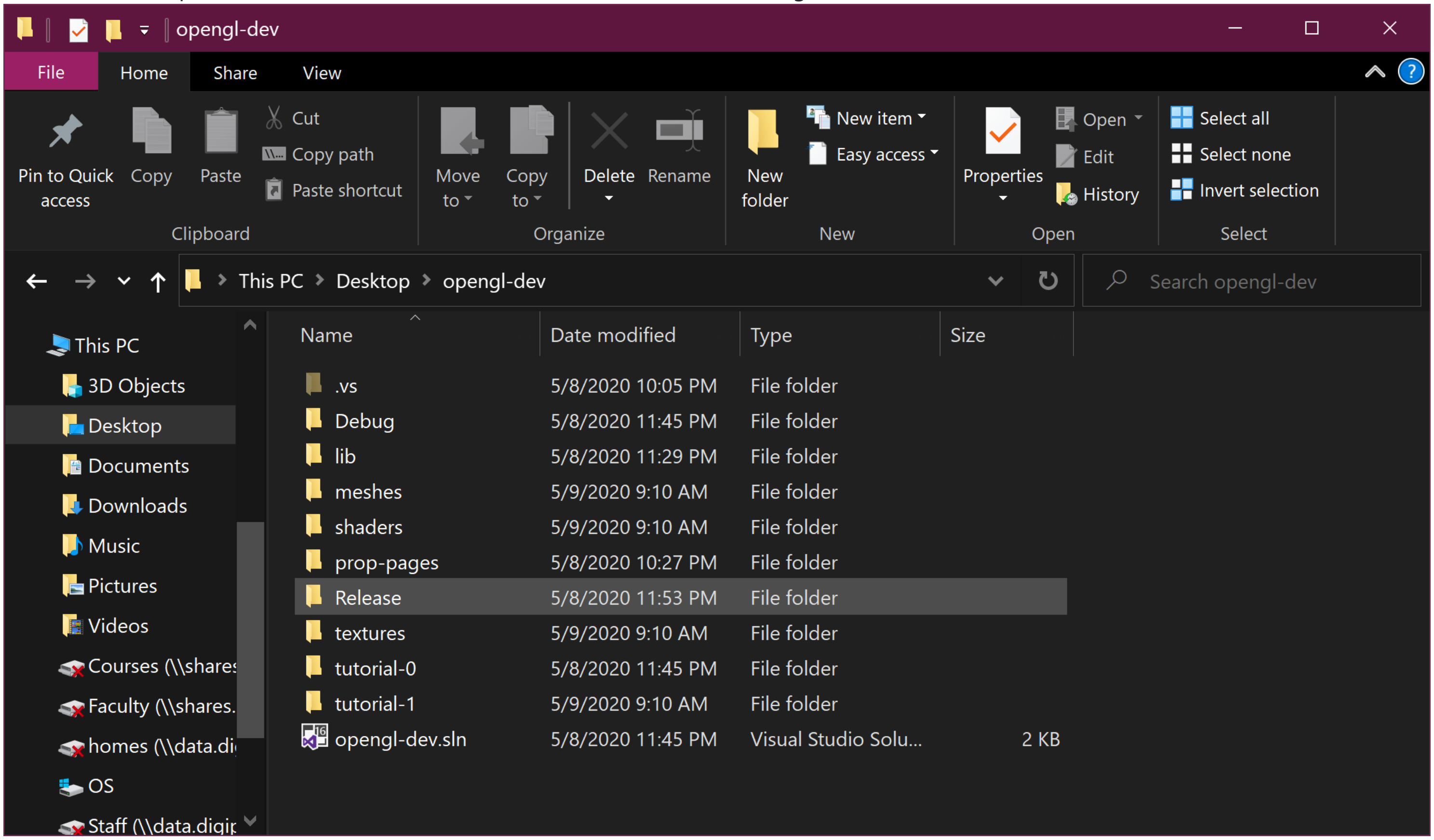
- In the **Release** > **x64** configuration (the same configuration in which the **openGL-glfw-glew-glm-stb-imgui** property page was created), compile, link and execute. If the parameters were correctly edited in the property page, the build and post-build events will be successful causing a window with a green background to be visible.



- Now, test the properties in the **Debug** > **x64** configuration. In the **Property Manager** tab, right-click on **Debug** > **x64** and choose **Add Existing Property Sheet ...**. Now, select the previously created **openGL-glfw-glew-glm-stb-imgui** property page. If the build and post-build events were successful, again a green window should be visible.
- Create a new project in the same solution using the same property page and source file. Ensure that an empty window with green background should be visible when this new project is compiled, linked, and executed for both **Debug** and **Release** configurations.
- Repeat the entire process at a different location on your machine. Create a new Visual Studio Solution in a different folder. Copy the previously created **openGL-glfw-glew-glm-stb-imgui** property pages to the new folder. Ensure that you can compile, link, and executenewly created projects for this new solution.
- One final test will require you to copy your `opengl-dev` folder to a different machine and repeat the compile, link, execute steps successfully.

Final Notes

Make sure that your directory structure matches the structure illustrated in the picture below. The `shaders` , `geom` , `textures` , `tutorial-1` directories don't yet exist but will in future tutorials and assignments. This is crucial!!! Any differences in directory structures will result in your submissions working correctly in your environments but not in the grader's environment. Your submission may not compile, or link, or execute because paths to textures, shaders, header files will not match in the grader's machine.



Submission

Course Site Submission

1. You are required to submit the following property page file `opengl-glfw-glew-glm-stb-imgui.props` .
2. You are required to submit a 64bit exe of the tutorial named `tutorial-0.exe`

Grading Rubric

- [ ] [core] Created `opengl-glfw-glew-glm-stb-imgui.props` file that configures eveything needed to build the tutorial for the Debug:x64/Release:x64 confifurations
- [ ] [core] Submitted `tutorial-0.exe` which displays a green background, imgui text with fps information, and a window title updated with the students name
- [ ] No hardcoded paths were used in the `opengl-glfw-glew-glm-stb-imgui.props` file but used macros like `$(SolutionDir)` and `$(TargetDir)`
- [ ] Properly used quotes `"` in the Post Build Event settings
- [ ] Added warning level 4, treat warnings as errors and C++17 language version settings to `opengl-glfw-glew-glm-stb-imgui.props`
- [ ] All source files compile without warnings and without errors.
- [ ] Correct files submitted with correct names. No unnecessary files submitted.
- [ ] Correct files committed with correct names. No unnecessary files committed.

Scores for Assignments will be given as the following:

Score	Assessment
Zero	Nothing turned in at all
Failing	Close to meeting core requirements
Rudimentary	Meets all of the core requirements
Satisfactory	Close to meeting all requirements
Good	Clearly meets all requirements
Excellent	High quality, well beyond the requirements