

Tutorial 0: Configuring OpenGL 4.5

Things to be covered in this tutorial:

- Overview of OpenGL
- Setting up OpenGL 4.5
- Generating your first OpenGL tutorial
- Tutorial task: Changing background color with key events
- Submission guidelines and pre-verification step
- Grading rubrics instructions

Overview of OpenGL

OpenGL stands for Open Graphics library and is a *specification* that describes a low-level library or application programming interface [API] 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.

- Older versions of graphics hardware implemented a graphics rendering pipeline consisting of rendering algorithms and formulae that were chosen by hardware designers and frozen in silicon. OpenGL exposed this fixed functionality of graphics hardware to application developers with the disadvantage of making it cumbersome for developers to provide a wide range of interesting onscreen features beyond the techniques frozen in silicon. Developers demanded a variety of new features in hardware in order to make it more convenient to provide more and more sophisticated onscreen effects. New graphics hardware designs in the 21st century became more and more programmable. For OpenGL this meant the creation of a standard, cross-platform, high-level shading language called the [OpenGL Shading Language](#) [GLSL] that allows application developers to write their own versions of graphics rendering algorithms that would then be executed by graphics hardware. Thus, GLSL lets application developers take full and total control over many important stages of the rendering pipeline and thus the ability to create stunning effects in real time. 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 [hereafter referred to as just Windows and is not to be confused with a window which is a viewing area on the display system generated by 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.

When you develop a graphics application, you define a scene consisting of objects, the geometrical and appearance properties for these objects, and a synthetic camera for viewing the scene. Objects' geometries are defined by their vertices, their normals, and their graphics primitives that encompass points, lines, or triangles. Appearance is specified by defining color, shading, materials, lighting, and texture maps. The properties of the synthetic camera are defined by viewing and projection transformation matrices. For graphics hardware to generate appropriate and correct images, objects' geometrical and appearance data must be associated with specific hardware

commands. The mathematical and algorithmic process of going from an object's geometry and appearance data to an on-screen image can be divided into a number of conceptual steps or stages. Orders issued to the hardware must have an explicit and well known order at every stage. Data and commands must follow a path and have to pass through some stages, and that path cannot be altered. This path is commonly called the [graphics rendering pipeline](#). Think of it like a pipe where you insert some data into one end - vertices, textures, shaders - and they start to travel through some small machines that perform very precise and concrete operations on their input data to generate new data that is then transmitted to the next machine until the final machine generates output at the other end: the rendered image.

When you develop a graphics application with the OpenGL API, you're using high-level C-based functions to control the graphics hardware's rendering pipeline. As explained before, the rendering pipeline consists of a large number of stages that each operate on a small and specific detail of the larger problem of creating a rendered image. The process of turning a stage's inputs into appropriate outputs is controlled by a large number of state settings. Cumulatively, these state settings define the behavior of the rendering pipeline and the way in which primitives are transformed into pixels.

OpenGL defines an internal data structure called a [context](#) to keep track of state settings and operations to be applied on input data. Every OpenGL-based application must create at least one context and make it active in order to perform any rendering. 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. But before the rendering context can be created, a GDI device context is required, and in turn, 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 a graphics driver and an operating system. Graphics drivers are implementations by hardware vendors of the OpenGL specification and extensions to manage graphics hardware. At runtime, drivers translate calls to OpenGL API functions 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:

- [gl.h](#) which contains declarations of functions defined in OpenGL version 1.1. This file ships with Windows.

- [glcorearb.h](#) which 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](#) which 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](#) [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 resolve references made to functions exported by [opengl32.dll](#). The import library further supplies the runtime system with information needed to load the DLL [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](#). The object files generated by compiling source 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.dll](#) into memory. At runtime, calls made by the user program to OpenGL 1.1 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, programmers must download header files [glcorearb.h](#) to obtain interfaces to higher versions of OpenGL specification and extensions and [wglext.h](#) for Windows-specific extension interfaces. Second, 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 higher versions of the 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.

- *GLFW: A platform-independent API for abstracting operating system details*
 - 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 module will use a popular toolkit called [GLFW](#) to implement programming assessments.
- *OpenGL Extension Wrangler Library [GLEW]: A cross-platform OpenGL extension loading library*
 - Recall that since Windows ships with OpenGL 1.1, programmers must download header files [glcorearb.h](#) [to obtain interfaces to higher versions of OpenGL specification and extensions] 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. The application will dereference these manually obtained function pointers and thereby obtain exposure to the functionalities provided by the latest OpenGL specifications and extensions. Unfortunately, there are many OpenGL specifications 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 [libraries](#) have been implemented to abstract away the header files and function pointer inconveniences. We'll use one such platform-independent library called [GLEW](#) to provide the necessary header files and expose the functionality of more modern OpenGL specifications and corresponding extensions.
- *OpenGL Mathematics [GLM]: A mathematics library for graphics programming*
 - 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.
- *Tinyobjloader: Header only 3D mesh loading*

- [This](#) is a lightweight, header-only C++ library for loading and parsing [Wavefront obj](#) files, which are commonly used for storing 3D geometry and material information. Certain assignments will require you to write code to manually parse **OBJ** files using the **OBJ** file format specification
- GLEQ: *global event queue for synchronizing GLFW events*
 - [GLEQ](#) is a lightweight, header-only C++ library designed for [event-driven](#) rendering for GLFW. It adds GLFW keyboard, mouse, and window events to a single global event queue and thus assist in synchronizing rendering with the application's event loop.

Setting up OpenGL programming environment

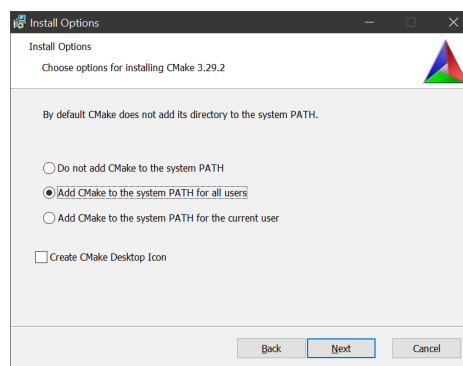
Prerequisites:

This module provides a OpenGL sandbox script [icg.bat](#) to streamline the setup and build process for C/C++-based OpenGL projects for Visual Studio 2022 IDE. The script provides developers a convenient workflow process that automates tasks such as configuring a Visual Studio 2022 solution, project initialization, and dependency management by creating resource directories for holding scene files, meshes, and images. Further, the script provides options for building [an executable file], cleaning [project of object and executable files], and running tests on the generated executable file.

Before running the script, complete the following installations:

1. Install CMake for Windows

1. CMake is a configuration tool that simplifies the setting up of a programming environment for writing OpenGL-based applications using an integrated programming environment such as Microsoft's Visual Studio 2022.
2. Begin by downloading the latest version of CMake Windows x64 Installer file [with extension **msi**] from [here](#).
3. Run the downloaded installer. It is important that you add CMake to the system PATH [see picture below]. Use default settings for all other installation instructions.



2. Install GIT for Windows

1. GIT is a version control system that we use to provide a consistent set of files including appropriate versions of libraries and appropriate versions of starter code for programming assignments. That is, GIT allows your application programs to remain compatible with updated libraries and starter code.
2. Begin by downloading the 64-bit version of GIT from [here](#).
3. Run the downloaded installer and follow the default installation instructions.
3. Install Microsoft's Visual Studio 2022 using the instructions listed [here](#).

4. Install Doxygen for Windows

1. Doxygen is an automated documentation formatter and generator for source files implemented in C, C++, and a variety of other programming languages. It allows you to specially tag comments in your source files that will be used to generate nicely formatted HTML and Latex output.
2. Download and install the latest 64-bit version of Doxygen from [here](#).

OpenGL 4.5 Sandbox Script

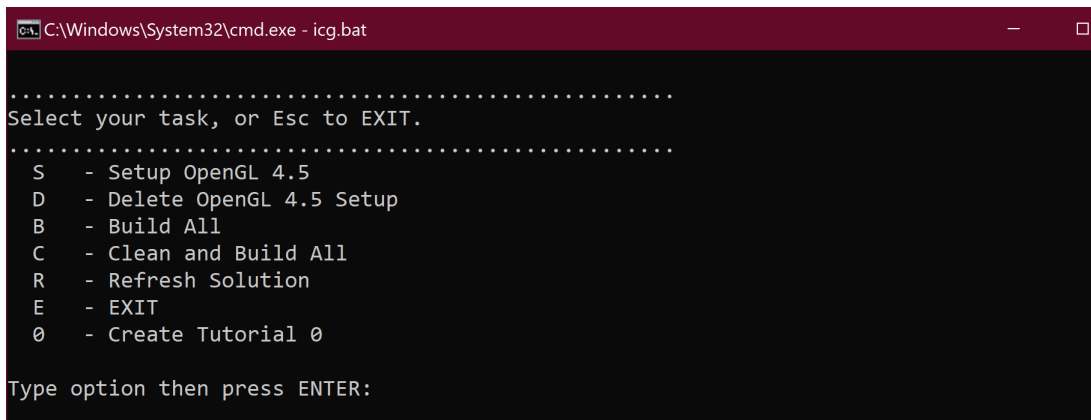
Download the OpenGL 4.5 sandbox script `icg.bat` from the assessment web page. Copy this file into a directory, say `icg-opengl-dev`. Use File Explorer to open directory `icg-opengl-dev`. Open the command-line shell by typing `cmd` [and pressing Enter in the Address Bar]. Execute script `icg.bat` [by typing the script's name in the shell and then pressing Enter].



```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.19045.4651]
(c) Microsoft Corporation. All rights reserved.

C:\Users\pghali\Desktop\icg-opengl-dev>
```

You will be presented with the following options:



```
C:\Windows\System32\cmd.exe - icg.bat

.....
Select your task, or Esc to EXIT.
.....
S  - Setup OpenGL 4.5
D  - Delete OpenGL 4.5 Setup
B  - Build All
C  - Clean and Build All
R  - Refresh Solution
E  - EXIT
0  - Create Tutorial 0

Type option then press ENTER:
```

Here's a summary of the menu options:

1. Option **S** - OpenGL Setup and Create Solution:

This option performs all of the boilerplate drudgery required for developing C/C++-based OpenGL applications the Visual Studio 2022 IDE on your machine. It creates the necessary directory structure [see picture below] and populates the `lib` directory with all the necessary helper libraries including GLFW, GLEW, and GLM from their Git repositories. It additionally produces a CMake configuration file called `CMakeLists.txt` which is utilized by CMake build commands to generate a Visual Studio 2022 solution within the `build` directory.

```
1  |  icg-opengl-dev/  #  📁 Sandbox directory for all assessments
2  |  |  build/         #  📁 Build files will exist here
3  |  |  lib/          #  📁 Helper libraries exist here
4  |  |  projects/     #  📁 Assessments implemented here
5  |  |  test-submissions #  ⚠️ Test submissions here before uploading
6  |  |  CMakeLists.txt #  ⚙️ CMake configuration file
7  |  |  icg.bat       #  📄 Automation Script
```

2. Option **D - Delete OpenGL Setup**:

This option undoes the activities performed by option **S** by removing libraries, build directories and deleting all **Git** and **CMake** configurations. This option is helpful in repairing broken **Git** and **CMake** configurations and for re-installing helper libraries and assets.

3. Option **B - Build All**:

This option compiles all projects at once and builds executables in release mode. It is an incremental build compiling only the source files that have been modified since the last build, rather than recompiling entire projects.

4. Option **C - Clean and Build All**:

For each project, this option recompiles source files followed by a link operation to generate an executable.

5. Option **R - Refresh Solution**:

Use this option when new source or header files are added to a project. In the background, the script parses the projects and test submissions for new source code and updates the **CMake** configuration file.

6. Option **E - Exit**:

Exits the OpenGL 4.5 sandbox.

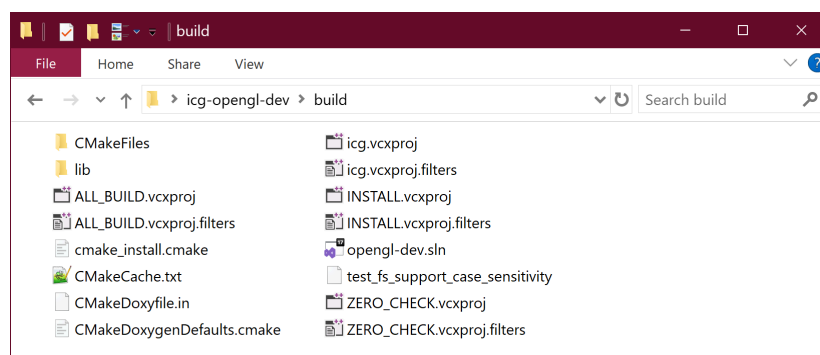
7. Option **0 - Create Project for Tutorial 0**:

This option create a Visual Studio 2022 project titled **tutorial-0**. As we proceed through this module, you will find more options to create subsequent assessments in updated versions of **icg.bat**.

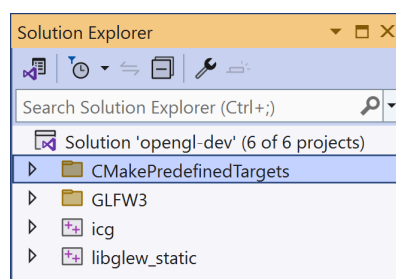
Implementing an OpenGL application

This section provides a detailed step-by-step guide to building your first successful OpenGL application:

1. Select option **S** to setup OpenGL. This option also create a Visual Studio 2022 solution called **opengl-dev.sln** in directory **icg-opengl-dev\build**:



Double click solution file **opengl-dev.sln** to open Visual Studio 2022 IDE with predefined projects of helper libraries. The *Solution explorer* pane of the IDE is shown below:



By default Visual Studio 2022 *IDE is set to Debug mode. Unless you're interested in debugging your code, always test and submit your code in Release mode.*

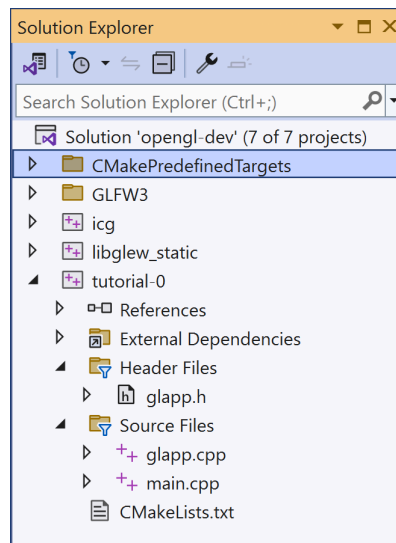
2. Select option 0 to create a nested directory **tutorial-0** in directory **projects** directory and pull starter code from a Git repository. The directory layout of **tutorial-0** is shown below:

```

1  |  icg-opengl-dev/  #  📁 Sandbox directory for all assessments
2  |  |  projects/    #  📁 Assessments implemented here
3  |  |  |  tutorial-0  #  ➕ Tutorial 0 assessment
4  |  |  |  |  include  #  📄 Header files - *.hpp and *.h files
5  |  |  |  |  src      #  🌟 Source files - *.cpp and *.c files
6  |  |  |  |  icg.bat   #  📄 Automation Script

```

This option also creates a Visual Studio 2022 project **tutorial-0.vcxproj** in the **build** directory which in turn will also update the Visual Studio 2022 solution file **opengl-dev.sln** by adding a new project titled **tutorial-0**. If you're currently in the Solution Explorer, a dialog box will be displayed requesting an update. Press the Reload button to refresh the Solution Explorer to reflect the updated solution file.



3. Right-click on project **tutorial-0** in the Solution Explorer pane and select option Set as Startup Project:
4. Build project **tutorial-0** in Visual Studio 2022 IDE using Ctrl + B.

By default Visual Studio 2022 *IDE is set to Debug mode. Unless you're interested in debugging your code, always test and submit your code in Release mode.*

Alternatively, select option **B - Build All** in script **icg.bat** to build all projects in Release mode.

5. Execute **tutorial-0** application in Visual Studio 2022 IDE using Ctrl+5. The expected output must resemble:



Alternatively, you can run the executable from [build](#) directory by typing the following in the command-line shell:

```
1 | C:\icg-opengl-dev\build> Release\tutorial-0.exe
```

🎉 Congratulations! Seeing this output indicates that you've successfully set up the OpenGL programming environment that will be used for all programming assessments in this module. Forthcoming assessments will provide detailed understanding of both the starter code and OpenGL API.

Command-line parameters

All programming assessments [built using the steps specified in the previous section] support command-line parameters. The `--help` command-line parameter reveals available options:

```
1 | C:\icg-opengl-dev\build> Release\tutorial-0.exe --help
```

```
C:\Windows\System32\cmd.exe

C:\Users\pghali\Desktop\icg-opengl-dev\build\Release>tutorial-0.exe --help
Usage: tutorial-0.exe [-h] [-f <filename>] [-r ] [-p ]
Options:
-a, --help      Display this help message
-f, --file      Specify a record or playback filename
-r, --record    Render the executable and record event into log file
-p, --play      Render the executable and playback the event from log file
-w, --width     Specify window width
-h, --height    Specify window height
```

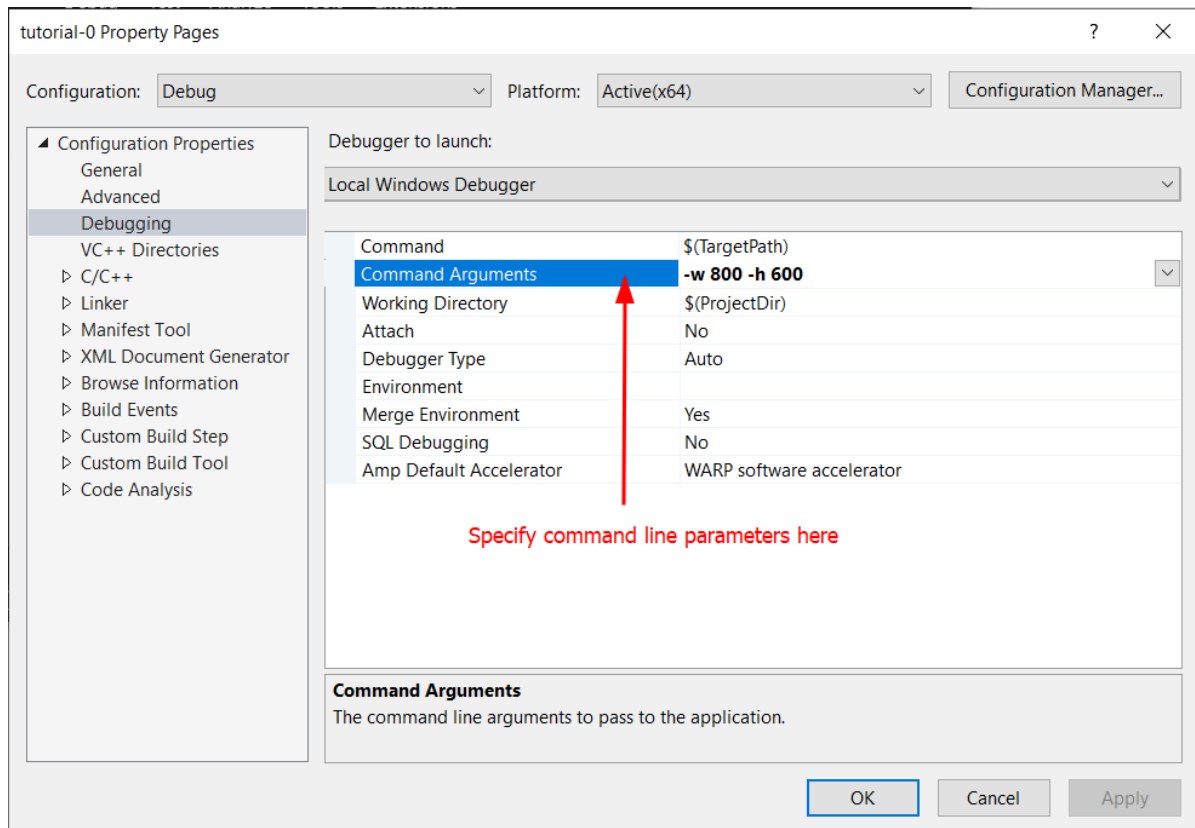
For example, you can specify the window's width and height using command-line parameters. In case you have a HD screen, you can display images using a higher resolution:

```
1 | C:\icg-opengl-dev\build> Release\tutorial-0.exe -w 2400 -h 1350
```

If you're unfortunately saddled with a low resolution display device, you specify a lower resolution:

```
1 | C:\icg-opengl-dev\build> Release\tutorial-0.exe -w 800 -h 600
```

To set command-line arguments in Visual Studio 2022, right click on project **tutorial-0**, then go to Properties. In the Properties Pane → Debugging pane, you'll find a field called Command Arguments. Add the command-line parameters you'd like to provide to the program in this field:



Directory Structure


It is crucial [⚠️ for your grades if nothing else] that your directory structure matches the structure illustrated in the picture below.

💡 **Retain this organizational structure of storing source files in directory **src** and header files in directory **include** in every new project that you create. The use of sandbox script **icg.bat** is strongly recommended because it will create and manage this directory structure for you.**

```

1  |  icg-opengl-dev/ # 📁 Sandbox directory for all assessments
2  |  |  build/      # 📁 All build files for your platform exist here
3  |  |  lib/        # 📁 All helper libraries are pulled into this directory.
4  |  |  tutorials/  # 📁 All programming assessments must exist here
5  |  |  |  tutorial-0 # 📁 Tutorial 0 assessment
6  |  |  |  |  include # 📁 Header files - *.hpp and *.h files
7  |  |  |  |  src     # 📁 Source files - *.cpp and .c files
8  |  |  |  tutorial-1 # 📁 Tutorial 1 assessment
9  |  |  |  |  include # 📁 Header files - *.hpp and *.h files
10 |  |  |  |  src     # 📁 Source files - *.cpp and .c files
11 |  |  test-submissions # ⚠️ Test submissions here before uploading
12 |  |  |  <name>-tutorial-0 # 📁 Assuming you are submitting Tutorial 0
13 |  |  CMakeLists.txt      # ⚙️ CMake configuration file
14 |  |  icg.bat              # 📄 Automation Script

```

 Directories **tutorial-1**, **tutorial-2** and so on don't yet exist but will be created in upcoming assessments.

Task 1: Change background color with key events

Objective:

The goal of this task is to dynamically alter the background color when the user presses keys R, G, or B which must change the background color of the display window to red, green, or blue, respectively.

1. Modify class `GLApp` class in header file `glapp.h` to integrate `GLboolean` key states for keys R, G, and B:

```
1 struct GLApp {
2     // other stuff ...
3     static GLboolean keystateR;
4     static GLboolean keystateG;
5     static GLboolean keystateB;
6 };
```

2. Do the following in source file `glapp.cpp`.
 - Begin by defining and initializing these static variables to `GL_FALSE`.
 - In function `GLApp::key_cb`, when key R is pressed, assign value `GL_TRUE` to static variable `keystateR`, and set the other two static variables to `GL_FALSE`. Implement similar logic when either key G or key B is pressed.
 - In function `GLApp::update`, adjust values in array `clear_color` based on the states of static variables `keystateR`, `keystateG`, and `keystateB` [as shown in the following pseudocode]:

```
1 void GLApp::update() {
2     // other stuff ...
3     if (GLApp::keystateR == GL_TRUE) {
4         // Set Red background
5     } else if (GLApp::keystateG == GL_TRUE) {
6         // Set Green background
7     } else if (GLApp::keystateB == GL_TRUE) {
8         // Set Blue background
9     }
10 }
```

3. Test your changes. Pressing key R or G or B should alter background color to the corresponding color [see sample executable for intended effect].

Submission Guidelines

1. Source and header files for **tutorial-0** must be placed in a directory labeled as: `<login>-<tutorial-0>`. If your Moodle student login is `foo`, then the directory would be labeled as `foo-tutorial-0` and would have the following structure and layout:

```

1 | 📁 foo-tutorial-0 # 🖋 You're submitting Tutorial 0
2 | └─ 📁 include    # 📄 Header files - *.hpp and *.h files
3 |   └─ 📁 src      # ⚡ Source files - *.cpp and .c files

```

2. Zip the directory and upload the resultant file **foo-tutorial-0.zip** to the assessment's submission page.

⚠ Before Submission: Verify and Test it ⚠

1. Unzip your archive file **foo-tutorial-0.zip** in directory **test-submissions** directory. Your directory and file layout should look like this:

```

1 | 📁 icg-opengl-dev # 🏠 Sandbox directory for all assessments
2 | └─ 📁 test-submissions # ⚠ Test submissions here before uploading
3 |   └─ 📁 foo-tutorial-0 # 🖋 Submitting Tutorial 0 with login foo
4 |     └─ 📁 include    # 📄 Header files - *.hpp and *.h files
5 |       └─ 📁 src      # ⚡ Source files - *.cpp and .c files
6 | └─ 📄 icg.bat        # 🖨 Automation Script

```

2. Select option **R** to reconfigure the solution with new project **foo-tutorial-0**.
3. Select option **B** to build the project. If there are no errors, an executable file **foo-tutorial-0.exe** will be created in directory **build/Release**. Alternatively, you can verify the project through the Visual Studio 2022 IDE.
4. Use the following checklist before uploading to the assessment's submission page:



Things to test before submission	Status
Assessment compiles without any errors	<input type="checkbox"/>
All warnings resolved, zero warnings during compilation	<input type="checkbox"/>
Executable generated and successfully launched in Debug and Release mode	<input type="checkbox"/>
Directory is zipped, ensuring adherence to naming conventions as outlined in submission guidelines	<input type="checkbox"/>
Upload zipped file to appropriate submission page	<input type="checkbox"/>

i *The purpose of this verification step is not to guarantee the correctness of your submission. Instead, it verifies whether your submission meets the most basic rubrics [it compiles, it doesn't generate warnings, it links, it executes] or not and thus helps you avoid major grade deductions. And, remember you'll need every single point from these programming assessments to ensure a noteworthy grade!!! Read the section on [Grading Rubrics](#) for information on how your submission will be assigned grades.*

Grading Rubrics

Competency **core1** simply says that if your source code doesn't compile nor link nor execute, there is nothing to grade. **core2** tests very basic competency related to setting up a framework to develop interactive computer graphics applications that are C/C++ and OpenGL based on Microsoft Windows platform using Visual Studio 2022 IDE.

- [**core1**] Submitted code must build an executable file with **zero** warnings. This rubric is not satisfied if the generated executable crashes or displays nothing.

[ **WARNING** ] *Prior to submission, verify your code using sandbox script [icg.bat](#) to ensure your submission neither crashes nor displays nothing. If the submitted code fails to meet the requirements, it indicates that you have failed to perform this verification step.*

- [**core2**] Completion of **Task 1** demonstrating an understanding of setting up OpenGL with GLFW framework and handling key events to dynamically update background color of display window.

Mapping of Grading Rubrics to Letter Grades

The core competencies listed in the grading rubrics will be mapped to letter grades using the following table:

Grading Rubric Assessment	Letter Grade
There is no submission.	<i>F</i>
core1 rubric is not satisfied. Submitted source code doesn't build. Or, executable generated by build crashes or displays nothing.	<i>F</i>
core2 rubric is satisfied.	<i>A+</i>