

Tutorial: Create 2D Game Engine using C++

URL: <https://www.youtube.com/playlist?list=PL-K0viiuJ2RctP5nJlqmHGeh66-GOZR>

Purpose

These notes summarize the you tube video playlist “How to Make a 2D Game Engine with C++ & SDL”. We will try to follow the videos in order but will not hesitate to re-arrange the material to make it easier to read and follow and understand the concepts presented. I will make corrections to the code and provide additional material (without attribution) from the Internet. These are my notes in trying to follow and understand the videos.

I was going to use the latest version of SDL – SDL3 but found too many differences between SDL2 and SDL3 so decided to stick to the older version.

Background

I am quite rusty with C++ and never worked with SDL2 library (I did start to write notes many years ago on SDL). I will describe things in a little bit more detail than what is covered in the video tutorial since I am using this tutorial to relearn C++ and SDL2.

1. Setup SDL, SDL_image, SDL_ttf in Code::Blocks

The first video walks you through obtaining and installing the application Code::Blocks. Code::Blocks will be used to create our C++ source files and compile and run our application.

In addition this video creates the initial project to be used throughout the series, which of course starts as a “Hello, World” project. Lastly, we install the two main SDL libraries – SDL2 and SDL_image.

Why use Code::Blocks?

Code::Blocks is a free open-source, cross-platform Integrated Development Environment (IDE) primarily designed for C, C++ and Fortran programming. The IDE was originally released in 2005, and its development is driven by a team of contributors using wxWidgets library for its graphical user interface (GUI). The first official stable version was 8.02 released in 2008. The release marks the year and month of the Code::Blocks version. Like most modern IDEs it provides for extensions via a plugin system.

| |
|--------------------|
| What is wxWidgets? |
|--------------------|

wxWidgets is a free and open-source C++ library used for creating cross-platform graphical user interfaces (GUIs). It allows developers to write applications that can run on multiple operating systems, such as Windows, macOS, Linux, and more, without needing significant code changes. This is achieved by using the native API of each platform, ensuring that applications have a native look and feel.

Originally created in 1992 by Julian Smart, wxWidgets was initially called wxWindows but was renamed in 2004 due to a trademark issue. It supports a wide range of features, including GUI components, event handling, multithreading, file handling, and more. Additionally, it has bindings for other programming languages like Python, Perl, and C#.

Developers often use wxWidgets for its portability and ability to create sophisticated applications with minimal effort. Let me know if you'd like to explore its features further!

Install Code::Blocks

The official website to obtain Code::Blocks is <https://www.codeblocks.org/downloads/>. You have the option of:

- Downloading the setup file for Code::Blocks and executing it.
- Download the source code and build from scratch

The Binary release has a Windows, Linux and Mac OS X version. The fact that there is a version of Code::Blocks that runs on all three platforms is the primary reason it has been selected for this and other tutorials. At the time of this writing the latest version is 25.03, which means it was released in March of 2025.

My choices for Windows are:


|  Microsoft Windows (64 bit, default) | |
|---|---------------------------------|
| File | Download from |
| codeblocks-25.03-setup.exe | Sourceforge.net |
| codeblocks-25.03-setup-nonadmin.exe | Sourceforge.net |
| codeblocks-25.03-nosetup.zip | Sourceforge.net |
| codeblocks-25.03mingw-setup.exe | Sourceforge.net |
| codeblocks-25.03mingw-nosetup.zip | Sourceforge.net |

Figure 1 - Selecting the right setup.exe to download

I decided to download codeblock-25.03mingw-setup.exe for 64-bit.

Tutorial: Create 2D Game Engine using C++

Side note: I am surprised that the project has not moved to GitHub.

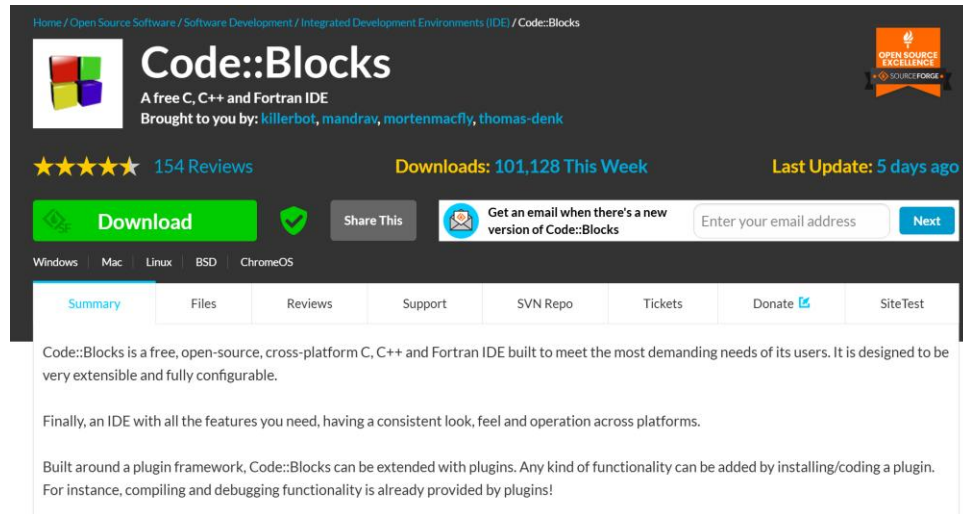


Figure 2 - Downloading file from Sourceforge

I then run the setup.exe file:

| Name | Date modified | Type |
|---------------------------------|-------------------|-------------|
| codeblocks-25.03mingw-setup.exe | 4/15/2025 6:26 PM | Application |

Figure 3 - Locating and executing the setup.exe file

Running the Setup

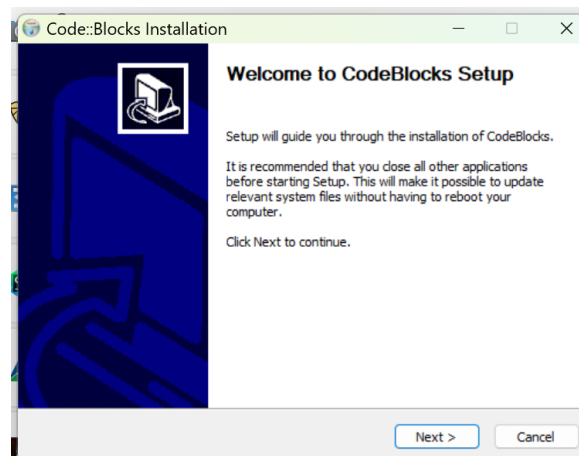


Figure 4 - Initial Code::Blocks Installation dialog

I did not find it necessary to close any other applications. I just pressed "Next >".

Tutorial: Create 2D Game Engine using C++

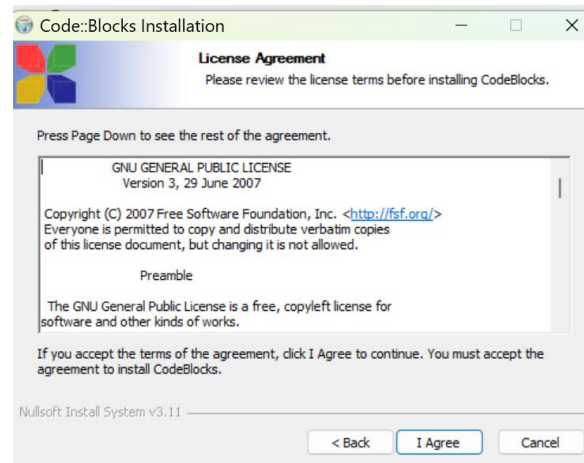


Figure 5 - The License Agreement screen

OK, I did not read the license agreement but knowing it is the GNU license assures me that it is open source and free. I just clicked on “I Agree”.

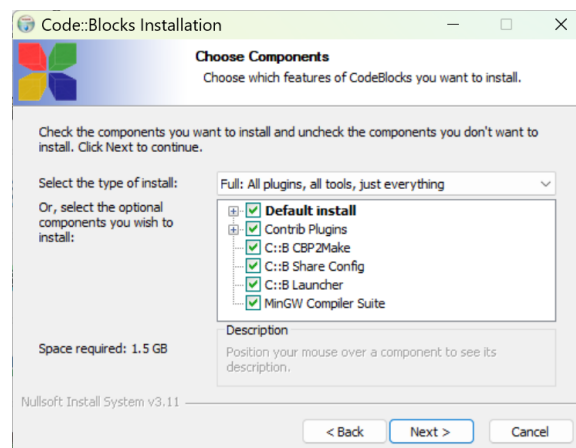


Figure 6 - The "Choose Components" screen

It appears by default all the components are selected. A quick description of each component:

- **Contrib Plugins** – these are additional plugins developed by the community to extend the functionality of the Code::Blocks IDE. These plugins are not part of the core set but have proven to be valuable enough to be included in the official Code::Blocks repository.
 - **Code Snippets Plugin:** Helps to manage and insert reusable code snippets.
 - **DoxyBlocks Plugin:** Integrates Doxygen for generating documentation from your code
 - **CppCheck Plugin:** Provides static code analysis to identify potential bugs or issues
 - **SpellChecker Plugin:** Checks spelling in comments and string literals
 - **Valgrind Plugin:** Integrates Valgrind for memory debugging and profiling

Tutorial: Create 2D Game Engine using C++

- **C::B CBP2Make** – is a tool designed to generate Makefiles from Code::Blocks project files (*.cbp) or workspace files. Essentially, it allows you to convert your Code::Blocks projects into Makefiles that can be used with GNU Make or other build systems.
- **C::B Share Config** – this is a tool that allows you to import and export parts of your Code::Blocks configuration. It's particularly useful when you want to transfer settings between different computers or configurations.
- **C::B Launcher** – is a utility that helps manage the launching of the Code::Blocks IDE. This tool is useful for advanced users.
- **MinGW Compiler Suite** – The MinGW (Minimalist GNU for Windows) is a development environment that provides a native Windows port of the GNU Compiler Collection (GCC). It allows you to build native Windows applications without relying on third-party runtime libraries.

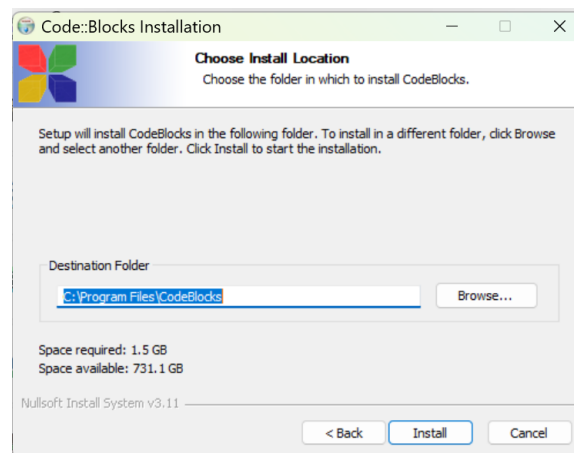


Figure 7 - Installation location on your PC

I usually take the default location.

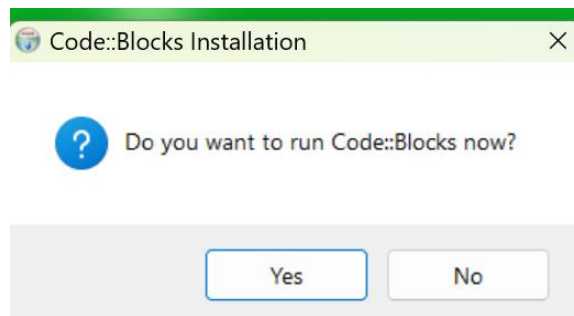


Figure 8 - Prompt to run Code::Blocks now?

I clicked on "Yes" so I can enter a simple "hello world" program to make sure everything works.

Tutorial: Create 2D Game Engine using C++

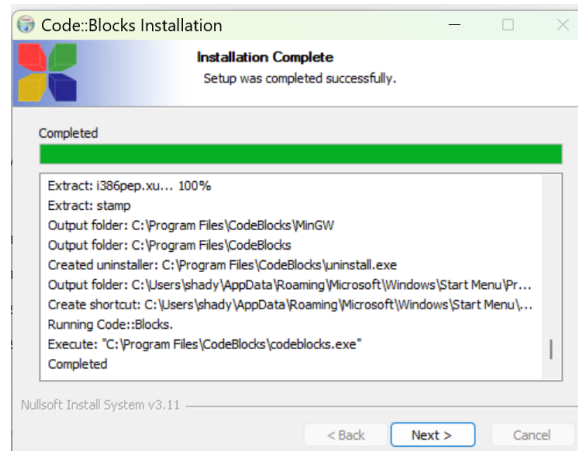


Figure 9 - Code::Blocks installation completed dialog

I clicked on “Next >”.

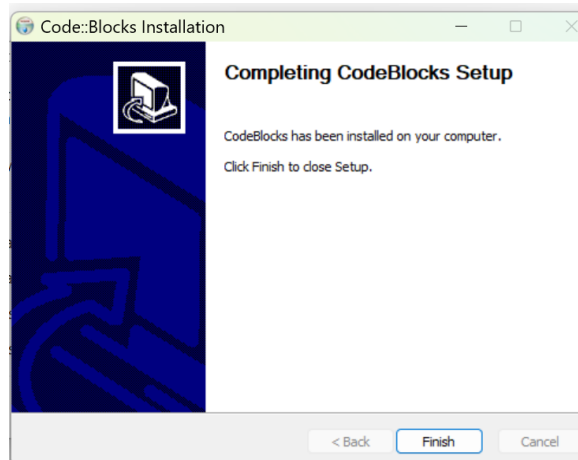


Figure 10 - The final Code::Blocks setup screen

Click on “Finish” and you will see Code::Blocks starting if you clicked “Yes” to start Code::Blocks now.

Tutorial: Create 2D Game Engine using C++

Starting up Code::Blocks

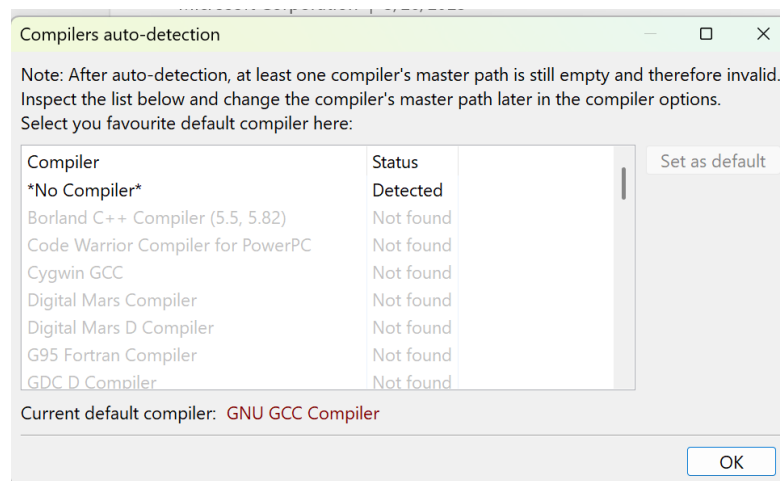


Figure 11 - Code::Blocks lists all the compilers it detected

As you can see from the list you have the option of using many different compilers in Code::Blocks. We do expect Code::Blocks to find MinGW compiler because that is the version we downloaded.

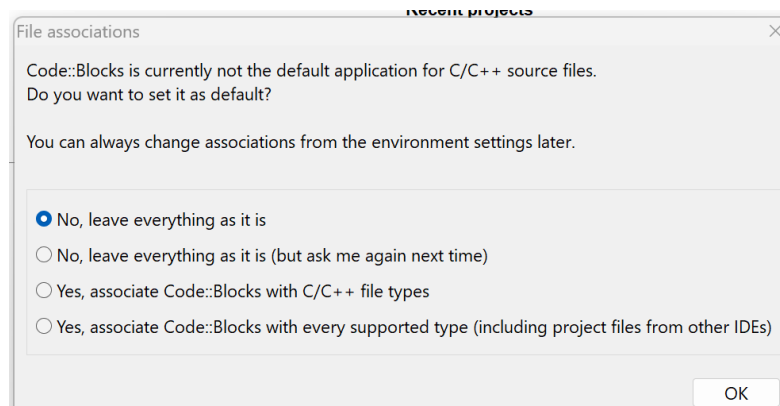


Figure 12 - Option to update file association

I opted for the default of “No, leave everything as it is” since I do use Visual Studio 2022 for other projects.

Setting Code::Blocks to use K&R Style braces

When Code::Blocks creates the initial code for you it does not use K&R brace style (also known as Kernighan and Ritchie style), you can adjust the formatting settings in the editor since a. I prefer K&R and b. the video presenter uses that style.

- Go to Settings ➔ Editor

Tutorial: Create 2D Game Engine using C++

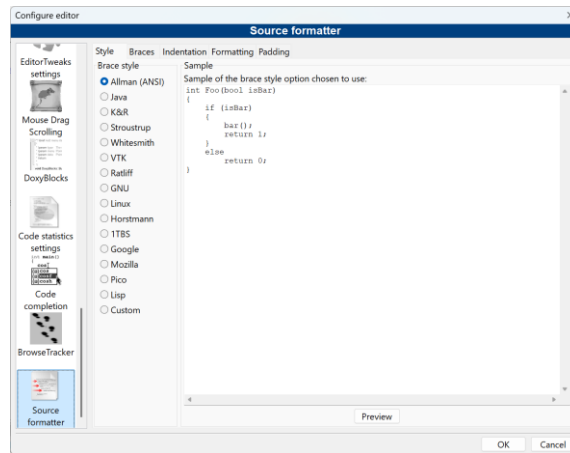


Figure 13 - Source formatter

- Select K&R

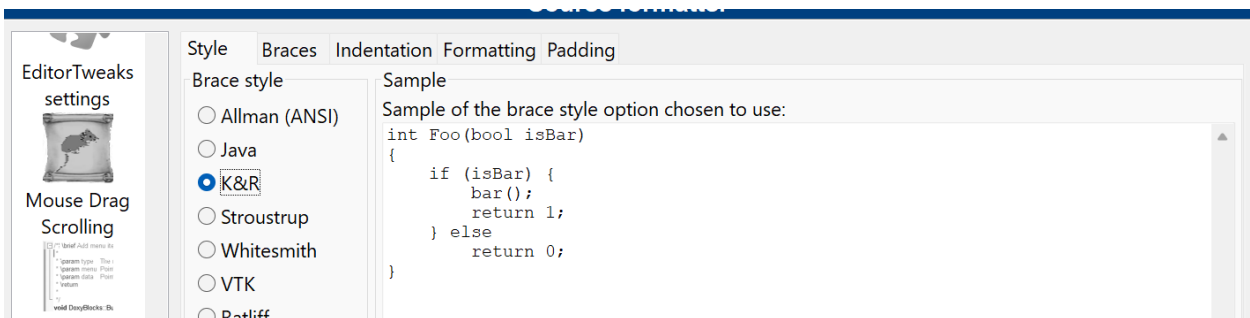


Figure 14 - Using K&R

I personally like Java style because I have mainly been a Java programmer for over 20 years (I know, I know, say no more!)

Note: I did not do this until after the end of video #3.

Tutorial: Create 2D Game Engine using C++

Starting a Project with Code::Blocks

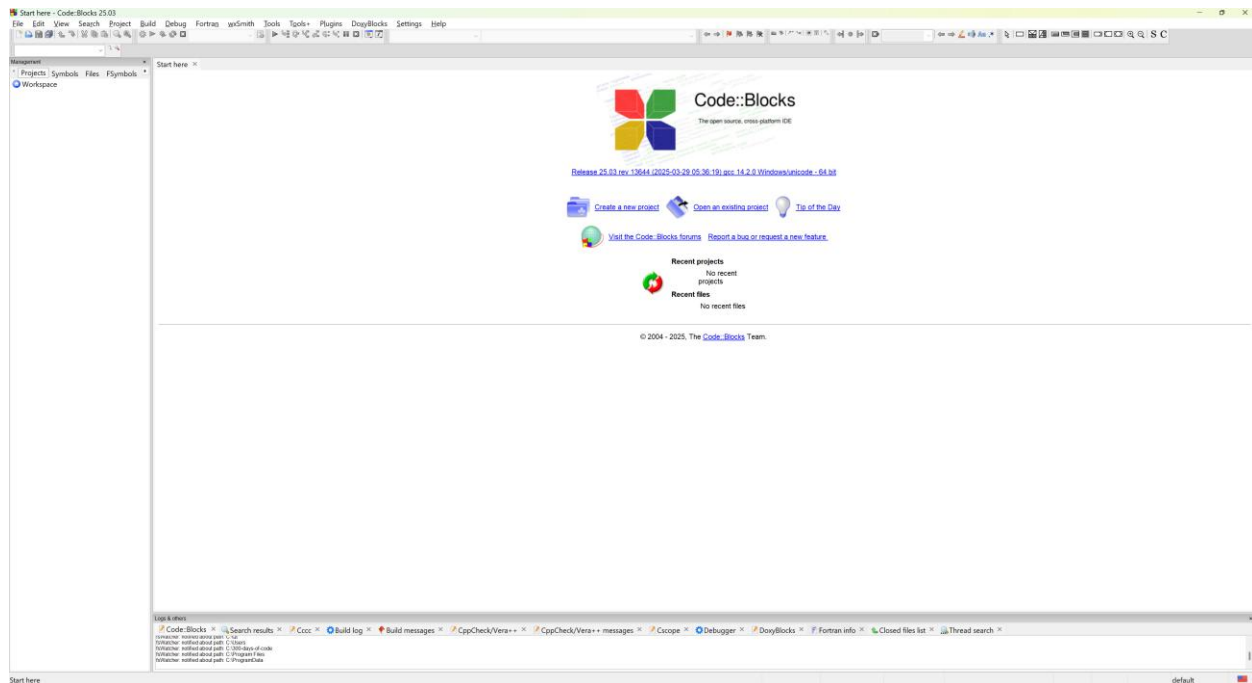


Figure 15 - The starting Code::Blocks screen

Note: I downloaded the Code : :Blocks manual but the screenshots of the application appear to be dated!

Making Adjustments to the screen

Another note: The screen icons and fonts appear to be small to me (it is probably due to my screen resolution). I made things larger by doing the following:

- Go to Settings ➔ Environment ➔ View
- Increase Message logs' font size the Toolbar icon size

Tutorial: Create 2D Game Engine using C++

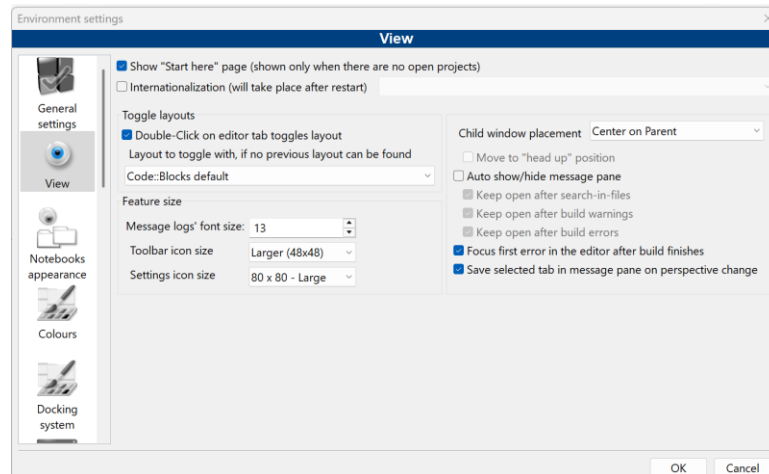


Figure 16 - Increasing the toolbar icon sizes

- Click “OK”
- Select File → Save Everything
- Restart Code::Blocks



I also increased the font size used by:

- Select Settings → Editor

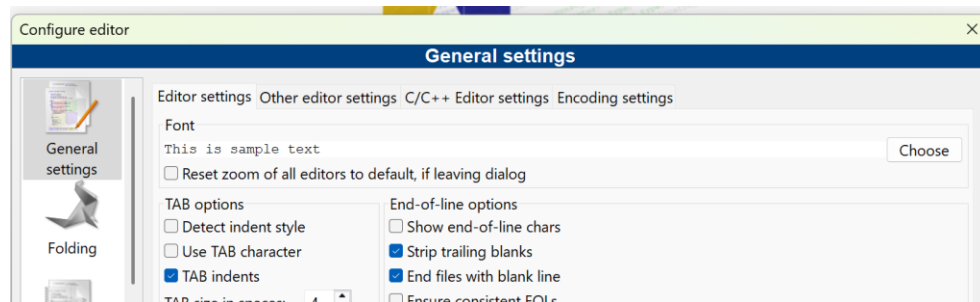


Figure 17 - Updating font-size

- Click on “Choose” button

Tutorial: Create 2D Game Engine using C++

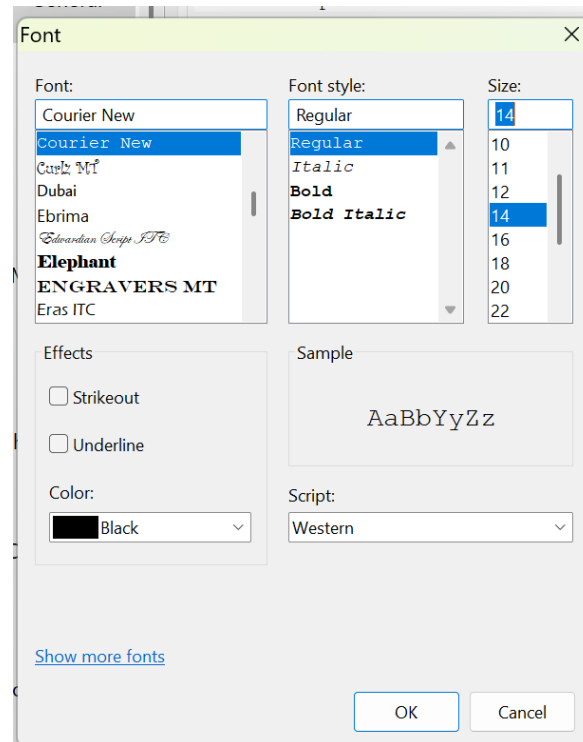


Figure 18 - Select desired font size

- I selected 14 and clicked on “OK”

Creating the Project

- Click on “Create a new project” link



[Create a new project](#)

Figure 19 - Creating a new Project

- Select “Console Application”

Tutorial: Create 2D Game Engine using C++

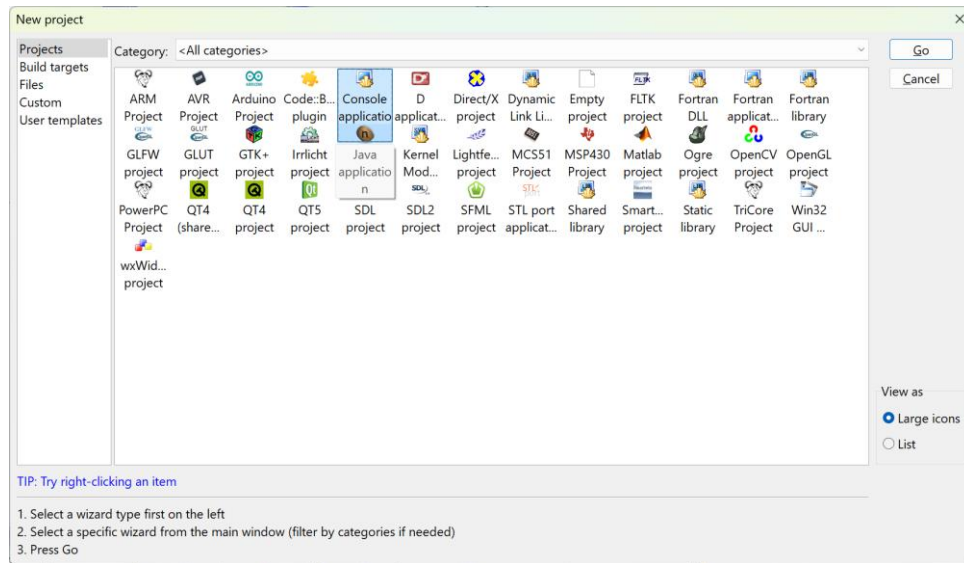


Figure 20 - Creating our first console application

- Click on “Go”, if this is your first time you will see the following dialog box appear:

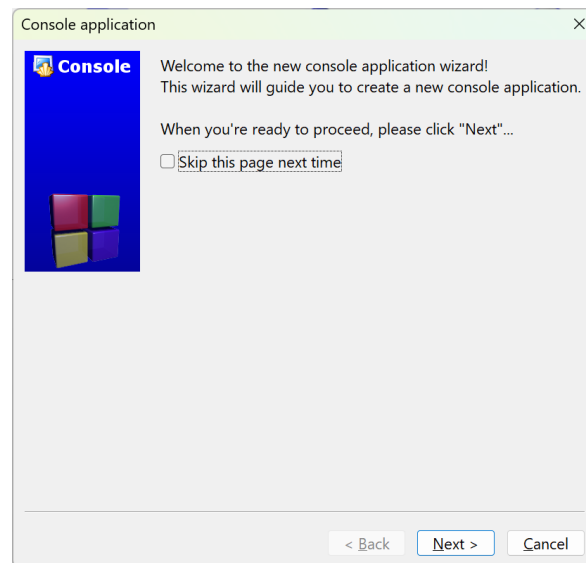
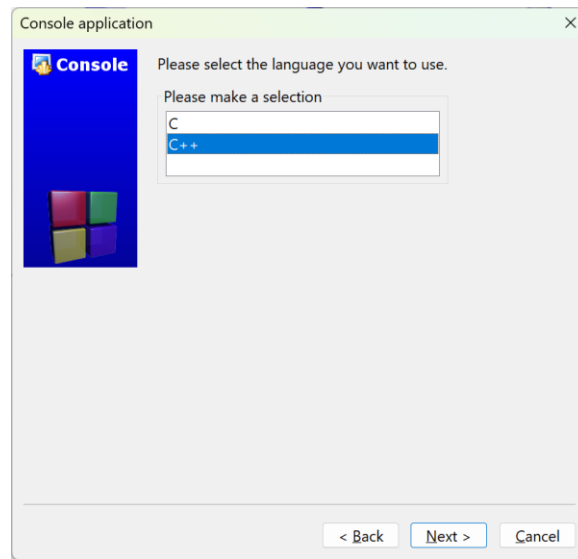


Figure 21 - The Console application wizard starting...

- Click on “Skip this page next time” and press on “Next >”

Tutorial: Create 2D Game Engine using C++



- Take the default C++ and click on "Next >"
- Fill in the Project information:

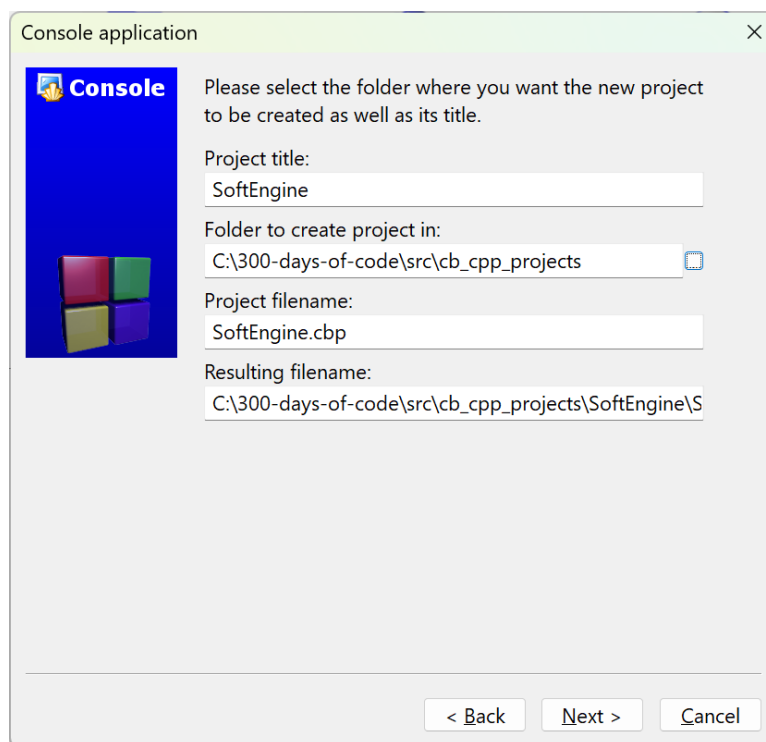


Figure 22 - Project information

The Project name will be SoftEngine. Since this is part of my 300-days-of-code effort I select a folder where I plan on placing all my Code :: Blocks C++ projects. Enter a location that makes sense for your setup.

Tutorial: Create 2D Game Engine using C++

- Click on “Next >”

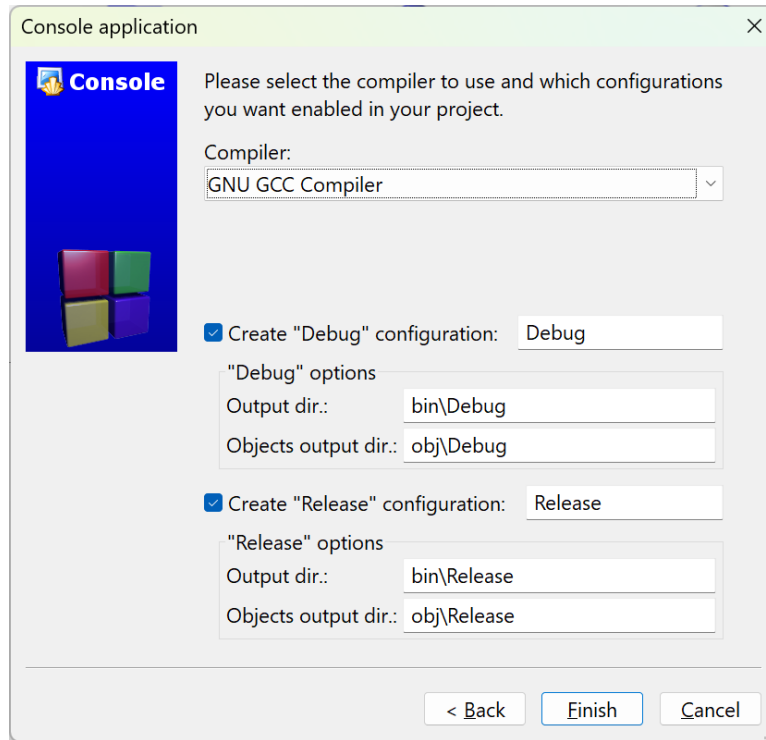


Figure 23 - Final step in creating our game engine project

- Make sure the “Debug” and “Release” configuration are selected and click on “Finish”

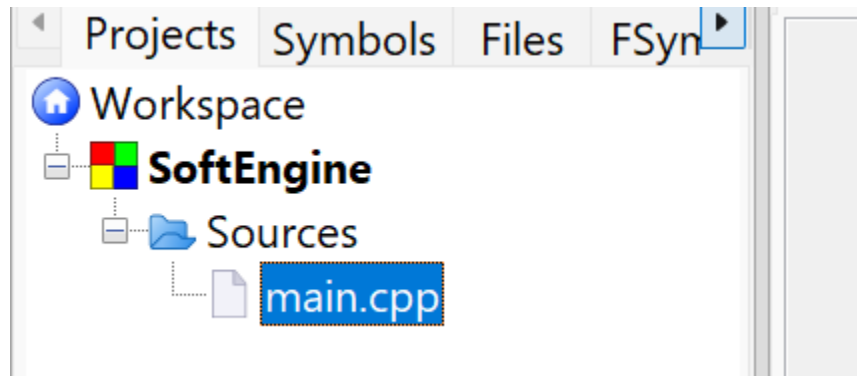


Figure 24 - Default main.cpp is created

- Following the video author’s convention, let’s rename the file main.cpp to Main.cpp
 - Right-click on the filename and select Rename file:

Tutorial: Create 2D Game Engine using C++

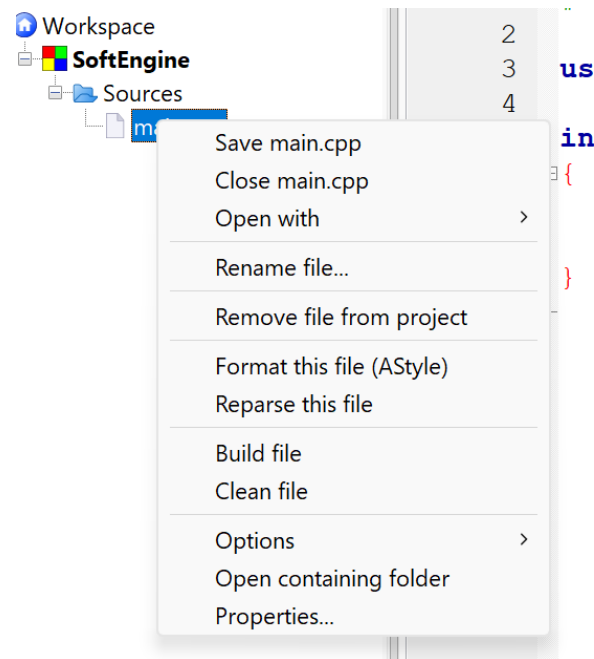


Figure 25 - Context menu for file

- Enter the name `Main.cpp`:

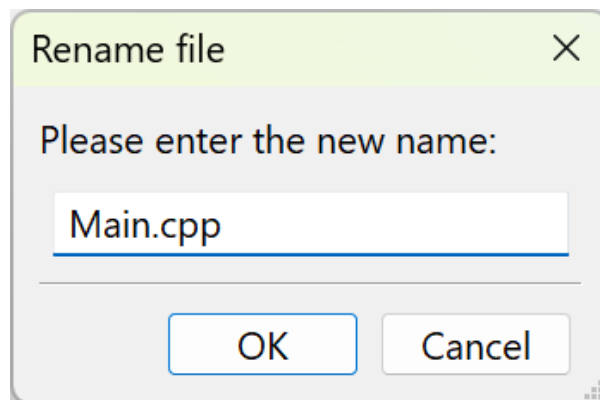


Figure 26 - Rename dialog box

- Click "OK"

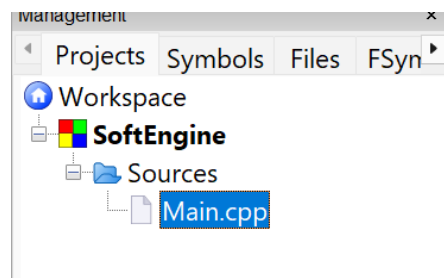


Figure 27 - Update name in Workspace

Build and Run the Program

The **Workspace** contains one or more projects, in our case it shall only contain one project – SoftEngine. The workspace is the most top-level container. A **project** contains one or more build targets and the project's files.

You should familiarize yourself with the following icons/operations:

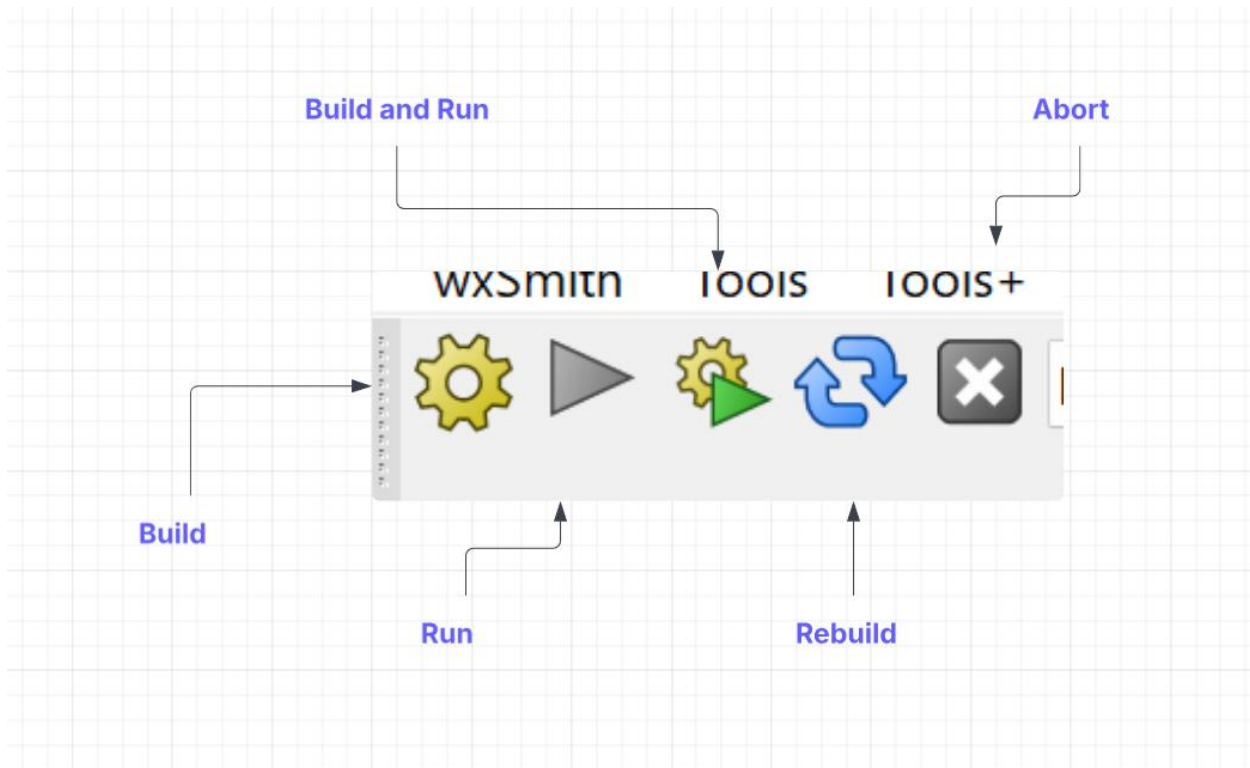


Figure 28 - Common operations

- **Build** – this function compiles your source code into an executable program. It processes the code files in your project, checks for errors, and links them to create a standalone application. This is where your written code is transformed into something that can run on your machine.
- **Run** – this function executes the program that you've built. If the program is a console application (which is true in our case), it will open a terminal or console window to display the output.
- **Build and Run** – it combines the two steps above
- **Rebuild** – This is a more thorough version of Build. While "Build" compiles *only* the files that have been modified since the last build, "Rebuild" forces the *entire* project to be recompiled from scratch, regardless of whether files have changed.
- **Abort** – this command is used to stop an ongoing build or compilation process. It is specifically for interrupting the build process.

Tutorial: Create 2D Game Engine using C++

Let's Build and run!

Our Main.cpp code is:

```
1. #include <iostream>
2.
3. using namespace std;
4.
5. int main()
6. {
7.     cout << "Hello world!" << endl;
8.     return 0;
9. }
10.
```

The code above is our simple “Hello, world!” program.

- Click on the “Build and Run” icon

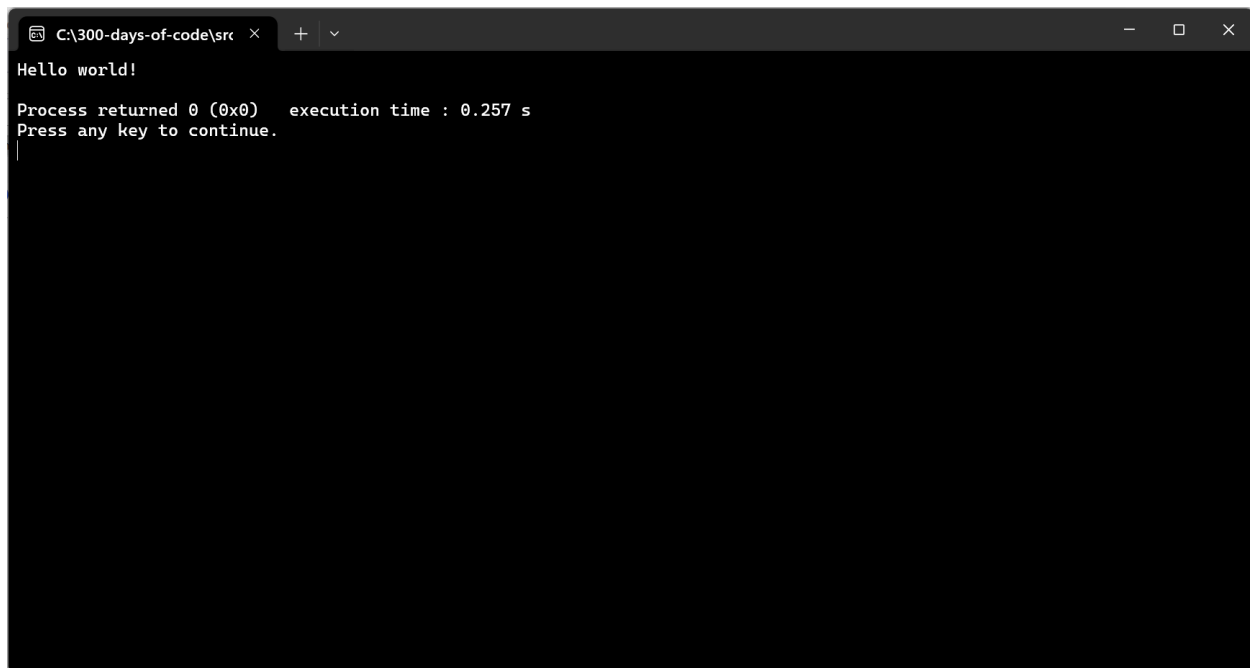


Figure 29 - Result of building and running our Main.cpp

- Press any key on the keyboard to dismiss

Install and Setup SDL

What is SDL?

SDL, or **Simple DirectMedia Layer**, is a cross-platform software development library designed to provide a hardware abstraction layer for multimedia hardware components. It was originally

created by Sam Lantinga in 1998. SDL is widely used for developing high-performance computer games and multimedia applications across various operating systems, including Android, iOS, Linux, macOS, and Windows.

The library is written in C and provides an application programming interface (API) in C, with bindings available for other programming languages. Over the years, SDL has evolved significantly, with major updates like SDL 2.0 in 2013, which introduced better support for 3D hardware acceleration. SDL 3.0, released in January 2025, brought further enhancements and new features.

SDL is free and open-source software, licensed under the zlib License since version 2.0, allowing developers to use it in both open-source and closed-source projects. It has been extensively used in the industry, with hundreds of games, applications, and demos built using the library.

The website to obtain more information is: <https://www.libsdl.org/>

Installing SDL2 and SDL2_image

- Create a folder that will hold both SDL2 and SDL2_image download files. I will create the folder D:\SDL2_dev folder.
- Download the latest SDL2
 - Go to: <https://github.com/libsdl-org/SDL>

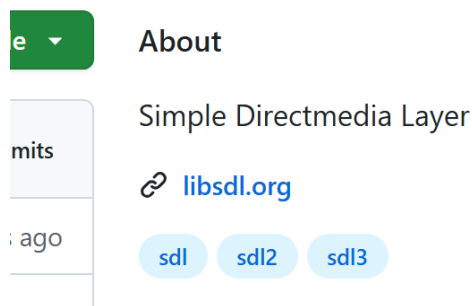


Figure 30 - Versions of sdl on Github

The sdl2 link displays a list of projects that uses sdl2.

- Go to <https://github.com/libsdl-org/SDL/releases/tag/release-2.32.4> to get the latest SDL2 version.

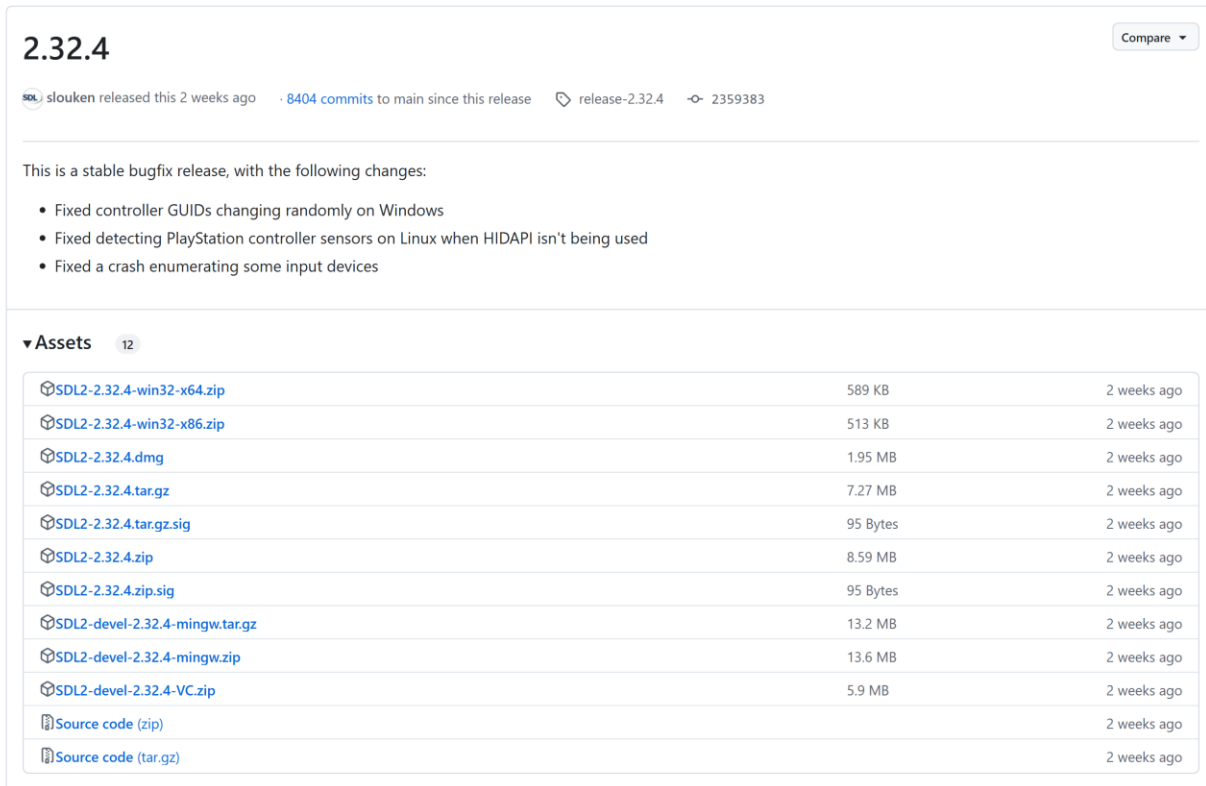
Note: We will be using the latest version of SDL – SDL2.32.4. The video series uses SDL2-2.0.10.

Another note: I prefer to learn and use SDL3 but there are too many differences between SDL2 and SDL3 that will introduce another set of issues that will get in the way of the video series!

Tutorial: Create 2D Game Engine using C++

- Navigate to the release github url (for me it is: <https://github.com/libsdl-org/SDL/releases/tag/release-3.2.10>)

What version of SDL2 should I use?



2.32.4 Compare

slouken released this 2 weeks ago · 8404 commits to main since this release · release-2.32.4 · 2359383

This is a stable bugfix release, with the following changes:

- Fixed controller GUIDs changing randomly on Windows
- Fixed detecting PlayStation controller sensors on Linux when HIDAPI isn't being used
- Fixed a crash enumerating some input devices

▼ Assets 12

| | | |
|--------------------------------|----------|-------------|
| SDL2-2.32.4-win32-x64.zip | 589 KB | 2 weeks ago |
| SDL2-2.32.4-win32-x86.zip | 513 KB | 2 weeks ago |
| SDL2-2.32.4.dmg | 1.95 MB | 2 weeks ago |
| SDL2-2.32.4.tar.gz | 7.27 MB | 2 weeks ago |
| SDL2-2.32.4.tar.gz.sig | 95 Bytes | 2 weeks ago |
| SDL2-2.32.4.zip | 8.59 MB | 2 weeks ago |
| SDL2-2.32.4.zip.sig | 95 Bytes | 2 weeks ago |
| SDL2-devel-2.32.4-mingw.tar.gz | 13.2 MB | 2 weeks ago |
| SDL2-devel-2.32.4-mingw.zip | 13.6 MB | 2 weeks ago |
| SDL2-devel-2.32.4-VC.zip | 5.9 MB | 2 weeks ago |
| Source code (zip) | | 2 weeks ago |
| Source code (tar.gz) | | 2 weeks ago |

Figure 31 - Versions of SDL3 to choose from

Since I am using Code::Blocks with mingw I will utilize `SDL2-devel-2.32.4-mingw.zip` version. If you are using Visual Studio you should download and install `SDL2-devel-2.32.4-VC.zip`

Note: If you want to utilize SDL2 you can find the latest SDL2 release on the same github website: <https://github.com/libsdl-org/SDL/releases>

- Unzip the file¹ to `D:\SDL2_dev`:

¹ I use 7-zip to manage my zip files

Tutorial: Create 2D Game Engine using C++

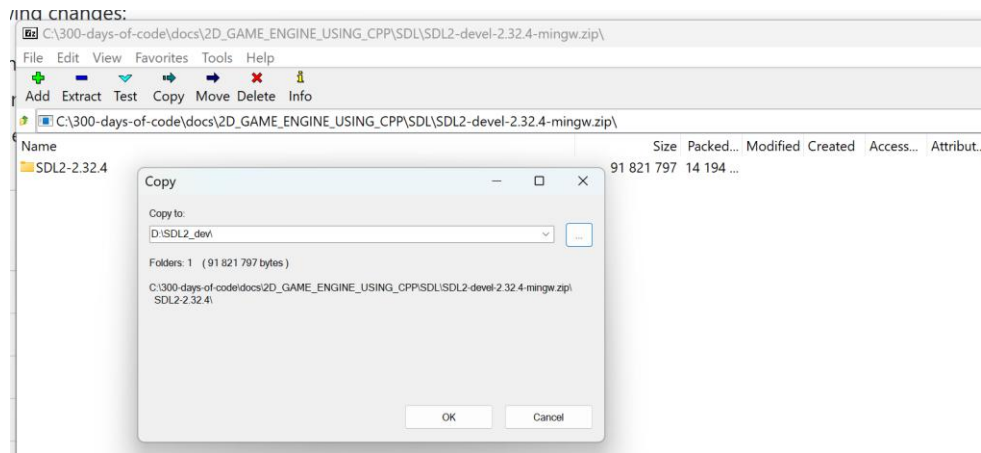


Figure 32 - Unzipping the SDL2 version

You will see:

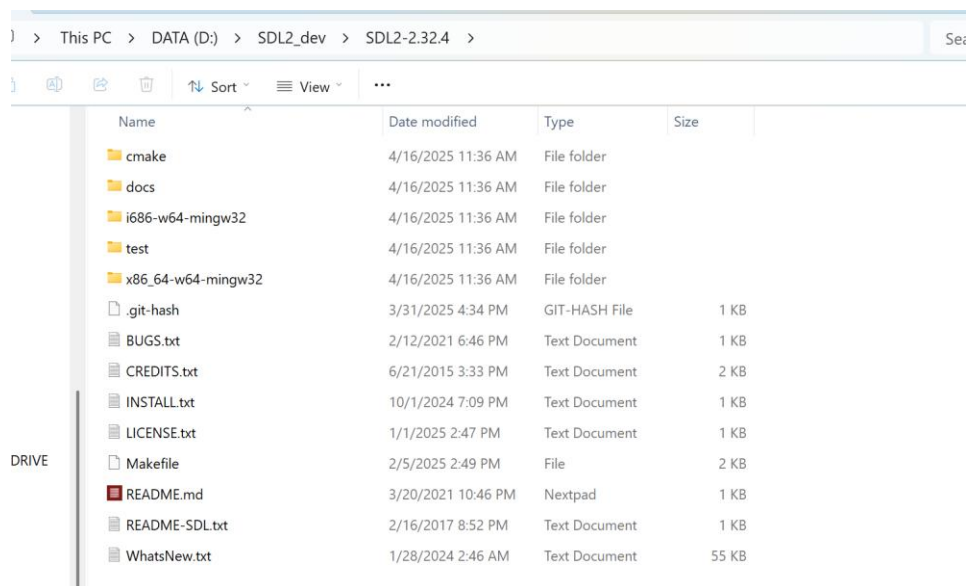


Figure 33 - Unzipped SDL2 file

- Open the INSTALL.md file to determine which of the folders you will use:

The 32-bit files are in i686-w64-mingw32
The 64-bit files are in x86_64-w64-mingw32

Figure 34 - Contents of INSTALL.txt file

Tutorial: Create 2D Game Engine using C++

I will be using the 64-bit architecture of the files in x86_64-w64-mingw32. Select the folder that makes sense for your machine and setup.

- Go to https://github.com/libsdl-org/SDL_image/releases to obtain the 2.8.9 release of SDL2_image

2.8.8

This is a stable bugfix release, with the following changes:

- Fixed alpha in less than 32-bit ICO and CUR images

▼Assets 12

| | | |
|-------------------------------------|----------|-------|
| SDL2_image-2.8.8-win32-x64.zip | 1.5 MB | Mar 3 |
| SDL2_image-2.8.8-win32-x86.zip | 1.16 MB | Mar 3 |
| SDL2_image-2.8.8.dmg | 1.48 MB | Mar 3 |
| SDL2_image-2.8.8.tar.gz | 6.56 MB | Mar 3 |
| SDL2_image-2.8.8.tar.gz.sig | 95 Bytes | Mar 3 |
| SDL2_image-2.8.8.zip | 10.1 MB | Mar 3 |
| SDL2_image-2.8.8.zip.sig | 95 Bytes | Mar 3 |
| SDL2_image-devel-2.8.8-mingw.tar.gz | 403 KB | Mar 3 |
| SDL2_image-devel-2.8.8-mingw.zip | 408 KB | Mar 3 |
| SDL2_image-devel-2.8.8-VC.zip | 4.01 MB | Mar 3 |
| Source code (zip) | | Mar 3 |
| Source code (tar.gz) | | Mar 3 |

Figure 35 - SDL2_image choices

- I downloaded SDL2_image-devel-2.8.8-mingw.zip to be consistent with the version of SDL2 I downloaded.
- Unzip in the D:\SDL2_dev folder

You will see the following two top-level folders in D:\SDL2_dev:

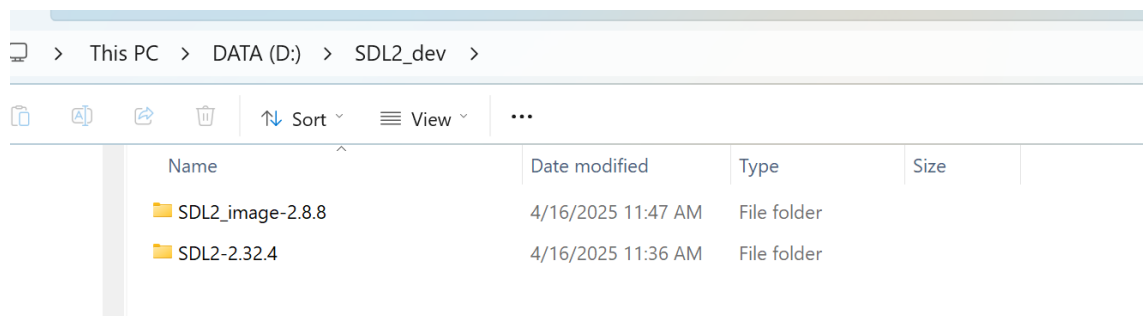


Figure 36 - Top-level folders

Tutorial: Create 2D Game Engine using C++

Note: Unlike the video – I will leave the files in the location I unzipped them.

Setting up Code::Blocks to access SDL folders

- Open the project (if not open) we created SoftEngine. To re-open the project
 - Open Code::Blocks
 - Click on “Open an existing project”
 - Navigate to the folder you placed your project and select SoftEngine.cbp and click “Open”

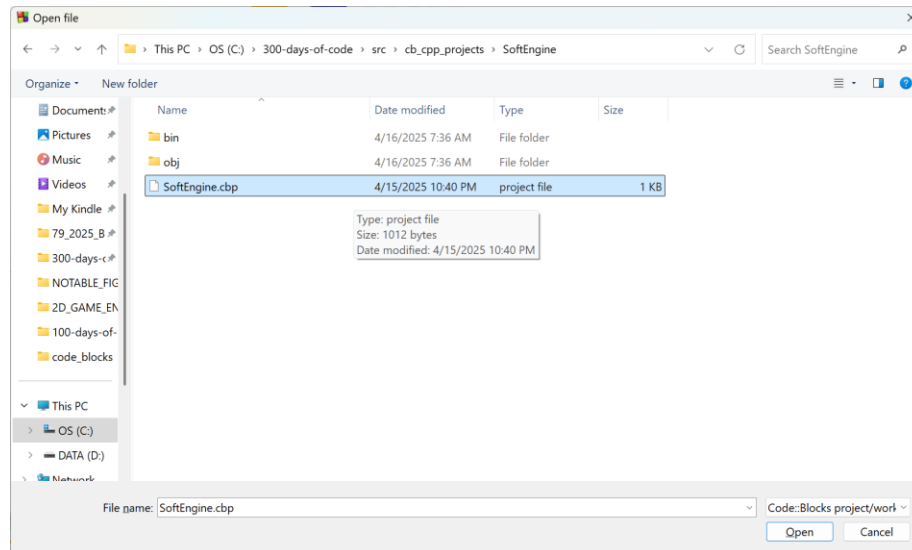


Figure 37 - Opening an existing project

- Go to Settings → Compiler...

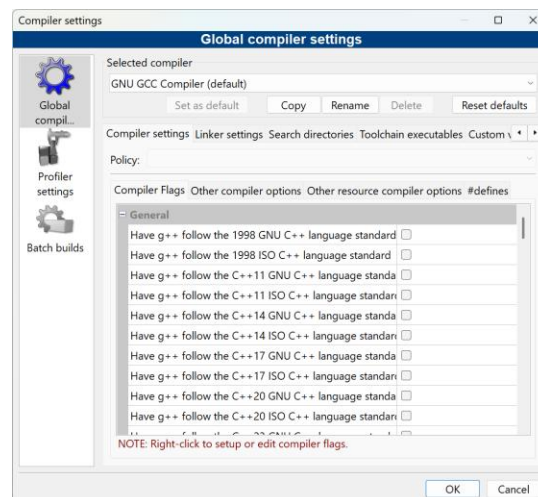


Figure 38 - Global compiler settings

Tutorial: Create 2D Game Engine using C++

- Click on the “Linker settings” tab

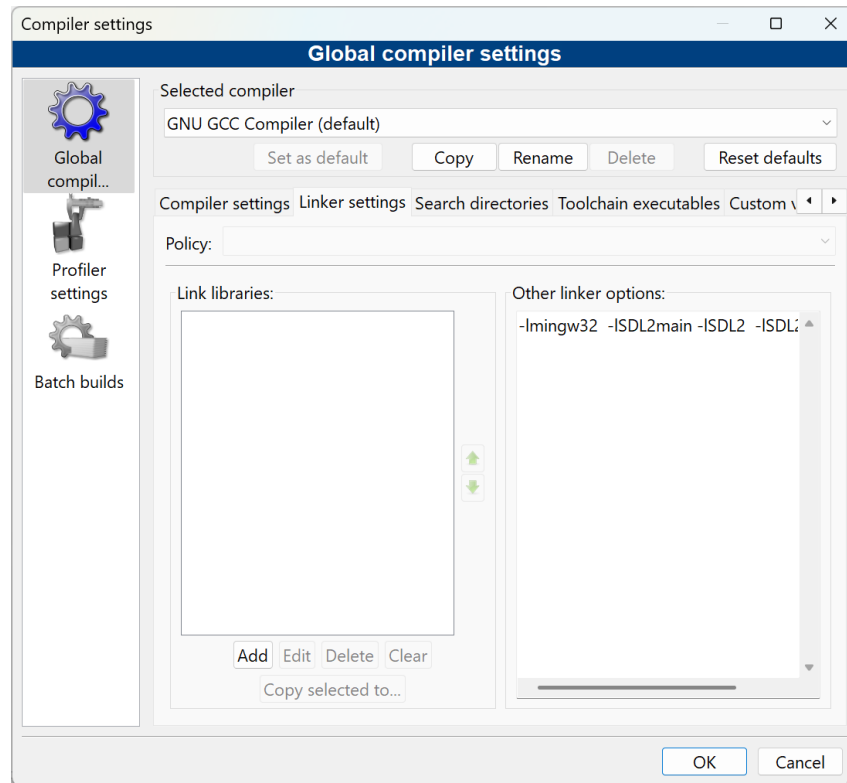
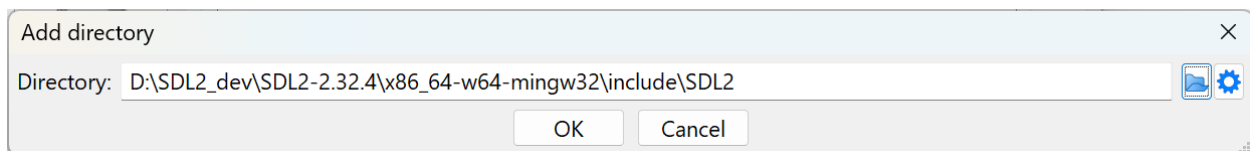


Figure 39 - Adding linker options

- In the “Other linker options:” input box enter:
`-lmingw32 -lSDL2main -lSDL2 -lSDL2_image`
- Click on the “Search directories” tab
- Make sure the “Compiler” tab is selected
- Click on “Add” and navigate to the include folder for SDL



Tutorial: Create 2D Game Engine using C++

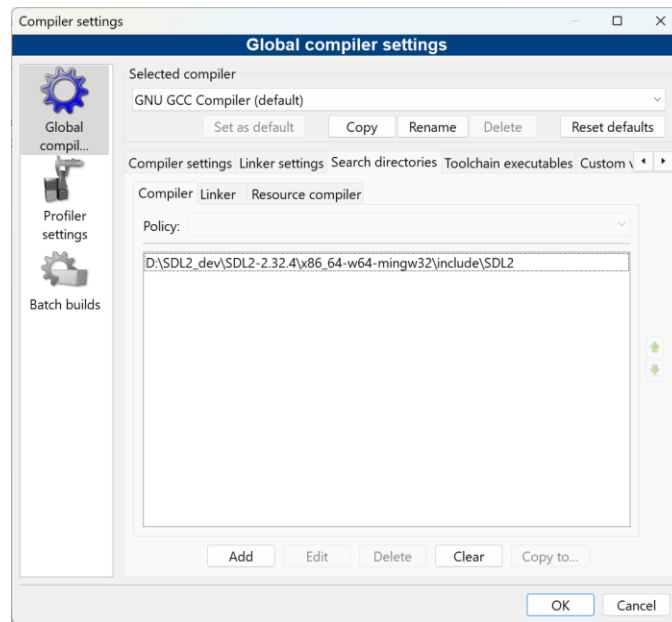
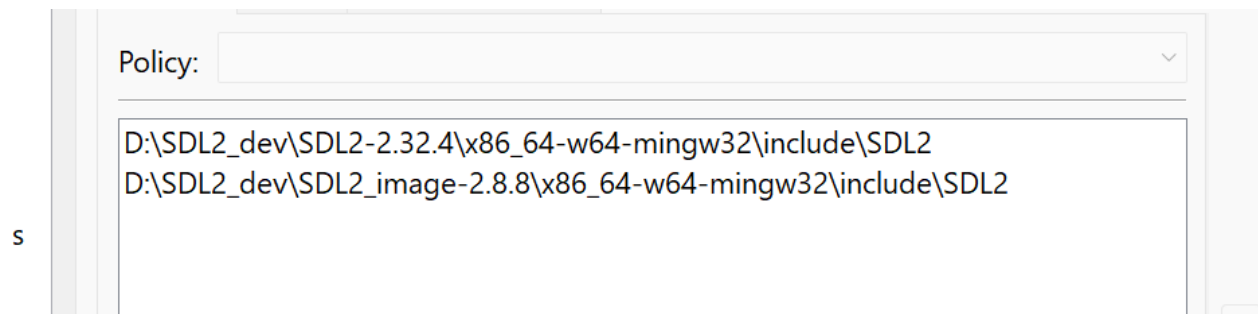


Figure 40 - Adding SDL3 include folder for the compiler

- Add the SDL2_image include as well



- Select the “Linker” sub-tab and add the location of the lib folder:

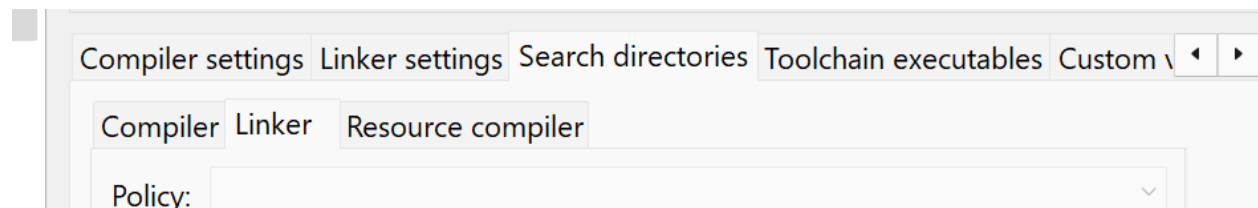
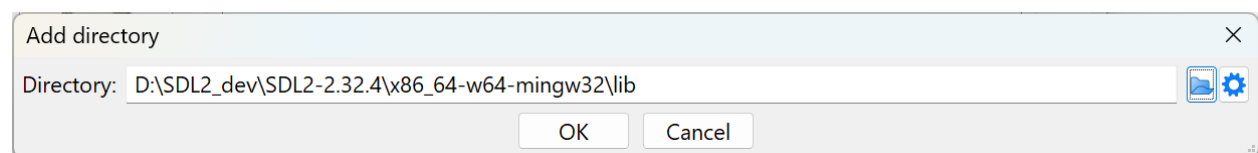


Figure 41 - Linker sub-tab

- Navigate to the SDL2 lib folder



Tutorial: Create 2D Game Engine using C++

- Click on OK
- Add lib folder under SDL_Image directory

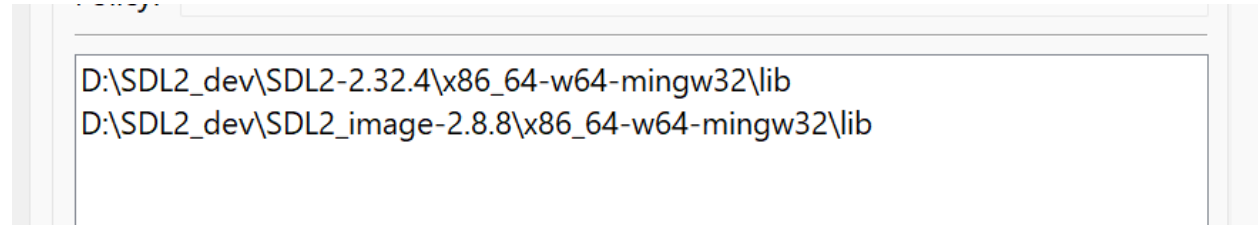


Figure 42 - SDL2 and SDL2_image lib folders for the linker

Testing the setup

- Update the code as follows:

```
1. #include <iostream>
2. #include "SDL.h"
3.
4. using namespace std;
5.
6. int main(int argc, char** argv)
7. {
8.     cout << "Hello world!" << endl;
9.     return 0;
10. }
11.
```

- Click on the “Build” or “Rebuild”



You should see no errors.

The video uses `SDL_INIT Everything` to initialize all the SDL subsystems at the same time. This is considered a bad practice and the flag `SDL_INIT Everything` no longer exists in SDL3. We will follow the video but note that it does not exist in SDL3.

- Let’s initialize SDL to make sure everything runs correctly. Update the Main.cpp:

```
1. #include <iostream>
2. #include "SDL.h"
3.
4. using namespace std;
```

Tutorial: Create 2D Game Engine using C++

```
5.  
6. int main(int argc, char** argv)  
7. {  
8.     if (SDL_Init(SDL_INIT_EVERYTHING) == 0) {  
9.         cout << "SDL_Init worked!" << endl;  
10.    }  
11.    SDL_Quit(); // clean up resources  
12.    return 0;  
13. }
```

- Try to build it again. It should build.
- Now, try to run the application. It fails:

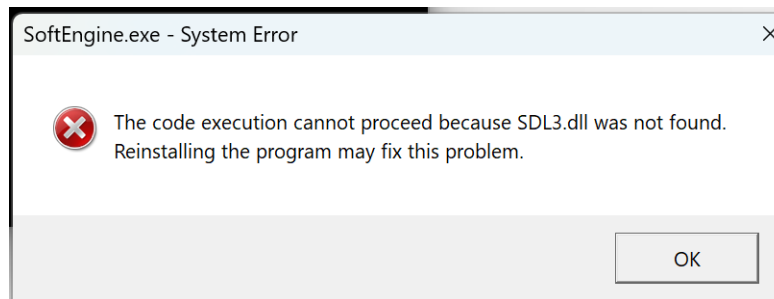


Figure 43 - Application failed to find SDL3.dll file

The problem is that the application could not find the SDL2.dll file that is located in the SDL2 \bin folder:

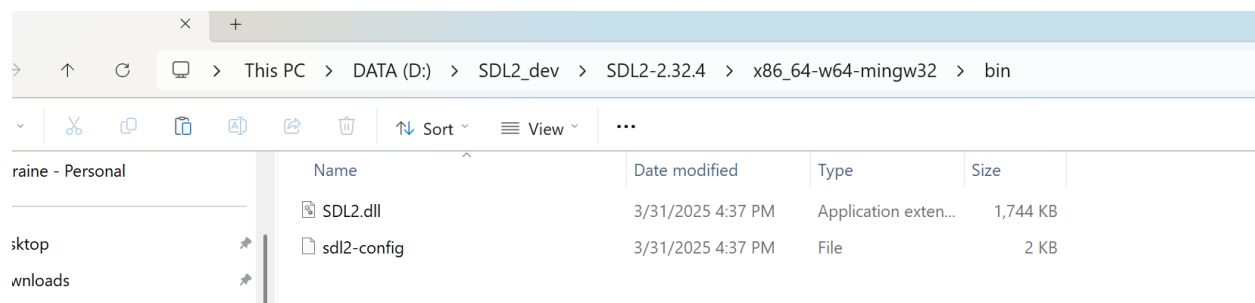
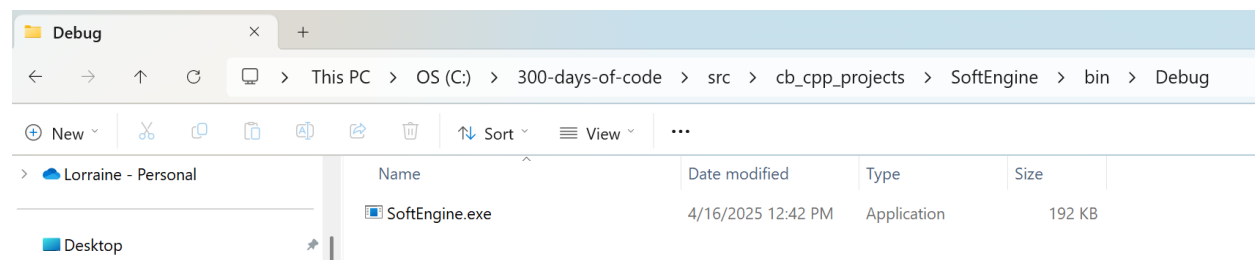


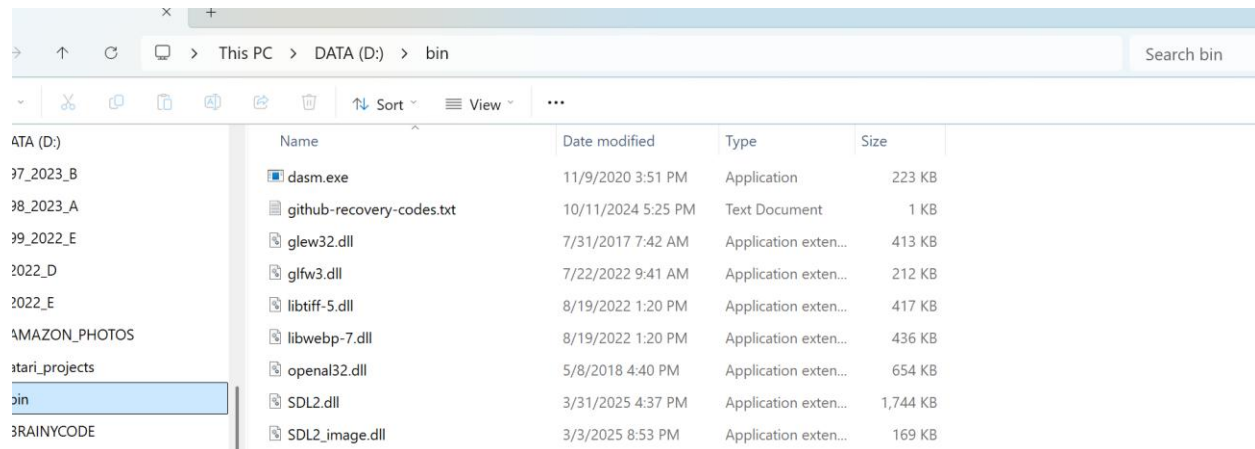
Figure 44 - The location of the SDL2.dll file

- The video presenter suggests putting the dll files for SDL2 and SDL2_image in the project debug directory:



Tutorial: Create 2D Game Engine using C++

I usually have a D:\bin or a C:\bin directory where I put in all my generic tools and common dlls that will be utilized across many projects:



I recommend that a more general location be created similar to the above and the folder be placed in the environment path.

Which ever you choose, once the *.dll files of SDL2.dll and SDL2_image.dll are made accessible the program should now execute successfully:

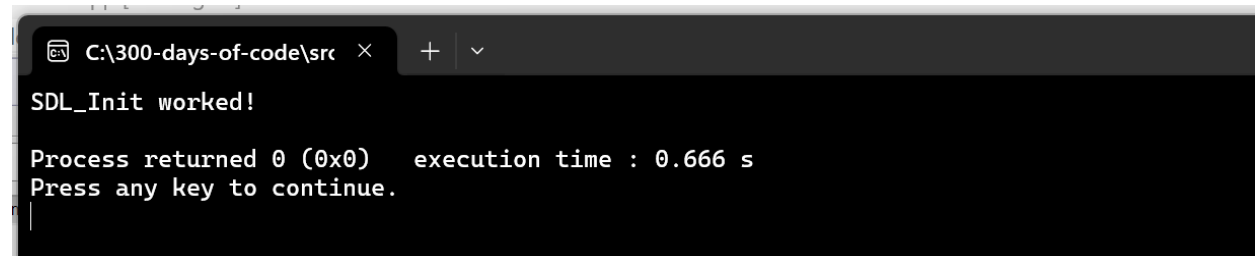


Figure 45 - Program with SDL component running ran successfully

2. Game Loop for SDL Game

In this video we implement the fundamental structure in every game and game engine – the **Game Loop**. We will add a new class – Engine.cpp (and Engine.h) that will have all the key operations that a game loop goes through. Our Main.cpp class will be modified to invoke the key game loop functions.

Opening the Project

If you are opening the project anew for the next video a fast way is to select File ➔ Recent projects and select the project you are working on .

Tutorial: Create 2D Game Engine using C++

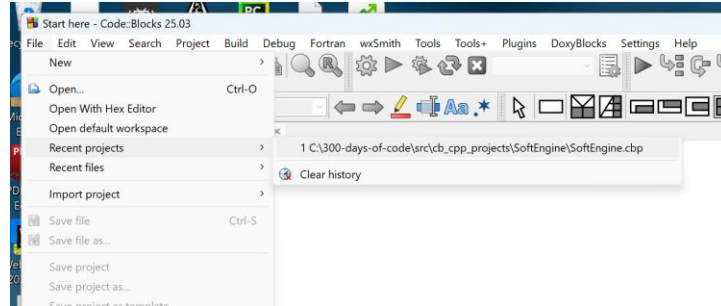


Figure 46 - Selecting a recent project

The goal of these set of videos is to create a 2D game engine that appears as follows:

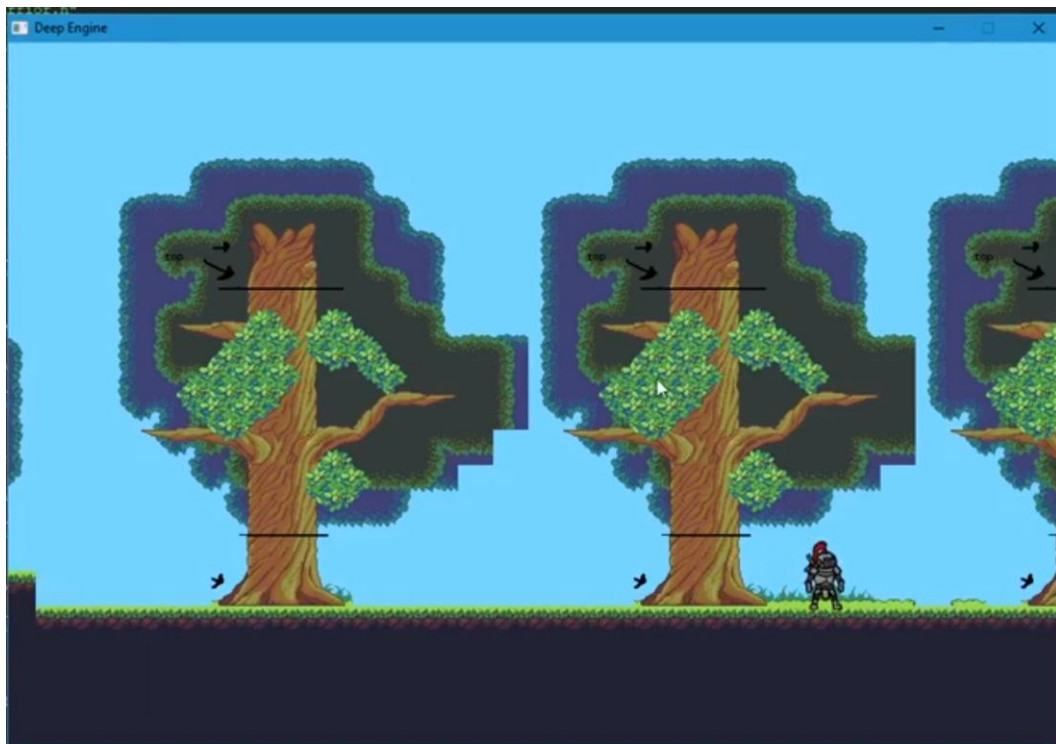


Figure 47 - The final look of our 2D Game Engine

TBD: Replace with an unmarked image

What is a game loop?

A game loop is the fundamental, repeating process that updates and renders a game's state. It's the core of how a game runs, constantly updating the game world, handling player inputs (via events) and drawing the graphics to the screen.

The core components of a Game Loop:

Tutorial: Create 2D Game Engine using C++

- Input: Processes player input from various sources like keyboard, mouse, or controller
- Update: Update the game's state, including physics, AI, and other game logic
- Render: Draws the updated game world to the screen

Initialization:

The game loop starts with an initialization phase where the game sets up its resources, including game states, graphics, and input systems.

Loop Execution:

The loop then repeatedly executes these phases:

- Process Input – The game detects player input and updates the corresponding entities.
- Update Game World – The game logic is applied, updating the state of the game world based on player input and game rules
- Render Graphics – The game renders the updated game world to the screen, creating the visual display for the player

Loop Continues:

This process continues until the game is closed or the loop's condition is no longer met (player won or player died).

Importance of the Game Loop:

- Smooth Gameplay: The game loop ensures the game runs smoothly and consistently, providing a fluid and responsive experience for the player
- State Management: The loop is responsible for managing the game's state, ensuring it remains consistent and up-to-date
- Foundation of Game Development: It's fundamental structure upon which most game development engines are built.

Tutorial: Create 2D Game Engine using C++

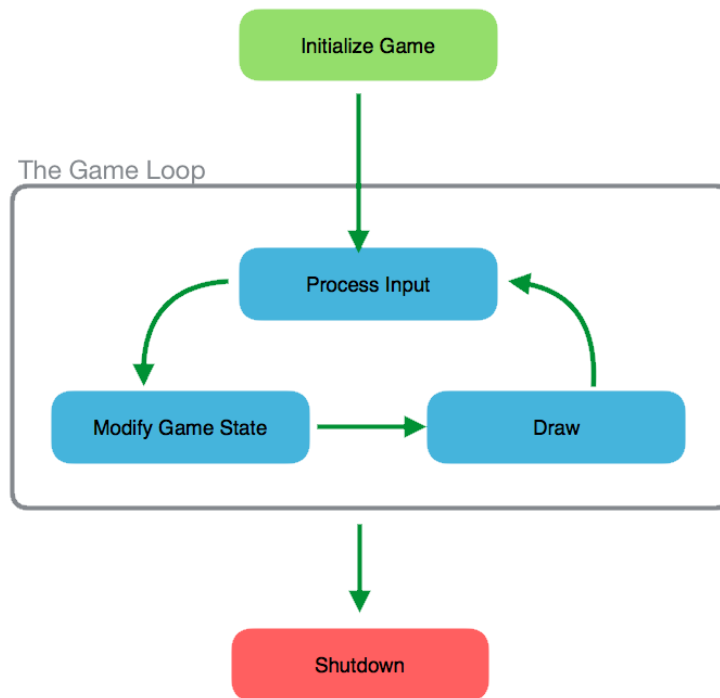


Figure 48 - A diagram of the "Game Loop"

Create an Engine class file

We will create an Engine.cpp and Engine.h C++ class that will capture the actions we want to implement a game loop.

- Select File → New → Class...

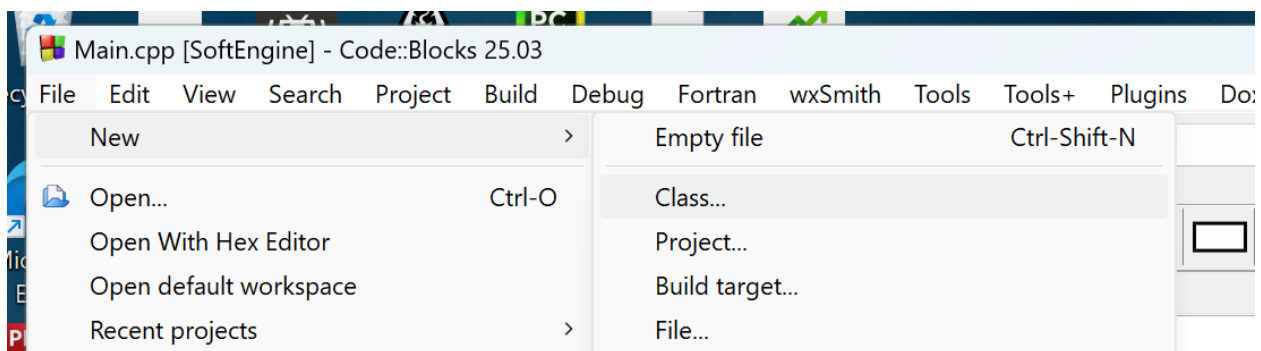


Figure 49 - Adding a new class to the project

- Fill in the "Create new class" dialog as shown:

Tutorial: Create 2D Game Engine using C++

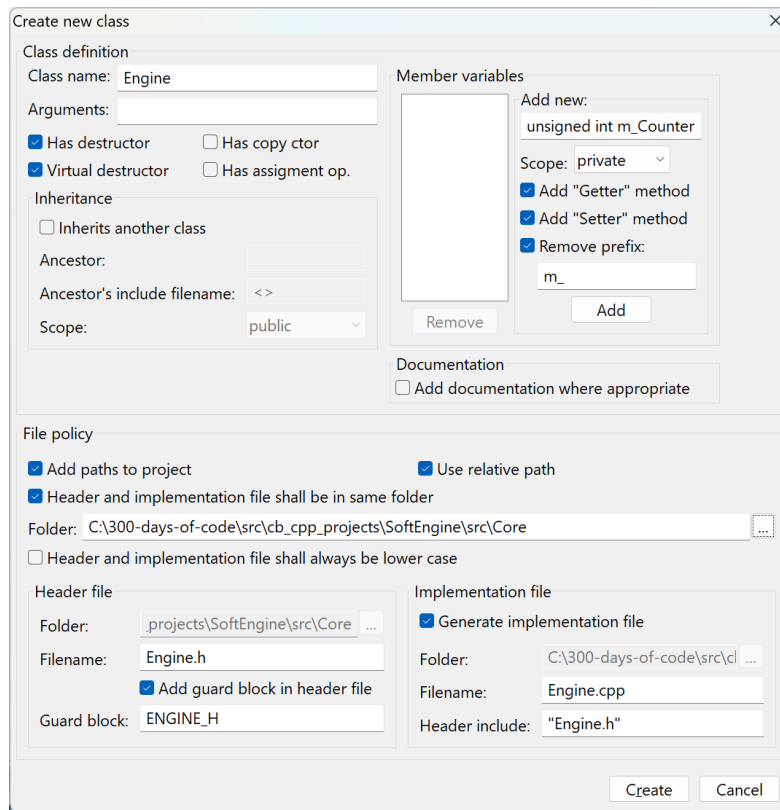


Figure 50 - Creating our Engine.cpp and Engine.h files

Note, the Folder for our new class is `\src\Core`. All the classes we will create for this project will be in their own folder under the `src` directory.

- Click on “Create”

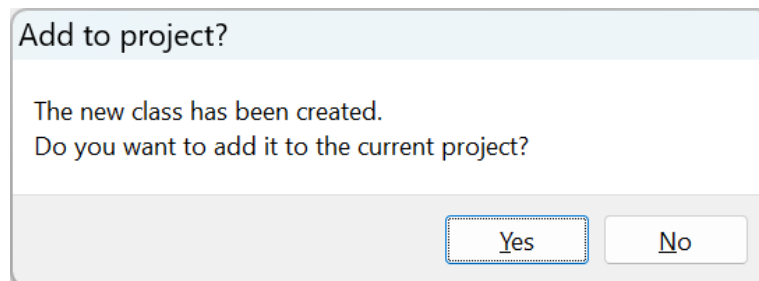


Figure 51 - Add to project prompt?

- Click “Yes” to the above prompt.

Tutorial: Create 2D Game Engine using C++

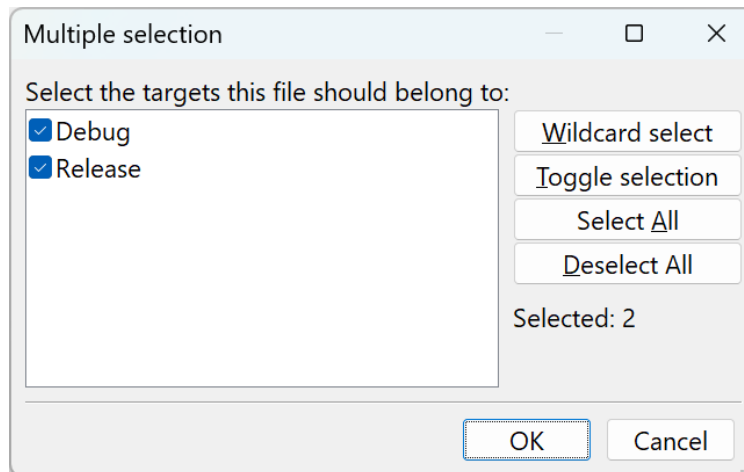


Figure 52 - Add to Debug and Release prompt

- Click “OK” to the above prompt.

The project will appear as:

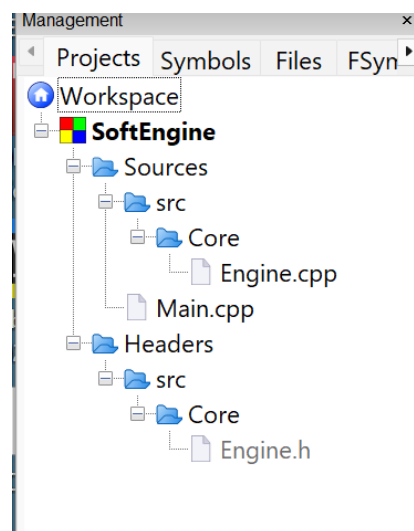
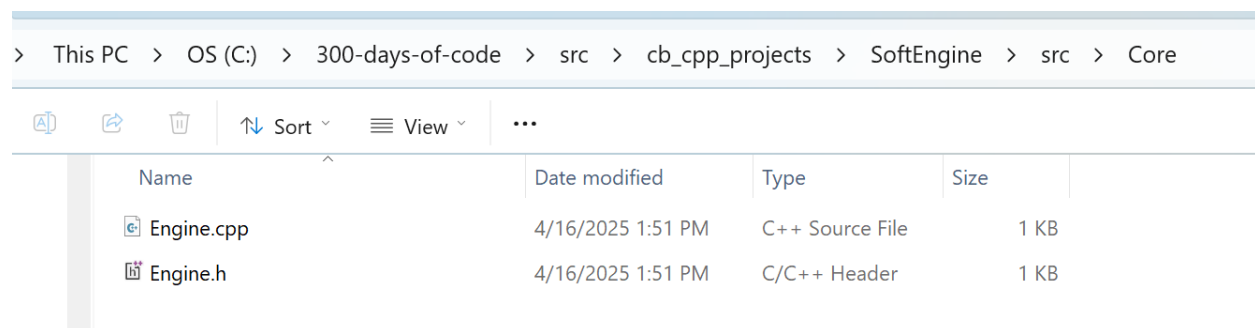


Figure 53 - Project view of the files

In fact, if you examine your folders:



Tutorial: Create 2D Game Engine using C++

You see that the Engine.cpp and Engine.h are actually in the same folder. The presenter likes to see the files together in the project view. Right-click on the SoftEngine project and select “Project tree” → “Categorize by file types”.

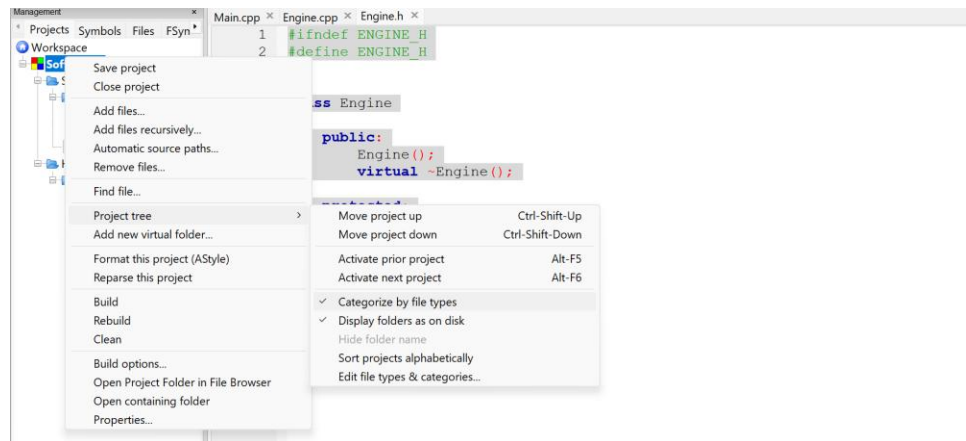
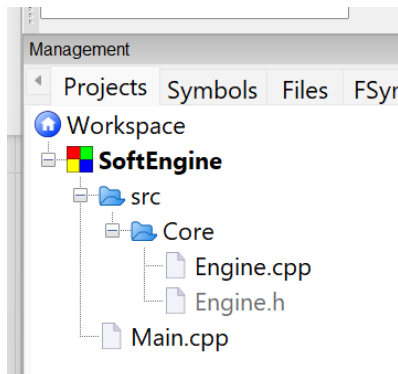


Figure 54 - Viewing the files as in actual folders

- Check “Categorize by file types” off

The result is:



Engine.cpp (initial):

```
1. #include "Engine.h"
2.
3. Engine::Engine()
4. {
5.     //ctor
6. }
7.
8. Engine::~~Engine()
9. {
10.    //dtor
11. }
```

Tutorial: Create 2D Game Engine using C++

The Engine.cpp file created has an empty constructor and empty destructor defined. This is the implementation class.

Engine.h (initial):

```
1. Engine.h (initial):
2. #ifndef ENGINE_H
3. #define ENGINE_H
4.
5.
6. class Engine
7. {
8.     public:
9.         Engine();
10.        virtual ~Engine();
11.
12.    protected:
13.
14.    private:
15. };
16.
17. #endif // ENGINE_H
18.
```

We want our Engine class to be a Singleton class.

What is a Singleton class?

A Singleton class in C++ is a design pattern that ensures a class has only **one instance** throughout the program and provides a global point of access to that instance. This is useful in cases where you need centralized management of a resource, like a configuration manager or a logging system.

The Singleton pattern is typically implemented by:

1. Using a private static pointer to the single instance of the class.
2. Making the constructor private to prevent the creation of multiple instances.
3. Providing a public static method that returns the single instance (creating it if it doesn't exist).

Here's a simple example:

```
1. #include <iostream>
2. #include <memory>
3.
4. class Singleton {
5. private:
6.     static std::unique_ptr<Singleton> instance; // Static pointer to the instance
7.     Singleton() {}                             // Private constructor
8.
9. public:
10.    Singleton(const Singleton&) = delete;        // Prevent copying
11.    Singleton& operator=(const Singleton&) = delete; // Prevent assignment
12.
13.    static Singleton& getInstance() {
14.        if (!instance) {
```

Tutorial: Create 2D Game Engine using C++

```
15.         instance = std::make_unique<Singleton>();
16.     }
17.     return *instance;
18. }
19.
20. void displayMessage() {
21.     std::cout << "Singleton instance accessed!\n";
22. }
23. };
24.
25. std::unique_ptr<Singleton> Singleton::instance = nullptr; // Initialize the static pointer
26.
27. int main() {
28.     Singleton& singleton = Singleton::getInstance();
29.     singleton.displayMessage();
30.
31.     return 0;
32. }
33.
```

The example provided is compatible with **C++11** and later versions. Features like `std::unique_ptr` for memory management and the explicit use of `delete` to prevent copying were introduced in C11. If you use this code with compilers supporting C11 or newer (e.g., C14, C17, C++20), it should work seamlessly.

We only want to have one instance of the Engine class.

Setting the compiler to use C++ 17

We probably should have done this earlier (in the first video) but we need to ensure that we use C++ 17 to match the presenter's version.

- Click on Settings ➔ Compiler...

We have several choices here:

| | |
|--|--------------------------|
| Have g++ follow the C++14 GNU C++ language standard (ISO C++ plus GNU extensions) [-std=gnu++14] | <input type="checkbox"/> |
| Have g++ follow the C++14 ISO C++ language standard [-std=c++14] | <input type="checkbox"/> |
| Have g++ follow the C++17 GNU C++ language standard (ISO C++ plus GNU extensions) [-std=gnu++17] | <input type="checkbox"/> |
| Have g++ follow the C++17 ISO C++ language standard [-std=c++17] | <input type="checkbox"/> |
| Have g++ follow the C++20 GNU C++ language standard (ISO C++ plus GNU extensions) [-std=gnu++20] | <input type="checkbox"/> |
| Have g++ follow the C++20 ISO C++ language standard [-std=c++20] | <input type="checkbox"/> |
| Have g++ follow the C++23 GNU C++ language standard (ISO C++ plus GNU extensions) [-std=gnu++23] | <input type="checkbox"/> |

The one that matches is:

| | |
|--|-------------------------------------|
| Have g++ follow the C++14 ISO C++ language standard [-std=c++14] | <input type="checkbox"/> |
| Have g++ follow the C++17 GNU C++ language standard (ISO C++ plus GNU extensions) [-std=gnu++17] | <input type="checkbox"/> |
| Have g++ follow the C++17 ISO C++ language standard [-std=c++17] | <input checked="" type="checkbox"/> |
| Have g++ follow the C++20 GNU C++ language standard (ISO C++ plus GNU extensions) [-std=gnu++20] | <input type="checkbox"/> |

Making Engine a Singleton

Engine.h:

```
1. class Engine
2. {
3.     public:
4.         Engine();
5.
6.         static Engine* GetInstance() {
7.             return s_Instance = (s_Instance != nullptr) ? s_Instance : new Engine();
8.         }
9.         virtual ~Engine();
10.
11.     protected:
12.
13.     private:
14.         static Engine* s_Instance;
15.
16. };
17.
```

We create a static class method that either returns an Engine* or creates it. The Engine instance is saved as a pointer in s_Instance.

We will only ever have one instance of our game Engine, therefore we will move the constructor into the private section.

```
1. class Engine
2. {
3.     public:
4.         static Engine* GetInstance() {
5.             return s_Instance = (s_Instance != nullptr) ? s_Instance : new Engine();
6.         }
7.         virtual ~Engine();
8.
9.     protected:
10.
11.     private:
12.         Engine() {};
13.         static Engine* s_Instance;
14.
15. };
16.
```

Adding key game loop functions to our Engine

The key game loop functions are Init() for initialization of our game, Events() to obtain input events (e.g. mouse move, keyboard entry, etc.), Update() to update the entities according to the game logic, Render() to update the graphical screen. The Clean() function is used to clean up all resources and finally Quit() to terminate the game.

Tutorial: Create 2D Game Engine using C++

We will also add an inline `isRunning()` function to be used for our game loop, as long as the member variable `m_IsRunning` is true, we execute a cycle of the game loop.

Adding to Engine.h:

```
1. #ifndef ENGINE_H
2. #define ENGINE_H
3.
4.
5. class Engine
6. {
7.     public:
8.         static Engine* GetInstance() {
9.             return s_Instance = (s_Instance != nullptr) ? s_Instance : new Engine();
10.        }
11.
12.        bool Init();
13.        bool Clean();
14.        void Quit();
15.
16.        void Update();
17.        void Render();
18.        void Events();
19.
20.        inline bool isRunning() {
21.            return m_IsRunning;
22.        }
23.
24.    protected:
25.
26.    private:
27.        Engine();
28.        static Engine* s_Instance;
29.        bool m_IsRunning;
30.
31. };
32.
33. #endif // ENGINE_H
34.
```

There are two class methods that are defined in Engine.h:

- `GetInstance()`
- `IsRunning()`

Adding to Engine.cpp

To get Code::Blocks to automatically add implementation functions for all the missing functions do the following:

- Right-click on the Engine.cpp page
- Select Insert/Refactor
- Select "All class method without implementation..."

Tutorial: Create 2D Game Engine using C++

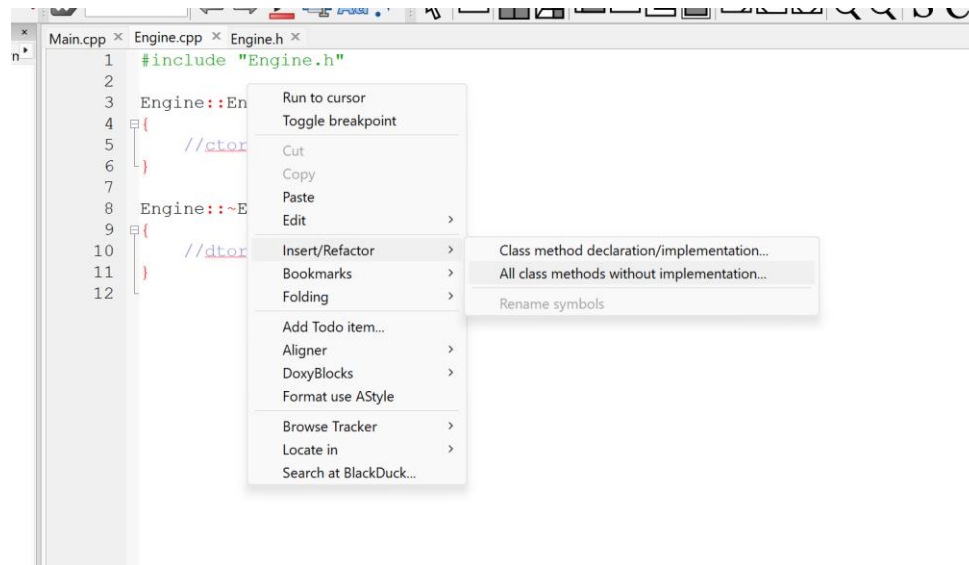


Figure 55 - Adding missing class methods

- Select all the methods you want the code to seed your class with:

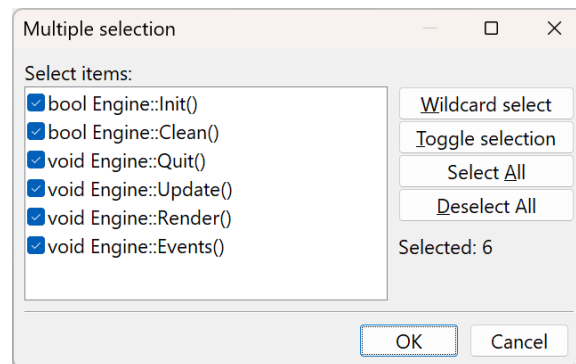


Figure 56 - Selecting the class methods to insert

- Click "OK"

Engine.cpp:

```
1. #include "Engine.h"
2.
3. Engine::Engine()
4. {
5.     //ctor
6. }
7.
8. bool Engine::Init()
9. {
10. }
11.
12.
13. bool Engine::Clean()
```

Tutorial: Create 2D Game Engine using C++

```
14. {
15.
16. }
17.
18. void Engine::Quit()
19. {
20.
21. }
22.
23. void Engine::Update()
24. {
25.
26. }
27.
28. void Engine::Render()
29. {
30.
31. }
32.
33. void Engine::Events()
34. {
35.
36. }
```

Updated Main.cpp:

```
1. #include "Engine.h"
2. #include <iostream>
3.
4. bool Engine::Init()
5. {
6. }
7.
8. bool Engine::Clean()
9. {
10. }
11.
12. void Engine::Quit()
13. {
14.
15. }
16.
17. void Engine::Update()
18. {
19.
20. }
21.
22. void Engine::Render()
23. {
24. }
25.
26. void Engine::Events()
27. {
28. }
29.
```

Let's now add minimal code to ensure that the above will work.

We will:

Tutorial: Create 2D Game Engine using C++

- Set s_Instance to nullptr;
- Insert std::cout statements to each function
- Add logic to Init() to set m_IsRunning

Engine.cpp:

```
1. #include "Engine.h"
2. #include <iostream>
3.
4. Engine* Engine::s_Instance = nullptr;
5.
6. bool Engine::Init()
7. {
8.     std::cout << "Initializing..." << std::endl;
9.     m_IsRunning = true;
10.    return true;
11. }
12.
13. bool Engine::Clean()
14. {
15.     std::cout << "Clean..." << std::endl;
16.     return true;
17. }
18.
19. void Engine::Quit()
20. {
21.
22. }
23.
24. void Engine::Update()
25. {
26.     std::cout << "Updating..." << std::endl;
27. }
28.
29. void Engine::Render()
30. {
31.     std::cout << "Render..." << std::endl;
32. }
33.
34. void Engine::Events()
35. {
36.     std::cout << "Events..." << std::endl;
37. }
38.
```

Now we implement the game loop in Main.cpp by invoking the methods in our Engine:

Main.cpp

```
1. #include "Engine.h"
2.
3. int main(int argc, char** argv)
4. {
5.     Engine::GetInstance()->Init();
6.
7.
8.     while (Engine::GetInstance()->isRunning()) {
9.         // Get all current events (e.g. mouse clicks, etc.)
10.        Engine::GetInstance()->Events();
11.    }
```


Tutorial: Create 2D Game Engine using C++

```
11.  
12.     // Update all objects/entities  
13.     Engine::GetInstance()->Update();  
14.  
15.     // Render/update the game graphics  
16.     Engine::GetInstance()->Render();  
17. }  
18.  
19. // Clean everything up  
20. Engine::GetInstance()->Clean();  
21.  
22. return 0;  
23. }  
24.
```

The code above implements the game loop! It does not do anything right now but print over and over again the `std::cout` messages associated with each Engine method.

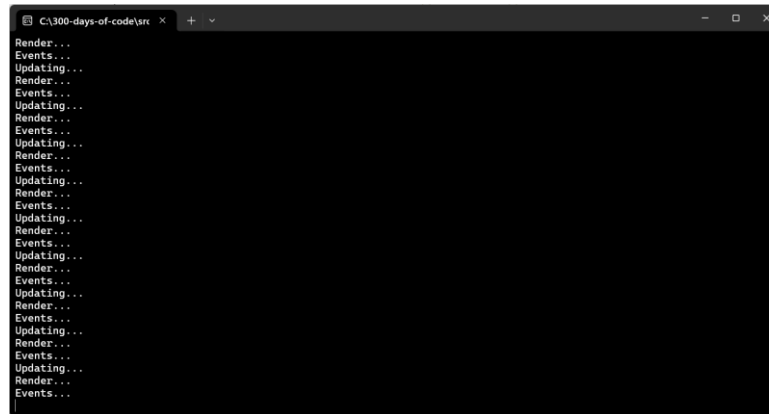


Figure 57 - Running our game loop

Note: If you followed the video you may have encountered an issue when you tried to run the program. The problem was due to the missing `bool` return values from the methods `Init()` and `Clean()`.

3. Create SDL Window and Renderer

TBD: Add Overview

The first file we will edit is `Engine.h`. We will add statements to the page to allow the Engine to access SDL functions:

```
1. #ifndef ENGINE_H  
2. #define ENGINE_H  
3. #include "SDL.h"  
4. #include "SDL_image.h"
```

The above `#include` will allow our Engine class to reference SDL and SDL_image elements.

In the private section we will add two new member class variables for an `SDL_Window` and an `SDL_Renderer`.

What is an `SDL_Window`?

An **`SDL_Window`** is a structure used in the **Simple DirectMedia Layer (SDL)** library to represent a window in your application. It serves as the foundation for rendering graphics, handling events, and interacting with the display. You can create an `SDL_Window` using the `SDL_CreateWindow` function, specifying parameters like the title, dimensions, position, and flags.

The window can have various properties, such as being resizable, fullscreen, or supporting OpenGL/Vulkan contexts. It also interacts with high-DPI displays and can handle input focus, mouse grabbing, and more.

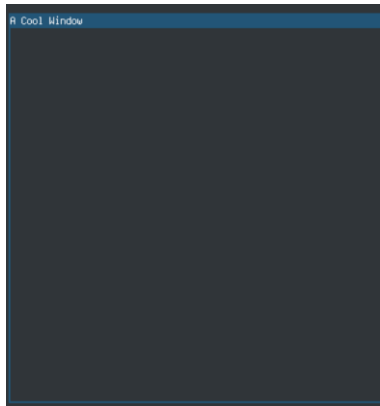


Figure 58 - An example of a Window (`SDL_Window`) from: <https://jakob.space/blog/sdl-tutorial-part-0x00.html>

Now that you have a window you want to draw or render objects on our window for that we create `SDL_Renderer`.

What is an `SDL_Renderer`?

An **`SDL_Renderer`** is a structure in the **Simple DirectMedia Layer (SDL)** library that handles rendering operations. It is tied to an **`SDL_Window`** and is used to draw graphics, such as textures, shapes, and other visual elements, onto the window.

Here are some key points about `SDL_Renderer`:

- **Creation:** You can create an `SDL_Renderer` using the `SDL_CreateRenderer` function, which links it to a specific `SDL_Window`.

Tutorial: Create 2D Game Engine using C++

- **Drawing:** It provides functions like `SDL_RenderCopy` to render textures, `SDL_SetRenderDrawColor` to set drawing colors, and `SDL_RenderClear` to clear the screen.
- **Hardware Acceleration:** `SDL_Renderer` can use hardware acceleration if supported by the system, making rendering faster and more efficient.
- **Presentation:** After drawing, you use `SDL_RenderPresent` to update the window with the rendered content.

Adding an `SDL_Window` and `SDL_Renderer` to `Engine.h`

We add two new member variables:

```
1. private:
2.     Engine() {}
3.     static Engine* s_Instance;
4.     SDL_Window* m_Window;
5.     SDL_Renderer* m_Renderer;
6.     bool m_IsRunning;
```

Since writing to the screen will be done across many other classes we will add an inline function to `Engine.h` to return the `m_Renderer` in the public section:

```
1. inline SDL_Renderer* GetRenderer() { return m_Renderer; }
```

Adding `SCREEN_WIDTH` and `SCREEN_HEIGHT`

A major consideration is what your game screen width and height will be, that is, the aspect ratio to use for your window.

Aspect ratios for your **`SDL_Window`** depend on the type of application you're developing and the devices you want to support. Here are some considerations:

1. Common Aspect Ratios:

- **16:9:** Widely used for modern displays, including HD and 4K screens.
- **4:3:** Suitable for older monitors or retro-style applications.
- **21:9:** Ideal for ultrawide monitors, often used in gaming.
- **1:1:** Square aspect ratio, useful for specific design or artistic purposes.

2. Maintaining Aspect Ratio:

Tutorial: Create 2D Game Engine using C++

- If your application allows window resizing, you can use functions like `SDL_SetWindowAspectRatio` to enforce a specific aspect ratio.
- When rendering, ensure your content scales proportionally to avoid distortion.

3. Application Context:

- For games, 16:9 is a safe choice as it's standard for most displays.
- For productivity tools or creative software, consider supporting multiple aspect ratios to accommodate different user setups.

The presenter decides to use 960x640:

```
1. #define SCREEN_WIDTH 960
2. #define SCREEN_HEIGHT 640
```

The resolution **960x640** corresponds to an aspect ratio of **3:2**. This aspect ratio is less common for modern displays, which often favor **16:9** or **4:3**, but it has its own merits depending on your use case:

- **Photography:** The 3:2 aspect ratio is widely used in photography, as it matches the native aspect ratio of many DSLR and mirrorless cameras.
- **Retro or Niche Applications:** If you're designing something with a retro aesthetic or targeting specific devices, 3:2 might be a good fit.
- **Content Scaling:** Keep in mind that using a less common aspect ratio might require additional effort to scale or crop content for modern screens.

If you're developing an SDL application, consider your target audience and devices. For general-purpose apps or games, **16:9** might be a safer choice for compatibility.

Editing Engine.cpp

- Remove all the `std::cout` statements in the file `Engine.cpp`
- Update the `Engine::Init()` to initialize SDL

```
1. bool Engine::Init()
2. {
3.     if (SDL_Init(SDL_INIT_VIDEO) != 0 && IMG_Init(IMG_INIT_JPG | IMG_INIT_PNG) != 0) {
4.         SDL_Log("Failed to initialize SDL: %s", SDL_GetError());
5.         return false;
6.     }
7.
8.     return m_IsRunning = true;
9. }
```

What is SDL_Init?

The `SDL_Init()` function is used to initialize the **Simple DirectMedia Layer (SDL)** library and its subsystems. It's one of the first functions you call when starting an SDL application. Here's a breakdown of its details:

Syntax:

```
1. int SDL_Init(Uint32 flags);
```

Parameters:

- **flags**: Specifies which SDL subsystems to initialize. You can combine multiple flags using the bitwise OR operator (`|`). Common flags include:
 - **SDL_INIT_VIDEO**: Initializes the video subsystem.
 - **SDL_INIT_AUDIO**: Initializes the audio subsystem.
 - **SDL_INIT_TIMER**: Initializes the timer subsystem.
 - **SDL_INIT_EVENTS**: Initializes the event handling subsystem.
 - **SDL_INIT_JOYSTICK**: Initializes the joystick subsystem.
 - **SDL_INIT_GAMEPAD**: Initializes the gamepad subsystem.
 - **SDL_INIT_HAPTIC**: Initializes the haptic (force feedback) subsystem.
 - **SDL_INIT EVERYTHING**: Initializes all available subsystems.

Return Value

- Returns 0 on success.
- Returns a negative error code on failure. You can use `SDL_GetError()` to retrieve a human-readable error message.

The video does not discuss the fact that for every subsystem you initialize you should also close it down in order to release resources `SDL_Quit()`;

What is IMG_Init?

`IMG_Init` is a function from the **SDL_image** library, an extension of SDL that provides support for loading and handling various image formats. This function initializes the `SDL_image` library and prepares it to work with specific image formats.

Syntax

```
1. int IMG_Init(int flags);
```

Parameters

- **flags**: Specifies the image formats to initialize. You can combine multiple flags using the bitwise OR operator (`|`). Common flags include:
 - **IMG_INIT_JPG**: Support for JPEG images.
 - **IMG_INIT_PNG**: Support for PNG images.
 - **IMG_INIT_TIF**: Support for TIFF images.
 - **IMG_INIT_WEBP**: Support for WebP images.

Return Value

- Returns a bitmask of the successfully initialized formats.
- If the return value doesn't match the requested flags, it means some formats failed to initialize.

Notes

- You must call `IMG_Init` before using any `SDL_image` functions that depend on specific image formats.
- Always call `IMG_Quit` to clean up when you're done using `SDL_image`.

What is `SDL_GetError()`?

`SDL_GetError()` is a function in the SDL library that retrieves a human-readable error message describing the last error that occurred in the current thread. It's incredibly useful for debugging, as it provides insight into why an SDL function might have failed.

Syntax

```
1. const char* SDL_GetError(void);
```

Return Value

- Returns a string containing the error message.

- If no error has occurred, it returns an empty string ("").

Key Points

- **Thread-Specific:** The error message is thread-local, meaning errors in other threads won't interfere with the current thread's error state.
- **Last Error Only:** It only retrieves the most recent error. If multiple errors occur, earlier ones are overwritten.
- **Doesn't Clear Errors:** Calling `SDL_GetError()` doesn't reset the error state. To clear the error, use `SDL_ClearError()`.

Notes

- Always check the return values of SDL functions to determine when to call `SDL_GetError()`.
- Use it alongside `SDL_SetError()` if you want to set custom error messages for debugging.

What is SDL_Log?

`SDL_Log()` is a function in the SDL library used for logging messages to the console or other output streams. It provides a simple way to debug and track the behavior of your application.

Syntax

```
1. void SDL_Log(const char *fmt, ...);
```

Parameters

- **fmt:** A printf()-style format string.
- **...:** Additional parameters that match the format specifiers in `fmt`.

Features

- **Thread-Safe:** You can safely call `SDL_Log()` from any thread.
- **Categories and Priorities:** SDL provides logging categories (e.g., `SDL_LOG_CATEGORY_APPLICATION`) and priorities (e.g., `SDL_LOG_PRIORITY_INFO`) for more structured logging. You can use functions like `SDL_LogMessage()` for categorized logging.

Notes

Tutorial: Create 2D Game Engine using C++

- By default, logs are quiet, but you can adjust the logging priority using `SDL_LogSetPriority()` or `SDL_LogSetAllPriority()`.
- On different platforms, logs are directed to different outputs (e.g., debug output stream on Windows, log output on Android, or `stderr` on others).

Resume Editing Engine.cpp

- Edit `Engine::Update()`

```
1. void Engine::Update()  
2. {  
3.     SDL_Log("Updating in the Game Loop...");  
4. }
```

At this point we have our `Engine::Init` initializing SDL subsystems.

- Test everything by running the application

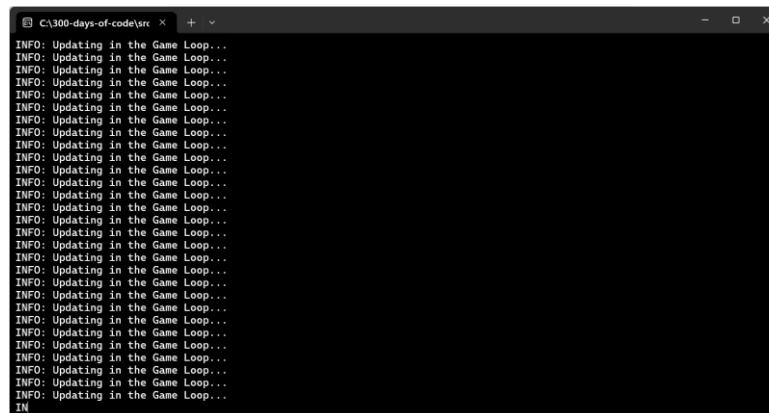


Figure 59 - Running the program with SDL Initialization

Creating `SDL_Window` and `SDL_Renderer`

We will now add the code to create our `SDL_Window` and allow ourselves to draw on it with `SDL_Renderer`.

We start to use <https://wiki.libsdl.org/SDL2/FrontPage> functions.

`SDL_CreateWindow`

The `SDL_CreateWindow` is used to create a window with specified position, dimensions, and flags.

Syntax:

Tutorial: Create 2D Game Engine using C++

```
SDL_Window * SDL_CreateWindow(const char *title,
                             int x, int y, int w,
                             int h, Uint32 flags);
```

Function Parameters:

| | | |
|------------------------|--------------|---|
| const char * | title | the title of the window, in UTF-8 encoding. |
| int | x | the x position of the window, SDL_WINDOWPOS_CENTERED , or SDL_WINDOWPOS_UNDEFINED . |
| int | y | the y position of the window, SDL_WINDOWPOS_CENTERED , or SDL_WINDOWPOS_UNDEFINED . |
| int | w | the width of the window, in screen coordinates. |
| int | h | the height of the window, in screen coordinates. |
| Uint32 | flags | 0, or one or more SDL_WindowFlags OR'd together. |

`SDL_WINDOWPOS_CENTERED` is a predefined constant in the SDL library that specifies the position of a window to be **centered** on the screen. When you use this flag while creating an SDL window, SDL automatically calculates the coordinates needed to place the window in the center of the display. It is used to simplify window positioning, especially when you want your application to look polished by centering the window.

Note: If you have multiple monitors, `SDL_WINDOWPOS_CENTERED` will center the window on the **primary display**. For centering on a specific display, you can use `SDL_WINDOWPOS_CENTERED_DISPLAY(n)` where n is the display index.

`SDL_WINDOWPOS_UNDEFINED` is a predefined constant in the SDL library that indicates you don't care about the initial position of a window when creating it. When you use this flag, SDL will allow the operating system to decide where to place the window on the screen.

Note: If you want to specify the position later, you can use `SDL_SetWindowPosition()`.

The flags parameter can be one or more `SDL_WindowFlags`:

```
1. typedef enum SDL_WindowFlags
2. {
3.     SDL_WINDOW_FULLSCREEN = 0x00000001,           /**< fullscreen window */
4.     SDL_WINDOW_OPENGL = 0x00000002,              /**< window usable with OpenGL context */
5.     SDL_WINDOW_SHOWN = 0x00000004,               /**< window is visible */
6.     SDL_WINDOW_HIDDEN = 0x00000008,              /**< window is not visible */
7.     SDL_WINDOW_BORDERLESS = 0x00000010,          /**< no window decoration */
8.     SDL_WINDOW_RESIZABLE = 0x00000020,          /**< window can be resized */
9.     SDL_WINDOW_MINIMIZED = 0x00000040,          /**< window is minimized */
10.    SDL_WINDOW_MAXIMIZED = 0x00000080,           /**< window is maximized */
11.    SDL_WINDOW_MOUSE_GRABBED = 0x00000100,        /**< window has grabbed mouse input */
12.    SDL_WINDOW_INPUT_FOCUS = 0x00000200,         /**< window has input focus */
13.    SDL_WINDOW_MOUSE_FOCUS = 0x00000400,         /**< window has mouse focus */
14.    SDL_WINDOW_FULLSCREEN_DESKTOP = ( SDL_WINDOW_FULLSCREEN | 0x00001000 ),
15.    SDL_WINDOW_FOREIGN = 0x00000800,             /**< window not created by SDL */
}
```

Tutorial: Create 2D Game Engine using C++

```
16.     SDL_WINDOW_ALLOW_HIGHDPI = 0x00002000,    /**< window should be created in high-DPI mode
if supported.
17.                                     On macOS NSHighResolutionCapable must be
18.                                     application's Info.plist for this to have
any effect. */
19.     SDL_WINDOW_MOUSE_CAPTURE   = 0x00004000,    /**< window has mouse captured (unrelated to
SDL_WINDOW_MOUSE_GRABBED) */
20.     SDL_WINDOW_ALWAYS_ON_TOP   = 0x00008000,    /**< window should always be above others */
21.     SDL_WINDOW_SKIP_TASKBAR    = 0x00010000,    /**< window should not be added to the taskbar
*/
22.     SDL_WINDOW_UTILITY         = 0x00020000,    /**< window should be treated as a utility
window */
23.     SDL_WINDOW_TOOLTIP         = 0x00040000,    /**< window should be treated as a tooltip */
24.     SDL_WINDOW_POPUP_MENU      = 0x00080000,    /**< window should be treated as a popup menu */
25.     SDL_WINDOW_KEYBOARD_GRABBED = 0x00100000,    /**< window has grabbed keyboard input */
26.     SDL_WINDOW_VULKAN          = 0x10000000,    /**< window usable for Vulkan surface */
27.     SDL_WINDOW_METAL           = 0x20000000,    /**< window usable for Metal view */
28.
29.     SDL_WINDOW_INPUT_GRABBED = SDL_WINDOW_MOUSE_GRABBED /**< equivalent to
SDL_WINDOW_MOUSE_GRABBED for compatibility */
30. } SDL_WindowFlags;
31.
```

Setting the **flags** parameter of `SDL_CreateWindow` to **0** means that no special properties or behaviors are applied to the window. The window will be created with default settings, and it won't have features like fullscreen mode, OpenGL/Vulkan support, or resizable borders.

Default Behavior

When the flags parameter is set to 0:

- The window is **visible** by default.
- It has **standard decorations** (title bar, borders, etc.).
- It is **not resizable**, unless explicitly specified later.

Notes

If you want to add specific features to the window, you can use flags like:

- **SDL_WINDOW_FULLSCREEN**: Makes the window fullscreen.
- **SDL_WINDOW_RESIZABLE**: Allows the window to be resized.
- **SDL_WINDOW_OPENGL**: Enables OpenGL rendering.

SDL_CreateRenderer

The `SDL_CreateRenderer` is used to create a 2D rendering context for a window.

Syntax:

```
1. SDL_Renderer * SDL_CreateRenderer(SDL_Window * window,  
2.                                int index, Uint32 flags);
```

Function Parameters:

| | | |
|------------------------------|---------------|--|
| SDL_Window * | window | the window where rendering is displayed. |
| int | index | the index of the rendering driver to initialize, or -1 to initialize the first one supporting the requested flags. |
| Uint32 | flags | 0, or one or more SDL_RendererFlags OR'd together. |

Return Value:

([SDL_Renderer](#) *) Returns a valid rendering context or NULL if there was an error;
call [SDL_GetError\(\)](#) for more information.

The `SDL_RendererFlags` are used to create a rendering context. Here are the possible values:

```
1. typedef enum SDL_RendererFlags  
2. {  
3.     SDL_RENDERER_SOFTWARE = 0x00000001,    /**< The renderer is a software fallback */  
4.     SDL_RENDERER_ACCELERATED = 0x00000002, /**< The renderer uses hardware  
5.                                     acceleration */  
6.     SDL_RENDERER_PRESENTVSYNC = 0x00000004, /**< Present is synchronized  
7.                                     with the refresh rate */  
8.     SDL_RENDERER_TARGETTEXTURE = 0x00000008 /**< The renderer supports  
9.                                     rendering to texture */  
10. } SDL_RendererFlags;
```

Updating Engine.cpp

We now update `Engine::Init` to create our `m_Window` and `m_Renderer`.

```
1. bool Engine::Init()  
2. {  
3.     if (SDL_Init(SDL_INIT_VIDEO) != 0 && IMG_Init(IMG_INIT_JPG | IMG_INIT_PNG) != 0) {  
4.         SDL_Log("Failed to initialize SDL: %s", SDL_GetError());  
5.         return false;  
6.     }  
7.  
8.     // Create our SDL window  
9.     m_Window = SDL_CreateWindow("Soft Engine", SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED,  
10.    SCREEN_WIDTH, SCREEN_HEIGHT, 0);  
11.     if (m_Window == nullptr) {  
12.         SDL_Log("Failed to create window: %s", SDL_GetError());  
13.     }  
14. }
```

Tutorial: Create 2D Game Engine using C++

```
12.         return false;
13.     }
14.
15.     m_Renderer = SDL_CreateRenderer(m_Window, -1, SDL_RENDERER_ACCELERATED |
SDL_RENDERER_PRESENTVSYNC);
16.     if (m_Renderer == nullptr) {
17.         SDL_Log("Failed to create Renderer: %s", SDL_GetError());
18.         return false;
19.     }
20.
21.     return m_IsRunning = true;
22. }
23.
```

Running and Testing the Application

If you run the application now you will notice that you cannot close the SDL_Window, this is because we did not add code to be responsive to users actions (via Events). So for now close the background window.

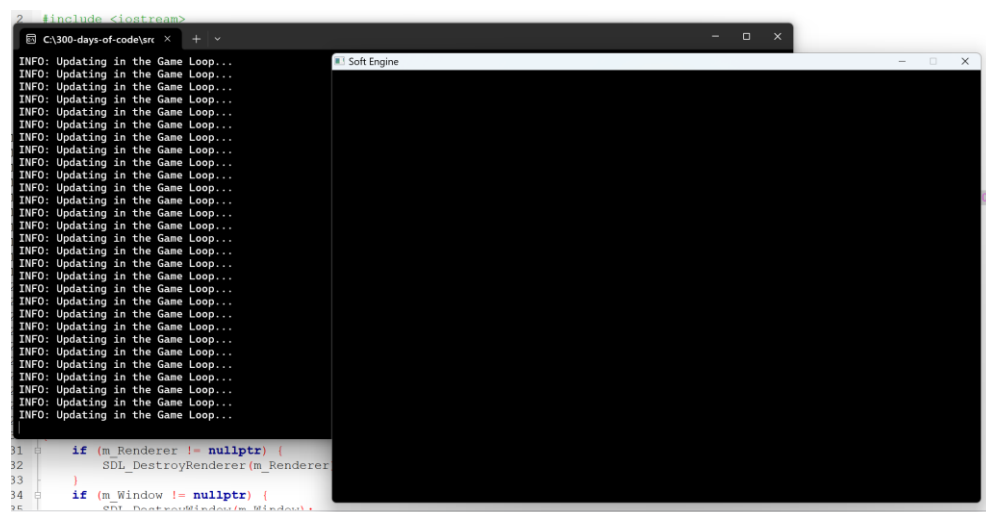


Figure 60 - Running the application at this stage of development

Handling Events

In order to provide user control of the SDL screen we need to be able to handle events like close window, etc.

What is an SDL_Event?

The data structure `SDL_Event` holds an event structure. An event can span various things like Window event (closing the window), Keyboard event (pressing a key), Mouse event (pressing a button). So `SDL_Event` is a union of other structures.

Tutorial: Create 2D Game Engine using C++

Syntax:

```
1. typedef union SDL_Event
2. {
3.     Uint32 type;                                /**< Event type, shared with all events */
4.     SDL_CommonEvent common;                     /**< Common event data */
5.     SDL_DisplayEvent display;                   /**< Display event data */
6.     SDL_WindowEvent window;                     /**< Window event data */
7.     SDL_KeyboardEvent key;                       /**< Keyboard event data */
8.     SDL_TextEditingEvent edit;                   /**< Text editing event data */
9.     SDL_TextEditingExtEvent editExt;             /**< Extended text editing event data */
10.    SDL_TextInputEvent text;                      /**< Text input event data */
11.    SDL_MouseMotionEvent motion;                  /**< Mouse motion event data */
12.    SDL_MouseButtonEvent button;                  /**< Mouse button event data */
13.    SDL_MouseWheelEvent wheel;                    /**< Mouse wheel event data */
14.    SDL_JoyAxisEvent jaxis;                       /**< Joystick axis event data */
15.    SDL_JoyBallEvent jball;                       /**< Joystick ball event data */
16.    SDL_JoyHatEvent jhat;                         /**< Joystick hat event data */
17.    SDL_JoyButtonEvent jbutton;                   /**< Joystick button event data */
18.    SDL_JoyDeviceEvent jdevice;                   /**< Joystick device change event data */
19.    SDL_JoyBatteryEvent jbattery;                 /**< Joystick battery event data */
20.    SDL_ControllerAxisEvent caxis;                 /**< Game Controller axis event data */
21.    SDL_ControllerButtonEvent cbutton;             /**< Game Controller button event data */
22.    SDL_ControllerDeviceEvent cdevice;             /**< Game Controller device event data */
23.    SDL_ControllerTouchpadEvent ctouchpad;         /**< Game Controller touchpad event data */
24.    SDL_ControllerSensorEvent csensor;             /**< Game Controller sensor event data */
25.    SDL_AudioDeviceEvent adevice;                 /**< Audio device event data */
26.    SDL_SensorEvent sensor;                       /**< Sensor event data */
27.    SDL_QuitEvent quit;                           /**< Quit request event data */
28.    SDL_UserEvent user;                           /**< Custom event data */
29.    SDL_SysWMEvent syswm;                         /**< System dependent window event data */
30.    SDL_TouchFingerEvent tfinger;                 /**< Touch finger event data */
31.    SDL_MultiGestureEvent mgesture;               /**< Gesture event data */
32.    SDL_DollarGestureEvent dgesture;              /**< Gesture event data */
33.    SDL_DropEvent drop;                           /**< Drag and drop event data */
34.
35.    /* This is necessary for ABI compatibility between Visual C++ and GCC.
36.       Visual C++ will respect the push pack pragma and use 52 bytes (size of
37.       SDL_TextEditingEvent, the largest structure for 32-bit and 64-bit
38.       architectures) for this union, and GCC will use the alignment of the
39.       largest datatype within the union, which is 8 bytes on 64-bit
40.       architectures.
41.
42.       So... we'll add padding to force the size to be 56 bytes for both.
43.
44.       On architectures where pointers are 16 bytes, this needs rounding up to
45.       the next multiple of 16, 64, and on architectures where pointers are
46.       even larger the size of SDL_UserEvent will dominate as being 3 pointers.
47.    */
48.    Uint8 padding[sizeof(void *) <= 8 ? 56 : sizeof(void *) == 16 ? 64 : 3 * sizeof(void *)];
49. } SDL_Event;
50.
```

Using events

The [SDL_Event](#) structure has two uses:

- Reading events from the event queue
- Placing events on the event queue

Reading events from the event queue

Reading events from the event queue is done with either [SDL_PollEvent](#) or [SDL_PeepEvents](#).

SDL_PollEvent

Poll for currently pending events.

Syntax:

```
1. int SDL_PollEvent(SDL_Event * event);
```

Function Parameters

| | | |
|-----------------------------|--------------|---|
| SDL_Event * | event | the SDL_Event structure to be filled with the next event from the queue, or NULL. |
|-----------------------------|--------------|---|

The [SDL_PollEvent](#) function takes a pointer to an [SDL_Event](#) structure that is to be filled with event information. We know that if [SDL_PollEvent](#) removes an event from the queue then the event information will be placed in our event structure, but we also know that the type of event will be placed in the **type** member of event. So to handle each event type separately we use a switch statement.

```
1. switch (event.type) {
```

We need to know what kind of events we're looking for and the event types of those events.

The `SDL_Event.type` field represents the type of event in SDL's event system. Here are some of the possible values:

Application Events

- **SDL_QUIT**: Indicates a user-requested quit (e.g., closing the window).
- **SDL_APP_TERMINATING**: The application is being terminated by the operating system.
- **SDL_APP_LOWMEMORY**: The application is low on memory.
- **SDL_APP_WILLENTERBACKGROUND**: The application is about to enter the background.
- **SDL_APP_DIDENTERBACKGROUND**: The application has entered the background.
- **SDL_APP_WILLENTERFOREGROUND**: The application is about to enter the foreground.

- **SDL_APP_DIDENTERFOREGROUND**: The application has entered the foreground.

Window Events

- **SDL_WINDOWEVENT**: Represents various window-related events (e.g., resizing, moving, focus changes).
- **SDL_SYSWMEVENT**: System-specific window events.

Keyboard Events

- **SDL_KEYDOWN**: A key was pressed.
- **SDL_KEYUP**: A key was released.
- **SDL_TEXTEDITING**: Text editing (composition).
- **SDL_TEXTINPUT**: Text input.

Mouse Events

- **SDL_MOUSEMOTION**: The mouse moved.
- **SDL_MOUSEBUTTONDOWN**: A mouse button was pressed.
- **SDL_MOUSEBUTTONUP**: A mouse button was released.
- **SDL_MOUSEWHEEL**: Mouse wheel motion.

Joystick Events

- **SDL_JOYAXISMOTION**: Joystick axis motion.
- **SDL_JOYBALLMOTION**: Joystick trackball motion.
- **SDL_JOYHATMOTION**: Joystick hat position change.
- **SDL_JOYBUTTONDOWN**: Joystick button pressed.
- **SDL_JOYBUTTONUP**: Joystick button released.
- **SDL_JOYDEVICEADDED**: A joystick was connected.
- **SDL_JOYDEVICEREMOVED**: A joystick was disconnected.

Controller Events

- **SDL_CONTROLLERAXISMOTION**: Game controller axis motion.
- **SDL_CONTROLLERBUTTONDOWN**: Game controller button pressed.

- **SDL_CONTROLLERBUTTONDOWN**: Game controller button released.
- **SDL_CONTROLLERDEVICEADDED**: A game controller was connected.
- **SDL_CONTROLLERDEVICEREMOVED**: A game controller was disconnected.

Other Events

- **SDL_AUDIODEVICEADDED**: An audio device was added.
- **SDL_AUDIODEVICEREMOVED**: An audio device was removed.
- **SDL_SENSORUPDATE**: Sensor data update.
- **SDL_DROPFILE**: A file was dropped onto the window.

This is just a subset of the possible event types. You can find a comprehensive list in the SDL documentation [here](#). Let me know if you'd like help handling specific events!

Each event type is associated with a different union member that holds more details about the event.

SDL_Quit

The **SDL_Quit** event is triggered when the user requests to terminate the application. This typically happens when the user closes the window or selects a quit option from the operating system. The event is part of SDL's event system and is represented by the **SDL_QUIT** constant.

Key Details:

- **Event Type**: **SDL_QUIT**
- **Purpose**: Indicates that the application should shut down gracefully.
- **Usage**: You can handle this event in your main event loop to perform cleanup tasks before exiting.

Update to Engine.cpp

In this section we add code to Engine.cpp to allow the user to “quit” or “close” the window application.

Add code the Engine::Events()

```
1. void Engine::Events()  
2. {  
3.     SDL_Event event;  
4.     SDL_PollEvent(&event);  
5.     switch(event.type) {  
6.         case SDL_QUIT:  
7.             Quit();
```


Tutorial: Create 2D Game Engine using C++

```
8.         break;
9.     }
10. }
11.
```

On line 3 we create an `SDL_Event` and then obtain the next item in the event queue by invoking `SDL_PollEvent` on line 4. The details of the next event is stored in the event variable.

The switch statement on 5 examines a value all events have type. If the event.type is `SDL_QUIT` we invoke an internal class method called `Quit()`.

Add code the `Engine::Quit()`

If we examine our game loop in `Main.cpp`:

```
1.     while (Engine::GetInstance()->isRunning()) {
2.         // Get all current events (e.g. mouse clicks, etc.)
3.         Engine::GetInstance()->Events();
4.
5.         // Update all objects/entities
6.         Engine::GetInstance()->Update();
7.
8.         // Render/update the game graphics
9.         Engine::GetInstance()->Render();
10.    }
```

The way to exit the application is to have `Engine::GetInstance()->isRunning()` to return `false`.

```
1.     inline bool isRunning() {
2.         return m_IsRunning;
3.     }
```

The value returned by `isRunning()` is just the current value of `m_IsRunning`. So, if the user closes the application we should just set the `m_IsRunning` class member variable to `false`. This should occur in `Quit()`.

```
1. void Engine::Quit()
2. {
3.     m_IsRunning = false;
4. }
```

Let's test and run the program again. You will see that you can move the window around (wondering how since we did not add an event handler for that?) and when we close the window the SDL window closes.