Tutorial: Create 2D Game Engine using C++

URL: https://www.youtube.com/playlist?list=PL-K0viiuJ2RctP5nlJlqmHGeh66-GOZR

Purpose

These notes summarize the you tube video playlist "How to Make a 2D Game Engine with C++ & SDL". We will 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 to match the video. I am using the latest version of SDL2 and `Code::Blocks`. Each section covers the material in the video faithfully[1]. So I don't feel like a code monkey I do add a lot of notes on the specific SDL functions so that I understand how the SDL functions work.

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 game program.

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.

[1] We may remove the oversights, errors and corrections

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!

The key feature of Code::Blocks is that you will be able to find an installation package for all modern operating systems – Windows, Linux and Mac OS. This is why the emphasis on cross-platform. You should have no issue following the video tutorials using any operating system. In addition, we use all open source libraries.

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:

Figure 1 - Selecting the right setup.exe to download

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

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

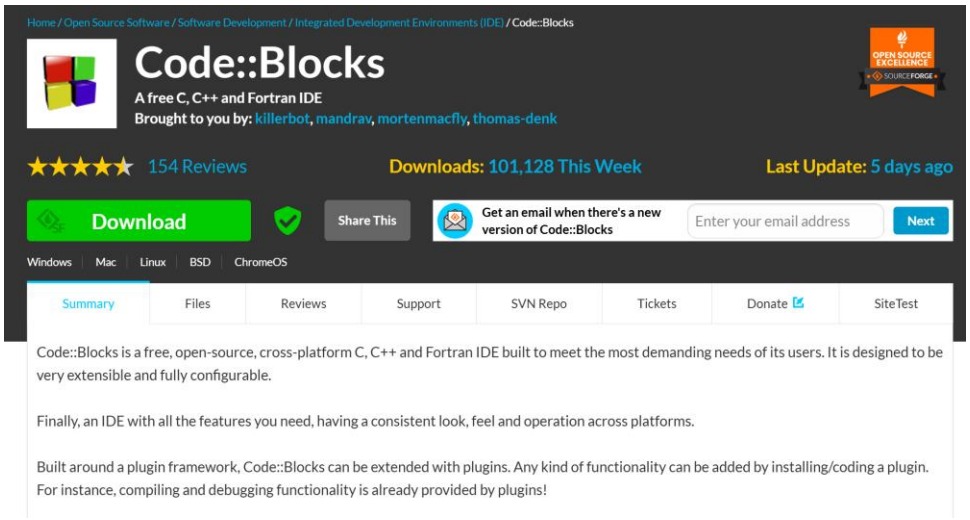


Figure 2 - Downloading file from Sourceforge

I then run the setup.exe file:

Figure 3 - Locating and executing the setup.exe file

Running the Setup

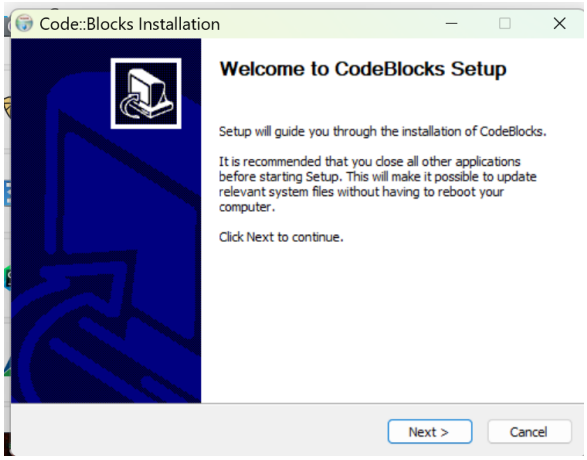


Figure 4 - Initial Code::Blocks Installation dialog

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

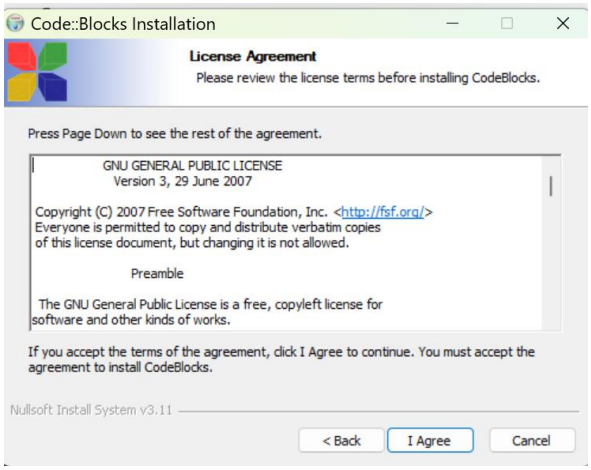


Figure 5 - The License Agreement screen

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”.

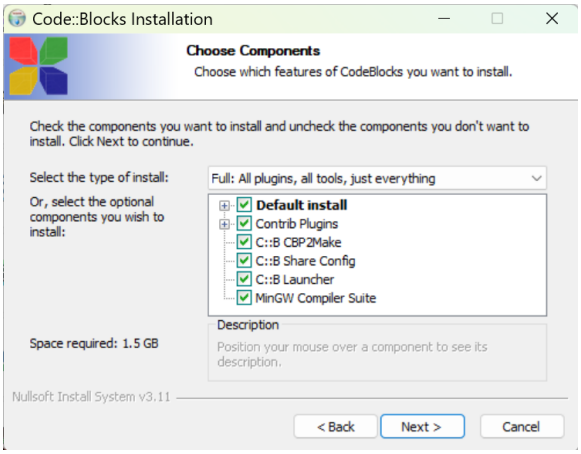


Figure 6 - The "Choose Components" screen

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
- **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 Window applications without relying on third-party runtime libraries.

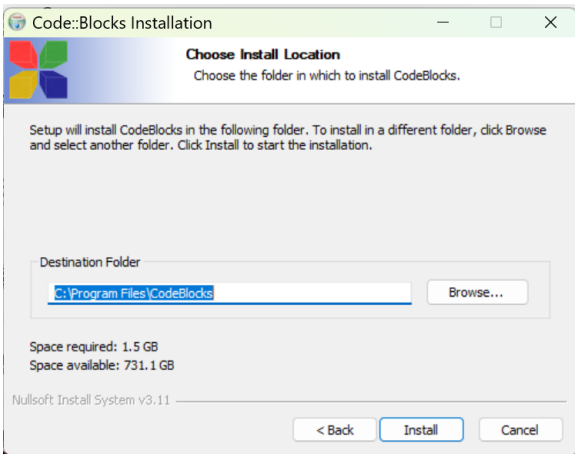


Figure 7 - Installation location on your PC

I usually take the default location.

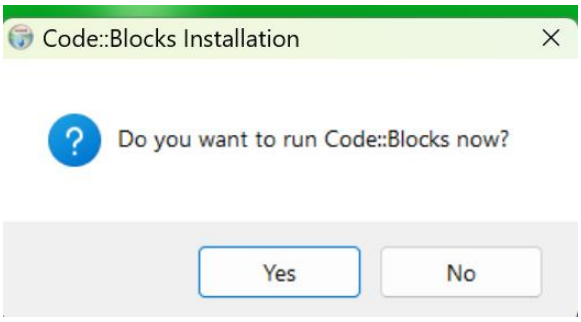


Figure 8 - Prompt to run Code::Blocks now?

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

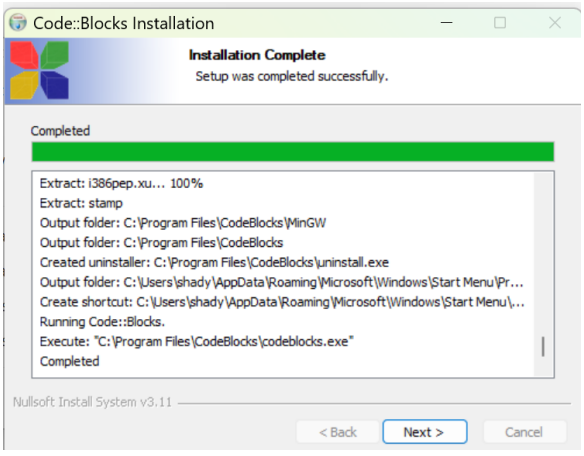


Figure 9 - Code::Blocks installation completed dialog

I clicked on “Next >”.

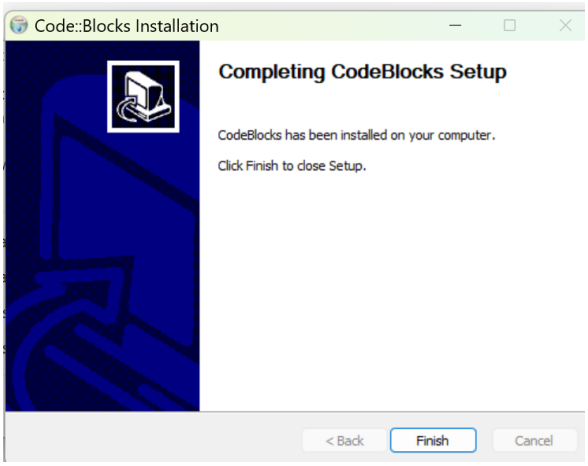


Figure 10 - The final Code::Blocks setup screen

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

Starting up Code::Blocks

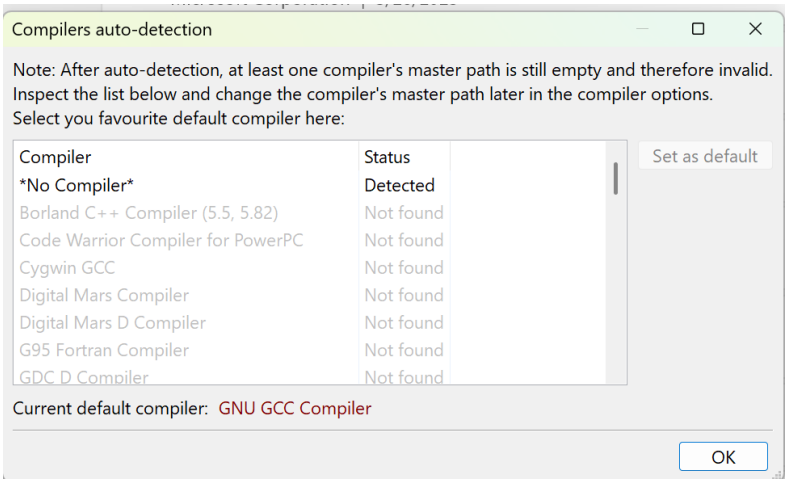


Figure 11 - Code::Blocks lists all the compilers it detected

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.

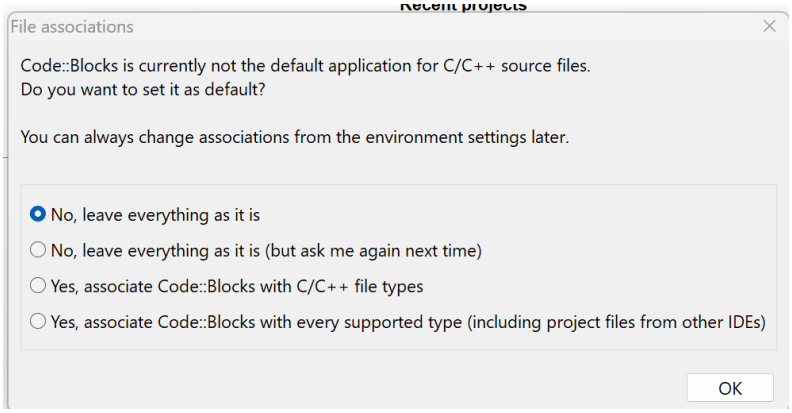


Figure 12 - Option to update file association

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

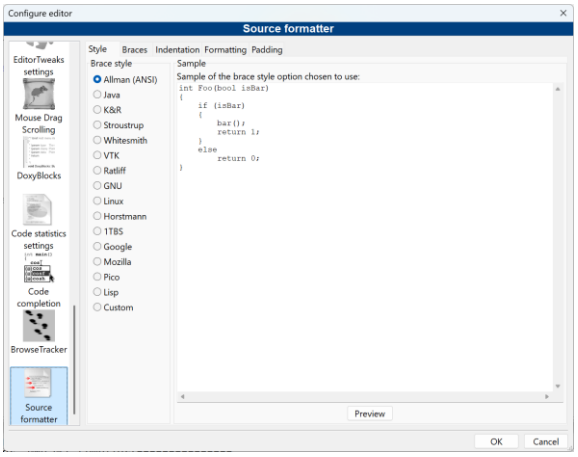


Figure 13 - Source formatter

- Select K&R

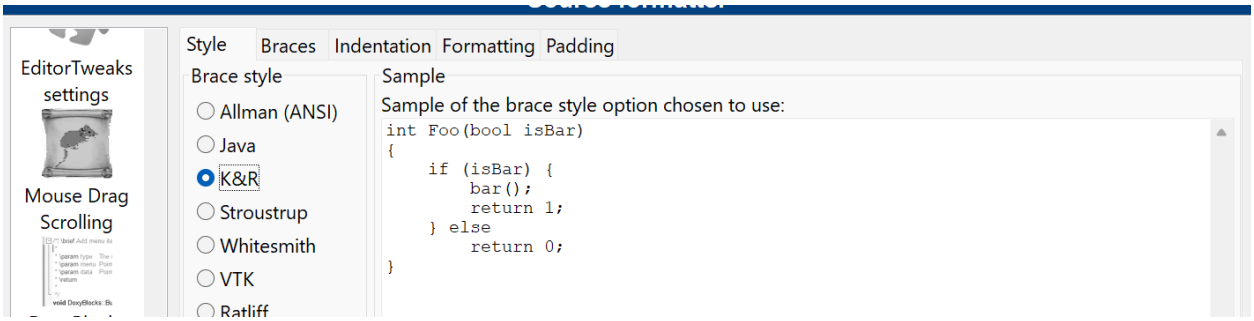


Figure 14 - Using K&R

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.

Starting a Project with Code::Blocks

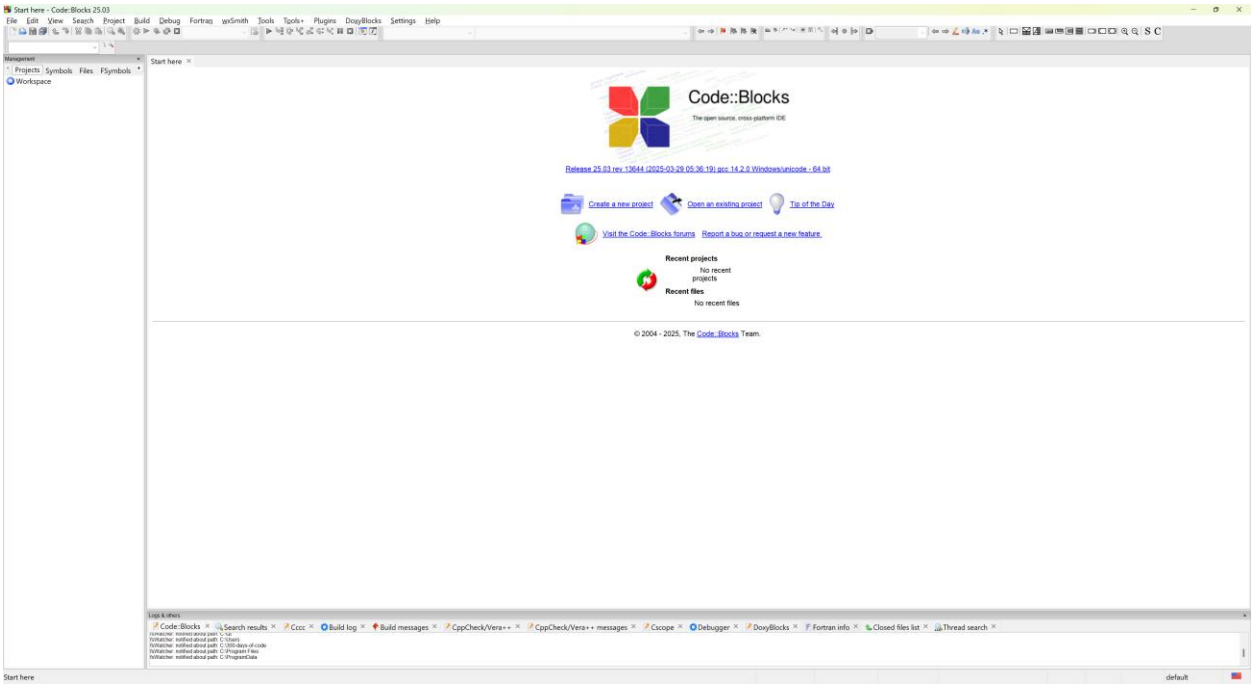


Figure 15 - The starting Code::Blocks screen

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

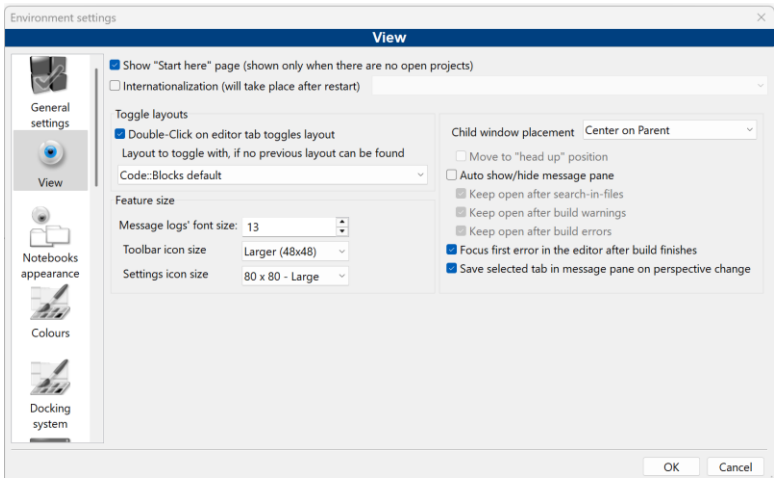


Figure 16 - Increasing the toolbar icon sizes

- Click "OK"
- Select File ➔ Save Everything
- Restart Code::Blocks

I also increased the font size used by:

- Select Settings ➔ Editor

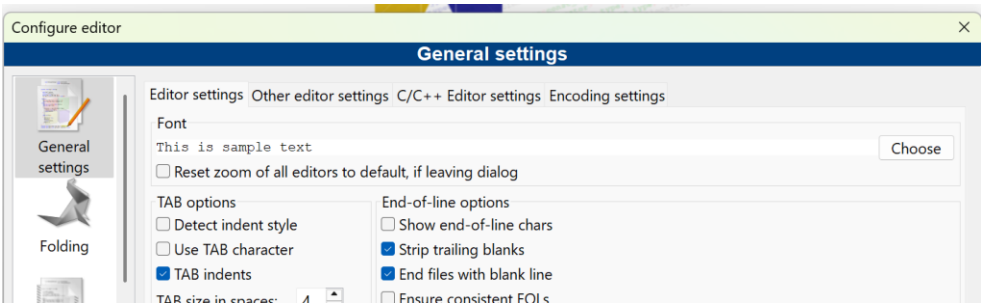


Figure 17 - Updating font-size

- Click on "Choose" button

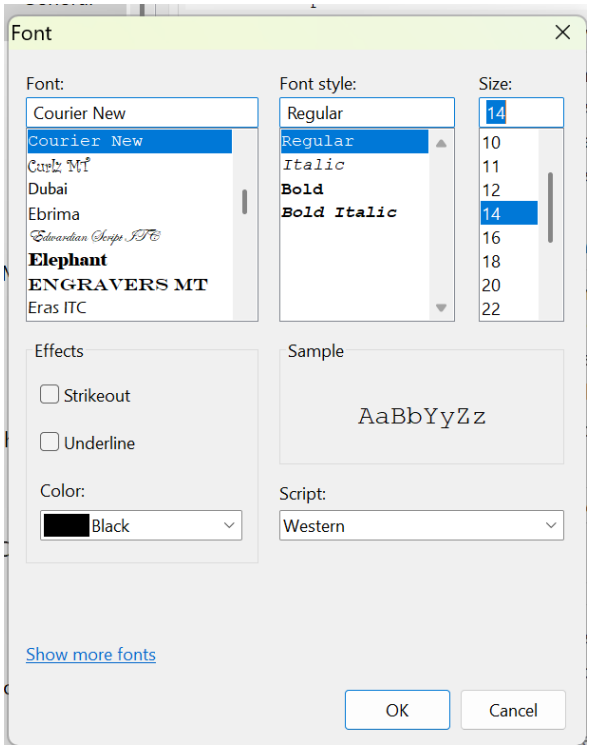


Figure 18 - Select desired font size

- I selected 14 and clicked on “OK”

Creating the Project

- Click on “Create a new project” link

Figure 19 - Creating a new Project

- Select “Console Application”

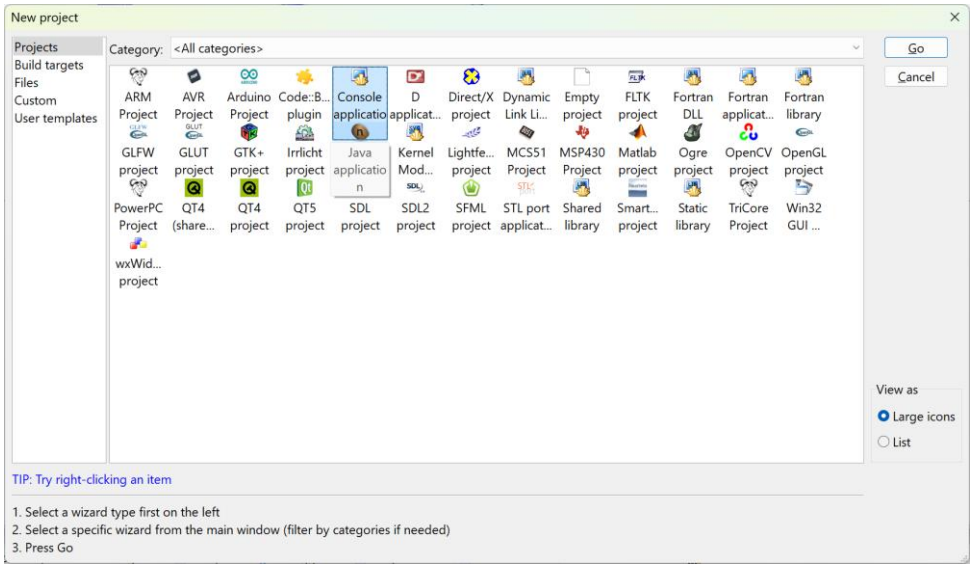


Figure 20 - Creating our first console application

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

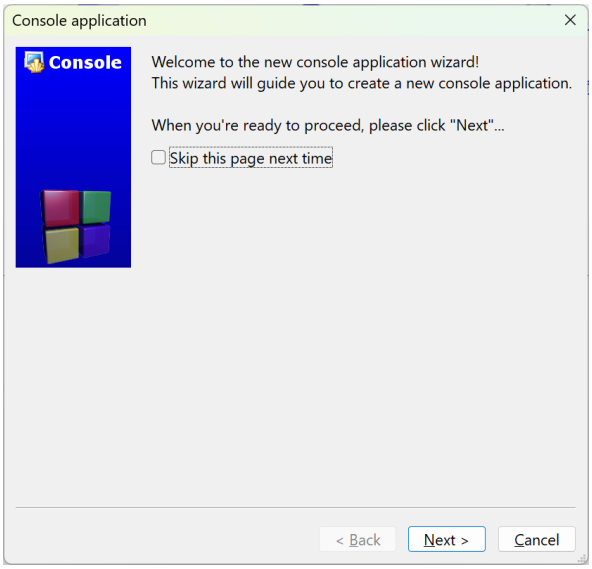
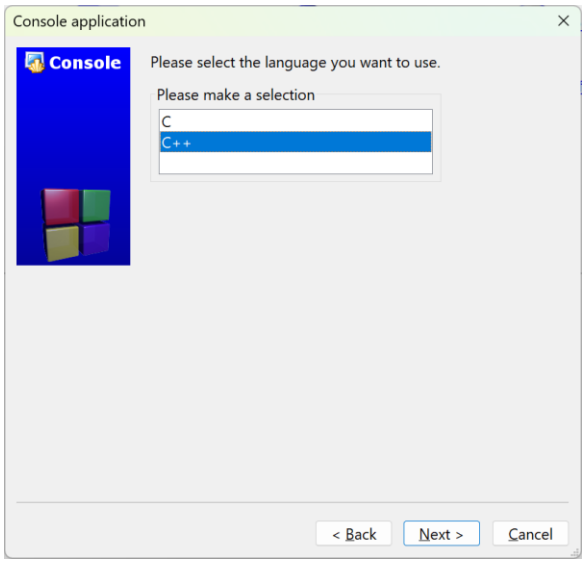


Figure 21 - The Console application wizard starting...

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



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

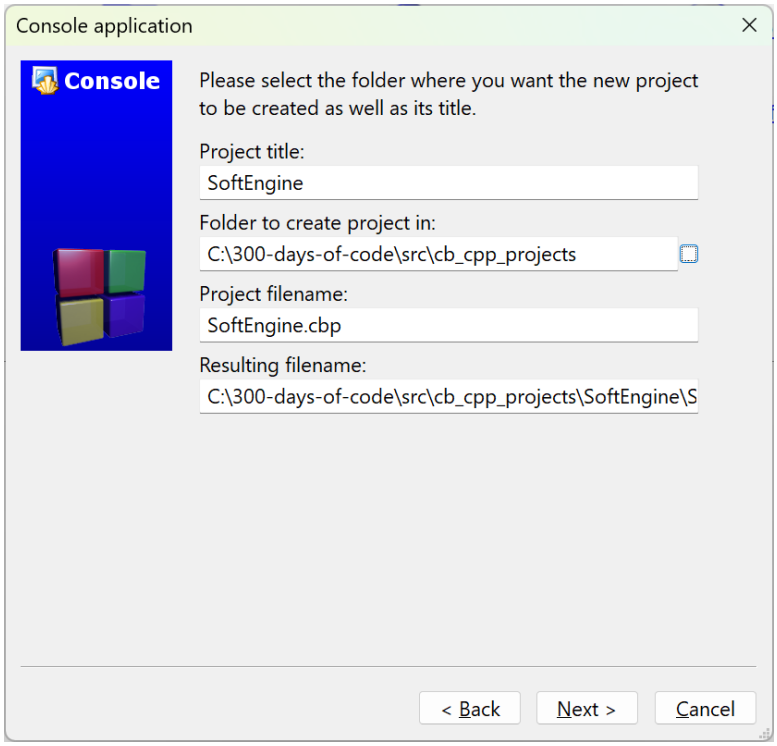


Figure 22 - Project information

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.

- Click on “Next >”

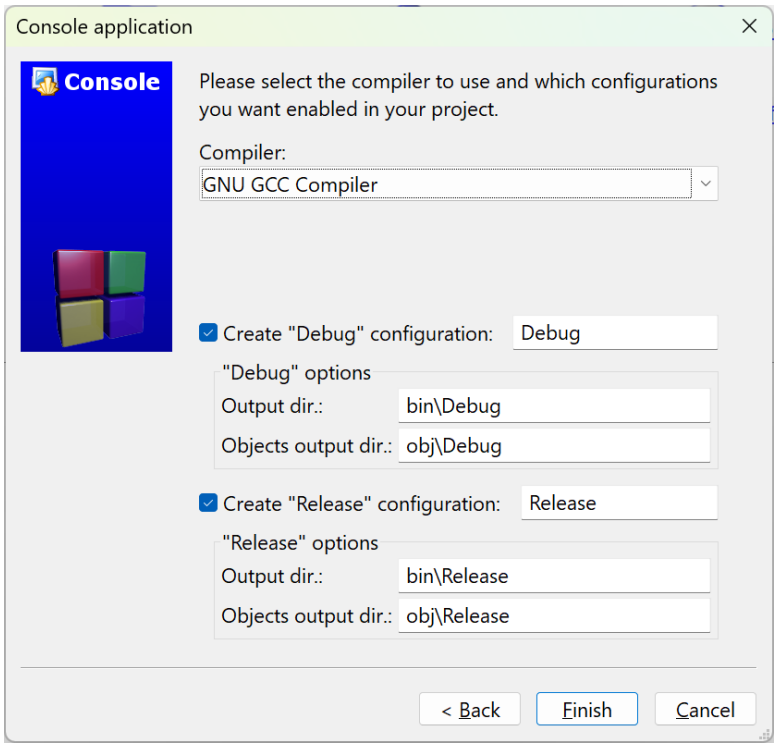


Figure 23 - Final step in creating our game engine project

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

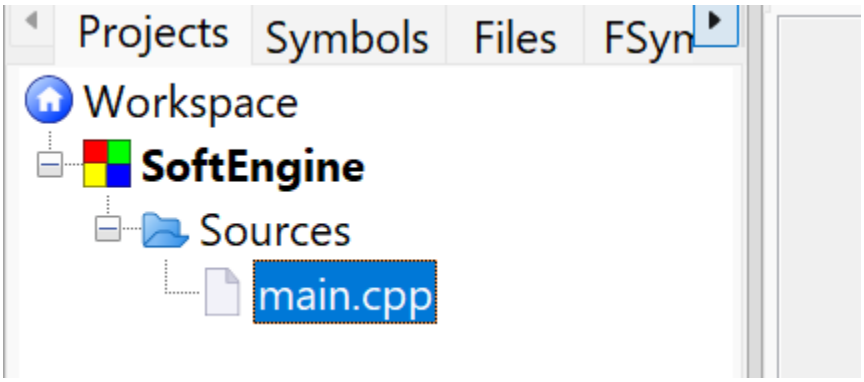


Figure 24 - Default main.cpp is created

- 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:

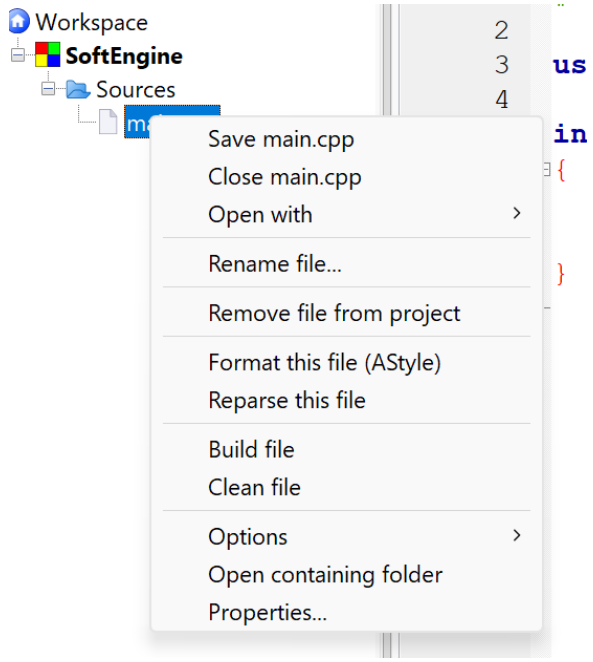


Figure 25 - Context menu for file

- Enter the name `Main.cpp`:

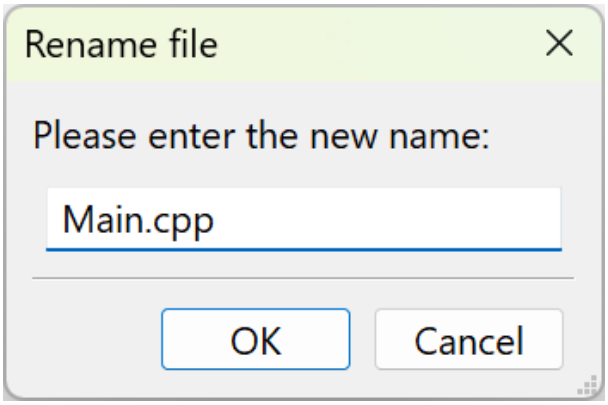


Figure 26 - Rename dialog box

- Click “OK”

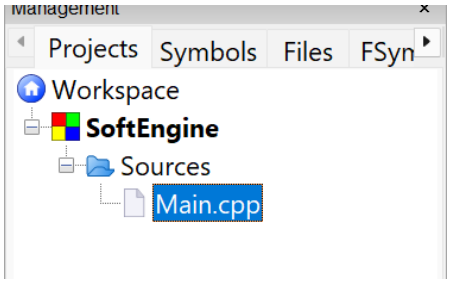


Figure 27 - Update name in Workspace

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:

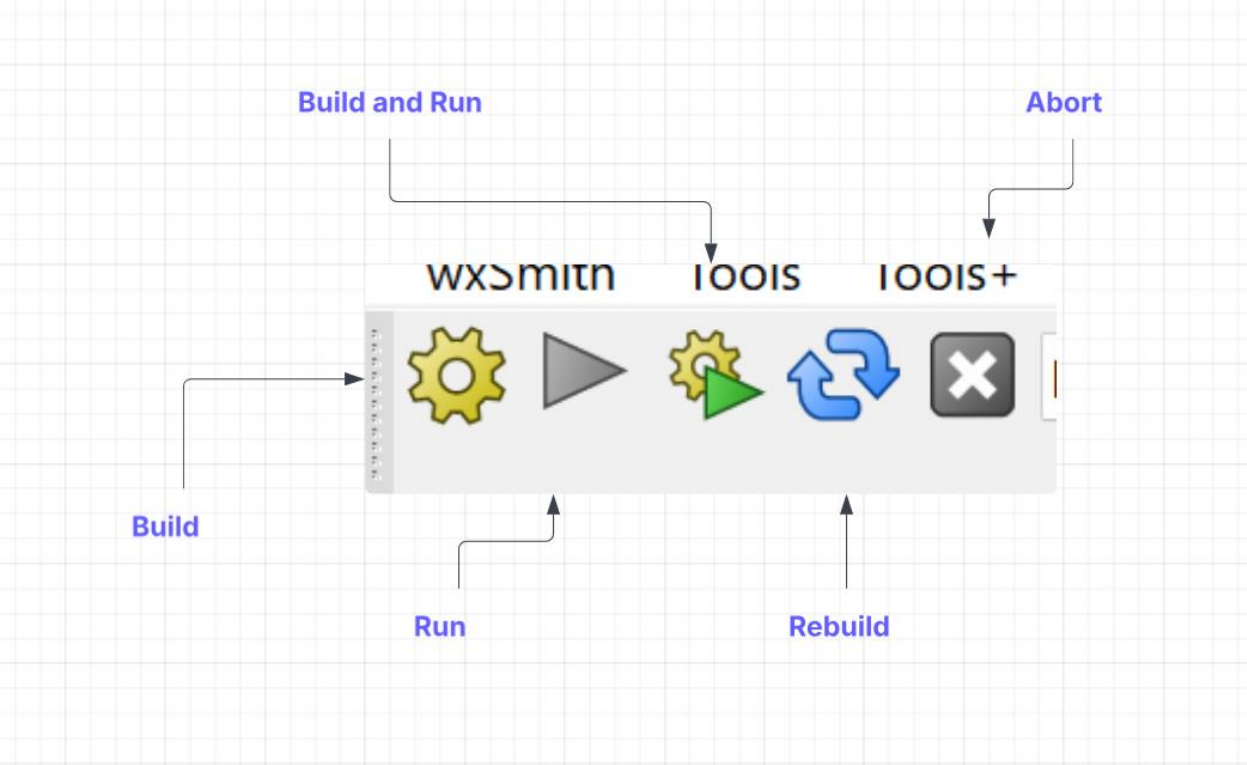


Figure 28 - Common operations

- **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 you 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.

Let's Build and run!

Our Main.cpp code is:

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world!" << endl;
    return 0;
}

```

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

- Click on the "Build and Run" icon

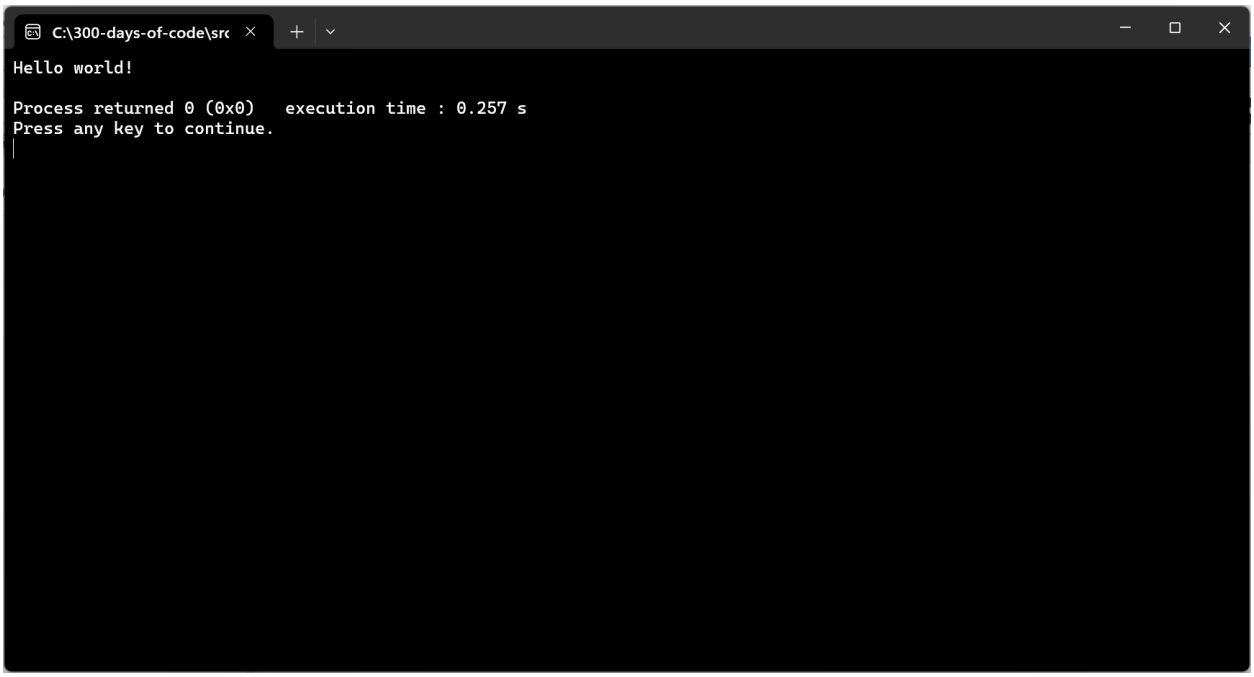


Figure 29 - Result of building and running our Main.cpp

- 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

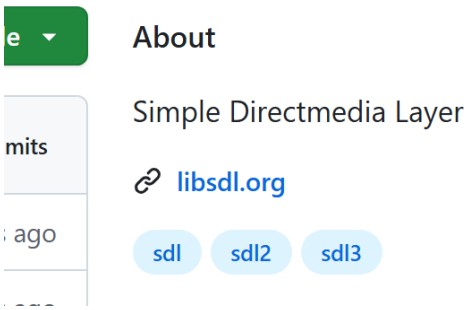


Figure 30 - Versions of sdl on Github

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!

- 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?

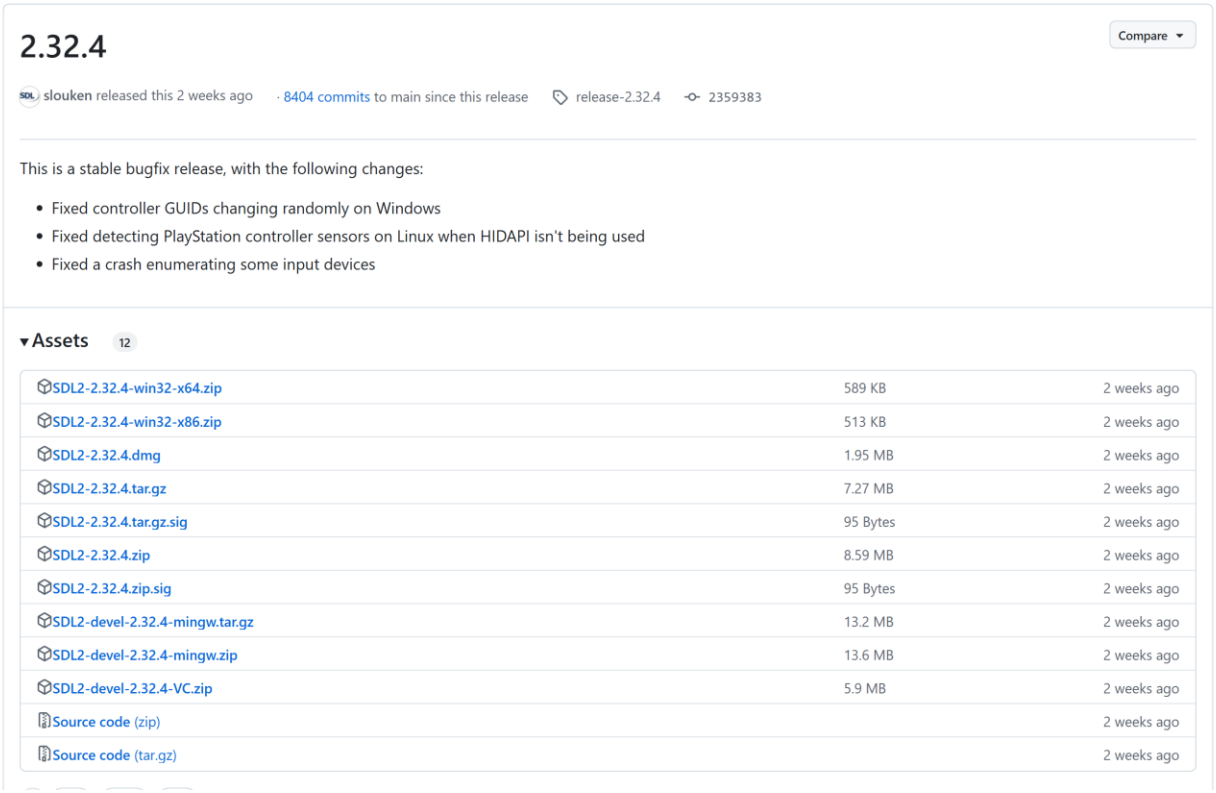


Figure 31 - Versions of SDL3 to choose from

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 file[2] to D:\SDL2_dev:

[2] I use 7-zip to manage my zip files

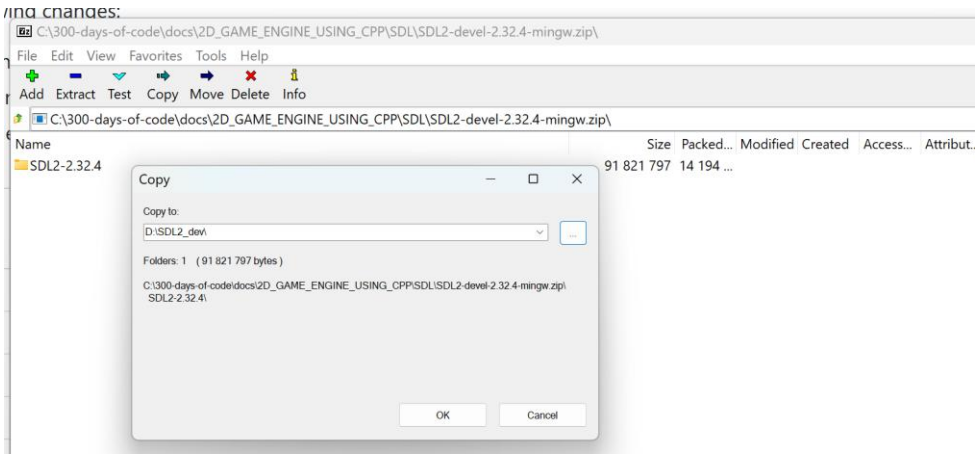


Figure 32 - Unzipping the SDL2 version

You will see:

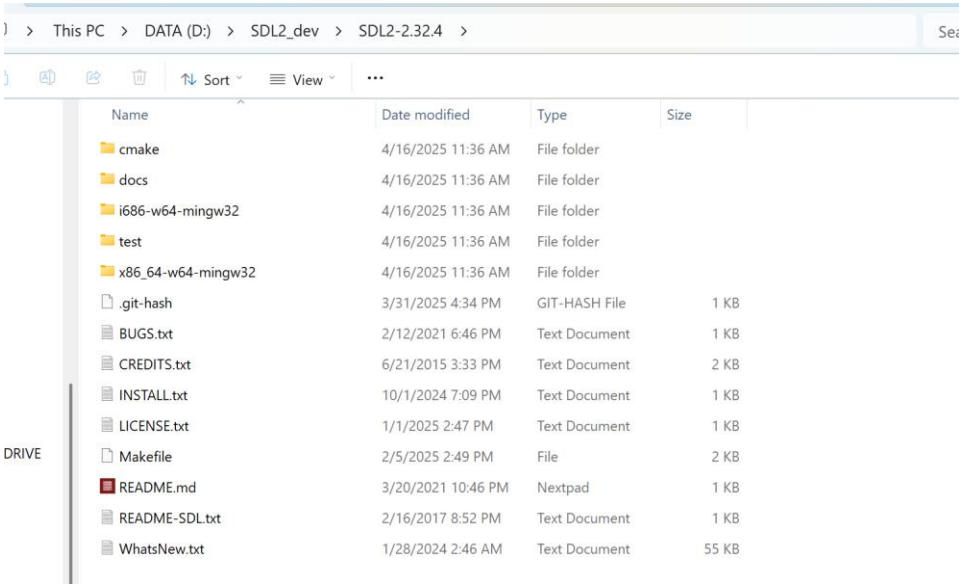


Figure 33 - Unzipped SDL2 file

- 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

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

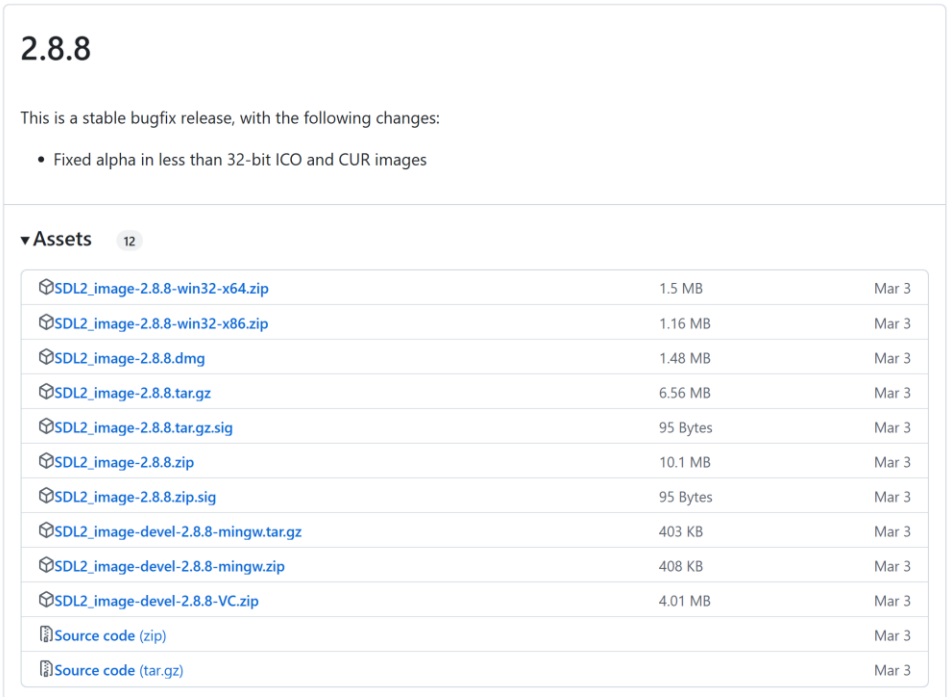


Figure 35 - SDL2_image choices

- 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:

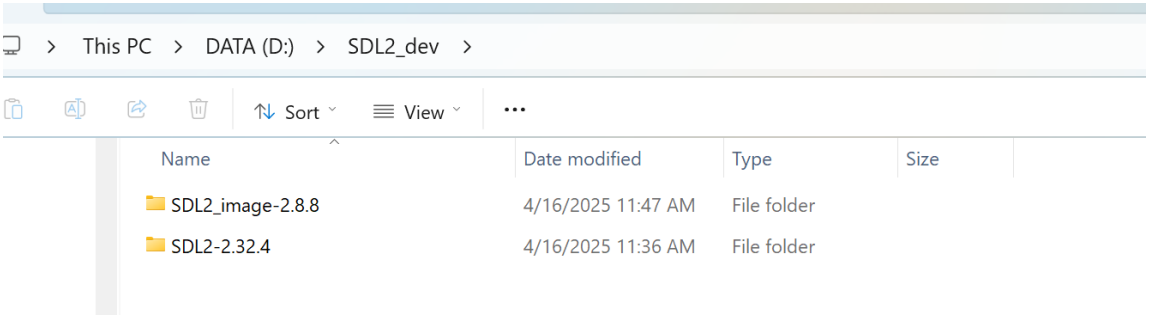


Figure 36 - Top-level folders

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”

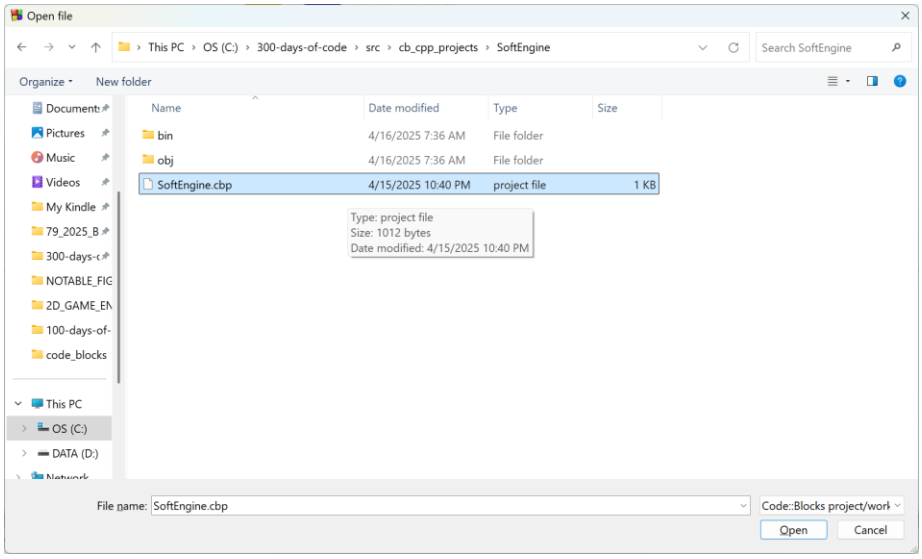


Figure 37 - Opening an existing project

- Go to Settings ➔ Compiler…

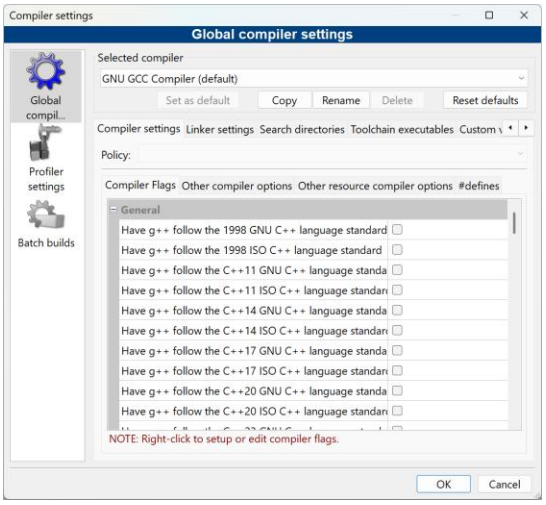


Figure 38 - Global compiler settings

- Click on the “Linker settings” tab

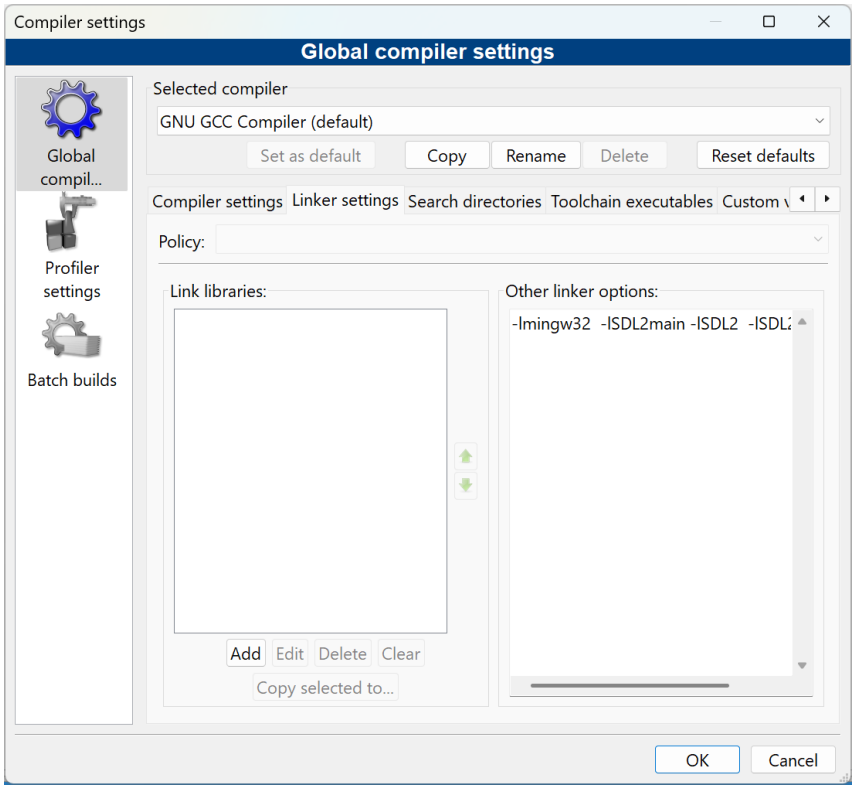
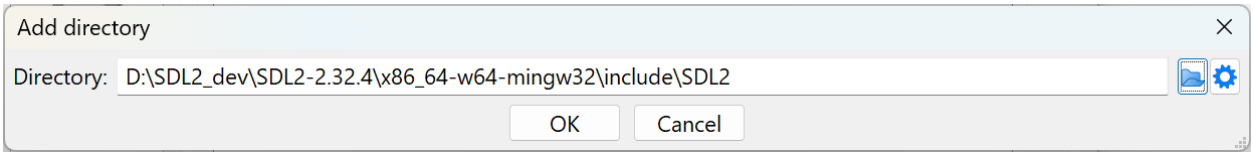


Figure 39 - Adding linker options

- 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



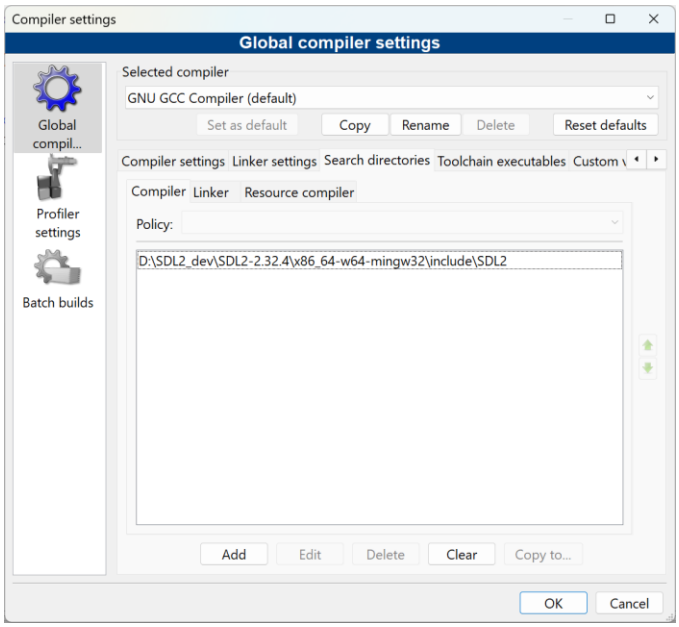
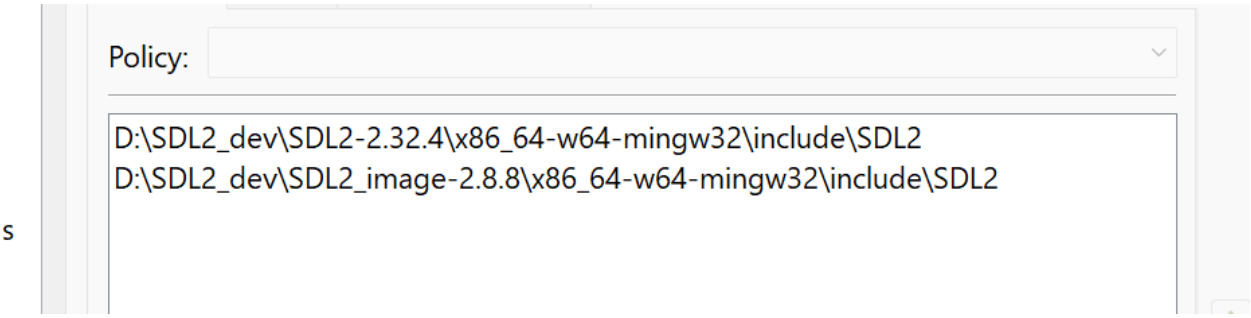


Figure 40 - Adding SDL3 include folder for the compiler

- Add the SDL2_image include as well



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

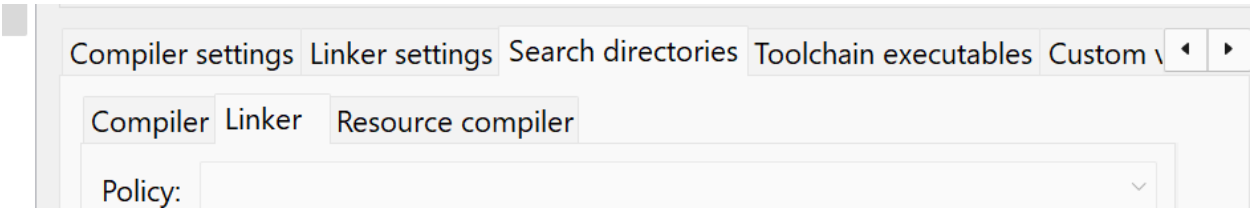
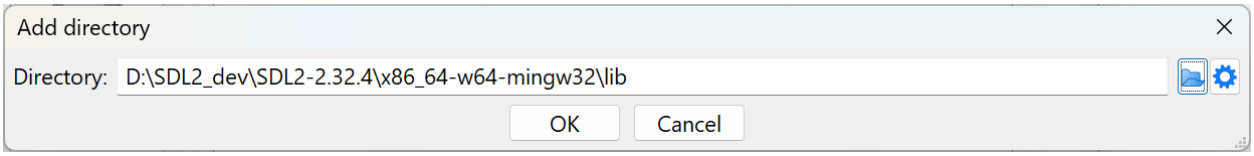


Figure 41 - Linker sub-tab

- Navigate to the SDL2 lib folder



- Click on OK
- Add lib folder under SDL_Image directory

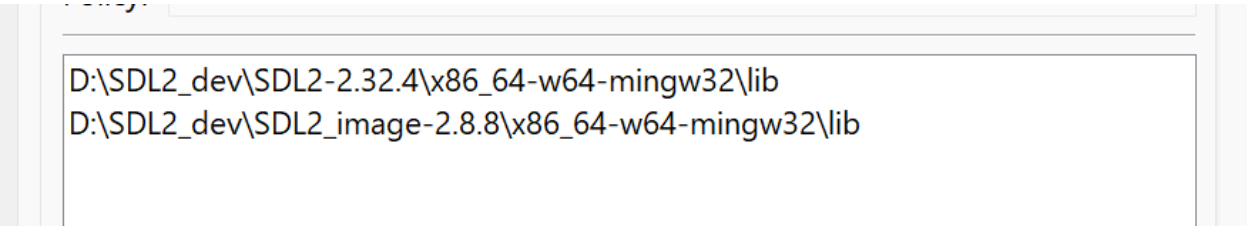


Figure 42 - SDL2 and SDL2_image lib folders for the linker

Testing the setup

- Update the code as follows:

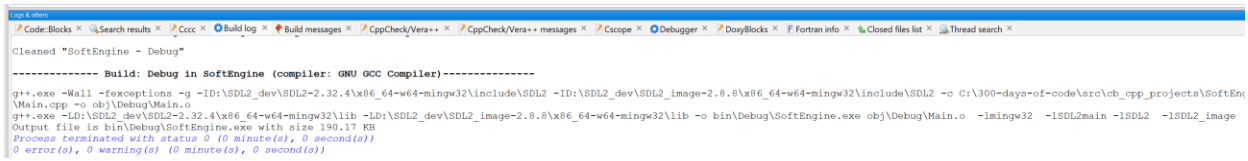
```
#include <iostream>
#include "SDL.h"

using namespace std;

int main(int argc, char** argv)
{
    cout << "Hello world!" << endl;
    return 0;
}

```

- 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:

```
#include <iostream>
#include "SDL.h"

using namespace std;
```

```
 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:

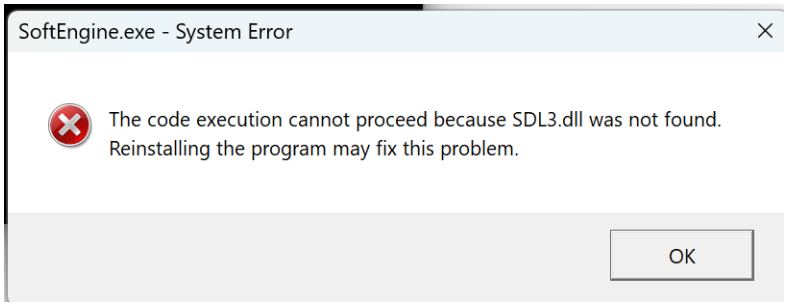


Figure 43 - Application failed to find SDL3.dll file

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

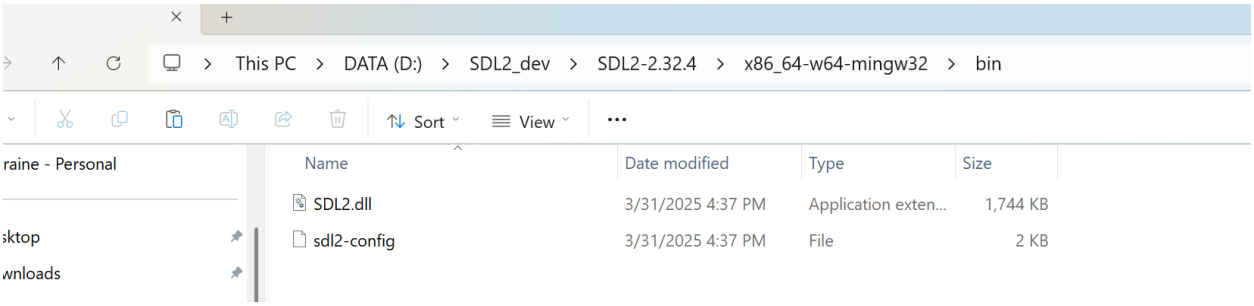
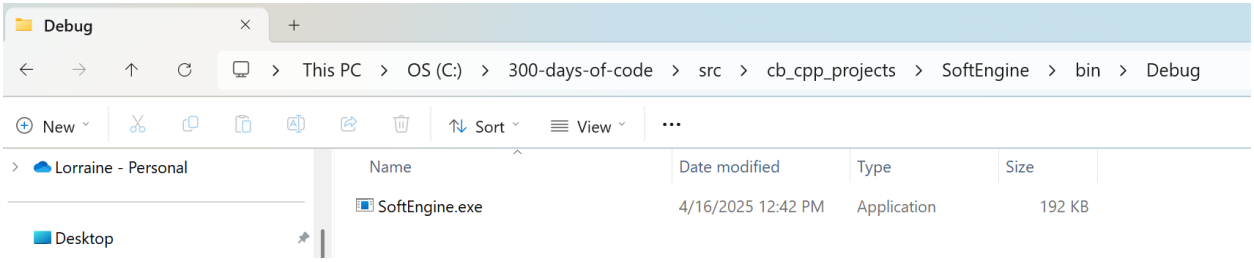
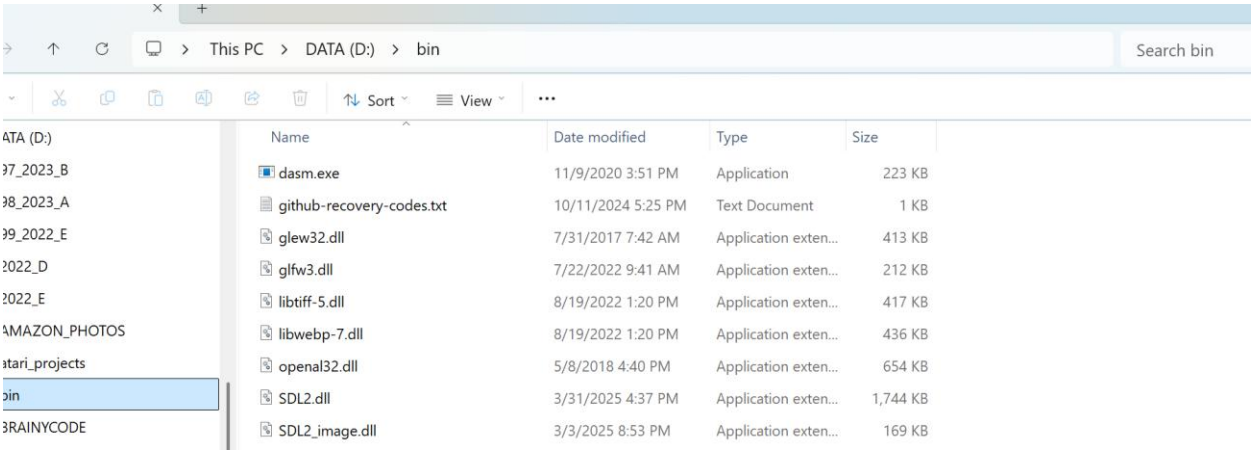


Figure 44 - The location of the SDL2.dll file

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



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:

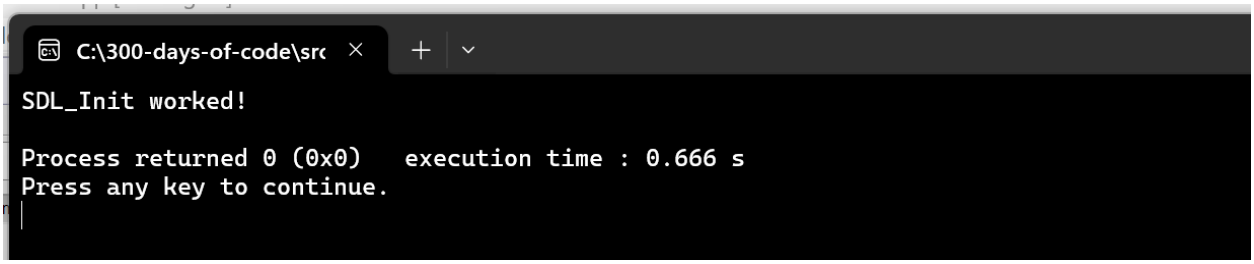


Figure 45 - Program with SDL component running ran successfully

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 .

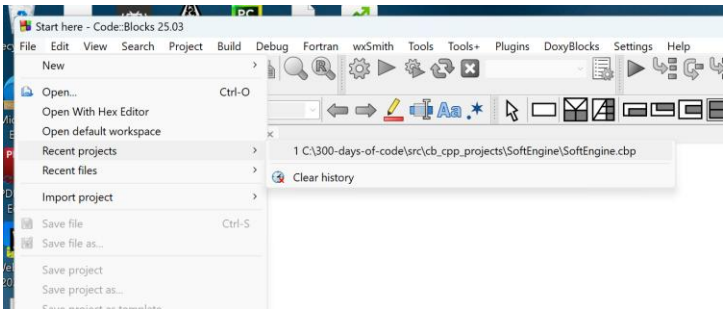


Figure 46 - Selecting a recent project

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

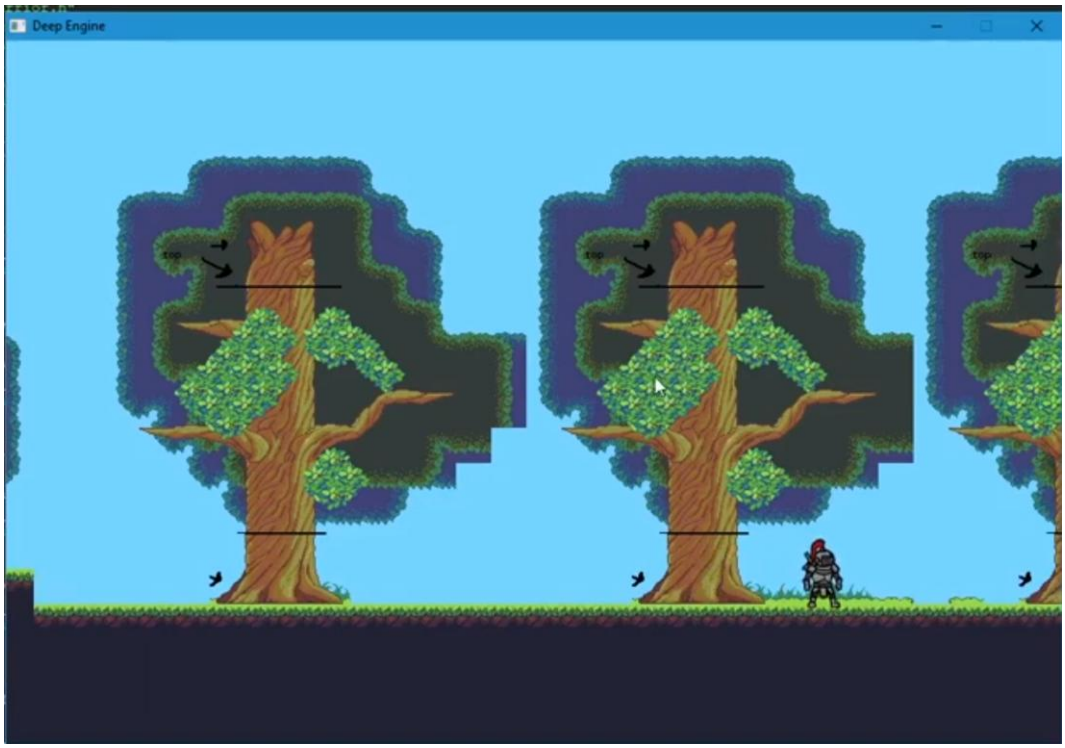


Figure 47 - The final look of our 2D Game Engine

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:

- 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.

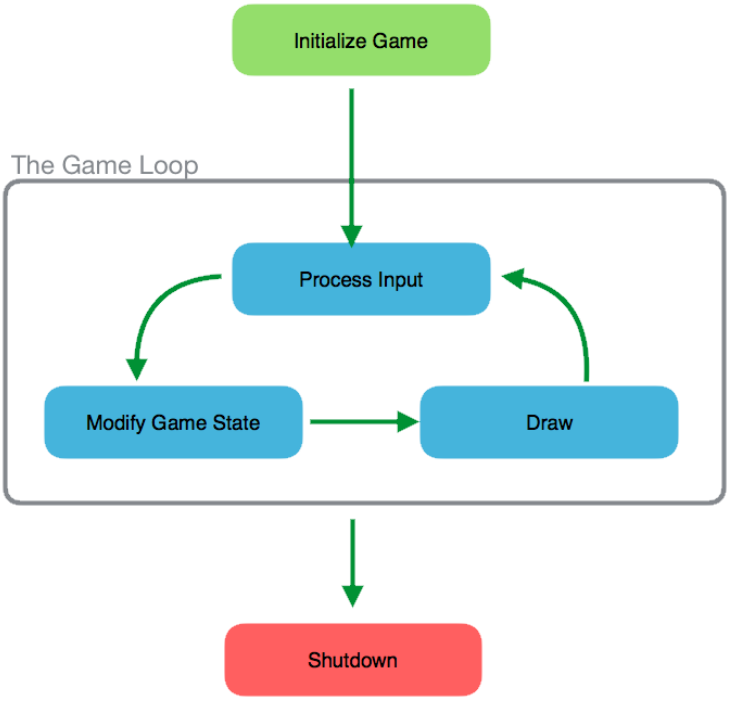


Figure 48 - A diagram of the "Game Loop"

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…

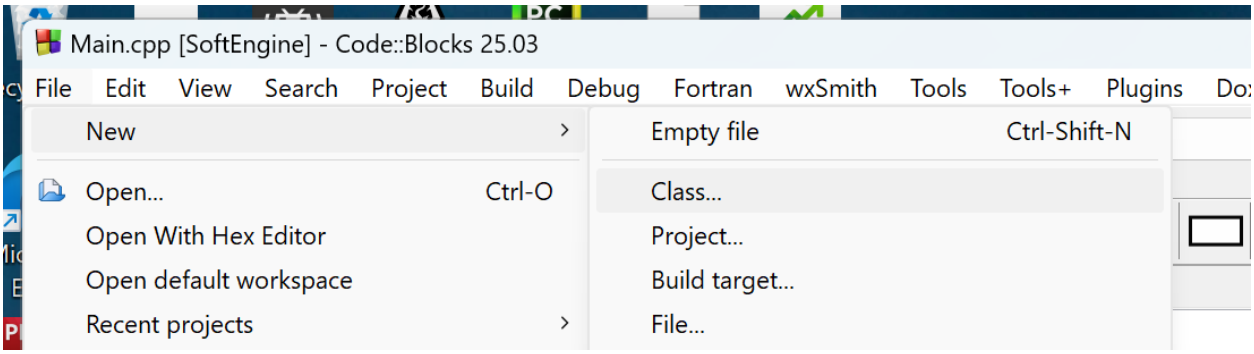


Figure 49 - Adding a new class to the project

- Fill in the “Create new class” dialog as shown:

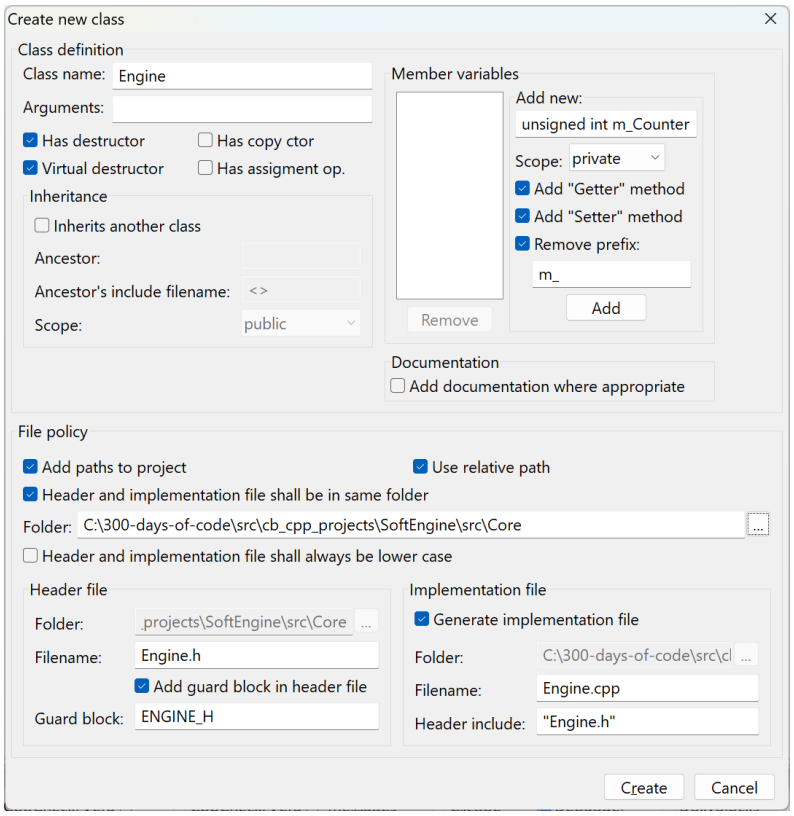


Figure 50 - Creating our Engine.cpp and Engine.h files

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”

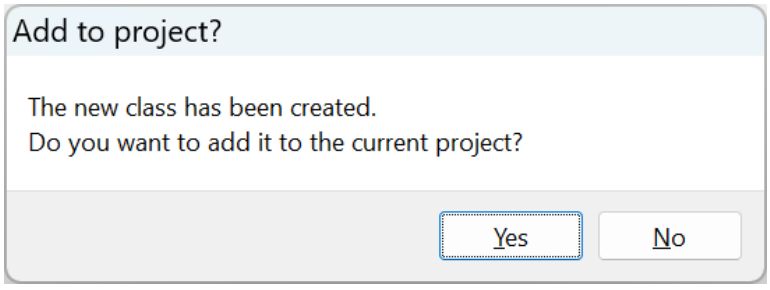


Figure 51 - Add to project prompt?

- Click “Yes” to the above prompt.

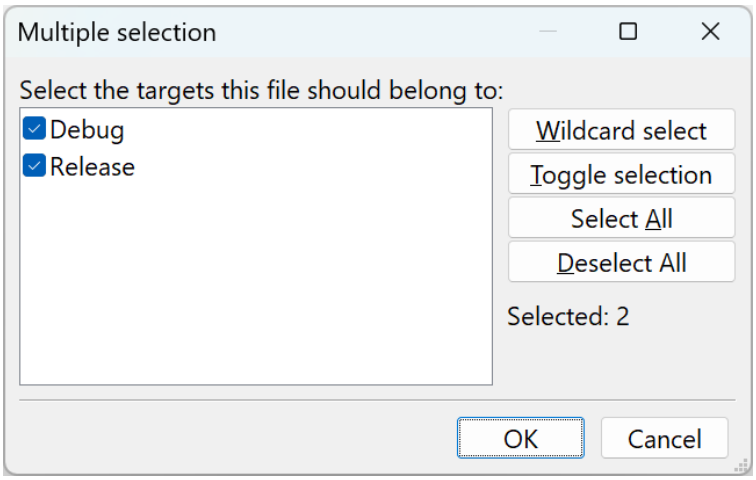


Figure 52 - Add to Debug and Release prompt

- Click “OK” to the above prompt.

The project will appear as:

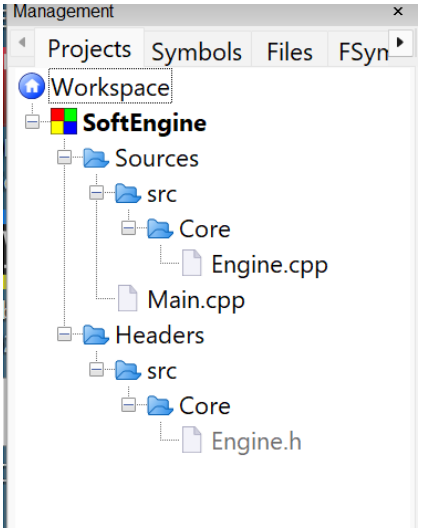
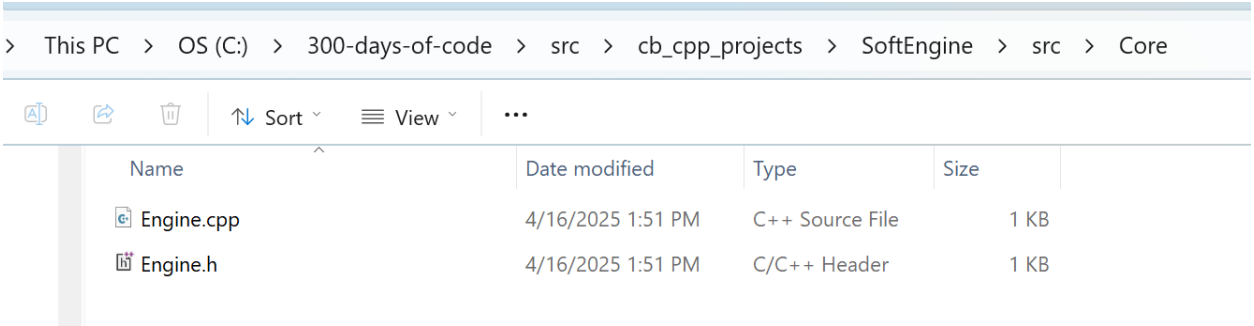


Figure 53 - Project view of the files

In fact, if you examine your folders:



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".

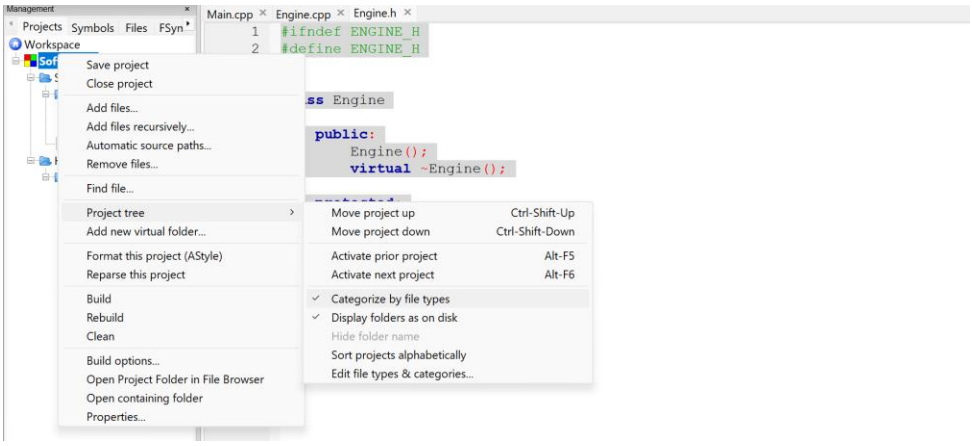
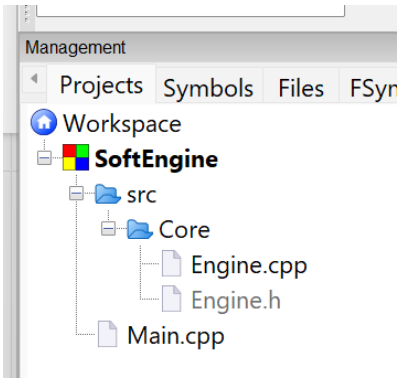


Figure 54 - Viewing the files as in actual folders

- 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. }
```

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

Engine.h (initial):

```
Engine.h (initial):
#ifndef ENGINE_H
#define ENGINE_H


class Engine
{
    public:
        Engine();
        virtual ~Engine();

    protected:

    private:
};

#endif // ENGINE_H

```

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:

```
#include <iostream>
#include <memory>

class Singleton {
private:
    static std::unique_ptr<Singleton> instance; // Static pointer to the instance
    Singleton() {}                              // Private constructor

public:
    Singleton(const Singleton&) = delete;      // Prevent copying
    Singleton& operator=(const Singleton&) = delete; // Prevent assignment

    static Singleton& getInstance() {
        if (!instance) {
```

```
            instance = std::make_unique<Singleton>();
        }
        return *instance;
    }

    void displayMessage() {
        std::cout << "Singleton instance accessed!\n";
    }
};

std::unique_ptr<Singleton> Singleton::instance = nullptr; // Initialize the static pointer

int main() {
    Singleton& singleton = Singleton::getInstance();
    singleton.displayMessage();

    return 0;
}

```

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.

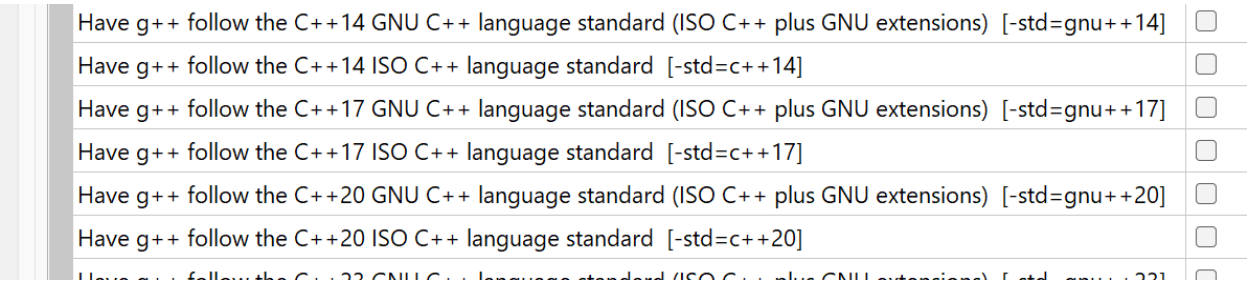
We only want to have one instance of the Engine class.

Setting the compiler to use C++ 17

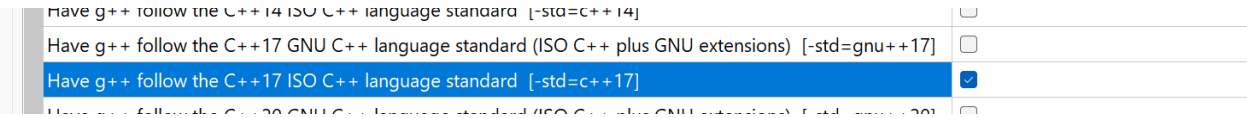
We probably should have done this earlier (in the first video) but we need to ensure that we use C++ 17 to match the presenter's version.

- Click on Settings ➔ Compiler…

We have several choices here:



The one that matches is:



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 != nullpter) ? 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 on 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 != nullpter) ? 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.

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:

```
#ifndef ENGINE_H
#define ENGINE_H


class Engine
{
    public:
        static Engine* GetInstance() {
            return s_Instance = (s_Instance != nullptr) ? s_Instance : new Engine();
        }

        bool Init();
        bool Clean();
        void Quit();

        void Update();
        void Render();
        void Events();

        inline bool isRunning() {
            return m_IsRunning;
        }

    protected:

    private:
        Engine();
        static Engine* s_Instance;
        bool m_IsRunning;

};

#endif // ENGINE_H

```

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…”

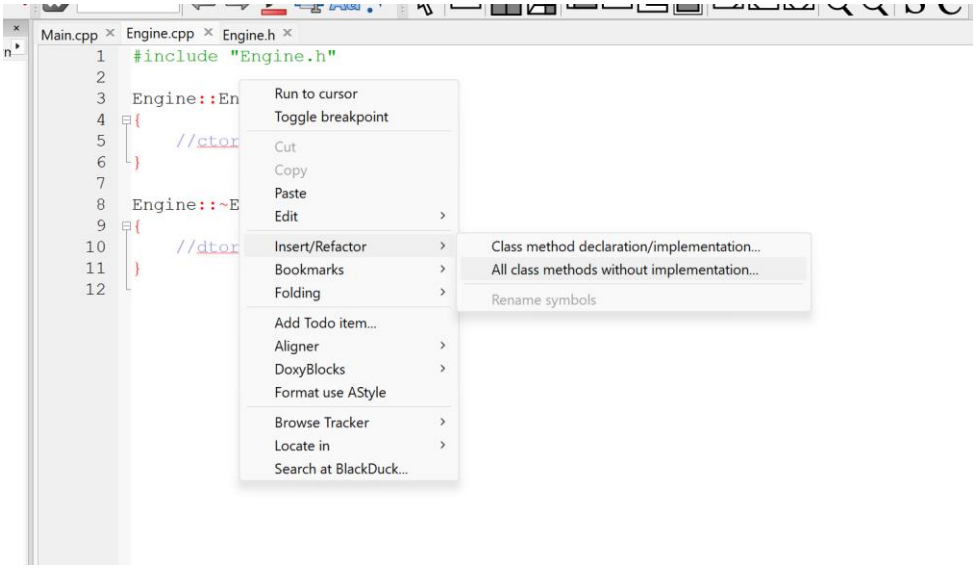


Figure 55 - Adding missing class methods

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

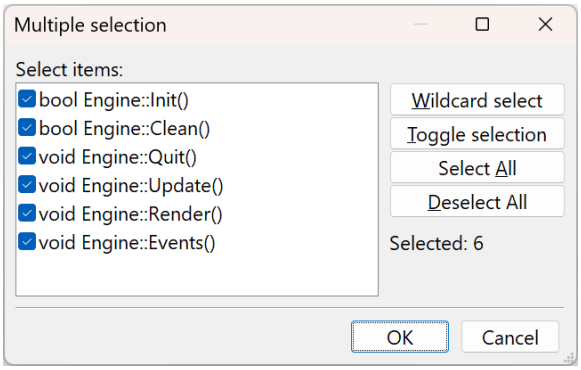


Figure 56 - Selecting the class methods to insert

- Click “OK”

Engine.cpp:

```
#include "Engine.h"

Engine::Engine()
{
    //ctor
}

bool Engine::Init()
{

}

bool Engine::Clean()
```

```
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:

- 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.
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.

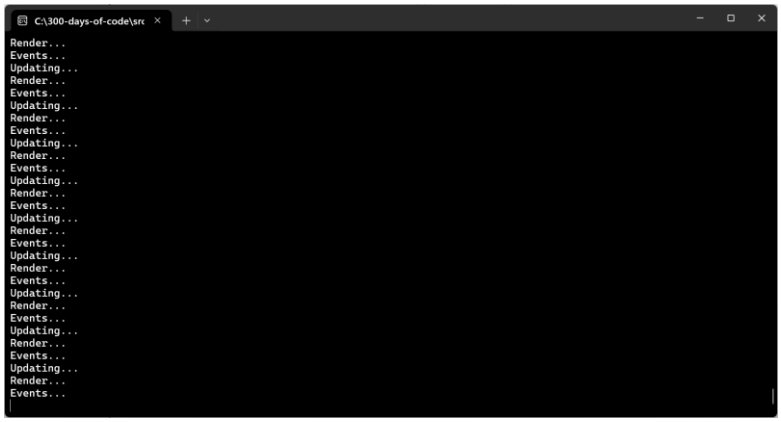


Figure 57 - Running our game loop

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()`.

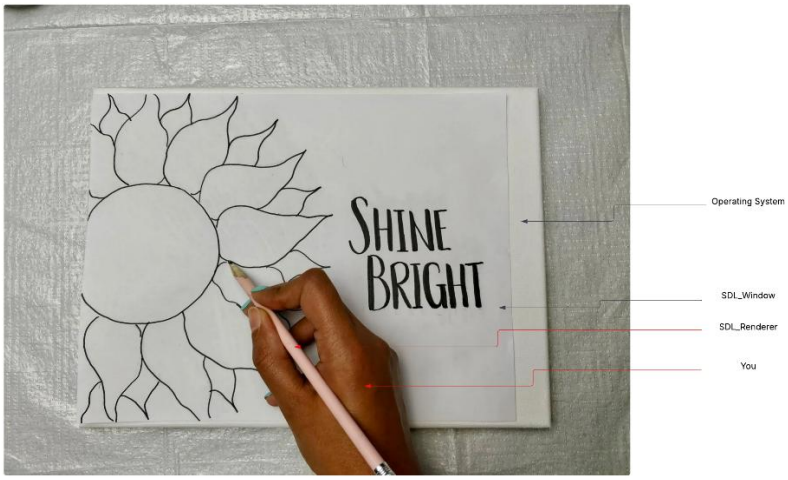
3. Create SDL Window and Renderer

This video starts the journey of filling in the components of game loop by

- Initializing SDL
- Creating our game window with SDL
- Creating the ability to draw in our window
- How to end our game

We cover/edit the key Engine.cpp methods in this video.

How I view things



I like to view drawing to screen by knowing that the operating system (OS) is the conductor of a computer's orchestra – it manages and coordinates all the hardware and software to ensure harmony. SDL libraries makes requests to the OS for resources. The paper represents the window (SDL_Window) you will create to draw on. You cannot draw on it directly. You obtain permission to draw by requesting a pencil or in our case an SDL_Renderer. It is through the SDL_Renderer that you draw the graphics in your window. You the do your own orchestration for your game application.

Using SDL in our Engine

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.

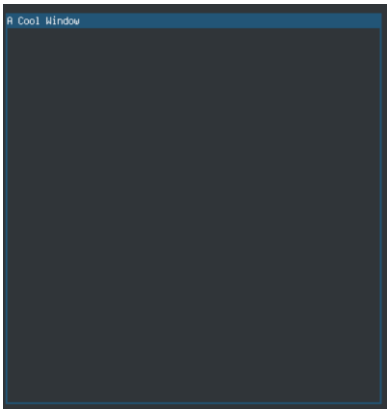


Figure 58 - An example of a Window (SDL_Window) from: https://jakob.space/blog/sdl-tutorial-part-0x00.html

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`.
- **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 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**:
    - 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:

```
#define SCREEN_WIDTH 960
#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

```
bool Engine::Init()
{
     if (SDL_Init(SDL_INIT_VIDEO) != 0 && IMG_Init(IMG_INIT_JPG | IMG_INIT_PNG) != 0) {
        SDL_Log("Failed to initialize SDL: %s", SDL_GetError());
        return false;
     }

     return m_IsRunning = true;
}
```

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:

```
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

- 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

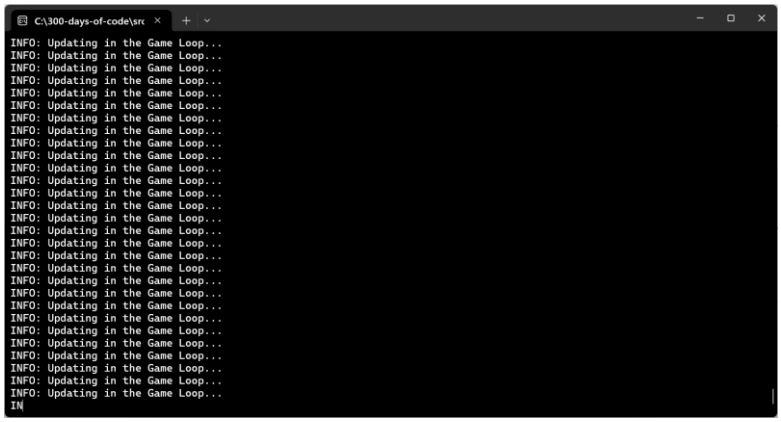


Figure 59 - Running the program with SDL Initialization

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:

```
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_WINDOPOS_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 */
16.     SDL_WINDOW_ALLOW_HIGHDPI = 0x00002000,      /**< window should be created in high-DPI mode
if supported.
17.                                                     On macOS NSHighResolutionCapable must be
set true in the
18.                                                     application's Info.plist for this to have
any effect. */
19.     SDL_WINDOW_MOUSE_CAPTURE    = 0x00004000,   /**< window has mouse captured (unrelated to
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.

Note: You should use `SDL_DestroyWindow` to close or return resources created from a `SDL_CreateWindow`.

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;
```

Note: You should return or close an `SDL_Renderer` with `SDL_DestroyRenderer()`.

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,
SCREEN_WIDTH, SCREEN_HEIGHT, 0);
10.     if (m_Window == nullptr) {
11.          SDL_Log("Failed to create window: %s", SDL_GetError());
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.
```

Adding a Background Color to our Application

Now that we have the “pencil” or `m_Renderer` set up. We need to update the `Engine::Render()` to make the background for the window.

```
1. void Engine::Render()
2. {
3.     SDL_SetRenderDrawColor(m_Renderer, 124, 218, 254, 255);
4.     SDL_RenderPresent(m_Renderer);
5. }
```

RGBA

One way to represent a color is to use RGBA. RGBA stands for **Red, Green, Blue, and Alpha**, which represents a color model used in graphics and digital imaging. Here's what each component means:

1. **Red (R)**: The intensity of the red color channel (typically from 0 to 255).
2. **Green (G)**: The intensity of the green color channel (also 0 to 255).
3. **Blue (B)**: The intensity of the blue color channel (again, 0 to 255).
4. **Alpha (A)**: The transparency or opacity of the color. A value of 0 means completely transparent, while 255 (or 1.0 in some systems) means completely opaque.

For example, RGBA(255, 0, 0, 0.5) would represent a semi-transparent red color.

RGBA is widely used in graphics programming, including web development (CSS), image editing, and game development, as it allows precise control over both color and transparency. Let me know if you'd like to explore it further!

Here is an example of displaying a red color using various alpha values:

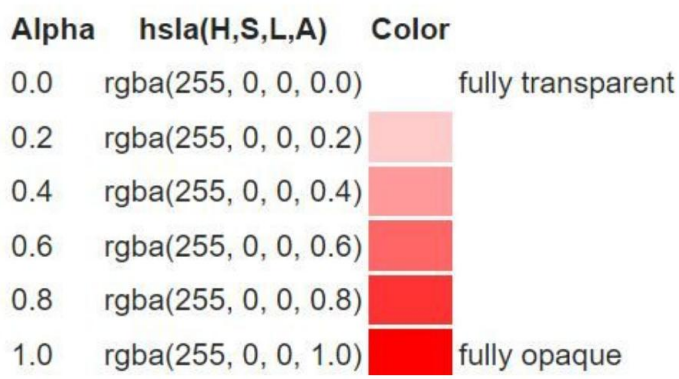


Figure 60 - Illustrating RGBA - from https://www.browserstack.com/guide/how-to-use-css-rgba

SDL_SetRenderDrawColor

This function is equivalent to setting or selecting the color of the pencil you plan on drawing with on the screen.

Syntax:

```
1. int SDL_SetRenderDrawColor(SDL_Renderer * renderer,
2.                     Uint8 r, Uint8 g, Uint8 b,
3.                     Uint8 a);
```

Function Parameters

SDL_Renderer *	**renderer**	the rendering context.
Uint8	**r**	the red value used to draw on the rendering target.
Uint8	**g**	the green value used to draw on the rendering target.
Uint8	**b**	the blue value used to draw on the rendering target.
Uint8	**a**	the alpha value used to draw on the rendering target; usually SDL_ALPHA_OPAQUE (255). Use SDL_SetRenderDrawBlendMode to specify how the alpha channel is used.

Return Value

(int) Returns 0 on success or a negative error code on failure; call SDL_GetError() for more information.

Remarks

Set the color for drawing or filling rectangles, lines, and points, and for SDL_RenderClear().

In the code we use:

```
1. SDL_SetRenderDrawColor(m_Renderer, 124, 218, 254, 255);
```

The alpha value is 255 so it will be opaque. What color is RGB=(124, 218, 254)?

The color should appear as:

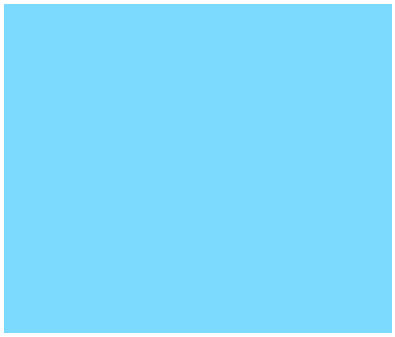


Figure 61 - The color specified to use

If you followed the steps in the video – we are not seeing this color because we have to draw something. Let's change the code to:

```
 1. void Engine::Render()
 2. {
 3.     // Set the draw color
 4.     SDL_SetRenderDrawColor(m_Renderer, 124, 218, 254, 255);
 5.
 6.     // Clear the screen with the draw color
 7.     SDL_RenderClear(m_Renderer);
 8.
 9.     // Present the renderer (update the window)
10.     SDL_RenderPresent(m_Renderer);
11. }
```

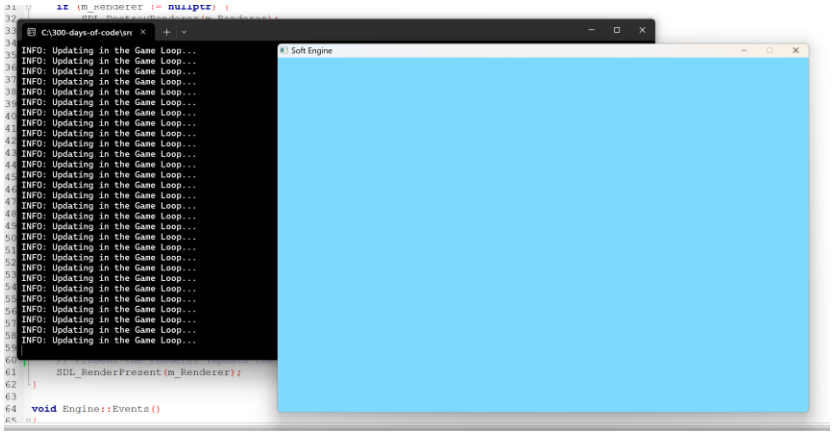


Figure 62 - Setting the screen to our draw color

SDL_RenderClear

This function is used to clear the current rendering target with the drawing color. In our case we use it to color the window background.

Syntax

```
1. int SDL_RenderClear(SDL_Renderer * renderer);
```

Function Parameters

SDL_Renderer *	**renderer**	the rendering context.

Return Value

(int) Returns 0 on success or a negative error code on failure; call SDL_GetError() for more information.

Remarks

This function clears the entire rendering target, ignoring the viewport and the clip rectangle.

SDL_RenderPresent

The `SDL_RenderPresent` update the screen with the current rendering values.

Syntax

```
1. void SDL_RenderPresent(SDL_Renderer * renderer);
```

Function Parameters

SDL_Renderer *	**renderer**	the rendering context.

Remarks

SDL's rendering functions operate on a backbuffer; that is, calling a rendering function such as SDL_RenderDrawLine() does not directly put a line on the screen, but rather updates the backbuffer. As such, you compose your entire scene and *present* the composed backbuffer to the screen as a complete picture.

Therefore, when using SDL's rendering API, one does all drawing intended for the frame, and then calls this function once per frame to present the final drawing to the user.

The backbuffer should be considered invalidated after each present; do not assume that previous contents will exist between frames. You are strongly encouraged to call SDL_RenderClear() to initialize the backbuffer before starting each new frame's drawing, even if you plan to overwrite every pixel.

Screen Buffering

This section is from: https://docs.oracle.com/javase/tutorial/extra/fullscreen/doublebuf.html

Suppose you had to draw an entire picture on the screen, pixel by pixel or line by line. If you were to draw such a thing directly to the screen, you would probably notice with much disappointment that it takes a bit of time. You will probably even notice visible artifacts of how your picture is drawn.

Figure 63 - The screen showing "tearing" [from: https://www.techspot.com/article/2192-screen-tearing-fix-pc-gaming/]

Rather than watching things being drawn in this fashion and at this pace, most programmers use a technique called *double-buffering*.

The traditional notion of double-buffering is fairly straightforward: create an off-screen image, draw to that image using the image's graphics object, then, in one step, call SDL_RenderPresent.

The screen surface is commonly referred to as the *primary surface*, and the off-screen image used for double-buffering is commonly referred to as the *back buffer*. The act of copying the contents from one surface to another is frequently referred to as a block line transfer, or *blitting* (blt is typically pronounced "blit" and shouldn't be confused with a BLT sandwich).

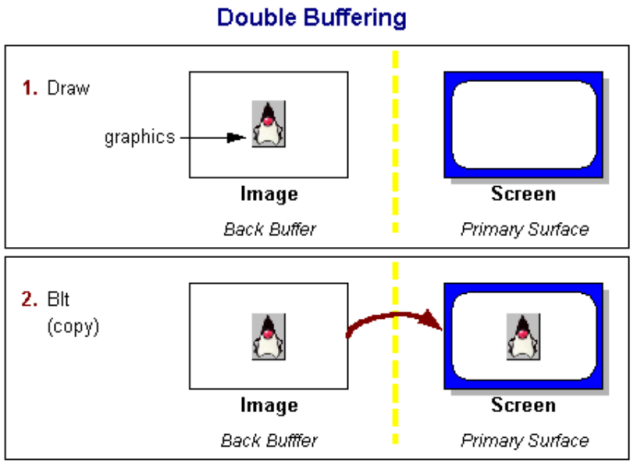


Figure 64 - Illustrating how all the drawing takes place in the backbuffer

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.

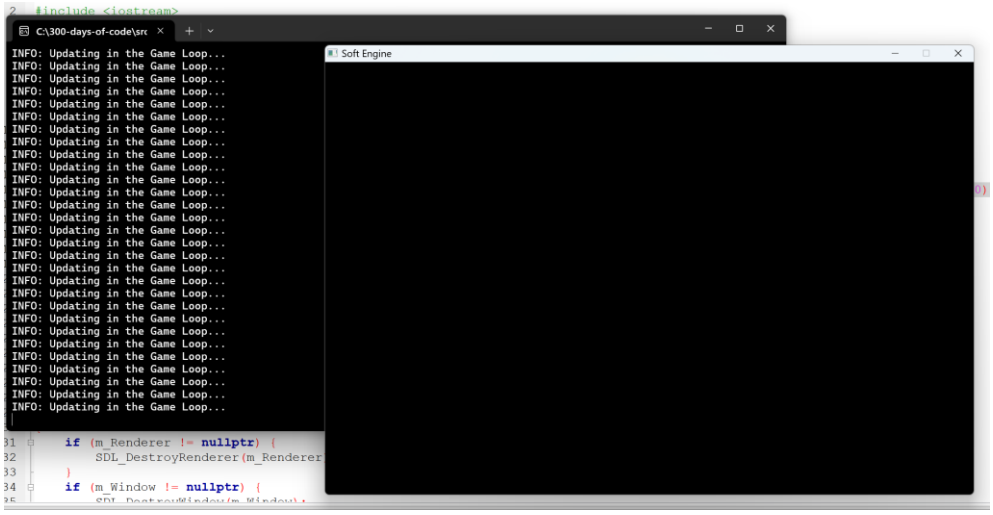


Figure 65 - Running the application at this stage of development

Note: The image above is using the original code from the video and not the update of actually clearing the screen to the specified color.

Handling Events

In order to provide user control of the SDL screen we need to be able to handle events like close window, etc. SDL manages an event queue for us. The event queue holds all the events that

have taken place like mouse move, keyboard press, mouse click, etc. Every time we do something on our SDL window an event is placed on a queue.

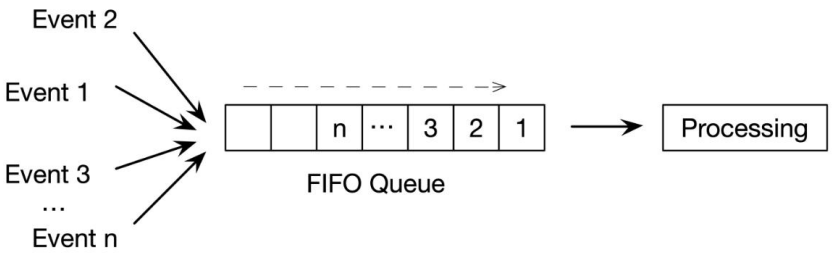


FIGURE 1. Events which occur asynchronously are posted to the FIFO queue and are processed in the order thy occur. The queue effectively decouples the processing from the events allowing for relatively complex applications to be built robustly.

Figure 66 - How an event queue works (from https://circuitcellar.com/resources/event-queues/)

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.

Syntax:

```
typedef union SDL_Event
{
    Uint32 type;                        /**< Event type, shared with all events */
    SDL_CommonEvent common;             /**< Common event data */
    SDL_DisplayEvent display;           /**< Display event data */
    SDL_WindowEvent window;             /**< Window event data */
    SDL_KeyboardEvent key;              /**< Keyboard event data */
    SDL_TextEditingEvent edit;          /**< Text editing event data */
    SDL_TextEditingExtEvent editExt;    /**< Extended text editing event data */
    SDL_TextInputEvent text;            /**< Text input event data */
    SDL_MouseMotionEvent motion;        /**< Mouse motion event data */
    SDL_MouseButtonEvent button;        /**< Mouse button event data */
    SDL_MouseWheelEvent wheel;          /**< Mouse wheel event data */
    SDL_JoyAxisEvent jaxis;             /**< Joystick axis event data */
    SDL_JoyBallEvent jball;             /**< Joystick ball event data */
    SDL_JoyHatEvent jhat;               /**< Joystick hat event data */
    SDL_JoyButtonEvent jbutton;         /**< Joystick button event data */
    SDL_JoyDeviceEvent jdevice;         /**< Joystick device change event data */
    SDL_JoyBatteryEvent jbattery;       /**< Joystick battery event data */
    SDL_ControllerAxisEvent caxis;      /**< Game Controller axis event data */
    SDL_ControllerButtonEvent cbutton;  /**< Game Controller button event data */
    SDL_ControllerDeviceEvent cdevice;  /**< Game Controller device event data */
    SDL_ControllerTouchpadEvent ctouchpad; /**< Game Controller touchpad event data */
    SDL_ControllerSensorEvent csensor;  /**< Game Controller sensor event data */
    SDL_AudioDeviceEvent adevice;       /**< Audio device event data */
    SDL_SensorEvent sensor;             /**< Sensor event data */
    SDL_QuitEvent quit;                 /**< Quit request event data */
    SDL_UserEvent user;                 /**< Custom event data */
    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_CONTROLLERBUTTONUP**: 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();
8.         break;
9.     }
10. }
```

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.

Cleaning up

The video presenter does not discuss one important topic – cleanup the SDL entities you initialize or create. I general principle is close or return everything you open or create. The table below lists the corresponding cleanup function for the ones we used:

SDL function that opens or creates a resource	Corresponding close and destroy function
SDL_Init	SDL_Quit
IMG_Init	IMG_Quit
SDL_CreateWindow	SDL_DestroyWindow
SDL_CreateRenderer	SDL_DestroyRenderer

SDL_Quit()

Note: This refers to the SDL function SDL_Quit() not the Event type of SDL_QUIT.

This function cleans up all initialized subsystems.

Syntax

```
1. void SDL_Quit(void);
```

Remarks

You should call this function even if you have already shutdown each initialized subsystem with SDL_QuitSubSystem(). It is safe to call this function even in the case of errors in initialization.

If you start a subsystem using a call to that subsystem's init function (for example SDL_VideoInit()) instead of SDL_Init() or SDL_InitSubSystem(), then you must use that subsystem's quit function (SDL_VideoQuit()) to shut it down before calling SDL_Quit(). But generally, you should not be using those functions directly anyhow; use SDL_Init() instead.

You can use this function with atexit() to ensure that it is run when your application is shutdown, but it is not wise to do this from a library or other dynamically loaded code.

What is atexit()?

Explanation:

1. **atexit(cleanup);**: Registers the cleanup function to be called automatically when the program exits.
2. **cleanup()**: Ensures that SDL_Quit() is called to clean up SDL subsystems.
3. **Automatic Cleanup**: Even if the program exits early (e.g., due to an error), the cleanup function will still be executed.

This approach is useful for ensuring that resources are properly released, especially in cases where the program might terminate unexpectedly.

Example:

```
#include <SDL.h>
#include <cstdlib> // For atexit()

// Cleanup function to be called on exit
void cleanup() {
    SDL_Quit();
    SDL_Log("SDL cleaned up successfully!");
}

int main(int argc, char* argv[]) {
    // Initialize SDL
    if (SDL_Init(SDL_INIT_VIDEO) != 0) {
        SDL_Log("Unable to initialize SDL: %s", SDL_GetError());
        return 1;
    }

    // Register the cleanup function with atexit
    atexit(cleanup);

    // Create an SDL_Window
    SDL_Window* window = SDL_CreateWindow("atexit Example", SDL_WINDOWPOS_CENTERED,
SDL_WINDOWPOS_CENTERED, 800, 600, SDL_WINDOW_SHOWN);
```

```
22.     if (!window) {
23.         SDL_Log("Could not create window: %s", SDL_GetError());
24.         return 1; // Cleanup will be handled by atexit
25.     }
26.
27.     // Your program logic here...
28.
29.     // Destroy the window before exiting
30.     SDL_DestroyWindow(window);
31.
32.     return 0; // Cleanup function will be called automatically
33. }
34.
```

As you know there can be more than one reason that an application closes. So using `atexit()` ensure that cleanup will take place.

IMG_Quit()

This function deinitialize SDL_image.

Syntax

```
1. void IMG_Quit(void);
```

Remarks

This should be the last function you call in `SDL_image`, after freeing all other resources. This will unload any shared libraries it is using for various codecs.

After this call, a call to IMG_Init(0) will return 0 (no codecs loaded).

You can safely call IMG_Init() to reload various codec support after this call.

Unlike other SDL satellite libraries, calls to IMG_Init do not stack; a single call to IMG_Quit() will deinitialize everything and does not have to be paired with a matching IMG_Init call. For that reason, it's considered best practices to have a single IMG_Init and IMG_Quit call in your program. While this isn't required, be aware of the risks of deviating from that behavior.

SDL_DestroyWindow()

This function is used to destroy or return all resources from using `SDL_CreateWindow()`.

Syntax

```
1. void SDL_DestroyWindow(SDL_Window * window);
```

Function Parameters

SDL_Window *	**window**	the window to destroy.

Remarks

If window is NULL, this function will return immediately after setting the SDL error message to "Invalid window". See SDL_GetError().

What is the difference between NULL and nullptr?

In a C++ program, **NULL** and **nullptr** are both used to represent null pointer values, but they have significant differences:

NULL

1. **Definition**: NULL is typically defined as 0 in C and C++ via the standard library (#define NULL 0).
2. **Type**: It is essentially an integer value (0) that is implicitly converted to a pointer when needed.
3. **Issues**:
    - It can lead to ambiguity in overload resolution, especially in C++ programs where functions may be overloaded based on argument typ

`nullptr` (introduced in C++11)

1. **Definition**: `nullptr` is a keyword that represents a literal of type `std::nullptr_t`.
2. **Type**: It has a distinct type, which ensures type safety. Unlike NULL, it isn't an integer and is specifically recognized as a null pointer.
3. **Advantages**:
    - Eliminates ambiguity in function overloads.
    - Provides clearer intent when used in code, making it obvious that it represents a null pointer.

Example

Here's how NULL vs nullptr might affect function resolution

```
1. #include <iostream>
2.
```

```
 3. void foo(int i) {
 4.     std::cout << "Integer function called!" << std::endl;
 5. }
 6.
 7. void foo(char* p) {
 8.     std::cout << "Pointer function called!" << std::endl;
 9. }
10.
11. int main() {
12.     foo(NULL);    // Ambiguous: Might call the integer version!
13.     foo(nullptr); // Clear: Calls the pointer version!
14.     return 0;
15. }
```

Key Takeaway

Always use `nullptr` in modern C++ programs, as it is type-safe, unambiguous, and more readable. NULL is considered outdated in C++ and is better suited for C programs. Let me know if you'd like more examples!

SDL_DestroyRenderer()

This function is used to destroy the rendering context for a window and free associated textures.

Syntax

```
1. void SDL_DestroyRenderer(SDL_Renderer * renderer);
```

Function Parameters

SDL_Renderer *	**renderer**	the rendering context.

Remarks

If renderer is NULL, this function will return immediately after setting the SDL error message to "Invalid renderer". See SDL_GetError().

Added code to Engine::Clean()

The code that I added cleans up as follows:

```
1. bool Engine::Clean()
```

```
 2. {
 3.     if (m_Renderer != nullptr) {
 4.         SDL_DestroyRenderer(m_Renderer);
 5.     }
 6.     if (m_Window != nullptr) {
 7.         SDL_DestroyWindow(m_Window);
 8.     }
 9.     IMG_Quit();
10.     SDL_Quit();
11.     return true;
12. }
```

4. Draw Texture in SDL

In this video we added a new class named `TextureManager`. The class implements everything we need to draw things on the screen. We also cover the clean up/destroy process that you should perform for every SDL resource you open or create.

While SDL2 itself does not have built-in image handling functions beyond basic surface and texture manipulation, it integrates seamlessly with an extension library called **SDL_image** for more advanced image handling capabilities.

With **SDL_image**, you can load various image formats (like PNG, JPEG, BMP, etc.) into SDL surfaces or textures, which can then be rendered in your application. Here's a high-level summary of how it works:

1. **Initialization**: `SDL_image` needs to be initialized alongside SDL. You can specify which image formats to support during this step.
2. **Loading Images**: It allows you to load images into an `SDL_Surface` or `SDL_Texture` using functions like `IMG_Load()` or `IMG_LoadTexture()`. These functions abstract the complexity of decoding image formats.
3. **Rendering**: Once loaded, the image can be converted into a texture and rendered using SDL's rendering API, offering efficient performance.
4. **Freeing Resources**: After you're done using an image, you'll need to free the resources using SDL functions like `SDL_DestroyTexture()` or `SDL_FreeSurface()`.

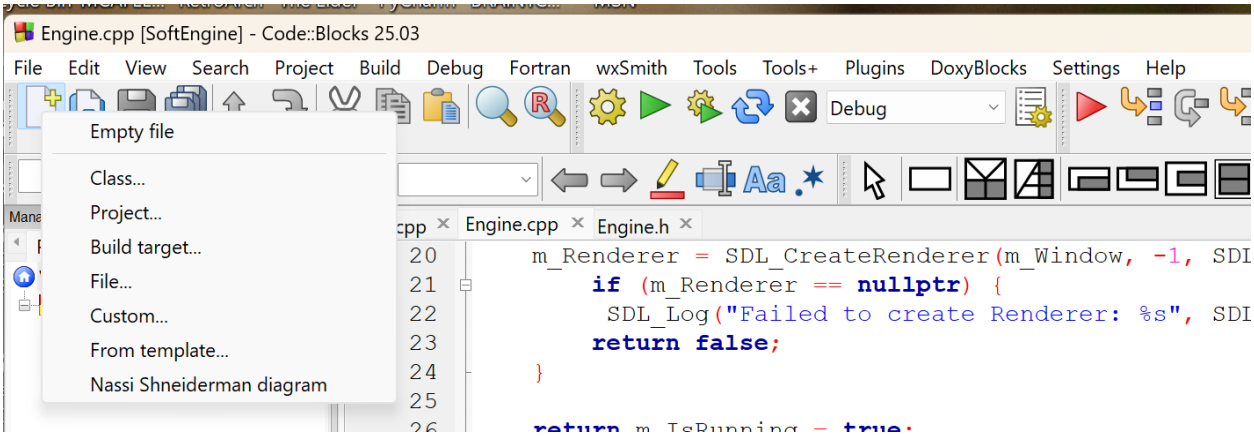
Download the Textures

The website to obtain the texture we are using for this video are located: https://www.patreon.com/posts/33671850 or you can just access the project as I have it on

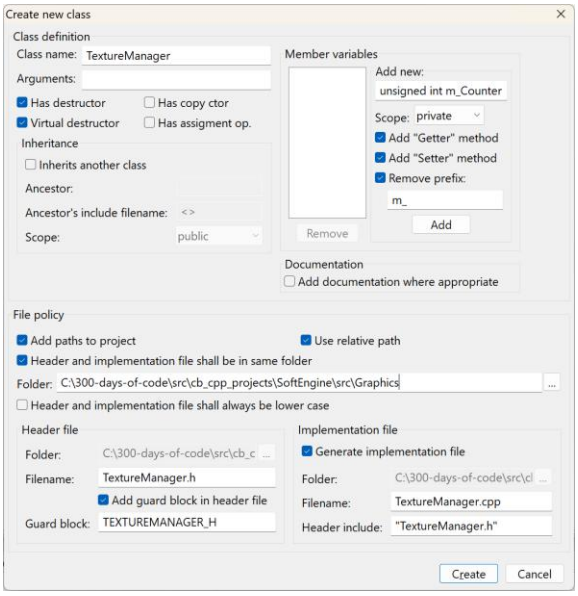
GitHub, located at: https://github.com/nyguerrillagirl/300-days-of-code/tree/main/src/cb_cpp_projects/SoftEngine

Note: It appears that only few of the files and hence assets are available for free. Users have the option of paying at Patreon to obtain code and assets for the set of videos.

- Create the new class `TextureManager` in a folder `src/Graphics` by first clicking on the icon shown, and selecting Class…



- Populate the “Create new class” dialog that appears as shown:



- Click on “Create” button and add to the project as before.

Like our Engine class we will construct the TextureManager as a singleton class.

- Edit the TextureManager.h file as shown:

```
 1. #ifndef TEXTUREMANAGER_H
 2. #define TEXTUREMANAGER_H
 3.
 4. #include <string>
 5. #include <map>
 6. #include "SDL.h"
 7.
 8. class TextureManager
 9. {
10.     public:
11.         static TextureManager* GetInstance() { return s_Instance = (s_Instance != nullptr) ?
s_Instance : new TextureManager();}
12.
13.         bool Load(std::string id, std::string filename );
14.         void Drop(std::string id);
15.         void Clean();
16.
17.         void Draw(std::string id, int x, int y, int width, int height, SDL_RendererFlip
flip=SDL_FLIP_NONE);
18.
19.     protected:
20.
21.     private:
22.          TextureManager() {};
23.          std::map<std::string, SDL_Texture*> m_TextureMap;
24.          static TextureManager* s_Instance;
25. };
26.
27. #endif // TEXTUREMANAGER_H
```

<string>

Programmers include the header file to access the functionality of the `std::string` class in the Standard Template Library (STL). The `std::string` class provides a convenient way to work with strings in C++, offering a variety of features that simplify string manipulation compared to traditional C-style strings (char[]).

Here are some key reasons why the file is commonly used:

1. **Dynamic Size Management**: Unlike C-style strings, `std::string` automatically handles memory allocation and resizing, making it more flexible and less error-prone.
2. **Rich String Manipulation Functions**: It provides methods for concatenation, searching, substring extraction, comparison, and more, all built-in.
3. **Ease of Use**: With `std::string`, you don't have to worry about managing null terminators ('\0') or buffer sizes manually.
4. **Integration with Other STL Components**: `std::string` works seamlessly with other STL classes like `std::vector` and `std::map`, making it essential for modern C++ programming.

Including ensures access to these features, making string handling in C++ far more robust and efficient.

Note to the reader: While I am writing these notes I am acquainting myself with the new features added to the standard library (yes, I am that old). The <string> header and class was added as part of the C++ Standard Library with the first official C++ standard C++98 which was finalized in 1998. I am accustomed to using C-style strings and in fact you will find that libraries like SDL2 expect C-style strings are arguments so we will often convert a string to a C-style string using the function `c_str()` of the string object.

<map>

The `std::map` provides an efficient way to store and manage key-value pairs, where keys are unique and values can be accessed using the keys. Here are some reasons why is commonly used:

1. **Associative Container**: `std::map` allows for fast retrieval of values based on their keys, making it ideal for scenarios like lookup tables or dictionaries.
2. **Automatic Sorting**: The keys in `std::map` are always sorted in ascending order by default, based on their comparison operator. This makes it useful for ordered data.
3. **Flexible Key Types**: You can use any type that supports comparison operators as keys, providing great versatility.
4. **Efficient Performance**: Internally, `std::map` uses a balanced binary search tree (like a red-black tree) for storing data, ensuring efficient time complexity for operations like insertion, deletion, and lookup.
5. **Built-in Functions**: It provides numerous member functions for manipulating and accessing data, like `insert()`, `find()`, `erase()`, and iteration support with iterators.

Without including , the compiler wouldn't recognize std::map, leading to errors.

Creating the GetInstance()

We have a static GetInstance() method similar to what we created in our Engine class:

```
1. static TextureManager* GetInstance() { return s_Instance = (s_Instance != nullptr) ? s_Instance : new TextureManager();}
```

Adding the key class methods for TextureManager

We then define our key public methods:

- **Load()** – Load a new texture or image
- **Drop()** – drop a texture or image (this usually occurs when we move from one level to the next in our game)
- **Clean()** – return all resources
- **Draw()** – Draw to the screen a particular texture or image

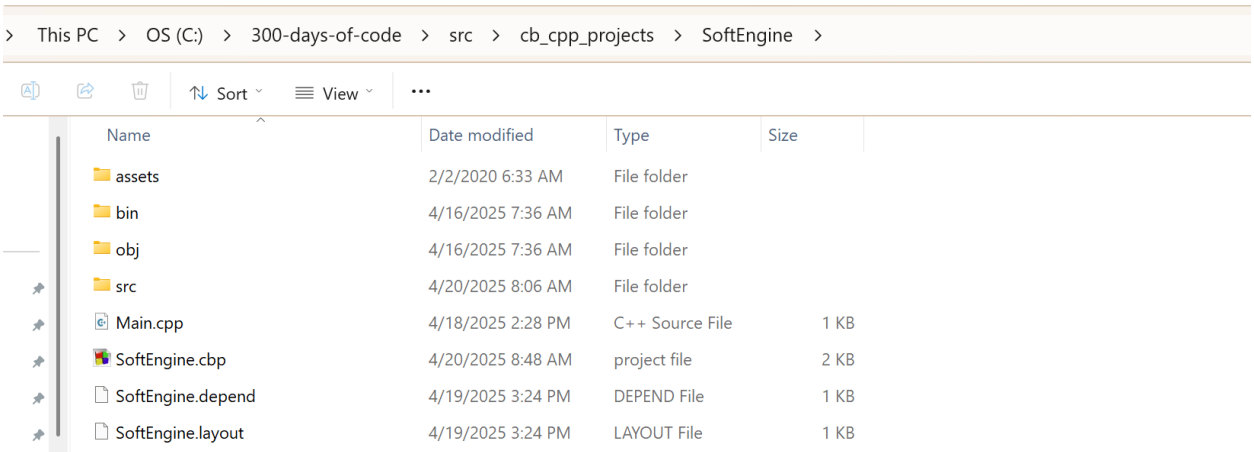
The key data structure used by this class is a map defined as:

```
1. std::map<std::string, SDL_Texture*> m_TextureMap;
```

The program will create many textures to display (the monsters, the background trees, sky, etc.) When you `Load()` a texture, you will provide a unique id to identify the texture/image and the `SDL_Texture` that stores the texture information will be saved to the map.

Edit TextureManager.cpp

- Delete the default provided methods
- First add template code for all unimplemented methods
    - Right-click on file
    - Select Insert/Refactor
    - Select “All class methods without implementation…”
- Create a folder for you project – called assets



- Copy or move the image tree.png into the assets folder

Edit TextureManager.cpp

Reference Engine.h

- Add header for `Engine.h`

```
1. #include "Engine.h"
```

Why do we need `Engine.h`? We added the ability to draw on the window in Engine via the `m_Renderer`. Any piece of code that needs to draw on the window needs to use:

```
1. Engine::GetInstance()->GetRenderer()
```

Since one of the key functions in `TextureManager` is `Draw()`, this is where we will need to access and utilize the `SDL_Renderer`. The philosophy being used is that we will only utilize one `SDL_Renderer` in the program rather than trying to manage one in every class that needs to draw.

Initialize the TextureManager::s_Instance

- Initialize the TextureManager::s_Instance

```
1. // Initialize our singleton to nullptr
2. TextureManager* TextureManager::s_Instance = nullptr;
```

Implement the Load() class method

- Implement the Load() class method:

```
 1. bool TextureManager::Load(std::string id, std::string filename)
 2. {
 3.     SDL_Surface* surface = IMG_Load(filename.c_str());
 4.     if (surface == nullptr) {
 5.         SDL_Log("Failed to load texture: %s, %s", filename.c_str(), SDL_GetError());
 6.         return false;
 7.     }
 8.
 9.     SDL_Texture* texture = SDL_CreateTextureFromSurface(Engine::GetInstance()->GetRenderer(),
surface);
10.     if (texture == nullptr) {
11.         SDL_Log("Failed to create texture from surface: %s",  SDL_GetError());
12.         return false;
13.     }
14.
15.     m_TextureMap[id] = texture;
16.
```

```
17.     return true;
18. }
```

In order to create an SDL_Texture we will need to first create an SDL_Surface:

The relationship between SDL_Surface and SDL_Texture in SDL2 involves how image data is managed and displayed on the screen:

1. **Definition**:
    - SDL_Surface: Represents a block of pixel data stored in system memory. It's mainly used for software rendering or manipulation of images and does not interact directly with the GPU.
    - SDL_Texture: Represents a block of pixel data stored in GPU memory. It's optimized for rendering and is used to interact directly with the graphics hardware.
2. **Conversion**:
    - If you load an image into an SDL_Surface, you can then convert it into an SDL_Texture to render it efficiently using GPU acceleration. This is achieved using the SDL_CreateTextureFromSurface() function.
3. **Usage**:
    - SDL_Surface is suitable for operations like editing pixel data or performing software-based transformations. However, it's not ideal for rendering because it relies on the CPU.
    - SDL_Texture is optimized for rendering on the screen and is used with SDL's rendering API (e.g., SDL_RenderCopy()).
4. **Performance**:
    - While SDL_Surface is easier to manipulate, SDL_Texture offers significantly better performance for rendering since it uses hardware acceleration.

So, you might load your image into an SDL_Surface for initial handling, then convert it to an SDL_Texture for efficient display.

A good illustration[3] is the following:

[3] Inspired by image at: https://www.freepascal-meets-sdl.net/surfaces-and-textures/

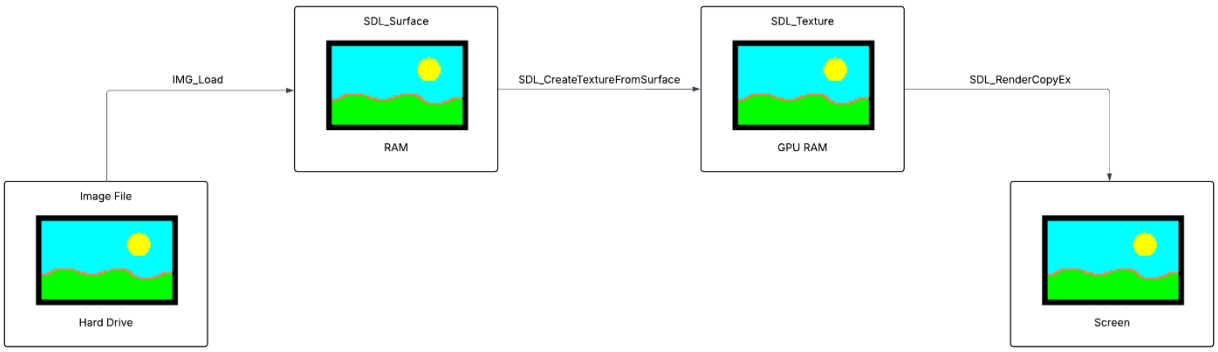


Figure 67 - How we draw an image to the screen

After we successfully create the `SDL_Texture` we place it into the data structure map - `m_TextureMap`.

SDL_Surface

`SDL_Surface` is a structure in SDL2 that represents an area of pixel data stored in system memory.

Syntax

```
typedef struct SDL_Surface
{
    Uint32 flags;               /**< Read-only */
    SDL_PixelFormat *format;    /**< Read-only */
    int w, h;                   /**< Read-only */
    int pitch;                  /**< Read-only */
    void *pixels;               /**< Read-write */

    /** Application data associated with the surface */
    void *userdata;             /**< Read-write */

    /** information needed for surfaces requiring locks */
    int locked;                 /**< Read-only */

    /** list of BlitMap that hold a reference to this surface */
    void *list_blitmap;         /**< Private */

    /** clipping information */
    SDL_Rect clip_rect;         /**< Read-only */

    /** info for fast blit mapping to other surfaces */
    SDL_BlitMap *map;           /**< Private */

    /** Reference count -- used when freeing surface */
    int refcount;               /**< Read-mostly */
} SDL_Surface;
```

Remarks

This structure should be treated as read-only, except for pixels, which, if not NULL, contains the raw pixel data for the surface.

IMG_Load()

Load an image from a filesystem path into a software surface.

Syntax

```
1. SDL_Surface * IMG_Load(const char *file);
```

Function Parameters

const char *	**file**	a path on the filesystem to load an image from.

Return Value

(SDL_Surface *) Returns a new SDL surface, or NULL on error.

Remarks

An SDL_Surface is a buffer of pixels in memory accessible by the CPU. Use this if you plan to hand the data to something else or manipulate it further in code.

There are no guarantees about what format the new SDL_Surface data will be; in many cases, SDL_image will attempt to supply a surface that exactly matches the provided image, but in others it might have to convert (either because the image is in a format that SDL doesn't directly support or because it's compressed data that could reasonably uncompress to various formats and SDL_image had to pick one). You can inspect an SDL_Surface for its specifics, and use SDL_ConvertSurface to then migrate to any supported format.

If the image format supports a transparent pixel, SDL will set the colorkey for the surface. You can enable RLE acceleration on the surface afterwards by calling: SDL_SetColorKey(image, SDL_RLEACCEL, image->format->colorkey);

There is a separate function to read files from an SDL_RWops, if you need an i/o abstraction to provide data from anywhere instead of a simple filesystem read; that function is IMG_Load_RW().

If you are using SDL's 2D rendering API, there is an equivalent call to load images directly into an SDL_Texture for use by the GPU without using a software surface: call IMG_LoadTexture() instead.

When done with the returned surface, the app should dispose of it with a call to SDL_FreeSurface().

SDL_Texture

An efficient driver-specific representation of pixel data

Syntax

```
struct SDL_Texture;
```

SDL_Texture is a structure in SDL2 that represents a block of pixel data stored in GPU memory. It is specifically designed for efficient rendering using hardware acceleration, making it the preferred choice for displaying images, sprites, and other graphical elements in modern applications or games

Key Features of SDL_Texture:

- **GPU-Based**: Unlike SDL_Surface, which works in system memory, SDL_Texture utilizes the graphics hardware (GPU) for fast rendering.
- **Rendering Optimization**: It allows for operations like scaling, rotation, and blending to be performed efficiently during rendering.
- **Immutable Pixel Data**: Once created, you can't directly modify the pixel data in an SDL_Texture. Instead, any changes need to be made at the SDL_Surface level before converting it into a texture.

Common Workflow:

1. **Create a Texture**: You typically create an SDL_Texture using SDL_CreateTexture() or SDL_CreateTextureFromSurface().
2. **Render the Texture**: Use SDL's rendering functions (e.g., SDL_RenderCopy()) to draw the texture on the screen.
3. **Destroy Texture**: Once you're done with the texture, free up its resources using SDL_DestroyTexture().

SDL_CreateTextureFromSurface

Create a texture from an existing surface.

Syntax

```
1. SDL_Texture * SDL_CreateTextureFromSurface(SDL_Renderer * renderer, SDL_Surface * surface);
```

Function Parameters

SDL_Renderer *	**renderer**	the rendering context.
SDL_Surface *	**surface**	the SDL_Surface structure containing pixel data used to fill the texture.

Return Value

(SDL_Texture *) Returns the created texture or NULL on failure; call SDL_GetError() for more information.

Remarks

The surface is not modified or freed by this function.

The SDL_TextureAccess hint for the created texture is SDL_TEXTUREACCESS_STATIC.

The pixel format of the created texture may be different from the pixel format of the surface. Use SDL_QueryTexture() to query the pixel format of the texture.

Implement Draw class method

This method is used to draw the texture to the screen

```
 1. void TextureManager::Draw(std::string id, int x, int y, int width, int height, SDL_RendererFlip
flip)
 2. {
 3.     // Defines the part of the image we plan on drawing
 4.     SDL_Rect srcRect = {0, 0, width, height};
 5.
 6.     // Defines where on the screen we will draw our image
 7.     SDL_Rect dstRect = {x, y, width, height};
 8.
 9.     // Draw our texture to the screen
10.     SDL_RenderCopyEx(Engine::GetInstance()->GetRenderer(), m_TextureMap[id], &srcRect, &dstRect,
0, nullptr, flip);
11.
12. }
```

Your game uses this function to draw a texture to the screen using SDL_RenderCopyEx. We could use SDL_RenderCopy but there are more features and capabilities in SDL_RenderCopyEx.

SDL_Rect

A rectangle, with the origin at the upper left (integer). This is an essential data structure since any graphics element you want to draw on the screen has to fit within a rectangular space – yes that goes for game paddle, image of monsters and even balls!

Syntax

```
1. typedef struct SDL_Rect
2. {
3.     int x, y;
4.     int w, h;
5. } SDL_Rect;
6.
```

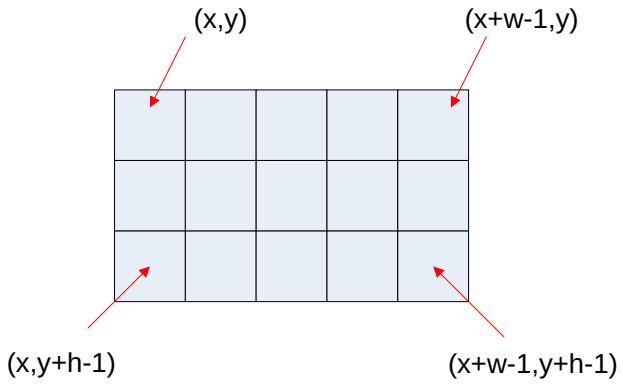


Figure 68 - The pixel coordinates part of the SDL_Rect

SDL_RendererFlip

Flip constants for SDL_RenderCopyEx

Syntax

```
1. typedef enum SDL_RendererFlip
2. {
3.     SDL_FLIP_NONE = 0x00000000,     /**< Do not flip */
4.     SDL_FLIP_HORIZONTAL = 0x00000001,    /**< flip horizontally */
5.     SDL_FLIP_VERTICAL = 0x00000002     /**< flip vertically */
6. } SDL_RendererFlip;
```

What is the difference between SDL_RenderCopy and SDL_RenderCopyEx?

The primary difference between SDL_RenderCopy and SDL_RenderCopyEx lies in the additional rendering options provided by SDL_RenderCopyEx, which are not available in the simpler SDL_RenderCopy function.

1. **SDL_RenderCopy**

This function is used to copy (render) a texture to the rendering target without any transformations. It simply places the texture on the screen.

Key Features:

- Renders the texture at the specified position and size.
- Supports clipping source and destination rectangles.
- No rotation or flipping is available.

Syntax:

```
1. int SDL_RenderCopy(SDL_Renderer* renderer,
2.                    SDL_Texture* texture,
3.                    const SDL_Rect* srcRect,
4.                    const SDL_Rect* dstRect);
```

2. **SDL_RenderCopyEx**

This function is an extended version of SDL_RenderCopy. It includes options for rotating, flipping, and specifying a center point for the rotation.

Key Features:

- Allows rotation by specifying an angle in degrees.
- Supports flipping the texture horizontally, vertically, or both.
- Lets you define the rotation center point.

Syntax:

```
1. int SDL_RenderCopyEx(SDL_Renderer* renderer,
2.                      SDL_Texture* texture,
3.                      const SDL_Rect* srcRect,
4.                      const SDL_Rect* dstRect,
5.                      const double angle,
6.                      const SDL_Point* center,
7.                      const SDL_RendererFlip flip);
```

3. **Key Parameters in SDL_RenderCopyEx**:

- **angle**: The rotation angle (in degrees).
- **center**: The point around which the texture is rotated (can be nullptr to default to the texture's center).
- **flip**: Controls whether the texture is flipped horizontally (SDL_FLIP_HORIZONTAL), vertically (SDL_FLIP_VERTICAL), or not flipped (SDL_FLIP_NONE).

In summary, use SDL_RenderCopy for simple, straightforward rendering, and turn to SDL_RenderCopyEx when you need to apply transformations like rotation or flipping.

Implement Drop()

There may be occasions where you want to remove/drop a texture from the map. One example is when you move from one level to another level where different textures are being used in the new level so you want to remove textures that you will no longer use.

```
1. void TextureManager::Drop(std::string id)
2. {
3.     SDL_DestroyTexture(m_TextureMap[id]);
4.     m_TextureMap.erase(id);
5. }
```

To use the method you will just need to provide the unique id used to create the texture.

SDL_DestroyTexture

The SDL function SDL_DestroyTexture destroys the specified texture.

Syntax

```
1. void SDL_DestroyTexture(SDL_Texture * texture);
```

Function Parameters

SDL_Texture *	**texture**	the texture to destroy.

Remarks

Passing NULL or an otherwise invalid texture will set the SDL error message to "Invalid texture".

Implement Clean()

This method is invoked when the application closes and all the `SDL_Textures` need to be returned.

```
// Removes everything from the map
void TextureManager::Clean()
{
    std::map<std::string, SDL_Texture*>::iterator it;
    for(it = m_TextureMap.begin(); it != m_TextureMap.end(); it++)
        SDL_DestroyTexture(it->second);

    m_TextureMap.clear();

    SDL_Log("texture map cleaned!");

}
```

Update Engine to invoke Clean()

Add TextureManager.h

```
#include "TextureManager.h"
```

Add code to cleanup Textures

We add code to invoke the TextureManager Clean() class method.

```
bool Engine::Clean()
{
    TextureManager::GetInstance()->Clean();

    if (m_Renderer != nullptr) {
        SDL_DestroyRenderer(m_Renderer);
    }
    if (m_Window != nullptr) {
        SDL_DestroyWindow(m_Window);
    }
    IMG_Quit();
    SDL_Quit();
    return true;
}
```

The results

The compilation and execution of the application for this video produces the following:

Figure 69 - Displaying a Texture to the Screen

5. Transform and Vector SDL Game

The start of this video gave us a peek into how the final product will look like. We will have a character that will have an interesting background with our game character being able to move, swing his sword and jump around our game.

Figure 70 - A peek into our game

Overview

In this video we add two header files (only the header files no corresponding cpp classes) that we will use to move objects around.

Coordinate System

The first thing we must establish is a coordinate system. What is a coordinate system?[4]

- A set of points used to identify the position of an object in space
- 2D coordinate system has an (x,y) location
- 3D coordinate system has (x,y,z)

Since we are building a 2D game we will only concern ourselves with the 2D coordinate system.

Visualizing the Coordinate System

We recall learning that the coordinate system looks like:

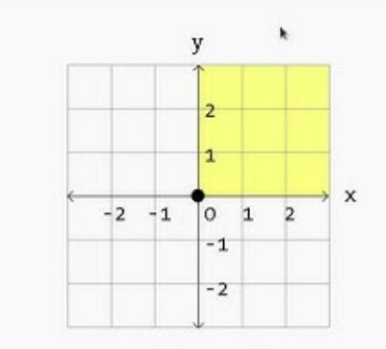


Figure 71 - Coordinate System in our math class

The X-axis is on the horizonal and the Y-axis on the vertical.

We specify the location using (x,y) values as shown when the point (3,5) is shown.

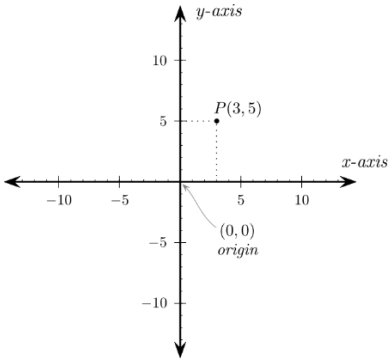


Figure 72 - Locating a point (from: http://programarcadegames.com/index.php?chapter=introduction_to_graphics&lang=en#section_5)

Going up the vertical Y-axis was going in the positive direction (0, 1, 2 …) going down the Y-axis was going in the negative direction (0, -1, -2, …)

[4] https://www.youtube.com/watch?v=M0RVM0-TuH4&ab_channel=BayoCode

When programmers gained the ability to draw graphics/images on the computer screen the Y-axis was changed so that going down the screen was going in the positive direction as shown below:

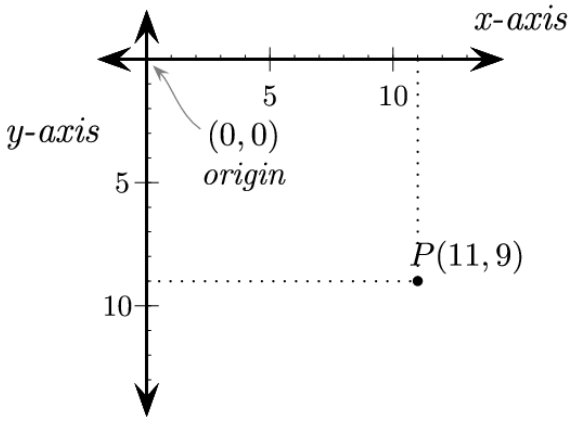


Figure 73 - How to visualize the coordinate system on the screen (from: http://programarcadegames.com/index.php?chapter=introduction_to_graphics&lang=en#section_5)

Physics

In a simple 2D game program, "Physics" refers to the simulation of real-world physical behaviors to make the game feel more natural and interactive. It's essentially a set of rules that define how objects in the game world behave when interacting with each other or the environment. Here's what it typically includes:

1. **Motion and Gravity**: Governs how objects move, fall, or accelerate (like a character jumping or an object dropping).
2. **Collisions**: Ensures that objects don't pass through each other and react realistically when they collide (e.g., bouncing off a wall).
3. **Forces**: Simulates forces like pushing, pulling, or friction that impact the object's movement.
4. **Constraints**: Defines limits on how objects can move, such as keeping a character within the screen boundaries.
5. **Elasticity**: Determines how objects respond after impact, like whether they bounce back or stop.

For example, in a game like "Pong," physics ensures the ball bounces off paddles and walls at the correct angles and speeds.

If you're creating a 2D game, most game engines (like Unity or Godot) have built-in physics engines to handle these elements for you, saving time on complex calculations.

We will implement the code to define the position of an object in our game – `Vector2D.h` and then how to move the object on the screen – `Transform.h`.

Vector2D.h

What is a vector?

A vector is an entity that has a direction and magnitude. What we are defining in this video is actually a `Point` not a vector.

In game programming, a ***vector*** is a mathematical concept used to represent quantities that have both ***magnitude*** and ***direction***. Vectors are fundamental in game development for tasks like positioning, movement, rotation, and scaling objects within a game world. They can be two-dimensional (for 2D games) or three-dimensional (for 3D games), depending on the game's requirements.

For example:

- **Positioning**: A vector can indicate the location of an object in the game world, such as (x, y) for 2D or (x, y, z) for 3D.
- **Movement**: Vectors can represent the direction and speed of an object's movement.
- **Rotation**: They help define the orientation of objects.
- **Scaling**: Vectors can be used to adjust the size of objects proportionally.

Vectors are also used in physics calculations, such as determining collision responses or simulating forces like gravity. Game engines like Unity and Unreal Engine have built-in support for vector operations, making it easier for developers to implement complex mechanics.

Note: We will be changing this header in some future video.

- Create only a header file named Vector2D in a new folder named Physics
    - We Create it in a new folder – Physics
    - We will not be generating the implementation file
    - Accept the prompts after the “Create” button is clicked to add the file to the project

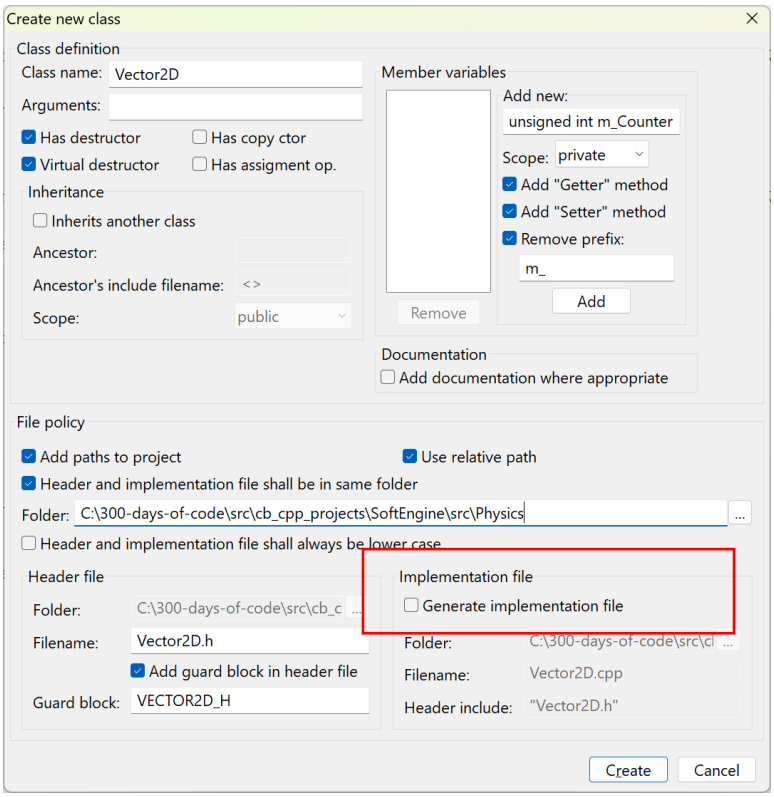


Figure 74 - Creating our new Vector2D.h file (only)

We will see a new folder added to the project:

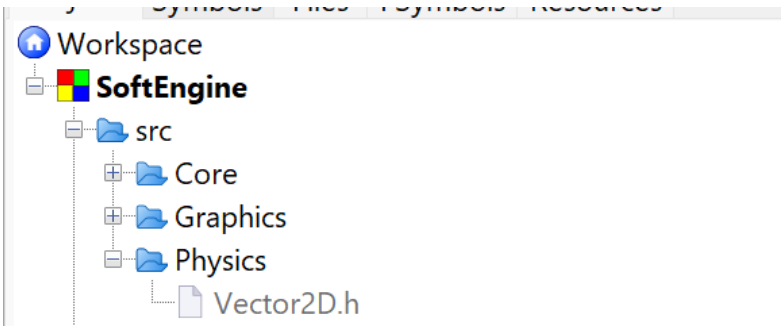


Figure 75 - New Vector2D.h file in Physics folder

```
 1. #ifndef VECTOR2D_H
 2. #define VECTOR2D_H
 3.
 4. #include <iostream>
 5.
 6. class Vector2D
 7. {
 8.     public:
 9.         float X, Y;
10.
11.     public:
```

```
12.         Vector2D(float x=0, float y=0): X(x), Y(y) {}
13.
14.     public:
15.         inline Vector2D operator+(const Vector2D& v2) const {
16.             return Vector2D(X + v2.X, Y + v2.Y);
17.         }
18.          inline Vector2D operator-(const Vector2D& v2) const {
19.             return Vector2D(X - v2.X, Y - v2.Y);
20.         }
21.
22.        void Log(std::string msg="") {
23.             std::cout << msg << "(X,Y)= (" << X << " " << Y << ")" << std::endl;
24.        }
25. };
26.
27. #endif // VECTOR2D_H
```

As you can see from the file the Vector2D makes public the position X, Y. Initializing something like:

Vector2D v1;

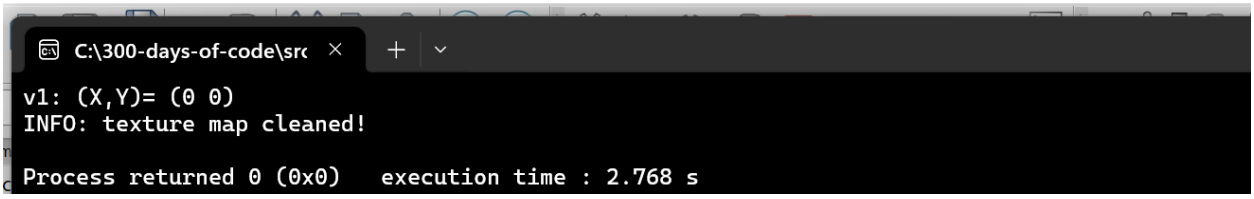
The above will automatically set v1 to X=0 and Y=0. Let's test this out by:

- Open for edit Engine.cpp
- Include Vector2D.h

- Add code in Engine::Init() to create and print out a new vector:

```
1.      TextureManager::GetInstance()->Load("tree", "assets/tree.png");
2.
3.      Vector2D v1;
4.      v1.Log("v1: ");
5.      return m_IsRunning = true;
6.
```

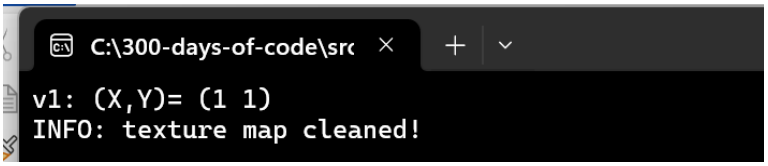
- Run and examine the console log



- Let's change the code to initialize the vector to a specific value:

```
1.      Vector2D v1(1,1);
2.      v1.Log("v1: ");
```

We see the following printed out:



Vector operations

There are several key operations we need to implement:

- Adding vectors and Subtracting vectors
- Multiplying and Dividing vector

Performing the operations above to vectors just means applying the operations to the individual components of a vector:

$$\boxed{v_3 = v_1 + v_2 = (x_1, y_1) + (x_2, y_2) = (x_1 + x_2, y_1 + y_2)}$$

Let $\vec{u} = \langle u_1, u_2 \rangle$ and $\vec{v} = \langle v_1, v_2 \rangle$ be two vectors.

Then, the sum of $\vec{u}$ and $\vec{v}$ is the vector

$$\vec{u} + \vec{v} = \langle u_1 + v_1, u_2 + v_2 \rangle$$

The difference of $\vec{u}$ and $\vec{v}$ is

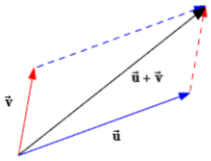
$$\vec{u} - \vec{v} = \vec{u} + \left(-\vec{v}\right)$$
$$= \langle u_1 - v_1, u_2 - v_2 \rangle$$

The sum of two or more vectors is called the resultant. The resultant of two vectors can be found using either the *parallelogram method* or the *triangle method*.

Figure 76 - From https://www.varsitytutors.com/hotmath/hotmath_help/topics/adding-and-subtracting-vectors

Parallelogram Method:

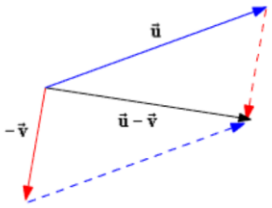
Draw the vectors so that their initial points coincide. Then draw lines to form a complete parallelogram. The diagonal from the initial point to the opposite vertex of the parallelogram is the resultant.



Vector Addition:

1. Place both vectors $\vec{u}$ and $\vec{v}$ at the same initial point.
2. Complete the parallelogram. The resultant vector $\vec{u}+\vec{v}$ is the diagonal of the parallelogram.

Figure 77 - From https://www.varsitytutors.com/hotmath/hotmath_help/topics/adding-and-subtracting-vectors



Vector Subtraction:

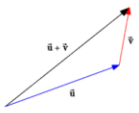
1. Complete the parallelogram.
2. Draw the diagonals of the parallelogram from the initial point.

Alternative method for visualizing how to add and subtract vectors:

Triangle Method:

Draw the vectors one after another, placing the initial point of each successive vector at the terminal point of the previous vector. Then draw the resultant from the initial point of the first vector to the terminal point of the last vector. This method is also called the *head-to-tail method*.

Vector Addition:



Vector Subtraction:

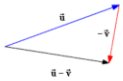


Figure 78 - From https://www.varsitytutors.com/hotmath/hotmath_help/topics/adding-and-subtracting-vectors

Add Code for Vector Addition and Subtraction

```
class Vector2D
{
    public:
        float X, Y;

    public:
        Vector2D(float x=0, float y=0): X(x), Y(y) {}

    public:
        inline Vector2D operator+(const Vector2D& v2) const {
            return Vector2D(X + v2.X, Y + v2.Y);
        }
         inline Vector2D operator-(const Vector2D& v2) const {
            return Vector2D(X - v2.X, Y - v2.Y);
        }

       void Log(std::string msg="") {
            std::cout << msg << "(X,Y)= (" << X << " " << Y << ")" << std::endl;
        }
};
```

- Test the above in Engine.cpp again:

```
    Vector2D v1(1,1), v2(2,2);
    Vector2D v3 = v1 + v2;

    v3.Log("v3: ");
```

We expect to see the sum (3,3) in the console:

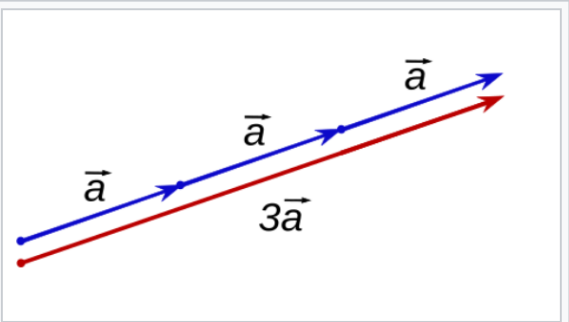
```
v3: (X,Y)= (3 3)
INFO: texture map cleaned!
```

Adding the multiplication operation

The next operation we will add to the Vector2D.h file is the multiplication:

```
        inline Vector2D operator*(const float scaler) const {
            return Vector2D(X * scaler, Y * scaler);
        }
```

What is the visualization on how multiplying a scaler to a vector does to it:



Scalar multiplication of a vector by a factor of 3 stretches the vector out.

The scaler increases or scales up the original vector. You can imagine it can be used to stretch or shrink the vector.

Transform.h

What does it mean to Transform an object?

In game programming, a **Transform** typically refers to an object's position, rotation, and scale in a game world. Vectors play a crucial role in defining and manipulating these properties. Here's how they work together:

1. **Position**: A vector represents the object's location in the game world. For example, in a 3D game, a position vector like (x, y, z) specifies where the object is located relative to the origin.
2. **Rotation**: Vectors are used to define the direction an object is facing. For instance, a forward vector might indicate the direction an object is moving or looking. Rotations can also be represented using quaternions or Euler angles, but vectors are often involved in calculations.
3. **Scale**: A vector can define how much an object is scaled along each axis. For example, a scale vector (1,2) would stretch the object twice as much along the y-axis while x-axis unchanged
4. **Transformations**: Vectors are manipulated to apply transformations like translation (moving an object), rotation, or scaling. For example:
    - **Translation**: Adding a vector to an object's position moves it in the specified direction.

- **Rotation**: Multiplying a vector by a rotation matrix or quaternion rotates the object
- **Scaling**: Multiplying a vector by a scaler value or a scale matrix adjust's the object's size

So Transform is used to change an object's position, rotate it, or scale it.

- Create the Transform.h file
  - Make sure you do not implement to cpp file
  - After pressing "Create" add the file to the project as before

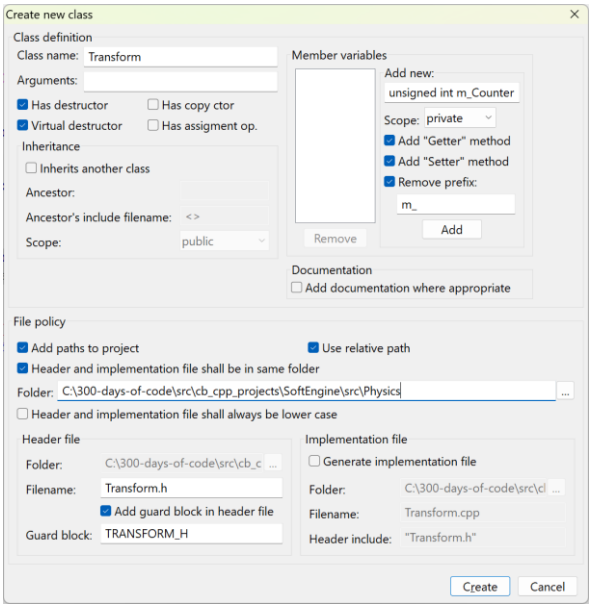


Figure 79 - Creating the Transform.h file

- Populate the Transform.h with the following code:

```
#ifndef TRANSFORM_H
#define TRANSFORM_H

#include "Vector2D.h"

class Transform
{
    public:
        float X, Y;

    public:
        Transform(float x=0, float y=0):X(x), Y(y) {}

    public:
        // Move in the X-axis direction
        inline void TranslateX(float x) { X += x; }

```

```
18.         // Move in the Y-axis direction
19.         inline void TranslateY(float y) { Y += y; }
20.
21.         // Move in the game world using a vector
22.         inline void Translate(Vector2D v) { X += v.X; Y += v.Y; }
23. };
24.
25. #endif // TRANSFORM_H
26.
```

Note: In the video the TranslateX, TranslateY and Translate are all private but this needs to be made public in order to be able to utilize the function.

We define a constructor and three class methods that allow us to move an object in the game world.

- Add a public Log class method to the Transform.h file:

```
1.         void Log(std::string msg="") {
2.             std::cout << msg << "(X,Y)= (" << X << " " << Y << ")" << std::endl;
3.         }
```

We added this method so we can test things out.

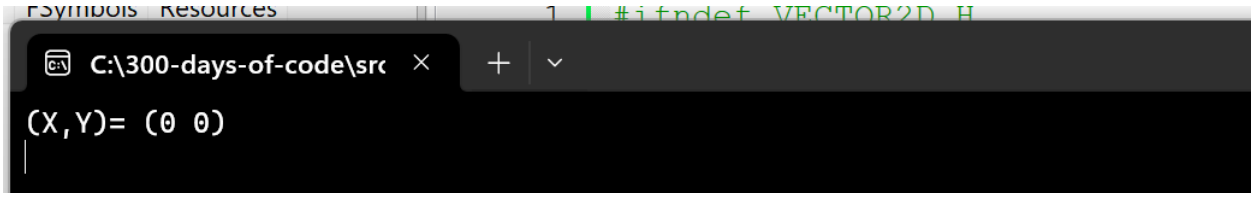
- Edit the file Engine.cpp again
    - Remove the include line for Vector2D.h
    - Add an include line for Transform.h

```
1. #include "Transform.h"
```

    - Remove the Vector2D variable and log and replace with:

```
1.     Transform tf;
2.     tf.Log();
```

    - Test run the program again



Is there a better way to Unit test our code?

The unit testing for the classes we are creating seems to be an old style way to test. I have been a Java developer for the last two decades and was wondering what it would take to do unit testing in a C++ development environment. This is what copilot tells me:

Unit testing in C++ is a crucial step in ensuring that individual components of a program work correctly. Developers typically use unit testing frameworks to automate and streamline the process. Here's how they go about it:

1. **Choose a Testing Framework:** Popular options for C++ include:
    - **Google Test (GTest):** Widely used due to its rich feature set and simplicity.
    - **Catch2:** A lightweight, single-header alternative.
    - **Boost.Test:** Part of the Boost C++ libraries, offering comprehensive testing capabilities.
    - **CppUnit:** Inspired by JUnit for testing C++ code.
2. **Write Test Cases:** Developers create small, isolated test functions to validate specific parts of the code. Each test function checks a particular scenario using assertions like EXPECT_EQ or ASSERT_TRUE in Google Test.
3. **Organize Tests:** Tests are usually grouped into suites based on related functionality, making it easier to run and debug them.
4. **Run Tests Automatically:** The testing framework executes all test cases, reporting failures and errors. Many developers integrate unit tests into **continuous integration (CI)** pipelines to ensure issues are caught early.
5. **Mock Dependencies:** If a function relies on external modules (like databases or network calls), mocking frameworks like **Google Mock** help simulate expected behavior.
6. **Analyze Code Coverage:** Tools like **gcov**, **LCOV**, or **Clang Sanitizers** provide insights into how much of the code is covered by tests, helping identify gaps.
7. **Refactor and Maintain Tests:** As the code evolves, developers regularly update tests to reflect changes, ensuring they remain useful.

I will pursue an new area (outside these notes and the video) to learn how one would unit test a C++ program

6. Game Object SDL Game

Overview

We will need to represent various “video game objects” in our game. An object is a fundamental building block within a game scene. It’s essentially a container that holds components, which determine the object’s visual appearance, behavior, and functionality.

An object can be the player, an enemy, or something we interact with in our game. We will start the video by creating an interface class – `IObject`. This interface defines what all objects (e.g. the sneaky monster out to get the player) must all implement, the functions:

- `Update()` – to update the object’s position in the game world
- `Draw()` – draw the object to the screen
- `Clean()` – draws the object

Within the game loop, we will invoke the functions within each object.

Create the interface IObject

- Create a new class in a new folder for our project named src\Object
    - The name of the class is IObject
    - This is only a header file (turn off implementation generation)
    - Add the file to the current project
    - Accept both Debug and Release targets

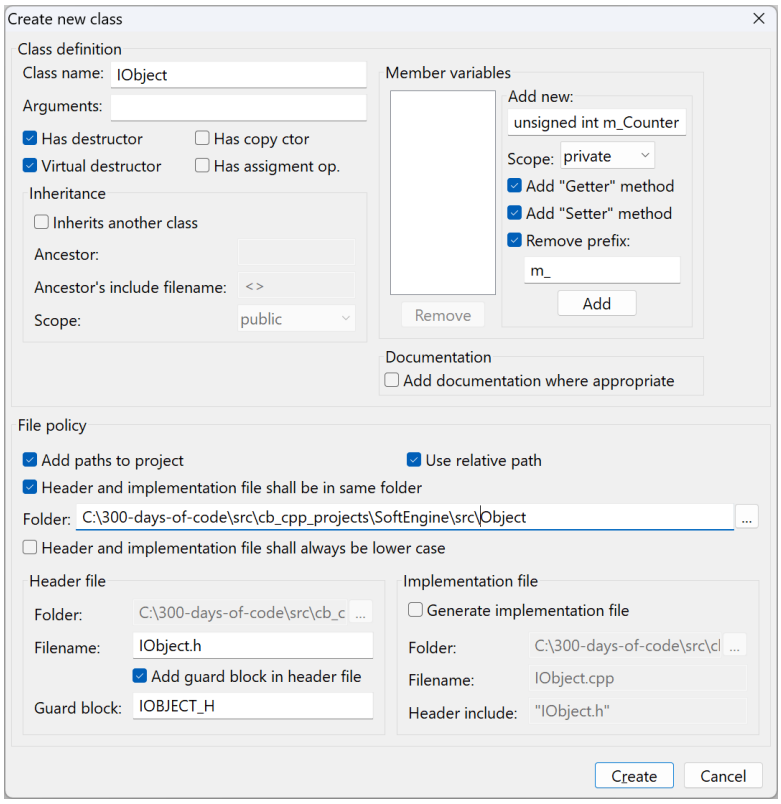


Figure 80 - Creating IObject header file

The contents of the IObject interface is:

```
#ifndef IOBJECT_H
#define IOBJECT_H


class IObject
{

    public:
        virtual void Draw() = 0;
        virtual void Update(float dt) = 0;
        virtual void Clean() = 0;

};

#endif // IOBJECT_H

```

The interface defines the three actions of functions that every object should implement:

- Draw – draw the image that represents the object
- Update – move or transform the position of the object
- Clean – clean up all resources associated with the object

Create GameObject class

This will also be just a header file:

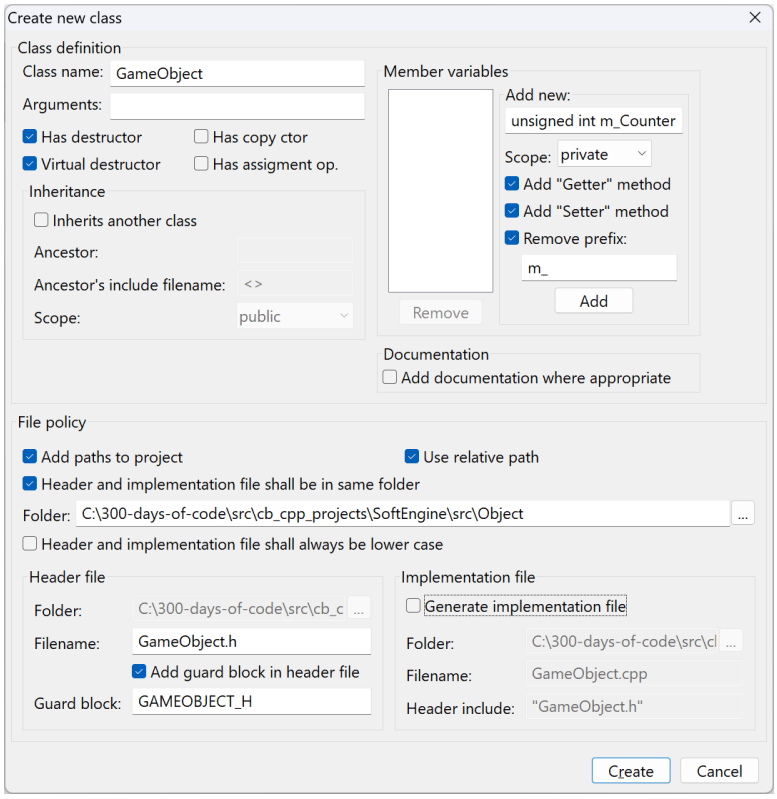


Figure 81 - Creating the GameObject header file

`GameObject` is also only a header file that inherits from `IObject`.

The GameObject tracks several key properties of an object:

- **Transform** – which is the point in the game world our object is in and how it is moving
- **Width and Height** – since every object defines a rectangular space in the game world this holds the width and height of the GameObject
- **textureID** – this holds the texture/image id of the current object (see TextureManager class)
- **flip** – this holds the orientation of the object

Figure 82 - From https://blenderartists.org/t/2d-character-sprites/436586

The initial code we will add to GameObject:

```
 1. #ifndef GAMEOBJECT_H
 2. #define GAMEOBJECT_H
 3.
 4. #include "IObject.h"
 5. #include "Transform.h"
 6.
 7. class GameObject : public IObject
 8. {
 9.     public:
10.         GameObject() {}
11.
12.         virtual void Draw() = 0;
13.         virtual void Update(float dt) = 0;
14.         virtual void Clean() = 0;
15.
16.     protected:
17.        Transform* m_Transform;
18.        int m_Width, m_Height;
19.        std::string m_TextureID;
20.        SDL_RendererFlip m_Flip;
21.
22. };
23.
24. #endif // GAMEOBJECT_H
25.
```

Lines 4-5 are the include for the interface and transform header files, respectively.

Lines 12-14 are placeholders for the functions `Draw`, `Update` and `Clean` defined by the interface `IObject`.

Lines 17-20 define the key properties of all objects.

We need to fill out the `GameObject` constructor but there are many properties that need to be provided by any client code and the list of arguments can potentially grow. A common practice

used is to create a `struct` that captures all the attributes that need to be provided in the constructor.

- In the same `GameObject.h` file add a new struct `Properties`:

```
 1. struct Properties {
 2.
 3.     public:
 4.         Properties(std::string textureID, int x, int y, int width, int height, SDL_RendererFlip
flip = SDL_FLIP_NONE) {
 5.             X = x;
 6.             Y = y;
 7.             Flip = flip;
 8.             Width = width;
 9.             Height = height;
10.             TextureID = textureID;
11.         }
12.
13.         std::string TextureID;
14.         int Width, Height;
15.         float X, Y;
16.         SDL_RendererFlip Flip;
17. };
18.
```

- Update the GameObject constructor to expect a Properties* argument:

```
1.         GameObject(Properties* props) :
2.             m_TextureID(props->TextureID),
3.             m_Width(props->Width),
4.             m_Height(props->Height),
5.             m_Flip(props->Flip) {
6.                 m_Transform = new Transform(props->X, props->Y);
7.             };
8.
```

Test Things out

I added:

`#include "GameObject.h"` to my GameEngine.cpp file and made sure everything compiled with no issues. I remove once I found all typos.

7. Sprite Sheet Animation for SDL Game – Part 1

In this video tutorial we add code to animate a warrior character from a sprite sheet. When all is said and done we should see the following image on the screen:

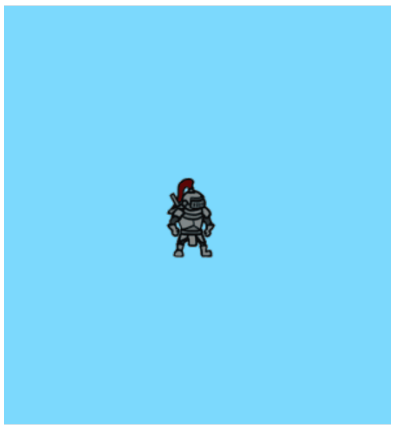


Figure 83 - Our animated Warrior/player

To display the Warrior/player we will need to iterate through several images of the player as he stands in idle mode. This video series starts the discussion on animation of our game characters.

What is a sprite sheet?

We start with what we call a sprite sheet – an image file consisting of many images with the intent that we utilize the images to display characters in our game walking, idling, running, or dying.

A **sprite sheet** is a single image file that contains multiple smaller images, called **sprites**, arranged in a grid or sequence. These sprites are used in **video games, animations, and UI design** to create characters, objects, and effects efficiently.

Why Use Sprite Sheets?

- **Performance Optimization**: Reduces the number of image loads, improving rendering speed.
- **Animation Efficiency**: Stores multiple frames of an animation in one file.
- **Memory Management**: Helps reduce memory usage by keeping assets compact.

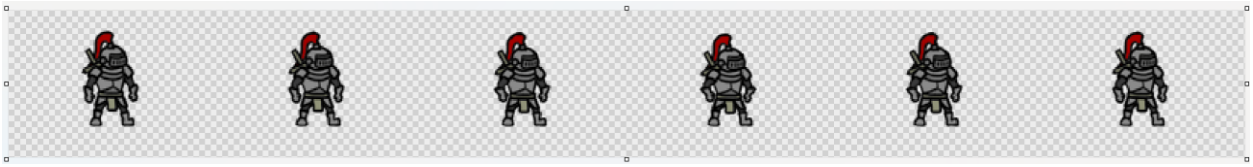
How Sprite Sheets Work

Each sprite in the sheet is mapped using **coordinates**, allowing game engines or software to extract and display specific frames when needed. This technique is commonly used in **2D games** for character movements, explosions, and UI elements.

If you're looking to create your own sprite sheet, you can check out tools like Piskel or Final Parsec's Sprite Sheet Maker.

For this video we use the image file `Idle.png`:

The file contains six images of a warrior as he stands idle. It is hard to tell from the above image that the warrior position changes slightly from frame to frame. It may be easier to see in the image below:



We want the code to input the sheet/file and display each “frame” or portion of the warrior to give us the illusion of the warrior moving up and down as he idles waiting for the next user commands.

How do we ensure the sprite animates in a timely manner?

One of the things we must do is know how to partition the sprite sheet into frames and set how fast to display every frame on the sheet to give the illusion of walking, fighting, or just standing there waiting.

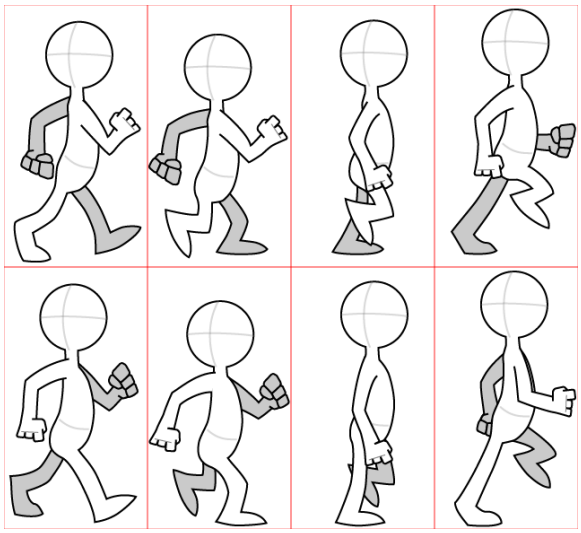


Figure 84 - Working with an animation sheet (from: https://www.cs.cornell.edu/courses/cs5152/2025sp/labs/design1/#task-2-the-sprite-sheet)

We know that as part of the game loop the repeated call to update the entities on the screen, with respect to a sprite it means moving to the next frame on the sheet – but not so fast! We don't want every call to the Update for our warrior to move to the next frame but wait some time between every frame. We will discuss how to do that later.

Updating TextureManager to handle a spritesheet

We add a new method to the `TextureManager` class to handle a sprite sheet. Normally, a sprite sheet may contain several rows of images where each row has one or more frames to display of characters in various modes:

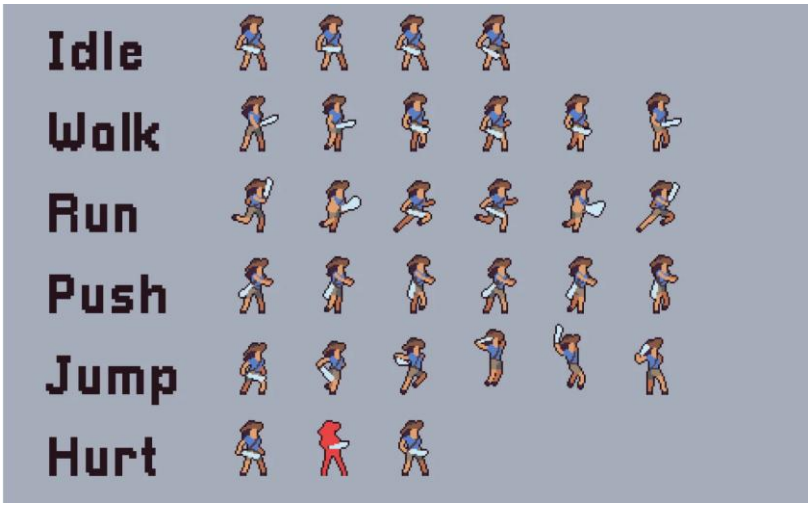


Figure 85 - A sprite sheet (from: https://craftpix.net/freebies/free-3-character-sprite-sheets-pixel-art/)

The sprite sheet for this video is rather simple, having only one row.

- Edit TextureManager.h adding a new DrawFrame method:

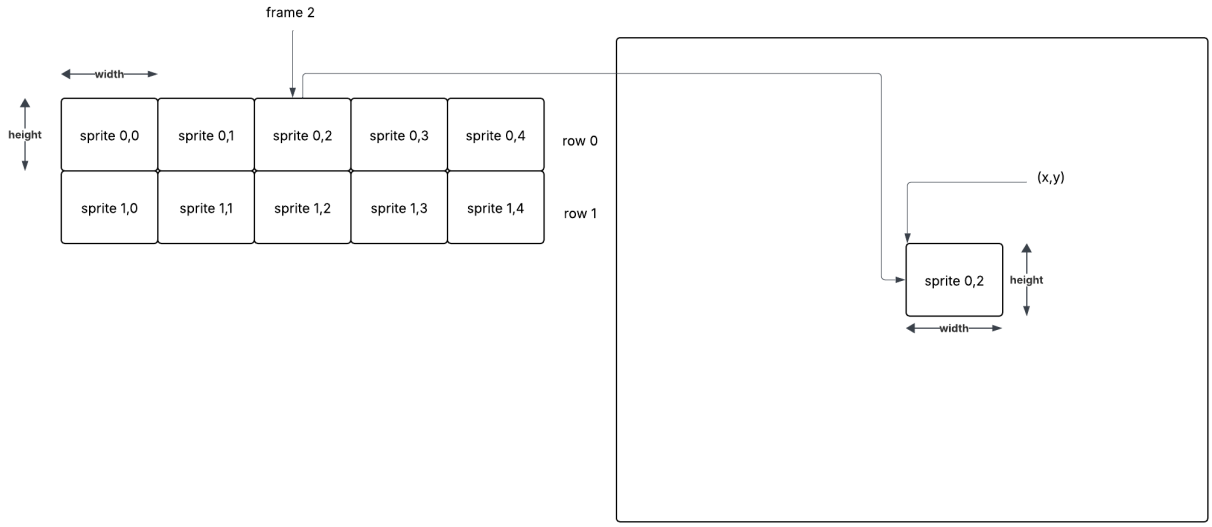
```
1.        void DrawFrame(std::string id, int x, int y, int width, int height, int row, int frame,
SDL_RendererFlip flip);
```

- Edit TextureManager.cpp to add the implementation:

```
1. void TextureManager::DrawFrame(std::string id, int x, int y, int width, int height, int row, int
frame, SDL_RendererFlip flip)
2. {
3.     SDL_Rect srcRect = {width*frame, height*row, width, height};
4.     SDL_Rect dstRect = {x, y, width, height};
5.
6.     SDL_RenderCopyEx(Engine::GetInstance()->GetRenderer(), m_TextureMap[id], &srcRect, &dstRect,
0, nullptr, flip);
7.
8. }
9.
```

The method `DrawFrame` has two more arguments (row and frame) to allow us to calculate which image on the sprite sheet we want to draw to the screen.

The `srcRect` represents the part of the texture/sprite sheet you want to extract and draw to the screen. The dstRect defines the location on the screen you want to draw that frame.



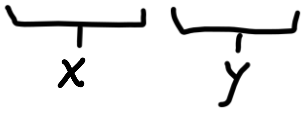
The above represents a sprite sheet with two rows, where each row has 5 frames[5].

If we invoke `DrawFrame` with the following arguments in order to draw sprite 0,2 (row 0, frame 2) to the screen at location (x,y).

- id – the sprite sheet texture
- x,y – the location on the screen to draw the frame we are interested in
- width – the width of the sprite
- height – the height of the sprite
- row – the row in the sprite sheet we want to draw from
- frame – the sprite in the row we want to draw
- flip – orientation of our sprite

We need to define the `srcRect` to be the location on the sprite sheet of the desired frame on the desired row:

```
SDL_Rect srcRect = {width*frame, height*row, width, height};
```



[5] We will maintain using our counts that start at 0 (for row 1).

The first two arguments define the x and y location in the sprite sheet we want to start the extraction of the source frame. When frame=0 and row = 0 we are going to extract sprite 0,0 (do the math). When we want to extract sprite 0,2 we calculate the x as width*2 and y=0 to determine the top left corner.

Create the Character.h file

- Create a new class (with no implementation) called `Character` in a new folder `src/Characters`
    - It will inherit from `GameObject`

```
#ifndef CHARACTER_H
#define CHARACTER_H

#include <GameObject.h>


class Character : public GameObject
{
    public:
        Character() {}
        virtual ~Character() {}

    protected:

    private:
};

#endif // CHARACTER_H
```

- Update the constructor to invoke the `GameObject` constructor:

```
Character(Properties props): GameObject(props) {}
```

- Remove the destructor class method
- We need to define or implement the methods defined in the interface `IObject`

```
#ifndef CHARACTER_H
#define CHARACTER_H

#include <GameObject.h>


class Character : public GameObject
{
    public:
        Character(Properties* props): GameObject(props) {}

        virtual void Draw() = 0;
        virtual void Update(float dt) = 0;
        virtual void Clean() = 0;
```

```
15.
16.     protected:
17.
18.
19. };
20.
21. #endif // CHARACTER_H
```

- Add an include for string:

```
1. #include <string>
```

- Add a protected member for the `m_Name` member variable:

```
1.     protected:
2.         std::string m_Name;
```

Creating the Warrior class

- Create a new class named `Warrior`
  - Save into the `\src\Characters` folder
  - Inherit from the `Character` class
  - Implement the `Warrior.cpp` file

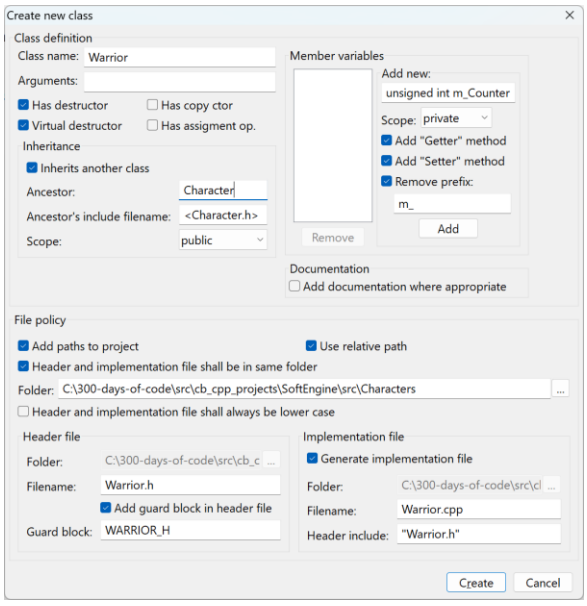


Figure 86 - Creating the Warrior.h and Warrior.cpp

UML Diagram of our current classes

Let's keep in mind the relationship between classes:

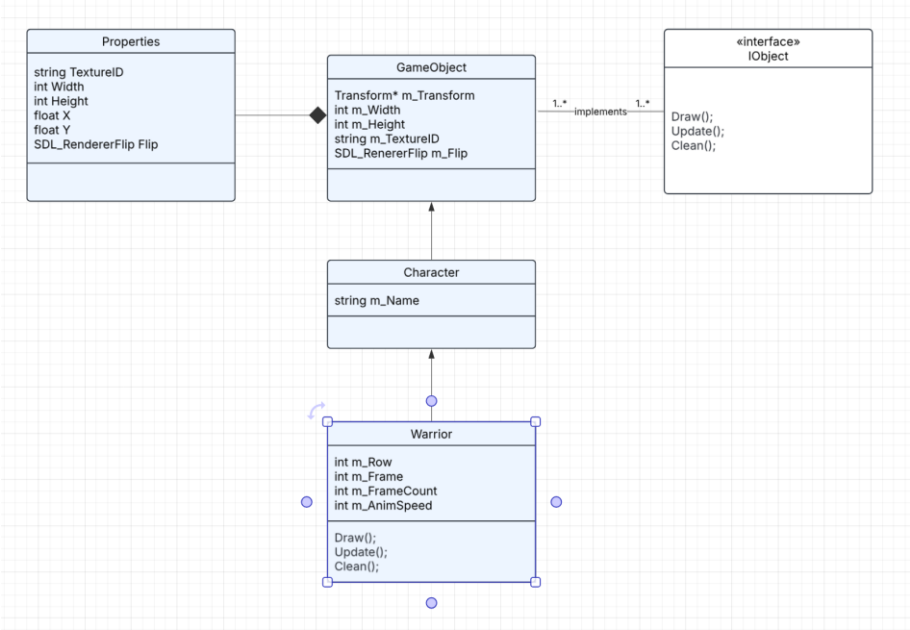


Figure 87 - UML Diagram our classes

When we instantiate a Warrior class we also have an m_Name from the Character class and m_Transform, m_Width, m_Heigh, m_TextureID and m_Flip from the GameObject class.

Add Code for Update

- Remove the destructor in the Warrior.h
- Add the Draw, Update, Clean virtual methods to the Warrior.h:

```
#ifndef WARRIOR_H
#define WARRIOR_H

#include <Character.h>


class Warrior : public Character
{
    public:
        Warrior(Properties* props);

        virtual void Draw() = 0;
        virtual void Update(float dt) = 0;
        virtual void Clean() = 0;

    protected:

    private:
};
```

```
20.
21. #endif // WARRIOR_H
22.
```

- Open the Warrior.cpp file
- Remove the destructor place holder
- Right-click on the file and select Insert/Rector and select All class methods without implementation

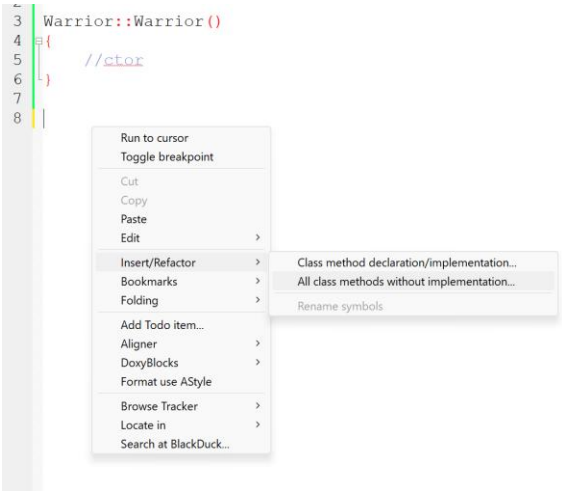


Figure 88 - Using IDE to add class method placeholders

- Edit the Warrior.h to add three private member variable to manage/track the frame animation that is being displayed for the Warrior:

```
1.     private:
2.         int m_Row, m_Frame, m_FrameCount;
```

Another important member variable is the animation speed – how fast should we be going through our frames:

```
1.     private:
2.         int m_Row, m_Frame, m_FrameCount;
3.         int m_AnimSpeed;
```

Add complete implementation classes for Draw, Update and Clean

Warrior::Warrior constructor

We add the following code to initialize our Warrior player:

```
#include "Warrior.h"
#include "SDL.h"
#include "TextureManager.h"


Warrior::Warrior(Properties* props): Character(props)
{
    m_Row = 0;
    m_FrameCount = 6;
    m_AnimSpeed = 80;
}

```

The sprite sheet we are working with only has one row with six frames or images. The animation speed will be examined in the Update().

Warrior::Draw()

This class method utilizes the `DrawFrame` method in the `TextureManager`:

```
void Warrior::Draw()
{
    TextureManager::GetInstance()->DrawFrame(m_TextureID, m_Transform->X, m_Transform->Y,
m_Width, m_Height, m_Row, m_Frame, m_Flip);
}
```

Note, how we access and utilize member variables defined in classes we inherited, we can see the members that are accessible via the UML diagram (see Figure 87).

Warrior::Update()

The `Warrior` class must handle what occurs when its `Update()`, `Draw()` and `Clean()` functions are invoked. In this example, we want the object to display the right frame, and after a bit of time move to the next frame. When it reaches the last frame we want it to wrap around back to frame #0. It is in this function that we determine how long to spend on frame x before we move to frame x+1.

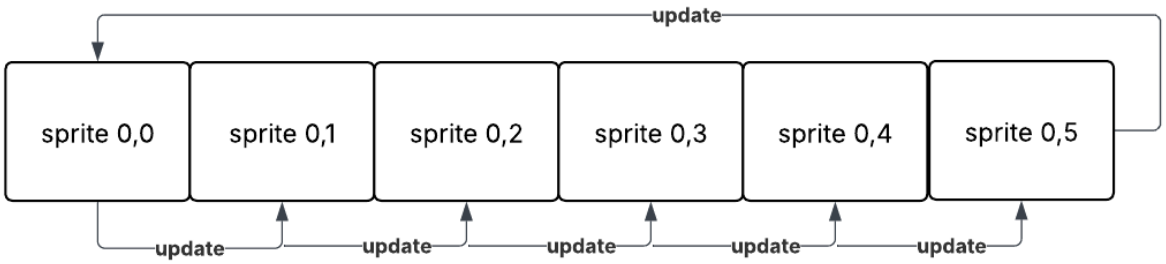


Figure 89 - Updating the frame to display

The figure illustrates the fact that we move from one frame in the sprite sheet to another until we reach the end and go back to the first frame. The ideal operation for cycling is the mod operator (%). Suppose we want to cycle from frames 0..5 and then back to 0. Imagine using the algorithm:

```
1. count = 0
2. repeat
3.        draw frame(count % 6)
4.        ++count;
5. until object_is_removed;
```

We see that because of % operator we will cycle from `0, 1, 2, 3, 4, 5`, back to `0, 1, 2, 3, 4, 5` as the count increases. The mod operator limits the frame number to a number from `0..5`.

The above is easy to understand but what if the count was increasing every millisecond? We don't want to iterate to a new frame every millisecond! It will just appear as one big blur rather than a Warrior idling waiting for a key press.

Suppose the loop was running every millisecond but we only wanted to move up to the next frame say every `80` milliseconds. How can we do that?

When we divide x by y (x/y) we get the quotient of the division. Suppose x = 100 and y = 10.

At first `x / y => 100/ 10 = 10`.

Increase x by 1. (this is our counter)

`x / y  => 101 / 10 = 10`

Increase x by 1 again.

`x / y => 102 / 10 = 10`

This continues to be 10 until x = 110. Then we get

`x / y => 110 / 10 = 11`.

We use this fact to "wait" `m_AminSpeed` before moving to the next frame. We divide the counter by `m_AminSpeed`.

But what is our counter?

What is SDL_GetTicks()?

`SDL_GetTicks()` is a function in the Simple DirectMedia Layer (SDL) library that returns the number of milliseconds that have elapsed since the SDL library was initialized. This function is useful for timing operations, measuring frame rates, and handling animations in games and multimedia applications.

Syntax

```
Uint32 SDL_GetTicks(void);
```

Function Parameters

NA

Returns

(Uint32) Returns an unsigned 32-bit value representing the number of milliseconds since the SDL library initialized.

Remarks

This value wraps if the program runs for more than ~49 days.

This function is not recommended as of SDL 2.0.18; use SDL_GetTicks64() instead, where the value doesn't wrap every ~49 days. There are places in SDL where we provide a 32-bit timestamp that cannot change without breaking binary compatibility, though, so this function isn't officially deprecated.

The SDL_GetTicks will be our “counter”.

Using the AnimSpeed to control the frame rate

The implementation of Update():

```
1. void Warrior::Update(float dt)
2. {
3.     m_Frame = (SDL_GetTicks() / m_AnimSpeed) % m_FrameCount;
4. }
```

The `(SDL_GetTicks() / m_AnimSpeed)`[6] changes to a new quotient value every `m_AnimSpeed` time which in turn maps to a new frame in our sprite sheet.

Warrior::Clean()

This function just calls the TextureManager Clean() function.

```
1. void Warrior::Clean()
2. {
3.     TextureManager::GetInstance()->Clean();
4. }
```

Testing our Warrior class

We will now add code to our Engine.cpp class to test out our Warrior class.

- Open the Engine.cpp class file
- Add an include for the Warrior.h file:

```
1. #include "Warrior.h"
```

- Initialize a Warrior object (the player):

```
1. Engine* Engine::s_Instance = nullptr;
2. Warrior* player = nullptr;
```

- Change code to load the Idle.png rather than tree.png and initialize the player

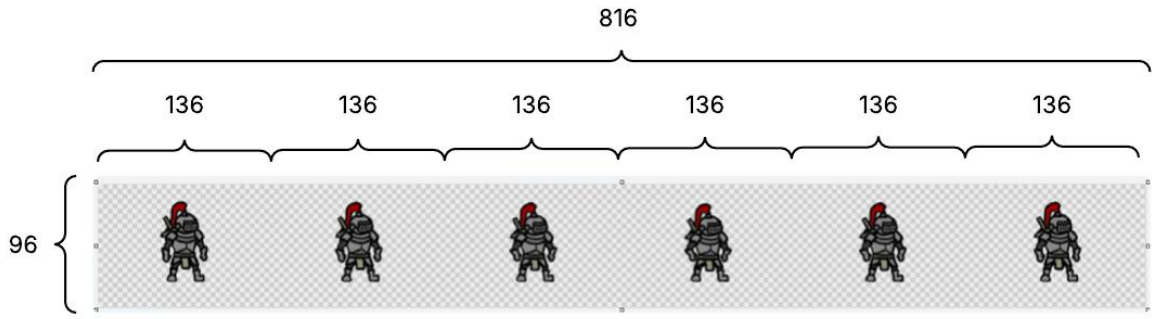
```
1.     TextureManager::GetInstance()->Load("player", "assets/Idle.png");
2.     player = new Warrior(new Properties("player", 100, 200, 136, 96));
```

You may be wondering how to figure out the width and height to provide as the properties values. We need to provide the width and height of the individual sprite.

The entire sprite sheet is 816 x 96 pixels. We know it has 6 sprite images therefore we want to partition the sprite sheet into 816 / 6 or 136 pixels wide and 96 pixels in height.

We can image the following:

[6] The first time, going from frame 0 to frame 1 the timing may be off since we don't know the starting value of the counter - SDL_GetTicks() is the number of milliseconds from the start of initial call to SDL_Init().



Things can get more complicated if the sprite sheet consisted of many rows where the sprites had different widths on every row. You would probably need to use another file to specify the width and height more accurately. In the case of our Warrior things are rather straightforward.

- Add code to the Engine::Update() to invoke the player's Update() function

```
1. void Engine::Update()
2. {
3.     player->Update(0);
4. }
```

- Add code to the Engine::Render to draw our player

```
 1. void Engine::Render()
 2. {
 3.     // Set the draw color
 4.     SDL_SetRenderDrawColor(m_Renderer, 124, 218, 254, 255);
 5.
 6.     // Clear the screen with the draw color
 7.     SDL_RenderClear(m_Renderer);
 8.
 9.     //TextureManager::GetInstance()->Draw("tree", 100, 100, 347, 465);
10.
11.     // Draw our Warrior
12.     player->Draw();
13.
14.     // Present the renderer (update the window)
15.     SDL_RenderPresent(m_Renderer);
16. }
17.
```

Note: We do not have to explicitly invoke the player->Clean() function, it will be invoked when we invoke:

`TextureManager::GetInstance()->Clean();`

- Compile and run the program

At this point you should see the figure animated in Figure 83.

8. Sprite Sheet Animation for SDL Game – Part 2

In this video, we cleanup/change the code to make it more convenient to create and manage objects that have animation in our game. The Warrior class should *only* be managing details pertaining to the Warrior character and not handling the Animation details shown below:

```
1.      private:
2.          int m_Row, m_Frame, m_FrameCount;
3.          int m_AnimSpeed;
```

Composition vs. Inheritance

In object-oriented programming, inheritance establishes an "***is-a***" relationship between classes, where a subclass inherits properties and behaviors from its parent class. Composition, on the other hand, creates a "***has-a***" relationship, where one object contains or uses another object's functionality. Composition is generally favored for its greater flexibility and loose coupling, especially in complex, evolving systems, whereas inheritance can lead to rigid designs and tight coupling.

Elaboration:

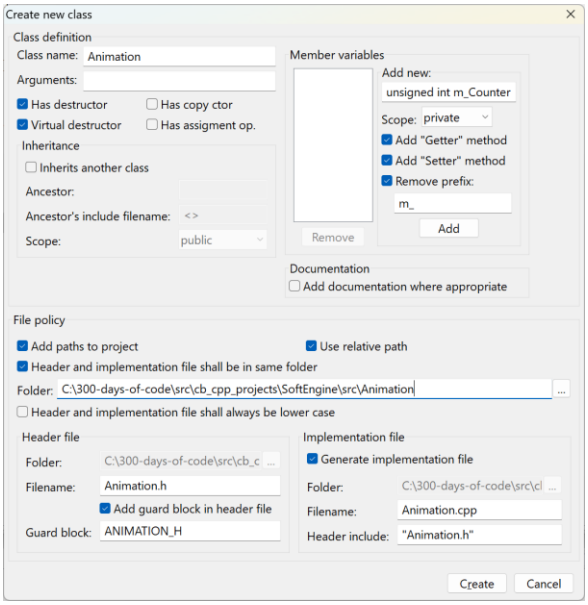
- **Inheritance:**
    - A class can inherit properties and methods from a parent class, creating a hierarchy of classes.
    - This can lead to code reuse and a more organized structure.
    - However, inheritance can create rigid designs and make it difficult to modify the code if changes are needed in the parent class.
- **Composition:**
    - A class contains instances of other classes, allowing it to use their functionality.
    - This approach provides greater flexibility and modularity, as objects can be swapped or modified without affecting the entire system.
    - Composition promotes loose coupling, making it easier to maintain and evolve the code.

- **Key Differences:**
    - **Relationship:** Inheritance establishes an "is-a" relationship, while composition establishes a "has-a" relationship.
    - **Coupling:** Inheritance can lead to tight coupling between classes, whereas composition promotes loose coupling.
    - **Flexibility:** Composition offers greater flexibility and adaptability to changes in the system.
    - **Maintainability:** Loose coupling in composition makes it easier to maintain and evolve the code.

In this video we see the presenter chooses to use composition since a Warrior class is not an Animation but has an animation component to it.

Create Animation class

- Create a new Animation class (with .cpp) in a new folder src/Animation.
    - Add to the project when prompted
    - Add to Debug and Release targets



- Update Animation.h to capture all the elements required to animate a sprite.

```
1. #ifndef ANIMATION_H
2. #define ANIMATION_H
3.
4. #include <string>
```

```
 5. #include "SDL.h"
 6.
 7. class Animation
 8. {
 9.     public:
10.         Animation();
11.
12.         void Update();
13.         void Draw(float x, float y, int spriteWidth, int spriteHeight);
14.         void SetProps(std::string textureID, int spriteRow, int frameCount, int animSpeed,
SDL_RendererFlip flip=SDL_FLIP_NONE);
15.
16.     private:
17.         int m_SpriteRow;            // holds the sprite sheet row we are processing
18.         int m_SpriteFrame;          // holds the sprite or frame in the row we will draw next
19.         int m_AnimSpeed;            // holds the number of milliseconds we will display a frame
20.         int m_FrameCount;           // holds the total number of frames per row
21.         std::string m_TextureID;    // holds the sprite sheet texture/image ID
22.         SDL_RendererFlip m_Flip;    // holds the orientation of the image to display
23. };
24.
25. #endif // ANIMATION_H
```

Notice we moved all the member variables in Warrior to manage the sprites on a sprite sheet to the Animation class.

- Update Animation.cpp

```
 1. #include "Animation.h"
 2. #include "TextureManager.h"
 3.
 4. Animation::Animation()
 5. {
 6.     //ctor
 7. }
 8.
 9. void Animation::Update()
10. {
11.    m_SpriteFrame = (SDL_GetTicks() / m_AnimSpeed) % m_FrameCount;
12. }
13.
14. void Animation::Draw(float x, float y, int spriteWidth, int spriteHeight)
15. {
16.     TextureManager::GetInstance()->DrawFrame(m_TextureID, x, y, spriteWidth, spriteHeight,
m_SpriteRow, m_SpriteFrame, m_Flip);
17. }
18.
19. void Animation::SetProps(std::string textureID, int spriteRow, int frameCount, int animSpeed,
SDL_RendererFlip flip)
20. {
21.    m_TextureID = textureID;
22.    m_SpriteRow = spriteRow;
23.    m_FrameCount = frameCount;
24.    m_AnimSpeed = animSpeed;
25.    m_Flip = flip;
26. }
```

Update Warrior to utilize the new Animation class

- Edit the Warrior.h file
    - Comment out the m_Row, m_Frame, m_FrameCount and m_AnimSpeed
    - Add include for Animation.h
    - Create a new private member variable m_Animation

```
#ifndef WARRIOR_H
#define WARRIOR_H

#include <Character.h>
#include <Animation.h>

class Warrior : public Character
{
    public:
        Warrior(Properties* props);

        virtual void Draw();
        virtual void Update(float dt);
        virtual void Clean();


    private:
        //int m_Row, m_Frame, m_FrameCount;
        //int m_AnimSpeed;
        Animation* m_Animation;
};

#endif // WARRIOR_H
```

- Change the Warrior.cpp to match the changes:

```
#include "Warrior.h"
#include "SDL.h"
#include "TextureManager.h"


Warrior::Warrior(Properties* props): Character(props)
{
    m_Animation = new Animation();
    m_Animation->SetProps(m_TextureID, 0, 6, 80, SDL_FLIP_HORIZONTAL);
}

void Warrior::Draw()
{
    m_Animation->Draw(m_Transform->X, m_Transform->Y, m_Width, m_Height);
}

void Warrior::Update(float dt)
```

```
18. {
19.     m_Animation->Update();
20. }
21.
22. void Warrior::Clean()
23. {
24.     TextureManager::GetInstance()->Clean();
25. }
```

The Warrior class relies on the m_Animation to manage the frame being displayed on the sprite sheet.

Note: We maintained the fact that our sprite row starts on the count of 0 and did not adjust for starting the count at 1. This is important if you don't "see" anything come up on the screen – you may be on the wrong row!

Test the program

- Compile and run the program you should see the same animation for the Warrior player as before

Figure 90 - Our Warrior class making use of the Animation class

9. Rigid Body and Game Physics in SDL Game

The concept of **RigidBody** in a 2D platformer game is crucial for handling physics-based movement and interactions. In game development, a **RigidBody** represents an object that follows the rules of physics, meaning it can move, collide, and be influenced by forces like gravity or impulse.

Here's why **RigidBody** is useful in a **2D platformer**:

1. **Realistic Movement:** It allows characters and objects to move naturally, with proper acceleration and deceleration, rather than just teleporting between positions.

2. **Collision Handling:** A RigidBody ensures that the character or object properly interacts with the game world, preventing unwanted clipping through walls or falling through platforms.
3. **Gravity Effects:** In a platformer, gravity plays a big role. By using a **RigidBody**, developers can control how characters fall, jump, or bounce off surfaces.
4. **Force-based Interactions:** RigidBody lets developers apply forces like jumps, pushes, or explosions that feel dynamic and believable.
5. **Platform Behavior:** Moving platforms, spring-like surfaces, or rotating objects can behave naturally using physics-based movement.

Most game engines, like **Unity**, allow developers to set up **RigidBody2D** components, where you can tweak properties like mass, drag, constraints, and collision layers to fine-tune the physics behavior.

We must start with our Vector2D class we created in video #5.

- The ***Vector2D*** class represents a 2D vector for position, velocity, and acceleration.
- The ***RigidBody*** class handles movement, force application, and gravity simulation
- ***Physics Update*** where acceleration affects velocity, which then updates the position
- ***Gravity Simulation***, where gravity continuously pulls the player downward
- ***Force Application***, where jump forces and external influences modify acceleration

An example to demonstrate physics

We will examine a simple example that demonstrates the physics around jumping and gravity. This example is intentionally simple, so you get a feel for what we are trying to do in this video.

- Create a new project “physics_example”
- Replace the contents of the main.cpp with the following code:

```
#include <SDL.h>
#include <iostream>

// Screen dimensions
const int SCREEN_WIDTH = 800;
const int SCREEN_HEIGHT = 600;

// Gravity constant
const float GRAVITY = 0.2f;
```

```
10.
11. // Player structure with physics properties
12. struct Player {
13.     SDL_Rect rect;  // Position and size
14.     float velocityY;  // Vertical velocity
15.     bool isOnGround; // Ground check
16.
17.     Player(int x, int y, int w, int h) {
18.         rect = { x, y, w, h };
19.         velocityY = 0.0f;
20.         isOnGround = false;
21.     }
22.
23.     void applyGravity() {
24.         if (!isOnGround) {
25.             velocityY += GRAVITY;
26.         }
27.     }
28.
29.     void jump() {
30.         if (isOnGround) {
31.             velocityY = -12.0f; // Jump force
32.             isOnGround = false;
33.         }
34.     }
35.
36.     void update() {
37.         applyGravity();
38.         rect.y += static_cast<int>(velocityY);
39.
40.         // Simulate ground collision (simple floor check)
41.         if (rect.y + rect.h >= SCREEN_HEIGHT) {
42.             rect.y = SCREEN_HEIGHT - rect.h;
43.             velocityY = 0;
44.             isOnGround = true;
45.         }
46.     }
47. };
48.
49. // Main SDL application loop
50. int main(int argc, char* argv[]) {
51.     if (SDL_Init(SDL_INIT_VIDEO) < 0) {
52.         std::cerr << "SDL Initialization Failed: " << SDL_GetError() << std::endl;
53.         return -1;
54.     }
55.
56.     SDL_Window* window = SDL_CreateWindow("2D Platformer Physics",
57.                                           SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED,
58.                                           SCREEN_WIDTH, SCREEN_HEIGHT, SDL_WINDOW_SHOWN);
59.
60.     SDL_Renderer* renderer = SDL_CreateRenderer(window, -1, SDL_RENDERER_ACCELERATED);
61.
62.     Player player(50, 50, 50, 50); // Initialize player position
63.
64.     bool running = true;
65.     SDL_Event event;
66.
67.     while (running) {
68.         while (SDL_PollEvent(&event)) {
69.             if (event.type == SDL_QUIT) {
70.                 running = false;
71.             }
72.             if (event.type == SDL_KEYDOWN) {
73.                 if (event.key.keysym.sym == SDLK_SPACE) {
74.                     player.jump();  // Jump when Space is pressed
```

```
                }
            }
        }

        player.update(); // Update physics

        // Rendering
        SDL_SetRenderDrawColor(renderer, 0, 0, 0, 255); // Clear screen (black)
        SDL_RenderClear(renderer);

        SDL_SetRenderDrawColor(renderer, 255, 255, 255, 255); // Draw player (white)
        SDL_RenderFillRect(renderer, &player.rect);

        SDL_RenderPresent(renderer); // Display frame

        SDL_Delay(16); // Simulate ~60 FPS
    }

    SDL_DestroyRenderer(renderer);
    SDL_DestroyWindow(window);
    SDL_Quit();

    return 0;
}

```

```
#include <SDL.h>
```

Line #1 adds the use of SDL functions to the program. I did not have to do the same setup we did for the current video project (adding SDL header and lib folders configuration) since I saved that configuration so that all my projects will start with same set of libraries.

```
// Gravity constant
const float GRAVITY = 0.2f;
```

One of the forces defined in the game is gravity and the other is the one initiated on the “player” or block representing the player when they click on the “jump” key (space bar). If you want gravity to have a stronger effect just increase the const value.

We need to remember that GRAVITY is in the down direction of y on the screen. The next figure illustrates how this works:

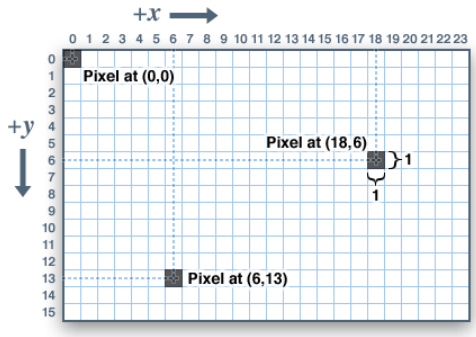


Figure 91 - The screen coordinates (from: https://learn.adafruit.com/adafruit-gfx-graphics-library/coordinate-system-and-units)

So to pull an object down we add +y value to it, this is why GRAVITY is set to 0.2f.

We start to define the Player (a rectangular block for us) at line 12:

```
12. struct Player {
13.     SDL_Rect rect;  // Position and size
14.     float velocityY;  // Vertical velocity
15.     bool isOnGround; // Ground check
```

The player will be represented on the screen by a rectangular block. The player will track two key values: current velocity (velocityY) and a boolean to indicate if the player is on the ground or not (isOnGround). The player can only have velocity in the Y direction and can only jump if it is on the ground.

Lines 17-21 initializes the player.

```
17.     Player(int x, int y, int w, int h) {
18.         rect = { x, y, w, h };
19.         velocityY = 0.0f;
20.         isOnGround = false;
21.     }
```

The (x,y) defines the top-left pixel location. We initialize the SDL_Rect. The player starts with 0 velocity and is not on the ground. We will start the program with the player falling down until it hits the ground.

The game loop start on line 67:

```
67.     while (running) {
```

In the game loop we do the following:

- Check for any keyboard events
    - If player closes the window => we stop running the game
    - If the player presses on SPACE BAR we invoke the jump command if the player is on the ground

```
29.     void jump() {
30.         if (isOnGround) {
31.             velocityY = -12.0f; // Jump force
32.             isOnGround = false;
33.         }
34.     }
```

Simulating a player jumping is the player moving up or in the -Y direction.

- Update the player location

```
36.     void update() {
37.         applyGravity();
38.         rect.y += static_cast<int>(velocityY);
39.
40.         // Simulate ground collision (simple floor check)
41.         if (rect.y + rect.h >= SCREEN_HEIGHT) {
42.             rect.y = SCREEN_HEIGHT - rect.h;
43.             velocityY = 0;
44.             isOnGround = true;
45.         }
46.     }
```

We first applyGravity, that is we modify the current velocity of the player moving up by the value of GRAVITY pulling the player down:

```
23.     void applyGravity() {
24.         if (!isOnGround) {
25.             velocityY += GRAVITY;
26.         }
27.     }
```

We really don't have time in this example but the game loop has a 15 ms delay:

```
90.         SDL_Delay(16); // Simulate ~60 FPS
```

Since 1000 ms / 60 = 16ms an SDL_Delay(16) ensures the game loop runs at 60 frames or updates per second.

- Draw the player on the screen.

Figure 92 - animated gif of gravity and jumping physics

Every time through the loop the player's velocity is adjusted by applying GRAVITY value to it, until the player hits the bottom of the screen.

This example is quite simple (only two possible forces) and we only move in one direction (up or down).

Incorporating Physics in our SDL2 game

The topic of how to handle physics can fill a book and in fact there are many good books that cover the topic. The example program above hopefully gave you an intuitive feeling for what we are trying to do by adding physics to the game.

In a 2D platform game, physics plays a crucial role in creating a smooth, responsive, and realistic experience. Let's break it down:

1. Velocity – This refers to the speed and direction of your character's movement. In a platformer, velocity often determines how fast a character moves horizontally across the level or falls vertically due to gravity. If a character has a positive velocity moving right, they will continue in that direction until acted upon by another force (such as friction or acceleration).

2. Acceleration – Acceleration is the rate at which velocity changes over time. When a player holds the movement key, the character accelerates to its top speed rather than instantly reaching it. Similarly, when a character jumps, an initial burst of acceleration propels them upward, until gravity slows them down.

3. Gravity – Gravity constantly pulls objects down toward the ground. In most platformers, gravity ensures that characters fall back to the ground after jumping. The strength of gravity can be adjusted—higher gravity makes jumps feel heavier and shorter, while lower gravity allows for floatier movements.

4. Friction – Friction slows down movement when an object is in contact with a surface. In a platformer, friction affects how quickly a character stops when the player releases the movement key. Low friction results in sliding effects, while high friction allows for quick stops.

Putting It Together in Gameplay:

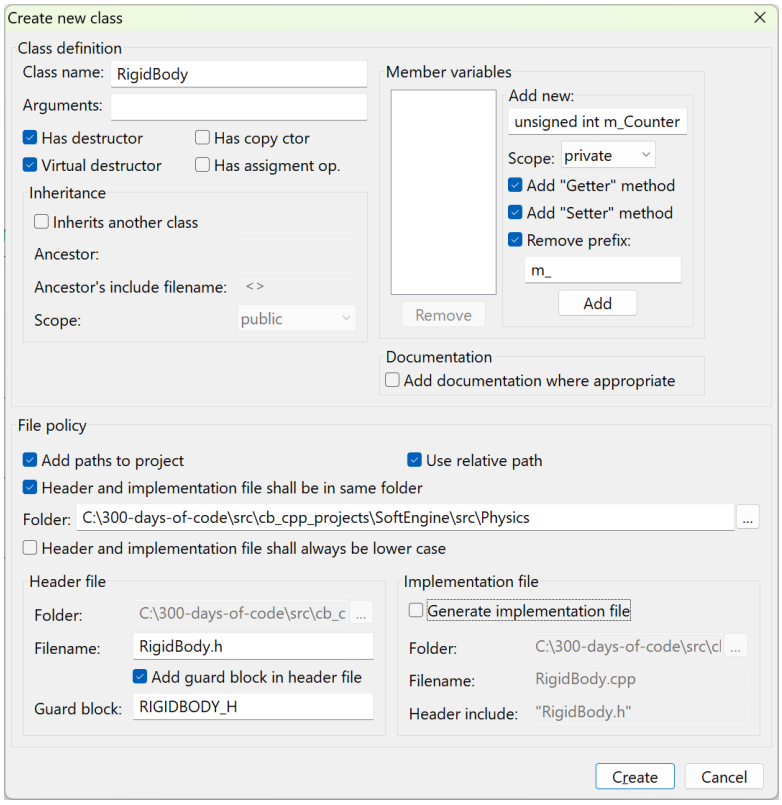
When a character jumps:

- **Upward velocity** is initially high.
- **Gravity** reduces velocity gradually until it reaches zero (the peak of the jump).
- As the character falls, **gravity accelerates them downward** until they land on the ground.

- If they land on a slippery surface, **low friction** could cause them to slide rather than stop immediately.
- If they are running, **acceleration** determines how quickly they reach top speed.

Creating a RigidBody.h file

- Create a RigidBody class (no implementation) in the Physics folder:



- We will be using Vector2D

```
#include "Vector2D.h"
```

- We will add the private member variables that define the elements that pertain to a rigid body:

```
1.      private:
2.          float m_Mass;
3.          float m_Gravity;
4.
5.          Vector2D m_Force;
6.          Vector2D m_Friction;
7.
8.          Vector2D m_Position;
```

```
 9.         Vector2D m_Velocity;
10.         Vector2D m_Acceleration;
11.
```

- Add #define for default values of mass and gravity:

```
1. // The default unit mass of a rigid body entity
2. #define UNIT_MASS 1.0f
3.
4. // The default gravity to use for any rigid body
5. #define GRAVITY 9.8f
```

- Remove the destructor
- Edit the constructor to set mass and gravity member variables to the default:

```
1.     public:
2.         RigidBody() {
3.             m_Mass = UNIT_MASS;
4.             m_Gravity = GRAVITY;
5.         }
6.
```

- Add setters for gravity and mass

```
1.     public:
2.         RigidBody() {
3.             m_Mass = UNIT_MASS;
4.             m_Gravity = GRAVITY;
5.         }
6.
7.     inline void SetMass(float mass) { m_Mass = mass;}
8.     inline void SetGravity(float gravity) { m_Gravity = gravity;}
9.
```

- Add similar functions for Force and Friction

```
 1.     // Force
 2.     inline void ApplyForce(Vector2D F) { m_Force = F;}
 3.     inline void ApplyForceX(float Fx) { m_Force.X = Fx; }
 4.     inline void ApplyForceY(float Fy) { m_Force.Y = Fy; }
 5.     inline void UnSetForce() { m_Force = Vector2D(0,0);}
 6.
 7.     // Friction
 8.     inline void ApplyFriction(Vector2D Fr) { m_Friction = Fr;}
 9.     inline void UnSetFriction() { m_Friction = Vector2D(0,0);}
10.
```

- Add getters for key attributes:

```
    // Add Getters
    inline float GetMass() { return m_Mass; }
    inline Vector2D Position() { return m_Position; }
    inline Vector2D Velocity() { return m_Velocity; }
    inline Vector 2D Acceleration() { return m_Acceleration;}
```

- Add code to update the key parameters for the rigid body

```
    // Update methods
    void Update(float dt) {
        m_Acceleration.X = (m_Force.X + m_Friction.X) / m_Mass;
        m_Acceleration.Y = m_Gravity + m_Force.Y/m_Mass;
        m_Velocity = m_Acceleration * dt;
        m_Position = m_Velocity * dt;
    }

```

The entire RigidBody.h file:

```
#ifndef RIGIDBODY_H
#define RIGIDBODY_H

#include "Vector2D.h"

// The default unit mass of a rigid body entity
#define UNIT_MASS 1.0f

// The default gravity to use for any rigid body
#define GRAVITY 9.8f

class RigidBody
{
    public:
        RigidBody() {
            m_Mass = UNIT_MASS;
            m_Gravity = GRAVITY;
        }

    inline void SetMass(float mass) { m_Mass = mass;}
    inline void SetGravity(float gravity) { m_Gravity = gravity;}

    // Force
    inline void ApplyForce(Vector2D F) { m_Force = F;}
    inline void ApplyForceX(float Fx) { m_Force.X = Fx; }
    inline void ApplyForceY(float Fy) { m_Force.Y = Fy; }
    inline void UnSetForce() { m_Force = Vector2D(0,0);}

    // Friction
    inline void ApplyFriction(Vector2D Fr) { m_Friction = Fr;}
    inline void UnSetFriction() { m_Friction = Vector2D(0,0);}


    // Add Getters
    inline float GetMass() { return m_Mass; }
    inline Vector2D Position() { return m_Position; }
    inline Vector2D Velocity() { return m_Velocity; }
    inline Vector2D Acceleration() { return m_Acceleration;}

    // Update methods
```

```
41.     void Update(float dt) {
42.         m_Acceleration.X = (m_Force.X + m_Friction.X) / m_Mass;
43.         m_Acceleration.Y = m_Gravity + m_Force.Y/m_Mass;
44.         m_Velocity = m_Acceleration * dt;
45.         m_Position = m_Velocity * dt;
46.     }
47.
48.     private:
49.         float m_Mass;
50.         float m_Gravity;
51.
52.         Vector2D m_Force;
53.         Vector2D m_Friction;
54.
55.         Vector2D m_Position;
56.         Vector2D m_Velocity;
57.         Vector2D m_Acceleration;
58.
59. };
60.
61. #endif // RIGIDBODY_H
62.
```

Update Our Warrior class

We will now add code to our `Warrior` class to utilize the `RigidBody` class to see the effects of gravity on our warrior.

- Add a new `RigidBody` member variable to `Warrior.h`:

```
 1. #ifndef WARRIOR_H
 2. #define WARRIOR_H
 3.
 4. #include "Character.h"
 5. #include "Animation.h"
 6. #include "RigidBody.h"
 7.
 8. class Warrior : public Character
 9. {
10.     public:
11.         Warrior(Properties* props);
12.
13.         virtual void Draw();
14.         virtual void Update(float dt);
15.         virtual void Clean();
16.
17.
18.     private:
19.         Animation* m_Animation;
20.         RigidBody* m_RigidBody;
21. };
22.
23. #endif // WARRIOR_H
24.
```

- Initialize our m_RigidBody in Warrior constructor:

```
1. Warrior::Warrior(Properties* props): Character(props)
2. {
3.     m_RigidBody = new RigidBody();
4.     m_Animation = new Animation();
5.     m_Animation->SetProps(m_TextureID, 0, 6, 80, SDL_FLIP_HORIZONTAL);
6. }
```

- Add code in the Warrior.cpp Update function to Update our RigidBody and apply to the Warrior:

```
1. void Warrior::Update(float dt)
2. {
3.     m_RigidBody->Update(0.2);
4.     m_Transform->TranslateX(m_RigidBody->Position().X);
5.     m_Transform->TranslateY(m_RigidBody->Position().Y);
6.
7.     m_Animation->Update();
8. }
```

Figure 93 - Our Warrior moving down due to gravity

We only expect to see movement in the Y direction due to gravity.

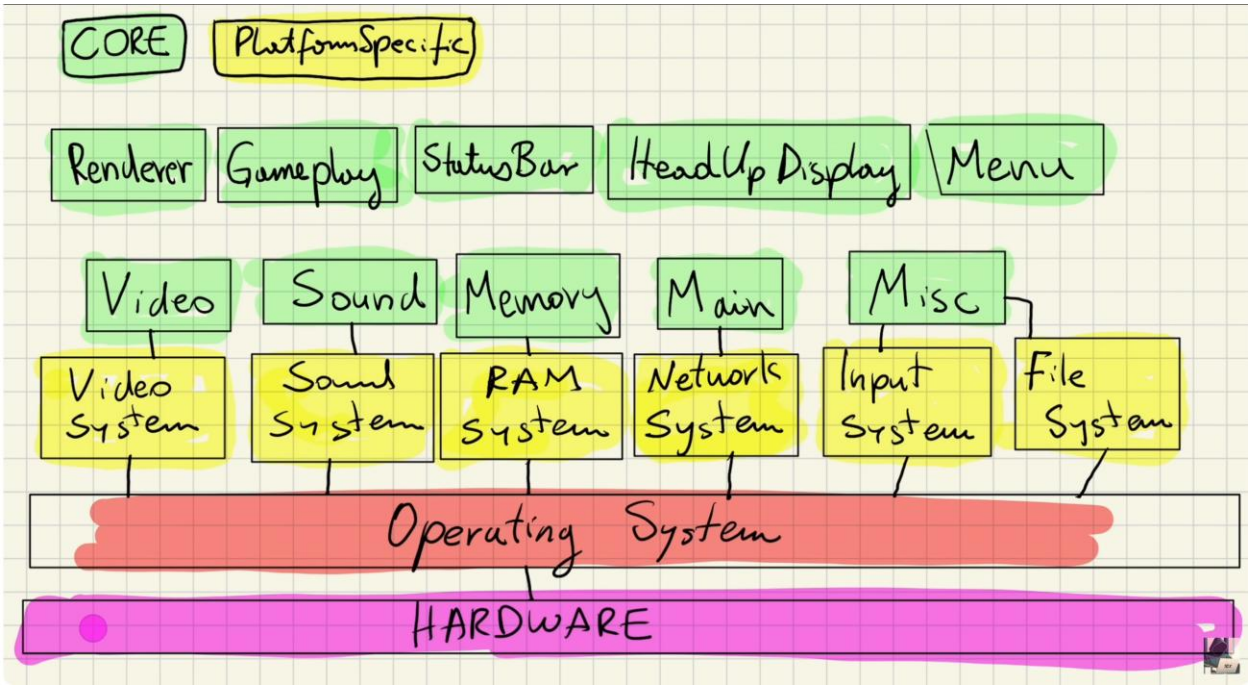
Let's add a force in the X direction to see what happens:

```
1. void Warrior::Update(float dt)
2. {
3.     m_RigidBody->Update(0.2);
4.     m_RigidBody->ApplyForceX(5.0);
5.     m_Transform->TranslateX(m_RigidBody->Position().X);
6.     m_Transform->TranslateY(m_RigidBody->Position().Y);
7.
8.     m_Animation->Update();
9. }
```

The player now moves in two directions – down due to gravity and right due to some force:

Figure 94 - Our Warrior moving down and to the right

Run – the player falls to the right



Tutorial: Create 2D Game Engine using C++

From: https://www.youtube.com/watch?v=cqL3jvlU61c&ab_channel=Tariq10x

Web sites to Learn SDL2

- https://lazyfoo.net/tutorials/SDL/index.php
- https://wiki.libsdl.org/SDL2/FrontPage
- https://wiki.libsdl.org/SDL2_image
- https://www.ferzkopp.net/wordpress/2016/01/02/sdl_gfx-sdl2_gfx/
- https://www.freepik.com/
- http://programarcadegames.com/
- https://giphy.com/
- https://github.com/nsklaus/SoftEngine
- https://forum.gdevelop.io/t/solved-how-do-i-slice-a-sprite-sheet/37755
- https://box2d.org/

Syntax

Function Parameters

Returns

Remarks