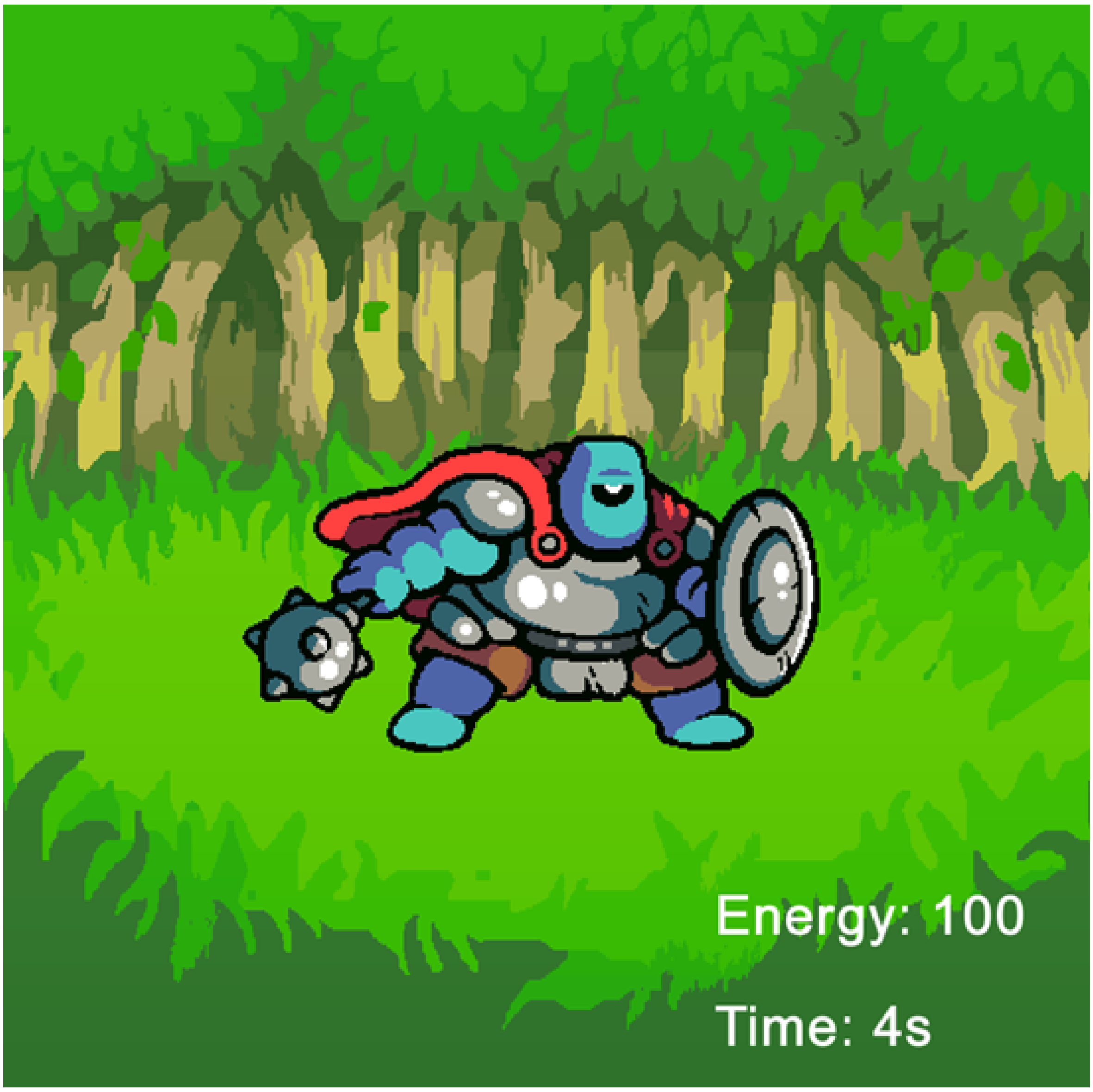
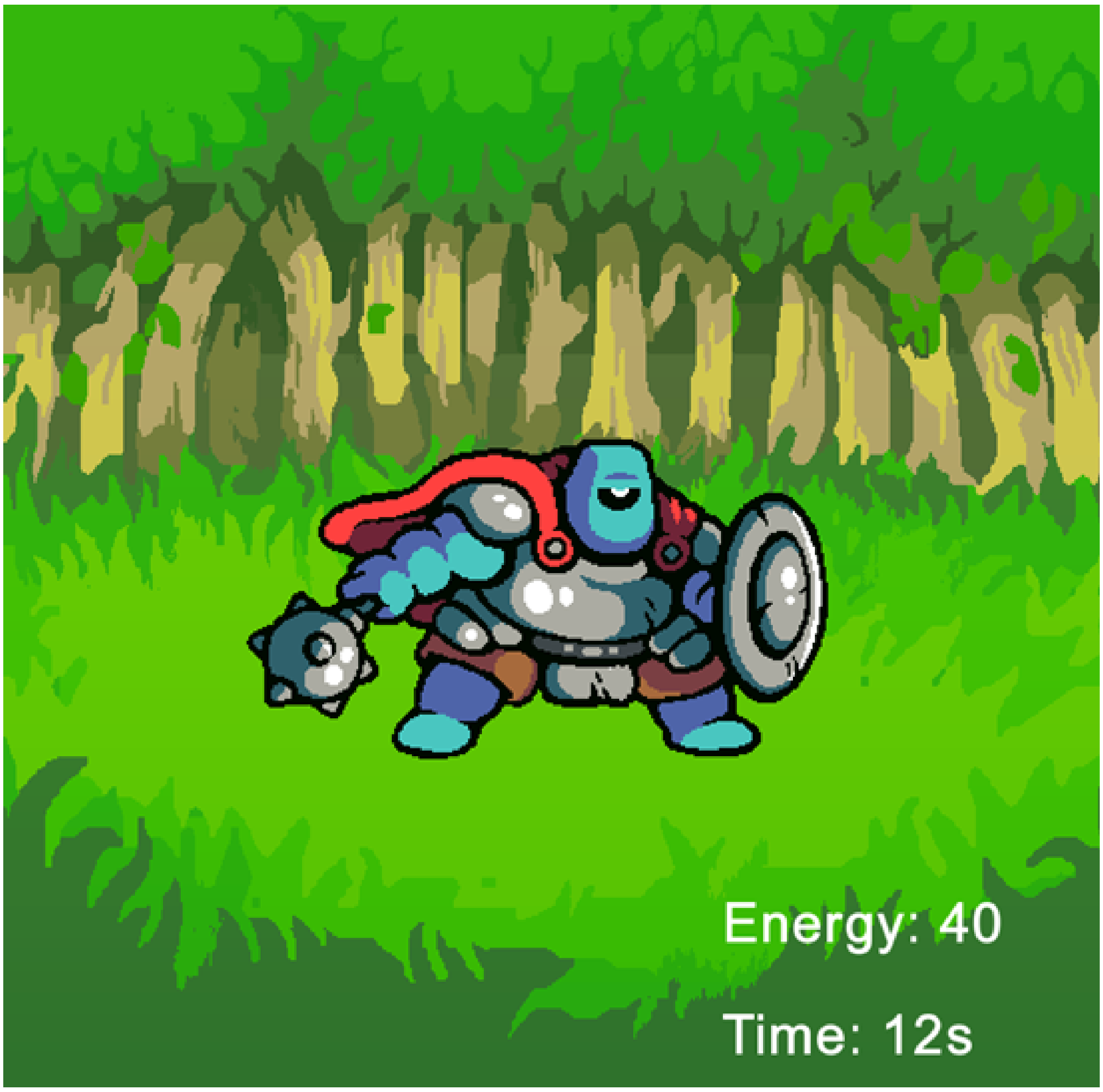
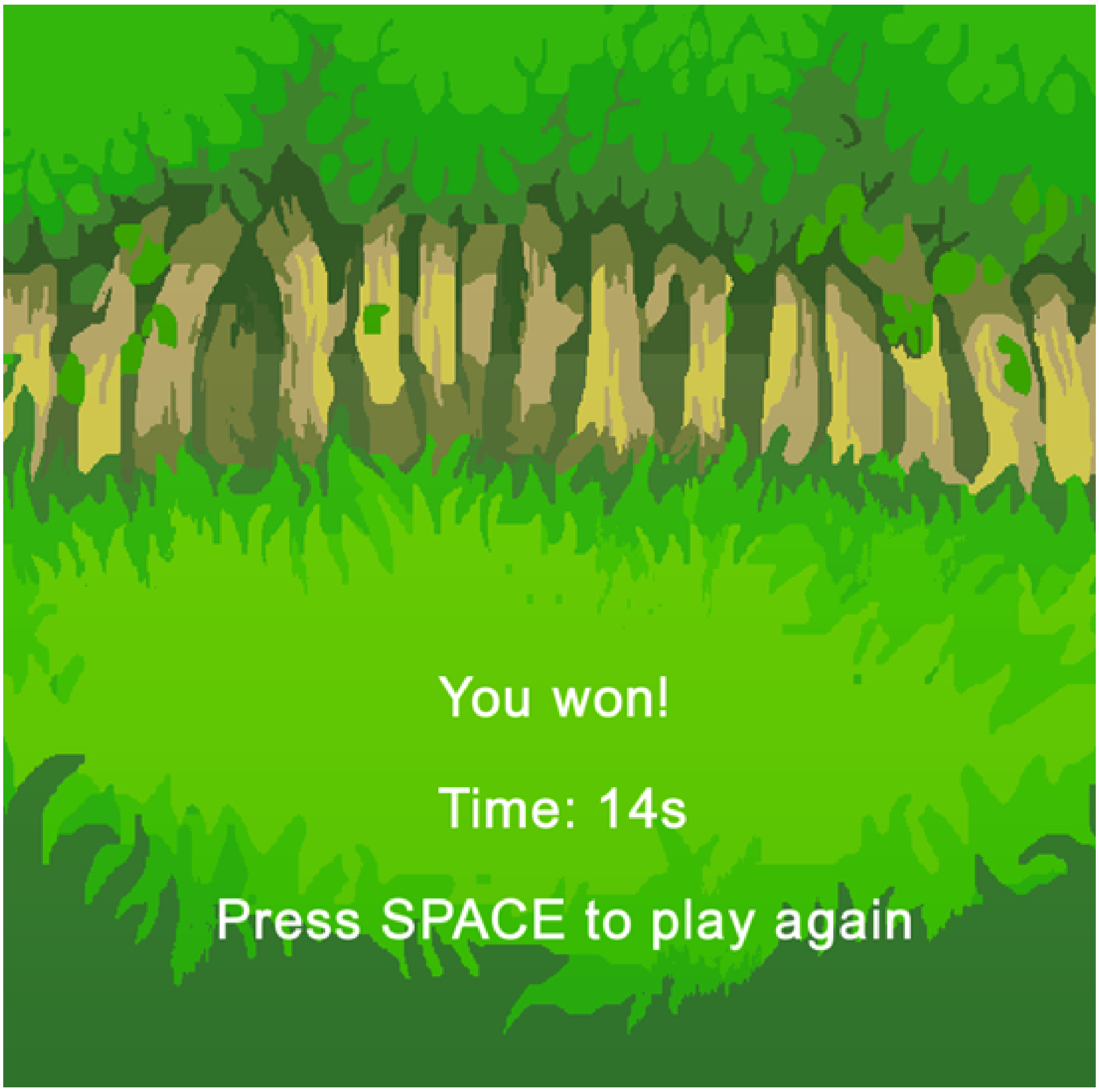
Hi everyone and welcome to our course on SFML with C++! Throughout this course, we will explore how to use the **library SFML to incorporate multimedia functionality in our C++ programs**. We will first cover several important components of SFML in detail. Then we will put all of the concepts together for our final project. By the end, we will have built a neat little point and click game in which the player clicks on an enemy to decrease energy and play a sound and allows a reset when the energy is at 0. It will display the current energy of the enemy along with the time it took to reduce the energy to 0. Here are three stages of the game:




Energy: 40
Time: 12s


You won!
Time: 14s
Press SPACE to play again

Before we start writing any code, let’s talk about what SFML is. **SFML** is an acronym that stands for **Simple and Fast Multimedia Library**. As the name would imply, it is a **software library used to add multimedia functionality** to applications and allows for seamless hardware-software communication. Although it is not a game engine, we often use the **data types** that it provides to represent game concepts such as drawing sprites and text, playing audio, and capturing and responding to user interactions. There are **5 main modules** that each contain tools for a different aspect of multimedia software: **Graphics, Window, Audio, System, and Network**. We will access SFML code through the C++ API although users can incorporate SFML into projects written in other languages.

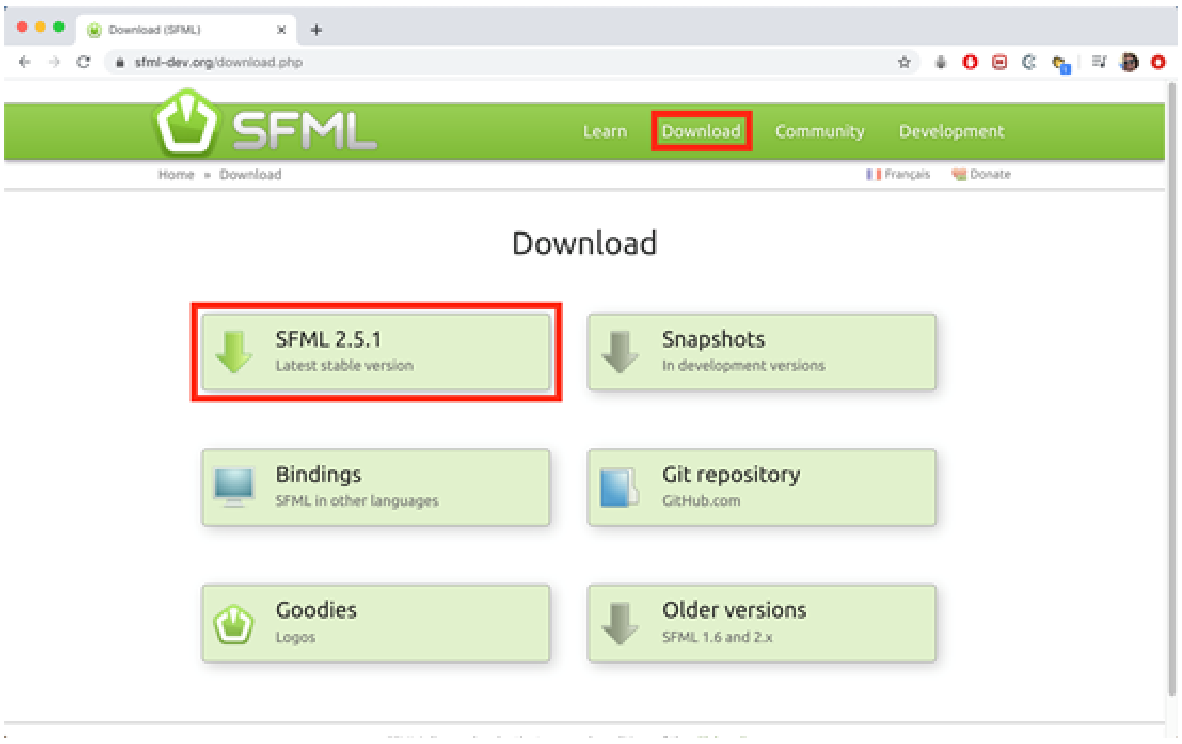
So SFML sounds pretty great, right? But why would we use this vs some other libraries out there? For starters, **SFML is written in C++** so the integration into C++ programs is seamless. Once the library is added, it’s extremely easy to get up and running and with its extensive data types, **it’s easy to capture and customize any kind of data that a multimedia program** would need. It’s great for writing 2D games, although it isn’t well-suited for 3D games and is faster than other platforms like Pycharm for Python. It’s used in many real-world production apps which serves as a testament to its reliability. All in all, if you need multimedia functionality in a C++ program, there is no better option!

Now that we know a bit about SFML, we need to install the library onto our system. The installations for Mac and PC (we won't cover Linux here) are slightly different so we will start with the Mac installation first. PC users, feel free to skip to the PC installation section.

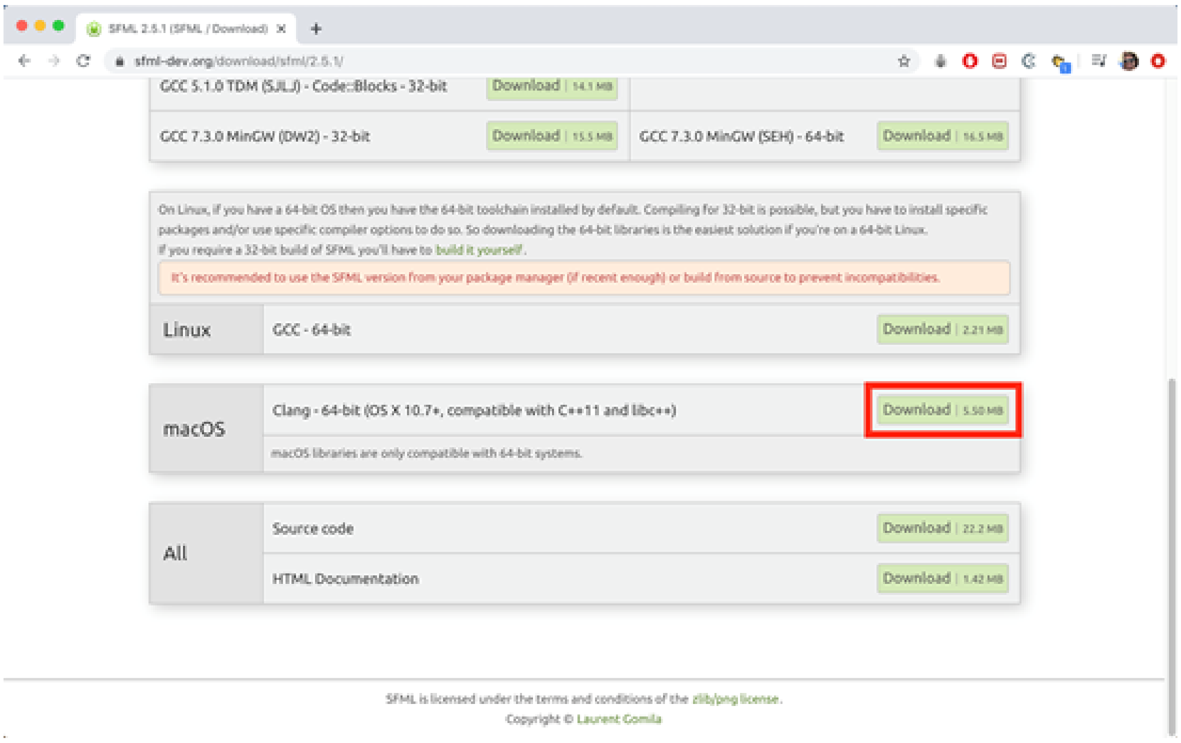
First, navigate to the SFML website…

https://www.sfml-dev.org/

… and click on "Download". Then select **"SFML 2.5.1"** or whatever the latest version is:



Once there, scroll down to where you see MacOS and click on the Download button:



SFML 2.5.1 (SFML / Download)
sfml-dev.org/download/sfml/2.5.1/

GCC 5.1.0 TDM (SJLJ) - Code::Blocks - 32-bit	Download | 14.1 MB	
GCC 7.3.0 MinGW (DW2) - 32-bit	Download | 15.5 MB	GCC 7.3.0 MinGW (SEH) - 64-bit — Download | 16.5 MB

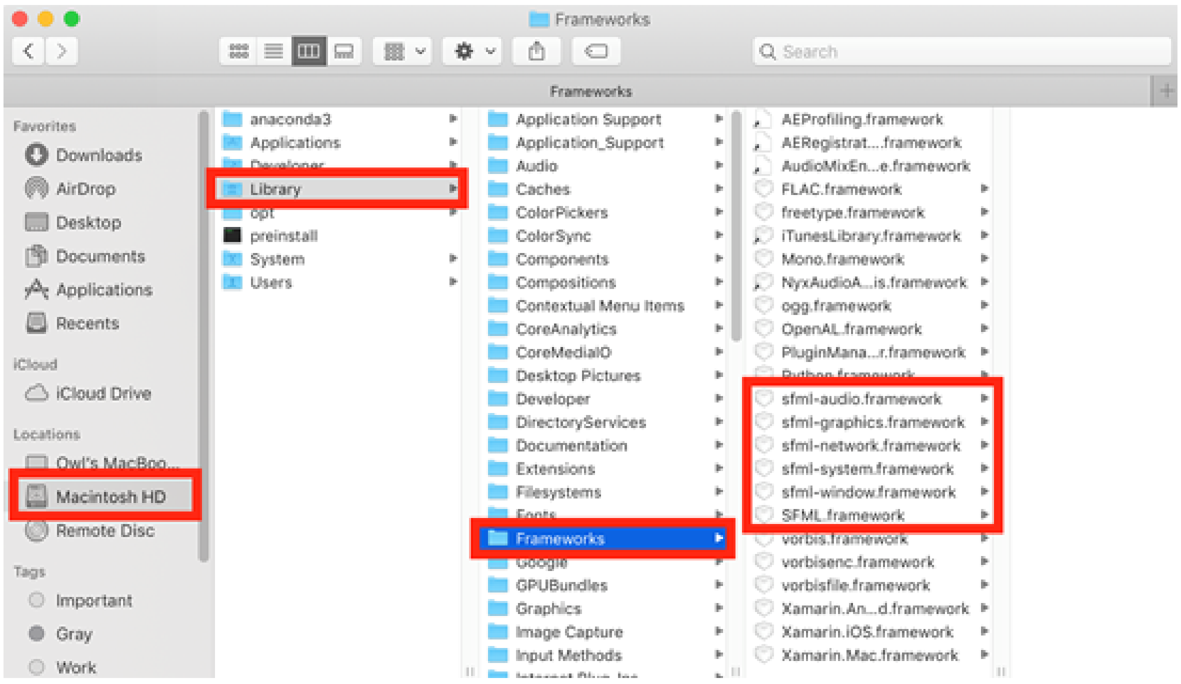
On Linux, if you have a 64-bit OS then you have the 64-bit toolchain installed by default. Compiling for 32-bit is possible, but you have to install specific packages and/or use specific compiler options to do so. So downloading the 64-bit libraries is the easiest solution if you're on a 64-bit Linux.
If you require a 32-bit build of SFML you'll have to build it yourself.

It's recommended to use the SFML version from your package manager (if recent enough) or build from source to prevent incompatibilities.

Linux	GCC - 64-bit	Download | 2.21 MB
macOS	Clang - 64-bit (OS X 10.7+, compatible with C++11 and libc++) macOS libraries are only compatible with 64-bit systems.	Download | 5.50 MB
All	Source code	Download | 22.2 MB
	HTML Documentation	Download | 1.42 MB

SFML is licensed under the terms and conditions of the zlib/png license.
Copyright © Laurent Gomila

This downloads a file called **"SFML-2.5.1-macOS-clang.tar.gz"**. Unzip the downloaded file by double-clicking on it. Now open the folder and you should see several folders and a few .md files. Open up the **"Frameworks" folder**. Select all of the **.framework files** and copy them into the location **"Library/Frameworks"** folder like this:



Although we probably don't need them in this course, it's a good idea to also copy the contents of the **"extlibs" folder** (in the downloaded SFML package) into Library/Frameworks as well. Once that's done, you've successfully installed SFML! You should be able to #include any SFML module in your projects now. However, when compiling and running C++ projects with SFML, you will likely need to link the frameworks in the compile command.

For now, let’s start a new project folder and open the folder up in **your choice of text editor**. Again, VSC is recommended but it doesn’t matter too much. If you are using VSC, you will want to install the Microsoft C/C++ extension and make sure you set the correct compiler in your Command Palette (Clang++) otherwise you will get a lot of annoying error messages when writing your code. **Create a main.cpp** file and copy the contents of the file **“templateCode.cpp”** into it. The file should be in your source code files. The reason we do this is because we want to compile and run everything though main.cpp but we will use the code in templaceCode.cpp to get a feel for how SFML works. Alternatively, copy it from here:

```
#include <SFML/Graphics.hpp>

int main()
{
  sf::RenderWindow window(sf::VideoMode(200, 200), "SFML works!");
  sf::CircleShape shape(100.f);
  shape.setFillColor(sf::Color::Green);

  while (window.isOpen())
  {
    sf::Event event;
    while (window.pollEvent(event))
    {
      if (event.type == sf::Event::Closed)
      window.close();
    }

    window.clear();
    window.draw(shape);
    window.display();
    }

  return 0;
}
```

We’ll examine the code in detail later but essentially, it will just show a **small window with a green circle** in it when it runs. We will compile and run our program through Terminal so open Terminal and navigate to the directory by running:

```
cd <path-to-directory>
```

If the project was saved in a folder call “sfml-project” on the Desktop for example, you would run the command:

```
cd Desktop/sfml-project
```

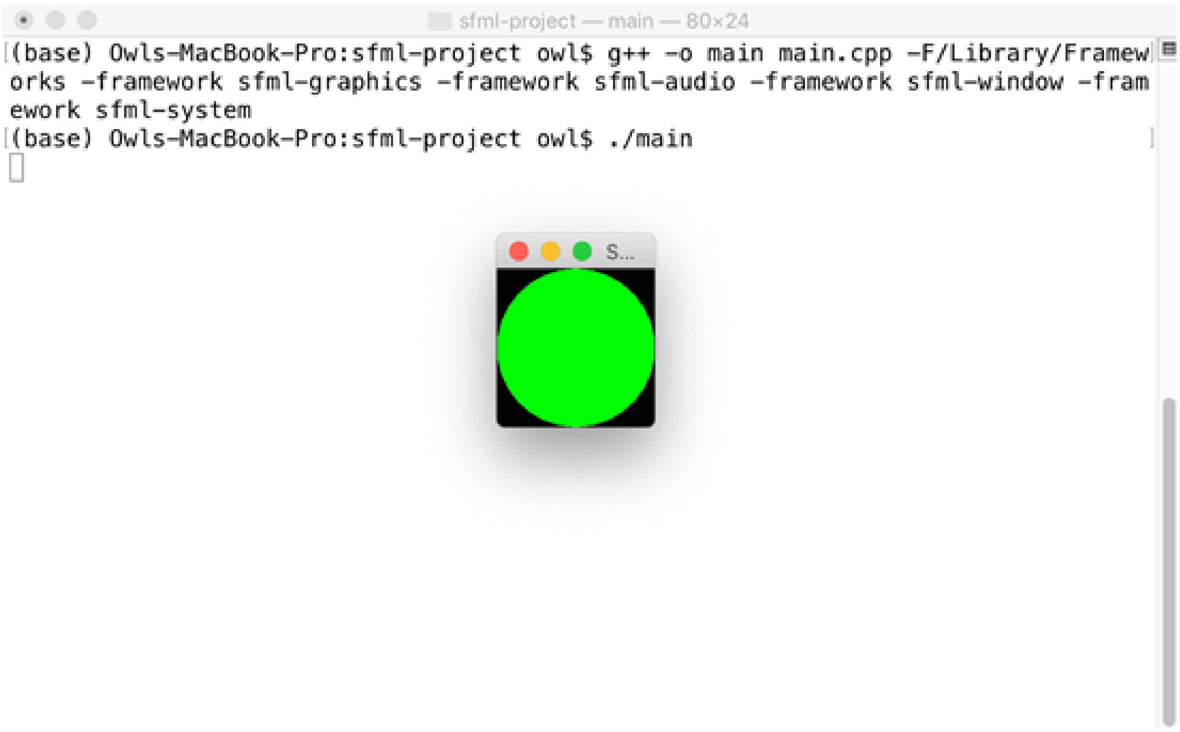
When compiling the code, **we need to tell the compiler where to find the SFML frameworks** as well as manually link the framework of each module that we need. As we will ultimately need Graphics, Audio, Window, and System, we may as well link them all now by running the command:

```
g++ -o main main.cpp -F/Library/Frameworks -framework sfml-graphics -framework sfml-
audio -framework sfml-window -framework sfml-system
```

This invokes the Clang++ compiler (run through g++ and creates an executable called “main” from the contents of main.cpp. Once we compile the project, we can run it with:

```
./main
```

This runs the **“main” executable**. You should see something like this:



```
(base) Owls-MacBook-Pro:sfml-project owl$ g++ -o main main.cpp -F/Library/Framew
orks -framework sfml-graphics -framework sfml-audio -framework sfml-window -fram
ework sfml-system
(base) Owls-MacBook-Pro:sfml-project owl$ ./main
```

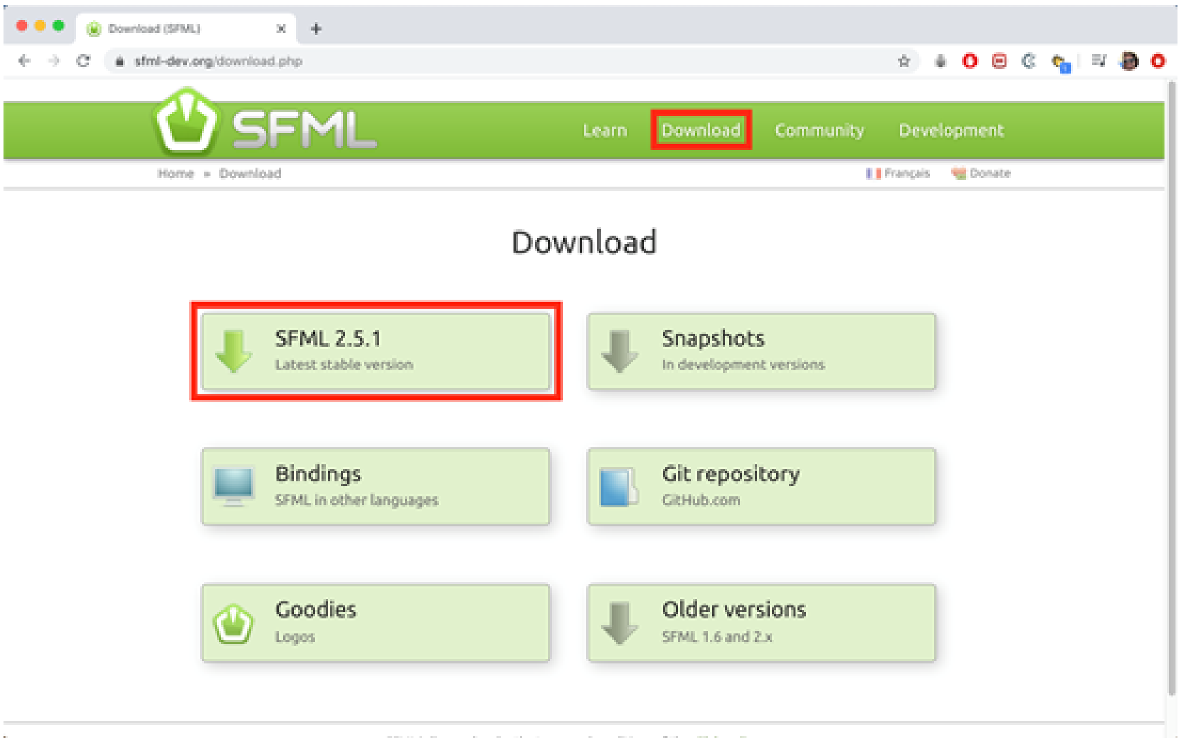
That’s it! You have SFML up and running and we can not start to break apart the starter code and talk about how SFML works.

The instructions in the video for this particular lesson have been updated, please see the lesson notes below for the corrected code.

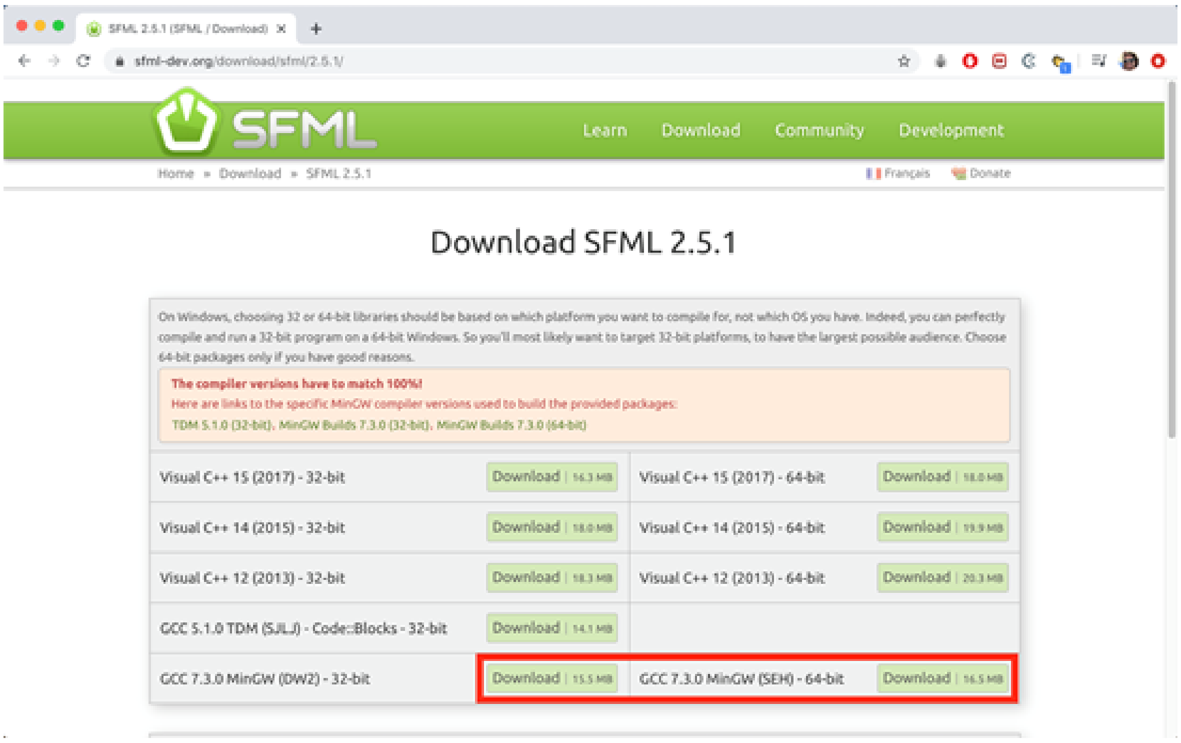
If you are running a PC instead of a Mac, follow these steps to install SFML on your system and run the starter project. Unfortunately, we don't support linux systems although the setup is similar to that of a PC.

First, navigate to the SFML website: https://www.sfml-dev.org/

...and click on "Download". Then select **"SFML 2.5.1"** or whatever the latest version is:



Once there, **choose the version of SFML that matches your compiler** and click on the download button. This will likely be either the **32 or 64 bit MinGW** version:


SFML 2.5.1 (SFML / Download)
sfml-dev.org/download/sfml/2.5.1/
SFML
Learn
Download
Community
Development
Home » Download » SFML 2.5.1
Français
Donate

Download SFML 2.5.1

On Windows, choosing 32 or 64-bit libraries should be based on which platform you want to compile for, not which OS you have. Indeed, you can perfectly compile and run a 32-bit program on a 64-bit Windows. So you'll most likely want to target 32-bit platforms, to have the largest possible audience. Choose 64-bit packages only if you have good reasons.

The compiler versions have to match 100%!
Here are links to the specific MinGW compiler versions used to build the provided packages:
TDM 5.1.0 (32-bit), MinGW Builds 7.3.0 (32-bit), MinGW Builds 7.3.0 (64-bit)

Visual C++ 15 (2017) - 32-bit	Download | 16.3 MB	Visual C++ 15 (2017) - 64-bit	Download | 18.0 MB
Visual C++ 14 (2015) - 32-bit	Download | 18.0 MB	Visual C++ 14 (2015) - 64-bit	Download | 19.9 MB
Visual C++ 12 (2013) - 32-bit	Download | 18.3 MB	Visual C++ 12 (2013) - 64-bit	Download | 20.3 MB
GCC 5.1.0 TDM (SJLJ) - Code::Blocks - 32-bit	Download | 14.1 MB		
GCC 7.3.0 MinGW (DW2) - 32-bit	Download | 15.5 MB	GCC 7.3.0 MinGW (SEH) - 64-bit	Download | 16.5 MB

This downloads a file called “SFML-2.5.1-windows-gcc-7.3.0-mingw-32-bit.zip” or something similar. Unzip the downloaded file by double-clicking on it to reveal a folder called “SFML-2.5.1” (or whichever version you downloaded). Rename it to just **“SFML” and move the folder into the C:\ folder**. Once that’s done, you’ve successfully installed SFML! You should be able to #include any SFML module in your projects now. However, when compiling and running C++ projects with SFML, you will likely need to link the dynamic libraries in the compile command.

For now, let’s start a new project folder and open the folder up in **your choice of text editor**. Again, VSC is recommended but it doesn’t matter too much. If you are using VSC, you will want to install the Microsoft C/C++ extension and make sure you set the correct compiler in your Command Palette, otherwise you will get a lot of annoying error messages when writing your code. Create a **main.cpp file** and copy the contents of the file **“templateCode.cpp”** into it. The file should be in your source code files. The reason we do this is because we want to compile and run everything though main.cpp but we will use the code in templaceCode.cpp to get a feel for how SFML works. Alternatively, copy it from here:

```
#include <SFML/Graphics.hpp>

int main()
{
  sf::RenderWindow window(sf::VideoMode(200, 200), "SFML works!");
  sf::CircleShape shape(100.f);
  shape.setFillColor(sf::Color::Green);

  while (window.isOpen())
  {
    sf::Event event;
    while (window.pollEvent(event))
    {
      if (event.type == sf::Event::Closed)
      window.close();
    }
    window.clear();
    window.draw(shape);
    window.display();
    }

  return 0;
}
```

We'll examine the code in detail later but essentially, it will just show a small window with a green circle in it when it runs. We will compile and run our program through **Cmd Prompt** so open Cmd Prompt up and navigate to the directory by running:

```
cd <path-to-directory>
```

If the project was saved in a folder call "sfml-project" on the Desktop for example, you would run the command:

```
cd Desktop/sfml-project
```

When compiling the code, **we need to tell the compiler where to find the SFML libraries** as well as manually link the library of each module that we need. As we will ultimately need Graphics, Audio, Window, and System, we may as well link them all now by running the command:

```
g++ -IC:\SFML\include -c main.cpp -o main.o
```

This links all of the include files and creates the main.o file. Next, we want to link all of the library files and create the executable file. Do so with this command. **The following instructions have been updated, and differs from the video:**

```
g++ -LC:\SFML\lib -o main.exe main.o -lsfml-graphics -lsfml-audio -lsfml-
window -lsfml-system
```

This creates an executable called "main.ext" from the contents of main.cpp. Once we compile the project, we can run it with:

```
main.exe
```

This runs the **"main" executable**. Before running, you're going to need to put the executable in a folder with the sfml dynamic libraries (dlls). So, copy the bin folder from the SFML installation folder to inside your project and move your executable to inside this bin folder.

When running the executable you should see a small window pop up with the title: "SFML Works!"

and a black background filled with a green circle. That’s it! You have SFML up and running and we can not start to break apart the starter code and talk about how SFML works.

Now that SFML is installed, we will examine how it runs. Any time we make changes to the codebase, we need to **recompile the code and rerun it**. Any compile and run commands shown here moving forwards will be mac-specific but if you are using a PC, simply substitute in the commands for the PC versions; the code written will be exactly the same across OSs.

When running a program with SFML, there are the two main stages: **the setup and the run loop**. **The setup stage** is where we create any persistent entities that the program needs as it runs or at least at the beginning, such as player sprites, backgrounds, audio, start game texts, etc. **The run loop** is where we check for events, update the game state, and render items. We can see these outlined in the template code.

First of all, **we need to include the Graphics module** which actually gives us access to Window and System with the line:

```
#include <SFML/Graphics.hpp>
```

If we need access to other modules such as Audio or Network, we can include those too. Inside of **main()**, we start by creating a window:

```
sf::RenderWindow window(sf::VideoMode(200, 200), "SFML works!");
```

This is a variable called **"window"** that is of the type sf::RenderWindow. **We use a RenderWindow as a screen** onto which we can draw various items. This both declares and initializes the variable in one go rather than doing a traditional:

```
sf::RenderWindow window = RenderWindow(sf::VideoMode(200, 200), "SFML works!");
```

RenderWindow takes two arguments: **a VideoMode object** that takes width and height, respectively and **a title string** to display as a header on the window. In this case, the window is 200 pixels wide and 200 pixels high. Next, we create a CircleShape object of radius 100 and fill it with the colour: sf::Color::Green . Creating the shape isn't enough; we have to draw it to the window but more on that in a second.

We have our initial state set up so we then start the run loop. We execute these commands as long as the window is open (window.isOpen() == true):

```
while (window.isOpen()) {

}
```

In the loop, **we poll for events** such as button presses, mouse clicks, or window closes:

```
sf::Event event;
while (window.pollEvent(event))
{
  if (event.type == sf::Event::Closed)
  window.close();
}
```

We have to poll for events in a while loop because there may be **multiple interactions triggering multiple events**. Above, we are only interested in the “Closed” event which will close the window and end the run loop and thus, the program.

After the loop, **we update state and draw objects** to the window although in our case, there is no state that needs to be updated. To draw objects, we first clear the window to make sure previous frames don’t interact with new ones, **then we draw whatever items we want** (in this case, just the circle shape we created earlier, and then display the window:

```
window.clear();
window.draw(shape);
window.display();
```

Now that we understand the basics of how to set an SFML program up and how a run loop works, let’s explore individual components of SFML needed for our game.

We've seen how to draw a basic shape but realistically, most of our games will use more than shapes. We will want to place **graphics, icons, and other images into our games** and we do so via **Sprites**. Setting up a Sprite is generally done in two steps with SFML: first, we load in a **texture**, then we create a **sprite** and pass in the texture. Once we have the sprite set up, we can set attributes such as the size and position and then display it by drawing it into the window during the run loop.

In our project, we should create a **folder called assets** that contains any assets needed in our game. We only have 4 so there's no need to further divide the assets folder. **Copy the "enemy.png" and "background.png" images, the "damage.ogg" audio file, and the "Arial.ttf" font files from your provided files and paste them into the assets folder**. We will only work with the Enemy.png image for now. To create a texture, we first create a **texture variable** and then call the **loadFromFile()** function, passing in the path to the file as a string. It returns true if successful and false otherwise so we should perform a boolean test to see if the load was successful. In our main function, write this code after the code to create the window and before the run loop:

```
sf::Texture texture;
if (!texture.loadFromFile("assets/enemy.png")) {
  std::cout << "Could not load enemy texture" << std::endl;
  return 0;
}
```

This will create a variable of type **sf::Texture** and load the image into it. If the load fails, we print an error message and return. Sometimes, the load for an asset fails for some unknown reason **so we should handle any errors properly**. You will need to #include <iostream> at the top of main.cpp as well. Once we have the texture, we can create a sprite object called enemySprite and pass in the texture like this:

```
sf::Sprite enemySprite;
enemySprite.setTexture(texture);
```

The sprite will only be as big as the image provided and will **default to be at position (0,0)** or in the upper left corner. We can change the position like this:

```
enemySprite.setPosition(sf::Vector2f(100,100));
```

Which brings the sprite 100 pixels out from the left and 100 pixels out from the right and increase the size of the sprite by scaling it like this:

```
enemySprite.scale(sf::Vector2f(1,1.5));
```

This will scale the width of the sprite by 1 (no change) and scale the height to 1.5x its original height. Notice how both functions take an **sf::Vector2f** as inputs and **each needs an x and y value**. In order to draw the sprite, we change the draw code to this:

```
window.clear();
window.draw(enemySprite);
window.display();
```

We always have to clear the window before drawing anything and display the window after we draw to it.

If we can draw sprites, drawing text is no more difficult. Instead of loading a texture, **we load a font**. Then we create a text object and set the font. After that, we can set attributes such as character size, fill colour, style, position, and the text itself. We start with the font:

```
sf::Font font;
if (!font.loadFromFile("assets/Arial.ttf")) {
  std::cout << "Could not load font file" << std::endl;
  return 0;
}
```

Again, we want to ensure that the asset is loaded properly first. Once we have it, we can create the text object and set attributes like this:

```
sf::Text text;
text.setFont(font);
text.setCharacterSize(30);
text.setFillColor(sf::Color::White);
text.setStyle(sf::Text::Bold);
text.setPosition(sf::Vector2f(100,100));
text.setString("Here is some text");
```

The character size is the size of the font in pixels. Note that setFillColor and setStyle take sf::Color and sf::Text values. For a full list of possible values, consult the documentation. Just like with the sprite, in order to draw it, we can replace the window.draw(enemySprite) with:

```
window.draw(text);
```

We can draw items on top of each other but **the object drawn last will appear on top**.

Playing audio is very simple with SFML and, like drawing sprites or texts, is a two-part process. First, we **load the audio file into an sf::SoundBuffer object** and then create an **sf::Sound object** and set the **buffer**. Before we can create sound objects, **we need access to the audio library** so add this #include statement at the top of main.cpp:

```
#include <SFML/Audio.hpp>
```

To load a sound buffer, we can do so like this:

```
sf::SoundBuffer buffer;
if (!buffer.loadFromFile("assets/damage.ogg")) {
  std::cout << "Could not load audio" << std::endl;
  return 0;
}
```

Then we create the sound object and set the buffer like this:

```
sf::Sound attackSound;
attackSound.setBuffer(buffer);
```

To play the sound, we call this code:

```
attackSound.play();
```

It will play the sound **once through and stop**, unless we play the sound inside of the run loop in which case it will continuously play until we close the game window.

There are many ways in which a user can interact with your program from pressing buttons, to opening and closing windows, and even simple events like moving a mouse. **Each of these interactions triggers an event** which we capture within the run loop and can either choose to handle or ignore. We could spend hours covering just events but we are only really interested in three for now: closing the window, clicking a mouse button, and pressing a keyboard key, specifically, the space bar.

To handle any event, we first need to **create an sf::Event object and call the function window.pollEvent()**. This gathers all of the events associated with a specific window (as you can have multiple windows open at once) and will continue to return true while there are events ongoing. For example, we are already handling the window close event with this code:

```
sf::Event event;
while (window.pollEvent(event)) {
  if (event.type == sf::Event::Closed) {
    window.close();
  }
}
```

This is only searching an event of type sf::Event::Closed, triggered when the user exits out of the current window. However, we also want to monitor mouse clicks and spacebar presses. Let’s start with mouse clicks. We can provide an else if case like this:

```
else if (event.type == sf::Event::MouseButtonPressed) {
  std::cout << "Mouse button pressed" << std::endl;
}
```

While this isn’t doing anything more than printing out “Mouse button pressed” whenever the user clicks the mouse somewhere in the window, we can see how easy it is to handle simple user events. To handle spacebar presses, we first need to **detect for a KeyPressed event** and then find out whether or not the key pressed was the spacebar like this:

```
else if (event.type == sf::Event::KeyPressed) {
  if (sf::Keyboard::isKeyPressed(sf::Keyboard::Space)) {
    std::cout << "Space bar pressed" << std::endl;
  }
}
```

The isKeyPressed() function will return true if the key we are looking for is the one that was pressed and false otherwise. If you try pressing anything other than the spacebar, nothing will happen but the spacebar will print the message “Space bar pressed”. We can put it all together like this;

```
sf::Event event;
while (window.pollEvent(event)) {
  if (event.type == sf::Event::Closed) {
    window.close();
  } else if (event.type == sf::Event::MouseButtonPressed) {
    std::cout << "Mouse button pressed" << std::endl;
  } else if (event.type == sf::Event::KeyPressed) {
    if (sf::Keyboard::isKeyPressed(sf::Keyboard::Space)) {
      std::cout << "Space bar pressed" << std::endl;
    }
  }
}
```

Note how all of this occurs within the run loop but before drawing anything. **Typically, the first thing we do in a run loop is process and handle interactions** as they may affect the state and thus the items that are drawn to the window.

We will often want to keep track of time in various parts of our programs. We can use this to trigger time-sensitive functionality or add stopwatches and game timers. In our case, we will just keep track of how many seconds it takes the player to defeat the enemy. To keep track of time, we create an **sf::Clock object** like this:

```
sf::Clock clock;
```

As soon as we create the object, **it starts running like a stopwatch**. If we want to reset it, we can do so with:

```
clock.restart();
```

To get the time from it, we need to **call the getElapsedTime() function** which returns an sf::Time object like this:

```
sf::Time time;
time = clock.getElaspedTime();
```

We can then extract specific time measurements such as seconds in the form of floats:

```
float seconds = time.asSeconds();
```

We may want to create a time variable and update it in the run loop like this:

```
sf::Clock clock;
sf::Time time;
while(window.isOpen()) {
  time = clock.getElapsedTime();
}
```

That way, we have the most up to date time values stored in the “time” variable.

At this point, we have learned all of the components that we need from the SFML library and can start on our actual project! We are building a point and click game so we **will need a background sprite, an enemy sprite, a timer, and some text objects** to display current energy and time passed. We also need to **handle mouse clicks, spacebar presses and window close events**. For the mouse clicks, we will need to perform some simple **collision detection** but more on that later. It's probably better practice to start building up our game world in a **GameWorld class**. That way, we can run any functionality we need to from main.cpp as well as perform any pregame setup in main if needed.

We should start separate gameWorld.h and gameWorld.cpp files in our project. G**ameWorld will need variables** to keep track of whether or not the game is over, the amount of damage to inflict on the enemy, a background texture and sprite, and the time passed for now. Later on, we will add the enemy and text objects as well. We also need functions to help us load the background, perform general setup, run the game, and of course, a constructor. We can add something like this in our gameWorld.h file:

```
#include <SFML/Graphics.hpp>

#ifndef GAMEWORLD_H
#define GAMEWORLD_H

class GameWorld
{
  bool isGameOver;
  int damage;
  sf::Texture backgroundTexture;
  sf::Sprite background;
  sf::Time time;
  bool loadBackground();
  public:
    GameWorld();
    bool performSetup();
    bool runGame();
};

#endif
```

As general good practice, we should include the #ifndef GAMEWORLD_H and #define GAMEWORLD_H along with the #endif and should make as much stuff private as possible.

Strangely enough, we need to maintain a **strong reference** to the backgroundTexture used in the background Sprite, otherwise the Sprite loses its texture. The loadBackground() and performSetup() functions return booleans because we want to check to see if they failed to load the assets. The runGame() function returns a boolean because we want to know whether or not to run the game again.

We can now turn to gameWorld.cpp to implement some of the functions starting at the top. Add this code to get things started:

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

GameWorld::GameWorld() {
  damage = 10;
}
```

We need access to everything within gameWorld.h, hence the include statement and we only want to set up the damage for now. We are hard-coding in the damage as 10 but feel free to change that or use a constructor parameter. Next, we want to load the background texture and set the background sprite in our loadBackground() function. We can do so like this:

```
bool GameWorld::loadBackground() {
  if (!backgroundTexture.loadFromFile("assets/background.png")) {
    std::cout << "Could not load background image" << std::endl;
    return false;
  }
  background.setTexture(backgroundTexture);
  background.scale(sf::Vector2f(1.6,2.25));
  return true;
}
```

Without the scaling, the background image will be quite small but these scaling values make the background cover the window nicely in a 1000×1000 window. Notice how this **returns false** if the asset loading fails. Next up, we want to perform basic set up by calling our performSetup() function like this:

```
bool GameWorld::performSetup() {
  isGameOver = false;
  return loadBackground();
}
```

We will add to this later but for now, let’s just set the isGameOver and call the loadBackground() function. Our **runGame function** will contain the **main run loop**. We want to start by creating our **window object and a clock object** within it. We want to check just for close events and draw only our background for now like this:

```
bool GameWorld::runGame() {
  sf::RenderWindow window(sf::VideoMode(1000, 1000), "Point and Click Game");
  sf::Clock clock;

  while (window.isOpen())
  {
    sf::Event event;
    while (window.pollEvent(event))
    {
      if (event.type == sf::Event::Closed) {
        window.close();
        return false;
      }
    }
    window.clear();
    window.draw(background);
    window.display();
  }
  return false;
}
```

Feel free to change the text or the size of the window but keep in mind that any scaling values will also need to change. Also note how **the function returns false when the window closes** so that we don’t get caught in an infinite loop.

In order to actually run this, we will create a GameWorld object in **main.cpp**, perform the set up, and call the run function. Clear out the code from the main() function in main.cpp, include gameWorld.cpp, and run the above functionality like this:

```
while(true) {
  GameWorld world = GameWorld();
  if (!world.performSetup()) {
    return 1;
  }
  if (!world.runGame()) {
    return 0;
  }
}
```

Within main, **we return if the performSetup() function fails as well as if the runGame() function returns false** (as the player will have closed the window in order for that to happen). Otherwise, we continuously run this in order to simulate the reset functionality. We will come back to this later. If we compile and run the program right now, we should see a medium-sized window with the background image filling it.

Now we need to create an enemy class and add an enemy to our game world. However, rather than just creating a sprite or even subclassing a sprite class, **it's better practice to create a custom class** that contains a sprite object as a variable. Start up an enemy.h and an enemy.cpp file. **The enemy will need a sprite and a texture as well as some energy (health)**. We will also attach **a sound and its sound buffer** to this class as the only time the sound plays is when the enemy takes damage. We will need a constructor as well as functions to perform setup, check for collision detection, take damage, and draw the sprite. Here is a complete example below:

```
#include <SFML/Graphics.hpp>
#include <SFML/Audio.hpp>

#ifndef ENEMY_H
#define ENEMY_H

class Enemy
{
  sf::Texture enemyTexture;
  sf::Sprite enemySprite;
  sf::SoundBuffer attackSoundBuffer;
  sf::Sound attackSound;
  public:
    int energy;
    Enemy(int);
    bool performSetup();
    bool checkIfHit(sf::Vector2i);
    bool takeDamage(int);
    void draw(sf::RenderWindow *);
};

#endif
```

Notice how **the constructor takes an int**; that will be used to set the initial energy. For now, we will ignore the checkIfHit and the takeDamage functions. Lastly, notice how the draw function is taking in an sf::RenderWindow pointer. This is because **we need to maintain a strong reference to the current window** when we draw the sprite, otherwise it would just copy the window and leave the original one unchanged.

Now we want to implement what we can (minus the checkIfHit() and takeDamage() functions) so let's turn to enemy.cpp. The initial setup should be fairly straightforward; we can follow the same logic when we're loading the texture and audio as with GameWorld, we can set the energy in the constructor, and the draw function will just call the window->draw() function (we use -> instead of . because we are passing in a pointer). Here is an example:

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

Enemy::Enemy(int e) {
  energy = e;
}

bool Enemy::performSetup() {
  if (!enemyTexture.loadFromFile("assets/enemy.png")) {
    std::cout << "Could not load enemy image" << std::endl;
    return false;
  }
  enemySprite.setTexture(enemyTexture);
  enemySprite.setPosition(sf::Vector2f(225,400));
  enemySprite.scale(sf::Vector2f(2,2));

  if (!attackSoundBuffer.loadFromFile("assets/damage.ogg")) {
    std::cout << "Could not load enemy audio" << std::endl;
    return false;
  }
  attackSound.setBuffer(attackSoundBuffer);

  return true;
}

void Enemy::draw(sf::RenderWindow * window) {
  window->draw(enemySprite);
}
```

The position and scale values are just some values found to work through trial and error, again with a 1000×1000 pixel window. It could be considered messy that we are loading the texture and audio in the same function but it doesn’t matter too much.

Now we can add the enemy to gameWorld.h. First, #include enemy.h:

```
#include “enemy.h”
```

And then add the variable as a private variable:

```
Enemy enemy;
```

To gain access to the Enemy functionality, we need to add:

```
#include “enemy.cpp”
```

Then, we need to call the **constructor in gameWorld.cpp** like this:

```
GameWorld::GameWorld(): enemy(100) {
  damage = 10;
}
```

And then change the performSetup() function to include the enemy setup like this:

```
bool GameWorld::performSetup() {
  isGameOver = false;
  enemy = Enemy(100);
  return loadBackground() && enemy.performSetup();
}
```

In order to draw the enemy, we can call on its **draw() function** within our run loop right after we draw the background like this:

```
enemy.draw(&window);
```

If you compile and run the program, you should now see the enemy sprite in the middle of the background.

We will now do something similar with a Text class. We are creating **a separate class because we want to display different texts under different conditions** and will actually display multiple text objects at any given time. We can start by creating **texts.h and texts.cpp files**. Texts will need a font, and 2 text objects to display energy and time as well as 3 text objects for the you-won-text, the end-game-time-text, and the play-again text. We will also want a function to perform setup of all texts and one to set up a specific text at a specific position. Finally, we will need functions for drawing the in-game and end-game texts. Here is a possible solution:

```
#include <SFML/Graphics.hpp>

#ifndef TEXTS_H
#define TEXTS_H

class Texts
{
  sf::Font font;
  void setUpText(sf::Text *, sf::Vector2f);
  public:
    sf::Text energyText;
    sf::Text timeText;
    sf::Text endGameWonText;
    sf::Text endGameTimeText;
    sf::Text endGameSpaceText;

    Texts();
    bool performSetup();
    void drawInGameText(sf::RenderWindow *, sf::Time, int);
    void drawEndGameText(sf::RenderWindow *, sf::Time);
};

#endif
```

Now there may be a lot of unexpected things going on here. For starters, our setUpText() function is taking in a pointer to a text. **This is because we will call this on each of the 5 text objects to save some space in the file but we need to reference the specific text objects rather than copying them**. It will become clearer in texts.cpp. Next the **drawInGameText** is taking in a window (just like the Enemy) as well as an **sf::Time object** so we can set the time string, and an energy value so we can set the energy string. The drawEndGameText is the same minus the energy because if that is called, the energy is at 0 anyway.

We can now turn to texts.cpp and add the includes and implement the constructor like this:

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

Texts::Texts() {
  endGameWonText.setString("You won!");
  endGameSpaceText.setString("Press SPACE to play again");
}
```

We can set the strings of two of the end game texts because they are never going to change. The other texts will all change at some point. Next up, we have the performSetup() function. We want to load the font then set up each individual text like this:

```
bool Texts::performSetup() {
  if (!font.loadFromFile("assets/Arial.ttf")) {
    std::cout << "Could not load font file" << std::endl;
    return false;
  }
  energyText.setFont(font);
  energyText.setCharacterSize(50);
  energyText.setFillColor(sf::Color::White);
  energyText.setStyle(sf::Text::Bold);
  energyText.setPosition(sf::Vector2f(650,800));
  return true;
}
```

However, that's only setting up one of the texts and it's annoying, not to mention bad practice, to set all the texts up this way if they will all be the same. The only thing that changes is the position of the texts (besides the string within the texts but we'll do that in a second). Instead, we can use our setUpText() function and pass in the currently working text object as well as the position. **We need a strong reference to each text object** as we pass it in, otherwise it will just copy the values into a different text object and leave the original one untouched. We can implement it like this:

```
void Texts::setUpText(sf::Text * text, sf::Vector2f position) {
  text->setFont(font);
  text->setCharacterSize(50);
  text->setFillColor(sf::Color::White);
  text->setStyle(sf::Text::Bold);
  text->setPosition(position);
}
```

Next, we can change performSetup() to this:

```
bool Texts::performSetup() {
  if (!font.loadFromFile("assets/Arial.ttf")) {
    std::cout << "Could not load font file" << std::endl;
    return false;
  }
  setUpText(&energyText, sf::Vector2f(650,800));
  setUpText(&timeText, sf::Vector2f(650,900));
  setUpText(&endGameWonText, sf::Vector2f(400,600));
  setUpText(&endGameTimeText, sf::Vector2f(400,700));
  setUpText(&endGameSpaceText, sf::Vector2f(200,800));
  return true;
}
```

That way, we can efficiently set up the 5 texts' default attributes. Finally, we want to implement the draw functions. Within them, we have to set the texts based on the inputs the function receives. We can use the **std::to_string() function** to convert floats and ints to strings and from the sf::Time object, we want the number of seconds as an integer (because we want to display the number of seconds as a whole number). We can implement all of that like this:

```
void Texts::drawInGameText(sf::RenderWindow * window, sf::Time time, int energy) {
  energyText.setString("Energy: " + std::to_string(energy));
  timeText.setString("Time: " + std::to_string((int) time.asSeconds()) + "s");
  window->draw(energyText);
  window->draw(timeText);
}

void Texts::drawEndGameText(sf::RenderWindow * window, sf::Time time) {
  endGameTimeText.setString("Time: " + std::to_string((int) time.asSeconds()) + "s");
  window->draw(endGameWonText);
  window->draw(endGameTimeText);
  window->draw(endGameSpaceText);
}
```

Note how we are converting the energy to a string as well as the int representation of the time.asSeconds(). Now we want to add a **Texts object to our GameWorld** so that we can display texts at the appropriate time. We can load everything in and then just draw the texts that we want. If we had loads of text to display, we might create separate classes for each set of Texts. In gameWorld.h, add the #include statement:

```
#include “texts.h”
```

And add the private variable:

```
Texts texts;
```

In gameWorld.cpp, add the #include statement:

```
#include “texts.cpp”
```

And add the **Texts constructor** to the GameWorld constructor:

```
GameWorld::GameWorld(): enemy(100), texts() {
  damage = 10;
}
```

Next, update the performSetup() function to set up texts as well:

```
bool GameWorld::performSetup() {
  isGameOver = false;
  enemy = Enemy(100);
  texts = Texts();
  return loadBackground() && enemy.performSetup() && texts.performSetup();
}
```

We have a different set of texts to draw when the game is over vs when it’s still running but for now, we will just draw the in-game texts by adding this line after drawing the enemy:

```
texts.drawInGameText(&window, time, enemy.energy);
```

We also need to start keeping track of the time using our time variable of GameWorld so at the top of the game loop, before the Event handling, add this line:

```
time = clock.getElapsedTime();
```

If we compile and run the code, we should see the time and score texts being displayed in the bottom right corner.

Now we can add some collision detection and event handling. We will decrease the enemy energy by damage amount when the user clicks on the enemy. Our game won't need anything very sophisticated to do this; **all we need to do is get the x and y values of the mouse click, the bounding box of the enemy sprite, and check to see if the mouse x and y are within the box**.

In our Enemy class, we have a function called checkIfHit() that takes in a sf::Vector2i object. We can use this to get the mouse x and y positions and we have the bounding box of the sprite in the Enemy class. We first get the **minimum x and y** positions from the sprite left and top boundaries. Then we get the **maximum x and y** through the x position + width and y position + height. Then we check to see if the mouse x is greater than the minimum x and less than the maximum x and the same for the y values like this:

```
bool Enemy::checkIfHit(sf::Vector2i mousePos) {
  float enemyMinX = enemySprite.getGlobalBounds().left;
  float enemyMaxX = enemySprite.getGlobalBounds().width + enemyMinX;
  float enemyMinY = enemySprite.getGlobalBounds().top;
  float enemyMaxY = enemySprite.getGlobalBounds().height + enemyMinY;

  float mouseX = mousePos.x;
  float mouseY = mousePos.y;

  return mouseX >= enemyMinX && mouseX <= enemyMaxX && mouseY >= enemyMinY && mouseY
<= enemyMaxY;
}
```

The **getGlobalBounds() function** allows us to get x, y, width, and height values and the left and top are x and y values, respectively. We return true if all conditions are met meaning that the mouse was clicked within the bounding box. We may as well implement the take damage function now too. It will subtract damage amount from the Enemy energy, play the attack sound, and return true if the energy <= 0, indicating that the Enemy is dead. We can do that like this:

```
bool Enemy::takeDamage(int damage) {
  energy -= damage;
  attackSound.play();
  return energy <= 0;
}
```

Now we want to add the mouse click detection in the runGame() function in GameWorld. We can do so by adding this code underneath the code checking for a window close event:

```
else if (event.type == sf::Event::MouseButtonPressed) {
  if (enemy.checkIfHit(sf::Mouse::getPosition(window))) {
    enemy.takeDamage(damage);
    std::cout << "Clicked on enemy" << std::endl;
  }
}
```

We first check to see if the mouse button was pressed. Then we get the position in the window and pass it into the enemy.checkIfHit() function. If that returns true, we have a hit and so we call the takeDamage() function. The print statement is unnecessary but there in case you want to run the code and test it out. If you compile and run the code, you should see the energy decreasing as you click on the enemy.

Right now, the game is almost complete. We can click on the enemy to subtract energy and play the attack sound. We can display the seconds elapsed and the current energy. However, **the game doesn't yet stop** when the energy is at 0 and we cannot restart the game. We just need to add a couple of checks to implement this functionality.

First, we want to stop the game when energy <= 0. We want to set isGameOver variable when the enemy.energy hits 0. We also only want to carry out the mouse click logic if the game is still running, otherwise we ignore mouse clicks. We can change the MouseButtonPressed logic to this:

```
else if (event.type == sf::Event::MouseButtonPressed) {
  if (!isGameOver) {
    if (enemy.checkIfHit(sf::Mouse::getPosition(window))) {
      isGameOver = enemy.takeDamage(damage);
      std::cout << "Clicked on enemy" << std::endl;
    }
  }
}
```

isGameOver is set to false as long as enemy.takeDamage() returns false. It only returns true when energy <= 0 at which point the game should end. We also want to stop keeping track of time when the game is over so we can change the clock.getElapsed time line of code to this:

```
if (!isGameOver) {
  time = clock.getElapsedTime();
}
```

That way, **we stop storing the elapsed time** in the "time" variable and can display its last value. isGameOver also affects which items we draw. **We only draw the in-game text and the enemy when the game is running**, otherwise, we draw the end game text only. We can change the draw statements to look like this:

```
window.clear();
window.draw(background);
if (isGameOver) {
  texts.drawEndGameText(&window, time);
} else {
  enemy.draw(&window);
  texts.drawInGameText(&window, time, enemy.energy);
}
window.display();
```

This way, we pick and choose which objects to display. The final thing we need to add is the reset functionality. Thanks to how we set up the **main.cpp**, the only thing we need to do is **return true from runGame() when the spacebar is pressed and isGameOver = true**. We just need to add this case to the event handling statements:

```
else if (event.type == sf::Event::KeyPressed) {
  if (isGameOver) {
    if (sf::Keyboard::isKeyPressed(sf::Keyboard::Space)) {
      return true;
    }
  }
}
```

We check to see if a key is pressed, if the key pressed is a space bar (but only as long as the game is over because we don’t want to reset mid-game), and return true if that’s the case. This will ensure that our loop in main.cpp will move onto the next loop iteration which restarts our GameWorld.

That’s it! Congratulations on finishing your first game with SFML! Feel free to run it and test out the functionalities. Although this is just a very simple game, you now have the tools to take it to the next level and add your own flair!

With our game done, that concludes the course as a whole. You are now versed in the SFML library and can begin using it to add multimedia functionality to your C++ programs. We started by learning about SFML and installing the library. Then we learned about how it runs and covered various individual components such as drawing sprites and texts, loading and playing audio, handling user interactions, and recording time. We put all of these concepts together as well as new ones such as collision detection and in-game logic to build a simple point and click game.

If you want to improve your SFML skills, I would first recommend improving upon our current game. We have something that we know works and it would be easy to add other small functionalities such as an enemy that fights back, different key ways to heal or do extra damage, implement movement, and so on. Once you've exhausted this game, you could either learn more about the SFML ecosystem by exploring other modules or even have a go at building new games and other projects from scratch. If starting brand new projects, it helps to start small or start with something that has already been done such as pacman or space invaders; that way you have a template for how things should look and work. Otherwise thanks so much for taking this course! I really hope that you enjoyed it and I can't wait to see what other cool projects you can come up with!