Chapter 1: C++, SFML, Visual Studio, and Starting the First Game

Welcome to *Beginning C++ Game Programming*. I will not waste any time in getting you started on your journey to writing great games for the PC using C++ and the OpenGL powered SFML.

This is quite a hefty first chapter, but we will learn absolutely everything we need so that we have the first part of our first game up and running. Here is what we will do in this chapter:

- Find out about the games we will build
- Meet C++
- Find out about Microsoft Visual C++
- Explore SFML and its relationship with C++
- Setting up the development environment
- Plan and prepare for the first game project, Timber!!!

Write the first C++ code of this book and make a runnable game that draws a background

The games we will build

This journey will be smooth as we will learn about the fundamentals of the super-fast C++ language one step at a time, and then put this new knowledge to use by adding cool features to the five games we are going to build.

The following are our five projects for this book.

Timber!!!

The first game is an addictive, fast-paced clone of the hugely successful Timberman, which can be found at http://store.steampowered.com/app/398710/. Our game, Timber!!!, will introduce us to all the basics of C++ while we build a genuinely playable game. Here is what our version of the game will look like when we are done and we have added a few last-minute enhancements:

Pong

Pong was one of the first video games to be made, and you can find out about its history here: https://en.wikipedia.org/wiki/Pong. It is an excellent example of how the basics of game object animation and dynamic collision detection work. We will build this simple retro game to explore

the concept of classes and object-oriented programming. The player will use the bat at the bottom of the screen and hit the ball back to the top of the screen:

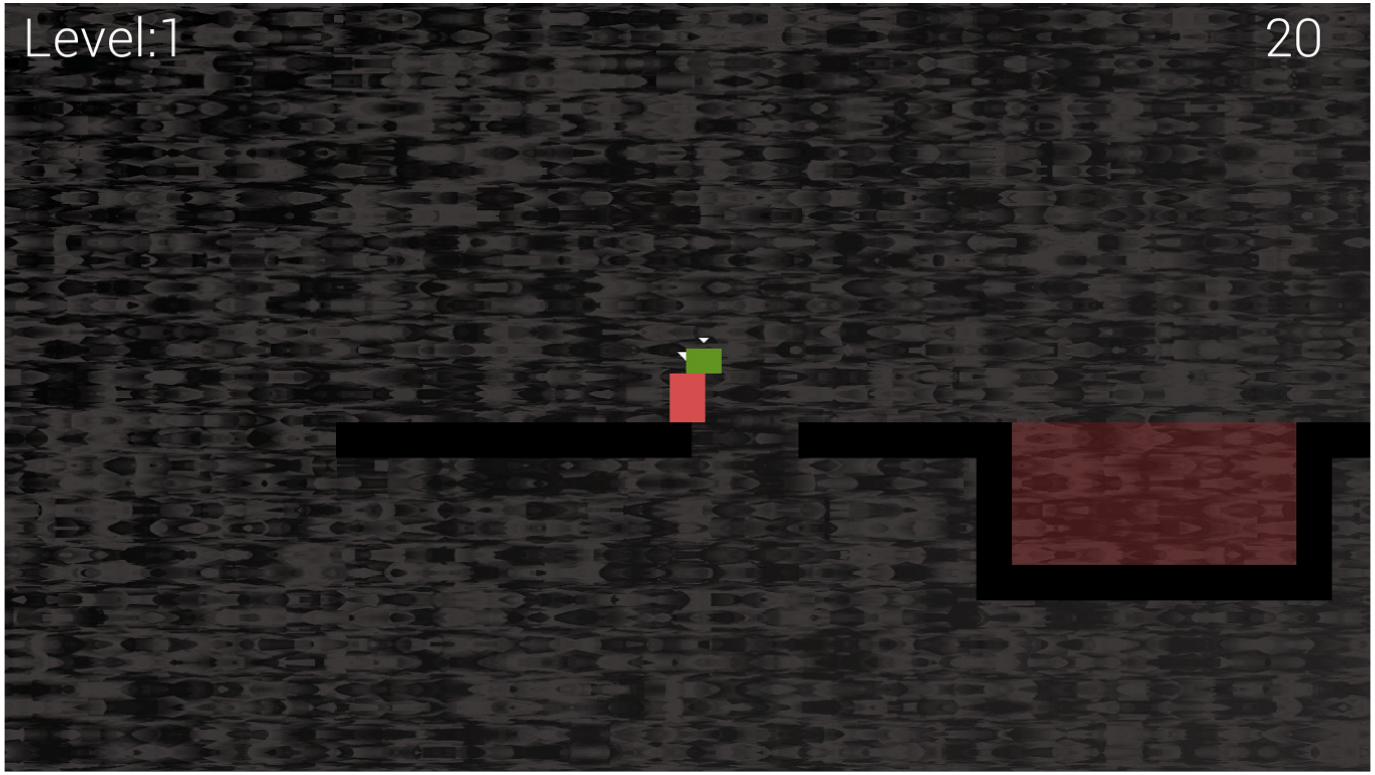
Zombie Arena

Next, we will build a frantic, zombie survival shooter, not unlike the Steam hit *Over 9,000 Zombies!*, which you can find out more about at http://store.steampowered.com/app/273500/. The player will have a machine gun and must fight off ever-growing waves of zombies. All this will take place in a randomly generated, scrolling world. To achieve this, we will learn about how object-oriented programming allows us to have a

large **code base** (lots of code) that is easy to write and maintain. Expect exciting features such as hundreds of enemies, rapid-fire weaponry, pickups, and a character that can be “leveled up” after each wave:

Thomas was late

The fourth game will be a stylish and challenging single-player and co-op puzzle platformer. It is based on the very popular game *Thomas was Alone* (http://store.steampowered.com/app/220780/). Expect to learn about cool topics such as particle effects, OpenGL Shaders, and split-screen cooperative multiplayer:



Tip

If you want to play any of the games now, you can do so from the download bundle in the `Runnable Games` folder. Just double-click on the appropriate `.exe` file. Note that, in this folder, you can run either the completed games or any game in its partially completed state from any chapter.

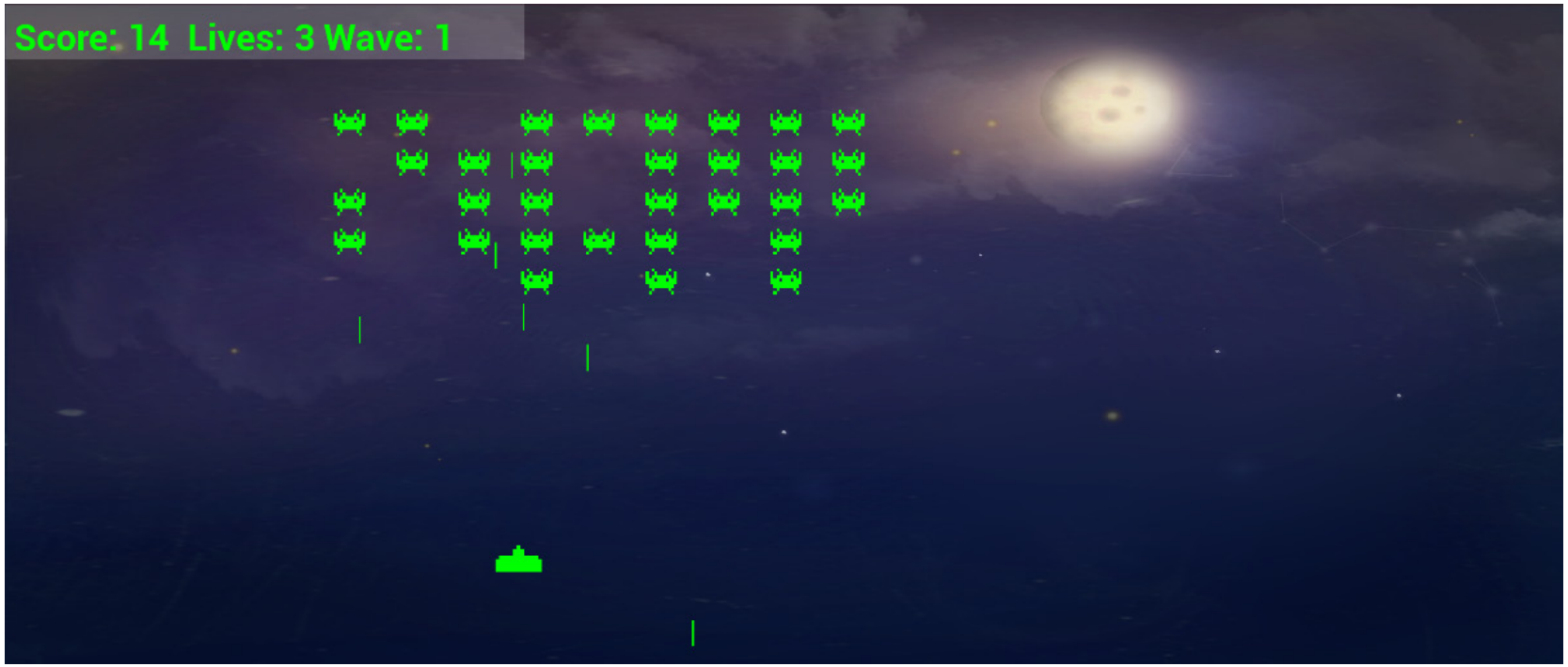
Space Invaders ++

The final game will be a Space Invaders clone. In some ways, the game itself is not what is important about this project. The project will be used to learn about game programming patterns. As will become abundantly clear as this book progresses, our code keeps getting longer and more complicated. Each project will introduce one or more techniques for coping with this, but the complexity and length of our code will keep coming back to challenge us, despite these techniques.

The Space Invaders project (called Space Invaders ++) will show us ways in which we can radically reorganize our game code also that we can take control of and properly manage our code once and for all. This will leave you with all the knowledge you need to plan and build deep, complex, and innovative games, without ending up in a tangle of code.

The game will also introduce concepts such as screens, input handlers, and entity-component systems. It will also allow us to learn how to let the player use a gamepad instead of the keyboard and introduce the C++

concepts of smart pointers, casts, assertions, breakpoint debugging, and teach us the most important lesson from the whole book: how to build your own unique games:



Let’s get started by introducing C++, Visual Studio, and SFML!

Meet C++

Now that we know what games we will be building, let's get started by introducing C++, Visual Studio, and SFML. One question you might have is, *why use the C++ language at all?* C++ is fast – very fast. What makes this true is the fact that the code that we write is directly translated into machine-executable instructions. These instructions are what make the game. The executable game is contained within a `.exe` file, which the player can simply double-click to run.

There are a few steps in the process of changing our code into an executable file. First, the **preprocessor** looks to see if any *other code* needs to be included within our own code and adds it. Next, all the code is **compiled** into **object files** by the **compiler** program. Finally, a third program, called the **linker**, joins all the object files into the executable file for our game.

In addition, C++ is well established at the same time as being extremely up to date. C++ is an **object-oriented programming** (**OOP**) language, which means we can write and organize our code using well-tested conventions that make our games efficient and manageable. The benefits as well as the necessity of this will reveal themselves as we progress through this book.

Most of this *other code* that I referred to, as you might be able to guess, is SFML, and we will find out more about SFML in just a minute. The preprocessor, compiler, and linker programs I have just mentioned are all part of the Visual Studio **integrated development environment** (**IDE**).

Microsoft Visual Studio

Visual Studio hides away the complexity of preprocessing, compiling, and linking. It wraps it all up into the press of a button. In addition to this, it provides a slick user interface for us to type our code into and manage what will become a large selection of code files and other project assets as well.

While there are advanced versions of Visual Studio that cost hundreds of dollars, we will be able to build all five of our games in the free "**Express 2019 for Community**" version. This is the latest free version of Visual Studio.

SFML

SFML is the **Simple Fast Media Library**. It is not the only C++ library for games and multimedia. It is possible to make an argument to use other libraries, but SFML seems to come through for me every time. Firstly, it is written using object-oriented C++. The benefits of object-oriented C++ are numerous, and you will experience them as you progress through this book.

SFML is also easy to get started with and is therefore a good choice if you are a beginner, yet at the same time it has the potential to build the highest-quality 2D games if you are a professional. So, a beginner can get started using SFML and not worry about having to start again with a new language/library as their experience grows.

Perhaps the biggest benefit is that most modern C++ programming uses OOP. Every C++ beginner's guide I have ever read uses and teaches OOP. OOP is the future (and the now) of coding in almost all languages, in fact. So why, if you're learning C++ from the beginning, would you want to do it any other way?

SFML has a module (code) for just about anything you would ever want to do in a 2D game. SFML works using OpenGL, which can also make 3D games. OpenGL is the de facto free-to-use graphics library for games when you want it to run on more than one platform. When you use SFML, you are automatically using OpenGL.

SFML allows you to create the following:

- 2D graphics and animations, including scrolling game worlds.
- Sound effects and music playback, including high-quality directional sound.
- Input handling with a keyboard, mouse, and gamepad.
- Online multiplayer features.
- The same code can be compiled and linked on all major desktop operating systems, and mobile as well!.

Extensive research has not uncovered any more suitable ways to build 2D games for PC, even for expert developers and especially if you are a beginner and want to learn C++ in a fun gaming environment.

In the sections that follow, we will set up the development environment, beginning with a discussion on what to do if you are using Mac or Linux operating systems.

Setting up the development environment

Now that you know a bit more about how we will be making games, it is time to set up a development environment so we can get coding.

What about Mac and Linux?

The games that we will be making can be built to run on Windows, Mac, and Linux! The code we use will be identical for each platform. However, each version does need to be compiled and linked on the platform for which it is intended, and Visual Studio will not be able to help us with Mac and Linux.

It would be unfair to say, especially for complete beginners, that this book is entirely suited for Mac and Linux users. Although, I guess, if you are an enthusiastic Mac or Linux user and you are comfortable with your operating system, you will likely succeed. Most of the extra challenges you will encounter will be in the initial setup of the development environment, SFML, and the first project.

To this end, I can highly recommend the following tutorials, which will hopefully replace the next 10 pages (approximately), up to the *Planning Timber!!!* section, when this book will become relevant to all operating systems.

For Linux, read this to replace the next few sections: https://www.sfml-dev.org/tutorials/2.5/start-linux.php.

On Mac, read this tutorial to get started: https://www.sfml-dev.org/tutorials/2.5/start-osx.php.

Installing Visual Studio 2019 Community edition

To start creating a game, we need to install Visual Studio 2019. Installing Visual Studio can be almost as simple as downloading a file and clicking a few buttons. I will walk you through the installation process a step at a time.

Important note

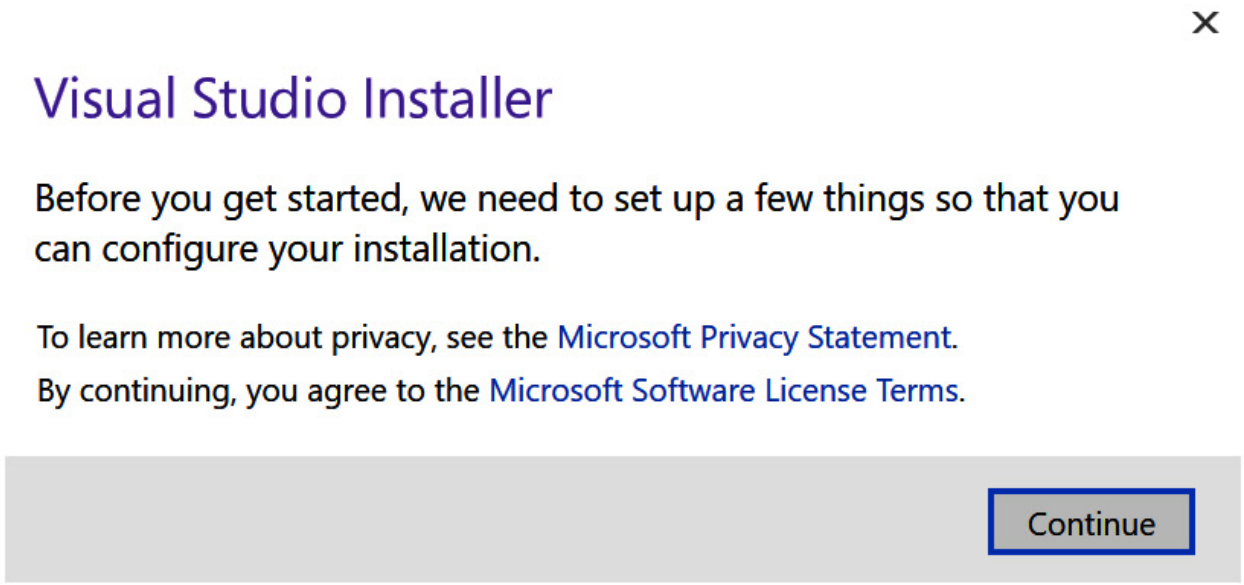
Note that, over the years, Microsoft is likely to change the name, appearance, and download page that's used to obtain Visual Studio. They are likely to change the layout of the user interface and make the instructions that follow out of date. However, the settings that we configure for each project are fundamental to C++ and SFML, so careful interpretation of the instructions that follow in this chapter will likely be possible, even if Microsoft does something radical to Visual Studio. Anyway, at the time of writing, Visual Studio 2019 has been released for

just two weeks, so hopefully this chapter will be up to date for a while. If something significant does happen, then I will add an up-to-date tutorial on http://gamecodeschool.com as soon as I find out about it.

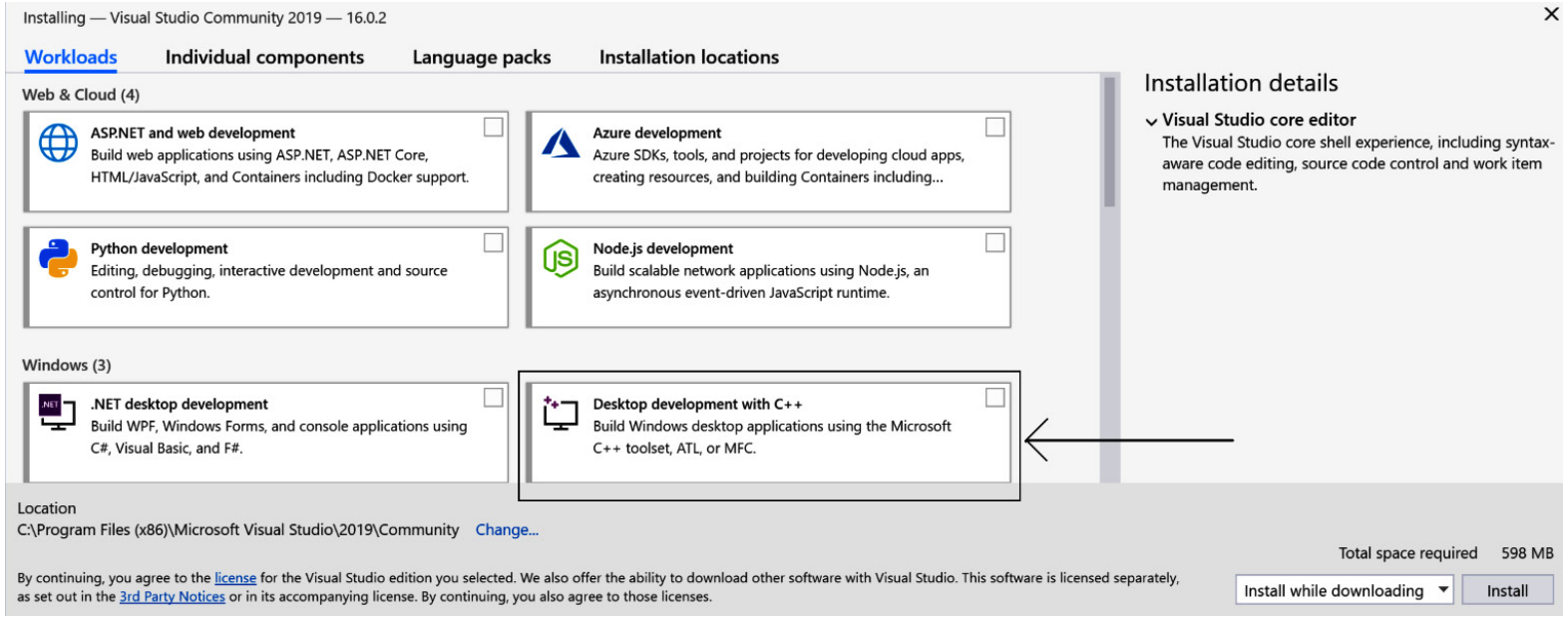
Let's get started with installing Visual Studio:

1. The first thing you need is a Microsoft account and login details. If you have a Hotmail or MSN email address, then you already have one. If not, you can sign up for a free one here: https://login.live.com/.
2. The next step is to visit https://visualstudio.microsoft.com/vs/ and find the download link for **Community 2019**. This is what it looks like at the time of writing:

3. Save the file to your computer.
4. When the download completes, run the download by double-clicking on it. My file, at the time of writing, was called `vs_community__33910147.1551368984.exe`. Yours will be different based on the current version of Visual Studio.
5. After giving permission for Visual Studio to make changes to your computer, you will be greeted with the following window. Click **Continue**:



6. Wait for the installer program to download some files and set up the next stage of the installation. Shortly, you will be presented with the following window:



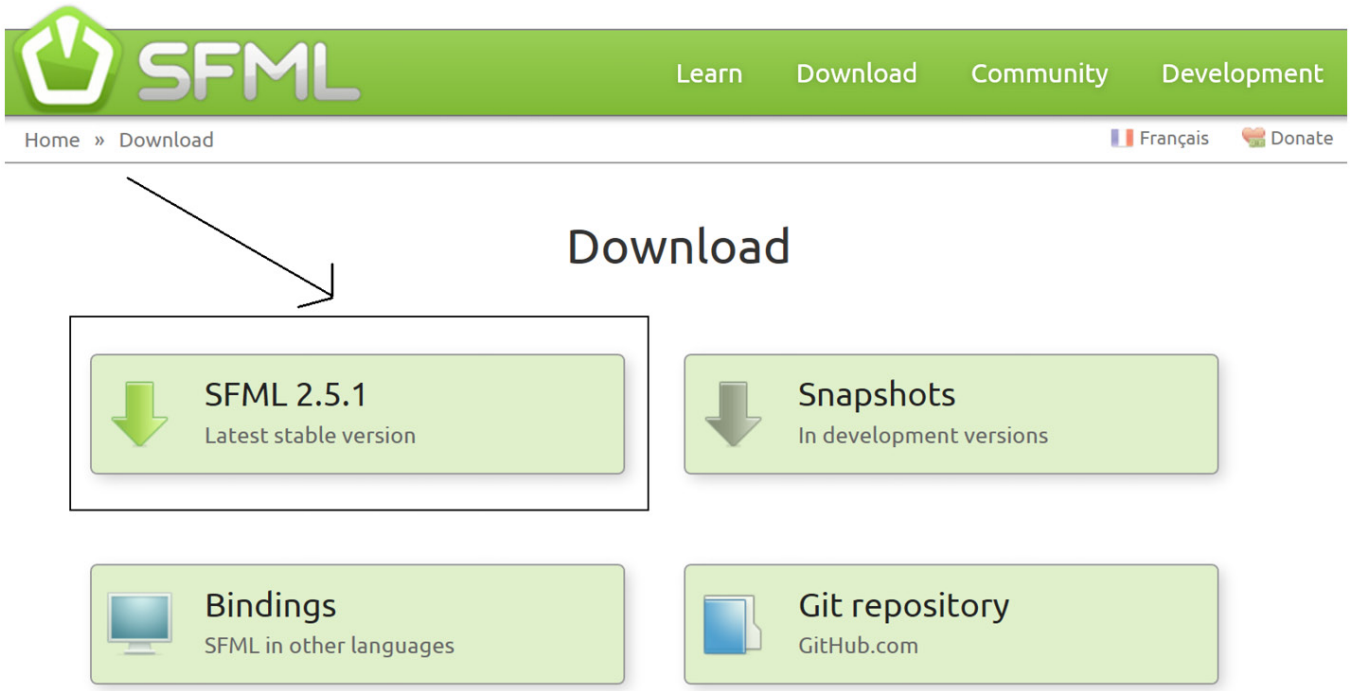
7. If you want to choose a new location to install Visual Studio, locate the **Change** option and configure the install location. The simplest thing to do is leave the file at the default location chosen by Visual Studio. When you are ready, locate the **Desktop development with C++** option and select it.
8. Next, click the **Install** button. Grab some refreshments as this step might take a while.
9. When the process completes, you can close all open windows, including any that prompt you to start a new project, as we are not ready to start coding until we have installed SFML.

Now, we are ready to turn our attention to SFML.

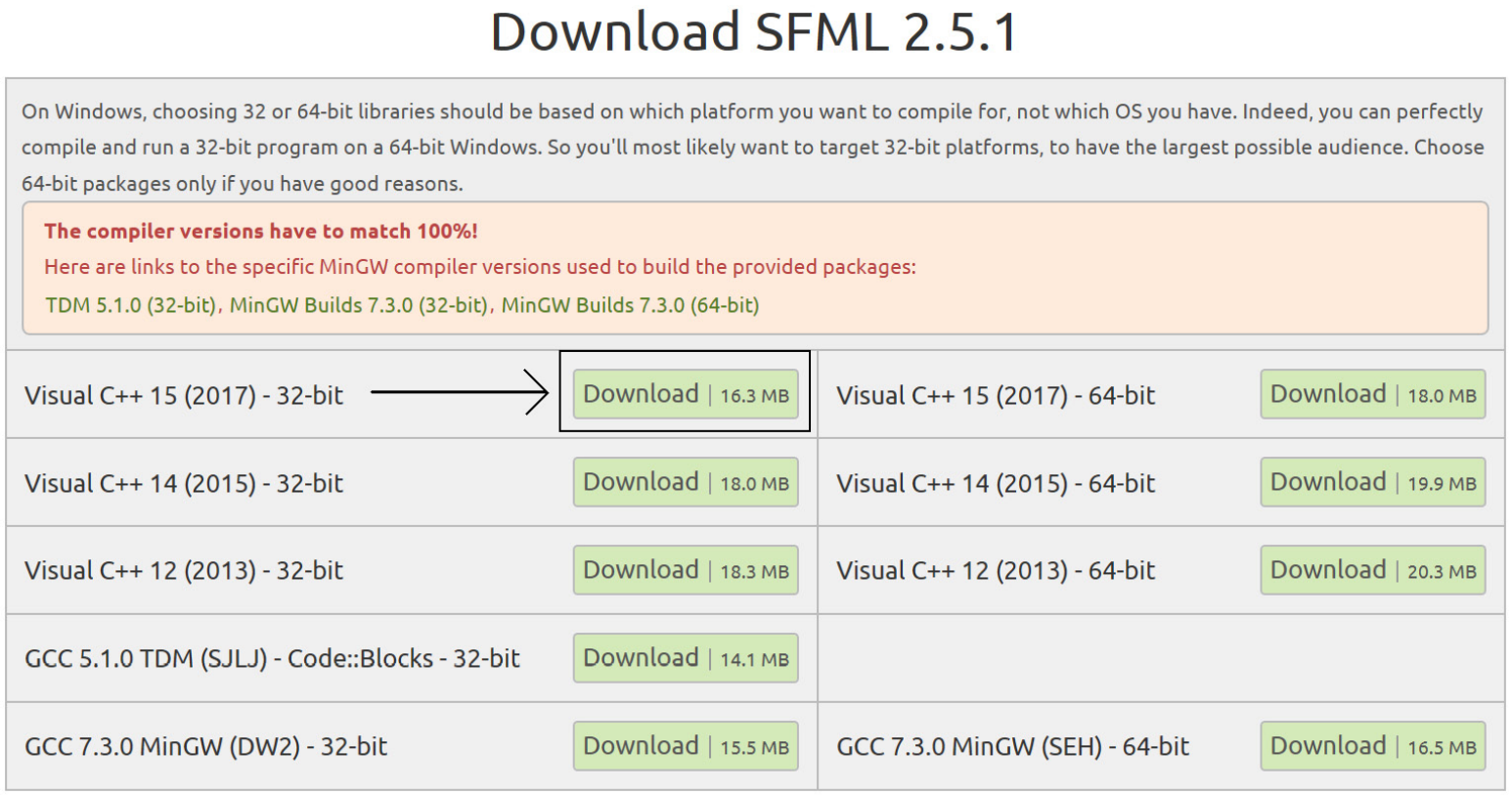
Setting up SFML

This short tutorial will guide you through downloading the SFML files that allow us to include the functionality contained in the library in our projects. In addition, we will see how we can use the SFML **DLL** files that will enable our compiled object code to run alongside SFML. To set up SFML, follow these steps:

1. Visit this link on the SFML website: http://www.sfml-dev.org/download.php. Click on the button that says **Latest stable version**, as shown here:



2. By the time you read this book, the latest version will almost certainly have changed. This won't matter as long as you do the next step just right. We want to download the **32-bit version** of **Visual C++ 2017**. This might sound counter-intuitive because we have just installed Visual Studio 2019 and you probably (most commonly) have a 64-bit PC. The reason we chose to download the 32-bit version is that Visual C++ 2017 is part of Visual Studio 2019 (Visual Studio does more than C++) and we will be building games in 32-bit so that they can run on *both* 32- and 64-bit machines. Click the **Download** button that's shown in the following screenshot:



3. When the download completes, create a folder at the root of the same drive where you installed Visual Studio and name it `SFML`. Also, create another folder at the root of the drive where you installed Visual Studio and call it `VS Projects`.
4. Finally, unzip the SFML download. Do this on your desktop. When unzipping is complete, you can delete the .zip folder. You will be left with a single folder on your desktop. Its name will reflect the version of SFML that you downloaded. Mine is called `SFML-2.5.1-windows-vc15-32-bit`. Your filename will likely reflect a more recent version. Double-click this folder to see its contents, then double-click again into the next folder (mine is called `SFML-2.5.1`). The following

screenshot shows what my `SFML-2.5.1` folder's content looks like. Yours should look the same:

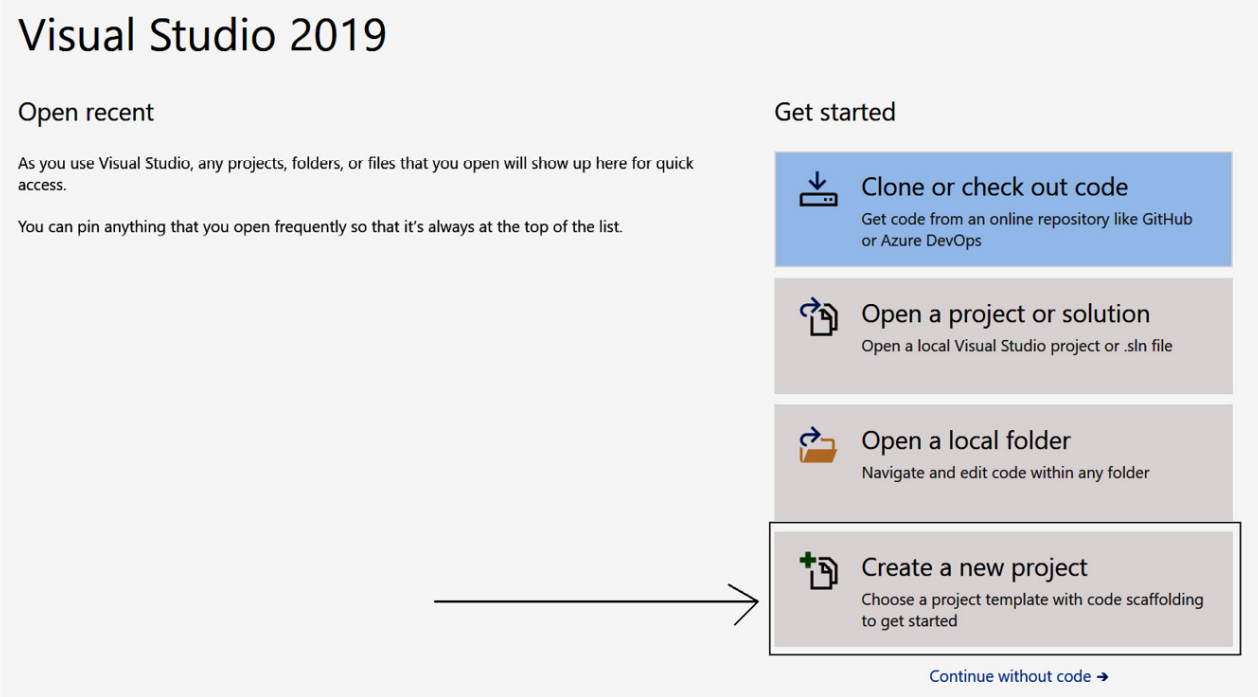
5. Copy the entire contents of this folder and paste all the files and folders into the `SFML` folder that you created in *Step 3*. For the rest of this book, I will refer to this folder simply as "your `SFML` folder".

Now, we are ready to start using C++ and SFML in Visual Studio.

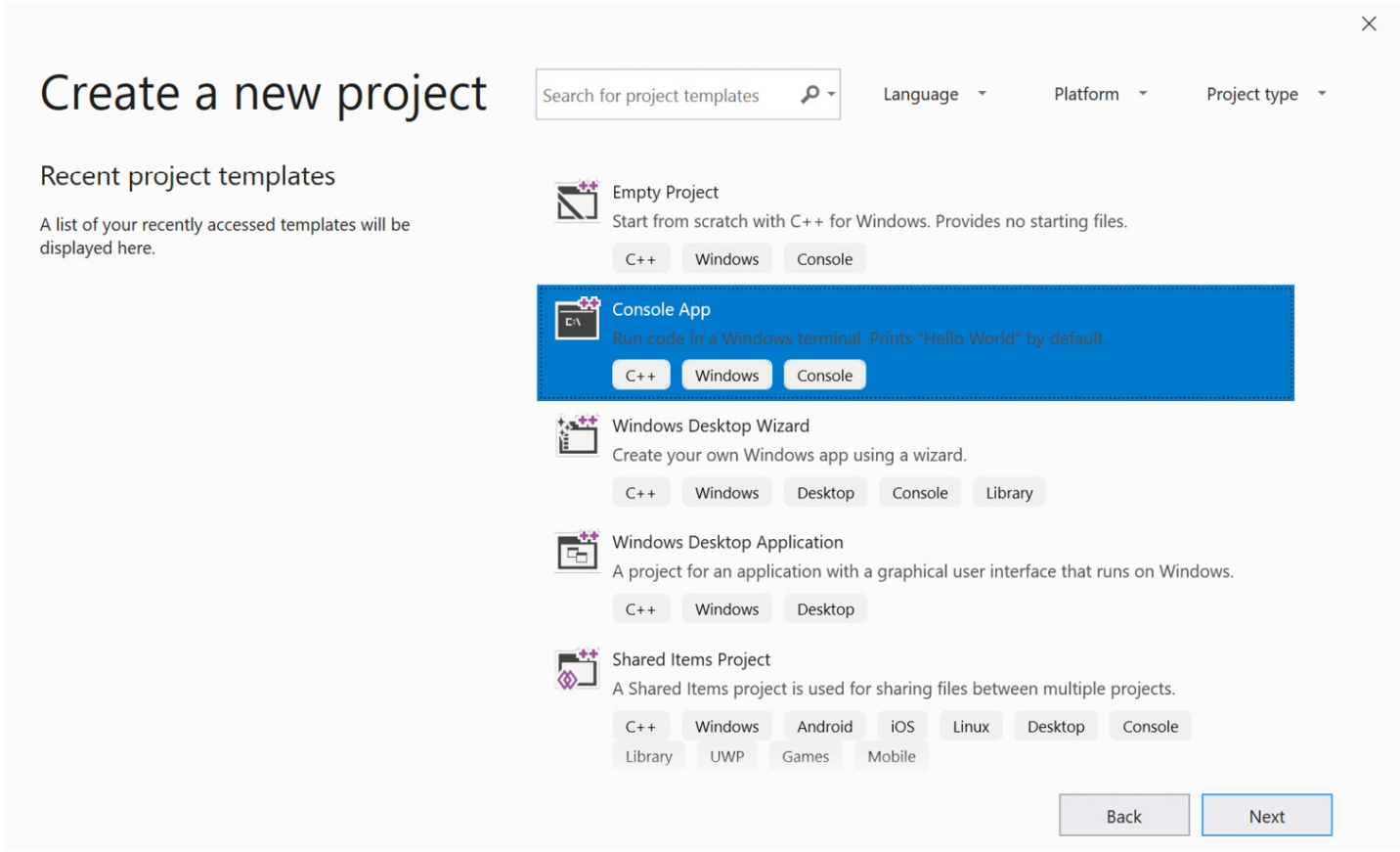
Creating a new project

As setting up a project is a fiddly process, we will go through it step by step so that we can start getting used to it:

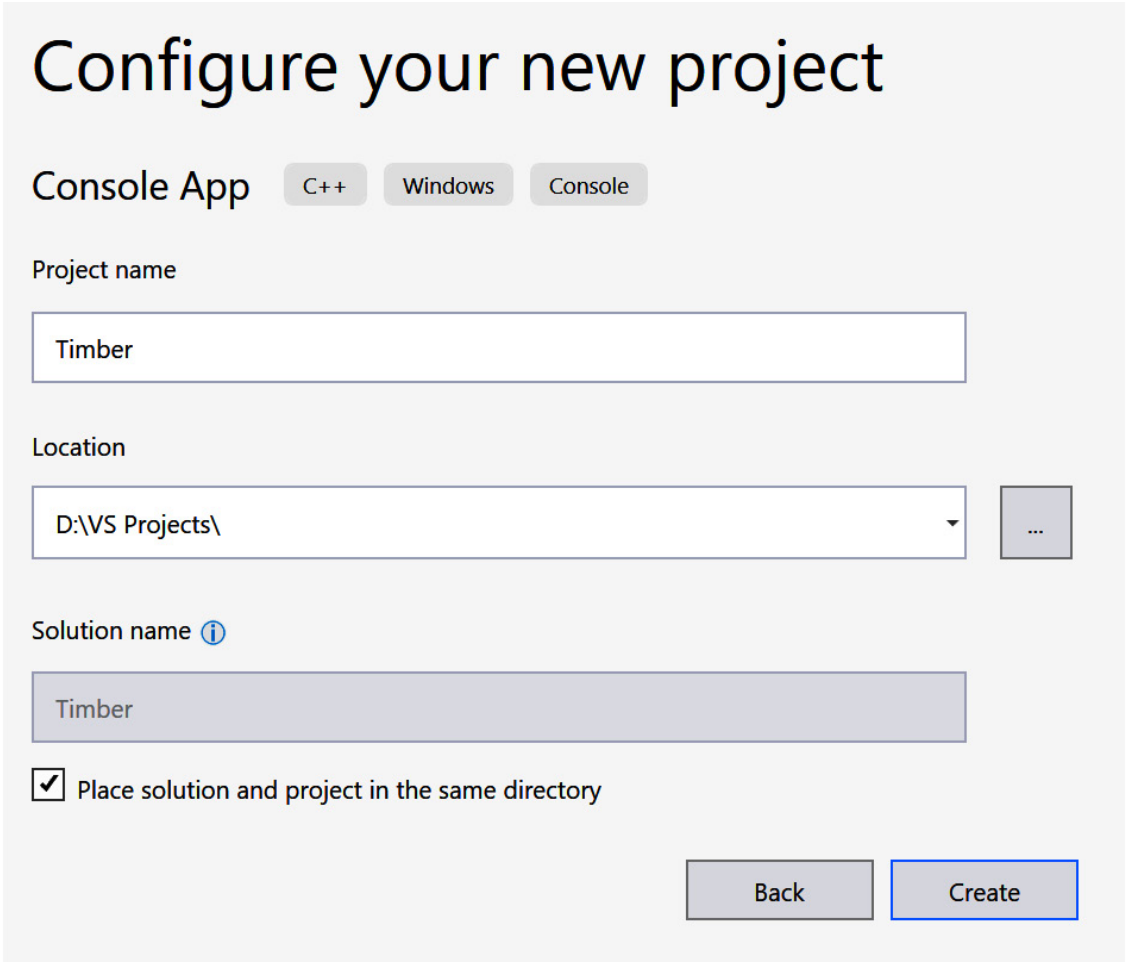
1. Start Visual Studio in the same way you start any app: by clicking on its icon. The default installation options will have placed a **Visual Studio 2019** icon in the Windows start menu. You will see the following window:



2. Click on the **Create a new project** button, as highlighted in the preceding screenshot. You will see the **Create a new project** window, as shown in the following screenshot:

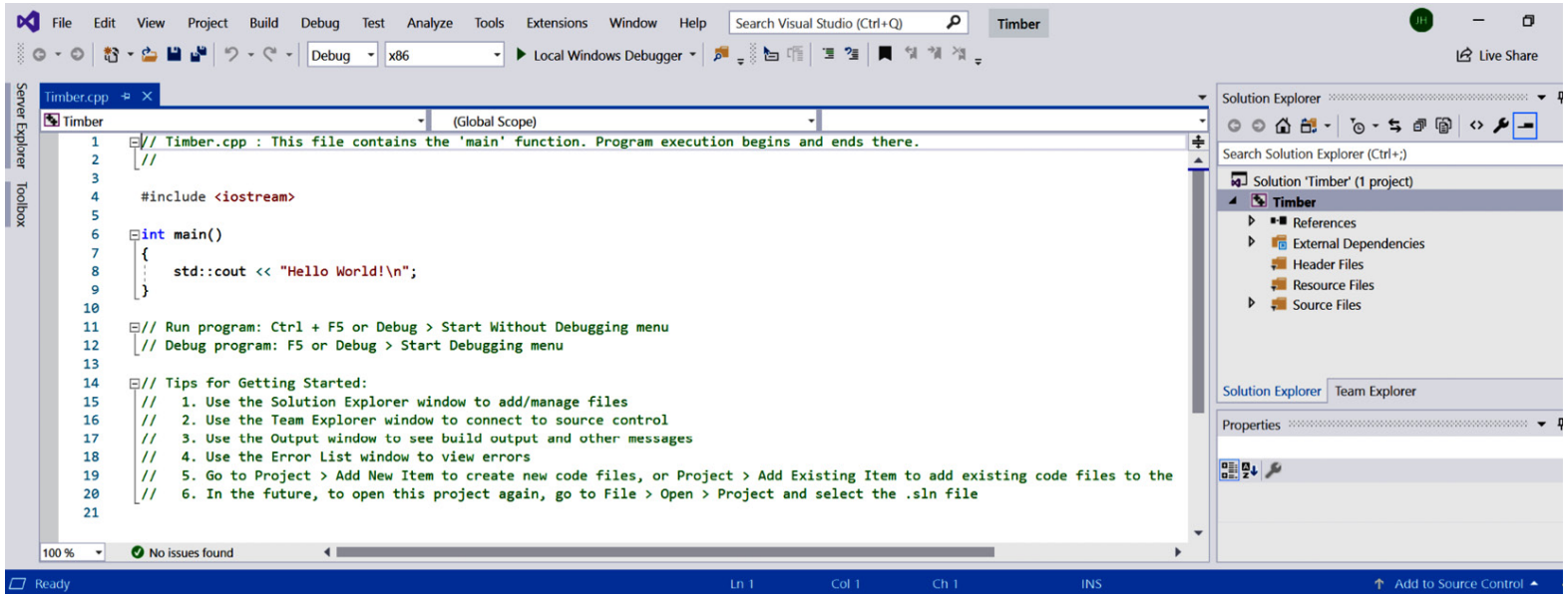


3. In the **Create a new project** window, we need to choose the type of project we will be creating. We will be creating a console app, so select **Console App**, as highlighted in the preceding screenshot, and click the **Next** button. You will then see the **Configure your new project** window. This following screenshot shows the **Configure your new project** window after the next three steps have been completed:

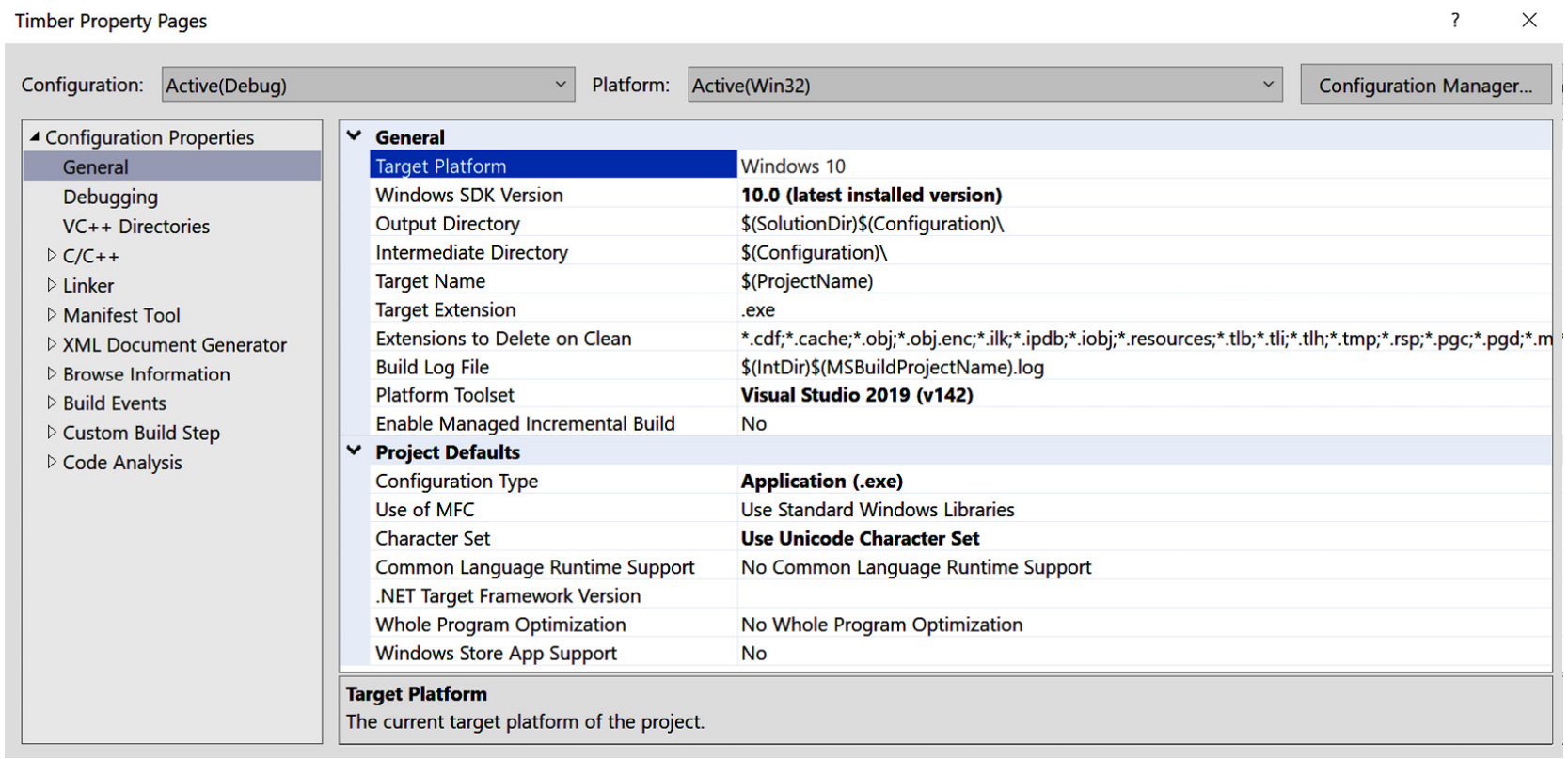


4. In the **Configure your new project** window, type `Timber` in the **Project name** field. Note that this causes Visual Studio to automatically configure the **Solution name** field to the same name.
5. In the **Location** field, browse to the `VS Projects` folder that we created in the previous tutorial. This will be the location that all our project files will be kept.
6. Check the option to **Place solution and project in the same directory**.

7. Note that the preceding screenshot shows what the window looks like when the previous three steps have been completed. When you have completed these steps, click **Create**. The project will be generated, including some C++ code. This following screenshot shows where we will be working throughout this book:



8. We will now configure the project to use the SFML files that we put in the `SFML` folder. From the main menu, select **Project** | **Timber properties...**. You will see the following window:



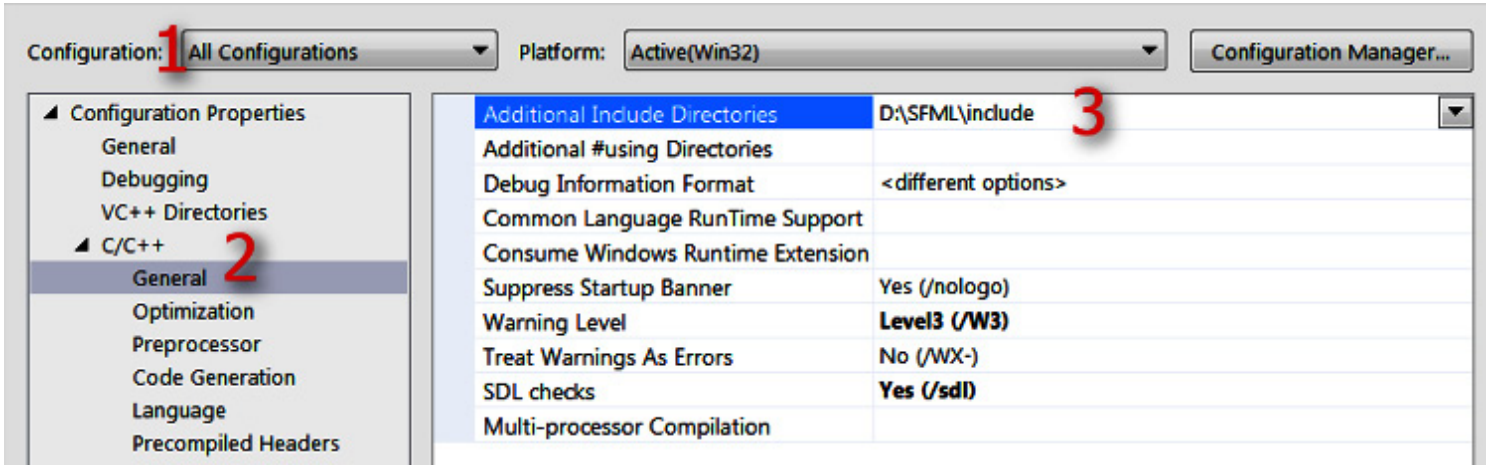
Tip

In the preceding screenshot, the **OK**, **Cancel**, and **Apply** buttons are not fully formed. This is likely a glitch with Visual Studio not handling my screen resolution correctly. Yours will hopefully be fully formed. Whether your buttons appear like mine do or not, continuing with the tutorial will be the same.

Next, we will begin to configure the project properties. As these steps are quite intricate, I will cover them in a new list of steps.

Configuring the project properties

At this stage, you should have the **Timber Property Pages** window open, as shown in the preceding screenshot at the end of the previous section. Now, we will begin to configure some properties while using the following annotated screenshot for guidance:



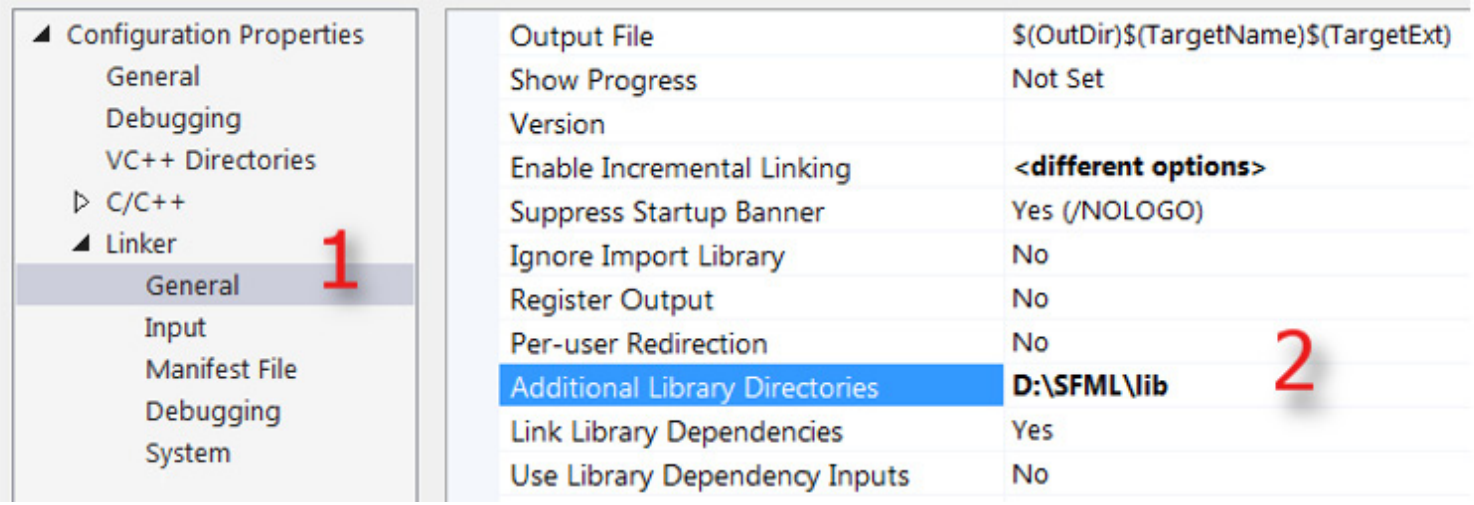
We will add some fairly intricate and important project settings in this section. This is the laborious part, but we will only need to do this once per project. What we need to do is tell Visual Studio where to find a

special type of code file from SFML. The special type of file I am referring to is a **header file**. Header files are the files that define the format of the SFML code so that when we use the SFML code, the compiler knows how to handle it. Note that the header files are distinct from the main source code files and they are contained in files with the `.hpp` file extension. All this will become clearer when we eventually start adding our own header files in the second project. In addition, we need to tell Visual Studio where it can find the SFML library files. In the **Timber Property Pages** window, perform the following three steps, which are numbered in the preceding screenshot:

1. First (**1**), select **All Configurations** from the **Configuration:** drop down.
2. Second (**2**), select **C/C++** then **General** from the left-hand menu.
3. Third (**3**), locate the **Additional Include Directories** edit box and type the drive letter where your SFML folder is located, followed by `\SFML\include`. The full path to type, if you located your `SFML` folder on your D drive, is as shown in the preceding screenshot; that is, `D:\SFML\include`. Vary your path if you installed SFML on a different drive.
4. Click **Apply** to save your configurations so far.
5. Now, still in the same window, perform these steps, which refer to the following annotated screenshot. First (**1**), select **Linker** and

then **General**.

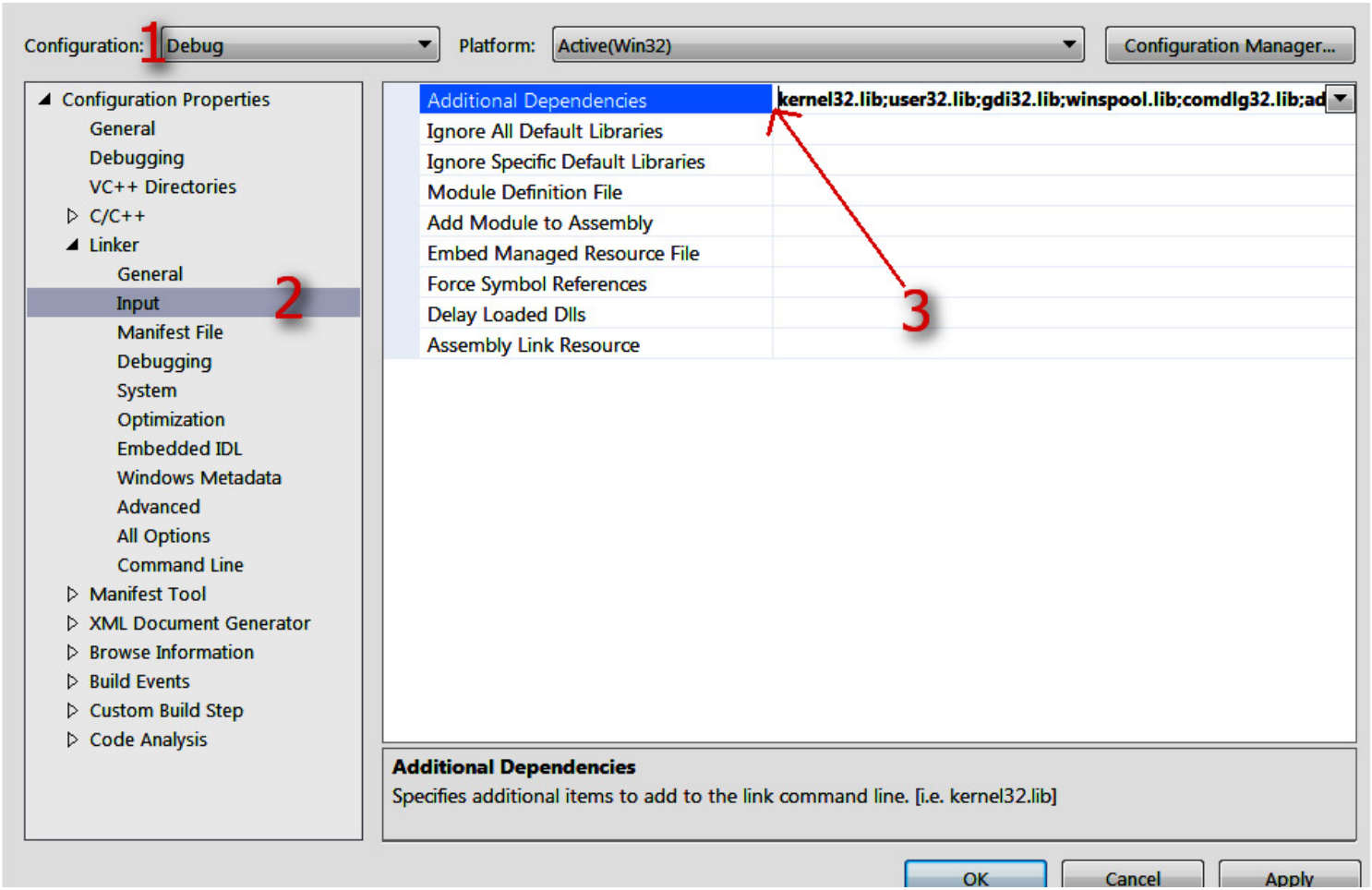
6. Now, find the **Additional Library Directories** edit box (**2**) and type the drive letter where your `SFML` folder is, followed by `\SFML\lib`. So, the full path to type if you located your `SFML` folder on your D drive is, as shown in the following screenshot, `D:\SFML\lib`. Vary your path if you installed SFML to a different drive:



7. Click **Apply** to save your configurations so far.
8. Finally, for this stage, still in the same window, perform these steps, which refer to the following annotated screenshot. Switch the **Configuration:** drop down (**1**) to **Debug** as we will be running and testing our games in debug mode.
9. Select **Linker** and then **Input** (**2**).
0. Find the **Additional Dependencies** edit box (**3**) and click into it at the far-left-hand side. Now, copy and paste/type the following: `sfml-graphics-d.lib;sfml-window-d.lib;sfml-system-d.lib;sfml-network-d.lib;sfml-audio-d.lib;` at the indicated

place. Be extra careful to place the cursor exactly in the right place and not to overwrite any of the text that is already there.

1. Click **OK**:



2. Click **Apply** and then **OK**.

Phew; that's it! We have successfully configured Visual Studio and can move on to planning the Timber!!! project.

Planning Timber!!!

Whenever you make a game, it is always best to start with a pencil and paper. If you don't know exactly how your game is going to work on the screen, how can you possibly make it work in code?

Tip

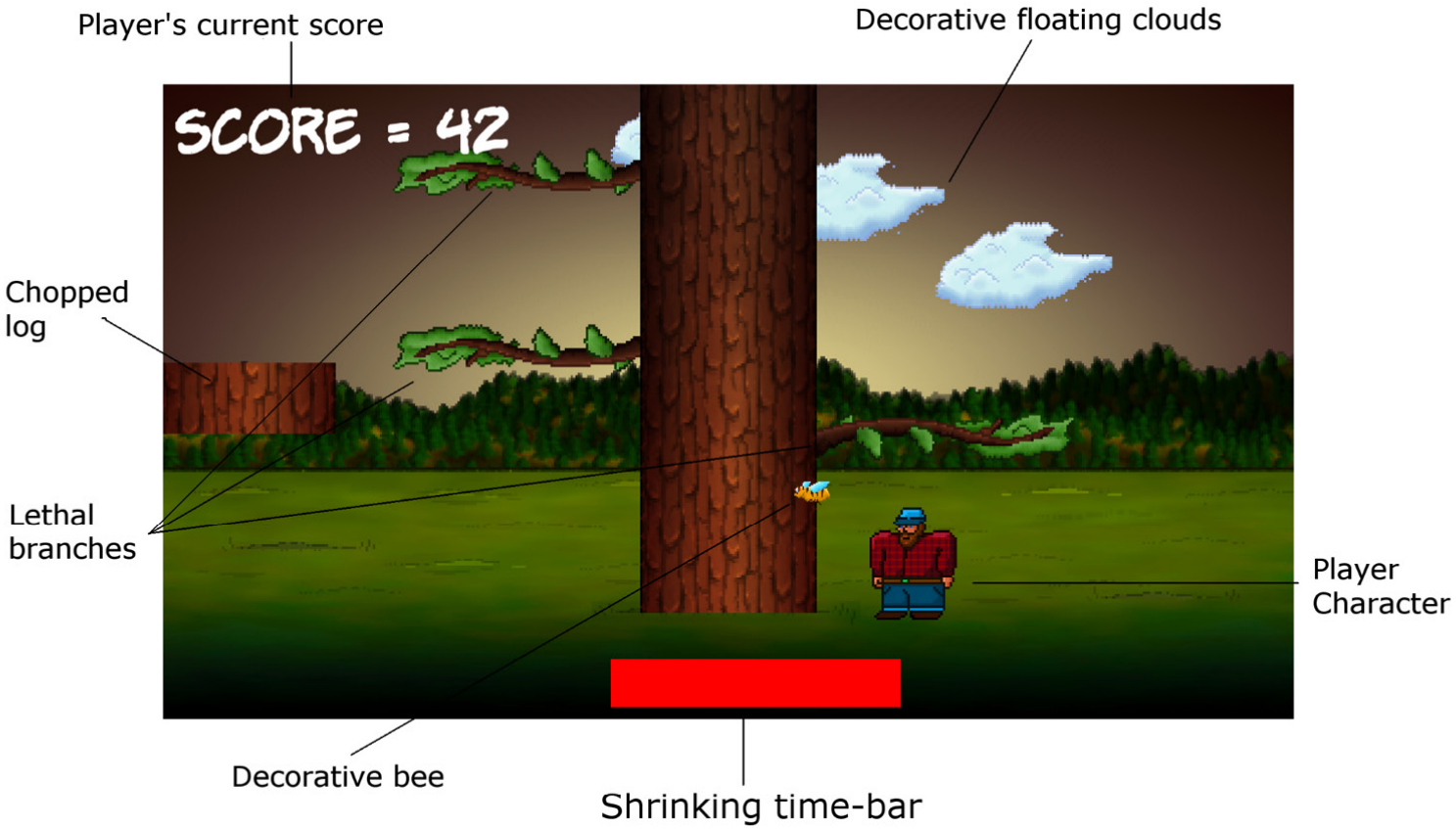
At this point, if you haven't already, I suggest you go and watch a video of Timberman in action so that you can see what we are aiming for. If you feel your budget can stretch to it, then grab a copy and give it a play. It is often on sale for under $1 on
Steam: http://store.steampowered.com/app/398710/.

The features and objects of a game that define the gameplay are known as the **mechanics**. The basic mechanics of the game are as follows:

Time is always running out.
You can get more time by chopping the tree.
Chopping the tree causes the branches to fall.
The player must avoid the falling branches.
Repeat until time runs out or the player is squished.

Expecting you to plan the C++ code at this stage is obviously a bit silly. This is, of course, the first chapter of a C++ beginner's guide. We can, however, take a look at all the assets we will use and an overview of what we will need to make our C++ code do.

Take a look at this annotated screenshot of the game:



You can see that we have the following features:

The player's score: Each time the player chops a log, they will get one point. They can chop a log with either the left or the right arrow (cursor) key.

Player character: Each time the player chops, they will move to/stay on the same side of the tree relative to the cursor key they use. Therefore, the player must be careful which side they choose to chop on.

When the player chops, a simple axe graphic will appear in the player character's hands.

Shrinking time-bar: Each time the player chops, a small amount of time will be added to the ever-shrinking time-bar.

The lethal branches: The faster the player chops, the more time they will get, but also the faster the branches will move down the tree and therefore the more likely they are to get squished. The branches spawn randomly at the top of the tree and move down with each chop.

When the player gets squished – and they will get squished quite regularly – a gravestone graphic will appear.

The chopped log: When the player chops, a chopped log graphic will whiz off, away from the player.

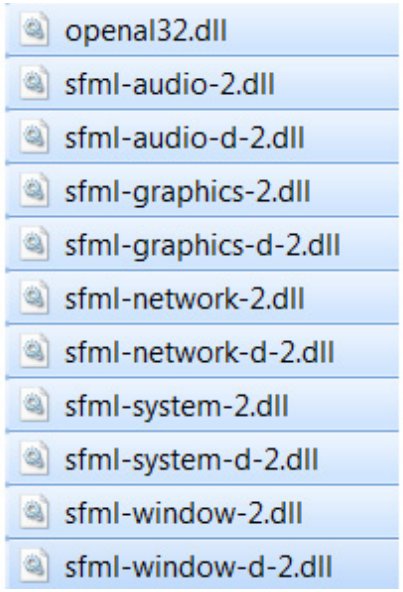
Just for decoration: There are three floating clouds that will drift at random heights and speeds, as well as a bee that does nothing but fly around.

The background: All this takes place on a pretty background.

So, in a nutshell, the player must frantically chop to gain points and avoid running out of time. As a slightly perverse, but fun consequence, the faster they chop, the more likely their squishy demise is.

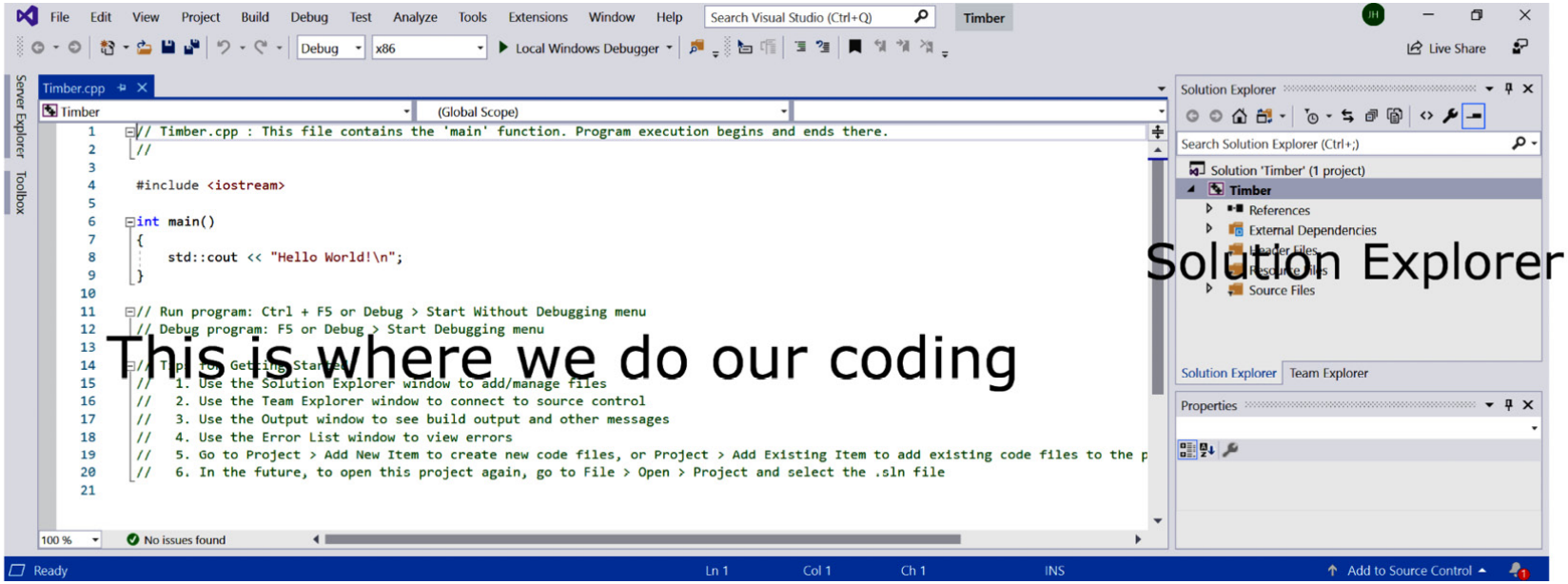
We now know what the game looks like, how it is played, and the motivation behind the game mechanics. Now, we can go ahead and start building it. Follow these steps:

1. Now, we need to copy the SFML `.dll` files into the main project directory. My main project directory is `D:\VS Projects\Timber`. It was created by Visual Studio in the previous tutorial. If you put your `VS Projects` folder somewhere else, then perform this step there instead. The files we need to copy into the project folder are located in your `SFML\bin` folder. Open a window for each of the two locations and highlight all the files in the `SFML\bin` folder, as shown in the following screenshot:



2. Now, copy and paste the highlighted files into the project folder, that is, `D:\VS Projects\Timber`.

The project is now set up and ready to go. You will be able to see the following screen. I have annotated this screenshot so you can start familiarizing yourself with Visual Studio. We will revisit all these areas, and others, soon:



Your layout might look slightly different to what's shown in the preceding screenshot because the windows of Visual Studio, like most applications, are customizable. Take the time to locate the **Solution Explorer** window on the right and adjust it to make its content nice and clear, like it is in the previous screenshot.

We will be back here soon to start coding. But first, we will explore the project assets we will be using.

The project assets

Assets are anything you need to make your game. In our case, these assets include the following:

- A font for the writing on the screen
- Sound effects for different actions, such as chopping, dying, and running out of time
- Graphics for the character, background, branches, and other game objects

All the graphics and sounds that are required for this game are included in the download bundle for this book. They can be found in the `Chapter 1/graphics` and `Chapter 1/sound` folders as appropriate.

The font that is required has not been supplied. This is because I wanted to avoid any possible ambiguity regarding the license. This will not cause a problem, though, as I will show you exactly where and how to choose and download fonts for yourself.

Although I will provide either the assets themselves or information on where to get them, you might like to create or acquire them for yourself.

Outsourcing the assets

There are a number of websites that allow you to contract artists, sound engineers, and even programmers. One of the biggest is Upwork (www.upwork.com). You can join this site for free and post your jobs. You will need to write a clear explanation of your requirements, as well as state how much you are prepared to pay. Then, you will probably get a good selection of contractors bidding to do the work. Be aware, however, that there are a lot of unqualified contractors whose work might be disappointing, but if you choose carefully, you will likely find a competent, enthusiastic, and great-value person or company to do the job.

Making your own sound FX

Sound effects can be downloaded for free from sites such as Freesound (www.freesound.org), but often the licence won't allow you to use them if you are selling your game. Another option is to use an open source software called BFXR from www.bfxr.net, which can help you generate lots of different sound effects that are yours to keep and do with as you like.

Adding the assets to the project

Once you have decided which assets you will use, it is time to add them to the project. The following instructions will assume you are using all the assets that are supplied in this book's download bundle. Where you are using your own, simply replace the appropriate sound or graphic file with your own, using exactly the same filename:

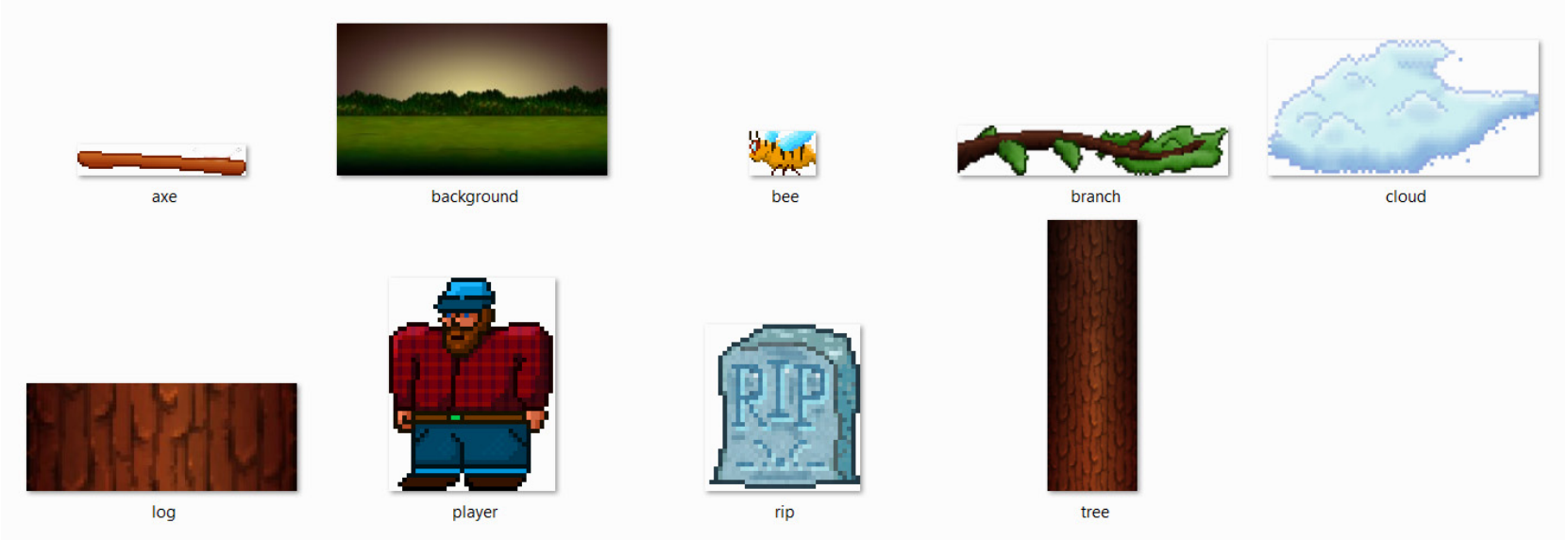
1. Browse to the project folder, that is, `D:\VS Projects\Timber`.
2. Create three new folders within this folder and name them `graphics`, `sound`, and `fonts`.
3. From the download bundle, copy the entire contents of `Chapter 1/graphics` into the `D:\VS Projects\Timber\graphics` folder.
4. From the download bundle, copy the entire contents of `Chapter 1/sound` into the `D:\VS Projects\Timber\sound` folder.

5. Now, visit http://www.1001freefonts.com/komika_poster.font in your web browser and download the **Komika Poster** font.
6. Extract the contents of the zipped download and add the `KOMIKAP_.ttf` file to the `D:\VS Projects\Timber\fonts` folder.

Let's take a look at these assets – especially the graphics – so that we can visualize what is happening when we use them in our C++ code.

Exploring the assets

The graphical assets make up the parts of the scene that is our Timber!!! game. If you take a look at the graphical assets, it should be clear where in our game they will be used:



The sound files are all in `.wav` format. These files contain the sound effects that we will play at certain events throughout the game. They were all generated using BFXR and are as follows:

`chop.wav`: A sound that is a bit like an axe (a retro axe) chopping a tree

`death.wav`: A sound a bit like a retro “losing” sound

`out_of_time.wav`: A sound that plays when the player loses by running out of time, as opposed to being squashed

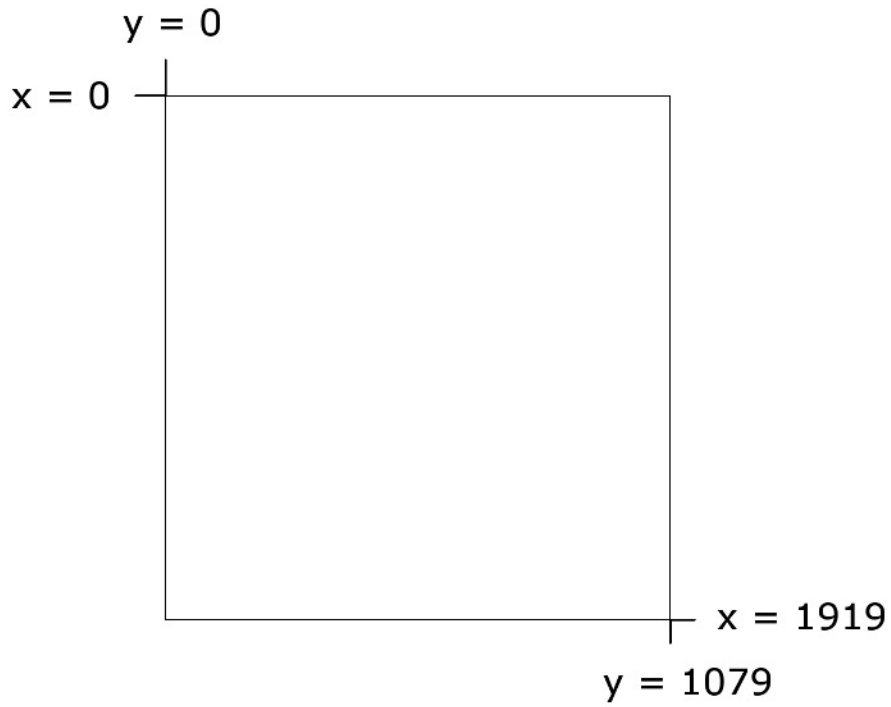
We have seen all the assets, including the graphics, so now we will have a short discussion related to the resolution of the screen and how we position the graphics on it.

Understanding screen and internal coordinates

Before we move on to the actual C++ coding, let's talk a little about coordinates. All the images that we see on our monitors are made out of pixels. Pixels are little tiny dots of light that combine to make the images we see.

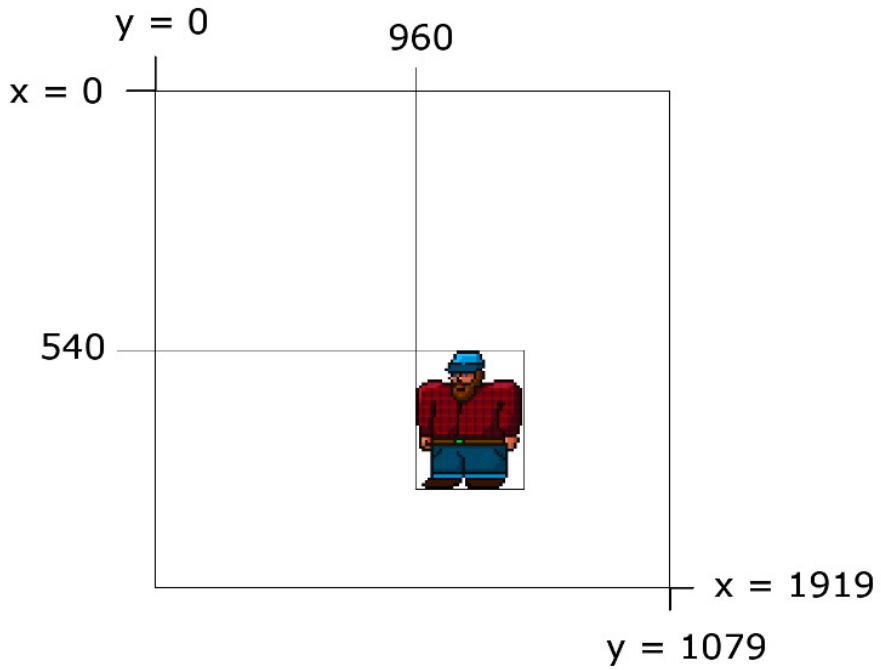
There are many different resolutions of monitor but, as an example, consider that a fairly typical gaming monitor might have 1,920 pixels horizontally and 1,080 pixels vertically.

The pixels are numbered, starting from the top left of the screen. As you can see from the following diagram, our 1,920 x 1,080 example is numbered from 0 through to 1,919 on the horizontal (x) axis and 0 through 1,079 on the vertical (y) axis:



A specific and exact screen location can therefore be identified by an x and y coordinate. We create our games by drawing the game objects such as the background, characters, bullets, and text to specific locations

on the screen. These locations are identified by the coordinates of the pixels. Take a look at the following hypothetical example of how we might draw at the approximately central coordinates of the screen. In the case of a 1,920 x 1080 screen, this would be at the 960, 540 position:

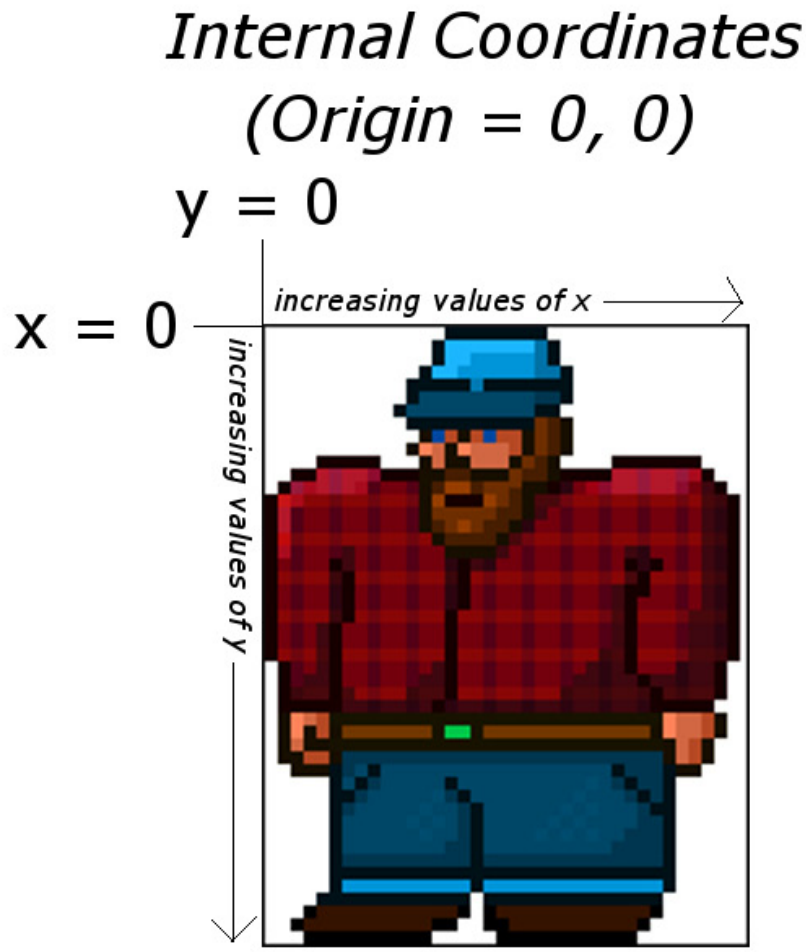


In addition to the screen coordinates, our game objects will each have their own similar coordinate system as well. Like the screen coordinate system, their **internal** or **local** coordinates start at 0,0 in the top left-hand corner.

In the previous image, we can see that 0,0 of the character is drawn at 960, 540 of the screen.

A visual, 2D game object, such as a character or perhaps a zombie, is called a **Sprite**. A sprite is typically made from an image file. All sprites have what is known as an **origin**.

If we draw a sprite to a specific location on the screen, it is the origin that will be located at this specific location. The 0,0 coordinates of the sprite are its origin. The following image demonstrates this:



Therefore, in the image showing the character drawn to the screen, although we drew the image at the central position (960, 540), it appears off to the right and down a bit.

This is important to know as it will help us understand the coordinates we use to draw all the graphics.

Important note

Note that, in the real world, gamers have a huge variety of screen resolutions, and our games will need to work with as many of them as possible. In the third project, we will see how we can make our games dynamically adapt to almost any resolution. In this first project, we will need to assume that the screen resolution is 1,920 x 1,080. If your screen resolution is higher, this will be fine. Don't worry if your screen is lower than this as I have provided a separate set of code for each chapter for the Timber!!! game. The code files are nearly identical apart from adding and swapping a few lines of code near the beginning. If you have a lower-resolution screen, then simply follow the code in this book, which assumes that you have a 1,920 x 1,080 resolution. When it comes to trying out the game, you can copy and paste the code files from the `lowres` folder in the first five chapters as appropriate. In fact, once the extra lines have been added from this first chapter, all the rest of the code will be identical, regardless of your screen resolution. I have supplied the low-

resolution code for each chapter, just as a convenience. How the few lines of code work their magic (scale the screen) will be discussed in the third project. The alternative code will work on resolutions as low as 960 x 540 and so should be OK on almost any PC or laptop.

Now, we can write our first piece of C++ code and see it in action.

Getting started with coding the game

Open up Visual Studio if it isn't already open. Open up the Timber!!! project by left-clicking it from the **Recent** list on the main Visual Studio window.

Find the **Solution Explorer** window on the right-hand side. Locate the `Timber.cpp` file under the **Source Files** folder.

Important note

.cpp stands for C plus plus.

Delete the entire contents of the code window and add the following code so that you have the same code yourself. You can do so in the same way that you would with any text editor or word processor; you could even copy and paste it if you prefer. After you have made the edits, we can talk about it:

```
// This is where our game starts from
int main()
{
    return 0;
}
```

Copy

This simple C++ program is a good place to start. Let's go through it line by line.

Making code clearer with comments

The first line of code is as follows:

```
// This is where our game starts from
```
Copy

Any line of code that starts with two forward slashes (`//`) is a comment and is ignored by the compiler. As such, this line of code does nothing. It is used to leave in any information that we might find useful when we come back to the code at a later date. The comment ends at the end of the line, so anything on the next line is not part of the comment. There is another type of comment called a **multi-line** or **c-style** comment, which can be used to leave comments that take up more than a single line. We will see some of them later in this chapter. Throughout this book, I will leave hundreds of comments to help add context and further explain the code.

The main function

The next line we see in our code is as follows:

```
int main()
```
Copy

int is what is known as a **type**. C++ has many types and they represent different types of data. An `int` is an **integer** or whole number. Hold that thought and we will come back to it in a minute.

The `main()` part is the name of the section of code that follows. The section of code is marked out between the opening curly brace (`{`) and the next closing curly brace (`}`).

So, everything in between these curly braces `{...}` is a part of `main`. We call a section of code like this a **function**.

Every C++ program has a `main` function and it is the place where the **execution** (running) of the entire program will start. As we progress through this book, eventually, our games will have many code files. However, there will only ever be one `main` function, and no matter what code we write, our game will always begin execution from the first line of code that's inside the opening curly brace of the `main` function.

For now, don't worry about the strange brackets that follow the function name `()`. We will discuss them further in *Chapter 4, Loops, Arrays, Switches, Enumerations, and Functions – Implementing Game Mechanics*, when we get to see functions in a whole new and more interesting light.

Let's look closely at the one single line of code within our `main` function.

Presentation and syntax

Take a look at the entirety of our `main` function again:

```
int main()
{
    return 0;
}
```

Copy

We can see that, inside `Main`, there is just one single line of code, `return 0;`. Before we move on to find out what this line of code does, let's look at how it is presented. This is useful because it can help

us prepare to write code that is easy to read and distinguished from other parts of our code.

First, notice that `return 0;` is indented to the right by one tab. This clearly marks it out as being internal to the `main` function. As our code grows in length, we will see that indenting our code and leaving white space will be essential to maintaining readability.

Next, notice the punctuation on the end of the line. A semicolon (`;`) tells the compiler that it is the end of the instruction and that whatever follows it is a new instruction. We call an instruction that's been terminated by a semicolon a `statement`.

Note that the compiler doesn't care whether you leave a new line or even a space between the semicolon and the next statement. However, not starting a new line for each statement will lead to desperately hard-to-read code, and missing the semicolon altogether will result in a **syntax error** and the game will not compile or run.

A section of code together, often denoted by its indentation with the rest of the section, is called a **block**.

Now that you're comfortable with the idea of the `main` function, indenting your code to keep it tidy, and putting a semicolon on the end of each statement, we can move on to finding out exactly what the `return 0;` statement actually does.

Returning values from a function

Actually, `return 0;` does almost nothing in the context of our game. The concept, however, is an important one. When we use the `return` keyword, either on its own or followed by a value, it is an instruction for the program execution to jump/move back to the code that got the function started in the first place.

Often, the code that got the function started will be yet another function somewhere else in our code. In this case, however, it is the operating system that started the `main` function. So, when `return 0;` is executed, the `main` function exits and the entire program ends.

Since we have a `0` after the `return` keyword, that value is also sent to the operating system. We could change the value of 0 to something else and that value would be sent back instead.

We say that the code that starts a function **calls** the function and that the function **returns** the value.

You don't need to fully grasp all this function information just yet. It is just useful to introduce it here. There's one last thing on functions that I will cover before we move on. Remember the `int` from `int main()`? This tells the compiler that the type of value that's returned from `main` must be an `int` (integer/whole number). We can return any value that qualifies as an `int`; perhaps 0, 1, 999, 6,358, and so on. If we try and return something that isn't an `int,` perhaps 12.76, then the code won't compile, and the game won't run.

Functions can return a big selection of different types, including types that we invent for ourselves! That type, however, must be made known to the compiler in the way we have just seen.

This little bit of background information on functions will make things smoother as we progress.

Running the game

You can even run the game at this point. Do so by clicking the **Local Windows Debugger** button in the quick-launch bar of Visual Studio. Alternatively, you can use the *F5* shortcut key:

You will just get a black screen. If the black screen doesn't automatically close itself, you can tap any key to close it. This window is the C++ console, and we can use this to debug our game. We don't need to do this now. What is happening is that our program is starting, executing from the first line of `main`, which is `return 0;`, and then immediately exiting back to the operating system.

We now have the simplest program possible coded and running. We will now add some more code to open a window that the game will eventually appear in.

Opening a window using SFML

Now, let's add some more code. The code that follows will open a window using SFML that Timber!!! will eventually run in. The window will be 1,920 pixels wide by 1,080 pixels high, and will be full screen (no border or title).

Enter the new code that is highlighted here to the existing code and then we will examine it. As you type (or copy and paste), try and work out what is going on:

```
// Include important libraries here
#include <SFML/Graphics.hpp>
// Make code easier to type with “using namespace”
using namespace sf;
// This is where our game starts from
int main()
{
    // Create a video mode object
    VideoMode vm(1920, 1080);
    // Create and open a window for the game
    RenderWindow window(vm, “Timber!!!”, Style::Fullscreen);
    return 0;
}
```

Copy

#including SFML features

The first thing we will notice in our new code is the `#include` directive.

The `#include` **directive** tells Visual Studio to *include*, or add, the contents of another file before compiling. The effect of this is that some other code, which we have not written ourselves, will be a part of our program when we run it. The process of adding code from other files into

our code is called **preprocessing** and perhaps unsurprisingly is performed by something called a **preprocessor**. The `.hpp` file extension means it is a **header** file.

Therefore, `#include <SFML/Graphics.hpp>` tells the preprocessor to include the contents of the `Graphics.hpp` file that is contained within the folder named `SFML.` It is the same folder that we created while setting up the project.

This line adds code from the aforementioned file, which gives us access to some of the features of SFML. Exactly how it achieves this will become clearer when we start writing our own separate code files and using `#include` to use them.

The main files that we will be including throughout this book are the SFML header files that give us access to all the cool game-coding features. We will also use `#include` to access the **C++ Standard Library** header files. These header files give us access to core features of the C++ language itself.

What matters for now is that we have a whole bunch of new functionalities that have been provided by SFML available to use if we add that single line of code.

The next new line is `using namespace sf;`. We will come back to what this line does soon.

OOP, classes, and objects

We will fully discuss OOP, classes, and objects as we proceed through this book. What follows is a brief introduction so that we can understand what is happening.

We already know that OOP stands for object-oriented programming. OOP is a programming paradigm, that is, a *way* of coding. OOP is generally accepted throughout the world of programming, in almost every language, as the best, if not the only, professional way to write code.

OOP introduces a lot of coding concepts, but fundamental to them all are **classes** and **objects**. When we write code, whenever possible, we want to write code that is reusable, maintainable, and secure. The way

we do this is by structuring our code as a class. We will learn how to do this in *Chapter 6, Object-Oriented Programming – Starting the Pong Game*.

All we need to know about classes for now is that once we have coded our class, we don't just execute that code as part of our game; instead, we create usable objects *from* the class.

For example, if we wanted 100 zombie **NPC**s (**non-player characters**), we could carefully design and code a class called `Zombie` and then, from that single class, create as many zombie objects as we like. Each and every zombie object would have the same functionality and internal data types, but each and every zombie object would be a separate and distinct entity.

To take the hypothetical zombie example further but without showing any code for the `Zombie` class, we might create a new object based on the `Zombie` class, like this:

```
Zombie z1;
```

Copy

The `z1` object is now a fully coded and functioning `Zombie` object. We could then do this:

```
Zombie z2;
Zombie z3;
Zombie z4;
Zombie z5;
```

Copy

We now have five separate `Zombie` **instances**, but they are all based on one carefully coded class. Let's take things one step further before we get back to the code we have just written. Our zombies can contain both behavior (defined by functions) as well as data, which might represent things such as the zombie's health, speed, location, or direction of travel. As an example, we could code our `Zombie` class to enable us to use our `Zombie` objects, perhaps like this:

```
z1.attack(player);
z2.growl();
z3.headExplode();
```

Copy

Important note

Note again that all this zombie code is hypothetical for the moment. Don't type this code into to Visual Studio – it will just produce a bunch of errors.

We would design our class so that we can use the data and behaviors in the most appropriate manner to suit our game's objectives. For example, we could design our class so that we can assign values for the data for each zombie object at the time we create it.

Let's say we need to assign a unique name and speed in meters per second at the time we create each zombie. Careful coding of the `Zombie` class could enable us to write code like this:

```
// Dave was a 100 metre Olympic champion before infection
// He moves at 10 metres per second
Zombie z1("Dave", 10);
// Gill had both of her legs eaten before she was infected
// She drags along at .01 metres per second
Zombie z2("Gill", .01);
```

Copy

The point is that classes are almost infinitely flexible, and once we have coded the class, we can go about using them by creating an object/instance *of* them. It is through classes and the objects that we create from them that we will harness the power of SFML. And yes, we will also write our own classes, including a `Zombie` class.

Let's get back to the real code we just wrote.

Using namespace sf

Before we move on and look more closely at `VideoMode` and `RenderWindow`, which as you have probably guessed are classes provided by SFML, we will learn what the `using namespace sf;` line of code does.

When we create a class, we do so in a **namespace**. We do this to distinguish our classes from those that others have written. Consider the `VideoMode` class. It is entirely possible that, in an environment such as Windows, somebody has already written a class called `VideoMode`. By using a namespace, we and the SFML programmers can make sure that the names of classes never clash.

The full way of using the `VideoMode` class is like this:

```
sf::VideoMode...
```

Copy

`using namespace sf;` enables us to omit the `sf::` prefix from everywhere in our code. Without it, there would be over 100 instances of `sf::` in this simple game alone. It also makes our code more readable, as well as shorter.

SFML VideoMode and RenderWindow

Inside the `main` function, we now have two new comments and two new lines of actual code. The first line of actual code is this:

```
VideoMode vm(1920, 1080);
```

Copy

This code creates an object called `vm` from the class called `VideoMode` and sets up two internal values of `1920` and `1080`. These values represent the resolution of the player's screen.

The next new line of code is as follows:

```
RenderWindow window(vm, “Timber!!!”, Style::Fullscreen);
```

Copy

In the previous line of code, we are creating a new object called `window` from the SFML-provided class called `RenderWindow`. Furthermore, we are setting up some values inside our window object.

Firstly, the `vm` object is used to initialize part of `window`. At first, this might seem confusing. Remember, however, that a class can be as varied and flexible as its creator wants to make it. And yes, some classes can contain other instances of other classes.

Tip

It is not necessary to fully understand how this works at this point, as long as you appreciate the concept. We code a class and then make useable objects from that class – a bit like an architect might draw a blueprint. You

certainly can't move all your furniture, kids, and dog into the blueprint, but you could build a house (or many houses) from the blueprint. In this analogy, a class is like a blueprint and an object is like a house.

Next, we use the `"Timber!!!"` value to give the window a name. Then, we use the predefined `Style::FullScreen` value to make our `window` object fullscreen.

Tip

`Style::FullScreen` is a value that's defined in SFML. It is useful because we don't need to remember the integer number the internal code uses to represent a full screen. The coding term for this type of value is `constant`. Constants and their close C++ relatives, **variables**, are covered in the next chapter.

Let's take a look at our window object in action.

Running the game

You can run the game again at this point. You will see a bigger black screen flash on and then disappear. This is the 1,920 x 1,080 fullscreen window that we just coded. Unfortunately, what is still happening is that our program is starting, executing from the first line of `main`, creating the cool new game window, then coming to `return 0;` and immediately exiting back to the operating system.

Next, we will add some code that will form the basic structure of every game in this book. This is known as the game loop.

The main game loop

We need a way to stay in the program until the player wants to quit. At the same time, we should clearly mark out where the different parts of our code will go as we progress with Timber!!!. Furthermore, if we are going

to stop our game from exiting, we had better provide a way for the player to exit when they are ready; otherwise, the game will go on forever!

Add the following highlighted code to the existing code and then we will go through it and discuss it all:

```
int main()
{
    // Create a video mode object
    VideoMode vm(1920, 1080);
    // Create and open a window for the game
    RenderWindow window(vm, "Timber!!!", Style::Fullscreen);
    while (window.isOpen())
    {
        /*
        ****************************************
        Handle the players input
        ****************************************
        */
        if (Keyboard::isKeyPressed(Keyboard::Escape))
        {
            window.close();
        }
        /*
        ****************************************
        Update the scene
        ****************************************
        */
        /*
        ****************************************
        Draw the scene
        ****************************************
        */
        // Clear everything from the last frame
        window.clear();
        // Draw our game scene here
        // Show everything we just drew
        window.display();
    }
```

Copy

```
    return 0;
}
```

While loops

The very first thing we saw in the new code is as follows:

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

The very last thing we saw in the new code is a closing `}`. We have created a **while** loop. Everything between the opening (`{`) and closing (`}`) brackets of the `while` loop will continue to execute, over and over, potentially forever.

Look closely between the parentheses `(...)` of the `while` loop, as shown here:

```
while (window.isOpen())
```
Copy

The full explanation of this code will have to wait until we discus loops and conditions in *Chapter 4, Loops, Arrays, Switches, Enumerations, and Functions – Implementing Game Mechanics*. What is important for now is that when the `window` object is set to closed, the execution of the code will break out of the `while` loop and move on to the next statement. Exactly how a window is closed is covered soon.

The next statement is, of course, `return 0;`, which ends our game.

We now know that our `while` loop will whiz round and round, repeatedly executing the code within it, until our window object is set to closed.

C-style code comments

Just inside the `while` loop, we can see what, at first glance, might look a bit like ASCII art:

```
        /*                                                                    Copy
        **************************************
        Handle the player's input
        **************************************
        */
```

Important note

ASCII art is a niche but fun way of creating images with computer text. You can read more about it here: https://en.wikipedia.org/wiki/ASCII_art.

The previous code is simply another type of comment. This type of comment is known as a C-style comment. The comment begins with (`/*`) and ends with (`*/`). Anything in between is just for information and is not compiled. I have used this slightly elaborate text to make it absolutely clear what we will be doing in each part of the code file. And of course, you can now work out that any code that follows will be related to handling the player's input.

Skip over a few lines of code and you will see that we have another C-style comment, announcing that in that part of the code, we will be updating the scene.

If you jump to the next C-style comment, it will be clear where we will be drawing all the graphics.

Input, update, draw, repeat

Although this first project uses the simplest possible version of a game loop, every game will need these phases in the code. Let's go over the steps:

1. Get the player's input (if any).
2. Update the scene based on things such as artificial intelligence, physics, or the player's input.
3. Draw the current scene.
4. Repeat these steps at a fast-enough rate to create a smooth, animated game world.

Now, let's look at the code that actually does something within the game loop.

Detecting a key press

Firstly, within the section that's identifiable by the comment with the `Handle the player's input` text, we have the following code:

```
if (Keyboard::isKeyPressed(Keyboard::Escape))
{
        window.close();
}
```

Copy

This code checks whether the *Esc* key is currently being pressed. If it is, the highlighted code uses the `window` object to close itself. Now, the next time the `while` loop begins, it will see that the `window` object is closed and jump to the code immediately after the closing curly brace of the `while` loop and the game will exit. We will discuss `if` statements more fully in *Chapter 2, Variables, Operators, and Decisions – Animating Sprites*.

Clearing and drawing the scene

Currently, there is no code in the `Update the scene` section, so let's move on to the `Draw the scene` section.

The first thing we will do is rub out the previous frame of animation using the following code:

```
window.clear();
```

Copy

What we would do now is draw every object from the game. However, we don't have any game objects.

The next line of code is as follows:

```
window.display();
```

Copy

When we draw all the game objects, we are drawing them to a hidden surface ready to be displayed. The `window.display()` code flips from the previously displayed surface to the newly updated (previously hidden) one. This way, the player will never see the drawing process as the surface has all the sprites added to it. It also guarantees that the scene will be complete before it is flipped. This prevents a graphical glitch known as **tearing**. This process is called **double buffering**.

Also note that all this drawing and clearing functionality is performed using our `window` object, which was created from the SFML `RenderWindow` class.

Running the game

Run the game and you will get a blank, full screen window that remains open until you press the *Esc* key.

That is good progress. At this stage, we have an executing program that opens a window and loops around, waiting for the player to press the *Esc* key to exit. Now, we are able to move on to drawing the background image of the game.

Drawing the game's background

Now, we will get to see some graphics in our game. What we need to do is create a sprite. The first one we will create will be the game background. We can then draw it in between clearing the window and displaying/flipping it.

Preparing the Sprite using a Texture

The SFML `RenderWindow` class allowed us to create our `window` object, which basically took care of all the functionality that our game's window needs.

We will now look at two more SFML classes that will take care of drawing sprites to the screen. One of these classes, perhaps unsurprisingly, is called `Sprite`. The other class is called `Texture`. A texture is a graphic stored in memory, on the **graphics processing unit** (**GPU**).

An object that's made from the `Sprite` class needs an object made from the `Texture` class in order to display itself as an image. Add the following highlighted code. Try and work out what is going on as well. Then, we will go through it, a line at a time:

```
int main()
{
    // Create a video mode object
    VideoMode vm(1920, 1080);
    // Create and open a window for the game
    RenderWindow window(vm, "Timber!!!", Style::Fullscreen);
    // Create a texture to hold a graphic on the GPU
    Texture textureBackground;
    // Load a graphic into the texture
    textureBackground.loadFromFile("graphics/background.png");
    // Create a sprite
    Sprite spriteBackground;
    // Attach the texture to the sprite
    spriteBackground.setTexture(textureBackground);
    // Set the spriteBackground to cover the screen
    spriteBackground.setPosition(0,0);
    while (window.isOpen())
    {
```

Copy

First, we create an object called `textureBackground` from the SFML `Texture` class:

```
Texture textureBackground;
```

Copy

Once this is done, we can use the `textureBackground` object to load a graphic from our `graphics` folder into `textureBackground`, like this:

```
textureBackground.loadFromFile("graphics/background.png");
```

Copy

Tip

We only need to specify `graphics/background` as the path is relative to the Visual Studio **working directory** where we created the folder and added the image.

Next, we create an object called `spriteBackground` from the SFML `Sprite` class with this code:

```
Sprite spriteBackground;
```

Copy

Then, we can associate the `Texture` object (`backgroundTexture`) with the `Sprite` object (`backgroundSprite`), like this:

```
spriteBackground.setTexture(textureBackground);
```
Copy

Finally, we can position the `spriteBackground` object in the `window` object at the `0,0` coordinates:

```
spriteBackground.setPosition(0,0);
```
Copy

Since the `background.png` graphic in the graphics folder is 1,920 pixels wide by 1,080 pixels high, it will neatly fill the entire screen. Just note that this previous line of code doesn't actually show the sprite. It just sets its position, ready for when it is shown.

The `backgroundSprite` object can now be used to display the background graphic. Of course, you are almost certainly wondering why we had to do things in such a convoluted way. The reason is because of the way that graphics cards and OpenGL work.

Textures take up graphics memory, and this memory is a finite resource. Furthermore, the process of loading a graphic into the GPU's memory is very slow – not so slow that you can watch it happen or that you will see your PC noticeably slow down while it is happening, but slow enough that you can't do it every frame of the game loop. So, it is useful to disassociate the actual texture (`textureBackground`) from any code that we will manipulate during the game loop.

As you will see when we start to move our graphics, we will do so using the sprite. Any objects that are made from the `Texture` class will sit happily on the GPU, just waiting for an associated `Sprite` object to tell it where to show itself. In later projects, we will also reuse the same `Texture` object with multiple different `Sprite` objects, which makes efficient use of GPU memory.

In summary, we can state the following:

> Textures are very slow to load onto the GPU.
> Textures are very fast to access once they are on the GPU.

We associate a `Sprite` object with a texture.

We manipulate the position and orientation of `Sprite` objects (usually in the `Update the scene` section).

We draw the `Sprite` object, which, in turn, displays the `Texture` object that is associated with it (usually in the `Draw the scene` section). So, all we need to do now is use our double buffering system, which is provided by our `window` object, to draw our new `Sprite` object (`spriteBackground`), and we should get to see our game in action.

Double buffering the background sprite

Finally, we need to draw that sprite and its associated texture in the appropriate place in the game loop.

Tip

Note that when I present code that is all from the same block, I don't add the indentations because it lessens the instances of line wraps in the text of the book. The indenting is implied. Check out the code file in the

download bundle to see full use of indenting.

Add the following highlighted code:

```
/*
**************************************
Draw the scene
**************************************
*/
// Clear everything from the last run frame
window.clear();
// Draw our game scene here
window.draw(spriteBackground);
// Show everything we just drew
window.display();
```

Copy

The new line of code simply uses the `window` object to draw the `spriteBackground` object, in between clearing the display and showing the newly drawn scene.

We now know what a sprite is, and that we can associate a texture with it and then position it on the screen and finally draw it. The game is ready to be run again so that we can see the results of this code.

Running the game

If we run the program now, we will see the first signs that we have a real game in progress:

It’s not going to get Indie Game of the Year on Steam in its current state, but we are on the way at least!

Let’s look at some of the things that might go wrong in this chapter and as we proceed through this book.

Handling errors

There will always be problems and errors in every project you make. This is guaranteed! The tougher the problem, the more satisfying it is when you solve it. When, after hours of struggling, a new game feature finally bursts into life, it can cause a genuine high. Without this struggle, it would somehow be less worthwhile.

At some point in this book, there will probably be some struggle. Remain calm, be confident that you will overcome it, and then get to work.

Remember that, whatever your problem, it is very likely you are **not** the first person in the world to have ever had this same problem. Think of a concise sentence that describes your problem or error and then type it into Google. You will be surprised how quickly, precisely, and often, someone else will have already solved your problem for you.

Having said that, here are a few pointers (pun intended; see *Chapter 10, Pointers, the Standard Template Library, and Texture Management*) to get you started in case you are struggling with making this first chapter work.

Configuration errors

The most likely cause of problems in this chapter will be **configuration errors**. As you probably noticed during the process of setting up Visual Studio, SFML and the project itself, there's an awful lot of filenames, folders, and settings that need to be just right. Just one wrong setting could cause one of a number of errors, whose text don't make it clear exactly what is wrong.

If you can't get the empty project with the black screen working, it might be easier to start again. Make sure all the filenames and folders are appropriate for your specific setup and then get the simplest part of the code running. This is the part where the screen flashes black and then closes. If you can get to this stage, then configuration is probably not the issue.

Compile errors

Compile errors are probably the most common error we will experience going forward. Check that your code is identical to mine, especially semicolons on the ends of lines and subtle changes in upper and lower case for class and object names. If all else fails, open the code files in the download bundle and copy and paste it in. While it is always possible that a code typo made it into this book, the code files were made from actual working projects – they definitely work!

Link errors

Link errors are most likely caused by missing SFML `.dll` files. Did you copy all of them into the project folder?

Bugs

Bugs are what happen when your code works, but not as you expect it to. Debugging can actually be fun. The more bugs you squash, the better your game and the more satisfying your day's work will be. The trick to solving bugs is to find them early! To do this, I recommend running and playing your game every time you implement something new. The sooner you find the bug, the more likely the cause will be fresh in your mind. In this book, we will run the code to see the results at every possible stage.

Summary

This was quite a challenging chapter and perhaps a little bit mean to be the first one. It is true that configuring an IDE to use a C++ library can be a bit awkward and long. Also, the concepts of classes and objects are well known to be slightly awkward for people who are new to coding.

Now that we are at this stage, however, we can totally focus on C++, SFML, and games. As we progress with this book, we will learn more and more C++, as well as implement increasingly interesting game features. As we do so, we will take a further look at things such as functions, classes, and objects to help demystify them a little more.

We have achieved plenty in this chapter, including outlining a basic C++ program with the main function, constructing a simple game loop that listens for player input and draws a sprite (along with its associated texture) to the screen.

In the next chapter, we will learn about all the C++ we need to draw some more sprites and animate them.

FAQ

Here are some questions that might be on your mind:

Q) I am struggling with the content that's been presented so far. Am I cut out for programming?

A) Setting up a development environment and getting your head around OOP as a concept is probably the toughest thing you will do in this book. As long as your game is functioning (drawing the background), you are ready to proceed with the next chapter.

Q) All this talk of OOP, classes, and objects is too much and kind of spoiling the whole learning experience.

A) Don't worry. We will keep returning to OOP, classes, and objects constantly. In *Chapter 6*, *Object-Oriented Programming – Starting the Pong Game*, we will really begin getting to grips with the whole OOP thing. All you need to understand for now is that SFML have written a whole load of useful classes and that we get to use this code by creating usable objects from those classes.

Q) I really don't get this function stuff.

A) It doesn’t matter; we will be returning to it again constantly and will learn about functions more thoroughly. You just need to know that, when a function is called, its code is executed, and when it is done (reaches a `return` statement), the program jumps back to the code that called it.