SFML Essentials

A fast-paced, practical guide to building functionally enriched
2D games using the core concepts of SFML

Milcho G. Milchev

SFML Essentials

A fast-paced, practical guide to building functionally
enriched 2D games using the core concepts of SFML

Milcho G. Milchev

[PACKT] open source*
community experience distilled

PUBLISHING

BIRMINGHAM - MUMBAI

SFML Essentials

Copyright © 2015 Packt Publishing

Credits

Author
Milcho G. Milchev

Reviewers
James Cowgill

Mark Fielbig

Stefan Schindler

Commissioning Editor
Edward Bowkett

Acquisition Editor
Harsha Bharwani

Content Development Editor
Natasha D'Souza

Technical Editor
Narsimha Pai

Copy Editors
Vikrant Phadke

Neha Vyas

Project Coordinator
Akash Poojary

Proofreaders
Simran Bhogal

Kevin McGowan

Indexer
Rekha Nair

Graphics
Valentina D'silva

Disha Haria

Production Coordinator
Komal Ramchandani

Cover Work
Komal Ramchandani

About the Author

Milcho G. Milchev has been a game programmer since he was in high school. He continues to work on games on a daily basis. Some of the games that he has worked on include *Unholy Flesh* and *WhiteCity*. Currently, he works for Dream Harvest, an indie game studio with a lot of talented people. They are developing their first strategy game. Milcho has also followed the development of SFML for several years and has developed his own game engine based on this library.

About the Reviewers

James Cowgill is a computer science student at the University of York, and has set aside a year to work with the MIPS team at Imagination Technologies. Over the last 10 years, he has experimented with and learned a variety of languages including C, C++, Python, and C#. In his free time, he often works on improving the Debian Linux distribution and helping out developers of open source projects.

Mark Fielbig was first introduced to game development through his love for games and his desire to learn how they worked. He started by developing and releasing numerous Flash and mobile games. Mark now holds a Masters of Computer Science degree from Stony Brook University and his interests include application development, distributed systems, and multi-tiered system architecture.

Stefan Schindler is a software developer and engineer. He loves C++, Python, Vim, Linux, Git, code design, and software tests.

Stefan has been in the SFML community since June 2008. He was looking for a modern C++ library to finish a small university game project, which was a *Tetris* clone. In 2014, several people were invited to join the SFML development team to help with their progress. Stefan accepted the invitation. His likes integrating automated software tests and continuous integration. He also hosts the website and the popular IRC channel #sfml at `irc.sfml-dev.org`.

In his free time, Stefan likes to develop libraries and gaming experiments in C++ and Python. SFGUI is one of his projects, and it is still maintained, mainly by its contributors. Stefan also likes to write articles. He has has published a couple of them under the name, *Game Development Design*, available for free at `www.optank.org`.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`https://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

SFML Essentials is a practical set of tutorials about the Simple and Fast Multimedia Library (SFML) that teaches you how to use the library quickly and easily. The book establishes the core concepts of game development by providing the best practices in this field.

Game development can be a difficult topic to understand. *SFML Essentials* will provide enough knowledge about SFML for you to start implementing your ideas as soon as possible. This book also includes a number of fully working examples, which you can use and modify to suit your needs.

What this book covers

Chapter 1, *Getting Started with SFML*, is an introductory chapter about the SFML library. It goes through the process of creating a window and rendering basic shapes on the screen. This chapter concludes with a functional mini-game, the code of which is explained in detail.

Chapter 2, *Loading and Using Textures*, introduces the concept of sprites and textures and their interaction with the window. At the end of this chapter, the problems of resource management are tackled by building a robust asset manager.

Chapter 3, *Animating Sprites*, builds on the sprite class by animating it with spritesheets. A fully functional Animator class is constructed by the end of this chapter. This class can also be used outside the context of this book.

Chapter 4, *Manipulating a 2D Camera*, introduces the concept of cameras in a scene and the ways they can be interacted with. This chapter also discusses direct rendering with OpenGL on an SFML window.

Chapter 5, *Exploring a World of Sounds and Text*, discusses the audio and text components of a game. The concept of spatialization (3D audio) is covered as well.

Chapter 6, *Rendering Special Effects with Shaders*, dives into the topic of shaders and their uses in special effects. Postprocessing is briefly covered by giving an example of a pixelation shader.

Chapter 7, *Building Multiplayer Games*, discusses the topic of networking. After a brief introduction, the transport layer (TCP and UDP sockets) is explored. A working networking example between two PCs is presented at the end of this chapter.

What you need for this book

For this book you will require Microsoft Visual Studio 2012 Express for Windows Desktop, which can be downloaded from `http://www.microsoft.com/en-gb/download/details.aspx?id=34673`. You also need to install SFML v2.1 Visual C++ 11 (2012) 32-bit from `http://sfml-dev.org/download/sfml/2.1/`.

You also require a graphics card capable for running at 1024 x 768 pixels or a higher display resolution, and is compatible with OpenGL 2.0.

Who this book is for

SFML Essentials is for people who have experience in the field of game programming but want to use SFML for their next project. Perhaps you have an idea for a game, but your current environment is too slow to accommodate the needs of your game, or you want a cross-platform solution in your favorite language. In either case, this book will quickly guide you towards your goal. We assume that you possess a very basic understanding of game development, but solid understanding in at least one of the supported languages is required.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and explanations of their meanings.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "This is the reference that initializes the member's `Sprite&` field."

A block of code is set as follows:

```
sf::Uint32 style = sf::Style::Titlebar | sf::Style::Close;
```

New terms and important words are shown in bold.

> Warnings or important notes appear in a box like this.

> Tips and tricks appear like this.

Reference images

The images used inside this book have been taken from the following locations:

- *Chapter 2*, *Loading and Using Textures,*
    - Leaf: `http://pixabay.com/en/leaf-maple-leaf-green-flora-maple-310682/`
    - Egg: `http://pixabay.com/en/egg-oval-food-round-157224/`
    - Tile: `http://pixabay.com/en/cube-pattern-seamless-tile-magenta-405259/`
- *Chapter 3*, *Animating Sprites,*
    - Crystals: `http://opengameart.org/content/rotating-crystal-animation-8-step`
- *Chapter 6*, *Rendering Special Effects with Shaders,*
    - The Tower (level 1 concept, pixelation): `http://www.dreamharvest.co.uk/?page_id=171`

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from: `https://www.packtpub.com/sites/default/files/downloads/7326OS_ColorImages.pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Getting Started with SFML

In this chapter, we will explore the window and graphic modules of SFML and focus on strengthening our basics. We will also see how a typical game loop looks like in SFML, and how we can handle input to manipulate shapes on the screen.

In this chapter, we will cover:

- Window creation
- The game loop
- Event handling
- Shape rendering and transformations

Creating windows

The first thing you would probably want to do when you start developing a game is open a window. In SFML, this couldn't have been made any easier. Only one line of code is necessary to create a window:

```cpp
Main.cpp
    #include <SFML/Window.hpp>

int main()
{
    sf::Window window(sf::VideoMode(300, 200), "The title");

    return 0;
}
```

The only thing that the main function does is initialize the `window` variable by calling the `sf::Window` constructor, after which the program exits. There is an alternative way to open a window by using the default constructor and calling `Window::create()` later on. This function takes exactly the same arguments as the constructor, which we just went through. If `Window::create()` is called on a window, which is already open, it closes the window and reinitializes it with the new set of parameters.

In the example given in the preceding screenshot, notice that both `Window` and `VideoMode` are in the `sf` namespace. Every class in SFML is under that namespace, which separates all the classes in SFML from the classes in other libraries.

If we run the code in the example, we won't see much. The program exits immediately after it creates the window. That is because we just create a window without doing anything with it, and the program naturally exits after it reaches the end of the `main()` method. The fact that we created a window doesn't mean that it is fully functional (at least not yet). We will have to program it according to what we want it to do. Now, let's block the main function from finishing, by delaying the window's thread. SFML provides a simple interface for that; just add the `sf::sleep(sf::seconds(3))` line after the line which creates the window. Now, the window is clearly visible for the duration of the sleep.

We can specify various configurations while creating the window—window size, title, style, and graphics settings. You may have noticed that we pass two arguments to the the `Window` constructor—an instance of `VideoMode` and a string (the title). The constructor can actually take up to four parameters, the last two being optional—`Style` and `ContextSettings`. The next part covers what those arguments mean, and how to use them.

VideoMode

The `VideoMode` class contains information about the video mode of the window, such as width, height, and bits per pixel. This last parameter is the number of bits used to represent the color of a single pixel. It has a default value of 32, which is unlikely to change on recent hardware. For example, a value of 8 would produce a monochrome result. If we want to create a fullscreen window, the supplied values have to be supported by the machine's monitor and graphics card. If we choose invalid arguments for a fullscreen window, the window creation will simply fail. The validity of the `VideoMode` class can can be checked with the `VideoMode::isValid()` method, which returns a boolean as the result.

If we need to create a window according to the size (or the pixel depth) of the desktop mode, `VideoMode::getDesktopMode()` is available as a static method. On the other hand, if we were to create a window in fullscreen mode, we might want to check the available resolutions with `VideoMode::getFullScreenModes()`. This returns `std::vector` of video modes, and we can choose one of the modes ourselves, or let the user decide which suits them best.

However, merely specifying fullscreen video mode is not enough to create a fullscreen window. We need to set a style as well.

Style

The `Style` argument is a bit mask. A mask is a combination of flags where each flag represents a specific bit of the mask. In this case, the flags are stored in an enum in the `sf::Style` namespace. We can use a combination of flags to create the desired mask. Here is what SFML offers in terms of styles:

Enum value	Description
`sf::Style::None`	The window doesn't have any decorations, and it cannot be combined with any other style.
`sf::Style::Titlebar`	This adds a titlebar.
`sf::Style::Resize`	This adds a maximize button. This also enables the window to be manually resized.
`sf::Style::Close`	This adds a close button.
`sf::Style::Fullscreen`	This launches the window in fullscreen mode. Note that this cannot be combined with any other style and requires a valid video mode.
`sf::Style::Default`	This combines `Titlebar`, `Resize`, and `Close`. This is the default style.

The way you can combine different styles is by using the bitwise OR operator. So in a case where we want a window with a titlebar and a close button, we write:

```
sf::Uint32 style = sf::Style::Titlebar | sf::Style::Close;
```

The only thing left to do here is to pass that style as the third argument of the `Window` construct.

ContextSettings

The final argument is an instance of `ContextSettings`. This structure is a collection of settings, which describes the desired rendering context. SFML uses OpenGL for rendering under the hood and thus those settings are directly relevant to it. The available context settings are as follows:

- `depthBits`: This refers to the number of depth buffer bits
- `stencilBits`: This refers to the number of stencil buffer bits
- `antialiasingLevel`: This refers to the requested number of multisampling levels
- `majorVersion` and `minorVersion`: These refer to the requested version of OpenGL
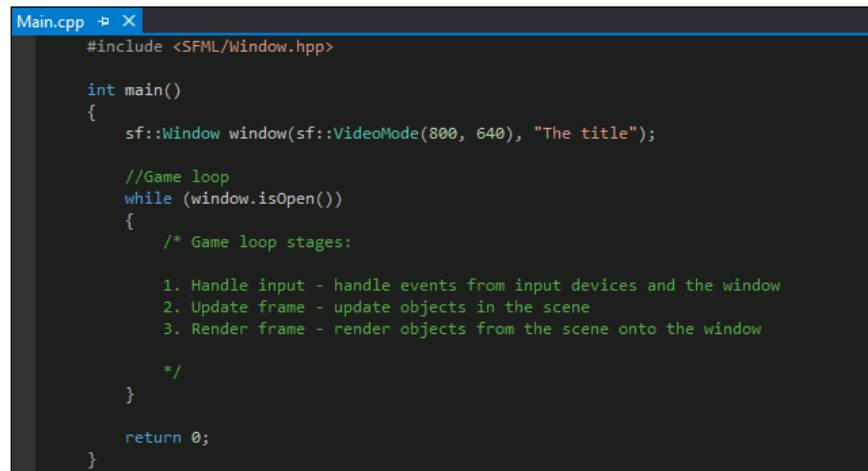
Each of these settings will be explained in more detail in *Chapter 5, Manipulating a 2D Camera*, where you will learn how to render things directly using OpenGL.

Disabling the mouse cursor

The `Window` class has a method which sets cursor visibility on or off—`Window::setMouseCursorVisible()`. This is useful for games that don't use a cursor, or when we want to change the image of the cursor to something different to the default.

The game loop

Every game needs a loop. This is what keeps it going. Otherwise the program will just end, and we will not be able to see much. Here is what a typical game loop looks like:

```cpp
#include <SFML/Window.hpp>

int main()
{
    sf::Window window(sf::VideoMode(800, 640), "The title");

    //Game loop
    while (window.isOpen())
    {
        /* Game loop stages:

        1. Handle input - handle events from input devices and the window
        2. Update frame - update objects in the scene
        3. Render frame - render objects from the scene onto the window

        */
    }

    return 0;
}
```

A typical game loop has three main stages:

- Input handling
- The update frame
- The render frame

Input handling in SFML can be done either through capturing events, which have been dispatched by the window, or by directly querying input devices for their current state. Both methods have different uses. For example, we might want to close the window on a button press event, or to move our main character to the right as long as a certain key is pressed (direct key query).

We reach the update frame stage after we capture and use our events. This is the stage where we want to advance our game logic and update our world state.

The final stage of the loop comes right after we finish updating our objects. Here, we clear everything that has been drawn from the last time and render every object on the screen again.

Going back to the example of our game loop, it currently doesn't perform the things it's supposed to and, if we try to run the code, it becomes obvious that the window doesn't respond to inputs. This is because we don't perform the first of the three important steps in the loop—handling the input.

Event handling

Events can be polled from the window by `bool Window::pollEvent(sf::Event& event)`. If there is an event waiting to be handled, the function will return `true`, and the event variable will be filled with the event data. If not, the function returns `false`. It is also important to note that there might be more than one event at a time; so we have to be sure to capture every event possible. Here is what a typical event loop looks like:

```cpp
while (window.isOpen())
{
    sf::Event event;
    while (window.pollEvent(event))
    {
    }

    //Update frame

    //Render frame
}
```

Running the code now produces more satisfactory results—we can move the window around, resize, and minimize it. However, there is still one problem—the close button doesn't work. SFML doesn't assume that the window should close after the user clicks on the close button. Maybe we want to save the player's progress or ask them if they are sure first. This means that we have to implement the close button functionality ourselves.

Before we proceed, it is important to note that the `Event` class in C++ contains a union. This means that only one of its members is valid. Accessing any other member will result in undefined behavior. We can get the valid member by looking at the type of the event.

The event types can be logically split into four sections, depending on what they are related to:

- Window
- Keyboard
- Mouse
- Joystick

Window related events

Enum value	Member associated	Description
`Event::Closed`	None	This event is triggered when the OS detects that the user wants to close a window—the close button, key combination, and so on.
`Event::Resized`	`Event::size` holds the new size of the window	This event is triggered when the OS detects that the window has been resized manually, or when `Window::setSize()` has been used.
`Event::LostFocus` `Event::GainedFocus`	None	This event is triggered when the window loses or gains focus. Windows which are out of focus don't receive keyboard events.

Keyboard related events

Enum value	Member associated	Description

`Event::KeyPressed`
 	
`Event::KeyReleased`
	`Event::key` holds the pressed/released key	This event is triggered when a single button is pressed or released on a focused window.

`Event::TextEntered`	`Event::text` holds the UTF-32 unicode value of the character	This event is triggered every time a character is typed. This produces a printable character from the user input, and is useful for text fields.

Mouse related events

Enum value	Member associated	Description
`Event::MouseMoved`	`Event::mouseMove` holds the new mouse position	This event is triggered when the mouse changes its position inside the window.

`Event::MouseButtonPressed`
 	
`Event::MouseButtonReleased`
	`Event::mouseButton` holds the pressed/ released button and the mouse position	This event is triggered when a mouse button is pressed inside a window. Five buttons are currently supported—left, right, middle, extra button 1, and extra button 2.

`Event::MouseWheelMoved`	`Event::mouseWheel` holds the delta ticks of the wheel and the mouse position	This event is triggered when the scroll wheel moves inside a window.

Joystick related events

Enum value	Member associated	Description

`Event::JoystickConnected`
 	
`Event::JoystickDisconnected`
	`Event::joystickConnect` holds the ID of the joystick just connected/disconnected	This event is triggered when a joystick connects or disconnects.

`Event::JoystickButtonPressed`
 	
`Event::JoystickButtonReleased`
	`Event::joystickButton` holds the number of the button pressed and the joystick ID	This is triggered when a button on a joystick is pressed. SFML supports a maximum of 8 joysticks with up to 32 buttons each.

`Event::JoystickMoved`	`Event::joystickMove` holds the moved axis, the new axis position, and the joystick ID	This is triggered when an axis of a joystick is moved. The move threshold can be set via `Window::setJoystick Threshold()`. SFML supports up to 8 axes.

Using events

After we get the event by calling `Window::pollEvent()`, we can check its type by looking at `Event::type`. The event is of type `Event::EventType`, which is an enum inside the `Event` class. Here is how a typical close event can be implemented:

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

Here, the `Window::close()` function will take care of closing the window. If a `window` variable gets out of scope, the destructor is called, and the `window` is closed.

If we want to handle more than one event, it makes sense to use the `switch` statement, as it improves readability. Let's see how the keyboard key presses and releases work:

```cpp
sf::Event event;
while (window.pollEvent(event))
{
    switch (event.type)
    {
    case sf::Event::EventType::Closed:
        window.close();
        break;
    case sf::Event::EventType::KeyPressed:
        //Change the title if the space is pressed
        if (event.key.code == sf::Keyboard::Key::Space)
            window.setTitle("Space pressed");
        break;
    case sf::Event::EventType::KeyReleased:
        //Change the title again if space is released
        if (event.key.code == sf::Keyboard::Key::Space)
            window.setTitle("Space released");
        //Close the window if the Escape key is released
        else if (event.key.code == sf::Keyboard::Key::Escape)
            window.close();
        break;
    default:
        break;
    }
}
```

The code in the preceding figure demonstrates how we can capture events to change the the title of the window every time the *Space* key is pressed and released. Apart from that, when the *Escape* key is released, the window closes. Notice that `Event::key` contains a member called `code`, which is an enum of type `Keyboard::Key`. You can use this formula to handle the rest of the event types without much difficulty. However, the case with `Event::EventType::TextEntered` is a bit more interesting. A single key press/release can be detected and handled in a relatively straightforward manner. When it comes to certain specific characters though, things start to get a bit more complex. For example, if we want to detect when the *!* symbol has been typed, we have to look up whether two individual keys have been pressed at the same time (*Shift + 1* on most keyboard layouts). In such cases, SFML saves us a lot of work by providing the simple and easy-to-use `TextEntered` event. The event is only fired when a combination of keys representing a character are pressed; meaning that a single key (only *Shift*, for example) might not trigger the event. Of course, if *K* alone is pressed, the event will be fired normally, and will contain the character.

Take a look at this example where a string is assembled out of characters using the `TextEntered` event and, when the *Enter* (or *Return*) button is pressed, the text is set as the title:

```cpp
sf::String buffer;
while (window.isOpen())
{
    sf::Event event;
    while (window.pollEvent(event))
    {
        switch (event.type)
        {
        case sf::Event::EventType::Closed:
            window.close();
            break;
        case sf::Event::EventType::TextEntered:
            //Add the character directly to the string
            buffer += event.text.unicode;
            break;
        case sf::Event::EventType::KeyReleased:
            //Change the title to the current buffer and clear the buffer
            if (event.key.code == sf::Keyboard::Key::Return)
            {
                window.setTitle(buffer);
                buffer.clear();
            }
            break;
        default:
            break;
        }
    }
}
```

Note that the string buffer that we are using is of type `sf::String` and not `std::string`. The `sf::String` class is created to automatically handle conversion between string types and encodings. As such, we do not have to worry about the language or symbols on a keyboard layout—it can store any character from any language.

To finish off event handling, it is important to mention that there is an alternative to how events are pulled from the window. Apart from using `Window::pollEvent()`, we can also use `bool Window::waitEvent(Event& event)`, which blocks the thread until an event is received. It only returns `false` when something wrong occurs inside (an error or exception of some sort), otherwise it always returns `true`. This can be useful when we require the user to do something before the application can continue, or if we want to handle the input on another thread, for example. In the latter scenario, only that thread will be blocked, allowing the game loop to continue running. Now that we've discussed events, let's move on to something more interesting—rendering.

Shape rendering and transformations

Obviously, we wouldn't need windows if there weren't any objects to be rendered, and we wouldn't need events if we didn't want to use the input to animate those objects. SFML provides quite a few ways for us to render objects on the screen, and we will explore the main ones in this book. Before we proceed to rendering though, we need to make sure that our render cycle is in place and in correct order.

The render frame

Do you, remember the `Window` class? That isn't of much use now, since it doesn't provide an interface to draw SFML shapes. We have to use a class called `RenderWindow` to do that. This class is derived from the `Window` class and adds the drawing functionality. Don't worry though, it doesn't strip any functionality from its parent, it just adds more functionality on top of it. So we can still create it, poll events, and so on, in the same way we do with the base class `Window`. Here is an example of a game loop with a render cycle:

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

int main()
{
    sf::RenderWindow window(sf::VideoMode(640, 480), "The title");

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

        //Update scene

        //Render cycle
        window.clear(sf::Color::Black);

        //Render objects

        window.display();
    }

    return 0;
}
```

It is important to note here that the `RenderWindow` class is from the SFML graphics module, which means that we have to include `<SFML/Graphics.hpp>`, rather than `<SFML/Window.hpp>`. However, since it is derived from the `Window` class it can still be used in our code without changing anything but the variable type.

The render cycle will look quite straightforward, if you have any previous game programming experience. Basically it breaks down to this:

- Clear the canvas you intend to draw on
- Draw onto the canvas
- Display the canvas

This render routine happens in every frame (the loop cycle). If you are not familiar with the rendering process, it might seem a bit odd and wasteful to throw away everything from the last frame and render all objects in the scene again (even those which haven't changed since the last time). However, graphics cards are well optimized to cope with this routine, and maximize efficiency and performance wherever possible. Avoid using any other structure, since it will only slow you down without bringing any major benefits.

Another thing to note is that the canvas which we are rendering on is double buffered. This is very common in rendering. The way this works is quite simple—the canvas has two sides that you can render on. Throughout the render frame, we work only on one of the sides—the one which is not shown on the screen. After we finish rendering, we flip the canvas and show what we've done. In the next frame, we work on the other side of the canvas, and so on. This technique allows us to show the scene only after we've finished rendering it. In SFML, we flip the canvas (it's also sometimes known as "swap the buffers") by calling `Window::display()`.

Apart from that, the `Window::display()` method can put the to thread sleep for a calculated amount of time to achieve a target framerate (frames per second). We can set the desired framerate by calling `Window::setFramerateLimit()` once at the beginning of the program. The function doesn't guarantee the limiting of the framerate to the exact amount we pass it, but rather it makes a close approximation.

`Window::clear()` clears the canvas for redrawing. Notice that it takes a `sf::Color` argument, which is an RGBA representation of a color. We can initialize it manually by calling the constructor and passing each value individually, or we can use one of the preset colors. For example `Color::Red`, `Color::Blue`, `Color::Magenta`, and more.

Shape drawing

Now that we are familiar with the render routine, let's render some shapes on the screen. We will start with the basic shapes and explore the alternatives later on. When we want to draw a shape, we have to create the object first. Here is the initialization code for two shapes. Place it just before the game loop.

```
sf::CircleShape circleShape(50);
circleShape.setFillColor(sf::Color::Red);
circleShape.setOutlineColor(sf::Color::White);
circleShape.setOutlineThickness(3);

sf::RectangleShape rectShape(sf::Vector2f(50, 50));
rectShape.setFillColor(sf::Color::Green);
```

A few new classes make an appearance in this example—`CircleShape`, `RectangleShape`, and `Vector2f`.

You can probably guess what the `Vector2f` class is for—it is a 2D vector which holds two `float`s. There are also classes such as `Vector2i` (for integers), `Vector2u` (for unsigned `integer`s), `Vector3i` (for 3D vectors which hold integers), and `Vector3f` (3D vectors which hold floats). We can even create our own 2D and 3D vectors, which hold custom types, by using the template classes `sf::Vector2<class>` and `sf::Vector3<class>`.

`CircleShape`, `RectangleShape`, and `ConvexShape` derive from the abstract class `Shape`, which is defined by a set number of vertices (points). The `CircleShape` is just a regular polygon with a set number of vertices. We can specify how detailed the circle should be with the second argument in the constructor, which is optional, with a default value of 30. On the other hand, `RectangleShape` always has four vertices. The constructors of both shapes take their dimensions—the radius of the circle and the width and height of the rectangle.

`ConvexShape` is a shape for which we have to specify the vertices explicitly. There isn't a restriction on the number of vertices, but they have to form a convex shape, otherwise the shape will not be drawn correctly.

Apart from that, shapes can have colors and outlines, which can be modified with `Shape::setFillColor()`, `Shape::setOutlineColor()` and `Shape::setOutlineThickness()`. This last one sets the number of pixels for the outline.

[19]

To render the preceding shapes, we can use the `RenderWindow::draw()` function. Here is how we can implement it into our render frame:

```cpp
//Render cycle
window.clear(sf::Color::Black);

window.draw(circleShape);
window.draw(rectShape);

window.display();
```

Running the code produces the following result:
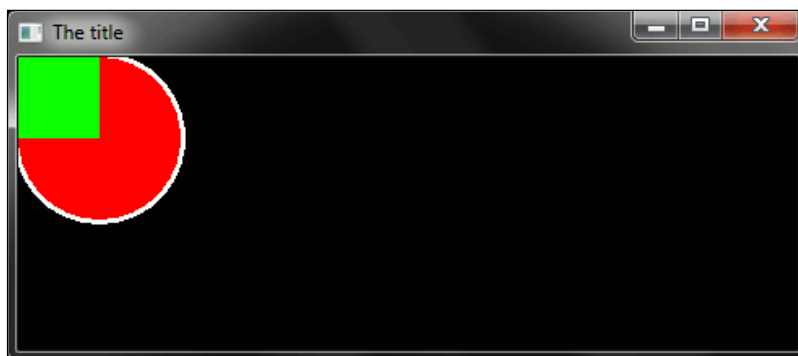
We can immediately note that the render order makes a big difference. Background objects have to be rendered first, followed by anything in the foreground. In this example, the circle is rendered first and so it is in the background, whereas the rectangle sits on top of the circle in the foreground.

We can use `ConvexShape` by specifying the number of points with the `ConvexShape::setPointCount()` function, and set those points in order with `ConvexShape::setPoint()`. Here is an example with a triangle:

```cpp
sf::ConvexShape triangle;
triangle.setPointCount(3);
triangle.setPoint(0, sf::Vector2f(100, 0));
triangle.setPoint(1, sf::Vector2f(100, 100));
triangle.setPoint(2, sf::Vector2f(0, 100));
triangle.setFillColor(sf::Color::Blue);
```

After drawing it in the window, we get a nice blue triangle:
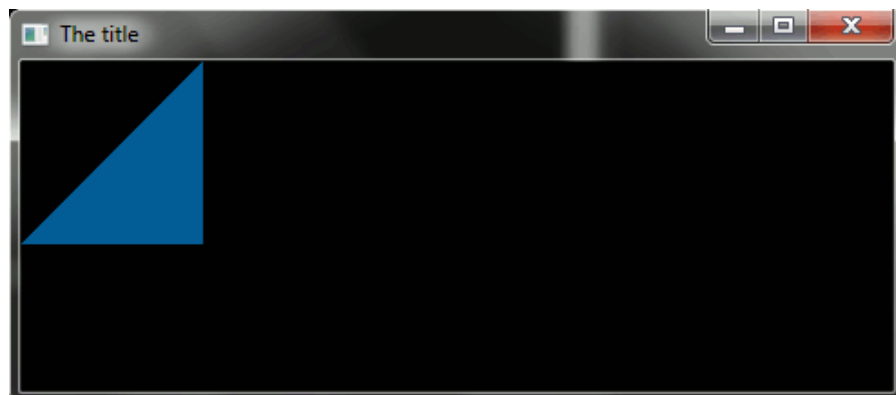
There is no support for a concave shape in SFML. However we can still draw concave shapes by creating multiple convex ones and rendering them in the correct places. If triangles are used for the job, the method is called **Polygon triangulation**.

Shape transformation

We know how to draw shapes on the screen now and that's great. However, no matter how many of them we draw, they all seem to go on the top-left side of the screen. This means that we need to change the position of the shapes. This can be done with the help of a function called `Shape::setPosition()`. There are also functions for the `Shape::setRotation()` rotation and the `Shape::setScale()` scale. Actually, those functions are all part of `sf::Transformable`, which the Shape class derives from. As with most things in SFML, using these functions is extremely easy and intuitive:

```
sf::RectangleShape rect(sf::Vector2f(50, 50));
rect.setFillColor(sf::Color::Red);
rect.setPosition(sf::Vector2f(50, 50));
rect.setRotation(30);
rect.setScale(sf::Vector2f(2, 1));
```

Don't forget to render the shape after you get it initialized. Here is the result:
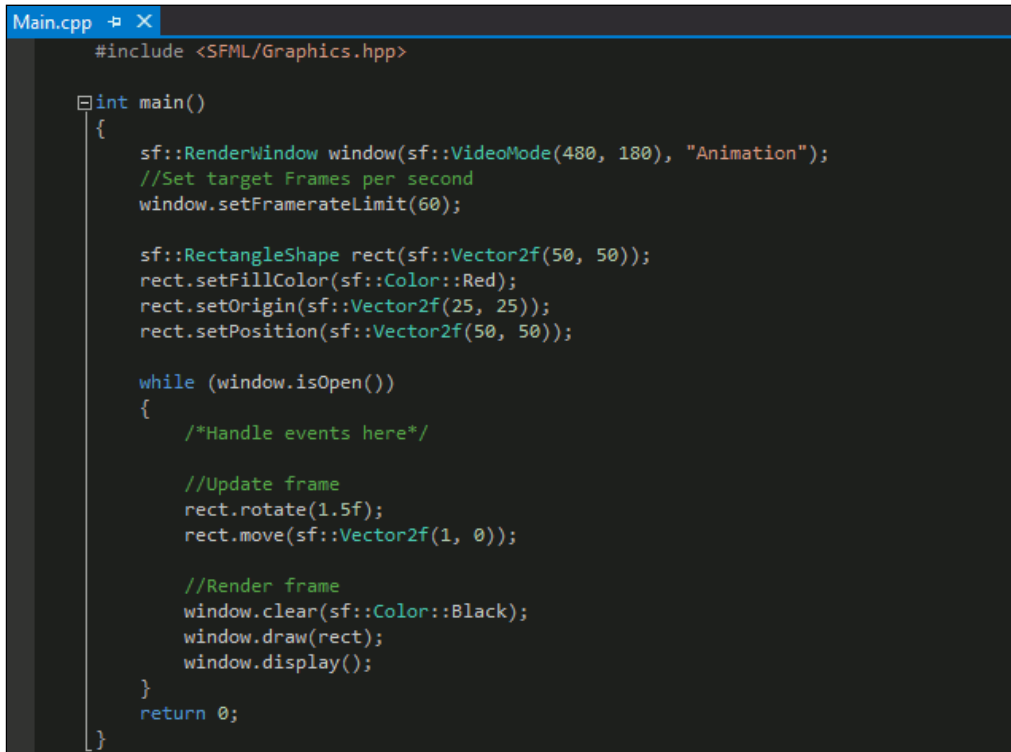
Note that we are creating a rectangle, which is actually a square, with a width and height of 50 pixels. However, we are scaling it at 2:1 and thus it is rendered longer than its original size. The next thing which we need to mention is that the rectangle is slightly tilted, which is expected, as we are rotating the rectangle by 30 degrees. In this example, we are setting the position directly to (50, 50). However there is an alternative method to move transformable objects and it's by calling `Transformable::move()` and passing a vector, which indicates how much we want to move the object from its current position.

Everything that we have created so far has been mostly static, so now let's add a little life to our objects. For that, we need to use the update frame, which we haven't been able to utilize yet. That's the part of the frame just before we start the render frame. Remember, typically a game frame (loop cycle) goes like this:

- Handle input
- The update frame
- The render frame

It is important to update objects before we render them, otherwise their current state will not be rendered correctly—they will be rendered with their state from the last frame.

The next example shows how we use a combination of translation and rotation to create a simple animation:

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

int main()
{
    sf::RenderWindow window(sf::VideoMode(480, 180), "Animation");
    //Set target Frames per second
    window.setFramerateLimit(60);

    sf::RectangleShape rect(sf::Vector2f(50, 50));
    rect.setFillColor(sf::Color::Red);
    rect.setOrigin(sf::Vector2f(25, 25));
    rect.setPosition(sf::Vector2f(50, 50));

    while (window.isOpen())
    {
        /*Handle events here*/

        //Update frame
        rect.rotate(1.5f);
        rect.move(sf::Vector2f(1, 0));

        //Render frame
        window.clear(sf::Color::Black);
        window.draw(rect);
        window.display();
    }
    return 0;
}
```

A couple of lessons can be taken from this example. The first one is how and where to set the framerate limit—just after the initialization of the window. This will limit our game logic somewhere close to 60 frames a seconds. Keep in mind that this controls the upper limit of the framerate. If the frames start taking more than 1/60 seconds to complete (handle events, update objects, and render), then the framerate will drop below 60. However, with our simple code, that is extremely unlikely.

You've probably noticed a new function called `RectangleShape::setOrigin()`. The origin of an object determines how it should be rendered on the screen. It serves as a center point for the translation, rotation, and scale for the object. In the preceding example, we have a square with size 50 x 50. The center of that square is (25, 25), so we need to set that as the origin of the object. Otherwise, the object will start rotating around its default origin—(0, 0). One last thing to note about the origin is that it's part of the `Transformable` class and so all of its derived classes have access to it.

As far as our animation goes, the process is quite simple. In every frame, we rotate the square by 1.5 degrees and move it by 1 pixel to the right. By setting the framerate to 60 FPS, we can estimate that, after each second, the square will rotate by approximately 90 degrees (1.5 x 60) and move by 60 pixels to the right (1p x 60). However, performing the game logic in such a way (relying on the framerate) is extremely unreliable and dangerous. We will explore how to manage time while performing animations and game logic in *Chapter 3*, *Animating sprites*.

Now, let's look at how we can control shapes in real time.

Controlling shapes

One way to make shapes move is by using event handling. We start moving the object when the player clicks on a key, and we can stop moving the object when that key is released. Here is the screenshot of a code example:

```cpp
bool moving = false;
while (window.isOpen())
{
    sf::Event ev;
    while (window.pollEvent(ev))
    {
        if (ev.type == sf::Event::EventType::KeyPressed &&
            ev.key.code == sf::Keyboard::Right)
            moving = true;
        if (ev.type == sf::Event::EventType::KeyReleased &&
            ev.key.code == sf::Keyboard::Right)
            moving = false;
    }
    //Update frame
    if (moving)
    {
        //Move the object
    }

    //Render the object
}

return 0;
}
```

The moving variable determines if we should move our object in the current frame. That variable's value is changed when we press or release the *Right Arrow Key*. However, that is a lot of code for essentially very little information—"is the *Right Arrow Key* currently pressed?" Fortunately for us, SFML provides a very simple interface to check the current state of the input devices (keyboard, mouse, and all joysticks).

Checking the state of a key is no harder than calling a single static function —`Keyboard::isKeyPressed()`. When we pass a key value as an argument, we get the status of whether that key is currently pressed. However, this function doesn't take into account the focus of the window. So imagine that a player minimizes the window and browses the internet. The function will still return `true` if the player presses the given key.

With this in mind, the given code can be very easily reworked to something much more pleasant:

```cpp
if (sf::Keyboard::isKeyPressed(sf::Keyboard::Key::Right))
{
    //Move the object
}
```

As you can see, we don't need to store any key state in this case—SFML does that for us.

We can check the states of other input devices in a similar way. The mouse has functions to get its position, the state of any of its buttons, as well as the setting of its position relative to the desktop (not any of the windows). To get the position, you can use `Mouse::getPosition()`. To set the position, you can use `Mouse::setPosition()`. Finally, to check whether a button is pressed, call `Mouse::isButtonPressed()`. All of these work out of focus as well.

Finally, there are joysticks. Since all functions are static, we need to specify which joystick we are looking for with its argument `Id`. The functions are as follows:

Function	Arguments	Description
`Joystick::isConnected()`	ID	This function checks whether the joystick with the given ID is connected
`Joystick::hasAxis()`	ID, axis	This function checks whether the joystick has the specified axis
`Joystick::getButtonCount()`	ID	This function gets the number of buttons on the joystick
`Joystick::getAxisPosition()`	ID, axis	This function gets the value of an axis in the range [0, 1]
`Joystick::isButtonPressed()`	ID, button	This function checks whether a button on a given joystick is pressed

Let's now discuss one final example, which combines a lot of topics from this chapter. We have taken a very simple game where the player plays as a green square, and he should reach the blue square without touching anything red. The following is a helper function, which helps us initialize similar `RectangleShape` objects easily and without much code repetition:

```cpp
void initShape(sf::RectangleShape& shape, sf::Vector2f const& pos, sf::Color const& color)
{
    shape.setFillColor(color);
    shape.setPosition(pos);
    shape.setOrigin(shape.getSize() * 0.5f); // The center of the rectangle
}
```

The `initShape()` function is quite straightforward—it takes a shape, vector, color, and assigns them to the `RectangleShape` object. The function also sets the origin point of the shape to its center.

The next step is to initialize the objects:

```cpp
sf::RenderWindow window(sf::VideoMode(480, 180), "Bad Squares");
//Set target Frames per second
window.setFramerateLimit(60);

sf::Vector2f startPos = sf::Vector2f(50, 50);
sf::RectangleShape playerRect(sf::Vector2f(50, 50));
initShape(playerRect, startPos, sf::Color::Green);
sf::RectangleShape targetRect(sf::Vector2f(50, 50));
initShape(targetRect, sf::Vector2f(400, 50), sf::Color::Blue);
sf::RectangleShape badRect(sf::Vector2f(50, 100));
initShape(badRect, sf::Vector2f(250, 50), sf::Color::Red);
```

The first thing we have to do is open a suitable window. Secondly, we need to set the framerate limit to the standard 60 frames per second. The next thing on the list is a `sf::Vector2f` variable, which we will use as a spawn point for the player. After we initialize the square of the player, we initialize the target, a blue square a bit further to the right in the world. The last shape is a square, which the player has to avoid. It stands somewhere in the middle.

The updated frame, where all the game logic happens, looks like this:

```
//Always moving right
playerRect.move(1, 0);
if (sf::Keyboard::isKeyPressed(sf::Keyboard::Key::Down))
    playerRect.move(0, 1);
if (sf::Keyboard::isKeyPressed(sf::Keyboard::Key::Up))
    playerRect.move(0, -1);

//Target reached. You win. Exit game
if (playerRect.getGlobalBounds().intersects(targetRect.getGlobalBounds()))
    window.close();
//Bad square intersect. You lose. Restart
if (playerRect.getGlobalBounds().intersects(badRect.getGlobalBounds()))
    playerRect.setPosition(startPos);
```

We can clearly see from the preceding code that the player always moves to the right—the player has no control over it. You can change that, so that the player has control of all four directions—out of personal preference. Currently, the only directions in which the player can move are *up* and *down* by using the arrow keys.

Apart from the input handling, we need to check whether the code has the logic for both win and lose conditions. We need a method to handle collision detection between these rectangles. Thankfully, all shapes in SFML have two functions called `Shape::getGlobalBounds()` and `Shape::getLocalBounds()`, which return `sf::FloatRect`, which represents the global or local bounds of the current shape. The bounding rectangle of a shape (sometimes called a **bounding box**) is the rectangle with the smallest possible surface area, which contains the whole shape. *Global* and *Local* refer to the transformation of the shape—if the shape is transformed in any way, the position, scale, and rotation are ignored in the local bounds; whereas in the global bounds, they are taken into consideration. Once we have the global bounds, we can use a function called `FloatRect::intersect()` that takes another `FloatRect` and returns if the two rectangles collide. Do not confuse `RectangleShape` with the `FloatRect` class though, they serve different purposes—the former is for drawing and the latter is a simple class for storing a rectangle's attributes (top, left, bottom, and right values).

The game logic itself is quite simple. If the player collides with the target square, the player wins (the player should exit the game.) If the player collides with a bad rectangle, the player loses (the player should restart the game.)

As you can imagine, there is nothing special about the render frame, you just have to draw all three shapes in any order you wish and run the code. The result will be similar to this:
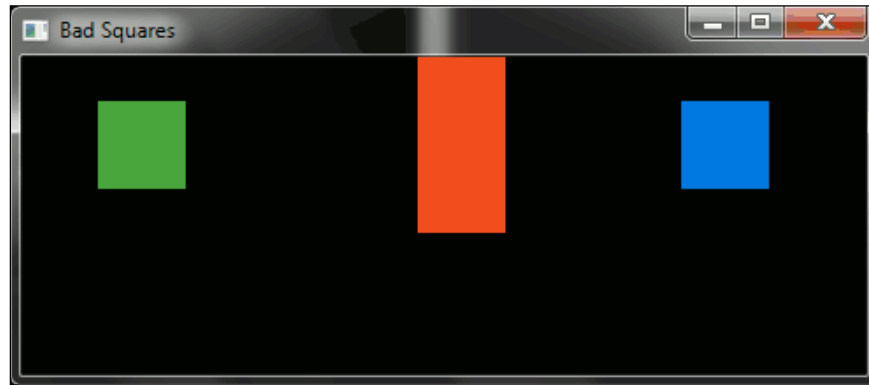
Feel free to change the code as much as you like and experiment with different settings (add more shapes and build on the game logic maybe?).

Summary

You should feel good about yourself after completing this chapter. This means that you are ready for everything that this book offers. This chapter builds the very foundations on which your game will eventually stand. Although the examples here are simple to understand, they aren't supposed to be copied and used as they are. They are there to help you understand how SFML is built, and how you can use that knowledge to your advantage.

In the next chapter, we will cover textures, sprites and resource management. Stick around—there are treats to be had.

2
Loading and Using Textures

Textures are important, both for 2D and 3D games. They allow us to map images onto objects. In this chapter, we will look into ways of loading textures into memory and mapping them onto shapes. The `Sprite` class will make an appearance, and we will see how it differs from the `Shape` class. Finally, we will see how to keep resources safe from destruction throughout the lifetime of the game.

In this chapter, we will cover:

- Loading textures
- Rendering shapes with textures
- What is a sprite?
- Managing resources

Loading textures

Textures are quite simple objects. A 2D texture is essentially an image that is typically stored in the **Graphics Processing Units' (GPU)** memory. SFML provides an `Image` and a `Texture` class to process and render images, respectively.

Images versus textures

The main difference between images and textures is their purpose-, manipulation and rendering, respectively. The `Image` class handles image loading, saving, and pixel manipulation, whereas the `Texture` class is used for rendering. These two classes differ in their behavior, but that doesn't change the fact that they hold the same data—an array of pixels. As such, SFML provides simple ways of creating one from the other. For example, if we want to load an image from a file and modify it a bit, we can then create a `Texture` from that `Image`. However, if we then want to change the `Texture` object again, we have to download it to `Image`, process it however we like, and only then upload it again as `Texture`. This whole process can be expensive, and we should avoid doing it in critical sections of the code.

Creating images

We will explore ways of creating and loading images before we jump into textures. Many of the functions we see in the `Image` class also exist in `Texture`. The following code demonstrates how to create a 50 x 50 image and fill it with red color:

```
sf::Image image;
image.create(50, 50, sf::Color::Red);
```

The first two arguments of the `Image::create()` function represent the width and height of the image, and the last argument is the color fill for the image. By default, the color is set to black with alpha 255.

Images can also be created by passing an array of pixels directly. The array must hold elements of the type `Uint8`, which is a single byte of memory. Since `Image::create()` requires the colors to be in an RGBA format, we need to make sure that the array holds exactly 4 bytes for each color (1 byte per color component). Every consecutive 4 bytes represents a pixel of the image grid, which is laid out as rows by columns. Here is an example of how this can be done:

```
const unsigned int kWidth = 5, kHeight = 5;

//Array size = width * height * 4(RGBA)
sf::Uint8 pixels[kWidth * kHeight * 4] =
{
    255, 255, 255, 255, //White
    0, 0, 0, 255,       //Black
    255, 0, 0, 255,     //Red
    128, 128, 128, 255, //Gray

    //...all other pixels
};

sf::Image image;
image.create(kWidth, kHeight, pixels);
```

The preceding code demonstrates how to create a very small image (5 x 5). Note how the array does not contain elements of the type `sf::Color`, but rather contains the RGBA components of a color. However, the method of specifying colors in both cases is the same—passing a byte (`Uint8`) for each of the four components. This means that each four bytes (`4xUint8`) represents a single pixel of the image.

Images can also be loaded from a file which is extremely straightforward and is shown as follows:

```
sf::Image image;
image.loadFromFile("myImage.png");
```

The code assumes that there is an image in the working directory of the program, named `myImage.png`. Loading from a file is an effective way to create images if you want to use an image which is already available on the machine. SFML supports the following file formats: `bmp`, `png`, `tga`, `gif`, `psd`, `hdr`, `pic`, and `jpg` (progressive JPEG is not supported.) If we try to load an image with a different file format, or the given file does not exist, `Image::loadFromFile()` returns `false` and prints a message in the console:

```
Failed to load image "myImage.png". Reason : Unable to open file
```

When an image fails to load (for any reason), we need to take an action (inform the user, terminate the program, and so on). The following code exits the `main()` function if the image is not loaded correctly. This is a safe way to prevent any future bugs from occurring in the code.

```
sf::Image image;
if (!image.loadFromFile("myImage.png"))
    return -1;
```

If `Image::loadFromFile()` fails, the image is left unchanged.

It is highly recommended that we use lossless file formats, such as PNG, to create a high-quality experience for the user. Lossy formats such as JPEG will degrade the quality of the image while proving greater compression. We should only consider using JPEGs when the program size matters, or we are not concerned with the quality of the images. For example, we can also choose to use JPEGs for large images (such as backgrounds) with a carefully selected compression level that will save us a lot of space, while providing minimal quality degradation.

The `Image` class provides very useful methods to manipulate an image. Functions such as `Image::getPixel()` and `Image::setPixel()` allow us to change individual pixels. If we want to read all pixels from an image, there is the `Image::getPixelPtr()` function, which returns the beginning of the pixel array. This array is in the same format as the array we used to create the image in our second example. Apart from that, `Image::flipHorizontally()` and `Image::flipVertically()` transform the whole image by flipping its pixels in a particular direction. Finally, we can save the image to a file by calling `Image::saveToFile()` and passing a filename. The supported formats for saving an image are: `bmp`, `png`, `tga`, and `jpg`.

Now that we know how to create and manipulate images, let's see how to create textures out of them.

Creating textures

The `Texture` class shares a lot of methods with the `Image` class. For example, we can load textures from files in the same way we did with images:

```
sf::Texture texture;
if (!texture.loadFromFile("myTexture.png"))
    return -1;
```

`Texture::loadFromFile()` offers a bit more functionality though. When loading a texture from a file, we can opt to load only a small section of the image. There is an optional argument in `Texture::loadFromFile()`, which allows us to do that. In the following code, we will only load a 32 x 32 pixel square from the original image, beginning at the top-left corner:

```cpp
sf::Texture texture;
if (!texture.loadFromFile("myTexture.png", sf::IntRect(0, 0, 32, 32)))
    return -1;
```

The preceding code loads the whole image and only then creates a texture from the specified rectangle. This method becomes inefficient if we want to use the same image multiple times. The alternative is to load the image file once (in `Image`), and use that to create our textures. Here's how to create textures directly from an image:

```cpp
sf::Image image;
image.create(50, 50, sf::Color::Red);

sf::Texture texture;
texture.loadFromImage(image);
```

Like the `Image` class, textures can be created by calling `Texture::create()`. However, we have to be careful with the dimensions we provide for the texture, since all existing graphics cards have limits for the texture size. Fortunately, there is a static function— `Texture::getMaximumSize()`, which returns `integer` with the maximum possible size of a texture on the machine. This can be used as an indication that we need to use lower resolution textures. If we want maximum compatibility from our program, we have to take the extra effort to use only textures, which have sizes equal to the power of 2 (32, 64, 256, 1024, and so on).

We can also make an image out of a `Texture` object by calling `Texture::copyToImage()`. We have to be careful where we call this though, since it is a slow operation which copies data from the GPU to the RAM.

Everything looks good so far. However, one problem still remains—we haven't seen how to display textures on the screen yet. Let's fix this immediately.

Rendering shapes with textures

Let's start with something important, which isn't always completely obvious to everyone. Textures cannot be rendered on their own. They need a surface to be mapped to and then that surface can be rendered. Textures, as we mentioned in the beginning of the chapter, are just a collection of pixels, which cannot be rendered directly on the screen without having some sort of reference (such as a position, rotation, and so on). However, in SFML, there are renderable classes which can use a texture for their surface. In fact, we used one of those classes in the previous chapter—the shape.

Apart from a fill color and an outline color, every `Shape` object can have a texture as well. We can apply a texture to a shape by calling `Shape::setTexture()` and passing a pointer to a texture. The last thing we need to do is render the shape in a window:

```cpp
sf::Texture texture;
texture.loadFromFile("myTexture.png");

sf::RectangleShape rectShape(sf::Vector2f(300, 150));
rectShape.setTexture(&texture);

while (window.isOpen())
{
    //Handle events

    window.clear(sf::Color::Black);
    window.draw(rectShape);
    window.display();
}
```

The first important thing that stands out is that `Shape::setTexture()` takes a pointer rather than a reference. That's why we pass the address of the texture with `&texture`. The shape then stores that pointer locally and uses it when it needs to be rendered. This means that the address, which we pass to the function, has to hold a valid texture throughout the lifetime of the shape. Moving the texture in memory or destroying it will lead to a dangling pointer inside the `Shape` object, resulting in an undefined behavior. That is why we always need to make sure that a texture does not get out of the scope of the object which uses it. We will explore resource management techniques in the last section of this chapter.

When we place a texture on `RectangleShape`, it tries to fit into the specified rectangle by scaling itself up or down. For our example, if the texture has a width of 200, height of 200, and the rectangle is of the size—300 as width and 150 as height, then the texture will appear stretched on the *x* axis and squeezed on the *y* axis:
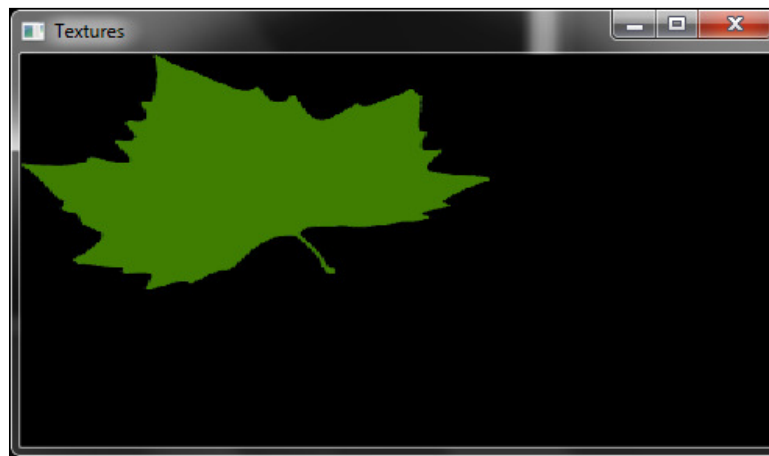
To set the shape to `RectangleShape` with the exact size of the texture, we can use a function from the `Texture` object—`Texture::getSize()`:

```cpp
sf::Vector2u textureSize = texture.getSize();
float rectWidth = static_cast<float>(textureSize.x);
float rectHeight = static_cast<float>(textureSize.y);
sf::RectangleShape rectShape(sf::Vector2f(rectWidth, rectHeight));
rectShape.setTexture(&texture);
```

Here, the result is undistorted:
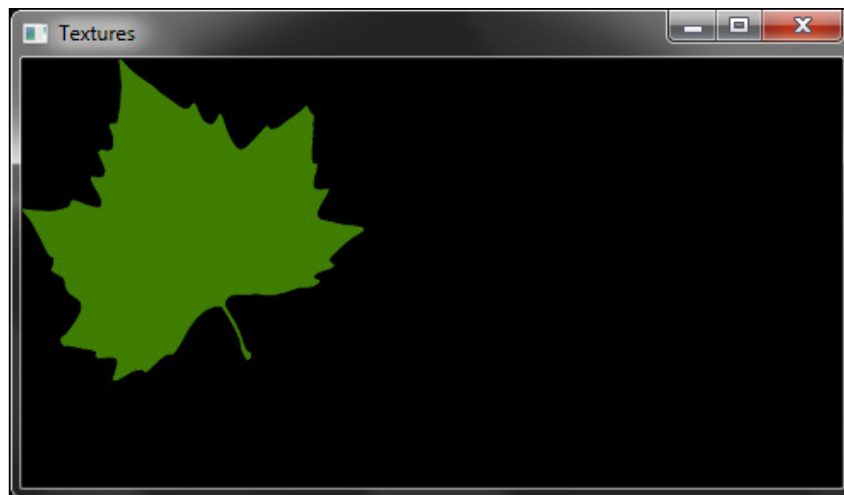
Textures can also be mapped to `CircleShape` and `ConvexShape` objects.
Here is an example of how we can limit the amount of texture that is shown
by using `ConvexShape`:

```cpp
sf::ConvexShape shape(5); //Convex shape has 5 points
shape.setPoint(0, sf::Vector2f(0, 0));
shape.setPoint(1, sf::Vector2f(200, 0));
shape.setPoint(2, sf::Vector2f(180, 120));
shape.setPoint(3, sf::Vector2f(100, 200));
shape.setPoint(4, sf::Vector2f(20, 120));
shape.setTexture(&texture);
shape.setOutlineThickness(2);
shape.setOutlineColor(sf::Color::Red);
shape.move(20, 20); //Move it, so the outline is clearly visible
```

We will create a simple polygon with five vertices and assign it a texture. For clarity,
we will also show the outline and move it a bit from the edges of the window; here is
the result:

Textures can also be repeated multiple times on a surface. Let's say that we want
to create the following surface from this tile:
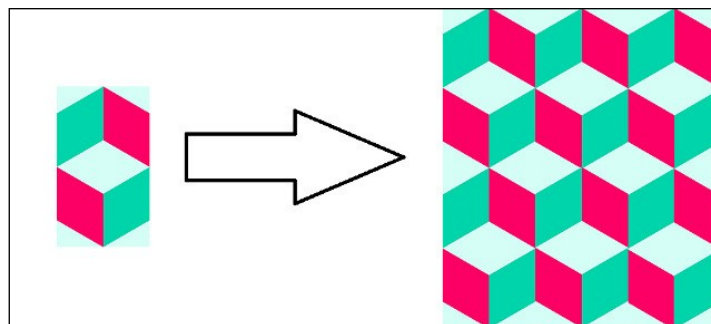
Since the tile on the left is completely seamless in all directions, we can place many of them side by side and create a bigger texture on the surface. One way of achieving this is by creating the larger image in an image editor, such as Microsoft Paint, GIMP, or Photoshop. However, that image will require more memory, which we do not necessarily want to give up. There is an alternative way where we load only the tile in a GPU memory and use it as a repeated texture over a given surface.

Our tile has dimensions 128;221, and we are replicating the tile three times on the *x* axis and two times on the *y* axis; meaning that we end up with a surface the size of 384;442. For such a shape, it is only logical to use the `RectangleShape` class. Here is our setup:

```
sf::Texture texture;
texture.loadFromFile("tile.png");

sf::RectangleShape rectShape(sf::Vector2f(128 * 3, 221 * 2));

rectShape.setTexture(&texture);
```

If we try to render the shape in its current shape, the result does not seem promising—the texture is just stretched on the whole surface of the rectangle. We need to configure the texture a bit more for it to work as we like. First of all, there is a function inside the `Texture` class—`Texture::setRepeated()`, which takes `bool` and marks the texture as repeatable if it is called with true. However, this is not enough; there is one more step.

When we map textures to surfaces, we typically have to specify texture coordinates for each vertex of the surface. In SFML, this is done automatically for the `Shape` class. If we were using the OpenGL API to render a square with a texture on top of it, we would have to specify the texture coordinates in a normalized format (0…1; 0…1). SFML does not use the normalized approach; rather it uses pixel space coordinates ([0… width −1], [0… height −1]). Here is a diagram to demonstrate how texture coordinates are mapped to a surface, which is then rendered on a screen (the gray rectangle):
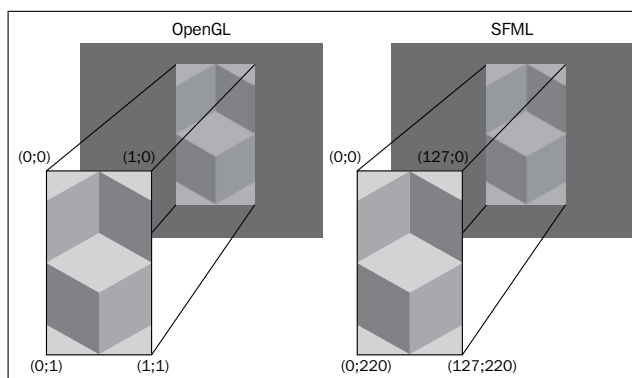
We saw what happens when we make the surface (or the shape) bigger—it just stretches the texture. We need to change the texture coordinates to repeat the texture multiple times on that surface. This is done by making the texture rectangle (all four texture coordinates in one structure) larger than the texture itself.

With that in mind, let's see how our code setup will change:

```cpp
sf::Texture texture;
texture.loadFromFile("tile.png");
//Set the texture in repeat mode
texture.setRepeated(true);

sf::RectangleShape rectShape(sf::Vector2f(128 * 3, 221 * 2));
//Bigger texture rectangle than the size of the texture
rectShape.setTextureRect(sf::IntRect(0, 0, 128 * 3, 221 * 2));

rectShape.setTexture(&texture);
```

The result is exactly what we expected:
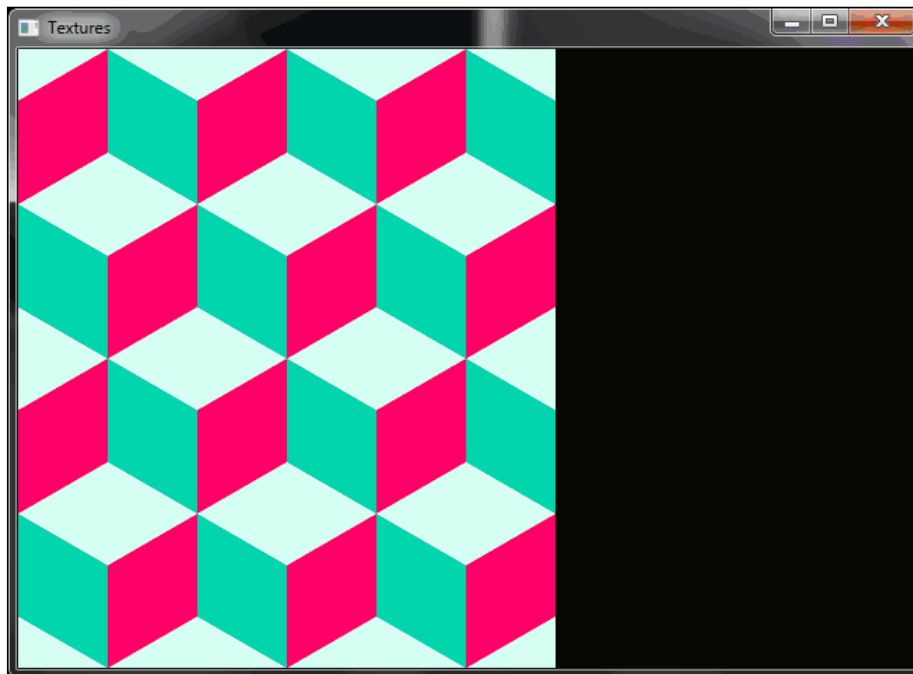
To clarify texture coordinates a bit further, the following diagram demonstrates how the default texture rectangle (that fits the texture perfectly) is mapped to a bigger surface, and what is the result when the texture rectangle is larger than the texture:
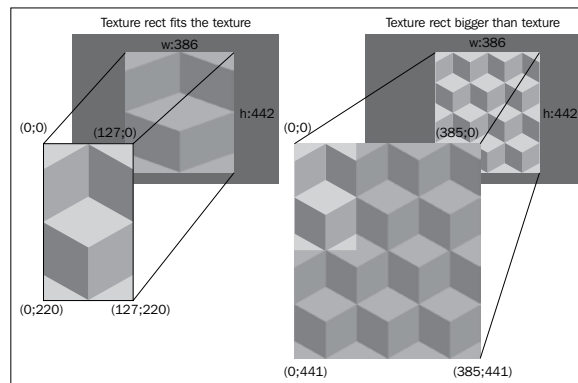
Apart from `Texture::setRepeat()`, there is one other property that changes the way a texture is rendered—the smooth filter, which is controlled by `Texture::setSmooth()`. If we only use texture on surfaces with their original size (pixel perfect), then we won't need this functionality. The function itself enables a smooth filter on the texture, which makes its edge pixels less visible. The effect is mostly visible when a texel (a pixel of a texture) cannot be directly mapped to a pixel on the screen (scaling, offsetting by noninteger values, and so on). For pixel perfect graphics, we would want to avoid using the smooth filter, since it will smudge the texture.

Here is how we can set the smooth filter on:

```cpp
sf::Texture texture;
texture.loadFromFile("myTexture.png");
texture.setSmooth(true);
```

The following is an example with two sets of textures, one that uses the filter and one that doesn't:
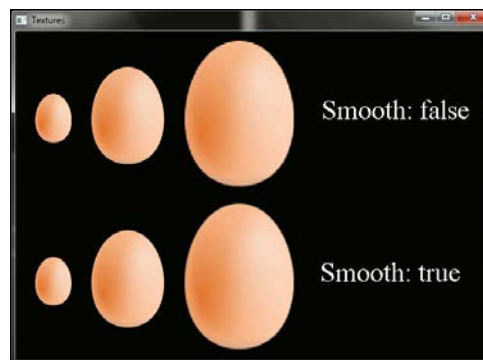
Now that we know how to play with textures, let's talk about sprites.

What is a sprite?

You have most likely heard the term *sprite* before. In its very essence, a sprite is a surface with a texture on it. "But wait…" I hear you saying, "Didn't we just cover that very same thing?" And my answer is "Yes, but sprites are a bit different." Apart from the `Shape` class, SFML provides a `Sprite` class as well, and that brings us to the question: "How are they different?"

Shapes versus sprites

Probably the most important difference is that a sprite is always rendered as a textured rectangle. We can use shapes without textures (just by setting a fill and outline colors), whereas sprites strictly require a texture to be attached to them. Since sprites are rendered as rectangles, we cannot cut parts of a texture like we did with `ConvexShape`.

Apart from that, the `Sprite` class has a `Sprite::setColor()` function, similar to `Shape::setFillColor()`. The effect of both the functions is the same as long as the shape has a texture attached to it as well—the texture gets its color multiplied by the selected color. The only difference is that if the sprite doesn't have a texture, nothing will be rendered, whereas the shape is rendered with the specified color.

Furthermore, the sprite dimensions are controlled by its texture. In `RectangleShape`, we set the size of the rectangle that we want to create. With sprite, there is no shape to define, just the texture. If we want the sprite to appear bigger or smaller, we have to change the scale of the `Sprite` object.

Here comes a good question—why would we want to use a sprite instead of a shape? It seems that the sprite is just a shape with reduced capabilities. This is exactly why we would want to use it—its simplicity.

Consider the following code:

```cpp
//Create a shape with a texture
sf::RectangleShape rectShape(sf::Vector2f(100, 100));
rectShape.setTexture(&texture);

//Create a sprite
sf::Sprite sp(texture);
```

We can quite clearly see that creating a sprite is a far simpler process. In fact, that is the main purpose of the `Sprite` class in SFML—to render a texture on the screen as quickly and painlessly as possible.

Next, let's see what exactly we can do with a sprite.

Transformables and drawables

The `Sprite` class is derived from two classes—`Transformable` and `Drawable`. The `Drawable` class is essentially an interface, which holds a single abstract method —`Drawable::draw()`. All children have to implement this method to be able to draw themselves onto a canvas (such as a `RenderWindow`). The `Transformable` class holds a position, rotation, scale and origin, as well as `accessor/mutator` functions for these fields. Some of them include: `Transformable::setPosition()`, `Transformable::getPosition()`, `Transformable::move()`, and so on.

These functions might sound familiar. This is because we have encountered them before in the `Shape` class. In fact, the `Shape` class inherits from `Drawable` and `Transformable`. This means that we can manipulate a sprite in the same way we do a Shape, and we can draw sprites by calling `RenderWindow::draw()`. In fact, if we look closely at `RenderWindow::draw()`, we will see that it takes a `Drawable` argument rather than a shape or sprite, which means that every class which derives from `Drawable`, can be passed to a window to be drawn.

We can also create our own classes that inherit from `Transformable` or/and `Drawable`. If we want to create an optimized circle sprite for example, we can create `CircleSprite` and implement a `draw` method for it. Then, it will be as easy as passing a `CircleSprite` object into a `draw` call to `RenderWindow`.

Final facts on sprites

`Sprite` has a few more things to offer. All features of `Texture` (smoothing and repeating) work on sprite as well. To repeat a texture, we have to change the texture rectangle (holding the texture coordinates) as we did with a Shape. For that purpose, the same function exists in the `Sprite` class—`Sprite::setTextureRect()`.

Apart from this, `Sprite::getLocalBounds()` and `Sprite::getGlobalBounds()` make an appearance here as well. They both calculate the **Axis-Aligned Bounding Box** (**AABB**) of the sprite. The local bounds are local to the sprite—they do not take into consideration the transformations. On the other hand, the global bounds transform the sprite with the position, scale, rotation and origin, and then capture the rectangle. As we did with the shape, these can be used for basic collision detection since `FloatRect` (returned by the bounds functions) has a `FloatRect::intersects()` function in it.

This is it for the `Sprite` class. We will talk about a very important topic next—resource management.

Managing resources

Whether we are making a game or a multimedia application, it is important to manage assets correctly and efficiently. Making sure that assets do not get accidentally destroyed, or that we do not load the same texture twice is important to maintain a solid and efficient code base. In this section, we will talk about building the groundwork for a resource manager, which you can use in your applications.

Let's start with a simple fact—objects on the stack are destroyed when they get out of scope. In some languages, all classes are allocated on the heap (where memory is managed automatically by garbage collection) and this is not a problem. However, in C++, if we want to keep objects on the stack, which makes memory much easier to manage, we have to make sure that they stay alive as long as they are used. This is an example where a texture will get destroyed as soon as the function exits:

```
sf::Sprite createSprite(std::string const& filename)
{
    sf::Texture texture;
    texture.loadFromFile(filename);

    //This is bad. As soon as the function returns
    //the texture will be destroyed
    return sf::Sprite(texture);
}
```

If we draw the sprite, we will see that only a white rectangle is drawn and the texture is nowhere to be seen.

To manage assets' lifecycle correctly throughout the runtime of the program, it is very useful to have a dedicated manager. For this very reason, let's create a class `AssetManager`, which will load, hold, and destroy all assets in our program. Here is how the header file looks like:

```cpp
#ifndef ASSET_MANAGER_H
#define ASSET_MANAGER_H

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

class AssetManager
{
public:
    AssetManager();

    static sf::Texture& GetTexture(std::string const& filename);

private:
    std::map<std::string, sf::Texture> m_Textures;

    //AssetManager is a singleton, so only one instance can exist at a time
    //sInstacne holds a static pointer to the single manager instance
    static AssetManager* sInstance;
};

#endif
```

The `AssetManager` class is singleton (only one instance is allowed to exist) and that is why it has a static pointer to itself. Actually, all static functions of that class will use this pointer as a way of referencing the instance of `AssetManager`, which will be created just after we initialize the window and will be destroyed when the program exits. Creating the asset manager in such a way is useful because we can call `AssetManager::GetTexture()` from anywhere in the program without having a reference to the `AssetManager` object. Doing it in such a way with small examples might seem pointless, but with larger projects it can save a lot of headaches of passing references around.

Apart from the pointer, the class holds a map of textures and a way of getting elements from that map with the `AssetManager::GetTexture()` function. A map is a collection of values and unique keys—each key has exactly one value associated with it. In our case, we have `string` keys and `Texture` values. The keys hold the filenames of the textures, and the values hold the `Texture` objects. Doing it this way, we can easily check whether a filename exists in `map` and add it if it doesn't.

Now, let's look at the constructor inside the source file:

```cpp
AssetManager.cpp
#include "AssetManager.h"
#include <assert.h>

AssetManager* AssetManager::sInstance = nullptr;

AssetManager::AssetManager()
{
    //Only allow one AssetManager to exist
    //Otherwise throw an exception
    assert(sInstance == nullptr);
    sInstance = this;
}
```

The first line after the `#include` statements initializes the static pointer `sInstance` to `nullptr` (which is null or 0). It is a good practice to set a pointer to `nullptr` just after we declare it as we can check later on if it is valid or not. This is exactly what we do in the constructor. We call the `assert` macro which checks whether an expression is `true`. If it is, nothing happens. However if it's `false`, the macro calls `abort()`, which terminates the program. This check prevents more than one instance of the class to be created, which is exactly what we want. After we've made sure that this is the only instance, we will set the static pointer to the `this` instance.

Here is the `AssetManager::GetTexture()` implementation:
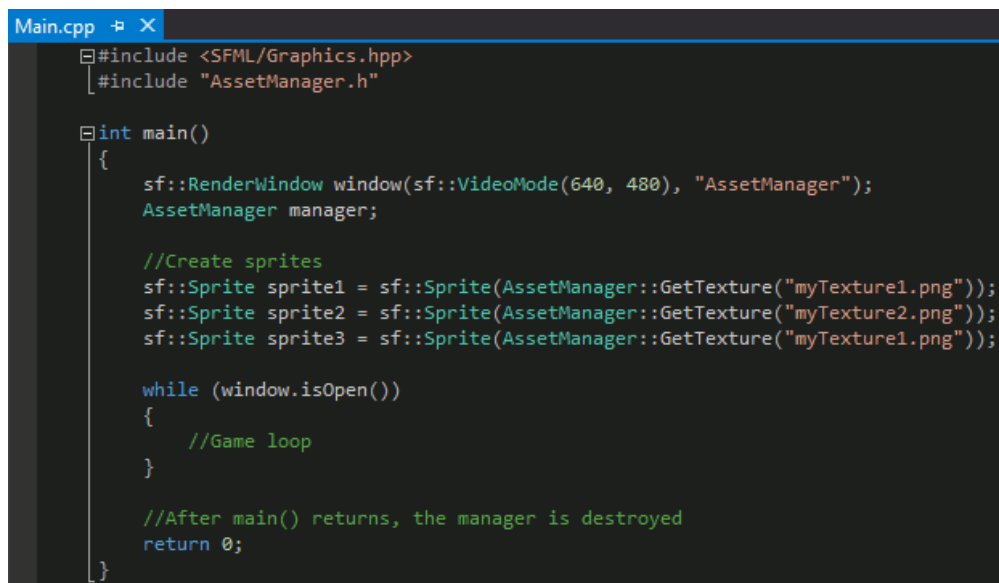
```cpp
sf::Texture& AssetManager::GetTexture(std::string const& filename)
{
    auto& texMap = sInstance->m_Textures;

    //See if the texture is already loaded
    auto pairFound = texMap.find(filename);
    //If yes, return the texture
    if (pairFound != texMap.end())
    {
        return pairFound->second;
    }
    else //Else, load the texture and return it
    {
        //Create an element in the texture map
        auto& texture = texMap[filename];
        texture.loadFromFile(filename);
        return texture;
    }
}
```

Note that this function is a static one, which means that we have to get the map from the instance through the static instance pointer, `sInstance`. After that, we will check whether the requested texture was already loaded by calling `map<>::find()`. The function returns an iterator pointing at the pair found. If no pair is found, it points to `map<>::end()`. If the filename is found, we will return the `texture` object in the pair (the second element). If the filename is not found, we will create a slot in the map for that texture and load the texture from the filename argument.

This is pretty much everything that `AssetManager` does; it just holds a map of textures and has an interface to access them. As we go on further in this book, we will add more assets in the manager, such as fonts, shaders, music, and sounds.

And finally, here is how we can initialize and use `AssetManager`:

```cpp
#include <SFML/Graphics.hpp>
#include "AssetManager.h"

int main()
{
    sf::RenderWindow window(sf::VideoMode(640, 480), "AssetManager");
    AssetManager manager;

    //Create sprites
    sf::Sprite sprite1 = sf::Sprite(AssetManager::GetTexture("myTexture1.png"));
    sf::Sprite sprite2 = sf::Sprite(AssetManager::GetTexture("myTexture2.png"));
    sf::Sprite sprite3 = sf::Sprite(AssetManager::GetTexture("myTexture1.png"));

    while (window.isOpen())
    {
        //Game loop
    }

    //After main() returns, the manager is destroyed
    return 0;
}
```

Loading and caching the textures is as easy as calling one method. In the preceding example, the first two calls to `AssetManager::GetTexture()` load and cache the new textures, but in the last call (`sprite3`), the manager only returns the cached texture, saving us the time and memory involved to load it again.

Summary

In this chapter, we learned what SFML has to offer in terms of textures and images. We saw how to correctly create or load a texture as well as multiple ways to render it on the screen. Although textures have more applications than what we just saw, this is a solid start.

In the next chapter, we talk about action, animation, and timing. You don't want to miss that!

3
Animating Sprites

Animation is quite important if we want to create the impression that an object is doing something naturally. Take a sprite of a campfire for example. If there is only one image of a flame, it seems like the fire is not burning. However, if you alternate between multiple images, it will create the illusion that there is something going on there. This is what we are going to discuss in this chapter. Before we do that, we need to explore the world of time-based simulations, since that is required for our animations to function properly.

In this chapter, we will cover the following topics:

- Capturing time
- Animating sprites
- Building an animator

Capturing time

Time is important. Even in computing we have to deal with it in most applications. However, let's say that Timmy (a friend of ours) does not believe that time is of any importance and, one day, he sits down and builds a multiplayer racing game where cars move by exactly one pixel in every frame. Happy with the result on his machine, Timmy sends the program to his friend Jimmy, who has just bought the latest Super Ultra X CPU for his machine. They get into the race without any problems but, as soon as the green light hits, Jimmy rushes ahead of Timmy, leaving him behind, all dusted and confused. Later on, Timmy realizes that Jimmy's machine executes the code a lot faster than his own machine and therefore, his car was slower. Timmy never overlooked the frame time ever again

Even though the preceding story demonstrates an oversimplified example, it reveals the greatest flaw in ignoring time in any simulation—frames are executed at different speeds on different machines and thus the simulation appears differently. Here is how Jimmy's code looked like before he fixed it:

```
//Car speed = 1 pixel per frame
const float carSpeed = 1.f;

//Advance the car
carSprite.move(carSpeed, 0);
```

The preceding code executes each frame, and thus it is very much dependent on the CPU and GPU speed. If on one machine this code runs at the speed of 30 frames per second, resulting in 30 pixels travelled for one second, then on another machine, that code can run at the speed of 60 frames per second, doubling the distance travelled for the same amount of time. Not to mention that, almost always, frames take a different amount of time to run on the same machine.

Clearly, we have a problem—game logic runs at different speeds on different systems. Fortunately, the solution is quite simple—use elapsed time between frames to update the game logic. Here is the same preceding example using movement, which depends on time:

```
//Seconds elapsed since last frame
float deltaTime;

/* Calculate deltaTime here */

//Change the car speed to pixels per second - 30 is reasonable
const float carSpeed = 30.f;

//Advance the car
carSprite.move(carSpeed * deltaTime, 0);
```

Not much has changed, has it? We have one extra variable (`deltaTime`), which holds the duration of the last frame (in seconds). When we pass the variable to the `Sprite::move()` method, we are effectively saying "I want to move this car by `carSpeed` pixels per second in the horizontal direction." It's very simple yet extremely effective.

However, here is the million dollar question—how do we capture that elapsed time?

sf::Time and sf::Clock

As always, SFML is our friend, as it has two classes which work very well with time. `Time` is a class which holds a duration. This means that it doesn't tell us anything about the current time of day or the time elapsed since the program started, it just has a variable which holds a time amount. It could be five microseconds or it can be 10 months—anything that represents a period of time.

We can use the functions `sf::seconds()`, `sf::milliseconds()`, and `sf::microseconds()` to construct a time object from seconds, milliseconds, and microseconds, respectively. Once we have that object, we can use arithmetic operations (add, subtract, and compare) on it. We can later convert the `Time` object into seconds, milliseconds, and microseconds by calling the functions `Time::asSeconds()`, `Time::asMilliseconds()`, and `Time::asMicroseconds()`. Here is an example of how the `Time` class works:

```cpp
sf::Time time = sf::seconds(5) + sf::milliseconds(100);
if (time > sf::seconds(5.09))
    std::cout << "It works";
```

The `Time` class is handy for storing time, but it doesn't provide us with a way of capturing it. The `Clock` class provides an interface to measure elapsed time by using the OS clock. It is simple to use as well.

```cpp
sf::Clock clock;

//Run heavy CPU code

sf::Time timePassed = clock.getElapsedTime();
```

Apart from `Clock::getElapsedTime()`, the `Clock` class has another function inside it and that is `Clock::restart()`, which returns the elapsed time and restarts the clock at the same time.

With that in mind, it should be obvious how to measure the frame time. We initialize a `Clock` variable outside the game loop, and at the beginning of the frame, we get the elapsed time and restart the clock. This gives us the amount of time that the previous frame took, so that we can use it to advance out objects. The amount of time between the beginning of the last frame and the beginning of the current frame is typically called `deltaTime` (or `dt` in short). Here is what the code looks like:

```cpp
sf::Time deltaTime;
sf::Clock clock;
while (window.isOpen())
{
    //Returns the elapsed time and restarts the clock
    deltaTime = clock.restart();
    float dtAsSeconds = deltaTime.asSeconds(); //Delta time as seconds

    //Handle input

    //Update frame

    //Render frame
}
```

Now that we know how to capture time, we should make sure to use it wherever the game logic requires it—in any sort of movement, animation, time-based events (to destroy an object after *N* seconds), and so on. Since we know the delta time between frames, we can also accumulate time in a different variable to measure time outside the frame structure. Here is an example where we want to close the window 5 seconds after opening it:

```cpp
sf::Time elapsedTime;
sf::Clock clock;
while (window.isOpen())
{
    sf::Time deltaTime = clock.restart();
    //Accumulate time with each frame
    elapsedTime += deltaTime;

    if (elapsedTime > sf::seconds(5))
        window.close();
}
```

Since we don't call `Window::pollEvent()` anywhere in the loop, the window won't process any events from the user (focus, move, resize, and so on). However, this won't stop it from executing the logic it is supposed to—closing the window after a period of time.

With time on our side, we can now safely move on to animation.

Sprites in action

Animation exists in many forms. The traditional approach to animation is drawing a sequence of images which differ slightly from each other, and showing them on a screen one after the other. Even though this approach is still widely used, there are more elegant alternatives. For example, drawing (or modelling in 3D) only the limbs of a character and then animating how they move relative to time is a technique that saves a lot of time for artists. It also creates smoother results because not every frame of the animation has to be redrawn. In this book, we are going to explore only the traditional approach, since it is the simpler solution for programmers, and in many cases it is enough to bring life to any sprite.

The setup

As we established earlier, the traditional approach involves a set of images that need to change over time. For our example, we will use a crystal, which rotates around its centre. Typically, an animation is kept in a single file (a **sprite sheet**), where each frame of the animation is stored, and in most cases, each frame is the same size—the size of the object. In our example, the sprite is, 32 x 32 pixels and has eight frames, which play for one second. Here is what the sprite sheet looks like:
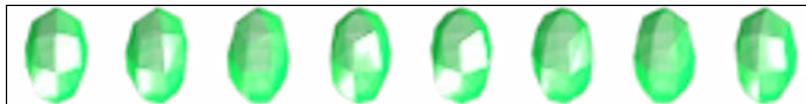
The following screenshot shows our animation setup in code:

```
sf::Vector2i spriteSize(32, 32);
sf::Sprite sprite(AssetManager::GetTexture("spriteSheet.png"));
//Set the sprite image to the first frame of the animation
sprite.setTextureRect(sf::IntRect(0, 0, spriteSize.x, spriteSize.y));

int framesNum = 8; //Animation consists of 8 frames
float animationDuration = 1; //1 second
```

First of all, note that we are using the `AssetManager` class (from *Chapter 2*, *Loading and using textures*) to load our sprite sheet. The next line sets the texture rectangle of the sprite to target the first image in our sprite sheet. Here is what this means in terms of the sprite sheet *texture*:
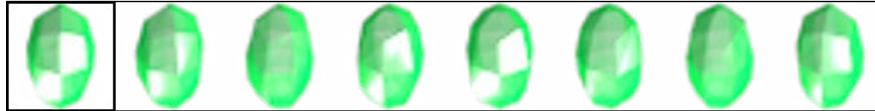
Next, we will move this texture rectangle once in a while to simulate a rotating crystal. In the previous code, we set the number of frames to eight (as many as there are in the sprite sheet), and set the time of the animation to one second in total, which means that each frame stays for about 0.125 seconds (the animation duration is divided by the number of frames) at a time. We know what we want to do now, so let's do it:

```cpp
while (window.isOpen())
{
    //Returns the elapsed time and restarts the clock
    sf::Time deltaTime = clock.restart();

    //Handle input

    //Accumulate time with each frame
    elapsedTime += deltaTime;
    float timeAsSeconds = elapsedTime.asSeconds();

    //Get the current animation frame
    int animFrame = static_cast<int>((timeAsSeconds / animationDuration) * framesNum) % framesNum;
    //Set the texture rectangle, depending on the frame
    sprite.setTextureRect(sf::IntRect(animFrame * spriteSize.x, 0, spriteSize.x, spriteSize.y));

    //Render frame
}
```

In the code, we first measure the delta time since the last frame and add it to the accumulated time. The last two lines of the code actually do all the work. The first one looks intimidating at first glance, but it is simply a way to choose the correct frame, based on how much time has passed and how long the animation is. The formula `timeAsSeconds / animationDuration` gives us the time relative to the animation duration. So let's say that 0.4 seconds have passed and our animation duration is 1 second. This leaves us with 0.4 seconds in local animation time. Multiply this 0.4 seconds by the number of frames, and we get the following result:

```
0.4 * 8 = 3.2
```

This gives us which frame we should be on at the moment, and how long we have been there. The current frame index is the whole part of 3.2 (which is three), and the fraction part (0.2) is how long we have been on that frame. In this case, we are only interested in the current frame so we will take that by casting the whole expression to `int`. This rounds the number down if the number is positive (which it always is in this case). The last part, `% frameNum` is there to restart the animation when it reaches beyond its last frame. So in the case where 2.3 seconds have passed, we have the following result:

```
2.3 * 8 = 18.4
```

We do not have a 19th frame to show, so we show the frame which corresponds to that in our local scale [0…7]. In this case:

```
18 / 8 = 2 (and 2 remainder)
```

Since the `%` operator takes the remainder of a division, we are left to show the frame with the index two, which is the third frame. (We start counting from zero as programmers, remember?)

The last line of the code sets the texture rectangle to the current frame. The process is quite straightforward—since we only have frames on the *x* axis, we do not need to worry about the *y* coordinate of the rectangle, and so we will set it to zero. The *x* is computed by `animFrame * spriteSize.x`, which multiplies the current frame by the width of the frame. In the case, the current frame is two and the frame's width is 32, so we get:

```
2 * 32 = 64
```

Here is what the texture rectangle will look like:
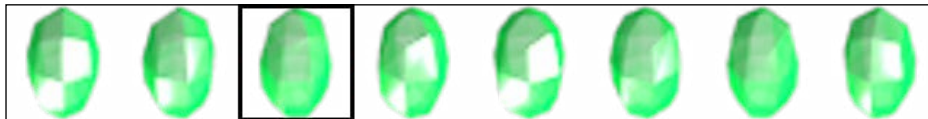
The last thing we need to do is render the sprite inside the render frame and we are done. If everything goes smoothly, we should have a rotating crystal on the screen with eight frames. With this technique, we can animate sprites of all kinds no matter how many frames they have or how long the animation is. There are problems with the current approach though—the code looks messy, and it is only useful for a single animation. What if we want multiple animations for a sprite (rotating the crystal in a vertical direction as well), and we want to be able to switch between them? Currently, we would have to duplicate all our code for each animation and each animated sprite. In the next section, we will talk about how to avoid these issues by building a fully featured animation system that requires as little code duplication as possible.

Building an animator

It is important to know exactly what we are doing before we start doing it, so let's list the specifications for our animator:

- The animator needs to animate a sprite from a single object or multiple texture objects

- Our animator needs to support animations with variable durations and a number of frames

- Our animator should hold multiple animations

- It needs to be able to switch between animations

- Each sprite should have its own animator object

- It should be easy to use

- Our animator needs to be able to perform automatic texture rectangle generation

It seems quite straightforward, doesn't it? Since we want to perform multiple animations per animator, we have to create a struct animation to hold each animation property. We are using struct rather than class with proper encapsulation to reduce the code size. In both cases though, each animation should have a duration time, a list of frames, a texture (used by the animation), loop information (does it loop?), and a name handle, which is used to reference that animation. Based on this design, here is what our structure should look like:

```cpp
struct Animation
{
    std::string m_Name;
    std::string m_TextureName;
    std::vector<sf::IntRect> m_Frames;
    sf::Time m_Duration;
    bool m_Looping;

    Animation(std::string const& name, std::string const& textureName,
        sf::Time const& duration, bool looping)
        : m_Name(name), m_TextureName(textureName),
        m_Duration(duration), m_Looping(looping)
    { }

    //Adds frames horizontally
    void AddFrames(sf::Vector2i const& startFrom,
        sf::Vector2i const& frameSize, unsigned int frames)
    {
        sf::Vector2i current = startFrom;
        for (unsigned int i = 0; i < frames; i++)
        {
            //Add current frame from position and frame size
            m_Frames.push_back(sf::IntRect(current.x, current.y, frameSize.x, frameSize.y));
            //Advance current frame horizontally
            current.x += frameSize.x;
        }
    }
};
```

It seems that we have managed to capture our design pretty well—there is one more additional method in there—`Animation::AddFrames()`. This particular method is there to ease our work when we create the animation. It takes a position, size, and a number of frames, and it iterates from that position horizontally to add frames into the `m_Frames` vector. It is essentially a shortcut we can use rather than putting in the frames one by one.

With the `Animation` structure done, we can now move on to `Animator`, which will actually use the animations to animate our sprite. The first thing that the animator needs to do is to be able to create and store animations for later use. Also, since animations are time dependent, it needs to have an `Animator::Update()` method, which gets called every frame with the delta time. And, last but not least, it needs a way to switch between animations.

With this in mind, here is what our private data looks like:

```
private:
    //Returns the animation with the passed name
    //Returns nullptr if no such animation is found
    Animator::Animation* FindAnimation(std::string const& name);

    void SwitchAnimation(Animator::Animation* animation);

    //Reference to the sprite
    sf::Sprite& m_Sprite;
    sf::Time m_CurrentTime;
    std::list<Animator::Animation> m_Animations;
    Animator::Animation* m_CurrentAnimation;

};
```

Ignore `Animator::FindAnimation()` and `Animator::SwitchAnimation()` for now, we will come back to the them later. What we are interested in right now is the data. As you can see, there isn't a sprite instance in `Animator`, but only a reference. This means that the sprite has to be constructed outside the class, and then should be passed to the constructor of `Animator`.

Other than the `Sprite` reference, there is the time accumulation counter (the same one which we used in the previous section of this chapter), a list of animations, and a pointer to the currently running animation. Note how we don't have a `vector` to store animations, but a list. This is mostly specific to C++, but you cannot keep pointers and references to the elements in a `vector`—they will become invalid once we start adding or removing elements from it. The `list` class, on the other hand, doesn't have such a problem, because its implementation ensures that pointers and references remain valid even after adding/removing elements from it.

Now, let's inspect the functionality that the `Animator` class provides by looking at its public member functions:

```cpp
public:
    struct Animation { ... };

    Animator(sf::Sprite& sprite);

    Animator::Animation& CreateAnimation(std::string const& name,
        std::string const& textureName, sf::Time const& duration, bool loop = false);

    void Update(sf::Time const& dt);

    //Returns if the switch was successful
    bool SwitchAnimation(std::string const& name);

    std::string GetCurrentAnimationName() const;
```

The first thing that grabs our eye is the `Animation` structure. This is the same structure that we talked about earlier in this section. It is declared inside the `Animator` class because both the classes are coupled with each other by design. Furthermore, the `Animation` class will not be heavily used outside `Animator`. The next line contains the constructor's declaration, which expects a `Sprite` reference. This is the reference which initializes the member `m_Sprite` field.

`Animator::CreateAnimation()` creates an animation from the given parameters, adds it to the list, and returns a reference to it. We will see exactly how that works in a moment.

`Animator::Update()` handles all the logic behind choosing the right frame for the right moment. `Animator::SwitchAnimation(string)` tries to switch the current animation to an animation with the given name.

It's time to look into the implementation of these functions and the ways in which we can use them. It makes sense to start from the constructor:

```cpp
Animator::Animator(sf::Sprite& sprite)
    :m_Sprite(sprite), m_CurrentTime(), m_CurrentAnimation(nullptr)
{
}
```

Not much to see here really—we have just initialized all our data. Note that the `Sprite` reference needs to be set in the initialization list otherwise, the code will not compile. This is how references work in C++—they need to be initialized as soon as possible. Another thing to take note of is the `nullptr` initialization of the current animation. It is always a good practice to initialize pointers to `nullptr`.

`Animator::CreateAnimation()` is a bit more interesting—let's take a look:

```cpp
Animator::Animation& Animator::CreateAnimation(std::string const& name,
    std::string const& textureName, sf::Time const& duration, bool loop)
{
    m_Animations.push_back(
        Animator::Animation(name, textureName, duration, loop));

    //If we don't have any other animations, use that as current animation
    if (m_CurrentAnimation == nullptr)
        SwitchAnimation(&m_Animations.back());

    return m_Animations.back();
}
```

`Animator::CreateAnimation()` creates an animation using a number of parameters. As we established earlier, each animation needs a name so that we can reference it from outside the class. Also, each animation has a texture and duration associated with it. The `loop` parameter determines whether the animation should loop or play only once. With all these parameters, we will initialize a new instance of the animation and place it in the `m_Animations` list.

If that was our first animation, we would want to set that animation as the current one. This ensures that we have something to play once the `Animator::Update()` method is called. `Animator::SwitchAnimation(Animation*)` does exactly that—it takes an animation and sets it as the current animation. We'll see how that works in a moment.

The last line of the method returns a reference to the animation that we just created. This enables the caller to further manipulate the animation by adding frames or changing some of its initial values.

As promised, here is one of the overloads of the `Animator::SwitchAnimation()` method:

```cpp
void Animator::SwitchAnimation(Animator::Animation* animation)
{
    //Change the sprite texture
    if (animation != nullptr)
    {
        m_Sprite.setTexture(AssetManager::GetTexture(animation->m_TextureName));
    }

    m_CurrentAnimation = animation;
    m_CurrentTime = sf::Time::Zero; //Reset the time
}
```

Since we are potentially switching to a new animation, we need to change the texture of the sprite to the texture of the new animation. However, since both `animation` and `m_CurrentAnimation` can be `nullptr`, we have to check for that first. If `animation` is not `nullptr`, we can safely use the texture name to call our `AssetManager` and ask for the texture's reference. In this case, even if the texture of the previous animation is the same as this one, the overhead of triggering a change will be minimal because we are only dealing with references.

Next, we set the `m_CurrentAnimation` pointer to `animation` for later use and reset the time. The `animation` parameter, being `nullptr`, is fine in this instance since it just means that we do not want an animation to be playing at this time.

There is an overload of the `Animator::SwitchAnimation(Animation*)` function, which takes `string` instead of `Animation*`. This function is public and is used to change animations based on their name, rather than direct pointers. Here is its implementation:

```cpp
bool Animator::SwitchAnimation(std::string const& name)
{
    auto animation = FindAnimation(name);
    if (animation != nullptr)
    {
        SwitchAnimation(animation);
        return true;
    }

    return false;
}
```

This method is quite straightforward—it tries to find an animation with the name given. If it fails, it simply returns `false`. If it succeeds, it uses this animation pointer to switch to that animation by calling its overload (the one that we just discussed). Finally, it returns `true` to indicate to the caller that this animation exists, and the switch was successful.

`Animator::FindAnimation()` is unremarkable but, for the sake of clarity, we will have a look at it:

```cpp
Animator::Animation* Animator::FindAnimation(std::string const& name)
{
    for (auto it = m_Animations.begin(); it != m_Animations.end(); ++it)
    {
        if (it->m_Name == name)
            return &*it;
    }

    return nullptr;
}
```

The only thing to note here is that this method returns the first occurrence of an animation with the same name, which means that we have to be careful when we create animations. In the current implementation, animations with the same name should be avoided because it is impossible to access them separately. We can also consider throwing an exception if we find animations with the same name to improve debugging.

There will be times when we would like to inspect which animation is currently playing from outside the class. For that, we implement a simple method which returns the name of the currently playing animation:

```cpp
std::string Animator::GetCurrentAnimationName() const
{
    if (m_CurrentAnimation != nullptr)
        return m_CurrentAnimation->m_Name;

    //If no animation is playing, return empty string
    return "";
}
```

Now that we've taken a look at all the supporting methods of the class, let's inspect what the most important of them looks like—the `Animator::Update()` method:

```cpp
void Animator::Update(sf::Time const& dt)
{
    //If we don't have any animations yet return
    if (m_CurrentAnimation == nullptr)
        return;

    m_CurrentTime += dt;

    //Get the current animation frame
    float scaledTime = (m_CurrentTime.asSeconds() / m_CurrentAnimation->m_Duration.asSeconds());
    int numFrames = m_CurrentAnimation->m_Frames.size();
    int currentFrame = static_cast<int>(scaledTime * numFrames);

    //If the animation is looping, calculate the correct frame
    if (m_CurrentAnimation->m_Looping)
        currentFrame %= numFrames;
    else if (currentFrame >= numFrames)//if the current frame is greater than the number of frames
        currentFrame = numFrames - 1; //Show last frame

    //Set the texture rectangle, depending on the frame
    m_Sprite.setTextureRect(m_CurrentAnimation->m_Frames[currentFrame]);
}
```

The preceding code should ring a bell or two—it is quite similar to what we did for the animation in the previous section. It starts off with a check for `m_CurrentAnimation`. Since that value can be `nullptr`, we do not need to do anything in that case. Next, we add the delta time to the local time buffer. We scale that time by the animation duration to get `scaledTime`, which is used to figure out which frame to show. As we did in the previous section, we find the current frame by multiplying the scaled time by the number of total animation frames and round that number down.

The next part is new. If we want the animation to keep looping, we use the modulo operator (%) to bring the frames around. Otherwise, we play the animation only once and use the last frame until the animation is changed (or replayed).

Now that we have the current frame, we only need to set the texture rectangle for the sprite. Since we hold all the frames inside the animation, retrieving them is extremely easy — use the `Animation::m_Frames` vector as an array by calling its index operator with the `currentFrame` index.

That's it — a powerful sprite animator is created without much effort at all. Next, we will look at some examples of how we can use it in practice.

Using the animator

Let's start with the example of the rotating crystal. Here is the initialization part:

```cpp
sf::Vector2i spriteSize(32, 32);
sf::Sprite sprite;
Animator animator(sprite);
//Create an animation and get the reference to it
auto& idleAnimation = animator.CreateAnimation("Idle", "spriteSheet.png", sf::seconds(1), true);
//Add frames to the animation
idleAnimation.AddFrames(sf::Vector2i(0, 0), spriteSize, 8);
```

It is important to note that a sprite has to exist already in order to create an animator because it takes a sprite reference in its constructor. Once we create an animator for the sprite, we will add an animation. Since we are replicating our previous example, there is only one animation. The animation's name is `Idle` and it uses the `spriteSheet.png` texture. It takes one second to complete and it is set to a looping state.

Once we get the animation reference, we will add some frames from the texture. We know how big the sprite is so, in order to get all the frames from the texture, we start from (0, 0) and continue horizontally eight times by moving with 32 pixels each time. This is all done inside the `Animation::AddFrames()` method.

That's pretty much it. The only thing left to do is to call the update method of `Animator` in the update frame of the game loop:

```cpp
sf::Clock clock;
while (window.isOpen())
{
    //Returns the elapsed time and restarts the clock
    sf::Time deltaTime = clock.restart();

    animator.Update(deltaTime);

    window.clear(sf::Color::Black);

    window.draw(sprite);

    window.display();
}
```

Multiple animations

Creating a single animation seemed quite straightforward, didn't it? With multiple animations, things are not much more complicated. Let's have a scenario where we have two textures—`spriteSheet.png` and `myTexture.png`, and we want to have four animations, each of which uses one of these textures. Here is how the setup might look like:

```cpp
Animator animator(sprite);
//Idle animation with 8 frames @ 1 sec looping
auto& idleAnimation = animator.CreateAnimation("Idle", "spriteSheet.png", sf::seconds(1), true);
idleAnimation.AddFrames(sf::Vector2i(0, 0), spriteSize, 8);

//IdleShort animation with 8 frames @ 0.5 sec looping
auto& idleAnimationShort = animator.CreateAnimation("IdleShort", "spriteSheet.png", sf::seconds(0.5f), true);
idleAnimationShort.AddFrames(sf::Vector2i(0, 0), spriteSize, 8);

//IdleSmall animation with 5 frames @ 1.5 sec looping
auto& idleAnimationSmall = animator.CreateAnimation("IdleSmall", "myTexture.png", sf::seconds(1.5f), true);
//Adding frames multiple times from different locations
idleAnimationSmall.AddFrames(sf::Vector2i(64, 0), spriteSize, 3);
idleAnimationSmall.AddFrames(sf::Vector2i(64, 32), spriteSize, 2);

//IdleOnce animation with 8 frames @ 0.5 sec not looping
auto& idleAnimationOnce = animator.CreateAnimation("IdleOnce", "myTexture.png", sf::seconds(0.5f), false);
idleAnimationOnce.AddFrames(sf::Vector2i(0, 0), spriteSize, 8);
```

Note all the different ways in which we can create animations. We are not limited to time, texture, name, number of frames, or even frame locations. We can even add frames one by one from the texture, by calling `Animation::AddFrames()` with 1 as the last parameter. Having many animations is definitely a nice thing to have, however, we need a way to switch between them. The following code demonstrates how to go between animations on key press events:

```cpp
sf::Event ev;
while (window.pollEvent(ev))
{
    if (ev.type == sf::Event::KeyPressed)
    {
        if (ev.key.code == sf::Keyboard::Key::Num1)
            animator.SwitchAnimation("Idle");
        else if (ev.key.code == sf::Keyboard::Key::Num2)
            animator.SwitchAnimation("IdleShort");
        else if (ev.key.code == sf::Keyboard::Key::Num3)
            animator.SwitchAnimation("IdleSmall");
        else if (ev.key.code == sf::Keyboard::Key::Num4)
            animator.SwitchAnimation("IdleOnce");
    }
}
```

It is as simple as calling the `Animator::SwitchAnimation()` method and passing the name of the new animation. Keep in mind that passing the name of the animation which is current playing will reset the time, and the animation will start from the beginning. This means that calling the method during each frame is not a good idea. If we want to check the name of the animation which is currently playing, we can call `Animator::GetCurrentAnimationName()` and only switch animations if the new animation is a different one.

As you can see, the `Animator` class makes animation easier to manage and, on top of that, it is extremely scalable. Creating multiple sprites isn't a problem either, as we can create as many animators as we want.

Summary

Sprite animations seem quite easy now, don't they? Just keep in mind that there is a lot more to explore when it comes to animation. Not only are there different techniques to doing them, but also perfecting what we've developed so far might take some time. Fortunately, what we have so far will work *as is* in the majority of cases so I would say that you are pretty much set to go.

Cameras and OpenGL rendering are coming up next. I wouldn't miss that if I were you.

4

Manipulating a 2D Camera

In this chapter, we will talk about cameras and **OpenGL**, and how we can use them to our benefit. We will cover the topics of cameras in depth, but with regards to OpenGL, we have only briefly mentioned its integration in **SFML**. The **OpenGL API** is too big to be covered in this book, let alone a single chapter. If you don't know how to write OpenGL code or do not have any desire to use OpenGL, then it is fine to skip the last part of this chapter. On the other hand, if you want to know how OpenGL can help you, take a look at what the second part of the chapter offers.

In this chapter, we will cover the following topics:

- What is a camera?
- Manipulating cameras with `sf::View`
- What is OpenGL?
- Using OpenGL inside SFML

What is a camera?

The chance that you've not come across cameras in games development is quite slim. They are an essential part of any game. Essentially, a camera is that point in space through which you can look at the game world. There are more parameters associated with cameras, both in 2D and 3D space, but in this chapter, we will only focus on what SFML has to offer.

Before we jump to the code, let's get some facts straight. Since SFML is used mainly with 2D games, the `camera` class exclusively uses an **orthographic** projection. In this projection, every object appears as if it has no perspective augmentations. The alternative, as you might have guessed, is a perspective projection, which actually changes how objects appear on the screen based on the physics of the human eye (object appear smaller in the distance, and so on). However, this projection is used mainly in 3D games, and it does not belong in the world of 2D. Here is a little comparison:
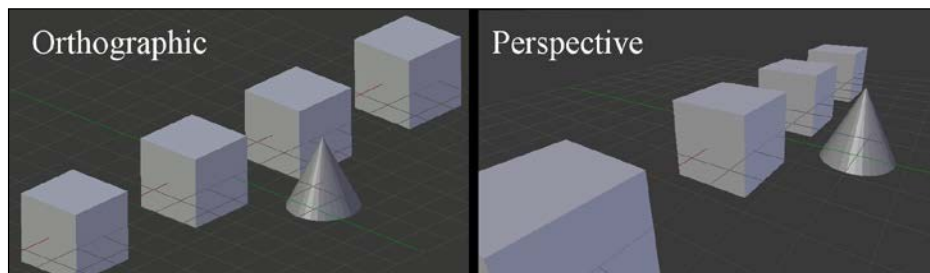
Using a perspective projection for a game which relies on two dimensions does not make much sense because the image of the sprites gets distorted. This is why SFML does not even offer the choice to use one. We can always create a custom camera with OpenGL though, a topic which we will discuss later in this chapter.

When should we use a camera?

We don't always have to manipulate the camera. If a game has only one screen (such as a match-3 game for example), then there is no point in modifying the camera since it will always be static in the centre of the board. Other examples might include navigating through the main menu—again the position of the camera is static, and thus there is no need to do anything.

But let's say that we are making an **Role Playing Games**(**RPG**), and there is a big world to explore. In that case, we would definitely want to implement a camera to either move with our character or in relation to it. Most platformer games work in the same way even though they have discrete levels. Basically, whenever we have a world (or level) to explore, we should consider using a camera.

Now that we know the basic concept of a camera, let's look at how to implement one.

How does SFML implement a camera?

If we want to modify the default camera which comes with each `sf::Window` instance, we have to tackle the `sf::View` class.

The `View` class behaves exactly like a typical camera—limiting what the player can see in the world by a set of parameters. This is how we create and use `View`:

```
auto wSize = window.getSize();
sf::View view(sf::FloatRect(0, 0, wSize.x, wSize.y));

//Initialize view

window.setView(view);
```

The constructor of the `View` class takes a single `FloatRect` parameter, which sets the desired view area of the world. If we have a bigger view area, its contents are scaled down to fit in the window, and vice versa. In this example, the area matches the window size, so it does not change how objects are rendered.

Finally, when we have everything in `View` configured, we need to tell the window to use it by calling `RenderWindow::setView()`. This copies over the view in the `RenderWindow` object, so we don't need to keep the original view alive, as we do with resources such as `Texture`.

Now, let's see what the `View` class can actually do.

Manipulating cameras with sf::View

Arguably, the most important feature of the `View` class is its ability to change the center of the view area. By default, the center of the view is the center of the view area, which means that, if our view area is of the size (640, 480), the center of the view will be (320, 240). This makes the rendering objects with position (`0`; `0`) appear in the top-left corner. This is the same behavior we get when using the default `View` of a SFML window. To change the center of the view we can call `View::setCenter()` or `View::move()`; here is an example:

```
auto wSize = window.getSize();
sf::View view(sf::FloatRect(0, 0, wSize.x, wSize.y));

//The view is centered around the world point (0; 0)
view.setCenter(sf::Vector2f(0, 0));

window.setView(view);

sf::Vector2f spriteSize = sf::Vector2f(32, 32);
sf::Sprite sprite(AssetManager::GetTexture("myTexture.png"));
sprite.setOrigin(spriteSize * 0.5f); // Sprite origin at it's center
```

If we center the view at position (`0`; `0`), that position in the world will appear in the center of the screen. The sprite in the code is positioned at (`0`; `0`) by default, so it would appear in the center of the screen; here is the result:
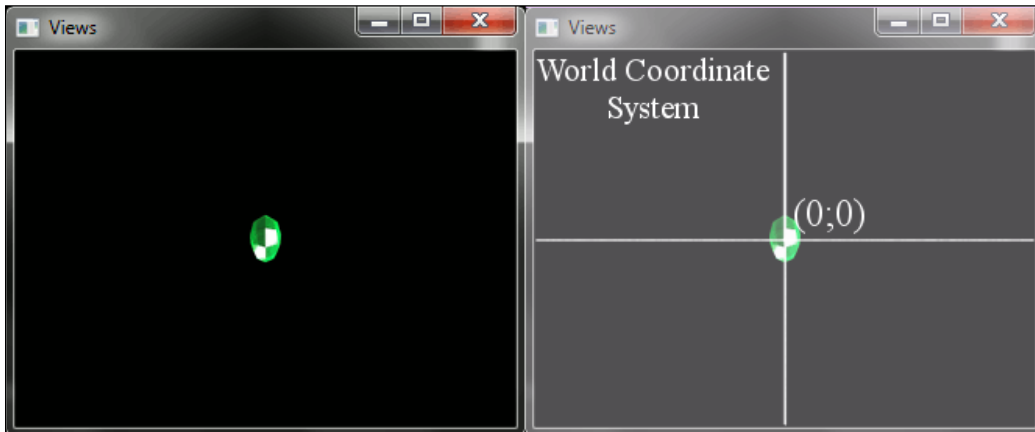
Typically, centering the view on the main character is a simple and effective technique to carry out the camera position logic. Also, it requires only two lines of code in the update frame:

```
view.setCenter(sprite.getPosition());
window.setView(view);
```

Note that we need to call `RenderWindow::setView()` and pass `view` again since `RenderWindow` only holds a copy of `view`. Just changing our old `view` instance will not affect `view` stored in `RenderWindow`.

Rotating and scaling a view

There are two more transformations that we can perform on any view instance—rotation and scale. Both of them have limited uses, but comes in handy when we want specific features from our camera.

To rotate a view, we call `View::setRotation()` or `View::rotate()`, depending on the type of rotation we want to perform. The `View::setRotation()` method assigns an absolute value to the rotation of `View`, whereas `View::rotate()` adds the rotation value passed. The latter is typically used when we want to gradually increase the rotation over a period of time.

The rotation itself works as one might expect—it rotates the scene (every object) around the center of the view. Here is a setup for a test we are going to perform:

```cpp
auto wSize = window.getSize();
//The view is centered around the world point (0; 0)
//The view has the size of the window
sf::View view(sf::Vector2f(0, 0), sf::Vector2f(wSize.x, wSize.y));

//Set rotation view.setRotation(...);

window.setView(view);

sf::Vector2f spriteSize = sf::Vector2f(32, 32);
auto& texture = AssetManager::GetTexture("myTexture.png");

//Top left
sf::Sprite sprite1(texture);
sprite1.setOrigin(spriteSize * 0.5f);
sprite1.setPosition(sf::Vector2f(-80, -80));

//Top right
sf::Sprite sprite2(texture);
sprite2.setOrigin(spriteSize * 0.5f);
sprite2.setPosition(sf::Vector2f(80, -80));

//Bottom right
sf::Sprite sprite3(texture);
sprite3.setOrigin(spriteSize * 0.5f);
sprite3.setPosition(sf::Vector2f(80, 80));

//Bottom left
sf::Sprite sprite4(texture);
sprite4.setOrigin(spriteSize * 0.5f);
sprite4.setPosition(sf::Vector2f(-80, 80));
```

This time, the view is called with a different constructor. Rather than passing a rectangle, we will pass a center position and size. This serves the same purpose, but allows us to specify the center point more easily.

After we initialize the view, we create four sprites in the four corners of the screen around the world point (`0;0`). The following are the tests done with a rotation of 0 and 45 degrees on the scene:
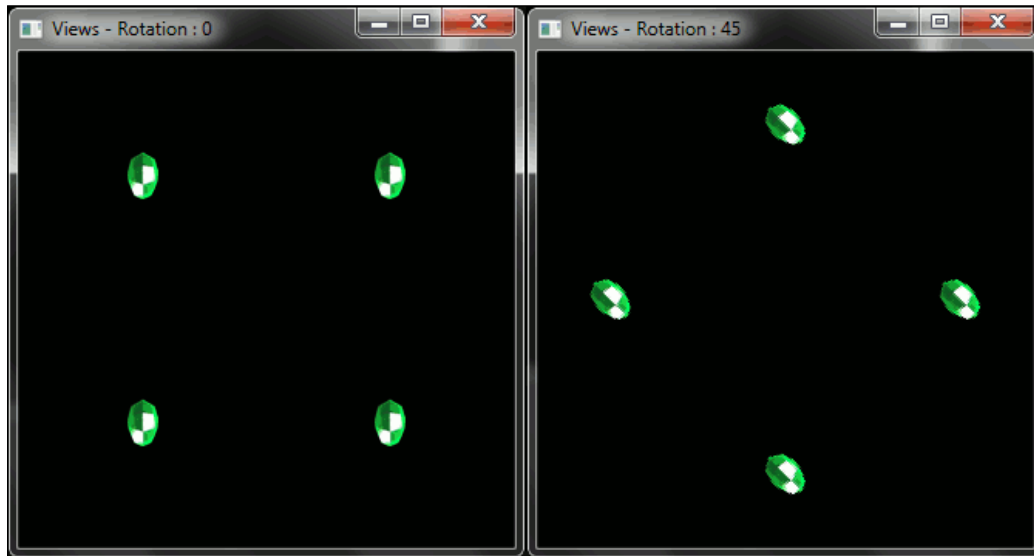
Rotating the view has limited uses in terms of game development. It can be used for animating a particular event—the death of the main character (a slow zoom in along with rotation), taking damage (a slight shake of the camera). We can also use it to rotate the world in a top-down game around a centered character. In general, rotation is useful, but has limited use cases.

Views can also be scaled. By scaling the view, we can show either more or less of the world, depending on the direction of the scale. In terms of games, this feature is typically called a **zoom** (either **in** or **out**). The zoom is connected directly to the size of the view. When we tell the view to display an area two times larger than the window, we are effectively zooming by a factor of two. On the other hand, if we want to show objects up close, we have to use zoom by a factor of `1/x`, where `x` is the zoom factor. Making the view two times smaller, for example, requires a factor of 1 / 2 = 0.5.

Let's demonstrate the scaling by showing two examples with the sprites, which we just used for rotation:
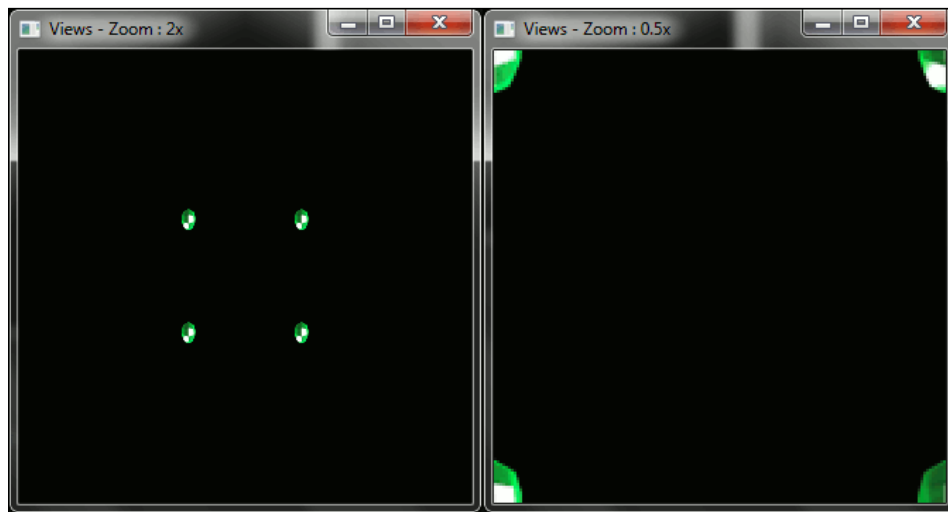
The process of setting the zoom itself is done either by calling `View::zoom()` with a zoom factor, or by changing the size of the view with `View::setSize()`. As it is, with `View::rotate()` and `View::move()`, `View::zoom()` is used mostly when we want to have a continuous motion throughout a number of frames.

It is important to note that `View` does not store a zoom factor, but only the size of the view. This means that calling `View::zoom()` with a factor different than `1` will changes the view size each time, even if we pass the same factor. The method does not set the zoom, it just modifies the size of the view by that factor. For example, if we call `View::zoom()` twice with a factor of `1/2`, we will end up with:

(1/2) * (1/2) * size = (1/4) * size

We are effectively doing the same thing as calling `View::zoom()` with a factor of `1/4`.

Since `View::zoom()` takes just a factor, it uses the current view size to estimate how much it has to modify both the width and height of the view. If we want to change the width and height by two different factors, we have to use `View::setSize()` with our own values of the width and height; here is an example:

```
auto wSize = window.getSize();
sf::View view(sf::Vector2f(0, 0), sf::Vector2f(wSize.x, wSize.y));

//First example
view.setSize(wSize.x * 2, wSize.y);
//Second example
view.setSize(wSize.x, wSize.y * 2);

window.setView(view);
```

The preceding code produces the following result:

As you can see, both the images appear squashed either in the *x* or *y* axis. This happens because we are trying to fit two times of the pixels on the given axis than the screen allows us to. You might recognize the effect since it is similar to placing a texture onto a surface, which has different dimensions from the texture.

There are a few more things that we can do with `View`. That's what we will explore in the next section.

Viewports

Every view has a viewport associated with it. A viewport is the area of the window in which the view is displayed. The area is represented by a rectangle, which uses normalized coordinates [0...1]. By default, the viewport is equal to the size of the view (0, 0, 1, 1). We can change this by calling `View::setViewport()`. Let's say that we want to render our scene only in the top-left quadrant of the screen. The following code will do it:

```cpp
auto wSize = window.getSize();
sf::View view(sf::Vector2f(0, 0), sf::Vector2f(wSize.x, wSize.y));

view.setViewport(sf::FloatRect(0, 0, 0.5f, 0.5f));

window.setView(view);
```

Here is the result of the preceding code in a widescreen window:
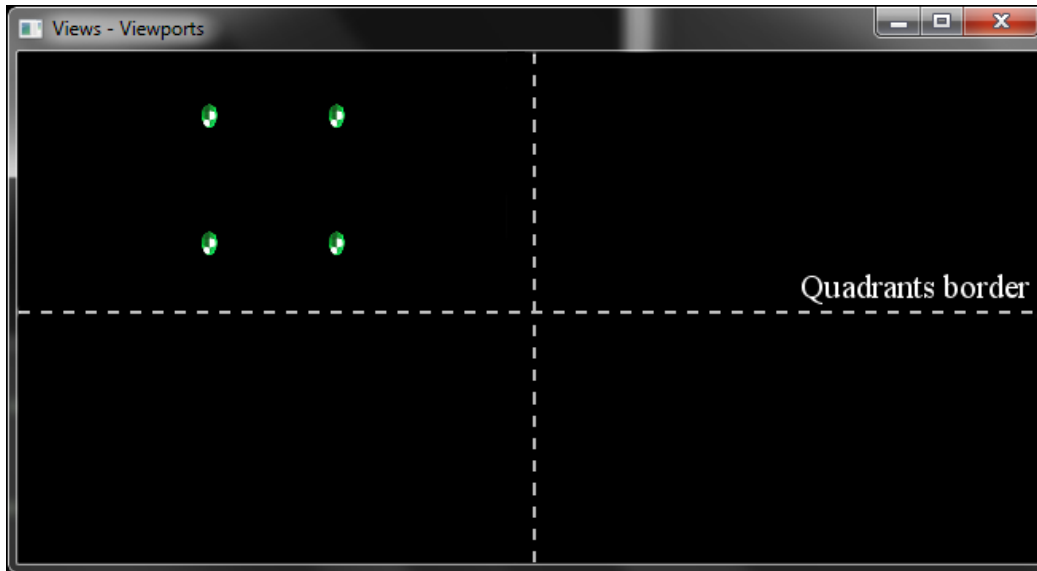
Since we can create multiple views, we can switch between them in the render frame and render the scene repeatedly. In this way, we will have the scene shown as they appear from the different views. In our previous example, we can render the same scene in all four quadrants, but with different transformations. To do that, we have to initialize the four views in the four corners of our screen with the following viewports: top left (0, 0, 0.5, 0.5), top right (0.5, 0, 0.5, 0.5), bottom left (0, 0.5, 0.5, 0.5), and bottom right (0.5, 0.5, 0.5, 0.5). Once we have these, we manipulate their transformation and put them in a `viewList` vector. Then our render frame is as simple as:
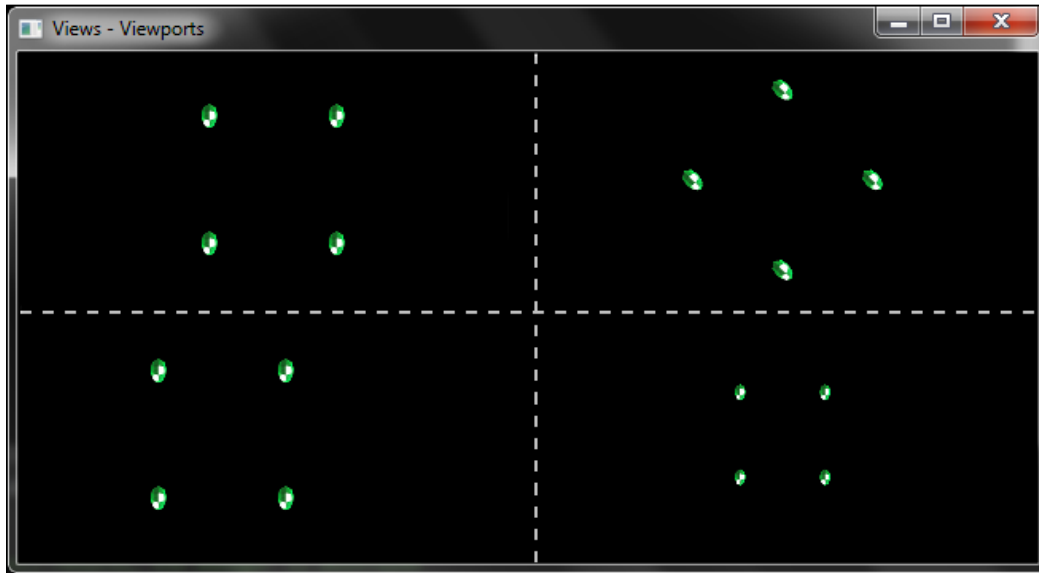
```cpp
window.clear(sf::Color::Black);

for (auto it = viewList.begin(); it != viewList.end(); ++it)
{
    //Set the view
    window.setView(*it);

    //Render sprites
}

window.display();
```

The result depends on how we choose to manipulate our views. Here is an example which you can find in the code samples provided with this book:

As you can see, viewports are extremely useful for having multiple views at the same time. This makes it possible to create local split screen multiplayer games quite easily. Apart from that, we can use a different view for our UI as an example. To do this, we render our game with the game camera, then switch to the UI camera (which doesn't move with the world), and render our game's UI. Maps and minimaps are the perfect targets for views, since we are essentially rendering the world inside another window (or view).

In general, views are extremely useful but they have one weakness—the window coordinates do not correspond with the view's contents once we start manipulating the cameras. For example, if we try to handle a click event, the mouse position will not point to the same position in the scene for different views. As always, SFML has got us covered—by introducing coordinate mapping.

Mapping coordinates

Once we have a view attached to a window, we can call `RenderWindow::mapPixelToCoords()` with a position from the window, and this transforms the position vector to a location in the scene (world coordinates). Here is an example of how to convert mouse coordinates to world coordinates in a button press event:

```cpp
sf::Event ev;
while (window.pollEvent(ev))
{
    if (ev.type == sf::Event::MouseButtonPressed)
    {
        sf::Vector2f sceneCoords = window.mapPixelToCoords(
            sf::Vector2i(ev.mouseButton.x, ev.mouseButton.y));

        //Do something at that location in the scene
    }
}
```

It is important to remember that the function will only work with `View`, which is currently applied to the window. This needs to be taken into consideration if we are using multiple views for our window.

We can also go the other way around—get the screen coordinate from a location in the scene. This is done by `RenderWindow::mapCoordsToPixel()`. This is useful when we want to map a location from a scene to other views. For example, if we want to display a health bar over characters, we get their location in scene, map the location to a screen pixel, then map that pixel to our UI view coordinates, and display a health bar there. This keeps everything nice and organized without much effort.

This covers the topic of `views`. The next part of the chapter will cover OpenGL code integration inside SFML.

What is OpenGL?

OpenGL is a cross-platform graphics API which is used as an interface to talk to the graphics card. The most important feature of any graphics API is its ability to render objects on the screen. While OpenGL certainly does that, it has many other useful features. However, since it has been around for years, and GPU technology has changed drastically, not all features of the new versions of OpenGL are supported on all graphics cards.

Should you use OpenGL?

To be fair, SFML supports a lot of the functionality that OpenGL provides. In fact, SFML uses OpenGL internally to implement that functionality. However, since SFML is a high-level library, there will always be a performance hit associated with using it. In most cases, the hit isn't a huge problem, since the benefit of such a high-level library is that it is extremely quick to write with. However, some circumstances just need that extra performance to achieve the target FPS. In that case, SFML provides an easy integration of OpenGL code without having to worry about too many things.

Perhaps you want to render tens of thousands of sprites on the screen at once, or maybe, you want to add a feature on top of the `Window` class. OpenGL is then the place to turn to.

Another reason to use OpenGL is to create 3D games. SFML provides some tools, which can be used in a 3D environment, but ultimately they are not enough to create a fully featured 3D experience (3D models, lighting, shadows, and so on). Therefore, we have to use OpenGL.

As mentioned at the beginning of this chapter, OpenGL is a huge API, and this chapter does not teach you how to use any of its features. It only gives you a general guideline of how to use it alongside SFML.

Using OpenGL inside SFML

Before we start using any of the OpenGL calls, we need to make sure that the graphics context is initialized. The context holds the data (states, default framebuffer, and so on) which allows OpenGL to function. This is done automatically when we create a `window` instance:

```cpp
#include <SFML/Window.hpp>
#include <SFML/OpenGL.hpp>

int main()
{
    sf::ContextSettings settings;
    settings.depthBits = 24;
    settings.stencilBits = 8;
    settings.majorVersion = 3;
    settings.minorVersion = 0;
    settings.antialiasingLevel = 2;
    sf::Window window(sf::VideoMode(640, 480), "OpenGL", sf::Style::Default, settings);

    //Window is ready to receive OpenGL calls here

    while (window.isOpen())
    {
        //Game loop
    }

    return 0;
}
```

To use any OpenGL calls, we need to include `<SFML/OpenGL.hpp>`. This is the common header file that SFML provides for us to use on all platforms. It is also important to mention that we do not need to use the familiar `RenderWindow` class if we are going to render everything using only OpenGL. The `Window` class will suite us just fine. On the other hand, if we want to use OpenGL with the graphics module for some reason, then we can go back to `RenderWindow`. In this example, we are only going to use the window module, which holds the `Window` class.

Note that the window takes an instance of `ContextSettings` (an optional parameter). We talked a little bit about this in *Chapter 1*, *Getting Started with SFML*, but let's see what these settings mean in more detail. The `ContextSettings` structure class has the following five fields that we can change.

Setting	Description	Common value ranges
`depthBits`	The field allows us to suggest the number of bits per pixel for the depth buffer	[0, 8, 16, 24, 32]
`stencilBits`	The field allows us to suggest the number of bits per pixel for the stencil buffer	[0, 8]
`majorVersion`	The field allows us to suggest the major version of OpenGL	[1…4]
`minorVersion`	The field allows us to suggest the minor version of OpenGL	[1…5]
`antialiasingLevel`	The field allows us to suggest the multisampling level	[0…16]. Typically a power of 2 yields the best results—1, 2, 4, 8, 16

These values do not force SFML to use them, and no exception will be thrown if we try to use them on hardware that does not support them. SFML chooses the closest (and probably best) option that is supported on the system. We can check which options were selected by getting the `ContextSettings` from the window:

```cpp
auto wSettings = window.getSettings();
std::cout << "depthBits: " << wSettings.depthBits << std::endl;
std::cout << "stencilBits: " << wSettings.stencilBits << std::endl;
std::cout << "antialiasingLevel: " << wSettings.antialiasingLevel << std::endl;
std::cout << "version: " << wSettings.majorVersion << "." << wSettings.minorVersion << std::endl;
```

Here is an example result:

```
depthBits: 24
stencilBits: 8
antialiasingLevel: 2
version: 4.4
```

Once we get everything set up, we can create our familiar loop:

```cpp
while (window.isOpen())
{
    sf::Event ev;
    while (window.pollEvent(ev))
    { /* Handle events */ }

    //Update frame

    //Set red clear color;
    glClearColor(1, 0, 0, 1);
    //Clear the screen and the depth buffer
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    //Render things here

    //SwapBuffers
    window.display();
}
```

Since the SFML window is built to work with OpenGL, integrating it is as simple as the preceding code. However, when it comes to mixing the graphics module and OpenGL, it gets slightly messier. We have to save and restore the OpenGL states each time we switch between rendering with SFML and rendering with OpenGL explicitly. Here is the code when we draw shapes using both the graphics module and OpenGL:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

//Draw shape using OpenGL

window.pushGLStates();

//Draw shape using SFML

window.popGLStates();

//Continue drawing using OpenGL

//SwapBuffers
window.display();
```

The preceding code seems absolutely fine for small examples which do not require many resources or do not have any class structure. Let's imagine that we have a `GameObejct` class, which has two nice and tidy methods `GameObject::update()` and `GameObject::render()`. In the `update()` method, we handle the game logic and in `render()`, we render the object. This is what our main loop looks like:

```
//Update frame
for (auto it = gObjects.begin(); it != gObjects.end(); ++it)
{
    it->update();
}

//Render frame
window.clear(sf::Color::Black);

for (auto it = gObjects.begin(); it != gObjects.end(); ++it)
{
    it->render(window);
}

window.display();
```

Now suppose that we want some extra functionality from our class, which we want to implement using OpenGL. Our class structure will look something like this:

```
class GameObjectGL : public GameObject
{
    void render(sf::RenderWindow& window) override
    {
        //Render object using OpenGL
    }
};

class GameObjectSFML : public GameObject
{
    void render(sf::RenderWindow& window) override
    {
        window.pushGLStates();
        //Render object using Graphics module
        window.popGLStates();
    }
};
```

Since `RenderWindow::pushGLStates()` is an expensive operation, our `GameObjectSFML` class seems awfully inefficient, doesn't it? Having to save and restore states in OpenGL for each object that uses SFML seems like a complete waste of resources. One solution to this problem is to create a second render function in our base class, which could be called `GameObject::renderGL()`, for example. In our main loop, things will change a bit:

```
//Render frame
window.clear(sf::Color::Black);

//Call GameObject::renderGL() on all objects

window.pushGLStates();

//Call GameObject::render() on all objects

window.popGLStates();

window.display();
```

In this way, we will save a lot of unnecessary driver calls and make the code inside the `GameObject` class a bit cleaner. Every object that does not use OpenGL will get this method empty, while others will get `GameObject::render()` empty.

OpenGL in multiple windows

SFML allows us to use multiple windows with each application. It is quite uncommon for games to have more than one window, but not for media applications. When we want to render an object in a specific window, we call `RenderWindow::draw()`. However, when we want to use OpenGL, we need to specify which window is affected by its function calls. This is simply done by `Window::setActive()`. When we want to start rendering on a window, we just call `setActive()` and start using OpenGL.

With this, our OpenGL session is over.

Summary

In this chapter, we went through the importance of cameras, and how they function in SFML. We saw how to transform the `View` class, and how we create split screen multiplayer setups. In the second section of this chapter, we talked about OpenGL and how it fits with SFML.

In the next chapter we will explore three conceptually simple, but ultimately crucial, features of any game—sounds, music, and text.

5
Exploring a World of Sound and Text

We have come a long way since starting with SFML, and we still have a little further to go before we reach the end. This chapter examines the sound features of SFML which are inside the audio module. This includes sound, music, and 3D sound environments. The last bit of this chapter explains how to render text on screen.

In this chapter, we will cover the following topics:

- Audio module—an overview
- Sound versus music
- Audio in action
- `sf::SoundSource` and sound in 3D
- Getting started with `sf::Text`

Audio module – overview

Until now, we have so far only used the window, graphics, and system modules of SFML. The window module handles native OS windows as well as the features associated with them. The graphics module makes it easy for us to draw objects on the screen. The system module holds the vector classes as well as encapsulating the features of an OS behind common classes such as the `Clock` and `Time` classes, which deal with time.

There are two more modules inside SFML—audio and network. We will talk about the network module in the last chapter of this book. This chapter is mostly dedicated to the audio module and its features.

The most important classes in this module are `sf::Sound` and `sf::Music`. They give us a way to play sounds and music. There is also a feature which allows you to play 3D sounds, which means that the sounds are played from different directions depending on the listener's position and orientation. We will explore all of these features in depth in this chapter. There are a few more things that the module contains, which are beyond the scope of this book, such as the `SoundRecorder` class, which can record sounds from an input device (a microphone for example).

That being said, we are now ready to start with the audio module. The first thing that we need to talk about is the fact that there are two different classes through which you can play audio—the `Sound` and the `Music` class. From the outside, it seems that they are doing the same thing—playing an audio file; however, as the saying goes "Don't judge a book by its cover".

Sound versus music

At first glance, the existence of these two classes might seem odd, but they ultimately serve different purposes, and it's all due to how they are implemented.

The `Sound` class loads all of its data into system memory, and this makes playing the audio sample very quick. The `Music` class, on the other hand, opens a stream to a file on the hard drive (or the RAM) and loads small chunks of data, which are played one after the other. Due to its design, the `Music` class has a playback delay due to transferring the data at such a slow place.

Both the classes provide different benefits—the `Sound` class almost instantly plays, but takes a lot of system memory, whereas the `Music` class is slower to play, but doesn't use much RAM at all. As such, both the classes are useful in different situations. For example, if the audio file is small enough to store in system memory, we should load it using the `Sound` class. This is applicable when the sound is to be played instantaneously after we call its `play()` method. Sometimes, we have to make memory sacrifices when the file is too big, and it has to be played immediately—that's why resource management is important (to remove unused assets and load new ones). The `Music` class, on the other hand, is mostly used for big audio files, where it is not important even if their playback is delayed a bit at the start. This class is mostly used for the background music in the game.

`Sound` and `Music` have the same class in their inheritance tree, `SoundSource`, which provides a common audio functionality, such as altering the pitch and 3D positioning. The `Sound` class derives from it directly, whereas `Music` derives from `SoundStream`. Finally `SoundStream` derives from `SoundSource`. The following diagram demonstrates their relation:
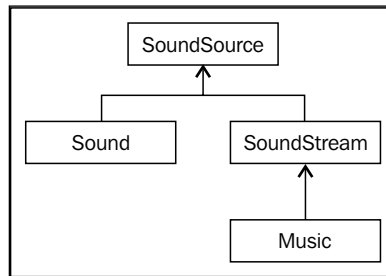
We will find out more about `SoundSource` and `SoundStream` later in this chapter.

This whole section can be summarized like this: make use of `Sound` to play sound effects (gun shots, footsteps, and so on) and use `Music` to play background music. Now that you understand the difference between the two, let's move on to some code.
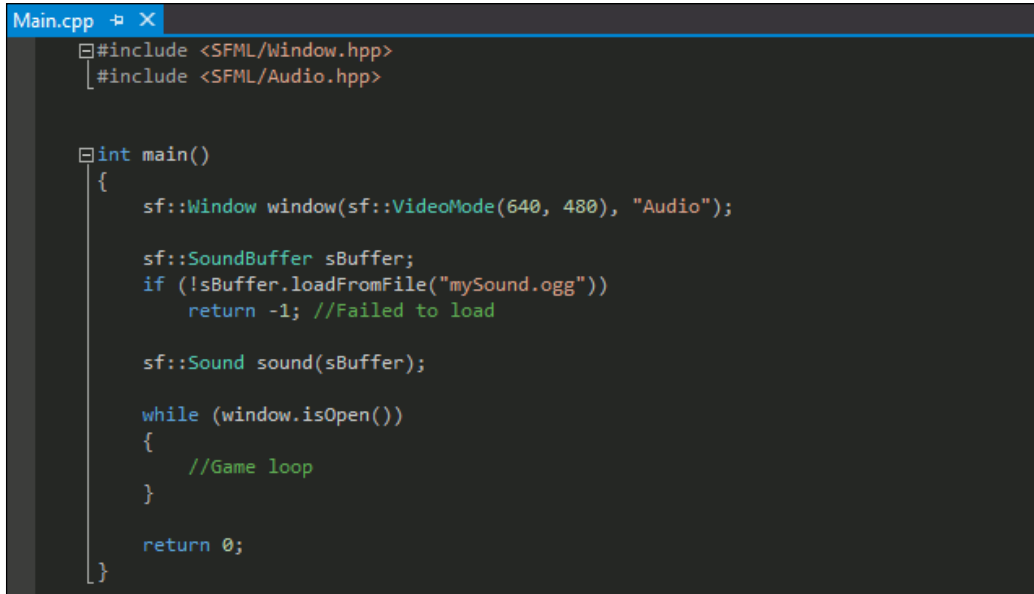
Audio in action

We will start with sound effects and move to music later on.

The sf::Sound class

A sound is composed of two classes, `Sound` and `SoundBuffer`. The `SoundBuffer` class is the resource in the memory and `Sound` is the wrapper that plays the resource. This should ring a bell, since it uses the same structure as `Sprite` and `Texture`—`Sprite` uses `Texture` as a resource. By designing it in such a way, multiple `Sound` instances can use the same `SoundBuffer` instance, reducing the amount of memory required significantly.

Enough with the explanations, here is how we create a sound:

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


int main()
{
    sf::Window window(sf::VideoMode(640, 480), "Audio");

    sf::SoundBuffer sBuffer;
    if (!sBuffer.loadFromFile("mySound.ogg"))
        return -1; //Failed to load

    sf::Sound sound(sBuffer);

    while (window.isOpen())
    {
        //Game loop
    }

    return 0;
}
```

We use the same format to load `Texture` as we do here with `SoundBuffer`—check whether the loading has failed and do something accordingly. In this case, without a sound buffer, it doesn't make sense to continue with the program execution, so we will terminate it. Some of the common supported audio formats are: OGG, WAV, FLAC, and so on. It is important to note that MP3 is not supported due to licensing issues.

Similar to `sf::Texture`, sound buffers have methods to load from memory (`SoundBuffer::loadFromMemory()`) and to load from a stream (`SoundBuffer::loadFromStream()`). A texture can also be loaded from an array of pixels but, in terms of sound buffers, this doesn't really make sense. Instead, the `SoundBuffer` class has a method of loading from an array of samples (`SoundBuffer::loadFromSamples()`).

As soon as we create `sf::Sound` by calling its constructor and passing the sound buffer, we can play it with `Sound::play()`. The method does different things, depending on the current status of `Sound`. The method plays the sound in another thread, so the current thread isn't blocked.

Every `SoundSource` object has `SoundSource::Status` (enum) associated with it. It can be one of three states: **Stopped**, **Paused**, or **Playing**. This state system is used internally in the `SoundSource` class to control the behavior of its methods. It is possible to get the status of a `Sound` or `Music` object by calling their `getState()` methods.

Going back to `Sound::play()`, in a state of stopped or paused, the method starts playing the sound from its current playing position. If it was in a playing state though, it would restart the sound. This means that we have to be careful when we want to play the same sound multiple times. The best solution to this is to create multiple `Sound` instances each time the sound needs to be played. Since all of these instances use the same `SoundBuffer` instance, creating sounds is extremely quick and lightweight so we do not have to worry about performance or memory.

The `Sound` class has methods for stopping and pausing the sound as well— `Sound::stop()` and `Sound::pause()`, respectively. The `Sound::stop()` method stops the sound and resets the playing position to the start if it is currently in a playing or paused state. If the sound is in a stopped state, the method does nothing. Similarly, `Sound::pause()` stops the sound playback, but doesn't reset the playing position. If the sound is in a stopped or paused state, the method does nothing.
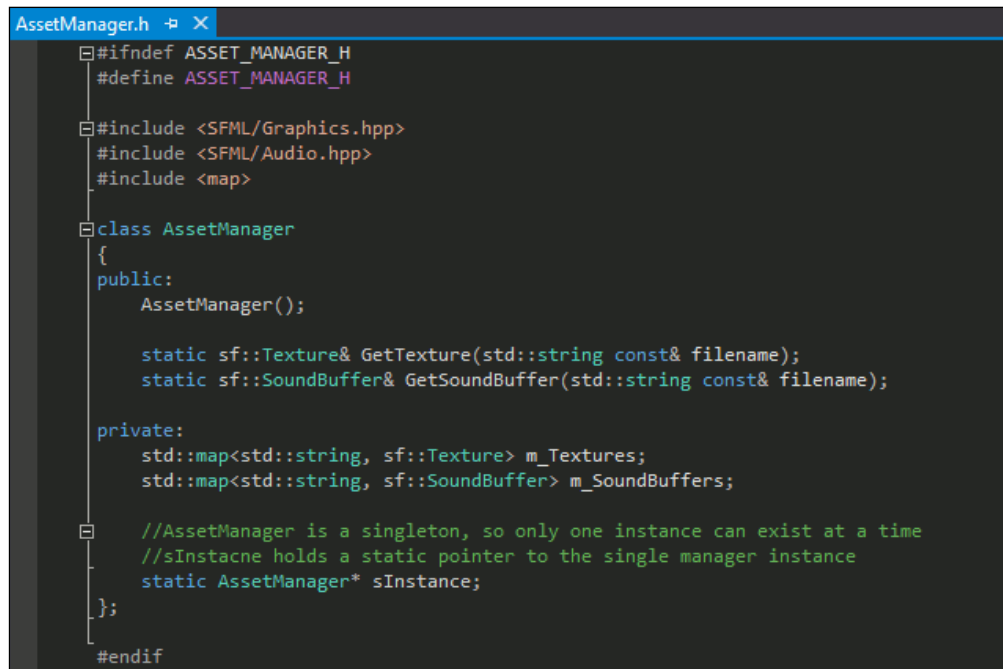
We can set if the sound should be loopable or not, with `Sound::setLoop()`. If the sound is set to loop, then it will restart from the beginning at the end. By default, each sound is not loopable.

The `sf::Sound` class has another feature, which is utilized by calling `Sound::setPlayingOffset()`. The method takes `sf::Time` and sets the playing position to that time. It can be used in both paused and playing state.

There is something important that we need to talk about in more detail and that is resource management. As it is, with the `Texture` and `Sprite` classes, the `SoundBuffer` instance has to be alive and stay in the same memory location while `Sound` is using it. To easily manage the lifetime of a `sf::Texture` object, we used our `AssetManager` class, and we will do the same for `SoundBuffer`.

Introducing AssetManager 2.0

Our old asset manager loads and holds only textures. We want to improve that by adding support for `SoundBuffer` objects. Both classes are similar when it comes to loading them, so our code for `SoundBuffer` looks almost the same as the one for the textures. This is what our updated `AssetManager.h` file looks like:

```cpp
AssetManager.h ⊬ ✕
#ifndef ASSET_MANAGER_H
#define ASSET_MANAGER_H

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

class AssetManager
{
public:
    AssetManager();

    static sf::Texture& GetTexture(std::string const& filename);
    static sf::SoundBuffer& GetSoundBuffer(std::string const& filename);

private:
    std::map<std::string, sf::Texture> m_Textures;
    std::map<std::string, sf::SoundBuffer> m_SoundBuffers;

    //AssetManager is a singleton, so only one instance can exist at a time
    //sInstacne holds a static pointer to the single manager instance
    static AssetManager* sInstance;
};

#endif
```

We use the audio module to load sound buffers so we need to include the audio module. In `AssetManager`, in addition to the code which handles textures, we can see a map to store sound buffers and a method to load and retrieve them—exactly the same approach as for textures.

Let's take a peek at the implementation of `AssetManager::GetSoundBuffer()`:

```cpp
sf::SoundBuffer& AssetManager::GetSoundBuffer(std::string const& filename)
{
    auto& sBufferMap = sInstance->m_SoundBuffers;

    auto pairFound = sBufferMap.find(filename);
    if (pairFound != sBufferMap.end())
    {
        return pairFound->second;
    }
    else
    {
        //Create an element in the SoundBuffer map
        auto& sBuffer = sBufferMap[filename];
        sBuffer.loadFromFile(filename);
        return sBuffer;
    }
}
```

The preceding code has the same structure as the code for
`AssetManager::GetTexture()`. First, we try to find whether a sound buffer
with that filename already exists and, if it does, we return it. If it doesn't exist,
we create an entry in our sound buffer map, load the buffer from a file, and
return the new buffer.

Outside `AssetManager`, we can create sounds by simply doing the following:

```cpp
int main()
{
    sf::Window window(sf::VideoMode(640, 480), "Audio");
    //Remember, we need an instance of the asset manager
    AssetManager manager;

    sf::Sound sound(AssetManager::GetSoundBuffer("mySound.ogg"));
    sound.play();

    while (window.isOpen())
    {
    }

    return 0;
}
```

By using our asset manager, we can be sure that our resource will not be accidentally deleted or moved in memory. Also, we can cache resources with the same filename so that when we want to get the same resource in multiple places, we just have to get the one which is already loaded, which will save us a lot of memory in the process.

The time has come for us to look at what the `Music` class has to offer.

sf::Music and sf::SoundStream

Playing an audio file using the `Music` class is as simple as:

```
sf::Music music;
if (!music.openFromFile("myMusic.ogg"))
    return -1;
music.play();
```

When we open a music stream from a file, the supported audio formats are the same as they are for `Sound`. Also, `Music::play()` launches the sound in another thread, so we don't have to worry about blocking the current thread.

A music file can also be streamed from a loaded file in memory (`Music::openFromMemory()`) or from `InputStream` (`Music::openFromStream()`).

Note how we *open from file* rather than *load from file* (as it is with `SoundBuffer`). The music class derives from a class called `SoundStream`, which implements a common behavior to stream audio. It doesn't load all of the data in system memory, but it loads small chunks of the whole asset and works with them. When it reaches an area of the audio that is not in the memory, it requests more data from the stream and starts working with that. The `Music` class works on top of that interface and just provides methods to open a stream (with the `Music::open*()` methods) and feeds these to the `SoundStream` beneath.

Since we are dealing with a data stream, the data cannot be destroyed while `SoundStream` is using it—when we use data which has already been allocated in system memory, we have to be sure to keep it alive.

As mentioned earlier, the `Music` class doesn't provide much functionality over `SoundStream`, apart from methods which open file streams. All the interesting functionality is located in the base class, `SoundStream`, where methods such as `SoundStream::play()`, `SoundStream::pause()`, and `SoundStream::stop()` are located. These three methods work in the exact same way as their cousins from the `Sound` class. The `SoundStream::setPlayingOffset()` and `SoundStream::setLoop()` methods are here as well.

With regards to `AssetManager`, we don't have to add an entry for `Music` objects, since that class is relatively lightweight and its internal resources are not reusable like `SoundBuffer`. If we were to use `Music::openFromMemory()` to load our music files, we would add these resources to the asset manager as well. However, this option is rarely used, since loading the whole file is memory expensive.

Now, it's time to look at sound in 3D.

sf::SoundSource and audio in 3D

As discussed earlier, the `Sound` and `SoundStream` classes are derived from `SoundSource`. First we will talk about some general features, and then give examples of how to use sound spatialization (in other words, 3D sounds).

Common audio features

As we create more and more complex scenes, we want more from our sounds and music. For example, some sounds might be louder than others, and we would like to lower their volume when we play them in the game. We can do that with `SoundSource::setVolume()`. The supported volume values are from 0 (mute) to 100 (full volume), and every sound source starts with a default value of 100. We can get the current volume with `SoundSource::getVolume()`.

Another characteristic of sound is pitch. The pitch is the perceived frequency of a sound. For example, a *C4* note has a lower pitch than a note of *E4*. We can change the pitch factor of the original audio source by calling `SoundSource::setPitch()`. This is an artificial method to increase the pitch, which results in speeding/ slowing the speed of the playback. The default value of the factor is `1`, which doesn't alter the pitch of the original sound. We can get the current pitch by calling `SoundSource::getPitch()`.

Audio in 3D

Each sound that we've used so far has been played at full volume (assuming that we haven't altered the volume with `SoundSource::setVolume()`). In other words, they weren't spatialized. In this section, we will talk about how to place sounds in the world so that they are played relative to a sound listener (the main character, for example). By doing this, we can simulate a realistic sound environment, where, if something explodes on our character's left, we hear the explosion only in our left speaker.

Before we go any further, it is important to note that a sound can be spatialized only if it has a single channel (a mono sound). Spatializing sources with multiple channels makes little sense since they explicitly define how to use the speakers.

Creating a sound environment requires two things: someone or something to play the sounds and someone or something to listen to them. In the previous sections of this chapter, we talked about how to play sounds and music. However, those sounds were not spatialized and thus did not require a listener—they just played at full volume in the speakers. If we want to place them in a world and create a realistic environment, we have to set up a listener who actually has the ability to hear them. This enables features such as sound direction and attenuation when the sound is played.

Setting up the listener

At any moment in our program's execution, we need (and have) only one listener for, hearing sounds. SFML provides a static class to manipulate the audio listener—`sf::Listener`. The listener has three properties which we can manipulate—position, orientation, and global volume.

The position of a listener is set by calling `Listener::setPosition()`. This function takes a three-dimensional vector because SFML doesn't assume that we are building a flat game. If we are building a 2D game though, our world would be flat and we wouldn't need all three axes. We can use the conventional x and y axes and leave z = 0 everywhere. We don't have to use this convention but, in most cases, following these conventions brings benefits—people understand the code and common examples are more relevant.

So, if we were to have a sprite as our main character, we would have to set the listener's position to the sprite's position each time it moves:

```cpp
sf::Sprite heroSprite(AssetManager::GetTexture("myHero.png"));

while (window.isOpen())
{
    /* Update the hero Position here */

    //Set the listener to the hero's position
    sf::Vector2f heroPos = heroSprite.getPosition();
    sf::Listener::setPosition(heroPos.x, heroPos.y, 0);
}
```

Apart from the position, we can also set the orientation of our listener by calling `Listener::setDirection()`. This defines the direction which the listener is facing. If we are talking about a top-down shooter, for example, the main character can rotate 360 degrees. The `Listener` can handle such behavior with the `Listener::setDirection()` function. It expects a 3D vector which represents the direction of the listener. Since this is just a direction, the vector passed should be normalized, but SFML doesn't require it.

Sticking with our example of the hero sprite, we can calculate the direction vector for the listener as well:

```cpp
#define PI_RADIANS 3.1415f
#define PI_DEGREES 180.f

sf::Sprite heroSprite(AssetManager::GetTexture("myHero.png"));

while (window.isOpen())
{
    /* Update the hero Position here */

    //Transform the rotation to radians
    float heroRot = heroSprite.getRotation() * PI_RADIANS / PI_DEGREES;
    //Set the listener's direction from the hero's rotation
    sf::Listener::setDirection(std::cos(heroRot), std::sin(heroRot), 0);
}
```

The preceding example uses the common formula of transforming an angle to a 2D vector direction by using the cosine and sine values of that angle. It is important to note that the angle needs to be in radians for the trigonometric functions to work. That's why we transform the angle from degrees to radians using the formula:

$$\boxed{Angle\_Radians = Angle\_Degrees * PI\_Radians / PI\_Degrees}$$

The final property which can be set to the `Listener` global volume. This can be done by calling `Listener::setGlobalVolume()`. The function expects `float` in the range [0…100]. It combines that value with the volume from each individual sound to calculate the final volume.

To finish the code for the listener, there are `Listener::get*()` functions for each of the properties covered in this section.

Audio sources

We covered the first part of the sound environment—the `Listener` actor, which is responsible for *listening* to the sounds being played in the environment. However, without any sounds being played, there is little point in having `Listener`. In this section, we will explore the functionality of the `SoundSource` class with regards to sound spatialization.

Every mono `SoundSource` is spatialized by default. If we do not touch any of the properties of `Listener` and `SoundSource` they remain at the same place, and it will appear that the sources play at full volume. However, once we start moving `Listener` from the origin (0; 0; 0), we start getting sounds which are fading away and disappearing. This is not ideal if we just want to play a non-spatialized sound. We will explore ways of dealing with this issue in this section.

The most important property of any `SoundSource` instance is its position. The position is the main factor which determines how loud and from which direction the source will be played. We can set the position of `SoundSource` by calling `SoundSource::setPosition()`. This is very similar to the `Listener` class as it expects a 3D vector. We will use the *z = 0* convention here as well:

```cpp
sf::Sprite zombie(AssetManager::GetTexture("zombie.png"));
sf::Sound growl(AssetManager::GetSoundBuffer("growl.ogg"));

/*Update zombie's position here*/

//Update sound's position
sf::Vector2f zombiePos = zombie.getPosition();
growl.setPosition(zombiePos.x, zombiePos.y, 0);
```

The example in the preceding image looks quite similar to the `Listener` example, with the important difference that there can be multiple sounds in the world, and thus each sound's position needs to be set separately. As a side note, in this example, the sound is not linked to the sprite in any way, and we need to make sure to update the position of `growl` each time the sprite moves to keep the sound coming from the correct place.

Also, we can set a **boolean** which determines whether the position is relative to the position of `Listener`. By default, the position of each sound is absolute, but we can make it relative to `Listener` by calling `SoundSource::setRelativeToListener()`. This is useful when we want to play a sound at full volume (footsteps, cloth rustling, gunshots, and so on). To do that, we just place the sound at the origin (0; 0; 0), and set it relative to the listener. This always plays the sound on top of the listener.

Apart from the position, each `SoundSource` has a property called *minimum distance*. This represents the furthest distance from the listener, from which the sound can be heard at its maximum volume. For example, if we have a sound that is 40 units away from the listener and it has a minimum distance of 45, it will be played at full volume. However, if the source has a minimum distance lower than 40, it will be played with a faded volume, which depends on the attenuation factor (discussed in the following paragraph). We can set the minimum distance of `SoundSource` by calling `SoundSource::setMinDistance()`. It expects a `float` parameter, which represents the distance in world coordinates (the value of `0` is forbidden). The default value of the minimum distance is 1.

The final property that we can set to a sound source is attenuation. Attenuation is the factor which determines how fast the sound fades away when it is beyond its minimal distance from the listener. For example, a factor of 1 fades the sound very slowly whereas, with a factor of 100, the sound has no transition period between its maximum volume and 0. A factor of 0 indicates that the sound is either heard by the listener at a maximum volume (if it's in range) or it's just not played (outside the sound's minimum distance). Attenuation can be set by calling `SoundSource::setAttenuation()`. The default value of the attenuation is 1.

Summarizing audio features

After all this talk about spatialization, it is a good idea to solidify everything by providing an example that covers it all. We will create a simple yet effective world with one listener and one audio source, which we can control with the mouse (position is controlled by mouse coordinates, and the sound is played with a mouse button click.) Let's start with the setup—one listener and one sound with appropriate visual indicators (`CircleShape`):

```cpp
sf::RenderWindow window(sf::VideoMode(640, 480), "Audio");
AssetManager manager;

//Listener at the center of the window
sf::Listener::setPosition(window.getSize().x / 2.f, window.getSize().y / 2.f, 0);
//The listener is facing UP (-Y)
sf::Listener::setDirection(0, -1, 0);

//Shape for the listener (world representation)
sf::CircleShape shapeListener(20);
shapeListener.setFillColor(sf::Color::Red);

sf::Sound sound(AssetManager::GetSoundBuffer("mySound.ogg"));
//Sound will start to fade away from the listener when it's
//more than 160 pixels away from the listener ( 640 / 4 = 160)
sound.setMinDistance(window.getSize().x / 4.f);
//Sound will fade quite quickly, once it passes the 160 pixel boundary
sound.setAttenuation(20.f);

//Shape for the sound (world representation)
sf::CircleShape shapeSound(10);
shapeSound.setFillColor(sf::Color::White);
```

Since we are using `AssetManager` to load the sound buffers, it is important to create an instance of it right after we create the window. Next, we place the listener at the center of the window, and set it to face toward the top-border of the window (the *y* direction). We also create a shape to display at the position of the renderer in the *world*. This will give us a visual indication of where exactly the shape stands. After that, we create the sound and set its minimal distance and attenuation properties. To finish off the initialization, we create a shape for the sound as well.

With that done we can move to our game loop. The first thing we want to do is handle events:

```cpp
//Handle events
sf::Event ev;
while (window.pollEvent(ev))
{
    //Close window on close button click
    if (ev.type == sf::Event::Closed)
        window.close();
    //Play the sound on mouse button click
    else if (ev.type == sf::Event::MouseButtonPressed)
        sound.play();
}
```

We can see the close window implementation on a `Closed` event. We want to do something on the click of a mouse button though—play the sound. This is done easily enough by calling `Sound::play()`.

Now, let's look at the update frame:

```cpp
//Get 2D listener position
sf::Vector2f listenerPos(sf::Listener::getPosition().x, sf::Listener::getPosition().y);
//Set listener position (constant for this example)
shapeListener.setPosition(listenerPos);

//Set sound position
sf::Vector2f soundPos(static_cast<sf::Vector2f>(sf::Mouse::getPosition(window)));
sound.setPosition(soundPos.x, soundPos.y, 0);
shapeSound.setPosition(soundPos);
```

The first two lines are there to account for a situation in which the `Listener` changes position in each frame. However, in this particular example, the listener is stationary and, as such, we can opt to omit them. The next four lines move the sound to the position of the mouse (relative to the current window). Note that we have updated the sprite as well.

The final task in the example is rendering the scene. We only have two shapes so that should be an easy enough job:

```
//Render frame
window.clear();

window.draw(shapeListener);
window.draw(shapeSound);

window.display();
```

With all that done, when we run the program, it should display two circles and one of them should be moving with the mouse—that is our sound entity. To play a sound, we simply click on any of the mouse buttons. Try to experiment with different distances between the sound and the listener to get a feel of how the fading works. Also, changing the attenuation and the minimum distance produces different effects. Feel free to experiment with the code until you are comfortable with the concept of sound spatialization.

That concludes our discussion on sounds and music. Text is what we are going to talk about next. It is important in its own way since we need a way to display in-game item statistics and character titles.

Getting started with sf::Text

Text is one of the most overlooked *features* of a game, but it is sometimes essential to the gaming experience—for example, using the correct font and size, displaying the appropriate amount of information, and so on. Yes, there are games where text is not used at all, but those are quite rare. If you make a game, you'll need to render text at some point during the development process (even for debugging data). In this section, we will talk about text and fonts.

Remember the `Sprite`—`Texture` and `Sound`—`SoundBuffer` relation? Take a guess which other part of SFML uses the same model. Yes, you're right—the `sf::Font` and `sf::Text` classes. In this case, the `Font` class is the resource which we want to keep safe while the `Text` class uses it. You should probably be able to picture the process of loading fonts and creating texts based on the fact that this is the third time that we've come across it. If you can't—don't worry, this is what this book is for. Let's start by creating a simple text label:

```cpp
sf::Font font;
//Try to load a font and exit if there was an error
if (!font.loadFromFile("awesomeFont.ttf"))
    return -1;

sf::Text text("Look at my awesome font.", font);
```

Once we have an instance of `Text`, we can draw it by calling the `RenderWindow::draw()` method. In fact, like the `Sprite` and `Shape` classes, the `Text` class inherits from `sf::Drawable` and `sf::Transformable` so we can transform it in the same manner by changing its position, rotation, scale, and origin.

We can set a few text-specific properties on a `Text` object. One of the important ones is the character size. This property determines how tall the characters are (in pixels) and it's set by calling `Text::setCharacterSize()`. We can also specify a third argument in the constructor which changes the character size with the creation (the default value is 30).

Additionally, we can change the string, which is displayed by the text object. This is done through `Text::setString()`. It expects a `sf::String` object, which has implicit conversion constructors from `std::string` and `std::wstring`. So, passing any of these arguments works absolutely fine:

```cpp
sf::String someString;

/*Fill the 'someString' variable here*/

text.setString(someString);
text.setString("This is a normal string");
text.setString(L"This is a wide-char string");
text.setString(std::string("This is a normal string"));
text.setString(std::wstring(L"This is a wide-char string"));
```

The color of the `Text` object can be set by calling `Text::setColor()`. Also, we can set the style of the font—`Text::Style::Regular`, `Text::Style::Bold`, `Text::Style::Italic`, and `Text::Style::Underlined`. This is done with the `Text::setStyle()` method. The expected argument is a bitmask of the elements in the `Text::Style` enum. For example, this produces bold text, which is underlined:

```
text.setStyle(sf::Text::Bold | sf::Text::Underlined);
```

An interesting feature of the `Text` class is the ability to get the global position of a specific character in the text. This is done with `Text::findCharacterPos()`. It expects the character's index in the string and returns its position, accounting for translation, rotation, scale, and origin. This is useful in instances where we want to place a graphic on top of a character, or as an indication of the cursor position in a text input box.

All the properties that can be set to a `Text` object also have *getters* with the common signature, `Text::get*()`.

Now let's talk about fonts. Since they are a resource which has to be loaded from a file, they need to be alive as long as a `Text` object is using them. Fortunately, we have the `AssetManager` which does exactly that for all of our resources. It is time for the next iteration on `AssetManager`.

AssetManager 3.0

Adding an entry for the `Font` class to our existing code will not look much different from the other two resources—`Texture` and `SoundBuffer`. We first need to add `std::map` and then a single method, which loads and caches the resource. This is what our header file looks like after we add the map and the method:

```cpp
class AssetManager
{
public:
    AssetManager();

    static sf::Texture& GetTexture(std::string const& filename);
    static sf::SoundBuffer& GetSoundBuffer(std::string const& filename);
    static sf::Font& GetFont(std::string const& filename);

private:
    std::map<std::string, sf::Texture> m_Textures;
    std::map<std::string, sf::SoundBuffer> m_SoundBuffers;
    std::map<std::string, sf::Font> m_Fonts;

    //AssetManager is a singleton, so only one instance can exist at a time
    //sInstacne holds a static pointer to the single manager instance
    static AssetManager* sInstance;
};
```

We are essentially following the same format as we did with the `Texture` and `SoundBuffer` resources. `AssetManager::GetFont()` doesn't look too different either:

```cpp
sf::Font& AssetManager::GetFont(std::string const& filename)
{
    auto& fontMap = sInstance->m_Fonts;

    auto pairFound = fontMap.find(filename);
    if (pairFound != fontMap.end())
    {
        return pairFound->second;
    }
    else
    {
        //Create an element in the Fonts map
        auto& font = fontMap[filename];
        font.loadFromFile(filename);
        return font;
    }
}
```

The process, this method follows is quite simple—if there is already a font with that filename, return it, else, create a new entry in the map and load it from the file. The supported font file formats are: TrueType (TTF), Type 1, CFF, OpenType, SFNT, X11 PCF, Windows FNT, BDF, PFR, and Type 42. The most commonly used formats are TrueType (TTF) and OpenType (OTF). The OpenType format is built similar to the TrueType format, but it has a few more benefits, including a larger character set. It is important to note that we need to have the font in the working directory of the project, or we need to specify the complete path as the filename. SFML cannot load system fonts directly, so we need to use font files—simply calling `font.loadFromFile("Arial")` does not work, as the function expects a file and not a system font.

Once we get the `AssetManager` functionality to load fonts, using it is as simple as ever:

```cpp
sf::RenderWindow window(sf::VideoMode(640, 480), "Audio");
AssetManager manager;

sf::Text text("Look at my awesome font.", AssetManager::GetFont("awesomeFont.ttf"));
```

Remember, when we are using `AssetManager`, we first need to instantiate it.

With that, we finish our little topic of discussion on sounds, text, and fonts. Enjoy the new toys.

Summary

We talked about quite a lot of things in this chapter. We started off by exploring a whole new module of SFML—the audio module, covering sounds, music, as well as general sound sources. Later on, we introduced the concept of sound spatialization and ways of implementing it with the help of SFML. The last section of the chapter was dedicated entirely to text and its resources-fonts. We also updated our asset manager to handle the new resources.

At the end of the day, our time was well spent. The final two chapters of the book introduce more advanced topics, such as shaders and networking, so put your thinking hats on, and let's dive in.

6
Rendering Special Effects with Shaders

Welcome to the *advanced* section of the book. These last two chapters introduce concepts, such as shaders and networking, which are difficult to understand for an inexperienced developer and might even trip up an experienced one. We will explain what they are and how they are represented in SFML, but we will not cover them in detail. They are large topics on their own with many books and tutorials covering the subject.

The main focus of this chapter is **shaders**. Shaders are typically the tools which we use to create interesting graphical effects, do anti-aliasing, optimize the graphics pipeline, and so on. They are essentially programs that run on the GPU and work with vertex and fragment (pixel) data. We will also talk about rendering directly onto a texture and using that texture to generate our final image on the screen.

In this chapter we will cover the following:

`sf::RenderTarget` and `sf::RenderWindow`

- Rendering directly to a texture
- Shader programming
- Combining everything

sf::RenderTarget and sf::RenderWindow

We are already familiar with the `RenderWindow` class, but we haven't covered it in detail. One reason is that we don't need to know too much besides its `RenderWindow::draw()` method to work with it in the early stages. However, now that we are going to talk about rendering on different targets (textures in this instance), it is important to understand what `RenderWindow` is.

We will start with the introduction of the concept of a render target. A render target is an object that we can use the graphics module to draw on. In order for a class to be considered a render target, it has to inherit from `sf::RenderTarget`. The class provides the interface needed in the child classes for drawing `Drawable` objects, using `Views` (or cameras), shaders, and so on. The `RenderWindow` class is one of those children classes.

The inheritance tree of the `RenderWindow` class looks like this:
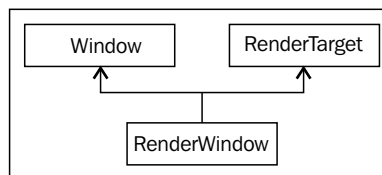
From this, we can get an understanding of what the class represents—a way for us to draw a `Drawable` object in a window. If we look at the base `Window` class we see no drawing functionality in it, only window related properties—such as handling events, changing size, title, position, and so on. In order to draw something in the base `Window` class, we have to use OpenGL. However, every class that implements the `RenderTarget` methods can serve as a canvas for all the features of the graphics module—most importantly the `Drawable` objects. And that is the reason why `RenderTarget` is so important—without the functionality that the class provides, we cannot utilize most of the module's features.

Most of the `RenderWindow` methods we have talked about before are actually declared in `RenderTarget`—`draw()`, `setView()`, `clear()`, `mapPixelToCoords()`, and so on. Some of those methods are implemented differently depending on `RenderTarget`, but most of them work in the same way. Before we go any further there is an important note which we have to mention about the `RenderTarget::draw()` method.

We have talked about the `RenderTarget::draw()` method in previous chapters, but we haven't yet mentioned the second (optional parameter) - an instance of `sf::RenderStates`. The class holds four values, which can be defined for each draw call—`sf::BlendMode`, `sf::Transform`, `Texture*`, and `Shader*`. We can set those before passing the render states to the `RenderTarget::draw()` method, but some `Drawable` objects set them automatically (for example, the `Sprite` class combines its transform with the one given and sets its own texture). The class also has implicit constructors for each of its fields, so passing just one of those types works, without having to create the `RenderState` object explicitly. For example, the following code works just fine:

```
window.draw(sprite, sf::BlendMode::BlendAdd);
```

Knowing the basics of the `RenderTarget` class allows us to start talking about a different implementation of it, in the form of `sf::RenderTexture`.

Rendering directly to a texture

We have covered textures before. We even created an `AssetManager` that loads and stores textures. However, the only thing that we have done with them so far has been to place them on a `sprite` or a `shape`. As useful as that is, textures can also be used in different ways in graphics programming. That is typically done in shaders where we have a lot of freedom to use every single texel (pixel of the texture) as we wish. We are going to talk about using textures in shaders later on in the chapter. We will now explore a different way of creating a texture—by directly rendering `Drawable` objects on it. In this case `RenderTexture` provides the functionality that we need.

The `RenderTexture` class inherits from `RenderTarget` and implements all of its drawing functionality using a **framebuffer** object (if it's available, otherwise it uses an alternative method). A framebuffer object is an OpenGL object, which allows rendering to an off-screen buffer. Using this technique, SFML is able to achieve seamless behavior so that we are in fact rendering directly to a texture. The `Texture` object itself is held as a field inside the `RenderTexture` class and we can get it by calling `RenderTexture::getTexture()`.

In many cases it is useful to render on a texture and then do something with it. For example, rendering the entire scene to a texture allows us to do post-processing effects with shaders on it. In some cases we might want to create a more complex sprite texture for some of our entities. Rendering multiple shapes or sprites per entity every frame can become expensive, so `RenderTexture` allows us to improve performance by only rendering those once and using the resulting texture.

The way we can use a render texture is similar to `RenderWindow`. Firstly, we need to create it with appropriate dimensions (width and height) and then draw on it with the familiar routine:

```cpp
sf::RenderTexture rTexture;
rTexture.create(32, 32, /*Depth buffer enabled = */ false);

sf::CircleShape circle(16); //Circle radius = 16

//Render routine - clear -> draw -> display
rTexture.clear();

rTexture.draw(circle);

rTexture.display();

//RenderTexture::getTexture() gets a ref to the Texture object
sf::Sprite sprite(rTexture.getTexture());

//Use the sprite in any way we like
```

After we declare a `RenderTexture` variable, we have to call the `RenderTexture::create()` method to actually make a valid render texture. The method takes width and height parameters as well as an indicator if the render texture uses a depth buffer. In this case, we are only rendering a single circle shape on it, so no depth buffer is required. The depth buffer allows us to discard any fragments (pixels) that are rendered behind already existing fragments. As such, if we were rendering 3D models to a texture (using OpenGL) we would probably want the depth buffer enabled. In this example our single circle shape is with radius 16, so we want a texture with a width and height of 2xRadius = 32.

After we create our circle shape, we render it onto the texture. `RenderTexture::clear()` resets each texel of the texture to the given color. The color argument is optional though, with `Color::Black` being the default color. Next, we draw our shape onto the texture with the `RenderTexture::draw()` method and finally display everything that has been drawn so far. The `RenderTexture::display()` method updates each pixel of the texture with everything that has been done to it since the last time it was called.

Now that we have our texture ready, we can use it by getting the underlying `Texture` object with `RenderTexture::getTexture()`. The method returns a reference to the texture object stored as a field inside `RenderTexture`. In this case we opt to create a sprite out of it, which we can use later on for any purpose that a sprite can be used for.

Like the `Texture` object, `RenderTexture` has to stay alive as long as an object is using its internal texture. Otherwise the inner texture will get destroyed as well and the resource is lost.

Apart from the drawing functionality, the `RenderTexture` class has some of the `Texture` methods as well, such as `RenderTexture::setSmooth()` and `RenderTexture::setRepeated()`. Also, if we want to start rendering to a render texture using OpenGL, we have to call `RenderTexture::setActive()` to enable the context for the OpenGL API calls. Otherwise, all the OpenGL calls will affect the previously enabled context (in most cases it will be the main window).

We are done talking about the `RenderTexture` objects for now. We will come back to them later on, when we have a firm understanding of shaders.

Shader programming

Shader programming has a long history and it exists in many variations. Interesting as they are, this book is not enough to cover even a small portion of the topics in shader programming. In this section, we will briefly mention what shaders are and what support SFML has in terms of loading and using them, but we will not go into too much detail. If you are interested in learning more about them, it is worth getting another book with the GLSL language as its core topic or following an online guide. I recommend the **GLSL 1.2 Tutorial** at Lighthouse3d.com - `http://www.lighthouse3d.com/tutorials/glsl-tutorial/`.

What is a shader?

Shaders come in different forms and use different languages, depending on their API but, ultimately, a shader is a program which executes on the GPU. And, as with any program, a shader requires an input and produces an output. The input and output of a shader depends on the shader's type. We will only cover the shader types that are supported by the `Shader` class in SFML and make sense in a 2D sprite-based context, the Vertex and Fragment (Pixel) shaders. We can visualize the graphics pipeline from the moment we want to render something to the actual writing on the render target with the following diagram:

```
┌─────────────────────────────────────────────────┐
│   ┌──────────────────┐                           │
│   │    Draw call     │                           │
│   └──────────────────┘                           │
│            │          Transfer all vertex data   │
│            ▼          along with any parameters   │
│   ┌──────────────────┐                           │
│   │  Vertex Shader   │   Progress each vertex     │
│   └──────────────────┘                           │
│            │          Output vertex data          │
│            ▼                                      │
│   ┌──────────────────┐   Use vertex data to      │
│   │ Fragment Shader  │   create each pixel        │
│   └──────────────────┘                           │
│            │          Output pixel data           │
│            ▼                                      │
│   ┌──────────────────┐   Update render target    │
│   │  Render Target   │   with pixel data          │
│   └──────────────────┘                           │
└─────────────────────────────────────────────────┘
```
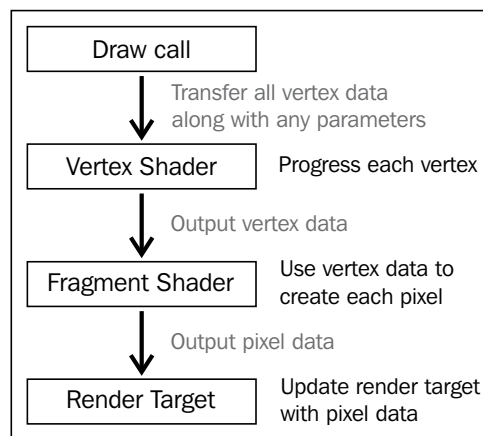
Each time that we call the `RenderTarget::draw()` method, OpenGL goes through the steps described in the preceding diagram. Firstly, we pack all the vertices (position, color, texture coordinate) in a package and send it over to be stored in GPU memory. When we consider a sprite, there are four vertices for the four corners of the rectangle. After getting the vertices in GPU memory, OpenGL runs the Vertex shader program on each of those vertices. We can then manipulate any of them in any way we like. More often though, a common shader is used where it just transforms each vertex with the Model, View, and Projection matrices. Those matrices represent the transformation (position, rotation, and scale) of the model, the position and orientation of the camera, and the camera's projection. After we transform each vertex, OpenGL calculates which fragments have to be painted on the target. This process is called as **rasterization**. Looking at the sprite example again, a 16x16 sprite with a scale of (1, 1) and a zero angle rotation has exactly 16x16 = 256 pixels. After OpenGL calculates all the pixels (or fragments), it calls our Pixel (Fragment) shader on each of them. The result of that shader is a color, which can be placed in the relevant fragment on the render target.

That ends the introduction section on the topic of shaders. Next we will talk about loading and using shaders.

Loading shaders

The first step is making sure that the current GPU supports shaders. OpenGL introduced shaders in its 2.0 version, so there are still graphics cards which do not support them (however it's safe to assume the large majority do). To check if shaders can be used on the current system we can use the static function `Shader::isAvailable()`:

```
if (!sf::Shader::isAvailable())
    return -1; //Shaders are not supported. Abort!
```

The first step in using a shader is to load and compile it. SFML does the compiling part internally, so we only need to worry about supplying the shaders. Shaders can be loaded in three ways—from a file, a string, or a stream. It uses the same format as other resources in SFML. Here is how we create a shader program by loading the vertex and fragment shaders:

```
sf::Shader shader;
if (!shader.loadFromFile("vertShader.vert", "fragShader.frag"))
    return -1;

//Use the shader
```

Since SFML uses OpenGL, the shaders have to be written in **OpenGL Shading Language** (**GLSL**). `Shader::load*()` also tries to compile the shader. If any problems arise, an error message appears in the console with the issue explained. the method also returns `false`.

Another way of loading shaders is to store them directly in the code and load them as strings:

```
std::string vertShader =
    "void main() {" \
    "gl_Position = gl_Vertex;" \
    "}";
std::string fragShader =
    "void main() {" \
    "gl_FragColor = vec4(1, 0, 0, 1);" \
    "}";

sf::Shader shader;
if (!shader.loadFromMemory(vertShader, fragShader))
    return -1;
```

Since shaders have to be loaded from a file or memory (a file is recommended), it is important that we don't have to load them several times. If this sounds familiar, it is because we used this technique before with our `AssetManager` class. Shaders can be considered as another resource that the manager needs to keep track of. Furthermore, we can get the cached shaders from anywhere in the program.

AssetManager 4.0

We have now added new resources to our manager several times. For shaders, the structure does not change much, with the exception that a shader program is defined by two filenames (*vertex* and *fragment*), rather than one (a texture for example).

First of all, we add our map and the `AssetManager::GetShader()` method:

```cpp
class AssetManager
{
public:
    AssetManager();

    static sf::Texture& GetTexture(std::string const& filename);
    static sf::SoundBuffer& GetSoundBuffer(std::string const& filename);
    static sf::Font& GetFont(std::string const& filename);
    static sf::Shader* GetShader(
        std::string const& vsFile,
        std::string const& fsFile);

private:
    std::map<std::string, sf::Texture> m_Textures;
    std::map<std::string, sf::SoundBuffer> m_SoundBuffers;
    std::map<std::string, sf::Font> m_Fonts;
    std::map<std::string, std::unique_ptr<sf::Shader>> m_Shaders;

    //AssetManager is a singleton, so only one instance can exist at a time
    //sInstacne holds a static pointer to the single manager instance
    static AssetManager* sInstance;
};
```

Note that for the `AssetManager::GetShader()` we return a pointer to the shader, rather than a reference. This is for convenience, since SFML requires a pointer when we want to use it. We don't store `Shader` instances as a value, but we actually use a smart pointer. The reason for this that `Shader` instances cannot be copied (which is necessary for the map container). This is what the method's implementation looks like:

```cpp
sf::Shader* AssetManager::GetShader(
    std::string const& vsFile,
    std::string const& fsFile)
{
    auto& shaderMap = sInstance->m_Shaders;

    //The key to be stored in the map
    auto combinedKey = vsFile + ";" + fsFile;
    auto pairFound = shaderMap.find(combinedKey);
    if (pairFound != shaderMap.end())
    {
        return pairFound->second.get();
    }
    else
    {
        //Create an element in the Shader map
        auto& shader = (shaderMap[combinedKey] = std::unique_ptr<sf::Shader>(new sf::Shader()));
        shader->loadFromFile(vsFile, fsFile);
        return shader.get();
    }
}
```

Since we have two filenames we need to create a single key for our map. The easiest solution is to concatenate them with a separator in between (**;** in this case) and use that as our key. In this case we will avoid most conflicts that can happen using different combinations of shader filenames. If we want a more robust solution we can create a custom `struct`, which holds the two filenames as strings and use it as the key. The rest of the code is the same as for the other resources, with the exception that we use and return pointers to the shaders rather than references.

Using shaders

Now that we have a method of easily loading and caching shaders, using it is straightforward:

```cpp
auto shader = AssetManager::GetShader("vertShader.vert", "fragShader.frag");

sf::Sprite sprite(AssetManager::GetTexture("myTexture.png"));

while (window.isOpen())
{
    window.clear();

    window.draw(sprite, shader);

    window.display();
}
```

We can also use shaders with `RenderTexture` objects in the same way.
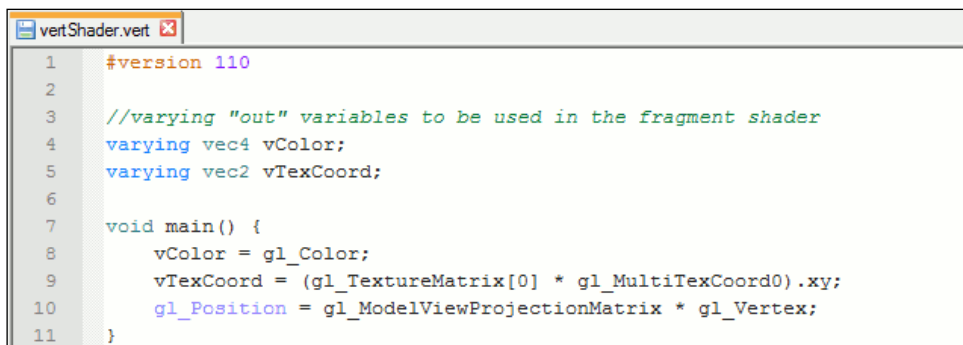
Setting shader uniforms

Most shaders have uniforms as a method of controlling some of the properties of the shader. A uniform is a variable inside the shader that can be set from outside the shader. We can set these uniforms by the name `Shader::setParameter()` and the value of the uniform's type. The name has to match the name of the uniform inside the shader. For example, if we want to set a 2D vector uniform, we call:

```
shader->setParameter("uSpecialVector", sf::Vector2f(3, 3));
```

The shader class has overloads of the `Shader::setParameter()` method for a number of uniform types. The supported types are: `float`, 2D vector, 3D vector, 4D vector, `Color`, `Transform` (matrix), and `Texture` (sampler).

Suppose we want to make a simple ripple shader, which distorts the sprite for a period of time. We will play with texture coordinates, rather than vertices, because it's simpler and more efficient than actually using a lot of vertices. Here is our Vertex shader:

```
vertShader.vert
1    #version 110
2
3    //varying "out" variables to be used in the fragment shader
4    varying vec4 vColor;
5    varying vec2 vTexCoord;
6
7    void main() {
8        vColor = gl_Color;
9        vTexCoord = (gl_TextureMatrix[0] * gl_MultiTexCoord0).xy;
10       gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
11   }
```

This is a common shader which doesn't do anything particularly interesting—it only sets a few varying variables for the fragment shader. Varying variables are used when we want the fragment shader to use data, computed in the vertex shader. Each varying variable is interpolated for each fragment from the nearby vertices.

The shader in the preceding example also uses the Model*View*Projection matrix to calculate the position of the vertex in screen space. The fragment shader is where the magic happens:

```
rippleShader.frag ☒
 1    #version 110
 2
 3    //varying attributes from the vertex shader
 4    varying vec4 vColor;
 5    varying vec2 vTexCoord;
 6
 7    //declare uniforms
 8    uniform sampler2D uTexture;
 9    uniform float uPositionFreq;
10    uniform float uSpeed;
11    uniform float uStrength;
12    uniform float uTime;
13
14    void main() {
15        vec2 texCoord = vTexCoord;
16        float coef = sin(gl_FragCoord.x * uPositionFreq + uSpeed * uTime);
17        texCoord.y +=  coef * uStrength;
18        gl_FragColor = vColor * texture2D(uTexture, texCoord);
19    }
```

There are a lot of uniforms in the ripple fragment shader to enable us to use it in different configurations from outside. We need the uTexture sampler to get the colors from the sprite's texture. uPositionFreq determines how steep the rippling effect is for neighboring pixels. uStrength controls the magnitude of the effect on the y-axis and uSpeed controls the speed of the ripple. Finally, uTime represents the elapsed time. All we do to produce the ripple effect is to calculate a coef function, which depends on the current fragment position and all of our uniforms. We use that offset to calculate the final texture coordinate for the current fragment. This texture coordinate is used to find out the color of the texel of uTextue at the current location.

This is what the code which uses that shader looks like:

```
auto shader = AssetManager::GetShader("vertShader.vert", "rippleShader.frag");

sf::Sprite sprite(AssetManager::GetTexture("myTexture.png"));

shader->setParameter("uTexture", *sprite.getTexture());
shader->setParameter("uPositionFreq", 0.1f);
shader->setParameter("uSpeed", 20);
shader->setParameter("uStrength", 0.03f);
```

This is the initialization part, where we create the shader and the sprite, which will use it. We also set all of the uniforms (except the time, which will have to change for each frame) to reasonable values. The actual rendering looks like this:

```
sf::Clock clock;
while (window.isOpen())
{
    window.clear();

    shader->setParameter("uTime", clock.getElapsedTime().asSeconds());

    window.draw(sprite, shader);

    window.display();
}
```

Notice that we can set uniforms before each object draw call (in this case it is only the time). That means that we can have the same shader for multiple types of objects by changing some of the parameter values.

The effect of the current shader is a wobble on the *y*-axis that goes through the whole sprite:

Try it out for yourself, using a variety of textures, and experiment with the uniforms to create different looking effects.

sf::Shader and OpenGL

If we were to draw things directly in OpenGL, but still wanted to use the `Shader` class to load and manage our shaders, we could do that without much difficulty—we just need to bind the shader. OpenGL uses the currently bound shader for its draw calls. The code structure for such a scenario looks like this:

```cpp
//Bind the shader by passing a pointer to the function
sf::Shader::bind(shader);

/* Render objects using OpenGL here */

//Stop using shaders
sf::Shader::bind(nullptr);
```

And that is it for shaders. We will finish the chapter with one final example, which uses both render textures and shaders to create a pixelation effect.

Combining it all together

We talked about the `RenderTexture` class and we talked about shaders. It is no coincidence that these two classes are covered in the same chapter—the reason is that they work very well together. Instead of drawing everything directly on `RenderWindow`, we can draw the scene on `RenderTexture` and then render that texture on `RenderWindow` with a post-processing shader. That will let us create different effects, which is harder to achieve when applying shaders to each object separately. Some post-processing effects include: **tinting** (applying a color to the whole texture), **blurring**, **Fast Approximate Antialiasing** (**FXAA**), **pixelation** (reducing the number of pixels, while increasing their size), and many more.

In this section, we will talk about how to set up `RenderTexture` and give a simple example of a post-processing pixelation shader which can be used as a scene transition.

Setting up RenderTexture

When we want to create a post-processing effect, in most cases we want our texture to be as big (if not bigger) as the screen which we are rendering on to preserve the quality of the resulting image. As such, the first step is to initialize our window and `RenderTexture`:

```cpp
sf::RenderWindow window(sf::VideoMode(800, 600), "Pixelation");
AssetManager m;

if (!sf::Shader::isAvailable())
    return -1; //Shaders are not supported. Abort!

sf::RenderTexture rTexture;
auto wSize = window.getSize();
rTexture.create(wSize.x, wSize.y);

//The sprite used for post-processing the texture
sf::Sprite ppSprite(rTexture.getTexture());
```

After we create the window, we also instantiate `AssetManager`. It is useful for all our scene assets (even if we don't use it for `RenderTexture`). Then we check for shader support and exit the program if the check returns negative. Next, we create `RenderTexture` with the exact size of the window and finally we create a sprite, which is used to render the texture to the window.

The first step is then completed. Two things are left: the scene objects and the shader. For scene objects we can use whatever we like—the shader works with anything, as long as it's not transparent (then we won't actually see anything).

To achieve the pixelation we use the vertex shader from the previous example (`vertShader.vert`) and this new fragment shader, which actually creates the effect:

```glsl
pixelationShader.frag
1   #version 110
2
3   //varying attributes from the vertex shader
4   varying vec4 vColor;
5   varying vec2 vTexCoord;
6
7   //declare uniforms
8   uniform sampler2D uTexture;
9   uniform vec2 uTextureSize;
10  uniform float uPixelSize;
11
12  void main() {
13      vec2 pixelSizeNorm = uPixelSize / uTextureSize;
14      vec2 texCoord = vTexCoord - mod(vTexCoord, pixelSizeNorm);
15      gl_FragColor = vColor * texture2D(uTexture, texCoord);
16  }
```

The shader requires three uniforms—the render texture, the size of the texture, and the pixel size (in screen space). The size of the texture is required to normalize the pixel size in the range of [0…1]. After we have that, we can clamp the value of the texture coordinate for the current fragment to the nearest pixel (depending on our pixel size). The final line sets the color of the fragment from the texture with the newly calculated texture coordinate.

Once we have the shader written, we load it and set the values for the uniform:

```
//The shader used for post-processing the texture
auto shader = AssetManager::GetShader("vertShader.vert", "pixelationShader.frag");

shader->setParameter("uTexture", rTexture.getTexture());
shader->setParameter("uTextureSize", static_cast<sf::Vector2f>(rTexture.getSize()));
shader->setParameter("uPixelSize", 8);

sf::Sprite sprite(AssetManager::GetTexture("myTexture.png"));
```

The texture and texture size uniforms should only be set once (if we are using the shader with a single texture), but the pixel size can be changed as many times as we like. It is not restricted to an integer—any positive `float` value works. To achieve a transition effect, we can set it to gradually increase from frame to frame, then load the new scene and start decreasing it back to one.

The next bit is our game loop:

```
while (window.isOpen())
{
    //Handle events
    sf::Event ev;
    while (window.pollEvent(ev)) {}

    /* Update frame here */

    //Render frame
    rTexture.clear();
    {
        /* Draw scene here */
    }
    rTexture.display();

    window.clear();
    {
        //Post processing by applying the shader to the RenderTexture
        window.draw(ppSprite, shader);
    }
    window.display();
}
```

As always, we start with even handling and update frame. These are not the focus of this chapter. The important bit is at the end where we first render the whole scene to the `RenderTexture` (as if it was the window) and then render the sprite which uses that texture to the window with the post-processing shader. That is all we have to do. Here is the result of an example scene with different pixel size configurations:
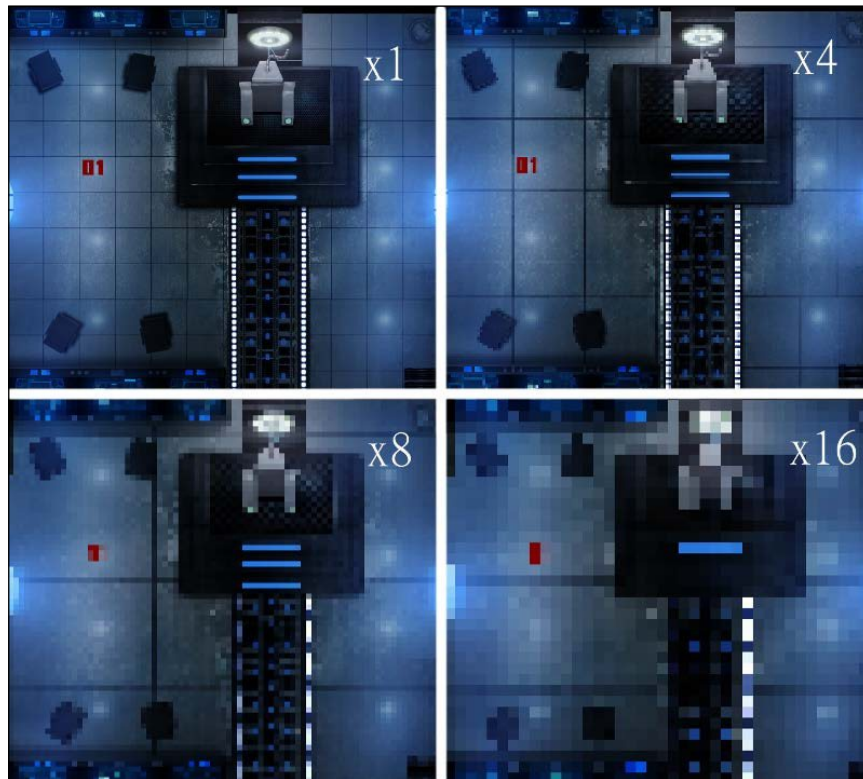
And with that we conclude our journey about shaders.

Summary

Shaders and render textures are extremely useful tools when it comes to graphics programming. In this chapter, we touched on their implementation in SFML and how they can be loaded and used to create interesting effects. There is a lot more to explore when it comes to shaders though, as GPUs have a lot of power nowadays that can be used to create amazing-looking scenes.

Now that we are done with shaders, we can move to an entirely different, but still fascinating, area of games programming—networking.

7
Building Multiplayer Games

This is it—the final chapter. Here we explore a whole new branch of games programming—networking. We will be talking about exchanging data between devices across a network. That might be a local network or a network connecting the opposite sides of the Pacific Ocean. This is particularly useful in games where we want to achieve a multiplayer experience across different machines.

However, as was mentioned in the previous chapter, you will not find in-depth explanations of the concepts behind network programming. This chapter covers the SFML networking module of SFML and its connection to commonly used architectures for network games programming, such as client-to-client and client-server.

In this chapter we will cover the following topics:

- Understanding networking
- The Transport layer—TCP versus UDP
- TCP with SFML
- UDP with SFML
- Exchanging packets
- Putting it all into practice

Understanding networking

The concept of networking has been around for quite a while but if we are going to write code which relies on information from another machine, it is important to grasp the main concept behind a network. Ultimately, networking is about sending and receiving data to and from different machines. That data can be used for a variety of things—simply redirecting it in another direction (like a router), performing a task and returning the result, updating something locally in a database, and so on.

Starting with the basics, if two systems want to exchange data between each other, they need to be on the same network (or on separate networks that are connected to each other). That is as simple as hooking an Ethernet cable from one network card to the other, creating a fairly simple ad-hoc network (only between two machines), but still being able to serve its purpose just fine. If all networks were so simple, the process of interacting between machines would be much simpler, but ultimately networks wouldn't be as useful as they are today.

When we start to introduce concepts such as more than two machines on the same network, multiple networks connected to each other, long distances between machines and so on, we start running into problems. The solutions involve clever engineering, which we will not get into, but the result is a structured layered networking model, which enables us to avoid some of the low-level problems that are inherent in networking. This is what a simplified version of the model looks like:

Layers	Protocols
Application	FTP, HTTP
Transport	TCP,UDP
Network	IP
Physical	Ethernet

In this chapter we will be working in the application layer and using the layer beneath—the Transport layer. When it comes to protocols, we will be dealing with TCP and UDP to transfer our data over a network. More information on the model can be found here: `http://wikipedia.org/wiki/OSI_model`. This covers the very basics of what networking actually is. Next we will be talking about networking in games and a comparison of TCP and UDP, since both of them are widely used in games development.

Networking in games

Networking (which includes the internet) has many uses in games, from having a simple high-score table to running a server for a **Massively Multiplayer Online** (**MMO**) game.

The most interesting (and arguably most difficult) use of networking is for creating multiplayer games. The difficulty comes from the fact that we have to show the same scene on multiple screens at any given time. This is extremely difficult when we have a lot of dynamic objects, events, players dropping in and out, and varying quality in the underlying networks. Programmers have been trying to solve these issues by introducing different networking design architectures, the most common of which are: peer-to-peer and client-server architectures.

Peer-to-peer was mostly used in the early days of network programming. In this architecture, each client (a device) communicates with every other client to update its local simulation of the game. Here is a diagram showing the connections between six clients (30 connections in total):
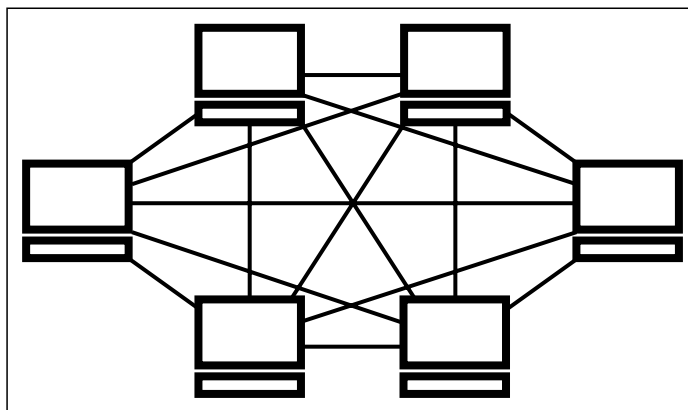
This works well with a small number of clients, but bandwidth can become an issue when we bring more players in (most notably on slow networks, such as the internet). Furthermore, the architecture runs into problems when dealing with synchronization. For example, one client might detect a collision between two objects at a particular moment, which doesn't happen on another client. Choosing which one is correct is not easy, since they are both running independently from each other and both simulations are technically valid ones. Apart from that, clients can easily cheat since there is no authority in the system and they can selectively ignore the states sent by the other clients. To solve these problems, and to streamline the process, an alternative networking model exists—the client-server.

The client-server model operates on a very simple basis—one server runs the simulation with all the logic and clients simply use that simulation to display their scene. In this scenario, the clients are not running any simulation code (ideally), but they get the state of each dynamic object from the server (position, rotation, and so on.) and display that object in that state. The connections between devices are shown in the following diagram:
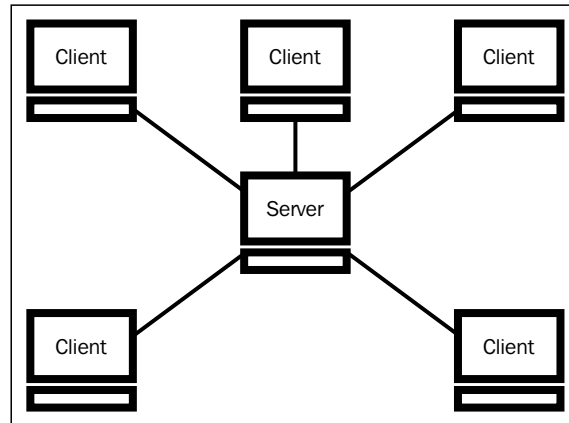
Once the server updates the simulation, every client receives the new game state and updates their objects locally. This removes synchronization problems almost completely, since only one device is running the simulation. It is also more bandwidth friendly for clients which do not have to communicate with every other client.

Both peer-to-peer and client-server can be useful in different situations. Peer-to-peer is suitable for games which are designed to run on LAN or do not update their entities every frame, but only occasionally, as in turn-based games and point-and-clicks. The benefit of using peer-to-peer is the fact that it's slightly simpler to set up than client-server. In most cases though, choosing client-server over the peer-to-peer model is better as it introduces fewer problems later on.

Now that we have a basic idea of what networking is and how it can be used, let us talk a bit about the Transport layer of the networking model.

Transport layer – TCP versus UDP

The chances are that if a person is dealing with the transport layer, they will opt to use either TPC or UDP to achieve their goal. In essence, both protocols exist to achieve one thing—deliver data from one machine to another over a network. The differences come in reliability and performance.

When we try to send data over the internet (in the form of a packet), the **IP** (**Internet Protocol**—Network layer) tries to guide the data the best it can so that it arrives at its destination. The destination of each packet is marked by an IP address, which represents the destination in the form of a number (IPv4 uses a 32-bit field, whereas IPv6 uses a 128-bit one). However, the protocol doesn't guarantee that a packet will arrive at its destination at all (it may get lost along the way), or if it does arrive, the data in it can be damaged. These problems are tackled by TCP and UDP, which are built on top of the Internet Protocol.

The **Transmission Control Protocol** (**TCP**) is **connection-oriented** and it's used when we want to ensure that our packets arrive at their destination intact and in the correct order. That makes the protocol **reliable**. In order for that to happen, the protocol has to send a lot more information with the original data, because it tracks each and every packet. If a packet gets lost, the protocol waits until a new one is sent over before continuing. This ensures that when we send packets {1, 2, 3, 4} they will arrive in the same order: 1, 2, 3, 4.

On the other hand, **User Datagram Protocol** (**UDP**) is unreliable and does not track if a packet goes missing or if the order of the packets is wrong. This technique requires much less memory for each packet and doesn't require any wait time for lost packets, because the protocol doesn't know if a packet got lost along the way.

When considering which protocol to use, we should ask ourselves one question, "can we wait for lost packets or not?" In many cases the answer to that question is yes. Most things on the internet use TCP—all web pages (HTML), e-mail, business applications, transaction applications, and so on. All of those have to guarantee that the data which was sent from one end is safely transferred to the other and they can afford to sacrifice a few milliseconds of wait time to achieve that. Some games fall into the same category. Genres such as turn-based strategies, card games, point-and-clicks, and even some puzzlers use TCP with no problems. In most cases though, when we have a lot of objects that need to be synchronized on all clients at the same time (action games, first person shooters, real time strategies), TCP can slow down the game by having to wait for packets (which we might be ok with losing—for example, some position updates can get lost with no impact). Also the additional data with each packet can cause the game to use a lot more bandwidth than the actual data requires. In such situations UDP provides a better alternative—as long as we take into account that some packets might never make it, we can use it to get much better performance out of sending and receiving packets.

TCP in SFML

Since TCP has to track all the packets that are being sent and received, each TCP client can be connected to only one other TCP client. This connection makes it possible to use the protocol similarly to how we might use a stream—we write and read data from it.

In SFML, TCP is represented by two classes: `sf::TcpSocket` and `sf::TcpListener`. The former tries to initiate a connection (the client), whereas the latter tries to receive the connection (the server). If a successful connection is established, `TcpListener` produces another `TcpSocket` on its end, resulting in one connected `TcpSocket` instance on each end.

Here is an example where `TcpSocket` tries to establish a connection with a remote client:

```
sf::TcpSocket tcpSocket;
if (tcpSocket.connect("192.168.0.123", 45000) != sf::Socket::Done)
{
    //Connection failed - abort!
    return -1;
}

//Send some data to the other client
```

The `TcpSocket::connect()` method takes an IP address (or a hostname) and a port and returns a `Socket::Status`, which indicates how the process has gone. It can be one of the following:

Status code	Description
Status::Done	The operation was successful
Status::NotReady	The socket is not ready to do that operation yet (relevant for non-blocking mode)
Status::Disconnected	The socket has been disconnected
Status::Error	There was an unexpected error during the operation

In the preceding example we try to connect to a computer on the local network (192.168.xxx.xxx) on port 45000. In most circumstances, the port indicates which process we want to reach (some processes can use multiple ports though). That's why we need to know the port before we try to connect.

Once we have a connection, we can send over data by calling the `TcpSocket::send()` method, which requires a pointer to an array of data and its size and returns a `Status` code. `Status` does not indicate if the other side has received the message, but only if it has been sent. Here is an example:

```cpp
const int msgSize = 100;
char message[msgSize] = "Nice hat you have there";
if (tcpSocket.send(message, msgSize) != sf::Socket::Done)
{
    //Something went wrong - data was not sent
}
```

On the other side we have a `TcpListener`, which waits for incoming connections and reads the data. Let us look at the following code:

```cpp
//Star listening for incoming sockets
sf::TcpListener listener;
listener.listen(45000);

//Wait until the listener has accepted a valid connection
sf::TcpSocket socket;
if (listener.accept(socket) != sf::Socket::Done)
    return -1;

sf::sleep(sf::seconds(1));

//Read the data
const std::size_t size = 100;
char data[size];
std::size_t readSize;
if (socket.receive(data, size, readSize) != sf::Socket::Done)
{
    //Something went wrong - data was not received
    return -1;
}

std::cout << data << std::endl;
```

First of all, we need to start listening for any incoming connections by calling the `TcpListener::listen()` method and passing the port, onto which we are expecting the connection. The `TcpListener::accept()` method waits until a connection attempt has been made and, if successful, it puts the result in the referenced `socket` parameter. In order to recover the data from the client, we need to have somewhere to put it, hence the `data` array. The `readSize` variable indicates how much data has been read by the socket, whereas the `size` constant is just the size of the array. The `TcpSocket::receive()` method retrieves the data (if it can) and returns a `Socket::Status` value. If everything goes smoothly, we display the data on the screen.

Once we are connected, we can check the IP address and port of the other client by calling `TcpSocket::getRemoteAddress()` and `TcpSocket::getRemotePort()`. These methods return `IpAddress::None` and `0`, respectively, if the socket is not currently connected.

It is important to note that, after the data exchange, the sockets are still connected and they can continue to send and receive data. If we keep the connection going for too long though, we have to take special precautions to ensure the connection stays alive. For this example though, we terminate the connection immediately after the data is sent over. We can terminate the connection from either end by calling:

```
tcpSocket.disconnect();
```

Now, let's move on to UDP.

UDP in SFML

Since UDP doesn't keep track of lost packets, it doesn't need to be connected to the other side. In fact, one socket can send multiple packets to different destinations with no problem.

Sending packets with UDP looks similar to TCP, the only difference is that we use the `sf::UdpSocket` class:

```
sf::UdpSocket udpSocket;

//Send some data to the other client
const int msgSize = 100;
char message[msgSize] = "Nice hat you have there";
if (udpSocket.send(message, msgSize,
    "192.168.0.123", 45000) != sf::Socket::Done)
{
    //Something went wrong - data was not sent
}
```

Note that we don't have to connect before sending over our packet. The remote port and IP address are needed only after we are ready to send the data across. The `UdpSocket::send()` method takes those as the last two parameters and returns a `Socket::Status` (the same as the `TcpSocket`). It is important to note that the status does not tell us if the other side has received the data, but it only indicates that the current action (in this case sending a packet) was successful or not. That means that the packet might not arrive at its destination at all and that is absolutely fine with the sender — after all that is part of the design behind UDP.

On the receiving end, the code looks like this:

```cpp
sf::UdpSocket socket;
//Bind the socket to a port so it can receive data
socket.bind(45000);

//Receive the data
const std::size_t size = 100;
char data[size];
std::size_t readSize;
sf::IpAddress senderIP;
unsigned short remotePort;
if (socket.receive(data, size, readSize,
    senderIP, remotePort) != sf::Socket::Done)
{
    //Something went wrong - data was not received
    return -1;
}

std::cout << "Received data from: " << senderIP
    << " on " << remotePort << std::endl;
std::cout << data << std::endl;

//Unbind the socket
socket.unbind();
```

This looks quite a bit different to how TCP receives packets. With UDP we do not need a connection and thus we do not need a listener. It is sufficient to have one socket (bound to a particular port) which can receive packets from any other socket. To see from which client (IP address and port) the data is coming from, we have to specify `IpAddress` and `ushort` references, which are filled with the address of the sender and the port. Apart from that, the method `UdpSocket::receive()` works in a similar way to its TCP alternative — it requires an array and fills the number of read bytes in the `readSize` variable. Once we have received the data with no problems, we print the address of the sender and display the message. In the preceding example , we exit the server (unbinding the socket) as soon as we have received something from any client.

One UDP socket has the functionality to send or receive messages to or from multiple other sockets. That makes it possible for a sender to send data to every other listening `UdpSocket` in a sub-network with a single `UdpSocket::send()` call. To do that we use the `IpAddress::Broadcast` field (255.255.255.255) as an IP address. This sends a message to every `UdpSocket` in the sub-network, bound to the `45000` port:

```
if (udpSocket.send(message, msgSize,
    sf::IpAddress::Broadcast, 45000) != sf::Socket::Done)
{
    //Something went wrong - data was not sent
}
```

This is very useful for LAN lobbies, since the host can call out to every other player who is currently looking at joining a game. In that way players don't need to know the exact IP address of the host machine.

A final remark for UDP specifically is that there is a maximum size of the packet which is sent with the `UdpSocket::send()` method - a little less than 2^16 (65536) bytes. We can also get this number by looking at `UdpSocket::MaxDatagramSize`.

Non-blocking sockets

By default, every socket (it doesn't matter if it's a `TcpListener`, `TcpSocket`, or `UdpSocket`) is a **blocking** socket. When a socket is blocking, each of its methods wait until the operation completes before it returns. For some methods, such as `TcpSocket::receive()` or `TcpListener::accept()` however, it can take an indefinite amount of time for something to happen. If the other side is not sending any data or if it's not trying to connect at all, the server side will get blocked until one of those actions happen. To solve that problem each socket can be set to a non-blocking state.

All three sockets inherit from the `sf::Socket` class, which implements the blocking or non-locking state system. When a socket is in the non-blocking state, its methods return immediately, if there is nothing happening at the moment or if the data transfer is not ready. For example calling `TcpListener::accept()` returns `Socket::NotReady` if there is no socket currently waiting to be connected. If the socket is in blocking mode, then that method waits until there is a client available. The same behavior goes for `TcpSocket::receive()` and `UdpSocket::receive()`.

This functionality is useful when we are dealing with real time games or applications, where we expect clients to come and go as well as send us data whenever they want. For that we need to continue running our main loop and only check once a frame if there is something waiting for us. If so, we can handle it appropriately or, alternatively, just continue with our loop and check again next time.

The method that enables the blocking/non-blocking mode is
`Socket::setBlocking()`, which requires a `boolean` parameter. This is an
example of how to handle `UdpSocket`, which checks for new data with every frame:

```cpp
sf::UdpSocket socket;
//Bind the socket to a port so it can receive data
socket.bind(45000);
socket.setBlocking(false);
while (window.isOpen())
{
    //Receive the data
    const std::size_t size = 100;
    char data[size];
    std::size_t readSize;
    sf::IpAddress senderIP;
    unsigned short remotePort;
    auto status = socket.receive(data, size, readSize, senderIP, remotePort);
    //Check to see if anyone has tried to send us any data
    if (status == sf::Socket::Done)
    {
        //Do something with the data
    }
    else
    {
        //No data available yet
    }

    /* Input + Update + Render here */
}
```

The code is the same as before with the difference that this time we use it in our main
loop and the socket doesn't block the loop from continuing with its execution.

Now that we know how to avoid blocking our main loop thread, let's look at some
common networking problems and how to solve them.

Exchanging packets

Until now we have been sending and receiving data in the form of bytes (`char` arrays
to be exact), which is absolutely fine. Things get a bit trickier if we start sending
types which are more than a single byte in memory, such as an `integer` or a `double`.

The first problem comes from the different sizes of the same data types between
platforms and processors. For example, an `integer` is 32-bit on some configurations
and 64-bit on others.

To solve that problem, we use the classes that SFML provides for primitive types, such as `sf::UInt8` (8-bit unsigned), `sf::Int16` (16-bit signed), `sf::Int32` (32-bit signed), and so on. With these, we can ensure that our data is the same size on all platforms. There are no custom `float` or `double` types, because they are always the same size (on the platforms on which SFML is supported).

Another (and more serious) problem comes in the endianness of different operating systems and different processors - in other words, how are the bytes of a data type interpreted by the processor. **Big endian** processors expect the most significant byte first, whereas **little endian** family processors place the least significant byte first. If we send a `sf::UInt16` (16-bit) value of '00000000 00000010' (in binary), which on our machine is 2, on another machine it can read as 512 (reading the bytes in the opposite order: "00000010 00000000").

To avoid that problem, SFML introduces an elegant solution — the `sf::Packet` class.

Constructing a packet

The main concept of the packet is this: fill it up with data, send it over, and read all the data from the packet on the other end. It will seem like the exact same packet has been received on the other end.

The `Packet` class is built with a streaming interface, so we can use the '`<<`' operator to write to it and the '`>>`' operator to read it:

```cpp
sf::Packet packet;
sf::Vector2f _position(1.5f, 0.5f);
sf::String _name = "Enemy";
sf::Int16 _ID = 1000;

packet << _ID << _name << _position.x << _position.y;
```

On the receiving end we can get those values back by doing the following:

```cpp
/* Receive the packet here */

sf::Vector2f position;
sf::String name;
sf::Int16 ID;

packet >> ID >> name >> position.x >> position.y;
```

We can also write operator overloads for custom data types. For example, SFML doesn't provide a read/write operator for the `Vector2f` (`Vector2<float>`) class. We can use our own implementation in this case:

```cpp
sf::Packet& operator <<(sf::Packet& packet, sf::Vector2f const& vec)
{
    return packet << vec.x << vec.y;
}

sf::Packet& operator >>(sf::Packet& packet, sf::Vector2f& vec)
{
    return packet >> vec.x >> vec.y;
}
```

Later on we can read and write vectors directly using the following interface:

```cpp
sf::Packet packet;
sf::Vector2f vector(1.0f, 0.5f);
sf::Int32 additionalData;

//Write a vector to a packet
packet << vector << additionalData;
//Read a vector from a packet
packet >> vector >> additionalData;
```

Sending and receiving a packet is similar to sending and receiving a byte array, we use the `*::send()` and `*::receive()` methods of the sockets. We start with the TCP client-side:

```cpp
sf::TcpSocket socket;
/* Connect to a listener */

sf::Packet packet;
/* fill the packet */

if(socket.send(packet) != sf::Socket::Done)
{ /* Something went wrong, handle it */ }
```

On the receiving side, the code looks like this:

```
sf::TcpListener listener;
sf::TcpSocket lSocket;
/* Accept the socket */

sf::Packet packet;
if (lSocket.receive(packet) == sf::Socket::Done)
{
    /* Packet received. Do something with it here */
}
```

Using UDP, we can send packets in the following manner:

```
sf::UdpSocket socket;
sf::Packet packet;
/* fill the packet */

if (socket.send(packet, "192.168.0.3", 45000) != sf::Socket::Done)
{ /* Something went wrong, handle it */ }
```

And finally, on the receiving side:

```
sf::UdpSocket lSocket;
/* bind the socket */

sf::Packet packet;
sf::IpAddress remoteIp;
unsigned short remotePort;
if (lSocket.receive(packet, remoteIp, remotePort) == sf::Socket::Done)
{
    /* Packet received. Do something with it here */
}
```

Now that we've covered the basics, let us demonstrate the power of networking with a final example.

Putting it all into practice

We are going to build a very simple client-to-client simulation of two players who both control a character that can move in four directions: up, down, left, and right. This is not a fluid simulation, but the player's movement is limited by one tile per frame. In this case, we can use TCP to achieve a decent networking experience.

We start with the world. It has to be split into tiles and each tile has to have a certain size (defined in pixels). Here is our `#include` and `#define` list:

```
Main.cpp  ⊞  X
⊟#include <SFML/Network.hpp>
 #include <SFML/Graphics.hpp>
 #include <iostream>

 #define TILE_SIZE 40.f
```

The graphics module is used for the `RenderWindow` and `RectangleShape` representations of the players and the networking module is for the TCP sockets. We use the console as a way of supplying the destination IP address. This can be done in many other ways, but the console seems appropriate for this example. Finally, we specify that a board tile is exactly 40 pixels.

Moving on to the first part of the program (the connection), we let the players type **host** to start waiting for a connection (acting as the listener) or type in an **IP address** to connect to a waiting host. This is the code that produces the behavior:

```cpp
//Establish a connection
sf::TcpSocket socket;
std::string consoleInput;
std::cin >> consoleInput;
if (consoleInput == "host")
{
    sf::TcpListener listener;
    listener.listen(45000);
    std::cout << "Waiting for connection..." << std::endl;
    if (listener.accept(socket) != sf::Socket::Done)
        return -1;
}
else
{
    std::cout << "Trying to connect..." << std::endl;
    //Timeout is set to 10 seconds. If nothing happens - Abort
    if (socket.connect(consoleInput, 45000, sf::seconds(10)) != sf::Socket::Done)
    {
        //Couldn't connect for some reason. Abort
        return -1;
    }
}
```

Firstly, we declare a `TcpSocket`, which is connected by the next part (no matter if that's the host or the connecting client). Then we ask the player to write a line of code. If that is the word **host** then we wait for a connection and, when we get one, the resulting socket is initialized from the listener. On the other hand, if the player types something else (we are expecting an **IP address**), then we try to use that (even if it's not an IP address) to connect to a waiting host. If that fails for any reason, we exit the application, since there is nothing else that we can do.

At the end, we will either have a valid and connected `TcpSocket` or the application would exit, because of an error. Once we have a connection going we can initialize the scene:

```cpp
//Setup the scene
sf::RenderWindow window(sf::VideoMode(640, 480), "Networking");

socket.setBlocking(false);

sf::Vector2f shapeSize(TILE_SIZE, TILE_SIZE);
sf::RectangleShape localShape(shapeSize);
sf::RectangleShape remoteShape(shapeSize);
```

Pretty straightforward there — we create a window and two shapes (for both players) as well as setting the socket to non-blocking mode so we can use it inside our game loop without blocking it. Each shape fills each tile of the board perfectly. You might notice that there isn't actually a board object anywhere and that is because we don't need it in such a simple example. We can simulate a board by moving a player by exactly `TILE_SIZE` pixels in any direction and create the illusion that the simulation is actually happening on a tiled board.

Moving on to the game loop itself, the first thing we want to do is handle input:

```cpp
while (window.isOpen())
{
    //Handle Input
    sf::Vector2i moveDir;
    sf::Event event;
    while (window.pollEvent(event))
    {
        switch (event.type)
        {
        case sf::Event::KeyPressed:
            if (event.key.code == sf::Keyboard::W)
                moveDir.y += -1;
            else if (event.key.code == sf::Keyboard::A)
                moveDir.x += -1;
            else if (event.key.code == sf::Keyboard::S)
                moveDir.y += 1;
            else if (event.key.code == sf::Keyboard::D)
                moveDir.x += 1;
            break;
        case sf::Event::Closed:
            window.close();
            break;
        }
    }
```

This event switch is nothing special. It checks against button presses and assigns a value to the move direction vector, depending on which key is pressed. We will later use that direction vector in our game logic in the update frame. We also handle the `Event::Closed` event as is customary.

In the game logic section of the loop we want to do two things:

1. Update the remote shape if a new packet with information has come in.
2. Update our local shape from the position vector and send the new position over the other side.

This is how we achieve that behavior:

```cpp
//Check for new packets
sf::Packet packet;
if (socket.receive(packet) == sf::Socket::Done)
{
    sf::Vector2f pos;
    packet >> pos.x >> pos.y;
    remoteShape.setPosition(pos);
}

//Update frame
if (moveDir.x != 0 || moveDir.y != 0)
{
    localShape.move(moveDir.x * TILE_SIZE, moveDir.y * TILE_SIZE);
    sf::Packet packet;
    packet << localShape.getPosition().x << localShape.getPosition().y;
    if (socket.send(packet) != sf::Socket::Done)
    {
        //Handle problem (probably the other disconnected)
        return -1;
    }
}
```

First we check for any new packets coming our way. If there is a packet waiting, we unpack its information (only a position in this example) in a `Vector2f` instance and set the position of the remote shape (the shape of the other client) to that position.

After we have updated the remote shape, it is time to handle our own. First we check if we have to move at all. If we don't have to move, we don't have to update the position of our shape and thus don't have to send a packet over. This prevents any unnecessary data being transferred over the network. However, if our shape has to move due to the input, we move the shape itself and compose a packet of its position. The last step is to send it over to the other side. There might be a problem with sending it—the other side might not be responding due to disconnection, internet problems, and so on. In this, we assume that it's a disconnection problem and exit the program.

The final part of the loop is to render the scene. In this example that's fairly simple—we just have to render the two shapes:

```
//Render frame
window.clear();

window.draw(localShape);
window.draw(remoteShape);

window.display();
}
```

We then have a fully functional (if somewhat simple) example of a networking application. This wouldn't be classified as a game by most people, as a game needs more gameplay elements than just moving around. However that's definitely not out of the question as you can use this example to build your own logic on top of it, using the same formula (or an alternative one, if you prefer).

Since this is a networking example, it is important to note that, if you try to use the program over the internet and the host is behind a router, they need to forward the 45000 TCP port. Otherwise the router will reject the connection and the client will fail to connect to the host. You can test the example on the same PC by using `IpAddress::LocalHost` IP, which is **127.0.0.1**—you just need to run two instances of the program - one as the host and one as the client.

And that concludes the last section of the networking chapter.

Summary

In this chapter, we covered the topic of networking. We talked about the different layers in the networking model, while focusing specifically on the transport layer and the differences between TCP and UDP. Later on we saw how to use sockets in SFML, while exchanging both byte arrays and SFML packets. Finally, we rounded everything up by implementing a fully functional networking example, which serves as the basis of a multiplayer game.

With that we conclude the beginning of the journey through the land of SFML. There is much more to explore, both on your own and by reading other books and materials online about game development.

Index

W

window
  ContextSettings argument 10
  creating 7, 8
  mouse cursor, disabling 10
  Style argument 9
  VideoMode class 8
Window::close() function 14
Window::display() method 18
window related events
  Event::Closed 12
  Event::GainedFocus 12
  Event::LostFocus 12
  Event::Resized 12
Window::setActive() method 79

Z

zoom feature 68

PACKT PUBLISHING
open source
community experience distilled

Thank you for buying
SFML Essentials

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `www.packtpub.com`.

About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.
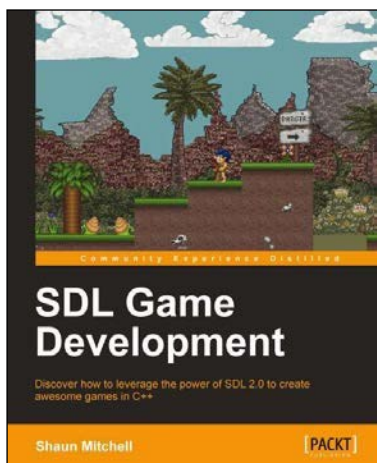
SFML Game Development

ISBN: 978-1-84969-684-5   Paperback: 296 pages

Learn how to use SFML 2.0 to develop your own feature-packed game

1. Develop a complete game throughout the book.

2. Learn how to use modern C++11 style to create a full featured game and support for all major operating systems.

3. Fully network your game for awesome multiplayer action.

4. Step-by-step guide to developing your game using C++ and SFML.

SDL Game Development

ISBN: 978-1-84969-682-1   Paperback: 256 pages

Discover how to leverage the power of SDL 2.0 to create awesome games in C++

1. Create 2D reusable games using the new SDL 2.0 and C++ frameworks.

2. Become proficient in speeding up development time.

3. Create two fully-featured games with C++ which include a platform game and a 2D side scrolling shooter.

4. An engaging and structured guide to develop your own game.

Please check **www.PacktPub.com** for information on our titles

GameMaker Game Programming with GML

ISBN: 978-1-78355-944-2   Paperback: 350 pages

Learn GameMaker Language programming concepts and script integration with GameMaker: Studio through hands-on, playable examples

1. Write and utilize scripts to help organize and speed up your game production workflow.

2. Display important user interface components such as score, health, and lives.

3. Play sound effects and music, and create particle effects to add some spice to your projects.

Instant HTML5 2D Platformer

ISBN: 978-1-84969-678-4   Paperback: 52 pages

Learn how to develop a 2D HTML5 platformer that is capable of running in modern browsers

1. Learn something new in an Instant!
   A short, fast, focused guide delivering immediate results.

2. Learn about HTML5 2D game development and how to create your own HTML5 games.

3. Embed your game within a webpage and share it with friends.

4. Create enhanced games and publish to the Apple app store, Google Play, Windows Marketplace, or Facebook.

Please check **www.PacktPub.com** for information on our titles