

C o m m u n i t y E x p e r i e n c e D i s t i l l e d

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*
PUBLISHING community experience distilled

www.allitebooks.com

SFML Essentials

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

Milcho G. Milchev



BIRMINGHAM - MUMBAI

SFML Essentials

Copyright © 2015 Packt Publishing

retrieval
itten
dded in
critical articles or reviews.

accuracy
book
nor Packt
damages
caused or alleged to be caused directly or indirectly by this book.

ll of the
companies and products mentioned in this book by the appropriate use of capitals.
However, Packt Publishing cannot guarantee the accuracy of this information.

First published: February 2015

Production reference: 1170215

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78439-732-6

www.packtpub.com

Credits

Author

Milcho G. Milchev

Project Coordinator

Akash Poojary

Reviewers

James Cowgill

Mark Fielbig

Stefan Schindler

Proofreaders

Simran Bhogal

Kevin McGowan

Commissioning Editor

Edward Bowkett

Indexer

Rekha Nair

Acquisition Editor

Harsha Bharwani

Graphics

Valentina D'silva

Disha Haria

Content Development Editor

Natasha D'Souza

Production Coordinator

Komal Ramchandani

Technical Editor

Narsimha Pai

Cover Work

Komal Ramchandani

Copy Editors

Vikrant Phadke

Neha Vyas

About the Author

Milcho G. Milchev has been a game programmer since he was in high school. He as worked on include *Unholy Flesh* and *WhiteCity*. Currently, he works for Dream Harvest, 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
Over the
es including
e Debian
Linux distribution and helping out developers of open source projects.

Mark Fielbig was first introduced to game development through his love
ing
and releasing numerous Flash and mobile games. Mark now holds a Masters of
Computer Science degree from Stony Brook University and his interests include
chitecture.

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

He is a
modern C++ library to finish a small university game project, which was a *Tetris*
team
to help with their progress. Stefan accepted the invitation. He likes integrating
ebsite
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 published a couple of
them under the name, *Game Development Design*, available for free at `www.optank.org`.

I would like to thank Laurent Gomila for creating a wonderful C++
library that so many people love and use, and for bringing together
people who would probably have never met otherwise. Thanks to
my wife and daughter as well. Tina and Mia, I love you!

www.PacktPub.com

Support files, eBooks, discount offers, and more

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

ith PDF

and ePub files available? You can upgrade to the eBook version at www.PacktPub.com 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>

online digital
ary 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
dentials
for immediate access.

Table of Contents

Preface	1
Chapter 1: Getting Started with SFML	7
Creating windows	7
VideoMode	8
Style	9
ContextSettings	10
Disabling the mouse cursor	10
The game loop	10
Event handling	11
Window related events	12
Keyboard related events	13
Mouse related events	13
Joystick related events	14
Using events	14
Shape rendering and transformations	17
The render frame	17
Shape drawing	19
Shape transformation	21
Controlling shapes	24
Summary	28
Chapter 2: Loading and Using Textures	29
Loading textures	29
Images versus textures	30
Creating images	30
Creating textures	32
Rendering shapes with textures	34
What is a sprite?	40
Shapes versus sprites	40

Transformables and drawables	41
Final facts on sprites	41
Managing resources	42
Summary	46
Chapter 3: Animating Sprites	47
Capturing time	47
sf::Time and sf::Clock	49
Sprites in action	51
The setup	51
Building an animator	54
Using the animator	60
Multiple animations	61
Summary	62
Chapter 4: Manipulating a 2D Camera	63
What is a camera?	63
When should we use a camera?	64
How does SFML implement a camera?	64
Manipulating cameras with sf::View	65
Rotating and scaling a view	66
Viewports	70
Mapping coordinates	73
What is OpenGL?	74
Should you use OpenGL?	74
Using OpenGL inside SFML	74
OpenGL in multiple windows	79
Summary	79
Chapter 5: Exploring a World of Sound and Text	81
Audio module – overview	81
Sound versus music	82
Audio in action	83
The sf::Sound class	83
Introducing AssetManager 2.0	86
sf::Music and sf::SoundStream	88
sf::SoundSource and audio in 3D	89
Common audio features	89
Audio in 3D	89
Setting up the listener	90
Audio sources	92
Summarizing audio features	93

Getting started with sf::Text	95
AssetManager 3.0	97
Summary	99
Chapter 6: Rendering Special Effects with Shaders	101
sf::RenderTarget and sf::RenderWindow	102
Rendering directly to a texture	103
Shader programming	105
What is a shader?	106
Loading shaders	107
AssetManager 4.0	108
Using shaders	109
Setting shader uniforms	110
sf::Shader and OpenGL	113
Combining it all together	113
Setting up RenderTexture	114
Summary	116
Chapter 7: Building Multiplayer Games	117
Understanding networking	117
Networking in games	118
Transport layer – TCP versus UDP	120
TCP in SFML	122
UDP in SFML	124
Non-blocking sockets	126
Exchanging packets	127
Constructing a packet	128
Putting it all into practice	130
Summary	134
Index	135

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 best practices in this field.

Game development can be a difficult topic to understand. *SFML Essentials* will deas 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 rect 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.

What you need for this book

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/>.


Who this book is for


Conventions

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."

```
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

tions:

- *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 our titles 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/73260S_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

t
questions@packtpub.com, and we will do our best to address the problem.

1

Getting Started with SFML

and
oop looks
reen.

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:

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

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

    return 0;
}
```



Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. 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.

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.

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

ator. So in
te:

```
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 for OpenGL. 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

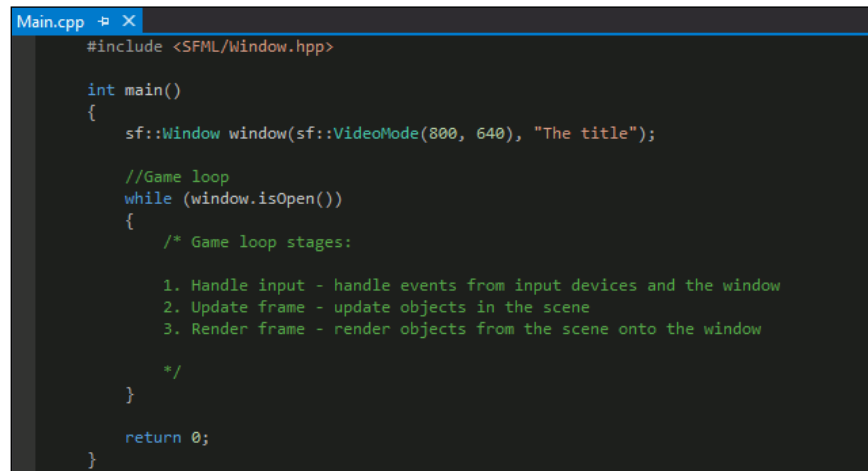
Each of these settings will be explained in more detail in *Chapter 5, Manipulating a 2D Camera*, where you will learn how to render things directly using OpenGL.

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:



```
Main.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 state to the right as long as a certain key is pressed (direct key query).

is the

.

The final stage of the loop comes right after we finish updating our objects. Here, every object is rendered on the screen again.

Going back to the example of our game loop, it currently doesn't perform the things 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; the typical event loop looks like:

```
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 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.

what they are
related to:

- Window
- Keyboard
- Mouse
- Joystick

Window related events

Enum value	Member associated	Description
<code>Event::Closed</code>	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.
<code>Event::Resized</code>	<code>Event::size</code> holds the new size of the window	This event is triggered when the OS detects that the window has been resized manually, or when <code>Window::setSize()</code> has been used.
<code>Event::LostFocus</code> <code>Event::GainedFocus</code>	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 <code>Window::setJoystickThreshold()</code> . 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:

```
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:

```
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 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.

s using the `TextEntered` event and, when the *Enter* (or *Return*) button is pressed, the text is set as the title:

```
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 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

endered,
e those
screen, and
ng though,
der.

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 `Window`. I 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:

```
Main.cpp  X
#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.

us game

programming experience. Basically it breaks down to this:

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

e not familiar

w away

(even those

well

optimized to cope with this routine, and maximize efficiency and performance

low 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 thread to sleep for a 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

es on the
later on.

When we want to draw a shape, we have to create the object first. Here is
op.

```
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 floats. There are also classes such as `Vector2i` (for integers), `Vector2u` (for unsigned integers), `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.
rm 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.

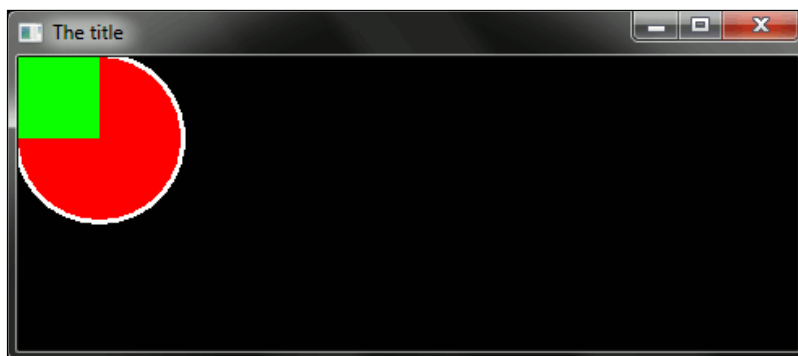
To render the preceding shapes, we can use the `RenderWindow::draw()` function. Here is how we can implement it into our render frame:

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

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

window.display();
```

Running the code produces the following result:



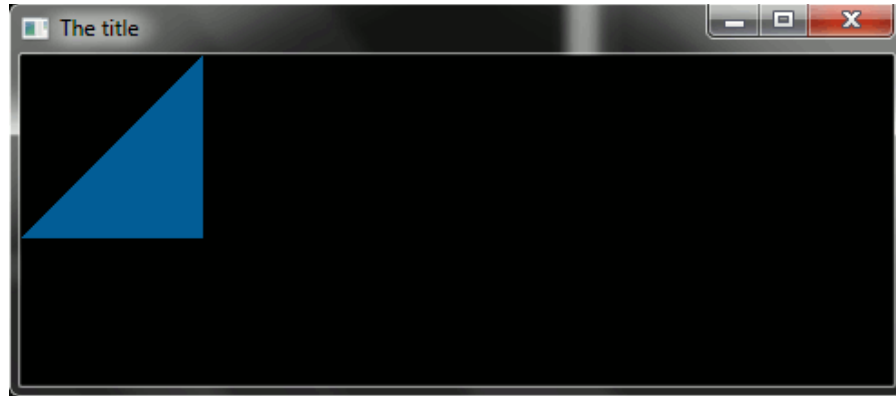
background

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:

```
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 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 50). However

calling

`Transformable::move()` and passing a vector, which indicates how much we want to move the object from its current position.

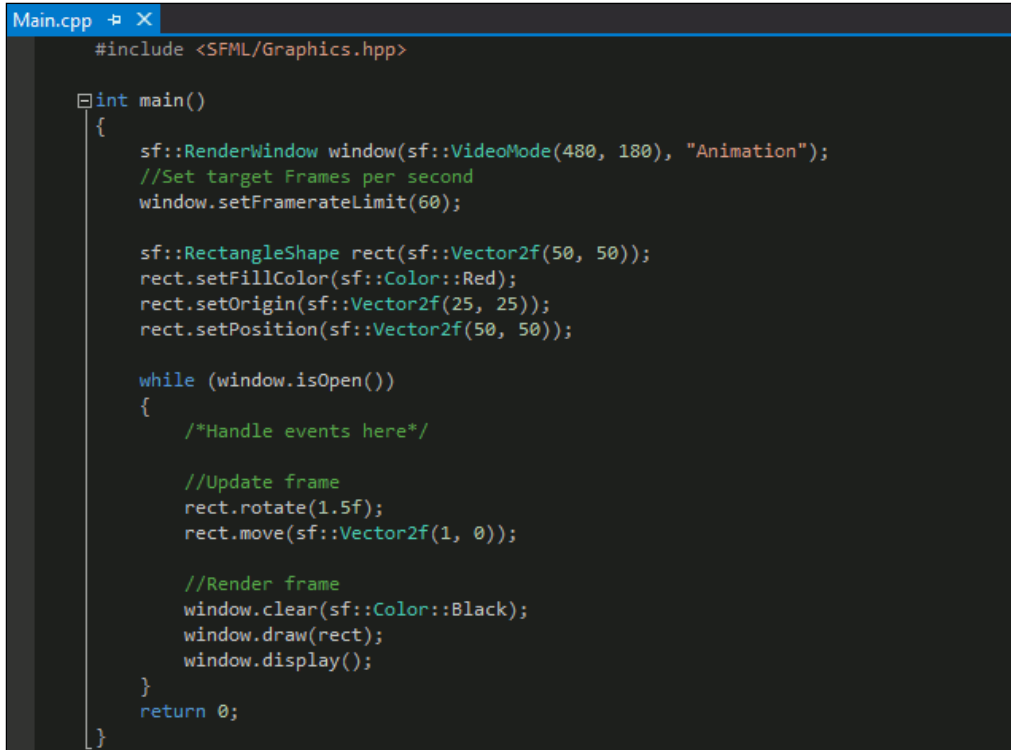
Everything that we have created so far has been mostly static, so now let's add a hich we haven't been able to utilize yet. That's the part of the frame just before we start the render frame. Remember, typically a game frame (loop cycle) goes like this:

- Handle input
- The update frame
- The render frame

current

state will not be rendered correctly — they will be rendered with their state from the last frame.

ion to
create a simple animation:



```

Main.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 In the preceding

s (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 will rotate by approximately 90 degrees (1.5 x 60) and move by 60 pixels to the right (1p x 60). ate) is hile 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

the
t when that
key is released. Here is the screenshot of a code example:

```
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*. 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.

ng much
more pleasant:

```
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.

se
has functions to get its position, the state of any of its buttons, as well as the
) . 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.

specify which
joystick we are looking for with its argument `id`. The functions are as follows:

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

```
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:

```
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 the
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 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);

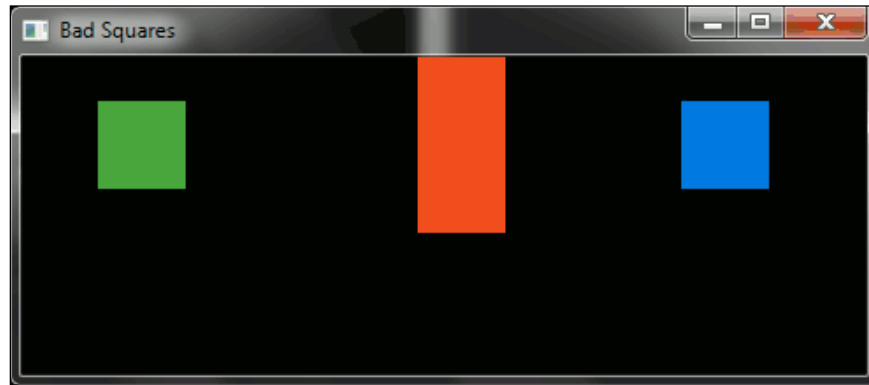
```

to the right—the player has no control over it. You can change that, so that the player can move in any direction, but the only directions in which the player can move are *up* and *down* by using the arrow keys.

logic for both win and lose conditions. We need a method to handle collision detection on shapes 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 smallest rectangle that contains the shape. *Global* and *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 creating rectangles (top, left, bottom, and right values).

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.)

just have
ult will be
similar to this:



ferent
settings (add more shapes and build on the game logic maybe?).

Summary

means
that you are ready for everything that this book offers. This chapter builds the

d and
t,
and how you can use that knowledge to your advantage.

ment.
Stick around – there are treats to be had.

2

Loading and Using Textures

images
s 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

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

ulation

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

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 ports 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
```

n (inform

he

main()

vent 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 providing 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

here

is the `Image::getPixelPtr()` function, which returns the beginning of the

eat 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 original image, beginning at the top-left corner:

```
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:

```
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.

aven't
seen how to display textures on the screen yet. Let's fix this immediately.

Rendering shapes with textures

ous to
to be
ed in the
be rendered
position,
rotation, and so on). However, in SFML, there are renderable classes which can
n 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:

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

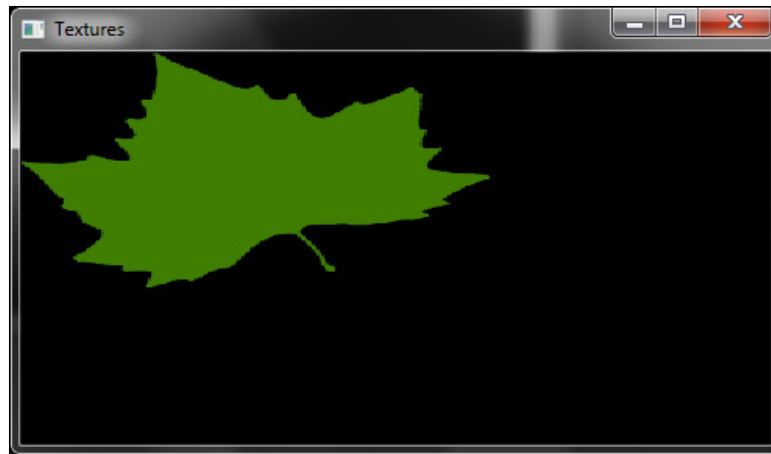
sf::RectangleShape rectShape(sf::Vector2f(300, 150));
rectShape.setTexture(&texture);

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 texture with
&textureo be
rendered. This means that the address, which we pass to the function, has to hold
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
re resource
management techniques in the last section of this chapter.

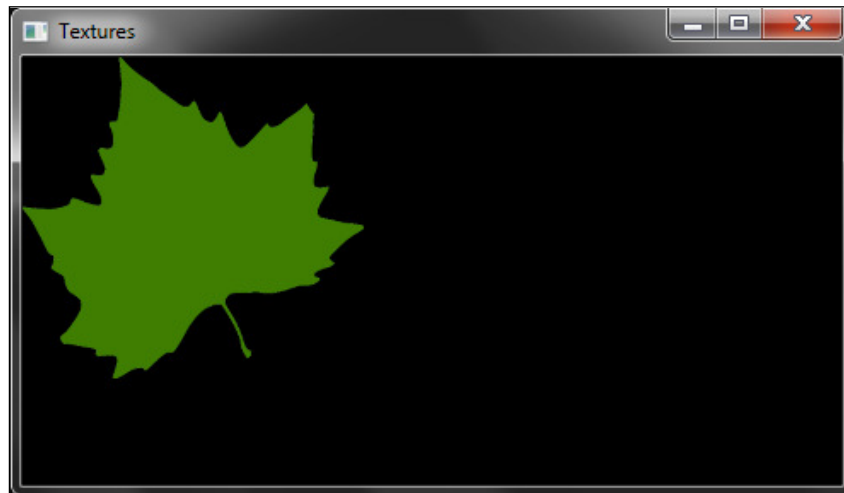
When we place a texture on `RectangleShape`, it tries to fit into the specified rectangle by scaling itself up or down. For our example, if the texture has a width and 150 as height,
then the texture will appear stretched on the *x* axis and squeezed on the *y* axis:



To set the shape to `RectangleShape` with the exact size of the texture, we can use a function from the `Texture` object—`Texture::getSize()`:

```
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.

wn

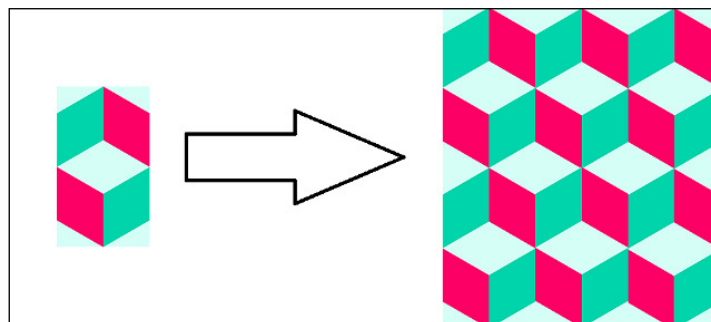
by using ConvexShape:

```
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, 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:



can place many
 of achieving
 this is by creating the larger image in an image editor, such as Microsoft Paint, GIMP,
 not
 nly the tile
 in a GPU memory and use it as a repeated texture over a given surface.

imes 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");

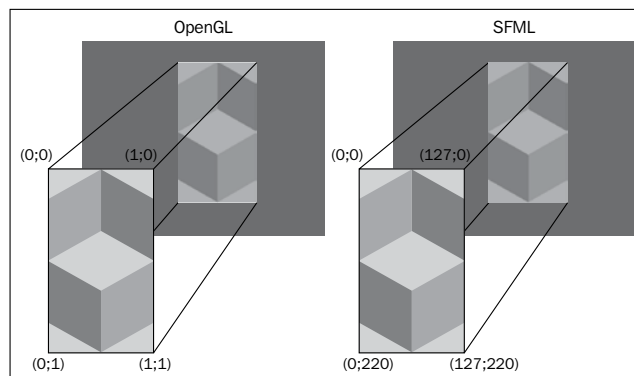
sf::RectangleShape rectShape(sf::Vector2f(128 * 3, 221 * 2));

rectShape.setTexture(&texture);
```

seem
 ectangle.

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`
 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 ($0 \dots 1$; $0 \dots 1$). SFML does
 not use the normalized approach; rather it uses pixel space coordinates ($[0 \dots \text{width} - 1]$, $[0 \dots \text{height} - 1]$). 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.

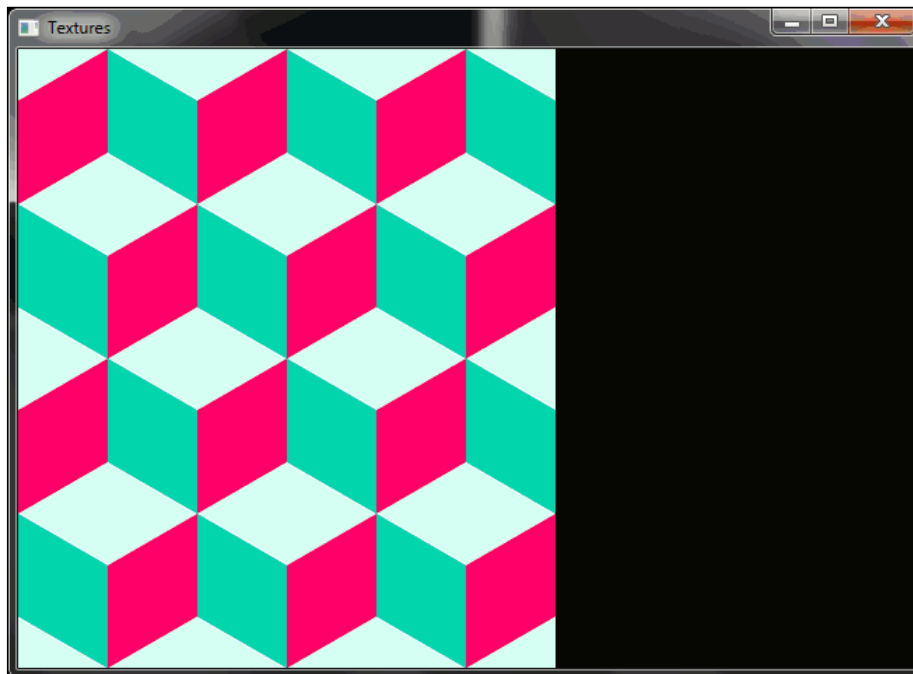
With that in mind, let's see how our code setup will change:

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

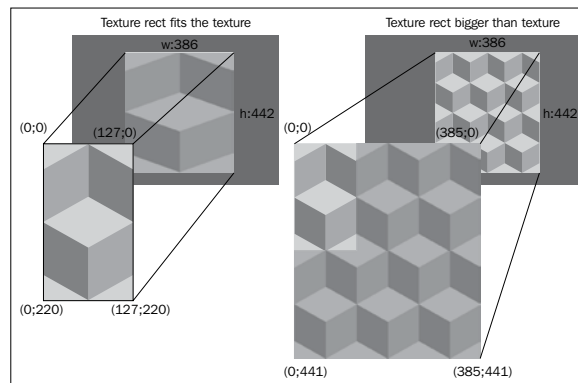
sf::RectangleShape rectShape(sf::Vector2f(128 * 3, 221 * 2));
//Bigger texture rectangle than the size of the texture
rectShape.setTextureRect(sf::IntRect(0, 0, 128 * 3, 221 * 2));

rectShape.setTexture(&texture);
```

The result is exactly what we expected:



illustrates 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:

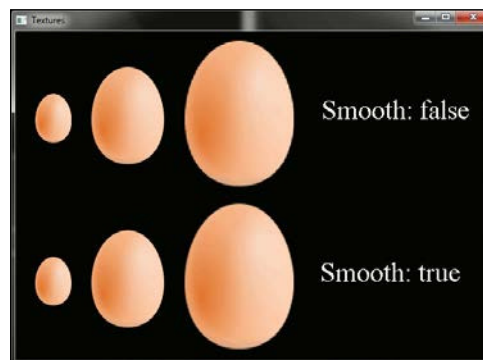


Apart from `Texture::setRepeat()`, there is one other property that changes the way a texture is rendered – the smooth filter, which is controlled by `Texture::setSmooth()`. If we only use texture on surfaces with their original size (pixel perfect), then we won't need this functionality. The function 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:

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

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

erent." 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.

of a
s. This is
exactly why we would want to use it—its simplicity.

Consider the following code:

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

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

. 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`

—`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` ch 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 r 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

to manage assets correctly and efficiently. Making sure that assets do not get important to maintain a solid and efficient code base. In this section, we will talk about building tions.

n they get out of scope. In some languages, all classes are allocated on the heap (where memory is However, easier used. This 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);
}
```

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:

```
AssetManager.h
#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
    //sInstance 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 is done when the

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 achieves the purpose 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:

```
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 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()`, of the e that this is the only instance, we will set the static pointer to the `this` instance.

Here is the `AssetManager::GetTexture()` implementation:

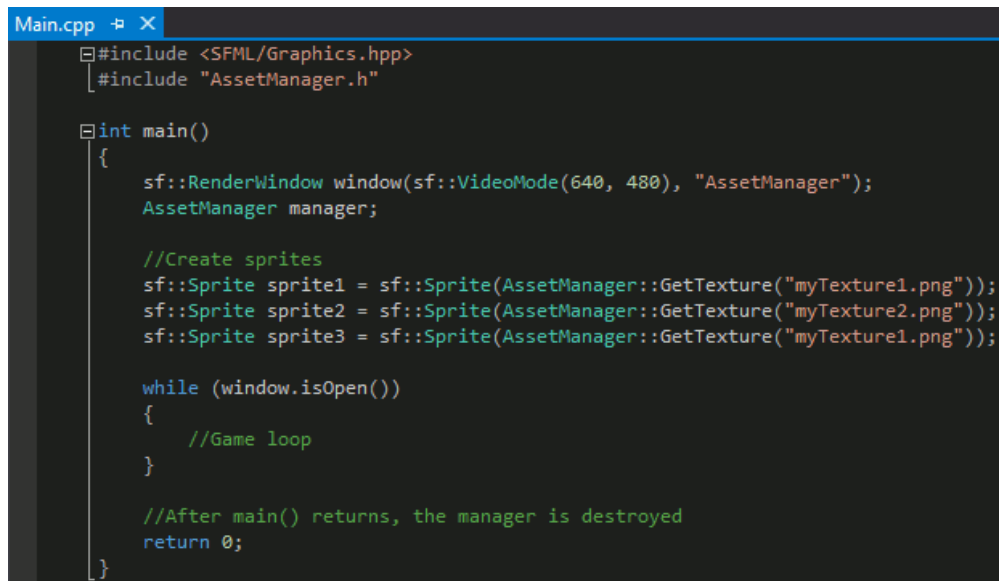
```
sf::Texture& AssetManager::GetTexture(std::string const& filename)
{
    auto& texMap = sInstance->m_Textures;

    //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;
    }
}
```

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 assets. In the next 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`:



```

Main.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;
}

```

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

and images.
s to render
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

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 things going on, we do that, 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

lications.

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 but, as 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, that code can run at the speed of 60 frames per second, doubling the distance, frames take a different amount of time to run on the same machine.

ferent
etween
g
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.

lapsed 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 on 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:

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

The `Time` class capturing it. The `Clock` class provides an interface to measure elapsed time by using the OS clock. It is simple to use as well.

```
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.

e initialize

a Clock to the

t the previous

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:

```
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
}
```

ever

ased events

(to destroy an object after N seconds), and so on. Since we know the delta time

measure

ose the

window 5 seconds after opening it:

```
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, the window after a period of time.

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

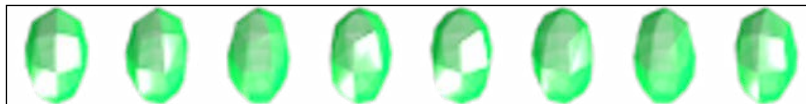
Sprites in action

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

ages that need
es 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 eight frames,
which play for one second. Here is what the sprite sheet looks like:

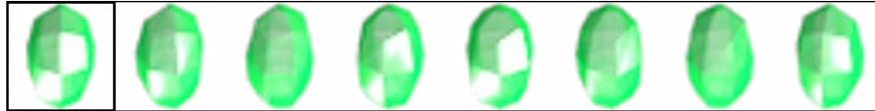


The following screenshot shows our animation setup in code:

```
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:

```
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:

$$18 / 8 = 2 \text{ (and 2 remainder)}$$

Since the `%`me

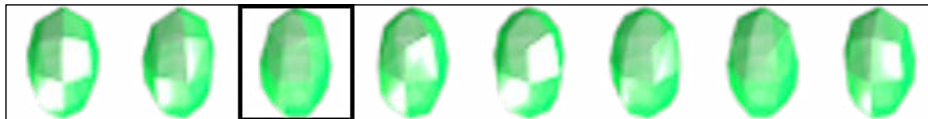
zero as programmers, remember?)

e. 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 sprite'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:

```
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;
        }
    }
};
```

is one more additional method in there—`Animation::AddFrames()`. This particular method is , size, and a frames into the `m_Frames` 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, st, 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 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 should be passed to the constructor of `Animator`.

Other than the `Sprite` reference, there is the time accumulation counter (the same 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, nters 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:

```
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, 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.

s in which we can use them. It makes sense to start from the constructor:

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

te that the `Sprite` 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 `m_CurrentAnimation` member.

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

```
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 lize 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 orks in a moment.

just created.
ames or
changing some of its initial values.

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

```
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 `nullptr` 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 here is its implementation:

```
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:

```
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

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.

rently

which returns the name of the currently playing animation:

```
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:

```
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 (%) 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 them is extremely easy — use the `Animation::m_Frames` vector as an array by calling its index operator with the `currentFrame` index.

That's 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:

```
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.

Next, we know how big the sprite is so, in order to get all the frames from the texture, we start xels 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:

```
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:

```
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);
```

t 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, trates

how to go between animations on key press events:

```
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 rt from the

good idea.

, 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

t there

different

r 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.

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 don't

hen 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?

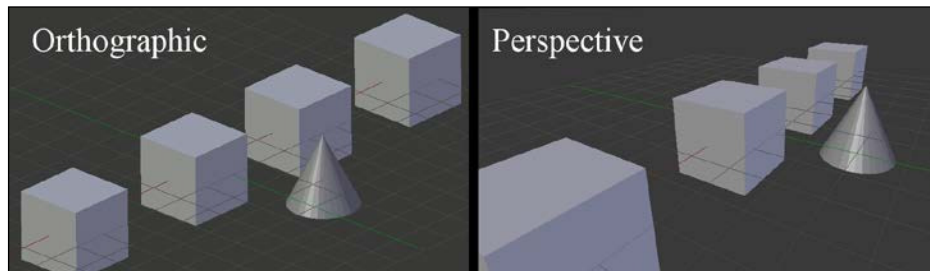
The chance that you've not come across cameras in games development is quite

hat point in

eters

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 other words, the camera actually has no field of view, and the projection is not affected by the distance to the objects. This 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

When should we use a camera?

We don't always have to manipulate the camera. If a game has only one screen, the camera is static, and thus there is no need to do anything. 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 let us work in a world (or level) to explore, we should consider using a camera. We can always create a custom camera 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 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 the center

position (0, 0);

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:

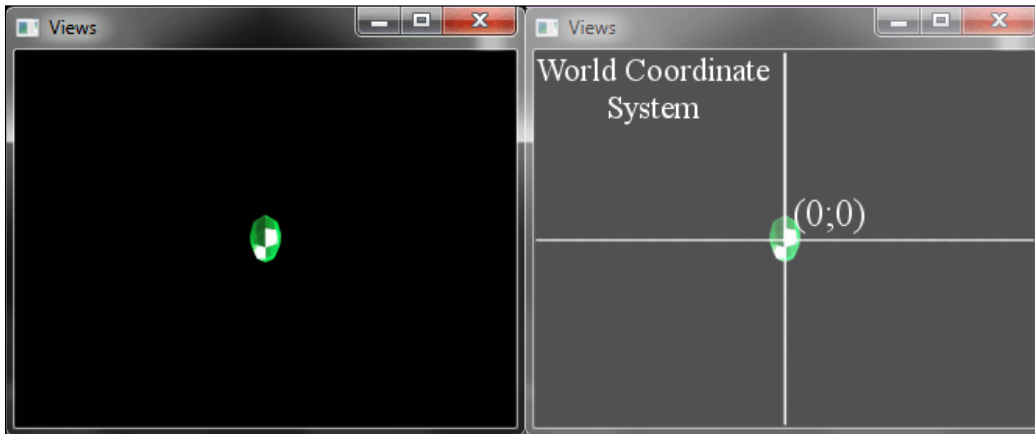
```
auto wSize = window.getSize();
sf::View view(sf::FloatRect(0, 0, wSize.x, wSize.y));

//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:



ctive
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

nce —
hen 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 rease the rotation over a period of time.

very object)
o perform:

```
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));

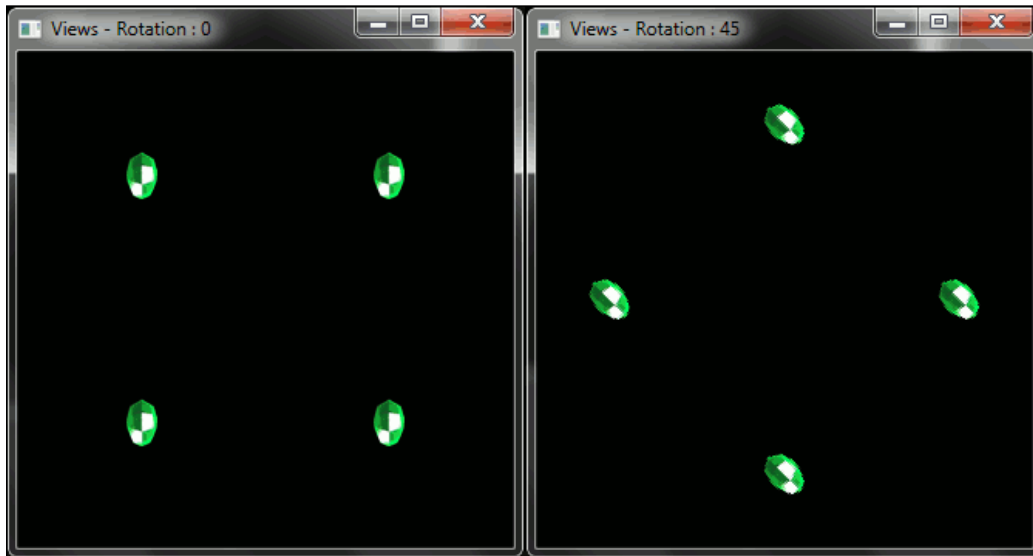
//Top right
sf::Sprite sprite2(texture);
sprite2.setOrigin(spriteSize * 0.5f);
sprite2.setPosition(sf::Vector2f(80, -80));

//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));
```

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.

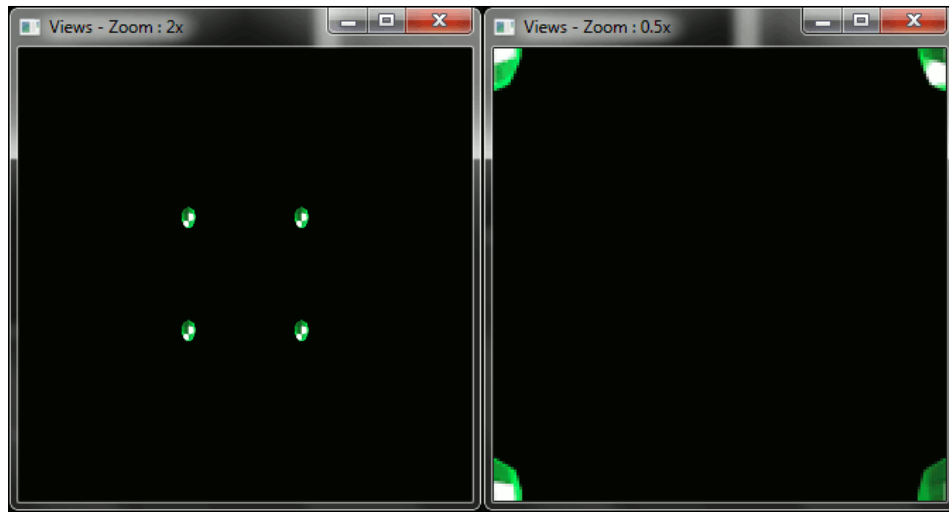
of the screen
around the world point (0;0). The following are the tests done with a rotation of 0
and 45 degrees on the scene:



be used
slow zoom in
along with rotation), taking damage (a slight shake of the camera). We can also use
n 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
his feature
is typically called a **zoom** (either **in** or **out**). The zoom is connected directly to the
ger than the
, if we
want to show objects up close, we have to use zoom by a factor of $1/x$, where x is
s a factor
of $1 / 2 = 0.5$.

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:

$$(1/2) * (1/2) * \text{size} = (1/4) * \text{size}$$

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 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 lar to placing e.

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

Viewports

the window
which uses
e size of the

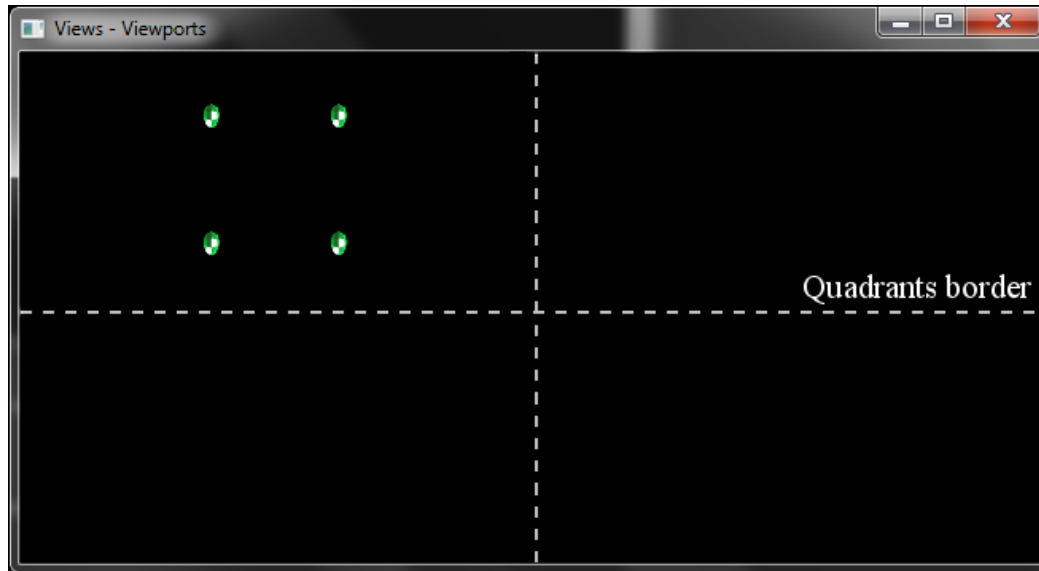
view (0, 0, 1, 1). We can change this by calling `View::setViewport()`. Let's say that we want to render our scene only in the top-left quadrant of the screen. The following code will do it:

```
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:



nder
ne shown
render the
do that, we
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),
anipulate their
transformation and put them in a `viewList` vector. Then our render frame is as
simple as:

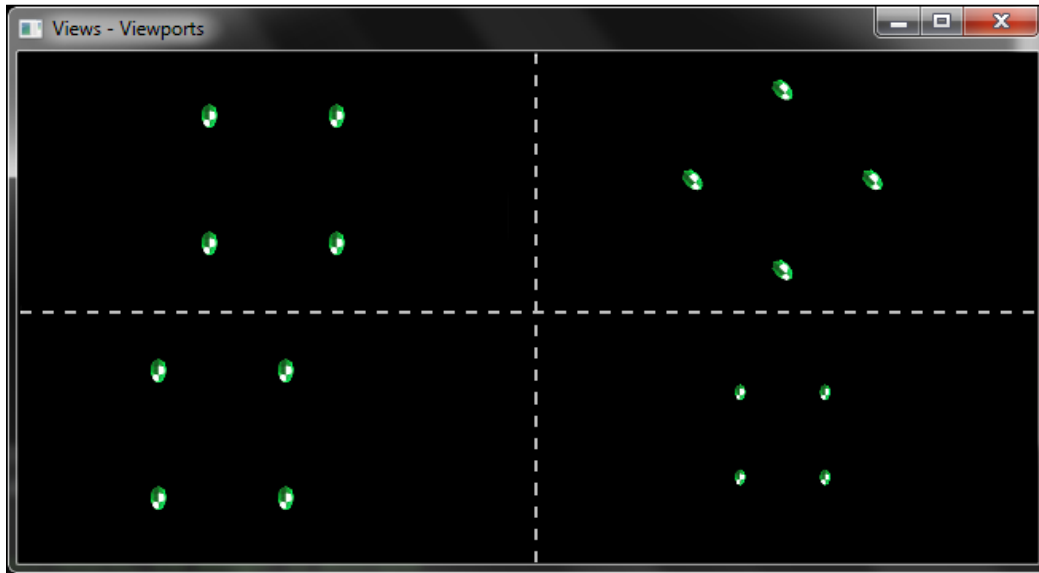
```
window.clear(sf::Color::Black);

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:



at the same time. This makes it possible to create local split screen multiplayer games an example.

UI

camera (which doesn't move with the world), and render our game's UI. Maps and dering 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 osition will

ys, 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 it into world coordinates. Here is an example of how to convert mouse coordinates to world coordinates in a button press event:

```
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 only 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 hardware. It is very useful and has been supported on all graphics cards.

Should you use OpenGL?

s. In fact, once 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 is not too many things.

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 do so on).

Therefore, we have to use OpenGL.

As mentioned at the beginning of this chapter, OpenGL is a huge API, and this gives you a general guideline of how to use it alongside SFML.

Using OpenGL inside SFML

the default framebuffer, and so on) which allows OpenGL to function. This is done automatically when we create a `Window` instance:

```

Main.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
<code>depthBits</code>	The field allows us to suggest the number of bits per pixel for the depth buffer	[0, 8, 16, 24, 32]
<code>stencilBits</code>	The field allows us to suggest the number of bits per pixel for the stencil buffer	[0, 8]
<code>majorVersion</code>	The field allows us to suggest the major version of OpenGL	[1...4]
<code>minorVersion</code>	The field allows us to suggest the minor version of OpenGL	[1...5]
<code>antiAliasingLevel</code>	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 choose 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:

```
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:

```
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 pushing and popping OpenGL states. Let's see how to integrate the 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 `GameObject` 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();
```

we want
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

It is 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 saw how to transform the `View` class, and how we create split screen with 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

title further
of SFML
which are inside the audio module. This includes sound, music, and 3D sound
n 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 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".

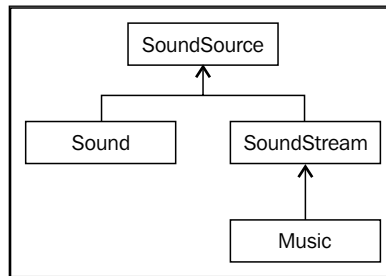
Sound versus music

At first, it seems that the `Sound` class is 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 pace.

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.

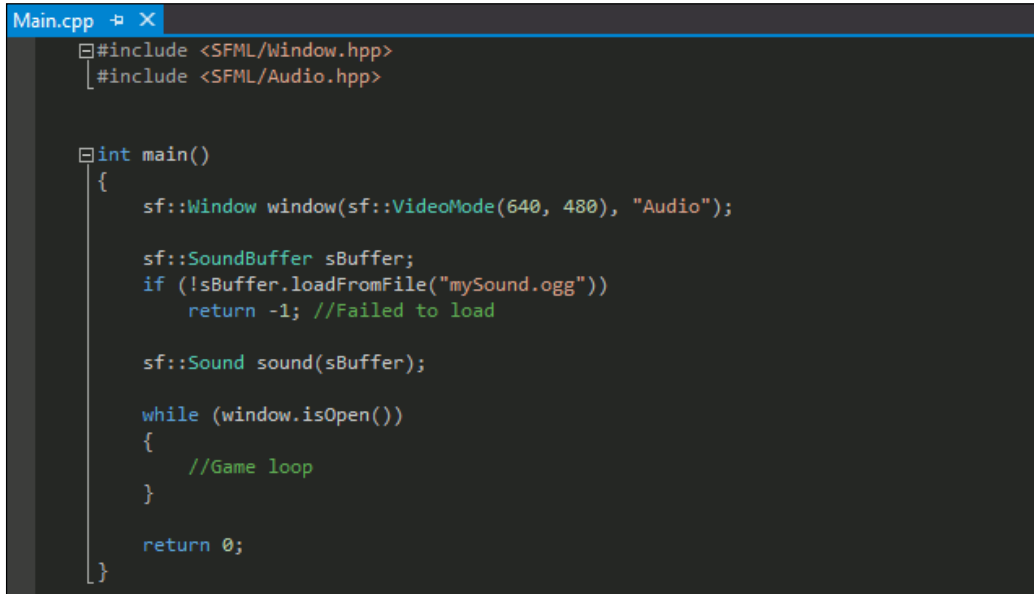
Audio in action

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:



```

Main.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 when we

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.

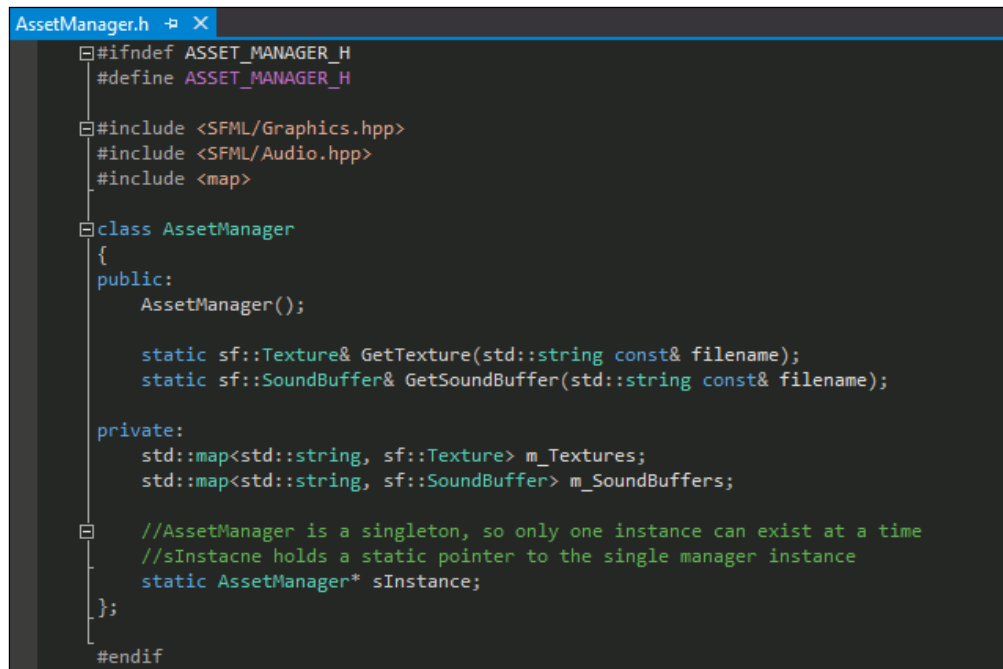
The `sf::Sound` class has another feature, which is utilized by calling `Sound::setPlayingOffset()`. The method takes `sf::Time` and sets the playing position to that time. It can be used in both paused and playing state.

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

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:

A screenshot of a code editor showing the `AssetManager.h` header file. The code is written in C++ and includes SFML headers for graphics and audio, as well as the `<map>` header. It defines a `class AssetManager` with a public constructor and two static methods: `GetTexture` and `GetSoundBuffer`, both taking a filename as a parameter. The private section contains two static maps: `m_Textures` for `sf::Texture` and `m_SoundBuffers` for `sf::SoundBuffer`. A comment indicates that `AssetManager` is a singleton, and a static pointer `sInstance` is declared. The file is enclosed in `#ifndef` and `#endif` guards.

```
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
    //sInstance 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()`:

```
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 file, 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:

```
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;
}
```

accidentally deleted or moved in memory. Also, we can cache resources with the same filename to 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 creates a 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.

file

`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()` reside.

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 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()`.

quency 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 n character, for example). By doing this, we can simulate a realistic sound environment, where, y in our left speaker.

Before we go any further, it is important to note that a sound can be spatialized only 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 following these conventions brings benefits—people understand the code and common examples are more relevant.

set the listener's position to the sprite's position each time it moves:

```
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);
}
```

by calling

`Listener::setDirection()`. This defines the direction which the listener
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.

tion vector

for the listener as well:

```
#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
s important to
note that the angle needs to be in radians for the trigonometric functions to work.
ula:

$$\text{Angle_Radians} = \text{Angle_Degrees} * \text{PI_Radians} / \text{PI_Degrees}$$

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
ndividual
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 moving `Listener` way lized 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:

```
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, re 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()`. cloth rustling, gunshots, and so on). To do that, we just place the sound at the origin (0; 0; 0), and he listener.

Apart from the position, each `SoundSource` has a property called *minimum distance*. sound can

40 units

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 fion

beyond

its minimal distance from the listener. For example, a factor of 1 fades the sound eriod

is

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`):

```
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:

```
//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 is done easily enough by calling `Sound::play()`.

Now, let's look at the update frame:

```
//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 `r` is stationary and, as such, we can opt to omit them. The next four lines move the sound to the `ave` updated the sprite as well.

The `fiso` that should be an easy enough job:

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

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

window.display();
```

es and

To play
with

how the fading
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, displaying here text need to render `ata`). 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:

```
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 `draw`, `translate`, `rotate`, and `scale`.

We can set a few text-specific properties on a `Text` object. One of the important ones are (in pixels) `fontSize` and `lineSpacing`. We can set `fontSize` by calling `Text::setCharacterSize()`. We can also specify a third `fontStyle` (the default value is `0`).

Finally, we can set the text itself. 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:

```
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
ere we want
sor 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*()`.

aded 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:

```
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:

```
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:

```
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

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 will explain what they are and how they are represented in SFML, but we will not cover tutorials covering the subject.

The main focus of this chapter is **shaders**. Shaders are typically the tools which we the graphics 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`

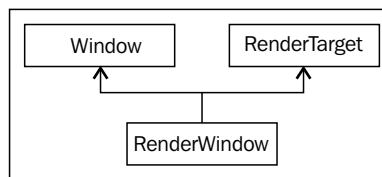
- Rendering directly to a texture
- Shader programming
- Combining everything

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:



way for us
to draw a `Drawable` object in a window. If we look at the base `Window` class we see
s handling
thing 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 `sf::BlendMode::BlendAdd` as 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 far has been to place them on a `sprite` or a `shape`. As useful as that is, textures can also be used elsewhere where 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 a software fallback). The

The `Texture` object itself is held as a field inside the `RenderTexture` class and we can get it by calling `RenderTexture::getTexture()`.

with it. For processing effects with shaders on it. In some cases we might want to create a more complex sprite per entity every frame can become expensive, so `RenderTexture` allows us to improve .

The way we can use a render texture is similar to `RenderWindow`. Firstly, we need to draw on it with the familiar routine:

```
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 $2 \times \text{Radius} = 32$.

After we create our circle shape, we render it onto the texture.

`RenderTarget::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 `RenderTarget::draw()` method and finally display everything that has been drawn so far. The `RenderTarget::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 `RenderTarget::getTexture()`. The method returns a reference to the texture object stored as a field inside `RenderTarget`. In this any purpose that a sprite can be used for.

Like the `Texture` object, `RenderTarget` 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 `RenderTarget` class has some of the `Texture` methods as well, such as `RenderTarget::setSmooth()` and `RenderTarget::setRepeated()`. Also, if we want to start rendering to a render texture using OpenGL, we have to call `RenderTarget::setActive()` to enable the context for the OpenGL API calls. Otherwise, all the OpenGL calls will affect w).

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

Shader programming

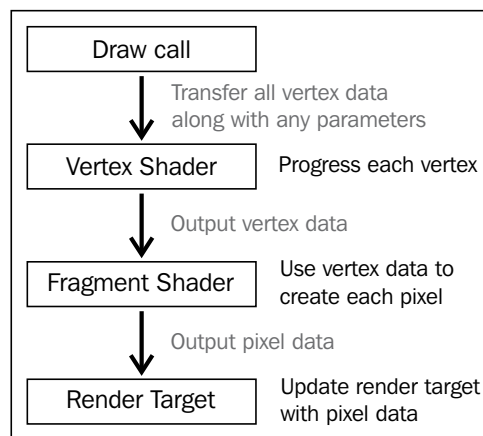
Shader programming has a long history and it exists in many variations. Interesting as they are, this book is not enough to cover even a small portion of the topics in shader programming. In this section, we will briefly mention what shaders are and what support SFML has in terms of loading and using 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/) - <http://www.lighthouse3d.com/tutorials/glsl-tutorial/>.

What is a shader?

in their API
s with any
d 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 on

the render target with the following diagram:



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 $16 \times 16 = 256$ 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.

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. To check

if the GPU supports shaders, you can use the `Shader::isAvailable()` method.

```
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 error occurs, 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 are no exception.

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:

```
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
    //sInstance 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 is that `Shader` instances cannot be copied (which is necessary for the map container). This is what the method's implementation looks like:

```

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:

```

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. We can set these uniforms by the name `Shader::setParameter()` and the uniform inside

```
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).

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 in the fragment shader to interpolate values across vertices.

The shader in the preceding example also uses the Model*View*Projection matrix to r is where the magic happens:

```

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 }

```

o 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, 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 `uTexture` 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);

```

te, which will
ve to change for

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

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

    window.draw(sprite, shader);

    window.display();
}
```

ase it is only
es 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

shader for its draw calls. The code structure for such a scenario looks like this:

```
//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, `t`.

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:

```
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 fihe shader.
nything, as
long as it's not transparent (then we won't actually see anything).

mple
(vertShader.vert) and this new fragment shader, which actually creates the effect:

```
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 }
```

e texture, and
to normalize the
e value of the
ding on our
pixel size). The final line sets the color of the fragment from the texture with the
newly calculated texture coordinate.

niform:

```
//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"));
```

e 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
me, then load
the new scene and start decreasing it back to one.

The next bit is our game loop:

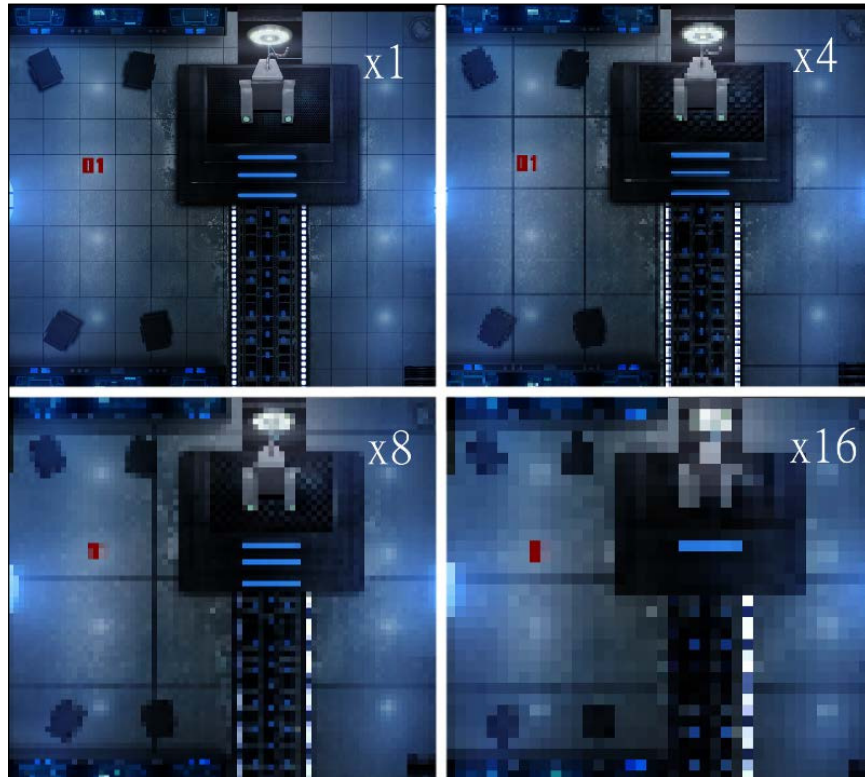
```
while (window.isOpen())
{
    //Handle events
    sf::Event ev;
    while (window.pollEvent(ev)) {}

    /* 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();
}
```


he focus of this chapter. The important bit is at the end where we first render the whole scene to the `RenderTarget` (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 figure out:



And with that we conclude our journey about shaders.

Summary

graphics
and
a lot more
nowadays
that can be used to create amazing-looking scenes.

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

e going
portant

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 (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 network, 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 model 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 dealing with the model

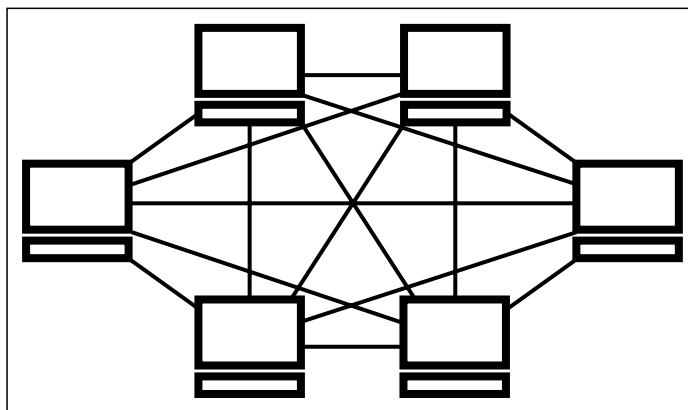
can be found here: http://wikipedia.org/wiki/OSI_model. This covers the very basics of what networking actually is. Next we will be talking about networking in games and a comparison of TCP and UDP, since both of them are widely used in games development.

Networking in games

having a simple high-score table to running a server for a **Massively Multiplayer Online (MMO)** game.

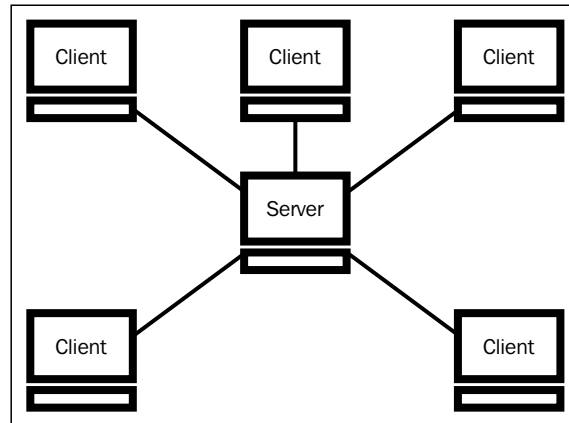
The most interesting (and arguably most difficult) use of networking is for creating multiplayer games. The difficulty comes from the fact that we have to show the same scene on multiple screens at any given time. This is extremely difficult when and varying quality in the underlying networks. Programmers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

n this
ent to update
ons 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 display
n code
(ideally), but they get the state of each dynamic object from the server (position,
tions
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.

ed,
let us talk a bit about the Transport layer of the networking model.

Transport layer – TCP versus UDP

ey will opt
ls exist to
work.
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, which is the IP address field, whereas a packet will arrive at its destination at all (it may get lost along the way), or if it does arrive, the packets 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 uses a sequence number. It tracks the sequence of packets. For example, if the sequence numbers are 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255, 256, 257, 258, 259, 260, 261, 262, 263, 264, 265, 266, 267, 268, 269, 270, 271, 272, 273, 274, 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289, 290, 291, 292, 293, 294, 295, 296, 297, 298, 299, 300, 301, 302, 303, 304, 305, 306, 307, 308, 309, 310, 311, 312, 313, 314, 315, 316, 317, 318, 319, 320, 321, 322, 323, 324, 325, 326, 327, 328, 329, 330, 331, 332, 333, 334, 335, 336, 337, 338, 339, 340, 341, 342, 343, 344, 345, 346, 347, 348, 349, 350, 351, 352, 353, 354, 355, 356, 357, 358, 359, 360, 361, 362, 363, 364, 365, 366, 367, 368, 369, 370, 371, 372, 373, 374, 375, 376, 377, 378, 379, 380, 381, 382, 383, 384, 385, 386, 387, 388, 389, 390, 391, 392, 393, 394, 395, 396, 397, 398, 399, 400, 401, 402, 403, 404, 405, 406, 407, 408, 409, 410, 411, 412, 413, 414, 415, 416, 417, 418, 419, 420, 421, 422, 423, 424, 425, 426, 427, 428, 429, 430, 431, 432, 433, 434, 435, 436, 437, 438, 439, 440, 441, 442, 443, 444, 445, 446, 447, 448, 449, 450, 451, 452, 453, 454, 455, 456, 457, 458, 459, 460, 461, 462, 463, 464, 465, 466, 467, 468, 469, 470, 471, 472, 473, 474, 475, 476, 477, 478, 479, 480, 481, 482, 483, 484, 485, 486, 487, 488, 489, 490, 491, 492, 493, 494, 495, 496, 497, 498, 499, 500, 501, 502, 503, 504, 505, 506, 507, 508, 509, 510, 511, 512, 513, 514, 515, 516, 517, 518, 519, 520, 521, 522, 523, 524, 525, 526, 527, 528, 529, 530, 531, 532, 533, 534, 535, 536, 537, 538, 539, 540, 541, 542, 543, 544, 545, 546, 547, 548, 549, 550, 551, 552, 553, 554, 555, 556, 557, 558, 559, 560, 561, 562, 563, 564, 565, 566, 567, 568, 569, 570, 571, 572, 573, 574, 575, 576, 577, 578, 579, 580, 581, 582, 583, 584, 585, 586, 587, 588, 589, 590, 591, 592, 593, 594, 595, 596, 597, 598, 599, 600, 601, 602, 603, 604, 605, 606, 607, 608, 609, 610, 611, 612, 613, 614, 615, 616, 617, 618, 619, 620, 621, 622, 623, 624, 625, 626, 627, 628, 629, 630, 631, 632, 633, 634, 635, 636, 637, 638, 639, 640, 641, 642, 643, 644, 645, 646, 647, 648, 649, 650, 651, 652, 653, 654, 655, 656, 657, 658, 659, 660, 661, 662, 663, 664, 665, 666, 667, 668, 669, 670, 671, 672, 673, 674, 675, 676, 677, 678, 679, 680, 681, 682, 683, 684, 685, 686, 687, 688, 689, 690, 691, 692, 693, 694, 695, 696, 697, 698, 699, 700, 701, 702, 703, 704, 705, 706, 707, 708, 709, 710, 711, 712, 713, 714, 715, 716, 717, 718, 719, 720, 721, 722, 723, 724, 725, 726, 727, 728, 729, 730, 731, 732, 733, 734, 735, 736, 737, 738, 739, 740, 741, 742, 743, 744, 745, 746, 747, 748, 749, 750, 751, 752, 753, 754, 755, 756, 757, 758, 759, 760, 761, 762, 763, 764, 765, 766, 767, 768, 769, 770, 771, 772, 773, 774, 775, 776, 777, 778, 779, 780, 781, 782, 783, 784, 785, 786, 787, 788, 789, 790, 791, 792, 793, 794, 795, 796, 797, 798, 799, 800, 801, 802, 803, 804, 805, 806, 807, 808, 809, 810, 811, 812, 813, 814, 815, 816, 817, 818, 819, 820, 821, 822, 823, 824, 825, 826, 827, 828, 829, 830, 831, 832, 833, 834, 835, 836, 837, 838, 839, 840, 841, 842, 843, 844, 845, 846, 847, 848, 849, 850, 851, 852, 853, 854, 855, 856, 857, 858, 859, 860, 861, 862, 863, 864, 865, 866, 867, 868, 869, 870, 871, 872, 873, 874, 875, 876, 877, 878, 879, 880, 881, 882, 883, 884, 885, 886, 887, 888, 889, 890, 891, 892, 893, 894, 895, 896, 897, 898, 899, 900, 901, 902, 903, 904, 905, 906, 907, 908, 909, 910, 911, 912, 913, 914, 915, 916, 917, 918, 919, 920, 921, 922, 923, 924, 925, 926, 927, 928, 929, 930, 931, 932, 933, 934, 935, 936, 937, 938, 939, 940, 941, 942, 943, 944, 945, 946, 947, 948, 949, 950, 951, 952, 953, 954, 955, 956, 957, 958, 959, 960, 961, 962, 963, 964, 965, 966, 967, 968, 969, 970, 971, 972, 973, 974, 975, 976, 977, 978, 979, 980, 981, 982, 983, 984, 985, 986, 987, 988, 989, 990, 991, 992, 993, 994, 995, 996, 997, 998, 999, 1000. They will arrive in the same order: 1, 2, 3, 4.

On the other hand, **User Datagram Protocol (UDP)** is unreliable and does not guarantee that packets arrive in the correct order. This technique requires much less memory for each packet and doesn't require any wait time for packets to arrive.

When considering which protocol to use, we should ask ourselves one question, "What is the application's requirements?" For example, in a business application, we might want to guarantee that packets arrive in the correct order and they can afford to sacrifice a few milliseconds of wait time to achieve that. Some games fall into the point-and-click category, and even some puzzlers use TCP with no problems. In most cases though, for real-time applications (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, in a first person shooter, losing a few packets might not be a big deal). In the end, we have to weigh the trade-offs. We want to get as much better performance out of sending and receiving packets.

TCP in SFML

each TCP
kes it
e 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

work
ates 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:

```
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:

```
//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;
```

calling

the `TcpListener::listen()` method and passing the port, onto which we are expecting the connection. The `TcpListener::accept()` method waits until a connection is accepted 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 `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 going for connection stays after the calling:

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

Now, let's move on to UDP.

UDP in SFML

ected to the destinations with no problem.

at 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
}
```

emote 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 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:

```
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 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 `isReady()` method, which returns `true` if the data transfer is ready. For example calling `TcpListener::accept()` returns `Socket::NotReady` if there is no socket currently waiting to be connected. If the data is 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:

```
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 function.

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 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 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 it same packet has been received on the other end.

The `Packet` to write to it and the `>>>` operator to read it:

```
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;
```

:

```
/* 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:

```
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;
}
```

iface:

```
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;
```

e array,
we use the `*::send()` and `*::receive()` methods of the sockets. We start with the TCP client-side:

```
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 */
}
```

making
with a final example.

Putting it all into practice

layers who
t, 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  ▢ ✕
#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` sockets.

s can be

ple.

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** r:

```
//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 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 initialize the scene:

```
//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:

```
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;
        }
    }
}
```

nd assigns

We will

e 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:

```
//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;
    }
}
```

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.

rst we

check if we have to move at all. If we don't have to move, we don't have to update

s prevents

shape

ket of its

t be a problem

ection,

on problem

and exit the program.

The final
simple – we just have to render the two shapes:

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

window.draw(localShape);
window.draw(remoteShape);

window.display();
}
```

orking
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
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.
ks
and materials online about game development.

Index

Symbols

3D

audio 89

`sf::String` 16

A

`Animation::AddFrames()` method 55

animator

building 54-59

considerations 54

multiple animations, creating 61, 62

using 60

`Animator::CreateAnimation()`

method 56, 57

`Animator::SwitchAnimation(Animation*)`

method 57

`Animator::Update()` method 55, 56, 59

`AssetManager 2.0` 86-88

`AssetManager 3.0` 97, 98

`AssetManager 4.0` 108, 109

`AssetManager` class 52

`AssetManager::GetTexture()` function 43

audio, 3D

about 89, 90

features 89

features, summarizing 93-95

listener, setting up 90, 91

sources 92, 93

audio module

about 81-83

`AssetManager 2.0` 86

`sf::Music` 88, 89

`sf::Sound` 83

`sf::SoundStream` 88, 89

Axis-Aligned Bounding Box (AABB) 42

B

big endian processors 128

blocking socket 126

boolean 92

bounding box 27

C

camera

about 63, 64

implementing, by SFML 64, 65

manipulating, with `sf::View` 65, 66

using 64

`Clock::getElapsedTime()` function 49

`Clock::restart()` function 49

ContextSettings

about 10

`antialiasingLevel` 10

`depthBits` 10

`majorVersion` 10

`minorVersion` 10

`stencilBits` 10

ContextSettings struct class

`antialiasingLevel` field 75

`depthBits` field 75

`majorVersion` field 75

`minorVersion` field 75

`stencilBits` field 75

`ConvexShape::setPointCount()` function 20

`ConvexShape::setPoint()` function 20

coordinates

mapping 73

D

Drawable class 41

E

event handling

- about 11
- joystick related events 14
- keyboard related events 13
- mouse related events 13
- window related events 12

events

- Event::EventType 14
- using 14-16

F

Fast Approximate Antialiasing (FXAA) 113

FloatRect::intersects() function 42

framebuffer object 103

G

game loop 10, 11

GameObject

- render() method 77
- update() method 77

games

- networking 119, 120

GLSL 1.2 Tutorial

- URL 105

Graphics Processing Unit (GPU) 29

H

host 131

I

Image class 30

Image::create() function 30

Image::flipHorizontally() function 32

Image::flipVertically() function 32

Image::getPixel() function 32

Image::getPixelPtr() function 32

images

- creating 30-32
- versus textures 30

Image::saveToFile() function 32

Image::setPixel() function 32

J

Joystick::getAxisPosition() function 25

Joystick::getButtonCount() function 25

Joystick::hasAxis() function 25

Joystick::isButtonPressed() function 25

Joystick::isConnected() function 25

joystick related events

- Event::JoystickButtonPressed 14
- Event::JoystickButtonReleased 14
- Event::JoystickConnected 14
- Event::JoystickDisconnected 14
- Event::JoystickMoved 14

K

keyboard related events

- Event::KeyPressed 13
- Event::KeyReleased 13
- Event::TextEntered 13

L

listener

- setting up 90, 91

Listener::setDirection() function 91

Listener::setGlobalVolume() function 91

Listener::setPosition() function 90

little endian family processors 128

M

Massively Multiplayer Online (MMO) 118

modulo operator (%) 60

mouse cursor

- disabling 10

Mouse::getPosition() function 25

mouse related events

- Event::MouseButtonPressed 13
- Event::MouseMove 13
- Event::MouseWheelMoved 13

Mouse::setPosition() function 25

multiplayer games

- implementing 130-134

multiple animations

creating 61, 62

Music class

versus Sound class 82, 83

N

networking

about 117, 118

in games 119, 120

non-blocking sockets 126

O

OpenGL

about 74

and sf::Shader 113

using 74

using, in multiple windows 79

using, inside SFML 74-77

orthographic projection 64

P

packets

constructing 128-130

exchanging 127, 128

pixelation 113

Polygon triangulation 21

R

rasterization 106

RectangleShape::setOrigin() function 23

render frame 17, 18

RenderTarget::draw() method 103

RenderTexture class

about 103, 113

setting up 114-116

RenderTexture::display() method 105

RenderTexture::draw() method 105

RenderTexture::getTexture() method 105

RenderTexture::setSmooth() method 105

RenderWindow class 75

RenderWindow::draw() function 20

RenderWindow::mapCoordsToPixel() method 73

resources

managing 42-45

RPG (Role Playing Game) 64

S

sf::Clock 49, 50

sf::microseconds() function 49

sf::milliseconds() function 49

SFML

audio module 81

camera, implementing 65

OpenGL, using 74-78

TCP 122

SFML styles

sf::Style::Close 9

sf::Style::Default 9

sf::Style::Fullscreen 9

sf::Style::None 9

sf::Style::Resize 9

sf::Style::Titlebar 9

sf::Music 88, 89

sf::RenderStates 103

sf::RenderTarget 102, 103

sf::RenderWindow 102, 103

sf::seconds() function 49

sf::Shader

and OpenGL 113

sf::Sound class 83-85

sf::SoundSource 89

sf::SoundStream 88, 89

sf::Text

about 95-97

AssetManager 3.0 97, 98

sf::Time 49, 50

sf::View

used, for manipulating camera 65, 66

shader

about 101-107

loading 107, 108

using 109

shader programming

about 105

uniforms, setting 110

shader uniforms

setting 110-113

shapes

- controlling 24-28
- drawing 19-21
- rendering 17
- rendering, with textures 34-39
- transforming 21-24
- versus sprites 40

Shape::setFillColor() function 40

Shape::setPosition() function 21

SoundBuffer class 83

SoundBuffer::loadFromMemory()
method 84

SoundBuffer::loadFromStream()
method 84

Sound class

- versus Music class 82, 83

Sound::pause() method 85

Sound::play() method 85

SoundSource::setRelativeToListener()
method 92

Sound::stop() method 85

SoundStream::pause() method 88

SoundStream::play() method 88

SoundStream::stop() method 88

Sprite class 29, 41

sprites

- about 40
- animating 51
- animating, setup 51-53
- Drawable class 41
- final facts 41
- Transformable class 41
- versus shapes 40

Sprite::setColor() function 40

sprite sheet 51

Style argument 9

T

TCP

- in SFML 122, 123
- URL 118
- versus UDP 120, 121

TcpListener::accept() method 123

TcpSocket::connect() method 122

TcpSocket::receive() method 123

Text::findCharacterPos() method 97

Text::setCharacterSize() method 96

Text::setString() method 96

Text::setStyle() method 97

Texture class 30

Texture::getMaximumSize() function 33

Texture::getSize() function 35

Texture::loadFromFile() 33

textures

- about 29
- creating 32, 33
- loading 29
- rendering 103-105
- shapes, rendering with 34-39
- versus images 30

Texture::setRepeated() function 37

Texture::setSmooth() property 39

time

- capturing 47, 48
- sf::Clock 49
- sf::Time 49

Time::asMicroseconds() function 49

Time::asMilliseconds() function 49

Time::asSeconds() function 49

tinting 113

Transformable class 41

transformations 17

Transmission Control Protocol. *See* TCP

U

User Datagram Protocol (UDP)

- in SFML 125, 126
- URL 118
- versus TCP 121

V

VideoMode class 8

VideoMode::isValid() method 9

view

- rotating 66-70
- scaling 66-70

View::move() method 69

viewport 70-72

View::rotate() method 67

View::setRotation() method 67

View::zoom() method 69

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



Thank you for buying **SFML Essentials**

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*

highly focused books on specific technologies and solutions.

ssionals in adapting

and customizing today's systems, applications, and frameworks. Our solution-based books

ies 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.

g 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

rprise, in order

Open Source brand,

home to books published on software built around open source licenses, and offering

. The Open

t gives a royalty

to each open source project about whose software a book is sold.

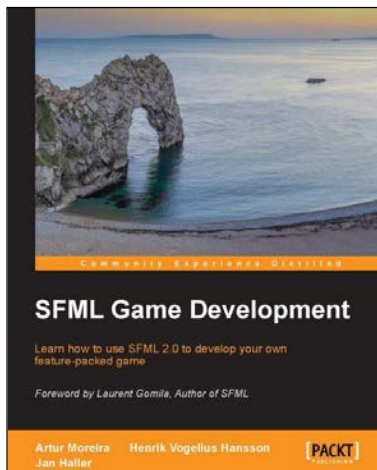
Writing for Packt

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.

al skills but no writing

r, or simply get some

additional reward for your expertise.



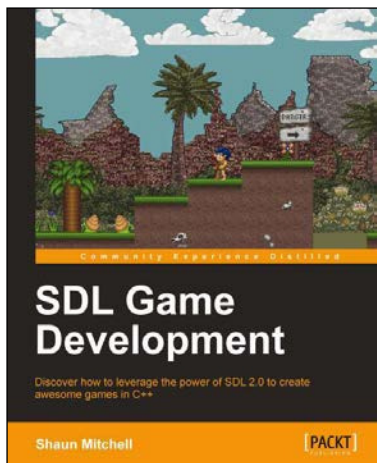
SFML Game Development

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