SFML Blueprints

SFML is a cross-platform software development library written in C++ with bindings available for many programming languages. This book contains useful information that you’ll need to create any kind of 2D games. Starting with the basics of game programming, resource management, and building simple 2D games, you’ll then delve into more complex features such as the physics engine, constructing a game UI, and more advanced concepts of 2D gaming using SFML. You’ll develop advanced functionality in your game using the concept of multithreading and learn how various threads interact. In later chapters, you’ll quickly grasp the usage and implementation of isometric views and image transformation in your 2D real-time tower defense game. You’ll wrap up by adding networking and database management systems to your game with SQLite using an ORM.

Who this book is written for

This book is for developers who have knowledge of the basics of the SFML library and its capabilities in 2D game development. Minimal experience with C++ is required.

What you will learn from this book

- Build a complete game and integrate advanced features by adding a multiplayer layer
- Get to grips with SFML resources and build a generic and reusable resource manager
- Gather knowledge about different entity models and build your own 2D games
- Explore the Box2D engine and add physics properties to your game
- Add a nice user interface to your game to make it more user-friendly
- Discover the SFGUI library and learn how to customize your game
- Delve into the importance of multithreading and boost your code
- Add networking and learn about serialization and database management using SQLite

Maxime Barbier


Free Sample
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Preparing the Environment'
- A synopsis of the book’s content
- More information on SFML Blueprints

About the Author

Maxime Barbier has recently finished his studies and is now a software engineer in Strasbourg, France. He loves programming crazy things and has been experimenting and sharing them with the open source community on GitHub since 2010. Also, he really likes game programming.

As his favorite technology is C++, he has become an expert in it because of his work. He has also developed several libraries with this language, and some of them are used in this book. Game programming is his hobby, and he really likes the challenges involved in such a project. He also loves sharing his knowledge with other people, which was the main reason he wrote this book and also the reason for his activity in the open source community.

Since 9 years, he has been working on different projects such as Anka Dreles, which is a pen and paper role-playing game, and is putting in effort to convert it into a computer game.

He also loves sailing and was a sailing teacher for several years while studying. His dream is to be able to combine sailing and computer sciences by traveling around the world.

Before starting with this book, Maxime had already reviewed some books, such as SFML Game Development and Getting Started with OUYA, both by Packt Publishing.
SFML Blueprints

Throughout this book, I'll try to share my knowledge on how to make video games and share them with you. Five different projects will be covered, which include many techniques and ways to resolve quite commons problems involved in game development.

The technologies used are the C++ programming language (2011 standard) and the SFML library (version 2.2).

Many aspects of game programming are developed over the different chapters and give you all the keys in hand to build every kind of game you want in 2D, with the only limit of your imagination.

What This Book Covers

Chapter 1, Preparing the Environment, helps you install everything needed for this book, and build a small application using SFML to test whether everything is fine.

Chapter 2, General Game Architecture, User Inputs, and Resource Management, explains general game architectures, managing user inputs and finally, how to keep track of external resources.

Chapter 3, Making an Entire 2D Game, helps you build Asteroid and Tetris clones, learning entity models and board management.

Chapter 4, Playing with Physics, provides a description of physical engines. It also covers the usage of Box2D paired with SFML, and turns our Tetris into a new game, Gravitris.

Chapter 5, Playing with User Interfaces, helps you create and use a game user interface. It introsductes you to SFGUI and adding them to our Gravitris game.

Chapter 6, Boost Your Code Using Multithreading, introduces multithreading and adapts our game to use it.

Chapter 7, Building a Real-time Tower Defense Game from Scratch – Part 1, helps you create animations, a generic tile map system (isometric hexagonal tiles), and an entity system. Finally, you will create all the game logic.

Chapter 8, Build a Real-time Tower Defense Game from Scratch – Part 2, Networking, introduces network architectures and networking. It helps you create a custom communication protocol, and modify our game to allow multiplayer matches over the network. Then, we finally add a save/load option to our game using Sqlite3 through an ORM.
Preparing the Environment

Through this book, I will try to teach you some elements to build video games using the SFML library. Each chapter will cover a different topic, and will require knowledge from the previous one.

In this first chapter, we will cover basics points needed for the future such as:

- Installing a compiler for C++11
- Installing CMake
- Installing SFML 2.2
- Building a minimal SFML project

Before getting started, let’s talk about each technology and why we will use them.

**C++11**

The C++ programming language is a very powerful tool and has really great performance, but it is also really complex, even after years of practice. It allows us to program at both a low and high level. It's useful to make some optimizations on our program such as having the ability to directly manipulate memory. Building software utilizing C++ libraries allows us to work at a higher level and when performance is crucial, at a low level. Moreover, the C/C++ compilers are very efficient at optimizing code. The result is that, right now, C++ is the most powerful language in terms of speed, and thanks to the zero cost abstraction, you are not paying for what you don't use, or for the abstraction you are provided.
Preparing the Environment

I'll try to use this language in a modern way, using the object-oriented approach. Sometimes, I'll bypass this approach to use the C way for optimizations. So do not be shocked to see some "old school code". Moreover, all the main compilers now support the standard language released in 2011, so we can use it everywhere without any trouble. This version adds some really useful features in the language that will be used in this book, such as the following:

- Keywords are one such important feature. The following are a few of them:
  - **auto**: This automatically detects the type of the new variable. It is really useful for the instantiation of iterators. The auto keyword already existed in the past, but has been deprecated for a long time, and its meaning has now changed.
  - **nullptr**: This is a new keyword introducing a strong type for the old NULL value. You can always use NULL, but it's preferable to use nullptr, which is any pointer type with 0 as the value.
  - **override** and **final**: These two keywords already exist in some languages such as Java. These are simple indications not only for the compiler but also for the programmer, but don't specify what they indicate. Don't hesitate to use them. You can take a look to the documentation of them here [http://en.cppreference.com/w/cpp/language/override](http://en.cppreference.com/w/cpp/language/override) and [http://en.cppreference.com/w/cpp/language/final](http://en.cppreference.com/w/cpp/language/final).

- The range-based **for** loops is a new kind of loop in the language **foreach**. Moreover, you can use the new **auto** keyword to reduce your code drastically. The following syntax is very simple:
  
  ```
  for(auto& var : table){...}.
  ```
  
  In this example, **table** is a container (vector and list) and **var** is a reference to the stored variable. Using & allows us to modify the variable contained inside the table and avoids copies.

- C++11 introduces the smart pointers. There are multiple pointers corresponding to their different possible utilizations. Take a look at the official documentation, this which is really interesting. The main idea is to manage the memory and delete the object created at runtime when no more reference on it exists, so that you do not have to delete it yourself or ensure that no double free corruptions are made. A smart pointer created on the stack has the advantages of being both fast and automatically deleted when the method / code block ends. But it is important to know that a strong use of this pointer, more especially **shared_ptr**, will reduce the execution speed of your program, so use them carefully.
Chapter 1

• The lambda expression or anonymous function is a new type introduced with a particular syntax. You can now create functions, for example, as a parameter of another function. This is really useful for callback. In the past, functor was used to achieve this kind of comportment. An example of functor and lambda is as follows:

```cpp
class Func(){ void operator()(){/* code here */}};
auto f = [](){/* code here*/};
```

• If you already know the use of the variadics function with the ellipse operator (...), this notion should trouble you, as the usage of it is different. The variadics template is just the amelioration of template with any number of parameters using the ellipse operator. A good example for this is the tuple class. A tuple contains any number of values of any type known at compile time. Without the variadics template, it was not really possible to build this class, but now it is really easy. By the way, the tuple class was introduced in C++11. There are several other features, such as threads, pair, and so on.

**SFML**

SFML stands for **Simple and Fast Multimedia Library**. This is a framework written in C++ and is based on OpenGL for its graphical rendering part. This name describes its aim pretty well, that is, to have a user-friendly interface (API), to deliver high performance, and to be as portable as possible. The SFML library is divided into five modules, which are compiled in a separated file:

• **System**: This is the main module, and is required by all others. It provides clocks, threads, and two or three dimensions with all their logics (mathematics operations).

• **Window**: This module allows the application to interact with the user by managing windows and the inputs from the mouse, keyboard, and joystick.

• **Graphics**: This module allows the user to use all the graphical basic elements such as textures, shapes, texts, colors, shaders, and more.

• **Audio**: This module allows the user to use some sound. Thanks to this, we will be able to play some themes, music, and sounds.

• **Network**: This module manages not only socket and type safe transfers but also HTTP and FTP protocols. It's also very useful to communicate between different programs.
Preparing the Environment

Each module used by our programs will need to be linked to them at compile time. We don't need to link them all if it's not necessary. This book will cover each module, but not all the SFML classes. I recommend you take a look at the SFML documentation at http://www.sfml-dev.org/documentation.php, as it's very interesting and complete. Every module and class is well described in different sections.

Now that the main technologies have been presented, let's install all that we need to use them.

Installation of a C++11 compiler
As mentioned previously, we will use C++11, so we need a compiler for it. For each operating system, there are several options; choose the one you prefer.

For Linux users
If you are a Linux user, you probably already have GCC/G++ installed. In this case, check whether your version is 4.8 or later. Otherwise, you can install GCC/G++ (version 4.8+) or Clang (version 3.4+) using your favorite packet manager. Under Debian based distribution (such as Ubuntu and Mint), use the command line:

```
sudo apt-get install gcc g++ clang -y
```

For Mac users
If you are a Mac user, you can use Clang (3.4+). This is the default compiler under Mac OS X.

For Windows users
Finally, if you are a Windows user, you can use Visual Studio (2013), Mingw-gcc (4.8+), or Clang (3.4+) by downloading them. I suggest you not use Visual Studio, because it's not 100 percent standard compliant, even for the C99, and instead use another IDE such as Code::Blocks (see the following paragraph).

For all users
I assume that in both cases, you have been able to install a compiler and configure your system to use it (by adding it to the system path). If you have not been able to do this, another solution is to install an IDE like Code::Blocks, which has the advantage of being installed with a default compiler, is compatible with C++11, and doesn't require any system configuration.
I will choose the IDE option with Code::Blocks for the rest of the book, because it does not depend on a specific operating system and everyone will be able to navigate. You can download it at http://www.codeblocks.org/downloads/26. The installation is really easy; you just have to follow the wizard.

**Installing CMake**

CMake is a really useful tool that manages the build process in any operating system and in a compiler-independent manner. This configuration is really simple. We will need it to build the SFML (if you choose this installation solution) and to build all the future projects of this book. Using CMake gives us a cross-platform solution. We will need version 2.8 or later of CMake. Currently, the last stable version is 3.0.2.

**For Linux users**

If you use a Linux system, you can install CMake and its GUI using your packet manager. For example, under Debian, use this command line:

```
sudo apt-get install cmake cmake-gui -y
```

**For other operating systems**

You can download the CMake binary for your system at http://www.cmake.org/download/. Follow the wizard, and that's it. CMake is now installed and ready to be used.

**Installing SFML 2.2**

There are two ways to get the SFML library. The easier way is to download the prebuilt version, which can be found at http://sfml-dev.org/download/sfml/2.2/, but ensure that the version you download is compatible with your compiler.

The second option is to compile the library yourself. This option is preferable to the previous one to avoid any trouble.

**Building SFML yourself**

Compiling SFML is not as difficult as we might think, and is within the reach of everyone. First of all, we will need to install some dependencies.
Preparing the Environment

Installing dependencies
SFML depends on a few libraries. Before starting to compile it, make sure that you have all the dependencies installed along with their development files. Here is the list of dependencies:

- pthread
- opengl
- xlib
- xrandr
- freetype
- glew
- jpeg
- sndfile
- openal

Linux
On Linux, we will need to install the development versions of each of these libraries. The exact names of the packages depend on each distribution, but here is the command line for Debian:

```
sudo apt-get install libglu1-mesa-dev freeglut3-dev mesa-common-dev libxrandr-dev libfreetype6-dev libglew-dev libjpeg-dev libsndfile1-dev libopenal-dev -y
```

Other operating systems
On Windows and Mac OS X, all the needed dependencies are provided directly with SFML, so you don't have to download or install anything. Compilation will work out of the box.

Compilation of SFML
As mentioned previously, the SFML compilation is really simple. We just need to use CMake, by following these steps:

1. Download the source code at http://sfml-dev.org/download/sfml/2.2/ and extract it.
2. Open CMake and specify the source code directory and the build directory. By convention, the build directory is called build and is at the root level of the source directory.
3. Press the **Configure** button, and select **Code::Blocks** with the right option for your system.

   Under Linux, choose **Unix Makefiles**. It should look like this:

   ![Linux Makefiles screenshot]

   Under Windows, choose **MinGW Makefiles**. It should look like this:

   ![MinGW Makefiles screenshot]
4. And finally, press the **Generate** button. You'll have an output like this:

![CMake output screenshot]

Now the Code::Blocks file is built, and can be found in your build directory. Open it with Code::Blocks and click on the **Build** button. All the binary files will be built and put in the `build/lib` directory. At this point, you have several files with an extension that depend on your system. They are as follows:

- `libsfml-system`
- `libsfml-window`
- `libsfml-graphics`
- `libsfml-audio`
- `libsfml-network`

Each file corresponds to a different SFML module that will be needed to run our future games.

Now it's time to configure our system to be able to find them. All that we need to do is add the `build/lib` directory to our system path.
Linux
To compile in Linux, first open a terminal and run the following command:

```bash
cd /your/path/to/SFML-2.2/build
```

The following command will install the binary files under `/usr/local/lib/` and the headers files in `/usr/local/include/SFML/`:

```bash
sudo make install
```

By default, `/usr/local/` is in your system path, so no more manipulations are required.

Windows
On Windows, you will need to add to your system path, the `/build/lib/` directory, as follows:

1. Go to the Advanced tab in System Properties, and click on the Environment Variables button:

   ![System Properties](image_url)

   - Computer Name
   - Hardware
   - Advanced
   - System Protection
   - Remote
   - Performance
     - Visual effects, processor scheduling, memory usage, and virtual memory
     - Settings...
   - User Profiles
     - Desktop settings related to your sign-in
     - Settings...
   - Startup and Recovery
     - System startup, system failure, and debugging information
     - Settings...
   - Environment Variables...
Preparing the Environment

2. Then, select **Path** in the **System variables** table and click on the **Edit...** button:

   ![Environment Variables](image1)

   2. Now edit the **Variable value** input text, add `;C:\your\path\to\SFML-2.2\build\lib`, and then validate it by clicking on **OK** in all the open windows:

   ![Edit System Variable](image2)

At this point, your system is configured to find the SFML dll modules.
Chapter 1

Code::Blocks and SFML

Now that your system is configured to find the SFML binary files, it's time for us to configure Code::Blocks and finally test whether everything is fine with your fresh installation. To do so, follow these steps:

1. Run Code::Blocks, go to File | New | Project, and then choose Console Application.
2. Click on GO.
3. Choose C++ as the programming language, and follow the instructions until the project is created. A default main.cpp file is now created with a typical Hello world program. Try to build and run it to check whether your compiler is correctly detected.
Preparing the Environment

If everything works correctly, you will have a new window created that has a Hello world! message, as follows:

![Hello world! output]

If you have this output, everything is fine. In any other case, make sure you have followed all the steps for the installations.

Now we will configure Code::Blocks to find the SFML library, and ask it to link with our program at the end of the compilation. To do this, perform the following steps:

1. Go to **Project | Build options** and select your project at the root level (not debug or release).
2. Go to **Search directories**. Here we have to add the path where the compiler and the linker can find the SFML.
3. For the compiler, add your SFML folder.
4. For the linker, add the `build/lib` folder, as follows:

![Project build options](image)

Now we need to ask the linker which libraries our project needs. All our future SFML projects will need the System, Window, and Graphics modules, so we will add them:

1. Go to the **Linker settings** tab.
2. Add `-lsfml-system`, `-lsfml-window` and `-lsfml-graphics` **in the Other linker options** column.
Preparing the Environment

3. Now click on OK.

Good news, all the configurations are now finished. We will eventually need to add a library to the linker in the future (audio, network), but that’s it.

A minimal example
It's now time for us to test the SFML with a very basic example. This application will show us the window as in the following screenshot:
The following code snippet brings about this window:

```cpp
int main(int argc, char* argv[])
{
    sf::RenderWindow window(sf::VideoMode(400, 400), "01_Introduction");
    window.setFramerateLimit(60);

    //create a circle
    sf::CircleShape circle(150);
    circle.setFillColor(sf::Color::Blue);
    circle.setPosition(10, 20);

    //game loop
    while (window.isOpen())
    {
        //manage the events
        sf::Event event;
        while (window.pollEvent(event))
        {
            if ((event.type == sf::Event::Closed)
                or (event.type == sf::Event::KeyPressed and
                    event.key.code == sf::Keyboard::Escape))
                window.close(); //close the window
        }
        window.clear(); //clear the windows to black
        window.draw(circle); //draw the circle
        window.display(); //display the result on screen
    }
    return 0;
}
```

All that this application does is to create a window with a width and height of 400 pixels and its title is `01_Introduction`. Then a blue circle with a radius of 150 pixels is created, and is drawn while the window is open. Finally, the user events are checked on each loop. Here we verify if the close event has been asked (close the button or click `Alt + F4`), or if the user has pressed the `Esc` button on his keyboard. In both case, we close the window, that will result to the program exit.
Summary
In this chapter we covered which technologies we will use and why to use them. We also learned the installation of the C++11 compiler on different environments, we learned about installing CMake and how this will help us build the SFML projects in this book. Then we installed SFML 2.2, and followed on to build a very basic SFML application.

In the next chapter we will gain knowledge on how to structure a game, manage user inputs, and keep trace of our resources.
Where to buy this book
You can buy SFML Blueprints from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.