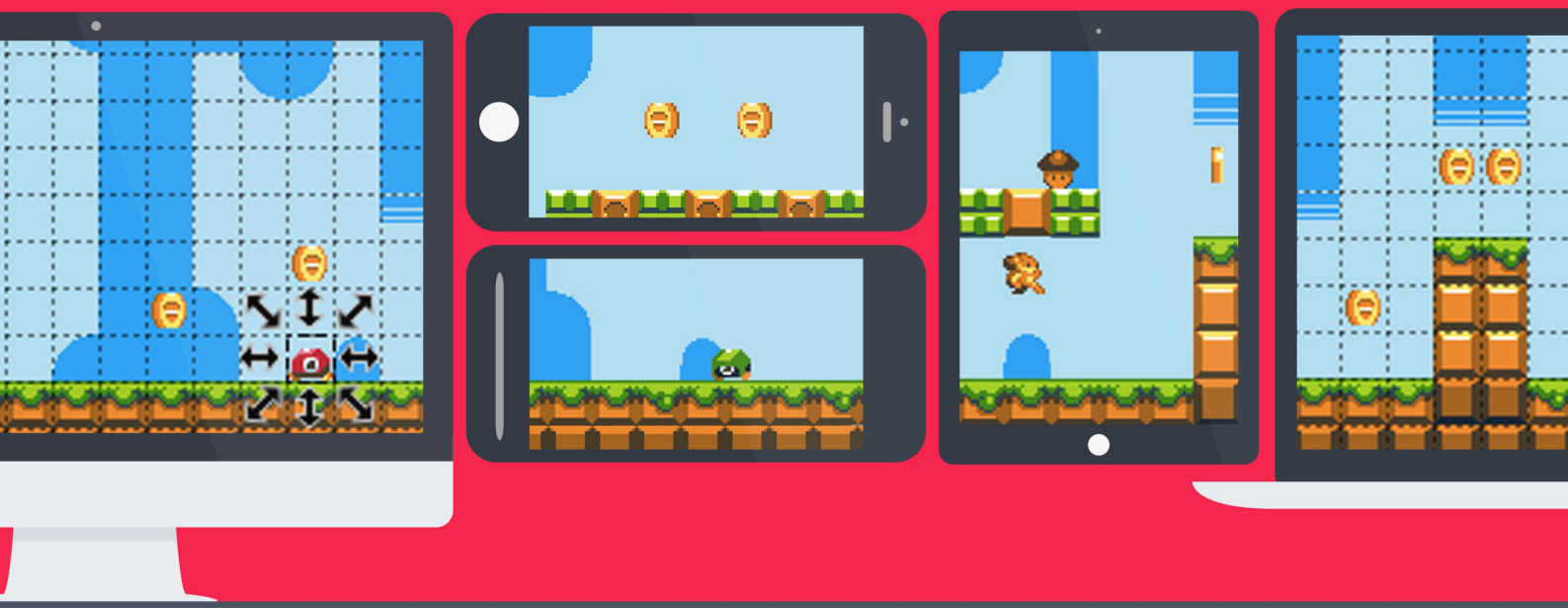
Discover HAXEFLIXEL



LEARN HOW TO DEVELOP & DISTRIBUTE YOUR GAME ON WINDOWS, MAC, LINUX, ANDROID, IOS & WEB ... FOR FREE!

To Cecilia and Beatrice, life's a game worth developing.

Introduction

What is HaxeFlixel

HaxeFlixel is an open source library optimized for 2D game development, which allows you to develop cross-platform applications. It is completely free for personal or commercial use.

That's a very short way to put it, but more than sufficient to start dwelling into game development! However I recommend you to keep on reading to discover the amazing technologies that back HaxeFlixel up:

HaxeFlixel is written using the Haxe Toolkit and the Open Flash Library (OpenFL). Haxe is a cross-platform toolkit which features an intuitive programming language and provides a powerful cross-platform compiler. OpenFL is a software framework powered by the Haxe Toolkit's cross-compiler which allows us to create multi-platform applications from a single code base.

HaxeFlixel, Haxe and OpenFL are open-source technology, completely free for personal or commercial use.

HaxeFlixel is based on flixel, a game development library developed by Adam "Atomic" Saltsman. The original flixel was developed to be used with the ActionScript 3 (AS3) language in Flash, which nowadays is poorly supported and fragmented. HaxeFlixel is meant to fix those limitations by embracing more efficient and well-supported solutions: Haxe and OpenFL.

Why HaxeFlixel

There are a myriad of game development tools out there - some are popular, others less; some are beginner-friendly, others are very technical - Why did I choose to develop games using HaxeFlixel, and went so far as to write a book about it?

Cross-Platform Development and Publishing

The HaxeFlixel framework can be used on Windows, Mac and Linux - everything you need is a text editor - no heavy programs required.

Apart from sparing me several headaches and leaving me the freedom of choosing whatever system I like the most, this allows me to do cool things like syncing my project between my Windows machine at work and my Linux machine at home.

Native Code

Several cross-platform game development tools end up using a virtual machine when deploying the game on multiple platforms. Thanks to Haxe, the code is translated into the platform's native language, making it much faster in comparison.

Free and Open Source

The entire HaxeFlixel source code is available online - same with the Haxe Toolkit and the OpenFL framework.

Even if maintained by a group of core developers, everyone is free to make additions to the code to improve the framework. The technologies are continually evolving and improving. Heck, you can even modify and adapt them for your own needs!

If you'd like to have a look at the HaxeFlixel source code yourself, browse the HaxeFlixel GitHub repository. If it looks like a foreign language to you, fear not - once you finish this book you'll have a greater understanding of its contents and will be able to tinker with the original code just fine.

Suited for me

Maybe most importantly, HaxeFlixel is suited to make the kind of games I want to make.

I want to make 2D games, most of them reminiscent of old-school video games (I'm a huge fan of pixel-art graphics), simple and easy to pick up gameplay.

This doesn't mean that the technology is not suitable for bigger projects. Successful Steam games like Cardinal Quest II, or the upcoming Defender's Quest II are made with HaxeFlixel. The award-winning game Papers, Please has been developed with Haxe and OpenFL.

I also want to develop for mobile platforms, since a huge amount of people use them for games nowadays and I want my games to reach as many people as possbile. However, with a click I can choose to release my game on PC as well. With another click, I can make my game playable in a browser. All using the same code base.

This is exactly what I did with the first game I developed using HaxeFlixel, Polaritron. I have tried several game developing tools in the past, and as of now, only HaxeFlixel ticks all the boxes for me.

There are things, however, for which HaxeFlixel is simply not suited for. If you want to make a 3D game with impressive graphics, you'd better steer towards applications like Unity or Unreal Engine. If you're not comfortable with programming and/or you are 100% sure you want a graphical user interface, you might want to try Game Maker or Construct 2 (although be warned: you will quickly change your mind about the beauty of programming!).

Who is this book for

This book is a great choice for people who have a small to medium amount of experience with coding (or, if you have none, are very eager to get some) and would like to get into game development, or have already developed games in the past and would like to know more about HaxeFlixel and its technologies.

However, this book requires you to be familiar with extremely basic concepts of programming, such as classes, variables and for loops. If those words don't ring any bells, it's better if you get accustomed with those basic building blocks before giving a shot at reading this book.

I recommend the free online Python course at Code Academy - it will start from the absolute basics and should get you up and running in less than a few days. The concepts you will learn in the class will be common to any programming language.

If you are an experienced developer and the previous paragraphs made you crack a smile - You'll be able to skim through most chapters quicker than the average user, and still learn about the powerful features of the HaxeFlixel framework.

How to read this book

This book is meant to be read sequentially. We'll create the project from zero and add elements to it throughout the chapters, as I expand on useful HaxeFlixel concepts.

I will go over each class and chunks of code we write and comment to explain its functionality. This is how formatted code will appear throughout the book:

```
class Player extends FlxSprite
{
    public function new()
    {
        super();
    }

    override public function update(elapsed:Float):Void
    {
        super.update(elapsed);
    }

    override public function destroy():Void
    {
        super.destroy();
    }
}
```

In later chapters, I'll assume you'll be pretty familiar with the core concepts and thus avoid explaining every single line of code, focusing on the more advanced concepts - make sure you fully understand what we did in each section before moving onto the next one.

> Sometimes we'll encounter powerful HaxeFlixel functions which are not used fully within our project and can be expanded on a lot.
>
> In those cases I'll write a few paragraphs explaining those functions in more detail, as it's very likely they'll come in handy for future games you'll make.
>
> Those sections will be wrapped around pink bars, like the ones you see in this one. They're not compulsory for the project we'll be building throughout the book, so feel free to skip them on your first read-through and come back to them later.

At the end of each chapter, I will also include a series of **Extra Reading** links freely available on the web which relate to the techniques explored in those chapters. Sometimes they might sound too technical

for the level you're at currently, so feel free to come back and refer to them throughout the book - try not to skim through them however, since they'll provide a serious amount of knowledge.

Now we are ready to start! In the next chapter we'll see how to install and setup HaxeFlixel and several useful tools.

Extra Reading

- HaxeFlixel: About
- HaxeFlixel: Introduction to Haxe
- HaxeFlixel: Introduction to OpenFL
- HaxeFlixel: Why a Haxe Version
- Lars Doucet: Flash is dead, long live OpenFL

Setup

In this chapter, we'll install the development tools you need to start making games with HaxeFlixel.

Installing Haxe

Head over to the Haxe Website. and download the Haxe installer corresponding to your operating system. Once you open the file, installation is pretty straightforward - you can change the installation path if you wish, I usually leave the default one. At the moment, the latest release of Haxe is not completely compatible with Haxeflixel - download the specific version linked above.

This file will install the Haxe programming language and compiler, and the Neko virtual machine (we'll learn more about this later).

If you are using a Linux machine, I recommend using the install script by Joshua Granick.

After installing, you'll be able to invoke the Haxe compiler from any command prompt / Terminal by typing `haxe`, as well as using the Haxelib tool - a package manager for Haxe packages, by typing `haxelib`.

You can open a command prompt on Windows by opening the start menu, typing `cmd` and selecting the "Command Prompt" application, or a Terminal in Mac by clicking on the Terminal application from the dock. I'll spare you the explanation if you're running a Linux distribution!

As a matter of fact, from now on we'll install the rest of the packages we need using `haxelib` itself.

Installing OpenFL

Open a command prompt / Terminal and run these commands:

```
haxelib install lime 7.8.0
haxelib install openfl
```

The Lime and OpenFL packages will be automatically downloaded and installed. Similarly to Haxe, we are pinning a specific Lime version for compatibility purposes.

Afterwards, run `haxelib run lime setup` to make the `lime` command available.

Installing flixel and flixel-addons

Now, onto the main course! Open a command prompt / Terminal and run `haxelib install flixel`. This will download and install the latest stable version of the flixel framework (as of this book release, 4.7.0).

We'll install `flixel-addons` as well - a collection of useful functions commonly used in HaxeFlixel games,

alongside with some useful templates and demos. Do this by running `lime setup flixel`.

Installing flixel-tools

Next, we'll install `flixel-tools`. This is a command line application which will allow us to quickly create new HaxeFlixel projects from a basic template, and simplifies opening them within our coding environment of choice.

To install them, run `haxelib install flixel-tools` in a command prompt / Terminal. Afterwards, run `haxelib run flixel-tools setup`. You will be prompted for a few options:

- When asked `"Do you want to setup the flixel command alias?"`, type `"y"`. This will setup a few custom commands to quickly create flixel projects.
- When asked to `"Choose your default IDE:"`, I reccomend typing `"3"` for `"[3] Visual Studio Code"` if you don't have a preferred editor. You could also opt for Sublime Text. We'll see how to setup both of the next section. This option will setup a project file when creating a new project, making you able to quickly open it in your editor of choice. Feel free to choose another editor from the list if you are more comfortable with it.
- When asked `Do you want to automatically open the created templates and demos with Visual Studio Code? [y/n]?`, type `y`.

Afterwards, run `flixel` and the flixel-tools should run and display a brief help screen with the list of available commands.

Creating a new project

We can use `flixel-tools` to create a new Haxeflixel project. To do so, open a command prompt / terminal, navigate into a convenient project and type `flixel tpl -n "new_project" -ide vscode`. You can enter any name you want inside the quotation marks. If you are not using Visual Studio Code as your editor of choice, you can omit "-ide vscode".

Try this now. `flixel-tools` will create a empty project for you, ready to run.

Developing HaxeFlixel Projects

Have a look inside the `source` folder of your newly created project - those `.hx` source files contain all the game code, and are common text files. They can be edited with your program of choice, although a proper IDE (integrated development environment) application is recommended, in order to be able to take advantage of auto-completion and other useful tools.

I am personally a fan of Visual Studio Code, and open-sourcea dn cross-platform code editor that can be customized with loads of extensions. It's free to use, and you can download it from its official website.

Another advante of using Visual Studio Code is being able to install the official Lime & Haxe extension which implements syntax highlighting for `.hx` files and code completion. To install it, simply go to the

"Extensions" tab, type "lime" in the search bar and install the Lime extension. This will automatically install the Haxe extension as well.

Do you remember the `-ide vscode` code snippet at the end of the command we used to create the project template> What it did was creating a `.vscode` folder in your project where Visual Studio Code will store some default settings about the project, associating it to the Haxe extensions and enabling useful features like automatic code completion.

For more in-depth information about setting up Visual Studio Code to play nicely with Haxe project, there's an excellent page on the HaxeFlixel website.

Another great alternative is Sublime Text 3, a fast and powerful code editor that can be customized with loads of extensions. You can download it from its official website. The software is free to use, although a message will occasionally pop-up asking you to buy a license to support the developer. However, there are no limitations in the software.

If you opt for using Sublime Text, you can install the official Haxe extension which implements syntax highlighting for `.hx` files and code completion, in a similar way to the Visual Studio Code one. To install it, follow those steps:

- Install the package control tool in Sublime Text
- After the package has been installed, restart Sublime. Then, press `ctrl+shift+P` - this will open the command menu. Type `install` and choose the `Package Control: Install Package` option.
- Start typing `Haxe` and the `Haxe` package should pop up. Select it.
- Wait until the installation is finished and restart Sublime - the extra Haxe features should be enabled.

There are several Sublime packages which can make your coding experience faster and generally easier, alongside with a vast collection of skins and color themes. Feel free to browse the Package Control website and install your favorites. I am personally a fan of Afterglow.

Running the game

We can quickly test the project by building it for the `Neko` virtual machine we mentioned earlier. `neko` builds run on Windows, Mac and Linux and are very quick to compile - making them ideal for quick testing. We'll learn more about `neko` and other compilation targets in later chapters.

You can build your game via the command line or make use of the build systems in your code editor of choice:

- To test the game manually / outside of the code editor open a terminal and navigate to your project directory. Here, execute the command `lime test neko`.
- If you are inside Visual Studio Code and you've got the Haxe extension installed, click the dropdown on the bottom-left of the screen next to the little cog icon which says "Lime", and select `neko` from the Lime Targets Configurations menu. Then, press `ctrl-shift-B` to start the build task.

- If you are inside Sublime Text 3 and you've got the Haxe extension installed, you can press `ctrl+shift+B` to bring up the build target menu, and choose `neko - test`. Afterwards, press `ctrl+shift+P` to bring up the Sublime Text command palette, and choose `Haxe - Run Build`. Note that you don't have to select the target again on subsequent tests - it will build with the last selected target.

If you try to do so with the recently created empty project, you'll see a new window appearing, showing the HaxeFlixel logo, and then a black screen (the project is empty, after all).

Troubleshooting & Compatibility Notes

It's possible that some bleeding edge Haxe / Lime releases might be incompatible with the current release of HaxeFlixel - this is usually fixed in a timely manner - this book will try and make notes of the currently compatibile version, but if unsure, refer to the HaxeFlixel's official "Getting Started" page (linked at the end of this chapter) for indications on the reccomended Haxe and Lime versions.

As of version 4.4.0, HaxeFlixel is fully compatible with OpenFL 8.0. This is great news, as it allow us to finally take advantage of several improvements that were long awaited by the HaxeFlixel community, however the Neko target might run considerably slower on some machines - a result of it being rewritten in non-native language for this latest release. If this happens to you , you might consider using `lime test html5` and test the game in the browser using the HTML5 target, which should compile as fast as Neko and got a considerable boost in performance.

You can also compile to native code for your platform machine - compilation time is longer, but performance is unrivaled. For more information on the build targets you can use, read chapter 18 on cross-platform deployement.

Note that HaxeFlixel 4.4.0 is still fully-backwards-compatible with OpenFL 3.6.1, which is the version this book was originally written for. If you wish, you revert back to OpenFL 3.6.1 and Lime 2.9.1 by executing the following commands: `haxelib set openfl 3.6.1` and `haxelib set lime 2.9.1` and confirming the download when prompted, although is reccomended to use the latest, up-to-date versions.

you can check with version you are currently using by typing `haxelib list` - the library version being used will be surrounded by square brackets.

In the next chapter we'll explore the folder structure of our recently created project and have a look at the basic HaxeFlixel classes.

Extra Reading

- HaxeFlixel: Getting Started
- Visual Studio Code
- HaxeFlixel: Visual Studio Code
- Sublime Text 3
- Sublime Text: The Haxe Package

HaxeFlixel Fundamentals

Before we have a look at our newly created project, let's discuss a few basic HaxeFlixel concepts.

HaxeFlixel is built around common classes and ideas. Some of them are vital to the game structure and are recurring in every HaxeFlixel project, others are quite specific and might never be used based on your gameplay.

Let's have a look at some of the common classes we will end up using the most:

FlxSprite

FlxSprites are the building blocks of HaxeFlixel. Think of this as your game objects - they can move around, be animated, scaled, be created or destroyed - everything that a respectable game object might be expected to do! Most of your game objects will be based upon this class.

FlxGroup

FlxGroups are a way of grouping FlxSprites together and run common operations on all of them at the same time. This functionality is at the base of the collision system in HaxeFlixel. Let's say you have all of your coins in a `FlxGroup`, you can check if the player is touching a coin by doing a collision-check against this group rather than every individual coin.

FlxGroups can be nested as well - for example, you can add the coins group to a bigger group called "collectibles" alongside other groups.

FlxState

Think of FlxStates as the "scenes" of your game - can be a menu, or a level. Under the bonnet, a `FlxState` is simply a big group for all of your game objects in the state. For those coming from a GameMaker background, they are the equivalent of rooms.

FlxG

`FlxG` is a global helper class that can provide access of several key aspects of your game, such as managing collisions and switching between states, or provide access to useful global variables such as the size of your game area.

The game sources

Now let's see what a bare-bones Haxeflixel project looks like. We can finally go back to the empty project we created using `flixel-tools` - Let's go inside its folder and have a look at its structure:

assets

This folder will contains any resource that your game might require - be it a sprite sheet for a monster, a shooting sound effect or a file containing a game level. It already comes neatly organized in `data`, `images`, `music` and `sound` sub-folders.

sources

This folder hosts the `.hx` files, which are the Haxe source code files which will be compiled into your final game. You can see that `flixel-tools` automatically generated most of the basic files we need to get our game started:

Main.hx

This file is the starting point of your game, and contains the method which gets the HaxeFlixel core engine up and running - nothing you should really worry about changing.

In this function call, however, you can change the arguments in order to alter your game's appearance:

```
addChild(new FlxGame(640, 480, MenuState));
```

- The first argument is width of the game in pixels
- The second argument height of the game in pixels
- The third argument is the `FlxState` the game will start in.

PlayState.hx & MenuState.hx

Those two `FlxState` are both empty, so there's not much difference between them at the moment.

AssetPaths.hx

This is a small utility class that auto-generates the references to asset paths - let's say you've got a file called `spaceship.png` inside your assets folder. As you start typing the name of that file, your IDE will suggest you the auto-completion (provided an Haxe language plugin is installed) with `AssetPaths.spaceship__png`. This will reference the file - no matter if it's placed inside `images`, `data`, or even `music` - the `AssetPaths` class will figure it out for you.

Project.xml

This file contains the settings defining aspects of your application such as:

- window size and aspect ratio settings on different targets (desktop, mobile, web, etc)
- where the source files (`.hx`) and assets files are located
- which Haxe libraries are we using in this project (why, the almighty flixel, of course!)
- what will the icon be for our final application

We will go through a fair amount of changes here to make the game look exactly like we want, especially when deploying on mobile devices - so keep this file in mind!

Although there's something about it worth mentioning now, while a few concepts are still fresh in your

mind. See the line?

```
<!--These window settings apply to all targets-->
<window width="640" height="360" fps="60"
 background="#000000" hardware="true" vsync="true" />
```

You might wonder how those differs from the argument of the `addChild()` function we saw earlier in `Main.hx`. It's simple: the values here define the **Window Size**, while the ones in `Main.hx` define the **Game Size**. For example, if your game size is 320x240 and the window size is 640x480, the game will be displayed in a 640x480 window at **2X scale**. If the game size was 640x480, it would be displayed at it original scale as the game size is the same as the window size.

That's it about the basics! In future chapters we'll expand on many of this concepts to customize our game's appearance.

In the next chapter we'll start editing our empty project (or you can create a new one, if you wish to start anew) and lay the foundation for our game.

Laying the Groundwork

In this chapter we'll start to edit our empty project. Open the `Main.hx` file and change some of the arguments we mentioned earlier:

- Change the game width and height to 320 and 180, respectively.
- Set the initial state to `PlayState`:

```
addChild(new FlxGame(320, 180, PlayState));
```

Afterwards, go to `Project.xml` and change the `window width` to 640 and `window height` to 360. This combination will give us a nice 2X scaled game in a 16:9 ratio.

You can change the game's name (which will be displayed on the window title) by changing the `title` attribute in the `app` tag at the top of the file. Feel free to change `company` as well:

```
<app title="haxeflixel-game" file="haxeflixel-game"
  main="Main" version="0.0.1"
  company="YourAwesomeGameCompany" />
```

The PlayState

As you saw earlier, the template tool created a bare-bones `PlayState.hx`. This will be our main game room. Let's open the file:

```
package;

import flixel.FlxG;
import flixel.FlxSprite;
import flixel.FlxState;
import flixel.text.FlxText;
import flixel.ui.FlxButton;
import flixel.math.FlxMath;

class PlayState extends FlxState
{
 override public function create():Void
 {
  super.create();
 }

 override public function update(elapsed:Float):Void
 {
  super.update(elapsed);
 }
}
```

This is the basic setup for a blank `FlxState`. We have 2 main functions:

- The method `create` is called when the state is created, here we will setup and initialize our game objects.
- The method `update` will called every game frame. Here we'll put most of our game logic.

Notice how those methods are marked by `override`: it means that the `FlxState` class contains all of those functions.

At the top of the class you'll see those imports statements:

```
import flixel.FlxG;
import flixel.FlxSprite;
import flixel.FlxState;
import flixel.text.FlxText;
import flixel.ui.FlxButton;
import flixel.math.FlxMath;
```

Every time we want to use a new class, we'll want to import the correct module to have access to all the types of that module and their functionality.

Basic map and player

We can see how this template project has already imported the most basic HaxeFlixel classes, however we'll add some more:

```
import flixel.tile.FlxTilemap;
import flixel.FlxObject;
```

`FlxTilemap` will allow us to build a level, while `FlxObject` contains some collision logic.

Forgetting to import classes is one of the recurrent beginner mistakes! If the compiler returns the error `Type not found : FlxTilemap` it means you are likely to be using functions of a class you haven't imported yet (in this example, `FlxTilemap`).

Now let's add some variables right after the class definition:

```
class PlayState extends FlxState
{
 var map:FlxTilemap;
 var player:FlxSprite;
 var mapData:Array<Int> = [
   1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,
   1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,
   1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,
   1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,1,
   1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,1,
   1,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,
   1,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,
   1,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,
   1,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,
   1,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
   1,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
   1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
  ];
...
```

We have the player `FlxSprite` and the map `FlxTileMap` for our level. We also add a `Array<Int>` which will describe what our level looks like: 0 is empty, and 1 is a solid block. If you squint your eyes, you can already make out the shape of the level!

Before moving on, we need to acquire the graphics we'll use to draw the level. For ease of use, I have prepared a set of assets to use and I'll reference them throughout the book.

You will find all the assets I refer to throughout the book inside the `book-asset` .zip archive you received when purchasing the book.

Whenever we need a new asset, you can copy it over from `book-assets` and I'll tell you in which location of your project it needs to be saved (usually a sub-folder inside `assets`).

Alternatively, when browsing the source code for a specific chapter in `book-sourcecode`, you can find all the assets used in that chapter inside its `assets` folder.

Copy the `tiles.png` file from `book-assets/images` and place it in the `assets/images` folder in your project. This file contains the graphic information to build a level from a tile map array.

Look back at the `mapData` array: when using the `tiles.png` file to draw this map, the first tile from the upper-left will be drawn on the slots with a value of `0`, the second one on the ones with a value of `1`, and so on.

Move to the `create` function:

```
override public function create():Void
{
 map = new FlxTilemap();
 map.loadMapFromArray(mapData, 20, 12, AssetPaths.tiles__png, 16, 16);
 add(map);

 player = new FlxSprite(64, 0);
 player.makeGraphic(16, 16, FlxColor.RED);
 player.acceleration.y = 420;
 add(player);

 super.create();
}
```

We initialize our `FlxTilemap`, and then call the `loadMapFromArray()` method. It takes a number of arguments:

- the `mapData` array we defined earlier
- the width and height of the level (20 tiles horizontally, 16 vertically)
- the graphic asset to use for the tiles (the file `tiles.png` in the `assets/graphics` folder)
- the width and height of the tile (16 by 16).

Finally, we add this map to the current state using the `add()` method.

`add()` is a `FlxGroup` function which tells an object it belongs to that state. When a object is added to a state, it will become active and visible when that state is played.

Calling `add` in a `FlxState` will add the specified object to the `FlxState` we are calling the function from, in this case `PlayState`.

Afterwards, we initialize the player. The `new` function (the constructor) will set the `x` and `y` coordinates to create our player at (in this case, it will be the fourth tile from the upper left of the level as HaxeFlixel coordinates are counted going left to right and up to down.)

The `makeGraphic()` function will create a graphic of a specified `width`, `height` and color. `FlxColor` contains several presets for existing color - now we're using `RED`. In order to do so, we need to add the import statement for the `FlxColor` utility module:

```
import flixel.util.FlxColor;
```

Finally, we give the player an acceleration value of `420` in the Y direction, which will act like our gravity, and we add the player to the state as well.

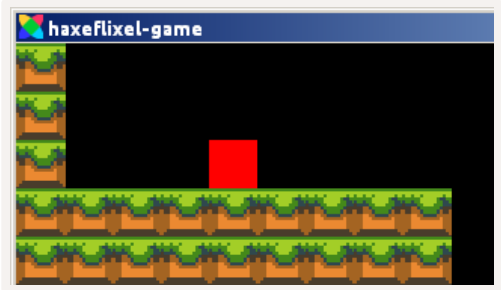
If we run our game now, we can see both the level and the player on the screen! Unfortunately, the player accelerates downwards and quickly falls through the floor, as we have no collisions set up yet! Well, that was a short play.

Collisions

Let's fix this. Go to the `update()` function:

```
override public function update(elapsed:Float):Void
{
 super.update(elapsed);

 FlxG.collide(map, player);
}
```

HaxeFlixel provides a very straightforward way of implementing collisions: you simply call the `FlxG.collide()` method and pass two `FlxObjects`. The two objects will be separated on collision, and you can pass a specific function to execute when the object collide as well - we'll see more of that later.



Inputs

If we run the game again, the player will now collide with the floor. But it's a bit of an anticlimactic ending, as we have no way to control him. Let's add some inputs! Make a new function called `movePlayer()`:

```
private function movePlayer():Void
{
    player.velocity.x = 0;

 if (FlxG.keys.pressed.LEFT)
  player.velocity.x -= 80;

 if (FlxG.keys.pressed.RIGHT)
  player.velocity.x += 80;

 if (FlxG.keys.justPressed.C && player.isTouching(FlxObject.FLOOR))
  player.velocity.y = -200;

}
```

`FlxG.keys` contains a catalog of all the keys on your keyboard and their current status. We check if `LEFT` or `RIGHT` are pressed, and modify the velocity in the x direction accordingly.

To make the player jump, we set its y velocity to a negative value (going up, remember?) whenever the `C` key is pressed, but careful - we want to do this only if the player is touching the ground, or else it will end up jumping repeatedly in mid-air.

We do this by calling the `isTouching()` method of `FlxSprite` (in this case, the `player` object), which takes a direction to check against and returns `true` if the object is currently colliding with something on that side.

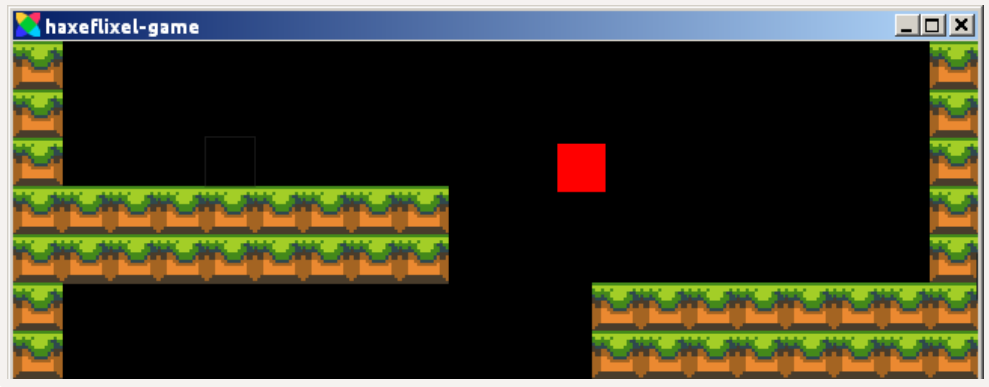
We check against `FlxObject.FLOOR`, which represents the lower side of an object. Other useful flags are `FlxObject.WALL`, which checks for left and right direction, and `FlxObject.CEILING`, which check for the upper direction.

We will use all them throughout the book. You can also specify individual collision directions as flags, such as `FlxObject.LEFT`, `FlxObject.UP`, and so on.

We will call this `movePlayer()` method in `update`, as these calculations need to happen every frame!

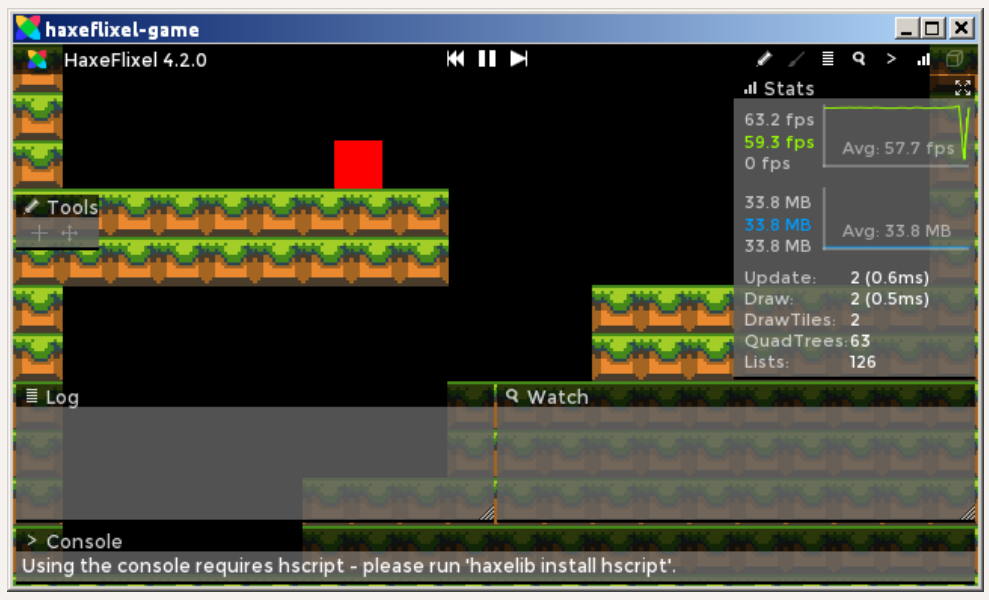
```
override public function update(elapsed:Float):Void
{
 super.update(elapsed);

 FlxG.collide(map, player);
 movePlayer();
}
```

Let's run the game again, and this is already looking sweet! Our player moves and jumps around in this little level, and everything feels smooth. Off to a great start!



The Debugger

Before we go on, let me tell you about a very useful HaxeFlixel feature - the debugger. Press `F2` and a status screen will appear on top of your game, displaying useful information such as the total amount of game objects and the frames per second your game is running at.



The HaxeFlixel debugger!

If the debugger is not showing when pressing `F2`, try the following:

- Run the game by typing `lime test neko -debug` (note the added `-debug` flag)
- OR Open `Project.xml` and comment out / delete the following line:

```
<haxedef name="FLX_NO_DEBUG" unless="debug" />
```

The debugger will display non-fatal errors as well. You can also manually print values to the debugger's log by using the `FlxG.log.add()` function.

Another way to print out message from the application is the `trace()` function, the Haxe equivalent of `print()` - we'll use it to strategically print debug messages later in the book.

Another new powerful feature of the debugger as of HaxeFlixel 4.2.0 is the ability to select objects using the pointer tool (The "+" icon on the "Tools" toolbar) while the game is running. You can then move those objects in real time by holding `SHIFT` while clicking and dragging them; or deleting them by pressing `DELETE`.

In the next section we'll see how to add a more complex level to our state!

Extra Reading

- HaxeFlixel Handbook: FlxState
- HaxeFlixel Handbook: Keyboard
- HaxeFlixel Handbook: Debugger

The Player

Every platformer game needs a great protagonist. In this chapter we'll give our `Player` object some tender loving care.

`Player` class & extending FlxSprite

Let's give our player a separate class. Create a new file called `Player.hx` inside the `source` folder and put this code inside:

```
class Player extends FlxSprite
{
    public function new()
    {
        super();
    }

    override public function update(elapsed:Float):Void
    {
        super.update(elapsed);
    }
}
```

As you can see, the base structure of a `FlxSprite` is very similar to a `FlxState`'s: instead of `create()` we have a `new()` function which will be executed when the object is initialized, and the `update()` function which runs automatically every frame.

Right after the class definition, let's write some variables which will be useful when coding the movement of the player:

```
class Player extends FlxSprite
{
 private static inline var ACCELERATION:Int = 320;
 private static inline var DRAG:Int = 320;
 private static inline var GRAVITY:Int = 600;
 private static inline var JUMP_FORCE:Int = -280;
 private static inline var WALK_SPEED:Int = 80;
 private static inline var RUN_SPEED:Int = 140;
 private static inline var FALLING_SPEED:Int = 300;

 public var direction:Int = 1;
 ...
```

The name should be self-descriptive, we'll see better how these values will fit in our code very shortly. Declaring those variables here in the class scope will make it easier to change them later in case we want to fine-tune our movement speed.

As you might have noticed, we define those value as `static inline` variables, which means they won't be able to get changed once they have been defined - useful when dealing with parameters that you

want to keep constant throughout the game.

Next, copy the `player.png` file from `book-assets/images` and place it in the `assets/images` folder in your project. This file contains the graphic of the player character.

Let's proceed and write our `new()` function:

```
public function new(x:Float, x:Float)
{
 super(x, y);
 loadGraphic(AssetPaths.player__png, true, 16, 16);

 animation.add("idle", [0]);
 animation.add("walk", [1, 2, 3, 2], 12);
 animation.add("skid", [4]);
 animation.add("jump", [5]);
 animation.add("fall", [5]);
 animation.add("dead", [12]);

 setSize(8, 12);
 offset.set(4, 4);

 drag.x = DRAG;
 acceleration.y = GRAVITY;
 maxVelocity.set(WALK_SPEED, FALLING_SPEED);
}
```

First of all, you can see we change the default constructor from `new()` to `new(x:Float, y:Float)`. This will allow us to pass the X and Y coordinates where the player will be created at when creating the `Player` object.

Then, we call the `loadGraphic()` function. It works in a similar way as the `makeGraphic()` we used in our previous example, but it will load the graphic from a sprite sheet image file we pass as the first argument. The `true` flag indicates that this sprite is going to be an animated image, and `16` by `16` is the size of a single frame of the animation.

We use `animation.add()` to define a new animation for the object, passing a name for the animation, an array of the frame indexes to use for the animation, and a number indicating how fast the animation will play (if the animation is made of a single still frame, you can omit this).

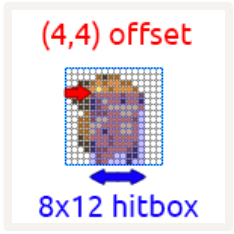
The frame indexes in a sprite sheet are counted from zero, going from left to right and top to bottom. Looking at the image, it's quite simple to figure out which frames to use for which animations.

We use `setSize()` to adjust the size of the sprite's hitbox - the area of the sprite which will collide with other objects, such as the map and the enemies.

Since there is a bit of empty space around our sprite, we don't want to use the use the full sprite as a

hitbox or else it will detect collisions even when one of those empty pixels touches an object.

Instead, we'll reduce it's hitbox to a `8x12` rectangle, ideally it's bottom-center. Problem is, when using `setSize()` the bounding box is resized from the upper-left corner. To fix this, we call `offset.set()` to adjust the position of the hitbox - in this case, we shift it both horizontally and vertically by `4` pixels.



You can use the debugger to display the hitboxes of every `FlxObject` added to the state by pressing the cube button in its right upper corner.

Next, we set the `acceleration.y` to our gravity value. The `drag` property works like friction, indicates how slowly an object will decelerate once it's stopped. Adding this will make our walking motion a bit smoother. Finally, since we don't want our player to fall too fast, we assign the values for its maximum velocity with `maxVelocity.set`.

Movement

Now we start writing the movement logic. Rather than putting everything in the `update` function, let's be clean and create a new function called `move()`:

```
private function move()
{
```

Let's start by taking care of the horizontal motion:

```
 acceleration.x = 0;

 if (FlxG.keys.pressed.LEFT)
 {
  flipX = true;
  direction = -1;
  acceleration.x -= ACCELERATION;

 }
 else if (FlxG.keys.pressed.RIGHT)
 {
  flipX = false;
  direction = 1;
  acceleration.x += ACCELERATION;
 }
```

First, we reset the player acceleration to 0. Then based on which key is pressed, we change the `acceleration` value on the appropriate direction using the amount defined in the `static` variable

`ACCELERATION`.

This way, if we want to change the acceleration amount, we can just change that static variable and the new value will be used whenever we reference that variable, instead of having to change it manually in every scenario.

The `flipX` value will flip the object graphic if set to `true`. We change the `direction` variable as well.

```
 if (velocity.y == 0)
 {
  if (FlxG.keys.justPressed.C && isTouching(FlxObject.FLOOR))
   velocity.y = JUMP_FORCE;

  if (FlxG.keys.pressed.X)
   maxVelocity.x = RUN_SPEED;
  else
   maxVelocity.x = WALK_SPEED;
 }
} //end of move() function
```

Now let's take care of the jump mechanic: in the same way as in the previous chapter, we check whether the `C` key is was just pressed and if the player is colliding in the bottom direction - if both conditions are satisfied, we increase `direction.y` to our `JUMP_FORCE` value (which, if you remember, is negative, so the player will go up).

Let's implement a run mechanic as well - we check if the `X` button is currently being pressed on the keyboard, and in that case we increase the `maxVelocity.x` value to our higher `RUN_SPEED` value. If not, we revert to the default `WALK_SPEED` value.

However, what would happen if we were to hold `X` in mid-air? The player would suddenly move faster in the middle of a jump, which doesn't exactly feel right. To fix this, we wrap the jump and run code inside this condition: `if (velocity.y == 0)`:

This will allow the player to switch to the running speed only when he's not in the middle of a jump.

Let's run this `move` function in our `update()` (don't forget to keep the `super()` call):

```
override public function update(elapsed:Float):Void
{
 move();
 super.update(elapsed);
}
```

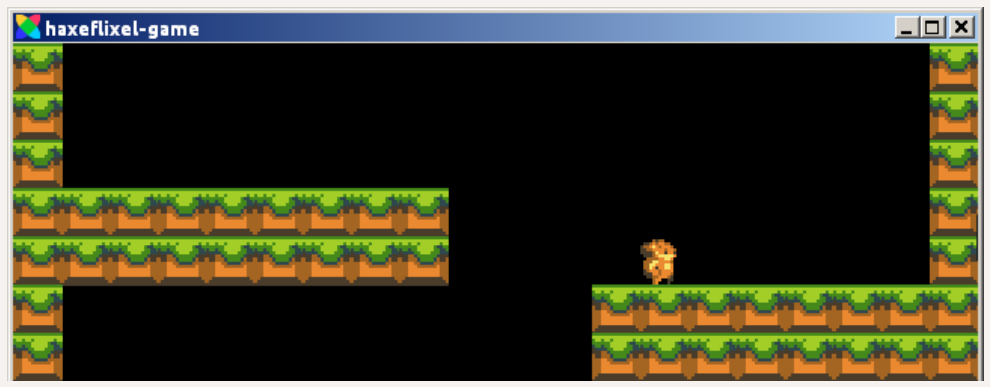
Out player class starts to make more and more sense, but in order to test it we need to get rid of the "placeholder" red square player we coded in the previous chapter and use this new class instead.

In the `create()` function of `PlayState.hx`, replace the old player definition with:

```
 player = new Player(64, 16);
 add(player);
```

Where `64` and `16` are the starting coordinates of the player. Moreover, get rid of the `movePlayer()` function in `update()` since we are now managing the movement from the `Player` class.

Try and run the game. Our new player will smoothly move around and jump on request!



The player, however, will always jump at the same height no matter how long we press the jump button. In platformer games, I have always liked how responsive it felt when the shorter you pressed the jump button, the shorter the player jump would be - it allowed for some really tight controls!

Let's implement this right now. Go back to our `move()` function and add this section of code right before the end:

```
  ...

  if ((velocity.y < 0) && (FlxG.keys.justReleased.C))
   velocity.y = velocity.y * 0.5;

} //end of move function
```

What's happening here? First, we check if the player is in the ascending portion of a jump (its `velocity.y` will be a negative value). Then we check if the `C` key has just been released with the `FlxG.keys.justReleased` method. If so, we take the player's `velocity.y` and reduce it by half. This will shorten its jump and graciously stop it. Fantastic!

You can see how the flags for input interactions differ between them:

- `justPressed` is true for a single frame, as soon as the button is pressed
- `pressed` keeps being true as long as the button is held down
- `justReleased` is true for a single frame, as soon as the button is let go

Animations

Remember all the animations we defined earlier? It's now time to put them into action. Create another function called `animate()` and write this inside:

```
private function animate()
{
 if ((velocity.y <= 0) && (!isTouching(FlxObject.FLOOR)))
  animation.play("jump");
 else if (velocity.y > 0)
  animation.play("fall");
 else if (velocity.x == 0)
  animation.play("idle");
 else
 {
  if (FlxMath.signOf(velocity.x) != FlxMath.signOf(direction))
   animation.play("skid");
  else
   animation.play("walk");
 }
}
```

This will check what state the motion the player is in and play the right animation using `animation.play` by supplying the name of an animation previously defined. Let's see what's happening:

- If the `velocity.y` is negative (or if it's equal to zero but the player is NOT touching the ground - means it will be at the apex of the jump), we'll play the `jump` animation.
- If the `velocity.y` is positive instead, meaning the character is traveling downwards, we play the `fall` animation.
- If the player if not moving in the vertical axis, it means it's on the floor. If its `velocity.x` is 0 as well, then it's `still`.

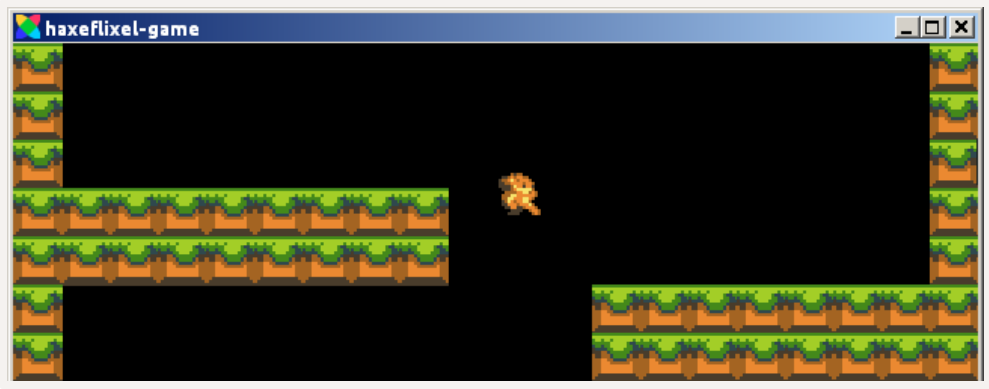
Finally, if none of the above is happening, it means its `velocity.x` is different than 0 and this must be walking or running, right? Yes, but let's do something smart first.

We check the sign of its `velocity.x` using the `signOf` function in the `FlxMath` class, which contains several useful function for mathematical calculations. `signOf` will return `-1` if the value is negative, and `1` if positive. We check this against the current direction of the player: if it's different, it means that player has just changed direction so we play the `skid` animation. If not, the player is indeed walking - we play the `walk` animation.

This should cover up all of our cases. Let's run the `animate()` function in our `update()` as well:

```
override public function update(elapsed:Float)
{
 move();
 animate();

 super.update(elapsed);
}
```

…and run the game. The way our character moves feels amazing! And if you don't like how fast or slow certain motion feels, you can go ahead and adjust the `static` variables we declared at the top of the class to your liking.



In the next chapter, we'll create a more complex map for our player to move in and load it inside our game.

Extra Reading

- HaxeFlixel Handbook: FlxSprite

Building & Loading a Level

In our previous chapter we defined the level map manually, writing each tile ID ourselves into an array. This would be a very cumbersome way of creating more complex levels!

To do so, we will use a free, dedicated program called **Tiled**.

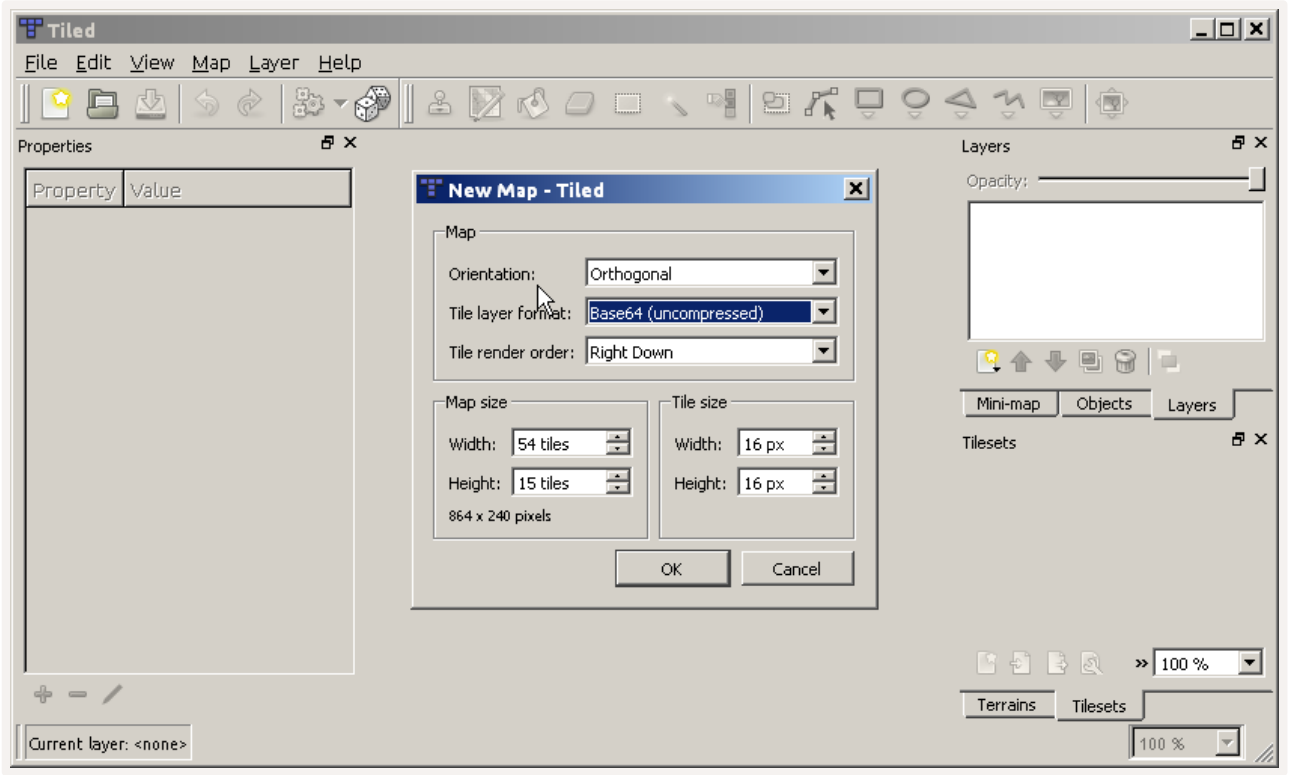
Tiled Editor

You can download Tiled for free from its website. It's a cross-platform tool - it will run on Windows, Mac and Linux.

Once opened, let's create a new map by going to File -> New. Leave the default options on, but choose `Base64 (uncompressed)` as `Tile Layer Format`. Set the Tile size to `16 px` for both Width and Height, and pick a map size. This is defined in tiles, and Tiled will automatically tell you the resulting map size under the spinners. Consider that one game screen will be 360x180 pixels big.

We are now ready to create our map! Tiled works with two separate kind of map layers:

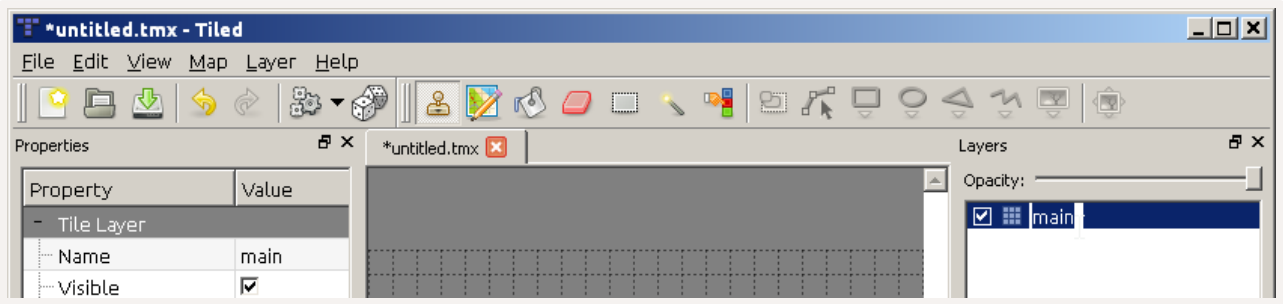
- Tile layers, usually used for the level's terrain and background
- Object layers, usually used to determine the position and features of particular game objects, such as enemy and bonuses.



You can have as many layers of each kind as you want. In this example, we will start by creating two tile layers, one for the terrain which will actively collide with the player, and one for the background

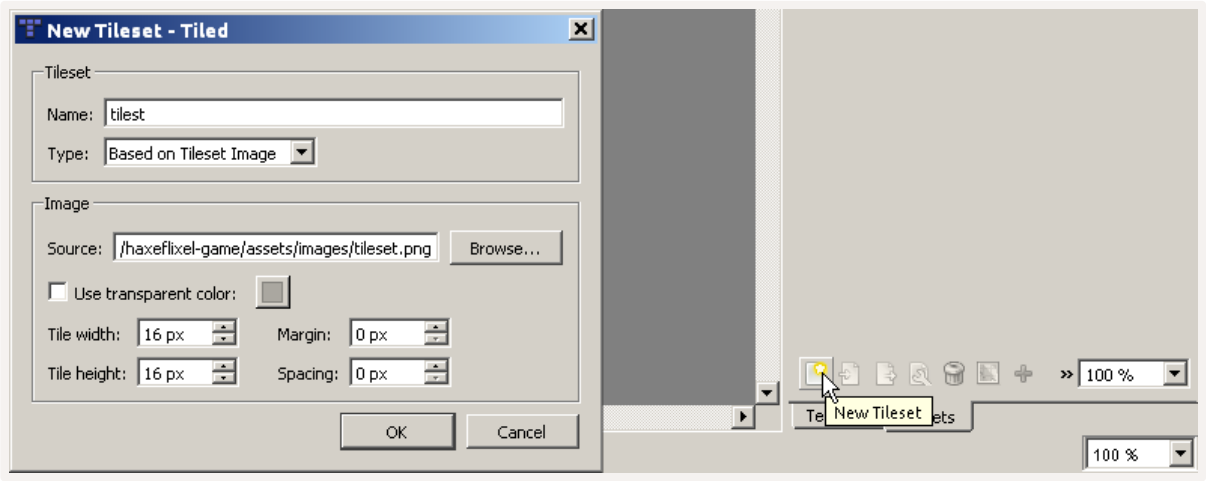
graphics.

Tiled will create a new map with an empty tile layer: you can see it in the “Layers” tab in the upper right. Double click on it and rename it to “Main”.

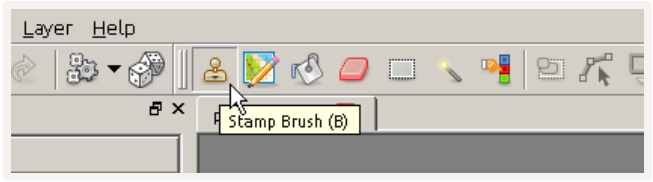


Now we need a tileset (an image containing all of our different map graphics) to draw on the map with!

In Tiled, click the `New Tileset...` button in the `Tileset` tab on the bottom right. Give the new tileset a name of your choice, set the tile width and height to `16px`, and as pick the `tileset.png` image you previously copied as the image source.



Our tileset will appear in the tab. You’ll now be able to select a tile from it and paint the map with it (if not, make sure that the `Stamp Brush` tool is selected).

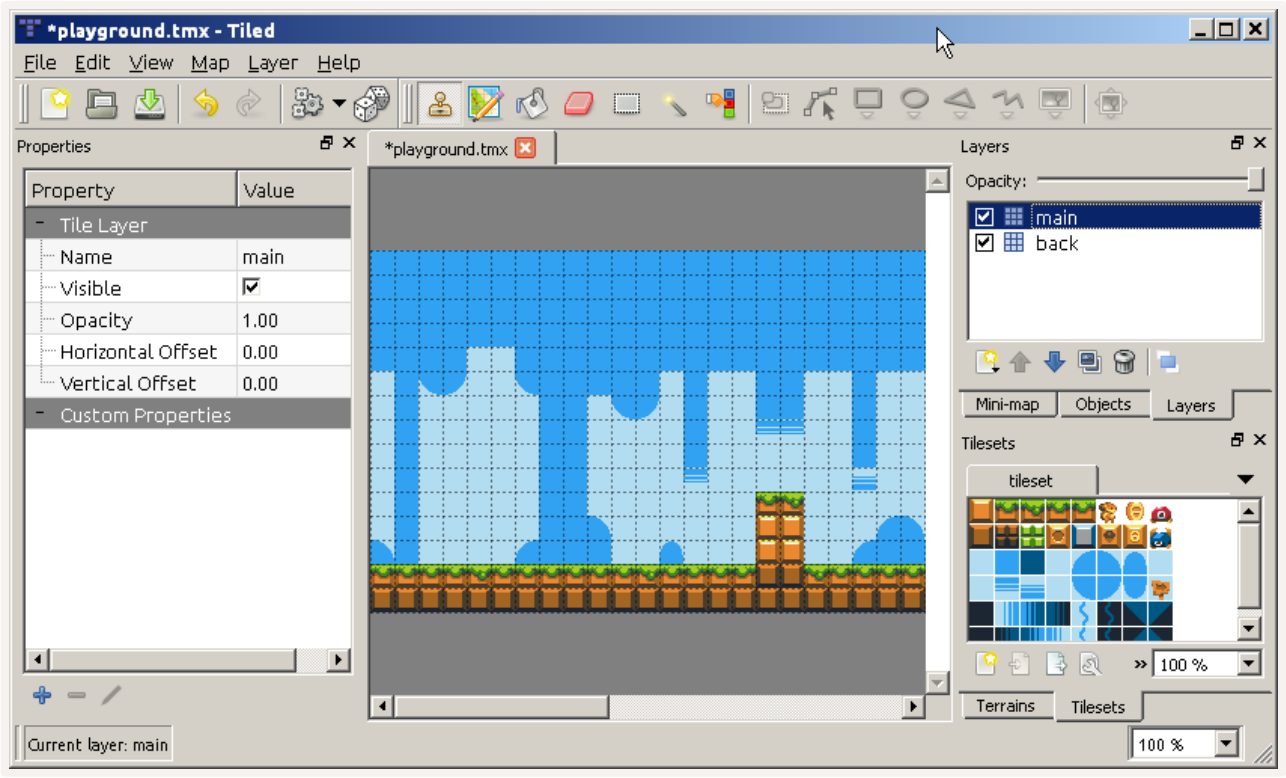


Don’t worry about making a crazy level straight away, we just want a simple environment to play with at the moment - we will come back and add more complex levels as our engine gets more robust.

For reference, keep in mind that with the settings we coded so far, the player can’t jump higher than **4 tiles**, so try not to make any insurmountable obstacles (you can always go back to `Player.hx` to change its `JUMP_FORCE` and `GRAVITY` to any setting you’d like)!

Once you’re done with the main layer, create a new Tile Layer, call it “Back” and paint on it with the blue

tiles to make the sky (and maybe some clouds and trees). Is the sky appearing in front of your main layer? In that case you might want to move the "Back" layer under the "Main" one: you can do this using the arrow buttons called "Raise Layer" and "Lower Layer". Layers are rendered in order, from the bottom to top.



Once you're happy with your map, save it in `assets\data` as `playground.tmx`. If you'd rather use the little level I created for the book, feel free to copy it from the source code for this chapter. We will now go back to HaxeFlixel and write the code to import this newly created map!

`LevelLoader` utility class

Let's create a new file `LevelLoader.hx` - we will write our map loading implementation here, to make things cleaner.

Before that, however, we have to make sure the `flixel-addons` library, which contains all of the Tiled-related functions, is added to the project. We can do this by opening the `Project.xml`, and making sure that the line where `flixel-addons` is added is uncommented.

You'll find it right under the `flixel` library, in the `LIBRARIES` section:

```
<!-- _____________________________ Libraries _____________________________ -->

 <haxelib name="flixel"/>

 <!--In case you want to use the addons package-->
 <haxelib name="flixel-addons"/>
```

Now we are ready to write the `LevelLoader.hx`. Let's add the imports we need for this class, including the Tiled addon:

```
import flixel.FlxG;
import flixel.FlxObject;
import flixel.FlxState;
import flixel.group.FlxGroup;
import flixel.math.FlxPoint;

import flixel.addons.editors.tiled.TiledLayer;
import flixel.addons.editors.tiled.TiledTileLayer;
import flixel.addons.editors.tiled.TiledMap;
import flixel.addons.editors.tiled.TiledObject;
import flixel.addons.editors.tiled.TiledObjectLayer;
import flixel.tile.FlxTilemap;
```

Now let's create a public static method: this will make it able to be called anywhere in the project without the need of building an instance of the `LevelLoader` class.

```
public static function loadLevel(state:PlayState, level:String)
{
```

This function will work by passing to it the `PlayState` we want to load the map in, and the name of the level as a string. Inside we will write:

```
var tiledMap = new TiledMap("assets/data/" + level + ".tmx");

var mainLayer:TiledTileLayer = cast tiledMap.getLayer("main");

state.map = new FlxTilemap();
state.map.loadMapFromArray(mainLayer.tileArray,
         tiledMap.width,
         tiledMap.height,
         AssetPaths.tiles__png,
         16, 16, 1);
state.add(state.map);
```

First we defined a new `TiledMap` object, loading the name of the map we pass to the function.

At that point, we get the "Main" TileLayer we've been drawing in Tiled. the `TiledMap.getLayer()` returns a generic layer, and the program is not able to know if we are loading a Tile Layer or an Object Layer. We use the `cast` expression to specify that we are indeed loading a TiledTileLayer.

Then we initialize the `map` FlxTilemap object in our state, and call the `loadMapFromArray()` function. It's quite lengthy, so let's have a look at the parameters we are passing to it:

- The level information as an array of data (just like the map we defined manually in the previous example, but with way more values than 0s and 1s - Tiled will take care of converting the graphical tiles into ID for us!),
- The width and height of the map (which we can extract dynamically from the TiledMap with

`tiledMap.width` and `tiledMap.height`)

- The image to use to draw the tileset - we'll pick the same one we used as a tileSet source graphic in Tiled to make sure they match!
- The width and height of a tile (both 16)
- The last argument (1) is the "Starting Index", a value used to offset the ID of the tile to draw.

Finally, we add the `map` to the state. Now the Main layer had been loaded into the game! Let's do the same for the `back` layer:

```
var backLayer:TiledTileLayer = cast tiledMap.getLayer("back");

var backMap = new FlxTilemap();
backMap.loadMapFromArray(backLayer.tileArray,
       tiledMap.width,
       tiledMap.height,
       AssetPaths.tiles__png,
       16, 16, 1);
backMap.solid = false;
```

We'll add it to a separate `FlxTilemap` that we are creating on the fly, since we don't really care about collisions with the background layer - we can just load it and forget about it!

Make sure to set the `solid` flag on the `back` layer map to `false` so that our player won't collide with it.

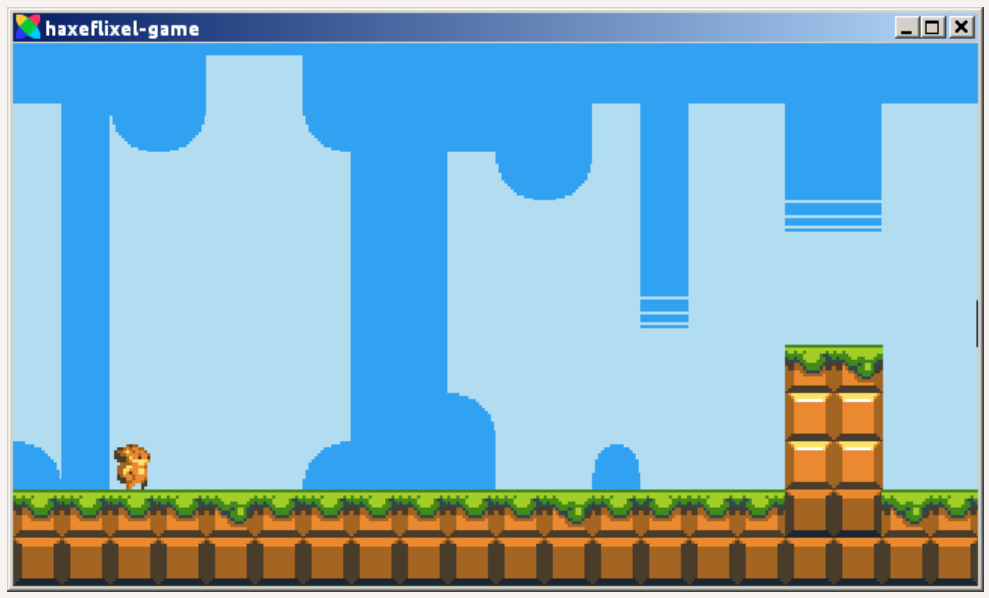
Then, we'll add both map to the state:

```
 state.add(backMap);
 state.add(state.map);

} //end of loadLevel() function
```

Don't forget to add `backMap` **before** `map` or else the background will be drawn over the foreground! Newly added object are always drawn on top of old ones.

Finally, in the `create()` function of `PlayState.hx`, replace the map loading logic from the previous example with:

```
override public function create():Void
{
 // Old stuff we can get rid of!
 // map = new FlxTilemap();
 // map.loadMapFromArray(mapData, 20, 12,
 //         AssetPaths.tiles__png,
 //         16, 16);
 // add(map);

 player = new Player(FlxG.width * 0.5, 10);

 LevelLoader.loadLevel(this, "playground");

 add(player);

 super.create();
}
```



When the game starts, the level is loaded

FlxCamera

If you run the game now, you'll be welcomed into the level we created! You might notice, however, that it is not playable yet: the camera of the game will start from the top-left of the screen, and if you followed my suggestions for the level dimensions we won't be able to see the floor when the player lands on it.

We need to make the game's camera follow the player so we'll never miss any of the action. To fix this, simply add those lines in the `create()` function of `PlayState.hx` after adding `player`:

```
...
add(player);

FlxG.camera.follow(player, FlxCameraFollowStyle.PLATFORMER);
FlxG.camera.setScrollBoundsRect(0, 0, map.width, map.height, true);
```

The first line will tell the game's camera (which by default is accessible via `FlxG.camera`) to follow the `player` object. `FlxCameraFollowStyle.PLATFORMER` will define the camera motion to be dynamic - in this case it will move when the object gets closer to the screen edges.

> There are several `FlxCameraFollowStyle` values you can use to customize your game camera:

> - `LOCKON`: Camera has no deadzone, just tracks the focus object directly.
> - `PLATFORMER`: Camera's deadzone is narrow but tall.
> - `TOPDOWN`: Camera's deadzone is a medium-size square around the focus object.
> - `TOPDOWN_TIGHT`: Camera's deadzone is a small square around the focus object.
> - `SCREEN_BY_SCREEN`: Camera will move screenwise.
> - `NO_DEAD_ZONE`: Camera has no deadzone, just tracks the focus object directly and centers it.

The second function will define the rectangular area in which the camera is allowed to scroll, in this case the entire area of the game level. This will stop the camera from following the player when he falls off a cliff.

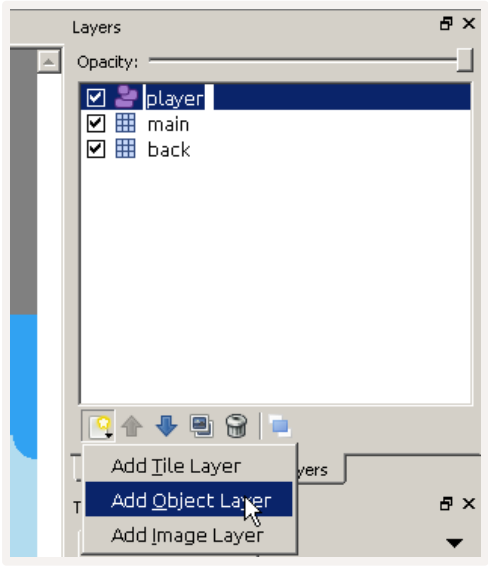
If you run the game now, the camera will track the player and move the camera alongside it while you freely move around your newly created map!

Creating the Player dynamically

Now we are able to create very complex levels, but the player's initial position is still hard-coded in our class: that means that there might be the chances of it being created inside a wall (might already have happened to you if you were really brave in creating your map)!

To fix this, we will create an object in our Tiled map representing the starting position of the player. This will serve as a small introduction to Tiled objects as well, since stuff like enemies and coins will be Tiled objects as well.

Go back to Tiled, and in the `Layers` area, click on the `Objects` tab. It will be empty as we have no objects yet - click on the `Add Object Layer` button, and name the layer `player`.

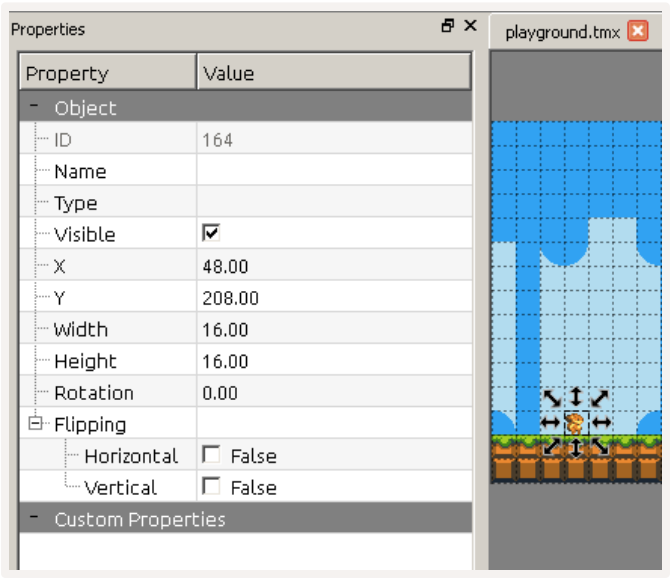


In object layers you can paint objects with dynamic shapes such as rectangles and circles using the corresponding buttons on the toolbar just right of the tilesets paint tools.

Or, if you'd rather use tiles to paint the objects, you can use the picture-shaped button on the furthest right called `Insert Tile (T)`, then pick your chosen tile and click on the canvas to insert the object. You'll see the object appear in the `Objects` tab layer's tree, alongside with its properties on the left side.

Once you placed an object, you can modify its position by making sure the `Select Objects (S)` button is pressed, then clicking on the object and dragging it around.

You can modify its `width` and `height` as well by grabbing the control arrows around it. We won't need to do it for most of our game objects as they will have a standard size of 16x16, but we'll do it for some system-related objects we will insert later in the book (such as the level exit).



Every object has common properties like `ID` (which you cannot change), `Name` (useful to identify the object on the layer tree) and `Type` (we'll use this later to create variations of the same object). We don't need any of those fields on our player object - its position will be enough (you can change the `Name` if you

want), so once you're happy with his location go back to our game code.

First, we are going to modify the player's position, but if you look inside `PlayState.hx` we defined `player` as a private variable. To make it accessible to other classes, we need to change it to a `public` variable, but we don't want other classes to change data of our `player` object - just read information from it.

To enable this functionality, we can use a property to define how the `player` object can be accessed. At the top of the `PlayState`, change the `player` definition to:

```
class PlayState extends FlxState
{
 ...
 public var player(default, null):Player;
```

This will define the behavior when `player` is read (`default`) and written (`null`), making `player` read-only outside of the `PlayState` class.

Now, in `LevelLoader.hx`, we can add the following code at the end of the `loadLevel()` function, right after calling `add()` on the state:

```
...
state.add(backMap);
state.add(state.map);

if (tiledMap.getLayer("player") != null)
{
    var playerLayer:TiledObjectLayer = cast tiledMap.getLayer("player");
 var playerPos:TiledObject = playerLayer.objects[0];
 state.player.setPosition(playerPos.x, playerPos.y - 16);
}
else
 trace("Player object layer not found!");
```

First we check if the object layer called `player` does indeed exist in our map, since loading a non-existing layer will crash our game. If it doesn't, we'll use `trace()` to print a message to the console.

If it does exist, we'll use `cast` to assign it the right type, similarly to what we did with the tile Layers, but this time we'll cast it to a `TiledObjectLayer`. Then, we get an instance of the first object present in that group - if more than one is present, only the first one will be used to specify the player position.

We get ahold of the `player` object from the `state`, and change its position to match the coordinates of the Tiled object.

We offset the y position by 16 since in HaxeFlixel's deafults the `y` origin point is at the top of the sprite, while Tiled's standard location is at at the bottom.

One last thing we'll do is limit the player's `x` position to avoid having him fall off the map if walking past the left edge of the level. Add this to `move()` in `Player.hx`:

```
if (x < 0)
 x = 0;
```

Also, since the player's position is now loaded from the Tiled level, there's no need to specify its position on creation - we can remove the `x` and `y` arguments from its constructor, in `Player.hx`:

```
public function new()
{
 super();
 loadGraphic(AssetPaths.player__png, true, 16, 16);
 ...
```

And when creating the player object, in `PlayState.hx`:

```
override public function create():Void
{
 player = new Player();
 ...
```

Loading a basic level and having the player move around in it works great so far! In the next chapter we'll start adding some collectibles to decorate our levels with.

Coin Collectible

Now that we have a working player and an amazing level, we are ready to start adding objects for the player to interact with, such as bonuses and enemies.

We will start with the classic platformer collectible item, the almighty coin!

Before that, however, let's do some cleanup in our folder structure since our `source` folder is starting to get crowded. In your `source` folder, create 3 folders called `objects`, `states` and `utils`. Re-arrange your files like this:

```
source
|    AssetPaths.hx
|    Main.hx
|
├──objects
|    Player.hx
|
├──states
|    MenuState.hx
|    PlayState.hx
|
└──utils
     LevelLoader.hx
```

It doesn't look like much of a difference at the moment but it will be useful later in the project when we will have several classes.

Since the source `.hx` files reside in different folders, it's a good idea to define their packages. Packages are the directories that contain modules (our `.hx` files). Those `.hx` files contains types defined in the package.

Based on the folder structure we just mentioned, let's declare the `states` package in `PlayState.hx` and `MenuState.hx` by adding this line at the beginning of the file:

```
package states;
```

Similarly, let's declare `package objects;` in `Coin.hx` and `Player.hx`; and `package utils;` in `LevelLoader.hx`.

You might notice how an empty package was declared for them previously, as they resided directly in the `source` folder, with no sub-directories.

Now, everytime we need to access the `PlayState` type of the `PlayState.hx` file, we can import it with `import states.PlayState;`, like we were already doing with HaxeFlixel modules.

Let's do this right now, in the `Main.hx` file: you can see how we are passing `PlayState` to the `addChild()` function - to be able to do so we'll need to import `PlayState`:

```
...
import states.PlayState;

class Main extends Sprite
{
    public function new()
    {
        super();
        addChild(new FlxGame(320, 180, PlayState));
    }
}
```

If we forget to do so, the compiler would return an error like `Type not found : PlayState` or `Unknown identifier : PlayState`. If we code some interaction with a new class in a file, don't forget to import the corrispondent type. If you get stuck, feel free to have a look at the source code for that chapter.

The `Coin` Class

Let's first create a new class in the root of `source` (right next to `Main.hx`) called `Reg.hx`.

This will be our helper class containing some static variables we will use to track the number of coins we got, the game score and the number of lives remaining:

```
class Reg
{
 public static var score:Int = 0;
 public static var coins:Int = 0;
 public static var lives:Int = 3;
}
```

Next, copy the `items.png` file from `book-assets/images` and place it in the `assets/images` folder in your project. This file contains the graphics for collectibles and power-ups we will use in our game.

We can then create a file called `Coin.hx` inside `objects`. We are ready to write our Coin class. Being an active game object, it will extend `FlxSprite`:

```
import flixel.FlxObject;
import flixel.FlxSprite;
import flixel.FlxG;

class Coin extends FlxSprite
{
 private static var SCORE_AMOUNT:Int = 20;

 public function new(x:Float, y:Float)
 {
  super(x, y);
  loadGraphic(AssetPaths.items__png, true, 16, 16);

  animation.add("idle", [0, 1, 2, 3, 4], 16);
  animation.play("idle");
 }

 public function collect()
 {
  Reg.score += SCORE_AMOUNT;
  Reg.coins++;
  if (Reg.coins >= 100)
  {
   Reg.lives++;
   Reg.coins = 0;
  }

  kill();
 }

}
```

In our `create()` function, we are loading the graphic and setting up the correct animation.

Then, we have a function called `collect()`, which we'll call once the player touches the Coin object. It will increase both the score (by a set value defined by the static variable `SCORE_AMOUNT`) and the coins counter, and will add an extra life to the game if we managed to collect 100 coins (a classic rule of every staple platformer!).

At the end of the `collect()` function we will call `kill()`. This is a built-in `FlxBasic` function which sets the `visible` and `active` flags of the object to `false`, effectively "killing" it (we'll learn more about those particular parameters in a few chapters, when dealing with enemies).

Now that the coin logic is there, we have to add some coins to the level. Let's create a `FlxGroup` in `PlayState.hx` to store all of our coins objects.

Import the helper class first:

```
import flixel.group.FlxGroup;
```

Then, right under the class definition where we defined the player variable, add:

```
public var items(default, null):FlxTypedGroup<FlxSprite>;
```

A `FlxTypedGroup` is a `FlxGroup` that can contain only one particular type of object, in our case a `Coin`. Like `player`, this object is read-only outside of `PlayState`.

In the `create()` function, instantiate the group and add it to our level:

```
override public function create():Void
{
 ...
 items = new FlxTypedGroup<FlxSprite>();
 add(items);
 ...
}
```

Now let's take care of the collisions - we'll do this :

```
override public function update(elapsed:Float):Void
{
 super.update(elapsed);

 FlxG.collide(map, player);
 FlxG.overlap(items, player, collideItems);
}

function collideItems(coin:Coin, player:Player):Void
{
 coin.collect();
}
```

The difference between `FlxG.collide()` and `FlxG.overlap()` is that the first one will treat both object as solid, making them stop on collision (it does so by calling an internal function called `FlxObject.separate()`), while the latter allows them to overlap.

Both `FlxG.collide()` and `FlxG.overlap()` can be pointed a callback function to run when the collision is triggered as a third argument. This function takes the two overlapping `FlxObjects` as arguments (in this case the `Coin` and the `Player`).

We didn't call any functions when the player collided with the map, as nothing really had to happen apart from the collision itself. With the coin, however, we have to make sure to make it disappear, and increase the score on overlap.

So, in our game, whenever a `Coin` object of the group `items` overlaps with the `Player` object, it will call the `collideItems()` function, which will call back the `collect()` function of the coin that we defined earlier in the `Coin` class.

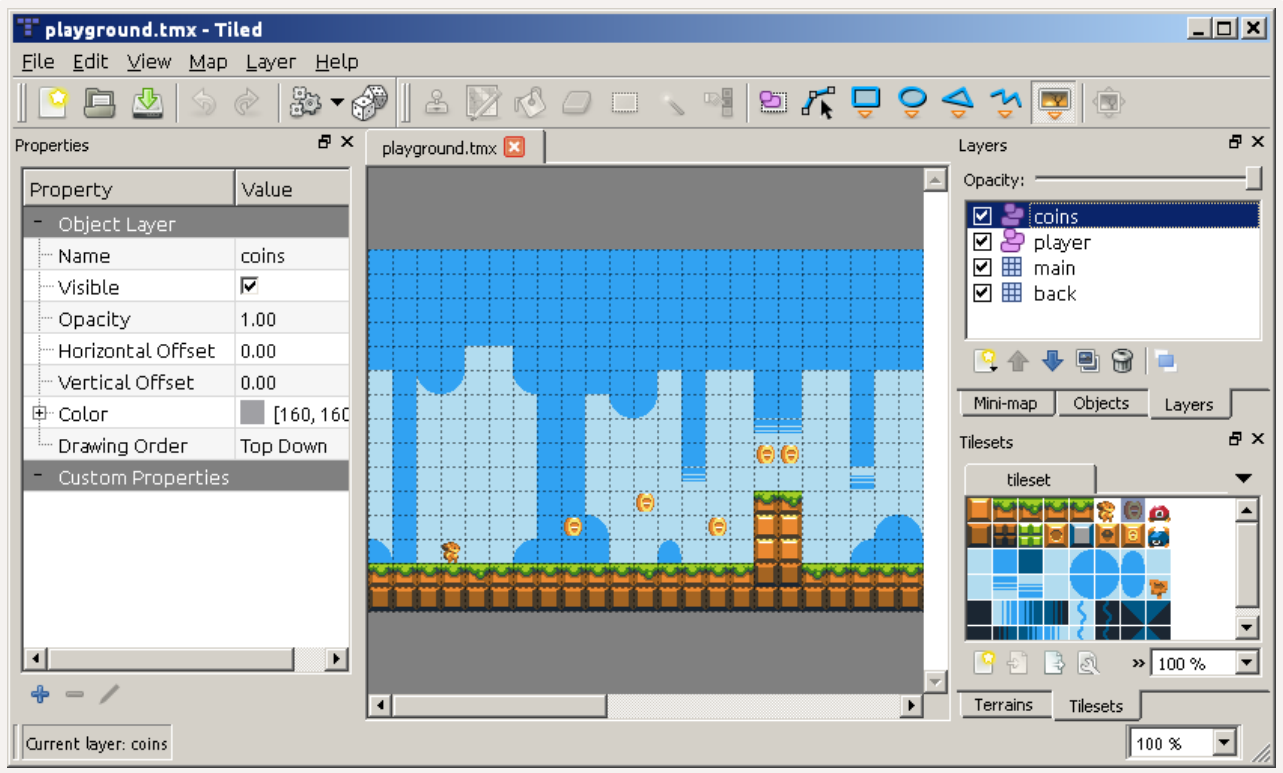
This works because in `collideItems()` we are specifying that the collision will happen with a `Coin` object. If we were to call this same function when the player collides with an enemy, the compiler will return an error since an enemy does not belong to the `Coin` type and therefore does not have the

`collect()` function defined. We'll learn more about this when we introduce more object types.

Placing coins in Tiled Editor

Now, all this would work great if only we had some coins in the level! Instead of adding them via code, we will create a coin object layer in our Tiled level and load them dynamically, in a similar way we did with the player.

Open the level in Tiled, and add a new object layer called `coin`. Create a few objects inside this layer where you want your coins to appear. Like we did with the `player` object, you can use the `Insert Tile (T)` tool to use the coin graphic to paint the objects in the canvas.



Then, in `LevelLoader.hx`, right before we load the player:

```
if (tiledMap.getLayer("coins") != null)
{
    var coinsLayer:TiledObjectLayer = cast tiledMap.getLayer("coins");
 for (coin in coinsLayer.objects)
  state.items.add(new Coin(coin.x, coin.y - 16));
}
else
 trace("Coins object layer not found!");
```

We load the coins by recursively going through each `TiledObject` present in the `coins` layer, and creating a new `Coin` object in its respective position inside the `items` group in `PlayState`.

Run the game and our coins will be floating around the level, right where we placed them in the editor. If

the player touches them, they will disappear, indicating that the collection code is working fine. We have no way of seeing the `score` and `coins` values yet, but we will implement the on-screen counter for them in the next chapter!

If the coins are not appearing, make sure you are adding the `items` group to the state **after** the `loadLevel()` function executes, or else we won't be able to see the objects which `loadLevel()` adds to our level when loading the Tiled map.

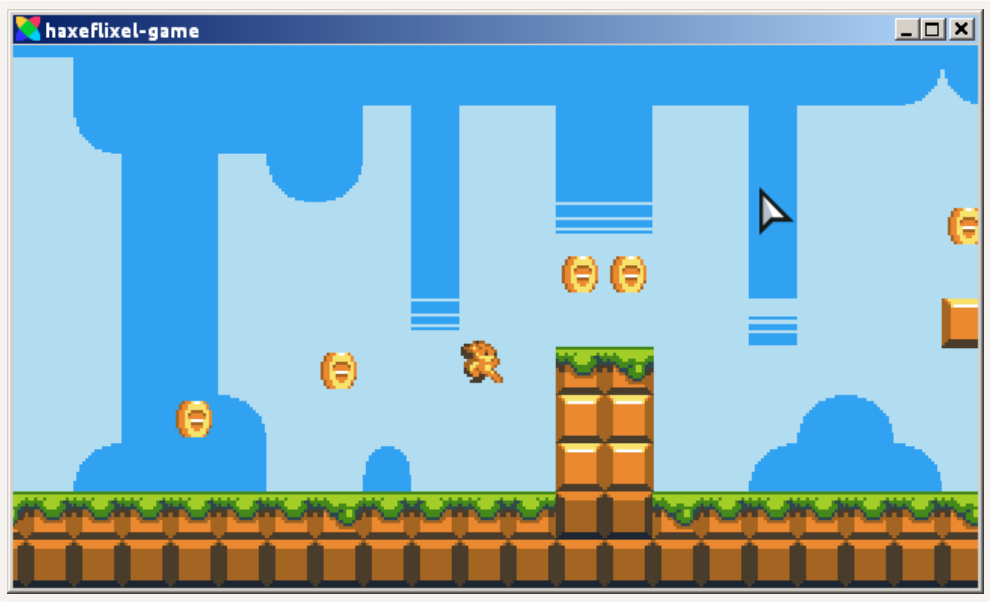
Going back to the coin loading logic, you may think it looks very similar to the one we use to load the player position from the map - and it fact it does - there's lot of code repetition. We can make things cleaner by creating a common function that returns all the `TiledObject` from a `TiledObjectLayer` of our `TiledMap` with a specific name:

```
public static function getLevelObjects(map:TiledMap,
            layer:String):Array<TiledObject>
{
 if ((map != null) && (map.getLayer(layer) != null))
 {
     var objLayer:TiledObjectLayer = cast map.getLayer(layer);
  return objLayer.objects;
 }
 else
 {
  trace("Object layer " + layer + " not found!");
  return [];
 }
}
```

If the supplied `TiledMap` or `TiledObjectLayer` does not exist, the function will simply return an empty array. We can then rewrite our coin and player objects loading logic using this new function:

```
for (coin in getLevelObjects(tiledMap, "coins"))
 state.items.add(new Coin(coin.x, coin.y - 16));

var playerPos:TiledObject = getLevelObjects(tiledMap, "player")[0];
state.player.setPosition(playerPos.x, playerPos.y - 16);
```

Way shorter and readable, isn't it? Give the project a quick test to make sure everything is working and you haven't missed anything.



In the next chapter we'll add some on-screen counters to display the score and coin amounts, alongside with more game-related information such as the current level and the time remaining.

Extra Reading

- HaxeFlixel Handbook: FlxGroup

HUD

In this chapter we will add a HUD (head-up display) to our game, which will display useful gameplay information such as score, lives and such.

First of all, let's modify `Reg.hx` to add a few more variables to track the game time and the current level:

```
class Reg
{
    public static var level:Int = 0;
    public static var score:Int = 0;
    public static var coins:Int = 0;
    public static var lives:Int = 3;
    public static var time:Float = 300;
}
```

The `HUD` class & FlxText

We will make extensive use of the `FlxText` class - a `FlxSprite` which is optimized for drawing text.

Our `HUD` class will be a `FlxSpriteGroup` containing several `FlxText` objects (and later we will add a few `FlxSprite` as well to display some fancy icons).

Let's start by creating the file `HUD.hx` inside the `states` folder and initializing its class:

```
import flixel.FlxBasic;
import flixel.FlxG;
import flixel.FlxSprite;
import flixel.group.FlxSpriteGroup;
import flixel.text.FlxText;
import flixel.math.FlxPoint;
import flixel.util.FlxColor;

class HUD extends FlxSpriteGroup
{
 private var _textScore:FlxText;
    static inline var OFFSET:Int = 4;

    public function new()
    {
        super();

        _textScore = new FlxText(OFFSET, OFFSET, 0);
        add(_textScore);

        _textScore.scrollFactor = FlxPoint.set(0, 0);
    }

    override public function update(elapsed:Float)
    {
        _textScore.text = "SCORE\n" + (Reg.score);
        super.update(elapsed);
    }
}
```

We'll limit our display to the score counter alone to start with and get a better understanding of its workings.

First, we import the required classes and define `HUD` as an extension of `FlxSpriteGroup`.

We then define a private `FlxText` object called `_textScore`, and a static inline variable `OFFSET` with the value of `4` which we will use for measurements later.

In our `new()` function, we initialize the `FlxText` object. It takes several arguments:

- The first two arguments are the `x` and `y` coordinates of the text starting from the top-left corner of the screen. We're passing the `OFFSET` variable we defined earlier, this means that the `FlxText` will be displayed 4 pixels away from the top-left corner.
- The third argument is the width of the text field - the size of the sprite the text will be displayed on. This can be a hard concept to grasp: imagine the text standing on top of a transparent sprite - text can be written only on that sprite's surface. If the text field is too small, the text will be cut off. A value of `0` enables automatic sizing based on the text's length.
- The fourth argument is the text to be displayed. This is omitted in this case since we're manually modifying the text later on to update the score.
- The fifth parameter is optional as well (omitted here), and it's the size of the text. If unsupplied it

will default to `8 pt`.

Afterwards we add the `FlxText` to our group (`add()` works for adding objects to a `FlxGroup` as well, as you saw when loading coins from the Tiled level and adding them to their group).

We then set the `scrollFactor` of `_textScore` to `(0, 0)`. This value is a `FlxPoint`, the HaxeFlixel implementation for a 2-dimensional array.

This stops the object being affected by camera scrolling - it will make it "stick" to the camera screen, keeping the same coordinates of `(4, 4)` even as the camera scrolls, following our player. A default `scrollFactor` value of `(1, 1)` would make the text behave like a standard game object, making it disappear as we move right.

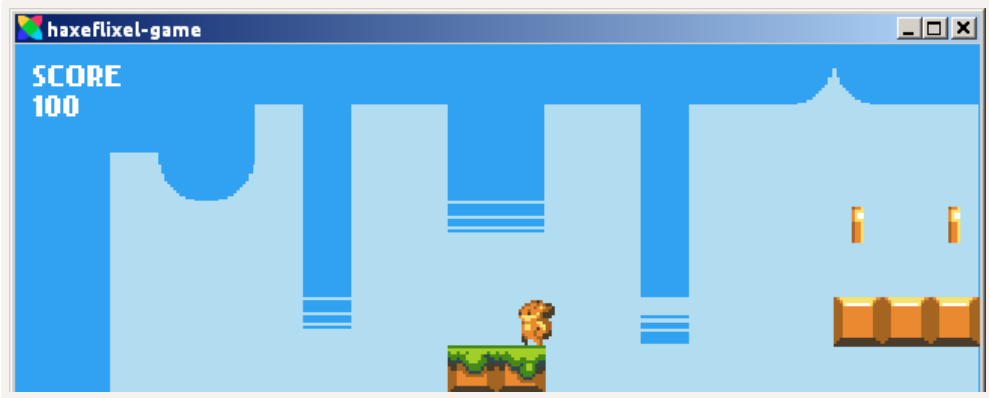
In the `update()` function we will update the `_textScore` `FlxText` to display the current score.

We will do this by setting its `text` field to the literal word `"SCORE"`, plus a new line (the `\n` symbol will force the text to go on a new line), followed by the literal value of our `Reg.score` variable.

Afterwards, let's define and add our `HUD` object in `PlayState.hx`:

```
class PlayState extends FlxState
{
 ...
 private var _hud:HUD;

 override public function create():Void
 {
  ...
  _hud = new HUD();

  LevelLoader.loadLevel(this, "playground");

  add(player);
  add(items);

  add(_hud);
  ...
```

Run the project and the a score label should be visible in the upper left, alongside the score value (try grabbing some coins and watch it increase)! Great!



Let's make it a bit cooler to look at. Change the `update()` function in `HUD.hx` to:

```
override public function update(elapsed:Float)
{
    _textScore.text = "SCORE\n" +
     StringTools.lpad(Std.string(Reg.score), "0", 5);
}
```

`StringTools.lpad()` will add a X number of characters as a left-side padding to a String, if the String is shorter than X. In our case, we are adding `"0"` if the score value is shorter than 5 digits, creating a very arcade-game-ish looking score text. Nice!

Extra counters and `forEach()`

Now that we figured out how `FlxText` works, let's add all the other fields in `HUD.hx`, alongside a small image for the coins counter - make sure to copy the `hud.png` file from `book-assets/images` and place it the `assets/images` folder in your project.

```
class HUD extends FlxSpriteGroup
{
 private var _textScore:FlxText;
 private var _textTime:FlxText;
    private var _textCoin:FlxText;
    private var _textWorld:FlxText;

    private var _iconCoin:FlxSprite;

    public static var OFFSET:Int = 4;

    public function new()
    {
        super();

        _textScore = new FlxText(OFFSET, OFFSET, 0);
        _textCoin = new FlxText(FlxG.width * 0.33, OFFSET + 10, 0);
        _textWorld = new FlxText(FlxG.width * 0.66, OFFSET, 0);
        _textTime = new FlxText(OFFSET, OFFSET, FlxG.width - OFFSET*2);

        add(_textScore);
```

```
        add(_textCoin);
        add(_textWorld);
        add(_textTime);

        _textScore.alignment = FlxTextAlign.LEFT;
        _textCoin.alignment = FlxTextAlign.CENTER;
        _textWorld.alignment = FlxTextAlign.CENTER;
        _textTime.alignment = FlxTextAlign.RIGHT;

        _iconCoin = new FlxSprite(FlxG.width * 0.33 - 4, OFFSET + 10 + 4);
        _iconCoin.loadGraphic("assets/hud.png", true, 8, 8);
        _iconCoin.animation.add("coin", [0], 0);
        _iconCoin.animation.play("coin");

        add(_iconCoin);

        forEach(function (member)
        {
           member.scrollFactor.set(0, 0);
        }, false);
    }

    override public function update(elapsed:Float)
    {
        _textScore.text = "SCORE\n" +
            StringTools.lpad(Std.string(Reg.score), "0", 5);
        _textCoin.text = "  x " +
            StringTools.lpad(Std.string(Reg.coins), "0", 2);
        _textTime.text = "TIME\n" +
            StringTools.lpad(Std.string(Math.floor(Reg.time)), "0", 3);
        _textWorld.text = "STAGE\n" + Reg.level;

        super.update(elapsed);
    }
}
```

Notice how we are multiplying `FlxG.width` by `0.33` and `0.66` to place the text respectively at one third and two thirds of the screen on the X axis.

We also change the `alignment` field to align the text on the left (default), center and right, by setting them to the correct `FlxTextAlign` enumerator value.

Let's have a close look at how we are creating a `FlxText` to be aligned at the right edge of the screen: the first thought would be to set its `x` coordinate at `FlxG.width` (the right edge of the screen), minus the width of its text field. This can be tricky when the text represent a dynamic value, and would require the coordinate to be re-calculated every time the text changes.

What we're doing here instead is placing the `FlxText` at the left edge of the screen, but giving its text field the same width as the screen: this way, `FlxTextAlign.RIGHT` will set its text's alignment to the right edge of the text field - which will be at the right edge of the screen.

We also use the `forEach()` function to set the `scrollFactor` to zero for all elements at once, instead of copy-pasting the same code 4 times.

`forEach()` is a function that can be called on `FlxGroup`, and takes as argument another function which will be executed on all the `member` objects of that group. The second boolean argument specifies whether or not to apply the function to members of subgroups as well.

We are also adding a simple `FlxSprite` to be used as icon for the `_textCoin` counter.

Styling FlxText

Once you make sure all of the `FlxText` objects are working fine, it's time to spice them up a little. `FlxText` is highly customizable, you can change font, add an outline or a shadow very easily.

Download the `pixel_font.ttf` file and place it in `assets/data`. This is the font file we will use to customize our `FlxText` objects.

We'll be using another loop to change font and set an outline for all the `FlxText` in the group. Right before setting the `alignment`, add:

```
forEachOfType(FlxText, function(member)
{
    member.setFormat(AssetPaths.pixel_font__ttf, 8, FlxColor.WHITE,
                     FlxTextBorderStyle.OUTLINE, 0xff005784);
});
```

We use the `forEachOfType()` function to access only `FlxText` objects, and then call the `setFormat()` function, which allows us to change several `FlxText` properties at once.

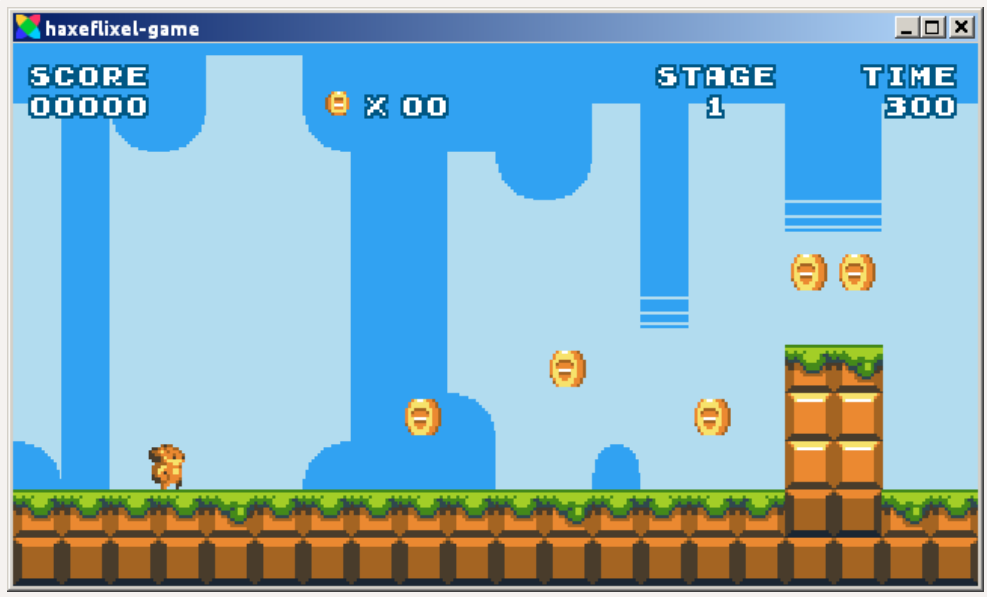
- As first argument, we'll pass the font asset to use.
- Our second argument is the font size - 8 (pt).
- The third argument is the color to use for the text, in this case white (colors in HaxeFlixel are written in 0xAARRGGBB format).
- Our fourth argument will be a border style for the text - `FlxTextBorderStyle.OUTLINE` to draw an outline around the text.
- The fifth argument will be the outline color, written in hexadecimal format as well.

> `setFormat()` can be useful to change more than one property at once. In this case we passed all the arguments for the function, but you can also pass only the arguments you

wish to change and the others will remain unchanged or keep their default values.

The `setFormat()` function will return the `FlxText` object itself in case you want to chain additional operations.

Run the project again and your text will look way more fashionable!



In the next chapter we will add our first real obstacle - a moving and dangerous enemy!

The Enemies

We are now ready to add some enemies to our game. We'll create a classic platformer enemy which will slowly walk in one direction, change direction when hitting an obstacle and can be defeated by the player by jumping on it.

Like we did with the Coin, let's define and add a `FlxTypedGroup` to host our enemy objects in `PlayState.hx` first:

```
class PlayState extends FlxState
{
    ...
    public var enemies(default, null):FlxTypedGroup<Enemy>;
    ...
    override public function create():Void
    {
        ...
        enemies = new FlxTypedGroup<Enemy>();
        ...
        add(enemies);
        ...
```

The `Enemy` Class

Let's create a new file `Enemy.hx` inside `objects`. It's gonna be a long class, so we will have a look at it section by section. Let's start with our imports:

```
import flixel.FlxObject;
import flixel.FlxSprite;
import flixel.FlxG;
import flixel.util.FlxTimer;
```

Then define the class, extending `FlxSprite`:

```
class Enemy extends FlxSprite
```

And define some static variables we will use to tune the enemy's motion:

```
    private static inline var GRAVITY:Int = 420;
    private static inline var WALK_SPEED:Int = 40;
    private static inline var FALLING_SPEED:Int = 200;
    private static inline var SCORE_AMOUNT:Int = 100;

    private var _direction:Int = -1;
    private var _appeared:Bool = false;
```

We'll use `_direction` to keep track of which direction the enemy is moving towards, (1 being right and -

1 being left) and `_appeared` to check whether the enemy is visible within the screen.

Now let's move onto the constructor function `new()`:

```
public function new(x:Float, y:Float)
{
    super(x, y);

    loadGraphic(AssetPaths.enemyA__png, true, 16, 16);
    animation.add("walk", [0, 1, 2, 1], 12);
    animation.add("dead", [3], 12);
    animation.play("walk");

    setSize(12, 12);
    offset.set(2, 4);

    acceleration.y = GRAVITY;
    maxVelocity.y = FALLING_SPEED;
}
```

Copy the file `enemyA.png` from `book-assets/images` and place it in the `assets/images` folder in your project. This file is the sprite sheet for our enemy.

After loading the graphics and setting up the animations of the enemy, we change its bounding box size using `setSize()` & `offset.set()` in the same way we did with `Player`, to avoid collisions with the empty parts of the sprite.

```
override public function update(elapsed:Float)
{
    if (!inWorldBounds())
        exists = false;

    if (isOnScreen())
        _appeared = true;

    if (_appeared && alive)
    {
        move();

        if (justTouched(FlxObject.WALL))
            flipDirection();
    }

    super.update(elapsed);
}

private function flipDirection()
{
    flipX = !flipX;
    _direction = -_direction;
}

private function move()
{
    velocity.x = _direction * WALK_SPEED;
}
```

We first check if the enemy is within the game area with `inWorldBounds()`. If this function returns `false`, it means that the enemy has either fallen off a cliff or exited the stage from the left or the right side. In this case, we can simply get rid of the enemy by setting its `exists` flags to `false`.

> So far, we've extended our game objects from `FlxSprite`. This class, however, extends from lower-level flixel classes which contains common attributes. Starting from the lowest-level object, the object hierarchy is made of:

- `FlxBasic`, the most "generic" Flixel object. Has no size, position or graphical data.`
- `FlxObject`, the base class for most of the display objects, extends `FlxBasic` with some basic attributes about game objects, basic state information, sizes, scrolling, and basic physics and motion.
- `FlxSprite`, The main "game object" class, extends `FlxObject` with a bunch of graphics options and abilities, like animation and stamping.

> Each `FlxObject` possesses four basic flags to control its status in the game - whether it's visible, collides with other objects, and such. They are:

- `active` an object being active will have his `update()` method called every frame.
- `visible` an object being visible will have his `draw()` method called every frame - basically will be drawn and visible on screen.

- `exists` whether the object has been destroyed or not. You can think of this as handling both `active` and `visible` at once.
- `alive` does not influence drawing nor updating, but it can be useful to track the status of the object.

> By default, the built-in function `kill()` sets `alive` and `exists` to false.

Afterwards, we check if the object is visible within the current camera with `isOnScreen()`. If so, we set our variable `_appeared` to `true`.

We do this because we don't want the enemy to move if it hasn't appeared on the screen, or else the player could just stand still at the beginning of the level and eventually all the enemies off-screen would end up walking up to him!

If the `_appeared` and `alive` variables are both true, we call the `move()` function, which we defined right after `update()`. `move()` translates the enemy towards its `_direction` by the `WALK_SPEED` amount. The velocity is measured in pixels / second.

We also check if the enemy has just touched a wall using the `justTouched()` function. If so, we invert its `_direction` variable so that it starts moving in the other direction.

> `justTouched()` returns `true` the moment the object has collided with another solid object. It takes the direction you want to check the collision against as an argument. The `FlxObject` class gives us a few set directions to use in common scenarios, such as:

- `FlxObject.WALL` Checks for `LEFT` and `RIGHT` direction. You can also use `FlxObject.LEFT` and `FlxObject.RIGHT` separately.
- `FlxObject.UP` Checks for `UP` direction.
- Both `FlxObject.FLOOR` and `FlxObject.FLOOR` checks for `DOWN` direction.
- `FlxObject.ANY` checks for any direction

We'll now take care of the interaction with the player. We want the enemy to be killed if the player jumps on him, and the player to bounce afterwards.

To do so, we'll wrap the player's jump functionality outside of `move()` into a small function called `jump()` that we'll be able to call from other objects. in `Player.hx`:

```
public function jump()
{
    velocity.y = JUMP_FORCE;
}

private function move()
{
   ...

    if (velocity.y == 0)
    {
        if (FlxG.keys.pressed.C && isTouching(FlxObject.FLOOR))
            jump();
    ...
```

We can now go back to `Enemy.hx` and write an `interact()` function which we will call later in the `PlayState` when managing the collisions with the player, in a similar way we did with the `Coin` object.

```
public function interact(player:Player)
{
    if (!alive)
        return;

    FlxObject.separateY(this, player);

    if ((player.velocity.y > 0) && (isTouching(FlxObject.UP)))
    {
        kill();
        player.jump();
    }
    else
        player.kill();
}
```

When interacting with the player, we first check if the enemy is alive - if not we'll stop the execution.

If the enemy is indeed alive, we check which direction the player is moving when touching the enemy.

We want to kill the enemy if the player landed on him after a jump. To do so, we first whether the player was going down on the vertical axis by validating that its `velocity.y > 0`.

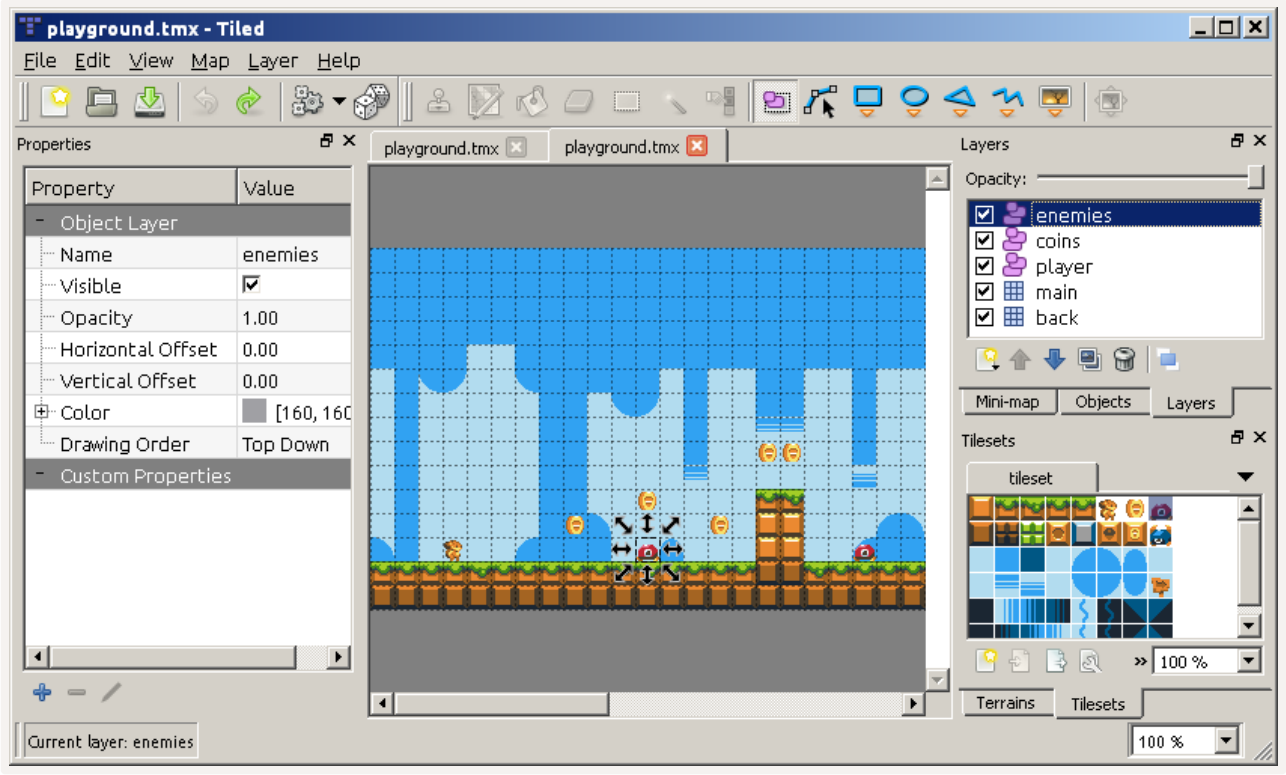
Then we check if the enemy has just collided with another object against the `FlxObject.UP` direction, which means the colliding object was coming from above.

If both conditions are satisfied, it indeed means the player just landed on the enemy - we'll call the `kill()` method on the enemy and `jump()` on the player to make him jump of the killed enemy.

If one or both conditions are not met, it means that the player touched the enemy in some other way and it'll be the player's turn to be damaged this time! We call `kill()` on the player.

Placing enemies in Tiled Editor

Now it's time to add some enemies to the level. Open the level in Tiled, add a new object layer called `enemies` and place some enemies around.



Then, let's load them in `LevelLoader.hx`, right after loading the `Coin` objects:

```
// Load enemies
for (enemy in getLevelObjects(tiledMap, "enemies"))
    state.enemies.add(new Enemy(enemy.x, enemy.y - 16));
```

We're almost there! If you run the game now, you'll be able to see the enemies moving but they'll fall right through the floor. That's because we need to add their collisions with both map and player in the `update()` function of `PlayState.hx`:

```
override public function update(elapsed:Float):Void
{
    super.update(elapsed);

    FlxG.collide(map, player);
    FlxG.overlap(items, player, collideItems);
    FlxG.overlap(enemies, player, collideEnemies);

    FlxG.collide(map, enemies);
    FlxG.collide(enemies, enemies);
}

function collideItems(coin:Coin, player:Player):Void
{
    coin.collect();
}

function collideEnemies(enemy:Enemy, player:Player):Void
{
    enemy.interact(player);
}
```

Run the project now, and the enemies should be able to walk on the map, dangerously approaching the player! If you touch them, the player will be killed by the enemy and will disappear.

Moreover, if we simply add a collision check of the enemies against themselves in `PlayState.hx`:

```
FlxG.collide(_enemies, _enemies);
```

The enemies will flip directions when they collide each other, thanks to the `justTouched(FlxObject.WALL)` check we run in their `move()` method.

We'll add proper death animations and restarting of the level in later chapters.

Enemy death animation

If you jump on enemis, they'll be the one being killed, disappearing after the player bounces on them.

We can make their death animation more exciting right now, by overriding their `kill()` method and leaving their corpse in the game for a few extra seconds after being killed. In `Enemy.hx`, add:

```
override public function kill()
{
    alive = false;
    Reg.score += SCORE_AMOUNT;

    velocity.x = 0;
    acceleration.x = 0;
    animation.play("dead");

    new FlxTimer().start(1.0, function(_)
    {
        exists = false;
        visible = false;
    }, 1);
}
```

This way, when we call `kill()`, their `alive` flag will be set to `false` (so that we don't check collisions with the player once they're dead) and the score will increase by the amount defined by `SCORE_AMOUNT`.

We'll stop the enemy as well setting it `velocity.x` and `acceleration.x` to zero (but not the `y` values, so that an enemy will still fall if killed in mid-air), and change their animation to `"dead"`.

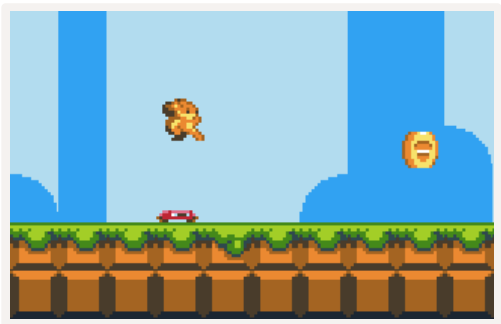
We don't want their corpse to stay in the level forever, so we'll create a new, non-repeating `FlxTimer`, start it and set it to call a function after 1 second, which will set the `exists` and `visible` enemy flags to `false`.

> `FlxTimers` are very useful objects which can trigger events after a set amount of time, either just once or at a repeating interval. They are initialized with the `start()` method, which takes this arguments:

- The amount of time to wait before triggering the function, in seconds
- The function to be called when the timer triggers.
- How many times to repeat the timer. A value of `-1` will repeat the events indefinitely.

Notice how with the `FlxTimer`, instead of defining the callback function separately and pointing to it, we are defining the function directly in the argument list. This works fine and saves some time but might get very messy because of the syntax, so I usually do this with short functions that are used only once.

Run the game, and squishing an enemy should feel way more satisfying than it was before!



Variable bounce height

Right now, when the player lands on an enemy it will bounce back up using the maximum jump height. Let's add some degree of control based on whether the player is holding down the jump button or not, similar to what we did with the player jump mechanics in a few chapters ago.

Change the `jump()` function in `Player.hx` to:

```
public function jump()
{
    if (FlxG.keys.pressed.C)
        velocity.y = JUMP_FORCE;
    else
        velocity.y = JUMP_FORCE / 2;
}
```

What we're doing here is increasing the vertical speed of the player to the full value of `JUMP_FORCE` only when the jump button is being held down: if not, we increase it by half that value.

This way, when the player lands on an enemy the user will have better control on how far it'll bounce back.

In the next chapter we'll add some death animation for our player, and we will also make a `lives` variable to check for an eventual game over and restarting the game.

Extra Reading

- HaxeFlixel Handbook: FlxText

Losing Lives

In this chapter we'll make the player's death a bit more engaging than having him simply disappear. We'll temporarily freeze the game, show a death animation, and have him fall off the screen before restarting the level.

Let's start by creating a variable called `pause` in `Reg.hx`:

```
public static var pause:Bool = false;
```

We'll use this boolean flag to "freeze" objects in the game whenever we don't want them to move or collide, for example while the player is dying.

Let's implement this in both `Coin.hx` and `Enemy.hx`: when `Reg.pause` is `true`, we'll want to skip their `update()` function so that their position is not updated nor their graphics re-drawn:

```
override public function update(elapsed:Float)
{
    ...
    if (!Reg.pause)
        super.update(elapsed);
}
```

While in `Enemy.hx` we can just add this implementation to its current `update()`, in `Coin.hx` we have to write the whole `update()` method since we weren't overriding it before.

We also don't want the player to move when paused, therefore we'll wrap its `move()` function around a check as well, in `Player.hx`:

```
override public function update(elapsed:Float)
{
    if (!Reg.pause)
        move();
```

If the player is dead, we want to stop checking for its collision against other objects, since we want it to fall through the map. In the `update()` function of `PlayState.hx`:

```
override public function update(elapsed:Float):Void
{
    super.update(elapsed);

    if (player.alive)
    {
        FlxG.collide(map, player);
        FlxG.overlap(items, player, collidePlayer);
        FlxG.overlap(enemies, player, collidePlayerEnemy);
    }
    ...
```

Afterwards, open `Player.hx` and change its `animate()` method to display the "death" animation as soon as its `alive` flag is set to `false`. We'll add it at the top of the conditions list, because if the player is dead we'll likely want to have its "death" animation have priority over any other animation:

```
private function animate()
{
    if (!alive)
        animation.play("dead");
    else if ((velocity.y <= 0) && (!isTouching(FlxObject.FLOOR)))
        animation.play("jump");
    else if ...
```

Now let's override the `kill()` method in `Player.hx`:

```
override public function kill()
{
    if (alive)
    {
        alive = false;
        velocity.set(0, 0);
        acceleration.set(0, 0);
        Reg.lives -= 1;
        Reg.pause = true;
        new FlxTimer().start(2.0, function(_)
            {
                acceleration.y = GRAVITY;
                jump();
            } , 1);
        new FlxTimer().start(6.0, function(_) {
                FlxG.resetState();
            });
    }
}
```

After checking that the player is indeed `alive`, we immediately set that parameter to `false` - we don't want to kill the player more than once! This will also activate the "death" animation, as defined earlier.

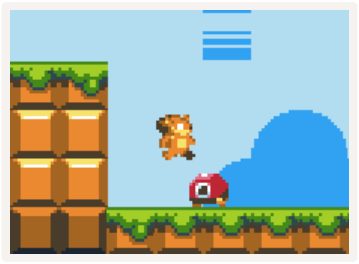
We stop all the player's movement by setting its `velocity` and `acceleration` to zero. Since we stopped the collision checks against the map when the player is dead, we need to temporarily cancel its `acceleration.y` as well or else it will fall through the map right away. The `set()` function can be used on two-dimensional points to quickly set both `x` and `y` value at once.

We reduce the `Reg.lives` variable by one, and set `Reg.pause` to `true` to "freeze" the other game objects.

We'll start two `FlxTimers`. The first one will be triggered after 2 seconds, and will make the player jump and restore its `acceleration.y` back to its original `GRAVITY` value, so it will look like it's bouncing off the map.

The second one will go off after 5 seconds, and it will call `FlxG.resetState()` to restart the current state.

Test the game and have the player touch an enemy - this new death process should work smoothly.



You'll notice, however, that upon restarting the room nothing will move! That's because we forgot to set `Reg.pause` back to `false` after the player was killed. Let's do that now, on top of the `create()` function in `PlayState.hx`:

```
override public function create():Void
{
    Reg.pause = false;
    ...
```

Into the void

Let's make the player able to die by falling off a cliff as well - more precisely, when it falls past the bottom edge of the screen.

To do so, we keep track of his `y` coordinate and call `kill()` as soon as it becomes higher than the map height (meaning the player has effectively fallen past the bottom limit of the level).

To do so, we need to access the variable `map` inside our `PlayState`. However, `Player` has no access to the `PlayState`!

There are many ways to fix this. We could execute the check in `PlayState` instead, which has access to both `map` and `player`. But I'd like to keep all the player-related functions inside `Player.hx`.

Another way to do this could be passing an instance of the current `PlayState` to the `Player` constructor and store it in a variable we can access within it. This is not a bad way to do it.

A common pattern among HaxeFlixel developers is to store a global reference to the current `PlayState` inside `Reg.hx` and access that every time we need to reference the `PlayState` from an object.

In `Reg.hx`, add:

```
public static var PS:PlayState;
```

And as soon as the `PlayState` is created, make it set a reference to itself inside that variable:

```
class PlayState extends FlxState
{
    ...
    override public function create():Void
    {
        Reg.PS = this;
        ...
```

This way we can access both `PlayState` and the public objects in it from anywhere within our game. This will come in handy in a few chapters as well, where we'll encounter cases where we'll need to create and `add()` game objects to the `PlayState` from other game objects, for example in the case of a bonus block generating coins.

But for now let's implement death by falling. At the end of the `move()` function in `Player.hx`, write:

```
if (y > Reg.PS.map.height - height)
    kill()
```

> Another way to accomplish this instead of using `Reg.PS` would be to use `FlxG.state`, a built-in HaxeFlixel variable which points to the current `FlxState`.

> It's similar to our `Reg` implementation, however it's set automatically when the states in the game are switched and it returns a generic `FlxState`, meaning we'll have to `cast` it to our class to access specific functions and variables like `map`.

Time's up!

Now we'll implement the time counter ticking down, giving the player a set amount of time to finish the level. If the time's up, we'll kill the player. Make a new function called `updateTime()` in `PlayState.hx`:

```
private function updateTime(elapsed:Float)
{
    if (!Reg.pause)
    {
        if (Reg.time > 0)
            Reg.time -= elapsed;
        else
        {
            Reg.time = 0;
            player.kill();
        }
    }
}
```

And call it at the end of `update()`, passing to it the `elapsed` value:

```
override public function update(elapsed:Float):Void
{
    super.update(elapsed);
    ...
    updateTime(elapsed);
}
```

We'll run this function only when `Reg.pause` is `false`, since it wouldn't be fair to make the clock tick while the game action is stopped!

If the `Reg.time` variable is larger than zero, we'll reduce it by the value contained in `elapsed`, an internal variable which tells how much time has passed since last frame - it will always add up to a second.

> `elapsed` is used with the concept of "Delta Time", dealing with time-based values which are independent from the speed your game is running at. Let's say that the game is running a 60 frames per second. `elapsed` will be equal to `1/60`, and throughout 60 frames those values will add up to 1. But if your game is running at 25 frames per second, `elapsed` will be equal to `1/25`. When added 25 times, that still adds up to 1!

If `Reg.time` is smaller than zero, we'll set it back to `0` (to avoid having the time counter in the HUD potentially displaying a negative value), and call the `kill()` method on the player.

If the player dies this way, however, we must not forget to revert the time back to its original value, or else the player will die again immediately when the room restarts, since the time value will be still at zero!

Let's reset the time first in the `create()` function of `PlayState.hx`, right after we reset `Reg.pause` to `false`:

```
override public function create():Void
{
    Reg.pause = false;
    Reg.time = 300;
    ...
```

In the next chapter we'll implement both a main menu and an intro screen which displays the remaining lives, checks for game overs and saves the high score throughout game sessions.

Refactors & Optimizations

In this chapter we'll take a break from adding new features and rewrite some of our methods to make them more flexible and powerful.

Nested groups and collisions

Let's take a moment to look at our collision logic right now:

```
override public function update(elapsed:Float):Void
{
 super.update(elapsed);

 if (player.alive)
 {
  FlxG.collide(map, player);
  FlxG.overlap(items, player, collideItems);
  FlxG.overlap(enemies, player, collideEnemies);
 }

 FlxG.collide(map, enemies);
 FlxG.collide(enemies, enemies);

 updateTime();
}

function collideItems(coin:Coin, player:Player):Void
{
 coin.collect();
}

function collideEnemies(enemy:Enemy, player:Player):Void
{
 enemy.interact(player);
}
```

We calculate the collisions between the Player and the coins, and call `collideItems()` to `collect()` the coin.

Afterwards, we calculate the collisions between the Player and the enemies, and call `collideEnemies()` to make them `interact()`.

It's working quite well right now, but think of the amount of game objects we are going to add to our game throughout its development - more enemies, power-ups, treasure chests, portals…

Whilst writing collision logic for each object type is correct from a coding perspective, it certainly takes way longer to write and is harder to modify or debug.

When making a game which is more complex than a few screens of gameplay, it's a good idea to stop often and think how well the code we're writing today is going to work tomorrow.

Wouldn't it be simpler if we were to check the collisions between the player and any game object, and only then query the type of the object and act accordingly?

We can do this by using nested groups. In the class declaration of `PlayState.hx`, define a new `FlxGroup` object:

```
private var _entities:FlxGroup;
```

And initialize it on `create()`:

```
_entities = new FlxGroup();
```

`FlxGroup` is a short for a `FlxTypedGroup` of `FlxBasic` objects. You can add any object to it, as long as it extends `FlxBasic` (which all of our gameplay objects do).

Now, instead of adding `enemies` and `items` directly to the state, we'll add them to `_entities` and add this group to `PlayState` instead:

```
...
add(player);

_entities.add(items);
_entities.add(enemies);
add(_entities);

add(_hud);
...
```

If you run the game, everything will keep working fine. That's because our enemies and items are both part of their specific groups `enemies` and `items`, where the collisions are calculated; and part of the bigger `_entities` group which is added to the state.

Now we can rewrite our collision logic using the `_entities` group:

```
override public function update(elapsed:Float):Void
{
 super.update(elapsed);

 if (player.alive)
 {
  FlxG.collide(map, player);
  FlxG.overlap(_entities, player, collideEntities);
 }

 FlxG.collide(map, _entities);
 FlxG.collide(enemies, enemies);

 updateTime();
}

function collideEntities(entity:FlxSprite, player:Player):Void
{
 if (Std.is(entity, Coin))
  (cast entity).collect();

 if (Std.is(entity, Enemy))
  (cast entity).interact(player);
}
```

We are simply testing the collisions of `player` against `_entities` and passing the function `collideEntities()` as a callback.

As you can see, this function sees the object colliding with the player as a generic `FlxSprite` (as `_entities` might contain several different types of objects), so we need to check which object the player is colliding with before taking action.

We do this using `Std.is()` and once the object type is confirmed, we can `cast` the generic `FlxSprite` entity to its type and use the appropriate function (like `collect()` for the `Coin` and `interact()` for the `Enemy`).

Doing so without casting would result in an error, since the generic `FlxSprite` doesn't possess any `collect()` or `interact()` methods!

We also call `FlxG.collide(map, _entities)` so that enemies will be able to walk on the map - this will save us a bit of time later when we'll create power-ups, since making them part of `_entities` will allow them to collide with the map as well.

No reason we can't still use the original groups, either - we still call `FlxG.collide(enemies, enemies)` to make the enemies collide with each other and trigger the direction change.

The Enemy Parent class

Another optimization we're going to write is adapting the current `Enemy` class we've got into a generic, parent class where we'll keep the functionality which is common to all enemies (such as dying or

changing direction when hitting a wall).

This will make it easier to create and add new enemy type, because we can write the functionality which is exclusive to the new type, such as graphics and movement, and the rest will be inherited from the generic parent class.

In `Enemy.hx`:

```
class Enemy extends FlxSprite
{
 ...
 public function new(x:Float, y:Float)
 {
  super(x, y);

  acceleration.y = GRAVITY;
  maxVelocity.y = FALLING_SPEED;

  flipX = true;
  _direction = -1;
 }

 ...

 private function move() {}
```

Change the `new()` function and remove the graphic-related method calls, and remove everything inside `move()` as well, effectively leaving an empty function.

We are moving those to a new class called `WalkEnemy` which will extend `Enemy`. Create a new file inside `objects` called `WalkEnemy.hx`:

```
import flixel.FlxObject;
import flixel.FlxSprite;
import flixel.FlxG;
import flixel.util.FlxTimer;

class WalkEnemy extends Enemy
{
 private static var WALK_SPEED:Int = 40;
 private static var SCORE_AMOUNT:Int = 100;

 public function new(x:Float, y:Float)
 {
  super(x, y);

  loadGraphic(AssetPaths.enemyA__png, true, 16, 16);
  animation.add("walk", [0, 1], 12);
  animation.add("dead", [2], 12);
  animation.play("walk");

  setSize(12, 12);
  offset.set(2, 4);
 }

 override private function move()
 {
  velocity.x = _direction * WALK_SPEED;
 }
}
```

As you can see, we're only loading the enemy graphics, setting its size and defining its `move()` function. Everything else will be loaded from its parent `Enemy` class.

The `override` in `move()` will make sure that the original function in the parent's class will be replaced by the newly defined one in the child. Same applies to its variable like `WALK_SPEED` and `SCORE_AMOUNT`.

The only missing step now is to change `LevelLoader.hx` to add a `WalkEnemy` when loading the map instead of `Enemy`, since now that's a generic class without graphics:

```
for (enemy in getLevelObjects(tiledMap, "enemies"))
 state.enemies.add(new WalkEnemy(enemy.x, enemy.y - 16));
```

Let's try and create a new type of enemy. Create a new file `SpikeEnemy.hx` inside `objects`:

```
import flixel.FlxObject;
import flixel.FlxSprite;
import flixel.FlxG;
import flixel.util.FlxTimer;

class SpikeEnemy extends Enemy
{
 private static var WALK_SPEED:Int = 40;
 private static var SCORE_AMOUNT:Int = 100;

 public function new(x:Float, y:Float)
 {
  super(x, y);

  loadGraphic(AssetPaths.enemyB__png, true, 16, 16);
  animation.add("walk", [0, 1], 12);
  animation.play("walk");

  setSize(12, 12);
  offset.set(2, 4);
 }

 override private function move()
 {
  velocity.x = _direction * WALK_SPEED;
 }

 override public function interact(player:Player)
 {
  if (alive)
   player.kill();
 }
}
```

The `create()` method is very familiar - we initialize graphics and size.

Make sure to copy the file `enemyB.png` from `book-assets/images` and place it in the `assets/images` folder in your project. This file is the sprite sheet for this new SpikeEnemy.

This enemy will move the same way as `WalkEnemy`, so the overridden `move()` function is identical.

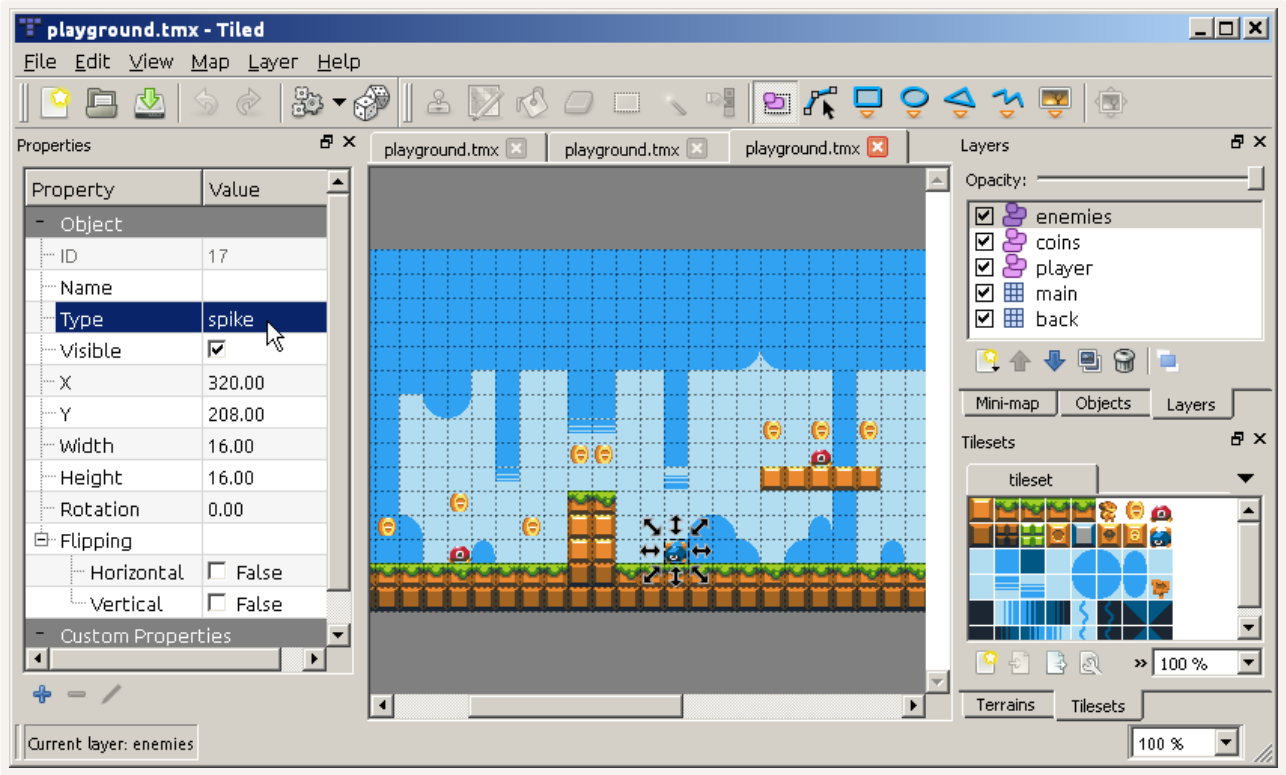
However, we don't want the enemy to die when the player jumps on him - since it has spikes on its back, we want the player to die instead!

To do so, we override the `interact` function and simply call `player.kill()` in it. This function will replace the one from its parent class, so when the player collides with this type of enemy, it will die no matter what.

Now we'll need to think about how to add those new enemies to our level. We could make a different layer for each enemy type, but if we add lots of enemies things could easily start going out of hand.

That's where the `type` parameter of a Tiled object comes in hand. Open Tiled, and add a new object inside the `enemy` layer. In the "Properties" tab on the right, you'll see a field called "Type". Write `spike`

into the field, and save the level.



We'll be able to query the `type` parameter when loading the map, and instantiate different enemy objects based on its value.

To do so, replace the enemy loading code in `LevelLoader.hx`:

```
// old code
// for (enemy in getLevelObjects(tiledMap, "enemies"))
//  state.enemies.add(new WalkEnemy(enemy.x, enemy.y - 16));

for (enemy in getLevelObjects(tiledMap, "enemies"))
{
 switch(enemy.type)
 {
  default:
   state.enemies.add(new WalkEnemy(enemy.x, enemy.y - 16));
  case "spike":
   state.enemies.add(new SpikeEnemy(enemy.x, enemy.y - 16));
 }
}
```

We use a `switch()` statement to check the `type` parameter of the enemy `TiledObject` being loaded. If the `type` is not recognized, we default to the generic `WalkEnemy` type.

In the next chapter we'll implement a small intro screen showing the number of lives remaining using SubStates.

Intro Screen

In this chapter we will implement a small intro screen at the beginning of each level showing the number of lives remaining. Instead of using a full-fledged `FlxState` to display it, we will use a `FlxSubState`.

The `IntroSubState` Class & FlxSubState

A substate is nothing more than a "secondary" state which pauses the existing state and is drawn on top of it. Once the substate is closed, the main state will run again.

You can already understand how substates are very useful for implementing pause screens, inventories, dialog boxes and such.

Create a new file `IntroSubState.hx` inside the `states` folder:

```
package states;

import flixel.FlxG;
import flixel.FlxSprite;
import flixel.text.FlxText;
import flixel.FlxCamera;
import flixel.FlxObject;
import flixel.FlxSubState;
import flixel.util.FlxTimer;
import flixel.util.FlxColor;

import flash.system.System;

class IntroSubState extends FlxSubState
{
    private var _text:FlxText;
    private var _iconLives:FlxSprite;
    private var _textLives:FlxText;

    private var _gameOver:Bool = false;
    private var _waitToDisappear:Float = 3.0;

    override public function create():Void
    {
        super.create();

        if (Reg.lives < 0)
            _gameOver = true;

        _text = new FlxText(0, FlxG.height/2 + 8, FlxG.width, "Get Ready!");
        _textLives = new FlxText(FlxG.width*0.5, FlxG.height/2 - 8,
                                 FlxG.width, "x " + Reg.lives);

        _iconLives = new FlxSprite(FlxG.width*0.5 - 20, FlxG.height/2 - 4);
        _iconLives.loadGraphic(AssetPaths.hud__png, true, 8, 8);
        _iconLives.animation.add("life", [1], 0);
        _iconLives.animation.play("life");
```

```
        add(_text);
        if (_gameOver)
        {
            _text.text = "Game Over";
            _text.setPosition(0, FlxG.height/2);
        }
        else
        {
            add(_iconLives);
            add(_textLives);
        }

        forEachOfType(FlxObject, function(member)
        {
           member.scrollFactor.set(0, 0);
        });

        forEachOfType(FlxText, function(member)
        {
            member.setFormat(AssetPaths.pixel_font__ttf, 8, FlxColor.WHITE,
                             FlxTextBorderStyle.OUTLINE, 0xff005784);
        });

        _text.alignment = FlxTextAlign.CENTER;

        new FlxTimer().start(_waitToDisappear, function(_)
        {
            if (_gameOver)
                System.exit(0);
            else
                close();
        } , 1);
    }
}
```

Let's see what's going on: we initialize our `FlxSubState`, alongside with the graphic and text we will use to display the remaining lives, a `_gameOver` variable we'll use to track the state of the game and a `_waitToDisappear` variable which will control how long the substate will be displayed for.

In the `create()` function, we first check if the player has any lives remaining - if not, `_gameOver` is set to `true`.

We then create the icon and text counter for the lives, and a small text under it which says "Get ready!". If `_gameOver` is `true`, this text will be changed to "Game Over" and the lives counter will not be displayed.

We then use a `forEachOfType()` to set the scrollFactor on all `FlxObject` objects to `(0, 0)` (this will happen on texts as well, as `FlxText` extends `FlxSprite`), and to customize the font and style in the same way we did with the `HUD` object for all `FlxText` objects. We set the alignment variable as well.

We then start a `FlxTimer` which will trigger after the amount of seconds defined in the `_waitToDisappear` variable. This action will check the `_gameOver` variable: if `true`, the game will quit; if not the `FlxSubState` will close.

We will initialize this substate in the `create()` function of the `PlayState` - this way, the player will first see the intro screen, and after the substate closes, the `PlayState` will be un-paused, effectively starting the game.

In `PlayState.hx`:

```
...
import flixel.util.FlxColor;

class PlayState extends FlxState
{
 ...

 override public function create():Void
 {
  // We get rid of the gameOver check, since it's being done
  // in the SubState now!
  Reg.pause = false;
  Reg.time = 300;

  ...

  openSubState(new IntroSubState(FlxColor.BLACK));

  super.create();
 }
 ...
```

You'll notice how we pass a color to the substate constructor: it sets the background color for the substate. If we were to pass `FlxColor.TRANSPARENT` instead, we'd see the intro state with the paused `PlayState` underneath.

Multiple cameras

Now, we'd want some extra info on the intro screen, such as the current level and the current score. It would be great if we could display the `HUD` on it, since it has all these statistics we want.

Unfortunately, the `HUD` is displayed in the `PlayState`, which is drawn underneath the `IntroSubState`, making it invisible.

We could create another `HUD` object and add it to the `IntroSubState`, or copy its `FlxText` objects. Those however are not very elegant solutions, since they create a lot of duplicated code.

Luckily, we have a smarter way of doing this: we'll draw the `HUD` on a separate camera.

Think of a `FlxCamera` as a "virtual canvas" - the game's sprites are drawn on it, and then the camera is displayed on the screen, showing you the game action.

By default one camera is created automatically, and all the sprites in the game are drawn on it. It can be useful to add more cameras and tell the game which objects are displayed on which cameras, and that's exactly what we'll do now.

In `PlayState.hx`, define two new `FlxCamera` objects:

```
private var _gameCamera:FlxCamera;
private var _hudCamera:FlxCamera;
```

And initialize them first thing in the `create()` function:

```
override public function create():Void
{
 Reg.PS = this;
 Reg.pause = false;

 _gameCamera = new FlxCamera();
    _hudCamera = new FlxCamera();
    FlxG.cameras.reset(_gameCamera);
    FlxG.cameras.add(_hudCamera);
    _hudCamera.bgColor = FlxColor.TRANSPARENT;
    FlxCamera.defaultCameras = [_gameCamera];
    ...
```

We first initialize the two `FlxCamera()` objects.

We then call `reset()` on `FlxG.cameras`, which is an object which manages all cameras in the game. `reset()` simply empties this array, and passing `gameCamera` as argument will tell it to make `_gameCamera` the new default camera.

We add `_hudCamera` to `FlxG.cameras`; and since it has been added after `_gameCamera`, it will be displayed on top of it.

We then set its `bgColor` to `FlxColor.TRANSPARENT` so we can see the objects on `_gameCamera` underneath (by default, the camera's background color is black).

We then call `FlxCamera.defaultCameras = [_gameCamera]`: this will tell the game that any object which was using the default camera and any new object created from now on has to be displayed on `_gameCamera`, unless specified.

Now we need to tell the `HUD` graphics to be displayed on `_hudCamera` instead. Go to `HUD.hx`, and create a new function:

```
public function setCamera(cam:FlxCamera)
{
    forEach(function (member)
    {
       member.scrollFactor.set(0, 0);
       member.cameras = [cam];
    });
}
```

This function will take a `FlxCamera` `cam` as an argument, and will go through each objects belonging to `HUD` and set its `cameras` attribute to this `cam` argument.

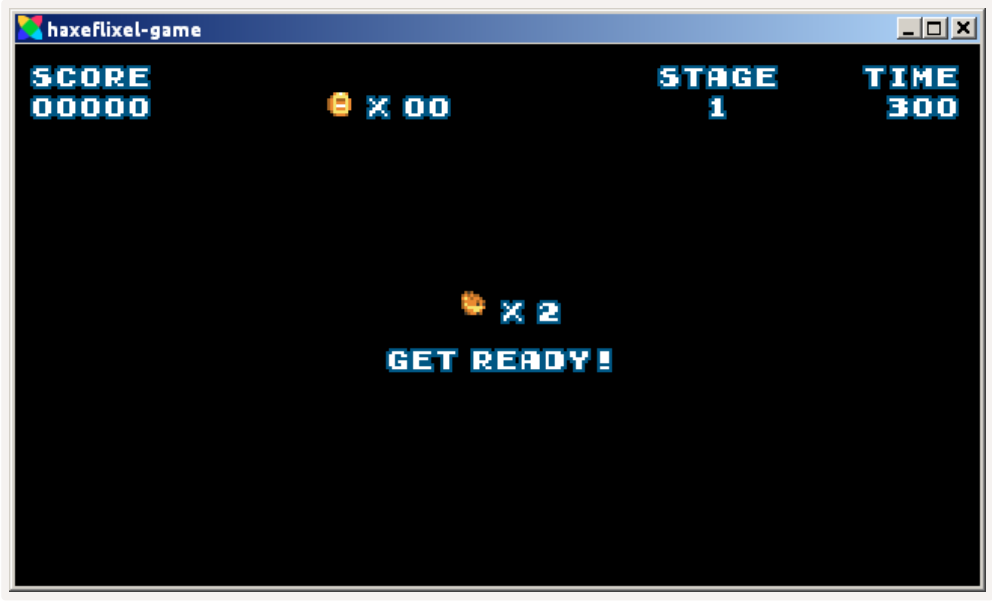
The `cameras` variable of objects simply sets on which cameras the object has to be displayed. It's possible to display an object on multiple cameras.

You'll notice we moved the `scrollFactor` set from `create()` to this function, since we'll call both one after another it makes no sense to use a extra `forEach()`.

Let's call this function in `PlayState.hx`, after the `HUD` is created and pass the `hudCamera` as argument:

```
_hud = new HUD();
_hud.setCamera(_hudCamera);
```

Run the game now, and the `HUD` will be displayed both on the `IntroSubState` and the `PlayState`.



In the next chapter we will keep working with states, and add a main menu to the game. We'll also see how to save scores permanently in-between games.

Menus & Saving

In this chapter we will create an actual main menu for our game, and go back to it after game over instead of quitting the game abruptly. We'll also learn how to save and load data, and save the high score value in-between game sessions.

Main Menu in `MenuState` class

When we created a new template project, `flixel-tools` created a `MenuState.hx` for us. It's basically empty, but we can modify it to host our main menu state. Open it and change it to this:

```
package states;

import flixel.FlxG;
import flixel.FlxObject;
import flixel.FlxSprite;
import flixel.FlxState;
import flixel.text.FlxText;
import flixel.util.FlxColor;
import flixel.math.FlxPoint;
import flixel.math.FlxMath;

import flash.system.System;

class MenuState extends FlxState
{
    static inline var OFFSET:Int = 4;

    private var _textScore:FlxText;
    private var _bg:FlxSprite;
    private var _cursor:FlxSprite;
    private var _selected:Int = 0;

    private static var _menuEntries:Array<String> = ["NEW GAME", "EXIT"];
    private static var _menuPos:FlxPoint = new FlxPoint(132, 112);
    private static var _menuSpacing:Int = 16;
```

We define a `_bg` sprite for the background image and a `_cursor` one for the menu cursor icon. The variable `_selected` which tell which option we are currently selecting in the menu.

`_menuEntries` is an array containing the options we want to display in the menu. We'll use this to dynamically generate `FlxText` on the fly, instead of pre-defining then.

`_menuPos` is a vector holding the position we want to draw the menu at, and `_menuSpacing` the vertical space between menu options.

Onto the `create()` method:

```
override public function create():Void
{
 _bg = new FlxSprite(0, 0);
 _bg.loadGraphic(AssetPaths.title__png, false, 320, 180);

 _cursor = new FlxSprite(_menuPos.x - _menuSpacing, _menuPos.y);
 _cursor.loadGraphic(AssetPaths.hud__png, true, 8, 8);
   _cursor.animation.add("cursor", [1]);
   _cursor.animation.play("cursor");
   _cursor.offset.set(0, -4);

   add(_bg);
   add(_cursor);

   for (i in 0..._menuEntries.length)
   {
    var entry:FlxText = new FlxText(_menuPos.x,
            _menuPos.y + _menuSpacing * i);
    entry.text = _menuEntries[i];
    add(entry);
   }

   forEachOfType(FlxText, function(member)
   {
       member.setFormat(AssetPaths.pixel_font__ttf, 8, FlxColor.WHITE,
                        FlxTextBorderStyle.OUTLINE, 0xff005784);
   });

 super.create();
}
```

Copy the file `title.png` from `books-assets/images` into the `assets/images/` folder in your project. This is the image file we'll use for the title screen, passing it as a graphic for the `_bg` `FlxSprite`.

After initializing and adding our `_bg` and `_cursor` objects, we run a `for()` loop: for each option in our `_menuEntries` array, we'll create a `FlxText` holding that option's text and move it to the correct menu position, offsetting its `y` value based on the option count.

Afterwards, we change the font style for all the `FlxText` (we are pretty familiar with this at this point).

Finally, our `update()` function:

```
 override public function update(elapsed:Float):Void
 {
  super.update(elapsed);

  if (FlxG.keys.justPressed.UP)
   _selected -= 1;

  if (FlxG.keys.justPressed.DOWN)
   _selected += 1;

  _selected = FlxMath.wrap(_selected, 0, _menuEntries.length - 1);

  _cursor.y = _menuPos.y + _menuSpacing * _selected;

  if (FlxG.keys.justPressed.ENTER)
  {
   switch (_selected) {
    case 0: FlxG.switchState(new PlayState());
    case 1: System.exit(0);
   }
  }
 }
} // end of class
```

Here we check if the user presses `UP` or `DOWN`, and move the cursor accordingly.

Afterwards we use the `wrap()` function of `FlxMath` to keep the `_selected` value between 0 and the amount of choices we've got in the menu, to make sure it doesn't move "out of range".

Once the user presses `ENTER`, we check which option is currently selected. If `NEW GAME` is selected, we call `FlxG.switchState()`. This function takes as argument the state you want to switch to, in this case `PlayState`.

Note that `PlayState` is an object itself (remember that a state is nothing more than a glorified `FlxGroup`!), and in this case needs to be constructed with `new` since we haven't defined one before.

To see our new, shiny menu, we need to change the third argument of the `addChild()` function in `Main.hx`:

```
addChild(new FlxGame(320, 180, MenuState));
```

Run the game now and you'll start in the main menu, being able to change options with the arrow keys and selecting one with the Enter button.

Now that we have a working menu, we can change the code so that after a game over the user is sent back to the main menu. More specifically in `IntroSubState`, in the `FlxTimer` being called at the end of `create()`:

```
new FlxTimer().start(_waitToDisappear, function(_)
 {
  if (_gameOver)
  {
   Reg.lives = 2;
   FlxG.switchState(new MenuState());
  }
  else
   close();
 }, 1);
```

Make sure to set `Reg.lives` back to `2` after a game over or when the player starts the game again, they will still have zero lives and go straight to the game over screen!

Saving & loading the high score

We want to keep track of the score in-between game sessions, in order to show the player their highest score and records.

This will require saving data to disk, and reading it back. This is a very common functionality for any kind of game.

Luckily, HaxeFlixel provides a very easy way to manage the saving and loading of the game's data with the `FlxSave` class.

Let's import this class and add two methods to save and load the score to `Reg.hx`:

```
import flixel.util.FlxSave;
```

```
inline static private var SAVE_DATA:String = "HAXEFLIXELGAME";

static public var save:FlxSave;

static public function saveScore():Void
{
 save = new FlxSave();

 if (save.bind(SAVE_DATA))
 {
  if ((save.data.score == null) || (save.data.score < Reg.score))
   save.data.score = Reg.score;
 }

 save.flush();
}

static public function loadScore():Int
{
 save = new FlxSave();

 if (save.bind(SAVE_DATA))
 {
  if ((save.data != null) && (save.data.score != null))
   return save.data.score;
 }

 return 0;
}
```

The `FlxSave` class is defined and initialized like every other object, but it needs to be bound to a specified "save slot". We can do this with the `bind()` method. This method takes a unique string identifier.

In our case we created a static variable called `SAVE_DATA` we use as an identifier.

The `bind()` method returns `true` upon a successful bind, and the data of the `FlxSave` object will be saved to that slot.

If you want to use multiple save slots in your game, all you need to do is define a series of strings to identify each slot and `bind()` to them.

We can save any basic data type into the `data` structure of the `FlxSave` object. In the `saveScore()` function, we first check if we saved a parameter called `score`. If not (or if we did, but it's lower than the current game score), we save the current game score into `data.score`.

After data has been written in the `FlxSave` object, we need to call `flush()` to immediately write this data to disk, or the data might get lost. This is a necessity only on non-Flash targets, on Flash the data is always saved when the application closes.

In the `loadScore()` function we bind the `FlxSave` object to our `SAVE_DATA` slot, and then look for the `score` parameter inside its `data`. If found, we return this value - if not, it means that no score has been saved yet so we just return `0`.

We will call `saveScore()` right after game over in order to save that game score. Right before we go back to the main menu in the game over scenario of `IntroSubState.hx`:

```
new FlxTimer().start(_waitToDisappear, function(_)
 {
  if (_gameOver)
  {
   Reg.saveScore();
   FlxG.switchState(new MenuState());
  }
  else
  ...
```

Let's now add a `FlxText` in the main menu to display the high score. In `MenuState.hx`:

```
class MenuState extends FlxState
{
 public static var OFFSET:Int = 4;

 private var _textScore:FlxText;
 ...

 override public function create():Void
 {
  ...
  _textScore = new FlxText(OFFSET, OFFSET, 0);
        _textScore.text = "HIGH\n" + StringTools.lpad(
              Std.string(Reg.loadScore()), "0", 5
              );

        add(_bg);
        add(_textScore);
        add(_cursor);
        ...
```

We created a new `FlxText` and set its text to display the value returned by `Reg.loadScore()`. Remember to `add()` after the `_bg` or else it will be displayed underneath.

Run the game and the high score should be displayed on the upper left corner of the main menu screen. It will be zero the first game, but run through the game a few times and it will display your highest score, even if you quit and run the game again!

In the next chapter we'll go back working on game objects, and add a special block which will reveal a bonus item when hit by the player from the bottom.

Extra Reading

- HaxeFlixel Handbook: FlxSave

Bonus Blocks

In this chapter we'll add a bonus block which will release a coin (and some other goodies in the future chapters) when hit from the bottom by the player.

The `BonusBlock` class

Create a new file called `BonusBlock.hx` inside `objects`:

```
package objects;

import flixel.FlxObject;
import flixel.FlxSprite;
import flixel.FlxG;
import flixel.tweens.FlxTween;
import flixel.tweens.FlxEase;

class BonusBlock extends FlxSprite
{
 private var _empty:Bool = false;

 public function new(x:Float, y:Float)
 {
  super(x, y);
  solid = true;
  immovable = true;

  loadGraphic(AssetPaths.items__png, true, 16, 16);
  animation.add("idle", [7]);
  animation.add("empty", [8]);
  animation.play("idle");
 }

 public function hit(player:Player)
 {
  FlxObject.separate(this, player);

  if (isTouching(FlxObject.DOWN))
   trace("Collided with a Bonus Block!");
 }

}
```

We define the class alongside an `_empty` variable which tells us whether the block still contains a bonus, or if it has already been hit by the player, and is therefore empty.

We add the graphics and we also make the object `solid` and `immovable`, so it won't move when colliding with the player.

The `hit()` method will be called upon collision with the player. It will be called in a `overlap()` check

within `PlayState`. `FlxG.overlap()` allows objects to overlap when colliding, but since this is a solid block we don't want this behavior. Therefore we immediately call `FlxObject.separate()` to avoid the two objects overlapping.

Remember that `FlxG.collide()` is nothing more than a `FlxG.overlap()` which internally calls `FlxObject.separate()` as well!

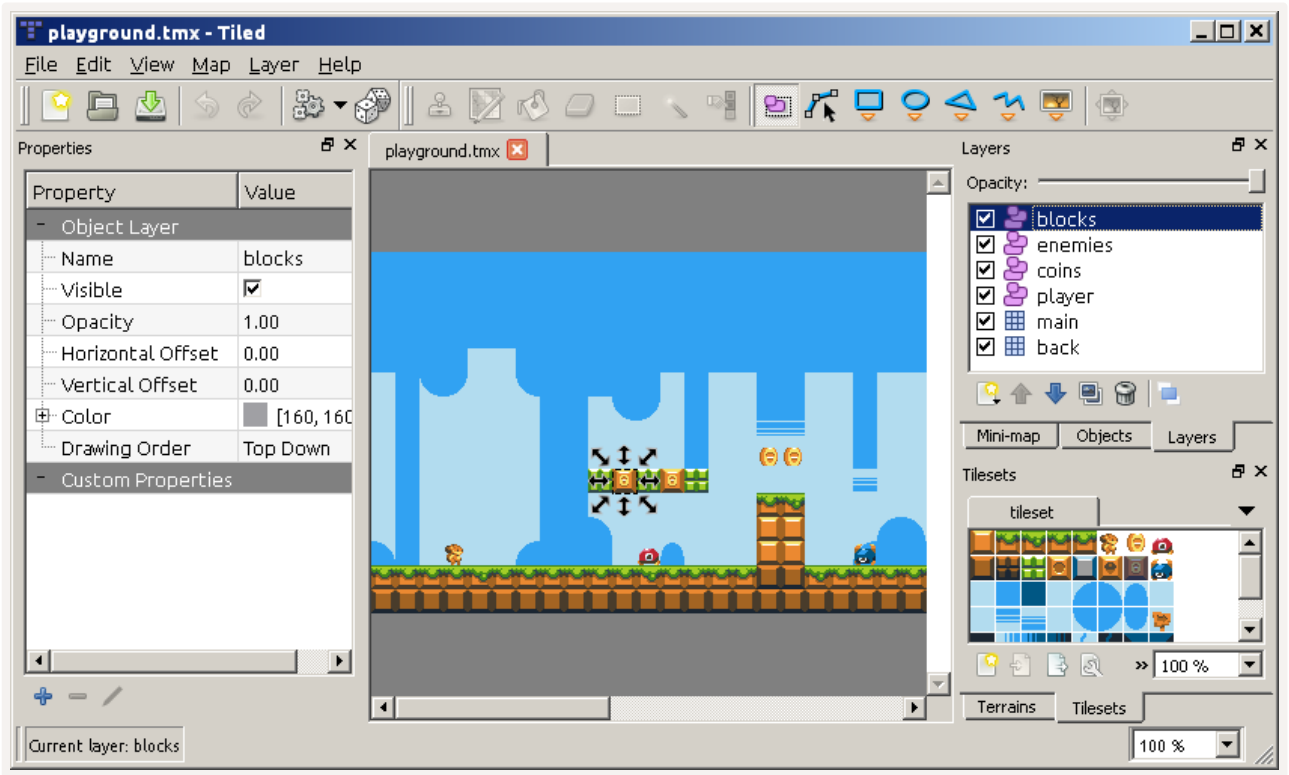
Right now, this `hit()` function is a simple placeholder, which prints a message to the console just to confirm that our collision logic is working. The `trace()` function can be used to display useful messages for debugging purposes.

Note how we print the message only when the block is hit from below - if we skipped the `if (isTouching(FlxObject.DOWN))` check, it would repeatedly print that message if the player was to walk on the block as well, as that counts as a collision too.

We'll leave the class like this for now and add the `BonusBlock` to both `PlayState` and `LevelLoader` to make sure it loads properly in our level first.

I'll expand on the rest of the block functionality later, since it will involve some important new HaxeFlixel concepts and I would like to not skim through them.

Let's open our level in Tiled, add a new object layer called "blocks" and add a few objects in there wherever you want the Bonus Blocks to appear.



Then, in `LevelLoader.hx`, let's add the logic for loading those blocks object in the same way as the other `TiledObject`:

```
// Loading bonus blocks
for (block in getLevelObjects(tiledMap, "blocks"))
 state.blocks.add(new BonusBlock(block.x, block.y - 16));
```

In `PlayState.hx`, let's define and add a new group for the blocks, alongside with the usual helper function:

```
...
private var blocks(default, null):FlxTypedGroup<FlxSprite>;
...
blocks = new FlxTypedGroup<FlxSprite>();
...
```

Now we'll need to think about how we'll be adding this block group to the level.

Blocks are a particular case of objects, since they have to act as an "Item" which the player can interact with, but also as part of the "map", since they can be used as platforms and be walked on by enemies and such.

We could implement this by going through every object / group which collides with `map` and making it collide with `blocks` as well.

However, this would generate a fair amount of extra code. And if we were to add more special objects like this in the future, we'd have to duplicate even more code for them.

A smarter solution is to use nested groups again: make a new group called `_terrain`, add both `map` and `blocks` to it, and use it to calculate the collisions for every object.

Nobody is stopping you from adding a `FlxTilemap` to a group!

Define and initialize a new `FlxGroup` called `_terrain`:

```
private var _terrain:FlxGroup;
...
_terrain = new FlxGroup();
...
```

Then, when you're adding the game objects to the states, add both `map` and `blocks` to `_terrain`.

```
...
LevelLoader.loadLevel(this, "playground");

add(player);
_entities.add(items);
_entities.add(blocks);
_entities.add(enemies);
add(_entities);
_terrain.add(map);
_terrain.add(blocks);
add(_hud);
...
```

Make sure to do so after calling `LevelLoader.loadLevel()`, or else the `map` object won't be initialized.

You'll notice that while we're adding `map` and `blocks` to the `_terrain` group, we are not actually adding `_terrain` to the state. Why is that?

Because both `map` and `blocks` have already being added to the state: `map` is added by `LevelLoader` after loading the Tiled map, while `blocks` is part of `_entities` and therefore added with it.

Afterwards, we rework the collision logic to work with the `_terrain` group in the `update()` function:

```
...
if (player.alive)
{
 FlxG.overlap(_entities, player, collideEntities);
 FlxG.collide(_terrain, player);
}

FlxG.collide(_terrain, _entities);
FlxG.collide(enemies, enemies);
...
```

It's important to test the Player's collision against the `_entities` group **first**, as we want the collision with the Bonus Block to be triggered in this case to callback to `collideEntities()`.

If an eventual block collision is detected when interacting with the `_terrain` group, the `collideEntities()` function would not be called.

Finally, add the Bonus Block case to the `collideEntities()` function:

```
if (Std.is(entity, BonusBlock))
 (cast entity).hit(player);
```

Make sure you added at least one bonus block in the Tiled level, and run the game. When you hit the block, your console should display the message we defined earlier in the `hit()` function.

Moreover, if you add some enemies on top of the Bonus Blocks they should be able to walk on them, showing that our nested groups are actually working.

Hitting the Block & FlxTween

Let's go back to `BonusBlock.hx` and finish our implementation. Add those functions:

```
private function empty(_):Void
{
 _empty = true;
 animation.play("empty");
}

public function hit(player:Player)
{
 FlxObject.separate(this, player);

 if (!_empty && (isTouching(FlxObject.DOWN)))
 {
  FlxTween.tween( this,
      {y: y - 4},
      0.05,
      {onComplete: empty} )
 }
}
```

The `empty()` method is a simple function which sets the variable `_empty` to `true` and changes the object's graphics to an empty block.

`empty()` will be invoked as a callback from a `FlxTween` (more on `FlxTween` in a few lines!). Usually would take a `tween:FlxTween` as an argument, but since we won't be doing anything with the tween itself we name it `_` to indicate that this argument is unused.

In the `hit()` method we'll know then if the object is not `_empty`, and if the collision has happened on

the lower end. If so, we want the block to bounce slightly and change to an empty block (we'll add the coin later).

To achieve this bouncing effect, we use the `FlxTween` functionality:

> A tween gradually fades a variable to another value across the period of time, easing at different speed. We could have a tween which change a value very quickly at the beginning and then slows down as the variable approached its target value.

- We first call `FlxTween.tween()` to start the tween and pass `this` as an argument, indicating that we want to tween a variable of the block object itself.
- We pass the set of variables we want to tween and the target value we want them to reach: in this case we want to subtract `4` from the `y` variable: `{y: y - 4}`. We can pass multiple variable at once like this: `{a: targetA, b: targetB}`
- Then, we specify in how many seconds the tween should be completed: `0.05`.
- Finally, we pass a set of options, which includes the function to be called once the Tween is over (`onComplete`). We set this to the function called `empty`.

> This fourth argument allows you to pass several parameters to customize speed and behavior of your Tween. those are:

> `type`: choose one of these:

> - `FlxTween.ONESHOT`: runs only once and destroys itself once it's finished;
> - `FlxTween.PERSIST`: stops when it finishes. Unlike `ONESHOT`, this type of tween doesn't get destroyed on finish - this means you can keep a reference to this tween and call `start()` whenever you need it. This does not work with `ONESHOT`;
> - `FlxTween.LOOPING`: restarts immediately when it finishes;
> - `FlxTween.PINGPONG`: plays tween once, and then again backwards. This is like `LOOPING`, but every second execution is in reverse direction;
> - `FlxTween.BACKWARD`: plays tween in reverse direction.

> `onComplete`: a callback function, which is called once the tween has finished. This is called every time the tween has finished one execution, therefore with `LOOPING` and `PINGPONG` will be executed multiple times. The function takes a `FlxTween` and returns nothing.

> `ease`: an optional easing function. Easing makes the tween motion smoother, varying acceleration and deceleration slightly. The `FlxEase` class provides many different kind of easing functions ready to be used:

> - `backIn`
> - `bounceIn`
> - `circIn`
> - `cubeIn`
> - `elasticIn`
> - `expoIn`

- `quadIn`
- `quartIn`
- `quintIn`
- `sineIn`

Each of those eases has a `In`, `Out` or `InOut` implementation based of which direction of the motion you want to apply the easing to (for example we can use `sineIn`, `sineOut` or `sineInOut`).

Back to the tween's parameters - `startDelay`: time to wait before starting this tween, in seconds.

`loopDelay`: time to wait before this tween is repeated, in seconds. This only applies to `LOOPING` and `PINGPONG`.

Test the game now, and upon collision the block will quickly bounce upwards. Problem is, it will stay there and not come down!

We can call `.then()` on a tween to chain multiple tweens together, or `.wait()` to wait a set amount of time. Let's implement another tween which takes the block back to its original position after the first tween, and move the `empty()` callback to the second tween's `onComplete`:

```
override public function hit(player:Player)
{
 FlxObject.separate(this, player);

 if (!_empty && (isTouching(FlxObject.DOWN)))
 {
  FlxTween.tween(this, {y: y - 4}, 0.05)
    .wait(0.05)
    .then(FlxTween.tween(this, {y: y}, 0.05,
            {onComplete: empty}));
 }
}
```

Test the game, and upon collision the block should quickly bounce up and then come down again, becoming an empty block once settled.

Creating a Coin

Looking good! Let's wrap this up by making a coin coming out of the block when hit. Instead of creating a new type of coin object for this, we'll modify the existing `Coin` to allow this behavior. Add this function to `Coin.hx`:

```
public function setFromBlock()
{
 solid = false;
 acceleration.y = 420;
 velocity.y = -90;
 new FlxTimer().start(0.3, function(_) {
   collect();
  } , 1);
}
```

When this function is called, `solid` is set to `false` to avoid collisions, and the `Coin` is given some gravity and vertical speed to make it look like it's springing out of the block. Then, after `0.3` seconds, its `collect()` function is called, raising the score and killing the object.

Now, make a `createItem()` function in `BonusBlock.hx`:

```
private function createItem(_)
{
 var _coin:Coin = new Coin(Std.int(x), Std.int(y - 16));
 _coin.setFromBlock();
 Reg.PS.items.add(_coin);
}
```

This function will define a new `Coin` object, call its `setFromBlock()` function and add it to the current `PlayState` (remember the `Reg.PS` reference we defined in the refactor chapter?).

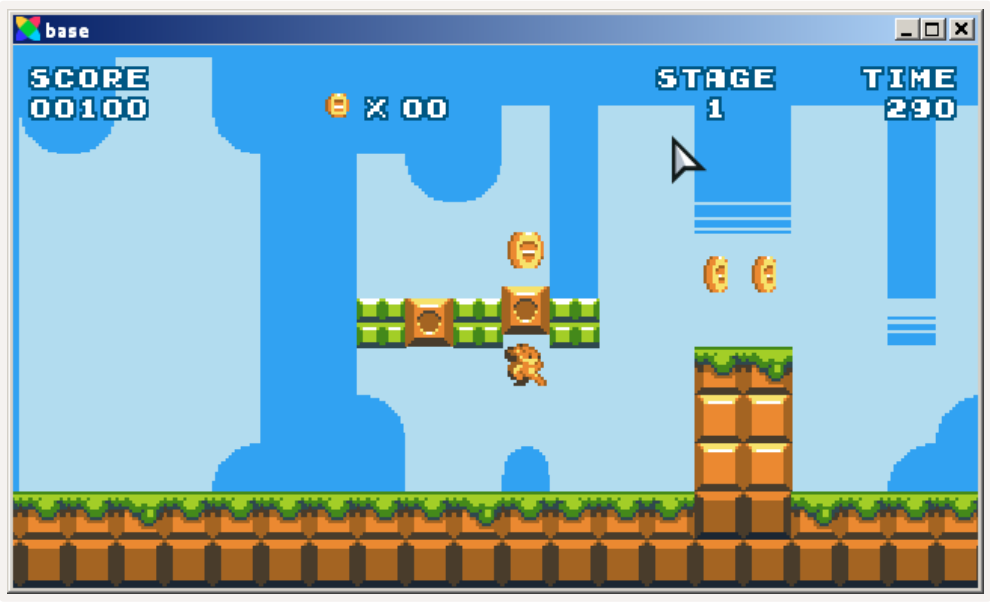
As this is a tween callback function Like `empty()`, it takes `tween:FlxTween` as an argument and returns nothing.

`Std.int()` is used to turn a float decimal value into an integer. We do this because we want to create the `Coin` at the `BonusBlock`'s coordinates, and while `x` and `y` are `Float`, the `new()` function in `Coin` requires integer values.

Let's call this function in the `onComplete` of the first bouncing tween:

```
FlxTween.tween(this, {y: y - 4}, 0.05, {onComplete: createItem})
  .wait(0.05)
  .then(FlxTween.tween(this, {y: y}, 0.05,
                  {onComplete: empty}));
```

Run the game again and hit the block: a coin should spring out. The tweens add a lot of personality to the block!



alt text

In the next chapter we'll add another item coming out of the block: a power-up bonus which will allow the player to take an extra hit from an enemy before dying.

Extra Reading

- HaxeFlixel Handbook: FlxTween
- HaxeFlixel Demos: FlxTween

Power-Up

In this chapter we'll create a power-up bonus item which will make our player bigger, stronger, better, faster (OK, maybe not faster)!

The `PowerUp` Class

Let's start by creating the Power Up object. Inside the `objects` folder, create a new file called `PowerUp.hx`:

```
package objects;

import flixel.FlxObject;
import flixel.FlxSprite;
import flixel.FlxG;

class PowerUp extends FlxSprite
{
 private static inline var MOVE_SPEED:Int = 40;
 private static inline var GRAVITY:Int = 420;

 public var direction:Int = -1;
 private var _moving:Bool = false;

 public function new(x:Float, y:Float)
 {
  super(x, y);

  loadGraphic(AssetPaths.items__png, true, 16, 16);
  animation.add("idle", [5]);
  animation.play("idle");

  velocity.y = -16;
 }
```

```
override public function update(elapsed:Float)
{
 if (!Reg.pause)
       {
  if (!_moving && (Math.floor(y) % 16 == 0))
  {
   velocity.y = 0;
   acceleration.y = GRAVITY;
   _moving = true;
  }

  if (_moving)
        velocity.x = direction * MOVE_SPEED;

  if (justTouched(FlxObject.WALL))
   direction = -direction;
       }

       super.update(elapsed);
}

public function collect(player:Player)
{
 kill();
 trace("Obtained Power Up!");
}
}
```

The `PowerUp` will be generated from a Bonus Block, we want it to slowly raise from the block and as soon as it is level with it, start moving horizontally.

We'll use the variable `_moving` to keep track of whether the bonus is moving or not. Initially, this will be set to `false`.

Setting `velocity.y` to a negative value when the `PowerUp` is created will make it move upwards. But how do we know when to stop it and make it move horizontally?

We'll do this check on the `update()` function: first we make sure that `_moving` is indeed false, then we check with the modulus operation (`%`) if the remainder after the division of the integer of the `y` coordinate by `16` is zero.

Since our game works on a grid of `16` by `16` pixel, if this operation returns zero it means that the object is perfectly aligned with the grid, meaning that the powerup has moved one full tile upwards after being generated by the block position.

And this is exactly when we want to stop its vertical movement, set its `_moving` to true, and set its `acceleration.y` to the chosen `GRAVITY` value so it is able to fall down in case it moves past the edge of the block.

Once `_moving`, the PowerUp's `velocity.x` will increase according to its direction, and will reverse motion when hitting a `FlxObject.WALL`.

We also have a `hit()` method which we'll call upon collision with the player, which contains a simple placeholder message for now.

The Power-Up Block

The `PowerUp` will be created when the player hits a special `BonusBlock`.

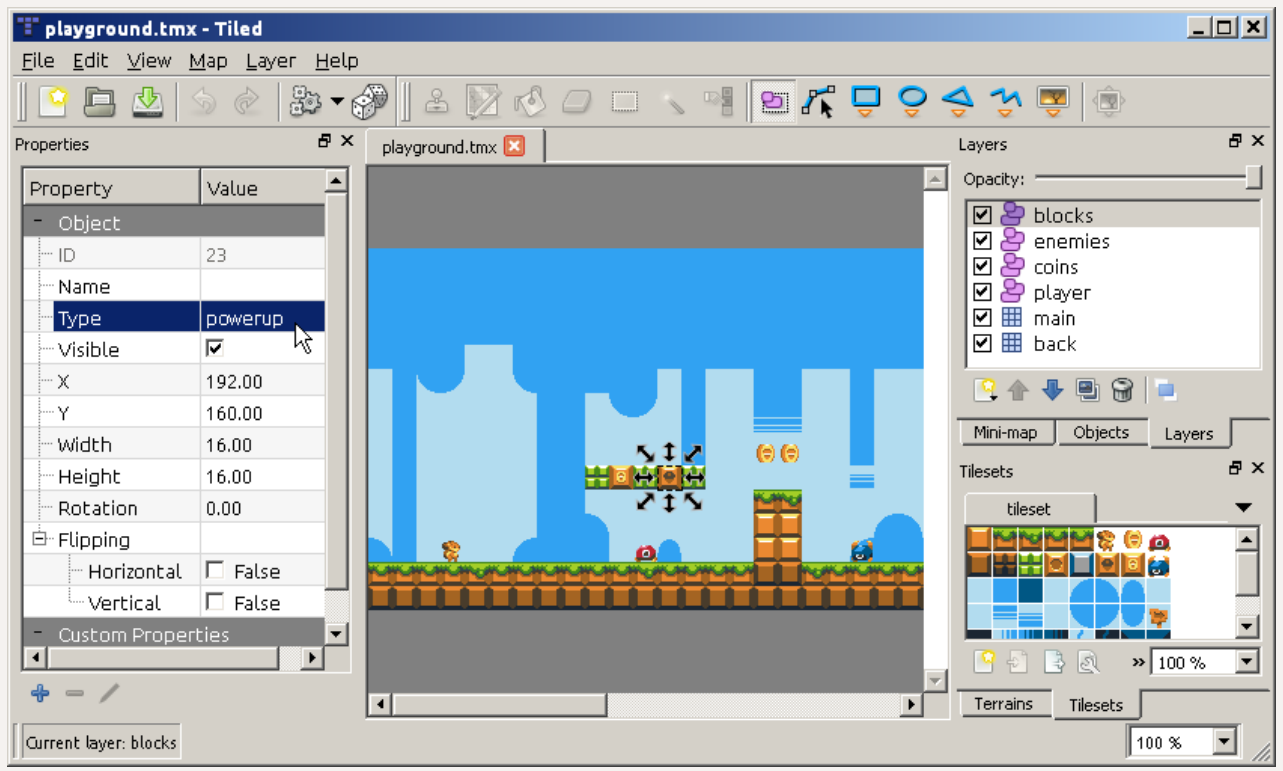
Instead of creating a separate `BonusBlock` class which will just create `PowerUp`, we can make use of the `type` parameter in the tiled map, like we did when creating extra enemies.

In `BonusBlock.hx`, define a new public string variable called `content`:

```
class BonusBlock extends FlxSprite
{
 public var content:String;

 private var _empty:Bool = false;
 ...
```

We'll use this to keep track of which item is contained inside the bonus block, and will read this value from the `type` property defined in Tiled.

Open your Tiled level, and create a new object inside the "blocks" object layer. This time, write `powerup` in its "type" property field.



In `LevelLoader.hx`, let's revise the block loading logic:

```
// Load bonus blocks
for (block in getLevelObjects(tiledMap, "blocks"))
{
 var blockToAdd = new BonusBlock(block.x, block.y - 16);
 blockToAdd.content = block.type;
 state.blocks.add(blockToAdd);
}
```

This way, for each block `TileObject`, we first create a relative `BonusBlock`, set its `content` variable to the `TiledObject` `type`, and finally add it to the map.

In `BonusBlock.hx`, let's now consider this new `content` variable when calling `createItem()`:

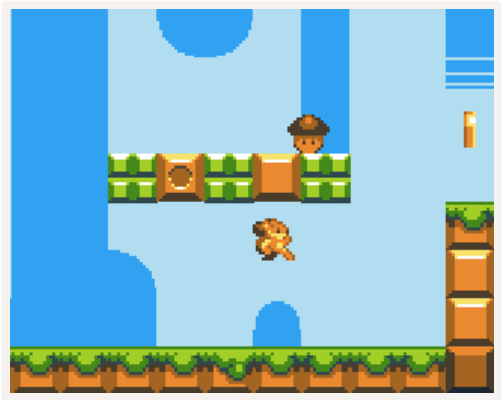
```
private function createItem(_)
{
 switch (content)
 {
  default:
   var _coin:Coin = new Coin(Std.int(x), Std.int(y - 16));
   _coin.setFromBlock();
   Reg.PS.items.add(_coin);

  case "powerup":
   var _pwrUp:PowerUp = new PowerUp(Std.int(x), Std.int(y));
   Reg.PS.items.add(_pwrUp);
 }
}
```

If `content` is set to `"powerup"` we'll create a `PowerUp` object. In any other case, we'll create a `Coin` instead. You can see how easy it is to eventually add extra items to our Bonus Block!

Finally, let's take care of the collision case of the `PowerUp` against the `Player` in the `update()` function of `PlayState.hx`:

```
function collideEntities(entity:FlxSprite, player:Player):Void
{
 ...
 if (Std.is(entity, PowerUp))
  (cast entity).collect(player);
 ...
}
```

Test the game, hit your `BonusBlock` of type `powerup` and try to catch the PowerUp which will come out of it! The console should display the placeholder message. Now we're all set to add the actual PowerUp function!



Powering-up the Player

Now let's proceed with the actual power up of the player! When touching a `PowerUp` bonus, the player character will grow into a bigger version of itself and will be able to sustain an extra hit by the enemies.

In `Player.hx` Define a new variable called `_stopAnimations`:

```
private var _stopAnimations = false;
```

We'll use this to turn on and off the player animations and have better control over them. In `update()`:

```
override public function update(elapsed:Float)
{
 ...
 if (!_stopAnimations)
  animate();
 ...
}
```

We'll also use the `health` variable to keep track of the Player's PowerUp status, `0` for its basic form and `1` for the powered up form.

The `health` variable is automatically defined in all `FlxSprite` objects, so there's no need to define it manually.

First of all, we need some new graphics for the powered up player. Download the file `player_both.png` and place it in `assets/images`. This is an updated sprite sheet for the Player object which includes the original player graphics, plus a powered-up form.

Change the `loadGraphic()` function in `Player.hx` to use this new image:

```
loadGraphic(AssetPaths.player_both__png, true, 16, 16);
```

The reason we are using only one sprite sheet is to blend animations - when the player powers up, we want to flash between the old and new form - and avoid having to repeat graphics.

However because of this, a new single player sprite is taller than before, `16x32` compared to the previous `16x16`.

When we load the small version of the player, we'll need to adjust it's hitbox and offset so that the empty space at the top is simply ignored:

Let's write a new function that will make those adjustments automatically and load the correct animations based on the player's `health`:

```
private function reloadGraphics()
{
 loadGraphic(AssetPaths.player_both__png, true, 16, 32);
 switch (health)
 {
  case 0:
   setSize(8, 12);
   offset.set(4, 20);
   animation.add("idle", [0]);
   animation.add("walk", [1, 2, 3, 2], 12);
   animation.add("skid", [4]);
   animation.add("jump", [5]);
   animation.add("fall", [5]);

  case 1:
   setSize(8, 24);
   offset.set(4, 8);
   animation.add("idle", [7]);
   animation.add("walk", [8, 9, 10, 9], 12);
   animation.add("skid", [11]);
   animation.add("jump", [12]);
   animation.add("fall", [12]);
 }

 animation.add("dead", [6]);
 animation.add("transform", [5, 12], 24);
}
```

You'll notice how the `dead` and `transform` animations are common to both forms. `transform` will simply quickly animate between the old sprite and the new sprite, creating a flashing growing-up effect.

Now let's create a function called `powerUp()`:

```
public function powerUp()
{
 if (health >= 1)
  return;

 Reg.pause = true;
 _stopAnimations = true;
 animation.play("transform");
 velocity.set(0,0);
 acceleration.set(0,0);

 new FlxTimer().start(1.0, function(_)
 {
  health++;
  reloadGraphics();
  Reg.pause = false;
  _stopAnimations = false;
 });
}
```

- We'll run this function only if the player `health` is at zero.
- We'll increase the `health` by one, and call `reloadGraphics()` to set the correct graphics for the powered up version.
- We then set `Reg.pause` to `true` to stop the game and start the `transform` animation, setting `_stopAnimations` to `true` so that it doesn't get overridden by the other animations.
- We also call `velocity.set(0,0)` and `acceleration.set(0,0)` to stop the player movement, since extra acceleration might cause him to keep moving for a short bit even after `Reg.pause` is set to `true`.
- A `FlxTimer` is set so that after one second, both `Reg.pause` and `_stopAnimations` are set back to `false` to allow the player to move again and restore the standard animation control.

Moreover, let's set the `health` to `0` on player creation so that the player does not start in a powered up state:

```
public function new()
{
 super();
 health = 0;
 reloadGraphics();
 ...
```

Finally, let's modify the `collect()` function in `PowerUp.hx` to call the `powerUp()` function of `Player.hx`. We'll set a `SCORE_VALUE` for the power up as well, so that if the player collects a power up while already being in a powered up state, he will earn a huge amount of points:

```
public function collect(player:Player)
{
 kill();
 if (player.health == 0)
  player.powerUp();
 else
  Reg.score += SCORE_AMOUNT;
}
```

Try and run the game now, and collect a `PowerUp` item. The player should grow!

There's a problem, though: if the player pick up a `PowerUp` object while he's jumping, his jump will stop in mid-air because we are setting both velocity and acceleration to zero.

To fix this, we need to store the `Player` velocity and acceleration in temporary variables before setting them to zero, and then restore them right before un-pausing the game.

Let's import a new class:

```
import flixel.math.FlxPoint;
```

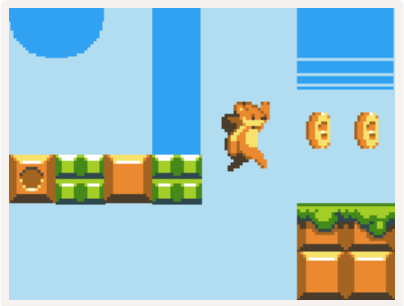
Then, let's modify the `powerUp()` function:

```
 public function powerUp()
 {
  if (health >= 1)
   return;

  var _prevVelocity:FlxPoint = new FlxPoint().copyFrom(velocity);
  var _prevAccel:FlxPoint = new FlxPoint().copyFrom(acceleration);

  Reg.pause = true;
  _stopAnimations = true;
  animation.play("transform");
  velocity.set(0,0);
  acceleration.set(0,0);

  new FlxTimer().start(1.0, function(_)
  {
   health++;
   reloadGraphics();
   y -= 16;

   Reg.pause = false;
   _stopAnimations = false;
   velocity = _prevVelocity;
   acceleration = _prevAccel;
  });
 }
```

`copyFrom()` is a helper function of `FlxPoint` which copies the values from another specified `FlxPoint`.

Run the game again and try to pick up a `PowerUp` object in mid-air: the game will stop to show you the

transform animation, but then the jump should proceed following its original trajectory.



Damaging the Player

Touching an enemy in your powered up form however, will still kill you instantly instead of reverting back to the smaller form, since we haven't introduced a "Power Down" function yet.

This function (let's call it `damage()`) will work in the same way as the `powerUp()`, but reducing the `health` variable instead. If the `health` is at 0 (player is at its base form), `damage()` will call `kill()`.

We also need to be careful about how this function is called. Since collisions are calculated up to 60 times per second, we might end up calling `damage()` several times, in such a quick way that the player will get damaged and killed without time to react!

To fix this, we'll introduce a small time window after being damaged where the player will be temporarily invincible. We'll make the character flicker during this state. Let's create a new boolean variable to identify this:

```
public var flickering:Bool = false;
```

HaxeFlixel provides several functions to manage a sprite's flickering status, and they are contained within the helper class `FlxSpriteUtil`. Let's import it now:

```
import flixel.util.FlxSpriteUtil;
```

Now, let's write a new function called `damage()` in `Player.hx`:

```
public function damage()
{
 if ((FlxSpriteUtil.isFlickering(this)) || (Reg.pause))
  return;

 if (health > 0)
 {
  var _prevVelocity:FlxPoint = new FlxPoint().copyFrom(velocity);
  var _prevAccel:FlxPoint = new FlxPoint().copyFrom(acceleration);

  Reg.pause = true;
  _stopAnimations = true;
  animation.play("transform");
  velocity.set(0,0);
  acceleration.set(0,0);

  new FlxTimer().start(1.0, function(_)
  {
   health--;
   reloadGraphics();
   y += 16;

   FlxSpriteUtil.flicker(this, 2.0, 0.04, true);
   Reg.pause = false;
   _stopAnimations = false;
   velocity = _prevVelocity;
   acceleration = _prevAccel;
  });

 }
 else kill();
}
```

First of all, we want to execute this function only when the character is not flickering and the game is not paused.

Then, we want to proceed with only damaging the player if the `health` variable is larger than zero. If not, it means that the `Player` is at its weakest form and will be killed by this hit - we'll just call `kill()`.

We run through the same operations we used in `powerUp()` (stop the game and the animations, play `transform`, store velocity and acceleration), but this time reducing the `health` variable.

Once the timer is triggered, however, we call `FlxSpriteUtil.flicker()` to start the flickering effect on the player. The `flicker()` function takes the following arguments:

- The object to flicker (in this case, the player itself)
- The total time the flickering effects will run for (2 seconds here)
- The flickering interval (here the player will flicker from invisible to visible every 0.04 seconds)
- The last boolean flag indicates whether to force the graphic to be always visible in the end, usually `True` (unless you don't care if the object ends up invisible)

Now let's make sure that upon collision with the enemy we replace our `player.kill()` calls with `player.damage()` in both `Enemy.hx`...:

```
public function interact(player:Player)
{
 if (alive)
 {
  FlxObject.separateY(this, player);

  if ((player.velocity.y > 0) && (isTouching(FlxObject.UP)))
  {
   kill();
   player.jump();
  }
  else
   player.damage();
 }
}
```

...and `SpikeEnemy.hx`:

```
override public function interact(player:Player)
{
 if (alive)
  player.damage();
}
```

For lethal scenarios, such as falling down the screen or running out of time, we want to keep the `kill()` calls, as those events result in certain death no matter the PowerUp status.

The game is starting to look great! In the next chapter we'll make a proper, longer level, and implement checkpoints.

Extra Reading

- HaxeFlixel Handbook: FlxTween

Exit & Checkpoints

In this chapter we'll implement a checkpoint system: if the player dies after reaching this checkpoint, the game will restart at the checkpoint's location rather than from the beginning of the level. We'll also create exits for the level in order for the player to progress through the game.

Implementing the Checkpoint

Let's make a `checkpointReached` variable to keep track of this in `Reg.hx`:

```
public static var checkpointReached:Bool = false;
```

And in `PlayState.hx`, a public `checkpoint` variable - A `FlxPoint` which will hold the `x` and `y` coordinates of the checkpoint:

```
class PlayState extends FlxState
{
 ...
 public var checkpoint:FlxPoint;
 ...
```

Let's write a new `updateCheckpoint()` function which will check if the player has surpassed the checkpoint and eventually update the `checkpointReached` value:

```
private function updateCheckpoint()
{
 if (checkpoint == null || Reg.checkpointReached)
  return;

 if (player.x >= checkpoint.x)
 {
  trace("Checkpoint Reached");
  Reg.checkpointReached = true;
 }
}
```

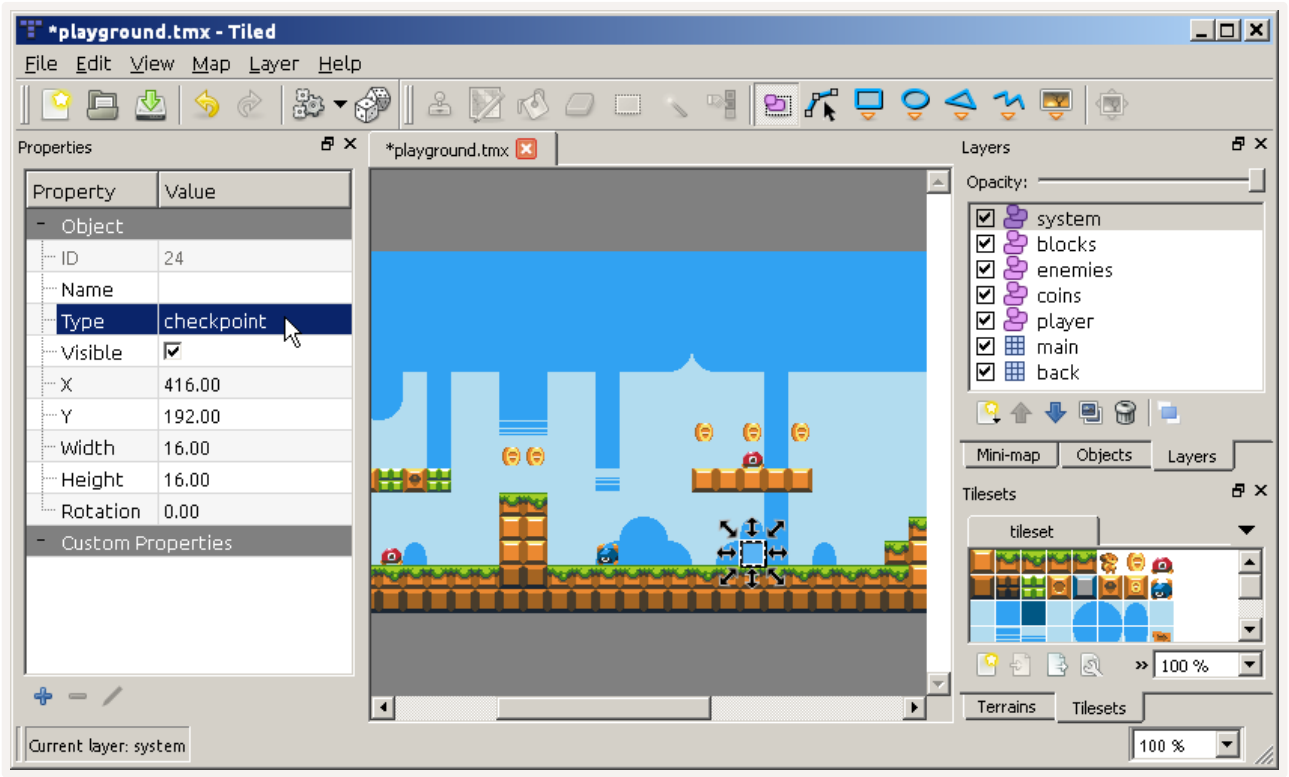
Then, we'll call this function in the `update()` method, right next to `updateTime()`:

```
override public function update(elapsed:Float):Void
{
 ...
 updateTime(elapsed);
 updateCheckpoint();
}
```

Now we need to create a checkpoint in our Tiled level. Open your Tiled level and create a new object layer called "system". In there, create a new object and set its type to "checkpoint". Place this object

wherever you want your player to re-spawn after reaching the checkpoint.

As we don't have a graphic for the checkpoint since it's just a location in space, use on of the drawing tools such as **"Insert Rectangle (R)"** to draw the checkpoint object on the level. You can change the Object layer color as well form the properties window.



Now, let's open `LevelLoader.hx` and within `loadLevel()` let's load this new checkpoint object:

```
// Load checkpoint
for (object in getLevelObjects(tiledMap, "system"))
{
 switch (object.type)
 {
  case "checkpoint":
   state.checkpoint = FlxPoint.get(object.x, object.y - 16);
 }
}
```

We are loading the checkpoint object's coordinate into the `checkpoint` `FlxPoint` we defined in the `PlayState`.

Now we just need to change the position the player is created at based on the value of `Reg.checkpointReached`. Let's modify the player loading logic:

```
var player:TiledObject = getLevelObjects(tiledMap, "player")[0];
var playerPosition:FlxPoint = new FlxPoint();
if (Reg.checkpointReached)
 playerPosition = state.checkpoint;
else
 playerPosition = FlxPoint.set(player.x, player.y - 16);
state.player.setPosition(playerPosition.x, playerPosition.y);
```

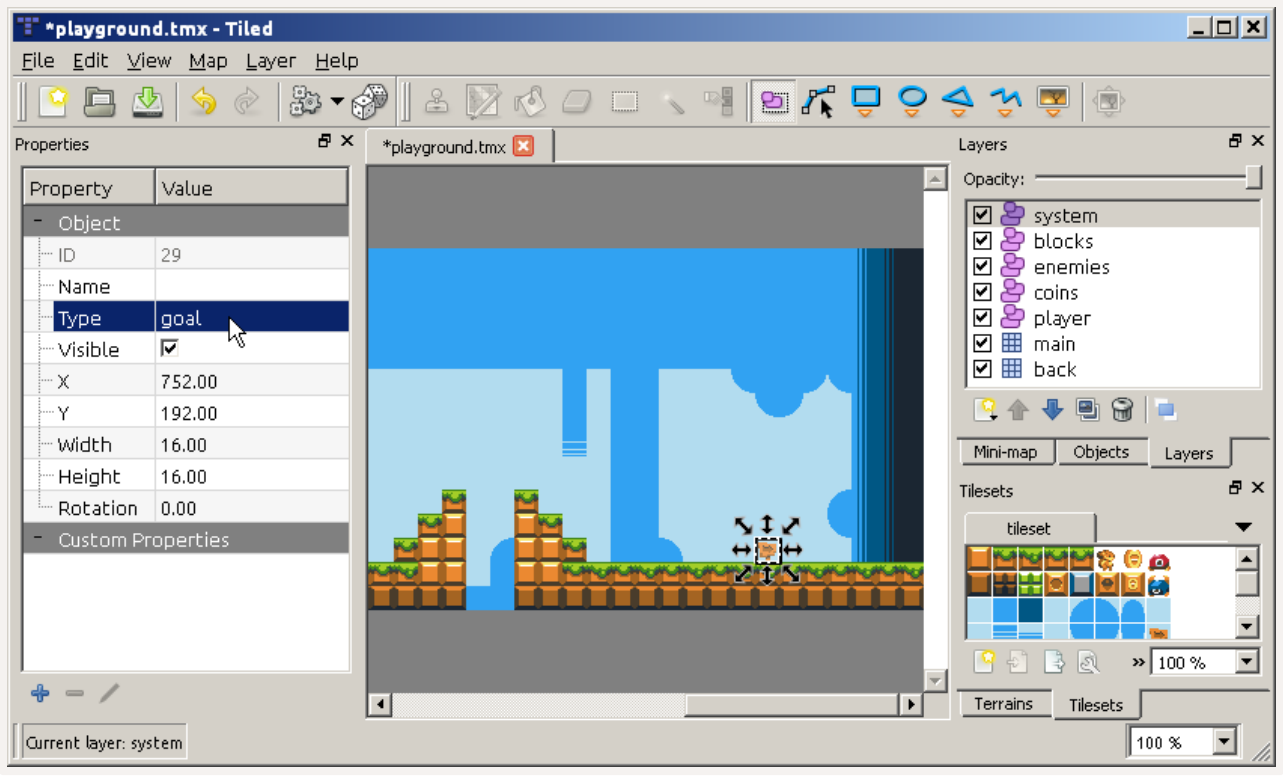
We use a temporary `FlxPoint` variable to hold the player position and change it based on the value of `Reg.checkpointReached`: if `false`, we'll use the original player's starting position; if `true` we'll use the checkpoint position instead.

Run the game and try to reach the checkpoint - it won't have any visible graphics, but the console should display our debug message upon reaching it. At that point, if you die, you should re-spawn in the checkpoint position.

Reaching the end of the level

Let's create a goal object which will represent the end of the level. Touching it will convert the remaining time into score points, and have the player walk off the screen and proceed to the next level.

Open your Tiled level and create a new object in the "system" object layer, and assign its type to "goal". Put it at the end of the level.



Then, create a new file called `Goal.hx` inside `objects` and let's start writing our class:

```
package objects;

import flixel.FlxObject;
import flixel.FlxSprite;
import flixel.FlxG;
import flixel.util.FlxColor;
import flixel.util.FlxTimer;

import objects.Player;

class Goal extends FlxSprite
{
 public var _calculateScore:Bool = false;

 public function new(x:Float, y:Float)
 {
  super(x, y);
  solid = true;
  immovable = true;
  makeGraphic(2, FlxG.height * 2, FlxColor.TRANSPARENT);
 }
```

In our `create()` function, we initialize the goal as a transparent object as tall as the whole screen, so that the player can't accidentally jump over it.

We also define a `_calculateScore` variable which we will use to manage the conversion of time into points.

```
public function reach(player:Player)
{
 solid = false;
 Reg.pause = true;

 player.velocity.y = Player.JUMP_FORCE / 2;
 player.velocity.x = Player.WALK_SPEED;
 player.acceleration.x = 0;
 player.drag.x = player.drag.x / 4;

 new FlxTimer().start(4.0, function(_)
 {
  _calculateScore = true;
  player.drag.x = player.drag.x * 4;
  player.acceleration.x = Player.WALK_SPEED;
 }, 1);
}
```

The `reach()` function will be called when the player overlaps with this object. We change `solid` to `false`, to make the player able to walk through it.

Afterwards we set the player's `acceleration` to zero, and reduce its `drag` value to a quarter of the original value. We also set it's `velocity.x` and `velocity.y` values to simulate a small celebratory jump through the goal.

To do so, we need to change a few of its `static` variables from `private` to `public`, in `Player.hx`:

```
class Player extends FlxSprite
{
 ...
 public static inline var JUMP_FORCE:Int = -280;
 public static inline var WALK_SPEED:Int = 80;
 ...
```

The `drag` value influences how quickly an object decelerates after it stopped moving. This change will make the player stop walking slower after reaching the goal.

The player will stand still for a few seconds, and then a `FlxTimer` will trigger, setting `_calculateScore` to `true` (We'll see how we use this in the `update()` function next) and restore `drag` and `acceleration` to their original value, making the player walk towards the right edge of the screen.

```
 override public function update(elapsed:Float)
 {
        super.update(elapsed);

        if (!_calculateScore)
         return;

     if (Reg.time > 0)
     {
      Reg.time -= 5;
      Reg.score += 50;
     }
     else
     {
      Reg.time = 0;
      _calculateScore = false;
      new FlxTimer().start(2.0, function(_) {
    FlxG.switchState(new MenuState());
   }, 1);
     }
 }
} // end of class
```

Most of our `update()` function will run only when `_calculateScore` is set to `true`. At that point, if there is still time left, we'll decrease `Reg.time` and increase `Reg.score`.

On the other end, if all the seconds remaining have been converted into points, we'll start a short `FlxTimer` of two seconds and then `switchState` back to the menu (We'll add more levels shortly).

Let's modify the `loadLevel()` function in `LevelLoader.hx` to load the goal object we added in Tiled as well:

```
// Load exit & checkpoint
for (object in getLevelObjects(tiledMap, "system"))
{
 switch (object.type)
 {
  case "checkpoint":
   state.checkpoint = FlxPoint.get(object.x, object.y - 16);
  case "goal":
   state.items.add(new Goal(object.x, object.y - 16));
 }
}
```

Moreover, let's add the collision case for the goal in `PlayState.hx`, and make it call its `reach()` method we wrote earlier:

```
function collideEntities(entity:FlxSprite, player:Player):Void
 {
  ...
  if (Std.is(entity, Goal))
   (cast entity).reach(player);
 }
```

Run the game and upon jumping over the goal, the player should stop and start walking off the stage shortly after, while the remaining time gets converted into points.

After that, the game will go back to the main menu. It doesn't really have a choice since we only have one level so far! Let's add some more levels, and make the game go to the next level once the stage is complete. We'll also add a celebratory message when you finish all levels!

Switching levels

Create a few levels in Tiled - let your imagination go wild and add all the elements we used throughout this book. For now, I will make 3 levels. Rename your levels into `level1.tmx`, `level2.tmx` and `level3.tmx`; or feel free to copy mine. Make sure they are located in `assets/data`.

Next, create the `levels` variable in `Reg.hx` as an array containing the names of all our levels, in order:

```
public static var levels:Array<String> =  [
              "level1",
              "level2",
              "level3"
             ];
```

Then in `PlayState.hx`, change the function call to `LevelLoader.loadLevel()` so that instead of loading `playground` it loads the element in the `Reg.levels` corresponding the the `Reg.currentLevel` value:

```
override public function create():Void
{
 ...
 LevelLoader.loadLevel(this, Reg.levels[Reg.currentLevel]);
 ...
```

This way, when `Reg.currentLevel` is 0, the first element of the `Reg.levels` array will be loaded (`level1.tmx`); when it's one, the second element will be loaded (`level2.tmx`), and so on.

We can control which level is loaded by changing the `Reg.currentLevel` variable. Let's create a new function called `nextLevel()` in `PlayState.hx`:

```
public function nextLevel():Void
{
 Reg.checkPointReached = false;
 checkpoint = null;
 Reg.currentLevel++;
 FlxG.resetState();
}
```

This function will simply increase `Reg.currentLevel` by one, reset the checkpoint information and reload the `PlayState` with `FlxG.resetState();`.

This will effectively take the game to the next level, as with `PlayState` restarting, the load `LevelLoader.loadLevel()` will be called again with the newly increased `Reg.currentLevel`.

Now it's only a matter of calling this function from `Goal.hx`, instead of going back to the menu:

```
override public function update(elapsed:Float)
{
    super.update(elapsed);

    if (!_calculateScore)
     return;

 if (Reg.time > 0)
 {
  Reg.time -= 5;
  Reg.score += 50;
 }
 else
 {
  Reg.time = 0;
  _calculateScore = false;
  new FlxTimer().start(2.0, function(_) {
   Reg.PS.nextLevel();
  }, 1);
 }
}
```

Run the game, and once you complete the level, the game should successfully advance to the next one!

If you get to the last level, however, the game will crash as it will try to access the level after the last one, which obviously doesn't exist.

We'll perform a check for that in the `IntroSubState`, and display a celebratory message once the player finishes the last level.

Let's change the `nextLevel()` function first to reset the `PlayState` only when we haven't reached the next level:

```
public function nextLevel():Void
{
 Reg.checkpointReached = false;
 checkpoint = null;
 Reg.currentLevel++;
 if (Reg.currentLevel < Reg.levels.length)
  FlxG.resetState();
 else
  FlxG.switchState(new IntroSubState(FlxColor.BLACK));
}
```

In the opposite case, we'll switch to the `IntroSubState` directly. Then modify `IntroSubState.hx`:

```
class IntroSubState extends FlxSubState
{
 ...
 private var _gameFinished:Bool = false;
 ...

 override public function create():Void
 {
  ...

  if (Reg.currentLevel >= Reg.levels.length)
   _gameFinished = true;

  ...
```

We define a new variable `_gameFinished` which is set to `true` in case the current level count is larger than the total amount of levels we got in the game.

```
...
if (_gameOver)
    {
     _text.text = "Game Over";
     _text.setPosition(0, FlxG.height/2);
    }
    else
    {
 if (_gameFinished)
    {
     _text.text = "Thanks for Playing!";
     _text.setPosition(0, FlxG.height/2);
     _waitToDisappear = 5.0;
    }
    else
    {
  add(_iconLives);
        add(_textLives);
    }
   }
   ...
```

Then, after checking for `_gameOver`, we check for `_gameFinished` as well before starting the game. If `true`, instead of displaying the life count we display a congratulations message, and set the substate to disappear after 5 seconds.

```
  new FlxTimer().start(_waitToDisappear, function(_)
  {
   if (_gameOver || _gameFinished)
   {
    Reg.saveScore();
    FlxG.switchState(new MenuState());
   }
   else
    close();
  }, 1);
 }
} // end of class
```

Finally, when the `FlxTimer` to close the substate is triggered, we change the condition to go back to the main menu if either `_gameOver` or `_gameFinished` condition is true.

Test the game, and try complete the levels of your game (provided you didn't make your levels too hard!). When finishing the last level, it should display the congratulation message and then take you back to the main menu, saving your high score.

Congratulation to you as well, game developer! I can say we now have a game which deserves to be called as such.

In the next chapter we'll add music and sounds to our game.

Sound and Music

Our game works well, but it's very silent. In this chapter we'll add music and sound effects.

Copy the sound effects (.wav) files to the `asset/sounds` folder. We are going to add definitions for those files in the `Project.xml` file, under the `Paths Settings` section:

```
<assets path="assets"/>

<assets path="assets/sounds">
    <sound path="coin.wav" id="coin" />
    <sound path="jump.wav" id="jump" />
 <sound path="coin.wav" id="coin" />
    <sound path="block.wav" id="block" />
    <sound path="defeat.wav" id="defeat" />
    <sound path="death.wav" id="death" />
    <sound path="dying.wav" id="dying" />
    <sound path="damage.wav" id="damage" />
    <sound path="goal.wav" id="goal" />
    <sound path="powerup-appear.wav" id="powerup-appear" />
    <sound path="powerup.wav" id="powerup" />
    <sound path="time-convert.wav" id="time-convert" />
</assets>
```

We give each sound file an `id` property, which we use to reference the sound effect in the code. This way, if we want to replace a sound effect in the future, we can just modify the `path` and leave the same `id`, without the need of changing anything in the code.

HaxeFlixel provides a very simple way to play sound effects, by calling the function `FlxG.sound.play()` and passing the `id` of the sound effect.

`FlxG.sound` is the HaxeFlixel's sound front end, a sort of global controller which manages all the music and sound in the game.

Let's add our first sound effect - for example, when the player picks a coin up. Open `Coin.hx` and modify the `collect()` function (which is called when the player collides with the coin, remember?) to call `FlxG.sound.play()` and pass the appropriate sound effect `id`:

```
public function collect()
{
 FlxG.sound.play("coin");
 Reg.score += SCORE_AMOUNT;
 Reg.coins ++;
 ...
```

Run the game, and pick up a coin - the sound should play just fine - it's as simple as that!

It is also possible to load the file sound file directly, without assigning it an `id`. Using `AssetPaths`, in the same way we do with the images:

```
FlxG.sound.play(AssetPaths.coin__wav)
```

While this could be a good choice for certain kind of games, like procedural ones where the music is influenced by the gameplay, in my opinion using an `id` provides better organization.

Go through the available sounds and change the game's code to play them in appropriate situation. Here are some examples:

- When the player jumps, in `Player.hx`:

```
...
private function move()
{
 ...
 if (velocity.y == 0)
 {
  if (FlxG.keys.pressed.C && isTouching(FlxObject.FLOOR))
  {
   FlxG.sound.play("jump");
   jump();
  }
  ...
```

- When a powerup is picked, in `PowerUp.hx`:

```
public function collect(player:Player) {
  kill();
  FlxG.sound.play("powerup");
  if (player.health == 0)
   player.powerUp();
  else
   Reg.score += SCORE_AMOUNT;
 }
```

- When an enemy is killed, in `Enemy.hx`:

```
...
override public function kill()
{
 alive = false;
 Reg.score += SCORE_AMOUNT;
 FlxG.sound.play("defeat");
 ...
```

Feel free to check the source code for this chapter to see where and what sound effects I added.

Once you're done let's add background music which will play throughout the level. The process is very similar to what we just did for the sound effects - first copy the music assets files to in `assets\music`.

If you're unable to compose music and can't hire an external musician to make the music for your game, there are several resources on the web where you can find free game music, such as Open Game Art. The

sound effects were created using the excellent tool Bfxr.

Requirements for using the file varies from file to file, some of them can be used only for non-commercial projects, while other might require you to list the name of the composer in your credits.

The music used in this example are "Pixelland" & "Loping Sting" by Kevin MacLeod (incompetech.com), Licensed under Creative Commons: By Attribution 3.0 License.

Add the music files definitions in the `Project.xml` file, right under the sound ones:

```
<assets path="assets/music">
    <music path="pixelland.ogg" id="pixelland" />
    <music path="victory.ogg" id="victory" />
</assets>
```

Then, in `IntroSubState.hx`, when the substate closes and we are going back to the `PlayState`, call `FlxG.sound.playMusic()`:

```
new FlxTimer().start(_waitToDisappear, function(_)
{
 if (_gameOver || _gameFinished)
 {
  Reg.saveScore();
  Reg.lives = 2;
  FlxG.switchState(new MenuState());
 }
 else
 {
  FlxG.sound.playMusic("pixelland");
  close();
 }
}, 1);
```

`FlxG.sound.playMusic()` will play and loop a music file in the background, and handily store the `FlxSound` object in the `FlxG.sound.music` variable, so that we can pause it and resume from anywhere in our code.

> All sound and music in HaxeFlixel are stored as `FlxSound` objects. When you call `FlxG.sound.play()`, HaxeFlixel checks if there are any "idle" `FlxSound` objects which have stopped playing and recycles them - if not, it will create a new one.

> You can store a a `FlxSound` object yourself to re-use it if you plan to play that specific sound several times, which can improve the performance in sound-intensive games - `FlxG.sound.play()` will return the instance of the `FlxSound` object being created / recycled: `var sound:FlxSound = FlxG.sound.play(...)`

> You can use the `FlxG.sound.load()` function as well, which takes the same arguments and works like `play()`, but without playing the sound.

> Moreover, both functions take a fourth optional boolean parameter called `AutoDestroy`, which should be set to "false" if you want to re-use the `FlxSound` instance.

If you run the game, you'll notice that the music keeps playing when the characted dies. Not exactly an happy moment!

We want to stop the music before playing the 'lose' jingle. In `Player.hx`:

```
override public function kill()
{
 if (!alive)
  return;

 FlxG.sound.play("death");
 FlxG.sound.music.stop();
 ...
 new FlxTimer().start(2.0, function(_)
  {
   FlxG.sound.play("dying");
   acceleration.y = GRAVITY;
   velocity.y = JUMP_FORCE;
  } , 1);
 ...
```

On the other hand, when the player reaches the level's goal, we want to fade out the music and play the victory jingle afterwards. In `Goal.hx`, add this to the `reach()` function:

```
public function reach(player:Player)
 {
  solid = false;
  Reg.pause = true;
  FlxG.sound.play("goal");
  FlxG.sound.music.fadeOut(1.0, 0);

  ...

  new FlxTimer().start(2.0, function(_) {
   FlxG.sound.playMusic("victory", 1, false);} , 1);

  ...
}
```

Whilst we can call `FlxG.sound.playMusic()` by pointing the music id only, we can pass some extra arguments for better control - in this case:

- `1` is the volume to play the song at (where `0` is mute, and `1` is the normal volume of the sound file) -`false` indicates that we don't want the song to loop. The same arguments apply to the `FlxG.sound.play` function.

Let's play a nice "ringing" sound as well when the time is converted into points, in `update()`:

```
override public function update(elapsed:Float)
{
    super.update(elapsed);

    if (_calculateScore)
    {
     if (Reg.time > 0)
     {
      FlxG.sound.play("time-convert", 0.75);
      Reg.time -= 5;
      Reg.score += 50;
     }
```

And that's it! In the next chapter we'll add the final finishing touches, and a few fancy effects.

Keep in mind that if you are deploying to the Flash target, .ogg music files are not supported, and you have to use .mp3 (moreover, .mp3 files will only work on the Flash target because of license royalties). You can copy the .mp3 versions of our music files from the book assets folder into `assets\music`, or using a free program like Audacity to convert them yourself.

Afterwards, we can tell the game to only use the .mp3 when targeting the Flash platform, using a conditional in the `Project.xml` (we'll learn more about conditionals in a few chapters):

```
<assets path="assets/music">
    <music path="pixelland.ogg" id="pixelland"/>
    <music path="victory.ogg" id="victory" />
    <music path="pixelland.mp3" id="pixelland" if="flash"/>
    <music path="victory.mp3" id="victory" if="flash"/>
</assets>
```

Extra Readings:

- HaxeFlixel API: FlxG.sound
- HaxeFlixel API: FlxSound

Finishing Touches

Our game is perfectly playable and has some sweet sounding music and sound effects. All that's left to a great product is adding a few extra effects and finishing touches.

Screen shake

Let's have our screen shake lightly as an added dramatic effect when the player dies. HaxeFlixel provides an easy way to do so with the `shake()` function in the `FlxCamera` class.

Open `Player.hx` and modify the first lines of the `kill()` position:

```
override public function kill()
{
 if (!alive)
  return;

 FlxG.sound.play("death");
 FlxG.sound.music.stop();
 FlxG.camera.shake(0.01, 0.2);
 ...
```

The two function arguments are respectively the intensity and the duration of the shake. In this case, the shake effect will have an intensity of `0.01`, and will last for `0.2` seconds.

Bear in mind that since we are calling the function of `FlxG.camera`, the shake effect will execute only on the default camera, which in our case is the `gameCamera` where all of our game objects are drawn in. The HUD texts will not shake as they are assigned to a different camera.

If you want to run the shake effect on all cameras, you can use `FlxG.cameras` instead (notice the extra "s" for the plural). This variable holds a reference to all the cameras in the game.

Running `FlxG.camera.shake()` will execute a shake effect on all cameras. We'll keep the shake for a single camera, but keep this in mind for the next effect we'll develop, the screen fade.

Transitions

Right now, all the transitions between states happen instantly. While this feels very retro and might be what you're looking for, for some games you'll want to develop some smoother transitions. We'll learn how to make the screen fade in and out in-between states.

Again, Haxeflixel provides a easy solution for this problem: the `fade()` function in the `FlxCamera` class. This function takes the following arguments:

- The color to fade to / from
- The fading duration in seconds

- A `Bool` value indicating whether the screen is fading in (`true`) or out (`false`)

Let's try this: open `IntroSubState.hx` and add a `fade()` to the beginning of the `create()` function:

```
override public function create():Void
{
 super.create();
 FlxG.cameras.fade(FlxColor.BLACK, .2, true);
 ...
```

If you run the game, the screen will fade in from black to showing the `IntroSubState`.

To have another transition when the actual game start, add the same `fade()` function when the `IntroSubState` is closed / disappears:

```
new FlxTimer().start(_waitToDisappear, function(_)
{
 ...
 else
 {
  FlxG.camera.fade(FlxColor.BLACK, .2, true);
  ...
  close();
 }
}, 1);
```

Right now, we only have fade-ins - The screen will first abruptly turn black, and then fade from black to the game. We want the screen to fade to black as well to get a smooth transition.

We can do this by fading the screen out first, then closing the `IntroSubState`, then finally fading the screen in.

Both `shake` and `fade` can take a optional extra parameter which is the function that will execute when the fade is complete, similar to how the `start()` function in `FlxTimer` works.

In this callback function we'll call another fade-in right before closing the state.

```
new FlxTimer().start(_waitToDisappear, function(_)
{
 ...
 else
 {
  ...
  FlxG.camera.fade(FlxColor.BLACK,.2, false, function()
  {
   ...
   FlxG.camera.fade(FlxColor.BLACK, .2, true);
   close();
  });
 }
}, 1);
```

Using the same logic, let's add a transition when the player dies as well, in `Player.hx`:

```
import flixel.util.FlxColor;
...
override public function kill()
{
 if (!alive)
  return;
 ...
 new FlxTimer().start(6.0, function(_)
 {
  FlxG.cameras.fade(FlxColor.BLACK, .2, false, function()
  {
   Reg.pause = false;
      FlxG.resetState();
  });

 });
}
```

And when the stage is completed, in the `nextLevel()` function of `PlayState.hx`:

```
public function nextLevel():Void
{
 Reg.checkpointReached = false;
 checkpoint = null;
 FlxG.cameras.fade(FlxColor.BLACK, .2, false, function()
 {
  Reg.level++;
  if (Reg.level < Reg.levels.length)
   FlxG.resetState();
  else
   FlxG.switchState(new IntroSubState(FlxColor.BLACK));
 });
}
```

Try and run the game - switching states should feel way smoother thanks to those transitions.

We are still missing transitions on the main menu - go ahead and try to add them yourself in `MenuState.hx` to practice. If you get stuck feel free to check the source code of this chapter.

Finishing up

We are now done programming a basic version of our game! Take a few minutes to bask in your masterpiece and pat yourself on the back as well - It's easy to start a project, but taking it to completion is the real challenge, even if it's a small demo or prototype.

Right now, we only tested the game on our PC (or Mac) using the `neko` target. In the next chapter we'll take a brief pause from coding and explore the different deployment targets Haxe supports, including mobile platforms!

Cross-Platform Deployment

Throughout the book we ran our game using the `lime test neko` command. This is just one of the several ways to compile our game through OpenFL.

in this chapter we'll explore the deployment options for the several supported targets we can export our game to.

Neko

This is the target we used so far in the book. Neko is a virtual machine that supports cross-platform compilation - it can run on Windows, Mac or Linux. It can run with `lime test neko`.

Neko compiles very quickly compared to native code and is therefore very useful during the testing stage.

The only drawback is that being a virtual machine, its performance may be somewhere inferior to native targets (`windows`, `mac` and `linux`).

Web Targets

As of HaxeFlixel 4.4.0 with OpenFL 8, the HTML5 target support in OpenFL 8 has vastly improved and is now the reccomended way to target web browsers. You can target HTML5 with `lime test html5`. A few considerations to keep in mind when deploying to the HTML5 target:

- Regular fonts tend to get blurry at low sizes (<12) - you might prefer using bitmap fonts to avoid text issues.
- Though .mp3 and .wav files are supported, SoundJS may not recognize them - You should use regular .ogg files.

When deploying to HTML5, an html page containing your game applet will be save into your project folder under `exports/html5/bin`.

OpenFL can also target flash export your game to a .swf file which can be played either with a stand-alone Flash Player or by embedding it in a web page, although as of 2021 this option is not reccomended as the Flash technology has been deprecated and most browsers are actively blocking it for security purposes. It can be run with `lime test flash`. When using this option, the game will be scaled to the full size of the browser window. You can directly export a HTML page with the .swf embedded in it with `lime test flash -web`. When using this option, the game size will be the same as the one defined in the `Project.xml` file.

This is the method I used to create the interactive applet you can play on the book's website.

Native Targets

When you compile for a native target, the Haxe code is converted into C++ by the Haxe compiler. A fully compiled executable is the generated thanks to the `hxcpp` library. Before being able to export to this target you need to install this library by typing `haxelib install hxcpp` in a command prompt / terminal.

The native builds will run on all the mainstream operative systems and are able to use the full power of your desktop machine. They can be deployed with different commands based on your operative system:

```
lime test windows

lime test mac

lime test linux -64
```

The drawback is that compilation time might be long, especially on the first build.

Of course you can only build for the operating system you're working on - you can't deploy to a Mac from a PC.

Before being able to compile on Windows or Mac you'll need to run the `openfl setup` command to install the dependencies needed to target that specific system:

- `openfl setup windows` will prompt download and install Visual Studio Community on a Windows system.
- `openfl setup mac` will download and install XCode on a Mac system. If you already download XCode previously, you can type `n` when prompted to download it to skip the download.
- There is no automatic setup for Linux systems, however you may need to install `g++` if it is not present already.

If you are releasing your game for a specific desktop platform, always try to build for the native target of that platform. This will make sure that your player will get the best performance possible from your game.

Project.xml Conditionals

All the nodes in the `Project.xml` supports `if` and `unless` attributes. This way you can customize your xml file to use only certain properties when building for certain targets.

The default `Project.xml` already comes with a few implementations for those properties:

```
<!--These window settings apply to all targets-->
<window width="640" height="360" fps="60"
     background="#000000" hardware="true" vsync="true" />

<!--Web-specific-->
<window if="web" orientation="landscape"  />

<!--Desktop-specific-->
<window if="desktop" orientation="landscape"
     fullscreen="false" resizable="true" />
```

You can see how `width` and `height` are set for all targets, but certain options will only be activated for specific targets.

For example, only on `desktop` targets (which include `neko`, `windows`, `mac` and `linux`), the game window will be re-sizable, since the property `resizable="true"` is wrapped inside a `if="desktop"` conditional.

Some of the values you can use for the conditionals are:

- `windows`, `mac`, `linux`
- `ios`, `android`, `blackberry`,
- `web`
- `desktop` (includes `windows`, `mac`, `linux`)
- `mobile` (includes `android`, `ios`, `blackberry`, `tizen`)
- `html5` (support is still experimental at this stage, but almost fully functional)
- `cpp`, `neko`, `flash`, `js`

Mobile Targets

A great advantage of using HaxeFlixel with OpenFL is being able to deploy your game to mobile devices.

Many cross-development frameworks boast about being able to deploy games to multiple mobile platform, while in reality they just export it to a Flash file and then wrap it around a web application, suffering severe performance hits.

HaxeFlixel with OpenFL, on the other hand, is able to convert the Haxe code to C++ code which is used natively either by XCode on iOS, or by the Android NDK on Android devices.

Android

To deploy to Android device, you'll need to have the following tools installed:

- Android SDK
- Android NDK
- Java SDK
- Apache ANT

You can quickly do this using the `openfl setup android` command. The setup will automatically

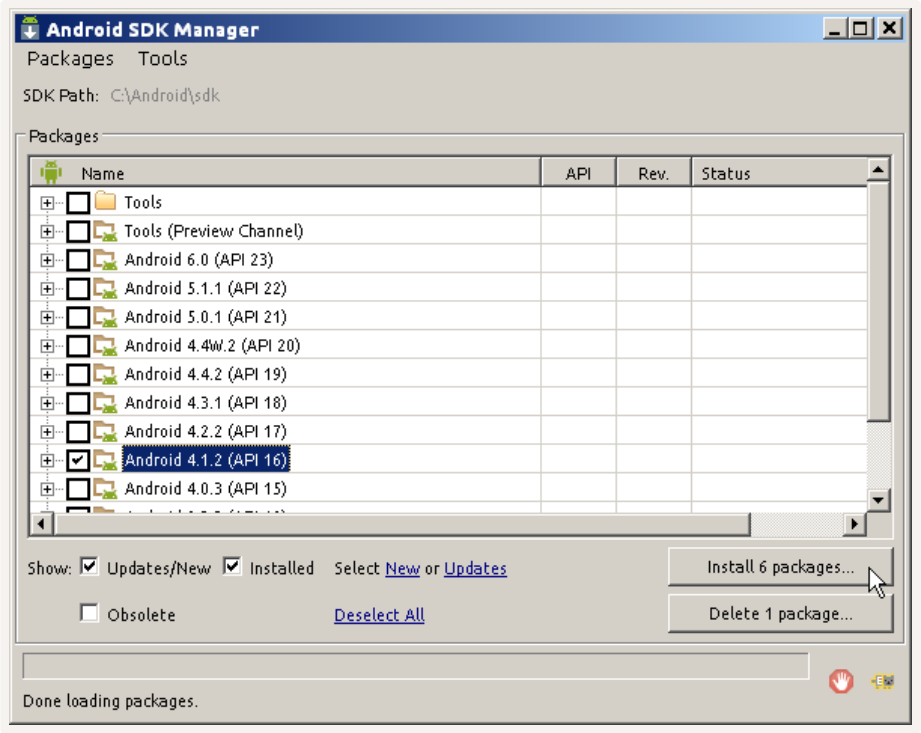
download and install each of those dependencies.

If you already installed the Android SDK and / or the Java SDK previously, you can run the command anyway but typing `n` when asked to download them. You will then be allowed to complete the setup by pointing to their local installation folders.

I recommend installing at least the Android NDK and Apache ANT through the automatic setup, as OpenFL needs a slightly older version of the software than the one being distributed online for compatibility reasons.

After installing the Android SDK, you should install the `Android SDK platform-tools` and `Android API 16` packages from the Android SDK Manager.

To do so, open up the folder where you installed the SDK to (usually "C:\Users\USERNAME\AppData\Local\Android\sdk"), and double-click "SDK Manager.exe" to open the SDK Manager. In this new window, scroll and tick the `Android 4.1.2 (API 16)` menu entry.



Press the "Install Packages" button on the bottom right, and after ticking the "Accept License" option box go ahead and press "Install".

HaxeFlixel games are compatible from `API 9` onwards, but they can use modern functionality when using the newer APIs (> 16). You'll only need the newest chosen API installed.

For the game to be deployed on a physical Android device, you'll have to make sure that its USB Debugging option is active and the device is correctly recognized.

To enable USB debugging on your Android device:

1. Open Settings > About > Software Information > More

2. Tap “Build number” seven times to enable Developer options
3. Go back to Settings menu and now you’ll be able to see “Developer options” there.
4. Tap it and turn on USB Debugging from the menu on the next screen.

If the device is still not recognized, you might need to download the USB drivers either from the SDK Manager (Download the `Google USB Driver` package inside the “Extra” folder) or from your device manufacturer’s website.

One last thing to do in order to make it able for the project to run on android at its current state is to comment out line 82 in the `Project.xml` file:

```
    <!-- <haxedef name="FLX_NO_KEYBOARD" if="mobile" /> -->
```

This will optimize inputs by disabling the keyboard commands on mobile, but requires you to write a few checks in your code to recognize whether the game is running on a mobile platform. For now just comment it out to make the game run - We’ll write those checks in the next chapter.

After this setup is complete, you are ready to to deploy your game to Android targets.

To test your game on the connected Android device, run `lime test android`. You can also use `lime test android -simulator` to run the game on a virtual device. In this case, make sure that the virtual device is API >=15 and has GPU enabled.

iOS

Deploying to iOS is a bit simpler, but is only possible when using a Mac system updated to the latest version of Max OS X. You will also need to have XCode installed on the machine. XCode can be downloaded for free from the Mac App store.

You’ll need to run the setup command for iOS platform as well: `sudo openfl setup ios`. If you didn’t download XCode previously, this command will automatically download and install it as well. If not, you can type `n` when prompted to download it.

To develop iOS applications you need to sign up and become a member of Apple’s iOS developer program as well.

To test your game on iOS, run `lime test ios`. You can also use `lime test ios -simulator` to run the game on the iOS simulator rather than using a physical device.

When building your game for iOS, OpenFL will generate a XCode project file, making it possible to use the XCode profiler and debugging tools.

If you managed to setup your mobile target correctly, try running `lime test android` or `lime test ios` to deploy the application to your android device or iPhone. You should be able to see your game running on the mobile device!

It will probably be displayed in the wrong aspect ratio and you’ll find yourself unable to start the game,

since you don't have any keyboard to enter commands with, but it's running nonetheless!

In the next few chapters we'll optimize the game to make it fully compatible with mobile devices thanks to the power of the **conditionals** attributes we saw earlier.

Lime Commands

A bit more about `lime` commands:

Normally, the `lime test` command will build your project and launch the executables created. When building a project, the executable files will be found inside the `export` folder.

There are other `lime` commands you can use throughout development:

- `lime test [platform] -release` when adding the `-release` flag to a command, the application will be built in release mode. This version will generate less information useful for debugging and development, and will give you a more accurate representation of performance of the final app. Make sure to use this flag when generating the build you want to publish and distribute.
- `lime run` will launch the application without rebuilding it. Useful when you close the application by mistake and you want to launch it again, and you haven't made any changes to the code.
- `lime build` will build the project without launching it.
- `lime test [target] -clean` using the `-clean` flag will force a full rebuild of the project from scratch, deleting all the intermediate files used to store information about the build. This can be useful when sometimes assets are not recognized, even if in the correct path.

Extra Reading

- HaxeFlixel Handbook: Desktop Targets

Optimizing for Mobile Platforms - Part I

After following the chapter on cross-platform deployment, you should have succeeded on getting your game to run on a mobile device. It's not looking the best probably, and you'll find yourself unable to continue as the touchscreen inputs are not yet recognized.

In this chapter we'll optimize the game to make it fully compatible with both desktop and mobile platform - one code base to rule them all!

Conditionals

In the previous section we briefly explored how we can use the `if` and `else` conditionals to set certain parameters exclusively for certain platforms in the `Project.xml`.

In the same way, conditionals can be applied to actual code in `.hx` files. This way we can execute only certain portions of code on a mobile platform. For example, add a separate section only on desktop.

Conditionals use the following syntax:

```
#if desktop
 // this code will run only on desktop platforms!
#end

#if mobile
 // this code will run only on mobile platforms!
#end
```

We can use the `else` flag as well:

```
#if ios
 // this code will run only on IOS platforms!
#else
 // this code will run on any other platform!
#end
```

Creating menu buttons

If you run your game on a mobile platform, you will get stuck at the `MenuScreen`, since you have to use the keyboard to select the "start" option - but there's no keyboard on mobile devices!

We'll add buttons that can be triggered by touch input, and use conditionals to make them appear only on mobile versions, while the desktop versions will retain the selection-based menu.

First of all, move the choice selection logic to a separate function called `resolveChoice()` in `MenuState.hx`:

```
private function resolveChoice(choice:Int):Void
{
 FlxG.cameras.fade(FlxColor.BLACK,.2, false, function()
 {
  switch (choice) {
   case 0: FlxG.switchState(new PlayState());
   case 1: System.exit(0);
  }
 });
}
```

```
override public function update(elapsed:Float):Void
{
 super.update(elapsed);

 ...

 if (FlxG.keys.justPressed.ENTER || FlxG.keys.justPressed.C)
  resolveChoice(_selected);
}
```

Now, onto creating the buttons: HaxeFlixel provides us with a straightforward class to implement interactive buttons - `FlxButton`.

Import this class and define two `FlxButton` - one for each menu option - at the top of the `MenuState` class:

```
class MenuState extends FlxState
{
 ...
 private var _buttonStart:FlxButton;
 private var _buttonExit:FlxButton;
```

And initialize them in the `create()` method:

```
_buttonStart = new FlxButton(_menuPos.x - 12, _menuPos.y,
        _menuEntries[0],
       resolveChoice.bind(0));
_buttonExit = new FlxButton(_menuPos.x - 12, _menuPos.y + _menuSpacing,
       _menuEntries[1],
       resolveChoice.bind(1));
```

The `FlxButton` constructor takes four arguments:

- The `x` and `y` coordinate of the button (in this case, the same ones we used to define the menu, plus an offset of 12 pixels on the x coordinate to better center the buttons)
- A text to be displayed on the button (in this case, the menu option)
- A callback function to be called when the button is clicked.

So we want to start the game when the first button is clicked, and quit the game when the second one is

pressed. We could define new functions, but we already have the functionality in `resolveChoice()`, so let's just pass it as a callback.

If the function has no arguments, we can just pass its name. When we pass a callback function which needs arguments however, we can't pass them with the classic notation `callbackFunction(ArgA)`. We need to use `bind()` instead, as you can see from the code: `callbackFunction.bind(argA)`.

If don't mind dipping into a more technical explanation, head over to the Function Binding section of the Haxe manual.

Finally, we can add the `FlxButtons` to the state:

```
add(_buttonStart);
add(_buttonExit);
```

When you run the game you'll see we have both the buttons AND the selection-based method. That's because we're adding everything to the state without making distinctions. Both options should be working fine - feel free to test the buttons.

What a mess!

We want the buttons to appear only on mobile platforms, while defaulting the selection-based menu on other platforms. To do so, we'll wrap this code section around a `#if mobile` conditional:

```
#if mobile
    add(_buttonStart);
    add(_buttonExit);
#else
    add(_cursor);
    for (i in 0..._menuEntries.length)
    {
        var entry:FlxText = new FlxText(_menuPos.x,
                                        _menuPos.y + _menuSpacing * i);
        entry.text = _menuEntries[i];
        add(entry);
    }
#end
```

And the code section related to the menu-based selection in the `update()` function inside a `#if !mobile` conditional (will run any platform which is not a mobile one):

```
override public function update(elapsed:Float):Void
{
    super.update(elapsed);

    #if !mobile
        if (FlxG.keys.justPressed.UP)
            _selected -= 1;

        if (FlxG.keys.justPressed.DOWN)
            _selected += 1;

        _selected = FlxMath.wrap(_selected, 0, _menuEntries.length - 1);

        _cursor.y = _menuPos.y + _menuSpacing * _selected;

        if (FlxG.keys.justPressed.ENTER || FlxG.keys.justPressed.C)
            resolveChoice(_selected);
    #end
}
```

If testing on mobile is too much of an hassle, feel free to change the `mobile` conditional to `desktop` in order to test this when building with `neko`. When you're developing for a mobile platform it's good practice to test on the actual device as much as possible.

With conditionals, only the buttons appear on mobile

You'll notice how the buttons look quite plain at the moment, but you can customize both graphic and text appearance on them.

You can access the button's `FlxText` object using the `label` variable, or its text directly as a string with `text`.

Let's extend the `forEach` loop we used to style the text to re-style the button's labels as well:

```
forEachOfType(FlxText, function(text)
{
    text.setFormat(AssetPaths.pixel_font__ttf, 8, FlxColor.WHITE,
                   FlxTextBorderStyle.OUTLINE, 0xff005784);
});
forEachOfType(FlxButton, function(btn)
{
    btn.label.setFormat(AssetPaths.pixel_font__ttf, 8, FlxColor.WHITE,
                        FlxTextBorderStyle.OUTLINE, 0xff005784);
});
```

You can see how we run `setFormat` on a `textToStyle` object we change dynamically based on what kind of object we are accessing in that loop.

If it's a `FlxText`, we can style directly; if it's a `FlxButton`, we need to access and style its `label` instead.

When using touch interactions on the mobile target, it's a good idea to disable the mouse in order to optimize the inputs (the application will stop tracking mouse clicks and such).

You can do this in your `Project.xml` file, under the `Haxedfines` section:

```
    <!--Optimise inputs, be careful you will get null errors
        if you don't use conditionals in your game-->
    <haxedef name="FLX_NO_MOUSE" if="mobile" />
    <haxedef name="FLX_NO_KEYBOARD" if="mobile" />
    <haxedef name="FLX_NO_TOUCH" if="desktop" />
    <!--<haxedef name="FLX_NO_GAMEPAD" />-->
```

As the comment say, when you do this you'll have to wrap any mouse event into a `#if !mobile` conditional.

Since the mouse will be disabled on mobile targets, if you call a mouse-related function on a mobile target the compiler will return an error.

We are not using any mouse-related function in our game so it will be fine, but keep this in mind for future development. You can also uncomment the `FLX_NO_KEYBOARD` line we disabled in the previous chapter, since we now added the conditionals.

In the next chapter we'll continue adding features for mobile platforms, including a virtual gamepad to control our character with.

Optimizing for Mobile Platforms - Part II

Once you start the game, we'll be facing a problem similar to the button-less menu: since we don't have a keyboard on mobile devices, it's impossible to move our character.

One way to fix this is to add several `FlxButton` objects to the state and place them to mimic a virtual gamepad - an approach used by many mobile games.

Adding and manually placing four to six buttons (the directions along with a jump and a run button) one by one would be somewhat tedious.

Luckily, HaxeFlixel provides us with a built-in class which will do this for us - the `FlxVirtualPad`.

The virtual gamepad

Open `PlayState.hx`, import the class and define a new `FlxVirtualPad` object:

```
...
import flixel.ui.FlxVirtualPad;
...

#if mobile
 public var virtualPad:FlxVirtualPad;
#end
```

Make sure to use a `#if mobile` conditional when defining the `virtualPad` object, and when adding it to the state:

```
#if mobile
 virtualPad = new FlxVirtualPad(LEFT_RIGHT, A_B);
 virtualPad.alpha = 0.75;
 add(virtualPad);
#end
```

Again, feel free to skip the conditionals or change them for now if you want to test the functionality without having to specifically deploy on mobile.

The arguments passed to the `FlxVirtualPad` constructor are, respectively, the direction button's layout and the action button's layout.

> The `FlxVirtualPad` can be initialized with several configurations, based on the chosen layouts for direction and action buttons.
>
> Layouts available for the direction buttons are:
>
> - `NONE`

- `UP_DOWN`

- `LEFT_RIGHT`
- `UP_LEFT_RIGHT`
- `FULL`

> Layouts available for the action buttons are:
>
> - `NONE`

- `A`
- `A_B`
- `A_B_C`
- `A_B_X_Y`

We also change its `alpha` value to `0.75` to make it partially transparent.

Run the game, and the virtual pad should be present on the game screen, with the “left” and “right” movement buttons on the bottom left of the screen, and two “A” and “B” action buttons on the right side.

Try and touch them - you’ll see the button graphic changing according to the touch state. All that’s left now is hooking their press / touch event to the character’s movement and action.

Open `Player.hx` - we’ll modify the `move()` function to make the character move right by responding to the correct control trigger: the keyboard when on a desktop platform and the virtualPad when on a mobile one.

```
private function move()
{
 ...
 else if (FlxG.keys.pressed.RIGHT)
 #if mobile
  || Reg.PS.virtualPad.buttonRight.pressed
 #end
 ) {
  flipX = false;
  direction = 1;
  acceleration.x += ACCELERATION;
 }
 ...
```

Let's see what's happening here. Normally, the condition for the right movement to happen is `FlxG.keys.pressed.RIGHT` ("The right arrow key of the keyboard has been pressed").

If we are on a mobile platform however, the code inside the `#if mobile` conditional tag will be taken into consideration as well, and the condition for the right movement will become "The right arrow key of the keyboard has been pressed" OR "The right button of the virtual pad has been pressed".

We check for the virtualPad button's status by accessing the `virtualPad` object inside our `PlayState` and checking the `status` flag on its `buttonRight` object.

That's right, the `FlxVirtualPad` is nothing more than a `FlxGroup` containing the `FlxButton` objects representing our `virtualPad`.

The `buttonRight` object is created automatically when the `FlxVirtualPad` is initialized, alongside with `buttonLeft`, `buttonA` and `buttonB`, reflecting our button's configuration options.

The status flag for a `FlxButton` can be accessed through the following variables, which will be `true` or `false` based on the button status:

- `justPressed` is `true` on the frame the button is pressed
- `pressed` is `true` while the button is pressed
- `justReleased` is `true` on the frame the button is released
- `released` is `true` while the button is not pressed

Run the game and press the right button on the virtualPad. The character should start to walk right, the same way as it would if we had pressed the right keyboard button.

Now we'd have to write similar conditions and conditionals for each action in the `Player` class - left movement, running, and jumping.

Although this would make the class a bit crowded. Let's manage the controls in a separate class.

Create a file called `ControlsHandler.hx` in the `utils` folder:

```
import flixel.FlxG;
import flixel.ui.FlxButton;

class ControlsHandler
{
 static public function keyPressedLeft():Bool
 {
  if ((FlxG.keys.pressed.LEFT)
   #if mobile
    || Reg.PS.virtualPad.buttonLeft.pressed
   #end
  ) {
   return true;
  }
  return false;
 }
```

```
 static public function keyPressedRight():Bool
 {
  if ((FlxG.keys.pressed.RIGHT)
   #if mobile
    || Reg.PS.virtualPad.buttonRight.pressed
   #end
  ) {
   return true;
  }
  return false;
 }
```

```
 static public function keyJustPressedJump():Bool
 {
  if ((FlxG.keys.justPressed.C)
   #if mobile
    || Reg.PS.virtualPad.buttonA.justPressed
   #end
  ) {
   return true;
  }
  return false;
 }
```

```
 static public function keyPressedJump():Bool
 {
  if ((FlxG.keys.pressed.C)
   #if mobile
    || Reg.PS.virtualPad.buttonA.pressed
   #end
  ) {
   return true;
  }
  return false;
 }
```

```
  static public function keyReleasedJump():Bool
  {
   if ((FlxG.keys.justReleased.C)
    #if mobile
     || Reg.PS.virtualPad.buttonA.justReleased
    #end
   ) {
    return true;
   }
   return false;
  }
```

```
  static public function keyPressedRun():Bool
  {
   if ((FlxG.keys.pressed.X)
    #if mobile
     || Reg.PS.virtualPad.buttonB.pressed
    #end
   ) {
    return true;
   }
   return false;
  }
} // end of class
```

It may look complicated at first glance, but it's really a simple class - we are repeating that right button logic for every control button.

We have a `static public` function corresponding to each key press / action, which returns a `Bool` value.

Inside the function, we run the keyboard / virtualPad button check in the same way we were doing in the `Player` class.

Instead of moving the player, we simply return `true` or `false`.

This way in `Player.hx`, we can change the `move()` function like this:

```
private function move()
{
 acceleration.x = 0;

 if (ControlsHandler.keyPressedLeft())
 {
  flipX = true;
  direction = -1;
  acceleration.x -= ACCELERATION;

 }
 else if (ControlsHandler.keyPressedRight())
 {
  flipX = false;
  direction = 1;
  acceleration.x += ACCELERATION;
 }

 if (velocity.y == 0)
 {
  if (ControlsHandler.keyJustPressedJump()
   && isTouching(FlxObject.FLOOR))
  {
   FlxG.sound.play("jump");
   jump();
  }

  if (ControlsHandler.keyPressedRun())
   maxVelocity.x = RUN_SPEED;
  else
   maxVelocity.x = WALK_SPEED;
 }

 if ((velocity.y < 0) && (ControlsHandler.keyReleasedJump())
  velocity.y = velocity.y * 0.5;
 ...
```

Not only does this makes the whole thing way more organized and modular, but it makes it very easy to re-use these functions in other classes that might need keyboard inputs, or to change the keys being used.

We can also change some of the code we wrote for the menu selection system and use `ControlsHandler.keyJustPressedJump()` everytime we query for the jump button, instead of manually hard-coding `FlxG.keys.justPressed.C` like we did in the previous chapter.

If you find yourself with a not-so-powerful device and you see the game slowing down, try deploying your game using the `release` flag as explained in the the cross-platform deployment chapter - it should improve the performance considerably.

Game Icon

When testing the game on your mobile device, you will have surely noticed how the icon for the game

app is the HaxeFlixel logo. Of course, this can be customized.

Icons on mobile are defined by a series of images with a specific size - be sure to check the iconography guide for Android and iOS platforms.

You can find some icons I made for both Android & iOS in `book-assets/images/icons`. Copy those two folders to the `assets/images/icons` folder in your project.

The icon parameters can be set in `Project.xml`. The name for your app will be the same one defined in the `title` attribute at the top of the file. This is not just for mobile platforms - icons will work on desktop targets as well.

To add a tag for the icon, scroll to the bottom of the file, where it says `<!--Place custom nodes like icons here (higher priority to override the HaxeFlixel icon)-->`. You'll want to add the code pointing to your icon image files here, based on your mobile target.

Android

```
<icon path="assets/icons/android/icon_48.png" size="48" if="android" />
<icon path="assets/icons/android/icon_72.png" size="72" if="android" />
<icon path="assets/icons/android/icon_96.png"  size="96" if="android" />
<icon path="assets/icons/android/icon_144.png"  size="144" if="android" />
<icon path="assets/icons/android/icon_192.png"  size="192" if="android" />
```

The iOS project has a few extra properties for the launch screens as well:

iOS

```
<set name="PRERENDERED_ICON" value="true" />

<icon path="assets/icons/ios/Icon.png" size="57" if="ios" />
<icon path="assets/icons/ios/Icon@2x.png" size="114" if="ios" />
<icon path="assets/icons/ios/Icon-72.png" size="72" if="ios" />
<icon path="assets/icons/ios/Icon-72@2x.png" size="144" if="ios" />

<launchImage path="assets/icons/ios/Default~iphone.png"
    width="320" height="480" if="ios" />
<launchImage path="assets/icons/ios/Default@2x~iphone"
    width="640" height="960" />
<launchImage path="assets/icons/ios/Default-Portrait~ipad.png"
    width="768" height="1024" if="ios" />
<launchImage path="assets/icons/ios/Default-Portrait@2x~ipad.png"
    width="1536" height="2048" if="ios" />
<launchImage path="assets/icons/ios/Default-Landscape~ipad.png"
    width="1024" height="768" if="ios" />
<launchImage path="assets/icons/ios/Default-Landscape@2x~ipad.png"
    width="2048" height="1536" if="ios" />
<launchImage path="assets/icons/ios/Default-568h@2x~iphone.png"
    width="640" height="1136" if="ios" />
```

You also have the possibility of using a .svg image for your icon (not included in the book assets), with:

```
<icon path="assets/your_vector_icon.svg" embed="true"/>
```

In this case, all the necessary icon size will be generated automatically from the vector image.

Extra Reading

- HaxeFlixel Handbook: Android
- - HaxeFlixel Handbook: iOS

Brick Block

In this chapter we will implement a block which will break when hit by a powered-up player, exploding into several falling bricks.

To implement the falling brick debris, we'll make use of the helper class `FlxParticle`, which allows us to quickly set parameters used in particle-like objects, such as `lifespan`.

We'll also make use of tweens to make the brick block bounce slightly when hit by a non powered-up player, and therefore not being broken.

The `BrickBlock` class

Create a file called `BrickBlock.hx` in `source/objects`, and start writing the `BrickBlock` class.

```
package objects;

import flixel.FlxObject;
import flixel.FlxSprite;
import flixel.FlxG;
import flixel.effects.particles.FlxParticle;
import flixel.tweens.FlxTween;
import flixel.tweens.FlxEase;


class BrickBlock extends FlxSprite
{
 private static var SCORE_AMOUNT:Int = 10;
 private static var GRAVITY:Int = 600;

 public function new(x:Float, y:Float)
 {
  super(x, y);
  immovable = true;

  loadGraphic(AssetPaths.items__png, true, 16, 16);
  animation.add("idle", [6]);
  animation.play("idle");
 }

 override public function update(elapsed:Float)
 {
  if (isOnScreen() && !Reg.pause)
         super.update(elapsed);
 }
```

We define a `SCORE_AMOUNT` variable to hold the amount of score gained when destroying the block, and a `GRAVITY` variable to specify how fast the brick debris will fall. The `new()` and `update()` functions should be familiar by now.

Now let's move onto the `hit()` function:

```
 public function hit(player:Player)
 {
  FlxObject.separate(this, player);

  if (!isTouching(FlxObject.DOWN))
   return;

  if (player.health > 0)
  {
   Reg.score += SCORE_AMOUNT;
   for (i in 0...4)
   {
    var debris:FlxParticle = new FlxParticle();
    debris.loadGraphic(AssetPaths.items__png, true, 8, 8);
    debris.animation.add("spin", [28, 29, 38, 39], 12);
    debris.animation.play("spin");
    FlxG.sound.play("brick");

    var countX:Int = (i % 2 == 0) ? 1 : -1;
    var countY:Int = (Math.floor(i / 2)) == 0 ? -1 : 1;

    debris.setPosition(4 + x + countX * 4 , 4 + y + countY * 4);
    debris.lifespan = 3;
    debris.acceleration.y = GRAVITY;
    debris.velocity.y = -160 + (10 * countY);
    debris.velocity.x = 40 * countX;
    debris.exists = true;

    Reg.PS.add(debris);
   }

   kill();
  }
  else
  {
   var currentY = y;
   FlxTween.tween(this, {y: currentY - 4}, 0.05)
     .wait(0.05)
     .then(FlxTween.tween(this, {y: currentY}, 0.05));

  }
 }
} //end of class
```

After calling `FlxObject.separate()`, we run a few checks on the dynamics of the collisions: first we check if the player is hitting the block from below with `if (isTouching(FlxObject.DOWN))`; then we check is the player is in a power-up state - `if (player.health > 0)`.

If both conditions are valid, we can break the BrickBlock.

FlxParticle

We run the next section of code in a loop to be repeated 4 times, since we want to create 4 different

brick particles, scattering in 4 directions.

Inside the loop, we initialize a new `FlxParticle` object called `debris`. A `FlxParticle` is an extended `FlxSprite`, with a few extra parameters and methods to easily implement particle behavior. We can therefore used standard `FlxSprite` methods such as `loadGraphic()` to load the debris graphic.

We are also playing a brick-breaking sound - make sure to copy the sound asset file `brick.wav` to the `assets\sounds` folder, and tag it in the `project.xml` file like we did with the other sounds a few chapters ago:

```
<assets path="assets/sounds">
    ...
    <sound path="brick.wav" id="brick" />
</assets>
```

We then define two variables `countX` and `countY`. You can see there's some math involved. What I'm doing here is using the current loop index (`i`) to calculate which debris particle I am modifying during that loop - either the top left, top right, bottom left or bottom right, based on the values of `countX` and `countY`.

This allows me to dynamically set the correct position and the velocity for each of those 4 brick particles in one line of code, instead of having to write the specific code for each particle.

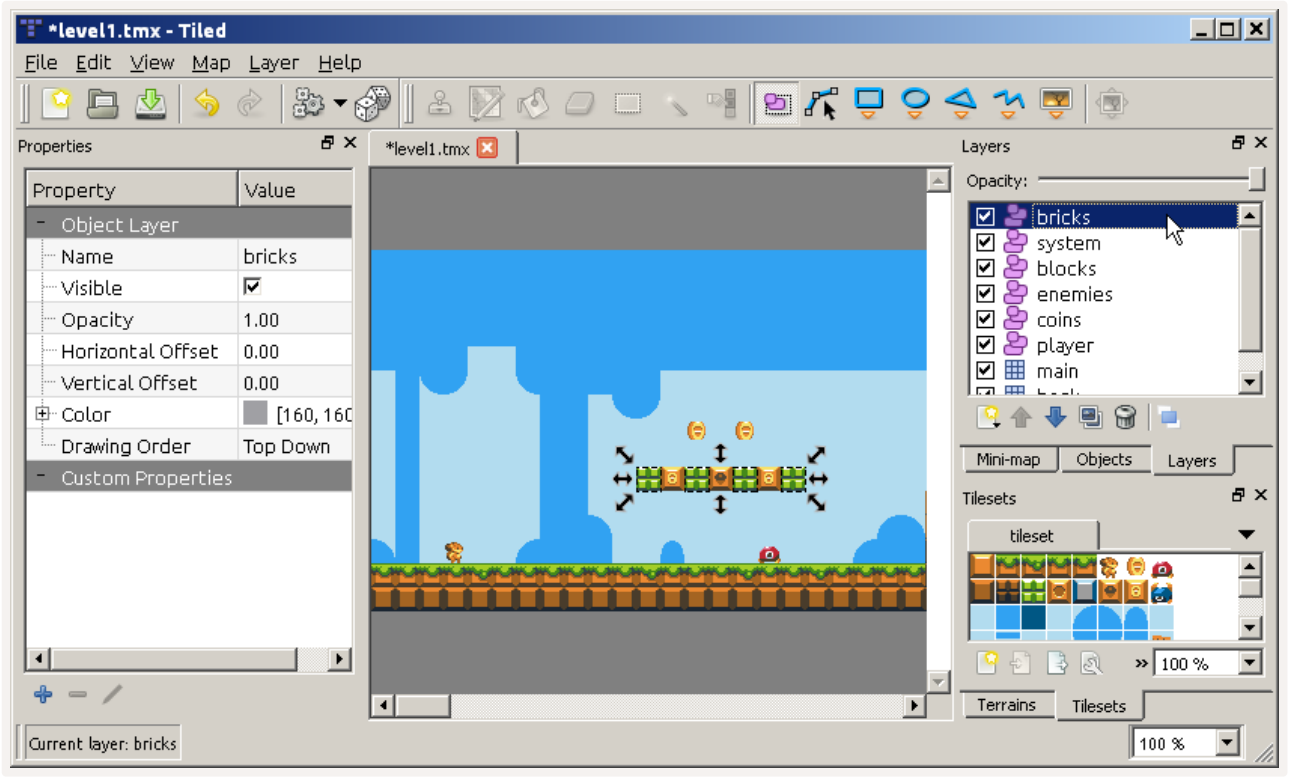
We also set their `acceleration.y` to gravity to make them fall down, and their `lifespan` to `3`. `lifespan` is a `FlxParticle` variable which indicates how long the particle will "live" - in this case, after 3 seconds the particle will call `kill()` on itself.

We also need to set `exists` to `true`, as particles are initialized in a non-existing state. We then can finally add the `debris` particle to the `PlayState`.

After all the brick particles have been created and added to the state, we can call `kill()` on the brickBlock to get rid of it.

If the player is in a non power-up state when he hits the block however, we don't want any of this to happen. In that case, we run a `FlxTween` chain which makes the block bounce slightly, similarly to what happens to the bonus block when hit.

To test our brick blocks, open the level in Tiled and add a new object layer called `bricks`. Add some bricks objects here, maybe replacing some static blocks we previously placed in the `main` tile layer (remember to erase the tiles in `main` in that case.)



Add the brick block loading logic in `LevelLoader.hx`:

```
// Load brick blocks
for (block in getLevelObjects(tiledMap, "bricks"))
  state.blocks.add(new BrickBlock(block.x, block.y - 16));
```

And the collision logic in `PlayState.hx`:

```
if (Std.is(entity, BrickBlock))
 (cast entity).hit(player);
```

You are now ready to test the game, grab a power-up, hit a block, and KA-BLAAM, enjoy a shower of brick particles.

> Whilst in this case we are creating a set number of `FlxParticle` objects ourselves, a more common pattern is to use a `FlxEmitter` object to generate such effects.

> `FlxEmitter` can be used for a one-time particle emission, or for continuous effects like rain or smoke. The advantage of using `FlxEmitter` for those scenarios is being able to set a specific range for particles' attributes such as velocity and acceleration, creating some nice-looking variation in the final effect.

> It's also very efficient performance-wise, as it handles particle destruction and recycling itself.

In the next chapter we'll create an invincibility power-up item which will make the player temporary invincible when picked up, and able to kill enemies by just touching them.

Extra Reading:

- HaxeFlixel Demo: Particles

Invincibility Bonus

In this chapter we'll create a bonus item for the player that will make them temporarily invincible and able to kill enemies just by touching them.

The `InvincibilityBonus` Class

Start by making a new file `InvincibilityBonus.hx` inside `source\objects`:

```
package objects;

import flixel.FlxObject;
import flixel.FlxSprite;
import flixel.FlxG;
import flixel.util.FlxSpriteUtil;

class InvincibilityBonus extends FlxSprite
{
 private static var MOVE_SPEED:Int = 80;
 private static var GRAVITY:Int = 420;
 private static var BOUNCE_FORCE:Int = -120;

 private var _direction:Int = 1;
 private var _moving:Bool = false;

 public function new(x:Float, y:Float)
 {
  super(x, y);
  loadGraphic(AssetPaths.items__png, true, 16, 16);
  animation.add("idle", [10, 11, 10, 12], 24);
  animation.play("idle");
  FlxG.sound.play("powerup-appear");

  velocity.y = -16;
 }

 override public function update(elapsed:Float)
 {
  if (Reg.pause)
   return;

     if (_moving)
     {
      velocity.x = _direction * MOVE_SPEED;

      if (justTouched(FlxObject.FLOOR))
   {
    y -= 1;
    velocity.y = BOUNCE_FORCE;
   }
     }
```

```
  if (!_moving && (Math.round(y) % 16 == 0))
  {
   velocity.y = 0;
   acceleration.y = GRAVITY;
   _moving = true;
  }

  if (justTouched(FlxObject.WALL))
   _direction = -_direction;

     super.update(elapsed);
 }

 public function collect(player:Player)
 {
  kill();
  trace("Obtained Invincible Bonus!")
 }
}
```

This class is very similar to the `PowerUp` class - this bonus will still come out from a bonus block and start moving once it's one full tile above the block.

The only difference is that instead of simply sliding on the floor, this bonus will bounce around, making it a bit harder to get. We'll implement this behavior by checking when it's colliding with the floor, and raising its `velocity.y` on collision.

This is happening inside the `if (_moving)` condition:

```
if (justTouched(FlxObject.FLOOR))
 velocity.y = BOUNCE_FORCE;
```

The `get()` function will print a simple debug message for now, while we take care of a few other classes.

Out of the Bonus Block

We want this bonus to come out of a Bonus Block as well. Remember how in the Tiled editor we specified which item was inside the bonus block with the `type` property? We only had `powerup` so far.

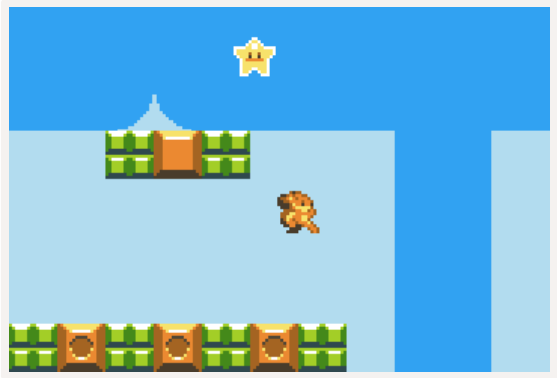
Create another block in the level inside the `blocks` object layer, and set its `type` to `invincible`.

Then, modify the `BonusBlock.hx` class to consider this new case:

```
private function createItem(_)
{
 switch (content) {

  ...

  case "invincible":
   var _invic:InvincibilityBonus = new InvincibilityBonus(Std.int(x),
               Std.int(y));
   Reg.PS.items.add(_invic);
 }
}
```

Let's implement its collision case with the player within `collideEntities()` in `PlayState.hx`:

```
if (Std.is(entity, InvincibilityBonus))
 (cast entity).collect(player);
```



Test the game and grab an InvincibilityBonus - the message should print on the debugger. We are now ready to implement the actual invincibility effect on the player.

`makeInvincible` on the player

Open `Player.hx` and add a new declaration for a private static variable `INVINCIBLE_DURATION` which will use to set the invincibility period time; and a public boolean variable `invincible` which we'll use to check whether the player is in an invincible state or not.

```
class Player extends FlxSprite
{
 ...
 private static var INVINCIBLE_DURATION = 5.0;

 public var direction:Int = 1;
 public var flickering:Bool = false;
 public var invincible:Bool = true;
 ...
```

Now let's add a function `makeInvincible()`:

```
public function makeInvincible():Void
{
    invincible = true;
    new FlxTimer().start(INVINCIBLE_DURATION, function(_)
 {
  invincible = false;
    });
}
```

This function will set the `invincible` flag to `true`, and then turn it to `false` again after the period of time specified in the `INVINCIBLE_DURATION` variable.

Right now, the player has no means to know when he is invincible, or when his invincibility period is over.

We will make the player's sprite flash with multiple colors whilst they are invincible. Add this to `update()`:

```
if (invincible)
 color = FlxColor.fromHSB((Reg.time * 1800) % 360, 1, 1);
else
 color = FlxColor.WHITE;
```

If the `invincible` flag is true, the `color` parameter of the sprite will cycle through multiple colors. We do this by using the `FlxColor.fromHSB()` function, which returns a color giving its hue, saturation, and brightness value.

The `color` parameter influences the tint of the sprite. Setting it to white is equal to having no tint, thus restoring the original color. We do this once the invincibility effect is over.

using `Reg.time` and the modulus operation, we cycle through the 360 values in the hue spectrum. We leave both saturation and brightness to 1.

If you try and run the game now, the player should start flashing after getting the InvincibilityBonus, and go back to normal after 5 seconds - a sign that the `invincible` flag is indeed working.

When touching an enemy however, he still dies! Doesn't sound very invincible to me for now. Let's fix that.

`checkIfInvincible` on the enemies

We want the enemy to die when colliding with an invincible player.

In the current implementation, the enemy dies by being "squished", as it only happens when the player jumps on him.

Moreover, the `SpikeEnemy` has no way to die, as it was supposed to always kill the player before we introduced the invincibility case.

We'll implement a new way for the enemy to die - by being knocked over and falling off the screen - and

make it common to every enemy.

Start by declaring a new private variable `_dieFlip` in the parent class `Enemy.hx` - we'll use this to identify whether the enemy will die in the "normal" way or being knocked over.

We'll also define a static `FLIP_FORCE` variable which indicates the strength at which the enemy will be thrown in the air when colliding with the invincible player.

```
class Enemy extends FlxSprite
{
 ...
 private static var FLIP_FORCE:Int = -100;
 ...
 private var _dieFlip:Bool = false;
```

Then, we can check the status of this flag in the `kill()` function. If `false`, we'll kill the enemy in the normal way. If `true`, we'll knock it over and have it fall off the screen.

```
override public function kill()
{
 alive = false;
 Reg.score += SCORE_AMOUNT;
 FlxG.sound.play("defeat");

 if (!_dieFlip)
 {
  velocity.x = 0;
  acceleration.x = 0;
  animation.play("dead");

  new FlxTimer().start(1.0, function(_)
  {
   exists = false;
   visible = false;
  } , 1);
 }
 else
 {
  flipY = true;
  velocity.y = FLIP_FORCE;
  acceleration.x = 0;
  solid = false;
 }
}
```

We do this by flipping the enemy's graphic with `flipY = true`, raising his `velocity.y` by the `FLIP_FORCE` value to throw it in the air and setting `solid` to `false` so he can fall through the floor.

Now we have to set the `_dieFlip` flag to true when colliding with the invincible player. Make a new function `checkIfInvincible()` which takes the `Player` object as argument:

```
private function checkIfInvincible(player:Player)
{
 if (player.invincible)
 {
  _dieFlip = true;
  kill();
 }
}
```

Then, call this function first in the `interact()` function, which, if your remember, is called when the enemy collides with the player:

```
public function interact(player:Player)
{
 checkIfInvincible(player);

 if (!alive)
  return;

 FlxObject.separateY(this, player);

 if ((player.velocity.y > 0) && (isTouching(FlxObject.UP)))
 {
  kill();
  player.jump();
 }
 else
  player.damage();
}
```

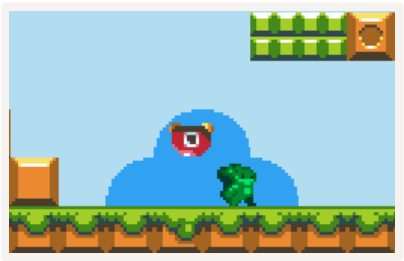
This way, when the player collides with the enemy, if will first check if the player is invincible - if so, it will call `kill()` with the `_dieFlip` flag.

Since calling `kill()` will set the `alive` variable to `false`, the rest of the `interact()` function where the player is damaged will not execute, since it's wrapped in the `if (alive)` condition.

Remember to make this change in the `SpikeEnemy` class as well, since we override the `interact()` method:

```
override public function interact(player:Player)
{
 checkIfInvincible(player);

 if (alive)
  player.damage();
}
```

Test the game now, and when invincible you should be able to knock enemies over on collision, killing them and earning score points!



We'll be using this new way of defeating the enemy quite often in the next few chapters, so let's wrap the functionality inside a function to call it quicker and more elegantly:

```
public function killFlipping()
{
 _dieFlip = true;
 kill();
}

private function checkIfInvincible(player:Player)
{
 if (player.invincible)
  killFlipping();
}
```

In the next chapter we'll create a new, more complex type of enemy - a `ShellEnemy` which after being hit will leave its shell on the ground, allowing the player to use it as a weapon to defeat other enemies.

Shell Enemy

In this chapter we will create a third enemy type. This enemy is covered by a hard shell, and when hit by the player he will hide inside it. The player can then kick its shell, which will start sliding on the ground, hitting other enemies! But be careful, as the moving shell can bounce off walls and hit the player back!

The `ShellEnemy` class

Let's take a moment to think about how to implement this behavior. The enemy will have 3 states:

- Normal state, walking around - this will be the starting state of the enemy. It will walk around and work like a normal enemy.
- `_isShell` state - when the player jumps on the enemy, it will enter this state - its empty shell will lie on the ground. At this point the enemy is not dangerous anymore, and when touched by the player, the shell will start moving. Which takes us to the third state…
- `_isMovingShell` state - in this state the shell if quickly sliding on the ground. It will damage other enemies when hitting them, but will damage the player as well. If the player manages to jump on it, it will stop and go back to the `_isShell` state.

We will use boolean variables to identify which state the enemy is in.

Create a new file `ShellEnemy.hx`:

```
package objects;

import flixel.FlxObject;
import flixel.FlxSprite;
import flixel.FlxG;

class ShellEnemy extends Enemy
{
 private static var WALK_SPEED:Int = 40;
 private static var SCORE_AMOUNT:Int = 100;

 private var _isShell:Bool = false;
 private var _isMovingShell:Bool = false;
 private var _waitToCollide:Float = 0;
```

```
 public function new(x:Float, y:Float)
 {
  super(x, y);

  loadGraphic(AssetPaths.enemyC__png, true, 16, 16);
  animation.add("walk", [0, 1, 2, 1], 12);
  animation.add("shell", [3], 12);
  animation.play("walk");

  setSize(12, 12);
  offset.set(2, 4);
 }

 override private function move()
 {
  if (_isMovingShell)
   velocity.x = _direction * WALK_SPEED * 4;
  else if (!_isShell)
   velocity.x = _direction * WALK_SPEED;
 }

 override public function update(elapsed:Float):Void
 {
     super.update(elapsed);
 }
}
```

A basic enemy class for now. Make sure to copy the file `enemyC.png` from `book-assets/images` and place it in the `assets/images` folder in your project. This file is the sprite sheet for this new ShellEnemy.

You can see how in the `move()` function the enemy will either move quicker if `_isMovingShell`, will stay still if `_isShell`, or move normally if none of the above.

Let's implement its `interact()` function, where most of this special enemy logic will happen. We'll take it easy and implement the states one by one.

```
override public function interact(player:Player)
{
 if (alive && _waitToCollide <= 0)
  return;

 checkIfInvincible(player);

 FlxObject.separateY(this, player);

 if (!_isShell)
 {
  if (player.velocity.y > 0 && isTouching(FlxObject.UP))
  {
   Reg.score += SCORE_AMOUNT;
   animation.play("shell");
   _isShell = true;
   velocity.x = 0;
   player.jump();
  }
  else
   player.damage();
 }
}
```

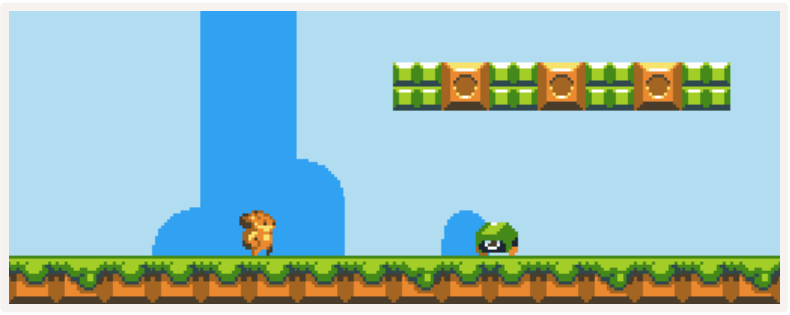
This is its standard behavior - it will move around and enter its `_isShell` state when jumped on by the player.

We can test it by opening our Tiled level and adding a new object in the "enemy" objectLayer, and setting its type to `shell`.

Then, implement the loading logic for this new enemy in `LevelLoader.hx`:

```
// Load enemies
for (enemy in getLevelObjects(tiledMap, "enemies"))
{
 switch(enemy.type)
 {
  ...
  case "shell":
   state.enemies.add(new ShellEnemy(enemy.x, enemy.y - 16));
 }
}
```

Run the game, and try and jump on a ShellEnemy - it should stop and turn into an empty shell, and set its `_isShell` flag to `true`. Nothing will happen if you touch it again at this point.



Let's fix this and make it so that the shell will start sliding on the ground when touched, damaging player and enemies.

Watch out for the moving shell

Let's change the `interact()` function to:

```
override public function interact(player:Player)
{
 if (!alive)
  return;

 checkIfInvincible(player);

 FlxObject.separateY(this, player);

 if (_isMovingShell)
 {
  if (player.velocity.y > 0 && isTouching(FlxObject.UP))
  {
   Reg.score += SCORE_AMOUNT;
   _isMovingShell = false;
   damageOthers = false;
   velocity.x = 0;
   player.jump();
  }
  else
   player.damage();
 }
 else if (_isShell)
 {
  if (player.velocity.y > 0 && isTouching(FlxObject.UP))
   player.jump();

  _direction = player.direction;
  _isMovingShell = true;
  damageOthers = true;
 }
```

```
  else // is walking
  {
   if (player.velocity.y > 0 && isTouching(FlxObject.UP))
   {
    Reg.score += SCORE_AMOUNT;
    animation.play("shell");
    _isShell = true;
    velocity.x = 0;
    player.jump();
   }
   else
    player.damage();
  }
}
```

Now when the player collides with the enemy and its `_isShell` flag is `true`, it will set its `_isMovingShell` flag to true as well (and make the player bounce on it if the collision happens from above).

This will get the shell moving, as outlined in the `move()` function. We also set the `damageOther` flag to `true`, we will soon implement this so that the moving shell will be able to damage other enemies.

If the player collides with a `_isMovingShell` ShellEnemy, it will get damaged too. If he manages to jump on it, however, the shell will stop, going back to the `_isShell` state.

But if you try and test this by running the game, you'll notice we have a little problem. When a player collides with a static shell, it will set the `_isMovingShell` flag to true - that means that the shell will be able to damage the player on collision from that moment onwards.

On the next frame calculation (remember that the game runs at 60 frames per seconds!), it's very unlikely for the shell to have moved away from the player - meaning that the collision will trigger again and the player will get damaged.

To fix this problem, we'll introduce a small time window after the shell gets moving where the collision check between the player and the shell will be skipped - just a fraction of seconds, enough for the shell to move far enough from the player.

Let's define a new `_waitToCollide` variable:

```
class ShellEnemy extends Enemy
{
 ...
 private var _waitToCollide:Float = 0;
```

In the `interact()` method, we'll increase this variable by a small amount everytime the `ShellEnemy` changes state:

```
...
if (_isMovingShell)
{
 if (player.velocity.y > 0 && isTouching(FlxObject.UP))
 {
  ...
  _waitToCollide = 0.25;
  ...
 }
 ...
}
else if (_isShell)
{
 if (player.velocity.y > 0 && isTouching(FlxObject.UP))
  player.jump();

 ...
 _waitToCollide = 0.25;
 ...
}
else // is walking
{
 if (player.velocity.y > 0 && isTouching(FlxObject.UP))
 {
  ...
  _waitToCollide = 0.25;
  ...
 }
 ...
}
...
```

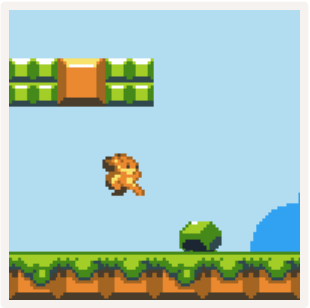
Then, in the `update()` method, we'll decrease this variable when it is larger than zero - effectively making it work like a mini-timer:

```
override public function update(elapsed:Float):Void
{
    super.update(elapsed);

    if (_waitToCollide > 0)
     _waitToCollide -= elapsed;
}
```

Finally, in the `interact()` function, we'll run through all the calculations only when the `_waitToCollide` variable is smaller or equal to zero - meaning that the small amount of time we added to the variable when `_isMovingShell` was set to true has now passed.

```
override public function interact(player:Player)
{
 if (!alive || _waitToCollide > 0)
  return;

 checkIfInvincible(player);

 FlxObject.separateY(this, player);

 if (_isMovingShell)
 ...
```

Test the game now, and the interactions between the player and the shell enemy should now work fine - play around and try to kick the shell, avoid it and jumping on it again. It's fun to make it bounce against walls!



Damaging other enemies

The last step now is making the moving shell able to damage the other enemies. Remember the `damageOthers` variable? We'll make use of it.

Open the `Enemy.hx` file and make a new public function called `collideOtherEnemy()`:

```
public function collideOtherEnemy(otherEnemy:Enemy)
{
 if (otherEnemy.damageOthers)
  killFlipping();
 else
  FlxObject.separate(this, otherEnemy);
}
```

We'll call this function when two enemies are colliding. We'll check if the second enemy's `damageOthers` variable is `true` - in that case we'll kill the first enemy after setting its `_dieFlip` variable to `true` (making it die by knocking if off the ground, like it was hit by an invincible player). If not, the enemies will bounce off each other as usual.

Now we have to use this function as a collision callback when we calculate collisions between enemies in our `PlayState`.

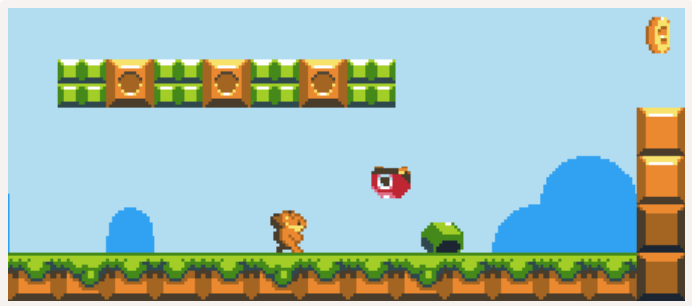
We could define a `collideEnemies` function which takes the two enemies objects, and the invoke the

`collideOtherEnemy` - same way we do with the player and `collideEntities`.

Since this is a small case, we can use an inline function and define the callback function right inside `FlxG.overlap()` and make the code a little more neat:

```
override public function update(elapsed:Float):Void
{
 ...
 FlxG.overlap(enemies, enemies, function(enemyA:Enemy, enemyB:Enemy) {
  enemyA.collideOtherEnemy(enemyB);
 } );
 ...
```

Run the game and try and get a scenario where a `ShellEnemy` is near another group of enemies. Jump on it and kick its shell towards them - you should see them being knocked over one after another!



Fireball PowerUp

In this chapter we'll create an extra power-up state for the player where they'll be able to shoot fireballs to defeat the enemies by pressing the run button.

The player will enter this new form by grabbing a PowerUp item while they are already in a powered-up state.

The `FireBall` class

Let's create our `FireBall` class first. We want our fireballs to bounce on the floor whilst rapidly moving in the direction the player is facing.

Copy the file `fireball.png` from `book-assets/images` and place it in the `assets/images` folder in your project to use the new fireball graphic. Then create a new file `FireBall.hx` inside the `objects` folder:

```
package objects;

import flixel.FlxObject;
import flixel.FlxSprite;
import flixel.FlxG;

class FireBall extends FlxSprite
{
 private static var MOVE_SPEED:Int = 140;
 private static var BOUNCE_POWER:Int = 160;
 private static var GRAVITY:Int = 960;

 public var direction:Int = -1;

 public function new(x:Float, y:Float)
 {
  super(x, y);
  loadGraphic(AssetPaths.fireball__png, true, 8, 8);
  animation.add("shoot", [0, 1, 0, 2], 24);
  animation.add("fade", [0, 3, 4], 24);
  animation.play("shoot");
  acceleration.y = GRAVITY;
 }
```

```
 override public function update(elapsed:Float)
 {
  if (Reg.pause)
   return;

     velocity.x = direction * MOVE_SPEED;

     if (justTouched(FlxObject.FLOOR))
   velocity.y -= BOUNCE_POWER;

  if (justTouched(FlxObject.WALL))
   kill();

         super.update(elapsed);
 }
}
```

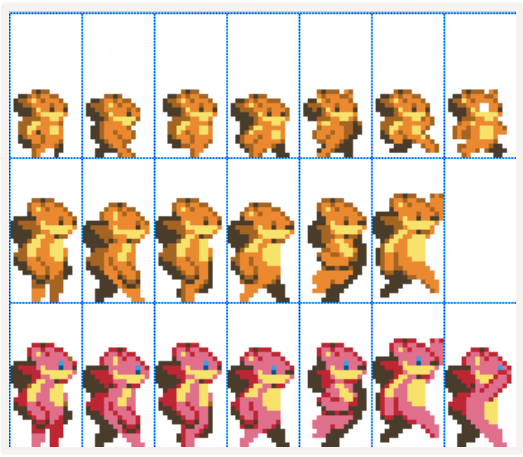
You can see how we launch the `FireBall` upwards every time it touches the ground by setting its `velocity.y` to (minus) the `BOUNCE_POWER` value. At the same time, the positive `GRAVITY` value in `acceleration.y` will drag the FireBall down. Those two forces combined will create the bouncing behavior.

If the fireball touches a wall from the side, it will simply be destroyed.

New player transformation

Now we need to define a new transformation form for the player. In its "base" form, the player's `health` is `0`. When he's in the "powered up" form, the `health` is `1`. We'll define a new "fireball" form when its `health` is `2`.

Copy the file `player-all.png` from `book-assets/images` and place it in the `assets/images` folder in your project. This file is an updated spritesheet for the player. You can see how we have a new third row which we'll use to define the graphic for this new transformation.



Change the `reloadGraphics()` function to:

```
private function reloadGraphics()
{
 loadGraphic(AssetPaths.player_all__png, true, 16, 32);
 switch (health)
 {
  case 0:
   setSize(8, 12);
   offset.set(4, 20);
   animation.add("idle", [0]);
   animation.add("walk", [1, 2, 3, 2], 12);
   animation.add("skid", [4]);
   animation.add("jump", [5]);
   animation.add("fall", [5]);
   animation.add("transform", [5, 12], 24);

  case 1:
   setSize(8, 24);
   offset.set(4, 8);
   animation.add("idle", [7]);
   animation.add("walk", [8, 9, 10, 9], 12);
   animation.add("skid", [11]);
   animation.add("jump", [12]);
   animation.add("fall", [12]);
   animation.add("transform", [12, 19], 24);
   animation.add("damage", [5, 12], 24);

  case 2:
   setSize(8, 24);
   offset.set(4, 8);
   animation.add("idle", [14]);
   animation.add("walk", [15, 16, 17, 16], 12);
   animation.add("skid", [18]);
   animation.add("jump", [19]);
   animation.add("fall", [19]);
   animation.add("shoot", [20]);
   animation.add("damage", [5, 19], 24);
 }

 animation.add("dead", [6]);
}
```

You can see we added new animations for the case when `health` is `2`. We are also getting rid of the common `transform` animation at the bottom and defining individual `powerup` and `damage` animations for each case, since they will now differ based on the current player's form. We also define a new exclusive `shoot` animation for the fireball form.

Although if we look at this long enough, we realize that most animations' index are shifted by fixed amounts (for example, the walk and run frames for the first power-up form are exactly 7 frames after the ones for the base form).

After all, the graphics are contained in a `7x3` spritesheet. Keeping this in mind, we can rewrite the `reloadGraphics()` function to be a little cleaner (and smarter):

```
private function reloadGraphics()
{
 loadGraphic(AssetPaths.player_all__png, true, 16, 32);
 animationOffset:Int = 0;
 switch (health)
 {
  case 0:
   setSize(8, 12);
   offset.set(4, 20);
   animation.add("powerup", [5, 12], 24);

  case 1:
   setSize(8, 24);
   offset.set(4, 8);
   animationOffset = 7;
   animation.add("powerup", [12, 19], 24);
   animation.add("damage", [5, 12], 24);

  case 2:
   setSize(8, 24);
   offset.set(4, 8);
   animationOffset = 14;
   animation.add("shoot", [20]);
   animation.add("damage", [5, 19], 24);
 }

 animation.add("idle", [0 + animationOffset]);
 animation.add("walk", [1 + animationOffset, 2  + animationOffset,
        3 + animationOffset, 2  + animationOffset], 12);
 animation.add("skid", [4 + animationOffset]);
 animation.add("jump", [5 + animationOffset]);
 animation.add("fall", [5 + animationOffset]);
 animation.add("dead", [6]);
}
```

Now let's change the `PowerUp()` function:

```
public function powerUp()
{
 if (health >= 2)
  return;
 ...
 animation.play("powerup");
 ...

 new FlxTimer().start(1.0, function(_)
  {
   ...;
   if (health == 1) y -= 16;
 ...
```

You can see how the function will execute even if the player's health is `1` - the player is in the powered up form. In this case, the "powerup" animation of the player going from powered up to fireball form will

now play, and the `health` will grow to `2`.

We also wrap the `y -= 16` around a condition, so that it happens only when the player goes from base form to powered-up form - since the fireball form is as tall as the powered up one, we don't want to shift its vertical position.

If the player grabs the `PowerUp` while being in the fireball form (the `health` will be at the new maximum value of `2`), it will just raise the score.

Change the animation in the `damage()` function as well:

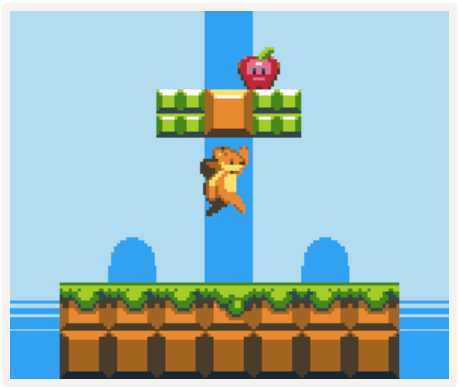
```
public function damage()
{
 if ((FlxSpriteUtil.isFlickering(this)) || (Reg.pause))
  return;

 if (health > 0)
 {
  ...
  health = 1;
  animation.play("damage");
  ...
```

Not much changes, when the player is hit by an enemy it will revert back to base form (`health` = `1`) whether he's on the powered-up or the fireball form.

Now let's change the `PowerUp` item to have a different graphic based on which form the player will transform to when grabbing it - we'll do so by checking its `health` variable:

```
class PowerUp extends FlxSprite
{
 ...
 public function new(x:Float, y:Float)
 {
  super(x, y);

  loadGraphic(AssetPaths.items__png, true, 16, 16);
  animation.add("powerup", [5]);
  animation.add("powerfire", [13]);

  if (Reg.PS.player.health < 1)
   animation.play("powerup");
  else
   animation.play("powerfire");

  FlxG.sound.play("powerup-appear");
  velocity.y = -16;
 }
```

Run the game, and try finding and grabbing a `PowerUp` when already in a powered-up form - the player should transform successfully and you can walk around with some new shiny graphics.



It's only an aesthetic at the moment, we still need to implement the shooting fireball functionality!

Shooting fireballs

We'll shoot fireballs with the same button we hold to make the player run. Make a new static function `keyJustPressedRun()` in `ControlsHandler.hx`:

```
static public function keyJustPressedRun():Bool
{
 if ((FlxG.keys.justPressed.X)
  #if mobile
   || Reg.PS.virtualPad.buttonB.justPressed
  #end
 ) {
  return true;
 }
 return false;
}
```

Let's create a function `shootFireball()` in `Player.hx`:

```
private function shootFireball()
{
 if (health != 2)
  return;

 if (ControlsHandler.keyJustPressedRun())
 {
  var fireball:FireBall = new FireBall(x, y);
  fireball.direction = direction;
  Reg.items.add(fireball);
  FlxG.sound.play("fireball");
 }
}
```

First of all, we want the player to be able to shoot fireballs only if he's in the fireball state (`health > 1`).

Then, we check if the run button has just been pressed with the newly created `ControlsHandler.keyJustPressedRun()` function.

If that's the case, we initialize a new `FireBall` object at the player's position and set its direction to be the same as the player's - we then add it to the `PlayState` in the `items` `FlxGroup`.

We are also playing a shooting sound - copy the sound asset file `fireball.wav` to the `assets\sounds` folder, and tag it in the `project.xml` file:

```
<assets path="assets/sounds">
    ...
    <sound path="fireball.wav" id="fireball" />
</assets>
```

Test the game, and when in the fireball form you should be able to shoot fireballs by pressing the run button.

However, we have no limitations on the shooting rate, and therefore repeatedly mashing the run button will create a waterfall of fireballs.

That's a bit overpowered and not very nice looking - we definitely want to fix this.

Define a new boolean `_canShoot` variable in `Player.hx`:

```
class Player extends FlxSprite
{
 ...
 private var _canShoot = true;
```

Let's now modify the `shootFireball()` function to make use it:

```
private function shootFireball()
{
 if (health != 2)
  return;

 if (ControlsHandler.keyJustPressedRun() && _canShoot)
 {
  var fireball:FireBall = new FireBall(x, y);
  fireball.direction = direction;
  Reg.PS.items.add(fireball);

  _canShoot = false;
  new FlxTimer().start(0.25, function(_) _canShoot = true);
 }
}
```

The function will now run only when the `_canShoot` variable is `true`.

Shooting a fireball will then set `_canShoot` to false, and initialize a `FlxTimer` to set it back to `true` after a short time interval.

Test the game again and the shooting rate should be a bit more manageable. You can modify the `FlxTimer` trigger time to make the "cooling" period between fireballs shorter or longer.

Let's implement the shooting animation too:

```
private function shootFireball()
{
 if (health != 2)
  return;

 if (ControlsHandler.keyJustPressedRun() && _canShoot)
 {
  var fireball:FireBall = new FireBall(x, y);
  fireball.direction = direction;
  Reg.PS.items.add(fireball);
  FlxG.sound.play("fireball");

  _canShoot = false;
  new FlxTimer().start(0.25, function(_) _canShoot = true);

  if (velocity.y == 0)
  {
   _stopAnimations = true;
   animation.play("shoot");
   new FlxTimer().start(0.1, function(_) _stopAnimations = false);
  }
 }
}
```

The `shoot` animation will play only if the player is not jumping (the current jumping animation looks like he's kind of shooting as well, so that saves us some work).

In a similar way to what we did before with the shooting rate, we temporarily set `_stopAnimations` to `true` to prevent the other animation to override the shooting one, and set up a `FlxTimer` to turn it back to `false` after a very short period of time.

Enemies and fireballs collision

Our fireballs are just for show at the moment, since they will pass through enemies without damaging them! Let's add a collision case for it.

In `Enemy.hx`, define a new boolean `_canFireballDamage` variable:

```
class Enemy extends FlxSprite
{
 ...
 private var _canFireballDamage:Bool = true;
```

And create a new function `collideFireball`:

```
public function collideFireball(fireball:FireBall)
{
 fireball.kill();
 if (_canFireballDamage)
  killFlipping();
}
```

This way, we can quickly make a specific type of enemy invincible to fireballs by setting its `_canFireballDamage` variable to `false` in the children enemy class.

Now to implement the actual collision case in `PlayState.hx`. Remember when we defined an anonymous function to manage the collision between enemies in the previous chapter?

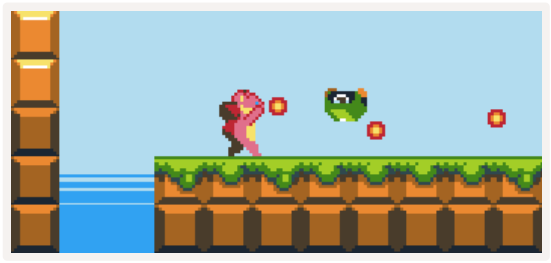
```
FlxG.overlap(enemies, enemies, function(enemyA:Enemy, enemyB:Enemy) {
 enemyA.collideOtherEnemy(enemyB);
} );
```

We can modify that function slightly to check for the collision of an enemy with any game entity (as opposed to specifically with another enemy), and then check that entity's type, and execute the appropriate operation:

```
FlxG.overlap(_entities, enemies, function(entity:FlxSprite, enemy:Enemy) {
 // overlapping another Enemy
 if (Std.is(entity, Enemy))
  enemy.collideOtherEnemy(cast entity);
 // overlapping a fireball
 if (Std.is(entity, FireBall))
  enemy.collideFireball(cast entity);
} );
```

It's very similar to what we're doing in the `collideEntities` function with the `Player`.

Run the game now, and the enemies should get appropriately knocked out when hit by a fireball. Blast!



You can see how that function is starting to grow up. It's probably a good idea to make a properly defined function and invoke it in the callback.

```
FlxG.overlap(_entities, enemies, enemyCollideEntities)

function enemyCollideEntities(entity:FlxSprite, enemy:Enemy):Void
{
 if (Std.is(entity, Enemy))
  enemy.collideOtherEnemy(cast entity);

 if (Std.is(entity, FireBall))
  enemy.collideFireball(cast entity);
}
```

You can see how the logic is very similar to what we do with the player and `collideEntities()`. In fact, we can actually rework `collideEntites` to work for both player the enemies, and make it a callback function for both collision cases.

Instead of calling `collideEntites` to check the collision of a game entity and the player, we can check between two generic `FlxSprite` game entities.

If one entity is the `Player` (we can check its type using `Std.is()`), we'll check if the other one is a `PowerUp`, a `Coin`, a `BonusBlock` - anything the player might be interested in colliding with.

If that entity is an `Enemy`, instead, we'll check if the other one is another `Enemy` or a `Fireball` - the collision cases we have to cover for an `Enemy`.

Keeping this new logic in mind, we can modify the collision functions in `update()`:

```
override public function update(elapsed:Float):Void
{
 super.update(elapsed);

 if (player.alive)
 {
  FlxG.overlap(_entities, player, collideEntities);
  FlxG.collide(_terrain, player);
 }

 FlxG.collide(_terrain, _entities);
 FlxG.overlap(_entities, enemies, collideEntities);

 updateTime(elapsed);
 updateCheckpoint();
}
```

And change the implementation of the `collideEntities()` callback function.

```
function collideEntities(entity:FlxSprite, subject:FlxSprite):Void
{
 if (Std.is(subject, Player))
 {
  var player:Player = cast subject;

  if (Std.is(entity, Coin))
   (cast entity).collect();

  if (Std.is(entity, Enemy))
   (cast entity).interact(player);

  if (Std.is(entity, PowerUp))
   (cast entity).collect(player);

  if (Std.is(entity, InvincibilityBonus))
   (cast entity).collect(player);

  if (Std.is(entity, BonusBlock))
   (cast entity).hit(player);

  if (Std.is(entity, BrickBlock))
   (cast entity).hit(player);

  if (Std.is(entity, Goal))
   (cast entity).reach(player);
 }
 else if (Std.is(subject, Enemy))
 {
  var enemy:Enemy = cast subject;

  if (Std.is(entity, Enemy))
   enemy.collideOtherEnemy(cast entity);

  if (Std.is(entity, FireBall))
   enemy.collideFireball(cast entity);
 }
}
```

Test the game, and… it should run exactly like it did before. This change is not having any visible change on our gameplay - it's simply making our code more organized and easier to read.

Conclusion

Run the game and play through some of your creations: you've probably grown used to it after all this testing, but if you were to show it to the past you who had just started to pick up the book, I can guarantee you'd be stunned!

After all, you've successfully created a playable, production ready game starting from zero. Congratulations! As I already said before, everyone can start a project, but only a small fraction of those people are dedicated enough to take it to the finish line.

You should now be comfortable enough with Haxe and the HaxeFlixel network and ready to take your game development journey forward. Feel free to modify the base of the game we've been developing together, refine it, or move to an entirely new project to challenge yourself.

No matter if your next game is going to be a shooter, a role-playing game, or a puzzle game - once you break the game idea down to basic elements, like we did in each chapter of this book, you'll see that, while approaching them one by one, they won't feel much different from what we've faced throughout the book.

However, you've just started to scrape the surface of the HaxeFlixel framework. There are many other classes, functions and game programming patterns to be discovered.

The full HaxeFlixel source code is freely available online, along with an extensive collection of API documentation and demos showcasing its functionality.

Those demos are without doubt the most precious learning resource you can find: you'll be able to see and interact with application and read the source code at the same time.

After reading this book, understanding the inner working of a demo's code won't be a problem, and you'll soon find yourself picking up new techniques and discovering new, useful classes.

What Next

- Visit the HaxeFlixel forums. Look at what issues people are encountering, and what is being suggested to fix them. Save solutions you consider pretty smart, or which might be useful to you in the future.
- Test and play the 75+ different demos. If you're impressed by any of them, explore its source code. Break it apart and see how it works, then try to implement some of its functionalities in your own game.
- Read the API of the classes you're already familiar with, like `FlxSprite` or `FlxText`. We only explored their basic functionality - you might discover several useful, less-known functions which in one line of code will do what you were trying to accomplish with 20.

Good luck and happy game development!

Thanks to my family and Rochelle for the help and the encouragement.

Special thanks to Samuel Batista for the kind support
and Jens Fischer for the extensive editing and proofreading.