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.

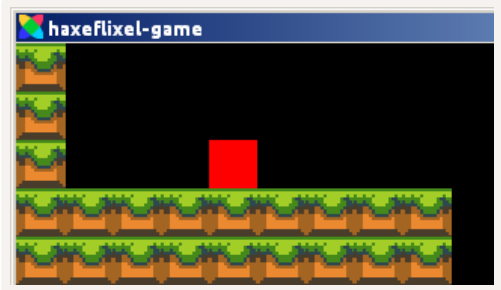
If we run our game now, we can see both the level and the player on the screen! Unfortunately, the player accelerates downwards and quickly falls through the floor, as we have no collisions set up yet! Well, that was a short play.

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.

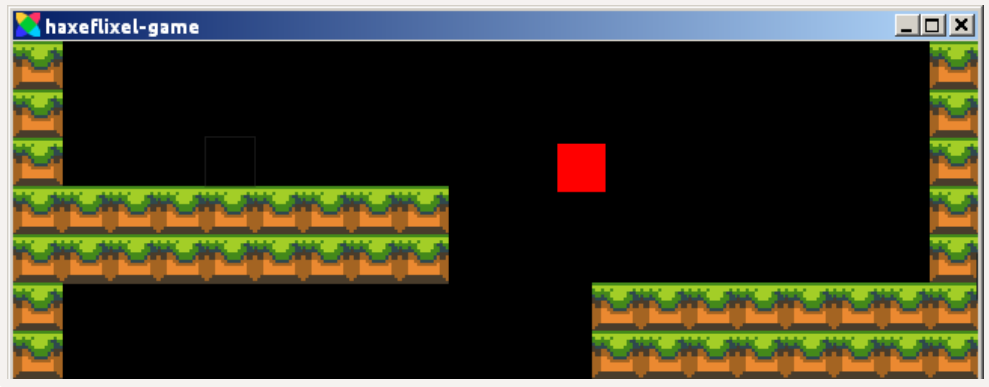
We check against `FlxObject.FLOOR`, which represents the lower side of an object. Other useful flags are `FlxObject.WALL`, which checks for left and right direction, and `FlxObject.CEILING`, which check for the upper direction.

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!

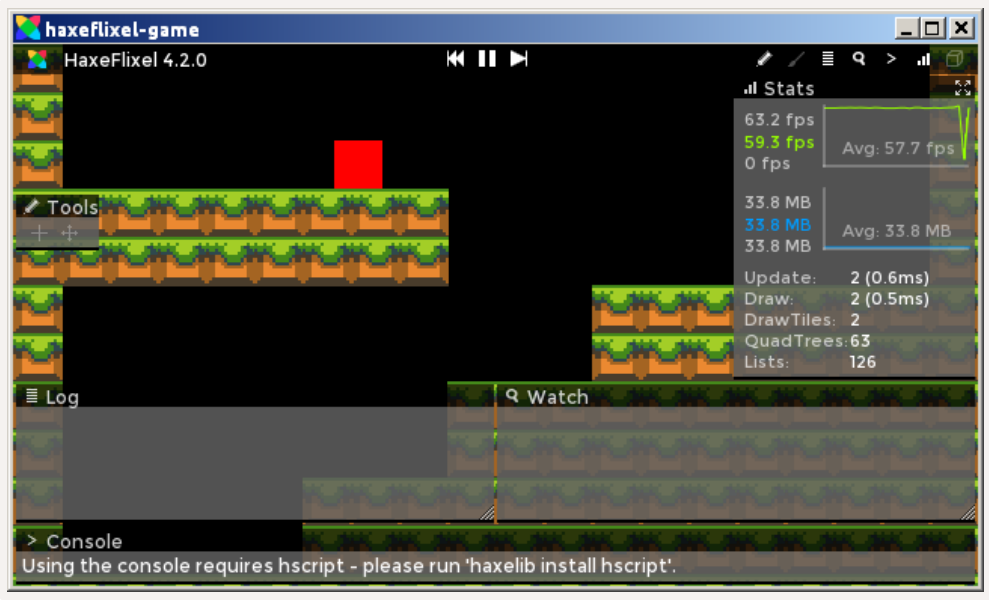
```
override public function update(elapsed:Float):Void
{
 super.update(elapsed);

 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