DISCOVER PHASER

learn how to make great HTML5 games

Thomas Palef

Discover Phaser

Learn how to make great HTML5 games

Thomas Palef

About the Book

- *Title: Discover Phaser*
- *Subtitle: Learn How to Make Great HTML5 Games*
- *Author: Thomas Palef*
- *Publication date: 1st July 2014*
- *Version: 1.1*
- *Purchased on* ***discoverphaser.com***

If you see any typos or have any feedback, please get in touch: ***thomas.palef@gmail.com***.

Contents

1 - Introduction

If you are reading this book it means that you are interested in HTML5 and games, and you've come to the right place to learn more about both.

The Phaser Framework

More and more people are talking about HTML5 games because it's a relatively new technology that has a huge potential. With HTML5 we can build games that are compatible with basically everything that has a browser on it (desktops, smartphones, tablets, gaming consoles, etc.) without the need to download anything.

There are now dozens of HTML5 game frameworks available out there, so it might seem hard to decide which one to choose. However, if you are looking for a free, open source, and actively maintained framework, the list quickly narrows down to just a handful of them. Each one has its pros and cons, but a lot of people consider Phaser to be the best one.

Phaser is a 2D HTML5 framework in Javascript, that was created in 2013 by Richard Davey. Phaser is free, open source, has new versions available every few weeks, and has a strong community. The framework itself is super powerful while being quite simple to use. Phaser is a really great framework to work with.

What You Will Learn

We will focus on building a full featured game from scratch. By the end of the book you will have a real game to play with, and enough knowledge to build your own games. You will learn how to install and set up Phaser, how to create an empty project, how to add a playable character with enemies, how to add sounds, animations, particles, tilemaps, how to make the game mobile friendly, and much more.

Requirements

To be able to fully understand this book, you need to already know how programming works (variables, loops, conditions, etc.), and be at least a little familiar with HTML and Javascript. If that's not the case, you should learn about these things before continuing.

About the Author

I discovered Phaser in December 2013. Since then I've built more than a dozen games with the framework that were played by hundreds of thousands of players around the world.

While building all these games, a lot of people asked me to teach them how to make HTML5 games. I wrote a few short tutorials on my blog, but that wasn't enough. That's why I decided to write this book, to help people learn more about this great framework.

2 - Get Started

The first thing we will see in this book is how to get started with the Phaser framework. We will discuss the most important Phaser resources, see how to set up a development environment, and code our first tiny project.

This is going to be a short chapter, since all of this is really simple.

2.1 - Useful Links

Thanks to the growing Phaser community, there are plenty of interesting resources to find online. I listed below the most important links related to Phaser, that you should visit and bookmark.

Phaser website phaser.io
The official website, it's the best place to stay updated with the latest Phaser news.

Phaser GitHub page github.com/photonstorm/phaser
The repository to download the framework. There are new versions available every few weeks.

Phaser documentation docs.phaser.io
If you're not sure how to use a function, this is where you should go first.

Phaser code examples examples.phaser.io
A really useful website that shows you a lot of short code examples. It's worth spending time there to see what Phaser is capable of.

Phaser forum html5gamedevs.com/forum/14-phaser
The best place to ask and answer questions about the framework.

For your information, all of these links can be found on the official Phaser website.

2.2 - Set Up Phaser

Let’s see what we should do to start making games with Phaser, in just 4 easy steps described below.

Download Phaser

The first thing we need is the Phaser framework itself. There are 2 main versions available on GitHub: Master (contains the most recent stable release) and Dev (the work in progress version). In this book we are going to use Phaser Master 2.0.5 that you can **download here**.

Phaser 2.0 is supposed to be forward compatible, it means that the code from this book should still work when a new version is available. However, it’s probably better that we both use the 2.0.5 version.

The most important things to look for in the Phaser directory are:

- docs/index.html, the offline documentation
- build/phaser.min.js, the Phaser framework that we will use in this book

Of course feel free to explore the folder in greater detail.

Important Tools

To make a game with Phaser, we only need 2 basic tools:

- A code editor. Any editor will do the job, but I can recommend **Brackets**.
- A browser with developer tools enabled. The developer tools will be really useful for debugging. I recommend **Google Chrome**.

When building games it’s also important to have some kind of image editing application like Gimp or Photoshop, but we won’t need one in this book.

Webserver

Running Phaser games directly in a browser doesn't work, that's because Javascript is not allowed to load files from your local file system. To solve that, we will have to use a webserver to play and test our games.

There are a lot of ways to set up a local webserver on a computer, and we are going to quickly cover a few below.

- Use Brackets. Open any HTML file in the **Brackets editor**, and click on the small "N" icon that is in the top right corner of the window. This will directly open your browser with a live preview from a webserver. That's probably the easiest solution.
- Use apps. You can download **WAMP** (Windows) or **MAMP** (Mac). They both have a clean user interface, with easy set up guides available.
- Use the command line. If you have Python installed and you are familiar with the command line, simply type "python -m SimpleHTTPServer" to have a webserver running in the current directory.

There are plenty of detailed tutorials online that can help you with this.

Test Phaser

Once all of the above is done, you should test that everything is working properly. For example, try to use your webserver to open the "resources/tutorials/01 Getting Started/hellophaser/index.html" file that is in the Phaser directory. If you can see the Phaser logo, it means that it works. If not your webserver is not properly configured.

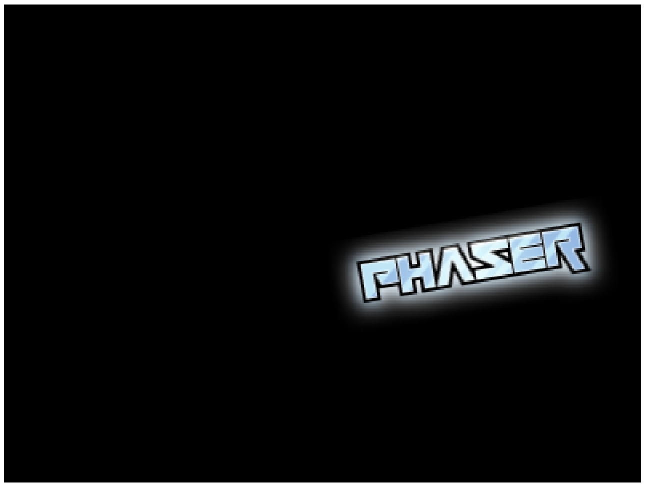
If you use Brackets, simply open the index.html file and click on the small "N" icon to see the Phaser example.

If you don't use Brackets:

1. Put the Phaser directory you downloaded earlier in your local webserver.
2. Open your browser and go to your webserver. This may be as simple as typing in "localhost" or "127.0.0.1", it depends on your configuration.
3. There you should see the list of files you have in your server. Click on the Phaser directory to see its content.
4. Then add "/resources/tutorials/01 Getting Started/hellophaser/index.html" at the end of the URL, to access the Phaser example.

2.3 - First Project

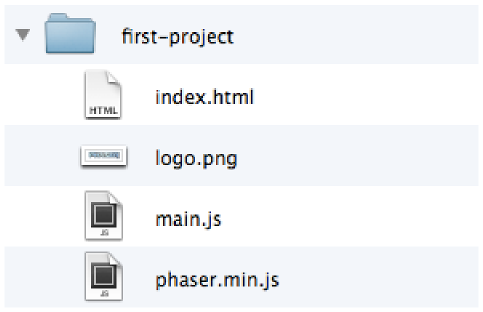
Now that Phaser is properly set up, it's time to code our first project. We will do something really simple: have a sprite rotate on the screen.



Set Up

We start by creating a new directory called "first-project". At the root of this folder we add all these files:

- phaser.min.js, the Phaser framework. It can be found in "phaser/build/".
- main.js, that will contain the game's code. For now it's just an empty file.
- index.html, that will display the game. Again, it's just an empty file for now.
- logo.png, the image that will rotate. It's in "phaser/resources/Phaser Logo/2D Text/Phaser 2D Glow.png", don't forget to rename it.



```
first-project
    index.html
    logo.png
    main.js
    phaser.min.js
```

Once it's done, the next steps are to code the index.html and main.js files, as described below.

HTML Code

Here's the code we should add in the index.html file. If you are familiar with HTML, you will see that it's really basic.

```
<!DOCTYPE html>
<html>

    <head>
        <meta charset="utf-8" />
        <title> First project </title>

        <script type="text/javascript" src="phaser.min.js"></script>
        <script type="text/javascript" src="main.js"></script>
    </head>

    <body>
        <p> My first project with Phaser </p>
        <div id="gameDiv"> </div>
    </body>

</html>
```

This code does 2 important things:

- Load all the Javascript files: phaser.min.js and main.js
- Add a div called "gameDiv", that will contain the game itself

Javascript Code

A game is usually divided into multiple scenes: a loading scene, a menu scene, a play scene, etc. In Phaser, a scene is called a "state", and this simple project we will only use one.

We will create our state, initialise Phaser, and start our state. Here's how we can do that:

```
// We create our only state
var mainState = {

    // Here we add all the functions we need for our state
    // For this project we will just have 3 functions

    preload: function() {
        // This function will be executed at the beginning
        // That's where we load the game's assets
    },

    create: function() {
        // This function is called after the preload function
        // Here we set up the game, display sprites, etc.
    },

    update: function() {
        // This function is called 60 times per second
        // It contains the game's logic
    }
};

// We initialising Phaser
var game = new Phaser.Game(400, 300, Phaser.AUTO, 'gameDiv');

// And finally we tell Phaser to add and start our 'main' state
game.state.add('main', mainState);
game.state.start('main');
```

Don't worry if you don't understand everything, we will see this in greater details in the next chapter.

All we have to do now is to fill the `preload`, `create` and `update` functions to have our rotating sprite:

```
var mainState = {
    preload: function() {
        // Load the image
        game.load.image('logo', 'logo.png');
    },

    create: function() {
        // Display the image on the screen
        this.sprite = game.add.sprite(200, 150, 'logo');
    },

    update: function() {
        // Increment the angle of the sprite by 1, 60 times per seconds
        this.sprite.angle += 1;
    }
};

var game = new Phaser.Game(400, 300, Phaser.AUTO, 'gameDiv');
game.state.add('main', mainState);
game.state.start('main');
```

Add the above code in the main.js file we created earlier.

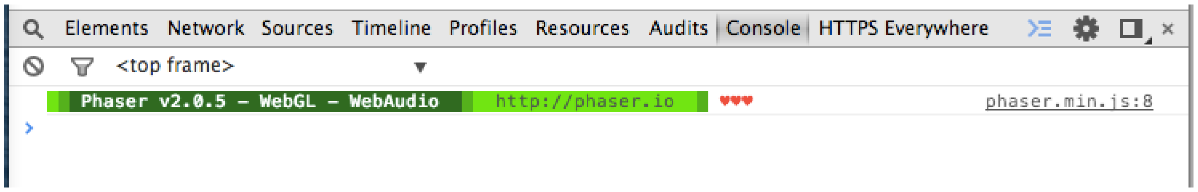
Test the Project

You just finished your first Phaser project. The next step is to test it, and it's basically the same process we followed in the previous part:

- Open the index.html file with Brackets, and click on the "N" icon
- Or put the "first-project" folder in your webserver, and try to access it from your browser

And you should see the logo rotating on the screen.

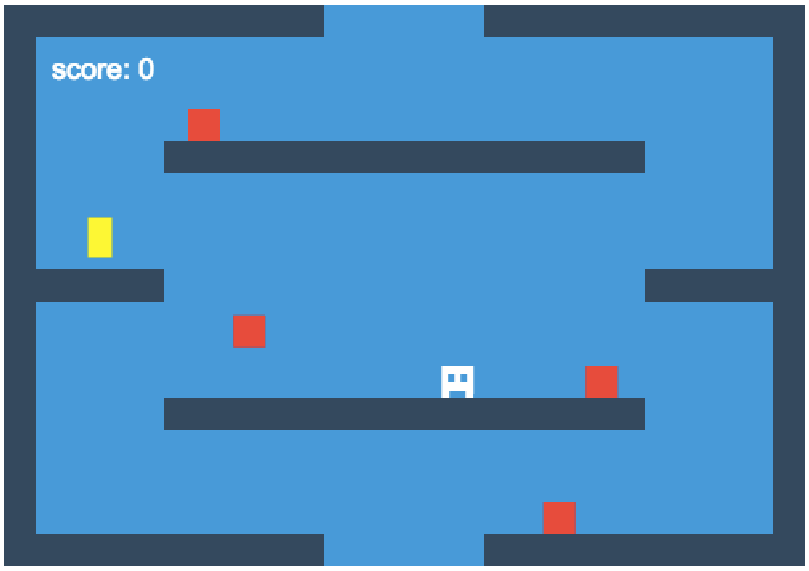
If it doesn't work for whatever reason, you need to bring up the debug console and see what errors are output. For Google Chrome you can do this by pressing Ctrl + Shift + J (Windows and Linux) or Cmd + Alt + J (Mac). Check what the errors are, and try to fix them.



```
Phaser v2.0.5 - WebGL - WebAudio   http://phaser.io ♥♥♥     phaser.min.js:8
```

3 - Basic Elements

We will see how to make a real Phaser game that is inspired by the popular Super Crate Box. We will control a small guy in a blue world that tries to collect coins while avoiding enemies.



When finished, the game is going to be full featured: menu, player, enemies, animations, sounds, tilemaps, mobile friendly, etc. And since that's a lot of information, we will cover all of this step by step in 5 different chapters.

This is the first chapter where we are going to create the basic elements of the game: a player, a world, some coins, and many enemies.

3.1 - Empty Game

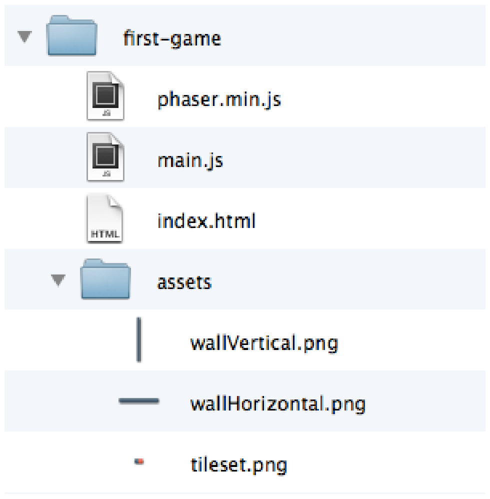
Let's start by creating an empty game, so this is going to be similar to what we did in the previous chapter. By the end of this part, we will have this:

Don't worry, it will quickly become more interesting.

Set Up

First we need to create a new directory called "first-game", where we should add:

- phaser.min.js, the Phaser framework.
- main.js, that will contain the game's code. For now it's just an empty file.
- index.html, that will display the game. We can use the same index file that we made in the previous chapter.
- assets/, a directory that contains all the images and sounds. The assets can be **downloaded here**



Code the Main File

The Javascript code for any new Phaser project is basically always going to be the same:

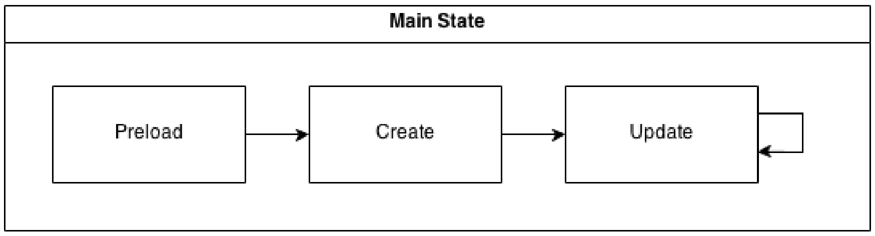
1. Create the states. Remember that a state is a scene of a game, like a loading scene, a menu, etc.
2. Initialise Phaser. That's where we define the size of our game among other things.
3. Tell Phaser what the states of our game are.
4. And start one of the states, to actually start the game.

These 4 steps are explained below in detail.

First, we create the states. For now we will have only one, that will include 3 of the default Phaser functions:

```
// We create our only state, called 'mainState'
var mainState = {

    // We define the 3 default Phaser functions

    preload: function() {
        // This function will be executed at the beginning
        // That's where we load the game's assets
    },

    create: function() {
        // This function is called after the preload function
        // Here we set up the game, display sprites, etc.
    },

    update: function() {
        // This function is called 60 times per second
        // It contains the game's logic
    },

    // And here we will later add some of our own functions
};
```

The `preload`, `create` and `update` functions are key to any Phaser project, so make sure to read the comments above to understand what they do. We will spend most of our time in these 3 functions to create our game.



Then, we initialise Phaser with `Phaser.Game`.

`Phaser.Game(gameWidth, gameHeight, renderer, htmlElement)`

- gameWidth: width of the game in pixels
- gameHeight: height of the game in pixels
- renderer: how to render the game, I recommend using `Phaser.AUTO` that will automatically choose the best option between webGL and canvas
- htmlElement: the ID of the HTML element where the game will be displayed

For our game, we add this below the previous code:

```
// Create a 500px by 340px game in the 'gameDiv' element of the index.html
var game = new Phaser.Game(500, 340, Phaser.AUTO, 'gameDiv');
```

Next, we tell Phaser to add our only state:

```
// Add the 'mainState' to Phaser, and call it 'main'
game.state.add('main', mainState);
```

And finally, we start our 'main' state:

```
game.state.start('main');
```

Our empty project is now done. But let's add a couple of things in it that are going to be useful.

Background Color

By default, the background color of the game is black. We can easily change that by adding this line of code in the `create` function:

```
game.stage.backgroundColor = '#3498db';
```

The `#3498db` is the hexadecimal code for a blue color.

Physics Engine

One of the great features of Phaser is that it has 3 physics engines included. A physics engine is what will manage the collisions and movements of all the objects in the game.

The 3 engines available are:

- P2. It's a full featured physics system that lets us build games with complex collisions, like Angry Birds.
- Ninja. It's less powerful than P2, but still has some interesting features to handle tilemaps and slopes.
- Arcade. It's the most basic system that only deals with rectangle collisions (called AABB), but it also has the best performance.

Which one is the best? It really depends on what type of game we want to build. In our case we will use Arcade physics, and to tell that to Phaser we just need this line of code in the `create` function:

```
    game.physics.startSystem(Phaser.Physics.ARCADE);
```

Conclusion

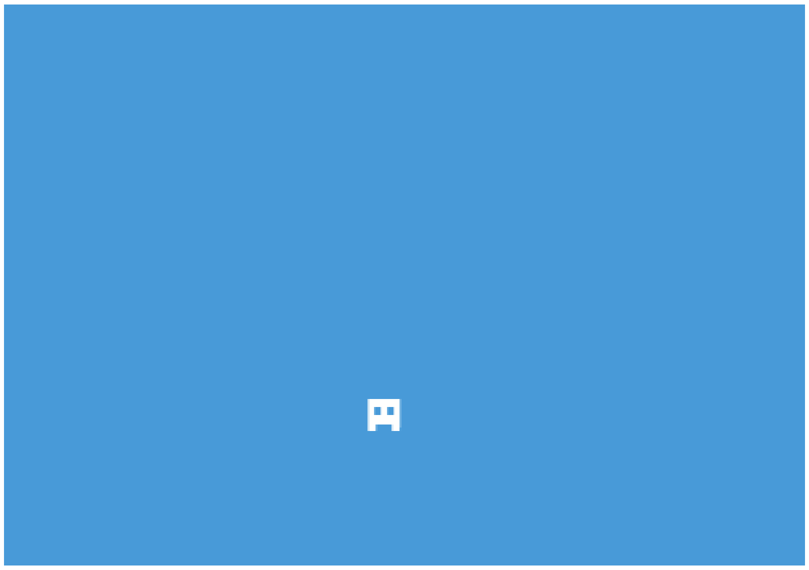
Once you put all of the Javascript code in the main.js file, you can test the game by either:

- Using the live preview of Brackets
- Or accessing the “first-game” folder from your webserver

If you see an empty blue screen, it means that everything is working properly.

3.2 - Add the Player

The first interesting thing we are going to add to the game is the player, and a way to control it. To do so, we will make some changes to the main.js file.



Load the Player

In Phaser, every time we want to use an asset (image, sound, etc.) we first need to load it. For an image, we can do that with the `game.load.image` function.

```
game.load.image(imageName, imagePath)
```

- imageName: the new name that will be used to reference the image
- imagePath: the path to the image

To load the player sprite, we add this in the `preload` function:

```
game.load.image('player', 'assets/player.png');
```

Display the Player

Once the sprite is loaded, we can display it on the screen with `game.add.sprite`.

```
game.add.sprite(positionX, positionY, imageName)
```

- positionX: horizontal position of the sprite
- positionY: vertical position of the sprite
- imageName: the name of the image, as defined in the preload function

Note that if we add a sprite at the 0, 0 position, it will be displayed in the top left corner of the game.

To add the sprite at the center of the screen we could write this in the `create` function:

```
// Create a local variable
var player = game.add.sprite(250, 170, 'player');
```

However, since we want to be able to use the player everywhere in our state, we need to use the `this` keyword:

```
// Create a state variable
this.player = game.add.sprite(250, 170, 'player');
```

And we can do even better by using some predefined variables for the x and y positions:

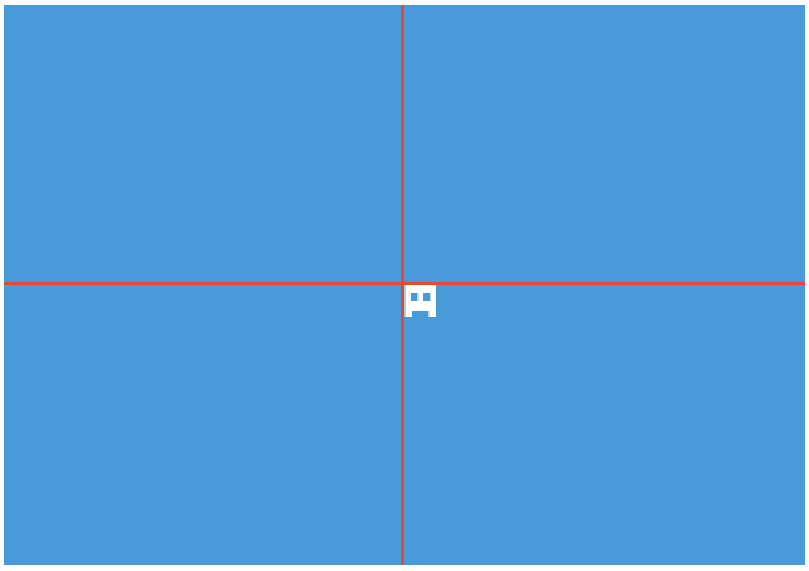
```
this.player = game.add.sprite(game.world.centerX, game.world.centerY, 'player');
```

That's the line we should actually add in the `create` function.

Some other useful Phaser variables include: `game.world.width`, `game.world.height`, `game.world.randomX`, `game.world.randomY`. They should be self explanatory.

Anchor Point

If you test the game, you might notice that the player is not exactly centered. That's because the `x` and `y` we set in `game.add.sprite` is the position of the top left corner of the sprite, also called the anchor point. So it's the top left corner of the player that is centered, and that's not what we want.



To fix that, we will need to change the anchor point's position. Here are some examples of how we can do that:

```
// Set the anchor point to the top left of the sprite (default value)
this.player.anchor.setTo(0, 0);

// Set the anchor point to the top right of the sprite
this.player.anchor.setTo(1, 0);

// Set the anchor point to the bottom left of the sprite
this.player.anchor.setTo(0, 1);

// Set the anchor point to the bottom right of the sprite
this.player.anchor.setTo(1, 1);
```

To center the player, we need to set the anchor point to the middle of the sprite, in the `create` function:

```
this.player.anchor.setTo(0.5, 0.5);
```

Add Gravity to the Player

Let's add some gravity to the player, to make it fall. To do so, we need to add this in the `create` function:

```
// Tell Phaser that the player will use the Arcade physics engine
game.physics.arcade.enable(this.player);

// Add vertical gravity to the player
this.player.body.gravity.y = 500;
```

Adding Arcade physics to the player is really important, it will allow us to use its `body` property to:

- Add gravity to the sprite to make it fall (see above)
- Add velocity to the sprite to be able to move it (see below)
- Add collisions (see in the next part)

Control the Player

There are a couple of things that need to be done if we want to move the player around with the arrow keys.

First, we have to tell Phaser which keys we want to use in our game. For the arrow keys, we simply add this in the `create` function:

```
this.cursor = game.input.keyboard.createCursorKeys();
```

And thanks to `this.cursor`, we can now add a new function (just after the `update`) that will handle all the player's movements:

```
movePlayer: function() {
    // If the left arrow key is pressed
    if (this.cursor.left.isDown) {
        // Move the player to the left
        this.player.body.velocity.x = -200;
    }

    // If the right arrow key is pressed
    else if (this.cursor.right.isDown) {
        // Move the player to the right
        this.player.body.velocity.x = 200;
    }

    // If neither the right or left arrow key is pressed
    else {
        // Stop the player
        this.player.body.velocity.x = 0;
    }

    // If the up arrow key is pressed and the player is touching the ground
    if (this.cursor.up.isDown && this.player.body.touching.down) {
        // Move the player upward (jump)
        this.player.body.velocity.y = -320;
    }
},
```

In Phaser, the velocity is expressed in pixels per second.

And finally, we have to call `movePlayer` inside the `update` function:

```
this.movePlayer();
```

This way, we check 60 times per second if an arrow key is pressed, and move the player accordingly.

More About Sprites

For your information, a sprite has a lot of interesting parameters, and here are the main ones:

```
// Change the position of the sprite
sprite.x = 50;
sprite.y = 50;

// Return the width and height of the sprite
sprite.width;
sprite.height;

// Change the transparency of the sprite. 0 = invisible, 1 = normal
sprite.alpha = 0.5;

// Change the angle of the sprite, in degrees
sprite.angle = 42;

// Remove the sprite from the game
sprite.kill();
```

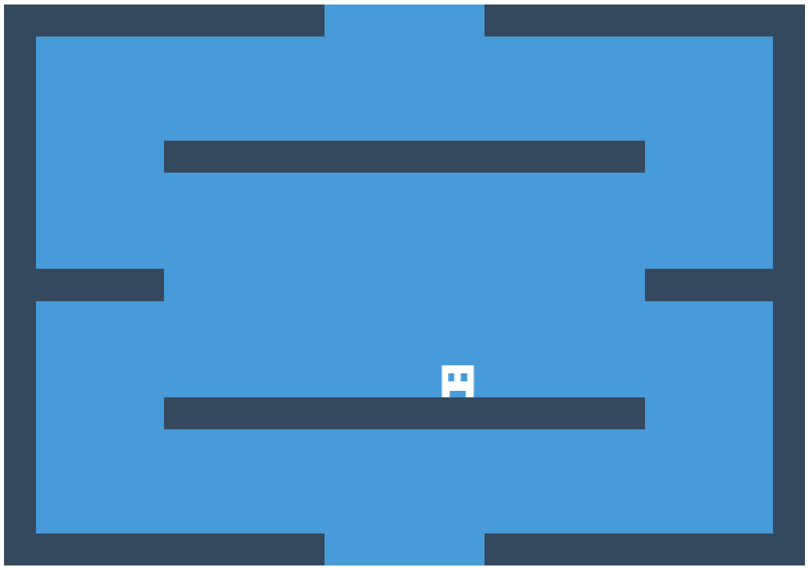
Conclusion

As usual, you should check that everything is working as expected. If so, you will be able to control the player while falling, and see him disappear from the screen.

If something is not working, you can get some help by looking at the finished source code at the end of this chapter.

3.3 - Create the World

Having a player falling is nice, but it would be better if there was a world in which he could move around. That's what we are going to fix in this part by doing this:



Load the Walls

With 2 sprites (an horizontal and a vertical wall) added at different locations, we will be able to create the whole new world.

As we explained previously, we need to start by loading our assets in the `preload` function:

```
game.load.image('wallV', 'assets/wallVertical.png');
game.load.image('wallH', 'assets/wallHorizontal.png');
```

You can see that the name of the image doesn’t have to be the same as its filename.

Add the Walls - Idea

Let’s create the left and right walls of the game:

```
// Create the left wall
var leftWall = game.add.sprite(0, 0, 'wallV');

// Add Arcade physics to the wall
game.physics.arcade.enable(leftWall);

// Set a property to make sure the wall won't move
// We don't want to see the wall fall when the player touches it
leftWall.body.immovable = true;

// Do the same for the right wall
var rightWall = game.add.sprite(480, 0, 'wallV');
game.physics.arcade.enable(rightWall);
rightWall.body.immovable = true;
```

That’s 6 lines of code for just 2 walls, so if we do this for the 10 walls it will quickly become messy. To avoid that we can use a Phaser feature called groups, which let us group objects together that can easily share some properties. Here’s how it works for our 2 walls:

```
// Create a new group
this.walls = game.add.group();

// Add Arcade physics to the whole group
this.walls.enableBody = true;

// Create 2 walls in the group
game.add.sprite(0, 0, 'wallV', 0, this.walls); // Left wall
game.add.sprite(480, 0, 'wallV', 0, this.walls); // Right wall
```

```
    // Set all the walls to be immovable
    this.walls.setAll('body.immovable', true);
```

You may notice that the `game.add.sprite` has 2 new optional parameters. It's the last one that's interesting to us: the name of the group to add the sprite in.

Add the Walls - Code

Adding walls is not very interesting, all we have to do is to create them at the correct positions. Here's the full code that does just that, in a new function:

```
createWorld: function() {
    // Create our wall group with Arcade physics
    this.walls = game.add.group();
    this.walls.enableBody = true;

    // Create the 10 walls
    game.add.sprite(0, 0, 'wallV', 0, this.walls); // Left
    game.add.sprite(480, 0, 'wallV', 0, this.walls); // Right

    game.add.sprite(0, 0, 'wallH', 0, this.walls); // Top left
    game.add.sprite(300, 0, 'wallH', 0, this.walls); // Top right
    game.add.sprite(0, 320, 'wallH', 0, this.walls); // Bottom left
    game.add.sprite(300, 320, 'wallH', 0, this.walls); // Bottom right

    game.add.sprite(-100, 160, 'wallH', 0, this.walls); // Middle left
    game.add.sprite(400, 160, 'wallH', 0, this.walls); // Middle right

    var middleTop = game.add.sprite(100, 80, 'wallH', 0, this.walls);
    middleTop.scale.setTo(1.5, 1);
    var middleBottom = game.add.sprite(100, 240, 'wallH', 0, this.walls);
    middleBottom.scale.setTo(1.5, 1);

    // Set all the walls to be immovable
    this.walls.setAll('body.immovable', true);
},
```

Note that for the last 2 walls we had to scale up their width with `sprite.scale.setTo(1.5, 1)`. The first parameter is the x scale (1.5 = 150%), the second is the y scale (1 = 100% = no change).

And we should not forget to call `createWorld` in the `create` function:

```
this.createWorld();
```

Collisions

If you test the game, you will probably see that there is a problem: the player is going through the walls. We can solve that by adding a single line of code at the beginning of the the `update` function:

```
// Tell Phaser that the player and the walls should collide
game.physics.arcade.collide(this.player, this.walls);
```

This works because we previously enabled Arcade physics for both the player and the walls. However, be careful to always add the collisions at the beginning of the `update` function, otherwise it might cause some bugs.

For fun, you can try to remove the `this.walls.setAll('body.immovable', true)` line, to see what happens. Hint: it's chaos.

Restart the Game

If the player dies by going into the bottom or top hole, nothing happens. Wouldn't it be great to have the game restart? Let's try to do that.

We create a new function `playerDie` that will restart the game by simply starting the 'main' state:

```
playerDie: function() {
    game.state.start('main');
},
```

And in the `update` function, we check if the player is in the world. If not, it means that the player has disappeared in one of the holes, so we will call `playerDie`.

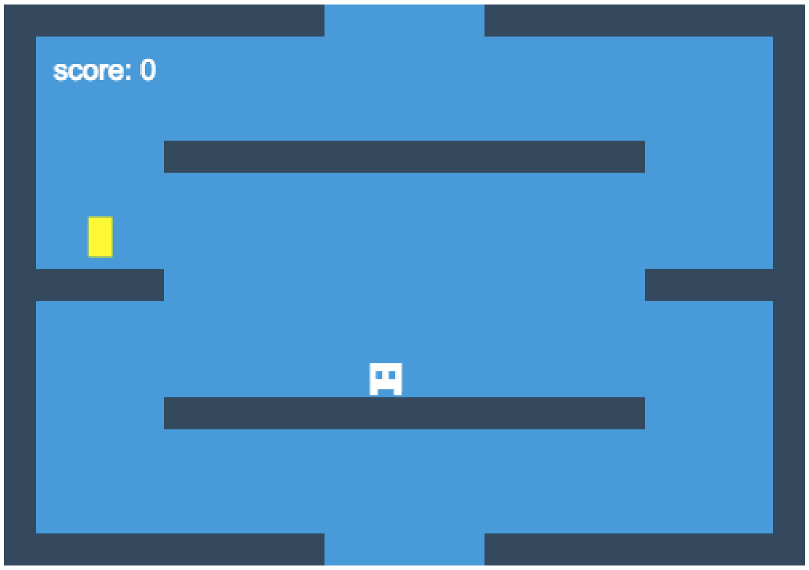
```
    if (!this.player.inWorld) {
        this.playerDie();
    }
```

Conclusion

If you test the game, you should be able to jump on the platforms, run around, and die in the holes. This is starting to look like a real game, and that's just the beginning.

3.4 - Add Coins

In this part we are going to give a goal to the player: collect coins. There will be only one coin in the game, and each time the player takes it, we will move it to another place. Let’s see how we can do that.



Load and Add the Coin

As usual, we start by loading the new sprite in the `preload` function:

```
game.load.image('coin', 'assets/coin.png');
```

And we add the coin in the `create` function:

```
    // Display the coin
    this.coin = game.add.sprite(60, 140, 'coin');

    // Add Arcade physics to the coin
    game.physics.arcade.enable(this.coin);

    // Set the anchor point of the coin to its center
    this.coin.anchor.setTo(0.5, 0.5);
```

This should look familiar to you by now.

Display the Score

Adding coins also means adding a score. To display a text on the screen we have to use `game.add.text`.

```
game.add.text(positionX, positionY, text, style)
```

- positionX: position x of the text
- positionY: position y of the text
- text: text to display
- style: style of the text

We can add the score in the top left corner of the game like this, in the `create` function:

```
    // Display the score
    this.scoreLabel = game.add.text(30, 30, 'score: 0',
        { font: '18px Arial', fill: '#ffffff' });

    // Initialise the score variable
    this.score = 0;
```

I kept things simple for the style of the text, but there are other properties we could have used: `fontWeight`, `align`, `stroke`, etc.

Collisions

In the previous part we used `game.physics.arcade.collide` for the collisions. However, this time the player doesn't need to walk on the coins, we just want to know when they overlap. That's why we are going to use `game.physics.arcade.overlap` instead.

```
game.physics.arcade.overlap(objectA, objectB, callback,
processCallback, callbackContext)
```

- objectA: the first object to check
- objectB: the second object to check
- callback: the function that gets called when the 2 objects overlap
- processCallback: if this is set then 'callback' will only be called if 'processCallback' returns true
- callbackContext: the context in which to run the 'callback', most of the time it will be 'this'

Each time a function has a 'callback', there will be a 'callbackContext' parameter. In this book we will always set the 'callbackContext' to 'this', in order to be able to use any of the state's variables in the 'callback' function.

In our case, we want to call `takeCoin` each time the player and a coin overlap, so we should add this in the `update` function:

```
game.physics.arcade.overlap(this.player, this.coin, this.takeCoin, null, this);
```

And now we create the new `takeCoin` function:

```
takeCoin: function(player, coin) {
    // Kill the coin to make it disappear from the game
    this.coin.kill();

    // Increase the score by 5
    this.score += 5;

    // Update the score label
    this.scoreLabel.text = 'score: ' + this.score;
    },
```

Note that this function has 2 parameters: `player` and `coin`. They are the 2 overlapped objects that are automatically sent by the `game.physics.arcade.overlap` function.

And as you can see, to edit a label we have to use its `text` property.

Move the Coin - Idea

Instead of killing the coin when the player takes it, we want to move it to another position. To do so we can use 2 handy functions:

```
// Return a random integer between a and b
var number = game.rnd.integerInRange(a, b);

// Change the coin's position to x, y
this.coin.reset(x, y);
```

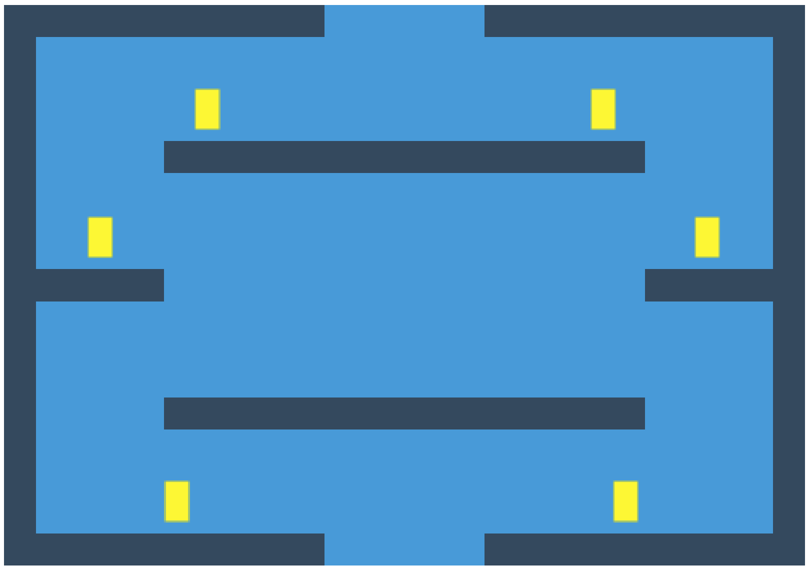
So we could replace the `this.coin.kill()` in the `takeCoin` function by something like this:

```
// Define 2 random variables
var newX = game.rnd.integerInRange(10, game.world.width - 20);
var newY = game.rnd.integerInRange(10, game.world.height - 20);

// Set the new coin position
this.coin.reset(newX, newY);
```

However, the result would not be great because the coin could appear in the walls or at some inaccessible spot. We need to find a better solution.

Move the Coin - Code

We are going to manually define 6 positions where the coin can appear, and randomly choose one of them.



Here’s a new function that does just that:

```
updateCoinPosition: function() {
    // Store all the possible coin positions in an array
    var coinPosition = [
        {x: 140, y: 60}, {x: 360, y: 60}, // Top row
        {x: 60, y: 140}, {x: 440, y: 140}, // Middle row
        {x: 130, y: 300}, {x: 370, y: 300} // Bottom row
    ];

    // Remove the current coin position from the array
    // Otherwise the coin could appear at the same spot twice in a row
    for (var i = 0; i < coinPosition.length; i++) {
        if (coinPosition[i].x === this.coin.x) {
            coinPosition.splice(i, 1);
        }
    }

    // Randomly select a position from the array
    var newPosition = coinPosition[
        game.rnd.integerInRange(0, coinPosition.length-1)];

    // Set the new position of the coin
    this.coin.reset(newPosition.x, newPosition.y);
},
```

The `for` loop in the middle might be a little hard to grasp at first sight, so here's what it does in detail:

- We start with the `coinPosition` array containing the 6 possible coin positions
- For each `coinPosition`, we check if its `x` position is the same as the coin
- If so, we remove the position from the array with the `coinPosition.splice(i, 1)` line
- This way, when the loop is over we have only 5 positions left in the `coinPosition` array, that are all different from where the current coin is

And finally, we edit the `takeCoin` function like this:

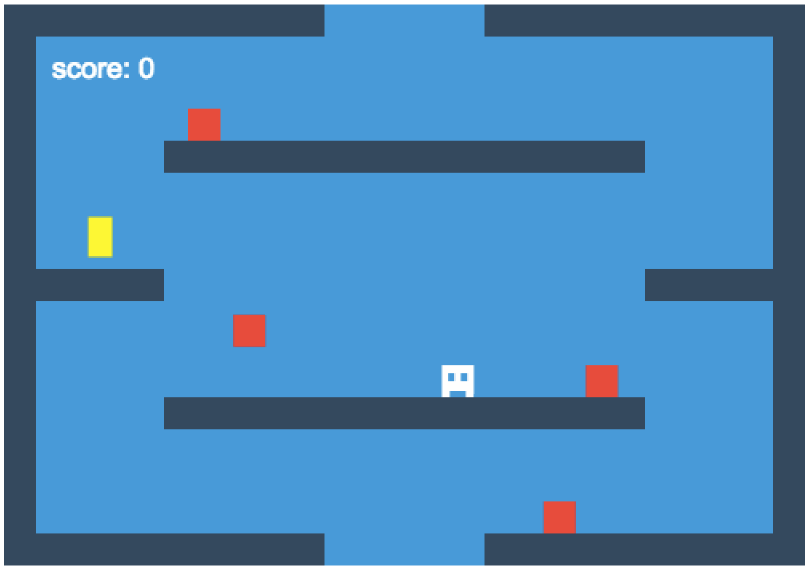
```
takeCoin: function(player, coin) {
    // Update the score
    this.score += 5;
    this.scoreLabel.text = 'score: ' + this.score;

    // Change the coin position
    this.updateCoinPosition();
},
```

Conclusion

Now you can try to collect some coins, and see your score increase.

3.5 - Add Enemies

For the last part of this chapter we are going to add enemies into the game, this way collecting coins will become more challenging.



Load the Enemy

Again, we start by loading the sprite in the `preload` function:

```
game.load.image('enemy', 'assets/enemy.png');
```

Enemy Group

Since we will deal with multiple enemies, we will use groups as we did with the walls. However, this time we can do better by doing some "recycling". Let me explain.

Instead of creating new enemies one by one as we need them, we can create a bunch of them in advance. And each time an enemy dies, we will be able to reuse it instead of erasing it from the memory. It's quite simple to do code-wise and it can greatly improve the game's performance.

Here's what we should add in the `create` function:

```
// Create an enemy group with Arcade physics
this.enemies = game.add.group();
this.enemies.enableBody = true;

// Create 10 enemies with the 'enemy' image in the group
// The enemies are "dead" by default, so they are not visible in the game
this.enemies.createMultiple(10, 'enemy');
```

Creating 10 enemies means that we will never see more than 10 of them at the same time. If you plan to have more enemies, simply increase that number. But a bigger number can decrease performance, so try to be reasonable.

Add the Enemies - Code

We want new enemies to appear every few seconds. For this we can use `game.time.events.loop`.

`game.time.events.loop(delay, callback, callbackContext)`

- delay: the delay in ms between each ‘callback’
- callback: the function that will be called
- callbackContext: the context in which to run the ‘callback’, most of the time it will be ‘this’

Here’s how we can add our event loop in the `create` function:

```
// Call 'addEnemy' every 2.2 seconds
game.time.events.loop(2200, this.addEnemy, this);
```

Next, we can create the new `addEnemy` function:

```
    addEnemy: function() {
        // Get the first dead enemy of the group
        var enemy = this.enemies.getFirstDead();

        // If there isn't any dead enemy, do nothing
        if (!enemy) {
            return;
        }

        // Initialise the enemy
        enemy.anchor.setTo(0.5, 1);
        enemy.reset(game.world.centerX, 0);
        enemy.body.gravity.y = 500;
        enemy.body.velocity.x = 100 * Phaser.Math.randomSign();
        enemy.body.bounce.x = 1;
        enemy.checkWorldBounds = true;
        enemy.outOfBoundsKill = true;
    },
```

There are a lot of new things here that are quite important, so we should spend some time to study them.

Add the Enemies - Explained

Let’s see what the `addEnemy` function does, line by line.

Earlier we created 10 dead enemies in the `this.enemies` group. We are going to pick one of them, and store it in a new `enemy` variable.

```
    var enemy = this.enemies.getFirstDead();
```

If there are no dead enemies available (they are all already displayed in the game), it means that we can’t add a new one. If so, we should stop the function with a `return` to prevent the game from crashing.

```
    if (!enemy) {
        return;
    }
```

At this point in the function, we are sure that the `enemy` variable contains a new enemy. All we have to do now is to initialise it.

We want the enemy to appear falling from the top hole:

```
    // Set the anchor point centered at the bottom
    enemy.anchor.setTo(0.5, 1);

    // Put the enemy above the top hole
    enemy.reset(game.world.centerX, 0);

    // Add gravity to see it fall
    enemy.body.gravity.y = 500;
```

We give some horizontal velocity to the enemy to make it move right or left. We use `Phaser.Math.randomSign()` that will randomly return 1 or -1, to have a velocity of 100 or -100:

```
    enemy.body.velocity.x = 100 * Phaser.Math.randomSign();
```

When an enemy is moving right and hits a wall, we want it to start moving left. One easy way to do this is to use the `bounce` property that can be set with a value between 0 and 1. 0 means no bounce, 1 means a perfect bounce. So this will make the enemy change direction when hitting a wall horizontally:

```
    enemy.body.bounce.x = 1;
```

And finally, these 2 lines will automatically kill the sprite when it's no longer in the world (when it falls into the bottom hole). This way we should never run out of dead enemies for `getFirstDead`.

```
    enemy.checkWorldBounds = true;
    enemy.outOfBoundsKill = true;
```

To summarise what the `addEnemy` function does:

1. Get a dead sprite from the group
2. If there isn't any dead sprite, do nothing
3. Initialise the sprite: position, physics values, autokill

It's quite common to have functions like this in a Phaser game to create enemies, bullets, bonuses, and so on.

Collisions

For the collisions, we want to do 2 things:

- The enemies should walk on the walls
- The player should die if it overlaps with an enemy

And we already know how to do both of these things, by simply adding the following lines in the `update` function:

```
// Make the enemies and walls collide
game.physics.arcade.collide(this.enemies, this.walls);

// Call the 'playerDie' function when the player and an enemy overlap
game.physics.arcade.overlap(this.player, this.enemies, this.playerDie,
    null, this);
```

More About Groups

We are done adding enemies to the game, but I wanted to show you more things you can do with groups:

```
// A group can act like a giant sprite, so it has the same properties you can edit
group.x;
group.y;
group.alpha;
group.angle;

// Return the number of living members in the group
group.countLiving();
```

Conclusion

You now have a real game to play with: controlling a player to collect coins while avoiding enemies. And all of this in just 130 lines of Javascript.

But this is not a really interesting game for now, that’s why the book is not over yet.

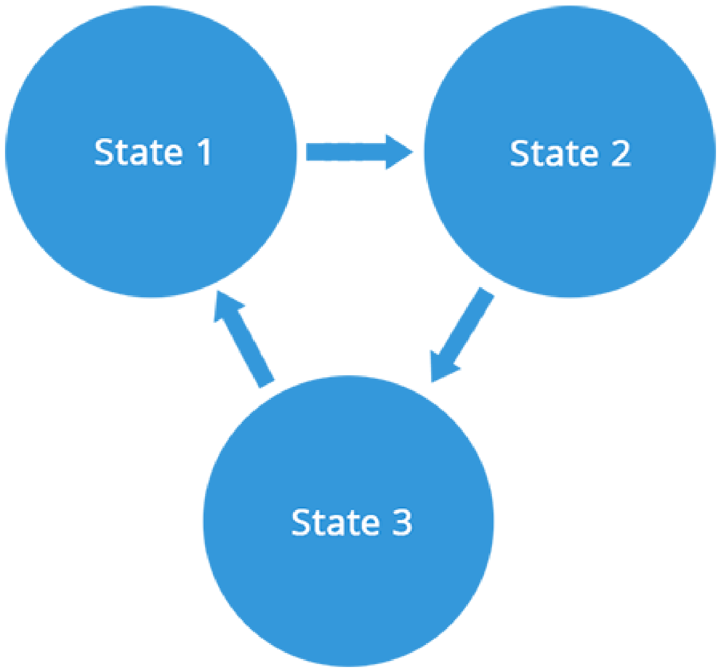
3.6 - Source Code

Here's the full source code of the game we've created so far.

```
var mainState = {

    preload: function() {
        game.load.image('player', 'assets/player.png');
        game.load.image('wallV', 'assets/wallVertical.png');
        game.load.image('wallH', 'assets/wallHorizontal.png');
        game.load.image('coin', 'assets/coin.png');
        game.load.image('enemy', 'assets/enemy.png');
    },

    create: function() {
        game.stage.backgroundColor = '#3498db';
        game.physics.startSystem(Phaser.Physics.ARCADE);
        this.cursor = game.input.keyboard.createCursorKeys();

        this.player = game.add.sprite(game.world.centerX, game.world.centerY,
            'player');
        this.player.anchor.setTo(0.5, 0.5);
        game.physics.arcade.enable(this.player);
        this.player.body.gravity.y = 500;

        this.enemies = game.add.group();
        this.enemies.enableBody = true;
        this.enemies.createMultiple(10, 'enemy');

        this.coin = game.add.sprite(60, 140, 'coin');
        game.physics.arcade.enable(this.coin);
        this.coin.anchor.setTo(0.5, 0.5);

        this.scoreLabel = game.add.text(30, 30, 'score: 0',
            { font: '18px Arial', fill: '#ffffff' });
        this.score = 0;

        this.createWorld();
        game.time.events.loop(2200, this.addEnemy, this);
    },
    update: function() {
        game.physics.arcade.collide(this.player, this.walls);
        game.physics.arcade.collide(this.enemies, this.walls);
        game.physics.arcade.overlap(this.player, this.coin, this.takeCoin,
            null, this);
        game.physics.arcade.overlap(this.player, this.enemies, this.playerDie,
            null, this);

        this.movePlayer();

        if (!this.player.inWorld) {
            this.playerDie();
        }
    },

    movePlayer: function() {
        if (this.cursor.left.isDown) {
            this.player.body.velocity.x = -200;
        }
        else if (this.cursor.right.isDown) {
            this.player.body.velocity.x = 200;
        }
        else {
            this.player.body.velocity.x = 0;
        }

        if (this.cursor.up.isDown && this.player.body.touching.down) {
            this.player.body.velocity.y = -320;
        }
    },

    takeCoin: function(player, coin) {
        this.score += 5;
        this.scoreLabel.text = 'score: ' + this.score;

        this.updateCoinPosition();
    },

    updateCoinPosition: function() {
        var coinPosition = [
            {x: 140, y: 60}, {x: 360, y: 60},
            {x: 60, y: 140}, {x: 440, y: 140},
            {x: 130, y: 300}, {x: 370, y: 300}
        ];

        for (var i = 0; i < coinPosition.length; i++) {
            if (coinPosition[i].x === this.coin.x) {
                coinPosition.splice(i, 1);
            }
        }

        var newPosition = coinPosition[game.rnd.integerInRange(0,
            coinPosition.length-1)];
        this.coin.reset(newPosition.x, newPosition.y);
    },

    addEnemy: function() {
        var enemy = this.enemies.getFirstDead();
        if (!enemy) {
            return;
        }

        enemy.anchor.setTo(0.5, 1);
        enemy.reset(game.world.centerX, 0);
        enemy.body.gravity.y = 500;
        enemy.body.velocity.x = 100 * Phaser.Math.randomSign();
        enemy.body.bounce.x = 1;
        enemy.checkWorldBounds = true;
        enemy.outOfBoundsKill = true;
    },

    createWorld: function() {
        this.walls = game.add.group();
        this.walls.enableBody = true;

        game.add.sprite(0, 0, 'wallV', 0, this.walls);
        game.add.sprite(480, 0, 'wallV', 0, this.walls);
        game.add.sprite(0, 0, 'wallH', 0, this.walls);
        game.add.sprite(300, 0, 'wallH', 0, this.walls);
        game.add.sprite(0, 320, 'wallH', 0, this.walls);
        game.add.sprite(300, 320, 'wallH', 0, this.walls);
        game.add.sprite(-100, 160, 'wallH', 0, this.walls);
        game.add.sprite(400, 160, 'wallH', 0, this.walls);

        var middleTop = game.add.sprite(100, 80, 'wallH', 0, this.walls);
        middleTop.scale.setTo(1.5, 1);
        var middleBottom = game.add.sprite(100, 240, 'wallH', 0, this.walls);
        middleBottom.scale.setTo(1.5, 1);

        this.walls.setAll('body.immovable', true);
    },

    playerDie: function() {
        game.state.start('main');
    },
};

var game = new Phaser.Game(500, 340, Phaser.AUTO, 'gameDiv');

game.state.add('main', mainState);
game.state.start('main');
```

4 - Manage States

In this chapter we will continue to work on the same game. This time our main focus will be to create states, it means that we will have a nice loading scene, a cool menu, and the game itself.



4.1 - Organisation

Currently our game has only one state, it means that:

- We need to start playing as soon as the game loads
- Each time the game restarts, all assets are re-loaded
- We don't have any menu to display the game's name, controls, or score

All of this is not great, so we should try to fix these problems.

What is a State

A state in Phaser is a part of a game, like a menu scene, a play scene, a game over scene, etc. And each one of them has its own functions and variables.

It means that if you have a `this.player` variable in one state, you won't be able to change it in another state. The only 2 things that are shared between states are the preloaded assets (so once a sprite is loaded you can use it everywhere) and global variables.

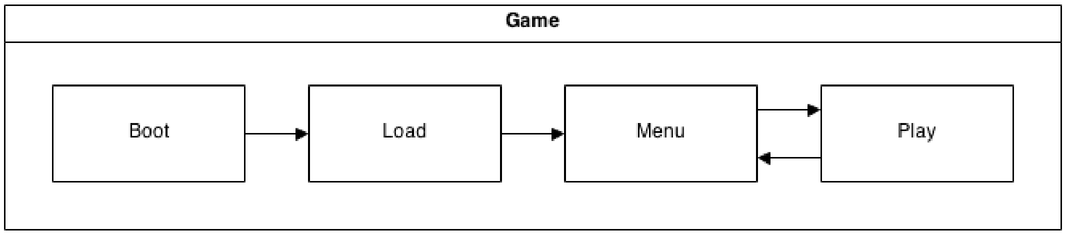
You can see below what a state looks like. It should be familiar to you, since that's what we used to build the game we have so far.

```
// Create one state called 'oneState'
var oneState = {

    // Define all the functions of the state

    preload: function() {
        // This function will be executed at the beginning
        // That's where we load the game's assets
    },

    create: function() {
        // This function is called after the preload function
        // Here we set up the game, display sprites, etc.
    },

    update: function() {
        // This function is called 60 times per second
        // It contains the game's logic
    },

    // And maybe add some other functions
};
```

In this chapter, we are going to use this code multiple times to create all our states.

New States

Our game will now have 4 states, organised as shown below:



- Boot. This is the first state of the game.
- Load. This state will load all the game's assets.
- Menu. That's the menu of the game.
- Play. This is where the actual game will be played. When dying, the game will go back to the menu.

File Structure

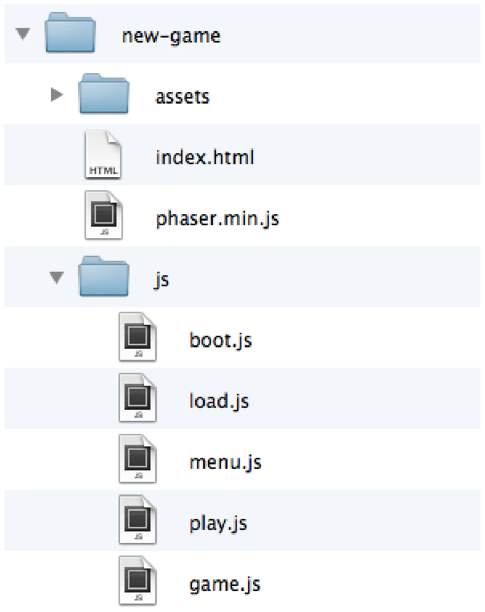
Since we are going to change a lot of things in this chapter, it's easier to start from scratch. So we create a new folder that contains:

- phaser.min.js, the Phaser framework.
- index.html, that will display the game. For now it's just an empty file.
- assets/, a directory with all the images and sounds. We are using the same assets that you can still **download here**.
- js/, a directory that will contain all our Javascript files.

And in the "js" folder, we create these 5 empty Javascript files:

- boot.js, the boot state

- load.js, the load state
- menu.js, the menu state
- play.js, the play state
- game.js, that will initialise all of our states



4.2 - Index

The first thing we are going to do is to code the index.html file. The code is basically the same we used so far. The only difference is that we are loading multiple Javascript files from the "js" directory.

```html
<!DOCTYPE html>
<html>

    <head>
        <meta charset="utf-8" />
        <title> First game</title>

        <script type="text/javascript" src="phaser.min.js"></script>
        <script type="text/javascript" src="js/boot.js"></script>
        <script type="text/javascript" src="js/load.js"></script>
        <script type="text/javascript" src="js/menu.js"></script>
        <script type="text/javascript" src="js/play.js"></script>
        <script type="text/javascript" src="js/game.js"></script>
    </head>

    <body>
        <p> My first Phaser game </p>
        <div id="gameDiv"> </div>
    </body>

</html>
```

4.3 - Boot

The boot state will be the first state of our game. We need it for 2 reasons:

- We want to show a nice loading bar in the load state, so we will need to display an image in the `preload` function of the load.js file. And the only way to make that work is to load the image before the load state, in the boot state.
- On top of that, it's good practice to have a state that can run some code before loading all the assets.

This state will be really short, since it only has 3 things to do:

- Load the 'progressBar' image
- Set some game settings: background color and Arcade physics
- Start the load state

We can do so like this:

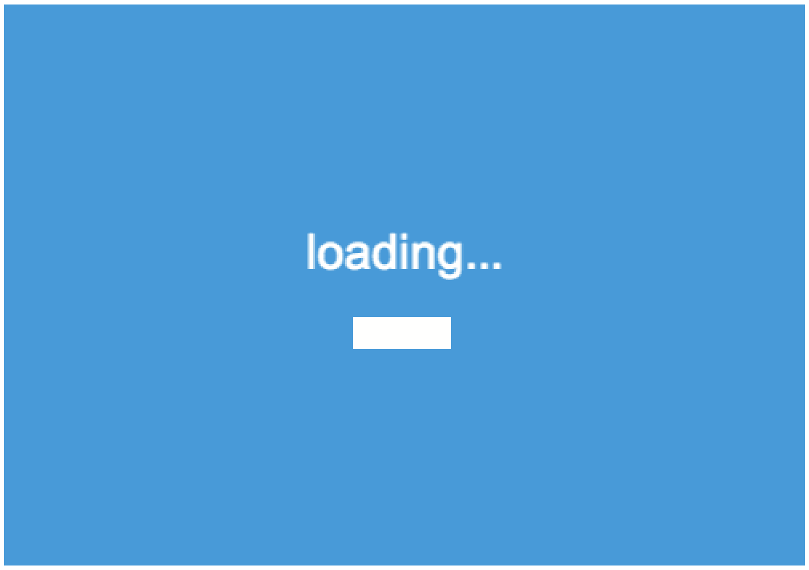
```
var bootState = {

    preload: function () {
        // Load the image
        game.load.image('progressBar', 'assets/progressBar.png');
    },

    create: function() {
        // Set some game settings
        game.stage.backgroundColor = '#3498db';
        game.physics.startSystem(Phaser.Physics.ARCADE);

        // Start the load state
        game.state.start('load');
    }
};
```

Notice that since we don't need the `update` function, we simply removed it.

You might be tempted to test the game right now but it won't work. We first have to create all our states and initialise Phaser, so you'll have to wait till the end of this chapter to play and test the game.

4.4 - Load

The load state is also quite simple. It will preload all the assets of the game, and displays the text “loading...” with a progress bar.



Since the game doesn’t have a lot of assets to load, you may not even have the time to see this state. But for people with a slow connection, it will be nice.

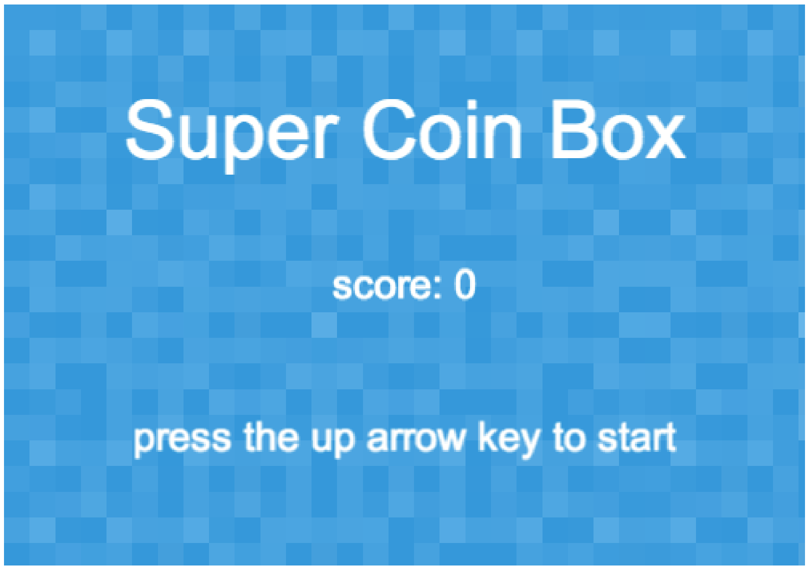
```
var loadState = {

    preload: function () {
        // Add a 'loading...' label on the screen
        var loadingLabel = game.add.text(game.world.centerX, 150, 'loading...',
            { font: '30px Arial', fill: '#ffffff' });
        loadingLabel.anchor.setTo(0.5, 0.5);

        // Display the progress bar
        var progressBar = game.add.sprite(game.world.centerX, 200, 'progressBar');
        progressBar.anchor.setTo(0.5, 0.5);
        game.load.setPreloadSprite(progressBar);

        // Load all our assets
        game.load.image('player', 'assets/player.png');
        game.load.image('enemy', 'assets/enemy.png');
```

```
            game.load.image('coin', 'assets/coin.png');
            game.load.image('wallV', 'assets/wallVertical.png');
            game.load.image('wallH', 'assets/wallHorizontal.png');

            // Load a new asset that we will use in the menu state
            game.load.image('background', 'assets/background.png');
        },

        create: function() {
            // Go to the menu state
            game.state.start('menu');
        }
    };
```

The only new thing in this code is the `game.load.setPreloadSprite` function. It will simply take care of scaling up the 'progressBar' progressively, as the game loads.

4.5 - Menu

The menu state is going to be more interesting, it will look like this:



Here’s the full source code below:

```
var menuState = {

    create: function() {
        // Add a background image
        game.add.image(0, 0, 'background');

        // Display the name of the game
        var nameLabel = game.add.text(game.world.centerX, 80, 'Super Coin Box',
            { font: '50px Arial', fill: '#ffffff' });
        nameLabel.anchor.setTo(0.5, 0.5);

        // Show the score at the center of the screen
        var scoreLabel = game.add.text(game.world.centerX, game.world.centerY,
            'score: ' + game.global.score,
            { font: '25px Arial', fill: '#ffffff' });
        scoreLabel.anchor.setTo(0.5, 0.5);

        // Explain how to start the game
        var startLabel = game.add.text(game.world.centerX, game.world.height-80,
            'press the up arrow key to start',
            { font: '25px Arial', fill: '#ffffff' });
        startLabel.anchor.setTo(0.5, 0.5);

        // Create a new Phaser keyboard variable: the up arrow key
        var upKey = game.input.keyboard.addKey(Phaser.Keyboard.UP);

        // When the 'upKey' is pressed, it will call the 'start' function once
        upKey.onDown.addOnce(this.start, this);
    },

    start: function() {
        // Start the actual game
        game.state.start('play');
    },
};
```

Most of this code should be pretty easy to follow, but here is some interesting information for you.

Background
For the background image, we used `game.add.image` instead of `game.add.sprite`. An image is like a lightweight sprite that doesn't need physics or animations. It's perfect for logos, backgrounds, etc.

Z-index
It's really important to add the background image at the beginning of the function, so everything that is created afterward will be added on top of the background. Otherwise the labels would be displayed below the image and they would not be visible. That's why the order of the code is actually quite important: the earlier an object is created, the lower it's z-index will be.

Score
We want to display the score in both the play state and the menu state. To do so, we need a global variable that can be shared between states. That's why we use `game.global.score` to store and display the score. This variable will be defined in the game.js file.

Key
We wait for the player to press the up arrow key to start the game. We could have used the `game.input.keyboard.createCursorKeys()` that we saw in the previous chapter, but since we only need to check for one key it's easier to use `game.input.keyboard.addKey`.

4.6 - Play

The play state is almost exactly the same as in the previous chapter. The only small changes we need to make are:

- The state is now called 'playState' instead of 'mainState'
- The preload function is no longer needed, since we already load all our assets in the load.js file
- The code to set the background color and Arcade physics are now in the boot.js file
- Instead of using the `this.score` variable, we now use `game.global.score`
- When the player dies, we want to go back to the menu state
- The Phaser initialisation and states definition will be done in the game.js file

I added comments on the things that changed, and removed the functions that stay the same:

```
// New name for the state
var playState = {

    // Removed the preload function

    create: function() {
        // Removed background color and physics system

        this.cursor = game.input.keyboard.createCursorKeys();

        this.player = game.add.sprite(game.world.centerX, game.world.centerY,
            'player');
        this.player.anchor.setTo(0.5, 0.5);
        game.physics.arcade.enable(this.player);
        this.player.body.gravity.y = 500;

        this.enemies = game.add.group();
        this.enemies.enableBody = true;
        this.enemies.createMultiple(10, 'enemy');

        this.coin = game.add.sprite(60, 140, 'coin');
        game.physics.arcade.enable(this.coin);
        this.coin.anchor.setTo(0.5, 0.5);

        this.scoreLabel = game.add.text(30, 30, 'score: 0',
            { font: '18px Arial', fill: '#ffffff' });

        // New score variable
        game.global.score = 0;

        this.createWorld();
        game.time.events.loop(2200, this.addEnemy, this);
    },

    // No changes

    takeCoin: function(player, coin) {
        // New score variable
        game.global.score += 5;
        this.scoreLabel.text = 'score: ' + game.global.score;

        this.updateCoinPosition();
    },

    // No changes

    playerDie: function() {
        // When the player dies, we go to the menu
        game.state.start('menu');
    },
};

// Removed Phaser and states initialisation
```

4.7 - Game

And finally, we need to initialise everything. We can do so like this in the game.js file:

```
// Initialise Phaser
var game = new Phaser.Game(500, 340, Phaser.AUTO, 'gameDiv');

// Define our 'global' variable
game.global = {
    score: 0
};

// Add all the states
game.state.add('boot', bootState);
game.state.add('load', loadState);
game.state.add('menu', menuState);
game.state.add('play', playState);

// Start the 'boot' state
game.state.start('boot');
```

Now you can test the game, and you should see all the new states in action. To summarise what is happening:

1. First, the boot state is called to load one image and set some settings
2. Then the load state is displayed to load all the game's assets
3. After that, the menu is shown
4. When the user presses the up arrow key, we start the play state
5. And finally, when the user dies, we go back to the menu

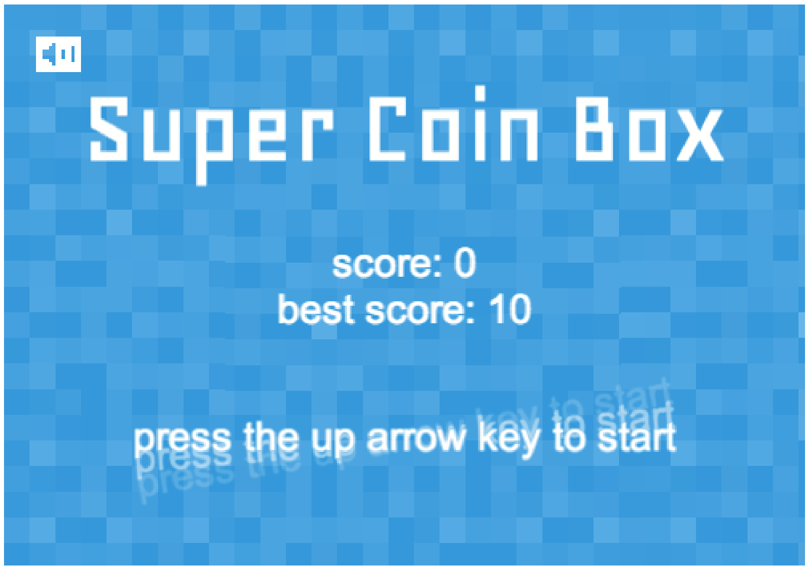
That's great, but once again the game feels like it can be improved. And that's what we are going to do in the next chapter.

5 - Various Improvements

Now that we have a solid base for our game, it's time to improve it. We won't change the game mechanics in this chapter, but we will add a lot of small things that will make the game feel better and be more responsive.

Some of the things we will cover include: adding animations, sound effects, best scores, custom fonts, etc. Needless to say, we are going to see a lot of new Phaser features.

Just be careful to add the code in the correct files, since we now have multiple states in our game.



5.1 - Add Sounds

We will start by adding sound effects to the game. And as you will see, this is quite simple.

Compatibility

There are a lot of audio formats we can use, the main ones are: wav, mp3, and ogg. So which one should we choose?

Each one has its pros and cons, but the most important thing to be aware of is their browser compatibility. Unfortunately, there isn't any format that works across all browsers for now:

	Chrome	Firefox	IE	Safari
wav	Yes	Yes	No	Yes
mp3	Yes	No	Yes	Yes
ogg	Yes	Yes	No	No

So we will need to use multiple audio formats to hear sounds on all browsers. For our game, we will have both mp3 and ogg files, which is considered a best practice.

Load the Sounds

As with the images, in order to use a sound we first need to load it. We can do so with `game.load.audio`.

We are going to load 3 sounds in the `preload` function of the load.js file:

```
// Sound when the player jumps
game.load.audio('jump', ['assets/jump.ogg', 'assets/jump.mp3']);

// Sound when the player takes a coin
game.load.audio('coin', ['assets/coin.ogg', 'assets/coin.mp3']);

// Sound when the player dies
game.load.audio('dead', ['assets/dead.ogg', 'assets/dead.mp3']);
```

You can see that we specified 2 different files for each sound. Phaser will be smart enough to use the correct one depending on the browser used.

Add the Sounds

The next step is to add the sounds to the game, in the `create` function of the play.js file:

```
this.jumpSound = game.add.audio('jump');
this.coinSound = game.add.audio('coin');
this.deadSound = game.add.audio('dead');
```

Play the Sounds

And finally, we want to actually play the sounds with the `play` function:

```
// Add this inside the 'movePlayer' function, in the 'if(player jumps)'
this.jumpSound.play();

// Put this in the 'takeCoin' function
this.coinSound.play();

// And this in the 'playerDie' function
this.deadSound.play();
```

You can now play the game and hear some nice sound effects.

More About Sounds

For you information, if you wanted to add a background music to the game, the process would be pretty much the same. Note that this is just a suggestion, there are no music files included in the assets of this book.

```
// Load the music in 2 different formats, in the load.js file
game.load.audio('music', ['assets/music.ogg', 'assets/music.mp3']);

// Add and start the music in the 'create' function of the play.js file
// Because we want to play the music when the play state starts
this.music = game.add.audio('music'); // Add the music
this.music.loop = true; // Make it loop
this.music.play(); // Start the music

// And don't forget to stop the music in the 'playerDie' function
// Otherwise the music would keep playing
this.music.stop();
```

However, be careful with the size of the music since it will probably be the biggest asset you will have. Games that take forever to load are something to avoid.

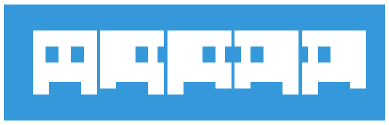
You should also know that it's possible to change the volume of any sound by doing `this.jumpSound.volume = 0.5` for example.

5.2 - Add Animations

Animations are an easy way to give life to sprites, so let's try to add some animations to the player.

Load the Player

Instead of just loading a simple image of the player, we are going to load a spritesheet. It's an image that contains the player in different positions. It looks like this (I tweaked the image for better readability):



On the far left you can see the frame `0` (stand still), then frames `1` & `2` (move right) and `3` & `4` (move left).

To load this image, we have to use `game.load.spritesheet`. It needs to know the width and height of the player, so Phaser can cut the spritesheet into the smaller frames.

We can replace `game.load.image('player', 'assets/player.png')` by this in the load.js file:

```
game.load.spritesheet('player', 'assets/player2.png', 20, 20);
```

Add the Animations

Whether we load a sprite with `game.load.image` or with `game.load.spritesheet`, it doesn't change the way to add a sprite in the game. However, we need to add the animations with the `animations.add` function.

`animations.add(name, frames, frameRate, loop)`

- name: name of the animation
- frames: an array of numbers that correspond to the frames to add and in which order
- frameRate: the speed at which the animation should play, in frames per second
- loop: if set to true, the animation will loop indefinitely

In the game we are going to add 2 different animations to the player in the `create` function of the play.js file:

```
// Create the 'right' animation by looping the frames 1 and 2
this.player.animations.add('right', [1, 2], 8, true);

// Create the 'left' animation by looping the frames 3 and 4
this.player.animations.add('left', [3, 4], 8, true);
```

Play the Animations

Now all we have to do is use `animations.play` to play the animations. We need to edit the `movePlayer` function like this:

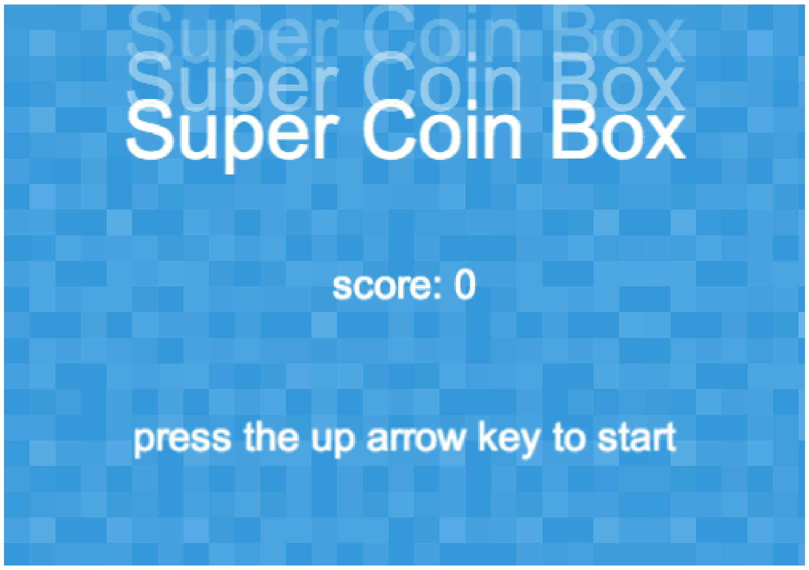
```
movePlayer: function() {
    // Move the player to the left
    if (this.cursor.left.isDown) {
        this.player.body.velocity.x = -200;
        this.player.animations.play('left'); // Start the left animation
    }

    // Move the player to the right
    else if (this.cursor.right.isDown) {
        this.player.body.velocity.x = 200;
        this.player.animations.play('right'); // Start the right animation
    }

    // Stop the player
    else {
        this.player.body.velocity.x = 0;
        this.player.animations.stop(); // Stop the animation
        this.player.frame = 0; // Set the player frame to 0 (stand still)
    }

    // Make the player jump
    if (this.cursor.up.isDown && this.player.body.touching.down) {
        this.player.body.velocity.y = -320;
        this.jumpSound.play();
        }
    },
```

And now we have a cute little guy running around.

5.3 - Add Tweens

Tweens are used all the time in games, they let us change objects' properties over time. We can move a sprite from A to B in X seconds, scale down a label smoothly, rotate a group indefinitely, etc.

We are going to see how they work by adding 4 tweens to our game.



Move the Label

We will start with something simple. In the menu, we want the 'nameLabel' to appear as if it was falling from the top of the screen.

So first thing, we need to set the label position above the game in the menu.js file:

```
// Changed the y position to -50, so we don't see the label
var nameLabel = game.add.text(game.world.centerX, -50, 'Super Coin Box',
    { font: '50px Arial', fill: '#ffffff' });
```

And now we can do the actual tweening:

```
// Create a tween on the label
var tween = game.add.tween(nameLabel);

// Change the y position of the label to 80, in 1000 ms
tween.to({y: 80}, 1000);

// Start the tween
tween.start();
```

The code above will move the label from its initial position (y = -50) to its new position (y = 80) in 1 second. These lines can be combined into one like this:

```
game.add.tween(nameLabel).to({y: 80}, 1000).start();
```

By default, a tween is moving the object in a straight line at a constant speed. We can change that by adding what we call an easing function:

```
game.add.tween(nameLabel).to({y: 80}, 1000).easing(Phaser.Easing.Bounce.Out)
    .start();
```

That's the line we should add in the `create` function of the menu.js file, and now the label will appear like it's falling and bouncing.

Rotate the Label

This time, we are going to tween the 'startLabel' of the menu. We want to rotate it slightly left and right indefinitely, to give it more emphasis. That's possible by using the `angle` property of the sprite, and the `loop` function like this:

```
    // Create the tween
    var tween = game.add.tween(startLabel);

    // Rotate the label to -2 degrees in 500ms
    tween.to({angle: -2}, 500);

    // Then rotate the label to +2 degrees in 500ms
    tween.to({angle: 2}, 500);

    // Loop indefinitely the tween
    tween.loop();

    // Start the tween
    tween.start();
```

Again, this can be reduced to one line of code, that we should add in the `create` function of the menu.js file:

```
    game.add.tween(startLabel).to({angle: -2}, 500).to({angle: 2}, 500).loop()
        .start();
```

Notice that when the rotation occurs, it takes place where the anchor point was defined (at the center of the label).

Scale the Coin

We saw how to tween the position and angle of an object, but we can also tween its scale. Let's try that by adding this in the `takeCoin` function, to scale the coin up when it appears:

```
    // Scale the coin to 0 to make it invisible
    this.coin.scale.setTo(0, 0);

    // Grow the coin back to its original scale in 300ms
    game.add.tween(this.coin.scale).to({x: 1, y: 1}, 300).start();
```

As you can see, we can tween multiple parameters at the same time. Here we do the 'x' and 'y' of the scale.

Scale the Player

Here's a last example: each time we take a coin we want to see the player grow slightly, then get back to its initial size. To do so, we add this in the `takeCoin` function:

```
game.add.tween(this.player.scale).to({x: 1.3, y: 1.3}, 50).to({x: 1, y: 1}, 150)
    .start();
```

With the previous explanations, you should be able to understand exactly what this line of code does.

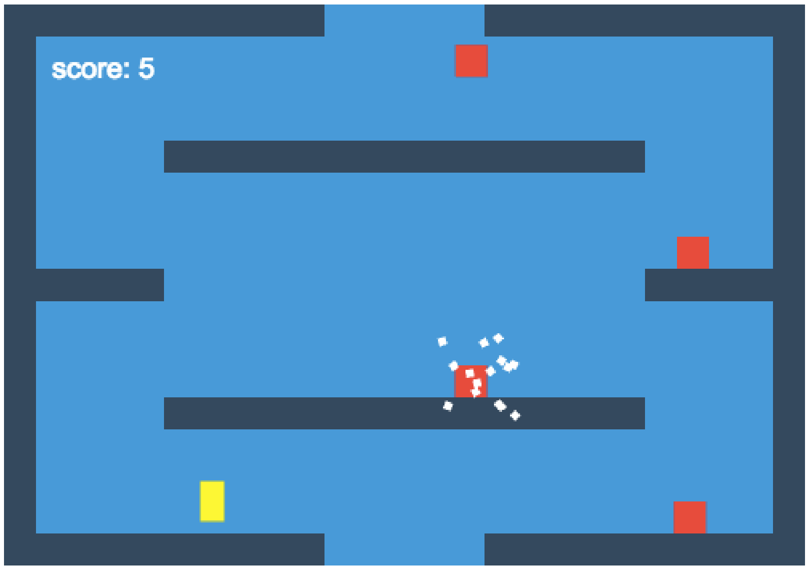
More About Tweens

As you see, tweens are really powerful and flexible. Here are just a few more examples of things you can do:

```
// Add a 100ms delay before the tween starts
tween.delay(100);

// Repeat the tween 5 times
tween.repeat(5);

// Stop the tween
tween.stop();

// Return true if the tween is currently playing
tween.isRunning;

// Will call 'callback' once the tween is finished
tween.onComplete.add(callback, this);

// And there are lots of other easing functions you can try, like:
tween.easing(Phaser.Easing.Sinusoidal.In);
tween.easing(Phaser.Easing.Exponential.Out);
```

5.4 - Add Particles

If we want to add explosions, rain, or dust to our game, we will probably use particle effects. We are going to use them to make our player explode when he hits an enemy.



Load the Particle

We need a new image for our particles: a small white square. We load it in the `preload` function of the load.js file:

```
game.load.image('pixel', 'assets/pixel.png');
```

Create the Emitter

We can't create the particles directly, we need to use an emitter to take care of that. And we can have one with `game.add.emitter`.

```
game.add.emitter(x, y, maxParticles)
```

- x: the x position of the emitter
- y: the y position of the emitter
- maxParticles: the total number of particles in the emitter

We should create and set up the emitter in the `create` function of the play.js file:

```
// Create the emitter with 15 particles. We don't need to set the x and y
// Since we don't know where to do the explosion yet
this.emitter = game.add.emitter(0, 0, 15);

// Set the 'pixel' image for the particles
this.emitter.makeParticles('pixel');

// Set the y speed of the particles between -150 and 150
// The speed will be randomly picked between -150 and 150 for each particle
this.emitter.setYSpeed(-150, 150);

// Do the same for the x speed
this.emitter.setXSpeed(-150, 150);

// Use no gravity for the particles
this.emitter.gravity = 0;
```

Setting the x and y speed like this means that the particles will go in every possible direction. For example, if we did `this.emitter.setXSpeed(0, 150)`, we wouldn't see any particles going to the left.

Start the Emitter

Now that we have our emitter, we want to start it with the `start` function.

```
start(explode, lifespan, frequency, quantity)
```

- explode: whether the particles should all burst out at once (true) or at a given frequency (false)
- lifespan: how long each particle lives once emitted in ms
- frequency: if explode is set to false, define the delay between each particles in ms
- quantity: how many particles to launch

So we add this in the `playerDie` function:

```
// Set the position of the emitter on the player
this.emitter.x = this.player.x;
this.emitter.y = this.player.y;

// Start the emitter, by exploding 15 particles that will live for 600ms
this.emitter.start(true, 600, null, 15);
```

Add a Delay

If you test the game right now, you will see no particles. That's because in the `playerDie` function we start the emitter at the same time that we go to the menu state. To fix that we need to add a delay before going to the menu.

First we create a new function that starts the menu state in the play.js file:

```
startMenu: function() {
    game.state.start('menu');
},
```

And then we edit the `playerDie` function like this:

```
playerDie: function() {
    // Kill the player to make it disappear from the screen
    this.player.kill();

    // Start the sound and the particles
    this.deadSound.play();
    this.emitter.x = this.player.x;
    this.emitter.y = this.player.y;
    this.emitter.start(true, 600, null, 15);

    // Call the 'startMenu' function in 1000ms
    game.time.events.add(1000, this.startMenu, this);
},
```

The `game.time.events.add` function works like `game.time.events.loop` that we used to create the enemies, except that it will call the function only once.

Fix the Sound

Now you see the particles, but if you make the player fall into the hole you will hear a really annoying noise. It's the 'deadSound' playing continuously. Why? When you fall, the `update` function will call `playerDie` 60 times per second. And since we now have a 1 second delay you will hear the 'deadSound' 60 times in a row.

We can edit the `playerDie` function like this to fix this issue:

```
playerDie: function() {
    // If the player is already dead, do nothing
    if (!this.player.alive) {
        return;
    }

    // Kill the player
    this.player.kill();

    // The part that will be executed only once
    this.deadSound.play();
    this.emitter.x = this.player.x;
    this.emitter.y = this.player.y;
    this.emitter.start(true, 600, null, 15);
    game.time.events.add(1000, this.startMenu, this);
    },
```

If the `playerDie` function is called multiple times in a row, only the first time will actually execute our code.

More About Particles

We are done for our little explosion, however you should know that we can do a lot more things with emitters and particles. For example:

```
    // Emit different particles
    emitter.makeParticles(['image1', 'image2', 'image3']);

    // Scale the particles
    emitter.minParticleScale = 0.2;
    emitter.maxParticleScale = 0.7;

    // Rotate the particles
    emitter.minRotation = 10;
    emitter.maxRotation = 100;

    // Change the size of the emitter
    emitter.width = 69;
    emitter.height = 42;
```

5.5 - Better Difficulty

Our game is quite hard, and it's difficult to score more than 30 points. Let's see how we can make the game easier at first, then harder over time by adding more enemies.

Static Frequency

Using a timer loop to create our enemies works well, however we don't have much control on what is happening exactly. So the first thing we should do is to build our own timer that will create new enemies every 2.2 seconds just like before.

So we replace the `game.time.events.loop(2200, this.addEnemy, this)` line from the play.js file by this variable:

```
// Contains the time of the next enemy creation
this.nextEnemy = 0;
```

And we add this in the `update` function of the play.js file:

```
// If the 'nextEnemy' time has passed
if (this.nextEnemy < game.time.now) {
    // We add a new enemy
    this.addEnemy();

    // And we update 'nextEnemy' to have a new enemy in 2.2 seconds
    this.nextEnemy = game.time.now + 2200;
}
```

The `game.time.now` is a Phaser variable that gives the time since the game started in ms.

If you test our game, you will see absolutely no change. But we now have complete control over the frequency of the enemies.

Dynamic Frequency

If we want to create more enemies over time, we first have to answer these 3 questions:

1. Start difficulty: how often should we create new enemies at the beginning?
2. End difficulty: how fast can we create enemies with the game still begin playable?
3. Progression: when do we reach the maximum difficulty?

Here's what we could use for our game:

1. Start difficulty: one new enemy every 4 seconds
2. End difficulty: one enemy per second
3. Progression: we reach the maximum difficulty when the player scores 100 points

With all this information, we can edit out timer in the `update` function like this:

```
if (this.nextEnemy < game.time.now) {
    // Define our variables
    var start = 4000, end = 1000, score = 100;

    // Formula to decrease the delay between enemies over time
    // At first it's 4000ms, then slowly goes to 1000ms
    var delay = Math.max(start - (start-end)*game.global.score/score, end);

    // Create a new enemy, and update the 'nextEnemy' time
    this.addEnemy();
    this.nextEnemy = game.time.now + delay;
}
```

We can run some numbers to make sure that the formula is working:

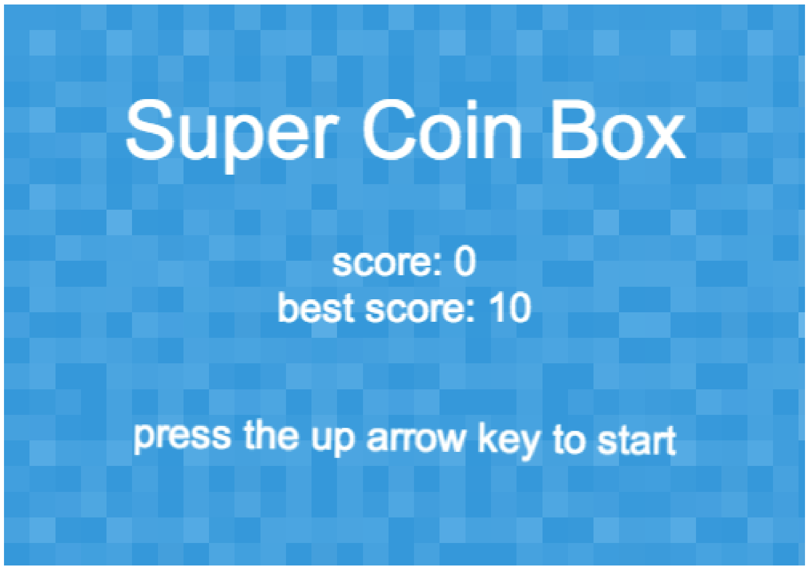
- If we have 0 points
  - `delay = max(4000 - 3000*0/100, 1000) = max(4000 - 0, 1000) = 4000`
  - That's one new enemy every 4 seconds
- If we have 50 points
  - `delay = max(4000 - 3000*50/100, 1000) = max(4000 - 1500, 1000) = 2500`
  - That's one new enemy every 2.5 seconds
- If we have 100 points
  - `delay = max(4000 - 3000*100/100, 1000) = max(4000 - 3000, 1000) = 1000`
  - That's one new enemy per second
- If we have 200 points
  - `delay = max(4000 - 3000*200/100, 1000) = max(4000 - 6000, 1000) = 1000`
  - That's still one new enemy per second

Now the game will be easy at first, and get harder as we take coins.

5.6 - Add a Best Score

One great feature of HTML5 is called "local storage", it lets us store information on people's computer. This is really useful to store a high score, the player's progress, or some settings.

We are going to use local storage to store the player's best score, and it's going to be really simple.



Store the Best Score

We only need to know 2 functions to use local storage:

- `localStorage.getItem('name')`, that returns the value stored for 'name'
- `localStorage.setItem('name', value)`, that stores 'value' in 'name'

We can use a combination of these functions to store a new best score, in the `create` function of the menu.js file:

```
// If 'bestScore' is not defined
// It means that this is the first time the game is played
if (!localStorage.getItem('bestScore')) {
    // Then set the best score to 0
    localStorage.setItem('bestScore', 0);
}

// If the score is higher than the best score
if (game.global.score > localStorage.getItem('bestScore')) {
    // Then update the best score
    localStorage.setItem('bestScore', game.global.score);
}
```

Display the Best Score

Displaying the score is even easier, we just have to use the `localStorage.getItem` function.

So far we used this to display the score in the menu.js file:

```
var scoreLabel = game.add.text(game.world.centerX, game.world.centerY,
    'score: ' + game.global.score,
    { font: '25px Arial', fill: '#ffffff' });
scoreLabel.anchor.setTo(0.5, 0.5);
```

All we have to do is to edit it like this:

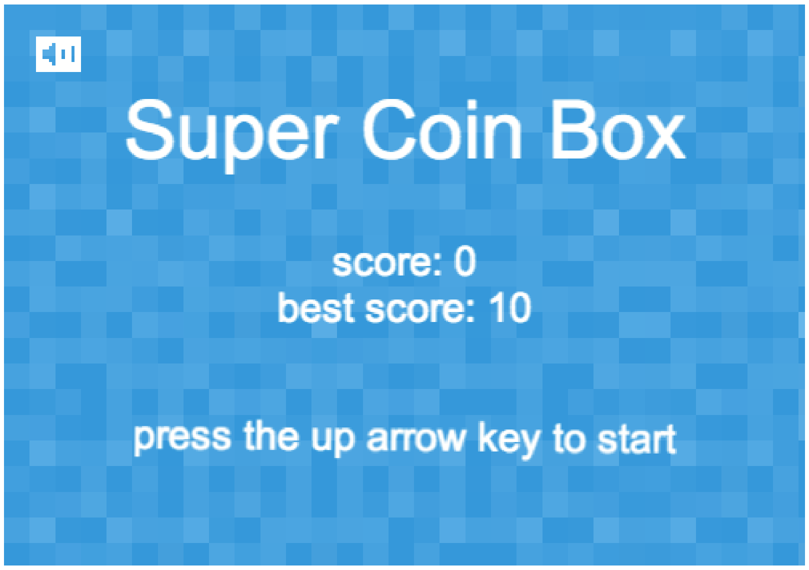
```
var text = 'score: ' + game.global.score + '\nbest score: ' +
    localStorage.getItem('bestScore');
var scoreLabel = game.add.text(game.world.centerX, game.world.centerY, text,
    { font: '25px Arial', fill: '#ffffff', align: 'center' });
scoreLabel.anchor.setTo(0.5, 0.5);
```

The '\n' will add a line break between the score and the best score, for better readability. And since the label is now on 2 lines, we added `align: 'center'` to center everything.

Just make sure to put this code after storing the best score, so that `localStorage.getItem('bestScore')` retrieves the newest best score.

5.7 - Add a Mute Button

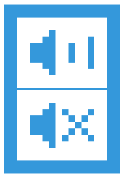
A lot of people will tell you that having a mute button in a game is a must have, and I agree. That's why we are going to add one in the menu of our game.



Load the Button

We need a button that we can press to mute the game. Since the button will have 2 different images (mute and unmute), we are going to load it as a spritesheet in the load.js file:

```
game.load.spritesheet('mute', 'assets/muteButton.png', 28, 22);
```



You can see that the sprite has 2 frames:

- Frame 0 (top), where the speaker shows some sound. It will be displayed when the game emits sound.

- And frame 1 (bottom), where the speaker shows no sound. It will be displayed when the game is muted.

Add the Button

Now we want to display the button in the top left corner of the menu with `game.add.button`.

```
game.add.button(x, y, name, callback, callbackContext)
```

- x: position x of the button
- y: position y of the button
- name: the name of the image to display
- callback: the function called when the button is clicked
- callbackContext: the context in which the 'callback' will be called, usually 'this'

To create the button, we add this in the `create` function of the menu.js file:

```
// Add the mute button that calls the 'toggleSound' function when pressed
this.muteButton = game.add.button(20, 20, 'mute', this.toggleSound, this);

// If the mouse is over the button, it becomes a hand cursor
this.muteButton.input.useHandCursor = true;
```

Next, we need to create the `toggleSound` function in the menu.js file:

```
// Function called when the 'muteButton' is pressed
toggleSound: function() {
    // Switch the Phaser sound variable from true to false, or false to true
    // When 'game.sound.mute = true', Phaser will mute the game
    game.sound.mute = ! game.sound.mute;

    // Change the frame of the button
    this.muteButton.frame = game.sound.mute ? 1 : 0;
},
```

For your information, the `this.muteButton.frame = game.sound.mute ? 1 : 0` line is exactly equal to this code, but shorter:

```
if (game.sound.mute) {
    this.muteButton.frame = 1;
}
else {
    this.muteButton.frame = 0;
}
```

Small Fix

However, we forgot to take into account one possible case. Imagine that you mute the game and die, what will you see in the upper left corner of the menu? Since the default frame of the button is `0`, we will see the speaker with sound, despite the fact that the game is muted.

To change that, we need to add this below the button creation:

```
// If the game is already muted
if (game.sound.mute) {
    // Change the frame to display the speaker with no sound
    this.muteButton.frame = 1;
}
```

5.8 - Better Keyboard Inputs

To move the player around we simply use the arrow keys, but it turns out that we can do better than that. Let's see what can be improved.

Avoid Browser Movements

Using the arrow keys or the spacebar in any browser game can be risky. For example, we might see the page scrolling down when we press the down arrow key, which is not great.

To make sure this never happens, we can add this in the `create` function of the play.js file:

```
game.input.keyboard.addKeyCapture([Phaser.Keyboard.UP,
    Phaser.Keyboard.DOWN, Phaser.Keyboard.LEFT, Phaser.Keyboard.RIGHT]);
```

This way the 4 arrow keys will be directly captured by Phaser, and not sent to the browser.

Use WASD Keys

Some people prefer to play a game with the WASD keys instead of the arrows. Let's see how we can make both work at the same time.

First we add this to the `create` function of the play.js file:

```
this.wasd = {
    up: game.input.keyboard.addKey(Phaser.Keyboard.W),
    left: game.input.keyboard.addKey(Phaser.Keyboard.A),
    right: game.input.keyboard.addKey(Phaser.Keyboard.D)
};
```

This creates a new variable that contains 3 Phaser keys: W, A, and D. The `wads` variable will work exactly like the `cursor` that we used so far.

Now we just have to edit the 3 `if` conditions of the `movePlayer` function:

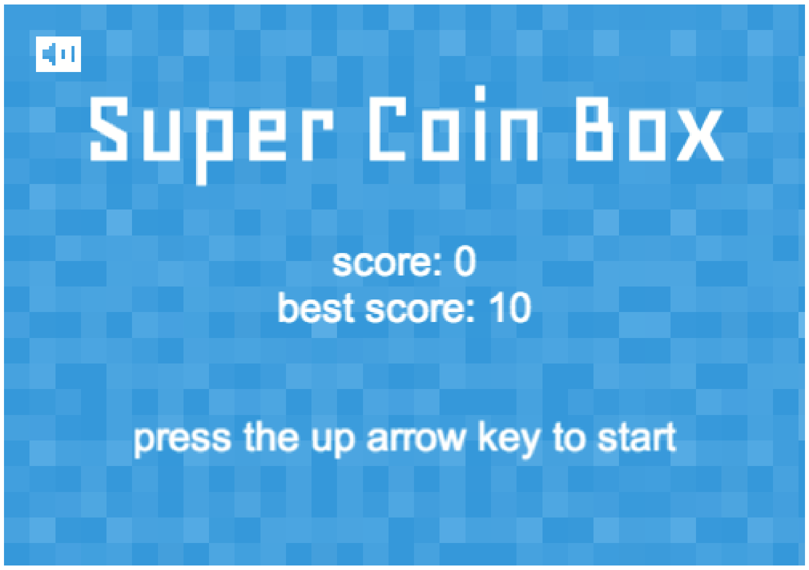
```
movePlayer: function() {
    // If the left arrow or the A key is pressed
    if (this.cursor.left.isDown || this.wasd.left.isDown) {
        this.player.body.velocity.x = -200;
        this.player.animations.play('left');
    }

    // If the right arrow or the D key is pressed
    else if (this.cursor.right.isDown || this.wasd.right.isDown) {
        this.player.body.velocity.x = 200;
        this.player.animations.play('right');
    }

    // If nothing is pressed
    else {
        this.player.body.velocity.x = 0;
        this.player.animations.stop();
        this.player.frame = 0;
    }

    // If the up arrow or the W key is pressed
    if ((this.cursor.up.isDown || this.wasd.up.isDown)
        && this.player.body.touching.down) {
        this.jumpSound.play();
        this.player.body.velocity.y = -320;
    }
},
```

And we can control the player with both the arrow and WASD keys.

5.9 - Use Custom Fonts

So far we always used the Arial font for the texts in our game. We can change that by simply using any other default font (Verdana, Georgia, etc.), or by using some custom fonts available on the web.

Google fonts is a great way to find new fonts.



Load the Font

Let's say we want to use the 'Geo' font. Google will give us a code to load the font, that we should add in the header of the index.html file:

```
<style type="text/css">
    @import url(http://fonts.googleapis.com/css?family=Geo);
</style>
```

But that's not enough. To make sure that the font will actually be loaded before the game starts, we need to add a text on the page that will use the new font. So we create a new CSS class called 'hiddenText', like this:

```
<style type="text/css">
    @import url(http://fonts.googleapis.com/css?family=Geo);

    .hiddenText {
        font-family: Geo;
        visibility: hidden;
    }
</style>
```

And then we create a simple 'p' tag with the 'hiddenText' style, containing at least one character:

```
<p class="hiddenText"> . </p>
```

This way we are sure that the font will be loaded, without actually seeing any change on the page since the text is hidden.

Use the Font

Now that the 'Geo' font is correctly loaded, we can use it anywhere in the game by just changing the style of a text. For example, we can do this for the 'nameLabel' of the menu:

```
// Replaced the '50px Arial' by '70px Geo'
var nameLabel = game.add.text(game.world.centerX, -50, 'Super Coin Box',
    { font: '70px Geo', fill: '#ffffff' });
```

5.10 - More Features

This part is going to be different that the previous ones, since we are not going to improve the game. Instead, we will quickly cover 4 important Phaser features that you might need for your own games.

Body Size

Collisions occur whenever 2 sprites barely touch each other. That's great, but sometimes you might want to reduce or increase the contact area of objects, and it's possible to do so with `sprite.body.setSize`.

`body.setSize(width, height, offsetX, offsetY)`

- width: new width of the body
- height: new height of the body
- offsetX: x offset of the body from the sprite anchor position
- offsetY: y offset of the body from the sprite anchor position

World Bounds

Here's another useful thing about collisions: `sprite.body.collideWorldBounds = true`. This line will make the sprite collide with the borders of the game and prevent it from leaving the screen.

World Size

It's possible to have a world bigger than the game size with `game.world.setBounds`.

`game.world.setBounds(x, y, width, height)`

- x: positions x of the top left corner of the world
- y: positions y of the top left corner of the world
- width: new width of the world
- height: new height of the world

Then, you can simply change the area displayed on the screen by either:

- Changing `game.camera.x` and `game.camera.y` values
- Using `game.camera.follow(sprite)` to make the camera automatically follow a sprite

Foreach

The last thing you need to know is the `forEachAlive` function, that allows you to iterate over every item of a group.

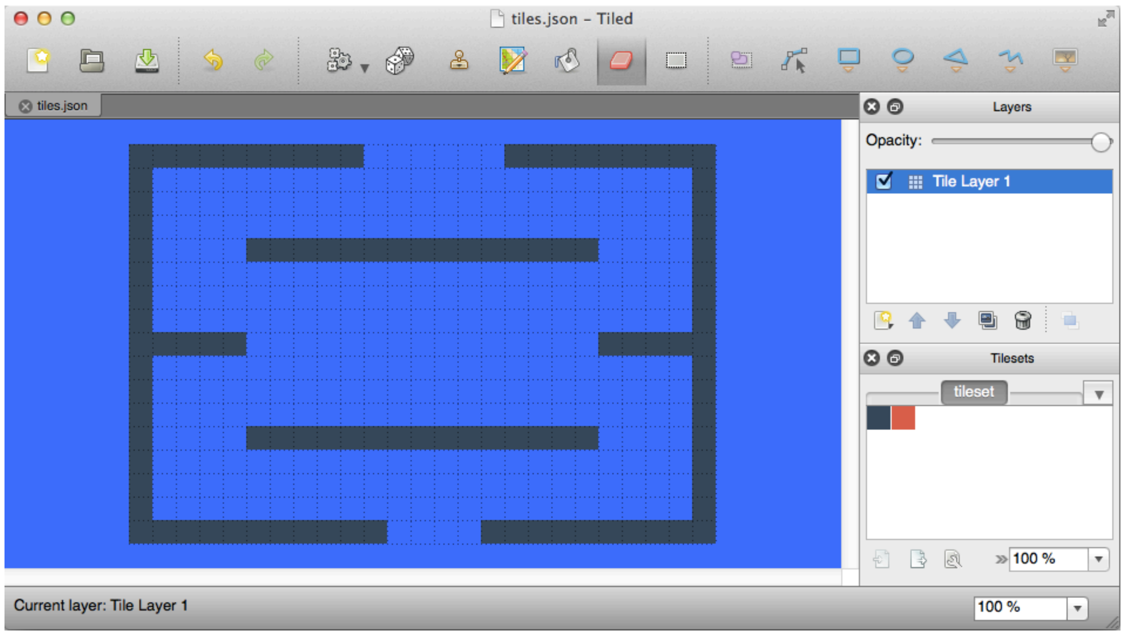
For example, let's say that we want to kill every enemy of our game as soon as they go below the player. We could do so like this:

```
// Add this line in the 'update' function of the play.js file
// It will call 'checkPosition' for each enemy alive
this.enemies.forEachAlive(this.checkPosition, this);

// And then we add the new function:
checkPosition: function(enemy) {
    if (enemy.y > this.player.y) {
        enemy.kill();
    }
},
```

6 - Use Tilemaps

In the beginning of the book we created the world manually by adding walls one by one. It worked, but we can definitely do better with tilemaps.

We are going to see how to use the software Tiled to draw our world on the screen, and then see how to display it in the game.



6.1 - Create the assets

Here are some basic definitions, to make sure that everyone is on the same page:

- Tile: a small image that represents a tiny part of a level
- Tileset: a sprite that contains all the different tiles
- Tilemap: the level, stored as a 2 dimensional array of tiles

For our game we will need 2 assets to create our new level: a tileset and a tilemap. Both of these are in the "assets" folder of the game, but we will see in this part how to create them from scratch.

The Tileset

Our tileset will be really simple since we only need one tile: a 20 by 20 pixels dark blue sprite for the walls. But to better show you how this works, we will add a second tile: a red square.

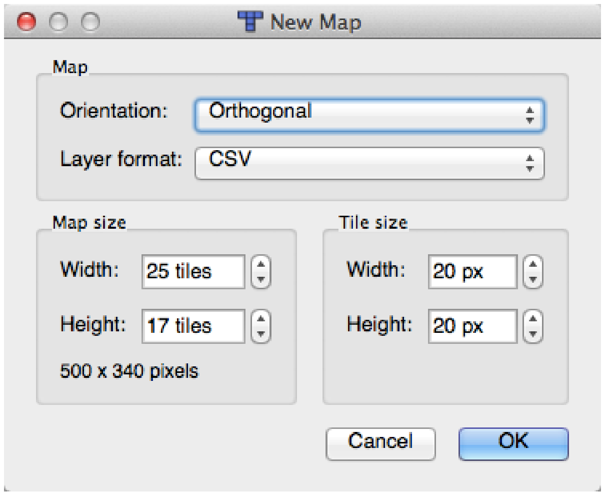
It looks like this:

Tiled

There are a lot of software available to create tilemaps, but the most popular one is probably Tiled. It's free, open source, and cross platform. You can **download it here**.

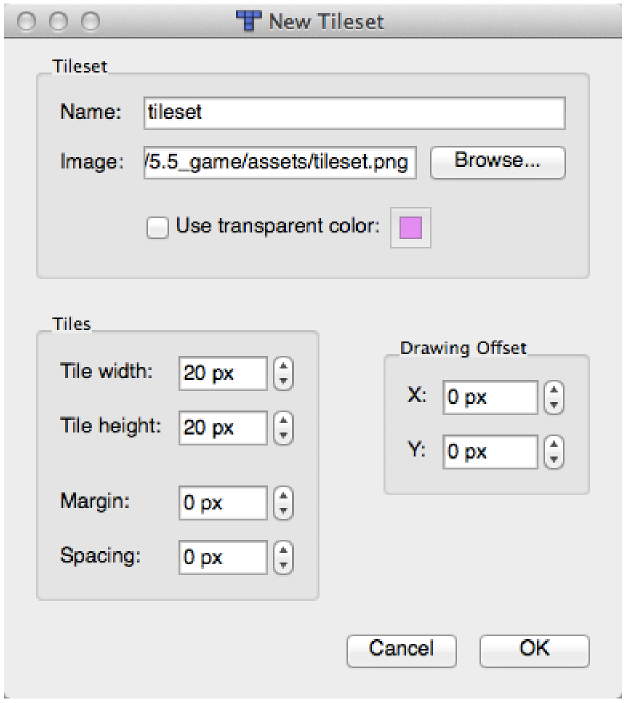
All the screenshots below were done on a Mac, and it should look similar on Windows and Linux.

Open the Tiled app, do "File > New", and fill the form like this to create an empty tilemap:



Orientation: Orthogonal
Layer format: CSV
Width: 25 tiles
Height: 17 tiles
500 x 340 pixels
Width: 20 px
Height: 20 px

Then, click on “Map > New Tileset”, and do this:

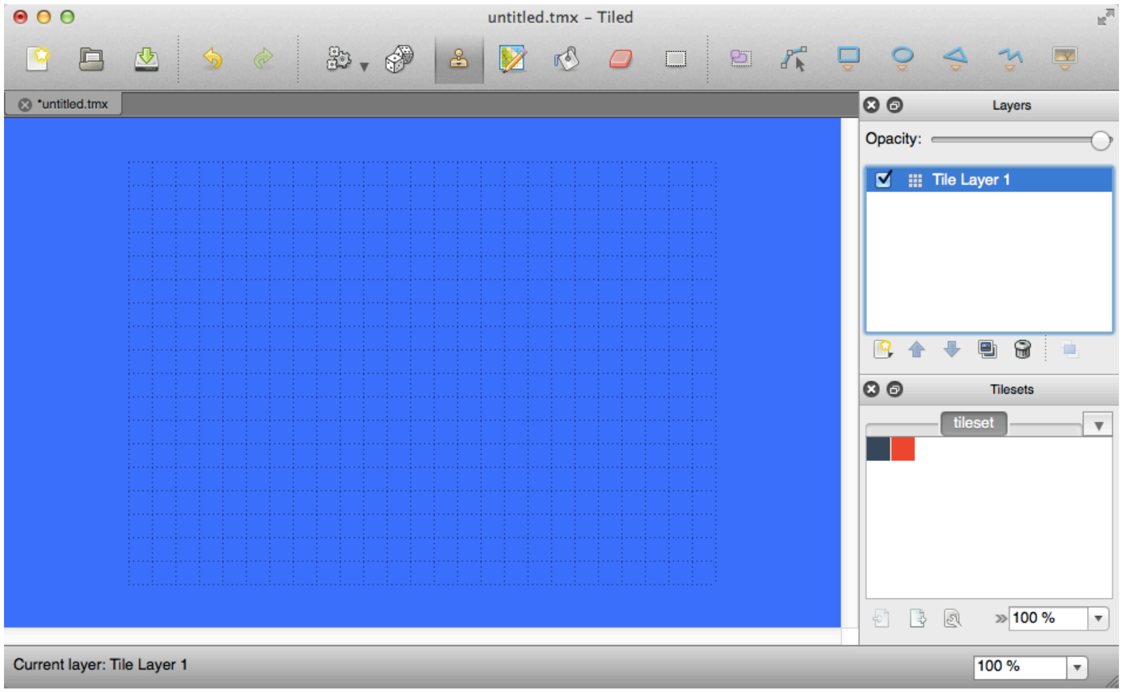


Name: tileset
Image: /5.5_game/assets/tileset.png
Use transparent color:
Tile width: 20 px
Tile height: 20 px
Margin: 0 px
Spacing: 0 px
X: 0 px
Y: 0 px

Next, do “Map > Map Properties” to set a blue background color to the map.

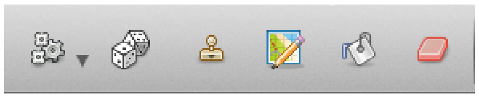
Finally, make sure that all these options are checked: “View > Show Grid”, “View > Tilesets”, and “View > Layers”.

Once done, you should see this on your screen:



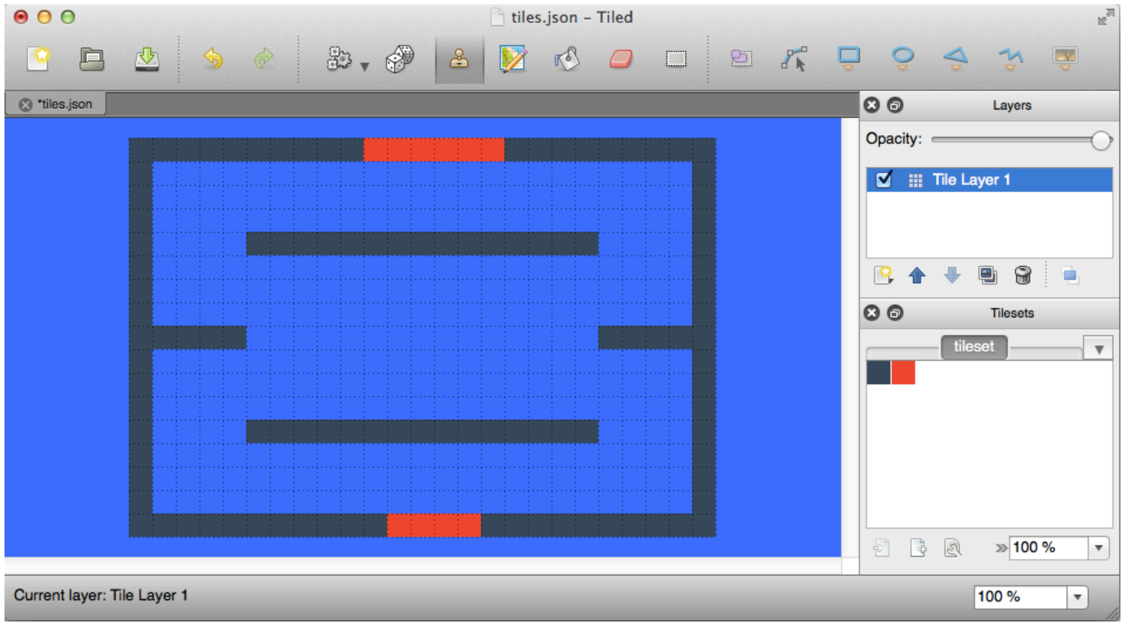
The Tilemap

Our empty tilemap is now properly set up, the next step is to actually create the level. For this we will need 2 tools: the stamp to draw tiles on the map, and the eraser to remove tiles. You can find both of them in the top menu of Tiled.



And now simply draw the map you'd like, using the tileset in the bottom right corner as your "color picker". Make sure to leave 2 holes for the enemies, at the top and bottom.

Here's the map I created. It's the same we used so far, but I added some red to fill the holes (this is just a cosmetic change).



When finished, simply hit “File > Save”, and select the json output format. You can also use the tilemap that is in the “assets” folder of the game.

Caution With Names

You may have noticed that I didn’t change the default name of the tileset (“tileset”), nor the name of the layer (“Tile Layer 1”). Well, I strongly recommend you to do the same, otherwise you will need to manually edit the json file to make the game work.

6.2 - Display the Tilemap

Now that everything is set up, we can get back to the code to have the tilemap displayed in our game.

Load Everything

We load our 2 new assets in the load.js file like this:

```
game.load.image('tileset', 'assets/tileset.png');
game.load.tilemap('map', 'assets/map.json', null, Phaser.Tilemap.TILED_JSON);
```

We can also remove these 2 lines since we don't need the walls anymore:

```
game.load.image('wallV', 'assets/wallVertical.png');
game.load.image('wallH', 'assets/wallHorizontal.png');
```

Display the Tilemap

And now we will completely change the `createWorld` function to use our new assets:

```
createWorld: function() {
    // Create the tilemap
    this.map = game.add.tilemap('map');

    // Add the tileset to the map
    this.map.addTilesetImage('tileset');

    // Create the layer, by specifying the name of the Tiled layer
    this.layer = this.map.createLayer('Tile Layer 1');

    // Set the world size to match the size of the layer
    this.layer.resizeWorld();
```

```
        // Enable collisions for the first element of our tileset (the blue wall)
        this.map.setCollision(1);
    },
```

Collisions

Since we removed the wall group from `createWorld`, we need to change the collisions of our game. Now the player and the enemies should collide with the tilemap layer like this, in the `update` function:

```
// Changed 'this.walls' into 'this.layer'
game.physics.arcade.collide(this.player, this.layer);
game.physics.arcade.collide(this.enemies, this.layer);
```

Jump

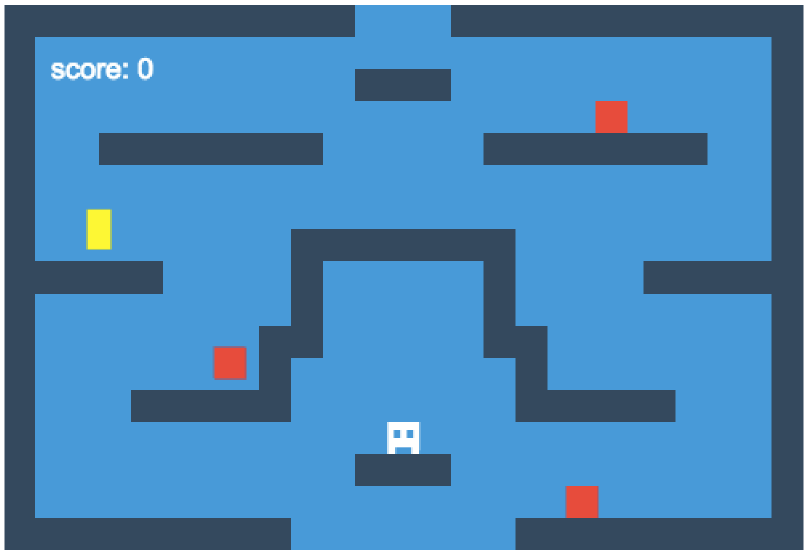
One last important thing we need to do is to update the code that makes the player jump. So far we used `body.touching.down` to check if the player could jump, but now that we're using tiles we have to use `body.onFloor()` instead:

```
if ((this.cursor.up.isDown || this.wasd.up.isDown)
    && this.player.body.onFloor()) {
    this.jumpSound.play();
    this.player.body.velocity.y = -320;
}
```

And now we can play the game. There are no visible differences, except that:

- The code is a lot cleaner, the `createWorld` function is now 5 line-long instead of 15
- And if we want to edit the level, we can do so by just opening the map.json file with Tiled

For example, here's what a new map could look like:



More About Tilemaps

Of course a lot more can be done with Tiled and tilemaps:

- Add objects in the game
- Dynamically edit the tilemap
- Add properties to tiles
- Etc.

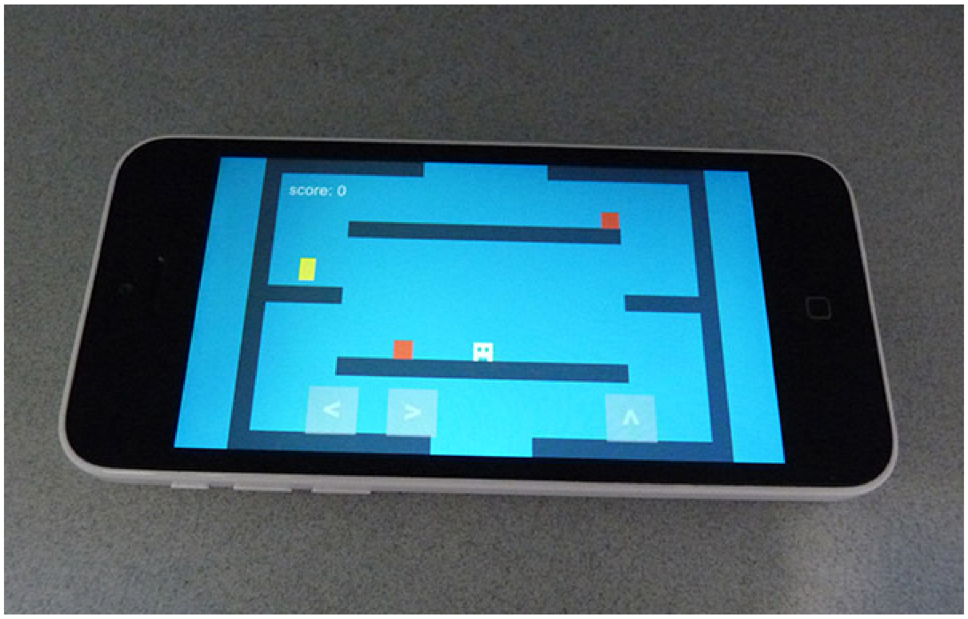
To learn more about all of this, I recommend to directly look at the Phaser and Tiled documentations.

7 - Mobile Friendly

As we mentioned in the first chapter, a great thing about HTML5 games is that they can work basically everywhere, including on phones and tablets.

To have our game playable on mobile devices, we will have to make some changes to: scale the game, handle touch inputs, and add buttons to control the player.

By the end of this chapter, our game will be mobile friendly.



7.1 - Testing

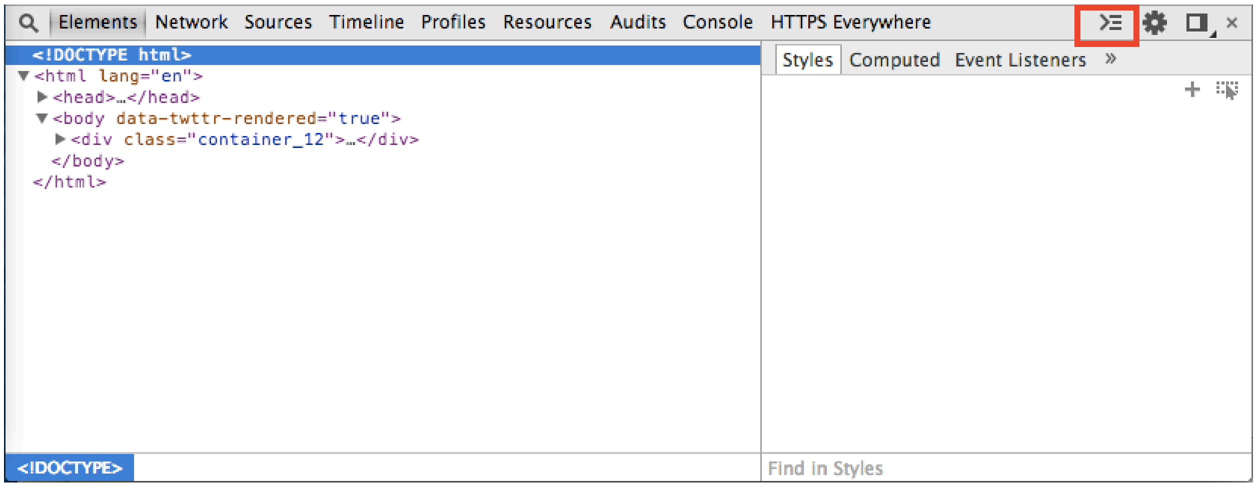
Before we start coding anything, we should know how to test our game. There are 2 different ways to do that: on a computer, and on a mobile device.

However, keep in mind that since we haven't made the game mobile friendly yet, testing it right now is not going to work very well.

On a Computer

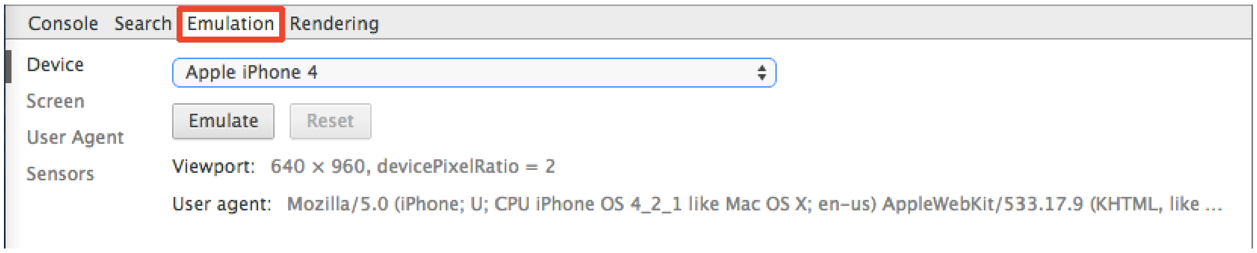
It's possible to test a mobile game from any computer with Google Chrome, though you can probably do the same on other browsers.

Open Google Chrome with the developer tools, and click on the small button in the upper right corner:



```
<!DOCTYPE html>
▼<html lang="en">
  ▶<head>…</head>
  ▼<body data-twttr-rendered="true">
    ▶<div class="container_12">…</div>
   </body>
</html>
```

You should see a new panel appearing. Click on the "emulation" tab:

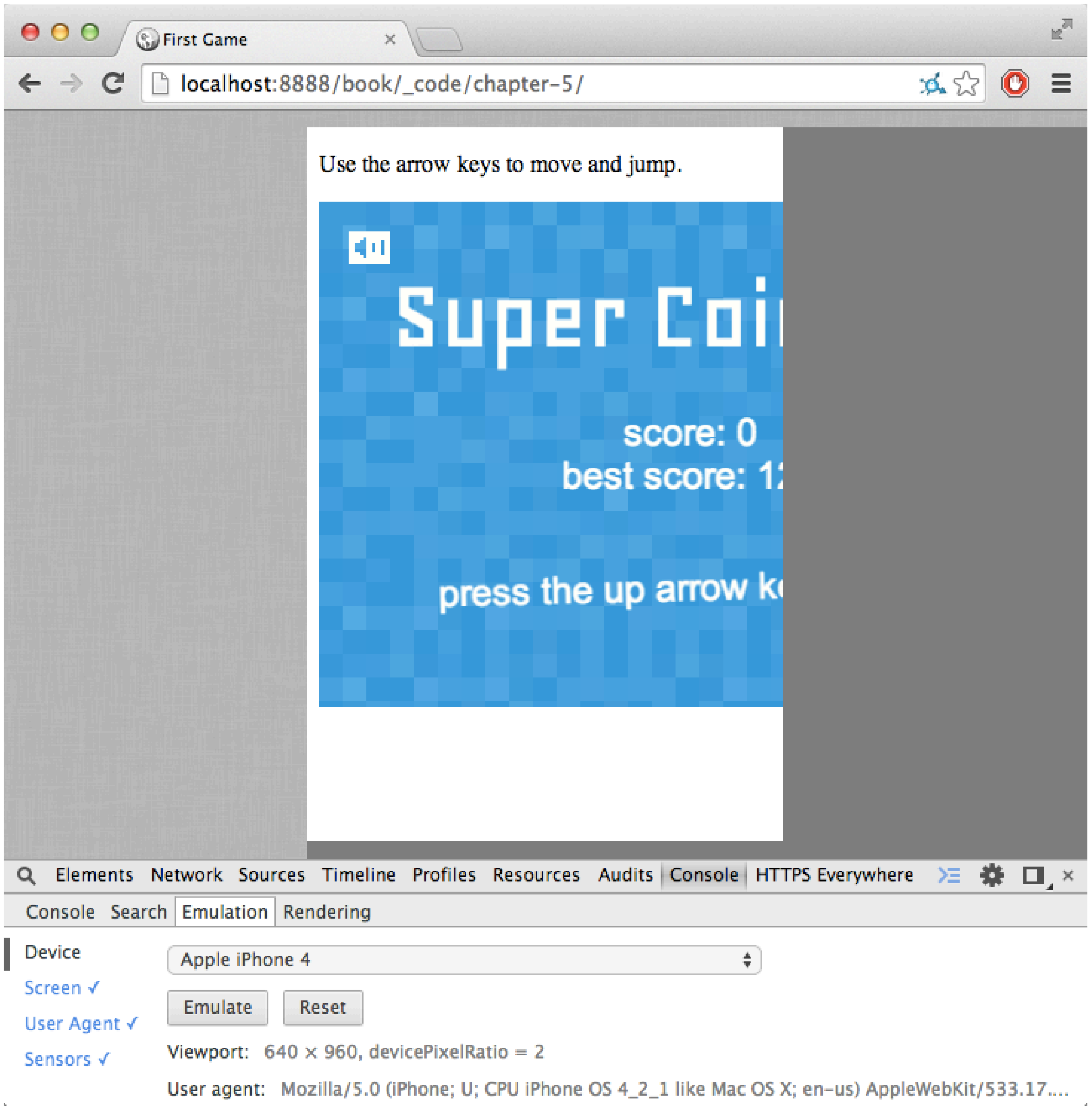


Device: Apple iPhone 4

Viewport: 640 × 960, devicePixelRatio = 2

User agent: Mozilla/5.0 (iPhone; U; CPU iPhone OS 4_2_1 like Mac OS X; en-us) AppleWebKit/533.17.9 (KHTML, like ...

There, select "iPhone 4" in the drop down menu (for example), press the "emulate" button, and then don't forget to reload the page. Now you should see the game displayed on a smaller screen.


First Game
localhost:8888/book/_code/chapter-5/
Use the arrow keys to move and jump.
Super Coi
score: 0
best score: 1
press the up arrow k
Elements Network Sources Timeline Profiles Resources Audits Console HTTPS Everywhere
Console Search Emulation Rendering
Device
Screen ✓
User Agent ✓
Sensors ✓
Emulate
Reset

Device: Apple iPhone 4

Viewport: 640 × 960, devicePixelRatio = 2

User agent: Mozilla/5.0 (iPhone; U; CPU iPhone OS 4_2_1 like Mac OS X; en-us) AppleWebKit/533.17....

To switch to portrait mode, you need to change the screen size in the "screen" tab.

As you can see, that's a really handy solution. However this technique is not 100% reliable, so use it with caution.

On a Mobile Device

The best way to test a mobile game is directly on a mobile device. To do so, we need:

- A real webserver to host the game. This way we can access the game with a URL, like www.domain.com/super-coin-box/.

- At least one mobile device. However, having multiple devices with different OS and screen sizes is better.

Then simply type the new URL of the game on a mobile device to play and test the game.

On Both

Testing on a computer is easier, but it's less reliable than on a real mobile device. So in the end, we should use both methods in this chapter.

7.2 - Scaling

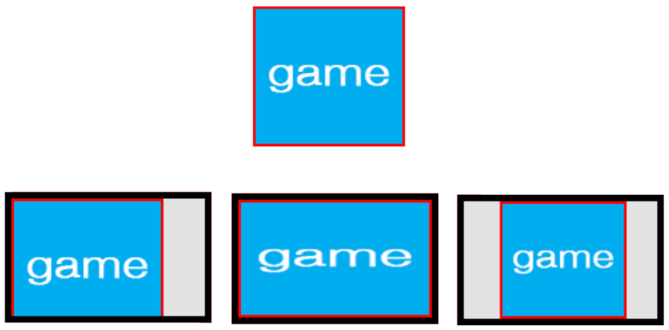
The main issue when porting games to mobile device is managing all the screen sizes. We will see see below how we can handle that.

Types of Scaling

There are 3 different types of scaling we can do with Phaser:

- No scale. That's the default behaviour, where the game doesn't change its size.
- Exact fit. The game is stretched to fill every pixel of the screen.
- Show all. It displays the whole game on the screen, without changing its proportions.

Here's a simple image to better show you the differences. Left: no scale, middle: exact fit, right: show all.



The most common way to do the scaling is with "show all", and that's what we are going to do in this part.

Edit the Boot File

We need to tell Phaser to use the "show all" scale only when the game is running on a mobile device. We can do so by adding this code in the `create` function of the boot.js file, just before the `game.state.start('load')` line:

```
// If the device is not a desktop, so it's a mobile device
if (!game.device.desktop) {
    // Set the type of scaling to 'show all'
    game.scale.scaleMode = Phaser.ScaleManager.SHOW_ALL;

    // Add a blue color to the page, to hide the white borders we might have
    document.body.style.backgroundColor = '#3498db';

    // Set the min and max width/height of the game
    game.scale.minWidth = 250;
    game.scale.minHeight = 170;
    game.scale.maxWidth = 1000;
    game.scale.maxHeight = 680;

    // Center the game on the screen
    game.scale.pageAlignHorizontally = true;
    game.scale.pageAlignVertically = true;

    // Apply the scale changes
    game.scale.setScreenSize(true);
}
```

It's always a good practice to specify a minimum and a maximum width and height for the scaling. This way, we will never have the game so small that it's unplayable, and we will avoid having the game too big with blurry assets.

Edit the Index File

Now we just need to do some changes to the index.html file:

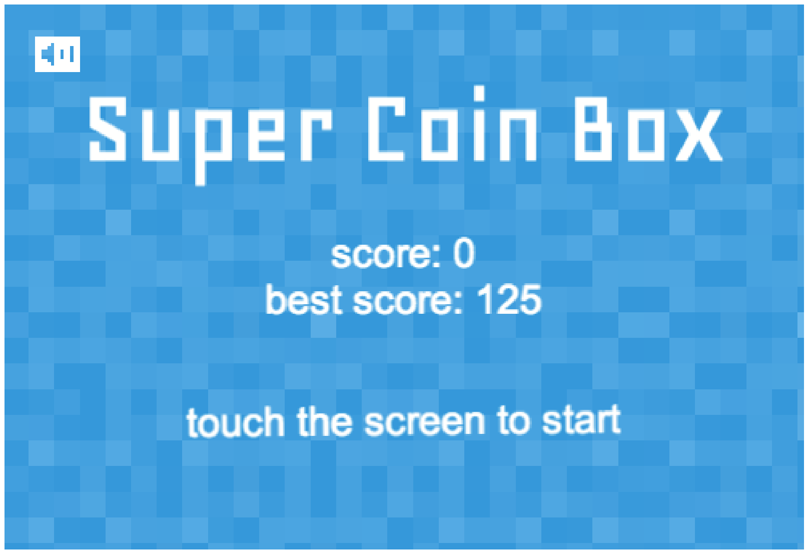
- Add CSS rules to remove every margin and padding, to make sure there are no gaps between the game and the borders of the screen
- Add some meta tags to try to force mobile phones to enter their full screen mode, but this hack won't work on every device
- Remove the text "my first Phaser game" from the page since we want the game to take all the space

Here's the new index.html that does just that:

```html
<!DOCTYPE html>
<html>

    <head>
        <meta charset="utf-8" />
        <title> First Game </title>

        <style type="text/css">
            @import url(http://fonts.googleapis.com/css?family=Geo);

            * {
                margin: 0;
                padding: 0;
            }

            .hiddenText {
                font-family: Geo;
                visibility: hidden;
            }
        </style>

        <meta name="viewport" content="initial-scale=1 maximum-scale=1
            user-scalable=0 minimal-ui" />
        <meta name="mobile-web-app-capable" content="yes">
        <meta name="apple-mobile-web-app-capable" content="yes">
        <meta name="HandheldFriendly" content="true" />

        <script type="text/javascript" src="phaser.min.js"></script>
        <script type="text/javascript" src="js/boot.js"></script>
        <script type="text/javascript" src="js/load.js"></script>
        <script type="text/javascript" src="js/menu.js"></script>
        <script type="text/javascript" src="js/play.js"></script>
        <script type="text/javascript" src="js/game.js"></script>
    </head>

    <body>
        <div id="gameDiv"> </div>
        <p class="hiddenText"> . </p>
    </body>

</html>
```

7.3 - Touch Inputs

Now the game is properly displayed on mobile devices, but we still can't go past the menu state because we can't press the up arrow key. So let's make it possible to start the game by simply touching the screen.



Change the Label

First, we should make it clear to the users that they can touch the screen to start playing. So we edit the 'startLabel' of the menu.js file like this:

```
// Store the relevant text based on the device used
if (game.device.desktop) {
    var text = 'press the up arrow key to start';
}
else {
    var text = 'touch the screen to start';
}

// Display the text variable
var startLabel = game.add.text(game.world.centerX, game.world.height-80, text,
    { font: '25px Arial', fill: '#ffffff' });
startLabel.anchor.setTo(0.5, 0.5);
```

Touch Event

Previously we did this to start the game when the up arrow key is pressed:

```
var upKey = game.input.keyboard.addKey(Phaser.Keyboard.UP);
upKey.onDown.addOnce(this.start, this);
```

For touch events, it's even easier. We just add this in the `create` function of the menu.js file:

```
game.input.onDown.addOnce(this.start, this);
```

When the input (either the mouse or a finger) is pressed, the `start` function will be called to start the play state.

More About Inputs

In Phaser there are a few ways to handle inputs. For example, we could have used something like that in the `update` function:

```
// If the input is pressed
if (game.input.activePointer.isDown) {
    // Do something
}

// If the input is not pressed
else {
    // Do something else
}
```

And we can also get the precise location of the input this way:

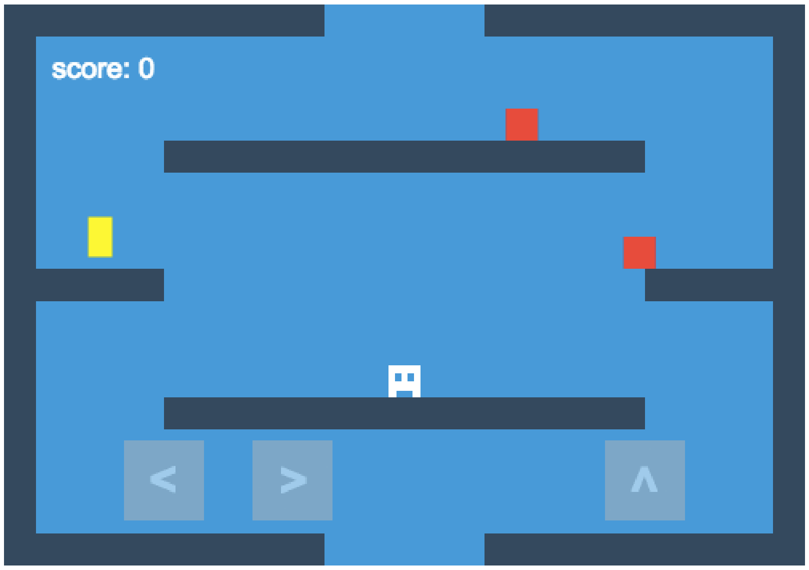
```
// Return the x position of the pointer
game.input.activePointer.x;

// Return the y position of the pointer
game.input.activePointer.y;
```

7.4 - Touch Buttons

The last step is to be able to actually play the game, by adding a new way to control the player. We could do this in a few different ways:

- Use a plugin that displays a virtual game controller in the game
- Handle touch gestures to control the player
- Add custom buttons on the screen that we can press

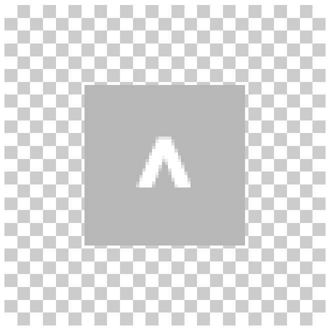
We will see how we can do the last option.



Load the Buttons

Since we have 3 inputs for our game (jump, left and right), we will load 3 images in the load.js file:

```
game.load.image('jumpButton', 'assets/jumpButton.png');
game.load.image('rightButton', 'assets/rightButton.png');
game.load.image('leftButton', 'assets/leftButton.png');
```



If you look at any of the buttons, you will see that they have big transparent borders. That's a technique to make the buttons bigger than they appear, so they are easier to press.

Display the Buttons

We could use some real Phaser buttons (like we did for the mute button), but in this case it's easier to use regular sprites. For each input, we will need to:

- Create a sprite with the button image at the correct position
- Enable inputs on the sprite, to be able to have some callbacks when the user interacts with it
- Make the sprite transparent, so we can see through it

Here's how we can do this with a new function in the play.js file:

```
addMobileInputs: function() {
    // Add the jump button
    this.jumpButton = game.add.sprite(350, 247, 'jumpButton');
    this.jumpButton.inputEnabled = true;
    this.jumpButton.alpha = 0.5;

    // Add the move left button
    this.leftButton = game.add.sprite(50, 247, 'leftButton');
    this.leftButton.inputEnabled = true;
    this.leftButton.alpha = 0.5;

    // Add the move right button
    this.rightButton = game.add.sprite(130, 247, 'rightButton');
    this.rightButton.inputEnabled = true;
    this.rightButton.alpha = 0.5;
},
```

And we should not forget to call `addMobileInputs` in the `create` function of the play.js file:

```
// If the game is running on a mobile device
if (!game.device.desktop) {
    // Display the mobile inputs
    this.addMobileInputs();
}
```

Handling Events

Since our buttons have `inputEnabled = true`, we can track precisely how the user interacts with them thanks to these 4 functions:

```
// Triggered when the pointer is over the button
button.events.onInputOver.add(callback, this);

// Triggered when the pointer is moving away from the button
button.events.onInputOut.add(callback, this);

// Triggered when the pointer touches the button
button.events.onInputDown.add(callback, this);

// Triggered when the pointer goes up over the button
button.events.onInputUp.add(callback, this);
```

By the way, these functions can be rewritten this way:

```
button.events.onInputOver.add(function(){ /* do something */ }, this);
button.events.onInputOut.add(function(){ /* do something */ }, this);
button.events.onInputDown.add(function(){ /* do something */ }, this);
button.events.onInputUp.add(function(){ /* do something */ }, this);
```

Let's see how we can use all these new functions to make our buttons useful.

Jump

First, let's take care of the jump button by adding this to the `addMobileInputs` function:

```
// Call 'jumpPlayer' when the 'jumpButton' is pressed
this.jumpButton.events.onInputDown.add(this.jumpPlayer, this);
```

Now we create the new `jumpPlayer` function:

```
jumpPlayer: function() {
    // If the player is touching the ground
    if (this.player.body.onFloor()) {
        // Jump with sound
        this.player.body.velocity.y = -320;
        this.jumpSound.play();
    }
},
```

And it should work. But before we move on, we should also edit the `movePlayer` function to use `jumpPlayer` and avoid duplicating the jump code.

```
movePlayer: function() {
   // Do not change the beginning

   // ...

    // That's the part we need to edit, to call our new function
    if (this.cursor.up.isDown || this.wasd.up.isDown) {
        this.jumpPlayer();
    }
},
```

Now each time a key or the 'jumpButton' is pressed, we will call `jumpPlayer`.

Move Right and Left - Idea

For the left and right inputs, we will have to do something different. Previously, we called the `jumpPlayer` function just once when the button was pressed. For the player movements we want to be able to keep pressing the button to keep moving, so using `button.events.onInputDown.add` won't be enough.

We will need to use the other input functions we saw earlier. Here's how it will work for the `rightButton`:

- We define a new variable `moveRight`, set to false by default
- If `onInputOver` or `onInputDown` is triggered, then we will set the variable to true
- If `onInputOut` or `onInputUp` is triggered, we will set the variable to false
- This way, if `moveRight` is true, it means that the user is pressing the right button

So by just looking at the `moveRight` variable in the `update` function, we know if we should move the player to the right or not.

Move Right and Left - Code

Here's the new `addMobileInputs` function:

```
addMobileInputs: function() {
    // Add the jump button
    this.jumpButton = game.add.sprite(350, 247, 'jumpButton');
    this.jumpButton.inputEnabled = true;
    this.jumpButton.events.onInputDown.add(this.jumpPlayer, this);
    this.jumpButton.alpha = 0.5;

    // Movement variables
    this.moveLeft = false;
    this.moveRight = false;

    // Add the move left button
    this.leftButton = game.add.sprite(50, 247, 'leftButton');
    this.leftButton.inputEnabled = true;
    this.leftButton.events.onInputOver.add(function(){this.moveLeft=true;}, this);
    this.leftButton.events.onInputOut.add(function(){this.moveLeft=false;}, this);
    this.leftButton.events.onInputDown.add(function(){this.moveLeft=true;}, this);
    this.leftButton.events.onInputUp.add(function(){this.moveLeft=false;}, this);
    this.leftButton.alpha = 0.5;

    // Add the move right button
    this.rightButton = game.add.sprite(130, 247, 'rightButton');
    this.rightButton.inputEnabled = true;
    this.rightButton.events.onInputOver.add(function(){this.moveRight=true;},
        this);
    this.rightButton.events.onInputOut.add(function(){this.moveRight=false;},
        this);
    this.rightButton.events.onInputDown.add(function(){this.moveRight=true;},
        this);
    this.rightButton.events.onInputUp.add(function(){this.moveRight=false;},
        this);
    this.rightButton.alpha = 0.5;
},
```

You can see that:

- We defined 2 new variables: `moveRight` and `moveLeft`
- For each right and left button, we added the 4 `events.onInput` functions
- Each time the user interacts with one of the buttons, we update the value of the new variables accordingly

So at this point, by simply looking at `moveRight` and `moveLeft`, we know where we should move the player. Let's do that by updating the 2 `if` conditions of the `movePlayer` function:

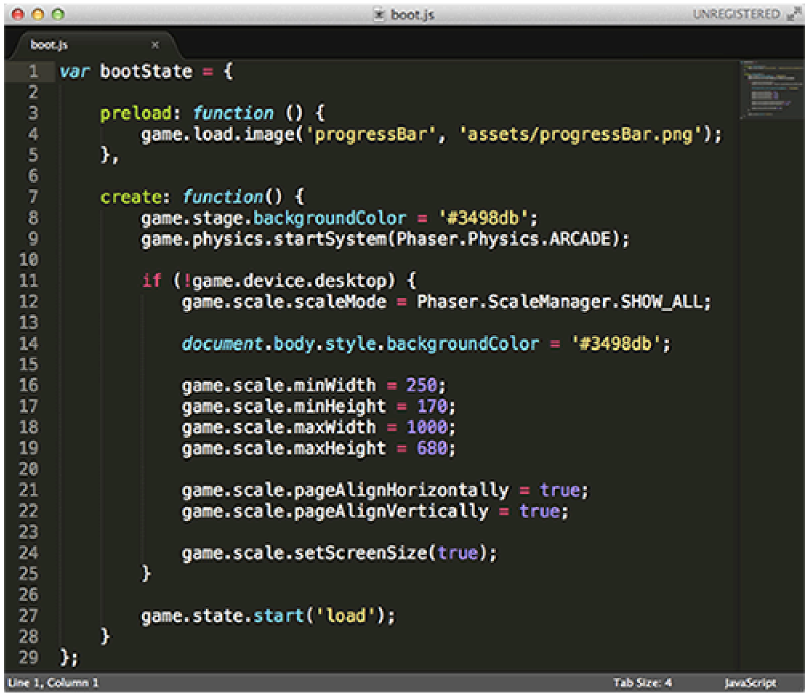
```
    movePlayer: function() {
        // Player moving left
        if (this.cursor.left.isDown || this.wasd.left.isDown || this.moveLeft) {
            this.player.body.velocity.x = -200;
            this.player.animations.play('left');
        }

        // Player moving right
        else if (this.cursor.right.isDown || this.wasd.right.isDown ||
            this.moveRight) {
            this.player.body.velocity.x = 200;
            this.player.animations.play('right');
        }

        // Do not change the rest of the function
    }
```

It means that 60 times per second we will check if a key or a button is pressed, in order to move the player.

Make sure to test the game on a mobile device, and it should work as expected.

8 - Full Source Code

Now that the game is finished, let's review the whole code we've made so far.



```
var bootState = {

    preload: function () {
        game.load.image('progressBar', 'assets/progressBar.png');
    },

    create: function() {
        game.stage.backgroundColor = '#3498db';
        game.physics.startSystem(Phaser.Physics.ARCADE);

        if (!game.device.desktop) {
            game.scale.scaleMode = Phaser.ScaleManager.SHOW_ALL;

            document.body.style.backgroundColor = '#3498db';

            game.scale.minWidth = 250;
            game.scale.minHeight = 170;
            game.scale.maxWidth = 1000;
            game.scale.maxHeight = 680;

            game.scale.pageAlignHorizontally = true;
            game.scale.pageAlignVertically = true;

            game.scale.setScreenSize(true);
        }

        game.state.start('load');
    }
};
```

8.1 - Index

Our webpage that will display the game in the `<div id="gameDiv"> </div>` element.

```html
<!DOCTYPE html>
<html>

    <head>
        <meta charset="utf-8" />
        <title> First Game </title>

        <style type="text/css">
            @import url(http://fonts.googleapis.com/css?family=Geo);

            * {
                margin: 0;
                padding: 0;
            }

            .hiddenText {
                font-family: Geo;
                visibility: hidden;
            }
        </style>

        <meta name="viewport" content="initial-scale=1 maximum-scale=1
            user-scalable=0 minimal-ui" />
        <meta name="mobile-web-app-capable" content="yes">
        <meta name="apple-mobile-web-app-capable" content="yes">
        <meta name="HandheldFriendly" content="true" />

        <script type="text/javascript" src="phaser.min.js"></script>
        <script type="text/javascript" src="js/boot.js"></script>
        <script type="text/javascript" src="js/load.js"></script>
        <script type="text/javascript" src="js/menu.js"></script>
        <script type="text/javascript" src="js/play.js"></script>
        <script type="text/javascript" src="js/game.js"></script>
    </head>

    <body>
```

```
            <div id="gameDiv"> </div>
            <p class="hiddenText"> . </p>
        </body>

    </html>
```

8.2 - Boot

The first state of our game. It will load the `progressBar` asset, and handle the scaling of the game.

```
var bootState = {

    preload: function () {
        game.load.image('progressBar', 'assets/progressBar.png');
    },

    create: function() {
        game.stage.backgroundColor = '#3498db';
        game.physics.startSystem(Phaser.Physics.ARCADE);

        if (!game.device.desktop) {
            game.scale.scaleMode = Phaser.ScaleManager.SHOW_ALL;

            document.body.style.backgroundColor = '#3498db';

            game.scale.minWidth = 250;
            game.scale.minHeight = 170;
            game.scale.maxWidth = 1000;
            game.scale.maxHeight = 680;

            game.scale.pageAlignHorizontally = true;
            game.scale.pageAlignVertically = true;

            game.scale.setScreenSize(true);
        }

        game.state.start('load');
    }
};
```

8.3 - Load

Here we load all the assets we need for our game.

```
var loadState = {

    preload: function () {
        var loadingLabel = game.add.text(game.world.centerX, 150, 'loading...',
            { font: '30px Arial', fill: '#ffffff' });
        loadingLabel.anchor.setTo(0.5, 0.5);

        var progressBar = game.add.sprite(game.world.centerX, 200, 'progressBar');
        progressBar.anchor.setTo(0.5, 0.5);
        game.load.setPreloadSprite(progressBar);

        game.load.spritesheet('player', 'assets/player2.png', 20, 20);
        game.load.image('enemy', 'assets/enemy.png');
        game.load.image('coin', 'assets/coin.png');
        game.load.image('pixel', 'assets/pixel.png');
        game.load.image('background', 'assets/background.png');
        game.load.spritesheet('mute', 'assets/muteButton.png', 28, 22);
        game.load.image('jumpButton', 'assets/jumpButton.png');
        game.load.image('rightButton', 'assets/rightButton.png');
        game.load.image('leftButton', 'assets/leftButton.png');

        game.load.audio('jump', ['assets/jump.ogg', 'assets/jump.mp3']);
        game.load.audio('coin', ['assets/coin.ogg', 'assets/coin.mp3']);
        game.load.audio('dead', ['assets/dead.ogg', 'assets/dead.mp3']);

        this.load.image('tileset', 'assets/tileset.png');
        this.load.tilemap('map', 'assets/map.json', null,
            Phaser.Tilemap.TILED_JSON);
    },

    create: function() {
        game.state.start('menu');
    }
};
```

8.4 - Menu

The menu of our game.

```
var menuState = {

    create: function() {
        game.add.image(0, 0, 'background');

        var nameLabel = game.add.text(game.world.centerX, -50, 'Super Coin Box',
            { font: '70px Geo', fill: '#ffffff' });
        nameLabel.anchor.setTo(0.5, 0.5);
        game.add.tween(nameLabel).to({y: 80}, 1000)
            .easing(Phaser.Easing.Bounce.Out).start();

        if (!localStorage.getItem('bestScore')) {
            localStorage.setItem('bestScore', 0);
        }

        if (game.global.score > localStorage.getItem('bestScore')) {
            localStorage.setItem('bestScore', game.global.score);
        }

        var text = 'score: ' + game.global.score + '\nbest score: ' +
            localStorage.getItem('bestScore');
        var scoreLabel = game.add.text(game.world.centerX, game.world.centerY,
            text, { font: '25px Arial', fill: '#ffffff', align: 'center' });
        scoreLabel.anchor.setTo(0.5, 0.5);

        if (this.game.device.desktop) {
            var text = 'press the up arrow key to start';
        }
        else {
            var text = 'touch the screen to start';
        }
        var startLabel = game.add.text(game.world.centerX, game.world.height-80,
            text, { font: '25px Arial', fill: '#ffffff' });
        startLabel.anchor.setTo(0.5, 0.5);

        game.add.tween(startLabel).to({angle: -2}, 500).to({angle: 2}, 500).loop()
```

```
            .start();

        this.muteButton = game.add.button(20, 20, 'mute', this.toggleSound, this);
        this.muteButton.input.useHandCursor = true;
        if (game.sound.mute) {
            this.muteButton.frame = 1;
        }

        var upKey = game.input.keyboard.addKey(Phaser.Keyboard.UP);
        upKey.onDown.addOnce(this.start, this);

        game.input.onDown.addOnce(this.start, this);
    },

    toggleSound: function() {
        game.sound.mute = ! game.sound.mute;
        this.muteButton.frame = game.sound.mute ? 1 : 0;
    },

    start: function() {
        game.state.start('play');
    }
};
```

8.5 - Play

The actual game.

```
var playState = {

    create: function() {
        this.cursor = game.input.keyboard.createCursorKeys();
        game.input.keyboard.addKeyCapture([Phaser.Keyboard.UP,
            Phaser.Keyboard.DOWN, Phaser.Keyboard.LEFT,
            Phaser.Keyboard.RIGHT]);
        this.wasd = {
            up: game.input.keyboard.addKey(Phaser.Keyboard.W),
            left: game.input.keyboard.addKey(Phaser.Keyboard.A),
            right: game.input.keyboard.addKey(Phaser.Keyboard.D)
        };

        game.global.score = 0;
        this.createWorld();
        if (!game.device.desktop) {
            this.addMobileInputs();
        }

        this.player = game.add.sprite(game.world.centerX, game.world.centerY,
            'player');
        game.physics.arcade.enable(this.player);
        this.player.anchor.setTo(0.5, 0.5);
        this.player.body.gravity.y = 500;
        this.player.animations.add('right', [1, 2], 8);
        this.player.animations.add('left', [3, 4], 8);

        this.enemies = game.add.group();
        this.enemies.enableBody = true;
        this.enemies.createMultiple(10, 'enemy');

        this.coin = game.add.sprite(60, 140, 'coin');
        game.physics.arcade.enable(this.coin);
        this.coin.anchor.setTo(0.5, 0.5);

        this.scoreLabel = game.add.text(30, 30, 'score: 0',
            { font: '18px Arial', fill: '#ffffff' });

        this.emitter = game.add.emitter(0, 0, 15);
        this.emitter.makeParticles('pixel');
        this.emitter.setYSpeed(-150, 150);
        this.emitter.setXSpeed(-150, 150);
        this.emitter.gravity = 0;

        this.jumpSound = game.add.audio('jump');
        this.coinSound = game.add.audio('coin');
        this.deadSound = game.add.audio('dead');

        this.nextEnemy = 0;
    },

    update: function() {
        game.physics.arcade.overlap(this.player, this.enemies, this.playerDie,
            null, this);
        game.physics.arcade.overlap(this.player, this.coin, this.takeCoin,
            null, this);
        game.physics.arcade.collide(this.player, this.layer);
        game.physics.arcade.collide(this.enemies, this.layer);

        if (!this.player.inWorld) {
            this.playerDie();
        }

        this.movePlayer();

        if (this.nextEnemy < game.time.now) {
            var start = 4000, end = 1000, score = 100;
            var delay = Math.max(start -
                (start-end)*game.global.score/score, end);

            this.addEnemy();
            this.nextEnemy = game.time.now + delay;
         }
    },

    movePlayer: function() {
        if (this.cursor.left.isDown || this.wasd.left.isDown
            || this.moveLeft) {
            this.player.body.velocity.x = -200;
            this.player.animations.play('left');
        }
        else if (this.cursor.right.isDown || this.wasd.right.isDown
            || this.moveRight) {
            this.player.body.velocity.x = 200;
            this.player.animations.play('right');
        }
        else {
            this.player.body.velocity.x = 0;
            this.player.animations.stop();
            this.player.frame = 0;
        }

        if (this.cursor.up.isDown || this.wasd.up.isDown) {
            this.jumpPlayer();
        }
    },

    jumpPlayer: function() {
        if (this.player.body.onFloor()) {
            this.jumpSound.play();
            this.player.body.velocity.y = -320;
        }
    },

    addEnemy: function() {
        var enemy = this.enemies.getFirstDead();
        if (!enemy) {
            return;
        }

        enemy.anchor.setTo(0.5, 1);
        enemy.reset(game.world.centerX, 0);
        enemy.body.gravity.y = 500;
        enemy.body.bounce.x = 1;
        enemy.body.velocity.x = 100 * Phaser.Math.randomSign();
        enemy.checkWorldBounds = true;
        enemy.outOfBoundsKill = true;
    },

    takeCoin: function(player, coin) {
        game.global.score += 5;
        this.scoreLabel.text = 'score: ' + game.global.score;

        this.updateCoinPosition();
        this.coinSound.play();
        game.add.tween(this.player.scale).to({x: 1.3, y: 1.3}, 50)
            .to({x: 1, y: 1}, 150).start();
        this.coin.scale.setTo(0, 0);
        game.add.tween(this.coin.scale).to({x: 1, y: 1}, 300).start();
    },

    updateCoinPosition: function() {
        var coinPosition = [
            {x: 140, y: 60}, {x: 360, y: 60},
            {x: 60, y: 140}, {x: 440, y: 140},
            {x: 130, y: 300}, {x: 370, y: 300}
        ];

        for (var i = 0; i < coinPosition.length; i++) {
            if (coinPosition[i].x === this.coin.x) {
                coinPosition.splice(i, 1);
            }
        }

        var newPosition = coinPosition[game.rnd.integerInRange(0,
            coinPosition.length-1)];
        this.coin.reset(newPosition.x, newPosition.y);
    },

    playerDie: function() {
        if (!this.player.alive) {
            return;
        }

        this.player.kill();

        this.deadSound.play();
        this.emitter.x = this.player.x;
        this.emitter.y = this.player.y;
        this.emitter.start(true, 600, null, 15);

        game.time.events.add(1000, this.startMenu, this);
    },

    startMenu: function() {
        game.state.start('menu');
    },
    createWorld: function() {
        this.map = game.add.tilemap('map');
        this.map.addTilesetImage('tileset');
        this.layer = this.map.createLayer('Tile Layer 1');
        this.layer.resizeWorld();
        this.map.setCollision(1);
    },

    addMobileInputs: function() {
        this.jumpButton = game.add.sprite(350, 247, 'jumpButton');
        this.jumpButton.inputEnabled = true;
        this.jumpButton.events.onInputDown.add(this.jumpPlayer, this);
        this.jumpButton.alpha = 0.5;

        this.moveLeft = false;
        this.moveRight = false;

        this.leftButton = game.add.sprite(50, 247, 'leftButton');
        this.leftButton.inputEnabled = true;
        this.leftButton.events.onInputOver.add(function(){this.moveLeft=true;},
            this);
        this.leftButton.events.onInputOut.add(function(){this.moveLeft=false;},
            this);
        this.leftButton.events.onInputDown.add(function(){this.moveLeft=true;},
            this);
        this.leftButton.events.onInputUp.add(function(){this.moveLeft=false;},
            this);
        this.leftButton.alpha = 0.5;

        this.rightButton = game.add.sprite(130, 247, 'rightButton');
        this.rightButton.inputEnabled = true;
        this.rightButton.events.onInputOver.add(function(){this.moveRight=true;},
            this);
        this.rightButton.events.onInputOut.add(function(){this.moveRight=false;},
            this);
        this.rightButton.events.onInputDown.add(function(){this.moveRight=true;},
            this);
        this.rightButton.events.onInputUp.add(function(){this.moveRight=false;},
            this);
        this.rightButton.alpha = 0.5;
    }
};
```

8.6 - Game

Here we initialise everything.

```
var game = new Phaser.Game(500, 340, Phaser.AUTO, 'gameDiv');

game.global = {
    score: 0
};

game.state.add('boot', bootState);
game.state.add('load', loadState);
game.state.add('menu', menuState);
game.state.add('play', playState);

game.state.start('boot');
```

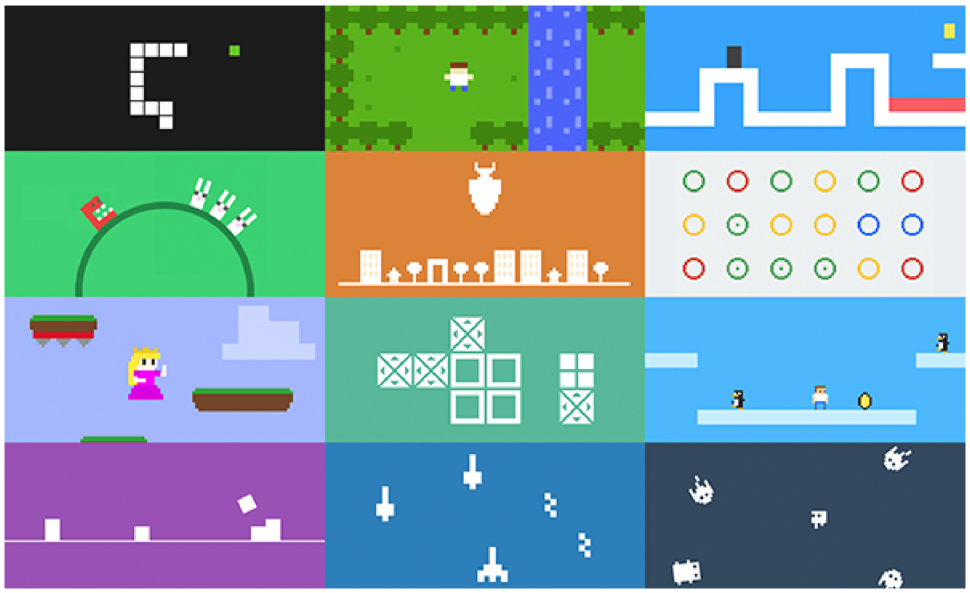
9 - Next Steps

Congratulations, you've made a full featured game with Phaser! And remember that we started all of this with an empty blue screen.

Here are the main Phaser features we covered in this book:

- States, sprites, labels, groups, recycling, Arcade physics
- Sounds, animations, tweens, particles
- Local storage, buttons, custom fonts, keyboard
- Tilemaps with Tiled
- Scaling, touch inputs, touch buttons

This last chapter will give you some ideas and tips on what to do next.



9.1 - Improve the Game

The first thing you could do is to improve the game we made together. To help you with that, I listed below a few ideas you could try to implement.

Customisation

An easy thing you can do is to customise the game. Change all the sprites and sounds, tweak all the values (gravity, velocity, tweens), and you will get a brand new game.

Add More Tweens

Tweens are a great way to make a game feel better, and you could add more of them. For example: make the coins rotate indefinitely, add a special animation when the user has a new best score, make the score grow each time you collect a coin, etc.

Multiple Lives

Instead of ending the game as soon as the player hits an enemy, we could simply lose a life and keep playing.

```
// In the 'create' function
define a life variables
add a life label on the screen

// In the 'playerDie' function
if (lives > 0) {
    // The player loses a life
    decrement the number of lives
    update the life label
    kill all the enemies
}
else {
    // It's game over
    start the menu state
}
```

Variable Jump Height

When we press the up arrow key, the player always jumps to the same height. Wouldn't it be cool to jump more or less high, depending on how long the up arrow key is pressed? Most platformer games do this.

```
// In the 'jumpPlayer' function
if (the player is touching the ground) {
    jump with an initial velocity of -320
}
else if (the jump started less than 200ms ago) {
    keep jumping with a velocity of -320
}
```

New Enemy Types

The current red enemies all have the same size and move at the same speed. That's a bit boring. To change that, you could create new types of enemies: smaller and faster ones, or bigger and slower ones.

```
// In the 'addEnemy' function
if (true 50% of the time) {
    // The enemy will be big and slow
    scale the enemy up
    decrease its velocity
}
else {
    // The enemy will be small and fast
    scale the enemy down
    increase its velocity
}
```

Add Weapons

The game is mostly about avoiding enemies. Avoiding is fine, but shooting is probably better. So give some weapons to the player, and let him kill the enemies near him. And when an enemy dies, make sure to add some great particle effects.

```
// In the 'create' function
create a bullet group
create a 'nextBullet' variable

// In the 'update' function
handle bullets collisions with enemies

// In the 'movePlayer' function
if (the space bar is pressed && nextBullet < time.now) {
    create the bullet in front of the player with an horizontal velocity
    update the 'nextBullet' variable
}
```

9.2 - Make Your Own Games

A bigger challenge would be to make your own games from scratch. If you've never done that before, you will quickly discover that it's a process that can be a little scary (and also amazing).

I wrote below a few simple tips that may help you to build your own games.

Start Small

It's easy to get into the trap of wanting to make a really complex game: "let's make a Zelda-like, with quests, dungeons, puzzles, and bosses!". But most of the time these projects are put on shelves because they are too long to make.

Instead, start making really simple mini games like Pong, Breakout or Space Invaders. And when you become comfortable making these types of games, then you can start more ambitious ones.

Graphics and Sounds

Graphics and sounds are really important, but you don't need to be a designer or a musician to make good games.

For graphics go on **opengameart** website, where they have tons of sprites available for free. For sounds you should use **Bfxr**, which lets you create nice sound effects by just pressing some buttons.

Keep Iterating

Instead of trying to make the perfect game from the start, try to divide it into smaller pieces, then build the game step by step. Want an example? Simply look at how we made our platformer in this book: we started with just some basic elements, then added menus, then added animations and sounds, and so on.

Making games step by step is the best way to go.

Game Ideas

If you don't have any idea on what games you could make, simply look at the games you like, and try to do something similar. I'm not advising you to exactly copy them, but just starting with one game in mind can help you get started.

9.3 - Conclusion

The book is now over, I really hope that you enjoyed reading it and that you learned new things.

The framework is still pretty new, so if you like it make sure to spread the word about **Phaser** and **this book**.

You now know enough to make almost any type of 2D games. So use your new knowledge wisely, and go make some awesome games :-)

More About the Author

If you want to hear more about me (Thomas Palef), you can:

- Go to my website **lessmilk.com** to play my games and read some tutorials
- Subscribe to **my newsletter** where I publish free content weekly
- Follow me on Twitter **@lessmilk_**
- Send me an email: **thomas.palef@gmail.com**

I spent a lot of time writing and proofreading this book, but I'm sure there are some mistakes left. So if you see any typos or have any feedback, please get in touch with me.

Thanks

A lot of people helped me directly or indirectly to make this book, and I wanted to thank some of them:

- Richard D. who showed me some best practice to follow for the game's code
- Theodore L. who beta tested the book and gave me some great insights
- Maryla U. who found a lot of ways to improve the book
- Ben B. who made sure that the book is accessible to everyone

---

Thanks for reading,

Thomas Palef