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*
- *First version published: 1 July 2014*
- *Current version: 2.1*
- *Purchased on **discoverphaser.com***

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

Contents

1 - Introduction 1

2 - Get Started 2

2.1 - Useful Links 3

2.2 - Set Up Phaser 4

2.3 - First Project 7

3 - Basic Elements 11

3.1 - Empty Game 12

3.2 - Add Player 17

3.3 - Create the World 23

3.4 - Add Coins 28

3.5 - Add Enemies 34

3.6 - Source Code 40

4 - Manage States 44

4.1 - Organization 45

4.2 - Index 48

4.3 - Boot 49

4.4 - Load 50

4.5 - Menu 52

4.6 - Play 54

4.7 - Game 56

5 - Jucify 57

5.1 - Add Sounds 58

5.2 - Add Animations 61

5.3 - Add Tweens 63

5.4 - Add Particles 68

5.5 - Better Camera 73

6 - Improvements 74

6.1 - Add Best Score 75

6.2 - Add Mute Button 77

6.3 - Better Keyboard Inputs 80

6.4 - Use Custom Fonts 82

6.5 - Better Difficulty 84

7 - Use Tilemaps 87

7.1 - Create Assets 88

7.2 - Display Tilemap 92

8 - Mobile Friendly 95

8.1 - Testing 96

8.2 - Scaling 98

8.3 - Touch Inputs 102

8.4 - Touch Buttons 104

8.5 - Device Orientation 111

8.6 - Native App 113

9 - Optimizations 117

9.1 - Create Atlas 118

9.2 - Use Atlas 122

9.3 - Create Audio Sprite 125

9.4 - Use Audio Sprite 128

9.5 - Concat and Minify 130

9.6 - Measure Improvements 132

9.7 - Code Refactoring 134

10 - More About Phaser 139

10.1 - New Functions 140

10.2 - Debugging 145

10.3 - Code Sprites 148

10.4 - Extend Phaser 151

11 - Next Steps 153

11.1 - Improve the Game 154

11.2 - Make New Games 156

11.3 - Conclusion 157

12 - Full Source Code 158

12.1 - The Code 159

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 work everywhere (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 one that is free, open source, and actively maintained, 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.

Its main advantage is that it is extremely powerful while still being simple to use.

What You Will Learn

We will focus on building a 2D platformer from scratch. When finished the game is going to be full featured: player, enemies, menu, animations, sounds, tilemaps, mobile friendly, and much more.

By the end of this book you will have a real game to play with, and enough knowledge to build your own games.

Requirements

I tried to make this book simple and accessible to beginners. That's why you just need to know the basics of programming to understand it. And if you know some HTML and Javascript that's even better.

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 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.

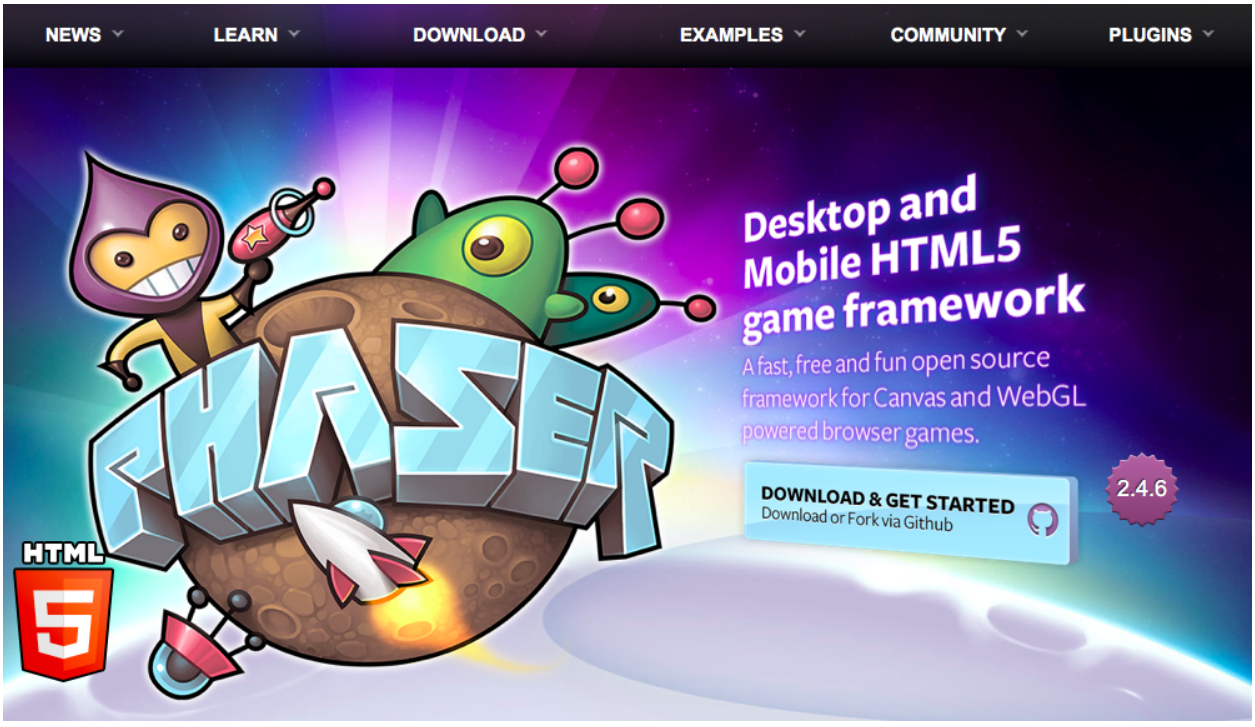
Phaser GitHub page github.com/photonstorm/phaser
The repository to download the framework. There are new versions available every few weeks.

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

Phaser code examples phaser.io/examples
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 2.5.0 that you can **download here**.

Phaser 2.x is supposed to be forward compatible. It means that the code from this book should still work when a new version is released. However, it's probably better that we both use the same 2.5.0 version.

The most important things to look for in the directory you just downloaded 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 text editor. Any editor will do the job, but I can recommend **Brackets**.
- A browser with the 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 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. Drag and drop a directory containing an HTML file in the **Brackets editor** and click on the small bolt icon that is in the top right corner. This will directly open your browser with a live preview from a webserver.
- Use apps. You can download **WAMP** (Windows) or **MAMP** (Mac). They both have a clean user interface with easy set up guides available online.
- 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.

If you are new to this, I strongly recommend to use the Brackets solution since it's the simplest one.

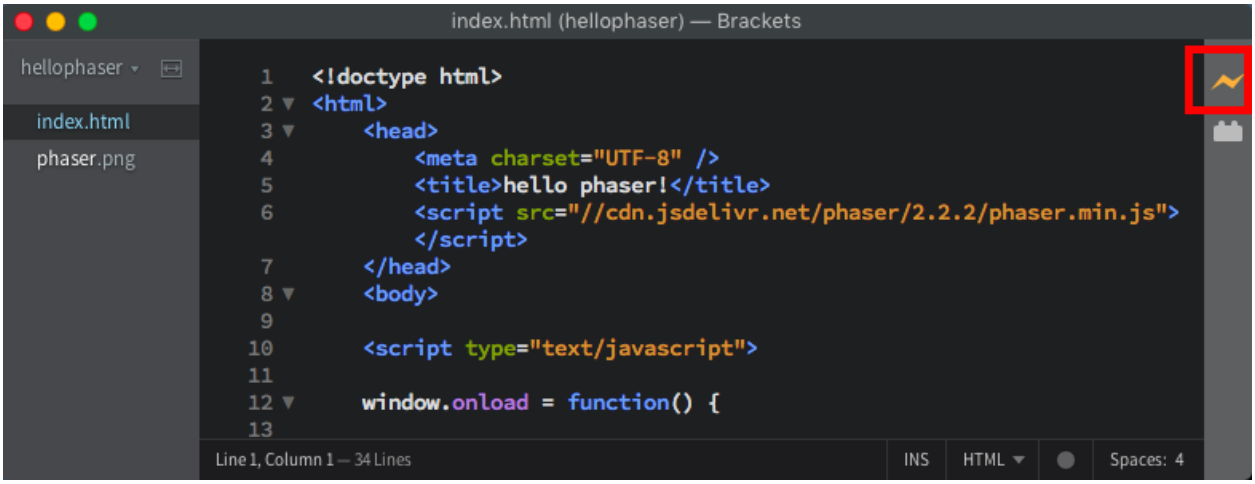
Test Phaser

Once all of the above is done, you should test that everything is working well. 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. Otherwise your webserver is not properly configured.

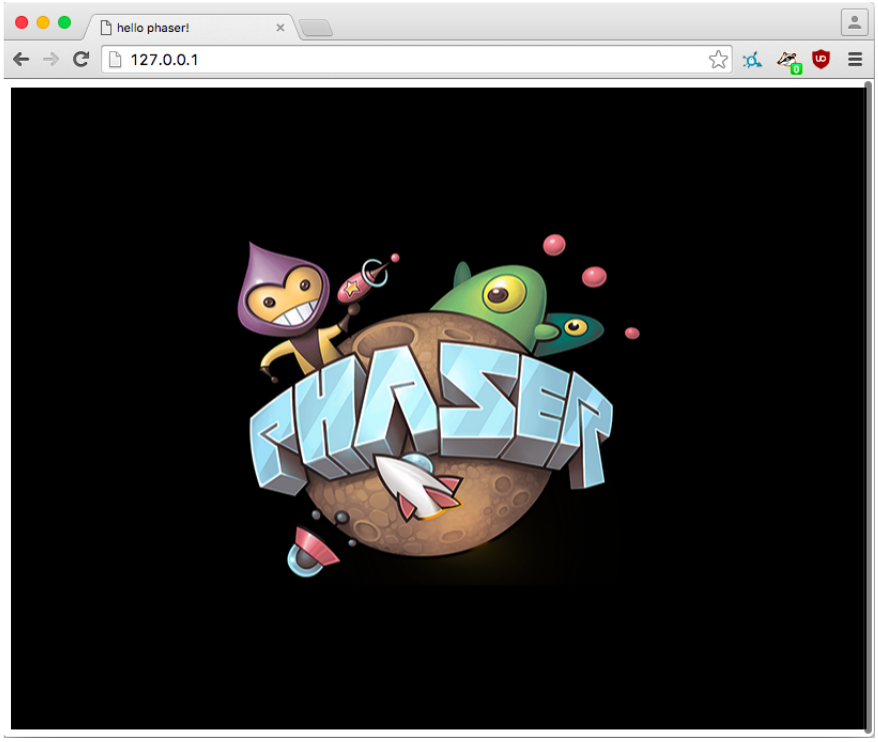
If you use Brackets:

1. Drag and drop the directory containing the index.html file in the editor.
2. Click on the small bolt icon in the top right corner to see the Phaser example.



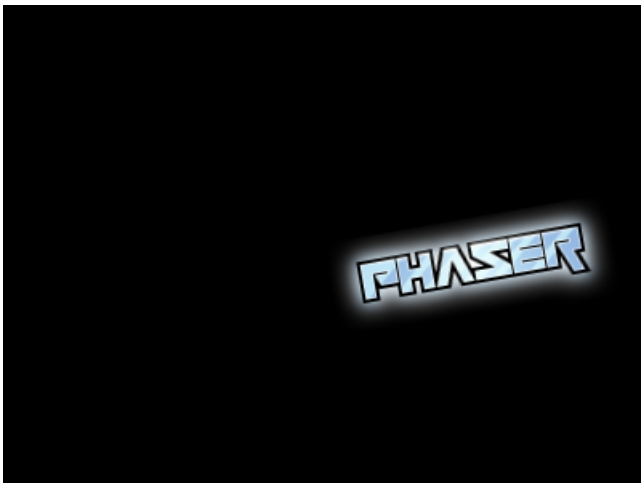
If you don't use Brackets:

1. Put the "resources/tutorials/01 Getting Started/hellophaser/" directory 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. Add "/hellophaser/" at the end of the URL to access the Phaser example.



2.3 - First Project

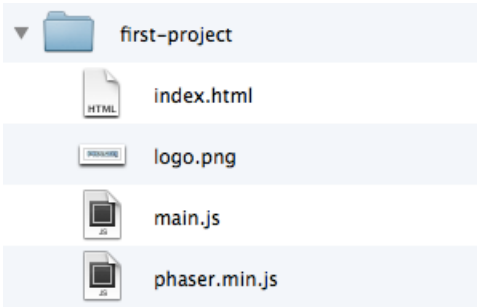
Now that Phaser is set up, it's time to code our first project. We will do something really simple: have a sprite rotate on the screen. This will give you a broad overview of how the framework works.



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.



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 main things:

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

The order of the Javascript files is important: main.js needs to be loaded last because it will use code from phaser.min.js.

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 will only have one.

We will create our state, initialize 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

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

// And 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 detail in the next chapter.

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

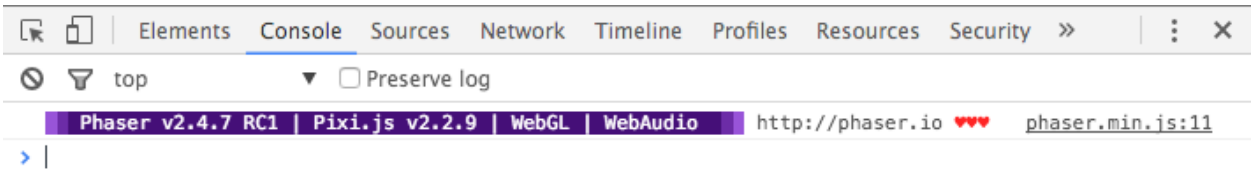
```
var mainState = {
    preload: function() {
        // Load the image
        game.load.image('logo', 'logo.png');
    },

    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. Now test it on your webserver (as explained in the previous part) and you should see a logo rotating on the screen.

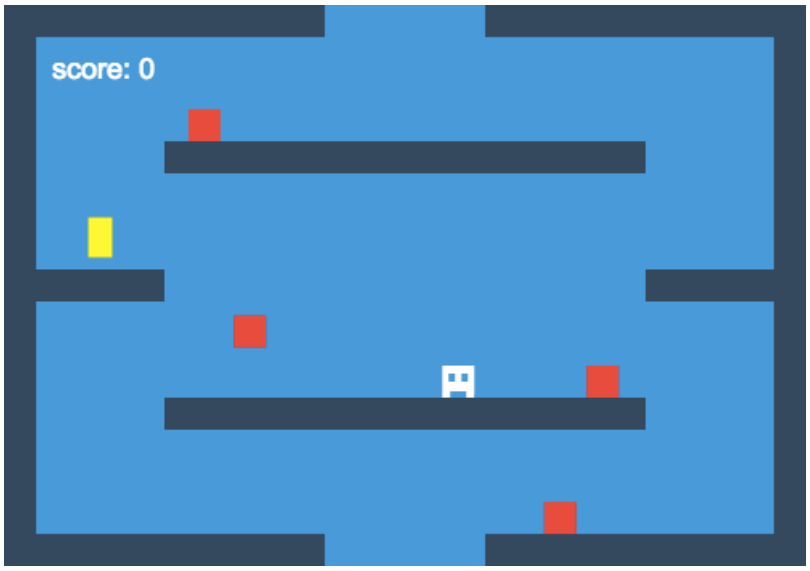
If it doesn't work, you need to bring up the console. Right-click anywhere on the page and select "inspect element". There you should see some error messages in the console tab that you need to fix.



3 - Basic Elements

Now that you are a little familiar with Phaser, we can actually start to make a game.

We will build a game inspired by Super Crate Box: a small guy that tries to collect coins while avoiding enemies.



When finished the game is going to be full featured: player, enemies, menu, animations, sounds, tilemaps, mobile friendly, and much more. All of this will be covered over the next 6 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

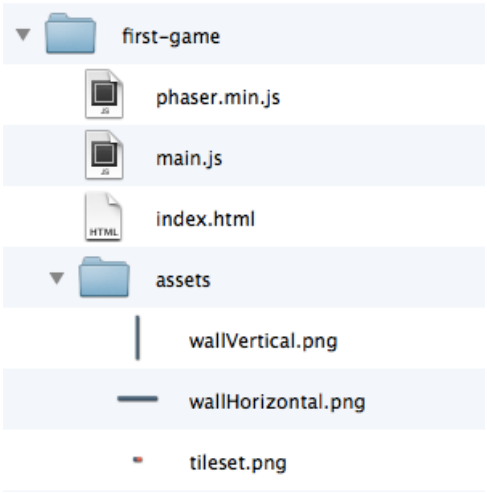
Let's start by creating an empty game, so this is going to be similar to what we did in the previous chapter. By the end of this part we will have this:

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

Set Up

First we need to create a new directory called "first-game". In it 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, to display the game. Use the one we made in the previous chapter.
- assets/, a directory with all the images and sounds. The assets can be **downloaded here**.



Code the Main File

The Javascript code for any new Phaser project is 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. Initialize Phaser. That's where we define the size of the game among other things.
3. Tell Phaser what the states of the game are.
4. And start one of the states, to actually start the game.

These 4 steps are explained below in detail.

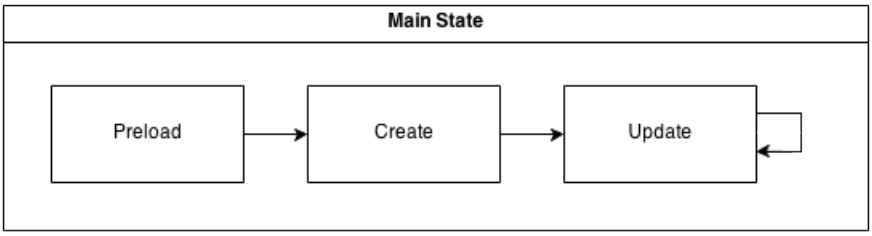
First, we create the states. For now we will have only one state that includes 3 of the default Phaser functions:

```
// 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.

Here's an image to better show you how they work together.



Then we initialize 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' 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 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 for 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 a 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 you want to build. In our case we will use Arcade physics, and to tell that to Phaser we need this line in the `create` function:

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

Crisp Pixels

We are going to use pixel art for the sprites of our game. To make sure that everything looks crisp, we should add this line in the `create` function:

```
    game.renderer.renderSession.roundPixels = true;
```

It will ensure that when sprites are displayed they are using integer positions. Without this, sprites can often render at sub-pixel positions, causing them to blur as Phaser tries to anti-alias them.

This is optional, but it will make our game look better.

Conclusion

At the end of every part you should test the game to check that everything is working properly.

Right now you should see an empty blue screen with no errors in the console.

3.2 - Add Player

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

Load the Player

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

- `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.

If we add a sprite at the 0 0 position, it would be in the top left corner of the game.

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

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

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

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

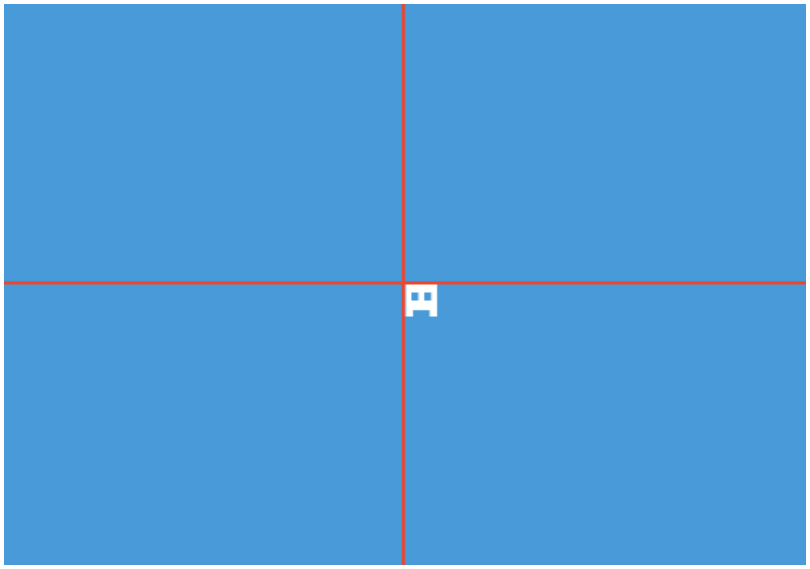
And we can do even better by using some predefined variables for the `x` and `y` positions:

```
this.player = game.add.sprite(game.width/2, game.height/2, 'player');
```

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

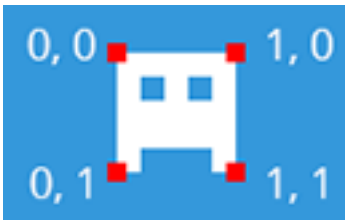
Anchor Point

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 as you can see here:



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
this.player.anchor.setTo(1, 0);

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

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



To center the player we need to set the anchor point to the middle of the sprite. So add this in the `create` function:

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

Add Gravity

Let's add some gravity to the player to make it fall, by adding 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 add this in the `create` function:

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

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

```
    movePlayer: function() {
        // If the left arrow key is pressed
        if (this.cursor.left.isDown) {
            // Move the player to the left
            // The velocity is in pixels per second
            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 on the ground
        if (this.cursor.up.isDown && this.player.body.touching.down) {
            // Move the player upward (jump)
            this.player.body.velocity.y = -320;
        }
    },
```

We created the `cursor` and the `player` variables in the `create` function, but we are using them in `movePlayer`. That works because we added the `this` keyword to make these variables accessible everywhere in the state.

And lastly we have to call `movePlayer` inside of the `update` function:

```
    // We have to use 'this.' to call a function from our state
    this.movePlayer();
```

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. Here are the main ones:

```
// Change the position of the sprite
sprite.x = 21;
sprite.y = 21;

// 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;

// Change the color of the sprite
sprite.tint = 0xff0000;

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

// Return false if the sprite was killed
sprite.alive;
```

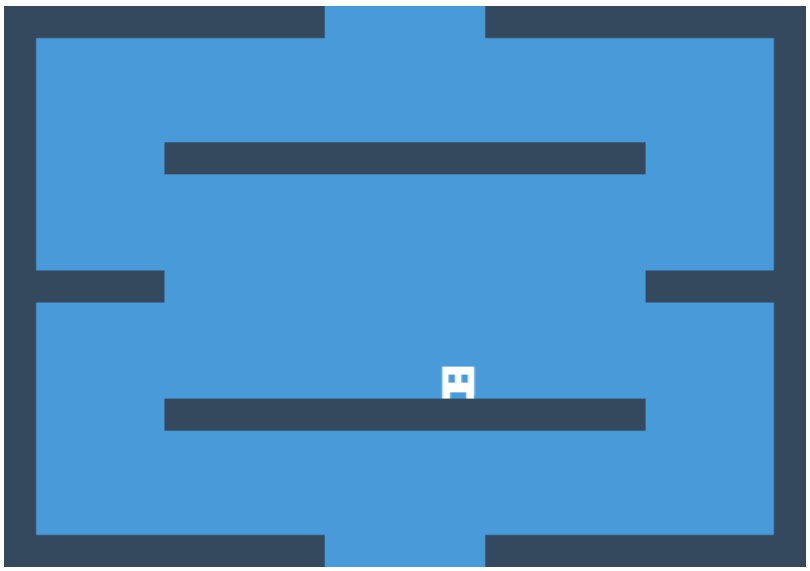
Conclusion

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. That's what we are going to do in this part.



Load the Walls

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

As we explained previously, we need to start by loading our new 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 (for collisions with the player)
game.physics.arcade.enable(leftWall);

// Set a property to make sure the wall won't move
// We don't want to see it 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 (like sprites) to share some common 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 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 group with Arcade physics
    this.walls = game.add.group();
    this.walls.enableBody = true;

    // Create the 10 walls in the group
    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%) and 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 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 `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. 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 call `playerDie`.

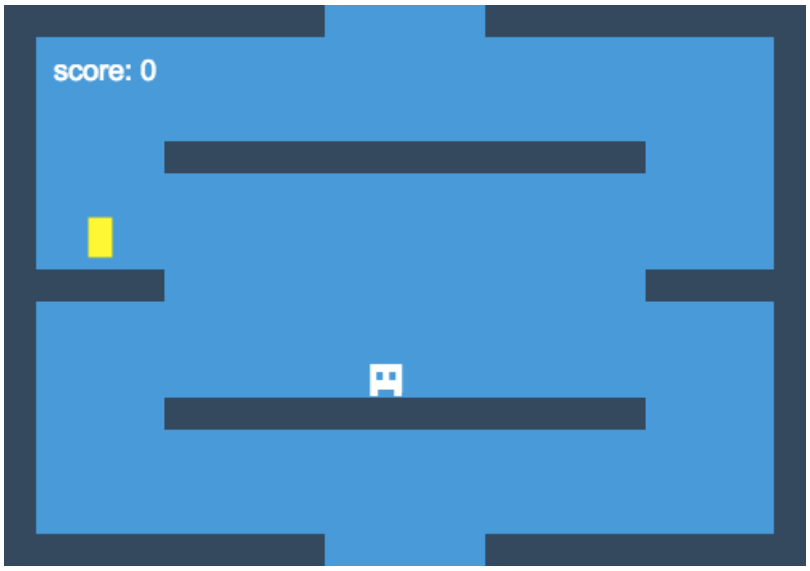
```
    if (!this.player.inWorld) {
        this.playerDie();
    }
```

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 it will change position each time the player takes it. Let's see how we can do that.



Load and Add the Coin

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 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' });

    // Initialize 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`, `backgroundColor`, 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, process, context)`
  - `objectA`: the first object to check.
  - `objectB`: the second object to check.
  - `callback`: the function that gets called when the 2 objects overlap.
  - `process`: if this is set then `callback` will only be called if `process` returns `true`.
  - `context`: the context in which to run the `callback`, most of the time it will be `this`.

Every time a function has a `callback` there will be a `context` parameter. In this book we will always set the `context` to `this`. This way we will be able to use any of the state's variables in the `callback`.

In our case we want to call `takeCoin` each time the player and a coin overlap, so we 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 by using its 'text' property
        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.

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 combine these 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);
```

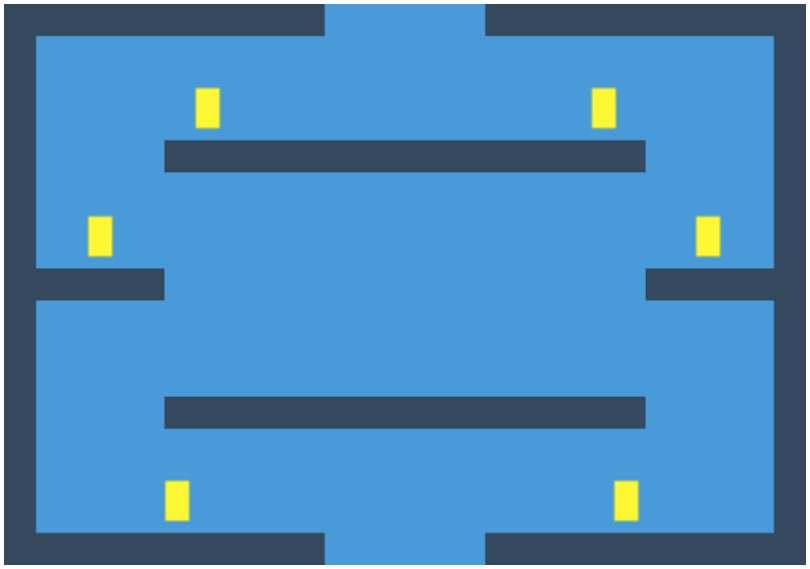
So we could replace the `this.coin.kill` in the `takeCoin` function by something like this:

```
// Get 2 random numbers
var newX = game.rnd.integerInRange(0, game.width);
var newY = game.rnd.integerInRange(0, game.height);

// 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 pick 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 with 'game.rnd.pick'
        var newPosition = game.rnd.pick(coinPosition);

        // 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, 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 it has the same position as the current coin.
- If it's the same, we remove it from the array with the `splice` function.
- This way we end up with only 5 positions in the array that are all different from where the current coin is.

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

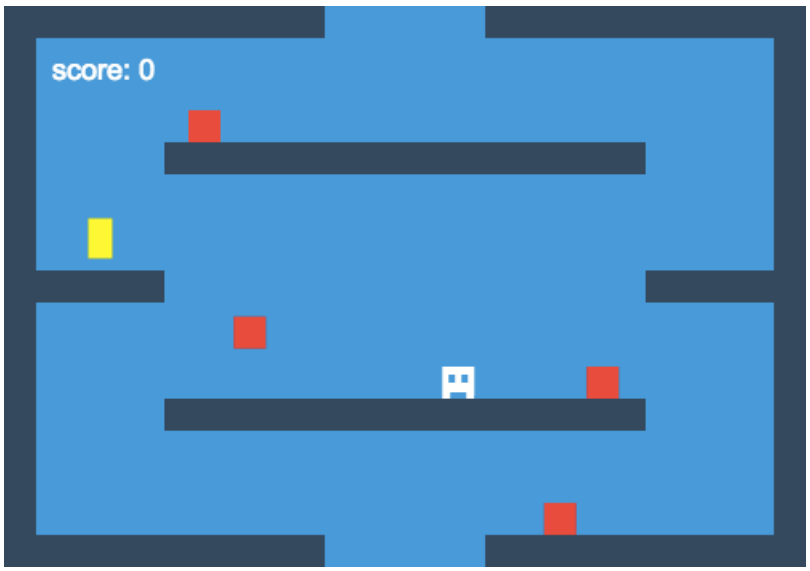
```
takeCoin: function(player, coin) {
    // Update the score
    this.score += 5;
    this.scoreLabel.text = 'score: ' + this.score;

    // 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 lots of enemies, we should use groups as we did with the walls. However, this time we can do even better by doing some "recycling":

- We create a dozen of enemies in advance. They are all "dead" by default since they shouldn't appear in the game at first.

- If we need an enemy, we pick one from the dead ones and display it on the screen.
- When we no longer need the enemy, we mark it as dead.

This way we keep reusing the same enemies over and over again, without the need to create new ones. And that will improve the game's performances.

Here's what we should add in the `create` function to initialize our group:

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

// Create 10 enemies in the group with the 'enemy' image
// 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, context)`
  - `delay`: the delay in ms between each `callback`.
  - `callback`: the function that will be called.
  - `context`: 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;
    }

    // Initialize the enemy
    enemy.anchor.setTo(0.5, 1);
    enemy.reset(game.width/2, 0);
    enemy.body.gravity.y = 500;
    enemy.body.velocity.x = 100 * game.rnd.pick([-1, 1]);
    enemy.body.bounce.x = 1;
    enemy.checkWorldBounds = true;
    enemy.outOfBoundsKill = true;
},
```

There are a lot of new things that are important here, 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 is no dead enemy 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 initialize 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.width/2, 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 `game.rnd.pick([-1, 1])` that randomly return 1 or -1 to have a velocity of 100 or -100:

```
    enemy.body.velocity.x = 100 * game.rnd.pick([-1, 1]);
```

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 (no bounce) and 1 (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 summarize what the `addEnemy` function does:

1. Get a dead sprite from the group.
2. If there is no dead sprite available, do nothing.
3. Otherwise initialize the sprite: position, speed, etc.
4. And make sure it dies at some point to never run out of dead enemies.

It's quite common to have functions like this in a Phaser game to create enemies, bullets, clouds, 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. Simply add 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 dead/alive sprites in a group
group.countDead();
group.countLiving();

// Add an object to a group. It can be a sprite, a label, etc.
group.add(object);
```

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 this 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);
    game.renderer.renderSession.roundPixels = true;

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

    this.player = game.add.sprite(game.width/2, game.height/2,
        'player');
    this.player.anchor.setTo(0.5, 0.5);
    game.physics.arcade.enable(this.player);
    this.player.body.gravity.y = 500;

    this.createWorld();

    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.enemies = game.add.group();
```

```
        this.enemies.enableBody = true;
        this.enemies.createMultiple(10, 'enemy');
        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();
```

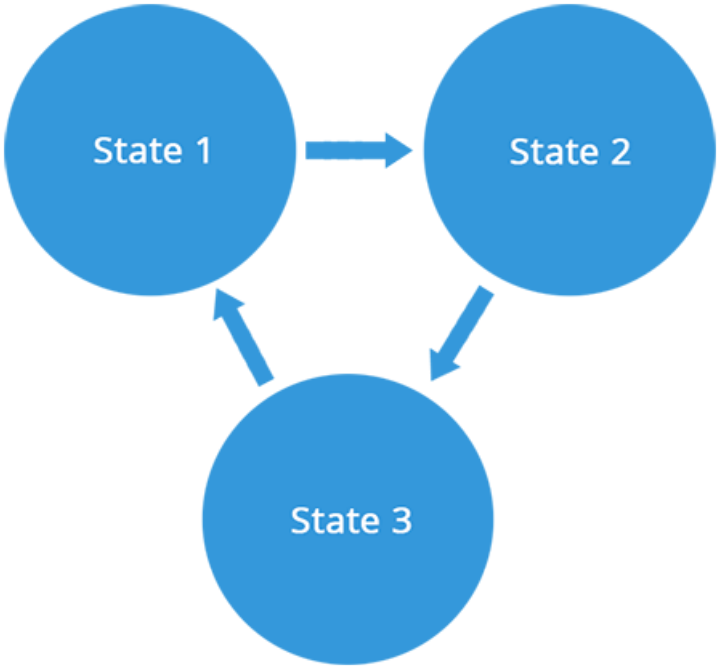
```
    },

    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 = game.rnd.pick(coinPosition);
        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.width/2, 0);
        enemy.body.gravity.y = 500;
        enemy.body.velocity.x = 100 * game.rnd.pick([-1, 1]);
        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 by the end of this chapter we will have a nice loading scene, a cool menu, and the game itself.



4.1 - Organization

Currently our game has only one state, and that creates some issues:

- 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.

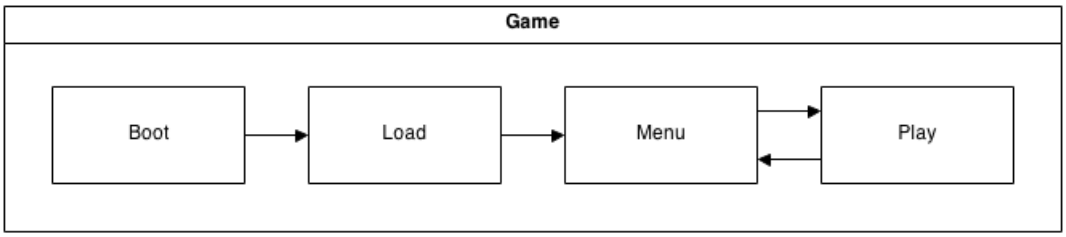
```
// Create one state called 'oneState'
var oneState = {

    // Define all the functions of the state

    preload: function() {
        // This function will be executed at the beginning
    },

    create: function() {
        // This function is called after the preload function
    },

    update: function() {
        // This function is called 60 times per second
    },
```

```
    // 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 like this:



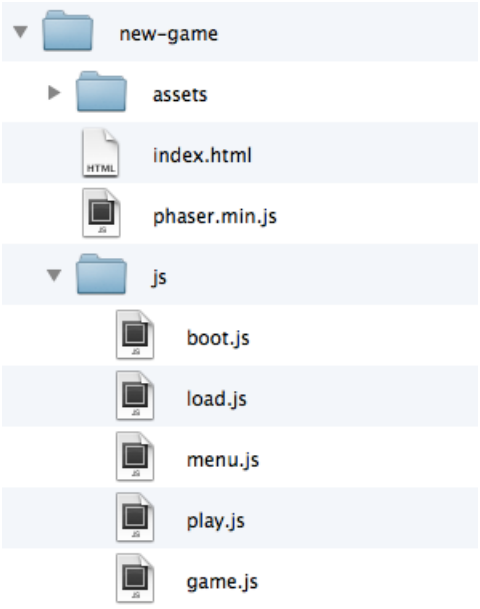
- Boot. This is the first state of the game.
- Load. It will load all the game's assets.
- Menu. That's the menu of the game.
- Play. This is where the actual game will be played. When dying, we will go back to the menu.

File Structure

We have to do some changes to the structure of our project's directory:

- Add an empty js/ directory.
- Rename main.js into play.js, and put it inside js/.
- Create 4 empty Javascript files in the js/ directory:
  - boot.js, the boot state.
  - load.js, the load state.
  - menu.js, the menu state.
  - game.js, that will initialise all of our states.

The new directory should look like this:


new-game
assets
index.html
phaser.min.js
js
boot.js
load.js
menu.js
play.js
game.js

4.2 - Index

The first thing to do is to edit the index.html file. The code is the same as before, except that we are loading all the Javascript files from the “js” directory.

```
<!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>
```

Make sure that game.js is the last file called since it will contain referances to functions defined in the other files.

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

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.
- It's a good practice to have a state that can run some code before loading all the assets to initialize some settings.

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

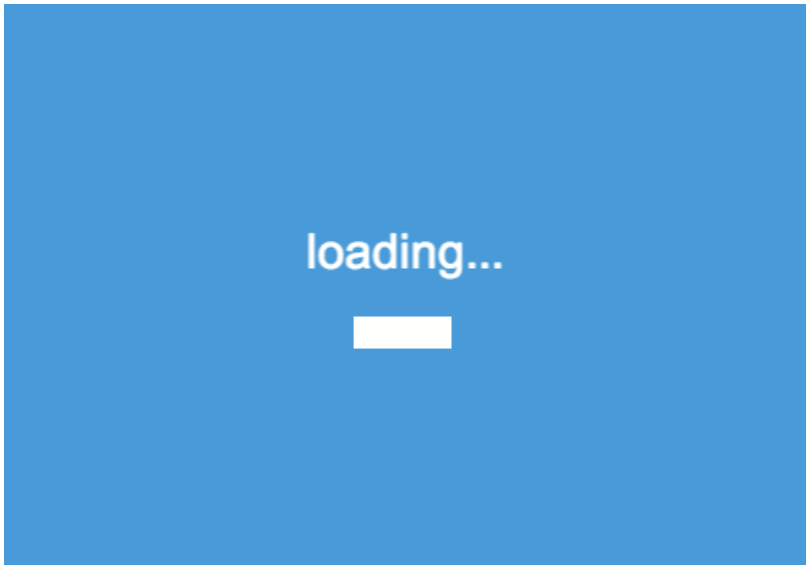
- Load the 'progressBar' image.
- Set some settings that we previously had in the main state.
- Start the load state.

Here's the code. Since we don't need the `update` function we simply removed it.

```
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);
        game.renderer.renderSession.roundPixels = true;

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

4.4 - Load

The load state is also quite simple. It will preload all the assets of the game and display 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.

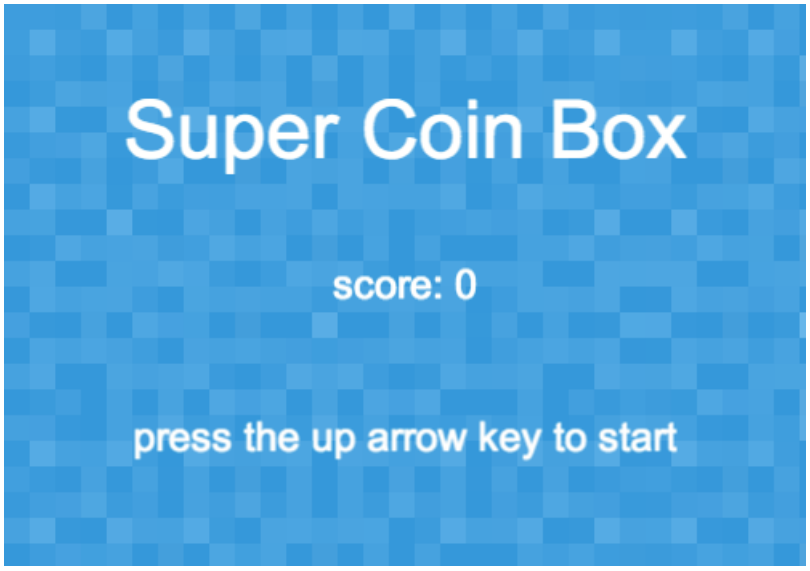
```
var loadState = {

    preload: function () {
        // Add a 'loading...' label on the screen
        var loadingLabel = game.add.text(game.width/2, 150,
            'loading...', { font: '30px Arial', fill: '#ffffff' });
        loadingLabel.anchor.setTo(0.5, 0.5);

        // Display the progress bar
        var progressBar = game.add.sprite(game.width/2, 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 take care of scaling up the 'progressBar' 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.width/2, 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.width/2, game.height/2,
            '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.width/2, game.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
        // When pressed, call the 'start'
        var upKey = game.input.keyboard.addKey(Phaser.Keyboard.UP);
        upKey.onDown.add(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's 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 it. Otherwise the labels would be displayed below the background 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 its 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 for the background color, Arcade physics, and `roundPixels` are now in the boot.js file.
- The `this.score` variable is replaced by `game.global.score`.
- The `playerDie` function starts the menu scene.
- And the Phaser initalization code is removed.

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

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

    // Removed the preload function

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

        // Then everything is the same, except at the end...

        // replace 'var score = 0' by this
        game.global.score = 0;
    },

    update: function() {
        // No changes
    },

    movePlayer: function() {
        // No changes
    },
```

```
    takeCoin: function(player, coin) {
        // Use the new score variable
        game.global.score += 5;

        // Use the new score variable
        this.scoreLabel.text = 'score: ' + game.global.score;

        // Then no changes
    },

    updateCoinPosition: function() {
        // No changes
    },

    addEnemy: function() {
        // No changes
    },

    createWorld: function() {
        // No changes
    },

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

// Delete all Phaser initialization code
```

4.7 - Game

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

```
// Initialize 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 summarize 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 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 - Jucify

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. This is often called "jucify a game".

Some of the things we will cover include: sound effects, animations, tweens, particles, 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 some sound effects to the game, and you will see that this is quite simple.

Compatibility

There are a lot of audio formats we can use, the main ones are: wav, mp3, and ogg. 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 a format that works everywhere:

	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.

Background Music

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 file since it will probably be the biggest asset you will have. Games that take forever to load are something to avoid.

More About Sounds

Here are some other interesting things you can do with sounds:

```
//  Change the volume of the sound (0 = mute, 1 = full sound)
sound.volume = 0.5;

// Increase the volume from 0 to 1 over the duration specified
sound.fadeIn(duration);

// Decrease the volume from its current value to 0 over the duration
sound.fadeOut(duration);
```

The `fadeIn` and `fadeOut` functions are especially usefull when playing a background music.

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`. This function needs to know the width and height of each frame. In this case it's 20px by 20px.

```
// Replace this in the load.js file
game.load.image('player', 'assets/player.png');

// By this
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 we 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 with the frames to add in the right order.
- `frameRate`: the speed of the animation, in frames per second.
- `loop`: if set to `true` the animation will loop indefinitely.

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:

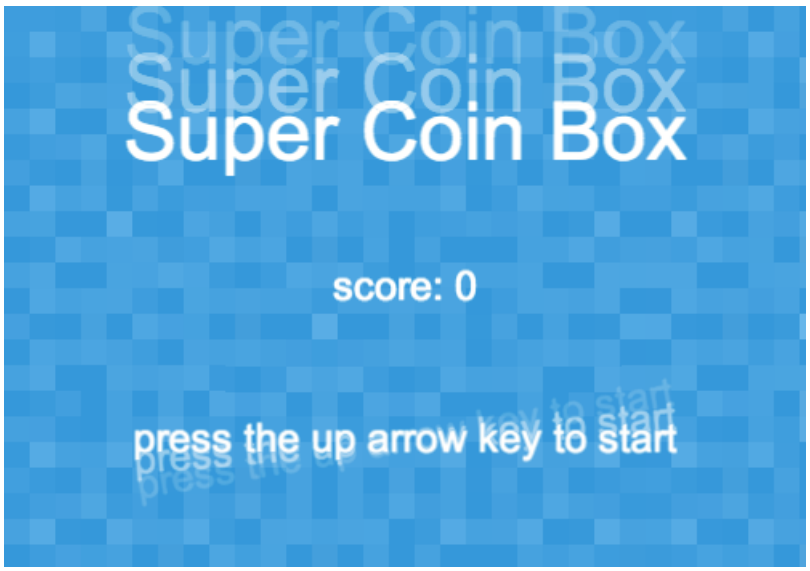
```
movePlayer: function() {
    if (this.cursor.left.isDown) {
        this.player.body.velocity.x = -200;
        this.player.animations.play('left'); // Left animation
    }
    else if (this.cursor.right.isDown) {
        this.player.body.velocity.x = 200;
        this.player.animations.play('right'); // Right animation
    }
    else {
        this.player.body.velocity.x = 0;
        this.player.animations.stop(); // Stop animations
        this.player.frame = 0; // Change frame (stand still)
    }

    // Then no changes to the jump code
},
```

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 we need to set the label's 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.width/2, -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 1000ms
    tween.to({angle: 2}, 1000);

    // And get back to our initial position in 500ms
    tween.to({angle: 0}, 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}, 1000)
        .to({angle: 0}, 500).loop().start();
```

Notice that the rotation 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 for a short amount of time. To do so we add this in the `takeCoin` function:

```
    game.add.tween(this.player.scale).to({x: 1.3, y: 1.3}, 100)
        .yoyo(true).start();
```

The `yoyo` function will do the previous tween in reverse. So after growing the player in 100ms, it will get back to its original size in 100ms.

More About Tweens

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

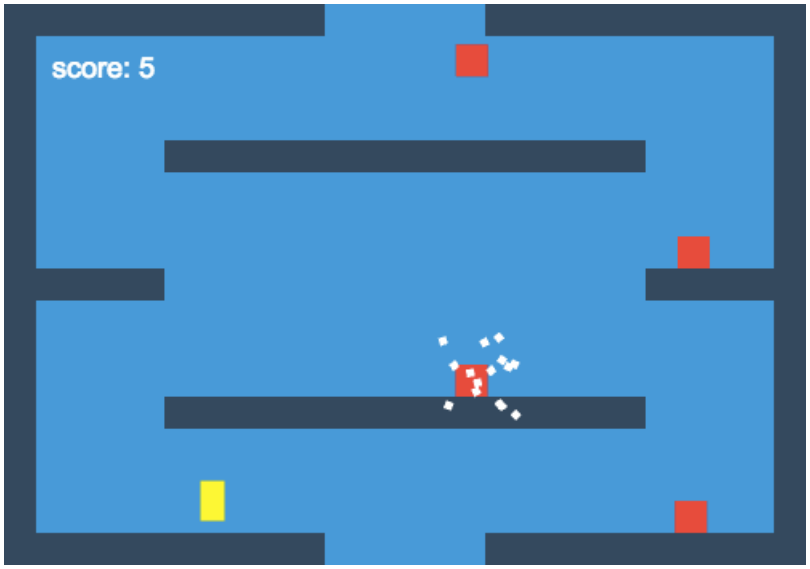
```
    // Add a 100ms delay before the tween starts
    tween.delay(100);

    // 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);
```

Anything that has a number can be tweened. So it can be: x/y position, angle, x/y scale, alpha, volume (for a sound), and so on.

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 an enemy hits him.



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

// Scale the particles from 2 time their size to 0 in 800ms
// Parameters are: startX, endX, startY, endY, duration
this.emitter.setScale(2, 0, 2, 0, 800);

// Use no gravity
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 can 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 update the `playerDie` function:

```
    playerDie: function() {
        // Set the position of the emitter on top of the player
        this.emitter.x = this.player.x;
        this.emitter.y = this.player.y;

        // Start the emitter by exploding 15 particles that will live 800ms
        this.emitter.start(true, 800, null, 15);

        // Play the sound and go to the menu state
        this.deadSound.play();
        game.state.start('menu');
    },
```

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, 800, 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 need to make sure we won't call `playerDie` or `movePlayer` when the player is dead. So add this in the `update` function, just after all the `collide` and `overlap`:

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

More About Particles

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

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

    // Set min and max rotation velocity
    emitter.setRotation(min, max);

    // Change the alpha value over time
    emitter.setAlpha(startAlpha, endAlpha, duration);

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

5.5 - Better Camera

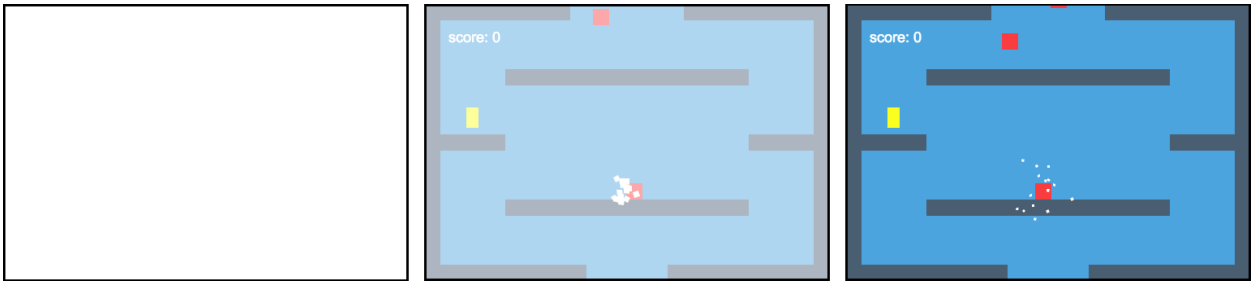
Phaser has a camera object that takes care of displaying the game on the screen. And with it, we can easily add some interesting effects to our game.

Flash Effect

For example we could make the camera flash. The idea is to display a color over the whole screen for a short amount of time, and then to fade it out nicely.

To do so, add this line in the `playerDie` function:

```
    // Flash the color white for 300ms
    game.camera.flash(0xffffff, 300);
```



Shake Effect

But we can do better than a flash: a camera shake.

Replace the previous line by this, to see the game screen shake.

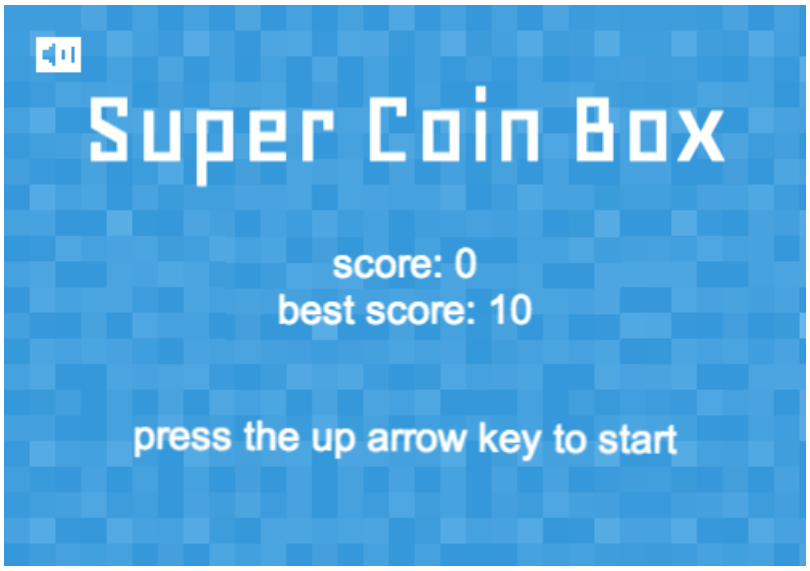
```
    // Shake for 300ms with an intensity of 0.02
    game.camera.shake(0.02, 300);
```

I can't show that with screenshots, but it really adds more impact to the player's death.

6 - Improvements

Our game is now fun to play but still misses some important features like a best score, a mute button, an increasing difficulty, and so on.

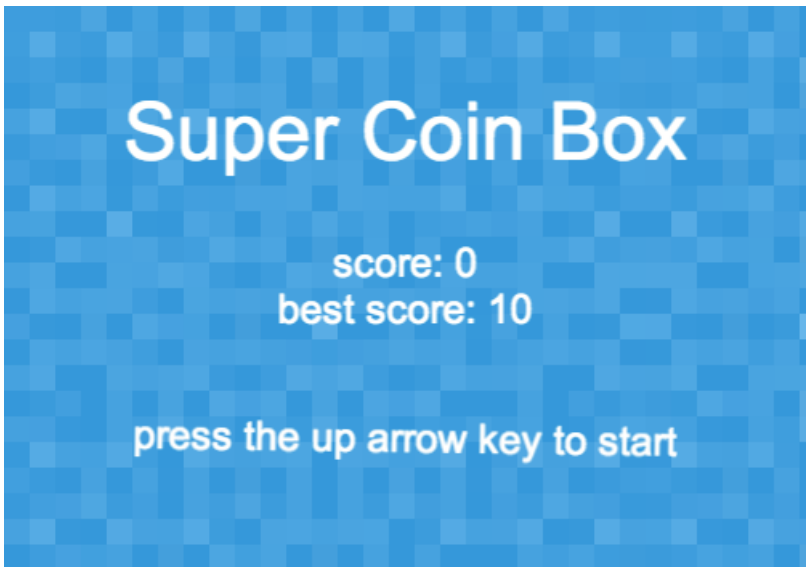
That's what we will cover in this chapter.



6.1 - Add 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 have a best score.



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 `localStorage.getItem`.

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

```
var scoreLabel = game.add.text(game.width/2, game.height/2,
    'score: ' + game.global.score,
    { font: '25px Arial', fill: '#ffffff' });
```

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

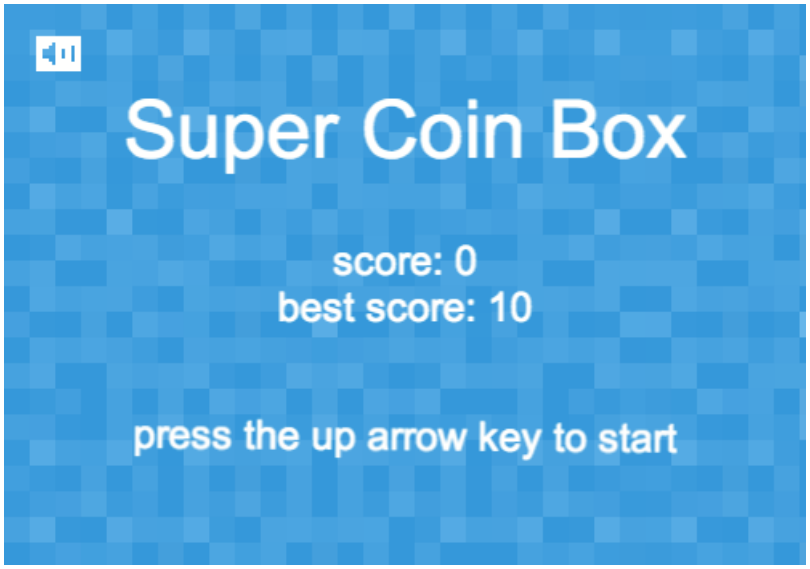
```
var text = 'score: ' + game.global.score + '\nbest score: ' +
    localStorage.getItem('bestScore');

var scoreLabel = game.add.text(game.width/2, game.height/2, text,
    { font: '25px Arial', fill: '#ffffff', align: 'center' });
```

The `\n` will add a line break. 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` retrieves the newest best score.

6.2 - Add 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);
```

Remember that the last 2 parameters are the width and height of each individual image in the spritesheet.

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

We can show the button in the top left corner of the menu with `game.add.button`.

- `game.add.button(x, y, name, callback, context)`
  - `x`: position x of the button.
  - `y`: position y of the butto.
  - `name`: the name of the image to display.
  - `callback`: the function called when the button is clicked.
  - `context`: 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 button that calls the 'toggleSound' function when pressed
    this.muteButton = game.add.button(20, 20, 'mute', this.toggleSound,
        this);
```

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

```
    // Function called when the 'muteButton' is pressed
    toggleSound: function() {
        // Switch the 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? By default the button displays the speaker with sound, so that's what we will see despite the fact that the game is muted.

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

```
// If the game is already muted, display the speaker with no sound
this.muteButton.frame = game.sound.mute ? 1 : 0;
```

6.3 - 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 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 `wasd` 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:

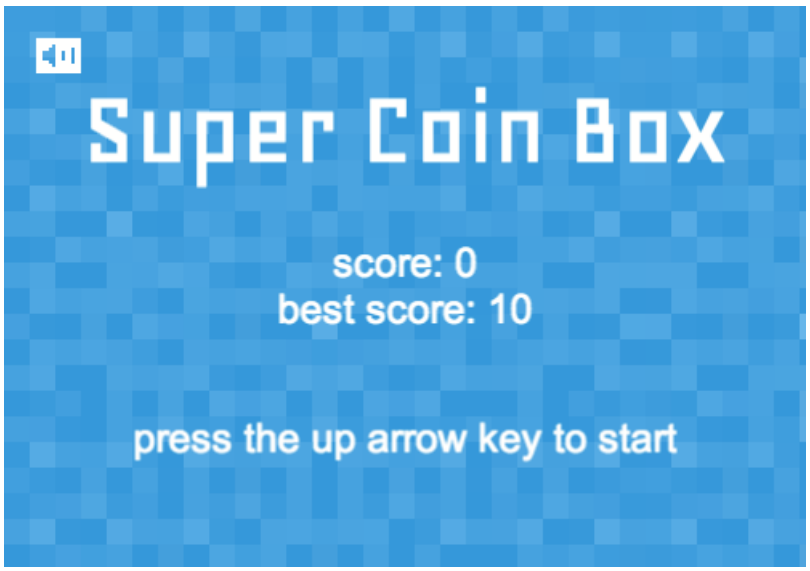
```
    movePlayer: function() {
        // If the left arrow or the A key is pressed
        if (this.cursor.left.isDown || this.wasd.left.isDown) {
            this.player.body.velocity.x = -200;
            this.player.animations.play('left');
        }

        // 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 (no changes)
        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.

6.4 - 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 because we need to be sure that the font is loaded before the game starts. A simple way to achive that is to add a text (at least one character) on the index.html page that will use the new font:

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

And then we create a new CSS class called `hiddenText` that will use the new font and hide the text. This way we are sure that the font will be loaded without actually seeing any change on the page.

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

    .hiddenText {
        font-family: Geo;
        visibility: hidden;
        height: 0;
    }
</style>
```

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.width/2, -50, 'Super Coin Box',
    { font: '70px Geo', fill: '#ffffff' });
```

6.5 - Better Difficulty

Our game is quite hard and it's difficult to score more than 30 points. We should try to make the game easier at first, and 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 to do is to build our own timer that will create new enemies every 2.2 seconds just like before.

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 the 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 of the game?
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 our 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 point:
  - `delay = max(4000 - 3000*0/100, 1000) = max(4000 - 0, 1000) = 4000`.
  - That's one new enemy every 4 seconds.

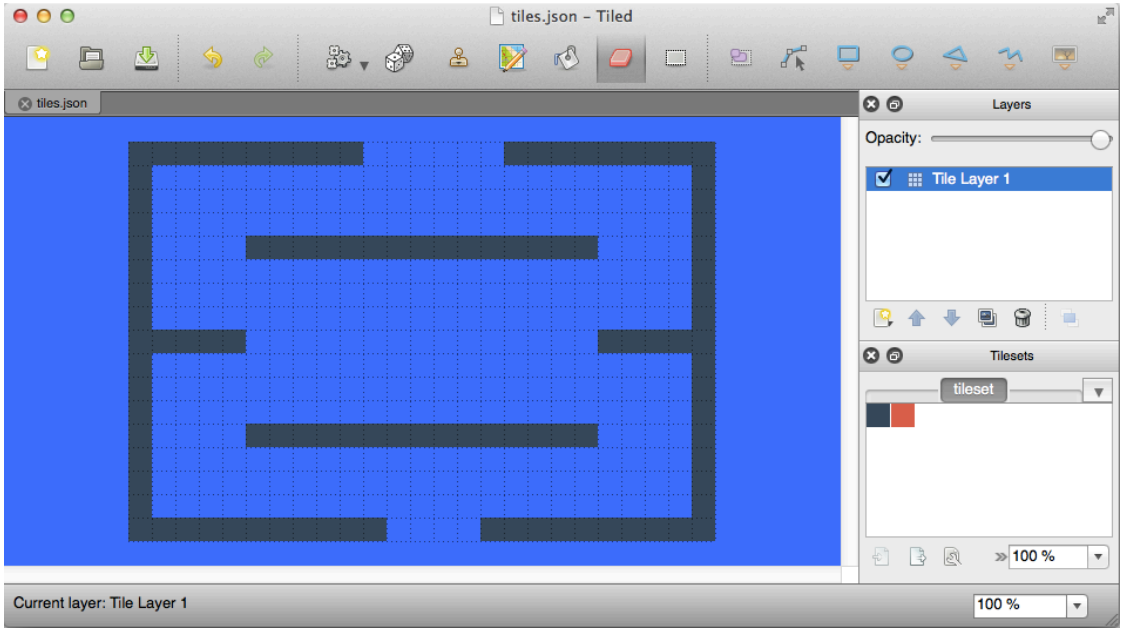
- If we have 50 points:
  - `delay = max(4000 - 3000*50/100, 1000) = max(4000 - 1500, 1000) = 2500`.
  - That's one new enemy every 2.5 seconds.
- If we have 100 points:
  - `delay = max(4000 - 3000*100/100, 1000) = max(4000 - 3000, 1000) = 1000`.
  - That's one new enemy per second.
- If we have 200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

7 - Use Tilemaps

In the beginning of the book we created the level manually by adding walls one by one. It works, 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.



7.1 - Create 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 spritesheet 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

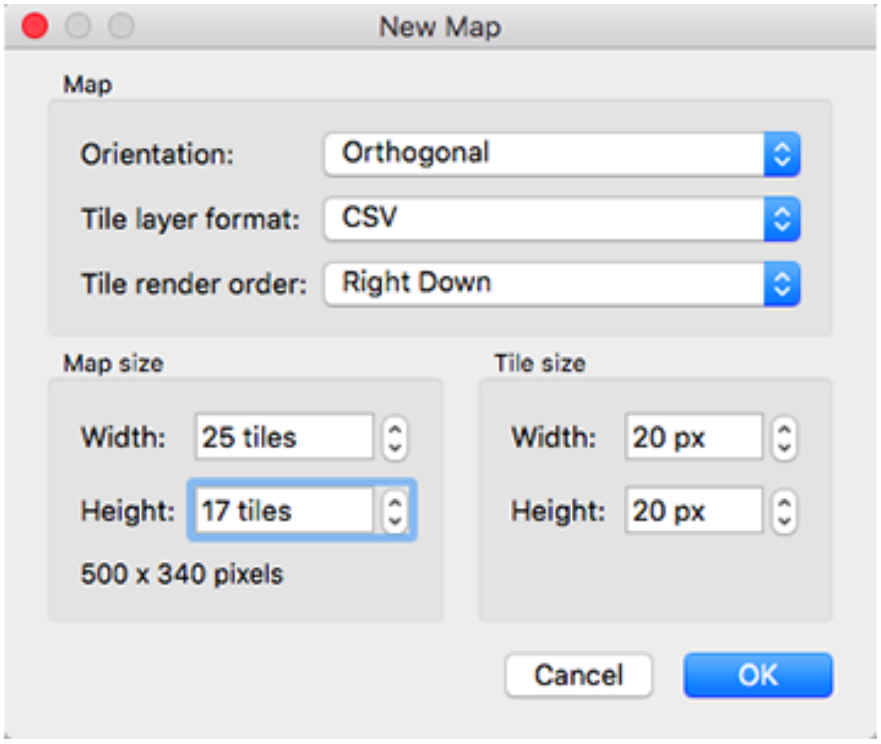
Our tileset will be really simple since we only need one tile: a 20 by 20 pixels dark blue square for the walls. But to better show you how this works we will add a second tile: a red square.

It looks like this:

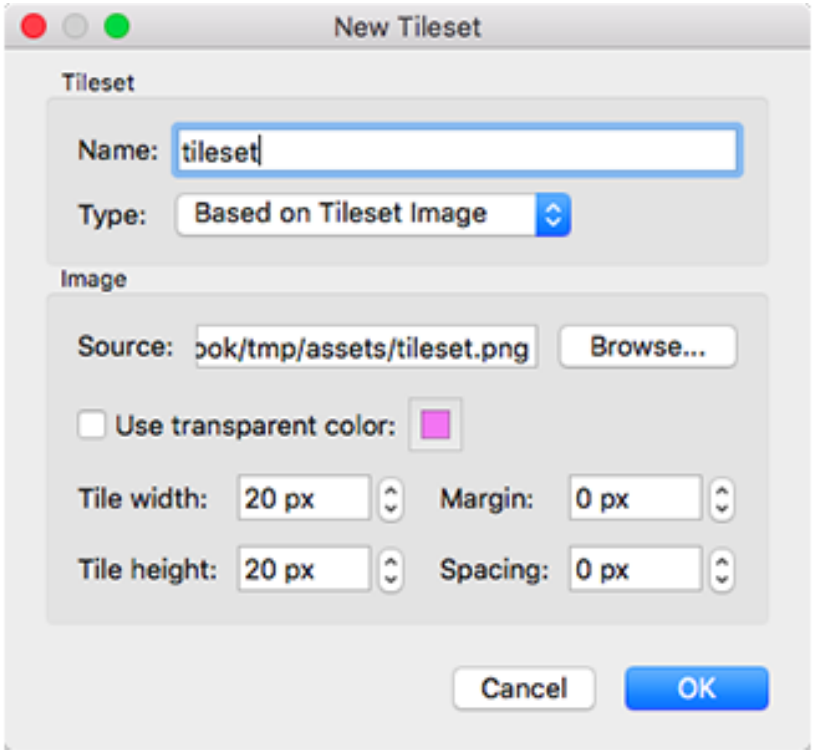
Tiled

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**.

Open the Tiled app, do “top menu > file > new”, and fill the form like this to create an empty tilemap:



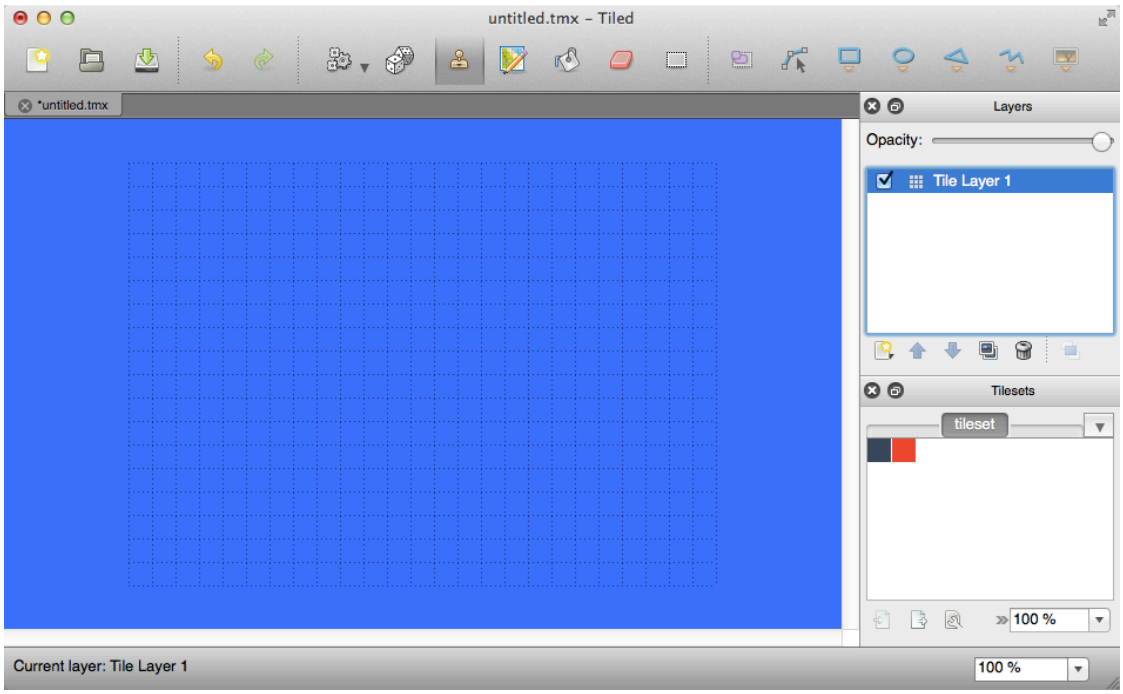
Then click on “top menu > map > new tileset” and do this:



Next do “top menu > map > map properties” to set a blue background color to the

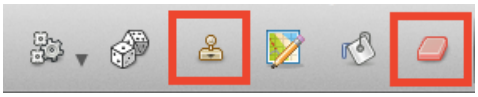
map.

Once done, you should see this on your screen:



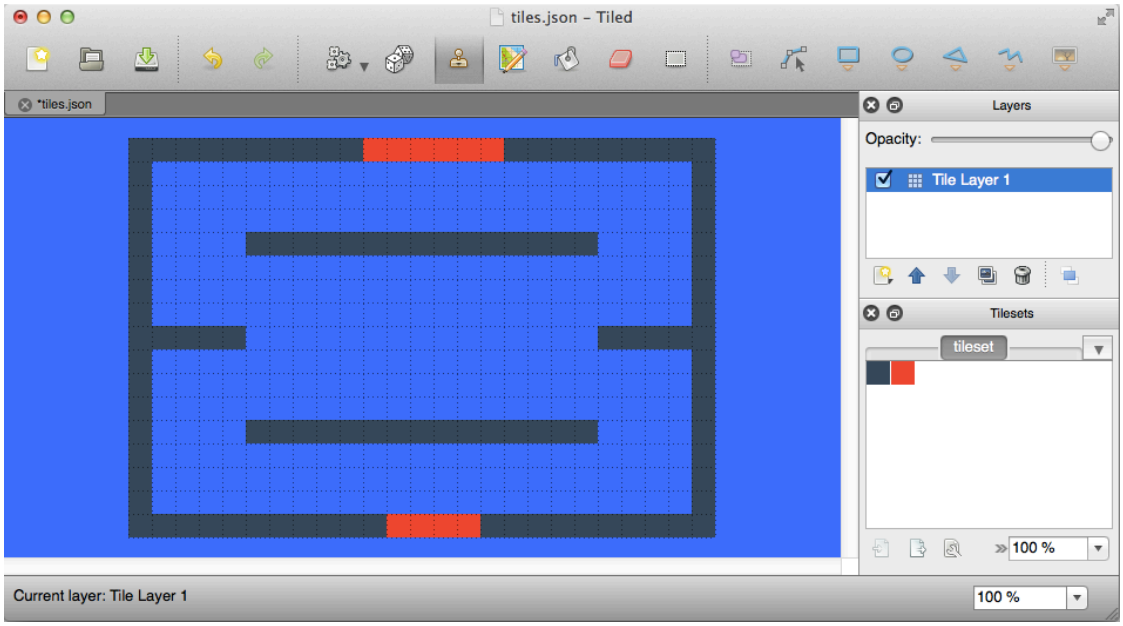
The Tilemap

Our empty tilemap is now properly set up, the next step is to design 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 at the top of Tiled.



And now simply draw a level 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 do “top menu > file > save”, select the json output format, and call it “map.json”. Or you can use the map.json that is already 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”). I strongly recommend you to do the same, otherwise you will need to manually edit the json file to make the game work.

7.2 - Display Tilemap

Now that everything is set up, we can get back to our 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 tilset element (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:

```
    // Replaced 'this.walls' by '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:

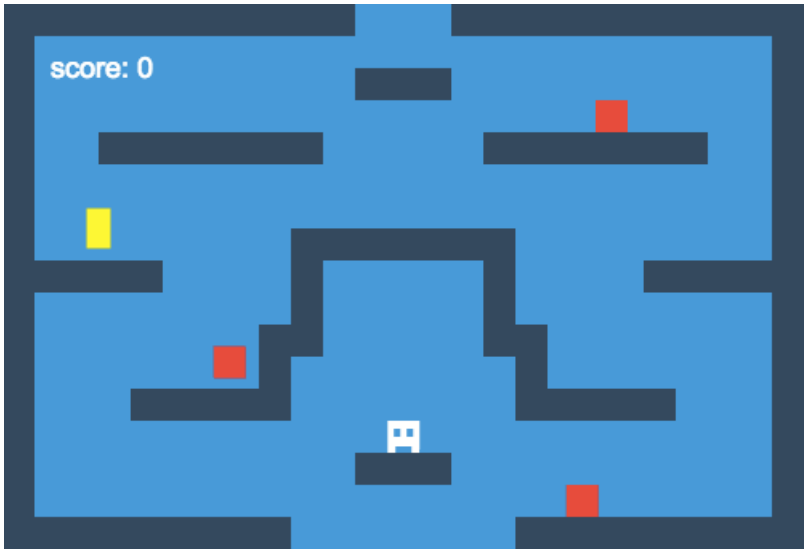
```
    if ((this.cursor.up.isDown || this.wasd.up.isDown)
        && this.player.body.onFloor()) {
        this.jumpSound.play();
        this.player.body.velocity.y = -320;
    }
```

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 change the level we just have to edit 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.

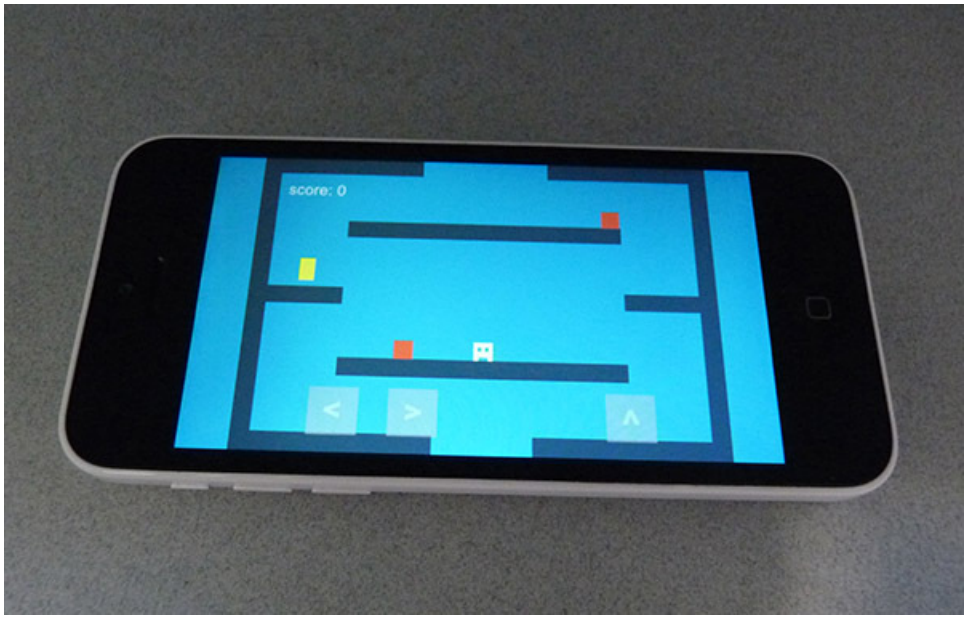
To learn more about all of this I recommend to directly look at the Phaser and Tiled documentations.

8 - Mobile Friendly

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

To make our game mobile friendly we will have to do some changes to our code that are covered in this chapter (scaling, touch inputs, and so on).

And in the last part we will see how to port our web game into a native app for iOS and Android.



8.1 - Testing

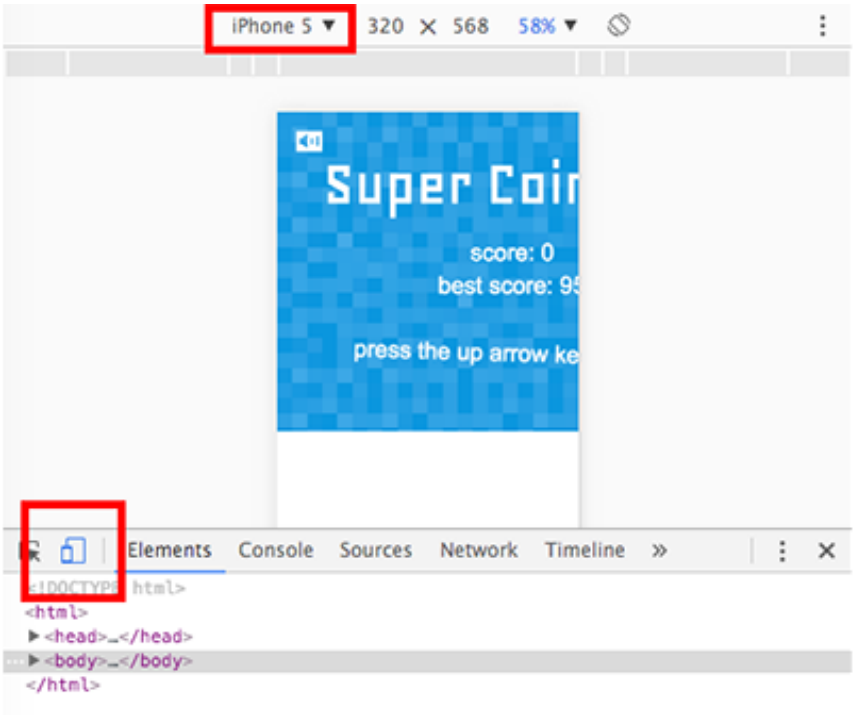
Before we start coding anything we should know how to test our game. There are 2 different ways to do that: on a computer and on a mobile device.

However, keep in mind that we haven't made the game mobile friendly yet. So 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 (right-click anywhere on the page and select "inspect element") and click on the small mobile icon in the upper left corner. Then select which device you want to emulate at the top of the window and reload the page.



You can press on the small icon in the top right to rotate the screen.

That's a really handy solution but it's 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 it.

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.

8.2 - Scaling

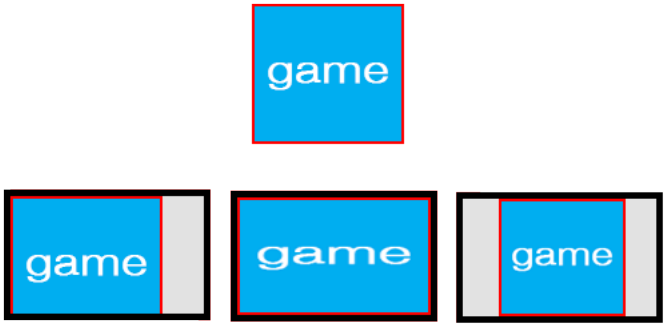
The main issue when making games for mobile is managing all the screen sizes. We will see below how we can handle that.

Types of Scaling

There are 3 main 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 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;

    // Set the min and max width/height of the game
    game.scale.setMinMax(game.width/2, game.height/2,
        game.width*2, game.height*2);

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

    // Add a blue color to the page to hide potential white borders
    document.body.style.backgroundColor = '#3498db';
}
```

It's always a good practice to specify a minimum and a maximum size 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 Game File

To make sure the game takes the whole page and is centered correctly, we should remove any mention of the 'gameDiv' when we create Phaser:

```
// Replace this in game.js
var game = new Phaser.Game(500, 340, Phaser.AUTO, 'gameDiv');

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

The last 2 parmeters are optionals, and their default value are `Phaser.AUTO` and `''`. So we can actually initialize Phaser like this:

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

Edit Index File

We also need to do some changes to the index.html file:

- Add a CSS rule to remove every margin and padding, to make sure there are no gaps between the game and the borders of the screen.
- Add a meta tag to scale the page properly on mobile devices.
- Remove the text “my first Phaser game” since we want the game to take all the space.
- Remove the “gameDiv”, as explained previously.

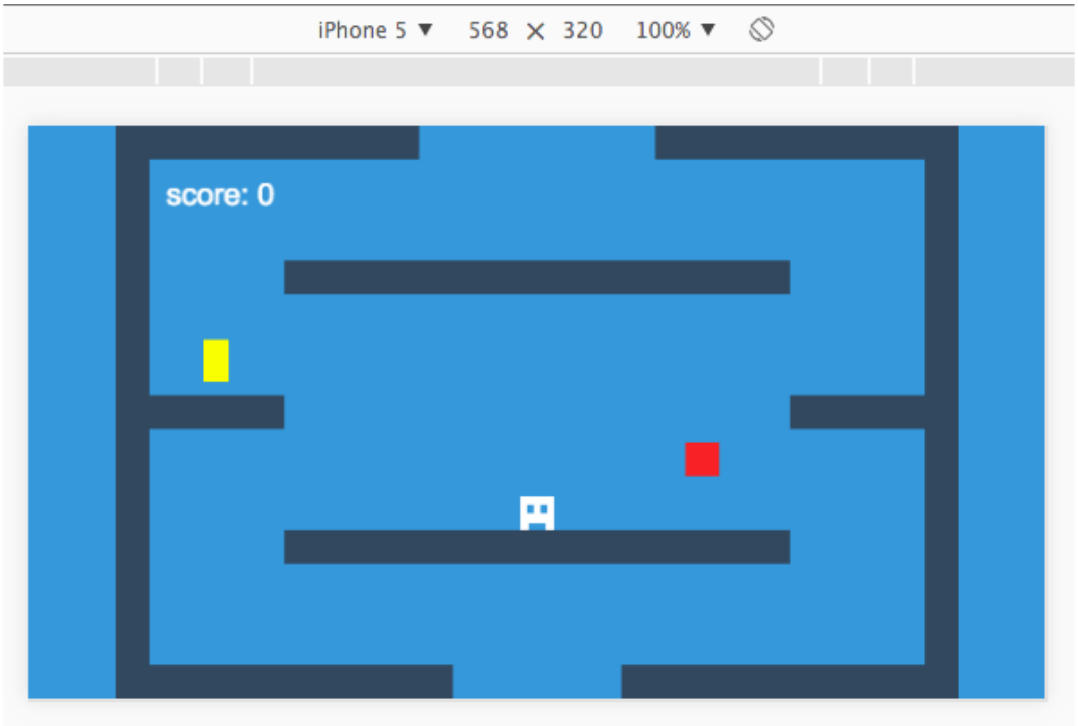
Here’s the new index.html with all the changes:

```
<!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;
                height: 0;
            }
        </style>

        <meta name="viewport" content="initial-scale=1
            user-scalable=no" />
```

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

</html>
```

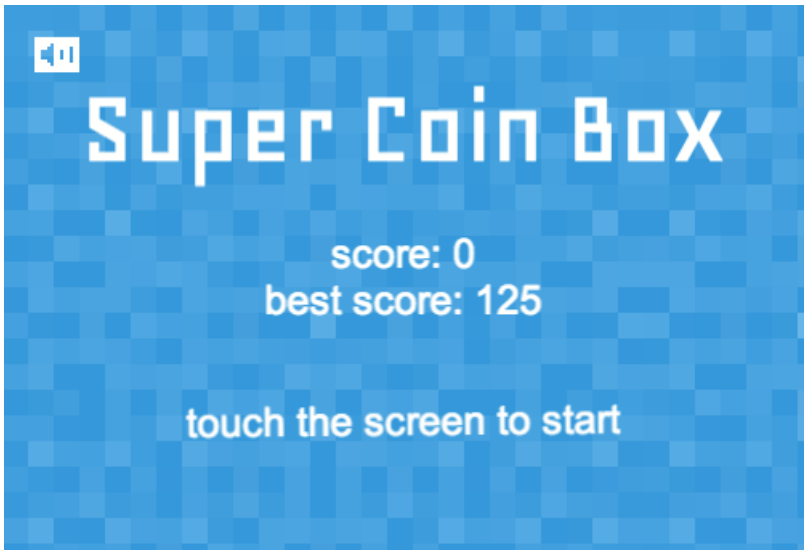
Result

The game looks way better now that it's properly scaled:



8.3 - Touch Inputs

Now the game is properly displayed on mobile devices, but we are stuck in 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.

Simply edit the 'startLabel' of the menu.js file like this:

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

// Display the text variable
```

```
    var startLabel = game.add.text(game.width/2, game.height-80, text,
        { font: '25px Arial', fill: '#ffffff' });
```

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.add(this.start, this);
```

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

```
    if (!game.device.desktop) {
        game.input.onDown.add(this.start, this);
    }
```

For your information the `input` can either be fingers or the mouse.

Fix Mute Button

A side effect of the code above is that as soon as we tap on the mute button on mobile, the game will start.

To prevent that, add this at the beginning of the `start` function:

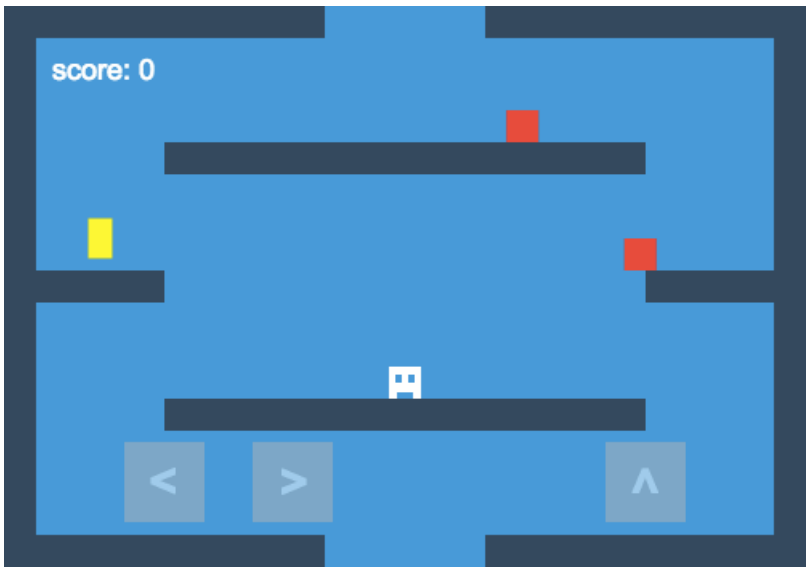
```
    // If we tap in the top left corner of the game on mobile
    if (!game.device.desktop && game.input.y < 50 && game.input.x < 60) {
        // It means we want to mute the game, so we don't start the game
        return;
    }
```

8.4 - 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 controller in the game.
- Handle touch gestures to control the player.
- Add custom buttons on the screen that we can press.

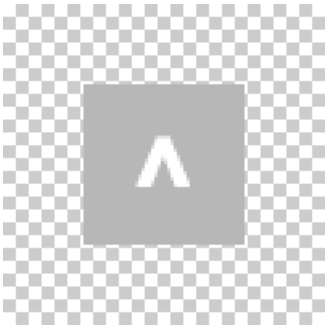
We will see how we can do the last option.



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 a little transparent to not hide the game behind it.

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

```
    addMobileInputs: function() {
        // Add the jump button
        var jumpButton = game.add.sprite(350, 240, 'jumpButton');
        jumpButton.inputEnabled = true;
        jumpButton.alpha = 0.5;

        // Add the move left button
        var leftButton = game.add.sprite(50, 240, 'leftButton');
        leftButton.inputEnabled = true;
        leftButton.alpha = 0.5;

        // Add the move right button
        var rightButton = game.add.sprite(130, 240, 'rightButton');
        rightButton.inputEnabled = true;
```

```
        rightButton.alpha = 0.5;
    },
```

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

```
    if (!game.device.desktop) {
        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
    sprite.events.onInputOver.add(callback, this);

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

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

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

We will use all these functions to make our buttons work.

Jump

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

```
    // Call 'jumpPlayer' when the 'jumpButton' is pressed
    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`. This way we 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's

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 event 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 currenlty 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 (no changes)
        var jumpButton = game.add.sprite(350, 240, 'jumpButton');
        jumpButton.inputEnabled = true;
        jumpButton.alpha = 0.5;
        jumpButton.events.onInputDown.add(this.jumpPlayer, this);

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

        // Add the move left button
        var leftButton = game.add.sprite(50, 240, 'leftButton');
        leftButton.inputEnabled = true;
        leftButton.alpha = 0.5;
        leftButton.events.onInputOver.add(this.setLeftTrue, this);
        leftButton.events.onInputOut.add(this.setLeftFalse, this);
        leftButton.events.onInputDown.add(this.setLeftTrue, this);
        leftButton.events.onInputUp.add(this.setLeftFalse, this);

        // Add the move right button
        var rightButton = game.add.sprite(130, 240, 'rightButton');
```

```
        rightButton.inputEnabled = true;
        rightButton.alpha = 0.5;
        rightButton.events.onInputOver.add(this.setRightTrue, this);
        rightButton.events.onInputOut.add(this.setRightFalse, this);
        rightButton.events.onInputDown.add(this.setRightTrue, this);
        rightButton.events.onInputUp.add(this.setRightFalse, this);
    },

    // Basic functions that are used in our callbacks

    setLeftTrue: function() {
        this.moveLeft = true;
    },

    setLeftFalse: function() {
        this.moveLeft = false;
    },

    setRightTrue: function() {
        this.moveRight = true;
    },

    setRightFalse: function() {
        this.moveRight = false;
    },
```

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 is new
            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 is new
            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.

Small Fix

But what if you start pressing on the left button, then drag your finger on the right button, and let go your finger? The player will keep moving right.

To fix that, add this at the beginning of the `movePlayer` function:

```
    // If 0 finger are touching the screen
    if (game.input.totalActivePointers == 0) {
        // Make sure the player is not moving
        this.moveLeft = false;
        this.moveRight = false;
    }
```

8.5 - Device Orientation

There's one last thing we could do to make our game better on mobile: prevent it from working in portrait orientation. Because when you hold your phone vertically, the game is too small to play comfortably.

Code

We will need a way to know when the device gets rotated. And when it does, we will check its new orientation:

- If it's portrait, we pause the game and display an error message.
- If it's landscape, we resume the game and remove the error message.

Here's a function that does just that, to add at the end of the play state:

```
orientationChange: function() {
    // If the game is in portrait (wrong orientation)
    if (game.scale.isPortrait) {
        // Pause the game and add a text explanation
        game.paused = true;
        this.rotateLabel.text = 'rotate your device in landscape';
    }
    // If the game is in landscape (good orientation)
    else {
        // Resume the game and remove the text
        game.paused = false;
        this.rotateLabel.text = '';
    }
},
```

In this code we used 2 new Phaser variables:

- `game.scale.isPortrait` that returns `true` when the game is currently in portrait orientation. We could have also used `game.scale.isLandscape`.

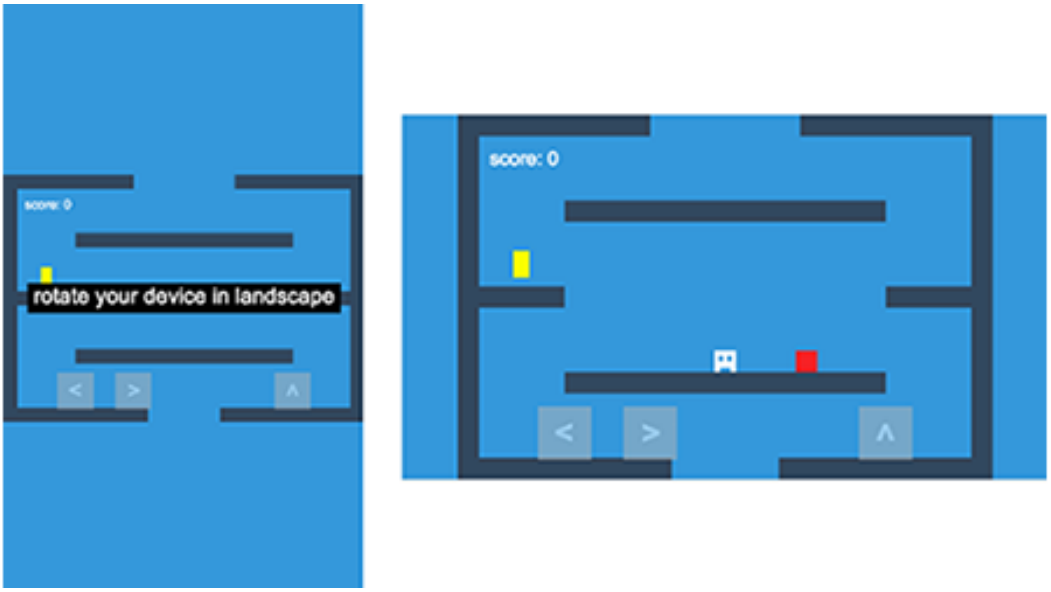
- `game.paused` that stops everything in the game when set to `true`.

And now we need to call that function in the `create` of the play state:

```
if (!game.device.desktop) {
    // Create an empty label to write the error message if needed
    this.rotateLabel = game.add.text(game.width/2, game.height/2, '',
        { font: '30px Arial', fill: '#fff', backgroundColor: '#000' });
    this.rotateLabel.anchor.setTo(0.5, 0.5);

    // Call 'orientationChange' when the device is rotated
    game.scale.onOrientationChange.add(this.orientationChange, this);

    // Call the function at least once
    this.orientationChange();
}
```

The Phaser function `onOrientationChange.add` will automatically call our new function whenever the device is rotated. But if the game starts in the wrong orientation our function won't be called. That's why I added a call to `orientationChange` at the end to make sure it is executed at least once.

Result



8.6 - Native App

Our game is now mobile friendly, it means that we can play it in the browser of any mobile device.

But we can do even better: transform our web game into a native app that can be downloaded on iOS and Android. We will see how to do that below, without writing a single line of code.

Main Solutions

There are 3 main services that let you convert an HTML5 game into a native app for free:

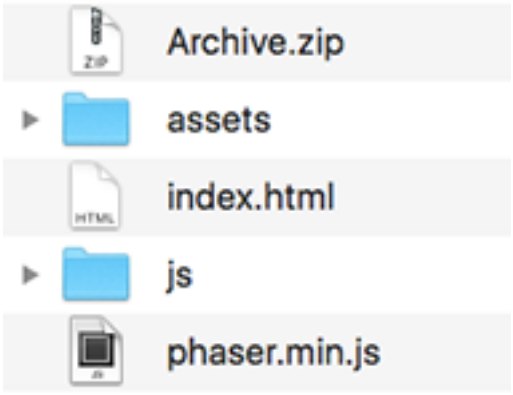
- **PhoneGap**, the most basic one.
- **CocoonJS**, offers the best performances.
- **Cordova**, the open source option.

In this part we will use CocoonJS.

Set Up

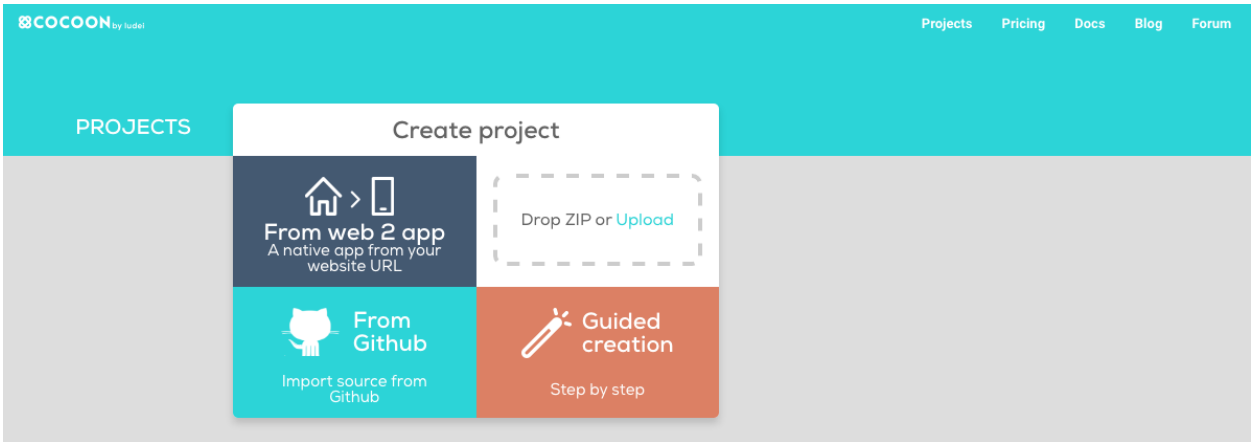
Since our game is already mobile friendly, the only thing we need to do is to zip it.

So open the game directory, and zip all the files. Don't zip the directory itself, but everything that's inside:

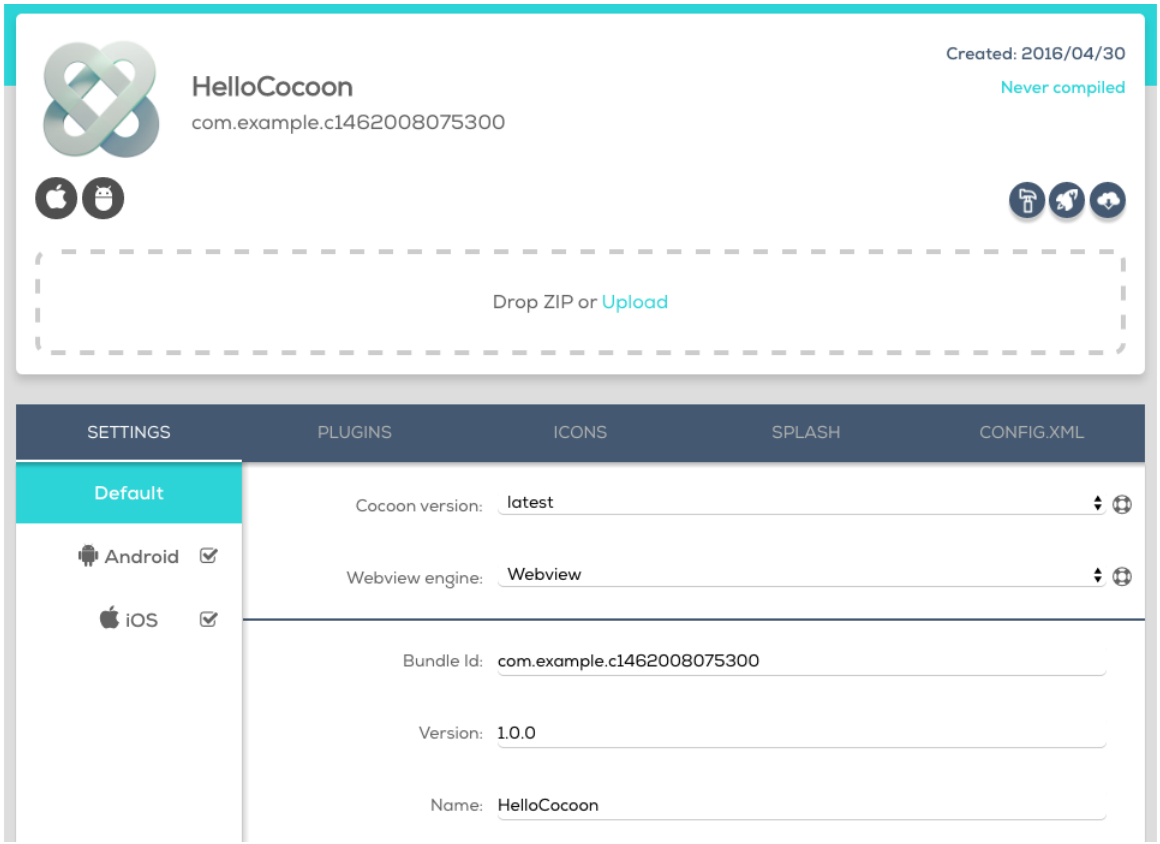


Build

First, you need to create a free account on CoocoonJS **here**. Once done, you will see this dashboard:



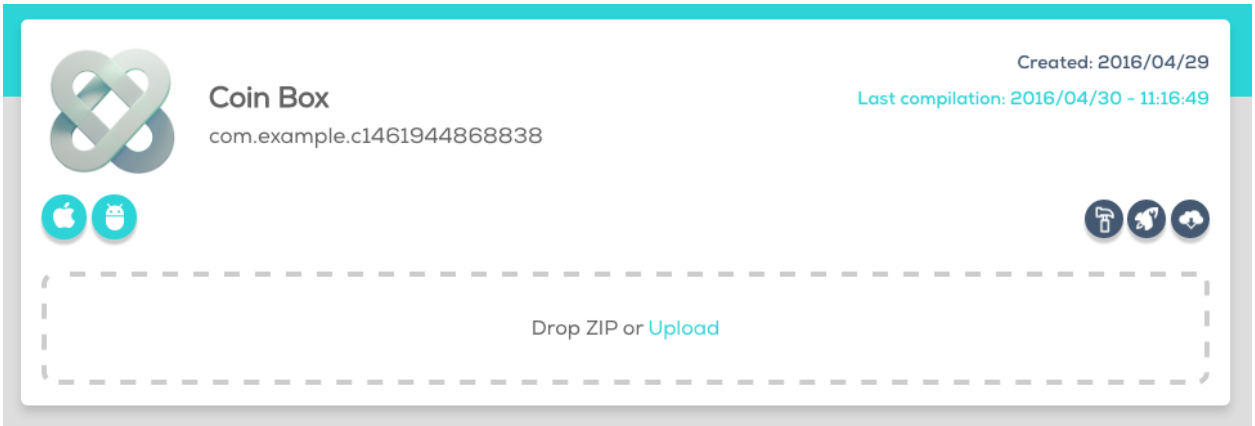
Click on the “upload zip” link, and select the zip we made earlier. The new project should look like this:



There you should edit some settings:

- The name of the app: Coin Box.
- The orientation of the game: landscape.
- Fullscreen: yes.

And press the save button. Next, click on the small hammer icon in the top right of the page to compile the app. Then wait about 5 minutes, and you will get an email to notify you that the compilation is finished. Download your app by clicking on the iOS or Android icon in the top left.



Test

You should now have 2 new files: .apk and .xcarchive. Let's see how to actually test these files.

The .apk file is for Android:

- On your mobile device do "settings > security" and check "unknown sources".
- Email yourself the andoid-debug.apk file.
- Click on the .apk from your device to download it, and click "install".

The .xcarchive is for iOS, and it's a lot more complicated to test. The process looks something like this:

- Create an Apple Developer account ($99), where you should:
  - Download and install a certificate on your computer.
  - Register your mobile device.

    - Get an ad hoc provisioning profile.
- Open the .xcarchive with Xcode and export it to an “ad hoc deployment” to get a .ipa file.
- Connect your iOS device to your computer.
- Open Xcode, do “top menu > window > devices”, and select your device.
- There click on the small “+” below the “installed apps”, and select the .ipa file.
- And finally you have the app on your device.

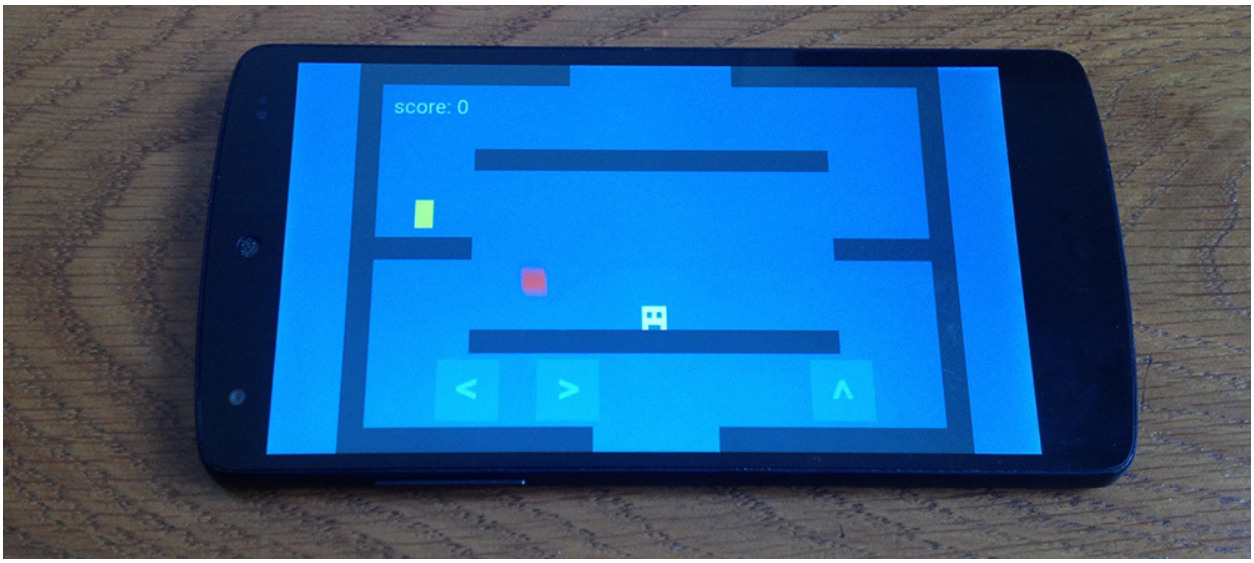
Submit

If you want to actually submit the game to the App Store or Play Store, you’ll need to do a few more things:

- Add all the required images for your app (icons, splash screen, etc). You can directly upload them on the CocoonJS project page.
- Create a Developer Account with Apple or Google, and this is not free.
- Add your developer keys in the CocoonJS project settings.
- And finally go through the process of submitting the app with Apple or Google.

After that you will have a game that people can download on their own mobile devices.

However don’t submit the game we are building together in this book, you should submit your own games.



9 - Optimizations

In this chapter we will see how to optimize our game in 2 ways:

- Make it load faster by reducing the number of HTTP requests.
- Improve the code readability with some refactoring.

Since our game is quite small you won't see a big change after the optimizations. But these are best practice that you could use for your own (bigger) games.

9.1 - Create Atlas

Our game is using a dozen of png files, it means that a dozen of HTTP requests need to happen to load all the sprites. That's a lot, but there's a way to drastically reduce this number.

The Idea

The idea is to combine all of our sprites into one big image. This way we load only one file instead of a dozen. However, this works only if we tell Phaser the coordinates and size of each of the sprites inside the big image. And doing that manually is going to be time consuming.

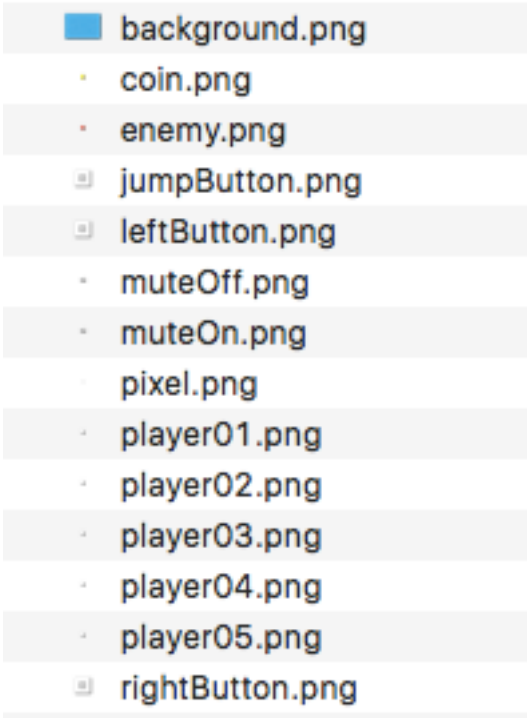
The solution is to create a single image containing all our sprites, along with a file that describes the position of each sprite in the image. This is called a texture atlas, and that's what we will do in this part.

Set Up

Start by downloading **TexturePacker**, it's a great tool to create atlases that has a free version.

Due to how TexturePacker works, we will need some slightly different assets. Each sprite must be in its own file, so our 2 spritesheets (muteButton and player) have to be split into multiple images.

You can **download here** a directory containing all the sprites we need for the atlas.

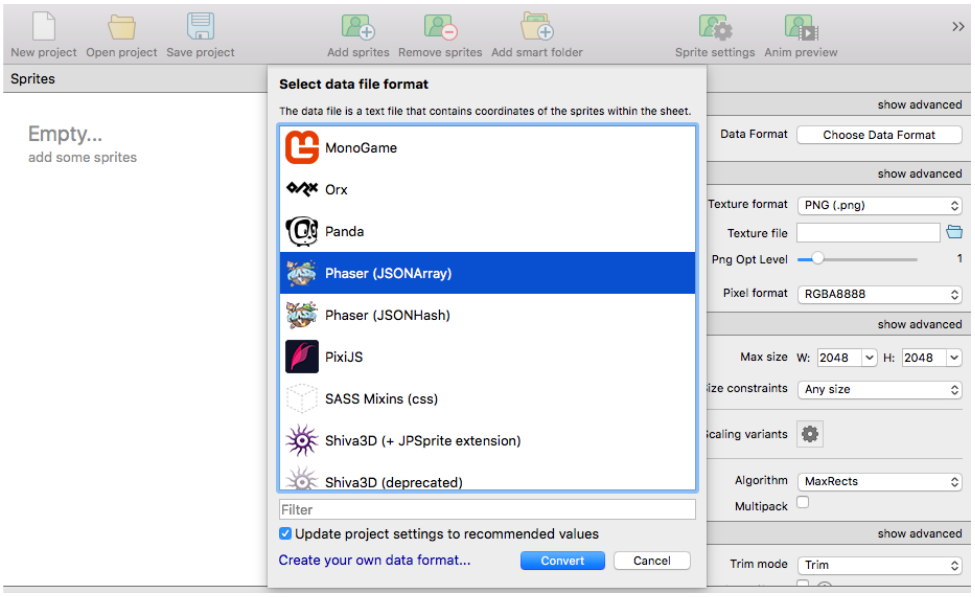


Notice that:

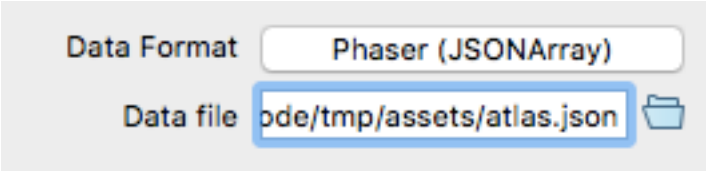
- There's no progressBar.png because we need to load it before all our assets, so we won't include it in the atlas.
- There's no tilset.png since Tiled is using this asset, and changing it might create some issues.
- Some of the names changed: 'muteOn' 'muteOff' for the mute button, and 'player01' 'player02' 'player03' 'player04' for the player.

Create Atlas

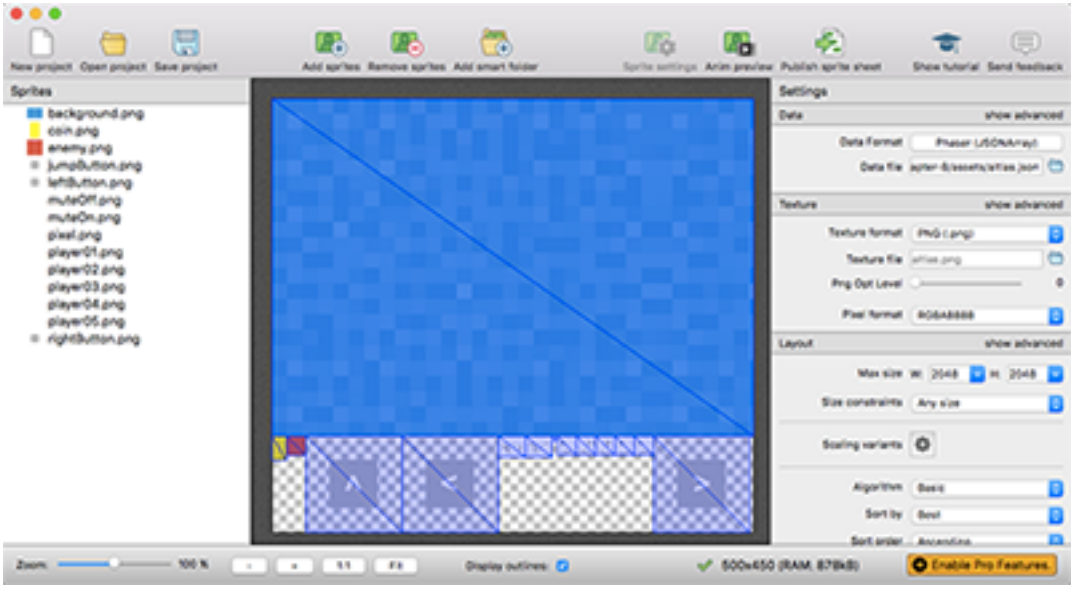
Launch TexturePacker, click on the "chose data format" button in the top right, and select Phaser in the list (the 'JSONArray' one).



Then just below the “chose data format” button you need to select a directory and a name for the file. Name it atlas.json and select the assets/ folder of the game.

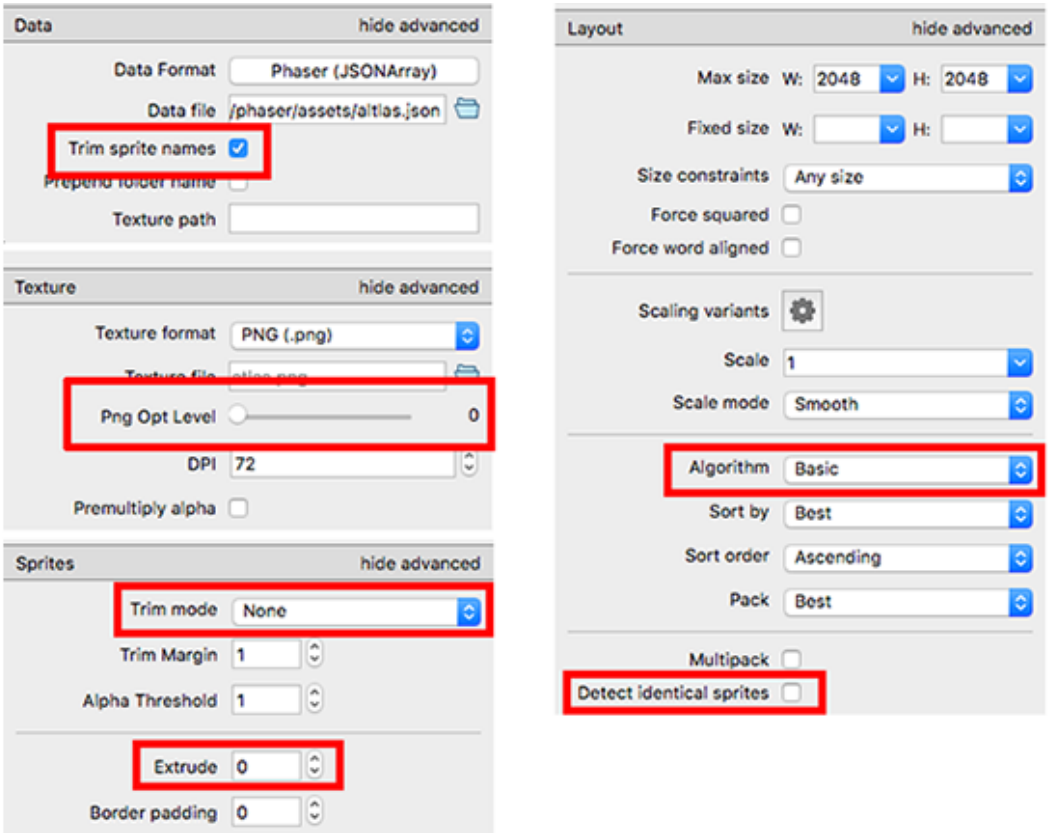


Now drag and drop everything that’s inside the atlas/ folder you downloaded earlier into TexturePacker.



You can see in the middle the single image containing all out sprites.

Next, change all of these settings. Most of these need to be edited because the default values are not available in the free version. You will have to click on “show advanced” to find some of these options.



And click on the “publish sprite sheet” button in the top of the window. You will get 2 new files in your assets/ folder: atlas.json and atlas.png.

Update Assets Folder

We can finally tidy our assets/ folder. Remove all the old png files except tileset.png and progressBar.png (as explained earlier).

9.2 - Use Atlas

Now that we have our atlas, it's time to use it in our game. This will require some changes in our code.

Edit load.js

Edit the load.js file to load our atlas instead of all our sprites:

```
    // Delete all of this
    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');

    // And replace it by this:
    game.load.atlasJSONArray('atlas', 'assets/atlas.png',
        'assets/atlas.json');
```

Now we have to edit all our `game.add.image` to use the atlas, as described below.

Edit menu.js

In the `create` function of the menu.js file, we need to do all these changes to use our new atlas:

```
    // Add the 'atlas' parameter to the background image
    game.add.image(0, 0, 'atlas', 'background');

    // Replace 'muteButton' by 'atlas' on this line
    this.muteButton = game.add.button(20, 20, 'atlas', this.toggleSound,
        this);

    // Edit this line to use 'frameName' instead of 'frame'
    this.muteButton.frameName = game.sound.mute ? 'muteOn' : 'muteOff';
```

We also need to edit the same line from the `toggleSound` function:

```
    this.muteButton.frameName = game.sound.mute ? 'muteOn' : 'muteOff';
```

Edit play.js

Change the player creation like this:

```
    // Add a new parameter 'atlas' and
    // Use the new name for the sprite: 'player01' instead of 'player'
    this.player = game.add.sprite(game.width/2, game.height/2, 'atlas',
        'player01');

    // Use the new frame names for the player's animations
    this.player.animations.add('right', ['player02', 'player03'], 8);
    this.player.animations.add('left', ['player04', 'player05'], 8);

    // Change 'this.player.frame = 0' by this line in 'movePlayer'
    this.player.frameName = 'player01';
```

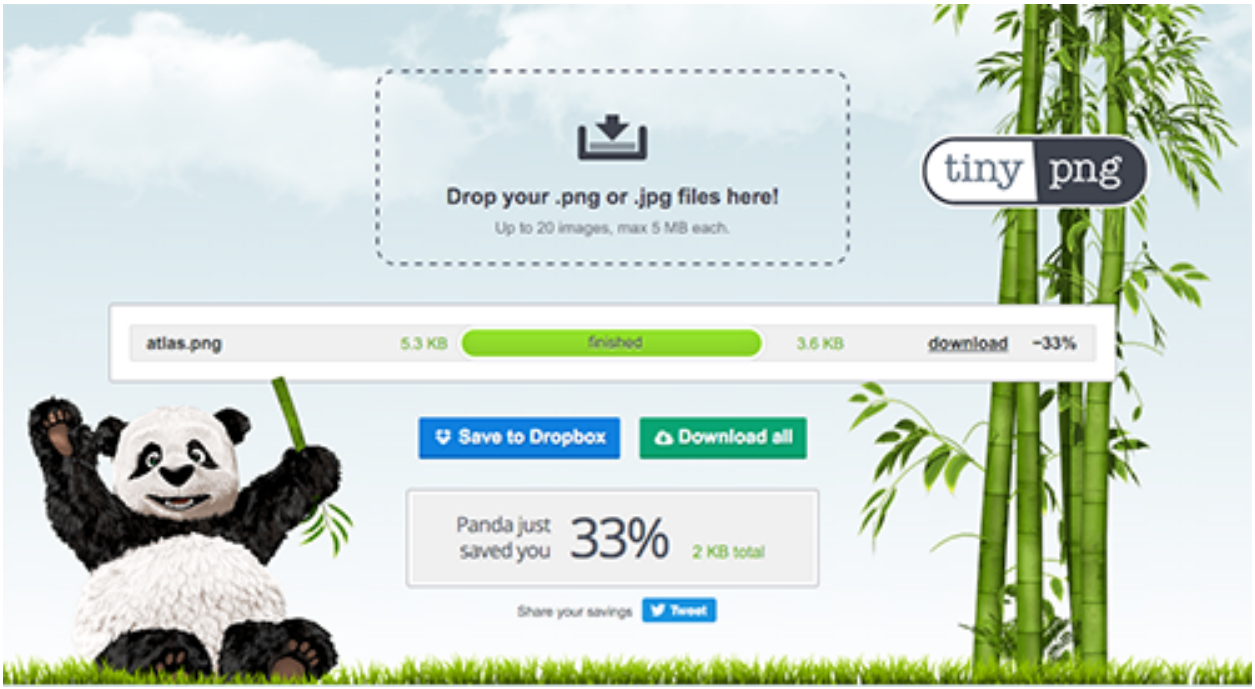
And add the new ‘atlas’ parameter to all the other sprites we are using:

```
// In the 'create' function
this.enemies.createMultiple(10, 'atlas', 'enemy');
this.coin = game.add.sprite(60, 140, 'atlas', 'coin');
this.emitter.makeParticles('atlas', 'pixel');

// In the 'addMobileInputs' function
this.jumpButton = game.add.sprite(350, 247, 'atlas', 'jumpButton');
this.leftButton = game.add.sprite(50, 247, 'atlas', 'leftButton');
this.rightButton = game.add.sprite(130, 247, 'atlas', 'rightButton');
```

Optimize the png

The last step is to optimize the big png of our atlas. You can do so with **tinypng.com** for example.

Go on the website, drag and drop the image there, and download the optimized version.



As you can see we reduced the size by 33%, not bad.

9.3 - Create Audio Sprite

What we did for the sprites in the previous parts can also be done for the audio files:

- Combine all sounds into one single file to reduce HTTP requests.
- And play the file at the right time to hear the sound effect we need.

The result is called an audio sprite, and we will see how to create one.

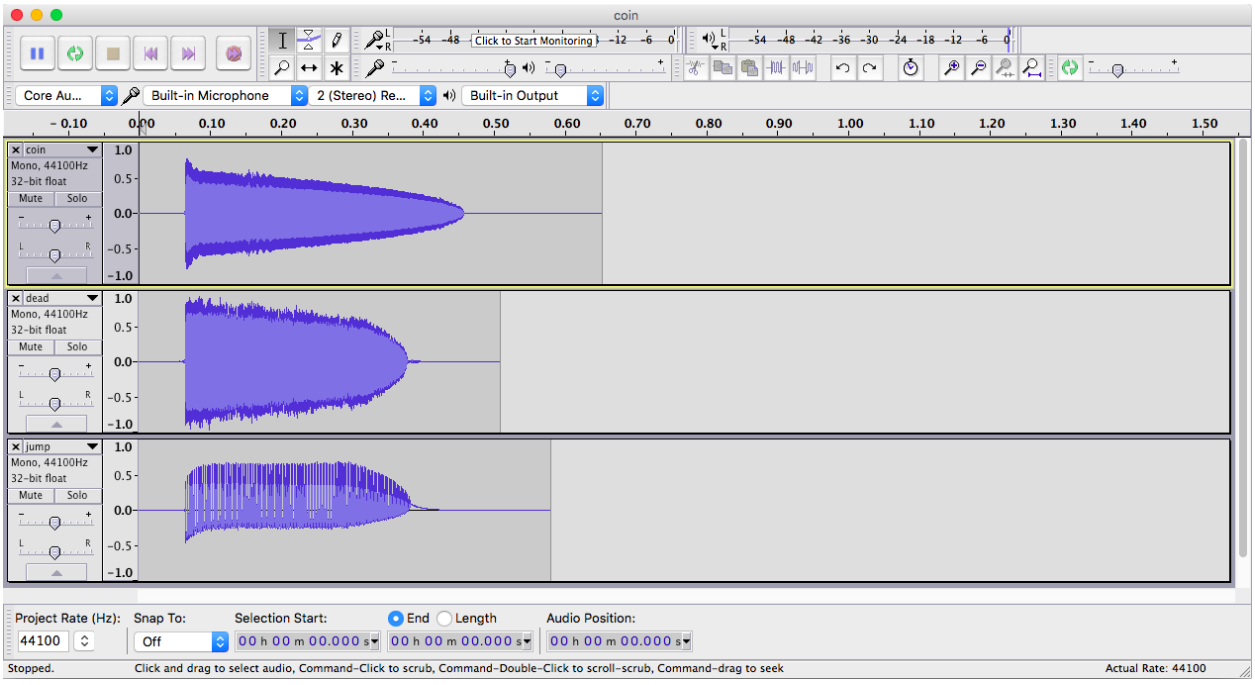
Set Up

First we need to combine all the audio files together. We can do that in 2 main ways:

- Manually with a software like **Audacity**.
- Automatically with a command line tool such as **Audiosprite**.

Since we only have 3 sound effects we are going to do that manually. So download **Audacity**, install it on your computer, and launch it.

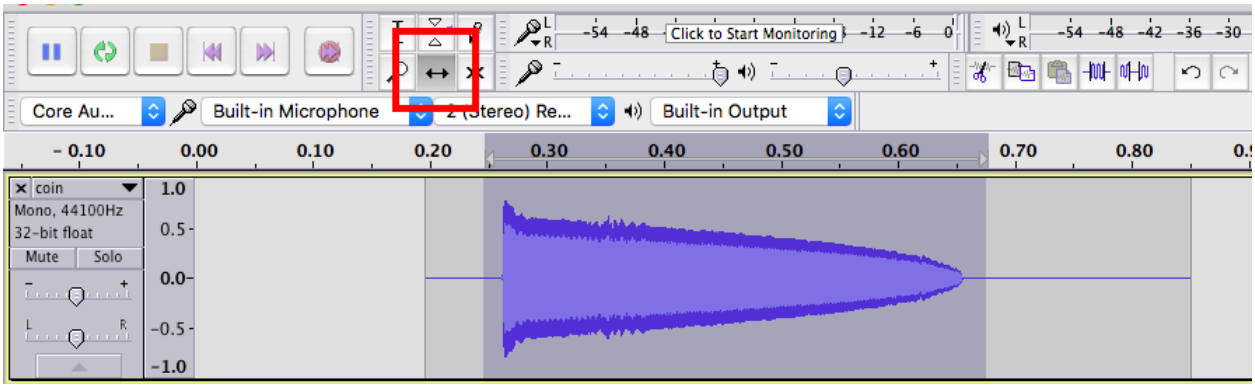
Then take all the ogg files of our project and drag them on Audacity. You should see this:



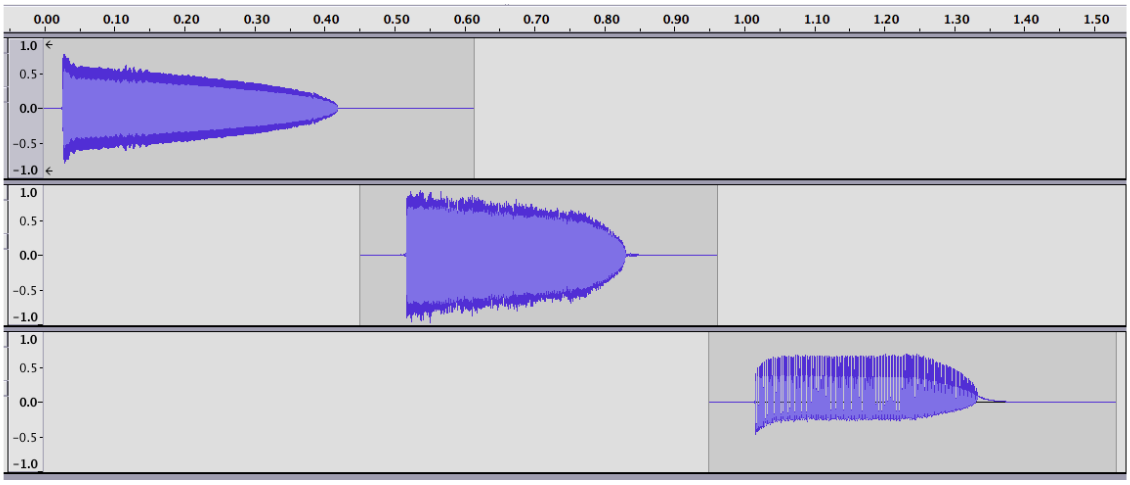
Move the Sounds

Right now all the sounds are on top of each other, so if you press the play button you will hear everything at the same time. That's why we need to move the sound effects around.

You can move a sound easily: pick the move tool and drag the wave to its new position.



Use this technique to position them like this:



Try to start each sound just after a round number. For example: 0.05 seconds, 0.55 seconds, and 1.05 seconds. That will make our life easier when we have to play the sounds in Phaser later.

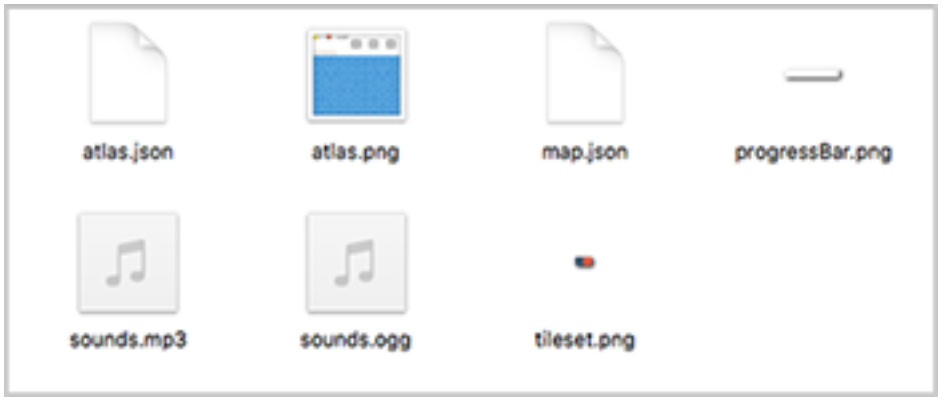
Export the Sound

Once you are done, export the sound by doing "top menu > file > export audio" in both ogg and mp3 formats.

For the mp3 export Audacity will ask you to download a third party encoder called LAME. You can either follow the instructions to do so, or simply use an online tool to convert the new ogg into an mp3.

Update Assets Folder

Put the new exported files into the assets/ folder and rename them sounds.mp3 and sounds.ogg. You can then delete all the old audio files from the folder.



That's a lot cleaner that before.

9.4 - Use Audio Sprite

Now we need to make some simple changes to our code, in order to use our new sound effects.

Edit load.js

The load state needs some straightforward edits to preload the new file:

```
    // Delete this
    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']);

    // And add this instead
    game.load.audio('sounds', ['assets/sounds.ogg', 'assets/sounds.mp3']);
```

Edit play.js

In the `create` function of the play state, things are more interesting.

First delete these lines because we no longer need them:

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

And add this code instead, to extract the 3 sound effects from our single file:

```
    // Add our new sound
    this.sounds = game.add.audio('sounds');

    // Tell Phaser that it contains multiple sounds
    this.sounds.allowMultiple = true;

    // Split the audio. The last 2 paramters are:
    // The start position and the duration of the sound
    this.sounds.addMarker('coin', 0, 0.45);
    this.sounds.addMarker('dead', 0.5, 0.45);
    this.sounds.addMarker('jump', 1, 0.45);
```

Finally we need to change the way we actually play the sounds like this:

```
    // Replace each of these lines
    this.jumpSound.play(); // In the 'jumpPlayer' function
    this.coinSound.play(); // In the 'takeCoin' function
    this.deadSound.play(); // In the 'playerDie' function

    // By the corresponding new line
    this.sounds.play('jump');
    this.sounds.play('coin');
    this.sounds.play('dead');
```

9.5 - Concat and Minify

The last optimization technique we will use is called “concat and minify”. It does 2 things:

- Concat: combine our 5 Javascript files into one.
- Minify: reduce the size of our code by removing unnecessary spaces and newlines.

You need to be comfortable using the command line to follow the instructions below. If that’s not the case, you can skip to the end of this part to learn how to do things manually.

Set Up

We will use Uglify to concat and minify our Javascipt files, and it requires Node.js to work. So go to the **Node.js website** to download it, then follow the steps to install it.

Once you have Node.js you can get Uglify by typing this:

```
npm -g install uglify-js
```

If it doesn’t work try adding `sudo` at the beginning, and then type your password.

Concat and Minify

Now we can use Uglify like this:

```
// Concat and minify one.js and two.js into three.js
uglifyjs one.js two.js -o three.js
```

For our game, do this once you are in the project’s directory:

```
uglifyjs phaser.min.js js/boot.js js/load.js
    js/menu.js js/play.js js/game.js -o min.js
```

Note that game.js should be the last since it is using variables defined in the previous files.

Update index.html

Instead of loading all our Javascript files in the index.html, we should just load the new min.js:

```
// Replace all of this
<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>

// By this
<script type="text/javascript" src="min.js"></script>
```

Now our whole game is in a single Javascrip file.

Manual

Here are some basic instructions on how to concat and minify manually:

- Go to **jscompress.com** and click on the “upload Javascript files” tab at the top.
- Upload all of these: phaser.min.js, boot.js, load.js, menu.js, play.js, and game.js (make sure game.js is the last one included).
- Click on the submit button, wait a few seconds, and download the output.min.js file.
- Put the new file into the game’s directory and rename it min.js.
- Edit the index.html as explained just above.

And you are done.

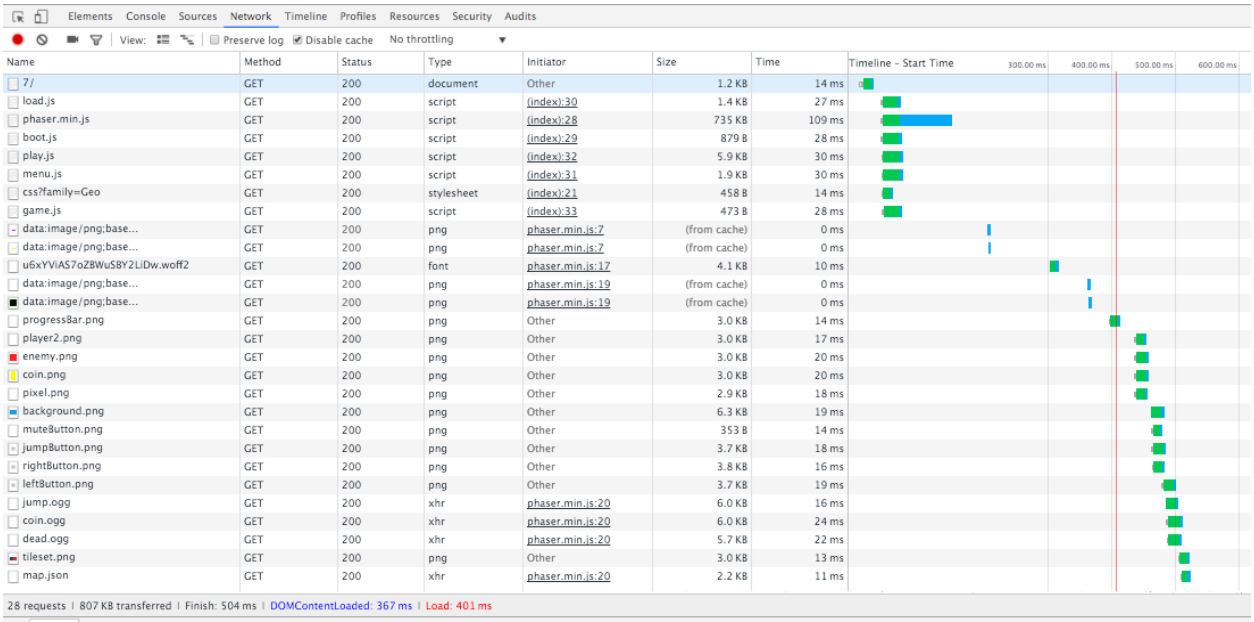
9.6 - Measure Improvements

In this chapter we used a lot of optimization techniques, it's now time see how fast our game has become.

It's really easy to precisely measure the speed of a webpage: open Google Chrome with the Developer Tools (right-click anywhere on the page and select "inspect element"), click on the 'network' tab, and reload the page.

Before

Here's what happens when I load the game from the previous chapter with no optimization:



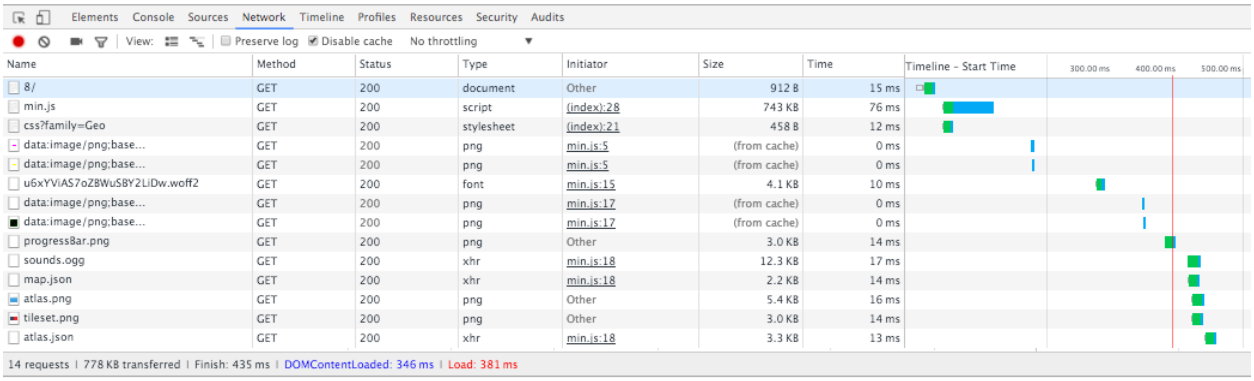
Each line is a file that is loaded by the browser with an HTTP request. The green bars on the right represent the time it took to load each file.

In the bottom left we can see 2 important numbers: there's 28 HTTP requests and the game takes 504ms to load.

That's pretty fast, but can we do better?

After

This is the result of the speed test with all the optimizations:



This time there's only 14 HTTP requests in 435ms.

Conclusion

Our optimizations made the game about 15% faster to load.

Since our game is quite small there isn't a big difference. But for a more complex project with more assets it can have a huge impact.

9.7 - Code Refactoring

Our play state is about 200 lines long. That's okay, but any bigger than that and it would become messy. In this part we will see how to refactor our code for better readability.

The code you will see below is based on the 3rd chapter of the book. So only one state, no animations, not mobile friendly, etc. This way we can focus more on the refactoring and less on the Phaser code itself.

The Idea

Instead of having one state that contains everything, we will split our code into a few objects: level, player, enemies, etc. And the state will only have to preload/create/update them.

Each object will have its code in a specific file. So if you want to change something related to the player, you just have to edit player.js. This is going to be a lot easier than directly modifying our current big state with all its functions.

Player Object

Here's one way to create an object in Javascript:

```
function objectName() {

    this.functionName1 = function() {
        // Do things
    };

    this.functionName2 = function() {
        // Do other things
    };
};
```

You can see that the syntax is a little different from the states we used so far:

- We do `function objectName() { }` instead of `var stateName { }`.
- And for the inner functions it's `this.functionName = function() { }` instead of `functionName: function() { }`.

Here's the code for the player in a Javascript object. It should look familiar to you since it's basically copy-pasting what we had in our state.

```
function player() {

    this.preload = function() {
        game.load.image('player', 'assets/player.png');
    };

    this.create = function() {
        this.sprite = game.add.sprite(game.width/2, game.height/2,
            'player');
        this.sprite.anchor.setTo(0.5, 0.5);
        game.physics.arcade.enable(this.sprite);
        this.sprite.body.gravity.y = 500;

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

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

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

        if (!this.sprite.inWorld) {
            this.die();
        }
```

```
        };

        this.die = function() {
            game.state.start('main');
        };
    };
```

There are a few important things to notice here:

- I didn't name the variable `this.player` but `this.sprite`. This way we will be able to do `this.player.sprite` to access the Phaser sprite from the state, as we will see later.
- I added the `cursor` variable in the player object. It makes sense since only the player is using the `cursor`.

And you should create similar code for: enemies, coins, score, and the level. Each in their own Javascript file. Don't forget to include them in the index.html page.

New State

Let's see what should be inside our new state after all the objects are created.

```
    var mainState = {
         // Add all the following code here
    };
```

First, the `preload` function:

```
    preload: function() {
        // Create all the objects with 'new objectName()'
        this.player = new player();
        this.enemies = new enemies();
        this.coin = new coin();
        this.level = new level();
        this.score = new score();

        // Call their `preload` function
```

```
        this.player.preload();
        this.enemies.preload();
        this.coin.preload();
        this.level.preload();
    },
```

I didn't do `this.score.preload` because the score object has nothing to load.

Then there's the `create` function:

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

        // Initialize all our objects with their `create` function
        this.player.create();
        this.enemies.create();
        this.coin.create();
        this.level.create();
        this.score.create();
    },
```

Technically we could have an object to handle the settings, but I think it makes sense to keep that in the main state.

Next, the `update` function:

```
    update: function() {
        // Handle all collisions
        game.physics.arcade.collide(this.player.sprite, this.level.group);
        game.physics.arcade.collide(this.enemies.group, this.level.group);
        game.physics.arcade.overlap(this.player.sprite,
            this.coin.sprite, this.takeCoin, null, this);
        game.physics.arcade.overlap(this.player.sprite,
            this.enemies.group, this.player.die, null, this);
```

```
        // Call the `update` of our objects (only the player has one)
        this.player.update();
    },
```

The important thing here is that the collisions work with Phaser variables (either a sprite or a group). So we have to use:

- `this.player.sprite` instead of `this.player`.
- `this.enemies.group` instead of `this.enemies`.
- And so on.

Finally, we need the `takeCoin` function because it does things on both the score and the coin:

```
    takeCoin: function() {
        // Increase the score and update the label
        this.score.increase();

        // Change the coin's position.
        this.coin.updatePosition();
    },
```

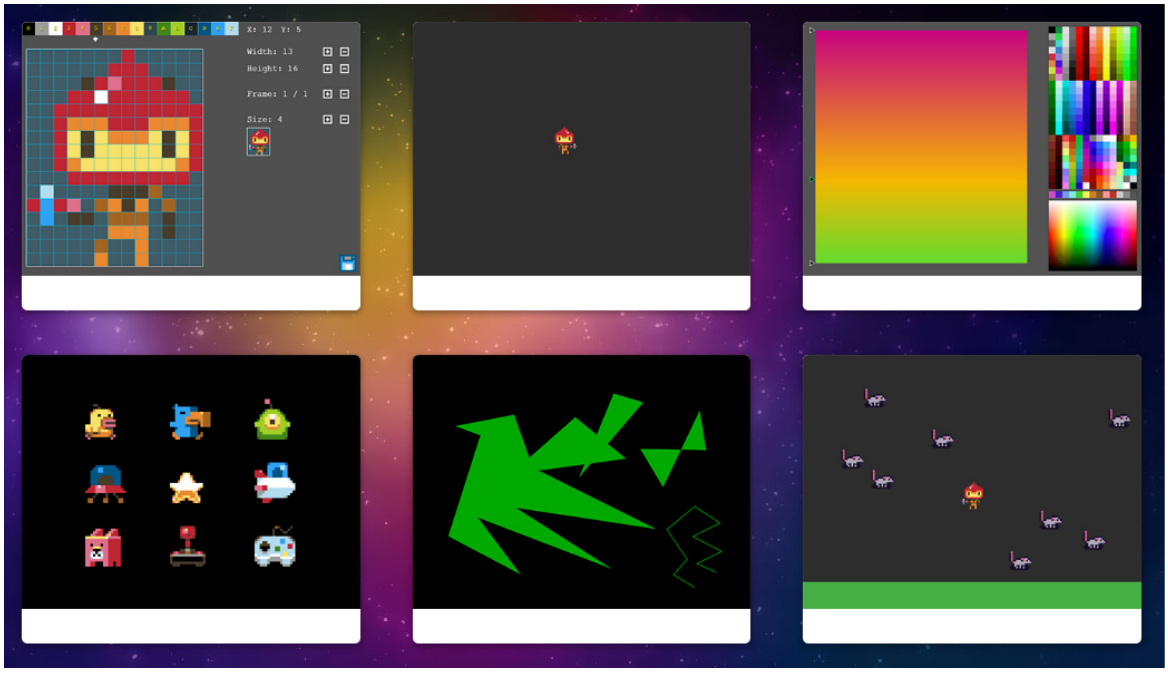
And that's it. All the other functions (`addEnemy`, `createLevel`, etc.) are directly inside the objects.

You can see that the state is now a lot shorter and cleaner.

10 - More About Phaser

We finished building our game, but we didn't cover everything that is possible to do with the Phaser framework.

That's why in this chapter we will see some important features that you might need for your own games.



10.1 - New Functions

Let's start this chapter with some useful Phaser functions.

Collisions

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. 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 from the sprite anchor position.
  - `offsetY`: y offset from the sprite anchor position.

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

We can have a world bigger than the game size with `game.world.resize`.

- `game.world.resize(width, height)`
  - `width`: new width of the world.
  - `height`: new height of the world.

Let's use an example to better understand how this works. Here's a 200px by 100px game:

If we do `game.world.resize(250, 150)`, we would have this:

The size of the game view is the same, but world is now bigger.

And if needed, you can access the size of the game with these variables:

```
// Get the size of the game view
game.width;
game.height;

// Get the size of the whole world
game.world.width;
game.world.height;
```

Camera

If the world is bigger than the game view, you will need a way to change the area displayed on the screen. You can do so by either:

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

But once you start moving the camera everything will move, including the score label or some buttons. To prevent that you can use this:

```
// Fix the object to the camera
```

```
    object.fixedToCamera = true;
```

The `object` can be a sprite, a label, or a group.

Fullscreen

If you want your game size to take the whole page, you just need to change the way Phaser is created:

```
    // Use % instead of pixels and don't specify a div at the end
    var game = new Phaser.Game("100%", "100%", Phaser.AUTO, '');

    // Which is similar to this
    var game = new Phaser.Game("100%", "100%");
```

If the browser window is resized, add this line to make sure the game always takes 100% of the screen:

```
    game.scale.scaleMode = Phaser.ScaleManager.RESIZE;
```

And this function will be automatically called each time the game is resized:

```
    resize: function() {
        // Add some code here if you need
    }
```

Randomness

Generating random numbers in a game can be very important. Here are some nice ways to do that with Phaser (some of which we already covered).

```
// Create a random int between min and max (4, 8, 15, 16, ...)
game.rnd.integerInRange(min, max);

// Create a random float between min and max (23.01, 42.69, ...)
game.rnd.realInRange(min, max);

// Pick a random element from the array
game.rnd.pick([a, b, c]);
```

Set To

We used a few times the `setTo` function to change the anchor's position of sprites and labels. You will probably use it a lot in your own games, so here's a little trick: the 2 lines below do the same thing.

```
// Set both x and y anchor to 0.5
object.anchor.setTo(0.5, 0.5);
object.anchor.setTo(0.5);
```

When you specify only the `x` parameter, the `y` will take the same value. This is a handy shortcut to center an object.

Frames Per Second

During development it can be useful to know how well your game is performing by checking its FPS (Frames Per Second). There's an easy way to do that with Phaser.

First, add this in the `create` function:

```
// Tell Phaser to compute the FPS
game.time.advancedTiming = true;

// Create a label to display the FPS
```

```
    this.fpsLabel = game.add.text(10, 10, '0');
```

Then put this in the `update` function:

```
    // Update the label with the current FPS
    this.fpsLabel.text = game.time.fps;
```

If things run smoothly you should get 60 FPS.

Foreach

The last thing you need to know is the `forEachAlive` function that iterates 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 this new function
    checkPosition: function(enemy) {
        // Kill the enemy if it's below the player
        if (enemy.y > this.player.y) {
            enemy.kill();
        }
    },
```

10.2 - Debugging

A big part of building games is unfortunately debugging. Here are some ideas to make you more efficient.

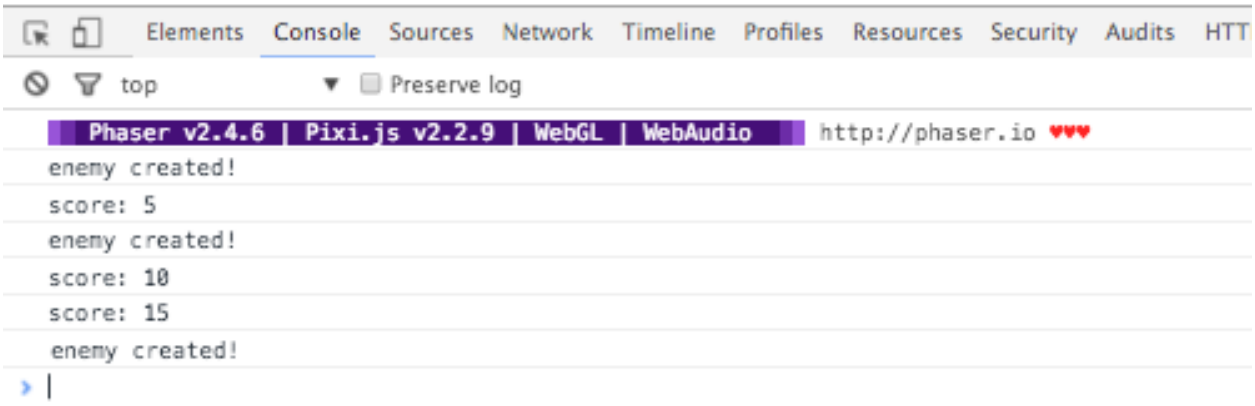
Javascript

The simplest way to debug some Javascript code is to use the `console.log` function. It lets you display information in the console to see what's going on.

Some examples:

- Add `console.log('enemy created!')` in the `addEnemy` function to check when new enemies appear.
- Add `console.log('score: ' + game.global.score)` in the `takeCoin` function to track the score.

Then you just need to look at the console in your browser to see the messages.

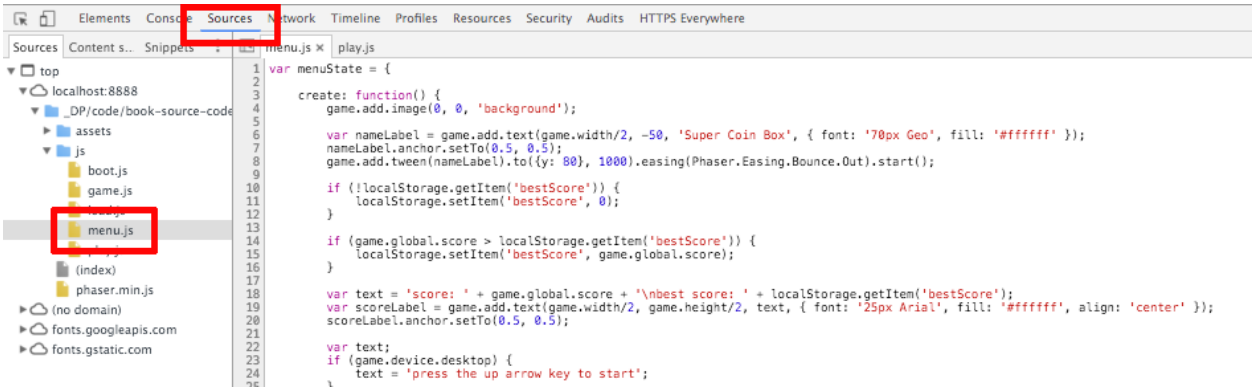


Don't forget to remove the `console.log` functions once finished, because they can slow down your game.

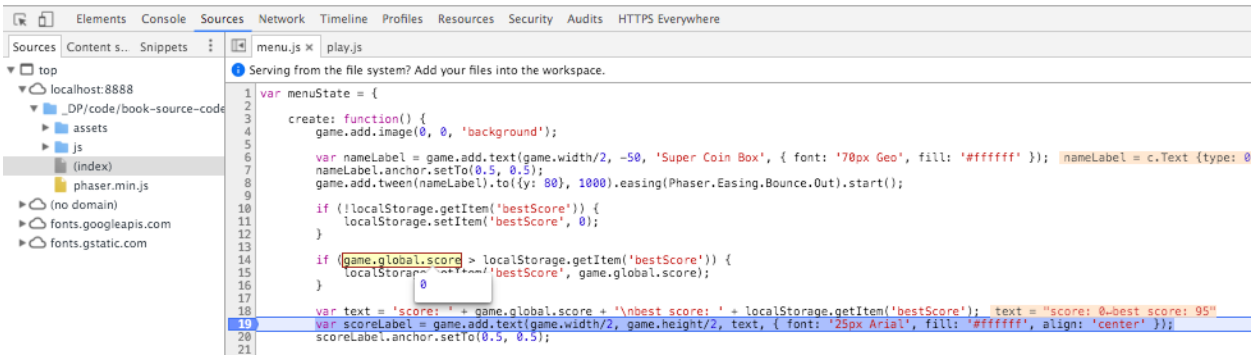
Developer Tools

The Developer Tools are really powerful for debugging. A feature that I like is called 'breakpoint', it lets you pause a script to check what's happening at that particular moment. Here's how it works.

Load your game in Google Chrome, open the Developer Tools, click on the 'sources' tab, and find your Javascript file.



Click on a line number to add a breakpoint, reload the page, and see the game stopping at your breakpoint.



Now you can check exactly what is happening in your code at that specific point in time. For example you can hover variable names to see their values.

Phaser

So far we only talked about generic things to help you debug your code. But Phaser is also full of debugging functions.

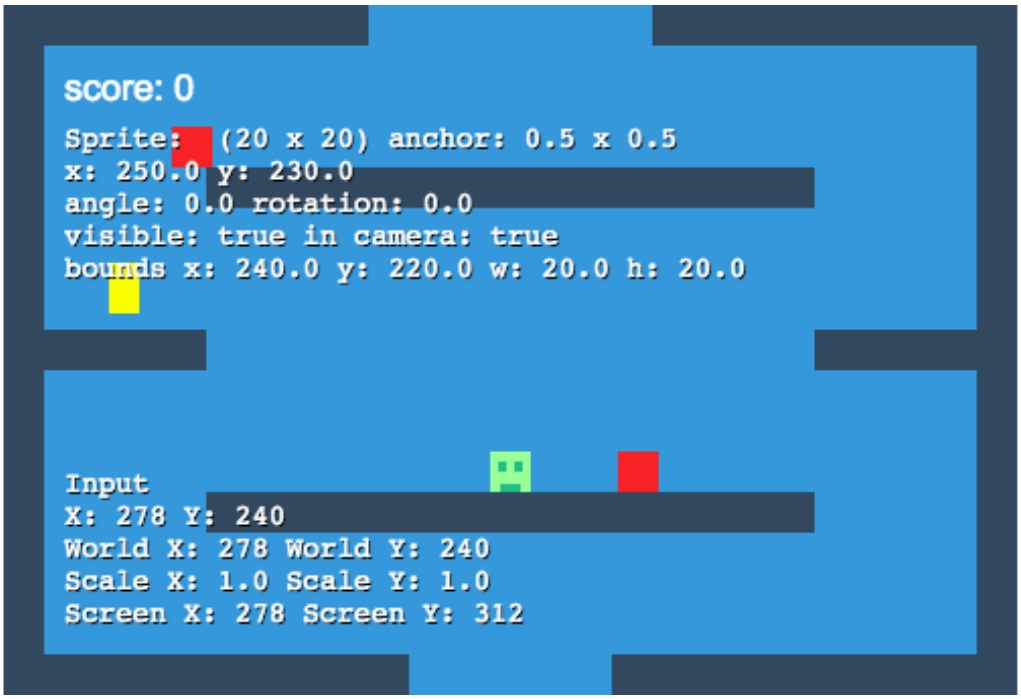
Here are the 3 most interesting ones:

- Sprite: `game.debug.spriteInfo(sprite, x, y)` displays information about the sprite.
- Physics: `game.debug.body(sprite)` shows the physics body of the sprite as a green overlay.
- Input: `game.debug.inputInfo(x, y)` displays information about the input.

For example, if you put this in the `update` function of the play state:

```
    game.debug.spriteInfo(this.player, 30, 70);
    game.debug.body(this.player);
    game.debug.inputInfo(30, 240);
```

You would see this:



10.3 - Code Sprites

It's often time consuming to design sprites in an image editor like Photoshop or Gimp. Well, it turns out Phaser has a great feature to help us be faster.

The Code

Remember the sprite we used for the player in our game?

It was designed in Photoshop, but we could do the same thing with code:

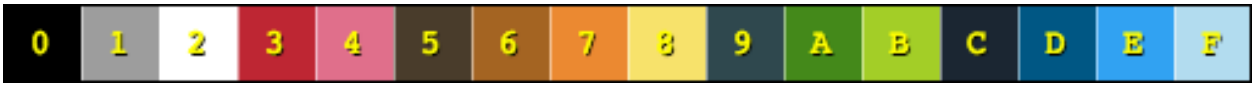
```
// Design the player
var player = [
    '22222',
    '2.2.2',
    '22222',
    '22222',
    '2...2',
];

// Create the texture
game.create.texture('player', player, 5, 5);

// Create the player
this.player = game.add.sprite(game.width/2, game.height/2, 'player');
```

Here's how this works:

- First, design the sprite in a 2D array where each character represents a color. Here '.' is transparent and '2' is white.

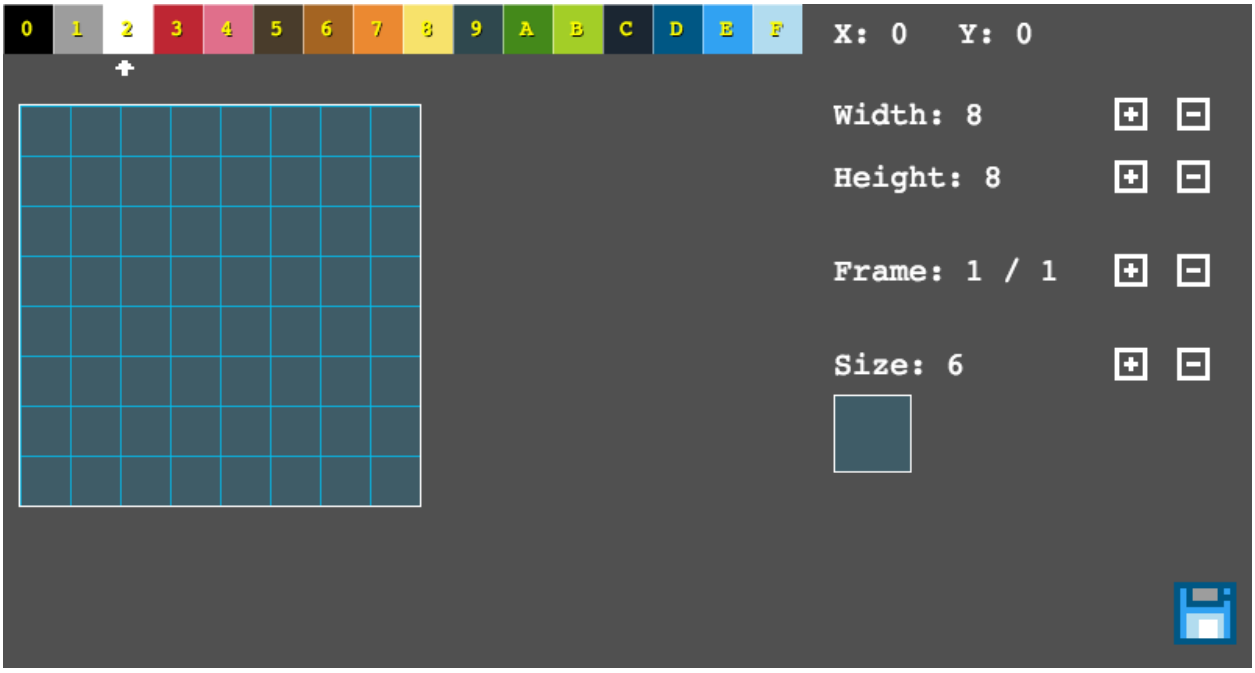
- Then use `game.create.texture` to transform the 2D array into a texture. The last 2 parameters are: the width and height of each character in pixels.
- And create the player with the new texture, like we would normally do.

Here are all the colors available with their corresponding number:



The Editor

Since it's not easy to draw something with numbers, the creator of Phaser built a simple tool to generate the array for us. You can access it **here**, and it looks like this:



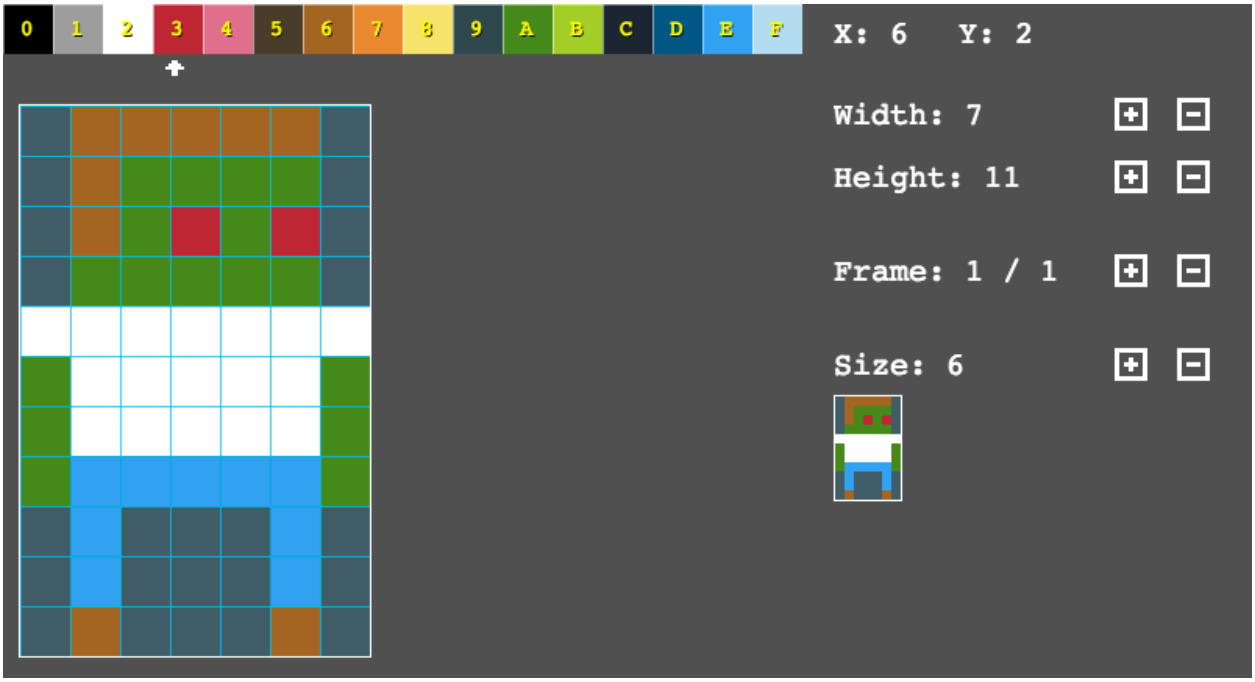
How to use it:

- Start by defining the width and height of your sprite on the right.
- Paint the sprite by using the colors at the top.
- Once finished, click on the save button in the bottom right corner.
- And look at the console to grab the generated code.

This way it's really easy and fast to create pixelated sprites for your games.

Example

Here's an example of what you can do with the sprite editor.



And this is the array:

```
var zombie = [
    '.66666.',
    '.6AAAA.',
    '.6A3A3.',
    '.AAAAA.',
    '2222222',
    'A22222A',
    'A22222A',
    'AEEEEEA',
    '.E...E.',
    '.E...E.',
    '.6...6.'
];
```

10.4 - Extend Phaser

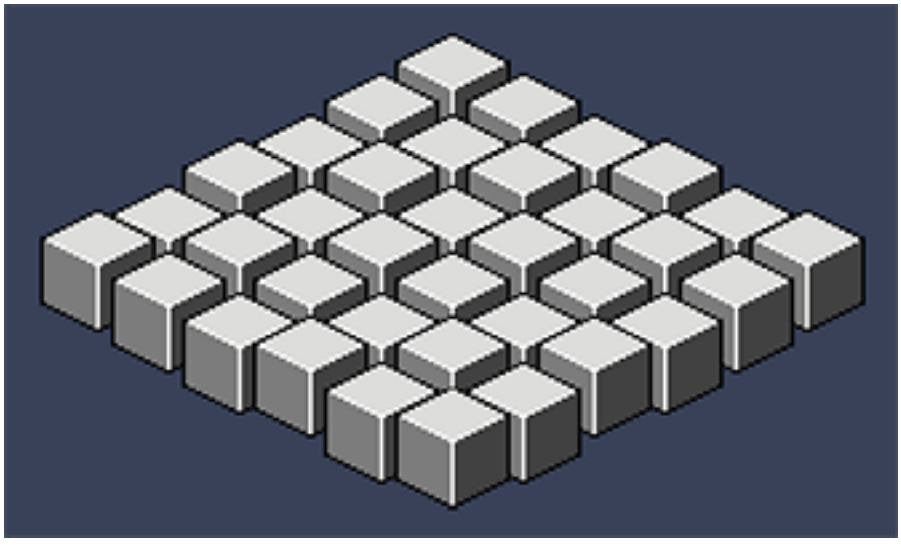
I hope that by now you realized how great Phaser is. And it can become even better when you combine it with other things like plugins or APIs. We will quickly cover the most important ones below.

Plugins

There are now a lot of Phaser plugins available to make your life easier.

There is no centralized way to find them, so Google is your best bet. But here are a few to get you started:

- **Isometric**: create isometric games (see image below).
- **Phaser-Tiled**: optimize complex maps made with Tiled.
- **Box2D**: paid plugin to add a powerful physics engine to your game.
- **Virtual joystick**: add a virtual joystick on mobile games.



Multiplayer

For browser games it makes sense to have some multiplayer since we know that the users are already online.

This is possible to do with a networking library. The best ones are probably **ws** and **socket.io**.

Keep in mind that creating a multiplayer game is a lot of work because you'll need to code both the client and the server.

Native Desktop Games

It's actually possible to convert a Phaser game into a native app for Windows, MacOS, and Linux.

Sounds interesting? Then check out **Electron**. You will have to spend some time reading the documentation to make things work.

Once done, you will be able to submit your HTML5 games to online stores like Steam.

Analytics

It's always a good idea to track what players are doing in your games. This way you know how fast they finish it, where they are getting stuck, how frequently they play, etc.

The easiest way to do that is to use **Google Analytics**, and send custom events when specific actions are done.

It's free and simple to set up.

Leaderboards

Adding leaderboards can make players more engaged in a game, they will play more to try to beat each other's score. But this can be a complex task if you want to build it by yourself.

Your best bet is to use an existing API. The most popular ones are the **Facebook API** and the **Google API**.

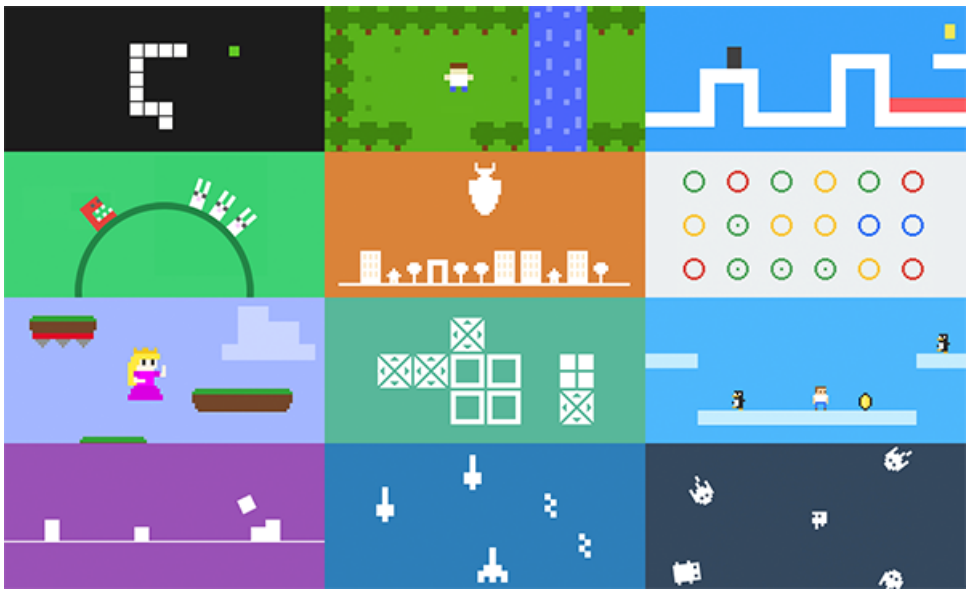
11 - Next Steps

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:

- Chapter 3: sprites, labels, groups, recycling, Arcade physics.
- Chapter 4: state management.
- Chapter 5: sounds, animations, tweens, particles, camera.
- Chapter 6: local storage, buttons, custom fonts, keyboard.
- Chapter 7: tilemaps with Tiled.
- Chapter 8: scaling, touch inputs, touch buttons, device orientation, cocoonJS.
- Chapter 9: atlas, audio sprite, concat, minify, refactoring.
- Chapter 10: debugging, code sprites, plugins.

This last chapter will give you some ideas and tips on what to do next.



11.1 - 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.

Customization

Customizing the game is easy. Change all the sprites and sounds, tweak all the values (gravity, velocity, tweens), and you will get a brand new game.

More Tweens

Tweens are a great way to make a game feel better, and you can add more of them. For example: make the coin rotate indefinitely, add a special animation when the user beat his best score, make the score label grow each time it's updated, etc.

Multiple Lives

Instead of ending the game as soon as the player hits an enemy, we could just lose a life and keep playing.

```
// In the 'create' function
define a life variable
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 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 -200
    set a startJump variable to game.time.now

else if (a jump started less than 200ms ago)
    keep jumping with a velocity of -200
```

Enemy Types

The current red enemies have the same size and move at the same speed. That's a bit boring. To change that you could create 2 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
```

11.2 - Make New 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 can be a little scary (and also amazing).

I wrote below 3 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 build 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.

Start making something really simple like Pong, Breakout or Space Invaders. And when you become comfortable making these types of games, then you can start more ambitious ones.

Keep Iterating

Instead of trying to make the perfect game from the start, try to build it 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.

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.

11.3 - Conclusion

This 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.
- Follow me on Twitter **@thomaspalef**.
- Send me an email: **thomas@lessmilk.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:

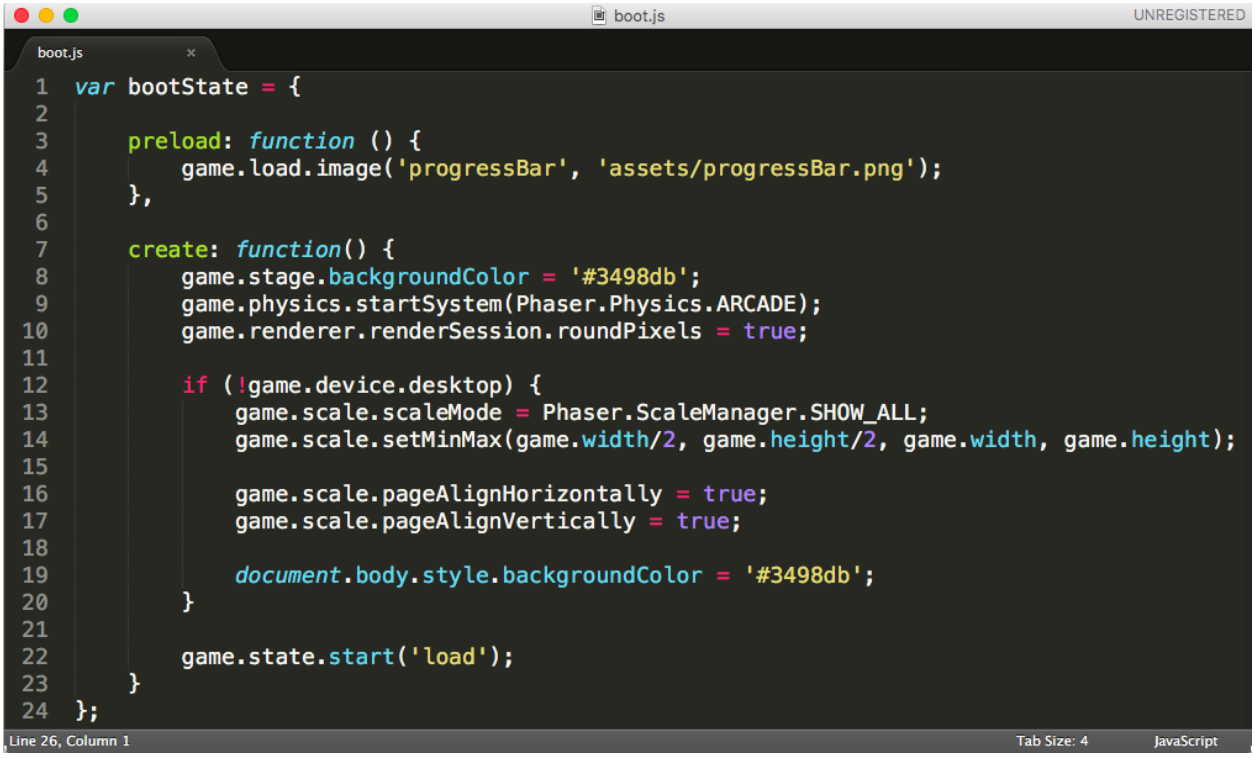
- Richard D. who showed me some best practice to follow for the game's code.
- Theodore L. who beta tested the book and gave me some great insights.
- Maryla U. who found a lot of ways to improve the book.
- Ben B. who made sure that the book is accessible to everyone.

Thanks for reading,

Thomas Palef

12 - Full Source Code

Let's review the full code of our game. This is the code before doing all the optimizations in chapter 9.



12.1 - The Code

Index

```
<!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;
                height: 0;
            }
        </style>

        <meta name="viewport" content="initial-scale=1
            user-scalable=no" />

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

</html>
```

Boot

```
var bootState = {

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

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

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

        game.scale.setMinMax(game.width/2, game.height/2,
            game.width*2, game.height*2);

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

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

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

};
```

Load

```
var loadState = {

preload: function () {
    var loadingLabel = game.add.text(game.width/2, 150, 'loading...',
        { font: '30px Arial', fill: '#ffffff' });
    loadingLabel.anchor.setTo(0.5, 0.5);

    var progressBar = game.add.sprite(game.width/2, 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('background', 'assets/background.png');
    game.load.image('pixel', 'assets/pixel.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.image('tileset', 'assets/tileset.png');
    game.load.tilemap('map', 'assets/map.json', null,
        Phaser.Tilemap.TILED_JSON);

    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']);
},

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

};
```

Menu

```
var menuState = {

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

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

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

    var nameLabel = game.add.text(game.width/2, -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();

    var text = 'score: ' + game.global.score + '\nbest score: ' +
        localStorage.getItem( 'bestScore');
    var scoreLabel = game.add.text(game.width/2, game.height/2, text,
        { font: '25px Arial', fill: '#ffffff', align: 'center' });
    scoreLabel.anchor.setTo(0.5, 0.5);

    var text;
    if (game.device.desktop) {
        text = 'press the up arrow key to start';
    }
    else {
        text = 'touch the screen to start';
    }
    var startLabel = game.add.text(game.width/2, game.height-80,
        text, { font: '25px Arial', A> fill: '#ffffff' });
    startLabel.anchor.setTo(0.5, 0.5);
    game.add.tween(startLabel).to({angle: -2}, 500)
        .to({angle: 2}, 1000).to({angle: 0}, 500).loop().start();
```

```
        var upKey = game.input.keyboard.addKey(Phaser.Keyboard.UP);
        upKey.onDown.add(this.start, this);
        if (!game.device.desktop) {
            game.input.onDown.add(this.start, this);
        }

        this.muteButton = game.add.button(20, 20, 'mute',
            this.toggleSound, this);
        this.muteButton.frame = game.sound.mute ? 1 : 0;
    },

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

    start: function() {
        if (!game.device.desktop && game.input.y < 50
            && game.input.x < 60) {
            return;
        }

        game.state.start('play');
    },

    };
```

Play

```
    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)
    };

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

    this.createWorld();
    if (!game.device.desktop) {
        this.addMobileInputs();
    }

    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' });
    game.global.score = 0;

    this.enemies = game.add.group();
    this.enemies.enableBody = true;
    this.enemies.createMultiple(10, 'enemy');
    this.nextEnemy = 0;

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

    this.emitter = game.add.emitter(0, 0, 15);
    this.emitter.makeParticles('pixel');
    this.emitter.setYSpeed(-150, 150);
    this.emitter.setXSpeed(-150, 150);
    this.emitter.setScale(2, 0, 2, 0, 800);
    this.emitter.gravity = 0;
```

```
        if (!game.device.desktop) {
            this.rotateLabel = game.add.text(game.width/2, game.height/2,
                '', { font: '30px Arial' , fill: '#fff',
                backgroundColor: '#000' });
            this.rotateLabel.anchor.setTo(0.5, 0.5);

            game.scale.onOrientationChange.add(this.orientationChange,
                this);

            this.orientationChange();
        }
    },

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

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

        this.movePlayer();

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

        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 (game.input.totalActivePointers == 0) {
            this.moveLeft = false;
            this.moveRight = false;
        }

        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.player.body.velocity.y = -320;
            this.jumpSound.play();
        }
    },

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

        this.updateCoinPosition();

        this.coinSound.play();
```

```
        this.coin.scale.setTo(0, 0);
        game.add.tween(this.coin.scale).to({x: 1, y: 1}, 300).start();
        game.add.tween(this.player.scale).to({x: 1.3, y: 1.3}, 100)
            .yoyo(true).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 = game.rnd.pick(coinPosition);
        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.width/2, 0);
        enemy.body.gravity.y = 500;
        enemy.body.velocity.x = 100 * game.rnd.pick([-1, 1]);
        enemy.body.bounce.x = 1;
        enemy.checkWorldBounds = true;
        enemy.outOfBoundsKill = true;
    },

    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);
    },

    playerDie: function() {
        this.player.kill();

        this.deadSound.play();
        this.emitter.x = this.player.x;
        this.emitter.y = this.player.y;
        this.emitter.start(true, 800, null, 15);
        game.time.events.add(1000, this.startMenu, this);
        game.camera.shake(0.02, 300);
    },

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

    addMobileInputs: function() {
        var jumpButton = game.add.sprite(350, 240, 'jumpButton');
        jumpButton.inputEnabled = true;
        jumpButton.alpha = 0.5;
        jumpButton.events.onInputDown.add(this.jumpPlayer, this);

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

        var leftButton = game.add.sprite(50, 240, 'leftButton');
        leftButton.inputEnabled = true;
        leftButton.alpha = 0.5;
        leftButton.events.onInputOver.add(this.setLeftTrue, this);
        leftButton.events.onInputOut.add(this.setLeftFalse, this);
        leftButton.events.onInputDown.add(this.setLeftTrue, this);
        leftButton.events.onInputUp.add(this.setLeftFalse, this);

        var rightButton = game.add.sprite(130, 240, 'rightButton');
        rightButton.inputEnabled = true;
        rightButton.alpha = 0.5;
```

```
        rightButton.events.onInputOver.add(this.setRightTrue, this);
        rightButton.events.onInputOut.add(this.setRightFalse, this);
        rightButton.events.onInputDown.add(this.setRightTrue, this);
        rightButton.events.onInputUp.add(this.setRightFalse, this);
    },

    setLeftTrue: function() {
        this.moveLeft = true;
    },

    setLeftFalse: function() {
        this.moveLeft = false;
    },

    setRightTrue: function() {
        this.moveRight = true;
    },

    setRightFalse: function() {
        this.moveRight = false;
    },

    orientationChange: function() {
        if (game.scale.isPortrait) {
            game.paused = true;
            this.rotateLabel.text = 'rotate your device in landscape';
        }
        else {
            game.paused = false;
            this.rotateLabel.text = '';
        }
    },

    };
```

Game

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

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');
```