

MAKE 2D GAMES

IN **JAVASCRIPT** WITH **PHASER**



THOMAS PALEF

Contents

1 - Introduction	1
2 - Get started	3
2.1 - Set up project	4
2.2 - Code project	7
3 - Core mechanics	12
3.1 - Add player	13
3.2 - Create world	18
3.3 - Add coins	22
3.4 - Add enemies	27
3.5 - View code	31
4 - Scenes	36
4.1 - Overview	37
4.2 - Index file	40
4.3 - Load file	41
4.4 - Play file	43
4.5 - Menu file	45
4.6 - Game file	47

CONTENTS

5 - Jucify	48
5.1 - Add sounds	49
5.2 - Add animations	52
5.3 - Add tweens	55
5.4 - Add particles effects	60
5.5 - Improve camera	64
 6 - Improvements	 66
6.1 - Add best score	67
6.3 - Use custom fonts	69
6.4 - Improve difficulty	71
6.5 - Improve loading	74
 7 - Tilemaps	 76
7.1 - Create assets	77
7.2 - Display tilemap	81
 8 - Mobile friendly	 83
8.1 - Test	84
8.2 - Resize game	86
8.3 - Add touch inputs	90
8.4 - Add touch buttons	93
8.5 - Handle orientations	99
 9 - Next steps	 102
9.1 - Improve the game	103

CONTENTS

9.2 - Make new games	105
9.3 - Conclusion	106

1 - Introduction

Here's a brief introduction about the book's content.

What you will learn

This book has two main objectives:

- Build 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.
- Give you all the knowledge needed to build your own 2D games. For example a retro point & click, a clever puzzle game, an original platformer... it's completely up to you!

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

The Phaser framework

Phaser is a JavaScript framework to build 2D games. It is great for many reasons, here are the main ones:

- It is free, open source, and actively maintained.
- It has a large community around it, so it's easy to get help.
- It is both powerful and easy to use, which is a rare combination.
- It allows you to create games that can be played on any device.

Requirements

I wrote this book with beginners in mind, so it's perfectly okay if you've never made a game or if you've never heard of Phaser. The only thing required to understand this book is to be familiar with JavaScript.

2 - Get started

In this chapter we will get started with the Phaser framework. This include setting up Phaser and coding the structure of our game.



2.1 - Set up project

Let's see what we need to start making games with Phaser, in just three easy steps described below.

Download Phaser

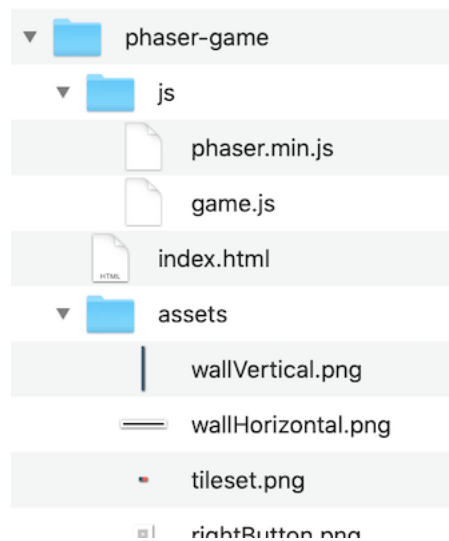
The first thing is to download the Phaser framework itself. This book uses Phaser version 3.55.2. I strongly recommend you to use the exact same version, to make sure there's no incompatibility.

You can explore the full [GitHub repository](#), but what we need is the phaser.min.js file that can be [downloaded here](#).

File structure

Create a new directory called "phaser-game". At the root of this folder add:

- "index.html", an empty file to display our game.
- "assets/", a directory containing all the sprites and sounds. The assets can be found in a folder that was included with this book.
- "js/" a directory that contains two JavaScript files:
 - "phaser.min.js", the Phaser framework that we downloaded in the previous step.
 - "game.js", an empty file that will contain our game code.



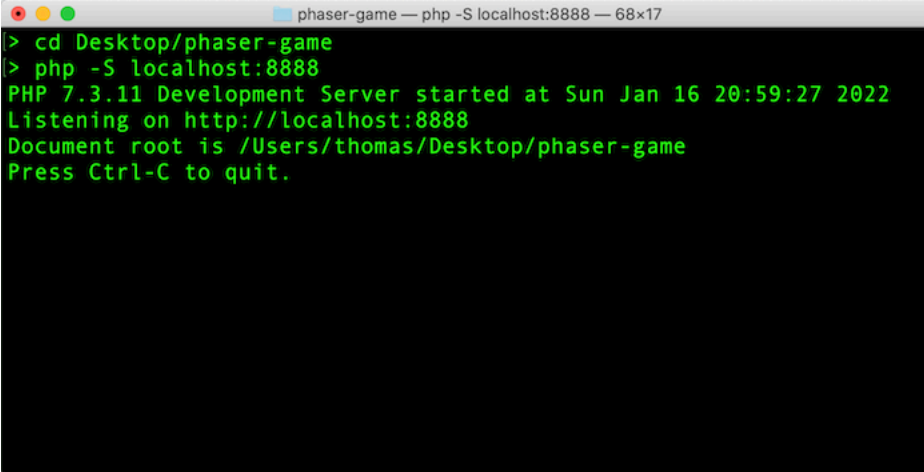
Run webserver

We need a local web server to test our game locally. Without it, JavaScript won't be able to load files from our local file system, like the game's assets.

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 apps. You can download **WAMP** (Windows) or **MAMP** (Mac). They both have a clean user interface with easy set up guides available online.
- Use an extension. If you are using Visual Studio Code, you could download an extension like **Live Server**.
- Use the command line with Python. Type `python -m http.server 8888` (or `python -m SimpleHTTPServer 8888` if you are using Python 2) to have a webserver running in the current directory.
- Use the command line with PHP. Type `php -S localhost:8888` to have a webserver running in the current directory.

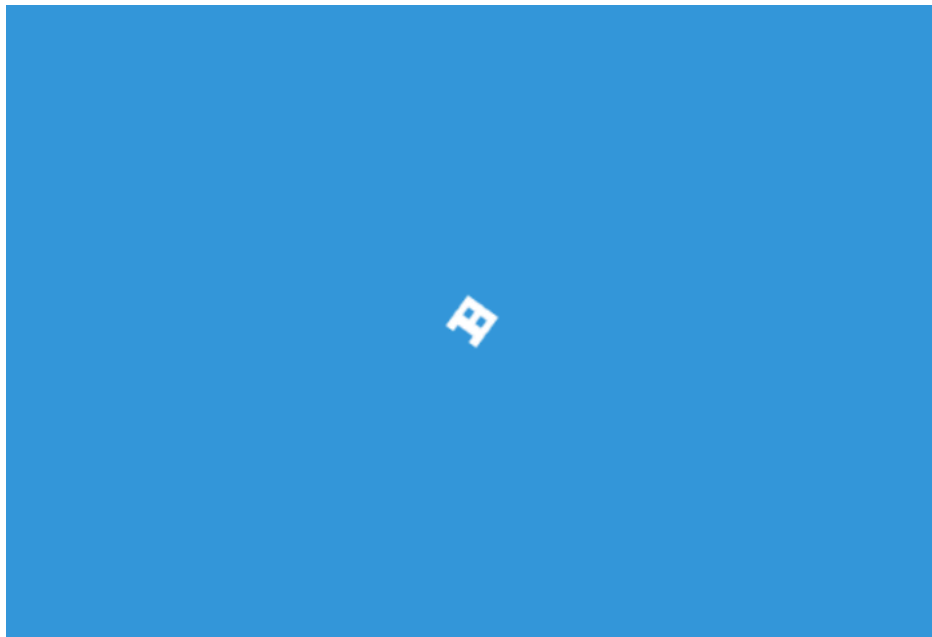
I personally use the last option.

A terminal window titled "phaser-game — php -S localhost:8888 — 68x17" with a black background and green text. The user has entered two commands: "cd Desktop/phaser-game" and "php -S localhost:8888". The terminal output shows that the PHP 7.3.11 Development Server has started at Sun Jan 16 20:59:27 2022, is listening on http://localhost:8888, and has a document root of /Users/thomas/Desktop/phaser-game. It also prompts the user to press Ctrl-C to quit.

```
phaser-game — php -S localhost:8888 — 68x17
> cd Desktop/phaser-game
> php -S localhost:8888
PHP 7.3.11 Development Server started at Sun Jan 16 20:59:27 2022
Listening on http://localhost:8888
Document root is /Users/thomas/Desktop/phaser-game
Press Ctrl-C to quit.
```

2.2 - Code project

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



HTML code

Here's the code to write in the index.html file.

```
1  <!DOCTYPE html>
2  <html>
3
4    <head>
5      <meta charset="utf-8" />
6      <title>First game</title>
7      <script src="js/phaser.min.js"></script>
8      <script src="js/game.js"></script>
```

```
9     </head>
10
11     <body>
12         <div id="game"></div>
13     </body>
14
15 </html>
```

This is basic HTML, but there are two important things to notice:

- We load two scripts: phaser.min.js (the Phaser framework) and game.js (our game).
- We have a `<div id="game">` element, that's where the game will be displayed.

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

Phaser scene

Every Phaser project is made out of scenes (like a loading scene, a menu scene, a play scene, etc.). For now we need only one that contains the whole game.

To create our scene we write a class with three methods. It's important to understand what each of these methods is doing, since they are key to how Phaser works.

So add this to the game.js file:

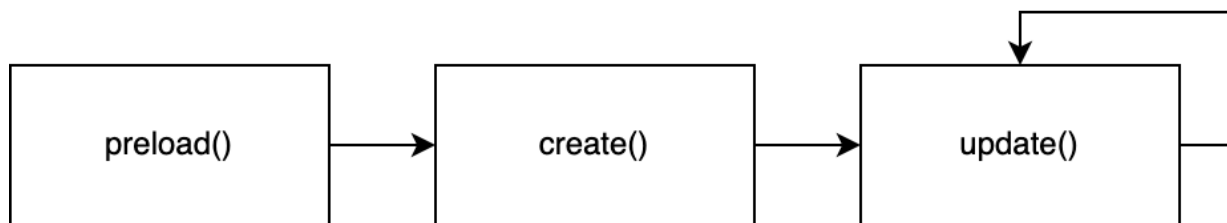
```
1  // Create our only scene called Main
2  class Main {
3      // The three methods (currently empty)
4
5      preload() {
6          // This method is called once at the beginning
7          // It will load all the assets, like sprites and sounds
8      }
9
10     create() {
```

```

11     // This method is called once, just after 'preload()'
12     // It will initialize our scene, like the positions of the sprites
13 }
14
15 update() {
16     // This method is called 60 times per second after 'create()'
17     // It will handle all the game's logic, like movements
18 }
19 }

```

To summarize, these methods will be called in this order: `preload()` > `create()` > `update()` > `update()` > `update()` > etc.



Now we need to fill these three methods with some code. Don't worry if you don't understand the code below, we will cover that in more details soon.

```

1  class Main {
2    preload() {
3      // Load the player image
4      this.load.image('player', 'assets/player.png');
5    }
6
7    create() {
8      // Add the player to the center of the screen
9      this.player = this.physics.add.sprite(250, 170, 'player');
10   }
11
12   update() {
13       // Increment the player angle 60 times per second
14       this.player.angle++;

```

```
15     }  
16 }
```

Phaser initialization

To initialize Phaser we call `Phaser.Game()` at the end of the JavaScript file. There are a lot of optional parameters available, but here are the main ones.

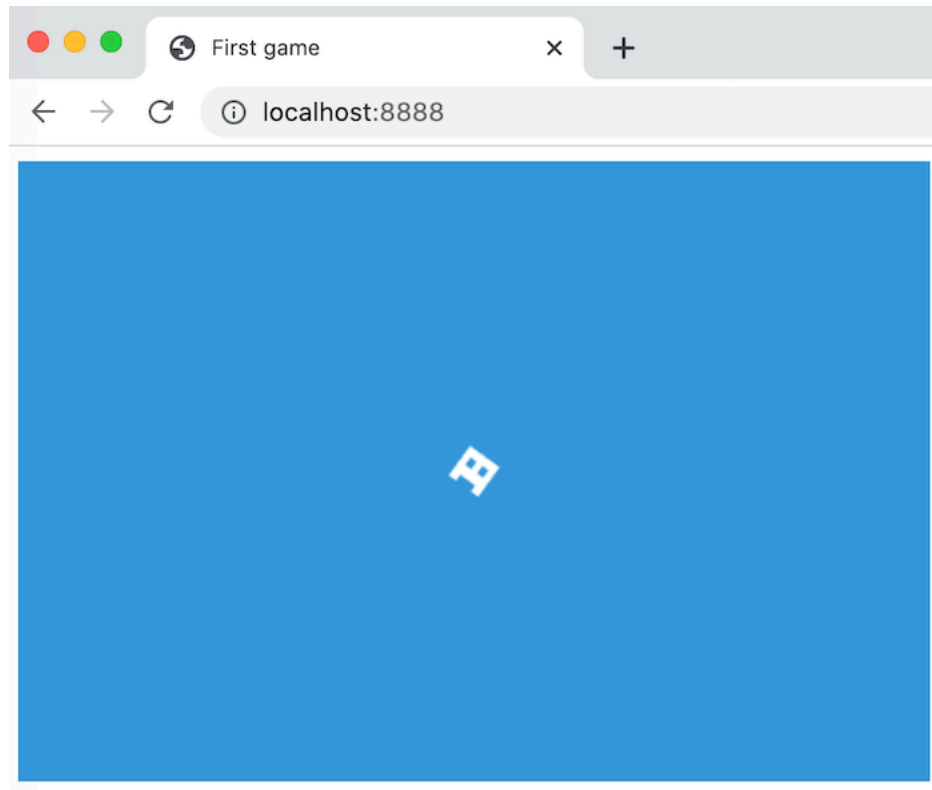
```
1  let game = new Phaser.Game({  
2    width: 500, // Width of the game in pixels  
3    height: 340, // Height of the game in pixels  
4    backgroundColor: '#3498db', // The background color (blue)  
5    physics: { default: 'arcade' }, // The physics engine to use  
6    parent: 'game', // The ID of the element that will contain the game  
7  });
```

Last but not least we need to add and start our scene like this:

```
1  // Add the scene to the game  
2  game.scene.add('main', Main);  
3  
4  // Start the scene  
5  game.scene.start('main');
```

Test the Project

And that's it! Now start the webserver (as explained in the previous part) to test that everything is working. You should see the player rotating on the screen.



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

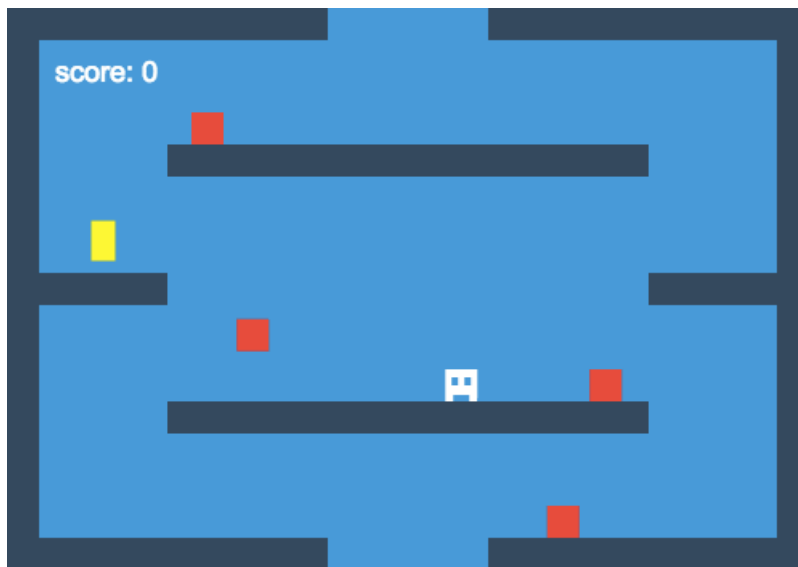
Clean up

Before moving to the next chapter, please delete the `this.player.angle++` line we wrote inside the `update()` method.

3 - Core mechanics

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.

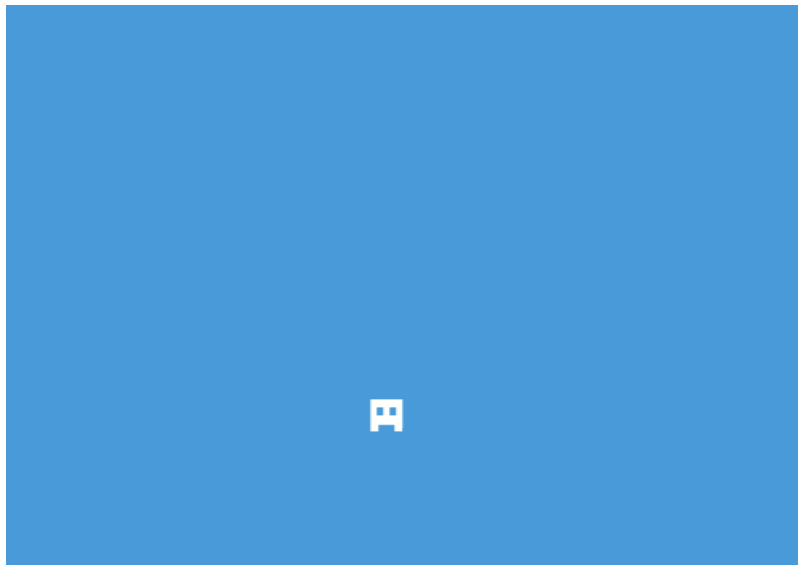


This chapter focuses on the core mechanics of the game: a player that can move around in a world, some coins to collect, and many enemies to avoid.

This might not look impressive at first, but keep in mind this is just the beginning!

3.1 - Add player

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



Load the player

Every time we want to use an asset in Phaser (image, sound, etc.) it has to be loaded first. For an image, we can do that with `this.load.image(imageName, imagePath)`:

- `imageName`, the new name that will be used to reference the image.
- `imagePath`, the path to the image.

That's why we added this line in the `preload()` method:

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

Display the player

Once the sprite is loaded we can display it on the screen with `this.physics.add.sprite(x, y, imageName)`:

- `x`, horizontal position of the sprite's center.
- `y`, vertical position of the sprite's center.
- `imageName`, the name of the image, as defined in the preload function.

So this line in the `create()` method is displaying the player in the center of the screen:

```
1 this.player = this.physics.add.sprite(250, 170, 'player');
```

A few important things to note here:

- We stored the player in the `this.player` property, this will allow us to access it in other methods of the class.
- We typed `this.physics` to create the player. It means the physics engine is enabled on that sprite, and that will make handling collisions effortless later on.
- If we set the player's position to `x = 0` and `y = 0`, then it would have been displayed in the top left corner of the screen.

Add gravity

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

```
1 this.player.body.gravity.y = 500;
```

The body property contains everything related to the physics of the sprite: gravity, velocity, collisions, etc.

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 could add this code to the `create()` method:

```
1  this.up = this.input.keyboard.addKey("up");
2  this.left = this.input.keyboard.addKey("left");
3  this.right = this.input.keyboard.addKey("right");
```

This works, but it's quite long to type. Since using arrow keys in a game is quite common, Phaser has a shortcut! So replace the previous code by this line:

```
1  this.arrow = this.input.keyboard.createCursorKeys();
```

Second, we create a new method to handle all the player's movements using `this.arrow`. Add this code just after the closing curly bracket of `update()`:

```
1  movePlayer() {
2      // If the left arrow key is pressed
3      if (this.arrow.left.isDown) {
4          // Move the player to the left
5          // Tthe velocity is in pixels per second
6          this.player.body.velocity.x = -200;
7      }
8
9      // If the right arrow key is pressed
10     else if (this.arrow.right.isDown) {
11         // Move the player to the right
12         this.player.body.velocity.x = 200;
13     }
14
15     // If neither the right or left arrow key is pressed
16     else {
17         // Stop the player
18         this.player.body.velocity.x = 0;
19     }
20
21     // If the up arrow key is pressed and the player is on the ground
22     if (this.arrow.up.isDown && this.player.body.onFloor()) {
23         // Move the player upward (jump)
24         this.player.body.velocity.y = -320;
```

```
25     }  
26 }
```

Finally we have to call `movePlayer()` inside of the `update()` method:

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

This checks 60 times per second if an arrow key is pressed, and moves the player accordingly.

More about sprites

For your information, here are some other things you can do with a sprite:

```
1  // Change the position of the sprite like this  
2  sprite.x = 21;  
3  sprite.y = 42;  
4  
5  // Or like this  
6  sprite.setPosition(21, 42);  
7  
8  // Return the width and height of the sprite  
9  sprite.width;  
10 sprite.height;  
11  
12 // Change the transparency of the sprite (0 = invisible, 1 = normal)  
13 sprite.alpha = 0.5;  
14  
15 // Change the angle of the sprite, in degrees  
16 sprite.angle = 84;  
17  
18 // Change the scale of the sprite  
19 sprite.setScale(2);  
20  
21 // Remove the sprite from the game
```

```
22 sprite.destroy();  
23  
24 // Return false if the sprite was destroyed  
25 sprite.active;
```

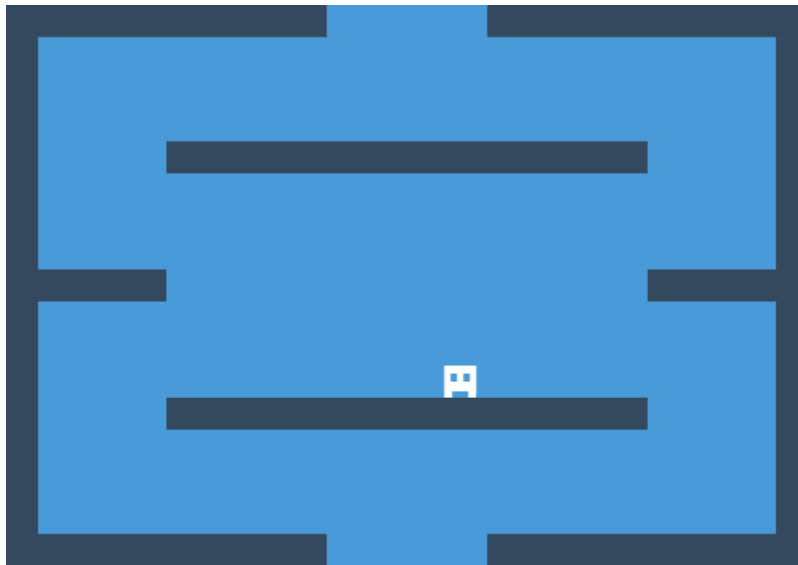
Conclusion

I recommend testing the game at the end of every section. Here you should be able to control the falling player before he disappears 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.

3.2 - Create world

Seeing the 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 code in this part.



Load the walls

With two sprites (a 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()` method:

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

As you can see, the name of the image doesn't have to be the same as its filename.

Add the walls

We have 10 walls to add. We could add them one by one, just like we did for the player. But there's a better way by using a Phaser feature called groups. This lets us group objects (like sprites) together.

Phaser has two types of groups: regular groups and static groups. We want our walls to be static because we don't want them to start falling when the player walks on them. That's why we need a static group here.

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 method:

```
1 createWorld() {
2     // Create an empty static group, with physics
3     this.walls = this.physics.add.staticGroup();
4
5     // Create the 10 walls in the group
6     this.walls.create(10, 170, 'wallV'); // Left
7     this.walls.create(490, 170, 'wallV'); // Right
8
9     this.walls.create(50, 10, 'wallH'); // Top left
10    this.walls.create(450, 10, 'wallH'); // Top right
11    this.walls.create(50, 330, 'wallH'); // Bottom left
12    this.walls.create(450, 330, 'wallH'); // Bottom right
13
14    this.walls.create(0, 170, 'wallH'); // Middle left
15    this.walls.create(500, 170, 'wallH'); // Middle right
16    this.walls.create(250, 90, 'wallH'); // Middle top
17    this.walls.create(250, 250, 'wallH'); // Middle bottom
18 }
```

And we should not forget to call `createWorld()` in the `create()` method:

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

Collisions

If we test the game we 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()` method:

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

This works because we previously used `this.physics` when creating the player and the walls.

For fun you could try to replace `this.physics.add.staticGroup()` we added earlier by `this.physics.add.group()` 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 method `playerDie()` that starts the main scene:

```
1  playerDie() {  
2    this.scene.start('main');  
3  }
```

And in the `update()` method we check the player's position. If he is below the bottom hole or above the top hole, we call `playerDie()`:

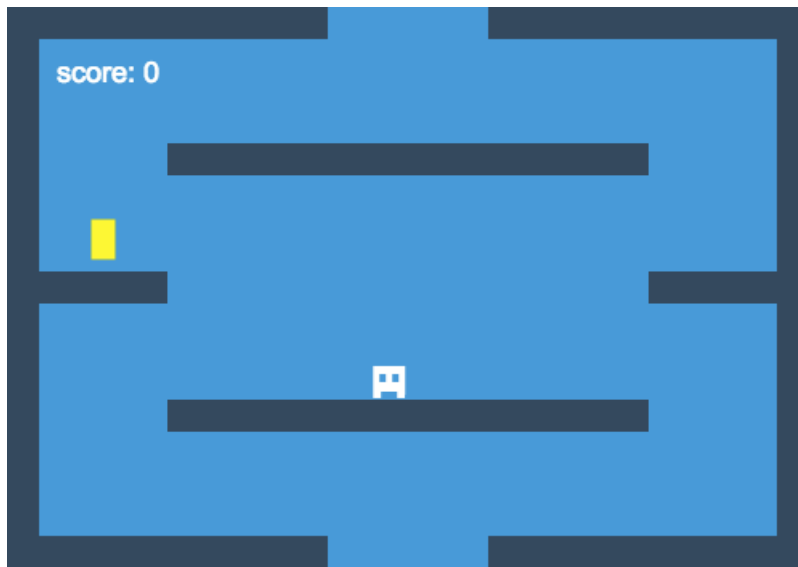
```
1  if (this.player.y > 340 || this.player.y < 0) {  
2    this.playerDie();  
3  }
```


Conclusion

We should be able to jump on the walls, run around, and die in the holes. This is starting to look like a game, and that's just the beginning.

3.3 - Add coins

In this part we will 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.



Load and add the coin

We start by loading the new sprite in the `preload()` method:

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

And we add the coin in the `create()` method:

```
1 this.coin = this.physics.add.sprite(60, 130, 'coin');
```

This should look familiar to you by now.

Display the score

Adding coins means we need a score. To display a text on the screen we have to use `this.add.text(x, y, text, style)`:

- `x`, horizontal position of the text's top left corner.
- `y`, vertical position of the text's top left corner.
- `text`, text to display.
- `style`, an object containing the style of the text.

Add the score in the top left corner of the game like this, in the `create()` method:

```
1 // Display the score
2 this.scoreLabel = this.add.text(30, 25, 'score: 0',
3   { font: '18px Arial', fill: '#fff' });
4
5 // Initialize the score variable
6 this.score = 0;
```

Here we 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 `this.physics.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 `this.physics.overlap(objectA, objectB)` instead:

- `objectA`, the first object to check.
- `objectB`, the second object to check

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

```
1  if (this.physics.overlap(this.player, this.coin)) {  
2    this.takeCoin();  
3  }
```

And now we create takeCoin():

```
1  takeCoin() {  
2    // Destroy the coin to make it disappear from the game  
3    this.coin.destroy();  
4  
5    // Increase the score by 5  
6    this.score += 5;  
7  
8    // Update the score label by using its 'text' property  
9    this.scoreLabel.setText('score: ' + this.score);  
10 }
```

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 two handy functions:

```
1  // Return a random integer between a and b  
2  let number = Phaser.Math.RND.between(a, b);  
3  
4  // Change the coin's position to x, y  
5  this.coin.setPosition(x, y);
```

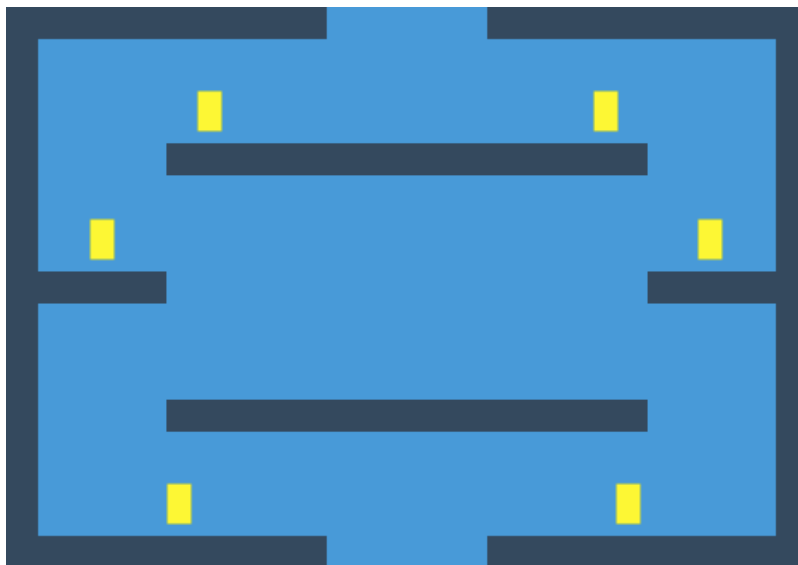
So we could replace the `this.coin.kill()` in the `takeCoin()` method by something like this:

```
1 // Compute two random numbers
2 var newX = Phaser.Math.RND.between(0, 500);
3 var newY = Phaser.Math.RND.between(0, 340);
4
5 // Set the new coin position
6 this.coin.setPosition(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 six positions where the coin may appear, and randomly pick one of them.



Here's a new method that does just that:

```
1  updateCoinPosition() {
2    // Store all the possible coin positions in an array
3    let positions = [
4      { x: 140, y: 60 },
5      { x: 360, y: 60 },
6      { x: 60, y: 140 },
7      { x: 440, y: 140 },
8      { x: 130, y: 300 },
9      { x: 370, y: 300 },
10   ];
11
12   // Remove the current coin position from the array
13   positions = positions.filter(coin => coin.x !== this.coin.x);
14
15   // Randomly select a position from the array
16   let newPosition = Phaser.Math.RND.pick(positions);
17
18   // Set the new position of the coin
19   this.coin.setPosition(newPosition.x, newPosition.y);
20 }
```

And finally we edit the takeCoin() method like this:

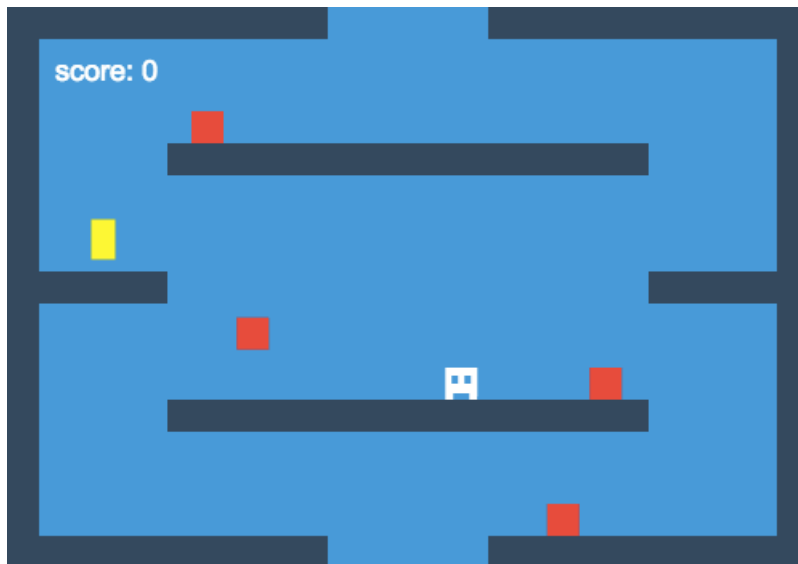
```
1  takeCoin() {
2    // Update the score
3    this.score += 5;
4    this.scoreLabel.setText('score: ' + this.score);
5
6    // Change the coin position
7    this.updateCoinPosition();
8  }
```

Conclusion

Now you can collect some coins and see your score increase!

3.4 - Add enemies

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



Load the enemy

As usual, we start by loading the sprite in the `preload()` method:

```
1 this.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. But this time we want the enemies to move, so we will use a regular group instead of a static group.

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

```
1  this.enemies = this.physics.add.group();
```

Add the enemies - code

We want new enemies to appear every few seconds. For this we can use `this.time.addEvent()`. It's a function that takes an object as parameter, and the main properties are:

- `delay`, the delay in ms between each callback.
- `callback`, the function that will be called.
- `loop`, whether the callback function should be repeated or not.

Here's how to add our event in the `create()` method:

```
1  // Call 'addEnemy' every 2.2 seconds
2  this.time.addEvent({
3    delay: 2200,
4    callback: () => this.addEnemy(),
5    loop: true,
6  });
```

Next we can create the new `addEnemy()` method:

```
1  addEnemy() {
2    let enemy = this.enemies.create(250, -10, 'enemy');
3
4    enemy.body.gravity.y = 500;
5    enemy.body.velocity.x = Phaser.Math.RND.pick([-100, 100]);
6    enemy.body.bounce.x = 1;
7
8    this.time.addEvent({
9      delay: 10000,
10     callback: () => enemy.destroy(),
11   });
12 }
```

There's a lot going on here, so we should spend some time reviewing this code.

Add the enemies - explained

First we create a new enemy in our group. We set its y position to -10 so that it's just above the top hole.

```
1 let enemy = this.enemies.create(250, -10, 'enemy');
```

We add some gravity to the enemy, just like we did for the player earlier.

```
1 enemy.body.gravity.y = 500;
```

We give some horizontal velocity to the enemy to make it move right or left. We use `Phaser.Math.RND.pick()` to either pick -100 or 100 for the velocity.

```
1 enemy.body.velocity.x = Phaser.Math.RND.pick([-100, 100]);
```

When an enemy is moving right and hits a wall, we want it to start moving left. One easy way to achieve this is to use the `bounce` property of the sprite. It 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:

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

Finally, these lines will automatically destroy the sprite after 10 seconds, that's roughly the time it takes for the enemy to fall into the bottom hole.

```
1 this.time.addEvent({  
2   delay: 10000,  
3   callback: () => enemy.destroy(),  
4 });
```

Collisions

For the collisions we want to do two 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()` method:

```
1 // Make the enemies and walls collide
2 this.physics.collide(this.enemies, this.walls);
3
4 // Call the 'playerDie' method when the player and an enemy overlap
5 if (this.physics.overlap(this.player, this.enemies)) {
6     this.playerDie();
7 }
```

More about groups

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

```
1 // A group can act like a giant sprite
2 // So it has the same properties you can edit
3 group.x;
4 group.y;
5 group.alpha;
6 group.angle;
7
8 // Return the number of sprites in a group
9 group.countActive();
10
11 // Add an object to a group. It can be a sprite, a label, etc.
12 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.

That's great, but the game is quite boring. That's why this book is not over yet!

3.5 - View code

To make sure we are on the same page, here's the code we wrote so far.

index.html

```
1  <!DOCTYPE html>
2  <html>
3
4    <head>
5      <meta charset="utf-8" />
6      <title>First game</title>
7      <script src="js/phaser.min.js"></script>
8      <script src="js/game.js"></script>
9    </head>
10
11   <body>
12     <div id="game"></div>
13   </body>
14
15 </html>
```

game.js

```
1  class Main {
2    preload() {
3      this.load.image('background', 'assets/background.png');
4      this.load.image('player', 'assets/player.png');
5      this.load.image('coin', 'assets/coin.png');
6      this.load.image('enemy', 'assets/enemy.png');
7      this.load.image('wallV', 'assets/wallVertical.png');
8      this.load.image('wallH', 'assets/wallHorizontal.png');
9    }
10
11   create() {
12     this.player = this.physics.add.sprite(250, 170, 'player');
13     this.player.body.gravity.y = 500;
14
15     this.arrow = this.input.keyboard.createCursorKeys();
16
17     this.createWorld();
18
19     this.coin = this.physics.add.sprite(60, 130, 'coin');
20
21     this.scoreLabel = this.add.text(30, 25, 'score: 0',
22       { font: '18px Arial', fill: '#fff' });
23     this.score = 0;
24
25     this.enemies = this.physics.add.group();
26     this.time.addEvent({
27       delay: 2200,
28       callback: () => this.addEnemy(),
29       loop: true,
30     });
31   }
32
33   update() {
34     this.physics.collide(this.player, this.walls);
35     this.physics.collide(this.enemies, this.walls);
36   }
```

```
37     if (!this.player.active) {
38         return;
39     }
40
41     this.movePlayer();
42
43     if (this.physics.overlap(this.player, this.coin)) {
44         this.takeCoin();
45     }
46
47     if (this.player.y > 340 || this.player.y < 0) {
48         this.playerDie();
49     }
50
51     if (this.physics.overlap(this.player, this.enemies)) {
52         this.playerDie();
53     }
54 }
55
56 createWorld() {
57     this.walls = this.physics.add.staticGroup();
58
59     this.walls.create(10, 170, 'wallV');
60     this.walls.create(490, 170, 'wallV');
61     this.walls.create(50, 10, 'wallH');
62     this.walls.create(450, 10, 'wallH');
63     this.walls.create(50, 330, 'wallH');
64     this.walls.create(450, 330, 'wallH');
65     this.walls.create(0, 170, 'wallH');
66     this.walls.create(500, 170, 'wallH');
67     this.walls.create(250, 90, 'wallH');
68     this.walls.create(250, 250, 'wallH');
69 }
70
71 movePlayer() {
72     if (this.arrow.left.isDown) {
```

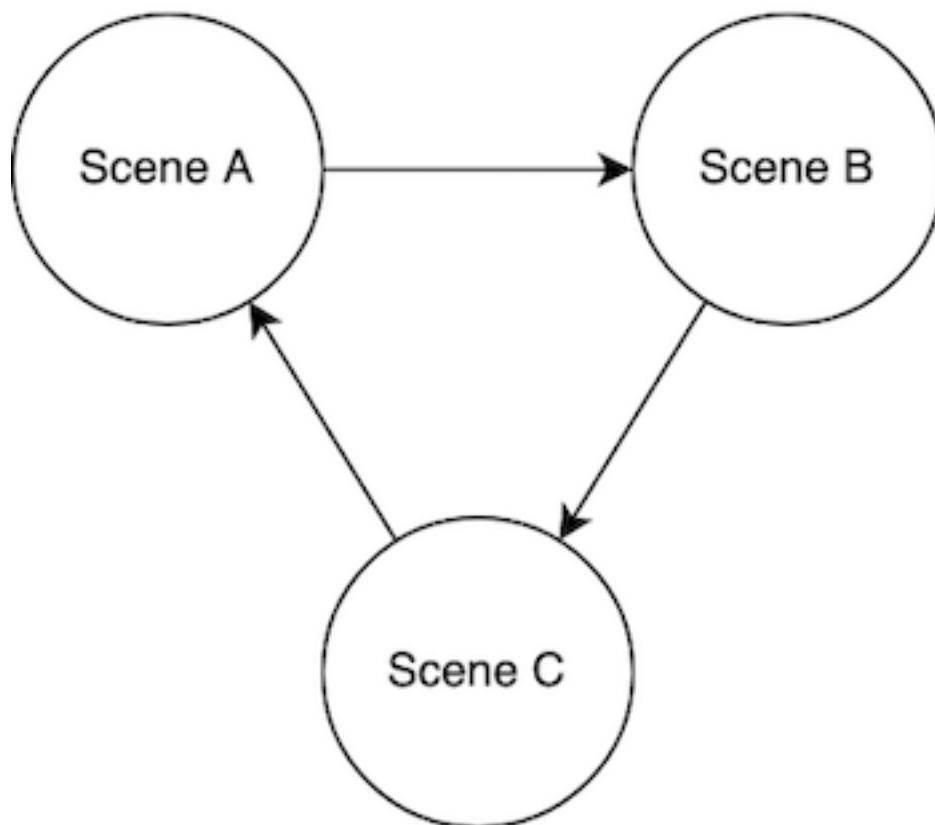
```
73     this.player.body.velocity.x = -200;
74   } else if (this.arrow.right.isDown) {
75     this.player.body.velocity.x = 200;
76   } else {
77     this.player.body.velocity.x = 0;
78   }
79
80   if (this.arrow.up.isDown && this.player.body.onFloor()) {
81     this.player.body.velocity.y = -320;
82   }
83 }
84
85 takeCoin() {
86   this.score += 5;
87   this.scoreLabel.setText('score: ' + this.score);
88
89   this.updateCoinPosition();
90 }
91
92 updateCoinPosition() {
93   let positions = [
94     { x: 140, y: 60 },
95     { x: 360, y: 60 },
96     { x: 60, y: 140 },
97     { x: 440, y: 140 },
98     { x: 130, y: 300 },
99     { x: 370, y: 300 },
100  ];
101
102   positions = positions.filter(coin => coin.x !== this.coin.x);
103
104   let newPosition = Phaser.Math.RND.pick(positions);
105
106   this.coin.setPosition(newPosition.x, newPosition.y);
107 }
108
```

```
109   addEnemy() {
110       let enemy = this.enemies.create(250, -10, 'enemy');
111
112       enemy.body.gravity.y = 500;
113       enemy.body.velocity.x = Phaser.Math.RND.pick([-100, 100]);
114       enemy.body.bounce.x = 1;
115
116       this.time.addEvent({
117         delay: 10000,
118         callback: () => enemy.destroy(),
119       });
120   }
121
122   playerDie() {
123       this.scene.start('main');
124   }
125 };
126
127 let game = new Phaser.Game({
128   width: 500,
129   height: 340,
130   backgroundColor: '#3498db',
131   physics: { default: 'arcade' },
132   parent: 'game',
133 });
134
135 game.scene.add('main', Main);
136 game.scene.start('main');
```

4 - Scenes

In this chapter we will continue to work on our game. This time our main focus will be to create scenes.

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 - Overview

Currently our game has only one scene, 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 scene

A scene in Phaser is a part of a game, like a menu scene, a play scene, a game over scene, etc. Each one is its own JavaScript class.

You can see below what a scene looks like. It should be familiar to you, since that's what we used to build our game.

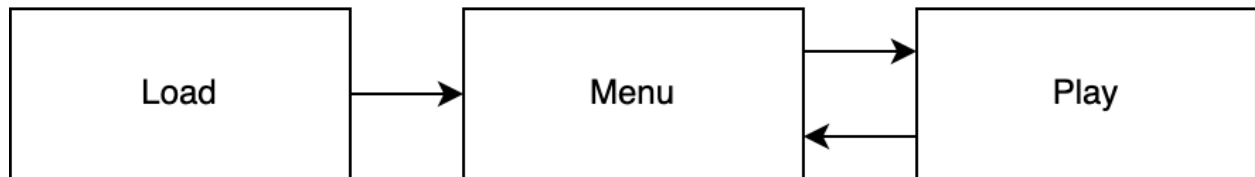
```
1  // Create one scene
2  class SceneName {
3
4      // Define all the method of the scene
5
6      preload() {
7          // This method will be executed at the beginning
8      }
9
10     create() {
11         // This method is called after the 'preload' method
12     }
13
14     update() {
15         // This method is called 60 times per second
16     }
```

```
17  
18  // And maybe add some other methods  
19 }
```

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

New scenes

Our game will have three scenes: Load, Menu, and Play.



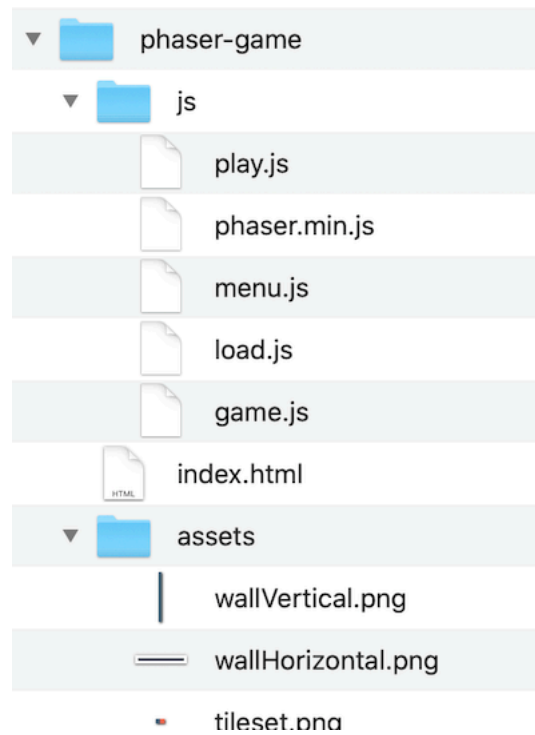
- Load, to load all the game's assets. Once everything is loaded, we will start the menu scene.
- Menu, to display the menu of the game. When we press the up arrow key, we will start the play scene.
- Play, to actually play the game. When dying, we will go back to the menu scene.

File structure

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

- Rename the game.js file into play.js.
- Create three empty JavaScript files in the js/ directory: load.js, menu.js, and game.js.

Things should look like this:



4.2 - Index file

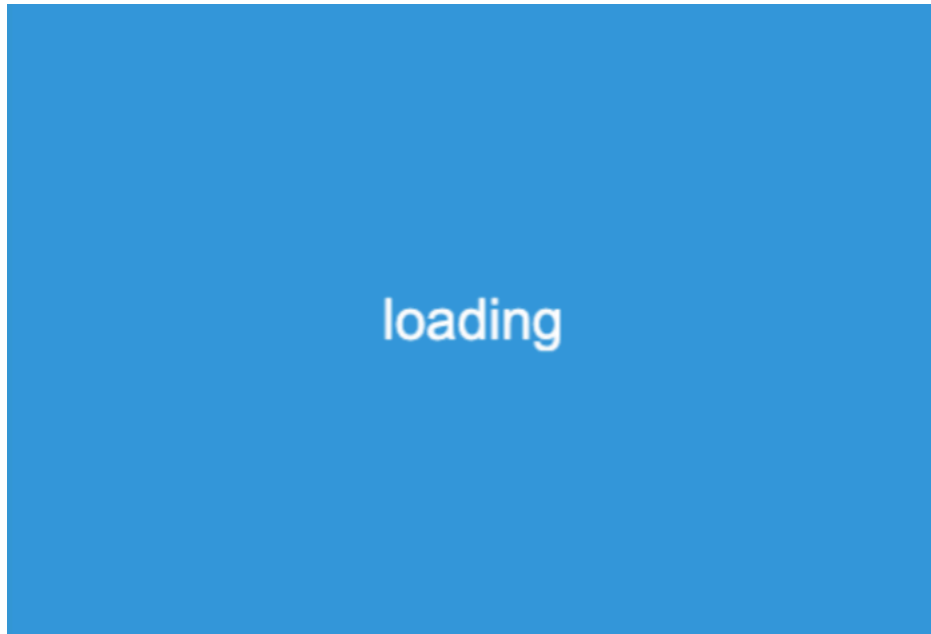
The first thing to edit is the index.html file. The code is the same as before, except that we are loading the new JavaScript files.

```
1  <!DOCTYPE html>
2  <html>
3
4    <head>
5      <meta charset="utf-8" />
6      <title>First game</title>
7      <script src="js/phaser.min.js"></script>
8      <script src="js/load.js"></script>
9      <script src="js/menu.js"></script>
10     <script src="js/play.js"></script>
11     <script src="js/game.js"></script>
12   </head>
13
14   <body>
15     <div id="game"></div>
16   </body>
17
18 </html>
```

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

4.3 - Load file

The load scene is quite simple. It will preload all the assets of the game.



Since the game doesn't have a lot of assets to load, you may not even have the time to see this scene.

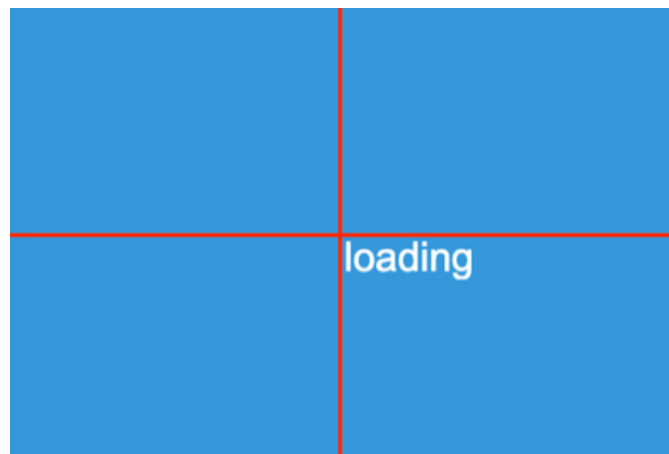
```
1  class Load {
2    preload() {
3      // Load all assets
4      this.load.image('background', 'assets/background.png');
5      this.load.image('player', 'assets/player.png');
6      this.load.image('coin', 'assets/coin.png');
7      this.load.image('enemy', 'assets/enemy.png');
8      this.load.image('wallV', 'assets/wallVertical.png');
9      this.load.image('wallH', 'assets/wallHorizontal.png');
10
11     // Display a loading label
12     let loadLabel = this.add.text(250, 170, 'loading',
```

```
13     { font: '30px Arial', fill: '#fff' });
14
15     // Change the point of origin of the text
16     // To make sure the text will be centered on the screen
17     loadLabel.setOrigin(0.5, 0.5);
18 }
19
20 create() {
21     // Start the menu scene
22     this.scene.start('menu');
23 }
24 }
```

Two things to note here.

First, we loaded a new asset: `assets/background.png`. It's a background image that we will use in the menu scene.

Second, we had to change the point of origin of the text. If we didn't, then the text would have started at the center of the screen but it would not have been centered:



Calling `setOrigin(0.5, 0.5)` is done by default for sprites, but not for labels.

4.4 - Play file

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

- The scene is now called Play instead of Main.
- The `preload()` method is no longer needed since we already load all our assets in the `load.js` file.
- The `playerDie()` method starts the menu scene.
- And the Phaser initialization code has been removed (we will put it back in the `game.js` file).

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

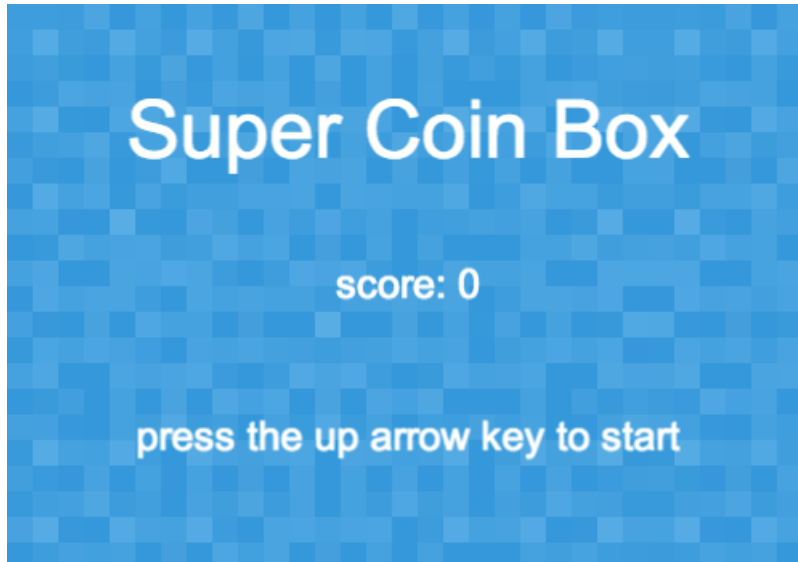
```
1  // New name for the scene
2  class Play {
3
4    // Removed the preload function
5
6    create() {
7      // No changes
8    }
9
10   update() {
11     // No changes
12   }
13
14   createWorld() {
15     // No changes
16   }
17
18   movePlayer() {
19     // No changes
```

```
20     }
21
22     takeCoin() {
23         // No changes
24     }
25
26     updateCoinPosition() {
27         // No changes
28     }
29
30     addEnemy() {
31         // No changes
32     }
33
34     playerDie() {
35         // When the player dies, we go to the menu
36         this.scene.start('menu', { score: this.score });
37     }
38 };
39
40 // Delete all Phaser initialization code
```

Notice that there is a second parameter when calling `this.scene.start()`. It's an object containing the score that will be passed to the `create()` method of the menu scene.

4.5 - Menu file

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



Here's the full source code below with lots of comments.

```
1  class Menu {
2
3    // The 'create' method now has a 'data' parameter
4    // It contains an object coming from the play scene
5    create(data) {
6      // Retrieve the score, if there is one
7      let score = data.score ? data.score : 0;
8
9      // Display a background image
10     // An image is like a lightweight sprite that
11     // doesn't need physics or animations
12     // It's perfect for logos, backgrounds, etc.
13     this.add.image(250, 170, 'background');
14
15     // Display the name of the game
```

```
16     let nameLabel = this.add.text(250, 80, 'Super Coin Box',
17         { font: '50px Arial', fill: '#fff' });
18     nameLabel.setOrigin(0.5, 0.5);
19
20     // Display the score
21     let scoreText = 'score: ' + score;
22     let scoreLabel = this.add.text(250, 170, scoreText,
23         { font: '25px Arial', fill: '#fff' });
24     scoreLabel.setOrigin(0.5, 0.5);
25
26     // Display how to start the game
27     let startText = 'press the up arrow key to start';
28     let startLabel = this.add.text(250, 260, startText,
29         { font: '25px Arial', fill: '#fff' });
30     startLabel.setOrigin(0.5, 0.5);
31
32     // Store the up arrow key
33     this.upKey = this.input.keyboard.addKey('up');
34 }
35
36 update() {
37     // When to up arrow key is down
38     if (this.upKey.isDown) {
39         // Start the play scene
40         this.scene.start('play');
41     }
42 }
43 }
```

It's important to add the background image at the beginning of the method, 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.

4.6 - Game file

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

```
1  // Initialize Phaser (no changes)
2  let game = new Phaser.Game({
3    width: 500,
4    height: 340,
5    backgroundColor: '#3498db',
6    physics: { default: 'arcade' },
7    parent: 'game',
8  });
9
10 // Add all the scenes
11 game.scene.add('load', Load);
12 game.scene.add('menu', Menu);
13 game.scene.add('play', Play);
14
15 // Start the 'load' scene
16 game.scene.start('load');
```

Now you can test the game to see all the new scenes in action. To summarize what is happening:

1. First the load scene is called to load all the game's assets.
2. After that the menu is shown.
3. When we press the up arrow key the play scene starts.
4. And when the player 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 focus on 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 to 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 scenes in our game.



5.1 - Add sounds

Let's add some sound effects to our game.

Compatibility

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

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, so I recommend using multiple audio formats.

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 `this.load.audio()`.

We load three sounds in the `preload()` method of the `load.js` file:

```
1 // Sound when the player jumps
2 this.load.audio('jump', ['assets/jump.ogg', 'assets/jump.mp3']);
3
4 // Sound when the player takes a coin
5 this.load.audio('coin', ['assets/coin.ogg', 'assets/coin.mp3']);
6
7 // Sound when the player dies
8 this.load.audio('dead', ['assets/dead.ogg', 'assets/dead.mp3']);
```

We specified two 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()` method of the `play.js` file:

```
1  this.jumpSound = this.sound.add('jump');
2  this.coinSound = this.sound.add('coin');
3  this.deadSound = this.sound.add('dead');
```

Play the sounds

Finally we want to actually play the sounds with `play()`:

```
1  // Add this inside the 'movePlayer' method, in the 'if (player jumps)'
2  this.jumpSound.play();
3
4  // Put this in the 'takeCoin' method
5  this.coinSound.play();
6
7  // And this in the 'playerDie' method
8  this.deadSound.play();
```

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

Background music

For your 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.

```
1 // Load the music in two different formats in the load.js file
2 this.load.audio('music', ['assets/music.ogg', 'assets/music.mp3']);
3
4 // Add and start the music in the 'create' method of the play.js file
5 // Because we want to play the music when the play state starts
6 this.music = this.load.audio('music'); // Add the music
7 this.music.loop = true; // Make it loop
8 this.music.play(); // Start the music
9
10 // And don't forget to stop the music in the 'playerDie' method
11 // Otherwise the music would keep playing
12 this.music.stop();
```

However be careful with the size of the music file since it may be quite large. Games that take forever to load are something to avoid.

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 below for better readability):



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

To load this image we have to use `this.load.spritesheet()`. This function needs to know the width and height of each frame.

```
1 // Replace this in the load.js file
2 this.load.image('player', 'assets/player.png');
3
4 // By this
5 this.load.spritesheet('player', 'assets/player2.png', {
6     frameWidth: 20,
7     frameHeight: 20,
8 });
```

Add the animations

Whether we load a sprite with `this.load.image()` or with `this.load.spritesheet()`, it doesn't change the way we display the sprite. However, we need to add the animations with the `animations.create()` function.

It takes a single object as a parameter. It can have many properties, but here are the most important ones:

- `key`, the name of the animation.
- `frames`, an array containing the frames of the animation.
- `frameRate`, the speed of the animation, in frames per second.
- `repeat`, how many times we want the animation to repeat (-1 means forever).

We are going to add two different animations to the player in the `create()` method of the `play.js` file:

```
1  // Create the 'right' animation by looping the frames 1 and 2
2  this.ans.create({
3    key: 'right',
4    frames: this.ans.generateFrameNumbers('player', {frames: [1, 2]}),
5    frameRate: 8,
6    repeat: -1,
7  });
8
9  // Create the 'left' animation by looping the frames 3 and 4
10 this.ans.create({
11   key: 'left',
12   frames: this.ans.generateFrameNumbers('player', {frames: [3, 4]}),
13   frameRate: 8,
14   repeat: -1,
15 });
```

Note that we used `this.ans.generateFrameNumbers()` for the frames. The alternative would have been to manually set the frame numbers like this: `[{ key: 'player', frame: 1 }, { key: 'player', frame: 2 }]`.

Play the animations

Now all we have to do is play the animations. We need to edit the `movePlayer()` method like this:

```
1 movePlayer() {
2   if (this.arrow.left.isDown) {
3     this.player.body.velocity.x = -200;
4     this.player.anims.play('left', true); // Left animation
5   } else if (this.arrow.right.isDown) {
6     this.player.body.velocity.x = 200;
7     this.player.anims.play('right', true); // Right animation
8   } else {
9     this.player.body.velocity.x = 0;
10    this.player.setFrame(0); // Change frame (stand still)
11  }
12
13  // Then no changes to the jump code
14 }
```

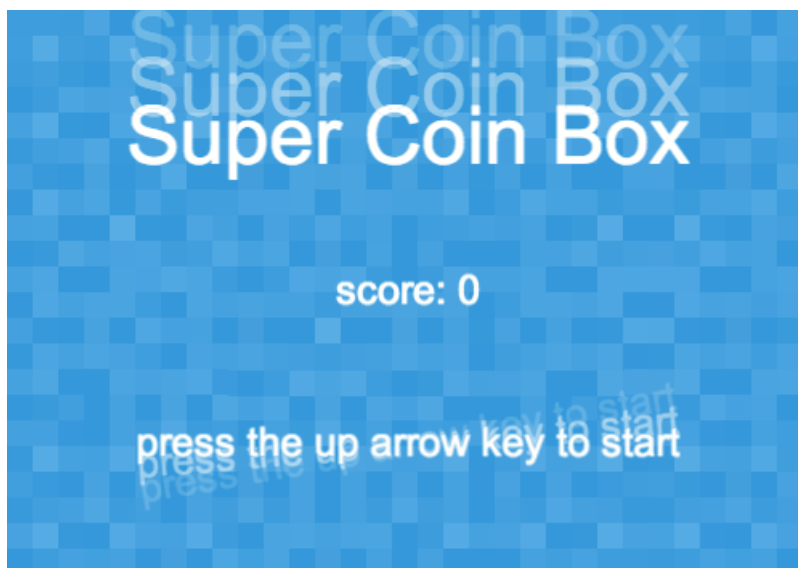
Setting the frame to 0 has the benefit of stopping the previous animation. Otherwise we would have to call `this.player.anims.stop()` to make sure the player was no longer running.

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 object 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 four tweens to our game.



Create a tween

First, here's a quick preview of how tweens work.

We only need one function: `this.tweens.add()`. It takes an object as a parameter that can have many properties. Here are the main ones:

- `target`, the object to tween.
- `x`, `y`, `scale`, `alpha`, and so on, the property of the object to tween.
- `duration`, the length of the tween in milliseconds.
- `ease`, the speed curve of the tween.
- `repeat`, how many times should the tween repeat (-1 means forever).
- `yoyo`, whether the tween reverses itself when it reaches the end.

Scale the coin

We start with something simple: scale the coin when it appears on the screen. So add this to the `takeCoin()` method:

```
1 // Scale the coin to 0 to make it invisible
2 this.coin.setScale(0);
3
4 // Create a tween
5 this.tweens.add({
6   targets: this.coin, // on the coin
7   scale: 1, // to scale it to 1 (its original size)
8   duration: 300, // in 300ms
9 });
```

And that's it!

Scale the player

Here's another similar 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()` method:

```
1 // Create a tween
2 this.tweens.add({
3   targets: this.player, // on the player
4   scale: 1.3, // to scale it to 1
5   duration: 100, // in 100ms
6   yoyo: true, // then perform the tween in reverse
7 });
```

The `yoyo` option will do the previous tween in reverse. So after scaling the player to 1.3 in 100ms, it will scale it back to 1 in 100ms.

Move the label

We saw how to tween the `scale` of a sprite, but we can tween any property of any object! For example let's tween the `y` position of a text.

In the menu we want the name of the game 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:

```
1 // Change the y position to -50 so we don't see the label
2 let nameLabel = this.add.text(250, -50, 'Super Coin Box',
3   { font: '50px Arial', fill: '#fff' });
```

And now we can start the actual tweening:

```
1 // Create a tween
2 this.tweens.add({
3   targets: nameLabel, // on the nameLabel
4   y: 80, // that changes the y position to 80
5   duration: 1000, // in 1000 ms
6   ease: 'bounce.out', // with a bouncing animation
7 });
```

The code above will move the label from its initial position (`y = -50`) to its new position (`y = 80`) in 1 second.

Rotate the label

Last example. 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 `repeat` option like this:

```
1 // Create a tween
2 this.tweens.add({
3   targets: startLabel, // on the startLabel
4   angle: { from: -2, to: 2 }, // change the angle from -2 to 2
5   duration: 1000, // in 1000 ms-
6   yoyo: true, // then perform the tween in reverse
7   repeat: -1, // and repeat the tween indefinitely
8 });
```

Note that the rotation takes place where the anchor point was defined (at the center of the label).

More about tweens

As you see, tweens are really powerful and flexible.

Here are just a few more examples of options you can set when creating a tween:

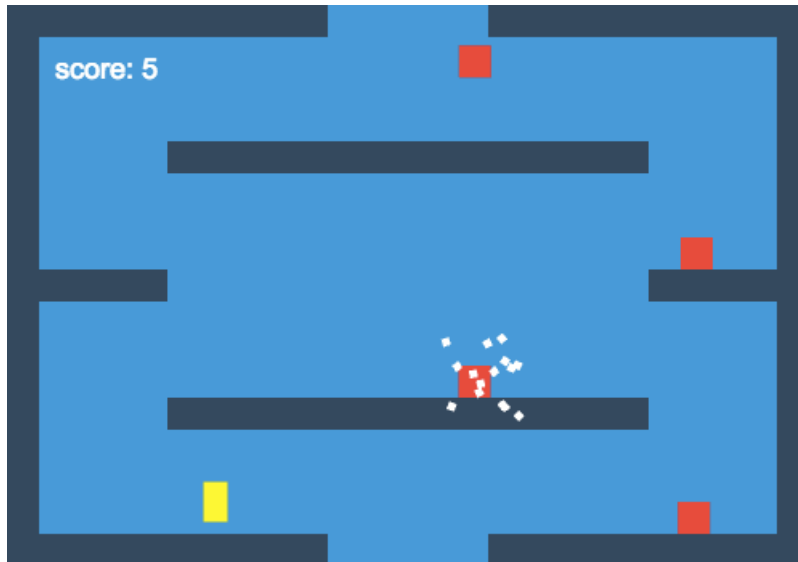
```
1 this.tweens.add({
2   x: '+=100', // Change a property relative to the current value
3   onComplete: () => console.log('done'), // Callback when the tween en\
4 ds
5   delay: 1000, // Delay before the tween starts
6   repeatDelay: 500, // Delay before the tween repeats
7   hold: 500, // Delay before the tween yoyos
8 });
```

And here are some easing you can try for the ease property:

```
1  // Bounce
2  'bounce.in' 'bounce.out' 'bounce.inout'
3
4  // Sine
5  'sine.in' 'sine.out' 'sine.inout'
6
7  // Quadratic
8  'quadratic.in' 'quadratic.out' 'quadratic.inout'
9
10 // Cubic
11 'cubic.in' 'cubic.out' 'cubic.inout'
12
13 // Exponential
14 'exponential.in' 'exponential.out' 'exponential.inout'
```

5.4 - Add particles effects

If we want to add explosions, rain, or dust to our game, then particle effects are the way to go. We will use them to make our player explode when it is hit by an enemy.



Load the particle

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

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

Create the emitter

Our explosion can be done in three steps:

1. Create the particles with the image we loaded.
2. Create an emitter to configure how the particles should behave.
3. Actually do the explosion.

The first two points should be done in the `create()` method of the `play.js` file:


```
1  // Create our particles
2  let particles = this.add.particles('pixel');
3
4  // Create the emitter
5  this.emitter = particles.createEmitter({
6    // Set the number of particles to 15
7    quantity: 15,
8
9    // Set the particles' speed between -150 and 150 pixels per seconds
10   speed: { min: -150, max: 150 },
11
12   // Set the particles' scale from two time their original size to 0
13   scale: { start: 2, end: 0.1 },
14
15   // Set the particles' lifespan to 800
16   lifespan: 800,
17
18   // Don't start the explosion right away
19   on: false,
20 });
```

Start the emitter

Now that we have our emitter, we can update the `playerDie()` method to actually do the explosion:

```
1  playerDie() {
2    // No changes
3    this.deadSound.play();
4
5    // Set the position of the emitter on top of the player
6    this.emitter.setPosition(this.player.x, this.player.y);
7
8    // Start the emitter
9    this.emitter.explode();
```

```
10
11  // No changes
12  this.scene.start('menu', { score: this.score });
13 }
```

We are almost done, we just need to fix a couple of issues.

Add a delay

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

So update the `playerDie()` method like this:

```
1  // Replace this
2  this.scene.start('menu', { score: this.score });
3
4  // By this
5  this.time.addEvent({
6    delay: 1000,
7    callback: () => this.scene.start('menu', { score: this.score }),
8  });
```

Kill the player

We see particles! But something weird is going on: we can still move the player around when we touch an enemy.

First, add this to the beginning of the `playerDie()` method to actually kill the player:

```
1  this.player.destroy();
```

Second, we need to make sure we won't call `playerDie()` or `movePlayer()` when the player is dead. So add this in the `update()` method, just after all the two `this.physics.collide()`:

```
1 // If the player is dead, do nothing
2 if (!this.player.active) {
3     return;
4 }
```

Now everything should work properly.

More about particles

We are done for our little explosion, however you should know that we can do a lot more with emitters. For example here are some additional properties to set when creating an emitter:

```
1 particles.createEmitter({
2     // Add gravity to the particles
3     gravityY: 500,
4
5     // Set the transparency of the particles
6     alpha: 1,
7
8     // Set the angle of the particles
9     angle: 45
10
11     // Set the bounce of the particles (if you use `physics.collide)
12     bounce: 0,
13 });
```

5.5 - Improve 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.

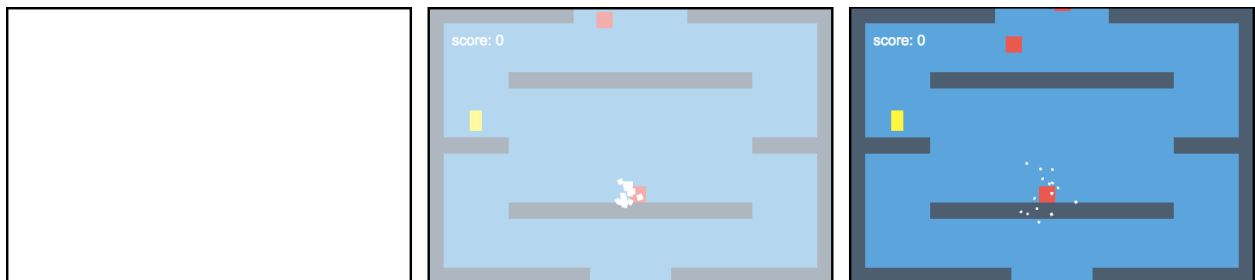
To do so, add this line in the `playerDie()` method:

```
1 // Flash the screen for 300ms
2 this.cameras.main.flash(300);
```

By default the color is white, but we can change that by passing three optional parameters: red, blue, and green values. Each parameter should be between 0 and 255. So it would look something like this for a red-ish flash:

```
1 // Lots of red, and a little of green and blue
2 this.cameras.main.flash(300, 255, 50, 35);
```

For our game I'll keep the flash white.



Shake effect

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

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

```
1 // Shake for 300ms with an intensity of 0.02
2 this.cameras.main.shake(300, 0.02);
```

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

More about the camera

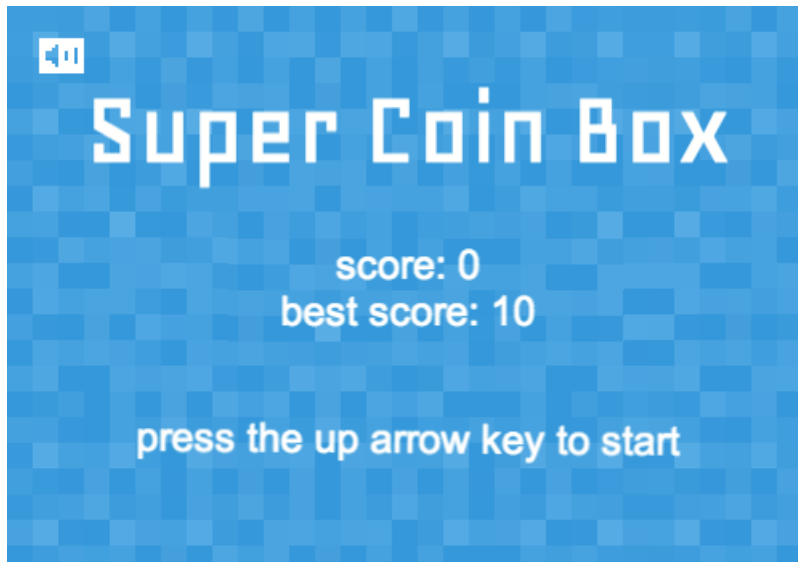
If needed, it's possible move the camera around by updating its x and y position:

```
1 this.cameras.main.x = 150;
2 this.cameras.main.y = 150;
```

6 - Improvements

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

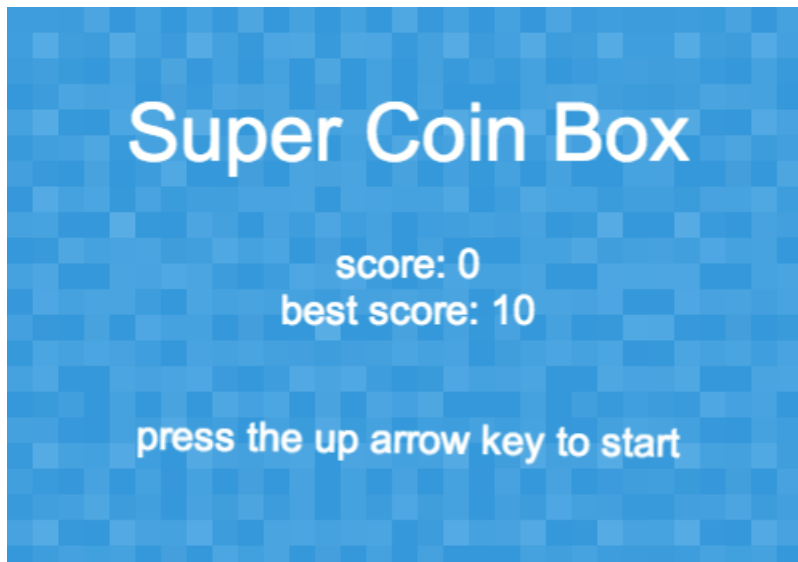
That's what we will cover in this chapter.



6.1 - Add best score

One great JavaScript feature is called “local storage”, it lets us store information on people’s browsers. This is really useful to store a short text, a number, a setting, etc.

In our case we are going to use local storage to keep track of the best score.



Store the best score

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

- `localStorage.setItem(keyName, keyValue)` that stores the 'keyValue' as 'keyName'. For example `localStorage.setItem("nickname", "Thomas")` would store 'Thomas' as the 'nickname'.
- `localStorage.getItem(keyName)` that returns the value stored for 'keyName'. For example `localStorage.getItem("nickname")` would return 'Thomas'.

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

```
1 // If 'bestScore' does not exist yet
2 if (localStorage.getItem('bestScore') === null) {
3   // Then set the best score to 0
4   localStorage.setItem('bestScore', 0);
5 }
6
7 // If the new score is higher than the best score
8 if (score > localStorage.getItem('bestScore')) {
9   // Then update the best score
10  localStorage.setItem('bestScore', score);
11 }
```

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:

```
1 let scoreText = 'score: ' + score;
2 let scoreLabel = this.add.text(250, 170, scoreText,
3   { font: '25px Arial', fill: '#fff' });
```

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

```
1 let scoreText = 'score: ' + score
2   + '\nbest score: ' + localStorage.getItem('bestScore');
3 let scoreLabel = this.add.text(250, 170, scoreText,
4   { font: '25px Arial', fill: '#fff', align: 'center' });
```

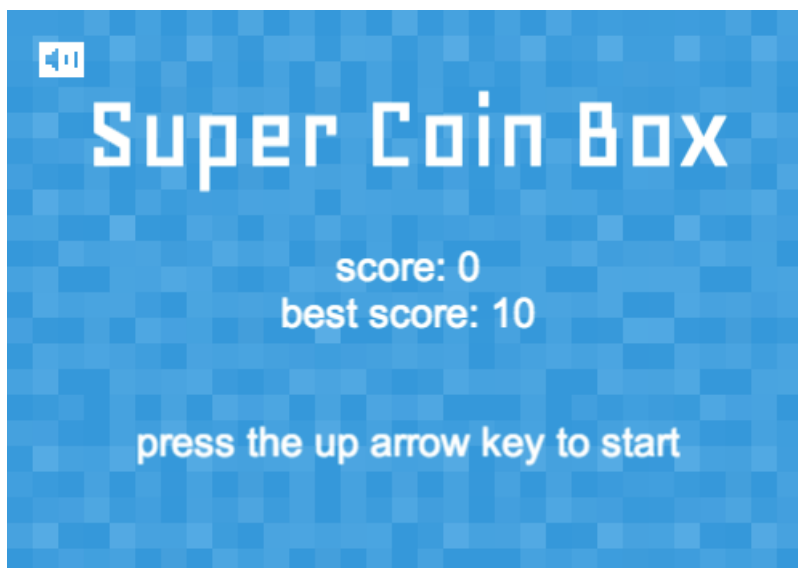
The `\n` adds a line break. And since the label is now on two lines, we add `align: 'center'` to center everything.

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

6.3 - 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 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:

```
1 <style>
2   @import url(//fonts.googleapis.com/css?family=Geo);
3 </style>
```

But that's not enough because we need to be sure that the font is loaded before the game starts. A trick to achieve that is to add a text (at least one character) on the index.html page that uses the new font:

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

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

```
1 <style>
2   @import url(//fonts.googleapis.com/css?family=Geo);
3
4   .hiddenText {
5     font-family: Geo;
6     visibility: hidden;
7     height: 0;
8   }
9 </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, that's how to change the game's name font:

```
1 // Before
2 let nameLabel = this.add.text(250, -50, 'Super Coin Box',
3   { font: '50px Arial', fill: '#fff' });
4
5 // After
6 let nameLabel = this.add.text(250, -50, 'Super Coin Box',
7   { font: '70px Geo', fill: '#fff' });
```

6.4 - Improve 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 progressively harder by adding more enemies.

Static frequency

Using a Phaser timer to create our enemies worked well so far, but now we need more control on what is happening exactly. So the first thing to do is to build our own custom timer that will create new enemies every 2.2 seconds just like before.

Replace the `this.time.addEvent({ ... })` line from the `play.js` file by this:

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

And we add this in the `update()` method of the `play.js` file:

```
1 // Get the current time in milliseconds
2 let now = Date.now();
3
4 // If the 'nextEnemy' time has passed
5 if (this.nextEnemy < now) {
6     // We add a new enemy
7     this.addEnemy();
8
9     // And we update 'nextEnemy' to have a new enemy in 2.2 seconds
10    this.nextEnemy = now + 2200;
11 }
```

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 have to answer these three questions:

1. How often should we create new enemies at the beginning of the game?
2. How fast can we create enemies while keeping the game playable?
3. When do we reach the maximum difficulty?

Here's what we could use:

1. Start difficulty: one new enemy every 4 seconds.
2. End difficulty: one enemy per second.
3. Score to reach end difficulty: 100 points.

With all this answers we can edit our timer in the `update()` method like this:

```
1  if (this.nextEnemy < now) {
2    // Define our three variables
3    let startDifficulty = 4000;
4    let endDifficulty = 1000;
5    let scoreToReachEndDifficulty = 100;
6
7    // Compute the progress, a number between 0 and 1
8    // 0 means we are just getting started with a score of 0
9    // 1 means we reached the scoreToReachEndDifficulty
10   let progress = Math.min(this.score / scoreToReachEndDifficulty, 1);
11
12   // Compute when the next enemy should appear
13   // It's a number between startDifficulty and endDifficulty
14   let delay = startDifficulty - (startDifficulty - endDifficulty) * progress;
15
16   // Create a new enemy and update the 'nextEnemy' time
17   this.addEnemy();
18   this.nextEnemy = now + delay;
19 }
```

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

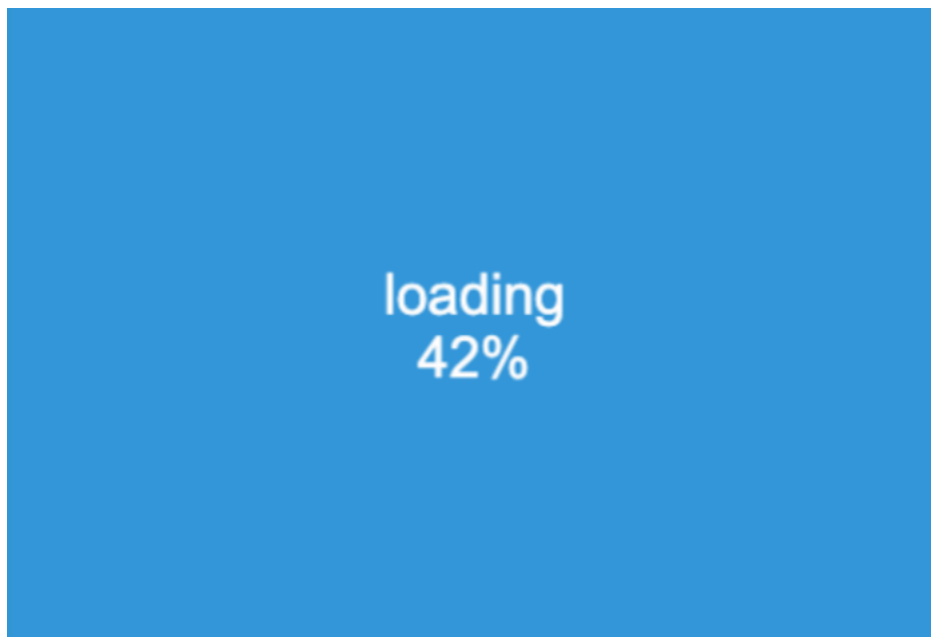
- If we have 0 point:
 - $\text{progress} = \min(0 / 100, 1) = \min(0, 1) = 0$
 - $\text{delay} = 4000 - (4000 - 1000) * 0 = 4000 - 0 = 4000$
 - That's one new enemy every 4 seconds.
- If we have 50 points:
 - $\text{progress} = \min(50 / 100, 1) = \min(0.5, 1) = 0.5$
 - $\text{delay} = 4000 - (4000 - 1000) * 0.5 = 4000 - 1500 = 2500$
 - That's one new enemy every 2.5 seconds.
- If we have 100 points:
 - $\text{progress} = \min(100 / 100, 1) = \min(1, 1) = 1$
 - $\text{delay} = 4000 - (4000 - 1000) * 1 = 4000 - 3000 = 1000$
 - That's one new enemy per second.
- If we have 200 points:
 - $\text{progress} = \min(200 / 100, 1) = \min(2, 1) = 1$
 - $\text{delay} = 4000 - (4000 - 1000) * 1 = 4000 - 3000 = 1000$
 - That's still one new enemy per second.

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

6.5 - Improve loading

We don't load many assets in this game, so the loading scene is barely visible. However for people on slow internet connection or for more complex games, showing a visual progress during the loading is a nice addition.

Let's see how to display a percentage that goes from 0% to 100% as the game loads.



Edit label

First, edit the existing label in the load.js file:

```
1 // Before
2 let loadLabel = this.add.text(250, 170, 'loading',
3   { font: '30px Arial', fill: '#fff' });
4 loadLabel.setOrigin(0.5, 0.5);
5
6 // After
7 this.loadLabel = this.add.text(250, 170, 'loading\n0%',
8   { font: '30px Arial', fill: '#fff', align: 'center' });
9 this.loadLabel.setOrigin(0.5, 0.5);
```

Progress event

Next, we want to be notified every time a new file is loaded to update our label. For this we should use events! So add this in the `create()` method:

```
1 // Call 'this.progress' every time a new file is loaded
2 this.load.on('progress', this.progress, this);
```

Finally, create the new `progress()` method. It has one parameter: a number between 0 and 1 representing the loading progress.

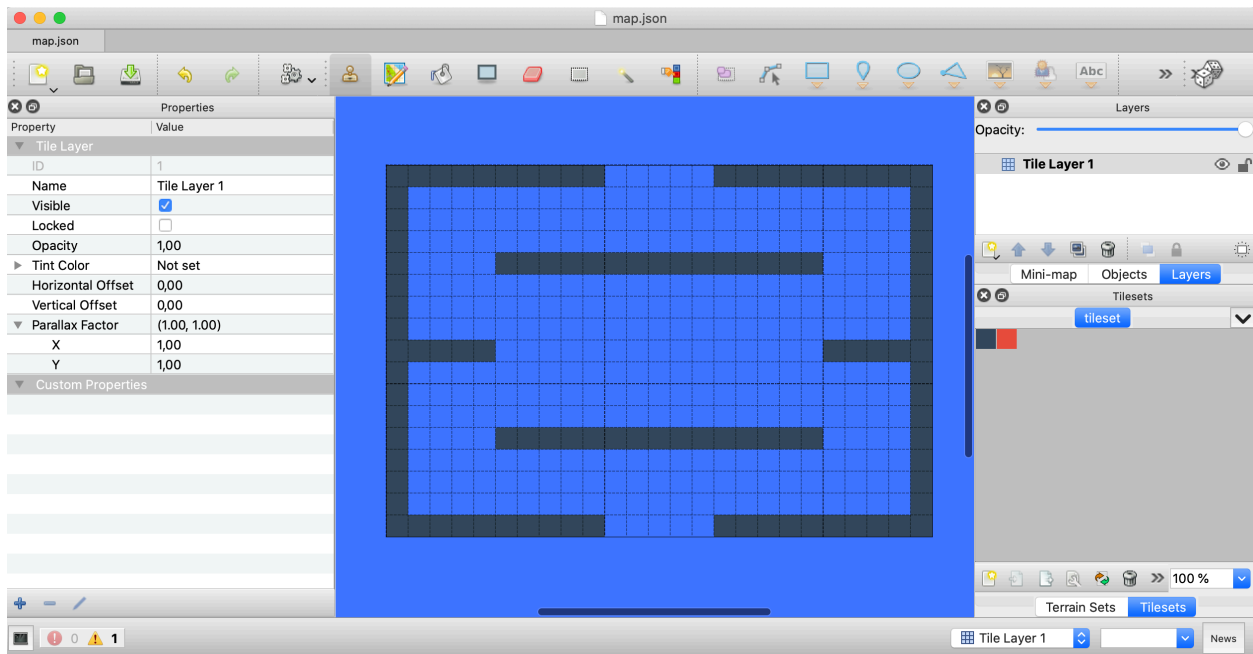
```
1 progress(value) {
2   // Compute the percentage, between 0 and 100
3   let percentage = Math.round(value * 100) + '%';
4
5   // Update the label
6   this.loadLabel.setText('loading\n' + percentage);
7 }
```

And that's it! But keep in mind that because our game is lightweight, you might not be able to see the loading scene.

7 - Tilemaps

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

Let's see how to use the software Tiled to draw our world on the screen and then display it in the game.



7.1 - Create assets

Here are some basic definitions to make sure we are 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: a two dimensional array of tiles that represents a level.

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

The tileset

Our tileset is 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 let's add a second tile: a red square.

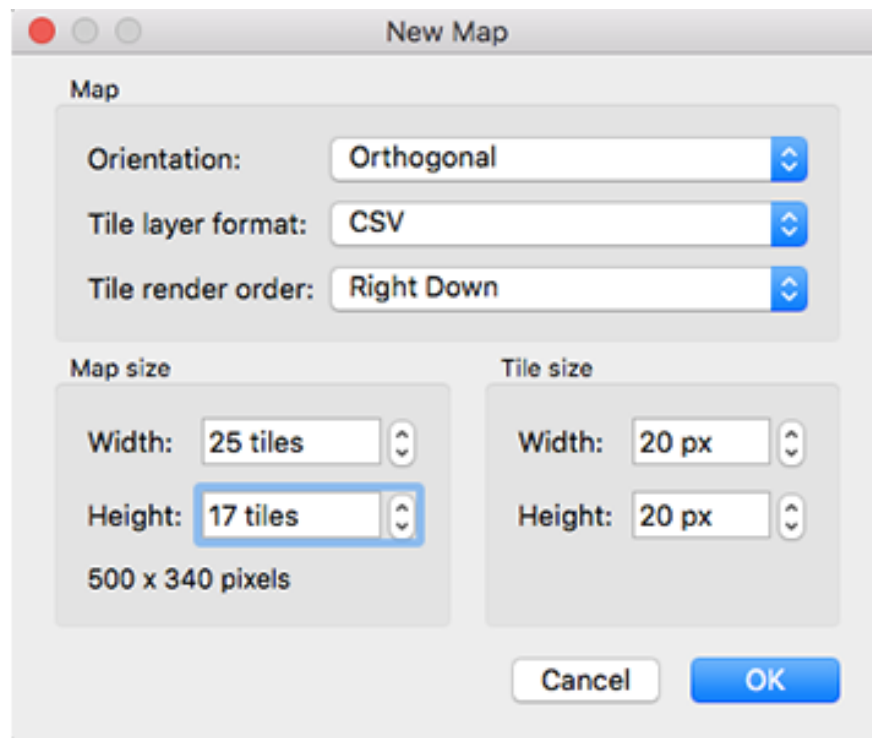
It looks like this:



Tiled

There is 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](#).

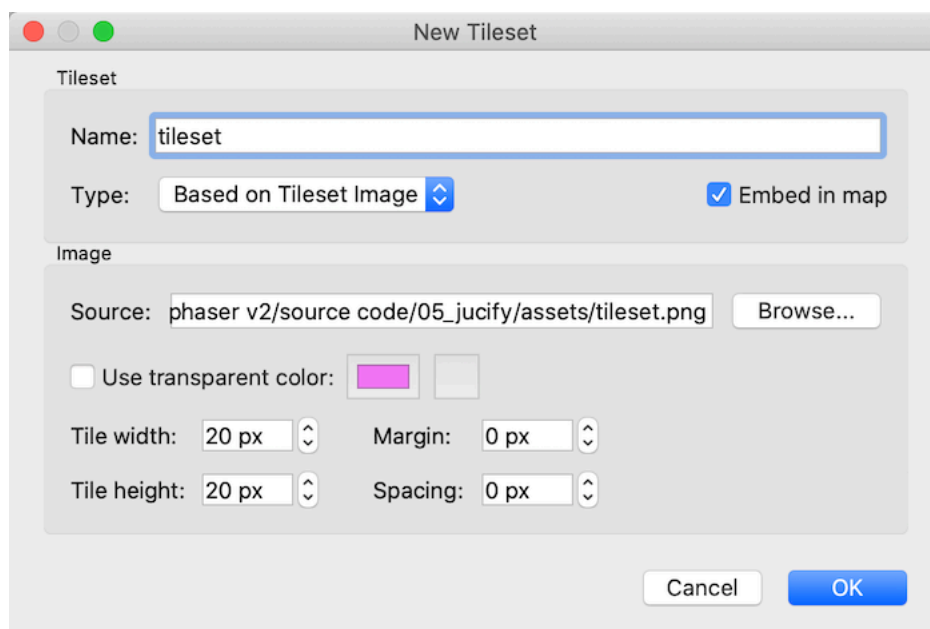
Open the Tiled app, click on "file > new > new map" and fill the form like this:



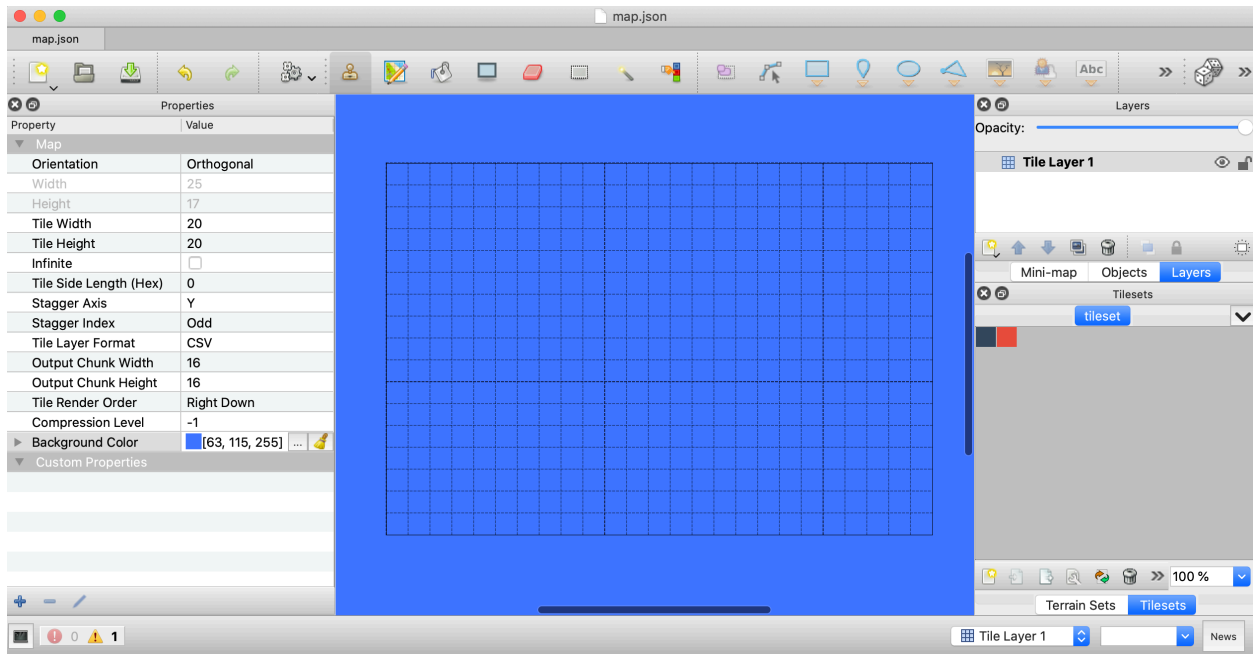
When saving the file, name it “map” and make sure to select the “JSON” format.

Then do “map > map properties”, and in the sidebar set a blue background color to the map.

Next click on “file > new > new tileset” and fill the form like this:

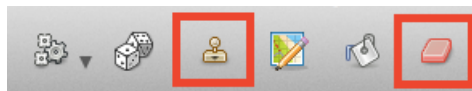


Once done, you should see this on your screen:



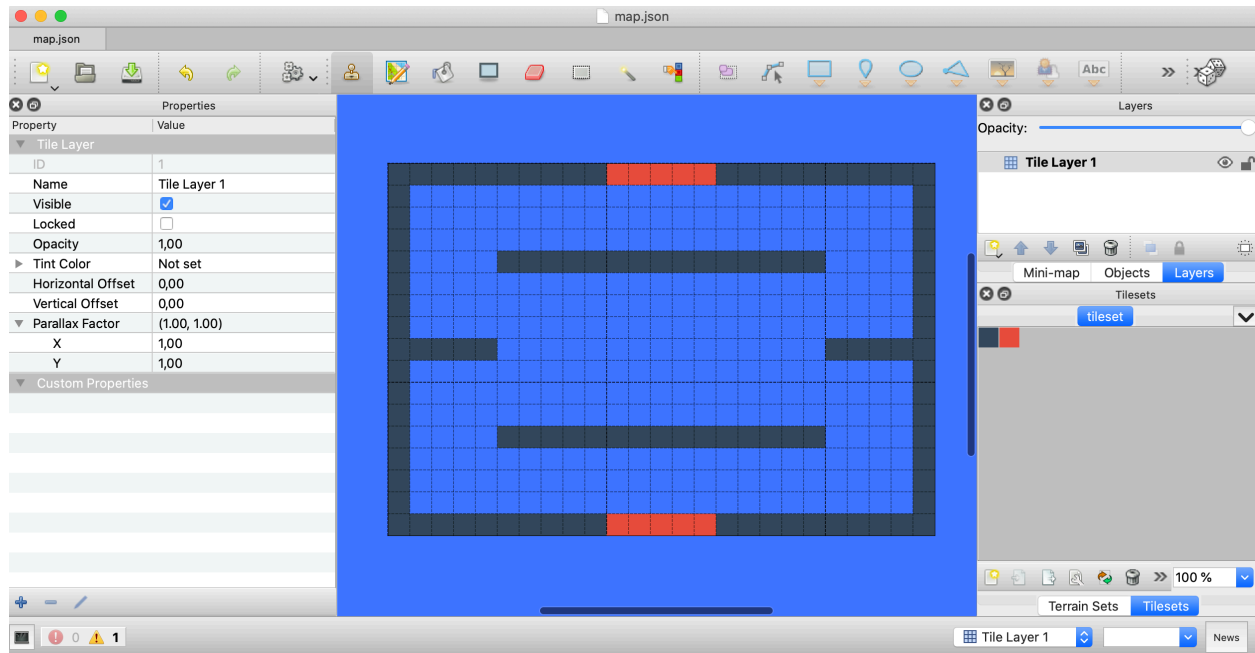
The tilemap

Our empty tilemap is now properly set up, the next step is to design the level. For this we need two 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.



Now simply draw the level using the tileset in the bottom right corner as your “color picker”. Make sure to leave some holes for the enemies at the top and bottom.

Here’s the map I created. It’s the same we used so far with some extra red tiles to fill the holes (this is just a cosmetic change).



When finished, save the file with “file > save”.

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 two new assets in the load.js file like this:

```
1 this.load.image('tileset', 'assets/tileset.png');
2 this.load.tilemapTiledJSON('map', 'assets/map.json');
```

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

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

Display the tilemap

And now we completely change the createWorld() method to use our new assets:

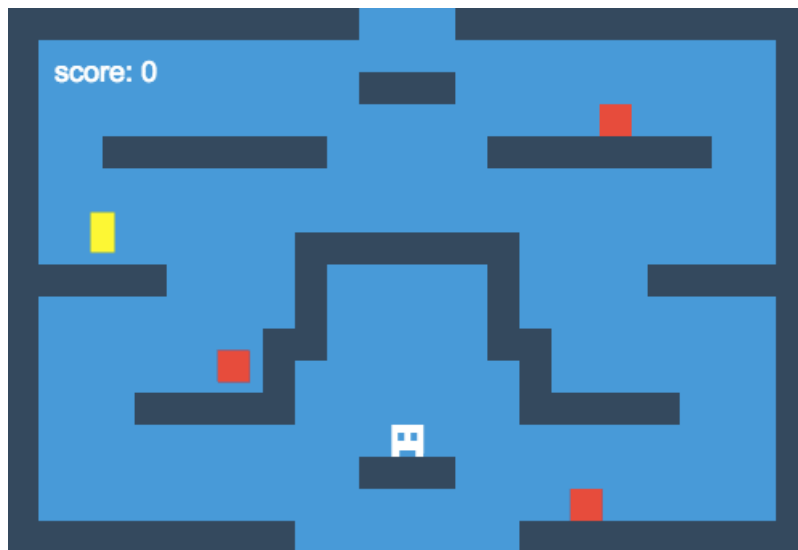
```
1 createWorld() {
2     // Create the tilemap
3     let map = this.add.tilemap('map');
4
5     // Add the tileset to the map
6     // First parameter: the name of the tileset in Tiled
7     // Second parameter: the name of the tileset in 'preload()'
8     let tileset = map.addTilesetImage('tileset', 'tileset');
9
10    // Create the layer by specifying the name of the Tiled layer
```

```
11  this.walls = map.createStaticLayer('Tile Layer 1', tileset);
12
13  // Enable collisions for the first tile (the blue wall)
14  this.walls.setCollision(1);
15 }
```

If we try the game, there will be no visible differences except that:

- The code is a lot cleaner, the `createWorld()` method is now 4 line-long instead of 11.
- And 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:

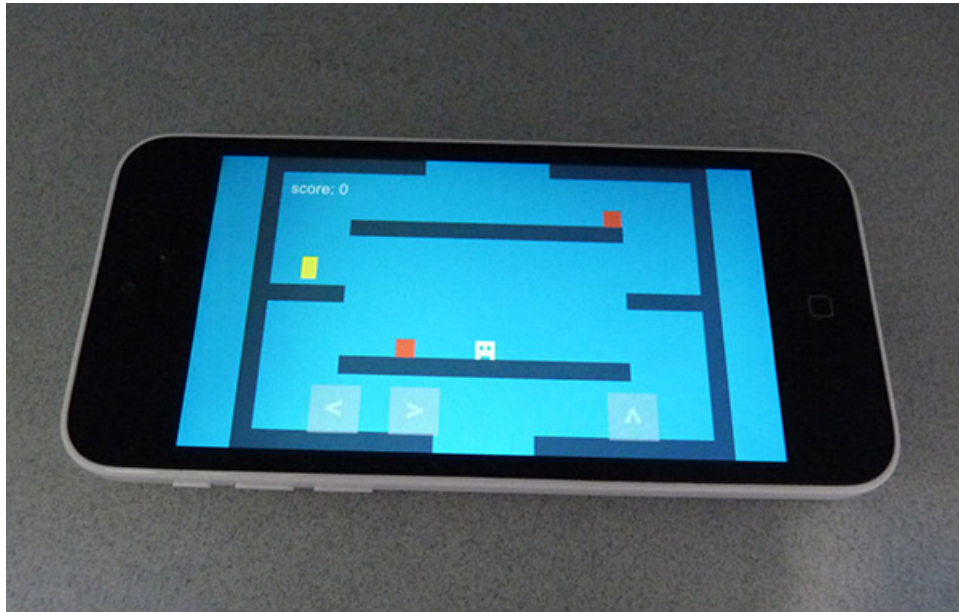
- Add objects in the game.
- Dynamically edit the tilemap.
- Add properties to tiles.
- Etc.

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

8 - Mobile friendly

A great thing about JavaScript games is that they can work basically everywhere. That includes phones and tablets.

In this chapter we will see how to make our game mobile friendly. That includes: scaling the game, adding touch inputs, handling device rotation, and more.



8.1 - Test

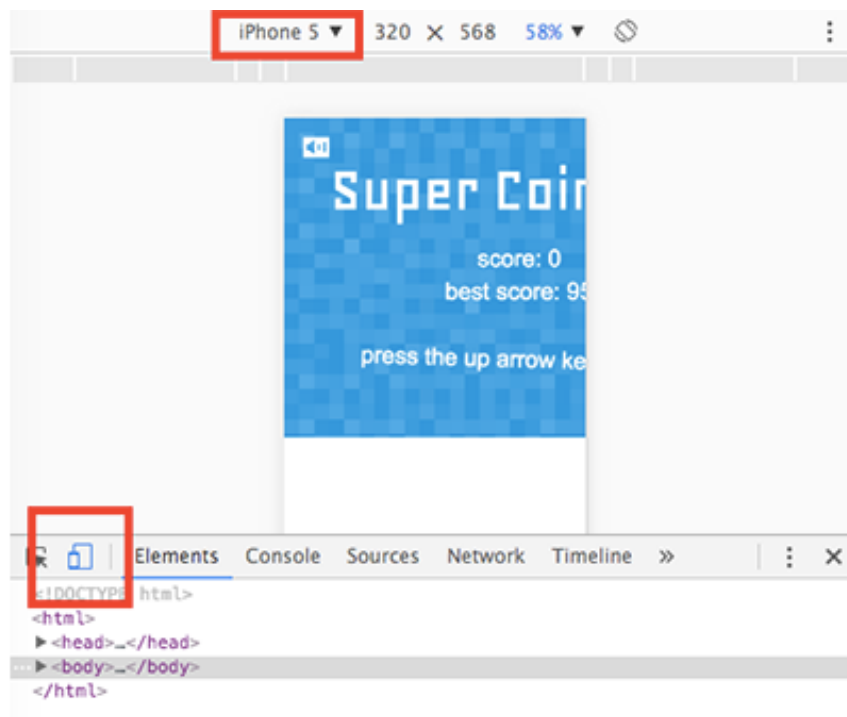
Before we start coding anything we should know how to test our mobile friendly game. There are two different ways to do this: on a computer and on a mobile device.

Just keep in mind that we haven't made the game mobile friendly yet. So testing it right now won't 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.



If needed you can press on the small icon in the top right corner to rotate the screen.

That's a really handy solution but it 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.example.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 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 I recommend using both methods.

8.2 - Resize game

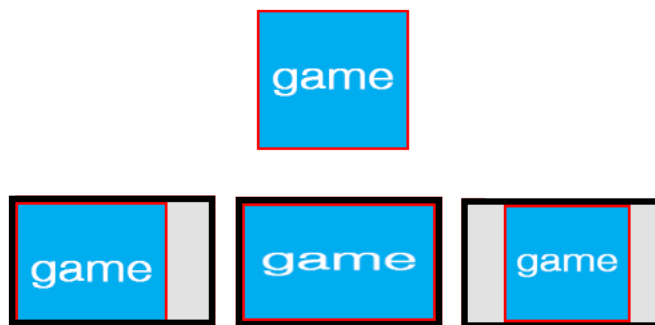
The main issue when making games for mobile is managing all the screen sizes. We will see below how to handle that.

Types of scaling

There are three main ways to scale a game with Phaser:

- No scale. The game doesn't change its size (that's the default behavior).
- Resize. The game is stretched to fill every pixel of the screen.
- Fit. The game is scaled to fit the screen without changing its aspect ratio.

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



The most common way to do the scaling is with "fit", and that's what we will cover in this part.

Edit game file

To tell Phaser to use the "fit" scale, we only need to edit the game.js file like this:

```
1  let game = new Phaser.Game({
2    width: 500,
3    height: 340,
4    backgroundColor: '#3498db',
5    physics: { default: 'arcade' },
6    parent: 'game',
7
8    // New code below
9    scale: {
10      mode: Phaser.Scale.FIT,
11    },
12  });
```

And that's pretty much it! But we can improve this in two ways.

First, we can ask Phaser to always center our game on the screen.

```
1  scale: {
2    mode: Phaser.Scale.FIT,
3
4    // New code below
5    autoCenter: Phaser.Scale.CENTER_BOTH,
6  },
```

Second, we can 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.

```
1  scale: {
2    mode: Phaser.Scale.FIT,
3    autoCenter: Phaser.Scale.CENTER_BOTH,
4
5    // New code below
6    min: {
7      width: 250,
8      height: 170,
9    },
```

```
10    max: {  
11        width: 1000,  
12        height: 680,  
13    },  
14 },
```

Edit index file

We also should add a 2 new CSS rules to the index.html file.

One to remove every margin and padding, to make sure there are no gaps between the game and the borders of the screen.

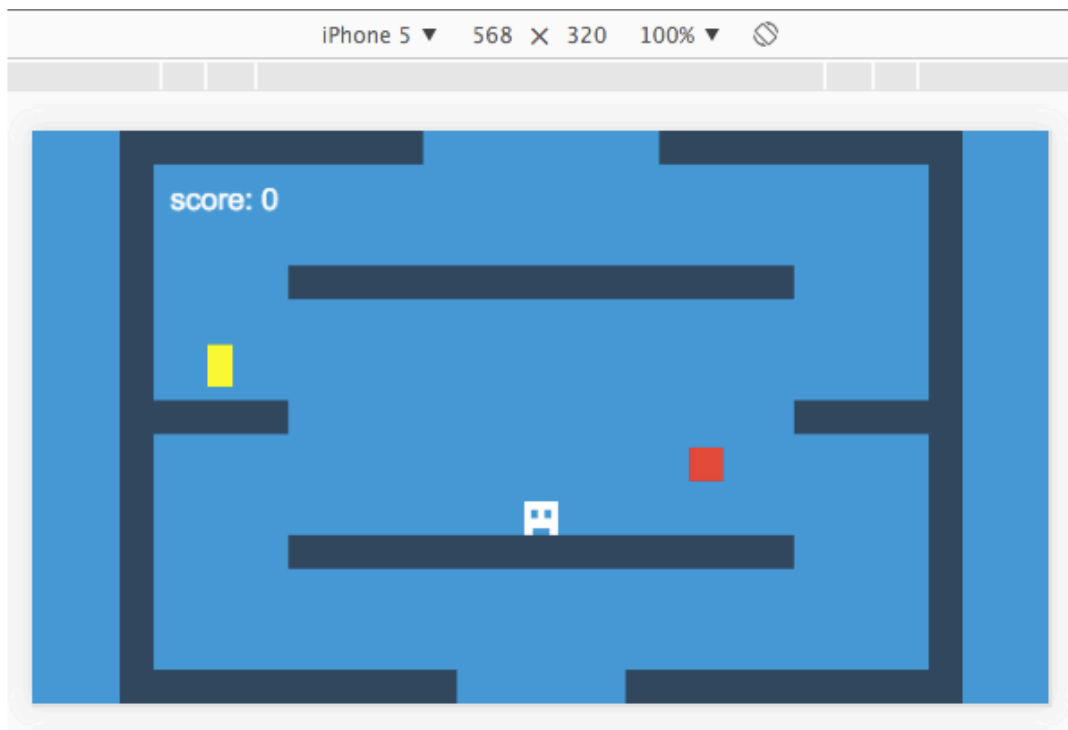
```
1 * {  
2     margin: 0;  
3     padding: 0;  
4 }
```

And another one to add a background color to the page, to avoid seeing white borders.

```
1 body {  
2     background-color: #3498db;  
3 }
```

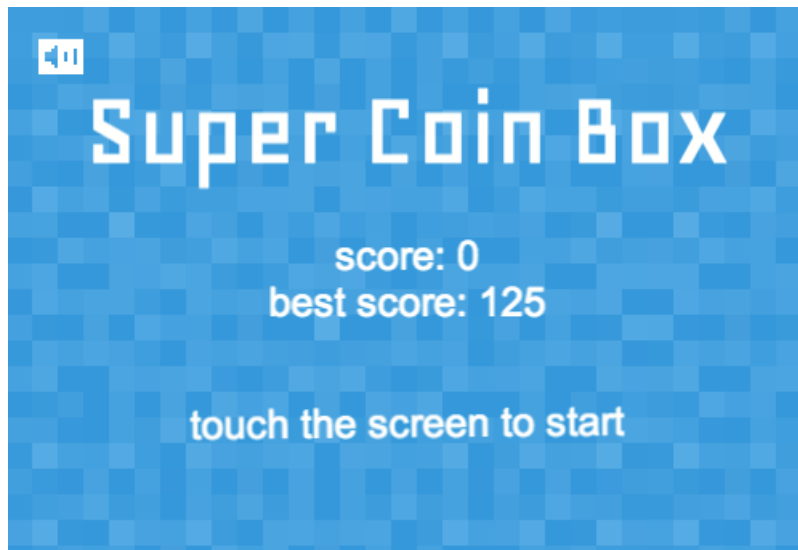
Result

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



8.3 - Add touch inputs

Now the game looks great on mobile devices, but we are stuck in the menu scene because we can't press the up arrow key on a touch screen. So let's make it possible to start the game by simply tapping the screen.



Change the label

First we should make it clear to the users that they can touch the screen to start playing. We can use `this.sys.game.device.os.desktop` to know whether or not we are on a mobile device.

So edit the `startLabel1` of the `menu.js` file like this:

```

1  // Set the correct text depending on the device
2  let startText;
3  if (this.sys.game.device.os.desktop) {
4      startText = 'press the up arrow key to start';
5  } else {
6      startText = 'touch the screen to start';
7  }
8
9  // Display the label on the screen
10 let startLabel = this.add.text(250, 260, startText,
11     { font: '25px Arial', fill: '#fff' });

```

Touch event

So far we used this to start the game when the up arrow key is pressed:

```

1  if (this.upKey.isDown) {
2      this.scene.start('play');
3  }

```

For touch events we update this code like this:

```

1  if (this.upKey.isDown || (!this.sys.game.device.os.desktop
2      && this.input.activePointer.isDown)) {
3      this.scene.start('play');
4  }

```

For your information the `activePointer.isDown` can either be a finger 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 `update()` method:

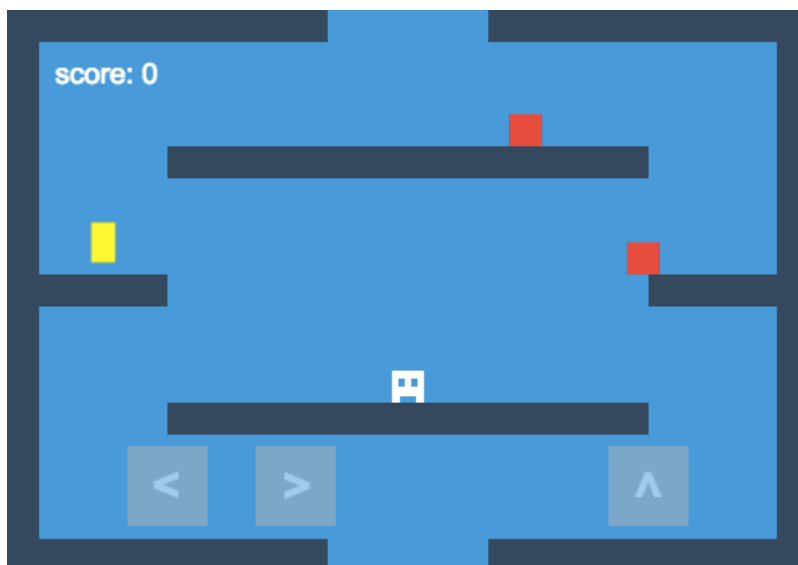
```
1  // If we tap near the top of the game on mobile
2  if (!this.sys.game.device.os.desktop
3      && this.input.activePointer.y < 60) {
4      // It probably means we want to mute the game
5      // So we don't start the play scene
6      return;
7  }
```


8.4 - Add touch buttons

The last step to actually play the game is to add 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 a few buttons on the screen that we can press.

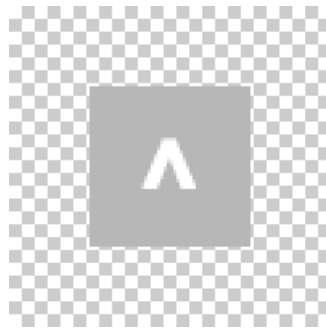
We will focus on the last option.



Load the buttons

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

```
1 this.load.image('jumpButton', 'assets/jumpButton.png');  
2 this.load.image('rightButton', 'assets/rightButton.png');  
3 this.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

For each button we need to:

- Create a sprite with the button image at the correct position.
- Set the sprite as interactive, to be able to listen to events when the user interacts with it.
- And make the sprite a little transparent to not hide the game behind it.

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

```
1  addMobileInputs() {
2    let jumpButton = this.add.sprite(400, 290, 'jumpButton');
3    jumpButton.setInteractive();
4    jumpButton.alpha = 0.5;
5
6    let leftButton = this.add.sprite(100, 290, 'leftButton');
7    leftButton.setInteractive();
8    leftButton.alpha = 0.5;
9
10   let rightButton = this.add.sprite(180, 290, 'rightButton');
11   rightButton.setInteractive();
12   rightButton.alpha = 0.5;
13 }
```

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

```
1  if (!this.sys.game.device.os.desktop) {  
2      this.addMobileInputs();  
3  }
```

Jumping

Let's take care of the jump button by adding this to the `addMobileInputs()` method:

```
1  // Call 'jumpPlayer' when 'jumpButton' is pressed  
2  jumpButton.on('pointerdown', this.jumpPlayer, this);
```

Now we create the new `jumpPlayer()` method:

```
1  jumpPlayer() {  
2      // If the player is touching the ground  
3      if (this.player.body.onFloor()) {  
4          // Jump with sound  
5          this.player.body.velocity.y = -320;  
6          this.jumpSound.play();  
7      }  
8  }
```

And it should work. But before we move on we should also edit the `movePlayer()` method to call `jumpPlayer()`. This way we avoid duplicating the jump code.

```
1  // Replace all of this
2  if (this.arrow.up.isDown && this.player.body.onFloor()) {
3      this.player.body.velocity.y = -320;
4      this.jumpSound.play();
5  }
6
7
8  // By this
9  if (this.arrow.up.isDown) {
10     this.jumpPlayer();
11 }
```

Now each time a key or the jump button is pressed, `jumpPlayer()` will be called.

Moving left and right - idea

Previously we called the `jumpPlayer()` method just once when the button was pressed. For the player's left and right movements we want to be able to keep pressing the buttons to keep moving, so using the `pointerdown` event won't be enough.

Here's how it will work for the `rightButton`:

- We define a new variable `moveRight` set to `false` by default.
- If the event `pointerover` is triggered, then we set `moveRight` to `true`.
- If the event `pointerout` is triggered, then we set `moveRight` to `false`.

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

And the exact same idea applies to the `leftButton`.

Moving left and right - code

Here's the updated `addMobileInputs()` method:

```
1  addMobileInputs() {
2    this.moveLeft = false;
3    this.moveRight = false;
4
5    let jumpButton = this.add.sprite(400, 290, 'jumpButton');
6    jumpButton.setInteractive();
7    jumpButton.on('pointerdown', this.jumpPlayer, this);
8    jumpButton.alpha = 0.5;
9
10   let leftButton = this.add.sprite(100, 290, 'leftButton');
11   leftButton.setInteractive();
12   leftButton.on('pointerover', () => this.moveLeft = true, this);
13   leftButton.on('pointerout', () => this.moveLeft = false, this);
14   leftButton.alpha = 0.5;
15
16   let rightButton = this.add.sprite(180, 290, 'rightButton');
17   rightButton.setInteractive();
18   rightButton.on('pointerover', () => this.moveRight = true, this);
19   rightButton.on('pointerout', () => this.moveRight = false, this);
20   rightButton.alpha = 0.5;
21 }
```

You can see that:

- We defined two new variables: `moveRight` and `moveLeft`.
- For each right and left button we added two events: `pointerover` and `pointerout`.
- Each time the user interacts with one of the buttons we update the value of `moveRight` or `moveLeft` 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 two `if` conditions of the `movePlayer()` method:

```
1 movePlayer() {
2   // New condition to check if we are moving left
3   if (this.arrow.left.isDown || this.moveLeft) {
4     // No changes here
5     this.player.body.velocity.x = -200;
6     this.player.anims.play('left', true);
7   }
8
9   // New condition to check if we are moving right
10  else if (this.arrow.right.isDown || this.moveRight) {
11    // No changes here
12    this.player.body.velocity.x = 200;
13    this.player.anims.play('right', true);
14  }
15
16  // Do not change the rest of the function
17 }
```

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

Multitouch fix

If you try the game, you might notice something annoying: it's not possible to jump while moving. That's because Phaser only handles one pointer by default, but in this case we need two.

To fix this, simply add this in the `create()` method of the `play.js` file:

```
1 // Tell Phaser to add one extra pointer (so two in total)
2 this.input.addPointer(1);
```

8.5 - Handle orientations

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 need a way to know when the device gets rotated. And when it does, we should 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 method that does just that, to add in the play.js file:

```
1 orientationChange() {  
2     if (this.scale.orientation === Phaser.Scale.PORTRAIT) {  
3         this.rotateLabel.setText(' rotate your device in landscape ');  
4         this.scene.pause();  
5     } else if (this.scale.orientation === Phaser.Scale.LANDSCAPE) {  
6         this.rotateLabel.setText('');  
7         this.scene.resume();  
8     }  
9 }
```

This code contains some interesting things:

- `this.scale.orientation` to know the orientation of the device.
- `this.scene.pause()` to stop everything in the scene.
- `this.scene.resume()` to resume the scene.

Now we need to call the new `orientationChange()` method in the `create()` of the `play.js` file:

```
1  if (!this.sys.game.device.os.desktop) {  
2    // Call 'addMobileInputs' just like before (no changes)  
3    this.addMobileInputs();  
4  
5    // Create an empty label to write the error message if needed  
6    this.rotateLabel = this.add.text(250, 170, '',  
7      { font: '30px Arial', fill: '#fff', backgroundColor: '#000' });  
8    this.rotateLabel.setOrigin(0.5, 0.5);  
9  
10   // Call 'orientationChange' when the device is rotated  
11   this.scale.on('orientationchange', this.orientationChange, this);  
12  
13   // Call the method at least once  
14   this.orientationChange();  
15 }
```

The Phaser function `this.scale.on()` will automatically call our new method whenever the device is rotated. But if the game starts in the wrong orientation our method won't be called. That's why we call `orientationChange()` at the end to make sure it is executed at least once.

Result

Here's the game in both portrait and landscape to see the difference.



9 - Next steps

Congratulations, you've made a full featured game with Phaser! And remember that we started all of this with an empty blue screen.

Here are the main Phaser features we covered in this book:

- Chapter 2: setting up the project.
- Chapter 3: sprites, labels, groups, physics.
- Chapter 4: scene management.
- Chapter 5: sounds, animations, tweens, particles, camera.
- Chapter 6: local storage, mute button, custom fonts, increasing difficulty, loading progress.
- Chapter 7: tilesets, tilemaps, Tiled.
- Chapter 8: scaling, touch inputs, touch buttons, device orientation.

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



9.1 - Improve the game

The first thing you could do is to improve the game we made together. Here are some ideas to help you get started.

Customization

Customizing the game is easy. Change the sprites and sounds or tweak the hard coded 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 could add new ones. For example: rotate the coin indefinitely, animate the menu when the user beats his best score, grow the score label 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.

```
1 // in the 'create' method
2 define a life variable
3 add a life label on the screen
4
5 // in the 'playerDie' method
6 if (lives > 0)
7     decrement the number of lives
8     update the life label
9     kill all the enemies
10
```

```
11 else
12     start the menu scene
```

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.

```
1 // in the 'jumpPlayer' method
2 if (the player is touching the ground)
3     jump with an initial velocity of -200
4     set a startToJumpAt variable to Date.now()
5
6 else if (a jump started less than 200ms ago)
7     keep jumping with a velocity of -200
```

Enemy types

The current red enemies all have the same size and move at the same speed. To make that more interesting you could create two types of enemies: smaller & faster ones, and bigger & slower ones.

```
1 // in the 'addEnemy' method
2 if (true 50% of the time)
3     scale the enemy up
4     decrease its velocity
5
6 else
7     scale the enemy down
8     increase its velocity
```

9.2 - Make new games

A bigger challenge would be to make your own game from scratch. If you've never done that before it may sound a little scary.

Here are three simple tips to help you 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 game with quests, dungeons, puzzles, and bosses!". But most of the time these projects are put on shelves because they are too long to make.

Instead, start making something simple like Pong, Breakout, or Space Invaders. And only when you become comfortable making these types of games you can move to 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 allows you to play the game early during development and see it improve over time. This is key to staying motivated.

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 check [opengameart](#), they have tons of sprites available for free. For sounds try [bfxr](#), a tool to create nice sound effects by just pressing some buttons.

9.3 - Conclusion

This book is now over, I hope that you enjoyed reading it! Now you should have a real game to play with and enough knowledge to make your own 2D games.

If you have the time, I have three asks for you:

- Let me know if you spotted a mistake or saw something that can be improved.
- Please leave a review for this book (there should be a link to do so in your email receipt).
- And don't forget to send me a link when your first Phaser game is released!

My email: thomas.palef@gmail.com

Thanks for reading!