HTML5 Cross Platform Game Development Using Phaser 3

Learn to build HTML5 games using Phaser 3 and other free software

Emanuele Feronato

HTML5 Cross Platform Game Development Using Phaser 3

Let me take you by hand from the basics of JavaScript programming and show you how to build complete cross platform HTML5 games using Phaser 3 framework and other free software.

Written by Emanuele Feronato

A little preface

I won't annoy you with boring introductions which everybody skips, I just want you to know why I am writing this book.

When I am about to start learning a new language or framework, and this happened a lot of times in more than 30 years – and counting! - spent studying and working on programming languages, I always desperately look for a book able to guide me from the bare bones to the creation of an actual, complete project.

Unfortunately, most beginner guides just cover a brief overview, leaving it up to you to continue learning and experimenting, and advanced guides assume you already are an expert user.

What I wanted to do with this book is to take you by hand even if you are an absolute beginner and give you everything you need to build complete cross platform games from scratch.

For those of you having some acquired programming skills, I will guide you deep enough into the mechanics of game development.

I did my best for you to like this book and find it useful.

And most of all I want to say **thank you**.

By buying this book you allow me to concentrate on book and course development, which is something I really like.

Let's start.

What is a cross-platform game and why should I make cross-platform games?

With the great interest in mobile games, capable of running on modern portable devices such as smart phones and tablets, there's a lot of talking about “cross-platform” term these days.

Although we are talking about modern devices, the cross-platform concept comes from an older computer age, before smart phones and tablets, probably before any kind of portable device smaller than a mid-sized suitcase.

In its original context, cross-platform is an attribute conferred to computer software or computing methods and concepts that are implemented and inter-operate on multiple computer platforms.

Such software and methods are also called “platform independent”.

To tell you the short story, a cross-platform software will run on any platform without special adaptation, or with a minimum special adaptation.

A good example of a cross-platform language is Java: a compiled Java program runs on all platforms under Java Virtual Machine, which you can find in all major operating systems, including Windows, Mac OS and Linux.

Stop with the boring theory and back to our days. A cross-platform game, the kind of game we are going to build, is a game able to run on various devices, such as smart phones and tablets – but also on desktop machines – each one with its own resolution and screen aspect ratio, giving players the same experience no matter the device used to play.

And here comes the main question: why should I make cross-platform games?

Listen to my story.

When HTML5 mobile gaming started to become popular, I had an iPad2 tablet and built a game which fitted perfectly in its resolution.

I was very happy with that game, it was a word game and looked really great on my brand new tablet, covering the entire screen with sprites and colors.

Once I completed the game, I started showing it to various sponsors. I already had a list of sponsor emails collected during Flash gaming era, so I was expecting a lot of offers.

Actually, I got offers, but most of them said something like "Hey, I like the game, but unfortunately it does not look that good on my iPhone".

"Don't worry", I said, "you'll get the game optimized for both iPad and iPhone".

After some extra work, I was told the game didn't look good on the Galaxy Note.

A few fixes and tweaks later, it happened the game didn't look good on the Samsung S4.

When the game was finally optimized for all required devices, it didn't look good anymore on my iPad.

You can imagine the rest of this story. I found myself almost rewriting the game with a series of exceptions, trying to make it look good on any device.

This is why you should make a cross-platform game: to code once and rule them all, focusing on code and game development while letting a framework do the dirty work for you.

What is HTML5?

HTML5 is the latest version of HTML, or Hypertext Markup Language, the code web pages are mostly built with.

We can basically split each web page in three kinds of code: HTML, which provides the structure; Cascading Style Sheets (CSS), which take care of presentation; and JavaScript, which makes things happen.

Unlike other tools like Flash or Java, HTML5 works without requiring any additional software like browser plugins or extensions, and it's capable of doing everything from animation to apps, can play music or movies, and can also be used to build incredibly complicated applications – or in this case, games – that run on almost any browser.

Moreover, HTML5 isn't proprietary, so you don't need to pay fees, subscriptions or royalties to use it. It's also cross-platform, which means it doesn't care whether you're using a tablet or a smart phone, or a laptop, or a smart TV: if your device browser supports HTML5 – and most of them do – your app will work flawlessly.

At this time you may think you have to learn to code HTML5 to build a HTML5 game.

That's not true. Actually, you won't be writing more than a couple of lines of HTML5, since your game will be written in JavaScript.

JavaScript is a scripting language that allows you to create dynamically updating content, control multimedia, animate images, and pretty much everything you need to build a game, especially if combined with JavaScript frameworks – like Phaser – created with game development in mind.

What is Phaser?

Phaser is a free HTML5 game framework which aims to help developers make powerful, cross browser HTML5 games really quickly using JavaScript.

JavaScript, being a familiar and intuitive language, is one of the most common languages so even if you didn't already developed JavaScript applications you will find a lot of books and tutorials around the web to get you started.

Anyway, you don't need anything else than this book to build your first complete game, so let's start having some fun.

Ok, I am lost. HTML5, CSS, JavaScript, Phaser… too much stuff

Actually, it's way easier than you may think. Your final game will be a HTML5 game, but actually the only HTML5 element you will be using in your game is the `<canvas>` element, which is only a container for the graphics.

You will use JavaScript to draw and update the graphics, as well as to define the game design.

Phaser is the JavaScript framework which you will use to handle sprites, sound effects, graphic effects, explosions, screen updates and basically everything capable of making your game look nice, move smooth and sound good.

CSS will be used for minor adjustments on the page which will host your game, mostly to define where `<canvas>` element should be placed.

Can I build games like GTA with Phaser?

To tell the truth, I don't know. Anyway, you shouldn't even think about it.

The first rule in game development is: build a game you are able to finish.

And when I say “finish a game” I don't mean seeing the congratulations screen in GTA because you completed all missions, I mean coding a game from scratch until the end.

According to this concept, if you are a one-man studio or a small indie studio, you should pick a genre of game you are able to code from the beginning until the end, in a reasonable amount of time.

Start with small, casual games and remember casual games is a genre gaining more and more popularity.

Quitting the development of a game saying “ok, I was about to finish it, I'll just start another game because I have a better idea” is not finishing to develop a game. It's just quitting it.

Your game should be complete, tested enough to avoid bugs, and with the code written and organized in a way you can easily update it, should you want to add content later on.

Ready? Let's turn our computer into a game development workstation at no cost thanks to Phaser and some other free software.

Choosing a free text editor

In order to start coding games, you'll first need a software to write code.

There are a lot of free offers:

Brackets (http://brackets.io/) is an open-source editor written in HTML, CSS, and JavaScript with a primary focus on web development and available for OS X, Windows, and Linux machines.

Atom (https://atom.io/) is the editor I am currently using for game development, open source and available for OS X, Windows, and Linux machines. It has everything I need, no more than the stuff I need and a nice selection of themes.

Sublime Text (https://www.sublimetext.com/) is the editor the Phaser team used to build the framework and all projects. Available for OS X, Windows, and Linux machines, although it's not free it's the one Phaser team use, so you may want at least to try the evaluation version.

Visual Studio Code (https://code.visualstudio.com/) is Microsoft's take on code editors available for Windows, OS X and Linux.

You are free to use one of the editors listed above, or one of the dozens editors you can find around the web. In my opinion you should choose one capable of highlighting JavaScript with some kind of auto complete.

Other free software you may need

Games basically are a collection of images and sounds which are moved and played accordingly to player actions and scripting logic, so during the development of the game you will also need to create and edit both images and sounds.

Audacity (https://www.audacityteam.org/) is a great free software to work with sounds available for OS X, Windows and Linux.

oceanaudio (http://www.ocenaudio.com/) is a cross-platform, easy to use, fast and functional audio editor. It is the ideal software for people who need to edit and analyze audio files without complications.

GIMP (http://www.gimp.org/) is a powerful image editor available for OS X, Windows and Linux..

Krita (https://krita.org/en/) is a great open source painting software available for OS X, Windows and Linux. You may find it really useful in the creation of textures, concept art, illustrations and backgrounds.

Choosing a free web server

To test your Phaser games, and more in general to test most web applications, you will need to install a web server on your computer to override browser security limits when running your project locally.

WAMP (http://www.wampserver.com/) is a complete Windows web development environment which allows you to create web applications with Apache2, PHP and a MySQL database.

MAMP (https://www.mamp.info/en/) runs on Mac and Windows, works pretty much the same way WAMP does, and also features a paid PRO version with more options.

Fenix Web Server (http://fenixwebserver.com/) is the web server I currently use because it's really simple, with no extra stuff, and open source. It's available for both Windows and Mac.

If you prefer, if you have a FTP space you can test your projects directly online by uploading and calling them directly from the web.

In this case, you won't need a web server installed on your computer, but I highly recommend using a local web server instead.

Most FTP spaces require a paid account, and you can only use them when you have an internet connection available.

Setting up a web server is a matter of minutes, refer to their official documentation to install and run them on your computer.

REALLY choosing a web server, rather than closing the book

I know at this time most of you may think “come on, it's just JavaScript, what's this server stuff, I quit!”.

This is the same thing I said when I first had to install and configure a web server just to run a JavaScript page.

Let me explain why you should really choose a web server, rather than quit reading: browsers do not simply allow you to properly display web pages and HTML5 games. They also take care of your security.

When you load a page locally in your browser, you won't have problems as long as it's just a static HTML web page.

But when you launch more complex scripts which load and handle resources from your hard disk such as images, audio files and every other kind of data, to prevent malicious scripts to access to virtually any file on your computer, browsers have a series of security measures which stop files to be accessed and – unfortunately but necessarily – this causes your games not to work.

The most frequent error you will get if you run a Phaser game directly in your browser is something like “Cross origin requests are only supported for protocol schemes: http, data, chrome, chrome-extension, https.”

With a web server, browsers will know they are running in a small, safe environment where only some files – the ones you placed in a given project folder – can be accessed, and they will give your scripts green light to work properly.

Believe me, it's necessary and way easier than you may think.

Choosing a web browser

Since your game will run on all modern browsers, you will also need a web browser to make your games run into and test them.

I am using **Google Chrome** (http://www.google.com/chrome/) but you are free to use the one you prefer as long as it supports HTML5 `<canvas>` element.

Having the latest version of your web browser installed on your computer should be enough.

Refer to your browser support page to see if it supports `<canvas>` element.

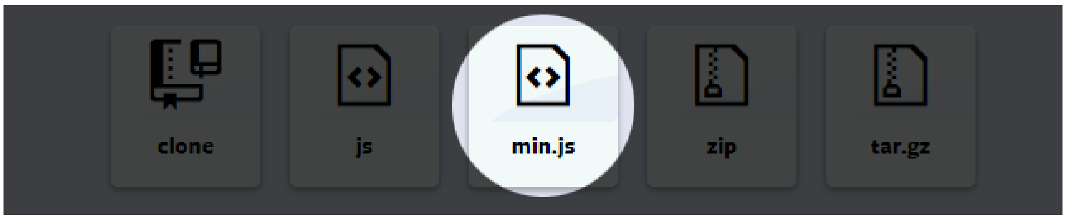
Once you choose a browser, it does not mean you should forget about the rest: you will test on your favorite browser during development to speed up the process, but from time to time you should also test it on the rest of the most used browsers.

Most of the times you won't notice any difference – all in all we are talking about cross platform games! – but the more you test, the better.

Downloading Phaser

Finally, it's time to download Phaser (http://www.phaser.io/download/stable) and you are ready to go.

In the official download page you will find various options to get your Phaser version, both with npm or through a CDN, but let's keep things simple and just download `phaser.min.js` file.



This is the only file you need to have in order to create your HTML5 game powered by Phaser.

The structure of your first Phaser project

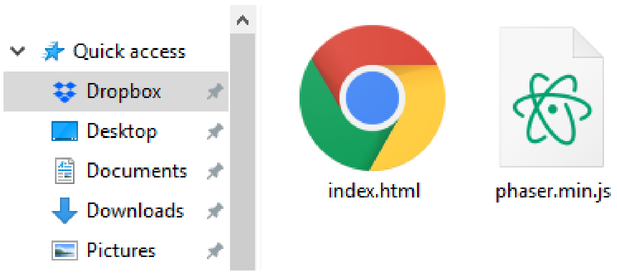
As said, every HTML5 game is a web page with some magic in it, so we are going to create a new folder which will contain the entire game.

The folder will contain the web page itself, the Phaser framework we just downloaded, and every other game file we will need as soon as the game gets more complex and needs more resources like sprites, backgrounds, sound effects and so on.

Done with the creation of the folder?

Create an `index.html` file which will be the web page you will call to launch the game, and you'll have all you need to start writing the first lines of code of your Phaser game.

If you followed all instructions, the folder containing your game should look like this:



Now let's edit `index.html`:

```
<!DOCTYPE html>
<html>
    <head>
        <script type="text/javascript" src="phaser.min.js"></script>
        <script type="text/javascript" src="game.js"></script>
    </head>
    <body>
    </body>
</html>
```

As you can see, it's just an empty web page calling two JavaScript files:

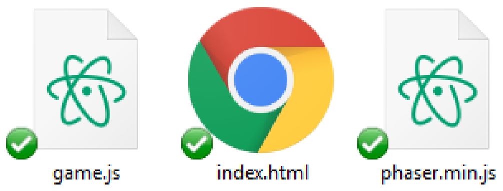
`phaser.min.js` is the file we just downloaded, and `game.js` will contain our game script.

Now create a new file, call it `game.js` and write this code:

```
window.onload = function() {
    var game = new Phaser.Game();
}
```

Congratulations, you just created your first Phaser script.

And this is how your game folder should look like now:



We will write the entire game code into `game.js` file.

Writing the whole game code into a single file may seem a malpractice, because the code would be way more understandable if organized in more files, such as a file for the configuration, a file handling player input, and so on.

The drawback is most game sponsors want the entire game to fit in just one file and probably won't accept the submission of your game if your code is scattered through a dozen files.

Since finding a sponsor – which means finding someone willing to pay for our hard work – is a top priority, that's why we are going to write the entire code in one single file.

Back to our code, let's see what these lines mean:

The `onload` event for the window fires after all objects in the DOM hierarchy (images, scripts, frames, and so on) have finished loading and the document object has been built up.

It checks for everything to be loaded, then calls the callback function.

> A JavaScript **function** is a block of code designed to perform a particular task, executed when "something" invokes it. When you make something to execute a function, we say you **call** the function.
>
> A function can contain any number of lines, and we use **curly brackets**, `{` and `}` to define the start and the end of a function, or more in general, the start and the end of a block of code.

Inside the function, there's only one line of code which initializes `game` variable as a new `Phaser.Game` instance:

```
var game = new Phaser.Game();
```

Translating what we done in English, it means “once the entire document has been loaded, create a new `Game` object and assign it to a variable called `game`.”

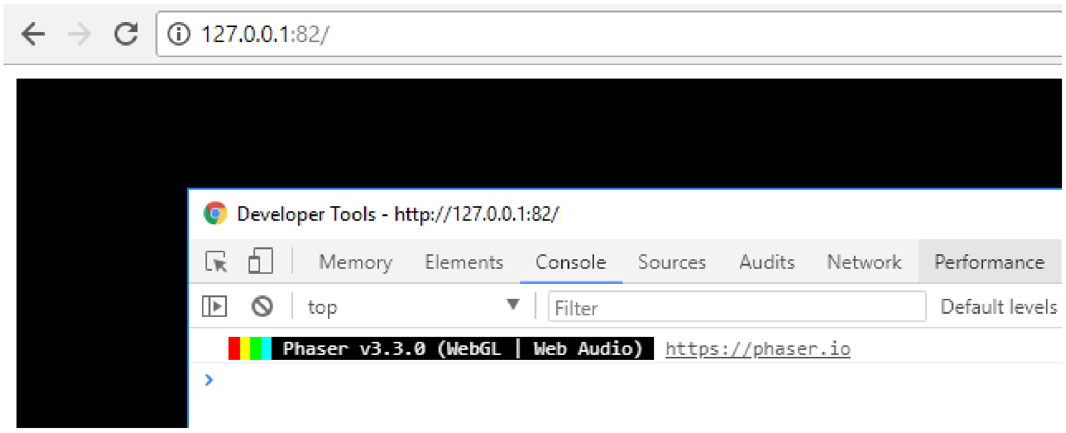
> A JavaScript **variable** is a container for storing data values. When you create a variable we say you **declare** a variable, using the `var` keyword. You assign a value to a variable with `=` operator.
>
> A JavaScript **object** is a particular kind of variable which can contain many values.

Well, this is our first Phaser game, so let's run it and see what happens.

Running your Phaser game

To run the game on your local web server, first setup the web server of your choice according to your preferences and the folder where you saved the game.

Then simply point your browser to `index.html` file in your game folder which in most cases will be something like http://localhost/yourphasergamefolder/ and this is what you should get if running it on your Google Chrome browser:



```
Phaser v3.3.0 (WebGL | Web Audio) https://phaser.io
```

The black square on the left is your game.

By default, Phaser creates a game on a black background.

You can also see which Phaser version you are using by opening the Console window pressing F12.

This book and all its examples and source codes refer to **Phaser 3.10.1**.

One of the things I love the most about Phaser? It is continuously updated, and this book too will be continuously updated to keep the code running and working on every major Phaser update.

Now let's inspect the elements in the page.

If you are using Chrome, right click on any page element and select Inspect Element.

You will see the following code:

```
<html>
    <head>
        <script type="text/javascript" src="phaser.min.js"></script>
        <script type="text/javascript" src="game.js"></script>
    </head>
    <body style="overflow: hidden;">
        <canvas width="1024" height="768"></canvas>
    </body>
</html>
```

Basically, Phaser created for you a 1024x768 `<canvas>` element with a black background, and this is where the game is running.

Awesome, but you may want a different game size and a different background color.

We need to see how to change some Phaser's default options.

Creating a Phaser Game configuration object

There are several options which can be configured when you are about to create a `Phaser.Game` instance, and you will need to place them inside an object which is passed as argument when you create the game.

> A variable passed as parameter inside a function is called **argument**.
>
> Arguments are passed to a function by placing them between the parentheses.
>
> Functions can have any number of arguments, separated by commas.

Add these lines to `game.js`:

```
window.onload = function(){
    var gameConfig = {
        width: 480,
        height: 640,
        backgroundColor: 0xff0000
    }
    var game = new Phaser.Game(gameConfig);
}
```

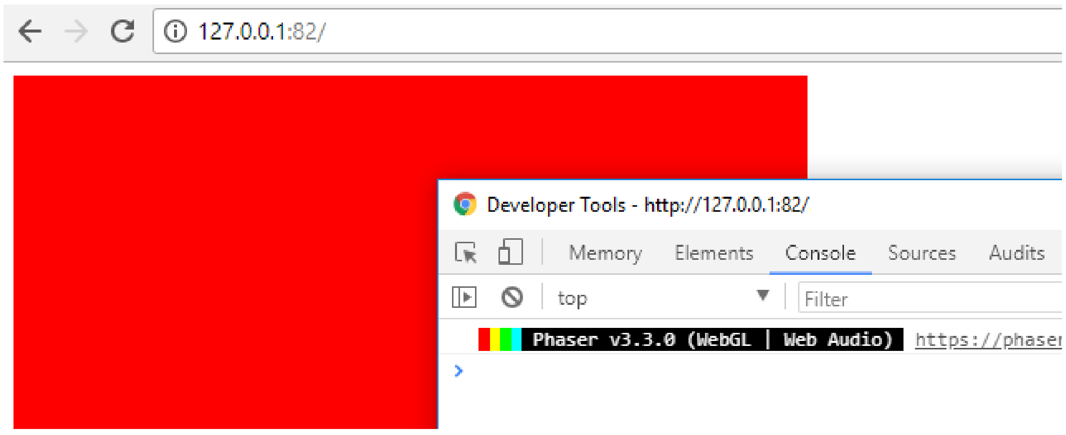
We created a new variable called `gameConfig` which is an object containing some game settings such as game width, height and background color.

The object is then passed as argument to `Phaser.Game` instance.

Now run the game, inspect the console and see what changed:

```
<html>
    <head>
        <script type="text/javascript" src="phaser.min.js"></script>
        <script type="text/javascript" src="game.js"></script>
    </head>
    <body style="overflow: hidden;">
        <canvas width="480" height="640"></canvas>
    </body>
</html>
```

Now the canvas has the same size we specified in `gameConfig` object and the background color of the game is red, and this is what you should see:



```
Phaser v3.3.0 (WebGL | Web Audio) https://phaser
```

Anyway, having the game in the top left corner of the page does not make it look that good. Let's center it in the page.

Adjusting CSS to make the game run in the center of the page

To make the game run in the center of the page, there's nothing of JavaScript involved.

We'll handle `<canvas>` element just like any ordinary web page element we want to center in its container.

Add these lines to `index.html`:

```
<!DOCTYPE html>
<html>
    <head>
        <title>My Awesome Game</title>
        <meta name = "viewport" content = "width = device-width, initial-scale =
               1.0, maximum-scale = 1.0, minimum-scale = 1.0, user-scalable = 0,
               minimal-ui" />
        <style type = "text/css">
            body{
                padding: 0px;
                margin: 0px;
            }
            canvas{
                display:block;
                margin: 0;
                position: absolute;
                top: 50%;
                left: 50%;
                transform: translate(-50%, -50%);
            }
        </style>
        <script type="text/javascript" src="phaser.min.js"></script>
        <script type="text/javascript" src="game.js"></script>
    </head>
    <body>
    </body>
</html>
```

We configured the viewport and added some style to make the `<canvas>` element be at the very center of the web page.

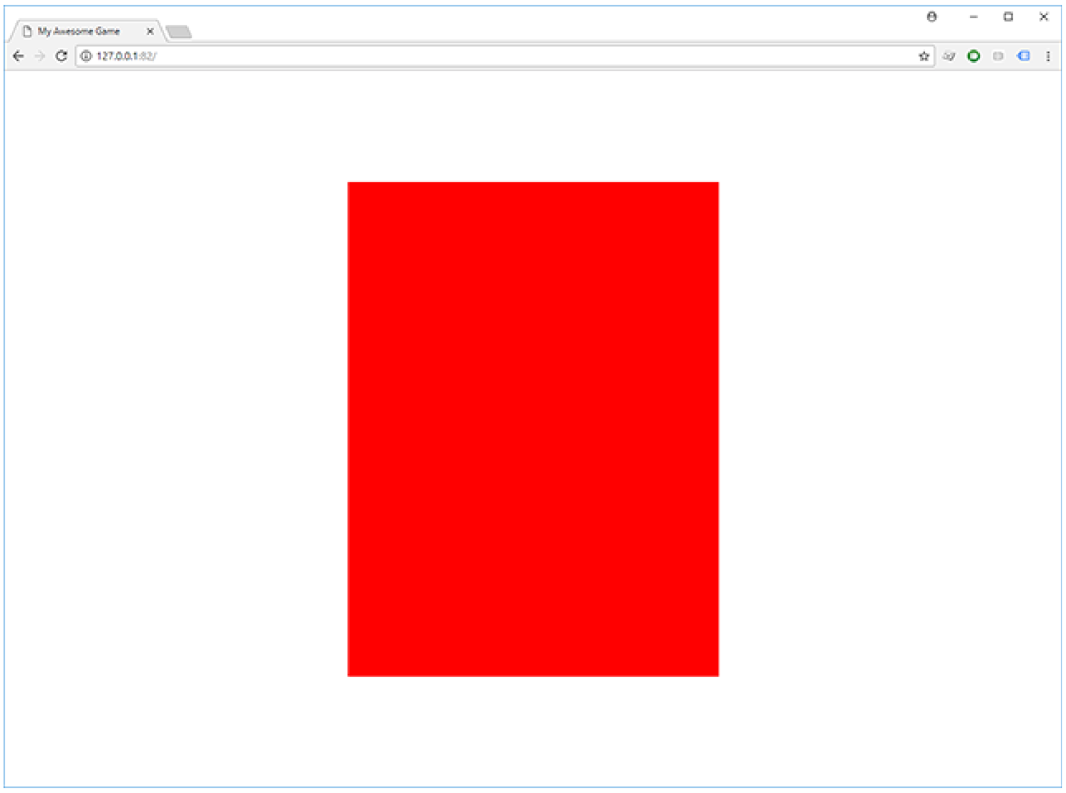
And – why not – we gave a title to the page.

> The browser's **viewport** is the area of the window in which web content can be seen.

Explaining CSS goes beyond the scope of the book, if you want more information you can Google for something like “center content with css” and you will find plenty of techniques and tutorials.

Or you can just take this CSS “as is” and enjoy the rest of game development.

What really matters now is the new page layout:



The game is now centered in the web page as we wanted, but it's not over yet.

No matter the actual size of the game – remember? 480x640 pixels – we want it to cover the wider area possible while keeping its size ratio.

This means the `<canvas>` element should be dynamically resized according to viewport size.

Resizing the game to cover the wider area possible

One of the most important goals to accomplish when building a cross platform game is to make it look good no matter the device it's running on.

For this reason, the first thing to do is try to cover the wider screen area possible while keeping the size ratio and make the game look pretty much the same on each device, no matter the resolution.

It gets frustrating when you run a game – especially on a mobile device – and you see it does not cover the entire screen. It just makes the game look old and outdated.

And obviously people want to play new and modern games.

We need to change a bit `game.js` rewriting some code and adding a new function.

```
var game;
window.onload = function(){
    var gameConfig = {
        width: 480,
        height: 640,
        backgroundColor: 0xff0000
    }
    game = new Phaser.Game(gameConfig);
    window.focus();
    resizeGame();
    window.addEventListener("resize", resizeGame);
}
// there's a whole new function below this comment
function resizeGame() {
    var canvas = document.querySelector("canvas");
    var windowWidth = window.innerWidth;
    var windowHeight = window.innerHeight;
    var windowRatio = windowWidth / windowHeight;
    var gameRatio = game.config.width / game.config.height;
    if(windowRatio < gameRatio){
        canvas.style.width = windowWidth + "px";
        canvas.style.height = (windowWidth / gameRatio) + "px";
    }
    else{
        canvas.style.width = (windowHeight * gameRatio) + "px";
        canvas.style.height = windowHeight + "px";
    }
}
```

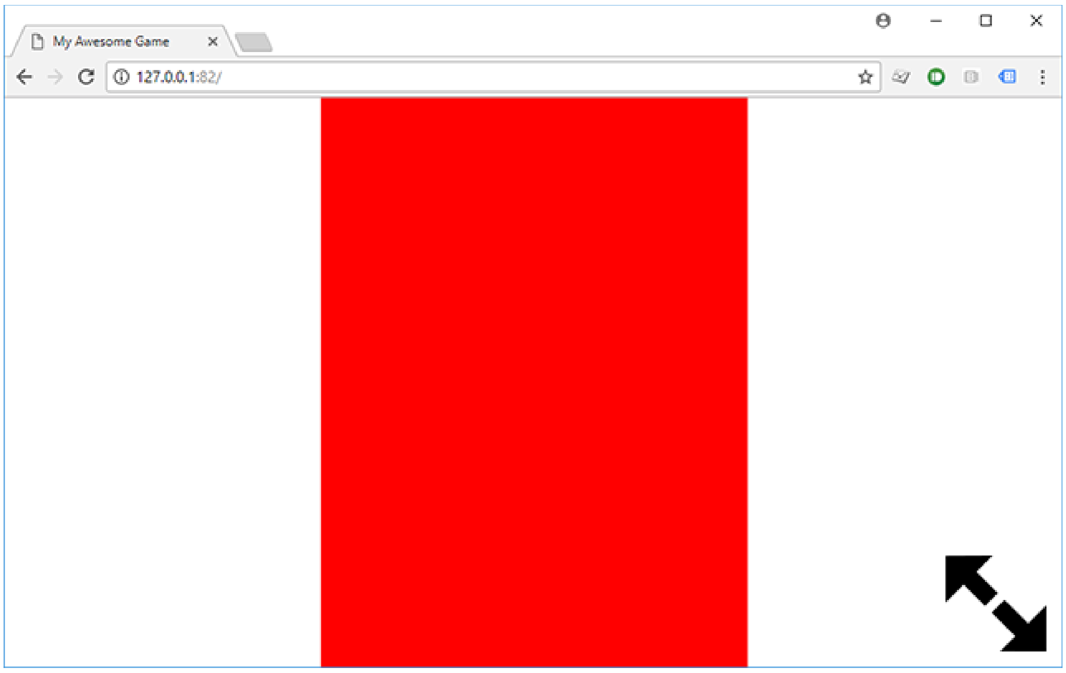
First, we declared `game` variable outside `window.onload` function.

We need to do it because we need to make `game` variable to be accessible not only inside `window.onload` function, but also in `resize` function.

In technical words, we changed the variable scope.

> In JavaScript, variables are only recognized inside their functions, and in functions inside their functions. Variables are created when a function starts, and deleted when the function is completed. The part of a script where a variable is recognized is called **scope**.

Now run the game again, and it should fill the whole height of the window.



Try to resize the window, and the game will resize accordingly.

What happened? First, let's see the changes inside `window.onload`:

```
game = new Phaser.Game(gameConfig);
```

`game` variable is now declared outside the function so there is no need for `var`. We just assign `game` its `Phaser.Game` instance.

```
window.focus();
```

Sometimes, especially when the game page is running in a iframe and it's controlled by keyboard, if you don't focus the window, keyboard controls simply won't work.

> `focus()` method sets focus to the current window, making a request to bring the current window to the foreground. It may not work as you expect in all browsers, due to different user settings.

Probably your game will work even if you don't focus it, but as it's just a line of code, let's add it.

```
resizeGame();
```

A call to `resizeGame` function is made. We'll explain it in a couple of minutes.

```
window.addEventListener("resize", resizeGame);
```

And the same `resizeGame` function will be called each time the window is resized, thanks to `resize` event listener.

> `addEventListener(event, callback)` method executes `callback` function whenever the specified `event` occurs.

In the end, `resizeGame` will be called immediately after the game instance has been created, and each time the windows is resized.

I have to say, it's not that usual for players to resize the window while they are playing, but since it takes us just one line of code to cover this case, why not covering it?

Have a look at the code in `resizeGame` function:

```
var canvas = document.querySelector("canvas");
```

First, we look for the canvas where the game is supposed to run.

```
var windowWidth = window.innerWidth;
var windowHeight = window.innerHeight;
```

`windowWidth` and `windowHeight` variables store browser window's inner width and height.

```
var windowRatio = windowWidth / windowHeight;
```

Given the width and the height, it's easy to know window size ratio, saved in `windowRatio` variable.

```
var gameRatio = game.config.width / game.config.height;
```

Same thing happens for game ratio, saved in `gameRatio` variable. Look how we can access to game configuration object through `config` property.

> `config` property of `Game` class access the game configuration object.
>
> `config.width` and `config.height` return respectively the width and the height of the game, in pixels.

At this point, we are in one of these two situations:

```
if(windowRatio < gameRatio){
    canvas.style.width = windowWidth + "px";
    canvas.style.height = (windowWidth / gameRatio) + "px";
}
```

This first block of code is executed when window ratio is lower than game ratio: this means the canvas can be resized to cover the full width of the window, and its height will fit anyway.

```
else{
    canvas.style.width = (windowHeight * gameRatio) + "px";
    canvas.style.height = windowHeight + "px";
}
```

This block of code is executed when window ratio is greater than or equal to game ratio: this means the canvas can be resized to cover the full height of the window, and its height width will fit anyway.

These few lines of code work pretty well with all common game sizes, no matter if they have a portrait or landscape width/height ratio.

`querySelector` method returns the first element within the document that matches the specified selector.

`innerWidth` property returns the width in pixels of the browser window viewport including, if rendered, the vertical scrollbar.

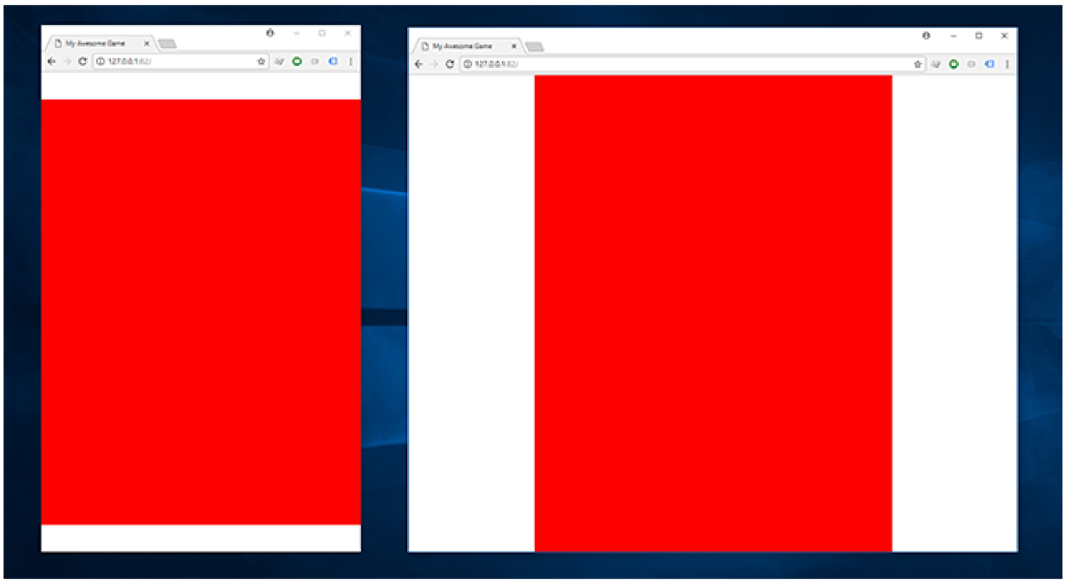
`innerHeight` property returns the height in pixels of the browser window viewport including, if rendered, the vertical scrollbar.

No matter the resolution and the size of your device, you will always find yourself in one of the two cases described above.

`if` statement executes a block of JavaScript code if a given condition is true, then you can optionally use `else` to specify the block of code to be executed if the same condition is false.

```
if(condition){
    // block to execute if the condition is true
}
else{
    // block to execute if the condition is false
}
```

Here is a real world example to show you how it works:



Here you can see two browser windows with two different size ratios compared to the ratio of the game.

In one case the canvas covers the full width, in the other case the canvas covers the full height.

What about the white stripes?

There are various ways to make them disappear, but we will see how to do it once the game is finished, as this will vary from game to game and it would be useless to talk about this feature without a real game to test.

Anyway, as you will see during the making of your first game, carefully choosing your game size and how in-game elements – especially the user interface – are displayed will help you to make your game look nice on every device.

At the moment we are done with game size, and we'll begin to focus on the game itself, as this red rectangle isn't a great game in my opinion, is it?

We are going to learn one of the most important concepts in the building of HTML5 games with Phaser.

Understanding Phaser scenes

Although managing Phaser scenes is an advanced feature, it's very important to learn to use scenes from the beginning of your Phaser programming course, as they will allow you to write better code and have a better resource management.

You are going to understand how scenes work right now in the making your first game, because learning good habits from the beginning is the key for a successful game development.

Let's think about a game, one of the games you are playing these days.

Although I don't know which games you are playing, I bet they all have at least a title screen, a screen with the game itself, a credits screen, a game over screen and so on, according to game complexity.

Each "screen" can be developed as a Phaser scene, which can be executed cleaning memory and resources before it starts, allowing us to easily switch through game "screens" without worrying about memory and resource management.

Questions like "Should I remove the title screen once the game starts" do not make sense if you are using Phaser scenes and developing the title screen in one scene, and the game itself in another scene.

To create the first scene, add this line to `window.onload` function in `game.js`:

```
window.onload = function(){
    var gameConfig = {
        width: 480,
        height: 640,
        backgroundColor: 0xff0000,
        scene: playGame
    }
    game = new Phaser.Game(gameConfig);
    window.focus();
    resizeGame();
    window.addEventListener("resize", resizeGame);
}
```

Scenes are added to the game by declaring them in game configuration object.

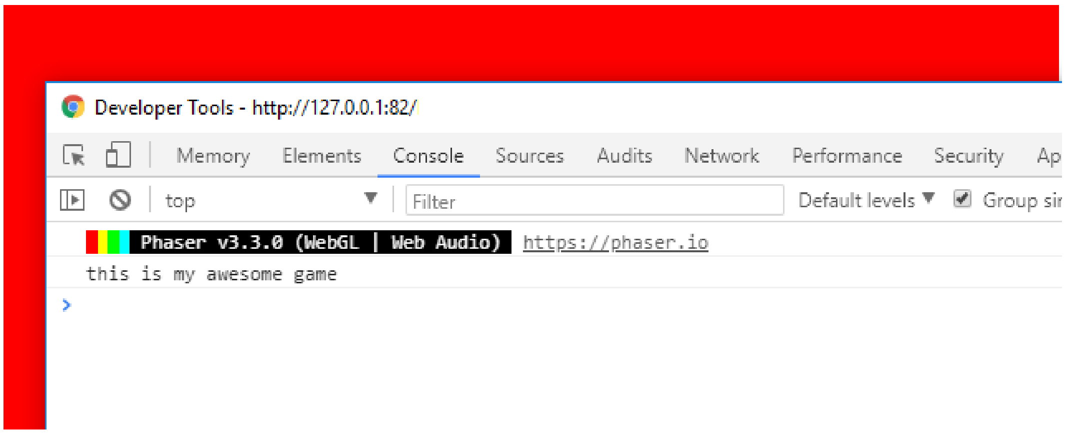
Don't forget the comma after `backgroundColor` value or you will get an error.

The new line means: there's a scene in the game, it's `playGame`, and being the only scene in the game, you have to run it as soon as the game starts.

And this is the code of the scene itself, to be added in `game.js`:

```
class playGame extends Phaser.Scene{
    constructor(){
        super("PlayGame");
    }
    create(){
        console.log("this is my awesome game");
    }
}
```

Launch the game again, and this is what you should see in the console:



```
Phaser v3.3.0 (WebGL | Web Audio) https://phaser.io
this is my awesome game
```

What happened? Let's have a look at the code, line by line.

```
class playGame extends Phaser.Scene{
    // rest of the code
}
```

We create `playGame` class – remember the game configuration file? – which extends `Phaser.Scene` class. `Phaser.Scene` is the original Phaser class which handles the scenes.

A **class** is a blueprint for an object. Classes describe the type of objects, while objects are usable instances of classes. Each object was built from the same set of blueprints and therefore contains the same properties and methods.

The `extends` keyword is used in class declarations to create a class which is a child of another class.

What's inside the class?

```
constructor(){
    super("PlayGame");
}
```

Since `playGame` class extends `Phaser.Scene` class, in `playGame` constructor we call the constructor of its parent class – `Phaser.Scene` – specifying a unique key which will be used to identify this scene: “`PlayGame`”.

It may seem a bit confusing, but we are only saying “hey, I have this scene and want it to behave like built-in Phaser scenes, but I'll probably add some more stuff”.

`constructor` method is a special method for creating and initializing an object created within a class.

`super` is used to access and call functions on an object's parent.

A constructor can use the `super` to call the constructor of a parent class.

We can say the rest of the script is the game itself:

```
create(){
    console.log("this is my awesome game");
}
```

We only have a function called `create`. This is a reserved name used by Phaser to know which function to execute once the scene has been called.

> A function inside an object is called **method**. For the same reason, when we refer to an object method, we mean a function declared inside the object itself.

Inside the method we just output something to browser console.

> `console.log(text)` will output the content of `text` in your browser console. Not all browsers support `console.log`, but Google Chrome does.

Your awesome game begins inside `create` method of the scene.

Adding more scenes to the game

Your Phaser game will have more than one scene, as it will have presumably more than one “screen” to show, and although your game can entirely fit in one single scene, it's highly recommended to have at least another scene where to preload the resources used in the game, such as graphics, sounds and fonts.

To add another scene to the game, first we declare it in game configuration file:

```
var gameConfig = {
    width: 480,
    height: 640,
    backgroundColor: 0xff0000,
    scene: [bootGame, playGame]
}
```

Now we have an array of scenes. The first scene in the array will be the one started at the beginning of the game.

An **array** is a special variable, which can hold more than one value at a time, under a single variable name, and you can access the values by referring to an index number.

`bootGame` scene will be built in the same way as we built `playGame`:

```
class bootGame extends Phaser.Scene{
    constructor(){
        super("BootGame");
    }
    create(){
        console.log("game is booting...");
        this.scene.start("PlayGame");
    }
}
```

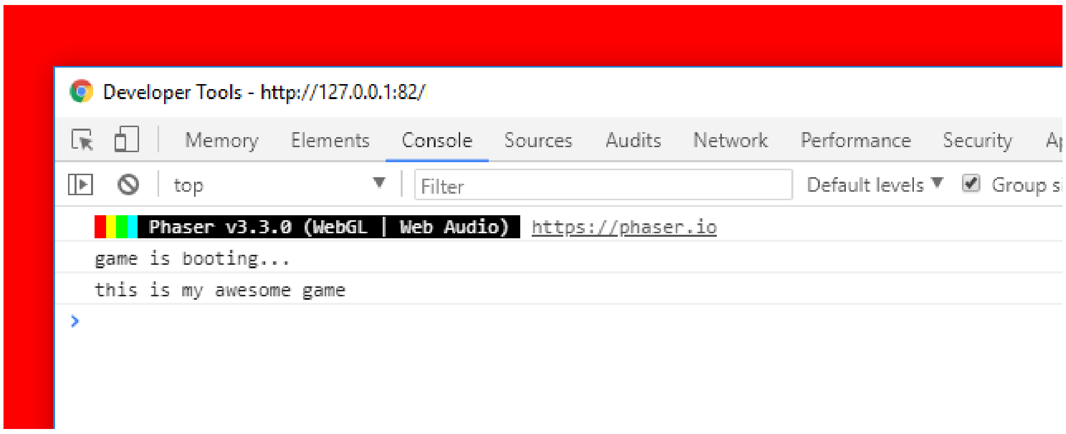
The only difference is in `create` method when we launch `PlayGame` scene.

`scene.start(key)` starts the scene identified by the unique `key` name, which is the same name you declared in the constructor with `super(key)`.

This way once `bootGame` is created, it immediately calls next scene.

> In JavaScript, `this` always refers to the owner of the function we're executing, or rather, to the object that a function is a method of.

Launch the game and this is what you'll see:



```
Phaser v3.3.0 (WebGL | Web Audio) https://phaser.io
game is booting...
this is my awesome game
```

The first scene called the second scene, as reported in the console window.

Here is the final structure of `game.js`:

```
var game;
window.onload = function(){
    var gameConfig = {
        width: 480,
        height: 640,
        backgroundColor: 0xff0000,
        scene: [bootGame, playGame]
    }
    game = new Phaser.Game(gameConfig);
    window.focus();
    resizeGame();
    window.addEventListener("resize", resizeGame);
}
class bootGame extends Phaser.Scene{
    // rest of the code
}
class playGame extends Phaser.Scene{
    // rest of the code
}
function resizeGame(){
    // rest of the code
}
```

This is only the beginning about scene management.

Scenes are also able to share information with other scenes, and they can even run in parallel, like in a game with a mini map in the top left corner: the game can be a scene, and the mini map another scene.

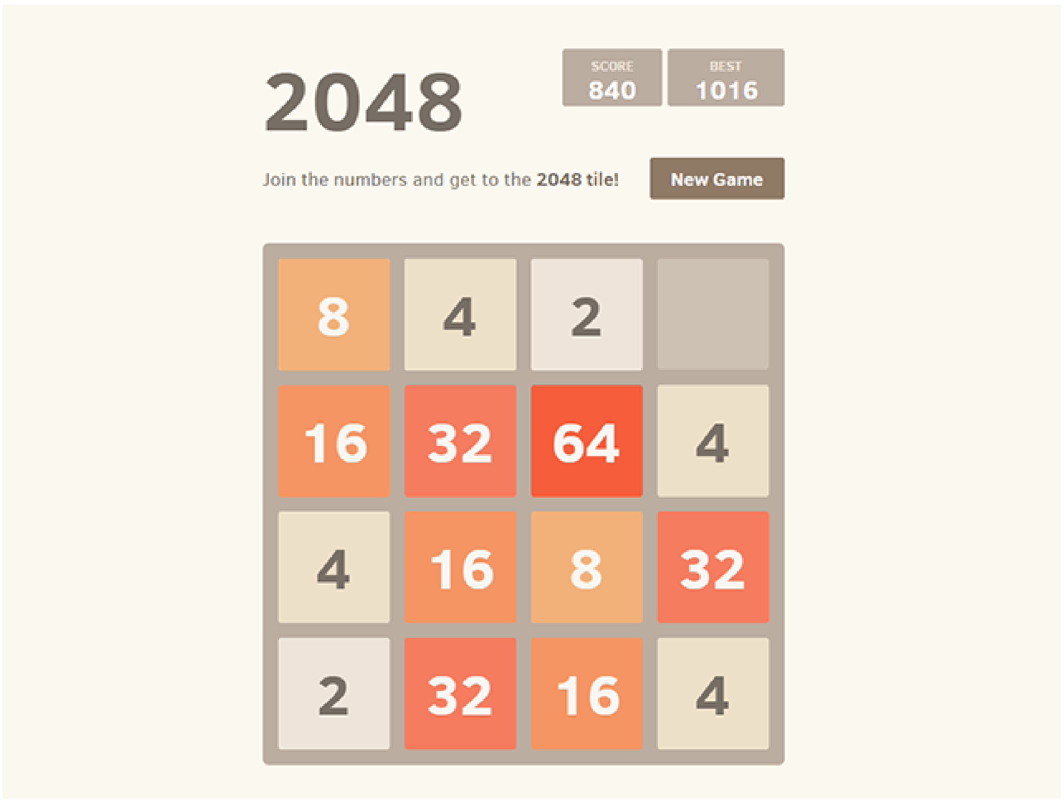
But now it's definitively time to start building our game, we'll dive into advanced scene management when we'll have an actual game to work with.

Just two words about the source code organization in this book: each main section has its own project folder for you to play with the source code.

> You will find the full project in the folder
>
> **000 - The structure of your first Phaser Project**

The game we are going to build: 4096

We are about to create a game called **4096**, which will be our take to **Gabriele Cirulli**'s **2048** smashing hit (https://gabrielecirulli.github.io/2048/), a puzzle game originally written in Javascript played on a 4x4 board.



Every turn, a new tile will appear in a random empty spot with a value of 2.

Using arrow keys, you slide tiles until they are stopped by either another tile or the edge of the grid.

If two tiles with the same number collide, they will merge into a tile with the total value of the two tiles that collided. Tiles can merge only once per turn.

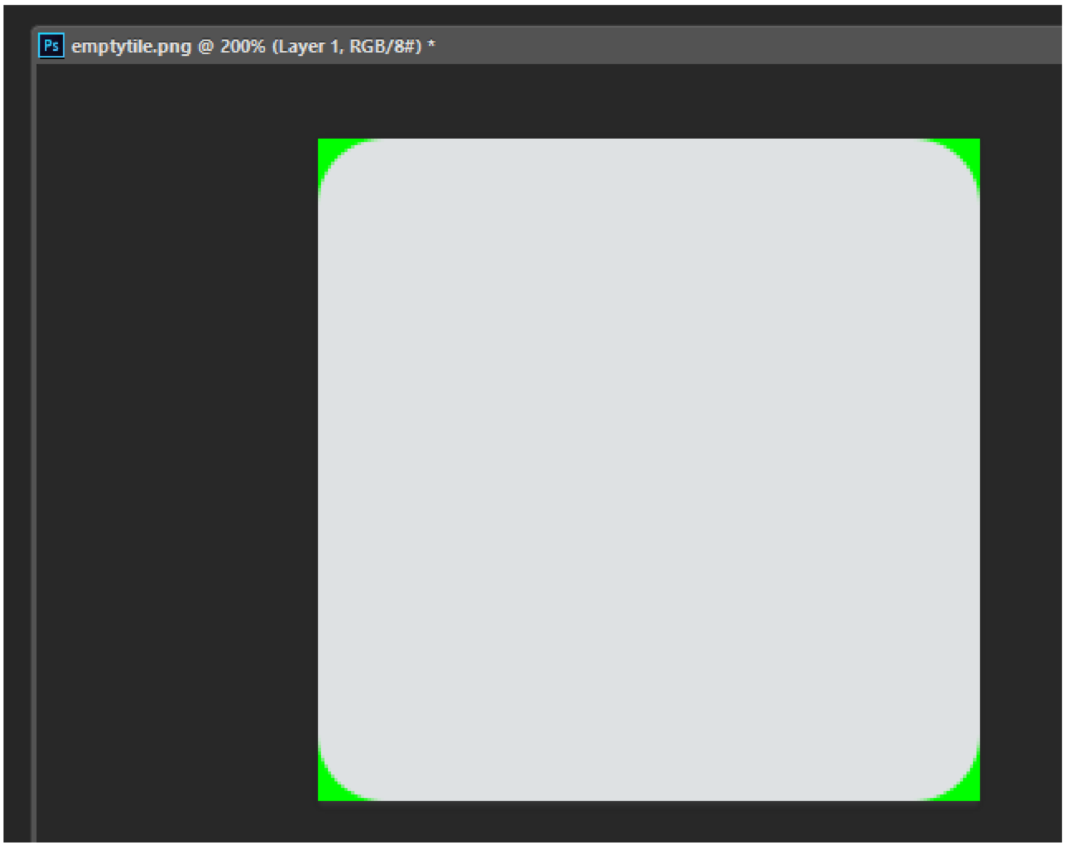
We are going to build this game, also adding some more features and room to customization, continuing from the game template we just built.

Preloading and adding images to the game

HTML5 games are a collection of images and sounds which move accordingly to game design and player input, so we need to learn how to preload and add images to the game.

Why do we need to preload images? Because one of the worst things you can do in the making of a game is to handle graphic assets before you actually loaded them.

I created a 200x200 pixels image called `emptytile.png` and will use this image to display the game board.



It's a PNG image with transparency, which I am showing you on a green background to let you see the transparency.

> Always save images as **PNG** as this format has the advantages of being lossless – it does not lose quality when saved – and supports alpha channel (transparency).
>
> Anyway, its lossless compression means larger files than JPEG.

To keep game folder well organized, the image has been saved into a folder called `sprites` which is created in a folder called `assets`.

You can obviously organize your game folder as you prefer, just keep in mind you should organize it in any way.

Once the image has been saved in its folder, it's time to preload it.

Add these lines to `bootGame` class:

```
class bootGame extends Phaser.Scene{
    constructor(){
        super("BootGame");
    }
    preload(){
        this.load.image("emptytyle", "assets/sprites/emptytile.png");
    }
    create(){
        this.scene.start("PlayGame");
    }
}
```

Just like `create`, `preload` is a reserved method of `Scene` class and is executed when the scene is preloading, and this is when we'll load the assets like the image we just created.

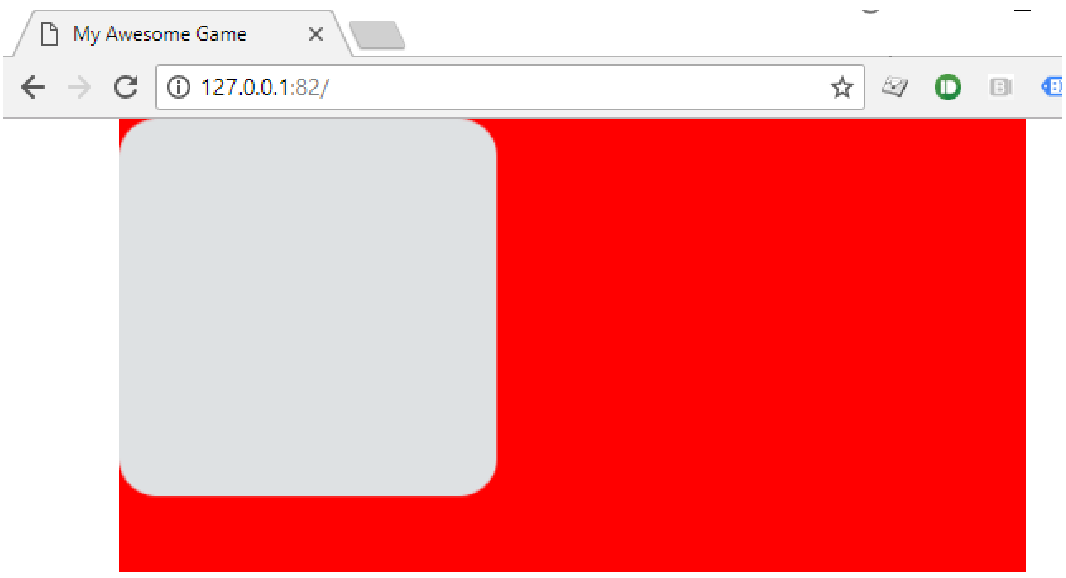
> `load.image(key, url)` loads an imagewants as arguments respectively the unique asset key of the image file and the URL of the image.

Once the execution of `preload` is over, Phaser executes `create` method which calls `PlayGame` scene, and this is where we are going to write the code to place the image somewhere in the canvas.

Add this line to `playGame` class:

```
class playGame extends Phaser.Scene{
    constructor(){
        super("PlayGame");
    }
    create(){
        this.add.image(100, 100, "emptytyle");
    }
}
```

Now launch the game, and this is what you should see:



Our 200x200 pixels image has now been placed in the game at coordinates (100, 100) starting from the upper left corner, where the origin (0, 0) coordinate is placed.

`add.image(x, y, key)` places an image on the stage and wants as arguments the x coordinate of the image, in pixels, the y coordinate of the image, in pixels, and the key of the image used.

From the image above is also easy to see the anchor point of the images added by `add.image` is the center of the images themselves.

This means the (100, 100) coordinate where we placed the image refers to the center of the image.

The **anchor point** of an image sets the origin point of the texture. The default anchor point is in the center, this means the texture's origin is centered.

Knowing the anchor point of your images is important because lets you know exactly how your images will be displayed in the game.

Adding a single image is not enough, as we said we are going to play on a 4x4 board. Let's display the entire board with the 16 images needed.

First, we need to make the game size bigger, as we need to display 4 images 200 pixels each, so we are changing `gameConfig` object:

```
var gameConfig = {
    width: 800,
    height: 800,
    backgroundColor: 0xecf0f1,
    scene: [bootGame, playGame]
}
```

Now the game will be 800x800 pixels, and we also change the background color from red to a light grey.

We are about to add 16 images, actually 16 instances of one image, so we don't need to preload anything else at the moment. Once you preloaded an image, you are free to use it as many times as you need.

Change `create` method inside `playGame` class this way:

```
create(){
    for(var i = 0; i < 4; i++){
        for(var j = 0; j < 4; j++){
            this.add.image(100 + j * 200, 100 + i * 200, "emptytyle");
        }
    }
}
```

The line which we used to add the image to the stage now is placed inside two `for` loops which will take care of executing `add.image` method 16 times, changing the coordinates according to `i` and `j` values.

The `for` loop continually repeats a block of code until a certain condition is reached.

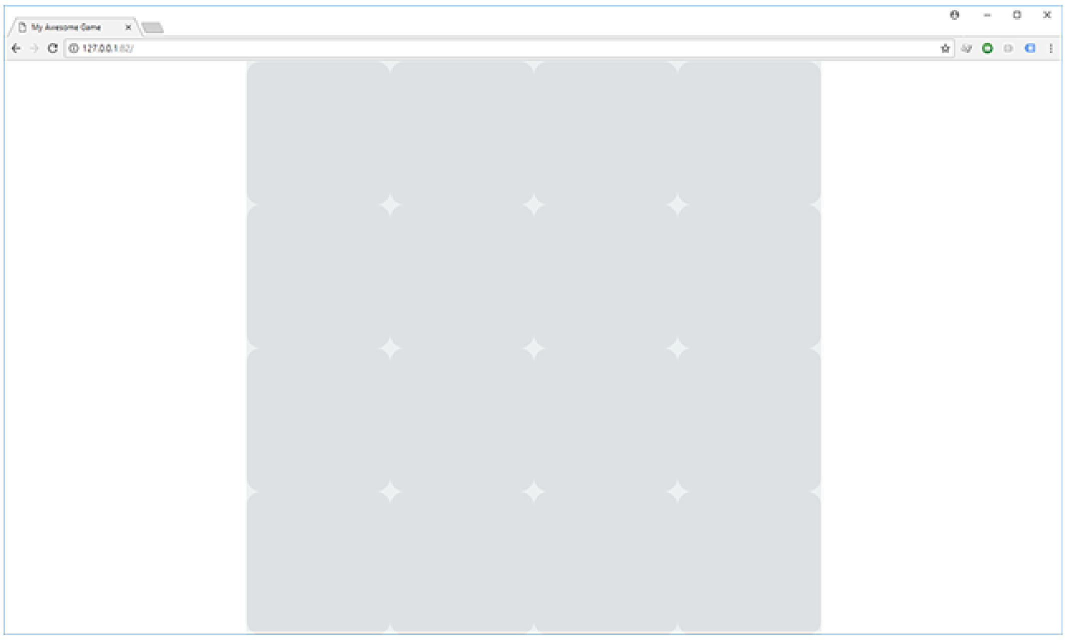
```
for(start action; condition; recurring action){
     code block to be executed
}
```

start action is executed only once before the loop starts.

condition defines the condition for running the loop. The loop will be executed as long as the condition is satisfied.

recurring action is executed each time after the loop has been executed.

Run the game and here's what you'll see:



Here we can see the 16 images placed on the state, but as they are too close to each other, the game will look better if insert some spaces between them.

`++` is the increment operator which increments (adds one to) its operand.

Both `i = i + 1` and `i++` add `1` to `i` variable

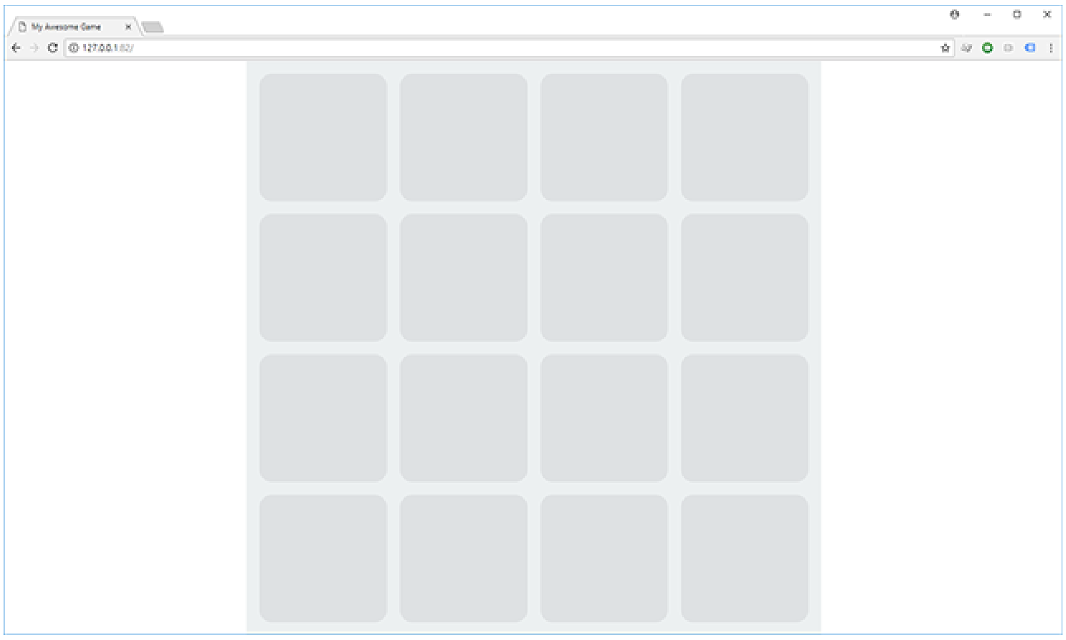
Let's add a 20 pixels spacing between tiles, by changing again `gameConfig` object to increase game size:

```
var gameConfig = {
    width: 900,
    height: 900,
    backgroundColor: 0xecf0f1,
    scene: [bootGame, playGame]
}
```

Then by modifying `create` method inside `playGame` class:

```
create(){
    for(var i = 0; i < 4; i++){
        for(var j = 0; j < 4; j++){
            this.add.image(120 + j * 220, 120 + i * 220, "emptytyle");
        }
    }
}
```

Now test the game and this is what you'll see:



Way better now.

We created our 4x4 game board filled with 200x200 pixels tiles with a spacing of

20 pixels, and resized the whole game accordingly, to display the entire board.

What if we decide to change tile size, or tile spacing, or maybe the number of rows and columns?

It's easy to replace the proper values in a 10 lines script, but what if we have a 1000 lines script with a lot of occurrences to search and replace?

What a big waste of time. There should be a better way to make our script customizable.

The importance of storing all game customizable variables in a single place

Every game has a series of variables which define the essence of the game itself.

In the making of 4096 game, for instance, we said we will be playing on a 4x4 grid, which is a grid with 4 rows and 4 columns, and each tile on the grid is a 200x200 pixels image with a 20 pixels spacing.

We are going to refer to these numbers a lot of times in the making of the game, each time we will need to know the size of the board, or the size of a tile, and believe me, it will happen quite often.

Rather than filling the source code with a series of "4", "200", and so on scattered here and there, it would be better to store these values somewhere safe and easy to access.

Not only our source code will be more readable, but above all your script will be a lot easier to modify should we decide to change the size of the board to, let's say, 5 rows by 3 columns, or use smaller or bigger tiles.

No more "search and replace" operations, but a single value to change.

That's why we are going to add a global object with some values we know we'll need a lot of times.

Add an object called `gameOptions` with these values:

```
var game;
var gameOptions = {
    tileSize: 200,
    tileSpacing: 20,
    boardSize: {
        rows: 4,
        cols: 4
    }
}
```

Just immediately after the declaration of `game` variable, `gameOptions` object contains:

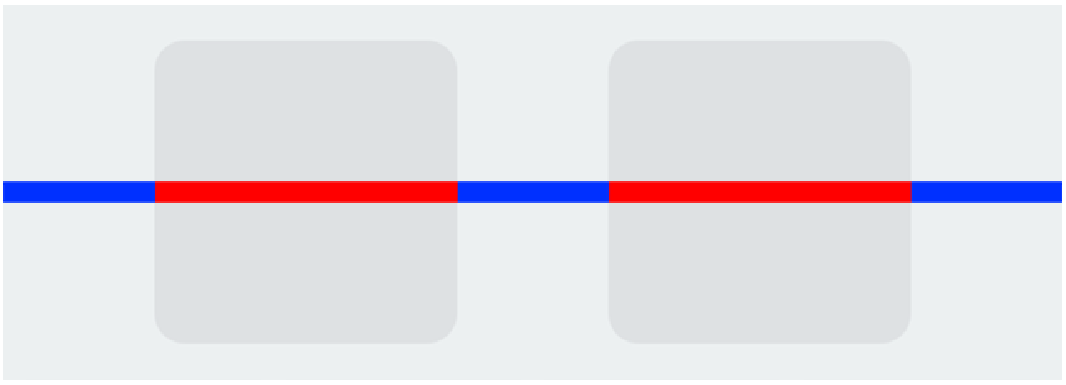
`tileSize`: the size of each tile, in pixels.

`tileSpacing`: the spacing between two tiles, in pixels.

`boardSize`: another object which contains the amount of rows and columns to be added to the board.

These values are the same ones we used before, injecting them directly inside the code. With this in mind, we have to change the rest of the game accordingly.

Look at this image:



To display two tiles, we need the width of the two tiles themselves – red segments – and the width of three tile spacings – blue segments.

To display n tiles, we can say we need n red segments and (n + 1) blue segments, simplified in:

`n * (red segment + blue segment) + blue segment`

Now we know how to set game width and height starting from the number of tiles, tile size and tile spacing:

```
var gameConfig = {
    width: gameOptions.boardSize.cols * (gameOptions.tileSize +
       gameOptions.tileSpacing) + gameOptions.tileSpacing,
    height: gameOptions.boardSize.rows * (gameOptions.tileSize +
       gameOptions.tileSpacing) + gameOptions.tileSpacing,
    backgroundColor: 0xecf0f1,
    scene: [bootGame, playGame]
}
```

By looking at the same picture, we can also determine the coordinate where to place tiles.

Assuming the first tile position is zero, the second tile position is one and so on, we can say the position of tile zero is given by the blue segment plus half the red segment.

The position of tile one is given by two blue segments plus one and a half red segments, and so on, simplified in:

`(n + 1) * blue segment + (n + 0.5) * red segment`

We are adding a new method to `playGame` class:

```
class playGame extends Phaser.Scene{
    constructor(){
        super("PlayGame");
    }
    create(){
        // same as before
    }
    getTilePosition(row, col){
        var posX = gameOptions.tileSpacing * (col + 1) + gameOptions.tileSize *
               (col + 0.5);
        var posY = gameOptions.tileSpacing * (row + 1) + gameOptions.tileSize *
               (row + 0.5);
        return new Phaser.Geom.Point(posX, posY);
    }
}
```

`getTilePosition` just applies the concept explained right above, and given a row and a column position, determines tile position in pixels.

> `return` statement stops the execution of a function and returns a value from that function.

The result is stored inside a `Phaser.Geom.Point` object, which is useful to store data with x and y coordinates.

> `Geom.Point` object represents a location in a two-dimensional coordinate system, where x represents the horizontal axis and y represents the vertical axis.

Finally we have to use `getTilePosition` method to place tiles inside `playGame`'s `create` method:

```
create(){
    for(var i = 0; i < gameOptions.boardSize.rows; i++){
        for(var j = 0; j < gameOptions.boardSize.cols; j++){
            var tilePosition = this.getTilePosition(i, j);
            this.add.image(tilePosition.x, tilePosition.y, "emptytyle");
        }
    }
}
```

First, we save into `tilePosition` the result of `getTilePosition` method, then we access to its `x` and `y` properties to get the coordinates where to place the tile.

Test the game and you'll see exactly the same thing as before.

Moreover, having a lot more math, I have to say.

Well, this was the goal of the new code: doing exactly the same stuff as before, just some math starting from a configuration object.

I know it worked like a charm even before these changes, but this is not the point: we worked a little more now to save a lot of time later on.

Do you want to play on a 5x4 board rather than a 4x4 board? Just change the proper value in `gameOptions` object.

Did you draw 150x150 pixels tile? Ajdust `tileSize` value in `gameOptions` object.

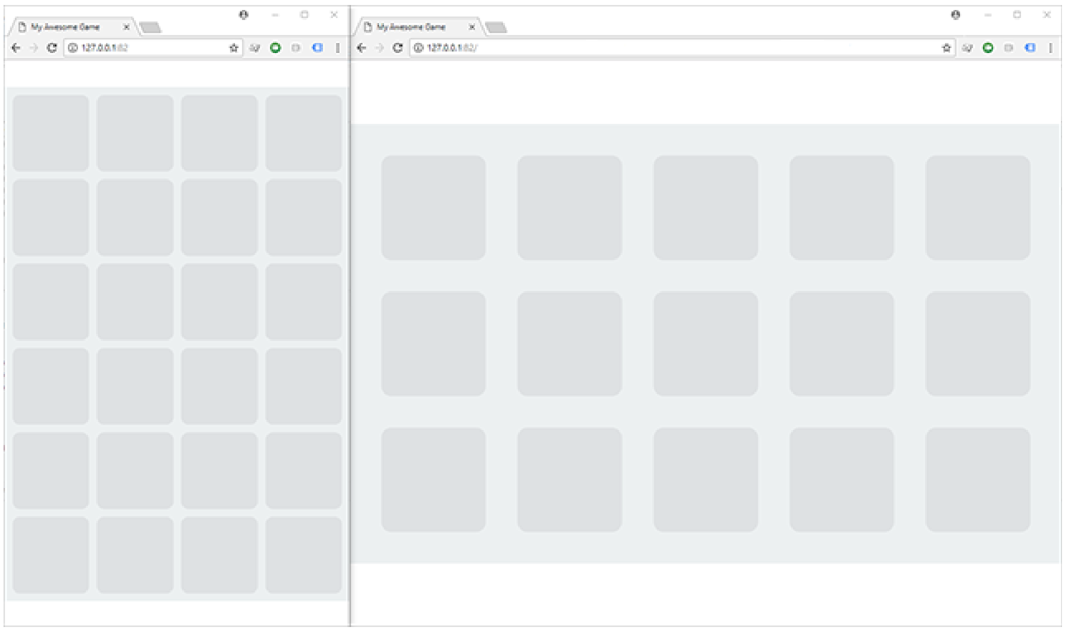
Need more spacing between tiles? I am sure you know how to do, changing just one value.

If you love TV series, we can say `gameOptions` object comes into play when it's time to change something in gameplay and you'd “Better call Saul”.

Believe me, it's a real pain when you wrote some thousands lines of code with values scattered here and there and you have to crawl line by line trying to adjust some option.

`gameOptions` is going to save you a lot of time in the long run.

Look at this picture:



Different rows and columns with different spacing just changing a couple of values.

> You will find the full project in the folder
>
> **001 - Preloading and adding images to the game**

Done with drawing the game board, let's create the tiles with the various numbers.

Creating tile graphics as a sprite sheet and using it in the game

We are going to create the tiles representing the numbers following the power of two, from 2 to 4096.

Since 4096 is 2^12, we need 12 tiles, one for the “2”, one for the “4”, one for the “8”, and so on until the one with “4096” on it. We need to draw 12 images.

At this time, we can save the 12 images in twelve distinct files or group them all into a sprite sheet.

> A **sprite sheet** is a series of images combined into a larger image. Usually the images are frames of an animation, thus a single image inside a sprite sheet is called **frame**.

Why using a sprite sheet?

Basically, every game is made by various graphical objects. In a space shooter you will find images representing spaceships, bullets and explosions, while in our 4096 game there will be different tiles. It does not matter the subject of the images. What we know is we are using all of them.

Each image has a width and a height, which represent the amount of pixels building such image, and each pixel requires some memory to hold its color information.

For each image – and more generally for each file – saved anywhere, there is a certain amount of memory that is wasted due to a series of features regarding the way the file system handles the files.

Explaining this concept goes beyond the scope of this book, just keep in mind the more files you have, the bigger the amount of memory wasted. It's not a problem when you are dealing with a dozen files, but in complex games with a lot of images, packing them into bigger images can save quite an amount of resources.

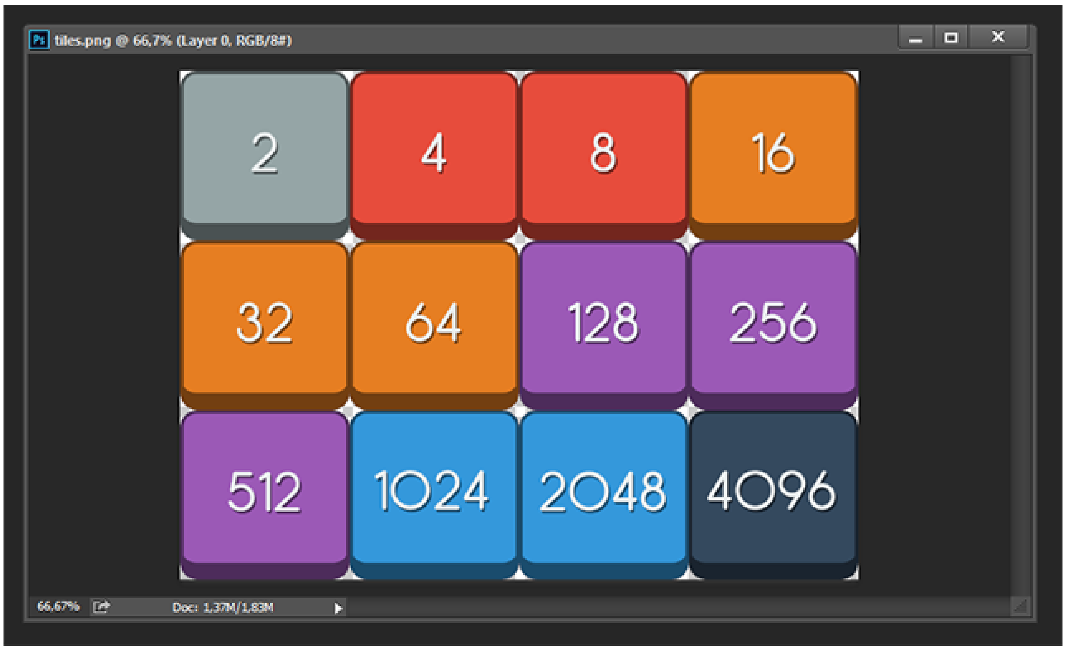
Moreover simply storing images is not enough. We also have to place them on the screen.

No matter the graphic engine your device will be using to display images, there will be a process which must know which image to paint, get the image from the place where it's stored, then know which part of the image to paint – normally the entire image – and where to paint it, and finally place it on the screen.

Once the first image has been placed on the screen, this process needs to be repeated for each other image, while your game until all images have been placed. Normally you don't notice it because it happens – or at least it should happen – in 1/60 second, but a lot of images to be placed on the screen of a slow device can slow down performance.

Using a sprite sheet, you will have all – or most of – your graphic assets placed into a big image, inside an invisible grid, in order to avoid the "what image should I load" question, keeping only the "which part of the image should I paint", and speed up the drawing process.

Look at this image:



Following this concept, I made a 800x600 pixels image containing all tiles, and now we need to repeat the process of preloading the image and using it in the game.

This image has also been saved in `assets/sprites` folder, so here's how we preload a sprite sheet in `preload` method of `bootGame` class:

```
preload(){
    this.load.image("emptytyle", "assets/sprites/emptytile.png");
    this.load.spritesheet("tiles", "assets/sprites/tiles.png", {
        frameWidth: gameOptions.tileSize,
        frameHeight: gameOptions.tileSize
    });
}
```

Basically the image is loaded in the same way as before, we just need an extra argument containing an object where to store the width and the height of the individual tile.

We are using the values stored inside `gameOptions` object.

> `load.spritesheet(key, url, config)` loads a sprite sheet and wants as arguments respectively the unique asset key of the file, the URL to load the texture file from and a configuration object with `frameWidth` value representing the frame width of each tile, in pixels, and `frameHeight` value representing the frame height of each tile, in pixels.

Once the sprite sheet has been loaded, it's easy to add the tiles to the game, in `create` method of `playGame` class:

```
create(){
    for(var i = 0; i < gameOptions.boardSize.rows; i++){
        for(var j = 0; j < gameOptions.boardSize.cols; j++){
            var tilePosition = this.getTilePosition(i, j);
            this.add.image(tilePosition.x, tilePosition.y, "emptytyle");
            this.add.sprite(tilePosition.x, tilePosition.y, "tiles", 0);
        }
    }
}
```

The syntax is almost the same, we only have one more argument to specify the frame we want to display, in this case zero = the first frame.

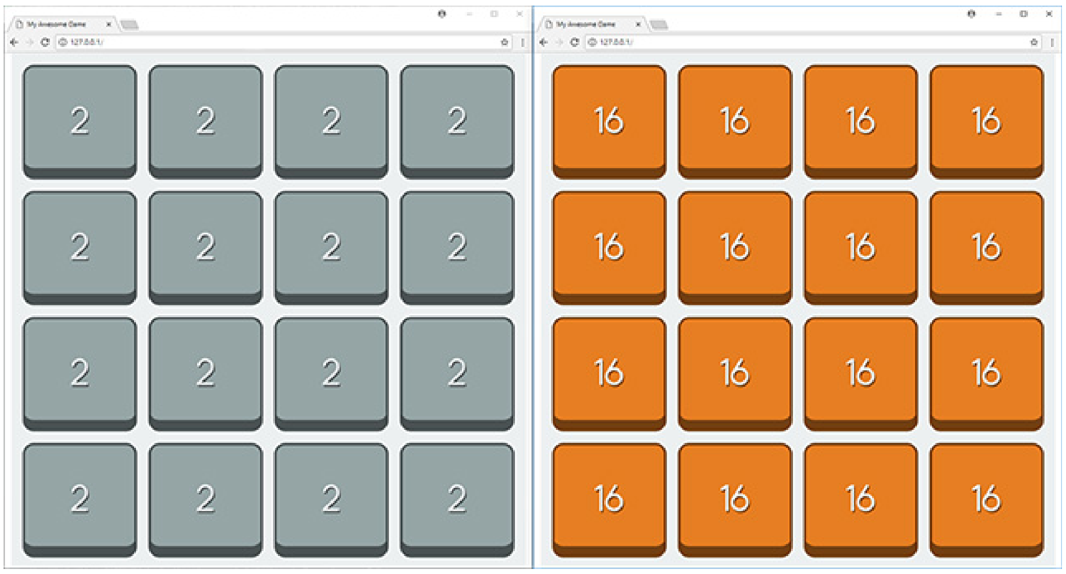
The big difference in with the previous line is we are adding a sprite rather than an image. This has nothing to do with the sprite sheet, although there's the “sprite” word in it you can load a sprite sheet and add its tiles as images too.

The main difference between a sprite and an image is you cannot animate images, and for this reason sprites takes a fraction longer to process.

I have to say, such “fraction” is not relevant unless you have thousands of images on the stage.

While you could to everything just using sprites and forget about images, I still use images to display static content such as the game interface, and sprites for the rest of the visual content, it makes the code look cleaner.

Run the code, and this is what you'll see:



On the left, the game with a series of “two”, showing the frame zero.

On the right, the same game with a series of “sixteen”, showing the frame 3 – actually the 4th frame, using

```
this.add.sprite(tilePosition.x, tilePosition.y, "tiles", 3);
```

Changing the last argument will change the frame displayed.

`add.sprite(x, y, key, frame)` places an image on the stage and wants as arguments the x coordinate of the image, in pixels, the y coordinate of the image, in pixels, the key of the image used and optionally the number of the frame to display, if a sprite sheet is used. Default value is zero.

Now that we are able to display images on canvas both using single images and sprite sheets, it's time to start defining game mechanics.

You will find the full project in the folder

002 - Creating tile graphics as a sprite sheet and using it in the game

We saw how to save resources by using a sprite sheet rather than multiple images. But we can save even more resource if we add as less images as possible.

The importance of adding as less sprites as possible during the game

If you played the original game, you'll see it's all a matter of adding and removing tiles. At each turn, a new tile is added. When two tiles merge into one, a tile is destroyed.

You can virtually create and destroy a lot of sprites, but the act of instantiating and destroying sprites can slow your projects down in the long run.

For this reason, the less sprites you create, the better. The problem is we need to create a new tile each turn, don't we? No, we don't.

The idea is to create the sixteen tiles – if you are playing on a 4x4 board – at once, then show or hide them according to board status.

All in all, we know we won't need more than sixteen tiles at the same time, when the board is full, and we already created them in the previous step. Now we only have to show or hide them according to what's happening in the game.

We'll start by hiding all tiles.

Change `create` method of `playGame` class this way:

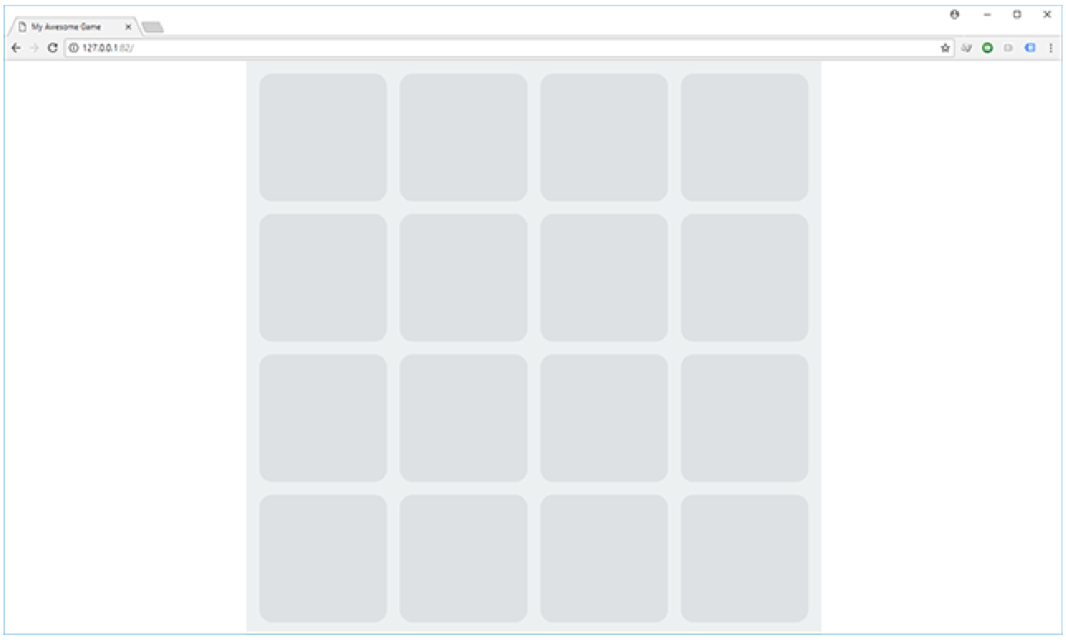
```
create(){
    for(var i = 0; i < gameOptions.boardSize.rows; i++){
        for(var j = 0; j < gameOptions.boardSize.cols; j++){
            var tilePosition = this.getTilePosition(i, j);
            this.add.image(tilePosition.x, tilePosition.y, "emptytyle");
            var tile = this.add.sprite(tilePosition.x, tilePosition.y, "tiles",
               0);
            tile.visible = false;
        }
    }
}
```

First, we assigned the tile sprite to a variable called `tile`. This will allow to manipulate the sprite rather than just place it and forget about it.

Then we hide the tile by setting its `visible` property to `false`.

> `visible` property sets visible state of the sprite. Non-visible sprites are not rendered.

Run the game and that's what you'll see – again:



It seems the empty board we created some steps ago, but actually it's full of tiles.

The game just does not render them at this time, thanks to `visible` Boolean property.

> A variable which can have only `true` or `false` values is called a **Boolean** variable.

Before we start making some tiles visible, we have to understand what's happens behind the curtains of the game.

Placing and removing tiles – or even better showing and hiding them – is only a visual effect to let the player see what's happening in the game.

The game itself runs silently behind the curtains, doing invisible operations which in the end lead to show, move or hide the elements players will see.

Using two-dimensional arrays to store board configuration

We already met arrays during the creation of a game configuration file, using an array to store the scenes used in the game.

We said arrays are special variables which can hold more than one value at a time, but arrays can also hold arrays.

Why should we use arrays holding arrays?

Array keep track of multiple pieces of information in linear order, a one-dimensional list. However, the data associated with certain environments like this board game lives in two dimensions, as the board has rows and columns.

To store this data, we need a multi-dimensional data structure, in this case a two-dimensional array.

A two-dimensional array is really nothing more than an array of arrays, a three-dimensional array would be an array of arrays of arrays and so on.

Let's create the two-dimensional array capable of storing board information, adding a few new lines to `create` method of `playGame` class:

```
create(){
    this.boardArray = [];
    for(var i = 0; i < gameOptions.boardSize.rows; i++){
        this.boardArray[i] = [];
        for(var j = 0; j < gameOptions.boardSize.cols; j++){
            var tilePosition = this.getTilePosition(i, j);
            this.add.image(tilePosition.x, tilePosition.y, "emptytyle");
            var tile = this.add.sprite(tilePosition.x, tilePosition.y, "tiles",
               0);
            tile.visible = false;
            this.boardArray[i][j] = {
                tileValue: 0,
                tileSprite: tile
            }
        }
    }
}
```

What happened? Let's break the new code line by line:

```
this.boardArray = [];
```

An empty array called `boardArray` is created. This will contain board information.

> **Empty arrays** are defined with open/close square brackets `[]`.

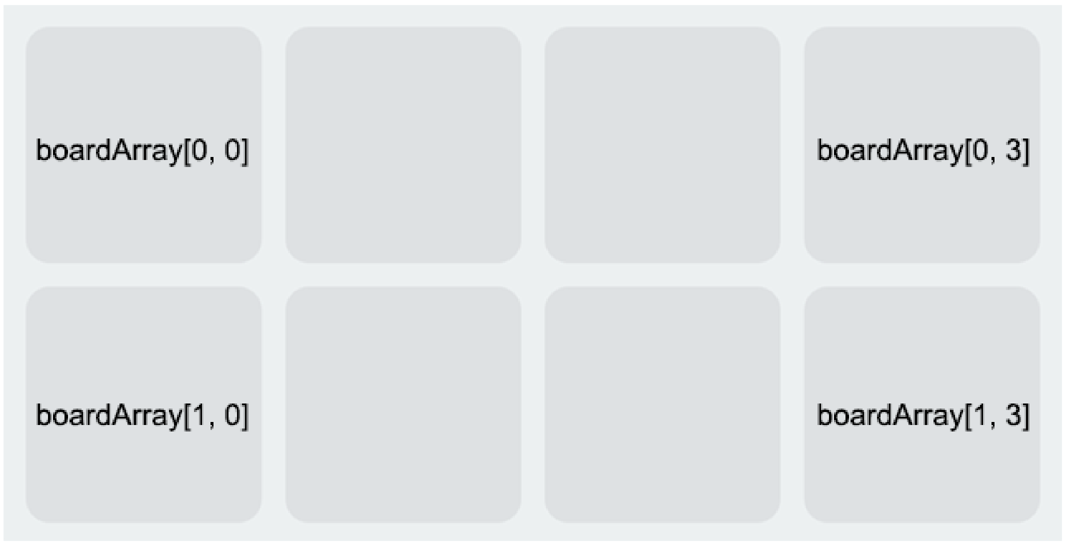
We said we are dealing with two-dimensional arrays, so each `boardArray` item itself must be declared as an empty array:

```
this.boardArray[i] = [];
```

The i-th element of `boardArray` is an array. Finally it's time to populate the array: each item will be an object, this way:

```
this.boardArray[i][j] = {
    tileValue: 0,
    tileSprite: tile
}
```

Now the board is defined this way:



`tileValue` is the value assigned to the tile, where zero means “empty tile”.

`tileSprite` is the sprite which will represent the tile. It's the tile sprite we created a couple of lines before.

Now the board has been defined as an array, other than being displayed on the screen.

If you run the game you won't see anything new, because we just defined the main base of data, the hidden but really important part of each game.

Now the strategy is to handle `boardArray` according to game events, then display the right sprites in the right places to make players see what happens, but the actual game lies inside `boardArray`.

At this time we can add tiles, or rather show tiles we previously added and made invisible.

Placing “two” tiles on empty spots on the board

At the beginning of the game, two new tiles with a “two” on them are added to the board on two empty spaces.

Then, a new “two” tile is added each turn.

This simple step needs to know which tiles are free, choose two random tiles among them, and show the “two” tile previously added and hidden.

We are going to add a custom method to `playGame` class which will take care of adding a new tile.

In `create` method, it will be called twice, this way:

```
create(){
    this.boardArray = [];
    for(var i = 0; i < gameOptions.boardSize.rows; i++){
        this.boardArray[i] = [];
        for(var j = 0; j < gameOptions.boardSize.cols; j++){
            var tilePosition = this.getTilePosition(i, j);
            this.add.image(tilePosition.x, tilePosition.y, "emptytyle");
            var tile = this.add.sprite(tilePosition.x, tilePosition.y, "tiles",
               0);
            tile.visible = false;
            this.boardArray[i][j] = {
                tileValue: 0,
                tileSprite: tile
            }
        }
    }
    this.addTile();
    this.addTile();
}
```

Once we'll code `addTile` method, we'll have our tiles added to the board.

Do you remember `boardArray` array and the concept of working behind the curtains?

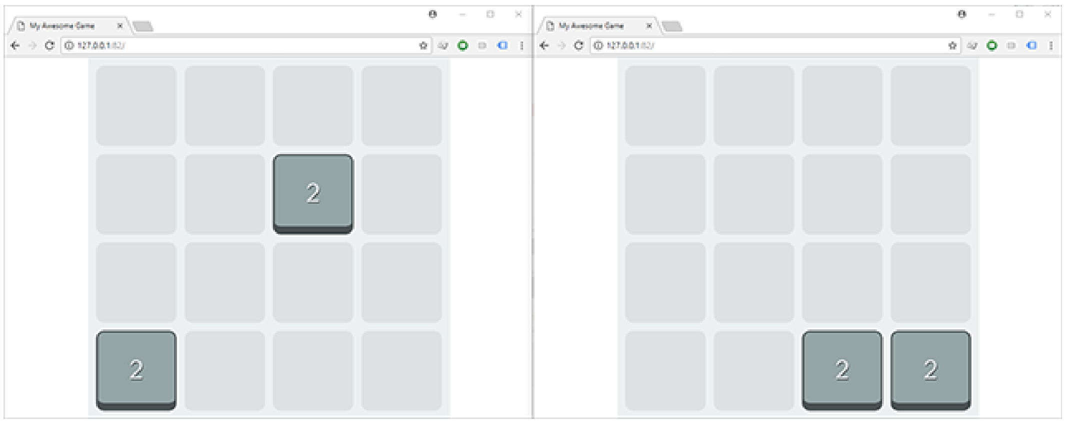
We can say a tile is empty when the corresponding `boardArray` item is an object with `tileValue` set to zero.

Looping through `boardArray` to look for empty tiles is the first thing to do.

Look at `addTile` method:

```
addTile(){
    var emptyTiles = [];
    for(var i = 0; i < gameOptions.boardSize.rows; i++){
        for(var j = 0; j < gameOptions.boardSize.cols; j++){
            if(this.boardArray[i][j].tileValue == 0){
                emptyTiles.push({
                    row: i,
                    col: j
                })
            }
        }
    }
    if(emptyTiles.length > 0){
        var chosenTile = Phaser.Utils.Array.GetRandom(emptyTiles);
        this.boardArray[chosenTile.row][chosenTile.col].tileValue = 1;
        this.boardArray[chosenTile.row][chosenTile.col].tileSprite.visible = true;
        this.boardArray[chosenTile.row][chosenTile.col].tileSprite.setFrame(0);
    }
}
```

Run the game, and each time you should see a couple of “two” tiles appearing in random empty positions:



Let's examine the code line by line:

```
var emptyTiles = [];
```

`emptyTiles` array will store all empty tiles we'll find. It starts as an empty array.

Then it's time to loop through all `boardSize` array, with two `for` loops running through all board rows and columns.

Look, we are using once again `gameOptions` object:

```
for(var i = 0; i < gameOptions.boardSize.rows; i++){
    for(var j = 0; j < gameOptions.boardSize.cols; j++){
        // rest of the code
    }
}
```

The first `for` loops through rows, the second one loops through columns.

What are we looking for? For empty tiles, that is `boardArray` elements whose `tileValue` is equal to zero.

```
if(this.boardArray[i][j].tileValue == 0){
    // rest of the code
}
```

At the beginning of the game, this condition will be always true, since the board is empty.

> `==` operator means **equal to**.

What happens when we find an empty tile?

```
emptyTiles.push({
    row: i,
    col: j
})
```

Yes, we add tile's coordinates as an object inside `emptyTiles` array.

> `push` adds new items to the end of an array.

What happens at the end of these two `for` loops? We have `emptyTiles` array filled of objects, each one representing the coordinates of an empty tile.

Now it's time to randomly pick a `emptyTiles` item, but the first thing to do is seeing is there is at least one empty tile, that is if `emptyTiles` array has at least one item.

```
if(emptyTiles.length > 0){
    // rest of the code
}
```

We can say we have an empty tile on the board if the length of `emptyTiles` array is greater than zero.

> `length` property returns the number of elements in an array.

This is how we pick a random array item:

```
var chosenTile = Phaser.Utils.Array.GetRandom(emptyTiles);
```

`chosenTile` now contains the object inside a randomly picked `emptyTiles` item.

> `Utils.Array.GetRandom(array)` method returns a random element from `array`.

At this time it's just a matter of showing the tile and updating `boardArray` array:

```
this.boardArray[chosenTile.row][chosenTile.col].tileValue = 1;
this.boardArray[chosenTile.row][chosenTile.col].tileSprite.visible = true;
this.boardArray[chosenTile.row][chosenTile.col].tileSprite.setFrame(0);
```

We set `tileValue` property to 1, show `tileSprite` sprite by setting its `visible` property to `true` and ensure we are showing the first frame.

> `setFrame(n)` method sets the frame the Game Object will use to render with.

You'll probably ask: “why should I assign `1` to `tileValue` if I am displaying a two? Wouldn't be better to assign `2`?”

I prefer to assign `tileValue` the frame number to be shown. This means I won’t store numbers representing powers of two like 2, 4, 8, 16, 32 and so on, but 1, 2, 3, 4 and so on.

Basically the concepts are similar, but storing frame numbers will allow you to keep the same code no matter the sequence of numbers you are using in the game: you can use powers of two like in this game, or a Fibonacci sequence, or completely random images.

> You will find the full project in the folder
>
> **003 - Adding tiles**

By the way, simply turn a tile visible is not enough as probably players expect some kind of animation when a new tile is added on the board.

Using tweens to animate tiles

While just showing the tile is an option, it's just a static image. No matter how cute your tiles look, they are just flat images if you don't animate them a bit.

That's why you are going to learn to create animations with Phaser tweens.

Tweens are a Phaser key feature. You will use a lot of tweens in the making of this game and more in general in the making of every game which requires animations.

Let's choose an animation speed, in milliseconds, which will be saved in `gameOptions` object:

```
var gameOptions = {
    tileSize: 200,
    tileSpacing: 20,
    boardSize: {
        rows: 4,
        cols: 4
    },
    tweenSpeed: 2000
}
```

The animation will take two seconds to complete, which is way too much to just show a tile, but it's just to let you see what happens, and we are going to change it later. Then we need to add some lines to `addTile` method to create the tween:

```
addTile(){
    // same as before
    if(emptyTiles.length > 0){
        var chosenTile = Phaser.Utils.Array.GetRandom(emptyTiles);
        this.boardArray[chosenTile.row][chosenTile.col].tileValue = 1;
        this.boardArray[chosenTile.row][chosenTile.col].tileSprite.visible = true;
        this.boardArray[chosenTile.row][chosenTile.col].tileSprite.setFrame(0);
        this.boardArray[chosenTile.row][chosenTile.col].tileSprite.alpha = 0;
        this.tweens.add({
            targets: [this.boardArray[chosenTile.row][chosenTile.col].tileSprite],
            alpha: 1,
            duration: gameOptions.tweenSpeed
        });
    }
}
```

The concept is to start with the tile completely transparent, and animate it until it's completely opaque.

Keep in mind “transparent” does not mean “invisible”. An invisible sprite is not rendered, thanks to its `visible` property, while a transparent sprite is rendered but you can see through it.

> `alpha` property sets the alpha – or the transparency – of the sprite. `alpha` range goes from zero – completely transparent – to one – completely opaque.

So the first line of our new code will make our sprite transparent by setting its alpha property:

```
this.boardArray[chosenTile.row][chosenTile.col].tileSprite.alpha = 0;
```

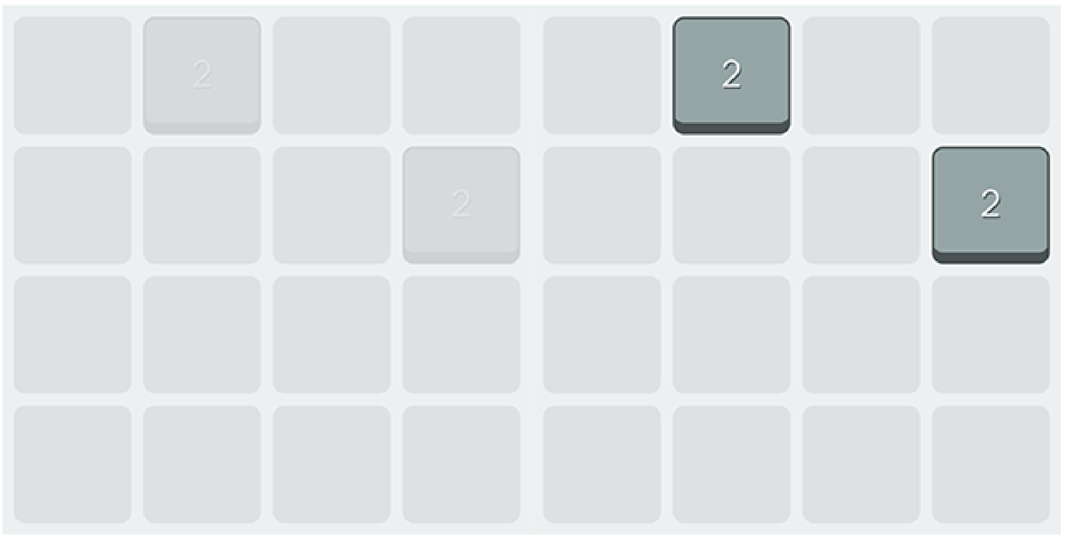
With the tile completely transparent, this is how we slowly turn it to opaque with a tween:

```
this.tweens.add({
    targets: [this.boardArray[chosenTile.row][chosenTile.col].tileSprite],
    alpha: 1,
    duration: gameOptions.tweenSpeed
});
```

Here you can see we are adding a tween to our tile sprite, bringing its alpha to 1 during `gameOptions.tweenSpeed` milliseconds.

> `tweens.add(config)` method creates and executes a tween with the options stored in `config` object.
>
> `targets` is the array containing all targets affected by the tween.
>
> `alpha` is the destination alpha.
>
> `duration` is the duration of the tween in milliseconds.

Have a look at the game and see how tiles appear smoothly thanks to tweens, the appeal of the entire game changes as it looks more polished.



Actually the animation is really too slow, it was just to let you see what happens, so we'll modify `tweenSpeed` value to make the game run smoother.

> You will find the full project in the folder
>
> **004 - Animating sprites with tweens**

Once the tiles are on the table, it's time to wait for player input, but first we have to wait for the tween to finish.

Checking when tweens end

Knowing when a tween ends is very important because in most cases we will need to do some things only after a tween ended.

Talking about this game, we have to wait for player input only after the tiles are completely visible.

How to determine when a tween ends? We know tween duration so we could merely wait for an amount of time, but it would be a very amateurish and inefficient way to solve this problem.

Too good Phaser comes in help with some more options to store in tween `config` object.

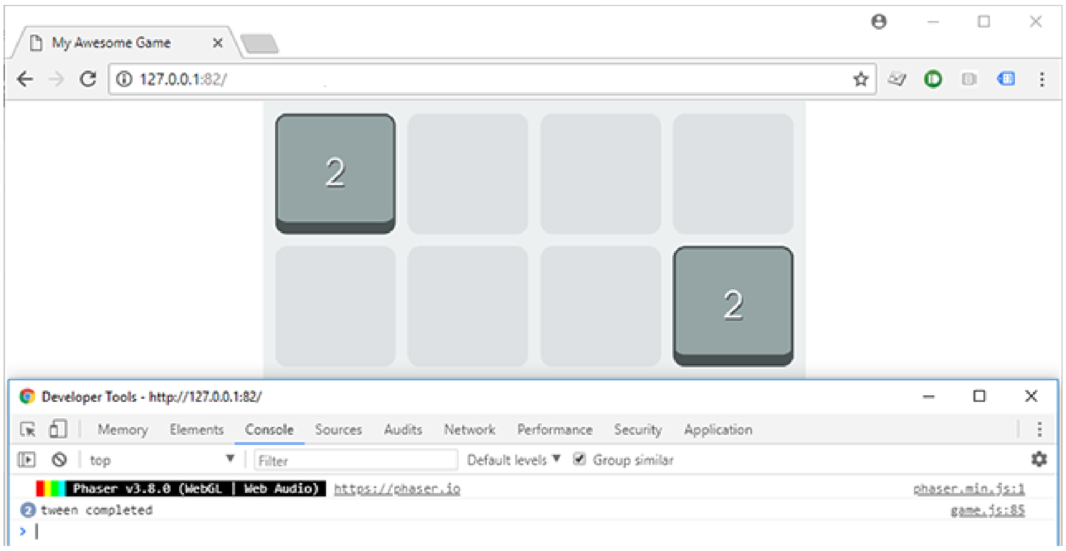
Let's say we have a property called `canMove` which we'll use to know if the player can move the tiles. At the very beginning of the game, when tiles are yet to be placed on the board, the player cannot move the tiles, so we'll add this line to `create` method:

```
create(){
    this.canMove = false;
    // rest of the script
}
```

`canMove` is a Boolean variable which starts with a `false` value, as the player cannot move. We have to set it to `true` once the tween which shows new tiles finishes, so here is how we change tween code inside `addTile` method:

```
this.tweens.add({
    targets: [this.boardArray[chosenTile.row][chosenTile.col].tileSprite],
    alpha: 1,
    duration: gameOptions.tweenSpeed,
    callbackScope: this,
    onComplete: function(){
        console.log("tween completed");
        this.canMove = true;
    }
});
```

Now run the game and when tiles are completely visible you should see "tween completed" appear in the console.


My Awesome Game
127.0.0.1:82/
2
2
Developer Tools - http://127.0.0.1:82/
Memory Elements Console Sources Audits Network Performance Security Application
top
Filter
Default levels
Group similar

```
Phaser v3.8.0 (WebGL | Web Audio) https://phaser.io          phaser.min.js:1
2 tween completed                                                  game.js:85
```

What happened? We added two options to the tween, to add a callback function which prompts some text to the console and sets `canMove` property to `true`, and to set the callback scope to `this`.

`onComplete` function is executed once the tween is completed.

`callbackScope` sets the scope of `onComplete` callback function.

Now we can know when a tween finishes and act consequently.

You will find the full project in the folder

005 - Checking when tweens end

Everything in game is ready, we only have to wait for player input.

Waiting for player input

There's no game without player input, so here we go straight to the point: being a cross platform game we will allow players to interact with the game using keyboard, mouse or fingers.

We'll add two lines to `create` method of `playGame` class:

```
create(){
    // same as before
    this.input.keyboard.on("keydown", this.handleKey, this);
    this.input.on("pointerup", this.handleSwipe, this);
}
```

These two lines will listen respectively for the player to press a key, calling `handleKey` method which we are about to write, and to release the pointer – no matter if mouse pointer or the finger – calling `handleSwipe` method, which we are also about to write.

From these two lines you can see two different ways of handling inputs.

Talking about keyboard, we define a keyboard input the action of pressing down a key, so that the callback function will be execute as soon as the player presses a key, and not when the key is released.

Talking about pointers, the pointer input calls the callback function once the player releases it, because we want the game to be controlled by swiping, and a swipe ends when the player raises the finger or releases the mouse button.

> `input.keyboard.on(“keydown”, callback, context)` executes `callback` function in `context` scope when a keyboard key is pressed.
>
> `input.on(“pointerup”, callback, context)` executes `callback` function in `context` scope when a pointer – mouse pointer or finger – is released.

Let's have a look at `handleKey` method. At the moment we just need to know which key has been pressed down.

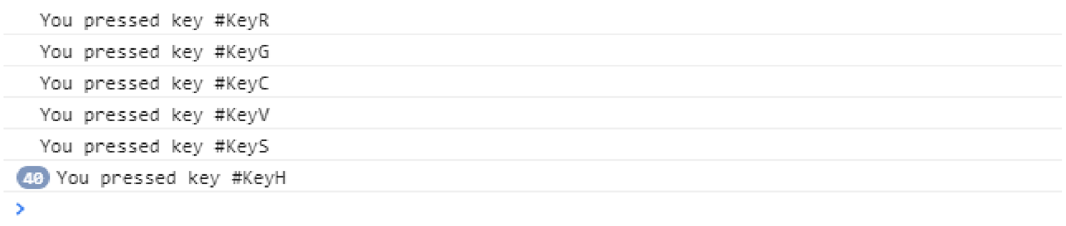
The game will be controlled with arrow keys, but we are going to add this feature later.

```
handleKey(e){
    var keyPressed = e.code
    console.log("You pressed key #" + keyPressed);
}
```

Like most callback functions, `handleKey` accepts the event itself as argument, so it's easy to save the code value of the key we just pressed in `keyPressed` variable.

> `code` property of a keyboard event returns the code of the key which fired the event.

Run the game, press some keys and this is what you should see in the console:



```
You pressed key #KeyR
You pressed key #KeyG
You pressed key #KeyC
You pressed key #KeyV
You pressed key #KeyS
40 You pressed key #KeyH
>
```

If you press and hold a key, `keydown` event will be triggered continuously, this is something to deal with when you'll move the tiles around the board.

And this is is `handleSwipe` method:

```
handleSwipe(e){
    var swipeTime = e.upTime - e.downTime;
    var swipe = new Phaser.Geom.Point(e.upX - e.downX, e.upY - e.downY);
    console.log("Movement time:" + swipeTime + " ms");
    console.log("Horizontal distance: " + swipe.x + " pixels");
    console.log("Vertical distance: " + swipe.y + " pixels");
}
```

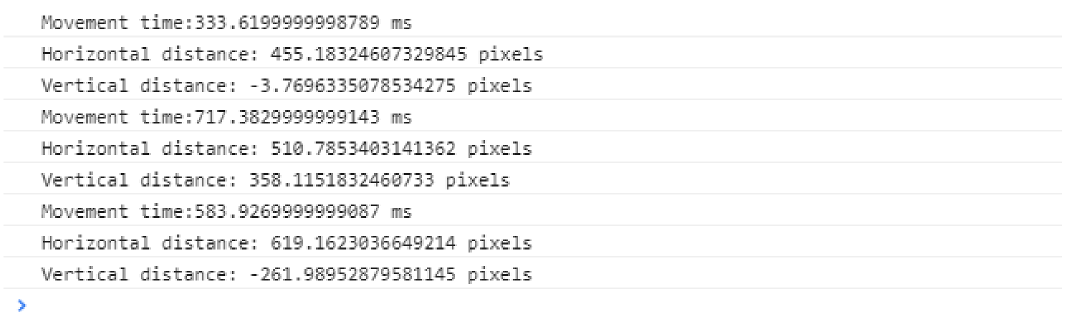
It's a bit more complicated because it's not just a matter of seeing which key has been pressed: to check for a swipe we need to know how much time passed since the player started the input – by clicking the mouse or touching the screen – and the distance traveled during such time.

As usual we can find this information in some event properties.

> `downTime` property of `pointerup` event returns the timestamp taken when the input started, in milliseconds.
>
> `upTime` property of `pointerup` event returns the timestamp taken when the input ended, in milliseconds.
>
> `downX` and `downY` properties of `pointerup` event return respectively the horizontal and vertical coordinates where the input started, in pixels.
>
> `upX` and `upY` properties of `pointerup` event return respectively the horizontal and vertical coordinates where the input ended, in pixels.

At the end of the script, `swipeTime` variable contains the duration of the swipe, in milliseconds, while `swipe` variable contains a `Point` object representing the distance traveled, in pixels.

Launch the game and drag here and there with the mouse, you should see something like this:



```
Movement time:333.6199999998789 ms
Horizontal distance: 455.18324607329845 pixels
Vertical distance: -3.7696335078534275 pixels
Movement time:717.3829999999143 ms
Horizontal distance: 510.7853403141362 pixels
Vertical distance: 358.1151832460733 pixels
Movement time:583.9269999999087 ms
Horizontal distance: 619.1623036649214 pixels
Vertical distance: -261.98952879581145 pixels
>
```

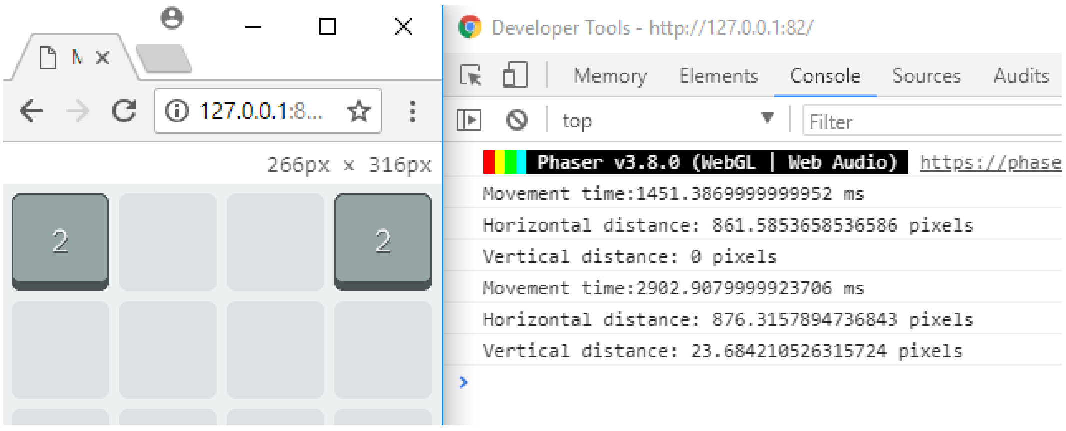
We have all required data to check if the player swiped, but there's another concept to cover about input events.

Understanding how pixels scale

We started talking about pixels, and how to measure distance in pixels, because pixels are a unit of measurement.

The problem is the game scales according to device resolution so different devices with different resolutions are supposed to have a different amount and density of pixels, and this should affect pixel measurements, right?

Wrong. Look at this picture:



```
Phaser v3.8.0 (WebGL | Web Audio) https://phase
Movement time:1451.3869999999952 ms
Horizontal distance: 861.5853658536586 pixels
Vertical distance: 0 pixels
Movement time:2902.9079999923706 ms
Horizontal distance: 876.3157894736843 pixels
Vertical distance: 23.684210526315724 pixels
```

We have a 266x316 pixels window, but if you move the pointer from side to side of you will always get a number around 900, which is the actual size of the game.

No matter the size of the canvas, you will be always working with game size.

This saves us from a lot of headache since we don't have to bear with screen resolutions or window sizes, we just have to set a game size and that's all.

You will find the full project in the folder

006 - Waiting for player input

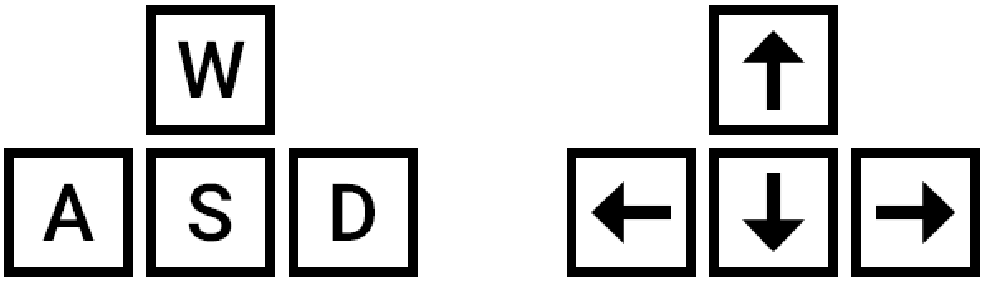
Ready to process player input?

Processing keyboard input

To process keyboard input, we first need to decide which keys can be used by the player to move tiles.

We'll move tiles both with ARROW and WASD keys, which are combinations working on most keyboards.

Do you notice it? The layout of both key combinations is similar. I usually play with WASD keys.



We have to rewrite `handleKey` method:

```
handleKey(e){
    if(this.canMove){
        switch(e.code){
            case "KeyA":
            case "ArrowLeft":
                this.makeMove(LEFT);
                break;
            case "KeyD":
            case "ArrowRight":
                this.makeMove(RIGHT);
                break;
            case "KeyW":
            case "ArrowUp":
                this.makeMove(UP);
                break;
            case "KeyS":
            case "ArrowDown":
                this.makeMove(DOWN);
                break;
        }
    }
}
```

Let's break the code line by line to see what happened.

First, we need to know if the player can move:

```
if(this.canMove){
    // rest of the script
}
```

Nothing happens if the player can't move, but if `canMove` is `true`, we need to know which key has been pressed:

```
switch(e.code){
    // rest of the script
}
```

Here we can do various things according to key pressed code.

`switch` statement executes a block of code depending on different cases.

```
switch(expression) {
    case 1:
        // code block
        break;
    case 2:
        // code block
        break;
    // case 3, 4, and so on
}
```

`expression` is compared with the values of each case. If there is a match, the associated block of code is executed.

`break` keyword breaks out of the `switch` block and stops the execution of more execution of code and case testing inside the block.

If `break` is omitted, the next code block in the `switch` statement is executed.

Let's see the cases we want to add to the script. If the player presses LEFT arrow key or A key, we have to move tiles to the left

```
case "KeyA":
case "ArrowLeft":
    this.makeMove(LEFT);
    break;
```

`KeyA` and `ArrowLeft` are the key codes of respectively A and LEFT arrow keys.

In this case we call `makeMove` method which will be written in a minute, then break the execution of the `switch` statement.

```
case "KeyD":
case "ArrowRight":
    this.makeMove(RIGHT);
    break;
```

The same concept is applied to D and RIGHT arrow keys: execution of `makeMove` method and break of the `switch` statement.

There is no need to explain the remaining two cases as they handle W or UP arrow keys and S or DOWN arrow keys in the same way.

As for `makeMove` method, it's just a placeholder at the moment:

```
makeMove(d){
    console.log("about to move");
}
```

Now I want to focus on the way we call `makeMove` method and its argument, which is the direction.

These are the four times we call `makeMove`:

```
this.makeMove(LEFT);
this.makeMove(RIGHT);
this.makeMove(UP);
this.makeMove(DOWN);
```

Where do these `LEFT`, `RIGHT`, `UP`, `DOWN` come from?

They are just global constants to be declared after `gameOptions` object:

```
var gameOptions = {
    tileSize: 200,
    tileSpacing: 20,
    boardSize: {
        rows: 4,
        cols: 4
    },
    tweenSpeed: 2000
}
const LEFT = 0;
const RIGHT = 1;
const UP = 2;
const DOWN = 3;
```

Let me explain: we have four directions, so we assign each direction a number, which will come in hand when it's time to see where to move tiles.

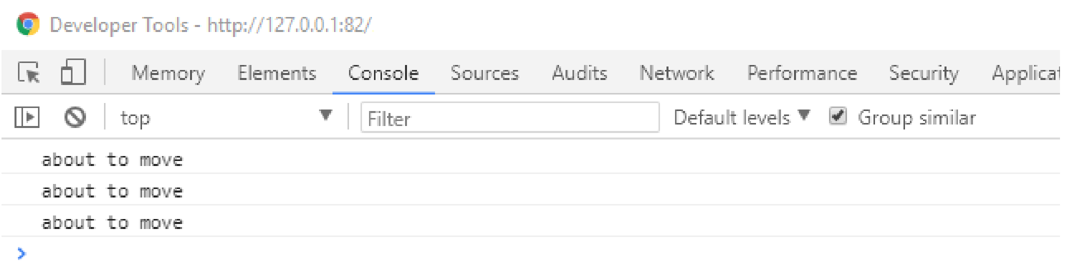
We could have written the first `switch` case this way:

```
case "KeyA":
case "ArrowLeft":
    this.makeMove(0);
    break;
```

But let's suppose you have to update the code in a couple of years, would you find more readable `this.makeMove(0)` or `this.makeMove(LEFT)`?

That's why we are using constants to store the directions: they make the code much more readable. And a code you can easily read is a code you can easily modify.

Now test the game:



```
about to move
about to move
about to move
```

You will see “about to move” in the console only when you press one of the keys whose key code is part of a case in the `switch` statement.

> A JavaScript **constant** is a variable that will never change and is declared with `const` keyword. Trying to modify a constant value will throw an error.

Actually, working with constants is not mandatory, you can use `var` with conventions like ALL_CAPS – writing all letters as capital letters – to identify values which should never change.

Moreover, `const` won't work on older browsers, which are by the way too old to be used to play HTML5 games, so it's up to you to use `var` or `const` to handle values you don't want to change.

> You will find the full project in the folder
>
> **007 - Processing keyboard input**

Done with the keyboard input, let's process pointer input.

Processing pointer input

Pointer input is a bit more complicated to process because it isn't just a matter check a key code but we have to see if the input was actually a swipe rather than a simple click/tap or an unintentional touch.

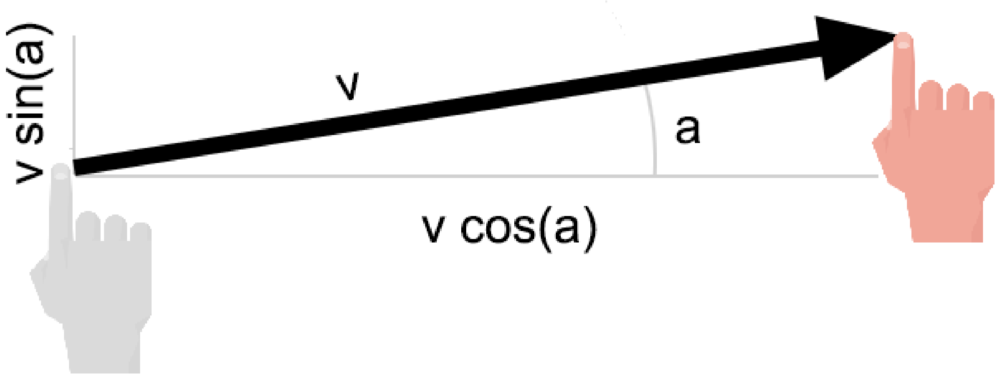
To say a pointer input was a swipe, we need three conditions to be satisfied:

First: the duration of the movement. Swipe is a quick gesture, unlike a drag movement which can last several seconds.

Second: the distance covered by the movement. Unlike a tap or a click, in a swipe we expect the ending point of the input to be at a certain distance from the starting point of the input.

Third: the direction. Swipes are supposed to be horizontal or vertical, so we don't have to consider diagonal movements.

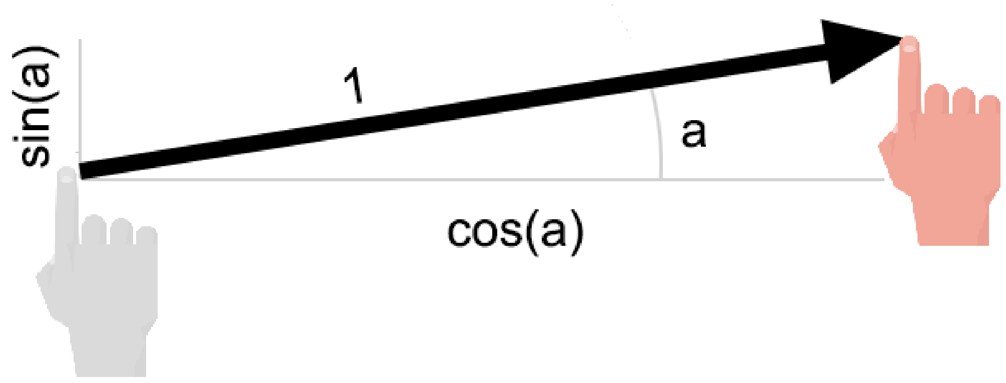
This is when the concepts of vector magnitude and normalization come into play.



We can assume each input move to be a vector, this way:The pointer moves generating a vector with a direction – the angle – and a magnitude – the length.

All vectors also have a horizontal and a vertical component, which can be calculated multiplying the magnitude respectively by the cosine and the sine of the direction.

Vectors can also be normalized, this happens when you take a vector of any length and, keeping it pointing in the same direction, change its length to 1, turning it into what is called a unit vector, this way:



End with the boring theory, let's think about processing pointer input.

We are going to add three new values in `gameOptions` global object to check for the three conditions mentioned above in order to see if a pointer event is a swipe.

As for the duration of the movement, we can say a swipe movement can't last more than one second.

The minimum distance between the start and the end of the input – that is the magnitude of the pointer input – will be set to 20 pixels.

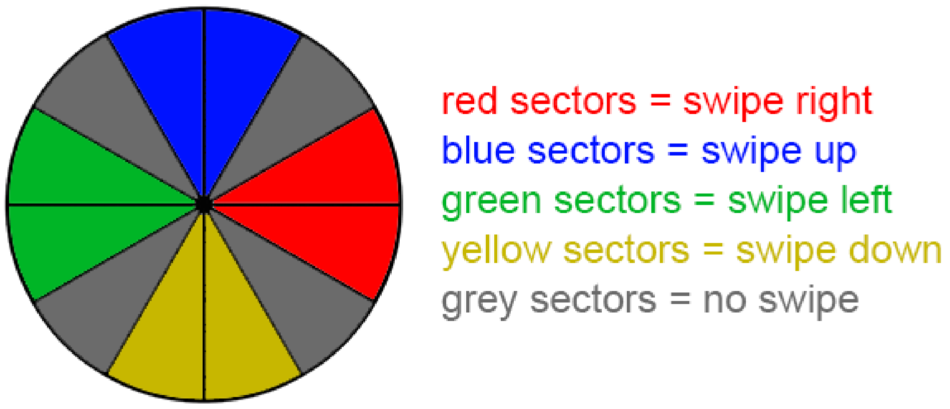
Any pointer movement shorter than 20 pixels won't be considered a swipe.

Keep in mind these are only my own options and you are free to use yours.

We are placing these values in `gameOptions` object with the precise purpose of being changed from you according to your own idea of swipe movement.

As for the direction, we said we only want to consider horizontal or vertical movements, excluding diagonal swipes.

We need a bit of tolerance because we can't force players to swipe with absolute precision as if they were surgeons, so we are going to detect direction this way:


red sectors = swipe right
blue sectors = swipe up
green sectors = swipe left
yellow sectors = swipe down
grey sectors = no swipe

If we split a circle into 12 sectors, each sector is a 30 degrees angle and wen assign each sector a swipe direction.

If we normalize the swipe vector, the horizontal movement of a left or right swipe ranges from cos(0) to cos(30), that is from 1 to 0.866.

In the same normalized vector, the vertical movement of an up or down swipe ranges from sin(90) to sin(60), that is from 1 to 0.866.

Now we can approximate 0.866 to 0.85 and say there is a valid swipe when the longest component of the normalized swipe vector measures at least 0.85.

Let's turn all these concepts into variables, and here is how `gameOptions` change:

```
var gameOptions = {
    tileSize: 200,
    tileSpacing: 20,
    boardSize: {
        rows: 4,
        cols: 4
    },
    tweenSpeed: 2000,
    swipeMaxTime: 1000,
    swipeMinDistance: 20,
    swipeMinNormal: 0.85
}
```

`swipeMaxTime` is the maximum amount of time allowed to swipe, in milliseconds.

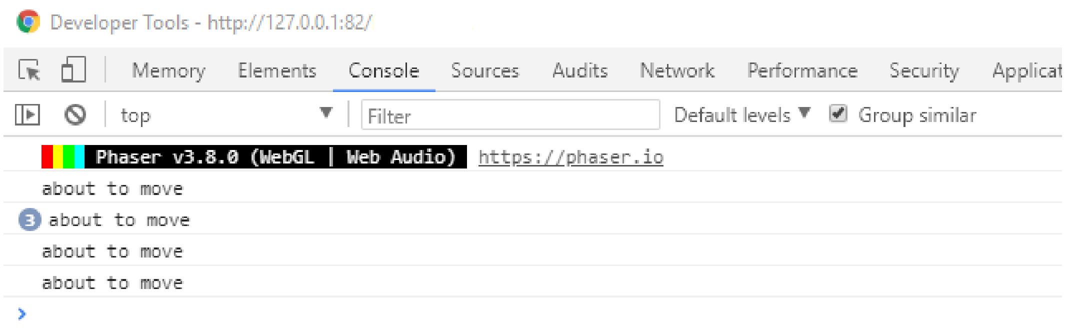
`swipeMinDistance` is the minimum swipe vector magnitude, in pixels.

`swipeMinNormal` is the minimum length of the longest component of the normalized swipe vector magnitude, in pixels.

Ready to build `handleSwipe` method according to all these options?

```
handleSwipe(e){
    if(this.canMove){
        var swipeTime = e.upTime - e.downTime;
        var fastEnough = swipeTime < gameOptions.swipeMaxTime;
        var swipe = new Phaser.Geom.Point(e.upX - e.downX, e.upY - e.downY);
        var swipeMagnitude = Phaser.Geom.Point.GetMagnitude(swipe);
        var longEnough = swipeMagnitude > gameOptions.swipeMinDistance;
        if(longEnough && fastEnough){
            Phaser.Geom.Point.SetMagnitude(swipe, 1);
            if(swipe.x > gameOptions.swipeMinNormal){
                this.makeMove(RIGHT);
            }
            if(swipe.x < -gameOptions.swipeMinNormal){
                this.makeMove(LEFT);
            }
            if(swipe.y > gameOptions.swipeMinNormal){
                this.makeMove(DOWN);
            }
            if(swipe.y < -gameOptions.swipeMinNormal){
                this.makeMove(UP);
            }
        }
    }
}
```

Now test the game and perform various pointer inputs, you should see the console window react accordingly:



```
Phaser v3.8.0 (WebGL | Web Audio) https://phaser.io
about to move
3 about to move
about to move
about to move
```

You'll get “about to move” only when you make an actual swipe gesture.

What happened? Let's split the code line by line.

As usual the whole code is executed if `canMove` property is `true`.

```
if(this.canMove){
    // rest of the code
}
```

You already know how to calculate swipe time, what you need to do now is compare with the maximum swipe time allowed in `gameOptions` object.

```
var swipeTime = e.upTime - e.downTime;
var fastEnough = swipeTime < gameOptions.swipeMaxTime;
```

The result of this comparison is saved in `fastEnough` Boolean variable.

Now we proceed with the creation of `swipe` variable which is a `Point` object containing the difference between the ending and the starting coordinates of the pointer input.

```
var swipe = new Phaser.Geom.Point(e.upX - e.downX, e.upY - e.downY);
```

And this is when magnitude comes into play:

```
var swipeMagnitude = Phaser.Geom.Point.GetMagnitude(swipe);
```

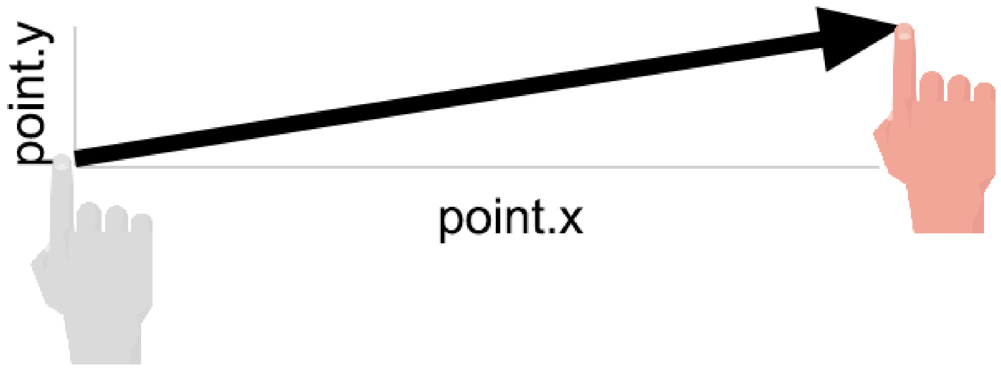
A variable called `swipeMagnitude` contains the magnitude of `swipe` point, that in the end is the amount of pixels traveled by the pointer.

> `GetMagnitude(point)` method of `Phaser.Geom.Point` class returns the magnitude of `point`, in pixels.

You are probably thinking there's something wrong with it, as we are dealing with a point and talking about magnitude which refers to vectors.

We can say sometimes `Point` objects can be used to represent vectors, if you don't consider their x and y values as horizontal and vertical coordinates but as horizontal and vertical components of a vector.

Phaser itself as you can see has plenty of `Point` methods which actually refer to vectors. Look at this picture:



It does really make sense if we split a vector into its horizontal and vertical components and save them in a Point object, and that's exactly what we are doing.

Then just like `fastEnough`, `longEnough` variable will let us know if the distance covered by the pointer input was long enough:

```
var longEnough = swipeMagnitude > gameOptions.swipeMinDistance;
```

We proceed only if `longEnough` and `fastEnough` are both true.

```
if(longEnough && fastEnough){
    // rest of the code
}
```

Once we know the pointer input was fast and covered some distance, we can say it could be a swipe.

> `&&` is the **and** logical operator used to connect two conditions. It returns `true` only when both conditions are true, and returns `false` when at least one of the conditions is false.

But it's not over yet. Do you remember? We are allowing only horizontal and vertical swipes, with some degrees of approximation.

We need to normalize swipe vector.

```
Phaser.Geom.Point.SetMagnitude(swipe, 1);
```

By setting its magnitude to `1` while keeping its angle, we normalized `swipe` vector.

`SetMagnitude(point, magnitude)` method of `Phaser.Geom.Point` class sets the magnitude of `point` to `magnitude` value. A value of `1` in `magnitude` normalizes `point`.

Now that magnitude is 1, we have to check if there is a component which is greater than `swipeMinNormal` and we finally managed to recognize a swipe:

```
if(swipe.x > gameOptions.swipeMinNormal){
    this.makeMove(RIGHT);
}
```

With the above code, we check if we are in the red sector of the picture showed a couple of pages ago, and therefore swiping right. We'll call the same makeMove method we used to do when processing keyboard input.

```
if(swipe.x < -gameOptions.swipeMinNormal){
    this.makeMove(LEFT);
}
```

Same concept is applied to check if we are in the green sector – left swipe – and the concept remains the same for vertical swipes.

You will find the full project in the folder

008 - Processing pointer input

Now that both input methods have been processed, it's time to move tiles on the board.

Moving tiles

Moving tiles is the core of the game, which actually is all about sliding tiles here and there, so this will be a long journey.

The rule of the game is: once the player moves, slide all tiles by one step according to the direction the player moved.

To tell the truth, this is not the rule of the game, but it's the first step we are going to develop: to move tiles by one step according to the direction the player moved.

Let's write the first version of `makeMove` method:

```
makeMove(d){
    var dRow = (d == LEFT || d == RIGHT) ? 0 : d == UP ? -1 : 1;
    var dCol = (d == UP || d == DOWN) ? 0 : d == LEFT ? -1 : 1;
    this.canMove = false;
    for(var i = 0; i < gameOptions.boardSize.rows; i++){
        for(var j = 0; j < gameOptions.boardSize.cols; j++){
            var curRow = dRow == 1 ? (gameOptions.boardSize.rows - 1) - i : i;
            var curCol = dCol == 1 ? (gameOptions.boardSize.cols - 1) - j : j;
            var tileValue = this.boardArray[curRow][curCol].tileValue;
            if(tileValue != 0){
                var newPos = this.getTilePosition(curRow + dRow, curCol + dCol);
                this.boardArray[curRow][curCol].tileSprite.x = newPos.x;
                this.boardArray[curRow][curCol].tileSprite.y = newPos.y;
            }
        }
    }
}
```

Just a few lines of code, but a lot of new concepts.

Let's start from the first two lines:

```
var dRow = (d == LEFT || d == RIGHT) ? 0 : d == UP ? -1 : 1;
var dCol = (d == UP || d == DOWN) ? 0 : d == LEFT ? -1 : 1;
```

To understand what these lines mean, we are going to introduce the conditional operator.

Understanding the conditional operator

You already know how to use the `if` statement to specify that a certain block of code should be executed it a certain condition is true.

Another block of code can be executed if the condition is not true thanks to `else`.

So let's suppose we have this script to choose between to movies to watch according to the age:

```
if(age >= 18){
    movieToWatch = “IT”;
}
else{
    movieToWatch = “The Lion King”;
}
```

It's very easy. If you are 18 or older, you'll watch “IT”, otherwise you'll watch “The Lion King”.

Thanks to the conditional operator, we can do the same exact thing in just one line of code, this way:

```
movieToWatch = (age >= 18) ? “IT” : “The Lion King”;
```

`age >= 18` is the condition you are testing.

`?` marks the beginning of the expression to evaluate if the condition is true.

`:` marks the beginning of the expression to evaluate if the condition is false.

A **conditional operator**, written as `condition ? expr1 : expr2` will return the value of `expr1` if `condition` is true, or the value of `expr2` if `condition` is false. Think about it as a short `if` statement like `if (condition) then expr1 else expr2`.

The conditional operator can also be called ternary operator because it has three operands.

Now let's have a look at the first line of `makeMove` method:

```
var dRow = (d == LEFT || d == RIGHT) ? 0 : d == UP ? -1 : 1;
```

You should notice the two nested conditional operators.

This line can also be written as:

```
var dRow;
if(d == LEFT || d == RIGHT){
    dRow = 0;
}
else{
    if(d == UP){
        dRow = -1;
    }
    else{
        dRow = 1;
    }
}
```

It's much better to handle `dRow` variable using conditional operators, isn't it?

> `||` is the **or** logical operator used to connect two conditions. It returns `true` if either condition is true, and returns `false` when both conditions are false.

At the end of the two conditional operators, `dRow` and `dCol` will have these values:

Moving left: `dRow = 0`, `dCol = -1`

Moving right: `dRow = 0`, `dCol = 1`

Moving up: `dRow = -1`, `dCol = 0`

Moving down: `dRow = 1`, `dCol = 0`

It's easy to see `dRow` and `dCol` represent the direction to move along rows and columns. When going left, there is no movement along rows – `dRow` is zero – and there is a decreasing movement along the column – `dCol` is -1.

The same concept applies to all directions.

Now that we know conditional operators, we can proceed with the code.

Keeping on moving tiles

`dRow` is the vertical direction where to move tiles. If we are moving left or right, `dRow` is `0`, because we are moving horizontally and we only move through columns.

If we are moving up, `dRow` is `-1` because we are moving up by one row. If we are moving down, `dRow` is 1 because we are moving down by one row.

The same concept applies to `dCol` which refers to the horizontal direction where to move tiles.

Now that are are about to move, we have to prevent the player to move while the game is already moving tiles:

```
this.canMove = false;
```

Then we need to iterate though the entire board:

```
for(var i = 0; i < gameOptions.boardSize.rows; i++){
    for(var j = 0; j < gameOptions.boardSize.cols; j++){
        // rest of the code
    }
}
```

According to the direction we are about to moving tiles, we will need to iterate through the board starting from different tiles.

If we are moving tiles to the right, we will start moving the rightmost tiles.

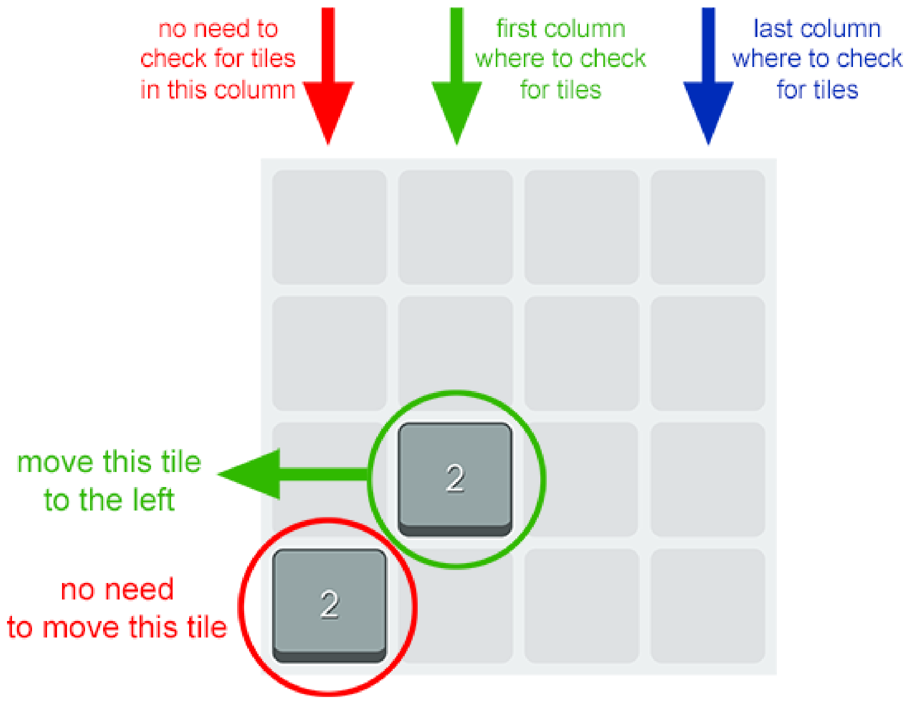
If we are moving tiles to the top, we will start moving the uppermost tiles, and the same concept applies when moving down or right.

Time to place another conditional operator:

```
var curRow = dRow == 1 ? (gameOptions.boardSize.rows - 1) - i : i;
var curCol = dCol == 1 ? (gameOptions.boardSize.cols - 1) - j : j;
```

To better understanding what we are trying to do, let's see what happens when we move tiles to the left.

Have a look at this picture:


no need to check for tiles in this column
first column where to check for tiles
last column where to check for tiles
move this tile to the left
no need to move this tile
2
2

Tiles in the red column – column zero – do not need to be moved to the left because they are already in the leftmost position.

We'll start checking and moving tiles in column one, then column two and so on, and the same concept then applies to other directions.

Back to the script, `curRow` and `curCol` represent respectively the current row and the current column to be examined at each iteration.

According to `dCol` and `dRow` values, `curCol` and `curRow` can range from zero to the amount of columns or rows minus one, or from the amount of rows or columns minus one down to zero.

Time for the final recap. we have four directions: left, right, up and down.

Each direction generates different `dRow` and `dCol` values, which represent the direction of the movement along the board.

When we loop through the table, `curRow` and `curCol` iterates board elements starting from the first row – or column – to the last one, or from the last row – or column – to the first one.

Direction	**dRow**	**dCol**	**curRow**	**curCol**
Left	0	-1	* first to last row	first to last column
Right	0	1	* first to last row	last to first column
Up	-1	0	first to last row	* first to last column
Down	1	0	last to first row	* first to last column

Values of `curRow` and `curCol` marked with an asterisk do not really matter as we are moving in a direction which does not affect row or column position of the tiles. Once we know which direction we are moving and which tile on the board we are going to move, let's see if we find a tile:

```
var tileValue = this.boardArray[curRow][curCol].tileValue;
```

Here we saved `tileValue` property of the current `boardArray` item into a variable called `tileValue`.

```
if(tileValue != 0){
    // rest of the code
}
```

The rest of the script will be executed only if `tileValue` is different than zero, that is if there is a tile in the current `boardArray` item.

`!=` is the **not equal** operator.

Given two values `x` and `y`, `x != y` returns `true` if `x` is not equal to `y`

We enter the `if` statement block when there is a tile in the current `boardArray` item so we need to update its position.

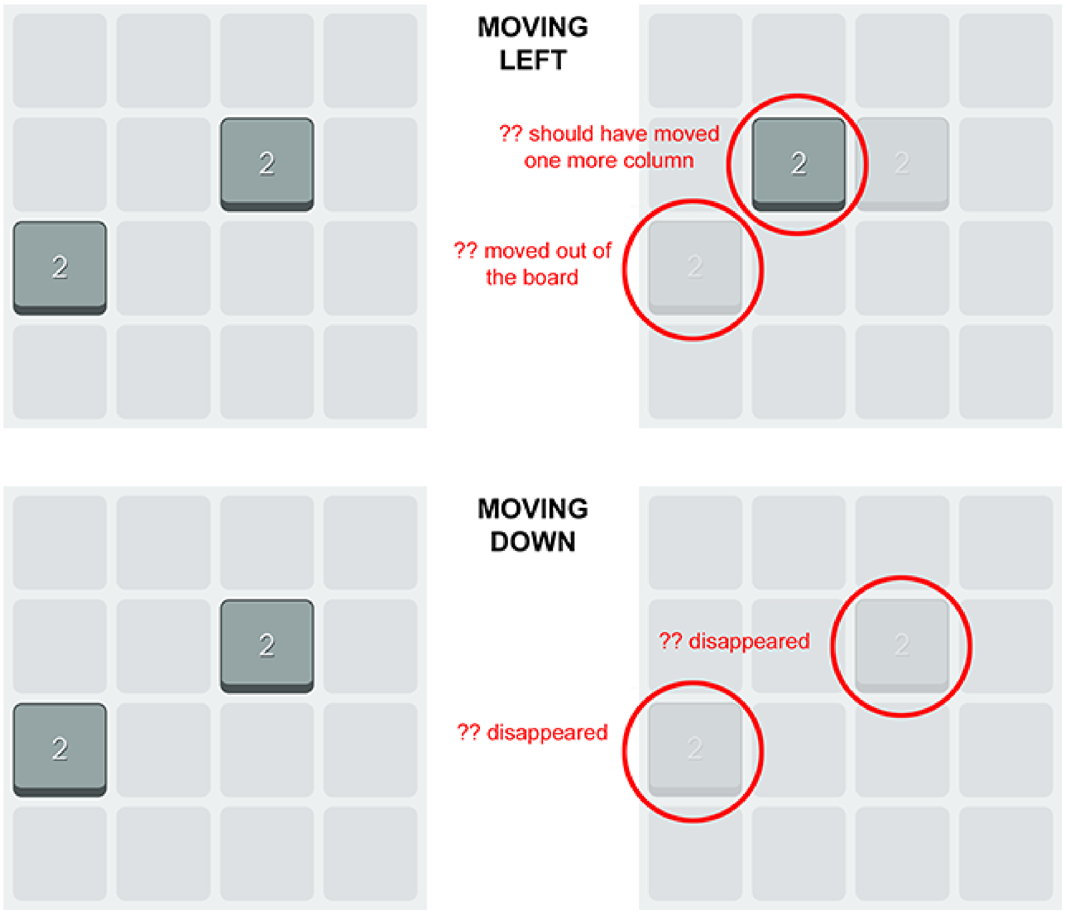
```
var newPos = this.getTilePosition(curRow + dRow, curCol + dCol);
this.boardArray[curRow][curCol].tileSprite.x = newPos.x;
this.boardArray[curRow][curCol].tileSprite.y = newPos.y;
```

`newPos` is a `Point` object which will contain the new tile position.

We are using `getTilePosition` method to determine it, and its arguments are the current row and column positions – `curRow` and `curCol` – added to respectively the row and column directions where to move – `dRow` and `dCol`.

Then `tileSprite` sprite position is updated to `newPos.x` and `newPos.y`.

Run the game, and make some moves, you'll see some bugs. Can you find them all? Here they are:


MOVING LEFT
?? should have moved one more column
?? moved out of the board
MOVING DOWN
?? disappeared
?? disappeared
2

Tiles move only by one row or one column even if there is more room for movement.

Tiles can be moved outside the board.

Moving down or right will cause tiles to disappear no matter their positions.

You can move only once.

Don't worry, we'll fix them all.

Before we continue, let's set `tweenSpeed` from 2000 to 200, to make animations run faster.

```
var gameOptions = {
    tileSize: 200,
    tileSpacing: 20,
    boardSize: {
        rows: 4,
        cols: 4
    },
    tweenSpeed: 200,
    swipeMaxTime: 1000,
    swipeMinDistance: 20,
    swipeMinNormal: 0.85
}
```

With this new tween speed, the game will be a bit more playable, and you won't have to wait two seconds for tiles to appear on the board.

> You will find the full project in the folder
>
> **009 - Moving tiles**

Now we'll talk about depth, or Z-order.

Understanding depth or Z-order

Although we are building a 2D game, where we are only allowed to place objects on coordinates based on x and y axis, there's something to say about Z-order.

In 2D environments, objects can overlap, and when there are overlapping objects, there is always one object which covers other objects.

Think about Photoshop layers. Top layers will cover bottom layers. Or think about overlapping windows on your desktop, where top windows cover bottom windows, until you bring them to front.

It's obvious even in 2D environments there is some kind of depth: the Z-order.

> **Z-order** is an ordering of overlapping two-dimensional objects, such as windows in a stacking window manager, or shapes in a graphics editor.
>
> When objects overlap, objects with a higher Z value hide part or all of objects with a lower Z value.

If you are used to web design, it's the same concept of CSS `z-index` property, which specifies the stack order of an element.

When we placed the images and sprites representing empty spots and tiles during the creation of the game table, the order we placed the images represents the Z-order used by Phaser to display objects.

This way, moving from left to right and from top to bottom, Z-order increased, and that's why when you move a tile to the right or to the bottom of the canvas, it disappears: it's simply rendered behind other objects which have been placed later, obtaining a higher Z value.

To avoid this issue, we need to group all moving objects and change their Z value – which Phaser calls "depth" – according to the direction they are moving.

This will be useful not only when we are simply moving tiles down and right, but also when we need two tiles with the same value to merge and transform into a tile with a greater value.

How to handle Z-order with Phaser? In just three lines of code to be added to `makeMove` method:

```
makeMove(d){
    var dRow = (d == LEFT || d == RIGHT) ? 0 : d == UP ? -1 : 1;
    var dCol = (d == UP || d == DOWN) ? 0 : d == LEFT ? -1 : 1;
    this.canMove = false;
    var movedTiles = 0;
    for(var i = 0; i < gameOptions.boardSize.rows; i++){
        for(var j = 0; j < gameOptions.boardSize.cols; j++){
            var curRow = dRow == 1 ? (gameOptions.boardSize.rows - 1) - i : i;
            var curCol = dCol == 1 ? (gameOptions.boardSize.cols - 1) - j : j;
            var tileValue = this.boardArray[curRow][curCol].tileValue;
            if(tileValue != 0){
                movedTiles ++;
                this.boardArray[curRow][curCol].tileSprite.depth = movedTiles;
                var newPos = this.getTilePosition(curRow + dRow, curCol + dCol);
                this.boardArray[curRow][curCol].tileSprite.x = newPos.x;
                this.boardArray[curRow][curCol].tileSprite.y = newPos.y;
            }
        }
    }
}
```

When placed on the stage, all sprites have a `depth` property initially set to zero.

At the beginning of the function, we can declare a new variable called `movedTiles` which will keep count of the tiles we moved.

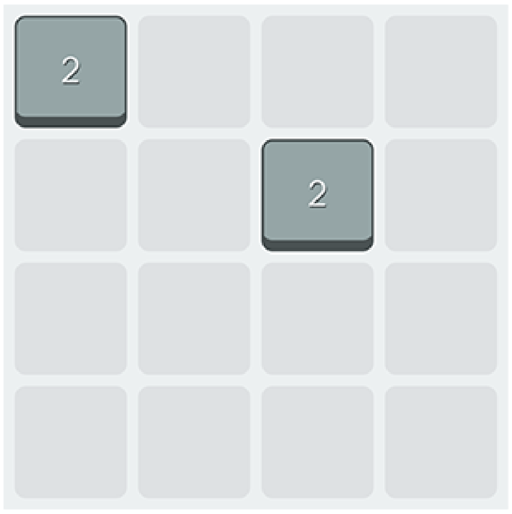
It starts at zero because no tiles have been moved yet.

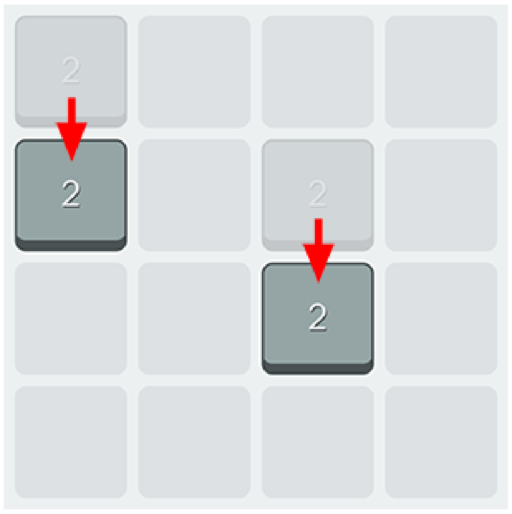
When we are about to move a tile, first we increment `movedTiles`, then we change `depth` property of the sprite we are about to move to `movedTiles`.

This way, depth increases as we move the tiles so that the last to be moved have a higher value and stay on the top.

> `depth` property of a game object sets its depth within the Scene, allowing to change the rendering order.
>
> It starts from zero and a game object with a higher depth value will always render in front of one with a lower value.

Now test the game and try to move tiles to the right or to the bottom, and you will see they will display correctly.





And the first bug has been solved.

You will find the full project in the folder

010 - Understanding depth or Z-order

Now we'll fix the bug which causes tiles to move outside the board.

Moving only necessary tiles

Do you know why sometimes tiles disappear from the stage? They disappear because they already are on an edge of the stage, and when we move tiles towards such edge, they simply fly off the stage.

We are moving unnecessary tiles.

When we are moving left, we do not need to move tiles in the leftmost column. They are already in the leftmost position, so why bothering moving them?

Same concept applies when moving up. We don't need to move tiles in the upmost row. And that's the same when you move right, or down.

According to the direction you are moving, there is one row or one column you don't need to move. And you won't move it, thanks to these lines added to `makeMove` method:

```
makeMove(d){
    var dRow = (d == LEFT || d == RIGHT) ? 0 : d == UP ? -1 : 1;
    var dCol = (d == UP || d == DOWN) ? 0 : d == LEFT ? -1 : 1;
    this.canMove = false;
    var movedTiles = 0;
    var firstRow = (d == UP) ? 1 : 0;
    var lastRow = gameOptions.boardSize.rows - ((d == DOWN) ? 1 : 0);
    var firstCol = (d == LEFT) ? 1 : 0;
    var lastCol = gameOptions.boardSize.cols - ((d == RIGHT) ? 1 : 0);
    for(var i = firstRow; i < lastRow; i++){
        for(var j = firstCol; j < lastCol; j++){
            var curRow = dRow == 1 ? (lastRow - 1) - i : i;
            var curCol = dCol == 1 ? (lastCol - 1) - j : j;
            var tileValue = this.boardArray[curRow][curCol].tileValue;
            if(tileValue != 0){
                movedTiles ++;
                this.boardArray[curRow][curCol].tileSprite.depth = movedTiles;
                var newPos = this.getTilePosition(curRow + dRow, curCol + dCol);
                this.boardArray[curRow][curCol].tileSprite.x = newPos.x;
                this.boardArray[curRow][curCol].tileSprite.y = newPos.y;
            }
        }
    }
}
```

Some more conditional operators here, let's split the code line by line and see how we avoid moving unnecessary tiles.

Both in the two `for` loops and when calculating `curRow` and `curCol` values, we do not refer anymore to `boardSize.rows` and `boardSize.cols` stored into `gameOptions` object, because we need to exclude one row or one column, let's see how we are doing it:

```
var firstRow = (d == UP) ? 1 : 0;
```

`firstRow` variable represents the highest row we have to move. Its value is zero – the top row – unless we are moving up, and in this case its value is 1, because we don't need to move tiles in the topmost row.

```
var lastRow = gameOptions.boardSize.rows - ((d == DOWN) ? 1 : 0);
```

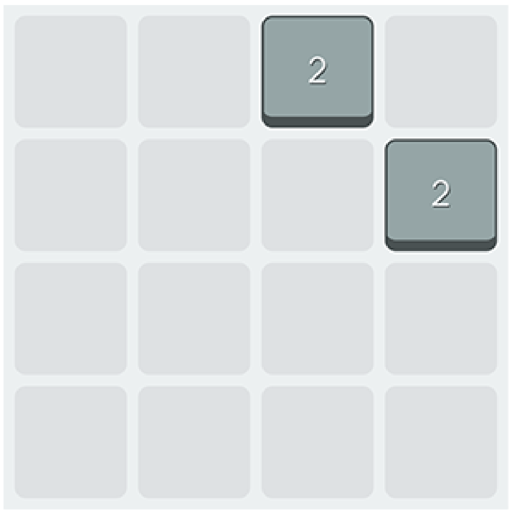
`lastRow` variable represents the lowest row we have to move. Its value is `boardSize.rows` unless we are moving down, in this case its value is `boardSize.rows – 1` because we do not need to move tiles in the bottommost row.

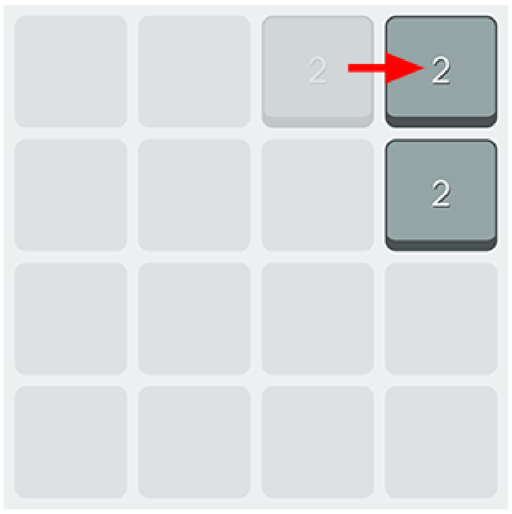
The same concept applies to the columns, and this is a comparison table resuming what happens to `firstRow`, `firstCol`, `lastRow` and `lastCol` variables according to the direction we are moving:

Direction	**firstRow**	**firstCol**	**lastRow**	**lastCol**
Left	0	**1**	boardSize.rows	boardSize.cols
Right	0	0	boardSize.rows	**boardSize.cols - 1**
Up	**1**	0	boardSize.rows	boardSize.rows
Down	0	0	**boardSize.rows - 1**	boardSize.rows

Highlighted values represent what changed in the two `for` loops and when calculating `curRow` and `curCol` values compared to before, when we simply looped from zero to `boardSize.rows` – or `boardSize.cols` – no matter the direction of movement.

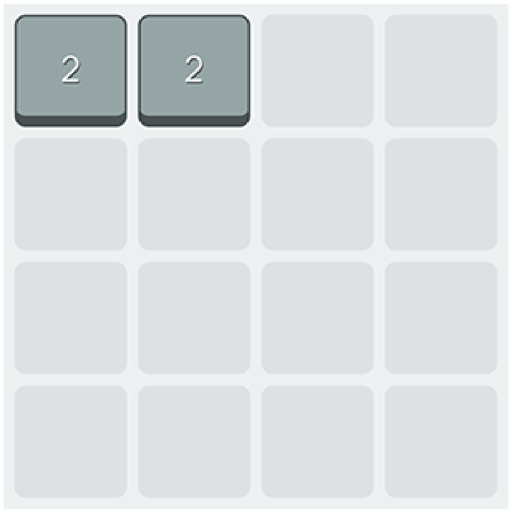
Let's run the game and try to move in a direction which previously would have resulted in a tile to disappear, like in this case when I move right:

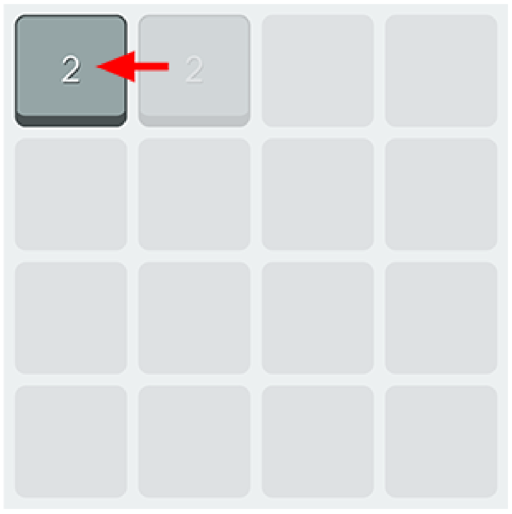




Rightmost tile did not disappear, while the other tile moved as usual.

We fixed this bug too, but this leaded to a new bug. Try to move in a way two tiles should merge in a greater tile, like in this case when I move left:





Tiles do not disappear, but simply overlap rather than merging into a greater tile.

It's a feature we still have to develop, so don't worry about it, but it's important to update the list of bugs and missing features, which now includes:

Tiles move only by one row/column rather than running through the whole board.

You can move only once.

Tiles do not merge.

> You will find the full project in the folder
>
> **011 - Moving only necessary tiles**

We are about to see how to move tiles properly.

Moving tiles as long as there is room for movement

To make tiles move until there is not any more room for movement rather than for just one step, we need to move them in the selected direction as long as they remain inside the board.

This can be easily done in just a few lines to be added to `makeMove` method:

```
makeMove(d){
    var dRow = (d == LEFT || d == RIGHT) ? 0 : d == UP ? -1 : 1;
    var dCol = (d == UP || d == DOWN) ? 0 : d == LEFT ? -1 : 1;
    this.canMove = false;
    var movedTiles = 0;
    var firstRow = (d == UP) ? 1 : 0;
    var lastRow = gameOptions.boardSize.rows - ((d == DOWN) ? 1 : 0);
    var firstCol = (d == LEFT) ? 1 : 0;
    var lastCol = gameOptions.boardSize.cols - ((d == RIGHT) ? 1 : 0);
    for(var i = firstRow; i < lastRow; i++){
        for(var j = firstCol; j < lastCol; j++){
            var curRow = dRow == 1 ? (lastRow - 1) - i : i;
            var curCol = dCol == 1 ? (lastCol - 1) - j : j;
            var tileValue = this.boardArray[curRow][curCol].tileValue;
            if(tileValue != 0){
                var newRow = curRow;
                var newCol = curCol;
                while(this.isLegalPosition(newRow + dRow, newCol + dCol)){
                    newRow += dRow;
                    newCol += dCol;
                }
                movedTiles ++;
                this.boardArray[curRow][curCol].tileSprite.depth = movedTiles;
                var newPos = this.getTilePosition(newRow, newCol);
                this.boardArray[curRow][curCol].tileSprite.x = newPos.x;
                this.boardArray[curRow][curCol].tileSprite.y = newPos.y;
            }
        }
    }
}
```

We first create two new variables, `newRow` and `newCol`, which initially take the values respectively of `curRow` and `curCol`.

Then we keep adding `dRow` and `dCol` values to `newRow` and `newCol` until we are sure tiles are still in a legal position.

The code will look more clear once you see how we are using the `while` loop.

```
while(condition){
    code block to be executed
}
```

Executes the block of code as long as a condition is true.

condition defines the condition for running the loop. If it returns `true`, the loop will start over again, if it returns `false`, the loop will end.

If the condition is always true, the loop will never end, your script will probably freeze and crash your browser.

Basically we are acting as if the player moved several times in the same direction, while next move is in a legal position, determined by `isLegalPosition` method we are about to write:

```
isLegalPosition(row, col){
    var rowInside = row >= 0 && row < gameOptions.boardSize.rows;
    var colInside = col >= 0 && col < gameOptions.boardSize.cols;
    return rowInside && colInside;
}
```

isLegalPosition sees if its arguments – the row and the column we are checking to be in a legal position – are inside the board.

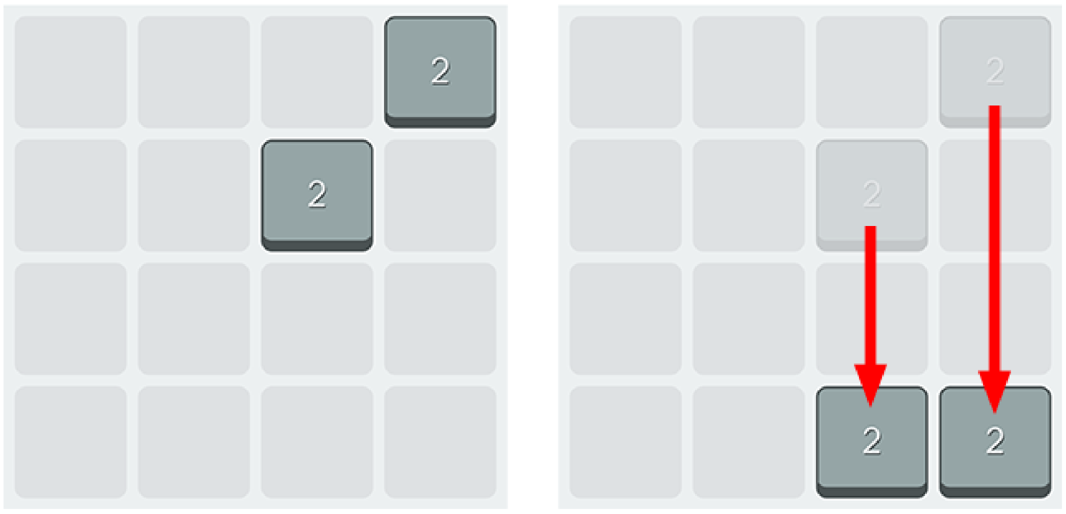
A row or a column are inside the board if they are greater than or equal to zero, and smaller than the number of rows or columns.

`rowInside` is a Boolean variable which is `true` if `row` argument is greater than or equal to zero and smaller than `boardSize.rows` value defined in `gameOptions` global variable.

`colInside` is a Boolean variable which is `true` if `col` argument is greater than or equal to zero and smaller than `boardSize.cols` value defined in `gameOptions` global variable.

Finally, the whole method returns `true` only if both `rowInside` and `colInside` have a `true` value.

Run the game and make your move, and see how tiles travel along the entire table before stopping on one of its edges, like in this case when the player is moving down.



Another feature has been added.

> You will find the full project in the folder
>
> **012 - Moving tiles as long as there is room for movement**

Tired of seeing only tiles with a “2” on them? We are about to merge them.

Merging tiles

Two tiles merge in a bigger tile when two tiles with the same value overlap.

In previous steps we only moved sprites here and there, this time we'll have to work behind the curtains – remember? Where most of the game takes part – reading and changing some `boardArray` values.

Some more lines to be added to `makeMove` method:

```
makeMove(d){
    var dRow = (d == LEFT || d == RIGHT) ? 0 : d == UP ? -1 : 1;
    var dCol = (d == UP || d == DOWN) ? 0 : d == LEFT ? -1 : 1;
    this.canMove = false;
    var movedTiles = 0;
    var firstRow = (d == UP) ? 1 : 0;
    var lastRow = gameOptions.boardSize.rows - ((d == DOWN) ? 1 : 0);
    var firstCol = (d == LEFT) ? 1 : 0;
    var lastCol = gameOptions.boardSize.cols - ((d == RIGHT) ? 1 : 0);
    for(var i = firstRow; i < lastRow; i++){
        for(var j = firstCol; j < lastCol; j++){
            var curRow = dRow == 1 ? (lastRow - 1) - i : i;
            var curCol = dCol == 1 ? (lastCol - 1) - j : j;
            var tileValue = this.boardArray[curRow][curCol].tileValue;
            if(tileValue != 0){
                var newRow = curRow;
                var newCol = curCol;
                while(this.isLegalPosition(newRow + dRow, newCol + dCol)){
                    newRow += dRow;
                    newCol += dCol;
                }
                movedTiles ++;
                this.boardArray[curRow][curCol].tileSprite.depth = movedTiles;
                var newPos = this.getTilePosition(newRow, newCol);
                this.boardArray[curRow][curCol].tileSprite.x = newPos.x;
                this.boardArray[curRow][curCol].tileSprite.y = newPos.y;
                this.boardArray[curRow][curCol].tileValue = 0;
                if(this.boardArray[newRow][newCol].tileValue == tileValue){
                    this.boardArray[newRow][newCol].tileValue ++;
                    this.boardArray[curRow]
                       [curCol].tileSprite.setFrame(tileValue);
                }
                else{
                    this.boardArray[newRow][newCol].tileValue = tileValue;
                }
            }
        }
    }
}
```

Before explaining the new code, let's make clear we moved tiles from row `curRow` and column `curCol` to row `newRow` and column `newCol`. As we move tiles, we save their values in `tileValue` variable

So the first line of the new code:

```
this.boardArray[curRow][curCol].tileValue = 0;
```

Sets `tileValue` value of `boardArray[curRow][curCol]` item to zero, because we said the tile moved from rom row `curRow` and column `curCol`.

```
if(this.boardArray[newRow][newCol].tileValue == tileValue){
    // rest of the code
}
```

Next thing to do is to see if in the landing spot at row `newRow` and column `newCol` there is already a tile with the same value as the tile we just moved.

```
this.boardArray[newRow][newCol].tileValue ++;
```

If this is the case, `tileValue` value of `boardArray[newRow][newCol]` item is increased because the destination tile now contains a merged tile.

```
this.boardArray[curRow][curCol].tileSprite.setFrame(tileValue);
```

And we also show the proper frame of `boardArray[curRow][curCol]` sprite.

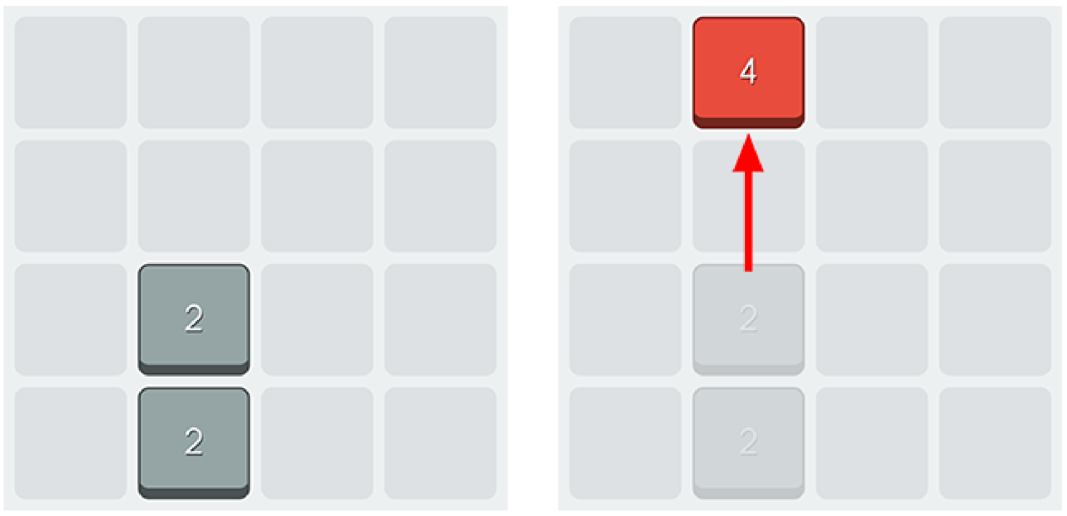
Why are we changing the frame of `boardArray[curRow][curCol]` sprite and not the frame of `boardArray[newRow][newCol]` sprite?

Because we moved the sprite of `boardArray[curRow][curCol]` item, which also has a greater depth then the one at the destination tile.

```
else{
    this.boardArray[newRow][newCol].tileValue = tileValue;
}
```

If we did not find a destination tile with the same value as the moving tile, we simply set `tileValue` value of `boardArray[newRow][newCol]` item to tileValue.

Test the game and try to make a move to merge two tiles:



The magic happened! Moving up, we managed to turn two “2” tiles into one “4”.

You will find the full project in the folder

013 - Merging tiles

The core of the game is almost completed, what about moving more than just once?

Keeping on moving

At the beginning of this project, we talked about the importance of adding as less sprites as possible during the game, and we already added all necessary images, saying we were going to show or hide them according to board status.

This is what we are doing now: once all tiles have been moved/merged, we refresh the board moving back all tiles in their original position, then showing or hiding them according to board status, which is just a visual representation of `boardArray` array.

At the end of `makeMove` method, we'll call a new method called `refreshBoard`: this will redraw the board and make it ready for next move.

```
makeMove(d){
    // same as before
    this.refreshBoard();
}
```

And this is `refreshBoard` content:

```
refreshBoard(){
    for(var i = 0; i < gameOptions.boardSize.rows; i++){
        for(var j = 0; j < gameOptions.boardSize.cols; j++){
            var spritePosition = this.getTilePosition(i, j);
            this.boardArray[i][j].tileSprite.x = spritePosition.x;
            this.boardArray[i][j].tileSprite.y = spritePosition.y;
            var tileValue = this.boardArray[i][j].tileValue;
            if(tileValue > 0){
                this.boardArray[i][j].tileSprite.visible = true;
                this.boardArray[i][j].tileSprite.setFrame(tileValue - 1);
            }
            else{
                this.boardArray[i][j].tileSprite.visible = false;
            }
        }
    }
    this.addTile();
}
```

We need to loop through the entire table, and adjust sprite positions.

The first two `for` loops iterate through all row and columns in the same way you already saw several times during the creation of the game.

```
var spritePosition = this.getTilePosition(i, j);
```

`spritePosition` variable gets the result of `getTilePosition` method which returns a `Point` object with the coordinates of tile default position according to its row and column.

```
this.boardArray[i][j].tileSprite.x = spritePosition.x;
this.boardArray[i][j].tileSprite.y = spritePosition.y;
```

And here we placed the tile in its proper place.

```
var tileValue = this.boardArray[i][j].tileValue;
```

`tileValue` variable gets the value of `boardArray`'s current item.

```
if(tileValue > 0){
    this.boardArray[i][j].tileSprite.visible = true;
    this.boardArray[i][j].tileSprite.setFrame(tileValue - 1);
}
```

if `tileValue` is greater than zero, that is we found a tile on the that spot, we show its sprite setting `visible` property to `true` and we set its frame with `setFrame` method.

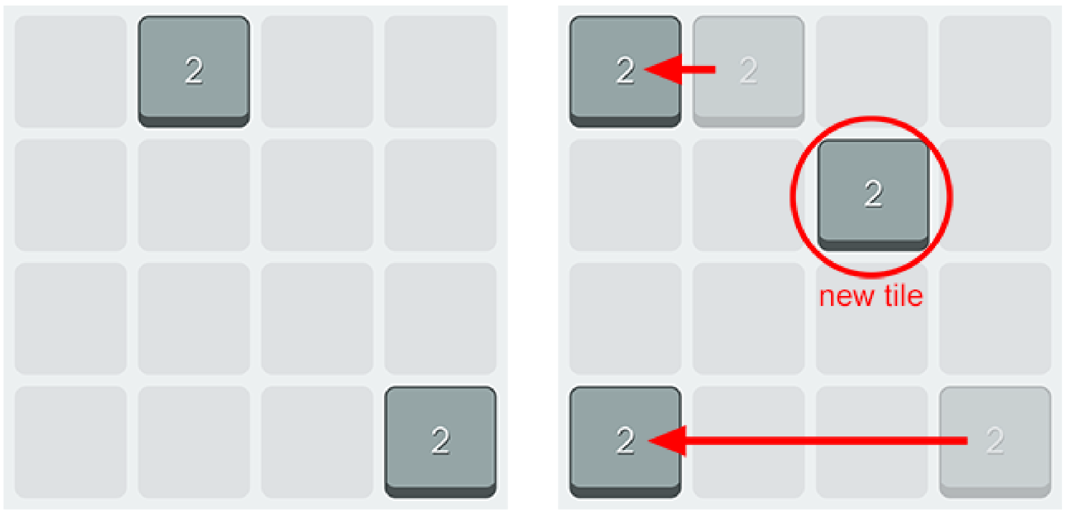
```
else{
    this.boardArray[i][j].tileSprite.visible = false;
}
```

If `tileValue` wasn't greater than zero, this means we didn't find a tile on that spot and simply hide the sprite.

```
this.addTile();
```

At the end of both for loops, once the board has been completely refreshed, we call `addTile` method to add a new “2” tile on the board. `addTile` also sets `canMove` to `true` so the player can move again.

Test the game:



In this case I moved to the left, tiles slid to the left and a new tile appeared, allowing me to move again.

Anyway, it should take only a few moves to see there's something wrong.

Tiles do no match properly and seem to disappear.

This happens because we didn't write the code to handle tile collisions: tiles with the same value have to overlap, but tiles with different values must stop.

But the good news is we are a few steps away from the complete game.

> You will find the full project in the folder
>
> **014 - Keeping on moving**

And now let's make tiles move properly.

Moving tiles according to game rules

According to game rules, a tile keeps moving in a direction until there is room for movement, and there is room for movement when the destination spot is empty or filled by a tile with the same value.

A couple of changes to `isLegalPosition` method will update the code according to actual game rules:

```
isLegalPosition(row, col, value){
    var rowInside = row >= 0 && row < gameOptions.boardSize.rows;
    var colInside = col >= 0 && col < gameOptions.boardSize.cols;
    if(!rowInside || !colInside){
        return false;
    }
    var emptySpot = this.boardArray[row][col].tileValue == 0;
    var sameValue = this.boardArray[row][col].tileValue == value;
    return emptySpot || sameValue;
}
```

First, we added an argument. `value` argument takes the value of the currently moving tile.

You already know how `rowInisde` and `colInside` Boolean variables work, and we return `false` and stop the execution if `rowInside` is `false` or `colInside` is `false`.

> `!` is the **not** logical operator and returns `false` if its single operand is `true`, or `true` otherwise.

`emptySpot` is a new Boolean variable which is `true` when the current board position has no tile on it – its `tileValue` value is zero.

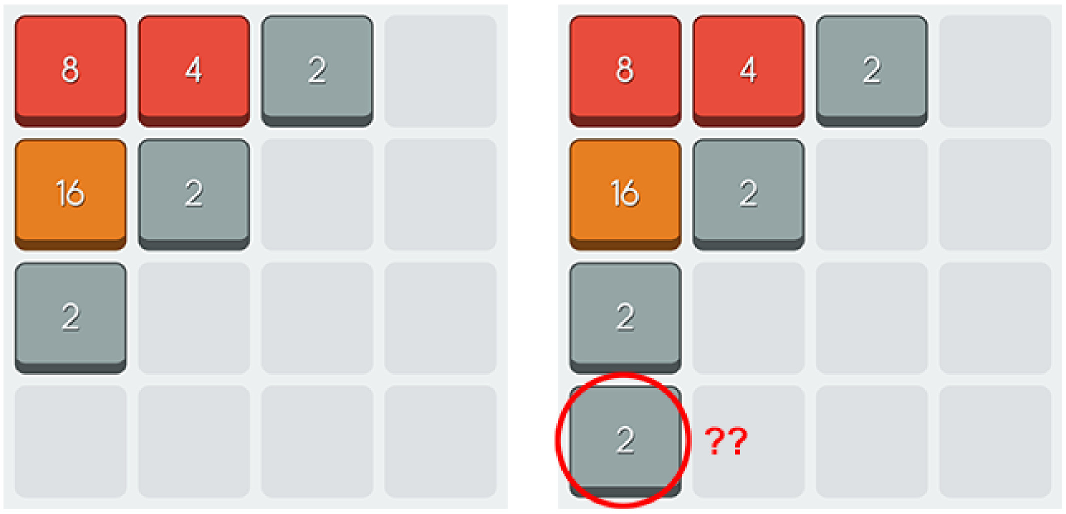
`sameValue` is another new Boolean variable which is `true` when the current board position as a tile of the same value as the tile we are moving – its `tileValue` value is equal to `value` argument.

This way, we can say we are in a legal position when we are inside the board and when we are on an empty spot or on a spot with a tile with the same value as the tile we are moving.

Since we added an argument to `isLegalPosition` method, we also have to update the way we call it inside `makeMove` method, and this is how we change the way we call it inside the condition in the `while` statement:

```
makeMove(d){
    // same as before
    while(this.isLegalPosition(newRow + dRow, newCol + dCol, tileValue)){
        newRow += dRow;
        newCol += dCol;
    }
    // same as before
}
```

Now play the game, and suddenly everything should work fine, unless you make a void move. We call it a void move when you make a move which actually does not move any tile, like in this picture when you move up:



If moving up we did not move anything, why did a new tile appear?

You will find the full project in the folder

015 - Moving tiles according to game rules

How to check if the player actually moved something on the board?

Checking if the player actually moved

The routine to check if the player actually moved is build around a Boolean variable called `movedSomething` inside `makeMove` method which starts with a `false` value because at the beginning of each move we did not move anything.

```
makeMove(d){
    // same as before
    var movedSomething = false;
    for(var i = firstRow; i < lastRow; i++){
        for(var j = firstCol; j < lastCol; j++){
            var curRow = dRow == 1 ? (lastRow - 1) - i : i;
            var curCol = dCol == 1 ? (lastCol - 1) - j : j;
            var tileValue = this.boardArray[curRow][curCol].tileValue;
            if(tileValue != 0){
                var newRow = curRow;
                var newCol = curCol;
                while(this.isLegalPosition(newRow + dRow, newCol + dCol,
                      tileValue)){
                    newRow += dRow;
                    newCol += dCol;
                }
                movedTiles ++;
                if(newRow != curRow || newCol != curCol){
                    movedSomething = true;
                    this.boardArray[curRow][curCol].tileSprite.depth = movedTiles;
                    var newPos = this.getTilePosition(newRow, newCol);
                    this.boardArray[curRow][curCol].tileSprite.x = newPos.x;
                    this.boardArray[curRow][curCol].tileSprite.y = newPos.y;
                    this.boardArray[curRow][curCol].tileValue = 0;
                    if(this.boardArray[newRow][newCol].tileValue == tileValue){
                        this.boardArray[newRow][newCol].tileValue ++;
                        this.boardArray[curRow]
                              [curCol].tileSprite.setFrame(tileValue);
                    }
                    else{
                        this.boardArray[newRow][newCol].tileValue = tileValue;
                    }
                }
            }
        }
    }
    if(movedSomething){
        this.refreshBoard();
    }
    else{
        this.canMove = true;
    }
}
```

We'll declare this new variable just before the two `for` loops iterating through the game board.

How can we see if we actually moved a tile? When at the end of the `while` statement `newRow` is different than `curRow` – the tile moved by at least one row – or `newCol` is different than `curCol` – the tile moved by at least one column, and this is how we check it:

```
if(newRow != curRow || newCol != curCol){
    movedSomething = true;
    // rest of the code
}
```

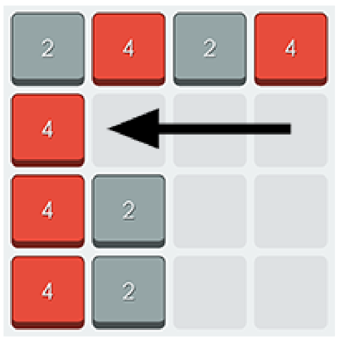
movedSomething is set to `true` after we checked if the tile moved by at least one row or at least one column.

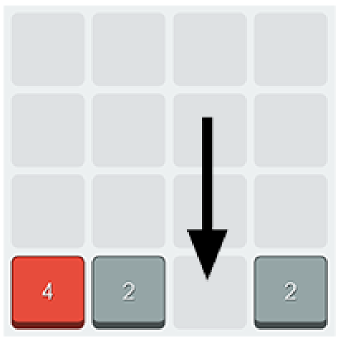
The part of the script which changes tile position and adjusts `boardArray` values has also been placed inside the `if` statement as there is no point in moving a tile and adjusting `boardArray` values if nothing changed.

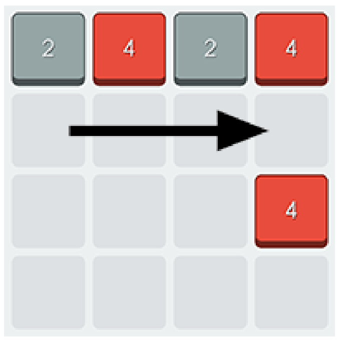
```
if(movedSomething){
    this.refreshBoard();
}
else{
    this.canMove = true;
}
```

Then we refresh the board only if `movedSomething` is `true`, or we just set `canMove` to `true` if we did not move anything, waiting for next player move.

Run the game, and if your move actually does not move anything – like in the pictures below when you try to move left, down or right – nothing happens.



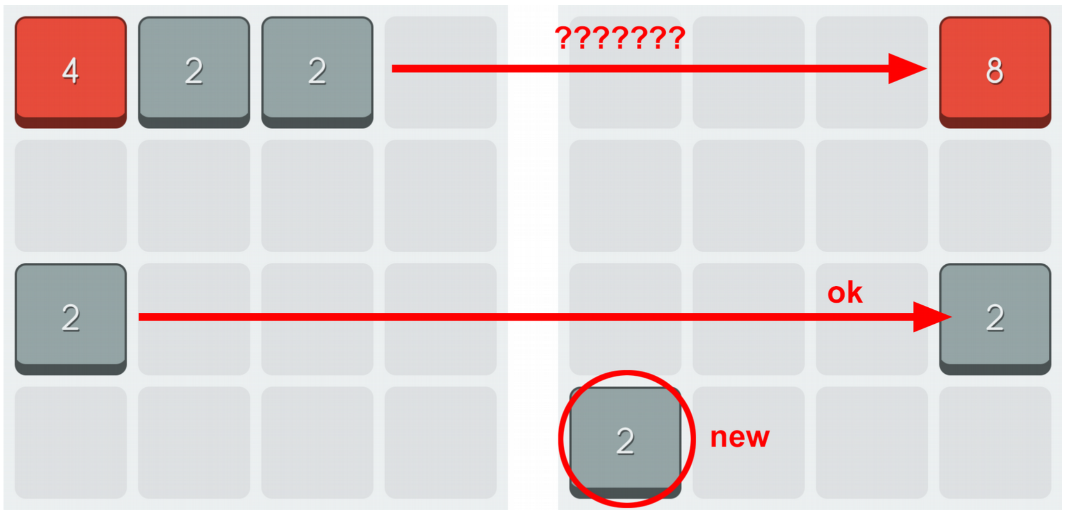




Did we finally completed the game logic? Almost.

There is one more case we need to handle.

Look at this picture:



When we move right, the “2” on the third row moves to the rightmost tile, a new “2” appears in the fourth row, but look what happens in the first row: the two “2” merged in to a “4”, which merges with the other moving “4” resulting in an “8” tile.

From a certain point of view there's nothing wrong, as tiles merged correctly, but in the original game updated tiles can't update again in the same turn.

The first row should have two “4” in it, not one “8”.

This is what we are going to fix now.

> You will find the full project in the folder
>
> **016 - Checking if the player actually moved**

One last little issue to fix.

Upgrading tiles only once per turn

To let the player upgrade tiles only once per turn, we need to add some lines of code in almost every method of `playGame` class.

The idea is to add a Boolean flag to each `boardArray` item, which we can call `upgraded`, whose default value is `false`, like we are doing in `create` method when it's time to create the object in `boardArray` array.

```
create(){
    // same as before
    this.boardArray[i][j] = {
        tileValue: 0,
        tileSprite: tile,
        upgraded: false
    }
    // same as before
}
```

Now every tile starts with its `upgrade` value to `false`, because at the beginning of the game no tiles have been upgraded.

When does `upgraded` get a `true` value? When it's merged with another tile.

```
makeMove(){
    // same as before
    if(newRow != curRow || newCol != curCol){
        movedSomething = true;
        this.boardArray[curRow][curCol].tileSprite.depth = movedTiles;
        var newPos = this.getTilePosition(newRow, newCol);
        this.boardArray[curRow][curCol].tileSprite.x = newPos.x;
        this.boardArray[curRow][curCol].tileSprite.y = newPos.y;
        this.boardArray[curRow][curCol].tileValue = 0;
        if(this.boardArray[newRow][newCol].tileValue == tileValue){
            this.boardArray[newRow][newCol].tileValue ++;
            this.boardArray[newRow][newCol].upgraded = true;
            this.boardArray[curRow][curCol].tileSprite.setFrame(tileValue);
        }
        else{
            this.boardArray[newRow][newCol].tileValue = tileValue;
        }
    }
    // same as before
}
```

In `makeMove` method, inside the routine which checks if we moved a tile, we set `upgraded` to `true` when we execute the block of code which handles tile merging.

At this time the code works this way: `upgraded` starts with `false` value, then when two tiles merge, the destination tile has `upgraded` value set to `true`.

At the end of each move, when it's time to refresh the board, all `boardArray` items will have their `upgraded` values set back to `false`, ready for the player to move again.

We are doing it in `refreshBoard` method:

```
refreshBoard(){
    for(var i = 0; i < gameOptions.boardSize.rows; i++){
        for(var j = 0; j < gameOptions.boardSize.cols; j++){
            var spritePosition = this.getTilePosition(i, j);
            this.boardArray[i][j].tileSprite.x = spritePosition.x;
            this.boardArray[i][j].tileSprite.y = spritePosition.y;
            var tileValue = this.boardArray[i][j].tileValue;
            if(tileValue > 0){
                this.boardArray[i][j].tileSprite.visible = true;
                this.boardArray[i][j].tileSprite.setFrame(tileValue - 1);
                this.boardArray[i][j].upgraded = false;
            }
            else{
                this.boardArray[i][j].tileSprite.visible = false;
            }
        }
    }
    this.addTile();
}
```

And we are done with setting `upgraded` values. How are we going to use it? To determine if a tile is in a legal position, updating `isLegalPosition` method this way:

```
isLegalPosition(row, col, value){
    var rowInside = row >= 0 && row < gameOptions.boardSize.rows;
    var colInside = col >= 0 && col < gameOptions.boardSize.cols;
    if(!rowInside || !colInside){
        return false;
    }
    var emptySpot = this.boardArray[row][col].tileValue == 0;
    var sameValue = this.boardArray[row][col].tileValue == value;
    var alreadyUpgraded = this.boardArray[row][col].upgraded;
    return emptySpot || (sameValue && !alreadyUpgraded);
}
```

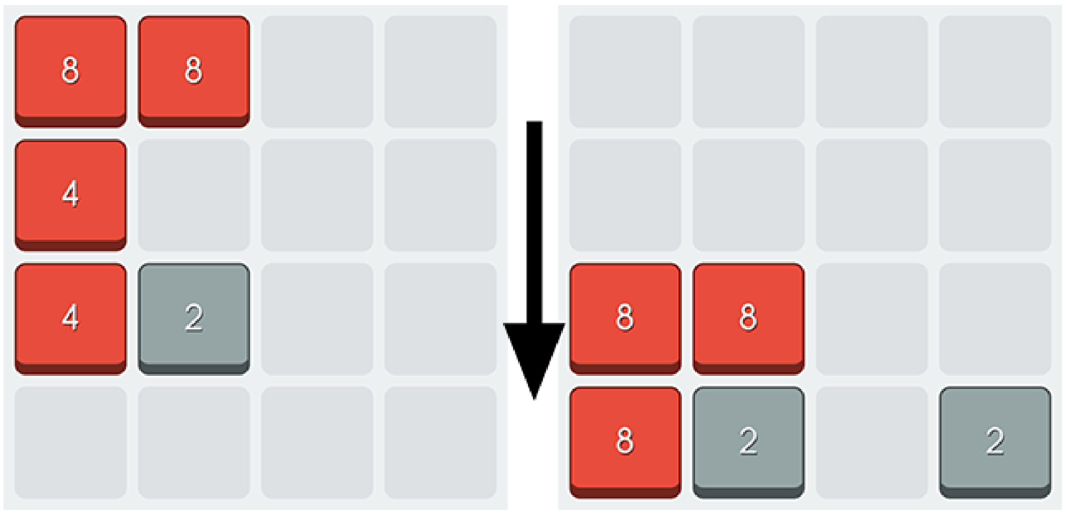
Following the same concept applied to `emptySpot` and `sameValue` variables, `alreadyUpgraded` is a Boolean variable which is `true` if the tile has been upgraded.

Then, this is the final value returned by the method:

```
return emptySpot || (sameValue && !alreadyUpgraded);
```

We can say a tile is moving in a legal position when it's on an empty spot or on another tile with the same value, but only if it hasn't been already upgraded.

Run the game, and you will be able to play with the original rules, like in this case when you move down:



If we look at the first column, the two “4” tiles merge into an “8”, but the remaining two “8” do not merge, just stack.

And finally you can play with the original game rules.

> You will find the full project in the folder
>
> **017 - Upgrading tiles only once per turn**

What about adding some animations now?

Animating tile movement

Do you miss the fancy animations made using tweens when new tiles appear?

We are going to animate tile movement. Each move the player does, a different number of tiles will need to be moved with a tween.

Tiles which need to move through more spots on the board will take more time than tiles which only need to move by one spot.

We have to keep track of all moving tiles and wait for all of them to reach their destination before letting the player move again.

We need to make some changes to makeMove method, and we'll start by adding a new property to count the tiles we are moving.

```
makeMove(d){
    this.movingTiles = 0;
    var dRow = (d == LEFT || d == RIGHT) ? 0 : d == UP ? -1 : 1;
    var dCol = (d == UP || d == DOWN) ? 0 : d == LEFT ? -1 : 1;
    this.canMove = false;
    // var movedTiles = 0;
    var firstRow = (d == UP) ? 1 : 0;
    var lastRow = gameOptions.boardSize.rows - ((d == DOWN) ? 1 : 0);
    var firstCol = (d == LEFT) ? 1 : 0;
    var lastCol = gameOptions.boardSize.cols - ((d == RIGHT) ? 1 : 0);
    // var movedSomething = false;
    // same as before
}
```

movingTiles will count the tiles we are moving, and starts at zero because at the beginning of a move there are no moving tiles.

Some variables now become obsolete, because the entire routine which will handle tile movement will be managed by a new method we are going to add as soon as we finish with makeMove.

This is why we do not need anymore movedTiles and movedSomething variables.

movingTiles will take care of everything. In the code above I commented them to let you see I won't use them anymore, but you can safely delete them.

Changes to makeMove method are not over yet, because there is a lot to do inside the routine which manages tiles, that is inside the block to be executed when `tileValue` is different than zero inside the two `for` loops which iterate through the board.

```
if(tileValue != 0){
    var newRow = curRow;
    var newCol = curCol;
    while(this.isLegalPosition(newRow + dRow, newCol + dCol, tileValue)){
        newRow += dRow;
        newCol += dCol;
    }
    // movedTiles ++;
    if(newRow != curRow || newCol != curCol){
        // movedSomething = true;
        // this.boardArray[curRow][curCol].tileSprite.depth = movedTiles;
        var newPos = this.getTilePosition(newRow, newCol);
        // this.boardArray[curRow][curCol].tileSprite.x = newPos.x;
        // this.boardArray[curRow][curCol].tileSprite.y = newPos.y;
        this.moveTile(this.boardArray[curRow][curCol].tileSprite, newPos);
        this.boardArray[curRow][curCol].tileValue = 0;
        if(this.boardArray[newRow][newCol].tileValue == tileValue){
            this.boardArray[newRow][newCol].tileValue ++;
            this.boardArray[newRow][newCol].upgraded = true;
            this.boardArray[curRow][curCol].tileSprite.setFrame(tileValue);
        }
        else{
            this.boardArray[newRow][newCol].tileValue = tileValue;
        }
    }
}
```

Since we removed `movedTiles` and `movedSomething` variables, we also have to remove the lines of code making use of them.

We'll also remove the two lines of code which set `tileSprite`'s `x` and `y` properties and add a call to a new method in place:

```
this.moveTile(this.boardArray[curRow][curCol].tileSprite, newPos);
```

`moveTile` method will handle tile animation and movement, and has two arguments: the sprite to move and the `Point` object with the new position of the sprite.

One little more change to `makeMove` method is needed: at the very end of the method we used to call `refreshBoard` or set `canMove` property to `true` according to what happened during the move.

We'll remove this piece of code and set `canMove` to `true` only if `movingTiles` is still at its starting value – zero – which means no tiles have been moved.

```
makeMove(d){
    // same as before
    // if(movedSomething){
    //     this.refreshBoard();
    // }
    // else{
    //     this.canMove = true;
    // }
    if(this.movingTiles == 0){
        this.canMove = true;
    }
}
```

The new `moveTile` method, which will handle all tile movement, position and depth, is ready to be coded:

```
moveTile(tile, point){
    this.movingTiles ++;
    tile.depth = this.movingTiles;
    var distance = Math.abs(tile.x - point.x) + Math.abs(tile.y - point.y);
    this.tweens.add({
        targets: [tile],
        x: point.x,
        y: point.y,
        duration: gameOptions.tweenSpeed * distance / gameOptions.tileSize,
        callbackScope: this,
        onComplete: function(){
            this.movingTiles --;
            tile.depth = 0;
            if(this.movingTiles == 0){
                this.refreshBoard();
            }
        }
    })
}
```

Let's split the code line by line and explain what's happening:

```
this.movingTiles ++;
```

We are moving a tile so we increase `movingTiles` property to update the number of moving tiles we are dealing with.

```
tile.depth = this.movingTiles;
```

This is where we set tile's Z-order.

```
var distance = Math.abs(tile.x - point.x) + Math.abs(tile.y - point.y);
```

`distance` variable holds the distance, in pixels, between current tile position and its destination.

> `Math.abs(n)` is a pure JavaScript method which returns the **absolute value** – the non-negative value without regard to its sign – of `n`.

Now we have a new tween, very similar to the one you saw when you animated newly added “2” tiles.

```
this.tweens.add({
    targets: [tile],
    x: point.x,
    y: point.y,
    duration: gameOptions.tweenSpeed * distance / gameOptions.tileSize,
    callbackScope: this,
    onComplete: function(){
        this.movingTiles --;
        tile.depth = 0;
        if(this.movingTiles == 0){
            this.refreshBoard();
        }
    }
})
```

This time the properties to tween are `x` and `y` position of `tile` sprite.

The duration of the tween is determined both by `tweenSpeed` value and the distance to travel. Since `distance` is measured in pixels and we want to work with tiles, we divided `distance` by `tileSize` to get the amount of tiles to travel.

What happens when each tween ends? First, there's one less moving tile:

```
this.movingTiles --;
```

Then, we reset tile Z-order:

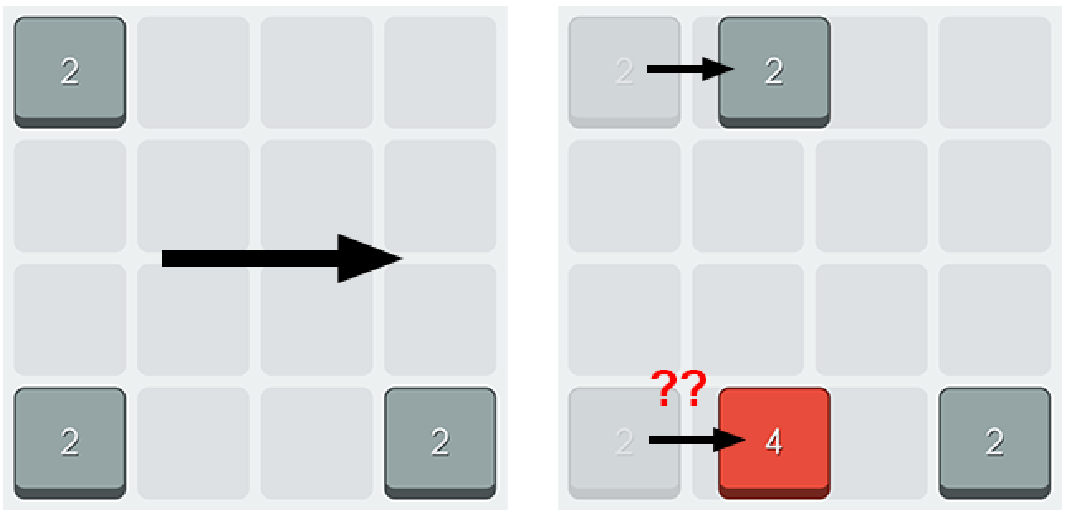
```
tile.depth = 0;
```

And finally if there are no more tiles to move, we call refreshBoard method which starts the cycle again:

```
if(this.movingTiles == 0){
    this.refreshBoard();
}
```

Run the game and see your tiles move smoothly, bringing to surface another bug.

Look at this picture:



When we move right, the “2” in the top row moves as we want, but the “2” in the bottom row suddenly turns into a “4” before actually merging with the other “2” in the bottom right of the board.

> You will find the full project in the folder
>
> **018 - Animating tile movement**

We'll fix this in a moment.

Animating tile upgrade

To animate tile upgrade, we'll use another tween at the end of the tween which moves the tile, but most of all we have to wait for the tile to be in its destination place before changing its frame.

Inside `makeMove` method, we have to make some changes to the routine which handles the case there's a tile to move:

```
if(newRow != curRow || newCol != curCol){
    var newPos = this.getTilePosition(newRow, newCol);
    var willUpdate = this.boardArray[newRow][newCol].tileValue == tileValue;
    this.moveTile(this.boardArray[curRow][curCol].tileSprite, newPos, willUpdate);
    this.boardArray[curRow][curCol].tileValue = 0;
    if(willUpdate){
        this.boardArray[newRow][newCol].tileValue ++;
        this.boardArray[newRow][newCol].upgraded = true;
        // this.boardArray[curRow][curCol].tileSprite.setFrame(tileValue);
    }
    else{
        this.boardArray[newRow][newCol].tileValue = tileValue;
    }
}
```

A new Boolean variable called `willUpdate` will tell us if the moving tile need to be updated.

It's the same check we performed before inside the `if` statement, we just saved the result in `willUpdate` variable because we are also passing it as a third argument to `moveTile` method.

To be more clear, before it was

```
if(this.boardArray[newRow][newCol].tileValue == tileValue){
    // rest of the code
}
```

Now it is:

```
var willUpdate = this.boardArray[newRow][newCol].tileValue == tileValue;
```

And then:

```
if(willUpdate){
    // rest of the code
}
```

The result is the same, but we also have the Boolean value to pass as argument to `moveTile` method.

About `moveTile`, it changed only a bit.

```
moveTile(tile, point, upgrade){
    this.movingTiles ++;
    tile.depth = this.movingTiles;
    var distance = Math.abs(tile.x - point.x) + Math.abs(tile.y - point.y);
    this.tweens.add({
        targets: [tile],
        x: point.x,
        y: point.y,
        duration: gameOptions.tweenSpeed * distance / gameOptions.tileSize,
        callbackScope: this,
        onComplete: function(){
            if(upgrade){
                this.upgradeTile(tile);
            }
            else{
                this.movingTiles --;
                tile.depth = 0;
                if(this.movingTiles == 0){
                    this.refreshBoard();
                }
            }
        }
    })
}
```

First, it accepts the third argument to see if the moved tile needs to be updated.

Then, inside `onComplete` callback function if `upgrade` is `true` we call a new method – `upgradeTile` – which will handle tile upgrade animation.

If `upgrade` is `false`, there is no need to upgrade the tile and there no need to do anything more than we previously did, so the rest of the code remains unchanged inside the `else` block.

In the original game, when a tile is being updated, it pulses a bit. To create this effect, we'll need to act on tile size, making it a little bigger, then setting it back to its original size.

This will lead to a couple of new concepts we are about to see in `upgradeTile` method:

```
upgradeTile(tile){
    tile.setFrame(tile.frame.name + 1);
    this.tweens.add({
        targets: [tile],
        scaleX: 1.1,
        scaleY: 1.1,
        duration: gameOptions.tweenSpeed,
        yoyo: true,
        repeat: 1,
        callbackScope: this,
        onComplete: function(){
            this.movingTiles --;
            tile.depth = 0;
            if(this.movingTiles == 0){
                this.refreshBoard();
            }
        }
    })
}
```

Some nice stuff here, start with this line

```
tile.setFrame(tile.frame.name + 1);
```

This is how we change the frame of a sprite, making it show next frame in sprite assigned texture.

You already know `setFrame` method, and `frame.name` is the name of the current frame, which in this case is the index number, so the first frame is named `0`, the second frame `1`, and so on.

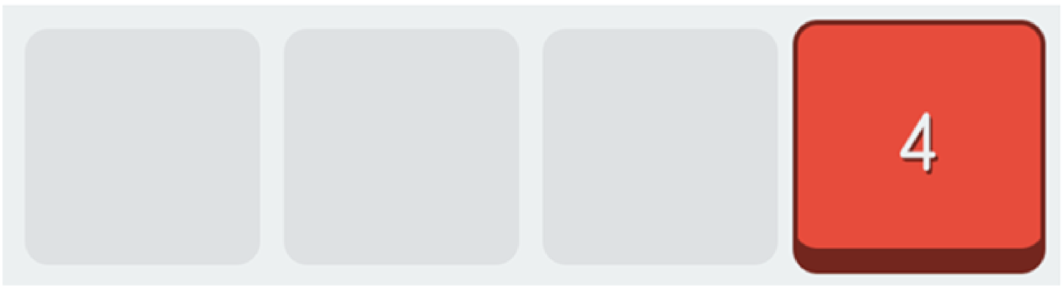
A **texture** consists of a source, usually a preloaded image, and a collection of fFrames which represent the different areas of the texture.

`name` property of a **Frame** object returns the unique name of the frame in the texture. When frames have no assigned names, they are named with their indexes inside the texture.

With the sprite showing the new frame, let's have a look at the tween we are using and how we can modify sprite size by tweening its `scaleX` and `scaleY` properties.

But let's see the interesting stuff in this new tween, that is the yoyo effect.

Setting `yoyo` to `true`, the tween will play forward and backward for `repeat` amount of times, and that's what is happening now:



Upgrade a tile, and you will see it growing then turning back to its original size, thanks to yoyo effect.

In a tween configuration object,

`scaleX` is the size along horizontal axis, where 1 = original size.

`scaleY` is the size along vertical axis, where 1 = original size.

`yoyo` plays the tween forward and backward if set to `true`.

`repeat` is the amount of times tween will be played.

Once the tween is completed, we decrease `movingTiles`, set sprite depth to zero and check if e have to refresh the board, which are exactly the same operations we make when we finish movement tween and there's no need to upgrade the tile.

```
onComplete: function(){
    this.movingTiles --;
    tile.depth = 0;
    if(this.movingTiles == 0){
        this.refreshBoard();
    }
}
```

And since there's no point in having two blocks of the same code, let's turn these few lines in a new method called `endTween`:

```
endTween(tile){
    this.movingTiles --;
    tile.depth = 0;
    if(this.movingTiles == 0){
        this.refreshBoard();
    }
}
```

Now `onComplete` callbacks of `moveTile` and `upgradeTile` methods will be respectively:

```
onComplete: function(){
    if(upgrade){
        this.upgradeTile(tile);
    }
    else{
        this.endTween(tile);
    }
}
```

and

```
onComplete: function(){
    this.endTween(tile);
}
```

Getting rid of duplicate code is always a good idea.

You will find the full project in the folder

019 - Animating tile upgrade

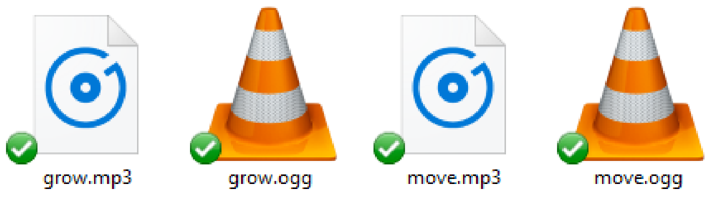
Done with animations, let's add sound effects to the game.

Adding sound effects

Our 4096 game will have two sound effects: one to be played when the player moves, and one to be played when a tile upgrades.

To keep the game folder structure well organized, we are creating a new folder inside `assets` folder, which already contains `sprites` folder, and we are calling it `sounds`.

Two sounds are added to `sounds` folder, in two different formats: **mp3** and **ogg**, making the folder content look like this:



Why did I use two sound formats?

It's a compatibility matter: not all browsers are capable to reproduce all kind of sound files. Using mp3 and ogg together should grant the best device and browser coverage.

Preloading sounds is not different than preloading images, as you can see in `preload` function in `bootGame` class:

```
preload(){
    this.load.image("emptytyle", "assets/sprites/emptytile.png");
    this.load.spritesheet("tiles", "assets/sprites/tiles.png", {
        frameWidth: gameOptions.tileSize,
        frameHeight: gameOptions.tileSize
    });
    this.load.audio("move", ["assets/sounds/move.ogg", "assets/sounds/move.mp3"]);
    this.load.audio("grow", ["assets/sounds/grow.ogg", "assets/sounds/grow.mp3"]);
}
```

We are preloading all sounds, then Phaser will choose which sound format to play according to browser capabilities.

`load.audio(key, audioFiles)` handles sound preloading. The first argument is the key, the unique name assigned to the sound, the second is an array of files to be loaded, in different formats.

With sounds ready to be played, it's time to see how we can reproduce them, but first let me speed up a bit the game, now that we are sure animations and tweens work the right way:

```
var gameOptions = {
    tileSize: 200,
    tileSpacing: 20,
    boardSize: {
        rows: 4,
        cols: 4
    },
    tweenSpeed: 50,
    swipeMaxTime: 1000,
    swipeMinDistance: 20,
    swipeMinNormal: 0.85
}
```

Much better now, the game is faster and more enjoyable thanks to this little change to `tweenSpeed` value in `gameOptions` global object.

Back to the sounds, we need to play a sound each time the player does a legal move, and each time a tile is upgraded, but first we have to create two properties storing the sounds themselves.

We are doing it at the end of `create` method of `playGame` class:

```
create(){
    // same as before
    this.moveSound = this.sound.add("move");
    this.growSound = this.sound.add("grow");
}
```

Adding sounds to the game is not that different than adding sprites, just add them.

`sound.add(key)` adds a new audio file to the sound manager. `key` is the unique name we gave to the sound.

Once we have the sounds stored into variables, we need to play them.

We said we are going to play the move sound each time the player makes a legal move, so we need to add a couple of lines to `makeMove` method:

```
makeMove(d){
    // same as before
    if(this.movingTiles == 0){
        this.canMove = true;
    }
    else{
        this.moveSound.play();
    }
}
```

We enter the else block if `movingTiles` is different than zero, that is if we are moving at least one tile, so it's time to play `moveSound` sound.

`play()` method plays a sound.

When to play `growSound`? Each time a tile is updated, inside `upgradeTile` method:

```
upgradeTile(tile){
    this.growSound.play();
    // same as before
}
```

This way `moveSound` is played only once per move, as it would be too noisy to play the same sound at each tile movement in a single turn.

`growsound` can be played each time a tile upgrades, even if it's happening in the same turn, because we want to emphasize this event.

You will find the full project in the folder

020 - Adding sound effects

Now, let's make this game look something better than a square with some tiles in it.

Covering the entire window area

At the moment the game is only a gray board inside a white window.

We have to improve the way the game looks, and we'll start by filling the entire window with the same background color used in the game, which is `#ecf0f1`.

In `index.html` file, add this new line to set the new background color.

```
<!DOCTYPE html>
<html>
    <head>
        <title>My Awesome Game</title>
        <meta name = "viewport" content = "width = device-width, initial-scale =
               1.0, maximum-scale = 1.0, minimum-scale = 1.0, user-scalable = 0,
               minimal-ui" />
        <style type = "text/css">
            body{
                background: #ecf0f1;
                padding: 0px;
                margin: 0px;
            }
            canvas{
                display:block;
                margin: 0;
                position: absolute;
                top: 50%;
                left: 50%;
                transform: translate(-50%, -50%);
            }
        </style>
        <script type="text/javascript" src="phaser.min.js"></script>
        <script type="text/javascript" src="game.js"></script>
    </head>
    <body>
    </body>
</html>
```

Also, the game should have an aspect ratio, it can't be a square.

We are going to make a portrait game, because HTML5 games are played into a browser and most people are used to hold their mobile devices in portrait mode while surfing the web.

Meanwhile, portrait mode allows to play only with one hand even if you use mobile phones with wider screens like the “plus” iPhone series, while landscape mode often requires the phone to be held with both hands.

An aspect ratio with should fit fairly good on every mobile device is 16/9, so we are going to add another value to global `gameOptions` object.

```
var gameOptions = {
    tileSize: 200,
    tileSpacing: 20,
    boardSize: {
        rows: 4,
        cols: 4
    },
    tweenSpeed: 50,
    swipeMaxTime: 1000,
    swipeMinDistance: 20,
    swipeMinNormal: 0.85,
    aspectRatio: 16/9
}
```

`aspectRatio` now stores the aspect ratio of the game.

Some small changes are required to `onload` method to adjust the height of the game.

We decided it's a portrait game, so we are going to keep game width just like before, and calculate the new height:

```
window.onload = function() {
    var tileAndSpacing = gameOptions.tileSize + gameOptions.tileSpacing;
    var width = gameOptions.boardSize.cols * tileAndSpacing;
    width += gameOptions.tileSpacing;
    var gameConfig = {
        width: width,
        height: width * gameOptions.aspectRatio,
        backgroundColor: 0xecf0f1,
        scene: [bootGame, playGame]
    }
    game = new Phaser.Game(gameConfig);
    window.focus();
    resizeGame();
    window.addEventListener("resize", resizeGame);
}
```

Although there are some new lines, the first three lines just calculate game width according to the number of columns, tile size and tile spacing just like we did the first time.

I only split the code through more lines to improve readibility.

The core of the concept now lies in this line:

```
height: width * gameOptions.aspectRatio,
```

The height of the game is no more calculated according to the number of rows, but by multiplying the width by the aspect ratio.

This won't allow us to display boards with a big difference between the number of rows and the number of columns, but it does not matter as the game won't be that fun with more than 6 rows or columns.

We have to make some compromise and it's fine to play with a board up to six or seven rows. All in all the original game was a huge success with a fixed 4x4 layout.

Changing game size also affects tile positioning, as we want the board to always be in the center of the screen.

Here is the new `getTilePosition` method:

```
getTilePosition(row, col){
    var posX = gameOptions.tileSpacing * (col + 1) + gameOptions.tileSize * (col +
       0.5);
    var posY = gameOptions.tileSpacing * (row + 1) + gameOptions.tileSize * (row +
       0.5);
    var boardHeight = gameOptions.boardSize.rows * gameOptions.tileSize;
    boardHeight += (gameOptions.boardSize.rows + 1) * gameOptions.tileSpacing;
    var offsetY = (game.config.height - boardHeight) / 2;
    posY += offsetY;
    return new Phaser.Geom.Point(posX, posY);
}
```

The way we calculate posX does not change, but we have to add an offset to posY to make the board be vertically centered.

```
var boardHeight = gameOptions.boardSize.rows * gameOptions.tileSize;
boardHeight += (gameOptions.boardSize.rows + 1) * gameOptions.tileSpacing;
```

First, we store in `boardHeight` variable the height of the board, in pixels, starting from the number of rows, the tile size and the tile spacing.

Again, I wrote the code in two lines for the sake of readability.

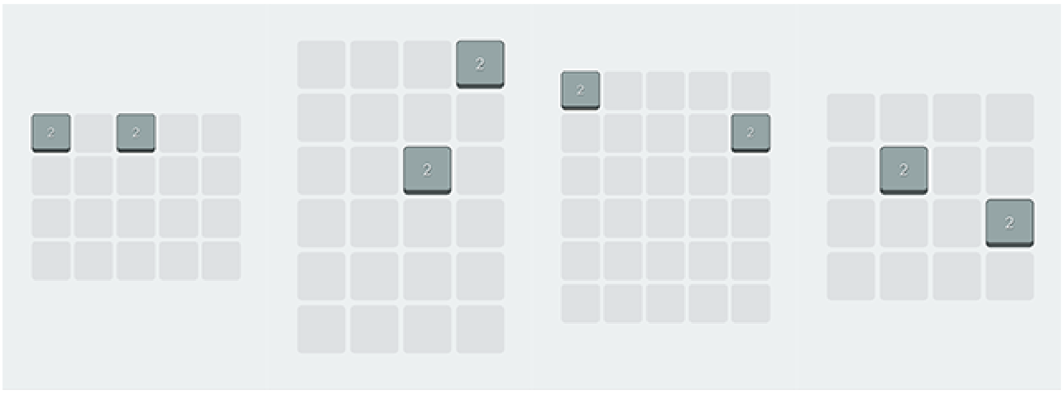
```
var offsetY = (game.config.height - boardHeight) / 2;
```

`offsetY` variable takes the vertical offset to be added to previously calculated `posY`. It's half the difference between game height and board height.

```
posY += offsetY;
```

And finally the offset is added to `posY`.

You can test the game:



No matter the number of rows and columns, as log as it's a fair number of rows and columns, the game is perfectly playable.

You will find the full project in the folder

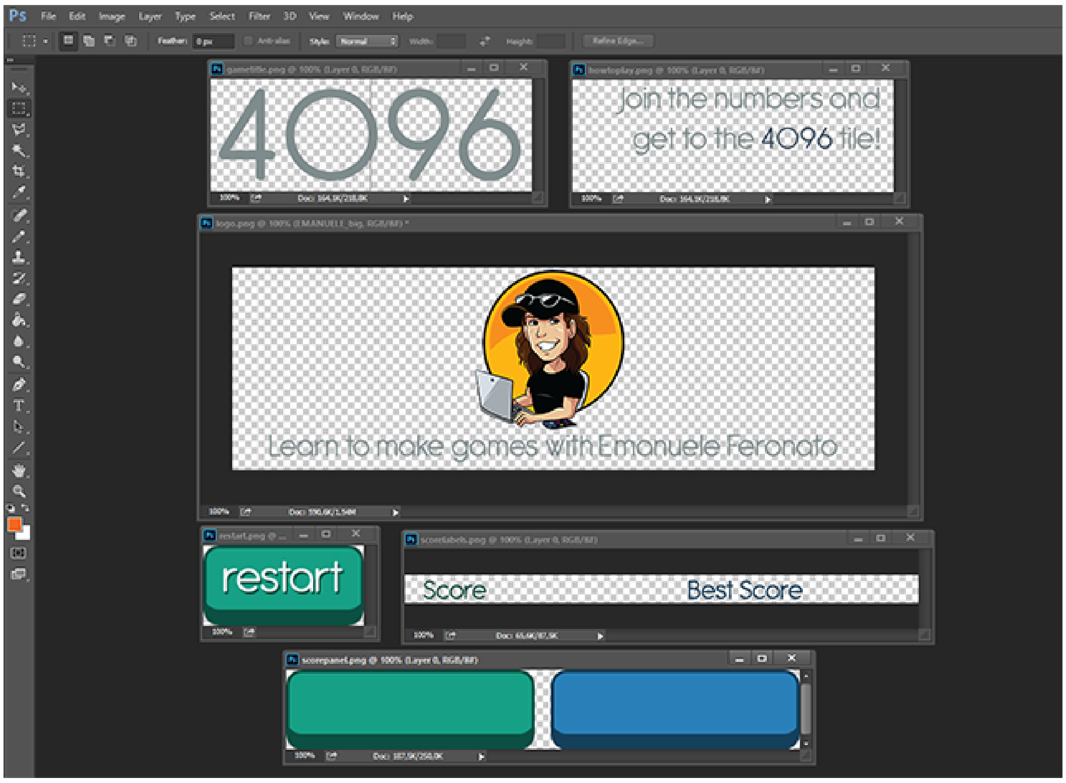
021 - Covering the entire window area

You may think nothing changed, we only added some empty space above and below the board.

We'll use this space to create the game graphical interface.

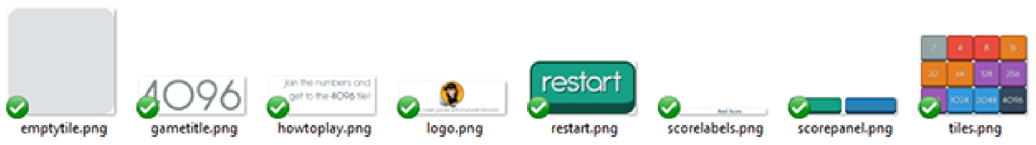
Adding game interface

With this new empty space to fill, let's create some graphic elements to build the game interface. Here are the ones I made:


4096
Join the numbers and get to the 4096 tile!
Learn to make games with Emanuele Feronato
restart
Score
Best Score

We have the game title, the game instructions, my branded logo, a restart button, a single image two rectangles where to display current score and best score, and a single image with two labels.

All these images will be saved in the good old sprites folder inside assets folder, which now should look this way:



This is the right time for you to unleash your creativity and draw your own assets.

Keep in mind all your images should have the same theme, or your game will look like early 1990 websites with a lot of random images popping here and there.

Also, you should draw buttons and panels which fit with each row/column board configuration, or at least the most common ones.

Then we add the required lines to `preload` method to preload the new images.

```
preload(){
    this.load.image("restart", "assets/sprites/restart.png");
    this.load.image("scorepanel", "assets/sprites/scorepanel.png");
    this.load.image("scorelabels", "assets/sprites/scorelabels.png");
    this.load.image("logo", "assets/sprites/logo.png");
    this.load.image("howtoplay", "assets/sprites/howtoplay.png");
    this.load.image("gametitle", "assets/sprites/gametitle.png");
    this.load.image("emptytyle", "assets/sprites/emptytile.png");
    this.load.spritesheet("tiles", "assets/sprites/tiles.png", {
        frameWidth: gameOptions.tileSize,
        frameHeight: gameOptions.tileSize
    });
    this.load.audio("move", ["assets/sounds/move.ogg", "assets/sounds/move.mp3"]);
    this.load.audio("grow", ["assets/sounds/grow.ogg", "assets/sounds/grow.mp3"]);
}
```

In `create` method we'll add to the stage the images we just preloaded.

Although it's only a matter of adding images, something you should already know, there is a couple of new concepts.

These are the new lines, we'll add them at the very beginning of `create` method.

```
create(){
    var restartXY = this.getTilePosition(-0.8, gameOptions.boardSize.cols - 1);
    var restartButton = this.add.sprite(restartXY.x, restartXY.y, "restart");
    var scoreXY = this.getTilePosition(-0.8, 1);
    this.add.image(scoreXY.x, scoreXY.y, "scorepanel");
    this.add.image(scoreXY.x, scoreXY.y - 70, "scorelabels");
    var gameTitle = this.add.image(10, 5, "gametitle");
    gameTitle.setOrigin(0, 0);
    var howTo = this.add.image(game.config.width, 5, "howtoplay");
    howTo.setOrigin(1, 0);
    var logo = this.add.sprite(game.config.width / 2, game.config.height, "logo");
    logo.setOrigin(0.5, 1);
    // same as before
}
```

Let's explain these new lines one by one.

We want to place the restart button above the rightmost column, so here is how we calculate restart button position:

```
var restartXY = this.getTilePosition(-0.8, gameOptions.boardSize.cols - 1);
```

`getTilePosition` method also accepts non-integers, non-negative numbers as argument. The method simply calculates a `Point` with `x` and `y` coordinates, it's up to you to adjust the values of its arguments to make your restart button look nice.

With these settings, restart button – which has been drawn with the same length as a tile – will be placed some pixels above the rightmost column, in the position stored in `restartXY` variable.

```
var restartButton = this.add.sprite(restartXY.x, restartXY.y, "restart");
```

And now we just added the button to the canvas.

```
var scoreXY = this.getTilePosition(-0.8, 1);
```

Same thing for `scoreXY` variable, will store the position of the score panel, which has been with the same length as three tiles and two tile spacings.

```
this.add.image(scoreXY.x, scoreXY.y, "scorepanel");
```

And we added the score panel.

```
this.add.image(scoreXY.x, scoreXY.y - 70, "scorelabels");
```

On top of the score panel we added the score labels.

Why having two images, one for the score panel and one for the score labels, rather than one single image with both score panels and score labels in it?

It's a matter of making the game customizable. I prefer to keep hardcoded text away from other graphic assets, should I need to change it or remove it I don't have to worry about the rest of the assets.

```
var gameTitle = this.add.image(10, 5, "gametitle");
```

The game title is added at the very top of the canvas.

```
gameTitle.setOrigin(0, 0);
```

And we set its origin, or anchor point, on the top left corner.

The **anchor** or **origin** of an image or sprite sets the origin point of the image.

When you add an image at `x, y` you actually add an image in a position so that its origin is `x, y`.

Setting the origin to `0.5, 0.5` – which is also the default value – means the image origin is the center of the image.

Setting the origin to `1, 1` means the image origin is on the bottom right corner.

Setting the origin to `0, 0` means the image origin is on the top left corner.

Any value from 0 to 1 is accepted.

`setOrigin(x, y)` method sets image origin to `x, y`.

Game instructions will be placed on the opposite side of the title, at the rightmost side of the canvas:

```
var howTo = this.add.image(game.config.width, 5, "howtoplay");
```

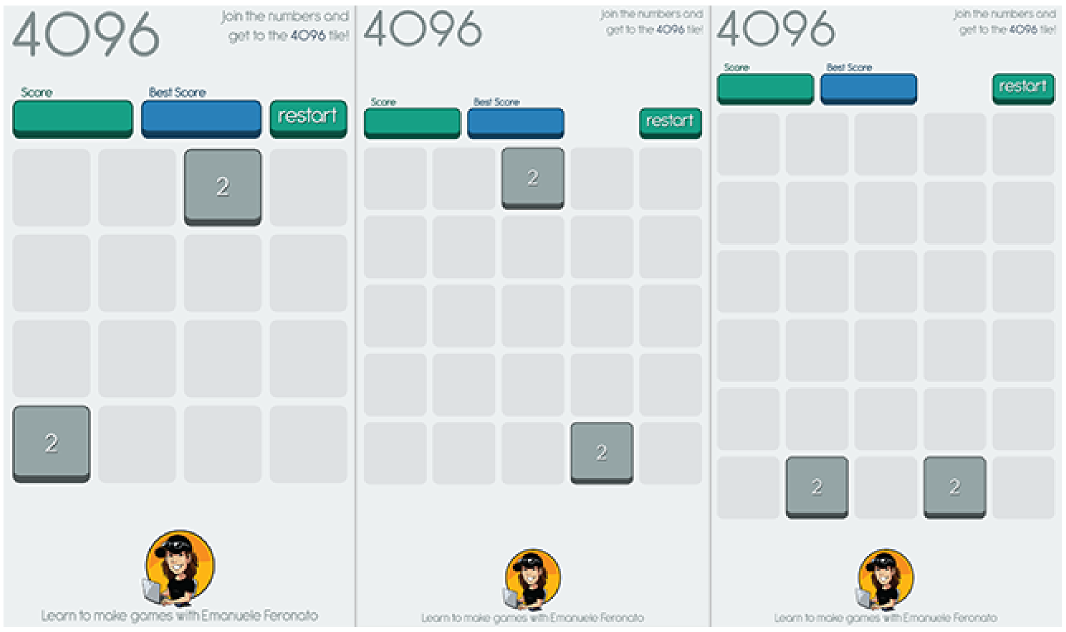
With the origin set on the top right corner.

```
howTo.setOrigin(1, 0);
```

And finally the logo, placed in the bottom center of the canvas, with its origin also in its bottom center:

```
var logo = this.add.sprite(game.config.width / 2, game.config.height, "logo");
logo.setOrigin(0.5, 1);
```

Run the game and see the interface in action:


4096
Join the numbers and get to the 4096 tile!
Score
Best Score
restart
2
Learn to make games with Emanuele Feronato

We aren't just building a HTML5 cross platform game, we are also allowing room for some customization without adding exceptions to the code.

Carefully designing your interface will allow you to save a lot of time.

You will find the full project in the folder

022 - Adding game interface

Too bad restart button does not work.

Making sprites interactive

Having a restart button is quite useless if you do not restart the game when clicking on it.

Adding interactivity on sprites turning them into buttons is very easy, a matter of a couple of lines of code, to be added to `create` method.

```
create(){
    var restartXY = this.getTilePosition(-0.8, gameOptions.boardSize.cols - 1);
    var restartButton = this.add.sprite(restartXY.x, restartXY.y, "restart");
    restartButton.setInteractive();
    restartButton.on("pointerdown", function(){
        this.scene.start("PlayGame");
    }, this);
    var scoreXY = this.getTilePosition(-0.8, 1);
    this.add.image(scoreXY.x, scoreXY.y, "scorepanel");
    this.add.image(scoreXY.x, scoreXY.y - 70, "scorelabels");
    var gameTitle = this.add.image(10, 5, "gametitle");
    gameTitle.setOrigin(0, 0);
    var howTo = this.add.image(game.config.width, 5, "howtoplay");
    howTo.setOrigin(1, 0);
    var logo = this.add.sprite(game.config.width / 2, game.config.height, "logo");
    logo.setOrigin(0.5, 1);
    logo.setInteractive();
    logo.on("pointerdown", function(){
        window.location.href = "http://www.emanueleferonato.com/"
    });
    // same as before
}
```

We start with making the restart button interactive, with:

```
restartButton.setInteractive();
```

At this point, the sprite is interactive and capable of firing interaction events, like `pointerdown` when the pointer – not strictly referred to a mouse pointer but any kind of input – is down, pressing the button.

Then a callback function can be called, just like we made when checking for swipes: an input event occurs, a callback function is called.

In this case, the callback function simply starts `playGame` scene again, actually restarting the game.

We only need to start `playGame` since all images have already been loaded in `bootGame` scene at the beginning of the game, so we won't need to start `bootGame` anymore.

If you want to restart the game, just start `playGame` and you'll already find all your sprites and sounds loaded.

> `setInteractive()` method enables the game object for input.

The same concept is applied to the logo, but in this case I simply point the browser to my blog.

> The `window.location` object can be used to redirect the browser to a new page.
>
> `window.location.href = URL` sets the href value to point to another web site located at `URL`.

Test the game and see how you can restart the game by pressing restart button or navigate to my blog by clicking on the logo.

> You will find the full project in the folder
>
> **023 - Making sprites interactive**

Are you ready to score a billion? I am afraid you're not, there isn't even a scoring system.

There is something more to do before displaying the score.

Using bitmap fonts

This will be one of the most difficult steps as we are going to introduce a lot of new concepts.

We need to show player score, and in order to do it with the highest level of customization possible we need to introduce bitmap fonts, so unfortunately a bit of boring theory is needed.

All the text displayed on your browser is rendered using a font file which contains all the information necessary to draw the shape of each character, no matter the size and scale.

When you print a string on the screen, each character is scaled and rendered, leading to two problems:

First, the process of scaling and rendering characters is quite CPU intensive, especially if you need to add run-time effects like colors, outlines or shadows.

Second, if you don't use a common font it's very likely most players won't have that font installed on their computers or mobile devices, so they won't be able to properly render it and a default font will be used instead.

This is where bitmap fonts come into play.

Basically, a bitmap font is an image file containing all the characters we need and a control file with the coordinates of each character in the image.

You may think about bitmap fonts as a special sprite sheet with letters and numbers.

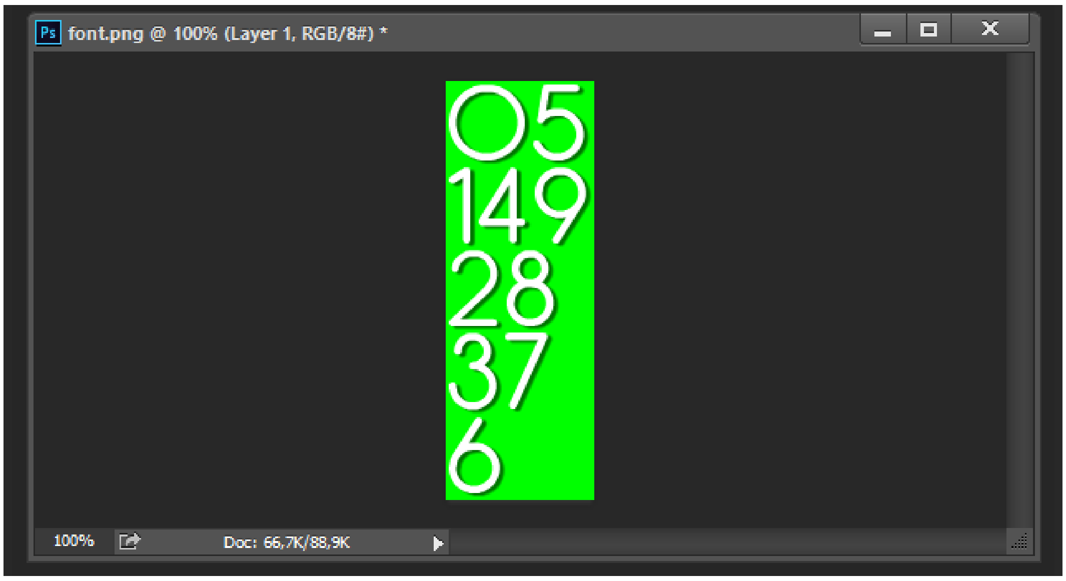
With bitmap fonts each character can be rendered using multiple effects, loaded as an image, and placed to the screen using very little resources.

The drawback is each bitmap font file can contain only one font size, but it won't be a problem in our case.

There is a wide choice of tools to generate bitmap fonts, but the one I prefer – and actually use – is **Littera bitmap font generator** (http://kvazars.com/littera/), a free web application to generate bitmap fonts with all options you need.

It's very easy to use and when you export the font you can keep the default settings so you will get two files: a png file which is the image containing all characters and a fnt file containing font information.

I created a 72 pixels white font with a soft shadow effect, and this is the image Littera created, displayed on a green background to allow you to see the transparency:



Both the png image called `font.png` and the fnt file called `font.fnt` will be placed in a new folder called `fonts` inside `assets` folder which already contains `sounds` and `sprites` folders.

What's the first thing you need to do when dealing with a new asset? Preloading it in `preload` method of `bootGame` class.

```
preload(){
    // same as before
    this.load.bitmapFont("font", "assets/fonts/font.png",
       "assets/fonts/font.fnt");
}
```

The bitmap font is now ready to be used in the game.

`load.bitmapFont(key, texturePath, xmlPath)` method adds new bitmap font loading request, giving it unique `key` name and looking for `texturePath` image file and `xmlPath` data file.

In `create` method of `playGame` class, the score will be added after the creation of score panels:

```
create(){
    var restartXY = this.getTilePosition(-0.8, gameOptions.boardSize.cols - 1);
    var restartButton = this.add.sprite(restartXY.x, restartXY.y, "restart");
    restartButton.setInteractive();
    restartButton.on("pointerdown", function(){
        this.scene.start("PlayGame");
    }, this);
    var scoreXY = this.getTilePosition(-0.8, 1);
    this.add.image(scoreXY.x, scoreXY.y, "scorepanel");
    this.add.image(scoreXY.x, scoreXY.y - 70, "scorelabels");
    var textXY = this.getTilePosition(-0.92, -0.4);
    this.scoreText = this.add.bitmapText(textXY.x, textXY.y, "font", "0");
    textXY = this.getTilePosition(-0.92, 1.1);
    this.bestScoreText = this.add.bitmapText(textXY.x, textXY.y, "font", "0");
    // same as before
}
```

Let's see how to add bitmap text:

```
var textXY = this.getTilePosition(-0.92, -0.4);
```

First, `textXY` variable is created with the coordinates where to place the text, always using `getTilePosition` method which works according to tile size.

I have to say, it's a matter of trial and error to find the right values.

There's nothing related to programming in this. You have some assets, an unit of measurement – tile size – and you have to find the best way to place stuff on the screen while leaving room for customization.

```
this.scoreText = this.add.bitmapText(textXY.x, textXY.y, "font", "0");
```

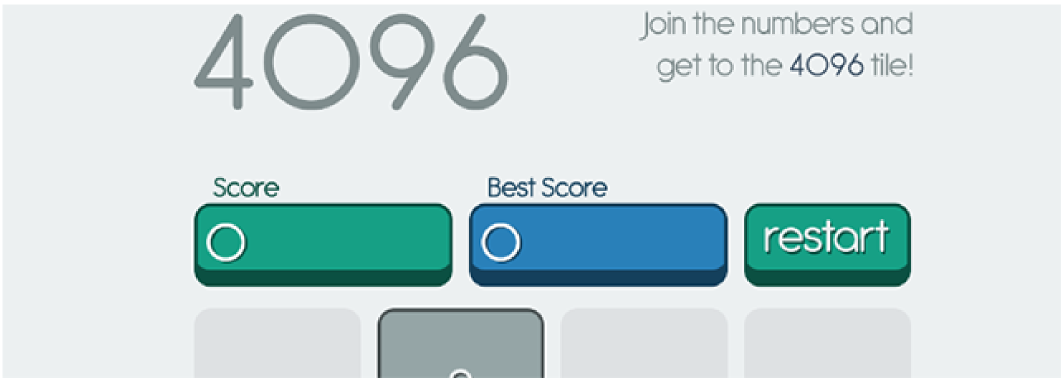
Our first bitmap text is added and saved in `scoreText` property.

> `add.bitmapText(x, y, key, text)` method adds `text` string written with `key` bitmap text at coordinates `x`, `y`.

The same concept is used to add the best score:

```
textXY = this.getTilePosition(-0.92, 1.1);
this.bestScoreText = this.add.bitmapText(textXY.x, textXY.y, "font", "0");
```

Test the game and see your bitmap text font in action:



And our fake score is correctly displayed.

> You will find the full project in the folder
>
> **024 - Using bitmap fonts**

Why am I talking about fake score? Because it never changes.

Let's add a proper scoring system.

Handling a score system

The original 2048 game has a simple yet challenging score system: each time you upgrade a tile, your score increases by the value of the upgraded tile.

If you merge two “2” tiles into a “4”, you will get 4 points. If you merge two “16” tiles into a “32”, you will get 32 points, and so on.

First, a new property needs to be added in `create` method of `playGame` class:

```
create(){
    this.score = 0;
    // same as before
}
```

`score` will keep count of the score made during the game, and it starts at zero.

Inside `makeMove` method in the routine which checks if the tile value needs to be updated, we'll add the line which handles the score:

```
if(newRow != curRow || newCol != curCol){
    var newPos = this.getTilePosition(newRow, newCol);
    var willUpdate = this.boardArray[newRow][newCol].tileValue == tileValue;
    this.moveTile(this.boardArray[curRow][curCol].tileSprite, newPos, willUpdate);
    this.boardArray[curRow][curCol].tileValue = 0;
    if(willUpdate){
        this.boardArray[newRow][newCol].tileValue ++;
        this.score += Math.pow(2, this.boardArray[newRow][newCol].tileValue);
        this.boardArray[newRow][newCol].upgraded = true;
    }
    else{
        this.boardArray[newRow][newCol].tileValue = tileValue;
    }
}
```

When a tile updates its value, the score is increased by the power of two of the new tile value.

> `Math.pow(base, exponent)` JavaScript method returns `base` to the `exponent` power.

Finally, each time we refresh the board, we also refresh the score, adding one more line at the beginning of `refreshBoard` method:

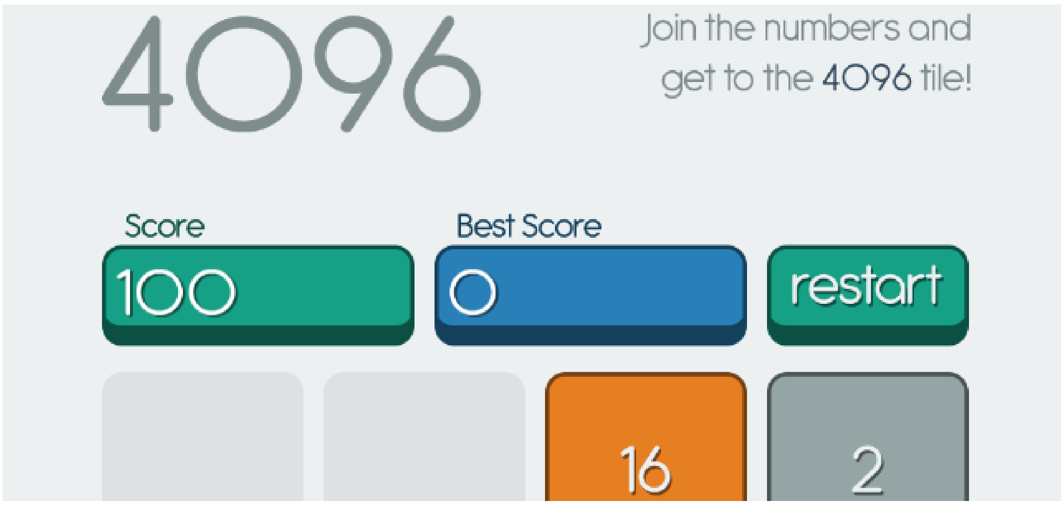
```
refreshBoard(){
    this.scoreText.text = this.score.toString();
    // same as before
}
```

And now `scoreText` bitmap text is updated.

`text` property of a bitmap text sets the text to show.

`toString` JavaScript method converts a number to a string.

Now play the game and see your score increase as you merge tiles:



If you restart the game, score also restarts from zero.

You will find the full project in the folder

025 - Handling a score system

And now let's save the best score.

Saving the best score

When playing a game, you will quickly realize there is no point in making a great score if you can't save it and try to beat it later.

We are going to cover how to save your best score, and keep it saved even if you close the browser window or turn off your computer or device.

All modern browsers support local storage, a way used by web pages to locally store data in a key/value notation.

The information you save will continue to be stored even when you shut down your device and can be read every time you launch your game.

This is exactly what we need.

Let's create a new item in `gameOptions` global object:

```
var gameOptions = {
    tileSize: 200,
    tileSpacing: 20,
    boardSize: {
        rows: 4,
        cols: 4
    },
    tweenSpeed: 50,
    swipeMaxTime: 1000,
    swipeMinDistance: 20,
    swipeMinNormal: 0.85,
    aspectRatio: 16/9,
    localStorageName: "topscore4096"
}
```

`localStorageName` item stores the name of the local storage variable, so each time you will change `topscore4096` with something else, you will reset your best score.

We are only saving the best score at the moment, but you can save anything you want: the number of games played, the total time spent playing the game, and so on.

As soon as we start the game, we have to check if there is a high score saved in local storage, so we are going to add a few lines to `create` method:

```
create(){
    this.score = 0;
    var restartXY = this.getTilePosition(-0.8, gameOptions.boardSize.cols - 1);
    var restartButton = this.add.sprite(restartXY.x, restartXY.y, "restart");
    restartButton.setInteractive();
    restartButton.on("pointerdown", function(){
        this.scene.start("PlayGame");
    }, this);
    var scoreXY = this.getTilePosition(-0.8, 1);
    this.add.image(scoreXY.x, scoreXY.y, "scorepanel");
    this.add.image(scoreXY.x, scoreXY.y - 70, "scorelabels");
    var textXY = this.getTilePosition(-0.92, -0.4);
    this.scoreText = this.add.bitmapText(textXY.x, textXY.y, "font", "0");
    textXY = this.getTilePosition(-0.92, 1.1);
    this.bestScore = localStorage.getItem(gameOptions.localStorageName);
    if(this.bestScore == null){
        this.bestScore = 0;
    }
    this.bestScoreText = this.add.bitmapText(textXY.x, textXY.y, "font",
       this.bestScore.toString());
    // same as before
}
```

Let's see what we did:

```
this.bestScore = localStorage.getItem(gameOptions.localStorageName);
```

A new property called `bestScore` is added and gets the value currently saved in local storage.

> `localStorage.getItem(keyName)` method of local storage returns `keyName`'s value or `null` if `keyName` does not exist.
>
> `null` represents JavaScript's intentional absence of any value.

At this time `bestScore` can have a value o can be `null`.

```
if(this.bestScore == null){
    this.bestScore = 0;
}
```

Actually, `null` is not a high score to be proud of, so we are setting `bestScore` to zero if we have a `null` value.

Now we just have to show the best score in `bestScoreText` bitmap text.

```
this.bestScoreText = this.add.bitmapText(textXY.x, textXY.y, "font",
       this.bestScore.toString());
```

At the beginning of the game, the best score – or zero – will be displayed in the proper place.

When to update it? After each successful move, when it's time to refresh the board, in `refreshBoard` method:

```
refreshBoard(){
    this.scoreText.text = this.score.toString();
    if(this.score > this.bestScore){
        this.bestScore = this.score;
        localStorage.setItem(gameOptions.localStorageName, this.bestScore);
        this.bestScoreText.text = this.bestScore.toString();
    }
    for(var i = 0; i < gameOptions.boardSize.rows; i++){
        for(var j = 0; j < gameOptions.boardSize.cols; j++){
            var spritePosition = this.getTilePosition(i, j);
            this.boardArray[i][j].tileSprite.x = spritePosition.x;
            this.boardArray[i][j].tileSprite.y = spritePosition.y;
            var tileValue = this.boardArray[i][j].tileValue;
            if(tileValue > 0){
                this.boardArray[i][j].tileSprite.visible = true;
                this.boardArray[i][j].tileSprite.setFrame(tileValue - 1);
                this.boardArray[i][j].upgraded = false;
            }
            else{
                this.boardArray[i][j].tileSprite.visible = false;
            }
        }
    }
    this.addTile();
}
```

When will you beat a best score? When your current score is higher than the best score. This is exactly what we are checking in this `if` statement.

```
if(this.score > this.bestScore){
    // rest of the script
}
```

Once `score` is greater than `bestScore`, first we update `bestScore`:

```
this.bestScore = this.score;
```

Then we save the best score in local storage:

```
localStorage.setItem(gameOptions.localStorageName, this.bestScore);
```

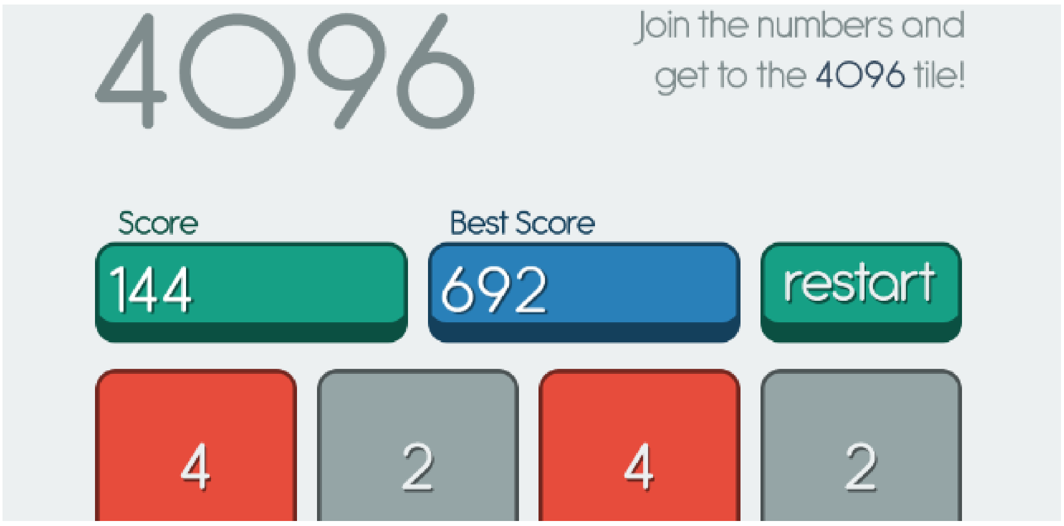
We are saving best score as soon as it updates, so if for some reason the player quits the game before it ends, the best score has already been saved.

`localStorage.setItem(keyName, keyValue)` method adds `keyName` to the storage, or updates it to `keyValue` if it already exists.

Finally, we update the best score bitmap text:

```
this.bestScoreText.text = this.bestScore.toString();
```

Test the game, quit it anytime, restart your device, your high score will remain.



What is your best score?

You will find the full project in the folder

026 - Saving the best score

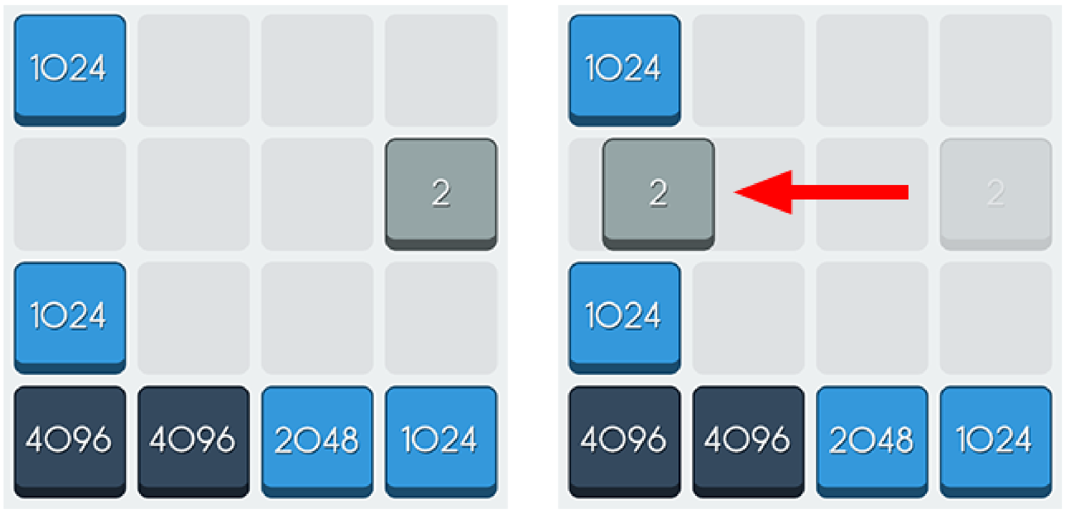
Now there's one last rule to develop.

Limiting tile value to “4096”

4096 is 2 by the power of 12, so items in `boardArray` with `tileValue` equal to `12` can't be upgraded, and we are going to add this rule in `isLegalPosition` method.

```
isLegalPosition(row, col, value){
    var rowInside = row >= 0 && row < gameOptions.boardSize.rows;
    var colInside = col >= 0 && col < gameOptions.boardSize.cols;
    if(!rowInside || !colInside){
        return false;
    }
    if(this.boardArray[row][col].tileValue == 12){
        return false;
    }
    var emptySpot = this.boardArray[row][col].tileValue == 0;
    var sameValue = this.boardArray[row][col].tileValue == value;
    var alreadyUpgraded = this.boardArray[row][col].upgraded;
    return emptySpot || (sameValue && !alreadyUpgraded);
}
```

This rule isn't in the original game, but I wanted to give it a twist. Run the game:



Moving right won't merge the “4096” tiles. Yes, I cheated when I took this picture.

You will find the full project in the folder

027 - Limiting tile value to 4096

And finally the game is completed.

Ever wanted to see your game in a picture like this one?

4096
Join the numbers and get to the 4096 tile!
Score
Best Score
124
692
restart
Learn to make games with Emanuele Feronato

Now you made it possible. Congratulations.

When you build a game following a tutorial or a book, I always suggest to build it twice: the first time following the tutorial and the second time on your own.

Take a deep breath, delete everything and try to build the game from scratch.

Then, add some basic features, like keeping track of the number of plays, or a mute sound button.

Thank you and let's keep in touch

Now you finished the book.

It's my first self-published book after three books written under a publishing label, so I apologize if you found some errors.

Please notify me any error you should find, and give me feedback dropping me a line to info@emanueleferonato.com

Also, follow my blog www.emanueleferonato.com where you can find new tutorials almost daily.

Finally, my Facebook fan page https://www.facebook.com/emanueleferonato

and Twitter account https://twitter.com/triqui

I would like to thank **Richard Davey** and all **Photon Storm** guys for making the incredible Phaser framework.

I hope you enjoyed reading this book as much as I enjoyed writing it.

Emanuele.