From null to full HTML5 cross platform game

using Phaser framework
and only FREE software

Emanuele Feronato

From null to full HTML5 cross platform game

Taking you by hand from the basics of JavaScript programming until the making of a complete cross-platform HTML5 game using Phaser framework.

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 it happened a lot of times in 30 years 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 a real, complete project.

Unfortunately, most beginner guides just make a brief overview, leaving it to you to continue learning and experimenting, and advanced guides assume you already are an advanced 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 create a complete game from scratch.

I really hope you will like this book and will find it useful.

But 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.

I will do my best to make this book worth every single cent of your money.

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 smartphones 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 smartphones 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 games we are going to build, is a game which will be able to run on various devices, such as smartphones and tablets – but also on desktop machines – each one with its own resolution and screen aspect ratio.

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 made 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 sponsors 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 did not look good on the Galaxy Note.

A few fixes and tweaks later, it happened the game did not look good on the Samsung S4.

When the game was finally optimized for all required devices, it did not 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.

Focus on coding and game development while a framework does the dirty work for you.

That's when **Phaser** comes into play.

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 if you did not 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.

Can I build a game like GTA with Phaser?

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

The first rule in game programming is: **create a game you are able to complete**. And when I say “complete a game” I don't mean seeing the congratulations screen in GTA, I mean coding a game from scratch until the end.

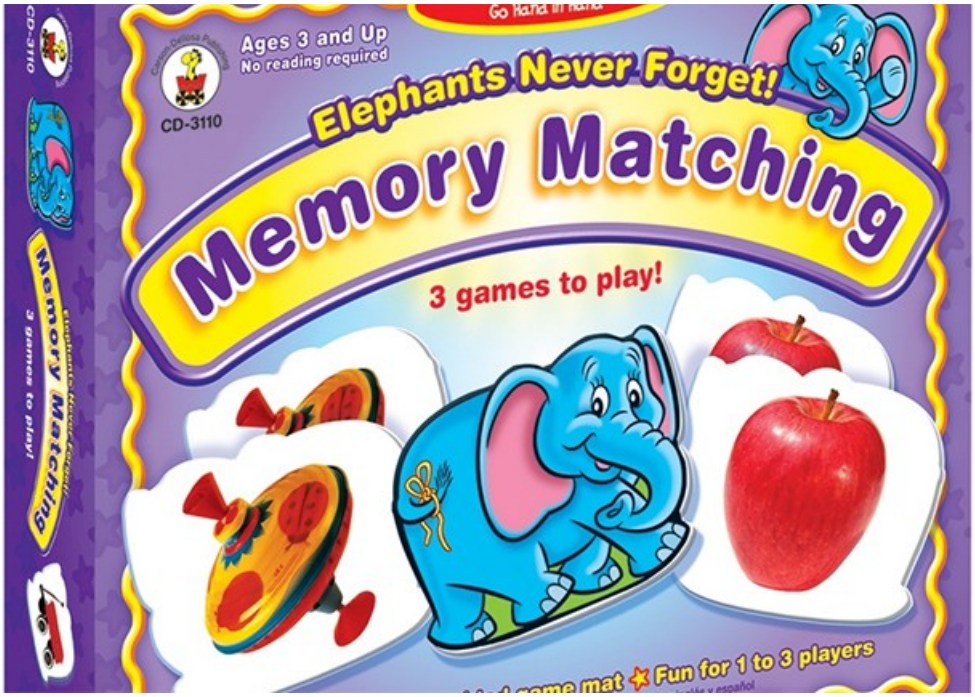
According to this concept, if you are an one-man studio or a small indie studio, you should choose a genre of game you are able to code from the beginning until the end.

The game we are going to build in this book is **Concentration**.

Concentration, also known as Memory, is a card game in which all of the cards are laid face down on a surface and two cards are flipped face up over each turn.

The object of the game is to remove all card by turning over pairs of matching cards.

I am sure you played with something similar when you were a kid:



And although it may seem a simple game – and actually it is – there is a lot to learn from the making of this game, and we'll add a couple of extra features to make it some more interesting.

Choosing a text editor

Let's turn your computer into a web development workstation with no cost thanks to Phaser and some other **free software**.

In order to start making games with Phaser, you'll first need a software to write code. There is a lot of free offers. I personally use **PSPad** (http://www.pspad.com/) on my Windows computer and **TextWrangler** (http://www.barebones.com/products/textwrangler/) on my Mac. Another alternative is **Brackets** (http://brackets.io/) but you can use your favorite text editor, I'd only suggest you choose one capable to highlight JavaScript syntax.

Choosing a 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 browsers security limits when running your project locally.

I am using **WAMP** (http://www.wampserver.com/) on my Windows machine, and **MAMP** (https://www.mamp.info/) on the Mac. Recently, MAMP also released a Windows version (https://www.mamp.info/en/mamp_windows.html).

Just like the text editors, both WAMP and MAMP are free to use as you won't need the PRO version, which is also available for MAMP.

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 to have a web server installed on your computer, but I highly recommend using WAMP or MAMP instead.

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

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 until 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 – this causes your games not to work.

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.

Other software you may need

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

Audacity (http://sourceforge.net/projects/audacity/) is a great free software to work with sounds, while I would suggest **GIMP** (http://www.gimp.org/) to work with images, which is also free.

You can also use the trial version of **Photoshop** (http://www.photoshop.com/) which allows you to use all features for free for a limited amount of time.

Downloading Phaser

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

Phaser comes in a zipped file with a lot of docs and resources for a download file greater than 30 MB, anyway we will need just one file, containing the framework itself.

The structure of your Phaser project

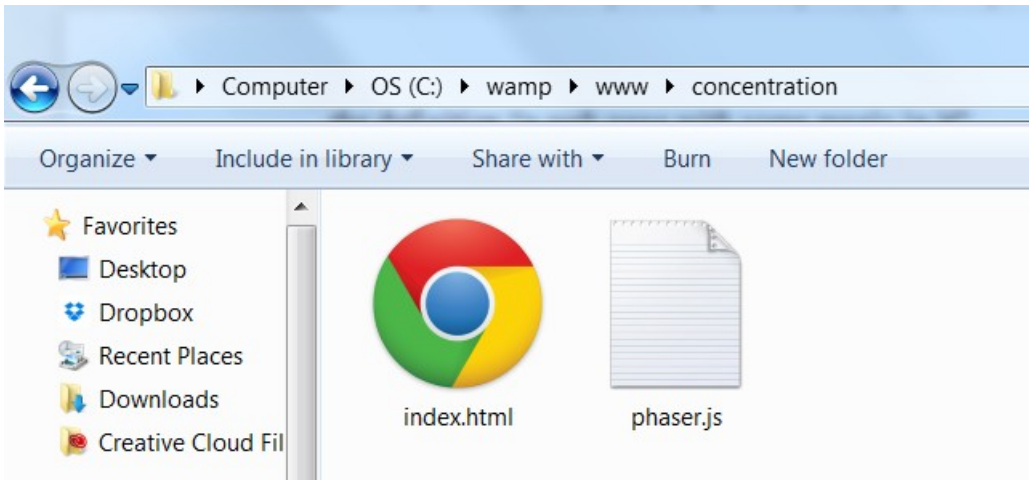
Every HTML5 game is a web page with some magic in it, so that's what we are going to do: the creation of a web page in a folder placed inside our local web server.

Without forgetting to include Phaser framework!

Inside `build` folder in the zipped package you just downloaded, you will find `phaser.js` file. That's the only Phaser file we will need during the development of our game. 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 first Phaser game.

Your project folder now should contain two files.

If you followed all instructions, your folder structure will look like this:



Now let's edit `index.html`:

```
<!DOCTYPE html>
<html>
     <head>
          <script type="text/javascript" src="phaser.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 with only a call to two JavaScript files: `phaser.js` is the file we just downloaded, and `game.js` will contain our game script.

This is the time to create a new file in the same folder, call it `game.js` and write this code:

```
window.onload = function() {
     var game = new Phaser.Game(500, 500);
}
```

Congratulations, you just created your first Phaser script.

You can use this first script as a starting template file for your future projects too.

Let's see what these lines mean:

`window.onload` is fired at the end of the document loading process, in our case when all scripts called in `index.html` have finished loading. This means everything is ready to be executed, so we can run the 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.

Inside the function there's only one line:

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

This is where everything begins.

> 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.

`Game` object will be the game itself. The two numbers inside the parentheses represent respectively the width and the height of the game, in pixels. `Game` object is assigned 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.

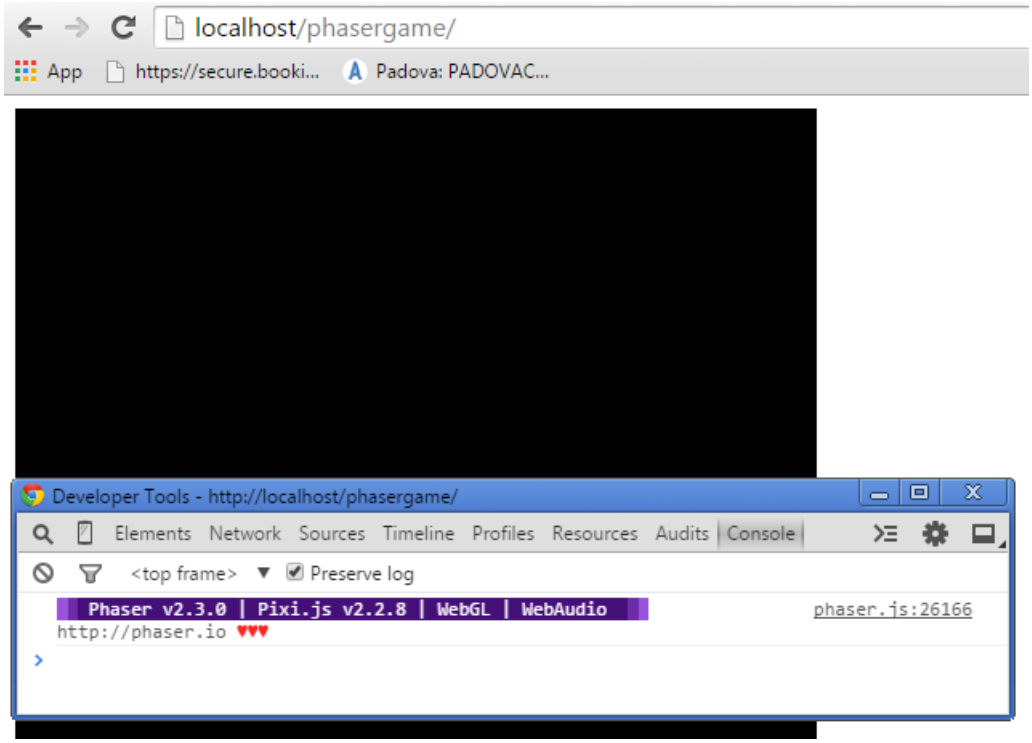
Translating what we done in English, it means “once the entire document has been loaded, create a new `Game` object with a width and height of 500 pixels and assign it to a variable called `game`.”

> 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 game

To run the game on your local server, simply point your browser to your game folder which in most cases will be http://localhost/phasergame/ and this is what you should get if running it on your Google Chrome browser:



The black square on the left is actually your 500x500 pixels game. By default, Phaser runs on a black background. You can also see Console window showing which version of Phaser you are using. This book is updated to **2.4.4**

Understanding Phaser states

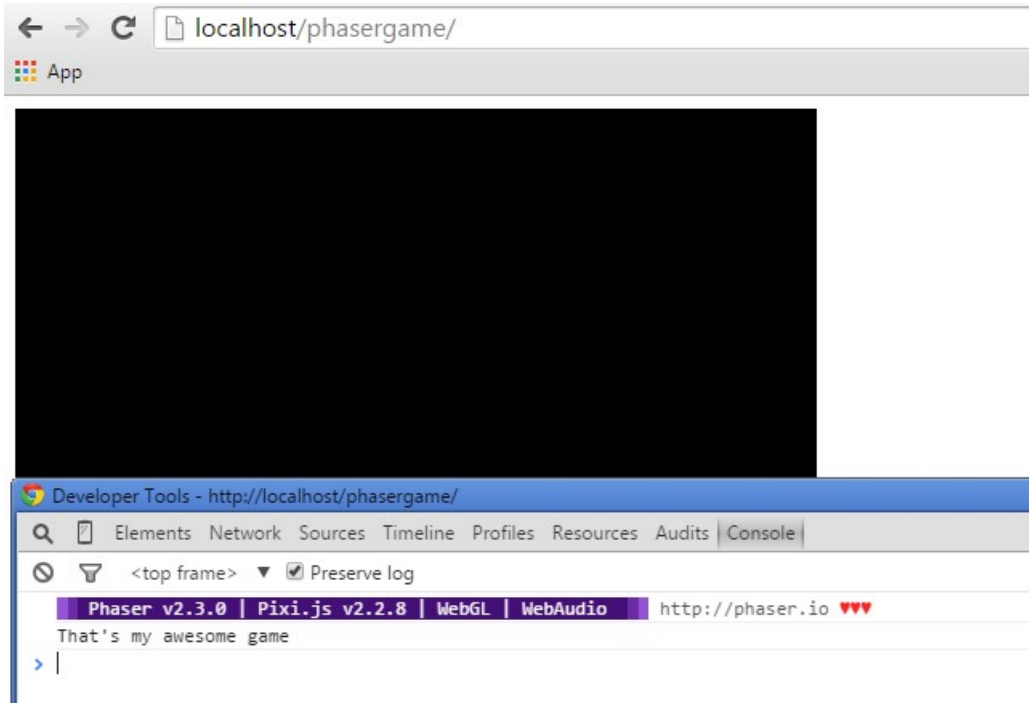
Although managing Phaser states is an advanced feature, it's very important to learn how to use states from the beginning of your Phaser programming course, as they will allow you to write better code and have a better resource management. You are going to understand this important feature while you are making your first game.

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, and a game over screen.

Each “screen” can be developed as a Phaser state, which can be executed cleaning memory and resources before it starts, allowing us to easily switch through game “screens”. To create the first state, add these lines to `game.js`:

```
window.onload = function() {
     var game = new Phaser.Game(500, 500);
     var playGame = function(game){}
     playGame.prototype = {
          create: function(){
               console.log("That's my awesome game");
          }
     }
     game.state.add("PlayGame", playGame);
     game.state.start("PlayGame");
}
```

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



What did we do? Let's have a look at the code, line by line:

```
var playGame = function(game){}
```

Here we assign a function to a variable called `playGame`. We also pass `game` variable to the function.

> 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.

```
playGame.prototype = {
     ...
}
```

We are about to define the prototype of the previously declared `playGame` variable.

> Every JavaScript object has a **prototype**. The prototype itself is also an object, and all JavaScript objects inherit their properties and methods from their prototype.

Then we have another function:

```
create: function(){
     console.log("That's my awesome game");
}
```

Inside `playGame` object we have a function called `create`. This is a reserved name used by Phaser to know which function to execute once the state 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 function 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.

Finally we can talk about the main topic of this section, Phaser states:

```
game.state.add("PlayGame", playGame);
```

Here is how we add a state to our game. The first parameter – from now on argument as said before – is the name we give to the state, and the second is the function called once the state is started.

> `state.add(key, state)` adds a new state. You must give each state a unique name in `key` argument by which you'll identify it. `state` is usually a JavaScript object or a function.

In other words, we bind `playGame` function to a state called `PlayGame`.

```
game.state.start("PlayGame");
```

And finally this is how a state is started.

> `state.start(key)` starts the state previously named with `key`.

At this time, `PlayGame` state is started, calling `playGame` function which will consequently call `playGame.create` function.

Creating tile graphics using a sprite sheet

We are going to create a board with 4 rows and 5 columns, for a total of 20 tiles on the stage.

Since each tile has a symbol on it, and since each symbol is placed twice on the board, we have to draw 20/2 = 10 images to represent the 10 different tiles, and one image to represent the back of each tile, for a grand total of 11 images.

At this time, we can save the 11 images in eleven 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 Concentration 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, and the game stops 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

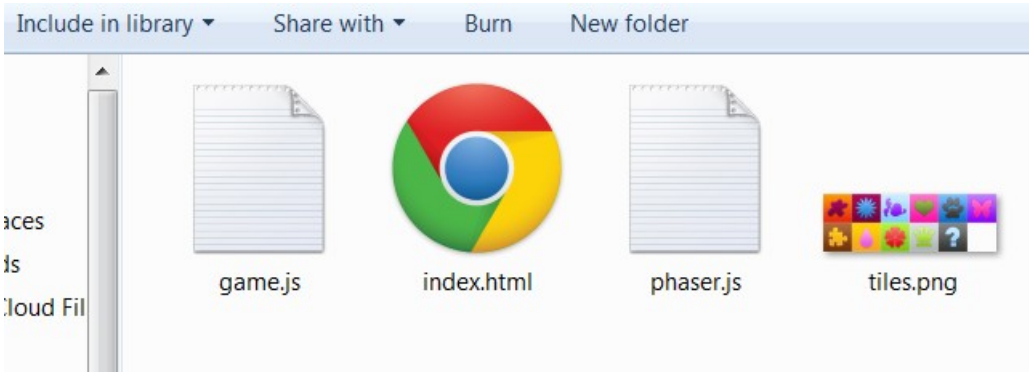
into a big image, inside an invisible grid, in order to avoid the “what image should I load” question and speed up the drawing process.

Following this concept, we are going to use one single image with all tile graphics. Here is the one I made:

It's a 6x2 grid where each tile is stored in a 80x80 pixels cell.

Now I am saving the image as `tiles.png` in the same folder where the entire project is located.

This is how your project folder should look like now:



> Always save images as **PNG** as this format has the advantages of being lossless (it does not lose quality when saved) and support alpha channel (transparency).

Now that we have the sprite sheet with all our game tiles, it's time to build the game itself.

Preloading images

One of the worst things you can do in the making of a game is to handle graphic assets before you actually loaded them.

That's why we will need to preload the sprite sheet. Let's add a couple of lines to `game.js`:

```
window.onload = function() {
     var tileSize =  80;
     var game = new Phaser.Game(500, 500);
     var playGame = function(game){}
     playGame.prototype = {
          preload: function(){
               game.load.spritesheet("tiles", "tiles.png", tileSize, tileSize);
          },
          create: function(){
               console.log("That's my awesome game");
          }
     }
     game.state.add("PlayGame", playGame);
     game.state.start("PlayGame");
}
```

What happened now?

We preloaded the sprite sheet we created some minutes ago. Let's see the new lines of code:

```
var tileSize =  80;
```

Do you remember we created the sprite sheet where each tile is an 80x80 pixels square?

We are going to refer to that number a lot of times in the making of the game, each time we will need to know the size of a tile.

Rather than filling the source code with a series of “80” scattered here and there, we are going to store this value in a variable called `tileSize`.

Not only our source code will be more readable, but above all it will be a lot easier to make changes to the game should we decide to change the size of the tile to, let's say, 60 or 100.

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

Also, notice the variable has been placed inside the `window.onLoad` function: this way we will be able to access to it from all functions declared inside `window.onLoad` function, and of course in `window.onLoad` function itself.

> 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, it's time to introduce another Phaser function to be placed inside a state.

```
preload: function(){
    ...
}
```

`preload` function, as the name suggests, is executed when the state preloads, and of course it will run before `create` function.

Since we are going to start the game itself in `create` function, it's obvious we will need to preload all stuff in `preload` function.

```
game.load.spritesheet("tiles", "tiles.png", tileSize, tileSize);
```

And finally, here's how we load a sprite sheet.

From now on, the sprite sheet is stored somewhere into the memory dedicated to the game, and we can access it whenever we need it.

> `load.spritesheet(key, url, frameWidth, frameHeight)` loads a sprite sheet and wants as arguments respectively the unique asset key of the sheet file, the URL of the sheet file, the width of each single frame and the height of each

single frame.

In our example it works this way:

`tiles` is the name we want to give to this sprite sheet. From now on, we will refer to it as “tiles”.

`tiles.png` is the name of the image we are using for the sprite sheet.

The remaining two `tileSize` arguments represent respectively the width and the height of tiles, in pixels.

We composed a sprite sheet image, and we loaded into our Phaser game. It's time to display it on the screen.

Placing images on the stage

As the sprite sheet is now preloaded, change `create` function this way:

```
create: function(){
     game.add.image(0, 0, "tiles");
}
```

Now, run the game, and that's what you are going to see:

The magic started! We finally have one image placed in the upper left corner of the stage. Which image? The first in our sprite sheet, counting from left to right, top to bottom.

`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.

In our case an image is placed at coordinates (0, 0) which is the upper left corner, with `tiles` key, which you will remember is the key we assigned to our sprite sheet image when we preloaded it.

Setting up the game field

Do you remember we are going to build a 4 rows x 5 columns game field?

Let's place some tiles on the table.

For the same reason that we declared `tileSize` variable, that is to have all main game settings assigned only once, we are going to create three new variables.

```
var tileSize =  80;
var numRows = 4;
var numCols = 5;
var tileSpacing = 10;
var game = new Phaser.Game(500, 500);
```

Let's see the meaning of these new variables:

`numRows` is the number of rows to be placed.

`numCols` is the number of columns to be placed.

`tileSpacing` is the distance between two contiguous tiles, in pixels.

We don't want to have all tiles to be placed next to each other and that's why we need to put some empty space among them: although it does not affect game play, it will make our game look better. You will spend a lot of time making your games look better, so this is definitively a good habit to be used to.

Now it's time to change `create` function once again, to add a new line:

```
create: function(){
     this.placeTiles();
}
```

Here you will see we can also define our own custom functions.

Actually, your average programming day will be full of creating custom functions.

This is the first custom function we created: it's called `placeTiles`, and will be used to place all game tiles.

> 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.

Using this, we have to write `placeTiles` function as a method of `playGame` object.

Remember, functions inside objects are called methods.

Our function declaration will be placed after `create` method, just like we wrote `create` after `preload` a few minutes ago.

```
create: function(){
     this.placeTiles();
},
placeTiles: function(){
     // function code goes here
}
```

Here is our `placeTiles` function created.

It still does nothing because there's no code to be executed inside, just a comment.

> In a line of code, everything after // is considered a **comment** and is not executed by JavaScript. Use comments to keep your source code readable.

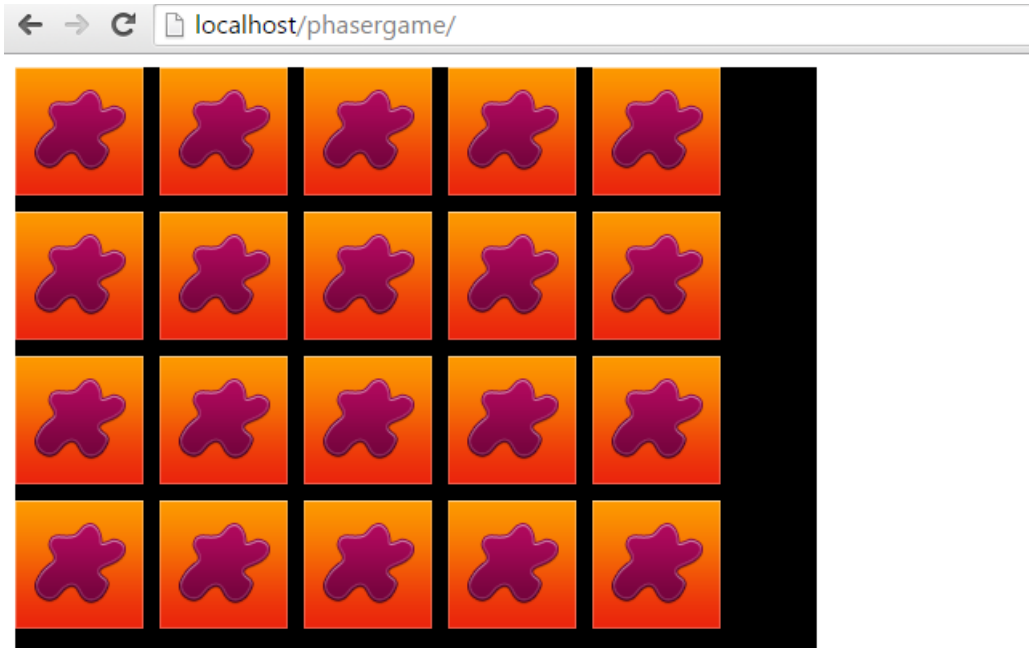
Also, remember to separate the methods inside an object with a comma, or you will get an error.

Finally we can write the code inside `placeTiles` to add the tiles on the stage:

```
placeTiles: function(){
     for(var i = 0; i < numCols; i++){
          for(var j = 0; j < numRows; j++){
          game.add.image(i * (tileSize + tileSpacing), j * (tileSize +
               tileSpacing), "tiles");
          }
     }
}
```

There's a lot of new stuff here, so let's start from the result.

Run the game, and that's what you will see:



Now we have all tiles placed in a 4 rows x 5 column grid. That's exactly what we wanted.Now let's have a look how we made it possible, analyzing `placeTiles` content line by line:

```
for(var i = 0; i < numCols; i++){
    ...
}
```

This is the first time you encounter a loop, in this case a `for` loop. Loops can execute a block of code – remember, everything included between curly brackets – a given number of times.

> The `for` loop is executed as long as a specified condition is satisfied.

`for` loops are handy, if you want to run the same code over and over again, each time with a different value. In this example the value which changes is that of variable `i`, ranging from zero to the greatest integer number smaller than `numCols` – that is the number of columns in the stage – increasing its value by one at each

loop iteration.

How can I say that? Let's analyze the syntax of a `for` loop:

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

start action is executed only once before the loop starts, just like it was on the previous line.

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.

So in our case the start action is setting `i` variable to zero. The condition is satisfied until `i` is less than `numCols`. The recurring action is adding 1 to `i` at the end of each iteration.

Inside the loop we just analyzed, we have another loop:

```
for(var j = 0; j < numRows; j++){
     ...
}
```

This works according to the same concept seen before, it just iterates through rows rather than columns and uses variable `j`.

Once `for` loops have been explained, we can jump to the core of this step, that is the line which places the tiles:

```
game.add.image(i * (tileSize + tileSpacing), j * (tileSize + tileSpacing),
"tiles");
```

You already saw how to place images with `add.image`, the concept does not change, it's just we place each image in a different position according to `i` and `j` values, which as you should already know change at each loop iteration.

At the end of both loops, we will have all the tiles correctly placed on the screen.

Anyway, there is a better way to place tiles: if you look how we placed the tiles, you will see the upper left tile is placed next to the upper left corner of the stage.

This is not an error, but the board does not look good as it's not centered in the stage.

That's what we are going to do now.

Adjusting assets placement according to stage size

Placing assets according to game size is really easy, once you know game size.

If you look at `game` declaration, you can see its width and height are 500 pixels, but let's suppose we don't know it, or we want to have the freedom to modify game size by changing only its declaration, without any further search/replace. Phaser gives us some properties to get the width and height of a Phaser `Game` instance.

> A **property** of an object is a value associated to a variable within the object itself. Usually properties can be read and written, but some of them can only be read.

So we are adding a couple of lines to `placeTiles` function:

```
placeTiles: function(){
     var leftSpace = (game.width - (numCols * tileSize) - ((numCols - 1) *
       tileSpacing))/2;
     var topSpace = (game.height - (numRows * tileSize) - ((numRows - 1) *
       tileSpacing))/2;
     for(var i = 0; i < numCols; i++){
          for(var j = 0; j < numRows; j++){
               game.add.image(leftSpace + i * (tileSize + tileSpacing), topSpace +
               j * (tileSize + tileSpacing), "tiles");
          }
     }
}
```

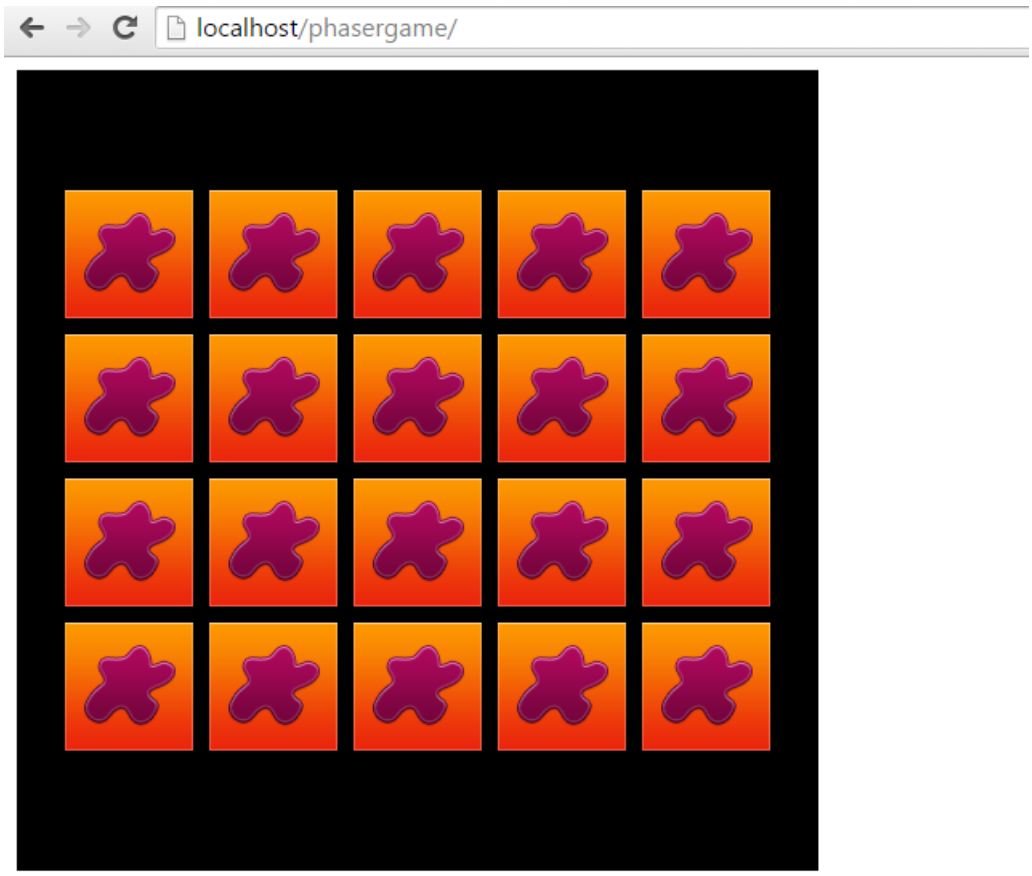
Let's have a look at how we determined the value of `leftSpace`, which is basically the left offset used to place tiles:

```
var leftSpace = (game.width - (numCols * tileSize) - ((numCols - 1) *
tileSpacing))/2;
```

We subtract to game width the length of all tiles (`numCols * tileSize`) and the length of all empty spaces between two tiles (`(numCols - 1) * tileSpacing`) getting the difference between game width and the width actually used by the tiles. Dividing this difference by two we can find the amount of pixels to move to the left each tile to have the board centered on the game stage.

`width` and `height` properties return width and height of a Phaser `Game` instance.

The same concept applies to vertical offset. Launch the game now:



Now we have our tile board centered in the game stage thanks to two game

properties.

Anyway, shouldn't the tiles start as covered, that is showing the question mark? It seems we are only able to show the first frame of our sprite sheet.

Let's show images properly.

Displaying given frames in a sprite sheet

Before we start typing some new code, here is how sprite sheet works: each image is identified by a number, called **index**, which starts from **zero**.

So starting from the upper left image and proceeding left to right, top to bottom, each image is given an index.

The first image has index “zero”, the second image has index “one” and so on, the third has index “two” and so on.

If you look at the sprite sheet, the question mark is the 11th image, and it means we have to show the 10th frame of the sprite sheet.

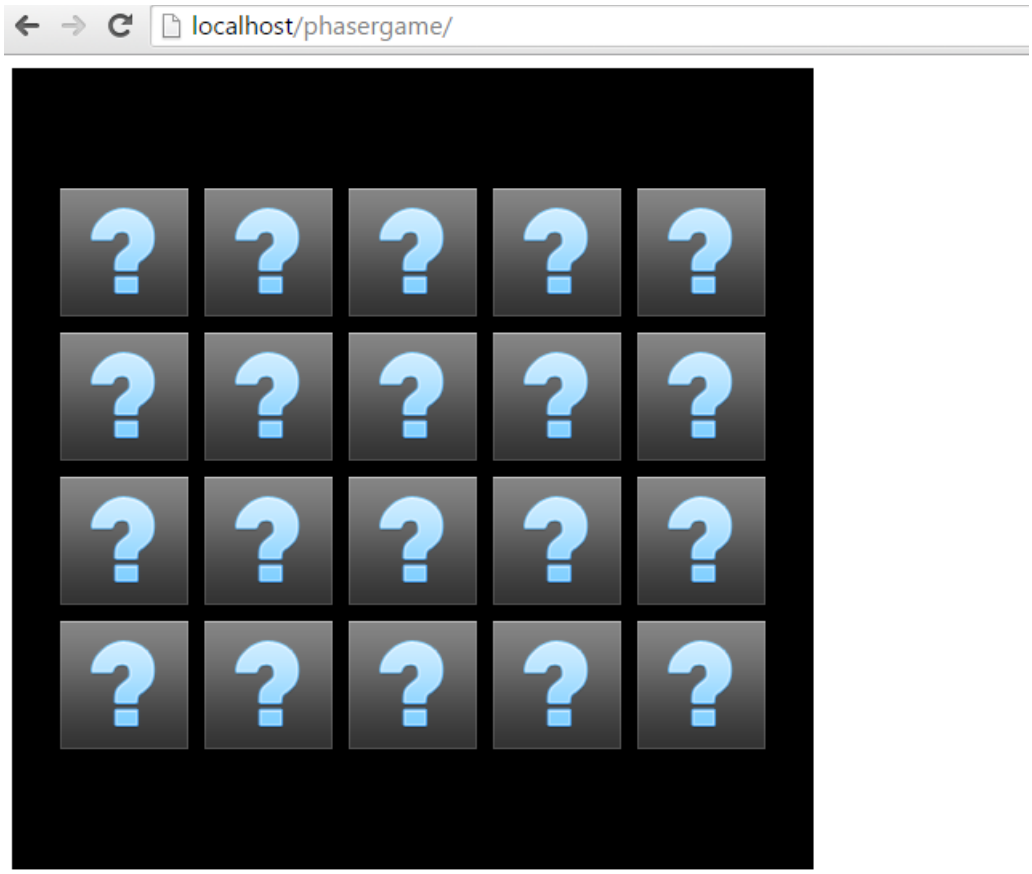
`frame` property of an `image` object allows us to show the frame we want.

With just a small change to `placeTiles` function:

```
placeTiles: function(){
     var leftSpace = (game.width - (numCols * tileSize) - ((numCols - 1) *
       tileSpacing))/2;
     var topSpace = (game.height - (numRows * tileSize) - ((numRows - 1) *
       tileSpacing))/2;
     for(var i = 0; i < numCols; i++){
          for(var j = 0; j < numRows; j++){
               var tile = game.add.image(leftSpace + i * (tileSize + tileSpacing),
               topSpace + j * (tileSize + tileSpacing), "tiles");
               tile.frame = 10;
          }
     }
}
```

We are able to turn all tiles covered with frame property, but first we have to assign the image a variable, here called `tile`.

Launch the game and see what happens:



Tiles now show the question mark, and are ready to be turned to show their actual symbol. Now it's time to let the player interact with the game.

Adding interactivity to images by turning them into clickable and touchable buttons

As you know we are going to build a cross platform game. This means it has to run properly on each device, from desktop computers to smart phones and tablets.

Unfortunately, different devices use different ways to let the player interact with the game: in our case, on desktop computers players will click on a tile with the mouse to select it, while on a smart phone a tile will be selected with a touch.

Do we have to check for both mouse clicks and finger touches? No, because

Phaser handles everything internally, letting us focus on the game itself.

To give tiles the capability of being clicked or touched, we have to turn them into buttons.

Time to edit `placeTiles` function once more:

```
placeTiles: function(){
     var leftSpace = (game.width - (numCols * tileSize) - ((numCols - 1) *
       tileSpacing))/2;
     var topSpace = (game.height - (numRows * tileSize) - ((numRows - 1) *
       tileSpacing))/2;
     for(var i = 0; i < numCols; i++){
          for(var j = 0; j < numRows; j++){
               var tile = game.add.button(leftSpace + i * (tileSize +
               tileSpacing), topSpace + j * (tileSize + tileSpacing), "tiles",
               this.showTile, this);
               tile.frame = 10;
          }
     }
}
```

When you use a button rather than an image, interactivity is automatically added by Phaser.

> `add.button(x, y, key, callback, callbackContext)` adds a button at coordinates (`x`,`y`) using the image stored with `key` value. `callback` is the function to call when the button is pressed. `callbackContext` is the context in which the callback will be called, is usually `this` because it's a reference to the object that owns the currently executing code.

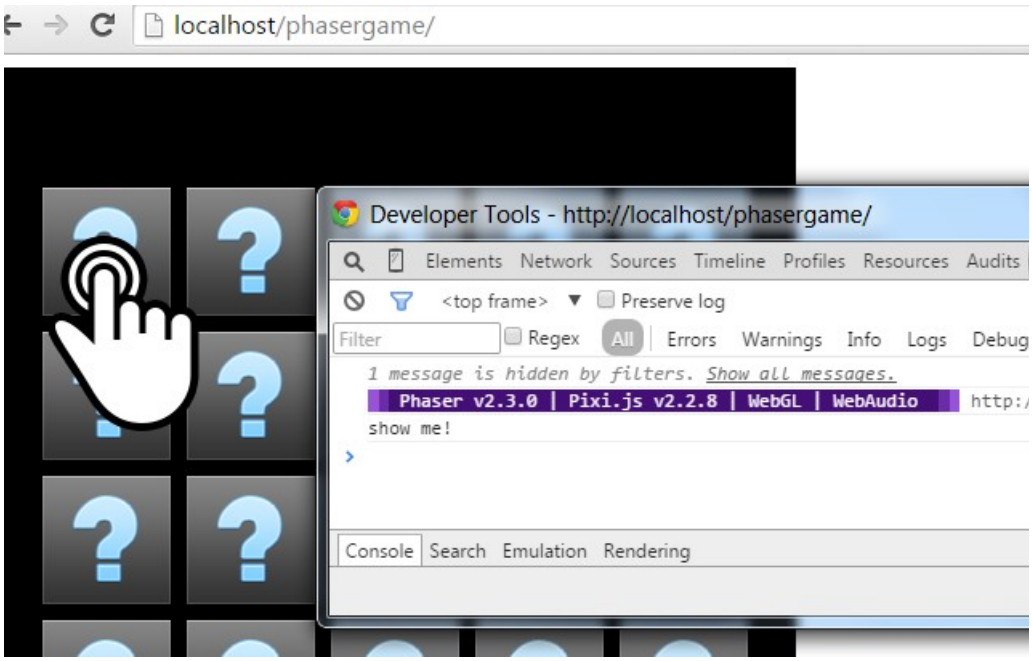
Looking at the syntax of button creation, you can see we have to add a function called `showTile`, and the presence of `this` suggests it's another method of `playGame` object.

```
PlaceTiles: function(){
     ...
},
showTile: function(){
     console.log("show me!");
}
```

We don't want `showTile` function to do that much at the moment, so we are just

showing a message in the console.

Run the game, click on any tile and see what happens:



Thanks to Phaser, turning an image into an interactive element was really easy, but now a difficult task awaits us: we have to turn a button into something more complex. Since each button represents a tile, we will need to store the value of each tile – which is the symbol to be shown when we turn it – somewhere.

Giving each button a custom property and accessing it when touched/clicked

Before we continue interacting with tiles and buttons, we have to store somewhere all tile values. And above all, assign each tile a value which will represent its image.

Since we have 10 different types of tiles, each tile type can easily be coded with a number ranging from 0 to 9. Also, having two tiles for each type, the numeric representation of all tiles will be: 0, 0, 1, 1, 2 … 8, 8, 9, 9.

Where to store these values? In an array.

> 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.

You will master arrays in a couple of minutes, at the moment let's create a new variable called `tilesArray` which will store all tiles information. At the beginning it will be an empty array, which we will populate later in the code.

```
var tileSize =  80;
var numRows = 4;
var numCols = 5;
var tileSpacing = 10;
var tilesArray = [];
var game = new Phaser.Game(500, 500);
```

`tilesArray` will now be populated with tile values as mentioned above.

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

Populate the array and assign a custom property to tiles changing `placeTiles`:

```
placeTiles: function(){
     var leftSpace = (game.width - (numCols * tileSize) - ((numCols - 1) *
       tileSpacing))/2;
     var topSpace = (game.height - (numRows * tileSize) - ((numRows - 1) *
       tileSpacing))/2;
     for(var i = 0; i < numRows * numCols; i++){
          tilesArray.push(Math.floor(i / 2));
     }
     console.log("my tile values: " + tilesArray);
     for(i = 0; i < numCols; i++){
          for(var j = 0; j < numRows; j++){
               var tile = game.add.button(leftSpace + i * (tileSize +
                      tileSpacing), topSpace + j * (tileSize + tileSpacing),
                      "tiles", this.showTile, this);
               tile.frame = 10;
               tile.value = tilesArray[j * numCols + i];
          }
     }
}
```

As you can see, the function has been modified with some new code: there's a new `for` loop at the beginning, something changed in the loop which iterates through columns and there's a new line near the end of the function.

Let's examine in detail what changed:

```
for(var i = 0; i < numRows * numCols; i++){
     tilesArray.push(Math.floor(i / 2));
}
```

This is a `for` loop – you should already be familiar with it – which iterates for `numRows * numCols` times, that is the total amount of tiles.

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

At each iteration, `push` method adds a new item to the end of `tilesArray` adding zero, then zero again, then one, then one again until a couple of nines.

> `Math.floor` rounds a number downward to its nearest integer.

Just to figure out what `tilesArray` contains at this time, let's output its content to browser console:

```
console.log("my tile values: " + tilesArray);
```

Now, let's see the start action of next loop:

```
for(i = 0; i < numCols; i++){
     ...
}
```

Now it's `i = 0` rather than `var i = 0`. Why?

Because you have already declared `i` variable in the previous loop inside the same function. You only need to declare a variable once.

```
tile.value = tilesArray[j * numCols + i];
```

And finally we can give each tile its proper value.

`value` isn't a built-in property of `tile` object, it's just a custom variable we are assigning to `tile`. In other words, we are storing each tile value somewhere inside

the tile itself. Somewhere, but exactly where? Inside `value` property.

> You **access an array element** with index i by including the index between square brackets. The array element with index 0 – the first element – is accessed with `[0]`.

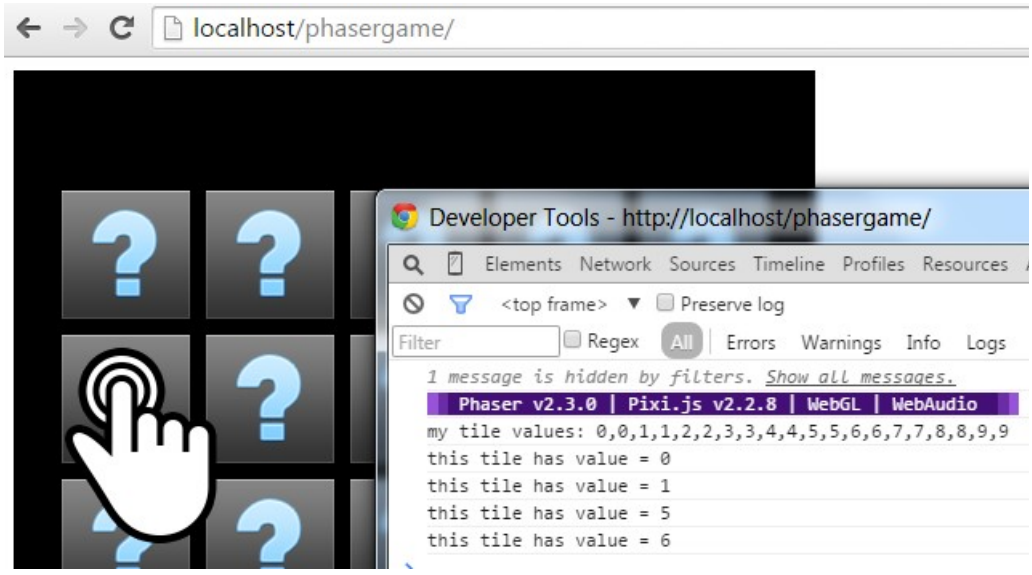
This way, the first tile placed will have `value` property equal to the first array element, that is `tilesArray[0]`, the second tile placed will have `value` property equal to the second array element, that is `tilesArray[1]`, and so on.

Are you ready for a test drive? Let's change `showTile` function this way:

```
showTile: function(target){
     console.log("this tile has value = " + target.value);
}
```

The button which is clicked/touched is passed in `showTile` function as an argument called `target`, then we access to its `value` property, which we previously set.

Run the game and select some tiles to see their values in the console window:



Everything seems to work the right way, at least behind the curtains. Actually,

players won't see the values of the tiles they select.

Showing tiles once selected

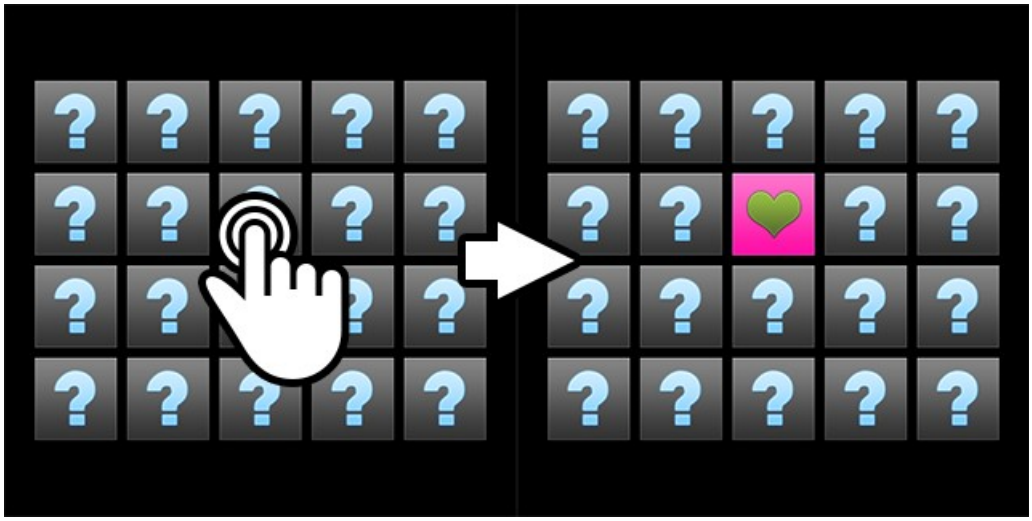
To give players a visual feedback of the tiles they select, we need to change their frame according to their value once selected.

We know each tile value since it's stored in `value` custom property, so we only need to change its frame when selected. Here is the one and only line we have to write to make this happen:

```
showTile: function(target){
     target.frame = target.value;
}
```

Do you remember `frame` property? That's all we need to show actual tile content.

Run the game, select tiles and see their contents.



This is a great step further in the making of the game, but it's not enough, as at the moment you are able to select how many tiles as you want, while in the original game you must select only two tiles each turn. We need a way to allow the player to select only two tiles.

Preventing the player to select more than two tiles each turn

As said, not only we have to let the player select only two tiles, but we also have to prevent the same tile from being selected twice. A selected tile can't be selected again. Moreover, we must store selected tiles somewhere, as later in the development of the game we must check if tile symbols match.

First things first, let's create a new variable, an array which will store selected tiles.

We will call it `selectedArray`, and it will start as an empty array:

```
var tileSize =  80;
var numRows = 4;
var numCols = 5;
var tileSpacing = 10;
var tilesArray = [];
var selectedArray = [];
var game = new Phaser.Game(500, 500);
```

Now, we said `selectedArray` will contain selected tiles, so it means we have to add tiles in it once they are selected by the player.

But according to game rules, before inserting a new tile in the array we have to check if the player hasn't already selected two tiles and the currently selected tile hasn't been already selected.

This may sound complicated, but can be achieved in only two lines of code added to `showTile` function:

```
showTile: function(target){
     if(selectedArray.length < 2 && selectedArray.indexOf(target) == -1){
          target.frame = target.value;
          selectedArray.push(target);
     }
}
```

Run the game, and you'll see you won't be able to select more than two different tiles.

What changed in the script? First, we added something you are already familiar with:

```
selectedArray.push(target);
```

You should remember `push` method adds a new item to the end of an array, in this case `selectedArray`.

Then, there's another line which introduces some new concepts:

```
if(selectedArray.length < 2 && selectedArray.indexOf(target) == -1){
     ...
}
```

First, let me introduce you the `if` statement.

Very often when you write code, you want to perform different actions for different decisions, or perform some actions only in some cases.

You can use conditional statements in your code to do this, and `if` is the most common statement.

> `if` statement executes a block of JavaScript code if a given condition is true.

We can write the average if statement in this form:

```
if(condition){
    // block of code to be executed if the condition is true
}
```

So now we know how if statement works.

Now let's see the condition in detail:

```
selectedArray.length < 2 && selectedArray.indexOf(target) == -1
```

We are checking the length of `selectedArray`. If it's less than two, this means we still haven't selected two tiles.

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

We are also checking if the currently selected tile is already in `selectedArray` array, which means we already selected that tile.

`indexOf` method returns the position of a given element in the array. If there is no occurrence, `indexOf` returns `-1`.

In this case, we check for `indexOf` to be equal to `-1`, to be sure there's no occurrence of the selected tile in `selectedArray` array.

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

The whole `if` condition will be true only if both conditions are true.

`&&` is the **and** logical operator used to connect two conditions

Translated in everyday language, we wrote "if the selected array has less than two elements and there's no occurrence in the array of the tile we just selected, then insert the tile in the array".

A lot of new concepts in just a couple of lines, but now the player will follow the rules of the game, at least during the selection of tiles.

Now there is another rule to develop, the most important one, because it's the one which will reward player's memory: if the player selects two tiles with the same value they will be removed from the stage.

We will cover back the tiles otherwise.

Checking for successful matches and removing tiles or turning them back

Once we have selected two tiles, then `selectedArray` has two elements, and we have to check for their values.

We will remove the tiles if values match, or cover back them again if values do not match.

This means some more lines added to `showTile` function:

```
showTile: function(target){
     if(selectedArray.length < 2 && selectedArray.indexOf(target) == -1){
          target.frame = target.value;
          selectedArray.push(target);
     }
     if(selectedArray.length == 2){
          if(selectedArray[0].value == selectedArray[1].value){
               selectedArray[0].destroy();
               selectedArray[1].destroy();
          }
          else{
               selectedArray[0].frame = 10;
               selectedArray[1].frame = 10;
          }
          selectedArray.length = 0;
     }
}
```

Let's see the new code in detail:

```
if(selectedArray.length == 2){
     ...
}
```

This is how we check for `selectedArray` to have two elements, so it's time to check for tiles to have the same value:

```
if(selectedArray[0].value == selectedArray[1].value){
     ...
}
```

We access `value` custom property of the first and the second `selectedArray` elements and check for them to be equal.

If they are equal, it's time to remove the tiles from the stage.

> `destroy` method permanently destroys the button, destroys the input event and animation handlers if present and nulls its reference to game, freeing it up for garbage collection.

It's the best way to remove the button and never think about it again, so let's destroy both buttons.

```
selectedArray[0].destroy();
selectedArray[1].destroy();
```

And now we are done when the player selected tiles with the same symbols.

What if tiles have different symbols?

We have to use an extension to `if` statement called `else`.

> Use the `else` statement to specify a block of code to be executed if the condition in the `if` statement is false.

It works this way:

```
if(condition){
     // block of code to be executed if the condition is true
}
else{
     // block of code to be executed if the condition is false
}
```

So when the condition “selected tiles have the same value” is false, this is what happens:

```
else{
     selectedArray[0].frame = 10;
     selectedArray[1].frame = 10;
}
```

We simply change `frame` properties of both tiles to turn them back again.

Finally, no matter whether the tiles have been removed or covered back, we need to empty `selectedArray` array to let the player select a new couple of tiles.

To empty an array, we set its length to zero:

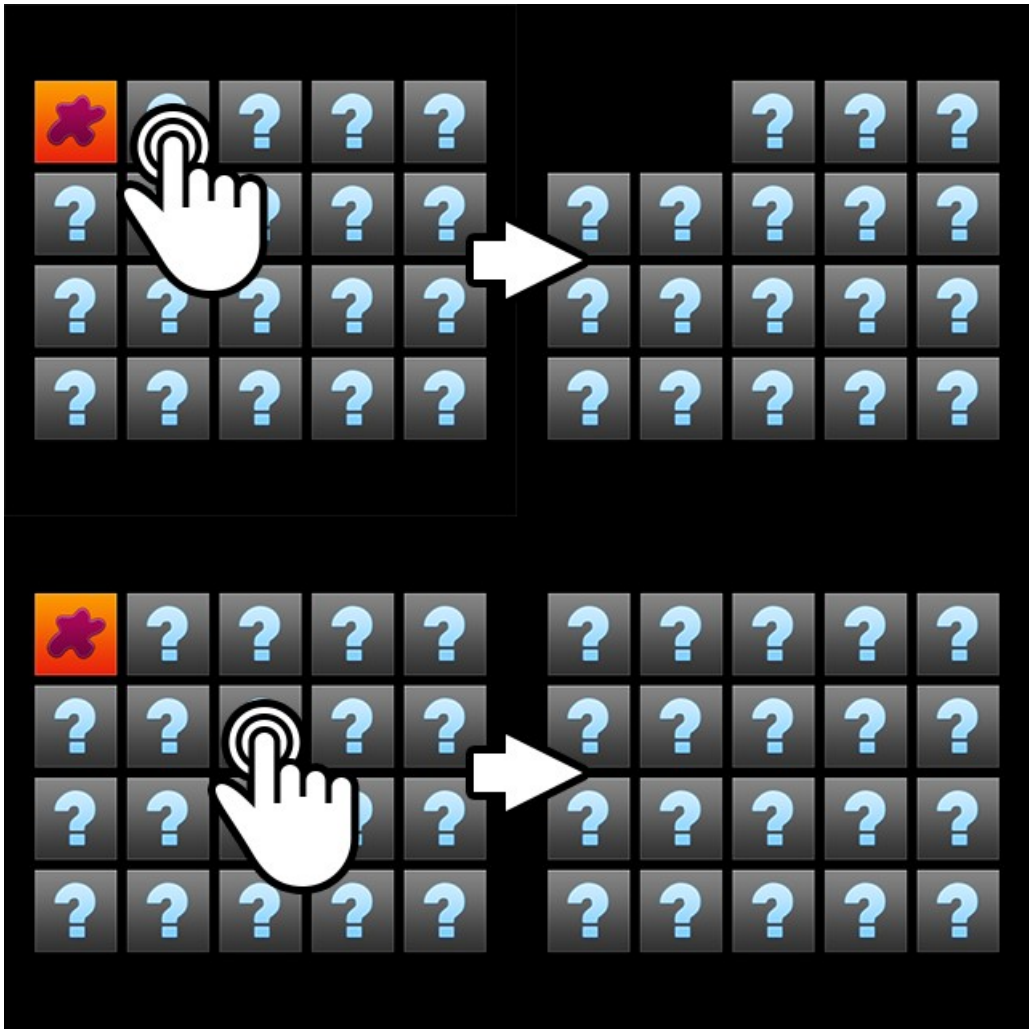
```
selectedArray.length = 0;
```

If you browse the web, you will find another way to empty arrays, for example by setting them back to empty array definition assigning it to `[]`.

I personally prefer setting array length to zero in JavaScript, and an explanation for this reason is beyond the scope of this project.

You are free to choose the way you prefer to empty an array, just make sure `selectedArray` is now empty or the player won't be able to select any more tile.

Time to run the game, and try to make both matching and non matching selections.



Now matching tiles will be removed from the stage, while non-matching tiles will be covered back.

Everything works fine, it's just you aren't able to see the second tile you select, are you?

This happens because as soon as you select the second tile, we perform the check for matching tiles and remove/cover selected tiles so fast you can't even see what happened.

This is obviously a bad game design practice, so we will wait a second with the selected tile before removing/covering them.

Using timers to schedule events

Phaser allows us to create a variety of time driven events, in a simple and intuitive way.

> `time.events.add(tick, callback, callbackContext)` adds a timer event which will execute `callback` function in the `callbackContext` context after `tick` time has passed.

`Phaser.Timer.SECOND` is a Phaser reserved variable which means “one second”, `checkTiles` is the name of the function we are going to call after a second we realized the player uncovered two tiles.

The use of `this` should now clearly suggest you how to declare `checkTiles` function.

```
showTile: function(target){
     if(selectedArray.length < 2 && selectedArray.indexOf(target) == -1){
          target.frame = target.value;
          selectedArray.push(target);
          if(selectedArray.length == 2){
               game.time.events.add(Phaser.Timer.SECOND, this.checkTiles, this);
          }
     }
},
checkTiles: function(){
     // function code goes here
}
```

What should we write in `checkTiles` function?

Exactly the same code we used for checking tiles when there is a match and

remove/cover them.

```
checkTiles: function(){
     if(selectedArray[0].value == selectedArray[1].value){
          selectedArray[0].destroy();
          selectedArray[1].destroy();
     }
     else{
          selectedArray[0].frame = 10;
          selectedArray[1].frame = 10;
     }
     selectedArray.length = 0;
}
```

Now run the game, and you will see the game waits a second with both tiles shown on the stage before removing or covering back them.

Now, the last – but not least – difference between our game and a real Concentration game: our tiles are always placed in the same, predictable, place.

We have to spice up a bit the game by shuffling the tiles.

Shuffling the tiles

The basics behind shuffling the tiles is to shuffle the array of tile values, that `tileArray` which will be used to assign each tile its own value.

There are a lot of ways to shuffle an array and they all have their roots on random number generation, which is a branch with countless theories.

While in a casino game with real money prizes true randomness is really important, in a quick puzzle game this does not make sense, as the basic random functions provide a great deal of randomness which is good enough to have a completely different boards each time we play.

So we are going to change `placeTiles` function in the most basic way with just randomly switching two `tilesArray` elements a given amount of times, and `numRows * numCols` times is enough to have a completely shuffled board.

Change `placeTiles` function this way:

```
placeTiles: function(){
     var leftSpace = (game.width - (numCols * tileSize) - ((numCols - 1) *
       tileSpacing))/2;
     var topSpace = (game.height - (numRows * tileSize) - ((numRows - 1) *
       tileSpacing))/2;
     for(var i = 0; i < numRows * numCols; i++){
          tilesArray.push(Math.floor(i / 2));
     }
     for(i = 0; i < numRows * numCols; i++){
          var from = game.rnd.between(0,tilesArray.length-1);
          var to = game.rnd.between(0, tilesArray.length-1);
          var temp = tilesArray[from];
          tilesArray[from] = tilesArray[to];
          tilesArray[to] = temp;
     }
     for(i = 0; i < numCols; i++){
          for(var j = 0; j < numRows; j++){
               var tile = game.add.button(leftSpace + i * (tileSize +
               tileSpacing), topSpace + j * (tileSize + tileSpacing),
               "tiles", this.showTile, this);
               tile.frame = 10;
               tile.value = tilesArray[j * numCols + i];
          }
     }
}
```

Now run the game and play, and have another go, then another one, and so on. You will notice each time you start a game, tiles are placed in a different way.

That's enough, and the code involved in this process is just another `for` loop where at each iteration two randomly chosen elements in `tilesArray` array are swapped, using a temporary variable because given two values A and B, JavaScript does not allow us to swap A with B, but we can save A value in C variable. Then assign A the value of B, then assign B the value of C which is the saved A value.

What I want you to focus in this loop is how I generate the random numbers to assign to each array index for swapping.

> `rnd.between(min, max)` generates a random integer number ranging from `min` to `max`, both included.

And this last feature completes our game, a nice game which nobody will want to play.

Although the game runs and works well, you can't believe people will play games

without a theme, without a goal, without any twist.

Probably in 1970's players would have loved your game, but today they won't.

Are we going to put the entire game in the trashcan?

No, we can still make something interesting out of it, by just adding a couple of features and polish a bit the game.

Turning the prototype into a real game adding a title screen with sound/mute options

The first thing we need to think about when turning the prototype into a real game, is the title screen, which means giving the game a name, create a visual look and feel that has something to do with the title, and a couple of nice sound/mute buttons to place in the title screen.

Since we are about to create a cross platform game which will be able to run on phones and mobile devices everywhere, and since it's a quick casual game, you can expect it to be played at the office or at school.

That said, it's very important to start with a muted title screen where the player can clearly choose whether to play with sounds or not.

So we will first need to create a new variable called `playSound`. Here we will store the player decision.

```
var tileSize =  80;
var numRows = 4;
var numCols = 5;
var tileSpacing = 10;
var tilesArray = [];
var selectedArray = [];
var playSound;
var game = new Phaser.Game(500, 500);
```

Remember that at the very beginning when the game was being created I introduced game states?? We only have one state at the moment, but now it's time to add more states and add complexity to our game in only a couple of lines.

This would have been way more difficult if you didn't learn how to manage states.

We are going to add a new state called `TitleScreen` which points to `titleScreen` object, and start the game launching it rather than `PlayGame`.

```
game.state.add("TitleScreen", titleScreen);
game.state.add("PlayGame", playGame);
game.state.start("TitleScreen");
```

Now, in the same way we created `playGame` object with all its `preload` and `create` methods at the beginning of this journey, we are doing the same with `titleScreen`. Most of the concepts you'll see here have been already explained.

```
var titleScreen = function(game){}
titleScreen.prototype = {
     preload: function(){
          game.load.spritesheet("soundicons", "soundicons.png", 80, 80);
     },
     create: function(){
          var style = {
               font: "48px Monospace",
               fill: "#00ff00",
               align: "center"
          };
          var text = game.add.text(game.width / 2, game.height / 2 - 100,
       "Crack Alien Code", style);
          text.anchor.set(0.5);
          var soundButton = game.add.button(game.width / 2 - 100 ,
       game.height / 2 +      100, "soundicons", this.startGame, this);
          soundButton.anchor.set(0.5);
               soundButton = game.add.button(game.width / 2 + 100 , game.height /
       2 + 100, "soundicons", this.startGame, this);
          soundButton.frame = 1;
          soundButton.anchor.set(0.5);
     },
     startGame: function(target){
          if(target.frame == 0){
               playSound = true;
          }
          else{
               playSound = false;
          }
          game.state.start("PlayGame");
     }
}
```

A lot of code as you can see, but you already know most of the concepts used.

Let me highlight the most interesting parts:

```
preload: function(){
     game.load.spritesheet("soundicons", "soundicons.png", 80, 80);
}
```

I created another sprite sheet with two 80x80 pixels images, one representing “sound on” button, and one representing “sound off”. Then I preloaded it.

Sound buttons are then added to the stage, inside `create` function:

```
var soundButton = game.add.button(game.width / 2 - 100 , game.height / 2 + 100,
"soundicons", this.startGame, this);
soundButton.anchor.set(0.5);
soundButton = game.add.button(game.width / 2 + 100 , game.height / 2 + 100,
"soundicons", this.startGame, this);
soundButton.frame = 1;
soundButton.anchor.set(0.5);
```

This is the standard code used to create a button. Just have a look at the callback function – `startGame` – and at a new concept called anchor point.

> The **anchor** sets the origin point of the texture. The default is `0,0` this means the texture's origin is the top left. Setting than anchor to `0.5,0.5` means the textures origin is centered. Setting the anchor to `1,1` would mean the textures origin points will be the bottom right corner. Two equal values can be written only once. `anchor.set(0.5,0.5)` can be written as `anchor.set(0.5)`.

In this case with we want buttons to have their anchor points in its horizontal and vertical center. Just remember the image should have an even with and height, or the final result will look a bit blurred.

Now, let's use some text to give the game a name: “Crack Alien Code”, something more catchy than “Concentration”.

```
var style = {
     font: "48px Monospace",
     fill: "#00ff00",
     align: "center"
}
```

First, let's decide the look of the text: an object called `style` which contains `font`,

`fill` and `align` properties. Once the style has been defined, it's time to write the text on the screen:

```
var text = game.add.text(game.width / 2, game.height / 2 - 100, "Crack Alien
Code", style);
```

Adding text is not that different than adding images or buttons.

> `add.text(x, y, text, style)` adds a text in x,y position, writing `text` string using `style` style

Launch the game, and have a look at the brand new game title screen:

Game name and two big buttons to choose sound preferences. This starts to look

like a real game.

Do you remember those two buttons calling `startGame` function? Here it is:

```
startGame: function(target){
     if(target.frame == 0){
        playSound = true;
     }
     else{
        playSound = false;
     }
     game.state.start("PlayGame");
}
```

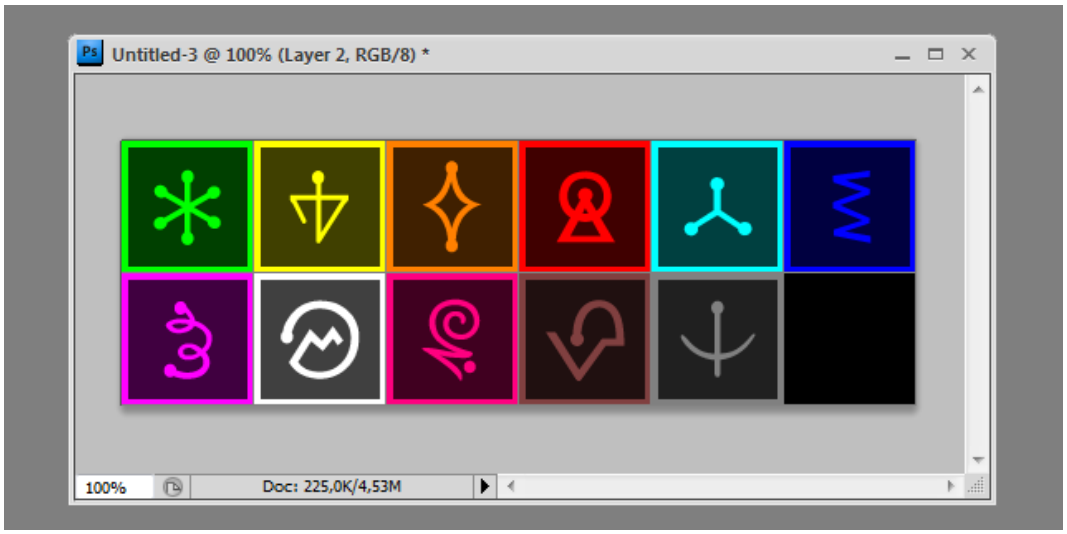
The first frame – remember, frame zero – represents the “sound on” icon, while the second frame – frame one – represents the mute button. By checking the frame of the selected button we can set `playSound` to `true` or `false`.

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

The last thing to do when the player presses sound buttons is starting the game:

```
game.state.start("PlayGame");
```

That's it, in a single line of code using states. Oh, and I changed tiles symbols:



Now they look less like “my first Photoshop graphics” and more like alien runes.

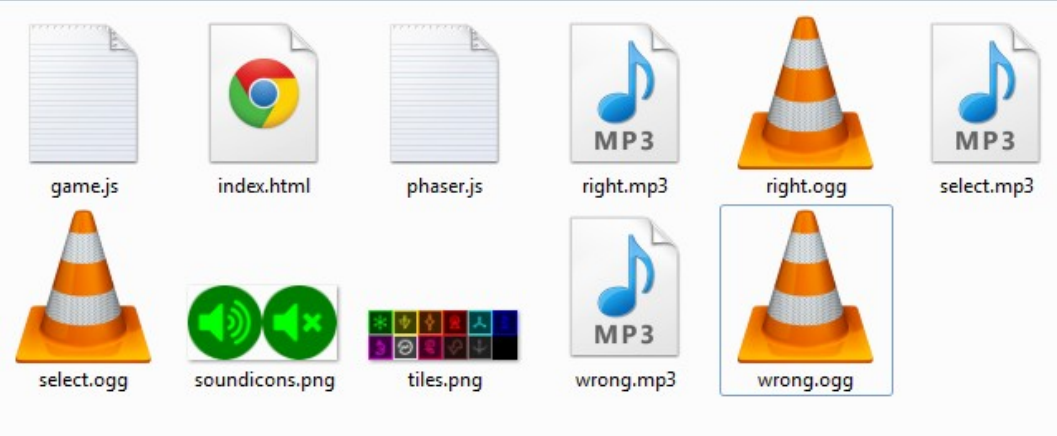
During the making of your games, you will often find yourself changing the graphics again and again, especially when you are turning a prototype into a playable game, so don't be surprised I made it on the book, it's part of the game creation process.

Preloading sounds

When the player chooses to play with sounds, it means the game should feature sounds.

Crack Alien Code will have three sounds: one to be played each time a tile is selected, one to be played when the player makes a successful match and one to be played when the player makes an unsuccessful match.

I added in my game folder three new sounds, in two different formats: **mp3** and **ogg** and now the folder looks 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 `playGame` object:

```
preload: function(){
     game.load.spritesheet("tiles", "tiles.png", tileSize, tileSize);
     game.load.audio("select", ["select.mp3", "select.ogg"]);
     game.load.audio("right", ["right.mp3", "right.ogg"]);
     game.load.audio("wrong", ["wrong.mp3", "wrong.ogg"]);
}
```

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 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.

Playing sounds

The idea is to store all sounds in an array and then play the right song according to what's going on in the game.

We are going a property in `playGame` object called `soundArray`.

It's defined as an empty array and will be visible only inside playGame object.

We declare it this way:

```
playGame.prototype = {
     soundArray: [],
     preload: function(){
          game.load.spritesheet("tiles", "tiles.png", tileSize, tileSize);
          game.load.audio("select", ["select.mp3", "select.ogg"]);
          game.load.audio("right", ["right.mp3", "right.ogg"]);
          game.load.audio("wrong", ["wrong.mp3", "wrong.ogg"]);
     },
     ...
}
```

In the same `playGame` object, inside `create` method once we called `placeTiles` and we add the audio if the player selected the option to play with sound, that is if `playSound` is set to `true`.

A small change to `create` method:

```
create: function(){
     this.placeTiles();
     if(playSound){
          this.soundArray[0] = game.add.audio("select", 1);
          this.soundArray[1] = game.add.audio("right", 1);
          this.soundArray[2] = game.add.audio("wrong", 1);
     }
}
```

To insert elements into an array we always used `push` method but you can also declare items one by one assigning a value for a given index, like I did in these latest lines of code.

> `add.audio(key, volume)` adds a new audio file to the sound manager. `key` is the name we gave to the sound, while `volume` is playing volume. It ranges from 0 to 1 where 1 means maximum volume.

Now that the sounds are preloaded and ready to be played, it's time to insert our first sound into `showTile` method. This will be played when a tile is shown:

```
showTile: function(target){
     if(selectedArray.length < 2 && selectedArray.indexOf(target) == -1){
          if(playSound){
               this.soundArray[0].play();
          }
          target.frame = target.value;
          selectedArray.push(target);
          if(selectedArray.length == 2){
               game.time.events.add(Phaser.Timer.SECOND, this.checkTiles, this);
          }
     }
}
```

Obviously before playing any sound effect we must check for `playSound` to be `true`.

> `play()` method plays a sound.

Play the game, pick tiles and you should hear the sound effect.

In the same way it's easy to add sounds to `checkTiles` method, one to be played when the player made a successful match and one to be played when there wasn't any match.

```
checkTiles: function(){
     if(selectedArray[0].value == selectedArray[1].value){
          if(playSound){
               this.soundArray[1].play();
          }
          selectedArray[0].destroy();
          selectedArray[1].destroy();
     }
     else{
          if(playSound){
               this.soundArray[2].play();
          }
          selectedArray[0].frame = 10;
          selectedArray[1].frame = 10;
     }
     selectedArray.length = 0;
}
```

Sounds gave the game a deeper experience, but there's still a missing key feature, which is the one we all play for: the score.

Showing the score

To keep track of the score, I am sure you find it obvious now, we need to store it in a variable. Let's create a variable called `score`.

```
var tileSize =  80;
var numRows = 4;
var numCols = 5;
var tileSpacing = 10;
var tilesArray = [];
var selectedArray = [];
var playSound;
var score;
var game = new Phaser.Game(500, 500);
```

Inside `playGame` object, we will need another variable to visually display the score on the stage. How about another text?

That's why I am going to create a `playGame` property called `scoreText`, which is declared as `null`.

> In JavaScript `null` means "nothing", or an empty value.

Having a `null` variable does not mean we can't assign it a value when we need.

```
playGame.prototype = {
     scoreText: null,
     soundArray: [],
     preload: function(){
          ...
     }
     ...
}
```

Once `create` method is executed, it means a new game is started so we set `score` to zero.

Also, you should be able to see what the remaining lines do:

```
create: function(){
     score = 0;
     this.placeTiles();
     if(playSound){
          this.soundArray[0] = game.add.audio("select", 1);
          this.soundArray[1] = game.add.audio("right", 1);
          this.soundArray[2] = game.add.audio("wrong", 1);
     }
     var style = {
          font: "32px Monospace",
          fill: "#00ff00",
          align: "center"
     }
     this.scoreText = game.add.text(5, 5, "Score: " + score, style);
}
```

We are adding a new text using `scoreText` property we defined before.

You should already know how to add and style to the text.

First, we defined a text style, then we created the text itself applying such style.

Also notice how we can write anything we want inside the text content. When we created the title screen we only wrote a string “Crack Alien Code” but now we are also writing the content of `score` variable.

Moreover, text strings are not static but we can change their content on the fly

> `text` property of a text will change the string displayed.

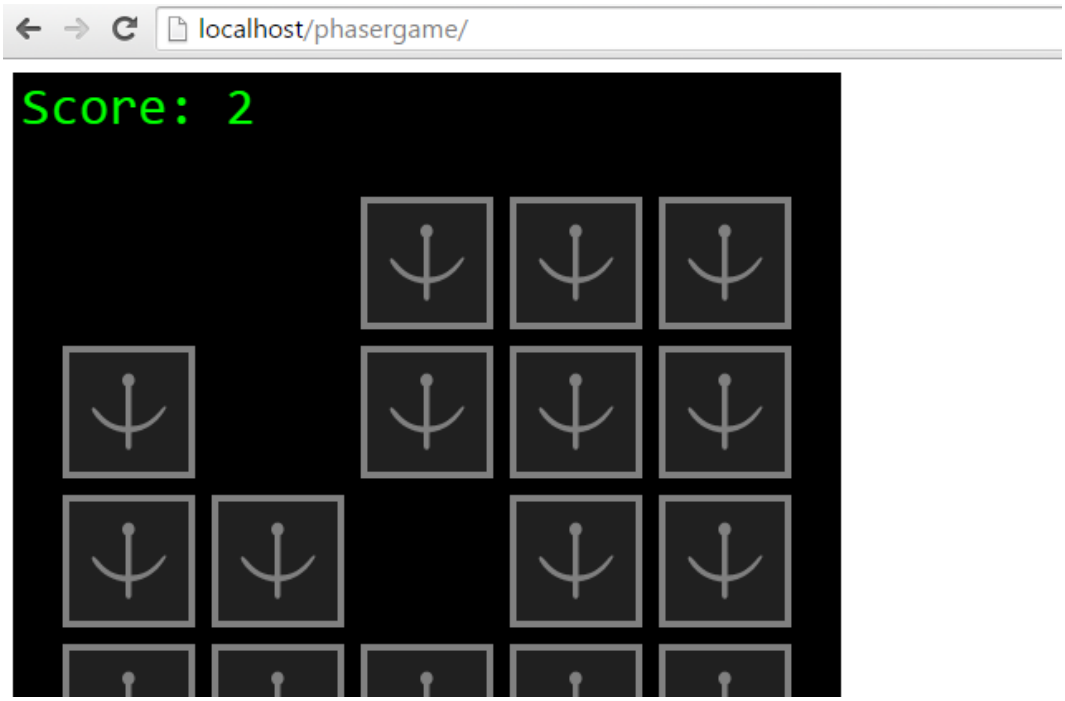
Do you want to show the score in real time?

Look at `checkTiles` function:

```
checkTiles: function(){
     if(selectedArray[0].value == selectedArray[1].value){
          if(playSound){
               this.soundArray[1].play();
          }
          score ++;
          this.scoreText.text = "Score: " + score;
          selectedArray[0].destroy();
          selectedArray[1].destroy();
     }
     else{
          if(playSound){
               this.soundArray[2].play();
          }
          selectedArray[0].frame = 10;
          selectedArray[1].frame = 10;
     }
     selectedArray.length = 0;
}
```

Once the player makes a successful match, `score` is incremented and then we can update a text simply by setting its `text` property.

Run the game and see how your score increments at each successful match.



Showing the score on the screen does not complete the game. There is more that

we can do.

Increasing difficulty by adding a timer

What if you had a time limit to complete the game? Let's increase the difficulty by giving you only a minute to solve the game.

`timeLeft` variable will keep track of the remaining time we have.

```
var tileSize =  80;
var numRows = 4;
var numCols = 5;
var tileSpacing = 10;
var tilesArray = [];
var selectedArray = [];
var playSound;
var score;
var timeLeft;
var game = new Phaser.Game(500, 500);
```

Now we have to prevent the player from cheating. If you remove the focus from the page, the game will pause. This means timer won't decrease, and this is cheating.

We are going to prevent this by setting `stage.disableVisibilityChange` property to `true`. Let's add it in the title screen.

```
create: function(){
     game.stage.disableVisibilityChange = true;
     var style = {
          font: "48px Monospace",
          fill: "#00ff00",
          align: "center"
     };
     var soundButton = game.add.button(game.width / 2 - 100 , game.height / 2 +
       100, "soundicons", this.startGame, this);
     soundButton.anchor.set(0.5);
     soundButton = game.add.button(game.width / 2 + 100 , game.height / 2 + 100,
       "soundicons", this.startGame, this);
     soundButton.frame = 1;
     soundButton.anchor.set(0.5);
     var text = game.add.text(game.width / 2, game.height / 2 - 100, "Crack Alien
       Code", style);
     text.anchor.set(0.5);
}
```

Now in the same way we added `scoreText` variable to have the score text

displayed, we add a new variable to show the remaining time.

```
playGame.prototype = {
     scoreText: null,
     timeText: null,
     soundArray: [],
        ...
}
```

And when we initialize the game, we set the time limit to 60, which means you'll have to complete the game in a minute.

There's no need to explain you how to create the text which will show the remaining time:

```
create: function(){
     score = 0;
     timeLeft = 60;
     this.placeTiles();
     if(playSound){
          this.soundArray[0] = game.add.audio("select", 1);
          this.soundArray[1] = game.add.audio("right", 1);
          this.soundArray[2] = game.add.audio("wrong", 1);
     }
     var style = {
          font: "32px Monospace",
          fill: "#00ff00",
          align: "center"
     }
     this.scoreText = game.add.text(5, 5, "Score: " + score, style);
     this.timeText = game.add.text(5, game.height - 5, "Time left: " + timeLeft,
        style);
     this.timeText.anchor.set(0, 1);
     game.time.events.loop(Phaser.Timer.SECOND, this.decreaseTime, this);
}
```

Anyway, there's a couple of lines I want to show you in detail:

```
this.timeText.anchor.set(0, 1);
```

This time `anchor.set(0, 1)` means the anchor point is on the bottom left angle.

The second line I want you to see is the one handling the timer.

```
game.time.events.loop(Phaser.Timer.SECOND, this.decreaseTime, this);
```

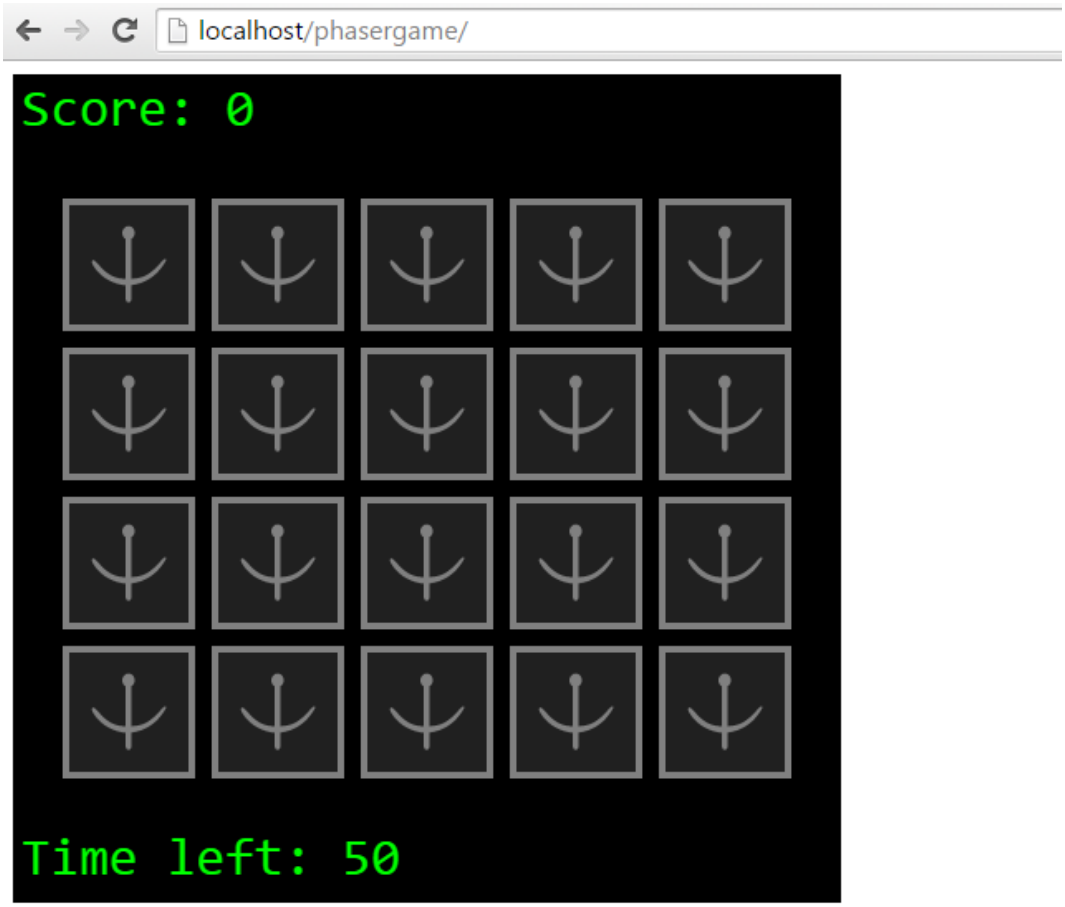
This time we aren't dealing with a timer which only runs once, but with a loop.

> `time.events.loop(delay, callback, callbackContext)` adds a looped event that will repeat forever or until it's stopped. It will call `callback` function in the `callbackContext` context.

Let's have a look at the callback function, called `decreaseTime`:

```
decreaseTime: function(){
     timeLeft --;
     this.timeText.text = "Time left: " + timeLeft;
}
```

It decreases the timer, then updates the text showing the time left. Run the game:



Having a timer is nice, but quite useless if once the time is over nothing happens.

Showing Game Over screen when running out of time

When the player runs out of time, we need to stop the game and show a game over screen. How can we make a game over screen?

Right, with another state.

```
game.state.add("TitleScreen", titleScreen);
game.state.add("PlayGame", playGame);
game.state.add("GameOver", gameOver);
game.state.start("TitleScreen");
```

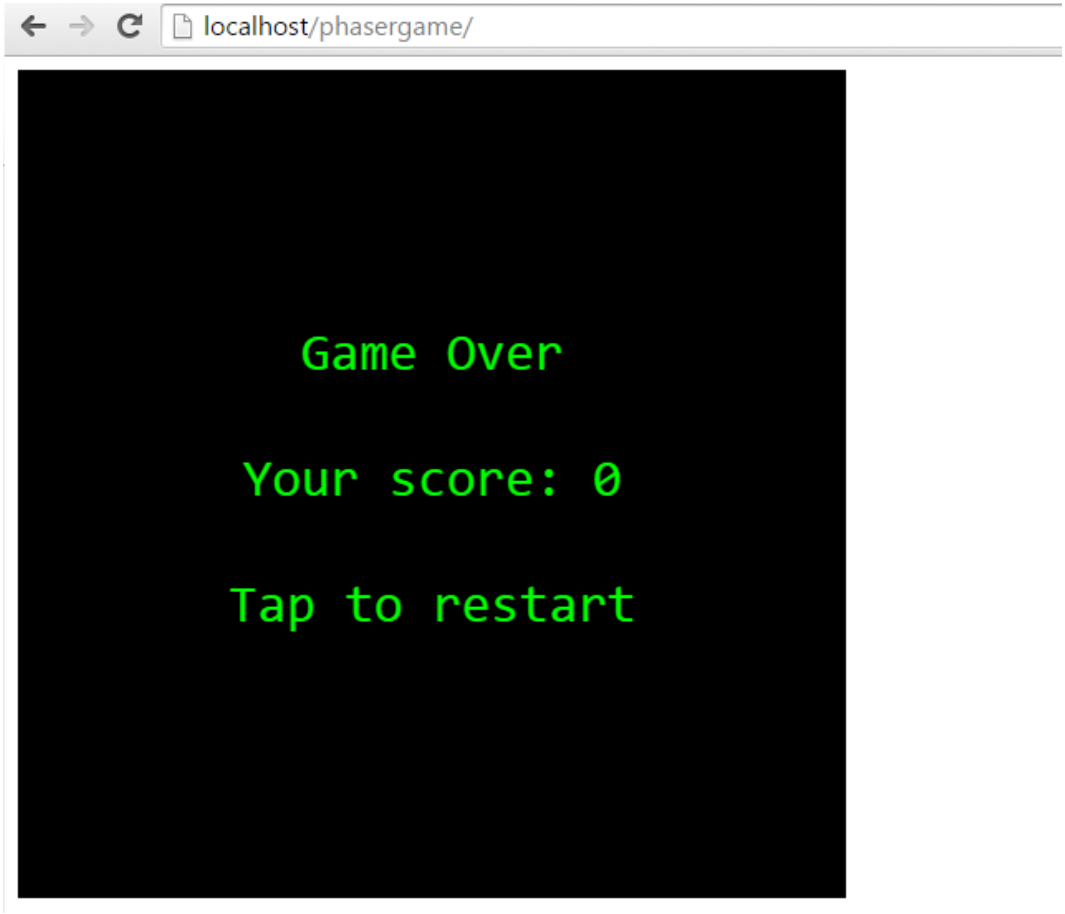
Once the state has been added to the game, we will call it once the timer reaches zero. Look at `decreaseTime` method:

```
decreaseTime: function(){
     timeLeft --;
     this.timeText.text = "Time left: " + timeLeft;
     if(timeLeft == 0){
          game.state.start("GameOver");
     }
}
```

And now it's just a matter of building the new state, with the “Game Over” text, the score and the message to tap to restart. Nothing new in the following code.

```
var gameOver = function(game){}
gameOver.prototype = {
     create: function(){
          var style = {
               font: "32px Monospace",
               fill: "#00ff00",
               align: "center"
          }
          var text = game.add.text(game.width / 2, game.height / 2, "Game
        Over\n\nYour score: " + score + "\n\nTap to restart", style);
          text.anchor.set(0.5);
     }
}
```

Play the game, and after a minute you should see something like this:



Ok, now tap to restart. You can't. We have to write another couple of lines.

Restarting the game

Until now, we learned how to detect the click/touch on a button, but this time we don't have buttons on the stage.

We will need to trigger a generic click/touch.

In the same way Phaser handles generic input on buttons, it also handles generic input events.

Look at these new changes:

```
gameOver.prototype = {
     create: function(){
          var style = {
               font: "32px Monospace",
               fill: "#00ff00",
               align: "center"
          }
          var text = game.add.text(game.width / 2, game.height / 2, "Game
        Over\n\nYour score: " + score + "\n\nTap to restart", style);
          text.anchor.set(0.5);
          game.input.onDown.add(this.restartGame, this);
     },
     restartGame: function(){
          tilesArray.length = 0;
          selectedArray.length = 0;
          game.state.start("TitleScreen");
     }
}
```

Apart from `restartGame` function which simply restarts the game by calling `TitleScreen` state after clearing out the arrays, I want you to focus on this line:

```
game.input.onDown.add(this.restartGame, this);
```

Wherever you click or touch the screen, `restartGame` method will be called.

> `input.onDown(callback, callbackContext)` is fired each time a pointer is pressed down. It runs `callback` function in `callbackContext` context)

And finally the game is complete. Anyway, what happens if you removed all tiles before time runs out? You will find yourself looking at a black screen while the timer ticks away without you being able to do anything.

That's why we are going to add another feature. The last one, in this game.

Giving the game a twist

We have to give the player something to do when all tiles have been removed and there's still time to play.

Listen to this idea: if you removed all tiles before the time runs out, another set of tiles is placed on the screen, so you can increase your score by making more matches.

To reward you for successful matches, you will be awarded with two extra seconds at each match.

Does it sound complicated?

Didn't you realize yet there's nothing complicated when you use Phaser?

Let's create a new variable which will inform us how many tiles are still placed on the board.

We'll call it `tilesLeft`.

```
var tileSize =  80;
var numRows = 4;
var numCols = 5;
var tileSpacing = 10;
var tilesArray = [];
var selectedArray = [];
var playSound;
var score;
var timeLeft;
var tilesLeft;
var game = new Phaser.Game(500, 500);
```

When the tiles are shown on the screen, `tilesLeft` is set to the entire amount of tiles in the game, that is `numRows * numCols`.

```
placeTiles: function(){
     tilesLeft = numRows * numCols;
     ...
}
```

The remaining code is all to be placed in `checkTiles` method.

When the player makes a successful match, we increase `timeLeft` by 2, actually adding two seconds to the game, and update `timeText` text.

At the same time, we decrease the amount of tiles left by two.

Once there aren't any tiles left on the table, we clear the arrays and place another set of tiles calling `placeTiles` function one more time. This will allow the player to continue to play.

```
checkTiles: function(){
     if(selectedArray[0].value == selectedArray[1].value){
          if(playSound){
               this.soundArray[1].play();
          }
          score ++;
          timeLeft +=2;
          this.timeText.text = "Time left: " + timeLeft;
          this.scoreText.text = "Score: " + score;
          selectedArray[0].destroy();
          selectedArray[1].destroy();
          tilesLeft -= 2;
          if(tilesLeft == 0){
               tilesArray.length = 0;
               selectedArray.length = 0;
               this.placeTiles();
          }
     }
     else{
          if(playSound){
               this.soundArray[2].play();
          }
          selectedArray[0].frame = 10;
          selectedArray[1].frame = 10;
     }
     selectedArray.length = 0;
}
```

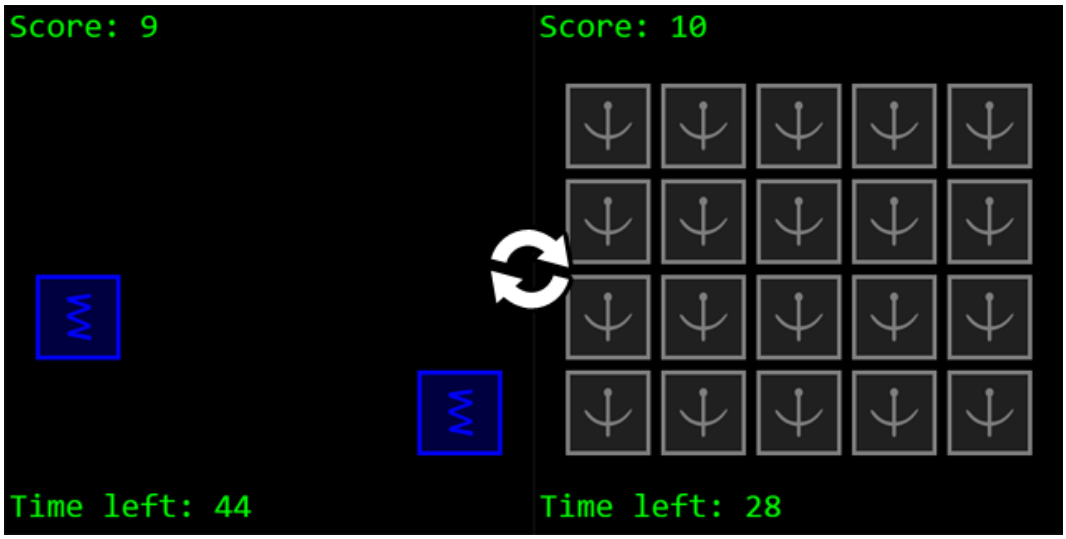
There's nothing new in this piece of code, only concepts that you already learned throughout this book.

I would only commented these lines which will make the board refill if the player removed all tiles:

```
if(tilesLeft == 0){
       tilesArray.length = 0;
       selectedArray.length = 0;
       this.placeTiles();
}
```

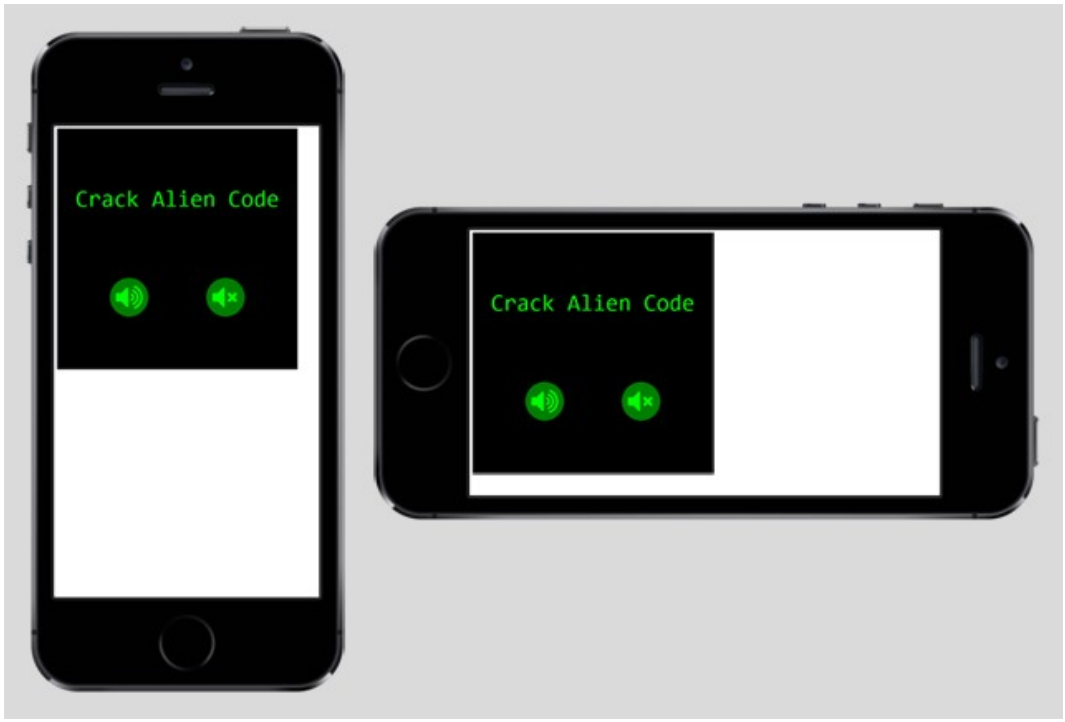
We check for `tilesLeft` to be zero, and in this case we empty `tilesArray` array and `selectedArray` array, then call `placeTiles` method to place all tiles again on the board.

Run the game and try to clear the board before time runs out, you will see that the game restarts. Will you be able to clear the board twice?



By the way, did you try to play it on a mobile device?

Here's how the game looks like on my iPhone 5:



Actually not the best way to play a game. There's nothing “cross-platform” in it.

Making it run nicely on any mobile device, no matter what the orientation is

We'll make the game look good on any device in two steps: first, we have to scale up the game to cover the largest area possible, then we'll make the HTML page which hosts the game more mobile friendly.

About scaling up the game, modify `create` method in `TitleScreen` state:

```
create: function(){
     game.scale.pageAlignHorizontally = true;
     game.scale.pageAlignVertically = true;
     game.scale.scaleMode = Phaser.ScaleManager.SHOW_ALL;
     game.stage.disableVisibilityChange = true;
     var style = {
          font: "48px Monospace",
          fill: "#00ff00",
          align: "center"
     };
     ...
}
```

Let's examine the new lines one by one:

```
game.scale.pageAlignHorizontally = true;
```

Setting `pageAlignHorizontally` to `true` will horizontally align the game in the Parent container or window.

```
game.scale.pageAlignVertically = true;
```

Same thing, for vertical aligment

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

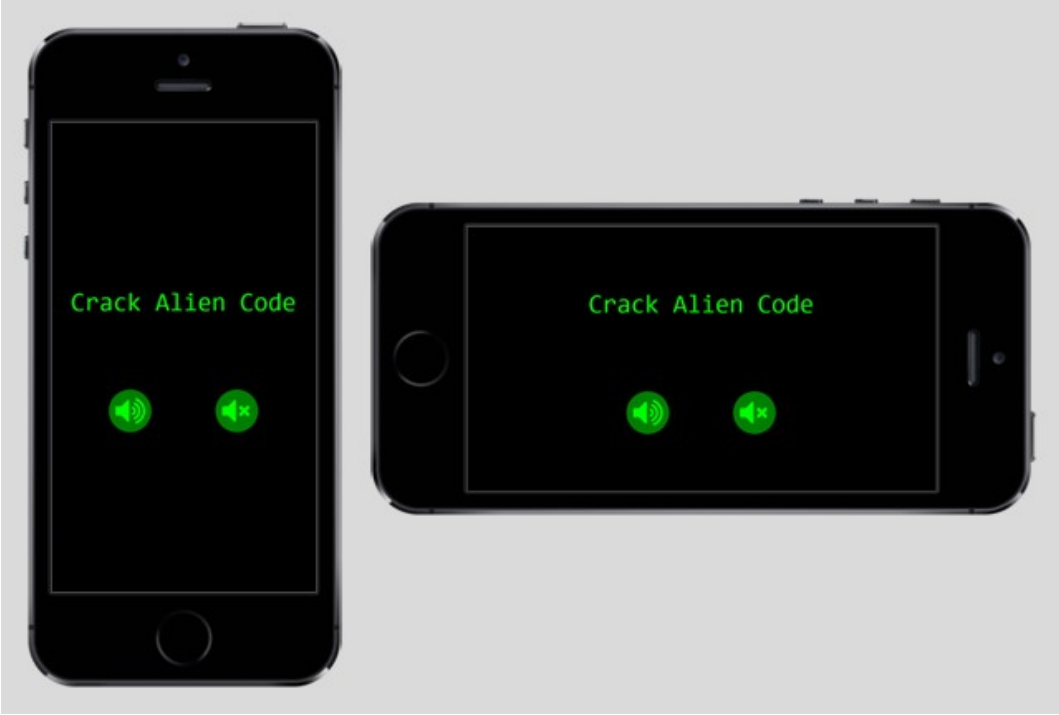
`scaleMode` sets the scaling method which in this case with `SHOW_ALL` we show the game at the largest scale possible while keeping the original aspect ratio.

The last thing to do is editing `index.html` file to add some meta tags which have

been specifically created for mobile devices. The new header content is:

```
<head>
     <script type="text/javascript" src="phaser.js"></script>
     <script type="text/javascript" src="game.js"></script>
     <meta name="viewport" content="width=device-width, initial-scale=1.0,
        maximum-scale=1.0, minimum-scale=1.0, user-scalable=no, minimal-ui" />
     <meta name="apple-mobile-web-app-capable" content="yes" />
     <meta name="apple-mobile-web-app-status-bar-style" content="black" />
     <meta name="HandheldFriendly" content="true" />
     <meta name="mobile-web-app-capable" content="yes" />
     <style type="text/css">
          body{
               padding:0px;
               margin:0px;
               background: #000;
          }
     </style>
</head>
```

There's no need to explain these lines as they are standard tags suggested by device manufacturers.



And now you finally completed the project.

Saving high 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 natively 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 two new variables:

```
var tileSize =  80;
var numRows = 4;
var numCols = 5;
var tileSpacing = 10;
var localStorageName = "crackalien";
var highScore;
var tilesArray = [];
var selectedArray = [];
var playSound;
var score;
var timeLeft;
var tilesLeft;
var game = new Phaser.Game(500, 500);
```

`localStorageName` variable stores the name of the local storage variable, so each time you will change crackalien with something else, you will reset your best score.

`highScore` will contain the actual high score number.

When we launch the game, before entering the title screen we will check the local storage to see if we already saved a high score, so we are going to add a line before starting `TitleScreen` state.

This line will introduce a lot of new concepts:

```
game.state.add("TitleScreen", titleScreen);
game.state.add("PlayGame", playGame);
game.state.add("GameOver", gameOver);
highScore = localStorage.getItem(localStorageName) == null ? 0 :
        localStorage.getItem(localStorageName);
game.state.start("TitleScreen");
```

Does it look strange?

It's a conditional operator, also called ternary operator because it requires three operands.

> 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`.

Writing the conditional operator as an `if` statement, it will look like:

```
if(localStorage.getItem(localStorageName) == null){
     highScore = 0;
}
else{
     highScore = localStorage.getItem(localStorageName);
}
```

`getItem` method of `localStorage` retrieves `localStorageName` data. If it's `null`, it means we never saved a high score.

> `localStorage.getItem(keyName)` returns `keyName`'s value.

Probably it's the first time we launch the game in this browser, or we never played until game over screen – where we will save the score – or we just changed `localStorageName` name.

In this case, `highScore` is set to zero.

If there is a value in local storage data, this means we previously saved a high score so we set `highScore` to this value.

And this is how we retrieve previously saved high score, if any.

To save it, we'll need to add a couple of lines to `create` method in `GameOver` state:

```
create: function(){
     highScore = Math.max(score, highScore);
     localStorage.setItem(localStorageName, highScore);
     var style = {
          font: "32px Monospace",
          fill: "#00ff00",
          align: "center"
     }
     var text = game.add.text(game.width / 2, game.height / 2, "Game Over\n\nYour
       score: " + score + "\nBest score: " + highScore + "\n\nTap to restart",
       style);
     text.anchor.set(0.5);
     game.input.onDown.add(this.restartGame, this);
}
```

Now launch the game and play a couple of times, then close the browser window or even restart your computer/device

Your high score will always remain saved:

Let's see how that was possible.

The first thing to do is to see if the current score is higher than `highScore` value and in that case update `highScore` variable setting it to `score`.

Rather than using an `if` statement, to see something new we'll always update `highScore` value setting it to the highest number between `score` and `highScore`.

> `Math.max(v1, v2, v3, … , vn)` method returns the highest number among its arguments.

Once `highScore` has been updated, it's time to save it to the local storage.

> `localStorage.setItem(keyName, keyValue)` method adds `keyName` to the storage, or updates it to `keyValue` if it already exists.

And finally we can output the score as well as the high score.

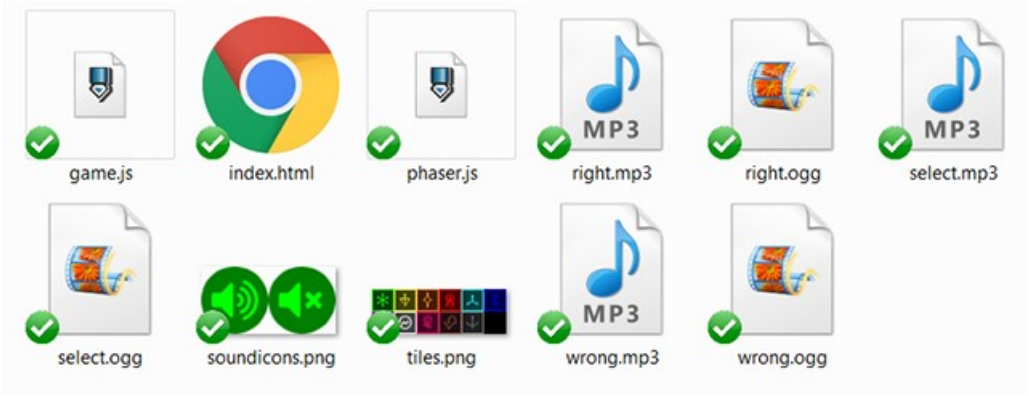
Now your game will remember your best score.

As a final advice, I don't recommend using simple names like “bestScore” or “currentLevel” for your local storage variables as other applications could use the same names, and you will overwrite other settings with your data, or get your data overwritten by other applications data.

Use long names, with your game name as prefix, such as `CrakAlienCodeScore`, so when you'll make another game called, let's say, Endless Jumper, you will save your score in a variable called `EndlessJumperScore`, and you won't have problems.

Organizing your folders

You should be proud of your first game, but having a look at your project folder it looks a bit unorganized.



We have images, sounds and scripts all placed in the same folder.

This is not a big problem when you are dealing with simple games like this one, but imagine a bigger project with a lot of images and sounds: you will end with a folder full of clutter.

That's why you should create a folder called `assets`, with more folders in it called `sprites` and `sounds` where to place all your files. You can give your folders the names you want, but keep in mind there are the names I will be using in future books and tutorials. Now your project will look like this:

Now it's more organized, and we just have to edit the preloading paths.

This is how we change `preload` method in `playGame` object:

```
preload: function(){
     game.load.spritesheet("tiles", "assets/sprites/tiles.png", tileSize,
tileSize);
     game.load.audio("select", ["assets/sounds/select.mp3",
"assets/sounds/select.ogg"]);
     game.load.audio("right", ["assets/sounds/right.mp3",
"assets/sounds/right.ogg"]);
     game.load.audio("wrong", ["assets/sounds/wrong.mp3",
"assets/sounds/wrong.ogg"]);
}
```

And the same concept applies to `preload` method in `titleScreen`:

```
preload: function(){
     game.load.spritesheet("soundicons", "assets/sprites/soundicons.png", 80, 80);
}
```

And now we have the same working game, with more organization.

Creating a preloader state

There's another small optimization to do in order to keep your code organized, especially if you are working on a small game which does not require that much images and sounds.

At the moment we preload assets in a way we can call “on demand”.

When in title screen we needed the sound buttons, we preloaded their images, then when in game state we needed the sounds and the tiles, we preloaded them.

It would be better if we could preload all assets before the game begins, so all code and references to images and sounds can be found in a single place.

That's why are going to create a new state called `PreloadAssets` which will handle all preloading process, freeing other states from having a `preload` method.

Obviously this newly created state will be the first the game will execute, to ensure all assets have been preloaded before we launch the game itself with `TitleScreen` state.

```
game.state.add("PreloadAssets", preloadAssets);
game.state.add("TitleScreen", titleScreen);
game.state.add("PlayGame", playGame);
game.state.add("GameOver", gameOver);
highScore = localStorage.getItem(localStorageName) == null ? 0 :
        localStorage.getItem(localStorageName);
game.state.start("PreloadAssets");
```

The code of `PreloadAssets` is just a cut/paste of the code of the `preload` methods in `PlayGame` and `TitleScreen` states:

```
var preloadAssets = function(game){}
preloadAssets.prototype = {
     preload: function(){
          game.load.spritesheet("tiles", "assets/sprites/tiles.png", tileSize,
tileSize);
          game.load.audio("select", ["assets/sounds/select.mp3",
"assets/sounds/select.ogg"]);
          game.load.audio("right", ["assets/sounds/right.mp3",
"assets/sounds/right.ogg"]);
          game.load.audio("wrong", ["assets/sounds/wrong.mp3",
"assets/sounds/wrong.ogg"]);
          game.load.spritesheet("soundicons", "assets/sprites/soundicons.png", 80,
80);
     },
     create: function(){
          game.state.start("TitleScreen");
     }
}
```

As you can see I only copied the content of the old `preload` methods – which you will have to remove now – and in create method, once all assets have been loaded, I launch `TitleScreen` state, starting the game.

Where to go now

When you make a game following a tutorial or a book, I always suggest to make it twice: the first time following the tutorial and the second time on your own.

Take a deep breath, delete everything and create the game from scratch.

Then, add some basic features, like keeping track of the highest score, or the number of unsuccessful matches.

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 of any errors 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.

Another special “thank you” goes to **Mario** (just “Mario”, do not know his surname) for hunting for typos and errors.

I hope you enjoyed reading this book as much as I enjoyed writing it.

Emanuele.