Create HTML5 Vertical Endless Runner

cross platform games

using Phaser framework
and only FREE software

Emanuele Feronato

Create HTML5 Vertical Endless Runner cross platform games

All the secrets behind the making of cross-platform HTML5 vertical endless runner games using Phaser framework and other free software.

Written by Emanuele Feronato

About endless runners and the game we are building

"Endless runner" or "infinite runner" game genre is becoming more and more popular as endless runners are generally easy to play, quite challenging and can be developed by small studios or solo indie developers.

In these kind of games the player character is continuously moving forward through a randomly generated endless game world.

Game controls are limited to the bare minimum, and most of times the player is only allowed to make only one action, such as jumping or changing direction.

The only object of these games is to get as far as possible before the character dies, trying to get a high score.

Endless runners have found particular success on mobile platforms since mobile devices are perfectly suited to the small set of controls these games require, often limited to a single screen tap or swipe, allowing to be played with only one hand.

The game we are about to create is heavily based on **Rise Above** by Norman Rozental (https://itunes.apple.com/au/app/rise-above/id1039989390?mt=8) which has been featured in the Apple app store itself.

It's a free game so you can download it, play it and have an idea about what you

are about do develop.

Through this book you will learn how to make a complete HTML5 cross-platform game like Rise Above, with all its original features, and even improve it and add room for customization.

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

A cross-platform game 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, providing the player with the same experience on different platforms.

From a programmer point of view, a cross-platform game is a game you code only once, and will adapt to any device no matter the resolution or aspect ratio.

I often think about cross-platform games as “one script to rule them all”, and that's exactly what I mean, you will focus on game development and let the framework do the dirty job to adapt it to various devices.

There are a lot of frameworks which allow you to create cross-platform games, and my suggested choice is **Phaser**.

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 tutorials around the web to get you started.

Should this be your first project, don't worry anyway: following this book you will get every information you need to create the game and get started into game development.

Together with Phaser, you are going to use other free software turning your computer into a game development workstation at no cost.

Choosing a text editor

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.

Other alternatives are **Brackets** (http://brackets.io/) and **Atom** (https://atom.io/) but you can use your favorite text editor, I'd only suggest you to choose one capable of highlighting 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).

If you are looking for something really simple, with no extra stuff, check **Fenix Web Server** (http://fenixwebserver.com/). All these web servers are free.

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.

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 actions 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 period.

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.

Setting up the project

The whole project is basically a web page including Phaser framework and another JavaScript file with our game.

There are two important things you should consider before you start coding your game: first, size matters. When importing third party frameworks like Phaser, always choose minified versions if provided.

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

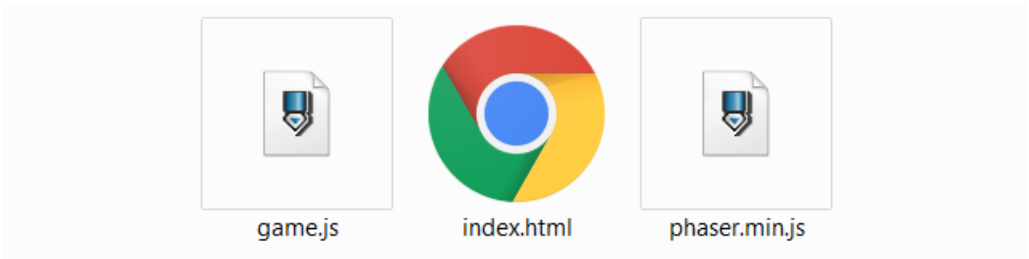
Second, writing the entire code of a game in a single file is generally considered a malpractice. You should create one JavaScript file for the splash screen, another for game logic, and basically one file for each actor you will include in your game.

The problem is some sponsors need to have the entire game in one file. And since finding sponsors and selling game licenses can be a great income source, we have to code games with sponsors needs in mind.

Our entire game will be written in a file called `game.js`.

Create an `index.html` file which is the web page you will call to launch the game, and you'll have all you need to start coding the game.

This is how your project folder will look like:



Icons may be different according to your file preferences.

Let's start with `index.html`:

```
<!DOCTYPE html>
<html>
     <head>
          <style type="text/css">
               body{
                    background: #000000;
                    padding:0px;
                    margin:0px;
               }
          </style>
          <script src="phaser.min.js"></script>
          <script 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.min.js` is the file we just downloaded, and `game.js` will contain our game script. There are very few lines in `game.js` too, at the moment:

```
var game;

window.onload = function() {
       game = new Phaser.Game(640, 960, Phaser.AUTO, "");
}
```

We just created a `game` variable and once the window loads, we create the instance of the game itself.

`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.

```
game = new Phaser.Game(640, 960, Phaser.AUTO, "");
```

`game` object is the heart of your game, providing quick access to common functions and handling the boot process.

> `new Game(width, height, renderer, parent)` creates a game `width` pixels wide, `height` pixels tall rendering it in `render` mode – which can be `Phaser.AUTO`, `Phaser.WEBGL`, `Phaser.CANVAS` or `Phaser.HEADLESS` – into `parent` DOM element.

Now let me explain why I used these values as arguments.

The common portrait resolution in smartphones is 640x960 pixels. That's why we are using it. Obviously there are a lot of exceptions, but since we are building a cross-platform game we start with 640x960 and we will only need to make some minor tweaks to make it work on all devices.

About the renderer, using `Phaser.AUTO` we let Phaser decide which renderer to use, and the empty string passed as DOM element will make Phaser inject the game directly in the body of the page.

The latest two arguments – `renderer` and `parent` – have been set to their default values, so you could have created the game this way:

```
game = new Phaser.Game(640, 960);
```

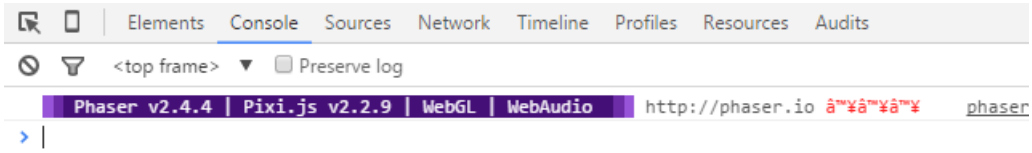
But it's always good to know what you are doing.

We said we will be using our web browser to test the game, so ensure you have

your web server up and running and let's start rocking.

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/yourgamefolder/ and this is what you should get if running it on your Google Chrome browser:



This is the default debug string Phaser prompts on the console window. You can generally access to your console window pressing F12 in your browser, anyway refer to your browser documentation.

Text output may vary a bit according to Phaser version, this book has been updated to **Phaser 2.4.7**.

Adding game 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.

Let's think about the game we are making. We still do not know to code it but we can easily imagine the game will 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”.

Now let's write this concept in a more detailed way, listing all the states we will actually use in our game:

Boot state: in the boot state we will make all adjustment to the game to be resized

accordingly to browser resolution and aspect ratio.

Preload state: we will use this state to preload all assets we will use in the game. It's the classic “loading” screen you see in most games.

Title Screen state: the title screen, showing your game name and a play button.

Game state: well, the game itself.

Game over state: the most hated screen, the one you won't want to see. The game over screen also features a “play again” button to let players restart the game.

This is the blueprint of the game, with all states defined and declared. Change `game.js` this way:

```
var game;

window.onload = function() {
     game = new Phaser.Game(640, 960, Phaser.AUTO, "");
     game.state.add("Boot", boot);
     game.state.add("Preload", preload);
     game.state.add("TitleScreen", titleScreen);
     game.state.add("PlayGame", playGame);
     game.state.add("GameOverScreen", gameOverScreen);
     game.state.start("Boot");
}

var boot = function(game){};
boot.prototype = {
     create: function(){
          console.log("game started");
     }
}

var preload = function(game){};
preload.prototype = {
}

var titleScreen = function(game){};
titleScreen.prototype = {
}

var playGame = function(game){};
playGame.prototype = {
}

var gameOverScreen = function(game){};
gameOverScreen.prototype = {
}
```

Inside `onload` method we add all game states, then call `Boot` state which is the

first state we want to be executed. Let's break the code in pieces:

```
game.state.add("Boot", boot);
```

Here is how we add a state to our game. The first parameter 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 `boot` function to a state called `Boot`. All other states are added following the same concept.

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

And finally this is how a state is started.

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

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

What's inside `boot` function?

```
var boot = function(game){};
boot.prototype = {
     create: function(){
          console.log("game started");
     }
}
```

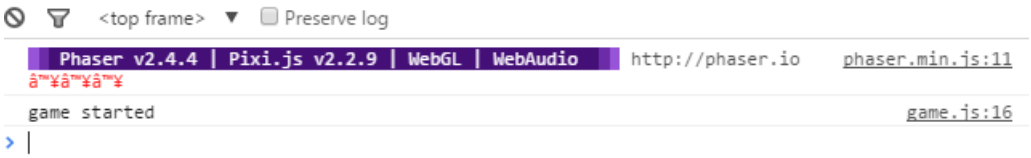
We called `boot` a “function” for the sake of simplicity, but actually it's an object with a prototype.

> 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.

Phaser will recognize if inside a state object there is a method called `create` and will execute it once the state has been launched. So everything inside `create` function will be executed when `Boot` state is 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.

Run the game and you will see a debug message in the console.



This is the string we wrote in `create` method of `boot` object called when we launched `Boot` state.

It's easier than you may think: you launch a state, and `create` method inside the state object is executed.

Now it's just a matter of coding all the states, one by one. Which actually is the content of the whole book.

Creation of the boot state

The boot state – as the name suggests – boots the game, and mainly scales the game adapting it to various resolutions. We need to make some changes to both boot and preload states, here is boot state:

```
var boot = function(game){};
boot.prototype = {
     preload: function(){
          this.game.load.image("loading","assets/sprites/loading.png");
     },
     create: function(){
          game.scale.pageAlignHorizontally = true;
          game.scale.pageAlignVertically = true;
          game.scale.scaleMode = Phaser.ScaleManager.SHOW_ALL;
          this.game.state.start("Preload");
     }
}
```

And this is the preload state:

```
var preload = function(game){};
preload.prototype = {
     create: function(){
          console.log("going to preload");
     }
}
```

The first thing you should notice is the `preload` method inside `boot` object.

Earlier we said `create` method is executed when the state launches, but before doing it Phaser also checks for a `preload` method. If `preload` method is found, it will be executed before the state loads, most of times loading graphic and sound assets, finally launching `create` method after all assets specified in `preload` method have been actually loaded.

Long concept made short: first Phaser executes `preload` method, then executes `create` method. Let's see the content of `preload` method in `boot` object:

```
this.game.load.image("loading","assets/sprites/loading.png");
```

This is how we say Phaser to load an image to be used later in the game.

> `load.image(key, path)` loads an image stored into `path` and assigns it `key` name.

I would like you to have a look at the path where I stored the image. I created an `assets` folder with a `sprites` folder inside.

That's where I saved the image. It's not mandatory to create folders and sub folders for your assets, but it's recommended to keep your files organized.

Also, this is the way I will use throughout the book.

Now, what's that `loading.png` file?

It's just a 320x20 white rectangle saved as a PNG image which will be used as a loading bar.

> 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 there are a few of new lines in `create` method to talk about:

```
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 alignment

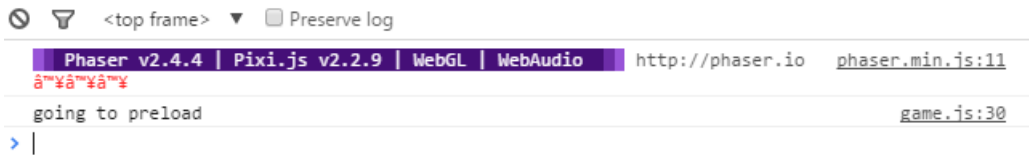
```
game.scale.scaleMode = Phaser.ScaleManager.SHOW_ALL;
```

`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.

```
this.game.state.start("Preload");
```

Now, `Preload` state is launched.

Run the game and you will see the `Preload` debug message in the console.



Now the game is booted and we are ready preload assets.

Creation of the preload state

The preload state is very important because it allows to preload images and other assets which will be available later in the game.

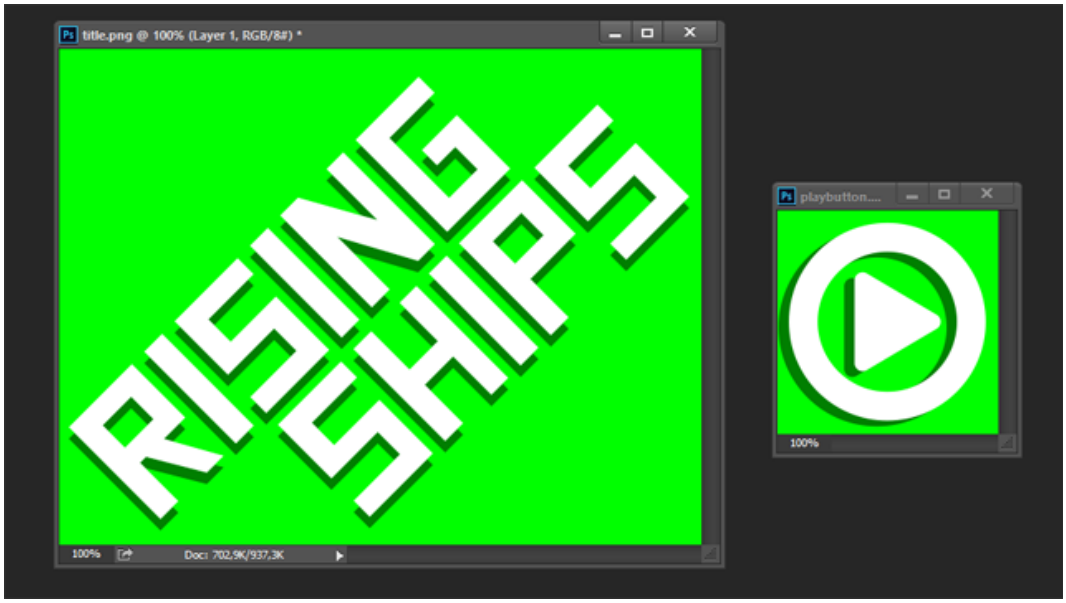
Also, we will make use of the loading bar image preloaded earlier in boot state.

`preload` object works like any other state object: Phaser first looks for a `preload` method then once everything has been loaded executes `create` method.

```
var preload = function(game){};
preload.prototype = {
     preload: function(){
          var loadingBar = this.add.sprite(game.width / 2, game.height / 2,
        "loading");
          loadingBar.anchor.setTo(0.5);
          game.load.setPreloadSprite(loadingBar);
          game.load.image("title", "assets/sprites/title.png");
          game.load.image("playbutton", "assets/sprites/playbutton.png");
     },
     create: function(){
          this.game.state.start("TitleScreen");
     }
}

var titleScreen = function(game){};
titleScreen.prototype = {
     create: function(){
          console.log("title screen here");
     }
}
```

As you can see, I loaded two more images: the game title and the play button, look at them:



I placed them on a green background so you can see them, but both images have transparent background.

Now I am showing you two lines which add a sprite on the canvas, but they will be explained in detail in a few minutes:

```
var loadingBar = this.add.sprite(game.width / 2, game.height / 2, "loading");
loadingBar.anchor.setTo(0.5);
```

The reason why I am not explaining these lines now – take them as "add `loadingBar` image" – is because now it's time to show you an important feature you will only find in this state, while you will have a lot of time to learn how to add stuff on the screen.

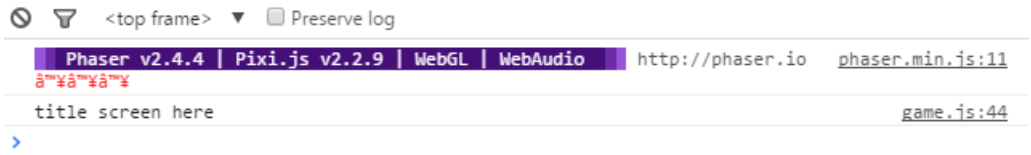
This is the core of `preload` method:

```
game.load.setPreloadSprite(loadingBar);
```

With `setPreloadSprite` method we can turn an image into a progressive loading bar which grows as assets are being loaded. Believe me, it would take quite some time to add this feature coding it from scratch.

> `load.setPreloadSprite(sprite)` sets `sprite` to be a preload sprite which has its width or height crop adjusted based on the percentage of the loader in real-time. This allows you to easily make loading bars for games.

Run the game, and you should see this debug message:



That's because you preloaded all graphic assets then switched to `TitleScreen` state.

Probably you did not see the loading bar growing as image were loading. That's

because at the moment the game loads just a couple of small images and you probably are testing the game on localhost.

Try to upload the game on some FTP space and launch it with a slow connection, and you will see the loading bar. Anyway, as we go completing the game adding more and more stuff, we will need to preload more and more files and sooner or later you will be able to see the loading bar in action.

We loaded some images, it's time to do something with them.

Creation of title screen

We want the title screen to have game title displayed – obviously – and a “play” button. Also, to give the game a more modern feeling, the background color should change at each play.

While just choosing a random color would be easy, you understand not all colors are suitable as background colors. We want to have only a small selection of background colors and randomly choose among them.

To choose nice background colors, if you don't have ideas you can inspire yourself by googling something like “color schemes”, you'll get a lot of ideas, satisfaction guaranteed.

All background colors selected will be stored into `bgColors` array which will be declared as a global variable. Declaring all customizable variables as global variables is a good practice from a game developer point of view because it will allow sponsors to easily edit the most important parameters if they need to tune the gameplay.

So, here is `bgColors` array with its 10 colors to be randomly chosen:

```
var game;
var bgColors = [0xF16745, 0xFFC65D, 0x7BC8A4, 0x4CC3D9, 0x93648D, 0x7c786a,
0x588c73, 0x8c4646, 0x2a5b84, 0x73503c];
```

Now we have to choose one color, then place the title and the play button we

preloaded before.

Change `titleScreen` object this way:

```
var titleScreen = function(game){};
titleScreen.prototype = {
     create: function(){
          game.stage.backgroundColor = bgColors[game.rnd.between(0,
        bgColors.length - 1)];
          var title = game.add.image(game.width / 2, 210, "title");
          title.anchor.set(0.5);
          var playButton = game.add.button(game.width / 2, game.height - 150,
        "playbutton", this.startGame);
          playButton.anchor.set(0.5);
     },
     startGame: function(){
          game.state.start("PlayGame");
     }
}

var playGame = function(game){};
playGame.prototype = {
     create: function(){
          console.log("play the game");
     }
}
```

You'll also need to add `create` method to `playGame` object to prompt some debug text on the console and check everything is working.

Before we test the game, there's a lot to say about `create` method of `titleScreen`, let's break new changes line by line:

```
game.stage.backgroundColor = bgColors[game.rnd.between(0, bgColors.length - 1)];
```

First we change the background color of the stage to a random color chosen in `bgColors` array.

> `backgroundColor` property of stage gets and sets the background color of the stage. The color can be given as a number: `0xff0000` or a hex string: `'#ff0000'`.

I told you would have seen how to add sprites, images and various stuff in a matter of minutes.

Now we are going to add the image representing the title of the game, and in a very similar way we will add the play button too.

```
var title = game.add.image(game.width / 2, 210, "title");
```

This is how we add an image to the stage. We add the title image and store it in a variable called `title`.

> `add.image(x, y, key)` places `key` image on the stage at coordinates `x`, `y`.

Title has been placed at the half of the game width and 210 pixels from the top.

> `game.width` and `game.height` return respectively the width and the height of the game, in pixels.

When you add an image or a sprite to the stage, its registration point is always set on the top left pixel. Registration point can be easily changed, anyway.

```
title.anchor.set(0.5);
```

Now the registration point is set in the horizontal and vertical center of the image.

> The **anchor** or **registration point** 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)`.

Once game title has been placed, you will see every other asset you will add to the stage will be placed in a very similar way, just like the play button we are about to add, which only has one more argument:

```
var playButton = game.add.button(game.width / 2, game.height - 150, "playbutton",
       this.startGame);
playButton.anchor.set(0.5);
```

In this case, it's easy to see the play button will use `playbutton` key and will be placed in the horizontal center of the game and 150 pixels above the bottom of the canvas.

It will also have its registration point in the center.

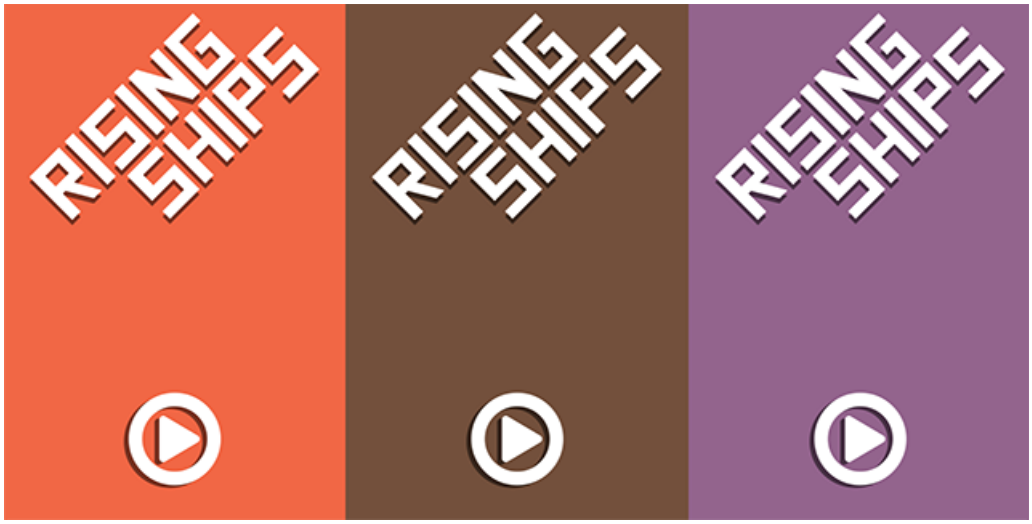
The only difference, being a button, is it will fire `startGame` method once clicked.

> `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, it's usually `this` because it's a reference to the object that owns the currently executing code.

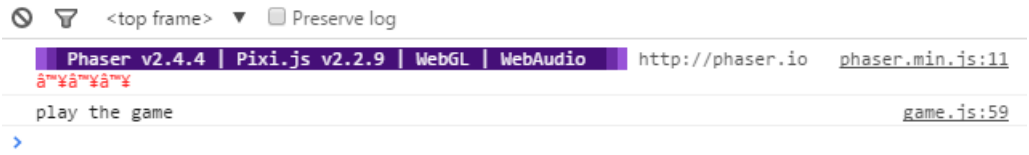
What about `startGame`? It's just a `titleScreen` method which calls `PlayGame` state. Here it is:

```
startGame: function(){
     game.state.start("PlayGame");
}
```

Run the game some times and you will see your game title screen with different background colors chosen among `bgColors` array.



Click or tap on the button, and you will be redirected on a black screen – actually `PlayGame` state – seeing a debug message on your console.



This has been a very important step as you learned how to place stuff on the stage. You can place anything you want with just a line or two.

Making play button more dynamic

While the button works well, it's just a static circle with a triangle inside. No matter how cute look your buttons, they are just flat images if you don't animate them a bit.

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

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

Let's add a tween in `create` method after the button is placed on the stage:

```
create: function(){
     game.stage.backgroundColor = bgColors[game.rnd.between(0, bgColors.length -
       1)];
     var title = game.add.image(game.width / 2, 210, "title");
     title.anchor.set(0.5);
     var playButton = game.add.button(game.width / 2, game.height - 150,
       "playbutton", this.startGame);
     playButton.anchor.set(0.5);
     var tween = game.add.tween(playButton).to({
          width: 220,
          height:220
     }, 1500, "Linear", true, 0, -1);
     tween.yoyo(true);
}
```

Everything is made to make play button grow a bit until its width and height reach 220 pixels.

> `add.tween(target)` alters one or more properties of a `target` object over a defined period of time.

Most of tween options are stored in `to` method.

> `to(properties, duration, ease, autoStart, delay, repeat)` tweens to `properties` object in `duration` milliseconds using `ease` easing. If `autoStart` is set to `true`, the tween starts as soon as it's been created. If `delay` is set, represents the `delay` in milliseconds before the tween starts, and `repeat` is the amount of times the tween should restart. If you want it to run forever, `restart` must be set to `-1`.

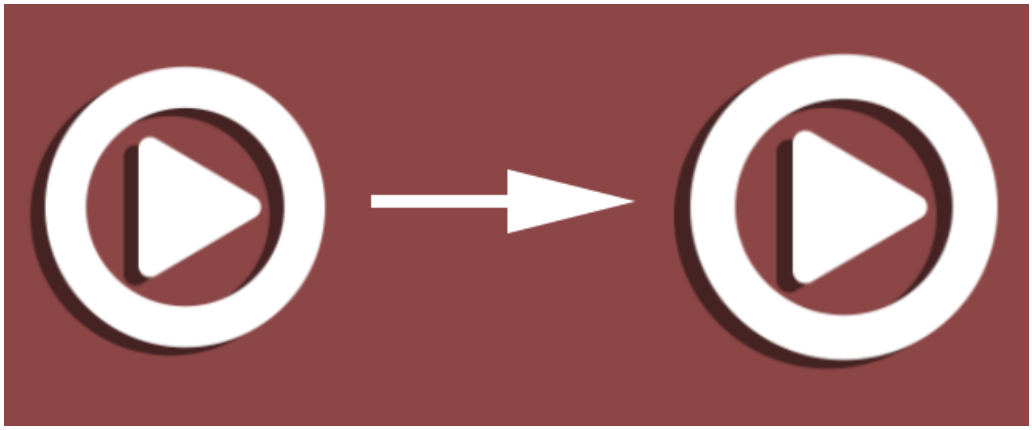
So let's translate the tween method we have just written in plain English.

Change button width and height to 220 pixels with a Linear animation which lasts 1500 milliseconds. Start now with no delay and repeat the animation forever.

To give the tween a yoyo effect rather than restarting the animation we use `yoyo` method.

> `yoyo(flag)` method if `flag` is set to `true` will make tween run through from its starting values to its end values and then play back in reverse from end to start. Used in combination with `repeat` allows you to create endless loops.

Run the game and see how play button now pulses.



With just a couple of new lines we gave life to a flat button.

Making background more interesting

You probably liked the idea of having the background with random colors, but it would be better if it could have some kind of pattern. I don't see plain backgrounds with only one color since 1990.

At this time you have two ways to have a pattern with a random color: you can draw and import a series of patterns each with its own color then randomly choose among them, or you can create one single pattern with only shades of gray then apply a tint with a random color.

Needless to say the second option is way better so we are going to draw a new image with only shades of gray. Here it is:

It's a 640x32 pixel image with some white stripes and some light gray stripes.

Let's preload it in `preload` method:

```
preload: function(){
     var loadingBar = this.add.sprite(game.width / 2, game.height / 2, "loading");
     loadingBar.anchor.setTo(0.5);
     game.load.setPreloadSprite(loadingBar);
     game.load.image("title", "assets/sprites/title.png");
     game.load.image("playbutton", "assets/sprites/playbutton.png");
     game.load.image("backsplash", "assets/sprites/backsplash.png");
}
```

Now the question is: why are we using a 32 pixels tall image when we have to cover a 960 pixels tall area?

Answer: because we will use tile sprites.

> A **tile sprite** is a sprite that has a repeating texture. The texture can be scrolled and scaled independently of the tile sprite itself. Textures will automatically wrap and are designed so that you can create game backdrops using seamless textures as a source.

You can think about them just like Photoshop patterns or CSS repeating backgrounds. Anyway, let's add these two lines to `create` method:

```
create: function(){
     var titleBG = game.add.tileSprite(0, 0, game.width, game.height,
       "backsplash");
     titleBG.tint = bgColors[game.rnd.between(0, bgColors.length - 1)];
     var title = game.add.image(game.width / 2, 210, "title");
     title.anchor.set(0.5);
     var playButton = game.add.button(game.width / 2, game.height - 150,
       "playbutton", this.startGame);
     playButton.anchor.set(0.5);
     var tween = game.add.tween(playButton).to({
          width: 220,
          height:220
     }, 1500, "Linear", true, 0, -1);
     tween.yoyo(true);
}
```

The first line places a tile sprite which covers the entire canvas.

> `add.TileSprite(x, y, width, height, key)` adds a tile sprite with the top left corner at `x`, `y` whose dimensions are `width` pixels wide, `height` pixels tall using the image with `key` name.

The second line applies a tint to `tileBG` which is the tile sprite we just added.

> `tint` property of a sprite is the tint applied to the sprite. This is a hex value. A value of `0xFFFFFF` will remove any tint effect.

Also, I think you already got the concept of “property”.

> A JavaScript **property** is a value associated with a JavaScript object. Properties can usually be changed, added, and deleted, but some are read only.

Run the game various times to see how we get a beautiful background of random colors.

Now the title screen looks better, and we can concentrate on the game itself.

The entire game will have random colored assets we will create only with `tint` property starting from images in shades of gray.

Creation of game background

The game is played inside a tunnel, and one nice thing we can do to allow as much customization as possible is to create a global variable called `tunnelWidth` which contains the width of the tunnel in pixels.

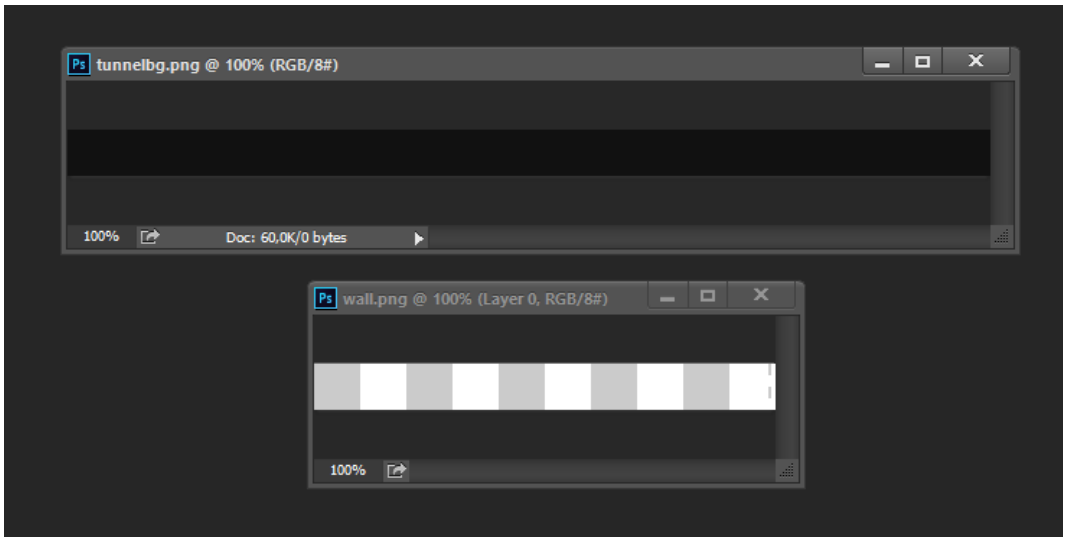
This way tunnel width can easily be adjusted and the code will look clearer.

Remember: the most customizable options and features you include in a game, the better first impression it will make.

```
var game;
var bgColors = [0xF16745, 0xFFC65D, 0x7BC8A4, 0x4CC3D9, 0x93648D, 0x7c786a,
       0x588c73, 0x8c4646, 0x2a5b84, 0x73503c];
var tunnelWidth = 256;
```

With the texture patterns I created for this game, the most good looking results are achieved when `tunnelWidth` is a multiple of 64.

The tunnel has a foreground and a background image, let's see them:



Tunnel background is a 640x32 pixels dark gray image, while tunnel wall is a 320x32 image with white and light gray stripes.

As you probably imagine this choice will lead to a tint effect applied to both background and foreground to give the tunnel random colors.

Save the images in `sprites` folder and preload them in `preload` method of `preload` object:

```
preload: function(){
     var loadingBar = this.add.sprite(game.width / 2, game.height / 2, "loading");
     loadingBar.anchor.setTo(0.5);
     game.load.setPreloadSprite(loadingBar);
     game.load.image("title", "assets/sprites/title.png");
     game.load.image("playbutton", "assets/sprites/playbutton.png");
     game.load.image("backsplash", "assets/sprites/backsplash.png");
     game.load.image("tunnelbg", "assets/sprites/tunnelbg.png");
     game.load.image("wall", "assets/sprites/wall.png");
}
```

See it? The number of preloaded assets is growing, so probably at this time you will be able to see the loading bar.

We loaded two new images.

Now we can place them as tiled sprites to create the tunnel in `create` method of `playGame` object:

```
var playGame = function(game){};
playGame.prototype = {
     create: function(){
          var tintColor = bgColors[game.rnd.between(0, bgColors.length - 1)]
          var tunnelBG = game.add.tileSprite(0, 0, game.width, game.height,
        "tunnelbg");
          tunnelBG.tint = tintColor;
          var leftWallBG = game.add.tileSprite(- tunnelWidth / 2, 0, game.width /
        2, game.height, "wall");
          leftWallBG.tint = tintColor;
          var rightWallBG = game.add.tileSprite((game.width + tunnelWidth) / 2, 0,
        game.width / 2, game.height, "wall");
          rightWallBG.tint = tintColor;
          rightWallBG.tileScale.x = -1;
     }
}
```

Let's see what we did breaking the new code in detail:

```
var tintColor = bgColors[game.rnd.between(0, bgColors.length - 1)]
```

First we draw a random tint color.

```
var tunnelBG = game.add.tileSprite(0, 0, game.width, game.height, "tunnelbg");
tunnelBG.tint = tintColor;
```

`tunnelBG` is the tunnel background, we add it to the game as a tile sprite and make it cover the entire canvas, then tint it with tint color.

```
var leftWallBG = game.add.tileSprite(- tunnelWidth / 2, 0, game.width / 2,
       game.height, "wall");
leftWallBG.tint = tintColor;
```

`leftWallBG` represent the left wall of the tunnel and it's added as a tile sprite just like the background, but we determine its position according to game width and tunnel width. Don't forget to tint it with tint color.

```
var rightWallBG = game.add.tileSprite((game.width + tunnelWidth) / 2, 0,
       game.width / 2, game.height, "wall");
rightWallBG.tint = tintColor;
```

And the same concept applies to right wall, a tile sprite with tint which position is

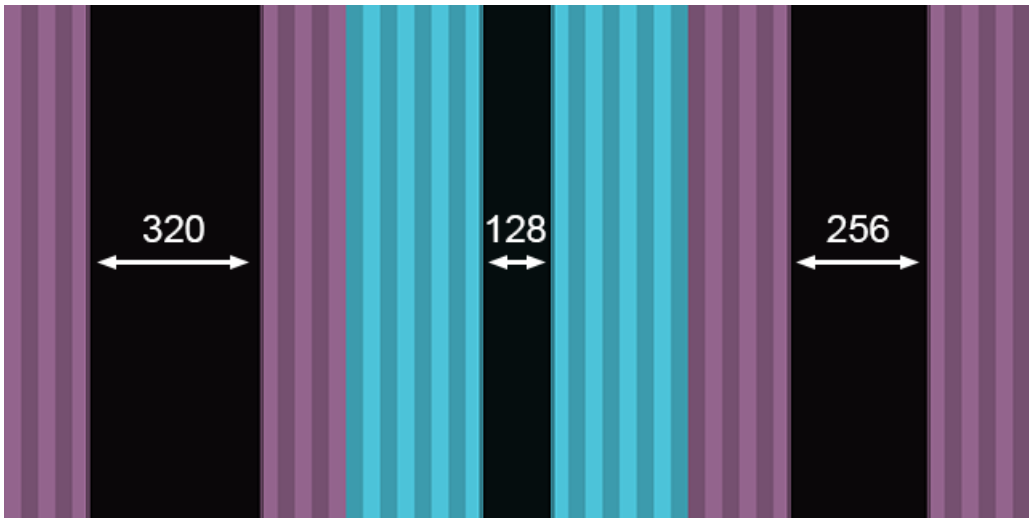
determined by game width and tunnel width.

```
rightWallBG.tileScale.x = -1;
```

To make things look properly, right wall should be specular to left wall. We can create two wall images – one for the left wall and one for the right wall – or flip horizontally the texture.

> `tileScale.x` and `tileScale.y` properties respectively set the scaling of the image that is being tiled respectively along `x` or `y` axis. Setting properties to `-1` will make the image to be flipped respectively horizontally or vertically.

Test the game with various `tunnelWidth` values and see how walls and textures are properly placed.

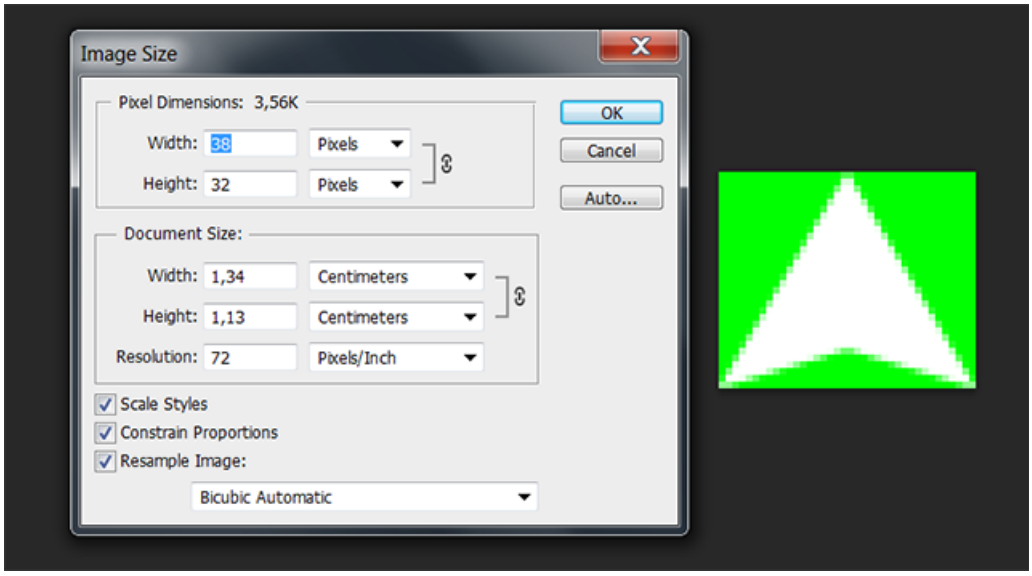


You can create your game with any tunnel width, but during the book I will be using 256 pixels as it's the width I think looks better.

Placing the spaceship

And now, ladies and gentlemen, in the red corner… I am trying to create some hype around the creation of the main actor, the protagonist of this game, the awesome spaceship!

Just kidding. The spaceship is just another sprite and adding it to the game will be a piece of cake. This is the image of the spaceship I used, and you should be bored to read I placed it into `assets/sprites` folder.



I zoomed it in a bit and placed on a green background to make you see how I created it, but it's just some kind of strange white triangle on a transparent background.

Just like all images, let's add it to the preload queue in `preload` method of `preload` object:

```
preload: function(){
     var loadingBar = this.add.sprite(game.width / 2, game.height / 2, "loading");
     loadingBar.anchor.setTo(0.5);
     game.load.setPreloadSprite(loadingBar);
     game.load.image("title", "assets/sprites/title.png");
     game.load.image("playbutton", "assets/sprites/playbutton.png");
     game.load.image("backsplash", "assets/sprites/backsplash.png");
     game.load.image("tunnelbg", "assets/sprites/tunnelbg.png");
     game.load.image("wall", "assets/sprites/wall.png");
     game.load.image("ship", "assets/sprites/ship.png");
}
```

The spaceship will be added in `create` method of `playGame` object, but this time it won't just a matter of adding a sprite. Have a look at the new lines:

```
create: function(){
     var tintColor = bgColors[game.rnd.between(0, bgColors.length - 1)]
     var tunnelBG = game.add.tileSprite(0, 0, game.width, game.height,
       "tunnelbg");
     tunnelBG.tint = tintColor;
     var leftWallBG = game.add.tileSprite(- tunnelWidth / 2, 0, game.width / 2, g
       ame.height, "wall");
     leftWallBG.tint = tintColor;
     var rightWallBG = game.add.tileSprite((game.width + tunnelWidth) / 2, 0,
       game.width / 2, game.height, "wall");
     rightWallBG.tint = tintColor;
     rightWallBG.tileScale.x = -1;
     this.shipPositions = [(game.width - tunnelWidth) / 2 + 32, (game.width +
       tunnelWidth) / 2 - 32];
     this.ship = game.add.sprite(this.shipPositions[0], 860, "ship");
     this.ship.side = 0;
     this.ship.anchor.set(0.5);
     this.game.physics.enable(this.ship, Phaser.Physics.ARCADE);
}
```

First, there's an array called `shipPositions` which contains the horizontal positions of the spaceship when it's on the left – index `0` – and on the right – index `1` – side of the tunnel. These positions are determined according to game width and tunnel width.

Then with this line:

```
this.ship = game.add.sprite(this.shipPositions[0], 860, "ship");
```

The spaceship is added to the game. We start with the spaceship on the left side of the game so it's placed at `shipPositions[0]` horizontal coordinate and `860` vertical coordinate, close to the bottom of the game.

```
this.ship.side = 0;
```

We also need to create a custom spaceship `side` property to quickly know which side it's traveling on. We start from zero as initially is placed on the left side.

```
this.ship.anchor.set(0.5);
```

We also want the spaceship to have the registration point on the horizontal and

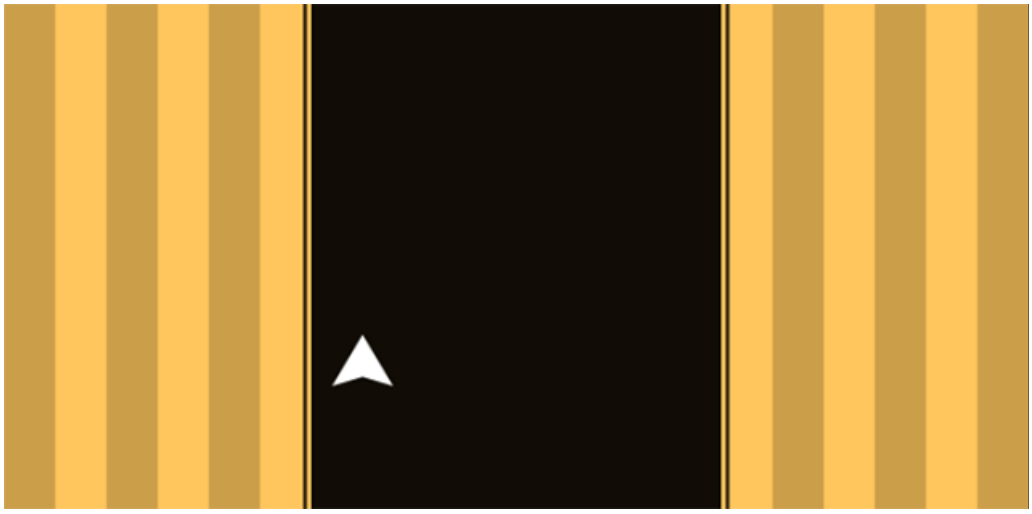
vertical center.

```
this.game.physics.enable(this.ship, Phaser.Physics.ARCADE);
```

And this is how the spaceship is enabled to be part of the physics world.

> `physics.enable(object, system)` creates a default physics body on `object` using `system` physics system.

That's all at the moment with the spaceship, test your game and see it on the bottom of the tunnel, in the left side.



Now we have to make the player able to move the spaceship.

You may wonder that is that ARCADE physics.

> `Phaser.Physics.Arcade` is a light weight AABB based collision system with basic separation.

While listing the pro and cons of all physics engines Phaser can handle is not in the scope of this book, concentrate on the “light weight” keyword because it's what we need.

Since physics systems can be very CPU expensive, always choose the one with the very minimum options you really and actually require.

Moving the spaceship horizontally

Moving the spaceship not only is very important to allow player interaction, but it will also introduce a Phaser key feature: player input.

Before diving into source code, we have to add two new global variables which will help us to fine tune the game.

The first variable is `shipHorizontalSpeed` which is the amount of milliseconds needed by the spaceship to move from a side of the tunnel to another.

The second variable is `shipMoveDelay` and is the delay in milliseconds we have to wait once the spaceship changed side of the tunnel before the player can be able to change side again.

Remember: always try to make the game as much customizable as possible in the easiest way possible.

```
var game;
var bgColors = [0xF16745, 0xFFC65D, 0x7BC8A4, 0x4CC3D9, 0x93648D, 0x7c786a,
        0x588c73, 0x8c4646, 0x2a5b84, 0x73503c];
var tunnelWidth = 256;
var shipHorizontalSpeed = 100;
var shipMoveDelay = 0;
```

In this case, the spaceship will move to the other side of the tunnel in 100 milliseconds and won't need to wait before the player can move it again.

You can play with these values and the game play will change accordingly.

Let's see how to create the movement.

First, we need a custom property to be added to the ship, to determine whether it can move or not, because there are some situations in which the spaceship won't be able to move: when it's already moving for example.

So you will already understand the first new line of `playGame` object as it's adding a `canMove` property and setting it to `true`.

You will also need to create a new method to be called each time you need to move the spaceship, have a look at the code:

```
create: function(){
     var tintColor = bgColors[game.rnd.between(0, bgColors.length - 1)]
     var tunnelBG = game.add.tileSprite(0, 0, game.width, game.height,
       "tunnelbg");
     tunnelBG.tint = tintColor;
     var leftWallBG = game.add.tileSprite(- tunnelWidth / 2, 0, game.width / 2,
       game.height, "wall");
     leftWallBG.tint = tintColor;
     var rightWallBG = game.add.tileSprite((game.width + tunnelWidth) / 2, 0,
       game.width / 2, game.height, "wall");
     rightWallBG.tint = tintColor;
     rightWallBG.tileScale.x = -1;
     this.shipPositions = [(game.width - tunnelWidth) / 2 + 32, (game.width +
       tunnelWidth) / 2 - 32];
     this.ship = game.add.sprite(this.shipPositions[0], 860, "ship");
     this.ship.side = 0;
     this.ship.canMove = true;
     this.ship.anchor.set(0.5);
     game.physics.enable(this.ship, Phaser.Physics.ARCADE);
     game.input.onDown.add(this.moveShip, this);
},
moveShip: function(){
     if(this.ship.canMove){
          this.ship.canMove = false;
          this.ship.side = 1 - this.ship.side;
          var horizontalTween = game.add.tween(this.ship).to({
               x: this.shipPositions[this.ship.side]
          }, shipHorizontalSpeed, Phaser.Easing.Linear.None, true);
          horizontalTween.onComplete.add(function(){
               game.time.events.add(shipMoveDelay, function(){
                    this.ship.canMove = true;
               }, this);
          }, this);
     }
}
```

Let's see the other new lines:

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

This is a very important line because it's where we detect a player tap or click.

> `input.onDown.add(callback, callbackContext)` is dispatched each time a pointer is pressed down, no matter if a tap or a click. `callback` function will be called in `callbackContext` context.

So when Phaser detects a click or a tap, `moveShip` method will be called in `this` context.

Basically the context is the status of a script in a given moment.

> 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.

The new lines in `create` method have been explained, time to see `moveShip` method:

```
moveShip: function(){
     if(this.ship.canMove){
          this.ship.canMove = false;
          this.ship.side = 1 - this.ship.side;
          var horizontalTween = game.add.tween(this.ship).to({
               x: this.shipPositions[this.ship.side]
          }, shipHorizontalSpeed, Phaser.Easing.Linear.None, true);
          horizontalTween.onComplete.add(function(){
               game.time.events.add(shipMoveDelay, function(){
                    this.ship.canMove = true;
               }, this);
          }, this);
     }
}
```

This is where the spaceship is moved and where there will be a pause from one move and another if `shipMoveDelay` is greater than zero.

Let's examine the script:

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

As said, the move routine is executed only if `canMove` property is `true`.

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

Now we are about to move the spaceship, so any further input to move the spaceship will be blocked until we completed the animation and waited for `shipMoveDelay`.

```
this.ship.side = 1 - this.ship.side;
```

This is how we switch spaceship `side` property from `0` to `1` and from `1` to `0`.

Now we can call the tween:

```
var horizontalTween = game.add.tween(this.ship).to({
     x: this.shipPositions[this.ship.side]
}, shipHorizontalSpeed, Phaser.Easing.Linear.None, true);
```

Everything is made to move the spaceship along x axis until it reaches the coordinate stored into `shipPositions` array at index `ship.side`.

You have already seen how tweens work and how useful they are. This is just another demonstration of their versatility.

With just a line we are able to move the spaceship from one side of the tunnel to another.

Run the game and click, tap, enjoy your spaceship as it flies from side to side.

There are still a couple of things to explain: what happens `horizontalTween` is complete? We wait for `shipMoveDelay` milliseconds then we set back `canMove` property to `true` and the spaceship is ready to move again.

> `time.events.add(delay, callback, callbackContext)` fires `callback` function in `callbackContext` context after the given amount of `delay` in milliseconds has passed.

`onComplete` is in charge of letting us know when the tween is completed.

> The `onComplete` event is fired when the tween completes. You can call its `add(callback, callbackContext)` method to run `callback` function in `callbackContext` context.

Let's see how to complete spaceship horizontal movement adding a new tween.

Adding the ghost effect

To give the spaceship a more intense feeling of speed, we will add a ghost spaceship once the player switches its position.

We will place a half-transparent copy of the spaceship in the same position the spaceship was before the player changed its side, and make it fade away then disappear.

We will be using only tween so there's nothing new, I am just showing you what you can do with the knowledge you already have.

In `moveShip` method, we add a copy of the spaceship called `ghostShip` in the same position and set its alpha to `0.5` – half transparent.

Then we create a tween to change the alpha to zero and once completed we destroy `ghostShip`.

```
moveShip: function(){
     if(this.ship.canMove){
          this.ship.canMove = false;
          this.ship.side = 1 - this.ship.side;
          var horizontalTween = game.add.tween(this.ship).to({
          x: this.shipPositions[this.ship.side]
          }, shipHorizontalSpeed, Phaser.Easing.Linear.None, true);
          horizontalTween.onComplete.add(function(){
               game.time.events.add(shipMoveDelay, function(){
                    this.ship.canMove = true;
               }, this);
          }, this);
          var ghostShip = game.add.sprite(this.ship.x, this.ship.y, "ship");
          ghostShip.alpha = 0.5;
          ghostShip.anchor.set(0.5);
          var ghostTween = game.add.tween(ghostShip).to({
               alpha: 0
          }, 350, Phaser.Easing.Linear.None, true);
          ghostTween.onComplete.add(function(){
               ghostShip.destroy();
          });
     }
}
```

Run the game, move the spaceship from one side to another and see the ghost ship give an extra feeling of high speed.

You already learned how to add sprites and images to the stage, now you learned how to remove them once you don't need them anymore: with `destroy` method.

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

To furthermore increase the feeling of high speed, we will add another effect.

Adding smoke trail with particles

We are about to introduce a new feature which I am sure you will find extremely useful to add special effect to your games: particles.

Thanks to Phaser particle system we can use a large number of very small sprites to simulate certain kinds of chaotic phenomena – such as fire, explosions, smoke, waterfalls and so on – which would be otherwise very hard to reproduce with conventional programming techniques. We will create a smoke trail with particles.

Talking about very small sprites, a new asset is needed, a 4x4 pixels white square which will represent a smoke particle. Here is how we load it in `preload` method:

```
preload: function(){
     var loadingBar = this.add.sprite(game.width / 2, game.height / 2, "loading");
     loadingBar.anchor.setTo(0.5);
     game.load.setPreloadSprite(loadingBar);
     game.load.image("title", "assets/sprites/title.png");
     game.load.image("playbutton", "assets/sprites/playbutton.png");
     game.load.image("backsplash", "assets/sprites/backsplash.png");
     game.load.image("tunnelbg", "assets/sprites/tunnelbg.png");
     game.load.image("wall", "assets/sprites/wall.png");
     game.load.image("ship", "assets/sprites/ship.png");
     game.load.image("smoke", "assets/sprites/smoke.png");
}
```

Then the idea is to create a particle emitter which will simulate the smoke and

placing it close to the rear of the spaceship.

> An **emitter** is a lightweight particle emitter that uses ARCADE Physics. It can be used for one-time explosions or for continuous effects like rain and fire. All it really does is launch Particle objects out at set intervals, and fixes their positions and velocities accordingly.

Since we are making a game which has to run smoothly on various platforms, some of them not that powerful, we will also need to be careful when placing a lot of stuff on the screen. That's why we will try to get to a compromise between the number of particles used and the realism of the effect.

We need to add some lines to `create` method in `playGame` object:

```
var playGame = function(game){};
playGame.prototype = {
     create: function(){
          // same as before
          this.smokeEmitter = game.add.emitter(this.ship.x, this.ship.y + 10, 20);
          this.smokeEmitter.makeParticles("smoke");
          this.smokeEmitter.setXSpeed(-15, 15);
          this.smokeEmitter.setYSpeed(50, 150);
          this.smokeEmitter.setAlpha(0.5, 1);
          this.smokeEmitter.start(false, 1000, 40);
     },
     moveShip: function(){
          // same as before
     },
     update: function(){
          this.smokeEmitter.x = this.ship.x;
          this.smokeEmitter.y = this.ship.y;
     }
}
```

This is a completely new concept so let's explain it line by line:

```
this.smokeEmitter = game.add.emitter(this.ship.x, this.ship.y + 10, 20);
```

Adding a particle emitter, in this case called `smokeEmitter`, is not that different than adding a sprite.

> `add.emitter(x, y, max)` places a particle emitter in position `x`, `y` capable of emitting up to `max` particles at the same time.

Now that the emitter has been placed just above the spaceship, let's assign it the image which will be used as particle:

```
this.smokeEmitter.makeParticles("smoke");
```

We are using the smoke image we just preloaded.

> `makeParticles(key)` generates a new set of particles using `key` image name.

Each particle now should have a horizontal and vertical speed.

Being a smoke trail of a spaceship in a vertical endless runner, vertical speed will be way higher than horizontal speed.

```
this.smokeEmitter.setXSpeed(-15, 15);
this.smokeEmitter.setYSpeed(50, 150);
```

This should grant a good smoke effect.

> `setXspeed(min, max)` and `setYspeed(min, max)` respectively set the horizontal and vertical speed of each particle as a random value from `min` to `max` pixels by second.

In the same way we can play with particle transparency:

```
this.smokeEmitter.setAlpha(0.5, 1);
```

Changing the transparency adds realism to smoke effect.

> `setAlpha(min, max)` sets the alpha of each particle to a random value from `min` to `max`. Remember `0` is completely transparent and `1` is completely opaque.

Finally we are ready to make the emitter start:

```
this.smokeEmitter.start(false, 1000, 40);
```

And the particles will be generated

> `start(explode, lifespan, frequency)` starts emitting particles. `explode` is a Boolean value which says hether the particles should all burst out at once (`true`) or at the frequency given (`false`). Particle will last for `lifespan` milliseconds and will be emitted every `frequency` milliseconds, if `explode` is set to `false`.

We also have to `create` update method to `playGame` object.

Phaser will recognize if in a state there's a method called `update` will execute it at each frame.

This will make particles follow the spaceship:

```
update: function(){
     this.smokeEmitter.x = this.ship.x;
     this.smokeEmitter.y = this.ship.y;
}
```

Now test the game and see your particle trail following your spaceship

It will always follow the spaceship because at each frame its position will be adjusted by `update` method.

Making the spaceship rise

We have to slowly move the spaceship to the top of the screen. It will be very easy as it's just another tween.

First, a new global variable which stores the amount of milliseconds needed to reach the top of the screen.

We'll call the variable `shipVerticalSpeed` and we want it to be customizable.

```
var game;
var bgColors = [0xF16745, 0xFFC65D, 0x7BC8A4, 0x4CC3D9, 0x93648D, 0x7c786a,
       0x588c73, 0x8c4646, 0x2a5b84, 0x73503c];
var tunnelWidth = 256;
var shipHorizontalSpeed = 100;
var shipMoveDelay = 0;
var shipVerticalSpeed = 15000;
```

The spaceship will reach the top of the screen in 15 seconds.

Now we need a tween which will last shipVerticalSpeed milliseconds and move the spaceship to the top of the game area, which is at `y` coordinate equal to zero.

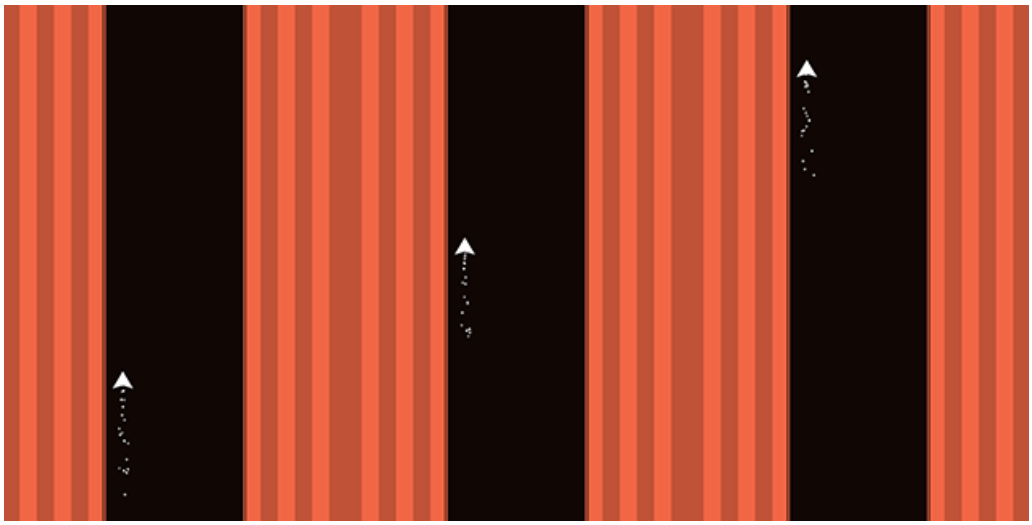
I am sure you already know how to do it:

In `create` method of `playGame` function, simply add this new tween:

```
create: function(){
     // same as before
     this.verticalTween = game.add.tween(this.ship).to({
          y: 0
     }, shipVerticalSpeed, Phaser.Easing.Linear.None, true);
}
```

I told it was easy once you get used to tweens.

Run the game, and watch your spaceship slowing moving up.



Obviously while the spaceship is moving up you are also able to move them left

and right, because tweens work independently.

Using swipe to move back the ship

The spaceship is able to rise, but we have to find a way to move it back down.

We are using a swipe to do it, and since Phaser does not feature a native swipe detection we are going to build it on our own.

First, we need a global variable to store how many pixels the player has to move the finger before we consider it a swipe.

```
var game;
var bgColors = [0xF16745, 0xFFC65D, 0x7BC8A4, 0x4CC3D9, 0x93648D, 0x7c786a,
       0x588c73, 0x8c4646, 0x2a5b84, 0x73503c];
var tunnelWidth = 256;
var shipHorizontalSpeed = 100;
var shipMoveDelay = 0;
var shipVerticalSpeed = 15000;
var swipeDistance = 10;
```

`swipeDistance` tells us any movement greater than 10 pixels will be considered a swipe.

You are free to change this value obviously, and I am encouraging you to do as it's been declared as global variable.

Now let's explain swipe idea: once the spaceship is moving from one side of the tunnel to another, we are checking if the player moves the finger and we detect a swipe, we stop the current `verticalTween` tween then we call a new tween to move the spaceship down. Perfect, but we still have to decide how to see if the player is swiping.

Each finger or mouse gesture starts with a mouse press or with a touch.

We are already able to detect when the player starts clicking or touching the game. It's in the routine which handles ship movement from side to side.

When the player interacts with the screen, Phaser can register the coordinates of the input. Inside its routines, Phaser knows there is an active pointer – that how it calls the finger or the mouse arrow – and call also track it over time.

The solution is simple: when we have an active pointer, we keep following it and see if it's moving. Then if the distance between the starting input coordinates and the current input coordinates is greater than `swipeDistance`, we can say the player is swiping.

We are going to modify `playGame` object heavily but easily, have a look:

```
var playGame = function(game){};
playGame.prototype = {
     create: function(){
          // same as before
          this.ship = game.add.sprite(this.shipPositions[0], 860, "ship");
          this.ship.side = 0;
          this.ship.canMove = true;
          this.ship.canSwipe = false;
          this.ship.anchor.set(0.5);
          game.physics.enable(this.ship, Phaser.Physics.ARCADE);
          game.input.onDown.add(this.moveShip, this);
          game.input.onUp.add(function(){
               this.ship.canSwipe = false;
          }, this);
          this.smokeEmitter = game.add.emitter(this.ship.x, this.ship.y + 10, 20);
          // same as before
     },
     moveShip: function(){
          this.ship.canSwipe = true;
          if(this.ship.canMove){
               // same as before
          }
     },
     update: function(){
          this.smokeEmitter.x = this.ship.x;
          this.smokeEmitter.y = this.ship.y;
          if(this.ship.canSwipe){
               if(Phaser.Point.distance(game.input.activePointer.positionDown,
                         game.input.activePointer.position) > swipeDistance){
                    this.restartShip();
               }
          }
     },
     restartShip: function(){
          this.ship.canSwipe = false;
          this.verticalTween.stop();
          this.verticalTween = game.add.tween(this.ship).to({
               y: 860
          }, 100, Phaser.Easing.Linear.None, true);
          this.verticalTween.onComplete.add(function(){
               this.verticalTween = game.add.tween(this.ship).to({
                    y: 0
               }, shipVerticalSpeed, Phaser.Easing.Linear.None, true);
          }, this)
     }
}
```

There's almost nothing new in the above code, anyway let's explain it line by line:

```
this.ship.canSwipe = false;
```

We set a `canSwipe` property to the spaceship, initially set to `false` because at the beginning the spaceship cannot swipe.

```
game.input.onUp.add(function(){
     this.ship.canSwipe = false;
}, this);
```

`canSwipe` is also set to `false` when the player releases the input – mouse or finger – from the game.

If there's no input, there can't be a swipe.

> `input.onUp.add(function, context)` executes `function` function in `context` context each time an input is released.

When `moveShip` method is called, that is when the player tapped or clicked on the game, we can set `canSwipe` to `true` and start checking for swipes.

```
this.ship.canSwipe = true;
```

In `update` method, if the ship can swipe, we check if the distance from the current mouse or finger position and the starting mouse or finger position is greater than `swipeDistance`.

This will be the case we can say the player is swiping.

```
if(this.ship.canSwipe){
     if(Phaser.Point.distance(game.input.activePointer.positionDown,
        game.input.activePointer.position) > swipeDistance){
          this.restartShip();
     }
}
```

In this case we can say the player is swiping.

`input.activePointer.positionDown` property contains the point with the x/y values of an input when is was set down.

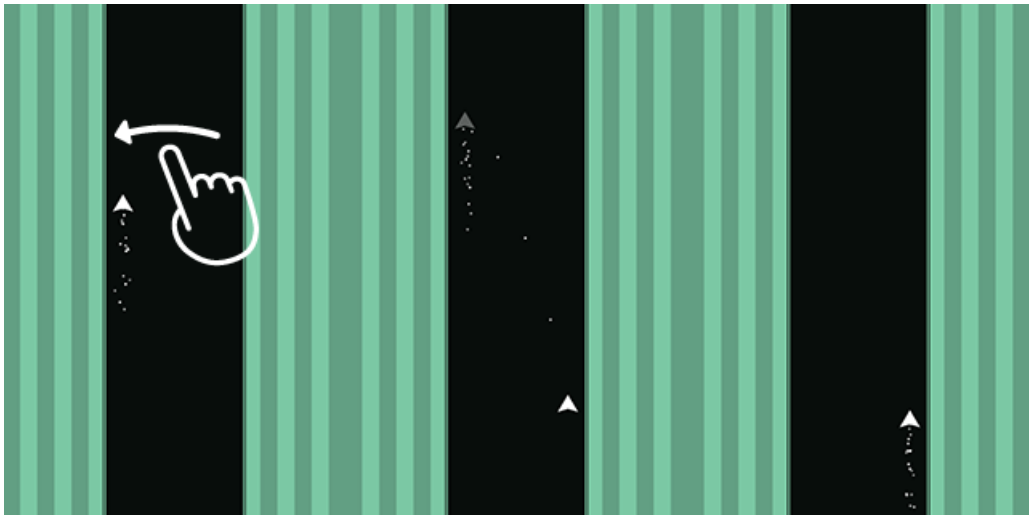
`input.activePointer.position` property contains the point with the current x/y values of the input.

`Point.distance(a, b)` returns the euclidean distance between `a` and `b` Point objects.

What happens when the player is swiping? We call `restartShip` method:

```
restartShip: function(){
     this.ship.canSwipe = false;
     this.verticalTween.stop();
     this.verticalTween = game.add.tween(this.ship).to({
          y: 860
     }, 100, Phaser.Easing.Linear.None, true);
     this.verticalTween.onComplete.add(function(){
          this.verticalTween = game.add.tween(this.ship).to({
               y: 0
          }, shipVerticalSpeed, Phaser.Easing.Linear.None, true);
     }, this)
}
```

Let's have a look at the game: once the spaceship is running up, swipe to see how it moves back down then restarts moving up.



There are four things to do in when restarting the spaceship: first we set `canSwipe`

to `false` because the spaceship is already reacting to a swipe.

Then we stop the vertical tween to create a fast tween to quickly move the ship down and when this tween is completed we restart the tween which should move the spaceship up.

`stop()` method of a tween stops the tween.

And the whole part managing spaceship movement is completed. Now let's put the spaceship in trouble.

Adding barriers

Before we start adding barriers, let's read again what I wrote in previous chapter:

“And the whole part managing spaceship movement is completed”.

Wait a moment – you may think – the spaceship isn't actually moving through the tunnel. I was expecting the spaceship to run through the tunnel, with the camera following it. All in all we are making an endless runner!

You're right. The spaceship **should** move through the tunnel and the camera **should** follow it. But we are game developers, and we make tricks. The ship is not moving. The camera is not moving. Only the obstacles – and other few things such as stuff to collect if any – move towards the ship.

The result is we give the player the same feeling as if the spaceship were flying with the camera following it, with far less effort. This is how most endless runner are made. Player does not run. Environment does.

Back to our game, we are going to add spaceship worst enemy: a barrier.

You will see how the barrier is the most important actor in this game, because it's the sprite which with its vertical velocity will give the player the fake feeling the spaceship is moving through a tunnel. Lie. The only things which move are the barriers.

Talking about moving the barriers, let's define a new global variable called `barrierSpeed` which will store the vertical barrier speed, in pixels per second.

```
var game;
var bgColors = [0xF16745, 0xFFC65D, 0x7BC8A4, 0x4CC3D9, 0x93648D, 0x7c786a,
       0x588c73, 0x8c4646, 0x2a5b84, 0x73503c];
var tunnelWidth = 256;
var shipHorizontalSpeed = 100;
var shipMoveDelay = 0;
var shipVerticalSpeed = 15000;
var swipeDistance = 10;
var barrierSpeed = 280;
```

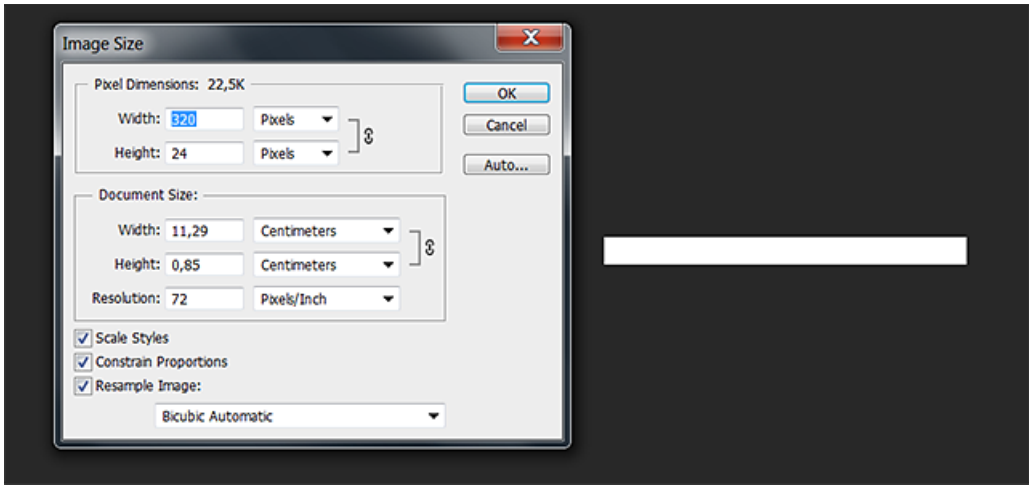
Before we start talking about placing barriers, let me introduce a bit of theory about the way we should be adding barriers.

In your games, it's always a good practice to group actors. Actors in a game can be seen like files in a folder. If you have a single folder with texts, images, sounds and movies it will be difficult to check for all texts or do some operation on just sounds and movies. That's why you know you have to create different folders to conveniently store different file types.

Actors in a game should follow the same rules, so the priority now is to know how to group all the barriers we are about to create. This is where Phaser groups help us a lot.

> A **Group** is a container for display objects including Sprites and Images.

We'll get back to groups in a moment, meanwhile here it is the barrier image we are saving into `sprites` folder:



It's a 320x24 white rectangle. It must be white to allow us to tint as we want.

Let's preload barrier sprite adding a line to `preload` method in `preload` object:

```
preload: function(){
     var loadingBar = this.add.sprite(game.width / 2, game.height / 2, "loading");
     loadingBar.anchor.setTo(0.5);
     game.load.setPreloadSprite(loadingBar);
     game.load.image("title", "assets/sprites/title.png");
     game.load.image("playbutton", "assets/sprites/playbutton.png");
     game.load.image("backsplash", "assets/sprites/backsplash.png");
     game.load.image("tunnelbg", "assets/sprites/tunnelbg.png");
     game.load.image("wall", "assets/sprites/wall.png");
     game.load.image("ship", "assets/sprites/ship.png");
     game.load.image("smoke", "assets/sprites/smoke.png");
     game.load.image("barrier", "assets/sprites/barrier.png");
}
```

Now it's time to see a very important feature, not only in Phaser but in the whole world of programming: extending classes.

You can imagine barriers will be created as sprites, just like we added the spaceship.

But there will be a lot of barriers, which will behave as sprites but also will do some extra stuff basic sprites aren't able to do, like moving from top to bottom of the game, and some other things you'll see later.

This said, we can consider the barrier as “sprites with something more”. That's why we will extend `Sprite` class.

The idea at the base of extending a class is simple but powerful: when you want to create a new class and there is already a class that includes some of the code that you want, you can build you new class over the existing class.

When we extend an existing class, the new class inherits all the attributes and methods of the parent class.

Back to our barriers, we could just create them as simple sprites, but I like to think the scope of this book is not just a “create this game for the sake of creating this game”.

I also want to give you all the core concepts which you will use to create more

complex games on your own.

Add some lines to `create` method in `playGame` object:

```
create: function(){
     // same as before
     this.barrierGroup = game.add.group();
     var barrier = new Barrier(game, barrierSpeed, tintColor);
     game.add.existing(barrier);
     this.barrierGroup.add(barrier);
}
```

This was a great start, because we have four lines with four new concepts, so let's break and explain the code line by line:

```
this.barrierGroup = game.add.group();
```

`barrierGroup` is the group which will contain all barriers.

You can add it just like you previously added sprites, using `add`.

`add.group()` adds a group to the game.

Now, let's add the barrier.

```
var barrier = new Barrier(game, barrierSpeed, tintColor);
```

`barrier` is the variable and `Barrier` is the name of the new class.

Since it's our custom class, we can also pass our own arguments, which in this case are the game itself, the speed and the tint color.

```
game.add.existing(barrier);
```

And this is how we add the barrier to the game, using `add` as we are used to.

`add.existing(displayObject)` adds an existing `displayObject` display object to the game world.

Finally we tell Phaser we want the barrier to be part of `barrierGroup` group:

```
this.barrierGroup.add(barrier);
```

This is when `add` comes into play once again.

> `add(object)` method of a group adds an existing `object` object as the top child in this group.

Everything is ready to place a barrier as an extended class, except we didn't already code it, so let's go straight to the point.

All the code needed to define `Barrier` class must be written outside any object, method or function, at the same level you declared `game` variable itself and all global variables. This will make your class to be available anywhere in the game.

```
Barrier = function (game, speed, tintColor) {
     var positions = [(game.width - tunnelWidth) / 2, (game.width + tunnelWidth) /
       2];
     var position = game.rnd.between(0, 1);
     Phaser.Sprite.call(this, game, positions[position], -100, "barrier");
     var cropRect = new Phaser.Rectangle(0, 0, tunnelWidth / 2, 24);
     this.crop(cropRect);
     game.physics.enable(this, Phaser.Physics.ARCADE);
     this.anchor.set(position, 0.5);
     this.tint = tintColor;
     this.body.velocity.y = speed;
};

Barrier.prototype = Object.create(Phaser.Sprite.prototype);
Barrier.prototype.constructor = Barrier;
```

That easy? Yes, that easy. We are extending a sprite so we have to code only the stuff regular sprites don't already do.

Look how we made `Barrier` class as there are some new concepts to see:

```
Barrier = function (game, speed, tintColor) {
     // content
}

Barrier.prototype = Object.create(Phaser.Sprite.prototype);
Barrier.prototype.constructor = Barrier;
```

This is the blueprint of the creation of a class which extends Phaser `Sprite` class.

First we declare the function which will create the barrier itself, with its arguments as shown before. Then we specify its prototype is built upon `Phaser.Sprite.prototype` which basically is Phaser `Sprite` class.

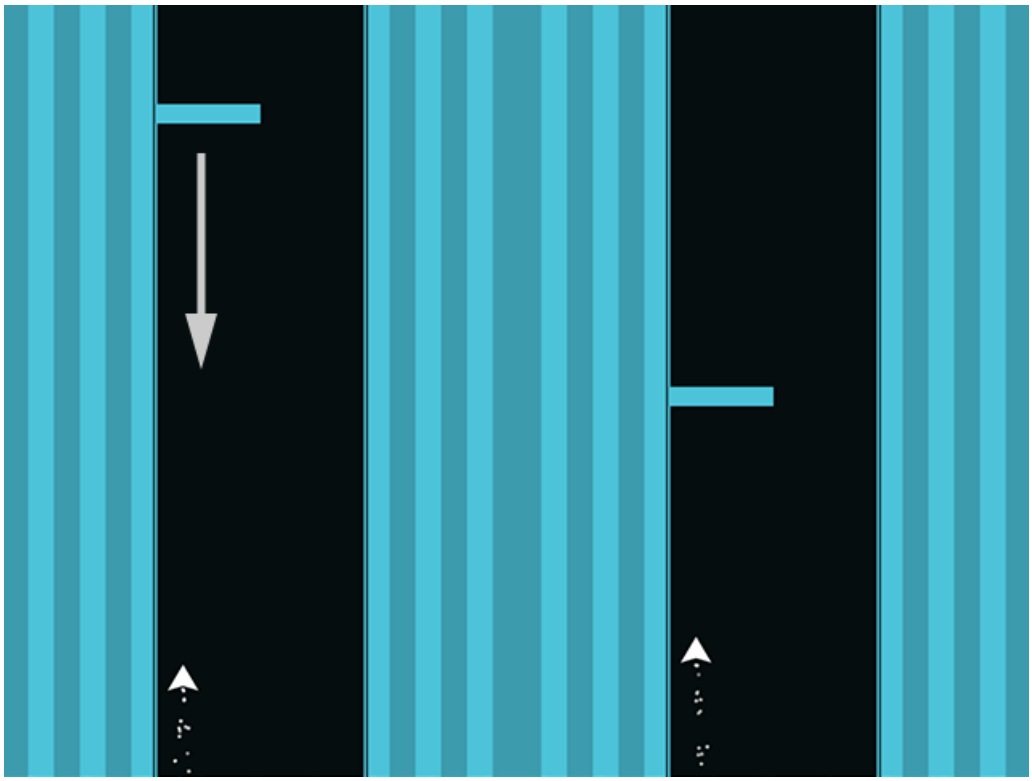
Finally we specify the constructor of the class which is Barrier. That's why we created the barrier with

```
var barrier = new Barrier(game, barrierSpeed, tintColor);
```

because `Barrier` is the constructor.

> In class based object oriented programming, a **constructor** of a class is a special function called to create an object which belongs to the class.

You will see how easy will be the process of adding how many barriers we want simply creating new `Barrier` instances, but before let's have a look at the game:



Look how the barrier travels from top to bottom of the stage.

Back to `Barrier` content now:

```
var positions = [(game.width - tunnelWidth) / 2, (game.width + tunnelWidth) / 2];
var position = game.rnd.between(0, 1);
```

Barriers can be created both on the left and the right side of the tunnel, so `positions` is the array which stores left and right barrier positions, according to game width and tunnel width. Then `position` variable randomly choose a number which can be `0` or `1`.

```
Phaser.Sprite.call(this, game, positions[position], -100, "barrier");
```

With `call` method we are invoking the creation of a sprite passing `this` as the barrier itself, the game references and the other arguments you already saw in the creation of a sprite, such has the horizontal and vertical positions and the key of the graphic resource.

```
var cropRect = new Phaser.Rectangle(0, 0, tunnelWidth / 2, 24);
```

A new concept ready to be explained: the barrier image is 320x24 pixels, but we don't need the barrier to be that wide.

Barrier width will vary according to tunnel width, so we have to crop the barrier accordingly.

The idea is to create a rectangle with the crop area – `cropRect` – then crop the sprite.

> `new Rectangle(x, y, width, height)` creates a new `Rectangle` object with the top left corner specified by the `x` and `y` parameters and with the specified `width` and `height` parameters.

Now that crop area has been defined, let's just crop the sprite:

```
this.crop(cropRect);
```

And now the barrier will have the proper width needed for our game.

> `crop(rectangle)` allows you to crop the texture being used to display the sprite using `rectangle` crop area. Sprite `width` and `height` properties will be adjusted accordingly.

Just like we did with the spaceship, we are enabling ARCADE physics on the barrier.

```
game.physics.enable(this, Phaser.Physics.ARCADE);
```

Then we tint the sprite and set its anchor point, which will be `(0, 0.5)` – left center pixel – for left barriers and `(1, 0.5)` – right center pixel – for right barriers.

```
this.anchor.set(position, 0.5);
this.tint = tintColor;
```

We can also assign a velocity to ARCADE physics bodies, and in this case we are assigning `speed` vertical velocity.

```
this.body.velocity.y = speed;
```

Now the barrier will move vertically by `speed` pixels by second.

> `body.velocity` is the speed of `body`, measured in pixels per second. `body.velocity.x` is the horizontal speed and `body.velocity.y` is the vertical speed.

Congratulations. I mean it. You survived barrier creation and class extension.

It will be easier from now on.

Removing Barriers

I know, you just made such an effort to create a barrier, and I am showing you how to remove it. That's rude.

There's no point in keeping a barrier in the game forever. Sooner or later, it will move outside game canvas, and this is the moment to free some memory removing it from the game.

So we will create a barrier `update` method which like all `update` methods in Phaser will be executed at each frame:

```
Barrier.prototype.update = function(){
     if(this.y > game.height){
          this.destroy();
     }
}
```

As soon as the vertical position is greater than the height of the game, we destroy the barrier. Have a look at how I am checking for the height of the game. The game is known for being 960 pixels tall, but I am not using `960` in the `if` statement but game `height` property. This choice will be very important later in the game.

Continuously adding barriers

Having one single, lonely barrier running towards your spaceship is not that exciting, because we expect a lot of barriers to be avoided.

The first thing to decide – and custom whenever we want – is the distance from a barrier and another. We need another global function called `barrierGap` which will store the distance between barriers, in pixels.

```
var game;
// same as before
var barrierGap = 120;
```

To achieve a 120 pixels distance between two barriers, we have to create a new

barrier once the previous barrier traveled for 120 pixels, then create a new one when the newborn barrier traveled for 120 pixels, and so on.

No matter the way we are going to add new barriers, it's easy to imagine we are about to write some code to be used a lot of time, each time a barrier needs to be added.

When you know you will be using the same code a lot of times in your script, always try to turn it into a function, or a method in this case, like `addBarrier` method which is added to `playGame` object.

It takes two arguments: the group where to add the barrier and the tint color of the barrier.

The code itself does not change, it's just the way we call it:

```
var playGame = function(game){};
playGame.prototype = {
     create: function(){
          // same as before
          this.barrierGroup = game.add.group();
          this.addBarrier(this.barrierGroup, tintColor);
     },
     moveShip: function(){
          // same as before
     },
     update: function(){
          // same as before
     },
     restartShip: function(){
          // same as before
     },
     addBarrier: function(group, tintColor){
          var barrier = new Barrier(game, barrierSpeed, tintColor);
          game.add.existing(barrier);
          group.add(barrier);
     }
}
```

With a new `playGame` method which can create a new barrier anytime, the idea is to add a new barrier when the previous barrier gets to a vertical position greater than `barrierGap` value.

Since each barrier will stay for some time in a vertical position greater than `barrierGap` value before it leaves the game to the bottom and gets destroyed, we

must ensure each barrier can create only one new barrier as soon as it gets to `barrierGap` vertical position.

For this reason we need to add a new property to each barrier, called `placeBarrier`.

Initially is set to `true` because each barrier when it's created is capable to place another barrier. Let's modify a bit `Barrier` class constructor:

```
Barrier = function (game, speed, tintColor) {
     // same as before
     this.placeBarrier = true;
};
```

And a couple of lines also need to be added to `update` method:

```
Barrier.prototype.update = function(){
     if(this.placeBarrier && this.y > barrierGap){
          this.placeBarrier = false;
          playGame.prototype.addBarrier(this.parent, this.tint);
     }
     if(this.y > game.height){
          this.destroy();
     }
}
```

What happens when `placeBarrier` is `true` and the vertical position of the barrier is greater than `barrierGap`?

As you can see, `placeBarrier` method is set to `false`.

This will ensure we won't enter anymore in the `if` statement because the condition now it will return `false` no matter the vertical position of the barrier.

Look what happens inside the `if` statement:

```
playGame.prototype.addBarrier(this.parent, this.tint);
```

We simply call `addBarrier` method we created earlier in `playGame` object, and a new barrier is created and ready to generate another barrier which will generate another barrier and so on.

Test the game and see how you can get infinite barriers in random positions.

This is the real meaning of an endless runner.

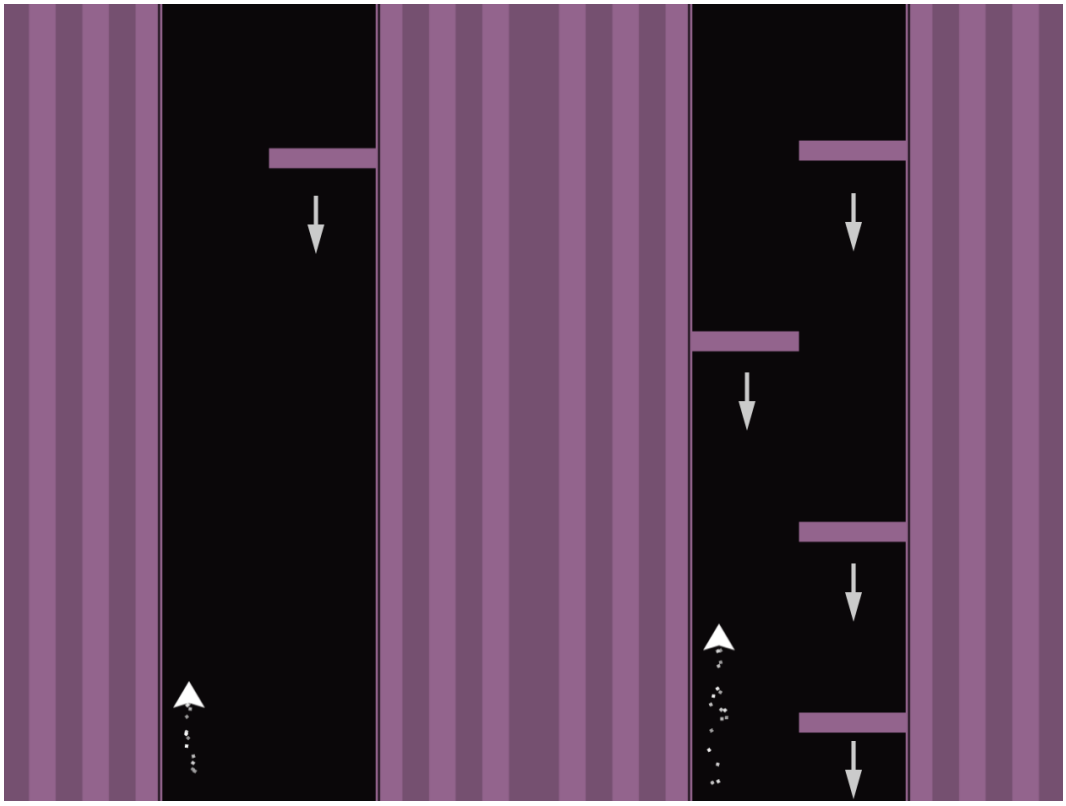
Now at each play the game will generate an infinite tunnel.

You will never play the same tunnel twice thanks to randomization, and the player will feel like the spaceship is actually traveling through an endless tunnel.

I already explained this is a fake effect, the spaceship is not moving but the whole environment moving towards the spaceship.

But it works, with a few JavaScript lines we are creating a realistic endless runner.

And if you were skeptic at first, maybe seeing a spaceship surrounded by approaching barriers will change your mind, run the game:



Now, let's turn barriers into deadly obstacles, you will see how easily Phaser together with ARCADE physics manage collisions between objects.

Checking collisions

Having a lot of barriers running is quite pointless if they do not harm your spaceship.

That's why we need to check for collisions between the spaceship and the barriers.

ARCADE physics provides a great support for collision detection, which will allow us to check for collisions with only one line of code rather than bothering with complex operations hunting for shape overlapping.

It's not the most accurate collision detection ever, but it's exactly what we need for these kind of games.

Let's add a couple of lines to `update` method in `playGame` object:

```
update: function(){
     this.smokeEmitter.x = this.ship.x;
     this.smokeEmitter.y = this.ship.y;
     if(this.ship.canSwipe){
          if(Phaser.Point.distance(game.input.activePointer.positionDown,
               game.input.activePointer.position) > swipeDistance){
               this.restartShip();
          }
     }
     game.physics.arcade.collide(this.ship, this.barrierGroup, function(s, b){
          game.state.start("GameOverScreen");
     })
}
```

At each frame we check for collision between the spaceship and any child of `barrierGroup` group, that is to any barrier.

> `collide(object1, object2, collideCallback)` checks for collision between `object1` and `object2`, which can be either sprites or groups. If an object is a group, the collision test is made against all the children of the group. `collideCallback(obj1, obj2)` is called in case of collision and its arguments are the two objects which collided.

In our case the callback function has two arguments which are the two objects which collided.

The first object is the spaceship – that's why I called it `s` – and the second object is

the barrier involved in the collision – `b`.

We aren't doing anything with them at the moment since we are only launching `GameOverScreen` state, but we will find them useful later in the development of the game.

And since we are going to launch `GameOverScreen` state, at least we want to prompt something on the console.

```
var gameOverScreen = function(game){};
gameOverScreen.prototype = {
     create:function(){
          console.log("game over");
     }
}
```

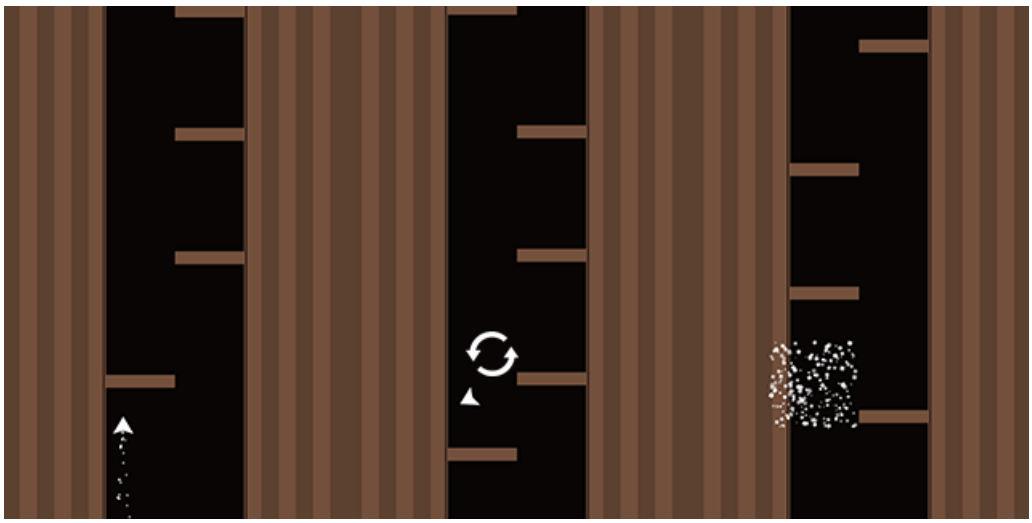
Run the game and crash into a barrier, you will be taken to a black screen – actually `GameOverScreen` – and you will see the game over message in your browser console.

With this collision test we can handle when the spaceship comes to a bad end, but there's still something to do to improve this feature.

Dying with style

Once the spaceship hits a barrier, we will add a tween to simulate the pilot lost control of the spaceship, then we'll make it explode with a particle effect.

This will look really nice, guaranteed.



First, we need to add a new property to spaceship. The property is called `destroyed` and will help us to know when the spaceship is going to be destroyed.

Initially starts at `false` since the spaceship is not going to be destroyed – yet.

Add this line to `create` property in `playGame` object:

```
create: function(){
     // same as before
     this.ship = game.add.sprite(this.shipPositions[0], 860, "ship");
     this.ship.side = 0;
     this.ship.destroyed = false;
     this.ship.canMove = true;
     this.ship.canSwipe = false;
     this.ship.anchor.set(0.5);
     //
}
```

Why are doing all this? The game already works. When the spaceship hits a barrier, we start GameOverScreen state. It's enough, isn't it?

When you are playing, there's nothing worse than suddenly switching from actively playing to game over screen without any kind of feedback. It's annoying and gives the feeling the game is somehow incomplete, so we are going to add some eye candy effects when the player dies.

The first thing is to run the collision check only if the spaceship isn't already about to be destroyed. That's why there's that `if` statement in `update` method:

```
update: function(){
     this.smokeEmitter.x = this.ship.x;
     this.smokeEmitter.y = this.ship.y;
     if(this.ship.canSwipe){
          if(Phaser.Point.distance(game.input.activePointer.positionDown,
        game.input.activePointer.position) > swipeDistance){
               this.restartShip();
          }
     }
     if(!this.ship.destroyed){
          game.physics.arcade.collide(this.ship, this.barrierGroup, function(s, b)
               {
               this.ship.destroyed = true
               this.smokeEmitter.destroy();
               var destroyTween = game.add.tween(this.ship).to({
                    x: this.ship.x + game.rnd.between(-100, 100),
                    y: this.ship.y - 100,
                    rotation: 10
               }, 1000, Phaser.Easing.Linear.None, true);
               destroyTween.onComplete.add(function(){
                    var explosionEmitter = game.add.emitter(this.ship.x,
                      this.ship.y, 200);
                    explosionEmitter.makeParticles("smoke");
                    explosionEmitter.setAlpha(0.5, 1);
                    explosionEmitter.minParticleScale = 0.5;
                    explosionEmitter.maxParticleScale = 2;
                    explosionEmitter.start(true, 2000, null, 200);
                    this.ship.destroy();
                    game.time.events.add(Phaser.Timer.SECOND * 2, function(){
                         game.state.start("GameOverScreen");
                    });
               }, this);
          }, null, this)
     }
}
```

Let's have a look at the new lines we added:

```
this.ship.destroyed = true;
this.smokeEmitter.destroy();
```

When we start the explosion routine, we have to set `destroyed` property to `true`

so we won't check for collisions anymore. Also, just like a sprite, we remove the smoke emitter using `destroy()` method.

```
var destroyTween = game.add.tween(this.ship).to({
     x: this.ship.x + game.rnd.between(-100, 100),
     y: this.ship.y - 100,
     rotation: 10
}, 1000, Phaser.Easing.Linear.None, true);
```

The animation showing the spaceship out of control is just a tween which modifies `x`, `y` and `rotation` properties together. Phaser allows to tween more than one property at once, it's an interesting feature because we can modify everything we place into `to` method. The tween lasts one second, then it's time to destroy the spaceship.

```
destroyTween.onComplete.add(function(){
     var explosionEmitter = game.add.emitter(this.ship.x,
       this.ship.y, 200);
     explosionEmitter.makeParticles("smoke");
     explosionEmitter.setAlpha(0.5, 1);
     explosionEmitter.minParticleScale = 0.5;
     explosionEmitter.maxParticleScale = 2;
     explosionEmitter.start(true, 2000, null, 200);
     this.ship.destroy();
     game.time.events.add(Phaser.Timer.SECOND * 2, function(){
          game.state.start("GameOverScreen");
     });
}, this);
```

When the animation of the spaceship out of control is complete, we remove the spaceship using `destroy` method so we will also free some memory and create an explosion of particles, using as particle the same image we used for the smoke.

Remember if you want the emitter to fire particles all at once at the same time like an explosion, the first argument of `start` method must be `true`.

> `minParticleScale` and `maxParticleScale` properties define respectively the minimum and maximum scale to be randomly applied to each particle.

Then we wait two seconds before switching to game over state. I also want you to notice this time we also need to pass the context to `collide` method, that's why we added two new arguments, using the full range of arguments provided by the

method.

> `collide(object1, object2, collideCallback, processCallback, callbackContext)` checks for collision between `object1` and `object2` then calls `collideCallback` in `callbackContext` context if `object1` and `object2` collide.

What about `processCallback`? This function can perform your own additional checks on the two objects that collided – for example you could test for velocity, health, or other game variables – and eventually decide whether to continue with the collision returning `true` then calling `collideCallback` and physically modifying objects body velocities or not to continue with the collision returning `false`. Don't worry about it at the moment as we absolutely want any collision to be deadly.

Invulnerability

It would be nice if once you swipe you could be invulnerable for some time. This will be another customizable feature, placed in a new global variable called `shipInvisibilityTime` containing the amount in milliseconds of invulnerability.

```
var game;
// same as before
var shipInvisibilityTime = 1000;
```

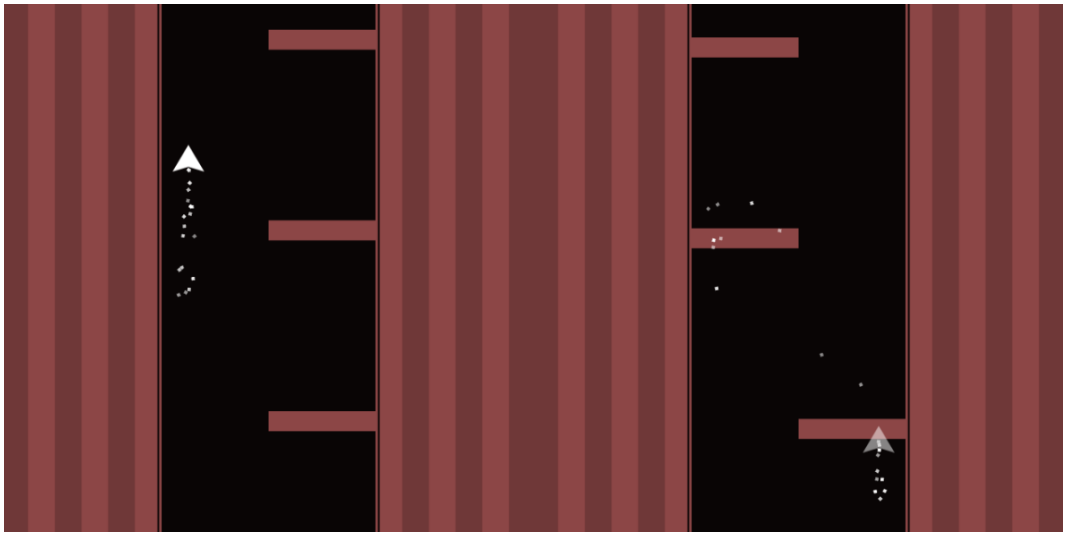
The idea is to make the spaceship semi transparent when it's invulnerable, so we will add a condition in the main `if` in `update` method in `playGame` object, so collisions will be checked only if the spaceship isn't already destroyed and is fully opaque.

```
update: function(){
     // same as before
     if(!this.ship.destroyed && this.ship.alpha == 1){
          // same as before
     }
}
```

We are going to set the alpha when the player swipes, so there are a couple of lines to add to `restartShip` method, setting spaceship alpha to 0.5 (half transparent) and launching a tween which will set the alpha back to 1 (fully opaque) in `shipInvisibilityTime` milliseconds, using a “bounce in” easing.

```
restartShip: function(){
     this.ship.canSwipe = false;
     this.verticalTween.stop();
     this.ship.alpha = 0.5;
     this.verticalTween = game.add.tween(this.ship).to({
          y: 860
     }, 100, Phaser.Easing.Linear.None, true);
     this.verticalTween.onComplete.add(function(){
          this.verticalTween = game.add.tween(this.ship).to({
               y: 0
          }, shipVerticalSpeed, Phaser.Easing.Linear.None, true);
          var alphaTween = game.add.tween(this.ship).to({
               alpha: 1
          }, shipInvisibilityTime, Phaser.Easing.Bounce.In, true);
     }, this)
}
```

Now launch the game, and swipe:



Your spaceship will be flashing and semi transparent and invulnerable for a while.

The flash effect is given by the “bounce in” easing. And this was the last feature to add to ship movement.

Swiping could have saved your spaceship when you were about to crash into a barrier, but most of the times once your ship was back to the bottom of the tunnel it crashed anyway into a barrier because barriers kept moving and you would often find one of them directly in front of your spaceship.

Some fixes here and there

In the making of any game, you will often find yourself in front of a working prototype which makes you really proud of it. Until you find a bug. And another. And another one.

Don't worry. It's perfectly normal. A great percentage of time spent in game developing is used to fix bugs.

There is only one reason why your game does not have bugs: you did not test it.

So my advice is: play the game! Play once, twice, more and more, do unusual things, you'll never know what players will try to do. But your game should work properly no mater what they do.

I am showing you three little bugs in the game, be honest with yourself and think "was I aware of them?"

Bug number one: your spaceship explodes when hitting a barrier only if the ships hits frontally the barrier. If you hit the barrier with the side of the spaceship while moving horizontally through the tunnel, you won't get destroyed.

To fix this, we have to change the arguments of collision detection method, setting `collideCallback` to `null` and `processCallback` to the function previously bound to `collideCallback`.

```
game.physics.arcade.collide(this.ship, this.barrierGroup, null, function(s, b){
     // same as before
}, this)
```

To be honest, I did not find a real reason for this kind of behavior, and I think it's a glitch of Phaser itself which will be fixed soon.

Bug number two: you can restart the spaceship – that is move it back down – even if it already collided into a barrier and is about to explode, and you can also restart it while it's still invisible.

To fix this, the whole code of `restartShip` method should be executed only if the spaceship isn't going to be destroyed and is completely opaque.

```
restartShip: function(){
     if(!this.ship.destroyed && this.ship.alpha == 1){
          // same as before
     }
}
```

Bug number three: once the spaceship hits a barrier, the barrier decreases its speed or even stops moving.

This happens because in a physics world when two rigid bodies collide their forces change.

Unfortunately this breaks the illusion that the spaceship is actually traveling in our fake endless tunnel, so we have to prevent it adding `immovable` property to barrier physics body.

```
Barrier = function (game, speed, tintColor) {
     var positions = [(game.width - tunnelWidth) / 2, (game.width + tunnelWidth) /
       2];
     var position = game.rnd.between(0, 1);
     Phaser.Sprite.call(this, game, positions[position], -100, "barrier");
     var cropRect = new Phaser.Rectangle(0, 0, tunnelWidth / 2, 24);
     this.crop(cropRect);
     game.physics.enable(this, Phaser.Physics.ARCADE);
     this.anchor.set(position, 0.5);
     this.tint = tintColor;
     this.body.immovable = true;
     this.body.velocity.y = speed;
     this.placeBarrier = true;
};
```

Launch the game, and the barrier won't be affected by collisions.

> A body with `immovable` property set to `true` will not receive any impacts from other bodies.

Now all the bugs are fixed, but remember to test properly your game. Always.

Increasing difficulty

We don't want players to abuse swipe feature, so at each swipe we will increase barriers speed. To make this feature customizable, let's add a new global variable:

```
var game;
var bgColors = [0xF16745, 0xFFC65D, 0x7BC8A4, 0x4CC3D9, 0x93648D, 0x7c786a,
       0x588c73, 0x8c4646, 0x2a5b84, 0x73503c];
var tunnelWidth = 256;
var shipHorizontalSpeed = 100;
var shipMoveDelay = 0;
var shipVerticalSpeed = 15000;
var swipeDistance = 10;
var barrierSpeed = 280;
var barrierGap = 120;
var shipInvisibilityTime = 1000;
var barrierIncreaseSpeed = 1.1;
```

`barrierIncreaseSpeed` is the number which will multiply current barrier speed. At each swipe, speed will be multiplied by `1.1`.

This way once the player swipes, starting from next barrier each barrier will have a higher speed, but what about existing barriers?

They keep the same speed they had when they were created. We cannot permit it, because in the real world when a spaceship increases its speed, everything standing in front of it seems to move faster towards it.

The problem is our spaceship does not move – remember? It's a fake runner – so we have to increase the speed of existing barriers too.

We can do it with four lines in `restartShip` method:

```
restartShip: function(){
     if(!this.ship.destroyed && this.ship.alpha == 1){
          barrierSpeed *= barrierIncreaseSpeed;
          for(var i = 0; i < this.barrierGroup.length; i++){
               this.barrierGroup.getChildAt(i).body.velocity.y = barrierSpeed;
          }
          this.ship.canSwipe = false;
          // same as before
     }
}
```

First we increase `barrierSpeed` value multiplying it by `barrierIncreaseSpeed`.

Then since we know all barriers are in `barrierGroup` group, and there are only barriers in `barrierGroup`, we loop through all the children in `barrierGroup` group and set their bodies `y` velocity to the new `barrierSpeed` value.

> `length` property of a group returns the number of children in the group.

Try to run the game and swipe, and you will see your spaceship increase its speed.

Busted! You will actually see the barriers increase their speed.

Obviously the spaceship did not actually increase its speed, just don't tell it to anybody we looped through all barriers and increased their speed.

> `getChildAt(i)` method of a group returns the `i`-th child of a group. You get the first child with `getChildAt(0)`.

The game was challenging, but too permissive when the player swiped to move the spaceship back down. Now each swipe will make the game harder.

Everything is ready to make you score a lot of points.

Scoring

This will be one of the most difficult steps as we are going to introduce a lot of new concepts.

We need to show score segments along the tunnel, and in order to do it with the highest level of customization possible we need to introduce bitmap fonts, so unfortunately a bit of boring theory is needed.

All the text displayed on your browser is rendered using a font file which contains all the information necessary to draw the shape of each character, no matter the size and scale.

When you print a string on the screen, each character is scaled and rendered, leading to two problems:

First, the process of scaling and rendering characters is quite CPU intensive, especially if you need to add run-time effects like outlines or shadows.

Second, if you don't use a common font it's very likely most players won't have that font installed on their computers or mobile devices, so they won't be able to properly render it and a default font will be used instead.

This is where bitmap fonts come into play.

Basically, a bitmap font is an image file containing all the characters we need and a control file with the coordinates of each character in the image.

Now each character can be pre-rendered using multiple effects, loaded as an image, and placed to the screen using very little resources.

The drawback is each bitmap font file can contain only one font size, but it won't be a problem in our case

End of the boring theory, let's dive into coding again: we need two new global variables to define score sectors.

`scoreHeight` will be the height of each score sector in pixels, and `scoreSegments` is an array containing the value of each score sector, from the highest to the lowest.

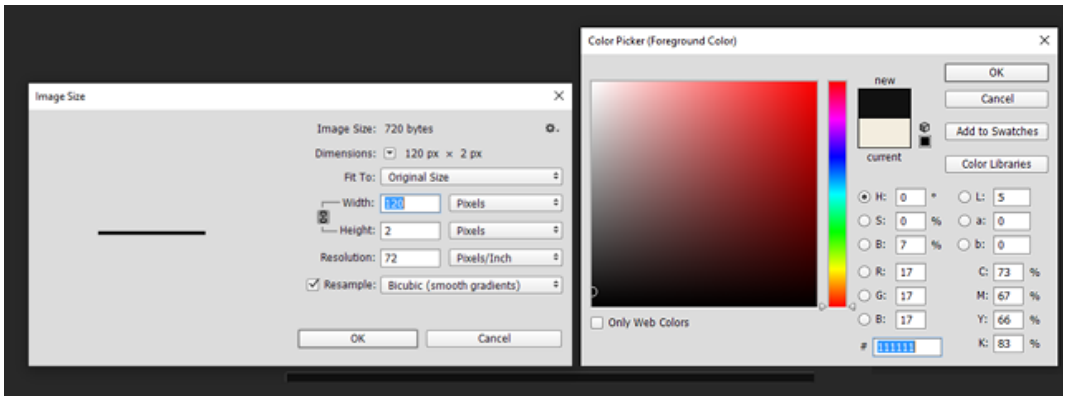
```
var game;
// same as before
var scoreHeight = 100;
var scoreSegments = [100, 50, 25, 10, 5, 2, 1];
```

In the example, I am using seven score sectors, each 100 pixels tall.

The highest score sector, the hardest to reach, will give you 100 points, the lowest score sector will give you only one point, and so on.

We need a new image called `separator` to delimit each score sector boundaries, so we are creating a 120x2 rectangle filled with the dark gray we used for the tunnel and saving it into `sprites` folder.

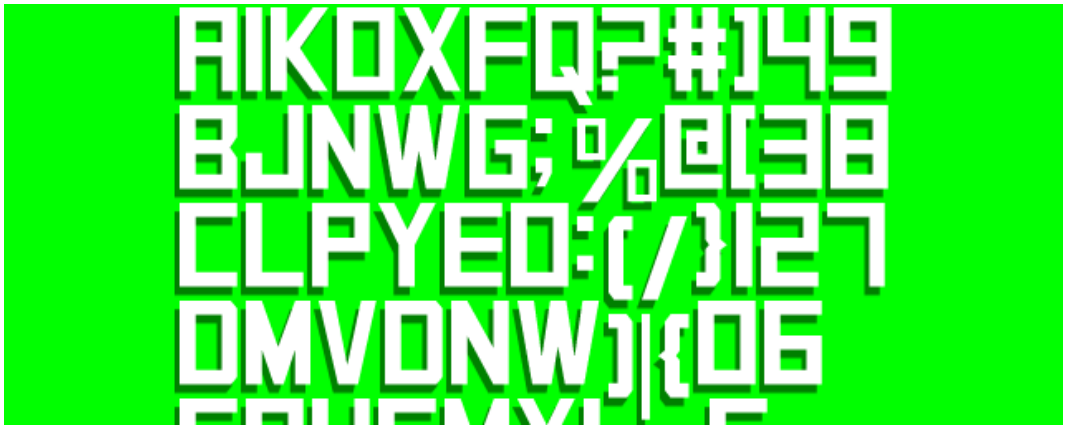
Look at our separator: it's dark gray but we will tint it too according to game randomly chosen color.



Now it's time to create the bitmap font. There is a wide choice of tools to generate bitmap fonts, but the one I prefer – and actually use – is **Littera bitmap font generator** (http://kvazars.com/littera/), a free web application to generate bitmap fonts with all options you need.

It's very easy to use and when you export the font you can keep the default settings so you will get two files: a `png` file which is the image containing all characters and a `fnt` file containing font information.

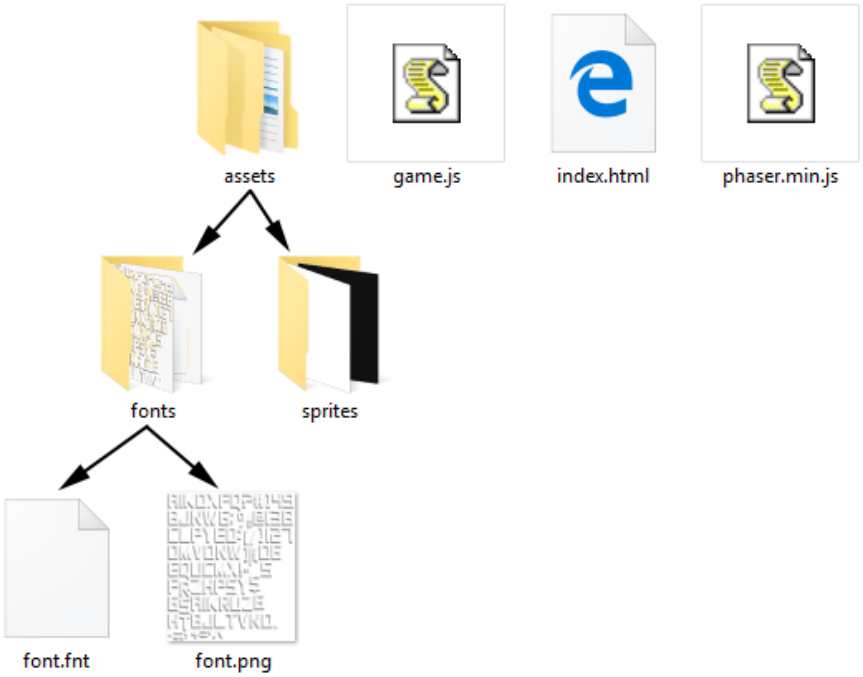
I used a square font with size 72 and applied a light shadow, here it is an excerpt of the PNG file Littera generated on a green background:



The actual image has a transparent background. Remember to include all letters, numbers and symbol you plan to use in your game.

Both the `png` image and the `fnt` file will be placed in a new folder called `fonts`

which you will create inside `assets` folder, this way:



As usual before using assets we need to preload them.

There's a couple of new lines to add to `preload` method of `preload` object:

```
preload: function(){
     var loadingBar = this.add.sprite(game.width / 2, game.height / 2, "loading");
     loadingBar.anchor.setTo(0.5);
     game.load.setPreloadSprite(loadingBar);
     game.load.image("title", "assets/sprites/title.png");
     game.load.image("playbutton", "assets/sprites/playbutton.png");
     game.load.image("backsplash", "assets/sprites/backsplash.png");
     game.load.image("tunnelbg", "assets/sprites/tunnelbg.png");
     game.load.image("wall", "assets/sprites/wall.png");
     game.load.image("ship", "assets/sprites/ship.png");
     game.load.image("smoke", "assets/sprites/smoke.png");
     game.load.image("barrier", "assets/sprites/barrier.png");
     game.load.image("separator", "assets/sprites/separator.png");
     game.load.bitmapFont("font", "assets/fonts/font.png",
        "assets/fonts/font.fnt");
}
```

And now we are ready to use our bitmap font.

`load.bitmapFont(key, textureURL, xmlURL)` adds new bitmap font loading request, giving it unique `key` name and looking for `textureURL` image file and `xmlURL` data file.

We are using both bitmap font and the separator image in `create` method of `playGame` object:

```
create: function(){
          // same as before
          rightWallBG.tileScale.x = -1;
          for(var i = 1; i <= scoreSegments.length; i++){
               var leftSeparator = game.add.sprite((game.width - tunnelWidth) / 2,
                      scoreHeight * i, "separator");
               leftSeparator.tint = tintColor;
               leftSeparator.anchor.set(1, 0)
               var rightSeparator = game.add.sprite((game.width + tunnelWidth) /
                      2, scoreHeight * i, "separator");
               rightSeparator.tint = tintColor;
               var posX = (game.width - tunnelWidth) / 2 - leftSeparator.width /
                      2;
               if(i % 2 == 0){
                    posX = (game.width + tunnelWidth) / 2 + leftSeparator.width /
                      2;
               }
               game.add.bitmapText(posX, scoreHeight * (i - 1) + scoreHeight / 2 -
                      18 , "font", scoreSegments[i - 1].toString(), 36).anchor.x =
                      0.5;
          }
          this.barrierGroup = game.add.group();
          //same as before
     }
```

It seems a lot to do but you'll see it's just a couple of concepts, let's see the new lines in detail:

```
for(var i = 1; i <= scoreSegments.length; i++){
     // code
}
```

This loop iterates through all score segments

```
var leftSeparator = game.add.sprite((game.width - tunnelWidth) / 2,
       scoreHeight * i, "separator");
leftSeparator.tint = tintColor;
leftSeparator.anchor.set(1, 0)
```

`leftSeparator` is the separator image we want to add to the bottom of each score segment. As the name suggests, it's the left separator so its horizontal position is half the difference between game width and tunnel width, which is the exact point where left wall ends.

The vertical position is determined solely by the height of the score sector.

Current tint color is assigned to the image, and we set its anchor point to the right top pixel.

This way the separator will start where the tunnel begins, and will continue to the right.

```
var rightSeparator = game.add.sprite((game.width + tunnelWidth) /
       2, scoreHeight * i, "separator");
rightSeparator.tint = tintColor;
```

Same thing goes with the right separator, it's just this time it has been placed at half the sum of game with and tunnel width.

Everything else remains the same, with the exception of the anchor point which should be the left top pixel. Since this is the default value, there's no need to specify its value.

```
var posX = (game.width - tunnelWidth) / 2 - leftSeparator.width / 2;
```

`posX` is a temporary variable which defines the horizontal position of the bitmap text we are about to write. In this case it's in the middle of the left separator.

```
if(i % 2 == 0){
     posX = (game.width + tunnelWidth) / 2 + leftSeparator.width / 2;
}
```

If `i` – current loop index – can be divided by two, then `posX` changes to be in the middle of the right separator.

This way at each loop iteration, `posX` will switch from the middle of the left

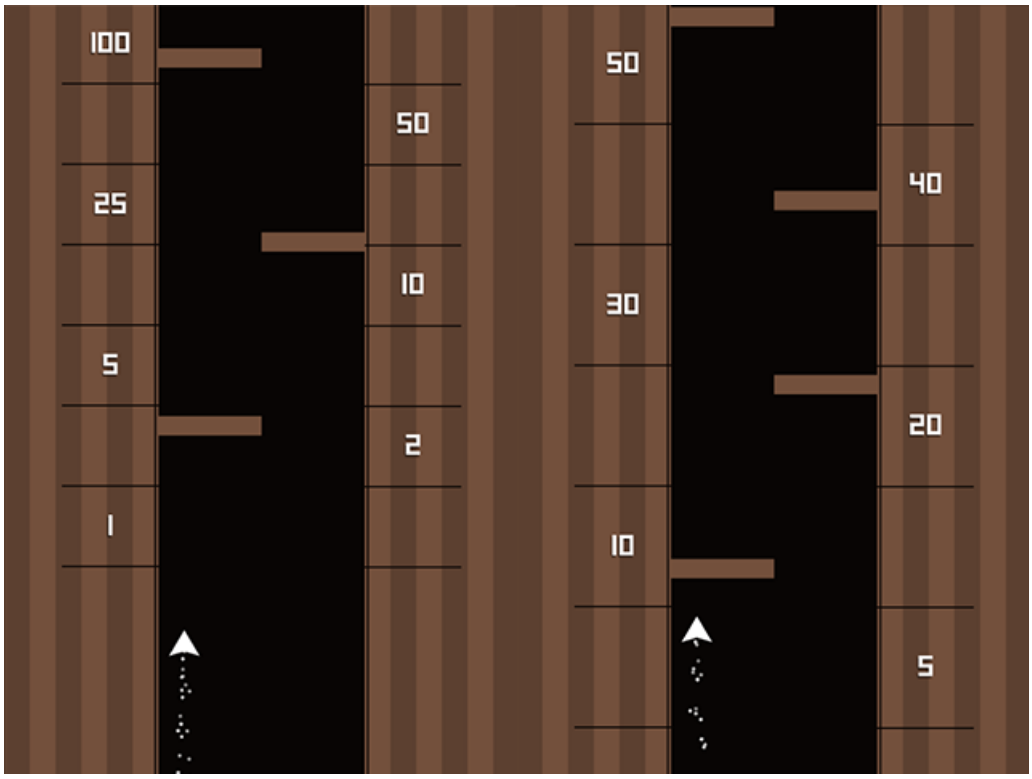
separator to the middle of the right separator.

```
game.add.bitmapText(posX, scoreHeight * (i - 1) + scoreHeight / 2 - 18 , "font",
        scoreSegments[i - 1].toString(), 36).anchor.x = 0.5;
```

Everything is ready to add the bitmap text, which is not that different than adding a sprite. Also notice how the anchor point can be set appending the property at the end of the line.

`add.bitmapText(x, y, font, text, size)` adds `text` string written with a `size` points font `font` at coordinates `x`, `y`.

Launch the game, and see how bitmap fonts are placed to the canvas to display the values for each score sector.



As you can see, everything is made to give you the most customization. Look how left and right tunnels differ for score sectors heights and values.

Highlighting score sector

In the original game when the spaceship enters a score sector, it highlights it.

We'll make the same effect using a tiled sprite with the image created for smoke.

You should already know how to create and add a tiled sprite so here are the lines we are going to add to `create` method in `playGame` object:

```
create: function(){
     // same as before
     this.highlightBar = game.add.tileSprite(game.width / 2, 0, tunnelWidth,
        scoreHeight, "smoke");
     this.highlightBar.anchor.set(0.5, 0);
     this.highlightBar.alpha = 0.1;
     this.highlightBar.visible = false;
}
```

`highlightBar` is the name of the tiled sprite, which `tunnelWidth` wide and `scoreHeight` tall. Have a look at `visible` property. The bar is not visible at the moment.

> `visible` property of a sprite sets its visibility. `true` means the sprite is visible, `false` means the sprite is not visible.

The core of this concept can be found in `update` method of `playGame` object.

First we see if the spaceship is inside a score zone, then we determine which score zone and finally we place the highlight bar accordingly and turn it visible.

```
update: function(){
     // same as before
     if(!this.ship.destroyed && this.ship.alpha == 1){
          if(this.ship.y < scoreHeight * scoreSegments.length){
               this.highlightBar.visible = true;
               var row = Math.floor(this.ship.y / scoreHeight);
               this.highlightBar.y = row * scoreHeight;
          }
          game.physics.arcade.collide(this.ship, this.barrierGroup, null,
               function(s, b){
               this.highlightBar.visible = false;
               // same as before
          }, this)
     }
}
```

Let's examine these lines in detail:

```
if(this.ship.y < scoreHeight * scoreSegments.length){
     // content
}
```

The spaceship is in the score zone if its vertical coordinate is less than `scoreHeight` multiplied by the number of score segments.

```
this.highlightBar.visible = true;
```

We still do not know where to place the highlight bar, but now it will be visible for sure.

```
var row = Math.floor(this.ship.y / scoreHeight);
```

This is how we determine which row the spaceship is traveling in.

The integer part of spaceship vertical position divided by `scoreHeight`.

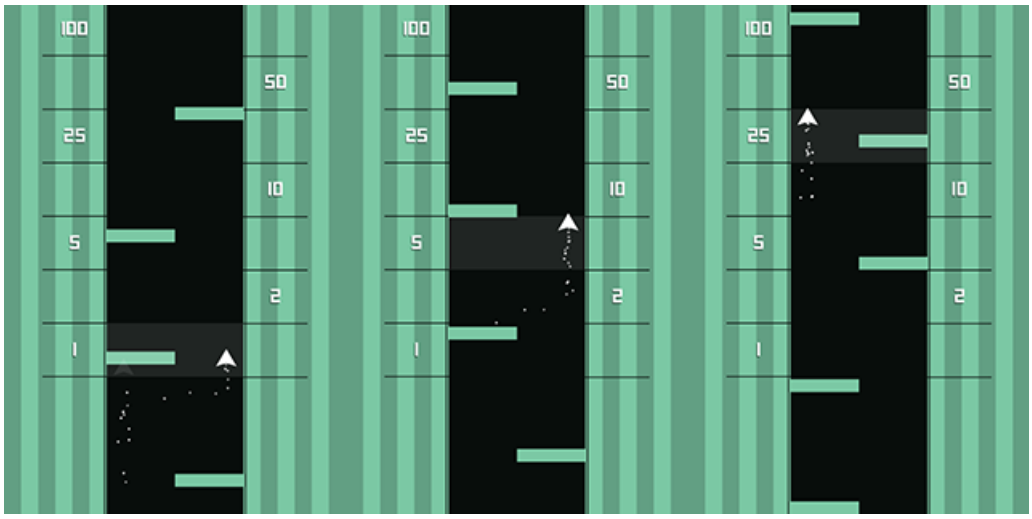
```
this.highlightBar.y = row * scoreHeight;
```

And now we place the highlight bar, multiplying `row` by `scoreHeight`.

When the spaceship collides with a barrier, we turn `highlightBar` invisible, as well as when we move back the ship down, so we need to add a new line to `restartShip` method:

```
restartShip: function(){
     this.highlightBar.visible = false;
     // same as before
}
```

And now when you test the game, you will see the spaceship highlight score segments as it travels towards the top of the screen, but only when the spaceship enters a score segment.



Crash against a barrier or swipe and the highlight bar will disappear.

Showing score

Everything is ready to show score as the spaceship flies in the tunnel, so let's create a global variable to keep track of the score. Guess its name: `score`.

```
var game;
var score;
// same as before
```

In `create` method of `playGame` object, first we have to set `score` to zero, then we place a bitmap text near the bottom of the screen showing the score.

```
create: function(){
     score = 0;
     var tintColor = bgColors[game.rnd.between(0, bgColors.length - 1)]
     // same as before
     for(var i = 1; i <= scoreSegments.length; i++){
          // same as before
     }
     this.scoreText = game.add.bitmapText(20, game.height - 90 , "font", "0", 48);
     this.barrierGroup = game.add.group();
     // same as before
     game.time.events.loop(250, this.updateScore, this);
}
```

`scoreText` property will be used to show player score.

We want to check every ¼ seconds which height we reached with the spaceship and increase score accordingly.

> `time.events.loop(delay, callback, callbackContext)` adds a looped event that will repeat forever and will fire after the given amount of `delay` milliseconds has passed, calling `callback` function in `callbackContext` context.

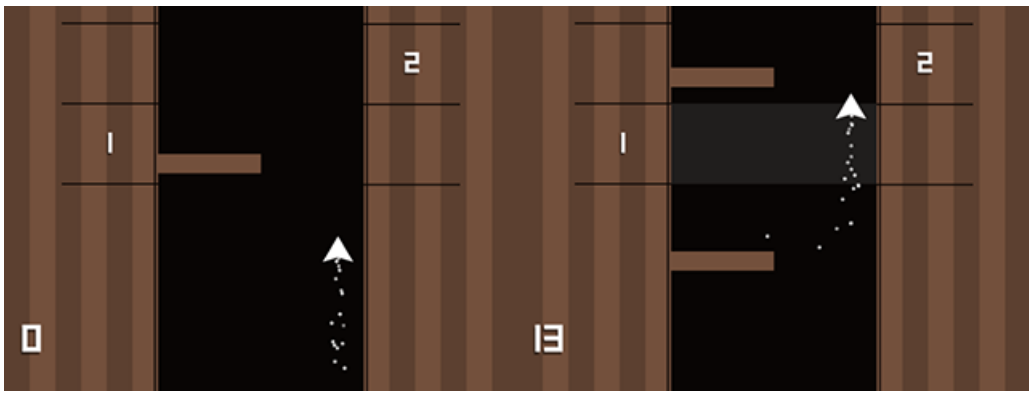
Every 250 milliseconds `updateScore` method will be called.

So we create an `updateScore` method to `playGame` object, and the code won't differ that much from what we wrote before to highlight score sector:

```
updateScore: function(){
     if(this.ship.alpha == 1 && !this.ship.destroyed){
          if(this.ship.y < scoreHeight * scoreSegments.length){
               var row = Math.floor(this.ship.y / scoreHeight);
               score += scoreSegments[row];
               this.scoreText.text = score.toString();
          }
     }
}
```

Once we determined the row in which the spaceship is flying, we simply add to `score` the corresponding `scoreSegments` array value, then update `scoreText` acting on its `text` property.

Launch the game and try to enter in some scoring sectors to see how your score increases.



Flying high is risky but you will get a lot of points. But there is another way to make points.

Adding friendly barriers

Not all barriers are deadly. To add a twist to the game, sometimes a friendly barrier will appear.

Friendly barriers can be recognized because they are white. If you hit a friendly barrier, you get a bonus score.

Moreover, once you'll get this concept – determine which kind of barrier the spaceship hits – you can create a lot of bonus effects, such as slow down the ship (busted! Slow down the barriers), give some extra seconds of invisibility, gain virtual currency which can be used in some kind of shop, and so on.

This would go beyond the scope of this book, anyway you are about to learn the main concept.

Let's start by adding a new global variable called `friendlyBarRatio` which tell us how frequent are friendly barriers.

```
var game;
var score;
var bgColors = [0xF16745, 0xFFC65D, 0x7BC8A4, 0x4CC3D9, 0x93648D, 0x7c786a,
       0x588c73, 0x8c4646, 0x2a5b84, 0x73503c];
var tunnelWidth = 256;
var shipHorizontalSpeed = 100;
var shipMoveDelay = 0;
var shipVerticalSpeed = 15000;
var swipeDistance = 10;
var barrierSpeed = 280;
var barrierGap = 120;
var shipInvisibilityTime = 1000;
var barrierIncreaseSpeed = 1.1;
var scoreHeight = 100;
var scoreSegments = [100, 50, 25, 10, 5, 2, 1];
var friendlyBarRatio = 2;
```

`friendlyBarRatio` is an integer number, and the lower the number, the more frequent friendly barriers will appear.

You can see how to create friendly barriers with some little changes to `Barrier`

class, nothing you can't handle:

```
Barrier = function (game, speed, tintColor) {
     var positions = [(game.width - tunnelWidth) / 2, (game.width + tunnelWidth) /
       2];
     var position = game.rnd.between(0, 1);
     Phaser.Sprite.call(this, game, positions[position], -100, "barrier");
     var cropRect = new Phaser.Rectangle(0, 0, tunnelWidth / 2, 24);
     this.crop(cropRect);
     game.physics.enable(this, Phaser.Physics.ARCADE);
     this.anchor.set(position, 0.5);
     this.levelTint = tintColor;
     if(game.rnd.between(0, friendlyBarRatio)!=0){
          this.tint = tintColor;
          this.friendly = false;
     }
     else{
          this.friendly = true;
     }
     this.body.immovable = true;
     this.body.velocity.y = speed;
     this.placeBarrier = true;
};
```

Let's see these changes in detail:

```
this.levelTint = tintColor;
```

First we create another property called `levelTint` where we save `tintColor` argument for later use.

```
if(game.rnd.between(0, friendlyBarRatio)!=0){
     this.tint = tintColor;
     this.friendly = false;
}
```

This is where `friendlyBarRatio` comes into play. We draw a random number from zero to `friendlyBarRatio` and if the number is different than zero then we apply the barrier the tint color, just like we did before. We also add a `friendly` property set to `false`. This barrier is not friendly. It's deadly.

```
else{
     this.friendly = true;
}
```

If the random number is zero, we just set `friendly` property to `true` and we don't apply tint color, so the barrier will be white and friendly.

In `update` method, we can't pass anymore `tint` property to `addBarrier` method or any barrier generated by a friendly – without tint – barrier, will remain white.

That's why we saved `levelTint` property. This way we can pass the proper tint value no matter whether the current barrier has tint or not.

```
Barrier.prototype.update = function(){
     if(this.placeBarrier && this.y > barrierGap){
          this.placeBarrier = false;
          playGame.prototype.addBarrier(this.parent, this.levelTint);
     }
     if(this.y > game.height){
          this.destroy();
     }
}
```

With `friendlyBarRatio` set to `2`, roughly 1 out of 3 barriers will be friendly. I suggest to set it to `14` to have 1/15 friendly barriers.

Finally we have to decide what to do when the spaceship hits a friendly barrier.

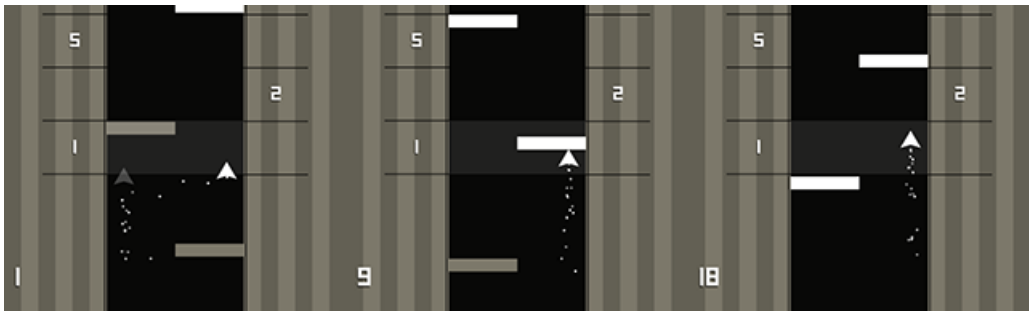
There are some changes to do to collision routine to add the case we hit a friendly barrier.

```
game.physics.arcade.collide(this.ship, this.barrierGroup, null, function(s, b){
     if(!b.friendly){
          // same as before
     }
     else{
          if(b.alpha == 1){
               var barrierTween = game.add.tween(b).to({
                    alpha:0
               }, 200, Phaser.Easing.Bounce.Out, true);
               if(this.ship.y < scoreHeight * scoreSegments.length){
                    var row = Math.floor(this.ship.y / scoreHeight);
                    score += scoreSegments[row] * 5;
                    this.scoreText.text = score.toString();
               }
          }
     }
}, this)
```

The idea is to keep unfriendly barriers lethal, while destroying friendly barriers

adds a bonus score according to the height of the spaceship.

Run the game and chase for white barriers, you will see your score increases as barriers get destroyed.



Have a look at the magic behind it:

```
if(!b.friendly){
     // same as before
}
```

Nothing changes if `friendly` property is `false`. Your spaceship explodes.

The `else` statement contains the code to run when `friendly` property is `true`.

```
if(b.alpha == 1){
     var barrierTween = game.add.tween(b).to({
          alpha:0
     }, 200, Phaser.Easing.Bounce.Out, true);
     if(this.ship.y < scoreHeight * scoreSegments.length){
          var row = Math.floor(this.ship.y / scoreHeight);
          score += scoreSegments[row] * 5;
          this.scoreText.text = score.toString();
     }
}
```

When the spaceship hits a friendly barrier, we use a tween on its alpha to make it disappear. That's why the first thing we do is checking for barrier alpha to be 1. To see if this is the first time we hit the barrier.

```
var barrierTween = game.add.tween(b).to({
     alpha:0
}, 200, Phaser.Easing.Bounce.Out, true);
```

Then we tween barrier alpha to zero

```
if(this.ship.y < scoreHeight * scoreSegments.length){
     var row = Math.floor(this.ship.y / scoreHeight);
     score += scoreSegments[row] * 5;
     this.scoreText.text = score.toString();
}
```

Finally we check if the spaceship is in the score zone, and multiply the score zone by five – you can change it as you want – then update the score.

I just would like you to see how we added a new game feature without introducing new concepts, mostly copying and pasting code we wrote before.

This means you are slowing mastering Phaser, as you don't need to learn new stuff each time you want to change something in your game.

Creation of game over state

It's been a long time we have a game over state which has been left empty.

Now we are going to add content to it, rewriting the entire `gameOverScreen` object.

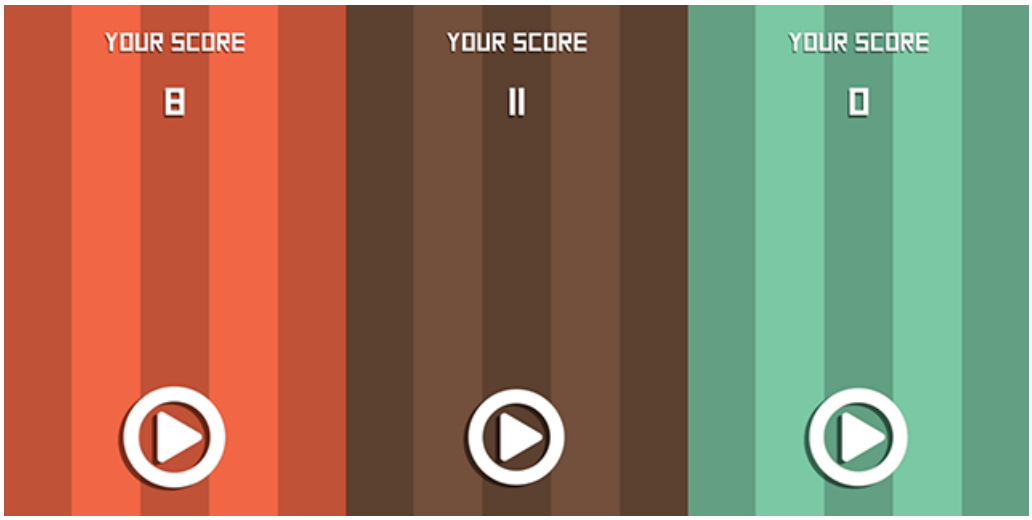
```
var gameOverScreen = function(game){};
gameOverScreen.prototype = {
     create: function(){
          var titleBG = game.add.tileSprite(0, 0, game.width, game.height,
               "backsplash");
          titleBG.tint = bgColors[game.rnd.between(0, bgColors.length - 1)];
          game.add.bitmapText(game.width / 2, 50 , "font", "Your score",
               48).anchor.x = 0.5;
          game.add.bitmapText(game.width / 2, 150 , "font", score.toString(),
               72).anchor.x = 0.5;
          var playButton = game.add.button(game.width / 2, game.height - 150,
               "playbutton", this.startGame);
          playButton.anchor.set(0.5);
          var tween = game.add.tween(playButton).to({
               width: 220,
               height:220
          }, 1500, "Linear", true, 0, -1);
          tween.yoyo(true);
     },
     startGame: function(){
          game.state.start("PlayGame");
     }
}
```

It's very similar to the title screen and there's nothing new in it.

We place the background, assign it a random tint color, display player score and add the play button with its pulse tween.

Once the player presses the button, the game restarts calling `PlayGame` state.

Run the game, crash into a barrel, see the game over screen and you can restart the game by pressing the button.



What about showing the best score so far?

Saving high scores

When playing a game, you will quickly realize there is no point in making a great score if you can't save it and try to beat it later.

We are going to cover how to save your best score, and keep it saved even if you close the browser window or turn off your computer or device.

All modern browsers support **local storage**, a way used by web pages to locally store data in a key/value notation.

The information you save will continue to be stored even when you shut down your device and can be read every time you launch your game.

This is exactly what we need.

Let's create two new variables:

```
var game;
var score;
var savedData;
var bgColors = [0xF16745, 0xFFC65D, 0x7BC8A4, 0x4CC3D9, 0x93648D, 0x7c786a,
       0x588c73, 0x8c4646, 0x2a5b84, 0x73503c];
var tunnelWidth = 256;
var shipHorizontalSpeed = 100;
var shipMoveDelay = 0;
var shipVerticalSpeed = 15000;
var swipeDistance = 10;
var barrierSpeed = 280;
var barrierGap = 120;
var shipInvisibilityTime = 1000;
var barrierIncreaseSpeed = 1.1;
var scoreHeight = 100;
var scoreSegments = [100, 50, 25, 10, 5, 2, 1];
var friendlyBarRatio = 10;
var localStorageName = "myrocketgame";
```

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

`savedData` will contain the information we want to save.

We are only saving the best score at the moment, but you can save anything you want: the number of games played, the total time spent playing the game, and so on.

Also, you aren't limited to saving one single value, you can also save arrays such as the best 10 scores.

It's up to you and your creativity.

Before entering the title screen we will check the local storage to see if we already saved something and load it into `savedData` variable.

If nothing has been previously saved, because it's the first time the game is played or we just change `localStorageName` value, we set a default `savedData` content.

Let's add some lines to `create` method of `titleScreen` object:

```
create: function(){
     savedData = localStorage.getItem(localStorageName)==null?
        {score:0}:JSON.parse(localStorage.getItem(localStorageName));
     var titleBG = game.add.tileSprite(0, 0, game.width, game.height,
        "backsplash");
     titleBG.tint = bgColors[game.rnd.between(0, bgColors.length - 1)];
     var title = game.add.image(game.width / 2, 210, "title");
     title.anchor.set(0.5);
     game.add.bitmapText(game.width / 2, 480 , "font", "Best score", 48).anchor.x
        = 0.5;
     game.add.bitmapText(game.width / 2, 530 , "font", savedData.score.toString(),
        72).anchor.x = 0.5;
     var playButton = game.add.button(game.width / 2, game.height - 150,
        "playbutton", this.startGame);
     playButton.anchor.set(0.5);
     var tween = game.add.tween(playButton).to({
          width: 220,
          height:220
     }, 1500, "Linear", true, 0, -1);
     tween.yoyo(true);
},
startGame: function(){
     game.state.start("PlayGame");
}
```

The first line added is the very core of the concept. Let's see it:

```
savedData = localStorage.getItem(localStorageName)==null?
       {score:0}:JSON.parse(localStorage.getItem(localStorageName));
```

No matter the experience programmers have, some of them never use or even did not know the 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`.

If the local storage does not contain any information, we set `savedData` to an object with `score` set to zero.

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

If the local storage already contains information, we load them and parse from

JSON notation.

> `JSON.parse(text)` converts `text` JSON string into a JavaScript object.

Why are we using JSON?

JSON (JavaScript Object Notation) is a lightweight data-interchange format used in cases where data serialization is needed.

It is easy for humans to read and write. It is easy for machines to parse and generate. It is an open standard format which is the one I prefer to transmit data objects consisting of attribute-value pairs.

Explaining JSON and its applications in everyday programming is beyond the scope of this book, but at the moment, for the purpose we have to use it, it's exactly what we are looking for.

No matter if there already was something previously stored in local storage, at the end of this line we have our best score in `savedData.score` so we can show it on the screen with the remaining new lines:

```
game.add.bitmapText(game.width / 2, 480 , "font", "Best score", 48).anchor.x
       = 0.5;
game.add.bitmapText(game.width / 2, 530 , "font", savedData.score.toString(),
       72).anchor.x = 0.5;
```

Since local storage information will be saved each time the game is over, hopefully to update the best score, we are going to load it again when we call `create` method in `playGame` object:

```
create: function(){
     score = 0;
     savedData = localStorage.getItem(localStorageName)==null?
        {score:0}:JSON.parse(localStorage.getItem(localStorageName));
     // same as before
}
```

And obviously `create` method of `gameOverScreen` object has the most changes, because we are showing the best score and also saving it to local storage:

```
create: function(){
     var bestScore = Math.max(score, savedData.score);
     var titleBG = game.add.tileSprite(0, 0, game.width, game.height,
       "backsplash");
     titleBG.tint = bgColors[game.rnd.between(0, bgColors.length - 1)];
     game.add.bitmapText(game.width / 2, 50 , "font", "Your score", 48).anchor.x =
       0.5;
     game.add.bitmapText(game.width / 2, 150 , "font", score.toString(),
       72).anchor.x = 0.5;
     game.add.bitmapText(game.width / 2, 350 , "font", "Best score", 48).anchor.x
       = 0.5;
     game.add.bitmapText(game.width / 2, 450 , "font", bestScore.toString(),
       72).anchor.x = 0.5;
     localStorage.setItem(localStorageName,JSON.stringify({
          score: bestScore
     }));
     var playButton = game.add.button(game.width / 2, game.height - 150,
       "playbutton", this.startGame);
     playButton.anchor.set(0.5);
     var tween = game.add.tween(playButton).to({
          width: 220,
          height:220
     }, 1500, "Linear", true, 0, -1);
     tween.yoyo(true);
}
```

`bestScore` variable is used to get the best score between current score and `savedData.score`, then both scores are shown, finally the best score is saved in the local storage.

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

In this case, JSON string is built starting from an object.

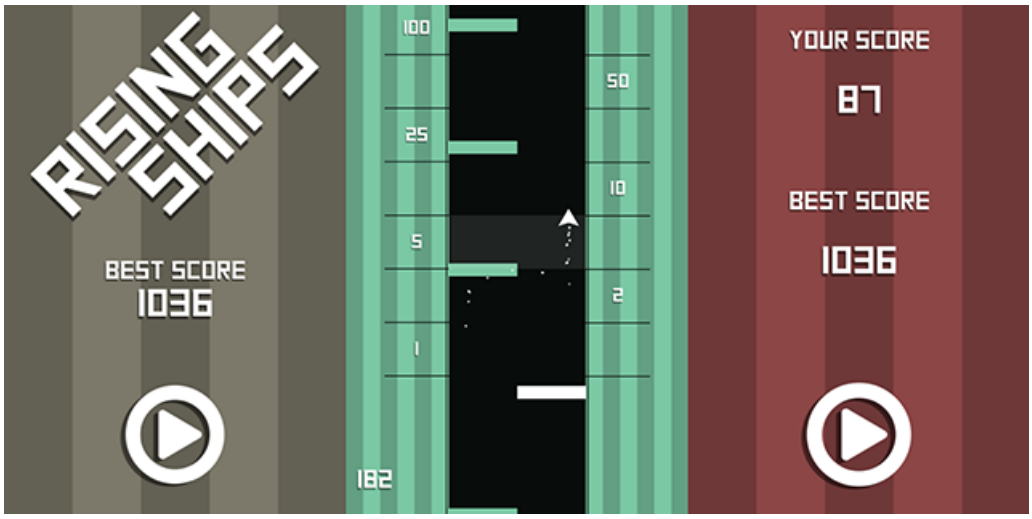
> `JSON.stringify(object)` method converts `object` JavaScript object to a JSON string.

Now run the game, play and try to beat your best scores.

As you will see, the best score is shown both in the title and in the game over screen.

Close the game, close the browser, restart or turn off the device, then turn it on and launch the game again.

Your best score remains saved.



Now your game offers some more challenge, just by saving the high score.

Adapting the game to various resolutions

The game works well at the moment, but since it's made for a 640x960 pixels resolution, it runs with an aspect ratio of 640/960 = 2/3.

This means you will see black bars around the game canvas if you run the game on a device or on a window which aspect ratio is not exactly 2/3.

Look at this picture: it shows the game running at 1024x768 and at 320x640.

Do you see the problem? In the 1024x768 window, with an aspect ratio greater than 2/3, we have vertical black bars, while in the 320x640, with an aspect ratio

smaller than 2/3, we have horizontal black bars.

These black bars are made by the black `body` background of the web page which is not entirely covered by the canvas running the game.

We have to prevent this to happen as we can't talk about “cross-platorm” until every aspect ratio is supported.

Obviously the game has been already developed so we won't change a single line of the game itself, we will just play on colors and proportions to give the player the feeling the game is covering the entire page, although it's a fake effect.

We will use two techniques to get rid of the black bars.

If we have to work with vertical bars, we'll just change `body` background color to match the tint color of the game.

If we have to work with horizontal bars, we have to create a taller game so that it will cover the entire available space, but game elements will perfectly adjust thanks to some choices we took during the creation of the game.

Let's start changing a bit `window.onload` function:

```
window.onload = function() {
     var width = 640;
     var height = 960;
     var windowRatio = window.innerWidth / window.innerHeight;
     if(windowRatio < width / height){
          var height = width / windowRatio;
     }
     game = new Phaser.Game(width, height, Phaser.AUTO, "");
     game.state.add("Boot", boot);
     game.state.add("Preload", preload);
     game.state.add("TitleScreen", titleScreen);
     game.state.add("PlayGame", playGame);
     game.state.add("GameOverScreen", gameOverScreen);
     game.state.start("Boot");
}
```

These changes will handle horizontal bars, let's see them in detail:

```
var width = 640;
var height = 960;
```

Rather than hardcoding game size, we are using two variables to store game width and height.

```
var windowRatio = window.innerWidth / window.innerHeight;
```

We need to know the aspect ratio of the window the game is currently running in, and `windowRatio` contains this information.

> `innerWidth` property returns the inner width of a window's content area.
>
> `innerHeight` property returns the inner height of a window's content area.

Next thing we have to do is to check if `windowRatio` is smaller than our game ratio, causing horizontal bars to appear.

```
if(windowRatio < width / height){
     var height = width / windowRatio;
}
```

How can we prevent horizontal bars to appear? Simply extending the height of the game as the actual game width we want divided by `windowRatio`.

This way we will get a taller game, but it won't be a problem because all game elements will be conveniently placed to the canvas in order to give the same gameplay experience no matter the height of the game.

```
game = new Phaser.Game(width, height, Phaser.AUTO, "");
```

Finally the game is created using `width` and `height` variables.

This will fix horizontal bars issue.

Vertical bars are even easier to fix as we only have to set the background color of the body of the HTML page where the game is running in to the same random tint color of the game.

This is the line we have to add to create method in `titleScreen` object:

```
create: function(){
     savedData = localStorage.getItem(localStorageName)==null?
       {score:0}:JSON.parse(localStorage.getItem(localStorageName));
     var titleBG = game.add.tileSprite(0, 0, game.width, game.height,
       "backsplash");
     titleBG.tint = bgColors[game.rnd.between(0, bgColors.length - 1)];
     document.body.style.background = "#"+titleBG.tint.toString(16);
     var title = game.add.image(game.width / 2, 210, "title");
     title.anchor.set(0.5);
     game.add.bitmapText(game.width / 2, 480 , "font", "Best score", 48).anchor.x
       = 0.5;
     game.add.bitmapText(game.width / 2, 530 , "font", savedData.score.toString(),
       72).anchor.x = 0.5;
     var playButton = game.add.button(game.width / 2, game.height - 150,
       "playbutton", this.startGame);
     playButton.anchor.set(0.5);
     var tween = game.add.tween(playButton).to({
          width: 220,
          height:220
     }, 1500, "Linear", true, 0, -1);
     tween.yoyo(true);
}
```

This is how we change the background color outside the game.

> `document.body.style.background` sets the style of the background of a document.

Since background color has to be set as hexadecimal string, we convert `tint` property of `titleBG` sprite to a hexadecimal value, starting the string with `#`. Don't worry as they are just operations on strings.

> `toString(radix)` converts a number to a string with `radix` base to use for representing a numeric value. Must be an integer between 2 and 36.

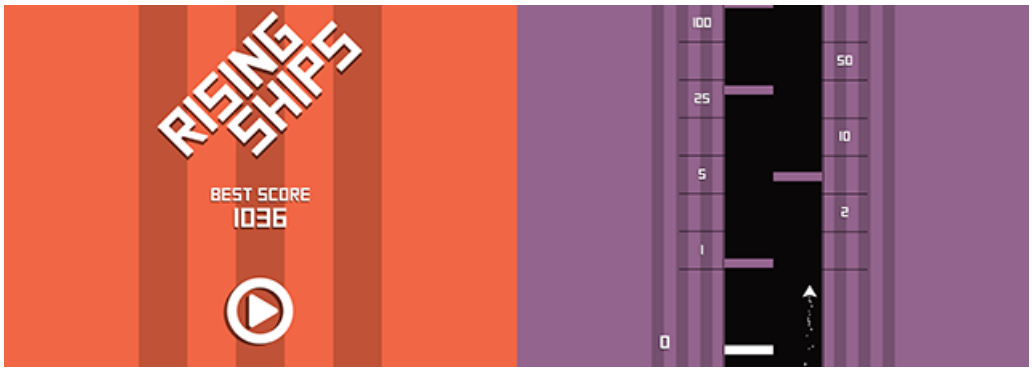
Same thing applies to `create` method in `playGame` object:

```
create: function(){
     score = 0;
     savedData = localStorage.getItem(localStorageName)==null?
       {score:0}:JSON.parse(localStorage.getItem(localStorageName));
     var tintColor = bgColors[game.rnd.between(0, bgColors.length - 1)];
     document.body.style.background = "#"+tintColor.toString(16);
     // same as before
}
```

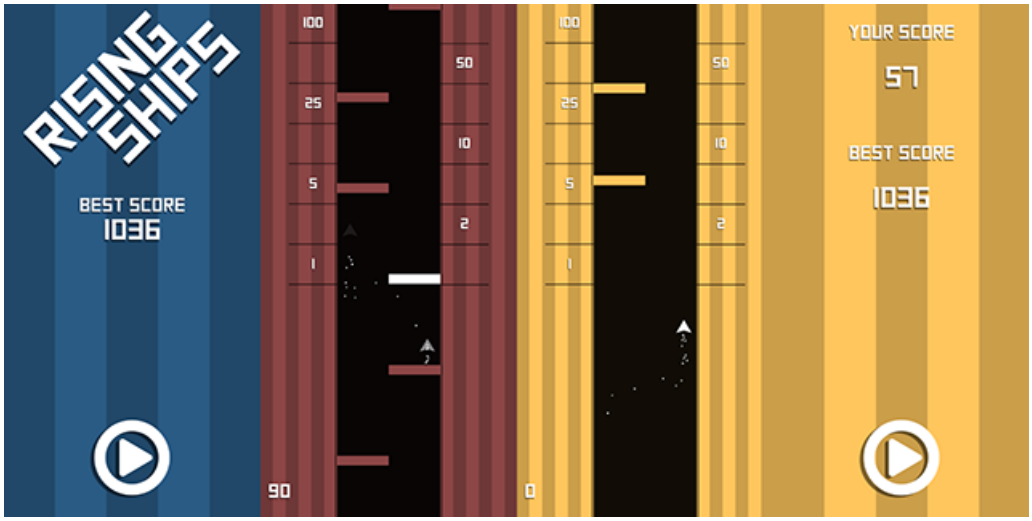
And this works well in `create` method of `gameOverScreen` object too:

```
create: function(){
     bestScore = Math.max(score, savedData.score);
     var titleBG = game.add.tileSprite(0, 0, game.width, game.height,
       "backsplash");
     titleBG.tint = bgColors[game.rnd.between(0, bgColors.length - 1)];
     document.body.style.background = "#"+titleBG.tint.toString(16);
     // same as before
}
```

Now test the game in a landscape window, and see how everything works well. Despite there still are vertical bars, giving them the same color of the game makes them part of the game itself.



And what happens when we have a tall portrait?



Horizontal bars are gone since the height of the game has been increased to match

with the height of the window, and sensible elements such as play button and score are placed close to the bottom giving players the feeling the game was written with their devices in mind.

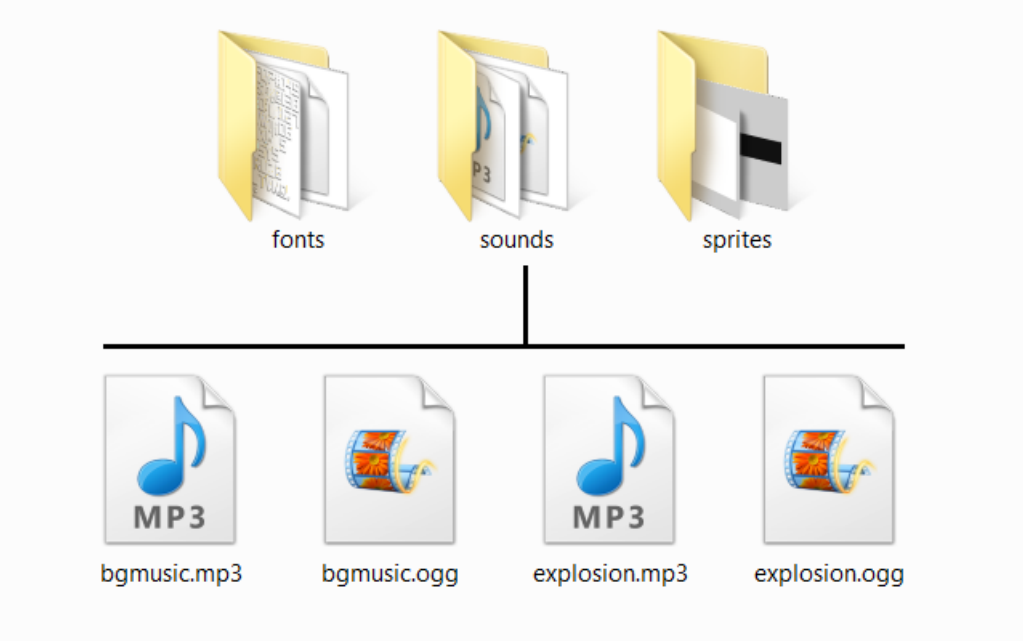
Other game actors, like the spaceship, maintain their absolute positioning, or the spaceship would have to travel a lot more before entering the scoring zone.

And with this couple of tricks, we made the game universal.

Adding sounds

To make the game more appealing, we are adding some sounds. You can add as many sounds as you want, but since it's just a repetitive task, in this game we are going to add only two sounds: a music loop to be played when the game starts, and an explosion sound effect to be played when the spaceship hits a barrier.

To keep game files organized, inside `assets` folder we need to create a new folder called `sounds`, which will contain our sound files, this way:



As you can see I added two sounds, in two different formats: `mp3` and `ogg`. As file names suggest, we have a background music and an explosion sound effect.

Why did I use two sound formats? It's a compatibility matter: not all browsers are capable to reproduce all kind of sound files. Including both `mp3` and `ogg` should grant the best device and browser coverage.

Preloading sounds is not different than preloading images, as you can see in `preload` method in `Preload` object:

```
preload: function(){
     var loadingBar = this.add.sprite(game.width / 2, game.height / 2, "loading");
     loadingBar.anchor.setTo(0.5);
     game.load.setPreloadSprite(loadingBar);
     game.load.image("title", "assets/sprites/title.png");
     game.load.image("playbutton", "assets/sprites/playbutton.png");
     game.load.image("backsplash", "assets/sprites/backsplash.png");
     game.load.image("tunnelbg", "assets/sprites/tunnelbg.png");
     game.load.image("wall", "assets/sprites/wall.png");
     game.load.image("ship", "assets/sprites/ship.png");
     game.load.image("smoke", "assets/sprites/smoke.png");
     game.load.image("barrier", "assets/sprites/barrier.png");
     game.load.image("separator", "assets/sprites/separator.png");
     game.load.bitmapFont("font", "assets/fonts/font.png",
        "assets/fonts/font.fnt");
     game.load.audio("bgmusic", ["assets/sounds/bgmusic.mp3",
        "assets/sounds/bgmusic.ogg"]);
     game.load.audio("explosion", ["assets/sounds/explosion.mp3",
        "assets/sounds/explosion.ogg"]);
}
```

We preload both sounds, then Phaser will choose which sound format to play according to browser capabilities.

> `load.audio(key, audioFiles)` handles sound preloading. The first argument is the key, the second is an array of files to be loaded, in different formats.

Once sounds have been preloaded, it's time to play them.

First, we are going to add the looping background music when the game begins, adding a couple of lines in `create` method of `playGame` object:

```
create: function(){
     this.bgMusic = game.add.audio("bgmusic");
     this.bgMusic.loopFull(1);
     // same as before
}
```

There is a new `bgMusic` property which contains the audio related to background

music.

> `add.audio(key)` adds a new audio file to the sound manager. `key` is the name we gave to the sound.

Once the audio file has been added to sound manager, `loopFull` method loops the entire sound.

> `loopFull(volume)` continuously loops a sound, playing it at `volume` volume, which ranges from zero to 100.

These two lines are all we need to loop a sound, but there's more we want for our game: once the spaceship hits a barrier, we want to stop looping background music and play once the explosion sound effect.

At the completion of `destroyTween` tween, when we create the particle explosion, we will also stop music and play explosion sound:

```
destroyTween.onComplete.add(function(){
     this.bgMusic.stop();
     var explosionSound = game.add.audio("explosion");
     explosionSound.play();
     var explosionEmitter = game.add.emitter(this.ship.x, this.ship.y, 200);
     explosionEmitter.makeParticles("smoke");
     explosionEmitter.setAlpha(0.5, 1);
     explosionEmitter.minParticleScale = 0.5;
     explosionEmitter.maxParticleScale = 2;
     explosionEmitter.start(true, 2000, null, 200);
     this.ship.destroy();
     game.time.events.add(Phaser.Timer.SECOND * 2, function(){
          game.state.start("GameOverScreen");
     });
}, this);
```

First, we stop the background music.

> `stop()` method stops playing the sound.

Then in the same way we added loop music, we add the explosion sound, and we play it.

> `play()` method plays a sound.

Now run the project, start the game, hear the background music and once you hit a

barrier, BOOM you're dead.

Resetting the game on restart

Although the game seems to be ready now, you probably noticed there is a new little bug.

I would call it “glitch” rather than “bug” because there's nothing wrong in the code, it's just that once you swipe to move back your ship, barriers speed increase and once you click on play button on the game over screen, barriers will start moving at the same speed they had in previous play.

In other words, barrier speed does not reset, and it quite obvious since we increased speed acting directly on `barrierSpeed` variable, changing its value and never setting it back to its default value.

Anyway, to fix this issue we only need two lines. The idea is to save default `barrierSpeed` value in a temporary variable and restore it once the game is over.

In `create` method of `playGame` object add this line to save `barrierSpeed` value in local `saveBarrierSpeed` property.

```
create: function(){
     this.saveBarrierSpeed = barrierSpeed;
     // same as before
}
```

`saveBarrierSpeed` value will never change during the game.

Then, just before you launch `GameOverScreen` state, you can restore `barrierSpeed` value by setting it to `saveBarrierSpeed`.

```
game.time.events.add(Phaser.Timer.SECOND * 2, function(){
     barrierSpeed = this.saveBarrierSpeed;
     game.state.start("GameOverScreen");
}, this);
```

And finally you can restart the game with barriers running at default speed.

Adding Keyboard controls

You managed to create a cross platform game, but there's still room for improvement to add different way to control the spaceship according to the device you are playing the game on.

If we want desktop computer players to enjoy the game, we should add keyboard control to the game. This way you will allow players with a keyboard to control the game the way they prefer.

Phaser features various method to handle keyboard input, let's have a look at the changes we can make to make the player be able to control the spaceship with SPACEBAR to move from a side to another and SHIFT key to move the spaceship back down.

Change `create` method of `playGame` object adding some new lines:

```
create: function(){
     // same as before
     this.spacebar = game.input.keyboard.addKey(Phaser.Keyboard.SPACEBAR);
     this.spacebar.onDown.add(this.moveShip, this);
     this.shift = game.input.keyboard.addKey(Phaser.Keyboard.SHIFT);
     this.shift.onDown.add(this.restartShip, this);
}
```

Waiting for a key to be pressed isn't that different than waiting for a mouse or tap event. Let's see the new lines in detail:

```
this.spacebar = game.input.keyboard.addKey(Phaser.Keyboard.SPACEBAR);
```

A property called `spacebar` is assigned to actual SPACEBAR key.

> `input.keyboard.addKey(keycode)` add callbacks to the keyboard handler so that each time the key with `keycode` code is pressed down or released the callbacks are activated.

At this time, the key can be handled like an input in the same way we saw when we dealt with mouse and tap events.

```
this.spacebar.onDown.add(this.moveShip, this);
```

See it? It's the same `add` method explained before, with its two arguments telling us which function to call in which context.

So in the end we wait for SPACEBAR key and we call `moveShip` method in the same way we called it when the player was clicking or tapping on the screen.

The same concepts applies to SHIFT key which calls `restartShip` method.

Basically now pressing SPACEBAR is equal to clicking or tapping the screen, while pressing SHIFT key is equal to swiping.

You may think we are already done with keyboard control, but there's one more thing to do as at this time we can have input conflicts.

We know swipes are processed only if `canSwipe` property is set to `true`, and such property is set to `true` when `moveShip` method is called.

What happens if the player presses SPACEBAR calling `moveShip` method and setting `canSwipe` to `true` then accidentally moves the mouse? Being `canSwipe` set to `true`, the movement would be seen as a swipe and this would be an error.

We have to check for swipes only if `moveShip` method is called after a click or tap event. If it's called after a keyboard event, we don't need `canSwipe` property as we already have a dedicated key – SHIFT – to make the ship move back down.

So the question is: how do we check if `moveShip` has been called after a keyboard event? Look how I changed `moveShip` method:

```
moveShip: function(e){
     var isKeyboard = e instanceof Phaser.Key;
     if(!isKeyboard){
          this.ship.canSwipe = true;
     }
     if(this.ship.canMove && !this.ship.destroyed){
          // same as before
     }
}
```

The first thing I want you to notice is the `e` argument in `moveShip` method.

Phaser always passes the event itself in the callback function, so inside `e` object we can find a lot of useful information about the event which called the callback.

> `object instanceof constructor` operator allows to check if `object` is created by `constructor`.

What we need to do is to see if the event has been created by a key. At this time we set `canSwipe` property to `true` only if `e` was not created by a keyboard event.

This way we can prevent player to make unwanted swipes.

Now test the game and try to play with keyboard. Remember: SPACESHIP to change side and SHIFT to move the ship down.

The game itself is quite easy to play, but more complex games may require an extra screen where you explain players how to play.

Adding “how to play” state

We are about to add a new state, in our `window.onload` function:

```
window.onload = function() {
     var width = 640;
     var height = 960;
     var windowRatio = window.innerWidth / window.innerHeight;
     if(windowRatio < width / height){
          var height = width / windowRatio;
     }
     game = new Phaser.Game(width, height, Phaser.AUTO, "");
     game.state.add("Boot", boot);
     game.state.add("Preload", preload);
     game.state.add("TitleScreen", titleScreen);
     game.state.add("HowToPlay", howToPlay);
     game.state.add("PlayGame", playGame);
     game.state.add("GameOverScreen", gameOverScreen);
     game.state.start("Boot");
}
```

This was easy, as we just told the game there's a new state. Another easy step is to tell the game when to call `HowToPlay` state.

The most obvious thing to do is to call it when the player clicks/taps the play

button on the title screen.

We'll change the state we call in `startGame` method of `titleScreen` object:

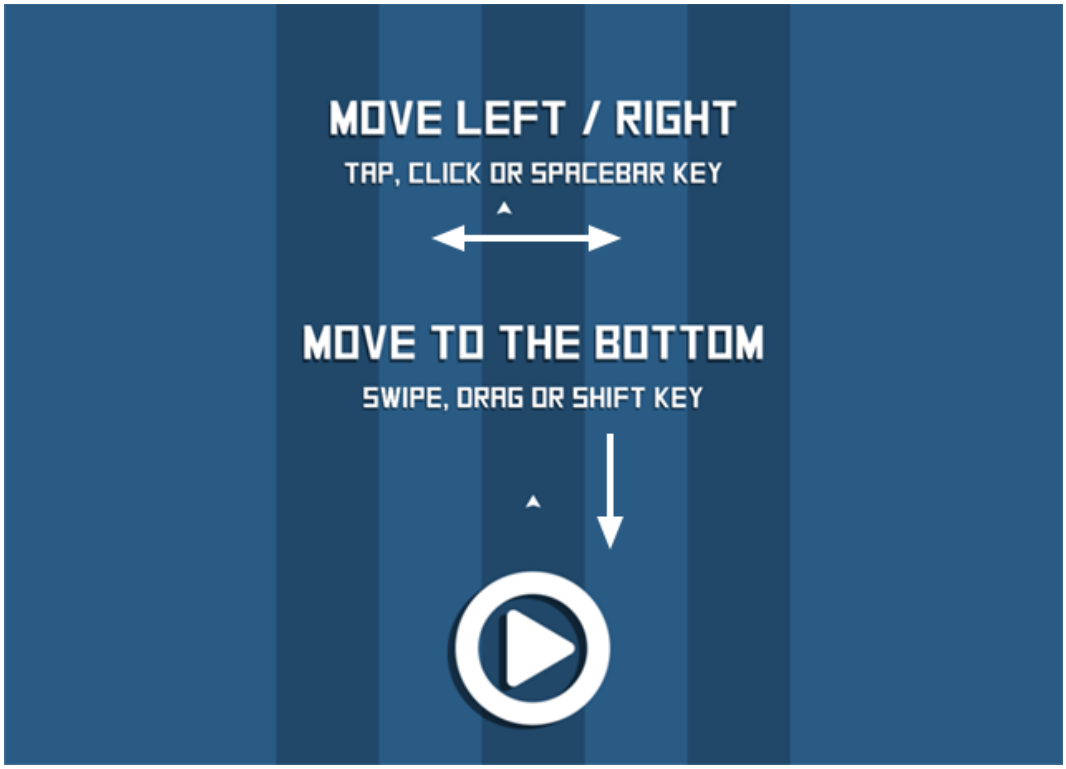
```
startGame: function(){
     game.state.start("HowToPlay");
}
```

Ok, now the funny part: we have to create the state itself: look at the source code we are going to add:

```
var howToPlay = function(game){};
howToPlay.prototype = {
     create: function(){
          var titleBG = game.add.tileSprite(0, 0, game.width, game.height,
               "backsplash");
          titleBG.tint = bgColors[game.rnd.between(0, bgColors.length - 1)];
          document.body.style.background = "#"+titleBG.tint.toString(16);
          game.add.bitmapText(game.width / 2, 120 , "font", "Move left / right",
               60).anchor.x = 0.5;
          game.add.bitmapText(game.width / 2, 200 , "font", "Tap, Click or
               SPACEBAR key", 36).anchor.x = 0.5;
          game.add.bitmapText(game.width / 2, 400 , "font", "Move to the bottom",
               60).anchor.x = 0.5;
          game.add.bitmapText(game.width / 2, 480 , "font", "Swipe, Drag or SHIFT
               key", 36).anchor.x = 0.5;
          var horizontalShip = game.add.sprite(game.width / 2 - 50, 260, "ship");
          horizontalShip.anchor.set(0.5);
          horizontalShip.scale.set(0.5);
          var horizontalShipTween = game.add.tween(horizontalShip).to({
               x: game.width / 2 + 50
          }, 500, "Linear", true, 0, -1);
          horizontalShipTween.yoyo(true);
          var verticalShip = game.add.sprite(game.width / 2, 540, "ship");
          verticalShip.anchor.set(0.5);
          verticalShip.scale.set(0.5);
          var verticalShipTween = game.add.tween(verticalShip).to({
               y: 640
          }, 500, "Linear", true, 0, -1);
          var playButton = game.add.button(game.width / 2, game.height - 150,
               "playbutton", this.startGame);
          playButton.anchor.set(0.5);
          var tween = game.add.tween(playButton).to({
               width: 220,
               height:220
          }, 1500, "Linear", true, 0, -1);
          tween.yoyo(true);
     },
     startGame: function(){
          game.state.start("PlayGame");
     }
}
```

It may seem a lot of stuff but there isn't anything new as it's just made by some bitmap texts, a couple of tweens and a button.

Runt he game at this time, and once you click/tap on the play button in the title screen, that's what you'll see:



Here it is the “how to play” screen, with moving spaceships and the “play” button which is the same we used in game title.

Everything is nicely explained and once the player presses the button, the game begins.

Let's break the code into pieces:

```
var titleBG = game.add.tileSprite(0, 0, game.width, game.height,
       "backsplash");
titleBG.tint = bgColors[game.rnd.between(0, bgColors.length - 1)];
document.body.style.background = "#"+titleBG.tint.toString(16);
```

This is the same old way we use to generate the random background color.

```
game.add.bitmapText(game.width / 2, 120 , "font", "Move left / right",
       60).anchor.x = 0.5;
game.add.bitmapText(game.width / 2, 200 , "font", "Tap, Click or
       SPACEBAR key", 36).anchor.x = 0.5;
game.add.bitmapText(game.width / 2, 400 , "font", "Move to the bottom",
       60).anchor.x = 0.5;
game.add.bitmapText(game.width / 2, 480 , "font", "Swipe, Drag or SHIFT
       key", 36).anchor.x = 0.5;
```

We have four bitmap texts here: they represent the instructions themselves and as you can see there's nothing new in them.

Now the idea is to place two spaceship sprites, one which moves from left to right and from right to left – that is with a yoyo effect – and one which moves from top to bottom without yoyo effect.

This is the horizontal yoyo tween:

```
var horizontalShip = game.add.sprite(game.width / 2 - 50, 260, "ship");
horizontalShip.anchor.set(0.5);
horizontalShip.scale.set(0.5);
var horizontalShipTween = game.add.tween(horizontalShip).to({
     x: game.width / 2 + 50
}, 500, "Linear", true, 0, -1);
horizontalShipTween.yoyo(true);
```

And this is the vertical tween:

```
var verticalShip = game.add.sprite(game.width / 2, 540, "ship");
verticalShip.anchor.set(0.5);
verticalShip.scale.set(0.5);
var verticalShipTween = game.add.tween(verticalShip).to({
     y: 640
}, 500, "Linear", true, 0, -1);
```

There isn't that much to say about these lines, because you already know how to create tweens.

This is the good side of learning a programming language: the more code you write, the most often you will find yourself reusing the code you have already written, speeding up the development.

Talking about reusing the code, the play button is nothing more than a copy/paste of the same code used in title screen:

```
var playButton = game.add.button(game.width / 2, game.height - 150,
       "playbutton", this.startGame);
playButton.anchor.set(0.5);
var tween = game.add.tween(playButton).to({
     width: 220,
     height:220
}, 1500, "Linear", true, 0, -1);
tween.yoyo(true);
```

And finally, `startGame` callback method launches the game itself:

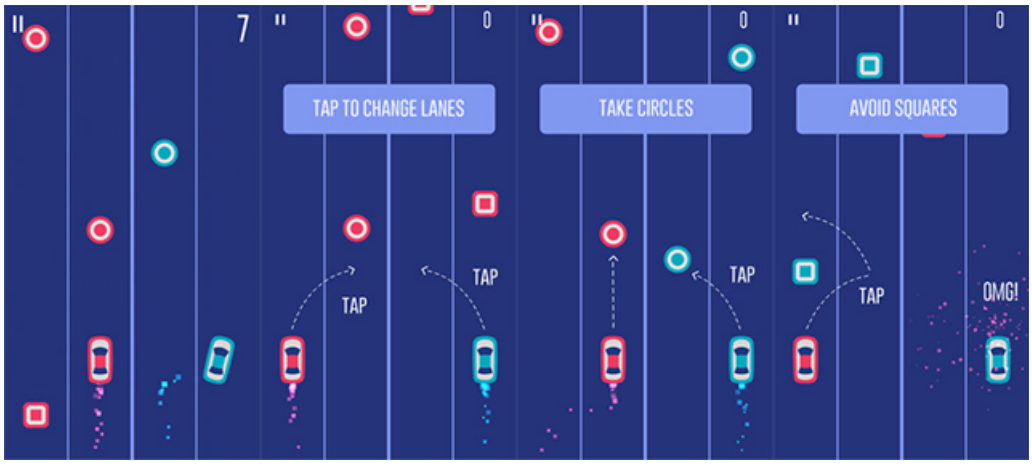
```
startGame: function(){
     game.state.start("PlayGame");
}
```

And now your game has a new state explaining how to play.

Now that your first endless vertical runner game has been completed, it will be easier to create different games using similar gameplay.

Making your next game

The game we are about to create now is heavily based on **2 Cars** by Ketchapp Studio (https://itunes.apple.com/en/app/2-cars/id936839198?mt=8).



This game has also been featured on the Apple app store and it's free so I suggest you to install it on your mobile phone and play it a bit, so you will have an idea about the game you are about to create.

You will see how faster we will proceed during the creation of this game, since we already know all core concepts.

Project folder structure will remain the same, with `assets`, `fonts`, `sprites` and `sounds` folders.

We'll keep the same background music, create a new bitmap font with Littera and this will be the content of our `sprites` folder:

Green background actually will be transparent, I've set it to green to let you see more clearly the transparency of each image.

Everything has been drawn in shades of gray, as we'll tint it later in the game.

The starting template we will be working on is this one, saved in `game.js`:

```
var game;

window.onload = function() {
     var width = 640;
     var height = 960;
     var windowRatio = window.innerWidth / window.innerHeight;
     if(windowRatio < width / height){
          var height = width / windowRatio;
     }
     game = new Phaser.Game(width, height, Phaser.AUTO, "");
     game.state.add("Boot", boot);
     game.state.add("Preload", preload);
     game.state.add("TitleScreen", titleScreen);
     game.state.add("HowToPlay", howToPlay);
     game.state.add("PlayGame", playGame);
     game.state.add("GameOverScreen", gameOverScreen);
     game.state.start("Boot");
}

var boot = function(game){};
boot.prototype = {
     create: function(){
     }
}

var preload = function(game){};
preload.prototype = {
     create: function(){
     }
}

var titleScreen = function(game){};
titleScreen.prototype = {
     create: function(){
     }
}

var howToPlay = function(game){};
howToPlay.prototype = {
     create: function(){
     }
}

var playGame = function(game){};
playGame.prototype = {
     create: function(){
     }
}

var gameOverScreen = function(game){};
gameOverScreen.prototype = {
     create: function(){
     }
}
```

I just created `game` variable, made a game with a basic 640x960 resolution which

adapts to different heights and set all the states we saw during the making of the previous game.

Now we will build the new game starting from this structure.

Creating boot state

Boot state remains exactly the same as before, preloading the loading bar and scaling the game to cover the entire screen:

```
var boot = function(game){};
boot.prototype = {
     preload: function(){
          game.load.image("loading","assets/sprites/loading.png");
     },
     create: function(){
          game.scale.pageAlignHorizontally = true;
          game.scale.pageAlignVertically = true;
          game.scale.scaleMode = Phaser.ScaleManager.SHOW_ALL;
          game.state.start("Preload");
     }
}
```

Then `Preload` state is launched.

If you test the project now, you won't see anything because actually nothing happens so don't worry if you see a black page.

Your previous game started with a black page too and you eventually came up with **Rising Ships** game.

A nice way to start adding content to your game is loading assets.

Creating preload state

Preloading assets will be basically the same process in all games you will develop with Phaser.

Once you setup the loading bar preloaded in boot state, it's just a matter of loading all sprites, texts and fonts you will need in the game.

I already showed you the graphics we are going to use. As for the sounds we have a background music, an explosion sound to be played when a car crashes or

misses a target, and a hit sound to be played when a car collects a target.

```
var preload = function(game){};
preload.prototype = {
     preload: function(){
          var loadingBar = this.add.sprite(game.width / 2, game.height / 2,
               "loading");
          loadingBar.anchor.setTo(0.5);
          game.load.setPreloadSprite(loadingBar);
          game.load.image("title", "assets/sprites/title.png");
          game.load.image("playbutton", "assets/sprites/playbutton.png");
          game.load.image("backsplash", "assets/sprites/backsplash.png");
          game.load.image("target", "assets/sprites/target.png");
          game.load.image("car", "assets/sprites/car.png");
          game.load.image("particle", "assets/sprites/particle.png");
          game.load.bitmapFont("font", "assets/fonts/font.png",
               "assets/fonts/font.fnt");
          game.load.audio("bgmusic", ["assets/sounds/bgmusic.mp3",
               "assets/sounds/bgmusic.ogg"]);
          game.load.audio("explosion", ["assets/sounds/explosion.mp3",
               "assets/sounds/explosion.ogg"]);
          game.load.audio("hit", ["assets/sounds/hit.mp3",
               "assets/sounds/hit.ogg"]);
     },
     create: function(){
          game.state.start("TitleScreen");
     }
}
```

Then we call the state which will display the title screen.

Creating title screen

Finally we are ready to put some content in the game.

The idea is to create the same title screen we made for **Rising Ships**, with a random background color, a tiled background image, the game title, an animated play button and the best score printed with bitmap fonts.

We need a couple of new global variables:

```
var game;
var localStorageName = "doublelanegame";
var bgColors = [0x54c7fc, 0xffcd00, 0xff2851, 0x62bd18];
```

Like in previous game, `localStorageName` contains the name of the local storage variable.

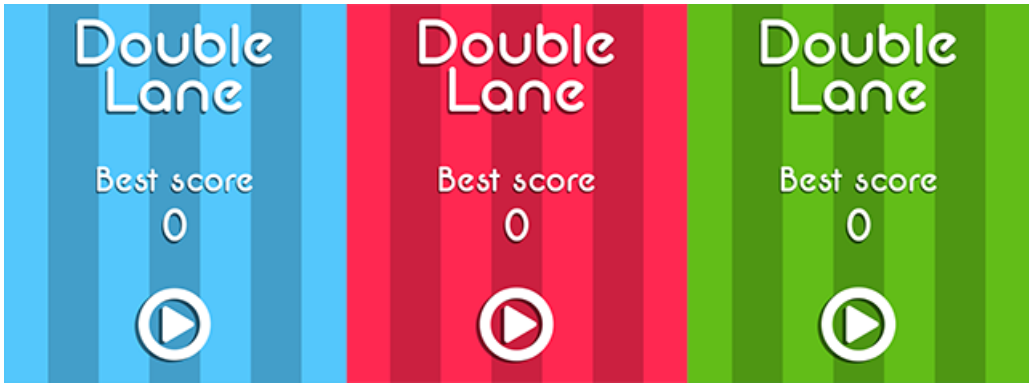
Keep in mind you have to use a different name for each game you make, or high scores will overwrite each other.

`bgColors` is an array of color codes we will randomly pick as background.

And this is the content of the final `titleScreen` object:

```
var titleScreen = function(game){};
titleScreen.prototype = {
     create: function(){
          savedData = localStorage.getItem(localStorageName)==null?
               {score:0}:JSON.parse(localStorage.getItem(localStorageName));
          var titleBG = game.add.tileSprite(0, 0, game.width, game.height,
               "backsplash");
          titleBG.tint = bgColors[game.rnd.between(0, bgColors.length - 1)];
          document.body.style.background = "#" + titleBG.tint.toString(16);
          var title = game.add.image(game.width / 2, 160, "title");
          title.anchor.set(0.5);
          game.add.bitmapText(game.width / 2, 400, "font", "Best score",
               90).anchor.x = 0.5;
          game.add.bitmapText(game.width / 2, 500, "font",
               savedData.score.toString(), 120).anchor.x = 0.5;
          var playButton = game.add.button(game.width / 2, game.height - 150,
               "playbutton", this.startGame);
          playButton.anchor.set(0.5);
          var tween = game.add.tween(playButton).to({
               width: 220,
               height: 220
          }, 1500, "Linear", true, 0, -1);
          tween.yoyo(true);
     },
     startGame: function(){
          game.state.start("PlayGame");
     }
}
```

There's nothing new in this code, so you can imagine the result:



We have our title screen filled with a random color, the glowing “play” button and the best score taken directly from local storage.

It's always nice when you can create a simple but effective title screen with only a few lines, but now it's time to start making the game itself.

Creating the road

The game takes place on a road. This is the first thing we are going to add to the game, as well as the background music as we already know how to add sounds to the game.

We'll add some new global variables first:

```
var game;
var localStorageName = "doublelanegame";
var bgColors = [0x54c7fc, 0xffcd00, 0xff2851, 0x62bd18];
var score;
var savedData;
var laneWidth = 138;
var lineWidth = 4;
```

`score` and `savedData` have the same meaning already been explained during the making of the previous game, so we have only two brand new variables: `laneWidth` and `lineWidth`.

`laneWidth` represents the width, in pixels, of each lane. Remember the game features two roads, and each road is has two lanes – it's called Double Lane – so choose wisely the width of each lane.

`lineWidth` stores the width, in pixels of the line which separates the two lanes of each road. You can see the values I am using in the game, but you are free to play with these numbers and the game will react accordingly.

Since we are starting to create the game itself, we are going to write code in `playGame` object.

You are about to see a lot of code, but don't worry as at this point you will find it ridiculously easy to understand.

```
var playGame = function(game){};
playGame.prototype = {
     create: function(){
          this.bgMusic = game.add.audio("bgmusic");
          this.bgMusic.loopFull(1);
          score = 0;
          savedData = localStorage.getItem(localStorageName)==null?
               {score:0}:JSON.parse(localStorage.getItem(localStorageName));
          var tintColor = bgColors[game.rnd.between(0, bgColors.length - 1)];
          document.body.style.background = "#"+tintColor.toString(16);
          var pickedColors = [tintColor];
          this.roadWidth = laneWidth * 2 + lineWidth;
          var roadSeparator = game.add.tileSprite(this.roadWidth, 0, game.width -
               (this.roadWidth * 2), game.height, "particle");
          roadSeparator.tint = tintColor;
          var leftLine = game.add.tileSprite(laneWidth, 0, lineWidth, game.height,
               "particle");
          leftLine.tint = tintColor;
          var rightLine = game.add.tileSprite(game.width - laneWidth - lineWidth,
               0, lineWidth, game.height, "particle");
          rightLine.tint = tintColor;
          this.carGroup = game.add.group();
          this.targetGroup = game.add.group();
          this.scoreText = game.add.bitmapText(game.width / 2, 40 , "font", "0",
               120)
          this.scoreText.anchor.x = 0.5;
     }
}
```

Let's see what we are going to do line by line:

```
this.bgMusic = game.add.audio("bgmusic");
this.bgMusic.loopFull(1);
```

We start looping the background music.

```
savedData = localStorage.getItem(localStorageName)==null?
       {score:0}:JSON.parse(localStorage.getItem(localStorageName));
```

Now we update `savedData` content, using the local storage.

```
var tintColor = bgColors[game.rnd.between(0, bgColors.length - 1)];
document.body.style.background = "#"+tintColor.toString(16);
var pickedColors = [tintColor];
```

And here is how we assign a random tint color, also using it for the background. The first new thing which differs from the making of Rising Ships is

`pickedColors` variable.

We want to store in an array the random color we just used for the background, because we will use more random colors later in the game and we don't want to use the same random color we already used for the background.

```
this.roadWidth = laneWidth * 2 + lineWidth;
```

`roadWidth` property stores the actual width of the road, in pixels, which is due by the sum of both lanes plus the separation line. With the values I used in this example, it will be 138 * 2 + 4 = 280 pixels.

```
var roadSeparator = game.add.tileSprite(this.roadWidth, 0, game.width -
       (this.roadWidth * 2), game.height, "particle");
roadSeparator.tint = tintColor;
```

`roadSeparator` is a tile sprite which fills the gap between the two roads, each one made by two lanes and a separation line.

```
var leftLine = game.add.tileSprite(laneWidth, 0, lineWidth, game.height,
       "particle");
leftLine.tint = tintColor;
```

`leftLine` is the separation line between the two lanes in left road.

```
var rightLine = game.add.tileSprite(game.width - laneWidth - lineWidth,
       0, lineWidth, game.height, "particle");
rightLine.tint = tintColor;
```

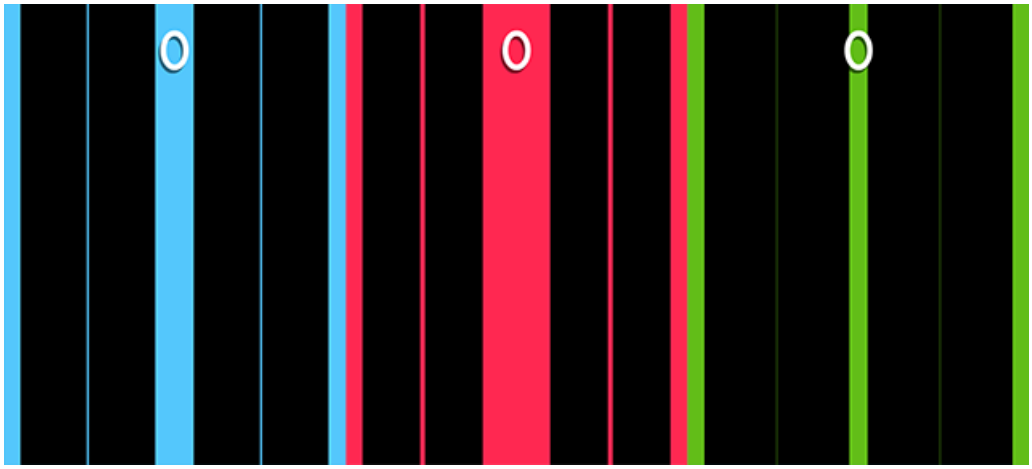
`rightLine` is the separation line between the two lanes in right road.

I want you to see how I am tinting all these sprites with `tintColor` and how I am using as tile `particle` image which is a little white square.

We are also preparing the game to host the cars and the targets, adding two groups, one for the cars and one for the targets, and last but not least we add the bitmap text which will show the current score, which is zero at the moment.

```
this.carGroup = game.add.group();
this.targetGroup = game.add.group();
this.scoreText = game.add.bitmapText(game.width / 2, 40 , "font", "0", 120)
this.scoreText.anchor.x = 0.5;
```

Now run the game and play with `laneWidth` and `lineWidth` values to create the kind of road which fits your needs. The entire gameplay will react accordingly.



Now it's time to add the cars.

Adding the cars

The game features two cars, so we need a variable to store them. `cars` array will have two items, the cars themselves.

```
var game;
var localStorageName = "doublelanegame";
var bgColors = [0x54c7fc, 0xffcd00, 0xff2851, 0x62bd18];
var score;
var savedData;
var laneWidth = 138;
var lineWidth = 4;
var cars = [];
```

To add the cars to the game, we need to write some more lines to `create` method in `playGame` object.

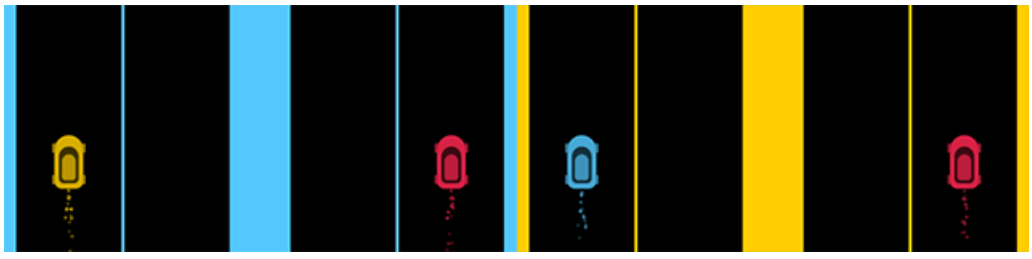
At this time we won't be able to control the cars but we'll add a particle effect to

their tails to simulate the exhaust.

Let's add these lines to `create` method:

```
create: function(){
     // same as before
     for(var i = 0; i < 2; i++){
          cars[i] = game.add.sprite(0, game.height - 120, "car");
          cars[i].positions = [(game.width + roadSeparator.width) / 2 * i  +
               laneWidth / 2, (game.width + roadSeparator.width) / 2 * i +
               laneWidth + lineWidth + laneWidth / 2];
          cars[i].anchor.set(0.5);
          do{
               tintColor = bgColors[game.rnd.between(0, bgColors.length - 1)];
          } while (pickedColors.indexOf(tintColor) >= 0)
          cars[i].tint = tintColor;
          pickedColors.push(tintColor);
          cars[i].canMove = true;
          cars[i].side = i;
          cars[i].x = cars[i].positions[cars[i].side];
          game.physics.enable(cars[i], Phaser.Physics.ARCADE);
          cars[i].body.allowRotation = false;
          cars[i].body.moves = false;
          cars[i].smokeEmitter = game.add.emitter(cars[i].x, cars[i].y +
               cars[i].height / 2 + 2, 20);
          cars[i].smokeEmitter.makeParticles("particle");
          cars[i].smokeEmitter.setXSpeed(-15, 15);
          cars[i].smokeEmitter.setYSpeed(50, 150);
          cars[i].smokeEmitter.setAlpha(0.2, 0.5);
          cars[i].smokeEmitter.start(false, 500, 20);
          cars[i].smokeEmitter.forEach(function(p){
               p.tint = cars[i].tint;
          });
          this.carGroup.add(cars[i]);
     }
}
```

And let's jump straight to what you are going to see on your screen:



You can see random colored cars with their particle trails, and if you look closer, you will also see cars will never be the same color of the road.

Remember `pickedColors` array we used before? Now it will come into play.

Let me show you what happened, line by line:

```
for(var i = 0; i < 2; i++){
     // rest of the script
}
```

We said we have two cars, so we need a loop which iterates two times, with index 0 for the first car – on the left – and 1 for the second car, on the right.

```
cars[i] = game.add.sprite(0, game.height - 120, "car");
```

`cars[i]` now contains the sprite of the `i`-th car, which is added to the stage. Its horizontal position is zero because we still have to decide where to place it, while `game.height - 120` will be its vertical, final position. Remember in vertical endless runners player never moves vertically, it's the whole environment to scroll towards the player.

```
cars[i].positions = [(game.width + roadSeparator.width) / 2 * i  + laneWidth / 2,
       (game.width + roadSeparator.width) / 2 * i + laneWidth + lineWidth +
       laneWidth / 2];
cars[i].anchor.set(0.5);
```

Cars can only move from one lane to another, just like the spaceship in Rising Ships, so for each car we have to store somewhere its two possible positions.

`position` attribute is an array with two elements: the first is the horizontal position when the car is on the left lane, and the second is the horizontal position when the car is on the right lane. It may seem a long formula but it's meant to work with any `laneWidth` and `lineWidth` value.

Let me show you what you'll get from this formula:

Left car, left position: `laneWidth` / 2.

Left car, right position: `laneWidth` * 1.5 + `lineWidth`.

Right car, left position: left position of left car + game width / 2 + `roadSeparator` width / 2.

Right car, right position: right position of left car + game width / 2 + `roadSeparator` width / 2.

Finally we set the anchor point of the car to its center.

```
do{
     tintColor = bgColors[game.rnd.between(0, bgColors.length - 1)];
} while (pickedColors.indexOf(tintColor) >= 0)
cars[i].tint = tintColor;
pickedColors.push(tintColor);
```

Here is when `pickedColors` array comes into play. We want background, left car and right cars colors to be randomly picked but never to be the same.

In the `do while` loop we keep choosing random colors until it's not a color we already picked. How can we know we already picked a color? Because it's inside `pickedColors` array.

> The `do while` statement creates a loop that executes a block of code once, before checking if the condition is `true`, then it will repeat the loop as long as the condition is `true`. Use it when you want to run a loop at least one time.

Now we need to create a couple of custom properties to see if cars can change lane and which lane are they running in, just like we made with the spaceship in Rising Ships.

```
cars[i].canMove = true;
cars[i].side = i;
cars[i].x = cars[i].positions[cars[i].side];
```

`canMove` is `true` if the car can change lane, `false` otherwise.

`side` property is zero if the car is on the left lane, 1 if it's on the right line.

You can see left car is starting on the left lane while right car is starting on the right lane.

Finally `x` property of the car is set to its actual value, placing the car in the proper horizontal coordinate.

```
game.physics.enable(cars[i], Phaser.Physics.ARCADE);
cars[i].body.allowRotation = false;
cars[i].body.moves = false;
```

We will be using ARCADE physics in this game to check for collisions and handle environment speed just like we made with Rising Ships, so cars are enabled but we don't want physics to move or rotate them, that's why `allowRotation` and `moves` properties are set to `false`.

> `allowRotation` property of a body allows the body to be rotated.
>
> `moves` property of a body when set to `true` allows the physics system to move the body, when set to `false` does not let physics to move the body, which must be moved manually.

You may wonder why we are setting these properties in this game when Rising Ships worked perfectly without setting them. I have to say, the game can work with or without setting `allowRotation` and `moves`, but I want cars to stop exactly where they are in case of collision, and this is not possible if you don't set these properties as colliding cars would bounce or move somehow. You can test it by yourself once you finish to make the game.

```
cars[i].smokeEmitter = game.add.emitter(cars[i].x, cars[i].y + cars[i].height / 2
        + 2, 20);
cars[i].smokeEmitter.makeParticles("particle");
cars[i].smokeEmitter.setXSpeed(-15, 15);
cars[i].smokeEmitter.setYSpeed(50, 150);
cars[i].smokeEmitter.setAlpha(0.2, 0.5);
cars[i].smokeEmitter.start(false, 500, 20);
```

With both cars placed on the stage and their physics bodies ready to check for collisions, we only have to create the emitters which will create a particle effect simulating the exhaust system. It's basically the same emitter with the same settings as the one created for the smoke trail in Rising Ships game.

But there's something more to do: particle image is white, while we want particles to have the same color as the cars. We can't tint an emitter, so we have to tint each particle, one by one.

```
cars[i].smokeEmitter.forEach(function(p){
     p.tint = cars[i].tint;
});
```

This is how we loop through all particles and apply them the same tint color applied to the car.

> `emitter.forEach(callback, context)` calls `callback` function on each particle in the emitter with `context` context.

And the last thing to do is to add cars to `carGroup` group.

```
this.carGroup.add(cars[i]);
```

With both cars on the stage, now it's time to let the player control them.

Controlling the cars

Player will control cars making them change lane by tapping on the screen.

A tap on the left half of the screen will make left car change lane, and a tap on the right half of the screen will make right car change lane.

The first thing we need to do is the creation of a new global variable storing the amount of milliseconds needed for the car to change lane.

```
var game;
var localStorageName = "doublelanegame";
var bgColors = [0x54c7fc, 0xffcd00, 0xff2851, 0x62bd18];
var score;
var savedData;
var laneWidth = 138;
var lineWidth = 4;
var cars = [];
var carTurnSpeed = 200;
```

I set `carTurnSpeed` to 200 but you are free to use the value you prefer, according to the gameplay you have in mind.

Just like Rising Ships, we need to wait for the player to tap or click the screen, so

let's add the listener to `create` method in `playGame` object:

```
create: function(){
     // same as before
     game.input.onDown.add(this.moveCar, this);
}
```

Now once the player interacts with the game, `moveCar` method is called.

We are moving cars in the same way we moved the spaceship in previous game, but this time we'll add a couple of features, because we need to know which car we are going to move and we want to add a steering effect.

Look at `moveCar` method, to be created inside `playGame` object:

```
moveCar: function(e){
     var carToMove = Math.floor(e.position.x / (game.width / 2));
     if(cars[carToMove].canMove){
          cars[carToMove].canMove = false;
          var steerTween = game.add.tween(cars[carToMove]).to({
               angle: 20 - 40 * cars[carToMove].side
          }, carTurnSpeed / 2, Phaser.Easing.Linear.None, true);
          steerTween.onComplete.add(function(){
               var steerTween = game.add.tween(cars[carToMove]).to({
                    angle: 0
               }, carTurnSpeed / 2, Phaser.Easing.Linear.None, true);
          })
          cars[carToMove].side = 1 - cars[carToMove].side;
          var moveTween = game.add.tween(cars[carToMove]).to({
               x: cars[carToMove].positions[cars[carToMove].side],
          }, carTurnSpeed, Phaser.Easing.Linear.None, true);
          moveTween.onComplete.add(function(){
               cars[carToMove].canMove = true;
          })
     }
}
```

Let's give a closer look at what happens in this method.

```
var carToMove = Math.floor(e.position.x / (game.width / 2));
```

At each player input, we need to know which car we have to move.

We said touching the left half of the game will move the left car and touching the right half of the game we will move the right car so we have to know the

coordinates of each input.

> `position` property of an input event returns a point with the coordinates of the input. `position.x` and `position.y` return respectively the horizontal and vertical coordinates of the input.

Do some math, and you will see `carToMove` will be zero if the player touches the left half of the game, and 1 if the player touches the right half of the game.

Since `cars[0]` is the left car and `cars[1]` is the right car, we can say `cars[carToMove]` is the car we need to move now.

```
if(cars[carToMove].canMove){
     // rest of the code
}
```

Before we move a car, we need to check if it can move, looking at `canMove` property.

```
cars[carToMove].canMove = false;
```

If you execute this line, it means the car can move and we are about to move it. Let's set `canMove` property to `false` now because we don't want the player to be able to move a car which is already moving.

```
var steerTween = game.add.tween(cars[carToMove]).to({
     angle: 20 - 40 * cars[carToMove].side
}, carTurnSpeed / 2, Phaser.Easing.Linear.None, true);
```

Here we go with a tween to simulate the steering. You already know how tweens work since you used them to change the position and the size of sprites. This time you will change the angle, setting to 20 if we are on left lane – check `side` property – or -20 if we are on the right lane.

> `angle` property of a sprite is the rotation in degrees from its original orientation. Values from 0 to 180 represent clockwise rotation. Values from 0 to -180 represent counterclockwise rotation.

As you can see, the tween lasts only `carTurnSpeed / 2` milliseconds, which is half of the time allowed to cars to change lane.

We will use the second half of the time to set car angle to zero again and complete the steering effect.

```
steerTween.onComplete.add(function(){
     var steerTween = game.add.tween(cars[carToMove]).to({
          angle: 0
     }, carTurnSpeed / 2, Phaser.Easing.Linear.None, true);
})
```

Another thing to do is changing `side` property switching it from zero to one or from one to zero.

```
cars[carToMove].side = 1 - cars[carToMove].side;
```

At this time we only changed the angle of the car but the position is still the same, so here is the tween to change `x` position, this time taking the whole `carTurnSpeed` amount of milliseconds.

```
var moveTween = game.add.tween(cars[carToMove]).to({
     x: cars[carToMove].positions[cars[carToMove].side],
}, carTurnSpeed, Phaser.Easing.Linear.None, true);
```

And finally we can update `canMove` property once `moveTween` has been completed:

```
moveTween.onComplete.add(function(){
     cars[carToMove].canMove = true;
})
```

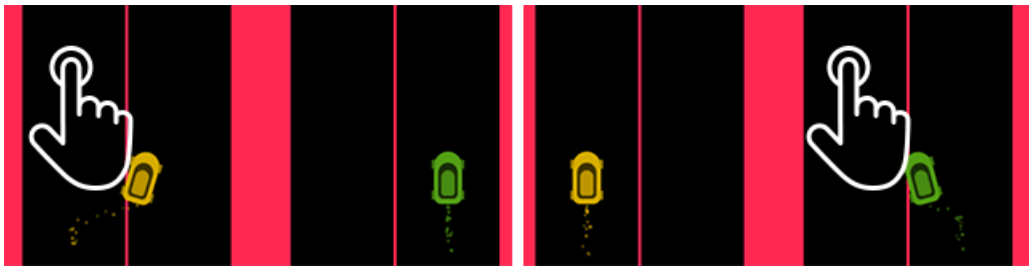
Cars will be able to move now, but the emitters which simulate the exhaust don't, so we need to manually update their position.

This is where `update` method of `playGame` object must come into play. Remember `update` method is executed at each frame, so at each frame we will

update smoke emitters position:

```
update: function(e){
     cars[0].smokeEmitter.x = cars[0].x;
     cars[1].smokeEmitter.x = cars[1].x;
}
```

Launch the game, and control cars by tapping on the left or on the right side of the game. Watch the cars steering as they change lane.



We are done with touch/click input but the true spirit of cross platform development wants us to let the player to control cars also with keyboard if needed.

Controlling the cars with keyboard

Players will be able to control the game with two keys: Z to make left car change its lane, and X to make right car change its lane.

We have to add four more lines to `create` method, to set the listeners:

```
create: function(){
     // same as before
     this.leftKey = game.input.keyboard.addKey(Phaser.Keyboard.Z);
     this.leftKey.onDown.add(this.moveCar, this);
     this.rightKey = game.input.keyboard.addKey(Phaser.Keyboard.X);
     this.rightKey.onDown.add(this.moveCar, this);
}
```

Now both Z and X keys once pressed will call `moveCar` method, which is the same method called when we received a touch/click input.

Following the same concept seen in the creation of Rising Ships, we will work on

`e` event passed as argument to see if we are dealing with a keyboard or an input event. If we have a keyboard event, we check which key fired the event and move the proper car accordingly.

These are the new lines to add to `moveCar` method to make it compatible with keyboard events.

```
moveCar: function(e){
     var carToMove;
     var isKeyboard = e instanceof Phaser.Key;
     if(isKeyboard){
          if(e.keyCode == 88){
               carToMove = 1;
          }
          else{
               carToMove = 0;
          }
     }
     else{
          carToMove = Math.floor(e.position.x / (game.width / 2));
     }
     if(cars[carToMove].canMove){
          // same as before
     }
}
```

If you test the game now, you will also be able to change cars lanes with Z and X keys.

> `keyCode` property returns the key code of a Phaser.Key object.

Finally we completed the part of the game which controls cars movement. It's time to add the other actors of this game: the targets to collect or avoid.

Adding targets

Let's spend a minute talking about targets: they appear from the top of the screen at a given interval and move at a given speed towards the bottom of the screen where they disappear.

Each target has a color, randomly chosen among cars colors, and each car must collect all targets with the same color and avoid all targets with the other color.

Collecting a target with a different color means game over. Not collecting a target

with the same color means game over.

The first thing we have to do is to define the delay between a target and next one, and the speed targets move from the top to the bottom of the game.

This means two new global variables to add:

```
var game;
var localStorageName = "doublelanegame";
var bgColors = [0x54c7fc, 0xffcd00, 0xff2851, 0x62bd18];
var score;
var savedData;
var laneWidth = 138;
var lineWidth = 4;
var cars = [];
var carTurnSpeed = 200;
var targetDelay = 1200;
var targetSpeed = 180;
```

`targetDelay` is the amount of time in milliseconds between the creation of a target and the creation of next target. `targetSpeed` is the speed of the target, in pixels per second.

The time loop which will create targets will be placed in `create` method of `playGame` object, at the end of the script:

```
create: function(){
     // same as before
     this.targetLoop = game.time.events.loop(targetDelay, function(){
          for(var i = 0; i < 2; i++){
               var target = new Target(game, i);
               game.add.existing(target);
               this.targetGroup.add(target);
          }
     }, this);
}
```

As you can see, we add two targets each time: one on the left road and one on the right road.

Targets are instances of `Target` class, it's exactly the same concept I explained during the creation of the barriers during Rising Ships development.

So we just have to create `Target` class, which constructor has two arguments: the

game itself and the lane where to place the target.

```
Target = function (game, lane) {
     var position = game.rnd.between(0, 1);
     Phaser.Sprite.call(this, game, cars[lane].positions[position], -20,
       "target");
     game.physics.enable(this, Phaser.Physics.ARCADE);
     this.anchor.set(0.5);
     var tint = game.rnd.between(0, 1);
     this.mustPickUp = tint == lane;
     this.tint = cars[tint].tint;
     this.body.velocity.y = targetSpeed;
};

Target.prototype = Object.create(Phaser.Sprite.prototype);
Target.prototype.constructor = Target;

Target.prototype.update = function() {
     if(this.y > game.height + this.height / 2){
          this.destroy();
     }
};
```

Although it's almost the same snippet of code used during the creation of barriers in Rising Ships, there are a couple of new lines I want to show in detail:

```
var position = game.rnd.between(0, 1);
```

`position` variable contains a number which can be 0 – target placed on the left lane – or 1 – target placed on the right lane.

```
var tint = game.rnd.between(0, 1);
```

`tint` also takes a number which can be 0 – same tint color as left car – or 1 – same tint color as the right car.

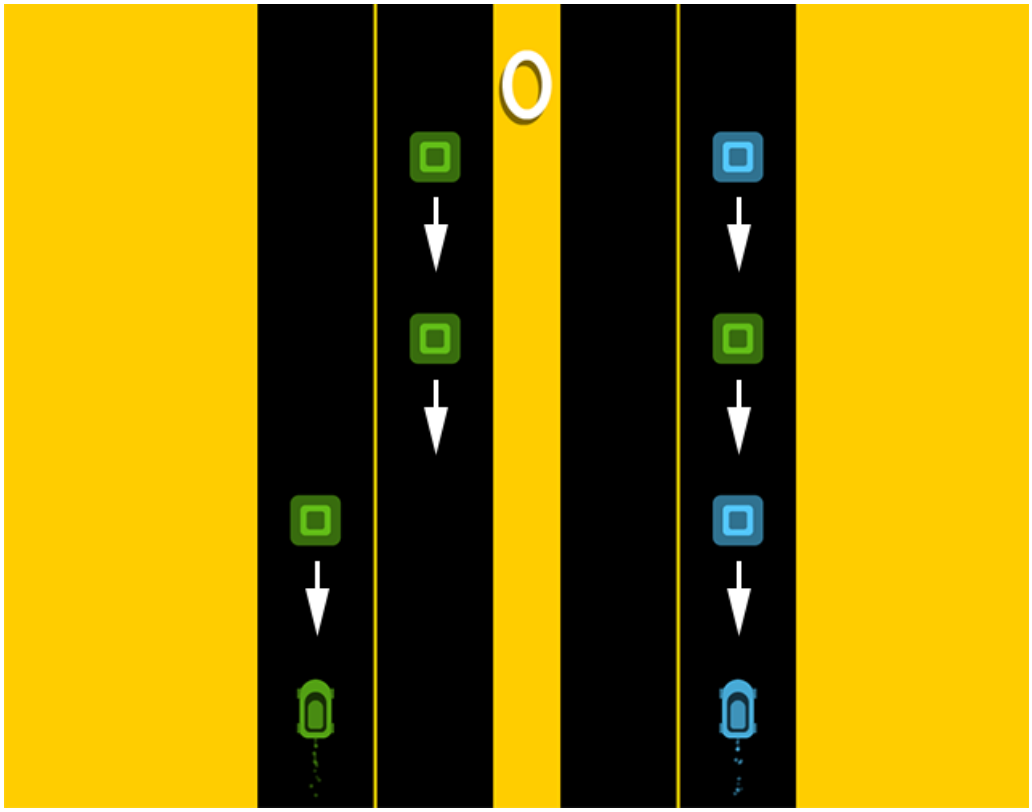
```
this.mustPickUp = tint == lane;
```

This line is the core of the game. `mustPickUp` property of each target is `true` if `tint` has the same value as `lane`. This means target color matches with car color.

It will come very handy when we'll have to check for collisions or missing targets.

Remember each car must collect all targets matching car color and avoid all targets which do not match car color.

Now run the game and look how targets scroll towards cars. This gives the illusion cars are running.



Time to add some interaction between cars and targets.

Collecting targets

Cars will be able to collect targets which match with their colors and will explode if they hit target with different colors.

We are going to explain this whole process, as it's not that different than what you already saw during the creation of the previous game.

We also need a couple of new sound effects, one to be played when a car collects

the right target, and one to be played when a car hits a wrong target.

The first sound will be added in `create` method of `playGame` object. We are going to play this sound a lot of times. Well, at least that's what should happen if players manage to survive enough.

```
create: function(){
     this.bgMusic = game.add.audio("bgmusic");
     this.bgMusic.loopFull(1);
     this.hitSound = game.add.audio("hit");
     // same as before
}
```

In `update` method – remember, it's executed at each frame – we are going to check if any of the cars is colliding with any of the targets.

`physics.arcade.collide` method you've already met in the making of Rising Ships also works with group versus group collisions, so we are going to check for collision between any car – in `carGroup` group – with any target – in `targetGroup` group.

These are the new lines to add to `update` method:

```
update: function(e){
     cars[0].smokeEmitter.x = cars[0].x;
     cars[1].smokeEmitter.x = cars[1].x;
     game.physics.arcade.collide(this.carGroup, this.targetGroup, function(c, t){
          if(c.tint == t.tint){
               t.destroy();
               this.hitSound.play();
          }
          else{
               this.targetFail(t);
          }
     }, null, this);
}
```

You can clearly see I am checking for collisions between `carGroup` and `targetGroup` and the callback function will contain in its arguments `c` and `t` the bodies which actually collided.

So to check for color match, we just have to see if `c` tint color is the same as `t` tint color. In this case we simply destroy the target and play the proper sound.

Things change when a car collects the wrong target, as we have to end the game.

`targetFail` method will handle everything we have to do before launching the game over state. The game is about to end but there's still a lot to do, look at `targetFail` method:

```
targetFail: function(t){
     game.input.keyboard.removeKey(Phaser.Keyboard.Z);
     game.input.keyboard.removeKey(Phaser.Keyboard.X);
     cars[0].smokeEmitter.on = false;
     cars[1].smokeEmitter.on = false;
     game.time.events.remove(this.targetLoop);
     game.tweens.removeAll();
     for(var i = 0; i < this.targetGroup.length; i++){
          this.targetGroup.getChildAt(i).body.velocity.y = 0;
     }
     game.input.onDown.remove(this.moveCar, this);
     var explosionEmitter = game.add.emitter(t.x, t.y, 200);
     explosionEmitter.makeParticles("particle");
     explosionEmitter.gravity = 0;
     explosionEmitter.setAlpha(0.2, 1);
     explosionEmitter.minParticleScale = 0.5;
     explosionEmitter.maxParticleScale = 3;
     explosionEmitter.start(true, 2000, null, 200);
     explosionEmitter.forEach(function(p){
          p.tint = t.tint;
     });
     t.destroy();
     this.bgMusic.stop();
     var explosionSound = game.add.audio("explosion");
     explosionSound.play();
     game.time.events.add(Phaser.Timer.SECOND * 2, function(){
          game.state.start("GameOverScreen");
     }, this);
}
```

First, `t` argument is the target which caused the game to end. It's very important to have it as argument as there are a lot of targets in game but we are going to make this one to explode.

Let's see the lines in detail, you'll see there's nothing new, just a collection of well known concepts merged into one big script.

```
game.input.keyboard.removeKey(Phaser.Keyboard.Z);
game.input.keyboard.removeKey(Phaser.Keyboard.X);
```

When you lose the game, you can't control the cars anymore, so we are removing

the listeners waiting for X and Z keyboard input.

> `input.keyboard.removeKey(keycode)` removes callbacks to the keyboard handler.

You will also find input listener is disabled, following pretty much the same concept.

```
game.input.onDown.remove(this.moveCar, this);
```

And now the player is not longer able to control any car.

> `input.onDown.remove(callback, callbackContext)` removes a listener waiting for an input.

We also want to stop exhaust emitter to emit particles:

```
cars[0].smokeEmitter.on = false;
cars[1].smokeEmitter.on = false;
```

This is how we stop an emitter to emit particles.

> `emitter.on` property determines whether the emitter is currently emitting particles (`true`) or not (`false`).

Now we must stop new targets to appear, by removing the time event which generated them.

```
game.time.events.remove(this.targetLoop);
```

New targets won't appear anymore.

> `time.events.remove(event)` removes `event`.

When it's time to stop cars, we have to remember cars do not move vertically, because the illusion of the endless runner is given by moving targets.

To make players believe cars stopped, we will stop targets. They all are in `targetGroup` group so we only have to iterate through all group elements and set

their `velocity.y` property to zero.

```
for(var i = 0; i < this.targetGroup.length; i++){
     this.targetGroup.getChildAt(i).body.velocity.y = 0;
}
```

You already know how to create an explosion and tint its particles. That's what we are doing now.

```
var explosionEmitter = game.add.emitter(t.x, t.y, 200);
explosionEmitter.makeParticles("particle");
explosionEmitter.gravity = 0;
explosionEmitter.setAlpha(0.2, 1);
explosionEmitter.minParticleScale = 0.5;
explosionEmitter.maxParticleScale = 3;
explosionEmitter.start(true, 2000, null, 200);
explosionEmitter.forEach(function(p){
     p.tint = t.tint;
});
```

If we really want cars to stop at once, we have to stop their tweens too.

```
game.tweens.removeAll();
```

This is how we stop and remove all existing tweens.

> `tweens.removeAll` method removes all tweens running and doesn't call any of the tween `onComplete` events.

If target explodes, once we created the explosion we have to remove the target itself.

```
t.destroy();
```

We also stop the background music, as we only want to hear the explosion sound effect.

```
this.bgMusic.stop();
```

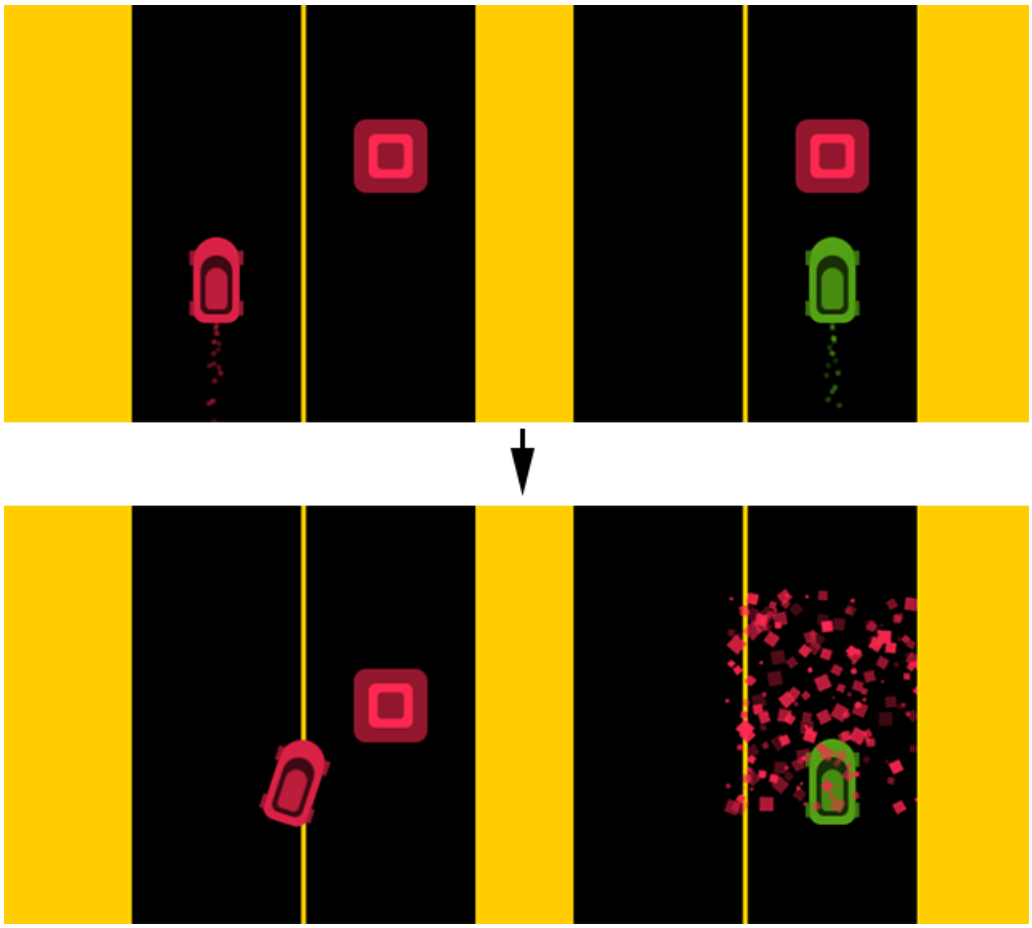
With the background music stopped, it's time to play the explosion sound effect.

```
var explosionSound = game.add.audio("explosion");
explosionSound.play();
```

And finally we'll wait two seconds before launching game over state.

```
game.time.events.add(Phaser.Timer.SECOND * 2, function(){
     game.state.start("GameOverScreen");
}, this);
```

Now you can test the game, try to collect targets and see what happens.



In the above picture, here's what happens when the green car collects the red

target. Boom, game over.

Game over state hasn't been written yet, so it's just a black screen at the moment, but don't worry, we'll create it after a couple more features.

Avoiding Targets

If you remember how we designed the game, we said player must collect all targets whose colors match cars colors.

This means if you do not collect a target which color matches the color of the car, it's game over.

To develop this feature, we'll introduce Phaser signals.

> A **Signal** is an event dispatch mechanism that supports broadcasting to multiple listeners.

In other words, a signal is a custom event listener. Just like we added listeners for player click or touch, or for certain keys being pressed, we can define custom listeners which are triggered by custom events and fire callback functions just like built-in Phaser event listeners.

The question now is: how we create such custom events? In `create` method inside `playGame` object, when we create a new target we have to add one line:

```
create: function(){
     // same as before
     this.targetLoop = game.time.events.loop(targetDelay, function(){
          for(var i = 0; i < 2; i++){
               var target = new Target(game, i);
               game.add.existing(target);
               this.targetGroup.add(target);
               target.missed.add(this.targetFail, this);
          }
     }, this);
}
```

This way each target has a `missed` listener which calls `targetFail` callback function in `this` context, just like in case of collision between a car and a wrong target.

This leads to another question: how can we define `missed` listener?

In `Target` class definition, we have to add a couple of lines:

```
Target = function (game, lane) {
     var position = game.rnd.between(0, 1);
     Phaser.Sprite.call(this, game, cars[lane].positions[position], -20,
       "target");
     game.physics.enable(this, Phaser.Physics.ARCADE);
     this.anchor.set(0.5);
     var tint = game.rnd.between(0, 1);
     this.mustPickUp = tint == lane;
     this.tint = cars[tint].tint;
     this.body.velocity.y = targetSpeed;
     this.missed = new Phaser.Signal();
};

Target.prototype = Object.create(Phaser.Sprite.prototype);
Target.prototype.constructor = Target;

Target.prototype.update = function() {
     if(this.y > game.height - this.height / 2 && this.mustPickUp){
          this.missed.dispatch(this);
     }
     if(this.y > game.height + this.height / 2){
          this.destroy();
     }
};
```

Let's explore them in detail:

```
this.missed = new Phaser.Signal();
```

`missed` property is a new Phaser signal. This is the constructor, this means with this line we declare there is a new listener attached to `missed` property.

We declared the listener and we made it active with a callback function to be called, we only have to fire it.

This is the code which fires the signal:

```
if(this.y > game.height - this.height / 2 && this.mustPickUp){
     this.missed.dispatch(this);
}
```

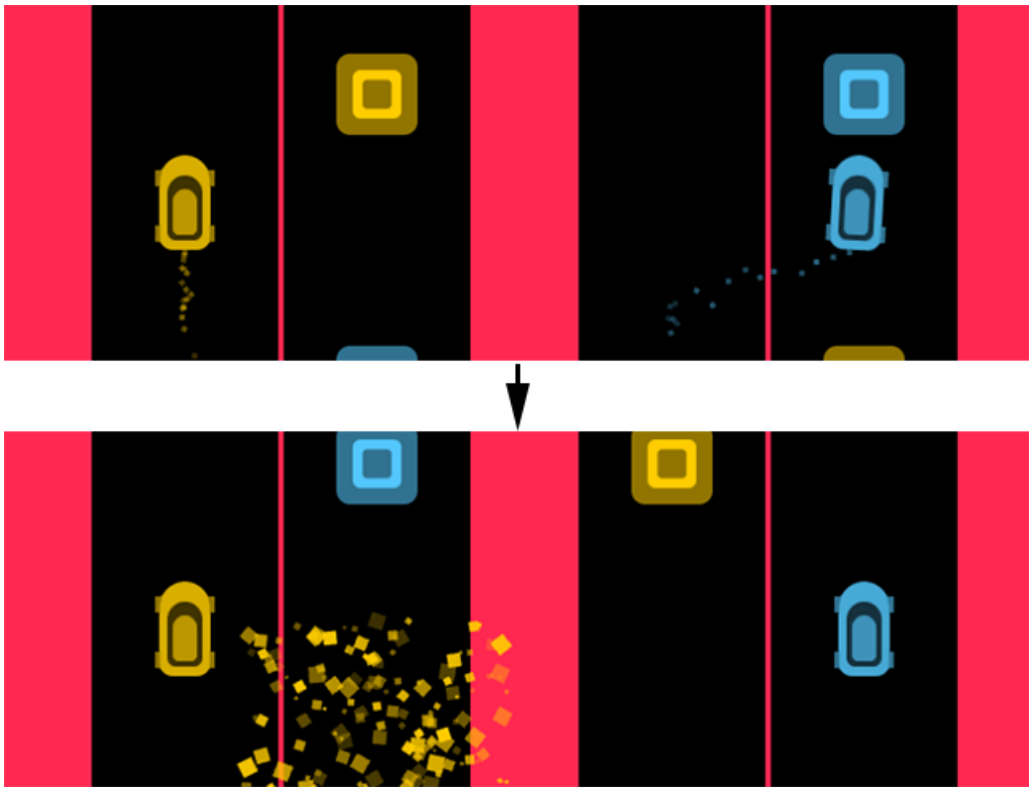
When the target is about to leave the stage to the bottom but `mustPickUp` property

is true `which` meaning the target must have been collected, we dispatch the signal passing the target itself as argument.

> `signal.dispatch(params)` dispatches `signal` signal to all listeners passing `params` parameters to each listener.

In other words, we have a `missed` listener attached to each target which will call `targetFail` function. The listener is fired when a target which must be picked up leaves the stage without being picked up.

And that's game over again. Test the game, and see what happens when you do not collect a target you should have collected.



Target explodes the same way as wrong targets do when they are collected, and that's quite obvious since it's the same snippet of code which handles both cases.

With this step, we are done with targets. Now we will add a scoring system, because as usual there's no point in playing a game if you can't make a high score.

Scoring

Handling player score at this time is very easy as we already defined a global `score` variable and we already have `scoreText` bitmap text showing current score.

At the moment it always shows zero because there isn't any way to increase score variable, and that's what we are going to do now.

Moreover, most of the code to handle saving the score on local storage has already been written, so let's focus on the score itself.

Scoring system in this game is quite easy: you gain one point each quarter second you survive.

In `create` method of `playGame` object, we need to add a time event to increase the score every 250 milliseconds, then update `scoreText` bitmap text.

```
create: function(){
     // same as before
     this.scoreLoop = game.time.events.loop(250, function(){
          score++;
          this.scoreText.text = score.toString();
     }, this);
}
```

Since we don't want score to keep increasing once the player collected a wrong target or missed a required target, we are stopping the score when we execute `targetFail` method.

```
targetFail: function(t){
     game.input.keyboard.removeKey(Phaser.Keyboard.Z);
     game.input.keyboard.removeKey(Phaser.Keyboard.X);
     cars[0].smokeEmitter.on = false;
     cars[1].smokeEmitter.on = false;
     game.time.events.remove(this.targetLoop);
     game.time.events.remove(this.scoreLoop);
     // same as before
}
```

As score is simply increased by a time event, removing the time event will stop the score to increase.

Run the game now and watch your score increase as you play and stop when you collect a wrong target or miss a required target.

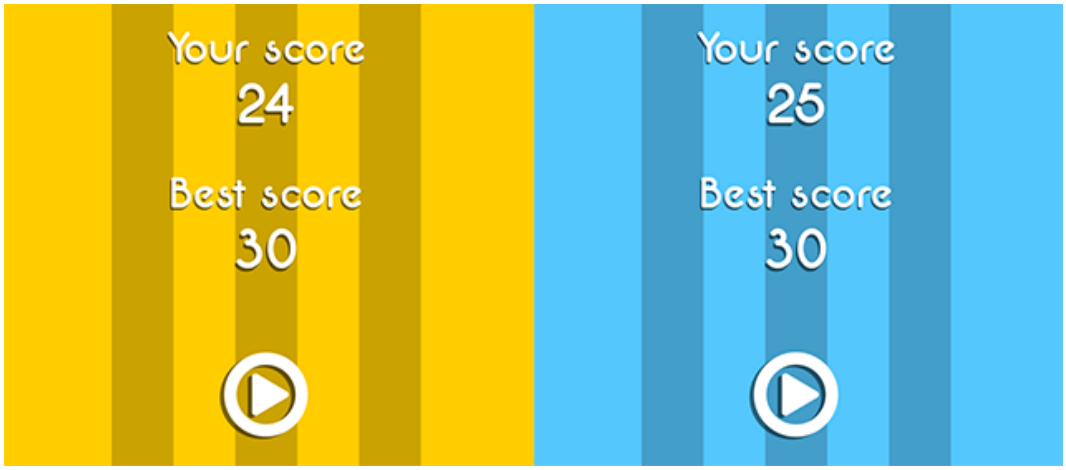
And this was the last core feature needed to create this game.

Now we only have to create the game over screen and the info screen with the instructions how to play.

Let's start from the screen you will never want to see.

Creating game over screen

The game over screen is exactly the same you created during the development of Rising Ships game, here is what you will get:



We have the random tint color, the bitmap text with both current and best score, a glowing “play” button and local storage management to saving the best score.

This is the content of `gameOverScreen` object:

```
var gameOverScreen = function(game){};
gameOverScreen.prototype = {
     create: function(){
          var bestScore = Math.max(score, savedData.score);
          var titleBG = game.add.tileSprite(0, 0, game.width, game.height,
               "backsplash");
          titleBG.tint = bgColors[game.rnd.between(0, bgColors.length - 1)];
          document.body.style.background = "#"+titleBG.tint.toString(16);
          game.add.bitmapText(game.width / 2, 50 , "font", "Your score",
               90).anchor.x = 0.5;
          game.add.bitmapText(game.width / 2, 150 , "font", score.toString(),
               120).anchor.x = 0.5;
          game.add.bitmapText(game.width / 2, 350 , "font", "Best score",
               90).anchor.x = 0.5;
          game.add.bitmapText(game.width / 2, 450 , "font", bestScore.toString(),
               120).anchor.x = 0.5;
          localStorage.setItem(localStorageName,JSON.stringify({
               score: bestScore
          }));
          var playButton = game.add.button(game.width / 2, game.height - 150,
               "playbutton", this.startGame);
          playButton.anchor.set(0.5);
          var tween = game.add.tween(playButton).to({
               width: 220,
               height:220
          }, 1500, "Linear", true, 0, -1);
          tween.yoyo(true);
     },
     startGame: function(){
          game.state.start("PlayGame");
     }
}
```

As you can see it's exactly the same code used to create the game over screen in Rising Ships. So we can move on to the last screen we need to add.

Creating how to play screen

The creation of a good info screen is very important as players will know how to interact with the game and won't quit the game because they don't know what to do with it.

You are about to see a lot of code now, but don't worry as there isn't absolutely anything new, I just reused a few concepts you already know:

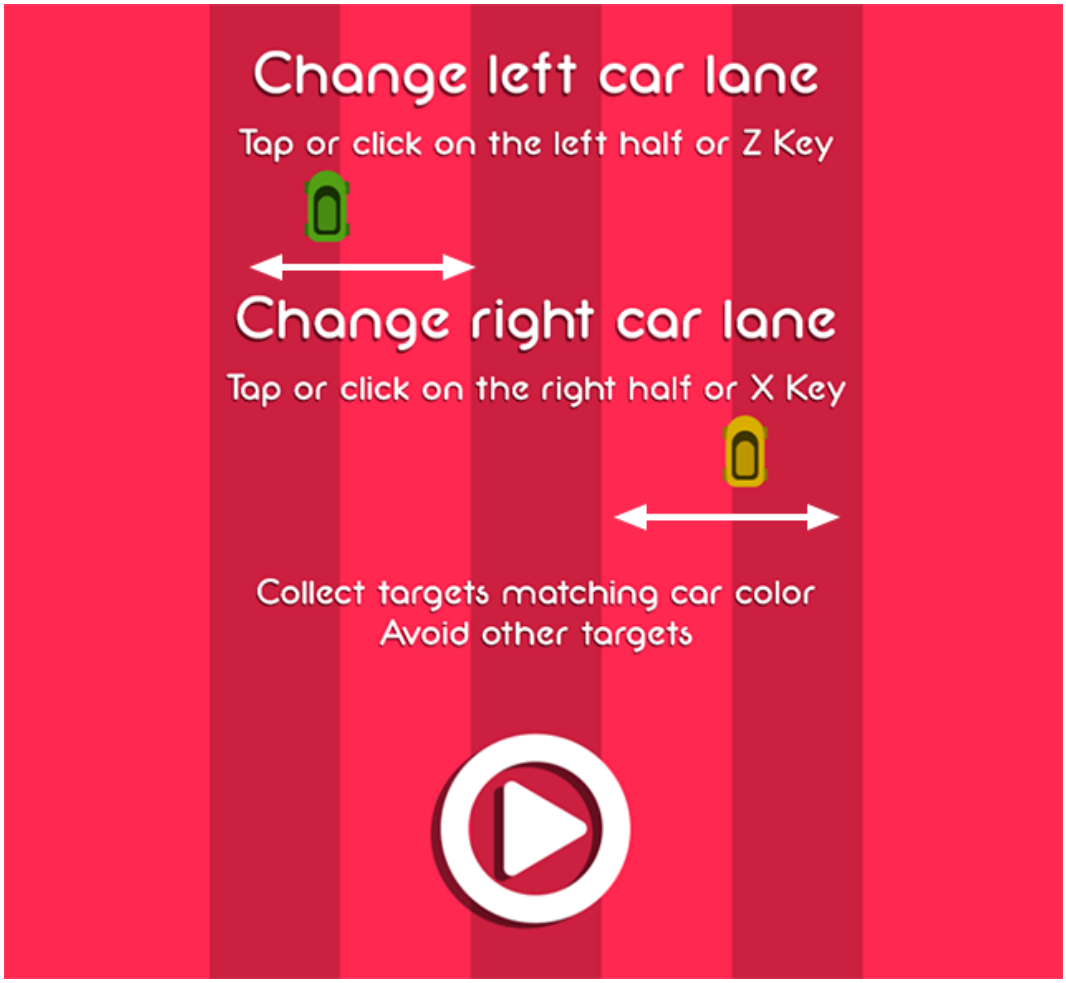
```
var howToPlay = function(game){};
howToPlay.prototype = {
     create: function(){
          var titleBG = game.add.tileSprite(0, 0, game.width, game.height,
               "backsplash");
          var tintColor = bgColors[game.rnd.between(0, bgColors.length - 1)];
          titleBG.tint = tintColor;
          var pickedColors = [tintColor];
          document.body.style.background = "#"+titleBG.tint.toString(16);
          game.add.bitmapText(game.width / 2, 40 , "font", "Change left car lane",
               60).anchor.x = 0.5;
          game.add.bitmapText(game.width / 2, 120 , "font", "Tap or click on the
               left half or Z Key", 36).anchor.x = 0.5;
          game.add.bitmapText(game.width / 2, 280 , "font", "Change right car
               lane", 60).anchor.x = 0.5;
          game.add.bitmapText(game.width / 2, 360 , "font", "Tap or click on the
               right half or X Key", 36).anchor.x = 0.5;
          game.add.bitmapText(game.width / 2, 560 , "font", "Collect targets
               matching car color", 36).anchor.x = 0.5;
          game.add.bitmapText(game.width / 2, 600 , "font", "Avoid other targets",
               36).anchor.x = 0.5;
          var leftCar = game.add.sprite(game.width / 2 - 250, 200, "car");
          leftCar.anchor.set(0.5);
          do{
               tintColor = bgColors[game.rnd.between(0, bgColors.length - 1)];
          } while (pickedColors.indexOf(tintColor) >= 0)
          leftCar.tint = tintColor;
          pickedColors.push(tintColor);
          var leftCarTween = game.add.tween(leftCar).to({
               x: game.width / 2 - 50
          }, 500, "Linear", true, 0, -1);
          leftCarTween.yoyo(true);
          var rightCar = game.add.sprite(game.width / 2 + 250, 440, "car");
          rightCar.anchor.set(0.5);
          do{
               tintColor = bgColors[game.rnd.between(0, bgColors.length - 1)];
          } while (pickedColors.indexOf(tintColor) >= 0)
          rightCar.tint = tintColor;
          var rightCarTween = game.add.tween(rightCar).to({
               x: game.width / 2 + 50
          }, 500, "Linear", true, 0, -1);
          rightCarTween.yoyo(true);
          var playButton = game.add.button(game.width / 2, game.height - 150,
               "playbutton", this.startGame);
          playButton.anchor.set(0.5);
          var tween = game.add.tween(playButton).to({
               width: 220,
               height:220
          }, 1500, "Linear", true, 0, -1);
          tween.yoyo(true);
     },
     startGame: function(){
          game.state.start("PlayGame");
     }
}
```

A whole page of code. Does it look scary? It shouldn't.

First, run the game and see what you get once you change `startGame` method in `titleScreen` object to launch `HowToPlay` state:

```
startGame: function(){
     game.state.start("HowToPlay");
}
```

Now, this is what you should see once you click on “play” button in the title screen:



We have a random background color, some texts written with bitmap fonts, two moving cars and a glowing “play” button which leads to the game itself.

Let's break the code in pieces and see it more in detail:

```
var titleBG = game.add.tileSprite(0, 0, game.width, game.height, "backsplash");
var tintColor = bgColors[game.rnd.between(0, bgColors.length - 1)];
titleBG.tint = tintColor;
var pickedColors = [tintColor];
document.body.style.background = "#"+titleBG.tint.toString(16);
```

This is the snippet which adds the background tile sprite and selects a random background color to tint it.

Look at `pickedColors` array to store picked colors and avoid to choose the same random color twice as we already did in play game state.

```
game.add.bitmapText(game.width / 2, 40, "font", "Change left car lane",
       60).anchor.x = 0.5;
game.add.bitmapText(game.width / 2, 120, "font", "Tap or click on the
       left half or Z Key", 36).anchor.x = 0.5;
game.add.bitmapText(game.width / 2, 280, "font", "Change right car
       lane", 60).anchor.x = 0.5;
game.add.bitmapText(game.width / 2, 360, "font", "Tap or click on the
       right half or X Key", 36).anchor.x = 0.5;
game.add.bitmapText(game.width / 2, 560, "font", "Collect targets
       matching car color", 36).anchor.x = 0.5;
game.add.bitmapText(game.width / 2, 600, "font", "Avoid other targets",
       36).anchor.x = 0.5;
```

All the texts you see in this screen are written by these lines, with different font size and position.

```
var leftCar = game.add.sprite(game.width / 2 - 250, 200, "car");
leftCar.anchor.set(0.5);
do{
     tintColor = bgColors[game.rnd.between(0, bgColors.length - 1)];
} while (pickedColors.indexOf(tintColor) >= 0)
leftCar.tint = tintColor;
pickedColors.push(tintColor);
var leftCarTween = game.add.tween(leftCar).to({
     x: game.width / 2 - 50
}, 500, "Linear", true, 0, -1);
leftCarTween.yoyo(true);
```

This is the code which handles the left car.

It's placed on the canvas, tinted with a random color then moved from left to right and from right to left with a yoyo tween.

The `do while` loop ensures the car will have a random color which does not match with background color.

```
var rightCar = game.add.sprite(game.width / 2 + 250, 440, "car");
rightCar.anchor.set(0.5);
do{
     tintColor = bgColors[game.rnd.between(0, bgColors.length - 1)];
} while (pickedColors.indexOf(tintColor) >= 0)
rightCar.tint = tintColor;
var rightCarTween = game.add.tween(rightCar).to({
     x: game.width / 2 + 50
}, 500, "Linear", true, 0, -1);
rightCarTween.yoyo(true);
```

The same concept applies to right car.

```
var playButton = game.add.button(game.width / 2, game.height – 150, "playbutton",
       this.startGame);
playButton.anchor.set(0.5);
var tween = game.add.tween(playButton).to({
     width: 220,
     height:220
}, 1500, "Linear", true, 0, -1);
tween.yoyo(true);
```

And finally the good old glowing “play” button which calls `startGame` method launching the game itself.

Now the game is complete and ready to be played.

I am sure building this second game was way easier than making Rising Ships.

As you continue to learn and practice with Phaser, building games will be easier and easier.

By the way, building a game is not just “make stuff work”, we also have to keep an eye on resource management, especially when we plan to port our games on mobile devices, which may not have the faster CPU.

Saving resources by using object pooling

According to boring theory, we call object pooling the technique which uses a set of initialized objects kept ready to use – a "pool" – rather than allocating and

destroying them on demand.

In our game, each time you collect a good target, or each time a wrong target leaves the stage off the bottom, their sprites are destroyed.

Each time a new target appears from the top of the screen, a new sprite is created.

This happens a lot of times during a game play, and although I am sure Phaser has a good memory management and garbage collection, this can be very resource-consuming in the long run. So the idea is never to delete removed/collected targets, which will be temporarily stored in a repository (in this case an array) until a new target is needed, and that's when we just recover the previously stored target.

It will work. Follow me. Let's start with the creation of a new global variable:

```
var game;
var localStorageName = "doublelanegame";
var bgColors = [0x54c7fc, 0xffcd00, 0xff2851, 0x62bd18];
var score;
var savedData;
var laneWidth = 138;
var lineWidth = 4;
var cars = [];
var carTurnSpeed = 200;
var targetDelay = 1200;
var targetSpeed = 180;
var targetPool = [];
```

As you can see, the “pool” is just an array.

We are going to initialize this array at each game play, setting it as an empty array. No targets in the pool when we start playing.

```
var playGame = function(game){};
playGame.prototype = {
    create: function(){
        targetPool = [];
        targetPool.length = 0;
        // same as before
    }
    // same as before
}
```

Once the pool has been initialized in `create` method of `playGame` object, we are

going to add a new method to Target class, to be called each time we need to destroy a target. Oh, and we won't talk about “destroying” anymore. When you destroy something, it does not exist anymore.

We are going to kill the target.

```
Target.prototype.prepareToDie = function(){
    this.kill();
    targetPool.push(this);
    console.log("target killed. Targets in the pool: " + targetPool.length)
}
```

As you can see, when `prepareToDie` method is called, the target is killed then placed in `targetPool` array. Then we just prompt what we did on the console.

> `kill` method kills a Game Object. A killed Game Object has its `alive`, `exists` and `visible` properties all set to `false`. Killing a Game Object is a way for you to quickly recycle it in an object pool, as it doesn't destroy the object or free it up from memory.

Now we know how to kill a target. Next question is: when are we going to kill it?

Obviously each time we would have destroyed it.

This is `update` method of `Target` class featuring object pooling:

```
Target.prototype.update = function() {
    if(this.alive && this.y > game.height - this.height / 2){
        if(this.mustPickUp){
            this.missed.dispatch(this);
        }
        this.prepareToDie();
    }
};
```

Before checking if the target left the screen off the bottom, we see if it's alive. Then we call `prepareToDie` method rather than destroying it like before.

> `alive` property checks if a Game Object is alive or dead.

`prepareToDie` method is also called inside update method of `playGame` object when we need to remove a target:

```
update: function(e){
    cars[0].smokeEmitter.x = cars[0].x;
    cars[1].smokeEmitter.x = cars[1].x;
    game.physics.arcade.collide(this.carGroup, this.targetGroup, function(c, t){
        if(t.alive){
            if(c.tint == t.tint){
                this.hitSound.play();
            }
            else{
                this.targetFail(t);
            }
            t.prepareToDie();
        }
    }, null, this);
}
```

Look as I always check for `alive` property.

Now we have to remove the line which destroys a target from `targetFail` method, as we kill the target no matter if the collected target is right or wrong.

```
targetFail: function(t){
    // same as before
    // t.destroy() <- remove this
    // same as before
}
```

From now on, when you run the game no targets will be destroyed anymore, as they will fill `targetPool` array.

What to do with these targets? Let's add another method to Target class to make them revive:

```
Target.prototype.prepareToRevive = function(lane){
    var position = game.rnd.between(0, 1);
    this.reset(cars[lane].positions[position], -20);
    var tint = game.rnd.between(0, 1);
    this.mustPickUp = tint == lane;
    this.tint = cars[tint].tint;
    this.body.velocity.y = targetSpeed;
    console.log("target revived. Targets in the pool: " + targetPool.length)
}
```

The code is almost the game as the one used in the constructor, with the exception we don't create a target as a new sprite with physics and signals attached, but

rather we reset it to its new position – which is determined in the same way we did in the constructor.

> `reset(x, y)` method resets the Game Object moving it to the given `x`/`y` world coordinates settings its `alive`, `exists` and `visible` properties to `true`.

When to call this method?

Each time we need a new target and we have one in the object pool. Look at the changes to create method of `playGame` object:

```
create: function(){
    // same as before
    this.targetLoop = game.time.events.loop(targetDelay, function(){
        for(var i = 0; i < 2; i++){
            if(targetPool.length == 0){
                console.log("target created from scratch");
                var target = new Target(game, i);
                game.add.existing(target);
                this.targetGroup.add(target);
                target.missed.add(this.targetFail, this);
            }
            else{
                var target = targetPool.pop();
                target.prepareToRevive(i);
            }
        }
    }, this);
    // same as before
}
```

We are inside the looped time event which creates two new targets each `targetDelay` milliseconds.

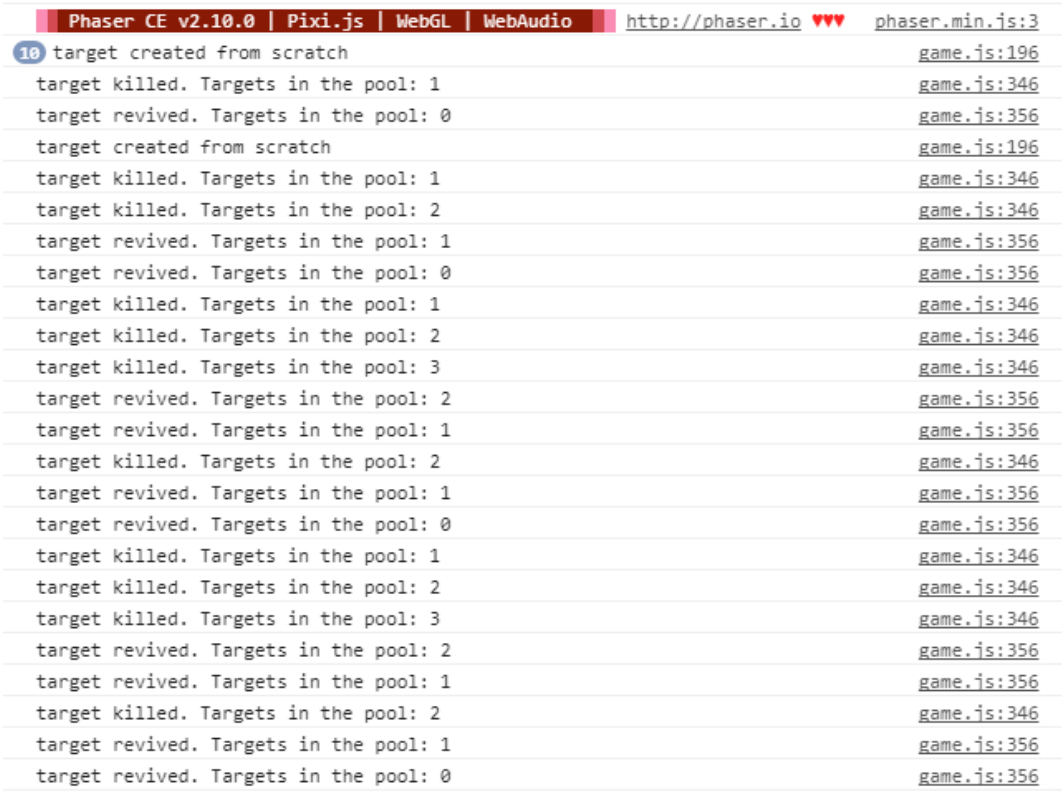
The first thing we check is if `targetPool` is populated.

If `targetPool` is empty, then we will create a target the same old way we are used to, I am just prompting it in the console.

If we find a target in `targetPool`, we remove it from the array and execute `prepareToRevive` method on it. The `i` argument is the lane, just like when we use the constructor.

> `pop` method removes the last element from an array and returns that element. This method changes the length of the array.

Now run the game. You wont' see anything different, but if you open the console, that's what you will see:



First targets are created from scratch, then as you progress into the game, more and more targets are picked from the pool until no new targets are created anymore.

This is the power of object pooling.

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 adding a new friendly barrier which gives you five seconds of invisibility if you hit it.

Thank you and let's keep in touch

Now you finished the book.

It's my second self-published book after three books written under a publishing label and one booklet, so I apologize if you found some errors. I am continuously updating and fixing my self published book so I hope sooner or later this book will be exactly what you expect.

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 **Norman Rozental** and to the guys at **Ketchapp Studio** for making such fun games.

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

Emanuele.