Game development with Phaser

Creating a Virtual Pet game in 5 steps

Nacho Iborra

IES San Vicente

Table of Contents

1. Introduction

Phaser is a lightweight 2D Javascript framework that can be used to create video games that can be run in HTML5 web pages. Besides, with some external tools such as Apache Cordova or PhoneGap, you can easily export this video games to some known platforms, such as iOS or Android devices.

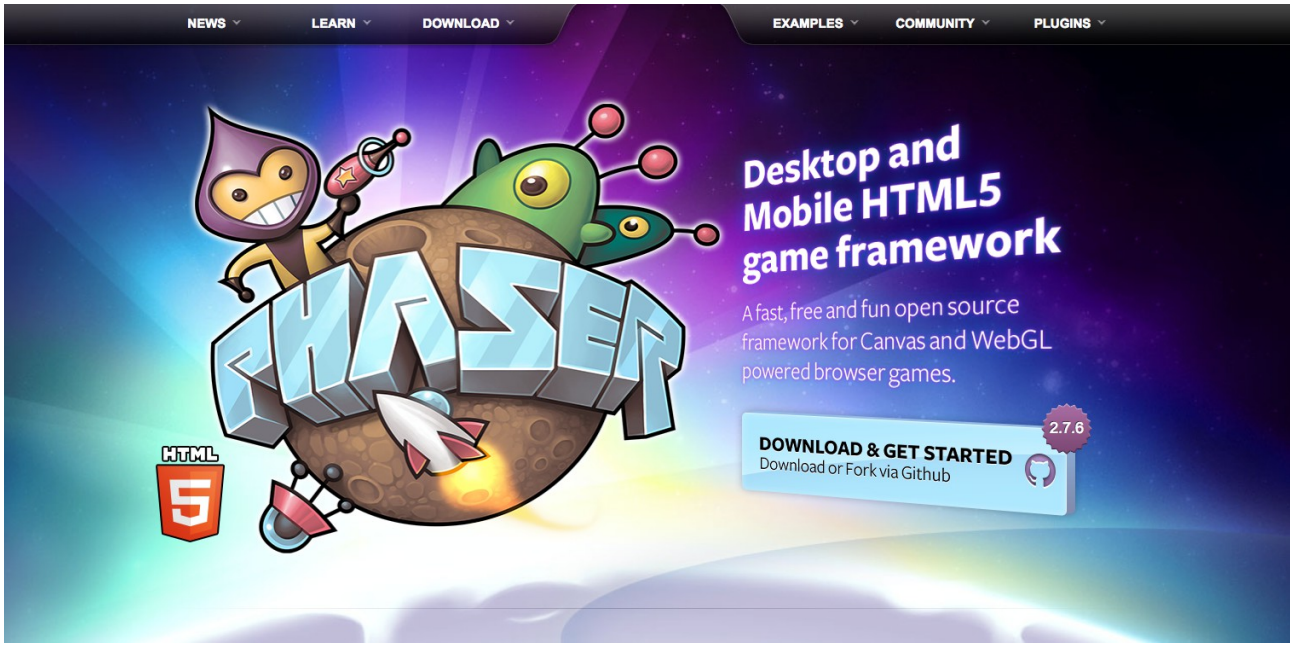
Among all its features, we can highlight the following ones:

- It uses WebGL library to render video games, but it can also rely on HTML5 *canvas* element to render them too (for devices that do not support WebGL).
- It has some basic implementations for some physics and sprite handling (rotations, collisions, gravity and so on)
- It also has some advanced features, such as particles system, a camera to look at or follow any element of the game, etc.

1.1. Downloading Phaser

You can download Phaser from its official web site:

http://phaser.io/



Go to the *Download* section in the upper menu, and download either the whole project in *zip* or *tar.gz* format, or just the library itself (*js* or *min.js* file). For the purpose of this tutorial, it is enough with the library (preferably *min.js* file).

1.2. What else do we need?

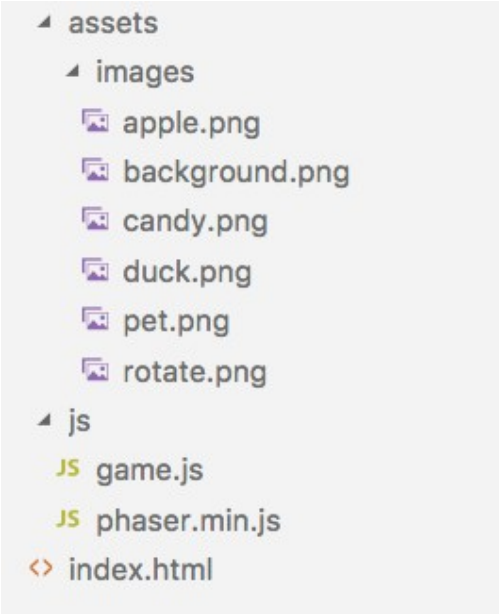
Apart from Phaser library, we may need to have a web server installed, so that our video game source code can access all the resources that it needs (images, sounds and so on). In some cases you may not need this local server (specially if you work with HTML5 canvas instead of WebGL).

Anyway, if you want to install a web server, or you have already installed one of them, you can use it. Some alternatives are:

- Brackets (you can get it here). This is a quite simple developing environment that lets us edit our source code and launch a local web server.
- Apache (by installing XAMPP or Apache server, depending on your operating system). This is an external web server, so we would need a code editor (VisualStudio Code, Notepad++ or something similar).

1.3. Downloading and checking the project template

Download the project resources from the platform and have a look at the project structure:



- In the **assets** folder we will place any additional file for the video game, such as images, sounds and so on. There is an *images* folder with all the images that we will need for this game.
- In the **js** folder we will place all the Javascript code, including Phaser library (already included) and our game source code (*game.js*, which is empty for now).

Regarding the HTML file, it justs loads the Javascript files and adds some CSS style to have a black background all over the page. It also configures the *viewport* so that it can be displayed properly in mobile devices.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1,
            maximum-scale=1, minimum-scale=1, user-scalable=no" />
        <title>Virtual Pet</title>
        <script type="text/javascript" src="js/phaser.min.js"></script>
        <style>
            body {
                padding:0px;
                margin:0px;
                background: black;
```

```
            }
        </style>
    </head>
    <body>
        <script type="text/javascript" src="js/game.js"></script>
    </body>
</html>
```

2. Game development

All the following steps are focused on *game.js* source file, so all the changes that you will make to the game in this example will only affect this file.

2.1. Setting the game structure and default methods

To begin with, let's define our *Phaser* game object, with the screen resolution, and the default methods that will be fired as game starts and runs.

```
var GameState = {

    init: function() {
      this.scale.scaleMode = Phaser.ScaleManager.SHOW_ALL;
      this.scale.pageAlignHorizontally = true;
      this.scale.pageAlignVertically = true;
    },

    preload: function() {

    },

    create: function() {

    },

    update: function() {

    }
};

var game = new Phaser.Game(360, 640, Phaser.CANVAS);

game.state.add('GameState', GameState);
game.state.start('GameState');
```

A Phaser video game is made of states. Each one may correspond to a different screen. In this case, we are going to manage one single screen. Inside each state, we can implement some default methods:

- *init* is executed once, when the game starts. It typically sets the game scaling and alignment.
- *preload* is executed before the game is shown. It usually loads the images and sounds so that we have them ready to be drawn or played.

- *create* is executed when the state is initially shown. We can use this function to draw all the initial elements in the screen, and define some events on them.
- *update* is executed periodically (multiple times per second), so we will use it to define some periodical instructions (transformations, state updates, and so on).

In this case we will just see a 360 x 640 video game (with a black background), displayed on a canvas renderer. We can use `Phaser.AUTO` instead of `Phaser.CANVAS` to auto-detect the best renderer according to our current browser (although with this value we may need to run the game on a web server).

2.2. Drawing images

Let's start by loading all the necessary images in our *preload* function. For each image, we associate each one to a given name:

```
preload: function() {
    this.load.image('background', 'assets/images/background.png');
    this.load.image('apple', 'assets/images/apple.png');
    this.load.image('candy', 'assets/images/candy.png');
    this.load.image('duck', 'assets/images/duck.png');
},
```

Then, we can draw them in the *create* function, using the associated name of each one:

```
create: function() {
    this.background = this.game.add.sprite(0, 0, 'background');

    this.apple = this.game.add.sprite(72, 570, 'apple');
    this.apple.anchor.setTo(0.5);

    this.candy = this.game.add.sprite(144, 570, 'candy');
    this.candy.anchor.setTo(0.5);

    this.toy = this.game.add.sprite(216, 570, 'duck');
    this.toy.anchor.setTo(0.5);
},
```

See how to use the *game.add.sprite* method to add each sprite in its corresponding coordinates (X and Y). The *anchor.setTo* method sets the anchor point of each image (in this case, it sets it in the middle of the image, so that we draw each image in the coordinates corresponding to this anchor point, and we can even rotate the image from this anchor point as well.

2.2.1. Dealing with a sprite sheet

If you pay attention to the pet image, it is a sprite sheets with some sprites that we will use to animate the pet later. For now, let's add the sprite sheet in the *preload* method this way:

```
preload: function() {
    ...
    this.load.spritesheet('pet', 'assets/images/pet.png', 97, 83, 5, 1, 1);
```

```
},
```

The parameters of this function are:

- Width of each sprite in pixels
- Height of each sprite in pixels
- Number of sprites
- Margin of the sprite sheet in pixels
- Separation between sprites in pixels

Then, we draw the first sprite of the sheet in the *create* method:

```
create: function() {
    ...
    this.pet = this.game.add.sprite(100, 400, 'pet');
    this.pet.anchor.setTo(0.5);
},
```

You can try the game now and see how all images are shown:

2.3. Adding events: drag and select objects

2.3.1. Dragging the pet

Let's add an event to drag the pet if we touch and drag it on the screen. To do this, we need to enable the input on the pet (*create* method), and then enable the dragging:

```
create: function() {
   ...
    this.pet.inputEnabled = true;
    this.pet.input.enableDrag();
}
```

You can try the game again and drag the pet all over the game screen.

2.3.2. Placing the objects

We are going to place some objects in the screen (candies, apples, ducks), so that if we touch them, and then touch in the screen, we will "copy" them in the selected location. To do this, we just enable the input on each object, and call a given method (we call it *pickItem,* for instance) whenever we touch them:

```
create: function() {
    ...
    this.apple.inputEnabled = true;
    this.apple.events.onInputDown.add(this.pickItem, this);
    this.candy.inputEnabled = true;
    this.candy.events.onInputDown.add(this.pickItem, this);
    this.toy.inputEnabled = true;
    this.toy.events.onInputDown.add(this.pickItem, this);

    this.objects = [this.apple, this.candy, this.toy];
    this.selectedItem = null;
}
```

Then, we define the *pickItem* method inside the game state, next to *init, preload, create* and *update* methods. With this method, we just mark the selected item with an alpha transparency

```
pickItem: function(sprite, event) {
    // Clear previous selection
    this.objects.forEach(function(element, index){
        element.alpha = 1;
    });
    this.selectedItem = null;

    // Select current sprite
    sprite.alpha = 0.4;
    this.selectedItem = sprite;
}
```

Finally, we enable the touch input in the screen (background), so that whenever we touch it, we will place a copy of the currently selected item:

```
create: function() {
    ...
    this.background.inputEnabled = true;
    this.background.events.onInputDown.add(this.placeItem, this);
}
```

And this is the *placeItem* method to place the item in the chosen position:

```
placeItem: function(sprite, event) {
    if(this.selectedItem) {
```

```
        var x = event.position.x;
        var y = event.position.y;

        var newItem = this.game.add.sprite(x, y, this.selectedItem.key);
        newItem.anchor.setTo(0.5);
    }
}
```

2.4. Animating the pet

Let's play an animation and movement on the pet every time we place a new object in the screen.

First of all, we can define the animation frame sequence in our *create* method. In this case, the pet will show sprites 1, 2, 3, 2, 1, at 7 frames per second, only once (*false,* which means 'not periodically'). This is what *animations.add* parameters mean. We also associate a name to this animation:

```
create: function() {
    ...
    this.pet.animations.add('funnyfaces', [1, 2, 3, 2, 1], 7, false);
}
```

Let's move to our *placeItem* method: whenever we place an item, we are going to:

- Move the pet towards the item
- Remove (destroy) the item once the pet reaches its position
- Play previous animation to eat the item

```
placeItem: function(sprite, event) {
    if(this.selectedItem) {
        var x = event.position.x;
        var y = event.position.y;

        var newItem = this.game.add.sprite(x, y, this.selectedItem.key);
        newItem.anchor.setTo(0.5);

        // Move pet towards the item (tween animation)
        var petMovement = this.game.add.tween(this.pet);
        petMovement.to({x: x, y: y}, 700);
        petMovement.onComplete.add(function(){

            //destroy the apple/candy/duck
            newItem.destroy();

            //play animation
            this.pet.animations.play('funnyfaces');
```

```
        }, this);

        petMovement.start();
    }
}
```

The tween animation to move the pet towards the item will last 700 milliseconds. Once it is completed, the item will be destroyed, and the "funnyfaces" animation defined above will be played.

2.5. Updating game state

Whenever the pet eats an item, it will add or lose some health and fun. Initially, these two properties will be set to 100 (in *create* method). Besides, each item will have some custom parameters to increase/decrease these properties:

```
create: function() {
    ...
    this.pet.customParams = {health: 100, fun: 100};
    this.apple.customParams = {health: 20, fun: 0};
    this.candy.customParams = {health: -10, fun: 10};
    this.toy.customParams = {health: 0, fun: 20};
}
```

We need to draw these properties in the screen, so let's learn how to write some text. We define an auxiliary method to draw the texts (health and fun):

```
refreshStats: function() {
    this.healthText.text = this.pet.customParams.health;
    this.funText.text = this.pet.customParams.fun;
}
```

And call it from *create* method... (we also define *healthText* and *funText* properties there):

```
create: function() {
    ...
    var style = { font: '20px Arial', fill: '#fff'};
    this.game.add.text(10, 20, 'Health:', style);
    this.game.add.text(140, 20, 'Fun:', style);

    this.healthText = this.game.add.text(80, 20, '', style);
    this.funText = this.game.add.text(185, 20, '', style);

    this.refreshStats();
}
```

... and also whenever the pet eats anything:

```
placeItem: function(sprite, event) {
    if(this.selectedItem) {
```

```
        var x = event.position.x;
        var y = event.position.y;

        var newItem = this.game.add.sprite(x, y, this.selectedItem.key);
        newItem.customParams = this.selectedItem.customParams;
        newItem.anchor.setTo(0.5);

        // Move pet towards the item (tween animation)
        var petMovement = this.game.add.tween(this.pet);
        petMovement.to({x: x, y: y}, 700);
        petMovement.onComplete.add(function(){

            this.pet.customParams.health += newItem.customParams.health;
            this.pet.customParams.fun += newItem.customParams.fun;
            this.refreshStats();

            //destroy the apple/candy/duck
            newItem.destroy();

            //play animation
            this.pet.animations.play('funnyfaces');

        }, this);

        petMovement.start();
    }
}
```

2.5.1. Decreasing health and fun periodically

In order to make our game more interesting, let's decrease the health and fun values every 5 seconds. To do this, we set a timer in the *create* method:

```
create: function() {
    ...
    this.statsDecreaser = this.game.time.events.loop(Phaser.Timer.SECOND * 5,
        this.reduceProperties, this);
}
```

And then define the *reduceProperties* method:

```
reduceProperties: function() {
    this.pet.customParams.health -= 10;
    this.pet.customParams.fun -= 15;
    this.refreshStats();
```

```
}
```

2.5.2. Setting the game over condition

Let's move to our *update* method. In this method, we are going to periodically check if pet's health or fun levels are 0. If so, the game is over, and we are going to restart it:

```
update: function() {
    if(this.pet.customParams.health <= 0 || this.pet.customParams.fun <= 0) {
        this.pet.frame = 4;

        this.game.time.events.add(2000, function() {
            this.game.state.restart();
        }, this);
    }
},
```

We just set pet's sprite to the last one, and after 2 seconds, we call a function that restarts the game state. We could also move to another game state with a "Game over" screen or something like that.