ZENVA

Game Development for Human Beings

Build Cross-Platform Games with Phaser

Kristen Dyrr

Pablo Farias Navarro

Renan Oliveira

Ben Sparks

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Table of Contents

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Introduction by Pablo Farias Navarro, founder of ZENVA

If you are fearing this will be a cheesy and uninteresting introduction, I'm afraid to tell you that might indeed be the case. Even though I'm lucky enough to have the chance to communicate with an ever-growing group of +150,000 people on a weekly basis via our mailing list and social channels, there aren't many occasions in which I get to explain what it is we actually do at Zenva and what is this all for.

Back in the 90's when I was a teenager living in the middle of the Atacama Desert (Chile), I taught myself programming with the intention of making games. Resources weren't readily available as they are today, so the process was quite hard. However, the feeling of wonder and possiblities that opened every time I got something to work made all the effort worth it's while.

Eventually, I moved on and buried this hobby to take on "more serious" career paths (my own "lost decade"). Fortunately, not all hope was lost.

Flash-forwarding to 2011, I came across some pioneer HTML5 game development demos and they blew my mind. The possibility of making games with JavaScript without having to install IDE's or compile code. I couldn't say no to that, so I immersed myself in this new technology and started building stuff again.

In 2012 I started teaching online how to make games with different HTML5 frameworks. It was meant to be only a side-project, but the response was so overwhelmingly positive that in 2013 it became my full-time occupation.

A lot has happened since then. From a personal project, to a company with a highly skilled international team, two successful Kickstarter campaigns and over 150,000 students from over 200 countries.

Yet the spirit remains the same. To give people the same feeling of wonder I experienced in the 90's, every time I got a sprite to show on the screen, or a game character to move. The feeling of possibilities, exploration and creativity.

One of the main ideas behind the Phaser game library according to it's creator Richard Davey, is to create explorers not experts.

Let's go and explore then! Enjoy the book and never stop learning!

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

The Complete Mobile Game Development Course – Platinum Edition

Learn to create 15 games using JavaScript and Phaser in the Internet's most comprehensive course on game development. Learn more.

Advanced Game Development with Phaser

Become a Black Belt Phaser developer and create advanced games. Multiplayer included. Learn more.

Zenva for Schools – Coding Courses for High School Students
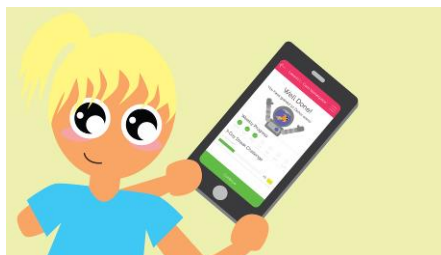
Zenva for Schools is an online platform for high schools, which includes interactive courses in programming, web development and game development. Learn more.

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

HTML5 Phaser Tutorial – SpaceHipster, A Space Exploration Game
By Pablo Farias Navarro

They say space is the final frontier. We haven't got very far when it comes to exploring it, but what if you could bypass this sad reality by playing a retro-style space exploration game called SpaceHipster?

Moreover, what if you could learn how to make this simple game from scratch using the awesome Phaser framework?
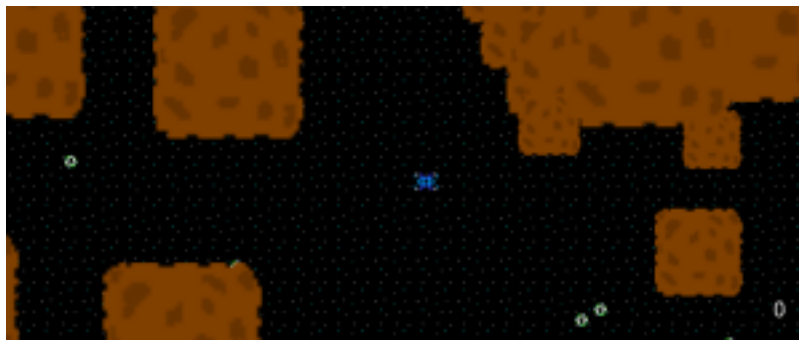
Phaser is a fun, free and fast HTML5 2D game framework created by **Richard Davey** and supported by a great community of developers who usually hang out at the HTML5GameDevs forums.

Source code files

You can grab them from here. If you want us to notify you when we release new tutorials on Phaser or HTML5 game development don't forget to click "subscribe to our list" when downloading the files (if you choose not to you won't hear from us!).

Tutorial goals

1 Learn the basics of Phaser by creating simple space exploration game (called SpaceHipster) that runs great on desktop and mobile.

2 Learn some basic 2D game mechanics.

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Tutorial requirements

-Basic to intermediate knowledge of JavaScript. If you need a refreshment, feel free to check our JavaScript video course at Zenva Academy.

-A code editor or IDE. Current Pablo prefers Sublime Text, but past Pablo has shown more affection for other IDE's and I'm sure future Pablo will have a say too.

-Download Phaser from it's Github repo. You can either clone the repo or download the ZIP file.

-You need to run the code and Phaser examples using a local or remote web server. Some popular options are Apache (WAMP if in Windows, MAMP if in Mac). A lightweight alternatives are Mongoose web server and Python's HTTP server. Take a look at this guide for more details.

-Download the full tutorial source code and game assets here. If you want us to let you know when we release new tutorials on Phaser or HTML5 game development don't forget to subscribe to our list!

-Have the documentation and the examples page at hand. Also don't forget that you can always find answers by looking at the source code of Phaser.

New Project

Let's begin with a new folder, an index file and the Phaser file, which you can grab from the "build" directory of the downloaded Phaser zip or cloned repository.

If you are on development mode (as opposed to production, which would be deploying your game to it's final destination for the world to play it) I recommend including the non-minified phaser.js file (as opposed to phaser.min.js). The reason being, it's not a good practice to treat your game engine as a black box. You want to be able to explore the contents of the file and debug properly. Not all the answers will be on Google so a lot of times the best way to do is just read the original source code to understand what's going on.

Our index.html file will look like this:

```
1 Game <script src="js/phaser.js"
type="text/javascript"></script>
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

There is nothing to see/show yet!

States

In Phaser, all the action occurs around *States*. You can think of them as main moments of your game. Think of a soccer game, you have a State when doors are open and people start coming in. Then you have a State when the pre-game show takes place. Then a State where the pre-game show stuff is removed from the field. Then a State when the game begins, and so forth.
Phaser gives you a lot of flexibility as what States you can have, but there is a de-facto convention which is used in many games and tutorials. The naming might vary a bit but it's usually something like:

Boot State: general game settings are defined, and the assets of the preloading screen are loaded (example the loading bar). Nothing is shown to the user.

Preload State: the game assets (images, spritesheets, audio, textures, etc) are loaded into the memory (from the disk). The preloading screen is shown to the user, which usually includes a loading bar to show the progress.

MainMenu State: your game's welcome screen. After the preload state, all the game images are already loaded into the memory, so they can quickly accessed.

Game State: the actual game where the FUN takes place.

Lets now create files for all our states and a file called main.js which is where we'll add them into the game. main.js will be the following, the other files are empty for now:

```
1 var SpaceHipster = SpaceHipster || {};
2
3 SpaceHipster.game = new Phaser.Game(window.innerWidth,
window.innerHeight, Phaser.AUTO, '');
4
5 SpaceHipster.game.state.add('Boot', SpaceHipster.Boot);
6 //uncomment these as we create them through the tutorial
7 //SpaceHipster.game.state.add('Preload',
SpaceHipster.Preload);
8 //SpaceHipster.game.state.add('MainMenu',
SpaceHipster.MainMenu);
9 //SpaceHipster.game.state.add('Game', SpaceHipster.Game);
10
11 SpaceHipster.game.state.start('Boot');
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

The first thing we do is create a unique namespace so that we avoid conflicts with other libraries we might be using. In this example the namespace will be SpaceHipser. You can use whatever you want for this, as long as it's a unique name which will be unlikely found elsewhere. You can also just not use it and work directly with game and the states names, but I'd recommend using it just for best practice.

```
1 var SpaceHipster = SpaceHipster || {};
```

Means that if the object exists already, we'll use it. Otherwise we'll use a new object.

```
1 SpaceHipster.game = new Phaser.Game(window.innerWidth,
window.innerHeight, Phaser.AUTO, '');
```

We initiate a new game and set the size of the entire windows. Phaser.AUTO means that whether the game will be rendered on a CANVAS element or using WebGL will depend on the browser. If WebGL is available it will be used as the first option (as the performance is better).

We then register the states and finally launch the Boot state.

Include these added files in index.html:

```
1 Learn Game Development at ZENVA.com<script src="js/phaser.js"
type="text/javascript"></script><script src="js/Boot.js"
type="text/javascript"></script><script src="js/Preload.js"
type="text/javascript"></script><script src="js/MainMenu.js"
type="text/javascript"></script><script src="js/Game.js"
type="text/javascript"></script>
```

```
1 <!-- include the main game file --><script
src="js/main.js"></script>
```

States methods

States have some reserved methods which serve specific purposes. These are the ones we'll use in this tutorial. You can find the full list here.

–**init**: is called upon State initialization. If you need to send parameters to the State this is where they'll be accessible (more on that later)

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

–**preload**: this is where the assets are loaded.

–**create**: this is called upon asset loading completion.

–**update**: this is called on every game tick. Basically it's called "many times per second" so this is where you want to include things that need to be constantly tested such as collision detection.

Boot State

Before talking about the Boot state I'd like to say that in theory, you could not use states where all the game takes place, but using states allows you to organize your code better.

The Boot state is a dark place. This is where we define the screen size and other general game configuration options such as the physics engine we'll use (Phaser has three of them, we'll use the simplest one). Also, in the Boot state we load the assets that will be shown in the Preload state.

So.. the Boot state loads the assets for the Preload state, and the Preload state loads the game assets. It's sounding like "a dream within a dream" isn't it? you might think, why instead not load the game assets there? Well nothing stops you from doing that, but the thing is, the game assets will most likely take much longer to load than the preload screen assets. By loading the preloading screen assets (which should be lightweight) we minimize the time where there is a blank screen (something users don't love). Then, on the Preload state we'll have time to load everything else with a nice and charming preload screen.

Content of Boot.js:

```
1 var SpaceHipster = SpaceHipster || {};
2
3 SpaceHipster.Boot = function(){};
4
5 //setting game configuration and loading the assets for the
loading screen
6 SpaceHipster.Boot.prototype = {
7 preload: function() {
8    //assets we'll use in the loading screen
9     this.load.image('logo', 'assets/images/logo.png');
10     this.load.image('preloadbar', 'assets/images/preloader-
bar.png');
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
11  },
12  create: function() {
13    //loading screen will have a white background
14      this.game.stage.backgroundColor = '#fff';
15
16      //scaling options
17  this.scale.scaleMode = Phaser.ScaleManager.SHOW_ALL;
18  this.scale.minWidth = 240;
19  this.scale.minHeight = 170;
20  this.scale.maxWidth = 2880;
21  this.scale.maxHeight = 1920;
22
23  //have the game centered horizontally
24  this.scale.pageAlignHorizontally = true;
25
26  //screen size will be set automatically
27  this.scale.setScreenSize(true);
28
29  //physics system for movement
30  this.game.physics.startSystem(Phaser.Physics.ARCADE);
31
32    this.state.start('Preload');
33  }
34 };
```

Preload State

Our preload screen will just a logo and the loading bar. Preload.js:

```
1 var SpaceHipster = SpaceHipster || {};
2
3 //loading the game assets
4 SpaceHipster.Preload = function(){};
5
6 SpaceHipster.Preload.prototype = {
7  preload: function() {
8    //show logo in loading screen
9    this.splash = this.add.sprite(this.game.world.centerX,
this.game.world.centerY, 'logo');
10     this.splash.anchor.setTo(0.5);
11
12     this.preloadBar = this.add.sprite(this.game.world.centerX,
this.game.world.centerY + 128, 'preloadbar');
13     this.preloadBar.anchor.setTo(0.5);
```

```
14
15    this.load.setPreloadSprite(this.preloadBar);
16
17    //load game assets
18    this.load.image('space', 'assets/images/space.png');
19    this.load.image('rock', 'assets/images/rock.png');
20     this.load.spritesheet('playership',
'assets/images/player.png', 12, 12);
21     this.load.spritesheet('power', 'assets/images/power.png',
12, 12);
22    this.load.image('playerParticle', 'assets/images/player-
particle.png');
23     this.load.audio('collect', 'assets/audio/collect.ogg');
24     this.load.audio('explosion',
'assets/audio/explosion.ogg');
25    },
26    create: function() {
27      this.state.start('MainMenu');
28    }
29 };
```

```
1 this.preloadBar = this.add.sprite(this.game.world.centerX,
this.game.world.centerY + 128, 'preloadbar');
```

Is how we load sprites into the screen. We define their coordinates and the name of the asset, if you go back to Boot.js you'll see that "preloadbar" is how we called the load bar.

```
1 this.load.setPreloadSprite(this.preloadBar);
```

The method setPreloadSprite in Loader entities allows us to grab a sprite (in this case this.preloadBar) and make it into a loading bar. More info in the documentation.

ZENVA

MainMenu State

In this screen we'll start by showing a scrolled stars background and some text.

```
1 SpaceHipster.MainMenu = function(){};
2
3 SpaceHipster.MainMenu.prototype = {
4   create: function() {
5     //show the space tile, repeated
6     this.background = this.game.add.tileSprite(0, 0,
this.game.width, this.game.height, 'space');
7
8     //give it speed in x
9     this.background.autoScroll(-20, 0);
10
11     //start game text
12     var text = "Tap to begin";
13     var style = { font: "30px Arial", fill: "#fff", align:
"center" };
14     var t = this.game.add.text(this.game.width/2,
this.game.height/2, text, style);
15     t.anchor.set(0.5);
16
17     //highest score
18     text = "Highest score: "+this.highestScore;
19    style = { font: "15px Arial", fill: "#fff", align: "center"
};
20
21     var h = this.game.add.text(this.game.width/2,
this.game.height/2 + 50, text, style);
22     h.anchor.set(0.5);
23   },
24   update: function() {
25     if(this.game.input.activePointer.justPressed()) {
26       this.game.state.start('Game');
27     }
28   }
29 };
```

```
1 this.background = this.game.add.tileSprite(0, 0,
this.game.width, this.game.height, 'space');
2
3     //give it speed in x
4     this.background.autoScroll(-20, 0);
```

<u>TileSprites</u> is when you repeat a tile many times to cover a certain area. By using autoscroll and setting a speed in x and y we can make that infinite scrolling effect.
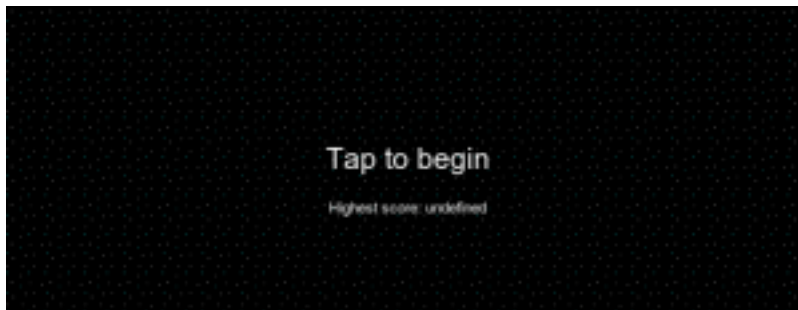
We want to listen for user tap/click to launch the game state and start playing and we do that in the update() method, as it's something that needs to be constantly checked upon.

Later on we'll show the highest score in this screen, for now it'll be undefined.

Game State – Player
Finally we move onto the actual game – open Game.js:

```
1 var SpaceHipster = SpaceHipster || {};
2
3 //title screen
4 SpaceHipster.Game = function(){};
5
6 SpaceHipster.Game.prototype = {
7   create: function() {
8   },
9   update: function() {
10   },
11 };
```

On create we'll start by setting the dimensions of the game world:

```
1 //set world dimensions
2 this.game.world.setBounds(0, 0, 1920, 1920);
```

The background will be the stars sprite again, repeated over and over, for which we can create a TileSprite:

```
1 this.background = this.game.add.tileSprite(0, 0,
this.game.world.width, this.game.world.height, 'space');
```

<u>Zenva Academy</u> – Online courses on Phaser and game programming
<u>Zenva for Schools</u> – Coding courses for high schools

The player will be a sprite, initially located in the center of the world:

```
1 //create player
2 this.player = this.game.add.sprite(this.game.world.centerX,
this.game.world.centerY, 'playership');
```

Make it a bit bigger:

```
1 this.player.scale.setTo(2);
```

The player will be animated the entire time:

```
1 this.player.animations.add('fly', [0, 1, 2, 3], 5, true);
2 this.player.animations.play('fly');
```

If you look at the player.png file you'll see it has 4 frames, and we defined the dimensions in Preload.js. 5 in here means the frequency of the change. For info in the documentation for Animation.

```
1 //player initial score of zero
2 this.playerScore = 0;
```

Initial score.

```
1 //enable player physics
2 this.game.physics.arcade.enable(this.player);
3 this.playerSpeed = 120;
4 this.player.body.collideWorldBounds = true;
```

If we want the player to move, to collide with rocks and to collect alien energy powerups we need to include it in the physics system (that we defined in the Boot state).

In the update method, we'll listen for taps/clicks and we'll set the speed to that location:

```
1 update: function() {
2     if(this.game.input.activePointer.justPressed()) {
3
4         //move on the direction of the input
5         this.game.physics.arcade.moveToPointer(this.player,
this.playerSpeed);
6     }
7 },
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

See how the player leaves the screen after a while. We need the camera to follow the player, so add the following in the create method, after the animation code:

```
1 //the camera will follow the player in the world
2 this.game.camera.follow(this.player);
```

Load sounds

We preloaded two audio files already (they are both ogg files, for full browser support load mp3 versions as well). Let's create the audio objects so we can play them later (using their play() method)

```
1 //sounds
2 this.explosionSound = this.game.add.audio('explosion');
3 this.collectSound = this.game.add.audio('collect');
```

Game State – Asteroids

Floating around space is fun (not that I've done it) but I'm sure it gets boring after a while. Let's add some huge floating rocks that will crash the ship upon collision.

Bellow the camera code in create(), add the following:

```
1 this.generateAsteriods();
```

And lets create that method (see how we are not creating our own methods int he State, it's fine as long as it doesn't use the *reserved method names*):

```
1 generateAsteriods: function() {
2     this.asteroids = this.game.add.group();
3
4     //enable physics in them
5     this.asteroids.enableBody = true;
6     this.asteroids.physicsBodyType = Phaser.Physics.ARCADE;
7
8     //phaser's random number generator
9     var numAsteroids = this.game.rnd.integerInRange(150, 200)
10    var asteriod;
11
12    for (var i = 0; i &lt; numAsteroids; i++) {
13        //add sprite
14        asteriod =
this.asteroids.create(this.game.world.randomX,
this.game.world.randomY, 'rock');
15        asteriod.scale.setTo(this.game.rnd.integerInRange(10,
40)/10);
16
17        //physics properties
```

```
18        asteriod.body.velocity.x =
this.game.rnd.integerInRange(-20, 20);
19        asteriod.body.velocity.y =
this.game.rnd.integerInRange(-20, 20);
20        asteriod.body.immovable = true;
21        asteriod.body.collideWorldBounds = true;
22      }
23  },
```

```
1 this.asteroids = this.game.add.group();
```

Phaser allows us to create groups of elements. This makes sense when you want to restrict say collision detection to a certain group. It also allows you to set some group-level properties.

```
1 var numAsteroids = this.game.rnd.integerInRange(150, 200);
```

How many asteroids will we have? lets make it random within a range using Phaser's method for random integer intervals:

Then we created the actual rocks. Set their size to be random-ish, and enable their physics properties:

By using immovable = true we made it so that their trajectories doesn't get affected when crashing with the player (we haven't set collision yet thou. After we do, try setting this property
to false and see the difference).

We don't want them to leave the game world:

```
1 asteriod.body.collideWorldBounds = true;
```

In update(), let's add collision detection between the player and the asteroids group:

```
1 //collision between player and asteroids
2 this.game.physics.arcade.collide(this.player, this.asteroids,
this.hitAsteroid, null, this);
```

Define the hitAsteroid() method where we play an explosion sound, destroy the ship and go to game over:

```
1 hitAsteroid: function(player, asteroid) {
2     //play explosion sound
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
3    this.explosionSound.play();
4
5    //player explosion will be added here
6
7    this.player.kill();
8
9    this.game.time.events.add(800, this.gameOver, this);
10   },
```

Particles and Game Over

When you hit a rock we can make the ship explode using particles. Particles are a game development technique that allows you to work with many individual elements or "particles". This can be used to simulate explosions, emissions and much more. Check out the examples for ideas.

Lets make the ship explode when hit by a rock. Remember we loaded an image asset called playerParticle, which was just a blue square. We'll use that for the ship explosion.

```
1 hitAsteroid: function(player, asteroid) {
2     //play explosion sound
3     this.explosionSound.play();
4
5     //make the player explode
6     var emitter = this.game.add.emitter(this.player.x,
this.player.y, 100);
7     emitter.makeParticles('playerParticle');
8     emitter.minParticleSpeed.setTo(-200, -200);
9     emitter.maxParticleSpeed.setTo(200, 200);
10     emitter.gravity = 0;
11     emitter.start(true, 1000, null, 100);
12     this.player.kill();
13
14     //call the gameOver method in 800 milliseconds, we haven't
created this method yet
15     this.game.time.events.add(800, this.gameOver, this);
16   },
```

When starting the emitter, the first "true" parameter is because this will be a single particle emission (a single explosion), which will last 1 second (1000 milliseconds), then we put null because that's also for repeating emissions (it defines how many per emission), lastly we'll send 100 particles on this single explosion.

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Kill the player sprite as well and call a gameOver method after 0.8 seconds (800 milliseconds), which we need to create:

Game State – Collectables

So far there is not much to do in the game. Let's create collectables and add score. Collectables will be some weird alien jewel that your ship wants to find in this hostile place.

We'll follow a similar approach than we did with the rocks. In create() add this before generateAsteroids (we want the collectables to go under the asteroids, if we add them afterwards they'll show on top of them):

```
1 this.generateCollectables();
```

Add the new method to create and animate these alien jewels/powerups:

```
1 generateCollectables: function() {
2    this.collectables = this.game.add.group();
3
4    //enable physics in them
5    this.collectables.enableBody = true;
6    this.collectables.physicsBodyType = Phaser.Physics.ARCADE;
7
8    //phaser's random number generator
9    var numCollectables = this.game.rnd.integerInRange(100,
150)
10     var collectable;
11
12    for (var i = 0; i &lt; numCollectables; i++) {
13       //add sprite
14       collectable =
this.collectables.create(this.game.world.randomX,
this.game.world.randomY, 'power');
15       collectable.animations.add('fly', [0, 1, 2, 3], 5,
true);
16       collectable.animations.play('fly');
17    }
18  },
```

On update() let's define not collision (which makes the ship stop/"hit" physically the object) but **overlap**, which won't affect the ship's speed:

```
1 //overlapping between player and collectables
2 this.game.physics.arcade.overlap(this.player,
this.collectables, this.collect, null, this);
```

Adding the method to collect and update score:

```
1 collect: function(player, collectable) {
2      //play collect sound
3      this.collectSound.play();
4
5      //update score
6      this.playerScore++;
7      //will add later: this.scoreLabel.text = this.playerScore;
8
9      //remove sprite
10      collectable.kill();
11  },
```

Now you should be able to wonder around and collect alien mana. The only thing we are missing is the score on the screen.

Score and High Score

We'll show the score using a similar approach to what we did in the MainMenu state. Add this to the end of create():

```
1 //show score
2 this.showLabels();
```

Let's add that new method where we'll take care of displaying the score (and perhaps other stats if you want to expand the game):

```
1 showLabels: function() {
2      //score text
3      var text = "0";
4      var style = { font: "20px Arial", fill: "#fff", align:
"center" };
5      this.scoreLabel = this.game.add.text(this.game.width-50,
this.game.height - 50, text, style);
6      this.scoreLabel.fixedToCamera = true;
7 }
```

```
1 this.scoreLabel.fixedToCamera = true;
```

This is so that the number stays on the same position of the screen regardless the camera movements.

When collecting (on collect()), update the text content of the label:

```
1 this.scoreLabel.text = this.playerScore;
```

Now, we also want to show the high score in the Menu screen, but hey that's a different State, how can we pass parameters to another state?

The solution is easy: adding an init() method to the state, where you can add all the parameters you want. Let's add such a method in MainMenu. We'll receive the score of the game that was just played (it will use zero if none is passed). Then it will check if it's the highest and will show it:

```
1 init: function(score) {
2    var score = score || 0;
3    this.highestScore = this.highestScore || 0;
4
5    this.highestScore = Math.max(score, this.highestScore);
6  },
```

**Note: using the localStorage API you could save this high score, if you want to learn how to use localStorage you can check this tutorial created by Ashley Menhennet, a course creator and trainer at Zenva.

Add a gameOver method in the Game state and modify how we send the player back to the MainMenu state:

```
1 gameOver: function() {
2    //pass it the score as a parameter
3    this.game.state.start('MainMenu', true, false,
this.playerScore);
4  },
```

The first "true" is because we want to refresh the game world, the second is set to "false" as we don't want to erase the game's cache (or we would have to reload all the assets, try
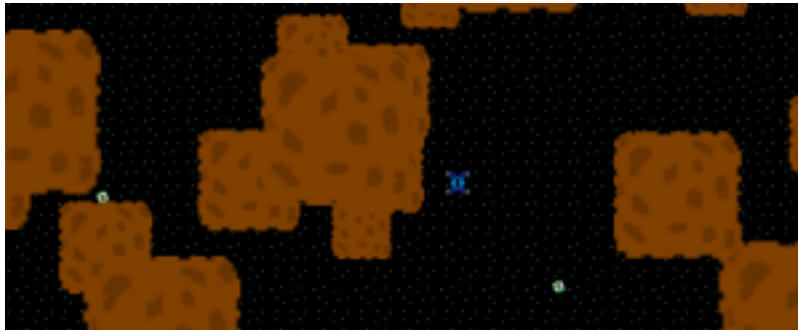
with true to see what happens!) then we pass the parameter.
Game finished!

You can now play an entire (very simple) game! Feel free to use this as a basis for your own games as long as you give us credit (link back to us).

If you haven't already, you can download this tutorial game files and assets here.

HTML5 Phaser Tutorial – Top-Down Games with Tiled
By Pablo Farias Navarro

I'm a huge fan of top-down 2D games, mainly RPG's, adventure games and dungeon crawlers.

In this tutorial we'll create a basic template you can use to make your own top-down games using Phaser, an awesome HTML5 game framework.

This template will be expanded in further tutorials to add the ability to move between levels, work on mobile and more cool stuff.

We'll be using a free map editor called Tiled to create our level. One nice thing about Phaser is that you can easily load map files exported from Tiled (JSON format).

This tutorial builds on top of the concepts covered in our previous Phaser tutorial, so if you are not familiar with States, preloading assets and rendering sprites you might want to check that tut first.

Source code files

You can grab them from here (updated to version 2.4.2). If you want us to notify you when we release new tutorials on Phaser or HTML5 game development don't forget to click "subscribe to our list" when downloading the files (if you choose not to you won't hear from us!).

Tutorial goals

1-Learn to create levels for your game using Tiled and load them using Phaser.
2-Learn how to create a basic top-down game starter.

Concepts covered

-Setting the game size in pixels
-Tiled basic usage
-Tile layers and object layers
-Tilemaps
-Creating sprites from object layers
-Moving a character in a top-down level

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Tutorial requirements
-This tutorial builds on the concepts covered in our previous Phaser tutorial HTML5 Phaser Tutorial – SpaceHipster, a Space Exploration Game, so it assumes you know how to setup a game, work with States, preload game assets and create sprites.

-Basic to intermediate knowledge of JavaScript. If you need a refreshment feel free to check our JavaScript beginners video course at Zenva Academy.
-A code editor or IDE. My personal preference at the moment is Sublime Text.

-Get Tiled, an Open Source and free level editor.

-Grab Phaser from it's Github repo. You can either clone the repo or download the ZIP file.

-You need to run the code and Phaser examples using a web server. Some popular options are Apache (WAMP if in Windows, MAMP if in Mac). Some lightweight alternatives are Mongoose web server and Python's HTTP server. Take a look at this guide for more details.

-Download the full tutorial source code and game assets here. If you want us to let you know when we release new tutorials on Phaser or HTML5 game development don't forget to subscribe to our list!

-Have the documentation and the examples page at hand. Also don't forget that you can always find answers by looking at the source code of Phaser itself!

Get Tiled-Face

Tiled is a really awesome level editor created by Thorbjørn Lindeijer which allows you to visually create tile-based levels for your games. The main file format these levels use is TMX. In order to use these levels in Phaser we need to export them as JSON.

When I talk about "tiles" and "tiled-based games" what I mean are games where the levels are composed of individual small blocks or pieces. Check out the Wikipedia definition if you are not 100% familiar with this concept.

To create a new map in Tiled go to File -> New:

On width and height specify the number of tiles your level will have on each dimension. Tile size refers to the size of the individual tiles. Make sure Orientation is set to "Orthogonal" and Layer format to "CSV".

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools
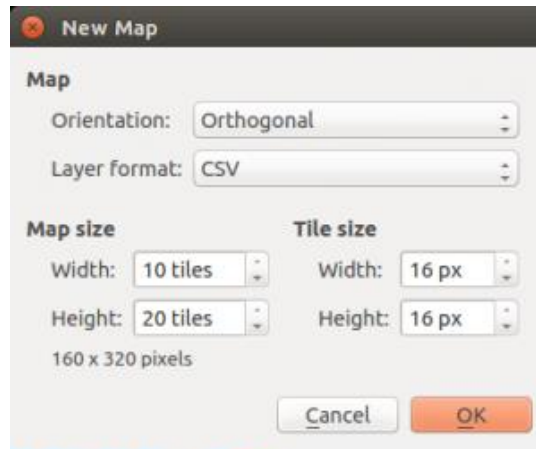
The next step is to create the Layers. A level will have different layers that sit on top of each other. The naming of the layers is important here as we need to refer to that in the code later on.

There are two types of layers:

-Tile layer: layer made of tiles/blocks.
-Objects layer: layer where you create vector objects that can contain metadata.

You can create layers using the "+" button under layers, to rename them just click on the name of the newly created layers. The order matters, you can rearrange them using the
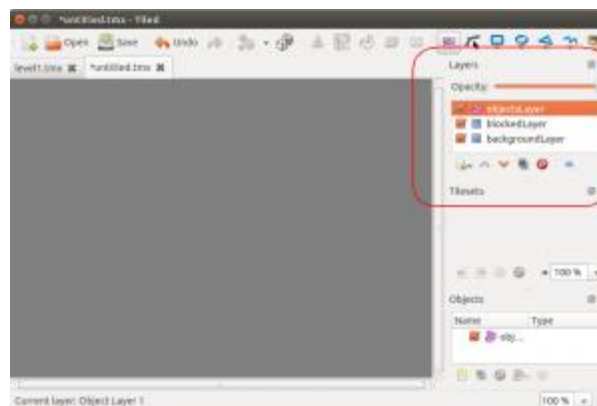
arrows.

In our example we'll have two tile layers: one for the background called backgroundLayer (doesn't collide with the player) and one for the blocking elements called blockedLayer (walls, trees, etc). We'll also have one objects layer (objectsLayer) that will represent game elements, in this case the player starting location in the level, the

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools
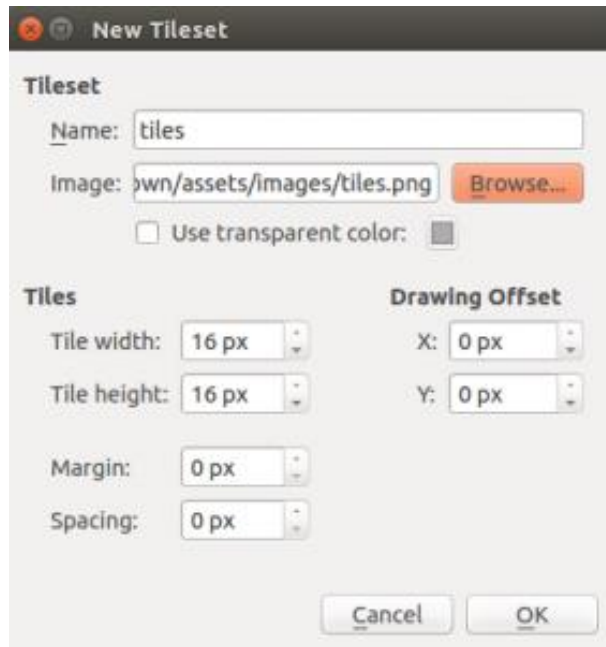
collectables/items, a door (which we'll extend in a future tutorial). In this layer you'd want to add enemies and more stuff. This is all just a suggestion and how I normally do it, it is by no means a convention or a rule of the framework.

Loading the tileset

A tileset is a set of individual tiles, which come in an image. You can have many tilesets on your tilemaps. We'll use just one: assets/images/tiles.png.
To load a tileset: Map -> New Tileset

The nice retro tileset we'll use is from Open Game Art and it's a public domain image (CC0), so you can use it for both non-commercial and commercial projects.
The tiles are 16×16 and there is no separation between them or margin. The "name" in the text field, keep that name in your head as we need to refer to it when joining the tileset with the map in Phaser.
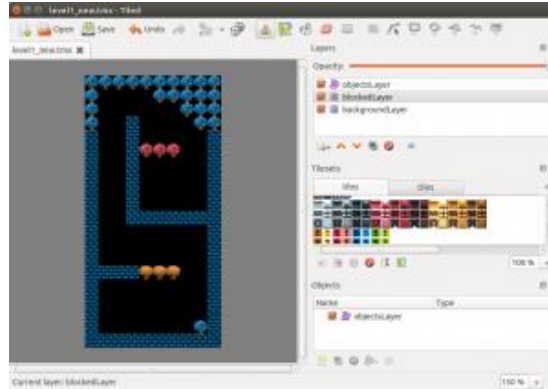
Create the level

You can now create your level. Now it's really up to you. What I did was fill the backgroundLayer with a black tile (using the bucket tool), then changed to the blockedTile and painted some walls and trees:

The object layer

We'll add 3 different types of objects or game elements: items that will be collected by the player, the player starting position, and a door that will take you to a new level (to be implemented in a follow-up tutorial).

These objects will be represented by sprites from the tileset, but that is just for OUR simplicity. Phaser will show the sprite you tell it to when creating a Sprite object.

To insert new objects: select the objectsLayer layer, click on the Insert Tile button (the one that shows a picture of the sunset) then click on the tile you want to show, then click
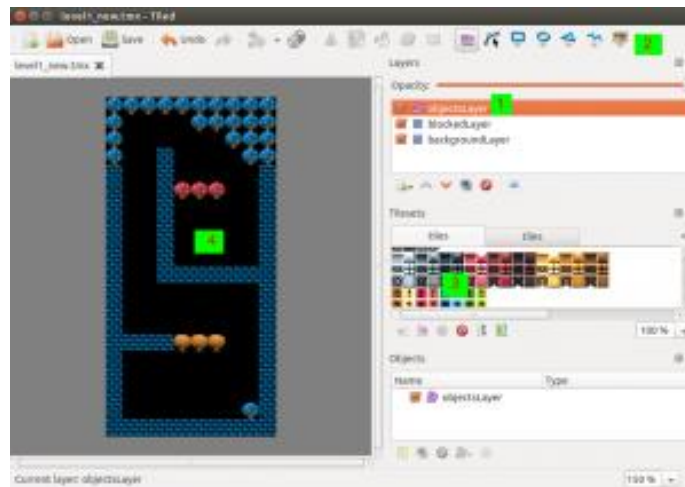
on the location of the object within the map.

We'll add some blue and green teacups, a door and a "player":

Adding properties to the objects

How is Phaser going to know what these elements represent and what sprite should be used to display them? Lets add that as object properties.

Click on the "Select object" button from the menu, then double click on any object and you'll see a dialog where you can enter properties.

Enter a property called "type" with value "item" to the tea cups (this is not the "type" select dropdown!, this is within the Properties area), "playerStart" for the player location and "door" to the door.

For the blue cups enter a property "sprite" with value "bluecup", "greencup" for the green cup, "browndoor" to the door. None to the player start as it won't represent a sprite, but a location for us to use. These values are the sprite keys we'll implement in Phaser for the corresponding image assets (peak at /assets/images).

**You can also just create one of each element, then select the element with the right click and click "duplicate", this will allow you to quickly replicate an object and then drag it to it's final position.

We won't implement the door behavior in this tutorial, but enter these properties so you can guess where I'm going with this:

Before we move any further, I'd like to make it clear that this will work with the game template we are building in this tutorial, it is not a default Phaser functionality. Phaser only provides the ability to create several sprites from objects using the same sprite key. Take a look at the Tilemap documentation to learn what Phaser provides in this aspect.

Once you are doing export the level as a JSON file. File -> Export As -> Json Files, save into /assets/tilemaps.

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools
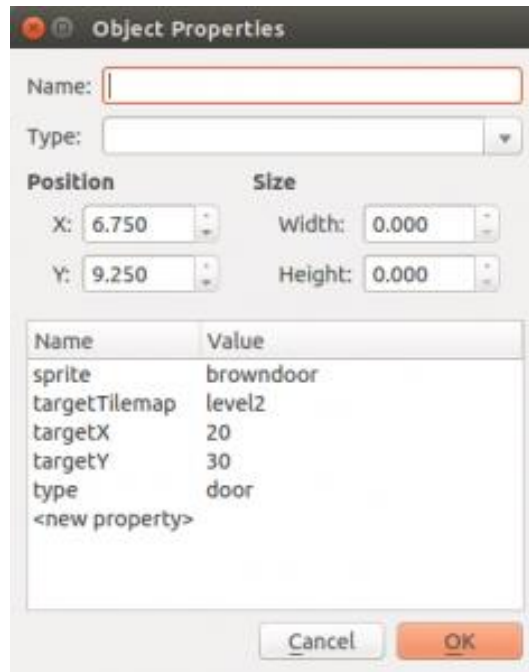
Hola Mundo Phaser

I'll go through this section without stepping into much details as these concepts were covered in the previous Phaser tutorial.

In this game we'll have three states: Boot, Preload and Game. You can always add a main menu screen as we did in the previous tutorial.

Let's start by creating index.html, where we'll include the game files:

```html
1 <!DOCTYPE html>
2 <html>
3
4 <head>
5 <meta charset="utf-8" />
6 <title>Learn Game Development at ZENVA.com</title>
7
8 <script type="text/javascript" src="js/phaser.js"></script>
9 <script type="text/javascript" src="js/Boot.js"></script>
10 <script type="text/javascript" src="js/Preload.js"></script>
11 <script type="text/javascript" src="js/Game.js"></script>
12
13 <style>
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
14    body {
15        padding: 0px;
16        margin: 0px;
17    }
18    </style>
19 </head>
20
21 <body>
22 <!-- include the main game file -->
23 <script src="js/main.js"></script>
24 </body>
25 </html>
```

I prefer using the non-minified version of Phaser when doing development (phaser.js) as it makes it easier to debug errors. When working with HTML5 game development libraries it's good practice to constantly refer to the source code itself to find answers. Not everything can be found on StackOverflow and getting familiar with the framework's code itself can save you hours.

main.js creates the game itself and defines the game size in pixels, which I've set to a very small size for this example. The reason being, the tileset I'll be using is also small and I wanted to have the game not showing the entire level (which is not that big) in the browser space.

By creating all the game objects inside a parent object called TopDownGame (this is what's called the namespace pattern) we don't pollute the global scope with our game elements. This is good practice as it could happen that you are including some other libraries in your project and their names match those of the objects you are creating so then you have a conflict!. Whereas if you put everything inside TopDownGame there won't be any conflicts, unless another library created an object called TopDownGame in the global scope which is unlikely.

(Btw this:

```
1 var TopDownGame = TopDownGame || {};
```

basically means that if TopDownGame exists, then use it, if it doesn't exist then initiate it as an empty object)

```
1 var TopDownGame = TopDownGame || {};
2
3 TopDownGame.game = new Phaser.Game(160, 160, Phaser.AUTO, '');
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
4
5 TopDownGame.game.state.add('Boot', TopDownGame.Boot);
6 TopDownGame.game.state.add('Preload', TopDownGame.Preload);
7 TopDownGame.game.state.add('Game', TopDownGame.Game);
8
9 TopDownGame.game.state.start('Boot');
```

Boot.js

In Boot.js we'll create the Boot state, where we load the images for the loading screen (just a loading bar in this example) and define other game-level configurations.

Since we are using Phaser.ScaleManager.SHOW_ALL the game will show as large as it can to fit the browser space, but it wont' show more than 160×160 pixels of the game world, as that's the limit we defined in main.js.

```
1 var TopDownGame = TopDownGame || {};
2
3 TopDownGame.Boot = function(){};
4
5 //setting game configuration and loading the assets for the
loading screen
6 TopDownGame.Boot.prototype = {
7   preload: function() {
8     //assets we'll use in the loading screen
9     this.load.image('preloadbar', 'assets/images/preloader-
bar.png');
10  },
11  create: function() {
12    //loading screen will have a white background
13    this.game.stage.backgroundColor = '#fff';
14
15    //scaling options
16    this.scale.scaleMode = Phaser.ScaleManager.SHOW_ALL;
17
18    //have the game centered horizontally
19    this.scale.pageAlignHorizontally = true;
20    this.scale.pageAlignVertically = true;
21
22    //physics system
23    this.game.physics.startSystem(Phaser.Physics.ARCADE);
24
25    this.state.start('Preload');
26  }
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
27 };
```

Preload.js

Preload.js will contain the loading of the assets, including the loading of the tilemap. See how the tileset itself is simply loaded as another sprite. It will be later on when we relate the tilemap level with the tileset:

```
1 var TopDownGame = TopDownGame || {};
2
3 //loading the game assets
4 TopDownGame.Preload = function(){};
5
6 TopDownGame.Preload.prototype = {
7   preload: function() {
8     //show loading screen
9     this.preloadBar = this.add.sprite(this.game.world.centerX,
this.game.world.centerY + 128, 'preloadbar');
10     this.preloadBar.anchor.setTo(0.5);
11
12     this.load.setPreloadSprite(this.preloadBar);
13
14     //load game assets
15     this.load.tilemap('level1', 'assets/tilemaps/level1.json',
null, Phaser.Tilemap.TILED_JSON);
16     this.load.image('gameTiles', 'assets/images/tiles.png');
17     this.load.image('greencup', 'assets/images/greencup.png');
18     this.load.image('bluecup', 'assets/images/bluecup.png');
19     this.load.image('player', 'assets/images/player.png');
20     this.load.image('browndoor',
'assets/images/browndoor.png');
21
22   },
23   create: function() {
24     this.state.start('Game');
25   }
26 };
```

Game.js

The following code in the create() method will show the map (without the objects) on your browser:

```
1 var TopDownGame = TopDownGame || {};
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
2
3 //title screen
4 TopDownGame.Game = function(){};
5
6 TopDownGame.Game.prototype = {
7   create: function() {
8     this.map = this.game.add.tilemap('level1');
9
10     //the first parameter is the tileset name as specified in
Tiled, the second is the key to the asset
11    this.map.addTilesetImage('tiles', 'gameTiles');
12
13    //create layer
14    this.backgroundlayer =
this.map.createLayer('backgroundLayer');
15    this.blockedLayer = this.map.createLayer('blockedLayer');
16
17    //collision on blockedLayer
18    this.map.setCollisionBetween(1, 100000, true,
'blockedLayer');
19
20    //resizes the game world to match the layer dimensions
21    this.backgroundlayer.resizeWorld();
22  }
23 }
```

The first thing we did was create a Tilemap object from the loaded JSON file. What's a map without the images? we need to add the tileset image to this map, which we do with this.map.addTilesetImage('tiles', 'gameTiles');, where "tiles" is the name we gave the tileset in Tiled, and "gameTiles" is the image key as defined in Preload.js (I made those two different in purpose to make it more clear what they are referring to).

Then we load the layers, and make the blockedLayer "collision enabled". The number between 1 and 2000 is an index range for the tiles for which we want to enable collision (in this case such a big number should include them all which is what I intended). In order to obtain this number open level1.json and in "layers" – "data" of the two layers find the largest number you can find, in this case 1896 so I put 2000 just to be sure I didn't miss a slightly larger number. So in theory, you could just use one single layer, and define certain tiles as collision-enabled by specifying their index range.

WARNING: A previous version of this tutorial said put 100000 instead of 2000. The problem with putting unnecessary large numbers is that you are running more loops affecting the performance of your game

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Page 34

I find it easier to work with one layer for the background and one for the collision elements as I don't want to count tiles one by one, that is just my preference here.

Lastly, we make the world the same size as the map.

Finding objects by type

Remember that "type" properties we gave to the objects in the level? Well I'd like to be able to find the objects of a certain type, so we'll build a utility method in Game.js for that.

Tilemap objects have a "objects" property which is an array with the objects of each layer (you could have many object layers!). We'll use this method to find the objects that have the property "type" of the value we want. I recommend you take a look at the file /src/Tilemap.js from the downloaded Phaser library and to the Tilemap documentation. You can also just use this method but it's always better to understand where it came from.

```
1  //find objects in a Tiled layer that containt a property
called "type" equal to a certain value
2  findObjectsByType: function(type, map, layer) {
3    var result = new Array();
4    map.objects[layer].forEach(function(element){
5      if(element.properties.type === type) {
6        //Phaser uses top left, Tiled bottom left so we have to
adjust the y position
7        //also keep in mind that the cup images are a bit
smaller than the tile which is 16x16
8        //so they might not be placed in the exact pixel
position as in Tiled
9        element.y -= map.tileHeight;
10        result.push(element);
11      }
12    });
13    return result;
14  },
```

When calling this method, it will give us an array of objects as they are represented in the assets/tilemaps/level1.json file.

Create sprites from Tiled objects

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

We are able to find tiled objects by type, and now we'd like to create sprites for them. This method in Game.js will help us out. This will become more clear over the next section where we create the tea cups.

```
1 //create a sprite from an object
2   createFromTiledObject: function(element, group) {
3     var sprite = group.create(element.x, element.y,
element.properties.sprite);
4
5       //copy all properties to the sprite
6       Object.keys(element.properties).forEach(function(key){
7         sprite[key] = element.properties[key];
8       });
9   },
```

We are copying all the properties you might have entered in Tiled. For example for the door we entered a bunch of properties. Well, this will all be copied to the sprite. Even if you entered sprite properties such as "alpha" (for transparency), this will be moved along to the sprite you are creating here.

** We are not using Tilemap.createFromObjects here because I wanted to be able to specify the type and the sprite inside of Tiled. If createFromObjects is what you need for your game I recommend you use that!

I love tea

```
1 var TopDownGame = TopDownGame || {};
2
3 //title screen
4 TopDownGame.Game = function(){};
5
6 TopDownGame.Game.prototype = {
7   create: function() {
8     this.map = this.game.add.tilemap('level1');
9
10     //the first parameter is the tileset name as specified in
Tiled, the second is the key to the asset
11     this.map.addTilesetImage('tiles', 'gameTiles');
12
13     //create layer
14     this.backgroundlayer =
this.map.createLayer('backgroundLayer');
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
15    this.blockedLayer = this.map.createLayer('blockedLayer');
16
17    //collision on blockedLayer
18    this.map.setCollisionBetween(1, 100000, true,
'blockedLayer');
19
20    //resizes the game world to match the layer dimensions
21    this.backgroundlayer.resizeWorld();
22
23    this.createItems();
24
25  },
26   createItems: function() {
27    //create items
28    this.items = this.game.add.group();
29    this.items.enableBody = true;
30    var item;
31    result = this.findObjectsByType('item', this.map,
'objectsLayer');
32    result.forEach(function(element){
33      this.createFromTiledObject(element, this.items);
34    }, this);
35  },
36  //find objects in a Tiled layer that containt a property
called "type" equal to a certain value
37  findObjectsByType: function(type, map, layer) {
38    var result = new Array();
39    map.objects[layer].forEach(function(element){
40      if(element.properties.type === type) {
41        //Phaser uses top left, Tiled bottom left so we have
to adjust
42        //also keep in mind that the cup images are a bit
smaller than the tile which is 16x16
43        //so they might not be placed in the exact position as
in Tiled
44        element.y -= map.tileHeight;
45        result.push(element);
46      }
47    });
48    return result;
49  },
50  //create a sprite from an object
51  createFromTiledObject: function(element, group) {
52    var sprite = group.create(element.x, element.y,
element.properties.sprite);
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
53
54        //copy all properties to the sprite
55        Object.keys(element.properties).forEach(function(key){
56            sprite[key] = element.properties[key];
57        });
58    }
59 };
```

Doors

Adding the door will follow the same approach:

```
1 this.createItems();
2 this.createDoors();
```

and:

```
1 createDoors: function() {
2     //create doors
3     this.doors = this.game.add.group();
4     this.doors.enableBody = true;
5     result = this.findObjectsByType('door', this.map,
'objectsLayer');
6
7     result.forEach(function(element){
8         this.createFromTiledObject(element, this.doors);
9     }, this);
10   },
```

But we can't move around yet and find it.

The player

We are representing the player start position with an object of type "playerStart". What we'll do is find that object, then create the player sprite on that location and set the camera to focus on the player, and activate cursor keys to move the player with the arrows (to be implemented shortly):

```
1 //create player
2 var result = this.findObjectsByType('playerStart', this.map,
'objectsLayer')
3
4 //we know there is just one result
```

```
5 this.player = this.game.add.sprite(result[0].x, result[0].y,
'player');
6 this.game.physics.arcade.enable(this.player);
7
8 //the camera will follow the player in the world
9 this.game.camera.follow(this.player);
10
11 //move player with cursor keys
12 this.cursors = this.game.input.keyboard.createCursorKeys();
```

Player movement

Listen to the cursor keys and adjust the player velocity accordingly. We'll do this in the update() method:

```
1 update: function() {
2     //player movement
3     this.player.body.velocity.y = 0;
4     this.player.body.velocity.x = 0;
5
6     if(this.cursors.up.isDown) {
7         this.player.body.velocity.y -= 50;
8     }
9     else if(this.cursors.down.isDown) {
10        this.player.body.velocity.y += 50;
11    }
12    if(this.cursors.left.isDown) {
13        this.player.body.velocity.x -= 50;
14    }
15    else if(this.cursors.right.isDown) {
16        this.player.body.velocity.x += 50;
17    }
18 },
```

Now you can move your player around! but there is no collision yet.

Collision

Add the following to update(). This works in the same way as we covered in the previous tutorial, but see how you can also just put in a layer, as we are doing with this.blockedLayer:

```
1 //collision
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
2 this.game.physics.arcade.collide(this.player,
this.blockedLayer);
3 this.game.physics.arcade.overlap(this.player, this.items,
this.collect, null, this);
4 this.game.physics.arcade.overlap(this.player, this.doors,
this.enterDoor, null, this);
```

Implementing collect():

```
1 collect: function(player, collectable) {
2     console.log('yummy!');
3
4     //remove sprite
5     collectable.destroy();
6   },
```

And enterDoor(), which is something left for a follow-up tutorial:

```
1 enterDoor: function(player, door) {
2     console.log('entering door that will take you to
'+door.targetTilemap+' on x:'+door.targetX+' and
y:'+door.targetY);
3   },
```

And that's it!

We can now create a level in Tiled, load it in Phaser and move around a character around with the cursor keys!

Feel free to use this code for your own games, and it's always nice if you link back to us □

If you haven't already, you can download this tutorial game files and assets here.

Platformer Tutorial with Phaser and Tiled - Part 1
By Renan Oliveira

Tiled is a free map editor that can be easily integrated with Phaser. In this tutorial, we will build a simple platformer game using Phaser and Tiled. While the map will be created using Tiled, we will use Phaser to read this map and load all the objects into the level. In this tutorial I'll assume you're familiar with the following concepts.

- Javascript and object-oriented programming.
- Basic Phaser concepts, such as: states, sprites, groups and arcade physics.

The following concepts will be covered in this tutorial:

- Using Tiled to create a platformer level.
- Writing a Phaser State that reads the Tiled map and instantiate all the game objects.
- Writing the game logic of a simple platformer.

Tutorial source code
You can download the tutorial source code here.

Creating the map
First, we have to create a new Tiled map. In this tutorial we will use an orthogonal view map with 20×10 tiles. Each tile being 35×35 pixels. For those not familiar with it, an orthogonal view means that the player views the game by a 90 degree angle (you can read more about the difference between orthogonal and isometric view here:

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

http://gamedev.stackexchange.com/questions/22277/difference-between-orthogonal-map-

and-isometric-map).
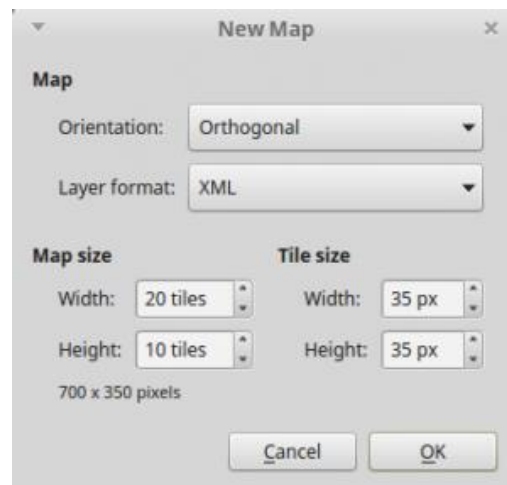
After creating the map, we have to add our tileset, which can be done clicking on "map -
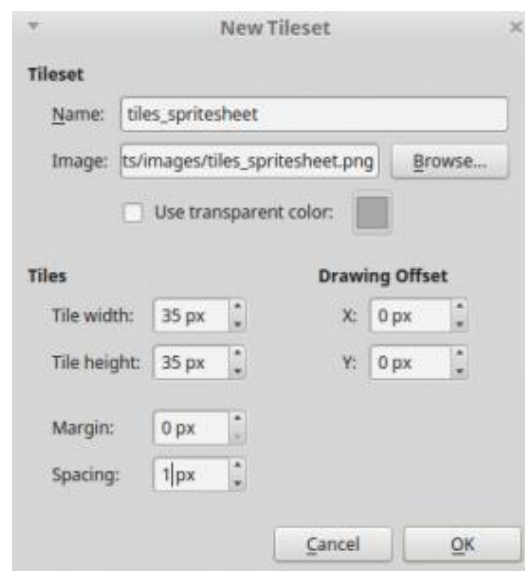
> new tileset". The tileset we're going to use has 35×35 pixels, and a 1 pixel spacing. In tiled, there are two types of layers: tile layers or object layers. Tile layers are composed by the sprites of the loaded tileset and are aligned in the map grid. In this tutorial, we will have two tile layers: background and collision. Object layers are not aligned to a grid, and represent the objects of the game. In this tutorial, we will have only one object layer, containing all our game objects.

<u>Zenva Academy</u> – Online courses on Phaser and game programming
<u>Zenva for Schools</u> – Coding courses for high schools

To create tile layers, just use the stamp brush tool to paint the layer with the desired tiles. For the object layer, select a tile image to use and put it in the desired position. Notice that the images used in the object layer don't represent the actual image that will be used by Phaser during the game. We will have to load the correct asset later in our code. Here
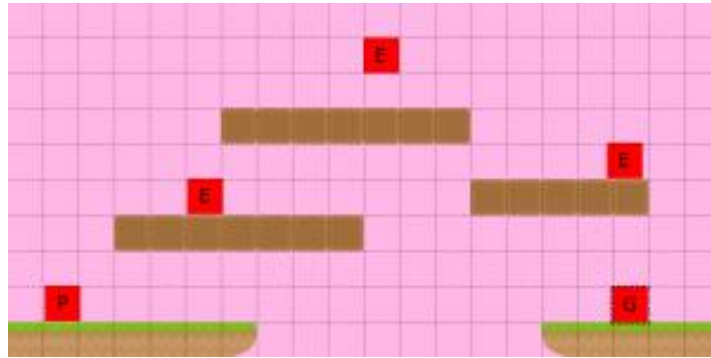
is an example of a created map. Feel free to create your own!

Now, we're going to set some properties that will allow our game to load the map. First, in the collision layer we will add a property telling our game that this layer may collide with other objects, as shown below:
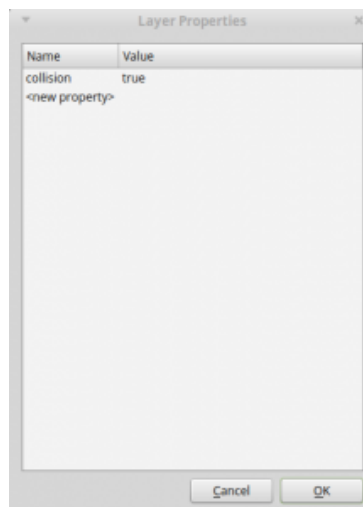
Finally, for each object, we set its properties. The name, type, group, and texture properties are required for our game, since we will use them to properly instantiate the game objects. Any other properties should be defined according to our game logic. For now,

we will set only the required properties, after we code the game logic, we can go back to
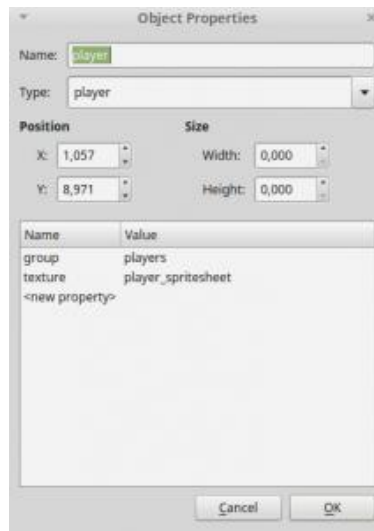
add each object properties.

With the map finished, export it to a json file, so our game can load it.

Json level file

There is some information our game needs to know before loading the map, like the game assets, groups and map information. We will keep all this information in a json file which will be read at the beginning of the game. Below is the json level file we will use in this game. Notice we have to define the assets we will need, the groups of sprites and the map information.

File levels.json in assets/levels

```
1  {
2      "assets": {
3          "map_tiles": { "type": "image", "source":
"assets/images/tiles_spritesheet.png" },
4          "player_spritesheet": { "type": "spritesheet",
"source": "assets/images/player_spritesheet.png", "frame_width":
28, "frame_height": 30, "frames": 5, "margin": 1, "spacing": 1
},
5          "slime_image": { "type": "image", "source":
"assets/images/slime.png" },
```

```json
6          "fly_spritesheet": { "type": "spritesheet", "source":
"assets/images/fly_spritesheet.png", "frame_width": 37,
"frame_height": 20, "frames": 2 },
7          "goal_image": { "type": "image", "source":
"assets/images/goal.png" },
8          "level_tilemap": { "type": "tilemap", "source":
"assets/maps/level1_map.json" }
9      },
10     "groups": [
11         "enemies",
12         "goals",
13         "players"
14     ],
15     "map": {
16         "key": "level_tilemap",
17         "tileset": "map_tiles"
18     }
19 }
```

Game states

We will use the following states to run our game:

- **Boot State**: loads a json file with the level information and starts the Loading State.
- **Loading Sate**: loads all the game assets, and starts the Level State.
- **Tiled State**: creates the map and all game objects.

The code for Boot State is shown below. The Boot State loads the json file described above, so the assets can be loaded in the Loading State.

File BootState.js in js/states:

```
1 var Platformer = Platformer || {};
2
3 Platformer.BootState = function () {
4     "use strict";
5     Phaser.State.call(this);
6 };
7
8 Platformer.prototype = Object.create(Phaser.State.prototype);
9 Platformer.prototype.constructor = Platformer.BootState;
10
11 Platformer.BootState.prototype.init = function (level_file) {
12     "use strict";
13     this.level_file = level_file;
14 };
15
16 Platformer.BootState.prototype.preload = function () {
17     "use strict";
18     this.load.text("level1", this.level_file);
19 };
20
21 Platformer.BootState.prototype.create = function () {
22     "use strict";
23     var level_text, level_data;
24     level_text = this.game.cache.getText("level1");
25     level_data = JSON.parse(level_text);
26     this.game.state.start("LoadingState", true, false,
level_data);
27 };
```

As shown below, the Loading State loads all the game assets in the preload method, and when it is finished, it starts the Tiled State in the create method. Notice that, since we specify the asset type in the json file, it is straightforward to load them, and we can load different kinds of assets.

```
1 var Platformer = Platformer || {};
2
3 Platformer.LoadingState = function () {
4     "use strict";
5     Phaser.State.call(this);
6 };
7
8 Platformer.prototype = Object.create(Phaser.State.prototype);
9 Platformer.prototype.constructor = Platformer.LoadingState;
10
11 Platformer.LoadingState.prototype.init = function
(level_data) {
12     "use strict";
13     this.level_data = level_data;
14 };
15
16 Platformer.LoadingState.prototype.preload = function () {
17     "use strict";
18     var assets, asset_loader, asset_key, asset;
19     assets = this.level_data.assets;
20     for (asset_key in assets) { // load assets according to
asset key
21         if (assets.hasOwnProperty(asset_key)) {
22             asset = assets[asset_key];
23             switch (asset.type) {
24             case "image":
25                 this.load.image(asset_key, asset.source);
26                 break;
27             case "spritesheet":
28                 this.load.spritesheet(asset_key, asset.source,
asset.frame_width, asset.frame_height, asset.frames,
asset.margin, asset.spacing);
29                 break;
30             case "tilemap":
31                 this.load.tilemap(asset_key, asset.source,
null, Phaser.Tilemap.TILED_JSON);
32                 break;
33             }
34         }
```

```
35    }
36 };
37
38 Platformer.LoadingState.prototype.create = function () {
39    "use strict";
40    this.game.state.start("GameState", true, false,
this.level_data);
41};
```

Finally, the Tiled State reads the map data and creates the game objects. We'll go through this state in more details, since it's the most important state of our game. I recommend that you take a look in the json file generated by Tiled, to have an idea of its structure. If you're confused about the Phaser map properties and methods, check Phaser documentation (http://phaser.io/docs/2.4.3/Phaser.Tilemap.html).

File TiledState.js in js/states:

```
1 var Platformer = Platformer || {};
2
3 Platformer.TiledState = function () {
4    "use strict";
5    Phaser.State.call(this);
6 };
7
8 Platformer.TiledState.prototype =
Object.create(Phaser.State.prototype);
9 Platformer.TiledState.prototype.constructor =
Platformer.TiledState;
10
11 Platformer.TiledState.prototype.init = function (level_data)
{
12    "use strict";
13    this.level_data = level_data;
14
15    this.scale.scaleMode = Phaser.ScaleManager.SHOW_ALL;
16    this.scale.pageAlignHorizontally = true;
17    this.scale.pageAlignVertically = true;
18
19    // start physics system
20    this.game.physics.startSystem(Phaser.Physics.ARCADE);
21    this.game.physics.arcade.gravity.y = 1000;
22
23    // create map and set tileset
24    this.map = this.game.add.tilemap(level_data.map.key);
```

```
25     this.map.addTilesetImage(this.map.tilesets[0].name,
level_data.map.tileset);
26 };
27
28 Platformer.TiledState.prototype.create = function () {
29     "use strict";
30     var group_name, object_layer, collision_tiles;
31
32     // create map layers
33     this.layers = {};
34     this.map.layers.forEach(function (layer) {
35         this.layers[layer.name] =
this.map.createLayer(layer.name);
36         if (layer.properties.collision) { // collision layer
37             collision_tiles = [];
38             layer.data.forEach(function (data_row) { // find
tiles used in the layer
39                 data_row.forEach(function (tile) {
40                     // check if it's a valid tile index and
isn't already in the list
41                     if (tile.index &gt; 0 &amp;&amp;
collision_tiles.indexOf(tile.index) === -1) {
42                         collision_tiles.push(tile.index);
43                     }
44                 }, this);
45             }, this);
46             this.map.setCollision(collision_tiles, true,
layer.name);
47         }
48     }, this);
49     // resize the world to be the size of the current layer
50     this.layers[this.map.layer.name].resizeWorld();
51
52     // create groups
53     this.groups = {};
54     this.level_data.groups.forEach(function (group_name) {
55         this.groups[group_name] = this.game.add.group();
56     }, this);
57
58     this.prefabs = {};
59
60     for (object_layer in this.map.objects) {
61         if (this.map.objects.hasOwnProperty(object_layer)) {
62             // create layer objects
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
63            this.map.objects[object_layer].forEach(this.create_
object, this);
64         }
65     }
66 };
67
68 Platformer.TiledState.prototype.create_object = function
(object) {
69     "use strict";
70     var position, prefab;
71     // tiled coordinates starts in the bottom left corner
72     position = {"x": object.x + (this.map.tileHeight / 2),
"y": object.y - (this.map.tileHeight / 2)};
73     // create object according to its type
74     switch (object.type) {
75     case "player":
76         prefab = new Platformer.Player(this, position,
object.properties);
77         break;
78     case "ground_enemy":
79         prefab = new Platformer.Enemy(this, position,
object.properties);
80         break;
81     case "flying_enemy":
82         prefab = new Platformer.FlyingEnemy(this, position,
object.properties);
83         break;
84     case "goal":
85         prefab = new Platformer.Goal(this, position,
object.properties);
86         break;
87     }
88     this.prefabs[object.name] = prefab;
89 };
90
91 Platformer.TiledState.prototype.restart_level = function () {
92     "use strict";
93     this.game.state.restart(true, false, this.level_data);
94 };
```

First, we have to tell Phaser what image represents each tileset we used in Tiled (the
Tiled tilesets are in this.map.tilesets). Since we have only one tileset, and we have the
image name from our json file, we can easily do that.

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Next, we have to create the map layers. The map object has a layers array that we will iterate. If a layer has the property collision that we added in Tiled, we have to make it available for collision. To do this, we need to tell Phaser which tiles can collide, so we iterate through layer.data, which contains all the layer tiles and add them to a list. At the end, we set the collision for all these tiles. After creating all layers, we resize the world to be the size of the current layer. Since all our layers have the same size, we don't care which one is the current layer.

The next step is to create the groups of our game. This can be easily done by iterating the groups array of our json file and adding a new group for each one of them. However, two things are important in this step: 1) the order of the groups define the order they are drawn on the screen; 2) groups must be created after layers, otherwise the layers would be drawn above them.

Finally, we go through all object layers (in our case, only one) and create the game objects. Since in our map we defined the object type, it is easy to instantiate the correct Prefab. Notice that our prefab position is not the same position of the Tiled object. That happens because Tiled coordinates start at the bottom left corner, while Phaser coordinates start at the top left corner. Also, we want our prefabs anchor point to be 0.5, so we have to set the position to be the center of our prefab.

Prefabs

In Phaser, prefabs are objects that extend Phaser.Sprite, acting as objects in our game. In our platformer game, we will need four prefabs: Player, Enemy, FlyingEnemy and Goal, which will all be explained now. This is our Prefab class:

File Prefab.js in js/prefabs:

```
1 var Platformer = Platformer || {};
2
3 Platformer.Prefab = function (game_state, position,
properties) {
4     "use strict";
5     Phaser.Sprite.call(this, game_state.game, position.x,
position.y, properties.texture);
6
7     this.game_state = game_state;
8
9     this.game_state.groups[properties.group].add(this);
10 };
11
```

```
12 Platformer.Prefab.prototype =
Object.create(Phaser.Sprite.prototype);
13 Platformer.Prefab.prototype.constructor = Platformer.Prefab;
```

Player

Our player will be able to walk, jump and kill enemies. For that we will need the
following properties: walking speed, jumping speed and bouncing, which are all
initialized in the constructor.

In the update method, we check all the player collision (collision layer and enemies) and
do the walking and jumping logic. There are some important details to notice regarding
player walking. First, we don't want to change the player direction while he's already
moving. For example, if he's moving left and the right arrow is pressed, we want the
player to keep moving left until the left arrow is released. So, we change the player
velocity only when the correct key is pressed and the player isn't moving in the opposite
direction. Second, we have to change the sprite direction accordingly. To do this, we use
the scale attribute of the sprite, which will invert the sprite direction.

To allow the player to jump, we can just check for the up arrow key in the update method
and change the velocity accordingly. The only important thing to notice here is that we
only want to allow the player to jump when it is touching the ground. Since the ground is
a tile, we have to use the blocked property of body, not touching (check the
documentation in http://phaser.io/docs/2.4.3/Phaser.Physics.Arcade.Body.html for more
information).

Another thing we have to check is if the player had fallen. For that, we check if the player
bottom y is equal to the world height. If so, the player dies.

Finally, the hit enemy method checks is called when the player collides with an enemy
and checks if the player is on top of the enemy. If that's the case, the enemy is killed,
otherwise the player dies.

File Player.js in js/prefabs:

```
1 var Platformer = Platformer || {};
2
3 Platformer.Player = function (game_state, position,
properties) {
4     "use strict";
5     Platformer.Prefab.call(this, game_state, position,
properties);
```

```
 6
 7     this.walking_speed = +properties.walking_speed;
 8     this.jumping_speed = +properties.jumping_speed;
 9     this.bouncing = +properties.bouncing;
10
11     this.game_state.game.physics.arcade.enable(this);
12     this.body.collideWorldBounds = true;
13
14     this.animations.add("walking", [0, 1, 2, 1], 6, true);
15
16     this.frame = 3;
17
18     this.anchor.setTo(0.5);
19
20     this.cursors =
this.game_state.game.input.keyboard.createCursorKeys();
21 };
22
23 Platformer.Player.prototype =
Object.create(Platformer.Prefab.prototype);
24 Platformer.Player.prototype.constructor = Platformer.Player;
25
26 Platformer.Player.prototype.update = function () {
27     "use strict";
28     this.game_state.game.physics.arcade.collide(this,
this.game_state.layers.collision);
29     this.game_state.game.physics.arcade.collide(this,
this.game_state.groups.enemies, this.hit_enemy, null, this);
30
31     if (this.cursors.right.isDown &amp;&amp;
this.body.velocity.x &gt;= 0) {
32         // move right
33         this.body.velocity.x = this.walking_speed;
34         this.animations.play("walking");
35         this.scale.setTo(-1, 1);
36     } else if (this.cursors.left.isDown &amp;&amp;
this.body.velocity.x &lt;= 0) {
37         // move left
38         this.body.velocity.x = -this.walking_speed;
39         this.animations.play("walking");
40         this.scale.setTo(1, 1);
41     } else {
42         // stop
43         this.body.velocity.x = 0;
44         this.animations.stop();
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
45          this.frame = 3;
46      }
47
48      // jump only if touching a tile
49      if (this.cursors.up.isDown &&
this.body.blocked.down) {
50          this.body.velocity.y = -this.jumping_speed;
51      }
52
53      // dies if touches the end of the screen
54      if (this.bottom >= this.game_state.game.world.height) {
55          this.game_state.restart_level();
56      }
57 };
58
59 Platformer.Player.prototype.hit_enemy = function (player,
enemy) {
60      "use strict";
61      // if the player is above the enemy, the enemy is killed,
otherwise the player dies
62      if (enemy.body.touching.up) {
63          enemy.kill();
64          player.y -= this.bouncing;
65      } else {
66          this.game_state.restart_level();
67      }
68 };
```

Enemy

Our enemy will be simple, it will only walk up to a maximum distance and then switch direction. For this, the properties we need are: walking speed, walking distance and direction. Notice that, in the constructor we set the initial velocity and scale according to the direction property. Also, we save the previous x position, which in the beginning is the sprite x.

In the update method we check if the walked distance (this.x – this.previous_x) is greater or equal to the maximum walking distance. If that's the case we switch the direction, updating the velocity, previous x and scale.

File Enemy.js in js/prefabs:

```
1 var Platformer = Platformer || {};
2
```

```
3 Platformer.Enemy = function (game_state, position, properties)
{
4     "use strict";
5     Platformer.Prefab.call(this, game_state, position,
properties);
6
7     this.walking_speed = +properties.walking_speed;
8     this.walking_distance = +properties.walking_distance;
9
10    // saving previous x to keep track of walked distance
11    this.previous_x = this.x;
12
13    this.game_state.game.physics.arcade.enable(this);
14    this.body.velocity.x = properties.direction *
this.walking_speed;
15
16    this.scale.setTo(-properties.direction, 1);
17
18    this.anchor.setTo(0.5);
19 };
20
21 Platformer.Enemy.prototype =
Object.create(Platformer.Prefab.prototype);
22 Platformer.Enemy.prototype.constructor = Platformer.Enemy;
23
24 Platformer.Enemy.prototype.update = function () {
25    "use strict";
26    this.game_state.game.physics.arcade.collide(this,
this.game_state.layers.collision);
27
28    // change the direction if walked the maximum distance
29    if (Math.abs(this.x - this.previous_x) &gt;=
this.walking_distance) {
30        this.body.velocity.x *= -1;
31        this.previous_x = this.x;
32        this.scale.setTo(-this.scale.x, 1);
33    }
34 };
```

Flying Enemy

Now that we have our regular enemy, creating a flying enemy is very easy. We will just create another prefab that extends Enemy, and that isn't affected by the gravity. Also, since your flying enemy asset is different, we have an animation too.

File FlyingEnemy.js in js/prefabs:

```
1 var Platformer = Platformer || {};
2
3 Platformer.FlyingEnemy = function (game_state, position,
properties) {
4     "use strict";
5     Platformer.Enemy.call(this, game_state, position,
properties);
6
7     // flying enemies are not affected by gravity
8     this.body.allowGravity = false;
9
10     this.animations.add("flying", [0, 1], 5, true);
11     this.animations.play("flying");
12 };
13
14 Platformer.FlyingEnemy.prototype =
Object.create(Platformer.Enemy.prototype);
15 Platformer.FlyingEnemy.prototype.constructor =
Platformer.FlyingEnemy;
```

Goal

Our goal is simple. It has a next level property, and overlap with the player. If the player reaches the goal, the next level should be started.

Notice that, to load the next level we only need to start the Boot State sending as parameter the path of the next level json file. In this tutorial, we will have only one level, but this structure makes it simple to make it a multilevel game. Try creating different levels, and see how it works.
File Goal.js in js/prefabs:

```
1 var Platformer = Platformer || {};
2
3 Platformer.Goal = function (game_state, position, properties)
{
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
4     "use strict";
5     Platformer.Prefab.call(this, game_state, position,
properties);
6
7     this.next_level = properties.next_level;
8
9     this.game_state.game.physics.arcade.enable(this);
10
11      this.anchor.setTo(0.5);
12 };
13
14 Platformer.Goal.prototype =
Object.create(Platformer.Prefab.prototype);
15 Platformer.Goal.prototype.constructor = Platformer.Goal;
16
17 Platformer.Goal.prototype.update = function () {
18     "use strict";
19     this.game_state.game.physics.arcade.collide(this,
this.game_state.layers.collision);
20     this.game_state.game.physics.arcade.overlap(this,
this.game_state.groups.players, this.reach_goal, null, this);
21 };
22
23 Platformer.Goal.prototype.reach_goal = function () {
24     "use strict";
25     // start the next level
26     this.game_state.game.state.start("BootState", true, false,
this.next_level);
27 };
```

Finishing the Game

Now that we have our game logic, we know what properties are necessary to be defined in the Tiled map. We can just set them on our map and our game is working!
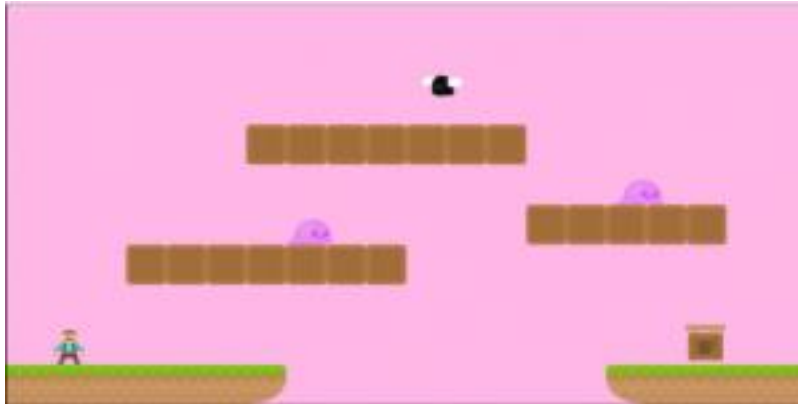
Platformer Tutorial with Phaser and Tiled – Part 2
By Renan Oliveira

In my last tutorial, we built a platformer game using Phaser and Tiled. In the following two tutorials we will add content to this game, making it more complete. In this tutorial, the following content will be added:

- Bigger levels, and two of them, so we can use the multilevel feature we implemented in the last tutorial.
- Smarter enemies, that don't fall from platforms.
- A new enemy, which follows the player when you get close to it.
- Player score, which will be increased when the player kill enemies or collect coins.
- A checkpoint, so if the player dies after reach it, it will be respawn at the checkpoint position.

For this tutorial, it is necessary that you finished the last tutorial, since we will continue from the code we already wrote.

Source code files

You can download the tutorial source code files here.

Game States

In this tutorial, we will use the same states from the last one:

- Boot State: loads a json file with the level information and starts the Loading State.
- Loading Sate: loads all the game assets, and starts the Level State.
- Tiled State: creates the map and all game objects.

The code for Boot and Loading states are exactly the same as before, so I will omit them. On the other hand, the Tiled State code had some changes. Some of those changes are necessary because of the new content added to the game and will be explained throughout the tutorial. However, there is one change that was made to make the code simpler, and I will explain it now.

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

In the previous tutorial, we had a "create_object" method that used a switch case to instantiate the correct prefab. To avoid having to add another case condition for each prefab we create, we will change it to use an object that maps each prefab type to its constructor, as shown below:

```
1 var Platformer = Platformer || {};
2
3 Platformer.TiledState = function () {
4     "use strict";
5     Phaser.State.call(this);
6
7     this.prefab_classes = {
8         "player": Platformer.Player.prototype.constructor,
9         "ground_enemy":
Platformer.GroundEnemy.prototype.constructor,
10        "flying_enemy":
Platformer.FlyingEnemy.prototype.constructor,
11        "running_enemy":
Platformer.RunningEnemy.prototype.constructor,
12        "goal": Platformer.Goal.prototype.constructor,
13        "checkpoint":
Platformer.Checkpoint.prototype.constructor,
14        "coin": Platformer.Coin.prototype.constructor,
15        "score": Platformer.Score.prototype.constructor
16    };
17 };
18
19 Platformer.TiledState.prototype =
Object.create(Phaser.State.prototype);
20 Platformer.TiledState.prototype.constructor =
Platformer.TiledState;
21
22 Platformer.TiledState.prototype.init = function (level_data)
{
23     "use strict";
24     var tileset_index;
25     this.level_data = level_data;
26
27     this.scale.scaleMode = Phaser.ScaleManager.SHOW_ALL;
28     this.scale.pageAlignHorizontally = true;
29     this.scale.pageAlignVertically = true;
30
31     // start physics system
32     this.game.physics.startSystem(Phaser.Physics.ARCADE);
```

```
33      this.game.physics.arcade.gravity.y = 1000;
34
35      // create map and set tileset
36      this.map = this.game.add.tilemap(level_data.map.key);
37      tileset_index = 0;
38      this.map.tilesets.forEach(function (tileset) {
39          this.map.addTilesetImage(tileset.name,
level_data.map.tilesets[tileset_index]);
40          tileset_index += 1;
41      }, this);
42  };
43
44  Platformer.TiledState.prototype.create = function () {
45      "use strict";
46      var group_name, object_layer, collision_tiles;
47
48      // create map layers
49      this.layers = {};
50      this.map.layers.forEach(function (layer) {
51          this.layers[layer.name] =
this.map.createLayer(layer.name);
52          if (layer.properties.collision) { // collision layer
53              collision_tiles = [];
54              layer.data.forEach(function (data_row) { // find
tiles used in the layer
55                  data_row.forEach(function (tile) {
56                      // check if it's a valid tile index and
isn't already in the list
57                      if (tile.index > 0 &&
collision_tiles.indexOf(tile.index) === -1) {
58                          collision_tiles.push(tile.index);
59                      }
60                  }, this);
61              }, this);
62              this.map.setCollision(collision_tiles, true,
layer.name);
63          }
64      }, this);
65      // resize the world to be the size of the current layer
66      this.layers[this.map.layer.name].resizeWorld();
67
68      // create groups
69      this.groups = {};
70      this.level_data.groups.forEach(function (group_name) {
71          this.groups[group_name] = this.game.add.group();
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
72      }, this);
73
74      this.prefabs = {};
75
76      for (object_layer in this.map.objects) {
77          if (this.map.objects.hasOwnProperty(object_layer)) {
78              // create layer objects
79              this.map.objects[object_layer].forEach(this.create
_object, this);
80          }
81      }
82
83      this.game.camera.follow(this.prefabs.player);
84 };
85
86 Platformer.TiledState.prototype.create_object = function
(object) {
87      "use strict";
88      var position, prefab;
89      // tiled coordinates starts in the bottom left corner
90      position = {"x": object.x + (this.map.tileHeight / 2),
"y": object.y - (this.map.tileHeight / 2)};
91      // create object according to its type
92      if (this.prefab_classes.hasOwnProperty(object.type)) {
93          prefab = new this.prefab_classes[object.type](this,
position, object.properties);
94      }
95      this.prefabs[object.name] = prefab;
96 };
97
98 Platformer.TiledState.prototype.restart_level = function () {
99      "use strict";
100     // restart the game only if the checkpoint was not
reached
101     if (this.prefabs.checkpoint.checkpoint_reached) {
102         this.prefabs.player.x = this.prefabs.checkpoint.x;
103         this.prefabs.player.y = this.prefabs.checkpoint.y;
104     } else {
105         this.game.state.restart(true, false,
this.level_data);
106     }
107 };
```

Notice that, in the TiledState constructor we create an object with all prefab types and their corresponding constructors. So, in the "create_object" method we can call the correct constructor using the type property from the map object. This is only possible because all prefabs use the same constructor. So, if you created your prefabs with different constructors you can change them to have the same constructor, or keep using the switch case strategy, as in the previous tutorial. Remember that all code I show here is based on my preference, and you should write it the way you think it's best.

New levels

Those are the two levels I created for this tutorial. Feel free to create your owns, trying to add different tiles and enemies. In those new maps, there are already the new prefabs I
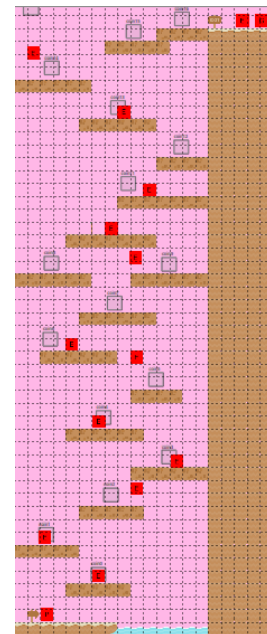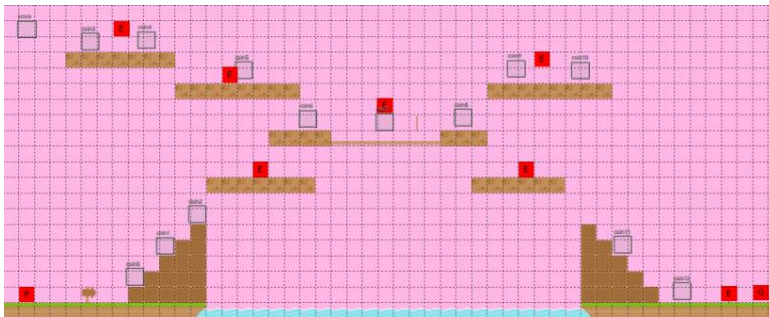
will explain later in the tutorial, there is, however, something I would like to show first.

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

In the first level, I added a bridge between two platforms, as you can see in the image. This bridge sprite is smaller than the tile, and I wanted its collision box to be smaller too. So, I had to add a new tileset using the same tileset image but different tile sizes. The tile size used was 35 pixels width and 11 pixels height with 1 pixel margin and spacing (as shown below). It is also possible to insert the bridge as an object, not a tile, but I found
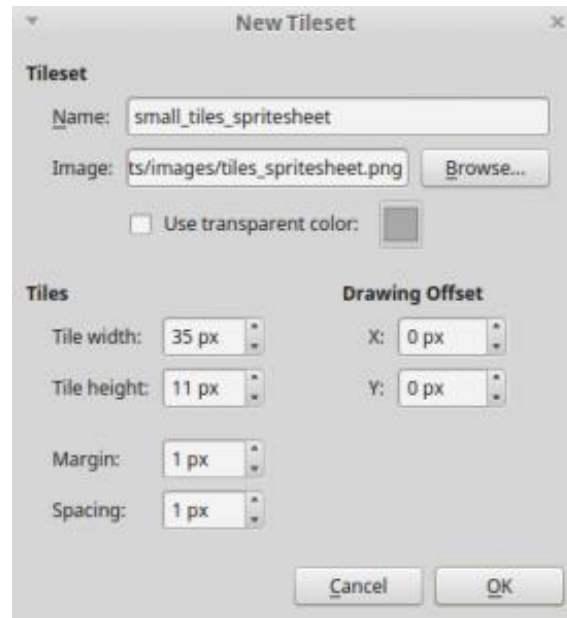
this way easier.

Also, notice that I divided the background layer in two: sky and background. This was done to add some background decoration, like the signs at the beginning and end of the level.

Smarter enemy

We will change our enemy so it will switch direction if arrives the edge of platform. The idea is to check if the next ground position is occupied by a tile and, if not so, switch the enemy direction. To find this next ground position we have to calculate its x and y coordinates. The y coordinate is just the enemy bottom y plus 1, since that's the y coordinate immediately below the enemy. The x coordinate depends on the enemy direction. So, it will be the enemy x coordinate plus tile width, if the enemy is moving right, or minus tile width if the enemy is moving left.

To check if a position is occupied by a tile, we will use the tilemap method "getTileWordlXY", which returns the tile object of a layer in a given position (for more

information, check Phaser documentation:
http://phaser.io/docs/2.4.3/Phaser.Tilemap.html). If the method returns a tile, we know
the enemy is not in the platform edge yet, so it can keep moving. If it returns null, the
enemy must switch direction to avoid falling. In this code, the method "switch_direction"
is the code we already had in the Enemy prefab, I just moved it to a function (you can
check it in the source code).

```
1 var Platformer = Platformer || {};
2
3 Platformer.GroundEnemy = function (game_state, position,
properties) {
4    "use strict";
5    Platformer.Enemy.call(this, game_state, position,
properties);
6 };
7
8 Platformer.GroundEnemy.prototype =
Object.create(Platformer.Enemy.prototype);
9 Platformer.GroundEnemy.prototype.constructor =
Platformer.GroundEnemy;
10
11 Platformer.GroundEnemy.prototype.update = function () {
12    "use strict";
13    Platformer.Enemy.prototype.update.call(this);
14
15    if (this.body.blocked.down && !this.has_tile_to_walk()) {
16        this.switch_direction();
17    }
18 };
19
20 Platformer.GroundEnemy.prototype.has_tile_to_walk = function
() {
21    "use strict";
22    var direction, position_to_check, map, next_tile;
23    direction = (this.body.velocity.x < 0) ? -1 : 1;
24    // check if the next position has a tile
25    position_to_check = new Phaser.Point(this.x + (direction *
this.game_state.map.tileWidth), this.bottom + 1);
26    map = this.game_state.map;
27    // getTileWorldXY returns the tile in a given position
28    next_tile = map.getTileWorldXY(position_to_check.x,
position_to_check.y, map.tileWidth, map.tileHeight,
"collision");
29    return next_tile !== null;
```

```
30 };
```

New enemy

Also, we will add an enemy with the following behavior:

- If the player is outside a detection range, it will act like a regular enemy.
- If the player is inside a detection range, it will increase its velocity and go in the player direction.

To do that, we will create a new prefab that extends the GroundEnemy prefab. Besides all the regular enemy properties, it will have a detection distance and a running speed. In the update method, we check if the player is inside the detection range. If so, the enemy change its velocity to the running speed and follows the player. Otherwise, it just calls the GroundEnemy update method.

To check if the player is inside the detection range, we calculate the distance between the player and enemy x positions. If the absolute value of this distance is less than the detection distance and both the player and enemy bottom y positions are the same (meaning they're standing in the same ground), the enemy detects the player.

```
1 var Platformer = Platformer || {};
2
3 Platformer.RunningEnemy = function (game_state, position,
properties) {
4     "use strict";
5     Platformer.GroundEnemy.call(this, game_state, position,
properties);
6
7     this.detection_distance = +properties.detection_distance;
8     this.running_speed = +properties.running_speed;
9 };
10
11 Platformer.RunningEnemy.prototype =
Object.create(Platformer.GroundEnemy.prototype);
12 Platformer.RunningEnemy.prototype.constructor =
Platformer.RunningEnemy;
13
14 Platformer.RunningEnemy.prototype.update = function () {
15     "use strict";
16     var direction;
17     this.game_state.game.physics.arcade.collide(this,
this.game_state.layers.collision);
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
18
19    if (this.detect_player()) {
20        // player is inside detection range, run towards it
21        direction = (this.game_state.prefabs.player.x <
this.x) ? -1 : 1;
22        this.body.velocity.x = direction * this.running_speed;
23        this.scale.setTo(-direction, 1);
24        this.previous_x = this.x;
25    } else {
26        // player is outside detection range, act like a
regular enemy
27        direction = (this.body.velocity.x < 0) ? -1 : 1;
28        this.body.velocity.x = direction * this.walking_speed;
29        this.scale.setTo(-direction, 1);
30        Platformer.GroundEnemy.prototype.update.call(this);
31    }
32 };
33
34 Platformer.RunningEnemy.prototype.detect_player = function ()
{
35    "use strict";
36    var distance_to_player;
37    distance_to_player = Math.abs(this.x -
this.game_state.prefabs.player.x);
38    // the player must be in the same ground y position, and
inside the detection range
39    return (this.bottom ===
this.game_state.prefabs.player.bottom) && (distance_to_player <=
this.detection_distance);
40 };
```

Player score

The player score will increase every time it kills an enemy or collect a coin. For this, we have to add a score property to the Enemy prefab and create a Coin prefab, as follows.

```
1 var Platformer = Platformer || {};
2
3 Platformer.Coin = function (game_state, position, properties)
{
4     "use strict";
5     Platformer.Prefab.call(this, game_state, position,
properties);
6
7     this.score = +properties.score;
8
9     this.game_state.game.physics.arcade.enable(this);
10     this.body.immovable = true;
11     this.body.allowGravity = false;
12
13     this.anchor.setTo(0.5);
14 };
15
16 Platformer.Coin.prototype =
Object.create(Platformer.Prefab.prototype);
17 Platformer.Coin.prototype.constructor = Platformer.Coin;
18
19 Platformer.Coin.prototype.update = function () {
20     "use strict";
21     this.game_state.game.physics.arcade.collide(this,
this.game_state.layers.collision);
22     this.game_state.game.physics.arcade.overlap(this,
this.game_state.groups.players, this.collect_coin, null, this);
23 };
24
25 Platformer.Coin.prototype.collect_coin = function (coin,
player) {
26     "use strict";
27     // kill coin and increase score
28     this.kill();
29     player.score += this.score;
30 };
```

For the enemy killing, we just have to increase the player score in the "hit_enemy" method, while for collecting coins we check for overlaps between the player and the

coins, and increase the player score according to the coin score. To keep track of the player score, we will add a score property in the Player as well.

```
1 var Platformer = Platformer || {};
2
3 Platformer.Player = function (game_state, position,
properties) {
4     "use strict";
5     Platformer.Prefab.call(this, game_state, position,
properties);
6
7     this.walking_speed = +properties.walking_speed;
8     this.jumping_speed = +properties.jumping_speed;
9     this.bouncing = +properties.bouncing;
10     this.score = 0;
11
12     this.game_state.game.physics.arcade.enable(this);
13     this.body.collideWorldBounds = true;
14
15     this.animations.add("walking", [0, 1, 2, 1], 6, true);
16
17     this.frame = 3;
18
19     this.anchor.setTo(0.5);
20
21     this.cursors =
this.game_state.game.input.keyboard.createCursorKeys();
22 };
23
24 Platformer.Player.prototype =
Object.create(Platformer.Prefab.prototype);
25 Platformer.Player.prototype.constructor = Platformer.Player;
26
27 Platformer.Player.prototype.update = function () {
28     "use strict";
29     this.game_state.game.physics.arcade.collide(this,
this.game_state.layers.collision);
30     this.game_state.game.physics.arcade.overlap(this,
this.game_state.groups.enemies, this.hit_enemy, null, this);
31
32     if (this.cursors.right.isDown && this.body.velocity.x >=
0) {
33         // move right
34         this.body.velocity.x = this.walking_speed;
```

```
35              this.animations.play("walking");
36              this.scale.setTo(-1, 1);
37         } else if (this.cursors.left.isDown &&
this.body.velocity.x <= 0) {
38              // move left
39              this.body.velocity.x = -this.walking_speed;
40              this.animations.play("walking");
41              this.scale.setTo(1, 1);
42         } else {
43              // stop
44              this.body.velocity.x = 0;
45              this.animations.stop();
46              this.frame = 3;
47         }
48
49         // jump only if touching a tile
50         if (this.cursors.up.isDown && this.body.blocked.down) {
51              this.body.velocity.y = -this.jumping_speed;
52         }
53
54         // dies if touches the end of the screen
55         if (this.bottom >= this.game_state.game.world.height) {
56              this.game_state.restart_level();
57         }
58 };
59
60 Platformer.Player.prototype.hit_enemy = function (player,
enemy) {
61     "use strict";
62     // if the player is above the enemy, the enemy is killed,
otherwise the player dies
63     if (enemy.body.touching.up) {
64          this.score += enemy.score;
65          enemy.kill();
66          player.y -= this.bouncing;
67     } else {
68          this.game_state.restart_level();
69     }
70 };
```

To show the score in the screen, we create a Score prefab, that extends Phaser.Text instead of Phaser.Sprite. Notice that Phaser.Text extends Phaser.Sprite too, so our prefab still is a Phaser sprite. This Score prefab must be fixed to the camera, so it won't move

with the player and in the update method it will change its text to be equivalent to the player current score.

```
1 var Platformer = Platformer || {};
2
3 Platformer.Score = function (game_state, position, properties)
{
4     "use strict";
5     Phaser.Text.call(this, game_state.game, position.x,
position.y, properties.text);
6
7     this.game_state = game_state;
8
9     this.game_state.groups[properties.group].add(this);
10
11     this.fixedToCamera = true;
12 };
13
14 Platformer.Score.prototype =
Object.create(Phaser.Text.prototype);
15 Platformer.Score.prototype.constructor = Platformer.Score;
16
17 Platformer.Score.prototype.update = function () {
18     "use strict";
19     // update text to player current score
20     this.text = "Score: " +
this.game_state.prefabs.player.score;
21 };
```

Here is an example of the score in our game. Try it yourself!

Checkpoint

Our checkpoint will be simple, if the player dies after touching the checkpoint, it will be respawned in the checkpoint position instead of its initial position. For this, we will add a checkpoint prefab that checks for overlap with the player, and if that happens, set a checkpoint variable to true. So, in the "restart_level" method from TiledState (presented above) we now check if the checkpoint was reached. If so, we just respawn the player to the checkpoint position, instead of restarting the level.

```
1 var Platformer = Platformer || {};
2
3 Platformer.Checkpoint = function (game_state, position,
properties) {
4     "use strict";
5     Platformer.Prefab.call(this, game_state, position,
properties);
6
7     this.checkpoint_reached = false;
8
9     this.game_state.game.physics.arcade.enable(this);
10
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
11      this.anchor.setTo(0.5);
12 };
13
14 Platformer.Checkpoint.prototype =
Object.create(Platformer.Prefab.prototype);
15 Platformer.Checkpoint.prototype.constructor =
Platformer.Checkpoint;
16
17 Platformer.Checkpoint.prototype.update = function () {
18      "use strict";
19      this.game_state.game.physics.arcade.collide(this,
this.game_state.layers.collision);
20      this.game_state.game.physics.arcade.overlap(this,
this.game_state.groups.players, this.reach_checkpoint, null,
this);
21 };
22
23 Platformer.Checkpoint.prototype.reach_checkpoint = function
() {
24      "use strict";
25      // checkpoint was reached
26      this.checkpoint_reached = true;
27 };
```


Here is an example of the checkpoint in our game. Try it yourself!

Finishing the game

Now that we have all the game code, we just have to add the prefab properties in our Tiled map, as we did in the last tutorial, and work game is working!

Platformer Tutorial with Phaser and Tiled – Part 3
By Renan Oliveira

Until my last platformer tutorial, we created a simple platformer game with some nice content, however there are still some things to add before making it playable. In this tutorial, we will add the following content:

- Player lives, so now you can actually lose.
- Items that will increase the player lives or give an attack.
- A level boss.

Source code files

You can download the tutorial source code files here.

Game states

In this tutorial, we will use the same states from the last one:

- Boot State: loads a json file with the level information and starts the Loading State.
- Loading Sate: loads all the game assets, and starts the Level State.
- Tiled State: creates the map and all game objects.

The code for all states are almost the same, except for two methods added in TiledState: one to do the game over and other to initialize the hud. So, I will omit them for now and only show the changes when they're necessary.

Player lives

Our player will start with a given number of lives, and it will lose one every time it is killed by an enemy. To do that, we will change the Player prefab to have a lives property and the "die" method to decrease the number of lives if the player was killed, as follows:

```
1 Platformer.Player.prototype.die = function () {
2     "use strict";
3     this.lives -= 1;
4     this.shooting = false;
5     if (this.lives > 0) {
6         this.game_state.restart_level();
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
7    } else {
8        this.game_state.game_over();
9    }
10 };
```

Also, if the player number of lives becomes zero, we will call the "game_over" method in TiledState, instead of "restart_level":

```
1 Platformer.TiledState.prototype.game_over = function () {
2    "use strict";
3    localStorage.clear();
4    this.game.state.start("BootState", true, false,
"assets/levels/level1.json");
5 };
```

Besides having the player lives, we have to show how many lives the player still have. So, we will create a Lives prefab that belongs to the hud group and show the current player lives. Our Lives prefab will have the player live asset, but it will be invisible, since we will use the asset only to create new sprites showing the player lives.

```
1 var Platformer = Platformer || {};
2
3 Platformer.Lives = function (game_state, position, properties)
{
4    "use strict";
5    Platformer.Prefab.call(this, game_state, position,
properties);
6
7    this.frame = +properties.frame;
8    this.visible = false;
9
10     this.spacing = +properties.spacing;
11
12     this.fixedToCamera = true;
13     // saving initial position if it gets changed by window
scaling
14     this.initial_position = new Phaser.Point(this.x, this.y);
15
16     this.lives = [];
17     this.dead_life = null;
18     this.create_lives();
19 };
20
```

```
21 Platformer.Lives.prototype =
Object.create(Platformer.Prefab.prototype);
22 Platformer.Lives.prototype.constructor = Platformer.Lives;
23
24 Platformer.Lives.prototype.update = function () {
25     "use strict";
26     // update to show current number of lives
27     if (this.game_state.prefabs.player.lives !==
this.lives.length) {
28         this.update_lives();
29     }
30 };
31
32 Platformer.Lives.prototype.create_lives = function () {
33     "use strict";
34     var life_index, life_position, life;
35     // create a sprite for each one of the player lives
36     for (life_index = 0; life_index <
this.game_state.prefabs.player.lives; life_index += 1) {
37         life_position = new
Phaser.Point(this.initial_position.x + (life_index * (this.width
+ this.spacing)), this.initial_position.y);
38         life = new Phaser.Sprite(this.game_state.game,
life_position.x, life_position.y, this.texture, this.frame);
39         life.fixedToCamera = true;
40         this.game_state.groups.hud.add(life);
41         this.lives.push(life);
42     }
43 };
44
45 Platformer.Lives.prototype.update_lives = function () {
46     "use strict";
47     var life, life_position;
48     life = this.lives[this.lives.length - 1];
49     if (this.game_state.prefabs.player.lives <
this.lives.length) {
50         // the player died, so we have to kill the last life
51         life.kill();
52         this.dead_life = life;
53         this.lives.pop();
54     } else {
55         // the player received another life
56         if (!this.dead_life) {
57             // if there is no dead life we can reuse, we
create a new one
```

```
58              life_position = new
Phaser.Point(this.initial_position.x + (this.lives.length *
(this.width + this.spacing)), this.initial_position.y);
59              life = new Phaser.Sprite(this.game_state.game,
life_position.x, life_position.y, this.texture, this.frame);
60              life.fixedToCamera = true;
61              this.game_state.groups.hud.add(life);
62          } else {
63              // if there is a dead life, we just reset it
64              life = this.dead_life;
65              life_position = new
Phaser.Point(this.initial_position.x + ((this.lives.length - 1)
* (this.width + this.spacing)), this.initial_position.y);
66              life.reset(life_position.x, life_position.y);
67              this.dead_life = null;
68          }
69          this.lives.push(life);
70      }
71 };
```

The Lives prefab will have a "create_lives" method that fills an array with a sprite for each one of the player lives. Since the player live asset is already loaded we can use its width and the Lives prefab position to find the position for each live and draw them on the screen. Finally, in the update method we have to check if the player number of lives has changed and, if so, we call an "update_lives" method.

In the "update_lives" method we assume the number of lives can increase or decrease by only one between two updates. This makes sense because the update method will be called many times per second, and the player can't die or get lives faster than that. So, in the "update_lives" method we only check if the number of lives has decreased or increased. In the first case, we have to kill the last live in the array. In the second case, we'll do some checking to avoid creating too many life objects. First, we check if there is a dead life we can reuse and, if so, we just reset it. Otherwise, we create a new life object.

There are two last things we have to do regarding the player lives and the player score that we didn't do in the last tutorial. First, you'll notice I added an "init_hud" method in TiledState (shown below) that create the hud objects in fixed positions instead of loading it from the Tiled map. I did this because sometimes the Phaser world scaling could mess with the hud objects positions when reloading the screen or updating the lives. I also save the lives prefab initial position for the same reason.

```
1 Platformer.TiledState.prototype.init_hud = function () {
2     "use strict";
```

```
3    var score_position, score, lives_position, lives;
4    score_position = new Phaser.Point(20, 20);
5    score = new Platformer.Score(this, score_position, {"text":
"Score: 0", "group": "hud"});
6    this.prefabs["score"] = score;
7
8    lives_position = new Phaser.Point(this.game.world.width *
0.65, 20);
9    lives = new Platformer.Lives(this, lives_position,
{"texture": "player_spritesheet", "group": "hud", "frame": 3,
"spacing": 16});
10    this.prefabs["lives"] = lives;
11 };
```

Second, we have to save the player lives and score before loading a new level, otherwise it will be restarted. For this, we will save this information in the browser localStorage when reaching the goal. Also, in the Player constructor, we first check if there's already a previous score or lives saved in the localStorage and if so, load it. Finally, in TiledState "game_over" method we clear the localStorage.

```
1 Platformer.Goal.prototype.reach_goal = function () {
2    "use strict";
3    // start the next level
4    localStorage.player_lives =
this.game_state.prefabs.player.lives;
5    localStorage.player_score =
this.game_state.prefabs.player.score;
6    this.game_state.game.state.start("BootState", true, false,
this.next_level);
7 };
```
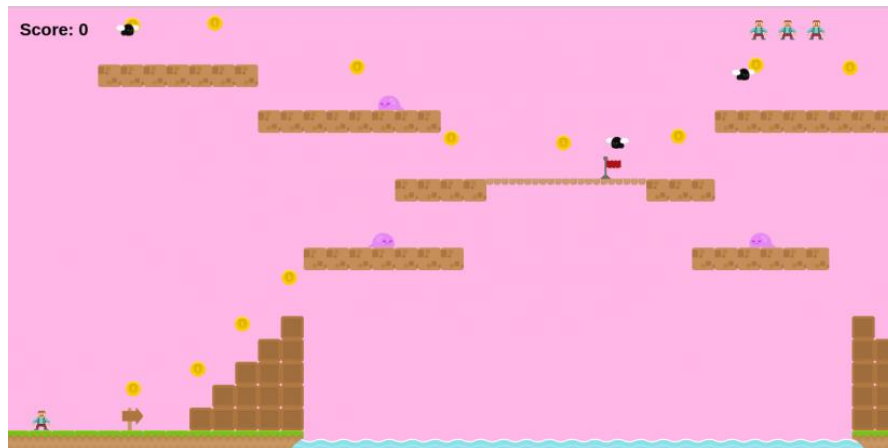
We can add the Lives prefab to our current levels and see how it's working already:

Items

We will create two different items:

1   A LifeItem, that will increase the player number of lives.
2   A ShootingItem, that will give the player the ability to attack.

First, we will create a generic Item prefab to reunite the common code between both items. All items will have an immovable physics body initialized in the constructor. Also, in the update method it will check for overlap with the player and call a "collect_item" method if that happens. By default, the "collect_item" method will only kill the Item, but we will overwrite it in our new items prefabs to do what we want.

```
1 var Platformer = Platformer || {};
2
3 Platformer.Item = function (game_state, position, properties)
{
4     "use strict";
5     Platformer.Prefab.call(this, game_state, position,
properties);
6
7     this.game_state.game.physics.arcade.enable(this);
8     this.body.immovable = true;
9     this.body.allowGravity = false;
10
11      this.anchor.setTo(0.5);
12 };
```

```
13
14 Platformer.Item.prototype =
Object.create(Platformer.Prefab.prototype);
15 Platformer.Item.prototype.constructor = Platformer.Item;
16
17 Platformer.Item.prototype.update = function () {
18     "use strict";
19     this.game_state.game.physics.arcade.collide(this,
this.game_state.layers.collision);
20     this.game_state.game.physics.arcade.overlap(this,
this.game_state.groups.players, this.collect_item, null, this);
21 };
22
23 Platformer.Item.prototype.collect_item = function () {
24     "use strict";
25     // by default, the item is destroyed when collected
26     this.kill();
27 };
```

Now, our LifeItem will be really simple. We only need to overwrite the "collect_item" method to increase the player number of lives after calling the original "collect_item" method, as below:

```
1 var Platformer = Platformer || {};
2
3 Platformer.LifeItem = function (game_state, position,
properties) {
4     "use strict";
5     Platformer.Item.call(this, game_state, position,
properties);
6 };
7
8 Platformer.LifeItem.prototype =
Object.create(Platformer.Item.prototype);
9 Platformer.LifeItem.prototype.constructor =
Platformer.LifeItem;
10
11 Platformer.LifeItem.prototype.collect_item = function (item,
player) {
12     "use strict";
13     Platformer.Item.prototype.collect_item.call(this);
14     player.lives += 1;
15 };
```

We can already see our game working with the LifeItem:
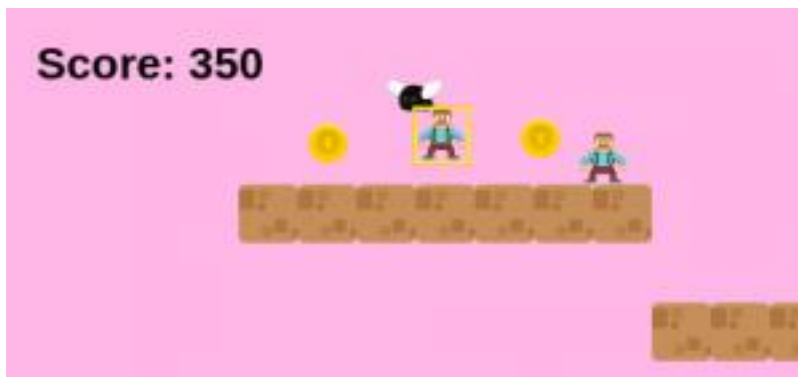
The ShootingItem is also simple, we only need to set a shooting variable in the Player prefab to true. However, now we have to add the shooting logic in the Player prefab.

```
1 var Platformer = Platformer || {};
2
3 Platformer.FireballItem = function (game_state, position,
properties) {
4     "use strict";
5     Platformer.Item.call(this, game_state, position,
properties);
6 };
7
8 Platformer.FireballItem.prototype =
Object.create(Platformer.Item.prototype);
9 Platformer.FireballItem.prototype.constructor =
Platformer.FireballItem;
10
11 Platformer.FireballItem.prototype.collect_item = function
(item, player) {
12     "use strict";
13     Platformer.Item.prototype.collect_item.call(this);
14     player.shooting = true;
15 };
```

To give the player the ability to shoot fireballs, we will first create a Fireball prefab. The Fireball prefab will create a physical body with a constant velocity given by its direction. Also, we will check for collisions, and when they happen, kill it.

```
1 var Platformer = Platformer || {};
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
 2
 3 Platformer.Fireball = function (game_state, position,
properties) {
 4     "use strict";
 5     Platformer.Prefab.call(this, game_state, position,
properties);
 6
 7     this.direction = properties.direction;
 8     this.speed = +properties.speed;
 9
10      this.game_state.game.physics.arcade.enable(this);
11      this.body.allowGravity = false;
12      // velocity is constant, but defined by direction
13      if (this.direction == "LEFT") {
14          this.body.velocity.x = -this.speed;
15      } else {
16          this.body.velocity.x = this.speed;
17      }
18
19     this.anchor.setTo(0.5);
20     // Fireball uses the same asset as FireballItem, so we
make it a little smaller
21     this.scale.setTo(0.75);
22 };
23
24 Platformer.Fireball.prototype =
Object.create(Platformer.Prefab.prototype);
25 Platformer.Fireball.prototype.constructor =
Platformer.Fireball;
26
27 Platformer.Fireball.prototype.update = function () {
28     "use strict";
29     // the fireball is destroyed if in contact with anything
else
30     this.game_state.game.physics.arcade.collide(this,
this.game_state.layers.collision, this.kill, null, this);
31      this.game_state.game.physics.arcade.overlap(this,
this.game_state.layers.invincible_enemies, this.kill, null,
this);
32 };
```

With the Fireball prefab created, we have to add the ability to shoot them in the Player prefab. For this, we start checking in the update method if the player is able to shoot and

if the shooting key was pressed (SPACEBAR). If so, we start a timer (created in the constructor) which will call the shoot method in a loop.

```
1 Platformer.Player.prototype.update = function () {
2     "use strict";
3     this.game_state.game.physics.arcade.collide(this,
this.game_state.layers.collision);
4     this.game_state.game.physics.arcade.overlap(this,
this.game_state.groups.enemies, this.hit_enemy, null, this);
5     // the player automatically dies if in contact with
invincible enemies or enemy fireballs
6     this.game_state.game.physics.arcade.overlap(this,
this.game_state.groups.invincible_enemies, this.die, null,
this);
7     this.game_state.game.physics.arcade.overlap(this,
this.game_state.groups.enemy_fireballs, this.die, null, this);
8
9     if (this.cursors.right.isDown && this.body.velocity.x >= 0)
{
10         // move right
11         this.body.velocity.x = this.walking_speed;
12         this.direction = "RIGHT";
13         this.animations.play("walking");
14         this.scale.setTo(-1, 1);
15     } else if (this.cursors.left.isDown &&
this.body.velocity.x <= 0) {
16         // move left
17         this.body.velocity.x = -this.walking_speed;
18         this.direction = "LEFT";
19         this.animations.play("walking");
20         this.scale.setTo(1, 1);
21     } else {
22         // stop
23         this.body.velocity.x = 0;
24         this.animations.stop();
25         this.frame = 3;
26     }
27
28     // jump only if touching a tile
29     if (this.cursors.up.isDown && this.body.blocked.down) {
30         this.body.velocity.y = -this.jumping_speed;
31     }
32
33     // dies if touches the end of the screen
```

```
34    if (this.bottom >= this.game_state.game.world.height) {
35        this.die();
36    }
37
38    // if the player is able to shoot and the shooting button
   is pressed, start shooting
39    if (this.shooting &&
   this.game_state.input.keyboard.isDown(Phaser.Keyboard.SPACEBAR))
   {
40        if (!this.shoot_timer.running) {
41            this.shoot();
42            this.shoot_timer.start();
43        }
44    } else {
45        this.shoot_timer.stop(false);
46    }
47 };
```

In the shoot method we will use a concept called pool of objects. To avoid create and deleting a lot of objects, which could negatively impact our game performance, we will keep a group of Fireballs (called a pool) and every time we have to create another Fireball we check if there isn't one dead Fireball in the pool. If so, we just reset it in the new position. Otherwise, we create a new one. This is similar to what we have done in the Lives prefab, and it is expected that as the game continues, we will have created all the necessary fireballs, and will start to reuse old Fireballs instead of creating new ones.

```
1 Platformer.Player.prototype.shoot = function () {
2    "use strict";
3    var fireball, fireball_position, fireball_properties;
4    // get the first dead fireball from the pool
5    fireball = this.game_state.groups.fireballs.getFirstDead();
6    fireball_position = new Phaser.Point(this.x, this.y);
7    if (!fireball) {
8        // if there is no dead fireball, create a new one
9        fireball_properties = {"texture": "fireball_image",
   "group": "fireballs", "direction": this.direction, "speed":
   this.attack_speed};
10        fireball = new Platformer.Fireball(this.game_state,
   fireball_position, fireball_properties);
11    } else {
12        // if there is a dead fireball, reset it in the new
   position
13        fireball.reset(fireball_position.x,
   fireball_position.y);
```

```
14          fireball.body.velocity.x = (this.direction == "LEFT")
? -this.attack_speed : this.attack_speed;
15      }
16 };
```


Now, we can see our game working with fireballs:

Boss level

For the boss level I added a new enemy and the boss, which are both invincible, so if the player touches them, he will always die. This can be done by adding a new group (called "invincible_enemies", which was actually present in some of the already shown code) and, if the player overlaps with it, he automatically dies.

Stone enemy

The stone enemy will be stopped in the ceiling until the player goes below it. When that happens, the enemy will fall over the player. To do that, in the update method we will check if the player is below the enemy, and if so, the enemy start falling.

To check for the player, we just have to compare the player x position to the enemy left and right coordinates, and the player y position must be below the enemy. To make the enemy falls, we just have to change its allowGravity property to true, as follows:

```
1 var Platformer = Platformer || {};
2
3 Platformer.StoneEnemy = function (game_state, position,
properties) {
4     "use strict";
5     Platformer.Prefab.call(this, game_state, position,
properties);
6
7     this.game_state.game.physics.arcade.enable(this);
8     this.body.allowGravity = false;
9
10    this.anchor.setTo(0.5);
11 };
12
13 Platformer.StoneEnemy.prototype =
Object.create(Platformer.Prefab.prototype);
14 Platformer.StoneEnemy.prototype.constructor =
Platformer.StoneEnemy;
15
16 Platformer.StoneEnemy.prototype.update = function () {
17     "use strict";
18     this.game_state.game.physics.arcade.collide(this,
this.game_state.layers.collision);
19
20     // if the player is below, the enemy will fall after some
time
```

```
21    if (this.check_player()) {
22        this.fall();
23    }
24 };
25
26 Platformer.StoneEnemy.prototype.check_player = function () {
27    "use strict";
28    var player;
29    player = this.game_state.prefabs.player;
30    // check if player is right below the enemy
31    return (player.x > this.left && player.x < this.right &&
player.y > this.bottom);
32 };
33
34 Platformer.StoneEnemy.prototype.fall = function () {
35    "use strict";
36    // start falling
37    this.body.allowGravity = true;
38 };
```
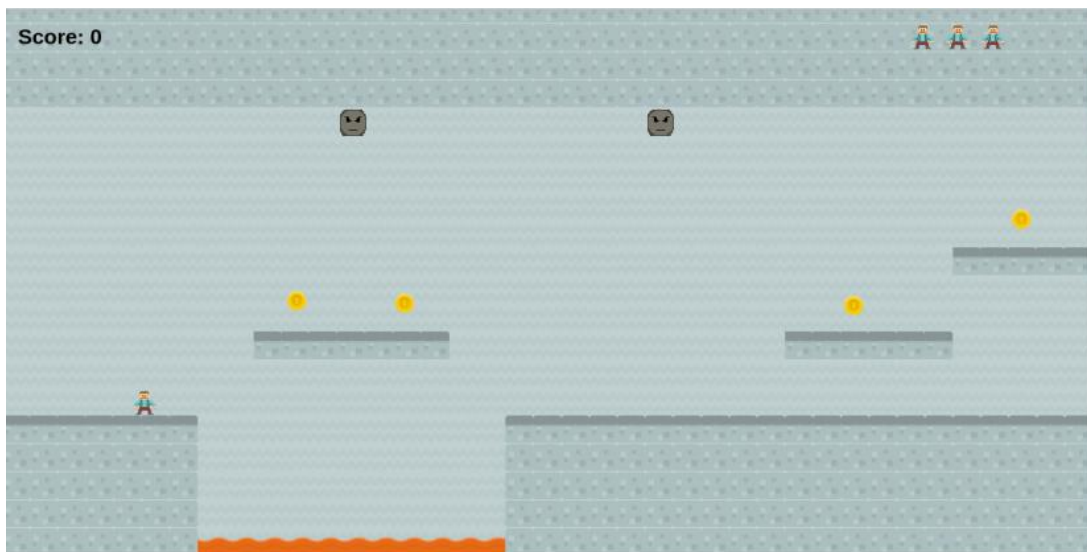

And here is our game with the StoneEnemy:

Boss

Our boss will have a simple behavior, it will keep walking forward and backward and shooting fireballs. If the player touches the boss, he will always die. For the boss behavior we will repeat a lot of things we used in other prefabs, so feel free can change the code to avoid code repetition, but I will keep it repeated here to simplify the tutorial.

```
1 var Platformer = Platformer || {};
2
3 Platformer.Boss = function (game_state, position, properties)
{
4     "use strict";
5     Platformer.Prefab.call(this, game_state, position,
properties);
6
7     this.attack_rate = +properties.attack_rate;
8     this.attack_speed = +properties.attack_speed;
9     this.walking_speed = +properties.walking_speed;
10    this.walking_distance = +properties.walking_distance;
11
12    // saving previous x to keep track of walked distance
13    this.previous_x = this.x;
14
15    this.game_state.game.physics.arcade.enable(this);
16    this.body.velocity.x = properties.direction *
this.walking_speed;
17
18    this.anchor.setTo(0.5);
19
20    // boss will be always attacking
21    this.attack_timer = this.game_state.game.time.create();
22    this.attack_timer.loop(Phaser.Timer.SECOND /
this.attack_rate, this.shoot, this);
23    this.attack_timer.start();
24 };
25
26 Platformer.Boss.prototype =
Object.create(Platformer.Prefab.prototype);
27 Platformer.Boss.prototype.constructor = Platformer.Boss;
28
29 Platformer.Boss.prototype.update = function () {
30    "use strict";
31    this.game_state.game.physics.arcade.collide(this,
this.game_state.layers.collision);
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
32
33    // if walked the maximum distance, change the velocity,
but not the scale
34    if (Math.abs(this.x - this.previous_x) >=
this.walking_distance) {
35         this.body.velocity.x *= -1;
36         this.previous_x = this.x;
37    }
38 };
39
40 Platformer.Boss.prototype.shoot = function () {
41    "use strict";
42    // works the same way player shoot, but using a different
pool group
43    var fireball, fireball_position, fireball_properties;
44    fireball =
this.game_state.groups.enemy_fireballs.getFirstDead();
45    fireball_position = new Phaser.Point(this.x, this.y);
46    if (!fireball) {
47         fireball_properties = {"texture": "fireball_image",
"group": "enemy_fireballs", "direction": "LEFT", "speed":
this.attack_speed};
48         fireball = new Platformer.Fireball(this.game_state,
fireball_position, fireball_properties);
49    } else {
50         fireball.reset(fireball_position.x,
fireball_position.y);
51         fireball.body.velocity.x = -this.attack_speed;
52    }
53 };
```

To make the Boss movement, we will do it like the Enemy movement, the only
difference is that we won't change the boss sprite scale, this way it will always be
looking in the same direction. To recall, we will save the Boss initial position before start
moving and, when it has moved a maximum distance amount we invert its velocity and
update the initial position.

The Boss attacks will be done like the player fireballs, except we will change the group to
be "enemy_fireballs" and we don't have to check for the boss direction, since it's always
the same. Notice that we use the same Fireball prefab and, since we're changing the
group, it won't collide with other enemies, and we don't have to create a new prefab for
the them.
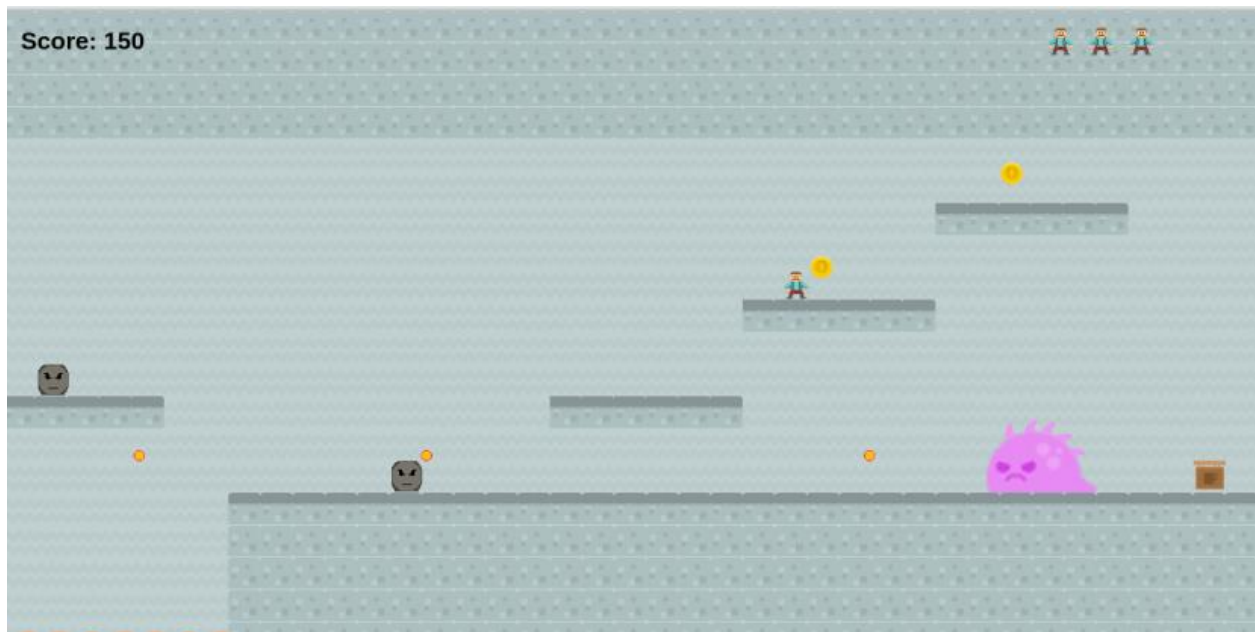
Finally, we can see our Boss in the game, and the boss level is complete:

Finishing the game

Now, you can add the new content to the other levels as you wish, and finish your game.

How to Make a Fruit Ninja Game in Phaser – Part 1
By Renan Oliveira

Fruit Ninja is a game where you have to cut fruits by swiping your cellphone screen while avoiding cutting bombs. In this tutorial, we will build a Fruit Ninja game using Phaser. To read this tutorial, it is important that you are familiar with the following concepts:

- Javascript and object-oriented concepts.
- Basic Phaser concepts, such as: states, sprites, groups and arcade physics

The following content will be covered in this tutorial:

- Reading the whole level (assets, groups and prefabs) from a JSON file
- Detect swipes in the screen and checking for collisions with game objects
- Differently handle swipes according to the cut object

Source code files

You can download the tutorial source code files here.

Assets copyright

The bomb asset used in this tutorial was made by Alucard under Creative Commons License (http://opengameart.org/content/bomb-2d), which allows commercial use with attribution.

Game states

We will use the following states to run our game:
– Boot State: loads a json file with the level information and starts the Loading State
– Loading Sate: loads all the game assets, and starts the Level State
– Level State: creates the game groups and prefabs

The code for BootState and LoadingState are shown below. BootState will load and parse the JSON file and starts LoadingState. Next, LoadingState load all the game assets, described in the JSON file. When all assets are loaded, LoadingState starts LevelState.

```
1 var FruitNinja = FruitNinja || {};
2
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Page 93

```
3 FruitNinja.BootState = function () {
4     "use strict";
5     Phaser.State.call(this);
6 };
7
8 FruitNinja.prototype = Object.create(Phaser.State.prototype);
9 FruitNinja.prototype.constructor = FruitNinja.BootState;
10
11 FruitNinja.BootState.prototype.init = function (level_file) {
12     "use strict";
13     this.level_file = level_file;
14 };
15
16 FruitNinja.BootState.prototype.preload = function () {
17     "use strict";
18     this.load.text("level1", this.level_file);
19 };
20
21 FruitNinja.BootState.prototype.create = function () {
22     "use strict";
23     var level_text, level_data;
24     level_text = this.game.cache.getText("level1");
25     level_data = JSON.parse(level_text);
26     this.game.state.start("LoadingState", true, false,
level_data);
27 };
```

```
1 var FruitNinja = FruitNinja || {};
2
3 FruitNinja.LoadingState = function () {
4     "use strict";
5     Phaser.State.call(this);
6 };
7
8 FruitNinja.prototype = Object.create(Phaser.State.prototype);
9 FruitNinja.prototype.constructor = FruitNinja.LoadingState;
10
11 FruitNinja.LoadingState.prototype.init = function
(level_data) {
12     "use strict";
13     this.level_data = level_data;
14 };
15
16 FruitNinja.LoadingState.prototype.preload = function () {
17     "use strict";
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
18    var assets, asset_loader, asset_key, asset;
19    assets = this.level_data.assets;
20    for (asset_key in assets) { // load assets according to
asset key
21        if (assets.hasOwnProperty(asset_key)) {
22            asset = assets[asset_key];
23            switch (asset.type) {
24            case "image":
25                this.load.image(asset_key, asset.source);
26                break;
27            case "spritesheet":
28                this.load.spritesheet(asset_key, asset.source,
asset.frame_width, asset.frame_height, asset.frames,
asset.margin, asset.spacing);
29                break;
30            }
31        }
32    }
33 };
34
35 FruitNinja.LoadingState.prototype.create = function () {
36    "use strict";
37    this.game.state.start("GameState", true, false,
this.level_data);
38 };
```

LevelState has a quite more things to do, as presented below. As you can see, our JSON file contains the information about assets, groups and prefabs. The assets were already loaded in the LoadingState, so now in the LevelState we only have to create the groups and instantiate the prefabs.

```
1 {
2    "assets": {
3        "fruit_image": {"type": "image", "source":
"assets/images/fruit.png"},
4        "bomb_image": {"type": "image", "source":
"assets/images/bomb.png"},
5        "background_image": {"type": "image", "source":
"assets/images/background.png"},
6        "fruits_spritesheet": {"type": "spritesheet", "source":
"assets/images/fruits.png", "frame_width": 28, "frame_height":
28}
7    },
8    "groups": [
```

```
 9          "background",
10           "spawners",
11           "fruits",
12           "bombs",
13           "cuts",
14           "hud"
15      ],
16      "prefabs": {
17          "background": {
18              "type": "background",
19              "position": {"x": 0, "y": 0},
20              "properties": {
21                  "texture": "background_image",
22                  "group": "background"
23              }
24          },
25          "fruit_spawner": {
26              "type": "fruit_spawner",
27              "position": {"x": 0, "y": 0},
28              "properties": {
29                  "texture": "",
30                  "group": "spawners",
31                  "pool": "fruits",
32                  "spawn_time": {"min": 1, "max": 3},
33                  "velocity_x": {"min": -100, "max": 100},
34                  "velocity_y": {"min": 850, "max": 1000},
35                  "frames": [20, 21, 23, 35, 38]
36              }
37          },
38          "bomb_spawner": {
39              "type": "bomb_spawner",
40              "position": {"x": 0, "y": 0},
41              "properties": {
42                  "texture": "",
43                  "group": "spawners",
44                  "pool": "bombs",
45                  "spawn_time": {"min": 1, "max": 3},
46                  "velocity_x": {"min": -100, "max": 100},
47                  "velocity_y": {"min": 850, "max": 1000}
48              }
49          }
50      }
51 }
```

LevelState code is shown below. To create level groups, we just have to go trough all groups in the JSON file. To create level prefabs we need the prefab name, its type, position, texture, group and properties. Then, we go through each one of the prefabs described in our JSON file and call the "create_prefab" method, which will instantiate the correct prefab. To do that, we'll keep an object that maps each prefab type to its correct constructor, as you can see in LevelState constructor. This way, to instantiate a new prefab we check if the type is present in the "prefab_classes" objects, and if so, we call the correct constructor. Notice that this only works because all prefabs have the same constructor.

```
1 var FruitNinja = FruitNinja || {};
2
3 FruitNinja.LevelState = function () {
4     "use strict";
5     Phaser.State.call(this);
6
7     this.prefab_classes = {
8         "fruit_spawner": FruitNinja.FruitSpawner.prototype.constructor,
9         "bomb_spawner": FruitNinja.BombSpawner.prototype.constructor,
10        "background": FruitNinja.Prefab.prototype.constructor
11    };
12 };
13
14 FruitNinja.LevelState.prototype = Object.create(Phaser.State.prototype);
15 FruitNinja.LevelState.prototype.constructor = FruitNinja.LevelState;
16
17 FruitNinja.LevelState.prototype.init = function (level_data) {
18     "use strict";
19     this.level_data = level_data;
20
21     this.scale.scaleMode = Phaser.ScaleManager.SHOW_ALL;
22     this.scale.pageAlignHorizontally = true;
23     this.scale.pageAlignVertically = true;
24
25     // start physics system
26     this.game.physics.startSystem(Phaser.Physics.ARCADE);
27     this.game.physics.arcade.gravity.y = 1000;
28
29     this.MINIMUM_SWIPE_LENGTH = 50;
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
30
31     this.score = 0;
32 };
33
34 FruitNinja.LevelState.prototype.create = function () {
35     "use strict";
36     var group_name, prefab_name;
37
38     // create groups
39     this.groups = {};
40     this.level_data.groups.forEach(function (group_name) {
41         this.groups[group_name] = this.game.add.group();
42     }, this);
43
44     // create prefabs
45     this.prefabs = {};
46     for (prefab_name in this.level_data.prefabs) {
47         if
(this.level_data.prefabs.hasOwnProperty(prefab_name)) {
48             // create prefab
49             this.create_prefab(prefab_name,
this.level_data.prefabs[prefab_name]);
50         }
51     }
52
53     // add events to check for swipe
54     this.game.input.onDown.add(this.start_swipe, this);
55     this.game.input.onUp.add(this.end_swipe, this);
56
57     this.init_hud();
58 };
59
60 FruitNinja.LevelState.prototype.create_prefab = function
(prefab_name, prefab_data) {
61     "use strict";
62     var prefab;
63     // create object according to its type
64     if (this.prefab_classes.hasOwnProperty(prefab_data.type))
{
65         prefab = new
this.prefab_classes[prefab_data.type](this, prefab_name,
prefab_data.position, prefab_data.properties);
66     }
67 };
68
```

```
69 FruitNinja.LevelState.prototype.game_over = function () {
70    "use strict";
71    this.game.state.restart(true, false, this.level_data);
72 };
```

Prefabs

In Phaser, prefabs are objects that extend Phaser.Sprite, allowing us to add custom properties and methods. In this tutorial, all our game objects will be implemented using prefabs, so we will write a generic Prefab, which all our prefabs will extend. Notice that our Prefab will have the Game State, in case we need to use it, and allows us to choose a group and frame.

```
1 var FruitNinja = FruitNinja || {};
2
3 FruitNinja.Prefab = function (game_state, name, position,
properties) {
4    "use strict";
5    Phaser.Sprite.call(this, game_state.game, position.x,
position.y, properties.texture);
6
7    this.game_state = game_state;
8
9    this.name = name;
10
11    this.game_state.groups[properties.group].add(this);
12    this.frame = properties.frame;
13
14    this.game_state.prefabs[name] = this;
15 };
16
17 FruitNinja.Prefab.prototype =
Object.create(Phaser.Sprite.prototype);
18 FruitNinja.Prefab.prototype.constructor = FruitNinja.Prefab;
```

Detecting swipes

We'll use the following strategy to detect swipes:

1   When the user touches the screen, we save the touch position as a starting point
2   When the user releases the screen, we save the touch position as a end point
3   If the distance between both starting and end points is greater than a minimum distance, we detect it as a swipe, and create a line between these two points

The code below shows how this works. First, we add onDown and onUp events to the game input. The onDown event will only save the starting point. On the other hand the onUp will save the end point and check if the screen was swiped. To do that we calculate the difference between both starting and end point using Phaser.Point.distance (Phaser already provides some geometric shapes and operations that you can check in the documentation: http://phaser.io/docs#geometry). If we detect a swipe, we create a Phaser.Line with the starting and end point and save the swipe for later collision checking.

```
1 FruitNinja.LevelState.prototype.start_swipe = function
(pointer) {
2     "use strict";
3     this.start_swipe_point = new Phaser.Point(pointer.x,
pointer.y);
4 };
5
6 FruitNinja.LevelState.prototype.end_swipe = function (pointer)
{
7     "use strict";
8     var swipe_length, cut_style, cut;
9     this.end_swipe_point = new Phaser.Point(pointer.x,
pointer.y);
10     swipe_length = Phaser.Point.distance(this.end_swipe_point,
this.start_swipe_point);
11     // if the swipe length is greater than the minimum, a
swipe is detected
12     if (swipe_length >= this.MINIMUM_SWIPE_LENGTH) {
13         // create a new line as the swipe and check for
collisions
14         cut_style = {line_width: 5, color: 0xE82C0C, alpha: 1}
15         cut = new FruitNinja.Cut(this, "cut", {x: 0, y: 0},
{group: "cuts", start: this.start_swipe_point, end:
this.end_swipe_point, duration: 0.3, style: cut_style});
16         this.swipe = new Phaser.Line(this.start_swipe_point.x,
this.start_swipe_point.y, this.end_swipe_point.x,
this.end_swipe_point.y);
17     }
18 };
```

We will also create a Cut prefab to show it in the screen. The Cut prefab will extend Phaser.Graphics instead of Phaser.Sprite. This class allows to draw lines in the screen,

which we will use to draw the cut. After drawing the line in the prefab constructor, we
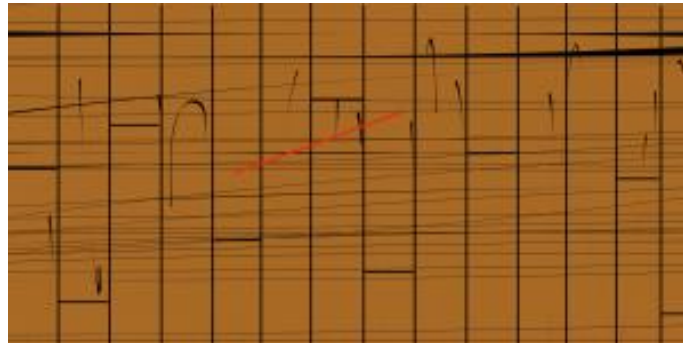
initialize a timer to kill it after some time, making it disappear.

By now, you can already try making some swipes and checking if the cuts are being properly draw.

Fruits and bombs

We will need Fruit and Bomb prefabs to be cut in our game. However, as you can already imagine, both will have a lot of things in common. So, to avoid code repetition, we will create a generic Cuttable prefab which both Fruit and Bomb will extend and will handle all the common things.

The Cuttable prefab code is shown below. It will have a starting velocity, which will be applied in the constructor. We expect the fruits and bombs to jump on the screen and then start falling, until leaving it. So, we will set both checkWorldBounds and outOfBoundsKill properties to true. Those are Phaser properties that kill a sprite when it leaves the screen, as we want it to work.

```
1 var FruitNinja = FruitNinja || {};
2
3 FruitNinja.Cuttable = function (game_state, name, position,
properties) {
4    "use strict";
5    FruitNinja.Prefab.call(this, game_state, name, position,
properties);
6
7    this.anchor.setTo(0.5);
8    this.scale.setTo(5);
9
10    this.game_state.game.physics.arcade.enable(this);
```

```
11
12    // initiate velocity
13    this.velocity = properties.velocity;
14    this.body.velocity.y = -this.velocity.y;
15    this.body.velocity.x = this.velocity.x;
16
17    // kill prefab if it leaves screen
18    this.checkWorldBounds = true;
19    this.outOfBoundsKill = true;
20 };
21
22 FruitNinja.Cuttable.prototype =
Object.create(FruitNinja.Prefab.prototype);
23 FruitNinja.Cuttable.prototype.constructor =
FruitNinja.Cuttable;
24
25 FruitNinja.Cuttable.prototype.reset = function (position_x,
position_y, velocity) {
26    "use strict";
27    Phaser.Sprite.prototype.reset.call(this, position_x,
position_y);
28    // initiate velocity
29    this.body.velocity.y = -velocity.y;
30    this.body.velocity.x = velocity.x;
31 };
```

Now we can create the Fruit and Bomb prefabs as follows. Notice that we only have to add a "cut" method in each one that will handle cuts. The Fruit prefab will increment a score variable, while the Bomb will end the game. This "cut" method will be called later, when we add the cutting logic. For the Fruit prefab, we will randomly select a frame from the fruits spritesheet, so we can create different kinds of fruits.

```
1 var FruitNinja = FruitNinja || {};
2
3 FruitNinja.Fruit = function (game_state, name, position,
properties) {
4    "use strict";
5    var frame_index;
6    FruitNinja.Cuttable.call(this, game_state, name, position,
properties);
7
8    this.frames = properties.frames;
9
```

```
10      frame_index = this.game_state.rnd.between(0,
this.frames.length - 1);
11      this.frame = this.frames[frame_index];
12
13      this.body.setSize(20, 20);
14 };
15
16 FruitNinja.Fruit.prototype =
Object.create(FruitNinja.Cuttable.prototype);
17 FruitNinja.Fruit.prototype.constructor = FruitNinja.Fruit;
18
19 FruitNinja.Fruit.prototype.reset = function (position_x,
position_y, velocity) {
20      "use strict";
21      var frame_index;
22      FruitNinja.Cuttable.prototype.reset.call(this, position_x,
position_y, velocity);
23      frame_index = this.game_state.rnd.between(0,
this.frames.length - 1);
24      this.frame = this.frames[frame_index];
25 };
26
27 FruitNinja.Fruit.prototype.cut = function () {
28      "use strict";
29      // if a fruit is cut, increment score
30      this.game_state.score += 1;
31      this.kill();
32 };


1 var FruitNinja = FruitNinja || {};
2
3 FruitNinja.Bomb = function (game_state, name, position,
properties) {
4      "use strict";
5      FruitNinja.Cuttable.call(this, game_state, name, position,
properties);
6
7      this.body.setSize(20, 20);
8 };
9
10 FruitNinja.Bomb.prototype =
Object.create(FruitNinja.Cuttable.prototype);
11 FruitNinja.Bomb.prototype.constructor = FruitNinja.Bomb;
12
13 FruitNinja.Bomb.prototype.cut = function () {
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
14    "use strict";
15    // if a bomb is cut, it's game over
16    this.game_state.game_over();
17    this.kill();
18 };
```

Spawning fruits and bombs

To spawn fruits and bombs, we'll do something similar to the Fruit and Bomb prefabs. Since we need one spawner for each cuttable object, we will create a generic Spawner and two more prefabs that will extend it: FruitSpawner and BombSpawner. The Spawner code is shown below. First, a timer is created and scheduled to spawn the first prefab. The time to spawn is randomly selected between an interval. To generate a random value between two integers, we use Phaser random data generator (for more information, check Phaser documentation). After the spawn time, the timer calls a "spawn" method, that will create the prefab.

```
1 var FruitNinja = FruitNinja || {};
2
3 FruitNinja.Spawner = function (game_state, name, position,
properties) {
4    "use strict";
5    FruitNinja.Prefab.call(this, game_state, name, position,
properties);
6
7    this.pool = this.game_state.groups[properties.pool];
8
9    this.spawn_time = properties.spawn_time;
10
11    this.velocity_x = properties.velocity_x;
12    this.velocity_y = properties.velocity_y;
13
14    this.spawn_timer = this.game_state.time.create();
15    this.schedule_spawn();
16 };
17
18 FruitNinja.Spawner.prototype =
Object.create(FruitNinja.Prefab.prototype);
19 FruitNinja.Spawner.prototype.constructor =
FruitNinja.Spawner;
20
21 FruitNinja.Spawner.prototype.schedule_spawn = function () {
22    "use strict";
23    var time;
```

```
24     // add a new spawn event with random time between a range
25     time = this.game_state.rnd.between(this.spawn_time.min,
this.spawn_time.max);
26     this.spawn_timer.add(Phaser.Timer.SECOND * time,
this.spawn, this);
27     this.spawn_timer.start();
28 };
29
30 FruitNinja.Spawner.prototype.spawn = function () {
31     "use strict";
32     var object_name, object_position, object, object_velocity;
33     // get new random position and velocity
34     object_position = new
Phaser.Point(this.game_state.rnd.between(0.2 *
this.game_state.game.world.width, 0.8 *
this.game_state.game.world.width),
this.game_state.game.world.height);
35     object_velocity = this.object_velocity();
36     // get first dead object from the pool
37     object = this.pool.getFirstDead();
38     if (!object) {
39         // if there is no dead object, create a new one
40         object_name = "object_" + this.pool.countLiving();
41         object = this.create_object(object_name,
object_position, object_velocity);
42     } else {
43         // if there is a dead object, reset it to the new
position and velocity
44         object.reset(object_position.x, object_position.y,
object_velocity);
45     }
46
47     // schedule next spawn
48     this.schedule_spawn();
49 };
50
51 FruitNinja.Spawner.prototype.object_velocity = function () {
52     "use strict";
53     var velocity_x, velocity_y;
54     // generate random velocity inside a range
55     velocity_x =
this.game_state.rnd.between(this.velocity_x.min,
this.velocity_x.max);
```

```
56    velocity_y =
this.game_state.rnd.between(this.velocity_y.min,
this.velocity_y.max);
57    return new Phaser.Point(velocity_x, velocity_y);
58 };
```

To create prefabs, we will use a pool of objects, as we used in another tutorial. A pool of objects is a group that we keep to reuse old objects instead of create new ones. So, to spawn a new prefab, first we query the pool group (which is a Phaser group) for the first dead prefab in it. If there is a dead prefab, we just reset it to the new position and velocity. Otherwise, we create a new prefab using the method "create_object". This method will be added in each Spawner we will create, to return the correct Prefab. After spawning the new prefab, we schedule the next spawn. Also, notice that the position and velocity are randomly defined inside a range, using Phaser random data generator.

The FruitSpawner and BombSpawner prefabs are shown below. We only have to add the "create_object" method, which will return a Fruit prefab for the first spawner and a Bomb in the second one. For the FruitSpawner, we have to create the Fruit with the frames we want to use from the spritesheet. In this tutorial I used frames 20, 21, 23, 35, 38. Feel free to use the ones you find best.

```
1 var FruitNinja = FruitNinja || {};
2
3 FruitNinja.FruitSpawner = function (game_state, name,
position, properties) {
4    "use strict";
5    FruitNinja.Spawner.call(this, game_state, name, position,
properties);
6
7    this.frames = properties.frames;
8 };
9
10 FruitNinja.FruitSpawner.prototype =
Object.create(FruitNinja.Spawner.prototype);
11 FruitNinja.FruitSpawner.prototype.constructor =
FruitNinja.FruitSpawner;
12
13 FruitNinja.FruitSpawner.prototype.create_object = function
(name, position, velocity) {
14    "use strict";
15    // return new fruit with random frame
```
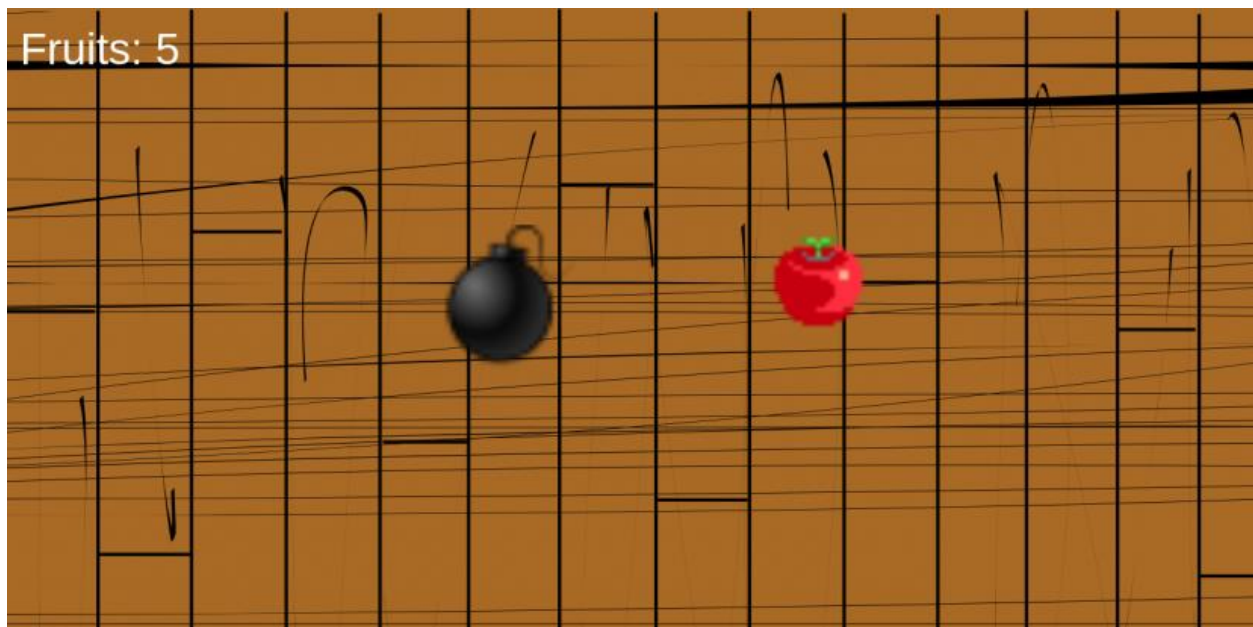
Fruits: 5

```
16     return new FruitNinja.Fruit(this.game_state, name,
position, {texture: "fruits_spritesheet", group: "fruits",
frames: this.frames, velocity: velocity});
17 };

1 var FruitNinja = FruitNinja || {};
2
3 FruitNinja.BombSpawner = function (game_state, name, position,
properties) {
4     "use strict";
5     FruitNinja.Spawner.call(this, game_state, name, position,
properties);
6 };
7
8 FruitNinja.BombSpawner.prototype =
Object.create(FruitNinja.Spawner.prototype);
9 FruitNinja.BombSpawner.prototype.constructor =
FruitNinja.BombSpawner;
10
11 FruitNinja.BombSpawner.prototype.create_object = function
(name, position, velocity) {
12     "use strict";
13     // return new bomb
14     return new FruitNinja.Bomb(this.game_state, name,
position, {texture: "bomb_image", group: "bombs", velocity:
velocity});
15 };
```
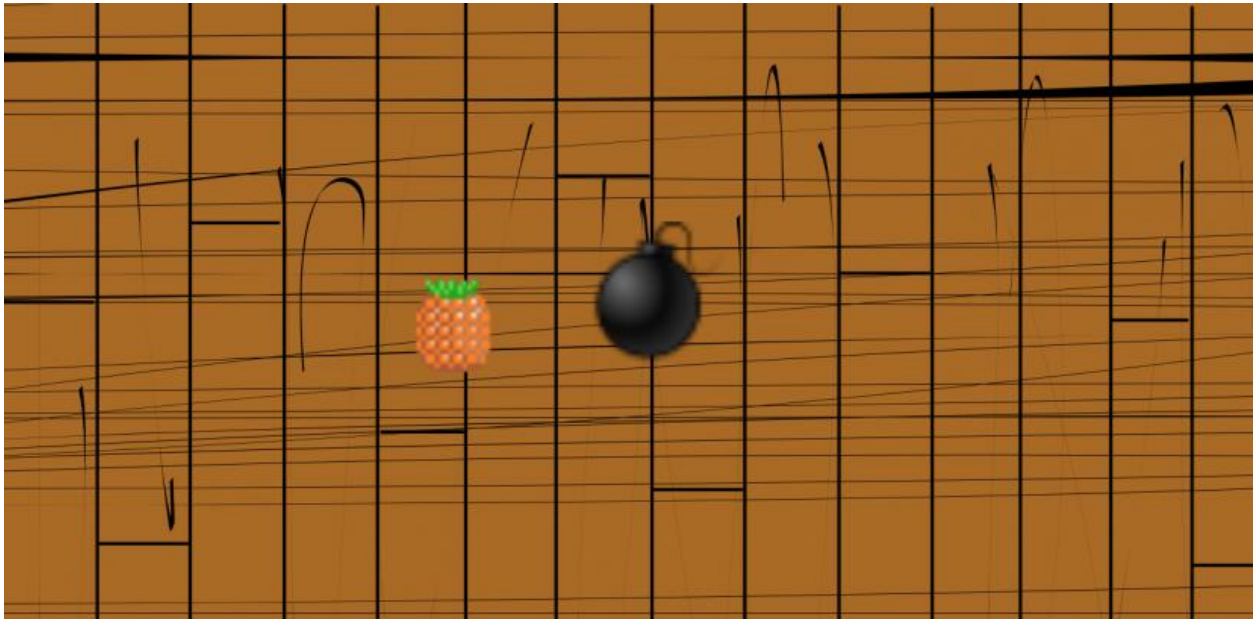
You can already run your game with the spawner and see if it is working correctly.

Cutting fruits and bombs

We still have to make it possible to cut fruits and bombs. Back in our LevelState, we're already detecting swipes. Now, after we detect a swipe we will check if it's colliding with any cuttable object and, if so, cut it. To do that, we go through all living sprites in the fruits and bombs groups calling the "check_collision" method.

The "check_collision" method will check for intersections between the swipe line (which we already have) and each one of the target object boundaries. To do that, we start creating a Phaser.Rectangle with the object body coordinates and size. Then, we build a Phaser.Line for each one of this rectangle edges and check for intersection with the swipe line. To check for intersection, we will use another one of Phaser geometry operations, which check for intersection between two lines.

```
1 FruitNinja.LevelState.prototype.check_collision = function
(object) {
2    "use strict";
3    var object_rectangle, line1, line2, line3, line4,
intersection;
4    // create a rectangle for the object body
5    object_rectangle = new Phaser.Rectangle(object.body.x,
object.body.y, object.body.width, object.body.height);
```

```
6      // check for intersections with each rectangle edge
7      line1 = new Phaser.Line(object_rectangle.left,
object_rectangle.bottom, object_rectangle.left,
object_rectangle.top);
8      line2 = new Phaser.Line(object_rectangle.left,
object_rectangle.top, object_rectangle.right,
object_rectangle.top);
9      line3 = new Phaser.Line(object_rectangle.right,
object_rectangle.top, object_rectangle.right,
object_rectangle.bottom);
10     line4 = new Phaser.Line(object_rectangle.right,
object_rectangle.bottom, object_rectangle.left,
object_rectangle.bottom);
11     intersection = this.swipe.intersects(line1) ||
this.swipe.intersects(line2) || this.swipe.intersects(line3) ||
this.swipe.intersects(line4);
12     if (intersection) {
13         // if an intersection is found, cut the object
14         object.cut();
15     }
16 };
```

You can already run your game and see if you can cut the fruits and bombs.

Keeping track of cut fruits

To finish, we have to show the score during the game. We're already saving the cut fruits, so we just have to add a HUD item to show it. To do that, we will add an "init_hud" method in LevelState. In this method we will create a Score prefab in a fixed position and add it to the HUD group.

```
1 var FruitNinja = FruitNinja || {};
2
3 FruitNinja.Score = function (game_state, name, position,
properties) {
4     "use strict";
5     Phaser.Text.call(this, game_state.game, position.x,
position.y, properties.text, properties.style);
6
7     this.game_state = game_state;
8
9     this.name = name;
10
11    this.game_state.groups[properties.group].add(this);
```

```
12
13     this.game_state.prefabs[name] = this;
14  };
15
16  FruitNinja.Score.prototype =
Object.create(Phaser.Text.prototype);
17  FruitNinja.Score.prototype.constructor = FruitNinja.Score;
18
19  FruitNinja.Score.prototype.update = function () {
20      "use strict";
21      // update the text to show the number of cutted fruits
22      this.text = "Fruits: " + this.game_state.score;
23  };
```

Now, you can play the game while it shows how many fruits you have already cut.Finishing the game

With that, we finish our game. In the following tutorials we'll add more content to our game, making it more fun to play.

How to Make a Fruit Ninja Game in Phaser – Part 2
By Renan Oliveira

In Part 1 of the Fruit Ninja tutorial we started creating a Fruit Ninja game. In our game we already have fruits and bombs, which we can cut. If you cut a fruit, you increase your score, which is shown in the screen. Otherwise, if you cut a bomb, you lose. In this tutorial we will the following content to our game, making it more fun to play:

- Player lives, so the player has a number of bombs he can cut before losing
- A game over screen, which will show the current score and the highest score so far (the highest score will be saved)
- Particle effects when the player cut something, making it visibly more attractive
- A special fruit, that stops when you cut it for the first time, allowing you to make more cuts in a row

To read this tutorial, it is important that you are familiar with the following concepts:

- Javascript and object-oriented concepts.
- Basic Phaser concepts, such as: states, sprites, groups and arcade physics

Assets copyright

The bomb asset used in this tutorial was made by Alucard under Creative Commons License (http://opengameart.org/content/bomb-2d), which allows commercial use with attribution.

Source code files

You can download the tutorial source code files here.

Game states

We will use the following states to run our game:

- Boot State: loads a json file with the level information and starts the Loading State
- Loading Sate: loads all the game assets, and starts the Level State
- Level State: creates the game groups and prefabs

The code for BootState and Loading is exactly the same from the last tutorial, so I will omit them.

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

The LevelState however, has some changes that were necessary to allow the new content. So, I will show those changes when they are necessary.

Player lives

To make the game work with player lives, we will first create a Lives prefab as shown below. The Lives prefab will have the lives asset texture, but it will be invisible. This way, we can use it to create new sprites with the same texture. This is done in the constructor, where we iterate through each life and create a new Phaser sprite. The position of this new sprite is given by the Lives prefab position plus a spacing, while the texture is the same as Lives prefab. All those sprites are stored in an array, so we can manipulate them later in the "die" method.

```
1 var FruitNinja = FruitNinja || {};
2
3 FruitNinja.Lives = function (game_state, name, position,
properties) {
4     "use strict";
5     var live_index, life;
6     FruitNinja.Prefab.call(this, game_state, name, position,
properties);
7
8     this.visible = false;
9
10    this.lives = properties.lives;
11    this.lives_sprites = [];
12    // create a sprite for each life
13    for (live_index = 0; live_index < this.lives; live_index
+= 1) {
14        life = new Phaser.Sprite(this.game_state.game,
position.x + (live_index * properties.lives_spacing),
position.y, this.texture);
15        this.lives_sprites.push(life);
16        this.game_state.groups.hud.add(life);
17    }
18 };
19
20 FruitNinja.Lives.prototype =
Object.create(FruitNinja.Prefab.prototype);
21 FruitNinja.Lives.prototype.constructor = FruitNinja.Lives;
22
23 FruitNinja.Lives.prototype.die = function () {
```

```
24    "use strict";
25    var life;
26    this.lives -= 1;
27    // kill the last life
28    life = this.lives_sprites.pop();
29    life.kill();
30    // if there are no more lives, it's game over
31    if (this.lives === 0) {
32        this.game_state.game_over();
33    }
34 };
```

The "die" method will be called when the player cuts a bomb. It will decrease the number of lives, kill the last life in the lives array and check the player remaining number of lives. If there are no more lives, it's game over.

Finally, we have to change the "cut" method in the Bomb prefab to call this new "die" method (remember that it was calling "game_over" before).

```
1 FruitNinja.Bomb.prototype.cut = function () {
2    "use strict";
3    FruitNinja.Cuttable.prototype.cut.call(this);
4    // if a bomb is cut, the player lose a life
5    this.game_state.prefabs.lives.die();
6    this.kill();
7 };
```


You can already try playing with the player lives and see if it's working.

Game over screen

Now that we already have player lives, it would be interesting to show a game over screen when the player actually loses, instead of just restarting the game. To make it simple, our game over screen will be only a panel which will be shown over the game, with a game over message, the current score and the highest score so far. We will create a GameOverPanel prefab for that, as shown below. First, we will lower the alpha of our panel, so the player can see the game behind it. Next, we will start a tween animation to show it. The idea is that the panel will come from the bottom of the screen and, when it arrives the top, it will show the game over message, calling the "show_game_over" method.

```
1 var FruitNinja = FruitNinja || {};
2
3 FruitNinja.GameOverPanel = function (game_state, name,
position, properties) {
4     "use strict";
5     var movement_animation;
6     FruitNinja.Prefab.call(this, game_state, name, position,
properties);
7
8     this.text_style = properties.text_style;
9
10    this.alpha = 0.5;
11    // create a tween animation to show the game over panel
12    movement_animation = this.game_state.game.add.tween(this);
13    movement_animation.to({y: 0}, properties.animation_time);
14    movement_animation.onComplete.add(this.show_game_over,
this);
15    movement_animation.start();
16 };
17
18 FruitNinja.GameOverPanel.prototype =
Object.create(FruitNinja.Prefab.prototype);
19 FruitNinja.GameOverPanel.prototype.constructor =
FruitNinja.GameOverPanel;
20
21 FruitNinja.GameOverPanel.prototype.show_game_over = function
() {
22    "use strict";
23    var game_over_text, current_score_text,
highest_score_text;
24    // add game over text
25    game_over_text =
this.game_state.game.add.text(this.game_state.game.world.width /
```

```
2, this.game_state.game.world.height * 0.4, "Game Over",
this.text_style.game_over);
26    game_over_text.anchor.setTo(0.5);
27      this.game_state.groups.hud.add(game_over_text);
28
29    // add current score text
30    current_score_text =
this.game_state.game.add.text(this.game_state.game.world.width /
2, this.game_state.game.world.height * 0.5, "Score: " +
this.game_state.score, this.text_style.current_score);
31    current_score_text.anchor.setTo(0.5);
32      this.game_state.groups.hud.add(current_score_text);
33
34    // add highest score text
35    highest_score_text =
this.game_state.game.add.text(this.game_state.game.world.width /
2, this.game_state.game.world.height * 0.6, "Highest score: " +
localStorage.highest_score, this.text_style.highest_score);
36    highest_score_text.anchor.setTo(0.5);
37      this.game_state.groups.hud.add(highest_score_text);
38
39    // add event to restart level
40    this.inputEnabled = true;
41      this.events.onInputDown.add(this.game_state.restart_level,
this.game_state);
42 };
```

The "show_game_over" method will add three Phaser texts on the screen for the
following information: 1) game over message; 2) current score; 3) highest score. Notice
that each text has its own style, which were all passed in the constructor through the
properties parameter. Also, the position is calculated using the game world width and
height, so it should work with different screen sizes. Finally, "show_game_over" will add
an input event to restart the game when the player touches the screen.

We still have to create this GameOverPanel in the LevelState, and we will do that in the
"game_over_method". First, we will update the highest score. To save data from your
game, you can use the browser localStorage. Any data saved this way will be kept even
when the browser is reloaded (for more information:
http://www.w3schools.com/html/html5_webstorage.asp). So, first we will check if our
score is higher than "localStorage.highest_score" (or if there is no highest score yet) and
if so, update it.

```
1 FruitNinja.LevelState.prototype.game_over = function () {
```

```
2      "use strict";
3      var game_over_panel, game_over_position, game_over_bitmap,
panel_text_style;
4      // if current score is higher than highest score, update it
5      if (!localStorage.highest_score || this.score >
localStorage.highest_score) {
6          localStorage.highest_score = this.score;
7      }
8
9      // create a bitmap do show the game over panel
10     game_over_position = new Phaser.Point(0,
this.game.world.height);
11     game_over_bitmap =
this.add.bitmapData(this.game.world.width,
this.game.world.height);
12     game_over_bitmap.ctx.fillStyle = "#000";
13     game_over_bitmap.ctx.fillRect(0, 0, this.game.world.width,
this.game.world.height);
14     panel_text_style = {game_over: {font: "32px Arial", fill:
"#FFF"},
15                         current_score: {font: "20px Arial",
fill: "#FFF"},
16                         highest_score: {font: "18px Arial",
fill: "#FFF"}};
17     // create the game over panel
18     game_over_panel = new FruitNinja.GameOverPanel(this,
"game_over_panel", game_over_position, {texture:
game_over_bitmap, group: "hud", text_style: panel_text_style,
animation_time: 500});
19     this.groups.hud.add(game_over_panel);
20 };
```

Next, we have to create GameOverPanel. The texture of this sprite will be a Phaser bitmap, which is a Phaser object with an HTML Canvas, so you can use it like a canvas (for more information, check the documentation). Our bitmap will be from the size of the world, and we change its color as we would do with a HTML Canvas. Finally, we set the text style for each one of the GameOverPanel texts, and create it.

You can already try playing with the game over screen to see if it works. Try scoring higher than before and see if highest score is properly updated, even if you reload the

browser.

Adding a particle effect

The next thing we are going to add to our game is purely visual, but you will see it adds a lot to the game. Currently, when we cut a fruit or a bomb, it simply disappears, which is kinda weird from the player's point of view. Now we are going to add a particle effect, which will work like a visual feedback to the player, so he automatically understands that he actually cut something.

This will be done changing the "cut" method in Cuttable. We start creating a Phaser emitter in the prefab position and setting the particles asset (you will notice I added a new asset called "particle_image" in the level JSON file). Next we have to set the particles minimum and maximum velocity in both directions, as well as the gravity acting on them. Finally, we start the emitter, telling it how long it should emit and how many particles (for more information about the Emitter class, check Phaser documentation).

```
1  FruitNinja.Cuttable.prototype.cut = function () {
2      "use strict";
3      var emitter;
4      // create emitter in prefab position
5      emitter = this.game_state.game.add.emitter(this.x, this.y);
6      emitter.makeParticles("particle_image");
7      // set particles speed
8      emitter.minParticleSpeed.setTo(-200, -200);
9      emitter.maxParticleSpeed.setTo(200, 200);
10      emitter.gravity = 0;
11      // start emitter
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Page 117

```
12     emitter.start(true, 700, null, 1000);
13 };
```

All the values I used as parameters for the emitter methods (such as velocity, number of particles) were decided experimentally by trial and error. This is usually done when we have to decide parameters in a game, so you should try some values and see which ones fits your game best.

By now, you can already try cutting some fruits and bombs and see the particles visual effect.

Special fruit

The last thing we will add is a special fruit. When the player cut this fruit, it will stop for some time, so the player can keep cutting it and increase the score. To do that, we will create a SpecialFruit prefab as shown below. In the constructor we will create a kill timer, which won't be started until the fruit is cut. Then, in the "cut" method, we increase the player score and check if the kill timer is not running. If that's the case, this was the first cut, so we should stop the fruit (which is done by setting body.allowGravity to false and making the velocity in both directions 0) and start the kill timer. The next time the fruit is cut, the kill timer will already be running, so it will only increase the player score.

```
1 var FruitNinja = FruitNinja || {};
2
3 FruitNinja.SpecialFruit = function (game_state, name,
position, properties) {
4     "use strict";
5     var frame_index;
6     FruitNinja.Cuttable.call(this, game_state, name, position,
properties);
7
8     this.body.setSize(20, 20);
9
10     // create a timer with autoDestroy = false, so it won't be
killed
11     this.kill_timer = this.game_state.game.time.create(false);
12 };
13
14 FruitNinja.SpecialFruit.prototype =
Object.create(FruitNinja.Cuttable.prototype);
15 FruitNinja.SpecialFruit.prototype.constructor =
FruitNinja.SpecialFruit;
16
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Page 118

```
17 FruitNinja.SpecialFruit.prototype.kill = function () {
18     "use strict";
19     Phaser.Sprite.prototype.kill.call(this);
20     // prepare the fruit so it can be reused
21     this.body.allowGravity = true;
22     this.kill_timer.stop();
23 };
24
25 FruitNinja.SpecialFruit.prototype.cut = function () {
26     "use strict";
27     FruitNinja.Cuttable.prototype.cut.call(this);
28     // if a fruit is cut, increment score
29     this.game_state.score += 1;
30     // if it's the first cut, stops the fruit and start the
timer to kill it
31     if (!this.kill_timer.running) {
32         this.body.allowGravity = false;
33         this.body.velocity.y = 0;
34         this.body.velocity.x = 0;
35         this.kill_timer.add(Phaser.Timer.SECOND * 3,
this.kill, this);
36         this.kill_timer.start();
37     }
38 };
```

There is important things to notice here, though. When the fruit is killed, we expect the object to be reused for the next fruit, because we're keeping it in a pool of objects and we don't want to create a new object. To allow this, we have to set body.allowGravity back to true and stop the timer in the "kill" method, so it will still be working when it is reused. Also, when we created the timer in the constructor, we set the parameter autoDestroy false, otherwise the timer would be destroyed when there were no more events.

Now that we have our special fruit prefab, we have to create a spawner for it. For that, we will create a SpecialFruitSpawner which will overwrite the "create_object" method to return a SpecialFruit. For the asset I just used a different fruit from our fruits spritesheet. Notice how it was easy to add a new kind of fruit since we have our Cuttable and Spawner generic prefabs. If we wanted to add another kind of fruit (or bomb), we could follow the same process, overwriting only the methods that are different from the original Cuttable and Spawner classes. This makes it much easier to add content to our game.

```
1 var FruitNinja = FruitNinja || {};
2
```

```
3 FruitNinja.SpecialFruitSpawner = function (game_state, name,
position, properties) {
4     "use strict";
5     FruitNinja.Spawner.call(this, game_state, name, position,
properties);
6
7     this.frames = properties.frames;
8 };
9
10 FruitNinja.SpecialFruitSpawner.prototype =
Object.create(FruitNinja.Spawner.prototype);
11 FruitNinja.SpecialFruitSpawner.prototype.constructor =
FruitNinja.SpecialFruitSpawner;
12
13 FruitNinja.SpecialFruitSpawner.prototype.create_object =
function (name, position, velocity) {
14     "use strict";
15     // return new fruit with random frame
16     return new FruitNinja.SpecialFruit(this.game_state, name,
position, {texture: "fruits_spritesheet", group:
"special_fruits", frame: 15, velocity: velocity});
17 };
```

Finally, our game is done and you can try cutting some special fruits to beat your highest

score.

Finishing the game

With that, we finish our game. In the following tutorial we will add a title screen and a new game mode!

How to Make a Fruit Ninja Game in Phaser – Part 3
By Renan Oliveira

In my last two tutorials we created a Fruit Ninja game and added some content to it. In this last tutorial, we will add a new game mode. This will imply in adding a title screen, refactoring part of our code and adding some extra content. I will cover the following content in this tutorial:

- Adding a custom font for all the texts being shown
- Adding a new game mode called Time Attack. In this mode, the player must cut as many fruits as possible before the time runs out
- Refactoring the code for our states, to reuse between different states
- Adding a title screen, where the player can choose between the two game modes
- Adding two cuttables prefabs for the Time Attack mode. The first one is a clock, which increases the remaining time when cut, and the second one is a time bomb, which reduces the remaining time when cut

To read this tutorial, it is important that you are familiar with the following concepts:

- Javascript and object-oriented concepts.
- Basic Phaser concepts, such as: states, sprites, groups and arcade physics

Assets copyright
The bomb asset used in this tutorial was made by Alucard under Creative Commons License (http://opengameart.org/content/bomb-2d), which allows commercial use with attribution.

Source code files

You can download the tutorial source code files *here*.

Game states

We will use the following states to run our game:

- Boot State: loads a json file with the level information and starts the Loading State
- Loading Sate: loads all the game assets, and starts the next State
- Title State: shows the title screen, allowing the player to choose between the two game modes: Classic and Time Attack

- Classic State: runs the game mode we were working in the last two tutorials, in which the player cuts fruits until runs out of lives
- Time Attack State: new game mode, where the player cuts fruits until runs out of time

There are a lot of changes in the states codes, since we're going to refactor the code to allow multiple game modes. So, to simplify, I will show the changes as they're necessary.

Refactoring the code for adding multiple game modes

Since we're going to have multiple states, we have to identify common code between them and put it in another state which will be extended. In order to do that, we'll create the following parent states to be extended:

- JSONLevelState: loads a JSON file as we were using in the last tutorials, with assets, groups and prefabs
- LevelState: represents a generic level. Is responsible for detecting swipes, checking for collisions and showing the score and game over screen

The states hierarchy is shown in the figure below. Notice that TitleState extends JSONLevelState, since we will start it from a JSON file, but it does not extend LevelState, since it's not a game state. On the other hand, both ClassicState and

TimeAttackState will extend LevelState, overwriting necessary methods and adding new
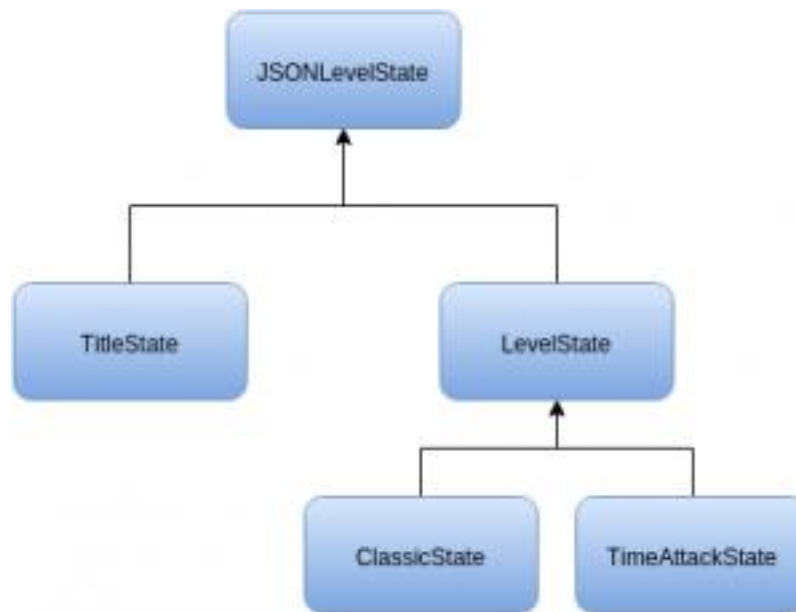
ones.

The code for JSONLevelState and LevelState are shown below. JSONLevelState has the "create_prefab" method and the code for creating them from the JSON file. LevelState has the "start_swipe", "end_swipe", "check_collision", "init_hud", "game_over" and "restart_level" methods. Notice that the highest score variable may be different according to the game mode, so we have to leave its name as a variable to be set in other state implementations, as you can see in the "game_over" method.

```
1 var FruitNinja = FruitNinja || {};
2
3 FruitNinja.JSONLevelState = function () {
4     "use strict";
5     Phaser.State.call(this);
6
7     this.prefab_classes = {
8     };
9 };
10
11 FruitNinja.JSONLevelState.prototype =
Object.create(Phaser.State.prototype);
12 FruitNinja.JSONLevelState.prototype.constructor =
FruitNinja.JSONLevelState;
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
13
14 FruitNinja.JSONLevelState.prototype.init = function
(level_data) {
15     "use strict";
16     this.level_data = level_data;
17
18     this.scale.scaleMode = Phaser.ScaleManager.SHOW_ALL;
19     this.scale.pageAlignHorizontally = true;
20     this.scale.pageAlignVertically = true;
21 };
22
23 FruitNinja.JSONLevelState.prototype.create = function () {
24     "use strict";
25     var group_name, prefab_name;
26
27     // create groups
28     this.groups = {};
29     this.level_data.groups.forEach(function (group_name) {
30         this.groups[group_name] = this.game.add.group();
31     }, this);
32
33     // create prefabs
34     this.prefabs = {};
35     for (prefab_name in this.level_data.prefabs) {
36         if
(this.level_data.prefabs.hasOwnProperty(prefab_name)) {
37             // create prefab
38             this.create_prefab(prefab_name,
this.level_data.prefabs[prefab_name]);
39         }
40     }
41 };
42
43 FruitNinja.JSONLevelState.prototype.create_prefab = function
(prefab_name, prefab_data) {
44     "use strict";
45     var prefab_position, prefab;
46     // create object according to its type
47     if (this.prefab_classes.hasOwnProperty(prefab_data.type))
{
48         if (prefab_data.position.x > 0 &&
prefab_data.position.x <= 1) {
49             // position as percentage
50             prefab_position = new
Phaser.Point(prefab_data.position.x * this.game.world.width,
```

```
51                                                      prefab_data.posi
tion.y * this.game.world.height);
52          } else {
53              // position as absolute number
54              prefab_position = prefab_data.position;
55          }
56          prefab = new
this.prefab_classes[prefab_data.type](this, prefab_name,
prefab_position, prefab_data.properties);
57      }
58 };

1 var FruitNinja = FruitNinja || {};
2
3 FruitNinja.LevelState = function () {
4     "use strict";
5     FruitNinja.JSONLevelState.call(this);
6
7     this.prefab_classes = {
8
9     };
10 };
11
12 FruitNinja.LevelState.prototype =
Object.create(FruitNinja.JSONLevelState.prototype);
13 FruitNinja.LevelState.prototype.constructor =
FruitNinja.LevelState;
14
15 FruitNinja.LevelState.prototype.init = function (level_data)
{
16     "use strict";
17     FruitNinja.JSONLevelState.prototype.init.call(this,
level_data);
18     // start physics system
19     this.game.physics.startSystem(Phaser.Physics.ARCADE);
20     this.game.physics.arcade.gravity.y = 1000;
21
22     this.MINIMUM_SWIPE_LENGTH = 50;
23
24     this.score = 0;
25 };
26
27 FruitNinja.LevelState.prototype.create = function () {
28     "use strict";
29     FruitNinja.JSONLevelState.prototype.create.call(this);
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
30
31     // add events to check for swipe
32     this.game.input.onDown.add(this.start_swipe, this);
33     this.game.input.onUp.add(this.end_swipe, this);
34
35     this.init_hud();
36 };
37
38 FruitNinja.LevelState.prototype.start_swipe = function
(pointer) {
39     "use strict";
40     this.start_swipe_point = new Phaser.Point(pointer.x,
pointer.y);
41 };
42
43 FruitNinja.LevelState.prototype.end_swipe = function
(pointer) {
44     "use strict";
45     var swipe_length, cut_style, cut;
46     this.end_swipe_point = new Phaser.Point(pointer.x,
pointer.y);
47     swipe_length = Phaser.Point.distance(this.end_swipe_point,
this.start_swipe_point);
48     // if the swipe length is greater than the minimum, a
swipe is detected
49     if (swipe_length >= this.MINIMUM_SWIPE_LENGTH) {
50         // create a new line as the swipe and check for
collisions
51         cut_style = {line_width: 5, color: 0xE82C0C, alpha:
1};
52         cut = new FruitNinja.Cut(this, "cut", {x: 0, y: 0},
{group: "cuts", start: this.start_swipe_point, end:
this.end_swipe_point, duration: 0.3, style: cut_style});
53         this.swipe = new Phaser.Line(this.start_swipe_point.x,
this.start_swipe_point.y, this.end_swipe_point.x,
this.end_swipe_point.y);
54         this.groups.fruits.forEachAlive(this.check_collision,
this);
55         this.groups.bombs.forEachAlive(this.check_collision,
this);
56         this.groups.special_fruits.forEachAlive(this.check_col
lision, this);
57         this.groups.time_bombs.forEachAlive(this.check_collisi
on, this);
```

```
58          this.groups.clocks.forEachAlive(this.check_collision,
this);
59      }
60  };
61
62  FruitNinja.LevelState.prototype.check_collision = function
(object) {
63      "use strict";
64      var object_rectangle, line1, line2, line3, line4,
intersection;
65      // create a rectangle for the object body
66      object_rectangle = new Phaser.Rectangle(object.body.x,
object.body.y, object.body.width, object.body.height);
67      // check for intersections with each rectangle edge
68      line1 = new Phaser.Line(object_rectangle.left,
object_rectangle.bottom, object_rectangle.left,
object_rectangle.top);
69      line2 = new Phaser.Line(object_rectangle.left,
object_rectangle.top, object_rectangle.right,
object_rectangle.top);
70      line3 = new Phaser.Line(object_rectangle.right,
object_rectangle.top, object_rectangle.right,
object_rectangle.bottom);
71      line4 = new Phaser.Line(object_rectangle.right,
object_rectangle.bottom, object_rectangle.left,
object_rectangle.bottom);
72      intersection = this.swipe.intersects(line1) ||
this.swipe.intersects(line2) || this.swipe.intersects(line3) ||
this.swipe.intersects(line4);
73      if (intersection) {
74          // if an intersection is found, cut the object
75          object.cut();
76      }
77  };
78
79  FruitNinja.LevelState.prototype.init_hud = function () {
80      "use strict";
81      var score_position, score_style, score;
82      // create score prefab
83      score_position = new Phaser.Point(20, 20);
84      score_style = {font: "48px Shojumaru", fill: "#fff"};
85      score = new FruitNinja.Score(this, "score",
score_position, {text: "Fruits: ", style: score_style, group:
"hud"});
86  };
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
87
88  FruitNinja.LevelState.prototype.game_over = function () {
89      "use strict";
90      var game_over_panel, game_over_position, game_over_bitmap,
panel_text_style;
91      // if current score is higher than highest score, update
it
92      if (!localStorage[this.highest_score] || this.score >
localStorage[this.highest_score]) {
93          localStorage[this.highest_score] = this.score;
94      }
95
96      // create a bitmap do show the game over panel
97      game_over_position = new Phaser.Point(0,
this.game.world.height);
98      game_over_bitmap =
this.add.bitmapData(this.game.world.width,
this.game.world.height);
99      game_over_bitmap.ctx.fillStyle = "#000";
100     game_over_bitmap.ctx.fillRect(0, 0,
this.game.world.width, this.game.world.height);
101     panel_text_style = {game_over: {font: "32px Shojumaru",
fill: "#FFF"},
102                         current_score: {font: "20px
Shojumaru", fill: "#FFF"},
103                         highest_score: {font: "18px
Shojumaru", fill: "#FFF"}};
104     // create the game over panel
105     game_over_panel = new FruitNinja.GameOverPanel(this,
"game_over_panel", game_over_position, {texture:
game_over_bitmap, group: "hud", text_style: panel_text_style,
animation_time: 500});
106     this.groups.hud.add(game_over_panel);
107 };
108
109 FruitNinja.LevelState.prototype.restart_level = function ()
{
110     "use strict";
111     this.game.state.restart(true, false, this.level_data);
112 };
```

Now, we can write the code for ClassicState, which represents the game mode we already had. Since most of its logic is already in LevelState and JSONLevelState, we only have

to set the highest score name and overwrite the "init_hud" method to include player lives, as shown below.

```
1 var FruitNinja = FruitNinja || {};
2
3 FruitNinja.ClassicState = function () {
4     "use strict";
5     FruitNinja.LevelState.call(this);
6
7     this.prefab_classes = {
8         "fruit_spawner":
FruitNinja.FruitSpawner.prototype.constructor,
9         "bomb_spawner":
FruitNinja.BombSpawner.prototype.constructor,
10         "special_fruit_spawner":
FruitNinja.SpecialFruitSpawner.prototype.constructor,
11         "background": FruitNinja.Prefab.prototype.constructor
12     };
13 };
14
15 FruitNinja.ClassicState.prototype =
Object.create(FruitNinja.LevelState.prototype);
16 FruitNinja.ClassicState.prototype.constructor =
FruitNinja.ClassicState;
17
18 FruitNinja.ClassicState.prototype.init = function
(level_data) {
19     "use strict";
20     FruitNinja.LevelState.prototype.init.call(this,
level_data);
21
22     this.lives = 3;
23
24     this.highest_score = "classic_score";
25 };
26
27 FruitNinja.ClassicState.prototype.init_hud = function () {
28     "use strict";
29     FruitNinja.LevelState.prototype.init_hud.call(this);
30     var lives_position, lives;
31     // create lives prefab
32     lives_position = new Phaser.Point(0.75 *
this.game.world.width, 20);
```

```
33    lives = new FruitNinja.Lives(this, "lives",
lives_position, {texture: "sword_image", group: "hud", "lives":
3, "lives_spacing": 50});
34 };
```

I'll leave the TimeAttackState code for when we're going to use it.

Custom font

We will add a custom font to show all the text content in our game, like title, menu items, and hud. You can download the font we are going to use here or with the source code. To add it to the game, you can simply add a font-face in your index.html file. However, it might not work for all browsers, according to some reports in Phaser's forums. To circumvent this, you can add a style to load the custom font and a div object in your index.html file to force the browser to load the font.

The index.html with both methods is shown below. Notice that we set the "left" property of the .fontPreload style to be -100px, so the div object won't be visible. After doing that, you can simply use the font family name (Shojumaru) in the style property of our text prefabs.

```
1 <!DOCTYPE html>
2 <html>
3
4 <head>
5 <meta charset="utf-8" />
6 <meta name="viewport" content="width=device-width, initial-
scale=1, maximum-scale=1, minimum-scale=1, user-scalable=no" />
7 <title>Learn Game Development at ZENVA.com</title>
8
9 <style>
10    body {
11       padding: 0px;
12       margin: 0px;
13       background-color: black;
14    }
15             @font-face {
16       font-family: 'Shojumaru';
17       src: url('assets/fonts/shojumaru.ttf');
18    }
19
20             .fontPreload {
21       font-family: Shojumaru;
```

```
22        position:absolute;
23        left:-100px;
24        }
25    </style>
26
27
28  </head>
29
30  <body>
31          <div class="fontPreload">.</div>
32
33          <script type="text/javascript"
src="js/phaser.js"></script>
34
35          <script src="js/states/BootState.js"></script>
36          <script src="js/states/LoadingState.js"></script>
37          <script src="js/states/JSONLevelState.js"></script>
38          <script src="js/states/TitleState.js"></script>
39          <script src="js/states/LevelState.js"></script>
40          <script src="js/states/ClassicState.js"></script>
41          <script src="js/states/TimeAttackState.js"></script>
42
43          <script src="js/prefabs/Prefab.js"></script>
44          <script src="js/prefabs/TextPrefab.js"></script>
45
46          <script src="js/prefabs/Spawners/Spawner.js"></script>
47          <script
src="js/prefabs/Spawners/FruitSpawner.js"></script>
48          <script
src="js/prefabs/Spawners/BombSpawner.js"></script>
49          <script
src="js/prefabs/Spawners/SpecialFruitSpawner.js"></script>
50          <script
src="js/prefabs/Spawners/TimeBombSpawner.js"></script>
51          <script
src="js/prefabs/Spawners/ClockSpawner.js"></script>
52
53          <script
src="js/prefabs/Cuttables/Cuttable.js"></script>
54          <script src="js/prefabs/Cuttables/Fruit.js"></script>
55          <script src="js/prefabs/Cuttables/Bomb.js"></script>
56          <script
src="js/prefabs/Cuttables/SpecialFruit.js"></script>
57          <script
src="js/prefabs/Cuttables/TimeBomb.js"></script>
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
58         <script src="js/prefabs/Cuttables/Clock.js"></script>
59         <script src="js/prefabs/Cuttables/Cut.js"></script>
60

61         <script src="js/prefabs/HUD/Score.js"></script>
62         <script src="js/prefabs/HUD/Lives.js"></script>
63         <script
src="js/prefabs/HUD/RemainingTime.js"></script>
64         <script
src="js/prefabs/HUD/GameOverPanel.js"></script>
65         <script src="js/prefabs/HUD/Menu.js"></script>
66         <script src="js/prefabs/HUD/MenuItem.js"></script>
67         <script
src="js/prefabs/HUD/StartGameItem.js"></script>
68

69         <script type="text/javascript"
src="js/main.js"></script>
70 </body>
71 </html>
```

Title screen

The title screen will have the game title and a menu with the game mode options. To do that, we have to create MenuItem and Menu prefabs. The MenuItem will be a text prefab that shows a game mode. When the menu selection is over an item, we will play a tween animation to change the item scale up and down. When the player selects an item, we want it to execute some action, by calling the "select" method. By default, this method does nothing, so we will write new prefabs that will extend MenuItem and overwrite this method.

The MenuItem prefab code is shown below. In the constructor, we create a tween animation for the scale object. Then, we add two children to this tween, by calling its "to" method twice, one for increasing the scale and one for decreasing it. Finally, we use to "repeatAll" method to keep the tween repeating after all children are executed. Finally, in the "selection_over" method we start the tween, while in the "selection_out" method we stop it.

```
1 var FruitNinja = FruitNinja || {};
2
3 FruitNinja.MenuItem = function (game_state, name, position,
properties) {
4     "use strict";
5     FruitNinja.TextPrefab.call(this, game_state, name,
position, properties);
```

```
6
7    this.anchor.setTo(0.5);
8
9    this.on_selection_animation =
this.game_state.game.add.tween(this.scale);
10    this.on_selection_animation.to({x: 1.5 * this.scale.x, y:
1.5 * this.scale.y}, 500);
11    this.on_selection_animation.to({x: this.scale.x, y:
this.scale.y}, 500);
12    this.on_selection_animation.repeatAll(-1);
13 };
14
15 FruitNinja.MenuItem.prototype =
Object.create(FruitNinja.TextPrefab.prototype);
16 FruitNinja.MenuItem.prototype.constructor =
FruitNinja.MenuItem;
17
18 FruitNinja.MenuItem.prototype.selection_over = function () {
19    "use strict";
20    if (this.on_selection_animation.isPaused) {
21        this.on_selection_animation.resume();
22    } else {
23        this.on_selection_animation.start();
24    }
25 };
26
27 FruitNinja.MenuItem.prototype.selection_out = function () {
29    "use strict";
30    this.on_selection_animation.pause();
31 };
32
33 FruitNinja.MenuItem.prototype.select = function () {
34    "use strict";
35    // the default item does nothing
36 };
```

Now, we will write the Menu prefab code, which will have a list of items and will navigate through them. The Menu code is shown below. In the constructor, the menu items are received from the properties parameter and the first one is under selection. In the "update" method we check for the directionals to navigate through the items and the spacebar to select the current item. Notice that, when navigating through the items, we have to remove the selection from the current item and change it to the new one. When the spacebar is pressed, we just have to select the current item.

```javascript
1 var FruitNinja = FruitNinja || {};
2
3 FruitNinja.Menu = function (game_state, name, position,
properties) {
4     "use strict";
5     var live_index, life;
6     FruitNinja.Prefab.call(this, game_state, name, position,
properties);
7
8     this.visible = false;
9
10     this.menu_items = properties.menu_items;
11     this.current_item_index = 0;
12     this.menu_items[0].selection_over();
13
14     this.cursor_keys =
this.game_state.game.input.keyboard.createCursorKeys();
15 };
16
17 FruitNinja.Menu.prototype =
Object.create(FruitNinja.Prefab.prototype);
18 FruitNinja.Menu.prototype.constructor = FruitNinja.Menu;
19
20 FruitNinja.Menu.prototype.update = function () {
21     "use strict";
22     if (this.cursor_keys.up.isDown && this.current_item_index
> 0) {
23         // navigate to previous item
24         this.menu_items[this.current_item_index].selection_out
();
25         this.current_item_index -= 1;
26         this.menu_items[this.current_item_index].selection_ove
r();
27     } else if (this.cursor_keys.down.isDown &&
this.current_item_index < this.menu_items.length - 1) {
28         // navigate to next item
29         this.menu_items[this.current_item_index].selection_out
();
30         this.current_item_index += 1;
31         this.menu_items[this.current_item_index].selection_ove
r();
32     }
33
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
34     if
(this.game_state.game.input.keyboard.isDown(Phaser.Keyboard.SPAC
EBAR)) {
35          this.menu_items[this.current_item_index].select();
36     }
37 };
```

Finally, we can create the TitleState by using the Menu and MenuItem prefabs. The menu items will be loaded from the JSON file (which you can check in the source code), so we just have to add the game title and the menu in the "create" method. The title is simply a text prefab. To pass the menu items to the menu we iterate through the "menu_items" group, saving them in an array, which will be passed to the menu.

```
1 var FruitNinja = FruitNinja || {};
2
3 FruitNinja.TitleState = function () {
4     "use strict";
5     FruitNinja.JSONLevelState.call(this);
6
7     this.prefab_classes = {
8          "start_game_item":
FruitNinja.StartGameItem.prototype.constructor,
9          "background": FruitNinja.Prefab.prototype.constructor
10     };
11 };
12
13 FruitNinja.TitleState.prototype =
Object.create(FruitNinja.JSONLevelState.prototype);
14 FruitNinja.TitleState.prototype.constructor =
FruitNinja.TitleState;
15
16 FruitNinja.TitleState.prototype.create = function () {
17     "use strict";
18     var title_position, title_style, title, menu_position,
menu_items, menu_properties, menu;
19     FruitNinja.JSONLevelState.prototype.create.call(this);
20
21     // adding title
22     title_position = new Phaser.Point(0.5 *
this.game.world.width, 0.3 * this.game.world.height);
23     title_style = {font: "72px Shojumaru", fill: "#FFF"};
24     title = new FruitNinja.TextPrefab(this, "title",
title_position, {text: "Fruit Ninja", style: title_style, group:
"hud"});
```

```
25      title.anchor.setTo(0.5);
26
27      // adding menu
28      menu_position = new Phaser.Point(0, 0);
29      menu_items = [];
30      this.groups.menu_items.forEach(function (menu_item) {
31          menu_items.push(menu_item);
32      }, this);
33      menu_properties = {texture: "", group: "background",
menu_items: menu_items};
34      menu = new FruitNinja.Menu(this, "menu", menu_position,
menu_properties);
35 };
```

You can already run the game with the title screen, to see it working.

Time attack state

Since we already have most of the game logic in the LevelState, the TimeAttackState is
easy to implement, as shown below. Notice that, besides setting the highest score
variable, we only have to add the remaining time to the hud, which is updated in the
"update" method.

```
1 var FruitNinja = FruitNinja || {};
2
3 FruitNinja.TimeAttackState = function () {
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```javascript
4     "use strict";
5     FruitNinja.LevelState.call(this);
6
7     this.prefab_classes = {
8         "fruit_spawner":
FruitNinja.FruitSpawner.prototype.constructor,
9         "time_bomb_spawner":
FruitNinja.TimeBombSpawner.prototype.constructor,
10        "clock_spawner":
FruitNinja.ClockSpawner.prototype.constructor,
11        "special_fruit_spawner":
FruitNinja.SpecialFruitSpawner.prototype.constructor,
12        "background": FruitNinja.Prefab.prototype.constructor
13     };
14 };
15
16 FruitNinja.TimeAttackState.prototype =
Object.create(FruitNinja.LevelState.prototype);
17 FruitNinja.TimeAttackState.prototype.constructor =
FruitNinja.TimeAttackState;
18
19 FruitNinja.TimeAttackState.prototype.init = function
(level_data) {
20     "use strict";
21     FruitNinja.LevelState.prototype.init.call(this,
level_data);
22
23     this.remaining_time = Phaser.Timer.SECOND * 60;
24
25     this.highest_score = "time_attack_score";
26 };
27
28 FruitNinja.TimeAttackState.prototype.update = function () {
29     "use strict";
30     if (this.remaining_time > 0) {
31         this.remaining_time -= this.game.time.elapsed;
32         if (this.remaining_time <= 0) {
33             this.game_over();
34             this.remaining_time = 0;
35         }
36     }
37 };
38
39 FruitNinja.TimeAttackState.prototype.init_hud = function () {
40     "use strict";
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
41    FruitNinja.LevelState.prototype.init_hud.call(this);
42    var remaining_time_position, remaining_time_style,
remaining_time;
43    // show remaining time
44    remaining_time_position = new Phaser.Point(0.5 *
this.game.world.width, 20);
45    remaining_time_style = {font: "48px Shojumaru", fill:
"#fff"};
46    remaining_time = new FruitNinja.RemainingTime(this,
"remaining_time", remaining_time_position, {text: "Remaining
time: ", style: remaining_time_style, group: "hud"});
47 };
```

To show the remaining time we create a RemainingTime prefab, which is simply a text prefab that shows the remaining time value. To update it, we use "this.game.time.elapsed", which contains the elapsed time since the last update, in milliseconds.

```
1 var FruitNinja = FruitNinja || {};
2
3 FruitNinja.RemainingTime = function (game_state, name,
position, properties) {
4    "use strict";
5    FruitNinja.TextPrefab.call(this, game_state, name,
position, properties);
6 };
7
8 FruitNinja.RemainingTime.prototype =
Object.create(FruitNinja.TextPrefab.prototype);
9 FruitNinja.RemainingTime.prototype.constructor =
FruitNinja.RemainingTime;
10
11 FruitNinja.RemainingTime.prototype.update = function () {
12    "use strict";
13    // update the text to show the remaining time in seconds
14    this.text = "Remaining time: " +
this.game_state.remaining_time / Phaser.Timer.SECOND;
15 };
```

You can already try playing the Time Attack mode with the regular prefabs and see if it's

working.

Clock and time bomb

Since we have our generic Cuttable and Spawner prefabs, adding the clock and the time bomb will be easily done as we did with the special fruit in the last tutorial.

First, we will create the Clock and TimeBomb prefabs, which will extend Cuttable, as shown below. For each one, the only thing we have to do is change the "cut" method. In the Clock prefab, we increase the remaining time by 3 seconds, while in the TimeBomb we decrease it by 5 seconds.

```
1 var FruitNinja = FruitNinja || {};
2
3 FruitNinja.Clock = function (game_state, name, position,
properties) {
4     "use strict";
5     FruitNinja.Cuttable.call(this, game_state, name, position,
properties);
6
7     this.body.setSize(20, 20);
8 };
9
10 FruitNinja.Clock.prototype =
Object.create(FruitNinja.Cuttable.prototype);
11 FruitNinja.Clock.prototype.constructor = FruitNinja.Clock;
12
13 FruitNinja.Clock.prototype.cut = function () {
14     "use strict";
15     FruitNinja.Cuttable.prototype.cut.call(this);
16     // if a time bomb is cut, increase the remaining time by 3
seconds
17     this.game_state.remaining_time += Phaser.Timer.SECOND * 3;
18     this.kill();
19 };
```

```
1 var FruitNinja = FruitNinja || {};
2
3 FruitNinja.TimeBomb = function (game_state, name, position,
properties) {
4     "use strict";
5     FruitNinja.Cuttable.call(this, game_state, name, position,
properties);
6
7     this.body.setSize(20, 20);
8 };
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Page 141

```
9
10 FruitNinja.TimeBomb.prototype =
Object.create(FruitNinja.Cuttable.prototype);
11 FruitNinja.TimeBomb.prototype.constructor =
FruitNinja.TimeBomb;
12
13 FruitNinja.TimeBomb.prototype.cut = function () {
14     "use strict";
15     FruitNinja.Cuttable.prototype.cut.call(this);
16     // if a time bomb is cut, decrease the remaining time by 5
seconds
17     this.game_state.remaining_time -= Phaser.Timer.SECOND * 5;
18     this.kill();
19 };
```

Next, we will create the spawner for both prefabs. Here the only method we have to change is "create_object", which will return a new Clock for the ClockSpawner and a new TimeBomb for the TimeBombSpawner.

```
1 var FruitNinja = FruitNinja || {};
2
3 FruitNinja.ClockSpawner = function (game_state, name,
position, properties) {
4     "use strict";
5     FruitNinja.Spawner.call(this, game_state, name, position,
properties);
6 };
7
8 FruitNinja.ClockSpawner.prototype =
Object.create(FruitNinja.Spawner.prototype);
9 FruitNinja.ClockSpawner.prototype.constructor =
FruitNinja.ClockSpawner;
10
11 FruitNinja.ClockSpawner.prototype.create_object = function
(name, position, velocity) {
12     "use strict";
13     // return new time bomb
14     console.log("spawning clock");
15     return new FruitNinja.Clock(this.game_state, name,
position, {texture: "clock_image", group: "clocks", velocity:
velocity});
16 };
```

```
1 var FruitNinja = FruitNinja || {};
```

```
2
3 FruitNinja.TimeBombSpawner = function (game_state, name,
position, properties) {
4    "use strict";
5    FruitNinja.Spawner.call(this, game_state, name, position,
properties);
6 };
7
8 FruitNinja.TimeBombSpawner.prototype =
Object.create(FruitNinja.Spawner.prototype);
9 FruitNinja.TimeBombSpawner.prototype.constructor =
FruitNinja.TimeBombSpawner;
10
11 FruitNinja.TimeBombSpawner.prototype.create_object = function
(name, position, velocity) {
12    "use strict";
13    // return new time bomb
14    return new FruitNinja.TimeBomb(this.game_state, name,
position, {texture: "time_bomb_image", group: "time_bombs",
velocity: velocity});
15 };
```

Now, you can add the two new spawner in the Time Attack mode and try playing it.

Finishing the game

And that concludes our Fruit Ninja game.

How to Make a Bomberman Game in Phaser – Part 1
By Renan Oliveira

In this tutorial we will start building a Bomberman game. We will add the basic structure in this tutorial and keep adding content in the following tutorials. In this tutorial, I will cover the following content:

- Reading a Tiled map
- Creating a player that moves in four directions and drop bombs
- Creating a bomb that explodes when its animation ends
- Creating explosions when a bomb explodes that kills the player and enemies
- Creating an enemy that walks in one axis (horizontal or vertical)

To read this tutorial, it is important that you are familiar with the following concepts:

- Javascript and object-oriented concepts.
- Basic Phaser concepts, such as: states, sprites, groups and arcade physics
- Creating maps using Tiled map editor

Assets copyright

The assets used in this tutorial were created by Cem Kalyoncu/cemkalyoncu and Matt Hackett/richtaur and made available by "usr_share" through the creative commons license, wich allows commercial use under attribution. You can download them in http://opengameart.org/content/bomb-party-the-complete-set or by downloading the source code.

Source code files

You can download the tutorial source code files here.

JSON level file

Although our map is created by Tiled, we will still use a JSON level file to describe the assets we need (including the map file), the groups of our game and the map properties. The JSON level file we are going to use is shown below. Notice that we describe each asset according to its type, the group names and the map key and tilesets.

```
1 {
2     "assets": {
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
3          "map_tiles": {"type": "image", "source":
"assets/images/bomberman_spritesheet.png"},
4          "player_spritesheet": {"type": "spritesheet", "source":
"assets/images/player_spritesheet.png", "frame_width": 16,
"frame_height": 16},
5          "bomb_spritesheet": {"type": "spritesheet", "source":
"assets/images/bomb_spritesheet.png", "frame_width": 16,
"frame_height": 16},
6          "enemy_spritesheet": {"type": "spritesheet", "source":
"assets/images/enemy_spritesheet.png", "frame_width": 16,
"frame_height": 16},
7          "explosion_image": {"type": "image", "source":
"assets/images/explosion.png"},
8
9          "level_tilemap": {"type": "tilemap", "source":
"assets/maps/level1_map.json"}
10     },
11     "groups": [
12         "explosions",
13         "bombs",
14         "enemies",
15         "players"
16     ],
17     "map": {
18         "key": "level_tilemap",
19         "tilesets": ["map_tiles"]
20     }
21 }
```

Creating the map

In this tutorial I will focus on building the game by using a Tiled map, and not in creating the map. So, feel free to create your own map or to use the one provided in the source code. Some details are important because of the way we're going to read this map:

1   For each collidable layer, you have to add a collision property that is equals true
2   For each object you have to properly define it's type (it will be used to instantiate the correct prefab) and add at least properties with its texture and group
3   The map must be saved in JSON format
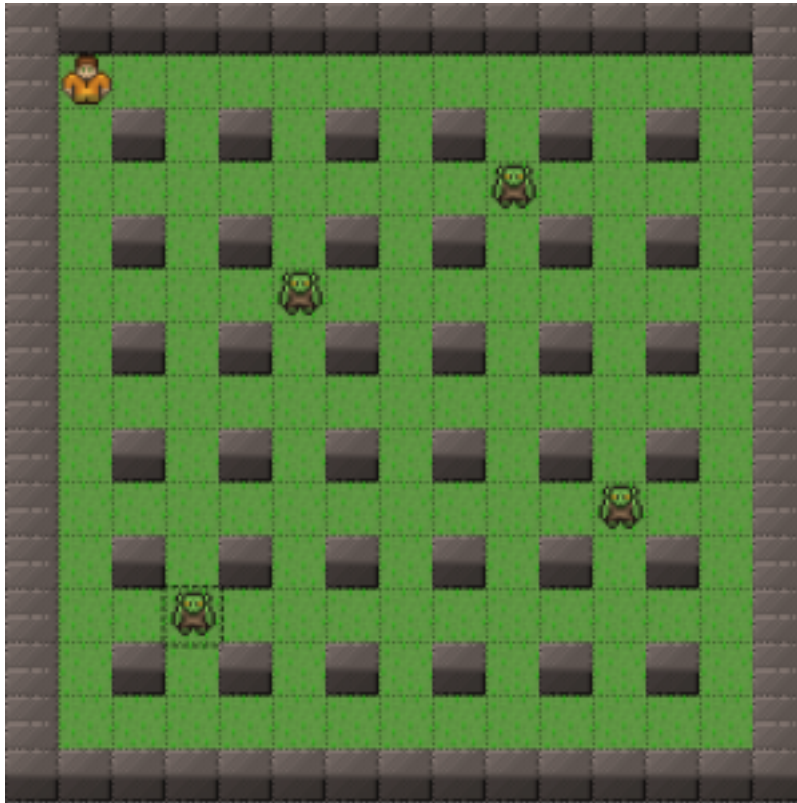
The image below shows the map I'm going to use:

Game states

We will use the following states to run our game:

- Boot State: loads a json file with the level information and starts the Loading State
- Loading Sate: loads all the game assets, and starts the next State
- Tiled State: loads a tiled map file

The BootState code is shown below. It is only responsible for loading the JSON level file and call the LoadingState.

```
1 var Bomberman = Bomberman || {};
2
3 Bomberman.BootState = function () {
4     "use strict";
5     Phaser.State.call(this);
6 };
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
7
8 Bomberman.BootState.prototype =
Object.create(Phaser.State.prototype);
9 Bomberman.BootState.prototype.constructor =
Bomberman.BootState;
10
11 Bomberman.BootState.prototype.init = function (level_file,
next_state) {
12      "use strict";
13      this.level_file = level_file;
14      this.next_state = next_state;
15 };
16
17 Bomberman.BootState.prototype.preload = function () {
18      "use strict";
19      this.load.text("level1", this.level_file);
20 };
21
22 Bomberman.BootState.prototype.create = function () {
23      "use strict";
24      var level_text, level_data;
25      level_text = this.game.cache.getText("level1");
26      level_data = JSON.parse(level_text);
27      this.game.state.start("LoadingState", true, false,
level_data, this.next_state);
28 };
```

The LoadingState is responsible for loading all the assets that will be used in this level, as shown below. As it can be seen, the "preload" method iterate through all assets defined in the JSON level file and load it calling the corresponding method. After all assets are loaded, it calls the next state, which will be TiledState.

```
1 var Bomberman = Bomberman || {};
2
3 Bomberman.LoadingState = function () {
4      "use strict";
5      Phaser.State.call(this);
6 };
7
8 Bomberman.LoadingState.prototype =
Object.create(Phaser.State.prototype);
9 Bomberman.LoadingState.prototype.constructor =
Bomberman.LoadingState;
10
```

```
11 Bomberman.LoadingState.prototype.init = function (level_data,
next_state) {
12     "use strict";
13     this.level_data = level_data;
14     this.next_state = next_state;
15 };
16
17 Bomberman.LoadingState.prototype.preload = function () {
18     "use strict";
19     var assets, asset_loader, asset_key, asset;
20     assets = this.level_data.assets;
21     for (asset_key in assets) { // load assets according to
asset key
22         if (assets.hasOwnProperty(asset_key)) {
23             asset = assets[asset_key];
24             switch (asset.type) {
25             case "image":
26                 this.load.image(asset_key, asset.source);
27                 break;
28             case "spritesheet":
29                 this.load.spritesheet(asset_key, asset.source,
asset.frame_width, asset.frame_height, asset.frames,
asset.margin, asset.spacing);
30                 break;
31             case "tilemap":
32                 this.load.tilemap(asset_key, asset.source,
null, Phaser.Tilemap.TILED_JSON);
33                 break;
34             }
35         }
36     }
37 };
38
39 Bomberman.LoadingState.prototype.create = function () {
40     "use strict";
41     this.game.state.start(this.next_state, true, false,
this.level_data);
42 };
```

Reading the Tiled map

We will read the Tiled map in the TiledState, which the source code is shown below.
First, in the init method we add the tilemap by using the asset described in the JSON file.

Then, for each tileset image described in the JSON file, the state adds it to the map. In our game, we will use only one tileset image.

```javascript
1 var Bomberman = Bomberman || {};
2
3 Bomberman.TiledState = function () {
4     "use strict";
5     Phaser.State.call(this);
6
7     this.prefab_classes = {
8         "player": Bomberman.Player.prototype.constructor,
9         "enemy": Bomberman.Enemy.prototype.constructor
10     };
11 };
12
13 Bomberman.TiledState.prototype =
Object.create(Phaser.State.prototype);
14 Bomberman.TiledState.prototype.constructor =
Bomberman.TiledState;
15
16 Bomberman.TiledState.prototype.init = function (level_data) {
17     "use strict";
18     var tileset_index;
19     this.level_data = level_data;
20
21     this.scale.scaleMode = Phaser.ScaleManager.SHOW_ALL;
22     this.scale.pageAlignHorizontally = true;
23     this.scale.pageAlignVertically = true;
24
25     // start physics system
26     this.game.physics.startSystem(Phaser.Physics.ARCADE);
27     this.game.physics.arcade.gravity.y = 0;
28
29     // create map and set tileset
30     this.map = this.game.add.tilemap(level_data.map.key);
31     tileset_index = 0;
32     this.map.tilesets.forEach(function (tileset) {
33         this.map.addTilesetImage(tileset.name,
level_data.map.tilesets[tileset_index]);
34         tileset_index += 1;
35     }, this);
36 };
37
38 Bomberman.TiledState.prototype.create = function () {
```

```
39      "use strict";
40      var group_name, object_layer, collision_tiles;
41
42      // create map layers
43      this.layers = {};
44      this.map.layers.forEach(function (layer) {
45          this.layers[layer.name] =
this.map.createLayer(layer.name);
46          if (layer.properties.collision) { // collision layer
47              collision_tiles = [];
48              layer.data.forEach(function (data_row) { // find
tiles used in the layer
49                  data_row.forEach(function (tile) {
50                      // check if it's a valid tile index and
isn't already in the list
51                      if (tile.index > 0 &&
collision_tiles.indexOf(tile.index) === -1) {
52                          collision_tiles.push(tile.index);
53                      }
54                  }, this);
55              }, this);
56              this.map.setCollision(collision_tiles, true,
layer.name);
57          }
58      }, this);
59      // resize the world to be the size of the current layer
60      this.layers[this.map.layer.name].resizeWorld();
61
62      // create groups
63      this.groups = {};
64      this.level_data.groups.forEach(function (group_name) {
65          this.groups[group_name] = this.game.add.group();
66      }, this);
67
68      this.prefabs = {};
69
70      for (object_layer in this.map.objects) {
71          if (this.map.objects.hasOwnProperty(object_layer)) {
72              // create layer objects
73              this.map.objects[object_layer].forEach(this.create
_object, this);
74          }
75      }
76 };
77
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Page 150

```
78 Bomberman.TiledState.prototype.create_object = function
(object) {
79    "use strict";
80    var object_y, position, prefab;
81    // tiled coordinates starts in the bottom left corner
82    object_y = (object.gid) ? object.y - (this.map.tileHeight
/ 2) : object.y + (object.height / 2);
83    position = {"x": object.x + (this.map.tileHeight / 2),
"y": object_y};
84    // create object according to its type
85    if (this.prefab_classes.hasOwnProperty(object.type)) {
86        prefab = new this.prefab_classes[object.type](this,
object.name, position, object.properties);
87    }
88    this.prefabs[object.name] = prefab;
89 };
90
91 Bomberman.TiledState.prototype.game_over = function () {
92    "use strict";
93    localStorage.clear();
94    this.game.state.restart(true, false, this.level_data);
95 };
```

In the create method, we create the layers, groups and prefabs. First, we iterate through all layers in the map object, creating and saving it. If a layer has a collision property we set it as collidable, by looking for all valid tile indices on it. To create the groups, we just iterate through all groups in the JSON file, creating them.

To create the prefabs, first we will define a generic Prefab class, as shown below. It extends Phaser.Sprite and is responsible for adding itself to its group. So, it is important that the properties parameter contain the texture and group information.

```
1 var Bomberman = Bomberman || {};
2
3 Bomberman.Prefab = function (game_state, name, position,
properties) {
4    "use strict";
5    Phaser.Sprite.call(this, game_state.game, position.x,
position.y, properties.texture);
6
7    this.game_state = game_state;
8
9    this.name = name;
10
```

```
11    this.game_state.groups[properties.group].add(this);
12    this.frame = +properties.frame;
13
14    this.game_state.prefabs[name] = this;
15 };
16
17 Bomberman.Prefab.prototype =
Object.create(Phaser.Sprite.prototype);
18 Bomberman.Prefab.prototype.constructor = Bomberman.Prefab;
```

Finally, to create the prefabs we iterate through the objects in the objects layer. There are three important things to notice in the "create_object": 1) Tiled coordinate system starts in the bottom left corner, while in Phaser it starts in the top left corner. As a consequence, we have to change the y coordinate accordingly. 2) We instantiate the correct prefab by accessing a "prefab_classes" map with the object type. This is possible only because all prefabs have the same constructor. 3) Since our generic Prefab class requires a texture and group properties, those properties must be defined in the Tiled map objects.

Creating the player

Our player prefab can walk in the four directions and drop bombs. First, in the constructor we have to set its properties, such as walking speed and bomb duration, and create the animations for walking in each direction.

In the update method, we check for the cursors for movement and the spacebar for dropping bomb. For each direction, we check if its corresponding cursor key was pressed and if the player is not already moving in the opposite direction. If that's the case, we set its velocity accordingly. If the player just started moving, we start playing its corresponding walking animation. Also, since we're using the same animation for both left and right directions, we have to set the prefab scale properly.

At the end of the update method, if the player velocity is 0 in both axis we stop the current animation and set the stopped frame according to the facing property of its physical body (the facing property is automatically updated by the physics engine. For more information, you can check the <u>documentation</u>).

To drop bombs, we check for the spacebar key, and if it is pressed we call the "drop_bomb" method. To keep the player from dropping multiple bombs at the same time, when one bomb is dropped, we set a "dropping_bomb" variable to true, and don't allow other bomb to be dropped. We only set this variable to false again when the spacebar key is released.

In the "drop_bomb" method we will use a pool of objects to avoid creating unnecessary objects. The idea is to keep a group with all the game bombs and, when we need a new one, we get the first dead object from this group. If there is not a dead object, we create a new one. If there is one, we just reuse it, avoiding to keep creating new objects unnecessarily. Also, since this is a method we will use in different places, I put it in a Utils file, receiving as a parameter the pool, the prefab constructor and its constructor parameters.

```
1 var Bomberman = Bomberman || {};
2
3 Bomberman.Player = function (game_state, name, position,
properties) {
4     "use strict";
5     Bomberman.Prefab.call(this, game_state, name, position,
properties);
6
7     this.anchor.setTo(0.5);
8
```

```
9     this.walking_speed = +properties.walking_speed;
10    this.bomb_duration = +properties.bomb_duration;
11    this.dropping_bomb = false;
12
13    this.animations.add("walking_down", [1, 2, 3], 10, true);
14    this.animations.add("walking_left", [4, 5, 6, 7], 10,
true);
15    this.animations.add("walking_right", [4, 5, 6, 7], 10,
true);
16    this.animations.add("walking_up", [0, 8, 9], 10, true);
17
18    this.stopped_frames = [1, 4, 4, 0, 1];
19
20    this.game_state.game.physics.arcade.enable(this);
21    this.body.setSize(14, 12, 0, 4);
22
23    this.cursors =
this.game_state.game.input.keyboard.createCursorKeys();
24 };
25
26 Bomberman.Player.prototype =
Object.create(Bomberman.Prefab.prototype);
27 Bomberman.Player.prototype.constructor = Bomberman.Player;
28
29 Bomberman.Player.prototype.update = function () {
30    "use strict";
31    this.game_state.game.physics.arcade.collide(this,
this.game_state.layers.collision);
32    this.game_state.game.physics.arcade.collide(this,
this.game_state.groups.bombs);
33    this.game_state.game.physics.arcade.overlap(this,
this.game_state.groups.explosions, this.kill, null, this);
34    this.game_state.game.physics.arcade.overlap(this,
this.game_state.groups.enemies, this.kill, null, this);
35
36    if (this.cursors.left.isDown && this.body.velocity.x <= 0)
{
37        // move left
38        this.body.velocity.x = -this.walking_speed;
39        if (this.body.velocity.y === 0) {
40            // change the scale, since we have only one
animation for left and right directions
41            this.scale.setTo(-1, 1);
42            this.animations.play("walking_left");
43        }
```

```
44      } else if (this.cursors.right.isDown &&
this.body.velocity.x >= 0) {
45          // move right
46          this.body.velocity.x = +this.walking_speed;
47          if (this.body.velocity.y === 0) {
48              // change the scale, since we have only one
animation for left and right directions
49              this.scale.setTo(1, 1);
50              this.animations.play("walking_right");
51          }
52      } else {
53          this.body.velocity.x = 0;
54      }
55
56      if (this.cursors.up.isDown && this.body.velocity.y <= 0) {
57          // move up
58          this.body.velocity.y = -this.walking_speed;
59          if (this.body.velocity.x === 0) {
60              this.animations.play("walking_up");
61          }
62      } else if (this.cursors.down.isDown &&
this.body.velocity.y >= 0) {
63          // move down
64          this.body.velocity.y = +this.walking_speed;
65          if (this.body.velocity.x === 0) {
66              this.animations.play("walking_down");
67          }
68      } else {
69          this.body.velocity.y = 0;
70      }
71
72      if (this.body.velocity.x === 0 && this.body.velocity.y ===
0) {
73          // stop current animation
74          this.animations.stop();
75          this.frame = this.stopped_frames[this.body.facing];
76      }
77
78      if (!this.dropping_bomb &&
this.game_state.input.keyboard.isDown(Phaser.Keyboard.SPACEBAR))
{
79          this.drop_bomb();
80          this.dropping_bomb = true;
81      }
82
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
83    if (this.dropping_bomb &&
!this.game_state.input.keyboard.isDown(Phaser.Keyboard.SPACEBAR)
) {
84        this.dropping_bomb = false;
85    }
86 };
87
88 Bomberman.Player.prototype.drop_bomb = function () {
89    "use strict";
90    var bomb, bomb_name, bomb_position, bomb_properties;
91    // get the first dead bomb from the pool
92    bomb_name = this.name + "_bomb_" +
this.game_state.groups.bombs.countLiving();
93    bomb_position = new Phaser.Point(this.x, this.y);
94    bomb_properties = {"texture": "bomb_spritesheet", "group":
"bombs", bomb_radius: 3};
95    bomb =
Bomberman.create_prefab_from_pool(this.game_state.groups.bombs,
Bomberman.Bomb.prototype.constructor, this.game_state,
bomb_name, bomb_position, bomb_properties);
96 };
```

You can already create a map with your player and see if it's walking correctly.
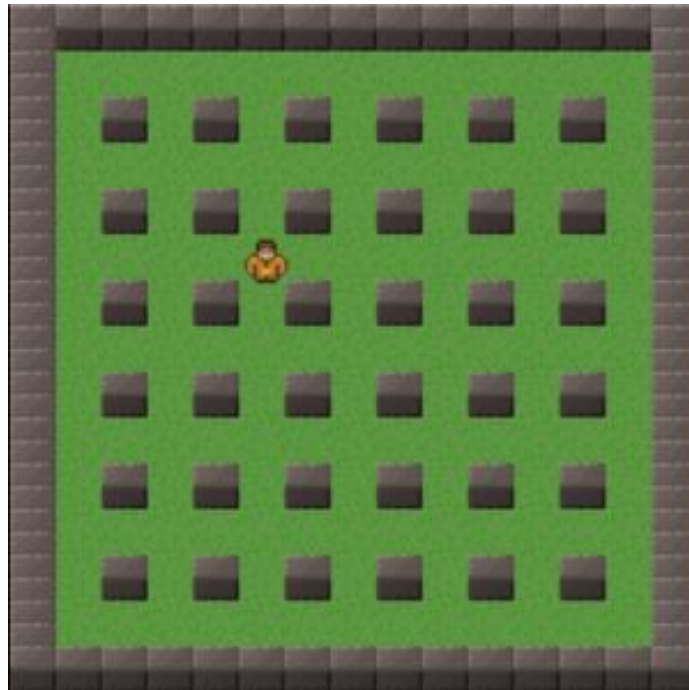
Creating the bomb

The Bomb prefab will start with an exploding animation and when the animation is done it will explode. For this, we set a bomb radius property in the constructor and create its animation, making it call the kill method when the animation is complete.

To create the explosions, we have to overwrite the kill method. In the new kill method we start creating an explosion in the bomb position, and then we call the "create_explosions" method to create them in each of the four directions. This method's parameters are an initial index, final index, step and axis. By iterating through those indices, we create new explosions in a position given by the current index and the axis. If a wall tile is found during this process, we stop it. To do this, we use the "getTileFromXY" from Phaser.Tilemap, which returns a tile given a position in the world (you can check the documentation for more information). Notice that, to allow negative indices we must compare the absolute values of "index" and "final_index". Also, we're using a pool of objects for the explosion, since we will be creating a lot of them.

There is one last thing we have to do in the Bomb prefab. In the reset method we have to restart the exploding animation, otherwise it would not play when we reused a bomb object from the bombs pool.

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```javascript
1 var Bomberman = Bomberman || {};
2
3 Bomberman.Bomb = function (game_state, name, position,
properties) {
4     "use strict";
5     Bomberman.Prefab.call(this, game_state, name, position,
properties);
6
7     this.anchor.setTo(0.5);
8
9     this.bomb_radius = +properties.bomb_radius;
10
11     this.game_state.game.physics.arcade.enable(this);
12     this.body.immovable = true;
13
14     this.exploding_animation =
this.animations.add("exploding", [0, 2, 4], 1, false);
15     this.exploding_animation.onComplete.add(this.kill, this);
16     this.animations.play("exploding");
17 };
18
19 Bomberman.Bomb.prototype =
Object.create(Bomberman.Prefab.prototype);
20 Bomberman.Bomb.prototype.constructor = Bomberman.Bomb;
21
22 Bomberman.Bomb.prototype.reset = function (position_x,
position_y) {
23     "use strict";
24     Phaser.Sprite.prototype.reset.call(this, position_x,
position_y);
25     this.exploding_animation.restart();
26 };
27
28 Bomberman.Bomb.prototype.kill = function () {
29     "use strict";
30     Phaser.Sprite.prototype.kill.call(this);
31     var explosion_name, explosion_position,
explosion_properties, explosion;
32     explosion_name = this.name + "_explosion_" +
this.game_state.groups.explosions.countLiving();
33     explosion_position = new Phaser.Point(this.position.x,
this.position.y);
34     explosion_properties = {texture: "explosion_image", group:
"explosions", duration: 0.5};
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
35     // create an explosion in the bomb position
36     explosion =
Bomberman.create_prefab_from_pool(this.game_state.groups.explosi
ons, Bomberman.Explosion.prototype.constructor, this.game_state,
37                                              explosio
n_name, explosion_position, explosion_properties);
38
39     // create explosions in each direction
40     this.create_explosions(-1, -this.bomb_radius, -1, "x");
41     this.create_explosions(1, this.bomb_radius, +1, "x");
42     this.create_explosions(-1, -this.bomb_radius, -1, "y");
43     this.create_explosions(1, this.bomb_radius, +1, "y");
44 };
45
46 Bomberman.Bomb.prototype.create_explosions = function
(initial_index, final_index, step, axis) {
47     "use strict";
48     var index, explosion_name, explosion_position, explosion,
explosion_properties, tile;
49     explosion_properties = {texture: "explosion_image", group:
"explosions", duration: 0.5};
50     for (index = initial_index; Math.abs(index) <=
Math.abs(final_index); index += step) {
51         explosion_name = this.name + "_explosion_" +
this.game_state.groups.explosions.countLiving();
52         // the position is different accoring to the axis
53         if (axis === "x") {
54             explosion_position = new
Phaser.Point(this.position.x + (index * this.width),
this.position.y);
55         } else {
56             explosion_position = new
Phaser.Point(this.position.x, this.position.y + (index *
this.height));
57         }
58         tile =
this.game_state.map.getTileWorldXY(explosion_position.x,
explosion_position.y, this.game_state.map.tileWidth,
this.game_state.map.tileHeight, "collision");
59         if (!tile) {
60             // create a new explosion in the new position
61             explosion =
Bomberman.create_prefab_from_pool(this.game_state.groups.explosi
ons, Bomberman.Explosion.prototype.constructor, this.game_state,
explosion_name, explosion_position, explosion_properties);
```

```
62              } else {
63                  break;
64              }
65          }
66  };
```

You can already try placing bombs to see if its working. Since we still didn't create the Explosion prefab yet, you can just comment this part of the code or create a dummy prefab for now.
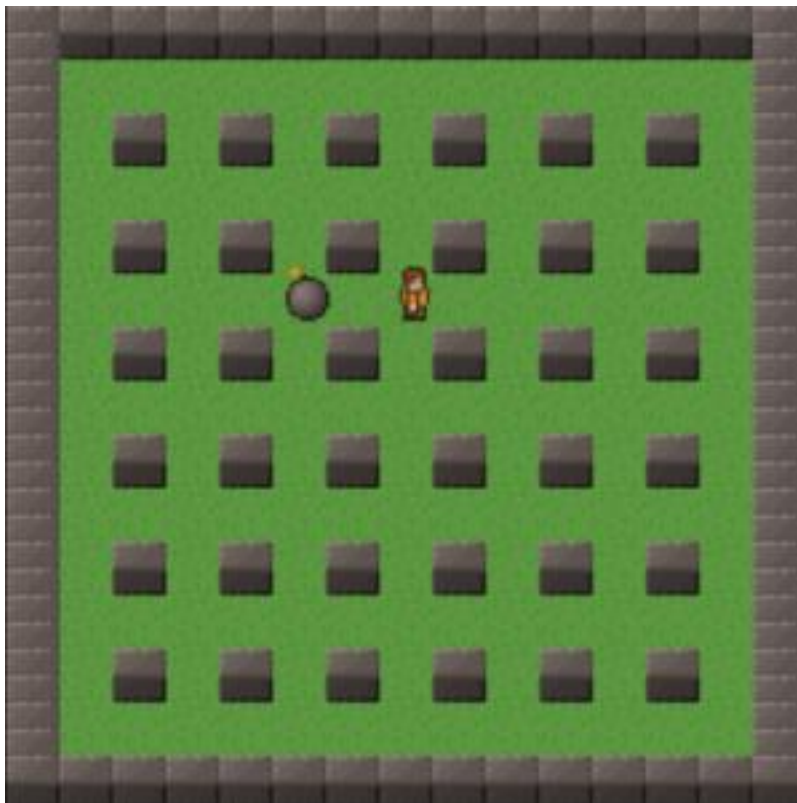
Creating explosions

The Explosion prefab will have a duration, which will define how long it will last until it disappears. This property is set in the constructor, and it is used when creating a timer which calls the kill method when completed. So, when the number of seconds defined by the duration property elapses, the Explosion prefab is killed.

Notice that we have to set the "autoDestroy" property of "kill_timer" (in "this.game_state.time.create") to false, so it won't be destroyed when its event finishes. Then, in the "reset" method we add another kill event, so it will keep working when an explosion is reused from the pool.

```
1 var Bomberman = Bomberman || {};
2
3 Bomberman.Explosion = function (game_state, name, position,
properties) {
4     "use strict";
5     Bomberman.Prefab.call(this, game_state, name, position,
properties);
6
7     this.anchor.setTo(0.5);
8
9     this.duration = +properties.duration;
10
11     this.game_state.game.physics.arcade.enable(this);
12     this.body.immovable = true;
13
14     // create the kill timer with autoDestroy equals false
15     this.kill_timer = this.game_state.time.create(false);
16     this.kill_timer.add(Phaser.Timer.SECOND * this.duration,
this.kill, this);
17     this.kill_timer.start();
18 };
19
20 Bomberman.Explosion.prototype =
Object.create(Bomberman.Prefab.prototype);
21 Bomberman.Explosion.prototype.constructor =
Bomberman.Explosion;
22
23 Bomberman.Explosion.prototype.reset = function (position_x,
position_y) {
24     "use strict";
25     Phaser.Sprite.prototype.reset.call(this, position_x,
position_y);
26     // add another kill event
27     this.kill_timer.add(Phaser.Timer.SECOND * this.duration,
this.kill, this);
28 };
```

Now you can try playing again to see if the explosions are being created and killed
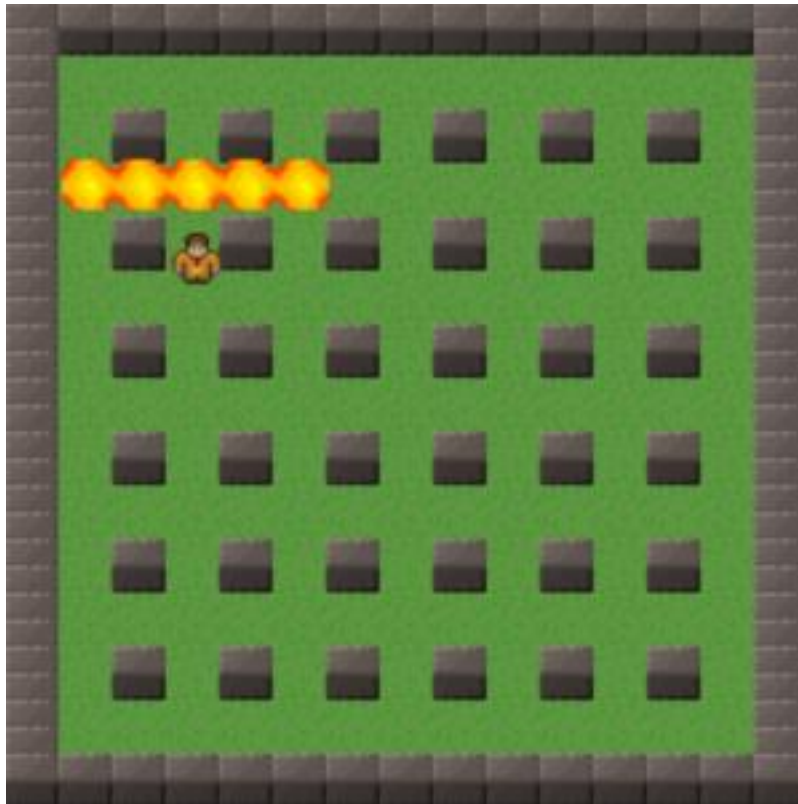
properly.

Creating the enemy

Our enemy will keep walking in a single axis (horizontal or vertical) and will switch direction when it reaches a maximum walked distance or it collides with something, like a wall or a bomb. To do this, we start by setting its walking speed, walking distance, direction and axis in the constructor. We also creates the enemy animations and start its velocity in the initial direction.

In the update method we manage the animations according to its velocity in each direction, in a similar way we did with the player. However, since it will always walk in a single direction at once, we don't have to check if it isn't already walking in another direction.

After starting or stopping animations, we check if it has reached its maximum distance, by subtracting the current position by the position in the beginning of the movement. If

this value is greater or equal than the maximum distance to walk, we have to switch its direction. The "switch_direction" method only reverts the velocity and saves the new initial position.

```
1 var Bomberman = Bomberman || {};
2
3 Bomberman.Enemy = function (game_state, name, position,
properties) {
4     "use strict";
5     Bomberman.Prefab.call(this, game_state, name, position,
properties);
6
7     this.anchor.setTo(0.5);
8
9     this.walking_speed = +properties.walking_speed;
10     this.walking_distance = +properties.walking_distance;
11     this.direction = +properties.direction;
12     this.axis = properties.axis;
13
14     this.previous_position = (this.axis === "x") ? this.x :
this.y;
15
16     this.animations.add("walking_down", [1, 2, 3], 10, true);
17     this.animations.add("walking_left", [4, 5, 6, 7], 10,
true);
18     this.animations.add("walking_right", [4, 5, 6, 7], 10,
true);
19     this.animations.add("walking_up", [0, 8, 9], 10, true);
20
21     this.stopped_frames = [1, 4, 4, 0, 1];
22
23     this.game_state.game.physics.arcade.enable(this);
24     if (this.axis === "x") {
25         this.body.velocity.x = this.direction *
this.walking_speed;
26     } else {
27         this.body.velocity.y = this.direction *
this.walking_speed;
28     }
29 };
30
31 Bomberman.Enemy.prototype =
Object.create(Bomberman.Prefab.prototype);
32 Bomberman.Enemy.prototype.constructor = Bomberman.Enemy;
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Page 163

```
33
34 Bomberman.Enemy.prototype.update = function () {
35     "use strict";
36     var new_position;
37     this.game_state.game.physics.arcade.collide(this,
this.game_state.layers.collision, this.switch_direction, null,
this);
38     this.game_state.game.physics.arcade.overlap(this,
this.game_state.groups.bombs, this.switch_direction, null,
this);
39     this.game_state.game.physics.arcade.overlap(this,
this.game_state.groups.explosions, this.kill, null, this);
40
41     if (this.body.velocity.x < 0) {
42         // walking left
43         this.scale.setTo(-1, 1);
44         this.animations.play("walking_left");
45     } else if (this.body.velocity.x > 0) {
46         // walking right
47         this.scale.setTo(1, 1);
48         this.animations.play("walking_right");
49     }
50
51     if (this.body.velocity.y < 0) {
52         // walking up
53         this.animations.play("walking_up");
54     } else if (this.body.velocity.y > 0) {
55         // walking down
56         this.animations.play("walking_down");
57     }
58
59     if (this.body.velocity.x === 0 && this.body.velocity.y ===
0) {
60         // stop current animation
61         this.animations.stop();
62         this.frame = this.stopped_frames[this.body.facing];
63     }
64
65     new_position = (this.axis === "x") ? this.x : this.y;
66     if (Math.abs(new_position - this.previous_position) >=
this.walking_distance) {
67         this.switch_direction();
68     }
69 };
70
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
71 Bomberman.Enemy.prototype.switch_direction = function () {
72     "use strict";
73     if (this.axis === "x") {
74         this.previous_position = this.x;
75         this.body.velocity.x *= -1;
76     } else {
77         this.previous_position = this.y;
78         this.body.velocity.y *= -1;
79     }
80 };
```

You can already try playing with the enemy prefab, to see if it's working. Don't forget to properly add the collisions in the prefabs update method.
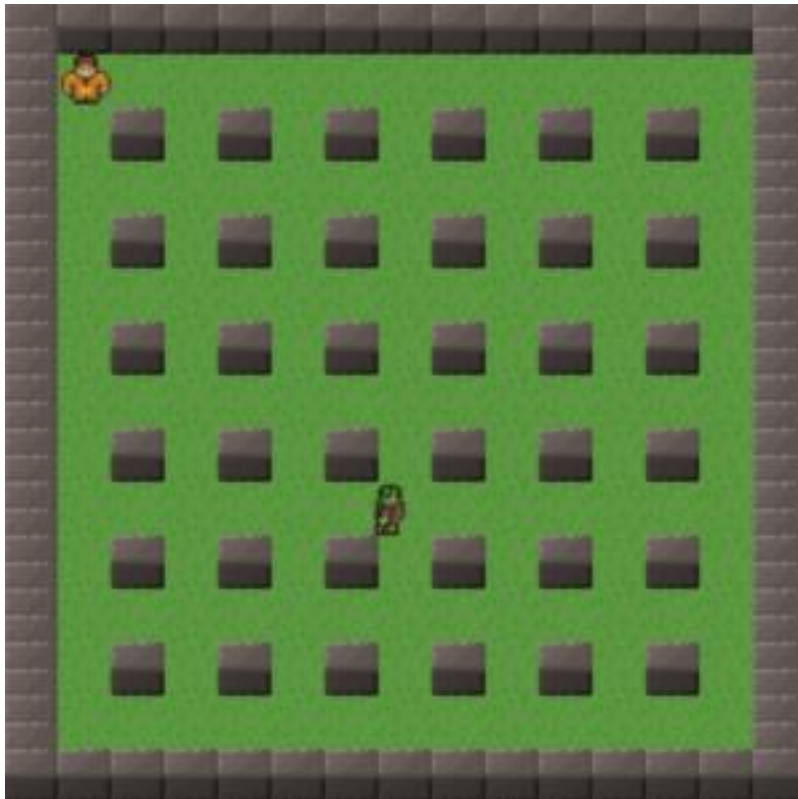
Finishing the game

And now our game is complete! In the next tutorials we will add some content to it.

How to Make a Bomberman Game in Phaser – Part 2
By Renan Oliveira

In the last tutorial we created the basic structure for a Bomberman game. In this tutorial we're going to add some more content to it and make it more fun to play, with a winning condition. In this tutorial we will add the following:

- Add targets that must be destroyed by the player to find the goal and advance levels
- Adding items, that may be found after exploding tiles
- Controlling the number of bombs the player can drop

To read this tutorial, it is important that you are familiar with the following concepts:

- Javascript and object-oriented concepts.
- Basic Phaser concepts, such as: states, sprites, groups and arcade physics
- Creating maps using Tiled map editor

Assets copyright

The assets used in this tutorial were created by Cem Kalyoncu/cemkalyoncu and Matt Hackett/richtaur and made available by "usr_share" through the creative commons license, wich allows commercial use under attribution. You can download them in http://opengameart.org/content/bomb-party-the-complete-set or by downloading the source code.

Source code files

You can download the tutorial source code files here.

New levels

In this tutorial we'll add new levels to our game. Since the focus of this tutorial is on creating the game, not using Tiled to create the maps, you're free to create your owns or use the ones provided with the source code. The figures below show the levels I'm going to use. Just remember to add the object and layer properties as explained in the last tutorial.

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools
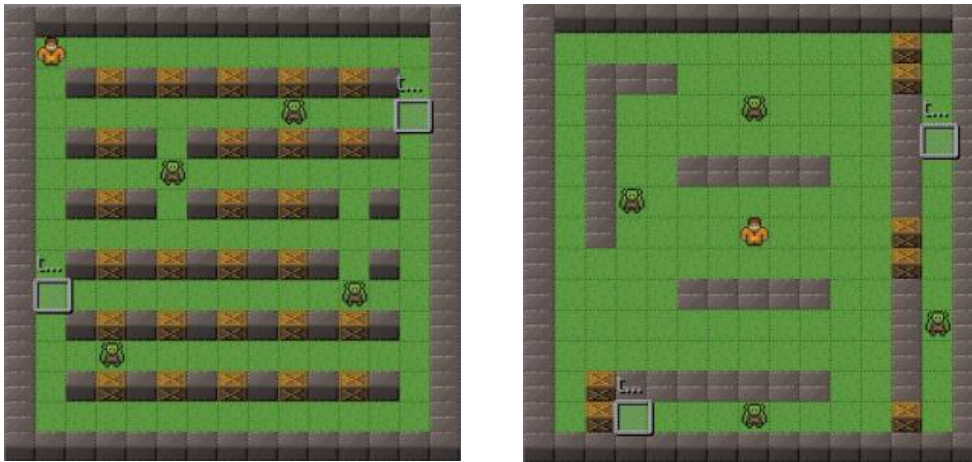
Showing the player lives

In this tutorial our player will have a starting number of lives, which will decrease every time it dies. When the number of lives reaches zero, it's game over. In addition we want to show the remaining number of lives in the screen.

First, we will modify the Player prefab to keep tracking of its number of lives. To do that, we start by adding a new property in the constructor. Notice that, since we don't want to reset the number of lives in each level, we will keep it in the localStorage when changing levels, and the player will get its value from there if available. When the first level is loaded, we clear the localStorage, so the player will reset its number of lives. The methods in TiledState that change the localStorage are "init" and "next_level", as shown below as well.

```
1 Bomberman.TiledState.prototype.init = function (level_data) {
2     "use strict";
3     var tileset_index;
4     this.level_data = level_data;
5
6     this.scale.scaleMode = Phaser.ScaleManager.SHOW_ALL;
7     this.scale.pageAlignHorizontally = true;
8     this.scale.pageAlignVertically = true;
9
10    // start physics system
11    this.game.physics.startSystem(Phaser.Physics.ARCADE);
12    this.game.physics.arcade.gravity.y = 0;
13
14    // create map and set tileset
15    this.map = this.game.add.tilemap(level_data.map.key);
16    tileset_index = 0;
17    this.map.tilesets.forEach(function (tileset) {
```

```
18          this.map.addTilesetImage(tileset.name,
level_data.map.tilesets[tileset_index]);
19          tileset_index += 1;
20      }, this);
21
22      if (this.level_data.first_level) {
23          localStorage.clear();
24      }
25 };
26
27 Bomberman.TiledState.prototype.next_level = function () {
28      "use strict";
29      localStorage.number_of_lives =
this.prefabs.player.number_of_lives;
30      localStorage.number_of_bombs =
this.prefabs.player.number_of_bombs;
31      this.game.state.start("BootState", true, false,
this.level_data.next_level, "TiledState");
32 };
```

We also have to add a "die" method in the Player prefab, which will decrease the player number of lives and check if it's still greater than 0. If so, it just resets the player to the initial position. Otherwise, it's game over. The changes in the Player prefab are shown below.

```
1 var Bomberman = Bomberman || {};
2
3 Bomberman.Player = function (game_state, name, position,
properties) {
4      "use strict";
5      Bomberman.Prefab.call(this, game_state, name, position,
properties);
6
7      this.anchor.setTo(0.5);
8
9      this.walking_speed = +properties.walking_speed;
10      this.bomb_duration = +properties.bomb_duration;
11
12      this.animations.add("walking_down", [1, 2, 3], 10, true);
13      this.animations.add("walking_left", [4, 5, 6, 7], 10,
true);
14      this.animations.add("walking_right", [4, 5, 6, 7], 10,
true);
15      this.animations.add("walking_up", [0, 8, 9], 10, true);
```

```
16
17     this.stopped_frames = [1, 4, 4, 0, 1];
18
19     this.game_state.game.physics.arcade.enable(this);
20     this.body.setSize(14, 12, 0, 4);
21
22     this.cursors =
this.game_state.game.input.keyboard.createCursorKeys();
23
24     this.initial_position = new Phaser.Point(this.x, this.y);
25
26     this.number_of_lives = localStorage.number_of_lives ||
+properties.number_of_lives;
27 };
28
29 Bomberman.Player.prototype.die = function () {
30     "use strict";
31     // decrease the number of lives
32     this.number_of_lives -= 1;
33     if (this.game_state.prefabs.lives.number_of_lives <= 0) {
34         // if there are no more lives, it's game over
35         this.game_state.game_over();
36     } else {
37         // if there are remaining lives, restart the player
position
38         this.x = this.initial_position.x;
39         this.y = this.initial_position.y;
40     }
41 };
```

To show the player number of lives in the screen we will have a Lives prefab. This prefab will show a heart image with the number of lives inside of it. To do that, we create the text in the prefab constructor with all the necessary properties and in the update method we change it to show the current number of lives of the player. The Lives prefab code is shown below.

```
1 var Bomberman = Bomberman || {};
2
3 Bomberman.Lives = function (game_state, name, position,
properties) {
4     "use strict";
5     var lives_text_position, lives_text_style,
lives_text_properties;
```

```
6    Bomberman.Prefab.call(this, game_state, name, position,
properties);
7
8    this.fixedToCamera = true;
9
10    this.anchor.setTo(0.5);
11    this.scale.setTo(0.9);
12
13    // create a text prefab to show the number of lives
14    lives_text_position = new Phaser.Point(this.position.x -
2, this.position.y + 5);
15    lives_text_style = {font: "14px Arial", fill: "#fff"};
16    lives_text_properties = {group: "hud", text:
this.number_of_lives, style: lives_text_style};
17    this.lives_text = new
Bomberman.TextPrefab(this.game_state, "lives_text",
lives_text_position, lives_text_properties);
18    this.lives_text.anchor.setTo(0.5);
19 };
20
21 Bomberman.Lives.prototype =
Object.create(Bomberman.Prefab.prototype);
22 Bomberman.Lives.prototype.constructor = Bomberman.Lives;
23
24 Bomberman.Lives.prototype.update = function () {
25    "use strict";
26    // update to show current number of lives
27    this.lives_text.text =
this.game_state.prefabs.player.number_of_lives;
28 };
```
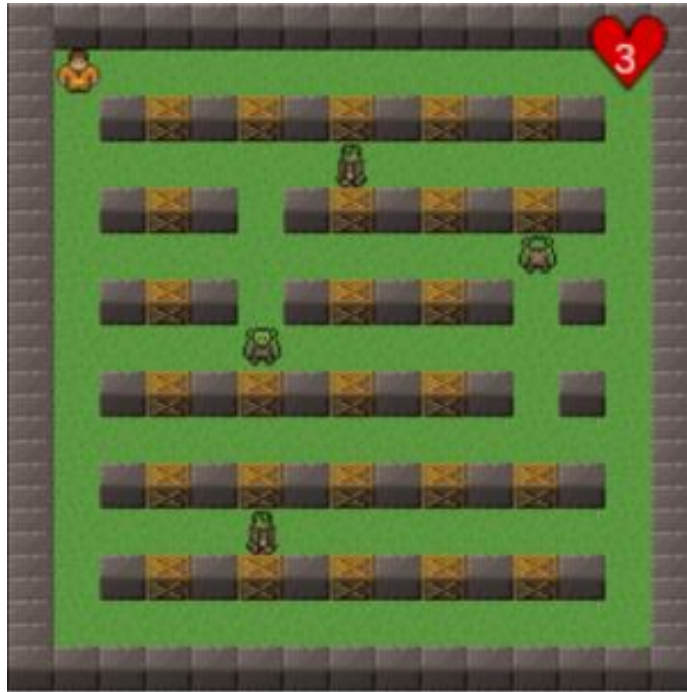
You can already try playing with the lives to check if everything is working.

Adding targets and the goal

In our game, to advance to the next level the player must destroy some targets in the level. Once all targets have been destroyed, a goal will appear in the level. When the player touches the goal, he advances to the next level.

To do that, we will create a Target prefab, which will have a physical body and is immovable. In the update method we check for overlap with explosions and, if so, we call the kill method. Finally, we overwrite the kill method to besides calling the Phaser.Sprite kill method, checking if this was the last living target. If so, we must create the goal. The code for the Target prefab is shown below:

```
1 var Bomberman = Bomberman || {};
2
3 Bomberman.Target = function (game_state, name, position,
properties) {
4     "use strict";
5     Bomberman.Prefab.call(this, game_state, name, position,
properties);
6
```

```
7     this.anchor.setTo(0.5);
8
9     this.game_state.game.physics.arcade.enable(this);
10     this.body.immovable = true;
11 };
12
13 Bomberman.Target.prototype =
Object.create(Bomberman.Prefab.prototype);
14 Bomberman.Target.prototype.constructor = Bomberman.Target;
15
16 Bomberman.Target.prototype.update = function () {
17     "use strict";
18     this.game_state.game.physics.arcade.overlap(this,
this.game_state.groups.explosions, this.kill, null, this);
19 };
20
21 Bomberman.Target.prototype.kill = function () {
22     "use strict";
23     var goal_position, goal_properties, goal;
24     Phaser.Sprite.prototype.kill.call(this);
25     if (this.game_state.groups.targets.countLiving() === 0) {
26         // create goal
27         goal_position = new
Phaser.Point(this.game_state.game.world.width / 2,
this.game_state.game.world.height / 2);
28         goal_properties = {texture: "goal_image", group:
"goals"};
29         goal = new Bomberman.Goal(this.game_state, "goal",
goal_position, goal_properties);
30     }
31 };
```

The Goal prefab is also simple, as you can see in the code below. It will have an immovable physical body too, but it will check for overlaps with the player. When the player touches the goal, it will call the "next_level" method from TiledState, which will advance the level. The "next_level" method was already shown before, since it changes the localStorage. Notice that this method uses a "next_level" property from the JSON file, so we have to add it there accordingly.

```
1 var Bomberman = Bomberman || {};
2
3 Bomberman.Goal = function (game_state, name, position,
properties) {
4     "use strict";
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Page 173

```
5     Bomberman.Prefab.call(this, game_state, name, position,
properties);
6
7     this.anchor.setTo(0.5);
8     this.scale.setTo(0.5);
9
10    this.game_state.game.physics.arcade.enable(this);
11    this.body.immovable = true;
12 };
13
14 Bomberman.Goal.prototype =
Object.create(Bomberman.Prefab.prototype);
15 Bomberman.Goal.prototype.constructor = Bomberman.Goal;
16
17 Bomberman.Goal.prototype.update = function () {
18    "use strict";
19    this.game_state.game.physics.arcade.overlap(this,
this.game_state.groups.players, this.reach_goal, null, this);
20 };
21
22 Bomberman.Goal.prototype.reach_goal = function () {
23    "use strict";
24    this.game_state.next_level();
25 };
```

You can already try playing with the targets and the goal, advancing levels and checking
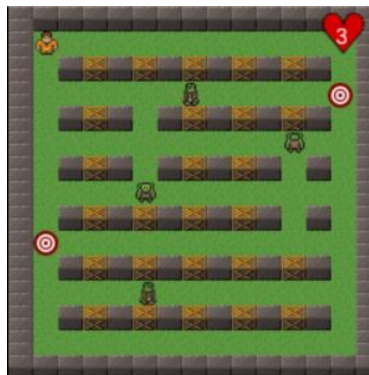
if they're working properly.

Limiting the number of bombs the player can drop

In this tutorial, we will limit the number of bombs the player can drop simultaneously. To do that we will change the way we control the bomb dropping to work in the following

way: we keep track of the total number of bombs the player can drop and the index of the current bomb. When the spacebar is pressed, we first check if it is possible to drop another bomb by comparing the index of the current bomb with the total number of bombs. If so, we drop the bomb only if it does not collide with an already dropped one. Finally, when a bomb is dropped, we have to update the index of the current bomb. The code below shows the changes in the Player prefab.

```
1 var Bomberman = Bomberman || {};
2
3 Bomberman.Player = function (game_state, name, position,
properties) {
4     "use strict";
5     Bomberman.Prefab.call(this, game_state, name, position,
properties);
6
7     this.anchor.setTo(0.5);
8
9     this.walking_speed = +properties.walking_speed;
10    this.bomb_duration = +properties.bomb_duration;
11
12    this.animations.add("walking_down", [1, 2, 3], 10, true);
13    this.animations.add("walking_left", [4, 5, 6, 7], 10,
true);
14    this.animations.add("walking_right", [4, 5, 6, 7], 10,
true);
15    this.animations.add("walking_up", [0, 8, 9], 10, true);
16
17    this.stopped_frames = [1, 4, 4, 0, 1];
18
19    this.game_state.game.physics.arcade.enable(this);
20    this.body.setSize(14, 12, 0, 4);
21
22    this.cursors =
this.game_state.game.input.keyboard.createCursorKeys();
23
24    this.initial_position = new Phaser.Point(this.x, this.y);
25
26    this.number_of_lives = localStorage.number_of_lives ||
+properties.number_of_lives;
27    this.number_of_bombs = localStorage.number_of_bombs ||
+properties.number_of_bombs;
28    this.current_bomb_index = 0;
29 };
30
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
31 Bomberman.Player.prototype.update = function () {
32     "use strict";
33     var colliding_bombs;
34
35     this.game_state.game.physics.arcade.collide(this,
this.game_state.layers.walls);
36     this.game_state.game.physics.arcade.collide(this,
this.game_state.layers.blocks);
37     this.game_state.game.physics.arcade.collide(this,
this.game_state.groups.bombs);
38     this.game_state.game.physics.arcade.overlap(this,
this.game_state.groups.explosions, this.die, null, this);
39     this.game_state.game.physics.arcade.overlap(this,
this.game_state.groups.enemies, this.die, null, this);
40
41     if (this.cursors.left.isDown && this.body.velocity.x <= 0)
{
42         // move left
43         this.body.velocity.x = -this.walking_speed;
44         if (this.body.velocity.y === 0) {
45             // change the scale, since we have only one
animation for left and right directions
46             this.scale.setTo(-1, 1);
47             this.animations.play("walking_left");
48         }
49     } else if (this.cursors.right.isDown &&
this.body.velocity.x >= 0) {
50         // move right
51         this.body.velocity.x = +this.walking_speed;
52         if (this.body.velocity.y === 0) {
53             // change the scale, since we have only one
animation for left and right directions
54             this.scale.setTo(1, 1);
55             this.animations.play("walking_right");
56         }
57     } else {
58         this.body.velocity.x = 0;
59     }
60
61     if (this.cursors.up.isDown && this.body.velocity.y <= 0) {
62         // move up
63         this.body.velocity.y = -this.walking_speed;
64         if (this.body.velocity.x === 0) {
65             this.animations.play("walking_up");
66         }
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
67      } else if (this.cursors.down.isDown &&
this.body.velocity.y >= 0) {
68          // move down
69          this.body.velocity.y = +this.walking_speed;
70          if (this.body.velocity.x === 0) {
71              this.animations.play("walking_down");
72          }
73      } else {
74          this.body.velocity.y = 0;
75      }
76
77      if (this.body.velocity.x === 0 && this.body.velocity.y ===
0) {
78          // stop current animation
79          this.animations.stop();
80          this.frame = this.stopped_frames[this.body.facing];
81      }
82
83      // if the spacebar is pressed and it is possible to drop
another bomb, try dropping it
84      if
(this.game_state.input.keyboard.isDown(Phaser.Keyboard.SPACEBAR)
&& this.current_bomb_index < this.number_of_bombs) {
85          colliding_bombs =
this.game_state.game.physics.arcade.getObjectsAtLocation(this.x,
this.y, this.game_state.groups.bombs);
86          // drop the bomb only if it does not collide with
another one
87          if (colliding_bombs.length === 0) {
88              this.drop_bomb();
89          }
90      }
91 };
92
93 Bomberman.Player.prototype.drop_bomb = function () {
94      "use strict";
95      var bomb, bomb_name, bomb_position, bomb_properties;
96      // get the first dead bomb from the pool
97      bomb_name = this.name + "_bomb_" +
this.game_state.groups.bombs.countLiving();
98      bomb_position = new Phaser.Point(this.x, this.y);
99      bomb_properties = {"texture": "bomb_spritesheet", "group":
"bombs", bomb_radius: 3};
100     bomb =
Bomberman.create_prefab_from_pool(this.game_state.groups.bombs,
```

```
Bomberman.Bomb.prototype.constructor, this.game_state,
bomb_name, bomb_position, bomb_properties);
101    this.current_bomb_index += 1;
102 };
```

In addition, we have to change the Bomb prefab to decrease the index of the current bomb when it explodes. This modification is shown in the code below.

```
1 Bomberman.Bomb.prototype.explode = function () {
2     "use strict";
3     this.kill();
4     var explosion_name, explosion_position,
explosion_properties, explosion, wall_tile, block_tile;
5     explosion_name = this.name + "_explosion_" +
this.game_state.groups.explosions.countLiving();
6     explosion_position = new Phaser.Point(this.position.x,
this.position.y);
7     explosion_properties = {texture: "explosion_image", group:
"explosions", duration: 0.5};
8     // create an explosion in the bomb position
9     explosion =
Bomberman.create_prefab_from_pool(this.game_state.groups.explosi
ons, Bomberman.Explosion.prototype.constructor, this.game_state,
10                                                        explosio
n_name, explosion_position, explosion_properties);
11
12     // create explosions in each direction
13     this.create_explosions(-1, -this.bomb_radius, -1, "x");
14     this.create_explosions(1, this.bomb_radius, +1, "x");
15     this.create_explosions(-1, -this.bomb_radius, -1, "y");
16     this.create_explosions(1, this.bomb_radius, +1, "y");
17
18     this.game_state.prefabs.player.current_bomb_index -= 1;
19 };
```

Try playing now and see if you can control the maximum number of dropped bombs. Try
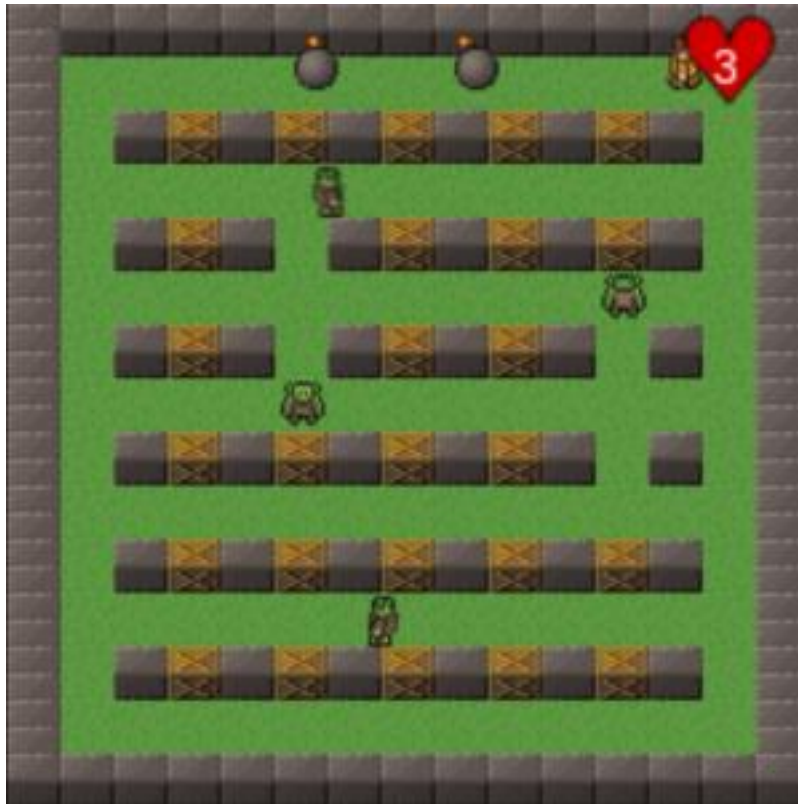
changing it and check if it works properly.

Adding items

When a tiled is destroyed by a bomb, eventually an item should appear to the player. To do that, first we will create a generic Item prefab, as shown below. This prefab will have an immovable physical body, which overlaps with explosions and the player. When it collides with an explosion, the item is destroyed, and when it collides with a player, it is collected. We also write a default "collect_item" method, which just kills it.

```
1 var Bomberman = Bomberman || {};
2
3 Bomberman.Item = function (game_state, name, position,
properties) {
4    "use strict";
5    Bomberman.Prefab.call(this, game_state, name, position,
properties);
6
7    this.anchor.setTo(0.5);
8
9    this.game_state.game.physics.arcade.enable(this);
10    this.body.immovable = true;
11
12    this.scale.setTo(0.75);
13 };
14
15 Bomberman.Item.prototype =
Object.create(Bomberman.Prefab.prototype);
16 Bomberman.Item.prototype.constructor = Bomberman.Item;
17
18 Bomberman.Item.prototype.update = function () {
19    "use strict";
20    this.game_state.game.physics.arcade.overlap(this,
this.game_state.groups.players, this.collect_item, null, this);
21 };
22
23 Bomberman.Item.prototype.collect_item = function () {
24    "use strict";
25    // by default, an item is destroyed when collected
26    this.kill();
27 };
```

Now, we are going to create other prefabs that will extend Item and will overwrite the "collect_item" method. We are going to create two items:
1) Life item, which will increase the player number of lives
2) Bomb item, which will increase the number of bombs the player can drop

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

The LifeItem prefab code is shown below. Since we already have our generic Item prefab, we only have to overwrite the "collect_item" method, which increase the player number of lives.

```
1 var Bomberman = Bomberman || {};
2
3 Bomberman.LifeItem = function (game_state, name, position,
properties) {
4    "use strict";
5    Bomberman.Item.call(this, game_state, name, position,
properties);
6 };
7
8 Bomberman.LifeItem.prototype =
Object.create(Bomberman.Item.prototype);
9 Bomberman.LifeItem.prototype.constructor = Bomberman.LifeItem;
10
11 Bomberman.LifeItem.prototype.collect_item = function (item,
player) {
12    "use strict";
13    Bomberman.Item.prototype.collect_item.call(this);
14    player.number_of_lives += 1;
15 };
```

The BombItem prefab code is similar and shown below. In the "collect_item" method we only have to increase the player number of bombs, limited by a maximum number.

```
1 var Bomberman = Bomberman || {};
2
3 Bomberman.BombItem = function (game_state, name, position,
properties) {
4    "use strict";
5    Bomberman.Item.call(this, game_state, name, position,
properties);
6
7    this.MAXIMUM_NUMBER_OF_BOMBS = 5;
8 };
9
10 Bomberman.BombItem.prototype =
Object.create(Bomberman.Item.prototype);
11 Bomberman.BombItem.prototype.constructor =
Bomberman.BombItem;
12
```

```
13 Bomberman.BombItem.prototype.collect_item = function (item,
player) {
14     "use strict";
15     Bomberman.Item.prototype.collect_item.call(this);
16     // increases the player number of bombs, limited by a
maximum
17     player.number_of_bombs = Math.min(player.number_of_bombs +
1, this.MAXIMUM_NUMBER_OF_BOMBS);
18 };
```

Finally, we have to create the items when a tile is destroyed. First, we will add a property in our TiledState containing all the items probabilities and properties, as shown below. One important thing is that the items are ordered by their probabilities, with the lower probability item being the first one.

```
1 Bomberman.TiledState = function () {
2     "use strict";
3     Phaser.State.call(this);
4
5     this.prefab_classes = {
6         "player": Bomberman.Player.prototype.constructor,
7         "enemy": Bomberman.Enemy.prototype.constructor,
8         "target": Bomberman.Target.prototype.constructor,
9         "life_item": Bomberman.LifeItem.prototype.constructor,
10        "bomb_item": Bomberman.BombItem.prototype.constructor
11    };
12
13    // define available items
14    this.items = {
15        life_item: {probability: 0.1, properties: {texture:
"life_item_image", group: "items"}},
16        bomb_item: {probability: 0.3, properties: {texture:
"bomb_item_image", group: "items"}}
17    };
18 };
```

Now, in the Bomb prefab, after we destroy a tile we have to check if a item must be created. To do that, we generate a random item using Phaser random data generator (for more information, you can check the documentation), and iterate through all the available items comparing the generated number with the item probability. If the generated number is lower than the probability of an item, we create it and stop the loop. Since the items are ordered by probability, the less likely ones will have priority. The Bomb prefab

modifications are shown below. Notice that we call the "check_item" method at the end of "create_explosions", and we use a pool of objects to create the items.

```
1 Bomberman.Bomb.prototype.create_explosions = function
(initial_index, final_index, step, axis) {
2     "use strict";
3     var index, explosion_name, explosion_position, explosion,
explosion_properties, wall_tile, block_tile;
4     explosion_properties = {texture: "explosion_image", group:
"explosions", duration: 0.5};
5     for (index = initial_index; Math.abs(index) <=
Math.abs(final_index); index += step) {
6         explosion_name = this.name + "_explosion_" +
this.game_state.groups.explosions.countLiving();
7         // the position is different accoring to the axis
8         if (axis === "x") {
9             explosion_position = new
Phaser.Point(this.position.x + (index * this.width),
this.position.y);
10         } else {
11             explosion_position = new
Phaser.Point(this.position.x, this.position.y + (index *
this.height));
12         }
13         wall_tile =
this.game_state.map.getTileWorldXY(explosion_position.x,
explosion_position.y, this.game_state.map.tileWidth,
this.game_state.map.tileHeight, "walls");
14         block_tile =
this.game_state.map.getTileWorldXY(explosion_position.x,
explosion_position.y, this.game_state.map.tileWidth,
this.game_state.map.tileHeight, "blocks");
15         if (!wall_tile && !block_tile) {
16             // create a new explosion in the new position
17             explosion = 18
Bomberman.create_prefab_from_pool(this.game_state.groups.explosi
ons, Bomberman.Explosion.prototype.constructor, this.game_state,
explosion_name, explosion_position, explosion_properties);
18         } else {
19             if (block_tile) {
20                 // check for item to spawn
21                 this.check_for_item({x: block_tile.x *
block_tile.width, y: block_tile.y * block_tile.height},
```

```
22                                       {x: block_tile.width, y:
block_tile.height});
23                   this.game_state.map.removeTile(block_tile.x,
block_tile.y, "blocks");
24               }
25           break;
26        }
27     }
28 };
29
30 Bomberman.Bomb.prototype.check_for_item = function
(block_position, block_size) {
31     "use strict";
32     var random_number, item_prefab_name, item,
item_probability, item_name, item_position, item_properties,
item_constructor, item_prefab;
33     random_number = this.game_state.game.rnd.frac();
34     // search for the first item that can be spawned
35     for (item_prefab_name in this.game_state.items) {
36         if
(this.game_state.items.hasOwnProperty(item_prefab_name)) {
37             item = this.game_state.items[item_prefab_name];
38             item_probability = item.probability;
39             // spawns an item if the random number is less
than the item probability
40             if (random_number < item_probability) {
41                 item_name = this.name + "_items_" +
this.game_state.groups[item.properties.group].countLiving();
42                 item_position = new
Phaser.Point(block_position.x + (block_size.x / 2),
block_position.y + (block_size.y / 2));
43                 console.log(item_position);
44                 item_properties = item.properties;
45                 item_constructor =
this.game_state.prefab_classes[item_prefab_name];
46                 item_prefab =
Bomberman.create_prefab_from_pool(this.game_state.groups.items,
item_constructor, this.game_state, item_name, item_position,
item_properties);
47                 break;
48             }
49         }
50     }
51 };
```

Now you can try playing to see if the items are being created correctly, and if they're
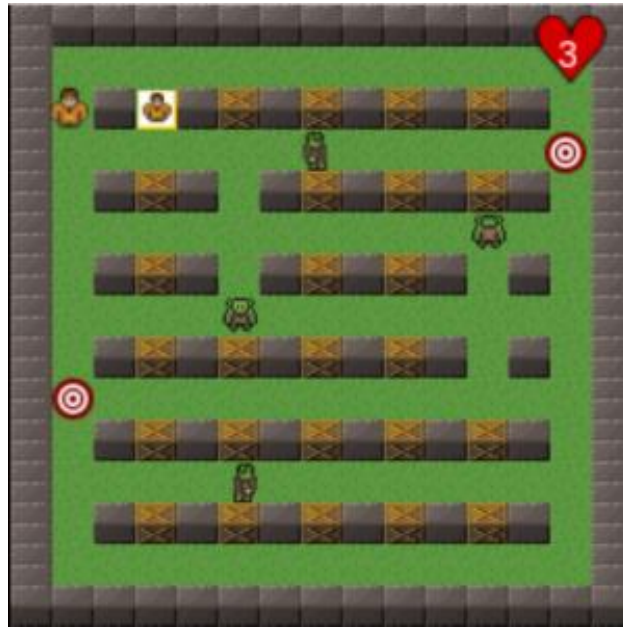
having the correct effect in the game.

Finishing the game

And now we finished this tutorial! We added some nice content making it more fun to play. In the next tutorial we will make it multiplayer, adding another player to it! Let me know your opinion of this tutorial and what you would like to see in future ones.

How to Make a Bomberman Game in Phaser – Part 3
By Renan Oliveira

In the last tutorial, we added content to our Bomberman game, such as lives, items and more levels. In this tutorial, we will make it multiplayer, by adding a second player. We will also add a battle game mode, where the two players must compete agains each other. The following topics will be covered:

- Creating a Phaser plugin to receive user input
- Making the game multiplayer
- Creating a new game mode
- Creating a title screen with the two game mode options

To read this tutorial, it is important that you are familiar with the following concepts:

- Javascript and object-oriented concepts.
- Basic Phaser concepts, such as: states, sprites, groups and arcade physics
- Creating maps using Tiled map editor

Assets copyright

The assets used in this tutorial were created by Cem Kalyoncu/cemkalyoncu and Matt Hackett/richtaur and made available by "usr_share" through the creative commons license, which allows commercial use under attribution. You can download them in http://opengameart.org/content/bomb-party-the-complete-set or by downloading the source code.

Source code files

You can download the tutorial source code files here.

Creating a Phaser plugin to receive user input

In Phaser, you can create plugins to add functionalities to your engine (you can check the documentation for more information). In this tutorial, we will create a Phaser plugin that will read a JSON file containing all the user input data. This JSON file must have the key for each action, and what action should be executed for each key.
The JSON file below shows the user input data I used for this tutorial. You can create your own by changing the key for each action as you prefer. In this file we define a list of

inputs for each possible key event (keyDown, keyUp, keyPress). For each user input we define the key, the callback function and the arguments. For example, in the keyDown events, the key "UP" will call "player.move" with the arguments "[0, -1]" and "true" (we will define this callback function later).

```
1  {
2      "keydown": {
3          "UP": {
4                "callback": "player.change_movement",
5                "args": [0, -1, true]
6          },
7          "DOWN": {
8                "callback": "player.change_movement",
9                "args": [0, 1, true]
10         },
11          "LEFT": {
12                "callback": "player.change_movement",
13                "args": [-1, 0, true]
14         },
15          "RIGHT": {
16                "callback": "player.change_movement",
17                "args": [1, 0, true]
18         },
19          "SPACEBAR": {
20                "callback": "player.try_dropping_bomb"
21         }
22      },
23      "keyup": {
24          "UP": {
25                "callback": "player.change_movement",
26                "args": [0, -1, false]
27         },
28          "DOWN": {
29                "callback": "player.change_movement",
30                "args": [0, 1, false]
31         },
32          "LEFT": {
33                "callback": "player.change_movement",
34                "args": [-1, 0, false]
35         },
36          "RIGHT": {
37                "callback": "player.change_movement",
38                "args": [1, 0, false]
39         }
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
40      }
41 }
```

Now, we have to write our plugin code. We start by creating a class that extends Phaser.Plugin. The Phaser.Plugin class has a method "init" which we must use as our constructor. This method will receive the user input data as parameter, and will create an object mapping, for each event type, each key to its callback function. The callback function must define the prefab and its method that will be called. Then, we add an event for each possible key event, calling the method "process_input".

```
1 var Phaser = Phaser || {};
2 var Bomberman = Bomberman || {};
3
4 Bomberman.UserInput = function (game, parent, game_state,
user_input_data) {
5     "use strict";
6     Phaser.Plugin.call(this, game, parent);
7 };
8
9 Bomberman.UserInput.prototype =
Object.create(Phaser.Plugin.prototype);
10 Bomberman.UserInput.prototype.constructor =
Bomberman.UserInput;
11
12 Bomberman.UserInput.prototype.init = function (game_state,
user_input_data) {
13     "use strict";
14     var input_type, key, key_code;
15     this.game_state = game_state;
16     this.user_inputs = {"keydown": {}, "keyup": {},
"keypress": {}};
17
18     // instantiate object with user input data provided
19     // each event can be keydown, keyup or keypress
20     // separate events by key code
21     for (input_type in user_input_data) {
22         if (user_input_data.hasOwnProperty(input_type)) {
23             for (key in user_input_data[input_type]) {
24                 if
(user_input_data[input_type].hasOwnProperty(key)) {
25                     key_code = Phaser.Keyboard[key];
26                     this.user_inputs[input_type][key_code] =
user_input_data[input_type][key];
27                 }
```

```
28                    }
29                }
30         }
31
32      // add callback for all three events
33      this.game.input.keyboard.addCallbacks(this,
this.process_input, this.process_input, this.process_input);
34 };
35
36 Bomberman.UserInput.prototype.process_input = function
(event) {
37      "use strict";
38      var user_input, callback_data, prefab;
39      if (this.user_inputs[event.type] &&
this.user_inputs[event.type][event.keyCode]) {
40          user_input =
this.user_inputs[event.type][event.keyCode];
41          if (user_input) {
42              callback_data = user_input.callback.split(".");
43              // identify prefab
44              prefab =
this.game_state.prefabs[callback_data[0]];
45              // call correct method
46              prefab[callback_data[1]].apply(prefab,
user_input.args);
47          }
48      }
49 };
```

The method "process_input" identifies the event type and key and gets the callback information. From the callback, it uses Javascript split function to identify the prefab and the method and call it passing the arguments defined in the user input file.

Now, we have to define the "move" and "try_dropping_bomb" methods in the Player prefab. The first method will receive the direction and if the player should start or stop moving. By doing so, we can use the same method to move or stop the player in each direction, as you can see in the user input file. The player prefab sets an object move according to the given direction. Then, in the Player prefab update method, instead of checking for pressed keys, we only check if the move object is true for each direction. For the "try_dropping_bomb" method we just moved the piece of code responsible for dropping bombs from the "update" method. The code below shows the necessary modifications in the Player prefab.

```
1 var Bomberman = Bomberman || {};
2
3 Bomberman.Player = function (game_state, name, position,
properties) {
4     "use strict";
5     Bomberman.Prefab.call(this, game_state, name, position,
properties);
6
7     this.anchor.setTo(0.5);
8
9     this.walking_speed = +properties.walking_speed;
10    this.bomb_duration = +properties.bomb_duration;
11
12    this.animations.add("walking_down", [1, 2, 3], 10, true);
13    this.animations.add("walking_left", [4, 5, 6, 7], 10,
true);
14    this.animations.add("walking_right", [4, 5, 6, 7], 10,
true);
15    this.animations.add("walking_up", [0, 8, 9], 10, true);
16
17    this.stopped_frames = [1, 4, 4, 0, 1];
18
19    this.game_state.game.physics.arcade.enable(this);
20    this.body.setSize(14, 12, 0, 4);
21
22    this.initial_position = new Phaser.Point(this.x, this.y);
23
24    this.number_of_lives = localStorage.number_of_lives ||
+properties.number_of_lives;
25    this.number_of_bombs = localStorage.number_of_bombs ||
+properties.number_of_bombs;
26    this.current_bomb_index = 0;
27
28    this.movement = {left: false, right: false, up: false,
down: false};
29 };
30
31 Bomberman.Player.prototype =
Object.create(Bomberman.Prefab.prototype);
32 Bomberman.Player.prototype.constructor = Bomberman.Player;
33
34 Bomberman.Player.prototype.update = function () {
35    "use strict";
36    this.game_state.game.physics.arcade.collide(this,
this.game_state.layers.walls);
```

```
37      this.game_state.game.physics.arcade.collide(this,
this.game_state.layers.blocks);
38      this.game_state.game.physics.arcade.collide(this,
this.game_state.groups.bombs);
39      this.game_state.game.physics.arcade.overlap(this,
this.game_state.groups.explosions, this.die, null, this);
40      this.game_state.game.physics.arcade.overlap(this,
this.game_state.groups.enemies, this.die, null, this);
41
42      if (this.movement.left && this.body.velocity.x <= 0) {
43          // move left
44          this.body.velocity.x = -this.walking_speed;
45          if (this.body.velocity.y === 0) {
46              // change the scale, since we have only one
animation for left and right directions
47              this.scale.setTo(-1, 1);
48              this.animations.play("walking_left");
49          }
50      } else if (this.movement.right && this.body.velocity.x >=
0) {
51          // move right
52          this.body.velocity.x = +this.walking_speed;
53          if (this.body.velocity.y === 0) {
54              // change the scale, since we have only one
animation for left and right directions
55              this.scale.setTo(1, 1);
56              this.animations.play("walking_right");
57          }
58      } else {
59          this.body.velocity.x = 0;
60      }
61
62      if (this.movement.up && this.body.velocity.y <= 0) {
63          // move up
64          this.body.velocity.y = -this.walking_speed;
65          if (this.body.velocity.x === 0) {
66              this.animations.play("walking_up");
67          }
68      } else if (this.movement.down && this.body.velocity.y >=
0) {
69          // move down
70          this.body.velocity.y = +this.walking_speed;
71          if (this.body.velocity.x === 0) {
72              this.animations.play("walking_down");
73          }
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
74      } else {
75          this.body.velocity.y = 0;
76      }
77
78      if (this.body.velocity.x === 0 && this.body.velocity.y ===
0) {
79          // stop current animation
80          this.animations.stop();
81          this.frame = this.stopped_frames[this.body.facing];
82      }
83  };
84
85  Bomberman.Player.prototype.change_movement = function
(direction_x, direction_y, move) {
86      "use strict";
87      if (direction_x < 0) {
88          this.movement.left = move;
89      } else if (direction_x > 0) {
90          this.movement.right = move;
91      }
92
93      if (direction_y < 0) {
94          this.movement.up = move;
95      } else if (direction_y > 0) {
96          this.movement.down = move;
97      }
98  };
99
100 Bomberman.Player.prototype.try_dropping_bomb = function () {
101     "use strict";
102     var colliding_bombs;
103     // if it is possible to drop another bomb, try dropping
it
104     if (this.current_bomb_index < this.number_of_bombs) {
105         colliding_bombs =
this.game_state.game.physics.arcade.getObjectsAtLocation(this.x,
this.y, this.game_state.groups.bombs);
106         // drop the bomb only if it does not collide with
another one
107         if (colliding_bombs.length === 0) {
108             this.drop_bomb();
109         }
110     }
111 };
```

Finally, we have to add our plugin in the game state. First, we have to change the JSON level file to tell the LoadingState which file contains the user input, as shown below.

```
1  {
2      "assets": {
3          "map_tiles": {"type": "image", "source":
"assets/images/bomberman_spritesheet.png"},
4          "player_spritesheet": {"type": "spritesheet", "source":
"assets/images/player_spritesheet.png", "frame_width": 16,
"frame_height": 16},
5          "bomb_spritesheet": {"type": "spritesheet", "source":
"assets/images/bomb_spritesheet.png", "frame_width": 16,
"frame_height": 16},
6          "enemy_spritesheet": {"type": "spritesheet", "source":
"assets/images/enemy_spritesheet.png", "frame_width": 16,
"frame_height": 16},
7          "explosion_image": {"type": "image", "source":
"assets/images/explosion.png"},
8          "heart_image": {"type": "image", "source":
"assets/images/heart.png"},
9          "target_image": {"type": "image", "source":
"assets/images/target.png"},
10         "goal_image": {"type": "image", "source":
"assets/images/portal.png"},
11         "life_item_image": {"type": "image", "source":
"assets/images/life_item.png"},
12         "bomb_item_image": {"type": "image", "source":
"assets/images/bomb_item.png"},
13
14         "level_tilemap": {"type": "tilemap", "source":
"assets/maps/level1_map.json"}
15      },
16      "groups": [
17          "targets",
18          "items",
19          "explosions",
20          "bombs",
21          "goals",
22          "enemies",
23          "players",
24          "hud"
25      ],
26      "map": {
27          "key": "level_tilemap",
```

```
28          "tilesets": ["map_tiles"]
29       },
30      "user_input": "assets/levels/user_input.json",
31      "next_level": "assets/levels/level2.json",
32      "first_level": true
33 }
```

Now, we have to change the LoadingState to load the user input file and add the plugin in the "create" method of TiledState. The code below show the modifications in both classes.

```
1 Bomberman.LoadingState.prototype.preload = function () {
2     "use strict";
3     var assets, asset_loader, asset_key, asset;
4     assets = this.level_data.assets;
5     for (asset_key in assets) { // load assets according to
asset key
6          if (assets.hasOwnProperty(asset_key)) {
7              asset = assets[asset_key];
8              switch (asset.type) {
9              case "image":
10                 this.load.image(asset_key, asset.source);
11                 break;
12              case "spritesheet":
13                 this.load.spritesheet(asset_key, asset.source,
asset.frame_width, asset.frame_height, asset.frames,
asset.margin, asset.spacing);
14                 break;
15              case "tilemap":
16                 this.load.tilemap(asset_key, asset.source,
null, Phaser.Tilemap.TILED_JSON);
17                 break;
18             }
19          }
20     }
21     // load user input file
22     if (this.level_data.user_input) {
23         this.load.text("user_input",
this.level_data.user_input);
24     }
25 };
```

```
1 Bomberman.TiledState.prototype.create = function () {
2     "use strict";
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
3     var group_name, object_layer, collision_tiles;
4
5     // create map layers
6     this.layers = {};
7     this.map.layers.forEach(function (layer) {
8         this.layers[layer.name] =
this.map.createLayer(layer.name);
9         if (layer.properties.collision) { // collision layer
10            collision_tiles = [];
11            layer.data.forEach(function (data_row) { // find
tiles used in the layer
12                data_row.forEach(function (tile) {
13                    // check if it's a valid tile index and
isn't already in the list
14                    if (tile.index > 0 &&
collision_tiles.indexOf(tile.index) === -1) {
15                        collision_tiles.push(tile.index);
16                    }
17                }, this);
18            }, this);
19            this.map.setCollision(collision_tiles, true,
layer.name);
20        }
21    }, this);
22    // resize the world to be the size of the current layer
23    this.layers[this.map.layer.name].resizeWorld();
24
25    // create groups
26    this.groups = {};
27    this.level_data.groups.forEach(function (group_name) {
28        this.groups[group_name] = this.game.add.group();
29    }, this);
30
31    this.prefabs = {};
32
33    for (object_layer in this.map.objects) {
34        if (this.map.objects.hasOwnProperty(object_layer)) {
35            // create layer objects
36            this.map.objects[object_layer].forEach(this.create
_object, this);
37        }
38    }
39
```

```
40    this.game.user_input =
this.game.plugins.add(Bomberman.UserInput, this,
JSON.parse(this.game.cache.getText("user_input")));
41
42    this.init_hud();
43 };
```

You can already try playing your game with the user input file. Nothing should change in the gameplay, the only difference is in the code.

Adding a new player

Now that we're handling all the user input with our plugin, it becomes easier to add another player. We just have to add another Player prefab in our game and add its correspondent user input in the user input file. The new user input file is shown below. Notice that the prefab "player" becomes "player1" and we add the user input for "player2" as well.

```
1  {
2      "keydown": {
3          "W": {
4              "callback": "player1.change_movement",
5              "args": [0, -1, true]
6          },
7          "S": {
8              "callback": "player1.change_movement",
9              "args": [0, 1, true]
10         },
11         "A": {
12             "callback": "player1.change_movement",
13             "args": [-1, 0, true]
14         },
15         "D": {
16             "callback": "player1.change_movement",
17             "args": [1, 0, true]
18         },
19         "Z": {
20             "callback": "player1.try_dropping_bomb"
21         },
22         "U": {
23             "callback": "player2.change_movement",
24             "args": [0, -1, true]
25         },
26         "J": {
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Page 197

```
27                  "callback": "player2.change_movement",
28                  "args": [0, 1, true]
29          },
30          "H": {
31                  "callback": "player2.change_movement",
32                  "args": [-1, 0, true]
33          },
34          "K": {
35                  "callback": "player2.change_movement",
36                  "args": [1, 0, true]
37          },
38          "N": {
39                  "callback": "player2.try_dropping_bomb"
40          }
41      },
42      "keyup": {
43          "W": {
44                  "callback": "player1.change_movement",
45                  "args": [0, -1, false]
46          },
47          "S": {
48                  "callback": "player1.change_movement",
49                  "args": [0, 1, false]
50          },
51          "A": {
52                  "callback": "player1.change_movement",
53                  "args": [-1, 0, false]
54          },
55          "D": {
56                  "callback": "player1.change_movement",
57                  "args": [1, 0, false]
58          },
59          "U": {
60                  "callback": "player2.change_movement",
61                  "args": [0, -1, false]
62          },
63          "J": {
64                  "callback": "player2.change_movement",
65                  "args": [0, 1, false]
66          },
67          "H": {
68                  "callback": "player2.change_movement",
69                  "args": [-1, 0, false]
70          },
71          "K": {
```

```
72            "callback": "player2.change_movement",
73            "args": [1, 0, false]
74        }
75    }
76 }
```

We also need to update the maps to add the new player. Those are the maps I created.
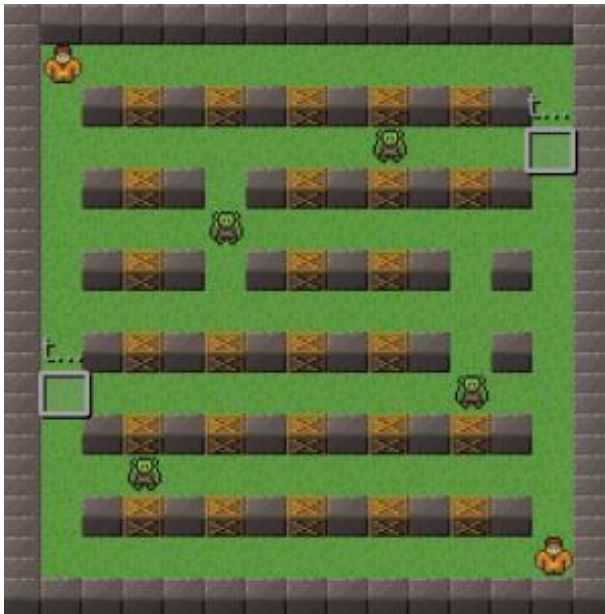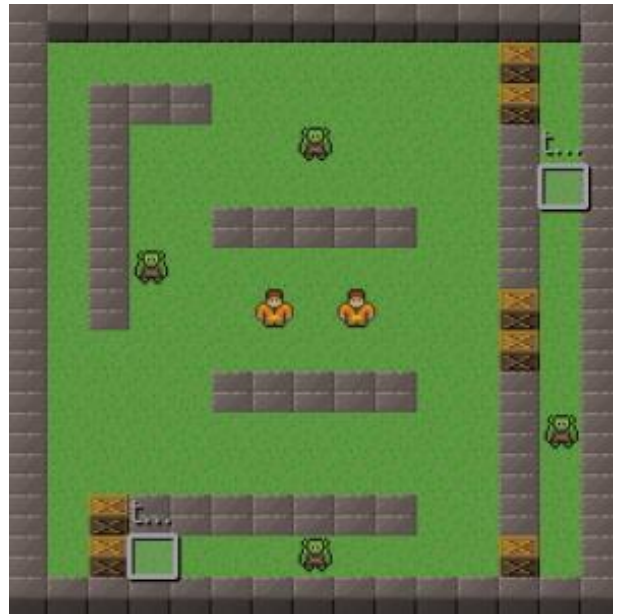
You can use them, which are available in the source code, or create your own.

There are some other things we must change so our game still works. First, we must change our Lives prefab so it knows which player lives it should show. We do this by

adding a property with the player prefab name, so in the "update" method it can get the correct Player prefab and update the number of lives correctly. We also change the "init_hud" method in TiledState to show the number of lives of each player. The codes below show the modifications in the Lives prefab and TiledState.

```
1 var Bomberman = Bomberman || {};
2
3 Bomberman.Lives = function (game_state, name, position,
properties) {
4     "use strict";
5     var lives_text_position, lives_text_style,
lives_text_properties;
6     Bomberman.Prefab.call(this, game_state, name, position,
properties);
7
8     this.player = properties.player;
9
10     this.fixedToCamera = true;
11
12    this.anchor.setTo(0.5);
13    this.scale.setTo(0.6);
14
15    // create a text prefab to show the number of lives
16    lives_text_position = new Phaser.Point(this.position.x -
2, this.position.y + 5);
17    lives_text_style = {font: "10px Arial", fill: "#fff"};
18    lives_text_properties = {group: "hud", text:
this.number_of_lives, style: lives_text_style};
19    this.lives_text = new
Bomberman.TextPrefab(this.game_state, "lives_text",
lives_text_position, lives_text_properties);
20    this.lives_text.anchor.setTo(0.5);
21 };
22
23 Bomberman.Lives.prototype =
Object.create(Bomberman.Prefab.prototype);
24 Bomberman.Lives.prototype.constructor = Bomberman.Lives;
25
26 Bomberman.Lives.prototype.update = function () {
27    "use strict";
28    // update to show current number of lives
29    this.lives_text.text =
this.game_state.prefabs[this.player].number_of_lives;
30 };
```

```
1 Bomberman.TiledState.prototype.init_hud = function () {
2     "use strict";
3     var player1_lives_position, player1_lives_properties,
player1_lives, player2_lives_position, player2_lives_properties,
player2_lives;
4
5     // create the lives prefab for player1
6     player1_lives_position = new Phaser.Point(0.1 *
this.game.world.width, 0.07 * this.game.world.height);
7     player1_lives_properties = {group: "hud", texture:
"heart_image", number_of_lives: 3, player: "player1"};
8     player1_lives = new Bomberman.Lives(this, "lives",
player1_lives_position, player1_lives_properties);
9
10    // create the lives prefab for player2
11    player2_lives_position = new Phaser.Point(0.9 *
this.game.world.width, 0.07 * this.game.world.height);
12    player2_lives_properties = {group: "hud", texture:
"heart_image", number_of_lives: 3, player: "player2"};
13    player2_lives = new Bomberman.Lives(this, "lives",
player2_lives_position, player2_lives_properties);
14 };
```

Also, when a bomb explodes, it decreases the current bomb index of its owner. Since now we have two possible owners, the bomb must know which one it should decrease the current bomb index. We can easily do that by adding a property with the bomb owner, as the code below shows. Then, when a player creates a bomb, it passes its object as a parameter to the bomb. This modifications are also shown below.

```
1 var Bomberman = Bomberman || {};
2
3 Bomberman.Bomb = function (game_state, name, position,
properties) {
4     "use strict";
5     Bomberman.Prefab.call(this, game_state, name, position,
properties);
6
7     this.anchor.setTo(0.5);
8
9     this.bomb_radius = +properties.bomb_radius;
10
11    this.game_state.game.physics.arcade.enable(this);
12    this.body.immovable = true;
13
```

```
14    this.exploding_animation =
this.animations.add("exploding", [0, 2, 4], 1, false);
15    this.exploding_animation.onComplete.add(this.explode,
this);
16    this.animations.play("exploding");
17
18    this.owner = properties.owner;
19 };
20
21 Bomberman.Bomb.prototype =
Object.create(Bomberman.Prefab.prototype);
22 Bomberman.Bomb.prototype.constructor = Bomberman.Bomb;
23
24 Bomberman.Bomb.prototype.explode = function () {
25    "use strict";
26    this.kill();
27    var explosion_name, explosion_position,
explosion_properties, explosion, wall_tile, block_tile;
28    explosion_name = this.name + "_explosion_" +
this.game_state.groups.explosions.countLiving();
29    explosion_position = new Phaser.Point(this.position.x,
this.position.y);
30    explosion_properties = {texture: "explosion_image", group:
"explosions", duration: 0.5};
31    // create an explosion in the bomb position
32    explosion =
Bomberman.create_prefab_from_pool(this.game_state.groups.explosi
ons, Bomberman.Explosion.prototype.constructor, this.game_state,
33                                          explosio
n_name, explosion_position, explosion_properties);
34
35    // create explosions in each direction
36    this.create_explosions(-1, -this.bomb_radius, -1, "x");
37    this.create_explosions(1, this.bomb_radius, +1, "x");
38    this.create_explosions(-1, -this.bomb_radius, -1, "y");
39    this.create_explosions(1, this.bomb_radius, +1, "y");
40
41    this.owner.current_bomb_index -= 1;
42 };


1 Bomberman.Player.prototype.drop_bomb = function () {
2    "use strict";
3    var bomb, bomb_name, bomb_position, bomb_properties;
4    // get the first dead bomb from the pool
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
5    bomb_name = this.name + "_bomb_" +
this.game_state.groups.bombs.countLiving();
6    bomb_position = new Phaser.Point(this.x, this.y);
7    bomb_properties = {"texture": "bomb_spritesheet", "group":
"bombs", bomb_radius: 3, owner: this};
8    bomb =
Bomberman.create_prefab_from_pool(this.game_state.groups.bombs,
Bomberman.Bomb.prototype.constructor, this.game_state,
bomb_name, bomb_position, bomb_properties);
9    this.current_bomb_index += 1;
10 };
```
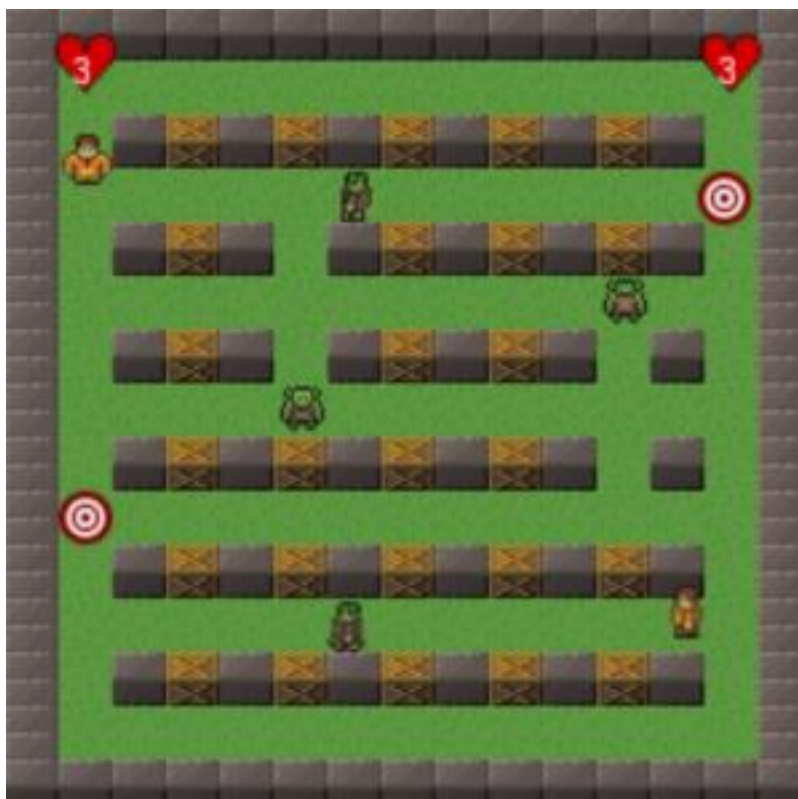

Now, you can already try playing with two players, and see if it is working.
Adding the battle mode

We will add a battle mode where each player should explode the other to win. To do that, we must create a new game state, called BattleState. However, this state will be very similar to the TiledState we already have. So, we will change the TiledState to have only the common code between the BattleState and our current game mode. Then, we will

create the new states, called ClassicState and BattleState, which will extend TiledState, only adding the differences.

The code for TiledState, ClassicState and BattleState is shown below. Notice that the only differences are the "game_over" method, which will show the winner in the BattleState, and the "next_level" method, which exists only for the ClassicState. Remember that we must change all references to "TiledState" in our code to be either ClassicState or BattleState.

```
1 var Bomberman = Bomberman || {};
2
3 Bomberman.TiledState = function () {
4     "use strict";
5     Phaser.State.call(this);
6
7     this.prefab_classes = {
8         "player": Bomberman.Player.prototype.constructor,
9         "enemy": Bomberman.Enemy.prototype.constructor,
10         "target": Bomberman.Target.prototype.constructor,
11         "life_item": Bomberman.LifeItem.prototype.constructor,
12         "bomb_item": Bomberman.BombItem.prototype.constructor
13     };
14
15     // define available items
16     this.items = {
17         life_item: {probability: 0.1, properties: {texture:
"life_item_image", group: "items"}},
18         bomb_item: {probability: 0.3, properties: {texture:
"bomb_item_image", group: "items"}}
19     };
20 };
21
22 Bomberman.TiledState.prototype =
Object.create(Phaser.State.prototype);
23 Bomberman.TiledState.prototype.constructor =
Bomberman.TiledState;
24
25 Bomberman.TiledState.prototype.init = function (level_data) {
26     "use strict";
27     var tileset_index;
28     this.level_data = level_data;
29
30     this.scale.scaleMode = Phaser.ScaleManager.SHOW_ALL;
31     this.scale.pageAlignHorizontally = true;
```

```
32     this.scale.pageAlignVertically = true;
33
34     // start physics system
35     this.game.physics.startSystem(Phaser.Physics.ARCADE);
36     this.game.physics.arcade.gravity.y = 0;
37
38     // create map and set tileset
39     this.map = this.game.add.tilemap(level_data.map.key);
40     tileset_index = 0;
41     this.map.tilesets.forEach(function (tileset) {
42         this.map.addTilesetImage(tileset.name,
level_data.map.tilesets[tileset_index]);
43         tileset_index += 1;
44     }, this);
45
46     if (this.level_data.first_level) {
47         localStorage.clear();
48     }
49 };
50
51 Bomberman.TiledState.prototype.create = function () {
52     "use strict";
53     var group_name, object_layer, collision_tiles;
54
55     // create map layers
56     this.layers = {};
57     this.map.layers.forEach(function (layer) {
58         this.layers[layer.name] =
this.map.createLayer(layer.name);
59         if (layer.properties.collision) { // collision layer
60             collision_tiles = [];
61             layer.data.forEach(function (data_row) { // find
tiles used in the layer
62                 data_row.forEach(function (tile) {
63                     // check if it's a valid tile index and
isn't already in the list
64                     if (tile.index > 0 &&
collision_tiles.indexOf(tile.index) === -1) {
65                         collision_tiles.push(tile.index);
66                     }
67                 }, this);
68             }, this);
69             this.map.setCollision(collision_tiles, true,
layer.name);
70         }
```

```
71        }, this);
72        // resize the world to be the size of the current layer
73        this.layers[this.map.layer.name].resizeWorld();
74
75        // create groups
76        this.groups = {};
77        this.level_data.groups.forEach(function (group_name) {
78            this.groups[group_name] = this.game.add.group();
79        }, this);
80
81        this.prefabs = {};
82
83        for (object_layer in this.map.objects) {
84            if (this.map.objects.hasOwnProperty(object_layer)) {
85                // create layer objects
86                this.map.objects[object_layer].forEach(this.create
_object, this);
87            }
88        }
89
90        this.game.user_input =
this.game.plugins.add(Bomberman.UserInput, this,
JSON.parse(this.game.cache.getText("user_input")));
91
92        this.init_hud();
93    };
94
95    Bomberman.TiledState.prototype.create_object = function
(object) {
96        "use strict";
97        var object_y, position, prefab;
98        // tiled coordinates starts in the bottom left corner
99        object_y = (object.gid) ? object.y - (this.map.tileHeight
/ 2) : object.y + (object.height / 2);
100       position = {"x": object.x + (this.map.tileHeight / 2),
"y": object_y};
101       // create object according to its type
102       if (this.prefab_classes.hasOwnProperty(object.type)) {
103           prefab = new this.prefab_classes[object.type](this,
object.name, position, object.properties);
104       }
105       this.prefabs[object.name] = prefab;
106   };
107
108   Bomberman.TiledState.prototype.init_hud = function () {
```

```
109     "use strict";
110     var player1_lives_position, player1_lives_properties,
player1_lives, player2_lives_position, player2_lives_properties,
player2_lives;
111
112     // create the lives prefab for player1
113     player1_lives_position = new Phaser.Point(0.1 *
this.game.world.width, 0.07 * this.game.world.height);
114     player1_lives_properties = {group: "hud", texture:
"heart_image", number_of_lives: 3, player: "player1"};
115     player1_lives = new Bomberman.Lives(this, "lives",
player1_lives_position, player1_lives_properties);
116
117     // create the lives prefab for player2
118     player2_lives_position = new Phaser.Point(0.9 *
this.game.world.width, 0.07 * this.game.world.height);
119     player2_lives_properties = {group: "hud", texture:
"heart_image", number_of_lives: 3, player: "player2"};
120     player2_lives = new Bomberman.Lives(this, "lives",
player2_lives_position, player2_lives_properties);
121 };
122
123 Bomberman.TiledState.prototype.show_game_over = function ()
{
124     "use strict";
125     var game_over_panel, game_over_position,
game_over_bitmap, panel_text_style;
126     // create a bitmap do show the game over panel
127     game_over_position = new Phaser.Point(0,
this.game.world.height);
128     game_over_bitmap =
this.add.bitmapData(this.game.world.width,
this.game.world.height);
129     game_over_bitmap.ctx.fillStyle = "#000";
130     game_over_bitmap.ctx.fillRect(0, 0,
this.game.world.width, this.game.world.height);
131     panel_text_style = {game_over: {font: "32px Arial", fill:
"#FFF"},
132                         winner: {font: "20px Arial", fill:
"#FFF"}};
133     // create the game over panel
134     game_over_panel =
this.create_game_over_panel(game_over_position,
game_over_bitmap, panel_text_style);
135     this.groups.hud.add(game_over_panel);
```

```
136 };
137
138 Bomberman.TiledState.prototype.create_game_over_panel =
function (position, texture, text_style) {
139     "use strict";
140     var game_over_panel_properties, game_over_panel;
141     game_over_panel_properties = {texture: texture, group:
"hud", text_style: text_style, animation_time: 500};
142     game_over_panel = new Bomberman.GameOverPanel(this,
"game_over_panel", position, game_over_panel_properties);
143     return game_over_panel;
144 };
```

```
1 var Bomberman = Bomberman || {};
2
3 Bomberman.ClassicState = function () {
4     "use strict";
5     Bomberman.TiledState.call(this);
6 };
7
8 Bomberman.ClassicState.prototype =
Object.create(Bomberman.TiledState.prototype);
9 Bomberman.ClassicState.prototype.constructor =
Bomberman.ClassicState;
10
11 Bomberman.ClassicState.prototype.init_hud = function () {
12     "use strict";
13     var player1_lives_position, player1_lives_properties,
player1_lives, player2_lives_position, player2_lives_properties,
player2_lives;
14
15     // create the lives prefab for player1
16     player1_lives_position = new Phaser.Point(0.1 *
this.game.world.width, 0.07 * this.game.world.height);
17     player1_lives_properties = {group: "hud", texture:
"heart_image", number_of_lives: 3, player: "player1"};
18     player1_lives = new Bomberman.Lives(this, "lives",
player1_lives_position, player1_lives_properties);
19
20     // create the lives prefab for player2
21     player2_lives_position = new Phaser.Point(0.9 *
this.game.world.width, 0.07 * this.game.world.height);
22     player2_lives_properties = {group: "hud", texture:
"heart_image", number_of_lives: 3, player: "player2"};
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
23     player2_lives = new Bomberman.Lives(this, "lives",
player2_lives_position, player2_lives_properties);
24 };
25
26 Bomberman.ClassicState.prototype.game_over = function () {
27     "use strict";
28     this.game.state.start("BootState", true, false,
"assets/levels/level1.json", "ClassicState");
29 };
30
31 Bomberman.ClassicState.prototype.next_level = function () {
32     "use strict";
33     localStorage.number_of_lives =
this.prefabs.player.number_of_lives;
34     localStorage.number_of_bombs =
this.prefabs.player.number_of_bombs;
35     this.game.state.start("BootState", true, false,
this.level_data.next_level, "ClassicState");
36 };
```

```
1 var Bomberman = Bomberman || {};
2
3 Bomberman.BattleState = function () {
4     "use strict";
5     Bomberman.TiledState.call(this);
6 };
7
8 Bomberman.BattleState.prototype =
Object.create(Bomberman.TiledState.prototype);
9 Bomberman.BattleState.prototype.constructor =
Bomberman.BattleState;
10
11 Bomberman.BattleState.prototype.init_hud = function () {
12     "use strict";
13     var player1_lives_position, player1_lives_properties,
player1_lives, player2_lives_position, player2_lives_properties,
player2_lives;
14
15     // create the lives prefab for player1
16     player1_lives_position = new Phaser.Point(0.1 *
this.game.world.width, 0.07 * this.game.world.height);
17     player1_lives_properties = {group: "hud", texture:
"heart_image", number_of_lives: 3, player: "player1"};
18     player1_lives = new Bomberman.Lives(this, "lives",
player1_lives_position, player1_lives_properties);
```

```
19
20     // create the lives prefab for player2
21     player2_lives_position = new Phaser.Point(0.9 *
this.game.world.width, 0.07 * this.game.world.height);
22     player2_lives_properties = {group: "hud", texture:
"heart_image", number_of_lives: 3, player: "player2"};
23     player2_lives = new Bomberman.Lives(this, "lives",
player2_lives_position, player2_lives_properties);
24 };
25
26 Bomberman.BattleState.prototype.show_game_over = function ()
{
27     "use strict";
28     if (this.prefabs.player1.alive) {
29         this.winner = this.prefabs.player1.name;
30     } else {
31         this.winner = this.prefabs.player2.name;
32     }
33     Bomberman.TiledState.prototype.show_game_over.call(this);
34 };
35
36 Bomberman.BattleState.prototype.create_game_over_panel =
function (position, texture, text_style) {
37     "use strict";
38     var game_over_panel_properties, game_over_panel;
39     game_over_panel_properties = {texture: texture, group:
"hud", text_style: text_style, animation_time: 500, winner:
this.winner};
40     game_over_panel = new Bomberman.BattleGameOverPanel(this,
"game_over_panel", position, game_over_panel_properties);
41     return game_over_panel;
42 };
43
44 Bomberman.BattleState.prototype.game_over = function () {
45     "use strict";
46     this.game.state.restart(true, false, this.level_data);
47 };
```

Before finishing, we will add a game over message, which will be different for each game mode. In the classic mode, it will only show the Game Over message, while in the battle mode it will also show the winner player. To do that, we will create a GameOverPanel prefab, as shown below. This prefab will have a bitmap texture, and will start with an animation to appear on the screen. Once the animation finishes, it shows the game over message. We create the GameOverPanel in the TiledState, as shown before.

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```javascript
1 var Bomberman = Bomberman || {};
2
3 Bomberman.GameOverPanel = function (game_state, name,
position, properties) {
4     "use strict";
5     var movement_animation;
6     Bomberman.Prefab.call(this, game_state, name, position,
properties);
7
8     this.text_style = properties.text_style;
9
10    this.alpha = 0.5;
11    // create a tween animation to show the game over panel
12    movement_animation = this.game_state.game.add.tween(this);
13    movement_animation.to({y: 0}, properties.animation_time);
14    movement_animation.onComplete.add(this.show_game_over,
this);
15    movement_animation.start();
16 };
17
18 Bomberman.GameOverPanel.prototype =
Object.create(Bomberman.Prefab.prototype);
19 Bomberman.GameOverPanel.prototype.constructor =
Bomberman.GameOverPanel;
20
21 Bomberman.GameOverPanel.prototype.show_game_over = function
() {
22    "use strict";
23    var game_over_text;
24    // add game over text
25    game_over_text =
this.game_state.game.add.text(this.game_state.game.world.width /
2, this.game_state.game.world.height * 0.4, "Game Over",
this.text_style.game_over);
26    game_over_text.anchor.setTo(0.5);
27    this.game_state.groups.hud.add(game_over_text);
28
29    // add event to restart level
30    this.inputEnabled = true;
31    this.events.onInputDown.add(this.game_state.game_over,
this.game_state);
32 };
```

Finally, to show a different message in the BattleState, we create a BatleGameOverPanel, which extends GameOverPanel but includes the message showing the winner. To allow this, we also change the "create_game_over_panel" method from BattleState to create this prefab accordingly.

```javascript
1 var Bomberman = Bomberman || {};
2
3 Bomberman.BattleGameOverPanel = function (game_state, name,
position, properties) {
4     "use strict";
5     var movement_animation;
6     Bomberman.GameOverPanel.call(this, game_state, name,
position, properties);
7
8     this.winner = properties.winner;
9 };
10
11 Bomberman.BattleGameOverPanel.prototype =
Object.create(Bomberman.GameOverPanel.prototype);
12 Bomberman.BattleGameOverPanel.prototype.constructor =
Bomberman.BattleGameOverPanel;
13
14 Bomberman.BattleGameOverPanel.prototype.show_game_over =
function () {
15     "use strict";
16     var winner_text;
17     Bomberman.GameOverPanel.prototype.show_game_over.call(this
);
18
19     // show the winner if it's in battle mode
20     winner_text =
this.game_state.game.add.text(this.game_state.world.width / 2,
this.game_state.game.world.height * 0.6, "Winner: " +
this.winner, this.text_style.winner);
21     winner_text.anchor.setTo(0.5);
22     this.game_state.groups.hud.add(winner_text);
23 };
```

You can already try playing the battle mode to see if it is working as expected. Try winning with each player to check if the game over message is correct.
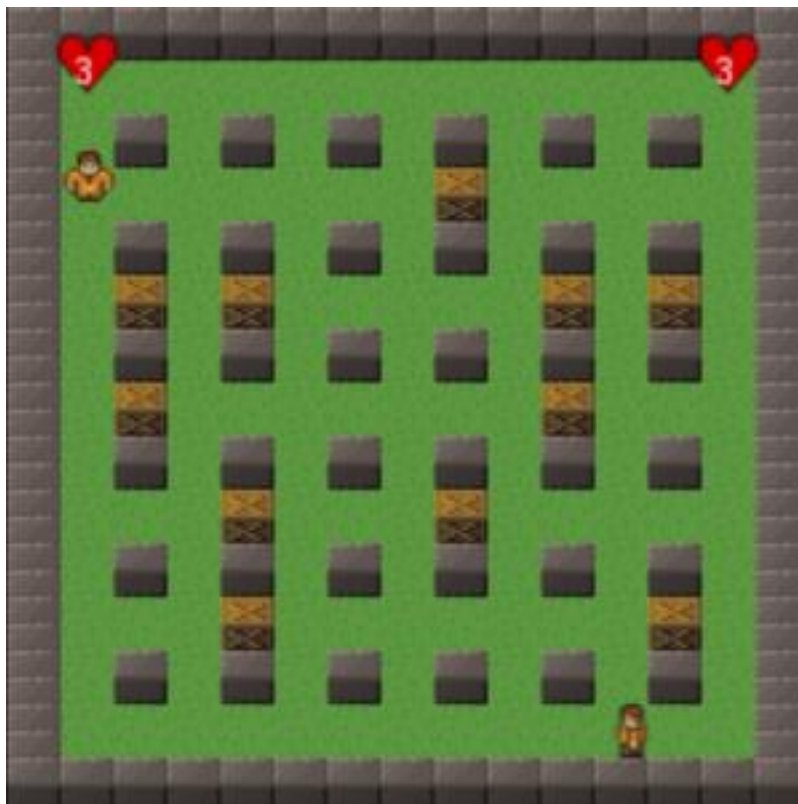
Adding a title screen with the menu

To create our menu, we will create a Menu and a MenuItem prefabs. The Menu prefab will have a list of menu items, and allows navigating through them using the arrow keys. When the user press the UP or DOWN arrow key, it changes the current menu item. When the SPACEBAR is pressed, it selects the current item. The MenuItem prefab will play an animation when it is the current item and has a "select" method to start the game. In this method it will call the BootState to start the game with the ClassicState or the BattleState. In our title screen we will have two menu items, one for each game mode. The code for both prefabs is shown below.

```
1 var Bomberman = Bomberman || {};
2
3 Bomberman.Menu = function (game_state, name, position,
properties) {
```

```
4     "use strict";
5     var live_index, life;
6     Bomberman.Prefab.call(this, game_state, name, position,
properties);
7
8     this.visible = false;
9
10     this.menu_items = properties.menu_items;
11     this.current_item_index = 0;
12     this.menu_items[0].selection_over();
13
14     this.cursor_keys =
this.game_state.game.input.keyboard.createCursorKeys();
15 };
16
17 Bomberman.Menu.prototype =
Object.create(Bomberman.Prefab.prototype);
18 Bomberman.Menu.prototype.constructor = Bomberman.Menu;
19
20 Bomberman.Menu.prototype.update = function () {
21     "use strict";
22     if (this.cursor_keys.up.isDown && this.current_item_index
> 0) {
23         // navigate to previous item
24         this.menu_items[this.current_item_index].selection_out
();
25         this.current_item_index -= 1;
26         this.menu_items[this.current_item_index].selection_ove
r();
27     } else if (this.cursor_keys.down.isDown &&
this.current_item_index < this.menu_items.length - 1) {
28         // navigate to next item
29         this.menu_items[this.current_item_index].selection_out
();
30         this.current_item_index += 1;
31         this.menu_items[this.current_item_index].selection_ove
r();
32     }
33
34     if
(this.game_state.game.input.keyboard.isDown(Phaser.Keyboard.SPAC
EBAR)) {
35         this.menu_items[this.current_item_index].select();
36     }
37 };
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
1 var Bomberman = Bomberman || {};
2
3 Bomberman.MenuItem = function (game_state, name, position,
properties) {
4     "use strict";
5     Bomberman.TextPrefab.call(this, game_state, name, position,
properties);
6
7     this.anchor.setTo(0.5);
8
9     this.on_selection_animation =
this.game_state.game.add.tween(this.scale);
10     this.on_selection_animation.to({x: 1.5 * this.scale.x, y:
1.5 * this.scale.y}, 500);
11     this.on_selection_animation.to({x: this.scale.x, y:
this.scale.y}, 500);
12     this.on_selection_animation.repeatAll(-1);
13
14     this.level_file = properties.level_file;
15     this.state_name = properties.state_name;
16 };
17
18 Bomberman.MenuItem.prototype =
Object.create(Bomberman.TextPrefab.prototype);
19 Bomberman.MenuItem.prototype.constructor =
Bomberman.MenuItem;
20
21 Bomberman.MenuItem.prototype.selection_over = function () {
22     "use strict";
23     if (this.on_selection_animation.isPaused) {
24         this.on_selection_animation.resume();
25     } else {
26         this.on_selection_animation.start();
27     }
28 };
29
30 Bomberman.MenuItem.prototype.selection_out = function () {
31     "use strict";
32     this.on_selection_animation.pause();
33 };
34
35 Bomberman.MenuItem.prototype.select = function () {
36     "use strict";
37     // starts game state
```

```
38      this.game_state.state.start("BootState", true, false,
this.level_file, this.state_name);
39 };
```

To show the title screen, we will create a TitleState. In this state, we only have to change
the "create" method to create the menu. To simplify the code, I put the menu items data
in a JSON file, shown below. The TitleState starts by showing the game title and then
reads the menu items data from this file, storing all menu items in an array. This array of
menu items is then used to create the menu. The code for TitleState is shown below.

```
1  {
2      "menu_items": {
3          "classic_mode": {
4              "position": {"x": 120, "y": 144},
5              "properties": {
6                  "text": "Classic mode",
7                  "style": {"font": "16px Arial", "fill":
"#FFF"},
8                  "group": "menu_items",
9                  "level_file": "assets/levels/level1.json",
10                  "state_name": "ClassicState"
11              }
12          },
13          "battle_mode": {
14              "position": {"x": 120, "y": 192},
15              "properties": {
16                  "text": "Battle mode",
17                  "style": {"font": "16px Arial", "fill":
"#FFF"},
18                  "group": "menu_items",
19                  "level_file":
"assets/levels/battle_level.json",
20                  "state_name": "BattleState"
21              }
22          }
23      },
24      "groups": [
25          "background",
26          "menu_items",
27          "hud"
28      ]
29 }
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Page 217

```
1 var Bomberman = Bomberman || {};
2
3 Bomberman.TitleState = function () {
4     "use strict";
5     Phaser.State.call(this);
6 };
7
8 Bomberman.TitleState.prototype =
Object.create(Phaser.State.prototype);
9 Bomberman.TitleState.prototype.constructor =
Bomberman.TitleState;
10
11 Bomberman.TitleState.prototype.init = function (level_data) {
12     "use strict";
13     this.level_data = level_data;
14
15     this.scale.scaleMode = Phaser.ScaleManager.SHOW_ALL;
16     this.scale.pageAlignHorizontally = true;
17     this.scale.pageAlignVertically = true;
18 };
19
20 Bomberman.TitleState.prototype.create = function () {
21     "use strict";
22     var title_position, title_style, title, menu_position,
menu_items, menu_properties, menu_item_name, menu_item, menu;
23
24     // create groups
25     this.groups = {};
26     this.level_data.groups.forEach(function (group_name) {
27         this.groups[group_name] = this.game.add.group();
28     }, this);
29
30     this.prefabs = {};
31
32     // adding title
33     title_position = new Phaser.Point(0.5 *
this.game.world.width, 0.3 * this.game.world.height);
34     title_style = {font: "36px Arial", fill: "#FFF"};
35     title = new Bomberman.TextPrefab(this, "title",
title_position, {text: "Bomberman", style: title_style, group:
"hud"});
36     title.anchor.setTo(0.5);
37
38     // adding menu
39     menu_position = new Phaser.Point(0, 0);
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Page 218

```
40    menu_items = [];
41    for (menu_item_name in this.level_data.menu_items) {
42        if
(this.level_data.menu_items.hasOwnProperty(menu_item_name)) {
43            menu_item =
this.level_data.menu_items[menu_item_name];
44            menu_items.push(new Bomberman.MenuItem(this,
menu_item_name, menu_item.position, menu_item.properties));
45        }
46    }
47    menu_properties = {texture: "", group: "background",
menu_items: menu_items};
48    menu = new Bomberman.Menu(this, "menu", menu_position,
menu_properties);
49 };
```

Now you can change main.js to start with TitleState and see if everything is working. Try playing both game modes to check if the menu is working correctly.
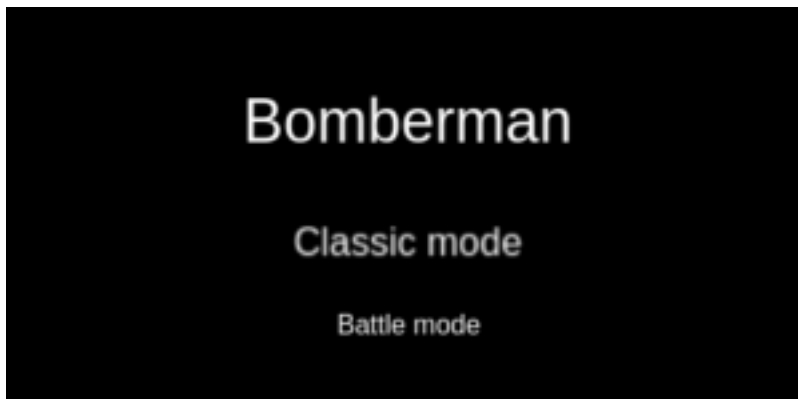
Finishing the game
And now we finished our Bomberman game!

How to Make a Turn-Based RPG Game in Phaser – Part 1
By Renan Oliveira

In this tutorial series, we will create a turn-based RPG game, such as the Final Fantasy series. First, we will create the battle state for our game. Then we will create a state for the world, which will change to the battle state every time an enemy is found. Finally, we will add content, such as items, levels, NPCs and save points. In this first tutorial I will cover the following content:

- Creating a battle state which will be called during the game
- Creating a menu to show the player and enemy units
- Creating a simple turn-based game, where each unit acts once

To read this tutorial, it is important that you are familiar with the following concepts:

- Javascript and object-oriented concepts.
- Basic Phaser concepts, such as: states, sprites, groups and arcade physics

Assets copyright

The monsters assets used in this tutorial are available in http://opengameart.org/content/2d-rpg-enemy-set and were created by the following artists: Brett Steele (Safir-Kreuz), Joe Raucci (Sylon), Vicki Beinhart (Namakoro), Tyler Olsen (Roots). The characters assets are available in http://opengameart.org/content/24×32-characters-with-faces-big-pack and were created by Svetlana Kushnariova (email: lana-chan@yandex.ru). All assets are available through the Creative Commons license.

Source code files

You can download the tutorial source code files here.

Boot and loading states

We will create boot and loading states to load all the game assets before it starts. All this content will be read from a JSON file, as shown below. Notice that in this file we have to define the state assets, groups and prefabs.

```
1 {
2     "assets": {
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Page 220

```
3          "rectangle_image": {"type": "image", "source":
"assets/images/rectangle.png"},
4          "grass_tile_image": {"type": "image", "source":
"assets/images/grass_tile.png"},
5          "male_fighter_spritesheet": {"type": "spritesheet",
"source": "assets/images/characters/fighter_m.png",
"frame_width": 24, "frame_height": 32},
6          "female_mage_spritesheet": {"type": "spritesheet",
"source": "assets/images/characters/mage_f.png", "frame_width":
24, "frame_height": 32},
7          "bat_spritesheet": {"type": "spritesheet", "source":
"assets/images/monsters/bat.png", "frame_width": 128,
"frame_height": 128}
8      },
9      "groups": [
10          "background",
11          "player_units",
12          "enemy_units",
13          "hud"
14      ],
15      "prefabs": {
16          "background": {
17              "type": "background",
18              "position": {"x": 0, "y": 0},
19              "properties": {
20                  "texture": "grass_tile_image",
21                  "group": "background",
22                  "width": 320,
23                  "height": 320
24              }
25          },
26          "enemy_rectangle": {
27              "type": "rectangle",
28              "position": {"x": 0, "y": 200},
29              "properties": {
30                  "texture": "rectangle_image",
31                  "group": "hud",
32                  "scale": {"x": 0.3, "y": 1}
33              }
34          },
35          "action_rectangle": {
36              "type": "rectangle",
37              "position": {"x": 96, "y": 200},
38              "properties": {
39                  "texture": "rectangle_image",
```

```
40                    "group": "hud",
41                    "scale": {"x": 0.3, "y": 1}
42                }
43           },
44           "player_rectangle": {
45                "type": "rectangle",
46                "position": {"x": 192, "y": 200},
47                "properties": {
48                    "texture": "rectangle_image",
49                    "group": "hud",
50                    "scale": {"x": 0.4, "y": 1}
51                }
52           },
53           "fighter": {
54                "type": "player_unit",
55                "position": {"x": 250, "y": 70},
56                "properties": {
57                    "texture": "male_fighter_spritesheet",
58                    "group": "player_units",
59                    "frame": 10,
60                    "stats": {
61                        "attack": 15,
62                        "defense": 5,
63                        "health": 100
64                    }
65                }
66           },
67           "mage": {
68                "type": "player_unit",
69                "position": {"x": 250, "y": 150},
70                "properties": {
71                    "texture": "female_mage_spritesheet",
72                    "group": "player_units",
73                    "frame": 10,
74                    "stats": {
75                        "attack": 20,
76                        "defense": 2,
77                        "health": 100
78                    }
79                }
80           },
81           "bat1": {
82                "type": "enemy_unit",
83                "position": {"x": 100, "y": 90},
84                "properties": {
```

```
85                    "texture": "bat_spritesheet",
86                    "group": "enemy_units",
87                    "stats": {
88                          "attack": 10,
89                          "defense": 1,
90                          "health": 30
91                    }
92               }
93          },
94        "bat2": {
95             "type": "enemy_unit",
96             "position": {"x": 100, "y": 170},
97             "properties": {
98                  "texture": "bat_spritesheet",
99                  "group": "enemy_units",
100                 "stats": {
101                       "attack": 10,
102                       "defense": 1,
103                       "health": 30
104                 }
105            }
106       }
107    }
108 }
```

The BootState code is shown below. It will only load the JSON file and call the LoadingState.

```
1 var RPG = RPG || {};
2
3 RPG.BootState = function () {
4    "use strict";
5    Phaser.State.call(this);
6 };
7
8 RPG.BootState.prototype =
Object.create(Phaser.State.prototype);
9 RPG.BootState.prototype.constructor = RPG.BootState;
10
11 RPG.BootState.prototype.init = function (level_file,
next_state) {
12    "use strict";
13    this.level_file = level_file;
```

```
14     this.next_state = next_state;
15 };
16
17 RPG.BootState.prototype.preload = function () {
18     "use strict";
19     this.load.text("level1", this.level_file);
20 };
21
22 RPG.BootState.prototype.create = function () {
23     "use strict";
24     var level_text, level_data;
25     level_text = this.game.cache.getText("level1");
26     level_data = JSON.parse(level_text);
27     this.game.state.start("LoadingState", true, false,
level_data, this.next_state);
28 };
```

The LoadingState is responsible for loading all the necessary assets. To do that, it will read the assets from the JSON file and load them accordingly. The code for LoadingState is shown below. Notice that the "preload" method loads the correct Phaser asset according to the asset type. At the end, it will call the next state (in our case, BattleState).

```
1 var RPG = RPG || {};
2
3 RPG.LoadingState = function () {
4     "use strict";
5     Phaser.State.call(this);
6 };
7
8 RPG.LoadingState.prototype =
Object.create(Phaser.State.prototype);
9 RPG.LoadingState.prototype.constructor = RPG.LoadingState;
10
11 RPG.LoadingState.prototype.init = function (level_data,
next_state) {
12     "use strict";
13     this.level_data = level_data;
14     this.next_state = next_state;
15 };
16
17 RPG.LoadingState.prototype.preload = function () {
18     "use strict";
19     var assets, asset_loader, asset_key, asset;
```

```
20      assets = this.level_data.assets;
21      for (asset_key in assets) { // load assets according to
asset key
22          if (assets.hasOwnProperty(asset_key)) {
23              asset = assets[asset_key];
24              switch (asset.type) {
25              case "image":
26                  this.load.image(asset_key, asset.source);
27                  break;
28              case "spritesheet":
29                  this.load.spritesheet(asset_key, asset.source,
asset.frame_width, asset.frame_height, asset.frames,
asset.margin, asset.spacing);
30                  break;
31              case "tilemap":
32                  this.load.tilemap(asset_key, asset.source,
null, Phaser.Tilemap.TILED_JSON);
33                  break;
34              }
35          }
36      }
37 };
38
39 RPG.LoadingState.prototype.create = function () {
40      "use strict";
41      this.game.state.start(this.next_state, true, false,
this.level_data);
42 };
```

Creating the battle state

We will create a battle state that will show the player party, enemy units and a menu so the player can choose which enemy to attack. Initially, we will just create the basic structure, showing the units without the menus. Also, in this tutorial all the units will be read from the JSON file (shown above). In the next tutorials, we will pass this data as a parameter in the "init" method.

The BattleState code is shown below. The "init" method only saves the level data and sets the game scale. The "create" method starts by creating all the groups and then create the prefabs (groups and prefabs are read from the JSON file).

```
1 var RPG = RPG || {};
2
```

```
3  RPG.BattleState = function () {
4      "use strict";
5      Phaser.State.call(this);
6
7      this.prefab_classes = {
8          "background": RPG.TilePrefab.prototype.constructor,
9          "rectangle": RPG.Prefab.prototype.constructor,
10          "player_unit": RPG.PlayerUnit.prototype.constructor,
11          "enemy_unit": RPG.EnemyUnit.prototype.constructor
12      };
13
14      this.TEXT_STYLE = {font: "14px Arial", fill: "#FFFFFF"};
15 };
16
17 RPG.BattleState.prototype =
Object.create(Phaser.State.prototype);
18 RPG.BattleState.prototype.constructor = RPG.BattleState;
19
20 RPG.BattleState.prototype.init = function (level_data) {
21      "use strict";
22      this.level_data = level_data;
23
24      this.scale.scaleMode = Phaser.ScaleManager.SHOW_ALL;
25      this.scale.pageAlignHorizontally = true;
26      this.scale.pageAlignVertically = true;
27 };
28
29 RPG.BattleState.prototype.create = function () {
30      "use strict";
31      var group_name, prefab_name, player_unit_name,
enemy_unit_name;
32
33      // create groups
34      this.groups = {};
35      this.level_data.groups.forEach(function (group_name) {
36          this.groups[group_name] = this.game.add.group();
37      }, this);
38
39      // create prefabs
40      this.prefabs = {};
41      for (prefab_name in this.level_data.prefabs) {
42          if
(this.level_data.prefabs.hasOwnProperty(prefab_name)) {
43              // create prefab
```

```
44               this.create_prefab(prefab_name,
this.level_data.prefabs[prefab_name]);
45          }
46      }
47 };
48
49 RPG.BattleState.prototype.create_prefab = function
(prefab_name, prefab_data) {
50      "use strict";
51      var prefab;
52      // create object according to its type
53      if (this.prefab_classes.hasOwnProperty(prefab_data.type))
{
54          prefab = new
this.prefab_classes[prefab_data.type](this, prefab_name,
prefab_data.position, prefab_data.properties);
55      }
56 };
```

For each prefab, the "create_prefab" method will instantiate the correct prefab according to its type. Two things are necessary for that to work:

1   All prefabs must have the same constructor. To achieve that, we create a generic Prefab class (shown below) which all Prefabs must extend.
2   We must have a property mapping each prefab type to its constructor. This property is defined in the BattleState constructor.Since all units are declared in the JSON file, by now they will already appear in our BattleState, as shown in the

figure below. Try executing the game now, to see it. You can create empty classes for the units, just to make it work for now.
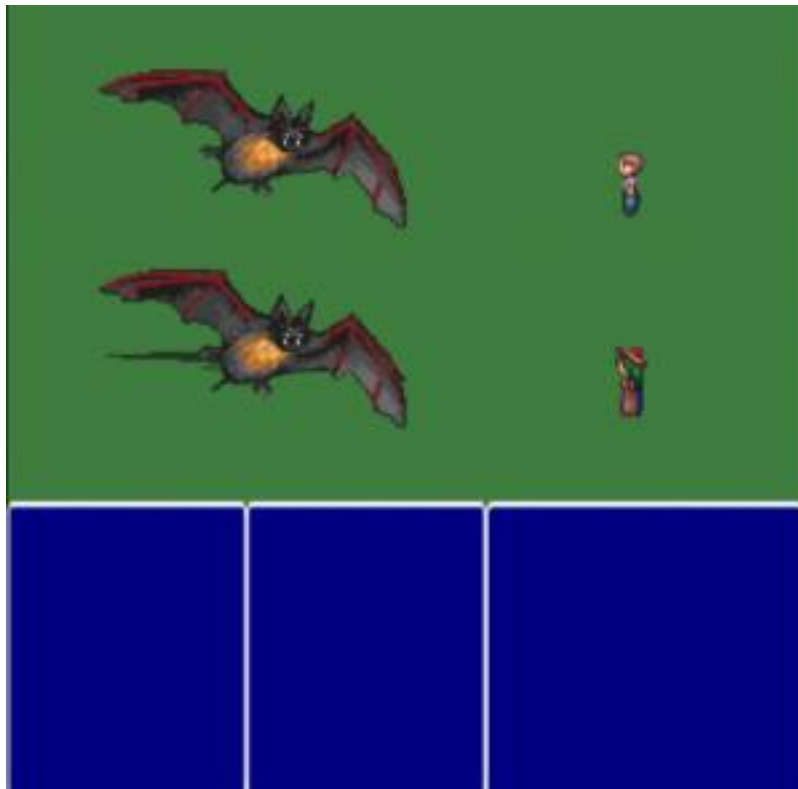
Creating the battle state HUD

Now, we are going to add three menus for our BattleState:

1. Player units menu: will show the player units and their health
2. Actions menu: will show the available actions of the player
3. Enemy units menu: will show the enemy units so the player can choose which one to attackBefore doing this, we will create a Menu and MenuItem prefabs, as shown below. The Menu prefab has an array of MenuItems and allows navigating through them when it is enabled. Since we will have more than one Menu at the same time, we need methods to enable and disable them when necessary. The enable method will add callbacks to the keyboard to allow menu navigation. The other methods from Menu will be used later on in this tutorial.

```
1 var RPG = RPG || {};
2
3 RPG.Menu = function (game_state, name, position, properties) {
4     "use strict";
5     var live_index, life;
6     RPG.Prefab.call(this, game_state, name, position,
properties);
7
8     this.visible = false;
9
10    this.menu_items = properties.menu_items;
11
12    this.current_item_index = 0;
13 };
14
15 RPG.Menu.prototype = Object.create(RPG.Prefab.prototype);
16 RPG.Menu.prototype.constructor = RPG.Menu;
17
18 RPG.Menu.prototype.process_input = function (event) {
19     "use strict";
20     switch (event.keyCode) {
21     case Phaser.Keyboard.UP:
22         if (this.current_item_index > 0) {
23             // navigate to previous item
24             this.move_selection(this.current_item_index - 1);
25         }
26         break;
27     case Phaser.Keyboard.DOWN:
```

```
28          if (this.current_item_index < this.menu_items.length -
1) {
29              // navigate to next item
30              this.move_selection(this.current_item_index + 1);
31          }
32          break;
33      case Phaser.Keyboard.SPACEBAR:
34          this.menu_items[this.current_item_index].select();
35          break;
36      }
37 };
38
39 RPG.Menu.prototype.move_selection = function (item_index) {
40      "use strict";
41      this.menu_items[this.current_item_index].selection_out();
42      this.current_item_index = item_index;
43      this.menu_items[this.current_item_index].selection_over();
44 };
45
46 RPG.Menu.prototype.find_item_index = function (text) {
47      "use strict";
48      var item_index;
49      for (item_index = 0; item_index < this.menu_items.length;
item_index += 1) {
50          if (this.menu_items[item_index].text === text) {
51              return item_index;
52          }
53      }
54 };
55
56 RPG.Menu.prototype.remove_item = function (index) {
57      "use strict";
58      var menu_item;
59      menu_item = this.menu_items[index];
60      // remove menu item
61      this.menu_items.splice(index, 1);
62      // update current_item_index if necessary
63      if (this.current_item_index === index) {
64          this.current_item_index = 0;
65      }
66      return menu_item;
67 };
68
69 RPG.Menu.prototype.enable = function () {
70      "use strict";
```

```
71      this.current_item_index = 0;
72      if (this.menu_items.length > 0) {
73          this.menu_items[this.current_item_index].selection_ove
r();
74      }
75      this.game_state.game.input.keyboard.addCallbacks(this,
this.process_input);
76 };
77
78 RPG.Menu.prototype.disable = function () {
79      "use strict";
80      if (this.menu_items.length > 0) {
81          this.menu_items[this.current_item_index].selection_out
();
82      }
83      this.current_item_index = 0;
84 };
```

The MenuItem prefab only has to implement the "selection_over" and "selection_out" methods (called by Menu). When a MenuItem is over the selection, it will change its color. Notice that MenuItem extends TextPrefab instead of Prefab. This class is similar to the generic Prefab class, but extends Phaser.Text instead of Phaser.Sprite, as shown below. One important thing to notice is that all texts will have the same text style, defined as "TEXT_STYLE" in BattleState. However, we must use Object.create() to use a copy of it when creating each TextPrefab, otherwise all of them would have a reference for the same object.

```
1 var RPG = RPG || {};
2
3 RPG.MenuItem = function (game_state, name, position,
properties) {
4     "use strict";
5     RPG.TextPrefab.call(this, game_state, name, position,
properties);
6 };
7
8 RPG.MenuItem.prototype =
Object.create(RPG.TextPrefab.prototype);
9 RPG.MenuItem.prototype.constructor = RPG.MenuItem;
10
11 RPG.MenuItem.prototype.selection_over = function () {
12     "use strict";
13     this.fill = "#FFFF00";
```

```
14 };
15
16 RPG.MenuItem.prototype.selection_out = function () {
17     "use strict";
18     this.fill = "#FFFFFF";
19 };
```

Also, notice that the MenuItem prefab does not implement the method "select", which is called by Menu. So, we have to create new prefabs that extend MenuItem and implement it. We do that by creating three new prefabs: AttackMenuItem, PlayerMenuItem and EnemyMenuItem. The first one will only disable the actions menu and enable the enemy units menu, so the player can choose the attack target.

```
1 var RPG = RPG || {};
2
3 RPG.AttackMenuItem = function (game_state, name, position,
properties) {
4     "use strict";
5     RPG.MenuItem.call(this, game_state, name, position,
properties);
6 };
7
8 RPG.AttackMenuItem.prototype =
Object.create(RPG.MenuItem.prototype);
9 RPG.AttackMenuItem.prototype.constructor = RPG.AttackMenuItem;
10
11 RPG.AttackMenuItem.prototype.select = function () {
12     "use strict";
13     // disable actions menu
14     this.game_state.prefabs.actions_menu.disable();
15     // enable enemy units menu so the player can choose the
target
16     this.game_state.prefabs.enemy_units_menu.enable();
17 };
```

The second one will not do anything when selected, since it will only be used to show the current player unit. However, we want to show the player unit health, so it will create a ShowState prefab (whose code is shown below), which will show the player unit health.

```
1 var RPG = RPG || {};
2
```

```
3 RPG.PlayerMenuItem = function (game_state, name, position,
properties) {
4    "use strict";
5    RPG.MenuItem.call(this, game_state, name, position,
properties);
6
7    this.player_unit_health = new RPG.ShowStat(this.game_state,
this.text + "_health", {x: 280, y: this.y}, {group: "hud", text:
"", style: properties.style, prefab: this.text, stat:
"health"});
8 };
9
10 RPG.PlayerMenuItem.prototype =
Object.create(RPG.MenuItem.prototype);
11 RPG.PlayerMenuItem.prototype.constructor =
RPG.PlayerMenuItem;
12
13 RPG.PlayerMenuItem.prototype.select = function () {
14    "use strict";
15 };
```

Finally, the EnemyMenuItem will be used to select the enemy to be attacked. We can do that by getting the enemy prefab (the menu item text will be the prefab name) and making the current unit attacking unit. We will add the current unit and implement the attack method later.

```
1 var RPG = RPG || {};
2
3 RPG.EnemyMenuItem = function (game_state, name, position,
properties) {
4    "use strict";
5    RPG.MenuItem.call(this, game_state, name, position,
properties);
6 };
7
8 RPG.EnemyMenuItem.prototype =
Object.create(RPG.MenuItem.prototype);
9 RPG.EnemyMenuItem.prototype.constructor = RPG.EnemyMenuItem;
10
11 RPG.EnemyMenuItem.prototype.select = function () {
12    "use strict";
13    var enemy;
14    // get enemy prefab
15    enemy = this.game_state.prefabs[this.text];
```

```
16     // attack selected enemy
17     this.game_state.current_unit.attack(enemy);
18     // disable menus
19     this.game_state.prefabs.enemy_units_menu.disable();
20     this.game_state.prefabs.player_units_menu.disable();
21 };
```

All the menus will be added in the "init_hud" method from BattleState, which will be called at the end of the "create" method. First, the "show_player_actions" method creates the actions menu, which in this tutorial will have only the Attack action. In the next tutorials we will add more actions, such as Magic and Item. Then, the "show_units" method is used to create the player and enemy units. Notice that this method receive as parameter the units group name and menu item constructor, so it can be used to create different kinds of units menu.

```
1 RPG.BattleState.prototype.show_units = function (group_name,
position, menu_item_constructor) {
2     "use strict";
3     var unit_index, menu_items, unit_menu_item, units_menu;
4
5     // create units menu items
6     unit_index = 0;
7     menu_items = [];
8     this.groups[group_name].forEach(function (unit) {
9         unit_menu_item = new menu_item_constructor(this,
unit.name + "_menu_item", {x: position.x, y: position.y +
unit_index * 20}, {group: "hud", text: unit.name, style:
Object.create(this.TEXT_STYLE)});
10         unit_index += 1;
11         menu_items.push(unit_menu_item);
12     }, this);
13     // create units menu
14     units_menu = new RPG.Menu(this, group_name + "_menu",
position, {group: "hud", menu_items: menu_items});
15 };
16
17 RPG.BattleState.prototype.show_player_actions = function
(position) {
18     "use strict";
19     var actions, actions_menu_items, action_index,
actions_menu;
20     // available actions
```

```
21     actions = [{text: "Attack", item_constructor:
RPG.AttackMenuItem.prototype.constructor}];
22     actions_menu_items = [];
23     action_index = 0;
24     // create a menu item for each action
25     actions.forEach(function (action) {
26         actions_menu_items.push(new
action.item_constructor(this, action.text + "_menu_item", {x:
position.x, y: position.y + action_index * 20}, {group: "hud",
text: action.text, style: Object.create(this.TEXT_STYLE)}));
27         action_index += 1;
28     }, this);
29     actions_menu = new RPG.Menu(this, "actions_menu",
position, {group: "hud", menu_items: actions_menu_items});
30 };
```

By now you can already run your game to see if the menu is being correctly displayed.
You can also try enabling some menus to see if you can correctly navigate through them.
The only thing still not working will be the menu selection.
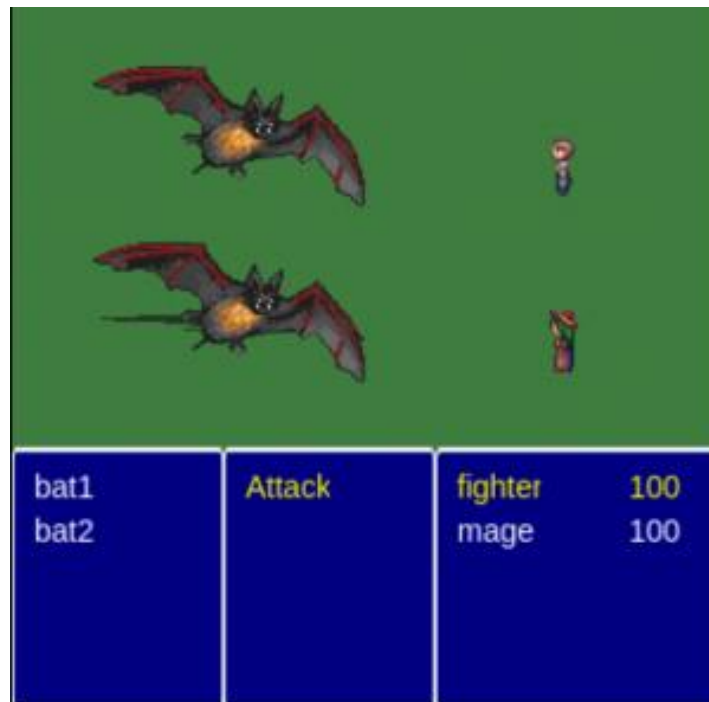
Implementing the turns

In our game we will create an array with all the units and in each turn the first unit in the array will act. First, we create the "units" array in the "create" method, then we call the "next_turn" method.

```
1  // create units array with player and enemy units
2      this.units = [];
3      this.units =
this.units.concat(this.groups.player_units.children);
4      this.units =
this.units.concat(this.groups.enemy_units.children);
5
6      this.next_turn();
```

The "next_turn" method takes the first unit in the array and, if the unit is alive, it acts and is pushed to the end of the units array. Otherwise, it calls the next turn. The code below shows the modifications to BattleState.

```
1  RPG.BattleState.prototype.next_turn = function () {
2      "use strict";
3      // takes the next unit
4      this.current_unit = this.units.shift();
5      // if the unit is alive, it acts, otherwise goes to the
next turn
6      if (this.current_unit.alive) {
7          this.current_unit.act();
8          this.units.push(this.current_unit);
9      } else {
10          this.next_turn();
11      }
12 };
```

Now, we have to implement the "act" method in both EnemyUnit and PlayerUnit. The EnemyUnit "act" method chooses a random player unit as the target and attack it. On the other hand, the "act" method for PlayerUnit highlights the current player unit and enables the enemy units menu, so the player can choose the enemy to attack.

```
1  RPG.EnemyUnit.prototype.act = function () {
2      "use strict";
3      var target_index, target, damage;
4      // randomly choose target
5      target_index = this.game_state.rnd.between(0,
this.game_state.groups.player_units.countLiving() - 1);
```

```
6     target =
this.game_state.groups.player_units.children[target_index];
7
8     this.attack(target);
9 };
10
11 RPG.PlayerUnit.prototype.act = function () {
12     "use strict";
13     var unit_index, player_units_menu_items;
14     // search for the index of this unit in the
player_units_menu
15     unit_index =
this.game_state.prefabs.player_units_menu.find_item_index(this.n
ame);
16     this.game_state.prefabs.player_units_menu.move_selection(u
nit_index);
17
18     // enable menu for choosing the action
19     this.game_state.prefabs.actions_menu.enable();
20 };
```

The "attack" method is the same for both units, so it will be implemented in the Unit prefab. It calculates the damage based on the unit attack and the target defense, and deals that damage to the target unit. Notice that the damage is randomized by multiplying the attack and defense by random multipliers between 0.8 and 1.2. All random generation is done using Phaser RandomDataGenerator (you can check the documentation for more information). After dealing the damage, an attack message is displayed, so the player can have some visual feedback. The ActionMessage prefab (shown below), simply shows a text inside a rectangle, which is killed after some time. One important detail is that the next turn is called when this ActionMessage is killed.

```
1 RPG.Unit.prototype.attack = function (target) {
2     "use strict";
3     var damage, attack_multiplier, defense_multiplier,
action_message_position, action_message_text, attack_message;
4     // attack target
5     attack_multiplier =
this.game_state.game.rnd.realInRange(0.8, 1.2);
6     defense_multiplier =
this.game_state.game.rnd.realInRange(0.8, 1.2);
7     damage = Math.round((attack_multiplier * this.stats.attack)
- (defense_multiplier * target.stats.defense));
8     target.receive_damage(damage);
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
9
10     // show attack message
11     action_message_position = new
Phaser.Point(this.game_state.game.world.width / 2,
this.game_state.game.world.height * 0.1);
12     action_message_text = this.name + " attacks " +
target.name + " with " + damage + " damage";
13     attack_message = new RPG.ActionMessage(this.game_state,
this.name + "_action_message", action_message_position, {group:
"hud", texture: "rectangle_image", scale: {x: 0.75, y: 0.2},
duration: 1, message: action_message_text});
14 };


1 var RPG = RPG || {};
2
3 RPG.ActionMessage = function (game_state, name, position,
properties) {
4     "use strict";
5     RPG.Prefab.call(this, game_state, name, position,
properties);
6
7     this.anchor.setTo(0.5);
8
9     // create message text
10     this.message_text = new RPG.TextPrefab(this.game_state,
this.name + "_message", position, {group: "hud", text:
properties.message, style:
Object.create(this.game_state.TEXT_STYLE)});
11     this.message_text.anchor.setTo(0.5);
12
13     // start timer to destroy the message
14     this.kill_timer = this.game_state.game.time.create();
15     this.kill_timer.add(Phaser.Timer.SECOND *
properties.duration, this.kill, this);
16     this.kill_timer.start();
17 };
18
19 RPG.ActionMessage.prototype =
Object.create(RPG.Prefab.prototype);
20 RPG.ActionMessage.prototype.constructor = RPG.ActionMessage;
21
22 RPG.ActionMessage.prototype.kill = function () {
23     "use strict";
24     Phaser.Sprite.prototype.kill.call(this);
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
25     // when the message is destroyed, call next turn
26     this.message_text.kill();
27     this.game_state.next_turn();
28 };
```

The "receive_damage" method is also the same for both units, and it reduces the unit health and check if it is dead. In addition, it starts an attacked animation, which changes the prefab tint to red and then goes back to the normal.

```
1 var RPG = RPG || {};
2
3 RPG.Unit = function (game_state, name, position, properties) {
4     "use strict";
5     RPG.Prefab.call(this, game_state, name, position,
properties);
6
7     this.anchor.setTo(0.5);
8
9     this.stats = properties.stats;
10
11     this.attacked_animation =
this.game_state.game.add.tween(this);
12     this.attacked_animation.to({tint: 0xFF0000}, 200);
13     this.attacked_animation.onComplete.add(this.restore_tint,
this);
14 };
15
16 RPG.Unit.prototype = Object.create(RPG.Prefab.prototype);
17 RPG.Unit.prototype.constructor = RPG.Unit;
18
19 RPG.Unit.prototype.receive_damage = function (damage) {
20     "use strict";
21     this.stats.health -= damage;
22     this.attacked_animation.start();
23     if (this.stats.health <= 0) {
24         this.stats.health = 0;
25         this.kill();
26     }
27 };
28
29 RPG.Unit.prototype.restore_tint = function () {
30     "use strict";
31     this.tint = 0xFFFFFF;
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
32 };
```

Finally, we have to change the "kill" method of both EnemyUnit and PlayerUnit to update their menus accordingly. First, when an enemy unit dies, it must remove itself from the enemy units menu, which can be done using the methods we already have in the Menu prefab. On the other hand, the player unit will not remove itself from its menu, but only change the alpha of its menu item, making it darker.

```
1 RPG.EnemyUnit.prototype.kill = function () {
2     "use strict";
3     var menu_item_index, menu_item;
4     Phaser.Sprite.prototype.kill.call(this);
5     // remove from the menu
6     menu_item_index =
this.game_state.prefabs.enemy_units_menu.find_item_index(this.na
me);
7     menu_item =
this.game_state.prefabs.enemy_units_menu.remove_item(menu_item_i
ndex);
8     menu_item.kill();
9 };
10
11 RPG.PlayerUnit.prototype.kill = function () {
12     "use strict";
13     var menu_item_index, menu_item;
14     Phaser.Sprite.prototype.kill.call(this);
15     // remove from the menu
16     menu_item_index =
this.game_state.prefabs.player_units_menu.find_item_index(this.n
ame);
17     this.game_state.prefabs.player_units_menu.menu_items[menu_
item_index].alpha = 0.5;
18 };
```

Finally, you can try playing the BattleState. Try changing the enemies and player stats to

see if everything is working accordingly.

And we finished the first part of this tutorial series. In the next tutorials we will add a world that the player can explore and find random enemies. We will also improve the battle state, adding more actions and a different turn-based approach based on the units speed.

How to Make a Turn-Based RPG Game in Phaser – Part 2
By Renan Oliveira

In the last tutorial, we created the BattleState for our turn-based RPG. Now, we are going to create a WorldState were the player can explore and eventually find enemies. In addition, we will improve our battle system to consider units speed. The following topics will be covered in this tutorial:

- Creating a WorldState which the player can navigate
- Creating enemy spawners that will initiate the BattleState
- Improve the battle system to consider units speed

To read this tutorial, it is important that you are familiar with the following concepts:

- Javascript and object-oriented concepts.
- Basic Phaser concepts, such as: states, sprites, groups and arcade physics
- Creating maps using Tiled

Assets copyright

The monsters assets used in this tutorial are available in http://opengameart.org/content/2d-rpg-enemy-set and were created by the following artists: Brett Steele (Safir-Kreuz), Joe Raucci (Sylon), Vicki Beinhart (Namakoro), Tyler Olsen (Roots). The characters assets are available in http://opengameart.org/content/24×32-characters-with-faces-big-pack and were created by Svetlana Kushnariova (email: lana-chan@yandex.ru). All assets are available through the Creative Commons license.

Source code files

You can download the tutorial source code files here.

Creating the WorldState

We are going to create a WorldState which will read a Tiled map (in JSON format) and allow the player to navigate through it. The figure below shows the map I created. Since the focus of this tutorial is not on creating Tiled maps, I'm not going into the details of it, and you can create your own or use the one provided in the source code. However, two

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools
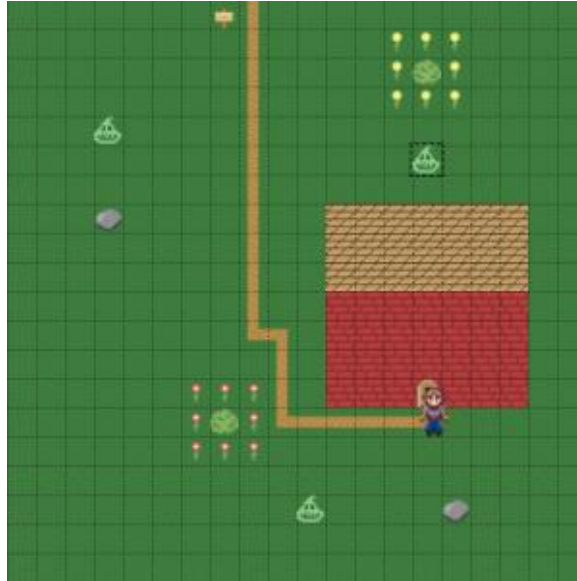
things are important about the map creating, due to the way we are going to create WorldState:

1   Any collidable tile layer must have a collision property set to true.
2   All game prefabs must be defined in the objects layer and each object must contain in its properties at least: name, type, group and texture. Any additional properties must also be defined.

The figures below show the properties of a collision layer and the player object, to illustrate those two conditions.
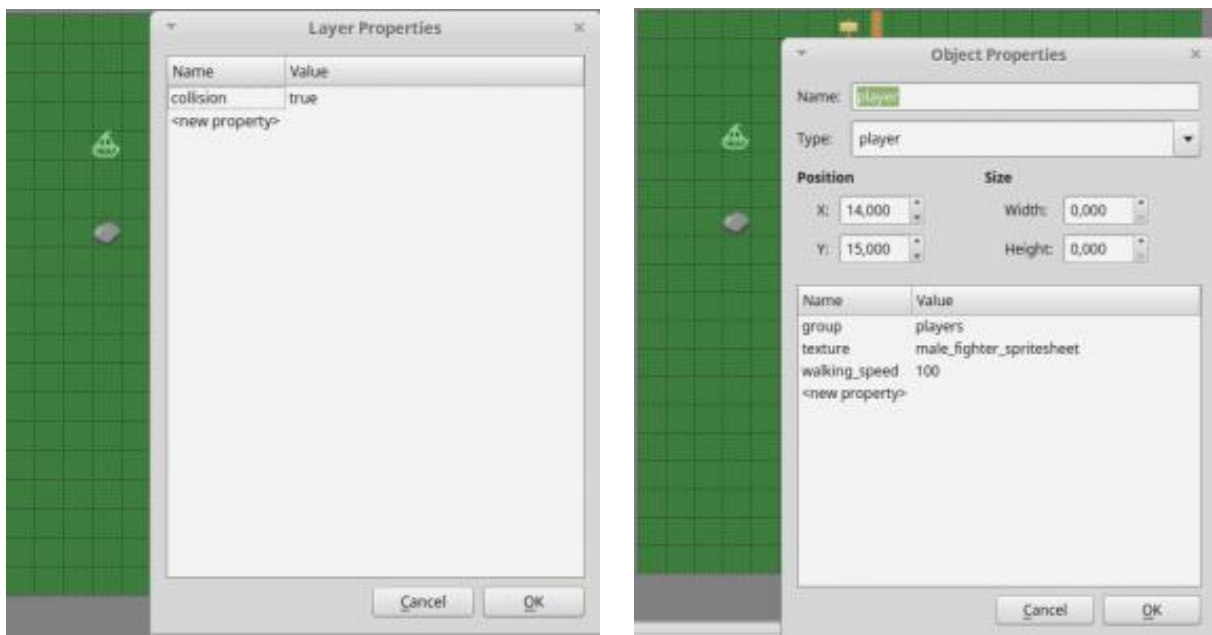
In addition to the Tiled map, WorldState will read another JSON file, such as the one below. Notice that the JSON file must specify the assets, groups and map information.

```
1  {
2      "assets": {
3          "map_tileset": {"type": "image", "source":
"assets/images/open_tileset.png"},
4          "enemy_spawner_image": {"type": "image", "source":
"assets/images/enemy_spawner.png"},
5          "male_fighter_spritesheet": {"type": "spritesheet",
"source": "assets/images/characters/fighter_m.png",
"frame_width": 24, "frame_height": 32},
6
7          "level_tilemap": {"type": "tilemap", "source":
"assets/maps/map1.json"}
8      },
9      "groups": [
10         "players",
11         "enemy_spawners"
12     ],
13     "map": {
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
14          "key": "level_tilemap",
15          "tilesets": ["map_tileset",
"male_fighter_spritesheet"]
16      }
17 }
```

The code below shows the WorldState. The "init" method initializes the physics engine and creates the map form the JSON file. It also creates a "party_data" object which contains the stats of all player units. Notice that this object can be passed as a parameter, which will be done after each battle.

```
1 var RPG = RPG || {};
2
3 RPG.WorldState = function () {
4     "use strict";
5     Phaser.State.call(this);
6
7     this.prefab_classes = {
8         "player": RPG.Player.prototype.constructor,
9         "enemy_spawner": RPG.EnemySpawner.prototype.constructor
10     };
11 };
12
13 RPG.WorldState.prototype =
Object.create(Phaser.State.prototype);
14 RPG.WorldState.prototype.constructor = RPG.WorldState;
15
16 RPG.WorldState.prototype.init = function (level_data,
extra_parameters) {
17     "use strict";
18     var tileset_index;
19     this.level_data = this.level_data || level_data;
20
21     this.scale.scaleMode = Phaser.ScaleManager.SHOW_ALL;
22     this.scale.pageAlignHorizontally = true;
23     this.scale.pageAlignVertically = true;
24
25     // start physics system
26     this.game.physics.startSystem(Phaser.Physics.ARCADE);
27     this.game.physics.arcade.gravity.y = 0;
28
29     // create map and set tileset
30     this.map = this.game.add.tilemap(this.level_data.map.key);
31     tileset_index = 0;
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Page 245

```
32      this.map.tilesets.forEach(function (tileset) {
33          this.map.addTilesetImage(tileset.name,
this.level_data.map.tilesets[tileset_index]);
34          tileset_index += 1;
35      }, this);
36
37      // if no party data is in the parameters, initialize it
with default values
38      this.party_data = extra_parameters.party_data || {
39          "fighter": {
40              "type": "player_unit",
41              "position": {"x": 250, "y": 70},
42              "properties": {
43                  "texture": "male_fighter_spritesheet",
44                  "group": "player_units",
45                  "frame": 10,
46                  "stats": {
47                      "attack": 15,
48                      "defense": 5,
49                      "health": 100,
50                      "speed": 15
51                  }
52              }
53          },
54          "mage": {
55              "type": "player_unit",
56              "position": {"x": 250, "y": 150},
57              "properties": {
58                  "texture": "female_mage_spritesheet",
59                  "group": "player_units",
60                  "frame": 10,
61                  "stats": {
62                      "attack": 20,
63                      "defense": 2,
64                      "health": 100,
65                      "speed": 15
66                  }
67              }
68          }
69      };
70
71      if (extra_parameters.restart_position) {
72          this.player_position = undefined;
73      }
74 };
```

```
75
76 RPG.WorldState.prototype.create = function () {
77     "use strict";
78     var group_name, object_layer, collision_tiles;
79
80     // create map layers
81     this.layers = {};
82     this.map.layers.forEach(function (layer) {
83         this.layers[layer.name] =
this.map.createLayer(layer.name);
84         if (layer.properties.collision) { // collision layer
85             collision_tiles = [];
86             layer.data.forEach(function (data_row) { // find
tiles used in the layer
87                 data_row.forEach(function (tile) {
88                     // check if it's a valid tile index and
isn't already in the list
89                     if (tile.index > 0 &&
collision_tiles.indexOf(tile.index) === -1) {
90                         collision_tiles.push(tile.index);
91                     }
92                 }, this);
93             }, this);
94             this.map.setCollision(collision_tiles, true,
layer.name);
95         }
96     }, this);
97     // resize the world to be the size of the current layer
98     this.layers[this.map.layer.name].resizeWorld();
99
100     // create groups
101     this.groups = {};
102     this.level_data.groups.forEach(function (group_name) {
103         this.groups[group_name] = this.game.add.group();
104     }, this);
105
106     this.prefabs = {};
107
108     for (object_layer in this.map.objects) {
109         if (this.map.objects.hasOwnProperty(object_layer)) {
110             // create layer objects
111             this.map.objects[object_layer].forEach(this.creat
e_object, this);
112         }
113     }
```

```
114
115     // if we came from BattleState, move the player to the
previous position
116     if (this.player_position) {
117         this.prefabs.player.reset(this.player_position.x,
this.player_position.y);
118     }
119 };
120
121 RPG.WorldState.prototype.create_object = function (object) {
122     "use strict";
123     var object_y, position, prefab;
124     // tiled coordinates starts in the bottom left corner
125     object_y = (object.gid) ? object.y - (this.map.tileHeight
/ 2) : object.y + (object.height / 2);
126     position = {"x": object.x + (this.map.tileHeight / 2),
"y": object_y};
127     // create object according to its type
128     if (this.prefab_classes.hasOwnProperty(object.type)) {
129         prefab = new this.prefab_classes[object.type](this,
object.name, position, object.properties);
130     }
131     this.prefabs[object.name] = prefab;
132 };
```

The "create" method must initialize the map layers and game groups and prefabs. The layers were all available in the Tiled map and were already read in the "init" method, so we just have to iterate through them creating each one. However, we must detect the collision layers (by means of the added property) and set all its tiles as collidable.

The groups are easily created by iterating through the ones defined in the level JSON file. However, to create the prefabs we must iterate through each object in the objects layer and instantiate the correct prefab. The prefab instantiation is done in the "create_object" method, which also adjust the prefab position due to the different coordinates used by Tiled. To properly instantiate each prefab, we define a "prefab_classes" property in the constructor that maps each prefab type to its constructor, as we did in the BattleState. As in BattleState, this is possible because all prefabs have the same constructor.

The last piece of code in the "create" method resets the player to a previously saved position. This must be done because the WorldState can be called after a BattleState, so we must keep saved the player previous position. If the battle is lost, we reset the game using the "reset_position" parameter in the "init" method.

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Adding the Player prefab

Now we are going to add the player prefab, which can navigate through the world. The code below shows the Player prefab. In the constructor it must initialize the walking speed, animations, and physics body. Then, in the "update" method, it is controlled by the keyboard arrow keys (obtained from the "cursors" object). We move the player to a given direction if its respective arrow key is pressed and if the player is not already moving to the opposite direction. Also, when the player starts moving to a given direction it plays the corresponding animation. When the player stops, the animation stops and reset its frame to the correct stopped frame, according to its facing direction.

```
1 var RPG = RPG || {};
2
3 RPG.Player = function (game_state, name, position, properties)
{
4     "use strict";
5     RPG.Prefab.call(this, game_state, name, position,
properties);
6
7     this.anchor.setTo(0.5);
8
9     this.walking_speed = +properties.walking_speed;
10
11     this.animations.add("walking_down", [6, 7, 8], 10, true);
12     this.animations.add("walking_left", [9, 10, 11], 10,
true);
13     this.animations.add("walking_right", [3, 4, 5], 10, true);
14     this.animations.add("walking_up", [0, 1, 2], 10, true);
15
16     this.stopped_frames = [7, 10, 4, 1, 7];
17
18     this.game_state.game.physics.arcade.enable(this);
19     this.body.setSize(16, 16, 0, 8);
20     this.body.collideWorldBounds = true;
21
22     this.cursors =
this.game_state.game.input.keyboard.createCursorKeys();
23 };
24
25 RPG.Player.prototype = Object.create(RPG.Prefab.prototype);
26 RPG.Player.prototype.constructor = RPG.Player;
27
```

```
28 RPG.Player.prototype.update = function () {
29     "use strict";
30     this.game_state.game.physics.arcade.collide(this,
this.game_state.layers.collision_back);
31     this.game_state.game.physics.arcade.collide(this,
this.game_state.layers.collision_front);
32
33     if (this.cursors.left.isDown && this.body.velocity.x <= 0)
{
34         // move left
35         this.body.velocity.x = -this.walking_speed;
36         if (this.body.velocity.y === 0) {
37             this.animations.play("walking_left");
38         }
39     } else if (this.cursors.right.isDown &&
this.body.velocity.x >= 0) {
40         // move right
41         this.body.velocity.x = +this.walking_speed;
42         if (this.body.velocity.y === 0) {
43             this.animations.play("walking_right");
44         }
45     } else {
46         this.body.velocity.x = 0;
47     }
48
49     if (this.cursors.up.isDown && this.body.velocity.y <= 0) {
50         // move up
51         this.body.velocity.y = -this.walking_speed;
52         if (this.body.velocity.x === 0) {
53             this.animations.play("walking_up");
54         }
55     } else if (this.cursors.down.isDown &&
this.body.velocity.y >= 0) {
56         // move down
57         this.body.velocity.y = +this.walking_speed;
58         if (this.body.velocity.x === 0) {
59             this.animations.play("walking_down");
60         }
61     } else {
62         this.body.velocity.y = 0;
63     }
64
65     if (this.body.velocity.x === 0 && this.body.velocity.y ===
0) {
66         // stop current animation
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
67          this.animations.stop();
68          this.frame = this.stopped_frames[this.body.facing];
69      }
70 };
```

You can already try moving the player in the WorldState. Check if the collisions are working properly. Notice that we check for collisions with all collision layers in the

update method.

Adding the EnemySpawner prefab

The EnemySpawner will be an immovable prefab that overlaps with the player. When such overlap occur, it will check for possible enemy encounters according to their probabilities. We will start by defining the enemy encounters in the JSON level file, as shown below. Each enemy encounter has a probability and the enemy data.

```
1  "enemy_encounters": [
2          {"probability": 1,
3           "enemy_data": {
4              "bat1": {
5                  "type": "enemy_unit",
6                  "position": {"x": 100, "y": 90},
7                  "properties": {
8                      "texture": "bat_spritesheet",
9                      "group": "enemy_units",
10                     "stats": {
11                         "attack": 10,
12                         "defense": 1,
13                         "health": 30
14                     }
15                 }
16             },
17             "bat2": {
18                 "type": "enemy_unit",
19                 "position": {"x": 100, "y": 170},
20                 "properties": {
21                     "texture": "bat_spritesheet",
22                     "group": "enemy_units",
23                     "stats": {
24                         "attack": 10,
25                         "defense": 1,
26                         "health": 30
27                     }
28                 }
29             }
30         }
31     }
32 ]
```

Now, we can create the EnemySpawner to check for one of the possible encounters. Its code is shown below. In the "update" method we check for overlaps with the player and call the "check_for_spawn" method when an overlap occurs. Notice that, to call this

method only once for each overlap we use the "overlapping" variable, and check for spawns only when it was false.

```javascript
1 var RPG = RPG || {};
2
3 RPG.EnemySpawner = function (game_state, name, position,
properties) {
4     "use strict";
5     RPG.Prefab.call(this, game_state, name, position,
properties);
6
7     this.game_state.game.physics.arcade.enable(this);
8     this.body.immovable = true;
9
10    this.overlapping = true;
11 };
12
13 RPG.EnemySpawner.prototype =
Object.create(RPG.Prefab.prototype);
14 RPG.EnemySpawner.prototype.constructor = RPG.EnemySpawner;
15
16 RPG.EnemySpawner.prototype.update = function () {
17     "use strict";
18     this.overlapping =
this.game_state.game.physics.arcade.overlap(this,
this.game_state.groups.players, this.check_for_spawn, null,
this);
19 };
20
21 RPG.EnemySpawner.prototype.check_for_spawn = function () {
22     "use strict";
23     var spawn_chance, encounter_index, enemy_encounter;
24     // check for spawn only once for overlap
25     if (!this.overlapping) {
26         spawn_chance = this.game_state.game.rnd.frac();
27         // check if the enemy spawn probability is less than
the generated random number for each spawn
28         for (encounter_index = 0; encounter_index <
this.game_state.level_data.enemy_encounters.length;
encounter_index += 1) {
29             enemy_encounter =
this.game_state.level_data.enemy_encounters[encounter_index];
30             if (spawn_chance <= enemy_encounter.probability) {
31                 // save current player position for later
```

```
32                      this.game_state.player_position =
this.game_state.prefabs.player.position;
33                      // call battle state
34                      this.game_state.game.state.start("BootState",
false, false, "assets/levels/battle.json", "BattleState",
{enemy_data: enemy_encounter.enemy_data, party_data:
this.game_state.party_data});
35                      break;
36                  }
37              }
38          }
39  };
```

The "check_for_spawn" method generates a random number using Phaser random data generator and compares it with the enemy encounters probabilities, choosing the first one whose probability is higher than the generated number. Notice that, for this to work the encounters must be sorted in ascending order of probability, prioritizing less likely encounters. When an encounter occurs, it calls BattleState with the correct enemy data and player party data.

Updating BattleState

Finally, we must update our BattleState to work with our last modifications. Instead of reading the enemy and player units from a JSON file, they will be passed as parameters in the "init" method. Then, we just have to iterate through them in the "create" method and create all their prefabs, using the same "create_prefab" method. Notice that the enemy units were stored in the enemy encounters data from the WorldState, while the player units were stored in the "party_data" variable from WorldState (shown before).

Now, we must properly go back to WorldState when the battle is finished. In the "next_turn" method, before making the next unit act, we check if there are remaining enemy and player units. If there are no remaining enemy units, we call an "end_battle" method and, if there are no remaining player units we call a "game_over" method.

The "end_battle" method will switch back to WorldState updating the "party_data" to reflect this battle. So, we must iterate through all player units saving their stats in the "party_data" variable. On the other hand, the "game_over" method will switch back to WorldState without sending any "party_data", which will reset it. Also, we must tell the WorldState to restart the player position, instead of keeping the last one as showed in the WorldState code. The code below shows the modifications in the BattleState.

```
1  var RPG = RPG || {};
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Page 254

```
2
3 RPG.BattleState = function () {
4     "use strict";
5     Phaser.State.call(this);
6
7     this.prefab_classes = {
8         "background": RPG.TilePrefab.prototype.constructor,
9         "rectangle": RPG.Prefab.prototype.constructor,
10         "player_unit": RPG.PlayerUnit.prototype.constructor,
11         "enemy_unit": RPG.EnemyUnit.prototype.constructor
12     };
13
14     this.TEXT_STYLE = {font: "14px Arial", fill: "#FFFFFF"};
15 };
16
17 RPG.BattleState.prototype =
Object.create(Phaser.State.prototype);
18 RPG.BattleState.prototype.constructor = RPG.BattleState;
19
20 RPG.BattleState.prototype.init = function (level_data,
extra_parameters) {
21     "use strict";
22     this.level_data = level_data;
23     this.enemy_data = extra_parameters.enemy_data;
24     this.party_data = extra_parameters.party_data;
25
26     this.scale.scaleMode = Phaser.ScaleManager.SHOW_ALL;
27     this.scale.pageAlignHorizontally = true;
28     this.scale.pageAlignVertically = true;
29 };
30
31 RPG.BattleState.prototype.create = function () {
32     "use strict";
33     var group_name, prefab_name, player_unit_name,
enemy_unit_name;
34
35     // create groups
36     this.groups = {};
37     this.level_data.groups.forEach(function (group_name) {
38         this.groups[group_name] = this.game.add.group();
39     }, this);
40
41     // create prefabs
42     this.prefabs = {};
43     for (prefab_name in this.level_data.prefabs) {
```

```
44          if
(this.level_data.prefabs.hasOwnProperty(prefab_name)) {
45              // create prefab
46              this.create_prefab(prefab_name,
this.level_data.prefabs[prefab_name]);
47          }
48      }
49
50      // create enemy units
51      for (enemy_unit_name in this.enemy_data) {
52          if (this.enemy_data.hasOwnProperty(enemy_unit_name)) {
53              // create enemy units
54              this.create_prefab(enemy_unit_name,
this.enemy_data[enemy_unit_name]);
55          }
56      }
57
58      // create player units
59      for (player_unit_name in this.party_data) {
60          if (this.party_data.hasOwnProperty(player_unit_name))
{
61              // create player units
62              this.create_prefab(player_unit_name,
this.party_data[player_unit_name]);
63          }
64      }
65
66      this.init_hud();
67
68      this.units = [];
69      this.units =
this.units.concat(this.groups.player_units.children);
70      this.units =
this.units.concat(this.groups.enemy_units.children);
71
72      this.next_turn();
73 };
74
75 RPG.BattleState.prototype.next_turn = function () {
76      "use strict";
77      // if all enemy units are dead, go back to the world state
78      if (this.groups.enemy_units.countLiving() === 0) {
79          this.end_battle();
80      }
81
```

```
82     // if all player units are dead, restart the game
83     if (this.groups.player_units.countLiving() === 0) {
84         this.game_over();
85     }
86
87     // takes the next unit
88     this.current_unit = this.units.shift();
89     // if the unit is alive, it acts, otherwise goes to the
next turn
90     if (this.current_unit.alive) {
91         this.current_unit.act();
92         this.units.push(this.current_unit);
93     } else {
94         this.next_turn();
95     }
96 };
97
98 RPG.BattleState.prototype.game_over = function () {
99     "use strict";
100     // go back to WorldState restarting the player position
101     this.game.state.start("BootState", true, false,
"assets/levels/level1.json", "WorldState", {restart_position:
true});
102 };
103
104 RPG.BattleState.prototype.end_battle = function () {
105     "use strict";
106     // save current party health
107     this.groups.player_units.forEach(function (player_unit) {
108         this.party_data[player_unit.name].properties.stats =
player_unit.stats;
109     }, this);
110     // go back to WorldState with the current party data
111     this.game.state.start("BootState", true, false,
"assets/levels/level1.json", "WorldState", {party_data:
this.party_data});
112 };
```

By now, you can already try playing with the EnemySpawners.

Considering units speed for the turns

In this tutorial, each unit will have a speed stat, which will be used to calculate the next turn that unit will act. The code below shows the modifications in the Unit prefab, and how it calculates the next act turn based on the current turn and its speed. Notice that I used an arbitrary rule for calculating the next turn, and you can use the one you finds best.

```
1 var RPG = RPG || {};
2
3 RPG.Unit = function (game_state, name, position, properties) {
4     "use strict";
5     RPG.Prefab.call(this, game_state, name, position,
properties);
6
7     this.anchor.setTo(0.5);
8
```

```
9    this.stats = Object.create(properties.stats);
10
11    this.attacked_animation =
this.game_state.game.add.tween(this);
12    this.attacked_animation.to({tint: 0xFF0000}, 200);
13    this.attacked_animation.onComplete.add(this.restore_tint,
this);
14
15    this.act_turn = 0;
16 };
17
18 RPG.Unit.prototype = Object.create(RPG.Prefab.prototype);
19 RPG.Unit.prototype.constructor = RPG.Unit;
20
21 RPG.Unit.prototype.calculate_act_turn = function
(current_turn) {
22    "use strict";
23    // calculate the act turn based on the unit speed
24    this.act_turn = current_turn + Math.ceil(100 /
this.stats.speed);
25 }
```

Now, we will change the BattleState to store the units in a priority queue, instead of an array. A priority queue is a data structure where all elements are always sorted, given a sorting criteria (if you're not familiar with priority queues, check this wikipedia link). In our case, the sorting criteria will be the unit next acting turn, which means the first unit from the queue is the one that will act earlier. Since the priority queue is a well known data structure, we are going to use the implementation provided by Adam Hooper instead of creating our own.

The code below shows the modifications in the BattleState to use the priority queue. First, in the end of the "create" method we initialize "units" as a priority queue which compares the units act turn, and add all units to the queue, calculating their first acting turns. Then, in the "next_turn" method, we must update the current unit act turn before adding it to the "units" queue again, so it will be put in the correct position.

```
1 RPG.BattleState.prototype.create = function () {
2    "use strict";
3    var group_name, prefab_name, player_unit_name,
enemy_unit_name;
4
5    // create groups
6    this.groups = {};
```

```
7      this.level_data.groups.forEach(function (group_name) {
8          this.groups[group_name] = this.game.add.group();
9      }, this);
10
11     // create prefabs
12     this.prefabs = {};
13     for (prefab_name in this.level_data.prefabs) {
14         if
(this.level_data.prefabs.hasOwnProperty(prefab_name)) {
15             // create prefab
16             this.create_prefab(prefab_name,
this.level_data.prefabs[prefab_name]);
17         }
18     }
19
20     // create enemy units
21     for (enemy_unit_name in this.enemy_data) {
22         if (this.enemy_data.hasOwnProperty(enemy_unit_name)) {
23             // create enemy units
24             this.create_prefab(enemy_unit_name,
this.enemy_data[enemy_unit_name]);
25         }
26     }
27
28     // create player units
29     for (player_unit_name in this.party_data) {
30         if (this.party_data.hasOwnProperty(player_unit_name))
{
31             // create player units
32             this.create_prefab(player_unit_name,
this.party_data[player_unit_name]);
33         }
34     }
35
36     this.init_hud();
37
38     // store units in a priority queue which compares the
units act turn
39     this.units = new PriorityQueue({comparator: function
(unit_a, unit_b) {
40         return unit_a.act_turn - unit_b.act_turn;
41     }});
42     this.groups.player_units.forEach(function (unit) {
43         unit.calculate_act_turn(0);
44         this.units.queue(unit);
```

```
45     }, this);
46     this.groups.enemy_units.forEach(function (unit) {
47          unit.calculate_act_turn(0);
48          this.units.queue(unit);
49     }, this);
50
51     this.next_turn();
52 };
53
54 RPG.BattleState.prototype.next_turn = function () {
55     "use strict";
56     // if all enemy units are dead, go back to the world state
57     if (this.groups.enemy_units.countLiving() === 0) {
58          this.end_battle();
59     }
60
61     // if all player units are dead, restart the game
62     if (this.groups.player_units.countLiving() === 0) {
63          this.game_over();
64     }
65
66     // takes the next unit
67     this.current_unit = this.units.dequeue();
68     // if the unit is alive, it acts, otherwise goes to the
next turn
69     if (this.current_unit.alive) {
70          this.current_unit.act();
71          this.current_unit.calculate_act_turn(this.current_unit
.act_turn);
72          this.units.queue(this.current_unit);
73     } else {
74          this.next_turn();
75     }
76 };
```

Now, you can already try setting some speed values and see if everything is working correctly. Below is the final enemy encounters and party data I used.

```
1 "enemy_encounters": [
2        {"probability": 0.3,
3         "enemy_data": {
4              "lizard1": {
5                   "type": "enemy_unit",
6                   "position": {"x": 100, "y": 100},
```

```
7                        "properties": {
8                            "texture": "lizard_spritesheet",
9                            "group": "enemy_units",
10                           "stats": {
11                                "attack": 30,
12                                "defense": 10,
13                                "health": 50,
14                                "speed": 15
15                           }
16                       }
17                   }
18               }
19           },
20           {"probability": 0.5,
21            "enemy_data": {
22                "bat1": {
23                    "type": "enemy_unit",
24                    "position": {"x": 100, "y": 90},
25                    "properties": {
26                        "texture": "bat_spritesheet",
27                        "group": "enemy_units",
28                        "stats": {
29                            "attack": 10,
30                            "defense": 1,
31                            "health": 30,
32                            "speed": 20
33                        }
34                    }
35                },
36                "bat2": {
37                    "type": "enemy_unit",
38                    "position": {"x": 100, "y": 170},
39                    "properties": {
40                        "texture": "bat_spritesheet",
41                        "group": "enemy_units",
42                        "stats": {
43                            "attack": 10,
44                            "defense": 1,
45                            "health": 30,
46                            "speed": 20
47                        }
48                    }
49                }
50           }
51       },
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
52          {"probability": 1.0,
53           "enemy_data": {
54              "scorpion1": {
55                  "type": "enemy_unit",
56                  "position": {"x": 100, "y": 50},
57                  "properties": {
58                      "texture": "scorpion_spritesheet",
59                      "group": "enemy_units",
60                      "stats": {
61                          "attack": 15,
62                          "defense": 1,
63                          "health": 20,
64                          "speed": 10
65                      }
66                  }
67              },
68              "scorpion2": {
69                  "type": "enemy_unit",
70                  "position": {"x": 100, "y": 100},
71                  "properties": {
72                      "texture": "scorpion_spritesheet",
73                      "group": "enemy_units",
74                      "stats": {
75                          "attack": 15,
76                          "defense": 1,
77                          "health": 20,
78                          "speed": 10
79                      }
80                  }
81              },
82              "scorpion3": {
83                  "type": "enemy_unit",
84                  "position": {"x": 100, "y": 150},
85                  "properties": {
86                      "texture": "scorpion_spritesheet",
87                      "group": "enemy_units",
88                      "stats": {
89                          "attack": 15,
90                          "defense": 1,
91                          "health": 20,
92                          "speed": 10
93                      }
94                  }
95              }
96          }
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
 97              }
 98      ]
 99
100  this.party_data = extra_parameters.party_data || {
101          "fighter": {
102              "type": "player_unit",
103              "position": {"x": 250, "y": 50},
104              "properties": {
105                  "texture": "male_fighter_spritesheet",
106                  "group": "player_units",
107                  "frame": 10,
108                  "stats": {
109                      "attack": 15,
110                      "defense": 5,
111                      "health": 100,
112                      "speed": 15
113                  }
114              }
115          },
116          "mage": {
117              "type": "player_unit",
118              "position": {"x": 250, "y": 100},
119              "properties": {
120                  "texture": "female_mage_spritesheet",
121                  "group": "player_units",
122                  "frame": 10,
123                  "stats": {
124                      "attack": 20,
125                      "defense": 2,
126                      "health": 100,
127                      "speed": 10
128                  }
129              }
130          },
131          "ranger": {
132              "type": "player_unit",
133              "position": {"x": 250, "y": 150},
134              "properties": {
135                  "texture": "female_ranger_spritesheet",
136                  "group": "player_units",
137                  "frame": 10,
138                  "stats": {
139                      "attack": 10,
140                      "defense": 3,
141                      "health": 100,
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
142                              "speed": 20
143                         }
144                    }
145              }
146       };
```

And now we finished this tutorial. In the next one we are going to add different actions during the battle, such as using magic and items. In addition, the player units will receive experience after each battle, being able to pass levels.

How to Make a Turn-Based RPG Game in Phaser – Part 3
By Renan Oliveira

In the last tutorial we added a WorldState where the player can navigate and linked it with the BattleState created in the first tutorial. Now, we are going to improve our BattleState, to include the following:

- Battle reward including experience and items
- A level system based on an experience table
- The possibility of using items during the battle
- Magic attacks along with the current physical attacks

To read this tutorial, it is important that you are familiar with the following concepts:
- Javascript and object-oriented concepts.
- Basic Phaser concepts, such as: states, sprites, groups and arcade physics
- Creating maps using Tiled

Assets copyright

The monsters assets used in this tutorial are available in http://opengameart.org/content/2d-rpg-enemy-set and were created by the following artists: Brett Steele (Safir-Kreuz), Joe Raucci (Sylon), Vicki Beinhart (Namakoro), Tyler Olsen (Roots). The characters assets are available in http://opengameart.org/content/24×32-characters-with-faces-big-pack and were created by Svetlana Kushnariova (email: lana-chan@yandex.ru). All assets are available through the Creative Commons license.

Source code files

You can download the tutorial source code files here.

Adding the battle reward

We will start by adding a battle reward for each enemy encounter. To simplify things, the reward will be always the same, but you can add multiple rewards with different probabilities using a strategy similar to the enemy encounters.

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Page 266

The reward will be described in the level JSON file with the enemy encounters data, as shown below. Notice that we are describing the experience, which will be used to increase the player units levels and the items that will be obtained from that encounter.

```
1  "enemy_encounters": [
2          {"probability": 0.3,
3           "enemy_data": {
4              "lizard1": {
5                  "type": "enemy_unit",
6                  "position": {"x": 100, "y": 100},
7                  "properties": {
8                      "texture": "lizard_spritesheet",
9                      "group": "enemy_units",
10                     "stats": {
11                         "attack": 30,
12                         "defense": 10,
13                         "health": 50,
14                         "speed": 15
15                     }
16                 }
17             }
18         },
19         "reward": {
20             "experience": 100,
21             "items": [{"type": "potion", "properties":
   {"group": "items", "health_power": 50}}]
22         }
23     },
24         {"probability": 0.5,
25          "enemy_data": {
26             "bat1": {
27                 "type": "enemy_unit",
28                 "position": {"x": 100, "y": 90},
29                 "properties": {
30                     "texture": "bat_spritesheet",
31                     "group": "enemy_units",
32                     "stats": {
33                         "attack": 10,
34                         "defense": 1,
35                         "health": 30,
36                         "speed": 20
37                     }
38                 }
39             },
40             "bat2": {
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
41                    "type": "enemy_unit",
42                    "position": {"x": 100, "y": 170},
43                    "properties": {
44                        "texture": "bat_spritesheet",
45                        "group": "enemy_units",
46                        "stats": {
47                            "attack": 10,
48                            "defense": 1,
49                            "health": 30,
50                            "speed": 20
51                        }
52                    }
53                }
54            },
55          "reward": {
56              "experience": 100,
57              "items": [{"type": "potion", "properties":
{"group": "items", "health_power": 50}}]
58          }
59        },
60        {"probability": 1.0,
61         "enemy_data": {
62            "scorpion1": {
63                "type": "enemy_unit",
64                "position": {"x": 100, "y": 50},
65                "properties": {
66                    "texture": "scorpion_spritesheet",
67                    "group": "enemy_units",
68                    "stats": {
69                        "attack": 15,
70                        "defense": 1,
71                        "health": 20,
72                        "speed": 10
73                    }
74                }
75            },
76            "scorpion2": {
77                "type": "enemy_unit",
78                "position": {"x": 100, "y": 100},
79                "properties": {
80                    "texture": "scorpion_spritesheet",
81                    "group": "enemy_units",
82                    "stats": {
83                        "attack": 15,
84                        "defense": 1,
```

```
85                              "health": 20,
86                              "speed": 10
87                          }
88                      }
89                  },
90              "scorpion3": {
91                  "type": "enemy_unit",
92                  "position": {"x": 100, "y": 150},
93                  "properties": {
94                      "texture": "scorpion_spritesheet",
95                      "group": "enemy_units",
96                      "stats": {
97                          "attack": 15,
98                          "defense": 1,
99                          "health": 20,
100                         "speed": 10
101                     }
102                 }
103             }
104         },
105     "reward": {
106         "experience": 100,
107         "items": [{"type": "potion", "properties":
{"group": "items", "health_power": 50}}]
108     }
109 }
```

Increasing the player units experience

We will use the same experience table for all player units, which will define the
necessary experience to reach each level and the correspondent stats increase. This table
is defined in a JSON file as shown below. Notice that I used the same stats increase for
all levels without considering the game balance, but you should use the ones you find
best.

```
1 [
2    {"required_exp": 100, "stats_increase": {"attack": 1,
"defense": 1, "speed": 1, "health": 10} },
3    {"required_exp": 200, "stats_increase": {"attack": 1,
"defense": 1, "speed": 1, "health": 10} },
4    {"required_exp": 300, "stats_increase": {"attack": 1,
"defense": 1, "speed": 1, "health": 10} },
5    {"required_exp": 400, "stats_increase": {"attack": 1,
"defense": 1, "speed": 1, "health": 10} },
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
6    {"required_exp": 500, "stats_increase": {"attack": 1,
"defense": 1, "speed": 1, "health": 10} },
7    {"required_exp": 600, "stats_increase": {"attack": 1,
"defense": 1, "speed": 1, "health": 10} }
8 ]
```

This experience table must be loaded in the "preload" method of BattleState, and initialized in its "create" method, as shown below.

```
1 RPG.BattleState.prototype.preload = function () {
2    "use strict";
3    this.load.text("experience_table",
"assets/levels/experience_table.json");
4 };
5
6 // save experience table
7    this.experience_table =
JSON.parse(this.game.cache.getText("experience_table"));
```

To add experience and levels to the player units we will add a "receive_experience" method for the PlayerUnit prefab. This method is shown in the code below and increases the unit experience. Then it checks if the current experience is enough to reach the next level. This is done using the experience table loaded in the game state. When a new level is reached, the stats are increased according to what is defined in the experience table. Also, the experience is reset to zero when a new level is reached.

```
1 RPG.PlayerUnit.prototype.receive_experience = function
(experience) {
2    "use strict";
3    var next_level_data, stat;
4    // increase experience
5    this.stats.experience += experience;
6    next_level_data =
this.game_state.experience_table[this.stats.current_level];
7    // if current experience is greater than the necessary to
the next level, the unit gains a level
8    if (this.stats.experience >= next_level_data.required_exp)
{
9        this.stats.current_level += 1;
10       this.stats.experience = 0;
11       // increase unit stats according to new level
12       for (stat in next_level_data.stats_increase) {
```

```
13                if
(next_level_data.stats_increase.hasOwnProperty(stat)) {
14                    this.stats[stat] +=
next_level_data.stats_increase[stat];
15                }
16            }
17        }
18 };
```

Finally, we must add the experience reward from each encounter in the "end_battle" method, as shown below. This can be done by dividing the experience for each player unit, and calling the "receive_experience" method for each one.

```
1 RPG.BattleState.prototype.end_battle = function () {
2    "use strict";
3    var received_experience;
4
5    // receive battle reward
6    received_experience = this.encounter.reward.experience;
7    this.groups.player_units.forEach(function (player_unit) {
8        // receive experience from enemy
9        player_unit.receive_experience(received_experience /
this.groups.player_units.children.length);
10        // save current party stats
11        this.party_data[player_unit.name].properties.stats =
player_unit.stats;
12    }, this);
13
14    // go back to WorldState with the current party data
15    this.game.state.start("BootState", true, false,
"assets/levels/level1.json", "WorldState", {party_data:
this.party_data});
16 };
```

Now, you can already try playing and check if the player units are receiving the correct

bat1 bat2	Attack	fighter 98 mage 110 ranger 110

amount of experience during the battles.

Adding items and an inventory

We will create Inventory and Item prefabs to represent the game items the player can carry and use during the battles. After we finish creating the items, we will add menus to the BattleState to allow the player to use those items.

First, the Inventory prefab is shown below. It will have a list of items and methods to collect and use items. The "collect_item" receives an object with the item type and properties, and instantiate the appropriate item prefab, adding it to the list. The "use_item" method consumes the item, removing it from the items array.

```
1 var RPG = RPG || {};
2
3 RPG.Inventory = function (game_state, name, position,
properties) {
4    "use strict";
```

```
5     RPG.Prefab.call(this, game_state, name, position,
properties);
6
7     this.item_classes = {
8         "potion": RPG.Potion.prototype.constructor
9     };
10
11     this.items = [];
12 };
13
14 RPG.Inventory.prototype =
Object.create(RPG.Prefab.prototype);
15 RPG.Inventory.prototype.constructor = RPG.Inventory;
16
17 RPG.Inventory.prototype.collect_item = function (item_object)
{
18     "use strict";
19     var item;
20     // create item prefab
21     item = new
this.item_classes[item_object.type](this.game_state,
item_object.type + this.items.length, {x: 0, y: 0},
item_object.properties);
22     this.items.push(item);
23 };
24
25 RPG.Inventory.prototype.use_item = function (item_name,
target) {
26     "use strict";
27     var item_index;
28     // remove item from items list
29     for (item_index = 0; item_index < this.items.length;
item_index += 1) {
30         if (this.items[item_index].name === item_name) {
31             this.items[item_index].use(target);
32             this.items.splice(item_index, 1);
33             break;
34         }
35     }
36 };
```

Now, we are going to create the item prefabs. First, we will create a generic Item prefab, which all items will extend. Initially, this prefab will only have an "use" method which destroys itself.

```
1 var RPG = RPG || {};
2
3 RPG.Item = function (game_state, name, position, properties) {
4     "use strict";
5     RPG.Prefab.call(this, game_state, name, position,
properties);
6 };
7
8 RPG.Item.prototype = Object.create(RPG.Prefab.prototype);
9 RPG.Item.prototype.constructor = RPG.Item;
10
11 RPG.Item.prototype.use = function () {
12     "use strict";
13     // by default the item is destroyed
14     this.kill();
15 };
```

After creating the Item prefab, we can create a Potion prefab, as shown below. The potion will have a health power, and must implement the "use" method to increase the health of the target player unit.

```
1 var RPG = RPG || {};
2
3 RPG.Potion = function (game_state, name, position, properties)
{
4     "use strict";
5     RPG.Item.call(this, game_state, name, position,
properties);
6
7     this.health_power = properties.health_power;
8 };
9
10 RPG.Potion.prototype = Object.create(RPG.Item.prototype);
11 RPG.Potion.prototype.constructor = RPG.Potion;
12
13 RPG.Potion.prototype.use = function (target) {
14     "use strict";
15     RPG.Item.prototype.use.call(this);
16     target.stats.health += this.health_power;
17 };
```

Finally, in the "end_battle" method we must collect the items obtained from defeating the enemies. This can be done by iterating through all the items in the reward and calling the "collect_item" from the inventory.

```
1  RPG.BattleState.prototype.end_battle = function () {
2      "use strict";
3      var received_experience;
4
5      // receive battle reward
6      received_experience = this.encounter.reward.experience;
7      this.groups.player_units.forEach(function (player_unit) {
8          // receive experience from enemy
9          player_unit.receive_experience(received_experience /
this.groups.player_units.children.length);
10         // save current party stats
11         this.party_data[player_unit.name].properties.stats =
player_unit.stats;
12     }, this);
13
14
15     this.encounter.reward.items.forEach(function (item_object)
{
16         this.prefabs.inventory.collect_item(item_object);
17     }, this);
18
19     // go back to WorldState with the current party data
20     this.game.state.start("BootState", true, false,
"assets/levels/level1.json", "WorldState", {party_data:
this.party_data, inventory: this.prefabs.inventory});
21 };
```

Now, you can already try playing and check if the items are being correctly collected. Since there are no gameplay difference yet, you would have to print some information in the console to check this.

Using the items during the battle

To use the items, we will add an item action to the actions menu. When it is selected, it will show the available items to be used. Since we must alternate through menus, we will add methods to hide and show in the Menu prefab, as shown below.

```
1  RPG.Menu.prototype.show = function () {
```

```
2      "use strict";
3      this.menu_items.forEach(function (menu_item) {
4          menu_item.visible = true;
5      }, this);
6  };
7
8  RPG.Menu.prototype.hide = function () {
9      "use strict";
10     this.menu_items.forEach(function (menu_item) {
11          menu_item.visible = false;
12     }, this);
13 };
```

Now, we must create the items menu and add the Item action in the actions menu. The items menu will be created in the Inventory prefab, by iterating through all its items to create menu items, in a similar way as done to create the units menus. To add the Item action, we must change the "show_player_actions" method in BattleState to add it, as shown below.

```
1  RPG.Inventory.prototype.create_menu = function (position) {
2      "use strict";
3      var menu_items, item_index, item, menu_item, items_menu;
4      // create units menu items
5      item_index = 0;
6      menu_items = [];
7      for (item_index = 0; item_index < this.items.length;
item_index += 1) {
8          item = this.items[item_index];
9          menu_item = new RPG.ItemMenuItem(this.game_state,
item.name + "_menu_item", {x: position.x, y: position.y +
item_index * 20}, {group: "hud", text: item.name, style:
Object.create(this.game_state.TEXT_STYLE)});
10         menu_items.push(menu_item);
11     }
12     // create units menu
13     items_menu = new RPG.Menu(this.game_state, "items_menu",
position, {group: "hud", menu_items: menu_items});
14     items_menu.hide();
15 };
16
17 RPG.BattleState.prototype.show_player_actions = function
(position) {
18     "use strict";
```

```
19     var actions, actions_menu_items, action_index,
actions_menu;
20     // available actions
21     actions = [{text: "Attack", item_constructor:
RPG.AttackMenuItem.prototype.constructor},
22            {text: "Item", item_constructor:
RPG.InventoryMenuItem.prototype.constructor}];
23     actions_menu_items = [];
24     action_index = 0;
25     // create a menu item for each action
26     actions.forEach(function (action) {
27         actions_menu_items.push(new
action.item_constructor(this, action.text + "_menu_item", {x:
position.x, y: position.y + action_index * 20}, {group: "hud",
text: action.text, style: Object.create(this.TEXT_STYLE)}));
28         action_index += 1;
29     }, this);
30     actions_menu = new RPG.Menu(this, "actions_menu",
position, {group: "hud", menu_items: actions_menu_items});
31 };
```

Notice that in those menus we used two new menu item prefabs: ItemMenuItem, used to select items to use and IventoryMenuItem, used to select the Item action. Both prefabs are shown below. The ItemMenuItem must disable the items menu and enable the player units menu to select the player unit in which the selected item will be used. On the other hand, the InventoryMenuItem must disable and hide the actions menu, while showing and enabling the items menu. Notice that the IventoryMenuItem can be selected only when there are remaining items.

```
1 var RPG = RPG || {};
2
3 RPG.ItemMenuItem = function (game_state, name, position,
properties) {
4     "use strict";
5     RPG.MenuItem.call(this, game_state, name, position,
properties);
6 };
7
8 RPG.ItemMenuItem.prototype =
Object.create(RPG.MenuItem.prototype);
9 RPG.ItemMenuItem.prototype.constructor = RPG.ItemMenuItem;
10
11 RPG.ItemMenuItem.prototype.select = function () {
12     "use strict";
```

```
13     // disable actions menu
14     this.game_state.prefabs.items_menu.disable();
15     // enable player units menu so the player can choose the
target
16     this.game_state.prefabs.player_units_menu.enable();
17     // save selected item
18     this.game_state.current_item = this.text;
19 };
```

```
1 var RPG = RPG || {};
2
3 RPG.InventoryMenuItem = function (game_state, name, position,
properties) {
4     "use strict";
5     RPG.MenuItem.call(this, game_state, name, position,
properties);
6 };
7
8 RPG.InventoryMenuItem.prototype =
Object.create(RPG.MenuItem.prototype);
9 RPG.InventoryMenuItem.prototype.constructor =
RPG.InventoryMenuItem;
10
11 RPG.InventoryMenuItem.prototype.select = function () {
12     "use strict";
13     // select only if there are remaining items
14     if (this.game_state.prefabs.inventory.items.length > 0) {
15         // disable actions menu
16         this.game_state.prefabs.actions_menu.disable();
17         this.game_state.prefabs.actions_menu.hide();
18         // enable enemy units menu so the player can choose
the target
19         this.game_state.prefabs.items_menu.show();
20         this.game_state.prefabs.items_menu.enable();
21     }
22 };
```

There are still some more things we must change. First, we must change the
PlayerMenuItem "select" method to use the current selected item. Second, we must
implement the "kill" method from the Item prefab to remove it from the items menu.
Both modifications are shown below.

```
1 RPG.PlayerMenuItem.prototype.select = function () {
2     "use strict";
```

```
3     var player_unit;
4     // get selected player unit
5     player_unit = this.game_state.prefabs[this.text];
6     // use current selected item on selected unit
7     this.game_state.prefabs.inventory.use_item(this.game_state.
current_item, player_unit);
8
9     // show actions menu again
10      this.game_state.prefabs.items_menu.disable();
11      this.game_state.prefabs.items_menu.hide();
12      this.game_state.prefabs.actions_menu.show();
13      this.game_state.prefabs.actions_menu.enable();
14  };
15
16  RPG.Item.prototype.kill = function () {
17      "use strict";
18      Phaser.Sprite.prototype.kill.call(this);
19      var menu_item_index, menu_item;
20      // remove item from the menu
21      menu_item_index =
this.game_state.prefabs.items_menu.find_item_index(this.name);
22      menu_item =
this.game_state.prefabs.items_menu.remove_item(menu_item_index);
23      menu_item.kill();
24  };
```

Finally, we must keep the inventory through the BattleState and WorldState. This can be done by adding a new parameter in the "init" method of both states that saves the inventory and which will be used in every state transition. In addition, if there is no inventory created yet (at the beginning of the game), we must create an empty one in the "create" method of BattleState.

Now, you can already try playing using items. Remember to check if all menus are
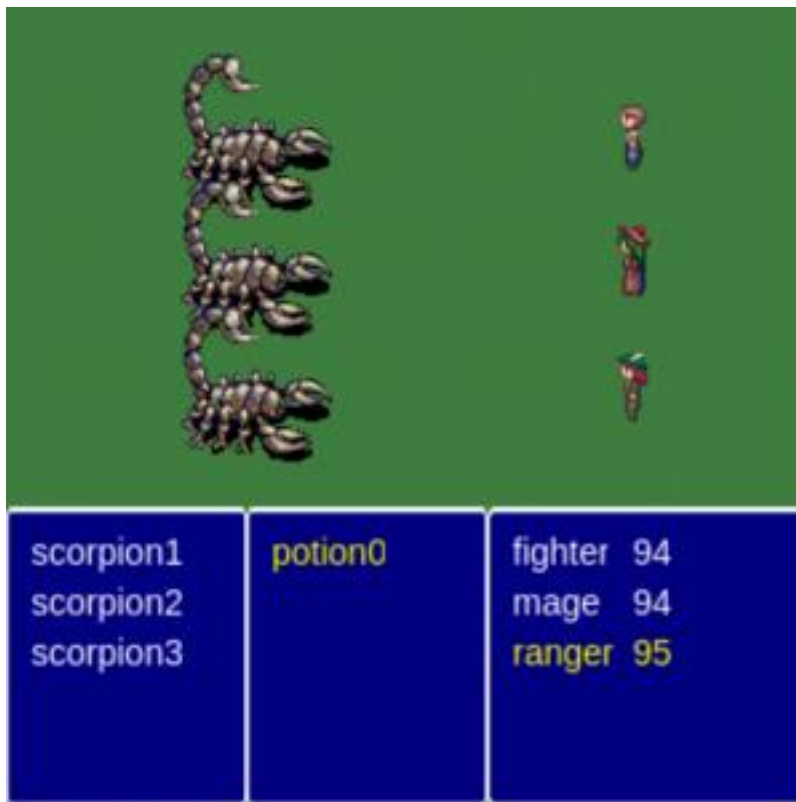
working correctly and if the item rewards are being updated properly.

Adding magic attacks

Our last modification will add a new possible attack for each unit, called magic attack. Those attacks will be based on a new stat and will have increased damage, but will consume mana.

To do that, first we must change our attack code to put it in a new Attack prefab, as shown below. The Attack prefab will have an owner unit, which represents the unit which is attacking and initially contains a "show_message" method, which will show the attack message, as we were already doing. Now, we are going to create a PhysicalAttack and a MagicAttack prefabs which will extend this one and implement a "hit" method. This method will execute the attack.

```
1 var RPG = RPG || {};
2
```

```
3 RPG.Attack = function (game_state, name, position, properties)
{
4    "use strict";
5    RPG.Prefab.call(this, game_state, name, position,
properties);
6
7    this.owner =
this.game_state.prefabs[properties.owner_name];
8 };
9
10 RPG.Attack.prototype = Object.create(RPG.Prefab.prototype);
11 RPG.Attack.prototype.constructor = RPG.Attack;
12
13 RPG.Attack.prototype.show_message = function (target, damage)
{
14    "use strict";
15    var action_message_position, action_message_text,
attack_message;
16    // show attack message
17    action_message_position = new
Phaser.Point(this.game_state.game.world.width / 2,
this.game_state.game.world.height * 0.1);
18    action_message_text = this.owner.name + " attacks " +
target.name + " with " + damage + " damage";
19    attack_message = new RPG.ActionMessage(this.game_state,
this.name + "_action_message", action_message_position, {group:
"hud", texture:     "rectangle_image", scale: {x: 0.85, y: 0.2},
duration: 1, message: action_message_text});
20 };
```

The code for both PhysicalAttack and MagicAttack is shown below. The first one is exactly the attack we already had. It calculates attack and defense multipliers based on the attack and defense stats, respectively, and apply the correct damage. On the other hand, the MagicAttack has three main differences: 1) it has a mana cost; 2) the attack multiplier is based in a magic attack stat; 3) The attack multiplier is higher, which results in increased damage.

```
1 var RPG = RPG || {};
2
3 RPG.PhysicalAttack = function (game_state, name, position,
properties) {
4    "use strict";
5    RPG.Attack.call(this, game_state, name, position,
properties);
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Page 281

```
6 };
7
8 RPG.PhysicalAttack.prototype =
Object.create(RPG.Attack.prototype);
9 RPG.PhysicalAttack.prototype.constructor = RPG.PhysicalAttack;
10
11 RPG.PhysicalAttack.prototype.hit = function (target) {
12     "use strict";
13     var damage, attack_multiplier, defense_multiplier,
action_message_position, action_message_text, attack_message;
14     // calculate random attack and defense multipliers
15     attack_multiplier =
this.game_state.game.rnd.realInRange(0.8, 1.2);
16     defense_multiplier =
this.game_state.game.rnd.realInRange(0.8, 1.2);
17     // calculate damage
18     damage = Math.max(0, Math.round((attack_multiplier *
this.owner.stats.attack) - (defense_multiplier *
target.stats.defense)));
19     // apply damage
20     target.receive_damage(damage);
21
22     this.show_message(target, damage);
23 };


1 var RPG = RPG || {};
2
3 RPG.MagicAttack = function (game_state, name, position,
properties) {
4     "use strict";
5     RPG.Attack.call(this, game_state, name, position,
properties);
6
7     this.mana_cost = properties.mana_cost;
8 };
9
10 RPG.MagicAttack.prototype =
Object.create(RPG.Attack.prototype);
11 RPG.MagicAttack.prototype.constructor = RPG.MagicAttack;
12
13 RPG.MagicAttack.prototype.hit = function (target) {
14     "use strict";
15     var damage, attack_multiplier, defense_multiplier,
action_message_position, action_message_text, attack_message;
16     // the attack multiplier for magic attacks is higher
```

```
17      attack_multiplier =
this.game_state.game.rnd.realInRange(0.9, 1.3);
18      defense_multiplier =
this.game_state.game.rnd.realInRange(0.8, 1.2);
19      // calculate damage using the magic attack stat
20      damage = Math.max(0, Math.round((attack_multiplier *
this.owner.stats.magic_attack) - (defense_multiplier *
target.stats.defense)));
21      // apply damage
22      target.receive_damage(damage);
23
24      // reduce the unit mana
25      this.game_state.current_unit.stats.mana -= this.mana_cost;
26
27      this.show_message(target, damage);
29 };
```

Now, we are going to change the Attack menu item to instantiate an Attack prefab and create the Magic menu item. The modification in AttackMenuItem is shown below. Instead of calling the "attack" method in the current unit it simply create a new Attack prefab. The MagicAttackMenuItem code is shown below. It can be selected only when the current unit has enough mana and create a new MagicAttack prefab. Notice that I'm using the same mana cost for all units, but you can change that if you want.

```
1 var RPG = RPG || {};
2
3 RPG.AttackMenuItem = function (game_state, name, position,
properties) {
4     "use strict";
5     RPG.MenuItem.call(this, game_state, name, position,
properties);
6 };
7
8 RPG.AttackMenuItem.prototype =
Object.create(RPG.MenuItem.prototype);
9 RPG.AttackMenuItem.prototype.constructor = RPG.AttackMenuItem;
10
11 RPG.AttackMenuItem.prototype.select = function () {
12     "use strict";
13     // disable actions menu
14     this.game_state.prefabs.actions_menu.disable();
15     // enable enemy units menu so the player can choose the
target
16     this.game_state.prefabs.enemy_units_menu.enable();
```

```
17    // save current attack
18    this.game_state.current_attack = new
RPG.PhysicalAttack(this.game_state,
this.game_state.current_unit.name + "_attack", {x: 0, y: 0},
{group: "attacks", owner_name:
this.game_state.current_unit.name});
19 };
```

```
1 var RPG = RPG || {};
2
3 RPG.MagicAttackMenuItem = function (game_state, name,
position, properties) {
4     "use strict";
5     RPG.MenuItem.call(this, game_state, name, position,
properties);
6
7     this.MANA_COST = 10;
8 };
9
10 RPG.MagicAttackMenuItem.prototype =
Object.create(RPG.MenuItem.prototype);
11 RPG.MagicAttackMenuItem.prototype.constructor =
RPG.MagicAttackMenuItem;
12
13 RPG.MagicAttackMenuItem.prototype.select = function () {
14     "use strict";
15     // use only if the current unit has enough mana
16     if (this.game_state.current_unit.stats.mana >=
this.MANA_COST) {
17         // disable actions menu
18         this.game_state.prefabs.actions_menu.disable();
19         // enable enemy units menu so the player can choose
the target
20         this.game_state.prefabs.enemy_units_menu.enable();
21         // save current attack
22         this.game_state.current_attack = new
RPG.MagicAttack(this.game_state,
this.game_state.current_unit.name + "_attack", {x: 0, y: 0},
{group: "attacks", mana_cost: this.MANA_COST, owner_name:
this.game_state.current_unit.name});
23     }
24 };
```

In the next step we must change the HUD to show those new features. First, we change the "show_player_action" method in BattleState to add the Magic menu item, as shown below. Then, we change the PlayerMenuItem prefab to show its remaining mana, and not only its health.

```
1 actions = [{text: "Attack", item_constructor:
RPG.AttackMenuItem.prototype.constructor},
2               {text: "Magic", item_constructor:
RPG.MagicAttackMenuItem.prototype.constructor},
3               {text: "Item", item_constructor:
RPG.InventoryMenuItem.prototype.constructor}];
```

Finally, now you can define the initial mana and magic attack stats for each unit and try playing the game with all its contents. Try using different attacks to see if everything is working properly.

And now, we finished this tutorial series!

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Phaser Tutorial – How to Create an Idle Clicker Game
By Ben Sparks

What is an idle click game?

Also known as clicker and incremental games, these type of games have you clicking something repeatedly or idling to gain some form of currency (e.g. cookies, money, energy) which you then use to buy upgrades. It's a very simple concept at the core, but quite addictive!

Clicker games were first made popular in 2013 with a game called Cookie Clicker. Since then many games have been created in this genre, the themes and flavor of these games vary widely. From cookies to fantasy rpg, sci fi, farming, sand castles, etc.

One such clicker game that I've played quite a bit is Clicker Heroes, it's a fantasy RPG style game where you click various monsters to progress. In this tutorial, we are going to build the foundations of a clicker game similar to Clicker Heroes that you can build upon

and add your own style and flair.

Tutorial source code

You can download the tutorial source code here.
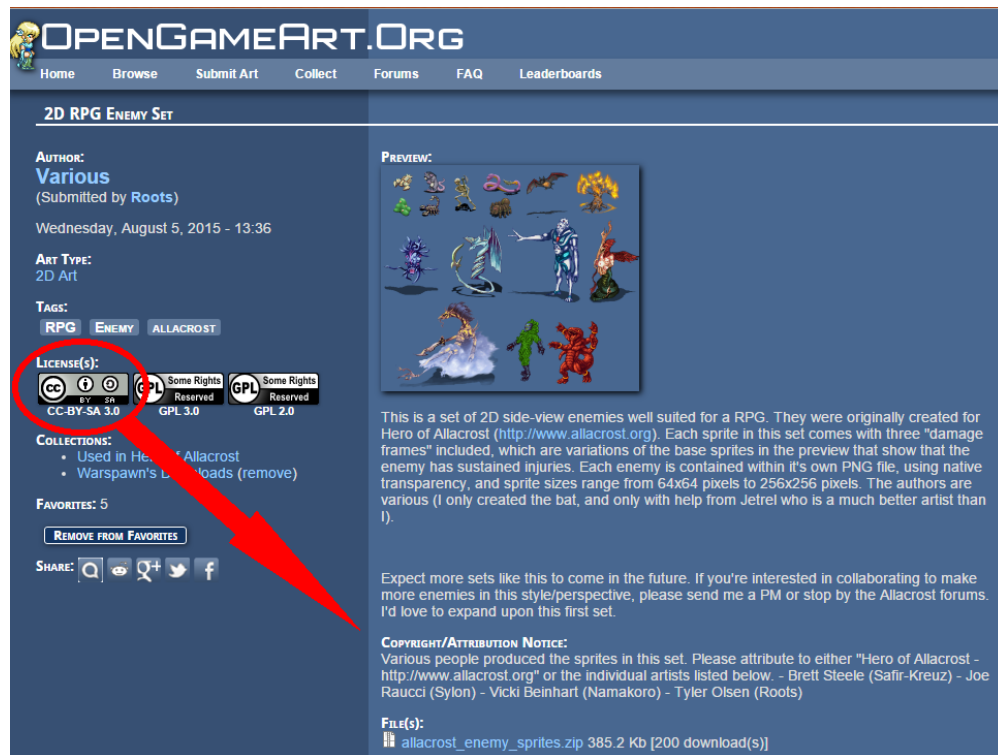
Source Control

The first thing that I do when starting any project is create a folder, and initialize it for git source control. If you don't already have git, you can download it for your OS from the official site. With any software development, it is critical to have backups. Using source control, specifically git, is a much more robust and safe solution than simply copying or zipping your files periodically. It's also important to do micro commits, save your progress in very small increments so that you can roll back to any point if you find that you've added some new code that is breaking everything. Also it prepares your codebase automatically for upload to a service like github or bitbucket. There are many things beyond simply committing your code that you can do with git, if you are new to git I encourage you to checkout the Git and Github Fundamentals course at Zenva Academy.

After installing git for your system, open the folder you created in a command window, and simply type `git init` and it will be ready to go.

Gathering Art

We need to get some sprites for the monsters. A great site to find free art that you can use for your projects is OpenGameArt.org. Much of the artwork is either public domain or licensed to the Creative Commons. A quick search and I've found a resource with a variety of monster sprites. This particular set is licensed under CC BY SA 3.0 which basically means that you must give credit to the original authors, and any modifications you make must be shared under the same license. There are other licensing options

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

available to artists, and some do not allow the use of the resources in commercial

applications, so be sure to check the licensing on the site.

The monster images that I've chosen come in various sizes, each with 4 frames representing various damage states. For this game, we don't really need the extra damage states, so I have cropped all of the images to just the first frame. For the purposes of this tutorial, I think that 1 image per monster is all that we'll need. It would be nice if these monsters were animated, but the single image works well for a placeholder right now.

In addition to the monsters, we'll need some rpg icons for the upgrades and loot. It's not likely that we'll use all 496 images in this set, but we can prune the collection later before we are ready to release of the ones that we don't use. We'll also need something to use as a background for our world, this forest scene will do nicely.

You'll find all of the sprites in the assets/images folder of the companion source code to this project. If you're downloading your own selection of images to use, place them in there as well.

Reminder: Once you've added your files, go ahead and commit them to source control.

Setup Phaser

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

You'll need to download Phaser, you can either clone the Github repo or download the release or use something like Bower to install it. Phaser games are web based games, so first we need a HTML page to host it. Create a file called index.html in the root of your project folder and put this for the contents.

```
1  <!doctype html>
2  <html>
3      <head>
4          <meta charset="utf-8">
5          <title>Click of the Titans</title>
6          <style>body {margin: 0;padding: 0;}</style>
7
8          <script src="lib/phaser.min.js"></script>
9
10         <script src="src/game.js"></script>
11     </head>
12     <body>
13
14     </body>
15 </html>
```

Reminder: micro commits, commit your index.html file right now!

You may have noticed that we added a reference to game.js located in the src folder. We don't have that file yet, let's create it now.

```
1  var game = new Phaser.Game(800, 600, Phaser.AUTO, '');
2
3  game.state.add('play', {
4      preload: function() {
5          game.load.image('skeleton',
   'assets/allacrost_enemy_sprites/skeleton.png');
6      },
7      create: function() {
8          var skeletonSprite = game.add.sprite(450, 290,
   'skeleton');
9          skeletonSprite.anchor.setTo(0.5, 0.5);
10     },
11     render: function() {
12         game.debug.text('Adventure Awaits!', 250, 290);
13     }
14 });
15
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Page 289

```
16 game.state.start('play');
```

In Phaser, everything starts with an instance of the <u>Game</u> class. We are passing it the width and height of the game screen that we want as well as the renderer. Phaser supports both Canvas and WebGL based rendering, Phaser.AUTO tells it to pick based on what's available in the browser (WebGL if available).

In the preload phase of our "play" game state we are going to load up one of our monster images so that we can have something to show other than a blank screen. Each state in Phaser has several <u>phases</u> in its lifecycle, and `preload` is one of the first.

Next up is `create` , after assets are loaded, it's safe to create sprites using them. The <u>Sprite</u> is one of the main game objects in Phaser, most everything that you create in the game world will generally have a sprite involved. Adding a sprite takes a few parameters: the x & y coordinates of where it should be located on the screen, and a key that can be used to reference it later. We're also going to change the anchor for the sprite to the center of the image. The anchor is the point in space where the sprite is positioned or rotated around. By default the anchor is set to the top left most point of the image `0`, `0` I usually like to work with the anchor in the center of the image, because if I want to spin the sprite, I want it to spin around the center instead of the top left (like a wheel).

Finally, `render` is one of the last phases in the lifecycle, it happens after update (we don't need update for this game). Most of the time Phaser handles rendering internally, but for now we'll just add some fun text to make our screen more exciting.

Reminder: now's a great time to commit your file!

Running your game

We're almost ready to fire up Phaser and see what we have, except that it's not just as simple as clicking on index.html and loading it in your browser. Due to security restrictions the browser will not allow you to load assets asynchronously. For that reason you need to run a server, or load the files in a different manner. PhotonStorm has posted an article listing <u>several options for servers,</u> however if loading the software to run a server isn't something that you are interested in, I have another option. If you are running Google Chrome, you can actually setup your game to run as a Chrome application quite easily. Doing this elevates your game out of the browser sandbox (and actually gives you a set of APIs to access native systems such as the file system and networking) and these restrictions no longer apply. There are however a different set of security restrictions involving resources located outside of the files packaged with the app. Depending on how

<u>Zenva Academy</u> – Online courses on Phaser and game programming
<u>Zenva for Schools</u> – Coding courses for high schools

you want to distribute your final product, creating a Chrome app may be a great choice. I'll include the basic files needed to launch this as a Chrome App, if you are interested in learning more I encourage you to read more at the <u>official documentation</u>.
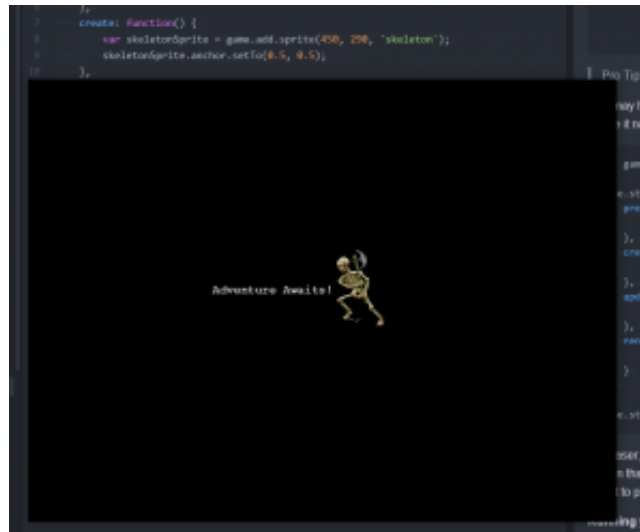
Once you launch your game, you should see this:
Now you know that Phaser is up and running. We've loaded a sprite and written a bit of text to the screen! We're almost done ☐ … well not quite, but we're ready to start actually making a game.

Setting the Stage

The first thing that we want to do is get rid of that boring black background and bring our world to life. In the `preload` method, add all of the images that make up the forest background. There happen to be four different images for this because it is designed to be used as a <u>parallax scrolling</u> background. Typically this type of background is used in side scrolling platformer games, but it gives us the option of doing an animation effect later.

```
1 this.game.load.image('forest-back',
'assets/parallax_forest_pack/layers/parallax-forest-back-
trees.png');
2 this.game.load.image('forest-lights',
'assets/parallax_forest_pack/layers/parallax-forest-
lights.png');
3 this.game.load.image('forest-middle',
'assets/parallax_forest_pack/layers/parallax-forest-middle-
trees.png');
```

<u>Zenva Academy</u> – Online courses on Phaser and game programming
<u>Zenva for Schools</u> – Coding courses for high schools

Page 291

```
4 this.game.load.image('forest-front',
'assets/parallax_forest_pack/layers/parallax-forest-front-
trees.png');
```

In the `create` method since we have 4 different images that make up the background, we'll create a <u>Group</u> of <u>TileSprite</u> objects. Using a javascript array of the image keys, we can use the built in forEach method to quickly and elegantly create all four in the correct order.

```
1 var state = this;
2
3 this.background = this.game.add.group();
4 // setup each of our background layers to take the full screen
5 ['forest-back', 'forest-lights', 'forest-middle', 'forest-
front']
6     .forEach(function(image) {
7         var bg = state.game.add.tileSprite(0, 0,
state.game.world.width,
8             state.game.world.height, image, '',
state.background);
9         bg.tileScale.setTo(4,4);
10     });
```

A TileSprite is a type of image that is meant to be repeated or tiled over a large area. To create one you need to pass it not only the x & y coordinates, but also a width and height for how big the area that it needs to cover should be. Just like the Sprite, you give it the image key to use. The final parameter that we pass it is our group, so that each one is automatically added to the background group. We want the background to cover the whole screen, so we'll pass in the game world's width and height. Also, in this case, our image isn't really meant to be tiled like a mosaic, so we'll scale the tiling so that it looks more like one image (play with the scale numbers to see what I mean about the tiling).

Let's take a moment to talk about the group that we created. Groups are an extremely powerful feature of Phaser that are capable of a whole lot more than just the name suggests. At the very base level, they act as a collection, sort of like an array, and they have several methods available for traversing and manipulating the items in the collection. Another important aspect of groups is that the children of the group are positioned, rotated, and scaled all relative to the group itself. If we set `group.x = 10;` then it will move *all* of the children over by 10 as well. The position of the children is relative meaning that if a child item also has it's x coord set to 10, then it will actually be placed at x 20 on the screen (assuming that the group itself is not a child of another parent that isn't at 0,0). We will use groups extensively in our game and I encourage you

<u>Zenva Academy</u> – Online courses on Phaser and game programming
<u>Zenva for Schools</u> – Coding courses for high schools

to read the documentation to find out more about all of the methods and properties that are available.

Reminder: time to commit your files!

The Army of Darkness

After setting the stage, the next thing we need is something to attack. Time to unleash the horde of monsters that we've collected. In order to be able to use all of our monsters, we need to load each image. Let's change the `preload` section of our `play` state to do just that. Each one needs a key to reference it by, and the path to the image file.

```
1     this.game.load.image('aerocephal',
'assets/allacrost_enemy_sprites/aerocephal.png');
2     this.game.load.image('arcana_drake',
'assets/allacrost_enemy_sprites/arcana_drake.png');
3     this.game.load.image('aurum-drakueli',
'assets/allacrost_enemy_sprites/aurum-drakueli.png');
4     this.game.load.image('bat',
'assets/allacrost_enemy_sprites/bat.png');
5     this.game.load.image('daemarbora',
'assets/allacrost_enemy_sprites/daemarbora.png');
6     this.game.load.image('deceleon',
'assets/allacrost_enemy_sprites/deceleon.png');
7     this.game.load.image('demonic_essence',
'assets/allacrost_enemy_sprites/demonic_essence.png');
8     this.game.load.image('dune_crawler',
'assets/allacrost_enemy_sprites/dune_crawler.png');
9     this.game.load.image('green_slime',
'assets/allacrost_enemy_sprites/green_slime.png');
10    this.game.load.image('nagaruda',
'assets/allacrost_enemy_sprites/nagaruda.png');
11    this.game.load.image('rat',
'assets/allacrost_enemy_sprites/rat.png');
12    this.game.load.image('scorpion',
'assets/allacrost_enemy_sprites/scorpion.png');
13    this.game.load.image('skeleton',
'assets/allacrost_enemy_sprites/skeleton.png');
14    this.game.load.image('snake',
'assets/allacrost_enemy_sprites/snake.png');
15    this.game.load.image('spider',
'assets/allacrost_enemy_sprites/spider.png');
```

```
16      this.game.load.image('stygian_lizard',
'assets/allacrost_enemy_sprites/stygian_lizard.png');
```

In the `create` method is where we'll want to create Sprite objects for these images so that they can exist in the game world. Much like the background images, it's faster and easier to read if we put all the information that we need to load into an array. This time however, we need more than just the image key. We're going to want to display the name of the monster on the screen below it, so that the player knows what they are facing. To do this we'll create an array of objects that has a name property suitable for display, as well as the image key.

```
1 var monsterData = [
2     {name: 'Aerocephal', image: 'aerocephal'},
3     {name: 'Arcana Drake', image: 'arcana_drake'},
4     {name: 'Aurum Drakueli', image: 'aurum-drakueli'},
5     {name: 'Bat', image: 'bat'},
6     {name: 'Daemarbora', image: 'daemarbora'},
7     {name: 'Deceleon', image: 'deceleon'},
8     {name: 'Demonic Essence', image: 'demonic_essence'},
9     {name: 'Dune Crawler', image: 'dune_crawler'},
10     {name: 'Green Slime', image: 'green_slime'},
11     {name: 'Nagaruda', image: 'nagaruda'},
12     {name: 'Rat', image: 'rat'},
13     {name: 'Scorpion', image: 'scorpion'},
14     {name: 'Skeleton', image: 'skeleton'},
15     {name: 'Snake', image: 'snake'},
16     {name: 'Spider', image: 'spider'},
17     {name: 'Stygian Lizard', image: 'stygian_lizard'}
18 ];
```

After that, we'll need to create the actual sprites for them in order to render them in the game world. Time for another group!

```
1 this.monsters = this.game.add.group();
2
3 var monster;
4 monsterData.forEach(function(data) {
5     // create a sprite for them off screen
6     monster = state.monsters.create(1000,
state.game.world.centerY, data.image);
7     // center anchor
8     monster.anchor.setTo(0.5);
9     // reference to the database
```

```
10     monster.details = data;
11
12     //enable input so we can click it!
13     monster.inputEnabled = true;
14     monster.events.onInputDown.add(state.onClickMonster,
state);
15 });
```

When it comes to sprites, groups have a special method create that we can use. For other game objects we need to create them using the normal game factories, and add them as children to the group, the create method only works for sprites. It takes the normal sprite parameters; position and image key, but also lets you pass in a flag as to whether or not the sprite "exists". This parameter is most often used in object pooling, which we'll cover a bit later. Each sprite we'll set the anchor to the center, and set a reference to our monsterData item so that we can reference it later (to display the name).

We're also going to want to be able to click the monsters (it's a *click*er game after all), we can tell Phaser to register click events by setting inputEnabled to true. Doing so will enable several input events, we're interested in what happens when we click the mouse button down (or tap it). For that we'll tell it to use the onClickMonster method of our state, and that it should be called in the context of our state (javascript "this"). We haven't created that method yet, we'll do that in a minute.

In this game, we only ever want to face one monster at a time, so we need a reference to the monster that we are currently facing. Another cool feature of a group is that we can let Phaser randomly select one using getRandom() . Then set the position of the monster to roughly the center of the screen, a little off so that we have room for some other things later.

```
1 this.currentMonster = this.monsters.getRandom();
2 this.currentMonster.position.set(this.game.world.centerX +
100, this.game.world.centerY);
```

Next we can modify our render method so that we can show the name of the monster that we are up against.

```
1 render: function() {
2     game.debug.text(this.currentMonster.details.name,
3         this.game.world.centerX - this.currentMonster.width /
2,
4         this.game.world.centerY + this.currentMonster.height /
2);
```

```
5 }
```

Finally, we need to add that `onClickMonster` handler so that something happens when we click them. For now, we'll just change out the monster on each click so that we can test running through our list (epic insta-kill!). Each click we'll set the currentMonster to another random one, and position it in the center of the screen. We'll move the other monster off screen so that we can't see it at the same time.

```
1 onClickMonster: function() {
2     // reset the currentMonster before we move him
3     this.currentMonster.position.set(1000,
this.game.world.centerY);
4     // now pick the next in the list, and bring him up
5     this.currentMonster = this.monsters.getRandom();
6     this.currentMonster.position.set(this.game.world.centerX +
100, this.game.world.centerY);
7 },
```
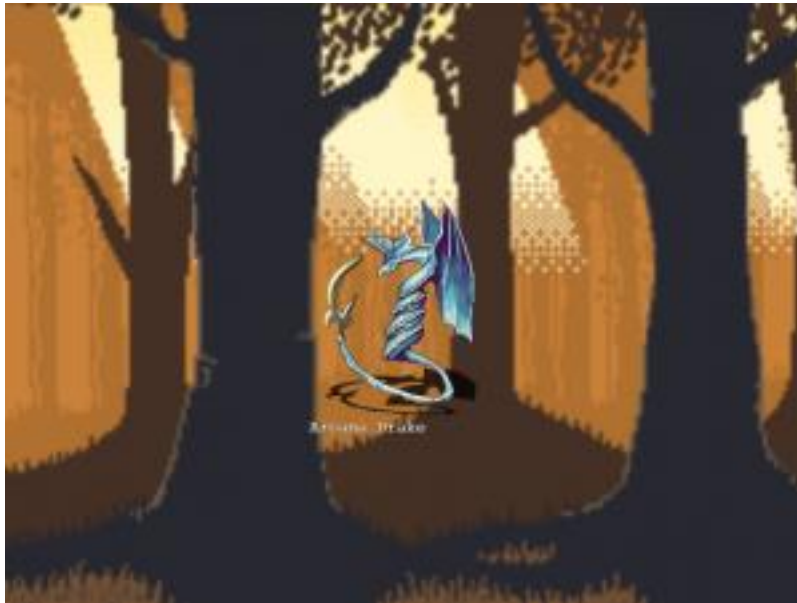

Reminder: time to commit your files again!

Clicking For Great Justice

Obviously just clicking to cycle through the monsters isn't much of a gameplay element. Instead we need to simulate RPG combat by clicking and dealing damage to the

monsters. In order to deal damage, we need to know something about our player. The main hero of this game isn't directly represented by a sprite on the screen, so we will just create a plain javascript object to hold useful information about them. Add this to the bottom of the `create` phase:

```
1  // the main player
2  this.player = {
3      clickDmg: 1,
4      gold: 0
5  };
```

Also in order to be able to kill the monsters, they need to have some health to damage. Let's go back to our monster database from before, and add a maxHealth attribute to each one. For now, for the values of maxHealth, I've just chosen somewhat arbitrary numbers, by looking at the sprite and just making a few tougher than the others. Later we can get deeper into gameplay and balance out the toughness of the monsters. I also adjusted the spacing so that it looks a little more like a table and is easier to edit.

```
1  var monsterData = [
2      {name: 'Aerocephal',     image: 'aerocephal',      maxHealth: 10},
3      {name: 'Arcana Drake',   image: 'arcana_drake',    maxHealth: 20},
4      {name: 'Aurum Drakueli', image: 'aurum-drakueli',  maxHealth: 30},
5      {name: 'Bat',            image: 'bat',             maxHealth: 5},
6      {name: 'Daemarbora',     image: 'daemarbora',      maxHealth: 10},
7      {name: 'Deceleon',       image: 'deceleon',        maxHealth: 10},
8      {name: 'Demonic Essence', image: 'demonic_essence', maxHealth: 15},
9      {name: 'Dune Crawler',   image: 'dune_crawler',    maxHealth: 8},
10     {name: 'Green Slime',    image: 'green_slime',     maxHealth: 3},
11     {name: 'Nagaruda',       image: 'nagaruda',        maxHealth: 13},
12     {name: 'Rat',            image: 'rat',             maxHealth: 2},
13     {name: 'Scorpion',       image: 'scorpion',        maxHealth: 2},
```

```
14     {name: 'Skeleton',              image:
'skeleton',         maxHealth: 6},
15     {name: 'Snake',                 image:
'snake',            maxHealth: 4},
16     {name: 'Spider',                image:
'spider',           maxHealth: 4},
17     {name: 'Stygian Lizard',    image:
'stygian_lizard',    maxHealth: 20}
18 ];
```

Phaser sprites come with both <u>Health</u> and <u>LifeSpan</u> components built in that provide all of the features that we need in order to get combat working. They provide us with a `damage()` and `heal()` method and take into account health and maxHealth in those methods. When you apply damage and the health drops to 0 (or below) it will automatically call `kill()` and fire the `onKilled` event. Similarly, there is a `revive()` method that allows you to bring a monster back from the dead fully healed. When a monster is revived, it fires the onRevived event. We will hook into those events to drop loot and change monsters. Inside the loop that creates the monster sprites, let's set the health component based on the value in our data array.

```
1 // use the built in health component
2 monster.health = monster.maxHealth = data.maxHealth;
3
4 // hook into health and lifecycle events
5 monster.events.onKilled.add(state.onKilledMonster, state);
6 monster.events.onRevived.add(state.onRevivedMonster, state);
```

Next update the `onClickMonster` method to deal damage to the monster based on the player's `clickDmg` attribute. When the monster's health reaches 0, it will be killed.

```
1 onClickMonster: function(monster, pointer) {
2     // apply click damage to monster
3     this.currentMonster.damage(this.player.clickDmg);
4 }
```

Clicking on a monster will do damage, but it's important to provide feedback to the player that this is happening. No one likes to click things and see nothing happen, it feels broken. Also, I think it's time to get rid of the debug text and use an actual text object to display the monster's name and health. Add these 2 <u>Text</u> objects to your create method. Again we'll create another group so that moving the monster information around is easy. Creating Text objects is relatively straight forward, like most display objects you provide it the position on the screen (or relative to its parent), but instead of an image key, you

pass it the string that you want to display. Optionally you can send in font styling data to change how it's displayed. This font style information is very similar to standard HTML Canvas fonts.

```
1 this.monsterInfoUI = this.game.add.group();
2 this.monsterInfoUI.position.setTo(this.currentMonster.x - 220,
this.currentMonster.y + 120);
3 this.monsterNameText =
this.monsterInfoUI.addChild(this.game.add.text(0, 0,
this.currentMonster.details.name, {
4    font: '48px Arial Black',
5    fill: '#fff',
6    strokeThickness: 4
7 }));
8 this.monsterHealthText =
this.monsterInfoUI.addChild(this.game.add.text(0, 80,
this.currentMonster.health + ' HP', {
9    font: '32px Arial Black',
10    fill: '#ff0000',
11    strokeThickness: 4
12 }));
```

Back in our `onClickMonster` event, we want to update the text now to reflect the monster's new health.

```
1 // update the health text
2 this.monsterHealthText.text = this.currentMonster.alive ?
this.currentMonster.health + ' HP' : 'DEAD';
```

When a monster is killed it fires the `onKilled` event, we want to push it back off screen and then select a new monster and revive them. The revive method optionally takes a health parameter that will set the health of the monster to the value provided. It then sets alive, visible, and exists to true, and then fires the onRevived event. Add the `onKilledMonster` method to our state now.

```
1 onKilledMonster: function(monster) {
2    // move the monster off screen again
3    monster.position.set(1000, this.game.world.centerY);
4
5    // pick a new monster
6    this.currentMonster = this.monsters.getRandom();
7    // make sure they are fully healed
8    this.currentMonster.revive(this.currentMonster.maxHealth);
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
9  },
```

When a monster is revived, we want to get them into position on the screen, and reset the monster display text to reflect the new monster. After the kill handler, let's add the `onRevivedMonster` event handler to our state as well. Here we move the monster to the center area of the screen and update our text objects to reflect the new information.

```
1  onRevivedMonster: function(monster) {
2      monster.position.set(this.game.world.centerX + 100,
this.game.world.centerY);
3      // update the text display
4      this.monsterNameText.text = monster.details.name;
5      this.monsterHealthText.text = monster.health + 'HP';
6  },
```

Ok, now we can see what is happening, and that is good. It still doesn't quite feel exciting enough though, I think we can add more. For every click, let's display the damage that we're doing. For that, we'll want to create a pool of text objects. Each click we need to display the damage number for a short period and then it should disappear. Instead of creating a new text object for each click (which we *could* do), it's better to create a bunch up front and just change the properties. The reason is that in almost all programming situations, especially in JavaScript, creating new objects is an expensive operation. Not something that you want to be doing a lot of every frame. Let's create a pool of about 50 text objects (I think I can do 30 clicks / second so that's a decent buffer). The following should go in your `create` method.

```
1  this.dmgTextPool = this.add.group();
2  var dmgText;
3  for (var d=0; d<50; d++) {
4     dmgText = this.add.text(0, 0, '1', {
5         font: '64px Arial Black',
6         fill: '#fff',
7         strokeThickness: 4
8     });
9     // start out not existing, so we don't draw it yet
10    dmgText.exists = false;
11    dmgText.tween = game.add.tween(dmgText)
12        .to({
13            alpha: 0,
14            y: 100,
15            x: this.game.rnd.integerInRange(100, 700)
16        }, 1000, Phaser.Easing.Cubic.Out);
17
```

```
18      dmgText.tween.onComplete.add(function(text, tween) {
19          text.kill();
20      });
21      this.dmgTextPool.add(dmgText);
22 }
```

As you can see in order to create a pool of damage text objects to use we didn't have to do anything special. We used the groups that we already know and love. A standard for loop will help us create 50 of them, and we'll set them all to `exists = false` so that they don't render on the screen until we tell them to. Next, we're going to add another super powerful tool in Phaser's arsenal, a <u>Tween</u>. A tween allows you to modify various properties of an object over time. It uses one of several mathematical equations to "ease" these values from start to finish and create animation effects. For the damage text, we want it to fly out from where it was clicked in a random direction and also fade out so that by the time that it reaches its destination, it can no longer be seen. In the `to` method of the tween, we set the final values that we want the alpha, y and x properties of the object to be, the starting values will be the values of the object when the tween begins. The second parameter is the time that it should take to complete the tween, we'll set it to 1000 (the value is in milliseconds, so 1 second). The final parameter is the <u>Easing</u> equation that we want to use. When a tween animation is completed, an event is fired, we can hook into there and `kill()` the text object (effectively setting it back to `exists = false`).

Now we'll turn these on so that clicking to do damage is really exciting. Every time we click, we'll grab the first available dmgText object (i.e. not killed) from the group using `getFirstExists(false)`, the false tells the group that we want one that *doesn't* exist. Then update the text to reflect the current click damage. We need to reset the alpha property from when the tween had adjusted it, and we want to start this one at the spot where the player clicked. Then we'll start the tween again, and get our animation.

```
1 // grab a damage text from the pool to display what happened
2 var dmgText = this.dmgTextPool.getFirstExists(false);
3 if (dmgText) {
4     dmgText.text = this.player.clickDmg;
5     dmgText.reset(pointer.positionDown.x,
pointer.positionDown.y);
6     dmgText.alpha = 1;
7     dmgText.tween.start();
8 }
```
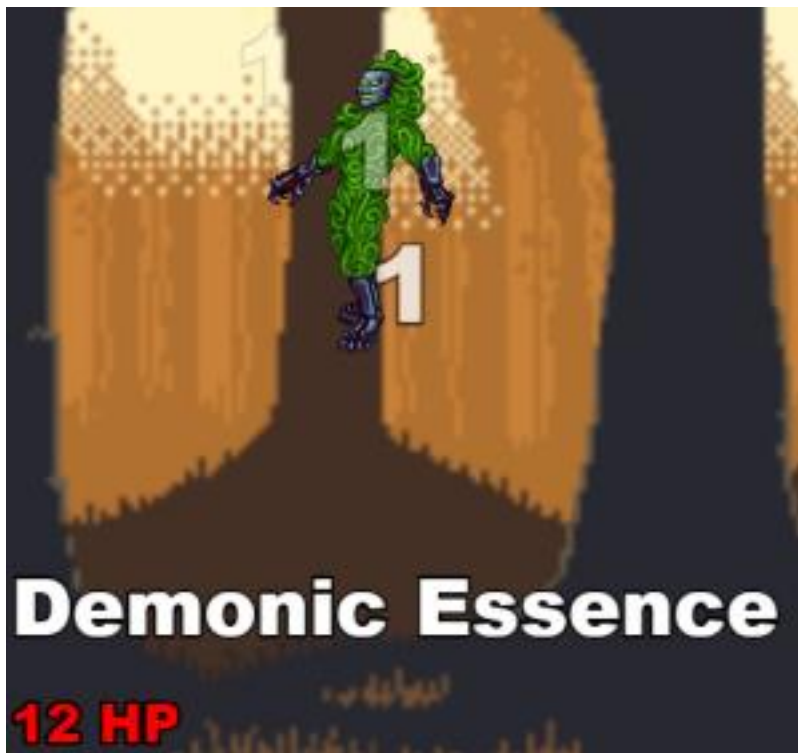
Reminder: commit that code!

Phat Lootz

Killing monsters isn't something that the player does just for sport, they're in it for the rewards of gold and treasure. We need to drop some loot when the monster dies, let's add a gold coin to our images loading in the `preload` method.

```
1 this.game.load.image('gold_coin',
'assets/496_RPG_icons/I_GoldCoin.png');
```

To start with, every monster will drop a single gold coin when it dies. The player will need to click on them to collect them, and so there could be quite a few laying around before they decide to do that. To create a pool of gold coins, again we'll utilize the group. This time, since we're creating a pool of sprites, we can use a special method called createMultiple. This method is very similar to the `create` method in that it only creates sprites, for this one, it takes a new parameter, the number that you want to create. Since we're not using a for loop this time to create them, we need a way to setup some defaults without having to loop through our newly created group, for that, we have another method called setAll. It takes the property and value and applies that to each object in the

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

group. We also want to register the onInputDown handler, since adding a handler to an event is a method, we use <u>callAll</u> to execute the `events.onInputDown.add` method on each child of the group.

```
1 // create a pool of gold coins
2 this.coins = this.add.group();
3 this.coins.createMultiple(50, 'gold_coin', '', false);
4 this.coins.setAll('inputEnabled', true);
5 this.coins.setAll('goldValue', 1);
6 this.coins.callAll('events.onInputDown.add',
'events.onInputDown', this.onClickCoin, this);
```

Again we need feedback in the UI so that we know that we are collecting gold. Add this text object to the `create` method.

```
1 this.playerGoldText = this.add.text(30, 30, 'Gold: ' +
this.player.gold, {
2     font: '24px Arial Black',
3     fill: '#fff',
4     strokeThickness: 4
5 });
```

Now we can add our click handler for the coins. When a coin is clicked, the `goldValue` of the coin is added to the player's gold. We also have to update the UI so that we can see the change. Finally, we kill the coin so that it disappears back into our pool for reuse.

```
1 onClickCoin: function(coin) {
2     // give the player gold
3     this.player.gold += coin.goldValue;
4     // update UI
5     this.playerGoldText.text = 'Gold: ' + this.player.gold;
6     // remove the coin
7     coin.kill();
8 }
```

Next we need to add to the `onKilledMonster` event so that we can actually drop these coins. Much like the dmgText objects, we'll grab the first available coin and bring it to life positioning it somewhere randomly in the center-ish of the screen. We update the goldValue (even though we're just setting it to 1 now) so that in the future we can drop different amounts of gold as we progress.
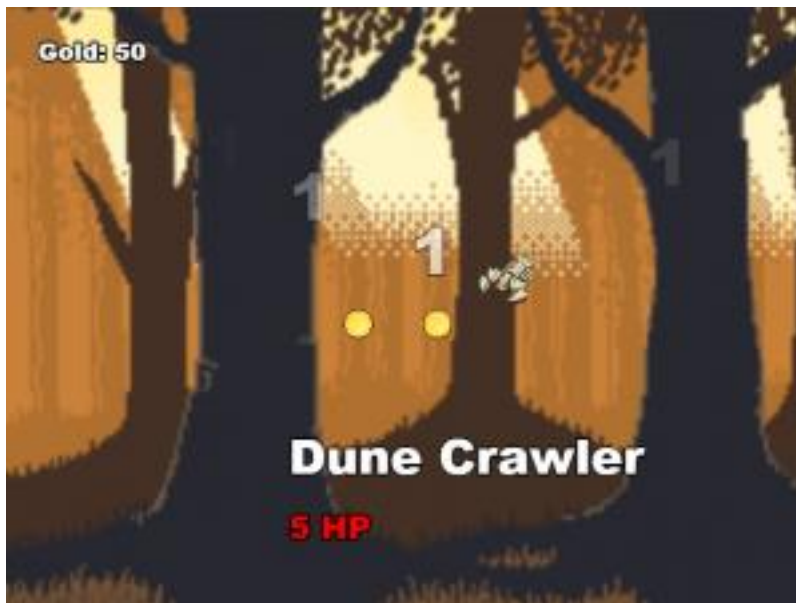
```
1 var coin;
```

<u>Zenva Academy</u> – Online courses on Phaser and game programming
<u>Zenva for Schools</u> – Coding courses for high schools

```
2 // spawn a coin on the ground
3 coin = this.coins.getFirstExists(false);
4 coin.reset(this.game.world.centerX +
this.game.rnd.integerInRange(-100, 100),
this.game.world.centerY);
```



```
5 coin.goldValue = 1;
```

As I mentioned earlier, the player has to click on the coins in order to collect them. After a few kills, if they don't do that, the floor will become quite littered with gold. While piles of cash is always fun, eventually we might run out of new coins to spawn from our pool. What we need to do is automatically collect gold for the player after a short time so that the world doesn't become cluttered.

To accomplish this, Phaser has a Timer object that we can use to create our own custom timed events. If the coin sits there not being clicked for 3 seconds, then we'll "click" it for them, by firing the same `onClickCoin` method that we use when they *do* click it. The timer event takes the duration, the handler, the context that the handler should fire in, and any other additional parameters that should be passed to the handler.

```
1 this.game.time.events.add(Phaser.Timer.SECOND * 3,
this.onClickCoin, this, coin);
```

Since we are going to be calling the `onClickCoin` method when the timer completes in addition to when the coin is clicked, we need to test and make sure that we don't call it twice. If the player clicks the coin before the timeout, the timeout will still fire, but the gold will already be collected, and the coin already killed. We can add a simple test at the top of the function to make sure that when we collect this coin it is still alive.

```
1 if (!coin.alive) {
2     return;
3 }
```

Reminder: commit the source, save your work!

Upgrades

I'm sure by now you're tired of doing just 1 point of damage every click, and we've got nothing to spend our precious gold on! Time to create some upgrades. The upgrades menu is going to be a box, with some buttons inside that you can click on to buy things. For the background of the upgrades menu, we could use more art from the web, or create an image ourselves in a paint program, but instead we can utilize the HTML5 Canvas to build it for us. Instead of loading an image, Phaser has an object called <u>BitmapData</u> that creates a canvas for us. The result of the canvas can then be used in place of an image on a sprite. Instead of setting the key when it is loaded like with an image, instead we need to manually add it to Phaser's asset cache. In the `preload` method we will generate images (colored rectangles) for the background of the upgrades menu and also the buttons.

```
1  // build panel for upgrades
2  var bmd = this.game.add.bitmapData(250, 500);
3  bmd.ctx.fillStyle = '#9a783d';
4  bmd.ctx.strokeStyle = '#35371c';
5  bmd.ctx.lineWidth = 12;
6  bmd.ctx.fillRect(0, 0, 250, 500);
7  bmd.ctx.strokeRect(0, 0, 250, 500);
8  this.game.cache.addBitmapData('upgradePanel', bmd);
9
10 var buttonImage = this.game.add.bitmapData(476, 48);
11 buttonImage.ctx.fillStyle = '#e6dec7';
12 buttonImage.ctx.strokeStyle = '#35371c';
13 buttonImage.ctx.lineWidth = 4;
14 buttonImage.ctx.fillRect(0, 0, 225, 48);
15 buttonImage.ctx.strokeRect(0, 0, 225, 48);
16 this.game.cache.addBitmapData('button', buttonImage);
```

<u>Zenva Academy</u> – Online courses on Phaser and game programming
<u>Zenva for Schools</u> – Coding courses for high schools

For the upgrades menu, we don't need something as heavy as a sprite (with all the health and other stuff). Instead for things that are going to be purely user interface objects, it is better to use a Image. An Image object has the same transform information (position, scale, rotation) just not all the extra things that a sprite does. Instead of passing in an image key, we'll pull the image out of the cache that we put it into earlier. The menu panel will also contain a group for the buttons, that way all of the buttons can follow the menu and be relative to its transforms. Because of the border on the background image, we'll move the button group in a little bit.

```
1 this.upgradePanel = this.game.add.image(10, 70,
this.game.cache.getBitmapData('upgradePanel'));
2 var upgradeButtons =
this.upgradePanel.addChild(this.game.add.group());
```



```
3 upgradeButtons.position.setTo(8, 8);
```

Our first upgrade will be to the click damage so we can start killing faster. Let's pick an icon from our library and add that to our image load.

```
1 this.game.load.image('dagger',
'assets/496_RPG_icons/W_Dagger002.png');
```

Phaser has another object in its bag of tricks, the Button object. A button is a subclass of an Image object that is designed to handle being clicked out of the box. It supports 4

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

different button states: Out, Over, Down, and Up. You can provide a spritesheet image with different looks for each of these states, it can make the button appear to actually be "pressed" for example, or change color when you hover over top of it. I've only generated a single background frame for our buttons for now, it still works fine without the other images. For the button we are going to add the icon, a label for what the upgrade is, and another text to display the gold cost for upgrading.

```
1 var button;
2 button = this.game.add.button(0, 0,
this.game.cache.getBitmapData('button'));
3 button.icon = button.addChild(this.game.add.image(6, 6,
'dagger'));
4 button.text = button.addChild(this.game.add.text(42, 6,
'Attack: ' + this.player.clickDmg, {font: '16px Arial Black'}));
5 button.details = {cost: 5};
6 button.costText = button.addChild(this.game.add.text(42, 24,
'Cost: ' + button.details.cost, {font: '16px Arial Black'}));
7 button.events.onInputDown.add(this.onUpgradeButtonClick,
this);
```

After creating the button, then add it to the buttons group.

```
1 upgradeButtons.addChild(button);
```

For now our button handler simply needs to test the one button that we have. First we'll check that we can afford to purchase the upgrade, if we can then we'll go through with it. We need to adjust the player's gold from the cost, and update the UI to reflect that change. We also need to improve the player's clickDmg attribute, and update the text on the button. Now we can spend our hard earned loot to get stronger!

```
1 onUpgradeButtonClick: function(button, pointer) {
2     if (this.player.gold - button.details.cost >= 0) {
3         this.player.gold -= button.details.cost;
4         this.playerGoldText.text = 'Gold: ' + this.player.gold;
5         this.player.clickDmg++;
6         button.text.text = 'Attack: ' + this.player.clickDmg;
7     }
```

```
8 }
```

Now it's time to add the second type of upgrade, the DPS upgrade. Until now the only way to deal damage to monsters is by directly clicking on them. While this is an important mechanic in our game, it can get tedious after a while. This is the part that makes this type of game fall into the "idle" genre. Players will continue to progress and deal damage to monsters even when they are not actively clicking them.

In order to add a new button, we need to repeat what we have, and change the effect of purchasing the upgrade. Doing this one button at a time will get unruly after a bit, so we want to create a database array of upgrades, just like we have for monsters. We'll need the icon to display, the name of the upgrade, the level for the number of times that it has been upgraded, the cost in gold to upgrade, and a callback function that we can fire to provide a different effect for each upgrade.

```
1 var upgradeButtonsData = [
2    {icon: 'dagger', name: 'Attack', level: 1, cost: 5,
purchaseHandler: function(button, player) {
3        player.clickDmg += 1;
4    }},
5    {icon: 'swordIcon1', name: 'Auto-Attack', level: 0, cost:
25, purchaseHandler: function(button, player) {
6        player.dps += 5;
7    }}
8 ];
```

We need to add the dps property to our player, so that our purchase handler can take effect.

```
1 this.player = {
2     clickDmg: 1,
3     gold: 0,
4     dps: 0
5 };
```

Now we can loop through our data and create all of our upgrade buttons in bulk. Because the buttons are about 48 pixels high, we'll separate each one by 50. Instead of hard coding the text strings and icon, they will be populated from the data.

```
1 var button;
2 upgradeButtonsData.forEach(function(buttonData, index) {
3     button = state.game.add.button(0, (50 * index),
state.game.cache.getBitmapData('button'));
4     button.icon = button.addChild(state.game.add.image(6, 6,
buttonData.icon));
5     button.text = button.addChild(state.game.add.text(42, 6,
buttonData.name + ': ' + buttonData.level, {font: '16px Arial
Black'}));
6     button.details = buttonData;
7     button.costText = button.addChild(state.game.add.text(42,
24, 'Cost: ' + buttonData.cost, {font: '16px Arial Black'}));
8     button.events.onInputDown.add(state.onUpgradeButtonClick,
state);
9
10    upgradeButtons.addChild(button);
11 });
```

When we click an upgrade button now we need to increase its level and update the text to reflect the change. Then we'll execute the purchaseHandler callback in the context of the state, passing the button and player along to the handler.

```
1 onUpgradeButtonClick: function(button, pointer) {
2     if (this.player.gold - button.details.cost >= 0) {
3         this.player.gold -= button.details.cost;
4         this.playerGoldText.text = 'Gold: ' + this.player.gold;
5         button.details.level++;
6         button.text.text = button.details.name + ': ' +
button.details.level;
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
7          button.details.purchaseHandler.call(this, button,
this.player);
8      }
9  }
```

Ok, so we're increasing both attack and dps (damage per second), but we haven't setup dps to do anything yet. For dps to make sense, if we have 1 dps then after 1 second has passed we do 1 damage. Waiting the full second to apply any damage at all though is too choppy and slow. Instead we'll update at 100ms (10 times a second). So, in order to apply 1 damage after 1 second, then we need to apply 0.10 damage every 100ms. Similar to the timer event that we used for the gold coins, there are also <u>loop</u> events that will repeat and call the handler at the duration over and over.

```
1  // 100ms 10x a second
2  this.dpsTimer = this.game.time.events.loop(100, this.onDPS,
this);
```

Our dps handler is going to get called every 100ms whether we have upgraded or not. So we check if we have something to do first, if so, then make sure the monster is still alive to apply the damage. Then we'll damage the monster and update the text to reflect the monster's health. I don't really want to display the decimals that might occur on the monster health from doing 10% damage, so we'll round it for the display (not the actual health).

```
1  onDPS: function() {
2      if (this.player.dps > 0) {
3          if (this.currentMonster && this.currentMonster.alive) {
4              var dmg = this.player.dps / 10;
5              this.currentMonster.damage(dmg);
6              // update the health text
7              this.monsterHealthText.text =
this.currentMonster.alive ?
Math.round(this.currentMonster.health) + ' HP' : 'DEAD';
8          }
9      }
10 }
```

Reminder: don't forget to commit!

Progression

Killing the same monsters over and over is boring. We need to keep increasing the difficulty so that the players can face a challenge and progress. Let's add some world level stats to the `preload` section.

```
1 // world progression
2 this.level = 1;
3 // how many monsters have we killed during this level
4 this.levelKills = 0;
5 // how many monsters are required to advance a level
6 this.levelKillsRequired = 10;
```

Each time we kill a monster now, we need to increase the kills stat. We also need to upgrade the monster's health based on the level. With increased risk, comes increased reward, and we'll also modify the coin value based on the world level as well. After 10 monsters have been killed, then we can progress to the next level.

```
1 onKilledMonster: function(monster) {
2     // move the monster off screen again
3     monster.position.set(1000, this.game.world.centerY);
4
5     var coin;
6     // spawn a coin on the ground
7     coin = this.coins.getFirstExists(false);
8     coin.reset(this.game.world.centerX +
this.game.rnd.integerInRange(-100, 100),
this.game.world.centerY);
9     coin.goldValue = Math.round(this.level * 1.33);
10    this.game.time.events.add(Phaser.Timer.SECOND * 3,
this.onClickCoin, this, coin);
11
12    this.levelKills++;
13
14    if (this.levelKills >= this.levelKillsRequired) {
15        this.level++;
16        this.levelKills = 0;
17    }
18
19    // pick a new monster
20    this.currentMonster = this.monsters.getRandom();
21    // upgrade the monster based on level
22    this.currentMonster.maxHealth =
Math.ceil(this.currentMonster.details.maxHealth + ((this.level -
1) * 10.6));
23    // make sure they are fully healed
```

```
24     this.currentMonster.revive(this.currentMonster.maxHealth);
25 }
```

Since we'll be getting a lot more money now, we also need to adjust the upgrade prices.

```
1 onUpgradeButtonClick: function(button, pointer) {
2     // make this a function so that it updates after we buy
3     function getAdjustedCost() {
4         return Math.ceil(button.details.cost +
(button.details.level * 1.46));
5     }
6
7     if (this.player.gold - getAdjustedCost() >= 0) {
8         this.player.gold -= getAdjustedCost();
9         this.playerGoldText.text = 'Gold: ' + this.player.gold;
10         button.details.level++;
11         button.text.text = button.details.name + ': ' +
button.details.level;
12         button.costText.text = 'Cost: ' + getAdjustedCost();
13         button.details.purchaseHandler.call(this, button,
this.player);
14     }
15 }
```

Finally, we'll add the level information to the UI so that the player knows how they are doing. By now adding text objects to reflect this information should be old hat to you!

```
1 // setup the world progression display
2 this.levelUI = this.game.add.group();
3 this.levelUI.position.setTo(this.game.world.centerX, 30);
4 this.levelText = this.levelUI.addChild(this.game.add.text(0,
0, 'Level: ' + this.level, {
5     font: '24px Arial Black',
6     fill: '#fff',
7     strokeThickness: 4
8 }));
9 this.levelKillsText =
this.levelUI.addChild(this.game.add.text(0, 30, 'Kills: ' +
this.levelKills + '/' + this.levelKillsRequired, {
10     font: '24px Arial Black',
11     fill: '#fff',
12     strokeThickness: 4
13 }));
```

We also need to update the level text when it changes inside the `onKilledMonster` method.

```
1 this.levelText.text = 'Level: ' + this.level;
2 this.levelKillsText.text = 'Kills: ' + this.levelKills + '/' +
this.levelKillsRequired;
```


Reminder: commit commit commit
Finished the basics, what's next?

What we've created here is a complete idle clicker RPG. Well, complete as in it's playable. There's still tons left to do in order to get this game turned into something that is really fun. In game development, they say that the last 10% of the work is the last 90% of your time. We're pretty much 90% done I think ☐

Here's a list of things that would be great to add: (possibly in future tutorials)

- monsters with animation
- sounds & music!
- more upgrades
- critical hits
- boss monsters
- timed boost powers
- different world backgrounds
- achievements
- leader boards

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

I hope that you have enjoyed this tutorial and making this game!

The Complete Guide to Debugging Phaser Games
By Ben Sparks

Game development is hard work. Just like any form of software development there are going to be bugs in the code. Debugging games can be especially challenging due to their fast paced real time nature. Fortunately, Phaser comes with quite a few tools built in to help you debug your games. Other than reading the API documentation or looking at the source code of examples, there isn't a whole lot of extra detail on how to debug them out there. In this guide, I'll explain what's available, and some real world ways to use them to combat the onslaught of code bugs.

About the Source Code Examples

All of the source code examples used in this tutorial are available in this zip file. Throughout the tutorial I'll be using ES2015 (ES6) JavaScript features. As of this writing, not all of these features are readily available in all of the browsers yet. The project that I have created uses jspm and Babel to enable the usage of these features now. If you're not familiar with ES2015, I recommend Exploring ES6 , an eBook that is a fantastic reference on the subject.

The Debug Object

Phaser comes built in with an instance of a Debug class attached to each game. It is always there though if you don't use it (making it become flagged as "dirty") it won't affect your performance. The debug object works by drawing various information, shapes, etc to a canvas. In the case of a canvas based renderer, it uses the same context as the game's canvas. In the case of WebGl, it needs to create a new canvas and render everything from that canvas to a sprite object to overlay over the game screen. In both cases (though especially true for WebGl) the debugging features should be used ONLY for debugging, and disabled when you ship the real version of the game.

When developing javascript applications in the browser, you might be used to using the developer console output for debug information. The problem with using that for games is that the information that we most often care about gets updated every frame instead of based on events. If we were to output the position of sprite for example, it would continuously scroll the console and we wouldn't be able to keep track of things very easily. This is why the ability to render debug information to our game screen is so important.

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Page 315

Text

Probably the simplest tool for debugging is drawing arbitrary text to the screen. The first parameter is the text that you want to display. Next you need to pass in the x & y coordinates that you want the text to start from. You can also pass a color and font information in if you wish.

```
1 render() {
2    this.game.debug.text(`Debugging Phaser ${Phaser.VERSION}`,
20, 20, 'yellow', 'Segoe UI');
3 }
```

Another way to draw text to the screen is using the line method. This one is actually used "internally" by many of the other Phaser debug methods. The debug object keeps track of the current position of a text cursor. Each time that line is called it uses the position of that cursor (x & y coords on the screen) and modifies the y for the next "line" of text. There are two more companion methods for this, start and stop. Calling start with x & y parameters passed in will set the cursor for the subsequent debug lines. It will also save the canvas state so that any other operations are not affected, so it's important to call stop again when you are done. You can call the line method without having called the start method first, however it's not recommended as the cursor's position will be unknown.

```
1 this.game.debug.start(20, 20, 'blue');
2 this.game.debug.line();
3 this.game.debug.line('First line.');
4 this.game.debug.line('Second line.');
5 this.game.debug.stop();
```

Camera

You could use the text and line methods to display information about the properties of a camera onto the screen, but instead another built in method is cameraInfo. This method automatically displays information about the camera's position, bounds, and number of visible sprites (based on the autoCull property). You need to pass in a reference to the camera you want info on, and the starting coordinates for the information. It will render 4 lines from that point to display all of the stats.

```
1  this.game.debug.cameraInfo(this.game.camera, 32, 32);
```

Input

Of course games are interactive, so how the controls behave is very important. Phaser has several ways to display information about various inputs for debugging purposes.

You can display information about a particular key being pressed on the keyboard using the `key` method. It takes a reference to a Phaser.Key object as well as the coordinates to display the information.
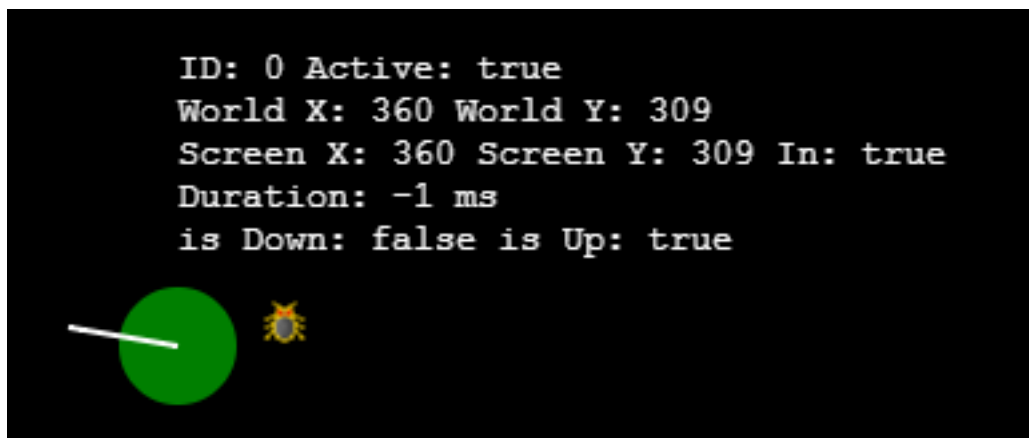
```
1  // inside create method
2  this.testKey =
game.input.keyboard.addKey(Phaser.KeyCode.SPACEBAR);
3
4  // inside render method
5  this.game.debug.key(this.testKey, 32, 128);
```

This will output a table of information about the key including the key code, whether or not it isDown, if it was justDown or justUp as well as a duration that it has been held

```
Key:            32              isDown:     false
justDown:       false           justUp:     false
Time Down:      1447256478203   duration:   0
```

down (if held down).

Generally we're also going to take input from a mouse or some other pointer like touch. For that, Phaser also has another method aptly named `pointer` . This one takes a reference to the pointer that you want to inspect and that's it. The information actually follows the pointer around in this case.

```
ID: 0 Active: true
World X: 360 World Y: 309
Screen X: 360 Screen Y: 309 In: true
Duration: -1 ms
is Down: false is Up: true
```

```
1 this.game.debug.pointer(this.game.input.activePointer);
```

In this case the white line starts from the last place the pointer was "clicked" and extends to the current position (useful for dragging or swiping). The green circle represents the pointer itself and follows it around. When you hold the button down, the duration counter ticks up. (The little bug image there is just a sprite, not part of the debug info.)

Perhaps you don't need all of that info following around the mouse, but you just need to know where it is, well then you're in luck, another simpler method is the inputInfo method which does just that. All that you need for this one is to tell it where to render.

```
1 this.game.debug.inputInfo(32, 32);
```

```
Input
X: 221 Y: 266
World X: 221 World Y: 266
Scale X: 1.0 Scale Y: 1.0
Screen X: 721 Screen Y: 381
```

Sprites

Sprites are one of the most important objects to any Phaser game, so not surprisingly, there are 4 debug methods specifically designed for them.
The first is spriteInfo which displays info about a sprite and its relation to the world. Important stuff like it's position, angle, anchor, bounds, and whether or not it's in view of

```
Sprite:   (15 x 15) anchor: 0.5 x 0.5
x: 266.0 y: 150.0
angle: 0.0 rotation: 0.0
visible: true in camera: true
bounds x: 258.5 y: 142.5 w: 15.0 h: 15.0
```

the camera.

```
1 this.game.debug.spriteInfo(sprite, 32, 32);
```

Next spriteCoords is all about position, where the sprite is on the screen, and where the sprite is in the world. Quite a bit less detailed than spriteInfo, but sometimes all that you really need to know is where your sprite is.

```
x:           266.00      y:           150.00
pos x:       266.00      pos y:       150.00
world x:     266.00      world y:     150.00
```
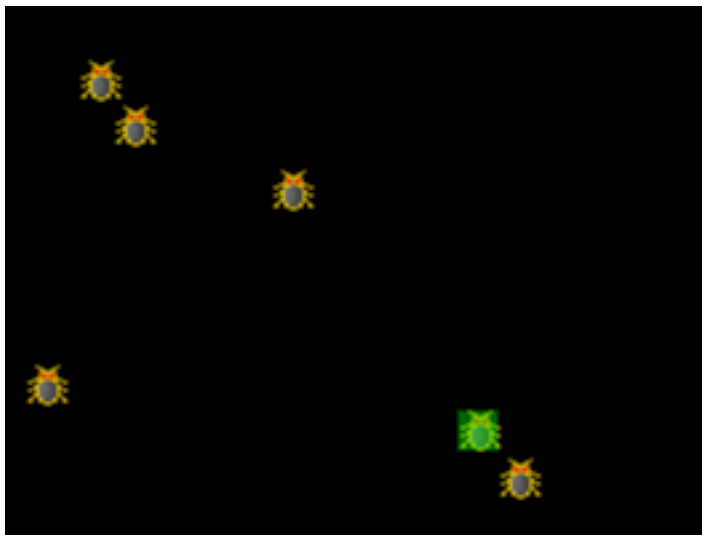
```
1 this.game.debug.spriteCoords(sprite, 32, 128);
```

You can get information about how a sprite is being interacted with by various input devices using spriteInputInfo. It displays a block of information at some coordinates just like spriteInfo, however this is all about input. In order for this one to work, the sprite actually needs to have input functions enabled, if they aren't by the time that you call this method, you'll get an error. Remember, for performance reasons, sprites do not have input enabled by default, you have to explicitly turn them on first.

```
1 this.game.debug.spriteInputInfo(sprite, 32, 192);
```

The last one is spriteBounds which will render a rectangle around the bounding box of the sprite, which might be the image, or the size of the sprite within a spritesheet if it is a part of one.

```
1 this.game.debug.spriteBounds(sprite);
```


Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
Sound: splat Locked: false
Is Ready?: true Pending Playback: false
Decoded: true Decoding: false
Total Duration: 2 Playing: true
Time: 813
Volume: 1 Muted: false
WebAudio: true Audio: false
```

Sounds

A small but handy block for displaying sound information is available using soundInfo. This one will show you whether the sound has been loaded and ready to use by the browser, whether or not it's using WebAudio as well as how long the sound has been playing and at what volume.

```
1  this.game.debug.soundInfo(this.splat, 32, 32, 'yellow');
```

Geometry

Geometry in Phaser is generally only used for logic purposes. It can be difficult however to wrap your brain around what's happening without a visual component. This is where the debug geom method comes into play. Geom can render a Circle, Rectangle, Point, or Line. Additionally, there is another method: rectangle, which as it suggests just renders a rectangle. In either case you can pass in an instance of one of Phaser's proper geometry classes, or an object that has the necessary properties to render the shape. For geom, if it's just an object, you need to pass in a fourth parameter to let it know what kind of shape it really is.

```
1  class GameState extends Phaser.State {
2      preload() {
3          this.game.load.image('bug', 'assets/bug.png');
4      }
5
6      create() {
7          let bugs = this.bugs = this.game.add.group();
8          bugs.createMultiple(30, 'bug', '', true);
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Page 321

```
9          bugs.setAll('anchor.x', 0.5);
10          bugs.setAll('anchor.y', 0.5);
11          bugs.setAll('inputEnabled', true);
12          bugs.forEach(bug => {
13              bug.position.set(this.game.rnd.integerInRange(0,
this.game.width), this.game.rnd.integerInRange(0,
this.game.height));
14              bug.detectRadius = 24;
15          });
16          bugs.callAll('events.onInputDown.add',
'events.onInputDown', this.onBugClick, this);
17
18          this.currentBug = null;
19      }
20
21      onBugClick(bug) {
22          this.currentBug = bug;
23      }
24
25      render() {
26          if (this.currentBug) {
27              this.game.debug.geom(new
Phaser.Circle(this.currentBug.x, this.currentBug.y,
this.currentBug.detectRadius * 2));
28          }
29      }
30 }
```

In addition to the geom method there is also a pixel method that will allow you do render as the name would suggest, a single pixel to the screen. Unlike rendering a point using geom, it does not have to be a Phaser Point object, instead just any arbitrary coordinates on the screen. You pass in x & y, the color to render your pixel, as well as the size. The default size is 2, and you might be saying well that's not a single pixel, true, but those can be hard to see (so tiny) so it makes sense to use 2 or perhaps even 4.

```
1 this.game.debug.pixel(100, 100, 'red', 4);
```

Physics

Phaser ships with three different physics options: Arcade, Ninja, and P2. Because they have a unified API you can actually use them all at the same time if you desire. Each physics enabled object will have a body attribute that will contain a reference to the

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

physics body that forces are applied to in each system. Because all of them will have a body property with similar attributes on it we can pass that body reference into `debug.bodyInfo`. That method will render information about position, velocity, acceleration, gravity, etc to the screen for the body passed in.

We can also highlight the physics body on the screen using the debug.body method. Since the physics body is a logical shape used for collision, this can be important to visualize as it might not always match up with the sprite that it is attached to.

```
1  class GameState extends Phaser.State {
2      init() {
3          this.game.physics.startSystem(Phaser.Physics.ARCADE);
4          this.game.physics.arcade.gravity.y = 1000;
5          this.game.physics.arcade.skipQuadTree = false;
6      }
7
8      preload() {
9          this.game.load.image('bug', 'assets/bug.png');
10     }
11
12     create() {
13         let bugs = this.bugs = this.game.add.group();
14         bugs.enableBody = true;
15         bugs.createMultiple(30, 'bug', '', true);
16         bugs.setAll('anchor.x', 0.5);
17         bugs.setAll('anchor.y', 0.5);
18         bugs.setAll('inputEnabled', true);
19         bugs.forEach(bug => {
20             bug.position.set(this.game.rnd.integerInRange(0,
this.game.width), this.game.rnd.integerInRange(0,
this.game.height));
21             bug.detectRadius = 24;
22         });
23         bugs.callAll('events.onInputDown.add',
'events.onInputDown', this.onBugClick, this);
24
25         bugs.forEach(bug => {
26             //this.game.physics.arcade.enable(bug);
27             bug.body.collideWorldBounds = true;
28             bug.body.bounce.set(0.8);
29             bug.body.velocity.x = Math.random() * 500 *
(Math.random() > 0.5 ? -1 : 1);
30         });
31         bugs.setAll('body.drag.x', 10);
```

```
32
33         this.currentBug = bugs.getRandom();
34     }
35
36    update() {
37         this.game.physics.arcade.collide(this.bugs,
this.bugs);
38    }
39
40    onBugClick(bug) {
41         this.currentBug = bug;
42         bug.body.velocity.y += -1000;
43         bug.body.velocity.x += Math.random() * 1000 *
(Math.random() > 0.5 ? -1 : 1);
44    }
45
46    render() {
47         this.game.debug.quadTree(this.game.physics.arcade.quad
Tree);
48         if (this.currentBug) {
49             this.game.debug.body(this.currentBug);
59             this.game.debug.bodyInfo(this.currentBug, 32, 32);
51         }
52    }
53 }
```

Another thing that we can render is a QuadTree. If you're not aware, a quadtree is a spatial partitioning algorithm that can help reduce the number of bodies that you need to test for collisions (or other spatially relevant information). In Phaser, it only exists as a part of the Arcade physics system, and is disabled by default. We can enable it by setting `game.physics.arcade.skipQuadTree = false;` and then each physics body will automatically be added to the quadtree. Phaser's QuadTree class

however can be used outside of the physics system for any purpose you'd like, and the
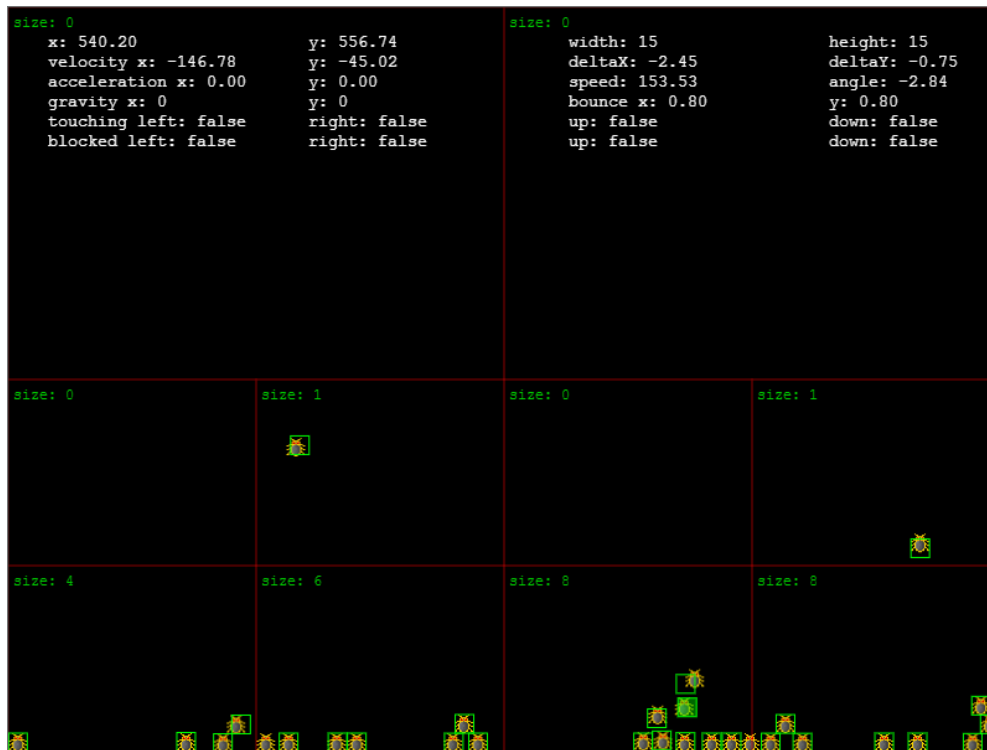
debug quadtree renderer will render any quadtree derived from that class.

Phaser also supports the Box2D physics system via a premium plugin available from the website. If you have this plugin, there are 2 built in functions in the Debug object for displaying information about Box2D bodies. The methods are box2dBody and box2dWorld. Both of them use the default Box2D debug canvas renderer, all Phaser is doing is simply providing access to the debug canvas context to Box2D. If you decide to purchase this plugin it comes with much more information and documentation.

Debugging in our own projects

The methods built into the Debug object cover some of the most common, and also most basic scenarios when it comes to getting debug information about your game. When we're building our own projects, there is likely going to be more information that you need to be able to see than Phaser can anticipate. This is where the basic blocks such as text and geom really come into play.

We should then dive into the source code of Phaser (it's not scary, it's just javascript!) and take a look at the implementation details for other methods such as spriteInfo.

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Page 325

```
1  spriteInfo: function (sprite, x, y, color) {
2
3        this.start(x, y, color);
4
5        this.line('Sprite: ' + ' (' + sprite.width + ' x ' +
sprite.height + ') anchor: ' + sprite.anchor.x + ' x ' +
sprite.anchor.y);
6        this.line('x: ' + sprite.x.toFixed(1) + ' y: ' +
sprite.y.toFixed(1));
7        this.line('angle: ' + sprite.angle.toFixed(1) + '
rotation: ' + sprite.rotation.toFixed(1));
8        this.line('visible: ' + sprite.visible + ' in camera: '
+ sprite.inCamera);
9        this.line('bounds x: ' + sprite._bounds.x.toFixed(1) +
' y: ' + sprite._bounds.y.toFixed(1) + ' w: ' +
sprite._bounds.width.toFixed(1) + ' h: ' +
sprite._bounds.height.toFixed(1));
10
11        this.stop();
12
13    }
```

We can see that they use the sprite reference that we pass in, as well as the x & y coordinates in combination with the start method that I mentioned at the beginning. Then using the line method, render lines of details about the sprite to the screen.

Let's say that we don't just want to display sprite information generically, but we have created a sprite object and attached several other properties to it, to make it represent the player in the game. The spriteInfo method doesn't give us any details about these extra properties (it doesn't even know what they are!), nor does it give us any information about other things that already exist on a sprite such as the health and max health. In this example our plane is tracking a lot of different things, and we will want to be able to report about some of them.

```
1          let bullets = game.add.group();
2      bullets.enableBody = true;
3      bullets.physicsBodyType = Phaser.Physics.ARCADE;
4      bullets.createMultiple(100, 'bullet');
5      bullets.setAll('anchor.x', 0.5);
6      bullets.setAll('anchor.y', 0.5);
7      bullets.setAll('outOfBoundsKill', true);
8      bullets.setAll('checkWorldBounds', true);
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
9
10        let plane = this.plane = this.game.add.sprite(200,
200, 'plane');
11        plane.heal(Infinity);
12        plane.anchor.setTo(0.5);
13        plane.scale.setTo(0.5,0.5);
14        this.game.physics.enable(plane,
Phaser.Physics.ARCADE);
15        // animation
16        plane.animations.add('fly');
17        plane.animations.play('fly', 30, true);
18        plane.sounds = {
19            engine: this.game.add.audio('plane_engine'),
20            gun: this.game.add.audio('plane_gun'),
21            die: this.game.add.audio('boom')
22        };
23        plane.sounds.engine.loopFull(0.35);
24
25        plane.bullets = bullets;
26
27        plane.fireTimer = this.game.time.now;
28
29        plane.events.onKilled.add(this.planeKilled, this);
30
31        plane.score = {
32            killsByCollision: 0,
33            killsByShooting: 0
34        };
```

In order to display some useful information, we'll need to create a custom debugging method. Notice that we can display bits and pieces that we might find in other debug methods, as well as our custom properties.

```
1    displayDebugInfo() {
2        this.game.debug.start(32, 32);
3        this.game.debug.line(`Health:
${this.plane.health}/${this.plane.maxHealth}`);
4        this.game.debug.line(`FireTimer:
${this.plane.fireTimer}`);
5        this.game.debug.line(`Gun Sound:
${this.plane.sounds.gun.currentTime} Playing:
${this.plane.sounds.gun.isPlaying}`);
6        this.game.debug.line(`Plane Loc: (${this.plane.x},
${this.plane.y})`);
```
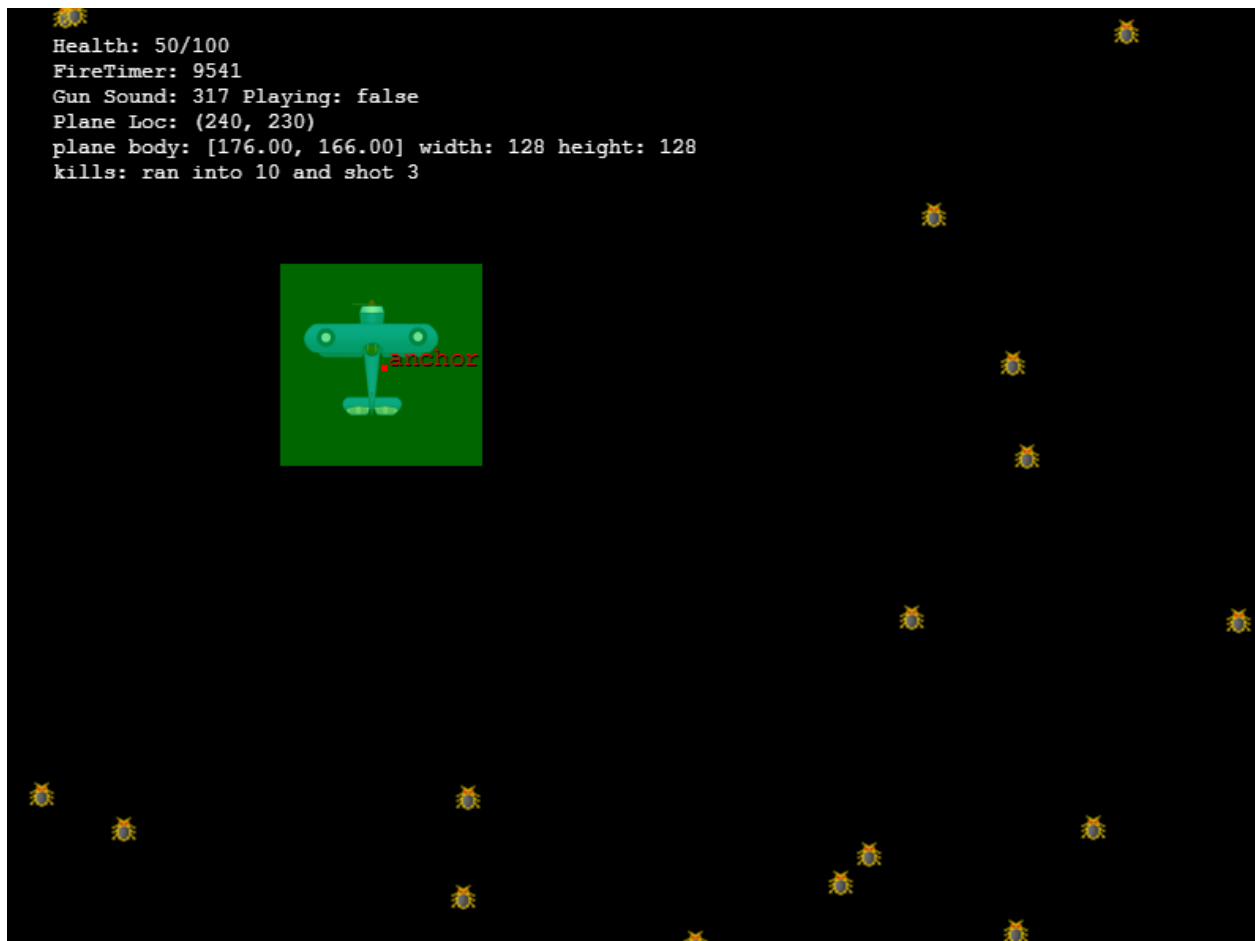
```
7          var body = this.plane.body;
8          this.game.debug.line(`plane body:
[${body.x.toFixed(2)}, ${body.y.toFixed(2)}] width:
${body.width} height: ${body.height}`);
9          this.game.debug.line(`kills: ran into
${this.plane.score.killsByCollision} and shot
${this.plane.score.killsByShooting}`);
10         this.game.debug.stop();
11
12         this.game.debug.body(this.plane);
13
14         this.game.debug.text('anchor', this.plane.x + 4,
this.plane.y, 'red');
15         this.game.debug.pixel(this.plane.x, this.plane.y,
'red', 4);
```



```
16      }
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Page 328

One thing that I immediately find useful is the rendering of the anchor and physics body. First we can see that the anchor for the sprite is not actually centered on the plane itself. Also the physics body is not centered on the plane, as well as it is taking up a lot more space than the actual plane image as well. This means that we are going to collide with enemies at such a distance that isn't going to feel right to players. From the plane body text information, we can see that the width and height are both 128. The actual size of the plane (each frame) is 176. Since I'm also scaling the plane down by half, it would seem that Phaser has originally bumped my image up to 256. Now I know where to look to solve the problem, something to do with the image size.

I used Texture Packer to generate the 3 frame sprite for the plane from 3 separate images. (If you haven't heard of Texture Packer, I highly recommend that you check it out, it's a really handy tool for dealing with lots of sprites in an organized and efficient way.) I loaded the JSON atlas file that it generated into Phaser, and this is the result I got. The original 3 images were each sized 256 x 256 so that is where the size came from (Phaser didn't do anything to them). It seems to be reading the source image size from the JSON file. As a quick test, I switched from loading the JSON atlas file to just loading a spritesheet and supplying the dimensions for each frame. That solved the problem, at least in the case that I only have 1 plane animation on the sheet, and all of the frames are the same size (which is true in this case). I looked online and found this forum post in which another user was having similar trouble. I found that using Texture Packer's "trim" feature was the issue, Phaser will use the source size in this case because the sprites are being packed for the purposes of image memory consumption, but still need to be rendered as if they were separate images. In my case however, I want to use "crop" mode as I don't want this behavior, I want to use it in packed form. Changing the output to

cropped did the trick! Now the collision body and the anchor line up, and everything is
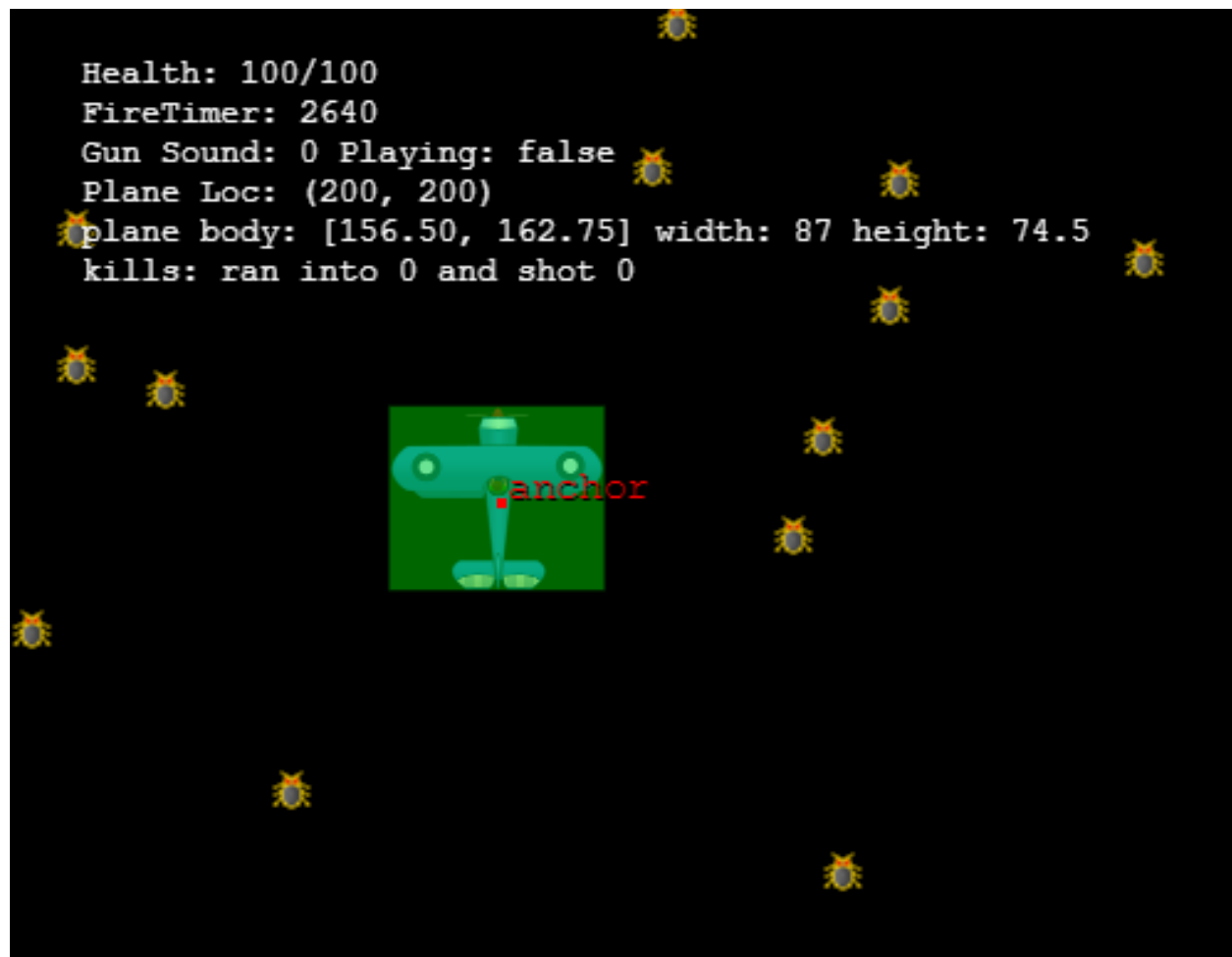
working the way that I wanted.

Hopefully you can see now how knowing how to display information about your game can help with debugging. The process that I just went through I did not simulate, I actually created the demo for this and found the issue with the sprites after I turned on the debugging tools. (Worked out great for me to have a real issue while writing the tutorial!) I mentioned earlier that having this debug information showing is expensive, and of course you don't want it to display in the final version of your game. You can go back and comment out or delete the code that displays the debug info, but that can get tedious after a while. Instead you can create a simple flag in the code that determines whether or not it should show.

```
1    init() {
2        this.game.stage.disableVisibilityChange = true;
3
```

```
4          this.debugMode = false;
5      }
```

First I add the test to the init phase of my state. Then I can simply wrap the *displayDebugInfo* method in a conditional in the render method.

```
1          render() {
2          if (this.debugMode) {
3              this.displayDebugInfo();
4          }
5      }
```

You can manually toggle that flag that is in the init method, or you can also setup a key to toggle it at runtime. Something like this should do the trick:

```
1          toggleDebug() {
2          this.debugMode = !this.debugMode;
3          if (!this.debugMode) {
4              this.game.debug.reset();
5          }
6      }
```

```
1 // inside create
2 this.debugKey =
this.game.input.keyboard.addKey(Phaser.Keyboard.D);
3 this.debugKey.onDown.add(this.toggleDebug, this);
```

Now you should be able to turn on and off debug mode using the D key on the keyboard (feel free to change it to whatever you like). One final method to mention about the Debug object in Phaser is the reset method. This will clear the canvas or sprite image so that even when debug mode is "off" it won't render the last thing that it rendered. Remember, debug is always there, you just can't see it until you start drawing to the surface.

Go forth and Squash!

Thanks for taking the time to read this guide about Phaser Debugging. As game developers (and software developers) we always have bugs to deal with, hopefully this guide will help you squash them quickly.

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Page 331

Phaser Texture Atlas Tutorial – How to Make Awesome Sprites From Scratch
By Kristen Dyrr

There are plenty of sites where you can find free images for your games, but sometimes it can be difficult to find exactly what you're looking for. What if you could create your own Phaser game images from scratch, even if you have no artistic abilities? In fact, wouldn't it be great if you could create game sprites and texture atlases based on objects, people, and animals from your own life? That is exactly what we will be covering in this tutorial!

Tutorial asset files

You can download the tutorial source code files here. All the project files are located in the main folder. The asset folder contains additional intermediate files that were used during the process of creating the game sprites.

Tutorial goals

This tutorial will explain how to create cartoon versions from real photos for use within any game, with a focus on Phaser.

1. Learn how to cut out a portion of a photo and clean it up in preparation for creating a cartoon version for use in a game.
2. Learn how to create a cartoonized version of a photo.
3. Learn what a texture atlas is and how to create one using TexturePacker.
4. Learn how to load a texture atlas into Phaser and use it to create animations.

Tutorial requirements

- Basic knowledge of JavaScript may be helpful for the Phaser portion of the tutorial.
- Download and install the free GIMP photo editing software.
- Download and install the free Inkscape vector image software. If you use Mac OS X, you will need to download and install XQuartz first.
- Download Phaser from it's Github repo. You can either clone the repo or download the ZIP file.
- Download and install TexturePacker. This is the only software we will be using that is not free, but you can use it to test your texture atlases and animations for free.

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Find an image

Since we are focusing on creating images for this tutorial, we will begin with an image. For my last tutorial, I wanted to create some dog animations. I couldn't find free dog photos that I needed, and I have absolutely no artistic drawing capabilities, so my brother provided me with a few photos. I will focus on just one of the photos, because it was the most difficult one to clean up, and covers just about all the processing you may come

across.

As you can see, the dog (Max) in this photo is not only skewed, but one of his feet is completely out of the frame, and he's wearing a harness! Don't worry, we can fix all of these problems, and it's not as difficult as you may think.

Cut out the image with GIMP

First of all, we need to cut Max out of the photo. If we open the photo in GIMP, we can use the Free Select Tool from the toolbox to select our shape. Simply start anywhere along the edge of the shape you wish to pull out and continue clicking along the entire perimeter. Finish the shape by clicking on the starting point.
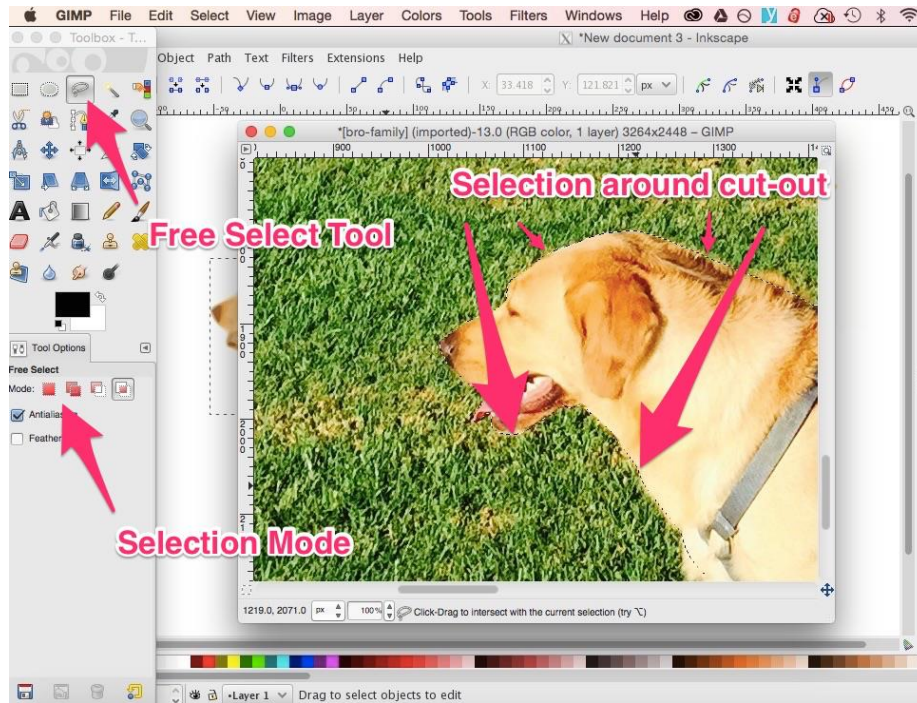
If this does not create a selection area around the object (moving dotted lines), check the "Mode" under the "Free Select" section of the Tool Options (which you can find under Windows -> Dockable Dialogs, if it's not already open). You will want to select a mode that either replaces the current selection or adds to it.

Once the object is selected, copy it (Edit -> Copy). Create a new GIMP file (File -> New). In the Create a New Image dialog box, choose Advanced Options -> Fill with -> Transparency, then press OK. The size of the new file doesn't matter, as long as it's
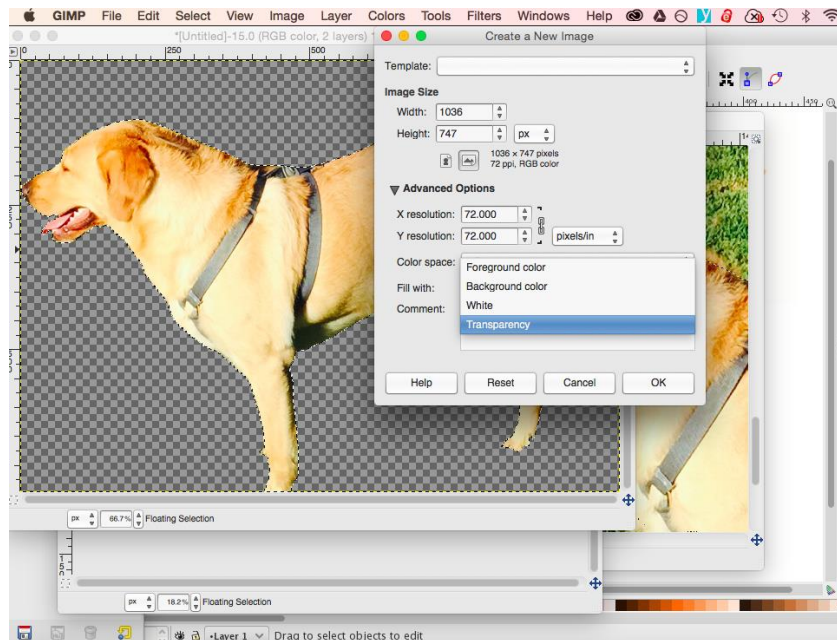
larger than the object you are pasting (you can fix it with Image -> Canvas Size). Paste your selection into the new file. This is where it can get difficult, but you don't really need to do more than necessary. Since we will be turning the photo into a small cartoon, it doesn't have to be perfect.

Edit the image in GIMP

The most glaring problem with the photo is that it is skewed. All we need to do to fix this is choose the Rotate Tool from the toolbox and begin rotating the image on the screen. A dialog box will appear, and you can just move it aside so you can see what you're doing.

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Once the two front feet appear to be flat on the ground (taking into account the foot that
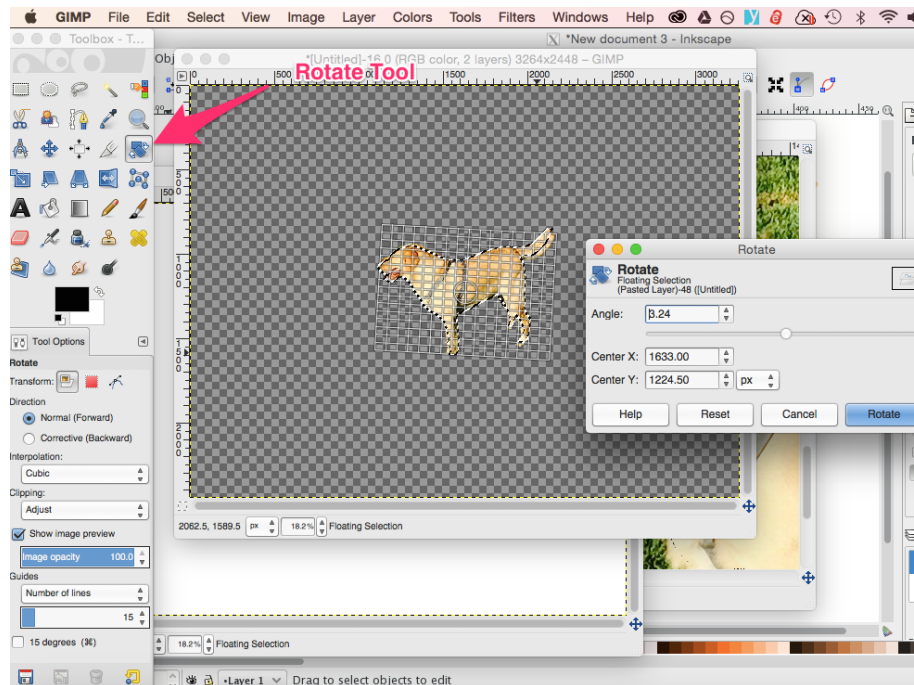
is outside the frame, which we'll fix later), we're finished rotating.

The next problem is the harness. This is relatively easy to fix by using the Clone Tool from the toolbox. Once the tool is selected, and the cursor is over the image, you will see a message telling you what key you need to press (it's different for every operating system) in order to set a source. Just hold down that key and click on some of the fur next to the harness.
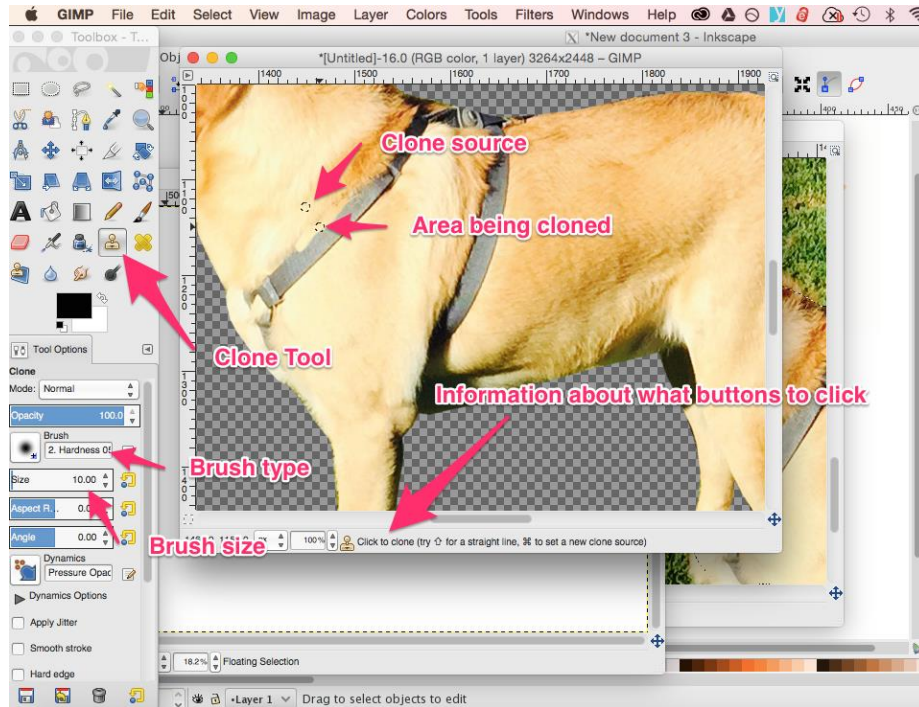
Once the source is set, begin coloring over the harness. You will want to keep the portion you are coloring in line with the source so you don't get too many artifacts. Select a new source whenever you move to a section that needs a different color. You want to try to

match the part you're coloring to the fur that is closest to it so it will look as seamless as
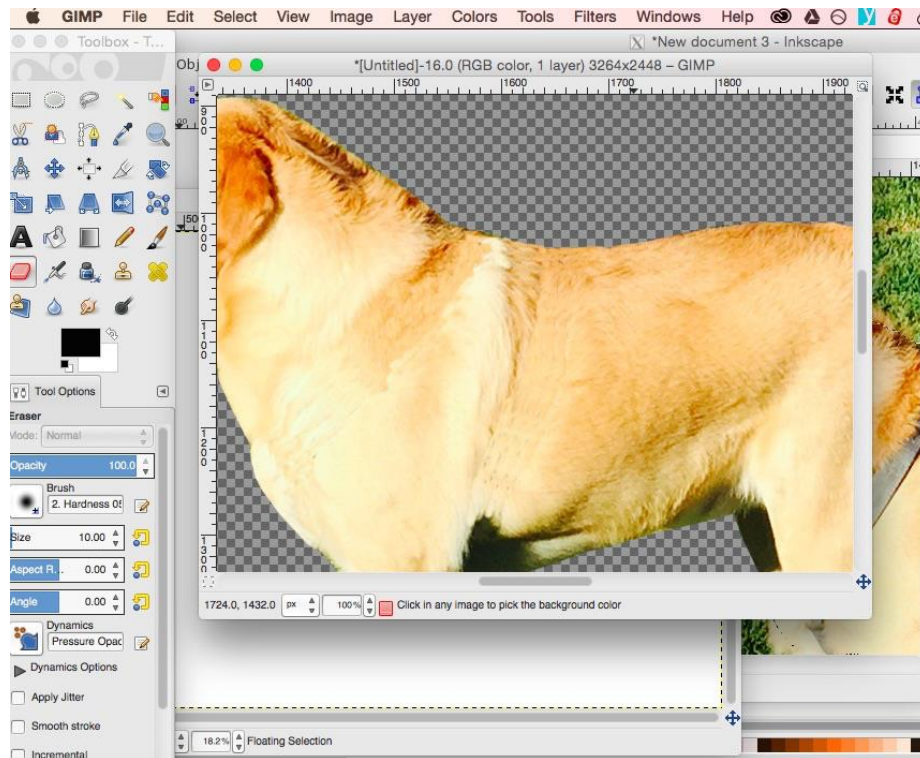
possible.

Amazingly, we can use the same Clone Tool trick for filling in blanks. For example, you can also use the Clone Tool to fill in the missing foot that was out of the frame!

We also want to flip the image so Max is facing in the other direction. That's easy, as you
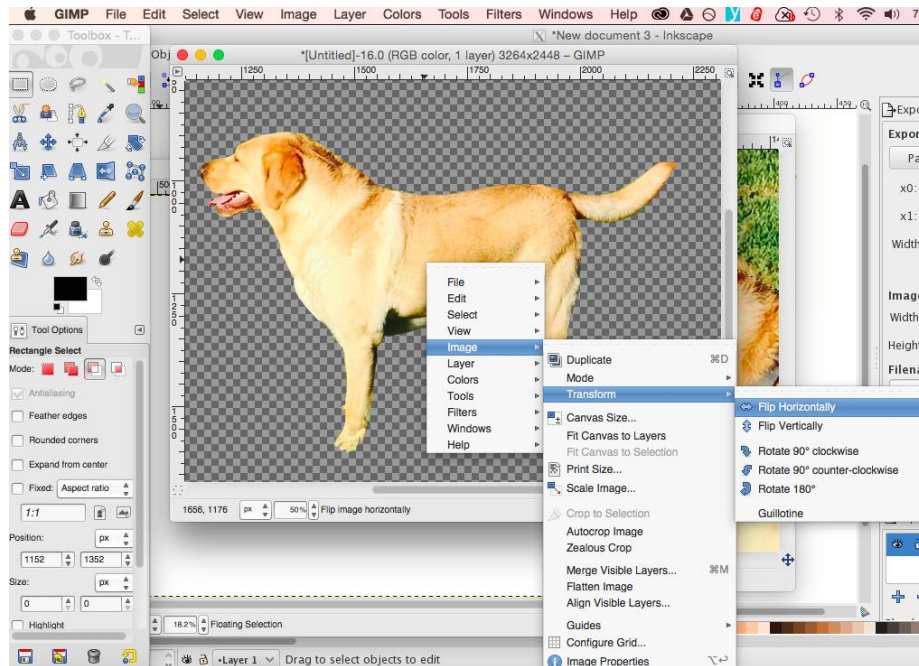
can see below:

See how the back foot seems way too far up for the perspective we are attempting to achieve? All I did for that was cut the back foot off using the Rectangle Select Tool, created a new layer in the Layers window (Windows -> Dockable Dialogs if it's not already there), then pasted the foot. Then I moved the foot down until I had the perspective I wanted, clicked back on the main layer, and used the clone tool to fill in the gap. I also copied a few details from the other leg, although that's not entirely necessary.

I also moved the layers around until I had them in an order that was easier to work with.

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Create a new animation frame

Once we are happy with the first image, we will have to decide how we want to animate it. I created a very simple animation with just two frames. I decided that it would be easiest to use the photo resulting from the edits already made as the "step" frame, then remove the back legs to create a frame where Max appears to be standing straight.

In order to do this, I made a copy of Max's front leg, pasted it in a new layer, rotated it lengthened it, and used the clone tool to fill in any problem areas (similar to what we did with the back foot. I also used the Eraser Tool to erase part of the foot so it wouldn't look
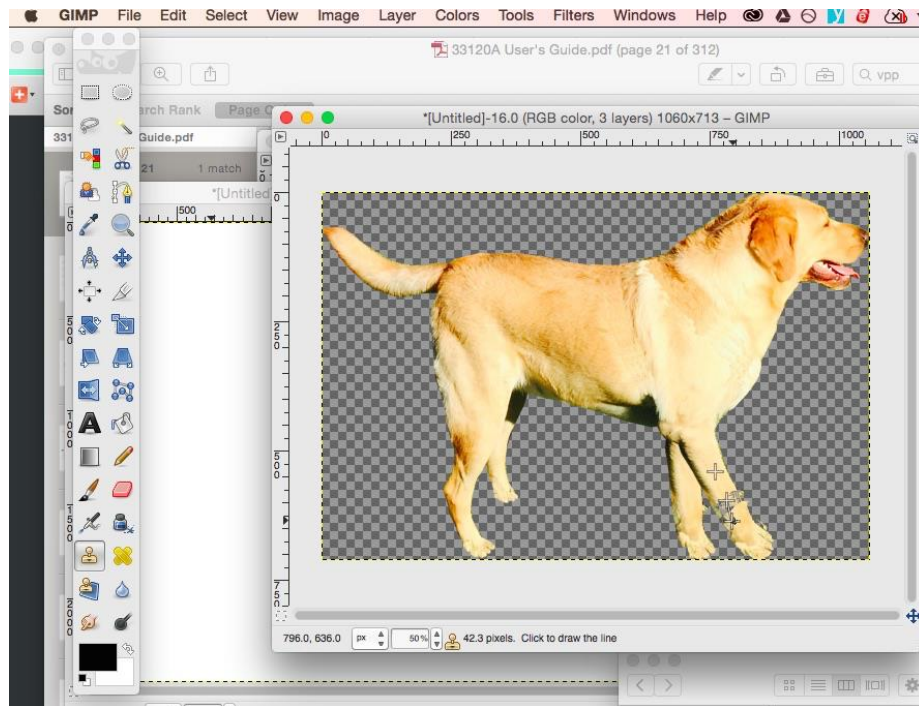
like he had two of the same feet.
At this point, we select Image -> Autocrop Image, to make it the exact size of the edited dog. Then, we save the file. Select File -> Export, choose GIF as the file type, and save it. Now select File -> Save As to create a copy of the file. This will be our other animation frame.

Now we can simply delete all the layers in the Layers window, except our main layer. We also want to use the Eraser Tool to remove that back leg. Save the new file. Again, choose File -> Export to export our new GIF. We now have our other frame!
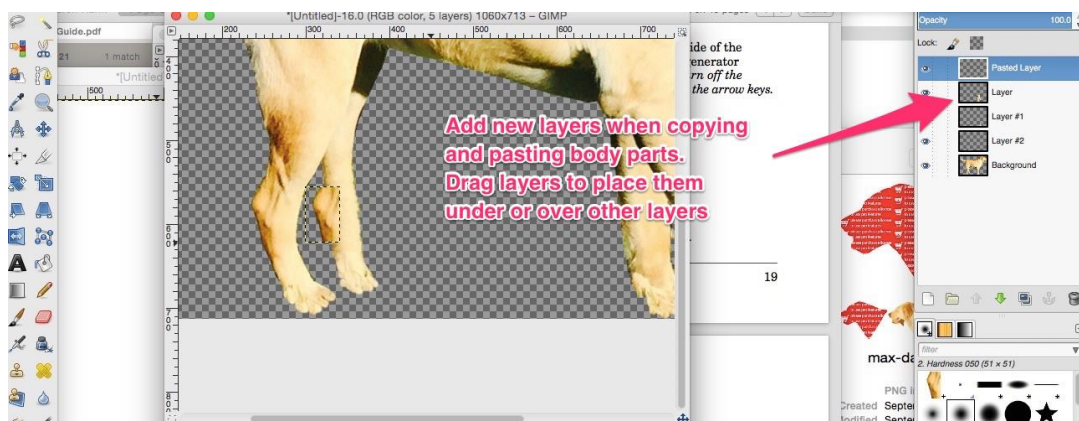
Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Convert photos to cartoons with Inkscape

Open one of the GIF files in Inkscape. It's OK to keep the default settings. Click the
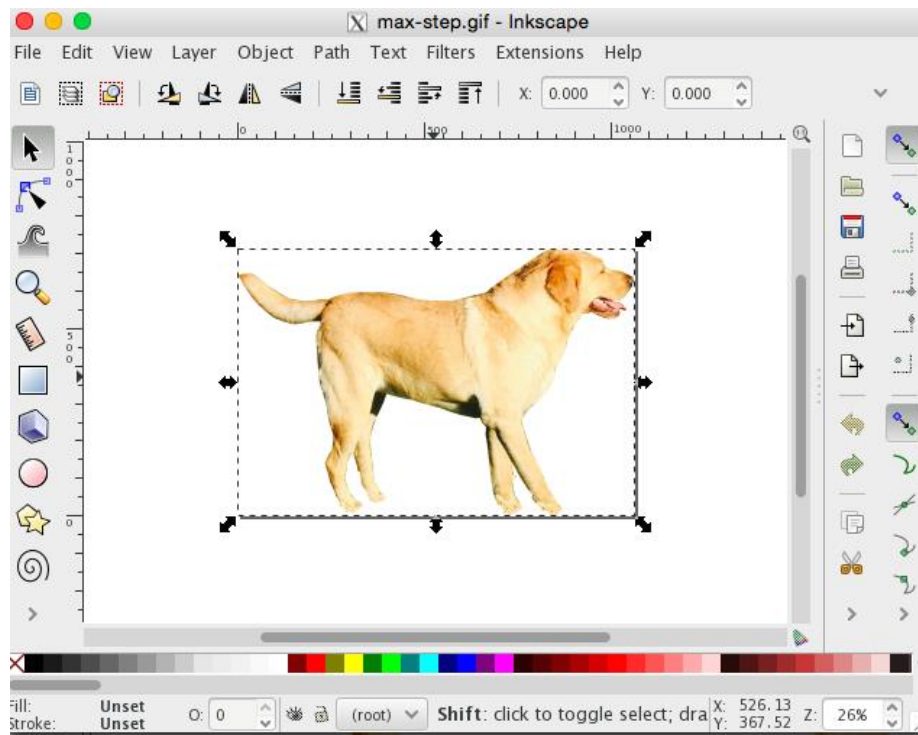
image on the page to select it, so that it has a bounding box around it like this:
Now select Path -> Trace Bitmap. Select "Colors," then check the "Remove Background" checkbox. The higher the number in the "Scans" dropdown, the more colors it will use, and the closer the vector will be to the original photo. Try different options and click

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

"Update" to preview them until you are happy with the outcome, then press OK. You
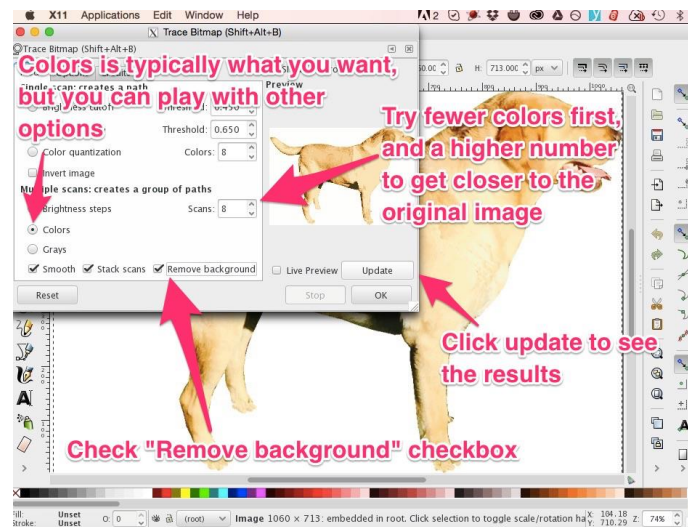
now have a vector cartoon version of your photo!

To export the new vector into the format needed for the game, click on the image you wish to select (so it has the bounding box around it), then go to File -> Export PNG Image. You will be presented with a large box on the right. Click the "Selection" tab at the top of the box, then set the final width and height you would like for the image in your game (you will probably want a fairly small image unless this will serve as a background image). Click "Export As" to choose the location, then click "Save." Note that this does not save the image! Farther down in the box is a button that says "Export." Clicking this button will complete the final export of the image.
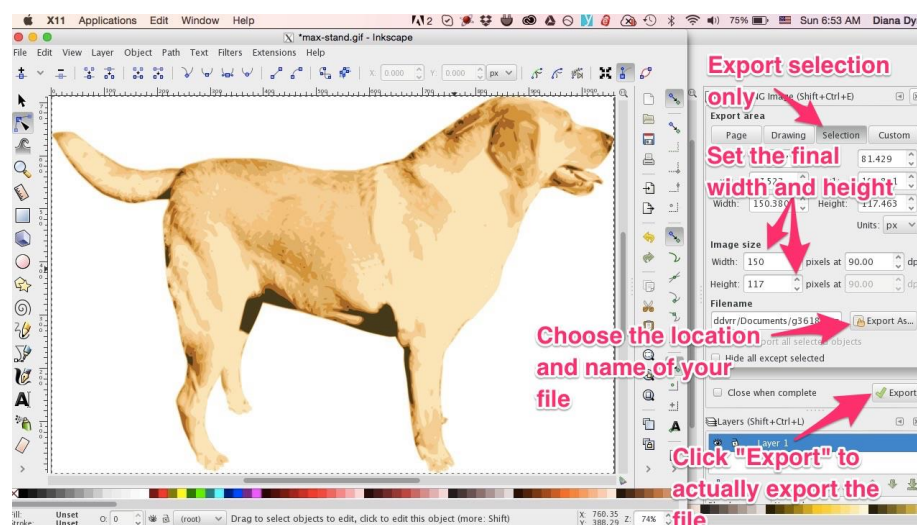
Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Page 342

You can now do the same with the other GIF photo you exported from GIMP. Another option is to create other frame or frames with Inkscape. This depends on what you want to do with your animation. Once you have converted your photo to a vector, click the "Edit paths by nodes" tool. When you move the cursor over the image, you'll see all sorts
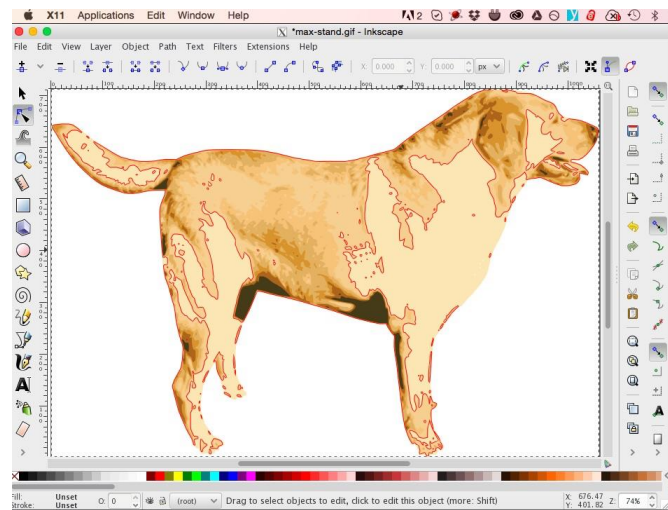
of lines:

If you click on the image, you'll wind up with many points that can be moved around. Depending on the complexity of your image and what sorts of changes you wish to make

in each frame, moving these points around may be the easier route to creating additional
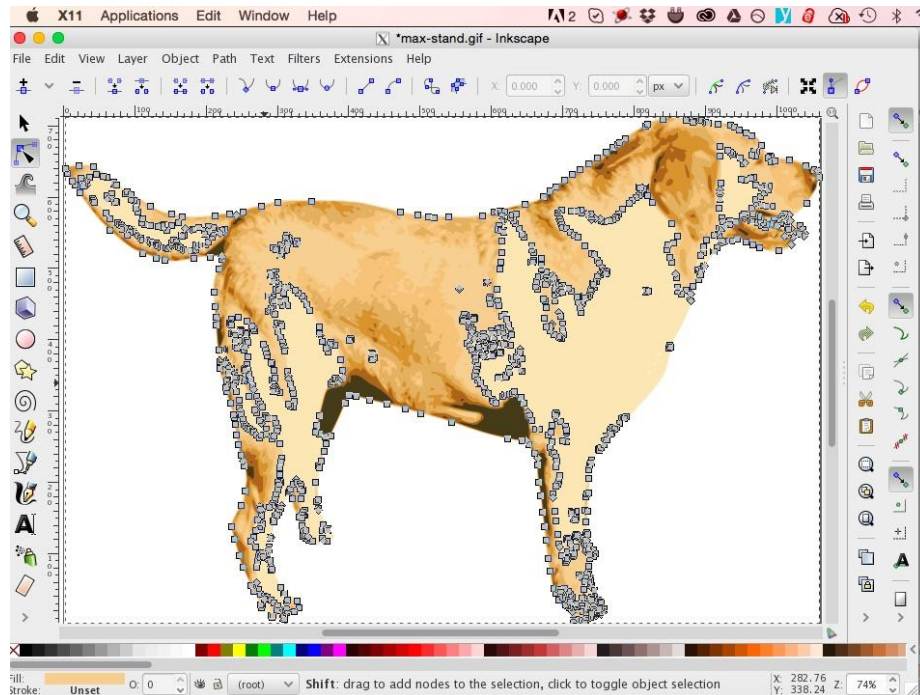
animation frames.

Create a texture atlas with TexturePacker

A texture atlas is a large "spritesheet" consisting of all the sprites (or images) that will be used in a game. The spritesheet is accompanied by a data file that details the exact location and size of every sprite, along with keys (labels) that can be used to reference each sprite.

TexturePacker makes it extremely easy to add all of your game images and animation frames into a single spritesheet file, as long as you follow a couple of important rules. The first important step is to choose a Phaser-specific option. In this case, we chose "Phaser (JSONHash)" as the data file format when creating a new TexturePacker project. Once you click "Create project," everything is simple.

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

You can either drag individual files or folders, or use the "Add sprites" or "Add smart folder" buttons to choose the images you will use in your game. This allows you to organize all of your images. If you have multiple characters that are animated with many frames, for example, you could create folders for each character that contain animation images for that character, then drag the entire parent folder into TexturePacker. The animation frames can contain a naming convention that will make adding the frames to the game more intuitive (TexturePacker automatically uses the file name as the key you will use to reference the frame in the game).

The second important rule when creating your file is one that you will probably never encounter unless you try to use the same image file in multiple game frameworks. If you

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

were to choose the generic "JSON (Hash)" for the data file type rather than the Phaser-specific option, TexturePacker would create smaller spritesheets by rotating some of the images on the sheet. The problem is that Phaser is not compatible with rotation. In order to fix the problem, you would click the "show advanced" link on the "Layout" section of

the Settings box on the right. Unchecking "Allow rotation" will fix the problem.
All that is left is to set the location and name for your "Data file" and "Texture file" in the Settings box on the right. Clicking the file selector button (the little folder picture) next to each box will allow you to search for the location of your project and set the name you would like in the "Save As" box. In most cases, you will not need to touch any other settings. Once you're finished, just click the "Publish sprite sheet" button and you're
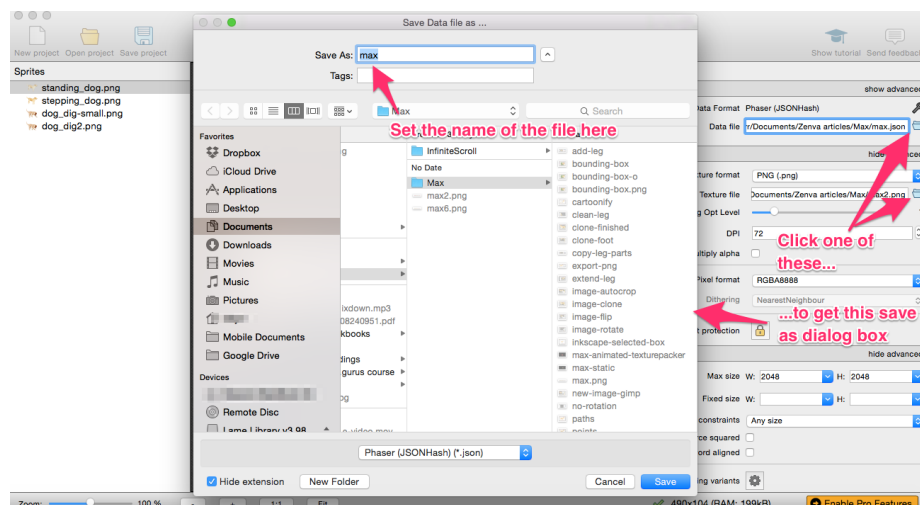
done!
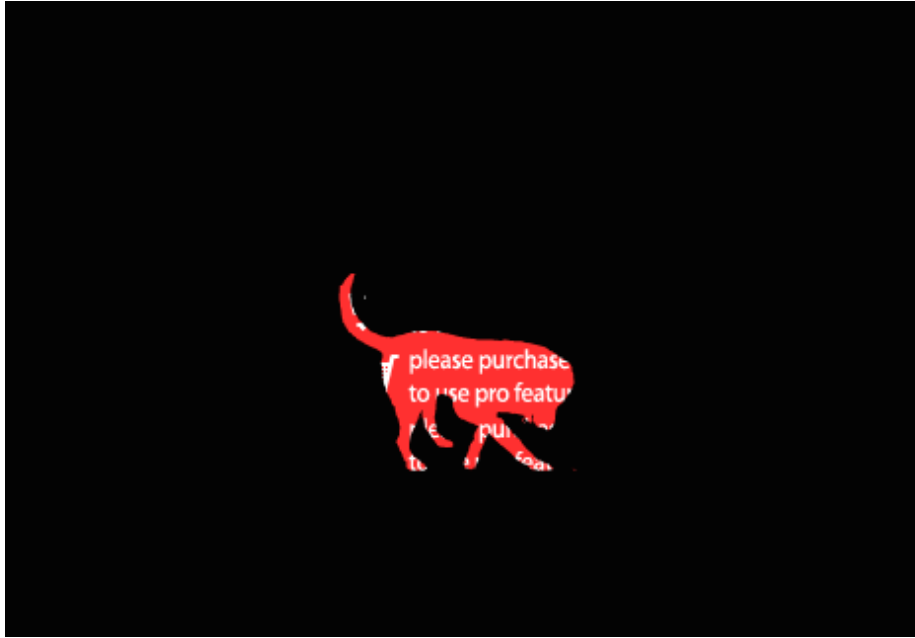
If you're using the free version of TexturePacker, some of your sprites will be filled in with a solid red color with white text stating, "please purchase a license to use pro features." It's OK for now, because the sprites retain their shape, which means that you can still use the texture atlas to test your project, as I've shown below:

Load the sprites into Phaser

The Phaser project begins in a new folder containing an index HTML file, the Phaser javascript file, and the spritesheet and JSON data files you created in TexturePacker. Phaser can be copied from the "build" directory of the downloaded Phaser zip or cloned repository.

We are creating an extremely simple Phaser project for the purpose of testing our texture atlas, so our file structure is going to be minimal. Normally, we would create a game with states and organize multiple files in additional folders. To learn more about states and the best way to create a full game, please see this Phaser tutorial. For now, the following code goes into the index.html file.

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <meta charset="utf-8" />
5          <title>Learn Game Development at ZENVA.com</title>
6          <script src="phaser.min.js"></script>
```

Zenva Academy – Online courses on Phaser and game programming
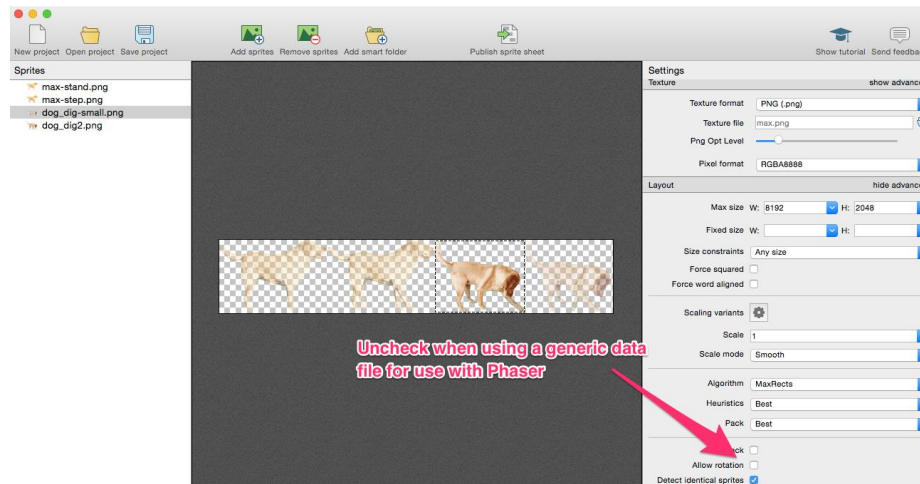Zenva for Schools – Coding courses for high schools

```
7     </head>
8     <body>
9        <!-- include the main game file -->
10       <script src="max-animation.js"></script>
11     </body>
12 </html>
```

Our game is actually initiated within the max-animation.js file, and the Phaser framework is included with the phaser.min.js file. For the purposes of testing out our images, we will just be doing a very simple test within a single javascript file rather than using a set of state files.

Test the animations

Here's the most basic javascript file needed for testing our animations in Phaser:

```
1 window.onload = function() {
2
3   var max;
4
5   var game = new Phaser.Game(800, 400, Phaser.CANVAS, '', {
preload: preload, create: create, update: update });
6
7   function preload () {
8      //load the json from TexturePacker
9      game.load.atlasJSONHash('max', 'max.png', 'max.json');
10   };
11
12   function create () {
13      //Load the initial sprite
14      max = game.add.sprite(0, 180, 'max', 'max-step1');
15
16      //create the animation for walking using the frame names
we want from max.json
17      max.animations.add('walk', [
18          'max-step1',
19          'max-step2'
20      ], 10, true, false);
21
22      //same for the digging animation
23      max.animations.add('dig', [
24          'max-dig1',
25          'max-dig2'
26      ], 10, true, false);
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Page 348

```
27
28      //start up the walk animation
29      max.animations.play('walk');
30    }
31
32    function update() {
33      //once Max walks a bit, have him stop and dig
34      if(max.x >= 400){
35        max.x = 400;
36        max.animations.play('dig');
37      }
38      else {
39        //otherwise, have him move to the right
40        max.x += 3;
41      }
42    }
43  };
```

The "max" variable is our main character in the game. The next statement creates our game canvas. To learn more about the game creation statement and the Phaser.CANVAS option, please see my previous tutorial.

Next comes three functions that are predefined in Phaser, and will need to be extended in order to make the test work. The preload function is where all images, sounds, and other assets should be loaded so they are available before the game actually begins.

In our case, we have just a single image with a data file, which must be loaded according to the type of data file we created in TexturePacker. We must use the atlasJSONHash spritesheet loading function, since that was the option we selected in TexturePacker. The first argument is the key that will be used to reference the texture atlas and create sprites.

The create function is where we create all of the game sprites and other things that will be used throughout the game. We begin by creating our initial character sprite. The first two arguments are the x and y location for placing the sprite, which begins at (0, 0) in the upper left corner of the canvas. The next argument is the key referencing the texture atlas we loaded in the preload function. The last argument references one of the frames listed in the JSON texture atlas file. In most cases, this will be the name of the image file loaded into TexturePacker, but you can also change the names manually in the JSON file.

The purpose of a texture atlas is to load one small image file that contains all the sprites you need for your game. Otherwise, you would have to load multiple files, which can

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

slow the loading time. That means you can include all the images you need for the game using TexturePacker, including the background image and other static images. For example, if you were to comment out everything after the following statement in the "create" method, you would be left with just the first static image:
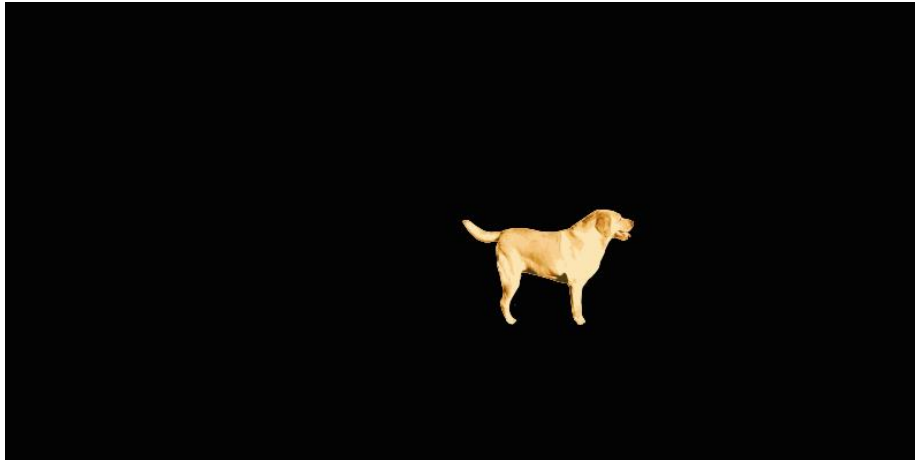

```
1 max = game.add.sprite(0, 180, 'max', 'max-step1');
```
The next two statements will create the animations that may be played at any time throughout the game. If we had used a simple spritesheet rather that a texture atlas, we would have loaded many separate spritesheets for each animation, added each sprite separately, then created an animation for each one. My previous Phaser tutorial discusses this method.

```
1 max.animations.add('walk', [
2       'max-step1',
3       'max-step2'
4   ], 10, true, false);
5
6 max.animations.add('dig', [
7       'max-dig1',
8       'max-dig2'
9   ], 10, true, false);
```

In this case, we need to reference each frame within the texture atlas data file, as we did for the single sprite. The first argument is the key you will use to initiate the animation, and the next argument is an array referencing each frame from the data file. The next argument is the speed in frames per second (fps). For more information on fps, please see this other Phaser tutorial. The next argument is true if we want the animation to loop, and having "false" in the last argument means that the listed frames are strings ("true" would

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

mean we listed the frames as numbers, where each number corresponds to the order it appears in a spritesheet).

Lastly, we begin one of the animations by using the play function, and passing in the key we created that references the walking animation. If we were to stop right here, our character would merely appear on the screen and animate, but stay in the same position. That looks a little weird, so we use the "update" function to make our animation do something interesting.

```
1 max.animations.play('walk');
```

The update method is automatically called "many times per second." Therefore, this is the place where main game play happens, because we do all of our testing here to see when things change. We can do all sorts of fun things with game physics, but since we are only testing our texture atlas, we are instead going to continually add to the "x" location of our main character so he appears to be walking across the screen.

```
1 max.x += 3;
```

In order to test the other animation I've added to the texture atlas, I change max.x to a constant variable when he's partway across the screen, then switch the animation by simply playing the "dig" animation. Rather than make things more complex by adding timers and continuing the animation, the page must be reloaded if you want to see it again. My previous tutorial explains how to add timers to allow an animation to run for a short time, then continue on.

```
1 if(max.x >= 400){
2    max.x = 400;
3    max.animations.play('dig');
4 }
```

And there we have it! You have tested your texture atlas, and it is now ready to use in a full game!

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Additional Resources

There are other types of game images that can be added to your games, and many ways to add various images in Phaser. For a complete discussion of tiling to create game levels, please see this Tiled tutorial. There is also this Phaser tutorial that discusses a different way to tile game levels. You can also check out this tutorial, which discusses an interesting way to add sprites and animation. Finally, my previous Phaser tutorial discusses ways to create random sprite animations for an infinitely scrolling game.

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

How to use Pathfinding in Phaser
By Renan Oliveira

Suppose you're building a strategy game. The game will have player and enemy units which navigate through a map full of obstacles. While the player units will be controlled by clicking on the map positions, the enemies will walk alone, following any AI strategy.

You may have noticed that, given an origin and target positions, the units must find a path connecting those two points avoiding any obstacles in the way. In addition, it is important that this path is the shortest as possible.

The process just described is called pathfinding and it is used in many games, as well as other AI problems. In this tutorial I'll show how to use a pathfinding library to solve this problem in your game. At the end, we will build a map where we can click to where we want our character to move, and it will find the shortest path from its position to the desired position.

To read this tutorial, it is important that you are familiar with the following concepts:

- Javascript and object-oriented concepts.
- Basic Phaser concepts, such as: states, sprites, groups and arcade physics
- Creating maps using Tiled

Source code files

You can download the tutorial source code files here.

What is pathfinding?

Before writing our pathfinder, I'm going to explain by means of an example, what is our goal, and how it can be achieved.

The figure below shows a character in a tile map with obstacles. Suppose we want this character to move to the highlighted position. Some possible paths are highlighted in the map, but we want to find the shortest one. How can we do that?

A common way of solving this problem is to start from the original position and to expand the search range by checking neighbor tiles, while avoiding obstacles, as showed in the example below.

However, we must choose an efficient way of expanding the search range to reduce the time we have to spend looking for a path. By simply expanding to all directions we can guarantee a shortest path, but at a high execution time, since we're expanding to directions that would never lead to the target position. This high execution time would not be noticed for a single path, but in a game where we must constantly calculate paths for several units, this runtime can easily become prohibitively long.

The A* (pronounced as "A star") algorithm is an efficient way of solving the pathfinding problem. The algorithm follows the idea of expanding the search range, but uses an heuristic to efficiently guide the search towards the target position. The idea is to calculate the cost of a tile by its distance to the original position plus an estimation of its distance to the target position. Then, by considering the lowest cost tile when expanding the search range, we can efficiently guide the algorithm. This video compares the A* algorithm with other algorithms that were not designed to handle obstacles, and you can clearly see its efficiency.

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

The EasyStar library

Instead of writing our own A* code, we are going to use a pathfinding library called <u>EasyStar</u>. This is an open source library created by Bryce Neal and available through the MIT license, which allows commercial usage. He also provided a <u>github repository</u> where you can check the source code and its usage. In this tutorial, I'll cover the basic methods we'll need for our pathfinding demo.

Creating the Tiled map

The figure below shows the Tiled map I created for this demo. Feel free to create your own or to use mine, provided in the source code. If you're not familiar with Tiled, you can check our tutorial series on how to create a Platformer using Tiled, which cover in details how to use it. There are only two things you must be careful, because they will be important in our code

later. First, all obstacles must be in a layer named collisions, which must have a collision
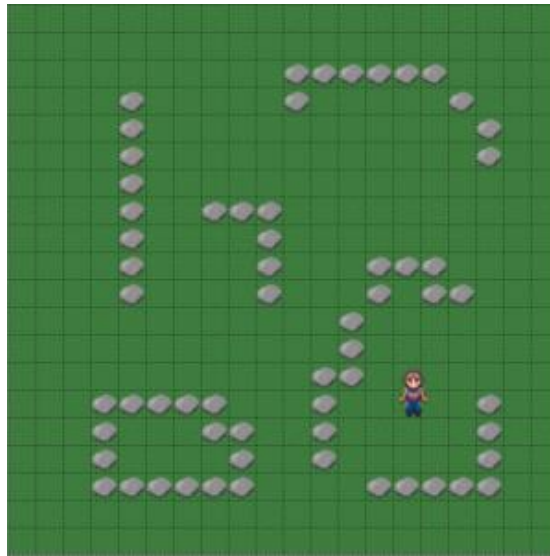
property defined as true, as shown below.
In addition, in the objects layer there must be a player object with its properties defined as below. This will be the character in our demo.

After creating your map, save it in the JSON format.

Phaser states of our demo

The assets, groups and map of our demo will be described in a JSON file, as below. Then, our game will have the following states, responsible for loading this file:
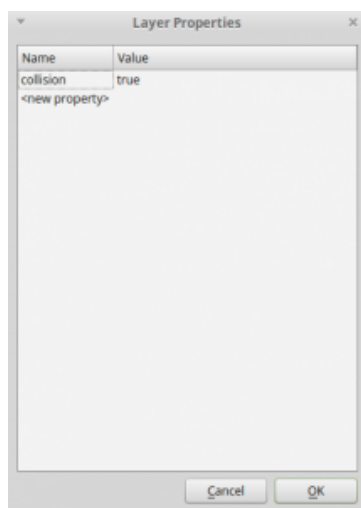
Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

- BootState: loads the JSON file before calling LoadingState.
- LoadingState: loads all game assets, including the JSON map file. After all assets are loaded, it calls WorldState.
- WorldState: create the map layers, game groups and prefabs.

The code for BootState and LoadingState are shown below. Notice that the LoadingState uses the asset key from the JSON file to load the correct asset.

```
1 var PathfindingExample = PathfindingExample || {};
2
3 PathfindingExample.BootState = function () {
4     "use strict";
5     Phaser.State.call(this);
6 };
7
8 PathfindingExample.BootState.prototype =
Object.create(Phaser.State.prototype);
9 PathfindingExample.BootState.prototype.constructor =
PathfindingExample.BootState;
10
11 PathfindingExample.BootState.prototype.init = function
(level_file, next_state) {
12     "use strict";
13     this.level_file = level_file;
14     this.next_state = next_state;
15 };
16
17  PathfindingExample.BootState.prototype.preload = function ()
{
18     "use strict";
19     this.load.text("level1", this.level_file);
20 };
21
22 PathfindingExample.BootState.prototype.create = function () {
23     "use strict";
24     var level_text, level_data;
25     level_text = this.game.cache.getText("level1");
26     level_data = JSON.parse(level_text);
27     this.game.state.start("LoadingState", true, false,
level_data, this.next_state);
28 };
```

```
1 var PathfindingExample = PathfindingExample || {};
2
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
3 PathfindingExample.LoadingState = function () {
4     "use strict";
5     Phaser.State.call(this);
6 };
7
8 PathfindingExample.LoadingState.prototype =
Object.create(Phaser.State.prototype);
9 PathfindingExample.LoadingState.prototype.constructor =
PathfindingExample.LoadingState;
10
11 PathfindingExample.LoadingState.prototype.init = function
(level_data, next_state) {
12     "use strict";
13     this.level_data = level_data;
14     this.next_state = next_state;
15 };
16
17 PathfindingExample.LoadingState.prototype.preload = function
() {
18     "use strict";
19     var assets, asset_loader, asset_key, asset;
20     assets = this.level_data.assets;
21     for (asset_key in assets) { // load assets according to
asset key
22         if (assets.hasOwnProperty(asset_key)) {
23             asset = assets[asset_key];
24             switch (asset.type) {
25             case "image":
26                 this.load.image(asset_key, asset.source);
27                 break;
28             case "spritesheet":
29                 this.load.spritesheet(asset_key, asset.source,
asset.frame_width, asset.frame_height, asset.frames,
asset.margin, asset.spacing);
30                 break;
31             case "tilemap":
32                 this.load.tilemap(asset_key, asset.source,
null, Phaser.Tilemap.TILED_JSON);
33                 break;
34             }
35         }
36     }
37 };
38
```

```
39 PathfindingExample.LoadingState.prototype.create = function
() {
40     "use strict";
41     this.game.state.start(this.next_state, true, false,
this.level_data);
42 };
```

The WorldState code is a bit more complex, and it is shown below. In the "init" method it initializes the physics engine and the map. Then, in the "create" method it creates all map layers, game groups and prefabs. Notice when a map layer has the collision property defined as true, as mentioned before, it is defined as collidable.

```
1 var PathfindingExample = PathfindingExample || {};
2
3 PathfindingExample.WorldState = function () {
4     "use strict";
5     Phaser.State.call(this);
6
7     this.prefab_classes = {
8         "player":
PathfindingExample.Player.prototype.constructor
9     };
10 };
11
12 PathfindingExample.WorldState.prototype =
Object.create(Phaser.State.prototype);
13 PathfindingExample.WorldState.prototype.constructor =
PathfindingExample.WorldState;
14
15 PathfindingExample.WorldState.prototype.init = function
(level_data) {
16     "use strict";
17     var tileset_index, tile_dimensions;
18     this.level_data = this.level_data || level_data;
19
20     this.scale.scaleMode = Phaser.ScaleManager.SHOW_ALL;
21     this.scale.pageAlignHorizontally = true;
22     this.scale.pageAlignVertically = true;
23
24     // start physics system
25     this.game.physics.startSystem(Phaser.Physics.ARCADE);
26     this.game.physics.arcade.gravity.y = 0;
27
28     // create map and set tileset
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Page 359

```
29    this.map = this.game.add.tilemap(this.level_data.map.key);
30    tileset_index = 0;
31    this.map.tilesets.forEach(function (tileset) {
32        this.map.addTilesetImage(tileset.name,
this.level_data.map.tilesets[tileset_index]);
33        tileset_index += 1;
34    }, this);
35 };
36
37 PathfindingExample.WorldState.prototype.create = function ()
{
38    "use strict";
39    var group_name, object_layer, collision_tiles;
40
41    // create map layers
42    this.layers = {};
43    this.map.layers.forEach(function (layer) {
44        this.layers[layer.name] =
this.map.createLayer(layer.name);
45        if (layer.properties.collision) { // collision layer
46            collision_tiles = [];
47            layer.data.forEach(function (data_row) { // find
tiles used in the layer
48                data_row.forEach(function (tile) {
49                    // check if it's a valid tile index and
isn't already in the list
50                    if (tile.index > 0 &&
collision_tiles.indexOf(tile.index) === -1) {
51                        collision_tiles.push(tile.index);
52                    }
53                }, this);
54            }, this);
55            this.map.setCollision(collision_tiles, true,
layer.name);
56        }
57    }, this);
58    // resize the world to be the size of the current layer
59    this.layers[this.map.layer.name].resizeWorld();
60
61    // create groups
62    this.groups = {};
63    this.level_data.groups.forEach(function (group_name) {
64        this.groups[group_name] = this.game.add.group();
65    }, this);
66
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
67    this.prefabs = {};
68
69    for (object_layer in this.map.objects) {
70        if (this.map.objects.hasOwnProperty(object_layer)) {
71            // create layer objects
72            this.map.objects[object_layer].forEach(this.create
_object, this);
73        }
74    }
75 };
76
77 PathfindingExample.WorldState.prototype.create_object =
function (object) {
78    "use strict";
79    var object_y, position, prefab;
80    // tiled coordinates starts in the bottom left corner
81    object_y = (object.gid) ? object.y - (this.map.tileHeight
/ 2) : object.y + (object.height / 2);
82    position = {"x": object.x + (this.map.tileHeight / 2),
"y": object_y};
83    // create object according to its type
84    if (this.prefab_classes.hasOwnProperty(object.type)) {
85        prefab = new this.prefab_classes[object.type](this,
object.name, position, object.properties);
86    }
87    this.prefabs[object.name] = prefab;
88 };
```

The "create_object" method is used to create all game prefabs. It identifies the prefab class by mapping the prefab type (defined in the Tiled map) to its constructor. This is done using the "prefab_classes" property, defined in the WorldState constructor. The prefab properties are obtained from the properties defined in the Tiled map. Notice that this is possible because all prefabs have the same constructor, shown in the code below.

```
1 var PathfindingExample = PathfindingExample || {};
2
3 PathfindingExample.Prefab = function (game_state, name,
position, properties) {
4    "use strict";
5    Phaser.Sprite.call(this, game_state.game, position.x,
position.y, properties.texture);
6
7    this.game_state = game_state;
8
```

```
9     this.name = name;
10
11      this.game_state.groups[properties.group].add(this);
12      this.frame = +properties.frame;
13
14      if (properties.scale) {
15          this.scale.setTo(properties.scale.x,
properties.scale.y);
16      }
17
18      this.game_state.prefabs[name] = this;
19 };
20
21 PathfindingExample.Prefab.prototype =
Object.create(Phaser.Sprite.prototype);
22 PathfindingExample.Prefab.prototype.constructor =
PathfindingExample.Prefab;
```

Creating a Pathfinding plugin

In Phaser you can create plugins that can be easily attached in your games, in a way you can reuse them. In this tutorial, we will write the pathfinding code in a plugin, so you can later use it in your own games. For more information about plugins, you can check Phaser documentation.

The Pathfinding plugin code is shown below. In the "init" method we initialize the EasyStar grid, which will be used during A*. EasyStar requires two things: a world grid, representing each tile with an identifier and an array of acceptable tiles, representing the tiles that are not obstacles. Notice that we are using the tile index from Phaser map to create the world grid, which is obtained from the Tiled JSON map.

```
1 var PathfindingExample = PathfindingExample || {};
2
3 PathfindingExample.Pathfinding = function (game, parent) {
4     "use strict";
5     Phaser.Plugin.call(this, game, parent);
6     this.easy_star = new EasyStar.js();
7 };
8
9 PathfindingExample.Pathfinding.prototype =
Object.create(Phaser.Plugin.prototype);
10 PathfindingExample.Pathfinding.prototype.constructor =
PathfindingExample.Pathfinding;
```

```
11
12 PathfindingExample.Pathfinding.prototype.init = function
(world_grid, acceptable_tiles, tile_dimensions) {
13     "use strict";
14     var grid_row, grid_column, grid_indices;
15     this.grid_dimensions = {row: world_grid.length, column:
world_grid[0].length};
16
17     grid_indices = [];
18     for (grid_row = 0; grid_row < world_grid.length; grid_row
+= 1) {
19         grid_indices[grid_row] = [];
20         for (grid_column = 0; grid_column <
world_grid[grid_row].length; grid_column += 1) {
21             grid_indices[grid_row][grid_column] =
world_grid[grid_row][grid_column].index;
22         }
23     }
24
25     this.easy_star.setGrid(grid_indices);
26     this.easy_star.setAcceptableTiles(acceptable_tiles);
27
28     this.tile_dimensions = tile_dimensions;
29 };
30
31 PathfindingExample.Pathfinding.prototype.find_path = function
(origin, target, callback, context) {
32     "use strict";
33     var origin_coord, target_coord;
34
35     origin_coord = this.get_coord_from_point(origin);
36     target_coord = this.get_coord_from_point(target);
37
38     if (!this.outside_grid(origin_coord) &&
!this.outside_grid(target_coord)) {
39         this.easy_star.findPath(origin_coord.column,
origin_coord.row, target_coord.column, target_coord.row,
this.call_callback_function.bind(this, callback, context));
40         this.easy_star.calculate();
41         return true;
42     } else {
43         return false;
44     }
45 };
46
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
47 PathfindingExample.Pathfinding.prototype.call_
callback_function = function  (callback, context, path) {
48    "use strict";
49    var path_positions;
50    path_positions = [];
51    if (path !== null) {
52        path.forEach(function (path_coord) {
53            path_positions.push(this.get_point_from_coord({row
: path_coord.y, column: path_coord.x}));
54        }, this);
55    }
56    callback.call(context, path_positions);
57 };
58
59 PathfindingExample.Pathfinding.prototype.outside_grid =
function (coord) {
60    "use strict";
61    return coord.row < 0 || coord.row >
this.grid_dimensions.row - 1 || coord.column < 0 || coord.column
> this.grid_dimensions.column - 1;
62 };
63
64 PathfindingExample.Pathfinding.prototype.get_coord_from_point
= function (point) {
65    "use strict";
66    var row, column;
67    row = Math.floor(point.y / this.tile_dimensions.y);
68    column = Math.floor(point.x / this.tile_dimensions.x);
69    return {row: row, column: column};
70 };
71
72 PathfindingExample.Pathfinding.prototype.get_point_from_coord
= function (coord) {
73    "use strict";
74    var x, y;
75    x = (coord.column * this.tile_dimensions.x) +
(this.tile_dimensions.x / 2);
76    y = (coord.row * this.tile_dimensions.y) +
(this.tile_dimensions.y / 2);
77    return new Phaser.Point(x, y);
78 };
```

The "find_path" method is responsible for running EasyStar to find a path from "origin"
to "target". This is done by calling EasyStar "findPath" and "calculate" methods. When

the path is found, the "call_callback_function" method is executed, which will create an array with all the points in the path and call the appropriate callback function in the correct context.

The remaining methods are helpful to make it easier to use the plugin. For example, the "get_coord_from_point" and "get_point_from_coord" methods convert a grid coord to a point coordinate in the world, and vice versa.

Now, we have to add our plugin in WorldState. This can be done in WorldState "init" method, as shown below. Notice that we pass the map collision layer data as the world grid, and the only acceptable tile is -1. This value is used by Phaser to represent positions without tiles. In practice, that means that any position without an obstacle is acceptable.

```
1 // initialize pathfinding
2    tile_dimensions = new Phaser.Point(this.map.tileWidth,
this.map.tileHeight);
3    this.pathfinding =
this.game.plugins.add(PathfindingExample.Pathfinding,
this.map.layers[1].data, [-1], tile_dimensions);
```

The Player prefab

Now that we have our Pathfinding plugin, we are going to create our Player prefab, which will move to a target location using this plugin.

The Player prefab code is shown below. In the constructor, we initialize the physical body. Since the player sprite is bigger than the tileset we used (as you may have noticed when building the map in Tiled), we are going to reduce the size of it's physical body using the "setSize" method from the body. Also, we are going to change its anchor point to the center of its physical body, to properly handle movements and collisions.

```
1 var PathfindingExample = PathfindingExample || {};
2
3 PathfindingExample.Player = function (game_state, name,
position, properties) {
4    "use strict";
5    PathfindingExample.Prefab.call(this, game_state, name,
position, properties);
6
7    //this.anchor.setTo(0.5);
8    //this.scale.setTo(0.666, 0.5);
9
```

```
10     this.walking_speed = +properties.walking_speed;
11
12     this.game_state.game.physics.arcade.enable(this);
13     // change the size and position of the collision box
14     this.body.setSize(12, 12, 0, 4);
15     this.body.collideWorldBounds = true;
16
17     // set anchor point to be the center of the collision box
18     this.anchor.setTo(0.5, 0.75);
19
20     this.path = [];
21     this.path_step = -1;
22 };
23
24 PathfindingExample.Player.prototype =
Object.create(PathfindingExample.Prefab.prototype);
25 PathfindingExample.Player.prototype.constructor =
PathfindingExample.Player;
26
27 PathfindingExample.Player.prototype.update = function () {
28     "use strict";
29     var next_position, velocity;
30     this.game_state.game.physics.arcade.collide(this,
this.game_state.layers.collision);
31
32     if (this.path.length > 0) {
33         next_position = this.path[this.path_step];
34
35         if (!this.reached_target_position(next_position)) {
36             velocity = new Phaser.Point(next_position.x -
this.position.x,
37                                         next_position.y -
this.position.y);
38             velocity.normalize();
39             this.body.velocity.x = velocity.x *
this.walking_speed;
40             this.body.velocity.y = velocity.y *
this.walking_speed;
41         } else {
42             this.position.x = next_position.x;
43             this.position.y = next_position.y;
44             if (this.path_step < this.path.length - 1) {
45                 this.path_step += 1;
46             } else {
47                 this.path = [];
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
48                      this.path_step = -1;
49                      this.body.velocity.x = 0;
50                      this.body.velocity.y = 0;
51                  }
52              }
53          }
54  };
55
56  PathfindingExample.Player.prototype.reached_target_position =
function (target_position) {
57      "use strict";
58      var distance;
59      distance = Phaser.Point.distance(this.position,
target_position);
60      return distance < 1;
61  };
62
63  PathfindingExample.Player.prototype.move_to = function
(target_position) {
64      "use strict";
65      this.game_state.pathfinding.find_path(this.position,
target_position, this.move_through_path, this);
66  };
67
68  PathfindingExample.Player.prototype.move_through_path =
function (path) {
69      "use strict";
70      if (path !== null) {
71          this.path = path;
72          this.path_step = 0;
73      } else {
74          this.path = [];
75      }
76  };
```

The "move_to" method will receive a target position and call our Pathfinding plugin to find the path from the current position to the target. When the path is found, the "move_through_path" method is called, which will save the found path and reset the "path_step" variable. The actual movement is done in the "update" method. It moves the player towards the position indicated by the "path_step" variable. If the player has reached that position, it increments "path_step" until all the path is completed. To check if the player has reached a given position, we must consider an error margin as done in the "reached_target_position" method, since the player will not be exactly in the desired position, but close to it.

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Finally, we must add in WorldState the code to move the player. We do this by adding an input event in the "create" method as shown below. This input event will call the "move_player" method (also shown below), which will move the player to the position clicked by the mouse. There is one last modification I did in the WorldState, which is showing the player physical body in the "render" method. This method is automatically called by Phaser after updating all objects, and we will use it to show the player body. This is useful to check if the movement and collisions are properly working.

```
1 // add user input to move player
2     this.game.input.onDown.add(this.move_player, this);
```

```
1 PathfindingExample.WorldState.prototype.move_player = function
() {
2     "use strict";
3     var target_position;
4     target_position = new
Phaser.Point(this.game.input.activePointer.x,
this.game.input.activePointer.y);
5
6     this.prefabs.player.move_to(target_position);
7 };
```
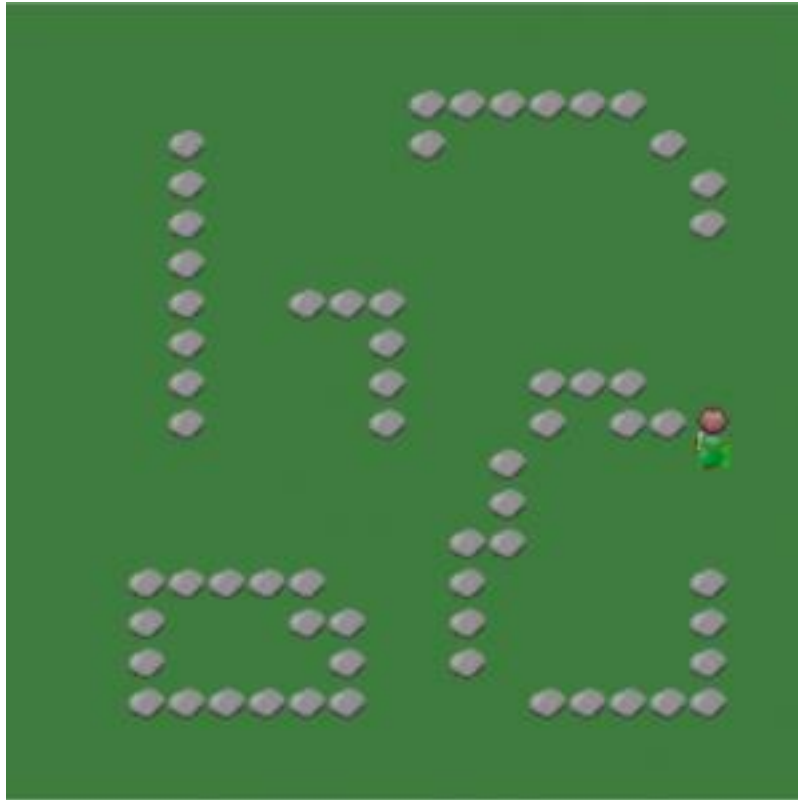
And now, our pathfinding demo is complete. Try moving the player to different positions and check if it is correctly avoiding obstacles.

How to Use State Machines to Control Behavior and Animations in Phaser
By Renan Oliveira

Suppose you're building a platformer game, where the hero can walk, jump and attack. The hero can jump while standing or walking, but he can't attack while jumping or walking. Also, the player can not jumping again while it is in the air (no double jumping).

We can start implementing the hero code as below. Depending on your coding experience, you may have noticed that this code is troublesome. To know if the player is jumping (and therefore, keeping it from attacking), we have to save its state in a variable, which says if the hero is jumping or not. Now, suppose we want the player to be able to block attacks, but only when he is standing without walking, jumping or attacking. We would have to add another variable to keep track of that. In addition, suppose we want to change the hero animation when it is jumping, blocking or attacking. You may already have noticed that it will be extremely difficult to manage this code as we increase the size of our game.

```
1 Hero = function (game_state, walking_speed, jumping_speed) {
2     Phaser.Sprite.call(this);
3     this.game_state = game_state;
4
5     this.walking_speed = walking_speed;
6     this.jumping_speed = jumping_speed;
7     this.is_jumping = false;
8
9     this.game_state.physics.arcade.enable(this);
10
11    this.cursors =
this.game_state.input.keyboard.createCursorKeys();
12 };
13
14 Hero.prototype = Object.create(Phaser.Sprite.prototype);
15 Hero.prototype.constructor = Hero;
16
17 Hero.prototype.update = function () {
18    if (this.cursors.left.isDown) {
19        this.body.velocity.x = -this.walking_speed;
20    } else if (this.cursors.right.isDown) {
21        this.body.velocity.x = this.walking_speed;
22    } else {
23        this.body.velocity.x = 0;
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Page 370

```
24      }
25
26      if (this.cursors.up.isDown && !this.is_jumping) {
27          this.body.velocity.y = -this.jumping_speed;
28          this.is_jumping = true;
29      }
30
31      if
(this.game_state.input.keyboard.isDown(Phaser.Keyboard.SPACEBAR)
&& !this.is_jumping) {
32          this.attack();
33      }
34  };
```

To handle such problem, there is a structure called state machine, which can efficiently model what we want in our game: an object that may assume different states during its life. In this tutorial, I will explain how to use a state machine to manage an object behavior and animations in your games.

First, I will explain the basics of state machines, for those not familiar with this concept. Then, I will show a possible code implementation, which will be used in a platformer demo.

To read this tutorial, it is important that you're familiar with the following concepts:

- Javascript and object-oriented concepts.
- Basic Phaser concepts, such as: states, sprites, groups and arcade physics
- Creating maps using Tiled

Source code files

You can download the tutorial source code files here.

What is a state machine?

A state machine is a mathematical model used to represent states and transitions between those states. Besides the fancy name, state machines are simple things, and I will explain them using our hero example. The figure below shows the state machine that controls our hero. In this figure, the circles are the possible states that our hero can be, while the arrows represent transitions between them. The text over the arrows are the necessary input (in our case actions) to execute that state transition. And that is it, the hero must

start in an initial state (for example, standing), and constantly verifies the input to execute any necessary transitions (such as walk and jump).
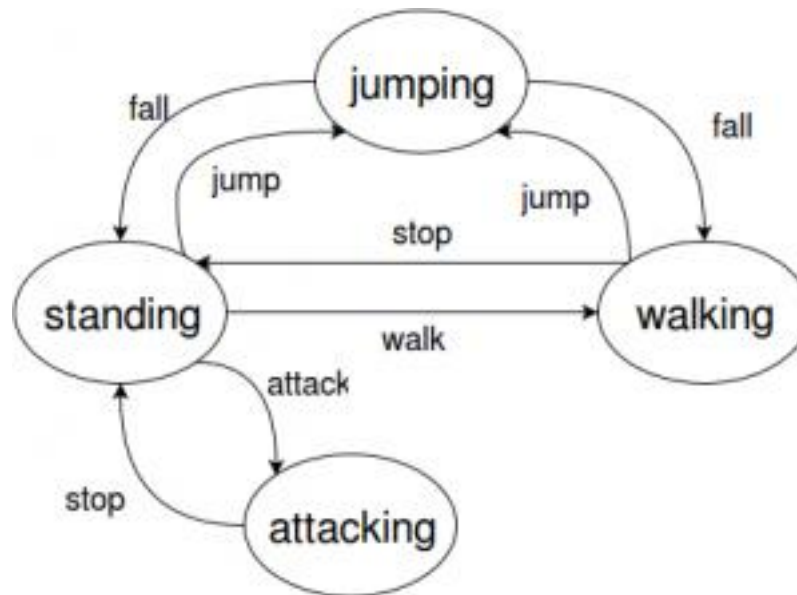
For example, if our hero is standing, and it receives a walk input, it should change to the walking state. On the other hand, if it is in the standing or walking states and receives a jump input, it must go to the jumping state. You may have noticed that the variables we used in the previous code to keep track of what the hero was doing (such as jumping or standing) were playing the role of the states. By using a state machine we can encapsulate all the hero behavior in their respective states, keeping the code cleaner and easier to manage.

The state machine code

Now that you know what is a state machine, it is time to write the code for ours. Remember that this is my suggestion of code, and you can implement yours the way you think it better fits your game.

Let's start by writing the code for the StateMachine class, which is shown below. A state machine has a set of states, which can be added by the "add_state" method. Each state is identified by its name (for example: standing, walking, jumping and attacking), and we can set the initial state by the "set_initial_state" method. The "handle_input" method must be called every time a new input is available (represented by the "command" variable). The current state will handle this input and will return the next state. If the next

state is different from the current one, we must exit the current state, and enter the new one.

```
1 var StateMachineExample = StateMachineExample || {};
2
3 StateMachineExample.StateMachine = function () {
4     "use strict";
5     this.states = {};
6 };
7
8 StateMachineExample.StateMachine.prototype.add_state =
function (state_name, state) {
9     "use strict";
10    this.states[state_name] = state;
11 };
12
13 StateMachineExample.StateMachine.prototype.set_initial_state
= function (state_name) {
14    "use strict";
15    this.current_state = this.states[state_name];
16    this.current_state.enter();
17 };
18
19 StateMachineExample.StateMachine.prototype.handle_input =
function (command) {
20    "use strict";
21    var next_state;
22    next_state = this.current_state.handle_input(command);
23    if (next_state && next_state !== this.current_state.name)
{
24        this.current_state.exit();
25        this.current_state = this.states[next_state];
26        this.current_state.enter();
27    }
28 };
```

The State class code is shown below. By default, any state contains its name and does nothing in the "enter", "exit" and "handle_input" methods. This is just the base class which will be extended by the states of our hero.

```
1 var StateMachineExample = StateMachineExample || {};
2
3 StateMachineExample.State = function (name, prefab) {
4     "use strict";
```

```
 5      this.name = name;
 6      this.prefab = prefab;
 7 };
 8
 9 StateMachineExample.State.prototype.enter = function () {
10      "use strict";
11 };
12
13 StateMachineExample.State.prototype.exit = function () {
14      "use strict";
15 };
16
17 StateMachineExample.State.prototype.handle_input = function
(command) {
18      "use strict";
19      return this.name;
20 };
```

The Command class is shown below. It will simply be an object with a name to identify it
and a set of properties. Those properties can be used to properly handle transitions. For
example, our "walk" command can have the direction to where the hero is walking, so we
can properly update its velocity.

```
 1 var StateMachineExample = StateMachineExample || {};
 2
 3 StateMachineExample.Command = function (name, properties) {
 4      "use strict";
 5      var property;
 6      this.name = name;
 7      for (property in properties) {
 8          if (properties.hasOwnProperty(property)) {
 9              this[property] = properties[property];
10          }
11      }
12 };
```

Phaser states of our demo

We will save the level data of our demo in a JSON file, which will be read when it starts.
The JSON file I'm going to use is shown below. Notice that we must define the assets,
groups and map information.

```
 1 {
```

```
2     "assets": {
3         "map_tileset": {"type": "image", "source":
"assets/images/tiles_spritesheet.png"},
4         "hero_spritesheet": { "type": "spritesheet", "source":
"assets/images/player_spritesheet.png", "frame_width": 28,
"frame_height": 30, "frames": 5, "margin": 1, "spacing": 1 },
5
6         "level_tilemap": {"type": "tilemap", "source":
"assets/maps/demo_map.json"}
7     },
8     "groups": [
9         "heroes"
10    ],
11    "map": {
12        "key": "level_tilemap",
13        "tilesets": ["map_tileset"]
14    }
15 }
```

We're going to use a map created using the Tiled level editor. If you're not familiar with Tiled, you can check one of my previous tutorials, where I cover it with more details. This is the map I'm going to use. You can use this one, provided in the source code or create your own. However, if you create your own map, you must be careful of two things: 1) you must set a property named "collision" to be true to any layers that are collidable; 2) you must define the hero object properties as shown below.
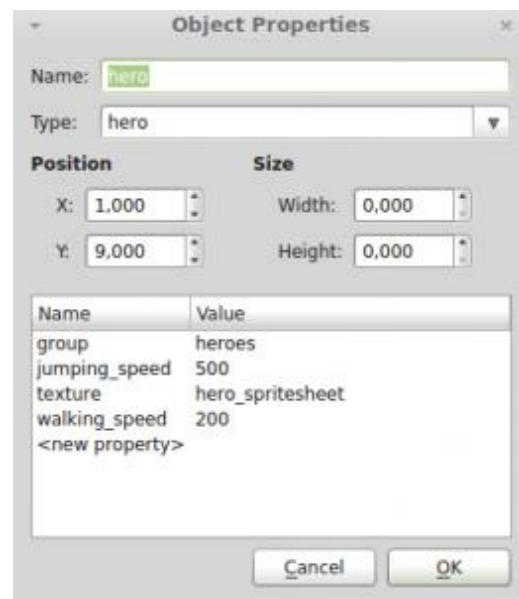
Our demo will have three states: BootState, LoadingState and DemoState. The code for BootState and LoadingState is shown below. Both are responsible for reading the JSON level file and loading all assets, before calling DemoState.

```
1 var StateMachineExample = StateMachineExample || {};
2
3 StateMachineExample.BootState = function () {
4     "use strict";
5     Phaser.State.call(this);
6 };
7
8 StateMachineExample.BootState.prototype =
Object.create(Phaser.State.prototype);
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
9 StateMachineExample.BootState.prototype.constructor =
StateMachineExample.BootState;
10
11 StateMachineExample.BootState.prototype.init = function
(level_file, next_state) {
12     "use strict";
13     this.level_file = level_file;
14     this.next_state = next_state;
15 };
16
17 StateMachineExample.BootState.prototype.preload = function ()
{
18     "use strict";
19     this.load.text("level1", this.level_file);
20 };
21
22 StateMachineExample.BootState.prototype.create = function ()
{
23     "use strict";
24     var level_text, level_data;
25     level_text = this.game.cache.getText("level1");
26     level_data = JSON.parse(level_text);
27     this.game.state.start("LoadingState", true, false,
level_data, this.next_state);
28 };


1 var StateMachineExample = StateMachineExample || {};
2
3 StateMachineExample.LoadingState = function () {
4     "use strict";
5     Phaser.State.call(this);
6 };
7
8 StateMachineExample.LoadingState.prototype =
Object.create(Phaser.State.prototype);
9 StateMachineExample.LoadingState.prototype.constructor =
StateMachineExample.LoadingState;
10
11 StateMachineExample.LoadingState.prototype.init = function
(level_data, next_state) {
12     "use strict";
13     this.level_data = level_data;
14     this.next_state = next_state;
15 };
16
```

```
17 StateMachineExample.LoadingState.prototype.preload = function
() {
18     "use strict";
19     var assets, asset_loader, asset_key, asset;
20     assets = this.level_data.assets;
21     for (asset_key in assets) { // load assets according to
asset key
22         if (assets.hasOwnProperty(asset_key)) {
23             asset = assets[asset_key];
24             switch (asset.type) {
25             case "image":
26                 this.load.image(asset_key, asset.source);
27                 break;
28             case "spritesheet":
29                 this.load.spritesheet(asset_key, asset.source,
asset.frame_width, asset.frame_height, asset.frames,
asset.margin, asset.spacing);
30                 break;
31             case "tilemap":
32                 this.load.tilemap(asset_key, asset.source,
null, Phaser.Tilemap.TILED_JSON);
33                 break;
34             }
35         }
36     }
37 };
38
39 StateMachineExample.LoadingState.prototype.create = function
() {
40     "use strict";
41     this.game.state.start(this.next_state, true, false,
this.level_data);
42 };
```

DemoState must load the map with all its prefabs. To do that, first it will initialize the map in the "init" method. Then, in the "create" method it will create all groups, map layers and prefabs. Notice that we use the "collision" property in the map layers to check if they are collidable. The "create_object" is used to create each prefab according to its type. The types are stored in the "prefab_classes" property, which is used to call the correct constructor. Notice that this is possible because all prefabs have the same constructor, as shown in the Prefab base class below.

```
1 var StateMachineExample = StateMachineExample || {};
2
```

```
3 StateMachineExample.DemoState = function () {
4     "use strict";
5     Phaser.State.call(this);
6
7     this.prefab_classes = {
8         "hero": StateMachineExample.Hero.prototype.constructor
9     };
10 };
11
12 StateMachineExample.DemoState.prototype =
Object.create(Phaser.State.prototype);
13 StateMachineExample.DemoState.prototype.constructor =
StateMachineExample.DemoState;
14
15 StateMachineExample.DemoState.prototype.init = function
(level_data) {
16     "use strict";
17     var tileset_index, tile_dimensions;
18     this.level_data = this.level_data || level_data;
19
20     this.scale.scaleMode = Phaser.ScaleManager.SHOW_ALL;
21     this.scale.pageAlignHorizontally = true;
22     this.scale.pageAlignVertically = true;
23
24     // start physics system
25     this.game.physics.startSystem(Phaser.Physics.ARCADE);
26     this.game.physics.arcade.gravity.y = 1000;
27
28     // create map and set tileset
29     this.map = this.game.add.tilemap(this.level_data.map.key);
30     tileset_index = 0;
31     this.map.tilesets.forEach(function (tileset) {
32         this.map.addTilesetImage(tileset.name,
this.level_data.map.tilesets[tileset_index]);
33         tileset_index += 1;
34     }, this);
35 };
36
37 StateMachineExample.DemoState.prototype.create = function ()
{
38     "use strict";
39     var group_name, object_layer, collision_tiles;
40
41     // create map layers
42     this.layers = {};
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
43     this.map.layers.forEach(function (layer) {
44         this.layers[layer.name] =
this.map.createLayer(layer.name);
45         if (layer.properties.collision) { // collision layer
46             collision_tiles = [];
47             layer.data.forEach(function (data_row) { // find
tiles used in the layer
48                 data_row.forEach(function (tile) {
49                     // check if it's a valid tile index and
isn't already in the list
50                     if (tile.index > 0 &&
collision_tiles.indexOf(tile.index) === -1) {
51                         collision_tiles.push(tile.index);
52                     }
53                 }, this);
54             }, this);
55             this.map.setCollision(collision_tiles, true,
layer.name);
56         }
57     }, this);
58     // resize the world to be the size of the current layer
59     this.layers[this.map.layer.name].resizeWorld();
60
61     // create groups
62     this.groups = {};
63     this.level_data.groups.forEach(function (group_name) {
64         this.groups[group_name] = this.game.add.group();
65     }, this);
66
67     this.prefabs = {};
68
69     for (object_layer in this.map.objects) {
70         if (this.map.objects.hasOwnProperty(object_layer)) {
71             // create layer objects
72             this.map.objects[object_layer].forEach(this.create
_object, this);
73         }
74     }
75 };
76
77 StateMachineExample.DemoState.prototype.create_object =
function (object) {
78     "use strict";
79     var object_y, position, prefab;
80     // tiled coordinates starts in the bottom left corner
```

```
81     object_y = (object.gid) ? object.y - (this.map.tileHeight
/ 2) : object.y + (object.height / 2);
82     position = {"x": object.x + (this.map.tileHeight / 2),
"y": object_y};
83     // create object according to its type
84     if (this.prefab_classes.hasOwnProperty(object.type)) {
85         prefab = new this.prefab_classes[object.type](this,
object.name, position, object.properties);
86     }
87     this.prefabs[object.name] = prefab;
88 };

1 var StateMachineExample = StateMachineExample || {};
2
3 StateMachineExample.Prefab = function (game_state, name,
position, properties) {
4     "use strict";
5     Phaser.Sprite.call(this, game_state.game, position.x,
position.y, properties.texture);
6
7     this.game_state = game_state;
8
9     this.name = name;
10
11     this.game_state.groups[properties.group].add(this);
12     this.frame = +properties.frame;
13
14     if (properties.scale) {
15         this.scale.setTo(properties.scale.x,
properties.scale.y);
16     }
17
18     this.game_state.prefabs[name] = this;
19 };
20
21 StateMachineExample.Prefab.prototype =
Object.create(Phaser.Sprite.prototype);
22 StateMachineExample.Prefab.prototype.constructor =
StateMachineExample.Prefab;
```

Hero states

Now that we have our state machine implemented, we're going to create the states of our hero. In this tutorial I'll show the standing, walking and jumping states. I'll leave the

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

attacking state (and any other you may think of) as an exercise, since it's similar to what we're doing for the other states. All states will extend the State base class, and will implement the necessary methods.

For example, the code below shows the StandingState. In its enter method it must set the hero frame to the standing frame and the velocity to 0. In the "handle_input" method it checks for the "walk" and "jump" commands. Notice that the "walk" command has a "direction" property so we can know if the player is moving left or right.

```
1 var StateMachineExample = StateMachineExample || {};
2
3 StateMachineExample.StandingState = function (name, prefab,
frame) {
4     "use strict";
5     StateMachineExample.State.call(this, name, prefab);
6     this.frame = frame;
7 };
8
9 StateMachineExample.StandingState.prototype =
Object.create(StateMachineExample.State.prototype);
10 StateMachineExample.StandingState.prototype.constructor =
StateMachineExample.StandingState;
11
12 StateMachineExample.StandingState.prototype.enter = function
() {
13     "use strict";
14     // set standing frame and velocity to 0
15     this.prefab.frame = this.frame;
16     this.prefab.body.velocity.x = 0;
17 };
18
19 StateMachineExample.StandingState.prototype.handle_input =
function (command) {
20     "use strict";
21     switch (command.name) {
22     case "walk":
23         if (command.direction === "left") {
24             return "walking_left";
25         } else {
26             return "walking_right";
27         }
28     case "jump":
29         return "jumping";
30     }
```

```
31     StateMachineExample.State.prototype.handle_input.call(this
, command);
32 };
```

The WalkingState contains the walking animation and walking speed, as shown below. In its "enter" method it will play the walking animation and set the hero velocity. In the "exit" method it will only stop the animation. We're not going to set the velocity to 0 in the "exit" method so the player can keep moving when it goes from the walking state to the jumping state. Finally, in the "handle_input" method it checks for the "stop" and "jump" commands.

```javascript
1  var StateMachineExample = StateMachineExample || {};
2
3  StateMachineExample.WalkingState = function (name, prefab,
direction, walking_speed) {
4      "use strict";
5      StateMachineExample.State.call(this, name, prefab);
6
7      this.walking_animation =
this.prefab.animations.add("walking", [0, 1, 2, 1], 6, true);
8
9      this.direction = direction;
10      this.walking_speed = walking_speed;
11  };
12
13  StateMachineExample.WalkingState.prototype =
Object.create(StateMachineExample.State.prototype);
14  StateMachineExample.WalkingState.prototype.constructor =
StateMachineExample.WalkingState;
15
16  StateMachineExample.WalkingState.prototype.enter = function
() {
17      "use strict";
18      // start animation and set velocity
19      this.walking_animation.play();
20      this.prefab.body.velocity.x = this.direction *
this.walking_speed;
21
22      if (this.direction === 1) {
23          this.prefab.scale.setTo(-1, 1);
24      } else {
25          this.prefab.scale.setTo(1, 1);
26      }
27  };
28
29  StateMachineExample.WalkingState.prototype.exit = function ()
{
30      "use strict";
31      // stop animation and set velocity to zero
32      this.walking_animation.stop();
33  };
34
35  StateMachineExample.WalkingState.prototype.handle_input =
function (command) {
36      "use strict";
37      switch (command.name) {
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
38      case "stop":
39          return "standing";
40      case "jump":
41          return "jumping";
42      }
43      StateMachineExample.State.prototype.handle_input.call(this
, command);
44 };
```

Finally, JumpingState has the jumping speed, which is applied to the velocity in its "enter" method. The only command it checks in the "handle_input" method is the "fall" command.

```
1 var StateMachineExample = StateMachineExample || {};
2
3 StateMachineExample.JumpingState = function (name, prefab,
jumping_speed) {
4      "use strict";
5      StateMachineExample.State.call(this, name, prefab);
6      this.jumping_speed = jumping_speed;
7 };
8
9 StateMachineExample.JumpingState.prototype =
Object.create(StateMachineExample.State.prototype);
10 StateMachineExample.JumpingState.prototype.constructor =
StateMachineExample.JumpingState;
11
12 StateMachineExample.JumpingState.prototype.enter = function
() {
13      "use strict";
14      // set vertical velocity
15      this.prefab.body.velocity.y = -this.jumping_speed;
16 };
17
18 StateMachineExample.JumpingState.prototype.handle_input =
function (command) {
19      "use strict";
20      switch (command.name) {
21      case "fall":
22          return "standing";
23      }
24      StateMachineExample.State.prototype.handle_input.call(this
, command);
25 };
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

The hero prefab

Now that we have the hero states, we can create its prefab as shown below. In the constructor, we create the state machine, adding all its states and setting the initial state. We also add callbacks to the keyboard events "onDown" and "onUp". These callbacks will be used to check user input and send commands to the state machine.

```
1 var StateMachineExample = StateMachineExample || {};
2
3 StateMachineExample.Hero = function (game_state, name,
position, properties) {
4     "use strict";
5     StateMachineExample.Prefab.call(this, game_state, name,
position, properties);
6
7     this.anchor.setTo(0.5);
8
9     this.walking_speed = +properties.walking_speed;
10    this.jumping_speed = +properties.jumping_speed;
11
12    this.game_state.game.physics.arcade.enable(this);
13    this.body.collideWorldBounds = true;
14
15    // create state machine and add states
16    this.state_machine = new
StateMachineExample.StateMachine();
17    this.state_machine.add_state("standing", new
StateMachineExample.StandingState("standing", this, 3));
18    this.state_machine.add_state("walking_left", new
StateMachineExample.WalkingState("walking_left", this, -1,
this.walking_speed));
19    this.state_machine.add_state("walking_right", new
StateMachineExample.WalkingState("walking_left", this, 1,
this.walking_speed));
20    this.state_machine.add_state("jumping", new
StateMachineExample.JumpingState("jumping", this,
this.jumping_speed));
21    this.state_machine.set_initial_state("standing");
22
23    // add callbacks to keyboard events
24    this.game_state.game.input.keyboard.addCallbacks(this,
this.process_on_down_input, this.process_on_up_input, null);
25 };
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
26
27 StateMachineExample.Hero.prototype =
Object.create(StateMachineExample.Prefab.prototype);
28 StateMachineExample.Hero.prototype.constructor =
StateMachineExample.Hero;
29
30 StateMachineExample.Hero.prototype.update = function () {
31     "use strict";
32     this.game_state.game.physics.arcade.collide(this,
this.game_state.layers.collision);
33
34     // touching ground tile
35     if (this.body.blocked.down) {
36         this.state_machine.handle_input(new
StateMachineExample.Command("fall", {}));
37     }
38 };
39
40 StateMachineExample.Hero.prototype.process_on_down_input =
function (event) {
41     "use strict";
42     switch (event.keyCode) {
43     case Phaser.Keyboard.LEFT:
44         // walk left
45         this.state_machine.handle_input(new
StateMachineExample.Command("walk", {direction: "left"}));
46         break;
47     case Phaser.Keyboard.RIGHT:
48         // walk right
49         this.state_machine.handle_input(new
StateMachineExample.Command("walk", {direction: "right"}));
50         break;
51     case Phaser.Keyboard.UP:
52         // jump
53         this.state_machine.handle_input(new
StateMachineExample.Command("jump", {}));
54         break;
55     }
56 };
57
58 StateMachineExample.Hero.prototype.process_on_up_input =
function (event) {
59     "use strict";
60     switch (event.keyCode) {
61     case Phaser.Keyboard.LEFT:
```

```
62          this.state_machine.handle_input(new
StateMachineExample.Command("stop", {}));
63          break;
64      case Phaser.Keyboard.RIGHT:
65          this.state_machine.handle_input(new
StateMachineExample.Command("stop", {}));
66          break;
67      }
68 };
```

The "process_on_down_input" and "process_on_up_input" show those callback functions. We use the keyCode to identify the user input and create the new command accordingly. Notice that when we issue the "walk" command we must specify the direction.

Finally, in the "update" method we check if the player is touching a ground tile and issue the "fall" command accordingly.

Now we can play our demo and move our hero using the state machine!

Possible extensions

Even though our demo is complete, there are several extensions you can make. First, you can improve our state machine to allow the hero to change direction while jumping or even double jump. You can also issue the commands in the "update" method by checking the keys that are pressed, to see the difference.

Try adding new states, like an invincible state when the hero gets a powerup item. Also, suppose that you want to create an item that changes the hero attack. You don't want to check if the hero has collected this item every time in the attack state. So, you can create another state machine only to handle this new attack when it is available, and issue commands simultaneously for the two state machines. Therefore, there are endless possibilities.

Finally, the main limitation of state machines is that sometimes they are too simple, and may not be suitable for complex AI. In this case, you can try different models, such as pushdown automatas and behavior trees.

And that concludes our state machines tutorial.

How to Procedurally Generate a Dungeon in Phaser – Part 1
By Renan Oliveira

Some games have a fixed number of levels created by a level designer. This way, the designer can create the levels so as to provide the desired gameplay experience to the player. However, it reduces the replay value of the game, since the levels will always be the same every time they are played.

Another alternative is to procedurally generate levels using an algorithm instead of a level designer. This results in a virtually infinite number of levels, and the player will never play the same level twice. However, since the game designers have no full control on the levels, they probably won't be so good as if they were generated by a experienced level designer.

Each kind of game level (manual or procedural) has advantages and disadvantages, and can be applied in different kinds of games (both strategies could even be used in the same game). In this tutorial, we will procedurally create a dungeon with multiple rooms, using a simple algorithm. Notice that there are several ways of doing so (as you can check here), and each one can be recommended for a specific kind of game. So, if you're building a game and need procedurally generated content (not just levels), take a look in different approaches to see which one better fits your game.

First, I will explain the algorithm we are going to use to generate our dungeon. Then, we will build a demo that creates a dungeon with multiple rooms and load them using Tiled maps.

To read this tutorial, it is important that you're familiar with the following concepts:

- Javascript and object-oriented concepts.
- Basic Phaser concepts, such as: states, sprites, groups and arcade physics
- Creating maps using Tiled

Source code files

You can download the tutorial source code files here.

Creating the Tiled maps

We will start by creating room templates using Tiled for each possible room in our game. Then, our game will load the correct room map according to the current room

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Page 390

configuration. For example, if the player is currently in a room with a single door in the north direction, it should load the "room_N.json" map.

The figure below shows an example of room created using Tiled. If you're not familiar with Tiled, you can check one of my previous tutorials, where I cover it with more details. Even if you're familiar with it, I highly suggest using the maps provided by the source code, since it is necessary to create one for each room configuration (resulting in a total of 15 maps). If you still wish to create your own maps, the only required things to follow is that you must create a layer called collision with a collision property set to true,

and you must set the object properties as shown below, as this will be required for our

demo.
The Dungeon Generation Algorithm

We're going to procedurally generate a dungeon with multiple rooms in a grid structure. We will do that by creating a given number of rooms by traveling a grid, and then connecting the neighbor rooms.
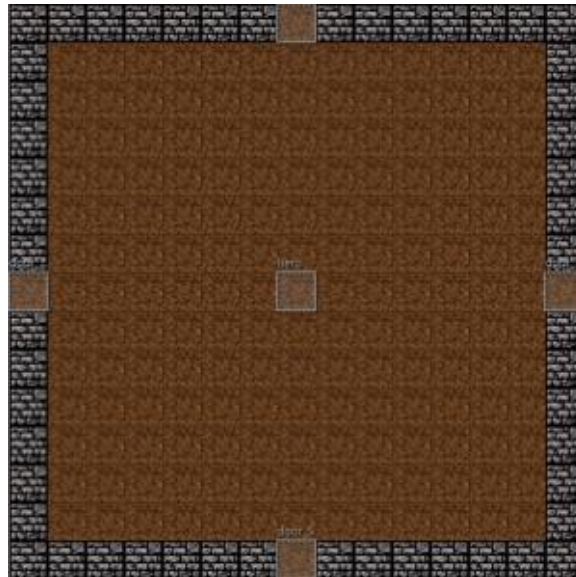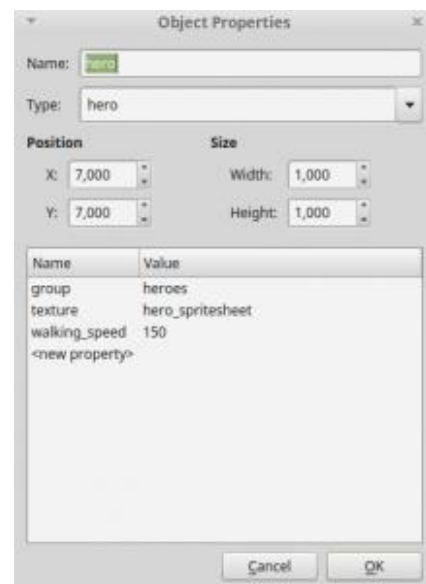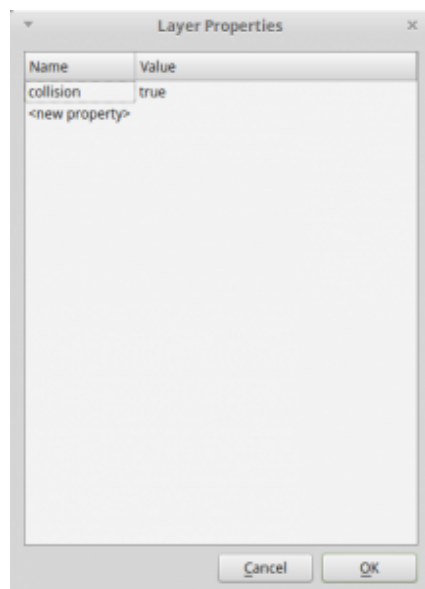

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Given a grid and a desired number of rooms "n", we generate the dungeon with the following steps:

1. Add the coordinate of the middle of the grid in a list called "rooms_to_create"
2. While the number of created rooms is less than "n", repeat:
    1. Get the first coordinate in "rooms_to_create"
    2. Create a room with this coordinate
    3. Add this room in a "created_rooms" list
    4. Add a random number of neighbor coordinates of the current room to "rooms_to_create"
3. After all rooms are created, connect all neighbor rooms

Notice that, for this algorithm to work we must guarantee that each iteration of the while loop in step two adds at least one new room to "rooms_to_create", so the algorithm may continue until all rooms are created. We guarantee that by making the grid big enough so the dungeon can always be expanded to any direction (this is possible if the width and height of the grid is twice the number of rooms, for example) and by forcing the last step of the while loop to always add at least one neighbor coordinate to "rooms_to_create".

The code below shows the implementation of this algorithm. The algorithm is implemented in the "generate_dungeon" method. First, it initialize the grid with its dimensions equals twice the number of rooms, and add the middle coordinate to the "rooms_to_create" list. Then, the while loop repeats the process described by the algorithm, creating a new room and adding a random number of neighbors to "rooms_to_create". After creating all rooms, it iterates through all of them connecting them.

```
1  var ProceduralGeneration = ProceduralGeneration || {};
2
3  ProceduralGeneration.Dungeon = function (game_state) {
4      "use strict";
5      this.TILE_DIMENSIONS = new Phaser.Point(32, 32);
6
7      this.game_state = game_state;
8  };
9
10 ProceduralGeneration.Dungeon.prototype.generate_dungeon =
function (number_of_rooms) {
11     "use strict";
12     var grid_size, rooms_to_creates, current_room_coordinate,
current_room, created_rooms, initial_room_coordinate,
```

```
final_room_coordinate, max_distance_to_initial_room,
distance_to_initial_room;
13    // initialize the grid
14    grid_size = 2 * number_of_rooms;
15    this.initialize_grid(grid_size);
16
17    // add the middle coordinate as initial
18    initial_room_coordinate = new Phaser.Point((grid_size / 2)
- 1, (grid_size / 2) - 1);
19    rooms_to_creates = [];
20    rooms_to_creates.push({row: initial_room_coordinate.y,
column: initial_room_coordinate.x});
21    created_rooms = [];
22    // iterate creating rooms
23    while (rooms_to_creates.length > 0 && created_rooms.length
< number_of_rooms) {
24        current_room_coordinate = rooms_to_creates.shift();
25        // create room with current coordinate
26        current_room = new
ProceduralGeneration.Room(this.game_state,
current_room_coordinate, this.TILE_DIMENSIONS);
27        this.grid[current_room_coordinate.row][current_room_co
ordinate.column] = current_room;
28        created_rooms.push(current_room);
29        // add random number of neighbors to rooms_to_create
30        this.check_for_neighbors(current_room,
rooms_to_creates);
31    }
32
33    // iterate through rooms to connect them
34    created_rooms.forEach(function (room) {
35        room.neighbor_coordinates().forEach(function
(coordinate) {
36            if (this.grid[coordinate.row][coordinate.column])
{
37                room.connect(coordinate.direction,
this.grid[coordinate.row][coordinate.column]);
38            }
39        }, this);
40    }, this);
41
42    return
this.grid[initial_room_coordinate.y][initial_room_coordinate.x];
43 };
44
```

```
45 ProceduralGeneration.Dungeon.prototype.print_grid = function
() {
46     "use strict";
47     var row_index, column_index, row;
48     for (row_index = 0; row_index < this.grid.length;
row_index += 1) {
49         row = "";
50         for (column_index = 0; column_index <
this.grid[row_index].length; column_index += 1) {
51             if (this.grid[row_index][column_index]) {
52                 row += "R";
53             } else {
54                 row += "X";
55             }
56         }
57         console.log(row);
58     }
59 };
60
61 ProceduralGeneration.Dungeon.prototype.initialize_grid =
function (grid_size) {
62     "use strict";
63     var row_index, column_index;
64     this.grid = [];
65     // initialize all rooms as null
66     for (row_index = 0; row_index < grid_size; row_index += 1)
{
67         this.grid.push([]);
68         for (column_index = 0; column_index < grid_size;
column_index += 1) {
69             this.grid[row_index].push(null);
70         }
71     }
72 };
73
74 ProceduralGeneration.Dungeon.prototype.check_for_neighbors =
function (room, rooms_to_creates) {
75     "use strict";
76     var coordinates_to_check, available_neighbors,
number_of_neighbors, neighbor_index, random_number, room_frac,
available_neighbor_index;
77     coordinates_to_check = room.neighbor_coordinates();
78     available_neighbors = [];
79     // find neighbor coordinates that are free
80     coordinates_to_check.forEach(function (coordinate) {
```

```
81          if (!this.grid[coordinate.row][coordinate.column]) {
82              available_neighbors.push(coordinate);
83          }
84      }, this);
85      // select random number of neighbors
86      number_of_neighbors = this.game_state.game.rnd.between(1,
available_neighbors.length - 1);
87
88      // select the neighbor coordinates
89      for (neighbor_index = 0; neighbor_index <
number_of_neighbors; neighbor_index += 1) {
90          random_number = this.game_state.game.rnd.frac();
91          room_frac = 1 / available_neighbors.length;
92          // assign a range to each neighbor and select the one
whose range contains room_frac
93          for (available_neighbor_index = 0;
available_neighbor_index < available_neighbors.length;
available_neighbor_index += 1) {
94              if (random_number < room_frac) {
95                  rooms_to_creates.push(available_neighbors[avai
lable_neighbor_index]);
96                  available_neighbors.splice(available_neighbor_
index, 1);
97                  break;
98              } else {
99                  room_frac += (1 / available_neighbors.length);
100             }
101         }
102     }
103 };
```

The method "check_for_neighbors" is responsible for adding the random neighbors to
"rooms_to_create". First, it finds how many neighbor coordinates are not occupied yet,
putting them in the "available_neighbors" list. Then, it selects the number of neighbors
that will be created randomly, using Phaser random data generator (you can learn more of
it, in Phaser documentation). Finally, for each neighbor coordinate to be created, it
randomly chooses one of the available neighbors. This is done by assigning a range to
each available neighbor and generating a random number between 0 and 1. The chosen
neighbor is the one whose range contains the generated number. For example, if there are
two available neighbors, the first one will be selected if the generated number is up to
0.5, otherwise the second neighbor is selected.

We still have to create our Room class, which is shown below. The room will have its coordinates and should be able to inform its neighbors and the name of its map file. The method "neighbor_coordinates" simply returns the neighbor coordinates in each direction. The method "connect" is used to define the neighbors that actually exist and the "template_name" method returns the name of the JSON map to be loaded for this room. This name always starts with "room_" and it's followed by the directions that there are rooms, in clockwise order. For example, if the room has doors in the north, south and west directions, its template name is "room_NSW.json".

```
1 var ProceduralGeneration = ProceduralGeneration || {};
2
3 ProceduralGeneration.Room = function (game_state, coordinate,
tile_dimensions) {
4     "use strict";
5     this.game_state = game_state;
6     this.coordinate = coordinate;
7     this.tile_dimensions = tile_dimensions;
8
9     this.neighbors = {};
10 };
11
12 ProceduralGeneration.Room.prototype.neighbor_coordinates =
function () {
13     "use strict";
14     var neighbor_coordinates;
15     neighbor_coordinates = [
16         {direction: "N", row: this.coordinate.row - 1, column:
this.coordinate.column},
17         {direction: "E", row: this.coordinate.row, column:
this.coordinate.column + 1},
18         {direction: "S", row: this.coordinate.row + 1, column:
this.coordinate.column},
19         {direction: "W", row: this.coordinate.row, column:
this.coordinate.column - 1}
20     ];
21     return neighbor_coordinates;
22 };
23
24 ProceduralGeneration.Room.prototype.connect = function
(direction, room) {
25     "use strict";
26     this.neighbors[direction] = room;
27 };
28
```

```
29 ProceduralGeneration.Room.prototype.template_name = function
() {
30     "use strict";
31     var template_name;
32     // the template name is room_ followed by the directions
with neighbors
33     template_name = "room_";
34     this.neighbor_coordinates().forEach(function (coordinate)
{
35         if (this.neighbors[coordinate.direction]) {
36             template_name += coordinate.direction;
37         }
38     }, this);
39     template_name += ".json";
40     return template_name;
41 };
```

Phaser states of our demo

We will save the level data of our demo in a JSON file, which will be read when it starts.
The JSON file I'm going to use is shown below. This file will define the game assets and
groups, which will be the same for any room. This way we can preload all the assets in a
LoadingState and create all groups before loading the map. Since the map file will be
different for each room, it will be passed as a parameter, instead of being defined in the
JSON file.

```
1 {
2     "assets": {
3         "dungeon_tileset": {"type": "image", "source":
"assets/images/terrains.png"},
4         "hero_spritesheet": {"type": "spritesheet", "source":
"assets/images/player.png", "frame_width": 31, "frame_height":
30}
5     },
6     "groups": [
7         "doors",
8         "heroes"
9     ]
10 }
```

Our demo will have four states: BootState, LoadingState, DungeonState and RoomState.
The first two states are simple and responsible for loading the game JSON file and all the
necessary assets before starting any room. Their codes are shown below. You can see that

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

BootState simply loads the JSON file in its "preload" method and starts the LoadingState in the "create" method. The LoadingState, by its turn, loads all the assets in the "preload" method, by using the asset type to call the appropriate Phaser method. When all assets are loaded, it starts the next state (in the "next_state" variable) in the "create" method.

```
1 var ProceduralGeneration = ProceduralGeneration || {};
2
3 ProceduralGeneration.BootState = function () {
4    "use strict";
5    Phaser.State.call(this);
6 };
7
8 ProceduralGeneration.BootState.prototype =
Object.create(Phaser.State.prototype);
9 ProceduralGeneration.BootState.prototype.constructor =
ProceduralGeneration.BootState;
10
11 ProceduralGeneration.BootState.prototype.init = function
(level_file, next_state, extra_parameters) {
12    "use strict";
13    this.level_file = level_file;
14    this.next_state = next_state;
15    this.extra_parameters = extra_parameters;
16 };
17
18 ProceduralGeneration.BootState.prototype.preload = function
() {
19    "use strict";
20    this.load.text("level1", this.level_file);
21 };
22
23 ProceduralGeneration.BootState.prototype.create = function ()
{
24    "use strict";
25    var level_text, level_data;
26    level_text = this.game.cache.getText("level1");
27    level_data = JSON.parse(level_text);
28    this.game.state.start("LoadingState", true, false,
level_data, this.next_state, this.extra_parameters);
29};
```

```
1 var ProceduralGeneration = ProceduralGeneration || {};
2
3 ProceduralGeneration.LoadingState = function () {
```

```
4     "use strict";
5     Phaser.State.call(this);
6 };
7
8 ProceduralGeneration.LoadingState.prototype =
Object.create(Phaser.State.prototype);
9 ProceduralGeneration.LoadingState.prototype.constructor =
ProceduralGeneration.LoadingState;
10
11 ProceduralGeneration.LoadingState.prototype.init = function
(level_data, next_state, extra_parameters) {
12     "use strict";
13     this.level_data = level_data;
14     this.next_state = next_state;
15     this.extra_parameters = extra_parameters;
16 };
17
18 ProceduralGeneration.LoadingState.prototype.preload =
function () {
19     "use strict";
20     var assets, asset_loader, asset_key, asset;
21     assets = this.level_data.assets;
22     for (asset_key in assets) { // load assets according to
asset key
23         if (assets.hasOwnProperty(asset_key)) {
24             asset = assets[asset_key];
25             switch (asset.type) {
26             case "image":
27                 this.load.image(asset_key, asset.source);
28                 break;
29             case "spritesheet":
30                 this.load.spritesheet(asset_key, asset.source,
asset.frame_width, asset.frame_height, asset.frames,
asset.margin, asset.spacing);
31                 break;
32             case "tilemap":
33                 this.load.tilemap(asset_key, asset.source,
null, Phaser.Tilemap.TILED_JSON);
34                 break;
35             }
36         }
37     }
38 };
39
```

```
40 ProceduralGeneration.LoadingState.prototype.create = function
() {
41    "use strict";
42    this.game.state.start(this.next_state, true, false,
this.level_data, this.extra_parameters);
43 };
```

The DungeonState will be responsible for generating the dungeon, and it is shown below.
It initializes the Dungeon object in the "init" method and generate a new dungeon in the
"create" method. After generating the dungeon it starts a RoomState with the initial room
of the dungeon.

```
1 var ProceduralGeneration = ProceduralGeneration || {};
2
3 ProceduralGeneration.DungeonState = function () {
4    "use strict";
5    Phaser.State.call(this);
6
7    this.LEVEL_FILE = "assets/levels/room_level.json";
8 };
9
10 ProceduralGeneration.DungeonState.prototype =
Object.create(Phaser.State.prototype);
11 ProceduralGeneration.DungeonState.prototype.constructor =
ProceduralGeneration.DungeonState;
12
13 ProceduralGeneration.DungeonState.prototype.init = function
(number_of_rooms) {
14    "use strict";
15    this.number_of_rooms = number_of_rooms;
16    this.dungeon = this.dungeon || new
ProceduralGeneration.Dungeon(this);
17 };
18
19 ProceduralGeneration.DungeonState.prototype.create = function
() {
20    "use strict";
21    var initial_room;
22    // generate new dungeon
23    initial_room =
this.dungeon.generate_dungeon(this.number_of_rooms);
24    // start RoomState for the initial room of the dungeon
25    this.game.state.start("BootState", true, false,
this.LEVEL_FILE, "RoomState", {room: initial_room});
```

```
26 };
```
Finally, RoomState is where most of the game will run. In the "init" method it starts the physics engine, sets the scale and save data for other methods. In the "preload" method it loads the map file given by the room template name. Then, in the "create" method it creates the map, the map layers, the groups and the prefabs. Notice that when creating the layers we check for the collision property to make it collidable. The "create_object" method is responsible for creating the prefabs. First, it adjusts the positions, since Tiled and Phaser coordinate systems are different, then it instantiate the correct prefab using the "prefab_classes" property (initially empty). Notice that, this is possible because all prefabs have the same constructor, defined in their base class shown below.

```
1 var ProceduralGeneration = ProceduralGeneration || {};
2
3 ProceduralGeneration.RoomState = function () {
4    "use strict";
5    Phaser.State.call(this);
6
7    this.MAP_KEY = "room_tilemap";
8    this.MAP_TILESET = "dungeon_tileset";
9
10   this.prefab_classes = {
11
12   };
13 };
14
15 ProceduralGeneration.RoomState.prototype =
Object.create(Phaser.State.prototype);
16 ProceduralGeneration.RoomState.prototype.constructor =
ProceduralGeneration.RoomState;
17
18 ProceduralGeneration.RoomState.prototype.init = function
(level_data, extra_parameters) {
19    "use strict";
20    var tileset_index, tile_dimensions;
21    this.level_data = this.level_data || level_data;
22
23    this.scale.scaleMode = Phaser.ScaleManager.SHOW_ALL;
24    this.scale.pageAlignHorizontally = true;
25    this.scale.pageAlignVertically = true;
26
27    // start physics system
28    this.game.physics.startSystem(Phaser.Physics.ARCADE);
29    this.game.physics.arcade.gravity.y = 0;
30
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
31      // get current room
32      this.room = extra_parameters.room;
33 };
34
35 ProceduralGeneration.RoomState.prototype.preload = function
() {
36      "use strict";
37      this.load.tilemap(this.MAP_KEY, "assets/maps/" +
this.room.template_name(), null, Phaser.Tilemap.TILED_JSON);
38 };
39
40 ProceduralGeneration.RoomState.prototype.create = function ()
{
41      "use strict";
42      var group_name, object_layer, collision_tiles;
43      // create map
44      this.map = this.game.add.tilemap(this.MAP_KEY);
45      this.map.addTilesetImage(this.map.tilesets[0].name,
this.MAP_TILESET);
46
47      // create map layers
48      this.layers = {};
49      this.map.layers.forEach(function (layer) {
50          this.layers[layer.name] =
this.map.createLayer(layer.name);
51          if (layer.properties.collision) { // collision layer
52              collision_tiles = [];
53              layer.data.forEach(function (data_row) { // find
tiles used in the layer
54                  data_row.forEach(function (tile) {
55                      // check if it's a valid tile index and
isn't already in the list
56                      if (tile.index > 0 &&
collision_tiles.indexOf(tile.index) === -1) {
57                          collision_tiles.push(tile.index);
58                      }
59                  }, this);
60              }, this);
61              this.map.setCollision(collision_tiles, true,
layer.name);
62          }
63      }, this);
64      // resize the world to be the size of the current layer
65      this.layers[this.map.layer.name].resizeWorld();
66
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
67     // create groups
68     this.groups = {};
69     this.level_data.groups.forEach(function (group_name) {
70         this.groups[group_name] = this.game.add.group();
71     }, this);
72
73     this.prefabs = {};
74
75     // create objects (prefabs)
76     for (object_layer in this.map.objects) {
77         if (this.map.objects.hasOwnProperty(object_layer)) {
78             // create layer objects
79             this.map.objects[object_layer].forEach(this.create
_object, this);
80         }
81     }
82 };
83
84 ProceduralGeneration.RoomState.prototype.create_object =
function (object) {
85     "use strict";
86     var object_y, position, prefab;
87     // tiled coordinates starts in the bottom left corner
88     object_y = (object.gid) ? object.y - (this.map.tileHeight
/ 2) : object.y + (object.height / 2);
89     position = {"x": object.x + (this.map.tileHeight / 2),
"y": object_y};
90     // create object according to its type
91     if (this.prefab_classes.hasOwnProperty(object.type)) {
92         prefab = new this.prefab_classes[object.type](this,
object.name, position, object.properties);
93     }
94     this.prefabs[object.name] = prefab;
95 };

1 var ProceduralGeneration = ProceduralGeneration || {};
2
3 ProceduralGeneration.Prefab = function (game_state, name,
position, properties) {
4     "use strict";
5     Phaser.Sprite.call(this, game_state.game, position.x,
position.y, properties.texture);
6
7     this.game_state = game_state;
8
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
9     this.name = name;
10
11     this.game_state.groups[properties.group].add(this);
12     this.frame = +properties.frame;
13
14     if (properties.scale) {
15         this.scale.setTo(properties.scale.x,
properties.scale.y);
16     }
17
18     this.game_state.prefabs[name] = this;
19 };
20
21 ProceduralGeneration.Prefab.prototype =
Object.create(Phaser.Sprite.prototype);
22 ProceduralGeneration.Prefab.prototype.constructor =
ProceduralGeneration.Prefab;
```

By now you can already try running the demo to see if its correcting loading the room maps. Try running multiple times and see if the initial room is changing.
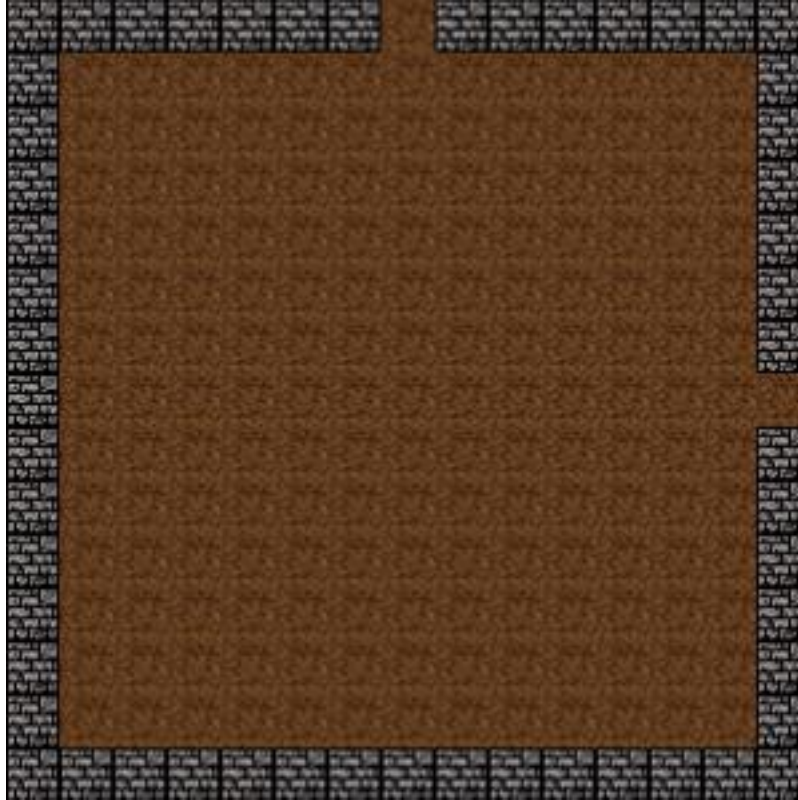
Navigating through the rooms

You may have noticed the Tiled maps provided in the source code have hero and doors prefabs, which will be used to allow our hero to navigate through the dungeon rooms. Now, we are going to implement them.

First, the Hero prefab code is shown below. The only thing it will do by now is walk, so it just needs a "walking_speed" property. In the "update" method we check for player input to move the hero, which is done with the "cursors" object. To properly control the hero, we move it to a given direction only if it is not already moving to the opposite direction. For example, the hero can move left only if it is not already moving right. Also, if the player is moving, we must play the walking animation, otherwise stop it.

```
1  var ProceduralGeneration = ProceduralGeneration || {};
2
3  ProceduralGeneration.Hero = function (game_state, name,
position, properties) {
4      "use strict";
5      ProceduralGeneration.Prefab.call(this, game_state, name,
position, properties);
6
7      this.anchor.setTo(0.5);
8
9      this.walking_speed = +properties.walking_speed;
10
11      this.game_state.game.physics.arcade.enable(this);
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
12      this.body.collideWorldBounds = true;
13
14      this.animations.add("walking", [0, 1], 6, true);
15
16      this.cursors =
this.game_state.game.input.keyboard.createCursorKeys();
17 };
18
19 ProceduralGeneration.Hero.prototype =
Object.create(ProceduralGeneration.Prefab.prototype);
20 ProceduralGeneration.Hero.prototype.constructor =
ProceduralGeneration.Hero;
21
22 ProceduralGeneration.Hero.prototype.update = function () {
23      "use strict";
24      this.game_state.game.physics.arcade.collide(this,
this.game_state.layers.collision);
25
26      if (this.cursors.left.isDown && this.body.velocity.x <= 0)
{ // move left if is not already moving right
27          this.body.velocity.x = -this.walking_speed;
28          this.scale.setTo(1, 1);
29      } else if (this.cursors.right.isDown &&
this.body.velocity.x >= 0) { // move right if is not already
moving left
30          this.body.velocity.x = +this.walking_speed;
31          this.scale.setTo(-1, 1);
32      } else {
33          this.body.velocity.x = 0;
34      }
35
36      if (this.cursors.up.isDown && this.body.velocity.y <= 0) {
// move up if is not already moving down
37          this.body.velocity.y = -this.walking_speed;
38      } else if (this.cursors.down.isDown &&
this.body.velocity.y >= 0) { // move down if is not already
moving up
39          this.body.velocity.y = +this.walking_speed;
40      } else {
41          this.body.velocity.y = 0;
42      }
43
44      if (this.body.velocity.x === 0 && this.body.velocity.y ===
0) {
45          this.animations.stop();
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
46          this.frame = 0;
47      } else {
48          // if not moving, stop the animation
49          this.animations.play("walking");
50      }
51 };
```

Now, we implement the Door prefab as shown below. It will have a "direction" property so we can know to where the player is navigating. In the "update" method we check for collisions with the hero, and if so, call the "enter_door" method. This method gets the next room using the door direction and starts a new RoomState for the next room.

```
1 var ProceduralGeneration = ProceduralGeneration || {};
2
3 ProceduralGeneration.Door = function (game_state, name,
position, properties) {
4     "use strict";
5     ProceduralGeneration.Prefab.call(this, game_state, name,
position, properties);
6
7     this.anchor.setTo(0.5);
8
9     this.direction = properties.direction;
10
11     this.game_state.game.physics.arcade.enable(this);
12     this.body.collideWorldBounds = true;
13 };
14
15 ProceduralGeneration.Door.prototype =
Object.create(ProceduralGeneration.Prefab.prototype);
16 ProceduralGeneration.Door.prototype.constructor =
ProceduralGeneration.Door;
17
18 ProceduralGeneration.Door.prototype.update = function () {
19     "use strict";
20     this.game_state.game.physics.arcade.collide(this,
this.game_state.groups.heroes, this.enter_door, null, this);
21 };
22
23 ProceduralGeneration.Door.prototype.enter_door = function ()
{
24     "use strict";
25     var next_room;
26     // find the next room using the door direction
```

```
27    next_room =
this.game_state.room.neighbors[this.direction];
28    // start room state for the next room
29    this.game_state.game.state.start("BootState", true, false,
"assets/levels/room_level.json", "RoomState", {room:
next_room});
30 };
```

Finally, we add both the Hero and Door prefabs to the "prefab_classes" property in the RoomState. And then, you can actually run the demo navigating through the different rooms in the dungeon.

```
1 this.prefab_classes = {
2       "hero":
ProceduralGeneration.Hero.prototype.constructor,
3       "door": ProceduralGeneration.Door.prototype.constructor
4 };
```

And this conclude this tutorial. In the next one we will populate the rooms with random obstacles and enemies. Then, we will add an exit, so the hero can leave the dungeon.

How to Procedurally Generate a Dungeon in Phaser – Part 2
By Renan Oliveira

In the previous tutorial we procedurally generated a dungeon with multiple rooms, allowing our hero to navigate through it. In this tutorial, we are going to populate those rooms with obstacles, enemies and add an exit, so the hero can leave the dungeon.

The following topics will be covered in this tutorial:

- A strategy to procedurally populate rooms with objects
- Populating the dungeon rooms with obstacle tiles
- Populating the dungeon rooms with enemies
- Adding an exit in one room so the player can leave the dungeon
- Locking the room doors until all enemies have been defeated

To read this tutorial, it is important that you're familiar with the following concepts:

- Javascript and object-oriented concepts.
- Basic Phaser concepts, such as: states, sprites, groups and arcade physics
- Creating maps using Tiled

Source code files

You can download the tutorial source code files here.

I modified some assets (images and maps), so I highly suggest you download them again even you have them from the previous tutorial.

Strategy to populate the rooms

Given an object and its dimensions (width and height) we need to find a free region in the room that fits the object. We do that by looking random regions with the object dimensions until we find one that is free. Since most of the room is free, this process is typically fast.

The "find_free_region" method below belongs to the Room class and is responsible for doing that. It runs a loop that starts by finding a random position that will be the center of the region. Then, it add the coordinates of the other positions that will be occupied by the object, according to its dimensions. Finally, it checks if the whole region is free. If so, we're done. Otherwise, it keeps running the loop.

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Page 411

```
1 ProceduralGeneration.Room.prototype.find_free_region =
function (size_in_tiles) {
2     "use strict";
3     var center_tile, region, x_coordinate, y_coordinate,
initial_x_coordinate, initial_y_coordinate;
4     do {
5         // pick a random coordinate to be the center of the
region
6         center_tile = new
Phaser.Point(this.game_state.game.rnd.between(2,
(this.game_state.game.world.width / this.tile_dimensions.x) -
3),
7                                 this.game_state.game.rnd.be
tween(2, (this.game_state.game.world.height /
this.tile_dimensions.y) - 3));
8         region = [center_tile];
9         initial_x_coordinate = center_tile.x -
Math.floor(size_in_tiles.x / 2);
10         initial_y_coordinate = center_tile.y -
Math.floor(size_in_tiles.y / 2);
11         // add all coordinates of the region, based in its
size
12         for (x_coordinate = initial_x_coordinate; x_coordinate
< initial_x_coordinate + size_in_tiles.x; x_coordinate += 1) {
13             for (y_coordinate = initial_y_coordinate;
y_coordinate < initial_y_coordinate + size_in_tiles.y;
y_coordinate += 1) {
14                 region.push(new Phaser.Point(x_coordinate,
y_coordinate));
15             }
16         }
17     } while (!this.is_free(region)); // stop if all the region
is free
18     return region;
19 };
20
21 ProceduralGeneration.Room.prototype.is_free = function
(region) {
22     "use strict";
23     var coordinate_index, coordinate;
24     for (coordinate_index = 0; coordinate_index <
region.length; coordinate_index += 1) {
25         coordinate = region[coordinate_index];
```

```
26          // check if there is an object occupying this
coordinate
27          if (this.population[coordinate.y][coordinate.x]) {
28              return false;
29          }
30      }
31      return true;
32 };
```

The "is_free" method iterate through all coordinates and check the "population" property. This property starts as empty and is filled every time an object is added to the room. So, if there is another object in that position, the method will return that the region is not free.

Population data

We will store the population in a JSON file like the one below. You can use the same, provided in the source code or create your own. The important thing is that this file describe two kinds of population: tiles and prefabs. For each one, you can also define different kinds (for example, you can add enemies and items in the prefabs section). To simplify the tutorial, I only added one type of tile and prefab. Finally, for each tile type you must define the layer it will use, the minimum and maximum number of objects that will be created, the possible sizes and possible tiles. On the other hand, for each prefab type you must define the minimum and maximum number of objects and the possible prefabs.

```
1 {
2     "tiles": {
3         "obstacles": {
4             "layer": "collision",
5             "number": {"min": 3, "max": 5},
6             "sizes": [{"x": 1, "y": 1}, {"x": 1, "y": 2}, {"x": 1, "y": 3}, {"x": 2, "y": 1}, {"x": 3, "y": 1}],
7             "possible_tiles": [10]
8         }
9     },
10     "prefabs": {
11         "enemies": {
12             "number": {"min": 1, "max": 3},
13             "possible_prefabs": [
14                 {
15                     "prefab": "enemy",
16                     "properties": {"texture": "enemy_image",
"group": "enemies"}
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
17                          }
18              ]
19          }
20      }
21 }
```

The population JSON file will be loaded in the DungeonState, since it will be used during the dungeon generation.

```
1 ProceduralGeneration.DungeonState.prototype.preload = function
() {
2     "use strict";
3     // load the population JSON file
4     this.load.text("population",
"assets/levels/population.json");
5 };
```

Populating rooms with obstacle tiles
We will start by populating our rooms with obstacle tiles. Each obstacle will use a random tile index (from a list of tiles) and will have random dimensions (from a list of possible sizes).

The code below show the "populate_tiles" method. For each tile, it randomly picks a tile index, a size (available from the population data) and finds a free region. Then it adds all coordinates of this region to a "tiles" object and to the population. The "tiles" object will be read by the RoomState to add the tiles in the game.

```
1 ProceduralGeneration.Room.prototype.populate_tiles = function
(number_of_tiles, layer, possible_tiles, possible_sizes) {
2     "use strict";
3     var index, tile, region_size, region, coordinate_index;
4     for (index = 0; index < number_of_tiles; index += 1) {
5         // pick a random tile index
6         tile = this.game_state.game.rnd.pick(possible_tiles);
7         // pick a random size
8         region_size =
this.game_state.game.rnd.pick(possible_sizes);
9         // find a free region with the picked size
10        region = this.find_free_region(region_size);
11        // add all region coordinates to the tiles property
12        for (coordinate_index = 0; coordinate_index <
region.length; coordinate_index += 1) {
```

```
13              this.tiles.push({layer: layer, tile: tile,
position: region[coordinate_index]});
14              this.population[region[coordinate_index].y][region
[coordinate_index].x] = tile;
15          }
16      }
17 };
```

The "populate" method from the Room class is shown below. First, it initializes the "population" property as empty, so that all coordinates are initially free. Then, it iterates through all obstacles in the population data creating them. For each obstacle, it chooses a random number of obstacles and call the "populate_tiles" method.

```
1 ProceduralGeneration.Room.prototype.populate = function
(population) {
2     "use strict";
3     var number_of_rows, number_of_columns, row_index,
column_index, tile_type, number_of_tiles, prefab_type,
number_of_prefabs;
4     number_of_rows = this.game_state.game.world.height /
this.tile_dimensions.y;
5     number_of_columns = this.game_state.game.world.width /
this.tile_dimensions.x;
6     // initialize the population object as empty
7     for (row_index = 0; row_index <= number_of_rows; row_index
+= 1) {
8         this.population.push([]);
9         for (column_index = 0; column_index <=
number_of_columns; column_index += 1) {
10             this.population[row_index][column_index] = null;
11         }
12     }
13
14     // populate the room with tiles
15     for (tile_type in population.tiles) {
16         if (population.tiles.hasOwnProperty(tile_type)) {
17             // pick a random number of tiles
18             number_of_tiles =
this.game_state.game.rnd.between(population.tiles[tile_type].num
ber.min, population.tiles[tile_type].number.max);
19             // create the tiles
20             this.populate_tiles(number_of_tiles,
population.tiles[tile_type].layer,
```

```
population.tiles[tile_type].possible_tiles,
population.tiles[tile_type].sizes);
21          }
22       }
23 };
```

We have to change the "generate_dungeon" method in the Dungeon class to load the
population data and populate the rooms, as shown below.

```
1 // load the population data from the JSON file
2     population =
JSON.parse(this.game_state.game.cache.getText("population"));
3
4     // iterate through rooms to connect and populate them
5     created_rooms.forEach(function (room) {
6         room.neighbor_coordinates().forEach(function
(coordinate) {
7             if (this.grid[coordinate.row][coordinate.column]) {
8                 room.connect(coordinate.direction,
this.grid[coordinate.row][coordinate.column]);
9             }
10        }, this);
11        // populate the room
12        room.populate(population);
13    }, this);
```

Now, we have to add the following code at the end of the "create" method in RoomState.
This code will iterate through all tiles in the "tiles" object and call the "putTile" method
from Phaser.Tilemap for each one. This method allows us to add new tiles to a previously
loaded Tiled map (you can learn more in Phaser documentation). There is another very
important change you have to make in this class. In the previous tutorial of this series, we
set the collision of layers for only the tiles in that layers. Since now we are going to add
new tiles to an already created layer, we have to change this code to set the collision for
all tiles, as shown below.

```
1 // add tiles to the room
2     this.room.tiles.forEach(function (tile) {
3         this.map.putTile(tile.tile, tile.position.x,
tile.position.y, tile.layer);
4     }, this);
```

```
1 // create map layers
2     this.layers = {};
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
3    this.map.layers.forEach(function (layer) {
4        this.layers[layer.name] =
this.map.createLayer(layer.name);
5        if (layer.properties.collision) { // collision layer
6            this.map.setCollisionByExclusion([-1], true,
layer.name);
7        }
8    }, this);
```


You can already try playing the demo with the obstacles.
Populating rooms with enemies

Populating the rooms with enemies will be very similar to how we did with obstacles, but we must save the prefab information instead of tiles. Also, to simplify the code we will assume all prefabs occupy a single tile.

The code below shows the "populate_prefabs" method. Instead of picking a random tile index, it picks a random prefab, and it finds a random region with size of only one tile (which results in a random position). Then it adds the prefab name, type, position and properties to a "prefab" object, which will be read by RoomState as well.

```
1 ProceduralGeneration.Room.prototype.populate_prefabs =
function (number_of_prefabs, possible_prefabs_data) {
2    "use strict";
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
3    var index, prefab_data, prefab, tile_position, position,
properties;
4    for (index = 0; index < number_of_prefabs; index += 1) {
5        // pick a random prefab
6        prefab_data =
this.game_state.game.rnd.pick(possible_prefabs_data);
7        prefab = prefab_data.prefab;
8        // find a free region of size one
9        tile_position = this.find_free_region({x: 1, y: 1});
10        position = new Phaser.Point((tile_position[0].x *
this.tile_dimensions.x) + (this.tile_dimensions.x / 2),
11                            (tile_position[0].y *
this.tile_dimensions.y) + (this.tile_dimensions.y / 2));
12        properties = prefab_data.properties;
13        // add the prefab to the prefabs property
14        this.prefabs.push({name: prefab + index, prefab:
prefab, position: position, properties: properties});
15        this.population[tile_position[0].y][tile_position[0].x
] = prefab;
16    }
17 };
```

We must add in the "populate" method the code to add the prefabs. Similarly to what we did with the tiles, we iterate through all prefab population data, choose a random number of prefabs and call the "populate_prefabs" method.

```
1    // populate the room with prefabs
2    for (prefab_type in population.prefabs) {
3        if (population.prefabs.hasOwnProperty(prefab_type)) {
4            // pick a random number of prefabs
5            number_of_prefabs =
this.game_state.game.rnd.between(population.prefabs[prefab_type]
.number.min, population.prefabs[prefab_type].number.max);
6            // create the prefabs
7            this.populate_prefabs(number_of_prefabs,
population.prefabs[prefab_type].possible_prefabs);
8        }
9    }
```

Finally, we add the following piece of code to the "create" method in RoomState, so it creates the prefabs after adding the obstacle tiles. This code simply goes through all added prefabs and create them using its "create_prefab" method.

```
1 // add prefabs to the room
```

```
2     this.room.prefabs.forEach(function (prefab) {
3         new_prefab = new
this.prefab_classes[prefab.prefab](this, prefab.name,
prefab.position, prefab.properties);
4     }, this);
```

To be able to verify if everything is working by now, you have to create a Enemy prefab
so the demo can run, as shown below. Remember that every time you create a new prefab
you must add it to the "prefab_classes" property in RoomState. By now you can already
try playing the demo with the enemies as well.

```
1 var ProceduralGeneration = ProceduralGeneration || {};
2
3 ProceduralGeneration.Enemy = function (game_state, name,
position, properties) {
4     "use strict";
5     ProceduralGeneration.Prefab.call(this, game_state, name,
position, properties);
6
7     this.anchor.setTo(0.5);
8
9     this.game_state.game.physics.arcade.enable(this);
10     this.body.immovable = true;
11 };
12
13 ProceduralGeneration.Enemy.prototype =
Object.create(ProceduralGeneration.Prefab.prototype);
```

```
14 ProceduralGeneration.Enemy.prototype.constructor =
```



```
ProceduralGeneration.Enemy;
```

Adding the dungeon exit

Before adding the exit, we must create its prefab, as shown below. The Exit prefab simply checks for collisions with the hero and, if it detects one, it restarts the demo by calling the DungeonState again.

```
1 var ProceduralGeneration = ProceduralGeneration || {};
2
3 ProceduralGeneration.Exit = function (game_state, name,
position, properties) {
4     "use strict";
5     ProceduralGeneration.Prefab.call(this, game_state, name,
position, properties);
```

<u>Zenva Academy</u> – Online courses on Phaser and game programming
<u>Zenva for Schools</u> – Coding courses for high schools

Page 420

```
 6
 7      this.anchor.setTo(0.5);
 8
 9      this.direction = properties.direction;
10
11      this.game_state.game.physics.arcade.enable(this);
12      this.body.immovable = true;
13 };
14
15 ProceduralGeneration.Exit.prototype =
Object.create(ProceduralGeneration.Prefab.prototype);
16 ProceduralGeneration.Exit.prototype.constructor =
ProceduralGeneration.Exit;
17
18 ProceduralGeneration.Exit.prototype.update = function () {
19      "use strict";
20      this.game_state.game.physics.arcade.collide(this,
this.game_state.groups.heroes, this.reach_exit, null, this);
21 };
22
23 ProceduralGeneration.Exit.prototype.reach_exit = function ()
{
24      "use strict";
25      if (this.game_state.groups.enemies.countLiving() === 0) {
26          // restart the game
27          this.game_state.game.state.start("DungeonState", true,
false, 10);
28      }
29 };
```

We will add the exit of the dungeon in the furthest room from the initial one. To do that, we must change the "generate_dungeon" method to keep track of the furthest room, as shown below. When populating a room, we calculate its distance to the initial room and save the coordinate of the furthest one. Then, after populating all rooms we add the exit to the room with the final room coordinate.

```
1 max_distance_to_initial_room = 0;
2     // iterate through rooms to connect and populate them
3     created_rooms.forEach(function (room) {
4         room.neighbor_coordinates().forEach(function
(coordinate) {
5             if (this.grid[coordinate.row][coordinate.column]) {
6                 room.connect(coordinate.direction,
this.grid[coordinate.row][coordinate.column]);
7             }
8         }, this);
9         // populate the room
10         room.populate(population);
11
12         // check distance to the initial room
13         distance_to_initial_room =
Math.abs(room.coordinate.column - initial_room_coordinate.x) +
Math.abs(room.coordinate.row - initial_room_coordinate.y);
14         if (distance_to_initial_room >
max_distance_to_initial_room) {
15             final_room_coordinate.x = room.coordinate.column;
16             final_room_coordinate.y = room.coordinate.row;
17         }
18     }, this);
19
20     this.grid[final_room_coordinate.y][final_room_coordinate.x
].populate_prefabs(1, [{prefab: "exit", properties: {texture:
"exit_image", group: "exits"}}]);
```


Try playing the demo now and search for the exit.

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Fighting enemies

Our hero still can't defeat any enemies, so you must add code to do that. Since this is not the focus of this tutorial, I will only make the hero kill the enemy when they overlap, as shown below. For an actual game you could add attack and defense stats and calculate the damage based on them. Feel free to improve this code in order to make something fun.

```
1 var ProceduralGeneration = ProceduralGeneration || {};
2
3 ProceduralGeneration.Enemy = function (game_state, name,
position, properties) {
4    "use strict";
5    ProceduralGeneration.Prefab.call(this, game_state, name,
position, properties);
6
7    this.anchor.setTo(0.5);
8
9    this.game_state.game.physics.arcade.enable(this);
10    this.body.immovable = true;
11 };
12
13 ProceduralGeneration.Enemy.prototype =
Object.create(ProceduralGeneration.Prefab.prototype);
14 ProceduralGeneration.Enemy.prototype.constructor =
ProceduralGeneration.Enemy;
15
16 ProceduralGeneration.Enemy.prototype.update = function () {
17    "use strict";
18    this.game_state.game.physics.arcade.overlap(this,
this.game_state.groups.heroes, this.kill, null, this);
19 };
```

Now that our hero can defeat enemies, we must lock the rooms (and the exit) until all enemies in that room have been defeated. To do that, we simply check if the enemies group has no alive objects in the "enter_door" and "reach_exit" methods, as shown below.

```
1 ProceduralGeneration.Door.prototype.enter_door = function () {
2     "use strict";
3     var next_room;
4     if (this.game_state.groups.enemies.countLiving() === 0) {
5         // find the next room using the door direction
6         next_room =
this.game_state.room.neighbors[this.direction];
7         // start room state for the next room
8         this.game_state.game.state.start("BootState", true,
false, "assets/levels/room_level.json", "RoomState", {room:
next_room});
9     }
10 };
```

```
1 ProceduralGeneration.Exit.prototype.reach_exit = function () {
2     "use strict";
3     if (this.game_state.groups.enemies.countLiving() === 0) {
4         // restart the game
5         this.game_state.game.state.start("DungeonState", true,
false, 10);
6     }
7 };
```

Now, you can try playing the demo again and check if you can find the dungeon exit. And that concludes our tutorial series about procedurally generated content.

How to Create a Game HUD Plugin in Phaser
By Renan Oliveira

In a game, the heads-up display (HUD) is how game information is visually showed to the player, providing a feedback from the game. Usually, it provides information about player stats like health, items and menus. The HUD is very important in many games to make sure the player understands what is happening in the game. Besides, it is highly dependent on the game and during the game development, it is useful to try different HUD strategies in order to find the best one. So, it is important to have a way of easily manage HUD elements in order to try different HUD configurations.

In this tutorial we will create a HUD plugin to manage HUD elements in the game screen. We will then use it by creating a simple Tiled level and some basic HUD elements. To read this tutorial, it is important that you're familiar with the following concepts:

- Javascript and object-oriented concepts.
- Basic Phaser concepts, such as: states, sprites, groups and arcade physics.
- Creating maps using Tiled.

Source code files

You can download the tutorial source code files here.

Game states
We're going to keep the game data in a JSON file as shown below. This file describes the game assets that must be loaded, the groups that must be created and the tiled map data. To load this file and setup the game data before it starts we will need three game states: BootState, LoadingState and WorldState.

```
1 {
2     "assets": {
3         "hero_spritesheet": { "type": "spritesheet", "source":
"assets/images/player.png", "frame_width": 31, "frame_height":
30 },
4         "weapon_image": { "type": "image", "source":
"assets/images/attack-icon.png" },
5         "coin_image": { "type": "image", "source":
"assets/images/coin.png" },
6         "potion_image": { "type": "image", "source":
"assets/images/potion.png" },
```

```
7          "shield_image": { "type": "image", "source":
"assets/images/shield.png" },
8          "chest_image": { "type": "image", "source":
"assets/images/chest.png" },
9          "healthbar_image": { "type": "image", "source":
"assets/images/healthbar.png" },
10
11         "level_tileset": { "type": "image", "source":
"assets/images/terrains.png" },
12         "level_tilemap": { "type": "tilemap", "source":
"assets/maps/world.json" }
13     },
14     "groups": [
15         "items",
16         "heroes",
17         "hud",
18         "stats"
19     ],
20     "map": {
21         "key": "level_tilemap",
22         "tilesets": ["level_tileset"]
23     }
24 }
```

BootState and LoadingState codes are shown below. The first one simply loads this JSON file and calls LoadingState with the level data. LoadingState, by its turn, load all game assets calling the correct Phaser method according to the asset type (for example, calling "this.load.image" to load an image).

```
1 var HUDExample = HUDExample || {};
2
3 HUDExample.BootState = function () {
4     "use strict";
5     Phaser.State.call(this);
6 };
7
8 HUDExample.BootState.prototype =
Object.create(Phaser.State.prototype);
9 HUDExample.BootState.prototype.constructor =
HUDExample.BootState;
10
11 HUDExample.BootState.prototype.init = function (level_file,
next_state, extra_parameters) {
12     "use strict";
```

```
13      this.level_file = level_file;
14      this.next_state = next_state;
15      this.extra_parameters = extra_parameters;
16  };
17
18  HUDExample.BootState.prototype.preload = function () {
19      "use strict";
20      this.load.text("level1", this.level_file);
21  };
22
23  HUDExample.BootState.prototype.create = function () {
24      "use strict";
25      var level_text, level_data;
26      level_text = this.game.cache.getText("level1");
27      level_data = JSON.parse(level_text);
28      this.game.state.start("LoadingState", true, false,
level_data, this.next_state, this.extra_parameters);
29  };
```

```
1 var HUDExample = HUDExample || {};
2
3 HUDExample.LoadingState = function () {
4      "use strict";
5      Phaser.State.call(this);
6 };
7
8 HUDExample.LoadingState.prototype =
Object.create(Phaser.State.prototype);
9 HUDExample.LoadingState.prototype.constructor =
HUDExample.LoadingState;
10
11 HUDExample.LoadingState.prototype.init = function
(level_data, next_state, extra_parameters) {
12      "use strict";
13      this.level_data = level_data;
14      this.next_state = next_state;
15      this.extra_parameters = extra_parameters;
16 };
17
18 HUDExample.LoadingState.prototype.preload = function () {
19      "use strict";
20      var assets, asset_loader, asset_key, asset;
21      assets = this.level_data.assets;
22      for (asset_key in assets) { // load assets according to
asset key
```

```
23          if (assets.hasOwnProperty(asset_key)) {
24              asset = assets[asset_key];
25              switch (asset.type) {
26              case "image":
27                  this.load.image(asset_key, asset.source);
28                  break;
29              case "spritesheet":
30                  this.load.spritesheet(asset_key, asset.source,
asset.frame_width, asset.frame_height, asset.frames,
asset.margin, asset.spacing);
31                  break;
32              case "tilemap":
33                  this.load.tilemap(asset_key, asset.source,
null, Phaser.Tilemap.TILED_JSON);
34                  break;
35              }
36          }
37      }
38 };
39
40 HUDExample.LoadingState.prototype.create = function () {
41     "use strict";
42     this.game.state.start(this.next_state, true, false,
this.level_data, this.extra_parameters);
43 };
```

Finally, WorldState (shown below) loads the Tiled map and create the game groups.
First, in the "init" method it starts the physics engine and creates the Tiled map from the
data in the JSON file. The "create" method, by its turn, creates the map layers, groups
and prefabs.

```
1 var HUDExample = HUDExample || {};
2
3 HUDExample.WorldState = function () {
4     "use strict";
5     Phaser.State.call(this);
6
7     this.MAP_KEY = "room_tilemap";
8     this.MAP_TILESET = "dungeon_tileset";
9
10     this.prefab_classes = {
11         "hero": HUDExample.Hero.prototype.constructor,
12         "item": HUDExample.Item.prototype.constructor,
```

```
13          "show_stat_with_sprite":
HUDExample.ShowStatWithSprite.prototype.constructor,
14          "show_stat_with_text":
HUDExample.ShowStatWithText.prototype.constructor,
15          "show_stat_with_bar":
HUDExample.ShowStatWithBar.prototype.constructor
16      };
17 };
18
19 HUDExample.WorldState.prototype =
Object.create(Phaser.State.prototype);
20 HUDExample.WorldState.prototype.constructor =
HUDExample.WorldState;
21
22 HUDExample.WorldState.prototype.init = function (level_data,
extra_parameters) {
23      "use strict";
24      var tileset_index;
25      this.level_data = level_data;
26
27      this.scale.scaleMode = Phaser.ScaleManager.SHOW_ALL;
28      this.scale.pageAlignHorizontally = true;
29      this.scale.pageAlignVertically = true;
30
31      // start physics system
32      this.game.physics.startSystem(Phaser.Physics.ARCADE);
33      this.game.physics.arcade.gravity.y = 0;
34
35      // create map and set tileset
36      this.map = this.game.add.tilemap(level_data.map.key);
37      tileset_index = 0;
38      this.map.tilesets.forEach(function (tileset) {
39          this.map.addTilesetImage(tileset.name,
level_data.map.tilesets[tileset_index]);
40          tileset_index += 1;
41      }, this);
42 };
43
44 HUDExample.WorldState.prototype.create = function () {
45      "use strict";
46      var group_name, object_layer, collision_tiles;
47
48      // create map layers
49      this.layers = {};
50      this.map.layers.forEach(function (layer) {
```

```
51            this.layers[layer.name] =
this.map.createLayer(layer.name);
52         if (layer.properties.collision) { // collision layer
53            this.map.setCollisionByExclusion([-1], true,
layer.name);
54         }
55      }, this);
56      // resize the world to be the size of the current layer
57      this.layers[this.map.layer.name].resizeWorld();
58
59      // create groups
60      this.groups = {};
61      this.level_data.groups.forEach(function (group_name) {
62         this.groups[group_name] = this.game.add.group();
63      }, this);
64
65      this.prefabs = {};
66
67      for (object_layer in this.map.objects) {
68         if (this.map.objects.hasOwnProperty(object_layer)) {
69            // create layer objects
70            this.map.objects[object_layer].forEach(this.create
_object, this);
71         }
72      }
73
74      // initialize the HUD plugin
75      this.hud = this.game.plugins.add(HUDExample.HUD, this,
this.level_data.hud);
76
77      // set the camera to follow the hero
78      this.game.camera.follow(this.prefabs.hero);
79 };
80
81 HUDExample.WorldState.prototype.create_object = function
(object) {
82      "use strict";
83      var object_y, position;
84      // tiled coordinates starts in the bottom left corner
85      object_y = (object.gid) ? object.y - (this.map.tileHeight
/ 2) : object.y + (object.height / 2);
86      position = {"x": object.x + (this.map.tileHeight / 2),
"y": object_y};
87      this.create_prefab(object.type, object.name, position,
object.properties);
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
88 };
89
90 HUDExample.WorldState.prototype.create_prefab = function
(type, name, position, properties) {
91    "use strict";
92    var prefab;
93    // create prefab according to its type
94    if (this.prefab_classes.hasOwnProperty(type)) {
95        prefab = new this.prefab_classes[type](this, name,
position, properties);
96    }
97    this.prefabs[name] = prefab;
98    return prefab;
99 };
```

When creating the map layers we must check if the layer has a collision property as true. If so, we must set this layer as collidable.

The "create_object" method is responsible for creating the game prefabs from the map objects. First, it calculates the prefab position considering that Tiled and Phaser coordinate systems are different. Then, it calls the "create_prefab" method, which instantiates the correct prefab according to the "prefab_classes" property. This property is defined in WorldState constructor and maps each prefab type to its correspondent constructor. Notice that this can be done because all prefabs have the same constructor, which is defined in a generic Prefab class as shown below.

```
1 var HUDExample = HUDExample || {};
2
3 HUDExample.Prefab = function (game_state, name, position,
properties) {
4    "use strict";
5    Phaser.Sprite.call(this, game_state.game, position.x,
position.y, properties.texture);
6
7    this.game_state = game_state;
8
9    this.name = name;
10
11    this.game_state.groups[properties.group].add(this);
12    this.frame = +properties.frame;
13
14    if (properties.scale) {
15        this.scale.setTo(properties.scale.x,
properties.scale.y);
16    }
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
17
18    if (properties.anchor) {
19        this.anchor.setTo(properties.anchor.x,
properties.anchor.y);
20    }
21
22    this.game_state.prefabs[name] = this;
23 };
24
25 HUDExample.Prefab.prototype =
Object.create(Phaser.Sprite.prototype);
26 HUDExample.Prefab.prototype.constructor = HUDExample.Prefab;
```

After creating all prefabs, the WorldState only has to initialize the HUD plugin and set the camera to follow the game hero.

The HUD plugin

We're going to create a HUD plugin (using the Phaser plugin class) to easily manage the HUD elements in the screen. To do this we are going to divide the screen in regions, as shown in the figure below by the red rectangles. Each HUD element must be in one region, and a given region may contain several elements. In addition, we should be able

to define margins around the screen, so the HUD regions can start in custom positions, as
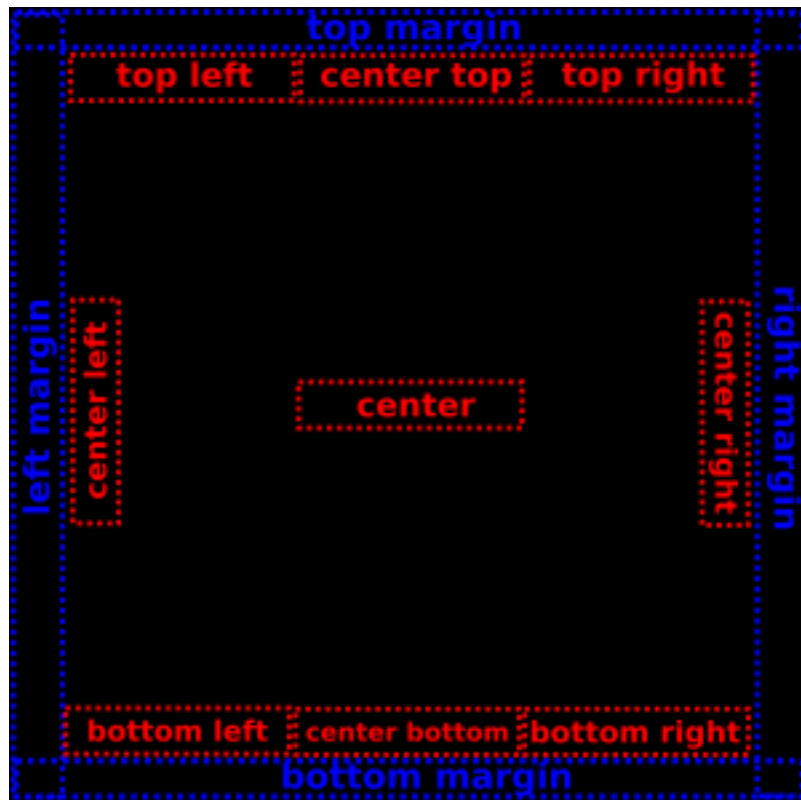
also shown in the figure (blue rectangles).

To keep the HUD margins and elements independent from the game, we are going to add the HUD information in the JSON file. Below is an example of JSON HUD data. It defines the HUD margins on all four borders (left, right, top, bottom) and the HUD elements. Each HUD element must specify: its prefab type, which will be used to instantiate the correct prefab (the prefabs in this example will be implemented later in this tutorial); the region that the element belongs; object properties, including the texture, group and custom properties specific to each prefab.

```
1 "hud": {
2         "margins": {"left": 20, "right": 50, "top": 20,
"bottom": 30},
3         "elements": {
4             "health": {
5                 "type": "show_stat_with_bar",
6                 "region": "top_left",
7                 "properties": {
```

```
8                       "texture": "healthbar_image",
9                       "group": "hud",
10                      "stat_to_show": "hero.health"
11                  }
12              },
13          "attack": {
14              "type": "show_stat_with_sprite",
15              "region": "center_bottom",
16              "properties": {
17                  "texture": "weapon_image",
18                  "group": "hud",
19                  "scale": {"x": 2, "y": 2},
20                  "anchor": {"x": 0.5, "y": 0.5},
21                  "stat_to_show": "hero.attack",
22                  "stats_spacing": {"x": 30, "y": 0},
23                  "stats_group": "stats"
24              }
25          },
25          "defense": {
26              "type": "show_stat_with_sprite",
28              "region": "center_bottom",
29              "properties": {
30                  "texture": "shield_image",
31                  "group": "hud",
32                  "scale": {"x": 2, "y": 2},
33                  "anchor": {"x": 0.5, "y": 0.5},
34                  "stat_to_show": "hero.defense",
35                  "stats_spacing": {"x": 30, "y": 0},
36                  "stats_group": "stats"
37              }
38          },
39          "money": {
40              "type": "show_stat_with_text",
41              "region": "top_right",
42              "properties": {
43                  "texture": "coin_image",
44                  "group": "hud",
45                  "scale": {"x": 2, "y": 2},
46                  "stat_to_show": "hero.money",
47                  "stats_group": "stats",
48                  "text_style": {
49                      "font": "32px Arial",
50                      "fill": "#FFFFFF"
51                  }
52              }
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
53                    }
54                 }
55        }
```

Given the JSON data, the HUD plugin code is shown below. In the "init" method it saves its properties and defines the regions begin and end coordinates, as well as assigns an empty "elements" array for each region. In the end, it calls the "create_elements" method to instantiate all HUD elements.

The "create_elements" method iterates through all elements creating them and adding them to the correct region. The elements are created using the "create_prefab" method from WorldState. That's why we need to define the element prefab type and properties, as we did in the JSON data. Since we don't know beforehand how many elements will be in each region, we start by creating all of them in the beginning of the region, and in the end update the elements positions in the "update_elements_positions" method.

The "update_elements_positions" method receives as a parameter a region and updates its elements positions according to the number of elements in the region. The strategy we are going to use is the following:

- If there is only one element in the region, it will be placed in the center of the region.
- If there are two elements in the region, the first one will be placed in the beginning and the second one in the end of the region.
- If there are more than two elements in the region, they will be placed equally spaced along the region.

The first two cases are easy to handle, as the code below shows. To deal with the third case, we calculate a step, which will be the space between every pair of elements. This step is given by the region dimensions divided by the number of elements. Then, it iterates through all the elements in the region, updating its positions and increasing the position by the calculated step. After all elements positions have been updated, they must be fixed to the camera, since we don't want them to move as the game screen moves.

```
1 var Phaser = Phaser || {};
2 var HUDExample = HUDExample || {};
3
4 HUDExample.HUD = function (game, parent) {
5     "use strict";
6     Phaser.Plugin.call(this, game, parent);
7 };
8
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Page 435

```
9 HUDExample.HUD.prototype =
Object.create(Phaser.Plugin.prototype);
10 HUDExample.HUD.prototype.constructor = HUDExample.HUD;
11
12 HUDExample.HUD.prototype.init = function (game_state,
hud_data) {
13     "use strict";
14     var camera_width, camera_height, camera_center;
15     this.game_state = game_state;
16     this.margins = hud_data.margins;
17     camera_width = this.game_state.game.camera.width;
18     camera_height = this.game_state.game.camera.height;
19     camera_center = new Phaser.Point(camera_width / 2,
camera_height / 2);
20     // define the HUD regions (begin and end points)
21     this.regions = {
22         top_left: {
23             begin: {x: this.margins.left, y:
this.margins.top},
24             end: {x: (camera_width / 3) - this.margins.right,
y: this.margins.top},
25             elements: []
26         },
27         center_top: {
28             begin: {x: (camera_width / 3) + this.margins.left,
y: this.margins.top},
29             end: {x: (2 * camera_width / 3) -
this.margins.right, y: this.margins.top},
30             elements: []
31         },
32         top_right: {
33             begin: {x: (2 * camera_width / 3) +
this.margins.left, y: this.margins.top},
34             end: {x: camera_width - this.margins.right, y:
this.margins.top},
35             elements: []
36         },
37         center_right: {
38             begin: {x: camera_width - this.margins.right, y:
(camera_height / 3) + this.margins.top},
39             end: {x: camera_width - this.margins.right, y: (2
* camera_height / 3) + this.margins.top},
40             elements: []
41         },
42         bottom_right: {
```

```
43              begin: {x: (2 * camera_width / 3) +
this.margins.left, y: camera_height - this.margins.bottom},
44              end: {x: camera_width - this.margins.right, y:
camera_height - this.margins.bottom},
45              elements: []
46          },
47          center_bottom: {
48              begin: {x: (camera_width / 3) + this.margins.left,
y: camera_height - this.margins.bottom},
49              end: {x: (2 * camera_width / 3) -
this.margins.right, y: camera_height - this.margins.bottom},
50              elements: []
51          },
52          bottom_left: {
53              begin: {x: this.margins.left, y: camera_height -
this.margins.bottom},
54              end: {x: (camera_width / 3) - this.margins.right,
y: camera_height - this.margins.bottom},
55              elements: []
56          },
57          center_left: {
58              begin: {x: this.margins.left, y: (camera_height /
3) + this.margins.top},
59              end: {x: this.margins.left, y: (2 * camera_height
/ 3) - this.margins.bottom},
60              elements: []
61          },
62          center: {
63              begin: {x: (camera_width / 3) + this.margins.left,
y: camera_center.y},
64              end: {x: (2 * camera_width / 3) -
this.margins.right, y: camera_center.y},
65              elements: []
66          }
67      };
68
69      // create the HUD elements
70      this.create_elements(hud_data.elements);
71  };
72
73  HUDExample.HUD.prototype.create_elements = function
(elements) {
74      "use strict";
75      var prefab_name, prefab_parameters, prefab_position,
region, prefab, region_name;
```

```
76      // create the HUD elements from the JSON file
77      for (prefab_name in elements) {
78          if (elements.hasOwnProperty(prefab_name)) {
79              prefab_parameters = elements[prefab_name];
80              // find the region beginning positions
81              region = this.regions[prefab_parameters.region];
82              prefab_position = new Phaser.Point(region.begin.x,
region.begin.y);
83              // create the element prefab in the beginning of
the region
84              prefab =
this.game_state.create_prefab(prefab_parameters.type,
prefab_name, prefab_position, prefab_parameters.properties);
85              // add the element to its correspondent region
86              region.elements.push(prefab);
87          }
88      }
89
90      // update the elements position according to the number of
elements in each region
91      for (region_name in this.regions) {
92          if (this.regions.hasOwnProperty(region_name)) {
93              this.update_elements_positions(this.regions[region
_name]);
94          }
95      }
96  };
97
98  HUDExample.HUD.prototype.update_elements_positions = function
(region) {
99      "use strict";
100     var region_dimensions, number_of_elements, step,
position;
101     region_dimensions = new Phaser.Point(region.end.x -
region.begin.x, region.end.y - region.begin.y);
102     number_of_elements = region.elements.length;
103     if (number_of_elements === 1) {
104         // if there is only one element, it should be in the
center of the region
105         region.elements[0].reset(region.begin.x +
(region_dimensions.x / 2), region.begin.y + (region_dimensions.y
/ 2));
106     } else if (number_of_elements === 2) {
107         // if there are two elements, they will be in
opposite sides of the region
```

```
108          region.elements[0].reset(region.begin.x,
region.begin.y);
109          region.elements[1].reset(region.end.x, region.end.y);
110      } else if (number_of_elements > 2) {
111          // if there are more than two elements, they will be
equally spaced in the region
112          step = new Phaser.Point(region_dimensions.x /
number_of_elements, region_dimensions.y / number_of_elements);
113          position = new Phaser.Point(region.begin.x,
region.begin.y);
114          region.elements.forEach(function (element) {
115              element.reset(position.x, position.y);
116              position.x += step.x;
117              position.y += step.y;
118          }, this);
119      }
120
121      // fix all elements to camera
122      region.elements.forEach(function (element) {
123          element.fixedToCamera = true;
124      }, this);
125 };
```

Creating the Tiled level

Since the focus of this tutorial is on the HUD plugin, I will not go into details on how to create the map using Tiled. If you're interested on learning more about Tiled to create your own map, I suggest you read one of my tutorials that explains it with more details. Otherwise, you can use the map I created, provided in the source code.

The figure below shows the map I'm going to use in this tutorial. If you're going to create your own map the only things you must be careful are: any collidable layer must have a collision property as true (since it was used in WorldState); all prefab properties
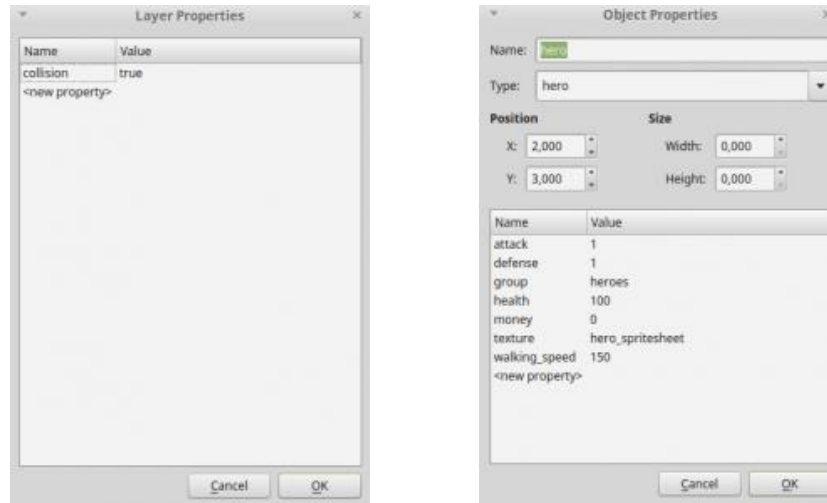
(including the prefab texture and group) must be defined as object properties, as shown
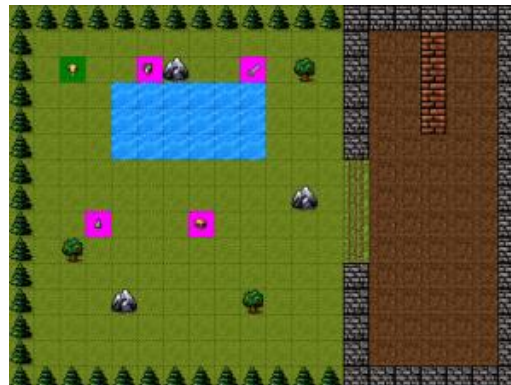
below.

Hero and Item prefabs

Before creating the HUD elements, we are going to create the Hero and Item prefabs, so the hero can walks in the level and collect items.

The Hero prefab code is shown below. It will have as parameters the walking speed and the initial stats. The constructor also enables the hero physical body and creates its walking animation.

In the "update" method we use the keyboard arrow keys to move the hero. Notice that the hero can only move to a given direction if it is not already moving to the opposite direction. Also, since we use the same walking animation for all directions, we have to change the sprite scale when changing from left to right directions.

```
1 var HUDExample = HUDExample || {};
2
3 HUDExample.Hero = function (game_state, name, position,
properties) {
4     "use strict";
5     HUDExample.Prefab.call(this, game_state, name, position,
properties);
6
7     this.anchor.setTo(0.5);
8
9     this.walking_speed = +properties.walking_speed;
10
11     this.stats = {
12         health: +properties.health,
13         defense: +properties.defense,
14         attack: +properties.attack,
15         money: +properties.money
16     };
17
18     this.game_state.game.physics.arcade.enable(this);
19     this.body.collideWorldBounds = true;
20
21     this.animations.add("walking", [0, 1], 6, true);
22
23     this.cursors =
this.game_state.game.input.keyboard.createCursorKeys();
24 };
25
26 HUDExample.Hero.prototype =
Object.create(HUDExample.Prefab.prototype);
27 HUDExample.Hero.prototype.constructor = HUDExample.Hero;
28
29 HUDExample.Hero.prototype.update = function () {
30     "use strict";
31     this.game_state.game.physics.arcade.collide(this,
this.game_state.layers.collision);
32
33     if (this.cursors.left.isDown && this.body.velocity.x <= 0)
{ // move left if is not already moving right
34         this.body.velocity.x = -this.walking_speed;
35         this.scale.setTo(1, 1);
36     } else if (this.cursors.right.isDown &&
this.body.velocity.x >= 0) { // move right if is not already
moving left
```

```
37          this.body.velocity.x = +this.walking_speed;
38          this.scale.setTo(-1, 1);
39      } else {
40          this.body.velocity.x = 0;
41      }
42
43      if (this.cursors.up.isDown && this.body.velocity.y <= 0) {
// move up if is not already moving down
44          this.body.velocity.y = -this.walking_speed;
45      } else if (this.cursors.down.isDown &&
this.body.velocity.y >= 0) { // move down if is not already
moving up
46          this.body.velocity.y = +this.walking_speed;
47      } else {
48          this.body.velocity.y = 0;
49      }
50
51      if (this.body.velocity.x === 0 && this.body.velocity.y ===
0) {
52          // if not moving, stop the animation
53          this.animations.stop();
54          this.frame = 0;
55      } else {
56          // if it is moving, play walking animation
57          this.animations.play("walking");
58      }
59 };
```

The Item prefab is shown below. We want to make the item collectible and allow it to change the hero stats when collected. For this, the constructor saves the stats this item increases and initializes its physical body. Then, the "update" method checks for collision with the hero and call the "collect_item" method. The "collect_item" method, by its turn checks what stats this item increases and update them on the hero accordingly. In the end, the item is killed.

```
1 var HUDExample = HUDExample || {};
2
3 HUDExample.Item = function (game_state, name, position,
properties) {
4     "use strict";
5     HUDExample.Prefab.call(this, game_state, name, position,
properties);
6
7     this.anchor.setTo(0.5);
```

```
8
9    this.stats = {
10        health: +properties.health,
11        defense: +properties.defense,
12        attack: +properties.attack,
13        money: +properties.money
14    };
15
16    this.game_state.game.physics.arcade.enable(this);
17    this.body.immovable = true;
18 };
19
20 HUDExample.Item.prototype =
Object.create(HUDExample.Prefab.prototype);
21 HUDExample.Item.prototype.constructor = HUDExample.Item;
22
23 HUDExample.Item.prototype.update = function () {
24    "use strict";
25    // when colliding with hero the item is collected
26    this.game_state.game.physics.arcade.collide(this,
this.game_state.groups.heroes, this.collect_item, null, this);
27 };
28
29 HUDExample.Item.prototype.collect_item = function (item,
hero) {
30    "use strict";
31    var stat;
32    // update hero stats according to item
33    for (stat in this.stats) {
34        // update only if the stat is defined for this item
35        if (this.stats.hasOwnProperty(stat) &&
this.stats[stat]) {
36            hero.stats[stat] += this.stats[stat];
37        }
38    }
39    this.kill();
40 };
```

By now you can already try running the game without the HUD, to see if the player is moving and collecting items correctly.

HUD elements

Now we can finally create the HUD elements. We will focus this tutorial in creating HUD elements that show the hero stats. For this, we will use a generic ShowStat prefab as shown below, which will be extended by the other HUD elements. This prefab saves the prefab and stat it is going to show in the constructor (for example, if we want to show the hero defense, this property will store "hero.defense"). Then, the "update" method checks if the stat has changed since the last update. If so, it calls the "update_stat" method to keep it updated in the ShowStat prefab.

```
1 var HUDExample = HUDExample || {};
2
3 HUDExample.ShowStat = function (game_state, name, position,
properties) {
4     "use strict";
5     HUDExample.Prefab.call(this, game_state, name, position,
properties);
6
7     this.stat_to_show = properties.stat_to_show;
8 };
9
10 HUDExample.ShowStat.prototype =
Object.create(HUDExample.Prefab.prototype);
11 HUDExample.ShowStat.prototype.constructor =
HUDExample.ShowStat;
```

<u>Zenva Academy</u> – Online courses on Phaser and game programming
<u>Zenva for Schools</u> – Coding courses for high schools

Page 444

```
12
13 HUDExample.ShowStat.prototype.update = function () {
14     "use strict";
15     var prefab_name, stat_name, new_stat;
16     prefab_name = this.stat_to_show.split(".")[0];
17     stat_name = this.stat_to_show.split(".")[1];
18     new_stat =
this.game_state.prefabs[prefab_name].stats[stat_name];
19     // check if the stat has changed
20     if (this.stat !== new_stat) {
21         // update the stat with the new value
22         this.update_stat(new_stat);
23     }
24 };
25
26 HUDExample.ShowStat.prototype.update_stat = function
(new_stat) {
27     "use strict";
28     this.stat = new_stat;
29 };
```

Now that we have the ShowStat prefab, we are going to create the following HUD elements, which will extend it:

- ShowStatWithText: prefab that will show the value of a stat using a text
- ShowStatWithBar: prefab that will show the value of a stat with a bar (like a health bar)
- ShowStatWithSprite: prefab that will show the value of a stat with sprites

Before going into the code, there is something important to mention. Remember that in the HUD plugin we first add all elements in the beginning of the region and then reset them to the correct position. Because of that, we can't use the HUD element position before it is reset because it will be incorrect. That's why in the following prefabs we use it on the "reset" method, as you will see.

The ShowStatWithText prefab is shown below. In the "reset" method (when the position is already correct) it creates the text that will show the stat value. Then, in the "update_stat" method it updates the text to show the next stat value.

```
1 var Engine = Engine || {};
2 var HUDExample = HUDExample || {};
3
4 HUDExample.ShowStatWithText = function (game_state, name,
position, properties) {
```

```
5    "use strict";
6    HUDExample.ShowStat.call(this, game_state, name, position,
properties);
7    this.text_style = properties.text_style;
8    this.stats_group = properties.stats_group;
9  };
10
11 HUDExample.ShowStatWithText.prototype =
Object.create(HUDExample.ShowStat.prototype);
12 HUDExample.ShowStatWithText.prototype.constructor =
HUDExample.ShowStatWithText;
13
14 HUDExample.ShowStatWithText.prototype.reset = function
(position_x, position_y) {
15    "use strict";
16    Phaser.Sprite.prototype.reset.call(this, position_x,
position_y);
17    // create the text to show the stat value
18    this.text = new Phaser.Text(this.game_state.game, this.x +
this.width, this.y, "", this.text_style);
19    this.text.fixedToCamera = true;
20    this.game_state.groups[this.stats_group].add(this.text);
21 };
22
23 HUDExample.ShowStatWithText.prototype.update_stat = function
(new_stat) {
24    "use strict";
25    HUDExample.ShowStat.prototype.update_stat.call(this,
new_stat);
25    // update the text to show the new stat value
27    this.text.text = this.stat;
28 };
```

The ShowStatWithBar is also simple. The only method we have to implement is "update_stat", which will change the prefab scale according to the stat value, making the bar bigger when the stat is greater.

```
1 var Engine = Engine || {};
2 var HUDExample = HUDExample || {};
3
4 HUDExample.ShowStatWithBar = function (game_state, name,
position, properties) {
5    "use strict";
```

```
6      HUDExample.ShowStat.call(this, game_state, name, position,
properties);
7 };
8
9 HUDExample.ShowStatWithBar.prototype =
Object.create(HUDExample.ShowStat.prototype);
10 HUDExample.ShowStatWithBar.prototype.constructor =
HUDExample.ShowStatWithBar;
11
12 HUDExample.ShowStatWithBar.prototype.update_stat = function
(new_stat) {
13     "use strict";
14     HUDExample.ShowStat.prototype.update_stat.call(this,
new_stat);
15     // use the stat to define the bar size
16     this.scale.setTo(this.stat, 2);
17 };
```

The ShowStatWithSprite prefab is a little bit more complex. In the "reset" method it shows the initial stats by calling the "show_initial_stats" method. This method creates a sprite for each value of the stat and add it to an array.

The "create_new_sprite" method is responsible for creating each sprite. First, it calculates the position of the next sprite from the number of sprites that already exist. Notice that the stat sprites use the same texture as the ShowStatWithSprite prefab, so all of them have the same width and height. After finding the position, it checks if there is a dead sprite in the stats group. If so, it reuses it by only resetting it to the desired position. Otherwise, it creates a new one.

Finally, the "update_stat" method will be different from the other two HUD elements. First, it checks if the new stat is lower or higher than the previous one. If it is higher, it must create new sprites (using the "create_new_sprite" method) until the new stat value is reached. Otherwise, it must kill the extra stats so as to show the correct stat value.

```
1 var Phaser = Phaser || {};
2 var Engine = Engine || {};
3 var HUDExample = HUDExample || {};
4
5 HUDExample.ShowStatWithSprite = function (game_state, name,
position, properties) {
6     "use strict";
7     HUDExample.ShowStat.call(this, game_state, name, position,
properties);
```

```
8     this.visible = false;
9     this.stats = [];
10     this.stats_spacing = properties.stats_spacing;
11     this.stats_group = properties.stats_group;
12     // it is necessary to save the initial position because we
need it to create the stat sprites
13     this.initial_position = new Phaser.Point(this.x, this.y);
14 };
15
16 HUDExample.ShowStatWithSprite.prototype =
Object.create(HUDExample.ShowStat.prototype);
17 HUDExample.ShowStatWithSprite.prototype.constructor =
HUDExample.ShowStatWithSprite;
18
19 HUDExample.ShowStatWithSprite.prototype.show_initial_stats =
function () {
20     "use strict";
21     var prefab_name, stat_name, initial_stat, stat_index,
stat;
22     // show initial stats
23     prefab_name = this.stat_to_show.split(".")[0];
24     stat_name = this.stat_to_show.split(".")[1];
25     initial_stat =
this.game_state.prefabs[prefab_name].stats[stat_name];
26     for (stat_index = 0; stat_index < initial_stat; stat_index
+= 1) {
27         // create new sprite to show stat
28         stat = this.create_new_stat_sprite();
29         this.stats.push(stat);
30     }
31     this.stat = initial_stat;
32 };
33
34 HUDExample.ShowStatWithSprite.prototype.reset = function
(position_x, position_y) {
35     "use strict";
36     Phaser.Sprite.prototype.reset.call(this, position_x,
position_y);
37     // it is necessary to save the initial position because we
need it to create the stat sprites
38     this.initial_position = new Phaser.Point(this.x, this.y);
39     this.show_initial_stats();
40     this.visible = false;
41 };
42
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
43 HUDExample.ShowStatWithSprite.prototype.update_stat =
function (new_stat) {
44     "use strict";
45     var stat_difference, stat_index, stat;
46     stat_difference = Math.abs(new_stat - this.stat);
47     if (new_stat > this.stat) {
48         // if the new stat is greater, we must create new stat
sprites
49         for (stat_index = 0; stat_index < stat_difference;
stat_index += 1) {
50             stat = this.create_new_stat_sprite();
51             this.stats.push(stat);
52         }
53     } else {
54         // if the new stat is lower, we must kill extra stat
sprites
55         for (stat_index = 0; stat_index < stat_difference;
stat_index += 1) {
56             stat = this.stats.pop();
57             stat.kill();
58         }
59     }
60     HUDExample.ShowStat.prototype.update_stat.call(this,
new_stat);
61 };
62
63 HUDExample.ShowStatWithSprite.prototype.create
_new_stat_sprite = function () {
64     "use strict";
65     var stat_position, stat, stat_property;
66     // calculate the next stat position
67     stat_position = new Phaser.Point(this.initial_position.x +
(this.stats.length * this.stats_spacing.x),
68                                     this.initial_positio
n.y + (this.stats.length * this.stats_spacing.y));
69     // get the first dead sprite in the stats group
70     stat =
this.game_state.groups[this.stats_group].getFirstDead();
71     if (stat) {
72         // if there is a dead stat, just reset it
73         stat.reset(stat_position.x, stat_position.y);
74     } else {
75         // if there are no dead stats, create a new one
76         // stat sprite uses the same texture as the
ShowStatWithSprite prefab
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
77          stat =
this.game_state.groups[this.stats_group].create(stat_position.x,
stat_position.y, this.texture);
78     }
79     // stat scale and anchor are the same as the prefab
80     stat.scale.setTo(this.scale.x, this.scale.y);
81     stat.anchor.setTo(this.anchor.x, this.anchor.y);
82     stat.fixedToCamera = true;
83     return stat;
84 };
```

Now, you can add the HUD elements to your HUD JSON data and see how it works for your game. Try different combinations and see which one looks better!

How to Use Phaser Signals to Save Game Statistics
By Renan Oliveira

Sometimes in a game you want to be aware of events that occur in your game the whole time, wether it would be to save game statistics or to build an achievement system. For example, your game may need to know when an enemy is killed to save the number of killed enemies, or because there is an achievement when the player kills a given number of enemies.

In this tutorial I will show how to use Phaser.Signal to listen to events in your game in order to save game statistics. We will implement it in a simple space shooter game. At the end, I will briefly explain how to extend this concept to create an achievement system. In order to read this tutorial it is important that you are familiar with the following concepts:

- Javascript and object-oriented concepts.
- Basic Phaser concepts, such as: states, sprites, groups and arcade physics.

Source code files

You can download the tutorial source code files here.

Game states
We're going to keep the game data in a JSON file as shown below. This file describes the game assets that must be loaded, the groups that must be created and the prefabs. We are going to create three game states to handle this JSON file: BootState, LoadingState and LevelState.

```json
1  {
2      "world": {
3          "origin_x": 0,
4          "origin_y": 0,
5          "width": 360,
6          "height": 640
7      },
8      "assets": {
9          "space_image": { "type": "image", "source": "assets/images/space.png" },
10         "ship_image": { "type": "image", "source": "assets/images/player.png" },
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
11          "bullet_image": { "type": "image", "source":
"assets/images/bullet.png" },
12          "enemy_spritesheet": { "type": "spritesheet",
"source": "assets/images/green_enemy.png", "frame_width": 50,
"frame_height": 46, "frames": 3, "margin": 1, "spacing": 1 }
13      },
14      "groups": [
15          "ships",
16          "player_bullets",
17          "enemies",
18          "enemy_bullets",
19          "enemy_spawners",
20          "hud"
21      ],
22      "prefabs": {
23          "ship": {
24              "type": "ship",
25              "position": {"x": 180, "y": 600},
26              "properties": {
27                  "texture": "ship_image",
28                  "group": "ships",
29                  "velocity": 200,
30                  "shoot_rate": 5,
31                  "bullet_velocity": 500
32              }
33          },
34          "enemy_spawner": {
35              "type": "enemy_spawner",
36              "position": {"x": 0, "y": 100},
37              "properties": {
38                  "texture": "",
39                  "group": "enemy_spawners",
40                  "spawn_interval": 1,
41                  "enemy_properties": {
42                      "texture": "enemy_spritesheet",
43                      "group": "enemies",
44                      "velocity": 50,
45                      "shoot_rate": 2,
46                      "bullet_velocity": 300
47                  }
48              }
49          }
50      }
51 }
```

BootState and LoadingState codes are shown below. The first one simply loads this JSON file and calls LoadingState with the level data. LoadingState, by its turn, loads all game assets calling the correct Phaser method according to the asset type (for example, calling "this.load.image" to load an image).

```
1 var SignalExample = SignalExample || {};
2
3 SignalExample.BootState = function () {
4    "use strict";
5    Phaser.State.call(this);
6 };
7
8 SignalExample.BootState.prototype =
Object.create(Phaser.State.prototype);
9 SignalExample.BootState.prototype.constructor =
SignalExample.BootState;
10
11 SignalExample.BootState.prototype.init = function
(level_file, next_state, extra_parameters) {
12    "use strict";
13    this.level_file = level_file;
14    this.next_state = next_state;
15    this.extra_parameters = extra_parameters;
16 };
17
18 SignalExample.BootState.prototype.preload = function () {
19    "use strict";
20    this.load.text("level1", this.level_file);
21 };
22
23 SignalExample.BootState.prototype.create = function () {
24    "use strict";
25    var level_text, level_data;
26    level_text = this.game.cache.getText("level1");
27    level_data = JSON.parse(level_text);
28    this.game.state.start("LoadingState", true, false,
level_data, this.next_state, this.extra_parameters);
29 };
```

```
1 var SignalExample = SignalExample || {};
2
3 SignalExample.LoadingState = function () {
4    "use strict";
```

```
5      Phaser.State.call(this);
6 };
7
8 SignalExample.LoadingState.prototype =
Object.create(Phaser.State.prototype);
9 SignalExample.LoadingState.prototype.constructor =
SignalExample.LoadingState;
10
11 SignalExample.LoadingState.prototype.init = function
(level_data, next_state, extra_parameters) {
12     "use strict";
13     this.level_data = level_data;
14     this.next_state = next_state;
15     this.extra_parameters = extra_parameters;
16 };
17
18 SignalExample.LoadingState.prototype.preload = function () {
19     "use strict";
20     var assets, asset_loader, asset_key, asset;
21     assets = this.level_data.assets;
22     for (asset_key in assets) { // load assets according to
asset key
23         if (assets.hasOwnProperty(asset_key)) {
24             asset = assets[asset_key];
25             switch (asset.type) {
26             case "image":
27                 this.load.image(asset_key, asset.source);
28                 break;
29             case "spritesheet":
30                 this.load.spritesheet(asset_key, asset.source,
asset.frame_width, asset.frame_height, asset.frames,
asset.margin, asset.spacing);
31                 break;
32             case "tilemap":
33                 this.load.tilemap(asset_key, asset.source,
null, Phaser.Tilemap.TILED_JSON);
34                 break;
35             }
36         }
37     }
38 };
39
40 SignalExample.LoadingState.prototype.create = function () {
41     "use strict";
```

```
42      this.game.state.start(this.next_state, true, false,
this.level_data, this.extra_parameters);
43 };
```

LevelState, by its turn, is responsible for creating the game groups and prefabs, as shown below. In the "create" method it starts by creating a Phaser.TileSprite to represent the background space. Then it creates all game groups from the JSON file. Finally, it creates all prefabs by iterating through all prefabs described in the JSON file calling the "create_prefab" method.

```
1 var SignalExample = SignalExample || {};
2
3 SignalExample.LevelState = function () {
4     "use strict";
5     Phaser.State.call(this);
6
7     this.prefab_classes = {
8         "ship": SignalExample.Ship.prototype.constructor,
9         "enemy_spawner":
SignalExample.EnemySpawner.prototype.constructor
10      };
11 };
12
13 SignalExample.LevelState.prototype =
Object.create(Phaser.State.prototype);
14 SignalExample.LevelState.prototype.constructor =
SignalExample.LevelState;
15
16 SignalExample.LevelState.prototype.init = function
(level_data) {
17     "use strict";
18     this.level_data = level_data;
19
20     this.scale.scaleMode = Phaser.ScaleManager.SHOW_ALL;
21     this.scale.pageAlignHorizontally = true;
22     this.scale.pageAlignVertically = true;
23
24     // start physics system
25     this.game.physics.startSystem(Phaser.Physics.ARCADE);
26     this.game.physics.arcade.gravity.y = 0;
27 };
28
29 SignalExample.LevelState.prototype.create = function () {
30     "use strict";
```

```
31    var group_name, prefab_name;
32
33    this.space = this.add.tileSprite(0, 0,
this.game.world.width, this.game.world.height, "space_image");
34
35    // create groups
36    this.groups = {};
37    this.level_data.groups.forEach(function (group_name) {
38        this.groups[group_name] = this.game.add.group();
39    }, this);
40
41    // create prefabs
42    this.prefabs = {};
43    for (prefab_name in this.level_data.prefabs) {
44        if
(this.level_data.prefabs.hasOwnProperty(prefab_name)) {
45            // create prefab
46            this.create_prefab(prefab_name,
this.level_data.prefabs[prefab_name]);
47        }
48    }
49 };
50
51 SignalExample.LevelState.prototype.create_prefab = function
(prefab_name, prefab_data) {
52    "use strict";
53    var prefab;
54    // create object according to its type
55    if (this.prefab_classes.hasOwnProperty(prefab_data.type))
{
56        prefab = new
this.prefab_classes[prefab_data.type](this, prefab_name,
prefab_data.position, prefab_data.properties);
57    }
58 };
```

The "create_prefab" method creates the correct prefab according to its type. Notice that this can be done because all prefabs have the same constructor, extended by the generic Prefab class shown below. Also, there is a "prefab_classes" property (in LevelState constructor) that maps each prefab type to its corresponding constructor.

```
1 var SignalExample = SignalExample || {};
2
```

```
3 SignalExample.Prefab = function (game_state, name, position,
properties) {
4     "use strict";
5     Phaser.Sprite.call(this, game_state.game, position.x,
position.y, properties.texture);
6
7     this.game_state = game_state;
8
9     this.name = name;
10
11     this.game_state.groups[properties.group].add(this);
12     this.frame = +properties.frame;
13
14     if (properties.scale) {
15         this.scale.setTo(properties.scale.x,
properties.scale.y);
16     }
17
18     if (properties.anchor) {
19         this.anchor.setTo(properties.anchor.x,
properties.anchor.y);
20     }
21
22     this.game_state.prefabs[name] = this;
23 };
24
25 SignalExample.Prefab.prototype =
Object.create(Phaser.Sprite.prototype);
26 SignalExample.Prefab.prototype.constructor =
SignalExample.Prefab;
```

The GameStats plugin

Now we are going to create a Phaser plugin to save game statistics by listening to phaser signals. To create a Phaser plugin we have to create a class that extends Phaser.Plugin, as shown below. The "init" method is called when the plugin is added to the game, and we use it to save all the plugin properties, such as the game stats position, text style and which signals the plugin will listen to. All this data will be loaded from the JSON level file and will be represented with the following structure:

```
1 "game_stats_data": {
2     "position": {"x": 180, "y": 300},
3     "text_style": {"font": "28pt Arial", "fill": "#FFFFFF"},
4     "game_stats": {
```

```
5          "shots_fired": {"text": "Shots fired: ", "value": 0},
6          "enemies_spawned": {"text": "Enemies spawned: ",
"value": 0}
7       },
8     "listeners": [
9          {"group": "ships", "signal": "onShoot", "stat_name":
"shots_fired", "value": 1},
10          {"group": "enemy_spawners", "signal": "onSpawn",
"stat_name": "enemies_spawned", "value": 1}
11       ]
12 }
```

The "listen_to_events" method is responsible for making the plugin to listen game events. It iterates through all listeners descriptions (saved in the "init" method) and creates the listeners for each one. Notice that each listener describes the group the plugin have to listen, then the plugin has to iterate through all sprites in that group to listen to each one separately. The callback of all events is the "save_stat" method, which receives as parameters the sprite that dispatched the event, the stat name and the value to be increased. This method simply increases the correct game stat with the given value. Notice that the "game_stats" object was already initialized in the "init" method with the initial values from "game_stats_data".

Finally, we are going to add a "show_stats" method, which will be called in the end of the game to show the final values. This method iterates through all game stats creating a Phaser.Text to each one that shows the final value. Notice that the initial position of the texts and their style were saved in the "init" method, while its final text is the text of the game stat and its final value.

```
1 var Phaser = Phaser || {};
2 var SignalExample = SignalExample || {};
3
4 SignalExample.GameStats = function (game, parent) {
5     "use strict";
6     Phaser.Plugin.call(this, game, parent);
7 };
8
9 SignalExample.GameStats.prototype =
Object.create(Phaser.Plugin.prototype);
10 SignalExample.GameStats.prototype.constructor =
SignalExample.GameStats;
11
12 SignalExample.GameStats.prototype.init = function
(game_state, game_stats_data) {
```

```
13      "use strict";
14      // save properties
15      this.game_state = game_state;
16      this.game_stats_position = game_stats_data.position;
17      this.game_stats_text_style = game_stats_data.text_style;
18      this.game_stats = game_stats_data.game_stats;
19      this.listeners = game_stats_data.listeners;
20 };
21
22 SignalExample.GameStats.prototype.listen_to_events = function
() {
23      "use strict";
24      this.listeners.forEach(function (listener) {
25          // iterate through the group that should be
listened
26          this.game_state.groups[listener.group].forEach(functio
n (sprite) {
27              // add a listener for each sprite in the group
28              sprite.events[listener.signal].add(this.save_stat,
this, 0, listener.stat_name, listener.value);
29          }, this);
30      }, this);
31 };
32
33 SignalExample.GameStats.prototype.save_stat = function
(sprite, stat_name, value) {
34      "use strict";
35      // increase the corresponding game stat
36      this.game_stats[stat_name].value += value;
37 };
38
39 SignalExample.GameStats.prototype.show_stats = function () {
40      "use strict";
41      var position, game_stat, game_stat_text;
42      position = new Phaser.Point(this.game_stats_position.x,
this.game_stats_position.y);
43      for (game_stat in this.game_stats) {
44          if (this.game_stats.hasOwnProperty(game_stat)) {
45              // create a Phaser text for each game stat showing
the final value
46              game_stat_text = new
Phaser.Text(this.game_state.game, position.x, position.y,
47                                          this.game_stats[ga
me_stat].text + this.game_stats[game_stat].value,
```

```
48                                              Object.create(this
.game_stats_text_style));
49              game_stat_text.anchor.setTo(0.5);
50              this.game_state.groups.hud.add(game_stat_text);
51              position.y += 50;
52          }
53      }
54 };
```

Now that we have the GameStats plugin created, we have to add it to our game. We do that by calling "this.game.plugins.add" in the end of LevelState "create" method. The parameters of this method are the plugin class, the game state and the game stats data, obtained from the JSON level file.

```
1 this.game_stats =
this.game.plugins.add(SignalExample.GameStats, this,
this.level_data.game_stats_data);
```

Game prefabs

To verify if our GameStats plugin is working correctly, we have to create the prefabs for our game. In this tutorial I will test this plugin in a simple space shooter game, but feel free to implement it in your own game.

The prefabs we are going to create are the Ship, Bullet, Enemy and EnemySpawner. All of them extends the Prefab class shown before.

The Ship prefab will allow the player to move the ship left and right, and will constantly shoot bullets at its enemies. The constructor initializes a timer that calls the "shoot" method according to the shoot rate. This method, by its turn, starts by checking if there is already a dead bullet that can be reused. If there is one, it just resets its position to the current ship position. Otherwise, it creates a new bullet.

The "update" method is responsible for moving the ship. The ship moves according to the mouse pointer. If the player clicks on the screen the ship moves towards the mouse cursor position. This is done by getting the "activePointer" position and setting the ship velocity according to it.

```
1 var SignalExample = SignalExample || {};
2
3 SignalExample.Ship = function (game_state, name, position,
properties) {
```

```
4    "use strict";
5    SignalExample.Prefab.call(this, game_state, name, position,
properties);
6
7    this.anchor.setTo(0.5);
8
9    this.game_state.game.physics.arcade.enable(this);
10    this.body.setSize(this.width * 0.3, this.height * 0.3);
11
12    this.velocity = properties.velocity;
13    this.bullet_velocity = properties.bullet_velocity;
14
15    // create and start shoot timer
16    this.shoot_timer = this.game_state.game.time.create();
17    this.shoot_timer.loop(Phaser.Timer.SECOND /
properties.shoot_rate, this.shoot, this);
18    this.shoot_timer.start();
19 };
20
21 SignalExample.Ship.prototype =
Object.create(Phaser.Sprite.prototype);
22 SignalExample.Ship.prototype.constructor =
SignalExample.Ship;
23
24 SignalExample.Ship.prototype.update = function () {
25    "use strict";
26    var target_x;
27    this.game_state.game.physics.arcade.overlap(this,
this.game_state.groups.enemy_bullets, this.kill, null, this);
28
29    this.body.velocity.x = 0;
30    // move the ship when the mouse is clicked
31    if (this.game_state.game.input.activePointer.isDown) {
32        // get the clicked position
33        target_x = this.game_state.game.input.activePointer.x;
34        if (target_x < this.x) {
35            // if the clicked position is left to the ship,
move left
36            this.body.velocity.x = -this.velocity;
37        } else if (target_x > this.x) {
38            // if the clicked position is right to the ship,
move right
39            this.body.velocity.x = this.velocity;
40        }
41    }
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Page 462

```
42 };
43
44 SignalExample.Ship.prototype.kill = function () {
45     "use strict";
46     Phaser.Sprite.prototype.kill.call(this);
47     // stop shoot timer
48     this.shoot_timer.stop();
49     // if the ship dies, it is game over
50     this.game_state.game_over();
51 };
52
53 SignalExample.Ship.prototype.shoot = function () {
54     "use strict";
55     var bullet, bullet_position, bullet_name,
bullet_properties;
56     // check if there is a dead bullet to reuse
57     bullet =
this.game_state.groups.player_bullets.getFirstDead();
58     bullet_position = new Phaser.Point(this.x, this.y);
59     if (bullet) {
60         // if there is a dead bullet reset it to the current
position
61         bullet.reset(bullet_position.x, bullet_position.y);
62     } else {
63         // if there is no dead bullet, create a new one
64         bullet_name = this.name + "_bullet" +
this.game_state.groups.player_bullets.countLiving();
65         bullet_properties = {texture: "bullet_image", group:
"player_bullets", direction: -1, velocity:
this.bullet_velocity};
66         bullet = new SignalExample.Bullet(this.game_state,
bullet_name, bullet_position, bullet_properties);
67     }
68 };
```

Also, in the update method we have to check when the ship overlaps with enemy bullets and, if so, kill it. When the ship is killed, the game should end, which is done in the "kill" method. The "game_over" method is shown below, and it simply show the final game statistics from the GameStats plugin.

```
1 SignalExample.LevelState.prototype.game_over = function () {
2     "use strict";
3     //this.game.state.start("BootState", true, false,
"assets/levels/level1.json", "LevelState");
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
4      this.game_stats.show_stats();
5  };
```

```
1  var SignalExample = SignalExample || {};
2
3  SignalExample.Bullet = function (game_state, name, position,
properties) {
4      "use strict";
5      SignalExample.Prefab.call(this, game_state, name, position,
properties);
6
7      this.anchor.setTo(0.5);
8
9      this.game_state.game.physics.arcade.enable(this);
10
11     this.body.velocity.y = properties.direction *
properties.velocity;
12 };
13
14 SignalExample.Bullet.prototype =
Object.create(Phaser.Sprite.prototype);
15 SignalExample.Bullet.prototype.constructor =
SignalExample.Bullet;
```

The Enemy prefab is similar to the Ship, where the main difference is that it only moves vertically in the screen. It sets its velocity in the constructor and creates the shoot timer, like the Ship. Its "shoot" method is almost the same as the Ship one, except the bullets are from the "enemy_bullets" group, and they have different direction and velocity. In the "update" method it only has to check for overlaps with player bullets.

```
1  var SignalExample = SignalExample || {};
2
3  SignalExample.Enemy = function (game_state, name, position,
properties) {
4      "use strict";
5      SignalExample.Prefab.call(this, game_state, name, position,
properties);
6
7      this.anchor.setTo(0.5);
8
9      this.game_state.game.physics.arcade.enable(this);
10
11     this.velocity = properties.velocity;
12     this.body.velocity.y = this.velocity;
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
13
14     this.checkWorldBounds = true;
15     this.outOfBoundsKill = true;
16
17     this.bullet_velocity = properties.bullet_velocity;
18
19     // create and start shoot timer
20     this.shoot_timer = this.game_state.game.time.create();
21     this.shoot_timer.loop(Phaser.Timer.SECOND /
properties.shoot_rate, this.shoot, this);
22     this.shoot_timer.start();
23 };
24
25 SignalExample.Enemy.prototype =
Object.create(Phaser.Sprite.prototype);
26 SignalExample.Enemy.prototype.constructor =
SignalExample.Enemy;
27
28 SignalExample.Enemy.prototype.update = function () {
29     "use strict";
30     // die if touches player bullets
31     this.game_state.game.physics.arcade.overlap(this,
this.game_state.groups.player_bullets, this.kill, null, this);
32 };
33
34 SignalExample.Enemy.prototype.kill = function () {
35     "use strict";
36     Phaser.Sprite.prototype.kill.call(this);
37     // stop shooting
38     this.shoot_timer.pause();
39 };
40
41 SignalExample.Enemy.prototype.reset = function (x, y) {
42     "use strict";
43     Phaser.Sprite.prototype.reset.call(this, x, y);
44     this.body.velocity.y = this.velocity;
45     this.shoot_timer.resume();
46 };
47
48 SignalExample.Enemy.prototype.shoot = function () {
49     "use strict";
50     var bullet, bullet_position, bullet_name,
bullet_properties;
51     // check if there is a dead bullet to reuse
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
52     bullet =
this.game_state.groups.enemy_bullets.getFirstDead();
53     bullet_position = new Phaser.Point(this.x, this.y);
54     if (bullet) {
55         // if there is a dead bullet reset it to the current
position
56         bullet.reset(bullet_position.x, bullet_position.y);
57     } else {
58         // if there is no dead bullet, create a new one
59         bullet_name = this.name + "_bullet" +
this.game_state.groups.enemy_bullets.countLiving();
60         bullet_properties = {texture: "bullet_image", group:
"enemy_bullets", direction: 1, velocity: this.bullet_velocity};
61         bullet = new SignalExample.Bullet(this.game_state,
bullet_name, bullet_position, bullet_properties);
62     }
63 };
```

Notice that in the Enemy prefab "kill" method we have to pause the shoot timer, and in the "reset" method we have to resume it and set the enemy velocity again. This happens because in our game enemies will be constantly spawning, and we want to reuse dead enemies, so we have to properly handle timers and the physical body when they are reused.

Finally, the EnemySpawner is shown below. In the constructor it starts a loop event that calls the "spawn" method. This method is responsible for creating enemies using the same strategy we are using to create the bullets: first, it checks if there is a dead enemy to reuse and if so, resets it to the desired position. Otherwise, it creates a new enemy. The main difference here is that the position is random between 10% and 90% of the game world width. We generate this random position using Phaser RandomDataGenerator (you can check the documentation if you are not familiar with it).

```
1 var SignalExample = SignalExample || {};
2
3 SignalExample.EnemySpawner = function (game_state, name,
position, properties) {
4     "use strict";
5     SignalExample.Prefab.call(this, game_state, name, position,
properties);
6
7     this.enemy_properties = properties.enemy_properties;
8
9     // start spawning event
```

```
10     this.game_state.game.time.events.loop(Phaser.Timer.SECOND
* properties.spawn_interval, this.spawn, this);
11 };
12
13 SignalExample.EnemySpawner.prototype =
Object.create(Phaser.Sprite.prototype);
14 SignalExample.EnemySpawner.prototype.constructor =
SignalExample.EnemySpawner;
15
16 SignalExample.EnemySpawner.prototype.spawn = function () {
17     "use strict";
18     var enemy, enemy_position, enemy_name, enemy_properties;
19     // check if there is a dead enemy to reuse
20     enemy = this.game_state.groups.enemies.getFirstDead();
21     // spawn enemy in a random position inside the world
22     enemy_position = new
Phaser.Point(this.game_state.game.rnd.between(0.1 *
this.game_state.game.world.width, 0.9 *
this.game_state.game.world.width), this.y);
23     if (enemy) {
24         // if there is a dead enemy reset it to the current
position
25         enemy.reset(enemy_position.x, enemy_position.y);
26     } else {
27         // if there is no dead enemy, create a new one
28         enemy_name = this.name + "_enemy" +
this.game_state.groups.enemies.countLiving();
29         enemy = new SignalExample.Enemy(this.game_state,
enemy_name, enemy_position, this.enemy_properties);
30     }
31 };
```

By now, you can try playing the game without the events, so it will work but the game statistics won't be saved. This way you can check if everything is working before moving
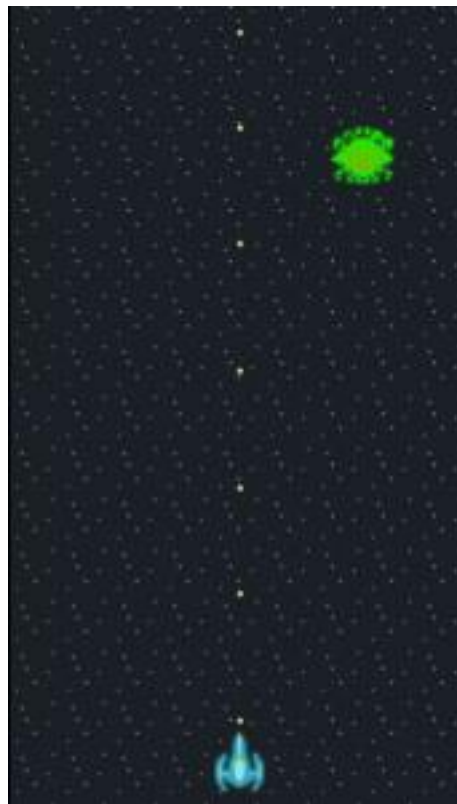
on to add the game signals.

Adding the signals

We are going to create two signals to save game statistics: onShoot, dispatched when the ship shoots, and onSpawn, dispatched when the EnemySpawner spawns an enemy.

The onShoot signal is created in the Ship constructor, and is dispatched in the "shoot" method, as shown below. The onSpawn signal, by its turn is created in the constructor of EnemySpawner, and dispatched in the "spawn" method. Notice that both signals have to send the sprite as a parameter dispatching the event, since the GameStats plugin is expecting it.

```
1 SignalExample.Ship = function (game_state, name, position,
properties) {
2     "use strict";
```

```
3     SignalExample.Prefab.call(this, game_state, name, position,
properties);
4
5     this.anchor.setTo(0.5);
6
7     this.game_state.game.physics.arcade.enable(this);
8     this.body.setSize(this.width * 0.3, this.height * 0.3);
9
10     this.velocity = properties.velocity;
11     this.bullet_velocity = properties.bullet_velocity;
12
13     // create and start shoot timer
14     this.shoot_timer = this.game_state.game.time.create();
15     this.shoot_timer.loop(Phaser.Timer.SECOND /
properties.shoot_rate, this.shoot, this);
16     this.shoot_timer.start();
17
18     // creating shooting event to be listened
19     this.events.onShoot = new Phaser.Signal();
20 };
21
22 SignalExample.Ship.prototype.shoot = function () {
23     "use strict";
24     var bullet, bullet_position, bullet_name,
bullet_properties;
25     // check if there is a dead bullet to reuse
26     bullet =
this.game_state.groups.player_bullets.getFirstDead();
27     bullet_position = new Phaser.Point(this.x, this.y);
28     if (bullet) {
29         // if there is a dead bullet reset it to the current
position
30         bullet.reset(bullet_position.x, bullet_position.y);
31     } else {
32         // if there is no dead bullet, create a new one
33         bullet_name = this.name + "_bullet" +
this.game_state.groups.player_bullets.countLiving();
34         bullet_properties = {texture: "bullet_image", group:
"player_bullets", direction: -1, velocity:
this.bullet_velocity};
35         bullet = new SignalExample.Bullet(this.game_state,
bullet_name, bullet_position, bullet_properties);
36     }
37
38     // dispatch shoot event
```

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

```
39    this.events.onShoot.dispatch(this);
40 };


1 SignalExample.EnemySpawner = function (game_state, name,
position, properties) {
2     "use strict";
3     SignalExample.Prefab.call(this, game_state, name, position,
properties);
4
5     this.enemy_properties = properties.enemy_properties;
6
7     // start spawning event
8     this.game_state.game.time.events.loop(Phaser.Timer.SECOND *
properties.spawn_interval, this.spawn, this);
9
10    // create spawn event to be listened
11    this.events.onSpawn = new Phaser.Signal();
12 };
13
14 SignalExample.EnemySpawner.prototype.spawn = function () {
15    "use strict";
16    var enemy, enemy_position, enemy_name, enemy_properties;
17    // check if there is a dead enemy to reuse
18    enemy = this.game_state.groups.enemies.getFirstDead();
19    // spawn enemy in a random position inside the world
20    enemy_position = new
Phaser.Point(this.game_state.game.rnd.between(0.1 *
this.game_state.game.world.width, 0.9 *
this.game_state.game.world.width), this.y);
21    if (enemy) {
22        // if there is a dead enemy reset it to the current
position
23        enemy.reset(enemy_position.x, enemy_position.y);
24    } else {
25        // if there is no dead enemy, create a new one
26        enemy_name = this.name + "_enemy" +
this.game_state.groups.enemies.countLiving();
27        enemy = new SignalExample.Enemy(this.game_state,
enemy_name, enemy_position, this.enemy_properties);
28    }
29
30    // dispatch spawn event
31    this.events.onSpawn.dispatch(this);
32 };
```

Finally, we call the "listen_to_events" method from the plugin after adding it to LevelState, so it will add the listeners described in the JSON file (shown before), which will be called when the new signals are dispatched.

```
1 this.game_stats.listen_to_events();
```

By now, you can already try playing the game with the game statistics. Check if all

statistics are being correctly saved, and try adding different signals.

Creating an achievement system

Before finishing the tutorial, I would like to briefly show how easily you can change the GameStats plugin to an GameAchievements plugin. There are two main differences between those plugins:

1   For the GameAchievements plugin it would be interesting to have listeners to specific prefabs, not only groups. For example, the game might have an achievement when a given boss is defeated, so it should be listened to that specific boss.
2   The callback of the listeners in the GameAchievements plugin would be more complex than simply saving a game statistic, so it will probably be necessary to have different callbacks for different achievements.

Zenva Academy – Online courses on Phaser and game programming
Zenva for Schools – Coding courses for high schools

Apart from those two differences, the signals and listeners can be created the same way. Try creating achievements for your game, and see how they work!