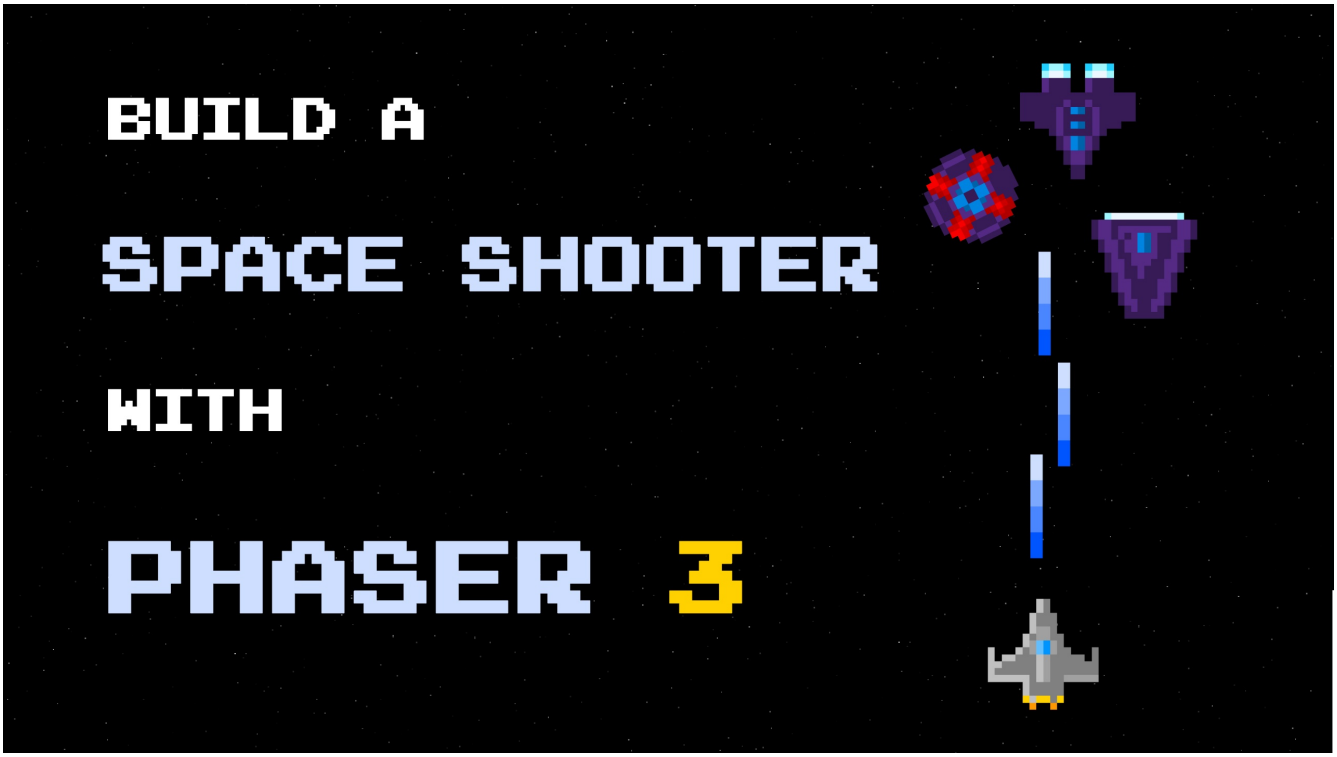


Build a Space Shooter with Phaser 3

v1.0

Written by Jared York

Introduction

With the latest release of Phaser 3 (as of now, v3.16.1), there has never been a better time to jump into Phaser than now! In this free course, we will be exploring creating a basic space shooter with Phaser 3. I will start by saying that it is quite beneficial to have the basics of JavaScript under your belt. However, no matter if you have never used Phaser before, or just dabbled in it for a little bit, this course is for you. By the end of this course, you should have adequate knowledge to start building games of your own with Phaser 3. You can check out Phaser at the official website.

Concepts you will learn about:

- Basics of JavaScript ES6 classes

- Drawing sprites
- Scaling and rotating sprites
- Playing sounds
- Changing scenes

Chapter 1

To begin, we will have to create the files and directories needed to build the game. After that we will create the main HTML file for our game, as well as adding the code to load our content.

Ensure a Web Server is Set Up

Even though Phaser games are ran in your browser, you unfortunately can't just run a local HTML file directly from your file system. When requesting files over http, the security of the server allows you to access only the files you're allowed to. When loading a file from the local file system (the file:// protocol), your browser highly restricts it for obvious security reasons. It's no good to allow code on a website to read anything in your raw file system. Because of this, we will need to host our game on a local web server.

We recommend checking out Phaser's official guide, "Getting Started with Phaser 3", to learn which web server is compatible with your system and there are links to each one. The guide also provides some detailed summaries on the each of the various web servers mentioned.

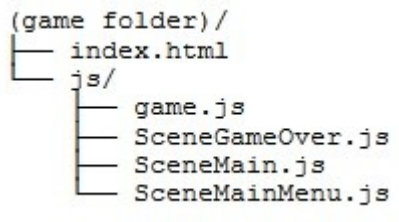
Create the Files and Folders Needed

First, find the location where your web server hosts files from (WAMP Server, for example, hosts files from the www directory within it's installation folder at C:/wamp64.) Once you have found the location, create a new folder inside it and call it anything you want.

Next, enter the folder and create a new file called, `index.html` Our index file is where we will declare the location of our Phaser script and the rest of our game scripts.

Now we will need to create two new folders, I called the first one content for our game content (sprites, audio, etc.), and the other one, js, which will contain our Phaser script and our other game scripts. Feel free to name these two folders anything you would like, after all, it is your game. One of the folders just needs to be dedicated to the content for our game, and the other for the JavaScript files. Since we have our folder for content and JavaScript, create three new files inside the newly created folder for JavaScript called: `SceneMainMenu.js`, `SceneMain.js`, `SceneGameOver.js`, and `game.js`. I will explain what those files will do shortly, but next we need to populate our content folder with the content for our game. After all, what's the point of a game if there's nothing to see? ☺

So far, the file structure we have created should look like:



Now to add content to our game, we will first need content. I have prepared some assets for this course that you can download for free here. Otherwise, you can create your own assets if you wish. Keep in mind that if you create your own assets, Phaser requires frames to be in a horizontal row for animation images. Below is a list of the content we will need:

Content needed:

Sprites (images)
sprBtnPlay.png (the play button)
sprBtnPlayHover.png (the play button when mouse is over)
sprBtnPlayDown.png (the play button when clicked)
sprBtnRestart.png (the restart button)
sprBtnRestartHover.png (the restart button on mouse over)
sprBtnRestartDown (the restart button when clicked)
sprBg0.png (a background layer of stars with transparency around the stars)
sprBg1.png (another background layer of stars with transparency around the stars)
sprEnemy0.png (the first enemy, this is an animation)
sprEnemy1.png (the second enemy, this is not an animation)
sprEnemy2.png (the third enemy, this is an animation)
sprLaserEnemy.png (laser shot by enemies)
sprLaserPlayer.png (laser shot by the player)
sprExplosion.png (an explosion animation)
sprPlayer.png (the player, this is an animation)
Audio (.wav files)
sndExplode0.wav (the first explosion sound)
sndExplode1.wav (the second explosion sound)
sndLaser.wav (the sound of a laser being shot)
sndBtnOver.wav (the sound of mouse moving over button)
sndBtnDown.wav (the sound of button when clicked)

Once you downloaded the assets (or made your own), we will move those files into the content directory we have made.

Finally, we need to download the latest Phaser script. One method of acquiring this (there are multiple), is to head over to GitHub (specifically here) and download the script meant for distribution. You will want either `phaser.js` or `phaser.min.js`. The file phaser.js contains the source code for Phaser in a readable form, which is useful if you wanted to contribute to Phaser, or to understand how something is implemented under-the-hood. The other file, `phaser.min.js` is meant for distribution, and is compressed to reduce file size. For our purposes, it won't really matter which one we download, so decide which and click the appropriate link. You will then be greeted by a page that displays a "View raw" button near the center of the page and roughly halfway down. Next, click the "View raw" link, right click anywhere on the page that appears, then click "Save Page As". A save dialog will appear where you should save the Phaser file in the JavaScript directory we created earlier.

With that, we will wrap up the first chapter of our free course, "Build a Space Shooter with Phaser 3".

Chapter 2

In the last chapter of this course, we finished creating the files and folders we need for our game. The next thing we need to do is start building out our `index.html`. In order to start coding our HTML and JavaScript files, you will need a text editor. If you have not programmed yet, any text editor will work such as Notepad, Visual Studio Code, Atom, etc. I personally use Visual Studio Code for web development, it's fast and simply works.

To start, open your `index.html` file and type the following:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta lang="en-us">
    <title>Space Shooter</title>
    <script src="js/phaser.js"></script> <!-- the file name should be the same as
the Phaser script you added. -->
  </head>
  <body>
    <script src="js/SceneMainMenu.js"></script>
    <script src="js/SceneMain.js"></script>
    <script src="js/SceneGameOver.js"></script>
    <script src="js/game.js"></script>
  </body>
</html>
```

In this last snippet, we declare the file to be an HTML file (hence the .html file extension), and declare the scripts we will use. Note the order we declare the scripts that contain "Scene" in them is comparison to the `game.js` script. The order is very important because JavaScript is interpreted from top to bottom. We will be referencing code from our three scene scripts in `game.js`.

Next, we will need to initialize our Phaser game inside `game.js`. Take a moment and open up game.js if you haven't already and create a JavaScript object like the following:

```
var config = {
};
```

This JavaScript object will contain the configuration properties that we will feed our upcoming Phaser game instance. Inside our config object add:

```
  type: Phaser.WEBGL,
  width: 480,
  height: 640,
  backgroundColor: "black",
  physics: {
    default: "arcade",
    arcade: {
      gravity: { x: 0, y: 0 }
    }
  },
  scene: [],
  pixelArt: true,
  roundPixels: true
```

So far inside our configuration object we are telling Phaser that we want to ensure our game is being rendered via WebGL instead of using ordinary Canvas rendering tech. The next part of the config includes `width` and `height` which define the size of our game on the page. The property `backgroundColor` defines the background color that will be shown behind our game. The next property, `physics` defines the default physics engine that we will be using, which is `arcade`. Arcade physics work well when we want basic collision detection without many bells and whistles. Inside the physics property, we also set default gravity of our physics world. The next property, scene is set to an array which we will add to in a bit. Finally we want Phaser to ensure pixels are crisp just like the great nostalgic video games we've come to know and love.

The scene array that we have defined inside our config object is needed in order to declare the order of our scenes in our game. A scene is effectively like any sort of "screen" you see in a video game. For example, the main menu, the play state, and the game over screen are what I call separate "screens". Well, a scene is pretty much the same as a screen. We will define our scenes as a JavaScript class (a class is still an object, the syntax is just different and helps us organize our code better.) Now we will define our scene classes inside the array of the scene property we just wrote:

```
  },
  scene: [
    SceneMainMenu,
    SceneMain,
    SceneGameOver
  ],
  pixelArt: true,
```

We are now finished with our configuration object! We will not need to touch it for the duration of this course. The final line of code we need (and arguably most important) after the last curly brace of our config object is:

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

And there we have it! Our `game.js` file is complete. Now we need to define a class for each of our scene scripts. Lets open SceneMainMenu.js first and add the following:

```
class SceneMainMenu extends Phaser.Scene {
  constructor() {
    super({ key: "SceneMainMenu" });
  }
  create() {
    this.scene.start("SceneMain");
  }
}
```

Here we are declaring a class with class `SceneMainMenu`. On the same line we added `extends Phaser.Scene`. Extending `Phaser.Scene` means to build on top of Phaser's Scene class (`Phaser.Scene` means a class named Scene which is a property of the object `Phaser`.) Inside the class we have just declared, there are two functions: constructor and create. The constructor function is called immediately when instantiating (creating an instance) the class. Within the constructor we have:

```
super({ key: "SceneMainMenu" });
```

which is effectively the same (technically speaking) as:

```
var someScene = new Phaser.Scene({ key: "SceneMainMenu" });
```

Instead of creating an instance of `Phaser.Scene` and assigning it to a variable, we are instead defining our scene as a class that we can assign a custom preload, create, and eventually an update function. This is what I mean when I say build on top of the existing `Phaser.Scene` class. The create function within our SceneMainMenu class will be called as soon as the scene is started. Inside our create function we included:

```
this.scene.start("SceneMain");
```

The plan is to redirect the player to the main play scene, where all the action is. I think it would be better to start on the game play and work backwards to the main menu. Who wants to work on those boring buttons and interface elements anyway? Especially when we can instead code something much more exciting… for now… right? ☺

We can finish up by the code we just typed for SceneMainMenu into `SceneMain.js` and `SceneGameOver.js`. Below is what you should end up with for all three scripts:

```
class SceneMainMenu extends Phaser.Scene {
  constructor() {
    super({ key: "SceneMainMenu" });
  }
  create() {
    this.scene.start("SceneMain");
  }
}
```

```
class SceneMain extends Phaser.Scene {
  constructor() {
    super({ key: "SceneMain" });
  }
  create() {

  }
}

class SceneGameOver extends Phaser.Scene {
  constructor() {
    super({ key: "SceneGameOver" });
  }
  create() {

  }
}
```

Chapter 3

In the last chapter, we finished setting up the basis for our scene files: `SceneMainMenu.js`, `SceneMain.js`, and `SceneGameOver.js`. At this point you can try running the game by navigating to localhost/(game folder name)/index.html. There might be a port that's needed when viewing files on some web servers, for example, localhost:8000. If everything is right, you should see a black rectangle on the page. The black rectangle is the area where your game will be drawn.

It's time to dive back into the code. Take a second and open up `SceneMain.js`. When we left off, our `SceneMain.js` should look like:

```
class SceneMain extends Phaser.Scene {
  constructor() {
    super({ key: "SceneMain" });
  }
  create() {

  }
}
```

At this point we need to add a new function called `preload`. This function should be inserted between constructor and create. All three functions inside `SceneMain` should look like:

```
constructor() {
  super({ key: "SceneMain" });
}

preload() {

}

create() {
```

```
}
```

Inside our new preload function, we need to add the code to load our game assets. To load image files, the line you would type inside the `preload` function would be:

```
this.load.image(imageKey, path);
```

If we start with our first image file we want to load, `sprBg0.png`, we should add:

```
this.load.image("sprBg0", "content/sprBg0.png");
```

The first parameter of the `load.image` function refers to the key (a string) that we will reference the image when we add images and sprites within the game. The second parameter is the path to the image we wish to load.

Lets finish adding the rest of our loading code for our images:

```
this.load.image("sprBg0", "content/sprBg0.png");
this.load.image("sprBg1", "content/sprBg1.png");
this.load.spritesheet("sprExplosion", "content/sprExplosion.png", {
  frameWidth: 32,
  frameHeight: 32
});
this.load.spritesheet("sprEnemy0", "content/sprEnemy0.png", {
  frameWidth: 16,
  frameHeight: 16
});
this.load.image("sprEnemy1", "content/sprEnemy1.png");
this.load.spritesheet("sprEnemy2", "content/sprEnemy2.png", {
  frameWidth: 16,
  frameHeight: 16
});
this.load.image("sprLaserEnemy0", "content/sprLaserEnemy0.png");
this.load.image("sprLaserPlayer", "content/sprLaserPlayer.png");
this.load.spritesheet("sprPlayer", "content/sprPlayer.png", {
  frameWidth: 16,
  frameHeight: 16
});
```

The lines that include spritesheet means we are loading an animation instead of a static image. A sprite sheet is an image with multiple frames, side-by-side. The third argument of the `load.spritesheet` function is an object defining the frame width and height in pixels. Now we will need to add the loading code for our sounds, which follow the same format as loading images:

```
this.load.audio("sndExplode0", "content/sndExplode0.wav");
this.load.audio("sndExplode1", "content/sndExplode1.wav");
this.load.audio("sndLaser", "content/sndLaser.wav");
```

Once we have loaded our content, we need to add a little bit more code to create our animations. At the top of the create function of our screen we will create our animations:

```
this.anims.create({
  key: "sprEnemy0",
  frames: this.anims.generateFrameNumbers("sprEnemy0"),
  frameRate: 20,
  repeat: -1
});

this.anims.create({
  key: "sprEnemy2",
  frames: this.anims.generateFrameNumbers("sprEnemy2"),
  frameRate: 20,
  repeat: -1
});

this.anims.create({
  key: "sprExplosion",
  frames: this.anims.generateFrameNumbers("sprExplosion"),
  frameRate: 20,
  repeat: 0
});

this.anims.create({
  key: "sprPlayer",
  frames: this.anims.generateFrameNumbers("sprPlayer"),
  frameRate: 20,
  repeat: -1
});
```

We also need to add our sounds to some sort of variable or object so we can reference it later. I like to organize my sound references by storing them as values of a sound effect object. If there are more than one sound of a type (say, three explosion sounds), I add an array as the value of a property (going with the last example, I would go with “explosion” as the property key.) Let’s add our sound effect object and hopefully it will make more sense:

```
this.sfx = {
  explosions: [
    this.sound.add("sndExplode0"),
    this.sound.add("sndExplode1")
  ],
  laser: this.sound.add("sndLaser")
};
```

We will later be able to play sound effects from our object with, for example:

```
    this.sfx.laser.play();
```

We will also have to load some images and sounds for the main menu and game over screen. Open up `SceneMainMenu.js` and create a preload function inside SceneMainMenu. Inside the new preload function, add the following for our buttons and sounds:

```
this.load.image("sprBtnPlay", "content/sprBtnPlay.png");
this.load.image("sprBtnPlayHover", "content/sprBtnPlayHover.png");
this.load.image("sprBtnPlayDown", "content/sprBtnPlayDown.png");
this.load.image("sprBtnRestart", "content/sprBtnRestart.png");
```

```
this.load.image("sprBtnRestartHover", "content/sprBtnRestartHover.png");
this.load.image("sprBtnRestartDown", "content/sprBtnRestartDown.png");
this.load.audio("sndBtnOver", "content/sndBtnOver.wav");
this.load.audio("sndBtnDown", "content/sndBtnDown.wav");
```

Once we have added the loading code for our images and sounds. We have also created our animations and added our sounds to an object for organization. Now we can navigate back to the game in your browser and a black rectangle should still being showing. If you haven't already, open the development tools in the browser you're using. If you are using Chrome or Firefox, you can simply press F12 to open it. Look under the Console tab to ensure there are no errors (they are displayed in red.) If you see no errors, we can proceed to proceed with adding the player!

Before we add the player spaceship to our game, we should add a new file in our JavaScript folder called `Entities.js`. This entities script will contain all of the classes for the various entities in our game. We will classify the player, enemies, lasers, etc. as entities. Be sure to also add a reference to Entities.js to our index.html a line before `SceneMainMenu.js`. After adding the reference to our `Entities.js` file, open it and declare a new class named `Entity`.

```
class Entity {
  constructor() {

  }
}
```

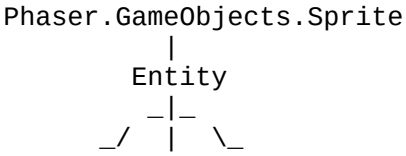
After declaring our Entity class, we need to add some parameters that our Entity class should take in. We will add these parameters between the parenthesis in our constructor:

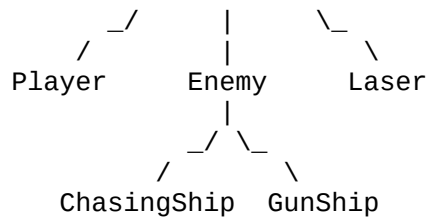
```
constructor(scene, x, y, key, type) {
}
```

Each of the parameters we've added to the constructor will be important, because we will be extending `Phaser.GameObjects.Sprite` for all entities we create. Much like how we extended `Phaser.Scene` when we first started this game, we will want to build on top of Phaser's `Sprite` class. Because of this, we will have to change class `Entity` { to:

```
class Entity extends Phaser.GameObjects.Sprite {
```

As always with extending a class, we will need to add the `super` keyword to our constructor. Since the player, enemies, and various projectiles we add will have the same basics properties, it helps to keep us from adding redundant, duplicate code. In this way we will be inheriting the properties and functions of the base `Phaser.GameObjects.Sprite` class for all of our entities. Here's (a crude diagram) of the inheritance hierarchy we will be implementing:





All of our entities will have the same basic properties (our scene, x position, y position, image key, and a type string we will use to fetch certain entities if we need to.) Lets add super into our constructor which should look like:

```
super(scene, x, y, key);
```

Essentially, what we are doing is taking the parameters in when an `Entity` is instantiated and providing `scene`, `x`, `y`, and `key` to the `Phaser.GameObjects.Sprite` base class.

After adding the super keyword to the constructor, on the next line, add the following lines:

```
this.scene = scene;
this.scene.add.existing(this);
this.scene.physics.world.enableBody(this, 0);
this.setData("type", type);
this.setData("isDead", false);
```

This piece of code assigns the scene of the `Entity`, as well as adding an instantiated `Entity` to the rendering queue of the scene. We also enable instantiated Entity's as physics objects in the physics world of the scene. Finally, we define the speed of the player. With that we should be finished adding to our `Entity` class. Now we can move on to creating our `Player` class.

By now, you should know the drill for adding a class. Add one right after the `Entity` class and name it `Player` and ensure it extends `Entity`. Add a constructor to the `Player` class with parameters: `scene`, `x`, `y`, and `key`. Then add our `super` keyword in the constructor providing it the following parameters:

```
super(scene, x, y, key, "Player");
```

We also will want a way to determine the speed that the player should move. By adding a speed key/value pair for the speed of the player, we can refer to that later for our movement functions. Under the `super` keyword, add the following:

```
this.setData("speed", 200);
```

To add the movement functions, add the following four functions after the constructor.

```
moveUp() {

}

moveDown() {
```

```
}

moveLeft() {

}

moveRight() {

}
```

Now add the following lines in each of the functions:

```
moveUp() {
  this.body.velocity.y = -this.getData("speed");
}

moveDown() {
  this.body.velocity.y = this.getData("speed");
}

moveLeft() {
  this.body.velocity.x = -this.getData("speed");
}

moveRight() {
  this.body.velocity.x = this.getData("speed");
}
```

These are the movement functions that will be called in the update function. Next, add the update function directly below the `moveRight` function. Inside the `update` function add:

```
this.body.setVelocity(0, 0);
this.x = Phaser.Math.Clamp(this.x, 0, this.scene.game.config.width);
this.y = Phaser.Math.Clamp(this.y, 0, this.scene.game.config.height);
```

We are now finished with the `Player` class! In the last bit of code, every game update the player's velocity will be set to zero. If none of the movement keys are pressed, the player will stay still. The next two lines of the player update code ensures that the player cannot move off-screen. At this point, we can create an instance of the player in the create function of `SceneMain`. Add the following to the `create` function of `SceneMain`:

```
this.player = new Player(
  this,
  this.game.config.width * 0.5,
  this.game.config.height * 0.5,
  "sprPlayer"
);
```

This is where we create the instance of the `Player` is created. We can refer to the player anywhere in `SceneMain`. The player is then positioned in the center of the canvas. If you were to try running the game, you still won't see the player move yet. This is because we first have to add the update function

to `SceneMain` and add the movement checks. Since `this.player` is now added, we can now add the update function. Add the update function right under the create function of `SceneMain` and add the following inside:

```
this.player.update();
if (this.keyW.isDown) {
  this.player.moveUp();
}
else if (this.keyS.isDown) {
  this.player.moveDown();
}
if (this.keyA.isDown) {
  this.player.moveLeft();
}
else if (this.keyD.isDown) {
  this.player.moveRight();
}
```

Like I mentioned before, `this.player.update();` will run the update code that will keep the player still, as well as ensure it can't move off-screen. With the next bit that contains `keyW`, `keyS`, `keyA`, `keyD`, you may wonder why none of those variables have been initialized yet. We will initialize the key variables in a second. These if statements check if the corresponding key is down, if so, move the player in the appropriate direction. In the create function of SceneMain, add the following to initialize our key variables after we initialize the player:

```
this.keyW = this.input.keyboard.addKey(Phaser.Input.Keyboard.KeyCodes.W);
this.keyS = this.input.keyboard.addKey(Phaser.Input.Keyboard.KeyCodes.S);
this.keyA = this.input.keyboard.addKey(Phaser.Input.Keyboard.KeyCodes.A);
this.keyD = this.input.keyboard.addKey(Phaser.Input.Keyboard.KeyCodes.D);
this.keySpace = this.input.keyboard.addKey(Phaser.Input.Keyboard.KeyCodes.SPACE);
```

If we run our code now, the player should be able to move via the W, S, A, D keys. In the next chapter, we will add the ability for the player to shoot lasers (which will use the space key.)

Chapter 4

In the last chapter of "Build a Space Shooter with Phaser 3", we finished writing our base Entity class, our player class, and the player movement. In this chapter we will implement a couple enemies and give them basic AI. At this point you should have an error-free game where you can move the player around via the W, S, A, D keys. If so, it's time to open `Entities.js` back up.

At the bottom of Entities.js under the Player class, add three new classes called ChaserShip, GunShip, and CarrierShip:

```
class ChaserShip extends Entity {
  constructor(scene, x, y) {
    super(scene, x, y, "sprEnemy1", "ChaserShip");
  }
}
```

```
class GunShip extends Entity {
  constructor(scene, x, y) {
    super(scene, x, y, "sprEnemy0", "GunShip");
    this.play("sprEnemy0");
  }
}
class CarrierShip extends Entity {
  constructor(scene, x, y) {
    super(scene, x, y, "sprEnemy2", "CarrierShip");
    this.play("sprEnemy2");
  }
}
```

Classes `ChaserShip`, `GunShip`, and `CarrierShip` should extend the `Entity` class that we have created in the last chapter. Then we effectively call the constructor of `Entity` with provide the corresponding parameters. We will be able to build on top of the `Entity` class and in a second build our simple AI for each enemy. For each enemy class, under the super keyword, add the following:

```
this.body.velocity.y = Phaser.Math.Between(50, 100);
```

The above line sets the y velocity of the enemy to be a random integer between 50 and 100. We will be spawning the enemies past the top of the screen, which will cause the enemy to move down the canvas.

Next, go back to `SceneMain.js`. We will need to create a Group to hold our enemies, the lasers shot by enemies, and the lasers shot by the player. In the create function after the line setting `this.keySpace`, add:

```
this.enemies = this.add.group();
this.enemyLasers = this.add.group();
this.playerLasers = this.add.group();
```

There still won't be any enemies spawning from the top of the screen yet if we run our game. We first have to create an event (it will act as a timer) which will spawn our enemies. After our `playerLasers` group, add the following code:

```
this.time.addEvent({
  delay: 100,
  callback: function() {
    var enemy = new GunShip(
      this,
      Phaser.Math.Between(0, this.game.config.width),
      0
    );
    this.enemies.add(enemy);
  },
  callbackScope: this,
  loop: true
});
```

If we try running the game now, we should see many `GunShip` enemies moving down from the top of the screen. Now, we will give our `GunShip` enemies the ability to shoot. First, we have to create

another class called `EnemyLaser` right after the Player class of our Entities.js file. `EnemyLaser` should extend `Entity` as well.

```
class EnemyLaser extends Entity {
  constructor(scene, x, y) {
    super(scene, x, y, "sprLaserEnemy0");
    this.body.velocity.y = 200;
  }
}
```

Now we can go back to our `GunShip` class, specifically the constructor. Under where we set the `y` velocity, we can add a new event.

```
this.shootTimer = this.scene.time.addEvent({
  delay: 1000,
  callback: function() {
    var laser = new EnemyLaser(
      this.scene,
      this.x,
      this.y
    );
    laser.setScale(this.scaleX);
    this.scene.enemyLasers.add(laser);
  },
  callbackScope: this,
  loop: true
});
```

Do note that we are assigning the above event to a variable called `this.shootTimer`. We should create a new function inside `EnemyLaser` called `onDestroy`. `onDestroy` is not a function used by Phaser, but you can call it anything. We will be using this function to destroy the shoot timer when the enemy is destroyed. Add the `onDestroy` function to our `GunShip` class and add the following inside:

```
if (this.shootTimer !== undefined) {
  if (this.shootTimer) {
    this.shootTimer.remove(false);
  }
}
```

When we run the game, you should see the army of gun ship enemies coming down from the top of the screen. All of the enemies should also be shooting lasers as well. Now that we see everything is working, we can cut back the amount of gun ships are being spawned at once. To do this, navigate to our `SceneMain.js` file and change the delay of the timer we made.

```
this.time.addEvent({
  delay: 1000, // this can be changed to a higher value like 1000
  callback: function() {
    var enemy = new GunShip(
      this,
      Phaser.Math.Between(0, this.game.config.width),
      0
```

```
    );
    this.enemies.add(enemy);
  },
  callbackScope: this,
  loop: true
});
```

When we run the game, we should see a much more reasonable amount of gun ships being spawned.

Back in `Entities.js`, we will need to add a little bit of code to the constructor of the ChaserShip class:

```
this.states = {
  MOVE_DOWN: "MOVE_DOWN",
  CHASE: "CHASE"
};
this.state = this.states.MOVE_DOWN;
```

This code does two things: create an object that has two properties which we can use to set the state of the chaser ship, and then we set the state to the value of the `"MOVE_DOWN"` property (the value is the string `"MOVE_DOWN"`.)

We can now add an update function to the `ChaserShip` class. The update function is where we will code in the AI for the `ChaserShip` class. We will code the intelligence for the `ChaserShip` enemy first, since it's slightly more complicated. Navigate back to `Entities.js` and in the update function of the `ChaserShip` class, add the following:

```
if (!this.getData("isDead") && this.scene.player) {
  if (Phaser.Math.Distance.Between(
    this.x,
    this.y,
    this.scene.player.x,
    this.scene.player.y
  ) < 320) {
    this.state = this.states.CHASE;
  }
  if (this.state == this.states.CHASE) {
    var dx = this.scene.player.x - this.x;
    var dy = this.scene.player.y - this.y;
    var angle = Math.atan2(dy, dx);
    var speed = 100;
    this.body.setVelocity(
      Math.cos(angle) * speed,
      Math.sin(angle) * speed
    );
  }
}
```

With this code, chaser enemies will move down the screen. However, as soon as it is within 320 pixels to the player, it will start chasing the player. If you want the chaser ship to rotate, feel free to add the following right after (or at the end of) our chase condition:

```
if (this.x < this.scene.player.x) {
  this.angle -= 5;
}
else {
  this.angle += 5;
}
```

In order to spawn the chaser ship, we will have to go back to `SceneMain.js` and add a new function called getEnemiesByType. Inside this new function add:

```
getEnemiesByType(type) {
  var arr = [];
  for (var i = 0; i < this.enemies.getChildren().length; i++) {
    var enemy = this.enemies.getChildren()[i];
    if (enemy.getData("type") == type) {
      arr.push(enemy);
    }
  }
  return arr;
}
```

The above code will allow us to provide an enemy type and get all the enemies in the enemies group. This code loops through the enemies group and checks if the type of the enemy in the loop is equal to the type that is given as a parameter.

Once we added the `getEnemiesByType` function, we will need to modify our spawner event. Within the anonymous function of the callback property let's change:

```
var enemy = new GunShip(
  this,
  Phaser.Math.Between(0, this.game.config.width),
  0
);
this.enemies.add(enemy);

to:

var enemy = null;
if (Phaser.Math.Between(0, 10) >= 3) {
  enemy = new GunShip(
    this,
    Phaser.Math.Between(0, this.game.config.width),
    0
  );
}
else if (Phaser.Math.Between(0, 10) >= 5) {
  if (this.getEnemiesByType("ChaserShip").length < 5) {
    enemy = new ChaserShip(
      this,
      Phaser.Math.Between(0, this.game.config.width),
      0
    );
  }
}
else {
```

```
  enemy = new CarrierShip(
    this,
    Phaser.Math.Between(0, this.game.config.width),
    0
  );
}
if (enemy !== null) {
  enemy.setScale(Phaser.Math.Between(10, 20) * 0.1);
  this.enemies.add(enemy);
}
```

Going through this block, we add a condition that picks one of our three enemy classes: `GunShip`, `ChaserShip`, or `CarrierShip` to be spawned. After setting the enemy variable to either enemy class, we then add it to the enemies group. If a `ChaserShip` is picked to be spawned, we check to ensure there are not more than five `ChaserShip`s before spawning another. Before we add an enemy to the group, we also apply a random scale to the enemy. Since each enemy extends our `Entity` class, which in turn extends `Phaser.GameObjects.Sprite`, we can set a scale to enemies, just as we can to any other `Phaser.GameObjects.Sprite`.

In the update function, we need to update enemies in the `this.enemies` group. To do so, add the following at the end of the update function.

```
for (var i = 0; i < this.enemies.getChildren().length; i++) {
  var enemy = this.enemies.getChildren()[i];
  enemy.update();
}
```

If we try running the game now, we should see that chaser ships should be moving towards the player ship once they get within distance.

Last, we will finish up this chapter by giving the player the ability to shoot. Navigate back to the Player class and in the constructor add:

```
this.setData("isShooting", false);
this.setData("timerShootDelay", 10);
this.setData("timerShootTick", this.getData("timerShootDelay") - 1);
```

We are setting up what I would call, a “manual timer”. We are not using events for the shooting ability of the player. This is because, we do not want a delay to shoot when initially pressing the space key. In the update function of the Player, we will add the rest of the logic for our “manual timer”:

```
if (this.getData("isShooting")) {
  if (this.getData("timerShootTick") < this.getData("timerShootDelay")) {
    this.setData("timerShootTick", this.getData("timerShootTick") + 1); // every
game update, increase timerShootTick by one until we reach the value of
timerShootDelay
  }
  else { // when the "manual timer" is triggered:
    var laser = new PlayerLaser(this.scene, this.x, this.y);
    this.scene.playerLasers.add(laser);
```

```
    this.scene.sfx.laser.play(); // play the laser sound effect
    this.setData("timerShootTick", 0);
  }
}
```

The only thing left we have to do is add the `PlayerLaser` class to our `Entities.js` file. We can add this class right under the Player class and before the `EnemyLaser` class. This will keep our player related classes together, and our enemy related classes together. Create a constructor inside the `PlayerLaser` class and add the same code to the constructor as we did with the `EnemyLaser` class. Then, remove the negate sign from where we set the y velocity value. This will cause player lasers to move up instead of down. The `PlayerLaser` class should now look like:

```
class PlayerLaser extends Entity {
  constructor(scene, x, y) {
    super(scene, x, y, "sprLaserPlayer");
    this.body.velocity.y = -200;
  }
}
```

The last thing we need to do to allow the player to shoot is go back to `SceneMain.js` and add the following condition under our movement code:

```
if (this.keySpace.isDown) {
  this.player.setData("isShooting", true);
}
else {
  this.player.setData("timerShootTick",
this.player.getData("timerShootDelay") - 1);
  this.player.setData("isShooting", false);
}
```

Try running the game now! We should see that when we press space, the player ship is able to shoot.

We are finished with adding the ability to shoot lasers for both the player and enemies! Before we move on to collisions, it will be a good idea to add what is called frustum culling. Frustum culling will allow us to remove everything that moves off screen, which frees up processing power and memory. Without frustum culling, if we let our game run for a while, we can see the enormous amount of lag.

In order to add frustum culling, we will have to move to the update function of `SceneMain`. Currently, we should have a for loop where we update enemies. Inside the for after the ending curly brace where we update the enemy, add the following code:

```
if (enemy.x < -enemy.displayWidth ||
    enemy.x > this.game.config.width + enemy.displayWidth ||
    enemy.y < -enemy.displayHeight * 4 ||
    enemy.y > this.game.config.height + enemy.displayHeight) {
    if (enemy) {
      if (enemy.onDestroy !== undefined) {
        enemy.onDestroy();
```

```
      }
      enemy.destroy();
    }
}
```

We can also add the same for enemy lasers and player lasers:

```
for (var i = 0; i < this.enemyLasers.getChildren().length; i++) {
  var laser = this.enemyLasers.getChildren()[i];
  laser.update();
  if (laser.x < -laser.displayWidth ||
    laser.x > this.game.config.width + laser.displayWidth ||
    laser.y < -laser.displayHeight * 4 ||
    laser.y > this.game.config.height + laser.displayHeight) {
    if (laser) {
      laser.destroy();
    }
  }
}

for (var i = 0; i < this.playerLasers.getChildren().length; i++) {
  var laser = this.playerLasers.getChildren()[i];
  laser.update();
  if (laser.x < -laser.displayWidth ||
    laser.x > this.game.config.width + laser.displayWidth ||
    laser.y < -laser.displayHeight * 4 ||
    laser.y > this.game.config.height + laser.displayHeight) {
    if (laser) {
      laser.destroy();
    }
  }
}
```

To add collisions, we will navigate to our `SceneMain.js` and at a look at our create function. We will need to add what's called a collider below our enemy spawn event. Colliders allow you to add a collision check between two game objects. So, if there's a collision between the two objects, the callback you specified will be called and you will receive the two instances that have collided as parameters. We can create a collider between this.playerLasers and this.enemies. In code, we would write this as:

```
this.physics.add.collider(this.playerLasers, this.enemies, function(playerLaser,
enemy) {

});
```

If we wanted to have the enemy destroyed upon being hit by a player laser, we can write inside the anonymous function:

```
if (enemy) {
  if (enemy.onDestroy !== undefined) {
    enemy.onDestroy();
  }
  enemy.explode(true);
  playerLaser.destroy();
```

```
}
```

The above code checks if the enemy is still active (and not destroyed), and then destroys it if true.

If we run the game, we should see that instances in the this.enemies group are able to destroy enemies. The next step is to add a collider between `this.player` and `this.enemies`:

```
this.physics.add.overlap(this.player, this.enemies, function(player, enemy) {
  if (!player.getData("isDead") &&
      !enemy.getData("isDead")) {
    player.explode(false);
    enemy.explode(true);
  }
});
```

We can also add a collider between this.player and `this.enemyLasers`. By essentially copying the code from above, we can accomplish the same effect, but instead with the enemy lasers.

```
this.physics.add.overlap(this.player, this.enemyLasers, function(player, laser) {
  if (!player.getData("isDead") &&
      !laser.getData("isDead")) {
    player.explode(false);
    laser.destroy();
  }
});
```

If we run this, we will get an error that explode is not a function. No worries though, we can just head back to `Entities.js` and take a look at the `Entity` class. In the `Entity` class, we need to add a new function called explode. We will be taking in `canDestroy` as the sole parameter of this new function. The `canDestroy` parameter determines whether when explode is called, if the entity will be destroyed, or just be set invisible. Inside the explode function we can add:

```
if (!this.getData("isDead")) {
  // Set the texture to the explosion image, then play the animation
  this.setTexture("sprExplosion");  // this refers to the same animation key we
used when we added this.anims.create previously
  this.play("sprExplosion"); // play the animation
  // pick a random explosion sound within the array we defined in this.sfx in
SceneMain
  this.scene.sfx.explosions[Phaser.Math.Between(0, this.scene.sfx.explosions.length
- 1)].play();
  if (this.shootTimer !== undefined) {
    if (this.shootTimer) {
      this.shootTimer.remove(false);
    }
  }
  this.setAngle(0);
  this.body.setVelocity(0, 0);
  this.on('animationcomplete', function() {
    if (canDestroy) {
      this.destroy();
    }
    else {
```

```
      this.setVisible(false);
    }
  }, this);
  this.setData("isDead", true);
}
```

If we run the game, you may notice that the player can still move around and shoot, even if the player ship explodes. We can fix this by adding a check around the player update call and the movement and shooting calls in SceneMain. The ending result should appear as:

```
if (!this.player.getData("isDead")) {
  this.player.update();
  if (this.keyW.isDown) {
    this.player.moveUp();
  }
  else if (this.keyS.isDown) {
    this.player.moveDown();
  }
  if (this.keyA.isDown) {
    this.player.moveLeft();
  }
  else if (this.keyD.isDown) {
    this.player.moveRight();
  }
  if (this.keySpace.isDown) {
    this.player.setData("isShooting", true);
  }
  else {
    this.player.setData("timerShootTick", this.player.getData("timerShootDelay") -
1);
    this.player.setData("isShooting", false);
  }
}
```

We have accomplished the “meat and potatoes” of this course in this chapter. We have added enemies, player lasers, enemy lasers, frustum culling, and collisions. In the next chapter I will be covering how to add a scrolling background, create the main menu, and the game over screen.

Chapter 5

In the last chapter of this course, “Build a Space Shooter with Phaser 3”, we have added the bulk of our code for our space shooter. We have covered adding enemies, player lasers, enemy lasers, frustum culling, and collisions in the last chapter. There are a few things we will finish up in this chapter to conclude this course. We will be adding a scrolling background, filling in the main menu, and create the game over screen.

We will start by adding the scrolling background. The scrolling background will have multiple, scrolling at different speeds. First, let’s go to our `Entities.js` file. At bottom of the file, we can add a new class, `ScrollingBackground`. It does not need to extend anything.

```
class ScrollingBackground {
  constructor(scene, key, velocityY) {

  }
}
```

Our constructor will be taking in the scene we instantiate a scrolling background, and an array of the image keys we want to create layers of. We will first set the scene of the instance to our parameter we've taken in. We will also store our keys into an instance of a scrolling background.

```
this.scene = scene;
this.key = key;
this.velocityY = velocityY;
```

We will be implementing a function called `createLayers`. Before we do however, we need to create a group inside our constructor.

```
    this.layers = this.scene.add.group();
```

Inside our new `createLayers` function, let's add the following code to create sprites out of the array of image keys we're given.

```
for (var i = 0; i < 2; i++) {
  // creating two backgrounds will allow a continuous scroll
  var layer = this.scene.add.sprite(0, 0, this.key);
  layer.y = (layer.displayHeight * i);
  var flipX = Phaser.Math.Between(0, 10) >= 5 ? -1 : 1;
  var flipY = Phaser.Math.Between(0, 10) >= 5 ? -1 : 1;
  layer.setScale(flipX * 2, flipY * 2);
  layer.setDepth(-5 - (i - 1));
  this.scene.physics.world.enableBody(layer, 0);
  layer.body.velocity.y = this.velocityY;
  this.layers.add(layer);
}
```

The above code iterates through each key we take in. For each key, we create two sprites with the key at each iteration of the first for loop. We then add the sprite to our layers group. We then apply a downwards velocity in which each layer is slower the farther back they are based on the value of i.

We can then call `createLayers` at the bottom of our constructor.

```
    this.createLayers();
```

Now we can head over to `SceneMain.js` and initialize the scrolling background. Insert the following code before we instantiate the player and add it after we define `this.sfx`.

```
this.backgrounds = [];
for (var i = 0; i < 5; i++) { // create five scrolling backgrounds
  var bg = new ScrollingBackground(this, "sprBg0", i * 10);
  this.backgrounds.push(bg);
}
```

Try running the game, you should see a the stars behind the player. All four backgrounds should be drawn now. We can now head back to `Entities.js` and add an update function with the following code inside:

```
if (this.layers.getChildren()[0].y > 0) {
  for (var i = 0; i < this.layers.getChildren().length; i++) {
    var layer = this.layers.getChildren()[i];
    layer.y = (-layer.displayHeight) + (layer.displayHeight * i);
  }
}
```

The above code allows the background layers to wrap back around to the bottom. If we didn't have this code, the backgrounds will just move off-screen and there will remain the black background. We can go back to `SceneMain.js` and call the update function of our scrolling background instance. In the update function of `SceneMain`, add the following code:

```
for (var i = 0; i < this.backgrounds.length; i++) {
  this.backgrounds[i].update();
}
```

That concludes adding our background to the game! If we run the game we should see multiple background layers scrolling down at different speeds.

We can finish up by adding our main menu and game over screen.

Navigate to `SceneMainMenu` and remove the line that starts `SceneMain`. Before we continue however, we should a sound effect object for `SceneMainMenu`. Add the following to the very top of the `create` function:

```
this.sfx = {
  btnOver: this.sound.add("sndBtnOver"),
  btnDown: this.sound.add("sndBtnDown")
};
```

We can then add the play button to the create function by adding a sprite.

```
this.btnPlay = this.add.sprite(
  this.game.config.width * 0.5,
  this.game.config.height * 0.5,
  "sprBtnPlay"
);
```

In order to start `SceneMain`, we will need to first set our sprite as interactive. Add the following directly below where we defined `this.btnPlay`:

```
this.btnPlay.setInteractive();
```

Since we set our sprite as being interactive, we can now add pointer events such as over, out, down, and up. We can execute code when each of these events are triggered by the mouse or tap. The first event we will add is the `pointerover` event. We will be changing the texture of the button to our

`sprBtnPlayHover.png` image when the pointer moves over the button. Add the following after we set our button as interactive:

```
this.btnPlay.on("pointerover", function() {
  this.btnPlay.setTexture("sprBtnPlayHover"); // set the button texture to
sprBtnPlayHover
  this.sfx.btnOver.play(); // play the button over sound
}, this);
```

If we run the game and move the mouse over the button we should see that the texture changes.

Now we can add the `pointerout` event. In this event we will reset the texture back to the normal play button image. Add the following under where we define the `pointerover` event:

```
this.btnPlay.on("pointerout", function() {
  this.setTexture("sprBtnPlay");
});
```

If we run the game again and move the mouse over the button, then off, we should see the button texture reset to the default image.

Next, we can add the `pointerdown` event. This is where we will change the texture of the play button to `sprBtnPlayDown.png`.

```
this.btnPlay.on("pointerdown", function() {
  this.btnPlay.setTexture("sprBtnPlayDown");
  this.sfx.btnDown.play();
}, this);
```

If we run the game, we should see the button texture change to `sprBtnPlayDown.png` when we move the mouse over the button and click. We can then add the `pointerup` event to reset the button texture after we click.

```
this.btnPlay.on("pointerup", function() {
  this.setTexture("sprBtnPlay");
}, this);
```

We can a line one more line inside our `pointerup` event to start `SceneMain`. The final `pointerup` event should look like:

```
this.btnPlay.on("pointerup", function() {
  this.btnPlay.setTexture("sprBtnPlay");
  this.scene.start("SceneMain");
}, this);
```

When we run the game and click the play button, it should now start `SceneMain`!

There are now just a couple things we can do to finish up our main menu. The first is adding a title. To add our title, we can create text. Add the following under the `pointerup` event

```
this.title = this.add.text(this.game.config.width * 0.5, 128, "SPACE SHOOTER", {
```

```
  fontFamily: 'monospace',
  fontSize: 48,
  fontStyle: 'bold',
  color: '#ffffff',
  align: 'center'
});
```

To center the title, we can set the origin of the text to half the width, and half the height. We can do this by writing the following under our title definition:

```
this.title.setOrigin(0.5);
```

The last thing we will do for our main menu is add our scrolling background in. We can copy the code from the create function of `SceneMain` to `SceneMainMenu`, but the code is available below as well.

```
this.backgrounds = [];
for (var i = 0; i < 5; i++) {
  var keys = ["sprBg0", "sprBg1"];
  var key = keys[Phaser.Math.Between(0, keys.length - 1)];
  var bg = new ScrollingBackground(this, key, i * 10);
  this.backgrounds.push(bg);
}
```

We can also create the update function as well. In the update function we will update our background layers. Add the following to the update function to update our scrolling backgrounds.

```
for (var i = 0; i < this.backgrounds.length; i++) {
  this.backgrounds[i].update();
}
```

Try running the game now. You may notice some green squares in the top-left corner of the screen.

The reason why we are not seeing our backgrounds, is because they haven't been loaded yet. They have been loaded in `SceneMain`, but if we look at our scene array in `game.js`, `SceneMain` is after `SceneMainMenu` so we are not able to access loaded content from `SceneMain`. To fix this, we will have to move the lines loading `sprBg0.png` and `sprBg1.png` to the `preload` function of `SceneMainMenu`. The `preload` function of `SceneMainMenu` should look similar to:

```
this.load.image("sprBg0", "content/sprBg0.png");
this.load.image("sprBg1", "content/sprBg1.png");
this.load.image("sprBtnPlay", "content/sprBtnPlay.png");
this.load.image("sprBtnPlayHover", "content/sprBtnPlayHover.png");
this.load.image("sprBtnPlayDown", "content/sprBtnPlayDown.png");
this.load.image("sprBtnRestart", "content/sprBtnRestart.png");
this.load.image("sprBtnRestartHover", "content/sprBtnRestartHover.png");
this.load.image("sprBtnRestartDown", "content/sprBtnRestartDown.png");
this.load.audio("sndBtnOver", "content/sndBtnOver.wav");
this.load.audio("sndBtnDown", "content/sndBtnDown.wav");
```

When we now run the game, we should see background looking how it should.

The final part of this course will be fleshing out `SceneGameOver` and adding the scene start code in the appropriate colliders.

We can copy over the title code from `SceneMainMenu` to `SceneGameOver`. In the create function of `SceneGameOver`, we can add the following code to create our title. This title code is essentially identical to the code we used previously. The only change we make is changing the string to draw from "SPACE SHOOTER" to GAME OVER".

```
this.title = this.add.text(this.game.config.width * 0.5, 128, "GAME OVER", {
  fontFamily: 'monospace',
  fontSize: 48,
  fontStyle: 'bold',
  color: '#ffffff',
  align: 'center'
});
this.title.setOrigin(0.5);
```

Next we can add our sound effect object, as we did with both `SceneMainMenu` and `SceneMain`.

```
this.sfx = {
  btnOver: this.sound.add("sndBtnOver"),
  btnDown: this.sound.add("sndBtnDown")
};
```

Once we have added the game over title and sound effect object, we can add in the restart button. The code is pretty much identical to that which we used with the play button. Add the following to the `create` function of `SceneGameOver`:

```
this.btnRestart = this.add.sprite(
  this.game.config.width * 0.5,
  this.game.config.height * 0.5,
  "sprBtnRestart"
);
this.btnRestart.setInteractive();
this.btnRestart.on("pointerover", function() {
  this.btnRestart.setTexture("sprBtnRestartHover"); // set the button texture to
sprBtnPlayHover
  this.sfx.btnOver.play(); // play the button over sound
}, this);
this.btnRestart.on("pointerout", function() {
  this.setTexture("sprBtnRestart");
});
this.btnRestart.on("pointerdown", function() {
  this.btnRestart.setTexture("sprBtnRestartDown");
  this.sfx.btnDown.play();
}, this);
this.btnRestart.on("pointerup", function() {
  this.btnRestart.setTexture("sprBtnRestart");
```

```
  this.scene.start("SceneMain");
}, this);
```

After our button code, we can create our background layers. As we have before, we can add the following code to the `create` function:

```
this.backgrounds = [];
for (var i = 0; i < 5; i++) {
  var keys = ["sprBg0", "sprBg1"];
  var key = keys[Phaser.Math.Between(0, keys.length - 1)];
  var bg = new ScrollingBackground(this, key, i * 10);
  this.backgrounds.push(bg);
}
```

Then add our update code to update the backgrounds in the `update` function:

```
for (var i = 0; i < this.backgrounds.length; i++) {
  this.backgrounds[i].update();
}
```

We finished adding the game over title, restart button, and scrolling background layers. That's great, except we can't see our changes yet because we haven't started `SceneGameOver` anywhere yet. To change this, we can go to our `Entities.js` file and create an `onDestroy` function for our player. The plan is, we will want to `create` an event that will start `SceneGameOver` after a slight delay. Inside our new `onDestroy` function, we can add the following code:

```
this.scene.time.addEvent({ // go to game over scene
  delay: 1000,
  callback: function() {
    this.scene.scene.start("SceneGameOver");
  },
  callbackScope: this,
  loop: false
});
```

In order to finish with our `onDestroy` function, we will need to go back to `SceneMain.js` and look in the create function. Specifically we will have to call the player's `onDestroy` function in every collider that involves the player. Just after we call `player.explode(false);`, we can insert: `player.onDestroy();`

There we have it! This concludes the end of our five chapter course, "Build a Space Shooter with Phaser 3"!

There are quite a features you could add to this project as well. A few I can think of are:
adding lives
adding a score
adding upgrades
adding bosses

If you decide to expand this project, I would really love to hear about it! Add a comment below or email me at jared.york@jaredyork.com. ☺

At this point, we have covered the majority of the components you need in order to make your own games with Phaser 3! I would also like to thank the Phaser 3 team for making such an awesome HTML5 game framework and for all the work they have done. You can learn more about Phaser at their official website, here. As always, please feel free to ask me questions, and I will be more than glad to help.