
Build Arcade Games with Phaser 3: Space Boulders
Version 1.0
Written by Jared York

In this course, we will be creating a game similar to the arcade classic, Asteroids. For the purposes of this course, we will call our game, “Space Boulders.” Before beginning, I highly recommend you have intermediate knowledge of JavaScript. If not, I’ve written a free course, “JavaScript Beginner Blocks”, which will cover the basics. There’s just a few steps we will have to do before we jump into the code.

Ensure a Web Server is Set Up

Even though Phaser games are ran in the browser, unfortunately you can’t simply run HTML files directly from your file system. When requesting files over http, the there server only allows you to access files you’re allowed to. Wen loading a file form the local file system ( the file:// protocol), your browser highly restricts it for obvious security reasons. It would be very bad to allow code on a website to read anything from your local file system. To get around this, we will need to server our game from a web server.

We highly recommend checking out Phaser’s official guide, “Getting Started with Phaser 3”, to learn which web server is best for your system, and links are included for each. The guide also provides summaries for each of the web servers mentioned.

Next, enter the folder and create a new file called, “index.html”. Our index file is where we will link our game scripts and the Phaser script.

We can also create two folders. I named the first one, “content” for our game content (sprites, audio, etc.), and the other one, “js”, which will contain our game scripts and the Phaser script. Feel free to name these folders anything you want. One of the folders just needs to be dedicated to content, and the other for JavaScript files. Once we have our two folders, create four new files inside the newly created JavaScript folder: SceneMainMenu.js, SceneMain.js, Entities.js, and game.js. I will go through what each of these files will contain.

So far, the file structure we created should look like:
(game folder)
|_ index.html
|_ content/
|_ js/
  |_ Entities.js
  |_ game.js
  |_ SceneMain.js
  |_ SceneMainMenu.js

In order to add content to our game, we first need content. I prepared some assets for this course which can be downloaded for free here. Otherwise, you can create your own assets if you wish.

Content needed:

- Sprites (images)
    - sprAsteroid0.png
    - sprAsteroid1.png
    - sprAsteroid2.png
    - sprAsteroid3.png
    - sprBall.png
    - sprBullet.png
    - sprHalfLine.png
    - sprIconLife.png
    - sprPaddle.png
    - sprPixel.png
    - sprPlayer.png
    - sprSaucerLarge.png
    - sprSaucerSmall.png
- Sounds (.wav files)
    - sndExplode.wav
    - sndLaserEnemy.wav
    - sndLaserPlayer.wav

After you've downloaded the assets (or created your own), we can move those files into the content directory we made.

Finally, the last thing we can do before jumping into the code is download the latest Phaser script. A common method for acquiring this script, is to head over to GitHub (specifically, here). You will want to download either phaser.js or phaser.min.js. The file phaser.js contains the Phaser source code, useful for contributing to Phaser or just taking a look under-the-hood. The other script, phaser.min.js is meant for distribution, and is compressed down to save file size. For our purposes, it doesn't really matter which one to download. Decide, then click the corresponding link. You will then see a page with a "View raw" link, in the center of the page, roughly halfway down. Click the "View raw" link, then right click anywhere on the page of code that appears. The context menu should appear where you can save the Phaser script in the JavaScript directory we created previously.

Now we can dive into the code. Open index.html with our text editor or code editor of choice. Add the following code to define our HTML document and link our scripts:

```
<!DOCTYPE html>
<html>
        <head>
                <meta charset="utf-8">
                <meta lang="en-us">
                <title>Asteroids</title>
```

```
        <script src="js/phaser.js"></script>
    </head>

    <body>
        <script src="js/Entities.js"></script>
        <script src="js/SceneMainMenu.js"></script>
        <script src="js/SceneMain.js"></script>
        <script src="js/game.js"></script>
    </body>
</html>
```

Once we've typed out our HTML file, we can move over to game.js. Add the following to game.js to create our Phaser game instance, and provide it some configuration properties:

```
var config = {
    type: Phaser.WEBGL,
    width: 640,
    height: 480,
    background: "black",
    physics: {
        default: "arcade",
        arcade: {
            gravity: { x: 0, y: 0 }
        }
    },
    scene: [
        SceneMain
    ],
    pixelArt: true,
    roundPixels: true
};
var game = new Phaser.Game(config);
```

Our game.js file is complete! We can now move on to our Entities.js file and define the entities for our game. The first thing we will do is create a base class for our entities which will extend a Phaser sprite. Let me back up though, when I say we will create a class, we will really be creating an object with some ES6 syntactic sugar. We will be creating four additional classes after our base entity class as well: Asteroid, Saucer, Bullet, and Player. Add the following to the Entities.js file to add our classes:

```
class Entity extends Phaser.GameObjects.Sprite {
    constructor(scene, x, y, key) {
```

```
        }
}

class Asteroid extends Entity {
        constructor(scene, x, y, key) {

        }
}

class Saucer extends Entity {
        constructor(scene, x, y, key) {

        }

        onDestroy() {

        }
}

class Bullet extends Entity {
        constructor(scene, x, y, isFriendly) {

        }
}

class Player extends Entity {
        constructor(scene, x, y) {

        }

        turnLeft() {

        }

        turnRight() {

        }

        moveForward() {

        }

        shoot() {
```

```
        }

        update() {

        }
}
```

Let's start with our base entity class, Entity. I mentioned before that our Entity class will extend a Phaser sprite, more specifically, Phaser.GameObjects.Sprite. As we have above, we can declare a class and extend it with the following syntax:

```
class <class name> extends <object> {

}
```

If we want code to run when we create an instance of a class, we can add a constructor. We can define any parameters we want in the constructor, just like any other function. Add the following to the constructor of the Entity class:

```
super(scene, x, y, key);
this.scene = scene;
this.scene.add.existing(this);
this.scene.physics.world.enableBody(this, 0);
```

When we provide arguments to the super keyword, we are effectively instantiating a Phaser.GameObjects.Sprite the same parameters we took in from the constructor. We also ensure that any class that extends Entity will be added to the display list and has an enabled physics body when instantiated.

Next, we will move on to the Asteroid class. The Asteroid class will extend Entity. Let's add the following to the constructor:

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

this.body.setCircle(this.displayWidth * 0.5);

this.body.setVelocity(
        Phaser.Math.Between(-100, 100),
        Phaser.Math.Between(-100, 100)
);

this.setData("level", 0);
```

Above, we use our usual super keyword to initialize the default Phaser sprite. We also set the velocity of the asteroid to a random value in both the x and y direction. Finally, we also set a little bit of data, “level”, which will be used to determine whether the asteroid has been broken off of a larger asteroid.

We can now move down to the Saucer class. In the constructor, we will be adding quite a bit more to this class than the previous ones. We will be setting the velocity of the saucer, but also a shoot timer which will be used for firing bullets. Let’s begin by adding the following to the constructor of the Saucer class:

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

this.body.setVelocity(
        Phaser.Math.Between(-100, 100),
        Phaser.Math.Between(-100, 100)
);

this.shootTimer = this.scene.time.addEvent({
        delay: 1000,
        callback: function() {

        },
        callbackScope: this,
        loop: true
});
```

We can utilize the addEvent method from the time subsystem of Phaser to create our timer. As a parameter of the addEvent method, we provide an object with the following properties: delay, callback: callbackScope, and loop. Inside the callback, we have declared an anonymous function. Inside the anonymous function, we want to check if the saucer is small or large. Depending on if a saucer is a small saucer or a large saucer, we want instances to shoot two different ways. If a saucer is a small saucer, we want to fire bullets directly at the player. If the saucer is a large saucer, we want to fire bullets in random directions. To start, let’s adding a check to ensure the scene is defined, and the following if-else-if statement to the anonymous function of the timer callback:

```
if (this.scene !== undefined) {
        if (key == “sprSaucerSmall”) {

        }
        else if (key == “sprSaucerLarge”) {
```

```
        }
}
```

Inside our if statement checking if the texture key is sprSaucerSmall, we need to add some logic to find the angle of the player relative to the saucer. Add the following three lines to calculate the angle:

```
var dx = this.scene.player.x - this.x;
var dy = this.scene.player.y - this.y;
var angle = Math.atan2(dy, dx);
```

Next, we want to create a new instance of the Bullet class and set it to a local variable, bullet:

```
var bullet = new Bullet(this.scene, this.x, this.y, false);
```

We also want to set a couple more properties. Let's add a line to set some data to represent that the bullet is not friendly:

```
bullet.setData("isFriendly", false);
```

Then, we can set the velocity of the bullet. Feel free to tweak 100 to any value you wish. This number represents the speed. We are multiplying the speed by the x and y components of the angle we calculated. Add the following:

```
bullet.body.setVelocity(
        100 * Math.cos(angle),
        100 * Math.sin(angle)
);
```

Finally, we need to add the bullet to the pool of bullets (which we will get to instantiating later):

```
this.scene.bullets.add(bullet);
```

Let's now take a look at the else-if statement checking if the texture key is sprSaucerLarge. We will not have to make the same calculation as above for calculating any angles. We will simply be picking an angle between 0 and 360 degrees the convert it to radians. Add the following to the else-if statement:

```
var angle = Phaser.Math.Between(0, 360) * Math.PI / 180;
```

The rest of the code is the same as above. Add the following block:

```
var bullet = new Bullet(this.scene, this.x, this.y, false);
```

```
bullet.setData("isFriendly", false);

bullet.body.setVelocity(
        100 * Math.cos(angle),
        100 * Math.sin(angle)
);

this.scene.bullets.add(bullet);
```

When a saucer is destroyed, we also want to destroy the shoot timer so it doesn't keep shooting where the saucer was. Inside the onDestroy method of the Saucer class, add the following:

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

We are now finished adding to the Saucer class. The next class is the Bullet class, which will be very small in comparison. Add the following two lines to the Bullet class:

```
super(scene, x, y, "sprBullet");
this.setData("isFriendly", isFriendly);
```

That is all the code we need for the Bullet class. The last class that's left that we have to add to, is the Player class. We will want to restrict the player to the bounds of the screen. Then, we also want to set a little bit of data to represent that the player is not moving by default. Add the following code to the constructor of the Player class:

```
super(scene, x, y, "sprPlayer");
this.body.setCollideWorldBounds(true);
this.setData("isMoving", false);
```

Next, we can move on to the three movement methods we've added to the Player class. Add the following code for: turnLeft, turnRight, and moveForward:

**turnLeft:**

```
this.setAngle(Math.round(this.angle - 3));
```

**turnRight:**

```
this.setAngle(Math.round(this.angle + 3));
```

**moveForward:**

```
this.setData("isMoving", true);

this.body.velocity.x += 2 * Math.cos(this.rotation);
this.body.velocity.y += 2 * Math.sin(this.rotation);
```

The next method we have to add to is the shoot method. We want the player to shoot in the direction the player's spaceship is facing. Add the following to the shoot function:

```
this.scene.sfx.laserPlayer.play();

var bullet = new Bullet(this.scene, this.x, this.y, true);
bullet.setOrigin(0.5);
bullet.setData("isFriendly", true);

var speed = 1000;
bullet.body.setVelocity(
        speed * Math.cos(this.rotation) + Phaser.Math.Between(-50, 50),
        speed * Math.sin(this.rotation) + Phaser.Math.Between(-50, 50)
);
this.scene.bullets.add(bullet);
```

The last method we need to fill in is the update method. If the player is not moving, we want the player ship to decelerate. By default, every frame we also want to set the player data, isMoving, to be false. We also want to decelerate the player when the player isn't moving. Add the following to the update method:

```
if (!this.getData("isMoving")) {
        this.body.velocity.x *= 0.995;
        this.body.velocity.y *= 0.995;
}
this.setData("isMoving", false);
```

With that, we are finished with the Entities.js file! The next step is to take a look at the SceneMain.js file. All scenes will be a class which declares Phaser.Scene. Add the following code block to declare SceneMain:

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

```
init(data) {

}

preload() {

}

create() {

}

createLivesIcons() {

}

createExplosion(x, y, amount) {

}

onLifeDown() {

}

getSpawnPosition() {

}

spawnAsteroid() {

}

spawnSaucer() {

}

addScore(amount) {

}

update() {

}
```

```
}
```

Now, let's take a look at the init method. You might see we're taking in a parameter called data. The plan is, when the player is destroyed, we will restart the scene. The problem is, usually all properties are reset, including the score, amount of lives, etc. We can actually save the data like the score, amount of lives, etc. by passing data as a parameter when the scene starts. So in the init method, the data parameter is the stored data we're taking in when a scene is started. Let's add the following code to the init method to store our data we received:

```
this.passingData = data;
```

There we have it! Let's move to the next function, preload. Add the following code to load our images and sounds:

```
this.load.image("sprIconLife", "content/sprIconLife.png");
this.load.image("sprPlayer", "content/sprPlayer.png");
this.load.image("sprBullet", "content/sprBullet.png");
this.load.image("sprSaucerSmall", "content/sprSaucerSmall.png");
this.load.image("sprSaucerLarge", "content/sprSaucerLarge.png");

for (var i = 0; i < 4; i++) {
        this.load.image("sprAsteroid" + i, "content/sprAsteroid" + i + ".png");
}

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

Let's also add the loading code to our preload method for loading our sounds:

```
this.load.audio("sndExplode", "content/sndExplode.wav");
this.load.audio("sndLaserEnemy", "content/sndLaserEnemy.wav");
this.load.audio("sndLaserPlayer", "content/sndLaserPlayer.wav");
```

Previously, we added code to the init method to retrieve stored data and assign it to the passingData object. However, if no data has been previously stored, that data parameter will be an empty object. When we start the game and there's no previous data, we want to define the passingData and provide it the properties we want (score, lives, etc.) Let's start by adding the following to the create method:

```
if (Object.getOwnPropertyNames(this.passingData).length == 0 &&
        this.passingData.constructor === Object) {

        this.passingData = {
                maxLives: 3,
```

```
            lives: 3,
            score: 0
        };
}
```

Let's also add an object to hold our sounds. This object will keep our sounds organized without cluttering the scene with extra properties. Add the following code to the create method:

```
this.sfx = {
        explode: this.sound.add("sndExplode"),
        laserEnemy: this.sound.add("sndLaserEnemy"),
        laserPlayer: this.sound.add("sndLaserPlayer")
};
```

Now, let's add the following code to instantiate the player and create the groups representing pools of sprites:

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

this.bullets = this.add.group();
this.asteroids = this.add.group();
this.saucers = this.add.group();
this.iconLives = this.add.group();
```

Next, let's define the lives left and create a text object to display the score:

```
this.maxLives = 3;
this.lives = this.maxLives;
this.score = 0;
this.textScore = this.add.text(
        32,
        32,
        this.score,
        {
            fontFamily: "monospace",
            fontSize: 32,
            align: "left"
        }
);
```

Let's also add some code to allow the game to reset when the player has zero lives. Add the following:

```
if (this.passingData.lives == 0) {
	this.textGameOver = this.add.text(
		this.game.config.width * 0.5,
		64,
		“GAME OVER”,
		{
			fontFamily: “monospace”,
			fontSize: 72,
			align: “left”
		}
	);
	this.textGameOver.setOrigin(0.5);

	this.time.addEvent({
		delay: 3000,
		callback: function() {
			this.scene.start(“SceneMain”, { });
		},
		callbackScope: this,
		loop: false
	});
}
```

We can also call the createLivesIcons method to create the sprites for our lives icons:

```
this.createLivesIcons();
```

Next, let’s add the keyboard properties assigned to key code objects that we can check if they’re being pressed down or not:

```
this.keyW = this.input.keyboard.addKey(Phaser.Input.Keyboard.KeyCodes.W);
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.SPACE);
```

Let’s also add a keyboard event that will call the player’s shoot method when the space key is pressed down:

```
this.input.keyboard.on(“keydown_SPACE”, function() {
	if (this.player.active) {
		if (!this.player.getData(“hasShot”)) {
			this.player.shoot();
		}
```

```
                this.player.setData("hasShot", true);
        }
}, this);
```

We also want to add a keyboard event to allow the player to shoot when the player releases the space key:

```
this.input.keyboard.on("keyup_SPACE", function() {
        if (this.player.active) {
                this.player.shoot();
        }
}, this);
```

The next part we have to add is the timer which will spawn asteroids and spawners. Add the following code block to add this timer and it's associated logic:

```
this.time.addEvent({
        delay: 500,
        callback: function() {
                this.spawnAsteroid();

                if (Phaser.Math.Between(0, 100) > 75) {
                        this.spawnSaucer();
                }
        },
        callbackScope: this,
        loop: true
});
```

After adding this timer event, we can add our collision checks. With Phaser, we can use what's called a a collider. Colliders allow us to check for collisions between two game objects (that includes sprites AND groups!) Add the following code block to add the first two colliders:

```
this.physics.add.collider(this.player, this.asteroids, function(player, asteroid) {
        this.createExplosion(player.x, player.y, asteroid.displayWidth);

        if (this.player) {
                this.onLifeDown();

                this.player.destroy();
        }
}, null, this);
```

```
this.physics.add.collider(this.player, this.saucers, function(player, saucer) {
	this.createExplosion(player.x, player.y, player.displayWidth);

	if (player) {
		this.onLifeDown();

		player.destroy();
	}
}, null, this);
```

Similar to colliders, Phaser also offers a similar method called “overlap.” Overlap allows you to check if one game object is overlapping another game object. We will need to add two calls to the overlap method to check if any bullets are moving through the player, and if any bullets are moving through an asteroid. Add the following below our colliders:

```
this.physics.add.collider(this.player, this.bullets, function(player, bullet) {
	if (!bullet.getData(“isFriendly”)) {
		this.createExplosion(player.x, player.y, player.displayWidth);

		if (player) {
			this.onLifeDown();

			player.destroy();
		}
	}
}, null, this);

this.physics.add.overlap(this.bullets, this.asteroids, function(bullet, asteroid) {
	if (bullet.getData(“isFriendly”)) {
		this.createExplosion(bullet.x, bullet.y, asteroid.displayWidth);

		var oldAsteroidPos = new Phaser.Math.Vector2(asteroid.x, asteroid.y);
		var oldAsteroidKey = asteroid.texture.key;
		var oldAsteroidLevel = asteroid.getData(“level”);
		if (asteroid) {
			asteroid.destroy();
		}

		// give points
		switch (oldAsteroidLevel) {
			case 0: {
				this.addScore(20);
				break;
```

```
        }

        case 1: {
                this.addScore(50);
                break;
        }

        case 2: {
                this.addScore(100);
                break;
        }
}

if (oldAsteroidLevel < 2) {
        for (var i = 0; i < 2; i++) {
                var scale = 1;
                var key = “”;
                if (oldAsteroidKey == “sprAsteroid0” || oldAsteroidKey ==
“sprAsteroid1”) {
                        scale = 0.5;
                        key = “sprAsteroid” + Phaser.Math.Between(0, 1);
                }
                else if (oldAsteroidKey == “sprAsteroid2” || oldAsteroidKey ==
“sprAsteroid3”) {
                        key = “sprAsteroid” + Phaser.Math.Between(0, 1);
                }

                var newAsteroid = new Asteroid(
                        this,
                        oldAsteroidPos.x,
                        oldAsteroidPos.y,
                        “sprAsteroid” + Phaser.Math.Between(0, 3)
                );
                newAsteroid.setScale(scale);
                newAsteroid.setTexture(key);
                newAsteroid.setData(“level”, oldAsteroidLevel + 1);

                newAsteroid.body.setVelocity(
                        Phaser.Math.Between(-200, 200),
                        Phaser.Math.Between(-200, 200)
                );
                this.asteroids.add(newAsteroid);
        }
```

```
                }

                if (bullet) {
                        bullet.destroy();
                }
        }
}, null, this);

this.physics.add.overlap(this.bullets, this.saucers, function(bullet, saucer) {

        if (bullet.getData("isFriendly")) {
                this.createExplosion(bullet.x, bullet.y, saucer.displayWidth);

                if (saucer.texture.key == "sprSaucerSmall") {
                        this.addScore(1000);
                }

                if (saucer.texture.key == "sprSaucerLarge") {
                        this.addScore(200);
                }

                if (bullet) {
                        bullet.destroy();
                }

                if (saucer) {
                        saucer.onDestroy();
                        saucer.destroy();
                }
        }

}, null, this);
```

Now that we're done with the create method, we can move on to the createLivesIcons method. We will be creating icons for the amount of lives the player has left. Using a for loop, we can accomplish this. Add the following code to the createLivesIcons method:

```
for (var i = 0; i < this.passingData.lives; i++) {
        var icon = this.add.sprite(
                this.textScore.x + (i * 16) + 12,
                this.textScore.y + 42,
                "sprIconLife"
        );
```

```
        icon.setOrigin(0.5);
        this.iconLives.add(icon);
}
```

We will also want to re-use the code for creating explosions where we need to. Let's add this code to the createExplosion method:

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

var explosion = this.add.particles("sprPixel").createEmitter({
        x: x,
        y: y,
        speed: { min: -500, max: 500 },
        scale: { start: 1, end: 0 },
        blendMode: "SCREEN",
        lifespan: 600
});

for (var i = 0; i < amount; i++) {
        explosion.explode();
}
```

The next method we will add is onLifeDown. If our lives are more than zero, simply destroy all entities and call the reset method. If there are zero lives left, set the game as over. Add the following code to the onLifeDown method:

```
if (this.passingData.lives > 0) {
        this.passingData.lives--;

        this.time.addEvent({
                delay: 1000,
                callback: function() {
                        this.scene.start("SceneMain", this.passingData);
                },
                callbackScope: this,
                loop: false
        });
}
else {
        this.passingData.lives = 0;
}
```

We also want reuse code to position asteroids and saucers spawned.  Add the following code block to getSpawnPosition:

```
var sides = [“top”, “right”, “bottom”, “left”];
var side = sides[Phaser.Math.Between(0, sides.length - 1)];

var position = new Phaser.Math.Vector2(0, 0);
switch (side) {
	case “top”: {
		position = new Phaser.Math.Vector2(
			Phaser.Math.Between(0, this.game.config.width),
			-128
		);

		break;
	}

	case “right”: {
		position = new Phaser.Math.Vector2(
			this.game.config.width + 128,
			Phaser.Math.Between(0, this.game.config.height)
		);

		break;
	}

	case “bottom”: {
		position = new Phaser.Math.Vector2(
			Phaser.Math.Between(0, this.game.config.width),
			this.game.config.height + 128
		);
	}

	case “left”: {
		position = new Phaser.Math.Vector2(
			0,
			Phaser.Math.Between(-120, this.game.config.height)
		);
	}
}

return position;
```

Now, we will want to add the code to spawn asteroids and saucers. We'll first start with the spawnAsteroid method:

```
var position = this.getSpawnPosition();

var asteroid = new Asteroid(this, position.x, position.y, "sprAsteroid" + Phaser.Math.Between(0, 3));

if (asteroid.texture.key == "sprAsteroid0" ||
    asteroid.texture.key == "sprAsteroid1") {

        asteroid.setData("level", 1);

}

this.asteroids.add(asteroid);
```

We can also add the spawning logic in our spawnSaucer method:

```
var position = this.getSpawnPosition();

var imageKey = "";
if (Phaser.Math.Between(0, 10) > 5) {
        imageKey = "sprSaucerLarge";
}
else {
        imageKey = "sprSaucerSmall";
}

var saucer = new Saucer(this, position.x, position.y, imageKey);
this.saucers.add(saucer);
```

Let's add the ability to easily add to the player's score where we want to. Add to the addScore method:

```
this.score += amount;
this.textScore.setText(this.score);
```

The last method we need to fill in will be the update method. In the update method, we will be checking if keys are down, then move the player in the corresponding direction. We will also be implementing frustum culling for bullets, asteroids, saucers. Before we jump too far ahead, let's add a condition to check that the player is still alive, and we will add the movement code inside this if statement:

```
if (this.player.active) {
	this.player.update();

	if (this.keyA.isDown) {
		this.player.turnLeft();
	}

	if (this.keyD.isDown) {
		this.player.turnRight();
	}

	if (this.keyW.isDown) {
		this.player.moveForward();

		// engine particles
		var gas = this.add.particles("sprPixel").createEmitter({
			x: this.player.x + Phaser.Math.Between(-2, 2),
			y: this.player.y + Phaser.Math.Between(-2, 2),
			speed: { min: -200, max: 200 },
			scale: { start: 1, end: 0 },
			angle: { min: this.player.angle +(180 - 5), max: this.player.angle + (180 +
		5) },
			blendMode: "SCREEN",
			lifespan: { min: 60, max: 320 }
		});

		for (var i = 0; i < 5; i++) {
			gas.explode();
		}
	}
}
```

With the above code, when the player moves forward, we will spawn some particles shooting from the engine of the player's ship. The next part is where we will be implementing frustum culling. Frustum culling is essentially not rendering or destroying objects offscreen to help reduce lag by not rendering the whole scene at once. We should only be displaying what's within the bounds of the screen. Let's start with adding the culling logic for the bullets first. Keeping this relatively simple, for each bullet, we will checking if the bullet is over 500 pixels form the center of the screen. If it is, we destroy the bullet. We should implement this logic below our condition checking if the player is active:

```
for (var i = 0; i < this.bullets.getChildren().length; i++) {
```

```
        var bullet = this.bullets.getChildren()[i];

        if (Phaser.Math.Distance.Between(
                bullet.x,
                bullet.y,
                this.game.config.width * 0.5,
                this.game.config.height * 0.5
        ) > 500) {
                if (bullet) {
                        bullet.destroy();
                }
        }
}
```

Let's add the identical code for the asteroid and saucer:

```
for (var i = 0; i < this.asteroids.getChildren().length; i++) {
        var asteroid = this.asteroids.getChildren()[i];

        if (Phaser.Math.Distance.Between(
                asteroid.x,
                asteroid.y,
                this.game.config.width * 0.5,
                this.game.config.height * 0.5
        ) > 500) {
                if (asteroid) {
                        asteroid.destroy();
                }
        }
}

for (var i = 0; i < this.saucers.getChildren().length; i++) {
        var saucer = this.saucers.getChildren()[i];

        if (Phaser.Math.Distance.Between(
                saucer.x,
                saucer.y,
                this.game.config.width * 0.5,
                this.game.config.height * 0.5
        ) > 500) {
                if (saucer) {
                        saucer.onDestroy();
                        saucer.destroy();
```

```
                }
        }
}
```

With that, this will conclude our SceneMain.js file. We can move on to editing our final file, SceneMainMenu.js. Let's start by declaring our scene class:

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

        preload() {

        }

        create() {

        }
}
```

We will start by adding the loading code for the play button and the sound effect for the button. Add the following to the preload method:

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

this.load.audio("sndBtn", "content/sndBtn.wav");
```

Next, we can move to the create method. First, we want to add an object to store our button sound effect:

```
this.sfx = {
        btn: this.sound.add("sndBtn")
};
```

We can also create the header text for our main menu:

```
this.textTitle = this.add.text(
        this.game.config.width * 0.5,
        64,
        "SPACE BOULDERS",
        {
```

```
                fontFamily: “monospace”,
                fontSize: 72,
                align: “center”
        }
);
this.textTitle.setOrigin(0.5);
```

The next part is creating a sprite for the play button:

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

We can also set the sprite as being interactive:

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

We can also add pointer events for when the mouse or touch point is over the play button, when pressing down, and releasing the play button on the sprite. Let’s add the following event listeners:

```
this.btnPlay.on(“pointerover”, function() {
        this.btnPlay.setTexture(“sprBtnPlayHover”);
        this.sfx.btn.play();
}, this);

this.btnPlay.on(“pointerout”, function() {
        this.setTexture(“sprBtnPlay”);
});

this.btnPlay.on(“pointerdown”, function() {
        this.sfx.btn.play();
}, this);

this.btnPlay.on(“pointerup”, function() {
        this.scene.start(“SceneMain”);
}, this);
```

Let’s also add SceneMainMenu to the scenes array of the config object in our game.js file. In game.js, add SceneMainMenu right above SceneMain in the scene array:

```
},
scene: [
        SceneMainMenu
        SceneMain
],
```

That now concludes this course, “Build Arcade Games with Phaser 3: Space Boulders.” As with all of my courses, don’t hesitate to contact me at jared.york@yorkcs.com if you have any questions, suggestions, or any other feedback. I would love to help you if you need any help after going through this course. The final source code for this course can be found on Github. You can find more of my courses and tutorials here. If you found this course valuable, and would like updates regarding future courses and tutorials I write, be sure to fill out the form.