Build Arcade Games with Phaser 3: Saucer Invaders
Version 1.0
Written by Jared York

In this course, we will be covering how to build a game similar to the arcade classic, *Space Invaders*, with Phaser 3! After completing this course, you should be able to tweak, modify, and develop your own games! For the purposes of this course, we will refer to our game as, "Saucer Invaders."

Course Requirements:

- Basic or intermediate knowledge of JavaScript
- Code editor (not necessarily required, but download one if you haven't)
- Tutorial assets (link is included)

Without further ado, let's get everything set up!

ENSURE A WEB SERVER IS SET UP

First, you will need to download a web server if you haven't already. I would recommend checking out Phaser's official guide, "Getting Started with Phaser 3," for details of this process. We need a web server, because by default, if we load a web page through HTTPS, the server checks to ensure we're allowed to access the file. There would be huge security issues if websites were able to simply read from the raw file systems of its users. Once you have downloaded and installed a web server, feel free to move on to the next part.

CREATE FILES AND DIRECTORIES

Next, we will need to create the files and directories for our game. Navigate to the location where your web server serves files from. For example, with WAMP Server, files are served from C:/wamp64/www. Other web servers will serve web pages from different locations, so be sure to find out. Create a directory in the location where you web server hosts files, and name it anything you wish. This folder will be the directory that contains our game. Inside that folder, create two new folders. I named the first one, "js", for Javascript, and the other, "content" for our game's content. In our game's directory, create a file called index.html. In our JavaScript folder, create four new files: game.js, Entities.js, SceneMain.js, and SceneMainMenu.js. We will also have to download the Phaser script as well. Navigate to this GitHub page. We can download either phaser.js, or phaser.main.js. The first, phaser.js, is useful for contributing back to the Phaser project, or just checking feature implementations under-the-hood. The other file, phaser.min.js, is useful for distribution, as the script is compressed to save file size. Decide, and click the link. A new page should greet you, with a "View raw" link roughly in the center of the page. Click this link, then right click on the page of code that appears. A context menu should appear where you can choose the option "Save Page As", or something similar. Then, a save dialog should pop up. Save the file in the JavaScript directory we just created. WIth that, let's jump right into the code!

Let's start by adding some code to our index.html file:

```
<!DOCTYPE html>
<html>
	<head>
		<meta lang=”en-us”>
		<meta charset=”utf-8”>
		<title>Saucer Invaders</title>
	</head>

	<body>
		<!-- LOAD FONT -->

		<!-- LOAD SCRIPTS -->
		<script src=”js/phaser.js”></script> <!-- use name of the downloaded script -->
		<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>
```

Before continuing, we can also downloaded a cool looking arcade font. For this course we can use the font, Arcadepix. Arcadepix is available for downloaded here. This awesome thing is, this font is 100% free for personal and commercial use. Download and extract the font folder. You don't have to install the font, just simply add the TTF file to our game's content folder. In the head tag of our index.html file, add the following under the title tag:

```
<style>
	@font-face {
		font-family: Arcadepix;
		src: url(“content/ARCADEPI.TTF”);
		font-weight: 400;
		font-weight: normal;
	}
</style>
```

Just under the “LOAD FONT” comment, add the following within the body tag:

```
<div style=”font-family:Arcadepix; position:absolute; left:-1000px; visibility:hidden;”>.</div>
```

The above line will load the font and ensure that we can use it in our game. Let's add the rest of our content to our game. I have made the assets for this course freely available here. Otherwise, feel free to add your own assets.

Here is a list of assets we'll need:

- **Sprites (images)**
  - sprBtnPlay.png (the play button)
  - sprBtnPlayHover.png (the play button when mouse moves over)
  - sprEnemy0.png
  - sprExplosion.png
  - sprLaserEnemy.png
  - sprLaserPlayer.png
  - sprPlayer.png
  - sprShieldTile.png
- **Sounds (.wav) files**
  - sprExplosion.wav
  - sprLaserPlayer.wav
  - sprLaserEnemy.wav

Now, we can head over to the game.js file. In the game.js file, we want to create a configuration object where we can add properties used to create our Phaser game:

```
var config = {
        type: Phaser.WEBGL,
        width: 480,
        height: 640,
        backgroundColor: “black”,
        physics: {
                default: “arcade”,
                arcade: {
                        gravity: { x: 0, y: 0 }
                }
        },
        scene: [
                SceneMain
        ],
        pixelArt: true,
        roundPixels: true
};
```

We can then create our Phaser game with the following line, and pass our configuration object as an argument:

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

Now, we're finished with our index.html and game.js file! Now, let's start creating classes for entities in our game inside the Entities.js file. First, we want to create a base entity class that will extend or "build on top" of Phaser's regular old sprite object. The rest of our entities will then extend the Entity class, which in turn extends the Phaser.GameObjects.Sprite class. Classes in JavaScript are just ordinary objects with ES6 syntactic sugar. Add the following to create our Entity class:

```
class Entity extends Phaser.GameObjects.Sprite {
	constructor(scene, x, y, key) {
		super(scene, x, y, key);
		this.scene.add.existing(this);
		this.scene.physics.world.enableBody(this, 0);
	}
}
```

The above code defines a class, provide the arguments to keep the Phaser.GameObject.Sprite happy. This initializes entities the same way as ordinary Phaser sprites. We then add any entity extending Entity to the display list, then enable the physics body.

The next class we can declare, is our Player class:

```
class Player extends Entity {
	constructor(scene, x, y) {
		super(scene, x, y, "sprPlayer");
		this.setScale(2);
	}
}
```

Next, we'll create the PlayerLaser class:

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

Then we can add the EnemyLaser class:

```
class EnemyLaser extends Entity {
	constructor(scene, x, y, key) {
		super(scene, x,y, "sprLaserEnemy");
	}
}
```

Let's also create a class for our saucers, and name it Enemy:

```
class Enemy extends Entity {
	constructor(scene, x, y, key) {
		super(scene, x, y, key);
		this.setOrigin(0);
	}
}
```

The only difference in the above class compared to the previous ones, is the call to the setOrigin method. Setting the origin affects where a sprite is placed, as well as the point of rotation. Usually this method takes in two arguments, one for the X axis and one for the Y axis. The neat thing however, is we can provide it just a single parameter, and it will apply the value to both the X and Y axis of the origin.

We can now add the class for our shield tiles. Since we're using very small sprites, we will double the scale of the tiles. This way we can keep our arcade-ish sprites while allowing us not to have to use a magnifying glass to see them. We also set the depth to -2, because we will be drawing black rectangles to give the illusion of broken tiles, which will be rendered on layer -1. Add the following to define our ShieldTile class:

```
class ShieldTile extends Entity {
	constructor(scene, x, y) {
		super(scene, x, y, "sprShieldTile");
		this.setOrigin(0);
		this.setScale(2);
		this.setDepth(-2);
	}
}
```

The last class we have to add is the Explosion class. In the constructor of this class, we will be playing the explosion animation, and when the animation is complete, destroy the instance. Add the following code to add the Explosion class:

```
class Explosion extends Entity {
	constructor(scene, x, y) {
		super(scene, x, y, "sprExplosion");
		this.play("sprExplosion");
		this.setOrigin(0);
		this.setScale(2);
		this.on("animationcomplete", function() {
			if (this) {
```

```
                        this.destroy();
                    }
                });
        }
}
```

Fantastic! We are now finished with the Entities.js file. We can now move on to the meat and potatoes, which is the SceneMain.js file. To start, we will define a class which will extend Phaser.Scene:

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

The first thing we add next is the init method. The plan is, when the player is destroyed, restart the scene. The problem is, by default all properties are reset, which would normally contain our score, lives, etc. The solution is to store all of our important properties in a property I name, passingData. The init method will allow us to retrieve the passingData from the last scene. Let’s create our init method and assign the data object to the passingData property:

```
init(data) {
        this.passingData = data;
}
```

Next, let’s load our content! Add the preload method and associated code block to load our content:

```
preload() {
        this.load.image(“sprPlayer”, “content/sprPlayer.png”);
        this.load.spritesheet(“sprEnemy0”, “content/sprEnemy0.png”, {
                frameWidth: 8,
                frameHeight: 8
        });
        this.load.image(“sprShieldTile”, “content/sprShieldTile.png”);
        this.load.image(“sprLaserEnemy”, “content/sprLaserEnemy.png”);
        this.load.image(“sprLaserPlayer”, “content/sprLaserPlayer.png”);
        this.load.spritesheet(“sprExplosion”, “content/sprExplosion.png”, {
                frameWidth: 8,
                frameHeight: 8
        });
```

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

Now, let's declare the create method:

```
create() {

}
```

The first thing we will add to the create method is a way to check if the passingData object is empty, and if so, create a new object containing the important properties we'll reference. Add the following code to check if the passingData object is empty:

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

        this.passingData = {
                maxLives: 3,
                lives: 3,
                score: 0,
                highScore: 0
        };
}
```

Let's also create an object to store our sound effects. This will keep our scene much more organized and less littered with extra properties. Add the following to create our sound effects object:

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

Next, we'll create our animations and add them to Phaser's animation manager:

```
this.anims.create({
        key: "sprEnemy0",
        frames: this.anims.generateFrameNumbers("sprEnemy0"),
        frameRate: 10,
        repeat: -1
```

```
});

this.anims.create({
        key: “sprExplosion”,
        frames: this.anims.generateFrameNumbers(“sprExplosion”),
        frameRate: 15,
        repeat: 0
});
```

We will also want a way to display our score and high score on the top of the screen. Phaser provides us a method to add text to the screen. First, we will add the label displaying “SCORE”. The plan is to display the score under this heading. Add the following to create our score label:

```
this.textLabelScore = this.add.text(
        32,
        32,
        “SCORE <1>”,
        {
                fontFamily: “Arcadepix”,
                fontSize: 16,
                align: “left”
        }
);
```

Then, we can add the text object for displaying our actual score:

```
this.textScore = this.add.text(
        32,
        64,
        this.passingData.score,
        {
                fontFamily: “Arcadepix”,
                fontSize: 16,
                align: “left”
        }
);
```

Next, we will add the text label for displaying “HI-SCORE”, just like in the original game.

```
this.textLabelHighScore = this.add.text(
        180,
        32,
        “HI-SCORE”,
```

```
        {
                fontFamily: “Arcadepix”,
                fontSize: 16,
                align: “left”
        }
);
```

Then we’ll add text for displaying the actual high score:

```
this.textHighScore = this.add.text(
        180,
        64,
        this.passingData.highScore,
        {
                fontFamily: “Arcadepix”,
                fontSize: 16,
                align: “left”
        }
);
```

The next step is creating the player. We can instantiate the player by adding the following to create an instance of the class, Player:

```
this.player = new Player(
        this,
        this.game.config.width * 0.5,
        this.game.config.height - 64
);
```

We will also need a way to handle key presses for the A, D, and space key. Add the following properties that we can utilize in our movement logic:

```
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);
```

Let’s also create the basis for a manual timer for the purposes of the player’s ability to shoot:

```
this.playerShootDelay = 30;
this.playerShootTick = 0;
```

When we create the shields, we will need some sort of “template” to know how to structure shield tiles into the shape of a shield. To do this, we can define a property with an array of arrays, defining the shape of the shield:

```
this.shieldPattern = [
        [0, 1, 1, 1, 1, 1, 0],
        [1, 1, 1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1, 1, 1],
        [1, 1, 0, 0, 0, 1, 1],
        [1, 1, 0, 0, 0, 1, 1],
        [1, 1, 0, 0, 0, 1, 1]
];
```

Next, we’ll have to add a way to hold all of our enemies, lasers, shieldTiles, etc. Phaser provides us groups to be able to store game objects in. Add the following code to create the groups we need:

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

We will also want a few properties for determining the direction the enemies are moving, the last direction enemies were moving, and the approximate bounding box of the army of invaders:

```
this.lastEnemyMoveDir = “RIGHT”;
this.enemyMoveDir = “LEFT”;
this.enemyRect = new Phaser.Geom.Rectangle(
        0,
        0,
        Math.round((this.game.config.width / 24) * 0.75) * 24,
        Math.round((this.game.config.height / 20) * 0.25) * 20
);
```

Next, we need to add the enemies to our scene. Add the following code:

```
for (var x = 0; x < Math.round((this.game.config.width / 24) * 0.75); x++) {
        for (var y = 0; y < Math.round((this.game.config.height / 20) * 0.25); y++) {
                var enemy = new Enemy(this, x * 24, 128 + (y * 20), “sprEnemy0”);
                enemy.play(“sprEnemy0”);
                enemy.setScale(2);
```

```
                this.enemies.add(enemy);
        }
}
```

Then, we can call the method updateEnemiesMovement, which will start the movement timer for the enemies. We will be defining that method shortly, but add the following method call:

this.updateEnemiesMovement();

We will also want to add a method call to updateEnemiesShooting, which will start a timer used for determining whether an enemy should shoot or not. Call updateEnemiesShooting:

this.updateEnemiesShooting();

Let's add a method call to updatePlayerMovement, for the timer determining when the player can move. Add the following line:

this.updatePlayerMovement();

Then, we can add a call to updatePlayerShooting, which is almost identical to updateEnemiesShooting, but we're checking whether the player can shoot or not. Add the following method call to updatePlayerShooting:

this.updatePlayerShooting();

Let's add a method call for updating the positions of the lasers. The method will be updating both the player's lasers and enemy lasers based on a timer. Add a method call to updateLasers:

this.updateLasers();

Finally, let's call the createLivesIcons method. This method will be used for creating the lives icons representing the amount of remaining lives the player has. Add the following line:

this.createLivesIcons();

Before we move on, I think it's important to explain why we will be using all these timers. If you were to play many arcade games, you'll notice the movement for objects usually isn't that smooth. Rather, many game objects move steps over time. We will be emulating this concept using timers in Phaser.

The next part we have to add to the create method are our collision checks. We will need collision checks between player lasers and enemies, player lasers and enemy lasers, player

lasers and shield tiles, enemy lasers and shield tiles, enemy lasers and the player, and enemies and the player. Phaser provides us a way to check if two sprites, or any sprites of a group, collide with each other. Let's add the first call to create an overlap check:

```
this.physics.add.overlap(this.playerLasers, this.enemies, function(laser, enemy) {
	if (laser) {
		laser.destroy();
	}

	if (enemy) {
		this.createExplosion(enemy.x, enemy.y);
		this.addScore(10);
		enemy.destroy();
	}
}, null, this);
```

We can go on to add the rest of our overlap checks:

```
this.physics.add.overlap(this.playerLasers, this.enemyLasers, function(playerLaser,
enemyLaser) {

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

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

}, null, this);

this.physics.add.overlap(this.playerLasers, this.shieldTiles, function(laser, tile) {
	if (laser) {
		laser.destroy();
	}

	this.destroyShieldTile(tile);
}, null, this);

this.physics.add.overlap(this.enemyLasers, this.shieldTiles, function(laser, tile) {
	if (laser) {
		laser.destroy();
	}
```

```
        this.destroyShieldTile(tile);
}, null, this);

this.physics.add.overlap(this.player, this.enemies, function(player, enemy) {
        if (player) {
                player.destroy();

                this.onLifeDown();
        }
}, null, this);

this.physics.add.overlap(this.player, this.enemyLasers, function(player, laser) {
        if (player) {
                player.destroy();

                this.onLifeDown();
        }

        if (laser) {
                laser.destroy();
        }
}, null, this);
```

We want four (kind of equally spaced) shields near the bottom of the screen. Add the following code to do so:

```
var totalShieldsWidth = (4 * 96) + (7 * 8);
for (var i = 0; i < 4; i++) {
        this.addShield(
                ((this.game.config.width * 0.5) - (totalShieldsWidth * 0.5)) + ((i * 96) + (7 * 8)),
                this.game.config.height - 128
        );
}
```

Finally, we need to add some code for our high score system. If a high score is not defined yet, set the high score to zero in our local storage. Otherwise, if a high score exists, retrieve it from local storage and assign it to the high score property in our passingData. We then set the high score text to display the current high score. Add the following to the bottom of our create method:

```
if (localStorage.getItem(“highScore”) == null) {
        localStorage.setItem(“highScore”, 0);
```

```
}
else {
        this.passingData.highScore = localStorage.getItem("highScore");
        this.textHighScore.setText(this.passingData.highScore);
}
```

With that we are finished with the create method, the longest method! Next we'll be adding quite a few methods to the SceneMain class. Splitting our logic into methods helps keep our codebase tidy and relatively efficient. Let's start off simple by adding the addScore method. We will be taking in a parameter, which is the amount of points we want to add to the player's score. Add the following to create our addScore method:

```
addScore(amount) {
        this.passingData.score += amount;
        this.textScore.setText(this.passingData.score);
}
```

Let's also add a short little method for setting our enemy direction. This method will be setting the last direction the enemies were moving, then assign the new direction from the parameter. Add this code to declare our setEnemyDirection method:

```
setEnemyDirection(direction) {
        this.lastEnemyMoveDir = this.enemyMoveDir;
        this.enemyMoveDir = direction;
}
```

Now, we will be getting into creating several methods that will start our timers. Let's start with updateEnemiesMovement. This method will move the enemies in the direction set from setEnemyDirection. If the area of enemies touches either side of the screen, we set the enemies direction to be the reverse of the current enemies direction. When the enemies are moved down, we also make the delay of the timer shorter, so enemies will move horizontally faster. Add the following to declare our method, updateEnemiesMovement:

```
updateEnemiesMovement() {

}
```

We can then add our timer, in the form of an event which we enable the property loop:

```
this.enemyMoveTimer = this.time.addEvent({
        delay: 1024,
        callback: function() {
```

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

Inside the callback of the timer, add the following code:

```
if (this.enemyMoveDir == “RIGHT”) {
        this.enemyRect.x += 6;

        if (this.enemyRect.x + this.enemyRect.width > this.game.config.width - 20) {
                this.setEnemyDirection(“DOWN”);
        }
}
else if (this.enemyMoveDir == “LEFT”) {
        this.enemyRect.x -= 6;

        if (this.enemyRect.x < 20) {
                this.setEnemyDirection(“DOWN”);
        }
}
else if (this.enemyMoveDir == “DOWN”) {
        this.enemyMoveTimer.delay -= 100;
        this.moveEnemiesDown();
}

for (var i = this.enemies.getChildren().length - 1; i >= 0; i--) {
        var enemy = this.enemies.getChildren()[i];

        if (this.enemyMoveDir == “RIGHT”) {
                enemy.x += 6;
        }
        else if (this.enemyMoveDir == “LEFT”) {
                enemy.x -= 6;
        }
}
```

Next, we can add a method for updateEnemiesShooting and add the following code:

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

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

                    if (Phaser.Math.Between(0, 1000) > 995) {
                        var laser = new EnemyLaser(this, enemy.x, enemy.y);
                        this.enemyLasers.add(laser);

                        this.sfx.laserEnemy.play();
                    }
                }
            },
            callbackScope: this,
            loop: true
        });
}
```

Let’s also add a new method named moveEnemiesDown and add the following code:

```
moveEnemiesDown() {
	for (var i = this.enemies.getChildren().length - 1; i >= 0; i--) {
		var enemy = this.enemies.getChildren()[i];

		enemy.y += 20;

		if (this.lastEnemyMoveDir == “LEFT”) {
			this.setEnemyDirection(“RIGHT”);
		}
		else if (this.lastEnemyMoveDir == “RIGHT”) {
			this.setEnemyDirection(“LEFT”);
		}
	}
}
```

Then, add the method updatePlayerMovement for updating our player’s movement:

```
updatePlayerMovement() {
	this.time.addEvent({
		delay: 60,
		callback: function() {

			if (this.keyA.isDown) {
				this.player.x -= 8;
			}

			if (this.keyD.isDown) {
				this.player.x += 8;
			}

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

Now let’s finish up the methods relating to the player by adding the updatePlayerShooting method:

```
updatePlayerShooting() {
	this.time.addEvent({
		delay: 15,
		callback: function() {
			if (this.keySpace.isDown && this.player.active) {
				if (this.playerShootTick < this.playerShootDelay) {
					this.playerShootTick++;
				}
				else {
					var laser = new PlayerLaser(this, this.player.x,
this.player.y);
					this.playerLasers.add(laser);

					this.sfx.laserPlayer.play();

					this.playerShootTick = 0;
				}
			}
		},
		callbackScope: this,
		loop: true
	});
}
```

Let’s add our method to create a timer updating laser positions:

```
updateLasers() {

}
```

Then, add the following code to it:

```
this.time.addEvent({
	delay: 30,
	callback: function() {
		for (var i = 0; i < this.playerLasers.getChildren().length; i++) {
			var laser = this.playerLasers.getChildren()[i];

			laser.y -= laser.displayHeight;

			if (laser.y < 16) {
				this.createExplosion(laser.x, laser.y);
```

```
                        if (laser) {
                                laser.destroy();
                        }
                }
        }
        },
        callbackScope: this,
        loop: true
});

this.time.addEvent({
        delay: 128,
        callback: function() {
                for (var i = 0; i < this.enemyLasers.getChildren().length; i++) {
                        var laser = this.enemyLasers.getChildren()[i];

                        laser.y += laser.displayHeight;
                }
        },
        callbackScope: this,
        loop: true
});
```

The next method we need to add is addShield. This method is pretty self-explanatory. It adds a shield at the position specified. Essentially, we are iterating through the shield pattern arrays we added in the create method. If any given value in an array is a 1, then we add a shield tile. Add the following to create the addShield method and corresponding code:

```
addShield(posX, posY) {
        for (var y = 0; y < this.shieldPattern.length; y++) {
                for (var x = 0; x < this.shieldPattern[y].length; x++) {
                        if (this.shieldPattern[y][x] == 1) {
                                var tile = new ShieldTile(
                                        this,
                                        posX + (x * 8),
                                        posY + (y * 8)
                                );
                                this.shieldTiles.add(tile);
                        }
                }
        }
}
```

Now we can add the following method, destroyShieldTile:

```
destroyShieldTile(tile) {

}
```

Inside this method, we will be creating an explosion, then rendering multiple black rectangles to make broken areas of the shield look, well, broken, then we destroy the tile. Add the following code to this method:

```
if (tile) {
	this.createExplosion(tile.x, tile.y);

	for (var i= 0; i < Phaser.Math.Between(10, 20); i++) {
		var shieldHole = this.add.graphics({
			fillStyle: {
				color: 0x000000
			}
		});
		shieldHole.setDepth(-1);

		var size = Phaser.Math.Between(2, 4);

		if (Phaser.Math.Between(0, 100) > 25) {
			var rect = new Phaser.Geom.Rectangle(
				tile.x + (Phaser.Math.Between(-2, tile.displayWidth + 2)),
				tile.y + (Phaser.Math.Between(-2, tile.displayHeight + 2)),
				size,
				size
			);
		}
		else {
			var rect = new Phaser.Geom.Rectangle(
				tile.x + (Phaser.Math.Between(-4, tile.displayWidth + 4)),
				tile.y + (Phaser.Math.Between(-4, tile.displayHeight + 4))
			);
		}

		shieldHole.fillRectShape(rect);

		this.shieldHoles.add(shieldHole);
```

```
        }
        tile.destroy();
}
```

We're getting down to our last few methods we have to add! Let's add the createExplosion method, and add the following lines to it to instantiate an explosion:

```
createExplosion(x, y) {
        this.sfx.explode.play();

        var explosion = new Explosion(this, x, y);
        this.explosions.add(explosion);
}
```

Now, let's define the createLivesIcons method. This method will create the sprites representing the amount of lives the player has left. Add the following to create this method and add the following code to it:

```
createLivesIcons() {
        for (var i = 0; i < this.passingData.lives; i++) {
                var icon = this.add.sprite(
                        32 + (i * 32),
                        this.game.config.height - 24,
                        "sprPlayer"
                );
                icon.setScale(2);
                icon.setDepth(5);
        }
}
```

Finally, we will add the onLifeDown method. This method will be called when the player is hit by an enemy or enemy laser. Add the following to declare the onLifeDown method and associated code:

```
onLifeDown() {
        if (this.passingData.lives === 0) {
                this.textLabelScore.setVisible(false);
                this.textScore.setVisible(false);
                this.textLabelHighScore.setVisible(false);
                this.textHighScore.setVisible(false);

                this.textGameOver = this.add.text(
                        this.game.config.width * 0.5,
```

```
                128,
                “GAME OVER”,
                {
                        fontFamily: “Arcadepix”,
                        fontSize: 60,
                        align: “center”
                }
        );
        this.textGameOver.setOrigin(0.5);
    }

    if (this.passingData.score > localStorage.getItem(“highScore”)) {
        localStorage.setItem(“highScore”, this.passingData.score);
    }

    this.time.addEvent({
        delay: 3000,
        callback: function() {
            if (this.passingData.lives > 0) {
                this.passingData.lives--;

                this.scene.start(“SceneMain”, this.passingData);
            }
            else {
                this.scene.start(“SceneMain”, { });
            }
        },
        callbackScope: this,
        loop: false
    });
}
```

We are now finished with SceneMain.js! Let’s navigate to our SceneMainMenu.js file and declare our class:

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

Next, add the preload method to load our main menu content. Add the loading content for our play button and button sound effect:

```
preload() {
	this.load.image(“sprBtnPlay”, “content/sprBtnPlay.png”);
	this.load.image(“sprBtnPlayHover”, “content/sprBtnPlayHover.png”);

	this.load.audio(“sndBtn”, “content/sndBtn.wav”);
}
```

Then, add our create method and create our sound effects object:

```
create() {
	this.sfx = {
		btn: this.sound.add(“sndBtn”)
	};
}
```

After declaring our sound effects object, let’s add the text for displaying our title:

```
this.textTitle = this.add.text(
	this.game.config.width * 0.5,
	64,
	“SAUCER INVADERS”,
	{
		fontFamily: “Arcadepix”,
		fontSize: 32,
		align: “center”
	}
);
this.textTitle.setOrigin(0.5);
```

Next, we can add the code for the play button. The play button will just be a Phaser sprite with interaction enabled. Add the following code for instantiating our button:

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

Finally, we need to add some pointer events to our play button:

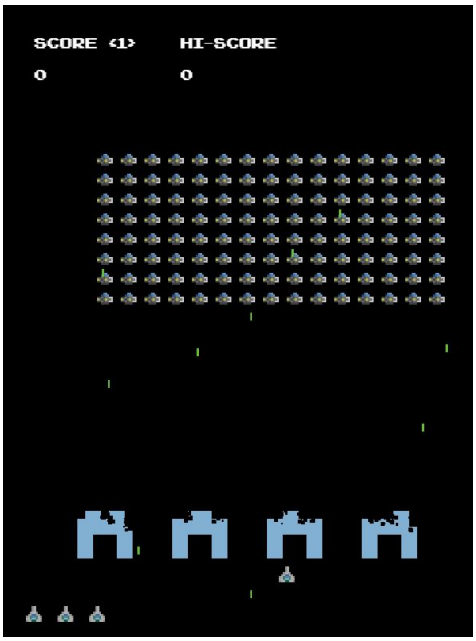
```
this.btnPlay.on(“pointerover”, function() {
```

```
        this.sfx.btn.play();
        this.btnPlay.setTexture("sprBtnPlayHover");
}, this);

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

this.btnPlay.on("pointerdown", function() {
        this.sfx.btn.play();
        this.scene.start("SceneMain");
}, this);

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

Depending how the mouse moves over the button, the texture will change appropriately.

Before we run the game, we need to insert SceneMainMenu before SceneMain in the scenes array of our game.js file.

If we navigate to the game in the browser, we should see our basic main menu, and a playable game. As always, do experiment tweaking any of the values. That's the great thing about tutorials and courses. You can explore, experiment, tweak, and modify anything to your heart's desire.



If you found this course valuable, I would love to hear from you! Please don't hesitate to contact me if you have any questions, comments, suggestions, etc. My email is jared.york@yorkcs.com and my Twitter handle is @jaredyork_. I would be more than happy to help if you have any issues while going through this course. The full source code for this course can be found on GitHub. If you found this helpful, be sure to fill out the form.

As of March 12, 2019, these are the other courses in the "Build Arcade Games with Phaser 3" series:

- "Build Arcade Games with Phaser 3: Table Tennis"
- "Build Arcade Games with Phaser 3: Brick Break"
- "Build Arcade Games with Phaser 3: Space Boulders"
- "Build Arcade Games with Phaser 3: Wormy"

Perhaps you may be interested in the rest of this course series if you enjoyed this course!