Build Arcade Games with Phaser 3: Brick Break
Version 1.0
Written by Jared York

In this course, we will traverse through the challenges of building a game similar to the arcade game, *Super Breakout*. However, we will call our game, "Brick Break" for the purposes of this course. Before we begin, it's highly recommended that you have intermediate knowledge of JavaScript. If not, no worries, I have written a free course, "JavaScript Beginner Blocks", which will help bring you up to speed. After you've went through that course, I highly recommend taking a look at my other course, "Build Arcade Games with Phaser 3: Table Tennis". In that course, you will explore the fundamentals of how to take advantage of the many features Phaser has. This will be useful because we will be utilizing much of the functionality Phaser provides us. Without further ado, let's jump right in!

Ensure a Web Server is Set Up

Although Phaser games are ran in the browser, unfortunately you can't run local HTML files directly from your file system. When requesting files over http, the security of the server only allows you to access 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 would be very bad to allow code on a website to read anything from your raw file system. Because of this, we will need to host our game on a local web server.

We highly 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 detailed summaries on each of the 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 ww 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 like.

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

We can now create two new 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 two folders anything you wish. One of the folders just needs to be dedicated to the content for our game, and the other for the JavaScript files. Once we have our folders for content and JavaScript, create four new files inside the newly created folder for JavaScript called: SceneMainMenu.js, SceneMain.js, Entities.js, and game.js. Shortly, I will go through what each of these files will contain.

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

(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 did prepare some assets for this course that can be freely downloaded here. Otherwise, you can create your own assets if you wish.

Content needed:

- Sprites (images)
  - sprBall.png
  - sprBrick.png
  - sprPaddle.png
  - sprWall.png
  - sprCeiling.png
- Sounds (.wav files)
  - sndBrickHit0.wav
  - sndBrickHit1.wav
  - sndBrickHit2.wav
  - sndBrickHit3.wav
  - sndWallHit.wav

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

Finally, the last thing we need to do before diving into the code is download the latest Phaser script. One common method for acquiring this script, is to head over to GitHub (specifically here). You will want either phaser.js or phaser.min.js. The file phaser.js contains the source code for Phaser in readable form, which is useful for contributing to the project, as well as understanding how features are 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 doesn't really matter which one we download. Simply decide between the two and 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. A dropdown menu should appear, then click "Save Page As". A save dialog will appear where you should save the Phaser script in the JavaScript directory we created previously.

Now we can jump into the code. Open index.html with your 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>CourseBrickBreak</title>
	</head>

	<body>
		<script src=”js/phaser.js”></script>

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

Now that our HTML file is finished, we can head over to game.js. Add the following to game.js to create an Phaser game instance, and provide it some configuration properties.

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

That’s it for our game.js file! We can now move over to Entities.js and define the entities for our game. We will be defining our game entities with ES6 classes. A class is just a normal,

ordinary object, with some syntactic sugar sprinkled on. Really, classes are just an easier way to keep code organized, while maintaining code in an object-oriented style. We will be create four classes: Brick, Wall, Ball, and Player. All of these classes will be extending, or "building on top" of Phaser's sprite object. In the Entities.js file, add the following code to define our classes, and their functions:

```
class Brick extends Phaser.GameObjects.Sprite {
        constructor(scene, x, y, color) {

        }

        setColor(color) {

        }

        setBreakable(bool) {

        }

        setPointValue(amount) {

        }

        setSoundIndex(index) {

        }
}

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

        }
}

class Ball extends Phaser.GameObjects.Sprite {
        constructor(scene, x, y) {

        }
}

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

```
        }
}
```

We will also be adding two functions to convert an RGB color to HEX format. We will be using this function when we need the HEX color value for coloring our bricks. Add the following to the bottom of our Entities.js file:

```
function rgbComponentToHex(c) {
        var hex = c.toString(16);
        return hex.length == 1 ? “0” + hex : hex;
}

function rgbToHex(r, g, b) {
        return rgbComponentToHex(r) + rgbComponentToHex(g) + rgbComponentToHex(b);
}
```

Let’s now fill in our classes from top to bottom. Starting with the Brick class, add the following to the constructor:

```
super(scene, x, y, “sprBrick”);
this.scene = scene;
this.scene.add.existing(this);
this.scene.physics.world.enableBody(this, 0);
this.body.setImmovable(true);

this.setPointValue(1);
this.setBreakable(true);
this.color = color;
this.setTint(“0x” + rgbToHex(this.color.r, this.color.g, this.color.b));
```

In the above block, we are filling in the arguments normally specified when adding a regular Phaser sprite. We accomplish this with the super keyword. When we call the add.existing function, instances of the Brick class will automatically be added to the display stack for rendering. We then enable the physics body, which we will be utilizing later when adding collisions. Then, we’re setting bricks to be immovable. We don’t want the ball to be able to move the bricks when it collides. After that, we’re calling the setPointValue method we defined above. We’ll be adding to that method shortly. We also assign the color we take in as a parameter to a property. Finally, we set the color of the brick after converting the RGB color we have to HEX as I mentioned previously. Normally when the setTint method is called, it would normally be used as follows:

```
brick.setTint(0xFFF056);
```

When defining hexadecimal numbers with JavaScript you add a “0x” in front of the value. We use string concatenation to add the string “0x” in front of the return value of the rgbToHex function. The purpose of the next method we’ve added to our class, setColor, is to be able to change the color after of a brick even if it’s already been instantiated. Add the following to the setColor method:

```
this.color = color;
this.setTint(“0x” + rgbToHex(this.color.r, this.color.g, this.color.b));
```

We have defined another method, similar to setColor, which sets whether a brick is breakable or not. In the setBreakable method, add:

```
this.setData(“isBreakable”, bool);
```

After that, we will finish by adding similar lines for the setPointValue and setSoundIndex methods. In the setPointValue method, add the following line:

```
this.setData(“pointValue”, amount);
```

Finally in the setSoundIndex method add:

```
this.setData(“soundIndex”, index);
```

That concludes our Brick class! Now we can move down to the Wall class. We will add code to this constructor not unlike that which we added to the Brick class. Add the following to the Wall constructor:

```
super(scene, x, y, key);
this.scene = scene;
this.scene.add.existing(this);
this.scene.physics.world.enableBody(this, 0);
this.body.setImmovable(true);
this.setOrigin(0);
this.setTint(0xbbbbbb);
```

The only line we haven’t encountered yet in the above code is the setOrigin method. Sprites are placed at their x and y positions relative the origin set. We are specifying that we want to be able to place walls relative to the top-left corner of their sprite. This is all the code needed for the Wall class.

Moving on to the Ball class, the code for the constructor will be similar to the Wall class but with a couple changes:

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

this.body.setBounce(1);

this.setData(“velocityMultiplier”, 5);
```

In addition to the usual constructor code, we also set the ball to be able to bounce, and set a bit of data that will be used for speeding up the ball. That will conclude the code for the Ball class. We can finish up by adding to the constructor of the Player class:

```
super(scene, x, y, “sprPaddle”);
this.scene = scene;
this.scene.add.existing(this);
this.scene.physics.world.enableBody(this, 0);
this.body.setImmovable(true);

this.setOrigin(0.5);

this.setTint(“0x” + rgbToHex(160, 160, 255));
```

When we call the setOrigin method above, we are setting the origin of the player paddle to be in the center of the sprite. After that, we set the tint of the player to a hexadecimal color representing light blue. Feel free to change the value to any color you like. With that, we are finished with our Entities.js file! We can now move on to the real meat-and-potatoes in the SceneMain.js file.

We will be creating a class to represent our scene in the SceneMain.js file. Add the following to our empty file to declare the class:

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

	preload() {

	}

	addScore(amount) {
```

```
        }

        createStage() {

        }

        addRowsToQueue(amount, isEmpty) {

        }

        generateRows(amount) {

        }

        moveBricksDown(amount) {

        }

        getLowestRowY() {

        }

        create() {

        }

        update() {

        }
}
```

Let's first start filling in our class by loading our content. In the preload method, add the following to load our images:

```
this.load.image("sprWall", "content/sprWall.png");
this.load.image("sprCeiling", "content/sprCeiling.png");
this.load.image("sprBrick", "content/sprBrick.png");
this.load.image("sprPaddle", "content/sprPaddle.png");
this.load.image("sprBall", "content/sprBall.png");
```

Then, let's add the loading code for our sounds:

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

for (var i = 0; i < 4; i++) {
        this.load.audio("sndBrickHit" + i, "content/sndBrickHit"  + i + ".wav");
}
```

Since we have four sounds that can be played when the ball hits a brick, we can just use a for loop to load all four sounds.

The next method, addScore will add the specified amount to the player's score. Add the following to that method:

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

The next function, createStage, will be very important in creating the stage for our game. We need to create the walls and ceiling. We will be creating the walls group later, which will contain instances of the Wall class. Let's add the code for our createStage function:

```
var wallLeft = new Wall(this, 0, 0, "sprWall");
this.walls.add(wallLeft);

var wallRight = new Wall(this, this.game.config.width - 32, 0, "sprWall");
this.walls.add(wallRight);

var wallCeiling = new Wall(this, 32, 0, "sprCeiling");
this.walls.add(wallCeiling);
```

Since the ceiling should have the same properties as a wall, we'll just say it's a wall for the sake of simplicity.

Now, the way we will add bricks to our stage is via a queue system. The queue system works by adding row objects to an array which will be the queue. Then, when we want to generate the bricks, we can use the properties of the next row object in the queue to create the bricks. Once a row of bricks has been created, we remove that row from the queue. In the classic game Super Breakout, there is a game mode called, "Progressive". We will be attempting to recreate something similar to that game mode, but I will lightly touch on the other modes as well. In order to implement a system similar to the "Progressive" game mode, we will be using a timer to generate bricks from the queue. The timer will allow the bricks to move down the screen periodically, as well as generate a new row of bricks at the same time.

To create our queue system, let's add the following to the addRowsToQueue method:

```
isEmpty = isEmpty | false;
```

```
for (var i = 0; i < amount; i++) {
	if (isEmpty) {
		this.rowQueue.push({
			isEmpty: true,
			canRemove: false
		});
	}
	else {
		var pointValue = (i + 1) * 3;

		this.rowQueue.push({
			pointValue: pointValue,
			soundIndex: i,
			canRemove: false
		});
	}
}
```

The next step is to add the logic to generate a specified amount of bricks. To generate rows, we will add the following code to the generateRows function:

```
amount = amount | 1;

for (var i = 0; i < amount; i++) {
	if (this.rowQueue.length > 0) {

		if (!this.rowQueue[0].isEmpty) {
			for (var x = 0; x < (this.game.config.width / 32) - 2; x++) {



			}
		}

		this.rowQueue[0].canRemove = true;
		this.amountRowsGenerated++;

		this.rowQueue = this.rowQueue.filter(function(row) {
			return !row.canRemove;
		});

		this.moveBricksDown(1);
```

```
        }
}
```

We have yet to fill in the second for loop, but the above code does a sequence of things. First, we set the amount of rows we want to generate to one if the parameter is null or undefined. The next is repeating our row generation code the amount of times specified as the parameter. Each time we generate a row, we first check if there are row objects in the queue. If there are row objects in the queue, make sure the first one is not empty. If the row object represents an empty row, we will simply ignore it and not generate the row. However, if the row is not empty, we will create the bricks spanning across the width of the screen. We are leaving a one brick width gap on both sides of the screen so there is room for the walls. In order to create our bricks, we will have to add some code inside the second for loop:

```
var color = { r: 0, g: 0, b: 0 };

var freq = 0.35;

color.r = Math.round(Math.sin(freq * (this.amountRowsGenerated + i * 0.5) + 0) * 127 + 128);
color.g = Math.round(Math.sin(freq * (this.amountRowsGenerated + i * 0.5) + 2) * 127 + 128);
color.b = Math.round(Math.sin(freq * (this.amountRowsGenerated + i * 0.5) + 3) * 127 + 128);
```

The above code block calculates the color the row of bricks should be. Tweak any of the values you like. This code should create a sort of rainbow effect. Continue adding the code below after the code above:

```
var brick = new Brick(this, 32 + (x * 32), 0, color);
brick.x += brick.displayWidth * 0.5;
brick.y += brick.displayHeight * 0.5;
brick.setPointValue(this.rowQueue[0].pointValue);
brick.setSoundIndex(this.rowQueue[0].soundIndex);
this.bricks.add(brick);
```

Next, we can move on to our moveBricksDown function. This function is fairly self-explanatory. It, well, moves the bricks down that are breakable. Add the following to the function:

```
amount = amount | 1;

for (var i = 0; i < amount; i++) {
        for (var j = 0; j < this.bricks.getChildren().length; j++) {
                var brick = this.bricks.getChildren()[j];

                if (brick.getData("isBreakable")) {
                        brick.y += brick.displayHeight;
```

```
                }
        }
}
```

We also need to add function to grab the y position of the lowest row of bricks. To do so, let's add to our getLowestRowY function:

```
var lowest = 0;

for (var i =  0; i < this.bricks.getChildren().length; i++) {
        var brick = this.bricks.getChildren()[i];
        if (brick.y > lowest) {
                lowest = brick.y;
        }
}

return lowest;
```

Now, let's move down to the create function. This is where we set the ball rolling to get our game to actually do things. First, we should create a property that will store our sound objects for each of our sounds. Creating a single property that contains our sounds will make it easy to organize our sounds into categories without cluttering the class. Add the following code to add our sound property and enclosed sounds:

```
this.sfx = {
        wallHit: this.sound.add("sndWallHit"),
        brickHit: [
                this.sound.add("sndBrickHit0"),
                this.sound.add("sndBrickHit1"),
                this.sound.add("sndBrickHit2"),
                this.sound.add("sndBrickHit3")
        ]
};
```

Since we only have a single sound for when the ball hits a wall, we can simply assign the sound to the property, "wallHit". Since we have four sounds for when the ball hits a brick, we can store them in an array. If we want to reference a sound from the array assigned to the brickHit property, we can call:

```
this.sfx.brickHit[array index]
```

The next line of code we want to add is for creating the player. Add the following line to do so:

```
this.player = new Player(this, this.game.config.width * 0.5, this.game.config.height - 48);
```

Then we can add the ball:

```
this.ball = new Ball(this, this.player.x, this.player.y - 32);
this.ball.body.setVelocity(
        Phaser.Math.Between(-50, 50),
        300
);
```

We will need to also add two groups so we can store all of our bricks and walls:

```
this.bricks = this.add.group();
this.walls = this.add.group();
```

The next step is to create our array for the queue of brick rows. At the same time, we will need to add two more properties. The first property will be for keeping track of the amount of total rows that have been generated. The second property will represent the number of times the ball hit the paddle. Add the following code to add all three properties:

```
this.rowQueue = [];
this.amountRowsGenerated = 0;
this.amountBallHitPaddle = 0;
```

We can then call the function that creates the game's stage, createStage:

```
this.createStage();
```

Then, let's create our two properties that will be used for keeping track of the score and lives used:

```
this.score = 0;
this.livesUsed = 0;
```

After that, we can add the following code to create our text objects for displaying the score and number of lives used.

```
this.textScore = this.add.text(
        64,
        this.game.config.height - 24,
        0,
        {
                fontFamily: “monospace”,
```

```
                fontSize: 24,
                align: “left”
        }
);
this.textScore.setOrigin(0.5);

this.textLivesUsed = this.add.text(
        this.game.config.width * 0.5,
        this.game.config.height - 24,
        0,
        {
                fontFamily: “monospace”,
                fontSize: 24,
                align: “center”
        }
);
this.textLivesUsed.setOrigin(0.5);
```

If you wanted to just create a stage similar to a class Breakout game, feel free to add the following three lines and skip the timer we’ll cover coming up:

```
this.addRowsToQueue(8, false);
this.addRowsToQueue(2, true);
this.generateRows(10);
```

If you would like to recreate something similar to the “Progressive” gamemode of *Super Breakout*, feel free to add the following code:

```
this.rowGeneratorTimer = this.time.addEvent({
        delay: 1,
        callback: function() {

                if (this.amountRowsGenerated == 0) {
                        this.addRowsToQueue(4, false);
                        this.addRowsToQueue(4, true);
                        this.addRowsToQueue(4, false);

                        this.generateRows(12);

                        this.rowGeneratorTimer.delay = 5000;
                }
                else {
```

```
                        if (this.amountRowsGenerated % 2 == 0) {
                                this.addRowsToQueue(4, true);
                        }
                        else {
                                this.addRowsToQueue(4, false);
                        }

                        this.generateRows(1);
                }
        },
        callbackScope: this,
        loop: true
});
```

The above code will spawn a new row of bricks at the top of the screen in intervals of four layers of bricks, four layers of air, etc. After we added the timer, let's add our collisions. We will want to use what's called a collider. Colliders in Phaser allow you to check between two game objects. Sprites and groups are game objects in Phaser.

First we will want to add a collider between the ball and the player paddle. Add the following code after the timer, or the call to generateRows:

```
this.physics.add.collider(this.ball, this.player, function(ball, player) {

}, null, this);
```

Currently the ball will bounce off the player's paddle without taking the distance from the center of the paddle into account. We want to add some extra logic to the callback of the above collider. Add the following code inside the collider callback from above:

```
var dist = Phaser.Math.Distance.Between(player.x, 0, ball.x, 0) * 2;

if (ball.x < player.x) {
        dist = -dist;
}

var velocityMultiplier = Math.abs(ball.body.velocity.y * 0.1);

if (velocityMultiplier > 8) {
        velocityMultiplier = 8;
}

ball.body.velocity.x = dist * velocityMultiplier;
```

```
this.amountBallHitPaddle++;

if (this.amountBallHitPaddle == 8 ||
   this.amountBallHitPaddle == 16 ||
   this.amountBallHitPaddle == 48) {

        ball.setData("velocityMultiplier", ball.getData("velocityMultiplier") * 1.1);
        ball.body.velocity.x += Math.sign(ball.body.velocity.x) + 0.001;
        ball.body.velocity.y += Math.sign(ball.body.velocity.y) + 0.001;
}

this.sfx.brickHit[Phaser.Math.Between(0, this.sfx.brickHit.length - 1)].play();
```

We will need to add another collider after the previous collider. This new collider will be checking for collisions between the ball and the bricks. In the following code, when a ball collides with a brick, if the brick is breakable, add to the score the point value stored in the brick, play a sound, then destroy the brick. Add this code block under the last collider:

```
this.physics.add.collider(this.ball, this.bricks, function(ball, brick) {
        if (brick.getData("isBreakable")) {
                if (brick) {
                        this.addScore(brick.getData("pointValue"));

                        if (brick.getData("soundIndex") !== undefined) {
                                if (this.sfx.brickHit[brick.getData("soundIndex")] !== undefined) {
                                        this.sfx.brickHit[brick.getData("soundIndex")].play();
                                }
                        }

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

After adding this code, we will add another collider that will check between the ball and the walls. The only thing the callback in this next collider will contain is playing the wall bounce sound. Add the following code:

```
this.physics.add.collider(this.ball, this.walls, function(ball, wall) {
	this.sfx.wallHit.play();
}, null, this);
```

With that, the create method is finished! Fantastic! Let's move on to the update method. The update method will contain much less code than the create method. In the update method we will set the x position of the player paddle to the value of the x position of the active pointer. Using the mouse to control the player's paddle will be much more precise than trying to play the game with the keyboard. Add the following code to set the player's x position to the active pointer's x position:

```
this.player.x = this.input.activePointer.x;
```

We then want to check if the ball has fallen offscreen. If the ball has fallen offscreen, we want to reset the position of the ball at the y position just below the bottom row of bricks. Then we want to increment the livesUsed property. We will then set the text of textLivesUsed to the newly updated livesUsed property. Add the following code to add this condition:

```
if (this.ball.y > this.game.config.height) {
	var respawnPosition = new Phaser.Math.Vector2(this.player.x, this.getLowestRowY() +
64);

	if (respawnPosition.y > this.player.y - 32) {
		respawnPosition.y = this.player.y - 32;
	}

	this.ball.x = respawnPosition.x;
	this.ball.y = respawnPosition.y;

	this.livesUsed++;
	this.textLivesUsed.setText(this.livesUsed);
}
```

After adding this condition, we will want to add some logic to destroy bricks close to the player. Let's add the following code block:

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

	if (brick.y > this.player.y - 48) {
		if (brick) {
			brick.destroy();
```

```
                }
        }
}
```

The last thing we will do is create the main menu. Let's head over to our SceneMainMenu.js file. We can also define our SceneMainMenu class as follows:

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

        preload() {

        }

        create() {

        }
}
```

Next, we will add our code to load our play button and sounds. 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");
```

The next step we have to do is add our sfx object which will contain the button sound. Add the following to the create method:

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

We can also add our button sprite to the create method too:

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

We can also set the button sprite as being interactable and add some pointer events:

```
this.btnPlay.setInteractive();

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.btnPlay.setTexture(“sprBtnPlay”);
        this.scene.start(“SceneMain”);
}, this);
```

Let’s add the title as well, while we’re at it:

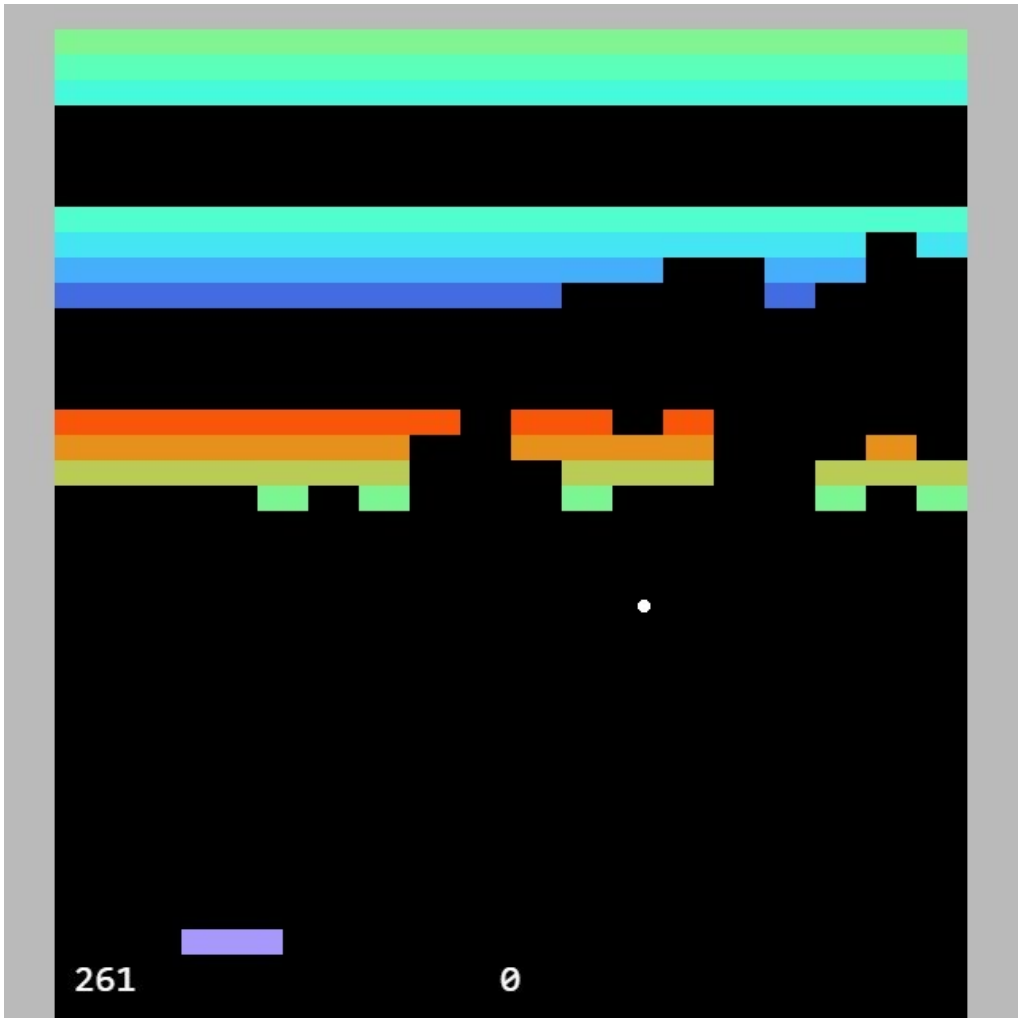
```
this.title = this.add.text(this.game.config.width * 0.5, 128, “BRICK BREAK”, {
        fontFamily: “monospace”,
        fontSize: 48,
        fontStyle: “bold”,
        color: “#ffffff”,
        align: “center”
});
this.title.setOrigin(0.5);
```

Finally, in the game.js file, let’s add a reference to SceneMainMenu right before SceneMain in the scenes array of the config object. The final game.js file should look like:

```
var config = {
        type: Phaser.WEBGL,
        width: 640,
        height: 640,
        background: “black”,
```

```
        physics: {
                default: “arcade”,
                arcade: {
                        gravity: { x: 0, y: 0 }
                }
        },
        scene: [
                SceneMainMenu,
                SceneMain
        ],
        pixelArt: true,
        roundPixels: true
};

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



With that, this concludes the course, “Build Arcade Games with Phaser 3: Brick Break”! You should have a functional main menu, and playable game. There are a few things that I will leave as an exercise for the reader such as:

- Trigger a game over after five (or however many) lives
- Multiple balls
- Multiple paddles
- Exploding bricks
- Increase difficulty over time
- Powerups
- Add the proper point values to each row of bricks
- Cool(er?) main menu

This course should give a good idea what can be done with Phaser. We’ve only scratched the surface, but the hope is you have some tools to be able to create your own awesome games! As usually, I would like to thank the Phaser team for making such a great HTML5 game framework. You can learn even more about Phaser on their website. If you run into any difficulties, or have any questions or feedback, I would love to hear it! You can tweet at me, at my handle @jaredyork. I can also contacted via my email, jared.york@yorkcs.com. The final source code for this project is available on GitHub.

If you wish to receive updates on future courses and tutorials I create, please fill out the form.