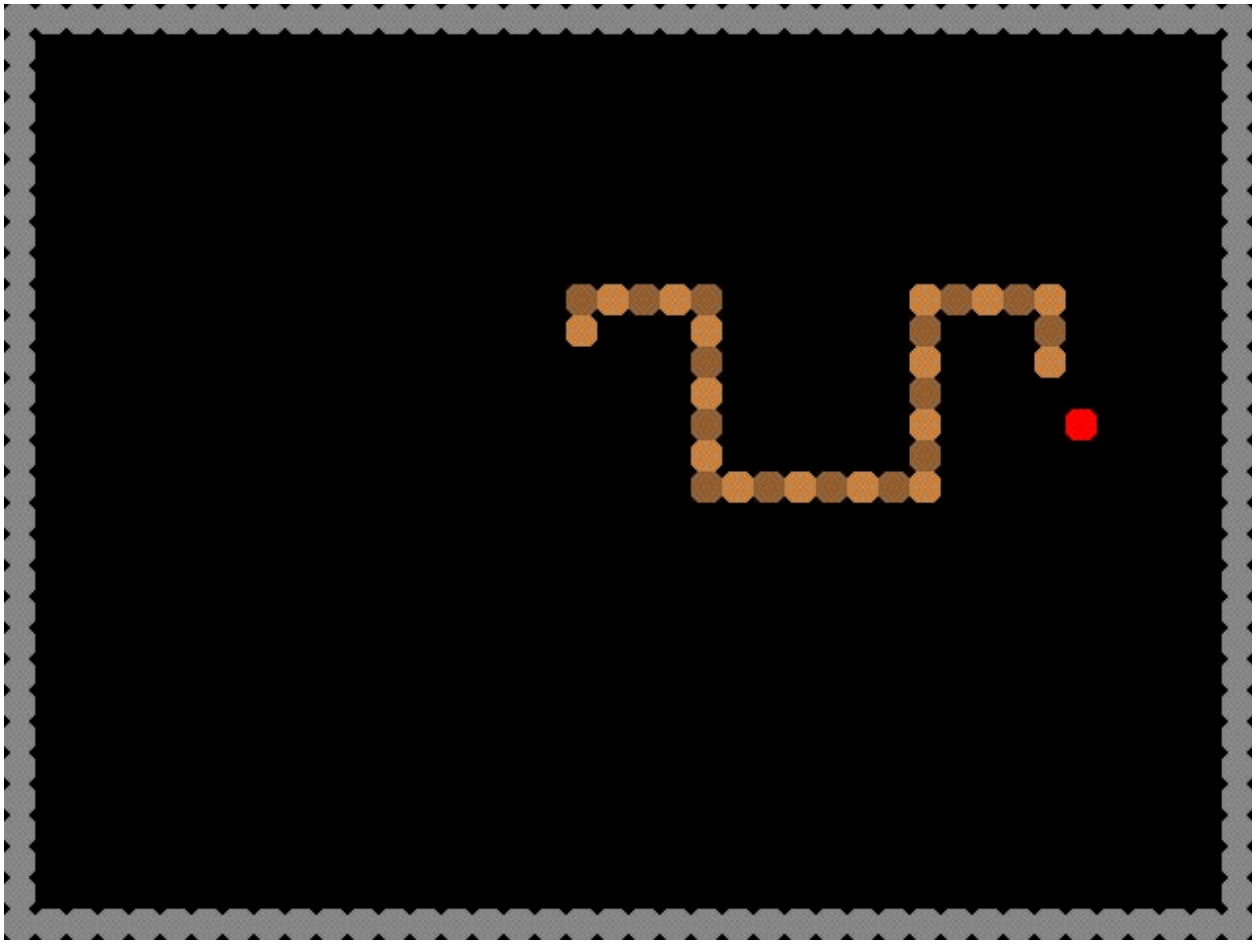
Build Arcade Games with Phaser 3: Wormy
Version 1.0
Written by Jared York

In this course, we will be creating a game similar to games in the popular Snake genre. For the sake of this tutorial, we will call this game “Wormy.”



Course Requirements:

- Basic to intermediate knowledge of JavaScript
- Code editor (not necessarily required, but highly recommended)
- Web Server
- Tutorial assets (download link can be found in the text)

Ensure a Web Server is Set Up

Although Phaser games are ran in the browser, unfortunately you can’t just run a local HTML file directly from your file system. When requesting files over http, the server’s security only allows you to access files you’re allowed ot. When loading a file form the local fiule system (the file:// protocol), your browser highly restricts it for obvious security reasons. It’s would be no good to allow 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 recommend checking out Phaser's official guide, "Getting Started with Phaser 3," to learn which web server is compatible with your system. There are summaries and links for each one.

Create the Folders and Files Needed

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

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

Now, we will also have to 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 Phaser script and our other game scripts. Feel free to name these two folders and think you like. Once we have our folder for content and JavaScript, create four new files inside the JavaScript folder called: SceneMainMenu.js, SceneMain.js, and game.js. I will explain what those files do shortly, first we need to populate our content folder with the content for our game. It's kind of hard to play a game if there's nothing to see. :)

So far our file structure should look like so:

(game folder)/
|_ index.html
|_ content/
|_js/
  |_ game.js
  |_ SceneMain.js
  |_ SceneMainMenu.js

In order to add the content to our game, we first need content to add. I have made some assets available for this course here, which will work great for our needs. Otherwise, you may create your own content as you see fit. Below is a list of content we need:

Content needed:

- Sprites (images)
  - sprBtnPlay.png (the play button)
  - sprBtnPlayHover.png (the play button when hovered over by mouse)
  - sprTile.png (a white square-ish image used to represent game objects)
- Sounds (.wav files)
  - sndBtn.wav
  - sndEat.wav
  - sndHit.wav

Once you have acquired assets, we will move those files into the content directory for our game.

Finally, before we dive into the code, we will need to download the latest Phaser script. One method of acquiring this (there are a few), is heading 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, or just taking a look under-the-hood. The other file, phaser.min.js, is meant for distribution, and is compressed to reduce file size. For our purposes, it won't really matter which you pick, so just decide. Then, click the link for the appropriate script, and you should see a new page with a "View raw" link near the center of the page. Click the "View raw" link, then when the page of code appears, right click the "Save Page As" or similar option when the context menu appears. A save dialog will appear, then you can save the script to the JavaScript directory we created earlier.

Now we can jump right into the code! Open up your code editor or text editor and navigate to our newly created index.html file. Add the following to it in order to declare our HTML document and link our scripts:

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

Now, we can navigate to our game.js file. In the game.js file, we will define some properties used for configuring our Phaser game when we instantiate it. Add the following to game.js:

```
var config = {
	type: Phaser.WEBGL,
	width: 640,
	height: 480,
	backgroundColor: “black”,
```

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

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

Next, we can move on to the Entities.js file! In this file, we will be declaring five classes: Entity, Wall, Food, BodySegment, and HeadSegment. A class is just an ordinary JavaScript object, but with some ES6 syntactic sugar sprinkled on top. First, we will have to declare a base entity class that will extend, or build on top of Phaser’s sprite object. When entities are created in our game world, we also want to add them to the display list, then enable their physics body. Add the following block to declare our class and the sequence of instructions I mentioned above:

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

The next class we will define is the Wall class. We will be creating many instances of the Wall class as a border around the game screen. The wall class will also be extending the Entity class we just created. Add the following:

```
class Wall extends Entity {
        constructor(scene, x, y, color) {
                super(scene, x, y, “sprTile”);
                this.body.setImmovable(true);
                this.setTint(color);
                this.setOrigin(0);
        }
}
```

Do notice, we are using the super keyword. The super keyword allows us to initialize the basic class we are inheriting. When we added scene, x, y, and key to the constructor in the Entity class, we will be able to access these arguments we just provided with the super keyword, in the Entity class.There are a couple new methods we are calling above. The method setTint will color our wall sprite to be any hexadecimal color we provide it. The next method, setOrigin is used for defining where relative to the top-left corner of the sprite the origin is. The origin is most commonly used for positioning the sprite and sprite rotation. We want to ensure for our wall that the origin remains in the top-left corner of the wall sprite.

Now we can move to the next class, Food. The Food class will be almost the same as the Wall class, except we will assign the color within the class. The hexadecimal value for red is #ff0000. The setTint method requires we define our hexadecimal color value without the pound sign, but instead with “0x” (without quotes.) So, we would use the value 0xff0000. Let’s create our Food class:

```
class Food extends Entity {
	constructor(scene, x, y) {
		super(scene, x, y, “sprTile”);
		this.body.setImmovable(true);
		this.setOrigin(0);
		this.setTint(0xff0000);
	}
}
```

Now, the player’s worm will be divided into segments. There will be two types of segments: head segments, and body segments. There will be one head segment, and more body segments will be progressively added as the player’s worm eats food. We will first start creating the worm segments by first defining the HeadSegment. The HeadSegment will extend our Entity base class. Add the following code block to define our class:

```
class HeadSegment extends Entity {
	constructor(scene, x, y, color) {
		super(scene, x, y, “sprTile”);
		this.scene = scene;
		this.body.setImmovable(true);
		this.setTint(color);
		this.setOrigin(0);
	}
}
```

We will also have to define some additional data. There are four types of data we will need to define: color, the move increment (how much the segment moves each time it moves), the

direction the head segment is facing, and the segment type (which in this case is head.) Add the following code to the constructor of the HeadSegment class to specify this data:

```
this.setData("color", color);
this.setData("moveIncrement", this.scene.tileSize);
this.setData("direction", "move");
this.setData("segmentType", "HEAD");
```

The next thing we will add to the constructor is the move timer. The move timer will allow our worm to move. First, we will move the head segment, then each body segment will update it's position to the last position of the segment ahead of it. We will be calling the addEvent method of the time subsystem of Phaser. We will provide the addEvent method with an object with the properties: delay, callback, callbackScope, and loop. Add the following code block to the constructor to add our move timer and the logic associated with it:

```
this.moveTimer = this.scene.time.addEvent({
        delay: 100,
        callback: function() {
                switch (this.getData("direction")) {
                        case "UP": {
                                this.moveUp();
                                break;
                        }

                        case "DOWN": {
                                this.moveDown();
                                break;
                        }

                        case "LEFT": {
                                this.moveLeft();
                                break;
                        }

                        case "RIGHT": {
                                this.moveRight();
                                break;
                        }
                }

                this.scene.updateSegments();
        },
        callbackScope: this,
```

```
        loop: true
});
```

That concludes the constructor of the HeadSegment class. We have to add five more methods: setDirection, moveUp, moveDown, moveLeft, and moveRight. The set direction method will be used for setting the direction of the head segment. The methods: moveUp, moveDown, moveLeft, and moveRight are being called in the move timer above. Add the following code block to add our remaining methods of the HeadSegment class:

```
setDirection(direction) {
        this.setData("direction", direction);
}

moveUp() {
        this.y -= this.getData("moveIncrement");
}

moveDown() {
        this.y += this.getData("moveIncrement");
}

moveLeft() {
        this.x -= this.getData("moveIncrement");
}

moveRight() {
        this.x += this.getData("moveIncrement");
}
```

Finally, we can add the BodySegment class. This class will be much more simple as it won't contain any movement logic. The HeadSegment class takes care of all the movement logic. Add the following code to create our BodySegment class:

```
class BodySegment extends Entity {
        constructor(scene, x, y, color) {
                super(scene, x, y, "sprTile");
                this.scene = scene;
                this.body.setImmovable(true);
                this.setTint(color);
                this.setOrigin(0);

                this.setData("color", color);
                this.setData("lastPosition", new Phaser.Math.Vector2(this.x, this.y));
```

```
                this.setData("segmentType", "BODY");
        }
}
```

We can now open up our SceneMain.js file. We will need to declare a Phaser scene called SceneMain. You can think of the scene in Phaser as a "screen" in a video game. For example, in many video games there is a main menu, a play state, a game over screen, etc. You can think of those different screens as states. We will be declaring our scene, SceneMain as a class which extends Phaser's Scene object. Add the following code to declare our scene:

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

Next, we will load our content. Add a method named, preload, to our SceneMain class with the following code to load our content:

```
preload() {
        this.load.image("sprTile", "content/sprTile.png");

        this.load.audio("sndEat", "content/sndEat.wav");
        this.load.audio("sndHit", "content/sndHit.wav");
}
```

Next, we can also add a method called generateWalls, which will generate our walls:

```
generateWalls() {

}
```

Then we can add the following code to it:

```
for (var x = 0; x < this.mapWidth; x++) {
        for (var y = 0; y < this.mapHeight; y++) {
                if (x == 0 || y == 0 ||
                   x == this.mapWidth - 1 || y == this.mapHeight - 1) {
                        var wall = new Wall(this, x * this.tileSize, y * this.tileSize, 0x888888);
                        this.walls.add(wall);
                }
        }
}
```

The next method we will add will be called addFood. The method addFood will add food in a random location within the walls. Add the following code to create the addFood method and the logic I just mentioned:

```
addFood() {
	var food = new Food(
		this,
		Phaser.Math.Between(1, this.mapWidth - 2) * this.tileSize,
		Phaser.Math.Between(1, this.mapHeight - 2) * this.tileSize
	);
	this.food.add(food);
}
```

We will also have to add a method that we can call to add additional segments. An interesting feature we can add is make every other segment slightly darker. This should make the worm look more like a worm! We can accomplish this by using the modulo operator, "%." Let's name this method, addSegment. To declare this method, add the following:

```
addSegment() {

}
```

Now, we can add this code block to the method:

```
var lastChild = this.playerSegments.getChildren()[this.playerSegments.getChildren().length - 1];

var color = this.getHeadSegment().getData("color");
if (this.playerSegments.getChildren().length % 2 == 0) {
	color  = 0x8c6130;
}

var segment = new BodySegment(
	this,
	lastChild.x,
	lastChild.y,
	color
);
this.playerSegments.add(segment);
```

Fantastic! Let's move on to the getHeadSegment method. This method will be used for retrieving the head segment from our group of segments later on. Add the following to fetch the head segment:

```
getHeadSegment() {
	var segment = null;
	for (var i = 0; i < this.playerSegments.getChildren().length; i++) {
		var head = this.playerSegments.getChildren()[i];
		if (head.getData(“segmentType”) == “HEAD”) {
			segment = head;
		}
	}
	return segment;
}
```

The next method we will need to add is setGameOver. This will, well, set the game as being over. Pretty much when the game is over, we want to play a hit sound effect, create text overlayed on the screen displaying “GAME OVER”, create a timer to remove segments from the worm, then add a timer to reset the game. First, let’s create our method and initial condition checking that the game has not already been set as being over:

```
setGameOver() {
	if (!this.isGameOver) {

	}
}
```

Next, we can play our sound effect inside the if statement (we will add the code to add our sounds to an easily accessible object, sfx, later on):

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

Let’s add the code to create our game over overlay text next, continue adding inside the if statement:

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

The cool thing about setting the origin with a single parameter, is it will set the specified value to both the x and y axis. Pretty much it's just a shorthand way to set the origin on both axis to the same value. Now, we will add our timer that will remove segments from the player, this will just be a cool visual effect, it has no practical use. Add this block inside the if statement:

```
this.time.addEvent({
	delay: 30,
	callback: function() {
		var lastSegmentIndex = this.playerSegments.getChildren().length - 1;

		if (lastSegmentIndex > 0) {
			var lastSegment = this.playerSegments.getChildren()[lastSegmentIndex];

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

We will finish up this method by adding the timer to reset the scene and set the game as over:

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

this.isGameOver = true;
```

The next method will be used for updating the position of the segments and testing for a collision between the head segment and any of the body segments. Add the following to declare our updateSegments method:

```
updateSegments() {

}
```

We can also add the following code inside the method to update the position of each segment to the last position of the segment in front:

```
for (var i = this.playerSegments.getChildren().length - 1; i > 0; i--) {
	var segment = this.playerSegments.getChildren()[i];

	var segmentInFront = this.playerSegments.getChildren()[i - 1];

	segment.setData(“lastPosition”, new Phaser.Math.Vector2(segment.x, segment.y));

	segment.setPosition(segmentInFront.x, segmentInFront.y);
}
```

After this block, let’s add in the for loop we will be using for detecting collisions between the head segment and body segments. This code should stay within the updateSegments method, but should be added after the for loop from above:

```
if (this.playerSegments.getChildren().length > 2) {
	var head = this.getHeadSegment();

	var headRect = new Phaser.Geom.Rectangle(
		head.x + 2,
		head.y + 2,
		head.displayWidth - 4,
		head.displayHeight - 4
	);

	for (var i = 0; i < this.playerSegments.getChildren().length; i++) {
		if (i > 1 &&
		this.playerSegments.getChildren()[i].getData(“segmentType”) == “BODY”) {
			var body = this.playerSegments.getChildren()[i];

			var bodyRect = new Phaser.Geom.Rectangle(
				body.x,
				body.y,
				body.displayWidth,
				body.displayHeight
			);

			if (Phaser.Geom.Intersects.RectangleToRectangle(
				headRect,
				bodyRect
```

```
                        )) {
                                this.setGameOver();
                        }
                }
        }
}
```

The last method we have to add to our SceneMain class is the create method. The create method is where we will create everything (hence the name.) First, let's create a object that we can reference our sound effects from, called sfx. This object will help keep our scene organized instead of littering our class with extra properties. Add the following to add our create method and sfx object:

```
create() {
        this.sfx = {
                eat: this.sound.add("sndEat"),
                hit: this.sound.add("sndHit")
        };
}
```

We will also define some properties that are used dynamically throughout the code. This allows us to change these properties anytime instead of hardcoding the values in the places these properties are referenced. Add the following to declare these properties:

```
this.isGameOver = false;
this.tileSize = 16;
this.mapWidth = Math.ceil(this.game.config.width / this.tileSize);
this.mapHeight = Math.ceil(this.game.config.height / this.tileSize);
```

We also need a way to store multiple segments of the player's worm, as well as a way to store multiple walls, and food. Phaser provides us a way to group game objects together with groups. Add the following to declare the three groups we need:

```
this.playerSegments = this.add.group();
this.walls = this.add.group();
this.food = this.add.group();
```

We can also call two of our methods: addFood, and generateWalls, to setup our game stage:

```
this.addFood();
this.generateWalls();
```

Next, we can create our head segment:

```
var playerHead = new HeadSegment(
        this,
        Math.round((this.game.config.width / this.tileSize) * 0.5) * this.tileSize,
        Math.round((this.game.config.height / this.tileSize) * 0.5) * this.tileSize,
        0xbf8543
);
this.playerSegments.add(playerHead);
```

We will also have to call addSegment to add our first body segment. We have to do this because when second segment updates it's position to the segment before (in this case the head), this first body segment will stay on top of the head segment no matter what. This way, each additional segment added will make the worm longer. Let's call our addSegment method:

```
this.addSegment();
```

The next step will be to add the keyboard events used to set the direction of the head segment:

```
this.input.keyboard.on("keydown_W", function() {
        if (playerHead.getData("direction") !== "DOWN") {
                playerHead.setDirection("UP");
        }
}, this);

this.input.keyboard.on("keydown_S", function() {
        if (playerHead.getData("direction") !== "UP") {
                playerHead.setDirection("DOWN");
        }
}, this);

this.input.keyboard.on("keydown_A", function() {
        if (playerHead.getData("direction") !== "RIGHT") {
                playerHead.setDirection("LEFT");
        }
}, this);

this.input.keyboard.on("keydown_D", function() {
        if (playerHead.getData("direction") !== "LEFT") {
                playerHead.setDirection("RIGHT");
        }
}, this);
```

Finally, we will add the collision checks between worm segments and the walls, as well as collisions with food. Phaser provides us a way to check if two game objects overlap. Add the following code to add the overlap logic between player segments and the walls:

```
this.physics.add.overlap(this.playerSegments, this.walls, function(segment, wall) {
	this.setGameOver();
}, null, this);
```

We can finish SceneMain by adding the last overlap check for testing for an overlap between worm segments and food:

```
this.physics.add.overlap(this.playerSegments, this.food, function(segment, food) {
	if (food) {
		this.sfx.eat.play();
		this.addSegment();
		this.addFood();
		food.destroy();
	}
}, null, this);
```

Now we can move on to our final file, SceneMainMenu.js! In this file, we can declare a new scene called SceneMainMenu. Add the following to do so:

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

Let’s now add the following code to the preload method in order to load the content for our main menu:

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

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

We can conclude this course by adding the create method. To begin, we can add our familiar sfx object for future referencing of the button sound effect. Add the following to create our create method and define our sfx object:

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

Next, we can add the title for our game:

```
this.textTitle = this.add.text(
        this.game.config.width * 0.5,
        64,
        "WORMY",
        {
                fontFamily: "monospace",
                fontSize: 72,
                align: "center"
        }
);
this.textTitle.setOrigin(0.5);
```

Now we can add our play button. To create the play button, we will add a sprite, and set it as interactive. We will be able to apply pointer events (mouse and touch) directly to the play button sprite. Add the following to add our play button and set it as being interactive:

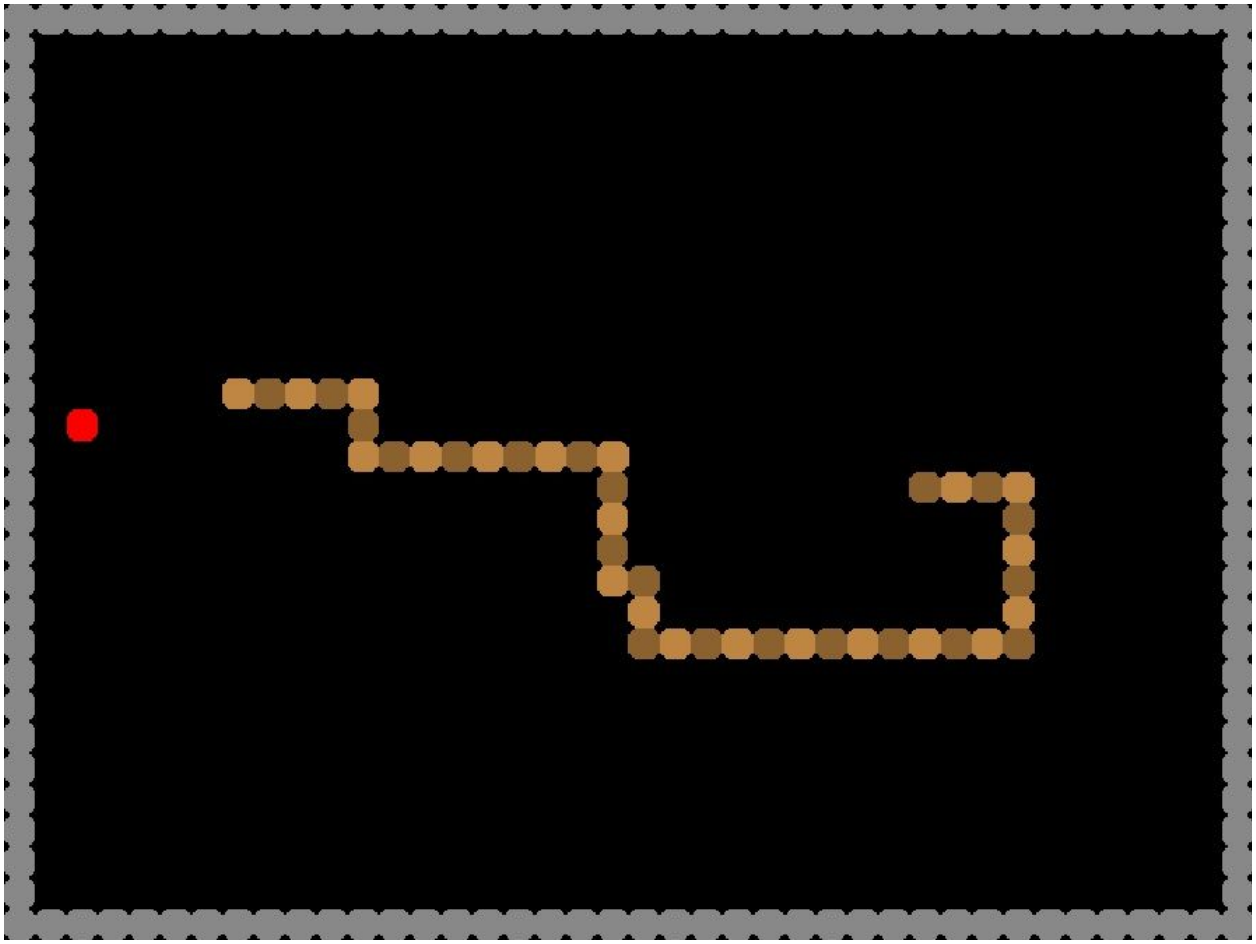
```
this.btnPlay = this.add.sprite(
        this.game.config.width * 0.5,
        this.game.config.height * 0.5,
        "sprBtnPlay"
);
this.btnPlay.setInteractive();
```

Then we can add our pointer events:

```
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");
});
```

The last thing we need to is head over to our game.js file, and insert SceneMainMenu above SceneMain in the scenes array. With that, we now should have a complete game! When you navigate to it in the browser, you should see a functional main menu, followed by our worm friend that can move around the screen doing what worms do.



I hope you have found this course valuable. The completed code for this course can be found on GitHub. If you found this course helpful, and would like to receive updates on my future courses and tutorials, be sure to fill out the form. If you have any questions at all, please don't hesitate to contact me. My email is jared.york@yorkcs.com and my Twitter handle is @jaredyork_. I'd also love to see anything awesome you're able to create from this course too!