Build Arcade Games with Phaser 3: Table Tennis

Version 1.0
Written by Jared York

INTRODUCTION

This is one of several courses in the, "Build Arcade Games with Phaser 3" series. All of these courses should be roughly the same difficulty, as in, not too terribly difficult. These courses will be based on the use of the Phaser framework, compatible with JavaScript and TypeScript. Phaser can be found on the official website. It is recommended that you have a grasp on the foundations of JavaScript. If not, no worries, I wrote a free course, "JavaScript Beginner Blocks", which I would highly recommend going through.

We will be trying to recreate a game similar to *Pong*! Before moving on to grander game ideas, I highly recommend going through this course to grasp how basic game logic works. In this course, we will also be exploring many features that Phaser provides.

SETTING UP

Though Phaser games are ran in your browser, unfortunately you can't simply run a local HTML file directly from your file system. When requesting files over http;, the security of the server allows you to access only the file's you're allowed to. When loading a file from the local file system (the file:// protocol), your browser highly restricts it for obvious reasons. It would be a huge problem to allow website code to read anything on your raw file system. Because of this, we will need to host our game on a local web server.

For more information on setting up a web server, we recommend checking out Phaser's official guide, "Getting Started with Phaser 3". The official guide will cover a variety of web servers compatible with your system and there are links to each one. The guide also provides detailed summaries on each of the various web servers mentioned.

CREATE THE FILES AND FOLDERS NEEDED

First, figure out 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 located the directory, create a new folder inside it and name it anything you wish.

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

We now need to create two new folders. I named the first one "content", this folder will be dedicated to containing any content for our game. The second one, "js", which will contain our Phaser script and our other game scripts. Feel free to name these two folders anything you like. One of the folders just needs to be dedicated to the content of our game, and the other for the JavaScript files. Now that we have a folder for JavaScript, and one for JavaScript, crate three new files inside the newly created folder for JavaScript called: SceneMainMenu.js, SceneMain.js, Entities.js and game.js. I will explain what these files do shortly, but we need to populate the content folder for the content for our game.

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

```
(game folder)
|_ index.html
|_content/
|_ js/
   |_ game.js
   |_ SceneMain.js
   |_ SceneMainMenu.js
```

The next step is to add content to our game. I have prepared some assets for this course that are freely available for download here. Otherwise, you may create your own assets if you wish. Below is a list of the content we will need:

Content needed:

- Sprites (images)
  - sprBall.png (the ball)
  - sprPaddle.png (the paddle)
  - sprHalfLine.png (the half line)
  - sprBtnPlay.png (the play button)
  - sprBtnPlayHover.png (the play button when mouse moves over)
  - sndBtn.wav (sound played when mouse hovers over button)

- sndHit.wav (sound played when ball hits paddle)

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

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

This concludes the first part of this course, "Build Arcade Games with Phaser: Pong". We can now move on to the meat and potatoes of building our game!

ADDING THE STARTING CODE

In the last part, we finished creating the files and directories we need for our game. The next thing we need to do is start adding to our index.html. In order to start coding our HTML and JavaScript files, you will need a text editor. Any text editor will work for this such as Notepad, Visual Studio Code, Atom, etc. I personally use Visual Studio Code for web development, it's snappy and just works.

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

```
<!DOCTYPE html>
<html>
        <head>
                <meta charset=”utf-8”>
                <meta lang=”en-us”>
                <title>Pong</title>
                <script src=”js/phaser.min.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>
```

In the above code, we are declaring the file to be an HTML document (hence the .html file extension). We are also declaring the scripts we will use. Do note the order we declare the scripts that contain “Scene” in them in comparison to the game.js script. The order is important because JavaScript files are interpreted from top to bottom.

First, we need to initialize a new Phaser game inside game.js. Take a second and open game.js if you haven’t already and create a JavaScript object like the following:

```
var config = {

};
```

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

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

Inside our configuration object, we are effectively telling Phaser we want our game to be rendered via WebGL instead of using regular Canvas rendering technology. The next part of the config includes width and height which will define the size of our game on the page. The property backgroundColor defines the background color that will be shown behind our game. The next property, physics defines the default physics engine we will be using, which is arcade. Arcade physics work well when we just want some basic collision detection, without many extra features. Inside the physics property, we also set the default gravity of the physics world. The next property, scene, is set to an array which we will add to in a little bit. Finally we want Phaser to render pixels crisp just like the great arcade games were back in the day.

The scene array we have defined inside our config object is needed in order to declare the order of scenes in our game. A scene is pretty much the equivalent of any sort of "screen" you see in a video game. For instance, the main menu, the play state, and the game over screen, are what I would call separate "screens". Well, a scene is pretty much the same as a screen. We will be defining our scenes as a JavaScript class. A class is still an object, the syntax is just difference and will help us organize our code a bit better. Classes in JavaScript are what is called in the development community, syntactic sugar. We will now add our scene classes inside the array that belongs to the scene property we just wrote.

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

We have completed our configuration object! We will not need to touch it for the duration of this course. The final line we need (and can be argued as most important) after the last curly brace of our config object is:

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

Our game.js file is complete! We now need to define a class for each of our scene scripts. Let's open SceneMainMenu.js first and add the following:

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

```
	preload() {

	}

	create() {
		this.scene.start("SceneMain");
	}
}
```

In the above code, we are declaring a class with class SceneMainMenu. On the same line, we also added extends Phaser.Scene {. When we extend Phaser.Scene, it really means building on top of Phaser's Scene class. Phaser.Scene refers to a class named Scene which is a property of the object, Phaser. Inside the class we just declared, there are two functions, constructor and create. The constructor function is called immediately after creating an instance of the class. Within the constructor we added:

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

Since we're extending Phaser.Scene, the above code essentially is the equivalent of:

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

However, since we're building on Phaser.Scene, we can use the super keyword to provide the parameters that you would have to if you instantiated Phaser.Scene normally. We can now go on to write similar code for SceneMain.js:

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

	preload() {

	}

	createHalfLine() {

	}
	reset() {

	}


	setGameOver() {

	}

	create() {

	}

	update() {

	}
}
```

In the only difference between the code for SceneMain and SceneMainMenu (besides the name change) is removing the scene.start method from the create function.

ENTITIES

The next step is to add the entity classes to our Entities.js file.
For a table tennis game, we will need four classes: Entity, Ball, PaddlePlayer, and PaddleCPU. The Entity class will be the base class we will inherit in the Ball, PaddlePlayer, and PaddleCPU classes. Define the four classes as so:

```
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);
	}
}

class Ball extends Entity {
	constructor(scene, x, y) {
        }

        update() {

        }
}



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

        }

        moveUp() {

        }

        moveDown() {

        }

        update() {

        }
}

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

        }

        update() {

        }
}
```

The Entity class will "build on top of" Phaser.GameObjects.Sprite. In the constructor, we allow entity sprites to immediately be added to the screen when instantiated, as well as enable physics on it.

We then go on to define the Ball, PaddlePlayer, and PaddleCPU classes. There are also a few functions we define in PaddlePlayer and PaddleCPU.

The next step will be to fill in the create function for the Ball class. There are various physics properties we will want to define. Inside the constructor of the Ball class add:

```
super(scene, x, y, "sprBall");
this.setOrigin(0.5);
this.body.setBounce(1);

this.setData("speed", 500);

this.body.setVelocity(this.getData("speed"), 0);
```

In the above block, we are setting the origin of the sprite to the center. The convenient thing about the setOrigin method is you only have to add the first parameter if both parameters (x and y) are the same. It will apply the first parameter to each axis. After setting the origin, we set the physics body of our ball to have the ability to bounce. We then set the default speed of the ball as a piece of data. After defining the default speed, we then set the velocity of the ball when there's an instance created.

In the update function of the Ball, we want our ball to bounce off the top and bottom of the screen. In the update function of the Ball class add the following:

```
if (this.y < this.displayHeight * 0.5 ||
    this.y > this.scene.game.config.height - (this.displayHeight * 0.5)) {

        this.body.velocity.y = -this.body.velocity.y;
}
```

That finishes up our Ball class! Fantastic.

Now, let's move on to the PaddlePlayer class. It's time to fill in the constructor, add the following:

```
super(scene, x, y, “sprPaddle”);
this.setOrigin(0.5);
this.body.setImmovable(true);
this.setData(“speed”, 500);
```

We will mostly be adding the movement code to our PaddlePlayer class. Let’s start by adding some code to moveUp and moveDown.

```
moveUp() {
        this.body.setVelocityY(-this.getData(“speed”));
}

moveDown() {
        this.body.setVelocityY(this.getData(“speed”));
}
```

Once we add our keyboard handlers, we will be able to move up and down, but the paddle won’t be able to stop. To allow the PaddlePlayer to stop, add the following to the update function of the PaddlePlayer class:

```
this.body.setVelocity(0, 0);

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

We are now finished with our PaddlePlayer class! Next, we can move to our PaddleCPU class. In this class we will be adding some basic AI (artificial intelligence) so we as the player, can verse a somewhat competent “opponent”. In the constructor of the PaddleCPU class, we need to add the super keyword as usual, and set a couple physics properties.

```
super(scene, x, y, “sprPaddle”);
this.setOrigin(0.5);
this.body.setImmovable(true);
```

Then we can add the same code from the update function of the PaddlePlayer class to the update function of the PaddleCPU class:

```
this.body.setVelocity(0, 0);

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

With that, we are finished with all of the entity classes! Let’s go back to SceneMain.js.

SCENEMAIN

In the preload function of SceneMain, let’s add the loading code for our assets:

```
this.load.image(“sprBall”, “content/sprBall.png”);
this.load.image(“sprPaddle”, “content/sprPaddle.png”);
this.load.image(“sprHalfLine”, “content/sprHalfLine.png”);

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

Moving down, we need to add a bit of code in the createHalfLine function. The crateHalfLine should create a dashed line down the middle of the screen. Let’s add the the following code to createHalfLine:

```
for (var i = 0; i < this.game.config.height / 16; i++) {
        var line = this.add.sprite(this.game.config.width * 0.5, i * 24 + 8, “sprHalfLine”);
        this.halfLines.add(line);
}
```

Instead of moving to the next function down, let’s instead add some code to the create function of SceneMain. In the create function, we want to create the ball, the player paddle, and the CPU paddle. After that, we will need a way to store the half line sprites together, create some labels, and add a few colliders. I will break this sequence in to more manageable chunks as we jump in. To begin, we need to create an object that will contain our sounds. We will call this object sfx, and it will help organize our sounds. Add the following code to the create function:

```
this.sfx = {
        hit: this.sound.add(“sndHit”)
};
```

Next we will want to define a variable that determines whether the game is over or not. In the create function, add:

```
this.isGameOver = false;
```

We can also define two variables to hold the score of the player and the CPU:

```
this.scorePlayer = 0;
this.scoreCpu = 0;
```

After that, we will want to create a ball, the player paddle, and the CPU paddle. Before defining the ball, I want to point out that we will first need to create a group called, balls.

```
this.balls = this.add.group();
```

The only reason why we need to create a group to hold one ball, is to maintain our collision checks between the ball and the player paddle and the CPU paddle. We will be destroying the ball once it moves off screen, which usually if we're checking collisions against a single instance, would make the collision check ineffective once the ball is destroyed. Now we can create an instance of Ball, and add it to the group.

```
var ball = new Ball(
	this,
	this.game.config.width * 0.5,
	this.game.config.height * 0.5
);
this.balls.add(ball);
```

Next we will add the player paddle and the CPU paddle:

```
this.player = new PaddlePlayer(
	this,
	32,
	this.game.config.height * 0.5
);

this.cpu = new PaddleCPU(
	this,
	this.game.config.width - 32,
	this.game.config.height * 0.5
);
```

Then, we can define a group for our half line sprites.

```
this.halfLines = this.add.group();
```

We can also call the function, createHalfLine, that we added to previously:

```
this.createHalfLine();
```

Next we will have to create three text objects. One text object will be for the player's score, another will be for the CPU's score, and the third will be for displaying who won.

```
this.textScorePlayer = this.add.text(
	this.game.config.width * 0.25,
	64,
	this.scorePlayer,
	{
		fontFamily: “monospace”,
		fontSize: 64
	}
);
this.textScorePlayer.setOrigin(0.5);

this.textScoreCpu = this.add.text(
	this.game.config.width * 0.75,
	64,
	this.scoreCpu,
	{
		fontFmaily: “monospace”,
		fontSize: 64
	}
);
this.textScoreCpu.setOrigin(0.5);

this.textWin = this.add.text(
	this.game.config.width * 0.5,
	64,
	“”,
    {
        fontFamily: “monospace”,
        fontSize: 64
    }
);
this.textWin.setOrigin(0.5);
this.textWin.setVisible(false);
```

Finally, we will need to add our collision checks, in Phaser known as a collider. Colliders allow you to provide it two gameObjects (usually sprites or groups), and then define a callback which is interpreted when a collision occurs. You can also access the two instances that have collided within the callback. To begin, let’s create a collider that checks between the balls group and the player. We can do so by adding the following:

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

}, null, this);
```

By default the ball would bounce off the paddle. However we want to add a bit of logic that will increase or decrease the angle of the ball the farther it is from the center of the paddle when it collides. Let’s add the code inside the callback:

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

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

ball.body.velocity.y = dist * 30;

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

Now we can add another collider that will check for collisions between the ball and the CPU paddle:

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

}, null, this);
```

Inside the callback, we can add the same code as we did for the last one:

```
var dist = Phaser.Math.Distance.Between(0, ball.y, 0, cpu.y);

if (ball.y < cpu.y) {
	dist = -dist;
}

ball.body.velocity.y = dist * 30;

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

Brilliant. We are now finished with the create function of SceneMain! The next step will be to add the update function. The first portion of the update function will depend on the ball existing. Because of the possibility that the ball may not existing, we should check that a ball exists first. Add the following to the update function of SceneMain:

```
if (this.balls.getChildren().length > 0) {

}
```

Inside this block, we want to call the update function of the ball, check if the ball is out of bounds so we know who to score a point to, and finally provide the basic AI for the CPU paddle. Add the following inside the code block above:

```
var ball = this.balls.getChildren()[0];

ball.update();

if (ball.x < 0) {
	this.scoreCpu++;
	this.textScoreCpu.setText(this.scoreCpu);

	this.reset();
}
else if (ball.x > this.game.config.width) {
	this.scorePlayer++;
	this.textScorePlayer.setText(this.scorePlayer);
```

```
        this.reset();
}

if (ball.body !== undefined) {
        var cpuVelY = ball.body.velocity.y;
        this.cpu.body.velocity.y = cpuVelY * Phaser.Math.Between(6, 14) * 0.1;
}
```

Great! The last two things we have to add to the update function is our call to the player's update function, as well allowing the player to move vertically with the mouse. Add the following to the update function, under of the previous block pertaining to the ball:

```
this.player.update();
this.player.y = this.input.activePointer.y;
```

The update function is now complete! The last two functions we have to add to is reset, and setGameOver. Let's start with the reset function, add the following inside it:

```
if (this.scorePlayer >10 || this.scoreCpu > 10) {
        this.setGameOver();
}
else {

}
```

The first part of the above code block checks if either player has won, if so, call the setGameOver function. We will get to that function in a bit, but the next thing we have to do is add the following code inside the else statement from above:

```
this.textScorePlayer.setVisible(true);
this.textScoreCpu.setVisible(true);
this.textWin.setVisible(false);

this.isGameOver = false;

this.balls.clear(true, true);
```

This bit of code ensures that while the game is being played, the text objects displaying the player's score and the CPU's score is visible, while the text showing which player won is invisible. The next line ensures isGameOver is false the next round. The final line of this code removes all the balls (of which there should be only one) from the group. Don't worry, we will create a new ball in the next block. The reason why we call the clear method of the balls group instead of the destroy group, is we don't want to actually destroy the group. We only want to destroy the children in the group while keeping the balls group intact.

This next block is where it starts to get a little complicated. We will be creating a timer event that is only triggered once. This probably doesn't sound too bad yet, but we will be creating another one-time timer inside the first timer. Inside the callback of the first timer, we will be resetting the positions of the paddle and we will create a new ball to replace the one we destroyed previously. Add the following code to create the first timer, and the callback code with the instructions I mentioned above:

```
this.time.addEvent({
        delay: 1000,
        callback: function() {

                // reset the paddle positions
                this.player.y = this.game.config.height * 0.5;
                this.cpu.y = this.game.config.height * 0.5;

                // remove all the balls (the only one we've added)
                this.balls.clear(true, true);

                // create a new ball
                var ball = new Ball(
                        this,
                        this.game.config.width * 0.5,
                        this.game.config.height * 0.5
                );
                this.balls.add(ball);

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

After the line where we add the new ball, we need to add the new timer. Add the following block:

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

                if (this.balls.getChildren().length > 0) {
                        var ball = this.balls.getChildren()[0];

                        ball.body.setVelocity(
                                ball.getData("speed"),
                                0
                        );
                }

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

With that, we are finished with the reset function! The last function we need to work on is setGameOver. Inside setGameOver, we should set the isGameOver variable to true, destroy the ball, set the win text objects to display the player who won, set the visibility of the text objects, and finally restart the scene. Let's add the following code in our setGameOver function:

```
this.isGameOver = true;

this.balls.clear(true, true);

if (this.scorePlayer > 10) {
        this.textWin.setText("YOU WON!");
}
else if (this.scoreCpu > 10) {
        this.textWin.setText("CPU WON!");
}

this.textScorePlayer.setVisible(false);
this.textScoreCpu.setVisible(false);
this.textWin.setVisible(true);
this.balls.clear(true, true);

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

Now, we are finished with SceneMain.js!

SCENEMAINMENU

The last thing we have to do is create the main menu. Navigate to the SceneMainMenu.js file. In the preload function, we need to add our two image files for the play button. Let’s add the loading code:

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

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

Now in the create function, let’s remove the scene.start method call and add the following:

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

this.title = this.add.text(
	this.game.config.width * 0.5,
	this.game.config.height * 0.15,
	“TABLE TENNIS”,
	{
		fontFamily: “monospace”,
		fontSize: 72,
                    align: “center”
          }
);
this.title.setOrigin(0.5);


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

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

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

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

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

CONCLUDING THOUGHTS

With that, we are finished with this course! If you run the game, you should be able to click on the play button and play a table tennis like game. There will be more courses in this Phaser 3 arcade series. You can learn more about Phaser 3 on the Phaser website.

The final code for this course can be found on GitHub.

As usual, I look forward to hearing comments, suggestions, and feedback! If anyone has any questions at all, please do not hesitate to reach out to me. You can find me on Twitter, and you can find more of my courses here.

If you found this course valuable and would like to know about new courses from us, please fill out the form.