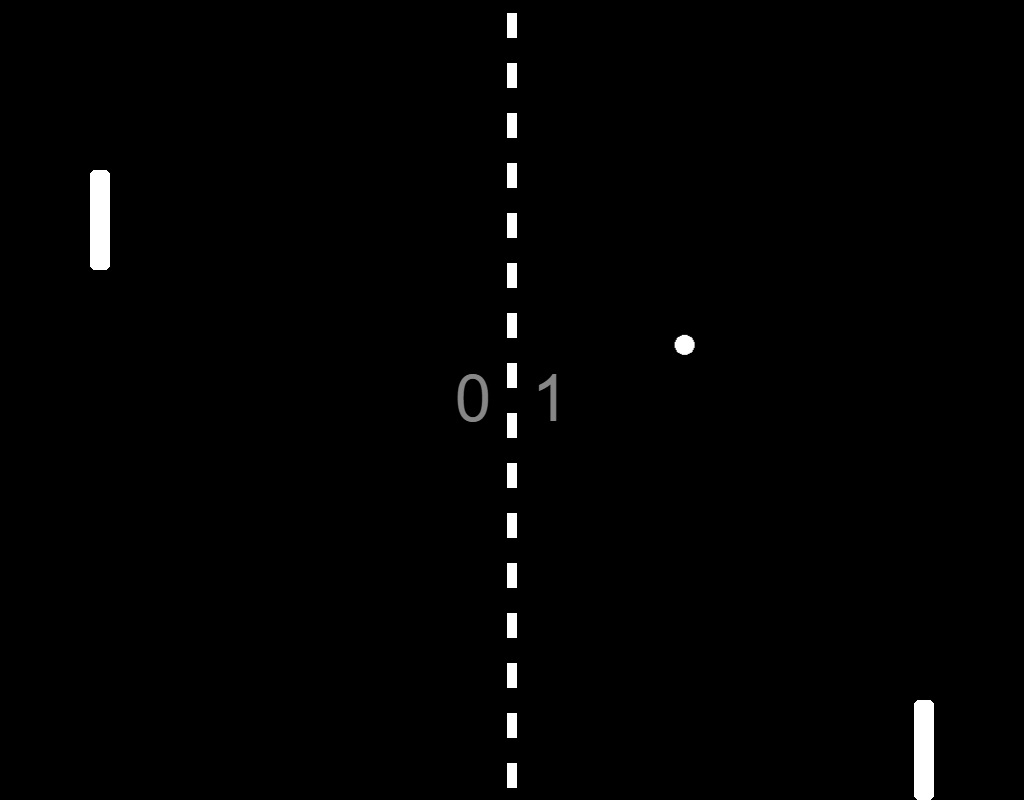
Phaser Pong Game Tutorial

# Intro

Pong is one of the earliest arcade video games. The game was originally manufactured by Atari, which released it in 1972. It’s an incredibly simple game, but even simple games can be deceiving in the effort needed to create them.

We will be remaking Pong while using Phaser.io as framework. Phaser will do the complicated stuff for us (e.g. collision detection, sprite movement), so we can focus on programming the actual gameplay.

Here you can see the original Pong (Left) and the version that we will make (Right).



# Prerequisites

* To download the code, you will need the git cli which you can download and install from the [git site](https://git-scm.com/)

# Steps

## Step 1: Downloading the project

Open a terminal in the folder you want to place the project in.

The project can be downloaded through git by using the command:

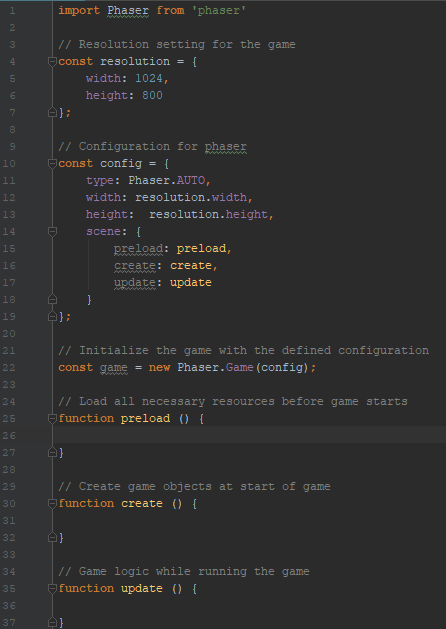
git clone https://github.com/PijnappelJ/phaser-pong.git

Switch to the tutorial code by using the following command:

git checkout tutorial

For now we are done with the terminal, but keep it open because we will need it later.

If you now open src/index.js, you should see the basic config object for Phaser and the 3 empty functions “preload”, “create” and “update”:



## Step 2: Running the game

Yes you read it correctly. We don’t have any game code yet, but we want to run this basic code to see if everything is set up right.

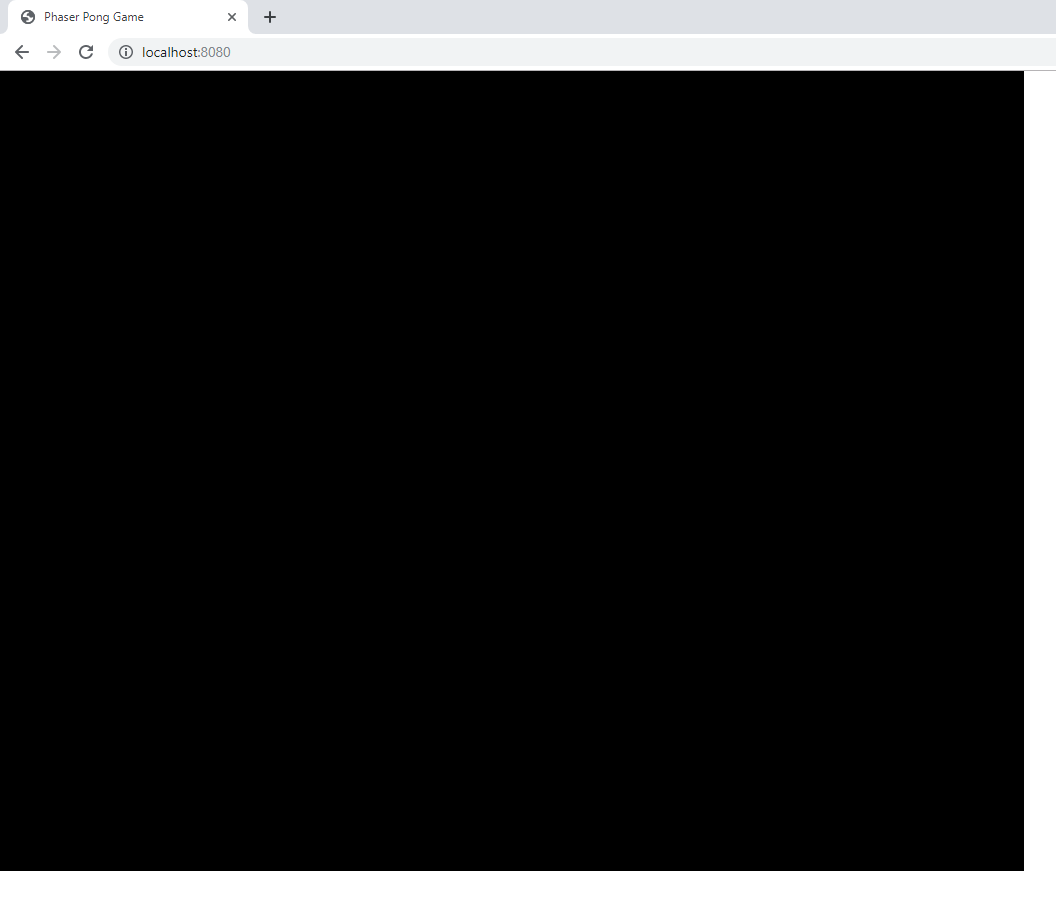
I have set up webpack to bundle our project and serve it locally.  
Use the following command in the terminal to start the webpack server locally:

npm run dev

You will see the server starting up and running the project at <http://localhost:8080/>.

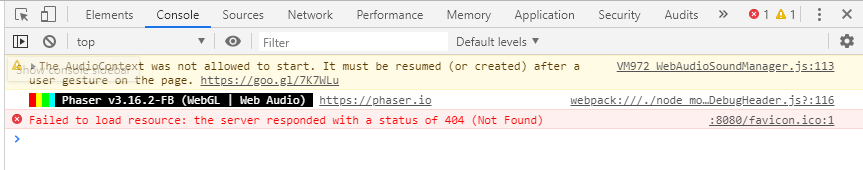
After the terminal displays the text: “Compiled successfully”, open your browser and enter the link: <http://localhost:8080/>.

You should now see a black game screen like so:



Open the developer tool in the browser (F12) and check the console.

It should display something like the following:



As you can see Phaser is running successfully!  
Don’t worry about the favicon error.

During development you can keep this screen open. When you have done some changes and want to see it in the browser, simply refresh the page and they will be loaded.

## Step 3: Creating the dotted middle line

Lets create our first game object.  
For pong we need a middle line to indicate the middle of the field.

Before adding the dotted line in the game screen, we first need to load the dotted line image into Phaser.  
To do this, add the following code to the “preload” function:

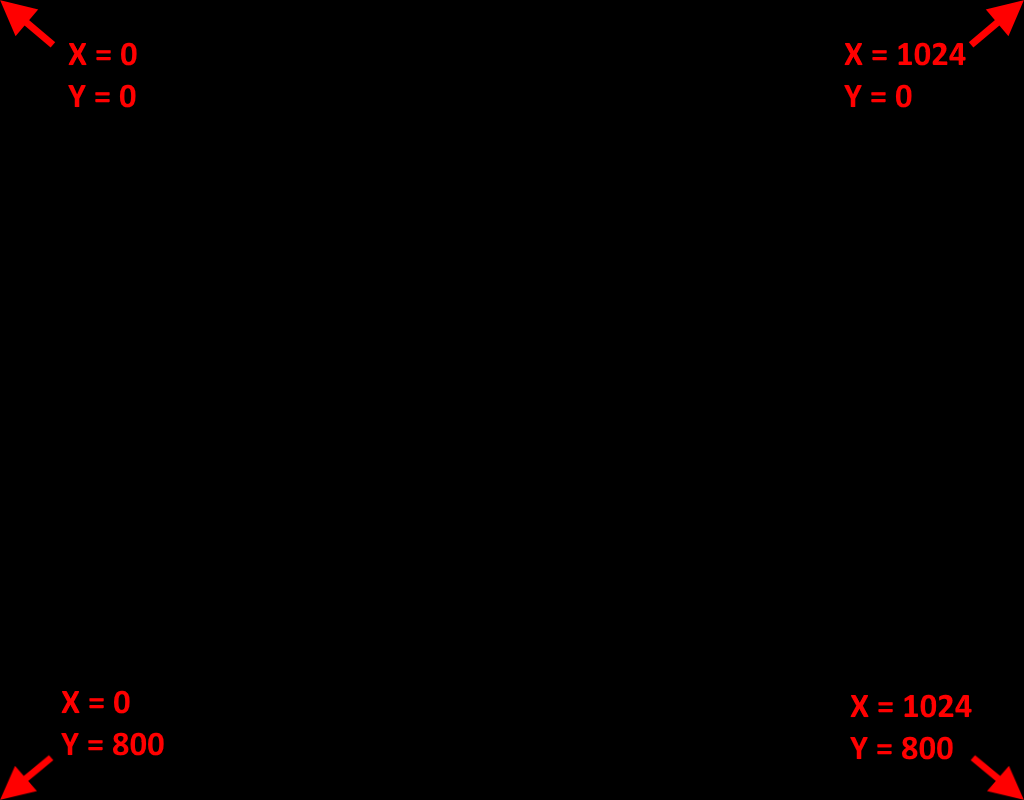
**this**.load.image('dotted-line', 'assets/dotted-line.png');

This code will register the dotted line image in Phaser under the name: “dotted-line”.

Before creating the line in the scene, it’s good to understand how positioning works in the game screen. Phaser interprets position data as distance from the upper left corner of the screen.

Horizontal distance is passed as X and vertical distance as Y.

In our specific setup the corners would have the following positions:



We want our dotted line exactly in the middle of the screen. So we will have to pass a position of half the total width (1024) and half the total height (800).

This width and height is stored in the resolution object, which we will use when initializing.

To create a dotted line in the scene, we refer to the image name as it is registered in Phaser.  
Add this code to the “create” function to add the dotted-line to the scene:

**this**.add.image(resolution.width \* 0.5, resolution.height \* 0.5, 'dotted-line');

Here is a breakdown about what this line is doing:

* “this.add.image” adds an image object to the scene
* The “this.add.image” function takes 3 parameters:
  + X-position (horizontal position)
  + Y-position (vertical position)
  + Image name
* The resolution object simply holds the width and height that we set in the config of Phaser
* Since we want the dotted-line in the middle of the page, the X-position is set to half of the game screen width and the Y-position is set to half the height.
* The last parameter refers to the dotted-line image we registered.

Save the code, and see how the game looks in the browser. It should now look something like this:



## Step 4: Creating the paddles

Our game is running and we have a nice dotted line, but we have nothing to play with.   
Let’s add some paddles that we can move around!

Before adding the paddles in the game screen, we first need to load the paddle image into Phaser.  
To do this, add the following code to the “preload” function:

**this**.load.image('paddle', 'assets/paddle.png');

This code will register the paddle image in Phaser under the name: “paddle”.

Of course we want our paddles to be moveable, otherwise the game would be very boring.  
If we want to dynamically move something in Phaser, it must be registered as a physics object.

In order to use physics objects, we first need to change the Phaser configuration to set up our physics preferences.

Add this property to the config object before the scene property:

physics: {  
 **default**: 'arcade'   
},

Adding this to the configuration will inform Phaser to use an arcade physics system. Phaser has multiple physics systems, where the arcade type is the most simple one. This physics system will take care of collisions between the ball and paddles or walls and will simply invert the current speed when hitting something.

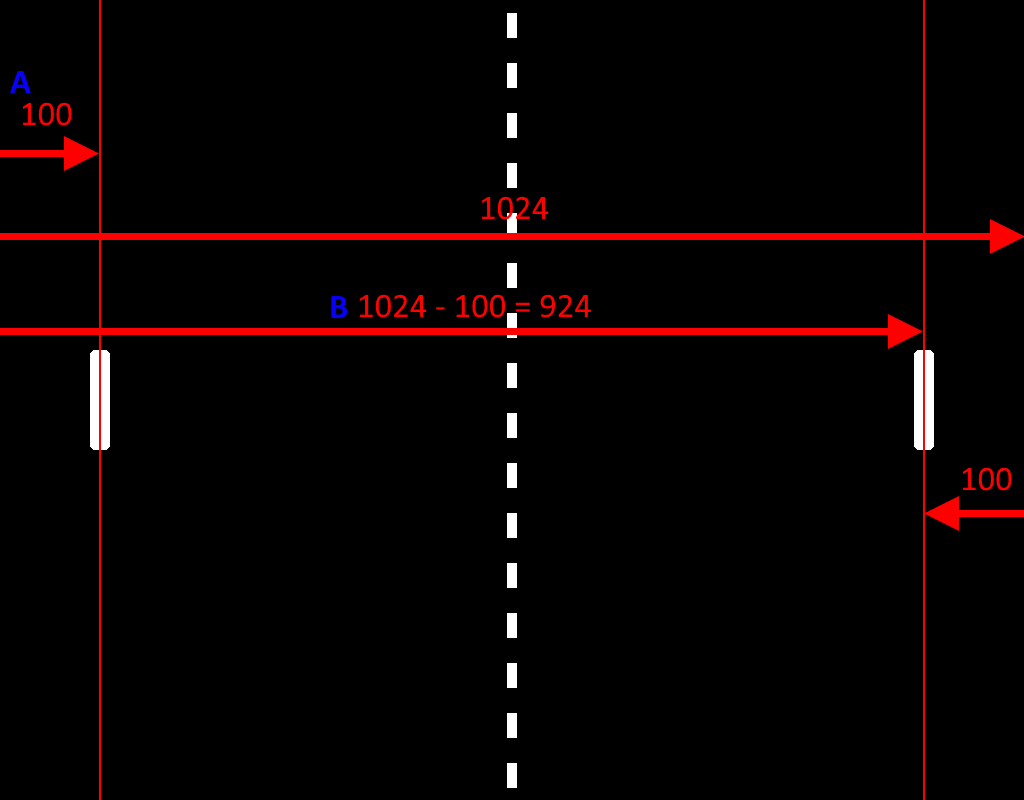
Before we create the actual paddles we must declare 3 variables like so:

// Game variables  
**let** paddles, player1Paddle, player2Paddle;

You can place this between the config and game object.

The “paddles” variable will function as a Phaser group object, holding both paddles. Why we need this will be explained later.  
“player1Paddle” and “player2Paddle” will hold the separate paddle objects.

Before we initialize the paddles, it’s good to think about their position in the field. When initializing we will have to provide the X- and Y-position. We want the paddles to be placed something like this, were A is the X for paddle 1 and B is the distance for paddle 2:



Because the distance between paddles and their respective screen sides will be the same (both 100), we can declare a constant that will hold this value. Add this code above or below the game variables you just added:

// Game constants  
**const** paddleDistFromSides = 100;

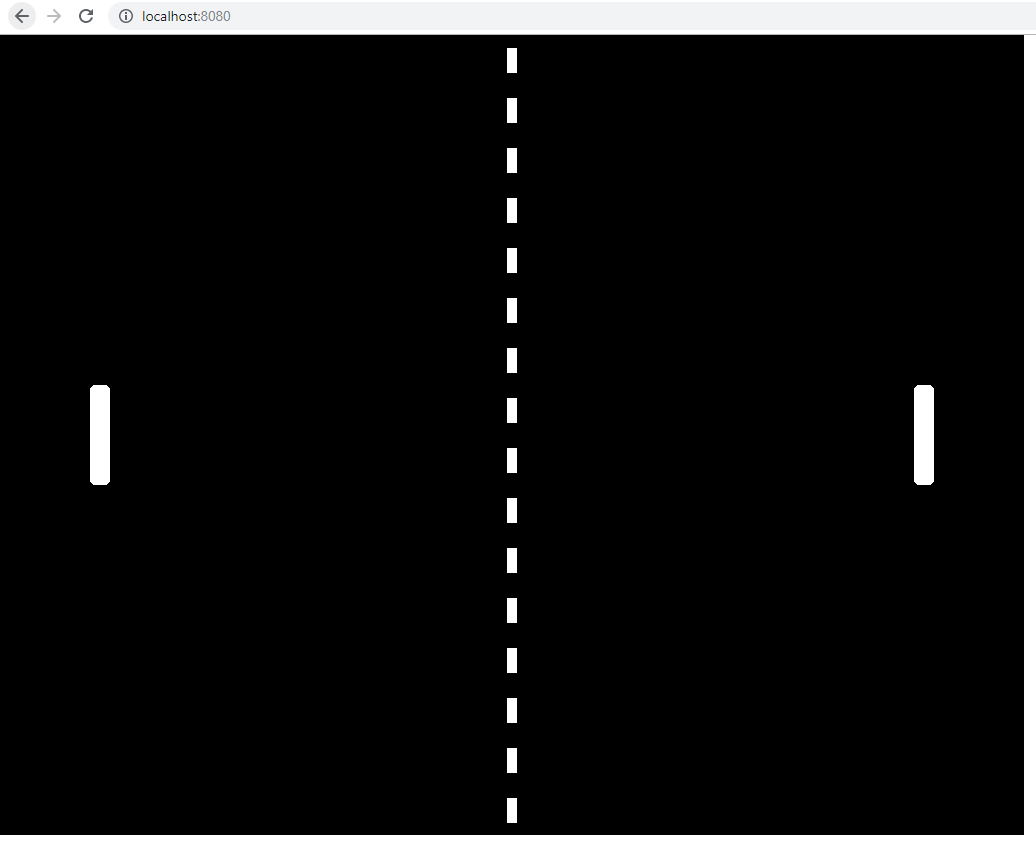
Now that the physics system is configured and the necessary variables have been declared, we can add the paddles to the scene in the “create” function:

paddles = **this**.physics.add.group();  
player1Paddle = paddles.create(paddleDistFromSides, resolution.height \* 0.5, 'paddle');  
player2Paddle = paddles.create(resolution.width - paddleDistFromSides, resolution.height \* 0.5, 'paddle');

Here’s the breakdown of this code:

* “paddles” is initialized as Phaser physics group. Having a physics group enables us to easily listen to the event of anything colliding with members of this group.
* “paddles.create” initializes “player1Paddle” and “player2Paddle” as member of the “paddles” group.
* Just like with the dotted line we need to provide the initialization function with 3 parameters: X-position, Y-position and an image name
* Paddle 1 will be on the left side. The distance between the paddle and the left side of the screen will then only be the distance that we declared before, “paddleDistanceFromSides” which is 100 (pixels).
* Paddle 2 will be 100 left of the right side, so the X-position will be the full screen width (1024) minus 100. The full screen width is stored in the resolution object, so the X-position will then be: resolution.width – paddleDistFromSides.
* As last parameter the name of the paddle image is passed so Phaser knows which image to display on the entered position.

When checking out the game in the browser, it should now look something like this:



## Step 5: Moving the paddles

A game is no game without user input. Let’s make the paddles move!

Let’s first declare a constant that will hold the speed we want the paddles to move with:

**const** paddleSpeed = 400;

This speed will be assigned to the paddles whenever the user presses the correct button.

To check whether buttons are pressed, we must declare variables that will hold references to the buttons we want to check. Add this line to your game variables:

**let** upButton, downButton, wButton, sButton;

To be able to listen to button presses, we must assign the right buttons to the button variables in the “preload” function:

upButton = **this**.input.keyboard.addKey('UP');  
downButton = **this**.input.keyboard.addKey('DOWN');  
wButton = **this**.input.keyboard.addKey('W');  
sButton = **this**.input.keyboard.addKey('S');

This code will assign the up-arrow button to the “upButton” variable, the down-arrow button to the “downButton” variable, etc.

Now that we have references to the correct keyboard buttons, we can write a function that will check the button statuses and move the paddles if the correct buttons are pressed. Add this function at the bottom of the file:

**function** updatePlayerControls () {  
 // Player 1 controls  
 **if** (wButton.isDown)  
 {  
 // Player 1 going up  
 player1Paddle.setVelocityY(-paddleSpeed);  
 }  
 **else if** (sButton.isDown) {  
 // Player 1 going down  
 player1Paddle.setVelocityY(paddleSpeed);  
 }  
 **else** {  
 // Player 1 stopping  
 player1Paddle.setVelocityY(0);  
 }  
  
 // Player 2 controls  
 **if** (upButton.isDown)  
 {  
 // Player 2 going up  
 player2Paddle.setVelocityY(-paddleSpeed);  
 }  
 **else if** (downButton.isDown) {  
 // Player 2 going down  
 player2Paddle.setVelocityY(paddleSpeed);  
 }  
 **else** {  
 // Player 2 stopping  
 player2Paddle.setVelocityY(0);  
 }  
}

As you can see pressing the up key will give the paddle a negative Y-velocity. Decreasing the Y-position means moving upwards in the screen. A positive Y velocity will move the paddle downwards and a velocity of 0 means the paddle will stay still.

To make this code work, it must run constantly to see if the user is pressing any button.

This is where we use the “update” function. This function is called by Phaser each cycle and is meant for the game logic. This is where things like button pressed or collisions are checked.

Since we want to check the player controls constantly, add this line to the “update” function:

updatePlayerControls();

If you now test your game in the browser, you should be able to make the paddles move up and down using the “W” and “S” button for player 1 and the up- and down-arrow buttons for player 2.

## Step 6: Implementing boundaries

You may have noticed a problem with the paddle movement. They can move completely out of the screen!

We need to limit the movement of the paddles to the ends of the screen.  
We could write a function that constantly checks the current position and prevents them from moving outside the screen, but Phaser has an easier way to limit physics objects.

After initializing the paddles, add this code to the “create” function:

player1Paddle.setCollideWorldBounds(**true**);  
player2Paddle.setCollideWorldBounds(**true**);

When setting these for both paddles, Phaser will take care of limiting the paddles to the edges of the screen.

If you check out the game in the browser, you should now be unable to move the paddles outside the game screen.

## Step 7: Adding a ball

Pong is no fun without a ball.

Let’s first register the ball image in Phaser. Add this line to the “preload” function:

**this**.load.image('ball', 'assets/ball.png');

Of course we want to keep track and perform actions on the ball, so let’s declare a variable for it. Declare the ball variable beneath the paddles declaration:

**let** ball;

Now we must create the ball in the scene and store the reference in the ball variable. Add this code to the “create” function:

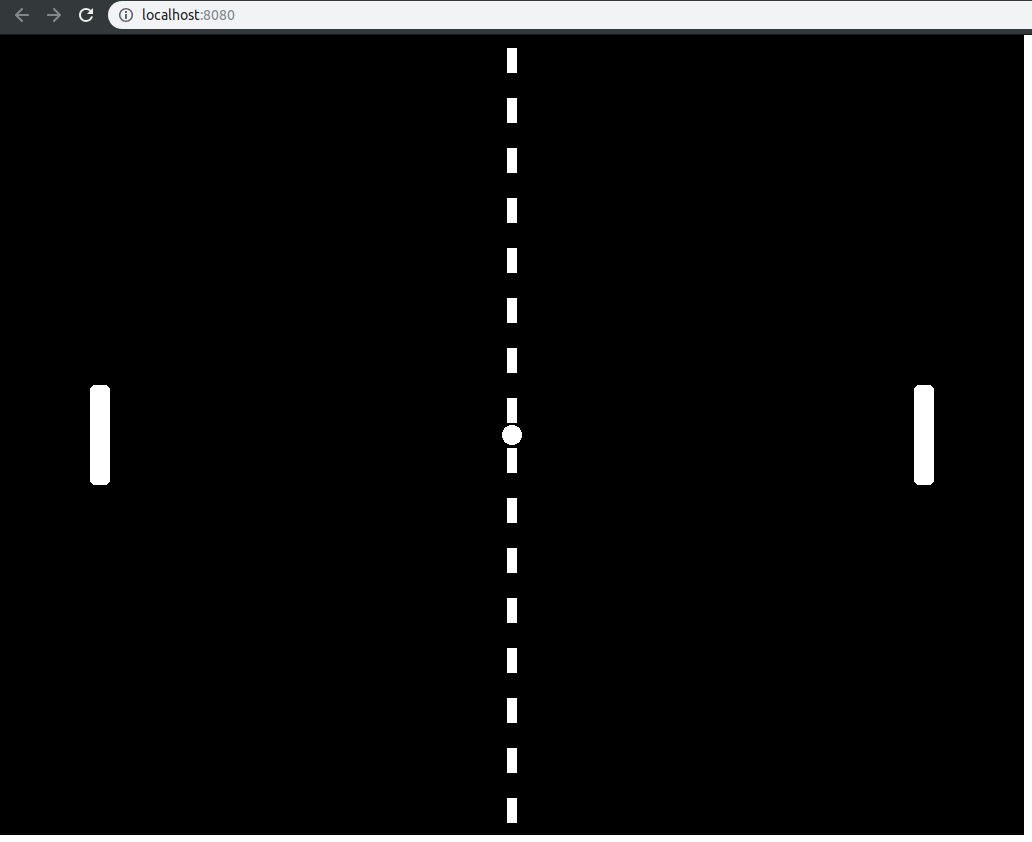
ball = **this**.physics.add.sprite(resolution.width \* 0.5, resolution.height \* 0.5, 'ball');  
ball.setCollideWorldBounds(**true**);

ball.setBounce(1**,** 1);

Here’s the breakdown of this code:

* “this.physics.add.sprite” initialize a new sprite that adheres to the physics rules. Have this as part of the physics group also enables us to listen to collision events (when it collides with the paddles for example).
* Again we provide the 3 needed parameters: X-position, Y-position and an image name
* Since we want the ball to always start in the middle of the screen, we initialize it at half the screen width and half the screen height.
* We want the ball to collide with the screen edges (to bounce off the top and bottom, and to be able to listen to the collision event when left or right is hit for resetting the ball and increasing the score).
* The ball must bounce off the screen edges, so we set the bounce factors for X and Y to 1 (meaning 100%).

If you look at the game in the browser, you should now see the added ball in the middle of the screen:



## Step 8: Launching the ball

Let’s launch the ball and see what happens.

Add this launchBall function to the end of your file:

**function** launchBall () {

**let** velocity = {x:-100, y:0}**;**

ball.setVelocity(velocity.x**,** velocity.y)**;**

}

For simplicity’s sake this function will simply launch the ball straight to the left.

Now we need to call this function after we created our ball in the “create” function:

launchBall()**;**

Now if you test your game, you’ll see it does not completely behave like we want it to.

The ball is launched, but it seems to just pass through the paddle, and it just bounces off the screen side edges instead of disappearing and starting from the middle again.  
We still need to code these functionalities. Let’s start with the handling of screen side edges.

## Step 9: Resetting the ball

Let’s create a new function that will take care of resetting the ball.

Add this function to the end of the file:

**function** resetBall () {

ball.setPosition(resolution.width \* 0.5**,** resolution.height \* 0.5)**;**

launchBall()**;**

}

As you can see this function set the position of the ball to the center of the screen and then launches the ball again. If we would not call the launchBall function, the ball would still move exactly the same, because the ball velocity has not been reset.  
Later we will randomize the ball velocity every time it is launched.

Of course we need to call this function when the ball hits the left or right screen edges.  
We can do this by checking every frame if the ball is colliding with the screen edges.  
Let’s first write a function that does this check:

**function** checkBallWallCollision () {

**if**(ball.body.onWall()) {

resetBall()**;**

}

}

Here we check the ball physics body if it is colliding with the “Walls” of the game. Phaser sees the left and right screen edges as the “Walls” of the game. The ball is colliding with the left or right screen edge when the “onWall()” function returns “true”.

Now call this function every frame by adding it to the update function:

checkBallWallCollision()**;**

Now checking the game again, we see the ball resetting as soon as it hits the left wall!

## Step 10: Colliding with the paddles

It’s all starting to look more and more like a game now!

The ball is still passing through the paddles. Let’s see if we can change that.

Let’s first create a function to deal with the ball-paddle collision:

**function** ballPaddleCollision (ballRef**,** paddleRef) {

**let** yDiff = ballRef.y - paddleRef.y**;**

ballRef.body.velocity.y += yDiff \* 5**;**

}

Here’s the breakdown of this code:

* The function will be called with a reference to the ball and a reference to the paddle that is hit.
* First the height difference of the ball and paddle position will be calculated.
* This difference will be used to set the vertical velocity of the ball when it is hit, multiplied by a factor 5 to enlarge the effect a little, for fun.
* This way the ball will bounce more upward the higher it hits the paddle, and more downward the lower it hits. The players will now be able to influence the direction the ball bouncing.
* Now the vertical velocity is added upon. If you’d like you could also completely overwrite the last velocity with a new one, negating the previous direction the ball is going.

You might be wondering, why do we only deal with the vertical (Y) velocity? The ball should bounce in the other horizontal (X) direction as well, right?   
That’s true, it should invert the X velocity when hitting the paddles. However, we don’t have to write this functionality ourselves. Phaser can do this for us!

The only this we need to do, is to tell Phaser we want to process a collision when the ball and a paddle collide.

To do this, add the following line to the “create” function:

**this**.physics.add.collider(ball**,** paddles**,** ballPaddleCollision)**;**

This line will tell Phaser it needs to respond to a collision between the ball object and the paddles group (which holds both the paddle objects). When a collision is found, Phaser will run its own normal collision response code and then call the function we passed as 3rd parameter.  
Phaser will call our function with 2 parameters, a reference to the ball and the paddle object that collided.

If we now check our game in the browser, we still see a problem.  
When a paddle is hit, the impact pushes the paddle away!  
This is the normal Phaser response between 2 physics objects hitting eachother. Phaser does not yet know we want our paddle to be immovable by other physics objects.

Luckily we can easily inform Phaser that our paddles should be immovable for other objects.   
Add this code directly after calling the “setCollideWorldBounds” function for both paddles in the “create” function:

player1Paddle.body.immovable = **true;**

player2Paddle.body.immovable = **true;**

Now our ball is bouncing like it should! We can even play with the paddles and see the ball changing direction! We have a pretty playable game already!

## Step 11: Keeping score

Our game is now getting much more fun, but it would even be more fun if we know who won!

Let’s create a scoring system that will display the current score of each player.

How about we display the scores next to the middle line. First we define a margin constant for the distance between the scores and the middle line:

**const** scoreMarginFromCenter = 20**;**

We’ll set it on 20 pixels, but feel free to adjust it later to whatever you find fitting.

We will also need to declare variables for holding the score text objects and variables to hold the actual current score for each player.   
Add this code to the game variables:

**let** scoreTextLeft**,** scoreTextRight**;**

**let** scoreLeft = 0**,** scoreRight = 0**;**

Now it’s time to create the score objects themselves. Add this to the “create” function:

scoreTextLeft = **this**.add.text(

resolution.width \* 0.5 - scoreMarginFromCenter**,**

resolution.height \* 0.5**,**

"0"**,**

{ font: "65px Arial", fill: "#878787" });

scoreTextLeft.setOrigin(1, 0.5)**;**

scoreTextRight = **this**.add.text(

resolution.width \* 0.5 + scoreMarginFromCenter**,**

resolution.height \* 0.5**,**

"0"**,**

{ font: "65px Arial", fill: "#878787" })**;**

scoreTextRight.setOrigin(0, 0.5)**;**

Here’s the breakdown of this code:

* “this.add.text” simply adds a text object to the scene, which will hold the score we want to add.
* We pass the following 4 parameters to the function:
  + X-position (which is the middle of the screen width, minus the margin we defined earlier for the left score and plus the margin for the right score)
  + Y-position (which we want to be the middle of the screen)
  + The initial text that the object displays. We will initialize with a score of “0”.
  + Additional text settings. We want to use the Arial font with a size of 65px and a greyish color. Feel free to customize this as you see fit.
* By default all initialized Phaser objects have their origin at the middle, which means that when setting their exact position, the middle of the object will be placed in that position.  
  We want the left text to never cross the middle line when changing, so we must change the origin to the right side of the text, which is 1 (meaning 100%) horizontally. Vertically we still want the text to have its origin in the middle, so the origin Y parameter will be 0.5 (meaning 50%).  
  For the right score we set the opposite to make sure the text never crosses the line left of it.

We should now be able to see our text in the game. However, it is not changing yet when a point is made.

Changing the score is not that hard, since we already have a function that is called when the left or right side of the screen is hit by the ball.  
Add this code to the “checkBallWallCollision” function, inside the if condition:

// If ball hits left wall

**if**(ball.body.left <= 0) {

scoreRight += 1**;**

scoreTextRight.setText(scoreRight)**;**

}

// If ball hits right wall

**else** {

scoreLeft += 1**;**

scoreTextLeft.setText(scoreLeft)**;**

}

Here’s the breakdown of this code:

* “if(ball.body.left <= 0)” checks whether the utmost left part of the ball sprite has a X-position lower than or equal to 0. This is true when the ball is hitting the left wall, but is false when hitting the right.
* If hitting the left wall, the score of the right player should be increased, which is done by increasing the “scoreRight” variable. The exact opposite is true for hitting the right wall, of course.
* After increasing the score value, it must still be updated in the score text. This is done by setting the text through the “setText” function. We pass the text we want to set, which is the newly updated score.

After you’ve implemented this change, you should now be able to see the scores update properly!

## Step 12: Randomizing the ball launch direction

We are almost done!

The ball is now always launched in the same direction, which is a bit too predictable. We want our players to be on their toes, so let’s make the ball launch direction random.

As you might have guessed we have to edit our existing “launchBall” function to do this.

Change the contents of this function to the following:

**let** randomVelocity = {x:0**,** y:0}**;**

randomVelocity = Phaser.Math.RandomXY(randomVelocity**,** 200)**;**

randomVelocity.x = (randomVelocity.x < 0) ? -200 : 200**;**

ball.setVelocity(randomVelocity.x**,** randomVelocity.y)**;**

Here we initialize an object with a x and y value and randomize these values between -200 and 200.  
For the Y velocity any of these values will be fine, but if X would be 1 for example, the ball would move much too slow towards the players.

This is why we set the X velocity to either -200 or 200 depending on the randomly generated value. If it is generated negative, set it to -200. If it is generated 0 or positive, set it to 200.

Now we have a fully functioning game of Pong!

There are still some improvements this game of Pong can use, if you still have some time left, feel free to take a shot at the extra assignments.

If you’d want to deploy this online, just run the command:

npm run build

Webpack should create a “docs” folder, which you can deploy on your server.

I hope you had fun!

If you’d like to keep an eye on my projects as hobby game developer, feel free to follow me on [itch.io](https://chocolatepinecone.itch.io/) or [GameJolt](https://gamejolt.com/@ChocolatePinecone). I’m currently making a moonlander type game where I’m actually building my own engine in the process. It’s called [Rocket Shipment](https://chocolatepinecone.itch.io/rocket-shipment).

# Extra assignments

In these assignments you will further improve your game to make it more fun. The answers for the challenges will not be provided and you will have to figure out a solution yourself.

Have fun!

## Assignment 1: Increasing the pressure

A good game of Pong is fun, but when both players are very good the game could use a little more pressure.

Try adding a new effect in the “ballPaddleCollision” function which will increase the horizontal (X) velocity. That way players need to react faster and faster to keep up with the speeding ball.

## Assignment 2: Pausing the game

What if you have to go to the bathroom? The game can’t be paused and your opponent will just keep scoring until you come back!

Add a pause functionality that is activated and de-activated when the user presses the space button.

## Assignment 3: AI opponent

What if you have no friends to play with? We could automate our opponent!

Program 1 of the paddles to automatically follow the ball’s Y position. Move it up if the ball’s Y position is higher than the paddle’s Y position, and move it down if it is lower.

Don’t forget to disable the buttons to control this paddle. Otherwise you could easily boycott your automated opponent.