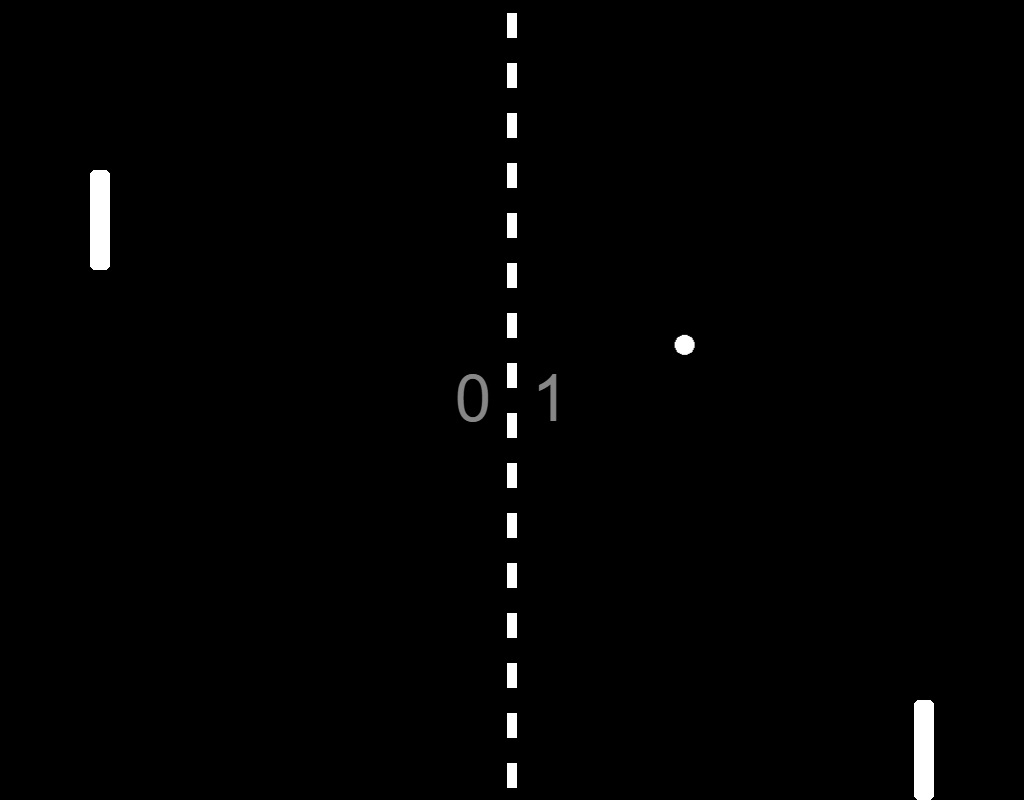
Phaser Pong Game Tutorial

# 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:

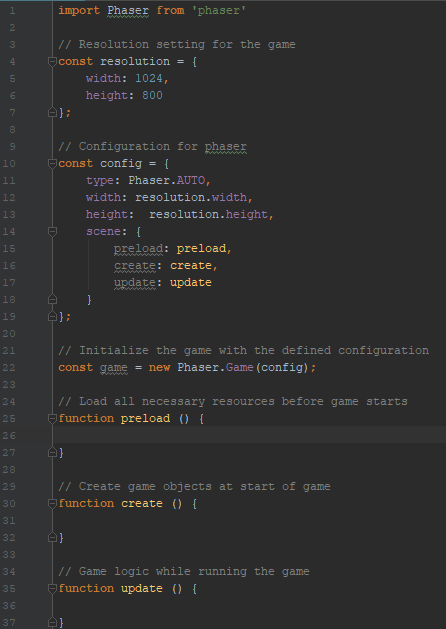
git clone https://github.com/PijnappelJ/phaser-pong.git

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.

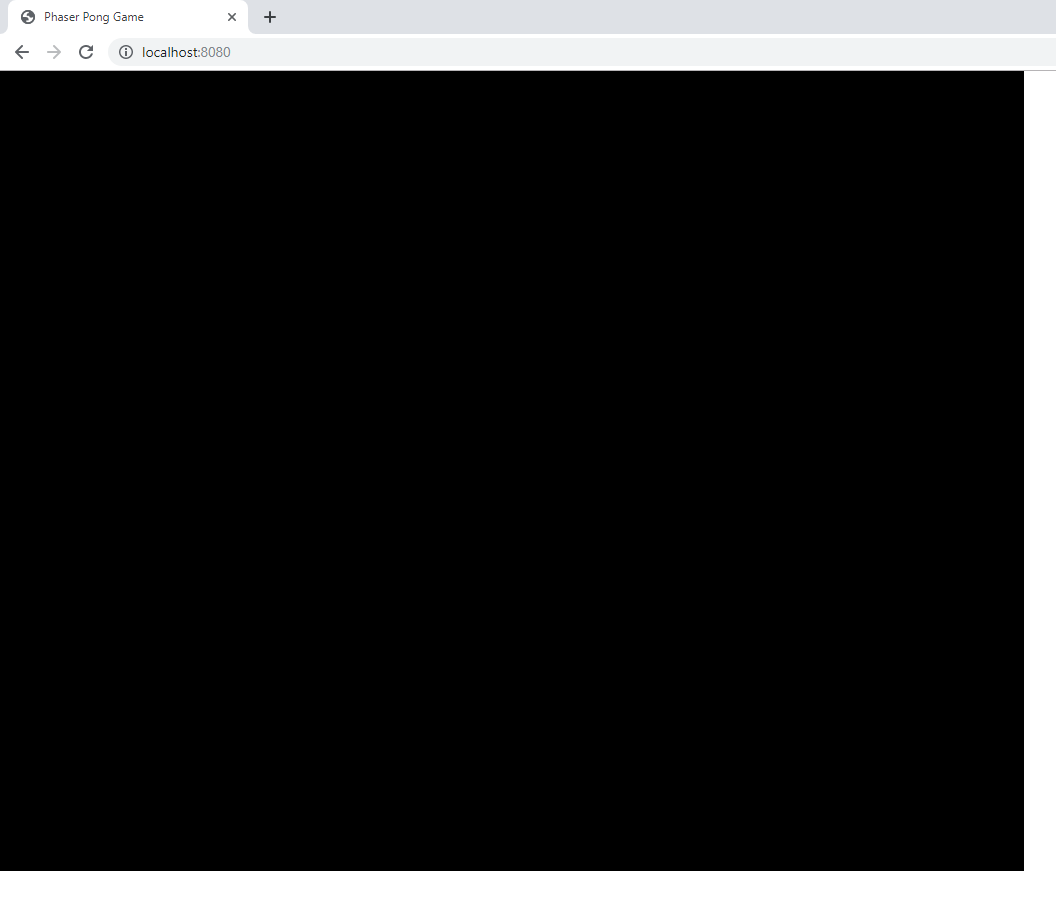
I have set up webpack to bundle our project and serve it locally.  
Use the following command in the terminal to start the webpack server locally:

npm run dev

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

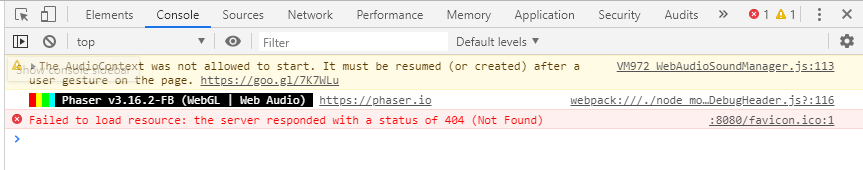
After the terminal displays the text: “Compiled successfully”, open your browser and enter the link: <http://localhost:8080/>.

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:

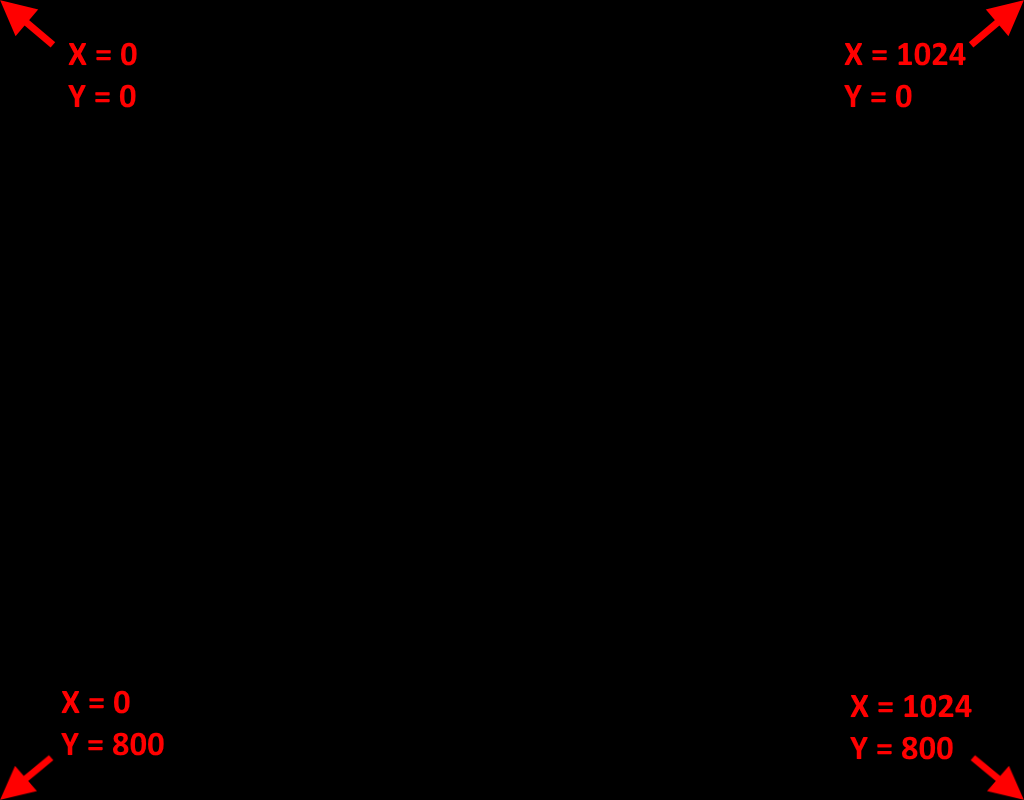
**this**.load.image('dotted-line', 'assets/dotted-line.png');

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.

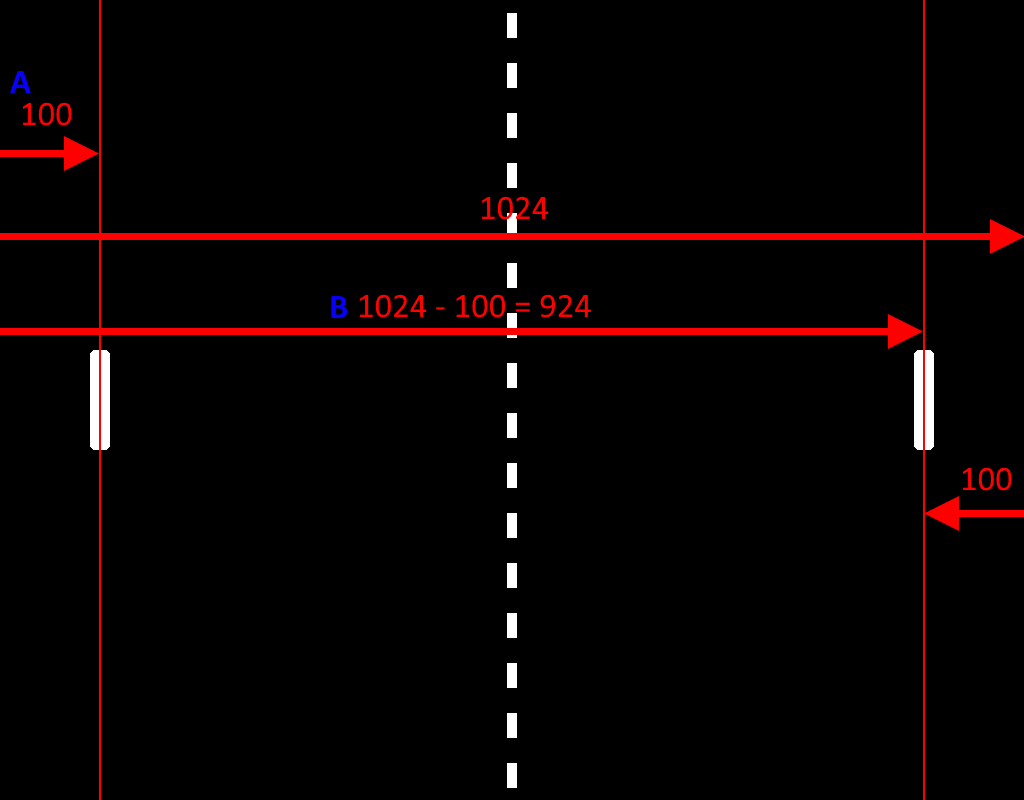
Before we create the actual paddles we must declare 3 variables like so:

// 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;

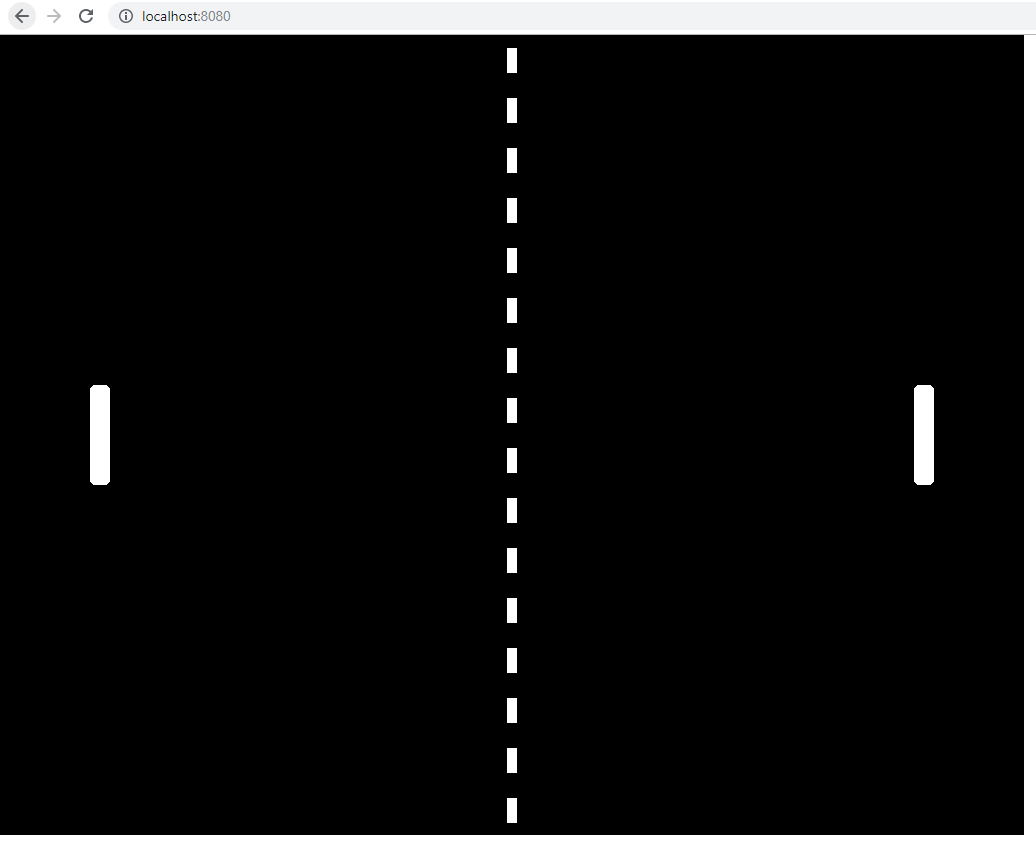
Now that the physics system is configured and the necessary variables have been declared, we can add the paddles to the scene in the “create” function:

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**);