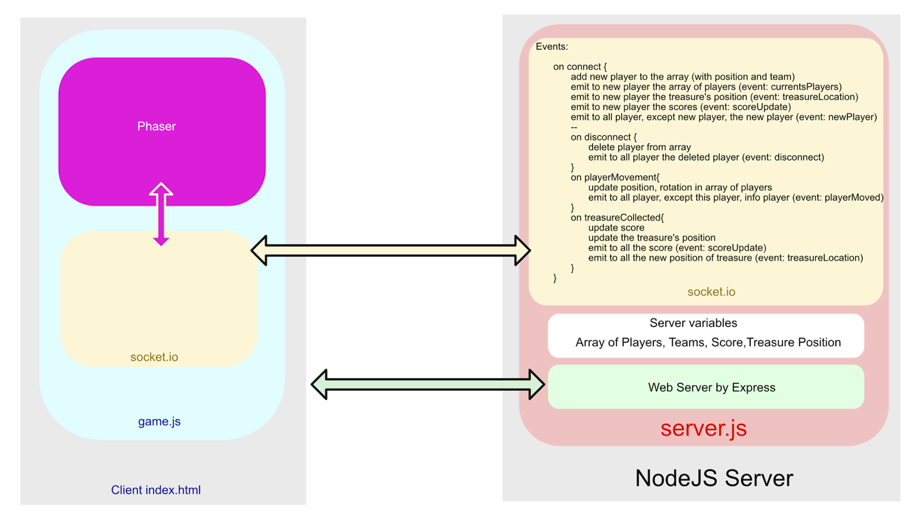
### Desarrollo de un juego 2D online

Tal como hemos indicado, vamos a desarrollar un juego 2D que empleará Phaser del lado del cliente para manejar los Sprites y la física del juego. Por otro lado, también usaremos socket.io tanto en el cliente como en el servidor para mantener a los clientes sincronizados mediante dichos sockets. El servidor mantendrá la lista de jugadores, así como sus características relevantes (equipo y física), la puntuación de cada equipo (rojo y azul) y la posición del tesoro.

De qué trata. Justo de esto. Cada vez que un jugador se conecte el servidor aleatoriamente le asignará uno de los dos equipos disponibles. Los jugadores tendrán que recoger un tesoro que a medida que sea recogido irá cambiando de posición aleatoriamente. Cada vez que sea recogido los puntos del equipo correspondiente aumentará y de esta forma tendremos lista la dinámica de nuestro juego cooperativo.

En primer lugar, vamos a centrarnos en desarrollar el backend, el lado del servidor. Para ello, tal como hicimos en el caso anterior, emplearemos un servidor nodeJS provisto de Express para proporcionar servicio web y de socket.io para gestionar las conexiones. La lógica de conexiones está descrita en el siguiente gráfico:



#### Código fuente

**Servidor de juegos online**

Comenzamos creando la carpeta de neutro proyecto denominado Phaser-Sockets y la establecemos como carpeta de proyecto en Visual Studio Code. Por motivos de claridad vamos a crear dos subcarpetas denominadas frontend y backend. Nosotros comenzaremos a trabajar en la carpeta backend (es importante que cuando ejecutemos comandos desde el terminal, nos cercioremos de que nos encontramos en la subcarpeta backed.

Nota: suponemos instalado nodeJs y npm.

Comenzamos generando nuestro archivo package.json mediante el comando:

npm init

Tras ejecutarlo y completar los datos, el contenido de nuestro archivo queda:

{

"name": "server",

"version": "1.0.0",

"description": "Online Server for Phaser by using Socket.io",

"main": "server.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1"

},

"author": "Developodo",

"license": "ISC"

}

Como hemos explicado, emplearemos express y socket.io, pero además lo haremos mediante Typescript, por lo que instalaremos los tres paquetes y los tipos de las librerías express y socket.io para acceder desde Typescript.

npm install -D typescript express @types/express socket.io @types/socket.io

En esta ocasión vamos a instalar una librería que nos permite ir recopilando nuestro código automáticamente a medida que vamos realizando cambios: ts-node-dev

npm install -D ts-node-dev

También instalaremos el plugin tslint que nos ayuda a mantener el código Typescript limpio y claro.

npm install typescript-tslint-plugin

Finalmente, instalamos el módulo de políticas CORS (Cross Origin Resource Sharing) para gestionar el acceso asíncrono al servidor.

npm install cors --save

Pasamos a configurar el archivo package.json para que el gestor npm “conozca” qué hacer cuando le pedimos que ejecute o compile el proyecto. Añadimos:

// package.json

{

...

"scripts": {

"dev": "ts-node-dev --respawn --transpileOnly ./src/server.ts",

"prod": "tsc && node ./dist/server.js”,

"start": "node ./dist/server.js"

},

...

}

Como observamos en caso de emplear el comando dev compilamos y ejecutamos el servidor con la propiedad de que cada vez que realicemos un cambio en server.ts se actualizará el servicio (server.js).

Para ejecutar el modo desarrollo:

npm run dev

Para ejecutar el modo producción:

npm run prod

Por último, para el despliegue en servidores nodeJs hemos incluido la entrada start. Para ejecutarlo:

npm start

En caso de emplear el parámetro prod, compilamos y ejecutamos el archivo server.js de la carpeta dist.

Por tanto, toca configurar el archivo tsconfig.json para que podamos organizar las carpetas de código fuente y compilado. Para ello creamos un archivo tsconfig.json en la carpeta /backend con el siguiente contenido:

// tsconfig.json

{

"compilerOptions": {

"module": "commonjs",

"esModuleInterop": false,

"target": "es6",

"noImplicitAny": true,

"moduleResolution": "node",

"sourceMap": true,

"outDir": "dist",

"baseUrl": ".",

"paths": {

"\*": [

"node\_modules/",

"src/types/\*"

]

}

},

"include": [

"src/\*\*/\*"

]

}

Toca comenzar a desarrollar el servidor. La estructura de archivos será la siguiente:

* /backend/src/server.ts es el archivo de entrada e instanciación del servidor completo.
* /backend/src/shipFighterServer.ts es la clase que empaqueta al servidor del juego.
* En la carpeta /model tendremos:
* /backend/src/model/Player.ts: es la interfaz que define los datos de un jugador.
* /backend/src/model/Score.ts: la interfaz de la puntuación del juego.
* /backend/src/model/Treasure.ts: la interfaz de los datos del objeto tesoro.

En la carpeta /commons tendremos:

/backend/src/commons/eventsName.ts: archivo que define en un enum los diferentes eventos que existen en la comunicación cliente-servidor.

Los códigos de la carpeta model son:

Player.ts

export interface Player{

playerId: any, // get from socket.id

team: string, //red or blue

rotation: number, //the rest of parameters are got from Phaser engine

x: number,

y: number

}

Score.ts

export interface Score{

blue:number, // blue teams’s points

red:number

}

Treasure.ts

export interface Treasure{

x:number, // position on screen

y:number

}

El código de la carpeta commons es:

eventsName.ts

export enum eventsName {

CONNECTION = 'connection', //when a player connects

DISCONNECT = 'disconnect', //when a player disconnects

CURRENTPLAYERS = 'currentPlayers', //when a new player connects: all info about players

NEWPLAYER = 'newPlayer', //when a new player connects: info about this player to the rest of them

TREASURELOCATION = 'treasureLocation', //to know where the treasure is

SCOREUPDATE = 'scoreUpdate', //to know what the scores are

PLAYERMOVEMENT = 'playerMovement', //when a user moves

TREASURECOLLECTED = 'treasureCollected' //when a player catches the treasure

}

El core del server.

server.ts

// src/server.ts

import { shipFighterServer } from './shipFighterServer';

let app = new shipFighterServer().app; //getter

export { app };

shipFighterServer.ts

import \* as express from 'express';

import \* as socketIo from 'socket.io';

import { eventsName } from './commons/eventsName';

import { Player } from './model/Player';

import { Score } from './model/Score';

import { Treasure } from './model/Treasure';

import { createServer, Server } from 'http';

var cors = require('cors');

interface IDictionary {

[key: string]: Player;

};

export class shipFighterServer {

// About the game

players:IDictionary; //let a associative array

treasure:Treasure;

score:Score;

static readonly horizontalSize = 800;

static readonly verticalSize = 600;

// About the server

public static readonly PORT: number = 8080;

private \_app: express.Application;

private server: Server;

private io: SocketIO.Server;

private port: string | number;

constructor () {

// setting web server

this.\_app = express();

this.port = process.env.PORT || shipFighterServer.PORT; // for deploying

this.\_app.use(cors());

this.\_app.options('\*', cors());

this.server = createServer(this.\_app);

// setting game variables

this.initGame();

// setting sockets over http server

this.initSocket();

// let's listen

this.listen();

}

private initSocket (): void {

this.io = socketIo(this.server);

}

private listen (): void {

this.server.listen(this.port, () => {

console.log('Running server on port %s', this.port);

});

//everthing starts here. When a player connects

this.io.on(eventsName.CONNECTION, (socket: any) => {

console.log('Connected client on port %s.', this.port);

//when a player connects

//1) add it to players array

this.players[socket.id]={

playerId:socket.id,

rotation:0,

x:this.getRandomH(),

y:this.getRandomV(),

team: Math.floor(Math.random()\*2)==0?"red":"blue" // not well balanced, but easy

}

//2) send all players to this new player

socket.emit(eventsName.CURRENTPLAYERS,this.players);

//3) send treasure's position to this new player

socket.emit(eventsName.TREASURELOCATION,this.treasure);

//4) and, of course, send the scores to this new player

socket.emit(eventsName.SCOREUPDATE,this.score);

//5) send this new player to the rest of players

socket.broadcast.emit(eventsName.NEWPLAYER,this.players[socket.id]);

// when this player moves

socket.on(eventsName.PLAYERMOVEMENT, (movementData:any) => {

this.players[socket.id].x=movementData.x;

this.players[socket.id].y=movementData.y;

this.players[socket.id].rotation=movementData.rotation;

// tell to other players that this player moved

socket.broadcast.emit(eventsName.PLAYERMOVEMENT,this.players[socket.id]);

});

// when this player catches the treasure

socket.on(eventsName.TREASURECOLLECTED, () => {

// update the score variable on server

this.players[socket.id].team === 'red'? (this.score.red+=10) : (this.score.blue+=10) ;

// move the treasure to a new position

this.treasure.x = this.getRandomH();

this.treasure.y = this.getRandomV();

// send new position to all players

this.io.emit(eventsName.TREASURELOCATION,this.treasure);

// send new score to all players

this.io.emit(eventsName.SCOREUPDATE,this.score);

});

// when this player disconnects

socket.on(eventsName.DISCONNECT, () => {

console.log('Client '+socket.id+'disconnected');

//1) remove it from players array

delete this.players[socket.id];

//2) sent to the rest of players this situation

this.io.emit(eventsName.DISCONNECT,socket.id);

//if any player left, reset score

if(Object.keys(this.players).length == 0){

this.score={

blue:0,

red:0

}

}

});

});

}

get app (): express.Application {

return this.\_app;

}

initGame(){

this.players={};

this.score={

blue:0,

red:0

}

this.treasure={

x: this.getRandomH(),

y: this.getRandomV()

}

}

getRandomH():number{

return Math.floor(Math.random()\*(shipFighterServer.horizontalSize-100)+50);

}

getRandomV():number{

return Math.floor(Math.random()\*(shipFighterServer.verticalSize-100)+50);

}

}

Podemos ver a estas alturas, la claridad y modularidad que proporciona Typescript comparando el desarrollo de este servidor con el del ejemplo del apartado anterior.

Como dijimos, podemos ejecutarlo a la vez que desarrollamos con el comando:

npm run dev

O compilarlo para desplegarlo y ejecutarlo con:

npm run prod

Toca el turno del frontend.

**Cliente del juego online**

Todo lo que vamos a desarrollar debe estar en la carpeta frontend de nuestro proyecto.

Vamos a preparar nuestro entorno para emplear Phaser3 con Typescript. Comenzamos creando nuestro proyecto con el comando:

npm init

A partir de aquí irá realizando una serie de preguntas para configurar el proyecto. Posteriormente se crea el archivo package.json que tendrá el siguiente contenido en nuestro ejemplo:

{

"name": "shipfighter",

"version": "1.0.0",

"description": "A multiplayer game made in Phaser + Typescript",

"main": “index.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1"

},

"author": "Developodo",

"license": "ISC"

}

Instalamos webpack para que al compilar empaquete todo el código de la aplicación. Además de Typescript, si no lo tuviéramos y los demás paquetes necesarios, como el cargador ts-loadeer, el servidor de pruebas y el propio phaser:

npm install -D typescript webpack webpack-cli ts-loader phaser live-server

Para que webpack funcione adecuadamente debemos crear en la raíz el archivo webpack.config.js con el siguiente contenido:

const path = require('path');

module.exports = {

entry: "./src/app.ts",

module: {

rules: [{ test: /\.tsx?$/, use: "ts-loader", exclude: /node\_modules/ }]

},

resolve: { extensions: [".ts", ".tsx", ".js"] },

output: { filename: "app.js", path: path.resolve(\_\_dirname, "dist") },

mode: "development"

};

Observamos, como todo nuestro código estará en un archivo /src/game.ts que será transpilado al archivo /dist/game.js.

Para typescript configuramos un archivo básico tsconfig.json en la raíz del proyecto:

{

"compilerOptions": {

"target": "es5"

},

"include": [

"src/\*.ts"

],

"exclude": [

"node\_modules",

"\*\*/\*.spec.ts"

]

}

Para interaccionar Typescript con la librería JS de Phaser, requerimos el archivo de tipados que podemos encontrar en la siguiente dirección <https://github.com/photonstorm/phaser/tree/master/types> (en el momento de escribir este documento no se encuentra disponible mediante npm). Descargaremos el archivo phaser.d.ts y lo copiaremos en la carpeta src que debemos crear.

Por último, debemos actualizar package.json para indicar a npm cómo desplegar la aplicación. Modificamos la línea:

"scripts": { "build": "webpack", "start": "webpack --watch & live-server —port=8085"}

A partir de aquí:

Si ejecutamos npm build crearemos game.js

Si ejecutamos npm start, por cada cambio que hagamos en el código live-server actualizará el contenido y recargará la web.

La aplicación estará disponible en la url <http://localhost:8085>

Es hora de comenzar creando la estructura de archivo, comenzamos con /index.html

<!DOCTYPE html>

<html>

<head>

<title>ShipFighter</title>

<script src=“dist/game.js"></script>

</head>

<body>

<div id="game"></div>

</body>

</html>

Y ahora con /src/game.ts

import "phaser";

import { GameScene } from "./mainscene";

const config: Phaser.Types.Core.GameConfig = {

title: "ShipFighter",

width: 800,

height: 600,

parent: "game", //HTML div id

physics: {

default: "arcade", //physics engine

arcade: {

debug: false,

gravity: { y: 0 }

}

},

scene: [GameScene], //Class Scene

backgroundColor: "#000"

};

export class ShipFighterGame extends Phaser.Game {

constructor(config: Phaser.Types.Core.GameConfig) {

super(config);

}

}

window.onload = () => {

console.log(config)

var game = new ShipFighterGame(config);

};

Para ver todas las opciones a la hora de crear un juego podemos consultar <https://photonstorm.github.io/phaser3-docs/global.html#GameConfig>

Podemos observar que carga un archivo de escenas. Para ello creamos /src/mainscene.ts con el siguiente contenido:

import "phaser";

export class GameScene extends Phaser.Scene {

constructor() {

super({key: "GameScene"}); //name scene

}

/\*\*

\* It is called when scene starts

\* @param params

\*/

init(params): void {

// TODO

}

/\*\*

\* It is called before the objects are created.

\* It contains loading assets and cacheable objects

\*/

preload(): void {

// TODO

}

/\*\*

\* It is called when all assets are loaded. Normally, it creates the objects

\*/

create(): void {

// TODO

}

/\*\*

\* It is create each frame or time. It contains all dymamic stuff.

\* @param time

\*/

update(time): void {

// TODO

}

}

Si ahora ejecutamos:

npm start

El resultado será una: bonita pantalla negra.



### Phaser y Websockets

Vamos a emplear la tecnología sockets.io que vimos anteriormente para mantener las sincronizados los diferentes jugadores.

Para ello comenzamos instalando sockets.io:

npm install --save socket.io

npm install @types/socket.io -s

Nos centramos en el archivo mainscene.ts que es importado por el archivo de entrada game.ts. El código de mainscene.ts queda:

import "phaser";

import {eventsName} from "../src/commons/eventsName"

import \* as io from "socket.io-client";

export class GameScene extends Phaser.Scene {

cursors: any;

ship: any;

treasure: Phaser.Physics.Arcade.Image;

otherPlayers: any;

socket: any;

blueScoreText: any;

redScoreText: any;

client: SocketIO.Client;

logo: any;

constructor() {

super({ key: "GameScene" }); //name scene

}

/\*\*

\* It is called when scene starts

\* @param params

\*/

init(params): void {

// TODO

console.log("Scene started");

}

/\*\*

\* It is called before the objects are created.

\* It contains loading assets and cacheable objects

\*/

preload(): void {

// TODO

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

console.log("preload");

this.load.image("ship", "assets/me.png");

this.load.image("otherPlayer", "assets/enemy.png");

this.load.image("treasure", "assets/treasure.png");

}

/\*\*

\* It is called when all assets are loaded. Normally, it creates the objects

\*/

create(): void {

this.logo=this.add.image(400, 300, 'logo');

this.socket = io("http://localhost:8080"); //conecting to the same server?

//We create a physics group for all the rest of player. The simplier way to

//detect collisions...

this.otherPlayers = this.physics.add.group();

// We will handle player input by using Phaser’s built-in keyboard manager.

this.cursors = this.input.keyboard.createCursorKeys();

this.socket.on(eventsName.CURRENTPLAYERS, (players) => {

//When this function is called, we loop through each of the players

//and we check to see if that player’s id matches the current player’s socket id.

//To loop through the players, we use Object.keys() to create an array of all the keys

//in the Object that is passed in. With the array that is returned we use the

//forEach() method to loop through each item in the array.

Object.keys(players).forEach((id) => {

if (players[id].playerId === this.socket.id) {

//Lastly, we called the addPlayer() function and passed it the current player’s

//information, and a reference to the current scene.

this.addPlayer(players[id]);

} else {

//We updated the function that is called when the currentPlayers event is emitted

//to now call the addOtherPlayers function when looping through the players object

//if that player is not the current player.

this.addOtherPlayers(players[id]);

}

});

});

this.socket.on(eventsName.NEWPLAYER, (playerInfo) => {

//When the newPlayer event is fired, we call the addOtherPlayers function

//to add that new player to our game.

this.addOtherPlayers(playerInfo);

});

//When the disconnect event is fired, we take that player’s id and we remove

//that player’s ship from the game. We do this by calling the getChildren() method on our

//otherPlayers group. The getChildren() method will return an array of all the game objects

//that are in that group, and from there we use the forEach() method to loop through that array.

this.socket.on(eventsName.DISCONNECT, (playerId) => {

this.otherPlayers.getChildren().forEach((otherPlayer) => {

if (playerId === otherPlayer.playerId) {

//Lastly, we use the destroy() method to remove that game object from the game.

otherPlayer.destroy();

}

});

//we will need to update the client side code to listen for this new event,

//and when this event is emitted

this.socket.on(eventsName.PLAYERMOVEMENT, (playerInfo) => {

this.otherPlayers.getChildren().forEach((otherPlayer) => {

if (playerInfo.playerId === otherPlayer.playerId) {

otherPlayer.setRotation(playerInfo.rotation);

otherPlayer.setPosition(playerInfo.x, playerInfo.y);

}

});

});

});

//In order to display the two teams scores, we will use Phaser’s Text Game Object.

//We created two new text game objects by calling this.add.text().

//When we created these two objects, we passed the location of where the object would be placed,

//the default text of the object, and the font size and fill that would be used for the text object.

this.blueScoreText = this.add.text(16, 16, "", {

fontSize: "32px",

fill: "#4444FF"

});

this.redScoreText = this.add.text(580, 16, "", {

fontSize: "32px",

fill: "#FF4444"

});

//When the scoreUpdate event is received, we then update the text of the game objects

//by calling the setText() method, and we pass the team’s score to each object.

this.socket.on(eventsName.SCOREUPDATE, (scores) => {

this.blueScoreText.setText("Blue: " + scores.blue);

this.redScoreText.setText("Red: " + scores.red);

});

//We check to see if the star object exists, and if it does, we destroy it

//We add a new star game object to the player’s game, and we use the information

//passed to the event to populate its location

//Lastly, we added a check to see if the player’s ship and the star object are

//overlapping, and if they are, then we emit the starCollected event.

//By calling physics.add.overlap, Phaser will automatically check for the overlap

//and run the provided function when there is an overlap.

this.socket.on(eventsName.TREASURELOCATION, (treasureLocation) => {

if (this.treasure) this.treasure.destroy();

this.treasure = this.physics.add.image(

//this.physics.add.image(

treasureLocation.x,

treasureLocation.y,

"treasure"

);

this.physics.add.overlap(

this.ship,

this.treasure,

()=> {

this.socket.emit(eventsName.TREASURECOLLECTED);

},

null,

self

);

});

this.logo.destroy();

}

addPlayer(playerInfo) {

//Created our player’s ship by using the x and y coordinates

//that we generated in our server code.

//Instead of just using self.add.image to create our player’s ship,

//we used self.physics.add.image in order to allow that game object to use the arcade physics.

//We used setOrigin() to set the origin of the game object to be in the middle

//of the object instead of the top left. The reason we did this because,

//when you rotate a game object, it will be rotated around the origin point.

//We used setDisplaySize() to change the size and scale of the game object.

//Originally, our ship image was 106×80 px, which was a little big for our game.

//After calling setDisplaySize() the image is now 53×40 px in the game.

this.ship = this.physics.add

.image(playerInfo.x, playerInfo.y, "ship")

.setOrigin(0.5, 0.5)

.setDisplaySize(53, 40);

//We used setTint() to change the color of the ship game object,

//and we choose the color depending on the team that was generated

//when we created our player info on the server.

if (playerInfo.team === "blue") {

this.ship.setTint(0xaaaaff);

} else {

this.ship.setTint(0xffaaaa);

}

//Lastly, we used setDrag, setAngularDrag, and setMaxVelocity to modify

//how the game object reacts to the arcade physics.

//Both setDrag and setAngularDrag are used to control the amount of resistance

//the object will face when it is moving.

//setMaxVelocity is used to control the max speed the game object can reach.

this.ship.setDrag(100);

this.ship.setAngularDrag(100);

this.ship.setMaxVelocity(200);

}

//This code is very similar to the code we added in the addPlayer() function.

//The main difference is that we added the other player’s game object to our

//otherPlayers group.

addOtherPlayers(playerInfo) {

const otherPlayer = this.add

.sprite(playerInfo.x, playerInfo.y, "otherPlayer")

.setOrigin(0.5, 0.5)

.setDisplaySize(53, 40);

if (playerInfo.team === "blue") {

otherPlayer.setTint(0xaaaaff);

} else {

otherPlayer.setTint(0xffaaaa);

}

otherPlayer["playerId"] = playerInfo.playerId;

this.otherPlayers.add(otherPlayer);

}

/\*\*

\* It is create each frame or time. It contains all dymamic stuff.

\* @param time

\*/

update() {

//We are checking if the left, right, or up keys are pressed down.

if (this.ship) {

//If the left or right key is pressed, then we update the player’s angular velocity

//by calling setAngularVelocity(). The angular velocity will allow the ship to rotate left and right.

if (this.cursors.left.isDown) {

this.ship.setAngularVelocity(-150);

} else if (this.cursors.right.isDown) {

this.ship.setAngularVelocity(150);

} else {

//if neither the left or right keys are pressed, then we reset the angular velocity back to 0.

this.ship.setAngularVelocity(0);

}

//If the up key is pressed, then we update the ship’s velocity, otherwise, we set it to 0.

if (this.cursors.up.isDown) {

this.physics.velocityFromRotation(

this.ship.rotation + 1.5,

100,

this.ship.body.acceleration

);

} else {

this.ship.setAcceleration(0);

}

//Lastly, if the ship goes off screen we want it to appear on the other side of the screen.

//We do this by calling physics.world.wrap() and we pass it the game object we want to wrap and an offset.

this.physics.world.wrap(this.ship, 5);

//We created three new variables and use them to store information about the player.

// emit player movement

let x = this.ship.x;

let y = this.ship.y;

let r = this.ship.rotation;

//We then check to see if the player’s rotation or position has changed by comparing

//these variables to the player’s previous rotation and position. If the player’s

//position or rotation has changed, then we emit a new event called playerMovement

//and pass it the player’s information.

if (

this.ship.oldPosition &&

(x !== this.ship.oldPosition.x ||

y !== this.ship.oldPosition.y ||

r !== this.ship.oldPosition.rotation)

) {

this.socket.emit(eventsName.PLAYERMOVEMENT, {

x: this.ship.x,

y: this.ship.y,

rotation: this.ship.rotation

});

}

// save old position data

//Lastly, we store the player’s current rotation and position.

this.ship.oldPosition = {

x: this.ship.x,

y: this.ship.y,

rotation: this.ship.rotation

};

}

}

}

Es interesante analizar el código para observar cómo se gestionan los eventos y la física del juego.

Como se observa requiere de una carpeta /assets con cuatro imágenes:

* enemy.png para representar a los otros jugadores.
* me.png para representar al jugador.
* treasure.png para representar el tesoro a capturar.
* logo.png para la imagen splash.

Además, comparte con el backend la carpeta commons por lo cual la copiaremos tal cual a la carpeta frontend.

Para realizar las pruebas en local tendremos en una terminal el servicio arrancado, para ello nos ubicaremos en la carpeta backend y ejecutaremos:

npm run prod

Por defecto, en local, escuchará por el puerto 8080. Es por este motivo, por el cual vemos en el archivo mainscene.ts la línea:

this.socket = io("http://localhost:8080");

Debido a que tenemos configurado el servidor del cliente para ejecutarse en el puerto 8085.

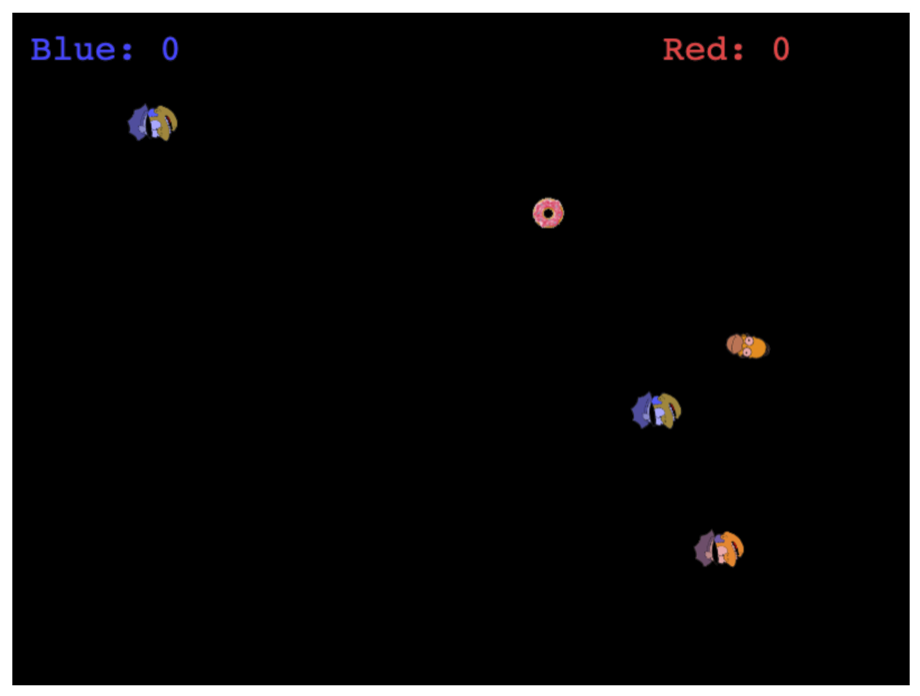
Para ejecutar el cliente, en otra terminal y en la carpeta frontend ejecutaremos:

npm start

Ya podemos visitar con tantos navegadores como queramos la dirección: <http://localhost:8085>.

Cada jugador será de un equipo y sumará puntos para él.

Los jugadores del equipo rojo están tintados levemente de rojo e igual ocurre con el equipo azul.

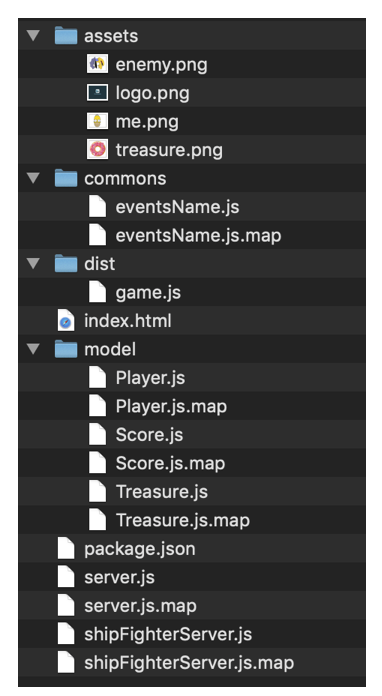


### Publicación

Existen varias consideraciones para el despliegue.

En primer lugar, vamos a realizarlo en Heroku siguiendo los pasos que se encuentran en el apartado anterior: Sockets.

Para realizar el despliegue vamos a empaquetar todo nuestro código en una carpeta que debe contener los siguientes archivos:



Como vemos hemos fusionado tanto el backend como el frontend. De hecho el servidor de sockets se va a comportar como servidor web, para ello vamos a añadir el siguiente código al archivo shipFighterServer.ts:

…

this.\_app.options('\*', cors());

/\*\* web server for heroku \*/

this.\_app.use("/dist",express.static('dist'));

this.\_app.use("/assets",express.static('assets'));

//told the server to serve the index.html file as the root page.

this.\_app.get('/', (req, res)=> {

res.sendFile(\_\_dirname + '/index.html');

});

this.server = createServer(this.\_app);

…

Nota: antes de copiar a la carpeta anterior debemos ejecutar:

npm run prod

Para compilar el código.

El cliente también debemos realizar un cambio en el archivo mainscene.ts:

Sustituir

this.socket = io("http://localhost:8080");

Por

this.socket = io();

Para que se conecte a localhost por el puerto adecuado.

Finalmente el archivo package.json debe tener el siguiente contenido:

{

"name": "server",

"version": "1.0.0",

"description": "Online Server for Phaser by using Socket.io",

"main": "server.js",

"scripts": {

"dev": "ts-node-dev --respawn --transpileOnly ./src/server.ts",

"prod": "tsc && node ./dist/server.js",

"start": "node server.js"

},

"author": "Developodo",

"license": "ISC",

"devDependencies": {

"@types/express": "^4.17.0",

"@types/socket.io": "^2.1.2",

"express": "^4.17.1",

"socket.io": "^2.2.0",

"ts-node-dev": "^1.0.0-pre.40",

"typescript": "^3.5.3"

},

"dependencies": {

"cors": "^2.8.5",

"express": "^4.17.1",

"socket.io": "^2.2.0",

"typescript-tslint-plugin": "^0.5.4"

}

}

Una vez desplegado ya tenemos nuestro juego online disponible: <https://shipfighter.herokuapp.com/>