Drunken Pirates

Ismail Matteh, Muhammad Ibrahim Khan, Farukh Shaikh

Muhammad Suhaib, Hafiz Muhammad Aafaq

Table of Contents

Introduction

- A multiplayer game based on NodeJS, ExpressJS and Socket.IO
- Any player can enter the game by using the game’s hyperlink
- This is a competitive game where players contend against each other to amass more points

Tools Used

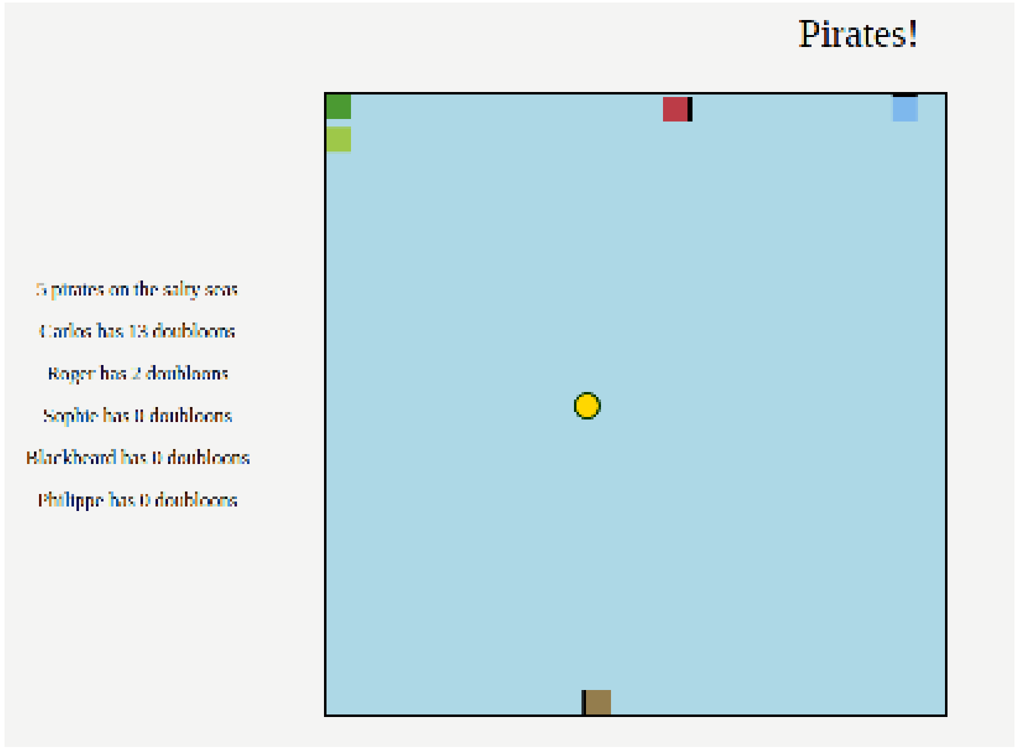
NodeJS for manipulating high-scores and other information on the server-side

ExpressJS to facilitate creation of the Web Application.

Socket.IO for real time, bidirectional, event-driven communication

Gameplay

- Any player can enter the game by using the game's hyperlink
- The theme of the game revolves around pirates
- Every player controls a pirate ship, and tries to collect gold coins



Pirates!

5 pirates on the salty seas

Carlos has 13 doubloons

Roger has 2 doubloons

Sophie has 0 doubloons

Blackbeard has 0 doubloons

Philippe has 0 doubloons

Gameplay continued

- Players must compete with others to obtain the coins (called doubloons within the game) first
- The game features collision detection and momentum transfer between players
- In following with the theme of pirate ships, turning motions carry lots of momentum, so sharp turns aren’t possible
- The arrow keys are used to move the ship

Coding Section

The Coding part is divided into 3 parts:

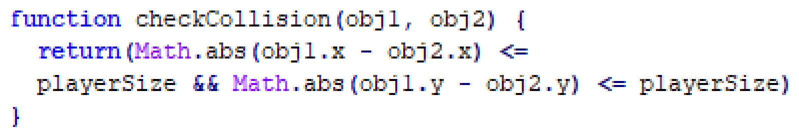
- Controllers Section
- Client Section
- Display Section

Controller Section

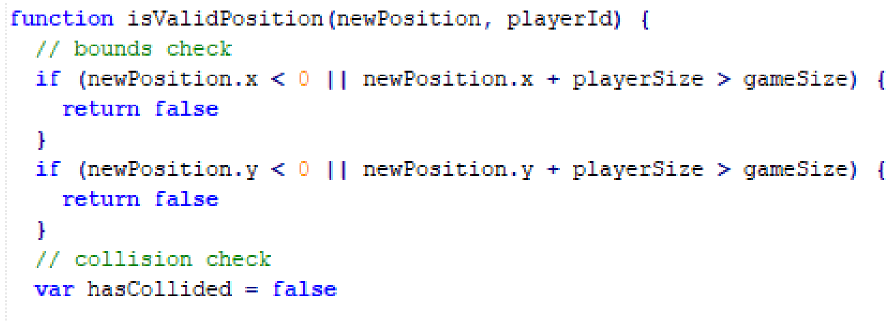
The central controller for the game, responsible for collision detection, position checking, and coin placement.

The checkCollision function checks for collision between two objects with their bounds.

The isValidPosition function makes sure the player does not exit the game board bounds.



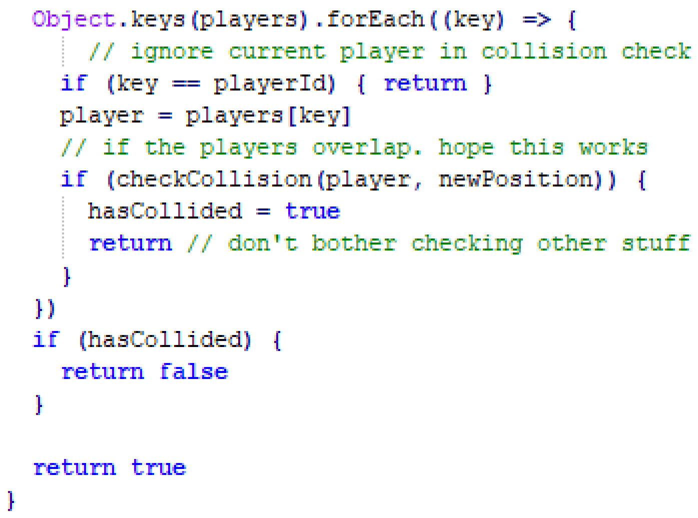
```
function checkCollision(obj1, obj2) {
  return(Math.abs(obj1.x - obj2.x) <=
  playerSize && Math.abs(obj1.y - obj2.y) <= playerSize)
}
```



```
function isValidPosition(newPosition, playerId) {
  // bounds check
  if (newPosition.x < 0 || newPosition.x + playerSize > gameSize) {
    return false
  }
  if (newPosition.y < 0 || newPosition.y + playerSize > gameSize) {
    return false
  }
  // collision check
  var hasCollided = false
```

Controller Section

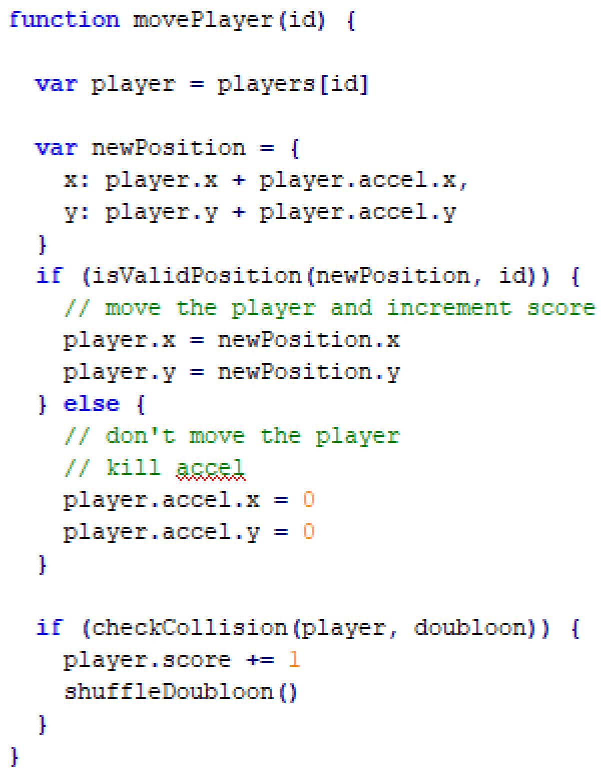
Placing the coin (doubloon) is just a simple function that randomly places it within valid bounds (the size of the playing board).



```
  Object.keys(players).forEach((key) => {
      // ignore current player in collision check
    if (key == playerId) { return }
    player = players[key]
    // if the players overlap. hope this works
    if (checkCollision(player, newPosition)) {
      hasCollided = true
      return // don't bother checking other stuff
    }
  })
  if (hasCollided) {
    return false
  }

  return true
}
```

Controller Section

The movePlayer function changes the ship’s position on the screen, it checks if the ship is in valid position and it also checks for the doubloons collisions.



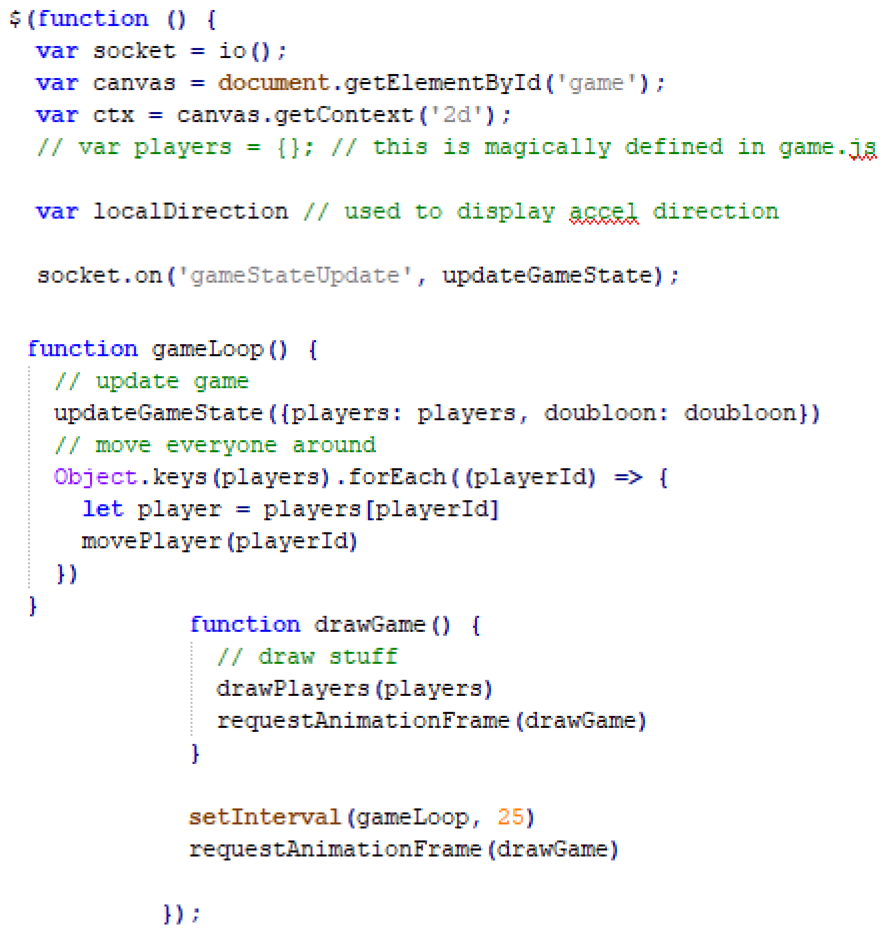
```
function movePlayer(id) {

  var player = players[id]

  var newPosition = {
    x: player.x + player.accel.x,
    y: player.y + player.accel.y
  }
  if (isValidPosition(newPosition, id)) {
    // move the player and increment score
    player.x = newPosition.x
    player.y = newPosition.y
  } else {
    // don't move the player
    // kill accel
    player.accel.x = 0
    player.accel.y = 0
  }

  if (checkCollision(player, doubloon)) {
    player.score += 1
    shuffleDoubloon()
  }
}
```

Client Section

A jQuery function that includes several different functions used at the client side.

The gameLoop function repeatedly updates the game state and renders the game board.

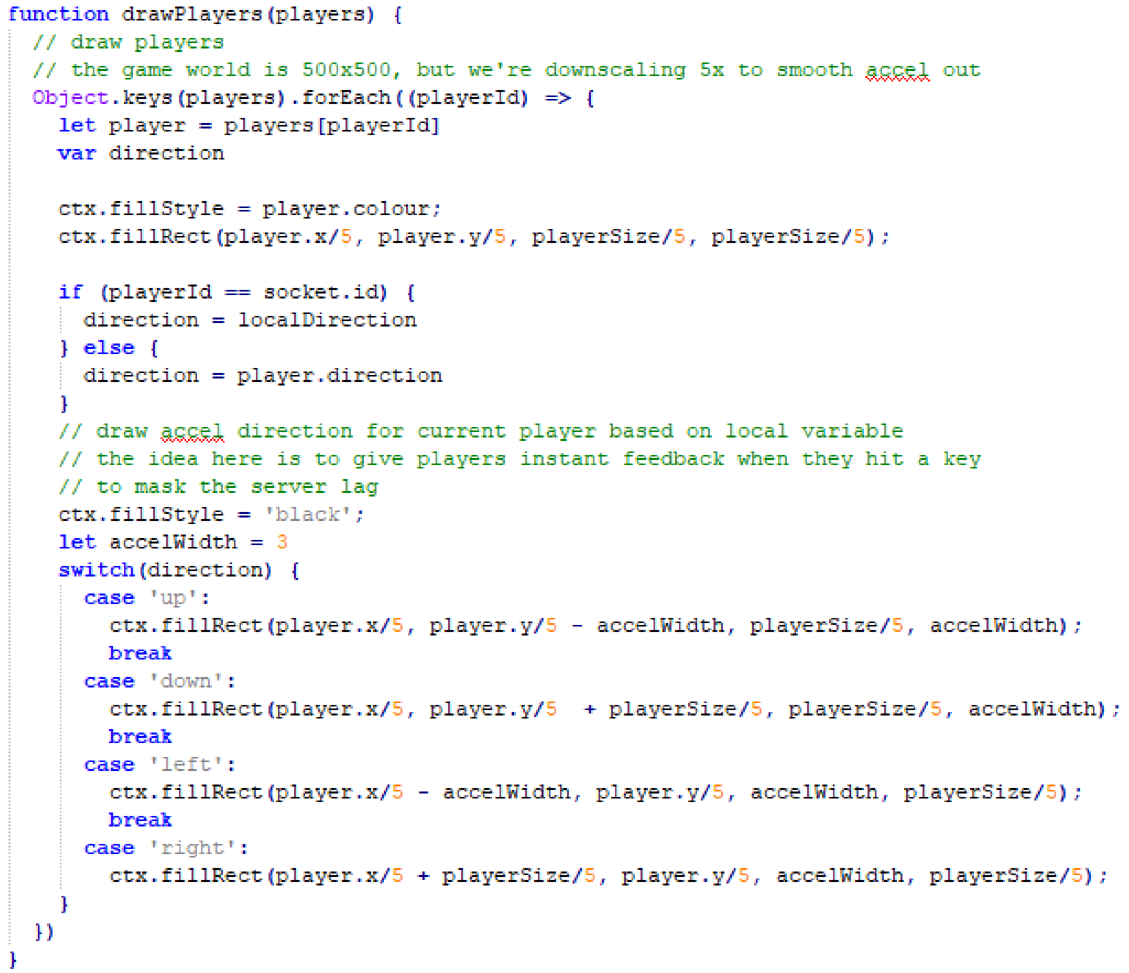
The drawGame function repeatedly renders the game board at fixed intervals.



```
$(function () {
  var socket = io();
  var canvas = document.getElementById('game');
  var ctx = canvas.getContext('2d');
  // var players = {}; // this is magically defined in game.js

  var localDirection // used to display accel direction

  socket.on('gameStateUpdate', updateGameState);

 function gameLoop() {
   // update game
   updateGameState({players: players, doubloon: doubloon})
   // move everyone around
   Object.keys(players).forEach((playerId) => {
     let player = players[playerId]
     movePlayer(playerId)
   })
 }
             function drawGame() {
               // draw stuff
               drawPlayers(players)
               requestAnimationFrame(drawGame)
             }

             setInterval(gameLoop, 25)
             requestAnimationFrame(drawGame)

           });
```

Client Section

The drawPlayers function renders the players along with the direction of their acceleration.

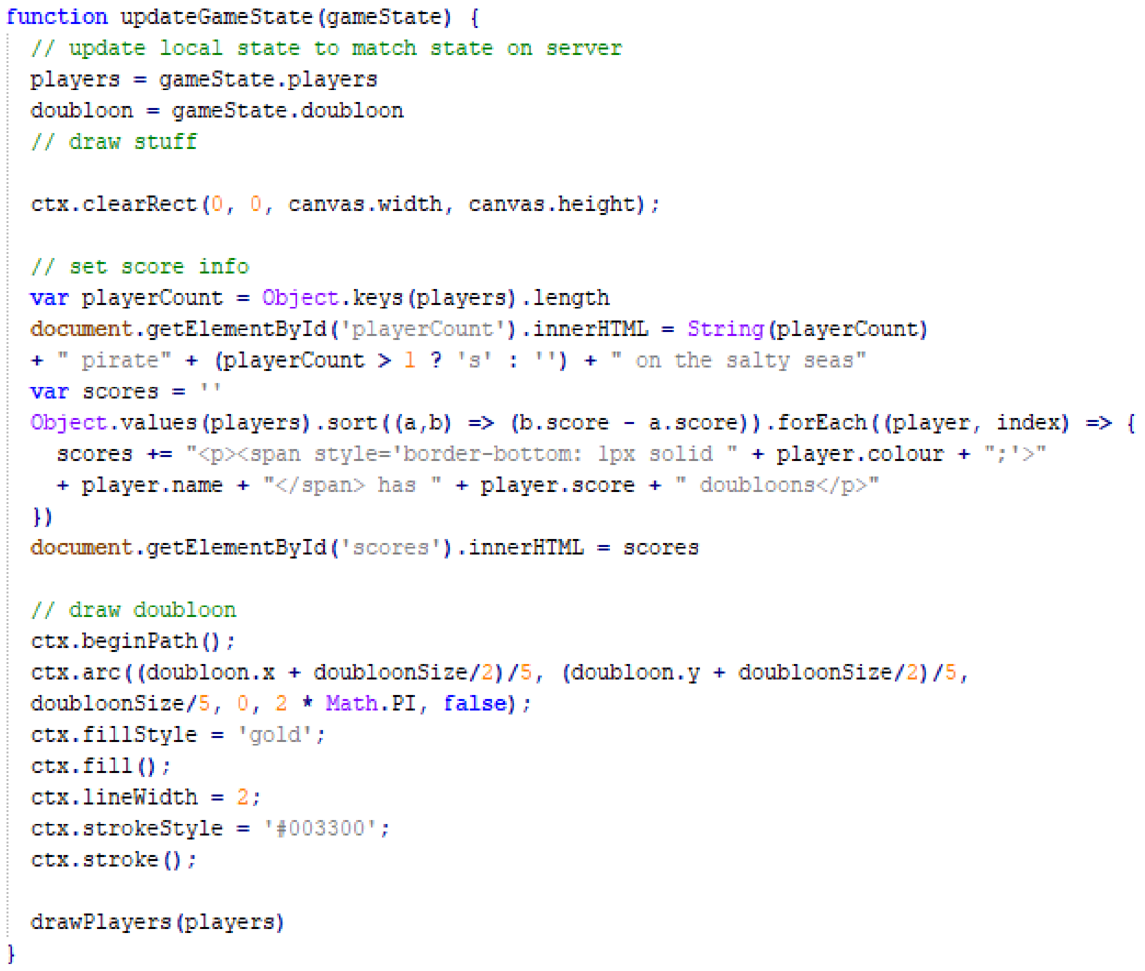


```
function drawPlayers(players) {
  // draw players
  // the game world is 500x500, but we're downscaling 5x to smooth accel out
  Object.keys(players).forEach((playerId) => {
    let player = players[playerId]
    var direction

    ctx.fillStyle = player.colour;
    ctx.fillRect(player.x/5, player.y/5, playerSize/5, playerSize/5);

    if (playerId == socket.id) {
      direction = localDirection
    } else {
      direction = player.direction
    }
    // draw accel direction for current player based on local variable
    // the idea here is to give players instant feedback when they hit a key
    // to mask the server lag
    ctx.fillStyle = 'black';
    let accelWidth = 3
    switch(direction) {
      case 'up':
        ctx.fillRect(player.x/5, player.y/5 - accelWidth, playerSize/5, accelWidth);
        break
      case 'down':
        ctx.fillRect(player.x/5, player.y/5  + playerSize/5, playerSize/5, accelWidth);
        break
      case 'left':
        ctx.fillRect(player.x/5 - accelWidth, player.y/5, accelWidth, playerSize/5);
        break
      case 'right':
        ctx.fillRect(player.x/5 + playerSize/5, player.y/5, accelWidth, playerSize/5);
    }
  })
}
```

Client Section

The updateGameState function simply updates the values of the client side with the values stored at the server.

These updated values are then rendered to the game board.



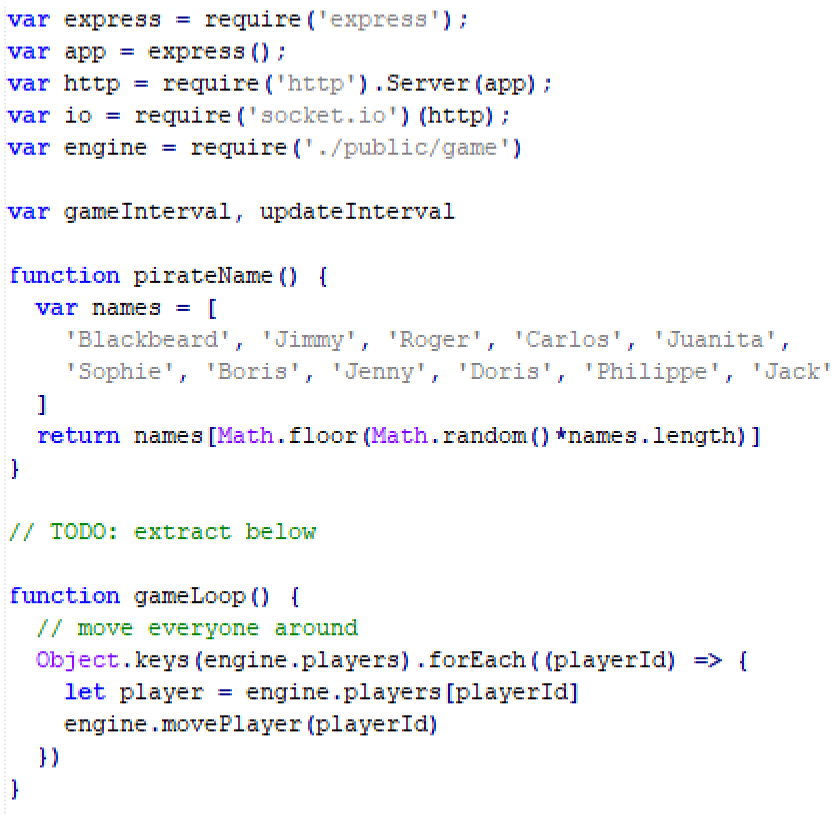
```
function updateGameState(gameState) {
  // update local state to match state on server
  players = gameState.players
  doubloon = gameState.doubloon
  // draw stuff

  ctx.clearRect(0, 0, canvas.width, canvas.height);

  // set score info
  var playerCount = Object.keys(players).length
  document.getElementById('playerCount').innerHTML = String(playerCount)
  + " pirate" + (playerCount > 1 ? 's' : '') + " on the salty seas"
  var scores = ''
  Object.values(players).sort((a,b) => (b.score - a.score)).forEach((player, index) => {
    scores += "<p><span style='border-bottom: 1px solid " + player.colour + ";'>"
    + player.name + "</span> has " + player.score + " doubloons</p>"
  })
  document.getElementById('scores').innerHTML = scores

  // draw doubloon
  ctx.beginPath();
  ctx.arc((doubloon.x + doubloonSize/2)/5, (doubloon.y + doubloonSize/2)/5,
  doubloonSize/5, 0, 2 * Math.PI, false);
  ctx.fillStyle = 'gold';
  ctx.fill();
  ctx.lineWidth = 2;
  ctx.strokeStyle = '#003300';
  ctx.stroke();

  drawPlayers(players)
}
```

Display Section

Requiring all the modules.

The pirateName function randomly assigns predefined names to the players that joins the game.

The gameLoop function moves the players on the screen.

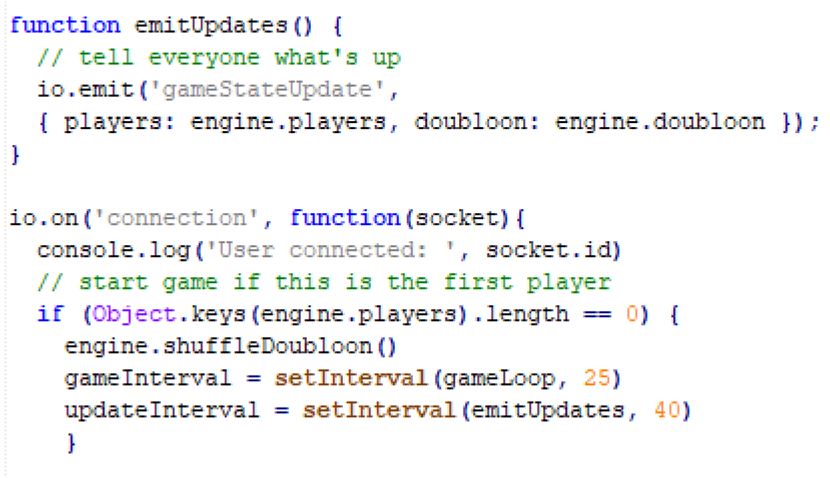


```
var express = require('express');
var app = express();
var http = require('http').Server(app);
var io = require('socket.io')(http);
var engine = require('./public/game')

var gameInterval, updateInterval

function pirateName() {
  var names = [
    'Blackbeard', 'Jimmy', 'Roger', 'Carlos', 'Juanita',
    'Sophie', 'Boris', 'Jenny', 'Doris', 'Philippe', 'Jack'
  ]
  return names[Math.floor(Math.random()*names.length)]
}

// TODO: extract below

function gameLoop() {
  // move everyone around
  Object.keys(engine.players).forEach((playerId) => {
    let player = engine.players[playerId]
    engine.movePlayer(playerId)
  })
}
```

Display Section

This section display the main JS file that runs on the client's browser.

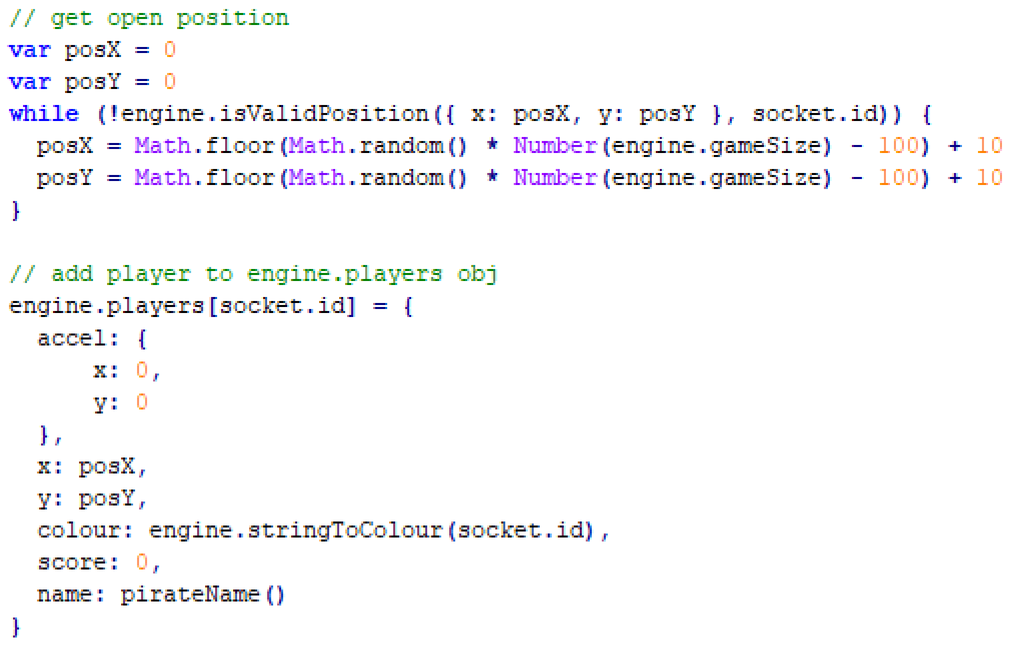
The emitUpdates function sets up the EventEmitter for updating game state.



```
function emitUpdates() {
  // tell everyone what's up
  io.emit('gameStateUpdate',
  { players: engine.players, doubloon: engine.doubloon });
}

io.on('connection', function(socket){
  console.log('User connected: ', socket.id)
  // start game if this is the first player
  if (Object.keys(engine.players).length == 0) {
    engine.shuffleDoubloon()
    gameInterval = setInterval(gameLoop, 25)
    updateInterval = setInterval(emitUpdates, 40)
    }
```

Display Section

Randomly spawn a new ship in some free space on the screen when a new player joins the game and adds the new player on all client’s screen.



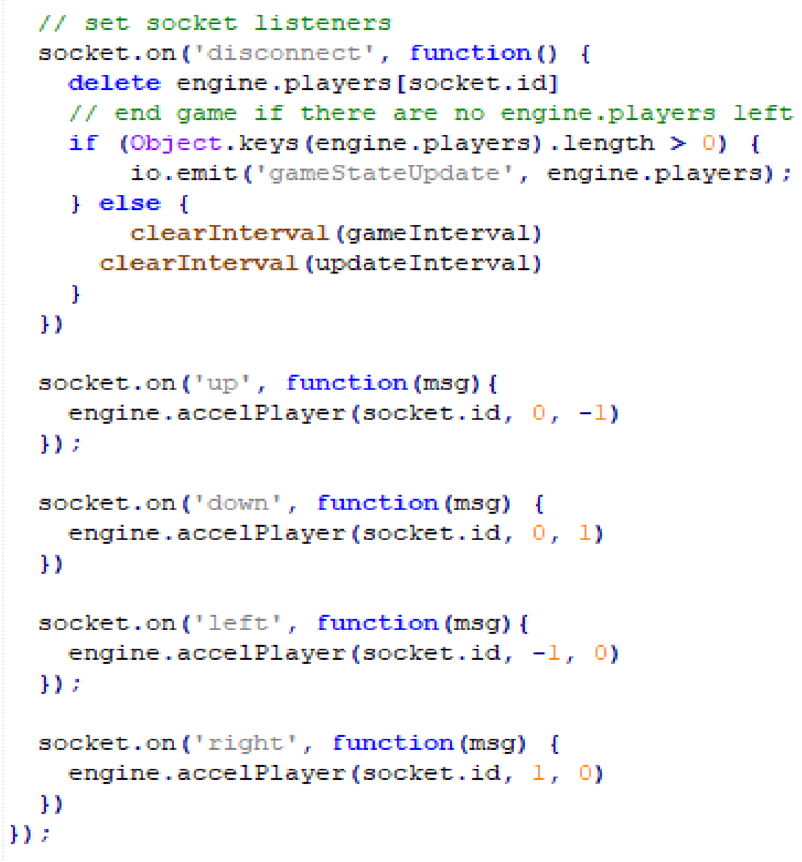
```
// get open position
var posX = 0
var posY = 0
while (!engine.isValidPosition({ x: posX, y: posY }, socket.id)) {
  posX = Math.floor(Math.random() * Number(engine.gameSize) - 100) + 10
  posY = Math.floor(Math.random() * Number(engine.gameSize) - 100) + 10
}

// add player to engine.players obj
engine.players[socket.id] = {
  accel: {
      x: 0,
      y: 0
  },
  x: posX,
  y: posY,
  colour: engine.stringToColour(socket.id),
  score: 0,
  name: pirateName()
}
```

Display Section

Various event emitters have been set up based on the socket events.

Upon disconnecting, the gameState is updated to remove the player that disconnected from the board.

Movement controls modify the player’s acceleration direction. For example, pressing the ‘up’ arrow will cause upward acceleration



```
 // set socket listeners
 socket.on('disconnect', function() {
   delete engine.players[socket.id]
   // end game if there are no engine.players left
   if (Object.keys(engine.players).length > 0) {
       io.emit('gameStateUpdate', engine.players);
   } else {
       clearInterval(gameInterval)
     clearInterval(updateInterval)
   }
 })

 socket.on('up', function(msg){
   engine.accelPlayer(socket.id, 0, -1)
 });

 socket.on('down', function(msg) {
   engine.accelPlayer(socket.id, 0, 1)
 })

 socket.on('left', function(msg){
   engine.accelPlayer(socket.id, -1, 0)
 });

 socket.on('right', function(msg) {
   engine.accelPlayer(socket.id, 1, 0)
 })
});
```

Problems Encountered

- Socket.IO has a large overhead. It’s a wrapper around WebSockets with a large amount of added functionality such as rooms, disconnects and reconnects, and fallback solutions in case the WebSocket connection fails.
- While useful for applications with fewer timing constraints such as a messaging app, it does cause performance issues for real-time games.
- An improvement would be to directly use WebSockets as most additional functionality of Socket.IO is unneeded in the game.

Problems Encountered

- Data between the client and server is transmitted in the form of serialized JSON, which is extremely inefficient in terms of bandwidth use.
- For real-time applications, a better solution is to use Protocol Buffers (protobuf) to serialize and send the data. This has a much lower bandwidth use compared to serialized JSON.

Conclusion

This was a very practical assignment.

We saw how a project can use already created libraries and frameworks such as ExpressJS and Socket.IO to rapidly create feature rich web applications.

We also learned about how Socket.IO works, and about other modes of client-server bidirectional communication as well such as WebSockets.

Thanks!

Contact us:

Muhammad Suhaib

Hafiz Muhammad Aafaq Tanveer **R**ana

Ismail Matteh

Muhammad Ibrahim Afsar Khan

Farukh Shaikh