Projeto de Interação Web em Tempo Real e Multi-utilizador
Abril 2020

José Lima n º 20732
Miguel Barbosa nº 16732

IPVC-ESTG

April 15, 2020

1 Resumo

Este trabalho foi-nos requisitado no âmbito da cadeira de Tecnologias Interativas e consiste na criação de uma aplicação de entretenimento com interação web e em tempo real.

Para este trabalho, após uma pesquisa, escolhemos utilizar o seguinte software:

- **Phaser3**;
- **Socket.io**;
- **Node.js & Express**;
- **GitHub**.

1.1 Phaser3

Phaser é uma estrutura de jogo 2D de software livre para criar jogos HTML5 para desktop e telemóvel. É desenvolvido pela Photon Storm.

O Phaser usa os renderizadores **Canvas** e **WebGL** internamente e pode alternar automaticamente entre eles com base no suporte ao navegador. Isso permite uma renderização rápida em computadores e dispositivos móveis.

Na parte de renderização e scripting usa a biblioteca **Pixi.js**. O Phaser pode ser renderizado em **WebGL** ou **Canvas**, dependendo do tipo de navegador utilizado. No scripting as aplicações podem ser feitas em **JavaScript** ou **TypeScript**.

Em termos de físicas o Phaser fornece o **Arcade Physics**, o **Ninja Physics** e o **P2.JS** que é um sistema de física de corpo inteiro. Para áudio e animação o Phaser fornece um sistema de **Spritesheet**, um **Texture Atlas** e com isto é possível criar uma sequência de animação. Para áudio é usado o **HTML 5** para reproduzir.

1.2 Socket.io

Socket.IO é uma biblioteca JavaScript para aplicações da web em tempo real. Permite comunicação bidirecional em tempo real entre clientes e servidores da Web. Ele tem duas partes: uma biblioteca do lado do cliente que é executada no navegador e uma biblioteca do lado do servidor para o Node.js. Ambos os componentes têm uma API quase idêntica.

O Socket.IO usa principalmente o protocolo WebSocket com pesquisa como opção de fallback, enquanto fornece a mesma interface. Embora possa ser usado como simplesmente um invólucro para o WebSocket, ele oferece muitos outros recursos, incluindo transmissão para vários sockets e armazenamento de dados associados a cada cliente.

1.3 Node.js & Express

Node.js é um interpretador de JavaScript em *open source* orientado a eventos, criado por Ryan Dahl.

O Node.js foca-se em migrar a programação do Javascript do cliente (frontend) para os servidores, criando aplicações de alta fidelidade (como um servidor web), manipulando milhares de conexões/eventos simultâneas em tempo real numa única máquina física.

A principal característica e diferença de outras tecnologias (como PHP, Java, C#) é a execução das requisições/eventos em single-thread, onde apenas uma thread (chamada de Event Loop) é responsável por executar o código Javascript, sem a necessidade de criar nova thread que utilizaria mais recursos computacionais (por exemplo memória RAM) e sem o uso da fila de espera.

O Express não é mais do que uma framework do Node.js que visa facilitar a integração do Node em aplicações Web e Mobile.

1.4 GitHub

GitHub é uma plataforma de hospedagem de código-fonte com controle de versão usando o Git. Ele permite que programadores, utilitários ou qualquer utilizador registado na plataforma contribuam em projetos privados e/ou Open Source de qualquer lugar do mundo. GitHub é amplamente utilizado por programadores para divulgação de seus trabalhos ou para que outros programadores contribuam com o projeto, além de promover fácil comunicação através de recursos que relatam problemas ou repositórios remotos (issues, pull request).
Link do GitHub para o repositório: `https://github.com/BoneMoon/trabalhoTI`

2 Introdução

O objetico deste trabalho é o desenvolvimento de um jogo em ecrã público, multi-utilizador, com controlo de interação através de dispositivo móvel. Onde o jogo deverá ser:

- simples de jogar, com um objetivo de interação através de despositivo móvel
- de interação intuitiva / auto-explicativa
- rápido
- tolerante aos atrasos de latência nas comunicações
- explorar a colaboração e/ou competição
- possível usar gráficos simples e com pouco detalhe

Neste relatório vamos abortar as várias fases de desenvolvimento o que foi implementado nas mesma e ainda vamos falar sobre os resultados finais do jogo

3 Desenvolvimento

3.1 Manual do Utilizador

O objetivo deste jogo é conseguir chegar à porta com um tempo menor que o outro jogador. Para isso tem que conseguir saltar pelas plataformas sem tocar na lava nem nos espinhos, usando as setas que estão no ecrã.

3.2 Jogo

A primeira fase de implementação foi a criação do nosso mapa usando o TILED. Feito isso adicionamos o mapa ao nosso jogo.

```
// tilesets
const map = this.make.tilemap({ key: "mapas" });
const chao = map.addTilesetImage("chao", "chao");
const porta = map.addTilesetImage("lava_porta", "porta");
const lava = map.addTilesetImage("lava_porta", "lava");
const espinhos = map.addTilesetImage("espinhos", "espinhos");

// layers
this.mapa = map.createStaticLayer("chao", chao, 0, 0);
this.obj = map.createStaticLayer("porta", porta, 0, 0);
this.danoLava = map.createStaticLayer("lava", lava, 0, 0);
this.danoEsp = map.createStaticLayer("espinhos", espinhos, 0, 0);

// colisões com as layers
this.mapa.setCollisionByExclusion(-1, true);
this.danoLava.setCollisionByExclusion(-1, true);
this.danoEsp.setCollisionByExclusion(-1, true);
this.obj.setCollisionByExclusion(-1, true);
```

A próxima funcionalidade a ser implementada foi adicionar os botões ao jogo e o que fazem quando são carregados que neste caso é o jogador andar de um lado para o outro.

```
// -- adionar botão direito
this.btndir = this.add.image(300, 500, "btndir").setInteractive();
this.btndir.setScale(0.2);
this.btndir.setScrollFactor(0);

this.btndir.on(
    "pointerover",
    function () {
        this.direita = true;
    },

    this
);
this.btndir.on(
    "pointerdown",
    function () {
        this.direita = true;
    },
    this
);

this.btndir.on(
    "pointerout",
    function () {
        this.direita = false;
    },

    this
);

this.btndir.emit("pointerover");
this.btndir.emit("pointerdown");
this.btndir.emit("pointerout");
```

Para cada botão inserimos uma variável booleana que vai corresponder a quando é que esse mesmo botão esta a ser carregado. Na funçao *update()* incrementamos a velocidade.

```
if (
    this.direita == false &&
    this.esquerda == false &&
    this.cima == false
) {
    this.player.body.setVelocityX(0);
    this.player.anims.stop();
} else if (
    this.direita == true &&
    this.esquerda == false &&
    this.cima == false
) {
    this.player.body.setVelocityX(180);
    this.player.anims.play("right", true);
    this.player.flipX = false;
}
if (
    this.esquerda == true &&
    this.direita == false &&
    this.cima == false
) {
    this.player.body.setVelocityX(-180);
    this.player.anims.play("left", true);
    this.player.flipX = true;
}

if (
    this.cima == true &&
    this.esquerda == false &&
    this.direita == false &&
    this.player.body.onFloor()
) {
    this.player.setVelocityY(-250);
}
```

E por último (o resto das funcionalidades estão implementadas em conjunto com o servidor) o texto do tempo. Para isso funcionar na função *update()* incrementamos a variável *timer* e assim de cada vez que essa mesma função é chamada o texto é atualizado.

```
this.timeOut = performance.now();
this.timer = 0;
```

```
this.timer++;
this.tempoText.setText("Tempo: " + this.timer);
```

3.3 Servidor

No servidor a primeira coisa que fizemos é quando um jogador se conecta com o jogo, criamos um array que guarda o x e o y onde que o jogador vai começar o jogo, o player id que vai corresponder ao socketID, a variável i que vai corresponder ao número de jogadores que vai incrementando sempre que um jogador entra e por fim o tempo.

```
io.on("connection", function (socket) {
    console.log("a user connected", socket.id);

    // create a new player and add it to our players object
    players[socket.id] = {
        x: /*30*/ Math.floor(Math.random() * 70),
        y: 400,
        playerId: socket.id,
        i: i++,
        tempo,
    };
```

Logo depois criamos um array *lista* onde lá dentro estará o array dos *players*. Fizemos isso para conseguirmos verificar o número de jogadores conectados e para que somente 2 jogadores possam jogar e começarem ao mesmo tempo.
Para isso começamos inicialmente por verificar o tamanho na lista e emitir os respetivos pacotes.

```
lista.push(players[socket.id]);
//console.log(lista);

if (lista.length < 2) {
    io.emit("espera");
}

if (lista.length == 2) {
    io.emit("ready");
}
```

Caso os jogadores conectados sejam menor 2, ou seja só 1, emitimos o *espera*. Este pacote vai fazer que quando o jogador se conectar vá para o jogo mas não vai conseguir fazer nada e vai aparecer uma mensagem dizendo "À espera de jogadores".

```
this.socket.on("espera", function () {
    self.wait();
});
```

```
wait: function () {
    this.jogo = 0;
    this.espera.setText("À espera de jogadores");
},
```

Caso se tenham conectado 2 jogadores emitimos o *ready*. Este pacote vai fazer com que ambos os jogadores comecem ao mesmo tempo. Para isso criamos uma variável *jogo*, quando essa variável for uma a 0 quer dizer que esta apenas 1 jogador conectado quando for 1 os jogadores conectados são 2. Quando isso acontece o texto que aparecia ao jogador que entrou sozinho desaparece e a função *update()* é chamada.

```
this.socket.on("ready", function () {
    self.start();
});
```

```
start: function () {
    this.jogo = 1;
    console.log(this.jogo);

    if (this.jogo == 1) {
        this.espera.setText("");
        this.update();
    }
},

update: function () {
    if (this.jogo == 1) {
```

Logo depois começamos a pôr os jogadores na ”scene” do jogo. Para isso emitimos 2 pacotes o *currentPlayers* e o *newPlayer*.

```
// send the players object to the new player
socket.emit("currentPlayers", players);

// update all other players of the new player
socket.broadcast.emit("newPlayer", players[socket.id]);
```

No *currentPlayers*, estamos a passar o objeto players para o novo player. Esses dados serão usados para preencher todos os sprites de jogadores no jogo do novo jogador.
No *newPlayer*, estamos a passar os dados do novo jogador para todos os outros jogadores, para que o novo sprite possa ser adicionado ao jogo.

```
this.socket.on("currentPlayers", function (players) {
    Object.keys(players).forEach(function (id) {
        if (players[id].playerId === self.socket.id) {
            self.addPlayer(self, players[id]);
        } else {
            self.addOtherPlayers(self, players[id]);
        }
    });
});
```

Usamos socket.on para escutar o *currentPlayers* e, quando esse evento é chamdo, a função que fornecemos será chamada com o objeto players que passamos do nosso servidor.
Quando essa função é chamada, percorremos cada um dos jogadores e verificamos se o ID desse jogador corresponde ao ID do socket do jogador atual.
Para percorrer os players, usamos *Object.keys()* para criar um array de todas as chaves do objeto que é transmitido. Com o array retornado, usamos o *forEach()* para percorrer cada item do array . Por fim, chamamos a função *addPlayer()* e passamos as informações do jogador atual e uma referência à cena atual.

Criamos um novo grupo chamado *addotherPlayers*, que será usado para gerenciar todos os outros jogadores do nosso jogo.
Atualizamos a função chamada quando o textitcurrentPlayers é emitido para agora chamar a função textitaddotherPlayers ao percorrer o objeto players se esse jogador não for o atual. Quando o *newPlayer* é acionado, chamamos a função *addotherPlayers* para adicionar esse novo jogador ao nosso jogo.

```
addPlayer: function (self, playerInfo) {
    self.player = self.physics.add.sprite(
        playerInfo.x,
        playerInfo.y,
        "player",
        6
    );
    self.player.setCollideWorldBounds(true);
    self.player.setScale(1.5);
    self.player.setVelocity(0);

    self.physics.add.collider(self.player, self.mapa);

    self.physics.add.collider(self.player, self.danoLava, () => {
        self.player.setPosition(30, 400);
    });

    self.physics.add.collider(self.player, self.danoEsp, () => {
        self.player.setPosition(30, 400);
    });

    self.physics.add.collider(self.player, self.obj, () => {
        self.player.disableBody(true, true);
        this.socket.emit("tempoFinal", self.timer);
    });
},
```

Na função *addPlayer()* e *addotherPlayers()* vamos juntar o player à nossa cena, para isso vamos usar as coordenadas x e y que estão geradas no servidor e depois vamos buscar o nosso sprite com a chave ”player”. Depois adionamos scale, velocidade, colisões com as bordas e ainda colisões com os objetos da cenas (lava, espinhos e porta).
Quando o jogador colide com a lava ou os espinhos o jogador volta ao inicio, para isso utilizamos o método *setPosition()* e dentro dele as coordenadas desejadas.

```
self.physics.add.collider(self.player, self.obj, () => {
    self.player.disableBody(true, true);
    this.socket.emit("tempoFinal", self.timer);
});
```

Para o fim do jogo fizemos que quando os dois jogadores colidem com a porta enviamos o tempo para o servidor emitindo o *tempoFinal* e no servidor vamos verificar qual deles ganhou.

```
socket.on("tempoFinal", function (timer) {
    players[socket.id].tempo = timer;
    listaTempo.push({ time: players[socket.id].tempo, socket });

    console.log(listaTempo);

    if (listaTempo.length ≥ 2) {
        if (listaTempo[0].time < listaTempo[1].time) {
            listaTempo[0].socket.emit("youWin");
            listaTempo[1].socket.emit("youLose");
            listaTempo = [];
        } else if (listaTempo[1].time < listaTempo[0].time) {
            listaTempo[1].socket.emit("youWin");
            listaTempo[0].socket.emit("youLose");
            listaTempo = [];
```

No servidor a primeira cisa que fazemos é guardar o tempo de cada jogador no array. Logo depois criamos um array chamado de listaTempo que nele vai estar o tempo e o socket de cada jogador.
Depois fizemos um *if()* para que só quando os dois jogadores tiverem colidido com a porta é que vamos emitir se ganhou ou perder. De seguida Fazemos a comparação entre eles, quem tiver o tempo mais baixo ganha e emitimos o *youWin* ou para quem perde *youLose*. Ambos estes pacotes vão emitir duas cenas diferentes onde estára texto a dizer ganhar ou perder.

```
// Emitir o movimento do jogador
var x = this.player.x;
var y = this.player.y;

if (
    this.player.oldPosition &&
    (x !== this.player.oldPosition.x ||
        y !== this.player.oldPosition.y)
) {
    this.socket.emit("playerMovement", {
        x: this.player.x,
        y: this.player.y,
    });
}

// Guardar a posição antiga
this.player.oldPosition = {
    x: this.player.x,
    y: this.player.y,
};
```

```
socket.on("playerMovement", function (movementData) {
    players[socket.id].x = movementData.x;
    players[socket.id].y = movementData.y;

    socket.broadcast.emit("playerMoved", players[socket.id]);
});
```

Ambas as coisas servem para conseguirmos atualizar a posição do jogador sempre que a função *update()* é chamada.

```
this.socket.on("disconnect", function (playerId) {
    self.otherPlayers.getChildren().forEach(function (otherPlayer) {
        if (playerId === otherPlayer.playerId) {
            otherPlayer.destroy();
        }
    });
});
```

```
socket.on("disconnect", function () {
    console.log("a user disconnected", socket.id);
    i--;

    lista.length--;
    //console.log(lista);
    delete players[socket.id];

    io.emit("disconnect", socket.id);
});
```

Quando o *disconnect* é chamdo, pegamos a identificação do jogador e removemos a player desse jogador do jogo. Fazemos isso chamando o método *getChildren()* no nosso grupo *otherPlayers*. O método *getChildren()* retornará um array de todos os objetos de jogo que estão nesse grupo e, a partir daí, usamos o método *forEach()* para fazer um loop nesse array.
Por fim, usamos o método *destroy()* para remover esse objeto do jogo.

4 Conclusão

Em suma este trabalho foi bastante completo e alguma dificuldade em implementar/programar. Tivemos algumas dificuldades em perceber o funcionamento devido dos softwares que escolhemos e das ferramentas no entanto tiramos daqui muito. Foi um trabalho que nos permitiu aprender muito acerca do funcionamento devido de servidores e como operam, também foi muito importante para desenvolvermos a nossa capacidade de desenvolver em Phaser 3 e linguagens de programação associadas a ele.

Usamos também software secundário para realizar tarefas simples do projeto, nomeadamente Tiled para fazer o mapa, Premiere para editar e montar o vídeo de apresentação do projeto.

References