RAFAEL COMPANHONI

1. INTRODUÇÃO

Neste jogo cada jogador controla um avatar que pode disparar contra os outros jogadores sendo o objetivo simplesmente eliminar o maior número de jogadores adversários possível. Quando a barra de energia de um dos jogadores termina, o score do jogador que o atacou é incrementado e o jogador eliminado é 'revivido' em um ponto aleatório do mapa. Além disso, o jogo conta com um chat simples que, por default, envia as mensagens para todos os demais jogadores. É possível enviar mensagens em privado no formato @NOME, MSG onde NOME é o username do jogador a receber a mensagem e MSG é o conteúdo da mensagem. O jogo conta com duas telas, sendo a primeira para o login (utilizado para definir o username) e a tela principal. Pode-se ainda alterar o mapa ao clicar no botão 'Trocar de mapa'. Apenas jogadores no mesmo mapa podem interagir. Logo abaixo da tela de chat está o inventório de cada jogador. A cada tiro que um jogador dispara existe 10% de chance de que uma 'Poção' seja incluída em seu inventório. Ao clicar na Poção o jogador habilita um 'Super Ataque' que, quando executado, faz com que o jogador dê uma série de tiros simultâneamente.

Esta aplicação pode ser executada pela maioria dos navegadores web modernos. Do lado do cliente utiliza as seguintes tecnologias:

- HTML5: Inclui dois elementos <canvas> sendo um utilizado para a renderização do jogo propriamente e outro para o score. Isto foi feito visando otimizar a performance já que não há necessidade de atualizações frequentes dos elementos de interface como o score.

- JavaScript: Utilizado para definir os tratadores/emissores de eventos e classes necessárias.

- CSS: utilizado para definir os estilos dos elementos HTML da página.

O código do servidor é implementado com as seguintes tecnologias:

- Node.js: É uma plataforma de desenvolvimento para aplicações web que interpreta código JavaScript.

- Express.js: Um módulo Node.js que visa facilitar o desenvolvimento de aplicações encapsulando o código que poderia ser feito apenas com a API 'pura' do Node.js. Ao utilizar este módulo obtemos um código mais enxuto e seguro.

A aplicação utiliza a biblioteca **socket.io** para a criação da conexão e troca de pacotes pelo cliente e servidor.

2. SOCKET.IO

O socket.io é uma biblioteca para o desenvolvimento de aplicações web em tempo real. Ele permite que se estabeleça comunicação bidirecional entre clientes e servidores. É dividido em duas partes: a **biblioteca cliente** que é executada pelo browser e a **biblioteca servidor**, implementada como um módulo para Node.js. Ambas as bibliotecas possuem uma API quase idêntica o que deixa o desenvolvimento da aplicação bastante intuitivo.

3. SERVIDOR

Para a incluir o módulo servidor do socket.io basta instalá-lo e incluí-lo com:

```
var io = require('socket.io')(serv, {});
```

Neste caso o módulo recebe como parâmetros o objeto 'serv' que é o servidor web criado previamente e configurações adicionais (não utilizadas neste exemplo). O objeto socket disponibiliza dois métodos principais que são utilizados extensivamente por toda a aplicação servidor:

- **on(NOME_DO_EVENTO, callback)**: Possibilita registrar um callback que será executado sempre que o evento NOME_DO_EVENTO for gerado por um dos clientes (ou por um cliente específico).

- **emit(NOME_DO_EVENTO, callback):** Gera eventos do tipo NOME_DO_EVENTO que possuirão pacotes com as informações enviadas aos clientes.

Pode-se vincular tratadores de eventos (que são gerados pelos múltiplos clientes conectados ao servidor) ou gerar eventos que serão tratados pelos clientes. No trecho abaixo é demonstrado como criar um tratador para o evento ‘connection’ que é gerado sempre que um novo cliente estabelece uma conexão.

```
io.sockets.on('connection', function (socket) {
        // realiza qualquer inicialização necessária para a aplicação

        // define tratadores de eventos adicionais
        socket.on('disconnect', function () {
                ...
        });
})
```

A função utilizada como tratador do evento ‘connection’ recebe como parâmetro o objeto socket que representa a conexão com o respectivo cliente. Com este objeto pode-se definir tratadores de evento específicos (por cliente) como por exemplo, para o evento ‘disconnect’ que é gerado pelo cliente quando a conexão é encerrada (o que pode ocorrer quando a janela do navegador é fechada).

3.1 GAME LOOP

O trecho abaixo representa o núcleo da parte do servidor do jogo e é chamada de ‘game loop’.

```
setInterval(function () {
        var packs = Entity.getFrameUpdateData();

        for (var i in SOCKET_LIST) {
                var socket = SOCKET_LIST[i];
                socket.emit('init', packs.initPack);
                socket.emit('update', packs.updatePack);
                socket.emit('remove', packs.removePack);
        }
}, 1000 / 25);
```

Consiste na definição do método setInterval(callback, time) que executa o método callback contínuamente a cada 40 milisegundos. Este método gera 3 tipos de eventos que são enviados a todos os clientes (sockets) conectados junto a um conjunto de pacotes para atualização do estado atual do jogo:

- **init**: Um conjunto de pacotes que conterá informações sobre novos jogadores que acabaram de se conectar a aplicação.

- **update**: Informações sobre o estado atual de cada entidade do jogo. É utilizado pelos clientes para atualizar seus elementos canvas.

- **remove**: Utilizado para manter os clientes atualizados em relação ao término da conexão de outros jogadores.

3.2 CLASSES

Estas são as classes que presentam as diversas entidades do jogo:

- **Player**: Representa um jogador e possui informações como username, hp (health points), score e etc. Cada instância de Player possui os métodos update, getInitPack e getUpdatePack que são gerados para a atualização do jogo e enviados nos eventos previamente discutidos no game loop.

- **Bullet**: Representa um 'tiro' dado por um jogador e possui informações como ângulo e velocidade. Assim como as instâncias de Player também é responsável por gerar as informações de inicialização, atualização e remoção enviados pelo game loop.

- **Entity**: Classe base que contém dados em comum para qualquer tipo de entidade do jogo (neste caso apenas Players e Bullets) como posição na tela e em qual mapa se encontram. O jogo disponibiliza dois mapas: 'field' (representado pela imagem 'map.png') e 'forest' (representado pela imagem 'map2.png').

As classes a seguir fazem parte da interface do jogo e são relacionadas aos itens e jogadas especiais:

- **Inventory**: Representa o conjunto de itens que cada jogador possui. É a única classe JavaScript compartilhada pela parte cliente e servidor ao mesmo tempo. Para distinguir qual parte da aplicação está utilizando é passado em seu construtor o parâmetro booleano *server*. Para exemplificar seu uso o trecho abaixo mostra como o método refreshRender se comporta quando for utilizado pelo servidor (emite lista de itens atualizada) e quando for utilizado pelo cliente (atualiza HTML)

```
self.refreshRender = function() {

    // quando for o servidor envia lista de itens atualizada
    if(self.server) {
        self.socket.emit('updateInventory',self.items);
            return;
    }

    // quando for o cliente atualiza HTML
    var inventory = document.getElementById("inventory");
     inventory.innerHTML = "";
     ...

}
```

- **Item**: Um item do inventário do jogador, possui as propriedades id, name e evento. A propriedade evento é na realidade uma função que 'aplica' o item no jogador. No caso de um item 'Poção' por exemplo, ele restaura o hp do jogador para 10.

```
Item("potion","Poção",function(player){
        player.hp = 10;
    });
```

4. CLIENTE

A aplicação cliente possui um código HTML bem simples e consiste em duas partes principais. Na primeira, identificada pela div “game” estão os elementos HTML5 <canvas> que são utilizados para renderização do jogo e elementos de interface:

```
<div id="game" style="position: absolute; width: 500px; height: 500px">
    <canvas id="ctx" …></canvas>
    <canvas id="ctx-ui"  …></canvas>
    …
 </div>
```

A outra parte é referente ao chat e consiste em uma div identificada por “chat-text” e um formulário onde o usuário pode inserir e enviar novas mensagens:

```
<div id="belowGame" style="margin-top: 520px">
    <div id="chat-text" style="width: 500px; height: 100px; overflow-y: scroll">
      <div>Bem vindo ao jogo! Digite suas mensagens privadas no formato @NOME,MSG</div>
    </div>

    <div id="inventory"></div>

    <form id="chat-form">
      <input id="chat-input" type="text" style="width: 500px"></input>
    </form>
  </div>
```

4.1 TRATADORES DOS EVENTOS DO GAME LOOP DO SERVIDOR

Como foi visto anteriormente a aplicação servidor envia três eventos a cada iteração do game loop: **init**, **update** e **remove**. Estes eventos são tratados no cliente da seguinte forma:

- socket.on(**'init'**, function (data) { ...}): verifica nos dados sendo recebidos se existe algum novo cliente e o instancia

- socket.on(**'update'**, function (data) { ...}): Para cada jogador recebido nos dados do servidor, se houver alguma nova informação ela é utilizada para atualizá-lo. Por exemplo, caso o jogador tenha se deslocado e esteja em uma nova posição:

  ```
  if (pack.x !== undefined)
      player.x = pack.x;
  ```

- socket.on(**'remove'**, function (data) {…}): Remove os jogadores informados nos dados recebidos do servidor.

4.2 GAME LOOP DO CLIENTE

Apenas receber os dados do servidor não seria suficiente para que os jogadores visualizassem as alterações no HTML. Para isso é implementado um game loop também no lado do cliente utilizando o método setInterval. No callback deste método os canvas do jogo e score são atualizados assim como cada jogador e bala.

```
setInterval(function () {
    // se ainda estiver na tela de login não precisa atualizar
    if (!selfId) {
      return;
    }

    // atualiza mapa e score
    ctx.clearRect(0, 0, 500, 500);
    drawMap();
    drawScore();

    // atualiza jogadores
    for (var i in Player.list) {
      Player.list[i].draw();
    }

    // atualiza balas
    for (var i in Bullet.list) {
      Bullet.list[i].draw();
    }
  }, 1000 / 25);
```

4.3 EVENTOS DE MOVIMENTO E ATAQUE

Analogamento aos eventos de atualização emitidos pelo servidor, os clientes também emitem eventos para diversas situações, sendo as mais frequentes relacionadas aos eventos de movimento e ataque. O trecho abaixo ilustra como é registrado o movimento da tecla ‘d’ que faz o jogador se movimentar para a direita.

```
document.onkeydown = function (event) {
    // d
    if (event.keyCode === 68)
      socket.emit('keypress', { input: 'right', state: true })

     …
```

4.4 CHAT

Para a implementação do chat foi utilizado um tratador para o evento ‘addToChat’ que insere a nova mensagem no elemento HTML:

```
socket.on('addToChat', function (data) {
    chatText.innerHTML += '<div>' + data + '</div>';
 });
```

Para o envio de novas mensagens dois tipos de mensagem são criadas. Quando o usuário envia uma mensagem em privado (i.e. no format @NOME,MSG) uma mensagem do tipo 'sendPmToServer' é gerada. Caso contrário uma mensagem do tipo 'sendMsgToServer' será enviada. Abaixo, como isto se dá no lado do cliente e como é tratado no lado do servidor

```
chatForm.onsubmit = function (e) {
    e.preventDefault();

    // mensagem privada
    if (chatInput.value[0] === '@') {
      socket.emit('sendPmToServer', {
        username: chatInput.value.slice(1, chatInput.value.indexOf(',')),
        message: chatInput.value.slice(chatInput.value.indexOf(',') + 1)
      });
    } else {
      socket.emit('sendMsgToServer', chatInput.value);
    }
    chatInput.value = '';
  }
```

EXECUTANDO O JOGO

O jogo está publicado na plataforma gratuita Heroku no seguinte endereço: https://frozen-dusk-69732.herokuapp.com

Para publicá-lo, basta adicionar o projeto Node.js em um repositório Git, e instalar a extensão Heroku CLI, que permite uma fácil integração com a plataforma de hospedagem. Primeiramente cria-se o repositório e adiciona-se o repositório Heroku como um servidor remoto:

```
$ git init
$ heroku git:remote -a NOME_DA_APLICAÇÃO
```

Em seguida basta fazer o commit das modificações e o push para o repositório remoto:

```
$ git add .
$ git commit -am "Descrição das modificações"
$ git push heroku master
```

Localmente o jogo pode ser executado com o seguinte comando (estando dentro do diretório da aplicação). Quando a porta não for informada será utilizado como default a 2000:

```
$ node app.js << PORTA >>
```

PROTOCOLO WEBSOCKET

É um protocolo da camada de aplicação (referente ao modelo ISO/OSI), orientado à conexão e que faz uso do protocolo TCP na camada de transporte.

Fundamentalmente o protocolo WebSocket reutiliza uma conexão TCP previamente estabelecida entre um cliente e servidor. Após o handshake HTTP o cliente e o servidor começam a troca de pacotes. Esta comunicação é feita em duas vias (full-duplex) e é utilizada pelo servidor para enviar pacotes (push) ao cliente de forma assíncrona. Com o canal de comunicação aberto o cliente define tratadores para o evento de recebimento de pacotes (em JavaScript isto pode ser implementado com métodos callbacks).

A biblioteca socket.io faz uso do protocolo WebSockets mas, de forma transparente, também pode estabelecer a comunicação via Ajax (i.e. com o objeto XHR) fazendo polling no servidor. Para forçar que o cliente sempre utilize WebSockets (o que causará falhas em navegadores mais antigos) utilize no servidor:

```
var io = require('socket.io')(serv, {});
io.set('transports', ['websocket']);
```

E no cliente:

```
var socket = io({transports: ['websocket']});;
```

REFERÊNCIAS E BIBLIOGRAFIA

https://rainingchain.com/tutorial/nodejs
http://rubentd.com/blog/creating-a-multiplayer-game-with-node-js
http://krasimirtsonev.com/blog/article/Real-time-chat-with-NodeJS-Socketio-and-ExpressJS
https://modernweb.com/building-multiplayer-games-with-node-js-and-socket-io/
http://stackabuse.com/node-js-websocket-examples-with-socket-io/
https://socket.io/get-started/chat/
https://tools.ietf.org/html/rfc6455

Powers, Shelley. Learning Node : moving to the server side. Sebastopol, CA: O'Reilly Media, 2016. Print.