VI - First Person Shoter em Three.js

Tiago Almeida Nº76366 Yuriy Muryn Nº76373

***Abstract* –This document appears within the scope of subject of Visualization of Information, in the University of Aveiro, taught by professors Paulo Dias and Beatriz Sousa Santos. This document presents the resolution of the first practical work, which consists in the creation of an application in WebGL / three.js. More specifically, a First Person Shooter game was developed.**

***Resumo* –Este documento surge no âmbito da unidade curricular de Visualização de Informação, na Universidade de Aveiro, lecionada pelos professores Paulo Dias e Beatriz Sousa Santos. Neste documento está apresentada a resolução do primeiro trabalho prático, que consiste na criação de uma aplicação em WebGL/three.js [1]. Mais concretamente, foi desenvolvido um jogo estilo *First Person Shooter*.**

I. Introdução

O tipo de jogo que foi escolhido segue o estilo de um jogo em primeira pessoa (*First Person Shooter*), onde o objetivo é obter o máximo de pontuação dentro de uma arena até acabar a vida do jogador. Os inimigos vão aparecendo aleatoriamente no mapa em intervalos de tempo progressivamente mais curtos e ganha-se pontos a eliminá-los.

Foi implementada ainda a mecânica temporal do jogo SuperHot [2], em que o tempo do jogo só anda quando o jogador efetua alguma ação.

II. Organização do código

Durante a criação deste jogo foi seguida uma abordagem de programação orientada por objetos, onde separamos cada entidade por ficheiros diferentes, para assim tornar este projeto mais escalável e modular. Cada entidade tem uma função *update*, responsável por mantê-la atualizada a cada frame do jogo. A entidade principal, responsável pela gestão de todas as outras, é o *Game*.

Visto que é necessário carregar vários modelos 3d para serem usados durante o jogo, foi criada uma entidade *Loader* que é responsável por carregar todos estes modelos em memória para depois serem reutilizados durante o jogo. Desta forma, não é necessário efetuar leituras do disco durante o jogo, o que iria afetar o desempenho do mesmo. O jogo só começa depois do *Loader* acabar todas as suas tarefas.

Todos os ficheiros javascript que estão na pasta threejs não são da nossa autoria.

III. Aplicação desenvolvida

A aplicação final é composta pelas entidades Player, Enemy, Weapon, Bullet e Game. Em seguida vai ser abordada cada uma dessas entidades, explicando o que foi desenvolvido.

A. Game

Entidade responsável pela gestão do jogo. Esta entidade tem conhecimento de todas as outras entidades envolvidas e é responsável por iniciá-las. Durante a inicialização do jogo é criada a cena do threejs, a câmara associada ao PointerLockControls [3] que usa o Pointer Lock API [4], as luzes, o WebGLRenderer, o jogador e o mapa de jogo.

Ao fazer o rendering da cena, esta entidade é também responsável por atualizar (*update*) todas as outras, de modo a elas se poderem mexer a cada frame.

O aparecimento dos inimigos é controlado segundo um contador que define o intervalo de tempo entre o aparecimento destes. Esse contador vai diminuindo a medida que os inimigos vão aparecendo, fazendo com que estes apareçam mais regularmente. Desta forma conseguimos aumentar a dificuldade do jogo a medida que o tempo passa.

Ao eliminar os inimigos, o jogador ganha pontos e o objetivo é obter o máximo de pontos que conseguir. Caso a vida do jogador chegue a 0, é mostrada a janela do fim do jogo com a pontuação obtida. Caso o utilizador queira repetir o jogo, o Game reinicializa algumas das variáveis, sem criar novamente a cena, e o jogo recomeça.

A velocidade do jogo, de modo a obter um comportamento semelhante ao jogo SuperHot [2], também é controlada pelo Game, embora esteja dependente das ações que o Player faz. Se o Player estiver parado, o tempo anda muito devagar, se se movimentar ou disparar, o tempo começa a andar normalmente.

A.1 Mapa de jogo

O mapa de jogo foi inteiramente criado por nós. Para criar o modelo 3d foi utilizado o programa SketchUp [5] e foi exportado em formato OBJ, que é carregado pelo Loader antes do jogo começar.

Para otimizar o desempenho do jogo, chegou-se a conclusão que a melhor solução seria repartir o mapa em 3 partes:

- Estética, que corresponde a aquilo que o jogador vê (Figura 1).
- Superfícies horizontais com quais o jogador pode colidir, que correspondem ao chão e que chama-

mos de *floors* (Figura 3).
- Superfícies verticais com quais o jogador pode colidir, que correspondem às paredes e que chamamos de *walls* (Figura 4).

Ao separar este mapa desta forma, quando estamos a detetar colisões com o chão, em vez de comparar com todas as faces que possam existir no mapa, compara-se apenas com aquelas que sabemos que corresponde ao chão. O mesmo acontece com as colisões com as paredes.

As faces desses dois objetos são transparentes, sendo que a única coisa que o utilizador vê é a parte estética. Nesta, foi feito um mapeamento de materiais de modo a criar a ilusão da existência de sombras no jogo (Figura 2), visto que o rendering de sombras em tempo real tornava o jogo demasiado lento.

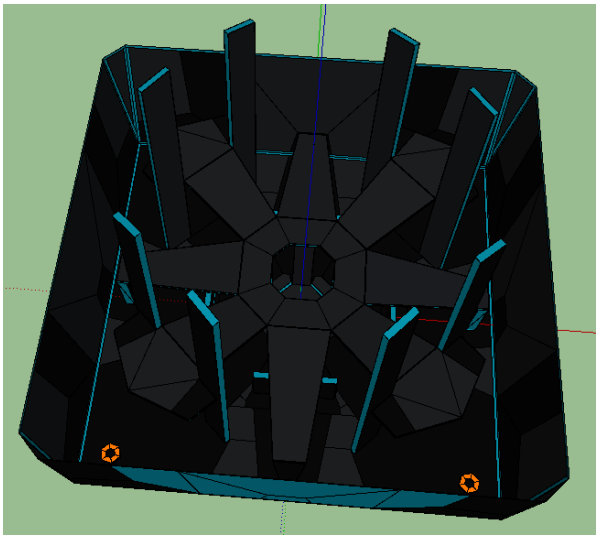

Fig. 1 - Parte estética do mapa criado (sem o teto)

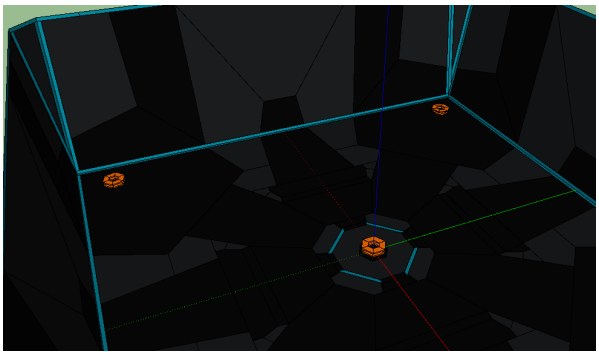

Fig. 2 - Simulação das sombras do mapa criado

B. Player

Entidade que representa a personagem do jogo que o utilizador controla. Esta possui a capacidade de deslocamento pelo cenário e de salto. Possui também duas armas, uma pistola e uma arma de disparo automático.

Em termos de físicas, são verificadas colisões com o chão, com as paredes do cenário e com as balas disparadas pelos inimigos.

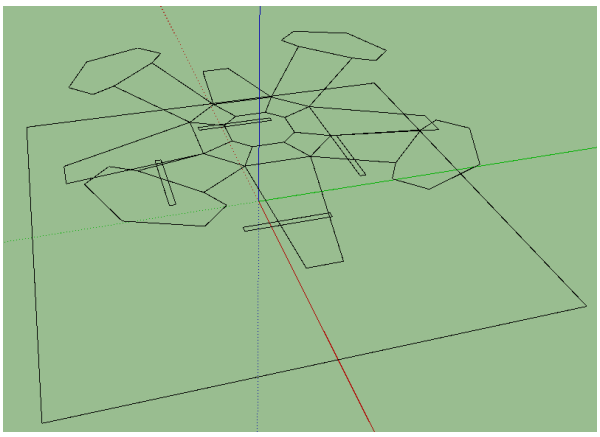

Fig. 3 - Mapeamento das superfícies horizontais (floors)

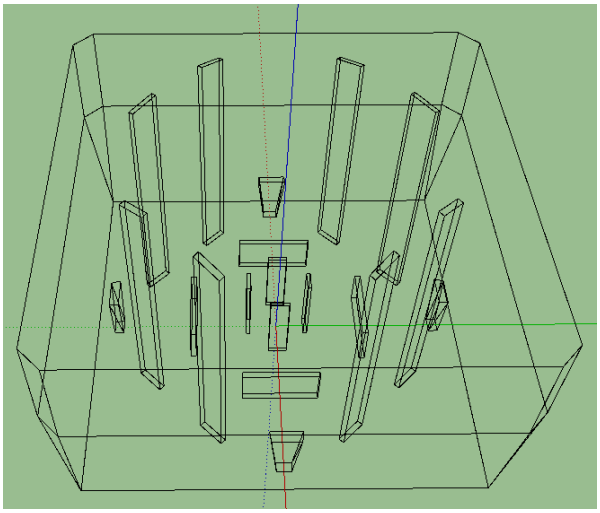

Fig. 4 - Mapeamento das superfícies verticais (walls)

B.1 Movimentos

A movimentação do Player é feita através do input do teclado, mais precisamente as teclas W,A,S,D que representam deslocamento para frente, para a esquerda, para a trás e para a direita, respetivamente. Estas direções são sempre respetivas à direção da câmara. Ainda existe o mapeamento da tecla SPACE que permite efetuar um salto.

O deslocamento é aplicado de forma instantânea. Foi ainda testado aplicar deslocamento de forma gradual mas essa opção foi descartada, este assunto é abordado na secção *Outras soluções testadas*.

Sempre que haja algum movimento por parte do Player, o tempo do jogo começa a andar normalmente. Nos outros casos é como se estivesse parado, andando muito mais devagar.

B.2 Física

Para simular a força gravítica, a cada frame do jogo é decrementada a variável de velocidade vertical de acordo com uma constante gravítica e a massa do jogador. Desta forma, o jogador fica sempre a cair, a não ser que tenha saltado anteriormente, sendo necessário verificar quando colide com o chão.

Para a colisão com o chão é utilizado o Raycast, onde direcionamos um raio com o ponto de inicio a cabeça do Player (posição da câmara), e a direção o vetor $\overrightarrow{v} = (0, -1, 0)$, isto é, para baixo, de comprimento o tamanho do Player. Caso exista interseção com o objeto *floors* é porque estamos a colidir com o chão e neste caso coloca-se a velocidade vertical do Player a 0 para ele deixar de cair, e repõe-se a posição y do jogador para corresponder a altura do mesmo. Esta abordagem, em geral, tem um bom desempenho mas é de notar que se o jogo estiver a renderizar muito poucos frames por segundo, pode acontecer que o Player num frame esteja por cima do chão e no frame seguinte já tenha passado completamente o chão. Neste caso, se o jogador sair do mapa, simplesmente o repomos para uma posição dentro do mapa de jogo.

As colisões com as paredes seguem a mesma lógica que com o chão. Também são feitas por Raycast, onde a direção do raio é a direção do deslocamento do Player e se existir interceção com o objeto *walls*, significa que o ele colidiu com uma parede e não se pode deslocar nessa direção. A deteção de colisões com as paredes idealmente deveria ser feita por *boundingbox*, na secção *Outras soluções testadas* é apresenta uma justificação para esta abordagem.

Para detetar as balas disparadas contra o Player utilizamos uma *boundingbox*, visto que o Player não possui *mesh*.

B.3 Armas

Existe o mapeamento do input *click* do rato para efetuar o disparo das armas. Embora a implementação da mecânica de disparo pertencer à entidade arma, o jogador é que indica à arma a intenção de disparar.

Existe mapeamento da tecla R que recarrega a arma e as teclas 1 e 2 para escolher, respetivamente, uma pistola ou uma arma automática. O modelo da arma que não for escolhida tem de ser escondido da cena.

C. Enemy

Esta é a entidade que o Player têm de eliminar de maneira a pontuar. Graficamente corresponde a um modelo de 3 dimensões com animações, esta possui físicas e colisão com o chão, têm a capacidade de se deslocar pelo mapa e de disparar balas contra o Player, sendo estas duas ultimas ações controladas pela inteligência artificial do inimigo.

Visto que o inimigo é uma entidade que está constantemente a aparecer e a desaparecer do cenário foi criada uma *pool* de inimigos de modo a poder reutilizar os inimigos já instanciados e assim evitar perdas de desempenho no que toda a instanciação e remoção deste tipo de objetos.

C.1 Modelo 3D e animações

Tanto o modelo 3D, as animações e o código para fazer a importação para three.js não é da nossa autoria, facto salientado na secção *Utilização código de terceiros*. No entanto foi necessário efetuar a sincronização da animação a quando do disparo, sendo que o disparo é efetuado $250ms$ depois da animação começar e ainda existe mais $250ms$ de espera depois do disparo que corresponde ao tempo da animação terminar. Em termos de controlo do movimento do inimigo também foram alteradas as velocidades por defeito que o modelo trazia.

C.2 Física

O inimigo também possui as mesmas mecânicas de simulação de física e de colisão com o chão implementadas no Player. Não foram implementadas colisões contra paredes ou contra outros inimigos visto que, se cada inimigo fizesse Raycast horizontal para encontrar obstáculos, o jogo ficava mais lento. Uma possível implementação seria ao nível da inteligência artificial ter o mapeamento do mapa de jogo e assim através de um algoritmo de *pathfinding* garantíamos que o inimigo nunca se ia deslocar contra paredes.

C.3 Inteligência artificial

A inteligência artificial implementada é muito simplista e ingénua. Esta consiste em de 3 em 3 segundos efetuar um disparo e em estar a andar sempre em frente e de x em x segundos rodar na direção do Player, sendo $x = 1 + random$.

Podia-se ter implementada a inteligência artificial mais complexa mas achamos que para o âmbito deste trabalho não havia necessidade disso.

C.4 Mecânica de disparo

O Enemy, ao contrário do Player, não utiliza a entidade arma para efetuar o disparo. Neste caso as balas são instanciadas diretamente no modelo do inimigo pois este já possui uma arma no modelo. Durante a instanciação a direção de movimento das balas corresponde ao vetor de direção do inimigo no momento do disparo ou caso o Player esteja em frente do inimigo corresponde ao vetor em direção ao jogador, isto é,

$$\overrightarrow{direcao_{bala}} = Posicao_{player} - Posicao_{bala}$$

D. Weapon

Esta entidade simula o comportamento de uma arma, tendo em base as suas caraterísticas como a quantidade de balas, a velocidade de disparo, o dano, a velocidade das balas, a precisão, o tempo que demora a recarregar, entre outras.

D.1 Funcionamento por estados

O funcionamento base das armas consiste numa máquina de estados. No estado normal, por exemplo, rotação x é 0, a mira é branca, a arma está pronta a disparar, etc. No estado "sem balas", a mira já é vermelha para dar um feedback visual, e no *update* da arma não acontece nada. No estado "disparou", é feita uma ligeira rotação do eixo x da arma, de modo a dar uma ideia de recuo desta. Enquanto a arma está neste estado, é feita uma contagem até à próxima vez que

a arma pode disparar, que depende da velocidade do disparo.

Esta abordagem facilitou o feedback visual tanto da própria arma, como da mira e HUD (*heads-up display*) da arma selecionada e as balas restantes, e permitiu que o método *update* se tornasse mais eficiente, visto que só se atualizava aquilo que realmente era necessário atualizar.

D.2 HUD da arma

Para mostrar que arma é que está selecionada e quantas balas ainda tem, usou-se diretamente o HTML e CSS, visto que era mais simples (Figura 5 e 6).

Fig. 5 - Representação da HUD da pistola

Fig. 6 - Representação da HUD da arma automática

D.3 Modelos 3d

Os modelo usados para as armas foram criadas por nós, usando o programa SketchUp [5]. Estes modelos foram baseados nos outros modelos já existentes, retirados do "3D Warehouse"deste programa, mas foram feitos de modo a reduzir o número de faces a usar e para adaptar ao esquema de cores que usamos no jogo (Figuras 7 e 8).

Estes modelos foram exportados em formato OBJ que posteriormente são carregados pelo Loader.

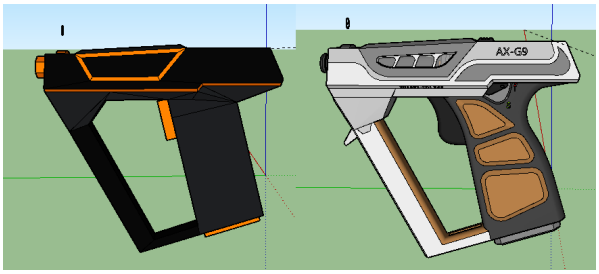

Fig. 7 - Modelo da pistola criado (esquerda), comparado com o modelo original (direita)

D.4 Mecânica de accuracy e spread

Tendo em conta que as balas são instanciadas no cano da arma e estas movem-se em frente e que a mira encontra-se no centro da tela, estas nunca iriam coincidir com a posição onde o utilizador aponta com a mira.

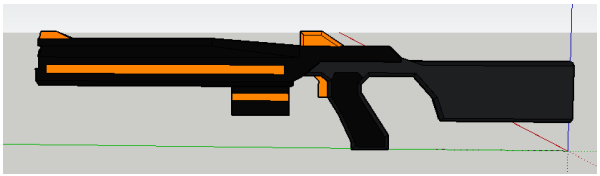

Fig. 8 - Modelo da arma automática criado

Para corrigir este problema, é utiliza a caraterística da arma accuracy que diz a distancia da arma onde esta irá ter 100% de precisão. A bala disparada segue a direção do vetor onde o ponto inicial é o cano da arma e o ponto final corresponde ao ponto no centro da tela deslocado com o valor da distancia de accuracy para a frente.

De modo a dar alguma aleatoriedade ao jogo e para diferenciar os diferentes tipos de armas, foi acrescentada uma caraterística chamada spread, que faz um deslocamento aleatório do ponto para onde se dispara. Na pistola este spread é quase inexistente, fazendo com que a precisão seja muito elevada. Por outro lado, visto que a arma automática consegue disparar várias balas por segundo, para compensar acrescentou-se um spread elevado, fazendo com que a precisão fosse mais baixa.

E. Bala

A bala que é disparada é composta por SphereGeometry, que corresponde à bala em si, e por um CylinderGeometry, que corresponde ao rasto deixado pela bala. A este rasto tem de se mudar o *quaternion* para coincidir com a direção da deslocação da bala. A bala possui colisões com o cenário, Player e inimigos. À semelhança da entidade *Enemy*, também foi criada uma pool de balas, visto que estas estavam a ser criadas e destruídas muitas vezes e isso afetava o desempenho do jogo.

E.1 Colisões

As colisões com o cenário e inimigos são detetadas utilizando o Raycast, onde o ponto inicial é ligeiramente atrás da esfera e a direção é a direção de deslocamento da bala, caso exista interceção com o cenário a bala é destruída, caso exista interceção com o inimigo a vida destes é decrementada de acordo com o dano da bala. De notar que antes de verificar a interceção do Raycast da bala com o inimigo é verifica a distancias destes, esta verificação tem por objetivo melhorar o desempenho computacional visto que a verificação de interseções do Raycast com o modelo 3D do inimigo é um processo mais demorado. Para a colisão com o Player é utilizado a sua *boundingbox*, para verificar se existe interceção entre as coordenadas da bala e as da *boundingbox*.

F. Boost

Esta entidade possibilita ao Player e ao Enemy efetuar um salto de elevada intensidade, em termos comparativos o boost tem uma intensidade aproximadamente

3 vezes superior ao salto normal do Player. Tanto o Player como o Enemy possuem a lista de boost que estão no cenário (os objetos a laranja na Figura 2). O boost é acionado quando o Player ou o Enemy passa por cima deste, para esta verificação é calculada a distancia entre a posição do boost e a da entidade ao nível (altura) do boost.

IV. Utilização código de terceiros

Nesta secção pretendemos referenciar e creditar o código/modelos que não fomos nós que fizemos.

- Modelo 3D do inimigo [6] e o código que permite manipular as animações de movimento e duplicalo [7]. Retirado do exemplo *md2/complex* [8] do *three.js*.
- Código para carregar modelos 3D [9].
- Exemplo *controls/pointerlock* [10] do *three.js*, onde foi utilizado o script de *pointlock* [3]

V. Outras soluções testadas

Na seguinte secção serão enunciadas soluções alternativas que foram testadas mas acabaram por não ser utilizadas.

A. Raycast em vez de BoundingBox nas colisões

Como já foi referido as colisões das balas com o inimigo são detetadas via Raycast. Nós testamos a opção de colocar uma *BoundingBox* a englobar o modelo do inimigo, mas esta solução acarreta problemas. O primeiro está relacionado ao nível da precisão da colisão, na Figura 9 é possível ver que existe muito espaço dito vazio à volta do modelo que seria considerado como colisão. O segundo problema desta estratégia está relacionado com o facto da *BoundingBox* não escalar bem com as animações do inimigo, além desta ter de ser atualizada a cada *frame* para conseguir acompanhar o modelo do inimigo nos seus movimentos. Sendo a única vantagem de usar *BoundingBox* o facto de ser computacionalmente mais rápido efetuar o calculo da interceção com outros objetos. Acrescentando ainda que o Raycast como é calculado ao nível da *mesh* dos modelos, apresenta um precisão superior. Esta experiência também permitiu concluir que as *BoundingBox* devem ser utilizadas em objetos estáticos que se encontrem no cenário.

Sendo assim, seria de esperar que se tivesse utilizado *BoundingBox* na deteção de colisões entre player e as paredes, que seria a abordagem correta, mas visto que o mapa (Figura 1) foi criado por nós e apresenta um nível de detalhe elevado, seria necessário fazer o mapeamento manual das *BoundingBox* para todas as paredes do cenário e sendo que se trata de um trabalho no âmbito escolar com limite de tempo de entrega, esta opção não era viável.

B. Movimento gradual

Como já foi visto, o movimento do Player encontra-se implementado de maneira instantânea, isto é, não existe uma transição gradual entre o estar parado e estar a andar ou ao contrário. Foi testada uma implementação de movimentação deste género mas por causa da mecânica do tempo que temos presente, trazia alguns problemas durante os saltos e então decidimos retirar essa implementação.

Fig. 9 - A azul *BoundigBox* que engloba o modelo do inimigo

C. Outros modelos utilizados

Antes de criamos os nossos modelos de mapa e armas, foram utilizados modelos encontrados online, mas no caso do mapa chegamos a conclusão que não não se adequavam ao estilo do jogo. E outro problemas que os modelos tinham é que a mesh destes eram composta por muitos triângulos, tornando o modelo mais pesado. Já por outro lado os modelos criados são todos *lowpoly*. Para usar vários modelos foram testados vários tipos de loaders presentes nos exemplos de three.js, dependendo do formato do modelo encontrado. Maioritariamente, os modelos encontrados vinham nos formatos .3ds, .dae (collada) e .obj. Como queriamos armas que ficassem bem com o resto do jogo, decidimos criar os nossos modelos e exportá-los em formato .obj.

D. Sombras e luzes

Uma das primeira ideias para o jogo foi fazer com que as balas fossem a única fonte de luz no jogo. Esta ideia foi rapidamente descartada por questões de desempenho. Com vários inimigos a disparar contra nós, um mapa relativamente complexo e com balas sempre a mexer-se, o jogo era impossível de jogar.

Numa fase posterior, depois de termos criado o mapa para o jogo, experimentamos meter luzes que criassem sombras (*castShadow*). Com única fonte de luz deste tipo e com vários inimigos, o jogo ficava outra vez lento. Para ultrapassar esta limitação, decidimos mapear as sombras diretamente no modelo carregado. Assim, o mapa parece que tem as sombras mas não é necessário estar a fazer rendering destas, melhorando o desempenho.

E. MD2Character

Para o modelo do inimigo foi usado inicialmente o MD2Loader com MD2Character, retirados dos exemplos de three.js. Este modelo já continha as animações mas quando tentamos colocar vários inimigos na mesma cena, deparamo-nos com o problema de ser necessário estar sempre a carregar o modelo para cada inimigo. Ainda tentámos modificar o MD2Character e introduzir uma função de clone para reutilizar o mesmo modelo várias vezes, mas era necessário fazer um *deep clone* de cada variável usada e não era viável. Posteriormente descobrimos que o MD2CharacterComplex (também dos exemplos do three.js) já contém essa função e então passamos a usar este.

F. Diferentes Browsers

Decidimos também testar o jogo nos seguintes browsers, *chrome*, *firefox*, *firefox quantum* e *edge*, para comparar o desempenho. Concluímos que em termos de estabilidade de *fps* e fluidez de jogo o *google chrome* destacava-se dos restantes, sendo que o *firefox quantum* é o browser com maior instabilidade. De notar que para executar o jogo no *google chrome* e é necessário utilizar a seguinte flag *–allow-file-access-from-files*.

Bibliografia

[1] "Three js - javascript 3d library", nov de 2017.
URL: `https://threejs.org/`

[2] "Superhot", nov de 2017.
URL: `https://superhotgame.com/`

[3] "Three.js examples - pointerlock", nov de 2017.
URL: `https://github.com/mrdoob/three.js/blob/master/examples/misc_controls_pointerlock.html`

[4] "Pointer lock api", nov de 2017.
URL: `https://developer.mozilla.org/en-US/docs/Web/API/Pointer_Lock_API`

[5] "Sketchup", nov de 2017.
URL: `https://www.sketchup.com/`

[6] "3d model ogro, three.js examples", nov de 2017.
URL: `https://github.com/mrdoob/three.js/tree/master/examples/models/md2/ogro`

[7] "Source code md2 complex", nov de 2017.
URL: `https://github.com/mrdoob/three.js/blob/master/examples/webgl_loader_md2_control.html`

[8] "Md2 complex example", nov de 2017.
URL: `https://threejs.org/examples/?q=md2#webgl_loader_md2_control`

[9] "Three.js loaders", nov de 2017.
URL: `https://github.com/mrdoob/three.js/tree/master/examples/js/loaders`

[10] "Controls pointlock example", nov de 2017.
URL: `https://threejs.org/examples/?q=Pointe#misc_controls_pointerlock`