GL*Bert - A Q*Bert WebGL Implementation

Jorge Ricardo Pinto de Figueiredo Catarino, Óscar Xavier Oliveira Pimentel

Resumo **- Este relatório descreve o processo de implementação do jogo Q*bert em WebGL e Javascript. O relatório está dividido em vários capítulos, começando por um de introdução ao jogo, meramente explicativo, seguindo-se a descrição dos requisitos para o projeto, e um capítulo de detalhes técnicos sobre as abordagens e decisões tomadas ao longo do seu desenvolvimento. Por fim, é dedicado um capítulo à interação do utilizador com o jogo desenvolvido, bem como os pontos a melhorar no futuro.**

Abstract **- This report describes the process to recreate the classic arcade game Q*bert using WebGL and Javascript. It's divided into several chapters, starting with a simple introduction to the original game, followed by the requirements of the WebGL implementation. Afterwards, we go through the technical details of the solution. To conclude, there's a chapter dedicated to the interactivity of the developed game and possible future improvements.**

Fig. 1 - Q*bert original.

I. INTRODUÇÃO

Neste projeto foi feita uma implementação 3D, em WebGL, do clássico jogo de arcade, Q*Bert (fig. 1).

*A. Introdução ao Q*Bert*

Q*bert é um jogo de arcade lançado pela Gottlieb em 1982 [1]. Constituiu um grande sucesso nos anos 80 e destacava-se por ser um jogo 2D que usava gráficos isométricos para criar um efeito pseudo-3D.

Neste jogo, o objetivo principal é o jogador mudar a cor de todos os blocos de uma pirâmide para uma dada cor alvo. Para tal, o jogador deve fazer o personagem titular andar de cubo em cubo, mudando as cores e evitando os inimigos. Tanto o personagem titular como os inimigos apenas se podem mover na diagonal. A arena não tem barreiras nas extremidades, pelo que é possível cair para fora desta e morrer.

Os elementos principais constituintes deste jogo são uma pirâmide de cubos, que corresponde a arena de jogo, o personagem titular Q*bert, os inimigos e discos de transporte de uso único que levam o jogador de volta ao topo da pirâmide.

II. REQUISITOS DA IMPLEMENTAÇÃO

Para simular o que seria a experiência do jogo original definimos um conjunto de requisitos, que serviram de etapas à sua implementação:

- Visualização de uma arena em forma de pirâmide, composta por blocos que mudam de cor com o toque;
- Visualização e movimento da personagem jogável;
- Visualização e movimento de um conjunto de inimigos;
- Visualização e funcionamento dos discos transportadores;
- Verificação das colisões entre o jogador e os obstáculos;
- Sons dos vários elementos do jogo;
- Verificação de quedas da arena;
- Condições de vitória e de fim de jogo.

Em acréscimo, foi tido em conta um outro grupo de requisitos mais direcionados a explorar as capacidades da aplicação 3D e do WebGL:

- Permitir várias perspectivas do jogo;
- Implementação de um modelo de iluminação;
- Permitir o ajuste da iluminação.

III. ABORDAGEM TÉCNICA

O jogo é composto por vários elementos gráficos, e dedicamos esta secção à explicação de como foi abordado a criação dos elementos no contexto WebGL. As classes correspondentes aos elementos que compõem o jogo, como o mapa, o Q*Bert e os inimigos, estão presentes no ficheiro *gameEntities.js*.

No ficheiro *animation.js* encontram-se as funções que calculam a animação do movimento para cada entidade móvel.

O ficheiro *utils/events.js* contém as funções relacionadas com os eventos de input do utilizador:

- movimento da personagem titular - teclas Up, Down, Left, Right;
- troca de perspetiva ou reset - clique do rato no botão respetivo;
- variação de iluminação - clique do rato nas barras deslizantes;

No ficheiro *gameMenus.js* encontram-se o conjunto de funções que correspondem aos vários menus do jogo, como o inicial, derrota e vitória.

O ficheiro *sounds.js* contém a função que permite tocar um som, bem como a lista de sons que o jogo utiliza.

Já o *GLBert.js* é o núcleo do jogo, onde são instanciadas todas as entidades correspondentes ao jogo e onde este é efetivamente posto a correr.

Outras funções de utilidade podem ser encontradas na pasta *utils/*.

A. Arena de jogo - criação da pirâmide

A base do jogo Q*bert é uma arena composta por cubos, em forma de pirâmide, que têm que ser trocados de cor. O nível que escolhemos implementar é composto por 28 cubos, dispostos por 7 linhas, que começam com cor azul e têm que ser alterados para amarelo.

O primeiro passo da abordagem à construção da pirâmide foi criar um modelo de cubo, que constitui o elemento básico do mapa. Para tal, foi desenvolvida a classe *MapPiece*, representativa deste elemento. A *MapPiece* instancia um cubo, contendo os vértices, as cores, sendo depois fácil alterar esses parâmetros, tal como a lógica para alteração de cor. Esta separação entre peças de mapa e o mapa é vantajosa, pois permite facilmente criar mapas em formatos diferentes.

Para dispor os cubos em forma de pirâmide foi necessário instanciar o modelo 28 vezes. Foi desenvolvida a classe Map, que contém um algoritmo que constrói a pirâmide de forma intuitiva, linha a linha. Foi criado um array de *MapPieces*, e este é preenchido com base numa propriedade da pirâmide em questão: a linha X da pirâmide contém X cubos. O excerto de código que se segue mostra criação de uma linha da pirâmide.

```
for(var i = 0; i<7; i++){
      this.mapPieces.push(new MapPiece(coordx,
      -0.75, -0.75, 7, i+1, i));
      this.rowCol[6].push([i+1,this.mapPieces[
      this.mapPieces.length-1]]);
      coordx = coordx + magicNumber;
}
```

Esta classe revela-se bastante útil ao longo do projeto, pois tem uma função que nos permite aceder a uma peça individual do mapa, getPiece(), de modo a que sejam feitas as alterações necessárias.

A1. Arena de jogo - lógica de colisões e posições

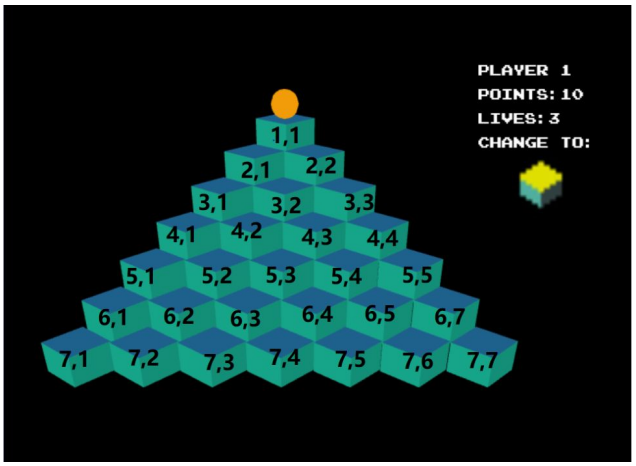

Fig. 2 - Numeração dos cubos na arena.

A implementação deste jogo implica o confronto com várias situações que advêm de vários objectos moverem-se em simultâneo no mapa. Cada elemento possui coordenadas que o marcam no espaço. Contudo, resolver problemas como colisões com inimigos ou a morte do personagem titular nos limites do mapa com base nas comparações de coordenadas dos elementos revelou-se inviável, devido ao cálculo meticuloso de posições que envolve muitas casas decimais, e uma pequena diferença numa milésima cancela a verificação para uma possível colisão.

Então, uma abordagem diferente surgiu, baseada num tabuleiro de xadrez (fig. 2). Cada cubo é representado por um par (linha, coluna). Esta representação é perfeita, pois identifica cada cubo de forma única e inequívoca. Desta forma, foram resolvidos vários problemas:

- A colisão com um inimigo (e perda de 1 vida do Q*bert) é feita ao verificar se o par (linha, coluna) do cubo para onde o Q*bert ou o inimigo se movem já possui alguém. Caso aconteça, o Q*bert volta à posição do topo e perde 1 vida;
- A troca de cores de um cubo ao toque do Q*bert foi também solucionada com esta notação. Na função *animateQbert()* é tratado este desafio com base no par (linha, coluna) que o Q*bert está atualmente. E é só trocar a cor desse cubo.
- O Q*bert e os inimigos não se podem mover para fora da pirâmide, naturalmente. Se for tentado um movimento nesse sentido, o Q*bert perde 1 vida ou, no caso do inimigo, ele reaparece. Com esta notação, as verificações de limites do mapa tornaram-se simples com umas comparações de linhas e colunas;

- O acesso do Q*bert aos discos que se encontram fora dos cubos mapa tornou-se simples com uma permissão de movimento lateral (quebrando o limite do mapa) caso o disco esteja ativo;

Esta notação permite saber a posição exata de qualquer elemento através do seu par (linha, coluna) que é único em cada momento.
Em adição a esta notação, os cubos foram também numerados de 1 a 28, de modo a facilitar o estabelecimento de cubos específicos para um inimigo reaparecer ou para discos existirem, com acesso simples ao array de peças através da função *getMapPieces()*. Para este efeito, um número identificativo único é o suficiente.

B. Personagem titular

A personagem titular do Q*bert é um boneco laranja que viaja de casa em casa, movendo-se nas diagonais.
O modelo escolhido para representar o Q*bert foi uma esfera, construída a partir da deslocação dos vértices de uma subdivisão recursiva por triângulos de um cubo para uma superfície esférica.
Para representar esta entidade, foi criada a classe *Qbert*, que implementa todos os atributos associados à personagem titular: o número de vidas, a pontuação do jogador, os 4 movimentos possíveis e a lógica associada a esses atributos.
Cada um dos 4 movimentos possíveis nas diagonais tem a sua função associada que, além de executar o movimento, verifica a legalidade do mesmo e se é feito para uma posição de morte.
O código que se segue ilustra como é feito o movimento descendente para a esquerda do Q*bert. É evidente a simplicidade que a notação de pares (linha, coluna) proporcionou a este processo para lidar com os movimentos.

```
moveLeftDown(){
        var rowTemp = this.row + 1;
        var collumnTemp = this.collumn;

        if(rowTemp<=7){
                            this.finalPosx   -=
(this.magicNumber/2);
         this.finalPosy -= (2.39*0.075);
                        this.direction       =
vec3(this.finalPosx-this.tx,
this.finalPosy-this.ty, 0);
         normalize(this.direction);
         this.row = rowTemp;
         this.collumn = collumnTemp;
         this.isMoving = true;

        }else{
            this.isMoving = false;
            qbert.isDead();
            fallSound.play();
        }

}
```

Esta classe permite manipular tudo o que for relacionado com o personagem Q*bert.
No ficheiro *animation.js* encontra-se a função *animateQbert()*, que através da posição atual do Q*bert, da posição final, da velocidade e da direção realiza a animação dos movimentos do Q*bert entre cubos, para os discos de transporte e da lógica dos cubos pisados. Durante esta animação, o jogador é impedido de fazer mais movimentos.
A função *handleKeys()* do ficheiro *events.js* processa os movimentos do Q*bert com base na tecla pressionada, invocando a função de movimento respetiva da classe *Qbert*. Foi adicionada lógica adicional ao evento para garantir que só existe um movimento mesmo se o jogador manter a tecla premida.

C. Inimigos

Analogamente ao personagem titular, foi desenvolvida uma classe *Enemy*, para representar os inimigos dos quais o Q*bert tem que evitar de modo a completar o jogo com sucesso.
O modelo escolhido para representar um inimigo foi um tetraedro, de cor roxa. Decidimos que a representação do jogo teria 2 inimigos, portanto 2 instâncias deste tetraedro.
Os inimigos aparecem no topo da pirâmide e vão descendo, tendo apenas 2 movimentos possíveis: diagonal esquerda e diagonal direita. A probabilidade de ir para a esquerda ou direita é 50% para cada uma. Ou seja, o inimigo age sozinho, descendo a pirâmide com movimento pseudo-aleatório, atrapalhando o jogador. Quando um inimigo atinge a base da pirâmide, reaparece no topo, aleatoriamente, num de 4 cubos possíveis: 22, 24, 25 e 26. Isto adiciona uma certa imprevisibilidade ao comportamento de um inimigo, tornando-o mais "autónomo" e constitui uma dificuldade acrescida ao jogo.
O inimigo, tendo 2 movimentos possíveis, tem 2 funções associadas que permitem o movimento descendente na diagonal esquerda e direita, feitas de forma análoga às do Q*bert.
No ficheiro *animation.js*, a função *animateEnemy()* trata da animação dos movimentos de cada inimigo, de forma análoga ao processo feito para o Q*bert, mas dado que o movimento aqui é autónomo, a direção é calculada aqui. Os inimigos rodam consoante o lado para que se vão mover.

D. Discos de transporte

Outro elemento presente no jogo Q*bert é um disco que transporta o jogador para o topo da pirâmide. Existem 2 destes discos, e são de uso único, pelo que devem ser utilizados moderadamente pelo jogador.
O modelo escolhido para representar este elemento foi o Prisma Hexagonal, com as faces laterais cinzentas e as hexagonais às cores, como no jogo original. Para dar o efeito de "disco", foi reduzido o comprimento das faces laterais do mesmo, através do parâmetro sy.
Sendo um elemento do jogo, foi lógica a criação da classe *Disk* para o representar. A construção da classe é análoga às anteriormente discutidas, com a criação dos arrays de vértices e cores, e funções que permitem ativar/desativar movimento dos discos e perceber se estão ativos ou não.

O disco, sendo uma entidade móvel a certo ponto do jogo, tem a sua função *animateDisk()* no ficheiro *animation.js*. Esta função tem a missão de animar tanto o disco como o Q*bert na trajetória ascendente para o topo da pirâmide e deixar o Q*bert no topo.

A animação do disco está sincronizada com o som que é emitido quando este é ativado, sendo então a entidade com maior velocidade do jogo. Durante a animação, a velocidade do Q*bert é temporariamente aumentada de forma a dar a sensação que está a ser transportado pelo disco. Na realidade estas entidades estão a ser animadas de forma independente, em direção a pontos com a mesma coordenada X e Y diferentes.

Quando o disco é usado, é feita a translação deste para um ponto fora da viewport, de forma a que não fique visível para o utilizador.

O jogador dispõe de 2 discos, de uso único, 1 de cada lado da pirâmide, ao lado dos cubos 13 e 17. Portanto, se o utilizador usar um disco e tentar mover na direção onde ele estava, na casa de acesso ao disco, vai naturalmente morrer, pois o disco já foi usado.

IV. EXPERIÊNCIA DO JOGO

Fig. 3 - Jogo desenvolvido.

Quando o utilizador pretende iniciar um jogo (fig.3), é presenteado com o logótipo criado para a nossa versão do jogo e é necessário que pressione a tecla Enter para começar. O objetivo deste nível é mudar a cor de todos os cubos para amarelo, ganhando 25 pontos de cada vez.

O movimento do Q*bert é feito através das teclas Up, Down, Right e Left, movendo-o diagonalmente nessas direções.

Com o objetivo de melhorar a experiência de jogo do utilizador foram introduzidos os sons do jogo original [4] que são audíveis em vários momentos: no início de um jogo, nos movimentos do Q*bert e dos inimigos, na morte do Q*bert, no transporte do disco e na vitória no jogo. Assim, recomenda-se jogar com o som ativo para uma experiência mais imersiva e interativa.

Fig. 4 - Ecrã de Game Over.

No canto superior direito, é possível ver a pontuação atual, o número de vidas e a cor alvo (fig.6). Quando o utilizador perder as 3 vidas, aparece o aviso de Game Over, como é possível observar na figura 5.

Fig. 5 - Ecrã de vitória no jogo.

Quando o utilizador consegue mudar a cor de todos os cubos para amarelo, a mensagem de nível completo aparece, mostrando a pontuação total.

Tanto no ecrã de Game Over como no de nível completo, o utilizador tem a opção de carregar no botão reset para jogar de novo.

Além da experiência do jogo em si, a nossa implementação permite que o utilizador a personalize através de algumas opções que lhe são permitidas através de input.

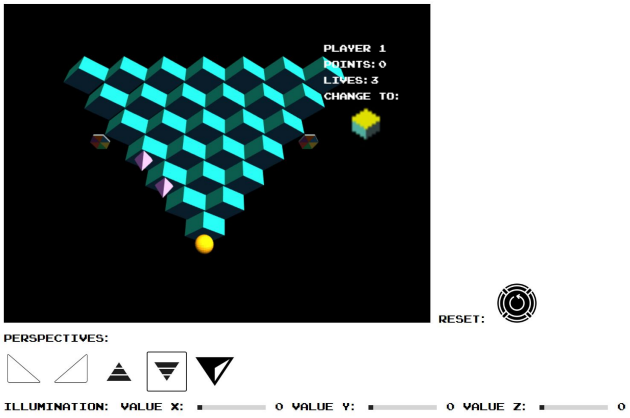

Fig. 6 - Perspetiva invertida do jogo.

Como é visto na figura 6, é permitido ao utilizador mudar a perspectiva do jogo. São permitidos 5 modos de rotação do mapa: rodar para a esquerda, direita, vista normal, invertida e do topo. Mudar de perspetiva torna a experiência de jogo mais difícil, pois os comandos de movimentos, que são os mesmos, têm que ser reajustados na mente do utilizador à nova perspetiva.

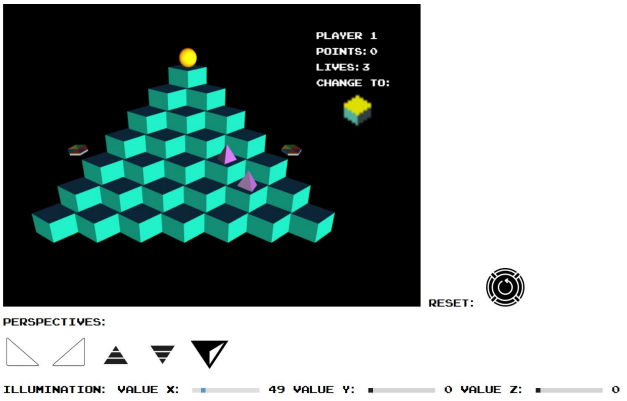

Fig. 7 - Variação da iluminação em X.

Outro parâmetro variável para o utilizador através de input é a iluminação (fig.7). Através de barras deslizantes o utilizador pode ajustar uma nova luz, rodando o foco em X, em Y ou em Z, sendo bem visível a variação do foco aquando das mudanças.

O botão de reset, pode ser utilizado a qualquer momento, e coloca o jogo no seu estado inicial: Q*bert no topo, inimigos nas posições iniciais, perspetiva do mapa normal, pontuação a 0, e vidas a 3.

V. CÓDIGO EXTERNO UTILIZADO

A base do projeto foi o código desenvolvido nas aulas. Os modelos cubo, esfera, tetraedro e prisma hexagonal foram modelos criados nas aulas. O modelo de iluminação direcional foi baseado no que é apresentado no artigo *Directional Lightning*[3] do site WebGL Fundamentals, e ligeiramente alterado para ser possível manter a cor dos vértices de cada objecto. Da mesma fonte foi retirada uma função que calcula a matriz inversa, *inverse()*.

VI. MELHORIAS

- Modelos: Foi desenvolvido um modelo .obj particular do boneco Q*bert em (nome do programa). Contudo, não foi possível utilizá-lo, pois o parser de ficheiros .obj fornecido nas aulas, e que foi utilizado neste projeto, apenas permite o processamento de modelos simples. Como é possível observar [2] é possível fazer um parser elaborado e é necessário caso se queira que o personagem titular e os inimigos tenham modelos mais elaborados que esferas e tetraedros.
- Texturas: Não foram implementadas texturas, pois o foco foi dedicado à visualização dos objetos, iluminação e lógica do jogo. Era possível serem implementadas, mas isso implicava redefinir os shaders e as várias classes que compõem o jogo.
- Mais inimigos: Foi ponderado a implementação de um novo tipo de inimigo, que teria o movimento de seguir o jogador. É implementável criando uma nova classe no *gameEntities.js* e uma função animate para esse inimigo, recorrendo a um algoritmo de pesquisa (por exemplo A*).
- Mais níveis: É possível a implementação de diferentes níveis, com diferentes formas. Basta alterar o algoritmo de geração da pirâmide de cubos para adicionar mais linhas, ou criar ciclos diferentes para criar arenas com formas diferentes.

VII. CONCLUSÕES

Este trabalho constituiu um bom desafio que recorreu à mobilização dos conhecimentos básicos aprendidos nas aulas de WebGL para a criação de algo mais elaborado. A aplicação de modelos 3D, iluminação e movimentos adicionados a uma lógica de jogo criada permitiram a criação de um produto final satisfatório, elegante, e que cumpriu os objetivos propostos.

O trabalho dividiu-se de forma igual pelos 2 membros do grupo, tendo ambos contribuído para o sucesso do projeto. Desta forma, 50% é o valor representativo da contribuição de cada membro.

REFERÊNCIAS

[1] Q*bert (2020, Junho 19). Em Wikipedia https://pt.wikipedia.org/wiki/Q*bert

[2] WebGL Load Ob (2017). Em WebGLFundamentalsj https://webglfundamentals.org/webgl/lessons/webgl-load-obj.html

[3] Directional Lightning (2017). Em WebGLFundamentals https://webglfundamentals.org/webgl/lessons/webgl-3d-lighting-directional.html

[4] http://www.realmofdarkness.net/sb/qbert/