

Jogo de Xadrez implementado em OpenGL

Computação Visual

Universidade de Aveiro

Diogo Silva 60337

Resumo – Este relatório descreve detalhadamente a estrutura e o motor de jogo de Xadrez implementado em OpenGL. Relatório inclui descrição de shaders, modelos, skybox, texturas, iluminação, movimentos, entre outras situações.

I. ESTRUTURA DA APLICAÇÃO

A aplicação do jogo inicialmente encontrava-se em C, tal como foi sugerido nas aulas, mas devido à necessidade de introduzir o conceito de herança nas peças de Xadrez, passou-se tudo para C++ e realizou-se o desenvolvimento a partir daí.

Para compilar o código foi criado um tutorial num ficheiro README com uma explicação breve a dizer as bibliotecas que são necessárias instalar e de como correr o programa.

A. Engine do Jogo de Xadrez

O motor do jogo tem as seguintes funcionalidades como:

1. Obter lista das referências para cada peça
2. Verificar se o jogo já acabou e quem ganhou
3. Verificar de quem é a vez de jogar
4. Realizar um movimento
5. Mostrar movimentos possíveis de uma peça

Tal como mostra a figura seguinte, pode-se verificar que estas funcionalidades são acedidas facilmente através da classe Chess.

```

17 class Chess {
18 private:
19     ChessPiece * table[8][8];
20     Player cPlayer;
21     vector<ChessPiece*> beated;
22     bool gameEnded;
23
24     Point2D<float> determineDeadPosition(ChessPiece*);
25 public:
26     Chess();
27     ~Chess();
28
29     Player getCurrentPlayer();
30     bool isGameFinished();
31     bool isFieldEmpty(Point2D<int>);
32     vector<ChessPiece*> getListPieces();
33     Point2D<int> getBoardPosition(ChessPiece *);
34     Point2D<float> getCompletePosition(ChessPiece *);
35     bool move(ChessPiece* src, Point2D<int> dst);
36     vector<Point2D<int>> getPossiblePositions(ChessPiece *);
37
38     friend ostream &operator<<(ostream &, const Chess &);
39 };

```

Fig. 1

IMPLEMENTAÇÃO DO CHESS

Por sua vez, a classe Chess contém uma matrix de 8 por 8 (64 lugares) em que no início, estão 32 ocupadas e as restantes apontar para NULL, esta matrix é uma matrix de referências de Peças de Xadrez, mais precisamente, objectos do tipo ChessPiece (descritos detalhadamente na secção seguinte).

A.1 Peças do Xadrez

As peças de Xadrez têm um módulo do qual permite distinguir a qual jogador pertence a peça, o tipo da peça (se é um Rei, um Cavalo, etc..). Consequentemente, também é o que permite distinguir movimentos das peças.

Todas as peças herdam uma classe principal chamada ChessPiece, que contém as funcionalidades referidas anteriormente, tal como mostra a figura seguinte:

```

1  #ifndef CHESSPIECE_H
2  #define CHESSPIECE_H
3
4  #include <vector>
5  #include <string>
6  #include "../utils/Points.hpp"
7
8  using namespace std;
9
10 enum Player {ONE, TWO};
11
12 class ChessPiece {
13 private:
14     vector<Point2D<int>> possibleMoves;
15 public:
16     Player player;
17     bool multipleMoves;
18     ChessPiece(Player, vector<Point2D<int>>, bool);
19     ~ChessPiece();
20
21     vector<Point2D<int>> getPossibleMoves();
22     virtual string getType();
23 };
24
25 #endif

```

Fig. 2

IMPLEMENTAÇÃO DO CHESSPIECE

Tal como foi referido, todas as peças têm de indicar na sua inicialização a lista de movimentos possíveis, se são movimentos múltiplos ou não e o jogador a qual pertence, na figura seguinte pode-se ver o exemplo da implementação da Rainha, em que mostra que os movimentos possíveis são em todas as casas as sua volta e são movimentos múltiplos.

```

1 #include "Queen.hpp"
2
3 using namespace std;
4
5 static vector<Point2D<int>> createPossibleMoves() {
6     vector<Point2D<int>> moves;
7     moves.push_back(createPoint(1, 1)); // Diagonal Right
8     moves.push_back(createPoint(1, -1)); // Diagonal Left
9     moves.push_back(createPoint(-1, 1)); // Diagonal Behind Right
10    moves.push_back(createPoint(-1, -1)); // Diagonal Behind Left
11    moves.push_back(createPoint(0, 1)); // Right
12    moves.push_back(createPoint(0, -1)); // Left
13    moves.push_back(createPoint(1, 0)); // Front
14    moves.push_back(createPoint(-1, 0)); // Behind
15    return moves;
16 }
17 Queen::Queen(Player player) : ChessPiece(player, createPossibleMoves(), true) {}
18
19 string Queen::getType() {
20     return "Queen";
21 }

```

Fig. 3
IMPLEMENTAÇÃO DA PEÇA RAINHA

B. Desenvolvimento OpenGL

Relativamente a implementação da parte gráfica da aplicação, decidiu-se dar continuação a estrutura já existentes das aulas, tendo os seguintes ficheiros:

1. `init.cpp`, que inicia todos os modelos, fontes de luz, estruturas, janela e callbacks
2. `models.cpp`, que permite ler os ficheiros obj do formato oficial
3. `globals.cpp`, contém algumas variáveis globais para permitir fácil manipulação durante o uso de callbacks
4. `callbacks.cpp`, este ficheiro é o que trata de re-escrever a imagem, trata também ainda dos movimentos do rato e cliques do teclado
5. `shaders.cpp`, este ficheiro é o que carrega o vertex e o fragment shader

Para além deste ainda foram criados dois ficheiros, um `LightModel.cpp` que contém apenas todas as características necessárias para a representação de um foco de iluminação, tais como, posição do foco, intensidade do foco e intensidade da luz ambiente. O outro ficheiro, é o ficheiro `GraphicModelChess.cpp` que contém as características gráficas de cada modelo de Xadrez (este ficheiro é descrito detalhadamente na secção seguinte).

B.1 Modelos

Estrutura do GraphicModelChess

A classe `GraphicModelChess` contém todas as características gráficas de cada modelo:

1. Número de vértices
2. Lista de vértices
3. Lista das normais
4. Valores de deslocamento, rotação e de redimensionamento
5. Coeficiente Ambiente, Difusão, Especular e Phong

Para além destes valores, ainda contém uma referência para o respectivo modelo de Xadrez, ou seja, objecto `ChessPiece`.

A figura seguinte mostra o cabeçalho deste ficheiro:

```

13 class GraphicModelChess
14 {
15 public:
16     /* Coordenadas dos vertices */
17     GraphicModelChess();
18     ~GraphicModelChess();
19
20     ChessPiece *piece;
21     int numVertices;
22     vector<float> arrayVertices;
23     vector<float> arrayNormais;
24     /* Propriedades do material */
25     float kAmb[4];
26     float kDif[4];
27     float kEsp[4];
28     float coefPhong;
29     /* Parametros das transformacoes */
30     Point3D<float> desl;
31     Point3D<float> anguloRot;
32     Point3D<float> factorEsc;
33
34     template<class T> static Point2D<float> convertChessPos(Point2D<T>
35         point;
36         point.x = pos.x * 0.5 - 1.75;
37         point.y = pos.y * 0.5 - 1.75;
38         return point;
39     }
40     static Point2D<float> convertBackToChessPos(float, float);
41     static GraphicModelChess* generatePreviewSquare(Point2D<float>, flo
42 };
43

```

Fig. 4
CABEÇALHO DO GRAPHICMODELCHES

Ainda se pode verificar a existência de 3 funções estáticas, duas das quais servem para converter coordenadas do tabuleiro de Xadrez para coordenadas do mundo 3D, e vice-versa. A outra função serve para gerar modelos quadrados que vão servir para representar todas as alternativas possíveis de cada movimento.

Load de cada modelo de Xadrez

As peças do Xadrez foram retiradas de modelos já existentes num projecto Open Source [1], desse projecto foram retirados do directório `/trunk/Chess/-Chess/Models`.

Sendo que esses modelos foram todos optimizados usando o programa Blender de maneira a ficarem da forma pretendida, tal como, colocar a base no plano xOy , ou seja, $z = 0$.

Após a recolocação dos modelos, foram todos exportados para ficheiros `.obj` no formato oficial.

Apesar de fazer a leitura do modelo oficial dos ficheiros `obj`, apenas são carregados os vértices, as normais e as faces, deixando de fora os vértices das texturas.

O load genérico dos ficheiros `obj` encontra-se em `/src/models.cpp`.

B.2 Shaders

Inicialmente os shaders apenas estavam a ser usados para a representação das cores, não fazendo qualquer processamento na gráfica, mas houve a necessidade de implementar a iluminação nos shaders devido ao tentar fazer uma animação mais complexa e notar-se que a imagem não era completamente fluida porque o tempo que o CPU demorava a fazer os cálculos das iluminações de cada vértice era superior ao tempo da rotação, verificando-se um delay no movimento.

Sendo assim, havia duas opções, tornar a animação mais rudimentar, ou passar a iluminação para os sha-

ders (fazendo com que a gráfica processa-se a animação).

```

1 #version 120
2
3 attribute vec3 v_coord3d;
4 attribute vec3 v_normal3d;
5 attribute vec2 texcoord;
6
7 uniform mat4x4 matrizProj;
8 uniform mat4x4 matrizModelView;
9 uniform vec4 posicaoFLuz;
10
11 varying vec3 fN;
12 varying vec3 fE;
13 varying vec3 fL;
14
15 varying vec2 f_texcoord;
16
17 void main( void )
18 {
19     vec3 pos = (matrizModelView * vec4(v_coord3d, 1.0)).xyz;
20
21     fN = v_normal3d;
22     fE = -pos;
23     fL = posicaoFLuz.xyz - pos;
24
25     gl_Position = matrizProj * matrizModelView * vec4(v_coord3d, 1.0);
26     f_texcoord = texcoord;
27 }

```

Fig. 5
VERTEX SHADER

Como se pode verificar no vertex shader é feito os calculos do ponto relativamente a um vertice, a sua posição, consoante a matrix projecção e o modelo, para além disso, ainda é calculado algumas fracções que vão ser usadas mais tarde pelo fragment shader, estas precisam de ser calculadas no vertex porque a iluminação depende da posição do vertice e do foco de iluminação.

Ainda se pode verificar que as coordenadas das texturas são passadas para o fragment shader.

```

1 #version 120
2
3 varying vec3 fN;
4 varying vec3 fL;
5 varying vec3 fE;
6 varying vec2 f_texcoord;
7 uniform sampler2D mytexture;
8
9 uniform vec4 ambientTerm, diffuseTerm, specularTerm;
10 uniform mat4x4 matrizModelView;
11 uniform vec4 posicaoFLuz;
12 uniform float coefPhong;
13
14 void main( void )
15 {
16     vec3 E = normalize(fE);
17     vec3 L = normalize(fL);
18
19     vec3 H = normalize(L + E);
20     vec3 N = normalize(matrizModelView * vec4(fN, 0.0)).xyz;
21     vec4 ambient = ambientTerm;
22
23     float Kd = max(dot(L, N), 0.0);
24
25     vec4 diffuse = Kd * diffuseTerm;
26
27     float Ks = pow(max(dot(N, H), 0.0), coefPhong);
28     vec4 specular = Ks * specularTerm;
29
30     // discard the specular highlight if the light's behind the vertex
31     if( dot(L, N) < 0.0 ) {
32         specular = vec4(0.0, 0.0, 0.0, 1.0);
33     }
34
35     vec4 fColor = ambient + diffuse + specular;
36     fColor.a = 1.0;
37
38     gl_FragColor = texture2D(mytexture, f_texcoord) + fColor;
39 }

```

Fig. 6
FRAGMENT SHADER

No fragment shader pode-se verificar que são usadas as variaveis passadas pelo Vertex Shader, neste caso, fE, fN e fL, calculando a respectiva cor do vertice,

dando o efeito de iluminação pretendido.

Para realizar esta parte do código foi utilizado os slides dados na teórica das aulas de Computação Visual no ficheiro PDF “Métodos de Iluminação e Sombreamento (PDF)” [2] nos slides 50 a 53 e ainda foi consultado wikibooks relativamente a OpenGL para conciliar iluminação e texturas [3].

Ainda se pode verificar que as texturas são introduzidas [4] com o texture2D que permite aplicar uma cor RGB a uma coordenada específica, fazendo isto com todas as coordenadas, para além disto, ainda se pode verificar que dá-se igual importância a textura e a iluminação, apesar de não ser o melhor método funciona relativamente bem.

II. ASPECTOS IMPORTANTES DO TRABALHO

Os aspectos importantes deste trabalho vai desde a representação gráfica dos objectos (como peças de xadrez, texturas, skybox, movimentos possíveis de cada peça) e interação com o utilizador (como clique nas peças de xadrez, teclado ou rato, e ainda manipulação da cena, teclado ou rato).

A. Interação com o utilizador

A.1 Clique nas peças do Xadrez (Interação Directa)

Para permitir o clique nas peças do Xadrez através do rato foi necessário pegar nas coordenadas X, Y do clique do rato e converter para o mundo 3D, para além disso, ainda foi necessário calcular a profundidade do pixel respectivo [5].

Para calcular a profundidade do pixel respectivo, ou seja, do eixo Z, usou-se a função de OpenGL glReadPixels na qual se obteve a coordenada Z do respectivo pixel (glReadPixels(x, int(winY), 1, 1, GL_DEPTH_COMPONENT, GL_FLOAT, &winZ)) e depois a partir das 3 coordenadas usou-se a função de OpenGL gluUnProject passando por argumento, as coordenadas 3D do clique, a matriz projecção e a dimensão do viewport, recebendo por referência o valor do ponto 3D nas coordenadas reais, podendo assim verificar onde é que o utilizar carrega, ou seja, em que local do tabuleiro carregou.

```

glGetDoublev(GL_MODELVIEW_MATRIX, modelview);
glGetDoublev(GL_PROJECTION_MATRIX, projection);
glGetIntegerv(GL_VIEWPORT, viewport);

winX = (float)x;
winY = (float)viewport[3] - (float)y;
for (int i = 0; i < 16; i++)
    projection[i] = matrizProj.m[i];

glReadPixels(x, int(winY), 1, 1, GL_DEPTH_COMPONENT, GL_FLOAT, &winZ);
gluUnProject(winX, winY, winZ, modelview, projection, viewport, &posX, &posY, &posZ);
Point2D<float> p = GraphicModelChess::convertBackToChessPos(posX, posY);

```

Fig. 7
CONVERSÃO DE VIEWPORT PARA POSIÇÃO DO MUNDO 3D

Para além do clique das peças também a possível intercalar entre cada peça usando a tecla + ou - Para alterar entre as posições possíveis da peça seleccionada

usa-se a tecla . ou , E ainda a tecla “m” para fazer a peça mover-se para a posição seleccionada.

A.2 Manipulação do cenário

A manipulação do cenário é possível através de rato e teclado, sendo que com o rato é possível visualizar o tabuleiro de qualquer ângulo, enquanto que com o teclado só roda sobre o eixo do Z, ou seja, a volta do tabuleiro.

Para fazer a manipulação do cenário com o rato, foi usada a callback `glutMotionFunc(onDrag)` que permite detectar movimentos contínuos do rato (cliques sem largar o botão), calculando a diferença com a posição anterior do ângulo e fazendo rodar essa diferença.

Também é possível aumentar ou diminuir a cena (efeito de zoom) usando o scroll do rato.

B. Iluminação na Placa Gráfica

Como foi referido anteriormente, na secção da estrutura dos Shaders, houve a necessidade de passar a iluminação do CPU para a placa gráfica devido ao processamento das animações estar muito lento.

Como se pode verificar na imagem seguinte, o foco de iluminação está colocado na posição $x = 0$ e $y = 0$, com uma intensidade média, apenas para provocar uma luz ambiente no centro do tabuleiro, ainda se pode verificar a reflexão da iluminação em cada peça do Xadrez apontar para o centro.



Fig. 8

ILUMINAÇÃO DO TABULEIRO

C. Texturas

Foram aplicadas duas texturas distintas, textura do tabuleiro, que inclui o quadriculado do Xadrez e a respectiva cor. Outra solução que podia ter sido aplicada era criar 64 planos (32 brancos e 32 pretos) a simular o tabuleiro de Xadrez, mas tendo em conta que é mais

realista aplicar uma textura, optou-se por essa solução, sendo que a textura aplicada foi a seguinte:

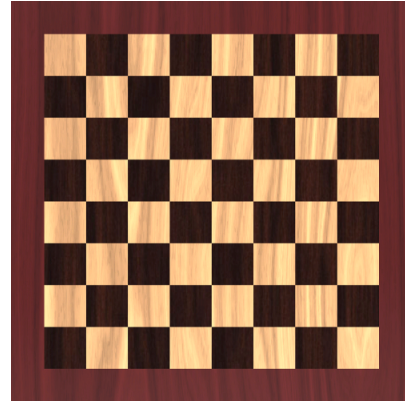


Fig. 9

TEXTURA DO TABULEIRO DE XADREZ

E para além disso ainda foi usada uma textura a simular a madeira, em que as peças do Xadrez podem usar madeira mais escura (perto de preto) ou madeira mais clara (perto de branco), conseguindo assim fazer a distinção das peças do jogador um e do jogador dois, sendo que o jogador um usa as peças com a textura branca e o jogador dois usa as peças com a textura preta. Como se pode ver na imagem seguinte o exemplo para as peças do jogador dois:

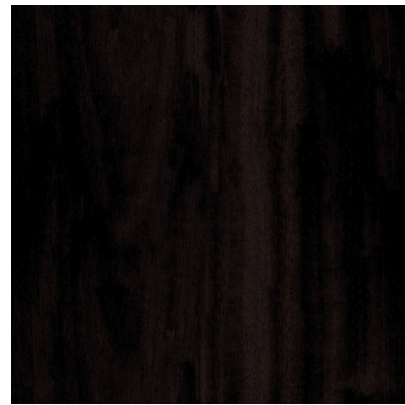


Fig. 10

TEXTURA DAS PEÇAS DO JOGADOR 2

O detalhe sobre como as texturas são carregadas é referido anteriormente no relatório na parte referente a estrutura do código em Shaders.

D. Skybox

A skybox é o conceito de criar um ambiente de fundo que simule o mundo 3D, como por exemplo, o céu, se colocar uma imagem simples do céu como fundo não vai dar a perspectiva 3D, porque se o utilizador rodar o ambiente a imagem vai se manter estática, então surgiu o conceito de Skybox que é um cubo em que as suas texturas são uma imagem completa da visão do céu a

360°, conseguindo assim dar a perspectiva que o céu também muda se mudarmos a câmera.

A imagem utilizada foi a seguinte:



Fig. 11

TEXTURA UTILIZADA PARA A SKYBOX

Sendo que esta imagem foi aplicada (através de texturas) a um cubo com dimensões superior ao tabuleiro e modificando a cor, mudando o coeficiente ambiente para (0.5, 0.8, 1) RGB, ou seja, uma espécie azul claro para simular o céu.

Se verificarmos a skybox de fora, é possível verificar que é apenas um cubo com texturas.

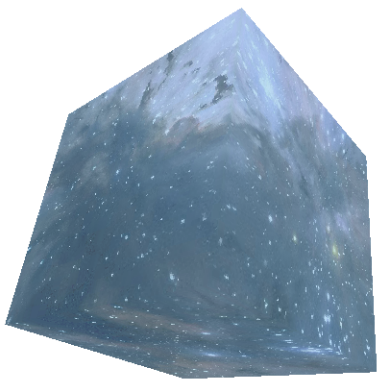


Fig. 12

TEXTURA UTILIZADA PARA A SKYBOX

Apesar que por defeito, não é permitido fazer zoom para fora da skybox, mas alterando o código nas call-backs pode ser possível.

E. Representação Gráfica

E.1 Movimentos possíveis das peças

Na imagem seguinte pode-se verificar que é possível ver todas movimentações possíveis de cada peça de Xadrez:



Fig. 13

TORRE BRANCA SELECIONADA

Nesta imagem é possível verificar que a torre branca é a peça seleccionada (casa amarela) e que pode matar a rainha preta ou o peão preto (casas vermelhas), mas também pode avançar para as casas verdes sem matar nenhuma das peças inimigas. Ainda é possível verificar que a torre tem a casa da rainha preta seleccionada.

A previsão dos movimentos são simples planos 2D, com dois triângulos para formar um quadrado proporcional a quadrícula do tabuleiro.

Distinção das casas de movimentos

A distinção dos movimentos existentes foi feita através de cores, sendo que a casa que está amarelo pertence a casa da peça que está seleccionada, as casas que estão a verde são aquelas para as quais a peça se pode movimentar sem matar nenhuma peça inimiga e as casas vermelhas são aquelas em que a peça pode-se movimentar matando a peça inimiga. Ainda temos a casa azul, que representa a casa para a qual a peça se vai mover, mais precisamente, a casa que está seleccionada.

E.2 Peças mortas

Para representar as peças quando morrem pensou-se na situação real, quando um jogador mata a peça do adversário por norma costuma-se colocar essa peça ao pé de si, neste caso, optou-se por colocar a peça do lado direito do tabuleiro em fila (a medida que vai matando as peças adversárias).

Para fazer este efeito apenas teve de ser efectuado uma translação sobre a peça que morre.

III. RESULTADO FINAL

O resultado final obtido foi o seguinte:



Fig. 14
RESULTADO FINAL

Pode-se verificar que OpenGL apesar de não ser uma biblioteca de muito alto nível, é possível fazer a representação gráfica de objectos bastante complexos usando apenas triângulos. Neste trabalho foi possível realizar texturas, skybox, leitura de ficheiros obj, iluminação na placa gráfica, motor do jogo e interacção com o utilizador (rato e teclado).

IV. COMPILAÇÃO E EXECUÇÃO

Código fonte disponível em:

<https://github.com/dbtds/chess-opengl>

Podendo ser obtido através do comando: `git clone https://github.com/dbtds/chess-opengl.git`

Este projecto foi desenvolvido unicamente em ambiente Linux, sendo assim, não foi criado qualquer ficheiro executável para ambiente Windows, apesar que também é possível fazê-lo alterando apenas a forma de compilação.

Note-se que todas as próximas indicações foram apenas realizadas em ambiente Linux no Ubuntu 14.10 64 bits.

Software essencial para compilar e correr o programa:

1. build-essential
2. qt4-qmake
3. libglew-dev
4. freeglut3-dev

Para obter cada dependência basta executar:

```
sudo apt-get install dependency_name
```

Após a obtenção de todas estas dependências, basta executar o Makefile na pasta principal (root).

Executando em bash / \$ `make`

Para executar o programa, basta ir a pasta gerada bin, e correr o executável com o nome chess. Executando em bash /bin \$ `./chess`

Já existe um executável pré-gerado apenas para 64 bits dentro da pasta bin com o nome chess_bin64, executando da mesma forma: Em bash /bin \$ `./chess_bin64`

BIBLIOGRAFIA

- [1] Pwag Chessboard, "Directx open source", <https://code.google.com/p/pwag-chessboard/>.
- [2] Joaquim Madeira, "Métodos de Iluminação e Sombreamento", 2014, http://sweet.ua.pt/jmadeira/CV/CV_07_Illumina%C3%A7%C3%A3o_e_Shading-BSS_JM.pdf.
- [3] Wikibooks, "Lightning textures surfaces", http://en.wikibooks.org/wiki/GLSL_Programming/GLUT/Lighting_Textured_Surfaces.
- [4] Wikibooks, "Modern opengl tutorial 06", http://en.wikibooks.org/wiki/OpenGL_Programming/Modern_OpenGL_Tutorial_06.
- [5] Wikibooks, "Object selection", http://en.wikibooks.org/wiki/OpenGL_Programming/Object_selection.