Universidade da Beira Interior

Departamento de Informática

Nº 9 - 2021: *Tetris 3D*

Elaborado por:

Afonso Correia - nº41130
Carlos Martins - nº41968
Rafael Louro - nº41855
Rui Ferreira - nº41064

Docente:

Professor Doutor Abel Gomes

7 de fevereiro de 2021

Conteúdo

Capítulo

1

Introdução

1.1 Enquadramento

De modo a realizar com sucesso o projeto da cadeira de computação gráfica, optámos por escolher o projeto “Tetris 3D”.

1.2 História do jogo

O Tetris é considerado um jogo de quebra-cabeças, desenvolvido por Alexey Pajitnov, enquanto trabalhava no Centro de Informática de Dorodnicyn, em Moscovo.
O seu lançamento ocorreu no dia 6 de junho de 1984, e o seu nome deriva da junção do prefixo grego “tetra” – uma vez que todos os blocos são formados por quatro segmentos – com a palavra Ténis, dado a ser o desporto favorito do programador.
O desenvolvimento do algoritmo do funcionamento do Tetris foi inspirado também num jogo de quebra-cabeças popular, o poliminó.
O mecanismo de funcionamento deste jogo baseia-se em empilhar tetraminós que descem a janela do jogo numa velocidade que aumenta à medida que a dificuldade do jogo também evolui, de forma a completar linhas horizontais.
No momento que se forma uma linha completa, as camadas superiores caem, e consequentemente o jogador ganha pontos. Quando a pilha de peças chega ao topo da tela, o jogo termina.

Desde a sua criação, o Tetris é um dos jogos mais jogados mundialmente.

1.3 Motivação

A principal motivação encontrada para o desenvolvimento deste projeto vai de encontro com o interesse em aplicar na prática os conhecimentos adquiridos ao longo do semestre, os quais podem vir a ser bastante úteis quer para nossa vida universitária quer para a futura vida pro- fissional.

1.4 Objetivos

Alguns dos objetivos que nos propomos a cumprir vão de encontro a oferecer uma jogabilidade simples, bem como ser visualmente apelativo, para além dos requisitos necessários para que tudo funcione de acordo com o pretendido.

1.5 Organização do Documento

De modo a refletir o trabalho que foi feito, este documento encontra-se organizado da seguinte forma:

1. O primeiro capítulo – **Introdução** – apresenta o projeto, a motivação para a sua escolha, o enquadramento para o mesmo, os seus objetivos e a respetiva organização do documento.

2. O segundo capítulo – **Tecnologias Utilizadas** – descreve os conceitos mais importantes no âmbito deste projeto, bem como as tecnologias utilizadas durante do desenvolvimento da aplicação.

3. No terceiro capítulo – **Desenvolvimento e Implementação** – explicamos de forma mais detalhada a gestão do projeto e a maneira como foi implementado o código.

4. No quarto capítulo – **Conclusões** – é feita uma reflexão crítica sobre os objetivos atingidos e não atingidos, bem como o que se poderia melhorar futuramente.

Capítulo

2

Tecnologias Utilizadas

Neste projeto, abordamos e explorámos mais profundamente algumas das tecnologias abordadas no decorrer das aulas (GLEW, GLFW, GLM, IRRKLANG, FRRETYPE).

Com base na biblioteca GLFW, conseguimos, quando o jogo está a decorrer, redirecionar os vários blocos do Tetris a partir das setas do teclado, mudar as posições de chegada dos mesmos. Para o rato definimos também funcionalidades relativas à mudança de perspetiva da janela do jogo.

A utilização da biblioteca GLM foi também bastante útil, dado a preencher as lacunas de programação em escrever a matemática básica de vetores e matrizes para aplicativos OpenGL, e essencial neste caso para a construção das várias peças do jogo e da atribuição de cor às mesmas.

Outras das bibliotecas usadas para a realização do projeto consiste na GLEW, responsável por fornecer mecanismo de ajuda na consulta e carregamento de extensões OpenGL. Esta biblioteca disponibiliza algoritmos de otimização de tempo de execução para determinais quais extensões de OpenGL são suportadas na plataforma destino. A GLEW está definida no cabeçalho do programa.

Com a realização deste projeto, procuramos também implementar bibliotecas que até então nunca tinham sido abordadas nas aulas, é o caso da biblioteca IrrKang, a qual torna possível a existência de uma música e fundo no decorrer do jogo.

Apesar de não estar implementada correta, tentámos também adotar a biblioteca FREETYPE, de modo a conseguir colocar texto gráfico (incluindo a pontuação) na tela do jogo. No entanto, não conseguimos concluir esse objetivo com sucesso.

Capítulo

3

Desenvolvimento e Implementação

3.1 Etapas do desenvolvimento

Inicialmente o grande foco centralizou-se na definição das arestas e dos pontos essenciais para a formação dos cubos, que não só iriam ser fundamentais na criação das peças do Tetris, como no desenvolvimento da janela de jogo. Deste modo, depois de criadas as várias peças e a janela de jogo estar definida focamo-nos na elaboração das funções que permitissem o movimento da mesma. Como consequência do movimento e das próprias funcionalidades do jogo, foi necessário desenvolver uma função que verifica—se as colisões ao longo da evolução do jogo. Em adjacência a esta função, foi também necessário criar uma função eu permitisse a atualização das várias posições onde já se encontravam peças e no movimento da peça para o local desejado para o utilizador, verificando se simultaneamente se existia colisão. Caso existisse os valores das variáveis de posições eram atualizados. Para os momentos em que uma linha se completava, foi também necessário criar uma função para verificar essa condição e, consequentemente, para eliminar a linha.
Em adição, foi também necessário criar e aplicar os shaders de modo a que a informação pudesse passar para a GPU e, por consequência, poder ser visualizada.

3.2 Descrição do funcionamento do Software

Para que a peça caia automaticamente na coluna em que se encontra, basta pressionar a seta para baixo. A cada linha completa, a mesma é eliminada, sendo que o jogo continua com as restantes linhas. Para efetuar movimentos da janela de jogo, basta pressionar "a", "s", "d" ou w", e para mover o ecrã é apenas necessário arrastar com o rato.
Se o utilizador decidir pausar o jogo, basta pressionar a tecla "p". Durante o decorrer do jogo, o jogo é acompanhado por uma música de fundo.
Caso o jogador não consiga completar linhas, se as peças atingirem o topo do painel do jogo, o mesmo irá terminar, uma vez que perdeu o jogo.

```
if (!pause) {
    switch (key)
    {
    case GLFW_KEY_P:
        pause = !pause;
        break;
    case GLFW_KEY_RIGHT:
        active_figure->MoveRight(field_frame.pos, cube_size, falled_cubes.buffer,
            field_frame.width, field_frame.height);
        break;
    case GLFW_KEY_LEFT:
        active_figure->MoveLeft(field_frame.pos, cube_size, falled_cubes.buffer,
            field_frame.width, field_frame.height);
        break;
    case GLFW_KEY_DOWN:
        while (active_figure->MoveDown(field_frame.pos, cube_size, falled_cubes.buffer,
            field_frame.width, field_frame.height));
        break;
    default:
        break;
    }
}
else {
    switch (key)
    {
    case GLFW_KEY_P:
        pause = !pause;
        break;
    default:
        break;
    }
}
```

Figura 3.1: Excerto da função ***KeyPressHandler (int key, int scancode, int action, int mods)***, responsável pela atribuição de funções ao teclado.

Neste caso, podemos verificar que a seta do lado direito do teclado é responsável pelo movimento lateral direito da peça na janela de jogo, o mesmo dizemos para a seta esquerda, responsável pelo movimento lateral esquerdo. Relativamente ao momento em que pressionados a tecla p, o jogo fica pausado. Já a seta com orientação para cima, permite girar a peça de modo a alcançar a melhor posição para encaixar nas outras na ótica do jogador. Todas estas funcionalidades tornam-se possíveis graças à utilização da biblioteca GLFW.
Através das teclas (“A”,” S”,” D”,” W”) conseguimos movimentar a camara do jogo, sendo deste modo possível a visualização da janela de jogo de diferentes perspetivas.

```
// "guardar" uma figura no momento da colisão
void Game::SaveFalledFigure()
{
    for (int y = 0; y < 4; y++)
        for (int x = 0; x < 4; x++)
            if (active_figure->form[y * 4 + x] != 0) {
                glm::vec3 pos = active_figure->form[y * 4 + x]->GetPosition();
                int xx = (int)((pos.x - field_frame.pos.x) / cube_size);
                int yy = (int)((field_frame.pos.y - pos.y) / cube_size - 1);
                if (y >= 0) {
                    falled_cubes.buffer[yy * field_frame.width + xx] = active_figure->form[y * 4 + x];
                }
                else
                    delete active_figure->form[y * 4 + x];
            }
    PushDownFalledCubes();
}
```

Figura 3.2: Função ***SaveFalledFigure()***

O principal objetivo com a implementação desta função destina-se a guardar a posição da peça após ter colido com as que já estavam na janela no jogo, tendo no final de verificar se com esta nova posição formamos uma linha completa (verificação feita a partir da função ***PushDownFalledCubes().***

```
// tirar uma linha completa
void Game::PushDownFalledCubes()
{
    for (int y = field_frame.height - 1; y > 0; y--)
    {
        bool row_is_full = true;
        for (unsigned int x = 0; row_is_full && x < field_frame.width; x++)
            if (falled_cubes.buffer[y * field_frame.width + x] == nullptr)
                row_is_full = false;
        if (row_is_full) {
            for (unsigned int x = 0; x < field_frame.width; x++)
            {
                delete falled_cubes.buffer[y * field_frame.width + x];
                falled_cubes.buffer[y * field_frame.width + x] = nullptr;
            }
            for (int i = y; i > 0; i--)
            {
                for (unsigned int j = 0; j < field_frame.width; j++)
                {
                    Cube* cube = falled_cubes.buffer[(i - 1) * field_frame.width + j];
                    if (cube != nullptr) {
                        falled_cubes.buffer[i * field_frame.width + j] = cube;
                        cube->SetPosition(glm::vec3(0.0f, -cube_size, 0.0f) + cube->GetPosition());
                        falled_cubes.buffer[(i - 1) * field_frame.width + j] = nullptr;
                    }
                }
            }
            y++;
        }
    }
}
```

Figura 3.3: Função ***PushDownFalledCubes()***

Através desta função, conseguimos eliminar uma linha completa na janela do jogo, caso a linha esteja completa.

```
//verificar colisões dos blocos
bool CheckCollision(const glm::vec3& frame_pos, float cube_size,
    Cube** falled_cubes, int frame_width, int frame_height)
{
    for (int i = 0; i < 4 * 4; i++)
        if (form[i] != nullptr) {
            glm::vec3 block_pos = form[i]->GetPosition();

            int xx = (int)((block_pos.x - frame_pos.x) / cube_size);
            if (xx < 0 || xx >= frame_width)
                return true;

            int yy = (int)((frame_pos.y - block_pos.y) / cube_size - 1);
            if (yy >= frame_height)
                return true;

            if (yy >= 0 && falled_cubes[frame_width * yy + xx] != 0)
                return true;
        }
    return false;
}
```

Figura 3.4: Função ***bool CheckCollision(..)***

Esta função tem como principal objetivo verificar se a peça que está em movimento já colidiu com alguma das que já se encontravam na janela do jogo, caso ainda não tenha acontecido a peça continua a mover-se até ao momento da colisão.

```
//construção da janela do jogo
Cube::Cube()
{
#define PATTERN edge_cube_pattern
    vertices_count = sizeof(PATTERN) / sizeof(glm::vec3);
    vertices = new Vertex3D[vertices_count];

    for (int i = 0; i < vertices_count; i++)
    {
        vertices[i].pos = PATTERN[i];
        vertices[i].color = glm::vec3(0.0f, 1.0f, 1.0f);
    }

    glGenBuffers(1, &vbo);
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glBufferData(GL_ARRAY_BUFFER, sizeof(Vertex3D) * vertices_count, vertices, GL_STATIC_DRAW);

    glEnableVertexAttribArray(0);
    glEnableVertexAttribArray(1);

    model_matrix = glm::mat4(1.0f);
}
```

Figura 3.5: Função **Cube ()**

Esta função é responsável pelo desenho da janela do jogo, a qual é formada por cubos.

```
void Game::SetUpFieldFrame()
{
    field_frame.cubes = new Cube[field_frame.cubes_count];

    unsigned int i;
    for (i = 0; i < field_frame.width * 2; i += 2)
    {
        field_frame.cubes[i].SetPosition(glm::vec3(
            field_frame.pos.x + cube_size * ((i / 2)),
            field_frame.pos.y,
            field_frame.pos.z
        ));
        field_frame.cubes[i + 1].SetPosition(glm::vec3(
            field_frame.pos.x + cube_size * (i / 2),
            field_frame.pos.y - cube_size * field_frame.height - cube_size,
            field_frame.pos.z
        ));
    }

    for (i = field_frame.width * 2; i < field_frame.cubes_count - 2; i += 2)
    {
        field_frame.cubes[i].SetPosition(glm::vec3(
            field_frame.pos.x - cube_size,
            field_frame.pos.y - cube_size * ((i - field_frame.width * 2) / 2),
            field_frame.pos.z
        ));
        field_frame.cubes[i + 1].SetPosition(glm::vec3(
            field_frame.pos.x + cube_size * field_frame.width,
            field_frame.pos.y - cube_size * ((i - field_frame.width * 2) / 2),
            field_frame.pos.z
        ));
    }

    field_frame.cubes[field_frame.cubes_count - 1].SetPosition(glm::vec3(
        field_frame.pos.x - cube_size,
        field_frame.pos.y - cube_size * (field_frame.height + 1),
        field_frame.pos.z
    ));

    field_frame.cubes[field_frame.cubes_count - 2].SetPosition(glm::vec3(
        field_frame.pos.x + cube_size * field_frame.width,
        field_frame.pos.y - cube_size * (field_frame.height + 1),
        field_frame.pos.z
    ));
```

Figura 3.5: Função **SetUpFieldFrame ()**

O objetivo central da função **SetUpFielFrame()** consiste na atualização dos valores das posições na janela de jogo aquando a inserção de uma nova peça, tendo um papel fundamental para o sucesso da função ***bool CheckCollision(..)***, dada a última necessitar das coordenadas atualizadas para realizar a verificação de colisões com o aparecimento de novas peças.

```
//função que vai permitir que os blocos se movam
void Move(const glm::vec3& mov)
{
    pos = pos + mov;
    for (int i = 0; i < 4 * 4; i++)
        if (form[i] != 0) {
            glm::vec3 pos = form[i]->GetPosition() + mov;
            form[i]->SetPosition(pos);
        }
}
```

Figura 3.5: Função **Move(…)**

Esta função é a responsável pelo movimento para baixo do bloco.

```
//construir peças de jogo
void GenerateNew(const glm::vec3& frame_pos, float cube_size, int frame_width, int frame_height)
{

    //Define os primeiros num bytes do bloco de memória apontado por ptr para o valor especificado (interpretado como um caractere sem sinal ).

    std::memset(form, 0, sizeof(Cube*) * 4 * 4);


    //vetor que guarda as cores do vetor
    const static glm::vec3 colors[3] = { {1.0f, 0.3f, 0.6f}, {1.0f, 0.8f, 0.7f}, {0.5f, 1.0f, 0.5f} };
    srand((unsigned int)(time(0)));

    int x = 1 + rand() % (frame_width - 2);
    int y =-5;

    pos = frame_pos;
    pos.x += x;
    pos.y += -y;

    int f     = 0 + rand() % 3;
    int color = 0 + rand() % 3;

    if (f == 0) {
        int index = 0;
        form[index] = new Cube();
        form[index]->SetPosition(glm::vec3(
            frame_pos.x + x * cube_size,
            pos.y - 0 * cube_size,
            frame_pos.z
        ));
        form[index]->SetColor(colors[color]);
        form[index]->Scale(scale_factor);

        index = 1 * 4 + 0;
        form[index] = new Cube();
        form[index]->SetPosition(glm::vec3(
            frame_pos.x + x * cube_size,
            pos.y - 1 * cube_size,
            frame_pos.z
        ));
        form[index]->SetColor(colors[color]);
        form[index]->Scale(scale_factor);

        index = 2 * 4 + 0;
        form[index] = new Cube();
        form[index]->SetPosition(glm::vec3(
            frame_pos.x + x * cube_size,
            pos.y - 2 * cube_size,
            frame_pos.z
        ));
        form[index]->SetColor(colors[color]);
        form[index]->Scale(scale_factor);
```

Figura 3.5: Função **GenerateNew(..)**

Esta função é responsável pela criação das várias peças do jogo.

Capítulo

4

Conclusões e Trabalho Futuro

4.1 Conclusões Principais

Elaborando uma reflexão critica relativamente aos objetivos propostos versus os objetivos atingidos, concluímos que estamos satisfeitos com o trabalho feito e as aprendizagens adquiridas e solidificadas com o desenvolvimento do projeto.
Com a execução deste trabalho, não só conseguimos aprofundar os nossos conhecimentos no OpenGL, como na exploração das diversas funcionalidades existentes.
Paralelamente, adquirimos também conhecimento sobre uma linguagem que pouco que tínhamos utilizado até então, e que pode ser útil quer na vida académica quer na futura vida profissional.
Relativamente à dinâmica do funcionamento do grupo, tentámos distribuir as tarefas de forma equivalente de modo a que todos tivéssemos a mesma carga de trabalho, contudo fomos estabelecendo contacto regularmente com o intuito de cooperar na elaboração e resolução dos problemas que apareciam com o desenvolvimento do projeto.
Resumidamente, podemos concluir que tivemos uma dinâmica bastante positiva.

4.2 Trabalho Futuro

Futuramente, um dos aspetos a melhorar no projeto desenvolvido assenta na criação de um menu na fase inicial do jogo, a partir do qual é possível selecionar o nível de dificuldade que se pretende jogar, passando a existir três níveis (nível iniciante, nível intermédio, nível avançado). Outro tema a abordar, foca-se na adição de pontuação a cada linha completa. Por fim, outro aspeto a ter em conta consiste numa opção no final de cada jogo que permite a possibilidade de voltar a jogar novamente.

Bibliografia

Learn OpenGL, extensive tutorial resource for learning Modern OpenGL (último acesso 2 fevereiro)

GitHub - https://github.com/ankostenko/glTetris (último acesso 1 de fevereiro)

GitHub - andykhv/Tetris3D: Tetris powered by OpenGL(último acesso 22 janeiro)

GitHub - carlos93/Tetris3D: Game based on Tetris. Uses OpenGL for 3D de- sing. (último acesso 13 janeiro)

Lecture Notes - http://www.di.ubi.pt/~agomes/cg/#ModulosTeoricos (último acesso 28 de janeiro)