Computação Gráfica - Tetris 3D

Universidade da Beira Interior

2022/2023

Afonso Oliveira, a44404
Nuno Monteiro, a43803
Rúben Abadesso, a43664
Margarida Silva, a43367

Motivação

Neste projeto pretende-se criar uma implementação do jogo Tetris, projetado em 3 dimensões, que consiste em empilhar blocos dentro de um campo de jogo retangular. Quando uma linha se forma, ela desintegra-se e as camadas superiores descem uma linha, o que resulta num acumular de pontos para o jogador. Quando a pilha de peças chega ao topo do repositório, a partida termina.

Este tema foi escolhido pela familiaridade do grupo de trabalho com o jogo, facilitando todo o processo de criação das diversas mecânicas do jogo.

Tecnologias Utilizadas

Pela necessidade de colaboração inerente a qualquer projeto de grupo foi necessário criar um repositório de git, para o qual escolhemos a plataforma GitHub.

Foi utilizada para a maior parte do desenvolvimente a IDE Visual Studio, da Microsoft, embora pequenos detalhes foram ajustados através de editores de texto mais simples, como o Sublime Text.

A implementação em C++ dependeu ainda de algumas bibliotecas externas:

- OpenGL, glad, GLFW, GLM e FreeType

Etapas de Desenvolvimento

Devido à natureza deste projeto, as etapas de desenvolvimento e a sua ordem nascem de forma orgânica, sem a necessidade de um planeamento prévio extensivo.

A primeira etapa baseou-se na definição dos pârametros da câmara e a criação do palco de jogo (o campo retângular), seguido da definição das várias peças necessárias e a posição onde devem ser geradas inicialmente.

Após estes passos, foi necessário implementar:

- um sistema de *inputs* que permitissem controlar as peças de acordo com as regras do jogo (translação e rotação, com os *offsets* necessários associados);
- um sistema de *gravidade* (isto é, um sistema que faça com que a peça *caia* ao longo do tempo);
- um sistema de deteção de colisão e
- um outro sistema que lide com essa mesma colisão (impedindo que a peça saia do frame lateralmente, que guarde a sua posição caso colida com uma peça de jogo ou frame descendentemente, e que gere uma nova peça caso isso aconteça).

Com a base do jogo criada, foi implementada uma *skybox*, a mecânica de **eliminação de linhas**, a mecânica de **final do jogo**, assim como um sistema de pontuação e recordes (incluíndo o *text rendering* associado) e a capacidade de instantâneamente pousar a peça.

Descrição do funcionamento do Software

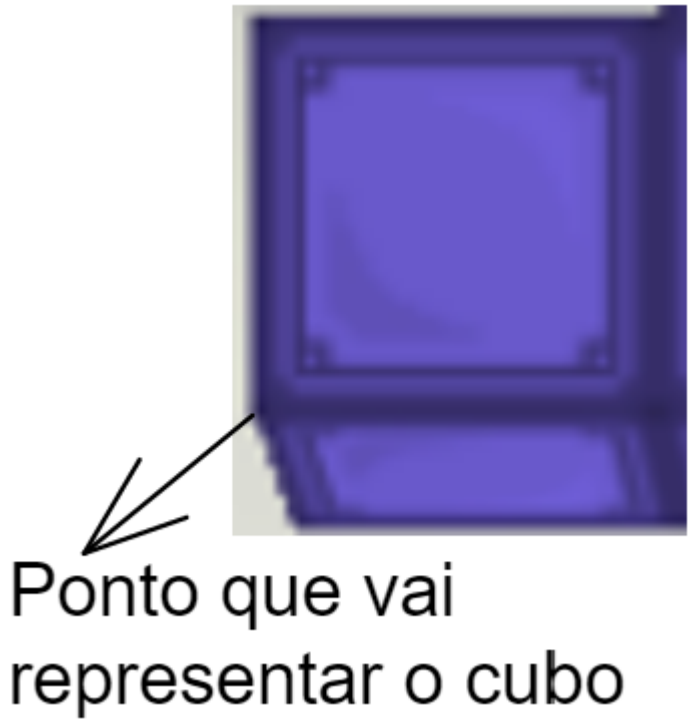
Controlos

O jogador pode interagir com as peças sobre o seu controlo utilizando o teclado:

- As setas para a direita, esquerda, e baixo controlam a direção do movimento da peça;
- As teclas X e Z permitem rodar as peças;
- O espaço permite pousar a peça instantâneamente;

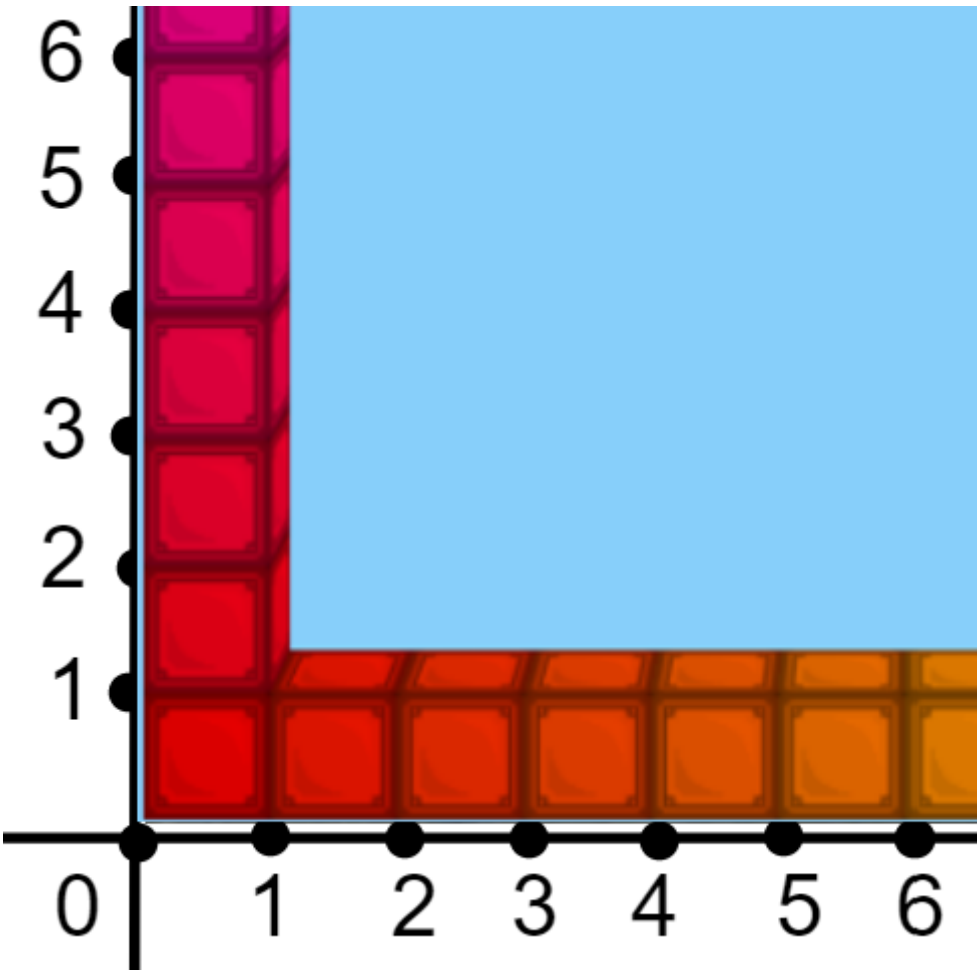
Cubo

Ambos as paredes (limites do mapa) e as próprias peças são simplesmente a repetição de um único cubo, através de translações.
Este cubo tem dimensão (1,1,1), e é representado (nas translações) pelo ponto inferior esquerdo.



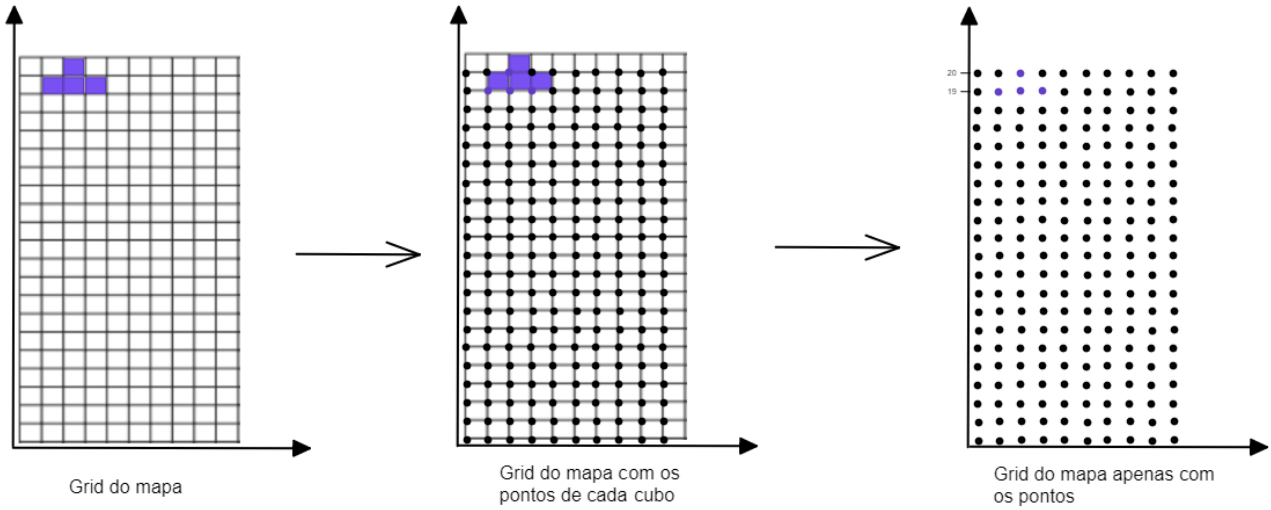
Grid

O campo do jogo, apesar de técnicamente ser em 3d, funciona em função da dimensão xy. Cada cubo (como referido anteriormente) tem 1 de lado, pelo que o mapa começa na origem (0,0) e então cada cubo é representado por um simples ponto.



Colisão

Para a colisão foi usado o método dos pontos, isto é, como cada cubo apenas se "mexe" em incrementos de 1 (por exemplo: 3 *1, 4* 1, nunca números decimais) então concluiu-se que eles podiam ser simplesmente representados pelos seus pontos usados na translação.
Por este motivo, o mapa do jogo é representado por uma matriz de dimensão 21X10, que vai corresponder aos 210 cubos (e, logo, posições) totais disponíveis no mapa do jogo.
Por fim, as posições das paredes são simplesmente os pontos usados para a translação.



Como se pode ver neste exemplo, cada cubo da peça é representado por um ponto. Em termos de código, significa que a matriz teria todos os valores a -1, exceto os valores na

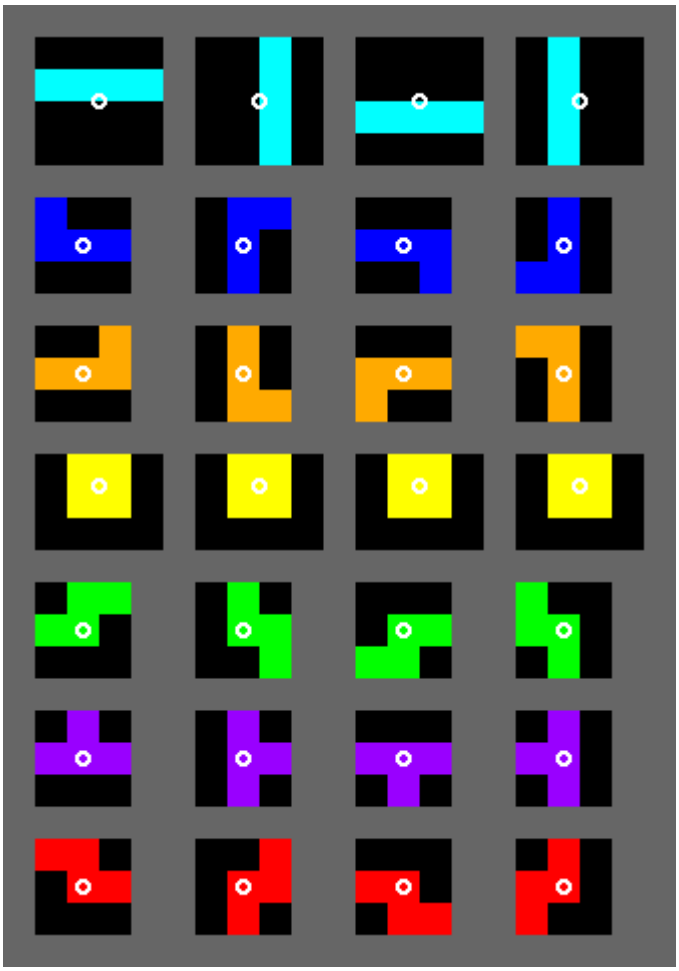
posição (1,19), (2, 19), (3,19) e (2,20).
Os valores da matriz são -1 quando não há nenhuma peça nessa posição, e quando há o valor vai ser um inteiro entre 0 a 6, dependendo da peça. A razão para isto é para posteriormente sabermos a cor do cubo que ocupa uma certa posição.

Construção das peças

Cada peça e respetiva cor é representado por um inteiro entre 0 e 6, sendo que este inteiro corresponde à posição num array de vetores das posições e num array de vetores das cores. Por exemplo, a peça "I" tem o valor de 0, ou seja, num array o index 0 vai corresponder aos vetores correspondentes à peça "I", e num outro vetor o index 0 corresponde à cor designada para a peça "I".

As peças, primeiramente, são construídas na base, isto é, na posição (0,0), sendo que o "centro" da peça tem a coordenada (0,0).
O centro da peça é dado pelas regras oficiais do tetris. Isto é feito devido à necessidade de rodar as peças posteriormente.
De seguida, as peças são transladadas para a posição atual no jogo, através de um único vetor de translação.

Rotacao



A rotação das peças ocorre na origem, antes da translação para a posição atual. Porém, ao contrário da translação que, quando ocorre uma colisão simplesmente fica na posição atual

(a não ser que tenha atingido o "fundo" do mapa), na rotação as peças têm obrigatóriamente de rodar (segundo as regras do tetris). Para isto, foram usados offsets (também estes tirados das regras do tetris) que, em caso de a rotação (mais especificamente, a translação após a rotação) causar uma colisão, as peças sofrem uma segunda translação até que se encontrem numa posição adequada.
Para o jogo do tetris existem três casos distintos:

- Se a peça for o quadrado, não sofre qualquer rotação
- Se for a peça I, usa uma tabela de offsets especializada (visto que é a única peça que pode ter mais dois cubos para além do cubo central)
- Para as restantes peças, é usada uma única matriz de offsets

I Tetromino Wall Kick Data

	Test 1	Test 2	Test 3	Test 4	Test 5
0>>1	basic rotation	(-2, 0)	(1, 0)	(-2,-1)	(1, 2)
1>>0	basic rotation	(2, 0)	(-1, 0)	(2, 1)	(-1,-2)
1>>2	basic rotation	(-1, 0)	(2, 0)	(-1, 2)	(2,-1)
2>>1	basic rotation	(1, 0)	(-2, 0)	(1,-2)	(-2, 1)
2>>3	basic rotation	(2, 0)	(-1, 0)	(2, 1)	(-1,-2)
3>>2	basic rotation	(-2, 0)	(1, 0)	(-2,-1)	(1, 2)
3>>0	basic rotation	(1, 0)	(-2, 0)	(1,-2)	(-2, 1)
0>>3	basic rotation	(-1, 0)	(2, 0)	(-1, 2)	(2,-1)

J, L, T, S, Z Tetromino Wall Kick Data

	Test 1	Test 2	Test 3	Test 4	Test 5
0>>1	basic rotation	(-1, 0)	(-1, 1)	(0,-2)[1]	(-1,-2)
1>>0	basic rotation	(1, 0)	(1,-1)	(0, 2)	(1, 2)
1>>2	basic rotation	(1, 0)	(1,-1)	(0, 2)	(1, 2)
2>>1	basic rotation	(-1, 0)	(-1, 1)[1]	(0,-2)	(-1,-2)
2>>3	basic rotation	(1, 0)	(1, 1)[1]	(0,-2)	(1,-2)

3>>2	basic rotation	(-1, 0)	(-1,-1)	(0, 2)	(-1, 2)
3>>0	basic rotation	(-1, 0)	(-1,-1)	(0, 2)	(-1, 2)
0>>3	basic rotation	(1, 0)	(1, 1)	(0,-2)[1]	(1,-2)

Os números representam a rotação atual, em funcção da posição original da peça:

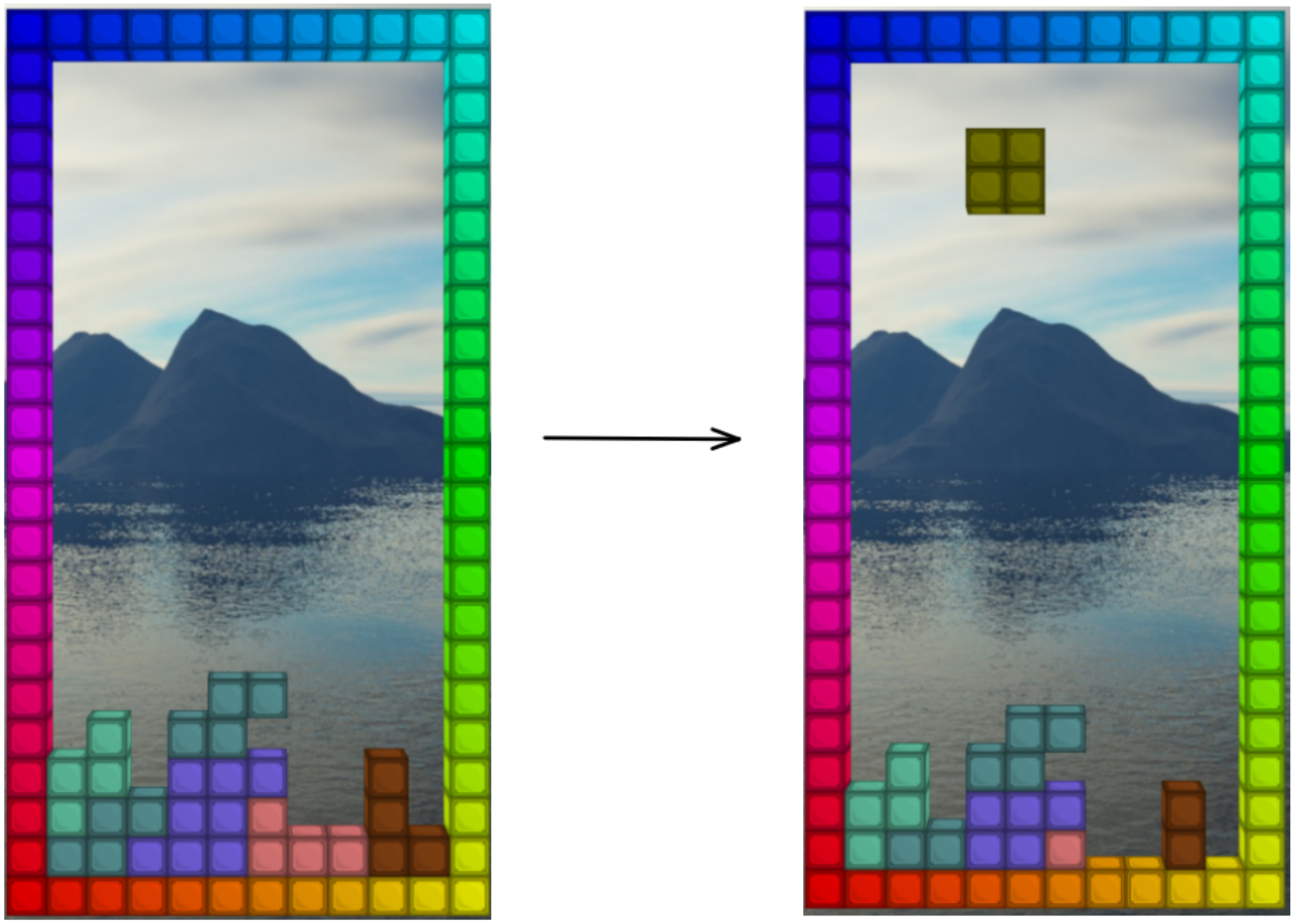
- 0: posição original
- 1: posição após uma rotação na direção dos ponteiros do relógio
- 2: posição após duas rotações em qualquer direção
- 3: posição após uma rotação contra-relógio a partir da origem, ou 3 rotações na direção dos ponteiros do relógio a partir da origem

Os offsets (designados por tests) são aplicados por ordem, até que um funcione (a peça deixa de causar colisão).
No caso de nenhum dos offsets funcionar (acontecimento muito raro), a peça não sofre qualquer rotação.

Um exemplo: no caso da peça "L", após 2 rotações contra-relógios, tentar efetuar uma rotação no sentido dos ponteiros do relógio e isto causar uma colisão, são aplicados os offsets representados na linha "2>>1" na segunda matriz. Neste caso, ele tenta aplicar uma translação na direção dada pelo vetor (-1.0, 0.0) (o primeiro teste foi executado assim que a rotação foi tentada). Se esta translação for bem sucedida, a peça fica com essa nova posição, caso contrário executa a translação dada pelo vetor (-1.0, -1.0), a partir da posição dada pela primeira translação normal (ou seja, ignorando a translação dada pelo test2). E por aí adiante até algum offset funcionar ou nenhum funcionar (a peça fica igual).

Movimentação, registo das peças e pontos

Como se sabe, no jogo tetris as peças descem sozinhas, isto é, executam uma translação na direçao (0,-1) a cada (apróximadamente) 1 segundo. Para isto foi criado uma thread que corre paralelamente ao processo principal, que vai tentar descer a peça até sofrer alguma colisão. Quando a colisão acontece, a peça é registada na matriz das peças no mapa (mudando, na posição de cada cubo na matriz, o valor de -1 para o valor do index da cor do cubo, pertencente à peça atual). Após este registo, a thread vai ainda verificar se com esta peça foi criada uma linha consecutiva de cubos. Caso tenha sido, os pontos aumentam em 1 (por cada linha) e a linha feita é apagada, e as linhas acima desta são movidas uma posição para baixo.



> Linha removida após a última peça ser introduzida (peça "L")

Se a última peça introduzida tiver algum cubo na linha mais superior possível do mapa (linha 21), isto significa que o jogador perdeu, pelo que o jogo sofre um reset, isto é, as peças previamente colocadas são apagadas (a matriz volta a ficar com todos os campos igual a -1) e os pontos voltam a 0.

Trabalhos Futuros

Embora o projeto tenha atingido os seus objetivos, no futuro poderiam ser desenvolvidas mais funcionalidades.

A criação de um sistema de dificuldade, por exemplo, assim como uma GUI que permitisse ao utilizador modificar os pârametros desse mesmo sistema antes de começar o jogo.
Seria também interessante implementar peças "fantasma", que permitiriam ao jogador pré-visualizar onde a peça atual irá cair, por exemplo, quando pressionar espaço.

Considerações Finais

Este projeto fomentou uma maior familiaridade com OpenGL e C++, assim como todas as bibliotecas necessárias para a aplicação desse API.
Forneceu a todo o grupo de trabalho o conhecimento necessário para o desenvolvimento de soluções gráficas em OpenGL, assim como permitiu a evolução natural de *soft skills* relacionadas com o desenvolvimento de soluções em grupo.

Afonso Oliveira, Nuno Monteiro, Rúben Abadesso, Margarida Silva | Computação Gráfica @ UBI