Tetris 2D

Relatório de Projeto da Unidade Curricular
Computação Gráfica

26 de Abril de 2019

Tiago Roxo
Joana Costa

Conteúdo

Acrónimos

2D	Duas Dimensões
3D	Três Dimensões
API	*Application Program Interface*
CR	Centro de Rotação
CG	Computação Gráfica
CPU	*Central Processing Unit*
GLEW	*OpenGL Extension Wrangler Library*
GLFW	*Graphics Library Framework*
GLM	*OpenGL Mathematics*
GLSL	*OpenGL Shading Language*
IDE	*Integrated Development Environment*
I/O	*Input/Output*
OpenGL	*Open Graphics Library*
UC	Unidade Curricular

Capítulo 1

Motivação

O uso de *Open Graphics Library* (OpenGL) permite desenvolver aplicações cuja parte gráfica é independente do Sistema Operativo e do Sistema de Janelas. A utilização da versão moderna desta *Application Program Interface* (API) permite uma melhor gestão de recursos, fazendo uso de *shaders* para o processamento gráfico. A separação entre as componentes vértices (*vertexshader*), cores (*fragmentshader*) e *Central Processing Unit* (CPU) permite uma melhor organização do projeto, justificando tal independência. Assim, uma das motivações associadas a este projeto é desenvolver um jogo de *Tetris*, em Duas Dimensões (2D), com a arquitetura *pipeline* mais recente, sem *fixed-function* (presentes em versões OpenGL antigas) [1].

Outra motivação associada a este projeto é o desenvolvimento do jogo de *Tetris* onde é possível aplicar conceitos estudados durante a Unidade Curricular (UC) de Computação Gráfica (CG), nomeadamente:

- Translação, que permite a movimentação da peça na janela de visualização e a descida contínua da mesma (associada a um temporizador);
- Rotação, permitindo rotação da peça sobre ela própria;
- Projeção, facultando a possibilidade de projetar o jogo num eixo de coordenadas personalizado;
- Interação com sistemas de *Input/Output* (I/O), que permite ao utilizador influenciar a posição da peça e a sua orientação.

Relativamente à escolha em particular do projeto de *Tetris* 2D, em detrimento dos restantes apresentados, deveu-se a uma maior familiaridade com este jogo o que facilita uma definição de quais as funcionalidades que deverão estar presentes neste. Quaisquer dúvidas relativamente ao comportamento de rotações ou translações podem ser facilmente colmatadas por *testing* de um *Tetris* já existente, o que permite que concentremos esforços na implementação das funcionalidades e não nas escolha de quais as que deveriam existir.

O desafio associado à implementação de translações e rotações das peças, bem como a gestão de colisão destas, surtiu num desafio associado ao projeto que nos

aliciou a o escolher. A possibilidade de migrar este projeto para Três Dimensões (3D), reutilizando implementações de 2D, foi uma motivação acrescida à escolha feita.

Capítulo 2

Tecnologias Utilizadas

Para a realização deste projeto foi utilizado o *Integrated Development Environment* (IDE) *Microsoft Visual Studio*, para a implementação do jogo de *Tetris* em 2D, utilizando como linguagens de programação *C++* e *OpenGL Shading Language* (GLSL). GLSL é uma linguagem de *shading* que tem por base a linguagem *C*, usada na criação de ficheiros com extensão *vertexshader* e *fragmentshader*, usados para o processamento de vértices e cor, respetivamente. A incorporação dos ficheiros de *shaders* no projeto desenvolvido foi feita usando como referência a metodologia apresentada nas aulas [3]. A API usada, OpenGL, permite estabelecer a comunicação entre o processador e a placa gráfica, contendo uma panóplia de bibliotecas que auxiliam na aplicação de mecanismos de translação, rotação de objetos e alteração do plano de projeção e perspetiva destes na janela de visualização. As bibliotecas utilizadas para o desenvolvimento deste projeto foram:

- ***OpenGL Extension Wrangler Library* (GLEW):** Estender funcionalidades do OpenGL;
- ***Graphics Library Framework* (GLFW):** Criar janelas e *viewports*;
- ***OpenGL Mathematics* (GLM):** Contém expressões matemáticas utilizáveis para projeções, translações e rotações.

A componente teórica subjacente às funcionalidades implementadas, usando OpenGL, é o conceito de geometria projetiva, presente no *pipeline* gráfico, que contém rotações, translações e reflexões, da geometria euclidiana e *scaling* e *shearing* da geometria afim [2]. Para melhor compreensão da organização do projeto e das funcionalidades implementadas, produziu-se um documento, gerado automaticamente a partir do código, por recurso à ferramenta *Doxygen*. O desenvolvimento do relatório de projeto foi feito utilizando o sistema de preparação de documentos LaTeX.

Capítulo 3

Etapas de Desenvolvimento

Para uma exposição clara e detalhada da evolução do trabalho optou-se por discriminar qual o trabalho feito em cada semana, fazendo referência da importância deste para a implementação do projeto escolhido.

Na semana de 04/03/2019, foi definido o tema do projeto e iniciou-se a redação preliminar de capítulos do relatório, nomeadamente:

- Capítulo 1, Motivação;
- Capítulo 2, Tecnologias Utilizadas;
- Capítulo 3, Etapas de Desenvolvimento;
- Capítulo 7, Referências Bibliográficas.

Na semana de 11/03/2019, explorou-se funcionalidades associadas ao OpenGL que pudessem ter aplicabilidade no projeto. Realizaram-se as seguintes tarefas:

- Aplicação de funcionalidades GLM (*glm::ortho*) para projeção de peças num eixo diferente do por defeito;
- Exploração de funcionalidades associadas a perspetivas, movimentando a posição da câmara (*glm::lookAt*), visando uma melhoria futura de projeto para 3D;
- Análise de um exercício exemplo [4], para melhor compreensão de funcionalidades e aplicações de perspetivas em OpenGL;
- Dando seguimento ao trabalho da semana anterior, complementou-se a redação do relatório.

Na semana de 18/03/2019, fez-se uma implementação de uma peça de *Tetris* e aplicaram-se os conceitos de translação e rotação, mimetizando mecanismos presentes no jogo. Ao longo da semana, as seguintes tarefas foram desenvolvidas:

- Modelação da peça com o formato "L" para a aplicação de conceitos de rotação e translação;

- Associação de *input* de utilizador a movimentos de rotação e translação presentes no *Tetris*. Um exemplo desta associação é o clique da seta para cima promover uma rotação de 90 graus no sentido contrário ao dos ponteiros do relógio;
- Ajuste de posições associadas a rotações nas fronteiras laterais do ecrã, de modo a evitar que a peça rode para fora da janela de visualização;
- Aplicação de queda constante da peça, ou seja, a cada segundo é feita uma translação no eixo vertical, promovendo o efeito de queda de peça que é visível no jogo *Tetris*. A velocidade de descida pode ser incrementada por *input* do utilizador, clicando na seta para baixo, mimetizando o mecanismo presente no jogo;
- Registo de mecanismos de colisão, com auxílio de uma matriz que regista as posições ocupadas pelas peças, de modo a garantir que uma peça não se sobrepõe a outras e não desce para além do ecrã de jogo;
- Dando seguimento ao trabalho das semanas anteriores, complementou-se a redação do relatório.

Na semana de 25/03/2019

- Término de separação de componentes;
- Tentativa de introdução de novas peças;
- Dando seguimento ao trabalho das semanas anteriores, complementou-se a redação do relatório.

Na semana de 01/04/2019

- Reestruturação de código para melhor refatoração de código;
- Continuação e finalização de implementação de novas peças;
- Dando seguimento ao trabalho das semanas anteriores, complementou-se a redação do relatório.

Na semana de 08/04/2019

- Desenvolvimento de gerador de peça aleatória;
- Tipo de peça gerada não é genérico (Resolver problema);
- Dando seguimento ao trabalho das semanas anteriores, complementou-se a redação do relatório.

Na semana de 15/04/2019

- Testes com incorporação de diferentes peças e manipulação de uma outra;
- Dando seguimento ao trabalho das semanas anteriores, complementou-se a redação do relatório.

Capítulo 4

Descrição do Funcionamento do Software

O trabalho desenvolvido até à data da 2ª entrega permite ao utilizador manipular peças de *Tetris*, contemplando mecanismo de translação e rotação, sendo estas movidas até colidirem com algo da grelha de jogo (outras peças) ou até colidirem com a base deste. À colisão de uma peça, será avaliada a possibilidade de eliminação de linhas, sendo eliminadas caso se justifique. A eliminação de linhas segue a lógica utilizada no jogo tradicional de *Tetris*, ou seja, caso uma linha esteja completa, ao longo da largura do ecrã de jogo, esta será eliminada e o conteúdo da linha imediatamente superior a esta substituirá a linha eliminada. Aquando da colisão, será também avaliado se o jogo terminou (peça colidida excede a altura definida para a grelha de jogo); caso não tenha terminado então será criada uma nova peça, de forma aleatória, de entre as sete possiveís peças de jogo.

Para construir cada peça, primeiramente definiram-se quais as características associadas a cada uma, nomeadamente qual a aparência desta, a altura, largura, centro de massa sob o qual mecanismos de rotação serão aplicados, entre outros. A aparência de cada peça segue as do jogo *Tetris* tradicional, tendo sido realizados *mockups* das mesmas, presente na figura 4.1, para melhor discriminação dos atributos associados a cada; as cores escolhidas tiveram como referência o jogo de *Tetris* do site, `https://tetris.com/play-tetris`. Todos os atributos referidos anteriormente estão incorporados em classes destinadas a cada peça, tendo sido criada uma para cada peça, classes essas que herdaram atributos de uma outra, `Peça`, mais genérica e que contém atributos e métodos comuns a todas as peças do jogo *Tetris*. Mais detalhe será dado ao longo deste capítulo sobre este tema.

A criação de peças passa pela definição dos seus vértices e cores associadas, ambos presentes em *buffers* próprios, à semelhança do apresentado ao longo das aulas de CG; estes *buffers* são próprios de cada classe da peça e diferem entre peças. Como é visível na figura 4.1, todas as peças são constituídas por 4 blocos, dispostos de forma diferente. Visando a representação de peças por recurso a conjuntos de triângulos, optou-se por construir cada peça bloco a bloco, sendo cada bloco constituído por 2 triângulos. A figura 4.2 exemplifica a criação de uma peça de *Tetris* (peça Z), sendo

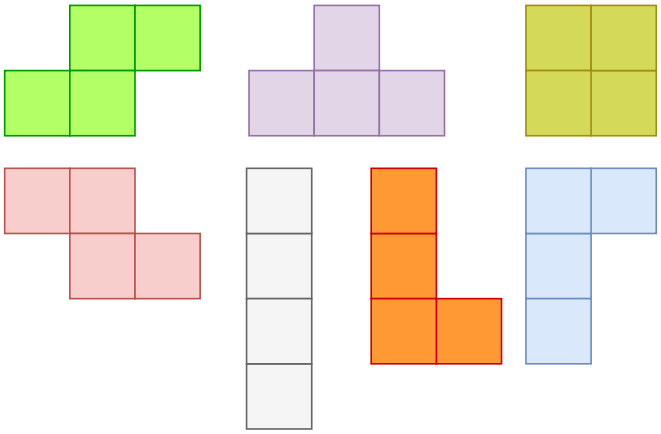

Figura 4.1: *Mockups* de peças criadas.

observável a construção de blocos, triângulo a triângulo, criados pela ordem exibida na imagem. A diferença de coloração entre os triângulos serve para ilustrar qual o triângulo a ser desenhado, em cada momento. De referir que para a construção dos blocos constituintes da peça foi necessário, previamente, definir quais os vértices essenciais para a construção da mesma, sendo estes incorporados num *buffer* de vértices; esta incorporação dos vértices em *buffer*, bem como as cores associadas a cada vértice, num outro *buffer*, não estão representados na imagem.

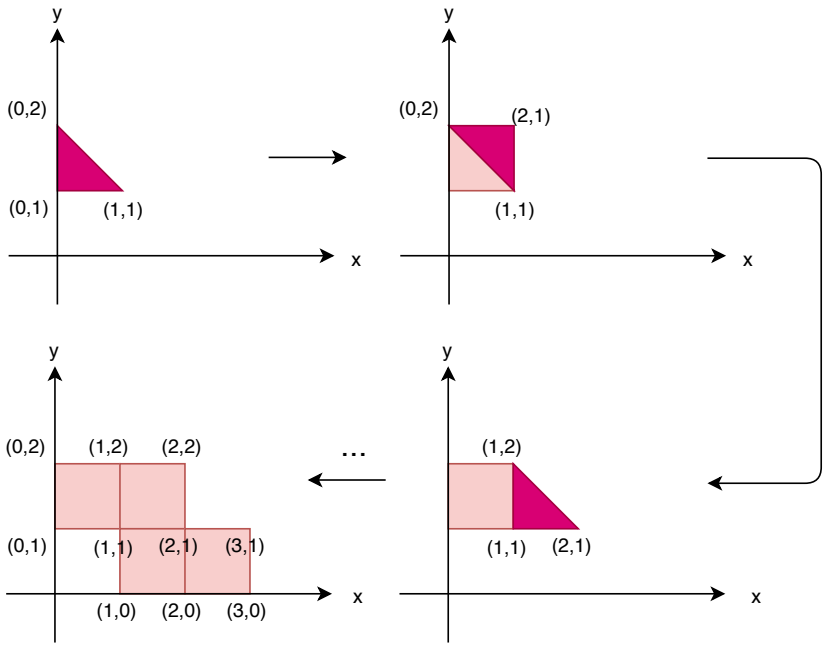

Figura 4.2: Criação de peça.

A possibilidade de jogar *Tetris* necessita da criação e instanciação de peças diferentes mas existem outros fatores a ter em conta, para poder ter um jogo de *Tetris* funcional, nomeadamente:

- Informação de matriz de jogo (matriz de inteiros), importante para determinar colisão da peça;

- Informação de peças anteriormente jogadas e sua posição na grelha de jogo (presente em `vertexBufferTot` e `colorBufferTot`, que armazenam informação de peças anteriormente jogadas), importante para garantir coerência entre os estados de jogo.

A figura 4.3 representa um exemplo de jogo de *Tetris*, exibindo a matriz de colisão associada e a representação visual da grelha de jogo, associada ao desenho de todos os vértices das peças anteriormente jogadas (bem como as cores associadas a cada um). A matriz de colisão, representada por uma matriz de inteiros, contém 1's nas posições ocupadas por peças na grelha de jogo e 0's nas restantes posições. A pertinência desta matriz é a de verificar quando uma peça colide com a base da grelha de jogo ou quando colide com outra peças (1's presentes nesta matriz), anteriormente jogadas. A existência de reticências na matriz indica que as restantes linhas e colunas são preenchidas com zeros. A grelha de jogo é a representação das peças anteriormente jogadas, mantendo coerência entre diferentes estados de jogo. Realça-se a concordância entre os 1's preenchidos na matriz e a posição das peças na grelha de jogo.

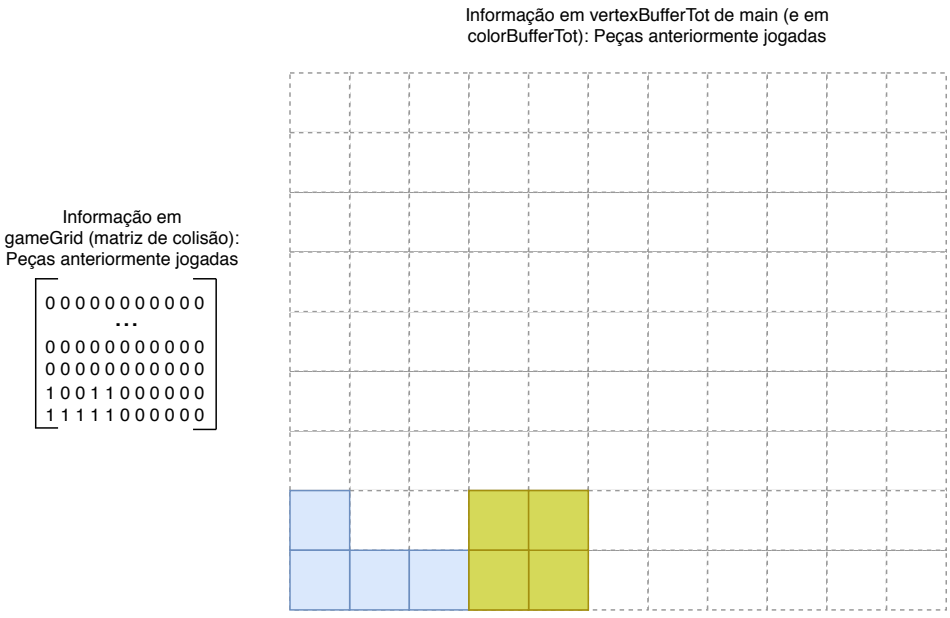

Figura 4.3: Matriz de colisão e grelha de jogo, respetivamente.

O estado de jogo representado na figura 4.3 é um bom exemplo para demonstrar a instanciação de novas peças no jogo. Para poder jogar a próxima peça, esta terá que ser aleatoriamente gerada e introduzida na grelha de jogo; o processo de geração aleatória será explicado mais tarde, neste capítulo. Para instanciar uma peça, devemos invocar o construtor associado à classe da mesma. Aquando desta invocação, a classe criará um *buffer* de vértices (`vertexBuffer`) e um *buffer* de cores (associadas a cada vértice), sendo definidos os seus vértices o mais próximo da origem [(x,y) = (0,0)], por efeitos de comodidade, como exibido na figura 4.4. Obviamente, não poderíamos representar esta peça de forma imediata na grelha de jogo pois resultaria na peça a ser incorporada no canto inferior esquerdo da grelha, derivado da definição de vértices próximos da origem. Assim, para poder representar a peça na grelha de jogo teremos que transladar esta para a posição horizontalmente central e verticalmente no topo.

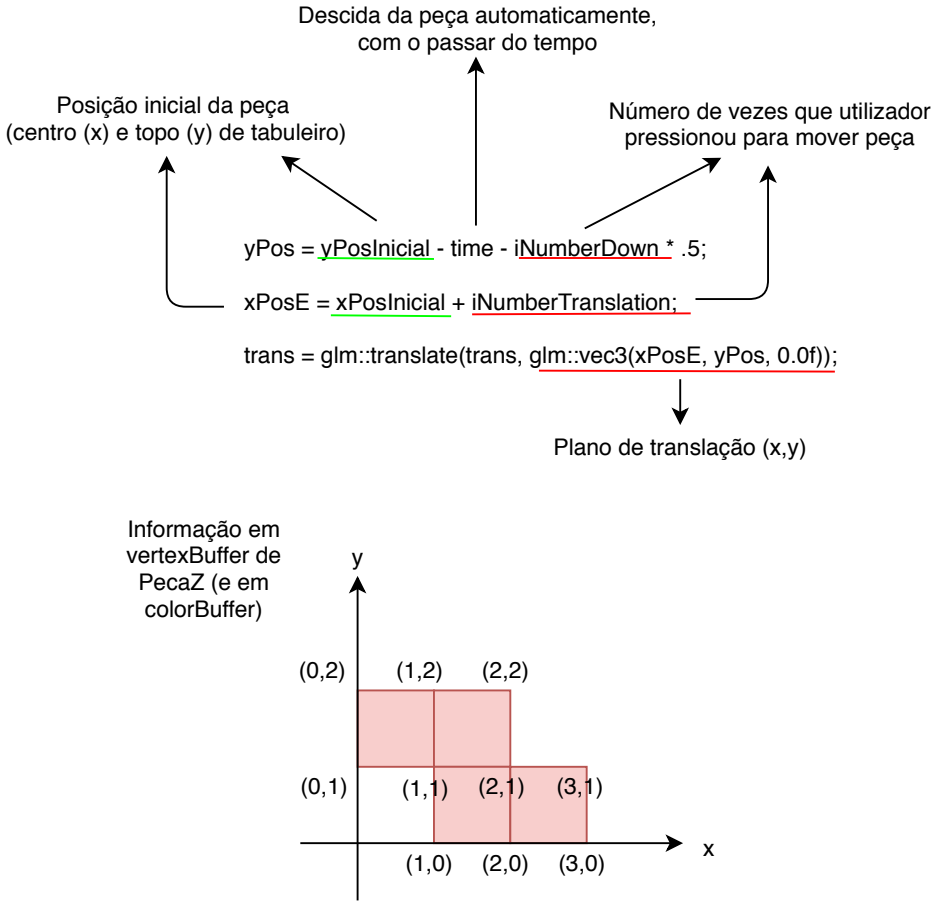

Figura 4.4: Operações associadas à translação da peça e definição de `vertexBuffer` e `colorBuffer` da mesma, respetivamente.

A transladação é representada pelo excerto de código, presente na figura 4.4, onde a posição vertical da peça (`yPos`) é influenciada pela posição inicial desta (topo da grelha de jogo), pelo tempo decorrido (desde a instanciação da peça) e pelo número de vezes que o utilizador pretendeu que a peça descesse mais rápido (`iNumberDown`). O tempo é para garantir que a peça, de forma automática, descerá, verticalmente, ao longo da grelha de jogo, e a variável associada ao registo de *inputs* de utilizador (`iNumberDown`) está multiplicada por um fator de 0.5, visando uma melhor jogabilidade. A posição horizontal da peça depende da sua posição inicial (situada a meio da grelha de jogo) e do número de vezes que o utlizador pretendeu que esta se movesse (`iNumberTranslation`); esta variável, caso seja para mover a peça para a esquerda, será decrementada e, caso se queria mover para a direita, incrementada. Com o registo destas variáveis é possível transladar a peça para o local desejado, por recurso a `glm::translate`, registando o resultado final em `trans`. Esta matriz será usada para multiplicação matricial, no *vertexShader*, visando a transladação das peças 4.1.

Listing 4.1: Diferença de tratamento de vértices entre peça atual e jogadas anteriormente.

```
#version 330 core
...

uniform mat4 mvp;
uniform mat4 rot;
uniform mat4 trans;
uniform bool bPreviousPieces;

void main(){

  gl_Position = mvp * trans * rot * vec4(vertexPosition, 1.0);
  // Previously played pieces
  if (bPreviousPieces) {
    gl_Position = mvp * vec4(vertexPosition, 1.0);
  }
  ...
}
```

Com a instanciação da peça a jogar, podemos desenhar os *buffers* (`vertexBuffer` e `colorBuffer`, da peça, e `vertexBufferTot` e `colorBufferTot`, de todas as peças jogadas anteriormente). Estes serão desenhados pelo *shader*, com atributo 0 associado aos vértices e atributo 1 associado às cores. Para permitir que, apenas o produto matricial entre a matriz de translação (`trans`, na figura 4.2) e matriz de rotação (`rot`, na figura 4.4) é feito com cada vértice de *buffer* da peça em jogo (e não das anteriormente jogadas, presentes na grelha de jogo), foi usada uma variável booleana, visível no excerto 4.1. Esta implementação tem particular importância para garantir que os mecanismos de rotação e translação, impostos pelo utilizador,

apenas afetam a peça atualmente em jogo e não as anteriormente jogadas. O resultado da translação da peça instanciada para o topo da grelha de jogos, em posição central, encontra-se na figura 4.5.

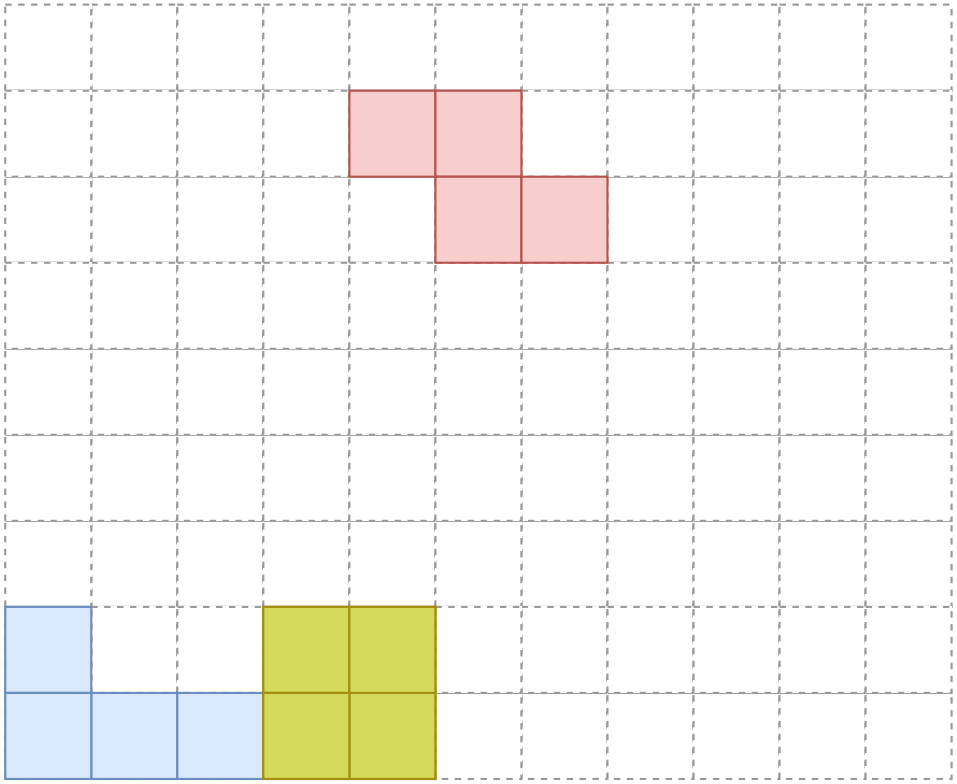

Figura 4.5: Grelha de jogo com representação de peça na sua posição inicial.

A grelha de jogo apresentada é concordante com o observado no *Tetris* tradicional, aquando de instanciação de nova peça, tendo a sua representação sido simplificada por questões de apresentação em relatório; a sua largura é concordante com a de jogo mas a altura está reduzida. Instanciada a peça, esta pode ser transladada e rodada pelo utilizador. Para registar estes *inputs* do utilizador, eventos associados ao clique em determinadas teclas foram captados, presentes numa função própria, na classe `main.cpp`. Naturalmente, mecanismos de translação e rotação de peças deverão ter um cuidado particular, visando a permanência da peça dentro da grelha de jogo. A título de exemplo, o pressionar de tecla correspondente à translação da peça para a esquerda não deverá promover a saída desta da grelha de jogo. Assim, medidas de segurança terão de ser implementadas ao nível da translação, presentes no excerto 4.2.

Listing 4.2: Captação de *inputs* de utilizador para aplicação de mecanismos de translação e rotação à peça.

```
void registerUserInputs(Peca& plPeca) {

  if (glfwGetKey(window, GLFW_KEY_UP) == GLFW_PRESS) {
    if (glfwGetKey(window, GLFW_KEY_UP) == GLFW_RELEASE) {
      plPeca.incNumberRotate();
    }
  }
  if (glfwGetKey(window, GLFW_KEY_RIGHT) == GLFW_PRESS) {
    if (glfwGetKey(window, GLFW_KEY_RIGHT) == GLFW_RELEASE) {
      if (plPeca.getXPosD() < iWidth) {
        plPeca.incNumberTranslation();
      }
    }
  }
  if (glfwGetKey(window, GLFW_KEY_LEFT) == GLFW_PRESS) {
    if (glfwGetKey(window, GLFW_KEY_LEFT) == GLFW_RELEASE) {
      if (plPeca.getXPosE() > 0) {
        plPeca.decNumberTranslation();
      }
    }
  }
  if (glfwGetKey(window, GLFW_KEY_DOWN) == GLFW_PRESS) {
    plPeca.incNumberDown();
  }
}
```

Um exemplo de mecanismo de segurança é a linha `if (plPeca.getXPosD() < iWidth)`, onde `iWidth` é a largura atribuída à janela de visualização, em que, caso o utilizador clique no botão "seta direita", não se verificará incremento da variável de translações, presente dentro do objeto `Peca`. A classe `Peca` é uma classe abstrata, importante para que a geração aleatória de diferentes peças seja possível. Todas as diferentes peças (cada uma com sua classe), implementam a interface desta classe, utilizando para tal a nomenclatura `class PecaZ : public Peca`, no ficheiro *hpp*, correspondente; a classe exemplificada é a da peça Z, presente no exemplo das figuras acima.

À semelhança da translação, o mecanismo de rotação de peças é capturado por *inputs* de utilizador, também visível no excerto 4.2, nomeadamente na linha `plPeca.incNumberRotate();`. A abordagem para imprimir rotação na peça implica uma translação para origem, rotação sobre o seu Centro de Rotação (CR) e nova translação para o local anterior, CR unidades. De forma esquemática a figura 4.6 exibe o comportamento da peça aquando de uma rotação de 90º.

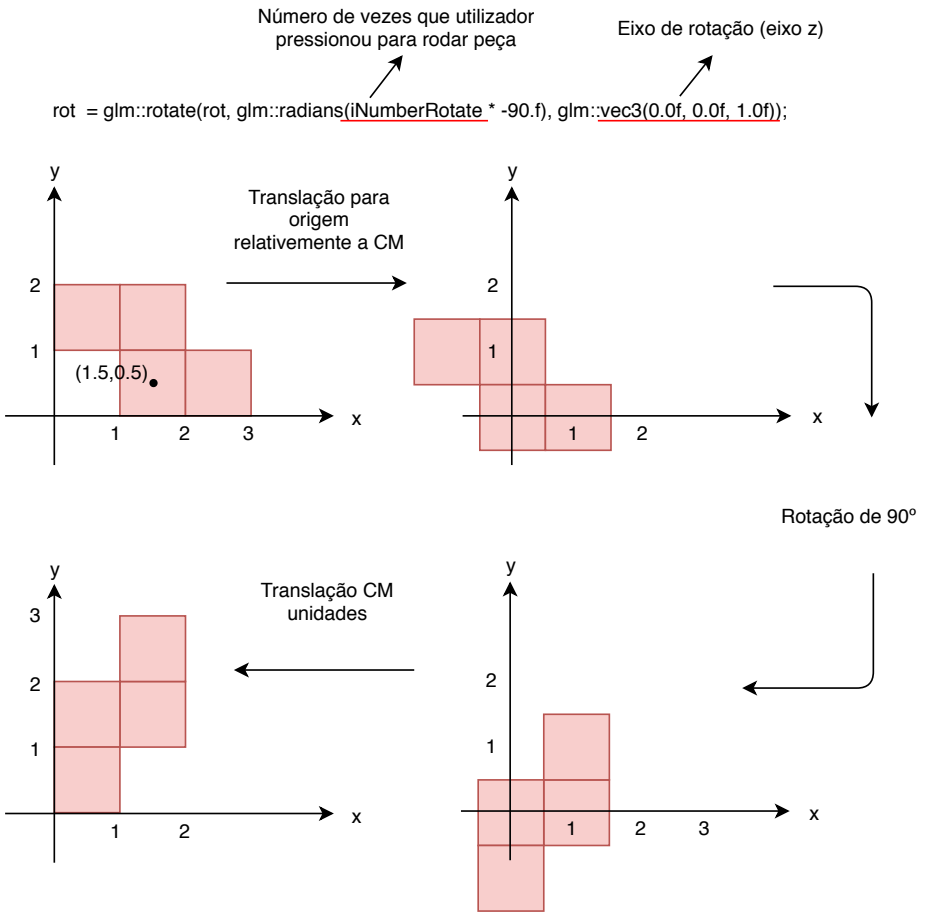

Figura 4.6: Operações de rotação associadas à peça.

Presente na figura 4.6 é visível a matriz `rot`, responsável pela mutiplicação matricial nos *shaders* (excerto 4.1), representando o mecanismo de rotação imprimido às peças. No excerto de código visível na figura podemos observar que a rotação das peças é feita no eixo Z e roda em incrementos de 90º, influenciado pelo número de vezes que o utilizador quis que esta rodasse (`iNumberRotate`). A esta mesma matriz, `rot`, é também aplicada translações, como evidenciado pelo esquema abaixo do excerto de código na figura 4.6, não tendo sendo exibido o código correspondente na figura, para efeitos de simplicidade. Naturalmente, no mecanismo de rotação também se tem que ter um cuidado particular, nomeadamente aquando da colisão com extremidades da grelha de jogo, fazendo acertos na posição da peça sempre que necessário, visando manter esta dentro da grelha de jogo. Mecanismos de segurança que visam cumprir o objetivo descrito, encontram-se presentes no excerto 4.3, uma função presente dentro da classe de cada uma das diferentes peças de jogo.

Listing 4.3: Exemplo de ajustes de posição da peça (neste caso a peça Z), aquando da rotação da mesma, visando mantê-la dentro da grelha de jogo. Estes ajustes são influenciados pela forma da peça, resultante da rotação desta.

```
void PecaZ::atualizaPos() {
  switch (iNumberRotate % 4) {
    ...
    case 2:
      // Tamanho da peca
      iPieceWidth = 3;
      iPieceHeight = 2;

      // Ajuste de posicoes
      xPosD = xPosE + iPieceWidth;
      yPos--;

      if (xPosE < 0) {
        // Garantir que peca se mantem dentro da janela
        // de visualizacao
        iNumberTranslation++;
        // Reajustar posicao da peca resultante de ajuste
        xPosE++;
        xPosD++;
      }
      break;
    ...
  }
}
```

O excerto 4.3 evidencia um exemplo de ajuste de posição da peça Z, aquando da implementação de rotação. Mediante as rotações impostas pelo utilizador, a peça pode assumir 4 posições possíveis. No caso de rodar 90 graus duas vezes, no sentido contrário dos ponteiros do relógio (`case 2`), a posição `y` terá que ser ajustada, bem como a delimitação da peça, a sua largura e altura. A chamada de atenção para este caso, neste método, é caso a posição à esquerda da peça seja inferior a 0 (`xPosE < 0`, o que significa que a peça deixou de ser totalmente visível na janela), então teremos que mover a peça para a direita (`iNumberTranslation++`), e consequentemente alterar a posição do lado esquerdo e direito da peça (`xPosE++` e `xPosD++`, respetivamente); esta situação ocorre quando a peça está junto ao lado esquerdo do ecrã e o utilizador tenta rodar a peça. Note-se que o incremento em `iNumberTranslation`, por exemplo, simula "empurrar"a peça para a direita, visando manter a peça dentro da janela de visualização em todos os momentos do jogo. Note-se também que, independentemente de situações de erro (caso `if (xPosE < 0)`), a peça terá de ser atualizada aquando da sua rotação, nomeadamente nas variáveis `xPosD` e `yPos`, como é visível no excerto 4.3. A pertinência destas variáveis é a deteção de colisão, e ga-

rantia que a peça se mantém dentro da grelha de jogo; `xPosD` influencia translação da peça, como visível no excerto 4.2, no método `plPeca.getXPosD()`.

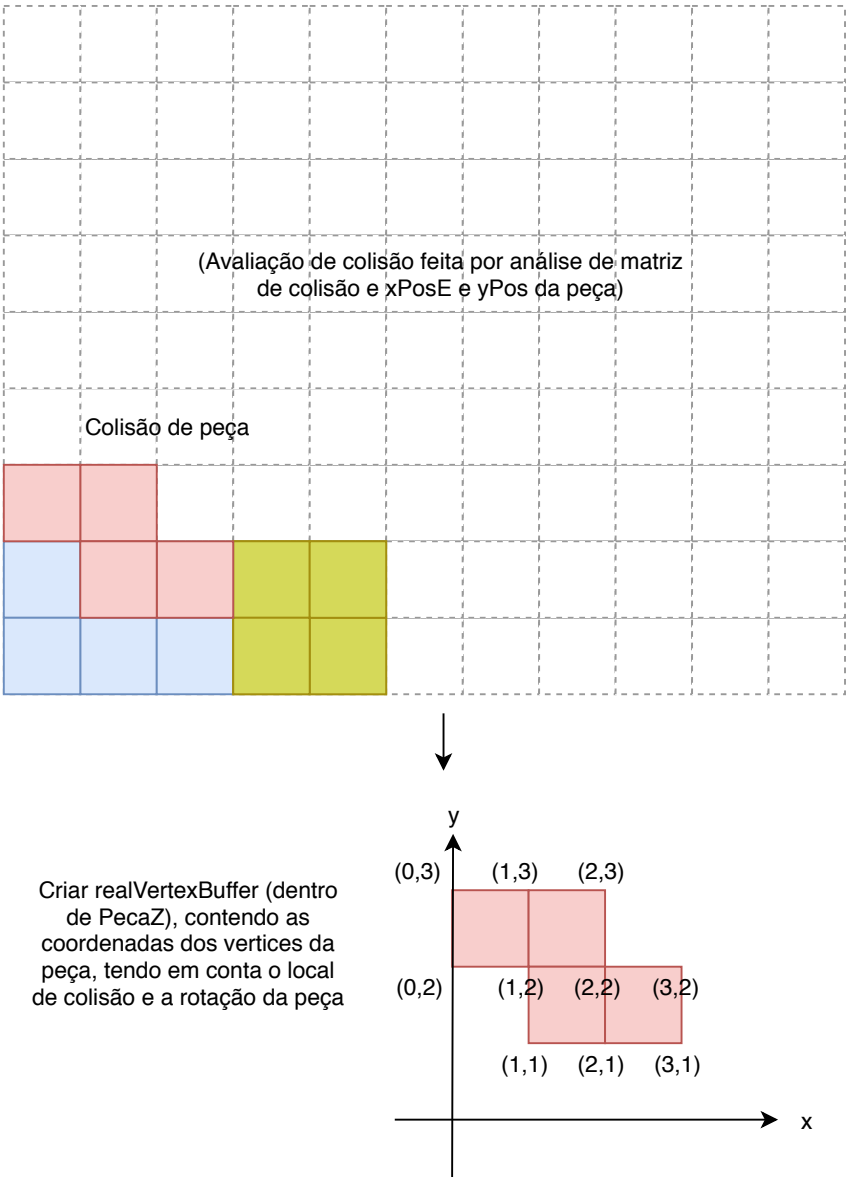

Figura 4.7: Colisão de peça e consequente criação de vertexbuffer, considerando a posição da peça e a sua rotação no momento da colisão.

Tendo como referência a figura 4.5, e aplicando mecanismos de translação e rotação à peça, poderíamos colidir a peça Z de tal modo que obteríamos a grelha de jogo presente na figura 4.7. Tal como referido na figura, a colisão da peça é feita por

recurso à posição desta na grelha em conjunto com a matriz de colisão, responsável por identificar as posições ocupadas, por peças anteriormente jogadas, na grelha. A posição da peça na grelha é conseguida usando as variáveis `xPosE` e `yPos`, que correspondem à posição esquerda e inferior da peça, respetivamente; usando esta informação, em conjunto com a sua altura e largura, podemos concluir se a peça colidiu com algumas das anteriormente jogadas. O que a figura 4.7 também evidencia é a criação de um *buffer* de vértices (`realVertexBuffer`), que contém as posições reais da peça, tendo como referência as rotações impostas a esta e o local de colisão. Este *buffer* de vértices difere do exibido na figura 4.4, pela razão que este contém os seus vértices de acordo com o local de colisão (e rotação imposta, embora neste caso seja idêntica à de 4.4). A pertinência de criar um novo vértice de *buffers*, com as considerações referidas, é para que este seja incorporado no *buffer* de vértices de todas as outras peças anteriormente jogadas (`vertexBufferTot`), presente em `main.cpp`; de forma análoga se tratará o *buffer* correspondente às cores dos vértices. A figura 4.8 representa, esquematicamente, o referido.

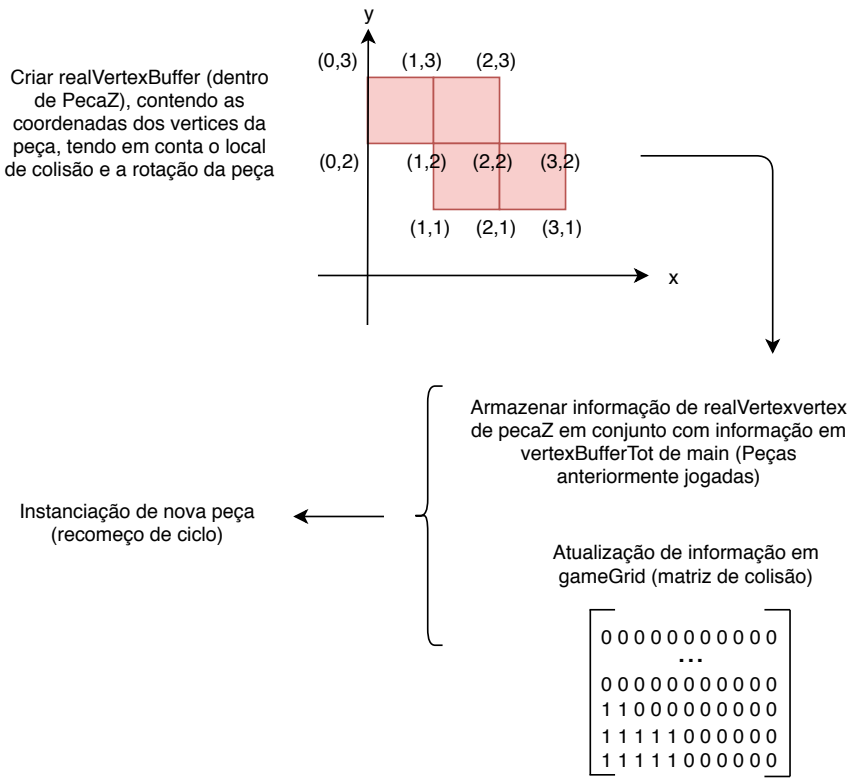

Figura 4.8: Atualização da informação da grelha de jogo (vertexBufferTot) e matriz de colisão, resultantes da colisão da peça.

A figura 4.8 representa também o efeito que a colisão entre peças terá na matriz de colisão. Realça-se o facto de esta ter 1's concordantes com a situação verificada

com a grelha de jogo, aquando da colisão, de forma análoga à grelha evidenciada na figura 4.3. Esta matriz apenas será atualizada nestas circunstâncias (colisão de peças) e terá como referência o local de colisão e a aparência da peça (altura, largura, forma resultante da rotação imposta pelo utilizador), aquando do preenchimento de 1's. A ocorrência de colisão pode promover a limpeza de linhas, caso se justifique, ou o término de jogo. Como o exemplo não contempla nenhuma das situações, o percurso natural é a instanciação de nova peça (recomeço de ciclo), como evidenciado na figura 4.8. Um esquema geral das figuras apresentadas será incorporado em Anexo, visando uma melhor compreensão do fluxo de acontecimentos.

A instanciação de novas peças exige, implicitamente, que exista um mecanismo responsável pela geração de novas peças. À semelhança do que ocorre num jogo de *Tetris* tradicional, o projeto desenvolvido contempla uma instanciação aleatória de peças, de entre as criadas (visíveis na figura 4.1). A abordagem escolhida para incorporar este mecanismo foi a criação de uma classe, denominada `GeradorPecas`, destinada a criar as diferentes peças, invocando os construtores respetivos, e devolver estas ao programa principal (`main.cpp`). A definição de qual a peça a instanciar foi feita por recurso à avaliação de um valor aleatório, variável entre 1 e 7, concordante com o número de peças disponíveis; cada peça criada estará associada a um número e será instanciada caso o valor aleatoriamente gerado seja o correspondente a esta. Para que as diferentes peças criadas, cada uma com uma classe correspondente, possam ser devolvidas da mesma forma pelo `GeradorPecas`, teve que ser implementada uma classe abstrata, `Peca`, que será o tipo devolvido por `GeradorPecas`. Esta classe abstrata conterá os métodos comuns a todas as classes de peças, e estas implementarão a interface de `Peca`, permitindo a existência de um tipo comum entre classes de peças; a implementação de interface é feita por recurso à nomenclatura `class PecaZ : public Peca` (a título de exemplo), no ficheiro *hpp* da peça.

Listing 4.4: Eliminação de linhas da grelha de jogo.

```
void eliminaLinha(int iLinha) {
  int i;
  newSize = g_vertex_buffer_dataTot.size();
  for (i = 0; i < newSize; i += 18){
    // Limpar bloco a bloco
    if (g_vertex_buffer_dataTot.at(i+1) == iLinha){
      for (int k = i; k < g_vertex_buffer_dataTot.size() - 18; k++)
      {
        g_vertex_buffer_dataTot.at(k) =
            g_vertex_buffer_dataTot.at(k + 18);
        g_color_buffer_dataTot.at(k) =
            g_color_buffer_dataTot.at(k + 18);
      }
      // Atualizacao de variaveis
      newSize -= 18;
      i -= 18;
```

```
    }
    else {
      if (g_vertex_buffer_dataTot.at(i + 1) > iLinha) {
        for (int j = 1; j < 18; j += 3){
          // Atualizar altura (y) de todos os vertices
          // de blocos superiores a linha a eliminar
          g_vertex_buffer_dataTot.at(i + j) =
              g_vertex_buffer_dataTot.at(i + j) - 1;
        }
      }
    }
  }
}
```

A eliminação de linhas é outra implementação necessária para que o projeto desenvolvido tenha a jogabilidade comumente associada ao *Tetris* tradicional. Uma linha é eliminada se todos os blocos dessa linha preencherem, em largura, a grelha de jogo. Seguindo esta abordagem, o excerto 4.4 demonstra a função, em `main.cpp`, responsável pela implementação deste mecanismo. A função no excerto de código recebe o identificador da linha a ser eliminada e regista o tamanho atual do *buffer* de vértices das peças presentes na grelha na variável `newSize`. O incremento da variável `i` em 18, a cada iteração do ciclo, justifica-se pela abordagem de ser analisado bloco a bloco de cada peça, verificando se este pertence à linha a eliminar; cada bloco é constituído por 2 triângulos (como exibido na figura 4.2), cada triângulo contém 3 vértices e cada vértice contém 3 coordenadas (x,y e z), daí o incremento ser 18 unidades. Caso um bloco tenha a coordenada y, do seu primeiro vértice, igual à linha a eliminar, proceder-se-á à substituição deste bloco pelo seguinte e assim sucessivamente até ter percorrido todo o *buffer*; esta substituição ocorre tanto no *buffer* de vértices como no de cor. Desta substituição resulta a atualização das variáveis `i` e `newSize`, decrementando-as em 18. A variável `newSize` tem particular importância pois a avaliação de vértices apenas é feita de 0 até o valor desta variável, evitando análise desnecessária de blocos; será esta a variável responsável por ajustar o tamanho dos *buffers* de vértices e cor, no final do processo de eliminação de linhas (não visível no excerto 4.4). Implicitamente associada à eliminação de linhas estará o procedimento de atualização de todas as linhas superiores a esta, decrementando a sua posição vertical em uma unidade, para garantir coerência de jogo. Esta atualização encontra-se também presente no excerto 4.4, nomeadamente no caso `else`. O que esta porção de código produz é a atualização de todas as coordenadas y do bloco avaliado nesta iteração de ciclo, reduzindo os seus valores em 1 unidade; o *buffer* de cor não necessita de ser alterado neste caso.

A eliminação de linhas é invocada a cada colisão de peça, caso a análise da matriz de colisão assim o indique. Esta matriz será avaliada para verificar se existem linhas com apenas 1's, concordantes com a grelha de jogo ter uma linha totalmente ocupada, na sua largura, por blocos de peças, e serão eliminadas tantas linhas quan-

tas as que tiverem esta caraterística. Naturalmente, após a eliminação de linhas (excerto 4.4), a matriz de colisão também será atualizada, visando uma consistência entre a grelha de jogo e esta matriz, aquando da instanciação da próxima peça.

Capítulo 5

Trabalhos Futuros

A próxima fase de desenvolvimento do projeto passará pela introdução de textura em peças, visando uma melhoria no aspeto visual do jogo, substituíndo as cores de cada peça. Uma implementação que, não sendo necessária para jogar *Tetris*, mas que facilita a sua jogabilidade e promove uma melhor experiência de jogo para o utilizador é a exibição de próxima peça a ser instanciada. A incorporação desta funcionalidade passaria pela introdução de Viewports, divindo o ecrã em grelha de jogo e janela de exibição da próxima peça a ser instanciada.

Para uma melhor jogabilidade poder-se-ia implementar um temporizador para que, aquando da colisão de uma peça com outras, o utilizador ainda consiga mover a peça. Como o projeto está desenvolvido atualmente, existe uma certa rigidez, na medida em que a colisão de uma peça, promove imediatamente a instanciação de uma nova, sem capacidade de o utilizador corrigir a posição da peça para uma mais favorável. Esta implementação sugerida não é inédita, estando presente em diversas versões do *Tetris*.

Associada à implementação de eliminação de linhas, presente na atual entrega de projeto, surge a necessidade de ter um mecanismo de contabilização de pontos, resultantes da eliminação de linhas, conferindo um maior propósito ao jogo desenvolvido. De forma natural, também associada à eliminação de linhas e consequente pontuação obtida, surge a necessidade de implementar o aumento de velocidade de queda de peças, simulando a passagem de níveis, caso o utilizador chegue a um determinado patamar de pontuação. Estas implementações apenas fazem sentido após implementações associadas à melhoria de jogabilidade serem incorporadas; caso contrário a dificuldade associada ao jogo desmotivará o utilizador e reduzirá o seu interesse no jogo.

Uma implementação futura, não contemplada no projeto, mas que conferiria maior valor ao trabalho desenvolvido, seria uma evolução do jogo *Tetris* para 3D.

Capítulo 6

Considerações Finais

No capítulo 1 foi apresentada a motivação para o desenvolvimento deste trabalho, referindo justificações para a escolha deste tema, enunciando a importância da matéria apresentada na UC para o desenvolvimento do projeto, tendo sido referido exemplos da sua aplicabilidade neste.

No segundo capítulo foram apresentadas quais as tecnologias usadas, referindo qual a linguagem, API, bibliotecas e software que auxiliaram o desenvolvimento do projeto, indicando qual a sua influência na implementação de funcionalidades neste.

Relativamente ao terceiro capítulo, foi exibido o cronograma das tarefas desenvolvidas, agrupadas por semanas, referindo qual a importância da tarefa para o trabalho final e indicando a bibliografia usada para a realização da mesma.

O capítulo 4 refere, em detalhe, as implementações com maior influência no desempenho das funções projetadas para o jogo *Tetris*, nomeadamente, criação de peças, gestão de mecanismos de colisão, controlo de rotação e translação personalizado para cada tipo de peça e separação de componentes em diferentes ficheiros (separação de programa principal em diferentes ficheiros de classe).

No capítulo imediatamente anterior projetou-se o trabalho a desenvolver no futuro, referindo quais as melhorias a implementar no projeto atual e quais as implementações futuras que terão que ser incorporadas para que este desempenhe as funções para o qual foi projetado.

No momento de entrega deste relatório, o projeto encontra-se em fase de desnvolvimento. O desenho de novas classes, associadas às diferentes peças do *Tetris*, com translações e rotações específicas das mesmas, será o próximo passo a desenvolver.

Toda a componente téorica associada ao OpenGL, foi aplicada no desenvolvimento do projeto, até à data. Com a apresentação de nova matéria na UC, outras funcionalidades de OpenGL serão ensinadas e, sempre que possivel, aplicadas ao trabalho.

Bibliografia

[1] Ed Angel and Dave Shreiner. An Introduction to OpenGL Programming. https://www.cs.unm.edu/~angel/SIGGRAPH13/An%20Introduction%20to%20OpenGL%20Programming.pptx, 2013. SIGGRAPH.

[2] A. Gomes. Computação Gráfica - Geometric Transformations, 2019.

[3] opengl tutorials. Tutorial 2 : The first triangle. http://www.opengl-tutorial.org/beginners-tutorials/tutorial-2-the-first-triangle/, 2018. [Acedido em 07/03/2019].

[4] opengl tutorials. Tutorial 6 : Keyboard and Mouse. http://www.opengl-tutorial.org/beginners-tutorials/tutorial-6-keyboard-and-mouse/, 2018. [Acedido em 13/03/2019].