Tetris 2D

Relatório de Projeto da Unidade Curricular
Computação Gráfica

2 de Junho 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*
MVP	*Model View Projection*
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 ou texturas (*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 um determinado eixo;
- 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;
- Desenho de diferentes peças, cada uma com o seu respetivo *Model View Projection* (MVP), usando partilha de atributos aquando da utilização de *shaders*, para cálculo de posição e coloração das peças.
- Implementação de texturização a peças, conferindo maior realismo a estas.

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. Desenho das diferentes peças de jogo, implementando mecanismos de rotação e translação apenas à atual (mantendo as anteriormente jogadas nas suas posições), bem como a incorporação da limpeza de linhas, aquando do preenchimento em toda a largura da grelha de jogo, foram outros desafios associados a este trabalho que motivaram a sua escolha. 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*, ficheiros estes utilizados 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 [7]. 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;
- ***stb_image.h*:** Um *header* para carregamento de imagens, usado aquando da incorporação de texturas [3];
- ***irrKlang*:** Utilizada para incorporação de áudio no jogo [2].

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 [4]. 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.

A referência *GitHub* fornecidas pelo professor não foi usada como modelo de desenvolvimento do trabalho, relativamente às metodologias utilizadas, tendo o projeto desenvolvido adoptado estratégias próprias, que diferem das usadas na referência fornecida [5]. As texturas usadas neste projeto tiveram como base as usadas num outro projeto *GitHub* [6], embora o ficheiro usado para texturas tenha sido também criado pelo grupo, para melhor aproveitamento de recursos. A referência *GitHub* dividiu as texturas em diferentes ficheiros enquanto que o grupo as concentrou todas num único.

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, exploraram-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 [8], 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.

Durante a semana de 25/03/2019 promoveu-se a continuação das tarefas realizadas na semana anterior, nomeadamente, a separação entre funcionalidades de ficheiro principal e de ficheiro associado a peça. Desenvolvimento de novas peças foi outra das tarefas realizadas nesta semana. Assim, o trabalho realizado dividiu-se nos seguintes pontos:

- Separação de funcionalidades entre ficheiro principal e ficheiros de classes de peças;
- 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 desenvolveu-se o projeto de modo a incorporar novas funcionalidades e permitir uma melhor escalabilidade para implementações futuras. Tendo este objetivo como referência desenvolveram-se as seguintes tarefas ao longo da semana:

- Reestruturação de código para melhor refatoração;
- 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 desenvolveram-se novas funcionalidades visando a jogabilidade de *Tetris*, nomeadamente, incorporação da geração de peças aleatoriamente e definição de tipos e classes associados a esta incorporação. As tarefas desenvolvidas foram:

- Desenvolvimento de classe de gerador de peça aleatória;
- Definição de classe abstrata (interface) de modo a que todas as classes implementadas para as peças tenham um tipo em comum;
- Dando seguimento ao trabalho das semanas anteriores, complementou-se a redação do relatório.

Na semana de 15/04/2019 testaram-se abordagens para instanciação de uma peça, seguida de outra, simulando o mecanismo apresentado no jogo, ou seja, haver peças anteriormente jogadas e uma peça instanciada no momento. Realizaram-se as seguintes tarefas:

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

Durante a semana de 22/04/2019 implementaram-se todas as funcionalidades de base associadas ao jogo *Tetris*, nomeadamente, a instanciação de diferentes peças e eliminação de linhas, sempre que se justifique. Dividiu-se este objetivo nas diferentes tarefas:

- Solidificação de implementação de classe abstrata de peças;
- Incorporação de instanciação de diferentes peças na classe geradora aleatória de peças;
- Desenho diferenciado entre peças anteriormente jogadas e peça atual na grelha de jogo;
- Eliminação de linhas da grelha de jogo e atualização de matriz de colisão associada;
- Dando seguimento ao trabalho das semanas anteriores, complementou-se a redação do relatório.

Na semana de 29/04/2019 implementaram-se funcionalidades visando uma melhor jogabilidade e uma melhoria do aspeto gráfico. A realização destes objetivos discriminaram-se nas seguintes tarefas:

- Incorporação de texturas nas peças de jogo;

- Implementação de temporizador, aquando da colisão de peças, permitindo movimentação e rotação desta durante um determinado período de tempo, visando melhor jogabilidade;
- Melhoria do mecanismo de deteção de colisão de peças;
- Incorporação de mecanismos visando a avaliação de possibilidade de rotação de peça, aquando da sua colisão com outras;
- Dando seguimento ao trabalho das semanas anteriores, complementou-se a redação do relatório.

Na semana de 06/05/2019 continuaram-se a desenvolver funcionalidades associadas à jogabilidade e melhoria do aspecto visual do jogo. Este objetivo foi concluído desempenhando as seguintes tarefas:

- Implementação de mecanismos de cálculo de pontuação e ajuste de níveis, de acordo com as linhas eliminadas;
- Incorporação de mecanismos de armazenamento de peça atual de jogo, usando funções de *callback*;
- Inclusão de imagem de fundo, usando texturas;
- Implementação de queda abrupta de peça, usando funções *callback*;
- Dando seguimento ao trabalho das semanas anteriores, complementou-se a redação do relatório.

Na semana de 13/05/2019 promoveu-se a continuação das tarefas realizadas na semana anterior, nomeadamente, a inclusão de mecanismos que visem uma melhor experiência de jogo. As tarefas desenvolvidas foram:

- Implementação de representação do local de colisão de peças instanciadas, tendo como referência o formato da peça e a sua rotação, por recurso a texturas;
- Adequar tempo de queda de peça, de acordo com o nível de jogo;
- Dando seguimento ao trabalho das semanas anteriores, complementou-se a redação do relatório.

Na semana de 20/05/2019 incorporaram-se aspetos de jogo com o intuito de promover uma melhor experiência de jogo ao utilizador. Nesta semana, dividiram-se os objetivos nas tarefas seguintes:

- Animação da eliminação de linhas;
- Incorporação de menu de Pausa e Controlos;

- Dando seguimento ao trabalho das semanas anteriores, complementou-se a redação do relatório.

Na semana de 27/05/2019 incorporaram-se as funcionalidades finais do projeto, com o intuito de cumprir todos os requisitos propostos pelo enunciado. A realização destes objetivos discriminaram-se nas seguintes tarefas:

- Incorporação de sons;
- Leitura de *inputs* do rato;
- Término da redação do relatório.

Capítulo 4

Descrição do Funcionamento do Software

O trabalho desenvolvido permite ao utilizador manipular peças de *Tetris*, contemplando mecanismos de translação e rotação, sendo estas movidas até colidirem com algo da grelha de jogo (outras peças) ou até colidirem com a base desta. 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. O restante capítulo será distribuído por secções, onde será explicado como cada componente do trabalho foi implementada, sendo facultados excertos de código e/ou imagens, visando uma melhor compreensão das implementações incorporadas.

4.1 Construção de Peças

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 outras características. A aparência de cada peça segue as do jogo *Tetris* tradicional, tendo sido realizados *mockups* das mesmas, presentes 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.

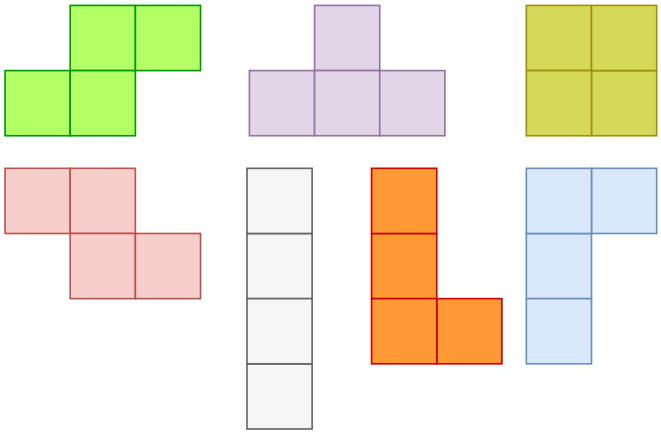


Figura 4.1: *Mockups* de peças criadas.

A criação de peças passa pela definição dos seus vértices e texturas 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 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 texturas associadas a cada vértice, num outro *buffer*, não estão representados na imagem.

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 `texturebufferTot`, que armazenam informação de peças anteriormente jogadas), importante para garantir coerência entre os estados de jogo.

Tendo em consideração os fatores mencionados, a secção seguinte vem explicar o modo de representação de peças no jogo, referindo a correlação entre as peças representadas com a matriz de colisão.

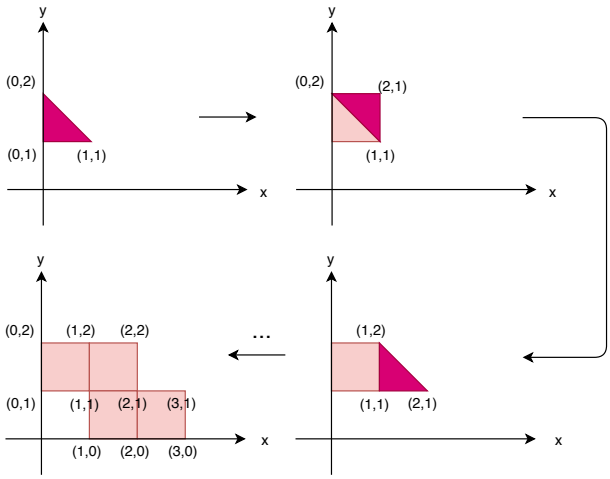


Figura 4.2: Criação de peça.

4.2 Representação de Peças em Grelha de Jogo

Um exemplo de jogo de *Tetris* pode ser visto na figura 4.3, onde é exibida a matriz de colisão e a representação visual da grelha de jogo, associada ao desenho de todos os vértices das peças anteriormente jogadas (bem como as texturas 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.

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, em seção própria. 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 texturas (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.

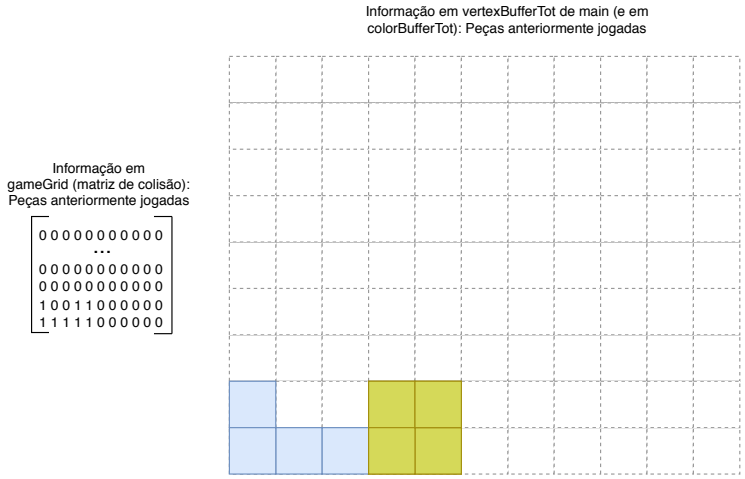


Figura 4.3: Matriz de colisão e grelha de jogo, respetivamente.

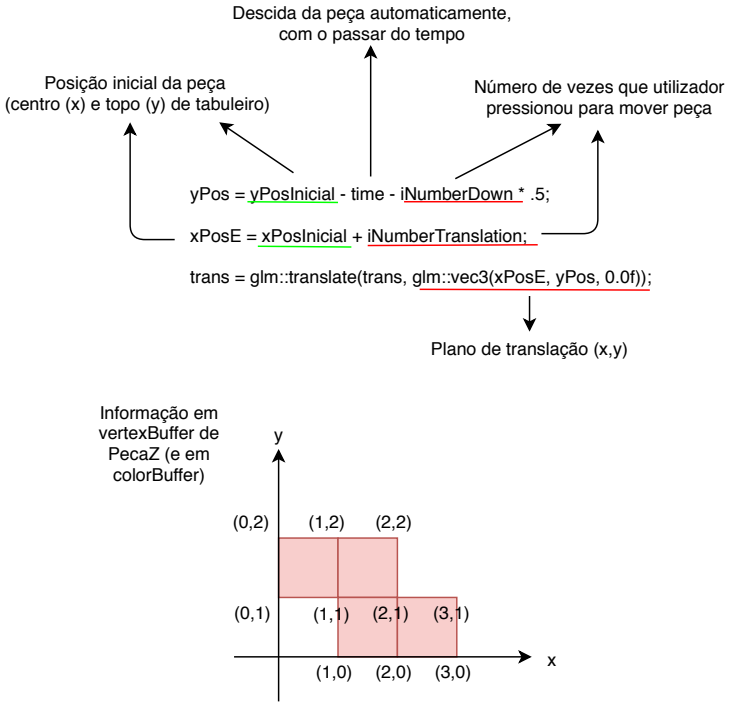


Figura 4.4: Operações associadas à translação da peça e definição de `vertexBuffer` e `colorBuffer` da mesma, respetivamente.

Os mecanismos envolvidos na translação e rotação de peças têm assim um papel preponderante para a correta representação de peças na grelha de jogo sendo, por isso, mais detalhadamente explicadas na secção seguinte.

4.3 Translação e Rotação de Peça

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 queira 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 representadas na figura 4.1.

```
bool drawCurrentObject(Peca& pPeca){
  ...
  pPeca.rotacaoPeca(rot);
  pPeca.translacaoPecaContorno(trans);
  glm::mat4 MVP_PecaColisao = Projection * View * trans * rot;
  glUniformMatrix4fv(MVP, 1, GL_FALSE, &MVP_PecaColisao[0][0]);
  ...
}
...
void drawObjects(int iIdentificador) {
  ...
  glm::mat4 MVP_Matrix = Projection * View;
  glUniformMatrix4fv(MVP, 1, GL_FALSE, &MVP_Matrix[0][0]);
  ...
}
```

Listing 4.1: Diferença de MVP entre peça atual (`drawCurrentObject`) e jogadas anteriormente (`drawObjects`).

Com a instanciação da peça a jogar, podemos desenhar os *buffers* (`vertexBuffer` e `textureBuffer`, da peça atual, e `vertexBufferTot` e `textureBufferTot`, de todas as peças jogadas anteriormente). Estes serão desenhados pelo *vertexshader*, com atributo 0 associado aos vértices e atributo 1 associado às texturas; no caso de texturas, *vertexshader* irá encaminhar estes para o *fragmentshader*. Para garantir 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 dividido o cálculo matricial de MVP em duas funções distintas, 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. De referir que, em ambos os casos, a variável conhecida pelo *vertexshader* associada a MVP é a mesma (nome `MVP`). O resultado da translação da peça instanciada (presente na figura 4.4) para o topo da grelha de jogos, em posição central, encontra-se na figura 4.5.

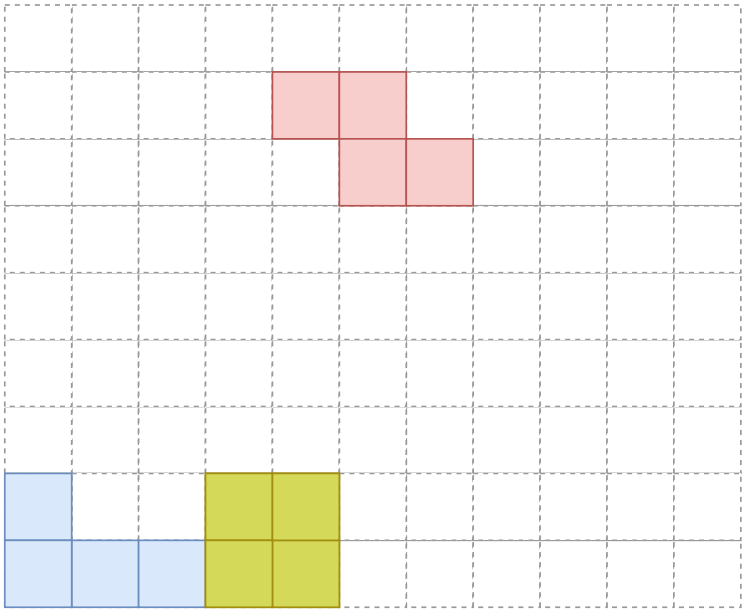


Figura 4.5: Grelha de jogo com representação de peça na sua posição inicial.

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.

```
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();
  }
  ...
}
```

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

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, seguindo como referência o apresentado nas aulas práticas [9]. 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.

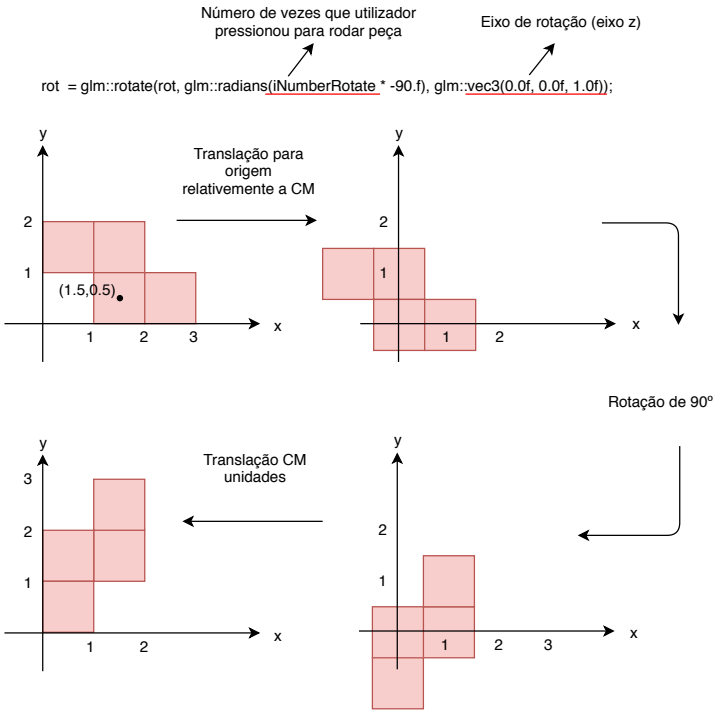


Figura 4.6: Operações de rotação associadas à peça.

Presente na figura 4.6 é visível a matriz `rot`, responsável pela mutiplicação matricial nos *shaders* (referido, aquando da exibição do 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; código correspondente a translações da figura não foi exibido, para efeitos de simplicidade.

Naturalmente, no mecanismo de rotação também será necessárioum 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. O excerto referido evidencia um exemplo de ajuste de posição da peça Z, aquando da implementação de rotação.

```
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;
    ...
  }
}
```

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.

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 o "empurrar"da 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 garantia 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()`.

Seguindo a lógica implementada para translação e rotação de peças, usando *inputs* de teclado do utilizador (excerto 4.2), e face aos requisitos do enunciado, foram também implementados *inputs* de rato para incorporação destes mecanismos. A lei-

tura dos *inputs* de rato promove a chamada das mesmas funções que são invocadas pelos *inputs* de teclado. A única diferença deste mecanismo de leitura, relativamente ao uso de teclado, é o cálculo do número de quadrículas da grelha relativamente ao movimento de rato imprimido pelo utilizador. Será relacionada a largura da peça, a posição do rato e a distância percorrida por este (proporcionado para o número de quadrículas) para relacionar o número de translações a imprimir à peça. Usando o rato também é possível imprimir rotação à peça.

Com a explicação dos mecanismos envolvidos na translação e rotação das peças, o que influencia a posição das peças na grelha de jogo, surge a pertinência de referir de que modo a posição de uma peça jogada pode influenciar a dinâmica do jogo. Assim, explicar o modo de colisão entre peças e a geração de novas são temas necesários de abordar para uma melhor compreensão das funcionalidades implementadas neste projeto. A seção seguinte tem como propósito a explicação do modo de implementação dos conceitos referidos.

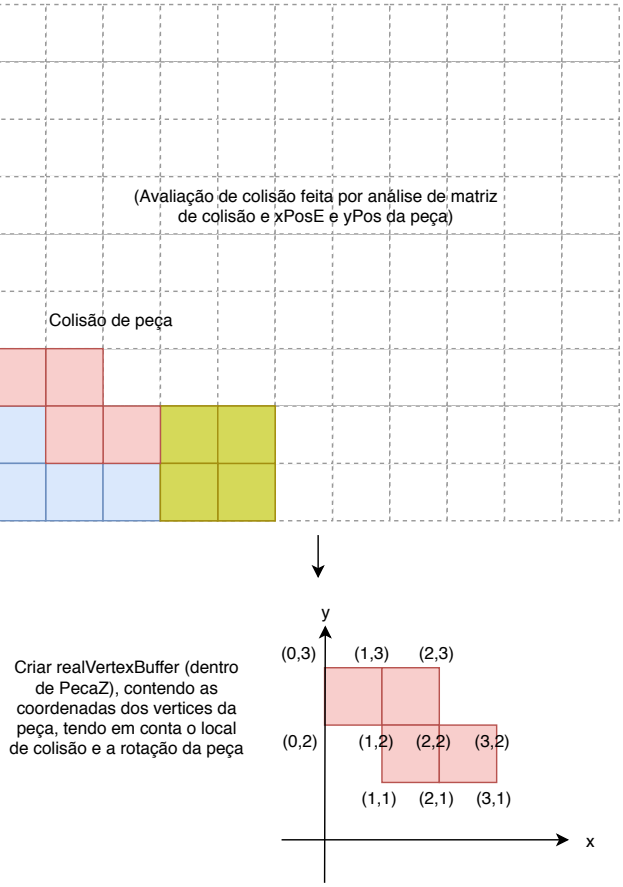


Figura 4.7: Colisão de peça e consequente criação de vertexbuffer, considerando a posição da peça e a sua rotação no momento da colisão.

4.4 Colisão entre Peças e Geração de Novas

A colisão entre peças promove alterações na grelha de jogo que terão de ser corretamente registadas para garantir a consistência do jogo.

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. 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.

Na figura 4.7 é também evidenciado a criação de um *buffer* de vértices (denominado `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 texturas dos vértices. A figura 4.8 representa, esquematicamente, o referido.

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; a mesma correspondência já era verificada no momento de instanciação de uma nova peça, como visível 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 e 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, representando o ciclo de instanciação de peça até à sua colisão, será incorporado em Anexo (anexo 1), 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

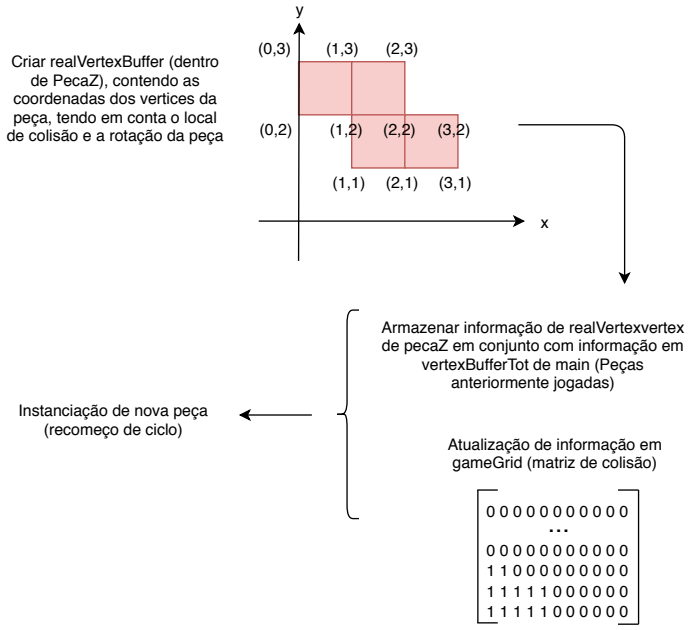


Figura 4.8: Atualização da informação da grelha de jogo (`vertexBufferTot`) e matriz de colisão, resultantes da colisão da peça.

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, seguindo como referência o código disponibilizado nas aulas práticas [10].

Com a explicação dos mecanismos associadas aos movimentos das peças, sua colisão e geração de novas, o processo de texturização é o próximo a carecer de esclarecimento. Assim, a seção seguinte contém os excertos de códigos mais relevantes para a explicitação das medidas implementadas, visando a incorporação de texturas neste projeto.

4.5 Texturização

Para uma melhor representação das peças, e face ao requisito apresentado no enunciado, evoluímos a representação destas de cores para texturas. A implementação de carregamento de texturas em OpenGL, usado no nosso projeto, seguiu como

referência o apresentado nas aulas práticas, em particular, [11]. O excerto 4.4 representa, de forma sucinta, o mecanismo envolvido no uso de texturas no nosso projeto.

```
void setMVP_And_UniqueLoads() {
  ...
  glGenTextures(6, TextureID);

  // Texturas de pecas
  ucaTexData = stbi_load(caTiles, &iTexWidth, &iTexHeight,
    &iTexNumChannels, 0);

  glActiveTexture(GL_TEXTURE0);
  glBindTexture(GL_TEXTURE_2D, TextureID[0]);
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, iTexWidth,
    iTexHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE, ucaTexData);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

  stbi_image_free(ucaTexData);
  ...

  // Use our shader
  glUseProgram(programID);
  globalBack = glGetUniformLocation(programID, "iBackground");
}
```

Listing 4.4: Carregamento de texturas em OpenGL.

A linha contendo `glGenTextures` indica o número de nomes de texturas a gerar; foram criados 6 pois aspectos como imagem de fundo, representação de próxima peça e em armazenamento, entre outras foram representadas via texturas. No mesmo excerto é evidenciado o carregamento da informação da imagem para a variável `ucaTexData`, usando para tal a biblioteca *stb_image*, guardando a informação de altura, largura e número de canais de textura de imagem; o caminho da imagem encontra-se associado a `caTiles`, um *array* de *chars*. A informação recolhida no excerto referido será posteriormente usada em `glTexImage2D`. Será ativada a unidade 0 à textura a carregar e será associado o carregamento da textura à posição 0 do *array* de identificador de texturas; esta informação está expresa na linha `glActiveTexture(GL_TEXTURE0)`. De seguida, será carregada a informação da imagem para a textura, com referência ao apontador usado (`ucaTexData`). Será feita a parametrização de filtros de magnificação e minificação, sendo de seguida libertado o conteúdo do apontador referente à imagem carregada. Por fim, será dada a conhecer a variável `iBackground`, no *fragmentshader*, usada para distinguir a textura a usar mediante o objeto.

No excerto 4.4 não é distinguido a textura a usar no *fragmentshader*, por recurso `iBackground`, pois tal é feito numa outra função, exibida em 4.5. A atribuição de um valor à variável `iBackground`, via `glUniform1i`, tem a si associado também a implementação da correspondência com identificadores de *vertex* e o texture.

```
void drawObjects(int iIdentificador) {
  ...

  switch(iIdentificador){
    // Desenhar ecra de Pause
    case 14:
      glUniform1i(globalBack, 3);
      iIDVertexBuffer = vertexbufferPause;
      iIDTextureBuffer = texturebufferPause;
      iDrawSize = g_vertex_buffer_dataPause.size();
      break;
    ...
  }
  ...

  // 1st attribute buffer : vertices
  glEnableVertexAttribArray(0);
  glBindBuffer(GL_ARRAY_BUFFER, iIDVertexBuffer);
  glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, (void*)0);

  // 2nd attribute buffer : textures
  glEnableVertexAttribArray(1);
  glBindBuffer(GL_ARRAY_BUFFER, iIDTextureBuffer);
  glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 0, (void*)0);

  // Draw all the previously played pieces
  glDrawArrays(GL_TRIANGLES, 0, iDrawSize);

  ...
}
```

Listing 4.5: Exemplo de desenho de um objeto, usando texturização.

Os identificadores de *vertex* e o texture são usados, como é visível na parte final do código, para serem associados aos atributos de *vertex* 0 e 1, respetivamente, valores estes usados no *vertexshader* para determinação de posição e passagem de informação para o cálculo de cor, no *fragmentshader*, também respetivamente. A função `drawObjects` exibida é usada para desenhar as diferentes texturas usadas no jogo, nomeadamente, ecrã de pausa, menu de controlos, textos "Score","Lines","Level","Next","Hold", imagens de fundo, peças de jogo, entre outros. A diferenciação das texturas e identificadores de *buffer* a serem usados é feita por recurso a um `switch`, usando como referência distintiva um inteiro (`iIdentificador`); no excerto 4.5, o valor 14 corresponde ao desenho do ecrã de *Pause*. Os diferentes identificadores usados, e a sua correspondência com a textura a desenhar, foi incorporado em `main.cpp`, na função `main`, sob a forma de comentário, para melhor compreensão das funcionalidades implementadas. A principal razão para esta implementação prende-se na reutilização de código.

A distinção das texturas a usar, no *fragmentshader*, é observável no excerto de código 4.6. Mediante o valor de `iBackground`, uma textura diferente será a usada para promover a coloração do objeto.

```
layout(binding=1) uniform sampler2D Backgroundtexture;
layout(binding=2) uniform sampler2D Gamegridtexture;
layout(binding=3) uniform sampler2D Pausetexture;
...

void main(){
  if (iBackground == 1) {
    color = texture(Backgroundtexture, fragmentTexture);
  }
  else if (iBackground == 2) {
    color = texture(Gamegridtexture, fragmentTexture);
  }
  else if (iBackground == 3){
    vec4 text = texture(Pausetexture, fragmentTexture);
    text.a = 0.5; // opacidade
    color = text;
  }
  ...
}
```

Listing 4.6: Texturização em *fragmentshader*.

No excerto de código 4.6 é também visível a alteração do valor α de uma textura, exibindo a existência de opacidade nas texturas; a linha `text.a = 0.5` evidencia o referido. O uso de opacidade pode ser observado na figura 4.9, onde foi carregada a textura do ecrã *Pause*, bem como da peça em jogo, das peças anteriormente jogadas, da próxima peça, peça armazenada, a imagem de fundo e todo o texto visível na imagem. É notório que, mesmo com a presença de uma textura sob a grelha de jogo, esta ainda assim é observável, reforçando a noção de uso de opacidade em texturas.

Existem outras evidências presentes na imagem 4.9 que merecem destaque. Uma dessa evidências é o facto de diferentes texturas estarem a serem representadas em locais diferentes do ecrã; exemplos destas são a próxima peça ou a peça armazenada. Tal foi conseguido por recurso a *Viewports*, onde houve diferenciação destes mediante o objeto a desenhar.

O armazenamento de uma peça é conseguido por uma função *callback*, nomeadamente, o clique da tecla C e a queda abrupta da peça é conseguida por clique em *Space*. Estas funções *callback* foram incorporadas no código, ao qual o excerto 4.2 se refere; não foi incorporado em 4.2, por efeitos de simplicidade. A instanciação de peça armazenada, aquando da sua invocação (por clique em C), é conseguida pela chamada de função `returnPeca`, presente em `main.cpp`, que recebe o objeto gerador de peças e um inteiro identificador, correspondente à peça a instanciar. Esta função é responsável por criar uma peça, correspondente ao identificador, com o nível de jogo apropriado e a grelha de jogo atualizada; a instanciação da nova peça contém também *buffers* de vértices e texturas adequados a esta (informação presente na classe de cada peça). De referir que este mecanismo é também utilizado para guardar peça e para indicar qual a peça que virá a seguir; cada um dos casos referido conterá um inteiro identificador distinto.

As opções de teclado disponibilizadas para este jogo são exibidas por recurso a *Enter*, o que promove a exibição de uma textura que contém todos os controlos do jogo. Tanto a textura de Controlos como a textura de *Pause*, promovem um bloqueio de todas as funções *callback* (excetuando a usada para invocar estas) bem como a queda automática da peça.

A gestão da queda automática é conseguida por recurso à passagem de um *boolean* à classe associada à peça instanciada; caso o *boolean* seja *true* a queda automática será interrompida, caso contrário caíra com uma velocidade concordante com o nível de jogo. O interrompimento da peça traduz-se num reajuste da altura da peça e do tempo inicial de queda; o tempo de queda é reiniciado a 0 e a altura da peça reajustada para ser concordante com a altura aquando do momento de pausa.

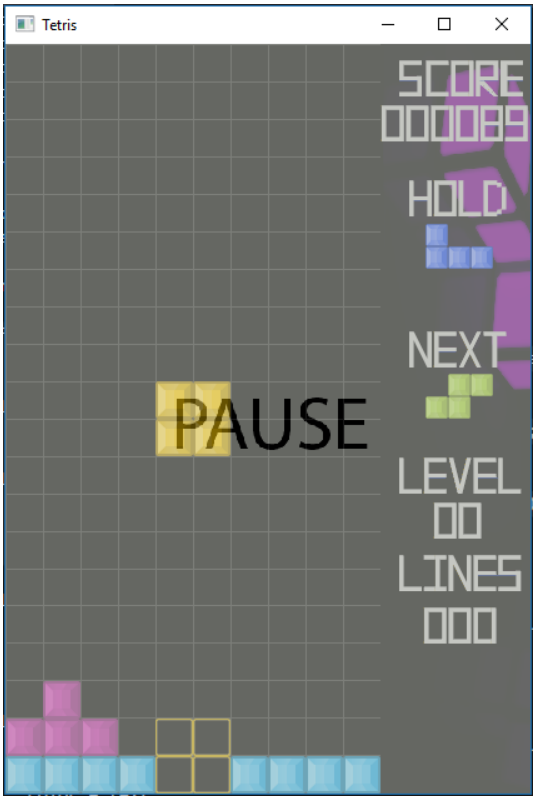


Figura 4.9: *Screenshot* de jogo de *Tetris*.

A imagem 4.9 permite visualizar também o uso de texturas para representar o local onde a peça irá colidir. A representação de peça é feita pelo uso de texturas sólidas, enquanto que a colisão da peça é representada por uma textura de contornos da mesma cor que a textura sólida. Para detetar o local de colisão, cada classe de peça tem uma função, denominada `collisionYPos`, que, dado o formato da peça e a rotação desta, imposta pelo utilizador, avalia a posição vertical mais baixa possível onde a colisão ocorrerá, representando esta nesse local. De forma análoga à repre-

sentação de peças comuns, também esta terá acertos de posição; como referência, acertos de posição de peças comuns pode ser visível no excerto 4.3.

Com a implementação das texturas, resta implementar mecanismos de jogo que promovam uma melhor jogabilidade. Conceitos como a eliminação de linhas e, consequentemente, pontuações, conferem um ambiente competitivo ao jogo e uma melhor experiência para o utilizador. Na seção seguinte, será feita uma explicação do modo de implementação de eliminação de linhas.

4.6 Eliminação de Linhas

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 4.7 demonstra a função, em `main.cpp`, responsável pela implementação deste mecanismo.

No excerto referido, a função 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 `newSizeVertex`. 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 texturas. É, no entanto, notório, no excerto de código 4.7, a diferença de proporção entre o *buffer* de textura e de vértices; cada textura é representada por conjuntos de pontos de 2 coordenadas, enquanto que o de vértices por de 3. Desta substituição resulta a atualização das variáveis `i` e `newSizeVertex`, decrementando-as em 18; a mesma lógica é aplicada para texturas, decrementando-as em 12.

A variável `newSizeVertex,` do excerto de código 4.7, 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, no final do processo de eliminação de linhas; de forma análoga, é feito o mesmo procedimento para o *buffer* de texturas, usando para tal a variável `newSizeTexture`.

No final de eliminação de um bloco e, consequente, ajuste de tamanho de *buffers* (de vértices e texturas), é invocada a função `desenhaAmbiente`, responsável por promover todas as atualizações de *buffers* dos restantes objetos da cena, sem instanciação de novas peças. Esta caraterística provoca um efeito visual apelativo da eliminação das linhas, ao invés de um desaparecimento abrupto destas. 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 4.7, 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 texturas não necessita de ser alterado neste caso.

```
void eliminaLinha(int iLinha) {
  int i;
  newSizeVertex = g_vertex_buffer_dataTot.size();
  newSizeTexture = g_texture_buffer_dataTot.size();
  for (i = 0; i < newSizeVertex; 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);
      }
      // Proporcao: por cada ponto no vertexbuffer ha 3 valores,
      // no entanto no texturebuffer so ha 2 valores
      for (int k = (i / 3 * 2); k < g_texture_buffer_dataTot.size()
         - 12; k++)
      {
        g_texture_buffer_dataTot.at(k) =
          g_texture_buffer_dataTot.at(k + 12);
      }
      // Atualizacao de variaveis
      newSizeVertex -= 18;
      newSizeTexture -= 12;
      i -= 18;

      // Atualiza tamanho de texture e vertice buffer e desenha pecas
          envolvidas
      // Confere efeito de desaparecimento de blocos aquando da
          eliminacao de linhas
      g_vertex_buffer_dataTot.resize(newSizeVertex);
      g_texture_buffer_dataTot.resize(newSizeTexture);
      desenhaAmbiente();
    }
    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;
        }
      }
    }
  }
}
```

Listing 4.7: Eliminação de linhas da grelha de jogo.

A eliminação de linhas é invocada a cada colisão de peça, caso a análise da matriz de colisão (visível na figura 4.8) 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 quantas as que tiverem esta caraterística. Naturalmente, após a eliminação de linhas (excerto 4.7), a matriz de colisão também será atualizada, visando uma consistência entre a grelha de jogo e esta matriz, aquando da instanciação da próxima peça.

Explicado a eliminação de linhas, é pertinente explicar outros aspectos relacionados com a jogabilidade do *Tetris* desenvolvido, nomeadamente cálculo de pontuação e a passagem de nível. Estes conceitos são abordados na seção seguinte.

4.7 Pontuação e Passagem de Nível

Visando uma sensação de competitividade e promoção de um objetivo ao jogo, implementou-se a contabilização de pontos, bem como a evolução de níveis. Tanto a representação palavras "Score","Lines","Level","Next","Hold" como o nível de jogo, a pontuação e o número de linhas eliminadas no nível são conseguidas por recurso a objetos texturizados; o modo de implementação de texturização foi explicado na seção 4.5. No entanto, contrariamente às palavras referidas, o nível de jogo, a pontuação e o número de linhas foram desenhadas dinamicamente, cuja representação foi feita algarismo a algarismo, promovendo alteração constante destas variáveis no decorrer de jogo. Para melhor organização do projeto, armazenou-se toda a informação de vértices e texturas referente a estas palavras e representação numérica de pontuação, nível e linhas em ficheiros .cpp e .hpp, denominados `Font`. Um exemplo da texturização de algarismo, usada para representação da pontução de jogo, pode ser vista em 4.10.

Figura 4.10: Representação de algarismo de pontuação, por recurso a texturas.

A passagem de nível é conseguida por eliminação de um determinado número de linhas e esta passagem promove um aumento da velocidade da queda de peças, como observado no jogo de *Tetris* tradicional. A abordagem para a passagem de níveis segue o modelo utilizado pela *Nintendo*, tendo sido usada a referência `https:`

`//tetris.fandom.com/wiki/Tetris_(NES,_Nintendo)`. O jogo termina caso a instanciação de uma nova peça promova um preenchimento de texturas de altura superior à da grelha de jogo. Aquando do término de jogo será apresentada uma textura com aparência similar ao ecrã *Pause*, visível na figura 4.9, com o texto *Game Over*; para sair da aplicação, aquando do *Game Over*, terá de pressionar *ESC*.

Na última seção será referido o modo implementação de áudio, uma funcionaliade que tem como objetivo melhorar a experiência do utilizador.

4.8 Incorporação de Áudio

A implementação de áudio no jogo de *Tetris* visa promover um ambiente apelativo de jogo, tendo para tal incorporado som ambiente, som aquando da eliminação de linhas, som aquando da mudança de nível e som aquando de término de jogo.

Para a implementação desta funcionalidade recorreu-se à biblioteca *irrKlang*, seguindo o exemplo em [2]. Resumidamente, invoca-se a biblioteca, instancia-se um objeto da mesma (`ISoundEngine`) e associa-se uma música e a caraterística de repetição associada a esta, por recurso ao método `play2D`; se o primeiro parâmetro é `GL_FALSE` a música não se repete, tocando apenas uma vez, enquanto que com `GL_TRUE` a música ficará continuamente a tocar. Adicionalmente, ajusta-se o volume de áudio, tomando a medida de segurança de verificar que o objeto `ISound` exite. O excerto de código em 4.8 exibe o acima descrito. Neste excerto é exibido a música a ser reproduzida aquando da limpeza de uma linha no jogo de *Tetris*.

```
#pragma comment(lib, "irrKlang.lib")
#include <irrKlang.h>
using namespace irrklang;
...

ISoundEngine *SoundEngine = createIrrKlangDevice();
bool evaluatePieceCollision(Peca& pPeca) {
  ...
  if (iNumeroLinhasAEliminar != 0) {
    ISound* sound = SoundEngine->play2D("resources/audio/levelUp.wav",
        GL_FALSE, GL_FALSE, GL_TRUE);
    if (sound) {
      sound->setVolume(0.25);
    }
    ...
  }
  ...
}
```

Listing 4.8: Reprodução de som, aquando da eliminação de uma linha.

Capítulo 5

Trabalhos Futuros

Findo a realização do projeto, e cumprindo todos os requisitos impostos pelo enunciado, fica projetado, como trabalhos futuros, implementações que confiram maior dinamismo ao jogo desenvolvido.

Uma implementação futura, não contemplada no projeto, mas que conferiria maior valor ao trabalho desenvolvido, seria uma evolução do jogo *Tetris* para 3D. A incorporação de modelos de iluminação, nesta implementação, elevaria a qualidade do trabalho e seria uma forma de incorporar funcionalidade lecionadas na aula. Uma aparência mais real das peças, com incorporação de ondulações de peças, aquando da sua queda, seriam outras implementações interessantes a considerar.

Alheio à melhoria de aparência visual do jogos, a criação de menu, capacitado para apresentar diferentes variantes do jogo, poderia ser uma outra funcionalidade a implementar. Dentro do modelo de jogo *Tetris*, podemos imaginar a criação de um modo de jogo em que linhas vão aparecendo, caso o utilizador demore muito tempo a limpar a grelha de jogo ou um modo de jogo em que as peças não estejam sempre visíveis, incrementando a dificuldade associada à limpeza de linhas. Estes dois exemplos demonstram tipos de jogo, com imensas similaridades ao trabalho desenvolvido, que conferem um maior desafio ao utilizador e, como tal, uma melhor experência de jogo a este. A implementação destes novos modos de jogo reutilizaria grande parte das funcionalidades impostas neste projeto pelo que seriam incorporações a considerar em trabalhos futuros.

Associado à melhoria de experiência de jogo do utilizador, o conceito de armazenar os melhores resultados obtidos, sob a forma de uma tabela, é também uma possível implementação futura. Para tal, armazenamento de informação entre sessões de jogo seria necessário, potencialmente sob a forma de ficheiro ou, preferencialmente, usando base de dados.

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 translação e rotação associados, instanciação aleatória de peças, gestão de mecanismos de colisão, eliminação de linhas, implementação de texturas, pontuações, níveis e áudio seguindo a filosofia de jogo do *Tetris* tradicional.

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 contenha valor acrescido, face ao estado atual do projeto.

No momento de entrega deste relatório, o projeto encontra-se terminado. Todos os requisitos impostos pelo enunciado foram cumpridos e novas funcionalidades, referidas no capítulo anterior, são possíveis melhorias a implementar ao projeto, no futuro.

Toda a componente téorica associada ao OpenGL, foi aplicada no desenvolvimento do projeto, com a exceção do modelo de iluminação e coloração, devido à natureza do trabalho proposto. A evolução deste projeto para 3D, promoveria um âmbito mais apropriado para a implementação das técnicas mencionadas.

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] Joey. de Vries. LearnOpenGL - Audio. https://learnopengl.com/In-Practice/2D-Game/Audio, 2015. [Acedido em 01/06/2019].

[3] Joey. de Vries. LearnOpenGL - Textures. https://learnopengl.com/Getting-started/Textures, 2016. [Acedido em 01/06/2019].

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

[5] Andy. Khov. Tetris3D by Andy Khov. https://https://github.com/andykhv/Tetris3D, 2018. [Acedido em 01/06/2019].

[6] Nikolay. Mayorov. GitHub - nmayorov/tetris-game: A simple Tetris game in OpenGL. https://github.com/nmayorov/tetris-game, 2017. [Acedido em 02/06/2019].

[7] 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].

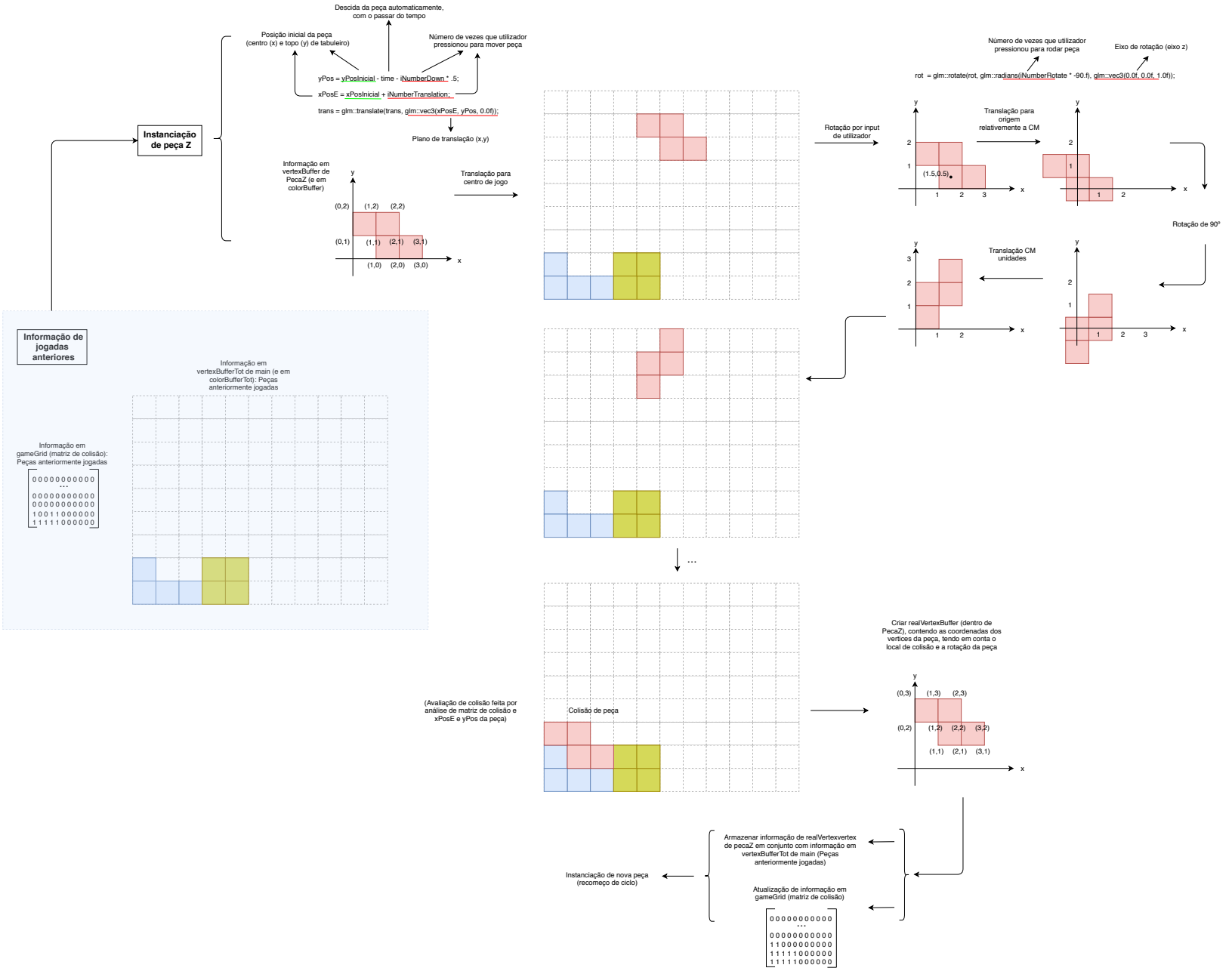
[8] opengl tutorials. Tutorial 6 : Keyboard and Mouse. http://www.opengl-tutorial.org/beginners-tutorials/tutorial-6-keyboard-and-mouse/, 2018. [Acedido em 13/03/2019].

[9] David. Wolff. *OpenGL 4 Shading Language Cookbook*. ISBN: 978-1-78216-702-0. Packt Publishing, 2nd edition, 2013.

[10] David. Wolff. glslcookbook/chapter02 at master · daw42/glslcookbook. https://github.com/daw42/glslcookbook/tree/master/chapter02, 2018. [Acedido em 27/04/2019].

[11] David. Wolff. glslcookbook/chapter04 at master · daw42/glslcookbook. `https://github.com/daw42/glslcookbook/tree/master/chapter04`, 2018. [Acedido em 22/05/2019].

Apêndice


Descida da peça automaticamente, com o passar do tempo
Posição inicial da peça (centro (x) e topo (y) de tabuleiro)
Número de vezes que utilizador pressionou para mover peça
yPos = yPosInicial - time - iNumberDown.* .5;
xPosE = xPosInicial + iNumberTranslation;
trans = glm::translate(trans, glm::vec3(xPosE, yPos, 0.0f));
Plano de translação (x,y)
Instanciação de peça Z
Informação em vertexBuffer de PecaZ (e em colorBuffer)
Translação para centro de jogo
Rotação por input de utilizador
Número de vezes que utilizador pressionou para rodar peça
Eixo de rotação (eixo z)
rot = glm::rotate(rot, glm::radians(iNumberRotate * -90.f), glm::vec3(0.0f, 0.0f, 1.0f));
Translação para origem relativemente a CM
Rotação de 90º
Translação CM unidades
Informação de jogadas anteriores
Informação em vertexBufferTot de main (e em colorBufferTot): Peças anteriormente jogadas
Informação em gameGrid (matriz de colisão): Peças anteriormente jogadas
(Avaliação de colisão feita por análise de matriz de colisão e xPosE e yPos da peça)
Colisão de peça
Criar realVertexBuffer (dentro de PecaZ), contendo as coordenadas dos vertices da peça, tendo em conta o local de colisão e a rotação da peça
Armazenar informação de realVertexvertex de pecaZ em conjunto com informação em vertexBufferTot de main (Peças anteriormente jogadas)
Atualização de informação em gameGrid (matriz de colisão)
Instanciação de nova peça (recomeço de ciclo)