

UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

"O Pesadelo de Fluffy" versão 2.0
implementação do modo retido e expansão

Vitor Carvalho de Melo

MONOGRAFIA FINAL
MAC 499 — TRABALHO DE
FORMATURA SUPERVISIONADO

Orientador: Prof. Dr. Paulo A. V. Miranda

São Paulo
2023

*O conteúdo deste trabalho é publicado sob a licença CC BY 4.0
(Creative Commons Attribution 4.0 International License)*

Resumo

Vitor Carvalho de Melo. "**O Pesadelo de Fluffy**" versão 2.0: *implementação do modo retido e expansão*. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2023.

O desenvolvedor de jogos deve sempre escolher quais das dezenas de aspectos envolvidos na criação de seu *software* ele deve focar sua atenção e trabalho em detrimento de alguns outros, a fim de cumprir sua ideia original em tempo hábil e de forma satisfatória. O jogo "O pesadelo de Fluffy", desenvolvido pela bacharelanda Giulia de Nardi como projeto de conclusão de curso, apesar de muito bem feito e divertido, utilizava um modo de renderização gráfica da API OpenGL descontinuado desde 2008, o modo imediato. Neste novo projeto, por meio de uma refatoração categórica de código-fonte e do emprego das boas práticas do desenvolvimento de jogos baseados em OpenGL, foi possível adaptar este jogo para que utilizasse a forma atual e recomendada de renderização gráfica, a renderização de modo retido, ou *core-profile*. Além disso, novos modos de jogo, funcionalidades e fases também foram desenvolvidas para este jogo, que honra Fluffy, o mascote do Instituto de Matemática e Estatística da USP.

Palavras-chave: OpenGL. SDL2. C++. Motor Gráfico. Jogo eletrônico. Quebra-cabeça. 3D.

Abstract

Vitor Carvalho de Melo. "**Fluffys' Nightmare**" versions 2.0: *retained mode implementation and game expansion*. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2023.

Every game developer must choose which of the dozens of aspects involved in developing their project they must focus their work and attention over every other, in order to fulfill their original idea both in able time and in a satisfactory manner. "O pesadelo de fluffy", a game developed by the senior graduation student Giulia de Nardi as her graduation thesis, though very well made and fun, used an OpenGL rendering mode that has been considered deprecated since 2008, the immediate mode. In this current project, through a categoric source code refactoring and the use of current standards of OpenGl coding, it was possible to adapt this game so that it now uses the current and recommended method of graphics rendering, the retained, or core-profile, mode. New game modes, functionalities and levels were also developed and integrated into the game, which honors Fluffy, the mascot of the Instituto de Matemática e Estatística, a faculty of the Universidade de São Paulo.

Keywords: OpenGL. SDL2. C++. Game Engine. Videogame. Puzzle. 3D.

Listas de figuras

1	O crescimento dos videogames, visualizado. Fonte: Pelham Smithers.	1
2	Maiores mercados em faturamento com videogames. Fonte: NewZoo.	2
1.1	Número de polígonos do personagem principal de um jogo (modificado)	4
1.2	Jogo "Portal", lançado em 2007 pela Valve Studios.	5
1.3	Jogo "Octopticom", lançado em 2018 pela UP Development.	5
1.4	Jogo "Midnight is Lost", lançado em 2021 pela Robotizar Games.	6
1.5	Jogo "Catherine", lançado em 2011 pela Atlus.	6
2.1	O Pesadelo de Fluffy, desenvolvido em 2022 por Giulia de Nardi.	9
2.2	Esquematização UML do projeto (simplificada).	10
2.3	Eixos de rotação de um ambiente 3D. Fonte: Wikipedia.	11
2.4	<i>Pipeline</i> gráfico do OpenGL. Fonte: Learn OpenGL.	15
2.5	Comparação entre o modo de jogo "Sokoban" e o jogo inspiração, de mesmo nome.	17
2.6	Comparação entre o modo de jogo "MiniFluffys" e o jogo inspiração para o modo.	18
2.7	Novos blocos do modo de jogo MiniFluffys.	19
2.8	Componentes de iluminação de Phong, separados e combinados. Fonte: Basic Lighting.	21
2.9	Diferença entre o formato do brilho especular nos modelos de Phong e Blinn–Phong. Fonte: Computação Gráfica.	22
2.10	Representação da técnica <i>forward ray tracing</i> . Note que alguns raios que emanam da fonte de luz nunca atingem a câmera. (GLASSNER, 1989)	23
2.11	Diagrama de traçamento de raios. Fonte: Wikipedia.	23

3.1 Exemplo de fase do jogo que inclui todos os novos modos de jogo.	26
--	----

Lista de programas

2.1 Função updateCameraVectors() da classe Camera.	11
2.2 Trecho da função handleKey().	12
2.3 Trecho da função setupBlockVao().	16

Sumário

Introdução	1
1 Motivação	3
1.1 Viabilidade de desenvolvimento	3
1.2 Relevância e permanência de jogos de <i>puzzle</i>	4
1.3 Por que OpenGL?	7
2 Desenvolvimento	9
2.1 Estrutura do projeto	9
2.2 Controles e Bibliotecas	12
2.3 Modo de renderização	13
2.4 Novas funcionalidades e modos de jogo	17
2.5 Otimização e tratamento de <i>bugs</i>	19
2.6 Modelos de iluminação	20
3 Resultados e Conclusões	25
Referências	27

Introdução

A indústria de *videogames* tem o poder de surpreender a muitos com seu tamanho, alcance e constante evolução. Pela Figura 1, estima-se que, em 2020, o faturamento total desta indústria tenha sido de 165 bilhões de dólares; como fator de comparação, a indústria de cinema mundial faturou, em 2019, cerca de 41 bilhões de dólares, o que representa menos de um quarto da indústria de jogos digitais.

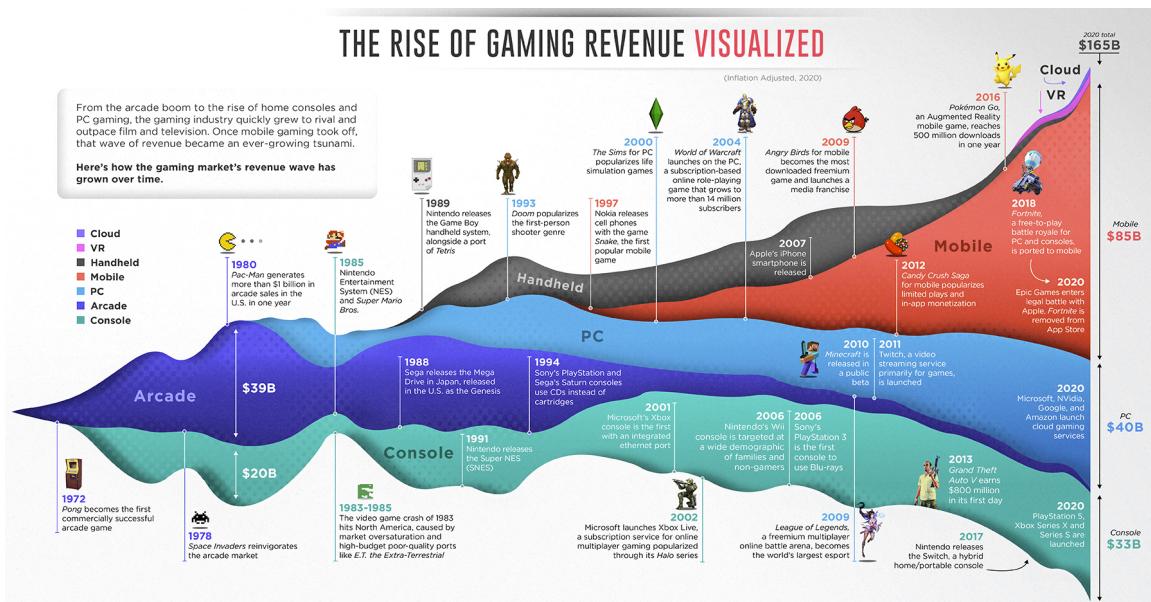


Figura 1: O crescimento dos videogames, visualizado. Fonte: Pelham Smithers.

Engana-se, também, quem imagina que este mercado no Brasil não é tão grande. Em décimo lugar no ranking, logo depois da Itália, o Brasil provê atualmente 2,6 bilhões de dólares à indústria mundial de *videogames* (ver Figura 2). Infelizmente, muito pouco deste dinheiro permanece no país, pois esta indústria nacional ainda é muito pequena e, segundo FLEURY *et al.*, 2014, a maior parte das empresas brasileiras que produzem jogos são de pequeno porte, com uma receita anual de até 240 mil reais. Apesar disso, não há motivos para desânimo pois, assim como a mundial, a indústria de jogos digitais brasileira tem crescido em enorme ritmo; segundo o G1, mais de 600% em apenas oito anos.

	Mercado	Receita (USD)	Jogadoras
1.	Estados Unidos	46,4B	209,8 milhões
2.	China	44,0B	699,6 milhões
3.	Japão	19,1B	73,4 milhões
4.	Coreia do Sul	7,4B	33,3 milhões
5.	Alemanha	6,5B	49,5 milhões
6.	Reino Unido	5,5B	38,5 milhões
7.	França	4,1B	38,8 milhões
8.	Canadá	3,3B	22,0 milhões
9.	Itália	3,1B	36,1 milhões
10.	Brasil	2,6B	102,6 milhões

Figura 2: Maiores mercados em faturamento com videogames. Fonte: [NewZoo](#).

Assim sendo, a área de desenvolvimento de jogos, pelo seu tamanho e relevância, merece mais espaço não só no mercado mas também nas universidades e escolas. Por isso, este projeto embarca neste mundo de desenvolvimento de *videogames* ao construir um jogo tridimensional de quebra-cabeças que utiliza como principal ferramenta a API (Interface de programação de aplicações) Gráfica [OpenGL](#). Esta API embarca consigo, além de funções matemáticas e geométricas, toda uma gama de comandos e variáveis que um desenvolvedor pode utilizar para controlar e se comunicar com a placa gráfica do computador.

Dito isso, este trabalho tem como objetivo primário a construção de *features* modernas do OpenGL e de outras ferramentas no jogo eletrônico estilo *puzzle* [O Pesadelo de Fluffy](#) e, secundariamente, expandir, otimizar e evoluir esse jogo, com novas fases e objetivos. O arquivo executável do jogo final pode ser encontrado no [Github](#).

Capítulo 1

Motivação

Não é surpreendente, dado o tamanho da indústria de *videogames* atual, que existam dezenas (senão centenas) de categorias e subcategorias de jogos no mercado ([PAGE, 2016](#)), então por quê escolher o estilo de *puzzle*, ou quebra-cabeças, para realizar este trabalho? Pode-se destrinchar esta resposta em algumas seções, o que deve também ajudar a esclarecer o leitor sobre a indústria de jogos virtuais e seu estado atual.

1.1 Viabilidade de desenvolvimento

A complexidade e quantidade de recursos dos *videogames* têm aumentado de maneira expressiva desde seu nascimento relativamente humilde nos *Arcades* ([CONSALVO, 2022](#)) até os dias atuais.

Para ilustrar este exemplo, observa-se a tabela abaixo: todos os 15 jogos listados são ambientados em um cenário tridimensional, e utilizam polígonos (principalmente triângulos) para modelar os objetos das cenas que, de forma simples, são em seguida renderizados pela unidade de processamento gráfico (GPU) do computador. Esta tabela referencia a quantidade de polígonos que modelam o personagem principal (*main character*) de alguns jogos-chave de três consoles de *videogame* famosos, o Playstation 2, lançado no Brasil em novembro de 2002, o Playstation 3, em agosto de 2010, e o Playstation 4, em fevereiro de 2014.

PS2	PS3	PS4
1.500 (2005) Fahrenheit	30.000 (2013) Beyond Two Souls	550.000 (2017) Horizon Zero Dawn
4.000 (2005) Metal Gear Solid 3 Snake	22.000 (2010) God of War 3	100.000 (2015) The Order 1886
4.000 (2001) Gran Turismo 3	40.000 (2006) Virtua Fighter 5	120.000 (2014) inFamous Second Son
4.000 (2001) Jak & Daxter	37.000 (2009) Uncharted 2	100.000 (2016) Final Fantasy XV
10.000 (2002) Mortal Kombat Deadly Alliance	28.501 (2012) Assassins Creed 3	250.000 (2014) Driveclub

Figura 1.1: Número de polígonos do personagem principal de um jogo (modificado)

Pela figura, nota-se um aumento médio de aproximadamente duas ordens de magnitude na quantidade de polígonos em apenas 10 anos. Nos casos extremos, para o PS2 haviam jogos cujo personagem principal é desenhado com 1500 polígonos, no caso de "Fahrenheit", enquanto no PS4, no caso do jogo "Horizon Zero Dawn", o protagonista é desenhado com 550.000 polígonos; isso representa um aumento de mais de 360 vezes, em dois jogos similares.

Ao entender o tamanho da complexidade do desenvolvimento de jogos, e considerando-se o interesse do autor em desenvolver um jogo tridimensional, decidiu-se que trabalhar com um jogo de puzzle, em que a qualidade gráfica não é a prioridade, forneceria um desafio concordante com o escopo de um trabalho de conclusão de curso de graduação.

1.2 Relevância e permanência de jogos de *puzzle*

Com tantas categorias e novos modos de jogar, como os óculos de realidade virtual e simuladores cada vez mais realistas, não seria justo julgar aqueles que imaginavam que jogos de quebra-cabeça, com seus gráficos e mecânicas geralmente mais simples, teriam perdido parte do interesse do grande público nos últimos anos. Entretanto, os jogos de quebra-cabeça, ou *puzzles*, representam no mercado de *videogames mobile* (telefones celulares, tablets, etc.) cerca de 8% do total de vendas mundial, que em 2020 era de quase 87 bilhões de dólares. Isso demonstra um contínuo interesse por essa categoria de jogos, e é possível arriscar explicar alguns motivos que justificam essa manutenção:

1. Diversas subcategorias

Muitas (e talvez todas) das subcategorias de quebra-cabeças já foram adaptadas, de uma forma ou outra, para *videogames*. Tal diversidade ajuda a manter o interesse e atrair mais jogadores para este tipo de jogo. Dentre estas subcategorias, temos:

- **Jogos de física**

Quebra-cabeças cuja jogabilidade envolve principalmente resolver problemas de física, como gravidade, flutuação e elasticidade dos elementos em tela. O melhor exemplo dessa sub-categoria é um dos jogos mais famosos da última década, *Portal* (ver Figura 1.2), em que os jogadores devem criar portais de entrada e de saída para atravessarem ou conduzirem objetos a fim de cumprir os objetivos propostos pelo jogo. No Brasil, este jogo já foi utilizado em aulas de Física nas escolas para ensinar conceitos importantes como gravidade, velocidade terminal e oscilações (COSTA e MIRANDA, 2017).



Figura 1.2: Jogo "Portal", lançado em 2007 pela Valve Studios.

- **Jogos de programação**

Alguns jogos de quebra-cabeça envolvem até a produção de código ou pseudo-código para solucionar os problemas apresentados, seja de forma escrita ou visual. No site de vendas de jogos online *Steam*, há uma lista de jogos de quebra-cabeças que envolvem programação, como o jogo *Octopticom* (ver Figura 1.3) , em que, segundo a descrição dos criadores, o jogador deve "utilizar lasers, espelhos e outros componentes para ler, transformar e escrever sequencias, a fim de desenvolver e otimizar aparelhos computacionais".

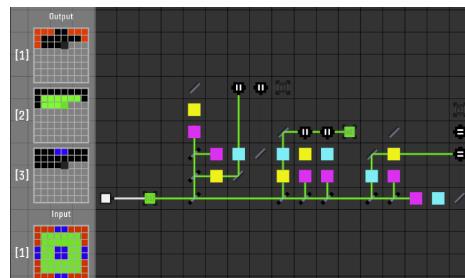


Figura 1.3: Jogo "Octopticom", lançado em 2018 pela UP Development.

- **Sokoban**

Original de 1981, este jogo é uma das maiores inspirações para a indústria de videogames de quebra-cabeças. Originalmente, o jogador assume o papel de um trabalhador de um armazém, que deve empurrar caixas a fim de organizá-las

de uma certa maneira. Entretanto, diversos jogos posteriores são baseados no modo de jogo de Sokoban, mas incluem jogabilidade e estrutura diferentes. Um exemplo é o jogo *Midnight is Lost* (ver Figura 1.4). Segundo os criadores deste jogo, "(o jogo) oferece uma experiência única e desafiadora, em que os quebra-cabeças de Sokoban devem ser solucionados de trás para frente".



Figura 1.4: Jogo "*Midnight is Lost*", lançado em 2021 pela Robotizar Games.

• Jogos de plataforma

Jogos de plataforma, em que se deve construir um caminho para chegar do ponto A ao B, também são muito comuns no mercado. Um expoente desta categoria é o jogo Catherine (ver Figura 1.5), em que, dentre outros *mini-jogos*, o jogador deve rearranjar blocos para escalar até o topo de uma torre.



Figura 1.5: Jogo "*Catherine*", lançado em 2011 pela Atlus.

Os dois últimos jogos citados, Sokoban e Catherine, serviram de inspiração para o jogo desenvolvido para este projeto de conclusão de curso.

2. Capacidade pedagógica

O uso de *videogames* como atividade de lazer está atrelado à diversos impactos psico-sociais, alguns possivelmente não tão positivos (**VON DER HEIDEN et al., 2019**). Embora essa área da psicologia ainda não seja tão bem estudada (o que chega a causar discussões públicas e polêmica na sociedade), já existem estudos que conseguiram traçar uma relação de efeitos positivos entre jogos de quebra-cabeça e o raciocínio matemático de crianças que participaram de programas de auxílio pedagógico (**AL-ABSI, 2017**). Ao entender isso, muitas escolas têm adotado *videogames* em seus currículos como forma de auxílio no aprendizado dos alunos em diversas áreas, desde raciocínio lógico até programação (**NBC, 2023**) (**NPR, 2010**).

3. Novidades técnico/criativas

Jogos eletrônicos de quebra-cabeça também não estão parados no tempo. Diversos são os exemplos de uso de criatividade, ferramentas e até novas tecnologias para criar novos jogos diferentes, o que ajuda a manter o interesse do grande público pela categoria. Um exemplo é o jogo [Plumber](#), para óculos de realidade virtual, em que o jogador deve conectar trechos de canos para fechar um circuito d'água e restaurar o fornecimento de água local.

1.3 Por que OpenGL?

Se esperançosamente os motivos pela escolha do tipo de jogo escolhido para desenvolver o trabalho já estiverem esclarecidos, resta ainda a pergunta sobre a principal ferramenta de trabalho utilizada: "Por que o OpenGL"? De fato, hoje estão presentes no mercado diversos motores gráficos (*graphic engines*), como o [Unreal Engine](#) e o [Unity](#), que trazem já integradas centenas de ferramentas matemáticas, físicas, de modelagem 3D, de áudio e de controles, dentre outras. Apesar dos desenvolvedores de jogos, ao se utilizarem do OpenGL, terem que criar todas estas ferramentas citadas por conta própria ou utilizar softwares terceiros, cuja integração com a API traz um certo custo em horas de trabalho, há algumas vantagens em se utilizar uma API gráfica como o OpenGL e, talvez ainda mais importante, desvantagens em utilizar motores gráficos.

- **Presença no mercado**

O OpenGL, desde meados de 2011, é integrado por praticamente todos os navegadores de Internet, como Chromium, Firefox, Edge e Opera, em uma versão um pouco simplificada apelidada de [WebGL](#). Com isso, o OpenGL tornou-se uma ferramenta de aprendizado de desenvolvimento de jogos amplamente utilizada por cursos de programação de jogos, além de ser a API gráfica utilizada em milhares de *web games*. Além disso, um levantamento feito na comunidade PCGamingWiki ([PCGAMINGWIKI](#), 2023) encontrou 1532 jogos, lançados desde 1991 até hoje, que utilizam OpenGL como principal ou única API gráfica. É importante lembrar que este número não representa o total de jogos já feitos utilizando a API, pois não contabiliza jogos feitos com WebGL ou com motores gráficos que se utilizam do OpenGL. Mesmo assim, este número representa um parcela significativa dos jogos disponíveis para computador no mercado.

- **Liberdade de desenvolvimento**

O fato de APIs gráficas como o OpenGL não trazerem embarcadas ferramentas de áudio, de modelagem 3D e de controles pode ser considerado uma faca de dois gumes. Por um lado, o desenvolvedor que estiver mais focado na entrega ou nos aspectos criativos do jogo precisará investir certo tempo procurando e integrando ferramentas externas (algo que não precisaria fazer se utilizasse um motor gráfico), por outro, o desenvolvedor que estiver interessado em aprender mais profundamente sobre desenvolvimento de jogos, como no caso de alguém realizando seu trabalho de conclusão de curso, ou aquele que quiser modificar/implementar suas próprias

ferramentas, encontrará no OpenGL o cenário ideal para trabalhar.

- **Especificação aberta**

Diferente da grande maioria dos motores gráficos, o OpenGL é uma ferramenta *Open Specification*. Este artigo não abordará as vantagens e desvantagens de um código ou especificação serem abertos, mas recomenda o excelente trabalho de [HERON et al., 2013](#), que discute este assunto com muita honestidade. Entretanto, os motores gráficos, por possuírem código fechado, sujeitam seus usuários a constantes mudanças de política de desenvolvimento e monetização. Um grande exemplo disso é o caso do motor gráfico [Unity](#) em 2023, que anunciou uma profunda mudança na sua política de monetização e acesso ([ADRENALINE, 2023](#)), o que impossibilitou que muitos desenvolvedores pequenos conseguissem ser recompensados financeiramente pelo seu trabalho, além de afastar muitos potenciais novos usuários do motor gráfico.

Capítulo 2

Desenvolvimento

Este trabalho de conclusão de curso trata-se de uma evolução do trabalho, também de conclusão de curso, realizado em 2022 pela Giulia Cunha de Nardi (disponível neste [website](#)). O jogo desenvolvido pela Giulia e pelo Prof. Paulo, orientador dos dois projetos, teve como proposta, segundo a autora, "o desenvolvimento de um jogo criando uma engine própria ... e teve como ponto de partida a jogabilidade do mini-jogo Rapunzel". Como resultado a autora conseguiu produzir um "jogo puzzle 3D com uma mecânica e lógica similar ao jogo-base, mas com elementos inovadores como tipos diferentes de blocos e formas variáveis de se completar uma fase.". Apesar da boa qualidade final do projeto original (ver Figura 2.1), havia espaço para melhoramento e expansão: na seção 2.3, a mudança da forma como os gráficos do jogo são renderizados é explicada em detalhes; na seção 2.4, os novos modos de jogo e funcionalidades são descritas e exemplificadas.



Figura 2.1: *O Pesadelo de Fluffy*, desenvolvido em 2022 por Giulia de Nardi.

2.1 Estrutura do projeto

Escrito na linguagem C++, o jogo foi arquitetado utilizando orientação de objetos, de forma que a maioria dos elementos que aparecem em tela sejam subclasses da classe "Entidade", que possui atributos e métodos que controlam a posição, velocidade, rotação e

estado do objeto. *Player* (jogador), *Block* (bloco) e *MiniFluffy* são exemplos de subclasses desta classe entidade. Além destas, há também as classes "Torre" e "Andar", cujos métodos e atributos representam a torre a ser enfrentada em uma fase, que é composta por diversos andares de blocos. Na figura 2.2, temos uma esquematização simplificada da estrutura do jogo em UML (Unified Modeling Language).

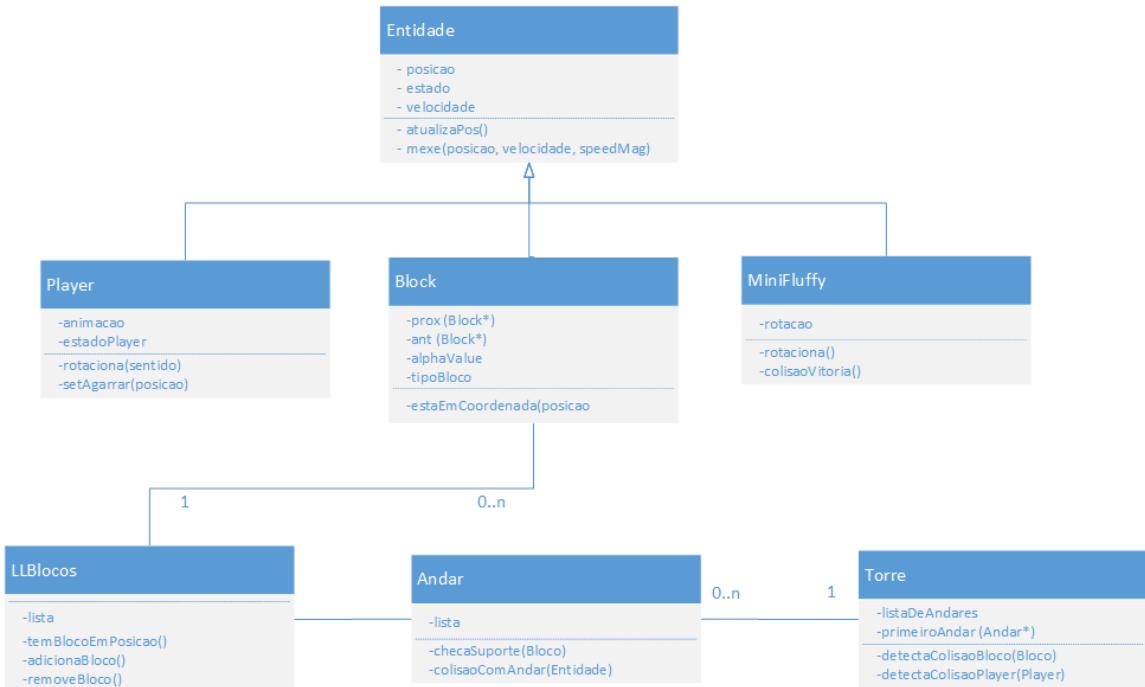


Figura 2.2: Esquematização UML do projeto (simplificada).

Pela esquematização acima, nota-se que há duas relações de natureza 0..n para 1, as relações entre *Block* e *LLBlocos* e entre *Torre* e *Andar*. Estas relações são a estrutura de listas ligadas e a relação entre torre e andares presentes no projeto, que provêm as relações e interações entre os blocos. Sempre que algo acontece com um bloco, o objeto da classe *LLBlocos* se atualiza para representar a nova estrutura da fase, por exemplo, quando um bloco é empurrado ou puxado para fora do mapa, tanto a lista de blocos deve se atualizar como o posicionamento dos andares, caso isso gere uma queda em cadeia de outros blocos.

Outras estruturas importantes do jogo também merecem atenção, como:

- **Câmera**

Para melhor controle da câmera, foi implementada uma classe definida em "camera.h" que define um objeto com posição, *yaw*, *pitch*, zoom e velocidade, este último caso ela esteja em movimento. Os controles são discutidos em breve, mas esta classe possui métodos para calcular os vetores necessários para se configurar a câmera em um ambiente 3D, como os vetores *front*, *up*, e *lookAt*. Esta classe (e todas as outras do projeto) utiliza funções da biblioteca *glm*, ou OpenGL Mathematics, para realizar cálculos geométricos. A figura 2.3 demonstra os eixos de rotação que a

câmera do jogo possui.

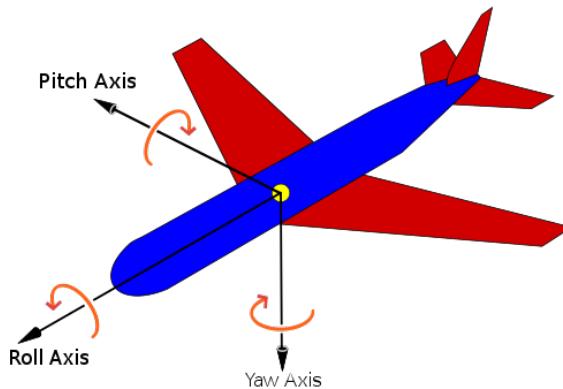


Figura 2.3: Eixos de rotação de um ambiente 3D. Fonte: [Wikipedia](#).

Abaixo temos a função responsável por atualizar todos os vetores necessários para o funcionamento correto da câmera, que é chamada sempre que há uma mudança de posição ou rotação deste objeto.

Programa 2.1 Função updateCameraVectors() da classe Camera.

```

1 // calcula o vetor Front (eixo frontal de visão), Right e Up a partir dos
2 // ângulos de Euler atualizados da câmera
3 void updateCameraVectors()
4 {
5     glm::vec3 front;
6     front.x = cos(glm::radians(Yaw)) * cos(glm::radians(Pitch));
7     front.y = sin(glm::radians(Pitch));
8     front.z = sin(glm::radians(Yaw)) * cos(glm::radians(Pitch));
9     Front = glm::normalize(front);
10    Right = glm::normalize(glm::cross(Front, WorldUp)); .
11    Up = glm::normalize(glm::cross(Right, Front));
12 }
```

- **Player**

O objeto da classe *Player* pode ser considerado o mais completo, ou complexo, do jogo. Só de possíveis estados, ele possui três atributos, um deles comum entre as entidades (em movimento, parado ou caindo), outro exclusivo (morto, pendurado, tentando se pendurar, etc.) e o último que define seu estado de animação (normal, pendurado, empurrando, puxando, etc.). Além dele, apenas a câmera e os MiniFluffys (objetos exclusivos do modo de jogo MiniFluffys) sofrem movimentos de rotação, e por isso o *Player* possui funções e variáveis próprias para possibilitar o controle pelo usuário e a manutenção de informações.

2.2 Controles e Bibliotecas

Além do OpenGL, a API que fornece a interação entre aplicação, CPU e GPU, outras ferramentas são utilizadas para fornecer os controles, interface de áudio, funções matemáticas e inicialização de bibliotecas. Especificamente, para controle de áudio e captação dos movimentos de mouse e teclado, o jogo utiliza o [SDL](#), ou Simple DirectMedia, que é uma biblioteca desenhada para fornecer acesso *low-level* (dados brutos) deste tipo de informação fornecida diretamente do sistema operacional para a aplicação que a utiliza. De forma simplificada, por exemplo, ao ser instanciado um objeto da classe *SDLKeyboardEvent*, ele passa a capturar o acionamento do teclado, e permite à aplicação acessar estes dados. No exemplo a seguir, temos um trecho da função *handleKey()*, responsável por lidar com acionamentos do teclado no jogo; neste trecho, ao apertar as teclas "w", "a", "s"ou "d", a função *ProcessKeyboard* do objeto do tipo *camera* é chamada, o que resulta num movimento da câmera.

Programa 2.2 Trecho da função handleKey().

```

1  switch (key->keysym.sym)
2  {
3  ...
4  case SDLK_w:
5      camera.ProcessKeyboard(FORWARD, speedMult);
6      break;
7  case SDLK_s:
8      camera.ProcessKeyboard(BACKWARD, speedMult);
9      break;
10 case SDLK_a:
11     camera.ProcessKeyboard(LEFT, speedMult);
12     break;
13 case SDLK_d:
14     camera.ProcessKeyboard(RIGHT, speedMult);
15     break;
16 ...
17 }
```

Outras bibliotecas e ferramentas externas notáveis utilizadas no projeto são:

- [GLAD](#)

Para fornecer ao OpenGL os endereços das funções da GPU, o projeto utiliza o *loader* GLAD, que os fornece baseado nas necessidades específicas do desenvolvedor/aplicação.

- [GLM](#)

O GLM, ou OpenGL Mathematics fornece quase todas as funções geométricas e matemáticas utilizadas pela aplicação, além de objetos úteis para comunicação com os *shaders*, como matrizes e vetores.

2.3 Modo de renderização

O maior e principal objetivo deste trabalho é reescrever por completo a forma como os gráficos da aplicação são gerados, armazenados e renderizados. De forma simples, há duas maneiras possíveis de se utilizar o OpenGL para produzir uma cena, o modo imediato e o modo retido.

- **Modo Imediato**

O modo imediato, já há muito descontinuado¹, consiste em vincular diretamente a placa gráfica do computador ao fluxo do programa. Isso significa que, dado um cenário em que, por exemplo, o programa precisa desenhar vértices na tela, o *driver* não pode solicitar à GPU que comece a renderizar os pixels antes que todos os vértices tenham sido transferidos da CPU e que o programa indique esta finalização por meio de um comando. Isso acontece porque a placa gráfica não sabe quando irá parar de receber dados, e como ela precisa processar e transferir estes dados também, ela fica presa até que a CPU "a libere" para trabalhar.

De forma prática, o modo imediato consiste em 4 passos:

1. O programa chama o comando *glBegin*, que instrui o *driver* que ele começará a desenhar um ponto, uma linha ou um polígono.
2. O programa atribui valores às variáveis que serão copiadas para a GPU, como *glTexCoord* (coordenada de texturas), *glColor* e *glNormal* (descreve o vetor normal, importante para renderização correta de superfícies sob diferentes condições de iluminação).
3. Com o comando *glVertex*, o programa reúne os valores atribuídos no passo 2 com os dados de posição informados.
4. O programa chama o comando *glEnd*, e os dados são transferidos, através do *driver*, para a GPU.

De fato, a única vantagem que esse modo oferece é o fato dele ser muito fácil de se usar. Em aplicações extremamente simples, o modo imediato não causa desvantagens perceptíveis no produto final mas, como visto na seção de resultados, até em um jogo pequeno como o deste projeto, este modo de renderização já não é adequado.

¹ Em 2008, muitas funções e variáveis do OpenGL passaram a ser consideradas deprecadas após o lançamento da versão 3.0 da aplicação, em prol da adoção do modo retido. Estas funções deprecadas fazem parte do que hoje é considerado **Legacy OpenGL**.

• Modo Retido

– Na teoria

O modo retido pode ser entendido como uma evolução natural do modo imediato quando se trata de renderização gráfica. Este modo oferece um alto nível de abstração que permite ao usuário desenvolvedor, após as configurações iniciais, poupar-se de alguns dos detalhes da renderização dos objetos. Este modo também supera sua contraparte por possuir resultados muito mais otimizados, uma vez que utiliza de melhor maneira as capacidades computacionais da placa gráfica. De forma simplificada, o funcionamento do modo retido em OpenGL é estruturado em duas fases:

1. A placa gráfica (GPU), ao invés do cliente, retém (origem do nome "modo retido") o modelo de objetos completo das primitivas a serem renderizadas (pontos, linhas ou polígonos).
2. As chamadas da aplicação às funções GPU indicam parâmetros e *flags* à placa gráfica.

Isso significa que, no modo retido, o cliente não chama diretamente o início da renderização, e sim atualiza os modelos internos abstratos do espaço de dados da GPU. Isso permite que a biblioteca gráfica otimize², à sua melhor forma, quando a renderização de fato acontecerá ao processar os objetos (primitivas) à ela transferidos.

– Na prática

Após os processamentos de primitivas, uma grande parte do trabalho do OpenGL consiste em transformar todas as coordenadas 3D em pixels 2D, uma vez que a tela dos dispositivos pode ser pensada como uma matriz bidimensional de pixels. Este processo é gerenciado pelo que chamamos de *pipeline* gráfico do OpenGL (ver Figura 2.4). Este *pipeline* pode ser segregado em duas partes: a primeira transforma as coordenadas 3D geradas em coordenadas 2D e a segunda transforma estas últimas em pixels, com atributos de cor. Mesmo o *pipeline* gráfico pode ser dividido em diversas etapas, de forma que cada uma requer como entrada (*input*) a saída (*output*) da etapa anterior. Todas essas etapas têm uma função específica e podem, a depender da placa gráfica e possibilidade, ser executadas em paralelo. Devido à sua natureza paralela, as placas gráficas de hoje possuem milhares de pequenos núcleos de processamento para processar rapidamente seus dados dentro do *pipeline* gráfico. Os núcleos de processamento executam programas pequenos na GPU para cada etapa do pipeline. Esses programas pequenos são chamados de *shaders* (VRIES, 2020).

² Algumas técnicas de otimização incluem: gerenciamento de **double buffering**; tratamento e exclusão de superfícies que estão "escondidas", e que não precisam ser processadas; transferir e processar apenas primitivas que mudaram de um quadro para outro.

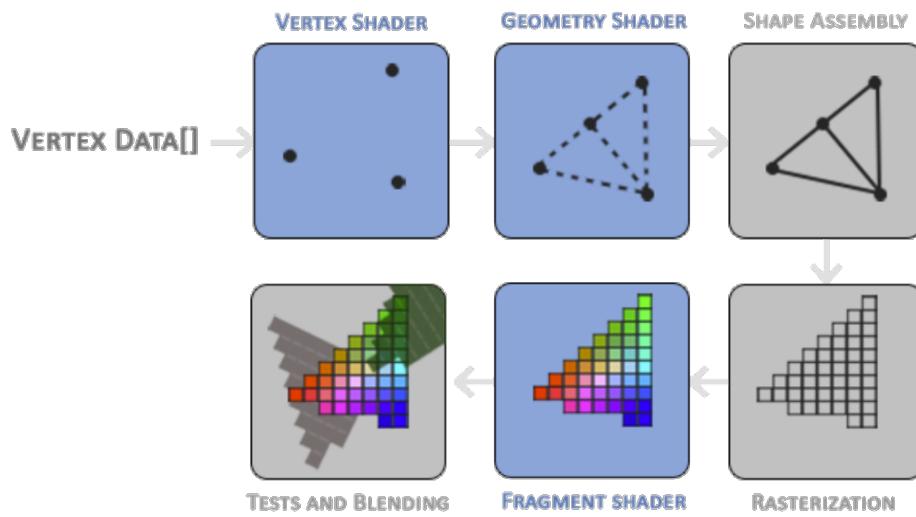


Figura 2.4: Pipeline gráfico do OpenGL. Fonte: [Learn OpenGL](#).

Neste trabalho, foi essencial refatorar as funções de produção e renderização dos vértices de todos os objetos em cena, que estavam utilizando o modo imediato, para o modo retido. Para isso, foi preciso produzir e armazenar **VBOs** e **VAOs** para todo o tipo de objeto do jogo. A grande vantagem é que isso precisa ser feito apenas uma vez (caso o objeto não sofra transformações em sua forma) durante a execução do jogo; bastando apenas solicitar à GPU que acesse estes objetos e os renderizem. Abaixo está um trecho da função *setupBlockVAO*, que realiza este trabalho para todos os blocos do jogo.

Programa 2.3 Trecho da função setupBlockVao().

```

1 void setupBlockVAO(float d)
2 {
3     // Define os vértices para todas as faces do cubo
4     float vertices[] = {
5         // positions // colors // texture coords //normals
6         -0.5f, -0.5f, -0.5f, 1.0f, 1.0f, 1.0f, 1.0f, 0.0f, 0.0f, 0.0f, -1.0f,
7         0.5f, -0.5f, -0.5f, 1.0f, 1.0f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f, -1.0f,
8         0.5f, 0.5f, -0.5f, 1.0f, 1.0f, 1.0f, 0.0f, 1.0f, 0.0f, 0.0f, -1.0f,
9         0.5f, 0.5f, -0.5f, 1.0f, 1.0f, 1.0f, 0.0f, 1.0f, 0.0f, 0.0f, -1.0f,
10        -0.5f, 0.5f, -0.5f, 1.0f, 1.0f, 1.0f, 1.0f, 0.0f, 1.0f, 0.0f, 0.0f, -1.0f,
11        -0.5f, -0.5f, -0.5f, 1.0f, 1.0f, 1.0f, 1.0f, 0.0f, 0.0f, 0.0f, -1.0f,
12        ... .};
13
14     unsigned int VBO, VAO;
15     int i;
16
17     // Multiplica os vértices pelo parâmetro dimensão do programa
18     for (i = 0; i < 36; i++){ vertices[i * 11 + 0] *= d; ... }
19
20     // Cria um vertex array e armazena sua posição em VAO e o mesmo para o VBO
21     glGenVertexArrays(1, &VAO);
22     glGenBuffers(1, &VBO);
23     glBindVertexArray(VAO);
24
25     // Bind o VBO criado no buffer GL_ARRAY_BUFFER da GPU
26     glBindBuffer(GL_ARRAY_BUFFER, VBO);
27     glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
28
29     // Atributo de posição
30     glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 11 * sizeof(float),
31                           (void *)0);
32     glEnableVertexAttribArray(0);
33
34     // ... Repete este processo para os atributos de cor, textura e normal
35
36     blockVAO = VAO;
37 }
```

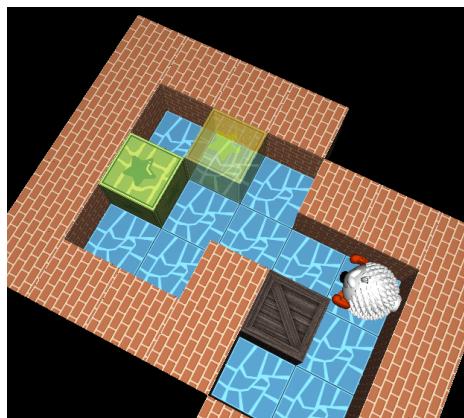
Os shaders criados para este trabalho são razoavelmente simples, mas permitiram o amplo e ótimo cálculo de vértices, a injeção de texturas, o uso de canais *alpha* em texturas e objetos e o uso de modelos de iluminação. Eles são escritos em **OpenGL Shading Language**, uma linguagem para placas gráficas baseada em C, e como têm objetivos muito específicos para o funcionamento correto do OpenGL, não cabe no escopo deste trabalho explicar seu funcionamento, mas o leitor mais curioso pode encontrá-los no **repositório** público deste projeto.

2.4 Novas funcionalidades e modos de jogo

Talvez a parte mais divertida do trabalho, novos modos de jogo e funcionalidades também foram implementadas. Originalmente, o único objetivo de Fluffy, o personagem principal do jogo, era escalar uma torre e alcançar/marcar os blocos finais em seu topo, o que o permitia avançar para a próxima fase. Agora, cada fase é dividida em diferentes estágios, cada um com um "modo de jogo" diferente, que são: *Rapunzel*, o modo original descrito anteriormente, *Sokoban* e *MiniFluffys*. Ao iniciar uma nova fase, apenas o primeiro estágio é visível e acessível por Fluffy, e só quando ele cumpre o objetivo daquela etapa, os blocos e objetivos do próximo estágio aparecem (lentamente deixam de ser transparentes) para o jogador, que vence a fase quando completa todos os estágios. O movimento da câmera também tem seu comportamento modificado a depender do estágio atual que o jogador enfrenta.

- **Sokoban**

Este modo de jogo é inspirado no jogo *Sokoban*, criado em 1981 por Hiroyuki Imabayashi. Neste jogo, o objetivo é empurrar todas as caixas de um ponto A a um ponto B, sem permitir que, ao cometer algum erro, o jogador fique preso em algum lugar ou não consiga mais organizar as caixas. O modo de jogo criado neste projeto é muito similar: na imagem comparativa abaixo, uma das caixas (que possui textura de madeira) já foi empurrada até seu objetivo, e assumiu a textura de "vitória"; a outra, entretanto não pode atingir seu objetivo, pois o jogador errou a ordem na qual deveria organizar as caixas e tornou a fase impossível de ser completada, e terá portanto que reiniciá-la.



(a) Modo de jogo *Sokoban*.



(b) *Sokoban original*.

Figura 2.5: Comparação entre o modo de jogo "Sokoban" e o jogo inspiração, de mesmo nome.

• MiniFluffys

Já o modo de jogo "MiniFluffys" é inspirado no jogo eletrônico **Krusty's Fun House**, criado em 1992 pela Virgin Games. Neste jogo, o jogador deve direcionar pequenos ratos até uma área de extermínio ao atravessar complexos labirintos. O jogador controla o personagem Krusty, da série de televisão "Os Simpsons", e deve utilizar objetos e obstáculos para criar caminhos para os ratos. O modo de jogo "MiniFluffys" criado tem um propósito similar: nele, o jogador deve organizar blocos direcionais ("esquerda" ou "direita") para alterar o movimento dos MiniFluffys, pequenas versões de si que estão presos no ambiente, até a porta de saída. Sem interagir com os blocos, os pequenos fluffys apenas andam em linha reta.



(a) Modo de jogo *MiniFluffys*.



(b) Jogo "*Krusty's Fun House*".

Figura 2.6: Comparação entre o modo de jogo "MiniFluffys" e o jogo inspiração para o modo.

Funcionamento

Neste modo de jogo, temos três tipos de blocos importantes, além dos minifluffys. Estes são:

– Bloco de spawn

Deste bloco (ver Figura 2.7a), do qual há quatro subcategorias que determinam a direção de ação (norte, sul, leste, oeste), até cinco minifluffys são gerados em intervalos de um segundo um bloco à sua frente. Ao nascerem, os minifluffys começam a andar em linha reta.

– Bloco de direcionamento

Este bloco redireciona cada minifluffy que encontra nele ou 90° para a direita ou 90° para a esquerda, a depender da subcategoria do bloco. Ver Figura 2.7b.

– Bloco de saída

Este bloco (ver Figura 2.7c), desenhado como uma porta de saída, é tratado como a linha de chegada dos minifluffys. Ao encostarem nele (chegarem até o

bloco em linha reta), os minifluffys são considerados "escapados" e, quando todos eles escapam, o jogador avança de estágio dentro da fase.

- Bloco de parede

Este bloco, considerado no estágio como uma parede, tem o efeito de fazer com que os minifluffys realizem um rotação de 180°, ou seja, não modificam sua direção mas invertem seu sentido de movimentação. Ver figura 2.7d

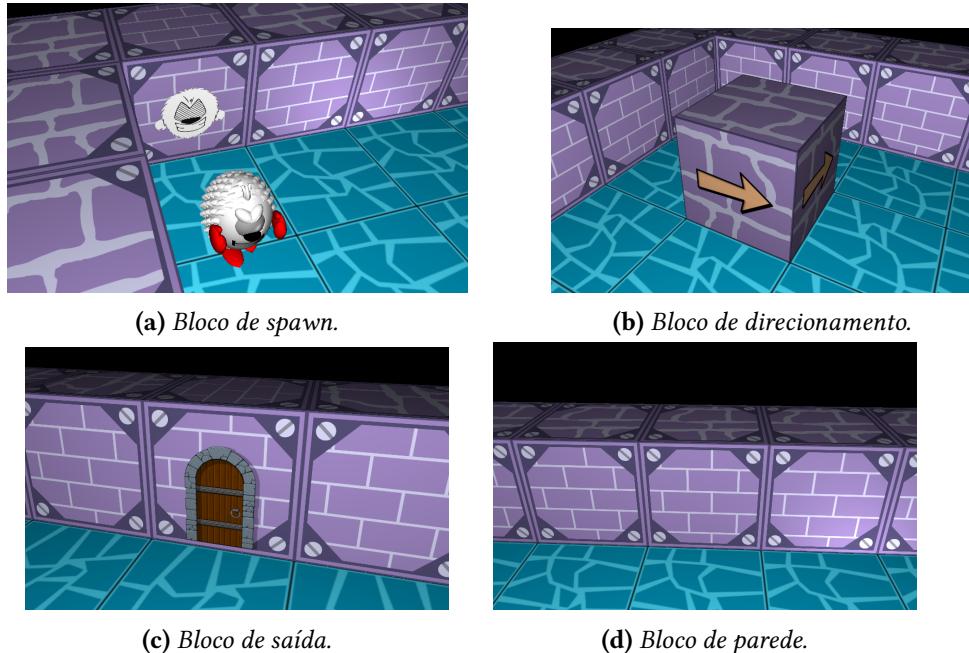


Figura 2.7: Novos blocos do modo de jogo *MiniFluffys*.

2.5 Otimização e tratamento de bugs

Além das mudanças já citadas, parte do trabalho deste projeto consiste em otimizar partes do código, seja na remoção de código e *loops* desnecessários, no carregamento e inicialização de bibliotecas e funções, no *handling* de erros ou até na correção de alguns *bugs* e erros que ocorriam. Alguns exemplos do que foi feito, dentre outros, são:

- Erros fatais das bilbiotecas e inicializadores SDL, GLM, SDLImage e GLAD agora são capturados e injetados no arquivo de *log* para facilitar a depuração no processo de desenvolvimento.
- A configuração de muitos atributos da biblioteca SDL acontecia após a criação da janela de visualização, isso fazia com que os valores atribuídos fossem descartados, o que causava certos comportamentos inesperados.
- As classes Imagem e Câmera foram criadas e suas configurações e funções segregadas do arquivo principal do jogo. Além de facilitar a modificação de comportamento, novas funções foram criadas que tornam desnecessários atalhos que foram criados

no projeto inicial, como a necessidade de inverter as imagens utilizadas no jogo para a correta renderização pelo OpenGL.

- Foi implementado tratamento de canais alpha; agora, objetos em cena podem possuir diferentes níveis de transparência.
- Criou-se um tratamento de tempo e uma variável global de multiplicador de velocidade, que permite ao usuário escolher o quanto "rápido" quer que o jogo execute.
- Foi implementada a contagem e o rastreio de *frames per second* (FPS), ou quadros por segundo, que é uma ferramenta muito útil para se mensurar a otimização e a necessidade computacional de uma aplicação como um jogo.

É importante também citar a forma como o jogo modela e carrega as fases. Para que seja possível, para o desenvolvedor ou até mesmo para o usuário, criar novas fases do jogo sem a necessidade de alterar o código-fonte dele, o tipo dos blocos e as posições do Fluffy, da câmera e de todos os blocos é feita a partir de um arquivo de texto que contém essas informações de forma estruturada e que o jogo carrega sempre que uma nova fase começa. Foi necessário para este projeto modificar a forma como o jogo interpreta este arquivo de forma a adequá-lo aos novos tipos de blocos e modos de jogo, das seguintes maneiras:

- O tipo dos blocos consistia de apenas um número (de 0 a 9). Isso era suficiente pois o jogo tinha menos que 10 tipos de blocos em sua totalidade. Foi necessário expandir este alcance, e agora o jogo espera dois números em sequência, possibilitando que sejam descritos e usados até 100 tipos de blocos diferentes.
- Após cada bloco é esperado uma vírgula (",") seguida do número (de 0 a 9) que indica o estágio da fase (ver seção 2.4) ao qual o bloco pertence.

2.6 Modelos de iluminação

Mesmo os computadores mais poderosos do mundo não conseguiram simular perfeitamente o comportamento da luz em um cenário complexo como o de um jogo de modo a produzir uma resposta rápida o suficiente para torná-lo "jogável". Para que uma simulação perfeita fosse executada, seria necessário modelar uma fonte de luz que emitisse de forma dispersa trilhões de **Fótons** todo segundo e, para cada um desses, calculasse a trajetória e as alterações físicas que neles ocorressem até que chegassem ao olho, ou câmera, da simulação. Por isso, a iluminação no OpenGL é baseada em aproximações da realidade, por meio de modelos simplificados que são processados pelo computador de forma muito mais rápida e que, apesar de mais simples, também são baseados e dependem dos conceitos de física que conhecemos. Com a implementação do modo retido, devido à possibilidade de se calcular a cor, o brilho, a opacidade e os efeitos de cada vértice, passa a ser possível implementar no projeto diversos modelos de iluminação, dos mais simples aos mais complexos.

- **Modelo de Phong**

Um dos modelos mais simples e eficazes de iluminação, amplamente utilizado

no OpenGL, é o modelo de Phong ([VRIES, 2020](#)) (ver Figura 2.8). Este modelo descreve três conceitos, ou componentes, chave de iluminação, que são:

1. Ambiente

Para que nenhum objeto fique completamente obscuro, considera-se que há sempre um resquício de luz no ambiente (e.g. uma luz distante, a lua, etc.). Para simular esta fonte de luz sempre presente e muito fraca, utiliza-se uma constante que sempre fornece um pouco de brilho aos objetos.

2. Especular

A luz especular simula os pontos fortes e brilhosos de luz que aparecem em objetos brilhantes. Estes pontos de brilho dependem mais da cor da fonte de luz do que da cor do objeto. Notavelmente, a força do brilho deste reflexo diminui a medida que o ângulo entre a visão da câmera e o ângulo "ideal" de reflexão da luz com o objeto aumenta. Isso acontece pois, a medida que este ângulo aumenta, a quantidade de micro-facetões do objeto que refletem a luz em direção à câmera diminui, o que causa essa perda de brilho ([SCRATCHPIXEL: 2022](#)).

3. Difusa

A luz difusa, a que produz resultado mais visível na imagem final, representa o impacto que uma fonte de luz tem sobre um objeto, e leva em consideração ângulo de impacto, distância, etc. Quanto mais um objeto é impactado pela fonte de luz, mais claro ele aparecerá.

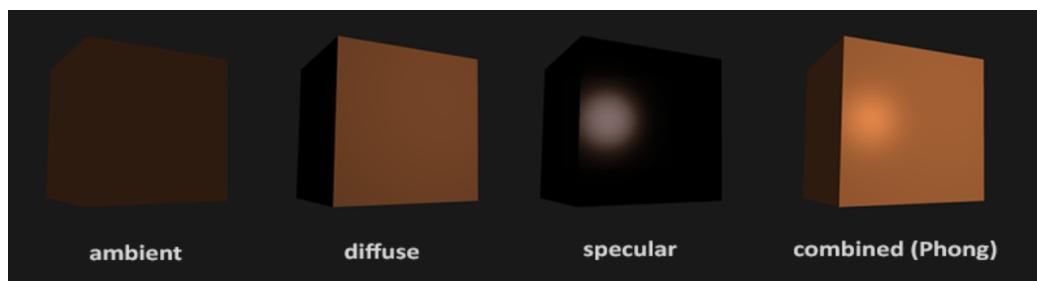


Figura 2.8: Componentes de iluminação de Phong, separados e combinados. Fonte: [Basic Lighting](#).

- **Modelo Blinn-Phong**

O modelo de Blinn-Phong (ver Figura 2.9) é uma modificação do modelo de reflexão de Phong. Ao usar o cosseno do ângulo entre o vetor normal e o **halfway vector** para calcular a direção da reflexão dos raios de luz simulados, o modelo de Blinn-Phong consegue reproduzir melhor o comportamento da reflexão especular. Uma visível melhora deste modelo sobre o anterior é que no modelo de Phong, o brilho especular é sempre redondo em uma superfície plana. No modelo de Blinn-Phong, o brilho especular é redondo quando a superfície é vista de frente, e

alongado verticalmente quando a direção de visão e a direção à fonte de luz estão rentes à superfície (BATAGELLO e MARQUES, 2021).

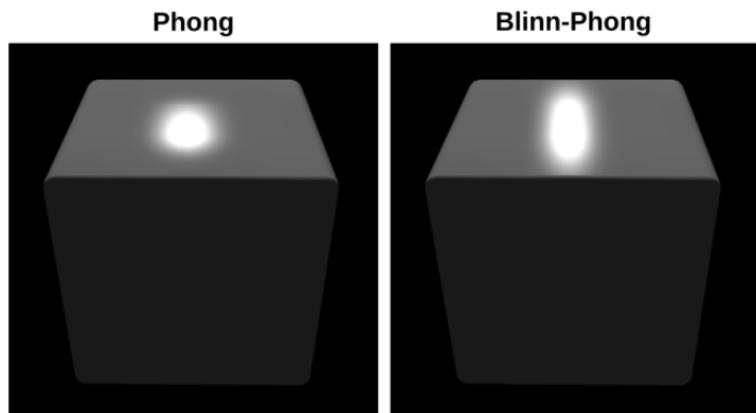


Figura 2.9: Diferença entre o formato do brilho espelhado nos modelos de Phong e Blinn-Phong. Fonte: *Computação Gráfica*.

- **Ray tracing**

Apesar de ter ganhado muita notoriedade nos últimos anos devido às novas tecnologias e aos avanços no poder computacional de placas que a permitem ser melhor utilizada, a técnica de síntese de imagens "ray tracing", ou rastreamento de raios (de luz), já é conhecida na academia desde meados da década de 1970. O artigo de GLASSNER, 1989, intitulado "An Introduction to Ray Tracing" já descreve em detalhes os métodos computacionais necessários para produzir imagens com este método, chamado de *forward ray tracing*: nele, consideramos uma fonte de luz, que possuí intensidade, cor e possivelmente direcionamento dos raios; a partir dessa fonte de luz, simulamos x raios que dela surgem e os rastreamos, levando em consideração suas interações com os objetos em cena, como refração, reflexão, etc., até que (alguns) atinjam o olho, ou câmera, do cenário e, após processamento final, resultem em um *pixel* em tela. Um dos grandes problemas desse método, segundo Glassner, é que muitos desses raios, ou fóttons, simplesmente não atingem a câmera da cena, e computacionalmente não faz sentido e é inviável calcular estes dados "inúteis".

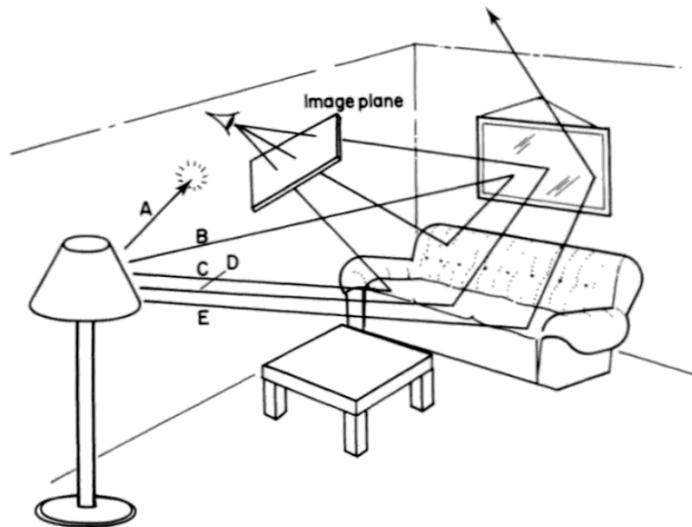


Figura 2.10: Representação da técnica *forward ray tracing*. Note que alguns raios que emanam da fonte de luz nunca atingem a câmera. (GLASSNER, 1989)

Hoje em dia, de forma prática, outro método de rastreamento de raios é utilizado por ser mais rápido em gerar imagens. Neste método, o algoritmo constrói uma imagem ao "lançar" raios a partir do olho (câmera) da cena, que após interceptar um *pixel* do quadro da imagem (ver Figura 2.11), reflete nos objetos e nas superfícies em direção à alguma fonte de luz inserida na cena. Assim, ele consegue aproximar a cor e a intensidade daquele *pixel* ao ponderar as características dos elementos dos quais o raio de luz lançado refletiu

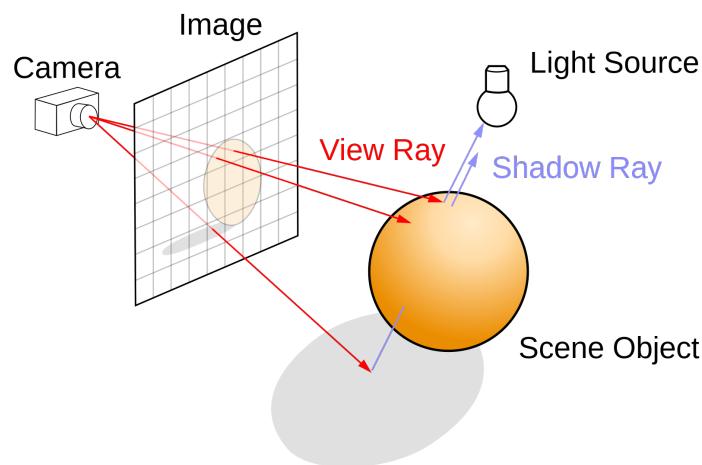


Figura 2.11: Diagrama de traçamento de raios. Fonte: Wikipedia.

Embora seja perfeitamente possível implementar esta técnica no OpenGL (BROWNLEE *et al.*, 2012), é mais recomendável utilizar alguma linguagem de GPGPU (Unidade de Processamento Gráfico de propósito geral), como CUDA ou OpenCL. Uma das

dificuldades ao tentar utilizar o *ray tracing* com o OpenGL é que, para implementar este método, é necessário calcular a interação dos raios de luz simulados com a geometria dos objetos em cena. Para isso, é necessário que os *shaders* tenham acesso à geometria dos objetos, algo nativamente não disponível no OpenGL, o que força o desenvolvedor a injetar estes dados dentro das estruturas descritivas dos objetos. Mesmo assim, com a implementação do modo retido, é possível desenvolver o *ray tracing* neste projeto em alguma versão futura.

Capítulo 3

Resultados e Conclusões

Além da adequação às melhores práticas, a mudança do modo imediato para o modo retido resulta também em incríveis ganhos de performance. Para além da melhor experiência de usuário ao jogar o jogo, este aumento de performance do jogo permitiu o desenvolvimento de funcionalidades mais complexas no projeto, como o modo de jogo MiniFluffys, em que diversos personagens móveis são renderizados e têm seus movimentos continuamente simulados em tempo real. Para formalizar estes ganhos de desempenho, foram realizados testes de *frames* por segundo (FPS), todos no mesmo computador e condições¹:

Número de quadros por segundo

- Jogo original

- Máximo: 27.65
- Mínimo: 26.12
- Médio: 21.47

- Nova versão

- Máximo: 127.18
- Mínimo: 98.66
- Médio: 123.19

Estes resultados demonstram um aumento médio de aproximadamente **6 vezes** no número de quadros por segundo em uma fase similar entre as versões do jogo.

Além disso, as novas modalidades de jogo e fases criadas também devem ser levadas em consideração. Com os modos de jogo Catherine, Sokoban e MiniFluffys, foi possível criar criativas e complexas novas fases de quebra-cabeças que devem desafiar e divertir os jogadores. Na Figura 3.1 está uma visão superior de uma fase do jogo que inclui todos os três modos de jogo.

¹ Especificações relevantes da máquina: processador AMD Ryzen 7 5800X 3100Mhz de 8 núcleos; 23.1Gbs de memória total disponível; placa de vídeo NVIDIA GeForce RTX 3070; sistema operacional Ubuntu 22.04.3 LTS

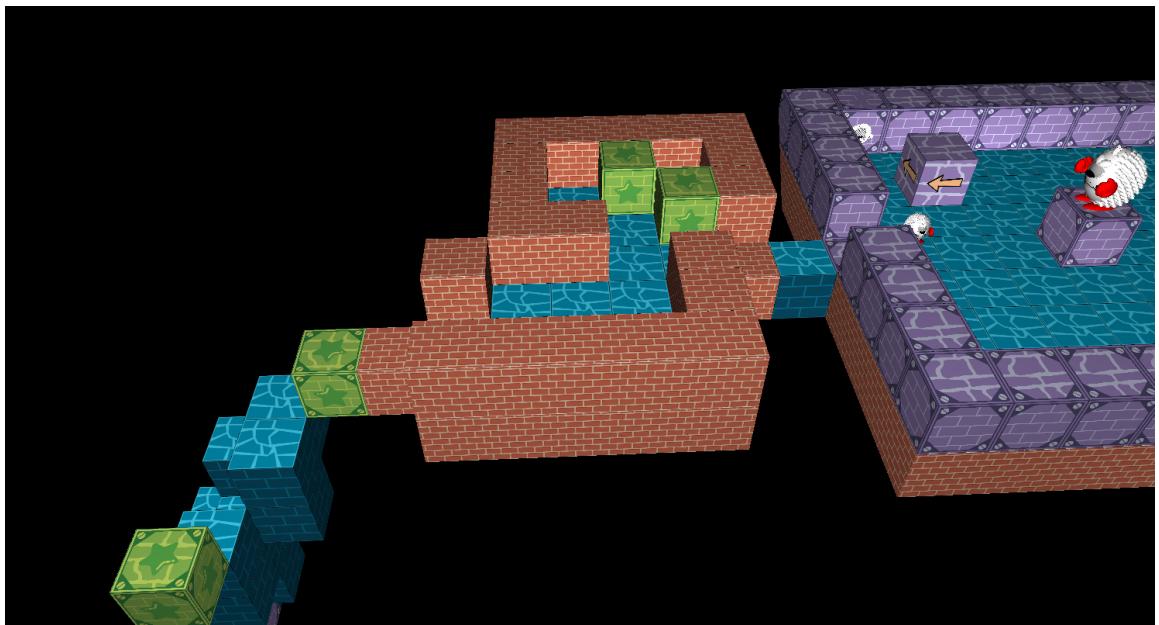


Figura 3.1: Exemplo de fase do jogo que inclui todos os novos modos de jogo.

Portanto, em se tratando das propostas do trabalho, que são a refatoração e atualização do código, os ganhos de performance e a criação de novos modos de jogo, pode se considerar que o projeto foi bem sucedido. Sem dúvidas, a parte mais trabalhosa e difícil desse projeto foi a completa reescrita dos modos de desenho dos objetos em cena: transicionar do modo imediato para o modo retido de renderização requer, de certa forma, repensar a forma como o seu código lida com a geometria dos objetos e entender mais profundamente as relações entre CPU, memória e GPU. Embora hajam bibliotecas e tutoriais extensivos disponíveis na Internet, um intenso trabalho de tentativa e erro foi necessário para resolver problemas e *bugs* persistentes. Dito isso, várias partes do trabalho foram também muito prazerosas e divertidas, como a criação de quatro novas fases, o desenvolvimento de novos modos de jogo e os *user tests* com amigos e familiares.

Referências

- [AL-ABSI 2017] Mohammad AL-ABSI. “The effect of using puzzles and games on students’ mathematical thinking at the faculty of educational sciences and arts (unrwa)”. *An-Najah University Journal for Research-B (Humanities)* 31.10 (2017), pp. 1867–1888 (citado na pg. 6).
- [ADRENALINE 2023] ADRENALINE. *Mudanças polêmicas nas políticas da Unity foram criadas às pressas, afirma reportagem.* <https://www.adrenaline.com.br/games/mudancas-polemicas-nas-politicas-da-unity-foram-criadas-as-pressas-afirma-reportagem/>. 2023 (citado na pg. 8).
- [BATAGELLO e MARQUES 2021] Harlen BATAGELLO e Bruno MARQUES. *Computação Gráfica.* <https://www.brunodorta.com.br/cg/>. 2021 (citado na pg. 22).
- [BROWNLEE *et al.* 2012] Carson BROWNLEE, Thomas FOGAL e Charles D HANSEN. “Glu-ray: ray tracing in scientific visualization applications using opengl interception”. In: *Eurographics Symposium on Parallel Graphics and Visualization*. Vol. 3. 2012 (citado na pg. 23).
- [CONSALVO 2022] Mia CONSALVO. *Atari to Zelda: Japan’s videogames in global contexts.* MIT Press, 2022 (citado na pg. 3).
- [COSTA e MIRANDA 2017] Lucas D Gonçalves da COSTA e Jéssica C Campos MIRANDA. “Ensinando com portais: ensinando o jogo portal 2 em aulas de física” (2017) (citado na pg. 5).
- [FLEURY *et al.* 2014] Afonso FLEURY, Davi NAKANO e José DEL OSSO. “Mapeamento da indústria brasileira e global de jogos digitais” (2014) (citado na pg. 1).
- [GLASSNER 1989] Andrew S GLASSNER. *An introduction to ray tracing.* Morgan Kaufmann, 1989 (citado nas pgs. v, 22, 23).
- [HERON *et al.* 2013] Michael HERON, Vicki L HANSON e Ian RICKETTS. “Open source and accessibility: advantages and limitations”. *Journal of Interaction Science* 1.1 (2013), pp. 1–10 (citado na pg. 8).
- [NBC 2023] NBC. *Maryland school uses video games to teach coding.* <https://www.nbcwashington.com/news/local-school-uses-video-games-to-teach-coding/3413271/>. 2023 (citado na pg. 6).

- [NPR 2010] NPR. *School Uses Video Games To Teach Thinking Skills*. <https://www.npr.org/2010/06/28/122800333/school-uses-video-games-to-teach-thinking-skills>. 2010 (citado na pg. 6).
- [PAGE 2016] Robert PAGE. *17 Sub-Genres of Platformer Games*. <https://web.archive.org/web/20190109062140/sub-genres-of-platformer-games-list.html>. 2016 (citado na pg. 3).
- [PCGAMINGWIKI 2023] PCGAMINGWIKI. *List of OpenGL games*. <https://www.pcgamingwiki.com>. 2023 (citado na pg. 7).
- [SCRATCHPIXEL: 2022] SCRATCHPIXEL: *The Phong Model, Introduction to the Concepts of Shader, Reflection Models and BRDF*. <https://www.scratchapixel.com/lessons/3d-basic-rendering/phong-shader-BRDF/phong-illumination-models-brdf.html>. 2022 (citado na pg. 21).
- [VON DER HEIDEN *et al.* 2019] Juliane M VON DER HEIDEN, Beate BRAUN, Kai W MÜLLER e Boris EGLOFF. “The association between video gaming and psychological functioning”. *Frontiers in psychology* 10 (2019), p. 1731 (citado na pg. 6).
- [VRIES 2020] J. de VRIES. *Learn OpenGL: Learn Modern OpenGL Graphics Programming in a Step-by-step Fashion*. Kendall & Welling, 2020. ISBN: 9789090332567. URL: <https://books.google.com.br/books?id=koWWzQEACAAJ> (citado nas pgs. 14, 21).