Curso de Ciência da Computação
Universidade Estadual de Mato Grosso do Sul

Um estudo da mecânica e arquitetura de motores de jogos

Lucas Câmara Otechar Sanches

MSc. Diogo Fernando Trevisan (Orientador)

Dourados - MS
2015

Curso de Ciência da Computação
Universidade Estadual de Mato Grosso do Sul

Um estudo da mecânica e arquitetura de motores de jogos

Lucas Câmara Otechar Sanches

Novembro de 2015

Banca examinadora:

Prof. MSc. Diogo Fernando Trevisan (Orientador)
Área de Computação - UEMS

Profa. MSc. Jéssica Bassani de Oliveira
Área de Computação - UEMS

Profa. Dra. Mercedes Rocío Gonzales Márquez
Área de Computação - UEMS

Um estudo da mecânica e arquitetura de motores de jogos

Lucas Câmara Otechar Sanches

Este exemplar corresponde à redação final da monografia da disciplina Projeto Final de Curso devidamente corrigida e defendida por Lucas Câmara Otechar Sanches e aprovada pela Banca Examinadora, como parte dos requisitos para a obtenção do título de Bacharel em Ciência da Computação.

Dourados, 23 de novembro de 2015.

Prof. MSc. Diogo Fernando Trevisan

(Orientador)

AGRADECIMENTOS

Acima de tudo à meus pais, Dirceu Sanches Junior e Andréa Câmara Otechar Sanches, que estiveram sempre presentes, acreditando e investindo em todas as escolhas que fiz durante minha vida.

À meu orientador, Prof. MSc. Diogo Fernando Trevisan, por sua disponibilidade, apoio e conhecimento que foram inestimáveis para a finalização deste projeto, e que me ajudaram nas decisões mais difíceis.

Ao Prof. MSc. Delair Osvaldo Martinelli Júnior, que me orientou anteriormente e abriu portas para que este trabalho pudesse ser continuado.

À todos os professores, autores e programadores que de alguma forma influenciaram em meu aprendizado.

À meus amigos, colegas, e todos que direta ou indiretamente me apoiaram e motivaram durante o desenvolvimento deste projeto.

Lucas Câmara Otechar Sanches

RESUMO

O mercado de jogos eletrônicos está se tornando cada vez mais popular, e com seu sucesso cresce também o número de interessados em seguir carreira no ramo de desenvolvimento de jogos. Entre as diversas profissões envolvidas nesta indústria está o programador, e uma de suas capacitações é saber escolher ou desenvolver um motor de jogos adequado, parte fundamental que fornece os componentes vitais para o bom funcionamento dos jogos. No geral, recomenda-se que desenvolvedores utilizem um motor já pronto, pois são de fácil reutilização, mas o programador pode se ver sob a necessidade de desenvolver seu próprio motor ou interessado em aprender mais sobre o assunto. Este trabalho busca reunir e apresentar o conhecimento básico necessário para entender como funcionam os motores de jogos e seus principais componentes, além de implementar um *framework* que pode ser utilizado na criação de jogos simples ou como material de estudo, visando ampliar a indústria de jogos em território brasileiro.

Palavras-chave: jogos, motor de jogos, desenvolvimento de jogos

ABSTRACT

The video game market is becoming increasingly popular, and with its success, the number of people interested in pursuing a career on video game development also grows. Programmers are one of the several kinds of developers involved in this industry, and one of their capabilities is being able to choose or develop a suitable game engine, the fundamental piece that provides vital components necessary for the proper operation of games. In general, it's recommended that developers use an already existing engine, because they can be easily reused, but the programmer may find himself in need to develop his own engine or interested in learning more about the topic. This work seeks to gather and provide the basic knowledge necessary to understand how a game engine and its most important components works, and also to implement a framework that can be used to create simple games or as an educational material, intending to amplify the brazilian game industry.

Keywords: games, game engine, game development

SUMÁRIO

LISTA DE SIGLAS

3D: Três Dimensões ou Tridimensional

AABB: Caixa Delimitadora Alinhada a Eixos (do inglês *Axis-Aligned Bounding Box*)

API: Interface de Programação de Aplicações (do inglês *Application Programming Interface*)

BMP: Arquivo de Mapa de *bits* (do inglês *Bitmap File*)

COLLADA: Arquivo de Atividade de Projeto Colaborativa (do inglês *Collaborative Design Activity File*)

FreeGLUT: Conjunto de Ferramentas Utilitárias Livre para *OpenGL* (do inglês *Free OpenGL Utility Toolkit*)

GLEW: Vaqueiro de Extensões para *OpenGL* (do inglês *OpenGL Extension Wrangler*)

IDE: Ambiente de Desenvolvimento Integrado (do inglês *Integrated Development Environment*)

INI: Arquivo de Inicialização (do inglês *Initialization File*)

JSON: Notação de Objetos de *JavaScript* (do inglês *JavaScript Object Notation*)

OBB: Caixa Delimitadora Orientada (do inglês *Oriented Bounding Box*)

OBJ: Arquivo de Objeto *Wavefront* (do inglês *Wavefront Object File*)

OpenGL: Biblioteca Gráfica Aberta (do inglês *Open Graphics Library*)

PC: Computador Pessoal (do inglês *Personal Computer*)

PNG: Gráficos de Rede Portáveis (do inglês *Portable Network Graphics*)

RGBA: Vermelho, Verde, Azul, Alfa (do inglês *Red, Green, Blue, Alpha*)

SDK: Kit de Desenvolvimento de *Software* (do inglês *Software Development Kit*)

STL: Biblioteca Padrão de Gabaritos (do inglês *Standard Template Library*)

XML: Linguagem de Marcação Extensível (do inglês *Extensible Markup Language*)

YAML: YAML Não é uma Linguagem de Marcação (do inglês *YAML Ain't Markup Language*)

ZIP: Arquivo *ZIP* (do inglês *ZIP File*)

LISTA DE FIGURAS

Página

LISTA DE TABELAS

Página

LISTA DE CÓDIGOS-FONTE

CAPÍTULO 1

INTRODUÇÃO

Em 2013, a indústria de jogos alcançava mais uma conquista: o jogo eletrônico *Grand Theft Auto V*, desenvolvido pela *Rockstar Games*, entrou para o *Guiness Book* como a "maior receita gerada por um produto de entretenimento em 24 horas". [1] Arrecadando um valor de 815.7 milhões de dólares, *Grand Theft Auto* não só marcou história por ter superado seus concorrentes, mas sim toda a indústria de entretenimento — inclusive a cinematográfica.

Diversas outras empresas possuem histórias semelhantes para somar à crescente pilha de vitórias envolvendo jogos eletrônicos nos últimos anos. [2] A popularização deste tipo de mídia está cada dia mais aparente e já é considerado o mercado de entretenimento mais importante da atualidade.

Com este sucesso e rápido crescimento, é natural que surjam novas profissões e outras sejam relacionadas aos jogos, como por exemplo, artistas de animação, engenheiros de áudio, designers de jogos e programadores de jogos. [3] Cada uma destas áreas de conhecimento possui suas peculiaridades e dificuldades, e portanto conhecimento e experiência são essenciais para exercê-las. Em particular, uma das dificuldades geralmente enfrentadas logo no início do aprendizado para programadores de jogos é relacionada ao desenvolvimento e uso de motores de jogos (do inglês *game engines*).

O termo "motor de jogos" se refere a qualquer ferramenta que forneça ao desenvolvedor as funcionalidades intrínsecas de um jogo, como receber informações de um controle, reproduzir sons e desenhar objetos na tela, de forma a permitir uma melhor separação destas funcionalidades vitais das características que variam entre diferentes jogos, como música, objetos, roteiro e cenários. [4] Esta separação abstrai dos programadores e demais desenvolvedores os

complexos componentes que estão presentes de forma semelhante em praticamente todos os jogos atuais. Isso traz a vantagem de ser necessário desenvolver o motor apenas uma vez para que diversos jogos diferentes possam ser produzidos em cima dele. Além disso, os desenvolvedores podem focar nas partes flexíveis do jogo que estão desenvolvendo, sem precisar de uma equipe especializada para implementar todos os componentes de um motor para cada novo jogo produzido. Outra vantagem é que empresas podem desenvolver motores de jogos especializados com a finalidade de vender licenças de uso destas ferramentas para outras empresas, como é o caso da *Unity*, ou até mesmo vender licenças de motores de jogos utilizados em seus próprios jogos, como é feito com a *Unreal Engine*, *Source* e *CryENGINE*. [5]

Com tantas opções de motores de jogos disponíveis no mercado, algumas inclusive de código aberto e livre de custos, como a *Spring*, [6] a comunidade prontamente recomenda usar um motor já existente e bem testado ao invés de desenvolver um próprio. No entanto, enquanto esta recomendação se prova válida na maioria dos casos, existem algumas situações onde é preferível implementar um motor de jogos próprio à terceirizá-lo. [7] Motivos variam entre a empresa não possuir capital para investir em um motor de jogos de ponta, preferir ter uma ferramenta bem conhecida pela equipe, cobrir requerimentos de um jogo que não estão presentes em nenhum motor existente, entre outros. Além disso, o programador pode querer se especializar no desenvolvimento de motores de jogos, e como a preferência é usar um motor de jogos já pronto, pode se deparar com escassez de material de aprendizado voltado para iniciantes, principalmente em uma língua como o português brasileiro, em um momento onde a carga tributária sobre jogos eletrônicos no país é alta, falta incentivo e reconhecimento, e as empresas brasileiras de jogos ainda estão em fase de crescimento. [8]

1.1 Objetivos

Este trabalho tem como objetivo realizar um estudo da arquitetura empregada em motores de jogos, assim como o funcionamento de seus componentes mais importantes, e com base nesta pesquisa desenvolver um *framework* simples para que sirva de material de estudo à qualquer programador que tenha necessidade ou interesse em entender a mecânica e arquitetura de motores de jogos.

1.1.1 Objetivos específicos

Os objetivos específicos são:

- Estudar o funcionamento de diferentes componentes empregados em um motor de jogos, assim como sua arquitetura;
- Projetar e desenvolver um *framework* simples implementando os componentes mais importantes de um motor de jogos;
- Fornecer uma documentação para auxiliar no uso do *framework* desenvolvido;
- Desenvolver um jogo simples para demonstrar um caso de uso do *framework* desenvolvido;
- Realizar testes de performance sobre os métodos implementados.

1.2 Justificativa e motivação

Programadores aspirantes a uma carreira na área de jogos geralmente são indicados à utilizar um motor de jogos de terceiros, assim como diversos profissionais experientes e empresas deste mercado, que preferem comprar uma licença de uso para uma ferramenta já existente. [7] Porém, em algumas situações, o desenvolvimento de um novo motor é inevitável, e o programador pode ter a necessidade ou interesse de aprender mais sobre este tipo de ferramenta. Este trabalho busca fornecer o conhecimento básico necessário para a implementação de motores de jogos, especialmente em língua portuguesa, onde material sobre o tema abordado é escasso, visando impulsionar a indústria de jogos no país.

1.3 Metodologia

Durante o estudo serão realizadas leituras e consultas a livros, artigos e websites escritos sobre o tema abordado, buscando conhecer técnicas atuais para implementação dos requisitos de motores de jogos.

A ferramenta será escrita na linguagem C++, por sua performance e proximidade com o *hardware*, e terá como plataforma alvo o sistema operacional *Microsoft Windows* 8.1. O *software* será projetado na forma de *framework* contido em uma biblioteca dinâmica, e será distribuído com um projeto para a IDE *Code::Blocks* com todas as dependências inclusas. O código-fonte fará uso das bibliotecas *OpenGL*, *GLU*, *FreeGLUT*, *GLEW*, *libpng* e *zlib*, além da biblioteca padrão da linguagem C++, e será compilado com a *GNU Compiler Collection*. O projeto estará sob controle de versão utilizando as ferramentas *git* e *GitHub for Windows*, hospedado em um repositório privado em *GitHub*. A documentação será gerada com a ferramenta *DoxyGen*.

O desenvolvimento e testes serão realizados em uma máquina com sistema operacional *Microsoft Windows 8.1* 64bit, processador de 3.5~4.2 GHz × 4, memória RAM de 16 GB e placa de vídeo de 1072~1137 MHz, 256bit e VRAM de 2048 MB.

1.4 Organização do texto

Este trabalho está organizado em cinco capítulos. O primeiro capítulo dá uma breve introdução ao tema abordado. O segundo capítulo apresenta a teoria sobre motores de jogos, seus componentes, arquitetura e funcionamento. O terceiro capítulo contém o projeto de *software* e o processo de implementação da ferramenta desenvolvida. O quarto capítulo apresenta um caso de uso sobre a ferramenta, os testes realizados e seus resultados. O quinto capítulo apresenta a conclusão deste trabalho, as dificuldades enfrentadas e trabalhos futuros.

CAPÍTULO 2

FUNDAMENTAÇÃO TEÓRICA

2.1 Jogos eletrônicos

A história dos jogos eletrônicos, ou *video games*, teve início em 1958 com uma impressionante invenção de William Higinbotham. [9] Para uma exibição no laboratório *Brookhaven National Laboratory*, onde trabalhava, William teve a ideia de fazer algo diferente, que não fosse uma simples exibição estática. Ele faria um display interativo contendo um campo de tênis e uma bola.

William juntou-se com Robert V. Dvorak, e eles montaram um computador analógico ligado à um osciloscópio. [9] Ao pressionar um botão, dois jogadores poderiam rebater a bola, que saltava realisticamente em contato com o campo ou a rede. Em três semanas finalizaram o projeto conhecido atualmente como *Tennis for Two*, considerado o primeiro *video game* da história. A Figura 2.1 mostra o jogo *Tennis for Two*, à esquerda, como exibido em 1961.

Figura 2.1: *Tennis for Two* [10]

Porém, para William, sua invenção não parecia algo único e revolucionário no momento. [9] Apesar de inovador e ter sido visto por centenas de pessoas, *Tennis for Two* passou despercebido e não foi suficiente para inspirar a criação de novos jogos e iniciar uma nova indústria.

Em 1961, um estudante do *Massachusetts Institute of Technology* chamado Steve Russel desenvolveu um jogo de dois jogadores para o computador DEC PDP-1. [9] O jogo, chamado *Spacewar*, consistia em duas espaçonaves controladas pelos jogadores, que deveriam atirar torpedos para acertar o adversário. Utilizando quatro botões, cada jogador poderia atirar um torpedo, impulsionar sua nave, e rotacionar em sentido horário ou anti-horário. Novas características foram sendo adicionadas no jogo com o tempo e *Spacewar* se tornou um grande sucesso dentro da universidade, mas devido ao custo exorbitante da plataforma para o qual foi desenvolvido na época, também falhou em ser amplamente reconhecido. A Figura 2.2 mostra o *Spacewar* rodando em um PDP-1 em 1961.

Figura 2.2: *Spacewar* [11]

A situação começou a mudar em 1966 com Ralph Baer. [9] Trabalhando com design de televisões para a empreiteira *Sanders Associates*, ele teve a ideia de produzir jogos iterativos para televisões domésticas. O projeto de Ralph foi recebido com criticismo pelo conselho executivo, mas seu chefe ficou impressionado e a ideia foi levada adiante. Depois de muitas tentativas de encontrar uma empresa para fabricar a plataforma, a companhia de televisão *Magnavox* finalmente assinou um contrato com a *Sanders Associates*. Em 1972 o sistema *Magnavox Odyssey*, visto na Figura 2.3, foi anunciado e se tornou o primeiro console de jogos eletrônicos da história. Porém, devido ao alto custo do aparelho e falta de marketing, passou despercebido.

Figura 2.3: *Magnavox Odyssey* [12]

Nolan Bushnell foi a peça chave para a popularização dos jogos eletrônicos. [9] Ele era um engenheiro apaixonado por jogos, mas diferente de seus antecessores, era também um empreendedor que conhecia de negócios. Tendo jogado *Spacewar*, de Steve Russel, ele queria recriar o jogo em uma máquina alimentada por moedas, e em 1969 criou um protótipo. Nolan encontrou uma parceira para fabricar seu sistema, a *Nutting Associates*, e a máquina foi lançada em 1971 com o nome de *Computer Space*, mas não vendeu bem. Acreditando ser capaz de fazer um marketing melhor, Nolan criou sua própria companhia, a *Atari*. Uma das máquinas rodando o *Computer Space* pode ser vista na Figura 2.4.

Figura 2.4: *Computer Space* [13]

Al Alcorn foi contratado por Nolan e começou a trabalhar em um jogo baseado em tênis de mesa. [9] O jogo, chamado *Pong*, foi lançado em 1972, mas chamou a atenção da *Magnavox*, que processou a *Atari* por infringir diversas patentes de Ralph Baer. Após concordar em pagar uma licença, a *Atari* estava livre para desenvolver e publicar jogos. *Pong* se tornou amplamente conhecido, dando o pontapé inicial para o desenvolvimento da indústria de jogos eletrônicos. A *Atari* desenvolveu seu próprio console caseiro, baseado em cartuchos, e logo foi seguida por outras empresas. A Figura 2.5 mostra o console desenvolvido por ela.

Figura 2.5: Console desenvolvido pela *Atari* [14]

Com a introdução dos cartuchos, começamos a ver uma transição dos jogos do *hardware* para o *software*. [9] Este foi um passo importante, pois a funcionalidade e lógica de alto custo foram retiradas dos aparelhos e colocadas em pequenas fitas. Com o rápido avanço dos microchips foi possível construir computadores mais versáteis e consoles especializados que permitiam oferecer uma grande variedade de jogos. [4]

Atualmente, os jogos existem dentro de uma incrível gama de gêneros, e são criados para uma vasta quantidade de plataformas. [4] Os gêneros variam de jogos de cartas a jogos multijogador massivos online, e podem rodar em PCs, consoles, celulares ou outros vários aparelhos. Além disso, o crescente potencial que o *hardware* oferece permite a criação de jogos cada vez mais realistas e complexos. Como resultado, a indústria precisa de profissionais mais capazes, o tamanho das equipes cresce, e com isto o custo de desenvolvimento também. Para compensar, as empresas passaram a utilizar *software* reusável na produção de seus *games*.

Neste ponto podemos entender melhor o que define um jogo eletrônico. Na maioria deles um ambiente, virtual ou real, é modelado matematicamente de forma que possa ser manipulado por um computador. [4] Como o *hardware* atual ainda está longe de possuir a capacidade para representar ambientes inteiros ao nível subatômico, estes modelos são uma simplificação da realidade. Em resumo, uma máquina computadora manipula um modelo

simplificado de um ambiente, portanto jogos podem ser considerados *simulações de computador*. Este ambiente é dinâmico e muda com o tempo, conforme a simulação progride, e por isto é chamada de *simulação temporal*.

Em um jogo, diversos agentes interagem entre si. [4] Estes agentes podem ser personagens, veículos como espaçonaves e carros, bolas de fogo, entre outros. Esta característica determina que os jogos são *baseados em agentes*.

O grande diferencial que separa jogos eletrônicos de filmes e livros é sua natureza *interativa*. [4] Os jogos recebem comandos dos jogadores que influenciam na progressão da simulação. Além disso, estes comandos são dados em *tempo-real*, o que significa que não existe demora entre o tempo que são recebidos e a resposta para eles – os resultados devem ser exibidos assim que o comando for recebido, dentro de um curto prazo de tolerância. Porém, se este prazo for ultrapassado, como no caso de um computador não conseguir computar um comando suficientemente rápido, não haverá nenhuma consequência catastrófica na vida real, e portanto este tempo-real é dito *flexível*. Isto difere de um sistema de controle de voo, por exemplo, onde um comando tardio pode resultar em centenas de vidas perdidas.

Juntando todas estas características, um jogo eletrônico pode ser definido como uma *simulação de computador interativa em tempo-real flexível baseada em agentes*. [4]

2.2 Motores de jogos

A transição dos jogos do *hardware* para o *software* foi um grande passo para baixar os custos de desenvolvimento e aumentar a popularidade deste mercado, mas devido às limitações do *hardware*, o código dos jogos ainda era intimamente ligado com a lógica e dados do jogo sendo desenvolvido, então pouca coisa podia ser reutilizada na produção de outro produto. [4] Com o tempo o *hardware* foi ficando mais potente, e a grande mudança veio em 1993 com o lançamento de *Doom*, um jogo de tiro em primeira pessoa criado pela *id Software*, assim como outros jogos similares. A Figura 2.6 mostra a interface do jogo *Doom*.

Figura 2.6: *Doom* [15]

Doom foi desenvolvido com uma separação bem definida entre seus componentes vitais e os dados do jogo, como mapas, imagens e regras de jogabilidade. [4] Esta separação não só permitiu o reuso dos principais sistemas em outros jogos, como também permitiu que fossem licenciados à outras empresas e deu início a uma comunidade de pessoas dedicadas a alterar ou incrementar os jogos existentes, chamados *modders*. Nesta época nasceu o termo motor de jogos, ou *game engine*, que se refere ao conjunto de ferramentas e componentes reusáveis criados especificamente para o desenvolvimento de jogos. A tendência virou um padrão na indústria, e hoje existem diversos motores.

Não existe uma definição clara sobre a diferença entre um jogo e um motor de jogos, pois a linha entre eles geralmente se mistura. [4] Cada tipo de jogo possui seus próprios requerimentos tecnológicos, e quanto mais o motor se distanciar de um gênero específico, mais ele tende a ser reusável. Porém, quanto mais reusável, menos otimizado ele tende a ser para os jogos que produz. No geral, a parcela de dados que o motor consegue manter externos determina o quão genérico ele é. Podemos dizer que esta separação entre lógica e dados é onde está a fronteira entre o motor de jogos e os jogos que ele pode produzir.

2.3 Arquitetura de motores de jogos

Motores de jogos são frameworks distribuídos na forma de um executável e, se possuir, uma suíte de ferramentas. [4] Eles geralmente são sistemas de *software* grandes, e são construídos em camadas, onde as superiores dependem das inferiores, evitando ao máximo dependências cíclicas, que é quando uma camada inferior depende de uma superior. Cada camada implementa um dos componentes principais do motor, ou faz parte de um grupo maior de

camadas que juntas formam um destes componentes. Cada motor possui sua própria arquitetura, mas elas quase sempre convergem em um modelo semelhante. A Figura 2.7 apresenta um exemplo de arquitetura empregada em motores de jogos.

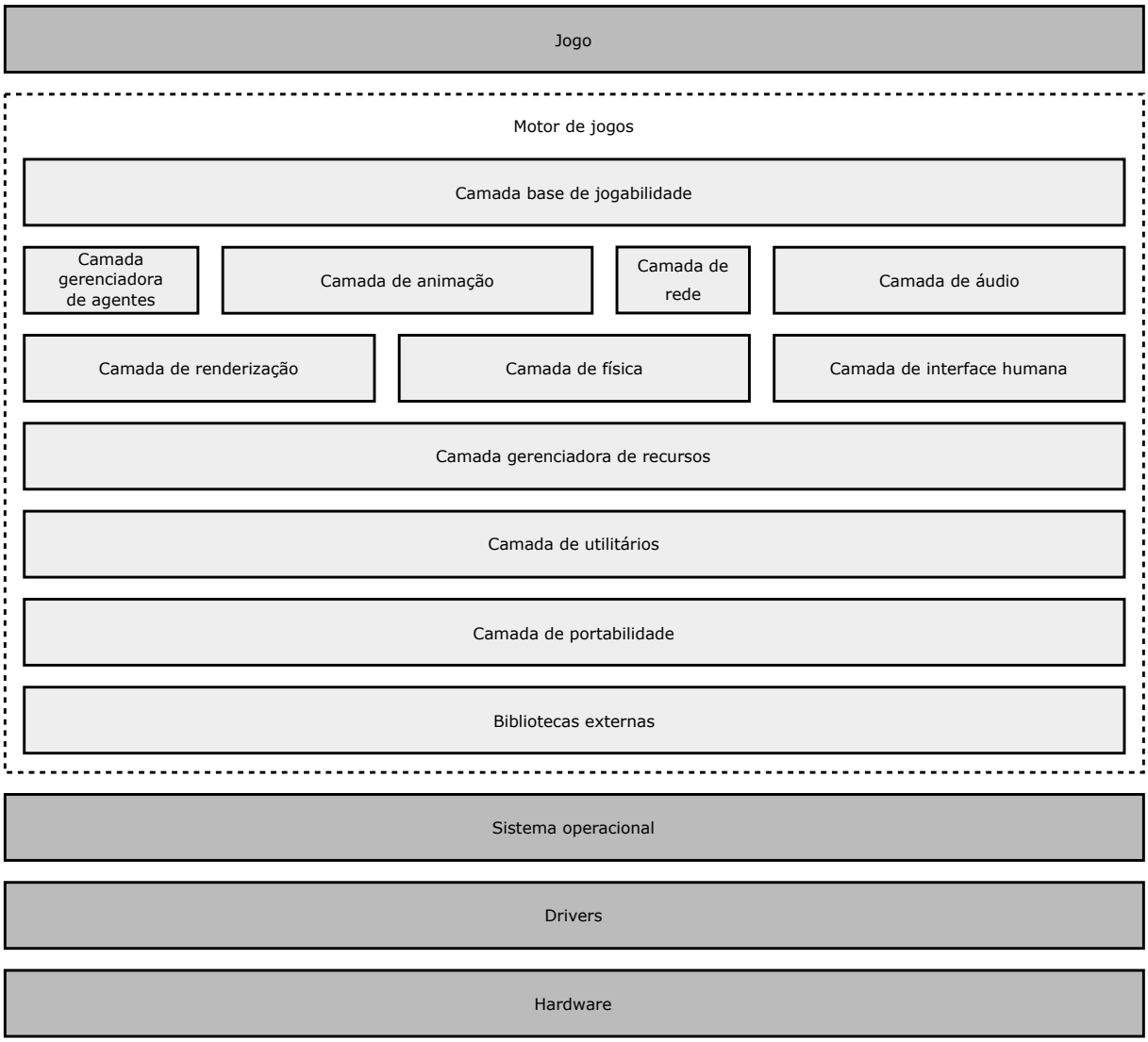

Figura 2.7: Exemplo de arquitetura de um motor de jogos (Do autor)

Todos os motores são construídos sobre uma ou mais plataformas, portanto suas camadas dependem diretamente do sistema-alvo. [4] No nível de plataforma temos o *hardware*, *drivers* e *sistema operacional* para qual o motor está sendo desenvolvido. Nem todas as plataformas utilizam um sistema operacional, como é o caso de consoles antigos.

Por serem grandes, motores de jogos tentam poupar tempo e código ao reutilizar *bibliotecas externas* para diversas finalidades, como estruturas de dados otimizadas e APIs gráficas. [4] Se forem bem escolhidas, estas bibliotecas externas possuem a vantagem de estarem sempre atualizadas e estáveis, devido à equipes dedicadas trabalhando sobre elas. Exemplos de

bibliotecas comumente utilizadas incluem *OpenGL*, *DirectX*, *STL* e *Boost*.

Caso o motor suporte mais de uma plataforma, deve existir uma *camada de portabilidade*, que abstrai as diferenças entre os sistemas e fornece uma interface unificada para as camadas superiores. [4] Subcamadas que podem compor a camada de portabilidade incluem acesso ao sistema de arquivos, métodos auxiliares e estruturas de dados, temporizadores e contadores de tempo, manipulação de *threads* e *networking*.

Diversas camadas dependem de recursos comuns, e por isto uma *camada de utilitários* é implementada para fornecê-los às camadas superiores. [4] Estes recursos incluem conjuntos de operações matemáticas, sistemas de configuração, leitores de tipos de arquivos, *loggers*, métodos de depuração e análise de performance, sistemas de localização, entre outros.

Todos os motores fornecem uma *camada de recursos* de alguma forma. [4] Ela é responsável por dar acesso unificado aos recursos do jogo como imagens, sons, modelos 3D e mapas. Ela pode tomar controle de todo o processo de carregamento e armazenamento dos recursos, ou deixar que o programador faça isto diretamente.

Deve existir algum meio de apresentar ao jogador o estado atual do jogo. [4] A melhor forma de fazer isto, presente em praticamente todos os jogos, é através da renderização do ambiente na tela. A renderização é um dos maiores e mais complexos componentes de um motor de jogos, e é encapsulada na *camada de renderização*. Dentro dela existem diversas outras subcamadas, que dependem da abordagem do motor de jogos. Geralmente estão presentes métodos de exibição de modelos 3D, exibição de primitivas gráficas, texturas, materiais, iluminação, câmera e *shaders*.

A forma mais frequente de interação entre os agentes do jogo é através da colisão. [4] Sem ela, os objetos do ambiente penetrariam uns aos outros. Além disso, alguns motores também fornecem métodos de simulação física semi-realista. A *camada de física* é responsável por fornecer estas funcionalidades.

Como jogos são por definição experiências interativas, mesmo o mais simples dos motores de jogos precisa de uma *camada de interface humana*. [4] Esta camada permite ao jogador enviar comandos para o motor através de um teclado, *mouse*, *joypad*, ou outro tipo de dispositivo de entrada. Os comandos podem então ser traduzidos em ações e influenciar na simulação do ambiente do jogo. Alguns motores também implementam diferentes tipos de resposta ao jogador nesta camada, como vibrações no *joypad*.

Os motores precisam armazenar os agentes que estão atuando no jogo de alguma

forma organizada, como uma lista, e para isto implementa-se a *camada de agentes*. [4] Em jogos de grande porte, se estes agentes não forem armazenados e tratados de modo eficiente, a simulação pode se tornar lenta demais. Existem algumas estruturas de dados que podem otimizar o armazenamento e tratamento destes agentes, como grafos de cena e árvores octais.

Durante a simulação, é comum que um agente tenha seu modelo alterado. Um personagem, por exemplo, deve mover sua perna para frente ao andar. Este tipo de alteração é tratado na *camada de animação*, que é necessária em todos os jogos onde existem agentes orgânicos ou semi-orgânicos, como humanos, animais ou robôs. [4]

Além da renderização gráfica, outro método amplamente usado para representar o estado atual do jogo é através de sons. [4] Para isto, os motores implementam a *camada de áudio*, responsável por reproduzir arquivos de áudio e manipulá-los.

A maioria dos jogos fornece algum tipo de jogabilidade cooperativa ou competitiva entre dois ou mais jogadores humanos, conhecido como *multiplayer*. [4] Antigamente, isto se limitava à divisões de tela, mas com o avanço da *internet* é cada vez mais comum que o *multiplayer* seja feito *online*. Nestes casos é necessário que o motor possua uma *camada de rede* que forneça um meio de estabelecer conexão entre duas ou mais máquinas, permitindo enviar e receber pacotes, entre outras características.

Por fim, o motor deve possuir alguma forma de extensibilidade para que jogos possam ser criados nele. [4] Na *camada base de jogabilidade* o motor pode, por exemplo, utilizar uma linguagem de *script* para que o programador possa definir as regras do jogo e como os agentes interagem entre si. Esta camada também pode realizar o carregamento do ambiente do jogo e disponibilizar um sistema de troca de mensagens entre os sistemas e agentes.

Dependendo dos requisitos do motor de jogos, algumas destas camadas não estarão presentes, ou outras serão necessárias. [4] É comum que motores estado-da-arte possuam muitas outras camadas, incluindo várias apenas para o componente de renderização.

Além do executável, alguns motores de jogos contam com uma suíte de ferramentas construídas para trabalhar com os formatos de arquivo que o motor suporta. [4] Outros motores preferem suportar formatos de arquivos já existentes e deixar que os artistas do jogo utilizem as ferramentas de criação de conteúdo que preferirem.

2.4 Tipos de arquivos

O que diferencia um jogo de outro é seu conteúdo na forma de regras de jogabilidade, história, personagens, músicas, entre outros. [4] O motor de jogos deve receber estes dados de algum lugar para que possa utilizá-los, e os dados podem ser representados de diferentes formas. O desenvolvedor do motor deve selecionar quais destas representações, ou tipos de arquivos, ele suportará e implementar rotinas de carregamento para estes tipos.

Além disso, algumas destas representações em arquivo podem não ser otimizadas para o uso em um jogo, portanto seu conteúdo deve ser pré-processado e otimizado antes de ser armazenado em memória pelo motor. [4]

2.5 Componentes

Esta seção reúne de forma resumida algumas técnicas utilizadas na implementação dos componentes mais importantes de um motor de jogos. Diferentes motores possuem diferentes requisitos, então outros métodos podem ser utilizados.

2.5.1 Inicialização e finalização

Motores de jogos possuem diversos sistemas complexos comunicando-se o tempo todo. [4] Vários destes sistemas precisam ser inicializados antes que possam ser usados. Além disso, alguns devem ser corretamente finalizados após o uso. Para complicar ainda mais, podem existir dependências entre os componentes, portanto a inicialização e finalização precisam ser feitas em uma ordem específica. Por exemplo, o sistema de física de um motor talvez só possa ser inicializado após o gerenciador de agentes. A finalização normalmente ocorre em ordem inversa, então neste caso o sistema de física deve ser finalizado antes. Diversas técnicas podem ser utilizadas para garantir uma sequência correta de inicialização e finalização dos componentes.

2.5.1.1 Método simples

A forma mais simples de oferecer esta ordem é explicitamente definindo funções de inicialização e finalização para cada sistema. [4] Estas funções tomam a responsabilidade dos construtores e destrutores em uma linguagem orientada à objetos, que por sua vez passam a não fazer nada, como mostrado no Código-fonte 2.1.

```
1 classe Componente:
2     construtor Componente():
```

```
3           # não faça nada
4
5       destrutor Componente():
6           # não faça nada
7
8       método inicialize():
9           # código de inicialização do componente
10
11      método finalize():
12          # código de finalização do componente
```

Código-fonte 2.1: Pseudo-código de um sistema de inicialização e finalização simples (Do autor)

As funções de inicialização e finalização podem então ser chamadas em ordem correta em alguma parte do código, como dentro da função principal do programa ou em uma função unificada responsável pela inicialização e finalização de todos os componentes. [4] O Código-fonte 2.2 apresenta um exemplo de uso das funções de inicialização e finalização.

```
1  classe Motor:
2      defina componente_1 como Componente
3      defina componente_2 como Componente
4      defina componente_3 como Componente
5
6      método inicialize():
7          componente_1.inicialize()
8          componente_2.inicialize()
9          componente_3.inicialize()
10
11     método finalize():
12         componente_3.finalize()
13         componente_2.finalize()
14         componente_1.finalize()
15
16 função principal():
17     defina motor como Motor
18     motor.inicialize()
```

Código-fonte 2.2: Pseudo-código de chamada personalizada aos métodos de inicialização e finalização (Do autor)

A desvantagem deste método é que, por erro do programador, os componentes podem ser inicializados ou finalizados em uma ordem incorreta, mas isto pode ser facilmente evitado com atenção, ou corrigido movendo algumas linhas de código. [4]

2.5.1.2 Construção sob demanda

Algumas linguagens de programação permitem o uso de objetos estáticos. [4] Estes objetos são construídos antes da chamada à função principal, e destruídos após o retorno da mesma. Isso permite a criação de objetos globais contendo os componentes, onde os construtores são chamados quando o programa inicia, e os destrutores quando o programa termina. Esta solução é apresentada no Código-fonte 2.3.

```
1  classe Componente:
2      construtor Componente():
3          # código de inicialização do componente
```

```
4
5       destrutor Componente():
6           # código de finalização do componente
7
8   defina componente_1 como Componente estático
9   defina componente_2 como Componente estático
10  defina componente_3 como Componente estático
```

Código-fonte 2.3: Pseudo-código de componentes como objetos estáticos (Do autor)

A desvantagem deste método é que não é possível prever em que ordem os construtores e destrutores de diferentes objetos estáticos serão chamados pelo sistema. [4] Se houver alguma dependência entre dois componentes, o motor poderá terminar de forma crítica ao tentar usar recursos de um componente ainda não inicializado.

Uma forma de contornar este problema é utilizando o padrão de projeto *singleton*. [4] Este padrão garante que uma classe tenha apenas uma instância com um ponto de acesso global. [16] *Singletons* possuem a característica de fornecer uma inicialização tardia.

Este padrão de projeto pode ser facilmente implementado em linguagens de programação que permitem a criação de objetos estáticos dentro de funções, onde os objetos são construídos apenas uma vez, antes da primeira chamada à função que os contém. [4] O Código-fonte 2.4 apresenta uma forma de implementação de *singletons*.

```
1   classe Componente:
2       construtor privado Componente():
3           # código de inicialização do componente
4
5       destrutor privado Componente():
6           # código de finalização do componente
7
8       método estático receba_instância() retorna referência para Componente:
9           # a variável instância é construída apenas uma vez
10          defina instância como Componente estático
11          retorne referência para instância
```

Código-fonte 2.4: Pseudo-código de componentes como *singletons* (Do autor)

Como o objeto só será inicializado quando a função para receber a instância for chamada, o problema de dependências durante a inicialização é resolvido. [4] Além disso, se o componente nunca for utilizado, ele não é inicializado. [16]

A desvantagem deste método é que ainda não há como saber em que ordem o sistema destruirá os objetos, nem em que ponto do programa os construtores serão chamados. [4] A inicialização tardia pode ser vantajosa em alguns casos, mas se acontecer no meio de um ciclo da simulação o jogo pode congelar até que a alocação e construção do objeto terminem.

2.5.1.3 Outros métodos

Existem outras maneiras de inicializar e finalizar os componentes do motor em ordem correta. [4] O programador pode registrar a prioridade dos componentes em uma lista e depois percorrê-la durante a inicialização e finalização, ou pode fazer com que os componentes listam suas dependências para que um algoritmo gere uma ordem otimizada.

2.5.2 Ciclo principal

Por definição, jogos são simulações dinâmicas em tempo-real, e portanto o tempo realiza um papel importante. [4] Ele é usado em várias partes do jogo, como para saber em que fração da simulação o jogo está, o estado atual de uma animação, entre outros inúmeros usos.

Praticamente tudo o que acontece durante a simulação está relacionado com o tempo decorrido. [4] Desta forma, ele deve ser contado precisamente. Qualquer erro pode resultar no jogo rodando rápido demais, lento demais, ou componentes saindo de sincronia com o resto da simulação.

Diversos sistemas do motor de jogos devem ser atualizados constantemente. Como máquinas eletrônicas por natureza não trabalham de modo contínuo, as atualizações devem ser feitas em intervalos de tempo discretos, periodicamente. Porém, a frequência com que elas devem acontecer depende do componente. [4] Enquanto o sistema de física precisa ser atualizado muitas vezes por segundo para funcionar adequadamente, o sistema de inteligência artificial pode ser atualizado apenas uma ou duas vezes por segundo, por exemplo.

A solução é utilizar um ciclo constante que executa uma função de atualização para cada um destes sistemas. [4] Este ciclo é chamado de ciclo principal ou ciclo de jogo. Ele começa a rodar após a inicialização de todos os componentes, e continua rodando até que o motor decida encerrar sua execução, geralmente quando o usuário escolhe fechar o jogo. Após seu encerramento, os componentes são finalizados e o programa termina. A cada iteração do ciclo principal, todos os componentes que precisarem são atualizados. Este processo, exemplificado na Figura 2.8, resume o fluxo de operações de praticamente todos os jogos existentes, mas pode ser implementado de diferentes formas.

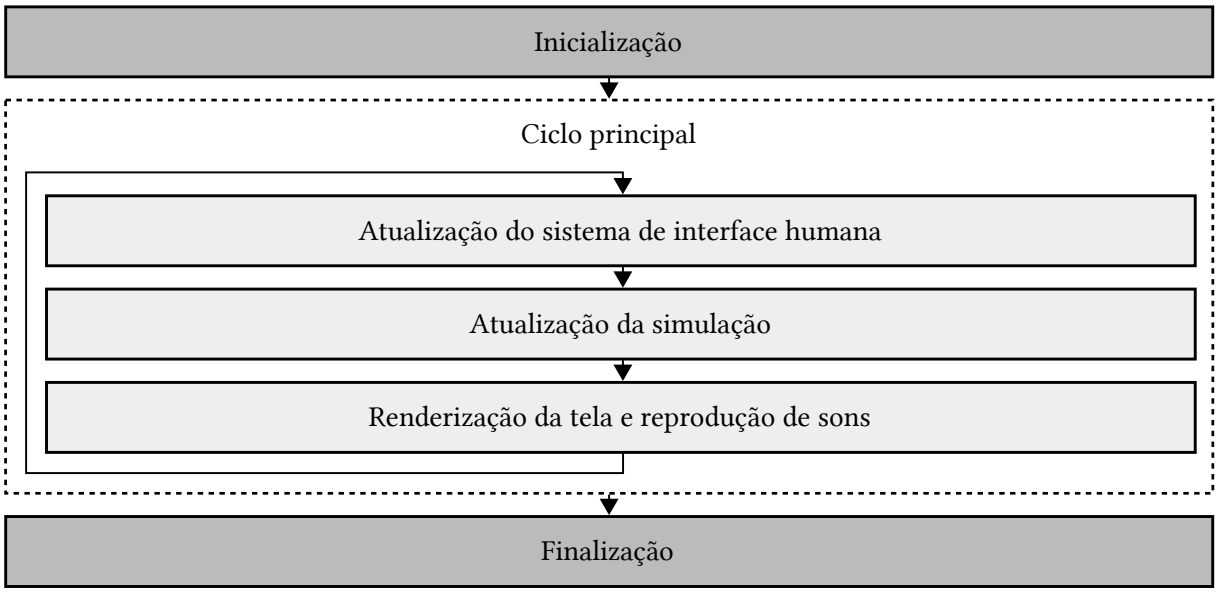

Figura 2.8: Ciclo principal utilizado para atualizar componentes do motor de jogos (Do autor)

2.5.2.1 Método simples

O funcionamento do ciclo principal pode ser implementado de forma bem simples. Uma variável de controle deve ser definida, e um ciclo é inicializado e continua rodando enquanto a variável de controle não indicar uma finalização. [4] A cada iteração, todos os sistemas são sistematicamente atualizados.

O problema desta abordagem é que não há nenhuma forma de calcular o tempo decorrido no jogo. [16] Como resultado, ele não poderá ser usado para computar o próximo estado da simulação. Se um agente precisa ser impulsionado para frente a 5 unidades por segundo, por exemplo, não há como saber quanto deslocá-lo por ciclo. É preciso passar como argumento para as funções de atualização quanto tempo se passou desde a última atualização para que elas possam progredir corretamente, usando este valor para escalar seus cálculos internos. Esta variação de tempo é muitas vezes chamada de *delta-tempo*, ou apenas *delta*.

Na maioria das situações, o *delta* progride junto com o tempo real, mas isto não é estritamente necessário. [4] Alguns jogos fornecem ao jogador a habilidade de congelar a simulação, característica popularmente conhecida como *pause*. Outros permitem que o jogador acelere ou retarde sua progressão. Estas funcionalidades podem ser facilmente obtidas alterando o *delta* usado na linha do tempo.

Além disso, podem existir várias linhas do tempo em uso pelo jogo, cada uma com sua própria velocidade. [4] A simulação do ambiente pode estar em *pause*, mas mantendo a habilidade do jogador de mover a câmera. Neste caso, a simulação da câmera usa uma linha

de tempo separada da simulação do ambiente. As animações de cada agente do jogo tendem a ter suas próprias linhas do tempo. A Figura 2.9 mostra a interface do jogo *StarCraft II*, onde é possível alterar o ritmo do jogo, e algumas unidades possuem habilidades que influenciam em suas velocidades individuais de movimento e animação.

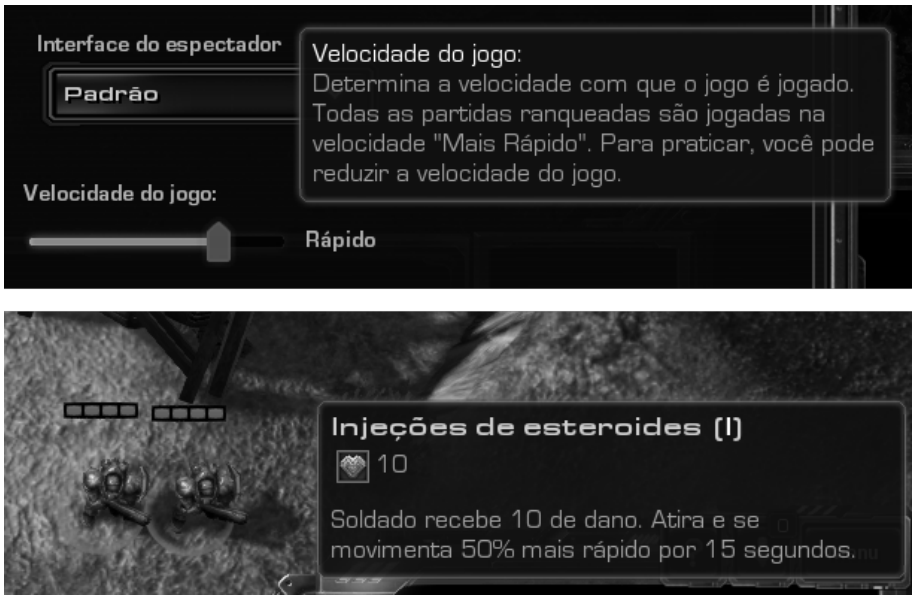

Figura 2.9: O jogo *StarCraft II* utiliza linhas do tempo separadas para o jogo e suas diferentes animações (Do autor)

2.5.2.2 Variação de tempo fixa

A solução mais trivial é passar para as funções de atualização um *delta* fixo. Se soubermos com antecedência que a taxa de renderização da tela é, por exemplo, de 60 quadros por segundo, podemos passar esta fração de tempo em cada ciclo para a simulação. [16]

Em um situação realística, porém, não há como saber a taxa de atualização da tela de antemão, ou a sincronização vertical pode estar desativada. Se o jogo for desenvolvido para PCs, onde o hardware varia muito, o jogo rodará rápido demais em máquinas poderosas, ou lento demais em máquinas antigas. Mesmo em um console com o mesmo hardware, o tempo para simular e renderizar cada ciclo pode variar de acordo com o número de agentes na tela. [16]

2.5.2.3 Variação de tempo flexível

Ao invés de fixar um valor para *delta*, podemos simplesmente deixar que ele flutue de acordo com o tempo gasto por cada quadro. [17] Se o quadro anterior executou rápido demais, gastando apenas 5 milissegundos para ser processado, o próximo quadro deverá simular estes 5 milissegundos perdidos. Por outro lado, se em uma máquina mais lenta um quadro levar 2 segundos para ser processado, basta simular 2 segundos no próximo quadro. Desta forma não

importa quanto tempo os ciclos levem, a simulação estará sempre em sincronia com o tempo real. O Código-fonte 2.5 exemplifica esta contagem e controle de tempo.

```
1  [...]
2  método ciclo_principal():
3      defina delta como decimal
4      defina tempo_anterior como decimal
5      defina tempo_atual como decimal
6
7      tempo_anterior = pegue_tempo_atual()
8      rodando = verdadeiro
9
10     enquanto rodando:
11         tempo_atual = pegue_tempo_atual()
12         delta = tempo_atual - tempo_anterior
13         tempo_anterior = tempo_atual
14
15         simule(delta)
16         renderize()
```

Código-fonte 2.5: Pseudo-código do ciclo principal com variação de tempo flexível (Do autor)

De início, esta solução parece perfeita. [16] Se um agente está se movendo pela tela, seu deslocamento por ciclo pode ser escalado por *delta*. Se os ciclos estiverem demorando demais, o agente se deslocará mais a cada ciclo, mas se estiverem sendo processados muito rápido, o agente se moverá menos. Assim mesmo que a velocidade dos ciclos varie, o agente se moverá em uma velocidade consistente em relação ao tempo real, seja em poucos ciclos lentos ou em muitos ciclos rápidos.

Apesar disto, este método possui alguns problemas, e o mais grave deles ocorre no sistema de física. [17] Imagine que a mesma simulação está rodando em duas máquinas diferentes, uma capaz de rodar o jogo a 50 quadros por segundos, e outra mais lenta, conseguindo apenas 5 quadros por segundo. Nestas duas máquinas há um agente se movendo rapidamente em direção a uma parede. Na máquina rápida, como o sistema de física testa colisões 50 vezes por segundo em pequenos intervalos, rapidamente notará que o agente colidiu com a parede e tomará medidas para impedir que ele a atravesse. No caso da máquina lenta, porém, como as colisões são testadas apenas 5 vezes em grandes intervalos, o agente poderá atravessar a parede sem encostá-la em nenhum ciclo.

Outro problema é que esta alternativa torna o jogo não-determinístico. [16] Praticamente todos os jogos 3D utilizam números de ponto flutuante para representar diversas variáveis usadas pelo sistema de física, como a posição dos agentes. Operações com ponto flutuante são propensas a erros de arredondamento. Como a máquina rápida executa mais destas operações por segundo, estes erros se acumulam mais vezes do que na máquina lenta. Como resultado, o mesmo agente se movendo na mesma simulação pode terminar em posições diferentes.

2.5.2.4 Variação de tempo semi-fixa

Misturando uma variação de tempo fixa com flexível podemos resolver o problema das atualizações com um intervalo muito grande, principalmente no caso do sistema de física. [17] Se determinarmos que a simulação se comporta bem apenas se o *delta* for menor ou igual a um limite, podemos dividir a simulação em diversas iterações que processam, no máximo, o *delta* limite estipulado.

Desta forma conseguimos manter a vantagem de rodar na mesma velocidade em máquinas diferentes, além de não permitir que o delta ultrapasse um limite seguro. [17] O problema é que a simulação deve passar por mais iterações em uma máquina que não consiga executar o ciclo em menos tempo que o limite estipulado. Isto não é nenhum empecilho se o atraso estiver no tempo gasto pela renderização, pois ela é executada apenas uma vez. Porém, se o atraso estiver na simulação, caímos em um problema coloquialmente chamado de espiral da morte.

No caso da simulação introduzir o atraso no ciclo, este atraso acumula-se para a próxima simulação, que terá mais iterações para calcular, causando ainda mais atraso para o próximo ciclo. [17] Se o atraso for um pico momentâneo, a simulação conseguirá recuperar o tempo perdido em algum ponto, mas se for causado por uma máquina lenta, este processo se repetirá exponencialmente.

Além disso, apesar de melhorar o método anterior quanto ao intervalo nas atualizações, esta solução continua não sendo determinística. [17] Isto não é um problema em alguns casos, mas para jogos *multiplayer* é importante que a simulação dos jogadores seja exatamente igual em todas as máquinas. Um jogo determinístico também tem outras vantagens interessantes. O programador pode, pode exemplo, querer implementar um sistema de *replay*, para que todos os comandos do jogador e o tempo quando foram recebidos sejam gravados em um arquivo e possam ser reproduzidos de forma fiel posteriormente.

2.5.2.5 Variação de tempo fixa controlada

O problema do método anterior acontece porque a simulação depende do *delta* utilizado. [16] A renderização, por outro lado, é imune a atrasos nos ciclos. Ela simplesmente captura um estado e representa-o na tela, sem depender de quanto tempo foi gasto entre um ciclo e outro.

Pensando nisso, podemos melhorar a solução anterior ao transferir o atraso da

simulação para a renderização. [17] A simulação pode tomar o tempo que precisar para processar e recuperar o tempo gasto no último ciclo, mesmo em máquinas mais lentas. Quando a simulação alcançar seu objetivo, renderizamos o estado calculado e prosseguimos para a próxima iteração. Em caso de atrasos na simulação, a taxa de renderização será afetada, mas a simulação jamais ficará para trás.

Podemos também aproveitar a ideia do método anterior, mas descartando a flexibilidade do *delta*. [16] Com isso o jogo passa a ser determinístico, pois a simulação sempre avançará em passos fixos, independente da máquina onde está sendo rodado.

Este método é apresentado no Código-fonte 2.6, e pode ser resumido da seguinte forma: o ciclo produz tempo durante seu processamento, e este tempo é consumido pela simulação em intervalos fixos do tamanho de *delta*. [17]

```
1  [...]
2  método ciclo_principal():
3      defina delta como decimal
4      defina delta_acumulado como decimal
5      defina tempo_anterior como decimal
6      defina tempo_atual como decimal
7
8      delta = 1 / 60
9      tempo_anterior = pegue_tempo_atual()
10     rodando = verdadeiro
11
12     enquanto rodando:
13         tempo_atual = pegue_tempo_atual()
14         delta_acumulado += tempo_atual - tempo_anterior
15         tempo_anterior = tempo_atual
16
17         enquanto delta_acumulado >= delta:
18             simule(delta)
19             delta_acumulado -= delta
20
21         renderize()
```

Código-fonte 2.6: Pseudo-código do ciclo principal com variação de tempo fixa controlada (Do autor)

Com esta solução temos uma simulação constante em relação ao tempo real, que atualiza em intervalos de tempo seguros mesmo em diferentes máquinas, para não causar problemas em sistemas como o de física, e que é determinística devido ao valor fixo de *delta*. [16] Todas as preocupações mais críticas foram resolvidas e apenas alguns problemas menores sobraram, mas que podem ser resolvidos de formas simples, se necessário.

Se *delta* for menor que o tempo necessário para realizar um passo da simulação em uma máquina lenta, ela nunca conseguirá recuperar o tempo gasto, trazendo novamente o problema da espiral da morte. [16] Porém, isto pode ser aliviado ao escolher um valor adequado para *delta*, que não seja muito baixo. Dependendo dos requisitos do jogo, o problema pode ser resolvido limitando o número de passos que um ciclo pode simular, ao custo de tornar a

simulação mais lenta, o que nem sempre é aceitável.

Como o tempo produzido é consumido em intervalos fixos, é bem provável também que reste algum meio-passo não simulado para o próximo ciclo. [16] Este atraso não traz problemas para a simulação, pois será adequadamente simulado no próximo ciclo, mas como o estado utilizado pela renderização não necessariamente representa o estado atual naquele momento, os quadros exibidos na tela não terão um sequenciamento completamente suave, e isto pode ser notado por jogadores mais atentos. Este problema, mostrado na Figura 2.10, é conhecido como *aliasing* temporal, e acontece principalmente quando a simulação usa uma taxa de quadros por segundo que não é múltipla da taxa de renderização. [17]

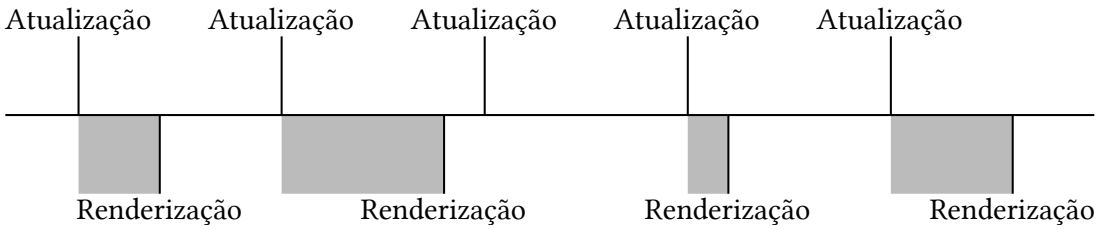

Figura 2.10: Demonstração do problema de *aliasing* temporal: os blocos em cinza não estão simulados no momento da renderização (Do autor)

2.5.2.6 Variação de tempo fixa controlada com interpolação

O problema de *aliasing* temporal acontece quando o tempo gasto para simular e renderizar um ciclo não é completamente consumido pela simulação. [16] Como este intervalo é menor que *delta*, não podemos inserí-lo na simulação sem quebrar o determinismo do jogo, mas podemos utilizá-lo para interpolar um estado temporário que representa o momento imediatamente antes da renderização. Este procedimento envolve criar uma cópia de todo o estado do jogo para que os dados da simulação não sejam perdidos, e então executar mais um passo da simulação utilizando o *delta* restante acumulado.

Como o estado interpolado não é utilizado pela simulação, é possível manter tanto a atualização dos sistemas quanto a renderização em uma taxa uniforme e sem travamentos. [16]

2.5.2.7 Outros métodos

Existem outras formas de implementar o ciclo principal. [4] Na plataforma *Windows*, os jogos precisam responder à mensagens do sistema, e por isso seu ciclo deve possuir alguma forma de receber estas mensagens, respondê-las, e se necessário tomar alguma medida interna como resultado da mensagem. Alguns motores baseiam todo o ciclo neste sistema de mensagens. Outros motores utilizar uma abordagem sobre *callbacks* para dirigir o fluxo da simulação, e

outros podem basear suas ações em um sistema de eventos. Alguns motores podem até mesmo misturar diferentes abordagens para atender seus requisitos.

2.5.3 Sistema de arquivos

Todos os motores de jogos precisam carregar dados gravados em arquivos para que possam ser utilizados, como modelos e texturas. [4] As plataformas fornecem acesso ao sistema de arquivos através de bibliotecas de chamada de sistema, mas a maioria dos motores de jogos implementa uma camada acima delas.

O principal motivo é facilitar a portabilidade do motor. [4] Esta camada fornece uma interface única para que o programador possa acessar o sistema de arquivos sem se preocupar com as diferenças entre plataformas. Além disso, as bibliotecas do sistema podem não oferecer recursos mais avançados, necessários por alguns motores de jogos, como *streaming*.

As funcionalidades geralmente presentes na camada de acesso ao sistema de arquivos incluem métodos para abertura, leitura, gravação e fechamento de arquivos, síncronas ou assíncronas, listagem de arquivos em um diretório e manipulação de nomes e caminhos de arquivos e diretórios. [4]

2.5.3.1 Manipulação síncrona de arquivos

A manipulação síncrona geralmente trata apenas de fornecer uma interface unificada sobre funções já presentes nas plataformas suportadas pelo motor. [4] Outras bibliotecas de nível mais alto também podem ser utilizadas, como a biblioteca de entrada e saída padrão da linguagem C.

O motor deve oferecer ao programador a opção de leitura e escrita utilizando *buffers*. [4] *Buffers* servem como um armazenamento temporário em memória para os dados sendo lidos ou gravados. Em alguns casos, porém, o programador pode precisar gerenciar seus próprios *buffers*.

2.5.3.2 Manipulação assíncrona de arquivos

A manipulação assíncrona de arquivos, também conhecida como *streaming*, realiza o mesmo papel da manipulação síncrona, com a diferença de ser executada em uma *thread* diferente. [4] Esta característica permite que a simulação continue rodando enquanto os dados são lidos, e é útil para a implementação de jogos sem telas de carregamento.

É importante que exista alguma forma de detectar quando os dados requisitados

foram completamente lidos, para que outros sistemas saibam que é seguro utilizá-los. [4] Alguns motores podem fornecer também estimativas de tempo, prazos e prioridades sobre o carregamento.

2.5.3.3 Manipulação de caminhos

Diferentes sistemas trabalham com formatos diferentes para representar caminhos para arquivos e diretórios. [4] Sistemas *UNIX*, por exemplo, utilizam uma barra (/) para separar nomes de diretórios, enquanto sistemas *Windows* utilizam uma barra invertida (\). A camada de sistema de arquivos deve fornecer um formato único ao programador, que possa ser traduzido para o formato utilizado pela plataforma na qual o jogo está rodando, além de funções para manipulação de caminhos, como isolamento do nome de arquivo ou extensão, e listagem de arquivos e diretórios em um caminho.

2.5.4 Sistemas de configuração

Por conta de sua complexidade, motores de jogos acabam tendo diversas opções que podem ser configuradas pelo jogador, através de um menu no jogo, ou pelos próprios desenvolvedores. [4] Apesar de ser possível implementá-las de forma trivial, como em variáveis globais, estas opções só são realmente úteis se puderem ser facilmente alteradas e armazenadas.

Existem diversos métodos de armazenamento, leitura e gravação de variáveis de configuração de modo que possam ser posteriormente modificadas sem a necessidade de recompilar o código do motor. [4] Alguns destes métodos são mais adequados que outros em situações específicas. O motor também pode implementar perfis de usuário, possibilitando que cada jogador possua um conjunto de configurações separadas.

2.5.4.1 Arquivos de texto

Por serem legíveis para humanos, arquivos de texto são o método mais comum de armazenamento de configurações em motores de jogos. [4] Existem diversos formatos que podem ser usados para esta finalidade, como JSON, XML, INI e YAML, por exemplo. [18]

2.5.4.2 Arquivos binários

Em plataformas com espaço limitado em disco, como consoles antigos, que geralmente utilizavam pequenos *memory cards*, os arquivos de configuração precisavam ocupar o

menor espaço possível. [4] Nestes casos a melhor solução é utilizar formatos binários, preferencialmente compressos.

2.5.4.3 Outros métodos

Existem outras formas de armazenar configurações, como registros do sistema, parâmetros da linha de comandos, variáveis de ambiente e perfis online. [4] Cada método possui vantagens e desvantagens, que devem ser ponderadas pelo desenvolvedor do motor. Diversos motores fornecem suporte a vários métodos diferentes.

2.5.5 Gerenciador de recursos

Com a grande quantidade e variedade de recursos usados em um jogo, é preciso que eles sejam eficientemente gerenciados. [4] Para isto, todos os motores de jogos possuem algum tipo de sistema gerenciador de recursos.

Em alguns motores, o gerenciador de recursos não é um único sistema, mas um conjunto de diferentes sistemas. [4] O gerenciador de recursos é responsável por tratar o carregamento, armazenamento e descarregamento de todos os recursos do jogo. Ele deve preparar e carregar os recursos e suas dependências para a memória, param que possam ser corretamente utilizados pelo jogo. Além disso, deve garantir que exista apenas uma cópia de cada recurso na memória de uma vez, e que quaisquer referências à este recurso apontem para a única cópia carregada. Também devem gerenciar o tempo de vida de cada recurso, para que sejam descarregados se não forem mais necessários.

2.5.5.1 Formatos de arquivos

O gerenciador de recursos deve ser capaz de carregar e interpretar diferentes tipos de recursos, como arquivos de áudio e texturas. [4] Assim, o motor deve fornecer suporte ao carregamento de vários tipos de arquivos. Texturas, por exemplo, geralmente são armazenadas como PNG ou BMP, enquanto modelos tridimensionais podem ser armazenados como OBJ ou COLLADA. Caso nenhum formato existente satisfaça os requisitos do motor de jogos, pode-se implementar um formato de arquivo personalizado.

Alguns formatos são flexíveis e armazenam informações sobre mais de um tipo de recurso. [4] Além disso, o motor pode fornecer suporte a pacotes de arquivos compactados, utilizando formatos como ZIP.

2.5.5.2 Identificadores únicos de recursos

Cada recurso carregado pelo gerenciador deve possuir um identificador único para que possa ser acessado por outros sistemas. [4] Na maioria das vezes, este identificador é o caminho para o arquivo em disco, mas alguns motores podem usar um *hash* deste caminho, ou um identificador único gerado pelo motor ou por uma ferramenta.

2.5.5.3 Armazenamento dos recursos

O gerenciador de recursos deve garantir que apenas uma cópia do recurso esteja carregada em memória. [4] Isto pode ser facilmente implementado com um dicionário, onde a chave é o identificador único do recurso, e o valor é um ponteiro para os dados do recurso em memória.

Se algum sistema tentar carregar um recurso, o gerenciador deve verificar se ele já existe no dicionário. [4] Se sim, o gerenciador simplesmente retorna o ponteiro para o recurso. Caso contrário, o recurso é carregado, alocado, e seu ponteiro é adicionado no dicionário junto com seu identificador único. Quando o recurso é descarregado, sua entrada é removida do dicionário.

Como o carregamento de recursos envolve operações com o disco, que são naturalmente lentas, é recomendado que todos os recursos necessários sejam carregados ao mesmo tempo em uma tela de *loading*, ou que este carregamento seja feito de forma assíncrona. [4]

2.5.5.4 Tempo de vida dos recursos

Alguns recursos são temporários, sendo usados apenas em algumas partes do jogo, enquanto outros são usados por muito tempo. [4] Quando um recurso não for mais necessário, deve ser descarregado pelo gerenciador.

Um método simples para saber quando um recurso pode ser desalocado é por contagem de referências. [4] Todos os recursos devem possuir um contador, e cada vez que o recurso é requisitado por um sistema, seu contador é incrementado. Quando o sistema não precisar mais do recurso, deve desreferenciá-lo, para que seu contador seja decrementado. Se a contagem de referências de um recurso for zero, sabemos que não está mais sendo usado e pode ser descarregado pelo gerenciador.

2.5.6 Gerenciador de agentes

Diversos sistemas de um jogo precisam manipular periodicamente os estados internos dos agentes sendo simulados. [4] Esta manipulação pode se dar de várias formas, como por exemplo, uma alteração na posição ou rotação de um objeto. O motor de jogos precisa, portanto, armazenar e gerenciar todos os agentes da simulação para que possam ser atualizados de modo eficiente.

Estas atualizações periódicas são executadas como parte da simulação, no ciclo principal, e podem ser vistas como o processo de determinar o estado atual de um agente a partir de seu estado anterior e uma variação de tempo *delta*. [4]

2.5.6.1 Método simples

Se os agentes possuírem uma função para realizar sua atualização, é possível iterar toda a lista de agentes uma vez por ciclo, chamando esta função para cada um deles. [4] O motor pode passar como parâmetro a variação de tempo desde o último ciclo, para que os agentes atualizem seus estados de forma consistente em relação ao tempo decorrido. Além disso, se a função for virtual, diferentes classes de agentes podem implementar funções de atualização personalizadas ao herdar de uma classe principal.

Geralmente, a coleção de objetos de um jogo é mantida dentro de um gerenciador de agentes de acesso global e unificado, como um *singleton*. [4] Como objetos são criados e destruídos frequentemente em um jogo, esta coleção deve ser dinâmica, e portanto a forma mais trivial de armazená-los é com uma lista-ligada de ponteiros ou referências para os agentes.

A função de atualização é responsável por manipular informações usadas pelos diferentes sistemas necessários pelo agente em questão. [4] Alguns podem precisar, por exemplo, atualizar suas posições no ambiente do jogo, enquanto outros precisam atualizar o estado atual de sua animação. O método simples é apresentado no Código-fonte 2.7.

```
1  classe Agente:
2      defina posição como real
3
4      método atualize(delta como real):
5          posição += 5.0 * delta
6
7  classe Motor:
8      [...]
9      defina lista_de_agentes como lista-ligada de ponteiro para Agente
10
11     método ciclo_principal():
12         enquanto rodando:
13             [...]
14             para cada agente em lista_de_agentes faça:
15                 agente.atualize(delta)
```

Código-fonte 2.7: Pseudo-código de exemplo de método simples para atualização de agentes (Do autor)

2.5.6.2 Atualização em lote

Apesar de simples, o método anterior possui alguns problemas que podem afetar jogos mais robustos. [4] Como os agentes são atualizados um a um, vários sistemas do jogo são utilizados ao mesmo tempo. Isto pode ocasionar gasto maior de memória e custo adicional de performance.

Um modo mais eficiente de executar a atualização dos agentes, como exibido no Código-fonte 2.8, é fazer cada sistema atualizar todos os agentes separadamente, em lotes. [4] Ao invés de atualizar seu próprio estado, o agente requisita a modificação, e o sistema armazena as informações necessárias para que esta operação seja executada posteriormente. Um agente pode, por exemplo, desejar que uma malha triangular seja renderizada, e transmite os dados da malha para o sistema de renderização, que mantém uma coleção de malhas a serem renderizadas de forma a maximizar seu próprio desempenho.

```
1  tipo RequisiçãoMovimento:
2      defina agente como ponteiro para Agente
3      defina deslocamento como real
4
5  classe SistemaDeFísica:
6      defina requisições como lista-ligada de RequisiçãoMovimento
7
8      método requisite_movimento(agente como ponteiro para Agente,
9      deslocamento como real)
10         defina requisição como RequisiçãoMovimento
11         requisição.agente = agente
12         requisição.deslocamento = deslocamento
13         requisições.insira(requisição)
14
15     método atualize(delta como real):
16         para cada requisição em requisições:
17             requisição.agente.posição += requisição.deslocamento
18
19         requisições.remova_todos()
20
21 defina física como SistemaDeFísica
22
23 classe Agente:
24     defina posição como real
25
26     método atualize(delta como real):
27         física.requisite_movimento(este_objeto, 5.0 * delta)
28
29 classe Motor:
30     [...]
31     defina lista_de_agentes como lista-ligada de ponteiro para Agente
32
33     método ciclo_principal():
34         enquanto rodando:
35             [...]
36             para cada agente em lista_de_agentes faça:
37                 agente.atualize(delta)
38
39             física.atualize(delta)
```

Código-fonte 2.8: Pseudo-código de atualização de agentes em lote (Do autor)

Esta solução possui problemas no caso de interdependência de agentes ou sistemas, mas que podem ser resolvidos ao separar os objetos em diferentes partições para serem atualizadas, e os sistemas em diferentes fases de atualização intercaladas. [4] O método também pode ser melhorado para atender a outros requisitos do motos de jogos, como paralelismo.

2.5.6.3 Grafos de cena

Armazenar os agentes em uma lista-ligada talvez não seja suficiente para a maioria dos motores de jogos modernos, pois alguns objetos podem precisar herdar modificações de outros objetos. [19] Em um modelo de sistema estrelar, por exemplo, as luas de um planeta devem sofrer a mesma rotação que seu planeta, e este por sua vez deve sofrer a mesma rotação que sua estrela, então cada agente precisaria de uma função de atualização complexa apenas para se manter na posição correta em relação aos outros objetos.

A estrutura de dados conhecida como grafo de cena traz características interessantes para otimizar este tipo de situação. [19] Um grafo de cena é uma árvore onde cada nó possui um número qualquer de nós-filhos. Todos eles possuem nós para representar agentes ou geometrias, e nós de transformações como rotação, translação e escala. É possível também implementar diversos outros tipos de nós estruturais para atender a outros requisitos do motor de jogos. A Figura 2.11 é um exemplo de como as partes móveis de um tanque de guerra poderiam ser representadas usando um grafo de cena.

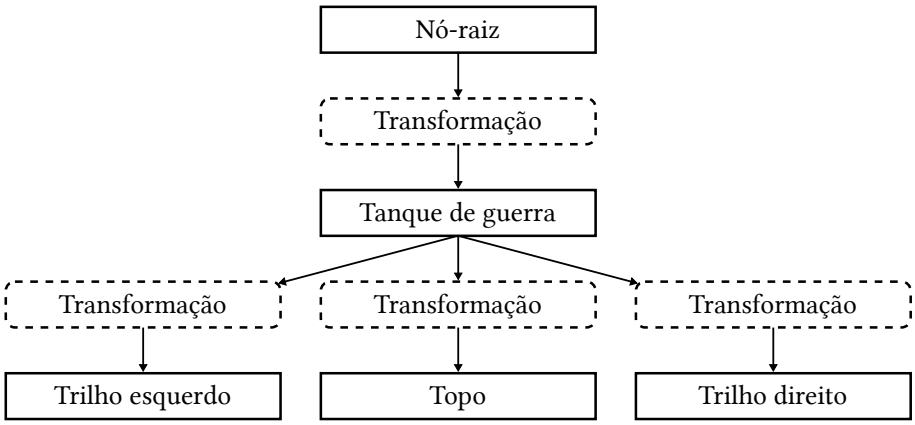

Figura 2.11: Exemplo de grafo de cena representando um tanque de guerra (Do autor)

A maior vantagem dos grafos de cena é a facilidade com que as transformações podem ser recursivamente propagadas para os nós-filhos. [19] Além disso, a estrutura de

árvore permite que partes do ambiente sejam descartadas se não forem necessárias, otimizando operações como detecção de colisão e *frustum culling*. [4]

Grafos de cena devem possuir uma classe base para os nós, de onde todos os diferentes tipos de nós podem herdar. [19] Além disso, eles precisam de um nó-raiz para que os outros nós possam ser acessados. [20] Um exemplo de implementação é apresentado no Código-fonte 2.9.

```
1  classe Nó:
2      defina filhos como lista-ligada de ponteiro para Nó
3
4      método insira_filho(filho como ponteiro para Nó):
5          filhos.insira(filho)
6
7      método remova_filho(filho como ponteiro para Nó):
8          filhos.remova(filho)
9
10     método percorra():
11         para cada filho em filhos:
12             filho.percorra()
13
14 classe GrafoDeCena:
15     defina raiz como ponteiro para Nó
```

Código-fonte 2.9: Pseudo-código de exemplo de um grafo de cena simples (Do autor)

2.5.7 Interface humana

Por serem experiências interativas, jogos precisam receber comandos dos jogadores, e diversos tipos de periféricos podem ser usados para esta finalidade. [4] Alguns foram criados especificamente para serem usados em jogos eletrônicos.

É responsabilidade do motor de jogos fornecer suporte a um ou mais periféricos que possam ser utilizados para guiar a simulação do jogo. [4] Cada tipo de periférico possui suas características, mas no geral, um sistema de interface humana deve realizar a leitura e processamento dos dados dos periféricos suportados, além de oferecer ao programador formas de identificar os comandos do jogador. Alguns periféricos também oferecem algum tipo de resposta, como vibrações em um *joypad*.

2.5.7.1 Tipos de periféricos

Existem diversos tipos de periféricos que podem ser utilizados por jogos, e várias plataformas já vem equipadas com um controlador próprio, como é o caso dos controles para *Xbox One* e *PlayStation 4*, mostrados na Figura 2.12. [4]

Figura 2.12: Diferentes periféricos do tipo *joypad*, para as plataformas *Xbox One* e *PlayStation 4* [21]

Para computadores pessoais, os periféricos mais comuns são a combinação de teclado e *mouse*, mas também existem *joypads* que oferecem suporte a PCs, como o *joypad* do *Xbox 360*. [4]

Os consoles possuem *joypads* próprios, padrão da plataforma, mas também podem oferecer controladores mais especializados que representam, por exemplo, guitarras e volantes. [4]

Máquinas de *arcade* geralmente possuem controles especializados para o jogo executado, que são construídos na própria máquina. [4]

2.5.7.2 Tipos de comandos

Os sinais enviados por periféricos para representar comandos do jogador podem ser divididos em categorias bem definidas. [4]

Praticamente todos os tipos de periféricos possuem botões digitais. [4] Estes botões podem estar apenas em dois estados: pressionado ou não pressionado.

Além dos digitais, alguns botões podem ser analógicos, como botões em forma de gatilho, por exemplo. [4] Neste caso, o sinal depende da força aplicada pelo jogador sobre o botão. Apesar de ser analógico, o sinal é geralmente digitalizado antes de ser enviado para a plataforma.

Sinais analógicos também podem representar eixos, e a união de dois eixos pode ser utilizada para representar a posição bidimensional de um *joystick*. [4]

Eixos também podem ser relativos, o que significa que não possuem um valor inicial. [4] Ao invés de transmitir uma posição absoluta, estes eixos enviam para a plataforma a variação entre seu último estado e o estado atual. Eixos relativos são usados em *mouses* e *trackpads*.

Alguns periféricos apresentam tipos de interação com o jogador mais avançados, como acelerômetros para posicionamento tridimensional do controlador e câmeras. [4]

2.5.7.3 Tipos de respostas

A maioria dos periféricos são usados para receber comandos do jogador, mas alguns também fornecem suporte a respostas de retorno. [4]

Em *joypads*, a forma mais comum de reposta é através de uma vibração no periférico, simulando diversos efeitos, como colisões do ator principal com outro objeto ou turbulência. [4] Esta vibração geralmente pode ser ligada, desligada, ou ter sua intensidade controlada pelo jogo.

Alguns periféricos também oferecem uma resistência forçada às ações do jogador. [4] Em periféricos que simulam volantes, por exemplo, o aparelho pode possuir um motor que resista ao esterçamento para simular dificuldade em uma curva.

Outros controladores podem oferecer algum tipo de reprodução de áudio, como na forma de um auto-falante de baixa qualidade. [4] Alguns *joypads*, como o de *Xbox 360*, possuem entradas para que o jogador possa conectar um *headset*.

2.5.7.4 Leitura dos dados

Diferentes periféricos trabalham de formas diferentes, e o motor de jogos deve implementar métodos apropriados para a leitura de dados, de acordo com as especificações técnicas do aparelho. [4]

Periféricos mais simples, como *joypads* antigos, geralmente requerem que o motor de jogos obtenha o estado dos botões diretamente dos registradores no hardware, ou através de uma porta de entrada e saída. [4] O sistema de interface humana então lê o estado dos botões periodicamente, durante sua atualização.

Alguns periféricos só transmitem informações quando o estado de um de seus botões é alterado, interrompendo a execução do processador temporariamente para que o estado seja registrado em memória e possa ser utilizado posteriormente. [4] É o caso de periféricos como o *mouse*, que passa a maior parte do tempo parado, sem necessidade de transmitir dados se não foi movido e nenhum botão foi pressionado.

Para dispositivos sem fio, que não possuem uma porta física, é necessário atender ao protocolo do aparelho para que uma comunicação seja estabelecida e o motor possa requisitar dados. [4] Normalmente uma interface adicional é implementada para abstrair este processo.

2.5.7.5 Processamento dos dados

A maioria dos motores de jogos implementa uma camada para processar os dados dos periféricos e armazenar informações sobre os comandos do jogador. [4]

Um tipo de processamento feito é a remoção de ruído dos botões e eixos analógicos, que podem possuir valores variando em torno de seu estado quando o botão está parado. [4] Para remover este tipo de ruído o programador deve definir uma margem de erro, chamada de *dead zone*, e qualquer valor do botão quando dentro desta margem deve ser redefinido como sendo o estado parado do botão, como mostrado no Código-fonte 2.10.

```
1  função processe_sinal(valor como real) retorna real:
2      defina erro como real
3      defina parado como real
4      erro = 5
5      parado = 0
6
7      se (valor >= parado - erro) e (valor <= parado + erro) então:
8          valor = parado
9
10     retorne valor
```

Código-fonte 2.10: Pseudo-código de remoção de ruído em torno do estado parado de um botão analógico (Do autor)

Para amenizar ruídos quando o botão está em uso é preciso filtrar o sinal do botão. [4] Isto pode ser feito ao combinar o valor atual do botão com o valor filtrado do ciclo anterior, assim como apresenta o Código-fonte 2.11.

```
1  função processe_sinal(anterior como real, valor como real) retorna real:
2      defina filtro como real
3      filtro = 0.9
4
5      retorne ((1.0 - filtro) * anterior) + (filtro * valor)
```

Código-fonte 2.11: Pseudo-código de amenização de ruído de um botão analógico em movimento (Do autor)

Ao invés de requisitar o estado atual de um botão, jogos as vezes precisam detectar mudanças de estado, como quando um botão foi pressionado ou solto. [4] O motor de jogos pode fornecer esta informação ao comparar o estado do botão no ciclo anterior com o estado no ciclo atual. Se o estado for diferente, houve uma mudança, e o motor pode armazenar esta informação para que possa ser utilizada pelo jogo, geralmente na forma de um evento.

Alguns motores implementam diversos outros tipos de processamento, como detecção de acordes, sequências, gestos ou pressionamento rápido de um botão. [4] Além disso, o motor pode oferecer suporte a vários periféricos sendo usados ao mesmo tempo, para a criação de jogos multijogador locais, ou suportar vários tipos diferentes de periféricos, implementando

uma camada de abstração sobre eles. Outro tipo comum de processamento é o remapeamento de botões, permitindo ao jogador configurar quais comandos são atribuídos a cada botão, ou como eles são tratados, de acordo com suas preferências.

2.5.8 Detecção e resolução de colisões

O papel de não permitir que objetos sólidos atravessem uns aos outros pertence ao sistema de detecção de colisões, que geralmente é integrado no sistema de física do motor de jogos. [4] Desenvolver um sistema de física e colisão é difícil e consome tempo, por isto diversas SDKs de alta-qualidade foram desenvolvidas. Nem todos os jogos precisam de um modelo físico avançado, e a adição de um sistema de física tem grande impacto em diversas áreas do jogo. Se for mal implementado, pode trazer mais desvantagens que vantagens. Ainda assim, todos os jogos que trabalham com objetos sólidos em movimento precisam de alguma forma de detectar quando dois objetos se intersectam para que possam simular uma interação entre eles.

É possível utilizar os dados das malhas tridimensionais para testar colisões, mas esta operação é extremamente custosa devido ao alto número de polígonos envolvidos, sendo inviável para uma simulação em tempo real. [22] Este custo pode ser minimizado ao utilizar uma representação mais simples que encapsule o volume do objeto, geralmente chamada de volume delimitador. Este volume deve oferecer métodos de checagem de interseção extremamente rápidos para não sobrecarregar a simulação. Muitas vezes a interseção entre dois volumes delimitadores é prova suficiente de que houve uma colisão, mas como mostrado na Figura 2.13, eles nem sempre representam fielmente o volume real do objeto, resultando em falsos-positivos. Nestes casos, o sistema de detecção pode ter certeza de uma colisão ao realizar a checagem mais custosa entre as malhas dos objetos após detectar uma interseção entre os volumes delimitadores.

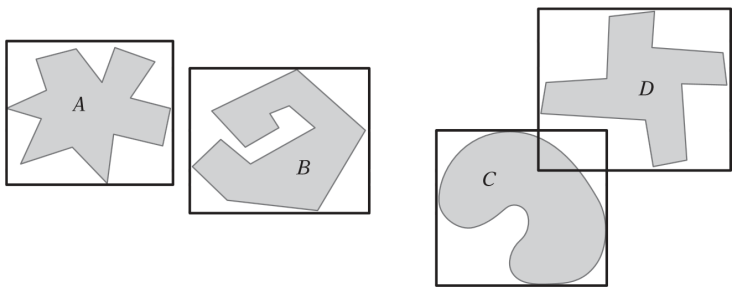

Figura 2.13: Os volumes delimitadores de A e B não possuem interseções, portanto os objetos não estão colidindo, mas não é possível descartar uma colisão entre C e D sem testar seus volumes reais [22]

Diversos tipos de formas servem para representar volumes delimitadores. [22] Estas formas precisam ter armazenamento leve, serem eficientes e fáceis de manipular, além de aproxi-

mar da melhor forma possível o volume real do objeto. Elas geralmente são pré-computadas e armazenadas junto com o modelo tridimensional, seja quando o motor carrega a malha do objeto, ou por uma ferramenta externa que armazena os dados do volume delimitador em um arquivo.

2.5.8.1 Esferas

A forma geométrica mais simples e eficiente para representar um volume delimitador é a esfera. [4] Ela pode ser representada como um ponto central e um raio, e a detecção de colisão pode ser feita comparando a distância de um ponto ao centro do volume com seu raio. Para determinar a colisão entre duas esferas, basta comparar a distância entre os pontos centrais com a soma dos raios. Outra vantagem de esferas é que não são alteradas por rotações. [22]

2.5.8.2 Caixas delimitadores alinhadas a eixos

Caixas delimitadoras alinhadas a eixos, ou AABBs, são caixas retangulares que possuem todas as normais das faces paralelas a algum eixo no sistema de coordenadas. [22] A verificação de interseção entre duas AABBs é uma operação extremamente rápida, e por isto é uma das formas mais usadas para representar volumes delimitadores.

AABBs podem ser representadas como dois pontos contendo os valores mínimos e máximos da AABB, ou por um ponto de origem e vetores representando as três dimensões da AABB: largura, altura e comprimento. [22] O ponto de origem também pode ser substituído por um ponto central, e neste caso os vetores passam a representar apenas metade das dimensões, também chamadas de meia-dimensões.

Duas AABBs só se intersectam se possuírem interseções em todos os eixos. Como as AABBs devem estar sempre alinhadas com o eixo do sistema de coordenadas, elas não podem ser rotacionadas junto com o objeto se este sofrer uma rotação, e portanto devem ser manipuladas ou recalculadas para encapsular corretamente o volume rotacionado do objeto. [22]

2.5.8.3 Caixas delimitadores orientadas

Caixas delimitadoras orientadas, ou OBBs, são parecidas com AABBs, mas não são alinhadas aos eixos do sistema de coordenadas, podendo assumir uma orientação qualquer. [22]

Elas podem ser representadas com oito vértices, com seis planos, com três *slabs* (par de planos paralelos), com um ponto de origem e três vetores de dimensões, ou com um ponto central, três vetores de meia-dimensões e uma matriz de orientação. [22]

Os testes de interseção entre duas OBBs são complicados, e fazem uso do teorema

do eixo de separação. [22] Duas OBBs só estão separadas se a soma da projeção de suas arestas em algum eixo for menor que a distância entre a projeção de seus centros.

Existem 15 eixos que devem ser testados para garantir que duas OBBs estão colidindo: os três eixos da primeira OBB, os três eixos da segunda OBB, e os produtos cruzados das 9 combinações dos três eixos de cada OBB. [22] Se para qualquer um destes eixos existir uma separação nas projeções, as OBBs não estão colidindo.

2.5.8.4 Outras formas

Existem muitas outras formas de representar volumes delimitantes, como pílulas, polítopos de orientação discreta, fechos convexos, cones, cilindros, entre outros. [22]

2.5.8.5 Resolução de colisões

As operações de detecção de colisão não só identificam se uma colisão aconteceu ou não, mas também retornam outras informações úteis sobre a colisão que ocorreu. [4] Estas informações podem ser usadas pelo motor de jogos para decidir como responder à colisão. Uma das informações geralmente retornadas é um vetor de separação, que indica como separar os objetos envolvidos. Utilizando este vetor, é possível transladar os objetos para que saiam do estado de colisão.

2.5.9 Sistema de renderização

Renderização tridimensional é um tópico extremamente amplo. [4] Ao ouvir falar de jogos eletrônicos, a primeira coisa que as pessoas imaginam são gráficos fotorrealistas. Este é provavelmente o componente mais bem coberto entre as tecnologias empregadas em motores de jogos, com inúmeros livros e materiais escritos sobre o assunto. Além disso, é também um dos componentes mais complexos que compõem um motor, podendo ser divido em vários subsistemas.

Uma SDK gráfica como DirectX ou OpenGL pode ser utilizada para realizar a transferência de dados do motor de jogos para o hardware gráfico, mas ainda assim, muitas camadas devem ser implementadas sobre estas SDKs para que o ambiente do jogo possa ser renderizado de forma satisfatória e eficiente. [4]

Geralmente o sistema de renderização possui menos de 34 milissegundos para renderizar um quadro a tempo de enviar a imagem para a tela em uma taxa aceitável de quadros por segundo. [4] Com a qualidade gráfica dos jogos de hoje, este é um feito impressionante,

visto que motores de renderização para filmes podem levar de minutos a algumas horas para renderizar um único quadro.

Além disso, em algumas plataformas como PCs, outra responsabilidade do motor de jogos é implementar alguma forma de integrar o contexto de renderização da SDK com uma janela gráfica do sistema operacional. [4]

2.5.9.1 Processo de renderização

Para que uma cena possa ser renderizada, o ambiente deve ser matematicamente descrito na forma de primitivas gráficas, como linhas ou malhas tridimensionais de triângulos. [4]

Além disso, a câmera virtual deve ser posicionada neste ambiente. [4] Ela geralmente é composta de um ponto focal e de uma superfície localizada a uma curta distância deste ponto, que é onde a imagem será projetada.

O ambiente também deve possuir uma ou mais fontes de luz, para que emitam os raios que irão interagir com os objetos da cena e eventualmente atingir a superfície de projeção da imagem. [4]

Por fim, as propriedades visuais de cada objeto devem ser definidas para que os raios de luz interajam corretamente com os objetos. [4] Possuindo todas estas informações, o sistema de renderização calcula a cor final de cada *pixel* na tela onde a imagem será exibida.

Existem diversas técnicas para realizar todo este processo. [4] Algumas delas buscam a melhor qualidade gráfica possível, sem se importar com o tempo necessário para renderizar a cena. Para jogos eletrônicos, onde o tempo de cada ciclo é extremamente limitado, a tecnologia empregada deve sacrificar fidelidade gráfica em troca de velocidade de renderização.

2.5.9.2 Definição de objetos

O mundo real é composto de objetos, que podem ser opacos, transparentes ou translúcidos. [4] Como a luz não pode penetrar objetos opacos, a renderização precisa se preocupar apenas com suas superfícies. Já para objetos transparentes ou translúcidos, propriedades adicionais devem ser definidas para indicar como a luz é refletida, refratada ou dispersa ao passar pelo objeto. Por trazerem complexidade adicional, a maioria dos motores de jogos não se preocupa com estas propriedades, utilizando apenas um valor *alpha* para resumir o quão opaca é a superfície. Mesmo nuvens e fumaça são renderizadas utilizando superfícies. Podemos dizer então que a maioria dos sistemas de renderização estão preocupados apenas com a renderização de superfícies.

Muitas técnicas existem para definir superfícies matematicamente para que possam ser renderizadas. [4] Algumas superfícies podem ser representadas com uma equação paramétrica, mas estas não são adequadas para representar todos os possíveis formatos de um objeto. Outro modo é utilizando superfícies de subdivisão, como feito pela *Pixar* em seus filmes. Para renderização em tempo real, porém, o método mais interessante é utilizando malhas de triângulos, pois são a forma mais simples de polígono, possuindo o menor número de vértices necessários para produzir uma superfície. Triângulos são sempre planares, e continuam sendo triângulos após a maioria dos tipos de transformação. Além disso, praticamente todas as placas gráficas são projetadas e otimizadas para realizar varredura de triângulos. A união de diversos triângulos pode aproximar o formato de qualquer outra superfície.

Triângulos podem ser definidos com três vetores tridimensionais, representando a posição de seus três vértices. [4] Como mostrado na Figura 2.14, as arestas podem ser calculadas subtraindo os vetores por seus adjacentes, e o produto cruzado normalizado de duas arestas define a normal da face do triângulo.

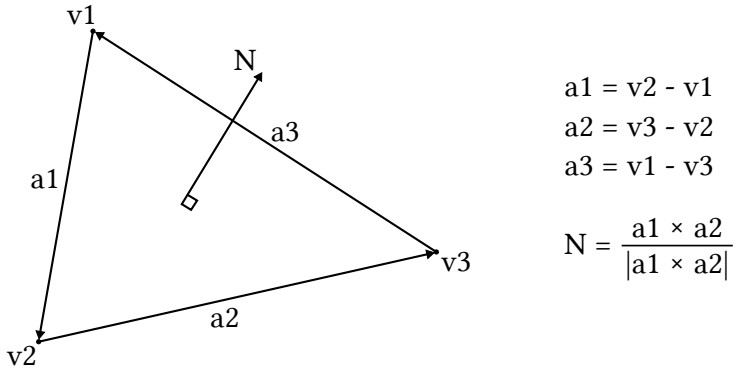

Figura 2.14: Triângulos podem ser definidos com três vértices, que são usados para calcular outros dados [4]

Para saber a direção da normal da face é necessário especificar qual face é a frontal e qual é a traseira. [4] Isto pode ser feito ao escolher um sentido de renderização para os triângulos: horário ou anti-horário. É importante que esta escolha seja consistente para todos os objetos do jogo, pois diversas APIs gráficas permitem escolher o sentido de renderização das superfícies para que possam otimizar o tempo de renderização, pulando o desenho das faces traseiras, que não são visíveis.

A malha de triângulos pode ser definida de diversas formas diferentes. [4] A mais simples delas é com uma primitiva chamada de lista de triângulos, que consiste em inserir sequencialmente os três vértices de cada triângulo da malha em uma lista. O problema desta abordagem é que várias superfícies podem compartilhar os mesmos vértices, e neste caso a lista

terá dados repetidos. Para solucionar este problema é possível utilizar uma lista indexada de triângulos, onde todos os vértices do objeto são inseridos em uma lista uma única vez, e outra lista é criada contendo índices para os vértices, em ordem para cada triângulo. Existem também outras primitivas, como faixas de triângulos e leques de triângulos.

2.5.9.3 Transformações

Para uma cena ser montada, as diversas malhas que representam os objetos devem ser posicionadas, orientadas e dimensionadas no espaço virtual do ambiente. [4] Para isto pode-se usar uma matriz 4 por 4 que armazena informações suficientes para transladar, rotacionar ou escalar o modelo tridimensional dentro do espaço virtual, chamada de matriz de transformação, que pode ser multiplicada por um vetor para calcular a posição final dele no ambiente, como exibido na Figura 2.15. A matriz identidade não produz nenhuma transformação. As matrizes para translação, escala e rotações em torno de cada eixo são mostradas nas Figuras 2.16, 2.17 e 2.18.

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} ax + by + cz + d \\ ex + fy + gz + h \\ ix + jy + kz + l \\ mx + ny + oz + p \end{bmatrix}$$

Figura 2.15: Multiplicação de uma matriz de transformação por um vetor [23]

$$T(t_x, t_y, t_z) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figura 2.16: Matriz de transformação para realizar uma translação [23]

$$E(e_x, e_y, e_z) = \begin{bmatrix} e_x & 0 & 0 & 0 \\ 0 & e_y & 0 & 0 \\ 0 & 0 & e_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figura 2.17: Matriz de transformação para realizar uma escala [23]

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sen\theta & 0 \\ 0 & sen\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}; R_y(\theta) = \begin{bmatrix} cos\theta & 0 & sen\theta & 0 \\ 0 & 1 & 0 & 0 \\ -sen\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}; R_z(\theta) = \begin{bmatrix} cos\theta & -sen\theta & 0 & 0 \\ sen\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figura 2.18: Matrizes de transformação para realizar uma rotação em torno dos eixos *x*, *y* e *z* [24]

Várias transformações podem ser combinadas sequencialmente ao multiplicar as matrizes. [23] A ordem em que as multiplicações são realizadas é importante, pois operações em ordem diferente não produzem o mesmo resultado. A transformação da matriz à direta da multiplicação é aplicada antes da transformação da matriz à esquerda.

Rotações através da multiplicação de três matrizes separadas utilizando ângulos de Euler estão sujeitas a diversos problemas. [25] Apesar de serem simples, a ordem em que são multiplicadas interfere no resultado final, e em alguns casos problemas piores podem acontecer. Um destes problemas é conhecido como *gimbal lock*, que ocasiona bloqueio em algumas rotações. Além disso, algumas operações são complicadas de realizar com ângulos de Euler, como interpolações.

Estes problemas podem ser solucionados com um sistema numérico chamado de quatérnio. [25] Quatérnios são capazes de armazenar rotações, e podem ser vistos como o conjunto de um eixo de rotação e um ângulo. Operações com quatérnios são mais simples, e portanto mais rápidas que operações utilizando ângulos de Euler, além de produzirem melhores resultados. Outra vantagem é que um quatérnio pode ser usado para gerar uma matriz de rotação sem problemas de ordem como com ângulos de Euler.

2.5.9.4 Câmera

A câmera de um jogo é geralmente tratada como um ponto focal e, flutuando a uma curta distância além dele, uma tela virtual. [4] Podemos imaginar esta tela como um retângulo contendo sensores virtuais de luz, representando os *pixels* na tela real onde a imagem será exibida. O processo de renderização tenta determinar que cor e intensidade de luz cada um destes sensores capturaria. De forma semelhante aos objetos do ambiente, a câmera também possui uma posição e orientação.

Os objetos tridimensionais devem ser projetados na tela bidimensional, e para isso utiliza-se uma matriz de projeção. [4] A matriz de projeção ortogonal preserva o tamanho dos

objetos em relação à sua distância da câmera, enquanto a perspectiva tenta imitar o efeito de câmeras reais, onde objetos que estão mais longe parecem ser menores, e por isto é o tipo de projeção mais usado em jogos eletrônicos. Estas projeções podem ser vistas na Figura 2.19.

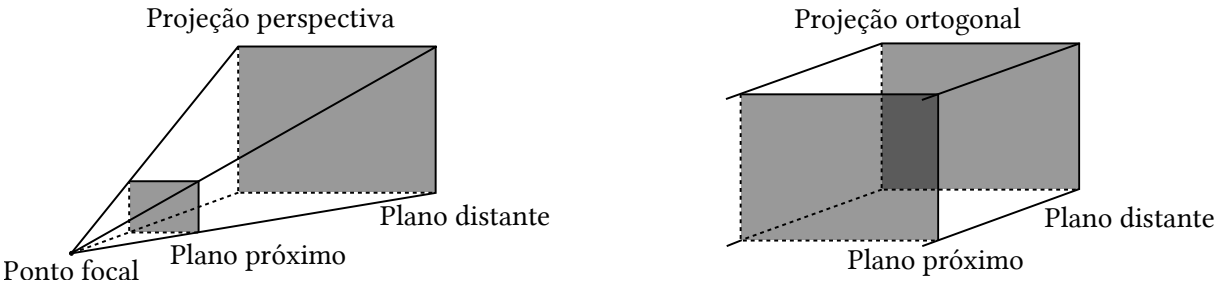

Figura 2.19: O volume de visualização nas projeções perspectiva e ortogonal (Do autor)

O espaço que a câmera consegue ver é chamado de volume de visualização, e é definido por seis planos. [4] O plano mais próximo é o plano que representa a tela virtual, enquanto o plano mais distante evita que objetos muito longe da câmera sejam desenhados, otimizando a renderização.

2.5.9.5 Iluminação

A luz interage com superfícies de quatro formas diferentes. [4] Ela pode ser absorvida, refletida, refratada ou difratada. A absorção ou reflexão da luz sobre uma superfície nos dá a percepção de cor em um objeto.

A reflexão pode ser difusa, especular e anisotrópica. [4] A reflexão difusa acontece quando um raio de luz é espalhado igualmente em todas as direções, a especular acontece quando o raio é refletido diretamente, ou é pouco espalhado em relação ao ângulo de onde atingiu a superfície, e a anisotrópica acontece quando a reflexão depende do ângulo de visualização.

Ao atravessar objetos transparentes ou translúcidos a luz pode ser espalhada, parcialmente absorvida ou refratada. [4] Em objetos semi-sólidos ela também pode atravessar uma parte da superfície, ser espalhada, e então retornar por um ponto diferente de onde entrou. Este efeito está presente em cera ou na pele, e é chamado de espalhamento sob superfície.

A reflexão da luz em uma superfície depende das propriedades da superfície e do ângulo de incidência do raio de luz. [4] Para auxiliar na identificação do ângulo, os modelos de iluminação definem um vetor normal à superfície. A direção deste vetor pode ter um grande impacto na aparência final do objeto.

O modelo de iluminação mais usado em sistemas de renderização é chamado de modelo de *Phong*, mostrado na Figura 2.20. [4] Ele define três tipos diferentes de reflexão, que

somados produzem a intensidade e cor final da superfície. O primeiro tipo é a reflexão ambiente, que tenta aproximar a quantidade de luz refletida indiretamente por toda a cena. O segundo tipo é a reflexão difusa, que representa a luz refletida uniformemente pela superfície para cada fonte de luz. O terceiro tipo é a reflexão especular, que representa a luz refletida em alinhamento com o ângulo de visão, formando pontos brilhantes na superfície. As propriedades de iluminação de um objeto são definidas em conjunto e armazenadas na forma de materiais, e um objeto pode ter mais de um material em partes diferentes da malha de triângulos.

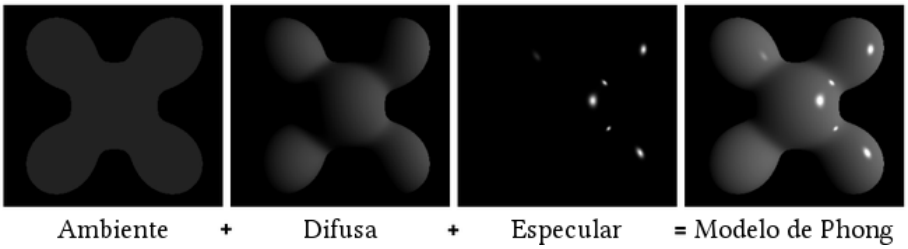

Figura 2.20: O modelo de iluminação de *Phong* [4]

2.5.9.6 Texturas

A forma mais simples de definir todos os parâmetros de uma superfície, como os vetores normais e materiais, é atribuindo-os aos vértices. [4] Porém, como os triângulos tendem a ser relativamente grandes, muita parte da superfície fica sem dados suficientes para um detalhamento melhor. É possível interpolar suas propriedades utilizando as informações dos vértices, mas isto pode gerar resultados indesejáveis. Esta limitação pode ser resolvida ao projetar texturas na malha, contendo as propriedades de iluminação da superfície.

Uma textura é medida em número de *texels*. [4] Dependendo da resolução da textura, ela pode projetar dentro de um único triângulo diversos *texels* contendo informações de iluminação. Os *texels* podem conter dados sobre os vetores normais, cor ambiente, difusa ou especular, ou o que for necessário para o cálculo da iluminação.

Para projetar uma textura bidimensional em uma superfície tridimensional, podemos definir um novo sistema de coordenadas. [4] Este sistema é chamado de espaço de textura, e utiliza um par de números (u, v) variando de 0 a 1 para representar uma posição dentro de uma textura de tamanho qualquer. Cada vértice do objeto pode então especificar uma coordenada (u, v), mapeando os triângulos da malha para o espaço de textura.

2.5.10 Sistema de áudio

O sistema de áudio é tão importante quanto o sistema de renderização para a imersividade de um jogo. [4] A maioria dos motores implementam interfaces para carregar, manipular e reproduzir sons, e variam bastante em termos de sofisticação. Mesmo que o motor de jogos utilize uma biblioteca ou sistema de áudio externo, é necessário bastante desenvolvimento e atenção para produzir áudio de alta-qualidade.

As responsabilidades de um sistema de áudio incluem carregamento de um ou mais formatos de arquivo de áudio, implementação de funções auxiliares para cálculo de efeitos sonoros, como reverberação, e reprodução dos dados através de um ou mais dispositivos. [26]

2.5.11 Base de jogabilidade

Por mais tecnologicamente avançado que seja um motor de jogos, sem elementos de jogabilidade, ele não é um jogo. [4] Para que um jogo possa ser construído com o motor, é necessário que ele forneça aos artistas formas de inserir regras de jogabilidade, terrenos, objetos, personagens, músicas, diálogos, entre outros, no jogo.

No início, os primeiros jogos tinham todo o conteúdo programado diretamente em seu código-fonte. [4] Por serem extremamente pequenos, esta prática não trazia problemas, mas com o tamanho atual da indústria surge a necessidade de produzir e inserir no jogo uma quantidade massiva de conteúdo.

Os jogos saíram então desta arquitetura rígida para uma arquitetura orientada a dados. [4] Sem necessidade de recompilar o código a cada alteração, o conteúdo do jogo pode ser modificado a qualquer momento, e a mudança pode ser observada imediatamente pelos artistas, melhorando a eficiência e economizando tempo no desenvolvimento do jogo, em troca de gastos para desenvolver o motor de jogos com uma arquitetura orientada a dados.

Além do conteúdo artístico do jogo, que pode ser carregado pelo motor de jogos na forma de arquivos pelos seus diversos sistemas, deve haver alguma forma de definir as mecânicas e regras de jogabilidade. [4] Existem diversos meios diferentes para fornecer esta interface, e cada motor de jogos aborda o problema de forma diferente, mas todos tentam dar aos desenvolvedores do jogo métodos para definir o ambiente do jogo, a sua progressão na forma de carregamento e transição de cenários, como os objetos do jogo são atualizados e interagem entre si e como o jogador interage com o ambiente do jogo, fornecendo acesso direto ou indireto aos sistemas do motor de jogos.

Para que os desenvolvedores definam estas operações de alto-nível do jogo, alguns motores são construídos na forma de um *framework*, enquanto outros oferecem suporte a linguagens de *script*. [4] Outras formas existem, e alguns motores podem até mesmo implementar várias soluções em conjunto.

2.5.12 Networking

Muitos jogos fornecem experiências multijogador através de um tipo de rede. [4] A forma de conexão entre os jogadores e a jogabilidade variam entre diferentes jogos, e enquanto alguns oferecem partidas rápidas entre poucos jogadores, outros chegam a unir milhares de personagens ao mesmo tempo em um único e imenso ambiente persistente.

A arquitetura destas redes varia muito de acordo com os requisitos do jogo. [27] Ela pode ser feita inteiramente *peer-to-peer* ou utilizando um servidor central, no formato cliente-servidor, e existem até mesmo modelos híbridos e os que utilizam uma rede de servidores centrais.

É importante que a decisão sobre a disponibilidade de uma camada de *networking* seja feita cedo durante o desenvolvimento, pois possui grande impacto em diversos outros componentes. [4]

Além disso, é importante que os desenvolvedores decidam o tipo de protocolo, TCP ou UDP, que será utilizado para a transmissão de dados. [17] Esta decisão depende do tipo de jogo que será construído, e cada um possui suas vantagens e desvantagens.

Enquanto o protocolo TCP fornece suporte a conexões, confiabilidade e ordenação, o modo como estas características são implementadas também o torna inviável para uso em alguns tipos de jogos, como jogos de tiro em primeira pessoa, onde os pacotes devem chegar o mais rápido possível ao seu destino, pacotes antigos podem ser descartados, e a ordem em que chegam, ou até mesmo se chegam, não é importante. [17]

Já o protocolo UDP, apesar de não oferecer as vantagens do TCP, pode ser implementado de forma otimizada para o tipo de jogo em que será usado, sendo praticamente a única alternativa para jogos que necessitam de velocidade na transmissão dos pacotes. [17] Sua desvantagem é que mais tempo e recursos devem ser gastos para implementar uma camada confiável o suficiente.

CAPÍTULO 3

IMPLEMENTAÇÃO

3.1 Ambiente de desenvolvimento

O motor de jogos foi desenvolvido na linguagem de programação C++, na versão do padrão C++11, e tendo como alvo a plataforma *Microsoft Windows 8.1*. Ele acompanha um projeto da IDE *Code::Blocks*.

3.2 Dependências

O projeto utiliza as bibliotecas *OpenGL*, *FreeGLUT* e *GLEW* para criar uma janela e fornecer acesso a funções de renderização, e utiliza a biblioteca *libpng* para realizar a leitura de arquivos PNG. A biblioteca *libpng* depende da biblioteca *zlib*. Todas as dependências estão anexadas ao projeto do motor.

3.3 Projeto de software

O motor de jogos foi separado em oito componentes principais. A arquitetura em camadas da Figura 3.1 e diagrama de pacotes da Figura 3.2 expõem as dependências entre eles.

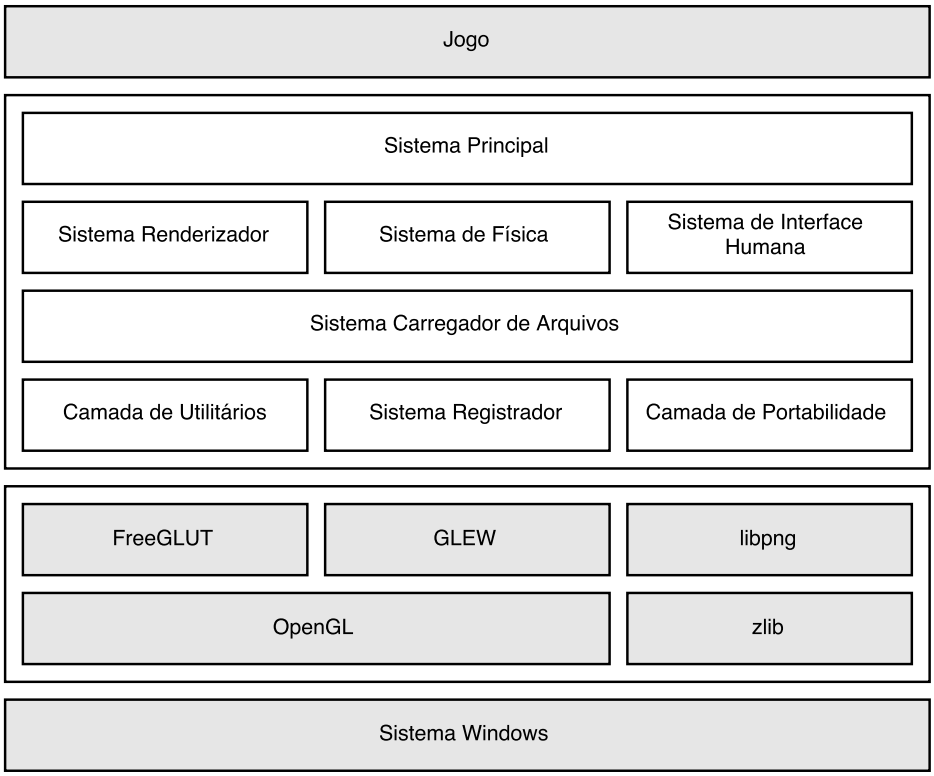

Figura 3.1: Arquitetura em camadas do motor de jogos (Do autor)

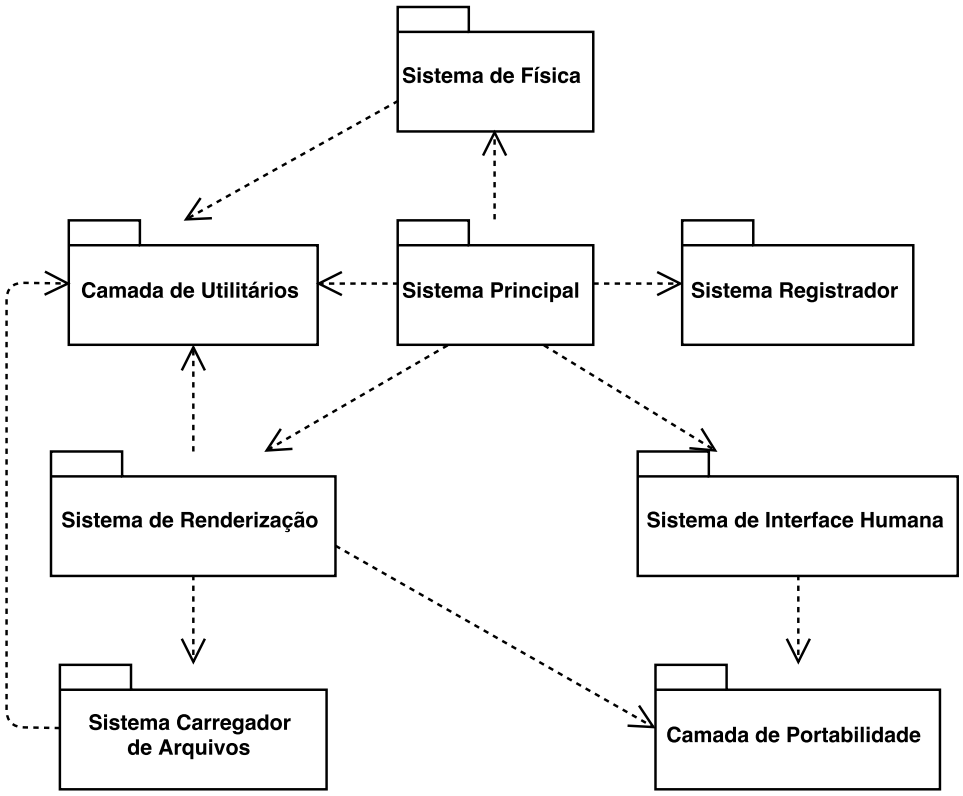

Figura 3.2: Diagrama de pacotes do motor de jogos (Do autor)

A camada de utilitários, mostrada na Figura 3.3, implementa algoritmos comuns e

estruturas de dados básicas, necessárias para o funcionamento dos outros sistemas do motor.

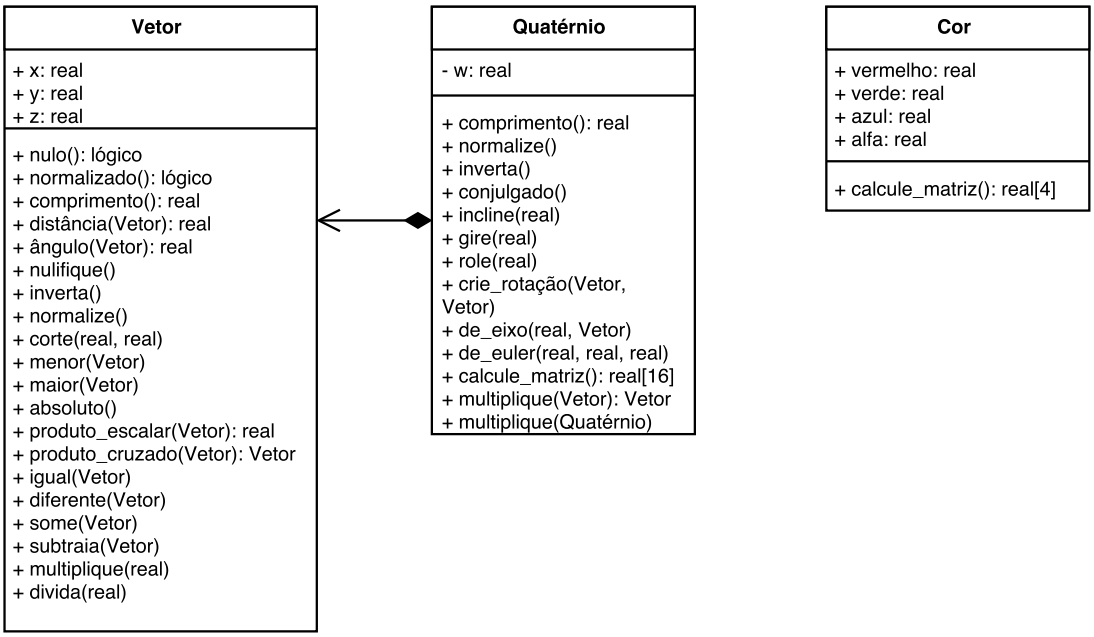

Figura 3.3: Diagrama de classes da camada de utilitários (Do autor)

O sistema carregador de arquivos, mostrado na Figura 3.4, é responsável por realizar o carregamento e processamento de dados dos tipos de arquivo suportados pelo motor para a memória.

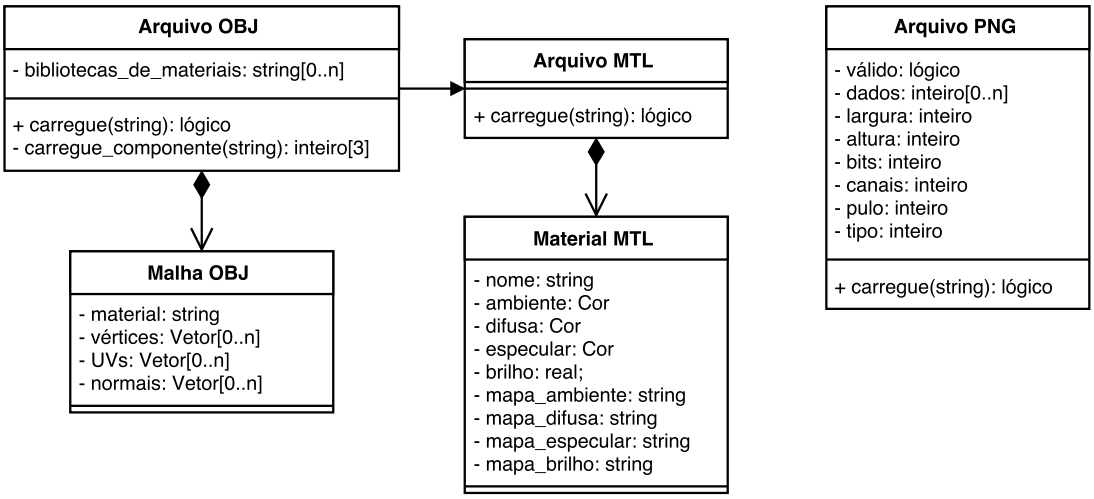

Figura 3.4: Diagrama de classes do sistema carregador de arquivos (Do autor)

O sistema principal, mostrado na Figura 3.5, controla a inicialização, finalização e fluxo de execução do ciclo principal do motor. Ele também fornece ao programador a base de jogabilidade na forma de *framework*, dando acesso às funcionalidades dos outros sistemas, além de implementar estruturas de alto-nível como céu e terreno.

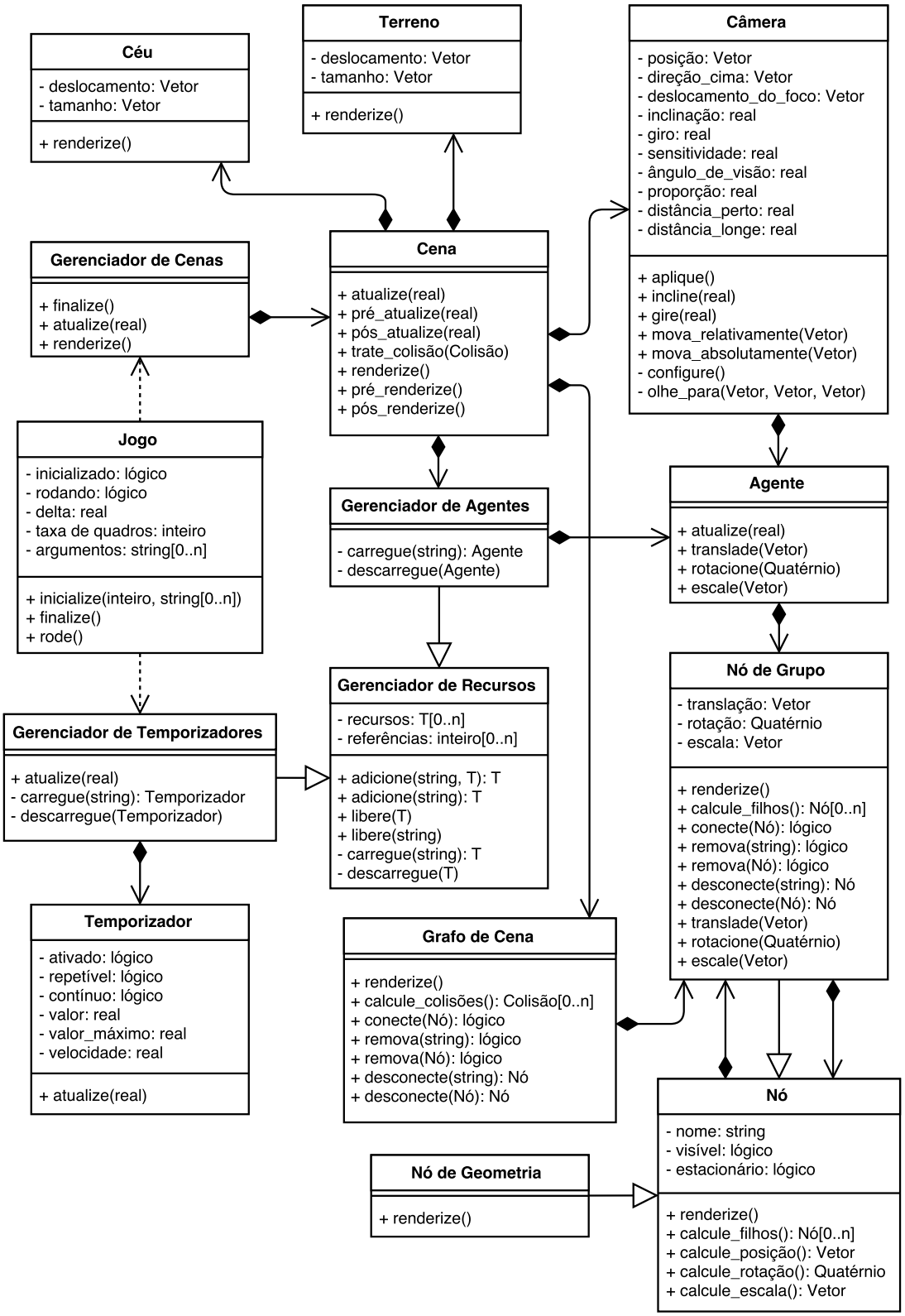

Figura 3.5: Diagrama de classes do sistema principal (Do autor)

O sistema registrador, mostrado na Figura 3.6, é responsável por propagar, exibir e armazenar mensagens de informação, erro ou depuração de todos os sistemas do motor ou do jogo em tempo de execução.

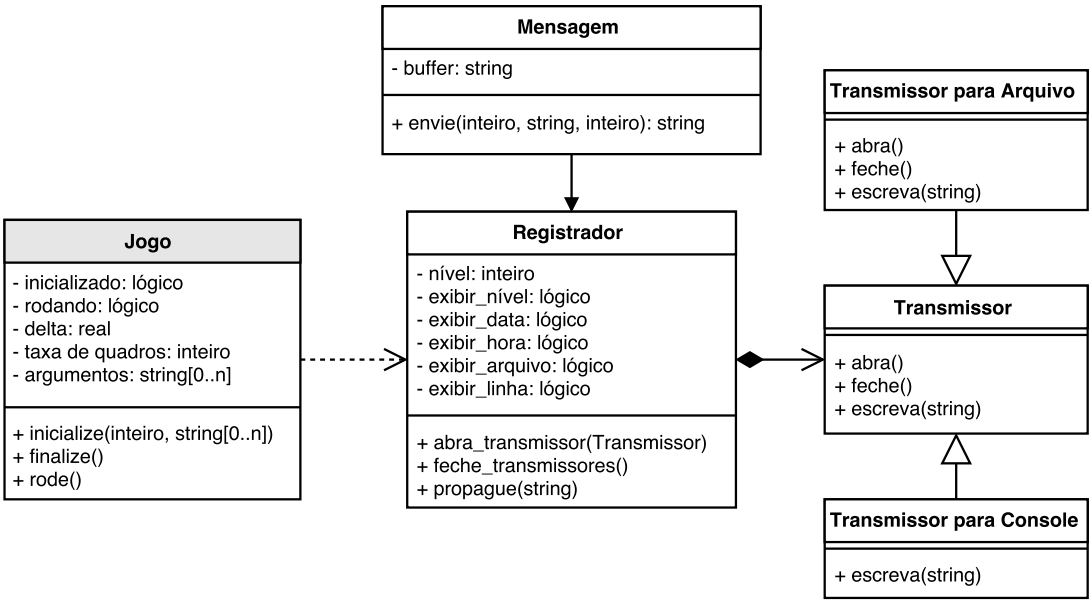

Figura 3.6: Diagrama de classes do sistema registrador (Do autor)

A camada de portabilidade, mostrada na Figura 3.7, fornece aos outros sistemas uma interface unificada para acesso de funcionalidades de baixo nível ou de dependências do motor.

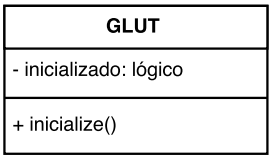

Figura 3.7: Diagrama de classes da camada de portabilidade (Do autor)

O sistema de renderização, mostrado na Figura 3.8, armazena dados de geometrias, materiais e texturas, além de criar e manter uma janela gráfica e fornecer acesso a funções de renderização.

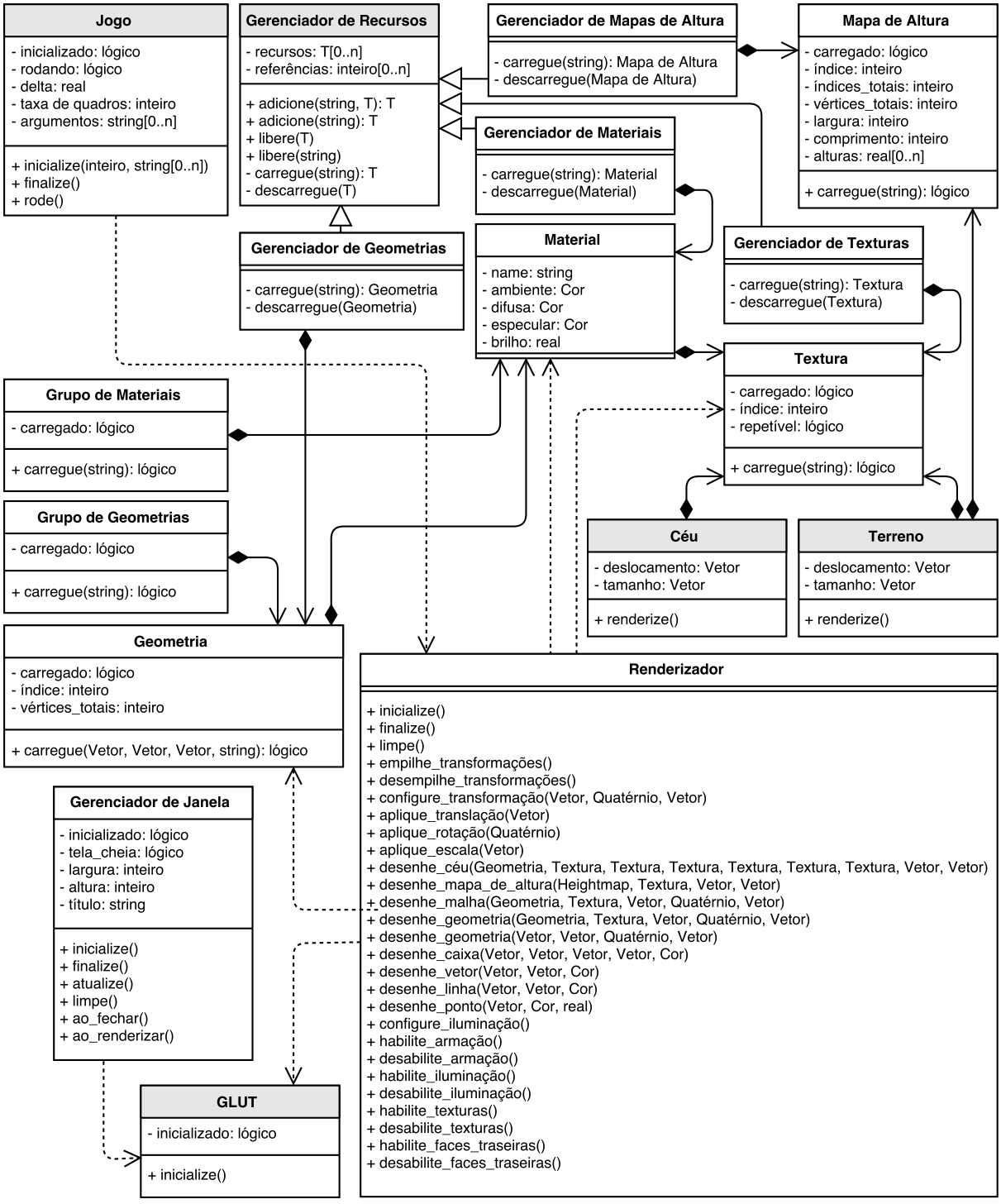

Figura 3.8: Diagrama de classes do sistema de renderização (Do autor)

O sistema de física, mostrado na Figura 3.9, implementa funções e estruturas de dados para detecção e resolução de colisões.

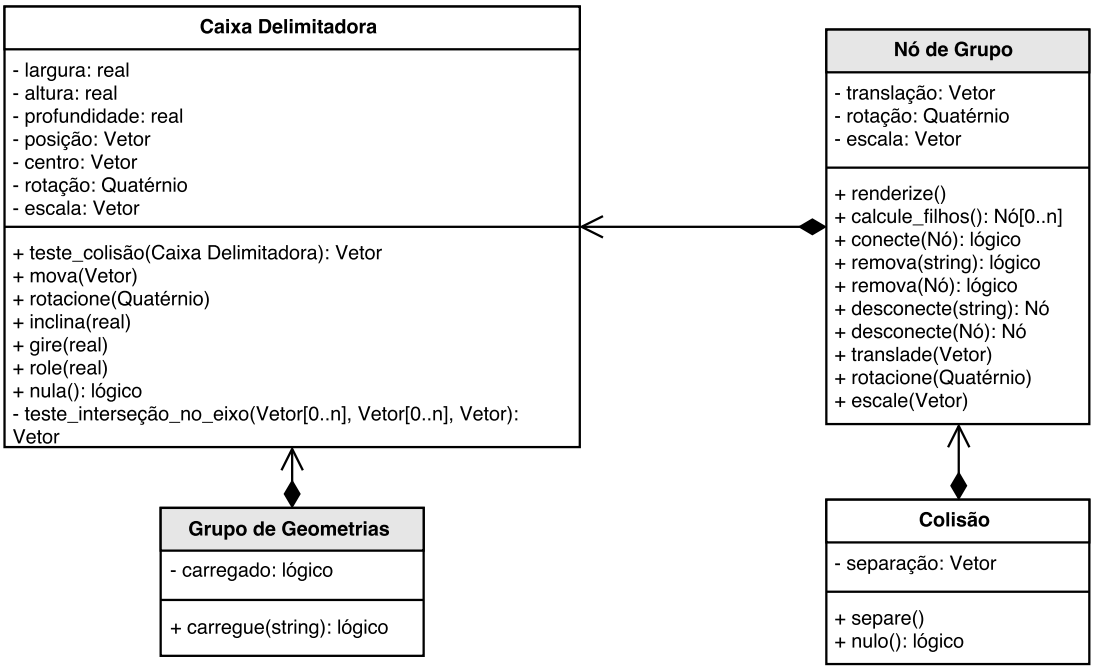

Figura 3.9: Diagrama de classes do sistema de física (Do autor)

O sistema de interface humana, mostrado na Figura 3.10, recebe e armazena comandos do jogador, fornecendo aos outros sistemas o estado atual do teclado e mouse.

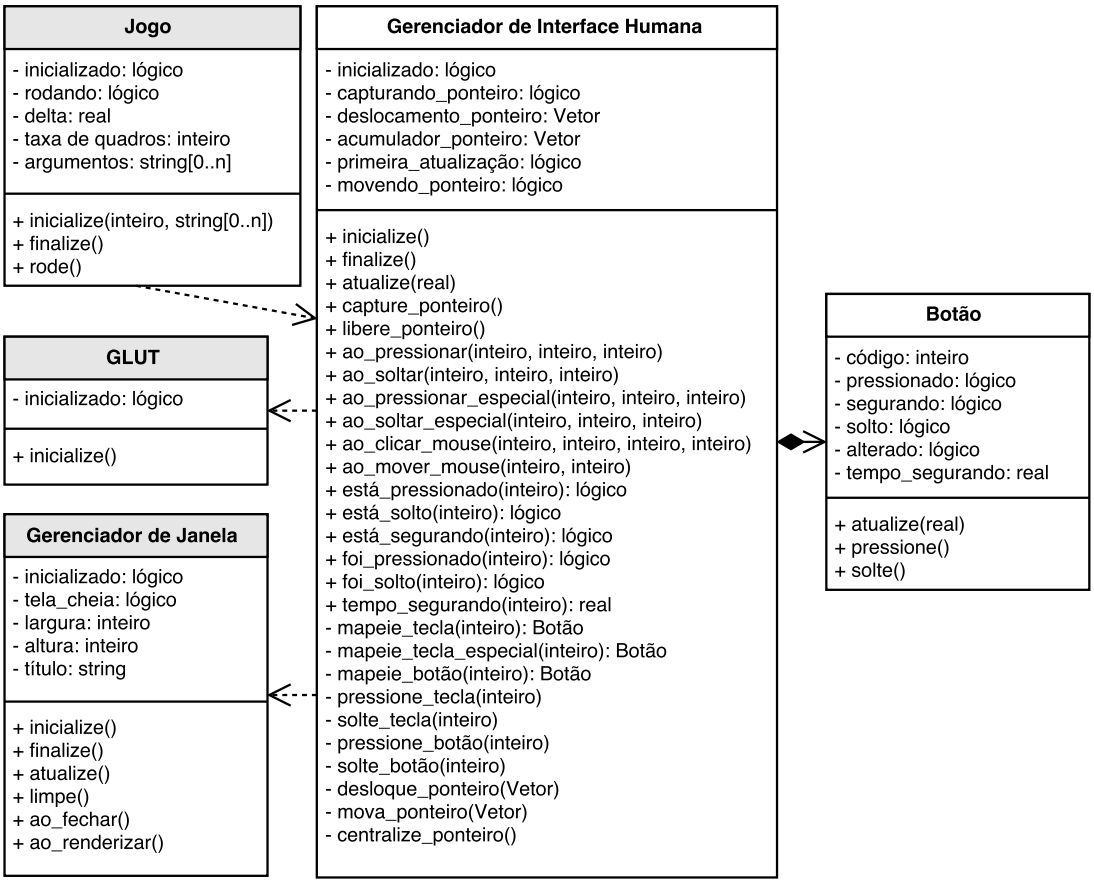

Figura 3.10: Diagrama de classes do sistema de interface humana (Do autor)

3.4 Implementação do motor de jogos

A inicialização e finalização do motor foram implementadas de forma simples. A inicialização apenas inicializa os componentes em ordem pré-estabelecida, e é responsável por configurar os sistemas, se necessário. A finalização dos componentes é executada logo após o término do ciclo principal, em ordem inversa da inicialização.

O ciclo principal utiliza a técnica de variação de tempo fixa controlada, sem interpolação. O tempo de cada quadro é calculado utilizando a biblioteca *chrono*, padrão da linguagem C++. Os eventos da biblioteca *FreeGLUT* são chamados, e então a simulação atualiza todos os temporizadores, o sistema de interface humana e a cena atual em intervalos de tempo fixos. Em seguida, a janela do jogo é limpada e o próximo quadro é renderizado.

No sistema registrador foi implementada uma lista de transmissores que é iterada para notificar cada transmissor quando uma mensagem é propagada. Antes de ser propagada, a mensagem é filtrada de acordo com seu nível e processada para adicionar mais informações, como data e hora, se necessário.

O gerenciador de cenas armazena uma referência para a cena atual do jogo, e é responsável por transferir requisições dos outros sistemas para esta cena, como atualizações e finalização da cena. As cenas possuem métodos virtuais que servem para definir o que deve ser executado antes e após uma atualização ou renderização, ou quando colisões são detectadas. Estes métodos podem ser sobrescritos pelo programador. A classe também mantém referências para um terreno, céu, câmera, grafo de cena e gerenciador de agentes.

Para facilitar a implementação de diversos sistemas de gerenciamento, foi criado um *template* de classe gerenciadora de recursos. Este *template* fornece uma interface para gerenciar uma lista de recursos de um tipo genérico e a quantidade de referências para eles, garantindo que cada recurso possua um nome único, apenas uma cópia em memória, e que seja devidamente destruído se não possuir nenhuma referência.

O gerenciador de agentes utiliza o *template* de gerenciador de recursos para executar o carregamento e armazenamento de entidades a partir de um arquivo OBJ. Cada entidade possui métodos virtuais de atualização e renderização que podem ser sobrescritos pelo programador. Elas também mantém uma referência para o nó de grupo que as representam no grafo de cena.

O gerenciador de temporizadores utiliza o *template* de gerenciador de recursos para armazenar as linhas do tempo do jogo. Cada temporizador representa uma linha do tempo que progride durante a simulação do ciclo principal, cada uma com seu ritmo de progressão. Para

elas foram implementados métodos que verificam se um limite foi atingido, quanto tempo se passou desde o início da contagem, além da capacidade de automaticamente voltar ao começo após atingir o limite determinado.

Para o grafo de cena, uma classe principal foi desenvolvida para armazenar o nó-raiz e fornecer métodos de manipulação e acesso ao grafo, como renderização de todas as malhas ou detecção de colisão entre os agentes. Três tipos de nós foram implementados: nós-base, nós de grupo e nós de geometria, sendo que os dois últimos herdam do primeiro. O programador pode implementar seus próprios tipos de nós se preferir, herdando de um destes três tipos. Os nós de grupo são responsáveis por armazenar transformações, como translações, rotações e escala, uma lista de nós-filhos e uma caixa delimitadora. Os nós de geometria são responsáveis por armazenar uma malha, e são sempre nós-folha, pois não aceitam nós-filhos.

Para representar a câmera virtual do jogo foi implementada uma classe de câmera. Ela possui uma posição, inclinação e giro, e pode ou não estar focada em uma entidade, possuindo uma referência para a entidade em foco. Nesta classe foram implementados métodos para manipular o volume de visualização, sua posição, inclinação, giro e foco. Ela também possui um método para aplicar estas transformações na câmera virtual antes da renderização.

Nas classes de terreno e céu foram implementados métodos para configurar e renderizar mapas de altura e céus no ambiente do jogo. A classe de terreno possui referências para uma textura e um mapa de altura, e a classe de céu possui referências a seis texturas, usadas nas faces internas de um cubo.

Na camada de utilitários foram implementadas classes para trabalhar com vetores, quatérnios e cores. As componentes x, y e z dos quatérnios são representadas por um vetor. A classe de cor trabalha com o modelo RGBA.

O sistema carregador de arquivos implementa métodos para carregamento de arquivos OBJ, MTL e PNG. Para arquivos PNG foi utilizada a biblioteca *libpng*, enquanto as classes de OBJ e MTL utilizam a biblioteca padrão da linguagem C++ para realizar a leitura e processamento dos arquivos em modo de texto.

A camada de portabilidade foi implementada apenas para fornecer aos sistemas de interface humana e renderização um ponto de acesso global para inicialização da biblioteca *FreeGLUT*.

No sistema de renderização, as classes de geometrias, materiais e texturas apenas armazenam os dados carregados dos arquivos OBJ, MTL e PNG, respectivamente. Para cada

uma destas classes foi implementado um gerenciador, utilizando o *template* de gerenciador de recursos. Alem destes, também foi implementada uma classe e um gerenciador para mapas de altura, que são carregados a partir de um arquivo PNG. Como um único arquivo OBJ ou MTL podem conter várias malhas ou materiais, foram criadas também classes para armazenar grupos de geometrias ou grupos de materiais, que possuem listas de referências para estes recursos.

Para manipular a janela do jogo foi desenvolvido um gerenciador de janela, responsável pela comunicação entre o motor de jogos e a biblioteca *FreeGLUT*. Ele implementa métodos para criar, atualizar e limpar a janela.

Foi implementada também uma classe renderizadora para realizar a comunicação com as bibliotecas *OpenGL*, *GLEW* e *FreeGLUT*. Ela implementa métodos para renderização de diferentes tipos de objetos e primitivas, além de aplicação de transformações.

Para o sistema de detecção de colisões foi desenvolvida uma classe de caixa delimitadora que utiliza a técnica de OBB, escolhida por aproximar com mais precisão o volume real do objeto mediante transformações, se comparada com AABBs, além de não ser necessário recalcular suas dimensões após transformações. Esta classe implementa os testes de colisão entre duas OBBs e permite que elas repliquem as translações, rotações e escalas aplicadas nos agentes através do grafo de cena.

A OBB de um agente é calculada durante o carregamento de sua malha tridimensional. Todos os vértices da malha são analisados e os menores e maiores valores em cada eixo do sistema de coordenadas são selecionados. No fim do carregamento, estes valores são comparados para determinar as dimensões e o ponto central da OBB.

Além disso, uma outra classe foi implementada para armazenar dados sobre uma colisão, que contém referências para os dois nós de grupo em colisão e o respectivo vetor de separação.

Por fim, para o sistema de interface humana foi implementada uma classe responsável por realizar a leitura dos dados da biblioteca *FreeGLUT* e armazenar uma lista de botões. A classe para os botões foi implementada para armazenar o estado de um único botão. Estas classes implementam métodos para verificar quando um botão está pressionado, solto ou segurado, e o tempo em que o botão permaneceu segurado. A classe de interface humana também possui métodos para manipular e ler dados de deslocamento do mouse.

CAPÍTULO 4

CASO DE USO E TESTES

Para que um jogo possa ser criado utilizando o motor de jogos desenvolvido, é necessário configurar os parâmetros do *linker* para vincular a biblioteca dinâmica do motor ao executável. A opção de linha de comandos para a suíte GCC é "-lengine". A Figura 4.1 mostra um exemplo desta configuração na IDE *Code::Blocks*.

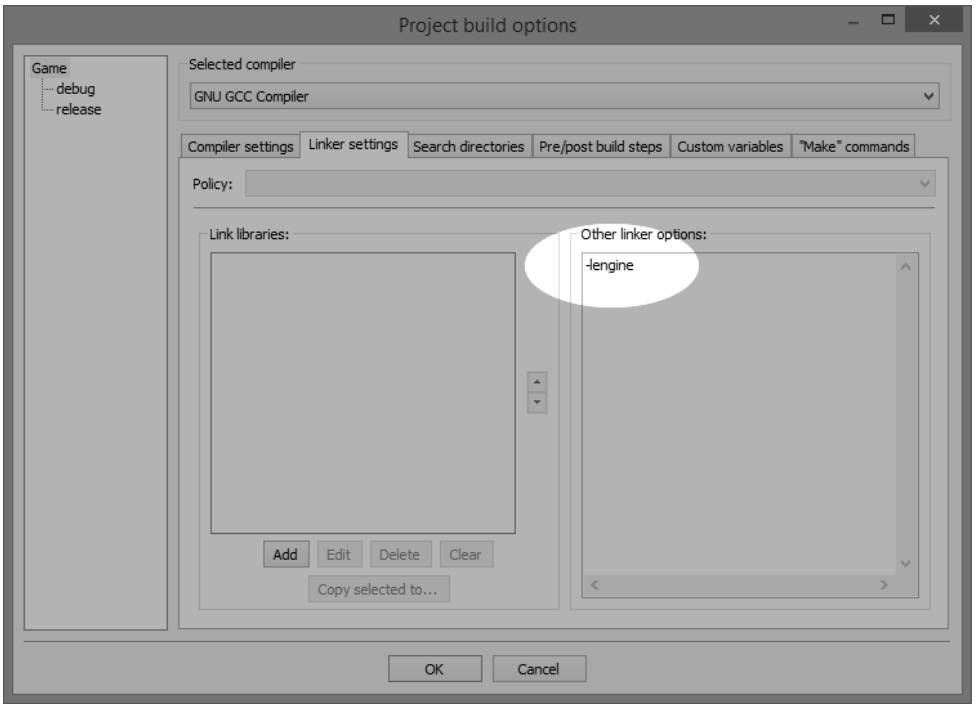

Figura 4.1: Configuração do *linker* para o projeto de um jogo na IDE *Code::Blocks* (Do autor)

Além disso, devem ser configurados também os caminhos para a localização das bibliotecas dinâmicas e arquivos de cabeçalho. Na suíte GCC, estes comandos correspondem a "-L<caminho>" e "-I<caminho>", respectivamente. As Figuras 4.2 e 4.3 mostram exemplos

destas configurações na IDE *Code::Blocks*.

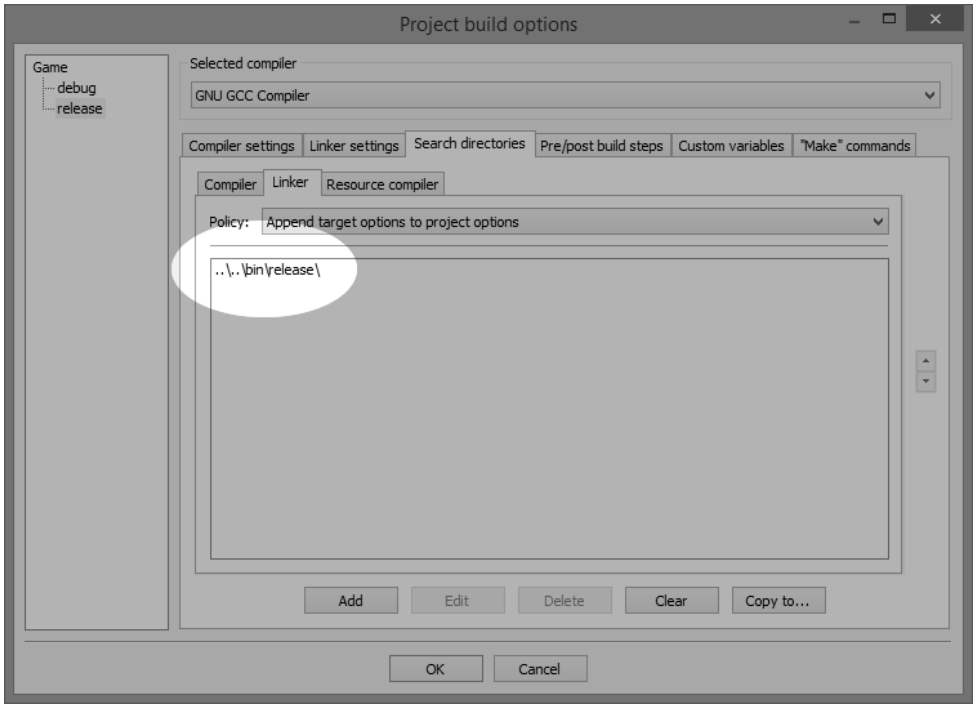

Figura 4.2: Configuração do diretório de importação de bibliotecas dinâmicas para o projeto de um jogo na IDE *Code::Blocks* (Do autor)

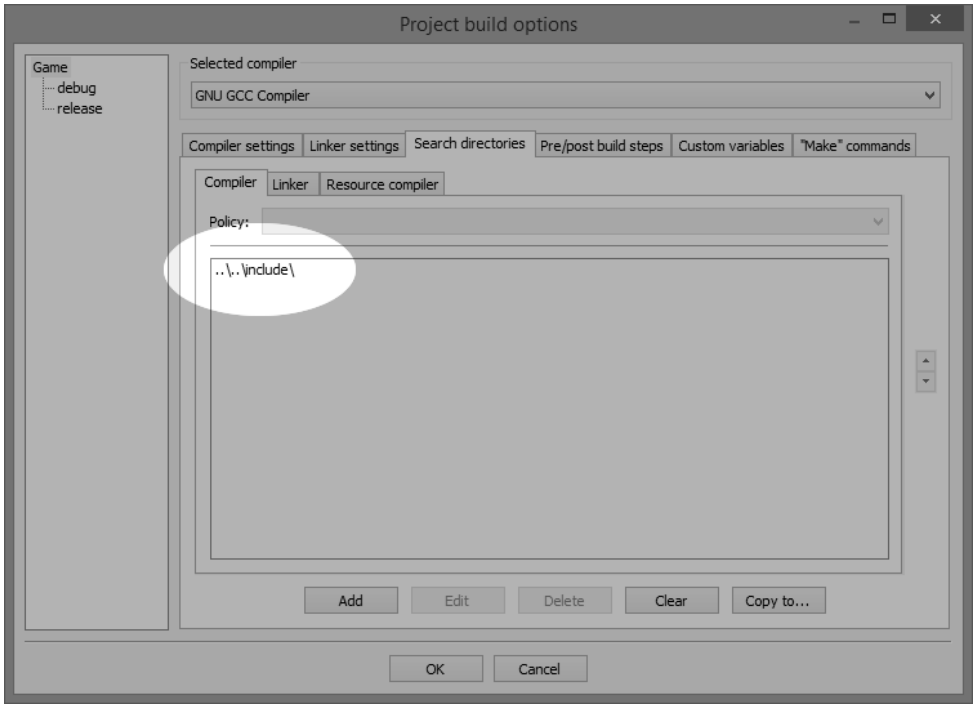

Figura 4.3: Configuração do diretório de importação de arquivos de cabeçalho para o projeto de um jogo na IDE *Code::Blocks* (Do autor)

Por fim, o executável do jogo deve ter acesso às bibliotecas dinâmicas do motor e

de suas dependências. Isto pode ser obtido ao copiar as bibliotecas para o mesmo diretório do executável.

4.1 Implementação de um jogo

Um jogo simples foi implementado como exemplo de utilização do motor de jogos desenvolvido. O jogo possui um agente representado por um carro, controlado pelo jogador, e um agente representado por uma pedra, posicionados em um ambiente contendo um terreno e um céu. Uma câmera virtual, posicionada atrás do carro, segue o agente do jogador. O carro reage a colisões com a pedra.

Este jogo tem como objetivo demonstrar a inicialização, execução e finalização do jogo, a implementação de uma cena personalizada, o acesso à interface humana para controle de um agente no jogo, a manipulação da câmera virtual, o uso das classes de terreno e céu, o uso de entidades e suas operações de transformação, e a interação entre agentes através de colisões. A Figura 4.4 mostra uma captura de tela do jogo implementado em execução.

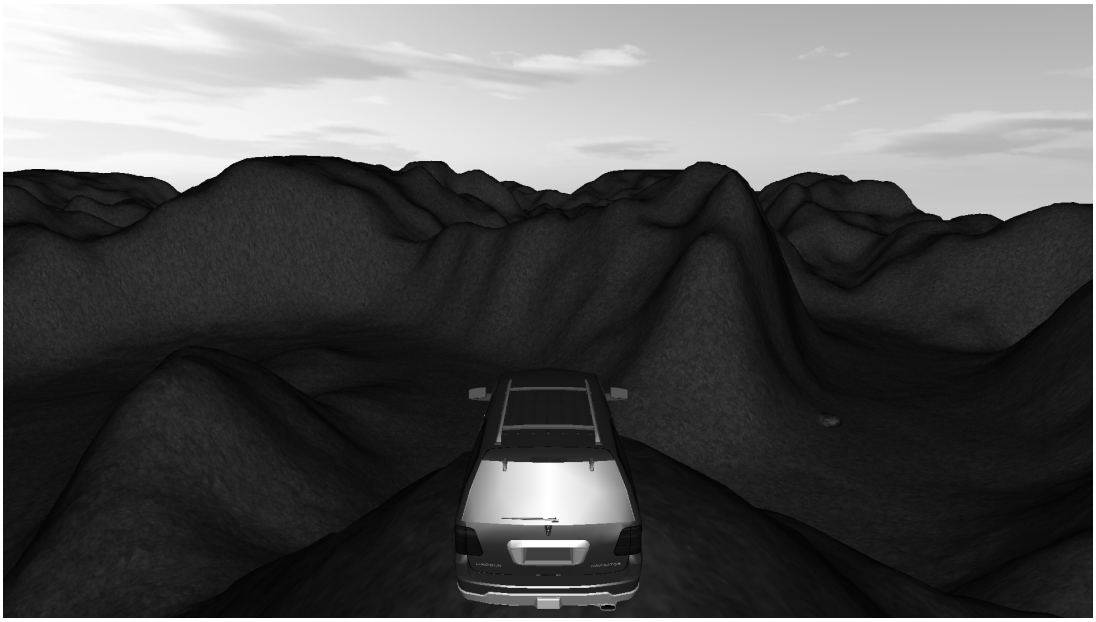

Figura 4.4: Jogo de exemplo criado com o motor de jogos desenvolvido (Do autor)

Para ter maior controle sobre a atualização do jogo é preciso criar uma cena personalizada. O motor fornece uma classe base *Scene*, da qual o programador pode herdar para implementar sua própria classe de cena. Os métodos “pre_update(float)”, “post_update(float)”, “pre_render()”, “post_render()” e “handle_collision(Collision&)” podem ser sobrescritos para alterar o comportamento da simulação, renderização, e reação a colisões. Para utilizar a classe

Scene, o programador deve importar o arquivo “engine/system/scene.hpp”. O Código-fonte 4.1 mostra a definição da classe de cena utilizada no jogo de exemplo desenvolvido.

```
1  #include <engine/system/scene.hpp>
2
3  class SceneMain: public Scene
4  {
5      public:
6          SceneMain();
7          ~SceneMain();
8
9          void pre_update(float delta);
10         void handle_collision(Collision& collision);
11
12     private:
13         Entity* m_object;
14
15         float m_speed;
16         float m_steer;
17 };
```

Código-fonte 4.1: Exemplo de definição de uma classe de cena personalizada (Do autor)

Para utilizar o sistema principal, o sistema de renderização, o sistema de interface humana e as classes de terreno e céu, é preciso incluir respectivamente os arquivos “engine/system/game.hpp”, “engine/graphics/graphics.hpp”, “engine/input/input.hpp”, “engine/system/terrain.hpp” e “engine/system/skybox.hpp”.

O construtor da classe de cena contém, além de sua implementação normal, a inicialização do ambiente e de suas entidades. O método “g_graphics.set_lights()” pode ser utilizado para inicializar uma configuração básica de iluminação.

Toda classe possui uma câmera virtual, que pode estar livre ou focada em uma entidade. Para atribuir um foco à câmera, utiliza-se o método da classe de cena “set_camera_focus(...)”, que recebe como argumento um ponteiro para a entidade que será o novo foco, ou *nullptr* para torná-la livre. O método “set_camera_focus_offset(Vector3f)” configura o deslocamento do ponto de foco da câmera em relação ao ponto de origem da entidade focada. O método “set_camera_position(Vector3f)” move a câmera para a posição determinada.

O método da classe de cena “set_terrain(...)” adiciona um terreno à cena, recebendo dois argumentos do tipo *string*. O primeiro indica o nome do arquivo de mapa de altura usado para construir a malha do terreno, enquanto o segundo indica o nome do arquivo de textura que o cobrirá. As dimensões do terreno podem ser manipuladas com o método “set_terrain_size(Vector3f)”.

O método da classe de cena “set_skybox(...)” pode ser utilizado para adicionar um céu ao jogo. Ele recebe até seis argumentos do tipo *string* contendo o nome dos arquivos de

textura utilizados nas faces frontal, traseira, esquerda, direita, superior e inferior do cubo, nesta ordem.

Uma nova entidade pode ser criada ao instanciar a classe *Entity* com o operador *new*. O construtor da classe *Entity* deve receber como argumento uma *string* contendo o nome do arquivo de modelo OBJ a ser carregado e um ponteiro para a raiz do grafo de cena, que pode ser obtido com o método da classe de cena "get_scene_graph_root()".

Para adicionar a entidade à cena, utiliza-se o método da classe de cena "add_entity(...)", que recebe dois argumentos. O primeiro, do tipo *string*, define o nome pelo qual a entidade será conhecida, e o segundo é um ponteiro para a entidade que deve ser adicionada. O nome da entidade pode ser personalizado, ou um valor quase garantidamente único pode ser gerado com a função auxiliar "uuid()".

A classe de entidades possui diversos métodos para manipular uma entidade. Um objeto pode ser transladado com "translate(Vector3f)", escalado com "scale(Vector3f)", ou rotacionado com "rotate(Quaternion)", "pitch(float)", "yaw(float)" e "roll(float)". Além disso, uma entidade pode ser visível e/ou estacionária. Uma entidade invisível não será renderizada, e esta propriedade pode ser definida com o método "set_visible(bool)". Um objeto estacionário não se moverá durante a separação de uma colisão, e esta propriedade pode ser definida com o método "set_stationary(bool)". As entidades podem definir também uma altura com "set_height(float)" para especificar sua distância vertical em relação ao terreno. Assim como com a classe de cena, o programador pode definir uma entidade personalizada ao herdar da classe *Entity* e sobrescrever o método "update(float)".

O Código-fonte 4.2 mostra o construtor utilizado na classe de cena do jogo de exemplo implementado. Em primeiro lugar, a iluminação do ambiente é configurada. Depois disso, um terreno e céu são adicionados à cena. O carro controlado pelo jogador é carregado do arquivo "navigator.obj" e armazenado no atributo "m_object" para que possa ser manipulado durante a atualização. Como o modelo tridimensional do carro é grande demais, e modelado com uma inclinação de 90 graus, é preciso também ajustar sua escala e inclinação. Ele é então posicionado e rotacionado no ambiente. A pedra é carregada do arquivo "rock.obj", posicionada na cena e configurada como estacionária, pois não se move em colisões. A câmera virtual é focada no carro com um deslocamento de 2 unidades no eixo *y* a partir de seu ponto de origem. Os atributos "m_speed" e "m_steer" controlam a velocidade e esterçamento do carro.

```
1 SceneMain::SceneMain()
```

```
2  {
3      g_graphics.set_lights();
4
5      set_terrain("heightmap.png", "terrain.png");
6      set_terrain_size(Vector3f(500.0, 50.0, 500.0));
7
8      set_skybox("front.png", "back.png", "left.png", "right.png", "top.png");
9
10     m_object = new Entity("navigator.obj", get_scene_graph_root());
11     add_entity(uuid(), m_object);
12
13     m_object->set_scale(Vector3f(0.4, 0.4, 0.4));
14     m_object->pitch(-90.0);
15     m_object->set_height(2.9);
16     m_object->set_translation(Vector3f(-125.0, 0.0, -160.0));
17     m_object->yaw(-40.0);
18
19     Entity* rock = new Entity("rock.obj", get_scene_graph_root());
20     add_entity("rock1", rock);
21     rock->set_translation(Vector3f(-138.0, 0.0, -133.0));
22     rock->set_stationary(true);
23
24     set_camera_focus(m_object);
25     set_camera_focus_offset(Vector3f(0.0, 2.0, 0.0));
26
27     m_speed = 0.0;
28     m_steer = 0.0;
29 }
```

Código-fonte 4.2: Exemplo de construtor de uma classe de cena personalizada (Do autor)

O destrutor da classe deve liberar as entidades adicionadas adequadamente, utilizando o método “release_entity(...)” e passando como argumento o ponteiro para a entidade a ser removida ou uma *string* contendo seu nome, para que as referências às entidades sejam removidas e os objetos possam ser descarregados. O Código-fonte 4.3 mostra o destrutor da classe de cena utilizada no jogo de exemplo implementado.

```
1  SceneMain::~SceneMain()
2  {
3      release_entity(m_object);
4      release_entity("rock1");
5  }
```

Código-fonte 4.3: Exemplo de destrutor de uma classe de cena personalizada (Do autor)

A classe de cena possui dois métodos de atualização que podem ser sobrescritos, “pre_update(float)” e “post_update(float)”, que são executadas em sequência pelo motor. Por padrão, o método “post_update(float)” translada todos os objetos para que estejam sempre acima do terreno e testa colisões entre todas as entidades. Por este motivo, é recomendado sobrescrever apenas o método “pre_update(float)”. O argumento *float* recebido equivale ao tempo decorrido desde o último quadro, ou *delta*, que pode ser utilizado para dar ao jogo uma referência de tempo em segundos.

O programador pode definir as regras de jogabilidade e o fluxo do jogo com estes métodos. Para decidir como progredir a simulação, é possível utilizar o sistema de in-

terface humana para verificar comandos do jogador, com os métodos “g_input.is_down(...)”, “g_input.is_up(...)”, “g_input.is_pressed(...)”, “g_input.is_holding(...)”, “g_input.is_released(...)” e “g_input.get_hold_time(...)”. Todos eles recebem como argumento a identificação de um botão.

O método do sistema principal “g_game.finalize()” pode ser chamado se o programador quiser finalizar o jogo, quando por exemplo, o jogador pressionar um botão.

O Código-fonte 4.4 mostra o método de pré-atualização da classe de cena utilizada no jogo de exemplo implementado. Em primeiro lugar, o carro é transladado para baixo em 50 unidades por segundo, para simular o efeito de gravidade. Se o carro for posicionado abaixo do terreno, a pós-atualização tratará de movê-lo corretamente. Os valores de velocidade e esterçamento são atualizados, perdendo força durante os quadros, para simular a resistência do ar e atrito do terreno. O estado das teclas *W*, *A*, *S* e *D* são analisados e os valores de velocidade e esterçamento são modificados adequadamente em resposta aos comandos do jogador. Outras verificações, que foram omitidas, garantem que a velocidade e esterçamento não extrapolem um limite configurado. A posição e rotação do carro são atualizadas a cada quadro, e a câmera é reposicionada para manter-se atrás do carro. Por fim, se o jogador estiver com tecla *escape* pressionada, o jogo é terminado.

```
1  void SceneMain::pre_update(float delta)
2  {
3      m_object->translate(Vector3f(0.0, -50.0, 0.0) * delta);
4
5      m_speed *= 1.0 - (0.6 * delta);
6      m_steer *= 1.0 - (5.0 * delta);
7
8      if (g_input.is_down(KEY_W)) {
9          m_speed += 1.2 * delta;
10     }
11
12     if (g_input.is_down(KEY_S)) {
13         m_speed -= 1.2 * delta;
14     }
15
16     [...]
17
18     if (g_input.is_down(KEY_A)) {
19         m_steer += 1.2 * delta;
20     }
21
22     if (g_input.is_down(KEY_D)) {
23         m_steer -= 1.2 * delta;
24     }
25
26     [...]
27
28     float speed_delta = m_speed * 15.0 * delta;
29     m_object->translate(m_object->get_final_rotation() * Vector3f(0.0, -1.0 *
           ↪ speed_delta, 0.0));
30
31     float steer_delta = m_steer * 15.0 * speed_delta;
32     m_object->yaw(steer_delta);
33
34     set_camera_position(m_object->get_final_translation() + m_object->
           ↪ get_final_rotation() * Vector3f(0.0, 8.0, 4.0));
35
36     if (g_input.is_down(KEY_ESCAPE)) {
```

```
37              g_game.finalize();
38          }
39      }
```

Código-fonte 4.4: Exemplo de pré-atualização de uma classe de cena personalizada (Do autor)

Para realizar o tratamento de colisões, é possível sobrescrever o método da classe de cena “handle_collision(Collision&)”. O objeto recebido por parâmetro contém informações e métodos necessários para resolver a colisão. O método “separate()” verifica quais objetos podem ser movidos e os translada utilizando o vetor de separação. Outras operações podem ser executadas para realizar a interação entre os objetos em colisão.

O Código-fonte 4.5 mostra o método de tratamento de colisão da classe de cena utilizada no jogo de exemplo implementado. A velocidade do carro é invertida e dividida pela metade, para simular a força contrária aplicada pela pedra no carro no momento da colisão, fazendo o carro recuar. Depois disso, a colisão é resolvida ao separar os objetos.

```
1  void SceneMain::handle_collision(Collision& collision)
2  {
3      m_speed = m_speed * -0.5;
4      m_steer = 0.0;
5
6      collision.separate();
7  }
```

Código-fonte 4.5: Exemplo de tratamento de colisão de uma classe de cena personalizada (Do autor)

Com a cena implementada, é necessário inicializar o motor de jogos e o ciclo principal, além de inserir nele a cena implementada. Para isto é preciso incluir o arquivo “engine/system/game.hpp”, que contém o sistema principal. O método “g_game.initialize(...)” inicializa os componentes do motor, e é importante que seja o primeiro método a ser chamado. Ele recebe como argumento um *int* contendo o número de parâmetros da linha de comandos e uma *array* de *strings* contendo os parâmetros. Cenas só podem ser construídas após a inicialização do motor, e para isto utiliza-se o operador *new*. O método “g_scene.set(...)” é utilizado para definir qual cena deve ser simulada pelo motor, e recebe como argumento um ponteiro para a cena. Com o motor inicializado e uma cena definida, o ciclo principal pode ser inicializado com o método “g_game.run()”. Este processo pode ser visto no Código-fonte 4.6, que mostra a inicialização e execução do jogo de exemplo implementado.

```
1  #include <engine/system/game.hpp>
2
3  [...]
4
5  int main(int argc, char* argv[])
6  {
```

```
7       g_game.initialize(argc, argv);
8
9       Scene* scene = new SceneMain();
10      g_scene.set(scene);
11
12      g_game.run();
13
14      return 0;
15  }
```

Código-fonte 4.6: Exemplo de inicialização e execução de um jogo (Do autor)

4.2 Testes de performance

Para a realização dos testes foram adicionadas chamadas a funções da biblioteca *chrono*, padrão da linguagem C++, diretamente no código-fonte do motor de jogos. Elas eram responsáveis pela geração de *timestamps* antes e depois de cada bloco de código a ser testado. Após o término da execução de cada bloco, a diferença entre os *timestamps* era calculada para obter o tempo total gasto nele. A taxa de quadros por segundo foi obtida diretamente do valor calculado internamente pelo motor de jogos. Os menores e maiores valores eram selecionados, e uma média era calculada a partir dos valores obtidos para cada quadro. Estes valores eram armazenados utilizando o sistema registrador.

O número de polígonos renderizados foi contado manualmente, a partir das faces contidas nos arquivos OBJ de cada modelo. O gasto de memória foi obtido no gerenciador de tarefas do sistema operacional *Microsoft Windows 8.1* durante a execução do ciclo principal.

As amostras foram coletadas a partir do jogo de exemplo implementado, executado em uma máquina com sistema operacional *Microsoft Windows 8.1* 64bit, processador de 3.5~4.2 GHz × 4, memória RAM de 16 GB e placa de vídeo de 1072~1137 MHz, 256bit e VRAM de 2048 MB. Os resultados dos testes são apresentados na Tabela 4.1.

Por executarem rápido demais e devido à falta de precisão nas operações de ponto flutuante, alguns testes podem apresentar resultado de 0 milissegundos.

Tempo de execução para amostragem	48,0505 segundos
Número de quadros amostrados[1]	50000 quadros
Número de polígonos renderizados por quadro	538480 triângulos
Gasto de memória durante a execução[2]	Entre 38,064 KB e 39,016 KB
Tempo gasto na inicialização dos componentes	99,363 milissegundos
Tempo gasto na finalização dos componentes	0 milissegundos
Tempo gasto no carregamento de arquivos	1,42265 segundos
Menor tempo gasto com a simulação	0 milissegundos
Maior tempo gasto com a simulação	2,002 milissegundos
Tempo médio gasto com a simulação	0,2631 milissegundos
Menor tempo gasto com a renderização	0 milissegundos
Maior tempo gasto com a renderização	3,003 milissegundos
Tempo médio gasto com a renderização	0,683478 milissegundos
Menor tempo gasto em um quadro	0,005 milissegundos
Maior tempo gasto em um quadro	4,858 milissegundos
Tempo médio gasto em um quadro	1,98343 milissegundos
Menor taxa de quadros por segundo	1032 quadros por segundo
Maior taxa de quadros por segundo	1067 quadros por segundo
Taxa média de quadros por segundo	1055,46 quadros por segundo

Tabela 4.1: Resultados obtidos com testes de performance sobre o jogo de exemplo (Do autor)

[1] Amostra coletada a partir do 10º quadro.
[2] Amostra coletada durante a execução do ciclo principal.

CAPÍTULO 5

CONCLUSÃO

Neste estudo foi possível notar quão grande e multidisciplinar se tornou a indústria de desenvolvimento de jogos, e quão importante é para os desenvolvedores possuírem um motor de jogos robusto e reusável que forneça a infraestrutura necessária para cortar os gastos de tempo e recursos na implementação dos diversos componentes comuns em um jogo.

Este trabalho apresentou diversas técnicas para solucionar problemas nos principais sistemas, que estão presentes em praticamente todos os jogos eletrônicos modernos. Vimos também que estas técnicas não são absolutas, possuindo muito espaço para pesquisa, e que a escolha das estruturas de dados e soluções mais adequadas para um projeto é muitas vezes situacional, variando com o gênero de jogo visado pelo motor. Além disso, podemos perceber com sistemas como o de renderização e detecção de colisões que os desenvolvedores devem constantemente colocar na balança diversos fatores, como otimização em troca de extensibilidade, ou desempenho em troca de precisão.

Vimos que é importante que o motor tenha uma arquitetura consistente, eliminando sempre que possível dependências cíclicas entre os componentes. Além disso, pudemos notar que a linha que separa motores de jogos dos jogos que podem ser construídos sobre eles é bastante vaga, sendo definida por cada projeto de motor de jogos a partir de sua arquitetura.

O motor de jogos implementado durante este trabalho provou-se suficiente para a construção de jogos extremamente simples, apresentando desempenho aceitável nos testes de performance realizados, mas que ainda pode ser expandido de muitas formas.

Esta pesquisa ajudou a entender melhor como surgiu e evoluiu a indústria de *video games*, como um jogo funciona por trás dos panos, e como os sistemas de um motor trabalham

juntos para trazer à vida as experiências interativas proporcionadas pelos jogos eletrônicos.

5.1 Dificuldades encontradas

Apesar de esperado, em muitos pontos da implementação foi frustante tomar decisões sobre como separar o motor desenvolvido de seus jogos, e até que ponto expor, ou como expor, a interface dos sistemas na forma de um framework.

Decisões sobre como realizar a interação entre os diferentes sistemas e resolver dependências entre eles se provou igualmente desafiador. O motor foi projetado e desenvolvido sem foco em um gênero ou tipo específico de jogo, e a tentativa de torná-lo extremamente genérico dificultou este tipo de decisão.

Diversos erros e comportamentos inesperados foram detectados durante o desenvolvimento do motor. Enquanto muitos deles foram corrigidos, um comportamento peculiar durante a separação de dois objetos em colisão ainda está presente, que pode ser notado no jogo de exemplo desenvolvido. Ao colidir o carro com a pedra em certos ângulos, o carro não sofre recuo corretamente, ficando preso à pedra por alguns quadros. Isto sugere que o vetor de separação foi calculado erroneamente em algum caso e, por ser pequeno demais, não separa os dois objetos suficientemente longe para que não estejam mais colidindo no próximo quadro.

5.2 Trabalhos futuros

Este trabalho apresentou a arquitetura de alto-nível de um motor de jogos, e resumiu o funcionamento dos diferentes sistemas e suas responsabilidades. Um motor de jogos é uma ferramenta extremamente grande, envolvendo diversas áreas que, sozinhas, precisariam de um estudo completo para serem entendidas profundamente. Muitas das tecnologias presentes em motores de jogos comerciais não foram implementadas no motor desenvolvido, e as que foram provavelmente se tornarão obsoletas conforme a indústria evolui. Algumas áreas onde este estudo poderia ser expandido incluem:

- Testar, depurar e consertar erros que possam ser eventualmente encontrados no motor implementado, torná-lo mais portável e extensível, e fornecer uma interface mais abstrata e simples de usar;
- Dar uma explicação mais aprofundada sobre sistemas que não foram completamente cobertos neste estudo, como áudio, *networking* e *scripting*, além de implementar estes sistemas no motor desenvolvido;

- Fornecer suporte a *shaders* programáveis pelo sistema de renderização, permitindo a implementação de diversas técnicas mais avançadas, como iluminação fotorrealista, como animações por esqueletos e múltiplas texturas aplicadas no terreno;
- Fornecer suporte a mais formatos de arquivos, como formatos de modelos tridimensionais, texturas ou arquivos de configuração.
- Fornecer suporte a mais tipos de periféricos, como *joypads*;
- Implementar técnicas e estruturas de dados mais avançadas do que as atualmente usadas, melhorando e otimizando os sistemas implementados, além de manter o motor a par com avanços tecnológicos na área;
- Implementar um sistema avançado de simulação física, utilizando um modelo de forças.
- Implementar um modelo de iluminação mais robusto, permitindo ao programador configurar vários emissores de luz.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Guinness World Records. *Highest revenue generated by an entertainment product in 24 hours*. Último acesso em 24 de setembro de 2015. Disponível em: http://www.guinnessworldrecords.com/world-records/88321-highest-revenue-generated-by-an-entertainment-product-in-24-hours

[2] CHATFIELD, T. *Videogames now outperform Hollywood movies*. Último acesso em 24 de setembro de 2015. Disponível em: http://www.theguardian.com/technology/gamesblog/2009/sep/27/videogames-hollywood

[3] MCKAY, D. R. *Jobs in the Game Industry*. Último acesso em 24 de setembro de 2015. Disponível em: http://careerplanning.about.com/od/occupations/a/videogamecareer.htm

[4] GREGORY, J. *Game Engine Architecture*. A K Peters/CRC Press, 1ª edição, 2009.

[5] MASTERS, M. *Unity, Source 2, Unreal Engine 4, or CryENGINE - Which Game Engine Should I Choose?* Último acesso em 24 de setembro de 2015. Disponível em: http://blog.digitaltutors.com/unity-udk-cryengine-game-engine-choose/

[6] Springrts.com. *About - Spring*. Último acesso em 24 de setembro de 2015. Disponível em: https://springrts.com/wiki/About

[7] KING, Y. *Opinion: Why On Earth Would We Write Our Own Game Engine?* Último acesso em 24 de setembro de 2015. Disponível em: http://gamasutra.com/view/news/128765/

[8] TEIXEIRA, M. *Desenvolvimento de Jogos no Brasil*. Último acesso em 24 de setembro de 2015. Disponível em:
https://techinbrazil.com.br/desenvolvimento-de-jogos-no-brasil

[9] STANTON, R. *A Brief History of Video Games*. Running Press Book Publishers, 2015.

[10] Brookhaven National Laboratory. *Tennis For Two*. Último acesso em 25 de setembro de 2015. Disponível em:
https://www.flickr.com/photos/brookhavenlab/3148601792/

[11] MIT Museum Collections. *Spacewar*. Último acesso em 25 de setembro de 2015. Disponível em:
http://museum.mit.edu/150/25

[12] Centre for Computing History. *Magnavox Odyssey*. Último acesso em 25 de setembro de 2015. Disponível em:
http://www.computinghistory.org.uk/det/16909/

[13] MONELLO, M. *Computer Space*. Último acesso em 25 de setembro de 2015. Disponível em:
https://www.flickr.com/photos/campfiremike/8311851950

[14] CAITO, J. *Atari Changed Gaming History*. Último acesso em 25 de setembro de 2015. Disponível em:
http://guardianlv.com/2014/04/atari-changed-gaming-history/

[15] EDDY, B. *Doom (id Software, 1993)*. Último acesso em 26 de setembro de 2015. Disponível em:
http://www.newretrowave.com/game-reviews/2015/7/16/doom-id-software-1993

[16] NYSTROM, R. *Game Programming Patterns*. Genever Benning, 2014.

[17] FIEDLER, G. *Fix Your Timestep!* Último acesso em 28 de setembro de 2015. Disponível em:
http://gafferongames.com/game-physics/fix-your-timestep/

[18] MACIAK, L. *What is your favorite config file format?* Último acesso em 29 de setembro de 2015. Disponível em:

http://www.terminally-incoherent.com/blog/2013/08/28/what-is-your-favorite-congfig-file-format/

[19] FOSTER, G. *Understanding and Implementing Scene Graphs*. Último acesso em 11 de outubro de 2015. Disponível em: http://archive.gamedev.net/archive/reference/programming/features/scenegraph/index.html

[20] BAR-ZEEV, A. *Scenegraphs: Past, Present, and Future*. Último acesso em 13 de outubro de 2015. Disponível em: http://www.realityprime.com/blog/2007/06/scenegraphs-past-present-and-future/

[21] ORLAND, K. *Head-to-head: Everything you need to know in the PS4 vs. Xbox One battle*. Último acesso em 14 de outubro de 2015. Disponível em: http://arstechnica.com/gaming/2013/11/head-to-head-everything-you-need-to-know-in-the-ps4-vs-xbox-one-battle/

[22] ERICSON, C. *Real-Time Collision Detection*. CRC Press, 2004.

[23] *Tutorial 3: Matrices*. Último acesso em 17 de outubro de 2015. Disponível em: http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/

[24] MURRAY, G. *Rotation About an Arbitrary Axis in 3 Dimensions*. Último acesso em 17 de outubro de 2015. Disponível em: http://inside.mines.edu/fs_home/gmurray/ArbitraryAxisRotation/

[25] *Tutorial 17: Rotations*. Último acesso em 17 de outubro de 2015. Disponível em: http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-17-quaternions/

[26] BOER, J. *Game Audio Programming*. Cengage Learning, 2002.

[27] ARMITAGE, G., CLAYPOOL, M., BRANCH, P. *Networking and Online Games: Understanding and Engineering Multiplayer Internet Games*. John Wiley & Sons, 2006.