THIAGO MATIAS BUSSO

SOLUÇÕES REUTILIZÁVEIS NO DOMÍNIO DE JOGOS COMPUTACIONAIS: A APLICAÇÃO DE PADRÕES DE PROJETO NO DESENVOLVIMENTO DE MOTORES DE JOGOS

São Paulo
2006

THIAGO MATIAS BUSSO

SOLUÇÕES REUTILIZÁVEIS NO DOMÍNIO DE JOGOS COMPUTACIONAIS: A APLICAÇÃO DE PADRÕES DE PROJETO NO DESENVOLVIMENTO DE MOTORES DE JOGOS

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Mestre em Engenharia.

Área de Concentração: Sistemas Digitais

Orientadora:
Profª. Dra.
Maria Alice Grigas Varella Ferreira

São Paulo
2006

Este exemplar foi revisado e alterado em relação à versão original, sob responsabilidade única do autor e com a anuência de seu orientador.

São Paulo, ... de ..... de .....

Assinatura do autor

Assinatura do orientador

.

FICHA CATALOGRÁFICA

Busso, Thiago Matias
Soluções reutilizáveis no domínio de jogos computacionais: a aplicação de padrões de projeto no desenvolvimento de motores de jogos / T. M. Busso. – ed. rev. – São Paulo, 2006.
131 p.

Dissertação (Mestrado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia da Computação e Sistemas Digitais.

1. Padrões de Software. 2. Jogos de Computador. I. Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia da Computação e Sistemas Digitais. II. t.

Dedico este trabalho aos meus pais, Ana e Ary, pelo apoio dado em todos esses anos.

AGRADECIMENTOS

À Prof. Dra. Maria Alice Grigas Varella Ferreira, por toda a dedicação, paciência e atenção dispensada ao longo de todo o Mestrado.

À minha namorada, Anna Flavia, pela compreensão nas horas difíceis e ajuda na revisão deste texto.

Aos meus amigos, Rodrigo Souza e Edson Saraiva, pelas discussões sobre *patterns*.

Aos meus amigos, Marcelo e Carlos Eduardo, pelo apoio e pelas palavras de incentivo.

Aos companheiros da Scopus, pelas dicas, caronas e apoio.

“Low-level programming is good for

the programmer's soul”

John Carmack

Resumo

BUSSO, T. M. **Soluções Reutilizáveis no Domínio de Jogos Computacionais: A Aplicação de Padrões de Projeto no Desenvolvimento de Motores de Jogos**. 2006. 131 f. Dissertação (Mestrado) – Escola Politécnica, Universidade de São Paulo, São Paulo, 2006.

Com o desenvolvimento da indústria de jogos computacionais, cresceu também o custo de produção destes jogos, a cada dia mais complexos, inviabilizando sua construção a partir do "nada". Isto fez com que as empresas de jogos passassem a desenvolver ou adquirir soluções reutilizáveis. Nesse trabalho, realizou-se um estudo sobre o emprego de reuso em jogos e, como conclusão, apresenta-se a forma como os padrões de projeto (*design patterns)* podem contribuir para o projeto da arquitetura de jogos computacionais, enfocando o desenvolvimento do núcleo de software que os compõe, denominado motor de jogo ou *game engine*. Para facilitar a manutenção e evolução do sistema, este componente deve apresentar as características de ser modular e extensível, além de algumas outras propriedades discutidas ao longo do texto. Apresenta-se o que é um motor de jogo e quais os módulos que o compõem, descrevendo-se, em seguida, os padrões de projeto e suas aplicações. Como experimento, analisa-se o uso de padrões de projeto por motores de jogo, utilizando-se, para isso, os que possuem código aberto ou que estão disponíveis na literatura, dando-se ênfase à abstração da API gráfica e observando-se os aspectos positivos e negativos de cada solução. Com base nessa análise, propõe-se uma solução que visa atender aos requisitos considerados relevantes para um motor do jogo.

Palavras-chave: Padrões de Projeto, motores de jogos, jogos eletrônicos.

Abstract

BUSSO, T. M. **Reusable Solutions in the Computer Games Domain: The Practical Use of Design Patterns in the Development of Game Engines.** 2006. 131 p. Thesis (Master's Degree) – Escola Politécnica, Universidade de São Paulo, São Paulo, 2006.

The development of computer games industry increased the costs of production of such games, that every day are more complex, and made almost impracticable their construction from the scratch. As a result, those companies started to develop or acquire reusable solutions. This study shows the application of reuse in games and, as a conclusion, it demonstrates how design patterns can contribute in a project of computer games, focusing on the development of the software core that composes them, called game engine. In order to facilitate the maintenance and evolution of the system, this device should be modular and extensible, beyond other characteristics discussed in this text. It is also presented the definition of a game engine, specifying the modules that compile it, describing, after that, the design patterns and their applications. As an experiment, it is analyzed the use of design patterns in already existing game engines, making use of the ones that are open source or that are available in the literature. In this experiment, it is emphasized the abstraction of the graphical API, pointing out the positive and negative aspects of each solution. Based on this analysis, it is proposed a solution which aims to fulfill the requirements considered relevant in a game engine.

Keywords: Design Patterns, Game Engine, Computer Games.

Lista de Figuras

Lista de Tabelas

Lista de Siglas

AI	–	*Artificial Intelligence*
API	–	*Application Programming Interface*
ARB	–	*Architectural Review Board*
COM	–	*Microsoft Component Object Model*
CORBA	–	*Common Object Request Broker Architecture*
GDC	–	*Game Developers Conference*
GPU	–	*Graphics Processing Unit*
HLSL	–	*High-Level Shader Language*
LAN	–	*Local Área Network*
LOD	–	*Level Of Detail*
MIT	–	Massachusetts Institute of Technology
MMO	–	*Massive Multiplayer Online*
NPC	–	*Non Player Characters*
OPENGL	–	*Open Graphics Library*
ORB	–	*Object Request Broker*
PPU	–	*Physics Processing Units*
QA	–	*Quality Assurance*
SGI	–	Silicon Graphics Incorporated
TI	–	Tecnologia da Informação

Sumário

1. Introdução

Jogos de computador são exemplos de sistemas de software que atraem milhões de pessoas e movimentam bilhões de dólares ao ano (BATTAIOLA et al., 2002). Também conhecidos como Jogos Computacionais, Jogos Eletrônicos ou Videogames, podem ser definidos como "qualquer jogo que pode ser simulado em um dispositivo computacional, seja ele analógico ou digital, mecânico ou eletrônico" (BIANCHINI, 2005).

Assim sendo, esta é uma área de pesquisa e desenvolvimento que se torna a cada dia mais atraente e promissora, tendo em vista o crescimento da indústria mundial de entretenimento eletrônico e a evolução das técnicas utilizadas para a criação de mundos virtuais interativos.

Sob o ponto de vista de arquitetura, um jogo de computador é composto por três partes principais: enredo ou trama, motor do jogo ou *game engine* e interface interativa. O enredo define os objetivos do jogo e sua definição pode envolver a participação de diferentes especialistas, como escritores, psicólogos, historiadores, etc. O motor de jogo controla a interação entre o usuário e a interface. Esta, por sua vez, exibe o estado corrente do jogo e permite ao usuário (jogador) interagir com o sistema. A criação da interface pode envolver aspectos artísticos, cognitivos, técnicos e perceptivos.

Segundo Battaiola et al. (2002), as características técnicas destes produtos são ainda mal conhecidas, porém, recentemente, muitos estudos - de usabilidade, por exemplo - estão sendo dirigidos para jogos; no campo de dispositivos móveis – celulares e *palm pods* – o próprio teclado está sendo adaptado para permitir maior desempenho por parte do jogador (FERNANDEZ, 2005). Cabe notar, entretanto, que muitos dos jogos existentes hoje foram construídos a partir de montagens de partes desenvolvidas ao longo dos últimos quinze anos e que, na maioria das vezes, não seguiram nenhuma metodologia de construção em especial.

Conforme diz Valente (2005, p. 1): "A área de desenvolvimento de jogos para computador é conhecida por tentar explorar ao máximo as capacidades da máquina em que o

jogo é executado, para não prejudicar a interatividade. Antigamente, essa característica era praticamente obrigatória em virtude das limitações existentes no equipamento disponível, mais precisamente baixo poder de processamento e pouca memória. Sendo assim, as maiores preocupações dos desenvolvedores eram relacionadas à eficiência de execução de algoritmos (aplicação de truques de programação e do paradigma é bom porque funciona) e com a aparência visual do programa (se é bonito então está certo), em detrimento a fatores como reusabilidade de código e manutenibilidade. Era comum também que camadas intermediárias entre a aplicação e o hardware fossem eliminadas, de forma a acessar o hardware diretamente, para obter a melhor eficiência possível. Dessa forma, tornou-se comum a prática de se implementar todas as funcionalidades necessárias para a criação de um jogo, a cada projeto, mesmo com o grande crescimento da capacidade das máquinas".

Somente nos últimos anos é que passou a existir a preocupação com metodologias e técnicas específicas de construção e se apresentou a necessidade de pensar o reuso de software.

Neste cenário, destaca-se a evolução dos motores de jogos utilizados para a criação dos mundos virtuais. Os motores de jogos de hoje, não apenas cuidam da parte de *rendering* do mundo virtual, mas também do som, da física, da animação e da interação com o usuário, e podem utilizar desde recursos gráficos até Inteligência Artificial (I.A.). Esses motores de jogos são concebidos para serem utilizados por diversas aplicações diferentes, tornando-se praticamente um *framework* de desenvolvimento. Motores gráficos genéricos, utilizáveis em diferentes projetos de jogos, começaram a aparecer no mercado, podendo ser adquiridos pela equipe de desenvolvimento e inseridos no jogo em desenvolvimento ("plugados"). Essa filosofia de desenvolvimento permite que a equipe de desenvolvimento se dedique mais aos aspectos criativos do jogo do que aos aspectos envolvidos com a arquitetura ou com a plataforma de desenvolvimento final onde ele será executado.

1.1. Motivação

Atualmente, tanto a parte de hardware das máquinas quanto as técnicas 3D aplicadas a jogos, evoluem rapidamente, tal que a cada seis meses costuma ser lançado novo hardware e incorporadas novas tecnologias/métodos ao software.

Como o processo de desenvolvimento de um jogo eletrônico pode levar de dezoito meses até cinco anos para ser concluído (FRISTOM, 2003), é bem provável que sua parte de *rendering*, por exemplo, sofra mudanças nesse período, tanto pela incorporação de nova tecnologia como pela necessidade de compatibilizar-se com o novo hardware. Necessita-se, assim, que o projeto de jogos leve em conta tais possibilidades e incorpore características que facilitem a atualização e, consequentemente, a manutenção do sistema (jogo). Na verdade, os jogos se comportam como a maior parte dos sistemas de software: eles evoluem ao longo do tempo após o seu lançamento no mercado consumidor.

Além disso, empresas de desenvolvimento de jogos, em razão dos “prazos apertados”, procuram soluções que se adaptem ao desenvolvimento de seu motor de jogo, de forma a poder implementá-lo em tempo reduzido, utilizando tecnologia de ponta. O desenvolvimento de software em tempo menor permite também reduzir custos, propiciando maior lucro às empresas.

Cabe assinalar que, além do mercado de entretenimento, a tecnologia de jogos encontra grande potencial no ambiente educacional (LEWIS, JACOBSON, 2002). Nos tempos atuais, em que se discute a descoberta de novos caminhos para a Educação a Distância, os jogos oferecem uma perspectiva atraente para a construção de conhecimentos, propiciando a motivação do aluno no conteúdo a ser aprendido.

Examinando esses dois grandes mercados consumidores, entretenimento e educação, constata-se a importância de empregar métodos de Engenharia de Software que permitam

facilitar a criação e a manutenção dos jogos computacionais, uma vez que a demanda deverá aumentar consideravelmente no futuro.

1.2. Objetivos

O objetivo desta dissertação está focado nos motores gráficos. O estudo desses sistemas do ponto de vista teórico – arquitetura e componentes - e o exame de motores gráficos existentes no mercado, destinados a constituírem o núcleo de aplicações gráficas, como jogos ou simuladores, visam a aquisição de conhecimento sobre vários aspectos desses subsistemas. São assim, objetivos desse trabalho:

- Estudo dos jogos eletrônicos e de seus subsistemas, tendo como foco principal a arquitetura e os subsistemas dos motores de jogos.

- O exame de aspectos de reutilização de software em motores gráficos, notadamente o emprego de padrões de projeto (*design patterns*). Este estudo buscou identificar os padrões de projeto da literatura mais utilizados em motores de jogos, bem como identificar padrões específicos para jogos computacionais. Essa identificação permitirá que as disciplinas ligadas à Engenharia de Software, ministradas em cursos de Desenvolvimento de Jogos Eletrônicos explorem esses padrões específicos com os alunos.

- Verificação de características arquitetônicas consideradas relevantes em motores de jogos eletrônicos que permitam o estabelecimento de uma arquitetura modular e flexível, que permita a fácil expansão e manutenção do jogo.

- Apresentação de uma estrutura – baseada em padrões de projeto – que permite o desacoplamento do núcleo do motor gráfico das bibliotecas de baixo nível, visando os aspectos de modularidade e extensibilidade. Este padrão de projeto, denominado *Graphical API Layer Decoupling Pattern*, permite abstrair a biblioteca gráfica a ser utilizada no motor. O estudo foi efetuado para o sistema gráfico mas, eventualmente, pode ser estendido para outros subsistemas. Esta filosofia permite que os diversos subsistemas de um motor gráfico

evoluam separadamente no futuro, sem impactar o produto final (jogo), facilitando a manutenção.

– Recomendação de uma arquitetura específica para os motores gráficos, que auxilie na resolução dos problemas de projeto, relativos à multiplataforma de operação.

Como a estrutura de um motor de jogo completo é relativamente complexa, toda a pesquisa sobre o uso de padrões de projeto será efetuada apenas numa pequena parte do motor de jogo, isto é, na abstração das APIs gráficas utilizadas para o desenho dos objetos do jogo. Esta simplificação foi feita para adequar o trabalho prático dessa dissertação aos prazos estipulados para o seu desenvolvimento, sem, contudo prejudicar a exemplificação da técnica em um caso real.

1.3. Metodologia

Desde o ingresso no programa de pós-graduação, os estudos do candidato foram direcionados aos motores gráficos, particularmente para a área de *rendering* de ambientes virtuais, tendo como objetivo adquirir conhecimento na criação/modularização do módulo que compõe o motor de jogo, contribuindo para sua evolução, através da pesquisa das técnicas mais atuais, como *pixel shader* e *vertex shader*. Esses conhecimentos visavam, assim, contribuir, futuramente, na concepção de um motor gráfico. Para atingir os objetivos, os seguintes passos foram realizados:

- Revisão de conceitos matemáticos: aplicados para geração de ambientes tridimensionais, visando obter um melhor entendimento de como as API gráficas utilizadas em jogos abstraem esses conceitos.
- Levantamento de estudos desenvolvidos em institutos de pesquisa e universidades, cujo foco fosse direcionado ao desenvolvimento de jogos, em especial a parte de *rendering* ou de arquitetura de sistemas gráficos 3D, utilizados em jogos;

- Pesquisa de motores gráficos existentes, utilizados em jogos encontrados no mercado e cujos princípios são de domínio público: foi realizado um levantamento de diversos motores gráficos de jogos de código livre, sendo que alguns destes são utilizados em jogos comerciais;
- Participação em congressos e encontros técnicos da área, visando à troca de idéias e à publicação de artigo técnico sobre o assunto;
- Pesquisa de/em empresas que atuam em desenvolvimento de *middleware* voltado à parte gráfica: como o objetivo dessa pesquisa é o estudo da modularização de um motor de jogo, dando-se ênfase à parte de *rendering* 3D, foi realizada uma pesquisa sobre empresas que possuíssem esse enfoque de mercado;
- Quanto a parte experimental da pesquisa, foram analisados motores de jogos de código aberto, tendo como enfoque a extração e análise de como é feita a abstração da API gráfica, visando a elaboração de um relatório comparativo das técnicas utilizadas, em confronto com soluções usando *desing patterns*.

1.4. Estrutura do trabalho

O presente trabalho está organizado conforme se descreve a seguir.

Esta introdução visou apresentar ao leitor as motivações que lhe deram origem e os objetivos do trabalho, além de descrever a metodologia de pesquisa utilizada e a organização da dissertação.

O capítulo dois apresenta a descrição do que é um motor de jogo e os módulos de software que o compõem, bem como alguns exemplos de sua utilização em várias áreas de conhecimento.

O capítulo três discorre sobre reuso no desenvolvimento de software e como padrões de projeto e *frameworks* podem auxiliar nessa tarefa.

No capítulo quatro é exposta parte experimental. É o capítulo onde serão apresentadas algumas abordagens utilizadas em motores de jogos , visando a abstração da API gráfica. São também discutidos alguns requisitos considerados fundamentais para isso, tais como modularidade e extensibilidade. Para o estudo experimental, foram selecionados nove motores de jogos de código aberto ou descritos na literatura. Em seguida estudou-se como eles atendem a uma coleção de requisitos considerados fundamentais para uma boa arquitetura de software, bem como examinou-se o uso de padrões de projeto em cada um deles, além de outros aspectos como o seu comportamento a múltiplas plataformas.

No capítulo cinco analisam-se os resultados obtidos no capítulo quatro. Finalmente, com base nos estudos realizados propõe-se uma camada (*Graphical API Layer Decoupling Pattern*) que permite desacoplar o módulo gráfico do restante do motor, abstraindo a sua interface.

O capítulo seis apresenta as considerações finais e os planos para trabalhos futuros.

O trabalho apresenta, ainda, além das referências, um apêndice com o Glossário de termos.

2. Motores de Jogos

Este capítulo tem como finalidade apresentar uma breve evolução histórica dos jogos eletrônicos, com o propósito de mostrar como a complexidade desses sistemas cresceu em um curto intervalo de tempo. Também apresenta a decomposição desses sistemas em um conjunto de subsistemas principais, descrevendo cada um deles, dentre os quais se destaca e conceitua-se o motor de jogo ou *game engine*, objeto de estudo dessa dissertação.

O capítulo se encerra com a apresentação de aplicações de motores de jogo.

Neste capítulo partiu-se de alguns pressupostos: primeiramente, foi admitido que o leitor estaria familiarizado com a linguagem dessa área, conhecendo os principais termos utilizados por ela e, em segundo lugar, que também conheceria os principais conceitos da computação gráfica interativa e pelo menos uma das bibliotecas gráficas existentes. Para aqueles leitores que não possuem tais conhecimentos, pode-se indicar Bianchni (2005), Valente (2005) e Azevedo e Conci (2003).

2.1. Principais constituintes de um jogo e evolução histórica

O desenvolvimento de jogos computacionais possui diversas partes (EBERLY, 2000):

- Criação de um bom roteiro e jogabilidade (*game play*), o que pode ser caracterizado como *game design*;
- Criação do conteúdo do jogo (*game content*), relativo ao roteiro e à jogabilidade, geralmente efetuada com o auxílio de pacotes de modelagem;
- Interação com o conteúdo do jogo, que, durante a execução, é controlada por Inteligência Artificial, mais conhecida como IA Para Jogos (*Game AI).* Os programadores devem criar ferramentas para integrar o conteúdo e a Inteligência

Artificial, conforme o roteiro e o *game play* definidos e, então, construir o motor de jogo.

O desenvolvimento de jogos computacionais evoluiu muito em termos de complexidade nos últimos dez anos (BLOW, 2004).

Nos anos 90, um típico jogo 2D possuía os módulos apresentados na Figura 2.1 e que são:

- *main/misc.:* módulo principal que controla o início da execução do jogo e o gerenciamento dos demais módulos;
- *streaming file I/O* : módulo responsável por carregar o conteúdo do jogo (imagens, sons, mapas) dos arquivos localizados em disco (disco rígido ou mídia removível);
- *sound*: módulo responsável pela comunicação com o hardware especializado em áudio e incumbido de gerenciar a geração de efeitos sonoros e as músicas;
- *simulation*: módulo responsável por controlar todas as regras do jogo e a colisão entre objetos;
- *fast 2D graphics*: módulo responsável por desenhar os objetos do jogo no monitor de forma otimizada; sendo assim, era geralmente escrito em linguagem Assembly, acessando diretamente o dispositivo de vídeo.

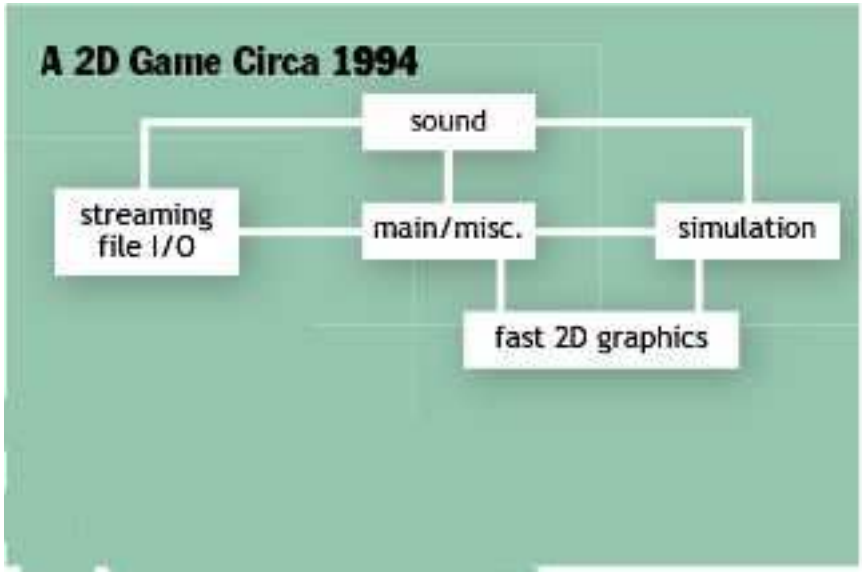

Figura 2.1 - Jogo 2D (BLOW, 2004)

Os motores de jogos que utilizavam a estrutura interna descrita acima eram também conhecidos como *2D Engines*, como, por exemplo, o SuperMetroid[1].

Caso o jogo possuísse uma IA muito complexa, a Figura 2.1 ganharia um módulo de IA, mas perderia, por exemplo, o módulo de *fast 2D graphics*. Todos os jogos que utilizavam essa abordagem eram baseados em turnos (*turn based games)*, como o Civilization[2].

Com a evolução dos jogos e a chegada dos primeiros jogos 3D, como MechWarrior[3], e com o aparecimento das primeiras placas aceleradores 3D de uso doméstico, surgiram o primeiros motores de jogos para geração de jogos em três dimensões, que se denominavam *3D Game Engines*.

A Figura 2.2 mostra a estrutura de um dos primeiros jogos 3D, destacando-se (retângulos rachurados) os novos módulos adicionados.

---

1 http://www.gamespot.com/snes/action/supermetroid/ Acesso: 17 maio 2005
2 http://www.gamespot.com/pc/strategy/civilization/index.html?q=civilization Acesso: 17 maio 2005
3 http://www.gamespot.com/pc/sim/mechwarrior231stcc/index.html?q=mechwarrior Acesso 08 ago 2005

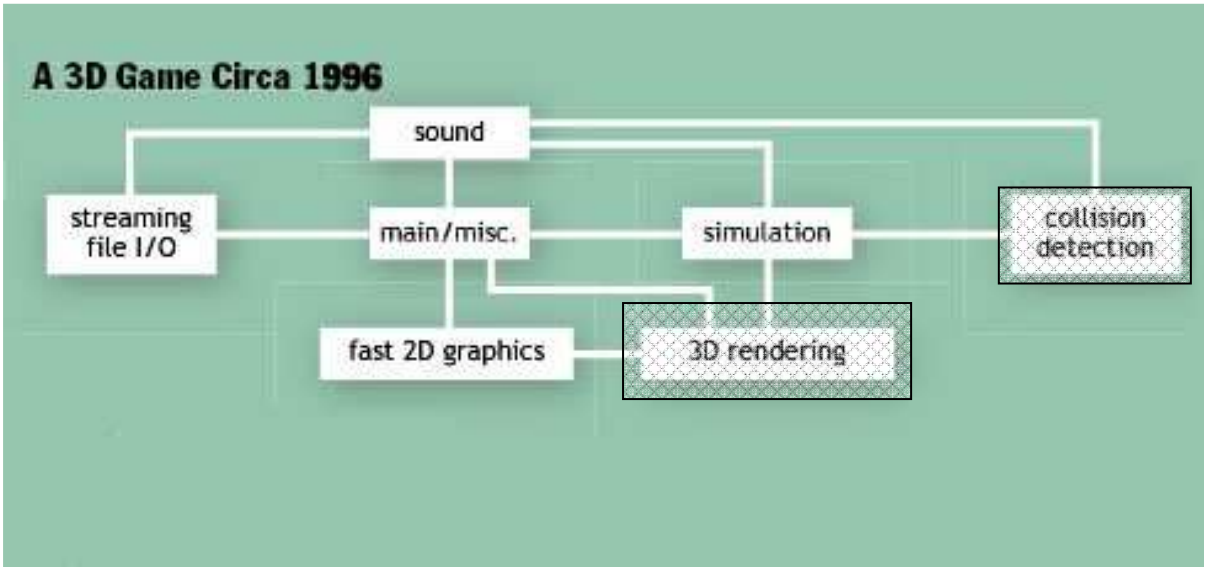

Figura 2.2 - Jogo 3D em 1996 (BLOW, 2004)

Além do módulo *fast 2D graphics*, fora acoplado um módulo *3D Rendering*, responsável pela parte de visualização do mundo em 3D e, caso necessário, permitir a comunicação com a placa aceleradora 3D (na Figura 2.2, embutida em *3D rendering*).

Dada a complexidade na detecção de colisão entre diversos tipos de sólidos, foi criado um módulo de detecção de colisão (*collision detection*).

Atualmente, como demonstra a , a estrutura interna do jogo envolve muito mais módulos, ainda que haja apenas um jogador.

Como se pode observar ocorreram diversas mudanças na estrutura:

- O módulo *fast 2D graphics* foi excluído e o módulo *3D rendering* foi dividido em dois novos módulos, um que gerencia a cena a ser gerada (*rendering: scene management)* e outro que faz a comunicação de baixo nível com o hardware (*rendering: low-level*);
- O módulo de som (*sound*) foi dividido em dois módulos, sendo um módulo para gerenciamento (*sound, management*) e outro para comunicação com hardware (*sound, low-level)*;
- Ao módulo de detecção de colisão foram adicionados algoritmos de Física para uma melhor simulação da colisão entre objetos (*collision detection/physics*).

- Dada complexidade e quantidade dos objetos em cena, foi adicionado um módulo responsável por otimizar o processo de busca por objetos, no caso o módulo *spatial partitioning and search*;
- O módulo de IA tornou-se obrigatório, por ser necessário que os personagens com os quais o jogador interage apresentassem um comportamento mais próximo ao de um ser humano;
- Foi adicionado o módulo de animação 3D (*3D animation*), permitindo uma animação mais realista dos objetos do mundo virtual;
- Em razão da complexidade do jogo, o processo de compilação se tornou custoso, podendo-se levar horas para se compilar o projeto de um jogo (BLOW, 2004). Em virtude disso, parte do código que sofria alterações constantes, como o módulo que controla as regras do jogo, foram recodificados em uma linguagem de mais alto nível, sendo esta interpretada pelo motor do jogo em tempo de execução. O módulo responsável por essa execução é o *scripting evaluator*;
- A produção de conteúdo (artes, mapas, imagens, sons, modelos) também pode ser complexa, exigindo a criação de ferramentas especificas (*tools*) que possibilitassem que os próprios artistas, modeladores e *designers* gerassem conteúdo já no formato do motor de jogo. Algumas dessas ferramentas são distribuídas aos usuários.

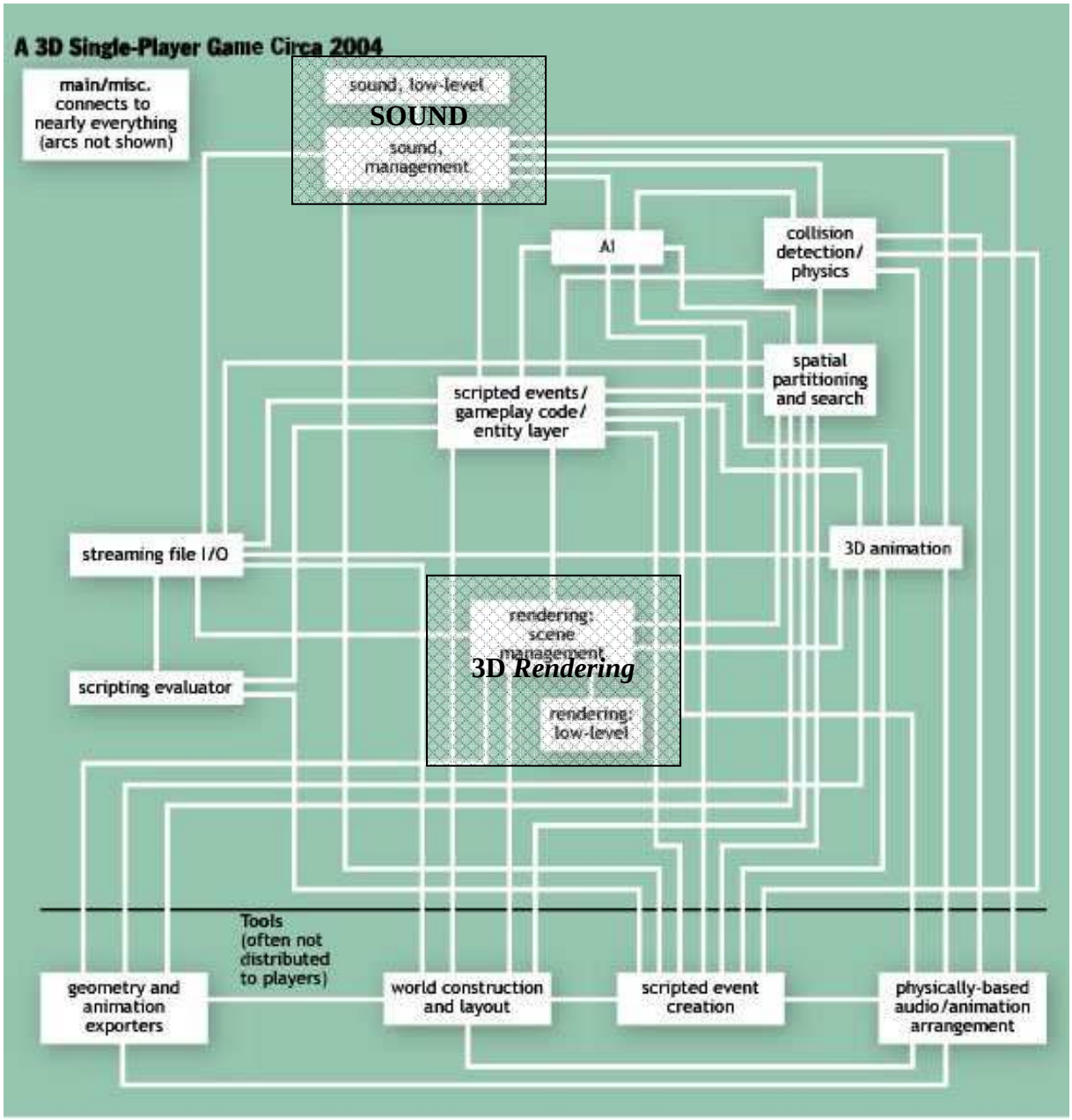

Figura 2.3 - Jogo 3D *Single Player* em 2004 (BLOW, 2004)

A Figura 2.3 ilustra um dos maiores projetos atuais, que são os jogos MMO (*Massive Multiplayer Online),* em que milhares de jogadores interagem entre si num mundo virtual persistente, isto é, onde tanto o personagem do jogador como outras entidades do jogo têm seus estados salvos no servidor entre as partidas, permitindo que o jogador saia do jogo e volte a jogar quando desejar.

As diferenças mais marcantes entre as figuras 2.3 e 2.4, que se pode destacar são a inclusão da parte de comunicação por rede e a divisão do motor de jogo em três camadas:

- Servidor (s*erver)*: responsável por controlar o mundo virtual como suas regras, personagens não controlados pelo jogador (NPCs - *non player characteres)*, gerenciamento de eventos e persistência dos dados do jogador e do mundo virtual, através de algum tipo de base de dados;
- Cliente (*client)*: responsável por enviar as ações do jogador para o servidor, *rendering* da cena, som e animação;
- Módulos compartilhados (*shared*): são módulos utilizados por ambos os lados (cliente/servidor), como detecção de colisão, comunicação de rede de baixo nível, controle e localização de objetos no jogo.

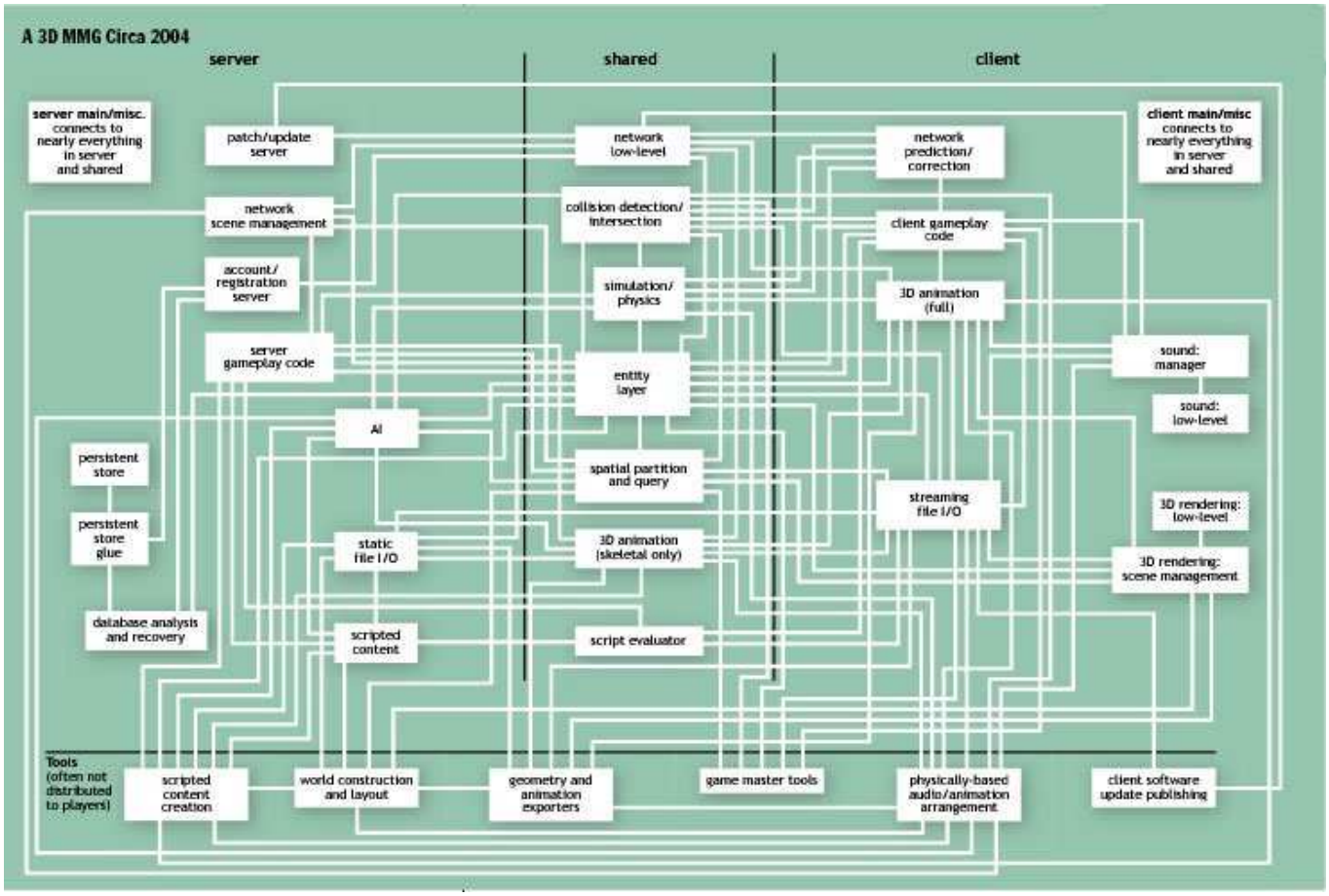

Figura 2.3 – Massive Multiplayer Online atual (BLOW, 2004)

Comparando a Figura 2.1 e a Figura 2.3, pode-se observar o quanto aumentou a complexidade no desenvolvimento de jogos ao longo dos anos.

Existe ainda um outro grupo de jogos chamados *multiplayer* que seriam um intermediário entre e a Figura 2.3. São jogos cliente/servidor de até 64 pessoas, não persistentes, isto é, uma vez que o jogador sai do jogo, seu estado não é armazenado em base de dados. Entre os jogos que utilizam essa abordagem pode-se citar Counter-Strike:Source (VALVE SOFTWARE, 2004c), Battlefield 2 (DIGITAL ILLUSIONS CE, 2005) e Unreal Tournament 2004 (EPIC GAMES, 2004b).

2.2. Organização de um *Motor de Jogo*

Lewis e Jacobson (2002) definem um motor de jogo como “uma coleção de módulos de simulação que não especifica o comportamento do jogo (lógica) ou o ambiente do jogo (mapa)”. O termo “motor de jogo” muitas vezes é associado a um módulo responsável por controlar os demais módulos (LAMOTHE, 2003). No entanto, nesse trabalho utilizar-se-á a definição de Lewis e Jacobson (2002).

A Figura 2.4 mostra os macro-módulos principais que fazem parte de um motor de jogo e que serão descritos nos itens que se seguem.

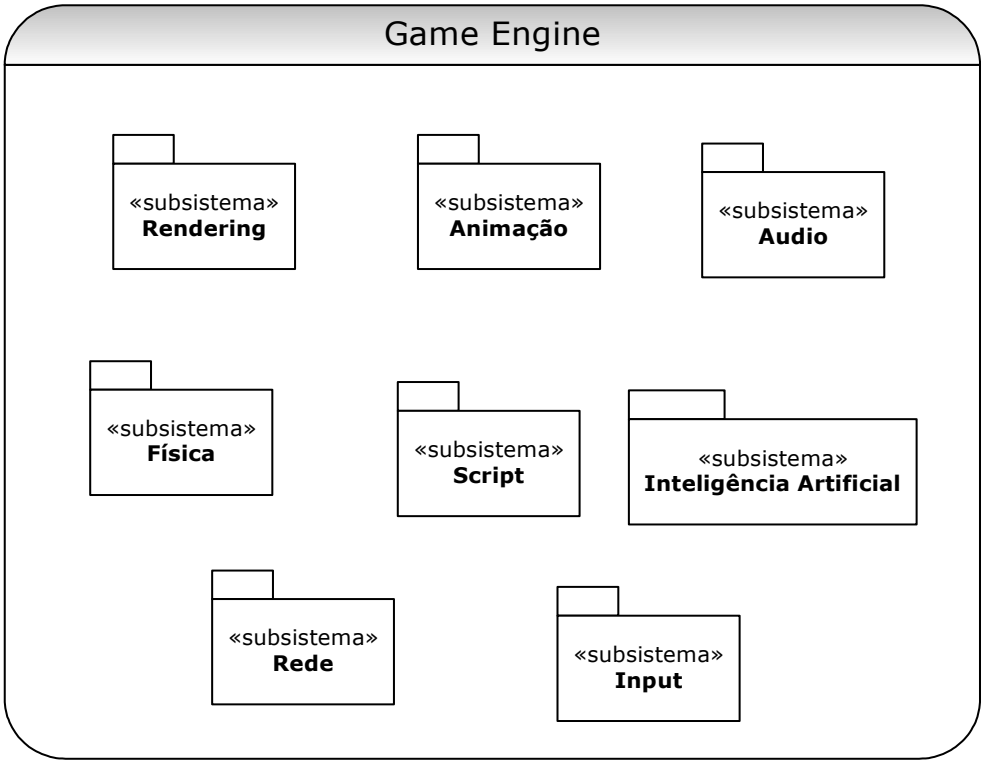

Figura 2.4 - Sistemas que compõem um motor de jogo

2.2.1. Subsistema de *Rendering* (*Rendering Subsystem*)

O subsistema de *rendering* é o módulo de software que "processa as estruturas de dados do mundo 3D, incluindo todas as luzes, ações, informações de estados e faz o *rendering* do ponto de vista do jogador ou câmera" (LAMOTHE, 2003).

Este subsistema executa duas tarefas: gerenciamento de cena (*scene management*) e *rendering*, que serão discutidos nos próximos sub-itens, apresentando-se, ao final, as bibliotecas OpenGL e Direct X e discutindo-se suas diferenças.

2.2.1.1. Gerenciamento de Cena (*Scene Management*)

Segundo Eberly (2004), a tarefa de gerenciamento de cena é responsável por gerenciar uma cena do jogo, como, por exemplo, uma fase específica do jogo, controlando todos os objetos da cena (luzes, malhas, objetos estáticos, etc.) e decidindo o "que" é mostrado ao usuário ou *culling*. O *culling* consiste no processo de filtragem de faces de um objeto que não

precisam ser passadas para o sistema de *render* durante o *rendering* da cena. Assim, ganha-se performance na geração da cena (YOUNG, 2005)

Segundo Valente (2005), os três tipos mais comuns de *culling* são:

- *Backface culling:* Um polígono possui dois lados: o lado da frente e o lado de trás. Normalmente, o observador visualiza apenas um desses lados (o lado da frente, por exemplo). O *backface culling* é usado para descartar os lados que não podem ser vistos. Esse processo está disponível em hardware atualmente.. Esse tipo de *culling* já vem embutido nas APIs de *render* atuais, como OpenGL e DirectX.
- *Frustum culling*: esse método utiliza o *frustum* (volume de visão) da câmera para eliminar objetos inteiros (coleções de polígonos). Para isso, verifica-se se o polígono está dentro do *frustrum* e, em caso contrário, o elimina da lista de objetos que serão enviados ao *render*. A desvantagem desse método é que muitos objetos próximos ao *near plane* (plano mais próximo da câmera) irão para o *render*, mesmo que não estejam visíveis.
- *Occlusion culling*: é a eliminação de objetos que estão obstruídos por outros, como por exemplo, uma mesa atrás de uma parede. Como o jogador não irá ver a mesa, ela pode ser eliminada da lista de objetos a serem enviados para o *render*. A parede em questão é chamada de *occluder* e corresponde ao objeto que se encontra mais próximo ao observador (jogador). Qualquer objeto pode obscurecer outros objetos da cena num dado momento, conforme a cena se desenvolve. A desvantagem desse método de *culling* é o alto custo computacional em cenas com muitos objetos.

Outro método apontado por Valente (2005) para otimização no processo de geração de cena é a utilização de níveis de detalhes (*Level Of Detail* ou LOD). Essa técnica se baseia no fato de que objetos muito distantes não são visíveis em todos os seus detalhes, de forma que sua representação não precisa ter o mesmo grau de detalhamento (medido pelo número de

polígonos) do que a de um objeto que esteja próximo ao jogador. Assim, pode-se adotar diversos modelos de um mesmo objeto, cada um com detalhamento diferente, mostrando a versão com mais detalhes quando o objeto está próximo do jogador, e as versões com menos detalhes, conforme o objeto se distancia.

2.2.1.2. *Render*

O *render* é responsável pela comunicação com o hardware, enviando-lhe os dados a serem desenhados na tela. Atualmente, os desenvolvedores não se preocupam com o desenvolvimento dessa parte de "baixo nível", e utilizam APIs gráficas já existentes no mercado, como a OpenGL (OPENGL ARCHITECTURE REVIEW BOARD, 2005) e DirectX (MICROSOFT, 2004a), que são as mais comuns. O desenvolvimento desse módulo só se daria se a plataforma de desenvolvimento não possuísse uma API existente, como, por exemplo, no caso de aparelhos *handheld* ou de telefonia celular que não tivesse suporte ao OpenGLES(EBERLY, 2004).

A Figura 2.5 mostra a *pipeline* de um processo de *redering* feito por software. Deve-se lembrar que, da parte relativa ao processo de *culling*, somente o *Backface removal* e o *frustum culling* são feitos pelas APIs OpenGL e DirectX.

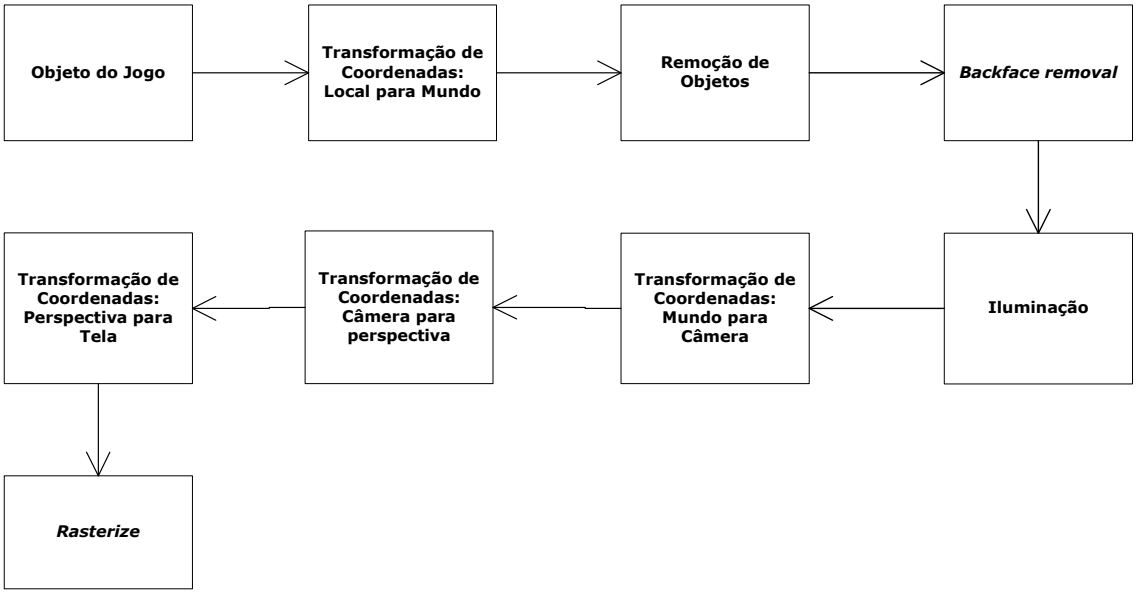

Figura 2.5 - Software *Rendering Pipeline*

Como se pode observar nessa *pipeline*, os objetos de uma cena têm suas coordenadas locais transformadas em coordenadas do mundo através dos algoritmos de transformações geométricas. Em seguida, usando-se técnicas de *culling*, os objetos que não estão no campo de visão são removidos da lista de objetos a serem processados. O *Backface removal* elimina as faces dos objetos que não estão visíveis ao usuário, para que sejam aplicadas a(s) fonte(s) de luz(es) sobre os objetos da cena, que têm, então, suas coordenadas convertidas para as coordenadas da câmera em questão. Após os ajustes de perspectiva, a imagem final é gerada (*Rasterize*).

2.2.1.3. *Vertex* e *Pixel Shaders*

As novas placas de vídeo (GPUs - *Graphics Processing Units*) disponibilizam para o desenvolvedor uma forma de criar pequenos programas para serem processados na própria GPU, que são denominados *vertex shaders* e *pixel shaders* (EBERLY, 2004).

Os *Vertex shaders* permitem controlar o que é desenhado na tela com base nos atributos de vértices como: suas posições, suas normais e cores (*shading*). Esse *shader* é executado para cada vértice enviado à *pipeline* de *rendering*.

Já os *Pixel shaders* ou *fragment shaders* permitem o controle do que é desenhado na tela, tendo como base os atributos da imagem. Um fragmento (*fragment*) *shader* é um ponto nas coordenadas da janela, associado com diversos atributos, como valores de cor interpolados, valores de profundidade e um ou mais conjuntos de coordenadas de textura. O fragmento modifica o *pixel* no *frame buffer* de mesma localização do espaço de janela, conforme um número de parâmetros e condições definidas pelos estágios do *pipeline*, após o bloco *Rasterize*. Algumas vezes, a noção do termo fragmento é confundida com a noção do termo *pixel*, mas segundo Ecker (2003), "um *pixel* é apenas a cor final escrita no *frame buffer*, e cada *pixel* no *frame buffer*, geralmente, corresponde a diversos fragmentos. Alguns desses fragmentos são descartados por causa de testes de profundidade e outros são combinados a fim de formar a cor do *pixel* final".

Os programas para *Vertex Shaders* e *Pixel Shaders* eram, inicialmente, escritos em linguagens de baixo nível, parecidas com *Assembly*, e necessitavam de um compilador separado. No entanto, atualmente, APIs como OpenGL e DirectX provêm suas respectivas linguagens para *shaders*, de alto nível, parecidas com C. Porém um programa *shader* escrito para OpenGL não é compatível com DirectX.

A linguagem Sh (MCCOOL, 2005) é uma linguagem de metaprogramação para GPUs programáveis. A proposta da Sh é criar uma linguagem de alto nível baseada em Orientação a Objetos que abstraia a criação de *shaders*, isto é, a criação de *shaders* fica independe da API que está sendo utilizada.

A versão atual da especificação de *shaders* é a 3.0; a diferença entre cada versão é a quantidade de registros e comandos suportados pela GPU.

2.2.1.4. OpenGL (*Open Graphics Library*)

Introduzida em 1992, a OpenGL é uma API para o desenvolvimento de aplicações 2D e 3D, cuja característica principal é o grande número de plataformas e sistemas operacionais que a suportam. Originalmente criada na SGI por Kurt Akely, atualmente é mantida pelo ARB (*Architectural Review Board*), que é composto por diversas empresas, como Dell, NVidia e ATI. Sua última versão é a 2.0 (OPENGL ARCHITECTURE REVIEW BOARD, 2005), que introduziu a OpenGL *Shader Language* para a criação de *Vertex* e *Pixel* S*haders*.

Segundo Ecker (2003), a entrada da OpenGL no mercado consumidor decorreu do lançamento do jogo Quake, que utilizava OpenGL como API para *rendering*, fazendo com que os fabricantes de placas de vídeo criassem *drivers* que a suportassem.

A Figura 2.7 mostra a *pipeline* de render do OpenGL.

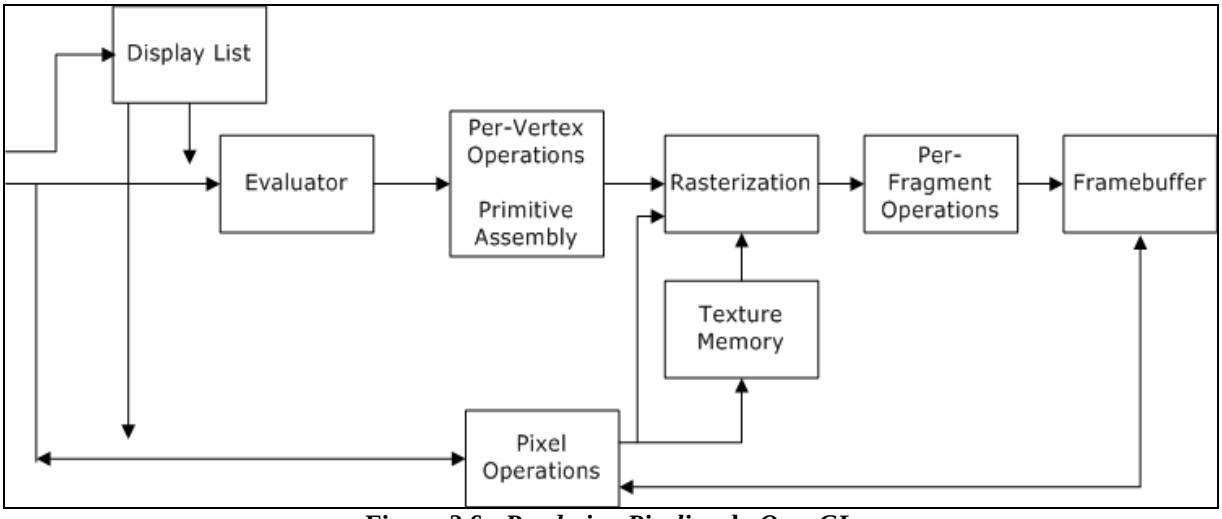

Figura 2.6 - *Rendering Pipeline* do *OpenGL*

A biblioteca OpenGL também possui um sistema de extensão de procedimentos que permite que fabricantes disponibilizem outras funcionalidades não definidas pelo padrão. Como essas extensões são relativas ao fabricante, isto é, não integram o padrão, os vários

fabricantes não precisam suportar tal funcionalidade, ainda que possam ser futuramente adicionadas à especificação básica do OpenGL (ECKER, 2003).

2.2.1.5. DirectX

É um pacote de APIs criado pela Microsoft para o desenvolvimento de jogos e aplicações multimídia, onde a API que trabalha com gráficos 2D e 3D é conhecida como DirectX Graphics (versão DirectX 8.0+), antigo Direct3D (versão DirectX 7.0-). Essas APIs apenas estão disponíveis para os sistemas operacionais da família Windows (MICROSOFT, 2004a), e se baseiam no *Microsoft Component Object Model* (COM) (MICROSOFT, 2005c), de forma que a biblioteca DirectX Graphics pode ser utilizada por qualquer linguagem que suporte COM, como o Visual Basic, por exemplo.

Este pacote, mantido pela Microsoft, sofre revisões em períodos de um ou dois anos (ECKER, 2003), que podem ser mínimas, para contemplar novas especificações de hardware, ou podem modificar totalmente a API, como ocorreu da versão DirectX 7.0 para a versão DirectX 8.0. A versão DirectX 7.0 possuía o DirectDraw para operações 2D e o Direct3D para operações 3D; já na versão DirectX 8.0, os dois pacotes passaram a integrar o módulo DirectX Graphics. A última versão do DirectX, até a redação desse documento, é a 9.0c, espera-se o lançamento da DirectX 10 em 2007 junto com o sistema operacional Windows Vista, sendo que esta nova versão será uma modificação total da API.

AFigura 2.7 mostra a *pipeline* de *rendering* do DirectX.

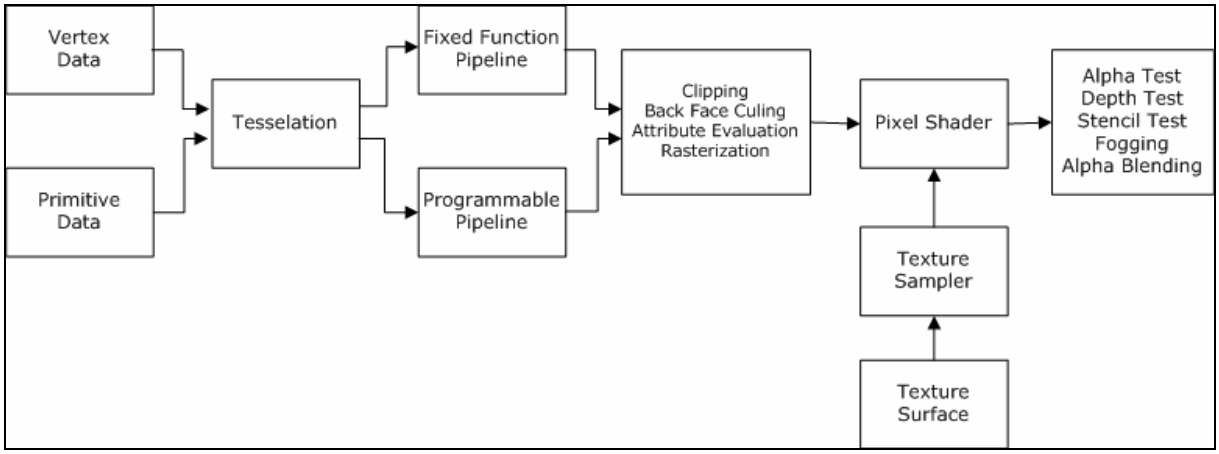

Figura 2.7 - *Rendering Pipeline* da biblioteca DirectX

Entre as alterações verificadas nessa versão pode-se citar a inclusão de linguagem para criação de *Vertex e Pixel Shaders*, a HLSL (*High-Level Shader Language*).

2.2.1.6. Diferenças entre OpenGL *e* DirectX

Ecker (2003) aponta diversas diferenças entre OpenGL e DirectX, entre as quais se destacam:

- Integração de novas funcionalidades: enquanto OpenGL possui um mecanismo de extensão embutido, DirectX, algumas vezes, sofre alterações maciças na sua especificação, o que obriga o desenvolvedor a reaprender a utilização de certos blocos da API;
- Sistema de coordenadas: Direct X usa a regra da mão esquerda, enquanto OpenGL utiliza a regra da mão direita;
- Convenção de Matrizes e Vetores: DirectX considera vetores e matrizes linearizados por linha (*row major*) e a multiplicação é feita entre um vetor e uma matriz (Figura 2.8), enquanto no OpenGL vetores e linhas são linearizados por coluna (*column major*) e a multiplicação é feita entre uma matriz e um por vetor (Figura 2.9);

$$v \cdot M = \begin{bmatrix} x & y & z \end{bmatrix} \cdot \begin{bmatrix} M_{0,0} & M_{0,1} & M_{0,2} \\ M_{1,0} & M_{1,1} & M_{1,2} \\ M_{2,0} & M_{2,1} & M_{2,2} \end{bmatrix}$$

Figura 2.8 - Matrizes e vetores DirextX (ECKER ,2003)

$$M \cdot v = \begin{bmatrix} M_{0,0} & M_{0,1} & M_{0,2} \\ M_{1,0} & M_{1,1} & M_{1,2} \\ M_{2,0} & M_{2,1} & M_{2,2} \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Figura 2.9 - Matrizes e vetores OpenGL (ECKER, 2003)

- *Clipping Volume*: as coordenadas do volume de corte (*clipping volume*) são diferentes. Seja [x y z w] um vértice. expresso em coordenadas homogêneas, transformado na coordenada de corte. O volume de corte no OpenGL é definido como:

$$-w < x < w$$
$$-w < y < w$$
$$-w < z < w$$

Na DirectX, o mesmo volume é definido como:

$$-w < x < w$$
$$-w < y < w$$
$$0 < z < w$$

Sistema de coordenadas de textura: no DirectX, a origem do sistema de coordenadas é no canto superior esquerdo do mapa de textura (Figura 2.10), enquanto no OpenGL é o canto inferior esquerdo, especificando-se as texturas invertidas no eixo vertical (Figura 2.11).

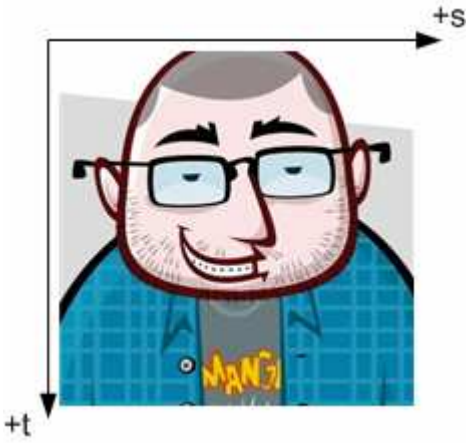

Figura 2.10 - Sistema de coordenadas de Textura na DirectX

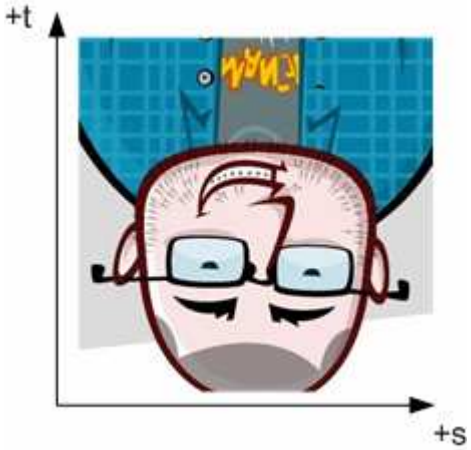

Figura 2.11 – Sistema de Coordenadas de Textura na OpenGL

2.2.2. Subsistema de Animação (*Animation subsystem*)

O subsistema de animação é responsável pelo controle de todo tipo de animação que ocorre durante a execução de um jogo. LaMothe (2003) aponta que num jogo podem ocorrer diversos tipos de animações: movimento simples, animações complexas ou animações baseadas em Física.

Os movimentos simples são animações que consistem em translações e rotações dos objetos do jogo. Esse tipo de movimentação pode ser controlado por arquivos de dados, IA, lógica simples, máquinas de estados etc.

Já as animações consideradas complexas são aquelas que têm uma hierarquia de objetos articulados, com ligações que permitem que um objeto se mova em relação ao outro, como, por exemplo, no caso de um tanque que possui um canhão articulado. A ligação está entre o tanque e o canhão, que pode sofrer rotações em torno do eixo superior do corpo do tanque. Esse tipo de movimento pode ser obtido de várias formas, sendo duas delas as mais utilizadas: a animação por quadros-chaves (*key-frames)* e a animação baseada em esqueletos (*skeletal animation*).

Na animação por quadros-chave, somente os quadros-chave da animação são armazenados e os quadros intermediários são gerados através de uma função de interpolação (FOLEY et al., 1997). Os jogos Quake I (ID SOFTWARE, 2004a) e Quake II (ID SOFTWARE, 2004b) utilizam esse tipo de animação.

No segundo tipo, a animação baseada em esqueletos (VALENTE, 2005), ao invés de se armazenar quadros de animação, armazena-se informação de como uma junta se comporta em relação à outra, sendo as animações geradas durante a execução do jogo. A maior desvantagem dessa abordagem é o alto custo de CPU envolvido nos cálculos para se gerar a animação. Em contra-partida, gera-se uma animação mais realística, dado que as articulações

se adaptam ao contexto do jogo, como, por exemplo, no caso de um objeto que se move em terreno irregular. Os jogos mais atuais como Doom III (ID SOFTWARE, 2004d) e Half-Life 2 (VALVE SOFTWARE, 2004d) utilizam esse tipo de animação.

As animações baseadas em modelos de Física utilizam modelos reais de Física para controlar o processo de animação. Um tipo de animação muito utilizado nos jogos é o *Ragdoll Physics* (WIKIPEDIA, 2005a), Na qual se substituem as animações estáticas de "morte de personagem, por uma movimentação gerada em tempo-real, respeitando as leis de Física.

Um *ragdoll* é uma coleção de múltiplos corpos rígidos, cada um relacionado a um *bone*, usando um subsistema de animações por esqueletos, relacionados por um sistema de regras, que restringem como cada *bone* pode se mover em relação a outro.

A troca do processo de animação estática pelo *ragdoll* é feita para obtenção de uma seqüência de animação mais realística em relação ao ambiente em que se encontra o personagem do jogo.

2.2.3. Subsistema de Física (*Physics subsystem*)

O subsistema de Física de um jogo é um sistema complexo de detecção de colisão, responsável por controlar e responder às ações que os objetos realizam, ou deixam de realizar, e, então, baseado nos modelos físicos implementados, tomam alguma ação (LAMOTHE, 2003).

O uso de um modelo baseado em Física visa produzir animações mais próximas ao mundo real, mas precisão não é um ponto chave nessa simulação em jogos de uso interativo: a meta principal é levar o usuário a acreditar que se encontra em um mundo real (JACOBSEN, 2003). O programador pode "trapacear o quanto quiser" se o jogador se sentir imerso no ambiente.

Outro ponto chave a considerar é a velocidade de execução: o subsistema de Física só poderá utilizar a CPU durante um certo período de tempo, isto é, a cada quadro do jogo.

O subsistema de Física foi um dos pontos de desenvolvimento de jogos, onde mais se fez investimentos nos últimos anos, dado o surgimento de diversos *middleware* de Física como o da Havok (2005), o Megon Physics SDK (MEQON, 2005) e o RenderWare Physics (RENDERWARE, 2005. Em fevereiro de 2006 foi lançado o primeiro *chip* especializado em cálculos de Física, chamado de PPU (*Physics Processing Unit*), e se espera que a simulação de Física em jogos dê um salto semelhante ao que deu a parte gráfica nos anos 90, com o surgimento das GPUs (AGEIA, 2006).

2.2.4. Inteligência Artificial (*Artifical Inteligence Subsystem*)

Segundo Valente (2005), "a função da inteligência artificial em jogos é produzir 'comportamento' convincente ao usuário para os objetos controlados pela aplicação, de modo que torne o jogo interessante".

O campo da Inteligência Artificial sempre esteve presente no desenvolvimento de jogos eletrônicos, mas foi só há alguns anos que começou a ganhar destaque.

"Historicamente, a IA de jogos sempre foi uma das 'últimas damas a entrar na dança' do processo de desenvolvimento" (WOODCOCK, 1998). Isso era um fato constante nos jogos, já que a CPU era responsável pela parte de *render*, som, IA e controle do jogo, que eram bem mais limitadas do que são hoje.

"Um efeito colateral interessante do avanço rápido do mercado de hardware 3D com o aparecimento das placas especializadas, é que a CPU tem agora mais tempo livre para mais tarefas relacionadas à IA" (WOODCOCK, 1998). Atualmente, grande parte dos jogos já faz uso da placa de aceleração gráfica 3D, o que libera a CPU para outras tarefas como, por

exemplo, o processamento da parte de IA, possibilitando, assim, tomadas de decisões e comportamentos mais realísticos nos jogos.

"A implementação de IA deve ser realista o bastante para representar algum desafio ao jogador, mas nunca dar a impressão de o jogo está trapaceando" (VALENTE, 2005). Isso se dá pelo fato do computador conhecer todos os "dados do jogador" e, então, se deve modelar o comportamento da IA do jogo de forma a simular uma partida justa, limitando, por exemplo, certas informações para IA.

Bianchini (2005) enumera diversas técnicas de IA utilizadas em jogos, destacando entre elas:

- Algoritmo A* (A estrela): algoritmo popular para solução do problema do caminho (*pathfinding*) que um personagem deve seguir;
- Sistemas de produção: são sistemas que utilizam uma série de implicações e manipulações lógicas para determinar qual ação deve ser executada em determinado instante;
- Máquinas de Estado: são basicamente um conjunto finito de estados, no qual o sistema (personagem) pode se encontrar e um conjunto finito de Regras de Transição, que dita para qual estado o sistema deve evoluir, conforme a informação recebida do motor e o estado em que se encontra no momento;
- Lógica *Fuzzy*: é um tipo de lógica baseada na Teoria dos Conjuntos *Fuzzy*, que é uma maneira de se especificar o quão bem um objeto satisfaz a uma descrição vaga, ou, equivalentemente, qual o grau de pertinência com que um dado elemento pertence a um determinado conjunto. Em jogos, esse tipo de lógica permite que decisões baseadas em valores "não completamente verdadeiros ou falsos" possam ser feitas, tornando a abstração de conceitos do mundo real menos restritivas e aumentando,

assim, o número de opções que um personagem tem para decidir sobre determinado assunto;

- Redes Neurais Artificiais: método que procura derivar soluções com base no que se conhece sobre o funcionamento do cérebro humano. Em uma rede neural, existe um conjunto de neurônios que se conectam, sendo que os resultados obtidos em uma camada de neurônios são utilizados como entrada para a próxima camada;
- Algoritmos Genéticos e Vida Artificial: algoritmos genéticos são um meio de solucionar problemas, imitando o processo de evolução utilizado pela “mãe natureza”. Esse processo se dá pela criação inicial de uma população, cujos indivíduos possuem seus atributos representados por uma cadeia binária (DNA). Esses indivíduos passam por um processo de avaliação, em que os piores são descartados, enquanto é feito um cruzamento entre as cadeias binárias dos melhores, gerando assim novo individuo, que se espera ser melhor que seus pais. Tal processo se repete até que se obtém uma solução que se considera aceitável. Na área de jogos, os algoritmos genéticos são muito usados para o aperfeiçoamento da IA de um personagem de jogo, implementada, por exemplo, por uma máquina de estados. Como existem diversos testes e formas como um personagem pode interagir com o jogador ou o ambiente, os algoritmos genéticos possibilitam analisar um conjunto maior de possibilidades de forma automatizada. Assim, ao final de um processo, pode-se verificar qual a melhor configuração encontrada e associá-la ao personagem do jogo.

No Game Developers Conference 2005 (Kirby, 2005), os tópicos discutidos sobre IA incluíram aprendizagem, como utilizar IA no processo de verificação da qualidade do jogo (QA - Quality Assurance) e como os próximos processadores dual-core (dois núcleos em um único processador) podem auxiliar numa simulação de IA mais realística, dado que, teoricamente, um núcleo poderia ser reservado apenas para cálculos da IA.

2.2.5. Subsistema de Áudio (Sound Subsystem)

O subsistema de áudio é responsável por controlar e reproduzir sons e músicas do jogo. Atualmente, é comum se falar em som tridimensional como o que é “utilizado como um estímulo sutil para melhorar a sensação de imersão do usuário na simulação” (VALENTE, 2005). O subsistema de áudio tridimensional é baseado em um sistema composto por ouvintes (*listeners*) que recebem o som (jogadores) e fontes sonoras (sources) que geram o som no jogo. Ambos os objetos possuem propriedades que variam desde posição, velocidade e orientação até propriedades que indicam como gerar o som com *roll of factor* (taxa de atenuação) ou *doppler factor* (taxa de efeito *Doppler*).

Com relação à comunicação com o hardware de som, as duas APIs mais utilizadas são a OpenAL (LOKI SOFTWARE, 2005) e o DirectSound (MICROSOFT, 2004b). Apesar de algumas diferenças internas e de utilização, ambas APIs provêm as mesmas funcionalidades, sendo que o OpenAL é multiplataforma e o DirectSound faz parte do pacote DirectX da *Microsoft Corporation*, ou seja, só é processado em plataforma que possua sistema operacional Windows.

Com relação às músicas, segundo Valente (2005), nos jogos atuais é aplicado o conceito de música dinâmica (*dynamic music*), que é a composição em tempo-real de música em um jogo, em resposta às ações do jogador ou eventos do mundo virtual. Esse tipo de técnica permite uma melhor imersão do jogador no jogo, mudando o ritmo da música conforme a situação em que o jogador se encontra, como música lenta em situações de calmaria ou rápida e forte em situações de combate. Vale lembrar que o programador de áudio e o compositor das músicas devem trabalhar juntos para que esses blocos de música possam ser encaixados de forma coerente.

2.2.6. Comunicação de rede (*Network Subsystem*)

"O módulo de comunicação de rede é utilizado para conectar vários computadores, de modo que vários jogadores possam participar da ação ao mesmo tempo. É responsável por informar o estado do jogo a todos os computadores, para que todos os jogadores possam estar sincronizados. Jogos em rede, tipicamente, utilizam redes locais *LAN* (*Local Area Network*) ou a Internet" (VALENTE, 2005).

La Mothe (2003) aponta dois tipos de arquitetura para jogos em rede: computador a computador (*peer to peer*) e cliente/servidor (*client/server*).

Na arquitetura computador a computador, cada participante possui uma conexão com os demais, conforme é mostrado na Figura 2.12.

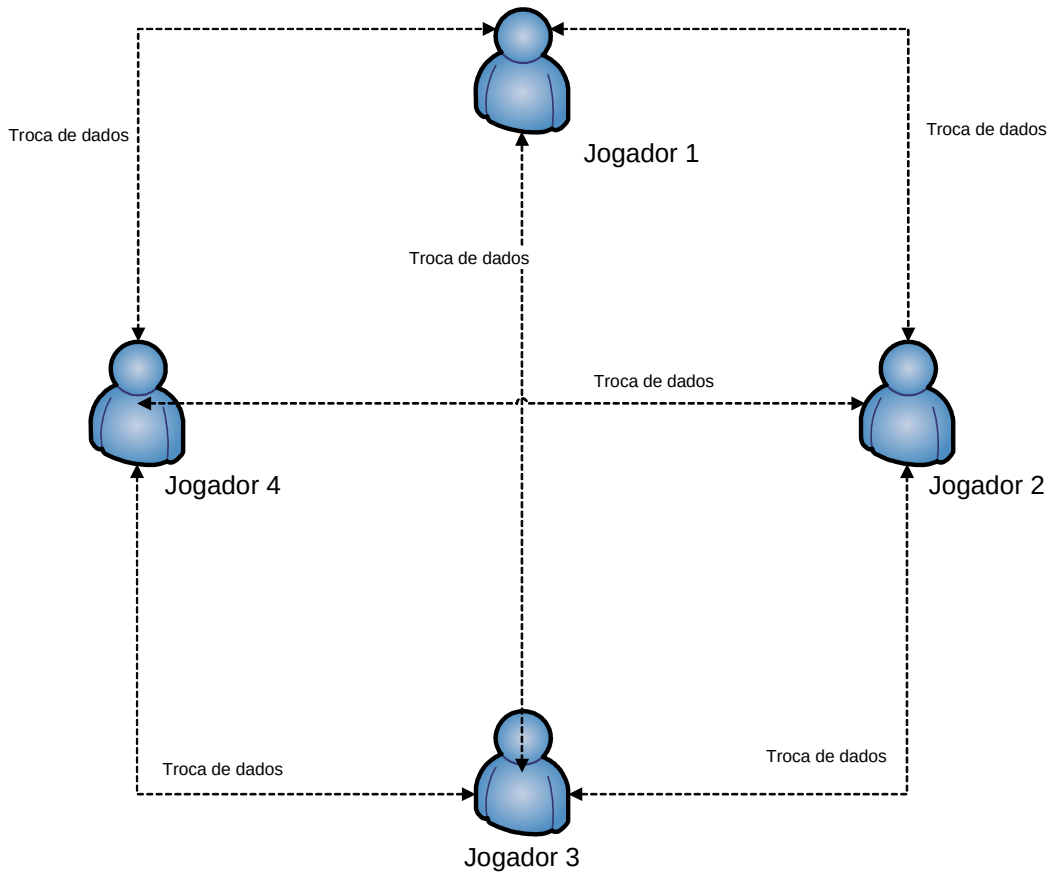

Figura 2.12 – Arquitetura computador a computador (*peer to peer*)

Essa arquitetura, possui como desvantagem pouca escalabilidade (aumento no número de participantes), sendo recomendada para jogos de até oito jogadores.

Na arquitetura cliente/servidor, um computador, geralmente com maior poder de processamento e largura de banda de comunicação, funciona como um servidor, controlando as ações dos jogadores (clientes). Esta arquitetura é apresentada na Figura 2.13.

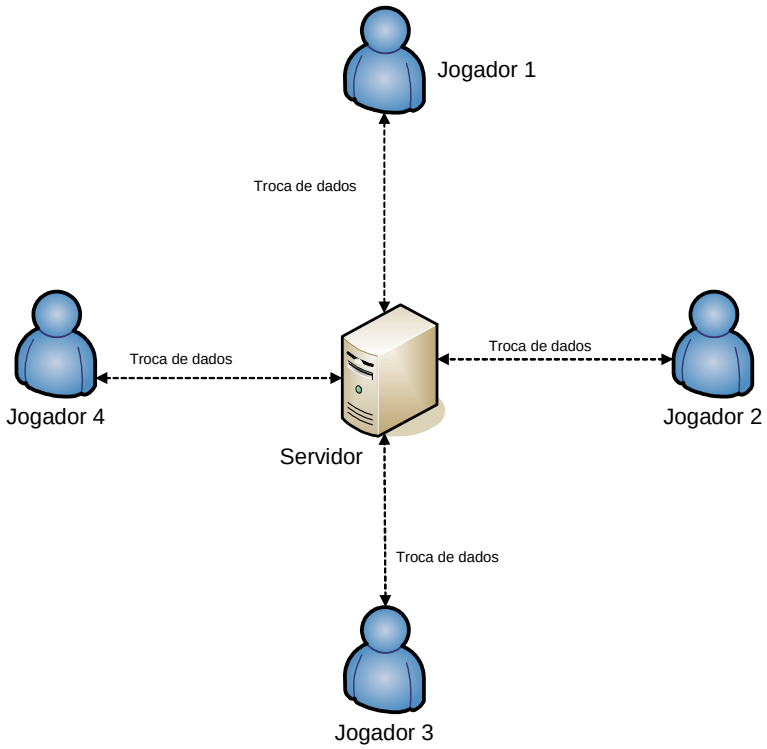

Figura 2.13 – Arquitetura cliente/servidor (*client*/*server*)

Esse tipo de arquitetura é mais complexo de ser implementado, mas é mais interessante, dado permitir uma melhor escalabilidade, redução do número de conexões e da largura de banda dos jogadores, já que cada um deles só se comunica com o servidor. Esse tipo de arquitetura é o mais utilizado nos jogos de hoje e é a arquitetura utilizada em jogos MMO (*Massive Multiplayer Online*).

2.2.7. Subsistema de Entrada de Dados (*Input Subsystem*)

O subsistema de entrada de dados gerencia como o jogador interage com o jogo. Atualmente, existem diversos dispositivos de interação entre o jogador e o jogo, e estes variam conforme a plataforma do jogo em questão.

Em computadores, os dispositivos mais encontrados são o teclado e o mouse; outros dispositivos de entrada muito conhecidos são os *joysticks*, que estão presentes tanto em computadores como em *videogames*, e possuem diversas configurações e formatos (VALENTE, 2005). Ainda entre os dispositivos para jogos, citam-se os manches para simuladores de avião e os tapetes de dança, usados no jogo *Dance Dance Revolution*.

Alguns desses dispositivos possuem um sistema chamado de *Force Feedback* que são atuadores e motores que exercem algum tipo de força sobre a mão ou corpo (LAMOTHE, 1999). Exemplos de dispositivos com esse sistema, seriam o *joystick Microsoft Force Feedback* e produtos como o mouse da *Immersion Corporation*.

Na linha de dispositivos mais recentes, pode-se destacar o BODYPAD[4] da empresa francesa XKPad e o *EyeToy*[5] da Sony. O BODYPAD é formado por uma série de sensores ligados às articulações do jogador e que capturam os seus movimentos e os transmitem para o jogo, geralmente, de luta.

O *EyeToy* é uma minicâmera ligada ao *videogame* Playstation 2 da própria *Sony*, que capta os movimentos feitos pelo jogador e os envia para a tela do jogo, como ocorre no *Groove*, que gera efeitos visuais conforme o jogador vai dançado em frente à câmera.

Dada a grande quantidade de dispositivos e às diferentes plataformas de jogos, é recomendado criar uma camada de abstração que abstraia o hardware (DALMAU, 2003), ou seja, age como um controle virtual. Por controle virtual, entende-se que o subsistema de

[4] www.bodypad.com/
[5] www.eyetoy.com/

entrada de dados não entra no mérito de que dispositivo está acessando, e sim que tipo de comando está sendo gerado por esse dispositivo e que ação no jogo esse comando representa. A Figura 2.14 mostra essa arquitetura.

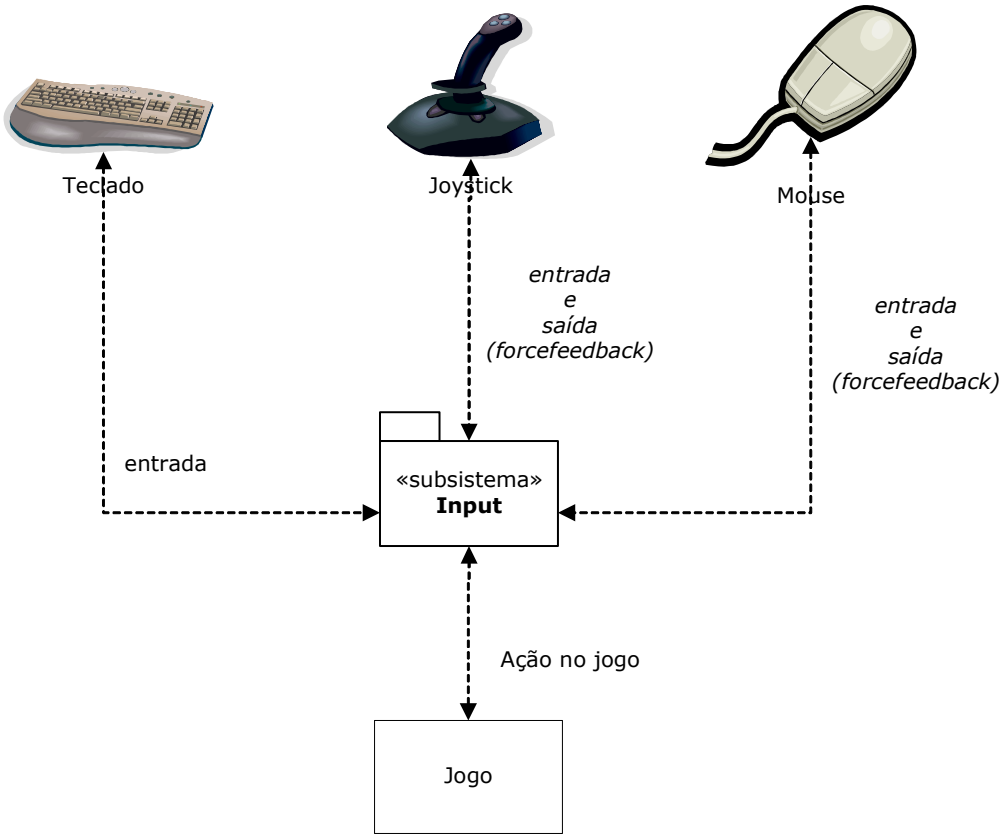

Figura 2.14 – Funcionamento do Controle Virtual

2.2.8. Subsistema Script (*Scripting Subsystem*)

Para desenvolvimento de um jogo, geralmente são escolhidas linguagens compiladas, como C ou C++ por um simples motivo: necessita-se desempenho.

Nas linguagens interpretadas, cada instrução é lida, interpretada e só então é executada. Nas linguagens compiladas, o código do aplicativo já é gerado em linguagem de máquina, ou seja, os passos de ler e interpretar não precisam ser feitos em tempo de execução, passando-se direto para a execução do aplicativo.

Um exemplo de linguagem compilada é C/C++, que é a linguagem mais utilizada em desenvolvimento de jogos (POIKER, 2003). Contudo, utilizar linguagens compiladas gera alguns problemas:

- Certas partes de um jogo, que sofrem muitas alterações durante o desenvolvimento, como a lógica do jogo, Inteligência Artificial e características de personagens, acarretam a necessidade de recompilação de todo o projeto;
- Necessidade de gerenciamento de memória manual: o programador deve se preocupar com a alocação e liberação de memória a todo o momento (DAWSON, 2002);
- Sobrecarga dos programadores: além de suas tarefas com a parte gráfica, entrada e saída, código de rede, etc., o programador tem de fazer todas alterações pedidas pela equipe de *design* e arte do jogo, já que esses profissionais não são especialistas em computação.

As linguagens interpretadas, por outro lado, são dinâmicas e trazem diversas vantagens:

- *Game designers* e artistas podem utilizar a linguagem para fazer ajustes em suas partes (lógica de jogo, ajustes de modelos de personagem), deixando o programador livre para suas tarefas como otimização das rotinas gráficas (ROUSE III, 2000);
- Os ciclos de testes e edição são reduzidos, já que alterações de lógica podem ser feitas e testadas sem necessidade de recompilação do jogo (ROUSE III, 2000);
- Os usuários finais (jogadores) podem alterar os parâmetros do jogo por conta própria, gerando modificações do jogo o que aumenta a vida útil do produto (DAWSON, 2002).

Entretanto, as linguagens interpretadas também possuem algumas desvantagens, como:

- Baixo desempenho, se comparadas com as linguagens compiladas, já que o jogo deve ler, interpretar e executar os *scripts*;

- A criação de uma linguagem *script* é demorada, já que esta deve ser rápida e estável, o que não é uma tarefa fácil.

Como solução para esse problema de escolha da linguagem, as empresas de hoje utilizam um misto dos dois tipos de linguagem. Utilizam linguagens compiladas para partes que necessitam desempenho, como rotinas gráficas e rede, e utilizam linguagens interpretadas para as partes que sofrem ajustes com maior freqüência e que não necessitam de muito desempenho como regras, inteligência artificial e parâmetros de personagens do jogo. Com isso as empresas garantem produtos de qualidade nos prazos de tempo desejados.

Exemplos de linguagens *scripts* utilizadas em jogos são a Unreal Script (SWEENEY, 1999), Lua (TECGRAF, 2005) e Python (ROSSUM, 2005).

2.3. Aplicações de Motores de Jogos

Além de jogos, os motores de jogos possuem outras três grandes áreas de atuação: comércio de *middleware*[6], modificações e educação/área científica.

2.3.1. Comércio de *Middleware*

Apesar de serem criadas para a comercialização de um jogo específico, muitas empresas estão se especializando no comércio do próprio motor de jogo, ou parte dele, vendendo-os para outras empresas desenvolverem seus jogos.

Como empresas que vendem motores de jogos completos pode-se destacar a ID Software (ID SOFTWARE, 2004), Epic Games (EPIC GAMES, 2005a), CryTek (CRYTEC, 2005), Valve Software (VALVE SOFTWARE, 2005) e RenderWare (RENDERWARE, 2005), sendo que a última oferece a opção de compra dos módulos em separado.

---

[6] Software que permite que uma aplicação interaja com outro software sem requerer que o usuário conheça o código do software.

Segundo Fristom (2003), reusar um motor de jogo já pronto em um novo projeto de jogo pode parecer uma idéia atrativa; porém, não se vê grande redução no tempo de desenvolvimento de um jogo, mas sim um ganho na parte de *design* de mapas, modelos etc., dado que *designers*, artistas e modeladores podem começar a desenvolver seus artefatos muito mais cedo, apresentando então um ganho de, aproximadamente, seis meses.

Outro cuidado que deve se tomar antes da aquisição de um motor de jogo é verificar para qual estilo de jogo ele foi projetado. Um motor de jogo construído para um jogo em primeira pessoa pode não servir para um jogo de estratégia em terceira pessoa. Se o jogo necessitar de componente de rede e o motor de jogo em questão não o possuir, poderá se gastar muito tempo no desenvolvimento e integração desses componentes.

Esse tipo de aquisição geralmente serve para empresas que não querem inovar em tecnologia de jogos, sendo que jogos que estabelecem novos padrões, como Half-Life 2 (VALVE SOFTWARE, 2005d) e Doom 3 (ID SOFTWARE, 2004d), levaram em média cinco anos de desenvolvimento cada.

2.3.2. Modificações

Muitos dos jogos hoje comercializados vêm acompanhados de ferramentas de edição que permitem ao usuário do jogo alterar o seu conteúdo, sem possuir grandes conhecimentos artísticos ou de programação. Esse tipo de modificação é conhecido como MOD.

Essas equipes de desenvolvimento, geralmente, são formadas pelos próprios jogadores que, como *hobby*, adicionam novos elementos ao jogo.

Muitas vezes, os próprios MODs são mais jogados que os jogos originais (LEWIS, JACOBSON, 2002), levando a uma maior longevidade do jogo em si. Um exemplo seria o MOD Counter-Strike para o jogo Half-Life, que, apesar de estar com quase seis anos, ainda é um dos jogos mais jogados na Internet (GAMESPY, 2005).

Tal sucesso leva as empresas desenvolvedoras dos jogos originais a disponibilizar ferramentas e documentação para possibilitar a alteração dos jogos e promover, muitas vezes, campeonatos com prêmios em dinheiro (EPIC GAMES, 2005d).

2.3.3. Educação e área Cientifica

Os jovens de hoje passaram anos jogando jogos complexos, estudos do exercito americano mostram que esses jovens são diferentes em termos de percepção e habilidades, comparados com seus pais e estão forçando uma mudança no ensino nas universidades e, entre essas mudanças, pode-se destacar (MACEDONIA, 2004):

- Multiprocessamento, ou seja, a capacidade de realizar diversas tarefas ao mesmo tempo;
- Mudança rápida de atenção, similar a de um executivo, mostrando uma rápida troca de contexto;
- Mudança na forma de raciocínio do dedutível e abstrato ao concreto;
- Mudança no foco de aprendizado, do ouvinte passivo para aprendizado baseado em descoberta e em exemplos.

Há mais de duas décadas o exército americano também vem realizando estudos e projetos utilizando jogos eletrônicos no desenvolvimento de simulações de treinamento (MACEDONIA, 2004) e muitas outras empresas (GAME2TRAIN, 2005) estão realizando pesquisas em educação e treinamento, utilizando jogos como base.

O motivo desse investimento é que muitos dos jogos hoje comercializados vêm acompanhados de ferramentas de edição que permitem ao usuário alterar o seu conteúdo sem possuir grandes conhecimentos artísticos ou de programação. Esse tipo de alteração é mais conhecida como MOD (*Modification)* e já foi mencionado no item anterior. Além da opção de criar um MOD, existem muitos motores de jogos disponíveis gratuitamente na Internet, que

possuem código-fonte aberto (SOURCEFORGE, 2005) Isso permite a adição de novos elementos tecnológicos ao jogo ou sua integração a sistemas já existentes, como sistemas de realidade virtual.

Alguns trabalhos apontam diversas vantagens em se utilizar motores de jogos ao invés de outras tecnologias como VRML ou Apple QuickTime VR (CALEF et al., 2002; SHIRATUDDIN et al., 2004). Entre essas vantagens estão: melhor desempenho, melhor apresentação visual para o usuário, uma interface gráfica já testada e grande compatibilidade com o hardware existente no mercado.

No site da Unreal Engine (EPIC GAMES, 2005c), é disponibilizada uma versão *runtime*[7] que pode ser utilizada gratuitamente em projetos educacionais ou sem fins lucrativos, e que não sejam relacionados a jogos de entretenimento.

O site Social Impact Games (GAME2TRAIN, 2005) também mantém uma lista com uma série de jogos voltados à educação, treinamento ou uso cientifico.

Como exemplo de jogo educacional, pode-se citar o jogo *Revolution*, que é uma modificação do jogo NeverWinter Nights (BIOWARE CORP, 2005) desenvolvido no Massachusetts Institute of Technology (MIT).

Esse jogo permite ao jogador vivenciar os acontecimentos nos Estados Unidos, no período de 1773 a 1783, assumindo o papel de um fazendeiro, escravo, político ou comerciante. O jogo também permite que diversos jogadores (estudantes) interajam entre si durante uma partida.

Seu projeto prevê como requisitos ser educacional e cativante, podendo ser jogado como uma atividade de sala de aula. O jogo foi dividido em capítulos, conforme a estrutura de ensino em sala de aula, onde os acontecimentos ocorridos em um período de tempo são ensinados por partes ou em capítulos, sendo que cada capítulo possui começo, meio e fim.

---

[7] *Runtime* – versão pré-compilada do *game engine*, sua utilização é similar à criação de uma modificação de um jogo.

Uma partida típica de *Revolution* envolve o estudante assumindo sua posição, em frente ao computador no começo da aula. Em seguida, ele entra no jogo e começa a jogar a partida num capítulo estabelecido pelo professor, com os outros alunos. Ao entrar no “mundo histórico”, são dadas ao jogador algumas tarefas e, ao longo da partida ocorrem eventos que exigem que os jogadores tomem uma decisão e cooperem entre si.

A Figura 2.15 mostra uma cena do *Revolution*, uma típica paisagem da época enfocada pelo jogo.

Figura 2.15 – Cena do jogo *Revolution*

2.4. Conclusões

Nesse capitulo procurou-se, rapidamente, mostrar ao leitor como se deu a evolução histórica dos motores de jogos e o crescimento de sua complexidade. Em seguida, discutiu-se cada um dos seus subsistemas, procurando caracterizá-los segundo o estado atual da arte. Finalizou-se o capítulo discutindo-se as aplicações dos motores e as vantagens da compra e aplicação de um motor “de prateleira” na indústria de jogos.

3. Padrões de Projeto e *Frameworks*

Padrões de Projeto ou *design patterns* tiveram sua origem, segundo Gamma et al. (2000), nos trabalhos do arquiteto Christopher Alexander (ALEXANDER, 1977 apud GAMMA et al., 2000) sobre padrões utilizados nas construções e nas cidades.

Na área de computação, padrões de projeto são "descrições de objetos e classes relacionadas, que são personalizadas para resolver um problema geral de projeto, num contexto particular" (GAMMA et al., 2000, p. 20), ou seja, capturam experiências bem sucedidas de estruturas de sistemas, que podem ser reutilizadas em outros projetos.

Em seu livro, Gamma et al. (2000) apresentam uma proposta de descrição de um padrão de projeto, como demonstra a Figura 3.1, que contém os principais campos que permitem descrever, detalhadamente, o padrão. Esta descrição detalhada permite que os interessados no reuso desses padrões possam compreendê-lo e determinar sua serventia, ou não, no problema que se apresenta no momento.

Nome e Classificação do padrão: o nome do padrão, que expressa a sua própria essência de forma sucinta e a sua classificação (Criação, Estrutural ou Comportamental)[8].

Intenção e Objetivo: uma curta declaração que diz o que o padrão de projeto faz, quais são seus princípios e sua intenção e qual o tópico ou problema particular de que o projeto de que trata. Esse tópico também é conhecido como "outros nomes bem conhecidos para o padrão", se estes existirem.

Motivação: um cenário que ilustra um problema de projeto e uma explicação sobre como o padrão ajuda a resolvê-lo.

Aplicabilidade: em quais situações o padrão de projeto pode ser aplicado.

Estrutura: uma representação gráfica das classes do padrão, usando uma notação adequada (UML ou OMT, por exemplo).

[8] A classificação dos padrões está apresentada no item 3.1 desse trabalho.

Participantes: as classes e/ou objetos que participam do padrão de projeto e suas responsabilidades.

Colaborações: como os participantes colaboram para executar suas responsabilidades.

Conseqüências: como o padrão suporta a realização de seus objetivos? Quais são seus custos e benefícios, se aplicado? Que aspecto da estrutura do sistema ele permite variar independentemente?

Implementação: o que se precisa conhecer quanto à implementação do padrão? Existem considerações específicas de linguagem?

Exemplo de código: fragmentos ou blocos de código, que ilustram como se pode implementar o padrão.

Usos conhecidos: Exemplos do padrão encontrados em sistemas reais.

Figura 3.1 - Uma proposta para a descrição de um padrão de projeto (GAMMA et al., 2000)

De acordo com Souza (SOUZA, 2004, p.20), “com as informações contidas na descrição de um padrão, é possível ao analista sintetizar de maneira ordenada o conhecimento adquirido em sua vivência, transformando esta experiência em um documento, permitindo que outros desenvolvedores e analistas compartilhem de seus conhecimentos”.

Os padrões de projeto permitem analisar uma solução de arquitetura num nível de abstração elevado, sem a necessidade de se detalhar sua implementação. Segundo Ecker, em seu projeto X-Engine (ECKER, 2003, p. 85), “as práticas reconhecidas de engenharia de software e padrões de projeto utilizadas no processo de desenvolvimento reduziram a quantidade de trabalho de recodificação e reprojeto nos estágios finais de desenvolvimento”.

Segundo Madeira (2001), o ponto de vista dos desenvolvedores pode afetar a interpretação do que é, ou não, um padrão. Um padrão pessoal - um padrão usado por um desenvolvedor especifico - pode ser um bloco qualquer de construção primitiva. Porém, os padrões de projeto têm um certo nível de abstração, pois eles não são projetados como listas ligadas e tabelas *hash*, que podem ser codificadas em classes e reutilizadas, e nem são

projetos de domínio específico complexos para uma aplicação, ou subsistemas. Padrões de projeto são descrições de comunicação de objetos e classes, que são personalizadas para resolver um problema de projeto geral, num contexto particular.

A seguir, discutem-se os vários tipos de padrões de projeto, classificando-os quanto à sua finalidade. Em seguida, apresentam-se os *frameworks* e caracterizam-se, como *frameworks*, os motores de jogos. Finalmente, discute-se o uso de padrões de projeto em jogos eletrônicos.

3.1. Classificação dos padrões de projeto

Como existem muitos padrões de projeto descritos na literatura, é necessária uma maneira de organizá-los para, assim, facilitar a consulta e relacioná-los de uma forma que auxilie no seu aprendizado.

Uma forma de classificar os diversos tipos de padrões é através de sua finalidade. Gamma et al. (1995) definem três tipos de finalidade para os padrões apresentados em seu livro:

- Padrões de Criação: preocupam-se com o processo de criação de objetos, ou seja, em como os objetos são instanciados ou construídos. Exemplo:
    - Padrão *Singleton*: impõe, por construção, que uma classe contenha somente uma instância num sistema e fornece um ponto global de acesso a essa instância.
- Padrões Estruturais: lidam com a composição de classes ou de objetos. Exemplo:
    - Padrão *Bridge*: separa uma abstração de sua implementação, de modo que as duas possam variar de forma, independente.
- Padrões Comportamentais: caracterizam a maneira pela qual classes ou objetos interagem e distribuem responsabilidades. Exemplo:

- Padrão *Iterator*: fornece uma maneira de acessar seqüencialmente os elementos de um objeto agregado (listas, árvores, vetores, etc.), sem expor sua representação subjacente.

Apesar de padrões de projeto estarem relacionados com a modelagem de uma aplicação (software), existem outros tipos de padrões utilizados pela área de computação, apontados por Souza (2004), como:

- Padrões de Análise: são padrões cujo foco está na modelagem do processo de negócio;
- Padrões de Reengenharia: são padrões cujo foco está na migração de sistema legados para um novo sistema.

Além disso, na área de jogos, estão surgindo os padrões de jogos ou *game patterns* (BJÖRK; LUNDGREN; HOLOPAINEN, 2003), que são um tipo de padrão mais focado na área de projeto de jogos ou *game design*, área relacionada às características e mecânica dos jogos do ponto de vista de jogabilidade (*gameplay*). Este tipo de padrão procura abstrair os elementos de jogo: terceira pessoa, isométrico, regras, etc., tendo em vista criar uma forma unificada e de fácil comunicação entre profissionais da área de jogos, mais especificamente, entre os projetistas de jogos ou *game designers*.

3.2. *Frameworks* e Motores de Jogos

"*Framework* é uma coleção de artefatos de software que é utilizável por várias aplicações diferentes. Esses artefatos são, em geral, classes, juntamente com o software exigido para utilizá-las" (BRAUDE, 2005). Como já definido, os *frameworks* podem ser utilizados para desenvolver diversas aplicações em um domínio especifico, isto é, um *framework* para aplicações Web não atende todos os requisitos de um *framework* para aplicações gráficas, já que esses domínios são diferentes. Os *frameworks* contêm diversos padrões de projeto em um

diferente nível de abstração, isto é, implementados na aplicação de um problema específico (MADEIRA, 2001).

Braude (2005) aponta três possíveis utilizações de *frameworks*:

- *Toolkits* são *frameworks* que são utilizados como um pacote de ferramentas de apoio. Exemplo: Java.math contém as classes `BigInteger` e `BigDecimal` para cálculos, mas esse pacote não faz nenhuma suposição arquitetônica do software que está sendo desenvolvido;
- *Frameworks* puros fazem suposições arquitetônicas sobre o sistema em construção, como, por exemplo, no pacote `Java.awt`, em que a classe `Container` dá suporte ao método `add(Component)`. Essa é uma propriedade arquitetônica;
- *Frameworks* que contêm algum tipo de controle, como, por exemplo, `Java.applet`, onde não se adiciona os métodos de controle `main`, mas sim os métodos da interface do `applet` que são implementados no método `init`, que é chamado quando o componente é acessado.

Silva (2003) descreve *frameworks* de componentes como “uma infra-estrutura de suporte aos componentes, sobre a qual estes funcionam e sem a qual os mesmos não poderiam ser executados e seriam, então, inúteis.” Exemplos:

- Entreprise Java Beans (EJB): *framework* de servidores e de *containers*;
- WaterBeans (SEI): visualização de tempo real de dados;
- COM+: implementação de *framework* integrado ao sistema operacional, no caso o MicrosoftWindows;
- CORBA: uma especificação de *framework* para objetos distribuídos.

Além disso, Madeira (2001) afirma que *frameworks* podem ser divididos em função da técnica usada para sua extensão:

- Caixa branca: a funcionalidade do *framework* é usada e estendida através de herança de classes abstratas e sobre-escrevendo métodos específicos, utilizando padrões de projeto como *Templates*. O problema desse tipo de *framework* é que os desenvolvedores devem ter profundo conhecimento de sua estrutura interna.
- Caixa preta: são definidas interfaces para componentes que podem ser "plugados" via composição de objetos ao *framework*. A desvantagem desses *frameworks* é que levam muito tempo para serem construídos e requerem uma definição das interfaces que antecipem um amplo conjunto de casos de uso "potencial".

Madeira (2001) mostra ainda uma lista de vantagens e desvantagens de *frameworks*:

Vantagens	Desvantagens
A média de tamanho dos projetos subseqüentes é encurtada	O primeiro projeto é muito longo
Realiza a independência de plataforma de maneira mais fácil	Bibliotecas que encapsulam os módulos devem ser desenvolvidas para cada plataforma
Apresenta código mais confiável	O código é mais genérico e mais difícil de desenvolver
Apresenta código reusável e gerenciável	Inicialmente, o código toma muito tempo para ser desenvolvido
O conhecimento é propagado entre todos os desenvolvedores	Novos desenvolvedores têm de que aprender a lidar com bibliotecas não familiares
Aumenta a visibilidade de progresso do projeto	Necessita de melhor administração
Aumenta a especialização dos desenvolvedores	Menor flexibilidade

Tabela 3.1 – Vantagens e Desvantagens de *frameworks* (Madeira, 2001)

3.2.1. Motores de Jogos

Motores de jogos são *frameworks* que atuam sobre o domínio de jogos eletrônicos. Uma das características dos motores de jogos, com relação a *frameworks*, é que dificilmente se pode ter um *framework* que atenda a todos os estilos de jogos disponíveis, isto é, o domínio de jogos eletrônicos possui subdomínios, que estão relacionados à forma como se joga e ao

tipo de jogo, sendo que cada um desses subdomínios possui um conjunto de algoritmos e estruturas de dados particulares, como ilustra a Figura 3.2.

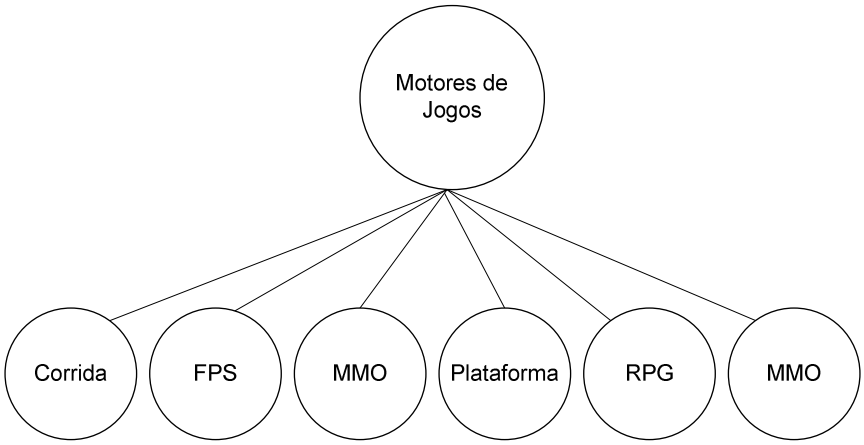

Figura 3.2 - Motores gráficos, classificados por estilo de jogo

Do ponto de vista de reuso, motores de jogos podem ser tanto caixa preta, como caixa branca, dependendo do contexto em que serão utilizados.

Já no contexto de modificações criadas pelos usuários, os motores de jogos são caixa branca, pois o desenvolvedor do MOD deve estender uma das classes básicas exportadas pelo motor e implementar sua opção. Os motores Unreal e Source demonstram bem esse uso.

Quando uma empresa compra uma licença para uso de um motor de jogo, ela pode adicionar componentes ou trocar componentes existentes no motor por outros que atendem a sua necessidade, sem corromper a arquitetura do motor. Nesse cenário, os motores gráficos são do tipo caixa preta. Como exemplo, o motor Unreal, apesar de não ter sido desenvolvido para suportar jogos *massive multiplayer*, foi usado no jogo LineAge II (NCSOFT, 2004), sendo feitas alterações no componente de rede, mas utilizando o resto do motor, sem alterações.

3.3. Aplicações de padrões de projeto

A seguir, descrevem-se algumas aplicações de padrões de projeto na área de jogos eletrônicos:

- Nguyen e Wong (2002) mostram o uso de padrões de projeto para abstrair o sistema de *rendering* de um jogo de tabuleiro qualquer. Através da utilização de padrões, o sistema de regras de um jogo de tabuleiro 2D pode ser separado da sua visualização, permitindo assim que o tipo do jogo (xadrez, damas, etc.) possa variar, independente da forma que é apresentado ao jogador (Visualização 2D, Visualização 3D, Isométrica, etc.).
- Ponder et al. (2003) mostram um *framework* (VHD++ Development Framework) para desenvolvimento de projetos de realidade virtual, que vem sendo usado em projetos europeus de simulação de personagens. O sistema é fortemente voltado à orientação a objetos, demonstrando uma clara separação das classes abstratas e concretas, propiciando a implementação de diversos padrões de projeto diferentes.
- Vidal et al. (2004) criaram uma ferramenta de autoria de ambientes virtuais que abstrai o motor gráfico utilizado. Nesse caso, foram utilizados os padrões *Wrapper* e *Façade* para abstrair o motor gráfico 3DSTATE da aplicação de realidade virtual.
- Em Dollner e Hinrichs (2002), é apresentado o *Generic Rendering System*, um *framework* para geração de gráficos, que abstrai o sistema de *rendering* que a aplicação utiliza. A proposta não é apenas abstrair a API gráfica, mas também o sistema de *rendering*, como, por exemplo, o sistema para tempo real ou foto-realístico, como um todo. Descreve-se o processo de abstração da API gráfica e dos diversos objetos utilizados por um *render*, como texturas, vértices, técnica de *rendering*, objetos emissores de luz, etc., que são todos abstraídos, permitindo uma migração fácil de uma API para outra.
- Na área de jogos pode-se citar o motor criado por Ecker (2003), no qual os padrões de projeto são utilizados para abstrair a versão da API gráfica (DirectX 8.1 ou DirectX 9) e a forma de construção do grafo de cena e dos modelos de triangularização (*mesh*).

3.4. Conclusões

Nesse capítulo, demonstrou-se como através de estruturas mais simples, como padrões de projeto, consegue-se construir estruturas mais complexas, no caso *frameworks*, permitindo o seu reuso em diversos projetos. Na área de motores de jogos, mostrou-se como estes podem ser reutilizados de diversas maneiras nos dias atuais, com ou sem alteração do núcleo (motor do jogo).

No capítulo 4, será mostrado como padrões de projeto são utilizados internamente no motor de jogo para abstração da API.

4. Abstração da API Gráfica Utilizando Padrões de Projeto

Este capítulo estuda a aplicação de padrões de projeto para abstração da API gráfica.

O capítulo se inicia especificando os requisitos considerados necessários à abstração da API gráfica através de padrões de projeto, utilizando-se, em seguida, dessa especificação para a análise de dez motores de jogos de código aberto existentes no mercado.

Justifica-se a escolha da API gráfica como foco de aplicação de padrões de projetos no fato de que diversos módulos de um motor de jogo, como som, física e rede, fazem uso de APIs e *middleware* de terceiros. Assim, o que for aqui aplicado na API Gráfica, poderá ser também utilizado para os demais módulos com as devidas adaptações.

No entanto, deve-se ressalvar, antes de se iniciar o capítulo, que não será aqui abordada a parte de gerenciamento de cena do módulo gráfico, porque, apesar de ser uma parte importante deste módulo, a mesma, por si só, possui um grau de complexidade elevado, e não é interesse desse texto se aprofundar neste tópico.

Finalmente, ressalte-se que os trechos de código mostrados estão baseados em C++, pois a maioria dos motores analisados estão escritos nessa linguagem. No entanto, na parte de análise, são abordados motores de jogos baseados também na linguagem Java, que começa a se impor também na área de jogos (NAKAMURA et al., 2003; BITTENCOURT et al., 2003).

4.1. Requisitos

Antes de se iniciar a análise dos motores de jogos, se faz necessário elencar uma especificação de requisitos básicos que estes devem apresentar e que sirva à realização de comparações entre eles.

No capítulo 1, demonstrou-se que os objetivos da utilização de padrões de projeto nesse trabalho são:

- Modularidade: permitir baixo acoplamento entre módulos, possibilitando a troca de um deles sem afetar (muito) os demais. Em primeira instância, a modularidade que é analisada diz respeito, principalmente, à API gráfica, ou seja, em se permitir a fácil substituição de uma API por outra. Consideram-se como candidatas ao uso a biblioteca OpenGL, a DirectX e eventuais bibliotecas desenvolvidas por usuários;
- Extensibilidade: permitir a evolução de um módulo, sem grandes impactos nos demais módulos.

Além desses objetivos, ponderou-se que é desejável que o módulo gráfico tenha as seguintes características:

- Interface de uso comum: para o motor do jogo, a API gráfica utilizada no momento deve ser transparente, isto é, as chamadas de métodos feitas pelo motor à API são sempre as mesmas, independente de qual delas esteja em uso;
- Global ao motor: o módulo gráfico deve poder ser acessado em qualquer ponto/módulo do motor do jogo;
- Unicidade: o módulo gráfico deverá ser único durante toda a execução do motor do jogo, isto é, não poderá haver duas instâncias do módulo gráfico, sendo executadas em paralelo.

Além disso, deve-se levar em consideração que as APIs gráficas OpenGL e DirectX são responsáveis, basicamente, por uma única tarefa: desenhar os objetos de um grafo de cena na tela do usuário. Essa tarefa, em si, possui diversas subtarefas associadas, que já foram apresentadas no Capítulo 2.

Assim, o módulo gráfico deverá permitir a escolha de qual API se vai utilizar, abstrair as suas funcionalidades de *rendering* específicas, além de atender aos requisitos apresentados. Problemas de migração entre plataformas deverão sempre ser levados em consideração, mas não serão aqui abordados em profundidade.

Os exemplos de código ou de arquivos aqui utilizados estarão baseados na plataforma x86, sendo executados no sistema operacional Windows XP.

4.2. Abordagem utilizando generalização

Neste item, se mostrará, em uma primeira iteração, como seria feita a modelagem do módulo gráfico em um paradigma orientado a objetos, abstraindo-se a biblioteca gráfica.

Uma primeira forma de abstrair a API gráfica é criar uma interface `IRenderer`, que o programador do jogo irá utilizar para desenhar os objetos e cenas na tela do usuário. A Figura 4.1 apresenta essa interface. A `IRenderer` funciona como uma representação de todo o subsistema de *rendering* para o programador do jogo, enquanto evidencia a decomposição do motor de jogo em dois subsistemas: o de *rendering* e o núcleo.

Figura 4.1 - Interface IRenderer

Dentro do paradigma de orientação a objetos, a interface pode ser definida como um conjunto de tipos abstratos de dados ou objetos. A interface descreve os dados que podem ser acessados no subsistema (de *rendering*) e as operações disponibilizadas (SOMMERVILLE, 2003). Assim, a `IRenderer` define o conjunto de serviços deste subsistema que podem ser usados pelo programador do jogo. Na Figura 4.1, os serviços ofertados são do tipo `init`, que faz a iniciação da API gráfica, e `draw,` que desenha os objetos na tela, descritos por uma lista de vértices (parâmetro `vertexList`). Mais adiante, na Figura 4.10, serão mostrados outros métodos que compõem esta interface.

Para que o programador do jogo utilize esse modelo, deverá se valer do trecho de código apresentado na Figura 4.2, onde se mostra a instanciação de um objeto que emprega a API DirectX. O Magic Engine 3, criado por Eberly (2000), utiliza essa abordagem.

```
void main() {
...
   IRenderer *render = (IRenderer *)new DirectXRender();
...
}
```

Figura 4.2 - Utilização de IRenderer pelo programador do jogo

O passo seguinte é detalhar a implementação interna do subsistema de *rendering*, considerando as várias APIs que se deseja utilizar. A Figura 4.3 apresenta este projeto, onde `OpenGLRenderer` e `DirectXRenderer` são classes-filhas de `IRenderer`. O código associado a cada API na figura retrata a instanciação da API através das funções `Init()` correspondentes.

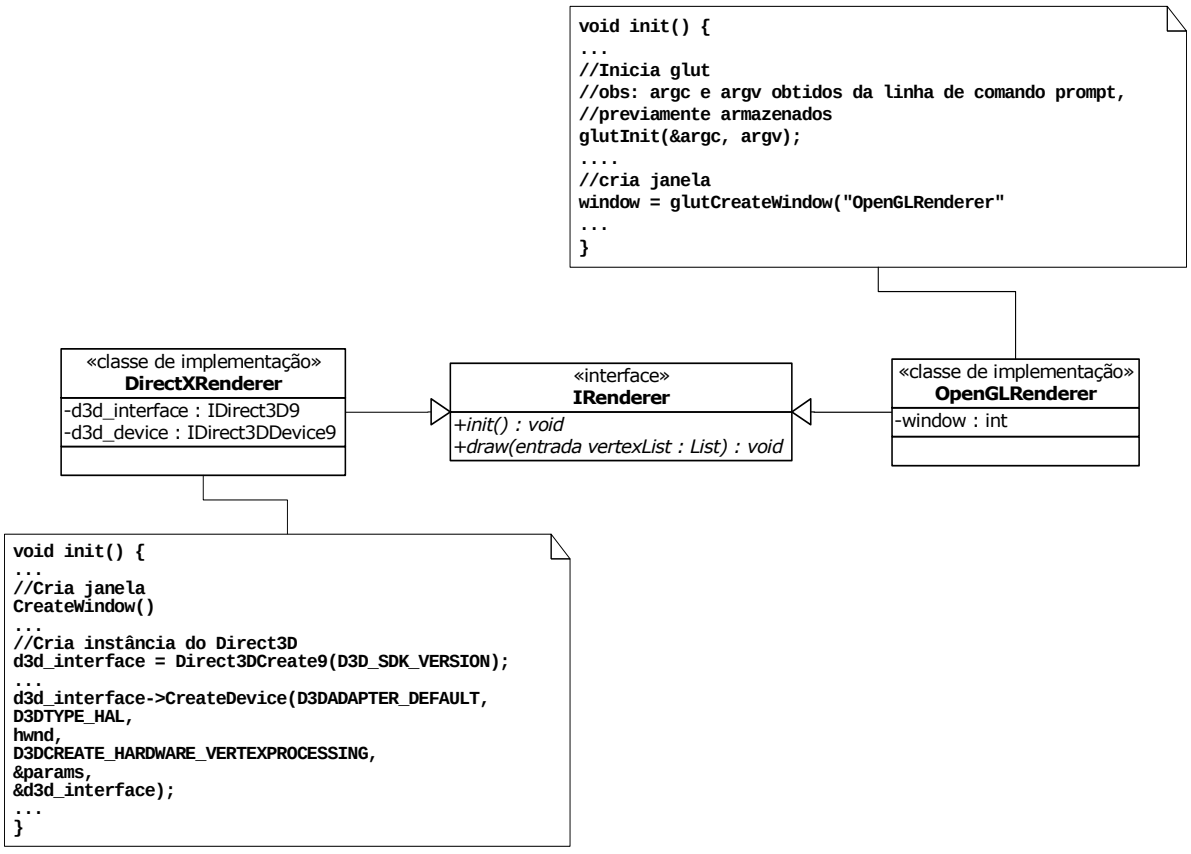

Figura 4.3 - Implementação do módulo gráfico utilizando generalização

Essa abordagem, entretanto, acarreta alguns problemas:

- O programador do jogo deve implementar algum tipo de mecanismo para definir qual API se vai utilizar. Uma solução mais amigável deveria deixar os detalhes desta implementação sob a responsabilidade da parte gráfica.
- Se, futuramente, tentar-se adicionar uma terceira API ao conjunto, por exemplo, um `SoftwareRenderer`, será necessário alterar o código dos jogos que a utilizam, quando se fizer a instanciação da parte gráfica. Este fato dificulta a manutenção e a evolução do jogo.

No próximo tópico, abordar-se-á a solução do ponto de vista de padrões de projeto, em que se pretende demonstrar uma forma de resolver esses problemas e atender aos requisitos do tópico 4.1.

4.3. Abordagem utilizando padrões de projeto

Como ponto de partida, é necessário refatorar o exemplo de generalização apresentado no tópico 4.2 para utilizar padrões de projeto.

Com o desacoplamento da API gráfica do motor de jogo, pode-se evoluir qualquer das APIs gráficas, sem causar impacto entre as implementações (OpenGL e DirectX) ou no *game engine* propriamente dito, já que a introdução de `IRenderer` permitiu isso.

Para facilitar a atualização, separa-se cada abstração de API em um componente isolado (biblioteca dinâmica), que será carregado em tempo de execução, caso necessário. Para isso, utiliza-se a divisão de componentes mostrada na Figura 4.4.

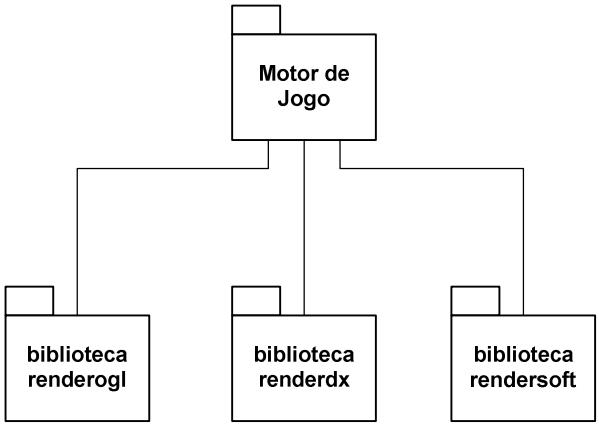

Figura 4.4 – Organização dos componentes do módulo gráfico

Considerando-se a Figura 4.4 tem-se:

- `biblioteca renderogl` - é a ligação do motor com a API OpenGL;
- `biblioteca renderdx`- é a ligação do motor com a API DirectX;
- `biblioteca rendersoft` - é um *renderer* implementado em software.

Esse tipo de divisão nos induz às seguintes questões:

- Como manter os requisitos e características desejadas, apresentadas no tópico 4.1?
- Como saber qual biblioteca utilizar?

Para resolver estes problemas, aplicam-se os padrões de projeto descritos por Gamma et al. (2000) e Grand (2002): *Singleton*, *Factory Method*, *Adapter* e *Dynamic Linkage.*

Em seu livro, Gamma et al. (2000) descrevem *Singleton* como um padrão que tem como objetivos “garantir que uma classe tenha somente uma instância e fornecer um ponto (global) de acesso para a mesma”. Essas funcionalidades são interessantes para a `IRenderer`, já que, com esta, é viável se atender às características de Unicidade e Global do motor do jogo, propostas no item 4.1.

Para aplicar este padrão, cria-se uma nova classe chamada `RendererManager`, que será o gerenciador de APIs. A Figura 4.5 mostra os atributos e métodos da classe.

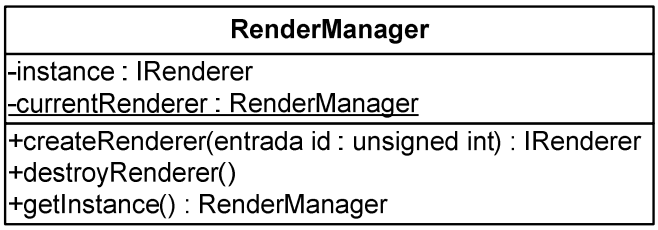

Figura 4.5 - Classe `RenderManager` (*Singleton*)

O atributo `instance` é estático e privado, e é nele onde ficará armazenada a instância de `RenderManager` e no atributo `currentRenderer` fica armazenada a API desejada. Os métodos apresentados são estáticos e públicos e servem para instanciar (`createRenderer`) e destruir (`destroyRenderer`) o objeto contido em `currentRenderer`. Outro aspecto que se deve observar (mas que não está mostrado na Figura 4.5), é que a classe `RenderManager` segue o padrão de projeto *Singleton*, assim sendo deve-se utilizar o método `getInstance` para recuperar uma instância da classe.

Para resolver o problema de qual API instanciar, utilizar-se-á o padrão *Factory Method* (Gamma et al., 2000) em conjunto com o padrão *Dynamic Linkage* (GRAND, 2002).

Factory Method é um padrão cujo objetivo é “definir uma interface para criar um objeto, mas deixar as subclasses decidirem que classe instanciar. *Factory Method* permite assim, deletagar a instanciação para as subclasses da classe-mãe”, ou seja, permite a criação de implementações diferentes, no exemplo, para OpenGL ou DirectX Graphics, sem que se tenha de conhecer as classes efetivamente utilizadas na implementação.

Dynamic Linkage permite que um programa, dada uma requisição, carregue uma classe arbitrária que implementa uma interface conhecida, ou seja, neste caso, especifica uma biblioteca de *rendering*.

Para adicionar estes padrões de projeto à solução, é preciso realizar duas alterações.

Primeiro, cada biblioteca dinâmica (`renderogl`, `renderdx` e `rendersoft`) deve exportar[9] uma função, `Create`, que é responsável por retornar uma instância da `IRenderer`, referente àquela implementação. A declaração desta função é mostrada na Figura 4.6.

```
__declspec(dllexport) IRenderer* Create (void);
```

Figura 4.6 - Função para criação do *render*

Em seguida, é adicionado o *Factory Method* propriamente dito, acrescentando-se ao método `createRenderer` da classe `RendererManager` um bloco de decisão usado para carregar a biblioteca desejada, como mostra a Figura 4.7.

```
const IRenderer*
RendererManager::createRenderer(int id) {
   if(currentRenderer == NULL) {
      //chama o create da biblioteca
correspondente
      loadLibrary(id);
   }
   return instance;
}
```

Figura 4.7 - *Factory Method* e *Dynamic Linkage* aplicados à carga da biblioteca gráfica

O método `loadLibrary` cuida da parte da carga da biblioteca e da chamada da função `Create`, ou seja, é a aplicação do padrão *Dynamic Linkage*.

Como se pode observar, a classe `RendererManager` administra a criação e destruição do tipo da biblioteca encarregada da tarefa de *rendering*, além de garantir que apenas uma instância do pacote gráfico estará em execução, num dado momento.

Assim, o programador do jogo, para iniciar o sistema de *rendering* precisaria apenas do trecho de código apresentado na Figura 4.8.

```
void main() {
...
   IRenderer *render =
RendererManager::createRenderer(RendererManager::DIRECTX);
...
}
```

Figura 4.8 - Instanciação da API gráfica após aplicação dos padrões de projeto

[9] Exportar é o termo usado para funções que são disponibilizadas para acesso externo em uma biblioteca.

Os atributos `RendererManager::DIRECTX` e `RendererManager::OPENGL` são constantes que indicam qual API deve ser instanciada.

Este método de instanciação da API gráfica poderia ser melhorado ainda mais se for utilizado um arquivo de configurações, em que estariam descritas todas as APIs que o sistema suporta. A Figura 4.9 traz um exemplo deste tipo de configuração gráfica, mostrando o diretório onde estão as bibliotecas gráficas (`RenderDir`), a quantidade de bibliotecas que se tem (`NumRenders`) e o nome dos arquivos das bibliotecas (`Render1`, `Render2` e `Render3`).

```
[renders]
RenderDir=/plugins
NumRenders=3
Render1=renderogl.dll
Render2=renderdx.dll
Render3=rendersoft.dll
SelectedRender=renderogl.dll
```

Figura 4.9 - Arquivo de configurações

Com isso, o método `createRender` lê o arquivo de configurações toda vez em que é chamado e verifica quais bibliotecas devem ser carregadas e qual a biblioteca deve ser usada como padrão (`SelectedRender`). Os motores Ogre3D (OGRE3D, 2005) e JFrog (BITTENCOURT et al., 2003) mostram este tipo de recurso com maior profundidade.

Ao projetar um sistema de *rendering* que suporte diversas APIs, um dos problemas com que se depara é como tratar as diversas estruturas de dados e assinaturas[10] de métodos. No caso do OpenGL e DirectX, algumas de suas diferenças foram mostradas no capítulo 2.

Criar uma interface contendo todas as assinaturas se torna inviável, já que a manutenção da interface e de suas implementações se tornaria muito trabalhosa, dificultando a evolução do software.

Para estes casos, deve-se definir as estruturas de dados e assinaturas da interface necessárias ao sistema de *rendering*, independente da API a ser usada, como, por exemplo,

[10] Assinatura de um método é o nome do método, seus parâmetros e o tipo desses parâmetros.

estruturas de vértices e matrizes. Para cada API deve ser criada uma “casca” que passe estes tipos de dados definidos pelo motor do jogo e os adapte à API que será utilizada.

Este processo de criação da “casca” é conhecido como padrão de projeto *Adapter* ou *Wrapper* (GAMMA et al., 2000).

O padrão *Adapter* é descrito como uma forma de converter a interface de uma classe em outra interface, esperada pelos clientes, permitindo que interfaces incompatíveis trabalhem em conjunto, o que, de outra forma, seria impossível.

Aplicar *Adapter* (no caso um *Adapter* de objetos) à solução que está sendo desenvolvida não requer muito trabalho, bastando que, para cada API, seja criada uma classe que implemente a interface `IRenderer`, como mostra a Figura 4.10.

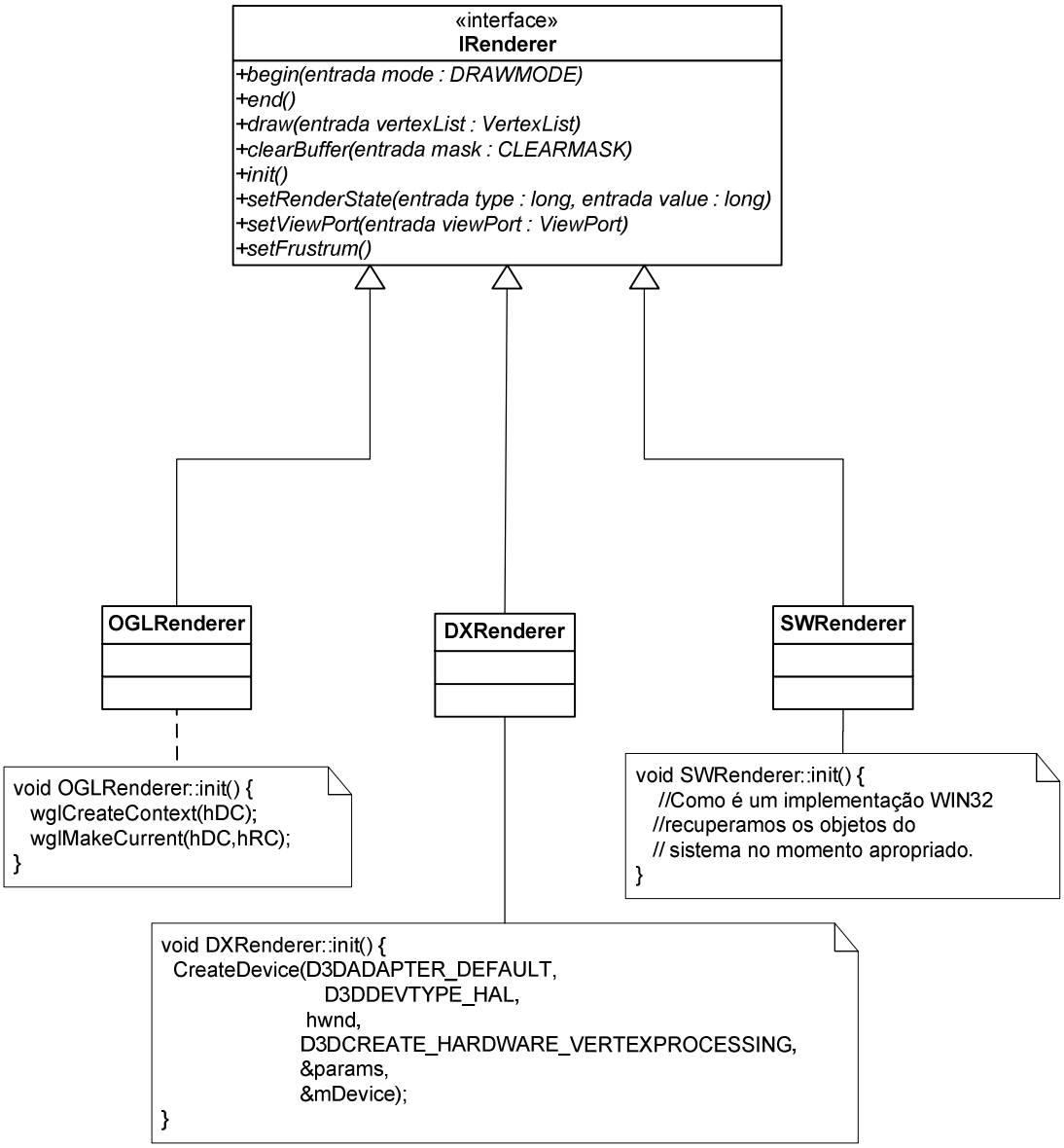

Figura 4.10 – Padrão de projeto *Adapter* aplicado à solução

Cada uma das classes que encapsulam uma API gráfica fica responsável, neste caso, por determinar o tipo básico definido no motor do jogo, como vértices, matrizes, regras do sistema de coordenadas, etc., e convertê-lo para o tipo esperado pela API.

Além disso, objetos do sistema de *rendering*, como texturas, *vextex*/*pixel* e *shaders*, também exigem que sejam criadas classes internas ao motor do jogo e, ao serem utilizadas, sejam convertidas para o formato esperado pela API. Este processo de abstração dos tipos de dados de um sistema de *render* é abordado com maior profundidade por Dollner e Hinrichs (2002).

Com os requisitos definidos, será mostrado, no próximo tópico, como diversos motores de jogos comerciais e acadêmicos fazem para abstrair, ou não, a API gráfica a ser utilizada.

4.4. Motores selecionados para análise

Entre os motores de jogos encontrados, estão listados a seguir os selecionados para análise:

- Família Quake Engine: composta por: Quake I (ID SOFTWARE, 2004a), Quake II (ID SOFTWARE, 2004b) e Quake III (ID SOFTWARE, 2004c);
- enJine (NAKAMURA et al., 2003);
- JFrog (BITTENCOURT et al., 2003);
- ForgeV8 (MADEIRA,2001) e ForgeV16 (TORRES, 2003);
- Ogre3D (OGRE3D, 2005);
- Wild Magic Engine (EBERLY, 2004);
- X-Engine (ECKER, 2003).

A escolha dos motores gráficos foi feita com base na disponibilidade do código fonte para análise, tendo-se priorizado os motores utilizados em algum jogo comercial.

A análise terá como foco mostrar como cada motor de jogo se relaciona com a API gráfica utilizada. Para isso, é feita uma introdução sobre o motor gráfico em discussão, demonstrando-se a estrutura interna da solução por este adotada, seguida de um texto explicativo, finalizando-se com quais dos requisitos mostrados no tópico 4.1 são atendidos (ou não) e como isso é feito.

4.4.1. Família Quake Engine

A família de motores de jogos Quake é composta por três membros, os quais são apresentados a seguir.

4.4.1.1. Quake I Engine

Quake I Engine foi desenvolvido pela empresa Id Software para o jogo Quake I, lançado em 1996. Este foi um dos primeiros motores de jogos a utilizar ambientes e atores em 3D, formados por texturas e polígonos.

Uma das primeiras características deste motor foi possibilitar aos usuários do jogo alterá-lo, usando uma linguagem particular, a QuakeC. Assim, o jogador pode adicionar novos elementos, como mapas, modelos e regras. Também foram vendidas licenças do motor - que incluíam o código fonte - a outras empresas de jogos, que, a partir deste jogo, criavam outros, com características mais avançadas. Pode-se citar o Half-Life (VALVE SOFTWARE, 2005c) e Daikatana (ION STORM, 2000).

No Quake I, os desenvolvedores tiveram de construir o sistema de *rendering* em software, já que, na época de seu lançamento, as placas aceleradoras 3D para uso doméstico não existiam, como ocorre nos dias de hoje.

Com o surgimento das primeiras placas aceleradoras 3D de uso doméstico, como as placas com *chipset* VooDoo da empresa 3DFX, foi lançada uma versão do Quake I com suporte ao OpenGL. A essa versão se deu o nome de GLQuake, que foi adaptada também à plataforma Linux/Unix.

Especificação do Módulo de *Rendering*

O sistema de *rendering*, totalmente implementado em software, foi escrito em uma combinação de linguagem C e Assembly e, mais tarde, foi lançada uma versão com suporte a placas 3D, utilizando a API OpenGL.

Para gerar o código executável para as diversas versões em software ou OpenGL, são feitas compilações diferentes, isto é, é gerada uma versão do programa para cada API. A Figura 4.11 ilustra esse processo:

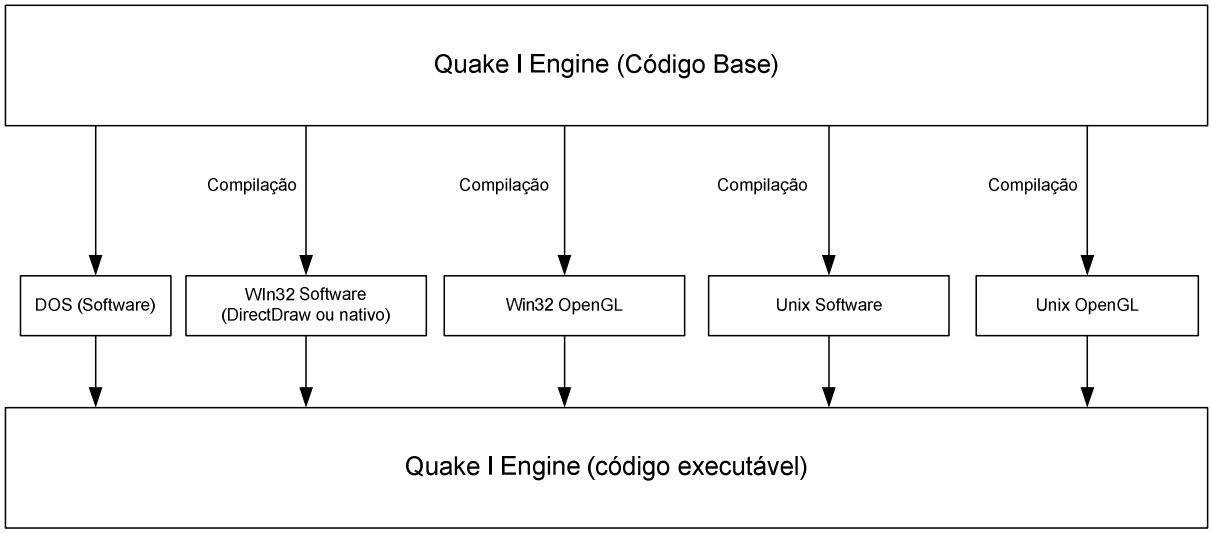

Figura 4.11 – Geração do código executável do Quake I Engine

Para fazer a ligação do programa principal com a API, Quake I Engine utiliza variáveis do tipo “ponteiro para função”, um recurso oferecido pela linguagem C. Estas variáveis guardam os endereços de funções, que assim podem ser fornecidos em tempo de execução. Assim, em tempo de execução, pode-se trocar a função apontada por essa variável e utilizar, para realizar as chamadas, as funções de *rendering* que se mostrarem mais convenientes. O uso de variáveis do tipo “ponteiro para função” é feito, porque são utilizadas bibliotecas carregadas em tempo de execução, que necessitam de funções específicas do Sistema Operacional para serem recuperadas.

Aderência aos Requisitos

Tabela 4.1 - Análise de Requisitos Quake I Engine

Requisito	Adere?	Observações
Modularidade	Parcialmente	Para APIs diferentes, é necessário recompilar todo o código.
Extensibilidade	Parcialmente	A extensão se dá pela criação de um *branch* do código-fonte base e troca de arquivos-fontes necessários.
Interface de uso comum	Sim	Funções abstraem a API base, necessitando apenas da inclusão de referência ao código (diretiva `#include` da linguagem C)
Global ao motor	Sim	Funções que abstraem a API acessam uma variável global que armazena uma referência para a API , utilizada.
Unicidade	Sim	Existe apenas uma referência à estrutura.

Como pode ser visto na Tabela 4.1, Quake I Engine adere a boa parte dos requisitos, mesmo apresentando alguns problemas no requisito Modularidade. Apesar de ser possível separar os módulos do motor em grupos de arquivos fonte, qualquer alteração necessita de nova geração de arquivo executável. Da mesma forma, a única maneira de estender as funções é através de alteração dos arquivos fontes existentes, que também leva à uma nova geração de versão executável.

Com relação aos padrões de projeto, Quake I Engine instancia a API gráfica através de um método que se assemelha ao padrão *Factory Method*, apresentado no tópico 4.3, em que uma função faz as chamadas ao método da API compilada na versão corrente do motor. Cabe notar que, como este motor não foi implementado segundo o paradigma de orientação a objetos, o conceito de padrão de projeto tem de ser estendido à programação procedimental para ser usado na análise.

4.4.1.2. Quake II Engine

Quake II Engine pertence à segunda geração de motores Quake, criados pela ID Software. Com a popularização das placas aceleradoras 3D, Quake II Engine foi desenvolvido com foco na sua utilização, mas ainda é dada ao projetista a opção de *rendering* por software.

Quake II Engine foi escrito em linguagem C, com otimizações feitas em linguagem Assembly.

Outra característica interessante desta segunda geração é a modularização do motor do jogo, separando os módulos do cliente (de uso do jogador) do módulo do servidor (controle de regras e dos estados do jogo), construindo-se uma arquitetura cliente/servidor. Permite-se a criação de servidores *stand-alone*, apenas para controlar as partidas disputadas através da Internet.

Especificação do Módulo de *Rendering*

A Figura 4.12 ilustra a forma de instanciação da API.

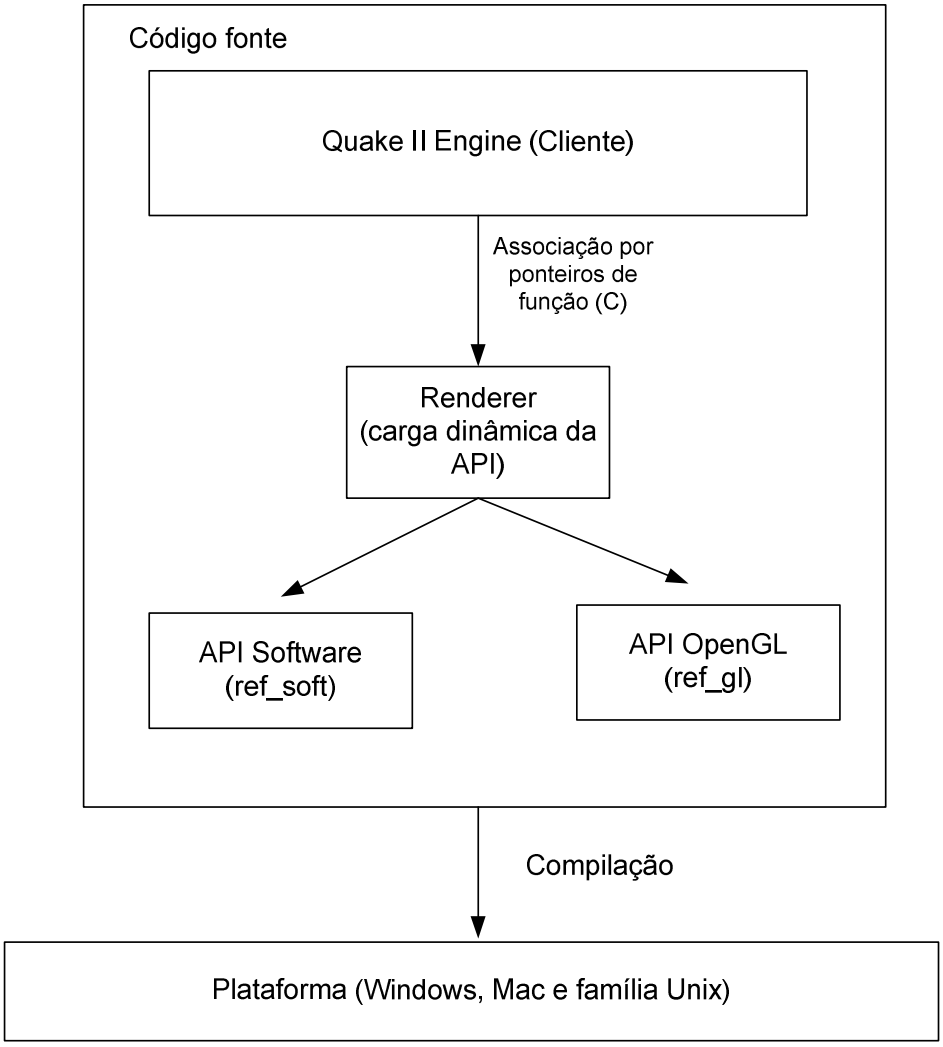

Figura 4.12 - Compilação e inclusão de bibliotecas Quake II Engine

Quake II Engine é composta por duas bibliotecas, a que realiza o *rendering* por software e a que realiza o *rendering* pela API OpenGL (OPENGL ARCHITECTURE REVIEW BOARD, 2004), que são carregadas dinamicamente pelo motor, dada a opção de *rendering* escolhida pelo usuário.

Ainda nessa versão, o motor gráfico utiliza variáveis do tipo ponteiro para função para, de uma forma única, realizar chamada aos métodos de *rendering* da API escolhida.

Outro ponto interessante é que, ao analisar o código da biblioteca que utiliza a OpenGL, pode-se notar caminhos de execução específicos para hardware de vários tipos diferentes (Windows, Mac e família UNIX). Isto ocorre porque as empresas de hardware distribuem *drivers* que não são 100% compatíveis com a especificação OpenGL. Assim, o motor gráfico da época tinha de realizar pequenos ajustes, dado o hardware utilizado pelo usuário.

Aderência aos Requisitos

Tabela 4.2 - Análise de Requisitos Quake II Engine

Requisito	Adere?	Observações
Modularidade	Parcialmente	Para APIs diferentes é necessário recompilar todo o código, mas observam-se sinais de melhora, podendo o código que abstrai as APIs ser compilado em separado.
Extensibilidade	Parcialmente	A extensão se dá pela criação de um *branch* do código fonte-base e alteração dos arquivos-fontes que fazem a carga das bibliotecas dinâmicas.
Interface de uso comum	Sim	Funções abstraem a API-base, necessitando apenas da inclusão de referência ao código (diretiva `#include` na linguagem C)
Global ao motor	Sim	Funções que abstraem a API acessam uma variável global que armazena uma referência para a API utilizada.
Unicidade	Sim	Existe apenas uma referência à estrutura.

Como mostra a Tabela 4.2, Quake II Engine mantém diversas características de Quake I Engine, sendo que a maior diferença está nos requisitos de Modularidade e Extensibilidade. O módulo de *rendering* possui o código que faz interface com as APIs dividido em bibliotecas, uma para cada API, carregadas em tempo de execução, o que possibilita que se adicionem outras APIs. Para isso, exige-se apenas a alteração do código do núcleo do motor que faz a carga das bibliotecas. Em caso de otimizações e correções de algumas das bibliotecas de API,

basta alterar as fontes da biblioteca em questão e gerar uma nova versão da biblioteca. Essa forma de carga de bibliotecas se assemelha à combinação dos padrões *Factory Method* e *Dynamic Linkage* e o uso de ponteiros de função para esconder a biblioteca que está sendo usada é parecido com o padrão *Adapter*, uma vez que esse ponteiro é o mesmo para o motor de jogo independete da API que está sendo usada, a função apontada é que cuidará da adaptação dos parâmetros para o padrão da API que está sendo utilizada.

4.4.1.3. Quake III Engine

Esta foi a terceira geração da Quake Engine, que teve como um de seus requisitos básicos a utilização de uma placa aceleradora 3D, ou seja, não se tem mais a opção, dentro do jogo, de escolher *rendering* por software.

Especificação do Módulo de *Rendering*

A Figura 4.13 ilustra a estrutura interna do motor Quake III Engine.

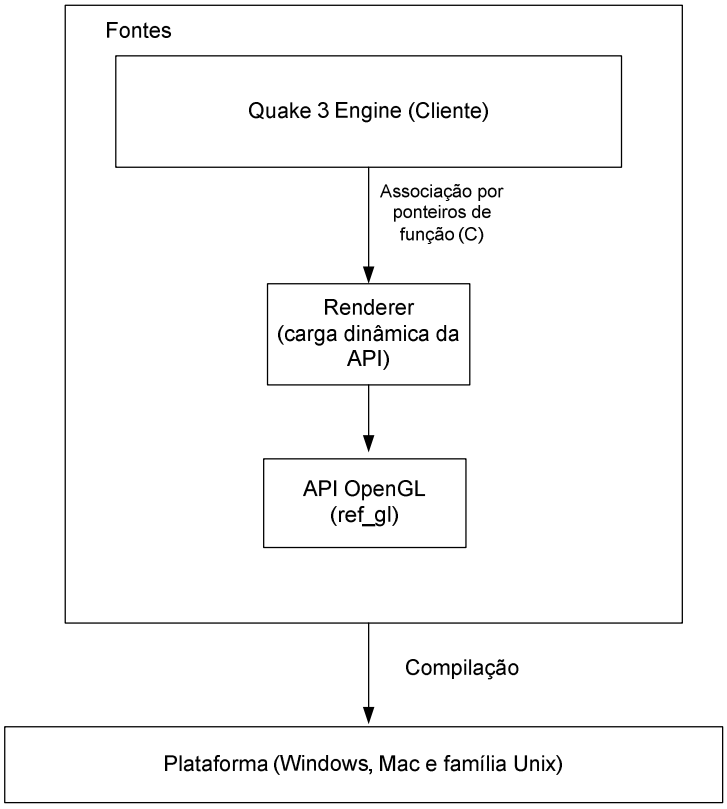

Figura 4.13 - Compilação e inclusão de bibliotecas para Quake III Engine

Quake III Engine adotou a OpenGL como API gráfica padrão, aproveitando-se de sua característica multiplataforma. Mesmo assim, entretanto, a carga da API é feita de forma dinâmica em todas as plataformas suportadas, o que exige o uso de ponteiro para função como dito anteriormente. Cada plataforma deve ser compilada separadamente.

Aderência aos Requisitos

Tabela 4.3 - Análise de Requisitos Quake III Engine

Requisito	Adere?	Observações
Modularidade	Parcialmente	Para APIs diferentes é necessário recompilar todo o código, mas mostra sinais de melhoria podendo o código que abstrai as APIs ser compilado em separado.
Extensibilidade	Parcialmente	A extensão se dá pela criação de um *branch* do código fonte base e alteração dos arquivos fonte que fazem a carga das bibliotecas dinâmicas.
Interface de uso comum	Sim	Funções abstraem a API base, necessitando apenas a inclusão de referência ao código (diretiva `#include` no C)
Global ao motor	Sim	Funções que abstraem a API acessam uma variável global que armazena uma referência para a API que está sendo utilizada.
Unicidade	Sim	Existe apenas uma referência à estrutura.

Como mostra a Tabela 4.3, a Quake III Engine basicamente manteve a estrutura da Quake II Engine, abolindo apenas o suporte a *rendering* por software, que, na época (2004), já se mostrava ineficiente, porque placas 3D com características avançadas já estavam disponíveis para o mercado consumidor.

4.4.2. enJine: Engine para Jogos Online em Java

O enJine é um projeto de motor de jogo baseado na linguagem Java, criado pelo Laboratório de Tecnologia Interativas (InterLab) da Universidade de São Paulo (NAKAMURA et al., 2003). A escolha da linguagem Java ao invés da tradicional linguagem C/C++ se deu pelos seguintes motivos:

- Suporte inerente a *multi thread* e comunicação de rede;
- Independência de plataforma;

- Utilização do modelo orientado a objetos, necessário para a abstração da camada da plataforma.

Atualmente, o enJine está sendo desenvolvido em conjunto com o jogo FootBot, em que o usuário cria robôs personalizados que jogam partidas entre si.

Especificação do Módulo de *Rendering*

Um dos requisitos da enJine é fornecer suporte a gráficos 3D através de uma camada que abstraia entidades de baixo nível e desacople o motor do jogo das bibliotecas de baixo nível. Neste caso, para o sistema de *rendering*, foi escolhido o uso do Java3D (SUN, 2005a).

A biblioteca Java3D provê um conjunto de classes para geração de ambiente 3D em tempo real, que permite seu uso em qualquer aplicação executada em uma plataforma que possua uma máquina virtual Java (*Java Virtual Machine* ou JVM)[11].

A Java3D utiliza para *rendering* a API disponível no sistema em que se encontra a aplicação desenvolvida, oferecendo suporte a *rendering* por OpenGL ou DirectX.

Para utilizar estas APIs, a Java3D utiliza o recurso da linguagem Java chamado JNI (*Java Native Interface*) (SUN, 2005b). O JNI permite que um código em Java, que rode em uma JVM, opere com aplicações e bibliotecas escritas em outras linguagens, como C/C++ e Assembly, assegurando que seu código Java seja, assim, portável para outras plataformas.

A Figura 4.14 mostra como a API Java3D faz a comunicação com os demais módulos.

[11] O Java3D não vem dentro do pacote de distribuição padrão da máquina virtual Java, necessitando *download* em separado.

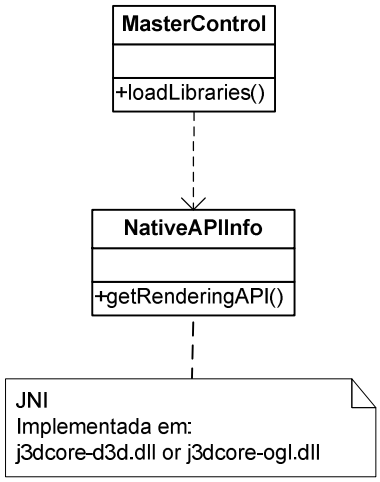

Figura 4.14 - Estrutura Interna do sistema de *rendering* de Java3D

Cada uma das APIs (DirectX e OpenGL) implementa um conjunto de classes utilizando o JNI, que é compilado, gerando a biblioteca dinâmica correspondente, a qual é carregada na chamada do método `loadLibraries` da classe `MasterControl`, conforme opção do sistema escolhido pelo usuário.

A Figura 4.15 mostra um exemplo de implementação JNI, no caso a classe `NativeAPIInfo` implementada pelo DirectX. Nesta figura, se mostra como é feita a ligação do método utilizado na linguagem Java (`NativeAPIInfo`) com o método na linguagem C (`JNICALL Java_javax_media_j3d_NativeAPIInfo_getRenderingAPI`).

```
#include <jni.h>
#include "javax_media_j3d_MasterControl.h"
#include "javax_media_j3d_NativeAPIInfo.h"

JNIEXPORT
jint JNICALL
Java_javax_media_j3d_NativeAPIInfo_getRenderingAPI(
    JNIEnv *env, jobject obj)
{
    return
(jint)javax_media_j3d_MasterControl_RENDER_DIRECT3D;
}
```

Figura 4.15 – Exemplo de implementação com JNI

A Figura 4.16 apresenta o diagrama de seqüência que mostra uma típica instanciação do sistema de *rendering*. A instância da classe `VirtualUniverse`, que representa o espaço 3D criado, ao ser ativada, chama o método `loadLibraries()` da classe `MasterControl`.

Esta, por sua vez, decide que API utilizar e consulta a classe `NativeAPIInfo` para obter as informações desta API. Uma vez recebida a informação de JNI, o pseudo-código destacado na Nota associada à `MasterControl` é executado, carregando a API escolhida.

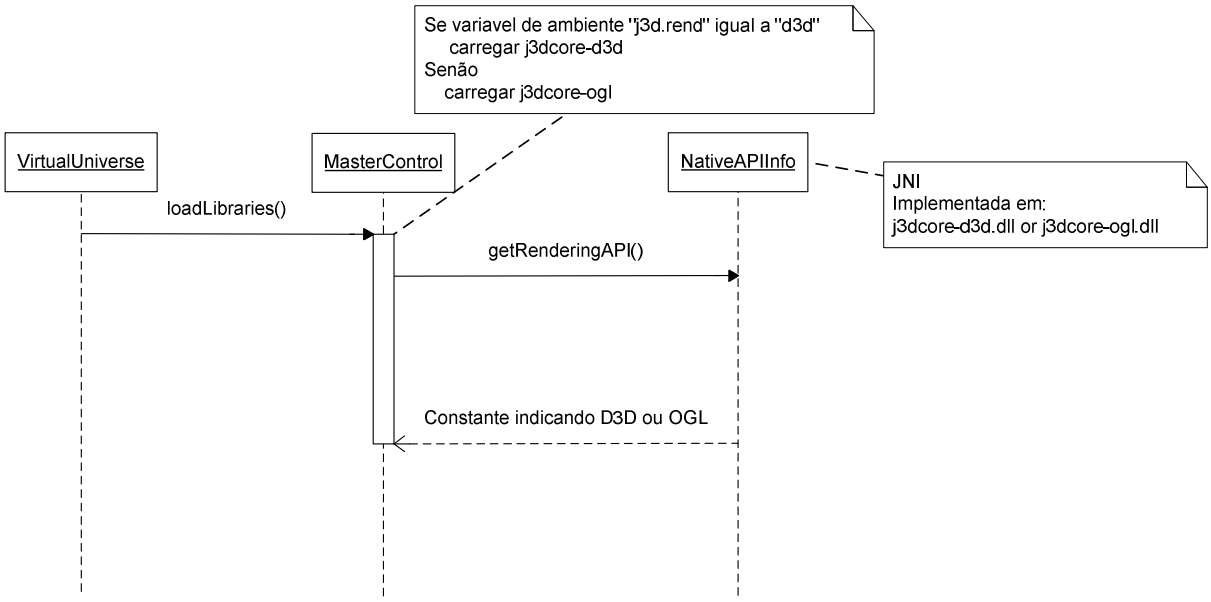

Figura 4.16 - Diagrama de sequência de instânciação do sistema de *rendering* para Java3D

A opção de selecionar uma API específica está apenas disponível em ambientes com o Sistema Operacional Windows, por causa do DirectX. Nos demais sistemas operacionais, utiliza-se como API a OpenGL.

Aderência aos Requisitos

Tabela 4.4 - Análise de Requisitos enJine (Java3D)

Requisito	Adere?	Observações
Modularidade	Parcialmente	Para APIs diferentes é necessário recompilar todo o código, mas o código que abstrai as APIs pode ser compilado em separado.
Extensibilidade	Parcialmente	A extensão se dá pela criação de um *branch* do código fonte base e alteração dos arquivos fontes que fazem a carga das bibliotecas dinâmicas.
Interface de uso comum	Sim	O usuário trabalha com classes como `Canvas3D` e `GraphicsContext3D`, que abstraem a implementação utilizada (OpenGL ou DirectX).
Global ao motor	Sim	A classe MasterControl recupera a API gráfica a ser utilizada.
Unicidade	Sim	A classe MasterControl mantém uma única instância da API.

Como mostra a Tabela 4.4, Java3D atende os requisitos de forma similar a Quake II Engine, onde ainda se mostra necessário recompilar o código da classe que faz a carga das

bibliotecas que abstraem a API gráfica utilizada. Java3D faz uso de diversos padrões apresentados no capítulo 4.3:

- *Singleton* é utilizado nas classes `VirtualUniverse` e `MasterControl`;
- *Factory Method* e *Dynamic Linkage* são usados pela `MasterControl` para a carga da bibliotecas;
- *Adapter* é utilizado, através de JNI, para adaptar classes utilizadas pelo usuário as implementações das APIs suportadas.

Uma observação interessante, é que o Java3D só não atende por completo os requisitos de Modularidade e Extensibilidade, por causa da forma que a classe `MasterControl` faz a carga das bibliotecas. Quando a classe `MasterControl` detecta que o Sistema Operacional é Windows, verifica qual o valor da variável de sistema `j3d.rend`, para decidir qual API utilizar; esta variável pode assumir apenas os valores `d3d` (DirectX) ou `ogl` (OpenGL), o que é fator limitante.

A Figura 4.17 mostra o bloco de código de decisão de qual API carregar. Note-se que o comentário dos próprios desenvolvedores mostra que o trecho deve ser alterado para suportar um processo mais dinâmico.

```
// If it is a Windows OS, we want to support
//dynamic native library selection (ogl, d3d)
String osName = System.getProperty("os.name");
if (osName != null && osName.startsWith("Windows")) {
   // XXXX : Should eventually support a more flexible dynamic
   // selection scheme via an API call.
   String str = System.getProperty("j3d.rend");
   if (str != null && str.equals("d3d")) {
      libName = d3dLibraryName;
   }
}
System.loadLibrary(libName);
return libName;
```

Figura 4.17 - Carga de biblioteca gráfica feita pela classe `MasterControl`

4.4.3. JFrog

O JFrog (BITTENCOURT et al., 2003a, 2003b, 2003c, 2004d, 2004) é um motor de jogo escrito em linguagem Java, que teve seu início de desenvolvimento na dissertação de mestrado em Ciência da Computação de João Ricardo Bittencourt, na Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS), e que, originalmente, se chamava Amphibian.

Este motor de jogo foi construído seguindo as práticas de programação orientada a objetos, utilizando diversos padrões de projeto, como *Singleton*, *Abstract Factory*, *Bridge*, *Adapter*, *Command*, *State*, *Observer*, *Layers* e *Model-View-Controller*. Além disso, o JFrog suporta aplicações *desktop* e dispositivos móveis, utilizando-se da API SuperWaba (SUPER WABA, 2005)..

Especificação do Módulo de *Rendering*

A Figura 4.18 mostra a estrutura interna do sistema de *rendering*.

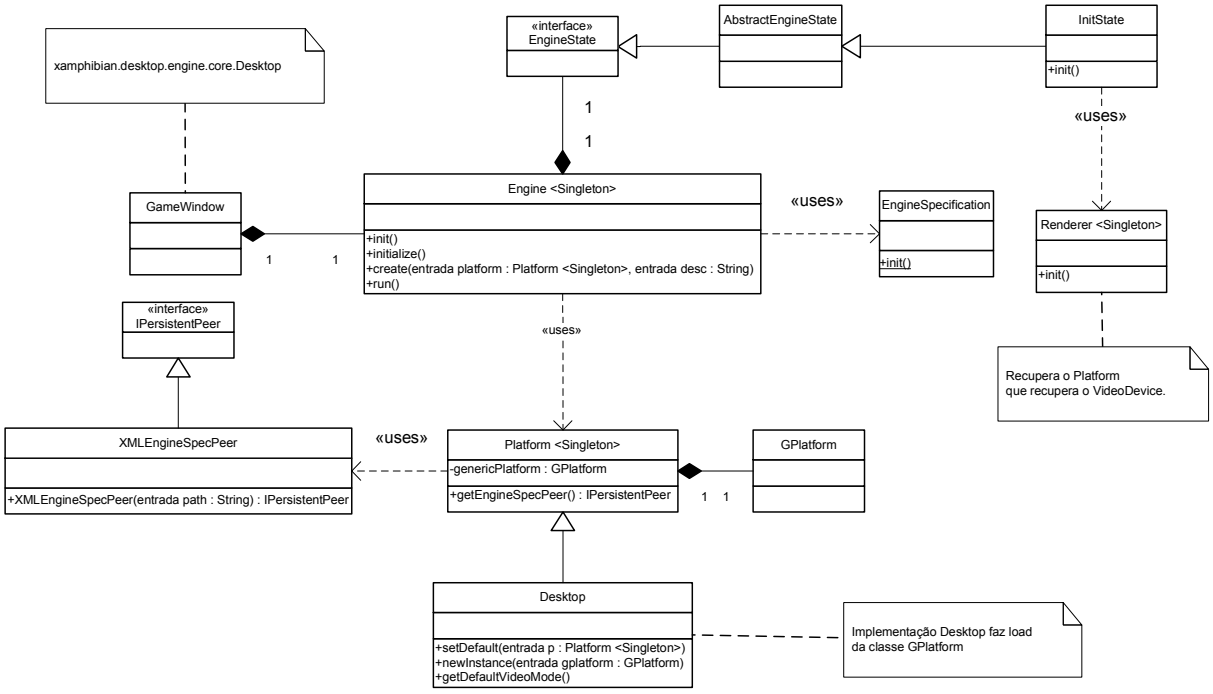

Figura 4.18 - Estrutura Interna do sistema de *rendering* de JFrog

Como se pode verificar, o JFrog possui um arquitetura complexa, dado que o objetivo do motor é o total desacoplamento das plataformas, possibilitando ao usuário a migração transparente entre estas. No caso do JFrog, abordam-se as plataforma *Desktop* e para dispositivos portáteis (celulares, *pocketpc*, etc.).

O controle do motor é feito por estados, sendo que cada novo estado deve generalizar a classe `AbstractEngineState.` Como exemplo, aponta-se que a classe `InitState` é responsável pela inicialização do motor e, com isso, pela instanciação do *renderer* a ser utilizado.

Nota-se, também, uma separação entre as classes que tratam da plataforma (*Desktop* ou *Mobile*) e as classes do motor, sendo que a classe `Engine` utiliza classes (classe `Platform`) que informam características da plataforma onde o motor esta sendo executado.

A Figura 4.19 mostra a troca de mensagens entre as classes do motor para iniciar o sistema de *rendering*.

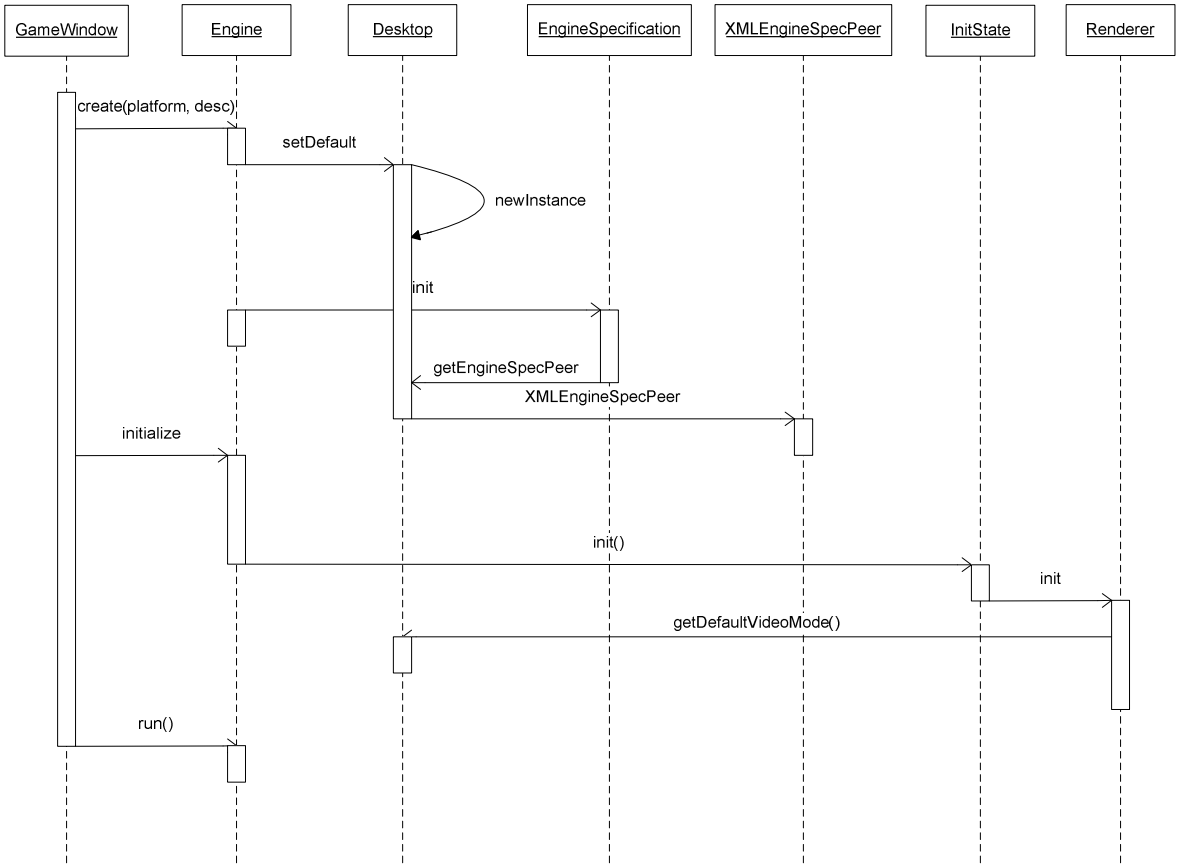

Figura 4.19 - Diagrama de sequência de iniciação do sistema de *rendering* em JFrog

Antes do sistema de *rendering* ser iniciado, o motor gráfico faz uma série de leituras de configuração de um arquivo no formato XML (classe `XMLEngineSpecPeer`) para descobrir que plataforma está sendo utilizada e qual classe de *render* deve ser carregada. Após esta carga inicial de configurações, finalmente o sistema de *render* é iniciado.

Aderência aos Requisitos

Tabela 4.5 - Análise de Requisitos JFrog

Requisito	Adere?	Observações
Modularidade	Sim	O usuário trabalha com interfaces e as classes de implementação são definidas através do arquivo de configuração.
Extensibilidade	Sim	As classes de implementação são definidas através do arquivo de configuração. Para trocar a API, basta criar uma nova classe seguindo as interfaces definidas e, depois, adicioná-la ao arquivo de configuração.
Interface de uso comum	Sim	O usuário trabalha com interfaces que abstraem as classes de implementação.
Global ao motor	Sim	A classe `Engine` recupera a API gráfica a ser utilizada.
Unicidade	Sim	A classe `Engine` mantém uma única instância da API.

Como mostra a Tabela 4.5, JFrog atende a todos os requisitos. Utilizando um arquivo de configuração em XML, os módulos que compõem o motor de jogo são definidos em tempo de execução, permitindo a abstração não apenas da API gráfica utilizada, mas também da plataforma em que se está executando o programa (*desktop*, *mobile*, etc.). O único problema encontrado durante a análise foi que este tipo de abstração exige diversas classes intermediárias. Então, ao tentar se visualizar um fluxo de execução, foi necessário se verificar diversas classes, o que ocupou um tempo considerável, e que, do ponto de vista de manutenibilidade e legibilidade, mostra-se complicado. Observou-se que a documentação estava incompleta e o código possuía poucos comentários, uma situação típica de muitos desenvolvimentos de projetos de software.

4.4.4. Forge V8 e ForgeV16

Forge V8 é um motor de jogo criado em 2001 por Charles Andryê Galvão Madeira, na Universidade Federal de Pernambuco. O desenvolvimento do motor de jogo teve como objetivo criar “um motor genérico e portável para jogos e aplicações multimídia, apresentando suporte a diversas características essenciais para estas aplicações como: interface gráfica, sonorização, inteligência artificial, modelagem física de personagens e multiusuários em rede, além de outras ferramentas tais como editores de cenários (MADEIRA, 2001)”. Além disso, o motor deveria suportar diversas plataformas e sistemas operacionais, bem como oferecer a possibilidade de construções de jogos 2D, 2D ½ e 3D.

O motor foi escrito em linguagem C++, dado que, em experiências passadas do autor com a linguagem Java, esta não se mostrou satisfatória nos requisitos de desempenho. O desenvolvimento seguiu as boas práticas de programação orientada a objetos e uso de padrões de projeto.

Como projetos desse porte são complexos no seu desenvolvimento, a primeira versão do motor gráfico tinha apenas suporte ao sistema operacional Windows, e utilizava a biblioteca DirectX para suporte a gráficos 2D.

Forge V16 é uma evolução de Forge V8 e foi desenvolvido na dissertação de mestrado de Eduardo Torres (TORRES, 2003) na Universidade Federal de Pernambuco, suportando apenas gráficos isométricos e sistema operacional Windows. Esta decisão se deu pelo fato de que o Forge V8, por ter tido como meta atender a um grande número de tipos de jogos, tecnologias e plataformas, gerou um projeto complexo. Forge V8 foi o primeiro projeto de uma equipe que estava se iniciando no desenvolvimento de motores de jogos, o que levou a uma solução com baixo desempenho e a problemas de integração de diversas tecnologias. Assim, a meta mais pretensiosa de Forge V8 foi abandonada em um primeiro momento.

Especificação do Módulo de *Rendering*

A Figura 4.20 mostra a hierarquia de classes de Forge V8, já que esta tenta ser o mais modular possível. Demonstra-se, nessa figura, como foi projetada a camada que abstrai a API Gráfica.

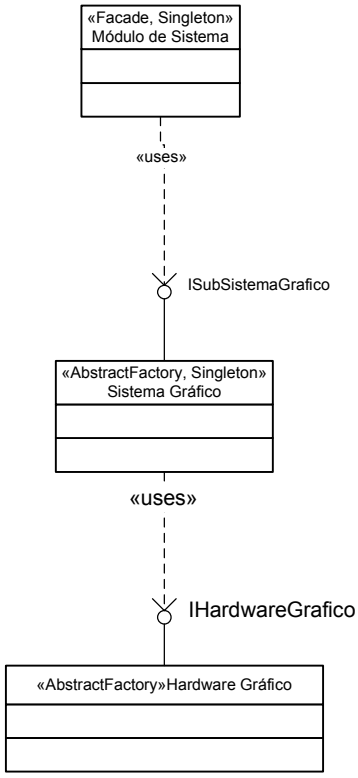

Figura 4.20 - Estrutura Interna do sistema de *rendering* de ForgeV8

Cada classe abstrai um nível do motor do jogo, sendo que o Módulo de Sistema, responsável por controlar todos os subsistemas do motor, "obtém" um Sistema Gráfico, que pode ser tanto 2D, 2D ½ ou 3D, através dos padrões *Abstract Factory* (instanciação do sistema de *rendering*) e *Singleton* (unicidade), e, por fim, obtém uma instância do hardware (API) gráfico.

Aderência aos Requisitos

Tabela 4.6 - Análise de Requisitos ForgeV8/V16

Requisito	Adere?	Observações
Modularidade	Não	O usuário trabalha com interfaces, mas o código dos módulos é compilado junto com o núcleo do sistema.
Extensibilidade	Parcialmente	A extensão se dá pela criação de novos arquivos fontes, que implementam as interfaces de *rendering*, sendo necessária a alteração do módulo (Sistema Gráfico) que faz a instanciação da API usada.
Interface de uso comum	Sim	O usuário trabalha com interfaces que abstraem as classes de implementação.
Global ao motor	Sim	O Sistema Gráfico é acessível através do Modulo de Sistema.
Unicidade	Sim	O Sistema Gráfico utiliza o padrão *Singleton*.

Como mostra a Tabela 4.6, ForgeV8 não atende ao requisito de Modularidade, porque os diversos módulos criados são compilados junto com o jogo em si, gerando um único módulo executável, ou é gerada uma única biblioteca dinâmica, com todos os módulos embutidos. Qualquer atualização no módulo de vídeo exige que um novo executável seja criado.

O requisito de Extensibilidade foi aceito parcialmente, pois, apesar do usuário trabalhar em cima de interfaces e o sistema de escolha de API ser baseado no padrão *Abstract Factory*, incluso no Sistema Gráfico na Figura 4.20, o acréscimo de uma nova API exige que se adicionem novos arquivos fontes que implementem a interface `IHardwareGrafico`. Também é necessário adicionar-se essa nova opção ao módulo Sistema Gráfico e recompilar tudo junto para gerar o executável. A aplicação do padrão *Dynamic Linkage* em conjunto com o *Abstract Factory* minimizaria esse número de recompilações.

4.4.5. OGRE3D

OGRE3D (*Object-Oriented Graphics Rendering Engine*) é um motor 3D flexível, orientado à cena, escrito em C++, e projetado para facilitar o desenvolvimento de aplicações que utilizam aceleração de gráficos 3D por hardware. A biblioteca de classes abstrai os sistemas de baixo nível, como Direct3D e OpenGL, e provê interfaces baseadas em objetos do mundo e outras classes intuitivas (OGRE3D, 2005a).

Trata-se apenas de um motor gráfico, isto é, não inclui os outros módulos, como os que constituem o sistema de IA, Física, Rede etc. Esta decisão foi tomada para se obter um motor gráfico que possa ser usado em outras aplicações além de jogos, como aplicações de negócios e simuladores. Sua inclusão na análise se deu pelo fato de ser utilizado como motor gráfico de diversos motores de jogos (OGRE3D, 2005b).

O OGRE3D possui uma versão para os sistema operacionais Windows, Linux e MacOS, suportando as APIs OpenGL e DirectX, estando a última apenas disponível na plataforma Windows.

Especificação do Módulo de *Rendering*

A Figura 4.21 mostra estrutura interna que abstrai a API gráfica.

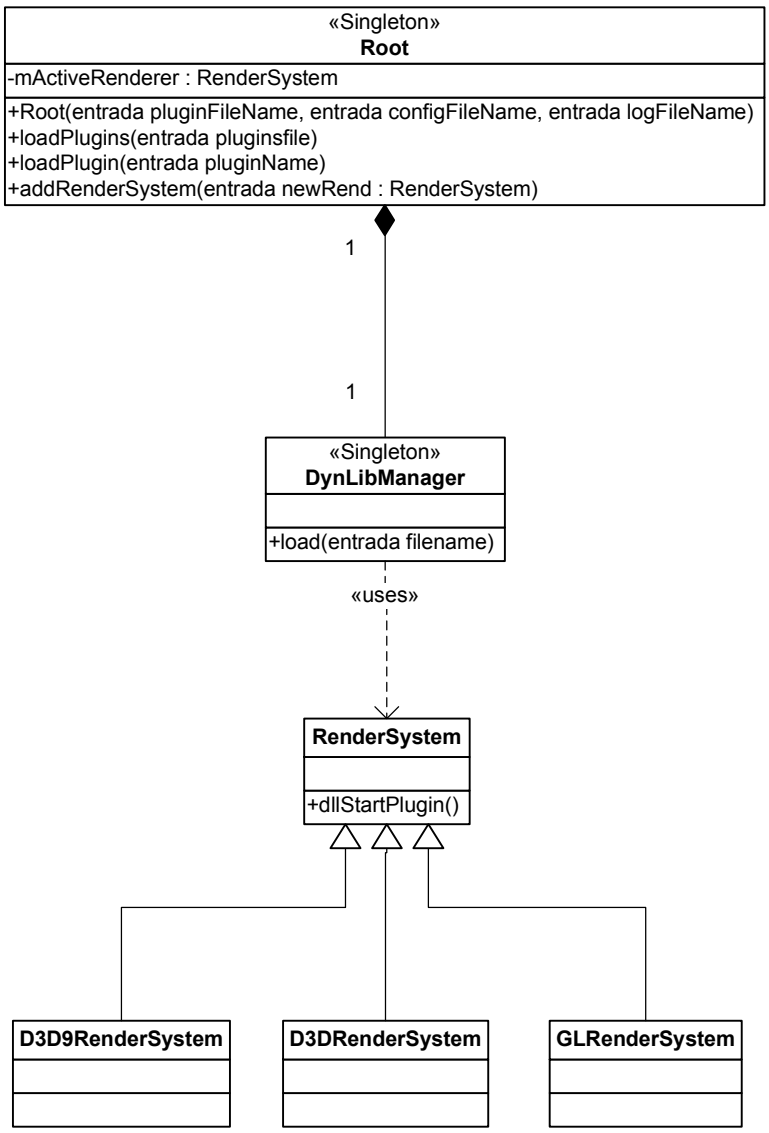

Figura 4.21 - Estrutura Interna do sistema de *rendering* de OGRE3D

Nesta figura, nota-se, mais uma vez, que, para sistemas operacionais diferentes é necessária a recompilação dos módulos. A figura destaca também o uso de padrões de projeto, como *Singleton*, para gerenciar a abstração da API sendo utilizada.

Outro ponto que se pode notar é a separação das versões do Direct3D, havendo uma classe para Direct3D versão 9 e outra para a versão 7 (e inferiores). Isto ocorre por causa da integração das linguagens de *Vertex* e *Pixel Shaders*, introduzidas na versão 8.1 e inteiramente integradas na versão 9. Também se nota que o `RenderSystem` não segue o padrão de projeto *Singleton*, porque o OGRE3D permite que se possa escolher dinamicamente entre o OpenGL ou DirectX, em tempo de execução.

Para uma melhor visualização, a Figura 4.22 mostra a ordem de chamada para instanciação da API gráfica.

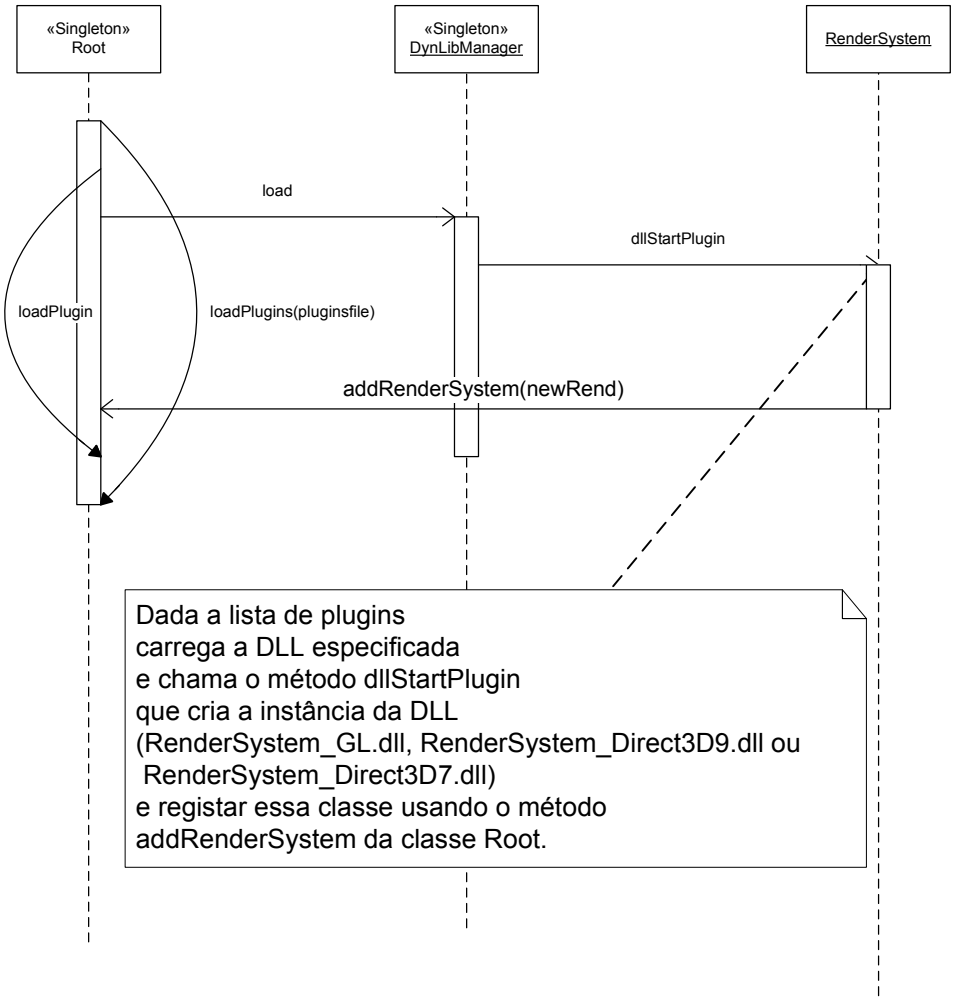

Figura 4.22 - Diagrama de seqüência de instanciação da API gráfica em OGRE3D

Aderência aos Requisitos

Tabela 4.7- Análise de Requisitos OGRE3D

Requisito	Adere?	Observações
Modularidade	Sim	O usuário trabalha com interfaces e as diversas APIs estão separadas em bibliotecas (uma biblioteca para cada API).
Extensibilidade	Sim	Se necessário criar a nova biblioteca, seguindo as interfaces definidas, basta adicioná-la no arquivo de configurações.
Interface de uso comum	Sim	O usuário trabalha com interfaces que abstraem as classes de implementação.
Global ao motor	Sim	Através de um ponto único, classe `Root`, pode-se obter o sistema de *rendering*.
Unicidade	Sim	A classe `Root` foi criada seguindo o padrão *Singleton*.

Como mostra a Tabela 4.7, OGRE3D atende a todos os requisitos e faz uso dos padrões de projetos sugeridos no capitulo 4.3.

Um dos pontos que chama atenção foi a forma como é feita a instanciação das APIs: através de um arquivo de configuração, em formato texto, o usuário diz quais bibliotecas devem ser carregadas (DirectX9, DirectX7, OpenGL, etc.) e, durante a carga, a instância da classe `Root`, classe gerenciadora, é passada para cada DLL, que a usa para se auto-registrar na lista de *renderers* disponíveis. Isso permite que se adicionem outros *renderers*, além de facilitar a manutenção, já que cada API pode ser alterada sem a necessidade de recompilar as demais APIs e o núcleo do motor.

4.4.6. Wild Magic Engine

Wild Magic Engine é um motor de jogo criado por David Eberly e que é descrito em seus livros (EBERLY, 2000, 2004). A versão mais atual é Wild Magic Engine 3.

Por ser totalmente descrito nos livros, esse motor de jogo se mostra um ótimo ponto de partida para pessoas que querem se iniciar no desenvolvimento de motores de jogos.

Wild Magic Engine foi escrito em C++ e desenvolvido seguindo os princípios de programação orientada a objetos, podendo ser usado nos sistemas operacionais Windows, Linux e MacOS.

Especificação do Módulo de *Rendering*

Na parte de *rendering*, esse motor de jogos define interfaces entre o corpo do motor e o sistema de *rendering*, implementadas para as API desejadas. No caso, as APIs implementadas são a DirectX9 e a OpenGL, mas outras APIs podem ser incluídas.

Outro fator a mencionar é que esse motor de jogos faz uso intensivo de instruções do pré-compilador C++ (meta-programação).

A Figura 4.23 mostra essa organização.

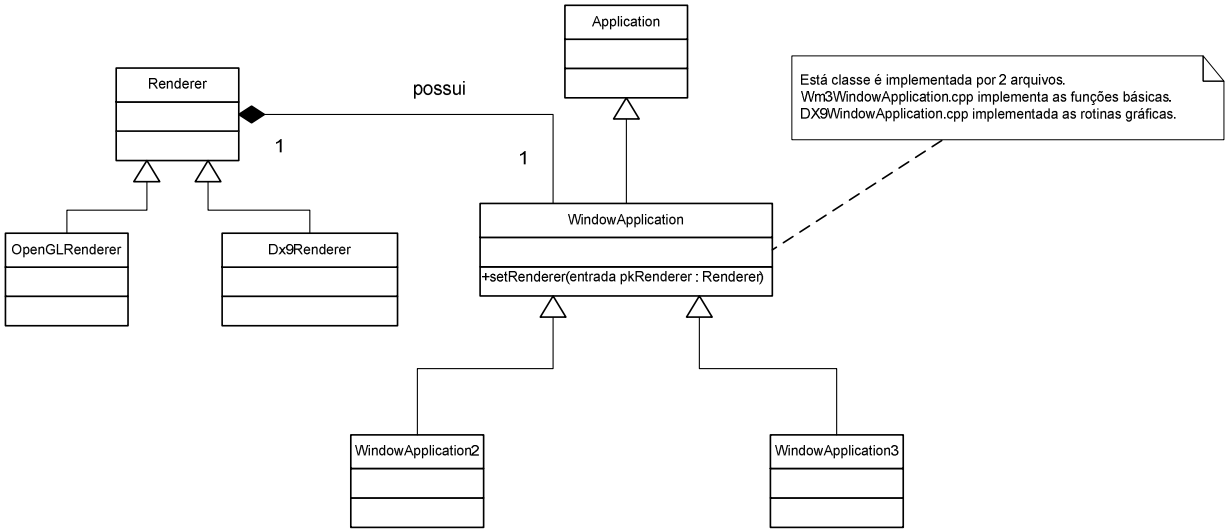

Figura 4.23 - Estrutura Interna do sistema de *rendering* de Wild Magic Engine

Uma aplicação que deseje implementar suporte a Wild Magic Engine necessita apenas estender uma das classes de aplicação (WindowApplication2 ou WindowApplication3), adicionando os demais componentes necessários como grafo de cena, gerenciador de som, gerenciador de física etc. O diagrama de seqüência foi omitido, porque a arquitetura do motor é baseada em generalização simples. Sendo assim, esse diagrama não agrega informação significativa ao diagrama da Figura 4.23.

Aderência aos Requisitos

Tabela 4.8 - Análise de Requisitos Wild Magic Engine

Requisito	Adere?	Observações
Modularidade	Não	O usuário trabalha com interfaces, mas o código dos módulos é compilado junto com o núcleo do sistema.
Extensibilidade	Parcialmente	A extensão se dá pela criação de um *branch* do código fonte-base e criação de novas classes para a API que se deseje implementar, bem como uma classe `WindowApplication` especifica para essa API..
Interface de uso comum	Sim	O usuário trabalha em cima de interfaces que abstraem as classes de implementação.
Global ao motor	Sim	A classe `WindowApplication` controla todo o sistema.
Unicidade	Parcialmente	A classe `WindowApplication` é que faz o controle de quantos *renderers* estão ativos e não a própria classe do *renderer*.

Como mostra a Tabela 4.8, o Wild Magic Engine segue o sistema de generalização simples, não usando características avançadas vistas nos outros motores analisados, como, carga dinâmica de biblioteca, uso de arquivos de configuração, etc.. Para cada API nova ou alteração feita nas APIs antigas, é necessário que uma nova compilação seja executada.

Um ponto a ser mencionado é a aplicação do padrão *Singleton*. O controle da instância do *renderer* é feita pela classe da aplicação (`WindowApplication`), ou seja, o usuário pode criar diversas instâncias da classe de *renderer*, sem controle por parte do *renderer*.

Como a Wild Magic Engine é um motor desenvolvido para fins didáticos e com foco no desenvolvedor iniciante, a falta desses recursos avançados não é grave. Como o motor é bem estruturado, a aplicação dos padrões *Singleton* (controle feito pela classe *renderer*), *Factory Method* e *Dynamic Linkage*, para obtenção destes recursos, não exigiria um grande esforço, caso se optasse por sua introdução.

4.4.7. X-Engine

O motor de jogo criado por Martin Ecker na sua dissertação de mestrado na Johannes Kepler Universität (Ecker, 2003) foi projetado segundo a orientação a objetos, utilização de

multiplataforma (Windows e Linux) e, ainda, para ter independência de API, suporte à programação de GPU, ser modulável e extensível.

Este motor foi escrito em linguagem C++ e utiliza diversos *middleware* de código aberto, como WxWidgets, STLPort, Boost, Spirit, DevIL, lib3ds e FFTW, para ajudar a constituir a infra-estrutura do projeto. Com isso, o projeto do motor focou apenas a área de interesse específico, ou seja, o sistema de *rendering*.

O sistema de *rendering* do X-Engine possui suporte a OpenGL 1.3, DirectX 8.1 e DirectX 9 e utiliza diversos padrões de projeto, como *Abstract Factory* e *Template*.

A Figura 4.24 mostra a organização interna do X-Engine.

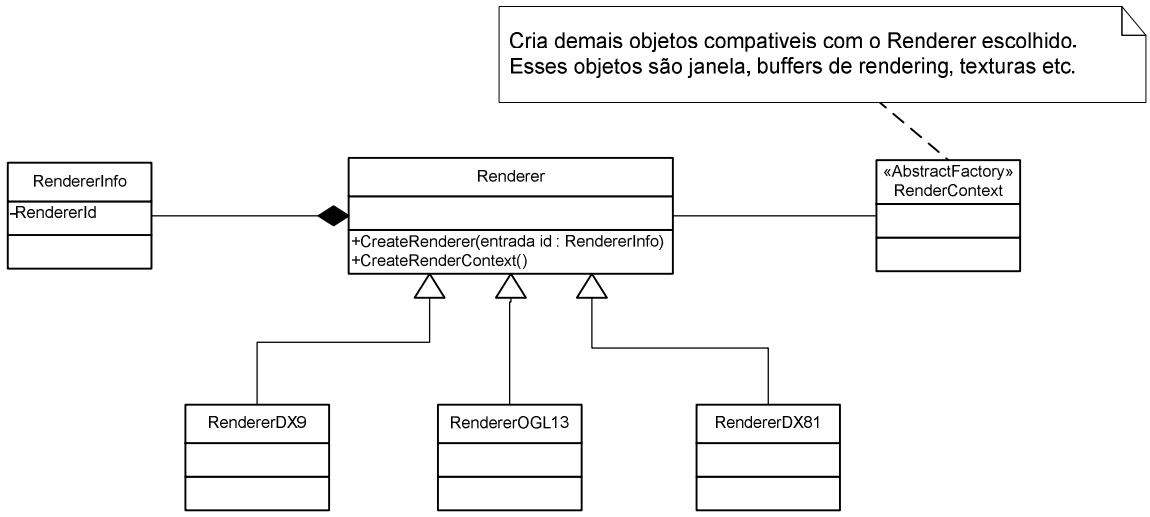

Figura 4.24 - Estrutura Interna do sistema de *rendering* de X-Engine

Como se pode observar, a classe base `Renderer` é especializada nas API desejadas (DirectX 8.1, DirectX 9 e OpenGL 1.3), sendo que cada classe deve fornecer uma implementação da classe `RenderContext`, que construirá os demais objetos utilizados pela API. A classe `RenderInfo` é um objeto que contém informação sobre a API (nome, identificador único).

Um fator interessante do X-Engine é o uso que este faz de *middlewares*, dentre os quais se encontram:

- wxWidgets: abstrai o sistema de janelas utilizado pelo software. Assim, ao escrever uma aplicação que utiliza o wxWidgets, necessita-se apenas recompilar o programa para a plataforma específica da aplicação. Além disso, o wxWidgets possui diversos utilitários, como gerenciador de *plugins*, que faz o controle de carga e liberação de bibliotecas dinâmicas. Isso é usado no X-Engine para controlar as bibliotecas das APIs.
- A biblioteca *Boost*: possui diversos utilitários para aplicações escritas em C++, como gerenciador de memória e *Smart Pointers*. Este último recurso é uma técnica utilizada em programas escritos em C/C++ para evitar *dangling pointers* e *memory leaks*, que ocorrem quando são utilizadas variáveis de ponteiro compartilhadas entre diversos objetos.

Para abstrair as diversas linguagens de *Shader*, o X-Engine fez uso das ferramentas de compilação flex++ (GNU, 2006a), gerador léxico e bison++ (GNU, 2006b), e gerador de *parsers* para geração de classes que fazem o *parser* dos arquivos fontes dessas linguagens.

O diagrama de seqüência da instânciação da API gráfica é omitido, pois este é similar ao da OGRE3D, não agregando informação significativa.

Aderência aos Requisitos

Tabela 4.9 - Análise de Requisitos X-Engine

Requisito	Adere?	Observações
Modularidade	Sim	O usuário trabalha com interface, e as APIs gráficas estão divididas em bibliotecas.
Extensibilidade	Sim	Se necessário criar a nova biblioteca, seguindo as interfaces definidas, basta adicioná-la no arquivo de configurações
Interface de uso comum	Sim	O usuário trabalha em cima de interfaces que abstraem as classes de implementação.
Global ao motor	Sim	A classe RenderContext controla todo o sistema de rendering.
Unicidade	Sim	A classe RenderContext segue o padrão *Singleton* e fornece acesso a criação de outros objetos referentes ao renderer escolhido.

Conforme mostra a Tabela 4.9, o X-Engine atende a todos os requisitos, seguindo um sistema similar ao da OGRE3D. Uma das características interessantes observadas foi com

relação à classe `RenderContext`, que representa um *rendering* associado a uma API, que segue os padrões de projeto *Singleton* e *Abstract Factory*. Através desta, são obtidos os demais objetos do sistema, como texturas, luzes, mechas, etc., garantindo, assim, que o usuário trabalhe apenas com objetos referentes à API em uso.

Outra característica interessante é o uso de *parsers* proprietários para abstrair as linguagens de *Shader*, permitindo que os efeitos escritos em uma linguagem, Direct3D ou OpenGL, sejam migrados para a outra sem alterações.

4.5. Conclusões

Como pôde ser observado neste capítulo, ocorreu uma grande evolução na criação dos motores de jogos ao longo dos anos, sendo que preocupações com modularidade e evolução se tornaram cada vez mais presentes nos projetos.

No capítulo 5, serão resumidos os resultados obtidos neste levantamento, realizando-se, ainda, observações específicas sobre cada motor.

5. Resultados

A seguir estão relacionados os principais resultados obtidos através da análise dos motores de jogos estudados no capítulo 4, organizados de acordo com os seguintes aspectos: requisitos relacionados à arquitetura de jogos, utilização de padrões de projeto e documentação da arquitetura. Aspectos sobre multiplataformas são também enfocados. Finalizando o capítulo apresenta-se a proposta de um padrão para desacoplamento entre o motor gráfico e a API, o *Graphical API Layer Decoupling Pattern*.

5.1. Requisitos relacionados à arquitetura

No Capítulo 4, os motores de jogos foram discutidos de acordo com alguns requisitos de arquitetura considerados fundamentais, a saber: independência funcional, extensibilidade, interface de uso comum, acessibilidade global ao motor e unicidade do módulo gráfico.

A Tabela 5.1. mostra a aderência dos motores analisados aos requisitos de arquitetura colocados. Nessa tabela, pode-se notar que os requisitos são satisfeitos, ainda que parcialmente, pela maioria dos motores de jogos, conforme os objetivos de cada um e sua época de lançamento.

Tabela 5.1 - Aderência dos motores aos requisitos colocados

Motor /requisito	Modularidade	Extensibilidade	Interface de uso comum	Global ao motor	Unicidade
Quake I	Parcialmente	Parcialmente	Sim	Sim	Sim
Quake II	Parcialmente	Parcialmente	Sim	Sim	Sim
Quake III	Parcialmente	Parcialmente	Sim	Sim	Sim
enJine(Java3D)	Parcialmente	Parcialmente	Sim	Sim	Sim
JFrog	Sim	Sim	Sim	Sim	Sim
ForgeV8/V16	Não	Parcialmente	Sim	Sim	Sim
OGRE3D	Sim	Sim	Sim	Sim	Sim
Wild Magic	Não	Parcialmente	Sim	Sim	Parcialmente
X-Engine	Sim	Sim	Sim	Sim	Sim

A extensibilidade e a modularidade são os requisitos menos atendidos. Entretanto, na família Quake, esses requisitos foram perseguidos nas versões mais novas dos motores. Na última versão, observou-se que apenas a biblioteca OpenGL foi utilizada, devido ao seu suporte multiplataforma.

Os motores da série Quake também evidenciaram a busca pela modularização e extensibilidade dos módulos de um motor de jogo com o passar do tempo. Na sua primeira versão, Quake I, observou-se que o motor era totalmente acoplado, sendo necessária a recompilação de todo o código para gerar versões distintas do aplicativo para *rendering* por software ou OpenGL. Na segunda versão do motor, o módulo gráfico foi separado em bibliotecas que se comunicavam com o resto do motor através de um conjunto de funções (variáveis do tipo “ponteiro para função”). E, na última versão, como mencionado no parágrafo anterior, optou-se pela OpenGL. Também, neste caso, foi introduzida uma camada que permitiu o desacoplamento do sistema operacional, isto é, para migrar o código de *rendering*, bastaria apenas implementar um conjunto de arquivos que especificassem as bibliotecas a serem utilizadas através de “ponteiros para função”.

O enJine e o JFrog, escritos em Java, fazem uso dos recursos desta linguagem para *rendering*. O enJine é acoplado ao Java3D, que, por sua vez, utiliza o recurso JNI para fazer a comunicação com as diversas APIs que este suporta. O JFrog é mais genérico, criando uma camada na própria linguagem Java, permitindo a variação do módulo gráfico utilizado, o que permitiria também modificar o componente usado, como, por exemplo, variar entre Java3D ou JOGL (SUN MICROSYSTEMS INC., 2005).

Os motores OGRE3D e X-Engine são os que mostraram maior preocupação com o desacoplamento do módulo gráfico, já que a modularidade é o foco de cada um destes motores. Neles se encontra a melhor arquitetura no sentido de modularidade e extensibilidade.

5.2. Utilização de Padrões de Projeto

A mostra um comparativo entre os motores de jogos com relação ao uso dos padrões de projeto, considerando os padrões mais comumente encontrados na literatura.

Tabela 5.2 - Motor x Padrão de Projeto.

Motor /Padrão de Projeto	***Singleton***	***Factory Method***	***Dynamic Linkage***	***Adapter***
Quake I	Sim	Sim	Não	Sim
Quake II	Sim	Sim	Sim	Sim
Quake III	Sim	Sim	Sim	Sim
Java3D	Sim	Sim	Sim	Sim
JFrog	Sim	Sim	Sim	Sim
ForgeV8	Sim	Sim	Não	Sim
Ogre3D	Sim	Sim	Sim	Sim
Wild Magic	Não	Não	Não	Sim
X-Engine	Sim	Sim	Sim	Sim

Como se vê, todos os motores utilizam padrões de projeto. No entanto, constatou-se que alguns não empregavam os padrões selecionados inicialmente, sendo, ainda, que alguns dos motores utilizam outros padrões além dos indicados na tabela acima.

A utilização de padrões atende às definições de padrões de projeto e *framework*s (Madeira, 2001): “padrões de projeto fazem mais fácil o sucesso do reuso de projetos e arquiteturas, consequentemente, tornando mais fácil o desenvolvimento de novos sistemas. Nesse contexto, se inserem os *frameworks*”, ou seja, os padrões de projetos são a base para a concepção de bons *frameworks.* Assim sendo, são também a base para concepção de bons motores de jogos.

Apesar dos motores da série Quake serem escritos na linguagem C, uma linguagem estruturada, justifica-se a resposta “Sim” em alguns padrões de projetos no fato de que, como mostrado em Eide (2002), os padrões de projeto também podem ser aplicados a linguagens estruturadas.

Os motores ForgeV8 e ForgeV16 mostraram o projeto de um *framework* com foco em padrões, possuindo, além do módulo gráfico, outros padrões para os demais módulos. Porém, o que mais chamou a atenção nos projetos, foi o relato de dificuldades de projeto e decisões que devem ser tomadas no momento do desenvolvimento, como a complexidade de criação do motor por desenvolvedores que estão há pouco tempo nessa área e a complexidade em dar suporte às diversas APIs, em razão das diferenças encontradas entre as estruturas de dados, como, por exemplo, nas coordenadas de textura.

Após a análise, constatou-se que existe uma certa tendência no uso de padrões de projeto na concepção do módulo gráfico e, possivelmente, também em outros módulos do motor que fazem uso de APIs. Para abstração do módulo gráfico, usou-se uma combinação dos padrões *Singleton*, *Factory Method*, *Adapter* e *Dynamic Linkage* Estes mesmos padrões poderiam ser usados para, por exemplo, abstrair o módulo de som, onde se tem o DirectSound (DirectX) e o OpenAL, que acessam o hardware de som e exportam uma série de funções para seu uso.

A aplicação destes padrões, voltada ao aspecto de abstração de *middleware*, segue outro padrão de projeto chamado *Layers* (BUSCHMANN, 1996), um padrão de arquitetura que estrutura aplicações, de forma que possam ser descompostas em subgrupos de tarefas, onde cada subgrupo está em um nível particular de abstração. Outro exemplo de aplicação deste padrão é o modelo de três camadas, que divide uma aplicação em apresentação, modelo de dados e negócio.

O Wild Magic Engine mostrou-se o motor que faz menos uso de padrões de projeto na sua concepção. Apesar disso, como o motor foi bem estruturado, a aplicação de padrões como *Singleton*, *Dynamic Linkage* e *Factory Method* exigiria poucas alterações.

5.3. Documentação da arquitetura dos motores gráficos

Apesar dos motores analisados utilizarem arquiteturas elaboradas para solucionar o problema da abstração da API gráfica, notou-se um problema com relação à documentação de alguns deles, pois, muitas vezes, o usuário destes motores necessita passar horas analisando o código, com pouco ou nenhum comentário e, ainda, depender de fóruns para encontrar alguma documentação significativa.

O Wild Magic, por exemplo, apesar de não utilizar muitos padrões, é o motor que tem uma das melhores documentações, já que sua arquitetura é descrita em livro por Eberly (2004).

Uma comparação interessante entre documentações se dá com o JFrog e Ogre3D: no primeiro, a documentação automática (elaborada através de Javadoc) apenas apresenta uma listagem de classes, com quase nenhum comentário significativo sobre qual a responsabilidade da classe ou a função de seus métodos. Já no Ogre3D, a documentação gerada automaticamente é rica, com uma descrição das responsabilidades da classe e função de seus métodos. Apesar disso, o JFrog, como um motor concebido no meio acadêmico, foi descrito em uma coleção de artigos de congressos, que dão uma visão geral da elaboração de sua arquitetura (BITTENCOURT, 2004, 2003a, 2003b, 2003c, 2003d).

Os motores da série Quake também possuem documentação precária, pelo menos na versão liberada para uso livre, contendo poucos comentários em trechos de código. Mais uma vez, o usuário da versão de código aberto destes motores depende das comunidades criadas em volta dele.

Os demais motores possuem uma boa documentação de arquitetura, uma vez que são descritos em dissertações de mestrado. Ao longo do código, podem existir, ou não,

comentários, com qualidade variável, sendo que, muitas vezes, blocos de código não possuem nenhum comentário.

5.4. Considerações sobre multiplataforma

Atender a diversas plataformas vem se tornando essencial para a área de jogos, dada a diversidade de plataformas que existem atualmente, como GameCube, Xbox, Playstation 2, Windows, Linux e Macintosh. Os altos custos para se desenvolver um novo motor de jogos, aliados aos requisitos de multiplataforma, acabam por tornar o processo de reutilização um requisito essencial para o desenvolvimento dos jogos atuais.

Dentre os motores analisados, muitos foram projetados para serem multiplataforma, atingindo este ponto de diversas maneiras, como se descreve a seguir:

- EnJine e JFrog utilizam a linguagem Java para criação dos seus motores. Assim, qualquer máquina que possua uma máquina virtual Java pode processá-los. O único inconveniente disso é que, geralmente, vídeo-games, possuem SDK próprios para desenvolvimento e recursos limitados se comparados a um computador pessoal.
- Forge V8 e Forge V16 foram escritos tendo em vista a plataforma Windows. Para migrarem para plataformas Linux ou Macintosh, exigiriam certa quantidade de esforço.
- Os motores Ogre3D, Quake e Wild Magic foram escritos em C/C++, onde o núcleo do motor, funções e classes utilizadas por quem desenvolve o jogo foram escritos seguindo a especificação da linguagem, o que permite que o programa seja recompilado em outra plataforma que possua um compilador que siga este padrão, como nas plataformas mencionadas acima.
- O X-Engine segue o mesmo processo adotado pelos motores Ogre3D, Quake e WildMagic, mas, diferentemente dos demais, fez uso de bibliotecas de terceiros, que

também são multiplataforma, como, por exemplo, WxWidgets (WXWIDGETS, 2005), que abstrai o sistema de janelas utilizado pelo desenvolvedor e possui conjuntos de classes utilitárias para gerenciamento de memória e conversão de tipos de dados. Isto possibilita desenvolvimento mais ágil do motor, permitindo que o desenvolvedor se foque no problema principal, que é o motor gráfico para jogos.

Em todos os casos, pode-se notar que os padrões de projeto colaboram para essa abstração de plataforma, desde que sua aplicação não utilize aspectos específicos da plataforma ou do sistema operacional. No caso do *Factory Method*, por exemplo, este não deve retornar objetos de tipos específicos do sistema operational Windows, mas sim objetos definidos pelo desenvolvedor, que utilize tipos básicos, em concordância com o padrão ANSI definido para a linguagem (no caso, a linguagem C++).

Outros cuidados que devem ser levados em consideração estão mostrados na Tabela 5.3 e estão relacionados à utilização da linguagem C++:

- Uso do pré-processador: para definir pacotes de bibliotecas, definições e propriedades diferentes durante a compilação do código, utilizam-se os comandos do pré-processador C (`#define, #ifdef` etc.), que permitem que, em tempo de compilação, se faça a geração correta.
- Tipos de variáveis: devem ser definidos tipos próprios de variáveis básicas a serem usadas durante a construção do motor (`int, float, double, char, byte` etc.), porque, em plataformas diferentes, os tipos básicos de C/C++ podem assumir tamanhos diferentes. Por exemplo, `int` pode ter 32 ou 64 bits. Num compilador da Microsoft, são definidos diversos tipos extras como `float64` e `int64,` que podem não existir em outros compiladores.
- *Branches*: é uma característica comum em sistemas de controle de versão de software. Realizar *branches* é gerar um novo projeto baseado nas fontes de uma versão já

existente, permitindo que ambas as versões evoluam separadamente. Caso no futuro se decida incorporar características de uma versão à outra, o software de controle de versão permite, através de um comando, que se faça o *merge* entre essas versões. O uso de *branches* permite que se tenha versões para plataformas diferentes sendo desenvolvidas em paralelo e dividindo um conjunto de arquivos em comum, como, por exemplo, as interfaces de classes e classes abstratas. Esse tipo de desenvolvimento depende, muitas vezes, de instruções do pré-processador C.

- *Endianness*: é a forma como um processador organiza mais de um byte em memória. *Big Endian* (Intel x86) é o byte mais significativo, que é armazenado na posição de memória com menor endereço, enquanto *Little Endian* (IBM PowerPC), o byte menos significativo, que é armazenado na posição de memória de menor endereço. Durante o processo de migração de uma aplicação entre plataformas diferentes, deve-se verificar como é feito o armazenamento de bytes, de forma que a conversão de dados entre essas plataformas seja feito corretamente. Exemplos de aplicação desse tipo de dado são os arquivos para salvamento de jogos e a comunicação de dados via rede.
- Temporizador: toda plataforma possui um temporizador que é usado para realizar cálculos com horários. A forma de acesso a esse temporizador varia conforme a plataforma (qual instrução *Assembly* utilizar) e sistema operacional (qual função do sistema operacional produz melhor resultado). Como o temporizador controla diversos aspectos de um motor de jogo (tempo de animação, tempo de envio de dados do servidor ao cliente, etc.), é importante que este item seja analisado com cuidado.
- Funções do sistema operacional/plataforma: deve-se abstrair totalmente chamadas a módulos como: gerenciadores de janelas, arquivos mapeados em memória e qualquer outra característica específica do sistema operacional, já que estes aspectos diferem conforme a plataforma. Por exemplo, o Linux possui diversos pacotes de

gerenciamento de janelas utilizando a biblioteca QT, a biblioteca GTK ou chamadas diretas ao servidor X11.

5.5. Proposta de um padrão de projeto

Como exposto no tópico 5.2, notou-se que é utilizada uma mesma estrutura de classes para abstrair a API gráfica nos diversos motores de jogos analisados. Com base nessa construção, foi definido o padrão de projeto *Graphical API Layer Decoupling Pattern*, cuja descrição segue o modelo apresentado na Figura 3.1, que nada mais é do que a forma utilizada por Gamma (2000). A Tabela 5.3 mostra os padrões que fazem parte dessa solução.

Tabela 5.3 - Padrões de Projeto da solução.

Graphical API Layer Decoupling Pattern	Abstrair a API gráfica a ser utilizada pelo motor do jogo.
Singleton	Garante que uma classe possua apenas uma instância e fornece um ponto de acesso global a ela.
Factory Method	Define uma interface para criar um objeto, mas deixa as subclasses decidirem qual objeto a ser instanciado.
Dynamic Linkage	Permite um programa, dada uma requisição, que carregue e use classes arbitrárias que implementam uma interface conhecida.
Adapter	Converte a interface de uma classe em outra interface esperada pelos clientes.

5.5.1. *Graphical API Layer Decoupling Pattern*

5.5.1.1. Intenção

Desacoplar o módulo gráfico de um motor de jogo da API gráfica usada.

5.5.1.2. Motivação

O processo de desenvolvimento de um jogo pode levar de dezoito meses até cinco anos para ser concluído (Fristom, 2003) e é muito provável que os subsistemas que o compõem venham a sofrer mudanças neste período devido a inúmeros e diferentes fatores. O subsistema de *rendering* pode ser alterado, quer seja pela adição de novas potencialidades ao software gráfico, quer seja pela necessidade de compatibilizar-se com um novo software ou hardware.

Dessa forma, o projeto deve levar em conta tal possibilidade, incorporando características que tornem os vários subsistemas que o compõem adaptáveis, facilitando, assim, a manutenção.

A existência de diversas APIs gráficas, como OpenGL e DirectX, traz alguns problemas:

- Como gerar um sistema que atenda às diversas APIs do mercado e suas diferentes versões sem causar impacto nos demais subsistemas?
- Como liberar o projetista de jogos da difícil tarefa de adquirir conhecimento profundo sobre os objetos ou funções que compõem uma determinada API gráfica, bem como de suas interfaces e estruturas internas? Desenvolvedores do jogo não precisariam saber, a princípio, todas as APIs que o motor deve suportar nem podem prever o que virá no futuro.

Estes problemas geram algumas necessidades durante o projeto do subsistema de *rendering* do motor do jogo:

- Uma interface comum única auxilia na reutilização de código;
- A abstração da API permite que desenvolvedores, não familiarizados com uma determinada API, gastem tempo aprendendo todos os seus detalhes;
- APIs gráficas mudam constantemente dado o avanço rápido de hardware nessa área, ocasionando mudanças constantes nas APIs de baixo nível, como DirectX e OpenGL, e, muitas vezes, quebrando a compatibilidade destas com suas versões anteriores;
- Um motor de jogo com diversas APIs pode atender a um numero maior de aplicações e ser migrado para outras plataformas, tomando-se o devido cuidado na codificação.

Para se atender a estas necessidades, disponibiliza-se uma Camada Gráfica (*Graphical API Layer Decoupling Pattern*) onde se obtém uma instância da API gráfica e seus objetos. Essa é uma especialização do padrão *Layers* (BUSCHMANN, 1996), em que a camada

esconde a API que está sendo utilizada e o desenvolvedor do jogo apenas precisa conhecer os objetos disponibilizados por ela.

Esta camada deve fornecer interfaces para abstrair os objetos necessários à geração de um jogo, como imagens, texturas, coordenadas, matrizes, vetores, janelas e *viewports* e vértices.

5.5.1.3. Aplicabilidade

O padrão *Graphical API Layer Decoupling Pattern* deve ser utilizado quando:

- Deseja-se suportar mais de um tipo de API gráfica num motor gráfico, como OpenGL ou DirectX;
- Deseja-se suportar mais de um sistema operacional, como Linux e Windows.

5.5.1.4. Estrutura

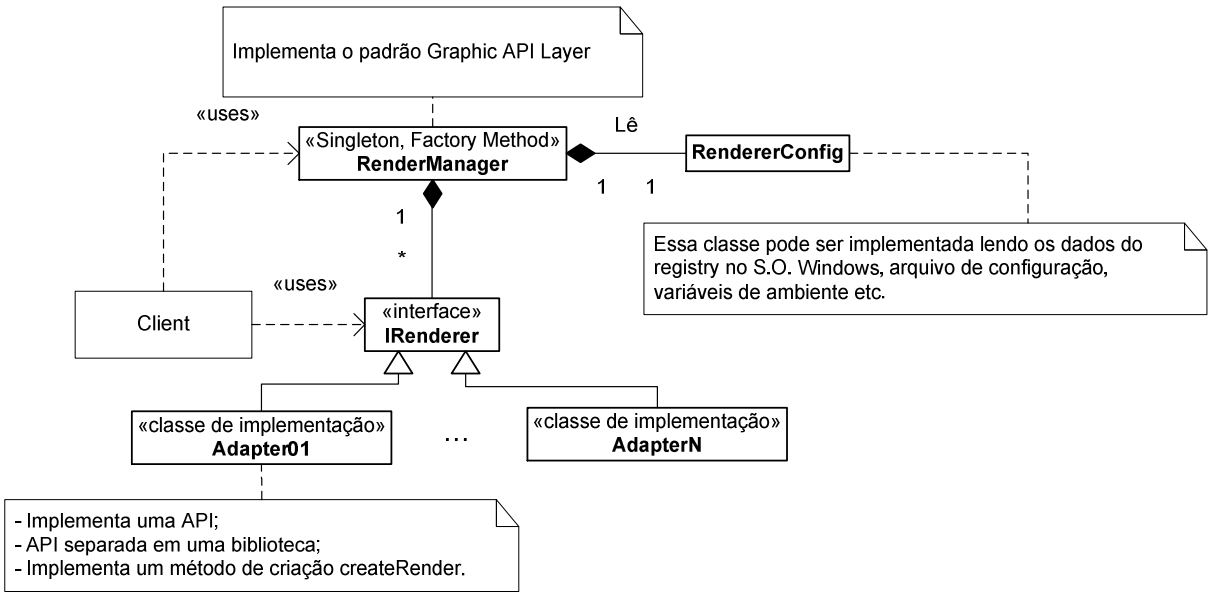

Figura 5.1 - Estrutura de *Graphical API Layer Decoupling Pattern*

5.5.1.5. Participantes

- **`Client`**: representa o módulo gráfico do motor do jogo que faz uso do padrão *Graphical API Layer Decoupling Pattern*;
- **`RenderManager`**: implementação o padrão *Graphical API Layer Decoupling Pattern*;

- **`IRenderer`:** interface que define os métodos gráficos a serem usados pelo motor gráfico;
- **`Adapters (Adapter01, ... AdapterN)`:** implementam a interface `IRenderer`;
- **Bibliotecas:** apesar de não serem propriamente uma classe, são usadas para a separação das APIs (`Adapters`) e devem disponibilizar um método para instanciação do `Adapter`.

5.5.1.6. Colaborações

- **`RenderManager`**: gerencia as instâncias de `IRenderer` e é o ponto de onde o `Client` recupera o `IRenderer` que está sendo usado no momento;
- **`Adapters`:** implementam o `IRenderer`, sendo que cada implementação abstrai uma API gráfica em especifico;
- **Bibliotecas:** existe uma biblioteca por Adapter. Cada biblioteca disponibiliza um método para que o RenderManager recuperar uma instância do Adapter.

5.5.1.7. Conseqüências

- ✓ O desacoplamento da API garante menor impacto em sua troca ou manutenção.
- ✓ A escolha entre diversas APIs permite aproveitar aceleradoras gráficas que tenham melhor desempenho em determinadas APIs. Por exemplo, a aceleradora gráfica X tem melhor desempenho em OpenGL ao invés de DirectX;
- ✓ O desenvolvedor do jogo trabalha sempre com as mesmas classes e objetos, não necessitando reaprender estruturas similares de outra API.
- ✓ A utilização de uma linguagem multiplataforma, como C++, seguindo padrão ANSI, fazem com que se permita uma melhor migração do motor para outras plataformas.

- A criação de uma camada entre a API gráfica e as chamadas do jogo pode causar impacto na performance final, dado este redirecionamento de chamadas.
- As interfaces dos objetos gráficos devem ser bem planejadas, prevendo usos futuros. Caso contrário, será necessária uma reestruturação geral dos objetos, o que afetará também as classes do jogo.

5.5.1.8. Implementação

O *Graphical API Layer Decoupling Pattern* é criado usando-se um conjunto de padrões formado por *Singleton*, *Abstract Factory*, *Dynamic Linkage* e *Adapter*.

A Figura 5.2 mostra o diagrama estrutural UML de *Graphical API Layer Decoupling Pattern*; este é implementado pela classe `RenderManager`.

A classe `RenderManager` gerencia as APIs gráficas, controlando a que está atualmente em uso e, também, a lista de APIs disponíveis. O padrão *Singleton* é usado para garantir a unicidade da API gráfica durante a execução do software.

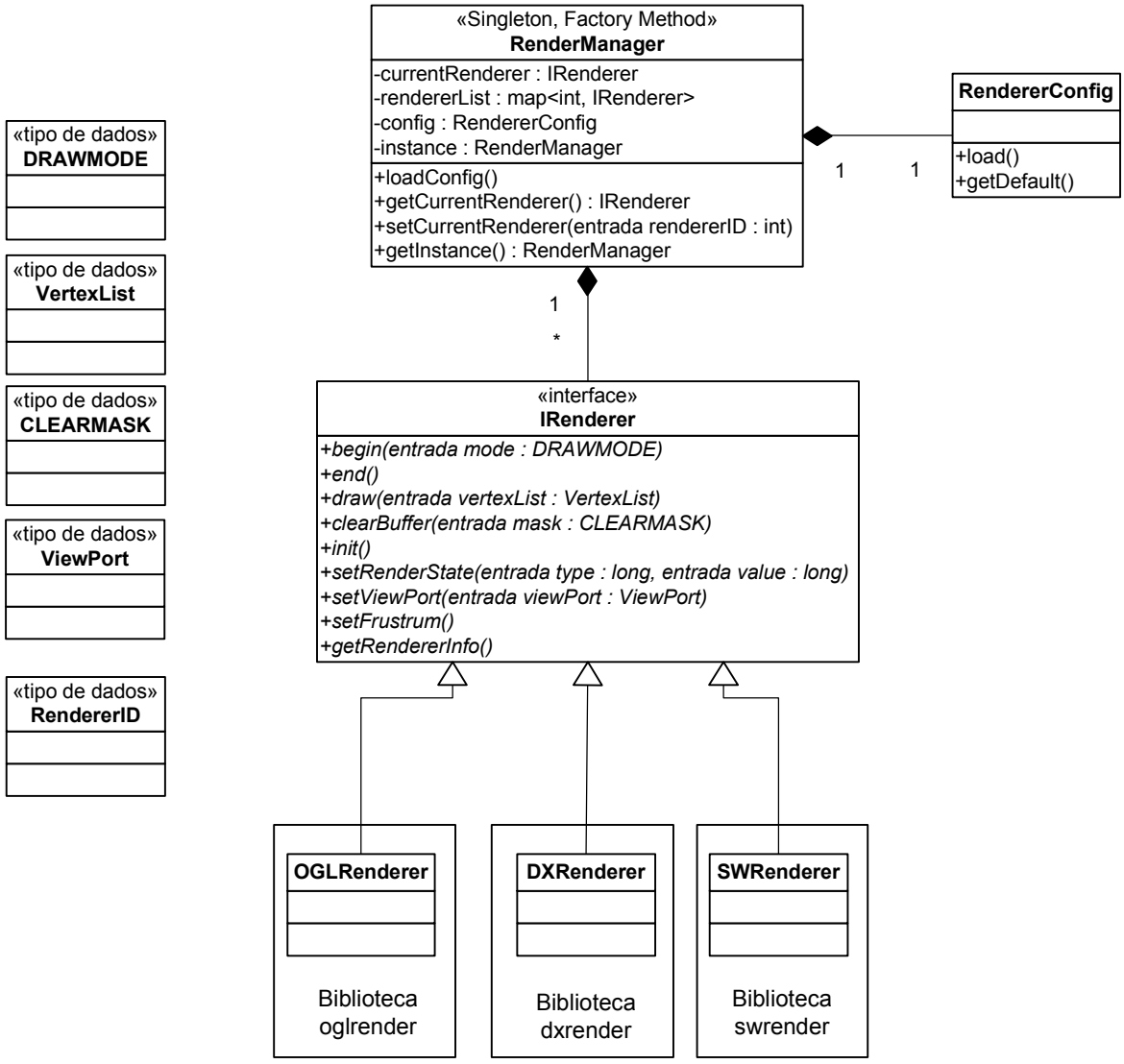

Figura 5.2 - Implementação *Graphical API Layer Decoupling Pattern*

A classe `RenderManager` tem como atributos:

- `currentRenderer`: a API de *rendering* que está em uso;
- `rendererList`: uma lista de *renderers* disponíveis, obtidos da classe `RendererConfig`.
- `config`: referência a classe `RendererConfig`. Essa classe de configuração pode ler dados do *registry* (Windows), de arquivos, variáveis de ambiente ou outro meio persistente qualquer. A Figura 5.3 mostra um exemplo dessa configuração implementada em um arquivo de configuração (render.ini).

```
[renders]
RenderDir=/plugins
NumRenders=3
Render1=renderogl.dll
Render2=renderdx.dll
Render3=rendersoft.dll
SelectedRender=renderogl.dll
```

Figura 5.3 - Exemplo de arquivo de configuração

A classe `RenderManager` tem como métodos:

- `loadConfig`: este método representa a implementação dos padrões Factory Method e Dynamic Linkage. Lê a configuração através da classe `RendererConfig`.
- `getCurrentRenderer`: recupera a API (*renderer)*, que está sendo usado atualmente;
- `setCurrentRenderer`: faz a troca da API por outra da lista.

Como outras características desta implementação, tem-se:

1. **Criação de tipos de dados específicos**: itens como coordenadas de textura, *viewport* e opção de estado de *rendering* devem ser implementados no motor do jogo, deixando que as classes *Adapter*, que implementam uma determinada API, convertam os dados para o seu formato.
2. **Criação de uma biblioteca para cada API**: para cada API a ser implementada deve-se criar uma biblioteca, por exemplo, uma dll no S.O. Windows, sendo que cada biblioteca armazena a classe Adapter correspondente e disponibiliza um método que retorna uma instância do Adapter, como mostra o trecho de código abaixo:

```
__declspec(dllexport) IRenderer* Create (void);
```

O método `Create` é a implementação do padrão de projeto Dynamic Linkage.

5.5.1.9. Exemplo de código

O código C++ a seguir, escrito para o sistema operacional Windows, implementa a classe `RenderManager`.

```
class RenderManager {

    private:
        static IRenderer *currentRenderer;
        static map<int, IRenderer*> rendererList;
        static RendererConfig config;
        static RenderManager *instance;
        void loadLibrary(wstring libName);
        RenderManager();

    public:
        virtual ~RenderManager();
        void loadConfig();
        static RenderManager* getInstance(){
           if(instance == NULL) {
              instance = new RenderManager();
            }
            return instance;
        };
        IRenderer* getCurrentRenderer();
        void setCurrentRenderer(RendererID rendererID);
};
void RenderManager::loadConfig() {

    //recupera a lista de configuração
    vector<std::wstring> *listLibs = config.load();

    //carrega as bibliotecas e armazena na lista
    for (int i = 0; i < listLibs->size(); i++) {
            loadLibrary((*listLibs)[i]);
      }
      //recupera o renderer definido para uso
    setCurrentRenderer(config.getDefault());
}

void RenderManager::loadLibrary(wstring libName) {
      HINSTANCE hDll;
      IRenderer *local;
        hDll = LoadLibrary(libName.c_str());
      if(hDll == NULL)
      {
            //trata erro
            return ;
       }
      CREATE createFunction = (CREATE)GetProcAddress(hDll,"Create");
      int result;
      if(createFunction != NULL){
          local = (createFunction)();
          if(local != NULL) {
              rendererList[rendererList.size()] = local;
          } else {
              //trata erro
          }
      }
}

IRenderer* RenderManager::getCurrentRenderer() {
    return currentRenderer;
}
```

```
      else
      {
           //trata erro
      }
void RenderManager::setCurrentRenderer(RendererID rendererID) {

    if(rendererList.count(rendererID) > 0)
        currentRenderer = rendererList[rendererID];
}
```

Abaixo está o código para instanciação de um Adapter, no caso, o que se comunica com a API Gráfica OpenGL. Essa função em conjunto com o método `loadLibrary` da classe `RenderManager`, implementa o padrão Dynamic Linkage

```
extern "C" {
      __declspec(dllexport) IRenderer* Create()
      {
            return new OGLRenderer();
      }
}
```

5.5.1.10. Usos conhecidos

- O motor gráfico multiplataforma OGRE3D, usa o padrão *Graphical API Layer Decoupling Pattern* para dar suporte a diversas APIs, como OpenGL e as diversas versões de DirectX de sofware.
- X-Engine é um motor de jogo que faz o uso do *Graphics API Layer* de forma similar ao OGRE3D.
- Java3D usa o *Graphical API Layer Decoupling Pattern* para decidir entre o uso de DirectX ou OpenGL na plataforma Windows.
- Generic Rendering System também usa um *Graphical API Layer Decoupling Pattern* para gerenciar a geração de gráficos 3D em OpenGL, RenderMan, POVRAY e Radiance.

5.5.1.11. Padrões relacionados ou com os quais interage

- *Singleton* garante uma única instância do objeto que representa a camada, garantido assim um ponto único de acesso;
- *FactoryMethod*, em conjunto com *DynamicLinkage*, fornece uma forma de obter a instância da API dinamicamente, permitindo troca de API em tempo de execução e configuração dinâmica.
- *Adapter* faz a conversão de formatos, objetos e sistemas de coordenadas para a API que está sendo utilizada;

5.6. Conclusões

Este capítulo resumiu as principais conclusões técnicas do estudo efetuado. O principal resultado foi o padrão *Graphical API Layer Decoupling Pattern*, que permite o desacoplamento entre o módulo gráfico (baixo nível) e o núcleo do motor gráfico. Constatou-se também o uso intenso de padrões de projeto nesses motores, bem como o reconhecimento por parte deles da necessidade de trabalhar aspectos como modularidade e extensibilidade.

O padrão *Graphical API Layer Decoupling Pattern* mantém as implementações para as diversas APIs separadas em bibliotecas, o que permite uma melhor manutenibilidade do motor.

Os métodos para criação de objetos gráficos, como texturas, foram aqui omitidos da classe `IRenderer`, dado que os trabalhos de Dollner e Hinrichs (2002) e Ecker(2003) abordam com maior profundidade esse assuntos.

6. Considerações Finais

6.1. Conclusões Gerais e Contribuições do Trabalho

Partindo da necessidade de se entender melhor a arquitetura do subsistema gráfico do motor de jogo para propor soluções de sua reutilização, discutiu-se nos capítulos 01 e 02 os aspectos econômicos com o desenvolvimento de jogos e a necessidade de busca por tecnologias mais avançadas para essa área; também, foram apresentados, no capítulo 2, os diversos módulos que compõem o motor de um jogo e suas funcionalidades e aplicações, tanto na área de entretenimento, quanto na área educacional.

No capítulo 3, é apresentada uma introdução sobre *design patterns* e *frameworks*, que, em conjunto com as análises do código de motores de jogos apresentadas no capítulo 4, foram usados para obtenção dos resultados listados no capítulo 5.

A análise efetuada no capítulo 4 demonstrou a crescente preocupação dos desenvolvedores dos motores de jogos com a evolução e o suporte a diversas plataformas e como os padrões de projetos estão presentes na maior parte dos motores de jogos existentes no mercado, tornando-se essencial seu conhecimento para os profissionais desta especialidade. Essa análise fornece importantes conclusões para as disciplinas ligadas a projeto de jogos, pois mostra a necessidade de conhecimento por parte do aluno da tecnologia de reuso baseada em padrões de projeto; além disso, listam-se os padrões de projeto mais comumente utilizados nesses projetos.

Em todos os motores analisados, pôde-se notar que padrões de projeto colaboram para a abstração de plataforma, desde que não sejam utilizados aspectos específicos da plataforma ou sistema operacional.

Apesar de terem sido apresentados os padrões de projeto como técnica para modelagem de aplicações orientadas a objetos, isso não significa que se deva apenas utilizar este tipo de

linguagem. Eide (2002) demonstra como utilizar padrões de projeto em linguagens imperativas e funcionais. Dessa forma, padrões de projeto se mostram ferramentas que permitem capturar o conhecimento de projetos, independente da linguagem de implementação utilizada.

Por fim, esse trabalho permitiu reconhecer um novo padrão de projeto para isolar o uso de diversas APIs e permitir que outras sejam integradas ao motor, de forma quase imperceptível para o usuário.

6.2. Sugestões de Pesquisas Futuras

Os padrões aqui mostrados foram determinados com foco na abstração da API gráfica usada, mas o motor do jogo é composto por outros subsistemas, como Áudio, Inteligência Artificial e Entrada de Dados, que, da mesma forma que o subsistema de *rendering*, possuem diversos *middlewares*. No caso do Áudio, por exemplo, temos OpenAL e DirectSound, que podem também utilizar esse conjunto de padrões apresentados, mas que, certamente, possuem outros mais específicos ao seu domínio.

Outro ponto que não pôde ser verificado nesse trabalho é o impacto que a utilização desta nova camada pode causar com relação ao desempenho geral do motor de jogo para averiguar se os ganhos com extensibilidade e manutabilidade compensão essa perda de performance.

REFERÊNCIAS

ADAMI C. **Introduction to Artificial Life.** Berlin:Springer. 1998.

AGEIA™ TECHNOLOGIES INC. AGEIA Press Release. Disponível em: <http://ageia.vnewscenter.com/press.jsp?id=1147177728171 > Acesso em 05 maio 2006.

ALEXANDER, C. **A Pattern Language**. USA: Oxford University Press. 1977.

AZEVEDO, E.; CONCI, A. **Computação Gráfica – Teoria e Prática**. Rio de Janeiro: Campus. 2003.

BATTAIOLA, A. L.; ELIAS, N. C.; DOMINGUES, R. G.; ASSAF, R.; RAMALHO, G. L. Desenvolvimento da Interface de um Software Educacional com base em Interfaces de Jogos. **IHC 2002 – V Simposium on Human Factors in Computer Systems**, Fortaleza, Proceedings. p. 214-225, out 2002

BIANCHINI, R. C. Uma Arquitetura BDI para Comportamentos Interativos de Agentes em Jogos Computacionais Tridimensionais. 2005. Tese(Doutorado) - Escola Politécnica da Universidade de São Paulo, São Paulo, 2003.

BIOWARE CORP. NeverWinter Nights. Disponível em: <http://nwn.bioware.com/>. Acesso em: 20 maio 2005.

BITTENCOURT, João R.; GIRAFFA, Lucia M.M.; SANTOS, Rafael C. Making Multplataform computer games with Amphibian. **II Congresso Internacional de Tecnologia e Inovação em Jogos para Computadores**, Curitiba: GameNet/CTIS, 2003.

_____*. Desenvolvendo Jogos Computadorizados Multiplaforma com Amphibian. **V Workshop sobre Software Livre**, Porto Alegre: SBC, 2004, 119-122p.

_____*. Amphibian - Um Framework para a Criação de Game Engines Multiplataforma. **II Workshop Brasileiro de Jogos e Entretenimento Digital**, Salvador: SBC, 2003.

_____*; SANTOS, Rafael C.; GIRAFFA, Lucia M.M. Desenvolvimento de um Framework para Criação de Game Engines Multiplataformas**. IV Salão de Iniciação Científica da PUCRS**, Porto Alegre: PUCRS, 2003.

_____*; SANTOS, Rafael C.; GIRAFFA, Lucia M.M. Amphibian - Um Framework para o Desenvolvimento de Jogos Educativos. **XV Salão de Iniciação Científica da UFRGS**, Porto Alegre: UFRGS, 2003.

BJÖRK, S.; LUNDGREN, S.; HOLOPAINEN, J. Game Design Patterns, **Level Up. Digital Games Research Conference**, Utrecht, The Netherlands, Proceedings, p. 4-6, nov. 2003

BLOW, J. Game Development: Harder Than You Think. ACM Queue, v. 1, n. 10, Feb 2004. Disponível em: < http://www.acmqueue.org/modules.php?name=Content&pa=showpage&pid=114>. Acesso em 08 ago 2005.

BRAUDE, E., **Projeto de Software: da programação à arquitetura: uma abordagem baseada em Java**. Tradução Edson Furmankiewicz, Porto Alegre: Bookman, 2005.

BUSCHMANN, F.; MEUNIER, R.; ROHNERT, H.; SOMMERLAD, P.; STAL, M. **Pattern-Oriented Software Architecture**. Volume 1**,** John Wiley & Sons, 1996.

Calef, .C.; Vilbrandt,T.; Vilbrandt, C.; Goodwin, J.; Goodwin,J. **Making it Realtime: Exploring the use of optimized real-time Environments for historical simulation and education**. Museums and the We., MW2002, Boston Mass, 2002. Disponível em: <http://www.archimuse.com/mw2002/papers/calef/calef.html> Acesso em: 16 maio 2004

CRITERION SOFTWARE. RenderWare. Disponível em: <http://www.renderware.com/> Acesso em: 20 maio 2005.

CRYTEK. Crytek Engine.Disponível em: <http://www.crytek.com/> Acesso em: 26 maio 2005.

DALMAU, D. **Core Techniques and Algorithms in Game Programming**. 1. ed. Indianapolis: New Riders. 2003.

DAWSON, B. GDC 2002: Game Scripting in Python. Disponível em: http://www.gamasutra.com/features/20020821/dawson_01.htm, 2002. Acesso em: 16 nov 2003.

DIGITAL ILLUSIONS CE. Battlefield 2. Disponível em: < http://www.battlefield2.com/> Acesso em 08 ago 2005.

DOLLNER, J.; HINRICHS, K. A generic rendering system, Visualization and Computer Graphics, **IEEE Transactions**, v. 8, n. 2 , p. 99 –118, Apr-Jun, 2002.

EBERLY, D. H. **3D Game Engine Design**: A Practical Approach to Real-Time Computer Graphics. Morgan Kaufmann, 2000

_____*. **3D Game Engine Architecture :** Engineering Real-Time Applications with Wild Magic (The Morgan Kaufmann Series in Interactive 3D Technology). Morgan Kaufmann, 2004

ECKER, M. X-Engine: A platform and API Independent Real-Time 3D Engine with Support for Programmable Graphics Pipeline Architectures. 2003. Dissertação (Mestrado) - Johannes Kepler Universität, Austria: Linz,2003.

EIDE, E. et al, Static and dynamic structure in design **patterns**, **Proceedings of the 24th international conference on Software engineering**, p.208 – 218, 2002.

EPIC GAMES. *Unreal Engine*. Disponível em: <http://www.epicgames.com/> Acesso em: 25 maio 2005.

_____*. Unreal Tournament 2004. Disponível em: <http://www.unrealtournament.com/> Acesso em: 25 maio 2005.

_____*. Unreal Technology. Disponível em: <http://www.unrealtechnology.com/> Acesso em: 25 maio 2005.

_____*. MSU: Make Something Unreal. Disponível em:<www.makesomethingunreal.com/> Acesso em: 25 maio 2005.

FERNANDEZ, A. Usabilidade Em Aparelhos Móveis. WUD – World Usability Day. 2005. Disponível em:< http://www.wud.com.br/programacao>. Acesso em: 10 junho 2006.

FOLEY, J. D.; VAN DAM, A.; FEINER, S. K.; HUGHES, JOHN F. Computer Graphics: principles and practice. 2 edição. Addison-Wesley, 1997.

FRAKES, W. B; TERRY, C. Software Reuse: Metrics and Models. **ACM Computing Surveys**, v. 28, n. 2 , p. 415-435, jun, 1996.

FREESCALE SEMICONDUCTOR'S METROWERKS, Game Produtcs, Disponível em: <http://www.metrowerks.com/MW/Develop/Games/Products.htm> Acesso em 11 dez 2005.

FRISTOM J. Manager In A Strange Land: Reuse and Replace. 09 janeiro 2003. Gamasutra. Disponível em: < http://www.gamasutra.com/features/20040109/fristrom_01.shtml> Acesso em: 08 ago 2005.

GAME2TRAIN. *Social Impact Games*. Website com jogos voltados a treinamento/educação. Disponível em: <http://www.socialimpactgames.com/> Acesso em: 25 maio 2005.

GAMESPY. GameSpy stats. Disponível: <http://archive.gamespy.com/stats/> Acesso em: 26 maio 2005.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Padrões de Projeto**. Trad. Luiz A. Meirelles Salgado. Porto Alegre: Bookman, 2000.

GILL, N. S. Reusability issues in component-based development, **ACM SIGSOFT Software Engineering Notes**, v. 28, n. 4 , p. 4, jul, 2003.

GNU. Project. Bison. Disponível em: <http://www.gnu.org/software/bison/bison.html>. Acesso em 12 fev 2006.

_____*.Flex. Disponível em: <http://www.gnu.org/software/flex/flex.html>. Acesso em 12 fev 2006.

GRAND,M. **Patterns in Java**, Volume 1 - A Catalog of Reusable Design Patterns Illustrated with UML, Second Edition, John Wiley & Sons, 2002.

HAVOK. Havok Physics SDK. Disponível em: <http://www.havok.com/>. Acesso em: 20 maio 2005.

ID SOFTWARE. Quake I Engine. Disponível em: <http://www.idsoftware.com/business/techdownloads/> Acesso em: 25 maio 2004.

_____*. Quake II Engine. Disponível em: <http://www.idsoftware.com/business/techdownloads/> Acesso em: 25 maio 2004.

_____*. Quake III Engine. Disponível em: <http://www.idsoftware.com/business/techdownloads/> Acesso em: 25 maio 2004.

_____*. Doom 3 Engine. Disponível em: <http://www.idsoftware.com/> Acesso em: 26 maio 2004.

ION STORM. Daikatana. Disponível em: <http://www.eidosinteractive.com/games/info.html?gmid=11>,2000, Acesso em: 26 maio 2004.

JACOBSEN, T. Advanced Character Physics. Gamasutra, 2003. Disponível em: <http://www.gamasutra.com/resource_guide/20030121/jacobson_01.shtml> Acesso em: 20 maio 2005.

JOGL, Java Bindings for OpenGL, Game Technology Group. Disponível em: <https://jogl.dev.java.net/> Acesso em 12 dez 2005.

KEARNEY, P., SKELTO, S. Teaching Technology To The Playstation Generation, **Proceedings of the 16th Annual NACCQ**. New Zealand, Plamerston North, p. 79 – 84, 2003.

KIRBY, N. GDC 2005 - AI Round Table. Disponível em: <http://www.gameai.com/cgdc05notes.kirby.html> Acesso em: 25 maio 2005

LAMOTHE, A. Tricks of the windows game programming gurus. 2. Ed. Indianapolis: SAMS. 1999.

_____*. **Tricks of the 3D Game Programming Gurus**: Advanced 3D Graphics and Rasterization. SAMS, 2003.

LARMAN, C. **Applying UML and Patterns**: An Introduction to Object-Oriented Analysis and Design. Prentice Hall PTR, 1998.

LEE, H. J. Software Reusability. Disponível em: <http://www.cs.utexas.edu/users/almstrum/cs370/hyon/research.html>. Acesso em: 28 jul 2005.

LENGYEL, E. Simultaneous Cross-Platform Game Development, GamaSutra, 2000 Disponível em: <http:// http://www.gamasutra.com/features/20000124/lengyel_01.htm> Acesso em 15 dez 2005.

LEWIS M., JACOBSON J., Game engines in scientific research: Introduction, **Comm. of the ACM**. v. 45, n. 1 ,p.27-31, jan, 2002.

LOKI SOFTWARE. OpenAL: Open Audio Library. Disponível em: <http://www.openal.org/>. Acesso em 08 ago 2005.

MACEDONIA, M. Games, Simulation and the Military Education Dilemma. Educause. Disponível em: <http://www.educause.edu/ir/library/pdf/ffpiu018.pdf> Acesso em: 16 maio 2004.

MADEIRA, C.A.G. Forge V8: Um Framework para desenvolvimento de jogos de computador e aplicações multimídia,Dissertação (Mestrado) - Centro de Informática, Universidade Federal de Pernambuco, Recife, 2001.

MCCOOL, M. Sh: A high-level metaprogramming language for modern GPUs. Disponível em: < http://libsh.org>. Acesso em 08 ago 2005.

MEQON RESEARCH AB. Meqon Physics SDK. Disponível em: <http://www.meqon.com/>. Acesso em: 20 maio 2005.

MICROSOFT. Microsoft DirectX Technology Overview. Microsoft Corporation. Disponível em: <http://www.microsoft.com/windows/directx/default.aspx?url=/windows/directx/productinfo/overview/default.htm> Acesso em: 25 maio 2004.

_____*. DirectSound. Microsoft Corporation. Disponível em: <http://www.microsoft.com/windows/directx/default.aspx?url=/windows/directx/productinfo/overview/default.htm> Acesso em: 25 maio 2004.

_____*. XNA Press Release. Microsoft Corporation. Disponível. em: <http://www.microsoft.com/xna/> Acesso em: 25 maio 2004.

_____* . Microsoft Foundation Libray. Microsoft Corporation. Disponível em: <http://msdn.microsoft.com/library/devprods/vs6/visualc/vcmfc/mfchm.htm> Acesso em: 28 maio 2005.

_____* . DirectX Frequently Asked Questions, Microsoft Corporation, Disponível em: <http://www.microsoft.com/windows/directx/default.aspx?url=/windows/directx/productinfo/faq/default.htm> Acesso em 12 dez 2005.

_____* . Microsoft Component Object Model, Microsoft Corporation, Disponível em: <www.microsoft.com/com/ > Acesso em 12 dez 2005.

NAKAMURA, R.; TORI, R.;JACOBER, E.;BIANCHINI, R.;BERNARDES JR., J. L.Development of a Game Engine using Java, **II Workshop de Jogos e Entretenimento Digital - WJogos'03**. Bahia(Salvador), 2003.

NCSOFT, LineAge 2 Disponível em: <http://www.lineage2.com>, 2004, Acesso em 13 dez 2005.

NGUYEN, D.; WONG, S. B. Design Patterns for Games, **Proceedings of the 33rd SIGCSE technical symposium on Computer science education**, Cincinnati, Kentucky, p.126 – 130, 2002.

NILSEN, J. Interface:The Importenace of Being Beautiful. **IEEE Software**, v. 8, n. 1 , p. 92 – 94, Jan, 1996.

OGRE3D. About,<http://www.ogre3d.org/> Acesso em 12 dez 2005.

_____*. Testimonials,<http://www.ogre3d.org/> Acesso em 12 dez 2005.

OPENGL ARCHITECTURE REVIEW BOARD. OpenGL Overview. Disponível em: <http://www.opengl.org/about/overview.html> Acesso em: 25 maio 2004.

_____*. OpenGL 2.0. Disponível em: <http://www.opengl.org/documentation/opengl_current_version.html>. Acesso em: 08 ago 2005.

POIKER, F. Scripting. . Relic Enterteiment. Disponível em: <http://www.gameai.com/cgdc03notes.html> Acesso em: 16 nov 2003.

PONDER, M.; PAPAGIANNAKIS, G.; MOLET, T.; MAGNENAT-THALMANN, N.; THALMANN, D.. VHD++ development framework: towards extendible, component based VR/AR simulation engine featuring advanced virtual character technologies. **Proceedings of Computer Graphics International 2003**, p.96 – 104, jul, 2003.

REALNETWORKS INC., Full Screen Vídeo with RealPlayer and RealPlayer Plus 5.0, G2, 7, and 8, Disponível em: <http://service.real.com/fullscreen/fsdxinfo.html> Acesso em 12 dez 2005.

REIMER, J. Cross-Platform Game Development And The Next Generation Of Consoles, Disponível em: <http://arstechnica.com/articles/paedia/hardware/crossplatform.ars/1> Acesso em 13 dez 2005.

RELIC ENTERTAINMENT. Homeworld Engine. Disponível em: <http://www.relic.com/rdn/index.php> Acesso em: 25 maio 2004.

RENDERWARE. RenderWare Engine. Disponível em:< http://www.renderware.com/>. Acesso e: 26 maio 2005.

ROUSE III, R. Game design: Theory & Practice. Richard Rouse. Wordware Punlishing Inc, 2001

_____*. Scripting Languages and Object Behaviors. Disponível em: <http://www.gamasutra.com/features/20000323/rouse_03.htm>, 2000, Acesso em: 16 nov 2003.

SHIRATUDDIN, M.F.; YAAKUB, A.R.; ARIF, A.S.C.M. Games Engine in Real World Virtual Reality Application. Disponível em: <http://www.nottingham.ac.uk/~enzrh/VRSIG7Proc/Shiratuddin/Shiratuddin.html> Acesso em: 16 maio 2004.

SILVA, F. F. Reutilização de software através de geração de código e de desenvolvimento de componentes: Estudo de caso. 2003. Dissertação de Mestrado. Universidade de São Paulo - Escola Politécnica. São Paulo:São Paulo, 2003.

Sommerville, I. Engenharia de Software. 6a. Ed. São Paulo: Addison-Wesley. 2003.

SOURCEFORGE. Lista de *game engines* de código aberto. Disponível em: <http://www.sourceforge.net/> Acesso em: 25 maio 2005.

SOUZA, R. A. Um Processo De Transformação De Arquiteturas De Sistemas Legados Baseado Em Reengenharia. 2003. Dissertação (Mestrado) - Universidade de São Paulo, Escola Politécnica, São Paulo:São Paulo, 2004.

STEINMEYER, P. Development Platforms for Casual Games, GamaSutra, 2005, Disponível em: <http://www.gamasutra.com/features/20050324/steinmeyer_01.shtml> Acesso em 12 dez 2005.

SUN MICROSYSTEM, Java3D, Disponível em: <http://java3d.dev.java.net/> Acesso em 12 dez 2005.

_____*. JNI - Java Native Interface. Disponível em: <http://java.sun.com/j2se/1.4.2/docs/guide/jni/> Acesso em 12 dez 2005.

SUPER WABA LTDA, SuperWaba, <http://www.superwaba.com.br> Acesso em 12 dez 2005.

SWEENEY, T. UnrealScript Language Reference. Disponível em: <http://udn.epicgames.com/pub/Technical/UnrealScriptReference/> ,1999, Acesso em: 26 jul 2005.

TECGRAF. Lua. Pontifícia Universidade Católica do Rio de Janeiro. Disponível em: <http://www.lua.org/> Acesso em: 26 jul 2005.

THE EDUCATION ARCADE. Education Arcade. Website com projetos de jogos voltados a treinamento/educação. Disponível em: <http://www.educationarcade.org/> Acesso em: 25 maio 2005.

TORRES, E. Forge V16: Um Framework para Desenvolvimento de Jogos Isométricos, Dissertação de Mestrado, Centro de Informática, Universidade Federal de Pernambuco, Recife, 2003.

VALENTE, L. GUFF: Um Sistema para Desenvolvimento de Jogos. Dissertação (Mestrado em Computação Visual e Interfaces) -Universidade Federal Fluminese, Rio de Janeiro, 2005.

VALVE SOFWARE. Source Engine. Disponível em: <http://www.valvesoftware.com/sourcelicense/> Acesso em: 26 maio 2005.

_____*. Counter-Strike: Source. Disponível em: <http://www.valvesoftware.com/games.html.> Acesso em: 26 maio 2005.

_____*. Half-Life. Disponível em: <http://www.valvesoftware.com/games.html.> Acesso em: 26 maio 2005.

_____*. Half-Life 2. Disponível em: <http://www.valvesoftware.com/games.html.> Acesso em: 26 maio 2005.

VAN ROSSUM, G.Phyton. Disponível em: <http://www.python.org/> Acesso em 26 jul 2005.

VIDAL, C. A.; GOMES, G. A. M.; MENDONÇA JUNIOR, G. M.; GOMES, H. O. º; CAVALCANTE NETO, J. B. . Uma Ferramenta de Autoria de Ambientes Virtuais Adaptável a Diferentes Motores Gráficos. In: Symposium on Virtual Reality, 2004, São Paulo. **Proceedings of the 7th Symposium on Virtual Reality**. São Paulo : Plêiade, 2004. v. 1. p. 15-26.

WIKIPEDIA. Polygon Mesh. Disponível em: <http://en.wikipedia.org/wiki/Polygon_mesh > Acesso em: 26 maio 2005.

WOODCOCK, S. AI:The State of The Industry, Game Developer Magazine, p. 28-35, oct, 1998.

_____*. GDC 2003 Notes. Disponível em: <http://www.gameai.com/cgdc03notes.html> Acesso em: 25 maio 2003

WXWIDGETS, WxWidgets framework, Disponível em: <http://www.wxwidgets.org/> Acesso em 12 dez 2005.

YOUNG V. **Programming A Multiplayer FPS in DirectX**. 1. ed. Massachusetts: Charles River Media. 2005.

ANEXO A – AUTORIZAÇÃO PARA USO DE IMAGENS

Abaixo está o e-mail de resposta de Jonathan Blow para uso das imagens de arquitetura de motores de jogos, utilizadas no capítulo 2.

```
From: Jonathan Blow <jon@number-none.com>
User-Agent: Thunderbird 1.5 (Windows/20051201)
MIME-Version: 1.0
To: Thiago <lonegunner@gmail.com>
Subject: Re: authorization for image usage
References:
<9b905b060602121455u73a5392ds4c1c9c2dff2f5f86@mail.gmail.com>
In-Reply-To:
<9b905b060602121455u73a5392ds4c1c9c2dff2f5f86@mail.gmail.com>
Content-Type: text/plain; charset=ISO-8859-1; format=flowed
Content-Transfer-Encoding: 7bit

Sure, go ahead.

Thiago wrote:
> Hi,
>
> I would like to use the images available in the article you wrote,
> intituled Game Development: Harder Than You Think, in my master's
> thesis, and I am asking if you authorize so.
>
> I thank you already.
>
> Thiago Matias Busso
```

GLOSSÁRIO

Cena do jogo	Repesentra um instante/momento de uma fase do jogo.
Dagling pointers	um ponteiro para memória na linguagem C/C++, que não está apontado para algo válido, exemplo: um objeto que já foi liberado da memória
DirectX	Pacote de APIs criado pela Microsoft para o desenvolvimento de jogos e aplicações multimídia.
Fase do jogo	Representa uma missão ou objetivo de um jogo. Um jogo pode ser composto de diversas fases podendo ser em ambientadas em mapa diferentes ou no mesmo mapa.
FPS	*First Person Shooter* ou Jogo de Tiro onde a visão do jogador é em primeira pessoa.
FRAMEWORK	Biblioteca de classes e interfaces, desenvolvida com o intuito de ser uma base customizável para o desenvolvimento de aplicações. A principal motivação para a utilização de frameworks é a eliminação de programação de funções comuns por parte do desenvolvedor ou projetista do sistema (LARMAN, 1998).
Game Content	Todo conteúdo utilizando por um jogo que não é código executável, como mapas, texturas, imanges, modelos, sons e personagens.
Game Design	Ë o que determina a forma de *game play*. O *game design* define que opções o jogador pode fazer no universo do jogo e que conseqüências essas opções (que, no contexto do jogo, são ramificações), terão durante o resto do jogo. O *game design* inclui que, o critério de vitória e derrota o jogo pode incluir, como o jogador controla o jogo, que informações o jogo mostrará e quão difícil será o jogo (ROUSE III, 2001)
Game IA	Inteligência artificial utilizada em jogos (BIANCHINNI, 2003)
Game play	Ou jogabilidade, é o grau e natureza da interatividade que o jogo inclui, isto é, como o jogador pode interagir com o universo do jogo e como esse universo reage às ações que jogador faz (ROUSE III, 2001)
GTK	Inicialmente criado para o editor de imagens GIMP, o GIMP Toolkit (GTK) é um *toolkit* gráfico para o sistema X11, usado para criação de interfaces gráficas.
Interface	Uma interface, dentro do paradigma de orientação a objetos, pode ser definida como um conjunto de tipos abstratos de dados ou objetos. A interface descreve os dados que podem ser acessados no subsistema (de *rendering*) e as operações disponibilizadas (SOMMERVILLE, 2003)
Jogabilidade	Ou *game play*, é o grau e natureza da interatividade que o jogo inclui, isto é, como o jogador pode interagir com o universo do jogo e como esse universo reage às ações que jogador faz (ROUSE III, 2001)
Mechas	Uma coleção de vértices e polígonos que define a forma de um objeto em 3D,

Memory leaks	Situação que ocorre quando um programa falha ao retorna um bloco de memória alocado para o sistema. Memory leaks podem levar a falha de sistema após longos períodos de execução
OpenGL	API para o desenvolvimento de aplicações 2D e 3D.
Padrão de Projeto Comportamental	Padrões que lidam com a composição de classes ou de objetos. Exemplo: *Pattern Bridge*
Padrão de Projeto de Criação	Padrões que se preocupam com o processo de criação de objetos, ou seja, como os objetos são instanciados ou construídos. Exemplo: *Pattern Singleton*
Padrão de Projeto Estrutural	Padrões que caracterizam a maneira pela qual classes ou objetos interagem e distribuem responsabilidades. Exemplo: *Pattern Iterator*
Pipeline	Partição de uma computação em estágios, que podem ser executados seqüencialmente, em elementos de processamento separados (FOLEY, 1997)
Qt	Qt é um *toolkit* de elementos gráficos multiplataforma para o desenvolvimento da interface de programas. Seu uso mais conhecido é o *K Desktop Enviroment* (KDE), ambiente gráfico usado nas plataformas Unix/Linux.
Renderer	Componente de sistema de computação gráfica que é responsável por desenhar os triângulos de um modelo, usando as informações de desenho atuais conhecidas como *render states*. Um *software renderer* é uma implementação que apenas utiliza a unidade central de processamento (CPU) para as computações. Um *hardware renderer* é uma implementação que depende de um unidade de processamento gráfica para as computações (EBERLY, 2000).
Rendering	Processo computacional executado pelo *renderer*. veja ***renderer***.
Turn based game	Estilo de jogo baseado em turnos, onde cada jogador faz suas jogadas individualmente e ao término da sua jogada passe a vez para o próximo jogador. Esse sistema de jogo é muito similar ao sistema de jogador de um jogo de tabuleiro como xadrez ou damas.
X11	*X Window System Protocol Version 11* é sistema de janelas para dispositivos bitmaps, que é usado para criação de interfaces gráficas no sistemas operacionais da família Unix e sistema derivados.
X-BOX	*Vídeo-Game* produzido pela Microsoft.
XNA	Plataforma de desenvolvimento criado pela Microsoft para acelerar o processo de criação de jogos eletrônicos integrando diversas ferramentas de apoio.