CRAbGE – UMA ARQUITETURA PARA MOTORES GRÁFICOS FLEXÍVEIS, EXPANSÍVEIS E PORTÁVEIS

por

José Gilvan Rodrigues Maia

Dissertação Apresentada ao
Mestrado em Ciência da Computação
Universidade Federal do Ceará

Orientador:
Joaquim Bento Cavalcante Neto, Dr.

Abril, 2005

Agradecimentos

Em primeiro lugar, agradeço a Deus por tudo que tem feito por mim e pelos meus entes queridos; à minha mãe, Antônia, uma mulher forte e batalhadora, que é com certeza o maior presente que Deus me concedeu; ao meu pai Raimundo; às minhas irmãs Eridan, Elian, Evarina, Leireane e Diana; à minha amada noiva Adriana, pelo seu amor, apoio, carinho, companhia e, principalmente, por me fazer feliz; aos meus orientadores Bento e Creto, pelo crédito, paciência e pela transferência de conhecimento; aos meus amigos George, Ney, Humberto, Edvan, Wellington, Patrícia, Henrique, Luiz Cláudio, "Dudu" Peixoto, "Dudu" Ferreira, David, Melo Júnior, Camilo, Emanuele, Ciro Vidal, "Paulinho", e qualquer um que porventura eu não tenha mencionado aqui; ao pessoal da Lunart, que me apoiou bastante; a quem acreditou e, principalmente, a quem não acreditou nesse trabalho; aos meus amigos e familiares.

Sumário

Os Sistemas de Realidade Virtual caracterizam-se tanto pelo uso de estímulos sensoriais, quanto pelo alto poder de interação com o usuário, devendo, portanto, considerar diversos aspectos de representação do ambiente virtual. A gerência desses aspectos demanda a existência de um conjunto de subsistemas que provêem serviços específicos, como Renderização e Detecção de Colisões, que são integrados no contexto de um sistema de RV.

A implementação desses complexos subsistemas implica em enorme esforço de desenvolvimento e geralmente foge ao escopo de desenvolvimento das aplicações de RV. Assim, o reuso de um framework que encapsule tais subsistemas é uma opção atraente no contexto do desenvolvimento dessas aplicações. Recentemente, os desenvolvedores de aplicações de RV passaram a adotar também Motores Gráficos, uma tecnologia originalmente utilizada para jogos tridimensionais, em seu processo de desenvolvimento. Porém, a maioria dos motores gráficos apresenta uma série de deficiências que impede sua plena utilização por parte das aplicações de RV.

O presente trabalho propõe uma nova arquitetura para motores gráficos baseada na integração de pacotes de domínio público, que utiliza as vantagens da orientação a objetos e visa conferir flexibilidade quanto ao uso de diversas tecnologias para prover os serviços básicos disponibilizados nesse *framework* (Renderização, Detecção de Colisões, Som Tridimensional e Suporte a Linguagens de Script). A arquitetura proposta permite a extensão do motor, que pode ser realizada através de novas implementações dos subsistemas pré-existentes ou ainda pela definição de outros serviços em novos componentes e subsistemas.

Summary

Virtual Reality Systems are characterized not only by the use of sensorial stimuli, as well as by the high level user interaction, therefore, they have to consider a diverse number of aspects in the representation of virtual environments. The management of these aspects demands the existence of a set of subsystems, which provide specific services, such as Real-time Rendering and Collision Detection, that are integrated in the context of a VR system.

The Implementation of these complex subsystems implies a significant development effort and usually is out of the development scope of the VR applications. Thus, reusing an existing framework that encapsulates such subsystems is an attractive option in the development context of such applications. Recently, many VR system developers are using Game Engines in the development process, especially in low-cost systems and Desktop platforms. However, the majority of game engines present a series of limitations, which hinder the use of their full power by the VR applications.

In this work, it is presented a new, flexible architecture for game engines based on the integration of public domain software packages, which relies on the OO design paradigm to provide independence of the technologies suitable for providing the basic services offered by the proposed framework (Real-time Rendering, Collision Detection, Spatial Sound and Scripting Languages). The proposed architecture allows for the engine's extension through new implementations of the existing services or, yet, through the definition of new services and components.

Índice

Índice de Figuras

Índice de Tabelas

Capítulo 1

Introdução

1.1 Motivação

As aplicações de Realidade Virtual (RV) têm-se tornado cada vez mais populares e comuns, sendo utilizadas em áreas tais como educação (Leite-Júnior et al. 2002; Kaufmann, 2002; Almendra et al. 2002), treinamento (Fisher, 2001; Garcia, 2002), entretenimento (Activision, 2004; Epic Games, 2004; Id Software, 2005; Leite-Júnior et al., 2002) e medicina (Diar, 2002; Rodrigues et al., 2002; Machado, 2003), entre outras.

As aplicações de RV caracterizam-se pelo uso de múltiplos estímulos sensoriais e pela intensa interação, oferecendo ao usuário a sensação de imersão num ambiente simulado por computador. Dessa maneira, tanto os aspectos de representação do ambiente virtual quanto os aspectos multissensoriais e os níveis de interação possíveis no ambiente são determinantes da qualidade dessas aplicações. Assim, a construção de uma aplicação de RV com um nível razoável de qualidade demanda um considerável esforço de desenvolvimento (Maia et al., 2003).

Sob o ponto de vista arquitetural, uma aplicação de RV é composta por uma série de subsistemas que coordenam cada um dos aspectos do ambiente, como, por exemplo, os sons, a renderização da cena e o reconhecimento das entradas do usuário. O desenvolvimento desses tipos de subsistemas é uma tarefa delicada que requer um imenso esforço de programação, demandando grandes alocações de tempo e pessoal, recursos que são geralmente escassos no desenvolvimento desse tipo de projeto.

Segundo Ponder et al. (2003), os recentes avanços nas tecnologias de computação gráfica e simulação em tempo-real põem uma luz completamente nova tanto nos sistemas de RV quanto nos de Realidade Aumentada, quanto nos jogos interativos de computador. De acordo com esses autores, há apenas uma regra a seguir no ambiente extremamente competitivo que vem se estabelecendo nesse setor: entregar sempre o produto mais novo, mais eficiente e em menor tempo.

Os jogos tridimensionais produzidos hoje em dia para computadores domésticos vêm atingindo um grau cada vez maior de realismo, sendo considerados como aplicações de RV semi-imersivas (Bernardes-Junior et al., 2004). Numa busca pela obtenção de efeitos audiovisuais cada vez mais realísticos em microcomputadores domésticos, os jogos têm contribuído para a evolução tecnológica, especialmente quanto ao barateamento do hardware gráfico e ao desenvolvimento de novas técnicas de renderização, animação e simulação, dentre outras (Brooks, 1999).

Em função da competição mercadológica, o processo de desenvolvimento de jogos tridimensionais tem demandado a utilização de avançadas técnicas de computação gráfica, interação e simulação na forma de bibliotecas (Cal3D, 2004; OGRE3D, 2004; OpenSteer, 2004; RenderWare, 2004; Tokamak, 2004; Terdiman, 2003) e kits de desenvolvimento (Id Software, 2004; Epic Games, 2004; Fly3D, 2004; Valve Corporation, 2004; Berger, 2002) que são conhecidos como *Game Engines* (Motores Gráficos).

Através dessa massa de tecnologia disponível, o processo de produção dos jogos tridimensionais tem influenciado fortemente o processo de produção das aplicações de RV, principalmente no que diz respeito à utilização cada vez mais freqüente de Motores Gráficos. Esse tipo de componente de software, originalmente projetado para dar suporte à programação de jogos, tem sido utilizado com sucesso no desenvolvimento de sistemas de RV, pois um motor gráfico encapsula as funcionalidades dos vários subsistemas que gerenciam o ambiente virtual sob diversos aspectos.

A adoção desse tipo de componente numa aplicação de RV permite que os desenvolvedores dessa aplicação utilizem os serviços oferecidos pelo motor e preocupem-se apenas com a implementação das características particulares dos ambientes utilizados em situações específicas dessa aplicação. Além disso, os motores gráficos são perfeitamente utilizáveis em sistemas de RV de baixo custo, uma vez que esses sistemas dispõem praticamente dos mesmos recursos de que um jogo tridimensional dispõe, sendo ainda compostos pelos mesmos subsistemas (Elias, 2002; Leite-Júnior et al. 2002; Almendra et al. 2002).

Um motor gráfico utilizável no desenvolvimento de aplicações de RV geralmente disponibiliza os seguintes serviços: renderização, detecção de colisões, som tridimensional e suporte a linguagens de script. Esses serviços básicos dão suporte aos principais aspectos

de uma aplicação de RV e podem ser oferecidos na maioria das plataformas em que essas aplicações são executadas, além de funcionarem como base para a implementação de outros serviços.

No contexto de desenvolvimento de aplicações de RV, o serviço mais básico que um motor deve prover é o estabelecimento de uma interface com os dispositivos de entrada e saída da plataforma-alvo. Para uma plena utilização do motor gráfico adotado, há outros importantes requisitos a considerar, tais como: disponibilidade de documentação; facilidade de aprendizado; capacidades de configuração, customização e expansão; e abstração das tecnologias usadas na implementação do motor.

As características do motor adotado podem acarretar problemas aos sistemas com ele desenvolvidos, atrelando-os fortemente à tecnologia utilizada em seu desenvolvimento ou a alguma plataforma específica (Maia et al., 2003). Além disso, o motor pode apresentar uma documentação insuficiente, dificultando seu aprendizado e utilização. Assim, a escolha de um motor inadequado para o desenvolvimento de uma aplicação pode limitá-la sob diversos aspectos, caso esse motor não ofereça um serviço importante para a aplicação ou não possua mecanismos para a implementação de novos serviços.

Segundo Bierbaun et al. (2001), um ambiente de desenvolvimento para aplicações de RV deve ser aberto, portável, expansível e fácil de aprender, além de fornecer um conjunto mínimo de funcionalidades. Além disso, a utilização de um motor gráfico cujo código fonte não se encontra disponível para modificações consiste num problema de manutenção das aplicações desenvolvidas. Isso porque o desenvolvimento do motor gráfico adotado pode ser descontinuado ou porque alguma funcionalidade importante para a aplicação seja removida do mesmo. Por outro lado, a disponibilidade de código fonte permite que desenvolvedores mais experientes realizem customizações num motor gráfico para adequá-lo às suas necessidades específicas sem que, para tanto, tenha de construir um motor próprio ou aprender a utilizar um motor diferente.

1.2 Objetivos

Este trabalho estuda e avalia a utilização de motores gráficos no desenvolvimento de aplicações de Realidade Virtual, com o objetivo de propor uma arquitetura para suportar a construção de motores gráficos adequados à utilização no desenvolvimento de jogos e de

aplicações de RV. Essa arquitetura visa contornar as limitações que um motor gráfico geralmente possui, quando utilizado como componente na construção desses tipos de aplicações. Um objetivo adicional deste trabalho é desenvolver, a partir da arquitetura proposta, um motor gráfico portável, flexível e configurável, funcionando como uma camada de abstração de tecnologias, APIs de baixo nível e plataformas.

1.3 Estrutura da Dissertação

A presente dissertação está estruturada em sete capítulos. No Capítulo 2, faz-se um levantamento sobre os principais aspectos de uma aplicação de Realidade Virtual e das soluções disponíveis para o desenvolvimento desse tipo de aplicação. No Capítulo 3, com o objetivo de facilitar a análise e a comparação de diferentes motores gráficos, apresenta-se uma divisão estrutural para um motor gráfico genérico. No Capítulo 4, analisam-se, de maneira comparativa, os principais motores gráficos disponíveis atualmente que representam o estado-da-arte desse tipo de componente de software, identificando as principais limitações que esse tipo de componente possui. No Capítulo 5, descreve-se a arquitetura CRAb *Graphics Engine* para motores gráficos, que foi desenvolvida especialmente para contornar as limitações identificadas no Capítulo 4. No Capítulo 6, são apresentados os principais aspectos de implementação dessa arquitetura e os resultados práticos obtidos através de uma bateria de testes realizados com o motor desenvolvido. Finalmente, no Capítulo 7, tecem-se algumas conclusões acerca da arquitetura CRAbGE, apresentam-se os principais resultados demonstrados através da implementação dessa arquitetura e apontam-se possíveis trabalhos futuros.

Capítulo 2

Soluções Para o Desenvolvimento de Aplicações de RV

2.1 Introdução

As aplicações de Realidade Virtual caracterizam-se pelo uso de estímulos sensoriais e pela intensa interação, oferecendo ao usuário a sensação de imersão num ambiente gerado por computador. Dessa maneira, a qualidade dos ambientes utilizados nessas aplicações é determinada pelos vários os aspectos de representação do mundo virtual, pelo nível de interação com usuário e pelas respostas multissensoriais geradas em conseqüência das interações.

Segundo Ponder et al. (2003), os jogos de computador e as aplicações de RV têm-se tornado cada vez mais sofisticados em função dos recentes avanços nas áreas de computação gráfica, simulação, dispositivos e técnicas de interação. Tais avanços possibilitam a popularização dessas tecnologias e a construção de simulações audiovisuais cada vez mais realistas, convidativas e acessíveis para o seu público-alvo. Por outro lado, esse incremento de sofisticação e qualidade culminou tanto numa demanda contínua dessas aplicações quanto no crescimento da complexidade atrelada ao desenvolvimento desses sistemas. Segundo esses autores, existe apenas uma regra para seguir nesse ambiente competitivo: entregar sempre o produto mais novo, mais rápido e em menor espaço de tempo. Sob essa perspectiva, o reuso de componentes de software durante o processo de desenvolvimento de sistemas de RV, além de ser uma opção bastante atraente para a construção dessas aplicações, torna-se imprescindível para acompanhar o ritmo frenético imposto pelo mercado.

Nas próximas subseções, serão apresentados os requisitos básicos para o desenvolvimento de uma aplicação de RV e os componentes de software disponíveis para atender a essas necessidades fundamentais.

2.2 Requisitos Básicos

Numa aplicação RV, o usuário interage intensamente com as entidades presentes num mundo virtual, que podem ser de natureza estritamente artificial ou outros usuários conectados àquele mundo virtual. Independentemente da natureza dessas entidades, praticamente toda interação é decorrente do reconhecimento das entradas do usuário pela aplicação através dos dispositivos adequados, como, por exemplo, o conjunto mouse-teclado, típicos de ambientes de *Desktop*, e como *Joysticks*, *Data Gloves* e dispositivos de *tracking*, próprios para aplicações de RV. Essas entradas são mapeadas nas formas de interação implementadas na aplicação, que determinam o comportamento do usuário no contexto do ambiente virtual. Como resultado desse comportamento, o sistema elabora as respostas sensoriais adequadas às interações, que são percebidas pelo usuário através dos dispositivos de saída e ajudam a envolver o usuário no ambiente. Esse processo de interação é ilustrado pela Figura 2.1.

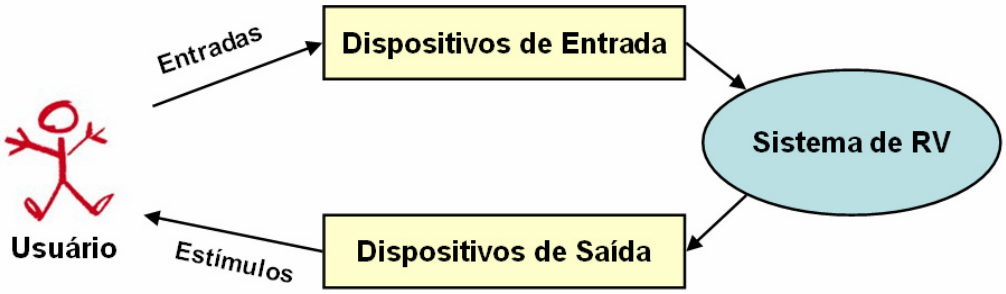


Figura 2.1 : Interação do usuário com um sitema de RV.

A utilização dos dispositivos de entrada disponíveis no ambiente de execução possibilita a comunicação do usuário com o sistema. Portanto, o acesso a tais dispositivos é um requisito fundamental para a construção de uma aplicação de RV. Por outro lado, também é preciso que a aplicação seja capaz de produzir os estímulos sensoriais adequados, além de transmitir esses estímulos como resposta ao usuário através dos dispositivos de saída apropriados. É importante salientar que, como a tecnologia de hardware utilizada em RV encontra-se em expansão, é interessante que o sistema possa acomodar novos dispositivos. Além disso, o sistema deve responder rapidamente às entradas do usuário, apresentando uma taxa de 10Hz, no mínimo, entre a entrada do usuário e a reposta correspondente (Shaw et al., 1993). A resposta háptica, por exemplo, requer taxas de pelo

menos 500Hz (Conrad, 2002). Já a resposta visual, deve ser gerada a uma taxa de pelo menos 20Hz, suficiente para gerar a ilusão de movimento no olho humano (Rogers & Adams, 1990).

Dependendo do nível de imersão almejado, uma aplicação de RV pode utilizar todos os 5 sentidos ou apenas um subconjunto destes na geração das respostas sensoriais. Geralmente, é possível utilizar pelo menos repostas audiovisuais, presentes em praticamente todo ambiente de execução desse tipo de aplicações, incluindo plataformas de *Desktop*, onde a utilização dessas duas respostas e o conjunto mouse-teclado-*joystick* com dispositivos de entrada em ambientes semi-imersivos caracteriza a Realidade Virtual de *Desktop*.

A elaboração da resposta visual em tempo-real requer a utilização de técnicas de renderização gráfica em tempo-real, de maneira a garantir uma taxa de exibição interativa. A construção da resposta auditiva demanda, no mínimo, uma interface para a reprodução de sons pré-amostrados. É preferível utilizar um sistema de simulação sonora capaz de reproduzir efeitos do ambiente (Loki Software, 2000) e para assegurar a qualidade da resposta auditiva.

É importante notar que, para que a aplicação de RV determine quando determinada interação deve ocorrer, é necessário detectar interferências entre os objetos do mundo virtual. Assim, para implementar situações de interação e simulação, as aplicações de RV precisam detectar a interseção entre os modelos, geralmente poligonais, do mundo virtual. Em Rodrigues et al. (2002), por exemplo, os autores realizam a simulação de um procedimento cirúrgico, onde o modelo de uma língua é deformado em conseqüência do contato com o instrumento utilizado nesse procedimento. Para tanto, verificam quais faces do modelo estão em contato com o instrumento através de um algoritmo próprio para a detecção de colisões.

Uma porta automática, por exemplo, deve abrir-se apenas quando existe algum avatar posicionado em frente à mesma e fechar-se apenas quando não há avatares nessa situação de proximidade. Esses exemplos servem para ilustrar a importância das técnicas para a detecção de colisões no contexto do desenvolvimento de ambientes virtuais, onde esse tipo de tarefa é fundamental para a implementação de interações realistas num ambiente virtual (Figueiredo et al., 2002).

Por outro lado, é importante utilizar mecanismos de configuração e prototipagem durante os processos de desenvolvimento e de manutenção em sistemas de grande porte, como as aplicações de RV. A utilização de linguagens de scripts é a solução ideal para esse tipo de situação, conferindo grande flexibilidade às aplicações, que passam a ser desenvolvidas em porções compiladas e porções interpretadas.

A renderização em tempo-real, a detecção de colisões, a utilização de som espacial e a utilização de linguagens de script consistem nos requisitos básicos à construção de uma aplicação de Realidade Virtual. Sob o ponto de vista de reuso de componentes de software, é interessante que cada um desses aspectos seja delegado a um subsistema específico, utilizado para facilitar o desenvolvimento do sistema de RV como um todo.

Assim, as subseções seguintes destinam-se à discussão dos principais tópicos relativos a esses aspectos, considerados básicos para a construção de um sistema de RV.

2.2.1 Renderização em Tempo-Real

No contexto da renderização em tempo-real, um ambiente virtual é representado por uma cena tridimensional composta por uma coleção de pontos de vista, representados por câmeras virtuais, e conjunto de objetos (Maia et al., 2004). Assim, para renderizar a cena como se estivesse sendo vista pelo avatar do usuário, a aplicação de RV especifica uma câmera câmera virtual de acordo com o posicionamento daquele objeto no ambiente virtual, e a utiliza como o ponto de vista corrente para a renderização da cena através de seu sistema de renderização, que efetivamente envia os objetos visíveis ao *frame buffer*.

Os objetos do mundo virtual podem ser classificados, quanto à sua natureza, em **estáticos**, que atuam como decoração e delimitadores do espaço na cena, e **dinâmicos**, aos quais está associado um comportamento no contexto da aplicação de RV. Isso significa que os objetos de natureza dinâmica podem sofrer deformações em conseqüência de alguma animação interessante para o envolvimento do usuário com o mundo virtual. Além disso, estão sujeitos a modificações em suas posições, orientações e dimensões durante uma visita ao mundo virtual, podendo, inclusive, serem removidos ou inseridos na cena em tempo de execução. Já os objetos de natureza estática, não menos importantes, compõem o cenário em que o usuário interage com os objetos dinâmicos, sendo utilizados principalmente para

ambientar a situação desejada. Assim, num mundo virtual realista, é de se esperar que os objetos estáticos apareçam em grande quantidade devido ao nível de detalhe desejado.

Para facilitar o processamento das cenas tridimensionais pela aplicação de RV, a manipulação dos objetos dinâmicos e das câmeras presentes numa cena fica mais naturalmente representada se tais entidades forem organizadas numa hierarquia. Isso é conveniente em uma série de situações onde a movimentação de uma entidade implica na mudança de posicionamento de outras entidades a ela associadas, o que acontece freqüentemente. Por exemplo, uma câmera e um livro podem estar hierarquicamente associados a um avatar, de forma que, quando esse avatar se move pelo ambiente, o livro move-se com ele e a câmera acompanha seu movimento. Por esse motivo, o sistema de renderização deve oferecer suporte à definição e à manutenção de uma estrutura hierárquica composta por objetos e câmeras. Além disso, é conveniente que nomes possam ser atribuídos aos agrupamentos hierárquicos de uma cena, permitindo que esses agrupamentos sejam manipulados de uma maneira mais intuitiva.

A renderização de modelos tridimensionais através das principais API gráficas (Segal & Akeley, 2004; DirectX, 2004) é suportada por malhas poligonais que oferecem uma representação dos objetos através de suas fronteiras ou "cascas", formadas a partir de polígonos convexos, geralmente triângulos. Isso significa que os modelos dos objetos no mundo virtual devem ser especificados em termos malhas poligonais, residentes na memória, quando do envio ao *frame buffer*. Atualmente, é possível utilizar a memória de alto desempenho das placas aceleradoras gráficas modernas para armazenar modelos poligonais, quando possível. Assim, a aplicação evita o reenvio desses modelos ao hardware gráfico a cada quadro, resultando numa melhora considerável no desempenho, especialmente no caso de modelos não deformáveis. Tendo em vista esse tipo de otimização, os modelos poligonais presentes numa cena geralmente devem ser definidos através de uma interface disponibilizada pelo subsistema de renderização.

Quanto mais complexos forem os modelos tridimensionais que compõem um ambiente virtual, mais demorado será o processo de renderização. Assim, a otimização desse processo torna-se imprescindível quando a plataforma–alvo da aplicação não possuir hardware gráfico adequado à renderização massiva de polígonos, amenizando as exigências sobre o hardware gráfico. Através das otimizações realizadas pelo subsistema de

renderização, o motor gráfico possibilita a utilização de ambientes virtuais detalhados e tem sua portabilidade aumentada sob esse aspecto.

Essas otimizações podem ser obtidas através da integração de técnicas baseadas em software para acelerar a renderização. Hoffman (2000) classifica esse tipo de técnicas em três grupos principais: **Processamento de Geometria**, que diminui o número de primitivas usadas na descrição de malhas complexas; **Renderização Baseada em Imagem**, que substitui modelos geométricos complexos por polígonos contendo uma imagem desses modelos; e **Remoção Seletiva de Objetos** (*culling*), que, dado um ponto de vista da cena, suprime a pintura de objetos que não contribuem para a imagem sintetizada. Diversas métricas podem ser usadas para estabelecer comparações entre as diversas técnicas de renderização existentes. Geralmente, essas métricas envolvem o número de quadros por segundo e o número de triângulos que as técnicas de aceleração da renderização são capazes de suportar, além dos pré-requisitos para a utilização dessas técnicas, como hardware gráfico e memória.

O subsistema de renderização deve manter um equilíbrio entre o desempenho e a qualidade visual das cenas renderizadas. Para obter bons resultados visuais, esse subsistema geralmente disponibiliza uma interface para a definição dos materiais que descrevem como os diversos efeitos visuais, como iluminação e filtros de textura, são aplicados aos objetos presentes na cena (CrystalSpace, 2004; OGRE3D, 2004; Fly3D, 2004; RenderWare, 2004; Maia et al., 2004).

Atualmente, a nova geração de dispositivos gráficos inclui processadores de vértices e de fragmentos programáveis, oferecendo suporte ao desenvolvimento de efeitos gráficos personalizados de alta qualidade antes inviáveis (ATI Technologies, 2004; NVIDIA, 2004). Dessa maneira, programas definidos pela aplicação em uma linguagem de baixo nível ou de alto nível, como OpenGL Shading Language (Akeley & Segal, 2004), High Level Shading Language (DirectX, 2004) ou Cg (Mark et al., 2003). Linguagens desse tipo, sejam de baixo ou de alto nível, serão referenciadas como *Shading Languages* ou simplesmente linguagens de *shading*, no restante do presente trabalho. Mais especificamente, programas destinados à execução na unidade de processamento de vértices são capazes de processar os vértices recebidos como entrada e seus atributos relacionados, sendo referenciados na literatura como *vertex shaders*. Da mesma maneira, programas destinados à execução na

unidade de processamento de fragmentos são capazes de processar fragmentos e seus valores associados, sendo referenciados na literatura como *pixel shaders*, devido a produzirem efeitos por *pixel* na imagem final.

Assim, programas escritos numa linguagem de *shading* podem substituir o processamento tradicional dos vértices durante a renderização, ou seja, as etapas de transformação, iluminação e geração de coordenadas de textura. Além disso, esses programas são capazes de realizar operações por fragmento, como mapeamento de textura, adição de cores e aplicação de névoa (fog). Isso oferece uma abordagem muito mais flexível e eficiente para a criação de efeitos do que a abordagem tradicionalmente presente nas APIs de renderização, referenciada na literatura como *fixed function*, onde efeitos mais elaborados requerem a combinação de uma seqüência de passos nos quais os modelos têm de ser pintados novamente a cada passo.

A utilização das características programáveis dos dispositivos gráficos é uma funcionalidade muito interessante para o sistema de renderização utilizado pela aplicação de RV, pois permite um maior envolvimento do usuário com o mundo virtual através da utilização de efeitos de alta qualidade. Por outro lado, é desejável que seja oferecida a flexibilidade de programar essas etapas da pipeline de baixo nível de acordo com qualquer uma das linguagens existentes na API de renderização. Além disso, é interessante que o sistema de renderização possa checar a presença desses recursos no hardware gráfico, sendo capaz de oferecer algum método alternativo quando esse tipo de funcionalidade não é suportado.

Durante a renderização de baixo nível, todos os estados internos necessários à obtenção de determinado efeito num objeto podem alterados com base no material que aquele objeto utiliza. Isso também permite que um sistema de renderização reordene os objetos antes de efetivar a pintura da cena, de maneira a diminuir o número de trocas de estados onerosos, como os relativos à transparência e à aplicação de texturas.

Não é difícil perceber a notável complexidade do subsistema de renderização, que demanda a utilização de estruturas de dados complexas em algoritmos delicados, além de APIs gráficas intricadas como OpenGL (Segal & Akeley, 2004) e DirectX Graphics (DirectX, 2004). Por outro lado, características como modularidade e portabilidade também são extremamente importantes nesse subsistema. Dessa maneira, as etapas de projeto e

implementação do subsistema de renderização em tempo-real para a utilização em sistemas de freqüentemente requerem um enorme esforço de desenvolvimento.

2.2.2 Detecção de Colisões

Para efetivamente detectar colisões entre os modelos poligonais que compõem uma cena tridimensional, não é suficiente aplicar a matemática envolvida nos cálculos de interseção. É preciso adequar esses algoritmos ao modelo numérico limitado e propenso a erros que é encontrado nos computadores (Dawson, 2004; Hollasch, 2004), de maneira a tornar esses algoritmos, robustos na teoria, robustos também na prática. Isso significa que elaborar um algoritmo para detectar a interseção entre um segmento de reta e um triângulo não é tão simples quanto parece, tornando o reuso de técnicas pré-existentes uma solução muito atraente para o desenvolvimento de aplicações de RV.

Segundo Figueiredo et al (2002), a implementação de interações realistas nos ambientes virtuais demanda algoritmos para o cálculo de interseções exatas entre os modelos da cena, geralmente suportados por malhas poligonais. Assim, soluções aproximadas nem sempre são uma alternativa viável quando se deseja desenvolver um sistema de RV que atenda a determinados pré-requisitos de realismo. Além da exatidão, as técnicas para o cálculo de interseções devem apresentar outras qualidades indispensáveis para a sua plena utilização nos sistemas de RV. Segundo Maia et al. (2003), essas técnicas devem ser:

- **Robustos**, pois precisam lidar com um grande número de polígonos, nas mais variadas posições e orientações;
- **Eficientes**, pois esses testes devem ser executados rapidamente de maneira a causar o menor impacto possível no desempenho do sistema. Isso porque, quando os testes de interseção são necessários, são realizados diversas vezes antes da renderização de cada quadro; e
- **Precisos,** no sentido de detectar apenas interseções válidas e retornar informações úteis para a camada de aplicação.

Como esses algoritmos de colisão devem lidar com os objetos dinâmicos presentes numa cena tridimensional, é de fundamental importância que mudanças nas posições, orientações e dimensões dos modelos poligonais sejam consideradas nesses algoritmos. Além disso,

alguns objetos do mundo virtual, como os avatares, por exemplo, estão sujeitos à deformação em conseqüência de animações, por exemplo. Esse cenário demanda técnicas de colisão que considerem a atualização das posições dos vértices presentes numa malha poligonal (Van der Bergen, 1997). Por outro lado, é interessante que essas técnicas permitam o compartilhamento de malhas por diversos objetos idênticos posicionados no mundo virtual, como as cadeiras de um auditório, por exemplo. Isso evita o desperdício de memória e de processamento necessários para acomodar várias cópias do mesmo objeto.

Considerar, ou não, a movimentação dos objetos envolvidos nos testes de interseção é outro aspecto importante dos algoritmos de colisão. De acordo com Sébastien (2002), os testes de interseção podem ser classificados, quanto à natureza dinâmica das entidades envolvidas, em:

- **Estáticos**, que calculam a interseção apenas entre entidades estacionárias, descartando informações sobre o movimento dessas entidades;
- **Dinâmicos,** que calculam a interseção entre entidades sujeitas ao movimento. Dadas às situações iniciais e finais das entidades que se submetem ao movimento, esse tipo de testes é capaz de detectar, além da existência de interseções, o momento em que ocorreram os contatos; e
- **Pseudo-dinâmicos**, que aproximam a interseção entre objetos em movimento através de uma série de testes estáticos. Embora esse tipo de teste esteja sujeito a erros devido as aproximações adotadas, é suficiente para um grande número de situações.

Os objetos num mundo virtual encontram-se distribuídos no espaço de acordo com posições, orientações e escalas. As técnicas de detecção de colisão geralmente valem-se da distribuição espacial desses objetos como um artifício para evitar rapidamente objetos que não podem estar colidindo. Essa heurística, conhecida na literatura como **coerência espacial**, utiliza estruturas de dados especiais para aproximar os modelos por uma subdivisão espacial conveniente para a rejeição de objetos e primitivas que não se interceptam (Figueiredo et al., 2002).

Num ambiente virtual onde os objetos movem-se suavemente, as técnicas de detecção de colisões podem tomar vantagem das pequenas alterações nas posições e orientações desses objetos entre os quadros exibidos. Essa heurística, conhecida na

literatura como **coerência temporal**, mantém uma estrutura de dados com as interseções anteriormente calculadas com o objetivo de acelerar o cálculo das próximas interseções.

Quando a aplicação de RV utiliza técnicas de interação pouco sofisticadas, é satisfatório determinar se um objeto está na iminência de interseção com outro. Esse tipo de situação pode ser resolvido através de testes de interseção entre volumes simplificados, como esferas e *bounding boxes*, que envolvem os modelos geométricos. Essa é uma alternativa simples e eficiente para um grande número de situações, evitando efetuar testes computacionalmente caros entre modelos complexos. Assim, é interessante que esses testes entre volumes estejam disponíveis para a aplicação de RV. Por outro lado, é conveniente que diversos tipos de volumes possam ser utilizados nesses testes, visto que alguns são mais adequados para certos propósitos. As *bounding boxes* orientadas (*Oriented Bounding Boxes*, OBBs) são mais adequadas para aproximar o cálculo de interseções entre modelos poligonais do que esferas ou *bouding boxes* alinhadas com os eixos (*Axis-Aligned Bounding Boxes*, AABBs) (Gottschalk et al., 1996), por exemplo.

Numa aplicação de RV, é conveniente que o usuário seja capaz de selecionar um objeto através do "toque" na imagem daquele objeto produzida pelo sistema de renderização. Esse tipo de operação é referenciado na literatura como *picking* (Rogers & Adams, 1990), e caracteriza-se pela obtenção do objeto mais próximo da câmera virtual que intercepta um raio partindo do foco da câmera em direção ao ponto selecionado pelo usuário na viewport, agora em coordenadas de mundo. Esse processo de interseção é ilustrado pela Figura 2.2. A operação de *picking* permite que o usuário realize tarefas de maneira intuitiva utilizando os objetos do mundo virtual, como, por exemplo, posicionar os móveis num ambiente virtual ou iniciar uma interação com outro usuário.

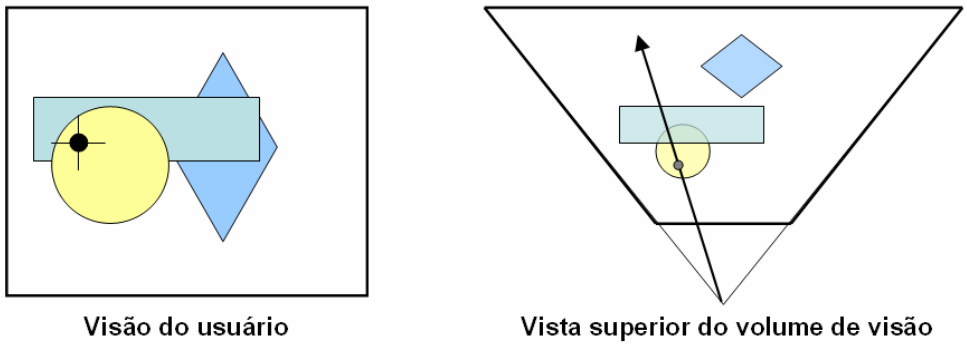


Figura 2.2: Operação de Picking.À esquerda, seleciona-se um ponto da viewport. À direta, o raio resultante, visto de cima, intercepta a esfera amarela.

Noutro tipo de situações, é interessante que a camada de aplicação possa selecionar objetos numa região descrita por um volume convexo. Isso permite implementar técnicas simples de interação com facilidade. Por outro lado, técnicas de interação e comunicação que tratam um conjunto de objetos podem ser tornadas mais eficientes, limitando os objetos processados por esse tipo de técnicas àqueles presentes numa determinada região de influência no mundo virtual. Quando um míssil explode durante um treinamento virtual de guerra, por exemplo, apenas os objetos no raio de ação do míssil precisam ser fragmentados.

Algumas situações requerem informações mais precisas sobre o contato dos objetos, determinadas através de testes computacionalmente caros entre modelos poligonais complexos. Geralmente, os polígonos que compõem um modelo são testados contra os que compõem o outro modelo para que seja determinada uma lista de pares de faces que estão em contato. Esses pares de faces podem ser usados, por exemplo, para gerar pontos de contato entre os modelos, de maneira a manter restrições de interpenetração durante uma simulação de corpos rígidos articulados (ODE, 2004). As principais funcionalidades de um sistema de detecção de colisão são sumarizadas, em alto nível, na Figura 2.3.

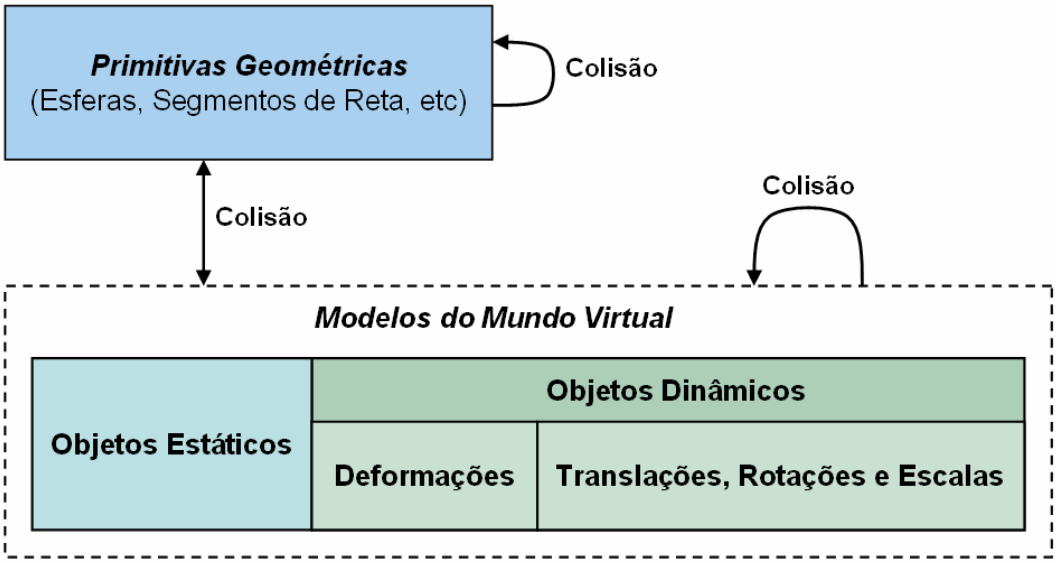


Figura 2.3: Os testes de interseção requeridos pelas aplicações de RV.

2.2.3 Som Espacial

Atualmente, é possível reproduzir as características sonoras de um ambiente virtual através de APIS de baixo nível que implementam modelos de simulação sonora (Loki software, 2000; Firelight Technologies, 2004; DirectX, 2004). Nesses modelos, é possível simular algumas das principais características dos sons num ambiente tridimensional:

- **Atenuação**, a diminuição da intensidade do sinal sonoro proporcional à distância entre o ouvinte e a fonte sonora devido à absorção de energia pelo meio de propagação do som. Distâncias de referência são usadas para descrever a atenuação;
- **Efeitos Doppler**, a variação aparente na no comprimento de onda do som em conseqüência do movimento da fonte sonora, do ouvinte ou ambos; e
- **Diferenças Perceptuais Interaurais**, as diferenças no som percebido pelos ouvidos esquerdo e direito em conseqüência do posicionamento das fontes sonoras em relação ao ouvinte;

Essas características são fundamentais para que o usuário seja capaz de localizar as fontes sonoras no espaço, além de perceber a natureza do meio em que o som se propaga. Assim, esses aspectos são considerados indispensáveis para a simulação de som num ambiente 3D. Atualmente, o modelo utilizado pelas APIs definem um ouvinte único que possui posição, orientação e uma velocidade. Esse ouvinte está imerso num ambiente contendo fontes sonoras que possuem amostras de som para reprodução, posição e velocidade como atributos básicos. Outros atributos descrevem somo o som é distribuído no espaço. Quando a fonte sonora não possui uma direção, o mesmo volume é percebido a uma dada distância independentemente da direção. Por outro lado, dois cones de som, descritos por dois ângulos entre a direção do som, podem ser utilizados nesses modelos para descrever a atenuação de sons direcionais de acordo com o ângulo entre a direção da fonte sonora e o vetor deslocamento da origem do som à posição do ouvinte. Quanto o ouvinte encontra-se no cone interno, não há atenuação. Na zona de transição entre o cone interno e o externo, o volume do som percebido decresce à medida que o ouvinte se afasta do cone externo, atingindo um valor constante na região exterior ao cone externo. Esse modelo de atenuação é ilustrado pela Figura 2.4.

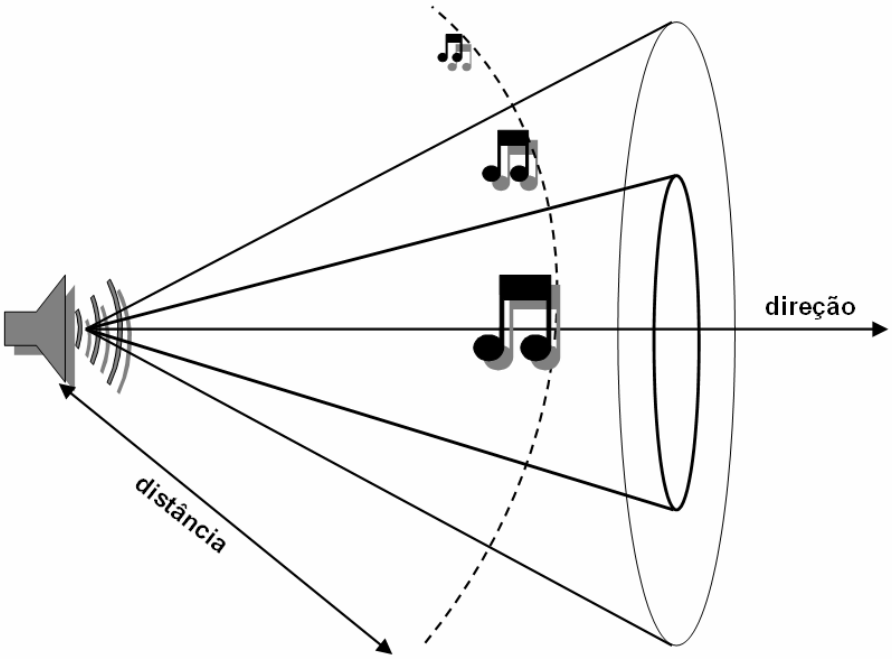


Figura 2.4 : Atenuação do som em fontes direcionais utilizando os cones de som.

Por outro lado, a propagação do som possui outras características importantes que complementam o realismo do ambiente sonoro, incrementando assim a sensação de imersão por parte dos usuários (Creative Labs, 2004). Isso permite que o usuário perceba outros efeitos sonoros dentro de um mundo virtual:

- **Reverberação**, o som percebido pelo ouvinte a partir de uma fonte sonora é o resultado da combinação dos sons que chegam diretamente ao ouvido com os que chegam após as reflexões no ambiente. Isso causa implicações na duração e do volume do sinal sonoro, ajudando o usuário a perceber as dimensões do ambiente e as qualidades reflexivas das paredes;
- **Obstrução**, ocorre quando algum objeto se interpõe entre a fonte sonora e o ouvinte. Assim, o som que chegaria diretamente ao ouvinte pode ser percebido em conseqüência da difração e da transmissão através daquele objeto. Isso implica numa atenuação apenas da porção do som que chegaria diretamente ao ouvinte;
- **Oclusão**, ocorre quando a fonte sonora encontra-se separada do ouvinte por uma parede, de maneira que o som é transmitido através desse obstáculo. Além disso, o som pode chegar ao ouvinte através de qualquer abertura entre a fonte sonora

e o ouvinte. Assim, o som é afetado pela difração e transmissão antes que alcance a porção do ambiente em que o ouvinte está. Além disso, o elemento do ambiente que passa a emitir o som é a abertura através da qual o som passa a ser transmitido. Como resultado, as componentes reflexivas daquele som são mais atenuadas.

Esses modelos sofisticados são capazes de descrever as propriedades sonoras do ambiente que circunda o ouvinte, incluindo novas características perceptuais que devem ser levadas em consideração quando do desenvolvimento dos sistemas de RV. A utilização desses fatores de representação sonora demanda maiores cuidados durante a modelagem dos ambientes virtuais para que uma simulação sonora de qualidade seja oferecida para os visitantes desses ambientes. Além disso, são necessários alguns testes de detecção de colisões para que as obstruções e as oclusões sejam simuladas adequadamente, visto que esses efeitos são controlados por um vasto conjunto de atributos nas principais APIs de áudio espacial (Creative Labs, 2004; DirectX, 2004) e vários desses atributos são dependentes do posicionamento das fontes sonoras em relação ao ouvinte.

Apesar da qualidade obtida por esses modelos sonoros, os mesmos possuem algumas limitações. Tsingos et al. (2003) destaca que, como as fontes sonoras geralmente são consideradas pontuais, é necessário um grande número dessas fontes para a obtenção de efeitos mais realísticos. Por outro lado, a maioria das APIs disponíveis não é portável e limita-se à reprodução de sons pré-amostrados, demandando a utilização de outros componentes de software para amostrar e processar áudio. Uma conferência envolvendo os usuários de um ambiente virtual em rede, por exemplo, requer a captura das vozes dos usuários em tempo de execução além da transmissão dessa informação.

2.2.4 Linguagens de Script

No processo de desenvolvimento tradicional dos programas de computador, os sistemas são descritos através de código de alto nível escrito numa linguagem de programação, e, após as etapas de compilação e montagem, o código de alto nível é transformado em código de máquina. Ao final desse processo, a aplicação está pronta para ser executada no computador, que interpreta o código de máquina durante a execução do programa, como se estivesse seguindo uma receita de bolo. Esse processo é ilustrado pela Figura 2.5.

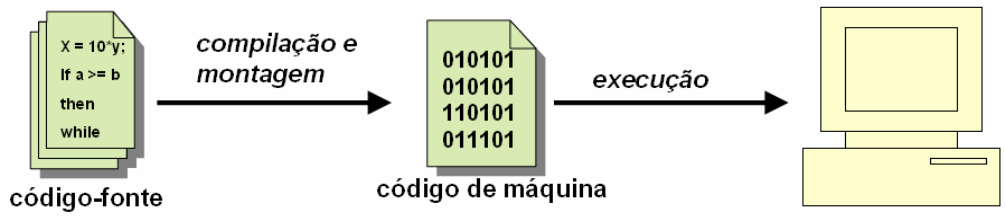


Figura 2.5: Etapas da construção de uma aplicação.

O processo tradicional possui a desvantagem de recompilar o código da aplicação quando houver alguma mudança, diminuindo a interatividade durante as etapas de edição e validação durante o desenvolvimento de aplicações de grande porte como os sistemas de RV. Esse problema pode ser contornado através da utilização de uma linguagem de script como um módulo adicional da aplicação. Assim, o código de máquina, já compilado, torna-se capaz de interpretar scripts em tempo de execução, capazes de alterar o estado do sistema, como ilustrado pela Figura 2.6.

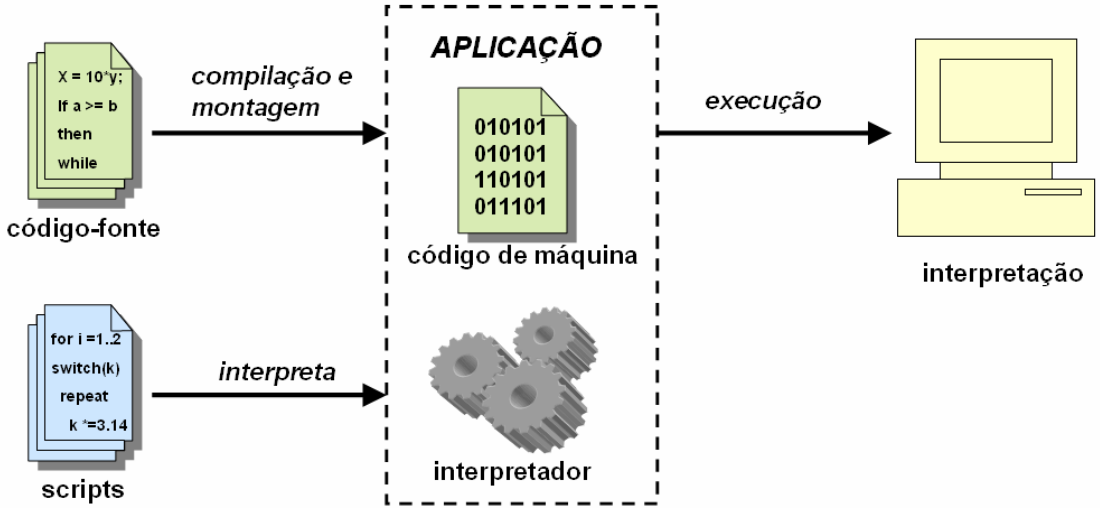


Figura 2.6: Interpretação de scripts, em alto nível, pelo código já compilado.

Isso significa que partes da aplicação podem ser escritas numa linguagem de programação de alto nível, sem que seja necessário reconstruir código de máquina a cada modificação na aplicação, conferindo ao sistema grandes capacidades de prototipagem, expansão e configuração (Ierusalimschy et al., 2003; Dawson, 2002; Thomas et al., 2004). Os tradicionais arquivos de configuração utilizados por um sistema, por exemplo, podem ser substituídos por scripts capazes de realizar configurações inteligentes após alguma espécie de processamento. Essas características tornam o uso de linguagens de scripts uma

ferramenta poderosa para a construção de sistemas de RV, pois conferem grande flexibilidade ao desenvolvedor (Maia et al., 2003).

A utilização de scripts como parte de sistemas em tempo-real implica numa queda de desempenho devido à interpretação do código. Apesar disso, num grande número de situações, é possível utilizar scripts como clientes do código compilado através do *binding* sem maiores perdas no desempenho (Dawson, 2002; Maia et al., 2003). Numa chamada a uma rotina de *Path Finding* através de um script, por exemplo, o gargalo computacional comumente encontra-se no algoritmo de busca implementado em código de máquina e não na interpretação do script.

Para que os scripts sejam capazes de alterar o estado de uma aplicação, é necessário que a aplicação forneça um módulo adicional ao interpretador de scripts, conhecido na literatura como *binding*, contendo o código responsável pela de ligação entre o ambiente de execução dos scripts e a camada de aplicação. O *binding* geralmente disponibiliza uma interface de programação similar à utilizada pela aplicação, incluindo constantes, funções e classes de objetos, quando possível.

A utilização de uma linguagem de scripts num sistema requer, no mínimo, a interpretação de scripts a partir de arquivos. Após a execução de um script, as variáveis afetadas internamente no interpretador podem ser utilizadas para realimentar a aplicação. Um script de configuração de um editor de imagens, por exemplo, pode ser usado para alterar o diretório utilizado para arquivos temporários.

Além disso, no contexto de desenvolvimento e configuração de uma aplicação, é interessante que scripts definidos interativamente pelo usuário sejam interpretados em tempo de execução. Para isso, é fundamental que o interpretador de scripts seja capaz de processar fragmentos de código residentes na memória. Por outro lado, alguns scripts precisam ser executados diversas vezes durante a execução da aplicação. Assim, é conveniente que um script muito utilizado pela aplicação possa ser mantido em memória de maneira a evitar que seja recompilado a cada execução, evitando gargalos de compilação.

2.3 Soluções de Baixo Nível

Para uma análise mais clara das soluções para o desenvolvimento de sistemas de RV, as mesmas serão classificadas em dois grupos principais analisados separadamente:

- **Baixo nível**, são soluções que ajudam no desenvolvimento de um determinado aspecto de um sistema. Ao utilizar esse tipo de soluções, o desenvolvedor geralmente se vê obrigado ora a resolver problemas inerentes ao aspecto do sistema, ora a integrar várias soluções de baixo nível para implementar o aspecto como um todo; e
- **Alto nível**, que tratam problemas como um todo e minimizam o esforço dos desenvolvedores que utilizam esse tipo de soluções. Geralmente, integram soluções de baixo nível para resolver um domínio mais específico do problema.

A utilização de soluções de baixo nível dificulta a construção de aplicações por desenvolvedores inexperientes, além de incrementar o esforço de desenvolvimento atrelado ao sistema. Mas, apesar de serem uma opção mais trabalhosa para o desenvolvimento de um sistema, as soluções de baixo nível possuem uma série de vantagens bastante atraentes, principalmente para pesquisadores que desejem adaptar ou criar novas técnicas. O desenvolvedor tem um maior controle de como o problema é resolvido, podendo criar componentes de software adequados para situações específicas, obtendo, assim, otimizações praticamente impossíveis de se conseguir quando uma solução de alto nível é usada. Essa é uma característica muito importante desse tipo de soluções, pois permite o desenvolvimento de novas abordagens para um problema, o que é essencial para pesquisadores.

As subseções seguintes destinam-se à apresentação das principais soluções de baixo nível disponíveis para o desenvolvimento dos quatro aspectos fundamentais de um sistema de RV: Renderização em Tempo-Real, Detecção de Colisões, Som Espacial e Utilização de Linguagens de Script.

2.3.1 Renderização em Tempo-Real

Vários algoritmos podem ser usados para acelerar a renderização e manter uma taxa interativa de quadros por segundo em aplicações de computação gráfica, como os sistemas de RV. Em Hoffman (2000), o autor afirma que taxas a partir de 15 quadros por segundo são consideradas interativas e que taxas maiores, em torno de 60Hz, são fundamentais para uma interação mais agradável com o usuário. Esse autor classifica as estratégias para a aceleração da renderização em dois tipos: baseadas em hardware e baseadas em software.

As técnicas baseadas em hardware visam pintar um maior número de polígonos em menor tempo, e, geralmente, não são economicamente viáveis. As técnicas baseadas em software consistem no emprego de algoritmos para tornar a renderização mais eficiente e são requeridas quando a aquisição de hardware gráfico é uma restrição proibitiva. Além disso, técnicas desse tipo não são mutuamente exclusivas, permitindo a combinação de diversas estratégias para atingir uma alta taxa de quadros por segundo.

As técnicas baseadas em software são, portanto, indispensáveis à portabilidade das aplicações de RV, permitindo sua execução sob diversas configurações de hardware gráfico. Hoffman (2002) classifica as técnicas baseadas em software para a aceleração da renderização em duas categorias principais:

- Redução do número de polígonos exibidos num quadro através de técnicas como nível de detalhe e remoção seletiva de objetos; e
- Projeto e implementação de uma pipeline gráfica eficiente, que visa otimizar a pintura dos polígonos enviados ao *frame bufffer*;

A renderização efetiva de uma cena tridimensional requer a utilização de uma biblioteca gráfica (Segal & Akeley, 2004; DirectX, 2004). Essa biblioteca tanto pode prover acesso ao hardware gráfico disponível atuando como um *driver*, ou implementar a pipeline gráfica inteiramente em software (Magic Software, 2004; CrystalSpace, 2004).

O processo de renderização através de *hardware* gráfico pode ser visto como um sistema formado por duas unidades de processamento: a unidade principal do computador (CPU), responsável pela execução das aplicações em geral, e a unidade de processamento gráfico (GPU). Durante a renderização de um quadro, a CPU comunica-se com a GPU através do *driver* que opera o dispositivo gráfico e acessa a memória de vídeo ou a memória AGP (*Accelerated Graphics Port*), enviando primitivas para serem processadas pela GPU (Wloka, 2004). Após as etapas de processamento, a CPU recupera o resultado, armazenado no *frame buffer*. Esse processo é ilustrado pela Figura 2.7.

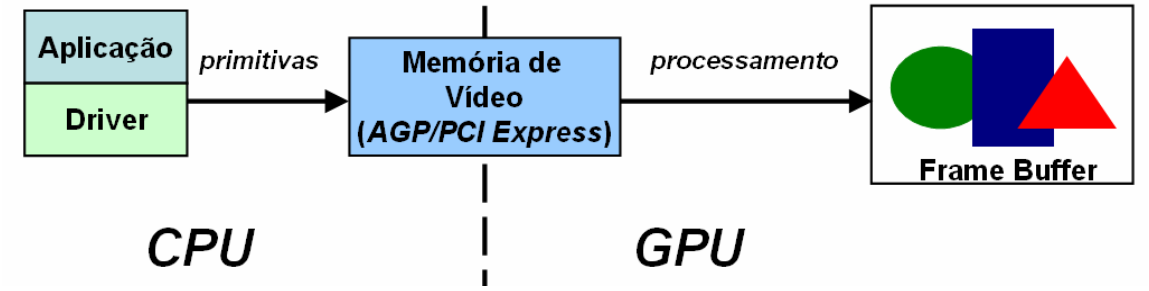


Figura 2.7: Visão geral do processo de renderização em baixo nível.

A identificação e remoção dos gargalos computacionais no processo de renderização são fundamentais para se obter um melhor desempenho. Para isso, é necessário entender melhor as etapas do processamento realizado em conjunto pela CPU e GPU. A Figura 2.8 ilustra as principais etapas envolvidas no processamento das primitivas pela GPU, onde ocorre o processamento dos vértices, que compõem as primitivas, e dos *pixels* no *frame buffer*, resultantes da projeção das primitivas (Wloka, 2004; Rege, 2004; Hart, 2004). Cada etapa conta com o resultado da etapa anterior para desempenhar o seu papel e representa um gargalo potencial para a renderização de uma cena tridimensional.

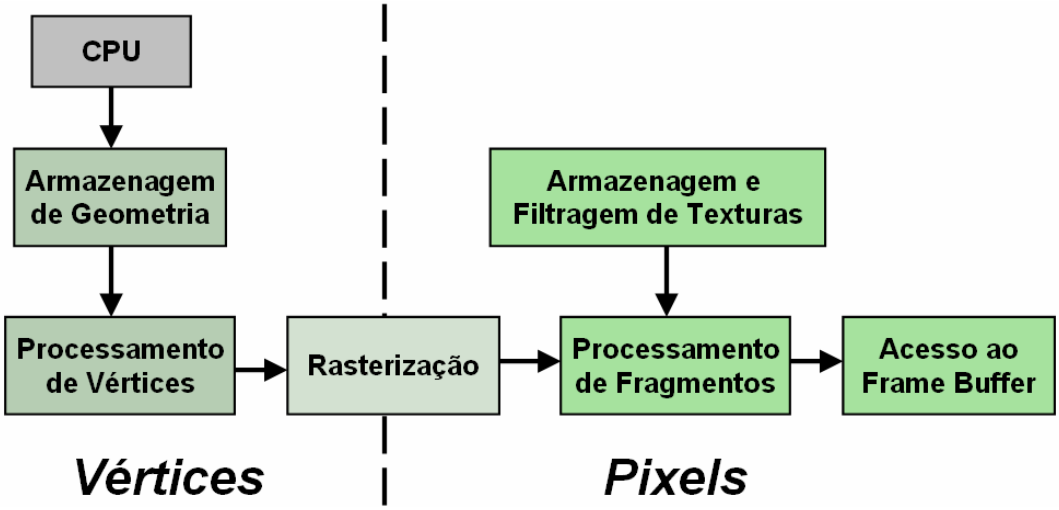


Figura 2.8: Principais etapas de uma pipeline gráfica.

As seguintes operações, realizadas durante as etapas ilustradas acima, podem comprometer o desempenho de uma aplicação de computação gráfica em tempo-real:

- Intenso processamento realizado pela CPU antes da renderização de cada quadro ou muitas trocas de estado a cada devido à má utilização da biblioteca gráfica, na etapa executada pela CPU;

- Transferência ineficiente da geometria para o subseqüente processamento na GPU, durante a etapa de Processamento de Vértices;
- Processamento ineficiente dos vértices na etapa de Transformação. Pode ser causado pelo excesso de vértices nos modelos, pela grande quantidade de cálculos por vértice e por deficiências de acesso aos vértices;
- Interpolação dos atributos nos vértices projetados durante a etapa de Rasterização. Essa operação dificilmente introduz gargalos, sendo influenciada principalmente pelo tamanho das primitivas projetadas e pelo número de atributos interpolados;
- Falta de memória para armazenar texturas, má utilização da cache de textura ou filtragem excessiva durante a etapa de Armazenagem e Filtragem de Texturas;
- Muitos cálculos realizados por fragmento devido ao grande número de fragmentos gerados ou pela execução de operações desnecessárias durante a etapa de Processamento de Fragmentos;
- Tráfego intenso de dados no *frame buffer* na etapa de Acesso ao Frame Buffer devido ao uso de múltiplos passos, renderização para gerar texturas e escritas desnecessárias no Z-buffer;

Durante a aplicação de efeitos visuais nos modelos poligonais renderizados, a camada de aplicação precisa definir os vários estados internos da biblioteca gráfica: desativar a iluminação, ativar a transparência, ativar o mapeamento de texturas, filtro de textura, etc. A ordem em que os objetos da cena são pintados num quadro determina uma série de mudanças nos estados internos da biblioteca gráfica, influenciando no desempenho da renderização desse quadro. Portanto, é interessante intervir na ordem de pintura para reduzir o número mudanças de estado e melhorar o desempenho da renderização. Uma estratégia eficaz e muito utilizada renderização em tempo-real consiste na implementação de uma pipeline gráfica intermediária, onde os objetos da cena são reordenados de acordo com os estados mais onerosos da biblioteca gráfica, como os estados que afetam a transparência e o número de texturas utilizadas num material (Sun Microsystems, 2002; OGRE3D, 2004; Maia et al., 2004; Fly3D, 2004; OpenSG, 2004), pois em cenas contendo um grande número de objetos, não é viável reordenar esses objetos de acordo com todos os atributos de seus materiais.

Durante a etapa de Armazenagem de Geometria, a camada de aplicação pode armazenar primitivas em blocos de memória de alto desempenho, disponíveis na nova geração de dispositivos gráficos (Segal & Akeley, 2004; DirectX, 2004), reduzindo drasticamente o tráfego ente CPU e GPU. Além disso, o custo do acesso aos vértices também é reduzido durante a etapa de Processamento de Vértices (Wloka, 2004; Rege, 2004; Hart, 2004). Outros gargalos na etapa de Processamento de Vértices podem ser evitados usando-se primitivas indexadas em modelos contendo um grande número de polígonos e principalmente pela decomposição desses modelos em um conjunto de primitivas do tipo *triangle strip* (tira de triângulos), mais eficientes por reduzir o processamento redundante dos vértices.

Para impedir gargalos na etapa de Armazenagem e Filtragem de Texturas, é importante que a memória seja desperdiçada pela banalização de texturas em alta definição. Além disso, é importante utilizar os filtros trilinear e anisotrópico apenas quando forem necessários, pois podem reduzir a taxa de preenchimento de *pixels* à metade (Hart, 2004). A utilização de níveis de detalhe nas texturas, conhecidos na literatura como *mipmaps*, tanto melhora a qualidade das imagens renderizadas quanto contribui para uma melhor utilização da *cache* de textura (Segal & Akeley, 2004; DirectX, 2004).

As otimizações mais dramáticas são obtidas pela utilização das técnicas para a redução do número de polígonos exibidos, pois diminuem a carga de processamento em todas as etapas da renderização sem penalizar a qualidade visual das imagens sintetizadas (Hoffman, 2000). Assim, a aplicação pode evitar o envio de polígonos que não estão presentes na imagem final, além de substituir objetos complexos por versões simplificadas, tarefas que demandam algoritmos e estruturas de dados especiais.

Técnicas de Processamento de Geometria podem ser realizadas numa etapa de pré-processamento da cena. Uma das técnicas comumente utilizadas é a conversão de uma malha de triângulos em um conjunto de tiras de triângulos, que, por terem uma representação mais concisa, tornam o processo de renderização mais eficiente. Outras técnicas muito utilizadas são as baseadas em níveis de detalhe (*Level of Detail* – LOD). Essas técnicas são capazes de substituir malhas complexas por versões simplificadas segundo um critério pré-especificado, tal como a distância ao observador (Funkhouster & Sequin, 1993). Embora essa substituição reduza o número de polígonos a serem

processados, em algumas ocasiões, a transição é perceptível na visualização da cena. Técnicas de LOD Contínuo solucionam esse problema através de uma transição suave entre versões de uma malha, proporcionada por uma representação contínua denominada malha progressiva (Hoppe, 1996). As técnicas descritas em (Ducheineau et al., 1997), (Garland & Heckbert, 1995) e (Lindstrom et al., 1996) implementam LOD de acordo com o ponto de visão em tempo de renderização, que utilizam a coerência entre os quadros para reduzir o processamento de atualização de suas estruturas de dados.

As estratégias de Renderização Baseada em Imagem baseiam-se na substituição de uma malha complexa por um polígono simples – geralmente um retângulo – no qual está mapeada uma imagem pré-renderizada da malha original. Esses polígonos, conhecidos na literatura como *billboards*, são reconstruídos a cada quadro de maneira a estarem sempre voltados para a câmera. Impostores (Schaufler, 1995) são utilizados com a mesma finalidade quando os objetos da cena movem-se lentamente em relação ao observador ou vice-versa. A malha complexa é renderizada em uma imagem que é aplicada como textura no impostor, permitindo ignorar totalmente o objeto original. Entretanto, o uso de impostores não está livre de problemas. Segundo Decoret et al. (1999), essa técnica apresenta problemas de: deformação causada pela representação do impostor; incompatibilidade entre a resolução do impostor e da textura; representação incompleta; *rubber sheet effect* e fissuras na imagem. Em Schaufler (1997), o autor propõe uma primitiva de renderização chamada *nailboard*, estendendo seu conceito de impostores dinamicamente gerados. Essa primitiva resolve o problema de visibilidade parcial quando da combinação dos impostores com a cena, usando uma textura no formato RGBΔ, onde os valores Δ são lidos do buffer de profundidade juntamente com os valores RGB da textura gerada.

Durante a renderização de uma cena tridimensional, os objetos situados fora do volume de visão da câmera (*frustum*) ou ainda sobrepostos por objetos opacos, por exemplo, não são vistos pelo usuário. Essa situação é muito comum em cenários fechados, podendo ocorrer em cenários abertos, como ilustrado pela Figura 2.9, onde os objetos de cor cinza não aparecem na imagem final.

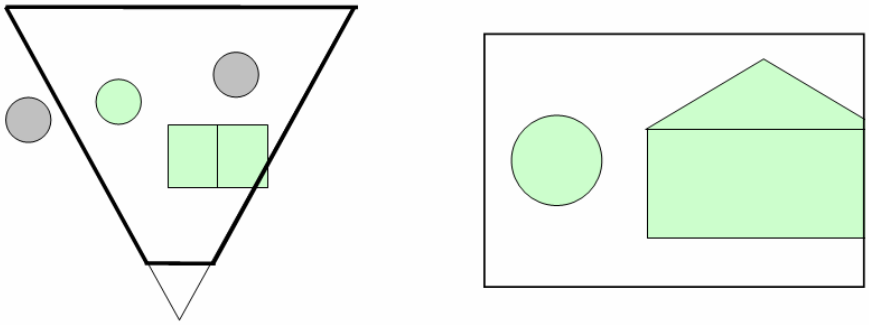


Figura 2.9: Um objeto é excluído pelo volume de visão enquanto o outro é obstruído.

Assim, a camada de aplicação pode tirar vantagem das técnicas de Remoção Seletiva de Objetos. Morley (2002), descreve uma estratégia simples e eficiente, chamada *View Frustum Culling*, onde os planos que formam o volume de visão são calculados a partir das matrizes de projeção e de visão. Durante a etapa de pintura, esses planos podem ser usados para rejeitar rapidamente objetos externos ao volume de visão.

A utilização de representações baseadas na subdivisão do espaço na construção da cena tridimensional permite explorar a coerência espacial e rejeitar rapidamente objetos situados fora do volume de visão. Assim, testes de visibilidade podem examinar apenas os objetos na vizinhança desse volume. O uso de decomposições hierárquicas como árvores de volumes envoltórios e estruturas de dados espaciais, como Octrees, Árvores BSP, K-d trees e R-trees, pode acelerar ainda mais esses testes de visibilidade (Figueiredo et al., 2002).

Essas estruturas de dados espaciais são geralmente utilizadas para representar a parte estática das cenas, enquanto hierarquias de volumes envoltórios são comumente usadas para acelerar a renderização de objetos dinâmicos organizados em um grafo de cena (Epic Games, 2004; OGRE3D, 2004; OpenSG, 2004). Essas hierarquias são compostas por volumes simples como esferas e *bounding boxes*, não necessariamente disjuntos, que englobam recursivamente o volume dos seus filhos. Assim, quando um determinado nó é considerado invisível, esse nó e todos os seus filhos não precisam ser renderizados.

Outras técnicas, conhecidas na literatura como Potentially Visible Sets (PVS), baseiam-se na pré-computação de visibilidade nas cenas. Dada uma divisão da parte estática de uma cena em setores, técnicas de visibilidade em potencial podem realizar uma pré-computação de visibilidade (Teller & Séquin, 1991) que resulta em uma estrutura de dados, geralmente uma matriz, capaz de indicar em tempo constante se um dado setor pode

ser visto a partir de um outro setor. Assim, considerando o observador em um dado setor, é possível descartar rapidamente setores e objetos que não são visíveis a partir daquele setor.

Cenas em ambientes fechados podem ser renderizadas rapidamente através do emprego de técnicas de portais, conhecidas na literatura como *Portal Rendering*, que decompõem a parte estática da cena em setores convexos interligados por polígonos denominados portais (Airey et al., 1990). Um algoritmo de renderização recursivo pode partir de um setor inicial, que é enviado imediatamente ao *frame buffer*, e avançar apenas para os setores conectados a esse setor inicial através dos portais visíveis pelo usuário. Além disso, o volume de visão pode ser incrementalmente reduzido à medida que o algoritmo percorre os portais. Por outro lado, técnicas como PVS e árvores BSP podem ser combinadas com *Portal Rendering* para aumentar drasticamente a velocidade de processamento.

Em cenas complexas, ricas em termos de detalhes, é comum que objetos opacos próximos ao observador obstruam a visão de outros objetos mais distantes. Uma casa próxima ao avatar do usuário, por exemplo, pode obstruir um enorme número de objetos situados atrás dela. Greene et al. (1993), Hudson et al. (1997) e Zhang et al. (1997) descrevem estratégias para a detecção de oclusões em tempo-real que exploram esse tipo de situações. Apesar de essas técnicas possuírem grande complexidade de implementação e alto custo computacional, elas são capazes de aumentar agressivamente a taxa de quadros por segundo (Hoffman, 2000). Por outro lado, os dispositivos gráficos modernos possuem suporte a testes de oclusão acelerados por hardware, capazes de determinar quantos *pixel*s são efetivamente afetados pelo processamento de um conjunto de primitivas (Segal & Akeley, 2004; DirectX, 2004). Por outro lado, esses testes podem ser usados para a seleção de níveis de detalhe dos objetos na cena e para a implementação de efeitos de iluminação. Para a obtenção de resultados mais expressivos através desses testes, é recomendável que os objetos com grande capacidade de obstrução, como paredes, casas e montanhas, por exemplo, sejam pintados primeiro (Segal & Akeley, 2004).

2.3.2 Detecção de Colisões

Como visto anteriormente, durante a execução de uma aplicação de RV, os objetos tridimensionais são representados por modelos poligonais, usados principalmente na

renderização das cenas. Geralmente esses modelos poligonais são malhas triangulares, caso contrário podem ser decompostos facilmente em malhas triangulares. Assim, os testes de interseção com triângulos são fundamentais para a elaboração de uma solução para o problema de detecção de colisões.

Os testes de interseção devem considerar um par de malhas triangulares $\mathbf{T_1}$ e $\mathbf{T_2}$ e suas matrizes $\mathbf{M_1}$ e $\mathbf{M_2}$, que aplicam transformações afins nesses objetos, respectivamente. Assim, é importante que esses testes suportem a utilização de escalas, rotações e translações nos objetos tridimensionais da cena através dessas matrizes. Por outro lado, é desejável que as técnicas utilizadas nesses testes possam suportar eficientemente modelos deformáveis, além de contar com as heurísticas de coerência espacial e temporal.

Em Möller (1997), o autor propõe um teste de interseção exato, eficiente e robusto entre dois triângulos arbitrários. Esse teste pode ser usado como base para a construção de testes mais complexos envolvendo malhas triangulares. Em Möller & Trumbore (1997), os autores propõem um teste elegante, eficiente e robusto para o cálculo da interseção entre um raio e um triângulo arbitrário. Esse tipo de teste é indispensável para a elaboração de algoritmos para a seleção de objetos tridimensionais (*picking*), úteis para a implementação de interações e para a movimentação de partículas.

Em Gottschalk (1990), o autor apresenta o Teorema do Eixo de Separação (*Separating Axis Theorem*), conhecido na literatura como SAT, que determina condições necessárias para que dois poliedros convexos não estejam se interceptando. Nesse teorema, um Eixo de Separação é definido como sendo um eixo onde a projeção dos poliedros não se intercepta. Assim, o teorema diz o seguinte: dois poliedros convexos são disjuntos se e somente se existe um Eixo de Separação ortogonal a uma face de cada poliedro ou ortogonal a uma aresta de cada poliedro. Como resultado direto do SAT, é suficiente verificar os eixos de separação em potencial nos testes de interseção entre poliedros convexos, um algoritmo bastante eficiente para um pequeno número de faces nos poliedros.

Em Van Der Bergen (1999B), o autor apresenta uma implementação robusta, eficiente e versátil do algoritmo iterativo de Gilbert-Johnson-Keerthi (Gilbert et al., 1988), conhecido na literatura como GJK, para a computação da distância entre um par de objetos convexos. Essa implementação utiliza o SAT combinado à coerência temporal para aperfeiçoar o algoritmo original e rejeitar rapidamente objetos que não se interceptam.

Além disso, os testes são, em geral, 5 vezes mais rápidos do que a computação dos pontos, arestas ou faces mais próximas proposto por Lin & Canny (1991). Por outro lado, essa implementação é capaz de detectar a colisão de objetos modelados através de várias primitivas geométricas manipuladas através de transformações afins. Apesar disso, o autor não apresenta formas de adaptar o algoritmo para lidar com objetos deformáveis.

Em Kim et al. (2002), os autores apresentam um algoritmo original e eficiente para computar a distância entre dois modelos poliedrais. Dados dois poliedros que se interceptam, o algoritmo computa a translação mínima para separá-los usando uma combinação de técnicas realizadas tanto no espaço de objeto quanto no espaço de imagem.

Em Magic Software (2004), o autor disponibiliza testes de interseção entre pares de primitivas geométricas variadas, como raios, segmentos de reta, esferas, bounding boxes, cones, cilindros, tetraedros e triângulos. O autor disponibiliza também artigos descrevendo, em detalhe, as técnicas utilizadas e, além disso, esses testes estão implementados em código C++ que pode ser utilizado livremente em projetos comerciais. Entretanto, o autor não disponibiliza testes de interseção entre malhas triangulares e primitivas geométricas, nem testes entre duas malhas triangulares.

Em Gottschalk et al. (1996), os autores baseiam-se no SAT para projetar um eficiente teste de interseção entre malhas triangulares de objetos sujeitos ao movimento rígido. Nesse trabalho, Gottschalk et al. utilizam OBBs para aproximar um modelo através de uma hierarquia de volumes envoltórios construída a partir do fecho convexo dos vértices que compõem o modelo e uma matriz de covariância, onde os nós internos da árvore binária resultante são OBBs e cada nó folha corresponde a um triângulo do modelo. Após esse processo, as hierarquias dos modelos são utilizadas nos testes de interseção, baseados no SAT, para rejeitar rapidamente pares de triângulos que não se interceptam. Assim, o algoritmo é acelerado pelos testes de interseção entre pares de OBBs, que checa os 15 potenciais eixos de separação entre as OBBs, 3 para cada face e 9 ortogonais aos pares de arestas. Apesar da sua eficiência comprovada, essa técnica não suporta eficientemente a interseção entre modelos deformáveis, pois, quando um modelo é deformado, a hierarquia de volumes correspondente precisa ser totalmente reconstruída e essa reconstrução constante não é viável em tempo de execução.

Em Van der Bergen (1997), o autor propõe uma adaptação da técnica introduzida por Gottschalk et al. de maneira suportar a modelos deformáveis. Nesse trabalho, o autor propõe a substituição da hierarquia de OBBs por uma hierarquia de AABBs, mais fácil de ajustar ao modelo deformado, e cuja reconstrução toma apenas 5% do tempo necessário para reconstruir uma hierarquia de OBBs. Assim, o autor utiliza AABBs num algoritmo aproximativo em que apenas 6 dos 15 eixos de separação em potencial são testados. Isso implica na perda de apenas 6% na taxa de rejeição dos testes, implicando num maior número de testes entre pares de triângulos. Segundo o autor, a técnica adaptada requer apenas 50% mais tempo do que a técnica original. Além disso, o algoritmo para atualizar a hierarquia é bastante simples: reajustam-se primeiro os nós folhas, e cada nó interno em função dos seus filhos.

Em Govindaraju et al. (2003) e Govindaraju et al. (2004), os autores propõem uma técnica original para detectar colisões entre múltiplos objetos deformáveis utilizando o hardware gráfico. Essa técnica não possui uma etapa de pré-processamento e pode ser utilizada em modelos quebráveis. Numa primeira etapa, testes de oclusão acelerados por hardware são utilizados para determinar que pares de objetos estão potencialmente colidindo. Numa segunda etapa, são realizados testes similares utilizando *bounding boxes*, grupos de triângulos e triângulos individuais, computados a partir dos objetos remanescentes da primeira etapa. Numa última etapa, são realizados testes exatos, em espaço de objeto, para obter os triângulos que se interceptam. Essa técnica apresenta como desvantagem a necessidade de hardware gráfico com suporte aos testes de oclusão e com uma alta taxa de preenchimento de *pixels*.

Por outro lado, é possível reaproveitar as estruturas de dados utilizadas na aceleração da renderização para acelerar também os testes de interseção (Maia et al. 2003). Uma *Octree* representando a parte estática do cenário e uma estrutura de PVS, por exemplo, podem ser utilizadas para rejeitar rapidamente objetos que não se interceptam, acelerando os testes de interseção. Além disso, essa estratégia de reutilização das estruturas de dados é econômica em termos de uso memória e de processamento.

2.3.3 Som Espacial

Atualmente, há uma variedade de bibliotecas, praticamente todas de baixo nível, disponíveis para a simulação de ambientes com som espacial (BASS, 2004; DirectX, 2004; Firelight Technologies, 2004; Loki Software, 2000; RAD Game Tools, 2004). Essas bibliotecas simulam diversos efeitos do ambiente e podem ser utilizadas livremente em projetos de pesquisa, possibilitando a construção de ambientes virtuais mais ricos e envolventes que oferecem áudio de qualidade para os visitantes do mundo virtual.

Apesar disso, a maioria dessas bibliotecas não é portável entre diversas configurações de plataforma e hardware, confinando as aplicações desenvolvidas a alguma plataforma específica ou exigindo hardware de áudio (DirectX, 2004; BASS, 2004). Assim, é preferível utilizar uma biblioteca gratuita e portável como OpenAL (Loki Software, 2004) ou FMOD (Firelight Technologies, 2004).

Nessas bibliotecas, a camada de aplicação especifica amostras de som através de *buffers* para que estas sejam reproduzidas no contexto do ambiente através de fontes sonoras que são percebidas pelo ouvinte da cena sonora. Em algumas dessas bibliotecas, os sons podem ser especificados através de arquivos de áudio, como MIDI, MP3, Ogg Vorbis ou WAV (Wireman, 2003). Esses formatos oferecem à camada de aplicação várias opções de amostragem de áudio contrabalanceando qualidade, alocação de memória e carga de processamento. Noutras bibliotecas, como o OpenAL, que suporta amostras apenas o formato WAV, a camada de aplicação necessita descompactar amostras de som noutros formatos antes de criar os *buffers* correspondentes.

Para se obter respostas visual e sonora coerentes, é preciso atualizar o ouvinte e as fontes de som à medida que a câmera do usuário e os objetos na cena tridimensional são modificados. Antes da renderização de cada quadro, a cena de áudio deve ser sincronizada com a cena tridimensional pela camada de aplicação, pois a biblioteca de áudio não tem conhecimento das posições e orientações dos objetos que compõem a cena renderizada.

2.3.4 Linguagens de Script

Uma solução de baixo nível para o suporte a linguagens de script consiste em desenvolver uma linguagem própria. A execução de um script envolve as etapas de análise léxica, análise sintática, tradução e interpretação do código gerado. Além disso, para utilizar a

linguagem de script num sistema de RV, a camada de aplicação precisa de um mecanismo de realizar um *binding* e de uma interface para controlar o interpretador. Para tratar das etapas de análise léxica e sintática, o desenvolvedor conta com ferramentas de domínio público como flex (Flex, 2004) e bison (Bison, 2004), além das mais recentes flex++ e bison++(Programmer's Toolbox, 2004). O flex e o flex++ são capazes de gerar analisadores léxicos, enquanto o bison e o bison++ são capazes de gerar analisadores sintáticos.

Em Dawson (2002), o autor defende o reuso de uma linguagem pré-existente, pois linguagens como Lua (Ierusalimschy et al., 2003), Python (Python Software Foundation, 2004) ou Ruby (Thomas et al., 2004) geralmente são mais flexíveis, eficientes e expressivas do que alguém conseguiria criar e manter. Além disso, essas linguagens são maduras e estão prontas para serem usadas num projeto, evitando os riscos de desenvolver uma linguagem própria.

2.4 Soluções de Alto Nível

As soluções de baixo nível consistem em técnicas e algoritmos para tratar problemas específicos, portanto, sua utilização geralmente requer algum esforço de desenvolvimento para a sua utilização em um sistema de RV. Por outro lado, as soluções de alto nível implementam internamente tais técnicas de baixo nível e oferecem ao programador uma solução reutilizável através de uma interface de alto nível. Durante as próximas subseções, serão apresentadas as principais soluções de alto nível para a implementação dos aspectos fundamentais de um sistema de RV.

2.4.1 Bibliotecas de Alto Nível

Cada aspecto do ambiente virtual pode ser implementado através de uma biblioteca de alto nível, adequada a implementação de um conjunto específico desses aspectos. Assim, é provável que o desenvolvedor tenha de utilizar várias dessas bibliotecas durante o desenvolvimento de um sistema de RV.

As bibliotecas de renderização em tempo-real integram diversas técnicas de aceleração através das principais APIs 3D. A biblioteca de código aberto OpenSG (OpenSG, 2004), implementada em C++ utilizando os princípios da Orientação a Objetos,

encapsula um sistema baseado em OpenGL para renderização em alto nível através de grafos de cena. As principais características dessa biblioteca são:

- **Código Aberto**. O código fonte dessa biblioteca é distribuído de acordo com a licença LGPL, permitindo alterações e livre utilização em projetos comerciais;
- **Portabilidade**. É possível utilizar OpenSG nas plataformas IRIX, Linux e Windows; e
- **Desempenho**. Essa biblioteca implementa diversas técnicas de aceleração da renderização, como *Frustum Culling* e reordenação dos objetos visíveis antes da pintura. Além disso, é possível que a cena seja exibida simultaneamente em várias *viewports*.

Essa biblioteca é utilizada por pesquisadores do mundo inteiro e possui uma extensa documentação contendo vários exemplos de código. Entretanto, não é possível utilizar outras bibliotecas para a renderização de baixo nível. Bibliotecas similares ao OpenSG, como OpenSceneGraph (OpenSceneGraph, 2004), OpenGL Performer™ (OpenGL Performer, 2004) e Open Inventor™ (Open Inventor, 2004), apresentam características semelhantes.

Similarmente, a biblioteca portável Java3D permite a renderização eficiente de cenas representadas através de um grafo de cena implementado em Java, podendo utilizar tanto Direct3D quanto OpenGL para a renderização de baixo nível. Além disso, essa biblioteca permite utilizar dispositivos de entrada e saída típicos de RV, como luvas, óculos e *trackers*. Java3D possui suporte a outros dois aspectos de um sistema de RV: som espacial e detecção de colisões. Apesar disso, a biblioteca apenas detecta a interseção entre um par de nós no grafo de cena e não retorna informações sobre o contato desses nós. Por outro lado, os sistemas desenvolvidos com Java3D sofrem de uma queda de desempenho devido à interpretação dos *bytecodes* Java.

A biblioteca aberta SmallVR (Pinho, 2002) baseada em OpenGL/GLUT e implementada em C++, oferece suporte a diversas características de uma aplicação de RV: portabilidade, renderização em tempo-real, carga de modelos geométricos, controladores de dispositivos e detecção de colisões. O controle de como os objetos no grafo de cena são pintados é a principal diferença entre a SmallVR e as demais bibliotecas. Nessa biblioteca, que implementa a técnica de *View Frustum Culling* para acelerar a pintura das cenas, os

objetos tanto podem ser definidos através de arquivos quanto através de código definido pela camada de aplicação. A leitura dos dispositivos de entrada é implementada através de um mecanismo onde os sensores físicos de um dispositivo afetam um objeto-sensor presente na cena. Essa biblioteca possui duas abordagens para a detecção de colisões: uma utiliza uma própria para os testes de interseção e a outra utiliza a biblioteca SOLID (Van der Bergen, 1999A). Entretanto, SmallVR não oferece suporte à utilização de outras bibliotecas gráficas (como Direct3D), a som espacial nem à utilização de linguagens de script.

Várias bibliotecas de alto nível estão disponíveis para a detecção de colisões entre modelos geométricos (Gamma Team, 2004; Terdiman, 2003, Van Der Berger, 1999A; ColDet, 2004; Klosowski et al., 1998). A maioria dessas bibliotecas é de domínio público e limita-se à detecção interseções entre pares de modelos. Além disso, várias dessas bibliotecas não são capazes de lidar com modelos deformáveis nem permitem aplicar escalas não-uniformes nos modelos geométricos. A biblioteca OPCODE, por exemplo, suporta modelos triangulares deformáveis mas não é capaz de lidar com escalas.

Atualmente, existem diversas linguagens de script maduras e prontas para utilização num sistema de RV. As linguagens de domínio público Lua, Python e Ruby são muito utilizadas em conceituados projetos comerciais e de pesquisa. As principais vantagens dessas linguagens são: portabilidade, desempenho, sintaxe simples, controle do interpretador e boa documentação. Além disso, essas linguagens facilitam o processo de *binding*. Lua, por exemplo, possui ferramentas auxiliares para geração automática de extensões.

2.4.2 *Frameworks* de Realidade Virtual

Os *Frameworks* de Realidade Virtual são bibliotecas que encapsulam soluções genéricas e escaláveis para o desenvolvimento de aplicações de RV (Bierbaun et al., 2001). A principal diferença entre uma biblioteca de alto nível e um Framework de RV está na maneira como a camada de aplicação interage com o *framework*. Quando um sistema de RV utiliza bibliotecas de alto nível, a camada de aplicação é responsável por coordenar o funcionamento dessas bibliotecas no contexto do sistema. Por outro lado, quando um sistema utiliza um *Framework* de RV, a camada de aplicação apenas descreve o ambiente

em que os usuários interagem, ou seja, o funcionamento da aplicação é coordenado pelo *framework*.

Dessa maneira, o *Framework* de RV é responsável por abstrair a complexidade de utilização de componentes locais e distribuídos no contexto de um sistema de RV. Como a aplicação se abstrai de praticamente todos os aspectos do o ambiente, os componentes de hardware e software, gerenciados pelo *framework*, podem ser trocados e configurados de maneira transparente. Isso permite, por exemplo, trocar facilmente os dispositivos de entrada e saída utilizados para a execução do ambiente sem que a aplicação seja modificada. Os *frameworks* de RV oferecem grande portabilidade aos sistemas desenvolvidos, permitindo a execução de aplicações em praticamente qualquer plataforma (Tramberend, 1999; Arsenault et al., 2001; Hartling et at., 2002; Bastos et al., 2004).

Segundo Pinho (2002), as maiores desvantagens na utilização desse tipo de soluções são a complexidade de aprendizado, e, principalmente, na falta de controle por parte da camada de aplicação. Isso dificulta ou impossibilita a utilização de soluções específicas, mais eficientes para uma determinada aplicação.

2.4.3 Motores Gráficos

Os jogos tridimensionais de computador têm-se tornado cada vez mais realistas e complexos, demandando a utilização das mais modernas técnicas de computação gráfica, interação e simulação na forma de bibliotecas (Cal3D, 2004; OpenSteer, 2004; RenderWare, 2004; Tokamak, 2004; Terdiman, 2003) e kits de desenvolvimento conhecidos como Motores Gráficos (Id Software, 2004; Epic Games, 2004; Fly3D, 2004; OGRE3D, 2004; Valve Corporation, 2004; Berger, 2002).

Embora o termo "Motor Gráfico" sugira que esse tipo de componente encapsula apenas subsistemas relacionados à representação de uma cena composta puramente por modelos tridimensionais, um Motor Gráfico geralmente encapsula uma abrangente gama de funcionalidades como comunicação em rede e reprodução de sons. Por muitas vezes, esse tipo de componente atinge um nível de complexidade comparável ao de um sistema operacional, gerenciando memória e escalonando tarefas.

Essa massa de tecnologia disponível para o desenvolvimento de jogos realistas tem despertado grande interesse por parte dos pesquisadores e desenvolvedores de RV. Esse

tipo de tecnologia representa um meio termo entre as bibliotecas de alto nível e os *frameworks* de RV, integrando soluções de baixo nível e delegando a gerência do ambiente virtual, em alto nível, à camada de aplicação (Maia et al., 2003).

Várias aplicações de RV de *Desktop* têm adotado, com sucesso, as tecnologias usadas originalmente no desenvolvimento de jogos (Leite-Júnior et al., 2002; Reis et al. 2003). Em Leite-Júnior et al. (2002) , os autores adotaram o motor gráfico 3DSTATE (Berger, 2002) para o desenvolvimento de seu cliente de interação em realidade virtual. Já em Reis et al. (2004), os autores julgaram conveniente a utilização da API Cal3D (Cal3D, 2004) para animar personagens virtuais durante o desenvolvimento de seu estudo de caso. Essa tendência natural de adotar tais tecnologias é conseqüência do avanço e do barateamento de hardware gráfico, que foi fortemente influenciado pela indústria dos jogos de computador (Brooks, 1999).

A utilização de motores gráficos e soluções similares no desenvolvimento de sistemas de RV têm incrementado drasticamente a produtividade, isolando a complexidade da programação de subsistemas em baixo nível, além de contribuir para uma maior portabilidade das aplicações desenvolvidas, vista a otimização dos recursos que esse tipo de componente geralmente realiza (Elias, 2002; Leite-Júnior et al., 2002). Apesar disso, a maioria dos motores disponíveis é proprietária e de código fechado, além de ser baseada numa plataforma e num conjunto fixo de tecnologias, dificultando o processo de manutenção das aplicações desenvolvidas. Assim, essas aplicações geralmente estão confinadas a uma plataforma e tecnologias específicas, reduzindo suas capacidades de configuração e portabilidade (Maia et al., 2003). Por outro lado, os motores que melhor contornam essas limitações geralmente são comerciais, associando um custo inviável ao orçamento disponível para o desenvolvimento de projetos de RV que visam confeccionar um produto vendável.

2.5 Considerações Finais

O desenvolvimento de aplicações de Realidade Virtual demanda a utilização de diversas tecnologias, principalmente renderização em tempo-real, detecção de colisões, som espacial e linguagens de script.

O desenvolvedor conta com leque de soluções disponíveis para o desenvolvimento de aplicações de RV. As bibliotecas de baixo e alto nível oferecem técnicas que podem ser integradas numa solução customizada, mais adequada para uma aplicação específica. Os *frameworks* de RV, por sua vez, apresentam uma solução genérica para aplicações, onde a camada de aplicação exerce pouco controle sobre como o ambiente virtual é gerenciado.

Por outro lado, os motores gráficos oferecem uma interessante combinação entre desempenho e controle, pois esse tipo de componente integra soluções de qualidade controladas pela camada de aplicação. No próximo capítulo, é apresentada a estrutura de um motor gráfico genérico.

Capítulo 3

Estrutura Genérica de Motores Gráficos

3.1 Introdução

Segundo Brooks (1999), os recentes avanços nas áreas de computação gráfica e o barateamento de hardware gráfico têm contribuído para a construção de ambientes de Realidade Virtual cada vez mais realistas. Por outro lado, esse incremento de sofisticação aumenta a complexidade dos sistemas de RV, cuja construção demanda componentes de software.

Sob essa perspectiva, a utilização de um Motor Gráfico é uma escolha interessante, principalmente para o desenvolvimento em ambientes de *Desktop*, pois esse tipo de componente oferece diversas vantagens, como:

- **Desempenho e qualidade**. Geralmente integra as mais modernas técnicas e soluções de baixo nível;
- **Confiabilidade**, os jogos produzidos são testados sob as mais diversas condições de hardware e software; e
- **Controle**, a camada de aplicação é responsável tanto pela descrição do ambiente virtual quanto pela gerência, sob vários aspectos, desse ambiente.

Dessa maneira, a aplicação pode ser construída com base num motor gráfico, utilizando os subsistemas que esse motor provê. As camadas dessa aplicação são ilustradas na Figura 3.1.

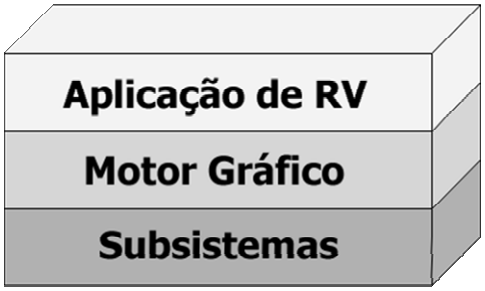


Figura 3.1: Camadas de uma aplicação utilizando um mtor gráfico.

Apesar das várias vantagens na utilização de um motor, esse tipo de componentes de software geralmente possui uma série problemas que impedem a sua plena utilização em sistemas de RV, como: baixa portabilidade, a falta de suporte a dispositivos não-convencionais, código fonte fechado, custo de utilização e capacidades de expansão e de configuração reduzidas (Maia et al., 2003).

Para uma que se tenha uma visão simplificada e mais compreensível de um componente de software, o mesmo pode ser dividido em módulos menores que compõem as partes de um todo. Essa divisão permite que cada módulo desse componente seja detalhado individualmente, facilitando sua explanação e, conseqüentemente, a compreensão tanto do módulo em questão quanto do componente como um todo. Essa abordagem é utilizada para descrever sistemas complexos de maneira detalhada em artigos, livros e manuais, como, por exemplo, na especificação do sistema gráfico OpenGL (Segal & Akeley, 2003).

Dessa maneira, essa abordagem será utilizada nesse capítulo, onde é proposta uma divisão de um motor gráfico genérico visando facilitar a compreensão e a comparação entre diferentes motores gráficos. Portanto, um motor gráfico será dividido em três módulos principais:

- **Núcleo**, ou módulo principal, que é responsável pela gerência do motor em alto nível e contém módulos fundamentais utilizados tanto pelos subsistemas quanto pela camada de aplicação;
- **Subsistemas**, que abrange cada subsistema provendo as principais funcionalidades do motor gráfico; e
- **Carregamento de Elementos do Mundo Virtual**, módulo responsável pela mediação necessária ao carregamento das mídias utilizadas pelo motor gráfico para especificar o conteúdo do mundo virtual, como, por exemplo, arquivos de som e modelos tridimensionais pré-moldados.

O restante desse capítulo encontra-se organizado da seguinte maneira. Na Seção 3.2, serão apresentados os principais aspectos do Núcleo. A Seção 3.3 destina-se ao detalhamento dos subsistemas. Na Seção 3.4, será abordado o módulo de Carregamento de Elementos do Mundo Virtual. Por fim, tecem-se alguns comentários finais na Seção 3-5.

3.2 Núcleo

O núcleo é a porção de um motor gráfico responsável por intermediar interações entre a camada de aplicação e os subsistemas disponíveis no motor gráfico, bem como as interações entre esses subsistemas. O núcleo é responsável em registrar, identificar, iniciar e coordenar o funcionamento de todos os módulos e subsistemas que compõem o motor gráfico. Dessa maneira, o núcleo também é encarregado de definir uma interface para o acesso e a comunicação entre os subsistemas.

Além dos subsistemas, o núcleo provê um conjunto de sub-módulos básicos utilizados por esses subsistemas e que, geralmente, também devem ser utilizados pela camada de aplicação. Assim, o núcleo encapsula funcionalidades essenciais para o funcionamento do motor, como o acesso ao sistema de arquivos e o cálculo de operações vetoriais. Num motor cujos subsistemas comunicam-se através de mensagens, por exemplo, o módulo responsável pela gerência dessas mensagens é considerado parte do núcleo.

Assim, o núcleo caracteriza-se por oferecer uma série de serviços comumente utilizados pelos subsistemas, além de estabelecer as políticas aplicadas ao carregamento de mídias; ao gerenciamento de memória; ao tratamento das particularidades da plataforma corrente; ao escalonamento das tarefas executadas pelo motor gráfico; ao sistema de eventos, que trata a interação do usuário com os dispositivos de entrada disponíveis; à manutenção do registro de atividades e erros do motor gráfico (*log*), possibilitando a extração de informações e estatísticas acerca da aplicação em execução. Além disso, o núcleo também descreve a interface para a especificação e manipulação da cena. É interessante que cada um desses serviços básicos possa ser expandido e configurado, o que raramente ocorre na prática (Maia et al., 2003). As principais funcionalidades do núcleo são ilustradas na Figura 3.2.

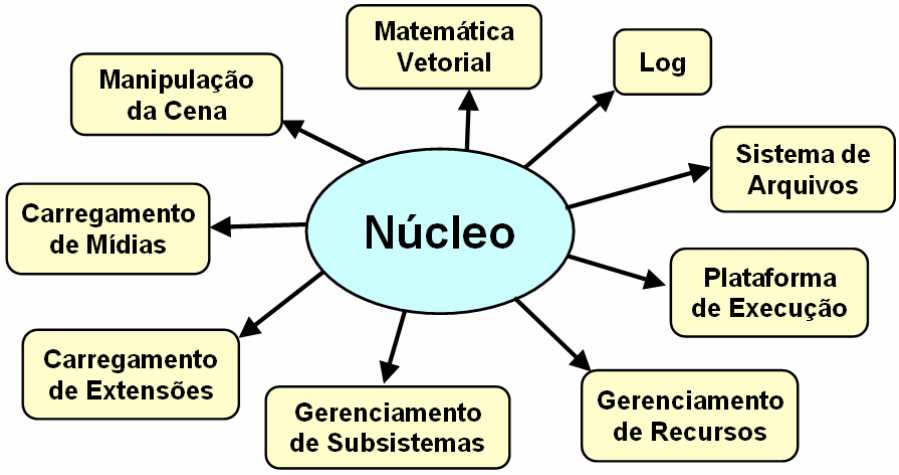


Figura 3.2: As principais funcionalidades do núcleo num motor gráfico.

Alguns motores podem ser expandidos em tempo de execução através de *plugins*, que são carregados como bibliotecas de vínculo dinâmico e registram módulos adicionais no motor gráfico (OGRE3D, 2004; Fly3D, 2004; Epic Games, 2004; Id Software, 2004). A utilização de *plugins* permite isolar o código de subsistemas e módulos semelhantes em diferentes *plugin*, de maneira a permitir que a camada de aplicação selecione o mais conveniente. Assim, é possível que a aplicação escolha utilizar o dispositivo X ou Y, que são implementados no *plugins* A e B, respectivamente.

Além disso, é possível corrigir falhas num módulo pela simples substituição do *plugin* correspondente, sem que seja necessário reconstruir o código executável do motor gráfico ou de aplicações. Quando o motor suporta o carregamento de *plugins*, o núcleo define as políticas adotadas durante a localização, o carregamento, a identificação e a inserção dos plugins na estrutura de execução adotada pelo motor gráfico.

Dentre todos os serviços básicos que o núcleo pode prover, é de fundamental importância que se forneça um sistema de eventos que facilite a conexão das entradas do usuário com as ações possíveis no ambiente virtual. Além disso, o registro de atividades e erros num *log* é imprescindível durante as etapas de desenvolvimento e manutenção de aplicações complexas e de extensões do motor gráfico.

3.3 Subsistemas

Um subsistema representa um módulo específico do motor gráfico que é utilizado pela camada de aplicação para a implementação de um conjunto de funcionalidades. Assim, um

determinado subsistema é caracterizado pelo domínio de problemas para o qual é destinado e pelas tarefas que desempenha como módulo do motor gráfico, além das tecnologias que utiliza para desempenhar essas tarefas. O subsistema de renderização em tempo-real, por exemplo, caracteriza-se por sintetizar imagens a uma taxa de exibição interativa com base numa cena tridimensional e, dentre as tarefas que desempenha, estão: a determinação da visibilidade dos objetos na cena, a geração texturas e a especificação sistemas de coordenadas através de matrizes. Para tanto, é necessário que esse subsistema se atenha a tecnologias específicas, como técnicas para a eliminação de faces ocultas e renderização em baixo nível através de uma API 3D.

Dessa maneira, cada subsistema é responsável por um aspecto específico da aplicação, bem como pela gerência desse aspecto em tempo de execução. Como componentes de software, os subsistemas são capazes de encapsular a complexidade de utilização de técnicas e subsistemas de baixo nível externos ao motor gráfico. Isso permite que o desenvolvedor de sistemas de RV utilize uma interface apropriada em alto nível, sendo geralmente composta por entidades e métodos.

Para cada subsistema de um motor gráfico, é muito provável que exista uma variedade de técnicas, algoritmos e bibliotecas adequados à sua construção. No caso do subsistema de renderização em tempo-real, por exemplo, existem várias APIs disponíveis para acessar o hardware gráfico, como OpenGL (Segal & Akeley, 2004) e Direct3D (DirectX, 2004). O subsistema de renderização pode ser fortemente acoplado a alguma API 3D, utilizando funções dessa API para a renderização de baixo nível. Por outro lado, esse subsistema pode ser projetado com base num conjunto de funcionalidades disponíveis na maioria das APIs gráficas disponíveis atualmente e utilizar uma camada de abstração sobre a API 3D durante a renderização de baixo nível. Esse princípio, ilustrado pela Figura 3.3, é adotado com sucesso por diversos sistemas de renderização (CrystalSpace, 2004; Epic Games, 2004; OGRE3D, 2004; RenderWare, 2004) visando oferecer maior portabilidade e capacidade de configuração.

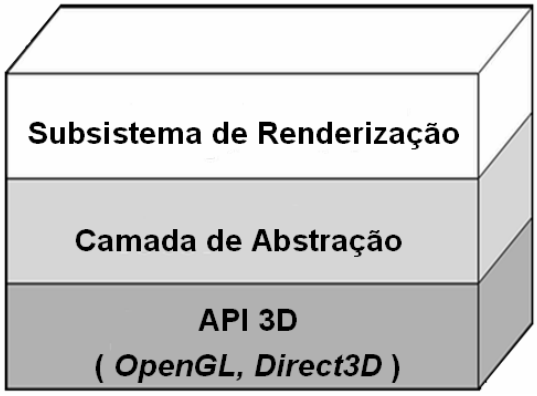


Figura 3.3: Camada de abstração de API 3D num subsistema de renderização.

Quando esse princípio de projeto é generalizado a todos os subsistemas do motor gráfico, esse motor pode oferecer inúmeras combinações de implementações distintas para cada subsistema. Dentre essas combinações, pode-se escolher a mais adequada em termos dos requisitos de desempenho e de qualidade inerentes à camada de aplicação. Numa aplicação desenvolvida com um motor gráfico que possui três implementações distintas para cada um dos seus três subsistemas, por exemplo, o usuário final da aplicação de RV desenvolvida com esse motor poderia escolher dentre um total de vinte e sete configurações possíveis.

Por outro lado, a aplicação de RV pode necessitar de algum outro subsistema importante que não esteja originalmente presente no motor gráfico. Assim, é de suma importância que o motor gráfico possua mecanismos de extensão que funcionem ao menos em tempo de compilação. Isso significa que o motor ou a aplicação precisa ser recompilada quando da inclusão de uma extensão ou de um subsistema no motor gráfico, o que pode ser bastante inconveniente durante o processo de desenvolvimento em projetos com vários milhares de linhas de código.

Dessa maneira, é preferível a utilização de mecanismos dinâmicos de extensão, como os *plugins*, possibilitando a identificação de extensões e novos subsistemas em tempo de execução. Essa abordagem pode ser generalizada inclusive para os subsistemas básicos, que podem atuar como extensões naturais e intercambiáveis do motor gráfico. Seguindo essa abordagem, o motor gráfico é composto apenas pelo núcleo quanto for iniciado. Os subsistemas de renderização, som espacial e scripts são registrados no motor gráfico após o

carregamento dos respectivos *plugins*. Além disso, outros subsistemas podem ser carregados sob demanda. Essas situações são ilustradas na Figura aki.

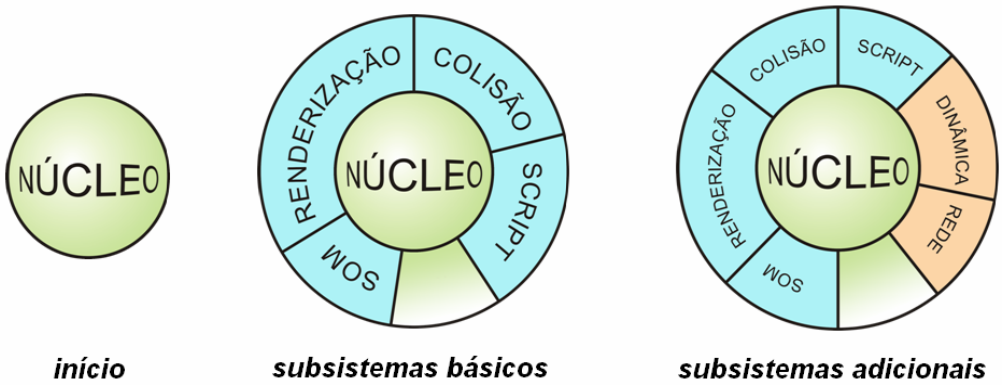


Figura 3.4: Situações num motor cujos subsistemas são registrados dinamicamente.

Dentre os subsistemas disponibilizados pelos motores gráficos existentes no mercado, alguns podem ser considerados essenciais:

- O subsistema de renderização em tempo-real é responsável pela geração da resposta visual interativa, fundamental para a construção de um sistema de RV;
- O subsistema de Som Espacial é responsável pela geração de estímulos auditivos, sendo capaz de reproduzir os efeitos sonoros do ambiente e introduzir emoção nos ambientes de RV;
- O subsistema de Detecção de Colisões é encarregado de determinar interseções e a situação de contato entre modelos tridimensionais, uma tarefa essencial para a utilização de inúmeras técnicas de computação gráfica e de simulação utilizadas nos sistemas de RV; e
- O subsistema de Script é responsável por interpretar trechos de código, escritos em alguma linguagem de script, capazes de alterar o estado da aplicação. Esse subsistema confere grandes capacidades de prototipagem ao desenvolvedor, imprescindíveis na construção de sistemas de grande porte, e, em particular, dos sistemas de RV.

Para o desenvolvimento de uma aplicação de RV usando um motor gráfico, é importante que esses quatro subsistemas estejam disponíveis, pois atuam como base para a implementação dos aspectos fundamentais da aplicação (Maia et al., 2003).

O subsistema de Script, em particular, é imprescindível para o desenvolvimento de sistemas de RV complexos, pois possuem necessidades especiais de configuração, prototipagem, manutenção e expansão. Os sistemas que dispõem de avatares controlados por computador, por exemplo, necessitam de freqüentes manutenções e maior capacidade de prototipagem durante seu desenvolvimento. Embora a utilização de scripts possa apresentar queda de desempenho, dependendo do ambiente de execução da linguagem e da interpretação dos scripts, a enorme flexibilidade conferida ao sistema justifica seu emprego (Dawson, 2002).

3.4 Carregamento de Elementos do Mundo Virtual

Numa aplicação típica de Realidade Virtual, diversos tipos de mídias são manipulados em tempo de execução, como, por exemplo, modelos tridimensionais animados, sons e imagens. O próprio mundo virtual pode ser considerado uma mídia criada e editada através de uma ferramenta de modelagem própria do motor gráfico ou por uma das ferramentas de modelagem disponíveis no mercado, como 3D Studio Max® (Discreet, 2004), Lightwave 3D® (NewTek, 2004), Milkshape3D® (Chumbalum Soft, 2004) e Maya® (Alias, 2004).

Antes de integrarem a aplicação, as mídias utilizadas no mundo virtual devem ser carregadas a partir algum formato de arquivo, seja proprietário do motor gráfico ou nativo das ferramentas de edição utilizadas durante a confecção dessas mídias. Uma vez que uma determinada mídia esteja armazenada nas estruturas de dados adequadas, o núcleo ou um subsistema do motor gráfico é capaz de manipular essa mídia de acordo com as necessidades da aplicação. Uma imagem, por exemplo, pode ser carregada na memória principal, possibilitando sua manipulação através de um subsistema de processamento digital de imagens para a criação de uma textura bidimensional.

Como o desenvolvimento de uma ferramenta de edição foge ao escopo de um motor gráfico e de uma aplicação de RV, torna-se mais conveniente que o motor possua meios para a utilização de formatos nativos das ferramentas amplamente utilizadas pelos profissionais que confeccionam os mundos virtuais. Além disso, o motor deve oferecer um bom suporte a dados científicos gerados através de computações diretas (Bierbaun et al., 2001). Além disso, é importante que o motor ofereça mecanismos de extensão para suportar novos formatos utilizados pela camada de aplicação. Dentre as estratégias que o motor

gráfico pode adotar para a interação com ferramentas de edição de mídia, algumas são mais comumente utilizadas.

O suporte direto a formatos de arquivos proprietários torna o motor capaz de carregar as mídias de acordo com os padrões mais utilizados na indústria. Essa opção é atraente no sentido de facilitar o uso de formatos de mídia utilizados na maioria das ferramentas de edição e aproveitar melhor o conhecimento que os profissionais da equipe de desenvolvimento já possuam sobre as ferramentas que reconhecem esses formatos.

Alguns motores disponibilizam ferramentas para a conversão de arquivos em formatos externos para formatos conhecidos pelo motor gráfico insere uma etapa adicional, e inconveniente, no processo de geração e publicação de mídias para os ambientes virtuais. Além disso, é possível que aconteça uma perda na qualidade das mídias durante esse processo de conversão.

Uma estratégia bastante conveniente consiste na expansão de alguma ferramenta de edição popular através de *plugins*. Isso possibilita a exportação direta das mídias confeccionadas no editor para os formatos suportados pelo motor gráfico. Além disso, pode-se expandir a ferramenta de edição para que ofereça suporte à edição de características particulares ao ambiente virtual. Com isso, evita-se a escrita de uma ferramenta completa e aproveitam-se vários aspectos da ferramenta, como o kit de expansão e a interface com usuário. Como exemplo desse tipo de expansão, é possível escrever *plugins* para o 3D Studio MAX®, contendo novos tipos de entidades da cena que são mapeados em entidades do ambiente virtual, como no motor gráfico Fly3D (Fly3dD, 2004).

A construção de uma ferramenta de edição própria implica em esforços de desenvolvimento e de manutenção adicionais. As ferramentas próprias geralmente não são tão amigáveis nem oferecem tantos recursos quanto outras ferramentas disponíveis no mercado. Embora a ferramenta desenvolvida ofereça suporte às características particulares dos ambientes virtuais desenvolvidos com um determinado motor, essa ferramenta passa a rivalizar com as outras disponíveis há anos no mercado. Além disso, os profissionais que criam os ambientes virtuais devem ser treinados antes de utilizar a ferramenta desenvolvida.

3.5 Considerações Finais

Os motores gráficos são componentes de software utilizados no desenvolvimento de jogos e que também podem ser utilizados em sistemas de RV. Esse tipo de componente é capaz de encapsular a complexidade de utilização das soluções de baixo nível empregadas na construção de um ambiente virtual. Apesar de um motor gráfico ser uma solução de alto nível com bom desempenho, qualidade e confiabilidade, a maioria dos motores apresenta uma série problemas que impede a sua plena utilização numa aplicação de RV: baixa portabilidade, forte acoplamento a tecnologias, custo de utilização e código fonte fechado.

Um motor gráfico pode ser visto como a união de três módulos principais. O núcleo é o módulo primordial, que gerencia os subsistemas e provê funcionalidades básicas para os subsistemas e para a camada de aplicação. Dentre outras possíveis funcionalidades do núcleo, a capacidade de registrar extensões em tempo de execução abre uma série de possibilidades interessantes no contexto de desenvolvimento, como correção de falhas, manutenção e intercâmbio de tecnologias. Cada subsistema encapsula as tecnologias de baixo nível adequadas à implementação de um conjunto de funcionalidades da aplicação. A utilização de uma camada de abstração de tecnologias de baixo nível é um requisito importante num subsistema, pois melhora suas capacidades de configuração e portabilidade. O módulo de carregamento de elementos do mundo virtual é responsável pela mediação necessária ao carregamento das mídias utilizadas para construir um mundo virtual no motor gráfico, como arquivos de som, imagem e modelos tridimensionais pré-moldados. A divisão de um motor gráfico genérico nesses três módulos principais será utilizada durante o próximo capítulo comparar diferentes motores gráficos.

Capítulo 4

Análise dos Motores Gráficos Existentes

4.1 Introdução

Os motores gráficos são componentes de software concebidos, originalmente, para a criação de jogos tridimensionais. Atualmente, esse tipo de componente tem sido utilizado com sucesso no desenvolvimento de aplicações de Realidade Virtual (RV). A boa qualidade visual e a economia de memória e processamento são as principais vantagens na utilização de um motor gráfico para a construção de um ambiente de RV.

Nos capítulos anteriores, foram apresentados os principais aspectos que permeiam o desenvolvimento de sistemas de RV. Além disso, apresentou-se uma divisão de um motor gráfico genérico em três módulos principais, bem como as principais funcionalidades comumente oferecidas por esses módulos. Essa divisão será utilizada, neste capítulo, para apresentar e analisar os principais motores gráficos disponíveis atualmente.

O restante deste capítulo está organizado da seguinte maneira. Na Seção 4.2, os critérios que são usados para analisar os motores gráficos são definidos. Na Seção 4.3, são apresentados os principais motores gráficos disponíveis na atualidade, detalhando seus núcleos e subsistemas, assim como os elementos de mundo virtual que eles podem carregar e quaisquer de suas outras características especiais. A Seção 4.4 destina-se à análise comparativa entre os principais motores gráficos. Por fim, a Seção 4.5 destina-se à apresentação das considerações finais do capítulo.

4.2 Critérios de Análise

Os seguintes critérios são adotados para analisar os motores gráficos apresentados nas próximas seções: portabilidade, interface de programação, disponibilidade de código-fonte, capacidade de configuração, suporte a *plugins*, renderização em tempo-real, detecção de colisões, som espacial, linguagens de script, carregamento de mídias, qualidade audiovisual, desempenho, documentação e complexidade de utilização.

Portabilidade. A capacidade de execução em diferentes plataformas é uma característica fundamental para a utilização de um motor gráfico na construção de sistemas de RV, visto que tais sistemas geralmente não são projetados para execução em plataformas e configurações de hardware pré-definidas.

Interface de Programação. A interface que o programador utiliza para acessar as funcionalidades de um motor gráfico possui um papel importante no processo de desenvolvimento de um sistema. Atualmente, o paradigma de orientação a objetos tem-se estabelecido como uma ferramenta importante na construção de sistemas de grande porte, visto que a utilização de classes facilita a decomposição de tais sistemas em módulos menores e mais simples, o que contribui ainda para a manutenção e expansão desses sistemas. Além disso, o reuso de padrões de projeto orientados a objetos é uma alternativa atraente para facilitar ainda mais a modelagem de sistemas em geral (Gamma et al., 2002). Dessa maneira, é preferível que o motor possua uma interface de programação orientada a objetos, pois a interface procedimental geralmente é menos produtiva do que sua equivalente orientada a objetos. Além disso, a adoção de um motor gráfico num sistema orientado a objetos demandaria a decomposição desse motor em classes para facilitar a sua utilização no contexto do sistema.

Disponibilidade de Código-Fonte. A utilização de um motor de código-fonte fechado representa um risco em potencial para as aplicações desenvolvidas com o mesmo, caso o desenvolvimento desse motor seja interrompido. A detecção de uma falha crítica no motor durante o desenvolvimento de uma aplicação, por exemplo, causaria um grande impacto no projeto, visto que a falha não pode ser removida do motor. Além disso, a indisponibilidade de código-fonte prejudica a realização de customizações no motor e praticamente impede que esse motor seja portado para novas plataformas.

Capacidade de Configuração. Quando se pretende utilizar um motor gráfico em aplicações de propósito geral, é desejável que o motor possa ser configurado para que satisfaça os requisitos de desempenho e qualidade que uma determinada situação demanda. Dessa maneira, é importante que o motor possua mecanismos de configuração que permitam adequá-lo às necessidades do usuário ou da plataforma de execução, como, por exemplo, hardware gráfico disponível, memória e capacidade de processamento.

Suporte a *Plugins*. O suporte a *plugins* permite que o motor registre módulos em tempo de execução, o que confere ao motor grandes capacidades de expansão e manutenção. Quando um determinado módulo apresenta alguma falha ou desempenho insatisfatório, por exemplo, é suficiente realizar manutenção apenas no *plugin* referente àquele módulo. Assim, evita-se a recompilação do motor gráfico e das aplicações já desenvolvidas com o mesmo, o que seria extremamente inconveniente.

Renderização em Tempo-Real. Em uma aplicação de RV, o subsistema de renderização é responsável por sintetizar imagens das cenas tridimensionais a uma taxa interativa de quadros por segundo. Sob os aspectos de configuração e portabilidade, é desejável que esse subsistema seja independente de API gráfica. Além disso, é desejável esse subsistema seja capaz de utilizar técnicas customizadas pelo usuário para acelerar a renderização das cenas, permitindo que pesquisadores utilizem o motor gráfico para desenvolver novas abordagens para a renderização em tempo-real.

Detecção de Colisões. A detecção de colisões é um problema fundamental em diversas áreas como Computação Gráfica, Realidade Virtual, Simulações Físicas e Robótica. Para que esses tipos de aplicações e simulações apresentem resultados convincentes, faz-se necessário não apenas renderizar imagens realísticas, mas também modelar precisamente interações entre objetos. Portanto, algoritmos geométricos capazes de detectar interseções, computar distâncias e determinar superfícies de contato são fundamentais para a modelagem de situações em que ocorre o contato entre objetos ou entre partes do próprio objeto, como acontece em abalroamentos e grandes deformações.

Som Espacial. Em uma aplicação de RV, o estímulo auditivo é uma ferramenta capaz de introduzir drama nas cenas tridimensionais e aumentar o envolvimento do usuário com as situações simuladas no mundo virtual. Dessa maneira, é desejável que um motor gráfico seja capaz de simular as características sonoras de um ambiente virtual de maneira a realçar a sensação de imersão num ambiente virtual.

Linguagens de Script. As linguagens de script são instrumentos importantes no desenvolvimento de sistemas complexos e de grande porte, pois conferem a esses sistemas enormes capacidades de configuração, expansão e prototipagem. Dessa maneira, é desejável que um motor gráfico incorpore um interpretador de *scripts*, de maneira a oferecer uma interface de programação mais flexível.

Carregamento de Mídias. É desejável que um motor gráfico suporte diretamente formatos de arquivos popularmente utilizados para descrever os elementos do mundo virtual, ou, pelo menos, que disponibilize ferramentas para a conversão desses formatos de arquivos para aqueles reconhecidos pelo motor gráfico. Isso viabiliza o processo de criação de conteúdo para utilização no motor gráfico, e, além disso, permite a reutilização de mundos virtuais pré-existentes nesse motor.

Qualidade Audiovisual. A qualidade audiovisual é uma métrica importante para a construção de ambientes virtuais envolventes, pois o aspecto de representação audiovisual é fundamental para que o usuário sinta-se presente num mundo simulado por computador.

Desempenho. Atualmente, os sistemas de RV não são projetados de acordo com condições pré-estabelecidas de hardware ou software. Dessa maneira, é desejável que o motor gráfico utilizado pela aplicação de RV apresente bom desempenho, de maneira a oferecer experiências agraváveis aos usuários da aplicação. A utilização de estratégias eficientes contribui ainda uma portabilidade da aplicação sobre diferentes configurações de hardware, pois tais estratégias demandam menos memória e capacidade de processamento.

Documentação. A disponibilidade de uma documentação abrangente é importante para que novos usuários conheçam as funcionalidades de um motor gráfico e se familiarizem com a utilização do mesmo. Essa documentação deve incluir instruções de instalação do motor, bem como os pré-requisitos mínimos para a utilização do mesmo. Além disso, é desejável que exemplos e tutoriais com código-fonte também estejam disponíveis, pois funcionam como a principal referência para quem deseja aprender a utilizar um motor gráfico.

Complexidade de Utilização. É importante que as funcionalidades de um motor gráfico possam ser acessadas de uma maneira simples e intuitiva, permitindo que os usuários sejam capazes de entender facilmente os principais mecanismos de funcionamento desse motor. Além disso, é conveniente que a utilização do motor dispense treinamento intensivo ou o conhecimento profundo das linguagens de programação adotadas, permitindo a utilização do motor por programadores menos experientes e em projetos de curta duração.

4.3 Principais Motores Gráficos

Esta seção destina-se à apresentação e à análise dos principais motores gráficos disponíveis na atualidade, à luz dos critérios apresentados na seção anterior. Além disso, serão apresentadas outras características importantes, particulares de cada motor gráfico.

Os motores aqui apresentados destacam-se dos demais motores existentes pelas seguintes características: maturidade, utilização em larga escala, desempenho, qualidade audiovisual, inovação tecnológica, portabilidade, capacidade de expansão e desenvolvimento como projeto de Software Livre.

Todos os motores gráficos analisados apresentam deficiências quanto à interface com dispositivos de entrada, pois, de maneira geral, não prevêem a utilização de dispositivos não-convencionais. Isso porque são projetados para utilizar apenas dispositivos típicos dos jogos de computador: mouse, teclado e *joysticks*.

4.3.1 3DSTATE

Até recentemente conhecido como Morfit, o motor proprietário 3DSTATE é amplamente utilizado, sendo bastante maduro e voltado para o desenvolvimento de jogos e aplicações tridimensionais interativas (Berger, 2002). Esse motor deve ser licenciado para utilização em projetos comerciais, e, também, possui versões educacionais que podem ser utilizadas em projetos de pesquisa. 3DSTATE permite inserir aplicações 3D como conteúdo em páginas da *Web* através de um *plugin* que é instalado no Internet Explorer. A Figura 4.1 ilustra exemplos de aplicações utilizando esse motor gráfico e o editor de mundos virtuais, a ele, integrado.

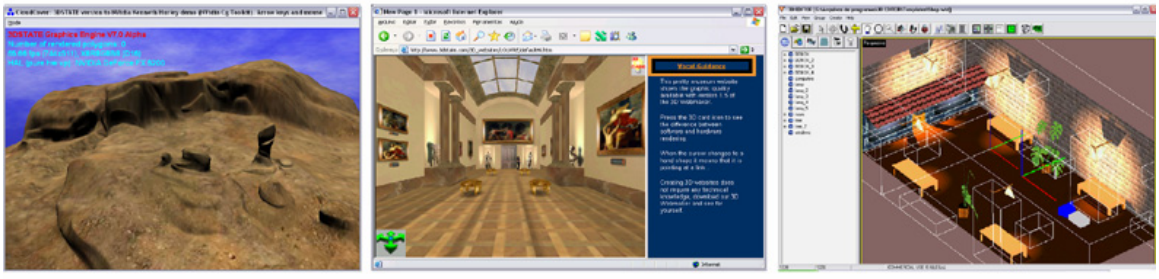


Figura 4.1: Duas aplicações usando o motor 3DSTATE: uma stand-alone (à esquerda) e outra na Web (ao centro). À direita, o editor de mundos integrado ao motor.

Portabilidade. A portabilidade do motor 3DSTATE é praticamente inexistente, restringindo-se a diferentes versões da plataforma Windows.

Interface de Programação. 3DSTATE é um motor procedimental cujas funcionalidades são implementadas numa biblioteca de vínculo dinâmico que pode ser acessada através de várias linguagens de programação, incluindo C/C++.

Disponibilidade de Código-Fonte. 3DSTATE é um motor comercial de código-fonte fechado.

Capacidade de Configuração. Esse motor utiliza um conjunto bem definido de tecnologias, e praticamente não pode ser configurado.

Suporte a *Plugins*. A estratégia de evolução desse motor baseia-se no suporte à interface padrão da versão anterior. Assim, para atualizar o motor, o desenvolvedor precisa apenas trocar a DLL atual por uma mais recente, evitando a reescrita do código que utiliza 3DSTATE. Entretanto, esse motor não suporta *plugins*.

Renderização em Tempo-Real. A interface com o hardware gráfico é feita através da API Direct3D. Esse motor encapsula o processo de renderização da cena, sem, no entanto, oferecer um controle adequado sobre esse processo. 3DSTATE apresenta uma taxa razoável de quadros por segundo devido à utilização de árvores BSP para representar o cenário do ambiente. No entanto, após a realização de testes com esse motor, notou-se que a estratégia implementada por 3DSTATE não é adequada à renderização de cenas externas. Entretanto, é possível utilizar uma matriz de visibilidade, construída manualmente pelo próprio usuário do motor, para acelerar os cálculos de visibilidade em determinados setores do cenário. Apesar disso, a construção dessa matriz de visibilidade deveria ser realizada automaticamente pelo próprio motor gráfico.

Detecção de Colisões. O algoritmo utilizado para a detecção de colisões é capaz de ignorar interseções nas partes transparentes de um polígono texturizado e apresenta bom desempenho. Porém, esse algoritmo é muito limitado, pois testa apenas segmentos de reta contra a geometria da cena. Embora esse teste seja adequado ao processo de *picking*, ele não é apropriado à detecção de colisão entre objetos complexos, o que exigiria o desenvolvimento de algoritmos mais elaborados no topo do motor gráfico.

Som Espacial. 3DSTATE possui suporte a som tridimensional através da biblioteca FMOD (Firelight Technologies, 2004), que apresenta bom desempenho e qualidade

razoável, além de suportar diversos formatos de áudio. Porém, essa biblioteca deve ser licenciada à parte quando distribuída em aplicações comerciais, sendo que, atualmente, sua licença é dez vezes mais cara do que a licença do próprio motor 3DSTATE.

Linguagens de Script. O motor gráfico 3DSTATE não possui um interpretador de scripts integrado.

Carregamento de Mídias. Esse motor suporta formatos de imagem populares, como BMP, GIF e JPEG. Além disso, um pacote de aplicações de suporte é disponibilizado no site desse motor, incluindo tradutores de formatos de arquivo e um modelador de mundos virtuais construído com o próprio motor gráfico.

Qualidade Audiovisual. O motor 3DSTATE é capaz de renderizar cenas com qualidade audiovisual razoável, pois, dentre outras coisas, não permite utilizar os modelos de iluminação comumente suportados pelo hardware gráfico. Além disso, o modelo sonoro suportado pela biblioteca FMOD é bastante limitado.

Desempenho. 3DSTATE apresenta desempenho razoável, não comprometendo a execução das aplicações que utilizam cenas com poucos milhares de polígonos. Em particular, o processo de carregamento de mundos virtuais complexos nesse motor é bastante demorado, mesmo quando a estrutura da cena é obtida a partir de um arquivo específico.

Documentação. 3DSTATE é um motor bem documentado que possui uma quantidade significativa de exemplos acompanhados de código-fonte.

Complexidade de Utilização. Apesar de utilizar uma interface de programação procedimental, é muito simples acessar as funcionalidades do motor 3DSTATE, pois elas encontram-se distribuídas em APIs menores e bem projetadas.

4.3.2 Genesis3D

Genesis3D é um motor gráfico distribuído gratuitamente que tem sido usado no desenvolvimento de jogos e aplicações 3D (Genesis3D, 2004). Esse motor conta com um sistema de arquivos virtual, possibilitando o carregamento de mídias a partir do sistema de arquivos e a partir de arquivos compactados, de maneira transparente para o programador. Essa característica facilita a distribuição e a manutenção de mundos virtuais, pois todas as mídias que compõem um mundo podem ser agrupadas num único arquivo. A Figura 4.2

ilustra uma aplicação construída com o motor Genesis3D e o editor de mundos, a ele, integrado.

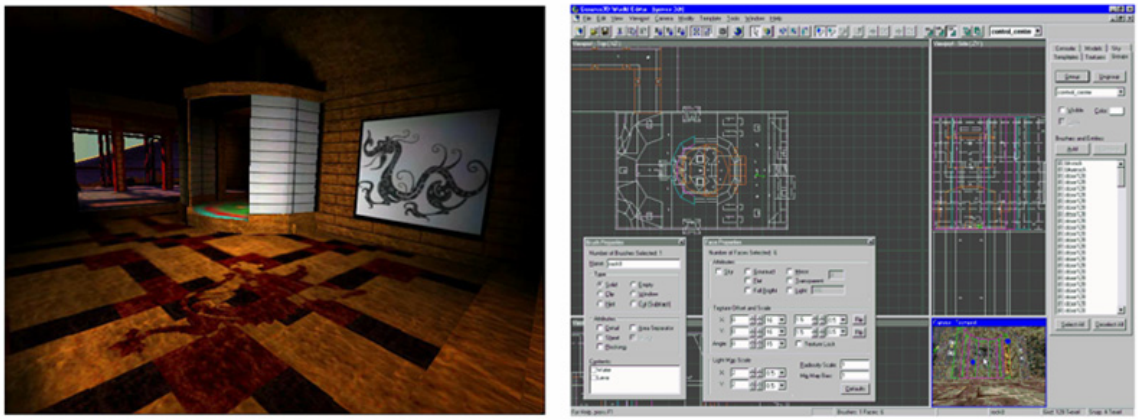


Figura 4.2: Uma aplicação construída com Genesis3D (à esquerda) e seu editor de mundos integrado (à direita).

Portabilidade. Genesis3D permite desenvolver aplicações apenas na plataforma Windows.

Interface de Programação. Genesis3D é um motor procedimental, cujas funcionalidades podem ser acessadas a partir das linguagens C/C++, Delphi e Visual Basic.

Disponibilidade de Código-Fonte. O código-fonte do motor Genesis3D é distribuído de acordo com os termos da licença GPL.

Capacidade de Configuração. Similarmente ao motor 3DSTATE, Genesis3D praticamente não apresenta capacidade de configuração.

Suporte a Plugins. O motor Genesis3D não suporta o carregamento de *plugins*. A estratégia de evolução adotada nesse motor baseia-se na realização de modificações em seu código fonte e na criação de módulos adicionais construídos no topo do motor.

Renderização em Tempo-Real. O subsistema de renderização adota a API Direct3D como interface única com o hardware gráfico. Nesse subsistema, o usuário pode controlar manualmente a renderização de menus e objetos, caso julgue necessário. Para acelerar a pintura das cenas, Genesis3D utiliza técnicas baseadas em árvores BSP. É possível incrementar a qualidade visual dos modelos através de mapeamentos de normais e de rugosidade (*bump*) para obter um modelo de iluminação mais convincente. Genesis3D permite utilizar curvas paramétricas na descrição do cenário, e, além disso, esse motor permite renderizar eficientemente terrenos gerados a partir de imagens.

Detecção de Colisões. O subsistema de detecção de colisões utiliza um algoritmo próprio, permitindo realizar testes de AABBs e segmentos de reta contra a geometria da cena, suportando também modelos deformáveis. Essa abordagem, apesar de ser mais apropriada do que a utilizada pelo motor 3D STATE, é bastante limitada, pois não permite determinar interseções exatas entre dois modelos geométricos.

Som Espacial. O motor Genesis3D suporta a definição de sons bidimensionais e tridimensionais nas cenas, com suporte a efeitos Doppler, espacialização e obstrução.

Linguagens de Script. Genesis3D não possui um subsistema de script integrado.

Carregamento de Mídias. A definição dos elementos do mundo virtual é baseada no carregamento de mundos, imagens, modelos tridimensionais e sons pré-amostrados a partir de arquivos em formatos próprios desse motor ou noutros formatos popularmente utilizados pelos profissionais que criam essas mídias. Além disso, esse motor disponibiliza um editor de mundos integrado.

Qualidade Audiovisual. O motor Genesis3D é capaz de gerar cenas com uma boa qualidade audiovisual. Esse motor conta com técnicas de renderização sofisticadas (até mais sofisticadas do que as utilizadas pelo motor 3DSTATE), e, além disso, o modelo sonoro utilizado por esse motor suporta efeitos Doppler, espacialização e atenuação.

Desempenho. Genesis3D integra técnicas para a renderização de cenários interiores e exteriores, obtendo um desempenho significativamente maior do que o apresentado pelo motor 3DSTATE, mantendo uma taxa satisfatória de quadros por segundo. Apesar disso, é recomendável a utilização de hardware gráfico recente para garantir um bom desempenho.

Documentação. Apesar de disponibilizar com vários exemplos com código-fonte, a documentação desse motor é bastante superficial.

Complexidade de Utilização. Embora Genesis3D disponibilize exemplos com código-fonte, esse motor define uma interface de programação procedimental bastante complexa.

4.3.3 Quake III

O motor gráfico criado para o desenvolvimento do jogo Quake III (Id Software, 2004) é, atualmente, um dos mais populares. Esse motor possui um editor de mundos integrado e pode ser utilizado livremente em projetos não-comerciais, sob os termos da licença GPL,

sendo que a sua utilização em projetos comerciais demanda a aquisição de uma licença específica.

O motor Quake III é implementado em C, apresentando como principais atrativos o excelente desempenho, a boa qualidade audiovisual e a portabilidade entre plataformas PC. A Figura 4.3 ilustra o jogo Quake III e o editor de mundos integrado.

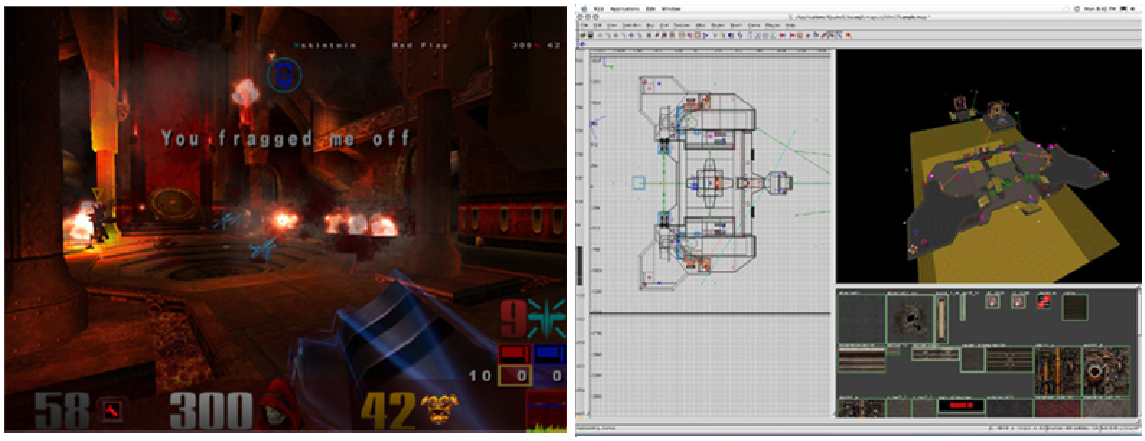


Figura 4.3: O jogo Quake III (à esquerda) e o editor de mundos integrado (à direita).

Portabilidade. O motor Quake III oferece portabilidade entre as plataformas Windows, Linux/Unix e MacOS.

Interface de Programação. Esse motor disponibiliza suas funcionalidades através de uma interface de programação procedimental.

Disponibilidade de Código-Fonte. O código-fonte do motor Quake III é distribuído de acordo com os termos da licença GPL.

Capacidade de Configuração. O motor Quake III pode ser configurado para ajustar-se às necessidades de desempenho e à qualidade do usuário. Além disso, o interpretador de scripts integrado é uma peça fundamental na arquitetura desse motor, o que lhe confere grandes capacidades de configuração e expansão.

Suporte a Plugins. No motor Quake III, as aplicações são carregadas como *plugins* e, tem tempo de execução, trocam mensagens com o motor gráfico. Dessa maneira, ao invés da camada de aplicação coordenar a utilização das funcionalidades desse motor gráfico, acontece justamente o contrário: o motor é quem coordena a execução das aplicações.

Renderização em Tempo-Real. No motor Quake III, a interface com o hardware gráfico é baseada na API OpenGL, disponível em quase todas as plataformas. Praticamente todos os

recursos da pipeline *fixed function* são utilizados na definição de materiais, o que permite criar efeitos visuais que combinam qualidade e desempenho. Malhas deformáveis, *billboards*, sistemas de partículas e curvas paramétricas podem ser utilizados na descrição da cena, cuja pintura é acelerada pela combinação das técnicas PVS, *Frustum Culling* e árvores BSP, resultando um excelente desempenho em cenários fechados.

Detecção de Colisões. O subsistema de detecção de colisões disponibiliza testes de interseção entre a geometria da cena e volumes convexos, bem como entre a geometria da cena e segmentos de reta. Caso a interseção exista, o ponto de interseção exato é calculado. Embora esses testes suportem superfícies paramétricas e modelos deformáveis, os mesmos não suportam o cálculo de interseções exatas entre dois modelos geométricos.

Som Espacial. O motor Quake III possui um subsistema de som espacial com suporte à espacialização e à atenuação de sons, além do suporte ao efeito Doppler.

Linguagens de Script. O motor Quake III conta com um poderoso subsistema de scripts, que utiliza uma linguagem interpretada própria e é amplamente utilizado nos demais módulos do motor. Além disso, é possível interpretar scripts interativamente durante a execução de aplicações através de um console virtual.

Carregamento de Mídias. O motor Quake III baseia-se num sistema de arquivos virtual para localizar e carregar os elementos do mundo virtual. Todos os elementos tridimensionais do mundo virtual são carregados a partir de arquivos em formatos próprios desse motor. Assim, a elaboração de conteúdos para esse motor demanda a utilização de seu editor de mundos integrado ou de ferramentas para converter formatos de arquivos.

Qualidade Audiovisual. O motor Quake III apresenta uma qualidade visual muito boa, devido à utilização de praticamente todos os recursos da pipeline *fixed funcion* para a renderização das cenas e pelos efeitos visuais oferecidos. Além disso, o modelo sonoro adotado suporta efeitos Doppler, espacialização e atenuação.

Desempenho. O motor Quake III prima pelo excelente desempenho, principalmente quando cenários fechados são utilizados.

Documentação. A documentação desse motor é pouco acessível – praticamente nenhum exemplo com código-fonte pode ser encontrado na *Web* e a maioria das informações sobre esse motor provém de fontes não-oficiais.

Complexidade de Utilização. O motor Quake III é bastante complexo e sua interface de programação procedimental utiliza recursos avançados da linguagem C, o que dificulta ainda mais a sua utilização. Além disso, as aplicações são módulos secundários que exercem um controle limitado sobre o motor gráfico.

4.3.4 Fly3D

Fly3D é um motor gráfico voltado para a criação de jogos tridimensionais e aplicações 3D em tempo-real (Fly3D, 2004), sendo distribuído gratuitamente na forma de um kit de desenvolvimento de software (*Software Development Kit*, SDK) relativamente maduro escrito em C/C++. Assim como no motor 3DSTATE, é possível distribuir aplicações tridimensionais desenvolvidas com Fly3D através de páginas da *Web*. A Figura 4.4 ilustra duas aplicações desenvolvidas com Fly3D: uma distribuída na *Web* e outra sendo executada no ambiente Windows.

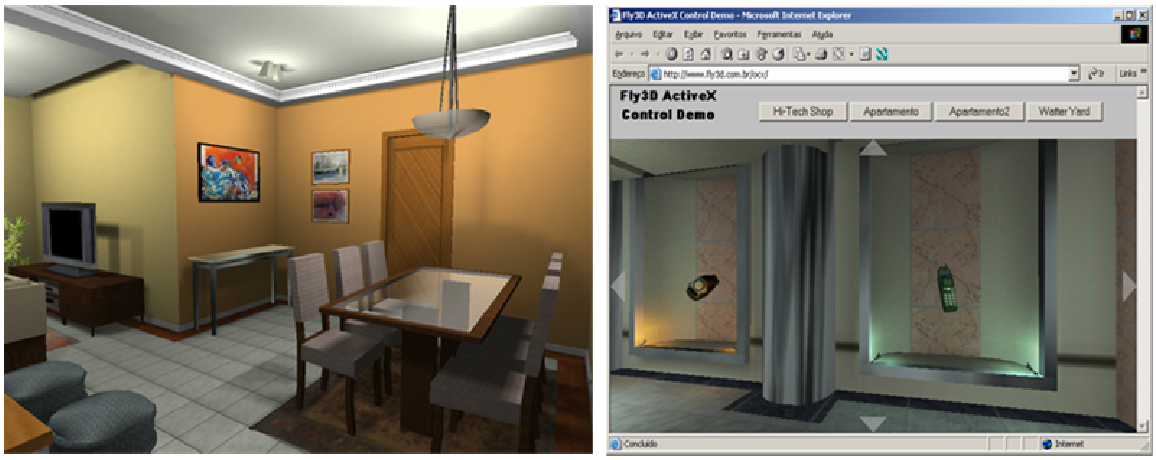


Figura 4.4: Uma aplicação stand-alone (à esquerda) e uma aplicação web (à direita) desenvolvidas com o motor Fly3D.

Portabilidade. A portabilidade do motor Fly3D restringe-se à plataforma Windows.

Interface de Programação. As funcionalidades do motor Fly3D são acessadas através de uma interface orientada a objetos, disponível em C++.

Disponibilidade de Código-Fonte. O código-fonte do motor Fly3D é distribuído de acordo com os termos da licença GPL.

Capacidade de Configuração. Esse motor permite modificar parâmetros das aplicações desenvolvidas em tempo de execução através de uma interface gráfica e permite registrar variáveis num console virtual.

Suporte a Plugins. Fly3D utiliza o mesmo princípio adotado no motor Quake III, onde cada *plugin* funciona como uma aplicação cuja execução é coordenada através da troca de mensagens.

Renderização em Tempo-Real. O subsistema de renderização adota a API OpenGL como interface única com o hardware gráfico. A cena é representada através de uma árvore BSP, que é utilizada em conjunto com as técnicas de PVS e *Frustum Culling* para acelerar a renderização. Esse motor permite utilizar superfícies adaptativas de Bézier na descrição das cenas, e, além disso, Fly3D suporta efeitos visuais como sombras dinâmicas e sistemas de partículas. Além disso, é possível que o usuário do motor influa na renderização das cenas através de comandos OpenGL, embora os objetos pintados dessa forma pareçam "etéreos" com relação aos demais objetos presentes na cena.

Detecção de Colisões. Fly3D utiliza algoritmos proprietários para o cálculo de interseções, suportando testes eficientes entre a geometria da cena e *bounding boxes*, sendo que a geometria dos objetos dinâmicos é representada internamente através de *Octrees*. Apesar disso, esse motor não suporta o cálculo de interseções exatas entre dois modelos geométricos.

Som Espacial. O subsistema de som tridimensional desse motor é baseado na API DirectSound (DirectX, 2004) e suporta efeitos Doppler e EAX, além da espacialização e atenuação sonora.

Linguagens de Script. Fly3D possui um pseudo-interpretador de scripts integrado, cuja principal funcionalidade consiste no registro e acesso de variáveis num console virtual.

Carregamento de Mídias. Fly3D possui um sistema de arquivos virtual e suporta os formatos de imagem BMP, JPEG e TGA, sendo que os modelos tridimensionais são carregados a partir de arquivos em formatos proprietários. Fly3D disponibiliza um pacote de ferramentas para a parametrização de aplicações em tempo de execução, a conversão de formatos de arquivos, além de um editor de materiais integrado e um *plugin* para a importação de cenas criadas no software 3D Studio MAX (Discreeet, 2004).

Qualidade Audiovisual. Fly3D é capaz de produzir cenas com uma boa qualidade audiovisual, comparável à obtida pelo motor Quake III.

Desempenho. Segundo Elias (2002), esse motor apresenta desempenho insatisfatório e demanda a utilização de uma boa placa gráfica. Os problemas de desempenho ocorrem, principalmente, quando cenários parcialmente abertos são utilizados.

Documentação. A documentação distribuída no site desse motor é superficial e, em geral, desatualizada (DevMaster, 2004).

Complexidade de Utilização. Apesar de utilizar um design orientado a objetos, esse motor encontra-se estruturado de uma maneira inadequada: várias funcionalidades do núcleo encontram-se espalhadas em diversos *plugins*, que são utilizados por outros módulos como se fizessem parte da interface padrão do motor gráfico, afetando negativamente a modularidade do motor. Além disso, as aplicações são desenvolvidas como módulos secundários do motor gráfico, o que reduz consideravelmente o controle do usuário sobre o mesmo.

4.3.5 Irrlicht

Irrlicht é um motor gráfico orientado a objetos, escrito em C++, que tem sido usado no desenvolvimento de jogos e de aplicações científicas (Irrlicht, 2004). Embora esse motor seja fruto de um projeto de código aberto, apenas um grupo restrito de desenvolvedores pode contribuir para a evolução do motor. Apesar de suas interessantes funcionalidades, de sua portabilidade, de seu desempenho, de sua qualidade e de sua flexibilidade, o mesmo não possui os subsistemas de som espacial e de script. Além disso, Irrlicht não possui um mecanismo de extensão apropriado. A Figura 4.5 ilustra uma aplicação utilizando o motor Irrlicht e o seu subsistema de interface gráfica.

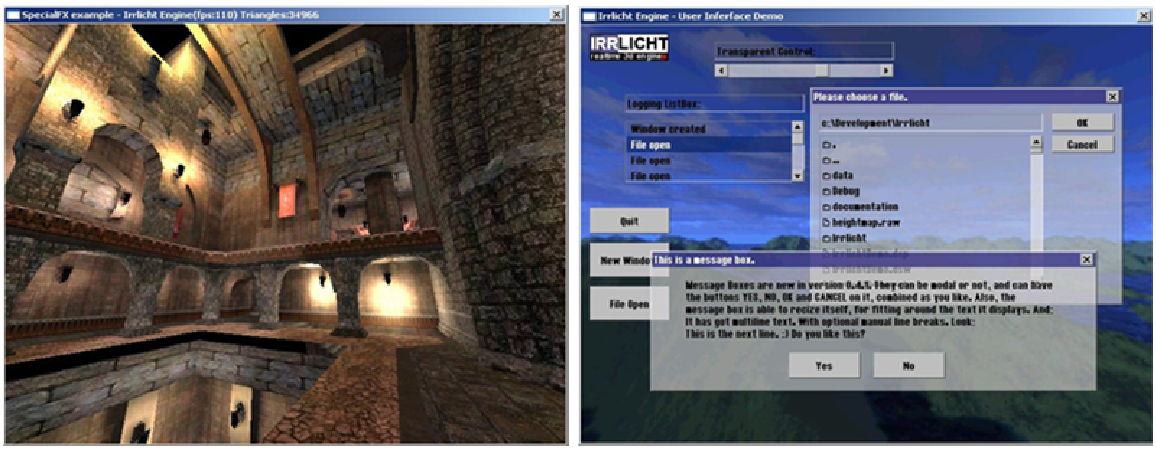


Figura 4.5: Um mundo virtual carregado no motor Irrlicht (à esquerda) e seu subsistema de interface gráfica (à direita).

Portabilidade. Irrlicht oferece portabilidade entre as plataformas Windows e Linux.

Interface de Programação. As funcionalidades do motor Irrlicht são disponibilizadas através de uma interface orientada a objetos em C++.

Disponibilidade de Código-Fonte. Irrlicht é distribuído gratuitamente sob os termos da licença zlib (Zlib, 2004), podendo ser utilizado livremente inclusive em projetos comerciais.

Capacidade de Configuração. Apesar de abstrair a API gráfica utilizada para enviar primitivas ao *frame buffer*, o motor Irrlicht apresenta uma capacidade de configuração bastante limitada, pois esse motor baseia-se num conjunto bem definido de tecnologias.

Suporte a Plugins. Irrlicht não oferece suporte ao carregamento de *plugins*. A estratégia de evolução adotada por esse motor baseia-se em manutenções no código-fonte e na construção de novos módulos no topo do motor.

Renderização em Tempo-Real. Irrlicht independe de API gráfica, sendo capaz de utilizar, transparentemente, OpenGL, DirectX ou uma implementação via software. O subsistema de renderização integra várias técnicas para a representação visual, como sombras dinâmicas, sistemas de partículas e animação esqueletal de personagens. Esse motor utiliza técnicas baseadas em *Frustum Culling* hierárquico, *Occlusion Culling* e *Octrees* para acelerar a pintura de cenários abertos e fechados.

Detecção de Colisões. A detecção de colisões é baseada em testes entre *bounding boxes*, semi-retas, segmentos de reta e triângulos. O suporte à operação de *picking* com nível de

detalhe é outra funcionalidade interessante desse subsistema. Apesar disso, não é possível detectar interseções exatas entre dois modelos geométricos.

Som Espacial. Irrlicht não oferece suporte a som espacial.

Linguagens de Script. Irrlicht não possui um subsistema de script integrado.

Carregamento de Mídias. O motor Irrlicht implementa um sistema de arquivos virtual baseado na biblioteca zlib e suporta diretamente vários formatos de arquivos amplamente utilizados, o que facilita o carregamento de mídias nesse motor gráfico.

Qualidade Audiovisual. Como resultado da integração de vários efeitos visuais, Irrlicht é capaz de renderizar cenários abertos e fechados com uma boa qualidade visual.

Desempenho. Irrlicht combina técnicas de aceleração da renderização de maneira a obter uma boa taxa de quadros por segundo em cenários abertos, fechados e mistos.

Documentação. O motor Irrlicht possui uma excelente documentação, além de uma grande quantidade de tutoriais e exemplos com código-fonte disponibilizados através da internet.

Complexidade de Utilização. A simplicidade de utilização é uma característica marcante desse motor, pois, diferentemente da maioria dos motores existentes, não depende de outras bibliotecas e sua interface de programação é bastante intuitiva.

4.3.6 CrystalSpace

O motor CrystalSpace tem sido utilizado com sucesso em alguns jogos e aplicações de computação gráfica interativa, sendo que a sua característica multiplataforma tem atraído a atenção de desenvolvedores (CrystalSpace, 2004). Atualmente, esse motor encontra-se numa versão relativamente estável (0.98), sendo distribuído gratuitamente na forma de um SDK que contém o código-fonte e a documentação do motor, além de algumas ferramentas para a conversão de formatos de arquivo.

CrystalSpace baseia-se na orientação a *plugins* como principal estratégia de expansão e configuração. Na arquitetura adotada, cada subsistema é projetado para ser independente dos demais e, portanto, intercambiável. A Figura 4.6 ilustra dois jogos desenvolvidos com o motor CrystalSpace.

Figura 4.6: Dois jogos construídos com o motor CrystalSpace.

Portabilidade. CrystalSpace permite desenvolver aplicações nas plataformas Windows, Linux e MacOS.

Interface de Programação. CrystalSpace oferece uma interface de programação orientada a objetos, acessada através da linguagem C++.

Disponibilidade de Código-Fonte. O motor CrystalSpace é desenvolvido como um projeto de software livre e pode ser utilizado nos termos da licença LGPL, o que garante sua livre utilização, inclusive em projetos comerciais.

Capacidade de Configuração. CrystalSpace é bastante flexível, permitindo que cada um de seus módulos seja configurado separadamente.

Suporte a Plugins. CrystalSpace suporta a utilização de *plugins* para registrar módulos adicionais. Várias funcionalidades de seu próprio núcleo também são implementadas em *plugins*.

Renderização em Tempo-Real. O subsistema de renderização é independente de API gráfica, podendo utilizar OpenGL, Direct3D ou uma implementação via software. Para acelerar a pintura das cenas, esse subsistema integra técnicas baseadas em portais, K-d trees e *Occlusion Culling*.

Detecção de Colisões. O subsistema de detecção de colisões desse motor é capaz de testar interseções exatas entre modelos tridimensionais e se abstrai da técnica utilizada. Atualmente, esse subsistema possui duas implementações distintas, baseadas nas bibliotecas OPCODE e RAPID (Terdiman, 2003; Gottschalk et al., 1996). A biblioteca OPCODE suporta eficientemente testes entre modelos deformáveis, embora não suporte

adequadamente a aplicação de escalas em tais modelos. A biblioteca RAPID é, em alguns casos, mais eficiente do que OPCODE, embora não ofereça um suporte eficiente a modelos deformáveis e à aplicação de escalas. Na prática, isso significa que, atualmente, CrystalSpace apresenta limitações que poderiam ser contornadas através da utilização de outras técnicas para o cálculo de interseções.

Som Espacial. O subsistema de som espacial oferece suporte aos efeitos de espacialização, atenuação e Doppler. Além disso, esse subsistema suporta efeitos EAX e o carregamento de sons pré-amostrados a partir de diversos formatos de arquivo.

Linguagens de Script. O subsistema de scripts, que é independente de linguagem, permite executar trechos de código, bem como alterar e recuperar o valor de variáveis no interpretador de scripts. Entretanto, esse subsistema não é capaz de armazenar scripts num formato pré-compilado, o que permitiria interpretar scripts freqüentemente utilizados de uma maneira mais eficiente.

Carregamento de Mídias. CrystalSpace implementa um sistema de arquivos virtual e suporta vários formatos de arquivo popularmente utilizados para imagens e modelos tridimensionais. Além disso, esse motor disponibiliza ferramentas para a conversão de vários formatos de arquivo.

Qualidade Audiovisual. CrystalSpace permite utilizar diversos efeitos visuais que conferem uma boa qualidade visual às cenas renderizadas. Além disso, esse motor adota um modelo sonoro que suporta espacialização e atenuação do som, além de efeitos EAX.

Desempenho. Apesar de integrar diversas técnicas para acelerar a pintura das cenas, CrystalSpace implementa, em software, parte da pipeline de renderização de baixo nível. Sob várias circunstâncias, essa pipeline híbrida compromete o desempenho desse motor. Segundo Elias (2002), de fato, esse motor apresenta desempenho insuficiente.

Documentação. CyrstalSpace possui uma documentação abrangente que inclui exemplos com código-fonte.

Complexidade de Utilização. No motor CrystalSpace, a implementação das funcionalidades do núcleo encontra-se distribuída entre diversos *plugins*, que geralmente são escritos em função de outros *plugins*. Assim, estabelecem-se dependências entre *plugins* que afetam a modularidade do motor. A enorme quantidade de módulos desse motor também dificulta a sua compreensão e utilização.

O principal autor desse motor admite que, embora CrystalSpace seja flexível e repleto de funcionalidades, não é um motor de fácil aprendizado e utilização (DevMaster, 2004). Ele afirma ainda que a documentação, apesar ser adequada aos propósitos de desenvolvimento, não está atualizada como deveria.

4.3.7 Unreal Engine 3

Segundo DevMaster (2004), o motor gráfico Unreal Engine 3 tem-se estabelecido como uma solução completa para o desenvolvimento de jogos tridimensionais comerciais. Esse motor é projetado sob o paradigma da orientação a objetos e implementado em C++, destacando-se dos demais motores pelo desempenho, portabilidade e excelente qualidade audiovisual que são proporcionados às aplicações desenvolvidas.

Todos os componentes de Unreal Engine 3 foram projetados em conjunto, de maneira a prover interoperabilidade entre componentes através de uma interface de programação consistente. Esse motor pode ser utilizado gratuitamente para treinamento e em projetos de pesquisa. Apesar disso, qualquer produto criado com Unreal Engine 3 não pode ser redistribuído. Esse motor possui todos os subsistemas básicos, além dos subsistemas de interface gráfica, simulação da dinâmica dos corpos rígidos articulados, comunicação em rede e inteligência artificial.

A distribuição comercial do Unreal Engine 3 inclui sua documentação e uma série de ferramentas para a edição de mundos, propriedades físicas dos objetos, sons espaciais, criação de animações através de cinemática, além de um ambiente para identificação de "gargalos" (*profiling*) em aplicações e depuração de código. Na Figura 4.7, estão ilustrados o jogo Unreal Tournament 2004, desenvolvido com Unreal Engine 3, e o editor de mundos integrado que utiliza o paradigma WYSIWYG (*What You See Is What You Get*).

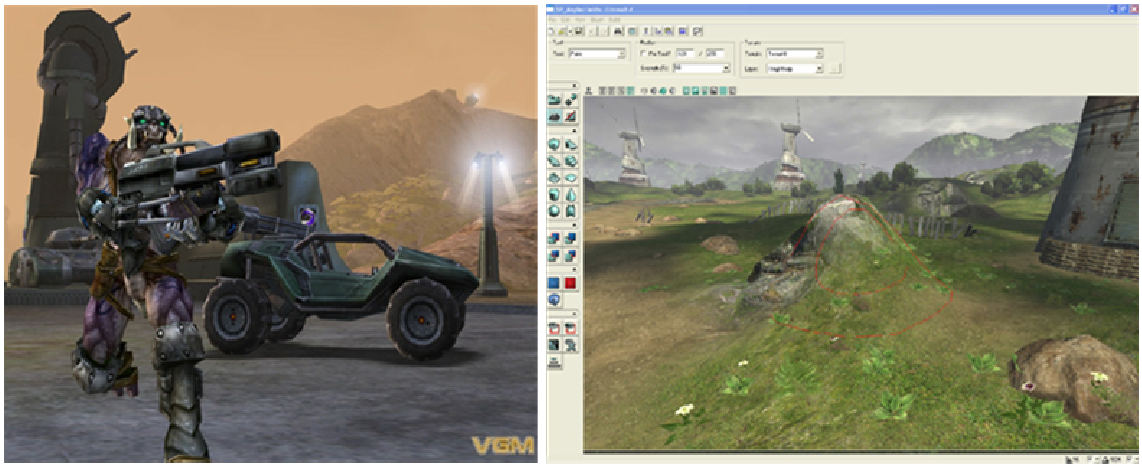


Figura 4.7: À esquerda, uma imagem do jogo Unreal Tournament. À direita, o editor integrado no motor Unreal Engine 3.

Portabilidade. Unreal Engine 3 oferece portabilidade entre as plataformas Windows, Linux e MacOS, além das plataformas de console Nintendo® GameCube®, Sony® Playstation2® e Microsoft Xbox®.

Interface de Programação. As funcionalidades do motor Unreal Engine 3 são acessadas através de uma interface orientada a objetos, disponível tanto em C++ quanto em UnrealScript, uma linguagem própria desse motor.

Disponibilidade de Código-Fonte. Unreal Engine 3 é um motor gráfico comercial de código-fonte fechado.

Capacidade de Configuração. O motor Unreal Engine 3 possui grande capacidade de configuração, permitindo que o usuário estabeleça um paralelo entre desempenho e qualidade. Além disso, o interpretador integrado permite realizar configurações do motor em tempo de execução.

Suporte a Plugins. A estratégia de evolução adotada por Unreal Engine 3 baseia-se na orientação a *plugins* e na utilização de *scripts*.

Renderização em Tempo-Real. O subsistema de renderização desse motor é independente de API gráfica, e, atualmente, suporta OpenGL, Direct3D e uma implementação via *software*. Unreal Engine 3 integra vários efeitos visuais, conferindo excelente qualidade à representação visual das cenas. Para acelerar a pintura das cenas, esse motor integra técnicas baseadas em árvores BSP, portais, *Occlusion Culling*, PVS, LOD estático e LOD contínuo. Além disso, esse subsistema utiliza um grafo dinâmico de cena, que é uma

combinação das técnicas de portais e grafo de cena consistindo em uma coleção de objetos nas imediações do observador.

Detecção de Colisões. O subsistema de detecção de colisões disponibiliza testes otimizados que utilizam cilindros como uma aproximação da geometria dos personagens humanóides e demais objetos do ambiente. Também é possível utilizar testes baseados em volumes convexos para lidar com regiões geometricamente complexas do cenário. Além disso, esse subsistema suporta *picking* e testes de interseção entre segmentos de reta e a geometria da cena. Entretanto, não é possível detectar interseções exatas entre um par de modelos geométricos.

Som Espacial. O subsistema de som espacial do motor Unreal Engine 3 implementa o padrão OpenAL nas plataformas Windows, Linux e MacOS, com suporte à espacialização e atenuação dos sons, além dos efeitos Doppler. Além disso, oferece suporte a efeitos EAX 3.0 em dispositivos de áudio compatíveis e à reprodução de som ambiente (não espacial), além de *streaming* Ogg Vorbis (Ogg vorbis, 2004).

Linguagens de Script. Unreal Engine 3 possui um interpretador integrado que possibilita utilizar UnrealScript, uma linguagem interpretada muito semelhante a Java e capaz de acessar praticamente todas as funcionalidades desse motor.

Carregamento de Mídias. No motor Unreal Engine 3, os elementos do mundo virtual são carregados a partir de arquivos em formatos proprietários, demandando a utilização do editor de mundos integrado ou a utilização de ferramentas de conversão de formatos de arquivo.

Qualidade Audiovisual. O motor Unreal Engine 3 destaca-se dos demais motores gráficos pela excelente qualidade audiovisual de seus mundos virtuais, que é resultado da implementação do padrão de áudio espacial OpenAL e dos mais modernos efeitos visuais.

Desempenho. O excelente desempenho desse motor deve-se à integração das mais modernas técnicas de aceleração, simulação e detecção de colisões. Entretanto, a utilização desse motor requer uma máquina potente e hardware gráfico de última geração.

Documentação. Muito pouca documentação do motor UnrealEngine 3 pode ser encontrada gratuitamente, sendo necessário adquirir uma licença para ter acesso à documentação oficial desse motor.

Complexidade de Utilização. Dada a complexidade do motor Unreal Engine 3, sua utilização requer treinamento nas ferramentas de modelagem integradas e um bom nível técnico da equipe de desenvolvimento.

4.4 Análise Comparativa dos Motores Gráficos

A maioria dos motores gráficos baseia-se na integração de um conjunto bem definido de tecnologias, pois, geralmente, são projetados de acordo com a situação mais comum ao tipo de jogo para o qual é concebido. Assim, é difícil encontrar um motor de propósito geral que atenda às necessidades das aplicações de RV.

De maneira geral, os subsistemas de detecção de colisões oferecem testes de interseção simplificados e encontram-se fortemente acoplados ao respectivo subsistema de renderização. Dessa maneira, a detecção de colisões, ao invés de ser um subsistema independente no motor gráfico, acaba sendo um serviço adicional oferecido pelo subsistema de renderização (Bernardes Jr. et al., 2004). Além disso, com exceção de CrystalSpace, os motores analisados são incapazes de determinar interseções exatas entre modelos geométricos.

A maioria dos motores gráficos adota apenas uma API gráfica e um conjunto bem definido de técnicas para a aceleração da renderização, o que dificulta ou impossibilita a utilização de técnicas desenvolvidas pelo usuário. Além disso, a maioria desses motores disponibiliza editores de mundos integrados e suporta diretamente diversos formatos de imagem e áudio que, geralmente, são bastante populares entre os criadores de mídias. Por outro lado, a definição manual de modelos é suportada de forma precária.

O motor Unreal Engine 3 destaca-se dos demais, especialmente pela qualidade, desempenho e capacidade de configuração. Apesar disso, o código-fonte desse motor é fechado, apresenta documentação insuficiente e sua utilização em projetos comerciais exige licenciamento, além de demandar hardware de ponta e treinamento.

Uma síntese das funcionalidades dos diversos motores apresentados na Seção 4.3 é mostrada na Tabela 4.1 para facilitar a comparação entre eles.

Tabela 4.1 : Análise comparativa dos motores gráficos apresentados na Seção 4.3.

Critério	**Motor Gráfico**						
	3DSTATE	***Genesis3D***	***Quake3***	***Irllicht***	***Fly3D***	***CrystalSpace***	***Unreal Engine 3***
Portável	não	não	sim	sim	não	sim	Sim
Configurável	não	não	pouco	pouco	sim	muito	Muito
Código	fechado	GPL	GPL	Zlib	GPL	LGPL	fechado
Interface	proc.	proc.	proc.	OO	OO	OO	OO
Plugins	não	não	sim	não	sim	sim	sim
Complexidade de Uso	pequena	razoável	grande	pequena	razoável	grande	grande
Documentação	excelente	razoável	pouca	excelente	razoável	razoável	pouca
Carregamento de Mídias	fácil	razoável	difícil	fácil	razoável	razoável	difícil
Qualidade Audiovisual	razoável	boa	muito boa	boa	muito boa	boa	excelente
Desempenho	razoável	bom	excelente	bom	bom	razoável	excelente
API Gráfica	Direct3D	Direct3D	OpenGL	Direct3D OpenGL Software	OpenGL	Direct3D OpenGL Software	Direct3D OpenGL Software
Som Espacial	razoável	sim	bom	não	sim	razoável	sim
Script	não	não	sim	não	parcial	sim	sim

4.5 Considerações Finais

Atualmente, existem diversos motores gráficos disponíveis para o desenvolvimento de jogos e sistemas de RV em plataformas de *Desktop*. Alguns desses motores são mantidos através de projetos de código aberto, e podem ser utilizados livremente tanto em projetos de pesquisa quanto em aplicações comerciais. Apesar disso, os motores mais maduros e confiáveis são, geralmente, comerciais.

Na maioria das vezes, os motores comerciais podem ser utilizados gratuitamente em projetos de pesquisa, desde que respeitadas as uma série de restrições sobre a distribuição das aplicações desenvolvidas. Uma dessas restrições, exigidas na licença, obriga a publicação do código-fonte da aplicação criada com o motor. Isso constitui-se em um tremendo obstáculo quando se deseja confeccionar um produto comercial. Nesse caso, tornar-se-ia mais conveniente licenciar o motor adotado, o que, dependendo do motor e do orçamento disponível para o projeto, torna-se uma alternativa inviável.

A maioria dos motores gráficos apresenta limitações que impedem a sua plena utilização em projetos de RV e em pesquisas. Algumas dessas limitações são: código-fonte fechado, baixa portabilidade, forte acoplamento a plataformas e tecnologias, capacidades limitadas de configuração, complexidade de utilização, falta de subsistemas básicos, documentação superficial e licença extremamente cara.

Além disso, os motores não abordam o problema de detecção de colisões de maneira apropriada, o que dificulta a sua utilização em simulações físicas realistas. Os motores também não são projetados para a utilização em sistemas de propósito geral, pois geralmente destinam-se à utilização em situações pré-determinadas. Isso significa que praticamente todos os motores gráficos não suportam a adaptação dos módulos às exigências específicas de um usuário, o que dificulta ou impede a utilização de um motor gráfico por pesquisadores para o desenvolvimento de novas técnicas e algoritmos.

O Capítulo 5 destina-se à apresentação da arquitetura CRAb Graphics Engine, desenvolvida especialmente para a construção de motores gráficos que contornem as limitações técnicas apresentadas anteriormente, e que, principalmente, sejam portáveis e possuam grandes capacidades de expansão, configuração e adaptação.

Capítulo 5

A Arquitetura CRAb *Graphics Engine*

5.1 Introdução

Os motores gráficos disponíveis atualmente possuem uma série de limitações que impedem sua plena utilização na construção de aplicações de Realidade Virtual. Tais limitações referem-se principalmente à utilização de dispositivos de entrada, à portabilidade, ao forte acoplamento a tecnologias e à capacidade de customização. Nas próximas subseções deste capítulo, é apresentada uma nova arquitetura para motores gráficos, denominada CRAb *Graphics Engine* (CRAbGE), projetada especialmente para superar essas limitações.

O restante deste capítulo está organizado da seguinte maneira. Na Seção 5.2, são apresentados os princípios adotados no projeto da arquitetura CRAbGE. A Seção 5.3 destina-se ao detalhamento dos módulos que compõem essa arquitetura. Na Seção 5.4, é feita uma análise da arquitetura CRAbGE à luz dos principais requisitos apresentados no Capítulo 4, sendo que uma análise mais detalhada é realizada, no próximo Capítulo, acerca da implementação dessa arquitetura. Por fim, a Seção 5.5 destina-se à apresentação das considerações finais deste capítulo.

5.2 Princípios de Projeto

O principal objetivo da arquitetura CRAbGE é dar suporte tanto à construção de motores gráficos portáveis que possuam grande capacidade de expansão, configuração e customização, quanto à construção de aplicações que se abstraiam das tecnologias utilizadas para implementar as funcionalidades do motor gráfico.

Nessa arquitetura, o motor gráfico é definido através de um conjunto de módulos fracamente acoplados entre si os quais encapsulam grupos de funcionalidades. Dessa maneira, cada módulo torna-se independente dos demais quanto às tecnologias utilizadas e aos aspectos de configuração, customização e expansão. A estruturação desses módulos é baseada no modelo *microkernel* (Tanembaum, 1999), isto é, existe um módulo central que

gerencia uma série de módulos secundários intercambiáveis e independentes. A adoção do modelo *microkernel* em um sistema contribui para sua portabilidade, bastando que se construa um módulo central portável, e que os módulos secundários do sistema sejam implementados adequadamente nas diversas plataformas de interesse. Isso é ilustrado pela Figura 5.1.

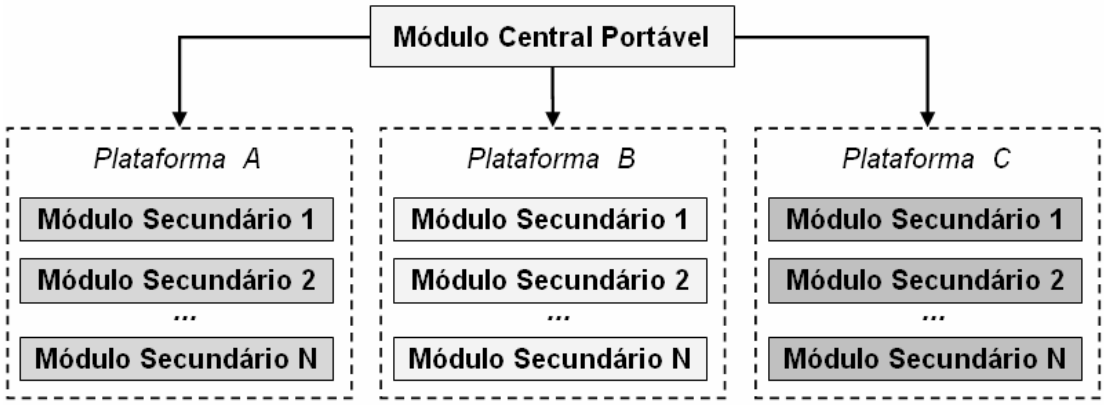


Figura 5.1: Portabilidade através de reimplementarão de módulos secundários.

Dessa maneira, a portabilidade de um sistema *microkernel* caracteriza-se pela implementação de cada módulo em diversas plataformas, o que demanda maior esforço de desenvolvimento. Entretanto, essas implementações podem ser realizadas individualmente por módulo e plataforma, simplificando os processos de desenvolvimento e manutenção. Para reduzir o esforço de desenvolvimento necessário à implementação dos módulos secundários presentes na arquitetura CRAbGE, em várias plataformas, recomenda-se fortemente a adoção de tecnologias portáveis e de domínio público.

Além de contribuir para a portabilidade do motor, o modelo *microkernel* é econômico em termos de memória, pois o sistema tem a liberdade de carregar apenas os módulos necessários a seu funcionamento. Esse modelo também permite a utilização de aplicações construídas com o motor para testar e comparar as diversas tecnologias externas que podem ser integradas a ele. Assim, por exemplo, é possível utilizar o módulo de renderização do motor para estabelecer uma comparação entre duas APIs gráficas no contexto de uma aplicação.

De uma maneira geral, os módulos da arquitetura CRAbGE são especificados com base nas principais funcionalidades presentes em tecnologias gratuitas, portáveis e de qualidade. CRAbGE prevê a expansão do motor, que pode ser realizada através de novas

implementações dos módulos secundários pré-existentes ou ainda pela definição de novos serviços através de módulos adicionais. Esses módulos secundários são construídos sobre um núcleo portável e podem ser registrados nesse núcleo em tempo de execução através de *plugins*. Entretanto, isso não impede que tais módulos sejam registrados em tempo de compilação, o que pode ser feito através dos mesmos mecanismos utilizados pelos *plugins*.

Com o intuito de facilitar a compreensão da estruturação de um grupo de entidades e de como essas entidades cooperam para o funcionamento de um módulo, a especificação dos módulos que compõem a arquitetura CRAbGE utiliza, quando possível, uma abordagem baseada em padrões de projeto (Gamma et al., 2002).

5.3 Componentes da Arquitetura

A arquitetura CRAbGE foi projetada de acordo com o princípio de encapsulamento do paradigma de orientação a objetos (OO), de maneira que cada módulo disponibilizado pelo motor fosse definido por meio de uma interface abstrata. Essa interface é única e, portanto, independente da implementação do módulo ou da tecnologia associada ao mesmo. Essa forma de padronizar o acesso às funcionalidades de cada serviço, oferecendo ao usuário a conveniência de conhecer e usar apenas as interfaces padrões, torna o sistema que usa o motor independente de tecnologias específicas (Maia et al., 2003). Apesar de CRAbGE utilizar analogias com o paradigma de orientação a objetos e recomendar que as implementações utilizem uma linguagem OO, as implementações dessa arquitetura podem utilizar, quando necessário, uma linguagem procedimental.

Em função das interfaces abstratas, a arquitetura CRAbGE realiza uma divisão do motor gráfico em quatro camadas:

- **Motor Gráfico**, que é responsável pela gerência transparente dos serviços, em alto nível;
- **Interface de Serviços**, que define as funcionalidades de cada módulo através de entidades abstratas, além de especificar como essas entidades são acessadas;
- **Implementação de Serviços**, que efetivamente implementa os módulos definidos pela camada de Interface de Serviços através da especialização das respectivas entidades abstratas; e

- **Tecnologias Externas**, que fornece soluções, geralmente em baixo nível, adequadas à implementação das funcionalidades definidas em cada módulo do motor.

Para cada tipo de serviço, sua respectiva interface define um pacote contendo entidades abstratas e concretas que compõem o modelo de subsistema adotado pelo motor gráfico. Em um determinado serviço do motor, as respectivas entidades encapsulam os aspectos funcionais comuns a qualquer implementação adequada desse serviço em seu contexto específico. Por exemplo, para o serviço de sistema de arquivos virtuais, pode-se usar qualquer implementação que possibilite usar um arquivo virtual para ler e escrever dados.

A interface de um serviço é também responsável por centralizar a instanciação das entidades correspondentes a cada implementação do mesmo, de forma que esse serviço mantenha apenas um conjunto coerente de instâncias. Assim, através da definição de uma "fábrica de entidades" baseada no padrão *Abstract Factory* (Gamma et al., 2002), impede-se a mistura de instâncias de implementações distintas que, muito provavelmente, não seriam compatíveis.

Essa estruturação do motor possibilita que qualquer tecnologia específica possa ser utilizada para prover um serviço, desde que sua respectiva interface padrão seja estendida e efetivamente implementada segundo as convenções dessa interface. Uma vez implementada a extensão, deve-se implementar também sua fábrica de entidades, que é registrada no motor em tempo de compilação ou de execução, usando o nome do serviço que fornece e o nome que identifica a tecnologia utilizada em sua implementação. Dessa maneira, o motor gráfico centraliza a gerência dos serviços e encapsula a interação do programador com as implementações, sendo que esse último precisa apenas saber o nome da tecnologia que deseja utilizar em sua aplicação, ou utilizar um conjunto de tecnologias padrões distribuídas juntamente com o motor.

A arquitetura CRAbGE define vários níveis de abstração. Em um alto nível de abstração, a aplicação de RV enxerga os serviços que o motor provê, abstraindo-se de conceitos relativos à implementação de modelos geométricos e fontes sonoras, por exemplo. Em um baixo nível de abstração, o desenvolvedor que se dispuser a implementar uma nova estratégia de renderização enxergará detalhes de baixo nível tais como: os modelos geométricos, as primitivas que os compõem, as estruturas de dados que os

armazenam e, muito provavelmente, as equações e testes necessários para se determinar a visibilidade de porções da cena. Em um outro nível de abstração, o programador de uma nova implementação de determinado serviço deve conhecer os detalhes operacionais da API que utiliza.

No restante desta seção, são apresentados, de maneira detalhada, os três módulos principais da arquitetura CRAbGE: núcleo, subsistemas e carregadores de elementos do mundo virtual. São também detalhados os principais aspectos relativos à integração desses módulos em uma camada de alto nível utilizada para gerenciar cenas tridimensionais.

5.3.1 Núcleo

O núcleo do motor gráfico é o módulo central responsável por registrar e gerenciar os módulos secundários do motor. Além disso, o núcleo é responsável por fornecer módulos básicos que podem ser utilizados tanto pela camada de aplicação quanto pelos módulos secundários do motor gráfico.

Na arquitetura CRAbGE, o núcleo é composto pelos seguintes módulos básicos: Relógio do sistema, *Log* do Motor (registro de atividades e erros), Matemática vetorial, Gerência de *plugins*, Sistema de arquivos virtuais, Gerência de recursos, Gerência de dispositivos de entrada e Gerência de subsistemas. Esses módulos são ilustrados na Figura 5.2.

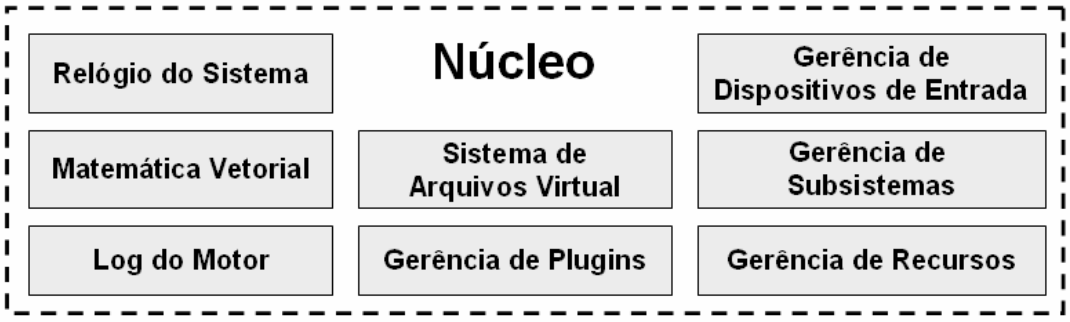


Figura 5.2: Os módulos básicos que compõem o núcleo do motor gráfico.

Relógio do Sistema. Este módulo implementa um relógio utilizado para temporizar eventos e sincronizar o funcionamento dos módulos no motor gráfico de acordo com o *clock* do sistema, sendo composto apenas pela entidade *Timer*, que possui operações para pausar, parar e reiniciar a contagem do tempo. Essa entidade adota o segundo como unidade de medida, descrevendo o tempo decorrido através de números de ponto flutuante, pois a

precisão do relógio pode variar de acordo com o sistema operacional e hardware presentes na plataforma de execução.

Log do Motor. O módulo de *Log* do Motor é responsável por manter, em tempo de execução, um arquivo de *log* confiável e legível descrevendo a execução das principais tarefas realizadas pelo motor gráfico e pela camada de aplicação. Esse módulo é composto pelas entidades abstratas *Logger* e *LoggerFactory*. *Logger* é responsável pela criação de arquivos de *log* e possui operações para adicionar mensagens com diferentes níveis de confiabilidade. Essas mensagens podem ser de vários tipos, como, por exemplo, erros, advertências ou mensagens genéricas. *LoggerFactory* representa uma fábrica responsável pela criação de instâncias de *Logger* de acordo com a formatação desejada, permitindo a criação de arquivos de texto simples ou de páginas HTML, por exemplo**.**

Matemática Vetorial. Este módulo encapsula várias operações de álgebra linear e de geometria freqüentemente utilizadas na construção de aplicações de computação gráfica. Essas operações são modeladas através das entidades *Vector2D*, *Vector3D*, *Vector4D*, *Matrix3x3*, *Matrix4x4, Plane*, *Quaternion* e *Frustum*.

Gerência de *Plugins*. Este módulo é responsável por prover uma interface para o carregamento, em tempo execução, de bibliotecas de vínculo dinâmico que são utilizadas no motor gráfico como *plugins*. Esse módulo é composto pelas entidades concretas *DynamicLibrary*, *Plugin* e *PluginManager*.

DynamicLibrary encapsula uma biblioteca de vínculo dinâmico, provendo operações para seu carregamento e finalização; e possui operações para acessar, através de nomes, funções e variáveis contidas nessa biblioteca.

PluginManager é uma entidade *singleton* (Gamma et al, 2000) responsável pelo gerenciamento de *plugins*, oferecendo operações para o carregamento e finalização dos mesmos.

A entidade *Plugin* é construída sobre *DynamicLibrary*, especificando dois nomes de funções que são utilizados para recuperar as funções de inicialização e finalização do *plugin* a partir da *DynamicLibrary*. Essas funções de inicialização e finalização são identificadas pelos nomes "*initPlugin*" e "*deinitPlugin*", respectivamente. O carregamento de um *plugin* acontece da seguinte maneira. A biblioteca correspondente ao *plugin* é

carregada na forma de uma *DynamicLibrary* que é utilizada para recuperar a função de inicialização do *plugin*, que é invocada para que novos módulos sejam registrados no motor gráfico. Analogamente, durante a finalização do *plugin*, a função de finalização é invocada antes que a biblioteca correspondente seja descarregada, permitindo que o *plugin* remova os módulos que registrou no motor gráfico e libere os recursos que alocou.

Sistema de Arquivos Virtuais. Este módulo é responsável pela abstração de localização, leitura e escrita dos arquivos utilizados pelo motor gráfico. Ele define uma interface para leitura e escrita de dados em arquivos virtuais, que provêem o encapsulamento de aspectos como compactação e comunicação através de uma rede, por exemplo. Isso permite, entre outras coisas, que o desenvolvedor crie uma rotina de alto nível capaz de carregar modelos tridimensionais, transparentemente, a partir de um arquivo compactado ou de um arquivo no diretório do sistema.

Na arquitetura CRAbGE, o sistema de arquivos virtuais é composto pelas entidades abstratas *InputStream*, *OuputStream*, *Directory* e *StreamFactory*, e pela entidade concreta *StreamFactoryEnumerator*. As entidades *InputStream* e *OutputStream* representam fluxos de entrada e saída, respectivamente, que são usados para realizar operações de entrada e saída de dados a partir de um arquivo virtual. *Directory* encapsula o conceito de diretório em um sistema de arquivos, sendo responsável pela localização de arquivos e pela criação de fluxos de dados para a leitura e escrita de arquivos em um diretório virtual. *StreamFactory* representa uma fábrica abstrata para a criação de instâncias de diretórios virtuais, que é registrada em uma instância *singleton* de *StreamFactoryEnumerator* através dos tipos de diretórios que é capaz de ler e escrever.

Gerência de Recursos. Este módulo estabelece as políticas utilizadas para localizar, carregar, acessar e liberar os recursos utilizados pelo motor gráfico, como, por exemplo, texturas e modelos geométricos pré-moldados. O módulo de gerência de recursos é composto pela entidade concreta *singleton ResourceDirectory* e pelas entidades abstratas *Resource*, *ResourceLoader* e *ResourceManager*. *ResourceDirectory* é a entidade principal responsável por centralizar o carregamento e o acesso aos recursos do motor, que são encapsulados por *Resource* e carregados a partir de uma lista de diretórios virtuais que são especificados pelo usuário. Essa entidade principal possui as seguintes operações:

- Registro e remoção de diretórios virtuais contendo recursos a carregar;

- Carregamento, recuperação e finalização de recursos;
- Registro e remoção de carregadores de recursos;
- Registro e remoção de gerenciadores de recursos; e
- Criação manual de recursos, suprindo o suporte à utilização de dados obtidos pelo usuário através de computação ou simulações.

As entidades *ResourceLoader* e *ResourceManager* encapsulam, respectivamente, o carregamento de recursos a partir de arquivos e o gerenciamento de recursos já carregados na memória. Essas entidades são registradas em *ResourceDirectory* através de uma lista de sufixos que é utilizada para identificar os tipos de recursos que são suportados por essas entidades.

Cada recurso no motor gráfico é identificado através de um *path* que é utilizado tanto para localizar o recurso na lista de diretórios, quanto para identificar, através do sufixo, qual o formato de arquivo que ele utiliza. O sufixo ".png" em "lareira.png", por exemplo, permite que *ResourceDirectory* identifique esse recurso como uma imagem no formato PNG.

De acordo com essa convenção, o usuário do motor utiliza um *path* do sistema para solicitar o carregamento de um recurso no motor, que é realizado pela entidade *ResourceDirectory* de acordo com os seguintes passos:

- Em primeiro lugar, essa entidade tenta localizar um gerenciador adequado, e, caso esse gerenciador não seja encontrado, *ResourceDirectory* retorna um recurso nulo e cria uma mensagem de erro no arquivo de *log*;
- Quando o gerenciador é encontrado, verifica-se se o recurso requisitado já foi carregado, caso já tenha sido carregado, esse recurso é retornado por *ResourceDirectory*;
- Caso o recurso requisitado não tiver sido carregado, *ResourceDirectory* tenta localizar um carregador adequado àquele formato de arquivo, e, caso o carregador não seja encontrado, retorna um recurso nulo e cria uma mensagem de erro no arquivo de *log*;
- Quando o carregador de recursos adequado é encontrado, o recurso é carregado, registrado no gerenciador de recursos adequado e retornado por *ResourceDirectory*.

A identificação de recursos através de nomes torna o acesso a esses recursos mais intuitivo. No entanto, a utilização de nomes muito extensos ou a existência de um grande número de recursos pode levar a uma grande demanda de memória. Assim, recomenda-se que um código de *hash* baseado em nomes seja utilizado para identificar os recursos internamente.

Embora o acesso aos recursos já carregados seja centralizado pela entidade *ResourceDirectory*, a busca por esses recursos é realizada de maneira hierárquica. Quando um recurso é requisitado, *ResourceDirectory* identifica, com base no sufixo do recurso em questão, a qual *ResourceManager* deve delegar a busca. Isso permite acelerar o processo de localização, evitando a criação de um mapa de recursos único que poderia tornar-se muito grande e, portanto, ineficiente.

Gerência de Dispositivos de Entrada. Na arquitetura CRAbGE, os dispositivos de entrada são classificados de acordo com o tipo de informação que entregam ao motor gráfico: rotações, orientações, translações, posições, entradas digitais (como botões em um *joystick*, por exemplo), entradas analógicas (como um manche, por exemplo), luvas, gestos, etc. O módulo de gerência de dispositivos de entrada permite que a camada de aplicação utilize um tipo de entrada, abstraindo-se de aspectos como *hardware* e *drivers* de dispositivos. Esse módulo define as entidades concretas *DeviceManager* e *DeviceEvent*, além das entidades abstratas *Device, DeviceEventTranslator* e *DeviceListener*. A estruturação dessas entidades é ilustrada pela Figura 5.3.

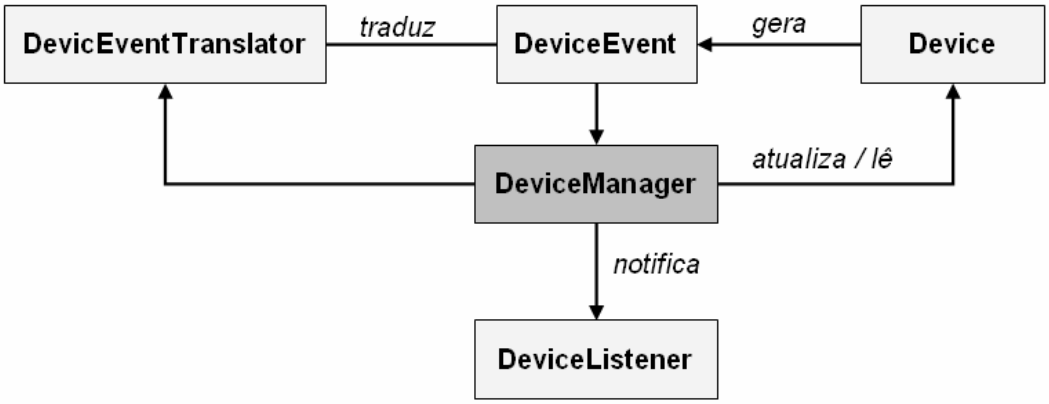


Figura 5.3: Estruturação das entidades que compõem o módulo de gerência de dispositivos de entrada.

DeviceManager encapsula o gerenciamento de dispositivos de entrada e a notificação de entidades-clientes que "escutam" eventos de entrada gerados por esses dispositivos. Essa entidade possui operações para registrar dispositivos e clientes, para criar eventos síncronos, para notificar os clientes desses eventos e para atualizar todos os dispositivos registrados.

DeviceEvent encapsula mensagens síncronas, representando mudanças imediatas nos dispositivos de entrada. Essa entidade contém uma referência para o dispositivo que causou o evento e uma cópia das informações obtidas pelo dispositivo, que são utilizadas pelos clientes. Cada entidade cliente é uma instância de *DeviceListener* que registra-se em *DeviceManager* para que seja notificada de eventos contendo certos tipos de informações, como rotação e translação, por exemplo. Essas informações podem ser utilizadas em uma especialização de *DeviceListener* para que um objeto no ambiente virtual mova-se em resposta às entradas do usuário, como os objetos-sensores adotados em Pinho (2002), por exemplo.

Por sua vez, a entidade abstrata *Device* representa um dispositivo de entrada que possui como principais atributos um identificador do tipo de informação que entrega à aplicação, o número de entradas lidas por vez e uma lista dessas entradas. Essa entidade possui operações para a inicialização, atualização e leitura assíncrona de dados desse dispositivo, além da finalização do dispositivo.

É possível que o cliente registrado em *DeviceManager* esteja esperando por um evento contendo um certo tipo de informação que não é obtido pelos dispositivos registrados naquele momento. Por exemplo, o cliente pode estar esperando por uma translação, enquanto apenas rotações são suportadas pelos dispositivos registrados. Nesse tipo de situação, especializações da entidade *DeviceEventTranslator* podem ser registradas em *DeviceManager* de maneira a traduzir um tipo de entrada noutro, de maneira a garantir a existência de entradas compatíveis com aquelas que o cliente espera. *DeviceEventTranslator* possui operações para a geração de eventos sintéticos em *DeviceManager* a partir da captura de eventos que não foram tratados por nenhum cliente, atuando como se fosse um dispositivo síncrono.

Gerência de Subsistemas. Um subsistema é um módulo secundário do motor gráfico que é responsável por resolver um domínio de problemas que identifica um serviço do motor,

como, por exemplo, renderização em tempo-real ou som espacial. O módulo de gerência de subsistemas é composto pelas entidades *Engine* e *Subsystem*. A entidade abstrata *Subsystem* representa a implementação de um subsistema do motor gráfico e possui os seguintes atributos:

- Nome do subsistema que essa entidade implementa no motor gráfico;
- Nome da tecnologia utilizada na implementação desse subsistema;
- Três números que identificam a versão desse subsistema; e
- Lista contendo os nomes dos subsistemas de que depende, permitindo que um subsistema seja especificado com base em outros subsistemas.

Subsystem possui operações para acessar esses atributos e para a inicialização e finalização do subsistema que implementa. Em uma instância de *Subsystem*, os três números que identificam a versão do subsistema devem ser atribuídos de acordo com a seguinte convenção:

- O número da esquerda identifica o conjunto de funcionalidades que devem estar presentes em uma versão final do subsistema;
- O número do meio é utilizado para determinar o percentual de funcionalidades da próxima versão final que esse subsistema implementa; e
- O número da direita é reservado para controlar as manutenções realizadas após a liberação de uma implementação desse subsistema, principalmente quando forem realizadas correções de falhas.

Um subsistema na versão 0.87.12, por exemplo, possui 87% das funcionalidades da versão 1.0.0 e já possui 12 manutenções realizadas. Já um subsistema na versão 1.6.0, por exemplo, conta com 60% das funcionalidades da versão 2.0.0.

A entidade *singleton Engine* encapsula a inicialização e a finalização dos subsistemas que compõem o motor gráfico. Essa entidade possui os seguintes atributos:

- Mapeamento entre o nome de um subsistema e uma lista contendo as implementações desse subsistema que estão disponíveis para a camada de aplicação; e
- Mapeamento entre o nome de um subsistema e sua implementação a ser utilizada pela camada de aplicação.

A entidade *Engine* possui operações para registro, seleção e remoção de implementações de subsistemas, além de operações para inicialização e finalização do motor gráfico.

Quando uma instância de *Subsystem* é registrada em *Engine*, ela é inserida na lista de implementações do subsistema que *Engine* provê. Caso essa lista esteja vazia, *Engine* assume que a instância registrada será utilizada sempre que a camada de aplicação não informar qual implementação do subsistema instanciado deseja utilizar. Assim, é possível que as implementações "OpenGL", "Direct3D" e "Software" estejam disponíveis para o subsistema "render" e que a camada de aplicação selecione a implementação que julgue mais conveniente, como ilustrado na Figura 5.4.

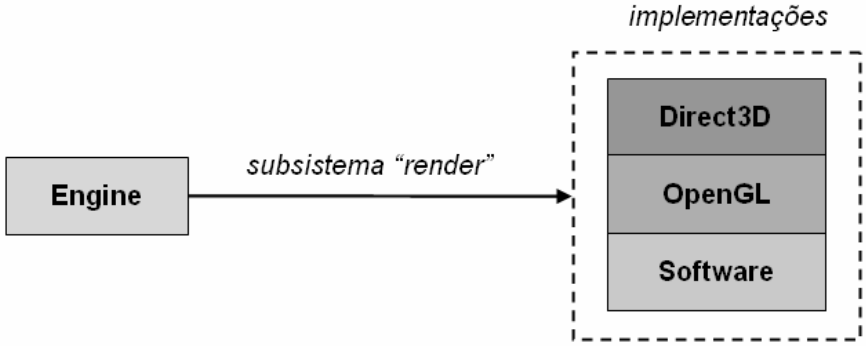


Figura 5.4: As instâncias "Direct3D", "OpenGL" e "Software" são registradas como implementações do subsistema "render".

Durante a fase de inicialização do motor gráfico, *Engine* carrega uma lista de *plugins* a partir de um diretório que é especificado pela camada de aplicação. Após isso, *Engine* tenta inferir a ordem de inicialização das instâncias de Subsystem correspondentes aos subsistemas que a camada de aplicação deseja utilizar. Isso é realizado da seguinte maneira. Em primeiro lugar, *Engine* monta um grafo de dependências entre essas instâncias a partir da lista de dependências que cada *Subsystem* possui. A partir de então, assumindo-se que esse grafo de dependências é acíclico, *Engine* utiliza a ordenação topológica desse grafo para determinar a ordem de inicialização das instâncias de *Subsystem*. Durante a fase de finalização, essa ordem é invertida para que cada instância de *Subsystem* seja finalizada corretamente. A Figura 5.5 ilustra a inicialização e finalização dos subsistemas S1, S2, S3, S4 e S5 a partir de suas listas de dependências.

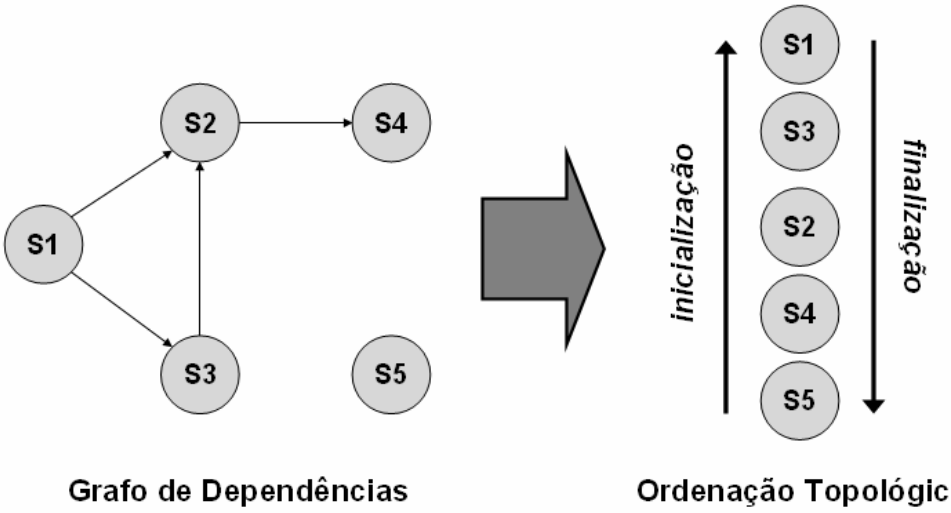


Figura 5.5: Determinação das ordens de incialização e finalização dos subsistemas S1, S2, S3, S4 e S5 a partir de um grafo de dependências: S1 depende de S2 e S3; S3 depende de S2; S2 depende de S4; S4 e S5 não possuem dependências.

5.3.2 Subsistemas Abstratos

Com o objetivo de abstrair o uso de tecnologias específicas, cada subsistema básico da arquitetura CRAbGE é projetado de acordo com as funcionalidades comumente presentes nas principais tecnologias existentes para a sua implementação. Para tanto, as principais funcionalidades de cada subsistema são especificadas através de entidades abstratas que encapsulam a utilização dessas tecnologias:

- O subsistema de renderização abstrai-se de API gráfica, da linguagem usada na *pipeline* programável, da reordenação de objetos visíveis para pintura e das técnicas utilizadas para acelerar a pintura das cenas;
- O subsistema de detecção de colisões abstrai-se das técnicas utilizadas internamente para realizar os testes de interseção contra modelos geométricos deformáveis;
- O subsistema de som espacial abstrai-se da API de áudio usada internamente para especificar as propriedades sonoras presentes nos modelos OpenAL e EAX, tomados como referência para esse subsistema por serem padrões abertos, abrangentes, portáveis e de qualidade; e
- O subsistema de scripts abstrai-se do interpretador utilizado internamente para pré-compilar e executar scripts.

Subsistema de Renderização em Tempo-Real. Na arquitetura CRAbGE, o subsistema de renderização em tempo-real independe de API gráfica e de técnicas usadas para acelerar a pintura de cenas tridimensionais. Geralmente, a pintura de uma cena tridimensional acontece em três etapas (Hofmann, 200), como ilustra a Figura 5.6.

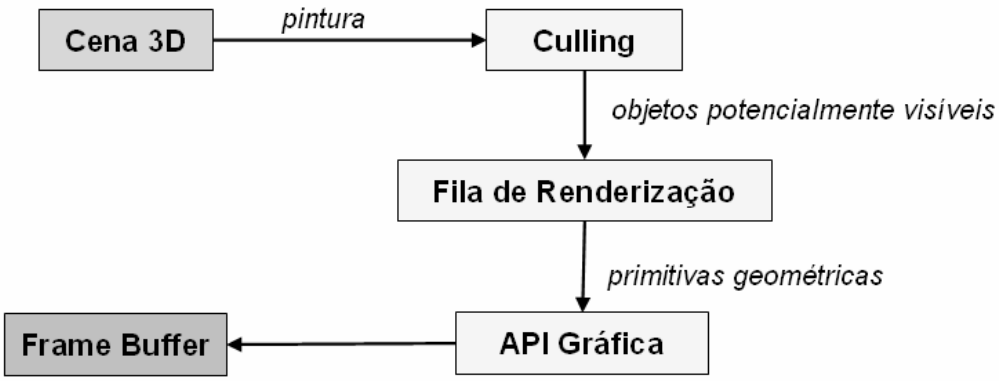


Figura 5.6: As três etapas utilizadas na renderização de uma cena tridimensional.

Na primeira etapa, utilizam-se técnicas de *culling* para evitar a pintura de objetos que não contribuem para a imagem final. Dessa maneira, os objetos potencialmente visíveis são enviados a uma fila de renderização. Na segunda etapa, essa fila é reordenada de acordo com o material que cada objeto utiliza, de maneira a reduzir o número de trocas internas de estado que são requeridas para a pintura desses objetos. Na última etapa, os comandos de uma API gráfica são usados para coordenar, em baixo nível, a pintura dos objetos presentes na fila de renderização.

A arquitetura CRAbGE independe das tecnologias e estratégias usadas em cada etapa da renderização de uma cena. Para tanto, divide o subsistema de renderização em três submódulos independentes que encapsulam cada uma dessas etapas:

- Renderização em baixo nível, responsável por encapsular as funcionalidades de uma API gráfica;
- Fila de renderização, construída sobre o módulo de renderização em baixo nível e responsável por reordenar a pintura dos objetos para reduzir as trocas de estados na API gráfica; e
- Renderização em alto nível, construída sobre os módulos de renderização em baixo nível e fila de renderização, é responsável por encapsular as técnicas usadas para acelerar a pintura de cenas tridimensionais.

Essa estruturação permite instanciar cada um desses submódulos de maneira independente, o que confere grande capacidade de configuração e expansão ao subsistema de renderização. Além disso, essa divisão em submódulos facilita os processos de desenvolvimento e manutenção do subsistema de renderização.

O submódulo de renderização em baixo nível baseia-se no modelo utilizado pelo sistema de renderização OGRE (OGRE3D, 2004). Nesse modelo, cada grupo de primitivas enviado ao *frame buffer* é representado por uma operação de renderização, encapsulada pela entidade concreta *RenderOperation*, que armazena as tabelas de vértices e índices que descrevem essas primitivas. Durante a pintura de uma cena, as operações de renderização são interpretadas pela entidade *RenderSubsystem*, uma especialização de *Subsystem* que encapsula a utilização de uma API gráfica para efetivamente enviar primitivas ao *frame buffer*. Esse processo é ilustrado pela Figura 5.7.

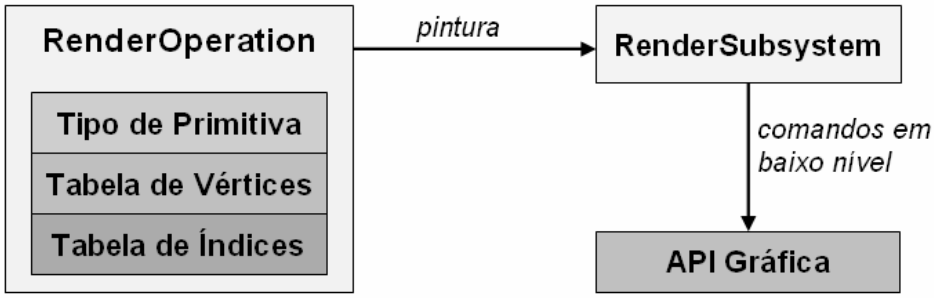


Figura 5.7: Abstração de API gráfica através da interpretação de operações de renderização.

Na arquitetura CRAbGE, o submódulo de renderização em baixo nível é composto pelas seguintes entidades:

- *HardwareBuffer*, entidade abstrata que encapsula um *buffer* residente, de preferência, na memória gráfica, ou na memória principal. Quando o buffer for residente na memória gráfica, a leitura desse buffer a partir da CPU pode comprometer o desempenho. Para contornar essa limitação, é possível criar uma instância de *HardwareBuffer* na memória principal que funciona como um buffer "sombra" sincronizado com o *buffer* original de maneira transparente;
- *VertexBuffer*, especialização abstrata de *HardwareBuffer* que é utilizada para armazenar vértices usados na descrição de primitivas;

- *IndexBuffer*, especialização abstrata de *HardwareBuffer* que é usada para armazenar índices que conectam vértices armazenados em uma instância de *VertexBuffer*, de maneira a formar primitivas;
- *HardwareBufferManager*, entidade abstrata que encapsula a criação e a destruição de instâncias de *VertexBuffer* e *IndexBuffer*;
- *DefaultVertexBuffer, DefaultIndexBuffer* e *DefaultBufferManager*, implementações padrões de *VertexBuffer, IndexBuffer* e *BufferManager*, respectivamente, usadas para a criação de *buffers* na memória principal;
- *VertexData*, entidade concreta que representa uma tabela de vértices contendo uma lista de atributos para cada vértice, como posição, normais e coordenadas de textura, que são armazenados em instâncias de *VertexBuffer*;
- *IndexData*, entidade concreta que representa uma tabela de índices armazenada em uma instância de *IndexBuffer*;
- *RenderOperation*, entidade concreta que representa um grupo de primitivas, indexadas ou não, que são armazenadas através de uma instância de *VertexData* e de uma instância opcional de *IndexData*, que é usada apenas quando as primitivas são indexadas;
- *Texture*, especialização abstrata de *Resource* que representa uma textura na API gráfica encapsulada por *RenderSubsystem*;
- *AnimatedTexture*, especialização concreta de *Texture* que representa uma seqüência de texturas usadas para descrever uma animação;
- *TextureManager*, especialização abstrata de *ResourceManager* que é responsável por gerenciar texturas;
- *TextureUnit*, entidade abstrata que encapsula o estado interno de um canal de textura no hardware gráfico, como a textura e a técnica de filtragem usada por esse canal;
- *VertexShader*, entidade abstrata que encapsula um programa do tipo *vertex shader* executado na GPU como parte da pipeline programável;
- *PixelShader*, entidade abstrata que encapsula um programa do *pixel shader* executado na GPU como parte da pipeline programável;

- *ShaderManager*, entidade abstrata responsável pela criação, ativação e destruição de instâncias de *VertexShader* e *PixelShader* que encapsulam programas escritos em uma mesma linguagem de *shading*;
- *ShaderManagerEnumerator*, entidade concreta responsável por enumerar instâncias de *ShaderManager* que correspondem às linguagens de *shading* suportadas pela API gráfica;
- *Pass*, entidade concreta que encapsula um passo de uma técnica de renderização representado através do conjunto de parâmetros da pipeline *fixed function*, uma instância de *VertexShader*, uma instância de *PixelShader* e de uma lista de camadas de textura encapsuladas por instâncias de *TextureUnit*;
- *Material*, especialização concreta de *Resource* que encapsula a utilização de efeitos visuais aplicados a um grupo de primitivas através de uma seqüência de passos encapsulada por uma lista de instâncias de *Pass*;
- *RenderTarget*, entidade abstrata que encapsula um dispositivo de saída usado para exibir as cenas renderizadas, como uma janela do sistema, por exemplo;
- *Viewport*, entidade concreta que encapsula uma *viewport* dentro de uma instância de *RenderTarget*;
- *RenderWindow*, especialização abstrata de *RenderTarget* que encapsula uma janela do sistema usada para a exibição do *frame buffer*;
- *RenderTexture*, especialização abstrata de *RenderTarget* que usa uma textura como dispositivo virtual de saída, o que é útil para implementação de diversos efeitos visuais, como, por exemplo, impostores dinamicamente gerados e reflexões; e
- *RenderSubsystem*, entidade abstrata que encapsula as funcionalidades de uma API gráfica, sendo também responsável por criar instâncias de *HardwareBufferManager*, *TextureManager* e de *ShaderManager*.

Assim, *RenderSubsystem* é a entidade central do submódulo de renderização em baixo nível. A Figura 5.8 ilustra a estruturação desse submódulo, enfatizando as principais entidades cuja instanciação é realizada a partir de uma implementação de *RenderSubsystem* registrada na entidade *singleton Engine*.

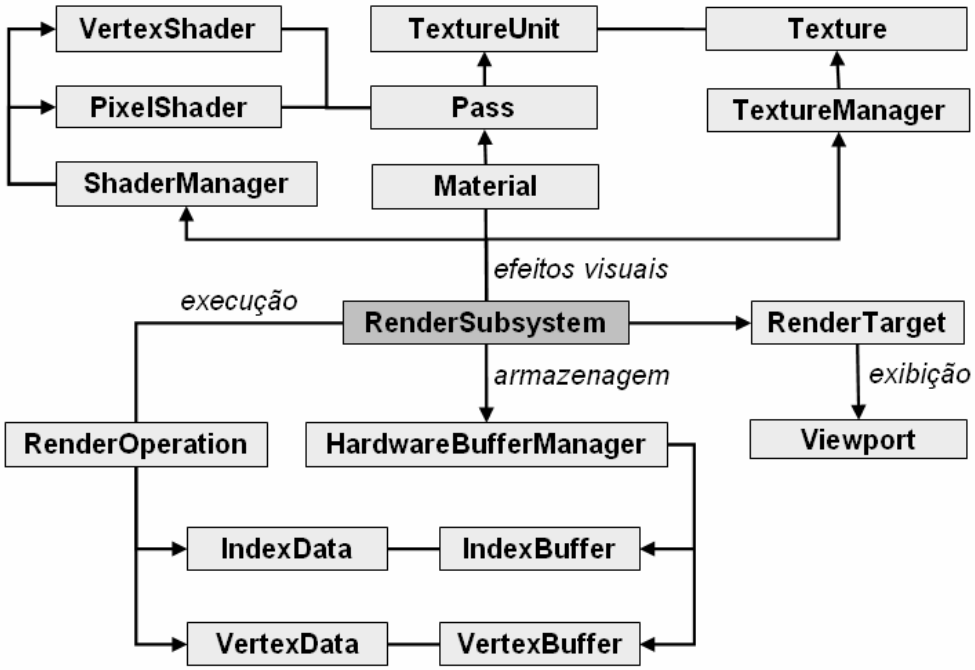


Figura 5.8: Estruturação das entidades que compõem o módulo de renderização em baixo nível.

O submódulo que representa a fila de renderização utilizada pelo motor gráfico é composto pela entidade abstrata *Renderable* e pelas entidades concretas *RenderQueue* e *RenderQueueManager*. *Renderable* define uma interface abstrata para a definição dos objetos que são pintados através da fila de renderização encapsulada pela entidade *RenderQueue*. Essa interface contém operações para acessar os seguintes atributos:

- Tipo de projeção usada pelo objeto (em perspectiva ou ortográfica);
- Matriz de transformação que descreve as primitivas do objeto em coordenadas de mundo;
- Material usado pelo objeto;
- Operações de renderização (instâncias de *RenderOperation*) usadas para enviar o objeto ao *frame buffer*;
- Quadrado da distância ao observador, usado para reordenar a pintura de objetos transparentes; e
- Prioridade requerida à pintura desse objeto.

A entidade *RenderQueue* representa uma fila de renderização com prioridades, usada para reordenar a pintura das instâncias de *Renderable* com base no material e na prioridade

atribuídos a cada uma dessas instâncias. Além disso, *RenderQueue* é responsável por utilizar a interface de *RenderSubsystem* para efetivamente enviar essas instâncias ao *frame buffer*. Instâncias de *RenderQueue* podem ser registradas na entidade concreta *RenderQueueManager*, de maneira a permitir que o motor utilize implementações customizadas pelo usuário.

Quando uma instância de *Renderable* é incluída em *RenderQueue*, essa instância é classificada, de acordo com o material que utiliza, em opaca ou transparente. Após isso, uma referência a essa instância é guardada em uma lista que corresponde à sua prioridade e ao tipo de material. Antes da renderização, as listas de objetos opacos são reordenadas de acordo com o material que esses objetos utilizam, enquanto as listas de objetos transparentes são reordenadas de acordo com o quadrado da distância desses objetos ao observador. Durante a renderização, essas listas são percorridas em ordem decrescente de prioridade, sendo que as listas de objetos opacos possuem maior prioridade do que as listas de objetos transparentes.

O submódulo de renderização em alto nível é responsável por abstrair as técnicas usadas para acelerar a pintura de objetos estáticos e dinâmicos que compõem uma cena tridimensional. Esse submódulo é composto pelas seguintes entidades:

- *Node*, especialização concreta de *Renderable* que encapsula um nó do grafo de cena usado para hierarquizar os objetos dinâmicos presentes em uma cena;
- *Mesh*, especialização concreta de *Resource* que oferece uma representação flexível para modelos geométricos;
- *SubMesh*, que encapsula uma submalha interna à entidade *Mesh* e que pode ser enviada ao *frame buffer* através de uma única instância de *RenderOperation*;
- *SceneManager*, entidade abstrata que encapsula uma cena tridimensional composta por objetos estáticos e dinâmicos, abstraindo as técnicas utilizadas para acelerar a renderização desses objetos; e
- *SceneManagerEnumerator*, entidade concreta responsável pelo registro e acesso a instâncias de *SceneManager* que são classificadas de acordo com o tipo de cena que essas instâncias representam (interior, exterior ou mista).

Ao contrário do que ocorre nos grafos de cena utilizados nas bibliotecas Java3D (Sun Microsystems, 2002) e OpenSG (OpenSG, 2004), na arquitetura CRAbGE, os nós de um

grafo de cena podem ser hierarquizados de maneira direta. Assim, não há distinção entre um nó folha ou um nó interno desse grafo, o que permite construir e modificar hierarquias de uma maneira mais flexível. Por exemplo, é possível que um nó do tipo câmera seja filho de um nó contendo um modelo geométrico e vice-versa.

A entidade *Node* encapsula um objeto atômico que é enviado ao *frame buffer* através de operações de renderização, possuindo como principais atributos uma referência ao nó-pai e uma lista de nós-filhos. *Node* possui operações para a aplicação de translações, rotações e escalas em três sistemas de coordenadas:

- O sistema local de coordenadas, adequado à movimentação de avatares e câmeras;
- O sistema do nó-pai, adequado à montagem de ramos do grafo de cena que representam objetos complexos, como, por exemplo, um carro montado a partir de nós que representam as rodas e o chassi; e
- O sistema global de coordenadas, adequado ao posicionamento dos objetos no mundo virtual.

Node também possui operações para a recuperação da posição, da orientação e da escala nesses três sistemas de coordenadas. Além disso, *Node* encapsula a determinação de visibilidade através da combinação das técnicas de *Frustum Culling Hierárquico* e *Occlusion Culling*, que utilizam uma hierarquia de OBBs construída a partir da dimensão e da orientação de cada nó presente no grafo de cena.

A entidade concreta *Mesh* encapsula uma malha tridimensional que representa um modelo geométrico através de uma lista de vértices (instância de *VertexData*) e uma lista de instâncias de *SubMesh*. Cada instância de *SubMesh* possui uma referência à *Mesh* que a contém, uma referência ao material que utiliza e uma lista de índices (instância de *IndexData*). Esses índices podem apontar para a lista de vértices da *Mesh* que contém a sub-malha ou, opcionalmente, para uma lista de vértices própria. Essa decomposição em sub-malhas permite representar modelos complexos de uma maneira bastante flexível, suportando a utilização de vários materiais em uma malha e a decomposição desses modelos em um conjunto de primitivas do tipo *triangle strip*, por exemplo. A Figura 5.9 ilustra como as primitivas podem ser formadas em uma instância de *SubMesh* a partir da indexação na lista de vértices compartilhada ou na lista local.

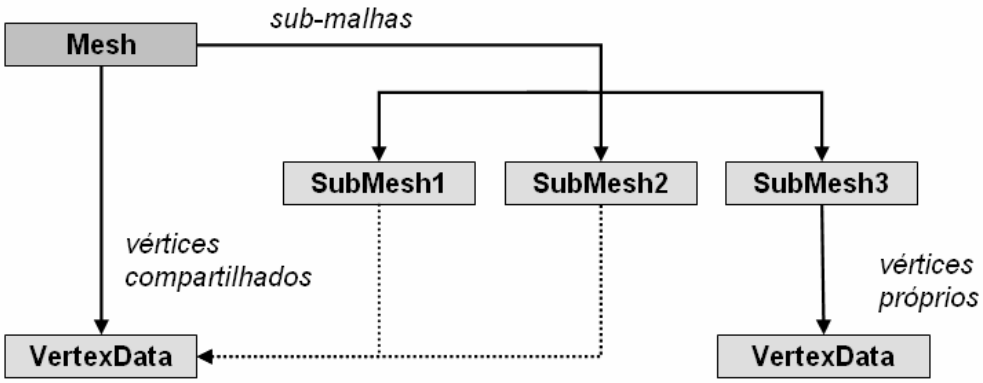


Figura 5.9: A representação interna usada pela entidade *Mesh*. Os índices em cada submalha podem apontar para os vértices compartilhados (*SubMesh*1 e *SubMesh* 2) ou para vértices próprios (SubMesh 3).

A entidade abstrata *SceneManager* representa uma cena tridimensional composta por objetos dinâmicos – representados através de grafo de cena – e por objetos estáticos que descrevem a geometria do mundo. Essa entidade abstrai as estruturas de dados usadas internamente para representar os objetos estáticos e acelerar a pintura desses objetos. Além disso, ela possui uma instância de *Node* que representa o nó-raiz do grafo de cena, devendo ser utilizada para definir a hierarquia de objetos visíveis em uma cena.

SceneManager também possui uma operação para a criação de especializações de *Node* que permite implementar estratégias de aceleração adicionais baseadas nas estruturas de dados usadas para representar os objetos estáticos internamente. Isso permite, por exemplo, combinar técnicas baseadas em estruturas de PVS e árvores BSP com as técnicas que o grafo de cena implementa para acelerar a pintura dos objetos dinâmicos, de maneira a obter um maior desempenho na renderização das cenas. Dessa maneira, essas instâncias especializadas de *Node* podem ser usadas para agrupar objetos dinâmicos para acelerar ainda mais a pintura de tipo de objetos.

Subsistema de Detecção de Colisões. Na arquitetura CRAbGE, o subsistema de detecção de colisões é construído sobre o módulo de matemática vetorial e baseia-se em testes de colisão entre objetos estacionários. Esse subsistema é composto pelas entidades concretas: *Line*, *Ray*, *AABB*, *OBB*, *Sphere*, *Capsule* e *PlaneSet*; e pelas entidades abstratas: *CollisionModel* e *CollisionSubsystem*.

As entidades *Line* e *Ray* encapsulam, respectivamente, um segmento de reta ligando um par de pontos no espaço tridimensional e uma semi-reta. *AABB* encapsula uma *bounding box* alinhada aos eixos coordenados. *OBB* representa uma *bounding box* orientada. *Sphere* representa uma esfera posicionada no espaço. *Capsule* representa uma cápsula, ou seja, o lugar geométrico definido pela varredura de uma esfera ao longo de um segmento de reta. *PlaneSet* encapsula a união de um conjunto de hiperplanos no espaço, sendo capaz de representar fechos convexos e o volume de visão definido por uma câmera, por exemplo.

Essas entidades concretas podem ser usadas em testes de interseção contra modelos deformáveis, possuindo operações para acessar e alterar os parâmetros que descrevem esses volumes convexos. Além disso, cada uma dessas entidades possui operações para o cálculo de interseção contra um volume qualquer ou contra um triângulo. Essas operações podem ser utilizadas diretamente pela camada de aplicação ou como base para a construção de algoritmos de interseção mais elaborados.

A entidade abstrata *CollisionModel* encapsula a construção e a atualização das estruturas de dados utilizadas internamente para representar um modelo geométrico deformável durante os testes de interseção. Essa entidade possui os seguintes atributos:

- Lista de vértices que formam o modelo geométrico;
- Lista de índices usados na descrição das faces do modelo; e
- Lista de normais pré-calculadas, utilizada como estrutura auxiliar na representação de modelos não-deformáveis, visto que as normais desse tipo de modelo não são modificadas.

CollisionModel possui operações para acessar essas propriedades e para atualizar as estruturas de dados quando o modelo sofre uma deformação. Além disso, essa entidade oferece operações para realizar testes de interseção contra volumes convexos (*AABB*, *OBB*, *Sphere*, *Capsule* e *PlaneSet*), segmentos de reta, semi-retas e contra outro modelo deformável. Nesses testes de interseção, é possível utilizar os seguintes parâmetros opcionais:

- Matriz de transformação descrevendo o modelo geométrico em coordenadas de mundo;
- *Flag* indicando a utilização de coerência temporal; e

- *Flag* de múltiplas interseções, indicando se o teste deve reportar apenas uma interseção quando houver múltiplas interseções.

Os testes contra semi-retas e segmentos de reta possuem um parâmetro adicional, indicando se as faces posteriores do modelo em relação ao segmento ou semi-reta são desconsideradas para o cálculo de interseções.

A entidade abstrata *CollisionSubsystem* é uma especialização de *Subsystem* que encapsula a criação e destruição das instâncias de *CollisionModel*. Essa entidade possui operações para a criação e atualização de instâncias de *CollisionModel* a partir de uma malha triangular baseada na conexão de vértices através de índices, a partir de uma instância de *RenderOperation* ou a partir de uma instância de *Mesh*.

Subsistema de Som Espacial. O subsistema de som espacial é baseado nos padrões abertos OpenAL e EAX, sendo composto pelas entidades abstratas *SoundContext*, *SourceListener*, *SoundSource*, *SoundBuffer*, *SoundBufferManager* e *SoundSubsystem*. A estruturação dessas entidades é ilustrada pela Figura 5.10.

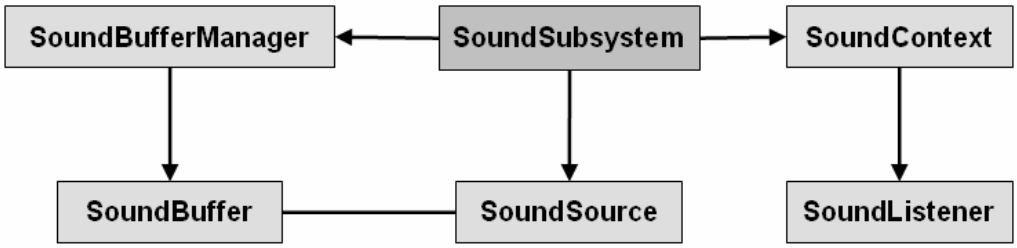


Figura 5.10: Estruturação do subsistema de som espacial na arquitetura CRAbGE.

A entidade *SoundContext* encapsula um contexto de simulação sonora, que, de maneira análoga a um contexto de renderização na API OpenGL, mantém os estados internos do ouvinte e do ambiente durante uma determinada simulação sonora. Dessa maneira, é possível que uma aplicação possua múltiplas instâncias de *SoundContext* que compartilham as fontes sonoras e amostras de som criadas no subsistema de som espacial, sendo que cada uma das instâncias de *SoundContext* corresponde a um visitante do mundo virtual. Essa entidade é instanciada a partir de *SoundSubsystem* e possui os seguintes atributos:

- Nome do *Driver* do dispositivo de hardware usado pelo contexto para mixar a saída de áudio;

- Freqüência para mixagem da saída, descrita em HZ;
- Taxa de atualização do contexto, descrita em HZ;
- Valor *Booleano* que indica se o contexto é atualizado de maneira síncrona;
- Escala utilizada para ajustar a unidade de distância usada para descrever as velocidades do ouvinte e das fontes sonoras aos cálculos de efeito Doppler;
- Fator de escala usado para exagerar ou diminuir a intensidade com que o efeito Doppler é percebido pelo ouvinte;
- Modelo de atenuação sonora, utilizado no contexto; e
- Uma instância de *SoundListener* que encapsula o ouvinte da simulação coordenada no contexto.

O *Driver* utilizado pelo contexto, a freqüência de mixagem, a taxa de atualização e a *flag* indicando um contexto síncrono são propriedades atribuídas durante a criação de uma instância de *SoundContext* por *SoundSubsystem*. *SoundContext* possui operações para acessar esses atributos, além de operações para modificar o modelo de atenuação e os atributos que interferem no cálculo de efeitos Doppler.

A entidade *SoundListener* encapsula o ouvinte em um contexto de simulação sonora e possuindo operações para acessar e modificar as suas propriedades. *SoundListener* possui os seguintes atributos:

- Uma referência à instância de *SoundContext* que criou esse ouvinte, que não pode ser modificada;
- Vetor posição no espaço global de coordenadas;
- Vetor velocidade no espaço global de coordenadas;
- Vetores foco e *up* no espaço global de coordenadas, descrevendo a orientação do ouvinte;
- Ganho, que afeta o volume dos sons percebidos pelo ouvinte; e
- Conjunto de propriedades EAX descrevendo fatores de reverberação, que são modificados apenas quando a plataforma de execução possui suporte a esse tipo de efeito sonoro.

A entidade *SoundBuffer* é uma especialização de *Resource* que representa as amostras de som que são reproduzidas através das fontes sonoras posicionadas no ambiente como instâncias de *SoundSource*. Essa entidade possui apenas uma operação, usada para

realizar o *upload* de amostras de som. A gerência das instâncias de *SoundBuffer* é delegada à entidade *SoundBufferManager*, uma especialização de *ResourceManager* que é recuperada a partir de uma instância de *SoundSubsystem*.

A entidade *SoundSource* encapsula uma fonte sonora posicionada no ambiente virtual que reproduz sons armazenados em instâncias de *SoundBuffer*, sendo instanciada através da entidade *SoundSubsystem*. *SoundSource* possui os seguintes atributos:

- Vetor posição no espaço global de coordenadas;
- Vetor velocidade no espaço global de coordenadas;
- Ganho, um fator escalar que multiplica a amplitude dos sons reproduzidos a partir dessa fonte sonora;
- Valores mínimo e máximo para o ganho efetivo, que é obtido através dos cálculos de atenuação sonora;
- Distância máxima e distância de referência, que são utilizadas no modelo de atenuação baseado no inverso da distância;
- *Rolloff*, usado para exagerar ou diminuir o efeito da atenuação sonora quando o modelo de atenuação baseado no inverso da distância é utilizado;
- Valor *Booleano* indicando se essa fonte sonora está reproduzindo sons de maneira cíclica;
- *Pitch*, que especifica um deslocamento tonal nos sons reproduzidos;
- Vetor direção no espaço global de coordenadas, indicando a direção que o som é emitido por essa fonte sonora;
- Ângulos mínimo e máximo (em graus), que, respectivamente, definem o cone interno e o cone externo, usados para o cálculo de atenuação baseado na direção de emissão do som em relação ao ouvinte;
- Fator multiplicativo do ganho fora do cone externo, usado no cálculo de atenuação baseado na direção de emissão do som em relação ao ouvinte; e
- Conjunto de propriedades EAX, descrevendo efeitos, como obstrução e oclusão, que são aplicados apenas quando forem suportados pela plataforma de execução.

SoundSource possui operações para acesso e modificação desses atributos, bem como para iniciar, pausar e suspender a reprodução de sons. Além disso, possui operações para enfileirar as amostras de áudio que serão reproduzidas, em seqüência, através dessa fonte

sonora. Isso permite decompor amostras extensas em um conjunto de amostras menores e, além disso, implementar mecanismos de *streaming* para áudio espacial em tempo-real.

SoundSubsystem é a entidade principal do subsistema de som espacial, encapsulando a criação de contextos de áudio, da instância de *SoundBufferManager* usada pelo subsistema e das fontes sonoras posicionadas no ambiente.

Subsistema de Script. Na arquitetura CRAbGE, o subsistema de scripts é projetado com base nas funcionalidades presentes nas linguagens de script mais utilizadas atualmente: Lua, Python e Ruby (Ierusalimschy et al., 2003; Python Software Foundation, 2004; Thomas et al., 2004). Esse subsistema é composto pelas entidades abstratas *ScriptSubsystem*, *Script* e *ScriptVariable*, cuja estruturação é ilustrada pela Figura 5.11.

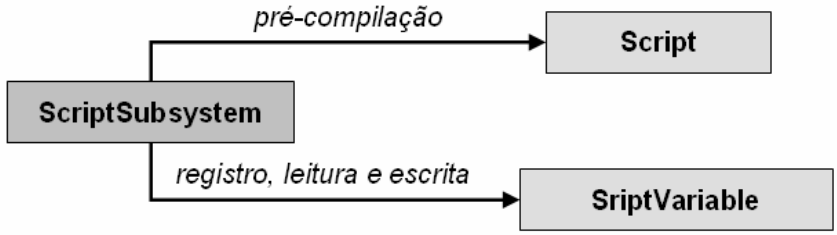


Figura 5.11: Entidades que compõem o subsistema de scripts na arquitetura CRAbGE.

A entidade *Script* encapsula um script armazenado em memória em um formato pré-compilado, permitindo executar scripts muito utilizados de uma maneira mais eficiente. Essa entidade possui apenas uma operação, utilizada para executar o script que representa. *ScriptSubsystem* encapsula um interpretador de scripts que possui as seguintes operações:

- Execução de arquivos de script;
- Execução de código armazenado na memória como uma cadeia de caracteres;
- Criação de instâncias de *Script* a partir de arquivos ou cadeias de caracteres;
- Acionamento do mecanismo de coleta de lixo, caso seja suportado pelo interpretador; e
- Registro e remoção de variáveis globais no ambiente de execução, que são representadas por instâncias de *ScriptVariable*;

A entidade *ScriptVariable* encapsula uma variável global no interpretador de scripts e possui operações para a leitura e escrita do valor armazenado nessa variável. Para facilitar a integração de scripts com os demais módulos do motor gráfico e com a camada de

aplicação, recomenda-se a realização do *binding* das principais funções e classes do motor gráfico para a linguagem utilizada pelo interpretador.

5.3.3 Gerência de Cenas

Na arquitetura CRAbGE, o módulo de gerência de cenas é composto por especializações de *Node* que são construídas sobre os demais módulos do motor gráfico. Esses nós encapsulam a utilização dos subsistemas básicos presentes no motor, que compõem uma camada de alto nível adequada à representação dos objetos dinâmicos presentes em uma cena tridimensional. O módulo de gerência de cenas é composto pelas entidades concretas: *SceneNode*, *Entity*, *BillboardSet*, *ParticleSystem*, *LodNode*, *CameraNode*, *ListenerNode* e *SoundNode* (especializações de *Node*); e pelas entidades: *Animation*, *AnimationSequence*, *AnimationState* e *AnimationManager* – abstrações da utilização de técnicas de animação.

SceneNode especializa a entidade *Node*, de maneira a associar uma instância de *CollisionModel* a um nó da cena para permitir a realização de testes de interseção contra esse nó. Além disso, essa entidade também permite usar um script pré-compilado (instância de *Script*) para atribuir comportamentos a esse nó da cena.

Entity é uma especialização de *SceneNode* que utiliza uma instância de *Mesh* para representar modelos geométricos complexos no mundo virtual. *BillboardSet* também especializa *SceneNode*, e utiliza a técnica de *billboards* para representar objetos no mundo virtual. *ParticleSystem* é uma especialização de *BillboardSet* que utiliza um sistema de partículas para representar objetos complexos (por exemplo, fogo, chuva e explosões) em um mundo virtual.

LodNode é uma especialização de *SceneNode* que encapsula a técnica de aceleração da renderização baseada em LOD estático, onde cada nível de detalhe é representado por uma instância de *Node*. Assim, *LodNode* não se limita apenas a alternar a resolução utilizada para exibir um malha tridimensional, permitindo definir níveis de detalhe de uma maneira flexível. Isso porque essa entidade suporta a definição de representações totalmente diferentes para um mesmo objeto, que são implementadas em instâncias de *Node*.

A entidade *CameraNode* é uma especialização de *SceneNode* que encapsula o modelo de câmera virtual utilizado por uma API gráfica, o que permite posicionar câmeras

em uma cena tridimensional através do grafo de cena. A entidade *ListenerNode* é uma especialização de *SceneNode* que encapsula um ouvinte de uma simulação sonora representado internamente por uma instância de *SoundListener*. *ListenerNode* é responsável por atualizar automaticamente a posição, a orientação e a velocidade de um ouvinte presente no mundo virtual. Isso permite que o usuário do motor apenas crie uma instância de *ListenerNode* e a insira no grafo de cena para que o ouvinte seja atualizado automaticamente. Analogamente, *SoundNode* especializa *SceneNode* de maneira a encapsular a atualização automática da posição, direção e velocidade de uma fonte sonora presente no ambiente virtual.

Apesar das entidades *CameraNode*, *SoundNode* e *ListenerNode* atuarem como objetos invisíveis no mundo virtual, essas entidades podem ser enviadas ao *frame buffer* e, além disso, podem possuir uma instância de *CollisionModel* associada. Assim, é possível visualizar tais entidades e realizar testes de interseção contra as mesmas, o que é bastante útil para o posicionamento dessas entidades através de editores de mundos virtuais e durante as etapas de depuração de uma cena.

A entidade abstrata *Animation* representa uma animação que é associada a instâncias de *Mesh* e é composta por um conjunto de seqüências de animação representadas por instâncias de *AnimationSequence*. Essa entidade é responsável por modificar malhas tridimensionais de acordo com uma técnica de animação qualquer, como, por exemplo, interpolação linear de quadros-chaves. Uma mesma instância de *Animation* pode ser compartilhada por diversas malhas, de maneira que o status da animação em cada malha-cliente é encapsulado pela entidade abstrata *AnimationState*. A estruturação dessas entidades é ilustrada pela Figura 5.12.

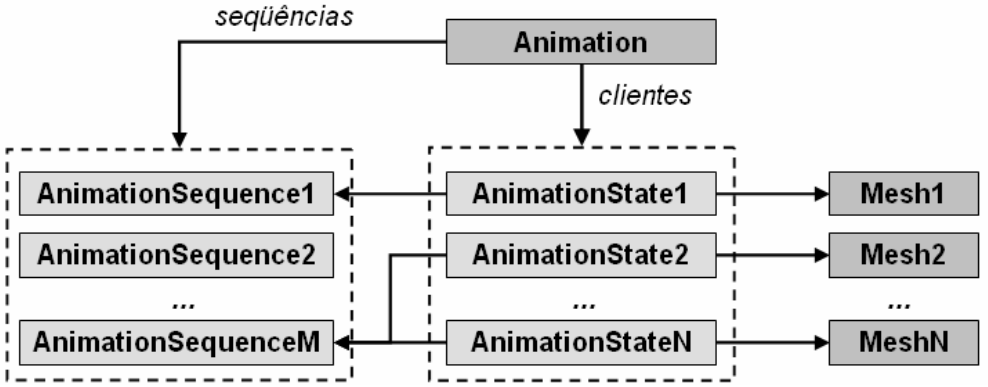


Figura 5.12: Estruturação das entidades que encapsulam técnicas de animação na arquitetura CRAbGE.

A entidade *Animation* possui uma lista de seqüências que compõem a animação, representadas por instâncias de *AnimationSequence*. Cada uma dessas seqüências representa uma animação utilizada no contexto de uma aplicação de RV, como, por exemplo, um avatar caminhando ou falando. Além disso, *Animation* possui uma lista de malhas-clientes referenciadas como instâncias de *AnimationState*. *Animation* possui operações para registrar e remover malhas clientes da animação que representa e para modificar essas malhas de acordo com a técnica de animação que utiliza internamente.

A entidade abstrata *AnimationSequence* possui como atributos o nome da seqüência de animação que representa e a duração, em segundos, dessa seqüência de animação. Além disso, as implementações de *AnimationSequence* devem possuir informações específicas sobre essa seqüência de animação, como, por exemplo, uma lista de quadros-chave, que são compartilhadas pelas malhas-clientes. *AnimationState* possui os seguintes atributos:

- Referência à malha-cliente (instância de *Mesh*);
- Seqüência utilizada para realizar animações nessa malha;
- Velocidade com que essa seqüência é reproduzida nessa malha;
- Duração, em segundos, da transição entre duas seqüências de animação; e
- Tempo, em segundos, transcorrido na animação através de uma seqüência ou durante a transição entre duas seqüências.

A entidade *singleton AnimationManager* gerencia o registro e destruição de instâncias de *Animation*, e a atualização de todas as malhas-clientes.

5.4 Análise da Arquitetura CRAbGE

A arquitetura CRAbGE é baseada no modelo *microkernel*, definindo suas principais funcionalidades através de módulos secundários que são registrados no motor gráfico através de *plugins*, o que contribui para a portabilidade do motor e lhe confere grande capacidade de configuração e expansão. Cada módulo presente no motor gráfico é responsável por prover uma interface de programação em alto nível, de modo que o usuário do motor abstraia-se de aspectos específicos, como, por exemplo, a implementação de estruturas de dados e a utilização de uma API em baixo nível.

Através desse princípio, a arquitetura CRAbGE é composta por quatro camadas, prevendo a atuação de desenvolvedores em três áreas principais: a construção de aplicações com o motor gráfico; a implementação de *plugins*; e o desenvolvimento de novas tecnologias adequadas à utilização pelo motor. Além disso, essa arquitetura provê uma camada de alto nível para criação e manipulação de cenas tridimensionais que integra as principais funcionalidades de cada subsistema básico, simplificando o processo de construção de aplicações de RV com o motor gráfico.

Na arquitetura CRAbGE, o núcleo do motor oferece uma série de módulos básicos à criação de subsistemas e de aplicações de RV. O núcleo possui um módulo de gerenciamento de recursos que prevê o reconhecimento de novos formatos de arquivos, o que facilita o carregamento de mídias no motor gráfico. Além disso, o motor possui um mecanismo simples e expansível para a utilização de dispositivos de entrada não-convencionais, típicos de aplicações de RV. O núcleo do motor também automatiza os processos de inicialização e finalização dos subsistemas através de um mecanismo capaz de identificar dependências entre esses subsistemas, estabelecendo uma ordem de inicialização e finalização, e, além disso, prevê a inclusão de novos subsistemas no motor gráfico.

O subsistema de renderização em tempo-real se abstrai da API gráfica utilizada internamente, da linguagem usada na pipeline programável e do processo de reordenação da pintura dos objetos que são enviados ao *frame buffer*. Esse subsistema provê suporte à definição de efeitos que conferem uma excelente qualidade visual às cenas renderizadas, pois permite usar os recursos da pipeline gráfica *fixed function* e da pipeline programável para a pintura de objetos em múltiplos passos.

Esse subsistema também oferece diversos mecanismos para a obtenção de uma elevada taxa de quadros por segundo durante a renderização de uma cena, ou seja:

- Oferece um mecanismo bastante flexível para a representação de modelos geométricos, suportando o pré-processamento desses modelos para que sejam pintados de maneira mais eficiente;
- Suporta, quando possível, o armazenamento de primitivas na memória de alto desempenho do hardware gráfico;
- Utiliza um grafo de cena para acelerar a renderização de objetos dinâmicos, abstraindo a representação gráfica e as técnicas utilizadas pontualmente para acelerar a renderização desses objetos; e
- Abstrai-se do conjunto de técnicas utilizado para acelerar a pintura dos objetos estáticos e prevê a integração dessas técnicas com a estrutura do grafo de cena.

Embora a taxa de quadros por segundo esteja fortemente atrelada a diversos aspectos de uma determinada aplicação, como o tipo de cenário utilizado e o nível de complexidade do mundo virtual, esses mecanismos providos pela arquitetura CRAbGE conferem grande flexibilidade ao subsistema de renderização, permitindo que ele se adapte às necessidades de uma aplicação específica.

O subsistema de detecção de colisões tanto se abstrai das técnicas que são usadas internamente para implementar os testes de interseção entre modelos geométricos deformáveis quanto se abstrai das estruturas de dados utilizadas internamente para representar esses modelos. Ele também oferece outros tipos de testes, como *picking* e interseção entre volumes convexos, além de testes entre volumes convexos e um modelo geométrico deformável.

Além disso, esse subsistema provê um mecanismo flexível para a construção e atualização das estruturas de dados que representam, internamente, os modelos geométricos envolvidos nos testes de interseção. A construção e a atualização dessas estruturas de dados são realizadas através de uma interface de alto nível que suporta a utilização de listas de faces, operações de renderização e de malhas tridimensionais.

O subsistema de som espacial é independente de APIs de áudio de baixo nível, especificando um modelo sonoro abrangente baseado nos padrões abertos OpenAL e EAX. Esse subsistema suporta a simulação de fenômenos sonoros no ambiente virtual com um

elevado grau de realismo, sendo que a camada de aplicação possui controle granular sobre os efeitos de espacialização e reverberação.

O subsistema de script se abstrai do interpretador utilizado internamente para executar e pré-compilar scripts a partir de arquivos ou trechos de código residentes na memória principal. Além disso, prevê o registro de variáveis no ambiente do interpretador, permitindo que o valor dessas variáveis seja lido e alterado pela camada de aplicação para realizar configurações de maneira bastante flexível.

5.5 Considerações Finais

A grande maioria dos motores gráficos disponíveis atualmente possui uma série de limitações que dificultam ou impedem sua utilização na construção de aplicações de Realidade Virtual, principalmente no que diz respeito à utilização de dispositivos de entrada não-convencionais, portabilidade, forte acoplamento a tecnologias e capacidade de customização.

Neste capítulo, foi apresentada a arquitetura CRAbGE que contorna essas limitações através da adoção do modelo *microkernel* para especificar os módulos do motor gráfico, que é dividido em quatro camadas: Motor Gráfico, Subsistemas Abstratos, Subsistemas concretos e Tecnologias Externas. Essa arquitetura especifica mecanismos para a construção de motores gráficos portáveis com grande capacidade de configuração, expansão e de customização.

Os módulos secundários do motor gráfico são independentes de tecnologias específicas, prevendo a realização de customizações por parte do desenvolvedor, que pode atuar na construção de aplicações, na implementação de novos *plugins* ou no desenvolvimento de tecnologias para utilização no motor gráfico.

Nessa arquitetura, cada subsistema básico é projetado de acordo com as funcionalidades comumente encontradas nas tecnologias adequadas à sua implementação, apresentando uma solução genérica, porém abrangente, para o domínio de problemas a que se destina. No próximo capítulo, são apresentados os principais aspectos da implementação da arquitetura, que constituem um estudo de caso.

Capítulo 6

Estudo de Caso

6.1 Introdução

No Capítulo 5 foi apresentada a arquitetura CRAbGE, projetada para contornar as limitações que a maioria dos motores gráficos apresentam, quanto ao desenvolvimento de jogos e de aplicações de Realidade Virtual. Essa arquitetura baseia-se no modelo *microkernel* para especificar um motor gráfico portável, configurável, customizável e expansível. Além disso, essa arquitetura oferece mecanismos para o encapsulamento de tecnologias utilizadas na implementação dos subsistemas do motor gráfico, permitindo que a camada de aplicação empregue essas tecnologias de maneira transparente através de uma interface de alto nível.

Neste capítulo, são apresentados os principais aspectos de implementação do motor CGE (acrônimo, relativo ao motor, para *CRAb Graphics Engine*), desenvolvido como um estudo de caso da arquitetura CRAbGE. O restante do capítulo está estruturado da seguinte maneira. A Seção 6.2 contém uma descrição dos principais aspectos de pré-projeto que influenciaram o processo de desenvolvimento do estudo de caso. Na Seção 6.3, são descritos os principais aspectos da implementação do motor CGE, apresentando detalhes sobre o desenvolvimento de cada um de seus módulos. Na Seção 6.4, é apresentada a análise do motor implementado, que é realizada de acordo com os requisitos apresentados no Capítulo 4. Por fim, a Seção 6.5 destina-se à exposição das conclusões acerca dos resultados obtidos pela implementação da arquitetura CRAbGE durante esse estudo de caso.

6.2 Aspectos de Pré-Projeto

Antes da realização do estudo de caso propriamente dito, que foi desenvolvido como um projeto de software, os principais aspectos do desenvolvimento foram considerados em uma etapa de pré-projeto. Dessa maneira, foram analisadas as limitações de pessoal e escolhidas

as ferramentas adotadas e a linguagem utilizada para implementação, sendo que a indisponibilidade de pessoal (apenas um desenvolvedor) teve a maior influência nas decisões tomadas durante a realização do presente estudo de caso. Além disso, o impacto do mecanismo de *late binding* dos compiladores C++, descritos a seguir, influenciaram a definição de alguns métodos e de classes abstratas.

6.2.1 Ferramentas Adotadas no Desenvolvimento

Para o desenvolvimento do projeto, optou-se pela utilização, sempre que possível, de ferramentas gratuitas, portáveis e de qualidade que dessem o suporte adequado ao desenvolvimento do estudo de caso através de uma linguagem de programação orientada a objetos.

Durante o desenvolvimento de um projeto, é importante utilizar um sistema de controle de versões para manter um histórico das modificações realizadas no código-fonte, e, além disso, permitir que diversos desenvolvedores trabalhem num mesmo projeto. Para esse fim, foi adotado o sistema Subversion (Tigris, 2004B) por apresentar diversas vantagens sobre o sistema CVS (CollabNet, 2004), principalmente a possibilidade de compactação de dados no servidor e a maior flexibilidade de realocação de diretórios num repositório remoto.

Para facilitar a manutenção da documentação do motor gráfico, optou-se pela utilização da ferramenta gratuita e portável Doxygen (Van Heesch, 2004), que é capaz de gerar automaticamente a documentação de um sistema a partir de *tags* que são inseridas como comentários no código-fonte do sistema. Essa ferramenta suporta, entre outras coisas, a geração de documentação no formato HTML para que seja publicada através de um site da *Web*. Além disso, as *tags* que essa ferramenta utiliza são lidas com facilidade pelo programador, permitindo que sejam usadas como documentação de cada pacote, classe, método e função presente no sistema.

Para a elaboração de diagramas UML, escolheu-se a ferramenta Argo UML (Tigris, 2004A), por ser uma ferramenta gratuita e portável, além de possuir uma interface gráfica bastante amigável. Devido às restrições de pessoal e à enorme quantidade de classes presentes no motor gráfico, optou-se pela utilização de apenas três diagramas UML:

- Diagrama de Componentes, que oferece uma visão geral dos principais componentes e pacotes de classes presentes no motor gráfico;
- Diagrama de Classes, que contém a estruturação das classes presentes em cada módulo do motor; e
- Diagrama de Colaboração, que reflete o inter-relacionamento de um conjunto de classes para a resolução de um problema.

A linguagem de programação C++ foi uma escolha natural para a implementação do motor CGE, pois essa linguagem possui compiladores em praticamente qualquer plataforma, o que propicia o desenvolvimento de um motor portável através de código C++ portável. Além disso, os compiladores C++ geralmente são capazes de gerar código de máquina bastante eficiente em termos de tamanho e velocidade, uma característica desejável a qualquer componente de software cuja eficiência é uma característica fundamental. C++ também possui vários recursos extremamente úteis para o programador, como, por exemplo, macros e templates; e oferece uma interface muito conveniente para a reutilização de bibliotecas portáveis e de domínio público, visto que, na maioria dos casos, tais bibliotecas são escritas em C ou C++.

Uma outra vantagem da linguagem C++ é que ela disponibiliza um conjunto padrão de classes através da Biblioteca Padrão de Gabaritos (*Standard Template Library* - STL), que contém a implementação de estruturas de dados genéricas que podem ser reutilizadas através de uma interface de alto nível. Além disso, a STL contém a implementação de diversos algoritmos que manipulam essas estruturas de dados genéricas de maneira eficiente, fazendo com que a reutilização dessa biblioteca seja uma alternativa bastante atraente para o desenvolvimento de um motor gráfico.

Utilizando esse conjunto de ferramentas, o motor CGE foi desenvolvido em C++, reutilizando estruturas de dados e algoritmos da biblioteca STL. O código-fonte do motor foi compilado sem modificações nas plataformas Windows e Mandrake Linux 10.1. Da mesma maneira, algumas aplicações de teste foram compiladas e executadas nessas plataformas. Na plataforma Windows, as versões 6 e 7 do compilador Visual C++ foram utilizadas, sendo que a implementação da STL distribuída nessas versões foi indeferida em favor da biblioteca STLport (STLport, 2004), uma implementação mais completa e eficiente da STL padrão do C++. Na plataforma Linux, foi utilizado o ambiente de

desenvolvimento GNU, composto pelos compiladores da família GCC (*Gnu Compiler Collection*) e por um conjunto de ferramentas de desenvolvimento. Além disso, o ambiente de desenvolvimento integrado MingW Studio foi utilizado para auxiliar na compilação do motor e de aplicações através dos compiladores GCC 3.2.1.

6.2.2 Os Mecanismos de *Late Binding* nos Compiladores C++

Abstração de dados, herança e polimorfismo são as características essenciais de uma linguagem de programação orientada a objetos. Na linguagem C++, o polimorfismo é implementado através de métodos declarados como virtuais numa classe, sendo que apenas esse tipo de método pode ser sobrescrito por uma especialização da classe original.

A declaração de um método virtual numa classe-base instrui o compilador C++ para não utilizar o mecanismo de *early binding* para esse método durante a etapa de ligação. Isso significa que, quando o compilador encontrar uma chamada a esse método virtual, o mecanismo de *late binding* deve ser utilizado ao invés de realizar uma chamada direta àquele método da classe-base.

O mecanismo de *late binding* geralmente implementado pelos compiladores C++ funciona da seguinte maneira (Eckel, 2000): primeiro, para cada classe que contém métodos virtuais, é criada uma tabela de funções virtuais representando a implementação de cada um desses métodos, e cada instância de uma classe virtual possui um ponteiro que aponta para essa tabela; segundo, durante a etapa de ligação de um arquivo objeto, quando uma subclasse realiza uma chamada a um método virtual, o compilador gera código de máquina "invisível" para recuperar o método correto a partir da tabela de funções virtuais dessa subclasse e realizar uma chamada a esse método.

Assim, a utilização do mecanismo de *late binding* geralmente envolve um custo adicional de processamento, visto que o compilador normalmente gera mais código de máquina para recuperar e invocar funções virtuais em tempo de execução. Além disso, cada instância de uma classe virtual possui um ponteiro implícito para a tabela de funções da sua classe, o que consiste num custo adicional de memória.

Como o mecanismo de *late binding* comumente implementado pelos compiladores C++ afeta o tamanho e o desempenho do código de máquina gerado para classes virtuais, esse recurso será utilizado de forma cautelosa. Assim, um método será declarado como

virtual apenas quando for necessário ou quando esse método realiza um processamento intenso, de maneira a diminuir o impacto do mecanismo de *late binding* no desempenho do motor gráfico.

6.3 CGE: Implementação da Arquitetura CRAbGE

Durante a fase de modelagem, a maioria das entidades que compõem a arquitetura CRAbGE foi mapeada em classes de objetos, e, devido ao grande número de classes presentes no motor, foi criado um pacote (*namespaces*, em C++) para cada grupo de classes que encapsula um conjunto de funcionalidades do motor gráfico. Para facilitar a navegação desses pacotes de acordo com a função que cada pacote desempenha no motor, os mesmos foram organizados em duas hierarquias principais, como ilustrado pela Figura 6.1.

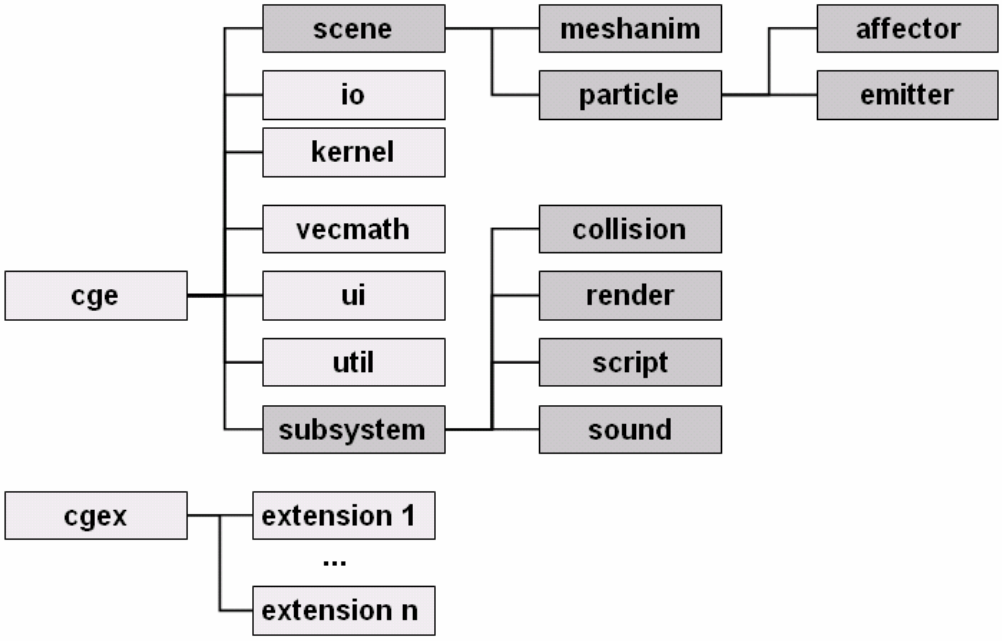


Figura 6.1: Hierarquia de pacotes de classes utilizada no motor CGE.

A hierarquia de pacotes que se inicia no pacote *cge* contém os pacotes, classes e padrões que definem as principais funcionalidades do motor gráfico. Similarmente, a hierarquia enraizada pelo pacote *cgex* é reservada para as classes e pacotes que representam extensões do motor gráfico através de plugins ou de módulos adicionais. Dessa maneira, os seguintes pacotes são utilizados para acessar as funcionalidades padrões do motor gráfico:

- *Scene*, que encapsula o módulo de Gerência de Cenas em alto nível;

- *Io*, que encapsula o Sistema de Arquivos Virtuais do motor gráfico;
- *Kernel*, que compreende os módulos de *Log* do Motor, Relógio do Sistema, Gerência de Recursos, Gerência de *Plugins* e Gerência de Subsistemas;
- *Vecmath*, que reúne as classes que compõem o módulo de Matemática Vetorial;
- *Ui*, que encapsula o módulo de Gerência de Dispositivos de Entrada;
- *Util*, que contém classes utilitárias que podem ser reusadas pelo motor, por extensões do motor e pela camada de aplicação; e
- *Subsystem*, que congrega os pacotes *collision*, *render*, *script* e *sound*, os quais encapsulam os subsistemas básicos do motor gráfico.

Durante a fase de implementação, o recurso de pré-processamento de arquivos através de macros foi utilizado para assegurar a portabilidade do código-fonte do motor, principalmente quanto à seleção de trechos de código para compilação numa plataforma específica. Macros também foram utilizadas para realizar configurações no motor gráfico e facilitar o uso de algumas funcionalidades, como, por exemplo, a inserção de mensagens no arquivo de *log*.

Embora a utilização de macros no motor gráfico prejudique um pouco a legibilidade do código-fonte, esse recurso é utilizado apenas nos métodos relacionados ao relógio do sistema, ao gerenciamento de bibliotecas de vínculo dinâmico e ao sistema de arquivos local.

Essa utilização de macros também evita banalizar o uso de *plugins* no motor gráfico, não sendo necessário delegar a um *plugin* a implementação de funcionalidades relativamente simples que as classes padrões do motor podem oferecer. Isso ainda contribui para facilitar a utilização do motor gráfico, pois, dessa maneira, previne-se que o usuário do motor dependa de um conjunto de *plugins* que varia de acordo com a plataforma.

Para reduzir o esforço de desenvolvimento necessário à implementação do motor CGE, a STL do C++ será utilizada sempre que possível. Essa biblioteca, além de possuir diversas funcionalidades para reuso, também possui containeres muito convenientes à implementação de algumas classes do motor gráfico, como, por exemplo, *vector* e *hashmap*.

O container *vector* utiliza um esquema conservador para a alocação de memória, ou seja, o espaço em memória reservado por uma instância dessa classe num dado momento

corresponde ao maior espaço já alocado nessa instância. Esse esquema de alocação de memória torna essa classe adequada à implementação de uma fila de renderização, oferecendo uma alternativa para contornar possíveis *overheads* de alocação de memória devido a variações no tamanho da fila. Por sua vez, a classe *hashmap* implementa o mapeamento de uma chave num valor de maneira bastante eficiente, realizando uma tarefa indispensável à implementação do módulo de gerência de recursos do motor gráfico.

6.3.1 Núcleo

As funcionalidades disponibilizadas pelo núcleo do motor CGE encontram-se implementadas nos pacotes *io*, *kernel*, *vecmath*, *ui* e *util*. Basicamente, esses pacotes contêm classes que representam as entidades definidas nos módulos que compõem o núcleo do motor na arquitetura CRAbGE. No restante dessa subseção, são apresentados os principais aspectos de implementação de cada um desses pacotes de classes.

Pacote io. No pacote *io*, encontram-se classes que implementam as funcionalidades de fluxos de entrada e saída de dados, representados, respectivamente, pelas classes abstratas *InputStream* e *OutputStream*. O núcleo do motor CGE disponibiliza as seguintes implementações padrões dessas classes abstratas:

- *FileInputStream* e *FileOutputStream*, responsáveis pela leitura e escrita de dados a partir do sistema de arquivos local;
- *ZipInputStream* e *ZipOutputStream*, que implementam a leitura e a escrita de dados compactados através do formato Zip; e
- *RamInputStream*, que implementa a leitura de dados a partir de um buffer da memória.

As bibliotecas Zlib (Gailly & Adler, 2004) e Zip/Unzip (Vollant, 2004) foram utilizadas como base para a implementação das classes *ZipInputStream* e *ZipOutputStream*, visto que essas bibliotecas são gratuitas e portáveis.

Pacote vecmath. Para a implementação do pacote *vecmath*, optou-se por abolir a utilização das possíveis relações de herança entre as classes Vector2D, Vector3D e Vector4D, evitando que o mecanismo de *late binding* afete o desempenho dessas classes, pois as mesmas serão utilizadas em praticamente todos os outros módulos do motor.

Além disso, praticamente todas as classes desse pacote utilizam a sobrecarga de operadores para realizar operações como adição, multiplicação, produto escalar e produto vetorial. Isso permite que o programador utilize expressões com esses tipos de objetos, obtendo uma representação mais compacta e compreensível dessas operações vetoriais. Por outro lado, essas operações também podem ser realizadas através de métodos, permitindo sua utilização por programadores pouco familiarizados com o uso de operadores sobrecarregados.

Pacote kernel. A classe *Timer*, presente no pacote *kernel*, implementa o acesso ao relógio do sistema de duas maneiras, sendo que a compilação do código-fonte dessa classe é controlada por macros de configuração. Na plataforma Windows, essa classe utiliza os mecanismos do sistema operacional para acessar o relógio do sistema com uma precisão altíssima, geralmente na casa dos nanossegundos. Nas demais plataformas, timer acessa o relógio do sistema com uma precisão de milissegundos através da função *clock*, disponível na biblioteca padrão C/C++.

Como implementação concreta do módulo de *Log* do motor, o pacote *kernel* disponibiliza uma implementação padrão da classe abstrata *Logger*, realizada na classe *HtmlLogger*, que armazena mensagens, advertências e erros como uma página HTML. Esse tipo de formatação facilita a leitura do arquivo de *log* e, além disso, permite sua imediata publicação como página da Web. A Figura 6.2 ilustra um arquivo de *log* gerado durante a execução de uma aplicação com o motor CGE.

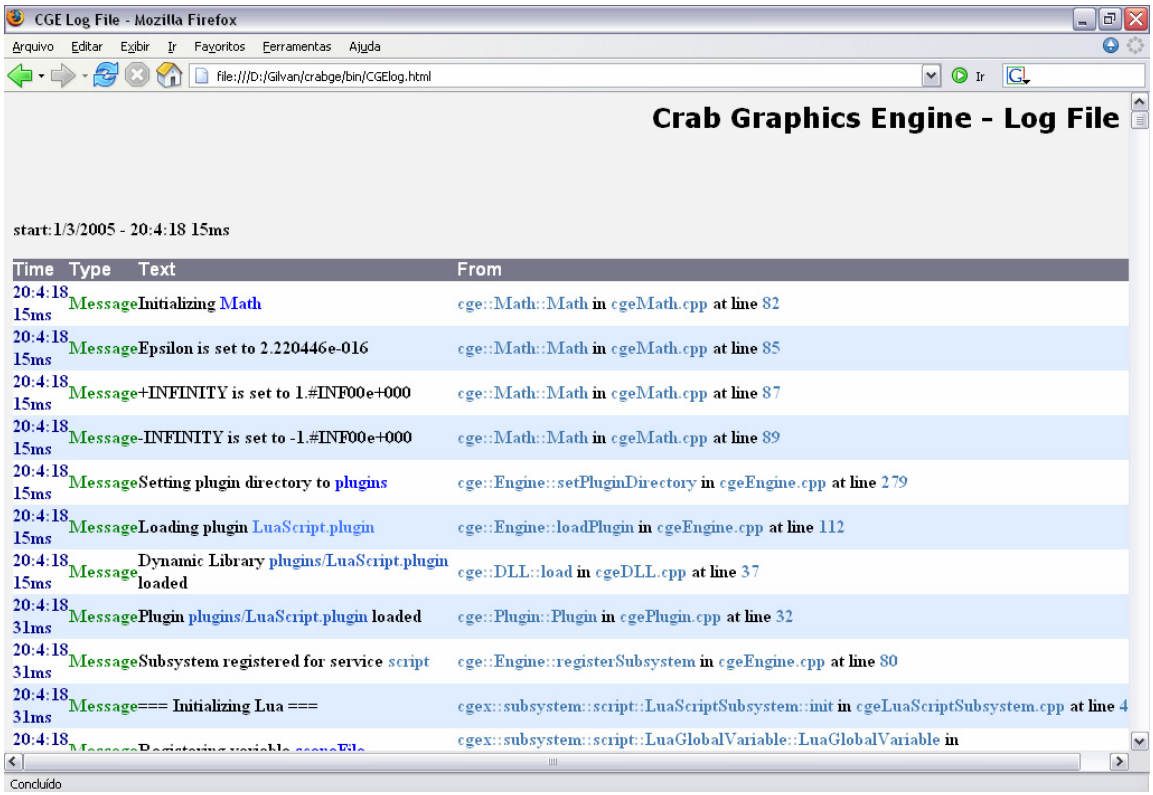


Figura 6.2: Arquivo de log gerado durante a execução de uma aplicação com o motor CGE.

Para facilitar ainda mais a inclusão de mensagens nesse arquivo de *log*, foram criadas as macros CGE_MSG, CGE_ERR e CGE_WARN, que recuperam a instância *singleton* de *HtmlLogger*, e, respectivamente, inserem mensagens genéricas, erros e advertências no arquivo de *log* do motor gráfico.

A classe *DynamicLibrary*, presente no pacote *kernel*, é essencial para a implementação do mecanismo de *plugins*, possuindo duas implementações para acessar uma biblioteca de vínculo dinâmico, sendo que a implementação adequada à plataforma é selecionada em tempo de compilação através de macros. Na plataforma Windows, esse acesso é realizado através das funções *LoadLibrary*, *GetProcAddress* e *FreeLibrary*, presentes na API do sistema operacional. Nas demais plataformas, esse acesso é realizado através das funções *dlopen*, *dlsym* e *dlclose*, que são disponibilizadas pela biblioteca *ld*, presente na maioria dos sistemas baseados no sistema UNIX (Shah & Xiao, 1998).

Pacote ui. O pacote ui encapsula a implementação do módulo de Gerência de Dispositivos de Entrada. Nesse pacote, as funcionalidades da maioria das classes foram definidas através de métodos virtuais, permitindo que o usuário do motor CGE realize customizações através

da implementação das interfaces definidas pelas classes abstratas *Device* e *DeviceEventTranslator*. A definição de *Device* como uma classe abstrata permite, por exemplo, que essa classe seja especializada para que produza eventos artificiais usados para controlar simulóides no mundo virtual com base em técnicas de visão computacional.

Pacote util. O pacote *util* congrega classes que podem ser reutilizadas tanto pela camada de aplicação quanto por implementações de subsistemas. Atualmente, esse pacote contém as classes *String* e *Bitset*. *String* representa uma cadeia de caracteres que pode ser manipulada através de operações tais como as de concatenação e comparação.

Apesar da STL do C++ possuir uma classe para armazenar um conjunto de bits que pode ser utilizada em algoritmos que manipulam um grande número de dados, essa classe não permite redimensionar o conjunto de bits. Assim, a classe *Bitset* foi desenvolvida para contornar essa limitação e prover um conjunto de bits de tamanho variável. Futuramente, outras classes utilitárias poderão ser adicionadas a esse pacote, como algoritmos para o cálculo de fechos convexos e para a pré-computação de iluminação global, por exemplo.

6.3.2 Subsistemas

As funcionalidades disponibilizadas pelos subsistemas do motor CGE encontram-se distribuídas nos pacotes *subsystem*, *collision*, *render*, *script* e *sound*. De maneira geral, esses pacotes contém classes que encapsulam as entidades definidas em cada um dos subsistemas básicos da arquitetura CRAbGE.

Pacote subsystem. O pacote *subsystem* contém apenas a classe abstrata *Subsystem*, que define uma interface comum a todos os subsistemas do motor gráfico.

Pacote collision. O pacote *collision* congrega as classes que encapsulam o subsistema de detecção de colisões, integrando diversas técnicas para o cálculo de interseção e distância entre objetos estacionários (Magic Software, 2004; Terdiman, 2003). A classe concreta *Triangle*, disponibilizada nesse pacote, representa um triângulo no espaço tridimensional e permite realizar o cálculo de normal, perímetro e área, além do cálculo de interseção com outro triângulo através da técnica descrita em Möller (1997).

Atualmente, esse pacote contém a implementação de testes de interseção entre pares de volumes estacionários, definidos pelas classes concretas *AABB*, *OBB*, *Sphere* e *Capsule*. Também é possível realizar testes de *picking* contra esses volumes, ou seja, detectar a

interseção de um volume estacionário contra um segmento de reta ou um raio. Além disso, esse subsistema permite calcular a distância entre um par de primitivas (*AABB*, *OBB*, *Sphere*, *Capsule*, *Ray* e *Line*), sendo que alguns dos testes de interseção são realizados através da computação dessas distâncias.

Esse pacote também permite realizar o teste de interseção entre um triângulo e uma dessas primitivas, o que serve como base para a implementação de algoritmos mais complexos, como o cálculo de interseção entre modelos geométricos representados através de uma hierarquia de volumes envoltórios, por exemplo.

A classe abstrata *CollisionModel* permite pré-calcular as normais às faces de um modelo geométrico rígido e armazená-las numa *cache* local, de maneira a evitar a realização desses cálculos durante a execução de uma aplicação, o que pode ser inviável, dependendo da aplicação e do número de faces presente no modelo.

Pacote render. Devido à complexidade do subsistema de renderização em tempo-real e à falta de pessoal durante o desenvolvimento do motor, decidiu-se utilizar uma abordagem baseada em versões para permitir uma implementação incremental desse subsistema. Dessa maneira, cada versão desse subsistema é definida com base num subconjunto de funcionalidades presente nessa versão e que continuará presente nas versões seguintes. De acordo com essa estratégia de desenvolvimento, foram definidas três versões principais de forma que, após a conclusão dessas versões, as funcionalidades previstas na arquitetura CRAbGE estarão totalmente implementadas nesse subsistema.

Essas versões variam, principalmente, de acordo com o tipo de material que pode ser aplicado aos objetos numa cena tridimensional e pela capacidade de customizar a fila de renderização. No que diz respeito ao impacto causado no motor gráfico pela evolução desse subsistema, estima-se que apenas a especificação de efeitos visuais através de materiais deva sofrer maiores modificações, de maneira que os demais módulos do motor permanecem inalterados. Assim, as seguintes versões foram definidas como etapas de desenvolvimento:

- A primeira versão baseia-se nos recursos da pipeline *fixed function* para definir os materiais que são aplicados, num único passo, aos objetos da cena que são armazenados numa fila de renderização não customizável;

- A segunda versão expande as funcionalidades da versão anterior, sendo que os materiais utilizam múltiplos passos para obter melhores efeitos visuais e a fila de renderização passa a ser customizável; e
- A terceira versão prevê a utilização dos recursos da pipeline programável através de linguagens de *shading*, bem como a implementação de um conjunto padrão de efeitos visuais, como, sombras dinâmicas e *bump mapping*, por exemplo.

Com o objetivo de simplificar o processo de geração de texturas a partir de arquivos de imagens, a biblioteca *Developer's Image Library* (DevIL, 2004) foi adotada como módulo auxiliar na implementação do subsistema de renderização, visto que essa biblioteca permite carregar imagens na memória a partir de inúmeros formatos de arquivo, além de ser gratuita, portável e oferecer suporte ao processamento dessas imagens. Essa biblioteca é utilizada pela classe abstrata *TextureManager* para carregar imagens na memória, de maneira que as implementações do subsistema de renderização abstraem-se dos formatos de arquivo utilizados e são responsáveis apenas pela criação efetiva de texturas a partir de imagens já carregadas na memória principal.

Durante a implementação do subsistema de renderização, foi desenvolvido o seguinte mecanismo de pré-processamento de texturas. A camada de aplicação fornece uma resolução máxima para as imagens usadas na criação de texturas, de maneira que imagens excedendo esse limite de resolução são redimensionadas e filtradas em memória antes da criação efetiva de uma textura a partir dessa imagem. Assim, esse mecanismo permite ajustar, de maneira automática, a resolução das texturas ao nível de detalhe suportado pelo hardware gráfico disponível ou a um nível de detalhe satisfatório para uma aplicação específica.

No que diz respeito à renderização em alto nível, a entidade *SceneManager* foi implementada como uma classe concreta, permitindo que a camada de aplicação utilize uma instância dessa classe para lidar com cenas compostas apenas por objetos dinâmicos representados através de um grafo de cena. Isso permite diminuir o impacto no desempenho causado pela utilização de vários métodos virtuais, e, além disso, permite que uma aplicação utilize apenas objetos dinâmicos quando lhe for mais conveniente.

Para realizar processamentos afins necessários a cada quadro pintado pelo subsistema de renderização, esse subsistema permite utilizar um mecanismo de *frame listener* similar ao utilizado no sistema OGRE e nos motores Fly3D e Quake III. O subsistema de renderização define uma interface através da classe abstrata *FrameListener*, de maneira que especializações dessa classe podem ser registradas numa instância de *RenderSubsystem* para realizar tarefas da camada de aplicação, como, por exemplo, fechar a aplicação ou movimentar um avatar de acordo com as entradas do usuário.

Pacotes script e sound. Basicamente, as implementações dos pacotes *script* e *sound* no motor gráfico foram realizadas através da definição da interface desses subsistemas através de classes abstratas, pois esses subsistemas não possuem funcionalidades passíveis de implementação sem que se utilize um interpretador ou uma API de som espacial. Portanto, a implementação das principais funcionalidades presentes nesses subsistemas é delegada às suas respectivas especializações concretas.

Futuramente, pretende-se adotar um esquema para o carregamento de sons pré-amostrados similar ao implementado no subsistema de renderização para o carregamento de texturas subsistema de renderização para carregar texturas. Dessa maneira, futuras implementações do subsistema de som espacial poderão abstrair-se de formatos de arquivos de áudio e da realização de possíveis pré-processamentos necessários para adequação de sons pré-amostrados ao hardware de áudio disponível ou ao nível de detalhe suficiente para uma determinada aplicação.

6.3.3 Gerência de Cenas em Alto Nível

Com o objetivo de reduzir o impacto causado pela utilização de métodos virtuais, as funcionalidades das entidades *Node* e *SceneNode* foram aglutinadas numa única classe: *Node*. A implementação da classe Node utiliza uma hierarquia híbrida composta por OBBs e esferas para acelerar a pintura através da técnica de *Hierarchical Frustum Culling*. Além disso, é possível habilitar o modo de depuração durante a pintura de uma cena, o que permite visualizar as arestas que compõem os objetos em geral, além dos volumes envoltórios que são usados como aproximação de hierarquias de objetos dinâmicos durante a aplicação de técnicas de *culling*.

Durante a execução de uma aplicação, as classes *Entity*, *BillboardSet* e *ParticleSystem* , que são especializações de *Node*, atualizam esses volumes envoltórios com base na geometria utilizada localmente para representar esses objetos dinâmicos. Assim:

- As instâncias de *Entity* utilizam uma AABB que envolve a instância de *Mesh* no espaço local de coordenadas; e
- As instâncias de *BillboardSet* e *ParticleSystem* utilizam uma AABB que engloba o centro de cada *billboard* que compõe a geometria desses objetos, sendo que essa AABB é ligeiramente extrapolada para melhor representar essa nuvem de *billboards*.

O cálculo do vetor velocidade é implementado pelas classes *SoundNode* e *SoundListener* da seguinte maneira. Computa-se o vetor deslocamento correspondente à variação da posição, no sistema global de coordenadas, entre dois quadros consecutivos. Esse vetor é utilizado, em conjunto com o tempo decorrido entre esses dois quadros, para calcular o vetor velocidade utilizado pelas instâncias dessas classes. Esse cálculo de velocidade, ilustrado pela Figura 6.3, possui a limitação de desconsiderar a trajetória descrita pelo objeto num intervalo de tempo.

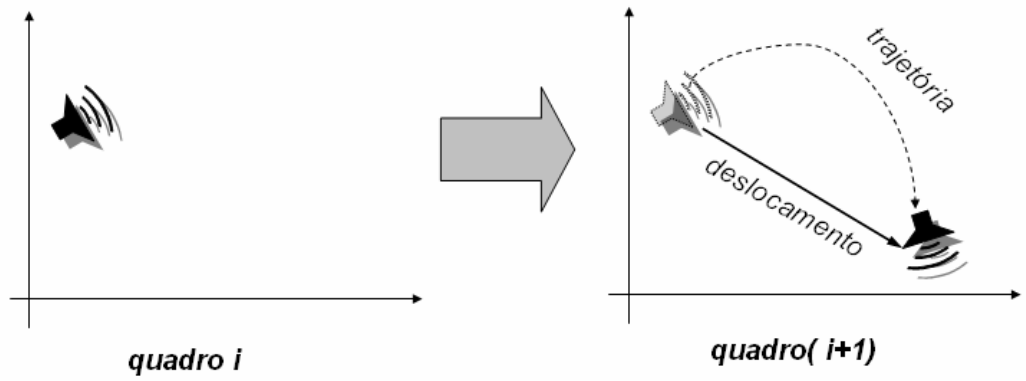


Figura 6.3: Cálculo de velocidade baseado no deslocamento entre dois quadros consecutivos.

Entretanto, essa abordagem é computacionalmente eficiente e produz resultados satisfatórios quando a cena é atualizada dezenas de vezes por segundo. Futuramente, pretende-se utilizar uma abordagem mais rigorosa para o cálculo de velocidades, considerando a trajetória dos objetos ao longo de uma curva.

6.3.4 *Plugins* Desenvolvidos

Diferentemente da filosofia adotada nos motores CrystalSpace e Fly3D, cada *plugin* padrão desenvolvido para o motor gráfico CGE é totalmente independente dos demais *plugins*, o que confere maior flexibilidade, capacidade de configuração e facilidade de uso ao motor CGE. Isso significa que o *plugin* "A" não é desenvolvido em função do *plugin* "B", que é reutilizado de maneira explícita para acessar funcionalidades específicas.

RenderSubsystems. Atualmente, o motor CGE possui apenas um *plugin* contendo uma especialização da classe abstrata *RenderSubsystem*, que encapsula o subsistema de renderização em baixo nível. Essa especialização é realizada pela classe *OpenGLRenderSubsystem*, que utiliza internamente a biblioteca gráfica OpenGL.

Essa classe foi implementada com base na versão 1.2.1 do OpenGL e possui a extensão de múltiplas texturas como requisito adicional. Entretanto, é preferível que o *driver* OpenGL utilizado por essa classe ofereça suporte às funcionalidades da versão 1.3 ou superior, pois as mesmas possuem mecanismos mais flexíveis para a utilização de texturas, como compressão, combinação em camadas e *cube maps*, por exemplo.

Esse *plugin*, além de ser portável, prevê a integração com diversas APIs para construção de interface gráfica com o usuário. Atualmente, esse subsistema permite utilizar janelas construídas com a biblioteca SDL (SDL, 2004) ou com a API do Windows. Devido às restrições de pessoal, o desenvolvimento de um *plugin* baseado na API gráfica Direct3D foi postergado para versões futuras do motor.

SceneManagers. Duas implementações de *SceneManager* foram desenvolvidas como *plugins* durante o estudo de caso. Um desses *plugins* baseia-se numa *octree* para acelerar a pintura de objetos estáticos que são carregados a partir de um arquivo no formato 3DS (Autodesk, 1994), sendo que essa técnica é mais adequada a renderização de cenários abertos. O carregamento de arquivos 3DS é realizado através de uma versão expandida da biblioteca Lib3ds (Lib3ds, 2004), distribuída segundo os termos da licença LGPL e que é capaz de ler esses tipos de arquivos através do Sistema de Arquivos Virtuais do motor CGE.

O outro plugin baseia-se na combinação das técnicas de PVS e de árvores BSP para acelerar a pintura dos objetos estáticos que são carregados a partir de arquivos de mundo no

formato do motor Quake III, sendo que essa técnica é mais adequada a renderização de cenários fechados, típicos de jogos de tiro em primeira pessoa.

CollisionSubsystems. Atualmente, o motor CGE possui apenas um *plugin* padrão contendo uma implementação do subsistema de detecção de colisões. Essa implementação utiliza a biblioteca OPCODE (Terdiman, 2003), que foi modificada durante o desenvolvimento para suportar a utilização de escalas nos testes de interseção contra modelos geométricos deformáveis.

Atualmente, estuda-se a possibilidade de desenvolvimento de mais duas implementações desse subsistema, uma utilizando uma biblioteca de detecção de colisões própria e a outra com a biblioteca SOLID (Van Der Bergen, 1999A).

SoundSubsystems. Atualmente, o motor CGE possui três implementações padrões do subsistema de som espacial que são distribuídas como *plugins* distintos e portáveis, sendo que esses *plugins* contêm as seguintes especializações de *SoundSubsystem*:

- *OpenALSoundSubsystem*, que se baseia na utilização da biblioteca OpenAL e oferece suporte à utilização de sons pré-amostrados no formato WAV;
- *FmodSoundSubsystem*, que se baseia na utilização da biblioteca *fmod*, a qual, apesar de utilizar um modelo limitado de som espacial, permite implementar as funcionalidades básicas desse subsistema e oferece suporte ao carregamento de sons pré-amostrados a partir de vários formatos, como, por exemplo, MP3 e Ogg Vorbis; e
- *NoSoundSubsystem*, implementação do subsistema de som espacial que, apesar de não produzir saídas de áudio, oferece suporte à execução de aplicações em plataformas limitadas que não permitem produzir estímulo sonoro e em situações em que se deseja omitir a elaboração desse tipo de estímulos, como, por exemplo, em editores de mundos virtuais.

ScriptSubsystems. O motor gráfico CGE possui uma implementação padrão do subsistema de script, realizada através de um *plugin* que utiliza a linguagem Lua (Ierusalimschy et al., 2003), escolhida por ser gratuita e simples, por apresentar bom desempenho e por ter grande poder de expressão. Atualmente, está sendo realizado o *binding* das principais

classes do motor para a linguagem Lua. Além disso, pretende-se desenvolver outras implementações desse subsistema baseadas nas linguagens Python e Ruby.

Técnicas de Animação. Atualmente, o motor gráfico CGE possui um *plugin* para o carregamento de modelos no formato MD2 (Schoenblum, 2004), que são animados através da técnica de interpolação linear de quadros-chaves. Cada um desses quadros-chaves é representado pela posição de cada vértice do modelo num determinado instante, de maneira que apenas a geometria do modelo é interpolada enquanto a topologia permanece constante. Futuramente, pretende-se implementar técnicas de animação baseadas numa hierarquia de *bones*, pois tais técnicas produzem efeitos mais realistas e, além disso, oferecem maior flexibilidade quanto à utilização pelo programador.

Leitores de Dispositivos. Devido à indisponibilidade de dispositivos não-convencionais para a realização de testes, foram desenvolvidos apenas dois *plugins* para a leitura de dispositivos de entrada. Um desses plugins utiliza a API do sistema operacional Windows para o reconhecimento de entradas a partir de mouse, teclado e j*oystick*. O outro *plugin* foi desenvolvido com a biblioteca portável SDL, permitindo utilizar esses dispositivos em diversas plataformas tais como Linux, Windows, Mac e Playstation 2.

6.4 Análise do Motor CGE

Nesta Seção são apresentados alguns aspectos de desenvolvimento de aplicações com o motor CGE, e é feita uma avaliação do motor implementado, segundo os critérios de avaliação apresentados no Capítulo 4.

6.4.1 Aplicações Desenvolvidas com o Motor CGE

A presente subseção tem por objetivo demonstrar a utilização do motor em aplicações, apresentando alguns exemplos através de trechos de código C++ e de imagens ilustrando esses exemplos em tempo de execução. O motor CGE é inicializado, antes que a aplicação comece a ser executada, através da seguinte seqüência de passos:

- Carregamento dos *plugins* utilizados pelo motor gráfico;
- Inicialização dos subsistemas, que são registrados no motor pelos *plugins*;
- Recuperação do gerenciador de cena, que opcionalmente é registrado no motor através de um *plugin*;

- Criação de janela para exibir as cenas renderizadas;
- Criação de uma câmera e de uma *viewport* nessa janela de renderização;
- Registro de diretórios virtuais contendo as mídias utilizadas pelo motor gráfico;
- Carregamento de uma cena tridimensional; e
- Entrada no *loop* principal, que inicia a renderização da cena.

A inicialização do motor CGE é bastante simples e pode ser realizada através de um pequeno número de linhas de código, como no exemplo ilustrado pela Tabela 6.1.

Tabela 6.1: Exemplo de código de uma aplicação básica utilizando o motor CGE.

```
#include "cge.h"
using namespace cge;

int main()
{
  // Recuperação da entidade central do motor.
  Engine& engine = Engine::getInstance();

  // Carregamento de plugins
  engine.loadPlugin("OpenGLRender.plugin");
  engine.loadPlugin("OpcodeCollision.plugin");
  engine.loadPlugin("OpenALSound.plugin");
  engine.loadPlugin("LuaScript.plugin");
  engine.loadPlugin("OctreeSceneManager.plugin");

  // Inicialização do motor gráfico
  engine.initSelectedSubsystems();

  // Recuperação do subsistema de renderização registrado no motor
  RenderSubsystem* render = (RenderSubsystem*)
                engine.getSelectedSubsystem(Subsystem::RENDER);

  // Recuperação do gerenciador de cenas registrado
  SceneManagerEnumerator* sme;
  SceneManager* sceneMan;
  sme = SceneManagerEnumerator::getInstance();
  sceneMan = sme->getSceneManager( SceneManager::SCENE_GENERIC );

  // Criação de uma janela de renderização
  RenderWindow* win = render->createRenderWindow("CGE Test",// título
                                                 32,       // bpp
                                                 1024, 768,// resolução
                                                 true      // tela cheia
                                                 );

  // Criação de uma câmera e uma viewport nessa janela de renderização
  Camera* cam = sceneMan->createCamera("test camera");
  Viewport* vp = win->addViewport( cam, 0,0,1,1 );

  // Adição de um diretório de recursos,
  // carregamento de uma cena e início da pintura
  ResourceDirectory::getInstance().addResourceDirectory("fortaleza.zip");
```

```
  sceneMan->setWorldGeometry("fortaleza.3ds");

  // Início do loop de pintura
  render->startRendering();

  return 0;
}
```

Várias aplicações de teste foram desenvolvidas de maneira incremental, ou seja, cada nova aplicação adicionou novas funcionalidades às aplicações anteriores. Essas aplicações, que são voltadas à visualização de modelos animados e à navegação em mundos virtuais, foram compiladas nos ambientes Windows e Mandrake Linux 10.1, sendo testadas em microcomputadores PC com as seguintes configurações de hardware:

- Processador Pentium 4 de 2.0 GHZ com 256 de memória RAM e placa de vídeo GeForce FX 5200 com 128MB de memória;
- Processador Athlon XP 2000+ de 1.66 GHZ e 512 de memória RAM e placa de vídeo GeForce 2 *onboard* com 32MB de memória;
- Processador Pentium 4 de 1.5 GHZ com 512 de memória RAM e placa de vídeo ATI RAGE 128 Pro de 32MB de memória.

Essas aplicações de teste construídas com o motor CGE foram executadas nas plataformas Windows e Linux, utilizando os seguintes mundos virtuais, que apresentam variações quanto ao número de polígonos e ao tipo de cenário:

- Hotel, um cenário fechado carregado a partir de um arquivo 3DS com 6 mil polígonos;
- Aeroporto, um cenário fechado carregado a partir de um arquivo 3DS com mais de 12 mil polígonos;
- Núcleo de Processamento de Dados da Universidade Federal (NPD-UFC), um cenário misto carregado a partir de um arquivo 3DS com 13 mil polígonos;
- Domo, um mundo virtual com dois ambientes, um aberto e outro fechado, carregado a partir de um arquivo 3DS com 37500 polígonos;
- Um mapa para o jogo Quake 3 contendo mais de 9 mil faces; e
- Um mapa para o jogo *Wolfenstein - Enemy Territory*, contendo mais de 87 mil faces.

As aplicações foram desenvolvidas incrementalmente, explorando as principais funcionalidades presentes no motor CGE como se segue. Os resultados aqui apresentados foram obtidos num PC com processador Pentium 4 de 2.0 GHZ e placa de vídeo GeForce FX 5200.

Numa primeira aplicação, são adicionados alguns diretórios e arquivos compactados ao sistema de arquivos virtuais, que são utilizados para carregar as texturas de uma cena simples, composta apenas por uma representação do céu através de um cubo texturizado. Além disso, essa aplicação registra uma especialização de *FrameListener* no subsistema de renderização para exibir a taxa de quadros por segundo obtida pelo motor gráfico. Essa aplicação, que é ilustrada pela Figura 6.4 (à direita), obtém uma taxa constante 194 quadros por segundo a uma resolução de 800 por 600 *pixels* com 32 bits de profundidade de cor no modo de janelas. Já no modo de tela cheia, essa aplicação obtém uma taxa de 358 quadros por segundo.

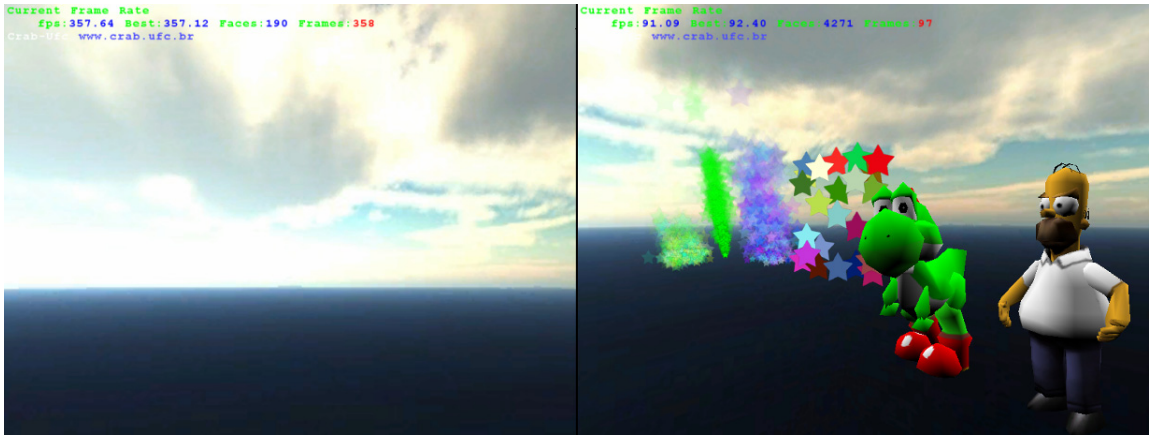


Figura 6.4: Cena simples contendo apenas uma representação do céu.

Numa segunda aplicação, foram alguns *billboards* e sistemas de partículas adicionados à aplicação anterior, além de dois modelos geométricos animados que são carregados a partir de arquivos no formato MD2. Para tanto, é necessário carregar um *plugin* para reconhecer esse formato de arquivos. Nas mesmas condições da aplicação anterior, obteve-se uma taxa de 97 quadros por segundo nessa segunda aplicação, que é ilustrada pela Figura 6.4, à direita.

Numa terceira aplicação, foram adicionadas duas fontes sonoras que são adicionadas como nós-filhos dos modelos utilizados na aplicação anterior. Da mesma maneira, foi adicionado um nó do tipo *Listener* como filho da câmera usada para navegar

no mundo. Através dessa aplicação, testaram-se os efeitos sonoros produzidos pelo subsistema de som espacial mediante a troca de suas implementações através das bibliotecas FMOD e OpenAL, além da emulação realizada pela implementação *NoSound*.

Numa outra aplicação, aprimorou-se o algoritmo de movimentação de câmera para impedir que a mesma atravesse os objetos da cena. Além disso, quaisquer objetos dinâmicos interceptados pela câmera são deslocados no espaço global de coordenadas de acordo com o deslocamento da câmera, o que produz um efeito semelhante a empurrar tais objetos. Foram testadas variações desse algoritmo de movimentação, alternando a utilização de segmentos de reta, esferas, AABBs, OBBs e Cápsula. Para tanto, é carregado um *plugin* contendo a implementação do subsistema de colisão através da biblioteca Opcode.

Numa variação dessa aplicação, testaram-se, através do algoritmo de movimentação, os cálculos de interseção entre modelos geométricos deformáveis em diversas posições, orientações e escalas. Os testes realizados demonstraram que esses testes de interseção causam um impacto praticamente inexistente no desempenho das aplicações, ao passo que a atualização da representação interna desses modelos mediante deformações possui um custo computacional mais significativo, pois todos esses modelos são atualizados a cada quadro. Futuramente, pretende-se desenvolver um mecanismo mais eficiente para a atualização desses modelos, postergando a atualização da representação interna desses modelos e limitando a taxa de atualização de cada modelo.

Numa outra aplicação, adicionou-se o carregamento de cenários abertos e fechados a partir de cenas pré-moldadas nos formatos 3DS e BSP (formato de cenas do motor Quake III). Para tanto, alternou-se o carregamento dos *plugins* para a utilização dos gerenciadores de cena apropriados. Essa aplicação, além de permitir avaliar como o motor CGE pinta cenas complexas, permitiu testar várias situações de interseção entre os modelos geométricos deformáveis e os objetos estáticos da cena. Vários mundos virtuais foram utilizados nessa aplicação, que foi executada nas plataformas Windows e Linux, como ilustrado pelas Figuras 6.5, 6.6, 6.7, 6.8, 6.9, 6.10, 6.11, 6.12, 6.13 e 6.14, apresentadas a seguir.

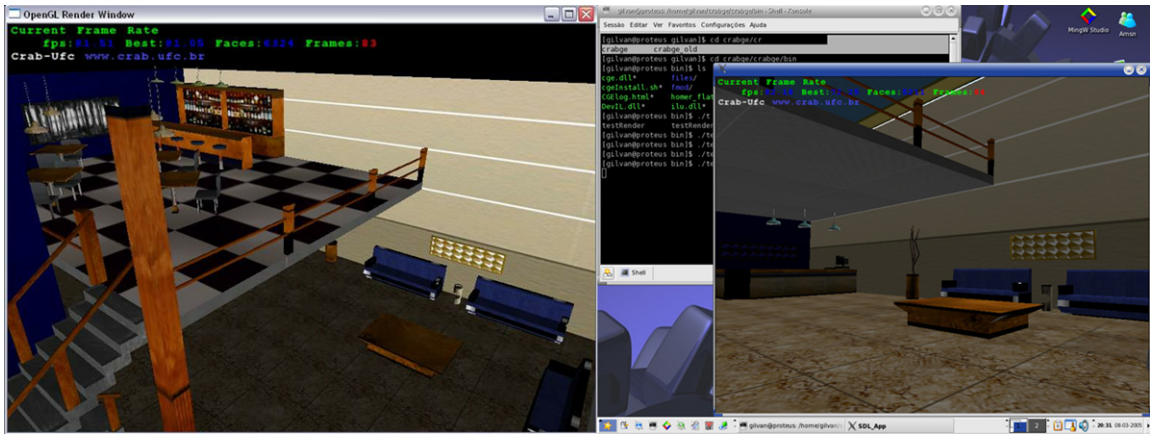


Figura 6.5: Mundo Aeroporto Virtual sendo visualizado nas plataformas Windows (à esquerda) e Linux (à direita)

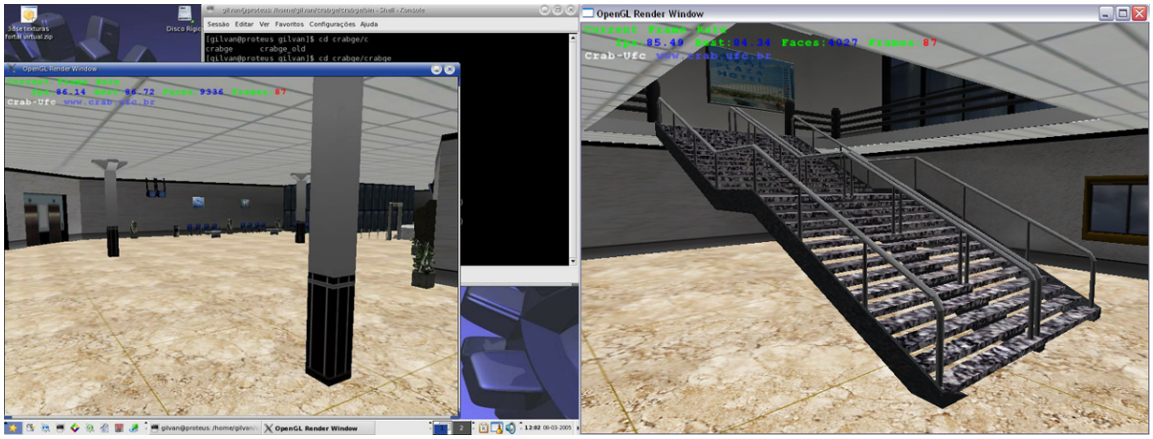


Figura 6.6: Mundo Aeroporto Virtual, sendo visitado nas plataformas Windows (à direita) e Linux (à esquerda).

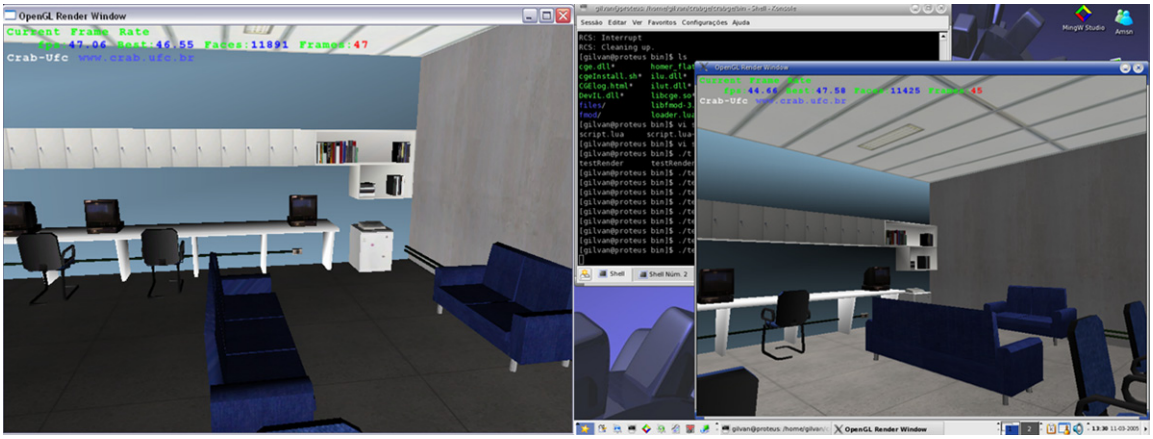


Figura 6.7: Mundo virtual representando o Núcleo de Processamento de Dados da Universidade Federal do Ceará, nas plataformas Windows (à esquerda) e Linux (à direita).

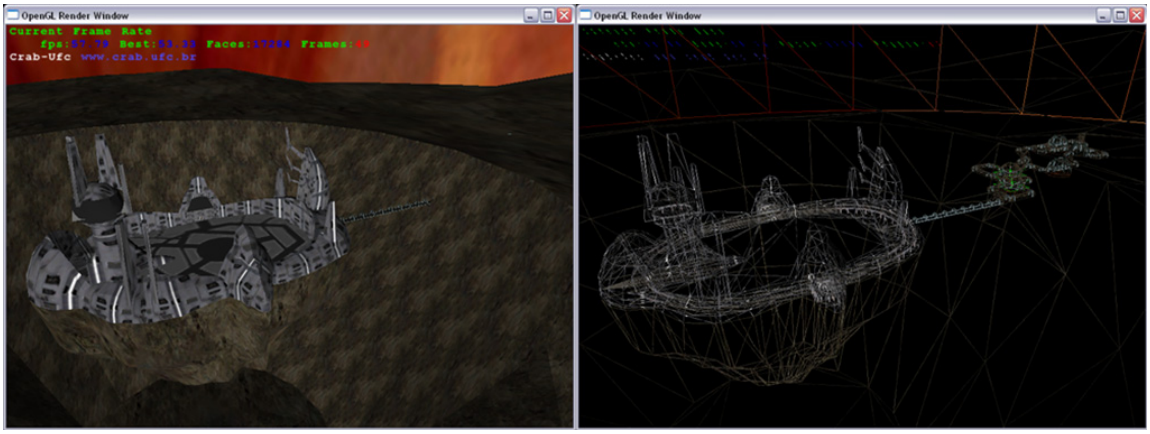


Figura 6.8: Domo, um mundo virtual complexo composto por 37500 polígonos, sendo explorado na plataforma Windows. À esquerda, uma visão aérea do mundo virtual, que é pintado a uma taxa de 49 quadros por segundo. À direita, essa mesma situação é renderizada em modo *wireframe*.

Figura 6.9: Mapa do jogo Quake 3 renderizado pelo motor CGE na plataforma Windows. A iluminação global é pré-calculada através do método de radiosidade e aplicada, em tempo de execução, através de uma camada modulativa de texturas. Esse tipo de texturas é conhecido como mapa de iluminação (*lightmap*).

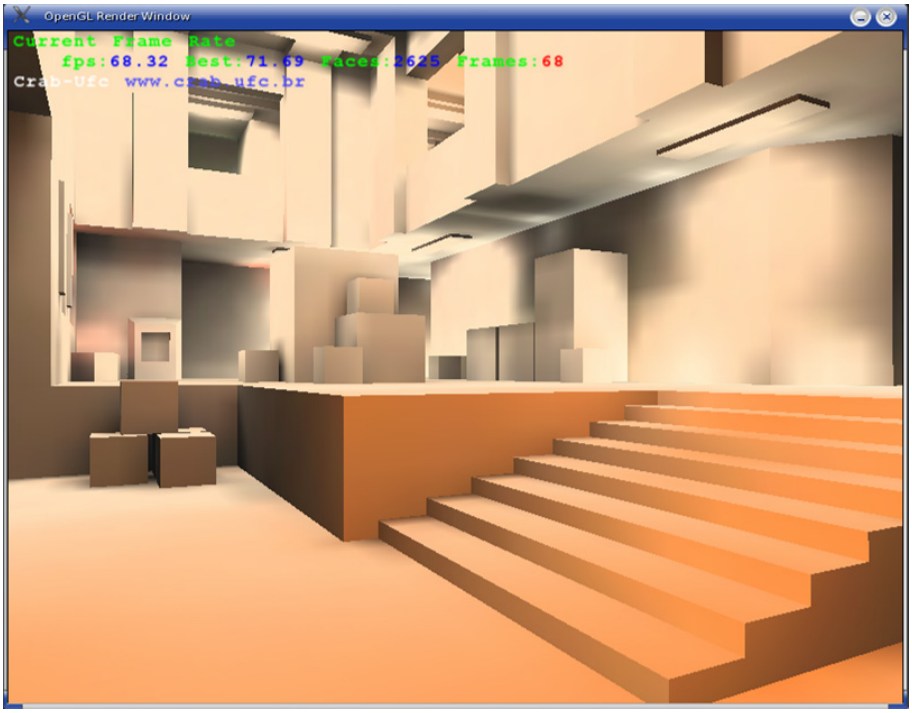


Figura 6.10: Um cenário no formado do jogo Quake III, renderizado apenas com os *lightmaps*, que são utilizados para obter o efeito de iluminação global na plataforma Linux.

Figura 6.11: Um cenário complexo do jogo Enemy Territory, contendo mais de 87 mil polígonos. Nessa figura, uma aplicação do Windows é usada para ilustrar um método alternativo de aplicar a iluminação global. Esse método baseia-se na modulação da cor dos vértices com uma camada de textura única.

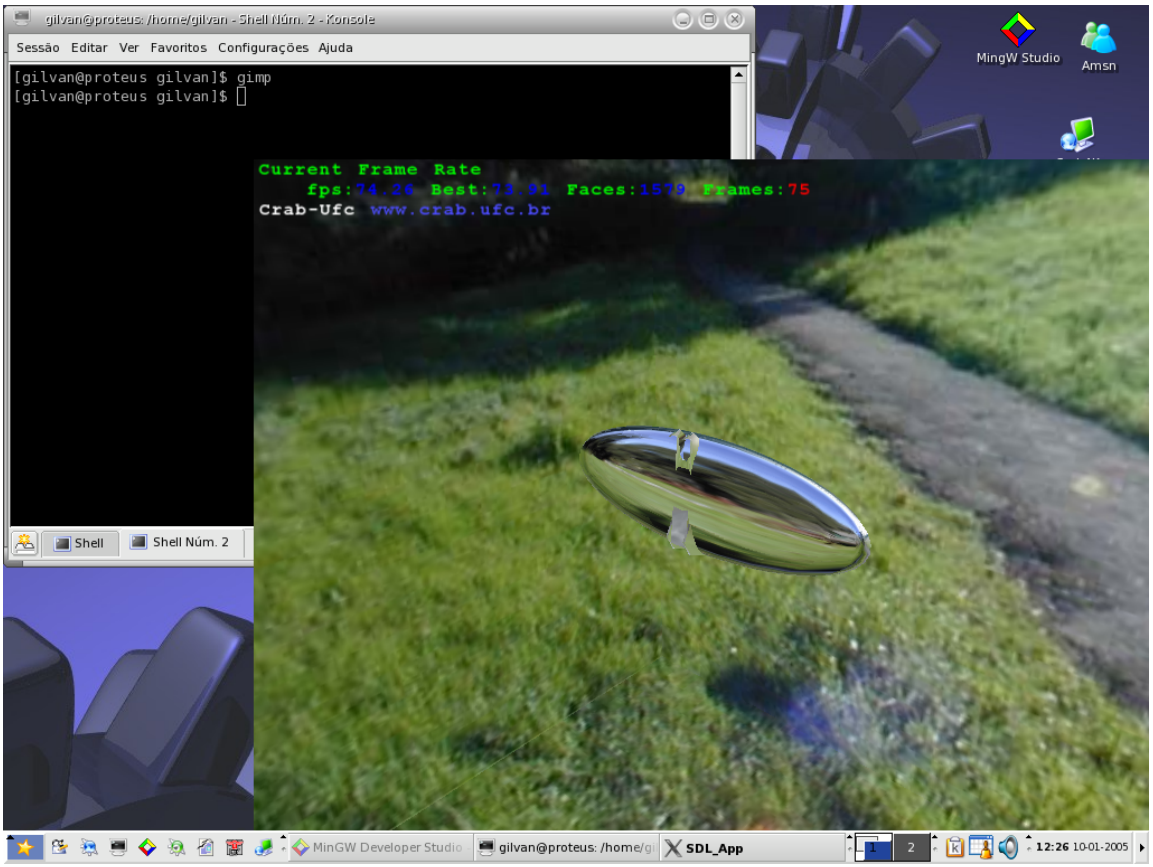


Figura 6.12: Efeito de reflexão obtido na primeira versão do subsistema de renderização do motor CGE, na plataforma Linux, através da utilização de *cubemaps* estáticos.

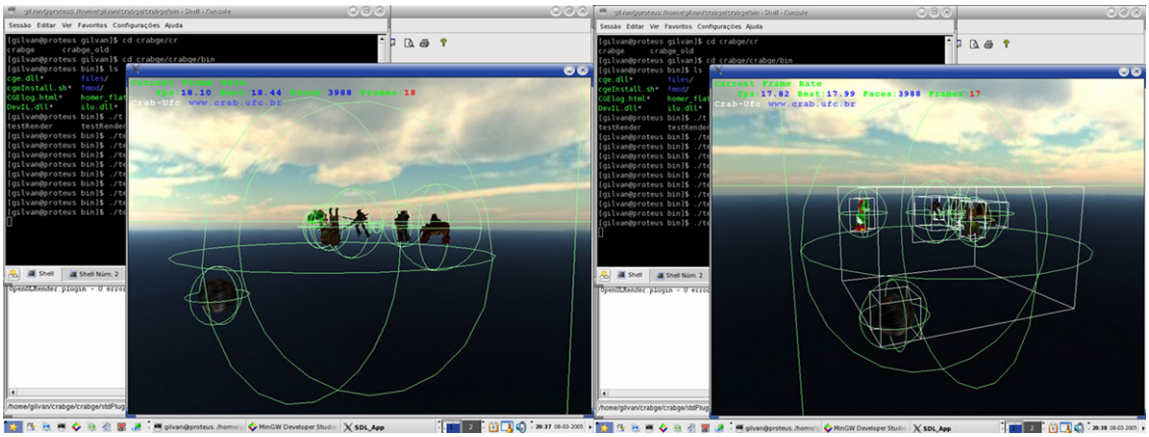


Figura 6.13: Modos de depuração de cenas disponíveis no motor CGE. Em verde, as esferas que envolvem hierarquias de objetos dinâmicos. Em branco, as OBBs que envolvem, de maneira mais compacta, esses objetos.

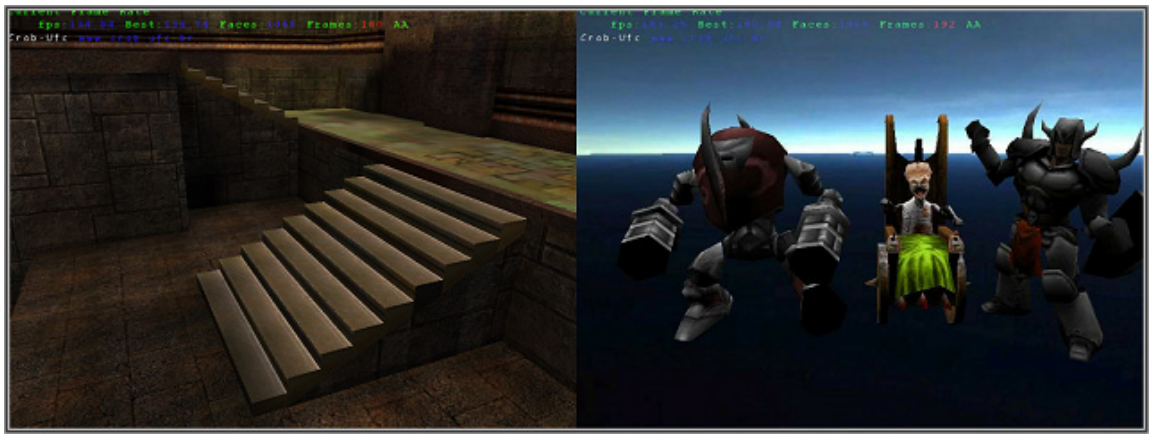


Figura 6.14: Outras duas aplicações multiplataforma construídas com o motor CGE. Navegação um ambiente virtual e pré-visualização de avatares animados.

Nessas aplicações, foi realizada a navegação nos mundos virtuais apresentados anteriormente para avaliar o desempenho e a qualidade audiovisual desses mundos mediante a utilização do motor CGE. Nos testes realizados, obtiveram-se excelentes taxas de quadros por segundo para cada um desses mundos, como descrito pela Tabela X. Além disso, constatou-se que o motor CGE obtém praticamente o mesmo desempenho nas plataformas Windows e Linux.

Tabela 6.2 : Mundos virtuais carregados no motor CGE e resultados obtidos.

Mundo	**Tipo de Cenário**	**Polígonos**	**Gerenciador de Cena**	**Quadros por Segundo (Média)**
Hotel	fechado	6.045	*Octree*	84
Aeroporto	fechado	12.133	*Octree*	87
NPD-UFC	misto	13.000	*Octree*	62
Domo	aberto	37.500	*Octree*	67
Mapa Q3	fechado	9.269	BSP/PVS	118
Mapa WET	misto	87.118	BSP/PVS	102

Além da realização desses testes, a aplicação de navegação em mundos virtuais foi expandida para avaliar a flexibilidade do gerenciamento de cenas da seguinte maneira. Foram utilizados dois gerenciadores de cenas e quatro câmeras que são usadas para pintar essas cenas de maneira independente através de quatro *viewports*. Nessa aplicação, ilustrada pela Figura 6.15, obteve-se uma taxa de refrescamento que varia de entre 45 e 67 quadros por segundo.

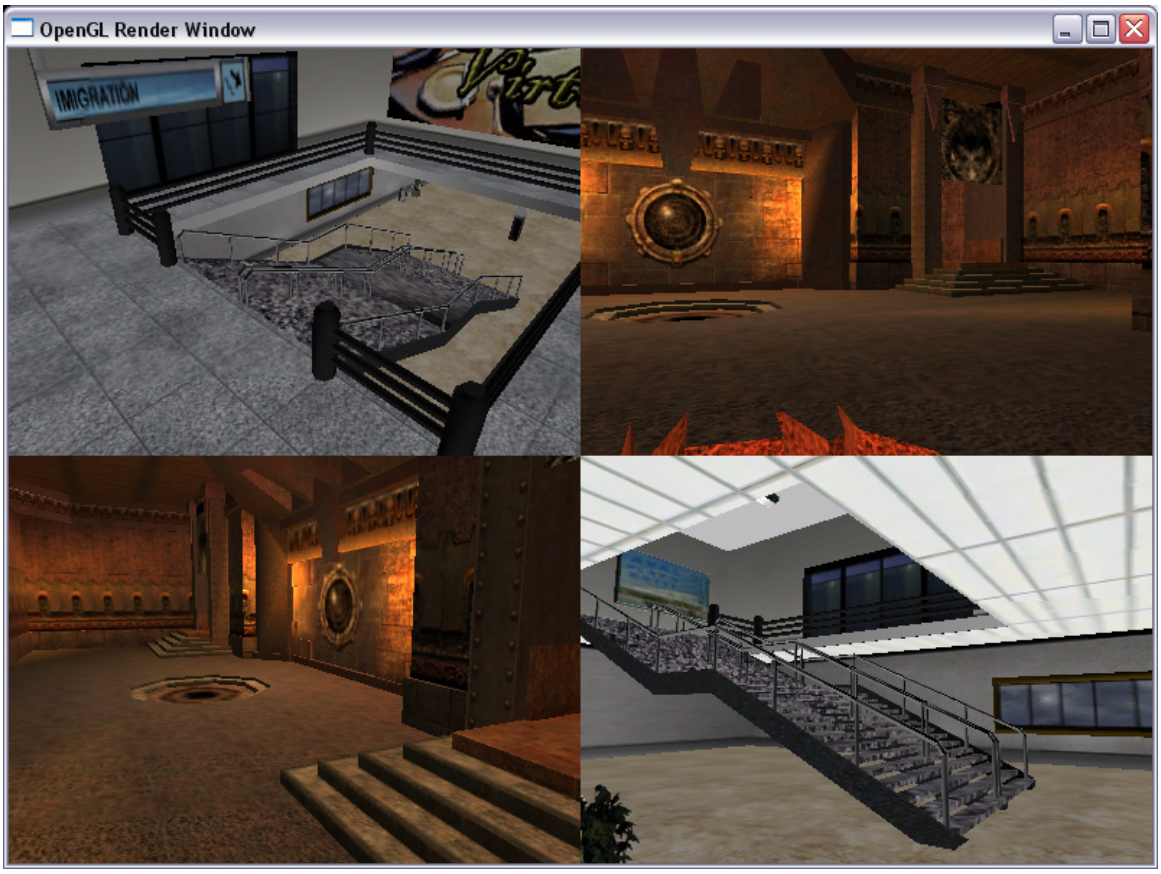


Figura 6.15: Utilização simultânea de dois gerenciadodores de cena para visualização em quatro *viewports*.

Por fim, foi desenvolvida uma aplicação de teste com o objetivo de validar o subsistema de scripts do motor CGE. Para tanto, foram utilizados três scripts Lua, que são escritos de maneira semelhante ao exemplo ilustrado na Tabela 6.3. Esses scripts são utilizados com as seguintes finalidades:

- Um script de configuração, responsável pelo carregamento de *plugins*, inclusão de diretórios no sistema de arquivos virtuais, carregamento de cenário e pela configuração da janela do sistema utilizada para exibir as cenas renderizadas;
- Um script para o carregamento e posicionamento de modelos geométricos no mundo virtual, além da definição das escalas, materiais e seqüências de animação utilizadas por esses modelos; e
- Um script pré-compilado a partir de um trecho de código Lua armazenado na memória, que é responsável por atribuir comportamento a um objeto dinâmico.

Tabela 6.3 : Exemplo de script Lua utilizado pela aplicação de teste.

```
-- Carregamento de plugins
loadPlugin("OpenGLRender.plugin")
loadPlugin("OpenALSound.plugin")
loadPlugin("OpcodeCollision.plugin")
loadPlugin("Md2Loader.plugin")
loadPlugin("OctreeSceneManager.plugin")

-- Inclusão de alguns arquivos ZIP no sistema de arquivos virtuais
addResourceDirectory("files/npd.zip")
addResourceDirectory("files/fortal.zip")
addResourceDirectory("files/praia.zip")

-- Criação de um material com textura
createBasicMaterial("yoshiMat", "yoshi.png" );

-- Criação de uma instância de Entity
yoshi = Entity("yoshi", "yoshi.md2" );
yoshi:setPosition( 50,0,0 Node.GLOBAL_SPACE)
yoshi:setMaterialName("yoshiMat")
yoshi:setAnimation("run")
```

6.4.2 Análise do Motor CGE

No decorrer dessa subseção, será apresentada a análise do motor gráfico CGE, que foi implementado como um estudo de caso da arquitetura CRAbGE. Essa análise é realizada de acordo com os critérios apresentados no Capítulo 4, que também foram utilizados para analisar os motores gráficos existentes.

Portabilidade. O código-fonte do motor CGE é portável entre as plataformas Windows e Linux. A utilização desse motor nessas plataformas baseia-se na recompilação do código-fonte do motor e dos *plugins* desenvolvidos. Além disso, durante a implementação do motor, foram utilizadas tecnologias que também podem ser utilizadas na plataforma Mac. Espera-se que o motor também seja compilado nessa plataforma sem problemas, no entanto, devido às restrições do projeto, não foi possível realizar testes de compilação na plataforma Mac.

Interface de Programação. Para desenvolver aplicações e extensões com o motor CGE, o programador utiliza uma interface orientada a objetos, que reusa vários padrões de projeto bastante conhecidos, o que confere maior facilidade de compreensão e uso ao motor gráfico. Além disso, essa interface de programação é bastante conveniente para a manutenção e para a expansão do motor gráfico como um todo.

Disponibilidade de Código-Fonte. O motor CGE é desenvolvido como um projeto de código aberto, podendo ser usado gratuitamente sob os termos da licença LGPL, que é adequada às necessidades dos desenvolvedores de aplicações comerciais e de pesquisadores. Além disso, essa licença permite a livre reutilização de partes do código-fonte do motor, caso algum desenvolvedor ache essa opção mais conveniente.

Capacidade de Configuração. O motor CGE oferece uma grande capacidade de configuração, pois permite que cada um dos subsistemas do motor gráfico seja configurado separadamente. O motor CGE também disponibiliza um poderoso subsistema de script que pode ser utilizado para realizar configurações mais complexas tanto do motor gráfico quanto das aplicações construídas com esse motor. Além disso, esse motor possui mecanismos para a realização de customizações em praticamente todos os seus módulos.

Suporte a *Plugins*. O suporte ao carregamento de extensões através de *plugins* é uma funcionalidade básica do motor CGE, visto que a arquitetura CRAbGE baseia-se no modelo microkernel para especificar o motor gráfico através de módulos fracamente acoplados que são gerenciados por um módulo principal.

Renderização em Tempo-Real. O subsistema de renderização do motor CGE é independente de API gráfica, oferecendo um *plugin* padrão que utiliza OpenGL para a renderização em baixo nível. Além disso, esse subsistema abstrai-se de como os objetos potencialmente visíveis em uma cena são reordenados para a pintura, permitindo que o usuário do motor realize customizações nesse processo.

O subsistema de renderização do motor CGE também se abstrai das técnicas utilizadas para acelerar a pintura de uma cena, suportando a utilização de técnicas customizadas pelo usuário do motor. Esse subsistema integra diversas técnicas para acelerar a pintura dos objetos dinâmicos, como *Hierarchical Frustum Culling*, LOD estático, *billboards* e sistemas de partículas, dentre outras. Além disso, esse subsistema implementa duas técnicas padrões para acelerar a pintura dos objetos estáticos. A primeira dessas técnicas utiliza uma *octree* para representar a geometria estática da cena, sendo mais adequada à pintura de cenários abertos. A outra técnica utiliza uma combinação das técnicas de PVS com árvores BSP, que é mais adequada à renderização de cenários fechados;

Detecção de Colisões. No motor gráfico CGE, o subsistema de detecção de colisões oferece testes para o cálculo de interseção tanto entre pares de volumes convexos (AABB, OBB, esfera e cápsula) quanto entre um volume convexo um segmento de reta ou raio. Além disso, esse subsistema se abstrai das técnicas utilizadas para detectar a interseção entre modelos geométricos deformáveis que são posicionados num ambiente virtual através de translações, rotações e escalas. Atualmente, esse subsistema conta com uma implementação através da biblioteca OPCODE, que foi modificada para suportar eficientemente a aplicação de escalas a esses modelos geométricos.

Som Espacial. Através de um modelo sonoro abrangente, baseado nos padrões abertos OpenAL e EAX, o subsistema de som espacial do motor CGE oferece suporte aos efeitos de espacialização, atenuação e Doppler. Além disso, esse subsistema suporta a utilização de efeitos de reverberação e oclusão EAX, além de contar com um mecanismo expansível para o carregamento de sons pré-amostrados a partir de arquivos.

Linguagens de Script. O subsistema de script do motor CGE, que é independente de linguagem, oferece suporte à execução de trechos de código em memória ou a partir de arquivos. Esse subsistema também permite armazenar scripts num formato pré-compilado, o que permite interpretar, de uma maneira mais eficiente, scripts utilizados freqüentemente por uma aplicação ou pelo motor gráfico. Além disso, esse subsistema permite acessar e modificar o valor de variáveis residentes no interpretador de scripts, uma tarefa bastante comum quando se deseja utilizar scripts para a configuração de um sistema.

Carregamento de Mídias. O motor CGE possui um mecanismo eficiente e flexível para o carregamento de mídias, que permite utilizar um sistema de arquivos virtuais para localizar as mídias usadas para descrever um mundo virtual. Esse motor suporta a utilização de diversos formatos de imagem através da biblioteca DevIL, sendo capaz de carregar cenários tridimensionais a partir de arquivos no formato 3DS e modelos animados a partir de arquivos no formato MD2. Além disso, esse motor permite que o desenvolvedor utilize outros formatos de arquivos através do registro de carregadores adequados.

Qualidade Audiovisual. Atualmente, o motor CGE suporta a utilização de praticamente todos os recursos da pipeline *fixed function*, permitindo a criação de efeitos visuais através da pintura em vários passos, assegurando uma qualidade visual muito boa às cenas renderizadas. Além disso, o motor CGE suporta a simulação de áudio espacial através de

um modelo sonoro de qualidade que é implementado, atualmente, através das APIs OpenAL e FMOD. Assim, é possível simular os efeitos Doppler, espacialização e atenuação, além de reverberações.

Desempenho. O motor integra diversas técnicas para acelerar a pintura das cenas tridimensionais, oferecendo um excelente desempenho tanto em cenários abertos quanto em cenários fechados. Além disso, o motor gráfico suporta a utilização de técnicas adicionais, que podem ser registradas através de *plugins* ou pela camada de aplicação, permitindo que o motor se adapte às necessidades de uma aplicação específica.

Documentação. A documentação do motor CGE é gerada automaticamente pela ferramenta Doxygen, de maneira que cada pacote, classe e função presente no motor gráfico são documentados através de *tags* que são inseridos como comentários no código-fonte do motor. A utilização dessa ferramenta facilita manter o código-fonte e sua respectiva documentação sempre atualizados, e, além disso, oferece ao usuário uma documentação *in loco* em cada arquivo de interface do motor. Por outro lado, essa ferramenta permite publicar a documentação do motor como páginas da *Web*.

Complexidade de Utilização. O motor CGE apresenta uma complexidade de uso aceitável, pois um desenvolvedor só precisa se deparar com os problemas que lhe interessam. Uma vez que o programador de aplicações aprende a inicializar os componentes do motor gráfico, ele pode utilizar a camada de alto nível para a gerência de cenas e abstrair-se de como as funcionalidades do motor são implementadas nos *plugins* que utiliza.

Além disso, a decomposição do motor em pacotes de classes é conveniente para a visualização das classes pelo desenvolvedor. Isso porque as classes são organizadas de acordo com suas funcionalidades, facilitando a compreensão de como cada grupo de classes é estruturado para resolver um conjunto de problemas.

6.5 Considerações Finais

A arquitetura CRAbGE foi desenvolvida para contornar as limitações presentes na maioria dos motores gráficos, suportando a construção de motores portáveis com grandes capacidades de configuração e customização que podem ser utilizados tanto para a construção de jogos quanto no desenvolvimento de sistemas de Realidade Virtual.

Nesse capítulo, foram apresentados os principais aspectos referentes à implementação do motor gráfico CGE, desenvolvido de acordo com a arquitetura CRAbGE. O estudo de caso realizado no decorrer desse capítulo demonstrou que essa arquitetura é viável na prática, permitindo construir um motor gráfico que é de fato portável e possui mecanismos de configuração e de customização que lhe conferem a capacidade de adaptar-se às necessidades de uma situação específica, permitindo sua utilização em aplicações de propósito geral.

No próximo capítulo, serão sumarizadas as principais contribuições da arquitetura CRAbGE, bem como as principais conclusões inferidas acerca dessa arquitetura. Além disso, serão apresentadas algumas sugestões para trabalhos futuros.

Capítulo 7

Conclusões

Neste trabalho, realizou-se um estudo sobre a utilização de motores gráficos no desenvolvimento dos principais aspectos de uma aplicação de Realidade Virtual, apresentando-se as principais soluções para a construção desse tipo de aplicação.

Dois tipos de soluções comumente utilizadas foram discutidos: as soluções de baixo nível e as soluções de alto nível. As soluções de baixo nível, como, por exemplo, técnicas otimizadas e APIs de baixo nível, são utilizadas em situações específicas para conferir eficiência à aplicação desenvolvida. Uma característica desse tipo de solução é o grande controle que a camada de aplicação exerce sobre os diversos aspectos da aplicação. Nessa abordagem, a aplicação geralmente é construída através da integração de técnicas e APIs de baixo nível com a finalidade precípua de atender suas necessidades específicas. Isso adiciona complexidade à camada de aplicação, tornando esse tipo de solução não-recomendável ao desenvolvedor inexperiente ou ao desenvolvimento de projetos de curta duração.

As soluções de alto nível, por sua vez, apresentadas pelos *frameworks* de RV, encapsulam a complexidade de implementação e integração das soluções de baixo nível, conferindo grande modularidade às aplicações desenvolvidas. Entretanto, esse tipo de componente geralmente impõe limitações ao controle que a camada de aplicação exerce sobre os mundos virtuais, visto que, a aplicação apenas define os mundos virtuais que o *framework* deve gerenciar.

Os motores gráficos utilizados no desenvolvimento de aplicações de RV apresentam soluções otimizadas e de qualidade através de uma interface de alto nível, encapsulando técnicas otimizadas e APIs de baixo nível. A utilização de um motor gráfico, diferentemente do que ocorre com o uso de um *framework* de RV, não restringe o controle que a camada de aplicação exerce sobre os mundos virtuais utilizados numa aplicação. Além disso, um motor gráfico possui vários dos subsistemas essenciais ao desenvolvimento de uma aplicação de RV: renderização em tempo-real, detecção de colisões, som espacial e

linguagens de script; os quais o tornam uma alternativa atraente para esse tipo de desenvolvimento, principalmente em plataformas de *Desktop*.

Como visto no Capítulo 3, um motor gráfico pode ser dividido em três módulos principais: núcleo, subsistemas e carregamento de elementos do mundo virtual. Essa divisão estrutural foi utilizada para analisar, de maneira comparativa, os principais motores gráficos disponíveis na atualidade e que representam o estado-da-arte desse tipo de componente de *software*. O resultado dessa análise apontou para uma série de limitações relacionadas, principalmente, aos seguintes aspectos:

- Ausência de algum subsistema básico;
- Código-fonte fechado;
- Forte acoplamento a tecnologias, técnicas de baixo nível e plataformas específicas;
- Baixa capacidade de configuração, expansão e customização;
- Complexidade de utilização;
- Utilização de dispositivos de entrada não-convencionais;
- Desempenho insatisfatório; e
- Abordagem superficial do problema de detecção de colisões.

A partir dessa constatação, foi desenvolvida a arquitetura CRAbGE, que se baseia no modelo de sistemas *microkernel* para contornar essas limitações. Esse modelo consiste de um núcleo portável e de subsistemas independentes de tecnologias específicas que oferecem uma abordagem ampla para resolver os problemas inerentes aos aspectos básicos de uma aplicação de RV.

Além disso, essa arquitetura oferece, no motor gráfico, um mecanismo eficiente e expansível para o carregamento de elementos do mundo virtual. Essa arquitetura oferece suporte à utilização de dispositivos não-convencionais e utiliza *plugins* como o mecanismo natural de expansão. Além disso, a arquitetura CRAbGE prevê a evolução do motor gráfico, suportando a subseqüente inclusão de novos subsistemas e módulos no motor.

A arquitetura CRAbGE foi implementada no motor CGE em C++ padrão. Esse motor é de fato portável, apresentando as seguintes vantagens sobre os motores analisados através de uma arquitetura simples e elegante:

- Provê todos os subsistemas básicos a uma aplicação de RV;

- Possui arquitetura e código-fonte abertos;
- Abstrai-se de tecnologias, técnicas de baixo nível e plataformas;
- Possui grande capacidade de configuração, de expansão e de customização;
- Apresenta uma grande facilidade de uso;
- Suporta a utilização de dispositivos de entrada não-convencionais;
- Considera os principais fatores determinantes da eficiência e da qualidade audiovisual, permitindo que o motor se adapte às necessidades de uma aplicação específica; e
- Suporta a detecção de colisões entre modelos geométricos deformáveis.

O código-fonte desse motor foi compilado, sem quaisquer modificações, nas plataformas Windows e Linux. Além disso, foram realizados testes nessas plataformas através de um conjunto de aplicações para comprovar a eficiência e viabilidade da arquitetura no desenvolvimento de uma aplicação real.

Da mesma forma que o motor CGE, essas aplicações de teste foram compiladas sem modificações nessas plataformas, demonstrando a eficiência e a facilidade de uso do motor desenvolvido. Além disso, através de sua capacidade de customização, o motor pode ser utilizado em aplicações de propósito geral, permitindo utilizar técnicas customizadas quando for necessário a uma aplicação específica.

Possíveis trabalhos futuros incluem:

1. A utilização de aplicações desenvolvidas com o motor CGE como cenários para a comparação de diferentes plataformas, tecnologias, APIs e técnicas de baixo nível.
2. A investigação da utilização de linhas de execução (*threads*) no motor CGE, já que, atualmente, diversos tipos de aplicações são concebidos para serem executados através de múltiplas linhas de execução.
3. A expansão do subsistema de detecção de colisões para lidar com situações mais complexas, como a detecção de auto-interseções através de técnicas contínuas em modelos articulados, deformáveis e quebráveis.
4. A expansão do subsistema de som espacial para que ofereça suporte à captura e transmissão de áudio em tempo-real.

5. O desenvolvimento de uma técnica de animação integrada com o subsistema de som espacial para a geração de áudio e *lip sync* a partir de um roteiro (ou legenda, como num filme) para a internacionalização dos ambientes de RV construídos com o motor gráfico, bem como para a utilização de sistemas de RV por parte dos portadores de deficiência.
6. O suporte à gravação e à reprodução dos acontecimentos no mundo virtual, para aplicações de treinamento, permitindo que o usuário realize auto-avaliações sobre o seu desempenho nas situações simuladas no ambiente virtual.
7. Inclusão de novos subsistemas padrões no motor gráfico, como, por exemplo:
   - Subsistema de computação acelerada pela GPU;
   - Subsistema de simulação da dinâmica de fluidos, corpos rígidos articulados ou deformáveis; e
   - Subsistema de comunicação em rede;
8. Expansão do motor para que ofereça maior suporte ao desenvolvimento de Ambientes Virtuais em Rede, incluindo novos subsistemas ou componentes encapsulando as seguintes funcionalidades:
   - Reprodução de vídeos no mundo virtual;
   - *Streaming* de áudio em rede;
   - Técnicas de *pathfinding* e *pathplanning*
   - Técnicas de *dead-reckoning*; e
   - Técnicas baseadas em setores para suportar eficientemente a renderização, comunicação e simulação de cenas contendo um grande número de visitantes.

Bibliografia

Activision (2004). http://www.activision.com. Visitado em Janeiro de 2004.

Airey, J.; Rohlf, J. & Brooks JR., F. (1990). *Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments*. In Symposium on Interactive 3D Graphics, vol. 24, no. 2, pp. 41-50, March 1990.

Alias (2004). *Maya 6*. Disponível em http://www.alias.com/eng/products-services/maya/index.shtml. Visitado em Outubro de 2004.

Almendra, C. C.; Vidal, C. A.; Serra, A. B.; Leite Jr, A. J. M. & Santos, E. M. (2002). *Integração de Aplicações de Realidade Virtual em Rede a Ambientes de Aprendizado na Web – Um Estudo de Caso*. Em Proceedings of the 5th SVR – Symposium on Virtual Reality, Fortaleza, Brasil, Outubro, p. 65-77.

Arsenault, L.; Kelso, J.; Kriz, R. & Das-Neves, F. (2001). *DIVERSE: a Software Toolkit to Integrate Distributed Simulations with Heterogeneous Virtual Environments*. Technical Report TR-01-10, Computer Science, Virginia Tech.

BASS (2004). BASS Sound System (Version 2.1). Disponível em http://www.un4seen.com/. Visitado em Outubro de 2004.

Bastos, T. A.; Silva, R. J. M.; Raposo, A. B. & Gattass, M. (2004). *ViRAL: Um Framework para o Desenvolvimento de Aplicações de Realidade Virtual*. Em Proceedings of the 7th SVR – Symposium on Virtual Reality, São Paulo, Brasil, Outubro, p. 51-62.

Berger, A. (2002). *3D Programming for Windows*. Disponível em http://www.3dstate.com. Visitado em Novembro de 2002.

Bernardes-Junior, J. L.; Bianchini, R.; Cuzziol, M.; Jacober, E.; Nakamura, R. & Tori, R. (2004). *Jogos Eletrônicos e Realidade Virtual*. Realidade Virtual: Conceitos e Tendências, Cláudio Kirner & Romero Tori, Editores. São Paulo, Brasil, Outubro, p. 159-176, ISBN -85-904873-1-8.

Bierbaun, A.; Just, C.; Hartling, P.; Meinert, K.; Baker, A. & Cruz-Neira, C. (2001). *VR Juggler: A Virtual Platform for Virtual Reality Application Development*. IEEE VR 2001 Proceedings, Yokohama, Japan, March. Disponível em http://www.vrjuggler.org. Visitado em Junho de 2004.

Bison (2004). *Bison GNU Project at Free Software Foundation*. Disponível em http://www.gnu.org/software/bison/bison.html. Visitado em Outubro de 2004.

Brooks-Jr., F.P. (1999). *What's Real About Virtual Reality?*, in IEEE Computer Graphics and Applications, 19, 6:16-27.

Cal3D (2004). *Cal3D Character Animation Library*. Disponível em http://cal3d.sourceforge.net. Visitado em Julho de 2004.

Chumbalum Soft (2004). Milkshape 3D. Disponível em http://www.milkshape3d.com/. Outubro de 2004.

ColDet (2004). *ColDet - Free 3D Collision Detection Library (Version: 1.1)*. Disponível em http://photoneffect.com/coldet. Visitado em Julho de 2004.

Conrad, S. (2002). *Local Haptics – Haptic Rendering using Intermediate Local Models*. Em Proceedings of the 5th SVR – Symposium on Virtual Reality, Fortaleza, Brasil, Outubro, p. 308-316.

Creative Labs (2004). *EAX Quality Assurance: A Guide to Testing Positional Audio and EAX*. Disponível em http://developer.creative.com/articles/article.asp?cat=1&sbcat=43&top=68&aid=94&file=EAX_QA.pdf. Visitado em Outubro de 2004.

CrystalSpace (2004). *CrystalSpace User's Manual (Stable Release 0.98)*. Disponível em http://crystal.sourceforge.net/docs/online/manual-0.98/cs_toc.html. Visitado em Julho de 2004.

Dawson, D. (2002). *GDC 2002: Game Scripting in Python*. Gamasutra Features. Disponível em http://www.gamasutra.com/features/20020821/dawson_01.htm. Visitado em Julho de 2004.

Dawson, B. (2004). *Comparing Floating Point Numbers*. Disponível em http://www.cygnus-software.com/papers/comparingfloats/comparingfloats.htm. Visitado em Julho de 2004.

Decoret, X., Silliom, F., Schaufler, G. & Dorsey, J. (1999). *Multi-layered Impostors for Accelerated Rendering*. In Eurographics'99, Computer Graphics Forum, Volume 18, Issue 3.

DevMaster (2004). *3D Engines Database*. Disponível em http://www.devmaster.net/engines/. Visitado em Outubro de 2004.

DirectX (2004). *Microsoft DirectX® SDK 9.0*. Disponível em http://www.microsoft.com/windows/directx/default.aspx?url=/windows/directx/downloads/default.htm. Visitado em Julho de 2004.

Discreet (2004). *3D Studio Max 7*. Disponível em http://www4.discreet.com/3dsmax/. Visitado em Outubro de 2004.

Duchaineau, M.; Wolinsky, M.; Sigeti, D. F.; Miller, M. C.; Aldrich, C. & Mineev-Weinstein, M. B. (1997) *ROAMing Terrain: Real-Time Optimally Adapting Meshes*. In Proceedings of the ACM Symposium on Volume Visualization'97, pp. 81-88, October 1997.

Dyarr, T. (2002). *A Virtual Environment (VE) System for Retraining Movements*, http://www.blender.org/modules/bc2002/ThomasDyar/Talk.ppt. Visitado em Novembro de 2002.

Ehmann, S. A. (2001). *S W I F T + + Speedy Walking via Improved Feature Testing for Non-Convex Objects (Version 1.0 Application Manual)*. Department of Computer Science, University of North Carolina, Chapel Hill, NC. Disponível em http://www.cs.unc.edu/~geom/SWIFT++/.Visitado em Julho de 2004.

Elias, N. C. (2002). *Análise de Técnicas de Produção de Jogos de Computador para Desenvolvimento de um Software de Ensino de Conceitos de Computação Gráfica*. Dissertação de mestrado, Universidade Federal de São Carlos/SP.

Epic Games (2004). *Unreal Engine*. Disponível em http://www.unrealtechnology.com/html/technology/ue30.shtml. Visitado em Julho de 2004.

Figueiredo, M.; Marcelino, L. & Fernando, T. (2002). *A Survey on Collision Detection Techniques for Virtual Environments*. Em Proceedings of the 5th SVR – Symposium on Virtual Reality, Fortaleza, Brasil, Outubro, p. 285-307.

Firelight Technologies (2004). *FMOD API Reference (Version 3.74)*. Disponível em http://www.fmod.org/docs/. Visitado em Outubro de 2004.

Fisher, J. B. (2001). *Using Virtual Reality to Train Air Traffic Controllers*. Disponível em http://www.tss.swri.edu/pub/pdf/2001iats_atcvr.pdf. Visitado em Fevereiro de 2003.

Flex (2004). *Flex GNU Project at Free Software Foundation*. Disponível em http://www.gnu.org/software/flex/flex.html. Visitado em Outubro de 2004.

Fly3D (2004). *Fly3D SDK 2.0 Documentation*. Disponível em http://www.fly3d.com.br/download/Fly3D.chm. Visitado em Julho de 2004.

Funkhouster, T. & Séquin, C. (1993) *Adaptive Display Algorithm for Interactive Frame Rates during Visualization of Complex Virtual Environments*. In SIGGRAPH'93 Proceedings, pp. 247–254.

Gamma, E.; Helm, R.; Johnson, R. & Vlissides, J. (2000). *Padrões de Projeto – Soluções Reutilizáveis de Software Orientado a Objetos*. Porto Alegre: Bookman. ISBN 85-7307-610-0.

Gamma Team (2004). *Collision Detection / Proximity Query Packages*. Disponível em http://www.cs.unc.edu/~geom/collide/index.shtml. Visitado em Junho de 2004.

Garcia, F. L. S.; Camargo, F. D. & Lorenzato, L. (2002). *Virtual TrainingPit: Um Sistema de Treinamento Virtual de Pilotos de Aeronaves*. Disponível em http://www.lrv.eps.ufsc.br/recursos/artigos/VRtrainingpit.pdf. Visitado em Fevereiro de 2003.

Garland, M. & Heckbert, P. (1995). Fast Polygonal Approximation of Terrains and Height Fields. Technical Report CMU-CS-95-181, School of Computer Science, Carnegie Mellon University. Disponível em http://graphics.cs.uiuc.edu/~garland/papers.html. Visitado em Janeiro de 2004.

Genesis3D (2004). *Genesis3D Game Engine*. Disponível em http://www.genesis3d.com. Visitado em Julho de 2004.

Gilbert, E. G.; Johnson, D.W. & Keerthi, S. S. (1988). *A Fast Procedure for Computing the Distance Between Complex Objects in Three-Dimensional Space*. In IEEE Journal of Robotics and Automation, 4(2):193–203, 1988.

Govindaraju, N. K.; Redon, S.; Lin, M. C. & Manocha, D. (2003). *CULLIDE: Interactive Collision Detection between Complex Models in Large Environments using Graphics hardware*. In Proceedings of ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware, pages 25-32, 2003.

Govindaraju, N. K.; Lin, M. C. & Manocha, D. (2004). *Fast and Reliable Collision Culling using Graphics Hardware*. Uiversity of North Carolina at Chapel Hill Technical Report, January, 2004. Disponível em http://gamma.cs.unc.edu/RCULLIDE/far.pdf. Visitado em Junho de 2004.

Gottschalk, S.; Lin, M. C. & Manocha, D. (1996). *OBB-Tree: A Hierarchical Structure for Rapid Interference Detection*. Proceedings of ACM Siggraph'96, pp. 171-180. Disponível em ftp://ftp.cs.unc.edu/pub/users/manocha/PAPERS/COLLISION/obb.ps.gz. Visitado em Julho de 2004.

Gottschalk, S. (1990). *Separating Axis Theorem*. Technical Report. TR96-024, UNC Chapel Hill, 1990.

Greene, N.; Kass, M. & Miller, G. (1993). *Hierarchical Z-Buffer Visibility*. In SIGGRAPH''93 Proceedings, pp. 231-238, August 1993.

Hart, E. (2004). *OpenGL Performance Tuning*. In Game Developer Conference, GDC'2004. San Jose, CA, 22-26 March, 2004. Disponível em http://www.ati.com/developer/gdc/PerformanceTuning.pdf. Visitado em Julho de 2004.

Hofmann, J. S. (2000) *A Survey of Real-Time Rendering Algorithms*. Disponível em http://wbwst.tripod.com/download/survey_of_real-time_rendering_algorithms.pdf. Visitado em Outubro de 2003.

Hollasch, S. (2004). *IEEE Standard 754 Floating Point Numbers*. Disponível em http://stevehollasch.com/cgindex/coding/ieeefloat.html. Visitado em Novembro de 2004.

Hoppe, H. (1996). *Progressive Meshes*. In SIGGRAPH'96 Proceedings, pp. 99-108, August 1996.

Hudson, T.; Manocha, D.; Cohen, J.; Lin, M.; Hoff, K. & Zhang, H. (1997). *Accelerated Occlusion Culling using Shadow Frusta*. In Thirteenth ACM Symposium on Computational Geometry, Nice, France, pp. 1-9, June 1997.

Id Software (2004). *Quake III Engine*. Disponível em http://www.idsoftware.com/business/techdownloads/. Visitado em Janeiro de 2004.

Ierusalimschy, R.; Figueiredo, L. H. & Celes, W. (2003). *Lua 5.0 Reference Manual*. Disponível em http://www.lua.org/ftp/refman-5.0.pdf. Visitado em Outubro de 2004.

Kessenich, J.; Baldwin, D. & Rost, Randi (2004). *The OpenGL® Shading Language*. Disponível em http://oss.sgi.com/projects/ogl-sample/registry/ARB/GLSLangSpec.Full.1.10.59.pdf. Visitado em Agosto de 2004.

Kim, Y. J.; Otaduy, M. A.; Lin, M. C. & Manocha, D. (2002). *Fast Penetration Depth Computation for Physically-Based Animation*. In Proceedings of the 2002 ACM SIGGRAPH, pp. 23-31, ACM Press, ISBN 1-58113-573-4.

Klosowski, J.; Held, M. & Mitchell, J. (1998). *QuickCD Software Library for Efficient Collision Detection*. Disponível em http://www.ams.sunysb.edu/~jklosow/quickcd/QuickCD.html. Visitado em Março de 2003.

Kufmann, H. (2002). *Construct3D: An Augmented Reality Application for Mathematics and Geometry Education*. Disponível em http://www.ims.tuwien.ac.at/media/documents/publications/C3D_Video_Description-Final.pdf. Visitado em Abril de 2003.

Leite-Júnior, A. J. M.; Vidal, C. A.; Almendra, C. C.; Santos, E. M.; Gomes, H. O. O. & Mendonça-Júnior, G. M. (2002). *Um Ambiente Virtual Compartilhado Voltado para Entretenimento*. Em Proceedings of the 5th SVR – Symposium on Virtual Reality, Fortaleza, Brasil, Outubro, p. 138-149.

Lin, M. C. & Canny, J. F. (1991). *A Fast Algorithm for Incremental Distance Computation*. In Proceedings of the IEEE International Conference on Robotics and Automation, pages 1008–1014, 1991.

Lindstrom, P.; Koller, D.; Ribarsky, W.; Hodges, L. F.; Faust, N. & Turner, G. A. (1996). *Real-Time Continuous Level of Detail Rendering of Height Fields*. In SIGGRAPH'96 Proceedings, pp. 109-118, August 1996.

Loki Software (2000). *OpenAL Specification and Reference*. Disponível em http://www.openal.org/openal_webstf/specs/oalspecs-specs.pdf. Visitado em Junho de 2003.

Machado, L. S. (2003). *Pesquisa e Desenvolvimento de Sistemas de Realidade Virtual para Treinamento em Oncologia Pediátrica*. Disponível em http://www.lilianesm.hpg.ig.com.br/rvmed_p.html. Visitado em Março de 2003.

Magic Software (2004). *Wild Magic 2.5*. Disponível em http://www.magic-software.com/SourceCode.html. Visitado em Agosto de 2004.

Maia, J. G. R.; Cavalcante-Neto, J. B. & Vidal, C. A. (2003). *CRAbGE: Um Motor Gráfico Customizável, Expansível e Portável Para Aplicações de Realidade Virtual*. Em

Proceedings of the 6th SVR – Symposium on Virtual Reality, Ribeirão Preto, Brasil, Outubro, p. 03-14.

Maia, J. G. R.; Cavalcante-Neto, J. B.; Vidal, C. A. & Gomes, H. O. O. (2004). *CRAbRender: Um Sistema de Renderização Para Aplicações de RV*. Em Proceedings of the 7th SVR – Symposium on Virtual Reality, São Paulo, Brasil, Outubro, p. 380-382, ISBN 85-7651-006-5.

Mark, W. R.; Glanville, R. S.; Akeley, K. & Kilgard, M. J. (2003). *Cg: A System for Programming Graphics Hardware in a C-like Language*. In Proceedings of SIGGRAPH'2003, San Diego, California, USA, pp. 896-907, July 2003.

Möller, T. (1997). A Fast Triangle-Triangle Intersection Test. In Journal of Graphics Tools, 2(2), pp. 25-30, 1997.

Möller, T. & Trubore, B. (1997). *Fast, Minimum Storage Ray-Triangle Intersection*. In Journal of Graphics Tools, 2(1):21-28, 1997.

NewTek (2004). *LightWave 3D®*. Disponível em http://www.newtek.com/products/lightwave/product/index.html. Visitado em Outubro de 2004.

Ogg Vorbis (2004). *Ogg Vorbis Áudio Compression Format*. Disponível em http://www.vorbis.com/.Visitado em Junho de 2004.

OGRE3D (2004). *Object-Oriented Graphics Rendering Engine*. Disponível em http://ogre.sourceforge.net/. Visitado em Junho de 2004.

ODE (2004). *Open Dynamics Engine (Version 0.5)*. Disponível em http://www.ode.org. Visitado em Julho de 2004.

OpenGL Performer (2004). *OpenGL Performer Programmer's Guide*. Disponível em http://www.cineca.it/manuali/Performer/ProgGuide24/html/. Visitado em Julho de 2004.

Open Inventor (2004). *Open Inventor*. Disponível em http://oss.sgi.com/projects/inventor/. Visistado em Julho de 2004.

OpenSceneGraph (2004). *OpenSceneGraph (Version 0.9.8)*. Disponível em http://www.openscenegraph.org. Visitado em Dezembro de 2004.

OpenSG (2004). *OpenSG (Version 1.4.0)*. Disponível em http://www.opensg.org. Visitado em Novembro de 2004.

OpenSteer (2004). *OpenSteer Steering Behaviors for Autonomous Characters*. Disponível em http://opensteer.sourceforge.net. Visitado em Agosto de 2004.

Pinho, M. S. (2002). *SmallVR Uma Ferramenta Orientada a Objetos para o Desenvolvimento de Aplicações de Realidade Virtual*. Em Proceedings of the 5th SVR – Symposium on Virtual Reality, Fortaleza, Brasil, Outubro, p. 329-340.

Ponder, M.; Papagiannakis G.; Molet, T.; Magnenat-Thalmann, N. & Thalmann D. (2003). *VHD++ Development Framework: Towards Extendible, Component Based VR/AR Simulation Engine Featuring Advanced Virtual Character Technologies*. In Computer Graphics International (CGI), Tokyo, Japan, July 2003, p.96.

Programmer's Toolbox (2004). *Bison++/Flex++*. Disponível em http://www.progtools.org/compilers/parserscan.html. Visitado em Outubro de 2004.

Python Software Foundation (2004). *Python 2.4 Documentation*. Disponível em http://www.python.org/doc/2.4/. Visitado em Dezembro de 2004.

RAD Game Tools (2004). *Miles Sound System*. Disponível em http://www.radgametools.com/miles.htm. Visitado em Outubro de 2004.

Rege, A. (2004). *Optimization for DirectX9 Graphics*. In Game Developer Conference, GDC'2004. San Jose, CA, 22-26 March, 2004. Disponível em http://download.nvidia.com/developer/presentations/GDC_2004/Dx9Optimization.pdf. Visitado em Setembro de 2004.

Reis, D. S.; Nedel, L. P.; Freitas, C. M. D. S. & Hübner F. J. (2004). *Uma Arquitetura para Simulação de Agentes Autônomos com Comportamento Social*. Em Proceedings of the 7th SVR – Symposium on Virtual Reality, São Paulo, Brasil, Outubro, p. 39-50.

RenderWare (2004). *RenderWare Graphics 3.7*. Disponível em http://www.renderware.com/ graphics.asp. Visitado em Julho de 2004.

Rodrigues, M. A. F.; Maia, J. G. R.; Mendonça, N. C. & Chaves, F. E. (2002). *Um Sistema Interativo Distribuído para Simulação de Procedimentos Cirúrgicos*. Em Proceedings of the 5th SVR – Symposium on Virtual Reality, Fortaleza, Brasil, Outubro, p. 179-190.

Rogers, D. F. & Adams, J. A. (1990). *Mathematical Elements for Computer Graphics, 2nd Edition*. WCB/McGraw-Hill, 1990, ISBN 0-07-053529-9.

Schaufler, G. (1995). *Dynamically Generated Impostors*. In GI Workshop on "Modeling - Virtual Worlds - Distributed Graphics", D.W. Fellner, ed., Infix Verlag, pp. 129-135, November 1995.

Schaufler, G. (1997). *Nailboards: A Rendering Primitive for Image Caching in Dynamic Scenes*. In Eurographics Rendering Workshop 1997, Budapest, Hungary, pp. 151-162, September 1997.

Sébastien, D. (2002). *Collision Detection Libraries*. Disponível em http://horizons.free.fr/eng/tech/rv_dossiers/lib_detect_coll.htm. Visitado em Outubro de 2004.

Segal, M. & Akeley, K. (2004). *The OpenGL Graphics System: A Specification (Version 2.0 - October 22, 2004)*. Disponível em http://www.opengl.org/documentation/specs/version2.0/glspec20.pdf. Visitado em Outubro de 2004.

Shaw, C.; Green, M.; Liang, J. & sun, Y. (1993). *Decoupled Simulation in Virtual Reality with MR Toolkit*. ACM Transaction of Information Systems, 11(3);287-317, July 1993.

Sun Microsystems (2002). *The Java 3D™ API Specification (Version 1.3)*. Disponível em http://java.sun.com/products/java-media/3D/releases.html. Visitado em Outubro de 2004.

Szenberg, F.; Carvalho, P. C. P. & Gatass, M. (2002). *Juiz Virtual – Um Sistema para Análise de Lances de Futebol*. Em Proceedings of the 5th SVR – Symposium on Virtual Reality, Fortaleza, Brasil, Outubro, p. 193-202.

Tanembaum, A. S. (1999). *Sistemas Operacionais: Projeto e Implementação*. 2ª Edição, Editora Bookman, Setembro de 1999, ISBN 8573075309.

Terdiman, P. (2003). *OPCODE User Manual*. Disponível em http://www.codercorner.com/Opcode.htm. Visitado em Janeiro de 2004.

Teller, S., & Séquin, C. (1991). *Visibility Preprocessing For Interactive Walkthroughs*. In SIGGRAPH'91 Proceedings, pp. 61-69, July 1991.

The Sims (2004). *The Sims*. Disponível em http://thesims.ea.com/. Janeiro de 2004.

Thomas, D.; Fowler, C & Hunt, A. (2004). *Programming Ruby The Pragmatic Programmer's Guide, Second Edition*. Addison Wesley. ISBN 0-9745140-5-5. October 2004. Primeira edição disponível em http://www.rubycentral.com/book/index.html. Visitado em Outubro de 2004.

Tokamak (2004). *Tokamak Game Physics SDK*. Disponível em http://www.tokamakphysics.com. Visitado em Janeiro de 2004.

Tramberend, H. (1999). *Avango: A Distributed Virtual Reality Framework*. In Proceedings of the IEEE Virtual Reality '99, JW Marriott Hotel, Houston, Texas, USA, March 13-17, 1999.

Tsingos, N.; Gallo, E. & Drettakis, G. (2003). *Breaking the 64 Spatialized Sources Barrier*. Disponível em http://www.gamasutra.com/resource_guide/20030528/tsingos_pfv.htm. Visitado em Outubro de 2004.

Valve Corporation (2004). http://www.valvesoftware.com. Visitado em Julho de 2004.

Van Der Bergen, G. (1999A). *User's Guide to the SOLID Interference Detection Library (Version 2.0)*. Disponível em http://solid.sourceforge.net/solid2_toc.html. Visitado em Outubro de 2003.

Van Der Bergen, G. (1999B). *A Fast and Robust GJK Implementation for Collision Detection of Convex Objects*. In Journal of Graphics Tools, Volume 4, Issue 2, pp. 7-25, ISBN 1086-7651, March 1999. Disponível em http://www.win.tue.nl/~gino/solid/jgt98convex.pdf. Visitado em Outubro de 2003.

Van Der Bergen, G. (1997). *Efficient Collision Detection of Complex Deformable Models using AABB Trees*. In Journal of Graphics Tools, 2, 4, pp. 1-13.

Wireman, C. (2003). *Sound Formats and Their Uses in Games*. In GameDev Articles, 2003. Disponível em http://www.gamedev.net/reference/articles/article1902.asp. Visitado em Janeiro de 2004.

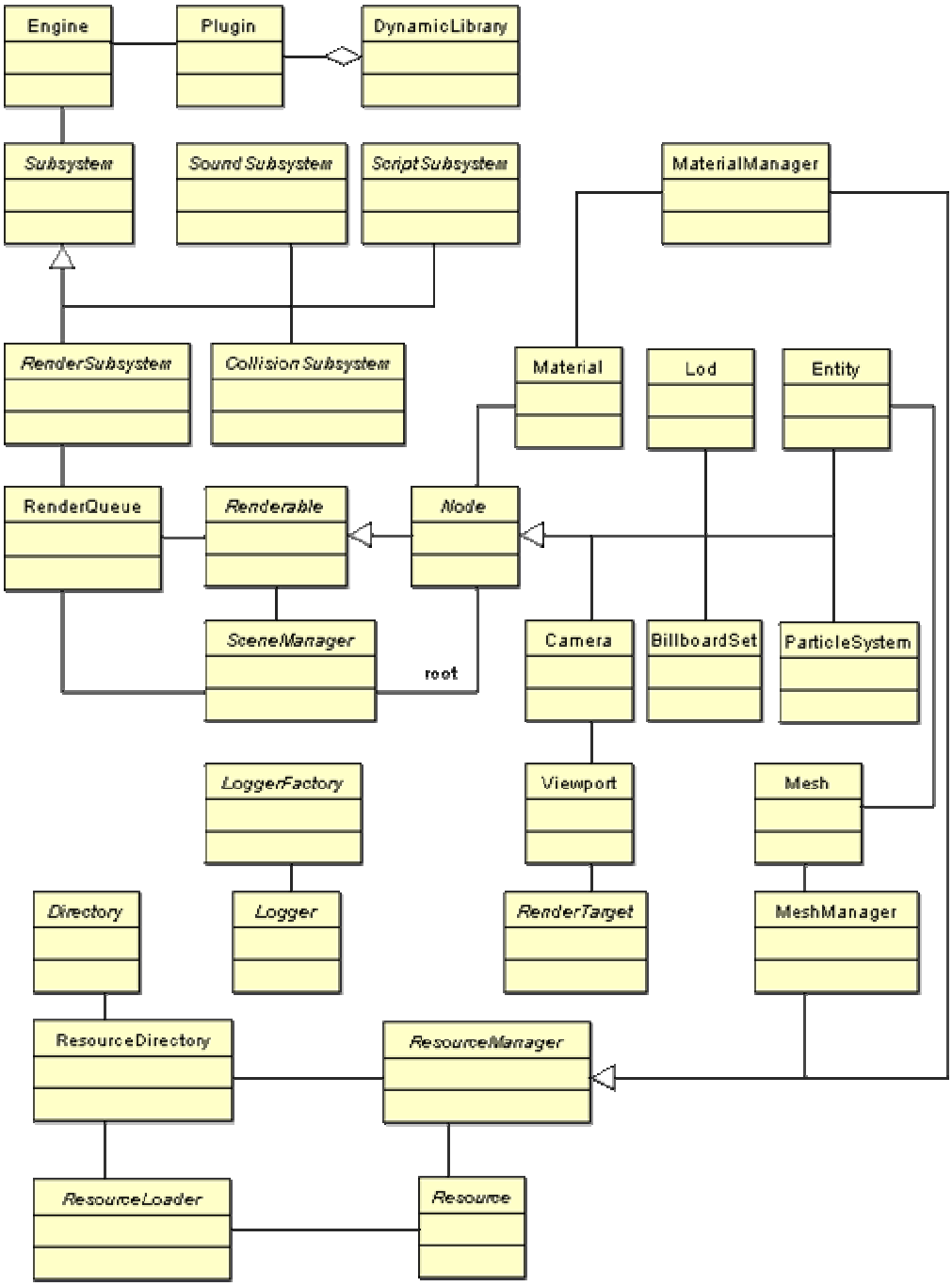
Wloka, M. M. (2004). *Optimizing the Graphics Pipeline*. In Annual Conference of the European Association for Computer Graphics, Eurographics'2004, 30 August to 3 September, 2004. Disponível em http://developer.nvidia.com/docs/IO/8230/GDC2003_PipelinePerformance.pdf. Visitado em Setembro de 2004.

Zhang, H.; Manocha, D.; Hudson, T. & Hoff III, K.E. (1997). *Visibility Culling using Hierarchical Occlusion Maps*. In SIGGRAPH'97 Proceedings, Los Angeles, Califórnia, pp. 77-88, August 1997.

Zlib (2004). *Zlib Licence*. Disponível em http://www.info-zip.org/pub/infozip/zlib/zlib_license.html. Visitado em Janeiro de 2004.

Apêndice A

Principais Classes do Motor CGE



Apêndice B

Distribuição do Código-Fonte do Motor CGE

O código-fonte do motor CGE é distribuído gratuitamente sob os termos da licença LGPL (GNU *Library General Public Licence*) e pode ser obtido através do seguinte repositório SVN: https://webmail.vdl.ufc.br/svn/projects/crabge/.

Alternativamente, é possível visualizar a versão mais recente do código-fonte e arquivos relacionados através do wiki oficial do motor CRAbGE. Esse wiki encontra-se no seguinte endereço da Web: https://webmail.vdl.ufc.br/trac/projects/crabge/.

A seguinte hierarquia de diretórios é utilizada para organizar o código-fonte do motor.

1. O subdiretório "cge" contém o código-fonte da biblioteca principal do motor;
2. O diretório "stdPlugins" contém o código-fonte dos *plugins* padrões que são distribuídos com o motor CGE;
3. O diretório "tests" contém o código-fonte de um conjunto de aplicações de teste;
4. O diretório "build" contém os projetos e scripts necessários à compilação do motor e dos *plugins*;
5. Por fim, o diretório "doc" contém a documentação do motor que é gerada automaticamente pela ferramenta Doxygen.

A compilação do motor CGE requer que apenas duas dependências sejam satisfeitas:

1. Tabelas de hash, presentes no GCC versão 3 ou superior. A falta dessas tabelas pode ser contornada com a instalação da biblioteca STLport, uma implementação otimizada da STL disponível no site http://www.stlport.org/.
2. A biblioteca DevIL (*Developer's Image Library*), também conhecida como OpenIL (Open Image Library). Essa biblioteca multiplataforma encontra-se disponível em diversas distribuições da plataforma Linux e seu código-fonte pode ser obtido no site http://openil.sourceforge.net/.

Apêndice C

Documentação de Sistemas Usando a Ferramenta Doxygen

A ferramenta Doxygen permite que o programador insira tags especiais em comentários padronizados que são inseridos antes da declaração de funções, tipos, classes, métodos, atributos e pacotes. Tais tags são utilizadas para introduzir textos explicativos que documentam bibliotecas e sistemas. Assim, uma ferramenta externa pode analisar o código-fonte de um sistema e gerar a documentação apropriada a partir desses comentários padronizados.

Uma das principais vantagens desse paradigma de documentação é a facilidade de atualização do código-fonte de um sistema com a sua documentação. Além disso, esses comentários padronizados funcionam como uma documentação in loco em cada arquivo do sistema.

A ferramenta Doxygen é capaz de gerar a documentação de um sistema em diversos formatos, como HTML, RTF e MAN, por exemplo. O formato HTML é particularmente interessante, pois permite publicar a documentação do sistema na Web. Além disso, é possível compilar toda a documentação num arquivo que, além de permitir navegar essa documentação, possui outras funcionalidades, como a localização de funções e métodos, por exemplo. A Figura C.1 ilustra a documentação compilada do motor CGE. A Figura C.2 ilustra a documentação publicada como um *website*.

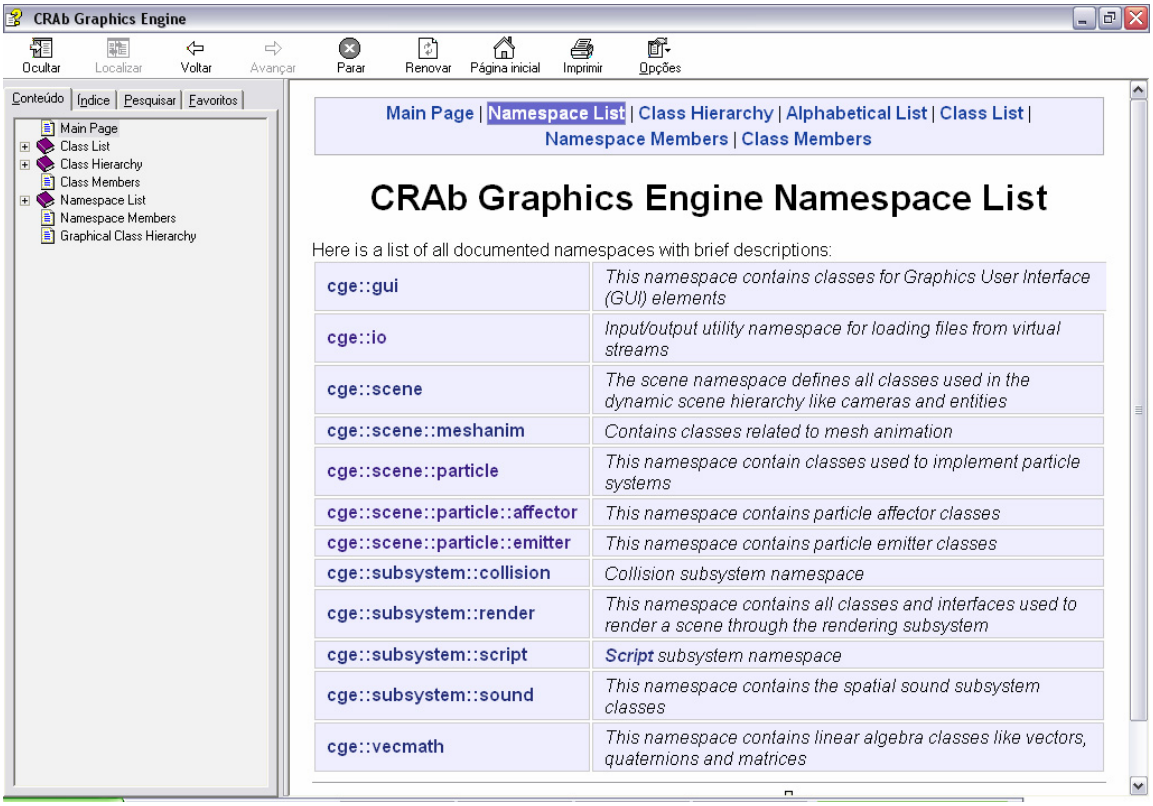


Figura C.1: Documentação do motor CGE compilada no formato CHM.

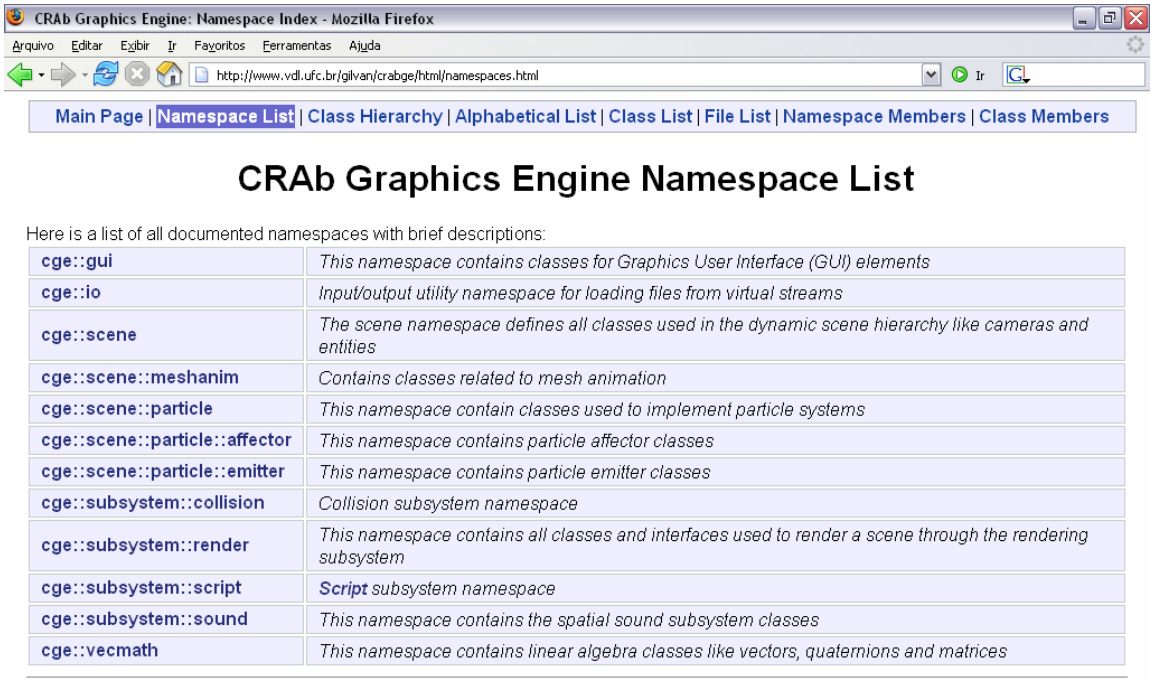


Figura C.2: Documentação do motor CGE publicada como uma página da *Web*.

Apêndice D

Exemplos de Código usando o Motor CGE

No motor CRAbGE, a classe *singleton Engine* centraliza os processo de inicialização e finalização do motor gráfico. A classe *Engine* é responsável por carregar plugins e recuperar os subsistemas utilizados pela camada de aplicação. A instância dessa classe é recuperada através de uma chamada ao método *getInstance*.

```
// o arquivo "cge.h" contém a interface das classes do motor CGE
#include "cge.h"
...

Engine& engine = Engine::getInstance();
```

Uma vez recuperada essa instância, é possível definir o diretório do sistema que é utilizado para carregar os *plugins* que a camada de aplicação utiliza.

```
engine.setPluginDirectory("./meusPlugins");
```

Após isso, a camada de aplicação pode iniciar o carregamento dos *plugins* que contém os subsistemas e demais módulos secundários utilizados pela aplicação.

```
engine.loadPlugin("OpenGLRender.plugin"); // renderizador OpenGL
engine.loadPlugin("OpenALSound.plugin");  // som espacial OpenAL
engine.loadPlugin("LuaScript.plugin");    // script Lua
engine.loadPlugin("OpcodeCollision.plugin");  // detecção de colisões
engine.loadPlugin("OctreeSceneManager.plugin"); // gerenciador de cena
engine.loadPlugin("Md2Loader.plugin");    // carregador de modelos MD2
```

Carregados os *plugins*, os subsistemas registrados são registrados através do método *initSelectedSubsystems* da classe *Engine*.

```
if( !engine.initSelectedSubsystems() )
{
      // cria uma mensagem de erro no Log
      CGE_ERR("Não foi possivel inicializar os subsistemas ");
      myErrorFunction();
}
```

Após isso, é possível recuperar os subsistemas registrados no motor que foram selecionados para prover um serviço, o que é realizado através de uma chamada ao método *getSelectedSubsystem* da classe *Engine*.

```
Subsystem* subsys = engine.getSelectedSubsystem(Subsystem::RENDER);
RenderSubsystem* rs = static_cast<RenderSubsystem*>(subsys);
```

Também é possível recuperar o gerenciador de cenas apropriadamente registrado, o que é realizado através de uma chamada ao método *getSceneManager* da classe *singleton SceneManagerEnumerator*.

```
SceneManagerEnumerator* sme = SceneManagerEnumerator::getInstance();
SceneManager* sman = sme->getSceneManager(Scene::SCENE_GENERIC);
```

Uma janela do sistema é criada através do método *createRenderWindow* da classe *RenderSubsystem*. Antes de inciar a pintura de uma cena no motor CGE, faz-se necessário utilizar um ponto de vista representado pela classe *Camera*. Essa classe pode ser instanciada diretamente, através do operador new, ou através do método *createCamera* da classe *SceneManager*. A principal vantagem de uma câmera através do gerenciador de cenas é que a destruição dessa câmera fica a cargo desse gerenciador, não sendo necessário destruir a câmera manualmente.

```
Camera* camera = sman->createCamera("TestCamera");
// posiciona a câmera no sistema global de coordenadas
camera->setPosition(100,200,100, Node::GLOBAL_SPACE);
// aponta a câmera para a origem
camera->lookAt(0,0,0, Vector3D::Y);
```

Após isso, é criada uma *viewport* que utiliza uma câmera para visualizar a cena que é exibida nessa janela do sistema. Criada uma janela do sistema para exibir as cenas renderizadas através de uma *viewport*, é possível entrar no laço de renderização. Isso é realizado através do método *startRendering* da classe *RenderSubsystem*. Alternativamente, as aplicações que desejem ter um controle mais fino sobre a pintura das cenas podem realizar chamadas ao método *updateAllRenderTargets* da classe *RenderSubsystem*, que é utilizado internamente pelo método *startRendering*.

```
Viewport* vp = win->addViewport(camera);
RenderWindow* win = rs->createRenderWindow("Minha Janela",
                                           800, 600, // dimensões
                                           true );   // modo fullscreen
// criação da viewport
Viewport* vp = win->addViewport(camera);

// entra no laço principal de pintura
rs->startRendering();

...
```

```
// pintura sob demanda, útil para editores de mundos, por exemplo
rs->updateAllRenderTargets();
```

Esses são os passos básicos para a construção de aplicações com o motor CGE. Os próximos trechos de código ilustram o carregamento de conteúdo no motor gráfico. O método *addDirectory* da classe *singleton ResourceDirectory* é utilizado para registrar diretórios virtuais utilizados para localizar os recursos que são carregados no motor. Tais diretórios virtuais podem ser arquivos compactados (zip, pk3 e compatíveis) ou diretórios do sistema de arquivos.

```
// entra no laço principal de pintura
ResourceDirectory* directory = ResourceDirectory::getInstance();
directory->addDirectory("./arquivos/hotel.zip");
directory->addDirectory("./arquivos/avatars.pk3");
directory->addDirectory("./arquivos/");
```

O método *setWorldGeometry* da classe *SceneManager* é utilizado para carregar os objetos estáticos que compõem a cena.

```
// usa um arquivo 3DS
sman->setWorldGeometry ("mundos/hotel.3ds");
```

O método *createEntity* dessa mesma classe é utilizado para criar objetos dinâmicos que são representados por uma malha tridimensional, que, dependendo do formato e arquivo utilizado, possui uma animação associada. Essa animação é manipulada através de uma instância de *AnimationState*.

```
Entity* obj = sman->createEntity("george","george.md2");
// usa a sequência "acenar" para animar o personagem virtual
AnimationState* state = obj->getMesh()->getAnimationState();
state->setCurrentSequence("aceno");
```

Entretanto, o objeto dinâmico criado através do método *createEntity* não possui um material adequado definido. Para tanto, é necessário criar um material e aplicá-lo nesse objeto dinâmico, o que é exemplificado pelo trecho de código abaixo.

```
// cria um material com uma camada de textura
MaterialManager* matman = MaterialManager::getInstance()
matman->createBasicMaterial("george.mat","george.png");
// cria um material com uma camada de textura
obj->setMaterialName("george.mat");
```

Por outro lado, o objeto dinâmico recém-criado não é exibido a não ser que seja inserido manualmente no grafo de cena, pois o gerenciador de cenas não tem como saber

qual deve ser o nó-pai desse objeto. A seguinte linha de código insere o objeto recém-criado na lista de filhos do nó-raiz do grafo de cena.

```
sman->getSceneRoot()->addChild(obj);
```

Fontes sonoras podem ser inseridas facilmente numa cena tridimensional através do método *createSound* da classe *SceneManager*. Diferentemente dos nós que são pintados na cena, os sons são imediatamente percebidos pelo usuário da aplicação, não sendo necessário inserir esse tipo de nó no grafo de cena. Entretanto, quando esses sons são emitidos por um avatar, por exemplo, é conveniente que sejam hierarquizados para que suas posições e velocidades sejam atualizadas de maneira automática.

```
// cria o som e toca no modo de repetição
Sound* som = sman->createSound("passaros","passaros.wav");
som->setLooping(true);
som->play();
```

As seguintes linhas de código recuperam o subsistema de script. Um trecho de código armazenado como uma cadeia de caracteres na memória é interpretado e um arquivo de script é pré-compilado para que seja interpretado posteriormente de maneira mais eficiente.

```
// recupera o subsistema de script
Subsystem* sys = engine.getSelectedSubsystem(Subsystem::SCRIPT);
ScriptSubsystem* script = static_cast<ScriptSubsystem*>(sys);

// executa um trecho de código
script->execute("obj:rotateY(90,Node.LOCAL_SPACE)");
// pré-compila um arquivo para interpretar depois
Script* code = script->compileFile("script.lua");
code->run();
```

Dado um par de objetos dinâmicos do grafo de cena, é possível detectar colisões facilmente entre esses objetos. Para tanto, recuperam-se os modelos de colisão, encapsulados pela classe *CollisionModel*, que representam internamente a geometria desses objetos. Isso é realizado através do método *getCollisionModel* da classe *Node*. Após isso, as interseções são calculadas através do método *collide* de uma instância de *CollisionModel*.

```
// assumindo que os nós objA e objB existem
CollisionModel* modeloA = objA->getCollisionModel();
CollisionModel* modeloB = objB->getCollisionModel();
```

```
// recuperam-se as matrizes de transformação desses objetos
Matrix4x4 mA, mB;
objA->getMatrix( mA, Node::GLOBAL_SPACE );
objB->getMatrix( mB, Node::GLOBAL_SPACE );

// uma lista de pares de faces, reportando as interseções
CollisionPairList faces;

if( modeloA->collide( faces, modeloB, &mB, &mA ) )
{
      // houve colisão! Cheque os pares de faces reportados!
      CollisionPairList::iterator it = faces.begin(),
                                          theEnd = faces.end();
      for(;it!=theEnd;it++)
      {
            int faceA = it->first;
            int faceB = it->second;

            // realizar os cálculos relevantes...
      }
}
```