UNIVERSIDADE FEDERAL DE PERNAMBUCO
GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
CENTRO DE INFORMÁTICA

TRABALHO DE GRADUAÇÃO

REVOLUTION ENGINE

ANÁLISE E IMPLEMENTAÇÃO DE UM MOTOR GRÁFICO 3D PARA JOGOS

Aluno: Leonardo Gesteira Costa

Orientador: Sílvio de Barros Melo

Recife
Agosto de 2005

AGRADECIMENTOS

Primeiramente, a meus pais e irmãos, por todo apoio e amor que me deram durante toda a vida e que, mesmo sem concordar, sempre apoiaram incondicionalmente as minhas escolhas.

Aos meus amigos da faculdade, impossível listá-los aqui. Por sempre me acompanharem durante a graduação, fosse jogando cinco cortes de futevôlei com bolinha de papel pra não ir pra aula, comendo bolovo no posto ou virando noite fazendo projetos.

Ao professor e orientador Silvio Melo, por todo o esforço que faz em prol da comunidade de computação gráfica no Centro de Informática, dando sempre motivação e suporte. Além da orientação e confiança depositadas em mim durante este trabalho.

Ao pessoal que faz ou fez parte da Jynx Playware, especialmente a André, Arthur, Fred, Jeff, Reginho, Rodrigo, Scylla e Tiago, por despertar meu interesse em desenvolver jogos. Graças a eles eu percebi o quão demasiadamente entediante é desenvolver qualquer outro tipo de sistema.

Finalmente, aos meus amigos, que sempre me acompanham nos bons e maus momentos. Por assim serem.

RESUMO

Nos últimos anos, o mercado mundial de jogos vem crescendo a uma taxa superior do que jamais esperado. Esta análise faz parte de um novo relatório elaborado pela *Screen Digest em suporte à Entertainment and Leisure Software Publishers Association*. O relatório mostra que nos últimos seis anos, o mercado de softwares de entretenimento digital é o que mais cresce em toda a indústria do entretenimento. Olhando para o futuro, de acordo com um relatório elaborado em junho de 2005 pela *PricewaterhouseCoopers*, a indústria de jogos mundial continuará ocupando essa posição mercadológica, chegando à soma de U$ 54.6 bilhões em 2009.

Nesse contexto, a qualidade e complexidade dos jogos vêm aumentando de tal forma, que a utilização de *frameworks* e padrões de projeto mostra-se crucial para a sobrevivência de qualquer projeto profissional. Este trabalho tem como objetivo principal a produção do principal *framework* de um jogo: o motor gráfico. Além disso, analisamos as tecnologias existentes mais promissoras para o desenvolvimento de uma nova ferramenta nessa área.

Dessa maneira, visamos incentivar a produção de jogos com qualidade mundial academicamente e profissionalmente no Centro de Informática e nas instituições que nele se apóiam.

ÍNDICE

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGOS

1 INTRODUÇÃO

Muito tempo se passou desde os dias de Doom. Lançado em dezembro de 1993 pela id Software[4], a primeira característica que o distinguia era os gráficos realistas tridimensionais, inexistentes nos jogos da época. Unindo a esse fato uma temática violenta, Doom rapidamente se tornou controverso e popular. Tão grande foi o seu sucesso, que outras empresas desenvolvedoras licenciaram o núcleo do seu software, e criaram outros jogos produzindo apenas o seu conteúdo, ou seja, os próprios gráficos, personagens, armas e fases. Assim, Doom não foi apenas um ótimo jogo, ele também evoluiu e popularizou um novo modelo de programação pra jogos: o motor de jogos.

Um motor de jogos é o software que compõe o núcleo de um jogo. O motor é responsável por tratar toda a tecnologia não específica a um jogo, incluindo renderização 2D e 3D, detecção de colisão entre os objetos do jogo, física, inteligência artificial, dentre outros. O elemento mais comum que um motor de jogos possui é o componente de renderização gráfica, conhecido como motor gráfico.

Um motor gráfico 3D é, simplesmente, um sistema que sintetiza imagens tridimensionais, em tempo real, respondendo a chamadas feitas por uma aplicação [1][2][3][5]. Para o desenvolvimento de tais sistemas, é necessário o conhecimento de uma ampla variedade de técnicas, algoritmos, estruturas de dados e conhecimentos matemáticos. Esse tipo de conhecimento é abstraído pelo usuário de um motor 3D, aumentando a produtividade do desenvolvimento de jogos.

1.1 MOTIVAÇÃO

O desenvolvimento de jogos é uma das áreas mais atraentes e promissoras da computação. A indústria de entretenimento interativo é formada por um mercado multibilionário que força ao limite as últimas tecnologias computacionais e guia pesquisas em áreas como Computação Gráfica ou Inteligência Artificial.

Essa busca incessante eleva os padrões de qualidade dos jogos, tornando o seu desenvolvimento cada vez mais complexo. De acordo com Júnior, em [5], *desenvolver um jogo é uma tarefa árdua e não-trivial, pois jogos são complexos sistemas de tempo-real*

multimídia e interativos. Os motores de jogos encapsulam esta complexidade em bibliotecas mais simples de utilizar ou modelos de arquitetura a serem adotados [2][3].

Contudo, os custos de utilização de um motor existente são significativos, muitas vezes limitando a qualidade dos jogos ao conhecimento das tecnologias por uma determinada empresa ou instituição.

O Centro de Informática (CIn) da Universidade Federal de Pernambuco (UFPE) vem realizando pesquisas em motores para o desenvolvimento de jogos. No entanto, essas pesquisas eram restritas a *frameworks* 2D e 2.5D, mudando apenas em [5], onde Júnior propõe um motor 3D para dispositivos móveis. Conseqüentemente, o desenvolvimento de jogos tridimensionais pra PCs e consoles (responsáveis pela maior fatia do mercado), tem sido inibido em todo o estado, tanto academicamente quanto profissionalmente.

1.2 OBJETIVOS

Este trabalho tem como objetivo principal a produção de um motor gráfico 3D para jogos. Essa produção é dividida em duas partes: a arquitetura e o desenvolvimento do motor.

A arquitetura é estudada e definida por um trabalho de graduação complementar, de autoria de Marco Túlio C. F. Albuquerque. Esse, analisa as funcionalidades existentes nos motores atuais e propõe uma arquitetura para a implementação do REvolution Engine.

Este trabalho pretende abordar o desenvolvimento, envolvendo uma análise das tecnologias existentes e necessárias à implementação do motor e a elaboração de um protótipo deste. Entretanto, o projeto de um motor gráfico exige uma demanda de mão-de-obra bastante além do possível para este trabalho. Portanto, iremos abordar as questões fundamentais e iniciais na construção de um motor: o gerenciamento de cena e colisão dos objetos em um jogo. Os problemas não contemplados serão considerados em uma futura extensão do projeto.

Além disso, temos a ambição de fomentar o desenvolvimento e a pesquisa no Centro de Informática na área de desenvolvimento de jogos tridimensionais, promovendo essa cultura ao meio profissional.

1.3 VISÃO GERAL

O restante deste trabalho encontra-se dividido da seguinte forma: na próxima seção serão apresentadas as tecnologias essenciais ao desenvolvimento do motor, assim como uma análise comparativa entre elas a fim de escolher as mais promissoras.

Na seção 3, o projeto do motor gráfico REvolution Engine é abordado, explicando as funcionalidades que o compõem, sua arquitetura, e o protótipo de sua implementação.

Por fim, na seção 4, são apresentadas as conclusões obtidas durante esse trabalho, os resultados obtidos as dificuldade encontradas e os possíveis trabalhos futuros nesse contexto.

2 TECNOLOGIAS

Como em todos projetos de software, uma atenção especial deve ser dada à escolha das tecnologias utilizadas no desenvolvimento de um motor gráfico 3D [5].

Dentre as tecnologias que envolvem o desenvolvimento de um motor 3D, foram analisadas linguagens de programação, plataformas, bibliotecas gráficas e componentes, dentre as mais promissoras existentes no mercado.

2.1 LINGUAGENS DE PROGRAMAÇÃO

A indústria de jogos utiliza como padrão de linguagem de programação C++. O principal motivo para adoção desta linguagem é que ela consegue utilizar técnicas mais modernas, como orientação a objetos, a um custo de performance reduzido.

É interessante notar que a performance é uma das peças chaves de um jogo. Tal fato é devido à natureza deste tipo de software, que requer processamento massivo e em tempo real de conteúdo multimídia e interativo, levando, às vezes, à utilização de códigos em C ou *Assembly* para tarefas críticas.

Apesar de linguagens como C# e Java terem sido largamente introduzidas no mercado de TI, o desempenho dessas ainda é aquém do oferecido por C++.

2.2 PLATAFORMAS

Com relação à plataforma de um jogo, o mercado que mais cresce atualmente é o de consoles. Nesse contexto, consoles são dispositivos eletrônicos criados especificamente para jogos, conhecidos antigamente como *vídeo games*. Entretanto, os ambientes de desenvolvimento particulares destas plataformas tem um custo bastante elevado e não são comercializados nacionalmente.

Assim, utilizando um computador (PC) como plataforma, o sistema operacional padrão dos usuários de jogos é o Microsoft Windows[45]. Esse, possui um ambiente de desenvolvimento altamente popular, sendo então escolhido para a implementação do motor.

2.3 BIBLIOTECAS GRÁFICAS

Na indústria de jogos, gráficos 3D se tornaram tão populares que bibliotecas (APIs) especializadas foram criadas para facilitar o processo em todos os estágios de geração de imagens. Essas APIs também têm se provado vitais para fabricantes de hardware, pois elas disponibilizam um modo de acessar as mais inovadoras e complexas vantagens dos dispositivos de hardware de uma maneira abstrata.

Dentre as bibliotecas existentes, OpenGL[6] e Direct3D[7] são as mais populares para a geração de gráficos tridimensionais. Muitas placas de vídeo modernas provêm aceleração de hardware baseada nessas APIs, possibilitando a criação de gráficos 3D complexos em tempo real.

2.3.1 OPENGL

OpenGL (Open Graphics Library) é uma API procedural para gráficos 3D. A interface consiste de aproximadamente 250 funções que podem ser usadas para renderizar desde simples primitivas a cenas complexas [10].

A biblioteca evoluiu de uma interface 3D antiga da SGI[8], Iris GL. Sua especificação e evolução de OpenGL são revisadas e administradas pela OpenGL Architecture Review Board (ARB), um consórcio independente formado em 1992. Dentre as empresas participantes da ARB, em novembro de 2004, estão: 3D Labs[11], Apple Computer[12], ATI Technologies[13], Dell Inc.[14], Evans & Sutherland[15], Hewlett-Packard[16], IBM[17], Intel[18], Matrox[19], NVIDIA[20], SGI e Sun Microsystems[21]. A Microsoft, uma das fundadoras da associação, saiu em março de 2003.

2.3.2 DIRECT3D

O Direct3D é um componente do DirectX. Esse, é uma coleção de bibliotecas para o desenvolvimento de jogos baseadas na tecnologia COM [9][38].

O DirectX é uma tecnologia proprietária da Microsoft[22] e foi concebido em 1996, após a Microsoft adquirir a companhia britânica Rendermorphics. A sua API 3D, Reality Lab, foi usada como base para o desenvolvimento do Direct3D.

2.3.3 ANÁLISE COMPARATIVA

Apesar de OpenGL e DirectX serem bastante parecidas atualmente, ainda existe uma série de diferenças. A fim de escolher uma das bibliotecas vários fatores devem ser analisados, levando em consideração não apenas a parte técnica, mas também a estratégica de um projeto usando essas tecnologias.

Curva de Aprendizagem

Antes da versão 8, Direct3D era conhecida como uma biblioteca de difícil utilização. Por exemplo, para habilitar transparência (*alpha blending*) era necessário criar um chamado *execute buffer*, travar ele para utilização, enviar a ele uma série de códigos e estruturas que definiam a operação, destravá-lo, e enviar para o *driver* executar. Por outro lado, com OpenGL apenas uma chamada a uma função resolveria o problema. Esse modelo era tão frustrante para os programadores que levou a John Carmack, sócio fundador da id Software, a escrever um famoso manifesto no qual questionava a Microsoft para abandonar o DirectX em favor de OpenGL.

A partir do Direct3D 8, várias mudanças foram feitas para simplificar a sua estrutura. Entretanto, as duas bibliotecas ainda seguem paradigmas diferentes: Direct3D é baseada na tecnologia COM da Microsoft, sendo, assim, fortemente voltada à orientação de objetos; OpenGL não possui um paradigma bem definido, é apenas uma máquina de estados acessada através de funções. A dificuldade de utilização das duas estruturas é uma questão subjetiva.

A máquina de estados de OpenGL já foi utilizada para criar uma interface orientada a objetos na biblioteca Fahrenheit. O projeto foi anunciado no SIGGRAPH 1998 (Special Interest Group on Computer Graphics) como um esforço conjunto entre a Microsoft, a SGI e a Hewlett-Packard com o objetivo de unir as interfaces de OpenGL e Direct3D. Apesar de promissor, por falta de recursos financeiros da SGI e de suporte da indústria, o projeto foi abandonado.

Uma diferença significante entre as APIs é que Direct3D é fundamentalmente tridimensional. A Microsoft atualiza constantemente a biblioteca com inovadoras tecnologias presentes nas placas de vídeos 3D. No entanto, se comparada com OpenGL, as renderizações 2D são feitas de uma maneira muito complexa.

Além disso, a documentação de Direct3D nem sempre é precisa e muitas vezes pouco intuitiva. Em contraste, OpenGL é considerada muito bem documentada e sua comunidade provê um suporte bastante útil.

Performance

Logo após Direct3D e OpenGL se estabelecerem como bibliotecas gráficas, Microsoft e SGI começaram a travar uma disputa com relação aos vários aspectos fundamentais de uma API. Entre esses, qual delas oferecia uma performance superior. Muito dessa batalha é relacionada ao fato de que, durante muito tempo, a implementação era voltada pra renderização em software, pois os custos de hardware eram muito altos. Assim, a habilidade de uma biblioteca ser mais rápida dependia exclusivamente da sua implementação e não de outras circunstâncias, como placas gráficas.

Inicialmente, a Microsoft criou uma implementação tanto de Direct3D como de OpenGL para sua plataforma. A comparação resultou como Direct3D tendo uma melhor performance, o que foi considerado justo, considerando a especificação rigorosa da OpenGL ARB. Contudo, no SIGGRAPH 1996 a SGI desafiou a Microsoft com uma própria implementação de OpenGL, chamada CosmoGL. Essa obteve uma performance igual, ou superior, a do Direct3D. Para a SGI, esse fato foi crucial, visto que a falta de performance não era devido à estrutura de OpenGL, mas à não otimização da implementação da Microsoft.

Desde então, a performance deixou de ser o foco da disputa entre as bibliotecas, visto que não há provas suficientes de qual API realmente é mais rápida. Isso porque a performance de uma aplicação depende muito mais da habilidade de um programador de escrever código eficiente e da presença de bons hardwares e *drivers* do que da biblioteca utilizada.

Estrutura

OpenGL é uma API procedural gráfica. A biblioteca foi estruturada para aproveitar o máximo de características gráficas independente da existência de suporte de hardware. De certa maneira, a especificação de OpenGL guia a implementação dessas características em hardware. E essas implementações automaticamente melhoram a performance das aplicações que usam a API. Assim, OpenGL realiza uma abstração implícita do hardware.

Por exemplo, quando a placa gráfica GeForce 256 foi lançada, jogos como Quake III Arena foram automaticamente beneficiados da implementação de T&L[1] em hardware, enquanto desenvolvedores Direct3D precisaram aguardar uma nova versão da API e alterar os jogos para usufruir dessa característica.

Essa abordagem inclusiva de OpenGL possui amplo sentido para características essenciais que podem ser executadas em software. Porém, a desvantagem é que a biblioteca não provê uma maneira direta de saber se certa propriedade é ou não suportada pelo hardware. Assim, mesmo que uma característica seja inviável de ser executada em software, há a possibilidade de a sua utilização, degredando a performance do aplicativo.

Por outro lado, Direct3D foi projetada desde o início com uma camada de abstração do hardware, não processando tacitamente as requisições de um aplicativo. Ou seja, a biblioteca possui uma abordagem exclusiva, impossibilitando o desenvolvedor de habilitar uma característica a ser executada em hardware se esse não for capaz.

Extensibilidade

O mecanismo de extensão de OpenGL é provavelmente a diferença mais disputada entre as duas bibliotecas. OpenGL tem um mecanismo onde qualquer *driver* pode propagar suas próprias extensões da API, introduzindo novas funcionalidades [6][10]. Isso permite que novas funcionalidades sejam distribuídas rapidamente, mas cria a possibilidade de ter versões conflitantes à medida que diferentes distribuidores implementam a mesma extensão de maneira distinta. Por esse motivo, muitas dessas extensões são periodicamente padronizadas pela OpenGL ARB e incluídas em futuras versões da biblioteca.

Direct3D não é extensível, sendo implementado apenas pela Microsoft. Isso leva a uma API mais consistente, mas impossibilita que alguns fabricantes habilitem algumas características específicas de seus produtos. Com exemplo, podemos citar a tecnologia UltraShadow[23] da NVIDIA, indisponível no Direct3D, que permite uma criação de sombras mais realistas com uma melhor performance.

---

[1] Transformação e Iluminação (T&L) são duas etapas da geração de imagens 3D em uma placa de vídeo. O suporte de hardware a T&L faz com que essas sejam efetuadas pelo processador de vídeo, melhorando a performance do aplicativo.

Um exemplo da diferença de extensão entre as duas bibliotecas foi a maneira como as placas gráficas adicionaram suporte a *pixel shaders*[2]. Em Direct3D havia um simples padrão denominado PS1.1, enquanto em OpenGL a mesma funcionalidade era acessada através de extensões customizadas. Teoricamente, a abordagem da Microsoft significava que era possível um único código suportar mais de uma placa de vídeo, mas em OpenGL o programador teria que codificar *pixel shaders* diferentes. Na prática, não era tão simples. Por exemplo, a implementação da NVIDIA para *shaders* variava tanto de acordo com as características da placa, que era necessário para o desenvolvedor produzir vários *shaders*. Contudo, essa situação mudou para as duas APIs com a introdução da segunda geração de *shaders*, os quais possuem uma linguagem padrão e são compilados de acordo com a placa de vídeo [24].

Portabilidade e Adoção da Indústria

A biblioteca OpenGL tem implementações em uma série de plataformas, incluindo Windows, sistemas Unix, Linux, e o Mac OS X. Quase qualquer sistema operacional é capaz de rodar OpenGL usando algum *driver* ou software como o Mesa 3D[25].

Direct3D é uma API específica para Windows. Uma terceira companhia, Coderus[26], desenvolveu uma interface compatível com DirectX para Mac, intitulada MacDX.

Placas de vídeo mais antigas tendem a oferecer um melhor suporte a Direct3D do que a OpenGL. Entretanto, atualmente, tanto a ATI como a NVIDIA, as duas maiores empresas de placas de vídeos, possuem *drivers* tanto de Direct3D como de OpenGL para diversas plataformas.

Com relação à indústria de consoles, a Microsoft anunciou em março de 2004 na Game Developers Conference (GDC)[27] que está migrando o DirectX para uma nova biblioteca, a XNA (DirectX New generation Architecture)[28]. O intuito da Microsoft é manter apenas o XNA como API de desenvolvimento de jogos para o Windows Vista – a nova versão do Windows – e o XBOX 360[29] – o próximo console da empresa.

---

[2] *Pixel shaders* são programas que são executados no processador da placa gráfica (GPU) uma vez para cada *pixel* de um objeto tridimensional especificado, podendo ter diversos objetivos, como colorir o objeto ou aplicar um *bump mapping*.

Por sua vez, em março de 2005, a Sony[30] revelou na GDC que a nova geração do PlayStation[31], conhecida como PS3, irá suportar APIs populares como parte de sua arquitetura, como a OpenGL ES. A escolha dessa versão de OpenGL foi baseada principalmente no fato de que a versão original inclui uma série de artefatos não utilizados por desenvolvedores de jogos.

2.3.4 CONCLUSÃO

Considerando a comparação apresentada na seção anterior, é notável que, após anos de disputa pelo mercado de desenvolvedores de jogos, as duas bibliotecas evoluíram a dois modelos distintos, mas que não apresentam nenhuma vantagem significativa à escolha entre uma das tecnologias.

Cada uma das diferentes abordagens entre OpenGL e Direct3D tem suas vantagens e desvantagens, tornando a definição da melhor biblioteca uma questão subjetiva. Contudo, há uma exceção em que OpenGL se destaca com relação a Direct3D: a curva de aprendizagem. A definição, a documentação e o suporte da comunidade de OpenGL podem ser cruciais ao desenvolvimento de um novo projeto

Deste modo, a biblioteca gráfica a ser adotada na implementação do motor gráfico é a OpenGL.

2.4 COMPONENTIZAÇÃO

Os sistemas modernos têm se tornado cada vez maiores, mais complexos e dificilmente controláveis, resultando em altos custos de desenvolvimento, baixa produtividade, baixa qualidade e um alto risco de dependência de uma tecnologia. Conseqüentemente, há uma crescente busca por um novo paradigma de desenvolvimento de software.

Uma das soluções mais promissoras é o desenvolvimento baseado em componentes. Essa abordagem é baseada na idéia que sistemas de software podem ser construídos selecionando componentes existentes e juntando-os com uma arquitetura bem definida [34][35].

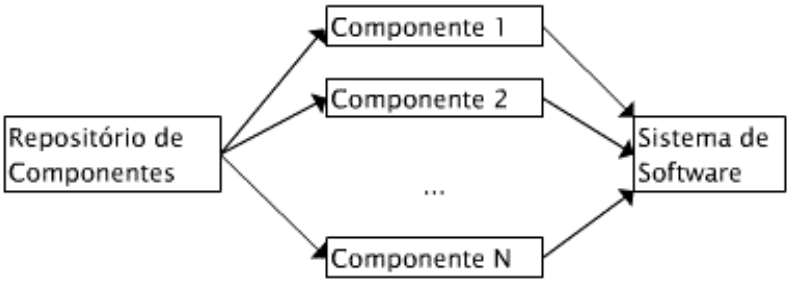

Figura 1. Desenvolvimento baseado em Componentes.

O uso de componentes no desenvolvimento de software pode significativamente reduzir o custo de desenvolvimento e o *time-to-market* e melhorar a manutenibilidade, confiabilidade, reusabilidade e qualidade de um software, dentre outros [32][33][34][36].

Considerando que o motor é o software que compõe o núcleo de um jogo e as vantagens supracitadas do uso de componentes, é essencial o desenvolvimento do motor como um componente.

As tecnologias mais usadas atualmente para o desenvolvimento de componentes, incluindo suas variações, são CORBA[37], Component Object Model[38] e Java Beans[39].

2.4.1 CORBA

CORBA é um padrão para interoperabilidade de aplicações definido e suportado pela Object Management Group (OMG)[40], uma organização de mais de 400 empresas de tecnologia. A tecnologia gerencia os detalhes da interoperabilidade entre os componentes e permite aos aplicativos se comunicar entre si independente da sua localização ou arquitetura. A única maneira que aplicativos e componentes podem se comunicar é através da interface.

A parte mais importante de CORBA é a chamada Object Request Broker (ORB). Esse, é o *middleware* responsável por estabelecer as relações de cliente e servidor entre os componentes. Usando a ORB, um cliente pode chamar um método no objeto servidor, de maneira totalmente transparente, sendo responsável por interceptar uma chamada, localizar um objeto que implemente o método requisitado, passar os parâmetros, chamar o método e retornar a resposta [41]. O cliente não precisa saber a localização do objeto, a linguagem de

programação, a plataforma ou nenhum outro aspecto que não seja relacionado com a interface.

2.4.2 COMPONENT OBJECT MODEL

Introduzida em 1993, Component Object Model (COM)[38] é uma arquitetura genérica para desenvolvimento baseado em componentes. Criada pela Microsoft, a tecnologia evoluiu da Object Linking and Embedding (OLE), uma tecnologia criada para a comunicação entre documentos e objetos do Microsoft Office [44]. Apesar de já ter sido implementada em várias plataformas, a tecnologia é usada basicamente no Windows.

COM define como os componentes e seus clientes interagem. Essa interação é feita sem nenhum sistema intermediário. A tecnologia provê um padrão binário para que os componentes e seus clientes se comuniquem, o que os torna independentes de linguagens [42][43].

A partir do Windows 2000, uma extensão do COM foi lançada, chamada de COM+. Essa, permite que os componentes usem o Microsoft Transaction Server (MTS), gerenciando automaticamente tarefas como *pool* de recursos, propagação de eventos, desconexão entre aplicativos e componentes distribuídos.

2.4.3 JAVABEANS

A tecnologia de desenvolvimento de componentes da Sun consiste basicamente em duas partes: JavaBeans para a criação do componente do ponto de vista do cliente, e Enterprise JavaBeans (EJB), do ponto de vista do servidor.

A plataforma Java provê uma solução eficiente para a portabilidade e segurança, usando *bytecodes* de Java e o conceito de *applets* e *sandbox*. Portanto, os componentes e clientes são restritos à linguagem de programação Java e herdam todas as suas vantagens e desvantagens.

2.4.4 ANÁLISE COMPARATIVA

Uma análise comparativa detalhada das tecnologias de desenvolvimento baseado em componentes pode ser encontrada em [35]. As diferenças avaliadas nessa análise são sumarizadas abaixo.

	CORBA	COM	EJB
Ambiente de desenvolvimento	Não existente.	Bastante diversificado com suporte de diversas linguagens.	Limitado devido à dependência de linguagem.
Interface binária	Não existente.	A interação entre os componentes é essencialmente por uma interface binária.	Baseada em COM, mas específica de Java, usando *bytecodes*.
Manutenibilidade	CORBA IDL como padrão de definição da interface, necessitando manutenções.	Microsoft IDL como padrão de definição da interface, necessitando manutenções.	A interface é definida entre o componente e o container, diminuindo manutenções.
Dependência de plataforma	Independente.	Implementada em várias plataformas, mas usada basicamente no Windows.	Independente.
Dependência de linguagem	Independente.	Independente.	Dependente da linguagem Java.
Implementação	Usada em aplicações corporativas tradicionais.	Usada em aplicações para usuários comuns.	Usada para desenvolvimento de servidores.

Tabela 1. Comparativo entre tecnologias para o desenvolvimento de Componentes.

2.4.5 CONCLUSÃO

No intuito de desenvolver um motor de jogos, a tecnologia de desenvolvimento de software baseado em componentes que mais se destaca entre as apresentadas é a Component Object Model, da Microsoft.

COM tem uma grande variedade de ambientes de desenvolvimento, o que facilita e aumenta a produtividade, diminuindo também fatores como a manutenibilidade.

A tecnologia é fortemente baseada em interfaces binárias, o que melhora a performance de comunicação entre os componentes e seus clientes. Conforme citado anteriormente, a natureza de um jogo requer que ele tenha uma performance bastante otimizada.

Além disso, a solução tem sido usada como padrão para aplicações voltadas a um ambiente *desktop*, sendo assim, praticamente comprovada.

3 REVOLUTION ENGINE

Essa seção é responsável por expor os componentes do motor, sua arquitetura e o protótipo da sua implementação.

3.1 COMPONENTES

Conforme explicado anteriormente na seção 1.2, o escopo do projeto envolve o gerenciamento de cenas e a colisão entre objetos no jogo. Uma breve descrição sobre os tópicos é apresentada nessa seção. Para maiores detalhes sobre as funcionalidades que compõem um motor gráfico, o leitor pode consultar [1][2][3][5].

3.1.1 GRAFO DE CENAS

O gerenciamento de uma cena tridimensional é feito basicamente através de um *Grafo de Cenas*. Esse, é uma representação hierárquica em árvore as unidades de uma cena, chamados nós. Essas unidades podem ser representadas por objetos, acrescentados de texturas, transformações, nível de detalhes (LOD), estados da renderização (como propriedades de um material, por exemplo), fontes de luz, e o qualquer outro componente que seja considerado apropriado.

De certa maneira, toda aplicação gráfica possui um grafo de cena, mesmo que este possua apenas a raiz e um conjunto de nós a serem renderizados [1]. A forma como um nó é representado varia de acordo com a arquitetura do sistema que o implementa.

A representação de grafos de cena pode ter vários propósitos. Basicamente, o modelo é usado para a renderização de uma cena tridimensional, entretanto, pode ser usado para detecção de colisão, *culling*[3] ou *picking*[4].

Um modelo de avião pode ser representado num grafo de cena como na figura abaixo.

---

[3] *Culling* é o processo de remover as porções de uma cena que não causam nenhuma contribuição no resultado final de um processo. A técnica pode ser usada para não renderizar polígonos invisíveis pela câmera, simplificar cálculos físicos ou de inteligência artificial, por exemplo.

[4] *Picking* se refere ao processo de selecionar um objeto 3D a partir de sua projeção 2D na tela com o auxílio de um apontador (seja ele um mouse ou um marcador controlado pelo teclado) [5].

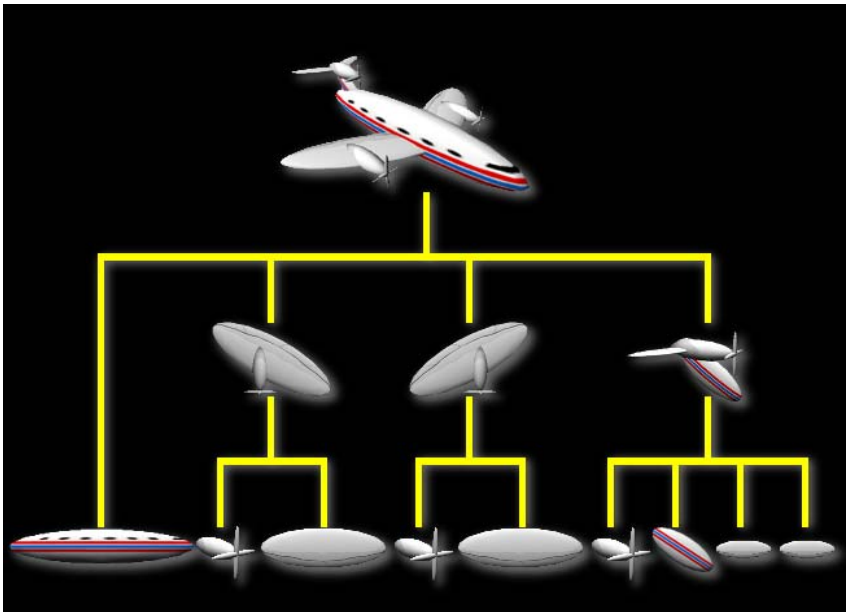

Figura 2. Exemplo de grafo de cena.

3.1.2 Detecção de Colisão

A detecção de colisões é um problema vital em várias aplicações gráficas e de realidade virtual. Dentre as áreas que incluem essa questão, podemos citar animação computadorizada, robótica, jogos e simuladores de tráfego [1].

O problema é parte do chamado gerenciamento de colisões. Esse, é dividido em três partes: detecção de colisão, determinação de colisão, e resposta de colisão. A detecção consiste na verificação de colisão entre dois ou mais objetos de uma cena; enquanto determinação de colisão indica as reais interseções entre os objetos; finalmente, a resposta é responsável por determinar quais ações devem ser tomadas ao acontecer uma colisão.

Existem diversos algoritmos e técnicas que visam solucionar esse problema. No entanto, esses são sensíveis ao contexto [1][2][3]. Uma colisão melhor detalhada pode ser obtida em troca de performance e processamento, tornando a escolha de uma técnica específica uma questão subjetiva. O não tratamento de colisões pode prejudicar sensivelmente a qualidade de um jogo, quebrando o realismo e a imersão, como exemplificado na Figura 3.

Figura 3. Deer Hunter 2004. Exemplo de colisão.

3.2 PROJETO

3.2.1 ARQUITETURA

A arquitetura do motor, conforme explicado anteriormente, foi definida no trabalho de graduação de Marco Túlio C. F. Albuquerque. Entre os requisitos não funcionais dessa modelagem, os mais importantes têm como objetivo a criação de uma estrutura portável e que permita um ótimo desempenho. Assim, o motor foi dividido em três módulos, compostos pelo *HandlerRegistry*, *Visitor*, e *SceneManager*.

Uma breve explanação sobre a responsabilidade e os motivos dessa abordagem será dada aqui. Para maiores detalhes, consultar o trabalho de Túlio Albuquerque.

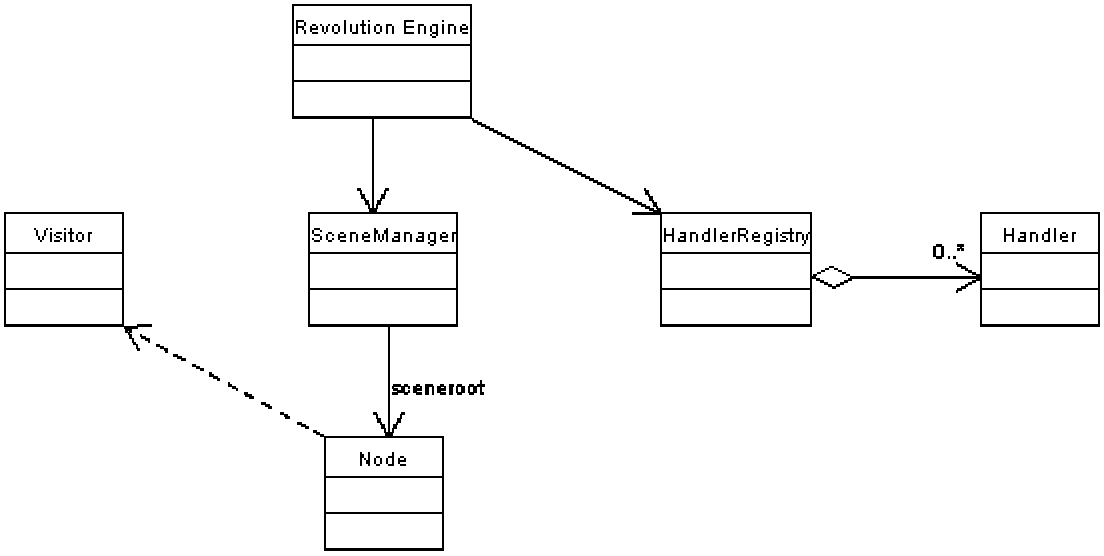

Figura 4. Arquitetura do motor.

O módulo composto pelos *Visitors* é o responsável pela execução de tarefas que utilizem o grafo de cena. Os *Visitors* principais serão implementados pelo motor, como, por exemplo, atualização e renderização dos objetos da cena. Esse módulo contém código específico da plataforma de execução do motor.

O segundo módulo, composto pelo *HandlerRegistry* é o responsável por executar as operações de renderização da cena em si. Essas tarefas, como aplicação de texturas, materiais, inserção de pontos de luz, dentre outras, utilizam código específico da biblioteca gráfica utilizada.

O *SceneManager* é o responsável pelo gerenciamento da cena, incluindo operações de inserção, alteração e remoção de objetos ou propriedades da cena. Todas as classes que compõem esse módulo utilizam algoritmos para construção e manutenção da árvore, como em BSPs ou Octrees. O módulo usa apenas código padrão de C++, necessitando apenas ser recompilado para diferentes plataformas de execução. O *SceneManager* possui a raiz do grafo de cena. A arquitetura do grafo de cena é mostrada na Figura 5.

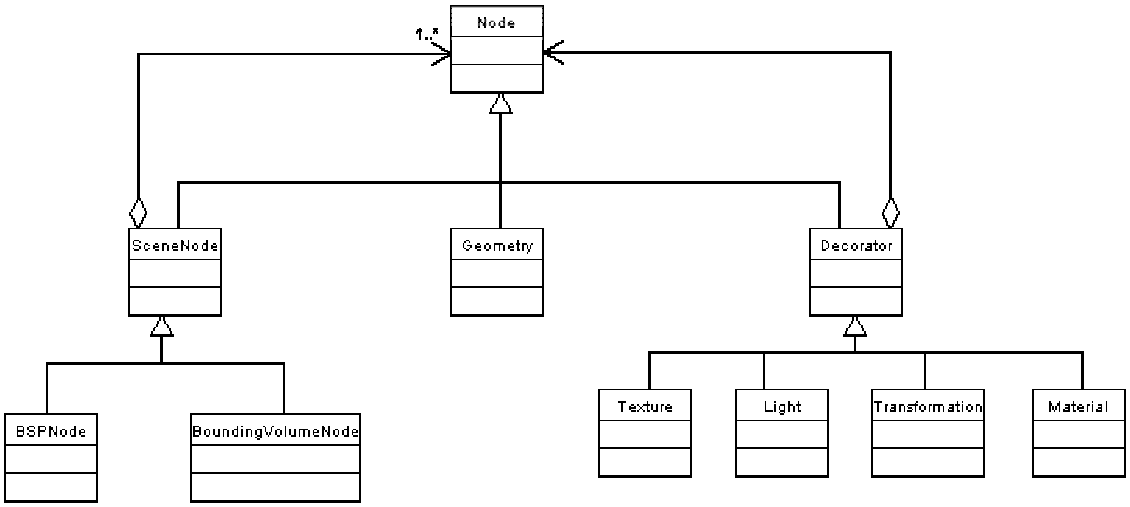

Figura 5. Arquitetura do Grafo de Cena.

Um nó do grafo de cena é representado pela classe *Node*. Essa, é extendida por três outras: *SceneNode*, *Geometry* e *Decorator*.

Os nós de cena (*SceneNode*) são responsáveis por representar a cena em si. A sua representação pode ser feita por ordenação espacial (*BSPNode*) ou de hierarquias de objetos (*BoundingVolumeNode*). Cada abordagem possui um propósito específico, ficando a cargo do usuário do motor a escolha.

Os nós representados por *Geometry* são folhas que possuem a malha, ou conjunto de triângulos que compõem um objeto da cena.

Finalmente, os nós *Decorator* são responsáveis pelas propriedades de uma cena ou um objeto. Essas propriedades podem indicar a orientação do objeto, uma textura a ser aplicada, a escala do objeto, um foco de luz existente, dentre outros. Os decoradores são processados pelo *HandlerRegistry* e enviados ao um *Handler* específico para aplicar a característica do decorador à cena.

Na representação do grafo de cena, a árvore é percorrida em profundidade para a sua renderização. Por exemplo, uma fonte de luz (*Light*) pode ser colocada em um nó interno, o qual afeta apenas o conteúdo da sua sub-árvore. Outro exemplo é quando uma há um nó de textura (*Texture*) na árvore. Essa textura deve ser aplicada a toda a árvore abaixo desse nó.

Cada módulo do motor é dependente apenas de uma tecnologia, podendo essa ser alterada sem afetar o sistema como um todo. Além disso, outras operações não

implementadas pelo motor, bem como particulares ao contexto de um jogo, podem ser construídas extendendo essa estrutura.

3.2.2 PADRÕES DE PROJETO

O projeto do motor REvolution Engine foi baseado no paradigma de orientação de objetos. Nesse paradigma, vários padrões de projeto foram estudados e analisados com o objetivo de aumentar a flexibilidade, modularidade, robustez e reusabilidade do projeto [46]. Os padrões utilizados serão explicados juntamente com a sua implementação.

3.2.3 IMPLEMENTAÇÃO

A implementação do motor foi baseada na arquitetura supracitada e utilizando as tecnologias analisadas na seção 2. Abaixo, segue uma descrição da implementação das partes mais sofisticadas dessa arquitetura.

RevolutionEngine

Essa classe é a classe principal de acesso ao motor gráfico. Considerando o motor como um subsistema, a classe utiliza o padrão *Facade.*

A classe implementa também a interface *IUnkown* da tecnologia COM, fazendo com que todo o motor gráfico se comporte como um componente. Essa característica, oferece ao projeto vantagens como facilidade de controle de versões e atualizações do motor em um determinado jogo, bem como todas as vantagens que envolvem o desenvolvimento de software baseado em componentes.

HandlerRegistry

O *HandlerRegistry* é o responsável por tratar todas as requisições de renderização de um objeto ou aplicação de estados de renderização (como textura ou materiais, por exemplo) e enviar ao *Handler* responsável por processar essa requisição. Assim, o *HandlerRegistry* é um gerenciador único no sistema, sendo, portanto, um *singleton*. Esse padrão de projeto é usado para garantir que em um sistema só vai existir uma, ou uma quantidade determinada, de instâncias de um determinado objeto [46].

A implementação do *singleton* foi baseada na proposta em [47], onde um *template* do *singleton* é implementado, a fim de facilitar o controle e a manutenção das classes que adotem esse padrão de projeto.

```
#include <cassert>

template <typename T> class Singleton
{
    static T* ms_Singleton;

public:
    Singleton( void )
    {
        assert( !ms_Singleton );
        int offset = (int)(T*)1 - (int)(Singleton <T>*)(T*)1;
        ms_Singleton = (T*)((int)this + offset);
    }
   ~Singleton( void )
        {  assert( ms_Singleton );  ms_Singleton = 0;  }
    static T& GetSingleton( void )
        {  assert( ms_Singleton );  return ( *ms_Singleton );  }
    static T* GetSingletonPtr( void )
        {  return ( ms_Singleton );  }
};

template <typename T> T* Singleton <T>::ms_Singleton = 0;
```

Código 1. Template do Singleton.

O objetivo desse *template* é criar uma classe pai para todos os *singletons* do sistema. Assim, todo o trabalho do *singleton* é feito no construtor do *template*, o qual é responsável por calcular o endereço relativo da instância do objeto e colocá-lo no ponteiro do *singleton*.

Deste modo, ao *HandlerRegistry* é apenas necessário derivar *template* supracitado para que se comporte como um *singleton*.

```
class HandlerRegistry : public Singleton <HandlerRegistry>
{

private:
    vector<Handler> handlers;

public:
    void apply(Decorator* pDecorator);

    // ...
};
```

Código 2. HandlerRegistry.

SceneManager

O *SceneManager* é o responsável pelo gerenciamento da cena. É através dele que o usuário pode manipular os objetos da cena. O *SceneManager* possui uma lista desses objetos e é responsável por montar o grafo da cena de acordo com a estrutura de ordenação requerida por uma operação.

Por ser uma estrutura de gerenciamento única no sistema, o padrão singleton também foi adotado para o *SceneManager*.

```
class SceneManager : public Singleton <SceneManager>
{
private:
    vector<Object> objects;

public:
    // ...
};
```

Código 3. SceneManager.

Um objeto do tipo *Object* é o elemento usado pelo usuário para inserir dados na cena, como, por exemplo, um foco de luz. O *SceneManager* usa essa lista para montar o grafo de cena, de forma transparente ao usuário.

Os objetos da cena são classificados como elementos estáticos ou dinâmicos. Os objetos estáticos são caracterizados por não alterar seus atributos (como sua posição, tamanho, forma, dentre outros) ao decorrer do jogo, como, geralmente, uma parede ou um prédio. Entretanto, os objetos dinâmicos sofrem essas modificações, como, por exemplo, um animal ou uma bola em um jogo de futebol, necessitando de certas operações características ao objeto ou ao contexto do jogo. Por exemplo, em um jogo de policial o personagem deveria colidir toda vez que entrasse em contato com uma parede. Porém, se esse policial for um mutante, ele poderia ter a possibilidade de atravessar uma parede. Assim, esses processamentos sensíveis ao contexto do jogo não devem estar acoplados ao motor. A fim de solucionar esse problema, resolvemos adotar o padrão de projeto *delegate* para essas operações.

Delegates são objetos que permitem a invocação de seus métodos por outros objetos. A grande particularidade é que esses objetos se comportam como funções livres,

não acopladas a um objeto específico. Como o nome indica, ele delega a chamada de um método a um objeto específico, em tempo de execução.

Linguagens como Java e da plataforma .NET (C#, VB.NET, MC++) possuem *delegates* acoplados à linguagem. Entretanto, o padrão não faz parte da linguagem C++. Assim, implementamos a construção de *delegates* genéricos através do uso de *templates*, permitindo a fácil utilização desses. O Código 4 expõe como foi implementada a técnica.

```
template <typename Class, typename T1, typename Result>
struct delegate_base {

    typedef Result (Class::* MethodType)(T1);
    Class*     obj;
    MethodType method;

    delegate_impl(Class* obj_, MethodType method_)
            : obj(obj_), method(method_) {
    }

    Result operator()(T1 v1) {
        return (obj->*method)(v1);
    }
};


template <typename T1, typename Result>
struct delegate {
    delegate_base<T1, Result>*  pDelegateBase;

    template <typename Class, typename T1, typename Result>
    delegate(Class* obj, Result (Class::* method)(T1))
        : pDelegateBase (new delegate_base<Class, T1, Result>(obj, method)) {

    }

    ~delegate() { }

    Result operator()(T1 v1) {
        return (*pDelegateBase)(v1);
    }
};
```

Código 4. Delegates.

A primeira estrutura, `delegate_base`, guarda um ponteiro para o objeto e para o método a ser invocado, cujos tipos são parametrizáveis. Utilizando-se dessa, a estrutura `delegate` recebe o objeto responsável por executar um determinado método e esse método a ser executado por ele.

Apesar de uma complexa implementação, a utilização de um delegate se torna bem simples. Continuando com o exemplo da colisão entre os objetos, cuja resposta à detecção é sensível ao contexto do jogo, o usuário do motor pode implementar um gerenciador de colisões delegando a ele o tratamento das colisões do sistema, conforme exemplificado abaixo.

```
class GerenciadorColisao
{

public:
    void colidir(void*) {

        // ...
    }
};


int main(int argc, char* argv[])
{
    GerenciadorColisao gerenciador;
    delegate<void*, void> resposta(&gerenciador,
                                   GerenciadorColisao::colidir);

    Object carro;
    // ...

    carro.setCollisionResponse(resposta);

    return 0;
}
```

Código 5. Exemplo de gerenciamento de colisões usando delegates.

Apesar de uma complexa implementação, o motor foi projetado com o objetivo de ter uma fácil utilização por um desenvolvedor de um jogo. Unidos a esse propósito básico, estavam requisitos primários como performance, portabilidade, extensibilidade, reusabilidade, dentre outros.

4 CONCLUSÃO

4.1 RESULTADOS ALCANÇADOS

O desenvolvimento de um motor 3D para jogos é um trabalho árduo e não-trivial. A implementação de um pequeno protótipo do motor foi um *milestone* de grande importância para comprovar a performance, qualidade, escalabilidade e flexibilidade da estrutura do motor.

O ambiente de teste do motor foi montado a partir de um jogo desenvolvido na cadeira Desenvolvimento de Jogos, do Centro de Informática. O jogo, denominado 7seas, foi o único jogo tridimensional já produzido no CIn. A estrutura do jogo foi modificada para se adequar à arquitetura do motor gráfico.

Devido ao curto prazo deste projeto, alguns algoritmos específicos de renderização não podiam ser portados para a biblioteca gráfica OpenGL. Conseqüentemente, na implementação do protótipo foi utilizada a biblioteca Direct3D, visto que era a tecnologia original do jogo.

Na implementação do motor, não foram desenvolvidos algoritmos sofisticados de ordenação espacial do grafo de cena. A prioridade foi voltada à arquitetura e aos objetos decoradores e os seus *Handlers*, pois eram os responsáveis pela aprovação ou não dos requisitos fundamentais do projeto.

O uso de animações e transformações comprovou a utilização de *Visitors* em objetos dinâmicos. A utilização de *Vertex Shaders*[5] na água, bem como efeitos especiais como um sistema de partículas[6], *skybox*[7], e *lens flare*[8] demonstrou a flexibilidade e

---

[5] Analogamente à explicação de *Pixel shaders, Vertex Shaders* são programas que são executados no processador da placa gráfica (GPU) uma vez para cada vértice de um objeto tridimensional especificado.

[6] É uma técnica utilizada para simular fenômenos caóticos, dificilmente representados fisicamente.

[7] Um *skybox* é um plano de fundo pré-renderizado de uma cena distante que cerca um ambiente tridimensional. É usado para criar a ilusão de estar cercado por um ambiente distante, como o céu, montanhas, oceano, dentre outros.

[8] É um fenômeno de produção de algumas imagens por sistemas óticos quando apontados a um foco de luz. É causado pela reflexão, refração e dispersão de luz nas lentes de um sistema.

extensibilidade do motor. Testes de qualidade e performance foram feitos em várias máquinas com configurações diferentes, mas todas rodando plataformas de execução equivalentes. O resultado é apresentado nas tabelas e figuras a seguir.

	Processador	Memória RAM	Placa de Vídeo	Memória de Vídeo
Máquina 1	Athlon XP 2800+	512 Mb	NVIDIA GeForce4 MX 440	128 Mb
Máquina 2	Sempron 2800+	512 Mb	NVIDIA GeForce4 FX 5200	128 Mb
Máquina 3	Celeron – 2.2Ghz	512 Mb	NVIDIA GeForce4 MX 4000	128 Mb
Máquina 4	Pentium 4 – 2.8Ghz	512 Mb	Intel OnBoard	64 Mb
Máquina 5	Pentium 4 – 2.0Ghz	512 Mb	NVIDIA GeForce2 MX 400	64 Mb
Máquina 6	Athlon XP 2200+	256 Mb	NVIDIA GeForce4 MX 440	64 Mb
Máquina 7	Sempron 2800+	512 Mb	AMD OnBoard	64 Mb
Máquina 8	Pentium III 800Mhz	128 MB	ATI Mobility	8MB

Tabela 2. Configurações de hardware usados nos testes.

Para uma medição mais correta, durante os testes, apenas a parte gráfica do jogo foi habilitada. Desta forma, processamentos não relacionados (como a lógica do jogo, som ou rede), não influíram na performance apurada. A Tabela 3 apresenta uma análise comparativa dos resultados encontrados nos testes.

	FPS[9]	Modelos	Skybox	Lens Flare	Reflexo	Vertex Shader
Máquina 1	48.25	X	X	X	X	X
Máquina 2	30.00	X	X	X	X	X
Máquina 3	29.97	X	X	X	X	X
Máquina 4	28.60	X	X	X		X
Máquina 5	20.00	X	X	X	X	X
Máquina 6	19.99	X	X	X	X	X
Máquina 7	14.61	X	X	X		X
Máquina 8	1.80	X	X	X		

Tabela 3. Resultados dos testes.

De acordo com o resultado obtido, podemos observar que o motor se comportou de forma bastante satisfatória com relação tanto à qualidade quanto à performance de processamento.

Com relação à performance, as taxas de quadros por segundo (FPS) obtidas foram bastante adequadas ao caráter interativo aplicação. Mesmo considerando que apenas a parte

[9] FPS, do inglês *Frames Per Second*. É uma medida padrão para a medição de performance em aplicações gráficas e indica a quantidade de quadros exibidos durante um segundo.

gráfica do jogo estava sendo executada, nenhuma técnica (como *LOD*[10]) foi utilizada para otimizar a quantidade de polígonos exibida na cena.

Qualitativamente, em todos os casos, as cenas foram renderizadas com todos os modelos, o *skybox* e a *lens flare*.

O problema da não aplicação do reflexo em algumas ambientes é comprovadamente devido à qualidade da placa de vídeo utilizada. O reflexo é obtido através da aplicação de uma textura formada por uma renderização projetada da cena, operação nem sempre suportada pelas placas de vídeo mais simples. Um exemplo prático dessa situação se dá ao comparar a Máquina 2 com a Máquina 7, que em termos de processador e memória são iguais, mas distintas com relação ao hardware de vídeo. Na Máquina 7, tanto a qualidade da cena foi prejudicada como a performance foi afetada, visto que o hardware não era apropriado.

Com relação ao *Vertex Shader*, esse foi usado para a colorir a água, devido à particularidade que essa tarefa possui. O único caso em que o *shader* não pode ser aplicado foi na Máquina 8. Essa, além de ser uma máquina já ultrapassada, possui uma placa de vídeo cujas especificações são impraticáveis à execução de qualquer aplicação gráfica complexa. Assim, esse fato foi desconsiderado, visto que não indica um mau funcionamento do motor.

As figuras a seguir mostram graficamente o resultado da utilização do motor no jogo.

[10] LOD, do inglês *Level of Detail*. É uma técnica usada para diminuir a quantidade de polígonos exibidos em uma aplicação tridimensional, reduzindo o nível de detalhe da cena de acordo com o perceptível pelo observador.

Figura 6. Teste na Máquina 1.

Figura 7. Teste na Máquina 2.

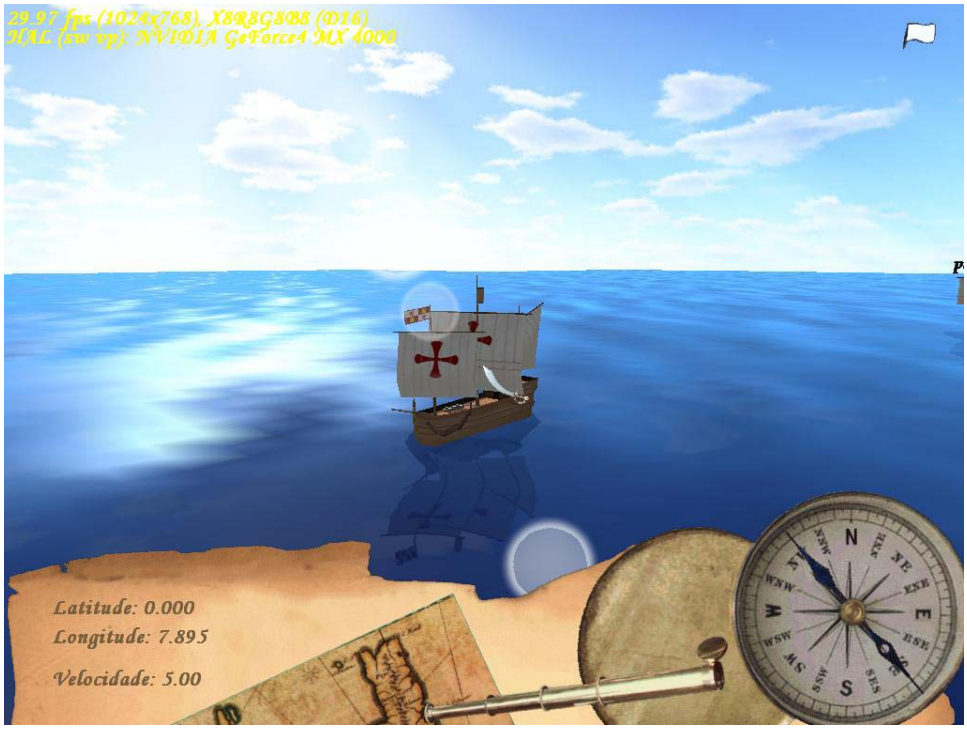

Figura 8. Teste na Máquina 3.

Figura 9. Teste na Máquina 4.

Figura 10. Teste na Máquina 5.

Figura 11. Teste na Máquina 6.

Figura 12. Teste na Máquina 7.

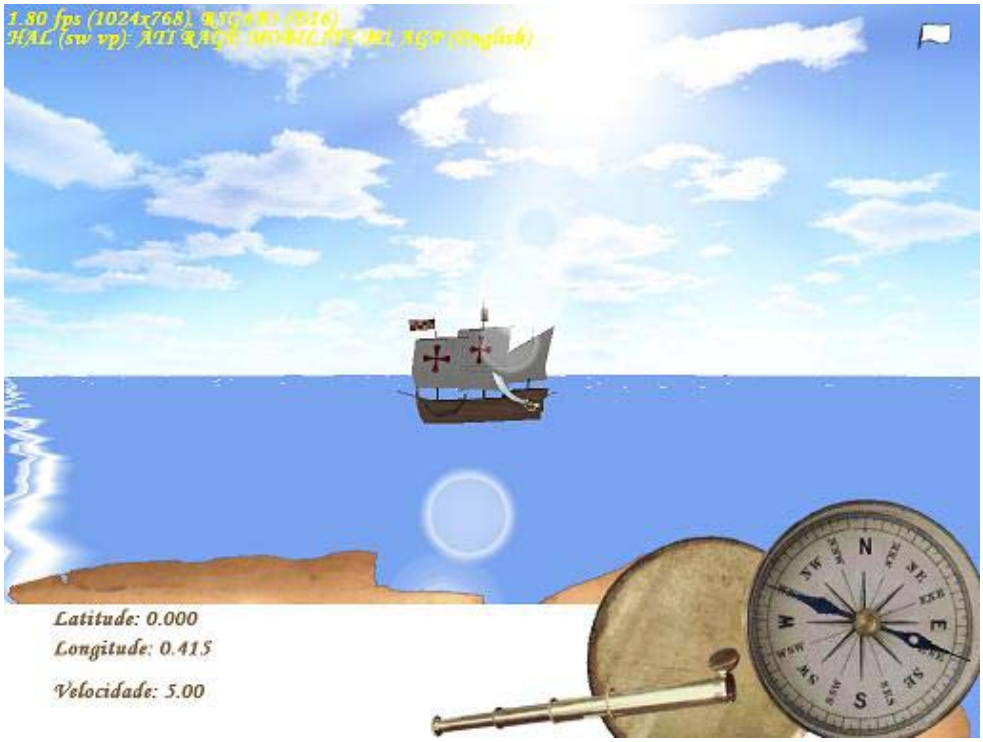

Figura 13. Teste na Máquina 8.

4.2 Dificultades Encontradas

Durante todo o projeto de produção do motor, uma série de dificuldades foram encontradas. Dentre essas, as principais são expostas abaixo:

- Por ser um trabalho pioneiro e pelo segredo de propriedade intelectual do mercado, não havia uma fonte de conteúdo de fácil acesso. Assim, vários testes tiveram que ser implementados com as diversas tecnologias propostas para testar e analisar a ad equação dessas.
- O próprio desenvolvimento do motor 3D é uma tarefa árdua e não trivial, requerendo um grande conhecimento de áreas como computação gráfica e matemática.
- A natureza de um jogo requer o máximo possível de aproveitamento de técnicas que ajudem a melhorar a performance do aplicativo. Com isso, uma série de técnicas e padrões de programação tiveram que ser reconsideradas, alteradas ou otimizadas.
- A falta de estrutura propícia ao desenvolvimento de jogos no Centro de Informática impossibilitou o progresso mais rápido do projeto. O CIn ainda não tem máquinas com hardware capaz de formar um ambiente de execução desse tipo de aplicativo, o que dificultou esse projeto, visto que ele é fruto de um trabalho em conjunto.

4.3 Trabalhos Futuros

O desenvolvimento desse protótipo foi feito utilizando a biblioteca gráfica Direct3D, por motivos já explicitados na seção 4.1. Entretanto, conforme analisado no tópico 2.3, a biblioteca ideal para a implementação do motor é a OpenGL. A mudança entre as tecnologias é um trabalho esperado.

Como citado anteriormente, o desenvolvimento de um motor 3D é um trabalho bastante amplo. O escopo desse projeto se limitou aos problemas de representação de uma cena e detecção de colisões. Assim, a evolução do motor a fim de incluir novas funcionalidades é uma possibilidade concreta.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Tomas Akenine-Möller and Eric Haines. Real-Time Rendering, 2nd Edition. A.K. Peters, 2002.

[2] David H. Eberly. 3D Game Engine Design: A Pratical Approach to Real-Time Computer Graphics. Morgan Kaufmann, 2000.

[3] David H. Eberly. 3D Game Engine Architecture: Engineering Real-Time Applications with Wild Magic. Morgan Kaufmann, 2004.

[4] id Software. http://www.idsoftware.com/, Julho de 2005.

[5] Ives J. A. Macêdo Júnior. mOGE – mObile Graphics Engine: O projeto de um motor gráfico 3D para a criação de jogos em dispositivos móveis. Trabalho de Graduação, Universidade Federal de Pernambuco, 2005.

[6] OpenGL - The Industry's Foundation for High Performance Graphics. http://www.opengl.org/, Agosto de 2005.

[7] Microsoft DirectX: Home Page. http://www.microsoft.com/windows/directx/, Agosto de 2005.

[8] Silicon Graphics – Welcome to SGI. http://www.sgi.com/, Agosto de 2005.

[9] Wolfgang Engel and Amir Geva. Beginning Direct3D Game Programming. Muska & Lipman/Premier-Trade, 2002.

[10] Dave Astle and Kevin Hawkins. Beginning OpenGL Game Programming. Muska & Lipman/Premier-Trade, 2004.

[11] 3d labs. http://www.3dlabs.com/, Agosto de 2005.

[12] Apple Computing, Inc. http://www.apple.com/, Agosto de 2005.

[13] ATI Technologies Inc. http://www.ati.com/, Agosto de 2005.

[14] Dell - Client & Enterprise Solutions, Software, Peripherals, Services. http://www.dell.com/, Agosto de 2005.

[15] Evans and Sutherland. http://www.es.com/, Agosto de 2005.

[16] Hewlett-Packard, http://www.hp.com/, Agosto de 2005.

[17] IBM, http://www.ibm.com/, Agosto de 2005.

[18] Intel Corporation - Welcome to Intel.com. http://www.intel.com/, Agosto de 2005.

[19] Welcome to Matrox. http://www.matrox.com/, Agosto de 2005.

[20] NVIDIA Home. http://www.nvidia.com/, Agosto de 2005.

[21] Sun Microsystems. http://www.sun.com/, Agosto de 2005.

[22] Microsoft Corporation. http://www.microsoft.com/, Agosto de 2005.

[23] UltraShadow Technology. http://www.nvidia.com/object/feature_ultrashadow.html, Agosto de 2005.

[24] Wolfgang Engel. Programming Vertex and Pixel Shaders. Delmar Thomson Learning, 2004.

[25] Mesa Home Page. http://www.mesa3d.org/, Agosto de 2005.

[26] Coderus ltd. http://www.coderus.com/, Julho de 2005.

[27] Game Developers Conference. http://www.gdconf.com/, Julho de 2005.

[28] Microsoft XNA. http://www.microsoft.com/xna/, Agosto de 2005.

[29] XBox 360. http://www.xbox360.com/, Agosto de 2005.

[30] Sony. http://www.sony.com, Agosto de 2005.

[31] PlayStation Global. http://www.playstation.com/, Agosto de 2005.

[32] Gilda Pour. Software Component Technologies: JavaBeans and ActiveX. Proceedings of Technology of Object-Oriented Languages and systems. IEEE Computer Society, 1999.

[33] G.Pour, M. Griss, J. Favaro, Making the Transition to Component-Based Enterprise Software Development: Overcoming the Obstacles – Patterns for Success. Proceedings of Technology of Object-Oriented Languages and systems. IEEE Computer Society, 1999.

[34] George T. Heineman and William T. Councill. Component-Based Software Engineering: Putting the Pieces Together. Addison-Wesley, 2001.

[35] Clemens Szyperski. Component Software: Beyond Object-Oriented Programming. Addison-Wesley, 1998.

[36] Jerry Gao, H-S Jacob Tsao, Ye Wu. Testing and Quality Assurance for Component-Based Software. Artech House, 2003.

[37] Welcome To The OMG's CORBA Website. http://www.corba.org/, Julho de 2005.

[38] COM: Component Object Model Technologies. http://www.microsoft.com/com/, Julho de 2005.

[39] JavaBeans. http://java.sun.com/products/javabeans/, Julho de 2005.

[40] Object Management Group. http://www.omg.org/, Julho de 2005.

[41] Jon Siegel and Dan Frantz. CORBA Fundamentals and Programming. John Wiley, 1997.

[42] Don Box. Essential COM. Addison-Wesley, 1998.

[43] Dale E. Rogerson. Inside COM. Microsoft Press, 1997.

[44] Kraig Brockschimdt. Inside OLE, 2nd Edition. Microsoft Press, 1995.

[45] Microsoft Windows Family Home Page. http://www.microsoft.com/windows/, Agosto de 2005.

[46] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Design Patterns. Addison-Wesley, 1995.

[47] Mark Deloura (Editor). Game Programming Gems. Delmar Thomson Learning, 2000.