Andréia Rodrigues de Assunção Schneider

Análise comparativa de bibliotecas gratuitas e multiplataforma para desenvolvimento de ambientes virtuais

Monografia apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras, como parte das exigências do Curso de Ciência da Computação, para obtenção do título de bacharel

Orientador
Prof. Bruno de Oliveira Schneider

Lavras
Minas Gerais - Brasil
2003

Andréia Rodrigues de Assunção Schneider

Análise comparativa de bibliotecas gratuitas e multiplataforma para desenvolvimento de ambientes virtuais

Monografia apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras, como parte das exigências do Curso de Ciência da Computação, para obtenção do título de bacharel

Aprovada em *18 de Junho de 2003*

Prof. Mário Luiz Rodrigues Oliveira

Prof. Guilherme Bastos Alvarenga

Prof. Bruno de Oliveira Schneider
(Orientador)

Lavras
Minas Gerais - Brasil

Agradecimentos

Agradeço a Deus pelos dons que recebi.

Aos meus pais, irmãos e demais familiares pelo apoio e carinho.

Ao Bruno pela paciência e pela ajuda.

Em especial à Nadir e Vó Carmen pelas orações.

Resumo

Durante a última década muito ouviu-se falar a respeito de Realidade Virtual e software livre. Algumas das pesquisas realizadas nessa época buscaram criar conjuntos de funções facilitadoras da programação (bibliotecas) distribuídas gratuitamente e que servissem para muitas plataformas. Em áreas como realidade virtual, que possuem programas complexos e alto custo de implantação, estas bibliotecas acabam por servir de beneficiadoras, haja vista sua função (facilitar a programação) sem nenhum custo adicional. Este trabalho visa a comparação de três bibliotecas gratuitas para realidade virtual, desenvolvidas para as linguagens de programação C/C++, através da utilização delas para a construção de um ambiente virtual; procurando assim facilitar a escolha feita pelos desenvolvedores de ambientes virtuais sobre qual bibloteca é melhor de usar.

Abstract

During the last decade most people heard about Virtual Reality and free software. Some research made at that time focused on creating freely distributable, multiplataform functions libraries that would ease the task of programming. In areas such as Virtual Reality, which imply complex programs and high cost of implantation, these libraries are much needed, since they ease the programming without any adicional cost. This work compares three free Virtual Reality libraries, for the C/C++ programming languages, through their use on development of virtual environment. It is meant as guiding document for virtual environment developers, by helping them choose which library best fits their needs.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Com a popularização dos computadores, torna-se cada vez mais evidente a importância de interfaces que facilitem as tarefas das pessoas envolvidas em atividades de infomática. A Realidade Virtual (RV) é um tipo emergente de interface de computadores caracterizada por um alto grau de imersão e interação, com o objetivo de fazer o usuário acreditar que está dentro de um ambiente gerado por computador ao invés de ser um observador externo [BELL & FOGLER (1995)]. A RV procura aproveitar o conhecimento natural que as pessoas possuem a respeito de como interagir com o ambiente real para realização de tarefas no computador, através da interação com um ambiente virtual.

A RV começa a se popularizar agora, sendo aplicada nas mais diversas áreas de conhecimento, da engenharia às artes plásticas. Recentemente a UFLA se prepara para entrar no seleto grupo de universidades brasileiras que desenvolvem trabalhos nesta área. A construção do Laboratório de Computação Científica (LCC) e a previsão de instalação de uma estação gráfica de alto desempenho permitirá o desenvolvimento de sistemas para visualização e manipulação de dados gráficos, envolvendo o uso de ambientes virtuais.

O desenvolvimento de programas que envolvem a apresentação de ambientes virtuais requer o uso de ferramentas de programação específicas que facilitem a tarefa de modelagem e apresentação de dados gráficos tridimensionais.

Em geral, bibliotecas de programação (conjuntos de métodos para facilitar a programação) para a modelagem e apresentação de ambientes virtuais são de custo elevado e geralmente destinam-se a uma única plataforma (hardware e sistema operacional). O problema é que cada sistema operacional oferece recursos gráficos diferentes. Isto faz com que um programa desenvolvido para um sistema operaci-

onal não possa ser facilmente portado para outro. Este problema dificulta muito o treinamento de alunos com habilidades suficientes para o desenvolvimento de programas competitivos e de qualidade.

Não é possível prever quais serão as tecnologias dominantes na área de informática daqui a dez anos. Por isso formar profissionais habituados com o desenvolvimento de programas em uma única plataforma fica cada dia mais arriscado, tornando as chances de que tais profissionais se vejam presos a uma pequena porção do mercado.

Não pode ser esquecido que hoje não existe um único padrão para equipamentos de informática e sistemas operacionais. Apesar do Windows ainda ser o mais comum, sistemas livres, como o Linux, vêm aumentando sua utilização a cada ano. É importante que os trabalhos desenvolvidos na UFLA sejam realizados por pessoas atentas à rapidez com que as mudanças acontecem na área de informática. Em trabalhos anteriores [Gomes (2001)] foram analisadas e escolhidas bibliotecas de programação multiplataforma para o desenvolvimento de interfaces gráficas. Estas bibliotecas constituem a base sob a qual são construídos vários programas que podem ser portados para uma variedade de plataformas diferentes como as que usam o sistema operacional Windows ou Linux. Entretanto, elas não são suficientes para o desenvolvimento de programas mais específicos como os que envolvem o uso de técnicas de RV.

1.1 Justificativa

O uso de ferramentas de programação multiplataforma gratuitas, possui diversas vantagens sobre o uso das comerciais ou proprietárias:

1. economia com licenças de uso de software;
2. preparação do corpo discente para atuação nas várias plataformas de computação;
3. maior independência com relação a fabricantes ou distribuidores de software;
4. incentivo a toda a comunidade em torno da universidade a escolher software baseado em suas características e funcionalidades, ao invés de simplesmente seguir a maioria.

Como mencionado anteriormente, está sendo criado na UFLA um Laboratório de Computação Científica (LCC). Neste laboratório serão desenvolvidos trabalhos

que beneficiarão todos os departamentos da UFLA. Para que se inicie o desenvolvimento de aplicações de RV, é importante que se faça uma análise comparativa das várias bibliotecas gráficas disponíveis para que se identifique a melhor.

É importante que as bibliotecas utilizadas sejam multiplataforma pois, no futuro, as empresas de desenvolvimento mais bem sucedidas serão aquelas que mais facilmente puderem se adequarem à diversidade de opções na área de sistemas operacionais. Atualmente é possível verificar que empresas ligadas ao desenvolvimento multiplataforma começam a aparecer em toda parte, passando de pequenas desconhecidas no mercado de software a grandes corporações.

Além de possibilitar o desenvolvimento de programas interativos de visualização de dados em três dimensões, estas bibliotecas poderão ser também utilizadas no desenvolvimento de projetos ligados à área de entretenimento, que movimenta milhões de dólares anualmente em todo o mundo.

Cabe aqui observar que o setor de entretenimento vem ocupando uma fatia crescente e importante na industria de software. O Brasil ainda não participa significativamente deste mercado e estimular o desenvolvimento de jogos vem de encontro aos interesses governamentais de aumentar a exportação de software. Além disto, a criação de grupos de desenvolvimento de programas de entretenimento já pode ser observada no meio acadêmico, como em diversas universidades no Brasil, como por exemplo a UFSCar, e no exterior.

Com a seleção de tais bibliotecas, ocorrerá uma maior capacitação de pessoal que irá trabalhar no LCC, desenvolvendo aplicativos gráficos avançados e fazendo total proveito dos recursos de hardware oferecidos pela estação gráfica deste laboratório.

1.2 Objetivos

Três bibliotecas de programação multiplaforma, gratuitas, para modelagem e apresentação de ambientes virtuais foram analisadas comparativamente. O trabalho desenvolvido irá nortear o desenvolvimento de software de RV dentro da UFLA.

De maneira mais geral, espera-se que este projeto contribua para:

- estimular a cultura de desenvolvimento multiplataforma;
- estimular a programação com código aberto.

Capítulo 2

Referencial teórico

2.1 Realidade Virtual

2.1.1 O que é Realidade Virtual

A realidade virtual é uma nova área da computação, que busca novas formas de comunicação entre o homem e a máquina.

Uma das definições encontradas para o termo Realidade Virtual é: "uma interface homem-máquina, assim como outras interfaces já existentes, tais como linhas de comando e menus". A idéia geral é transportar o conhecimento natural de interação, que as pessoas adquirem no mundo real.

Uma das vantagens da realidade virtual sobre outras formas de interação homem-máquina é que o ambiente pode ser visualizado a partir de qualquer ângulo, à medida que vão sendo feitas alterações em tempo real. Comportamentos e atributos podem ser dados a objetos pertencentes ao ambiente, o que propicia a simulação de respostas e funções do mundo real enfocado.

O ser humano está constantemente em contato com informações diferentes do ambiente ao seu redor, precisando, pois, assimilá-las através de processamento. Desde a infância, porém, ele aprende que possue cinco sentidos para interpretar estas informações. Entretanto, quando lidamos com o computador, estas idéias não tem mais senso prático, devido ao fato de as interfaces e também os meios disponíveis de interação com o usuário serem limitados.

A realidade virtual tenta quebrar essa barreira, procurando transportar todo o conhecimento natural, adquirido pelas pessoas, para o computador, fazendo com que qualquer ser humano interaja com elementos computacionais como se estivesse lidando com elementos do seu cotidiano. Desta maneira percebe-se que o

grande desafio desta área é fornecer um ambiente sintético no qual o usuário sinta um grau de realismo tão grande que fique à vontade para trabalhar, como se estivesse em um ambiente real.

2.1.2 Origem da Realidade Virtual

Esta nova área da computação, chamada de realidade virtual, tem origem em três outras áreas:

- Computação gráfica interativa: devido ao grande avanço do hardware e do software, os programas feitos podiam incluir cada vez mais detalhes, até chegar às descrições de objetos tridimensionais, podendo então imitar uma cena real.

- Sistemas de teleoperação: Tarefas arriscadas como trabalhos em altas profundidades ou em usina nucleares causaram o aparecimento de robôs teleoperados, guiados a distância. Para que o trabalho fosse realizado com maior precisão e sucesso, as interfaces foram evoluindo, até levar à reprodução do ambiente aonde o robô deveria operar.

- Simuladores de veículos: a construção de protótipos de veículos era sempre bastante dispendiosa e oferecia perigo aos pilotos de prova. Assim, houve o aprimoramento dos simuladores para que certos fenômenos físicos e naturais pudessem entrar na simulação, oferecendo assim um ambiente de prova tal qual o real.

2.1.3 Características da Realidade Virtual

As principais características de um sistema de realidade virtual são:

- Imersão: o usuário deve possuir a sensação de que está presente no ambiente, ou seja, o ambiente deve ser percebido como se ele fosse real (verdadeiro). Deve-se ter em mente que as pessoas usam principalmente o sentido de visão e audição para ter a sensação de que um ambiente é real, de maneira que som e imagem estéreos são os principais elementos para se ter um ambiente imersivo.

- Interação: um sistema de realidade virtual deve sempre responder aos estímulos do usuário, seja alterando características de seus objetos (deformação) ou enviando estímulos de resposta ao usuário.

- Envolvimento: O desejo do usuário em participar do ambiente deve ser estimulado. Por isso é tão importante a sensação de real a ser passada pelo ambiente através da apresentação de um ambiente realístico, bem como um meio de interação natural.

2.1.4 Aplicações da realidade virtual

A realidade virtual pode ser aplicada nas mais diversas áreas do conhecimento e de forma bastante diversificada. A todo instante surgem novas aplicações devido à demanda e à criatividade do ser humano. Dentre as aplicações existentes podemo citar:

- Visualização científica: visualização de superfícies planetárias, síntese molecular.
- Aplicações médicas e em saúde: simulação cirúrgica, ensino de anatomia, saúde virtual, visualização com realidade aumentada, tratamento de fobias.
- Educação: laboratórios virtuais, ensino de geometria, educação à distância.
- Treinamento: simuladores de vôo, treinamento de astronautas.

2.2 Critérios para se desenvolver um ambiente de Realidade Virtual

A principal característica de um sistema de Realidade Virtual é o envolvimento humano através da imersão sensorial. Com parâmetros humanos envolvidos no sistema, a sua avaliação torna-se subjetiva, mas essencial em função das questões tecnológicas, da qualidade da aplicação, e do impacto psicológico e social.

A avaliação do sistema de realidade virtual deve garantir que:

- As capacidades e limitações dos seres humanos, bem como as necessidades específicas de determinadas tarefas estão sendo consideradas no projeto do sistema;
- o hardware e o software estarão fornecendo o ambiente virtual com bom índice relacionado com custo e benefício.

Deve ser considerado também:

- a atuação dos dispositivos e os fatores ergonômicos gerais;
- os aspectos gráficos e sua influência na visão;
- a influência na taxa de quadros por segundo;
- a percepção de profundidade;
- a discriminação das cores visuais;
- os aspectos visuais;
- as questões auditivas, de tato e de força;
- o comportamento, o desempenho e as consequências da simulação.

Kalawsky [1], citado por Martins & Kirner [Martins & Kirner (1997)], defende que um ambiente virtual típico deve agregar características que o tornem:

- Sintético: O ambiente deve ser gerado em tempo real por um sistema computacional.
- Tri-dimensional: O ambiente que cerca o usuário é representado em 3D e existem recursos que dão a idéia de profundidade ao ambiente e que o usuário pode mover-se através dele.
- Multisensorial: Mais de uma modalidade sensorial deve ser usada para representar o ambiente, tais como sentido visual, sonoro, de profundidade, etc.
- Interativo: O programa computacional deve ser capaz de detectar as entradas do usuário e modificar instantaneamente o mundo virtual e as ações sobre ele.
- Realístico: O ambiente virtual deve reproduzir com precisão os objetos reais, as interações com o usuário e o próprio modelo do ambiente.
- Com presença: O usuário deve sentir como se estivesse dentro do ambiente.

Como qualquer sistema de software, a construção de ambientes virtuais requer um processo de desenvolvimento adequado às suas peculiaridades. Porém, devido ao fato da área de realidade virtual ser muito recente, ainda não existem na literatura trabalhos suficientes sobre como desenvolver tais aplicações.

[1] KALAWSKY, R. S. The Science of Virtual Reality and Virtual Environments. Addison-Wesley Reading, 1993

2.2.1 Requisitos de um sistema de Realidade Virtual

Um sistema de realidade virtual de grande porte é caro e complexo, em função de todos os recursos envolvidos. Para que o projeto obtenha sucesso, é necessário que sejam satisfeitos um conjunto de requisitos. Para este trabalho os requisitos a serem satisfeitos são:

- **Requisitos de interface com o usuário**

    1. Um sistema de realidade virtual deve reagir rapidamente às ações do usuário.
    2. Um sistema de Realidade Virtual deve se aproximar ao máximo da realidade. Por isso, quanto mais detalhes do mundo real estiver no ambiente, mais realista ele será.

- **Requisitos de engenharia de software**

    1. Portabilidade das aplicações: As aplicações deverão ter facilidades para execução em diversas instalações, exigindo no máximo uma recompilação do código.
    2. Suporte para uma larga faixa de dispositivos de entrada e saída de dados: Como a tecnologia de hardware para a realidade virtual ainda se encontra em expansão, o sistema deve ser capaz de acomodar novos dispositivos.
    3. Independência das aplicações com relação à localização física do usuário e de seus dispositivos de entrada e saída de dados: O sistema deve ser capaz de se adequar a diferentes configurações de localização física do usuário e de seus dispositivos de E/S.
    4. Flexibilidade de ambiente de desenvolvimento de aplicações de realidade virtual: O sistema deve ser flexível o suficiente para que permita a utilização de ambientes de desenvolvimento diferentes, bem como a execução de testes com outros dispositivos.

A montagem de um sistema de realidade virtual requer um cuidadoso planejamento, em função da variedade de componentes, qualidade destes e preços no mercado. Por causa disto a definição desejável de um ambiente virtual deve satisfazer uma série de requisitos e características citadas a seguir[Kirner & Ipólio (1996)]:

- Definição da aplicação;

- Caracterização da imersão;
- Avaliação dos dispositivos de visualização;
- Estabelecimento das capacidades de rastreamento;
- Avaliação de outros dispositivos de E/S;
- Avaliação do conjunto de recursos e capacidades;
- Seleção do sistema de desenvolvimento de realiadade virtual: Criação e edição da geometria; criação e edição de texturas; requisitos de programação; caracterização da visão estereoscópica; modelagem do comportamento físico; suporte a periféricos; requisitos do sistema; portabilidade; suporte de rede; suporte de distribuição;
- Seleção do hardware: Quantidade e característica das portas e slots; características do acelerador gráfico; Conversores de sinais de vídeo; Capacete de visualização (HMD); monitor externo; Óculos estereoscópico; rastreadores/posicionadores; navegadores 3D; luvas e dispositivos de força; outros dispositivos especiais.

2.3 Critérios para se comparar bibliotecas

A programação de Realidade Virtual requer conhecimentos de sistemas de tempo real, orientação a objetos, redes, modelagem física, entre outros. Para facilitar essa tarefa, empresas e algumas universidades produziram sistemas de desenvolvimento de realidade virtual. Esses sistemas são bibliotecas ampliáveis de funções orientadas a objetos, voltadas para especificações de realidade virtual, onde um objeto do mundo passa a ser uma classe e herda seus atributos padrão, o que simplifica a tarefa de programar mundos complexos.

É muito difícil encontrar na literatura existente estudos comparativos de bibliotecas, pois as pessoas costumam utilizar aquela biblioteca que é considerada por elas a mais fácil de ser manipulada, sem pensar em requisitos como custo e benefício, existência de documentação para facilitar o aprendizado e maior quantidade de recursos disponíveis.

Segundo Mário Marcelo Guimarães Ribeiro [2] para a comparação de bibliotecas usadas na construção de mundos virtuais, deve-se levar em conta os seguintes

[2]RIBEIRO, M. M. G. (mario@labtin.logicway.com.br). A dúvida. E-mail para FREITAS, R. M. (rmf@triang.com.br). 6 de Novembro de 2002

aspectos:

- Facilidade na obtenção da biblioteca e direito proprietário: Existem bibliotecas disponíveis na internet e que são gratuitas. Algumas dessas são fruto de pesquisas em universidades.

- Documentação existente: Quanto mais ampla a documentação, mais fácil o aprendizado. Ela deve ser rica em exemplos e demonstrações, além de possuir um tutorial passo a passo.

- Portabilidade entre plataformas: muitas vezes um sistema de realidade virtual é desenvolvido em uma plataforma, mas é utilizado em outra.

- Desempenho de sistemas desenvolvidos com esta: O sistema deve ter um bom desempenho (velocidade de processamento, pouca demora entre a entrada de dados e a saída) para ter um realismo.

- Qualidade de sistemas desenvolvidos com esta: quanto maior a qualidade de um sistema, maior seu realismo.

- Aceitabilidade por parte da comunidade desenvolvedora (se alguem mais usa e tem mais experiência na área).

- Facilidade de uso: Nem sempre o melhor é o mais fácil, ou o pior é o mais difícil. À vezes é melhor gastar um tempo maior estudando a biblioteca para ter melhores resultados gráficos.

- Recursos disponíveis na biblioteca: Quanto mais recursos disponíveis para uso, melhor será o sistema.

Quanto mais conhecimento se tiver a respeito das bibliotecas a serem comparadas, melhor e mais fácil será esta comparação. Por isso é de extrema importância o estudo aprofundado delas através do desenvolvimento de ambientes virtuais simples e também complexos.

Outro fator importante a ser observado são os recursos mínimos necessários para que o ambiente criado com determinada biblioteca possa ser executado. Algumas bibliotecas exigem recursos caríssimos, o que para algumas aplicações seria inviável.

2.4 Bibliotecas de RV

2.4.1 Maverik

Desenvolvido pela Universidade de Manchester (*University of Manchester*), Maverik - *MAnchester Virtual Environment Interface Kernel* é um sistema de gerenciamento gráfico e interação em aplicações de realidade virtual. Ela foi criada para atender aos desafios de ambientes virtuais de alta interatividade, contendo muitos objetos com complexa geometria. Esta biblioteca é, na realidade, um *toolkit* (para C) para gerenciamento de disposição e interação de aplicações de ambientes virtuais para um usuário apenas e não ligadas à rede. Existe um sistema complementar ainda em desenvolvimento, para a criação de ambientes multi-usuário e conectados à rede.

A distribuição de Maverik é gratuita, pois esta biblioteca é componente oficial da *Free Software Foundation's GNU Project* [Open Source (2002)].

Existem várias bibliotecas para realidade virtual disponíveis, desde as de baixo nível, para desenhar gráficos em 3D e interagir com periféricos, até bibliotecas de altíssimo nível, como os editores próprios para realidade virtual. Maverik está entre estes extremos: ela provê uma aplicação com as ferramentas necessárias para criar, ver, interagir e navegar no ambiente enquanto faz o mínimo de suposições sobre a natureza da aplicação. O importante de se observar é que Maverik não é um aplicativo para usuário final: não há interface gráfica para o usuário, nem editores de mundo. Ela é estritamente uma ferramenta para programação.

Maverik tem dois componentes:

1. Um micro-kernel, que provê a estrutura aonde a aplicação será construída;
2. uma coleção de módulos de suporte, os quais provêm gerenciamento de disposição, gerenciamento espacial, interação e técnicas de navegação, controle de dispositivos de entrada e saída otimizados.

Esta biblioteca é disponibilizada como código fonte e pode ser compilada em sistemas Windows, MacOS e UNIX - essencialmente qualquer sistema que tenha OpenGL, Mesa, IrisGL ou DirectX. Ela é conhecida por rodar nos seguintes sistemas operacionais:

- SGI Irix 5.3, 6.3 e 6.5;
- RedHat 5.2 e 6.x;
- FreeBSD 3.2;

- SuSE 7.1;
- SunOS 5.7;
- Windows 98, 2000 e NT;
- MacOS.

Uma compilação padrão de Maverik provê suporte para mouse, teclado e monitor. Porém está incluído no código a distribuição para suportar alguns periféricos 3D tais como *trackers Polhemus FASTRAK* e *ISOTRAK II* com seis graus de liberdade, *Magellan space mouse*, etc. O interesse deste trabalho está apenas no suporte dos periféricos padrões, haja visto que são os únicos disponíveis para a execução deste projeto.

O desenvolvimento de Maverik começou em 1997. Desde então ela tem sido usada em diferentes projetos e aplicações, incluíndo: pesquisas no provimento de interfaces para tarefas complexas em engenharia, tais como o projeto e operação de plataformas de petóleo, modelagem estereoscópica de cenas de crime, visualização de dados abstratos, visualização de simulações baseadas na física, modelagem de nanotecnologia, modelagem arquitetural, etc [GNU Maverik (2002)]. Pode-se observar um ambiente criado com Maverik na Figura 2.1.

A biblioteca Maverik contém mais de 550 funções. Didaticamente elas foram divididas em três níveis:

1. Nível 1: contém funções simples de serem utilizadas, as quais fazem uso de construtores padrão da biblioteca e capacita usuários a criar aplicações de maneira rápida.

2. Nível 2: contém funções que permitem um uso mais avançado da biblioteca. Exemplos podem incluir definição de novas classes (orientação a objetos), ou definição de novos métodos de navegação no ambiente virtual.

3. Nível 3: contém funções para pesquisa e desenvolvimento de ambientes mais complexos. São funções de baixo nível que provêm interfaces para o Kernel do Maverik e módulos associados. Por exemplo, as funções do nível três poderiam serem requeridas para a criação de novos algoritmos de processamento de nível de detalhes, ou para prover suporte para novos tipos de dispositivos de entrada.

Maverik foi desenvolvida para ser um sistema de realidade virtual o qual possui dois conceitos chaves: customização fácil para alcançar as demandas de diferentes

aplicações e operação eficiente, pois só assim grandes ambientes podem ser alcançados. Ela adota um design de micro-kernel que minimiza suposições sobre como ambientes são representados e armazenados pelo sistema.

O sistema Maverik

Assim como o OpenGL, Maverik pode ser vista como um biblioteca gráfica que se liga à uma aplicação e diretamente usa suas estruturas de dados e algoritmos. A diferença crucial é que ela também define uma estrutura padrão no qual uma aplicação fornece Maverik com as maneiras de se acessar seus objetos. Através do uso desta estrutura, Maverik pode fornecer alto nível de funcionalidade sem ditar o uso de qualquer representação específica de um objeto.

Sua implementação é em C. Desta forma ela pode ser facilmente portável para diferentes plataformas e ser usada por pessoas com conhecimentos básicos em C ou C++. Métodos são implementados através do uso de funções callback, com dados passados por ponteiros sem tipo.

Para evitar ter que escrever funções *callback* todas as vezes que uma nova aplicação for implementada, Maverik fornece métodos padrão para as primitivas mais comuns, como poligonos, esferas, círculos, etc. Estes métodos padrão são distribuídos como código fonte, fornecendo um conjunto de exemplos e facilitação da customização.

Devido ao fato dos objetos em Maverik apenas manterem ponteiros para (e não cópias de) estruturas de dados e classes, eles não precisam ser notificados de qualquer mudança com eles próprios, o que é considerada uma vantagem.

O design de Maverik fornece uma interface limpa, que possibilita customização para ser configurado dinamicamente em tempo de execução. Este design adotado possue três vantagens [GNU Maverik (2002)]:

- Primeiramente, nenhum dos dados da aplicação é importado para (ou replicado dentro de) Maverik. Isto evita o problema de troca de sincronismo para representações múltiplas;

- Em segundo, a estrutura encapsula todas as informações necessárias para Maverik acessar dados e métodos armazenados externamente à aplicação. Desta maneira classes podem ser facilmente reusadas em outras aplicações.

- Em terceiro, é simples de entender e usar, além de ser fácil de ligar às aplicações já existentes. Este último ponto é importante em domínios do tipo

Figura 2.1: VRAD - A Radiosity system for VR using Maverik

CAD, aonde existem bancos de dados grandes e códigos já existentes que não podem ser dispensados.

2.4.2 VR Juggler

VR Juggler é uma estrutura de aplicação e um conjunto de classes em C++ para a implementação de aplicações em realidade virtual. Ela foi desenvolvida para permitir ao desenvolvedor acesso direto a vários gráficos API para o máximo de controle em cima das aplicações, enquanto ainda fornece um resultado (visão do ambiente gerado) fácil de ser entendido. Ela comecou a ser desenvolvida em Junho de 1996, pela Dr. Carolina Cruz-Neira e uma equipe de estudantes do centro de aplicações de realidade virtual, da Universidade estadual de Iowa (Iowa State University).

Esta biblioteca suporta uma grande variedade de hardware de realidade virtual, incluindo sistemas de projeção como o CAVE e o C2, mesas de projeção, *flock of birds*, *mouse 3D Logitech*, entre outros. O suporte para dispositivos de entrada inclui caixa de imersão, luvas (*CyberGlove*). Um conjunto de dispositivos de interface genéricos permite que novos dispositivos sejam adicionados facilmente, além de existirem simuladores para algum dispositivo não disponível. Ela também inclui suporte para aplicações distribuídas.

Ela é distribuída e licenciada como código aberto, de acordo com as regras do *Open Source* [Open Source (2002)].

Podem ser citados como exemplos da utilização desta biblioteca: CuevaDeFuego, CFD in VR, Cave Quake]I[Arena (vide Figura 2.2), MetVR [VR Juggler (2002)].

Vr Juggler é dividida em duas partes:

- Um micro kernel: controla o sistema inteiro em tempo de execução e gerencia todas as comunicações dentro do sistema. Os benefícios de sua exis-

Figura 2.2: Cave Quake]I[Arena

tência são maior portabilidade, flexibilidade e extensibilidade, enquanto o custo é a complexidade de implementá-lo, o que não é uma tarefa trivial.

- Plataforma virtual (JVP): seu propósito é separar os componentes de software do sistema de realidade virtual dependentes de hardware dos independentes. Ela fornece um ambiente operacional simples para o desenvolvimento da aplicação. A plataforma virtual é independente de arquitetura de hardware, sistema operacional e hardware de realidade virtual disponível. Assim, o desenvolvedor pode escrever uma aplicação em um sistema e executá-la em outro. Devido a esta independência de sistema operacional o VR Juggler é uma bibloteca verdadeiramente multiplataforma.

2.4.3 PLIB

PLIB é um conjunto de bibliotecas para C++ e Java, criadas por Steve Baker e equipe, para implementação de jogos e outras aplicações interativas em tempo real. Este conjunto de bibliotecas incluem efeitos sonoros, música e funções matemáticas em 3D, entre outros.

Para uma boa visualização gráfica em 3D, é necessário um acelerador gráfico compatível com OpenGL que seja suportado pelo sistema operacional usado. Muitos sistemas podem rodar OpenGL em software, porém a preformance da aplicação torna-se ruim. Para a utilização de efeitos sonoros e música é necessária uma placa de som.

A tentativa da equipe criadora da PLIB foi minimizar a dependência desta de outras bibliotecas com o intuito de facilitar para os usuários finais a instalação do software da aplicação. Desta forma não é necessário que o usuário visite várias páginas na internet para coletar as bibliotecas que seriam necessárias. Porém é necessário que se tenha OpenGL ou Mesa e GLUT (*OpenGL Utility Toolkit*, criada por Mark Kilgard), que é uma biblioteca com suporte para mouse e teclado. Dependendo de como for a aplicação desenvolvida, não é necessário o uso da GLUT.

A portabilidade de PLIB é excelente. Todos os tipos de Windows e linux são suportados, além de BSD, IRIX, Solaris, OS-X. MacOS roda tudo, exceto a biblioteca de *joystick*. O suporte para BeOS ainda é problemático.

A distribuição de PLIB é gratuita e licenciada de acordo com as regras da Library GNU Public License (LGPL) [FSF.GNU (2002)].

Vários são os projetos que utilizam PLIB. Podem ser citados como exemplos[3]: Tux the Penguin - A Quest for herring , TuxKart, FlightGear (vide figura 2.3), Majik3D,TORCS, Flight Dynamics Simulator, entre outros.

PLIB é compreendido de um número de bibliotecas semi autônomas que podem ser misturadas usando o quanto for necessário da PLIB. As bibliotecas componentes são [PLIB (2002)]:

1. Picoscopic User Interface(*PUI*): é um simples conjunto de classes de C++ que permitem programas escritos em OpenGL e GLUT criarem botões, menus, entre outros dispositivos de interface gráfica.

2. Sound Library (*SL*): biblioteca de áudio para C++, GLUT e aplicações em tempo real. Inclui um simples MOD music loader/player.

3. Standard Geometry Library(*SG*): é um conjunto de funções matemáticas para matrizes e vetores que foi escrita especificamente para simplificar a escrita de programas eficientes em OpenGL.

4. Simple Scene Graph Library (*SSG*): biblioteca para cenas gráficas. Também contém código para carregar e para conservar lotes de arquivos em formato 3D.

[3]Disponíveis, respectivamente em: http://tuxaqfh.sourceforge.net/, http://tuxkart.sourceforge.net/, http://www.flightgear.org, http://www.majik3d.org, http://www.torcs.org, http://www.flight-dynamic-simulator.de, citado em 25 de Novembro de 2002

Figura 2.3: flightGear - simulador de vôo

5. SSG Auxiliary Library (*SSGA*): Funcionalidade adicional para a SSG - nem todo programa que utilize SSG necessitará dela, mas em alguns casos ela poupa esforços de programação.

6. Joystick wrappers (*JS*): suporta Joysticks com mais eixos e botões em relação à GLUT.

7. Fonts'n'Text Library (*FNT*): suporta saída de texto em OpenGL usando fontes mapeadas por textura.

8. Utility Library (*UL*) : biblioteca portável para incompatibilidades básicas do sistema operacional.

9. Pegasus Network Library (*NET*): Pegasus é uma biblioteca em C++ que auxilia o programador na adição de rede nos jogos, ou seja, ajuda na criação de jogos para a rede.

De acordo com Steve Baker [4]

> Não há tutorial explícito para PLIB, porém há vários programas exemplos que vão junto com a biblioteca, os quais auxiliam na aprendizagem da biblioteca.

Isto, entretanto, pode dificultar o aprendizado da PLIB.

[4]BAKER, S. (sjbaker1@airmail.net). Re: [Plib-users] plib tutorial. E-mail para SCHNEIDER, A. R. A. (andreia@comp.ufla.br). 7 de Maio de 2002

Capítulo 3

Metodologia adotada

A primeira etapa do projeto foi pesquisar quais das bibliotecas disponíveis são multiplataforma e gratuitas. Dentre as bibliotecas encontradas foram escolhidas três para uma análise comparativa, levando-se em consideração nesta escolha a quantidade de trabalhos já desenvolvidos com as mesmas.

Uma das três bibliotecas escolhidas inicialmente foi a MR Toolkit[1], desenvolvida pelo Grupo de Realidade Virtual da Universidade de Alberta, no Canadá. Porém, este projeto foi abandonado devido à mudança para Hong Kong de seu principal mentor: Dr. Mark Green. Desta forma ela foi substituída pela biblioteca PLIB.

Em uma segunda etapa, as bibliotecas foram usadas para o desenvolvimento de um ambiente virtual, no caso um parque, composto por árvores, bancos, e um parquinho com balanços, escorregador e uma gangorra que se move, o qual utilizou técnicas de computação gráfica (forma dos objetos, iluminação.), com o intuito de fornecer o máximo de realismo ao sistema, e suas diferenças foram registradas com relação aos seguintes aspectos:

- facilidade de uso
- facilidade de obtenção
- documentação disponível
- eficiência do programa produzido
- adequação aos princípios do estilo de programação orientada a objetos

[1] `http://http://www.cs.ualberta.ca/~graphics/MRToolkit/`

- abrangência das várias plataformas computacionais existentes
- tamanho do programa produzido
- nível de abstração e abrangência das funções e métodos implementados
- recursos disponíveis

Estes aspectos foram escolhidos após um estudo sobre como se comparar bibliotecas para desenvolvimento de ambientes virtuais, ou seja, quais as características que são fatores importantes de diferenciação entre as bibliotecas.

Após a criação desses ambientes e com base na análise entre as três bibliotecas, pôde-se indicar qual das três é considerada a melhor para determinados casos, de acordo com os critérios acima descritos.

Para o presente trabalho o hardware utilizado foi um computador Athlon XP 1.5Ghz, com 256M de RAM e uma placa de vídeo ATI RAGE 128, com 32M de RAM. O monitor é um Samsung SyncMaster 753DFX, de 17 polegadas. Não houve uso de outros dispositivos como HMD, luvas, dispositivos de força, etc. Os ambientes gerados são não-imersivos e a saída é através do monitor. A entrada de dados é através do teclado e do mouse.

Desta forma, o trabalho foi primeiramente o estudo teórico das funções existentes nas bibliotecas escolhidas, com implementações de dispositivos simples de realidade virtual, e depois o estudo prático através da construção de ambientes virtuais mais rebuscados e utilizadores do máximo de recursos dessas bibloteca. Foi através deste estudo prático que se tornou possível fazer considerações úteis a respeito de cada biblioteca.

Capítulo 4

Análise das bibliotecas

4.1 Maverik

Qualquer aplicação Maverik tem sempre a mesma estrutura lógica simples, compreendida por:

1. inicialização do Maverik;
2. definição dos objetos que compõem o mundo virtual;
3. definição das características da aplicação (como os objetos são gerenciados, respostas às iterações, definição da navegação.);
4. loop de renderização (uma vez inicializado o loop, este nunca pára até o fechamento da aplicação). Em cada ciclo são checados os eventos dos dispositivos de entrada e desenhado o ambiente.

Maverik provê um *framework* que permite customização de métodos de gerenciamento espacial. De uma maneira análoga a definição de objetos, Maverik usa classes (*structs*) e métodos para armazenar e acessar um dado espacial. No geral uma aplicação define uma classe para cada técnica de armazenamento de objeto, registra as funções correspondentes para os diferentes métodos de cada *struct* e define a estrutura de gerenciamento do objeto, chamada estrutura de gerenciamento espacial ou *Spatial Management Structure (SMS)*, para armazenar e gerenciar objetos.

Esta biblioteca possui objetos primitivos padrão (cubo, cilindro, esfera e outros) já definidos, com várias funções implementadas, sendo as principais:

- função de renderização do objeto;
- função de identificação do objeto;
- função que acessa a matriz de transformação ;
- função que acessa os parâmetros de renderização (cor, textura material, etc.).

Através destes objetos, qualquer outro pode ser definido. Todos os objetos são definidos por dois atributos em comum (além de outros diferentes para cada um), um que determina o conjunto de parâmetros usados na sua renderização, tais como cor, luminosidade, etc, e outro que é a matriz de transformação, a qual mapeia o sistema de coordenadas local do objeto dentro das coordenadas do mundo usadas na aplicação, determinando sua posição, escala e orientação.

O objeto utilizado para criar figuras compostas a partir dos objetos padrão chama-se SMSObj. Ele contém um SMS com objetos que são primeiro transformados por uma matriz de transformação comum, para depois serem transformados por suas próprias matrizes de transformação. Objetos já feitos em formato de arquivo VRML97, Lightwave e AC3D podem ser aproveitados no Maverik através da leitura de seu arquivo. Isto pode ser considerado uma vantagem, pois o programador pode reutilizar objetos já feitos em outros formatos, poupando-lhe tempo de implementação (ao invés de criar o objeto, ele apenas indica o local onde ele deve ficar e manda o Maverik interpretar o arquivo passado).

O *loop* de renderização tem a seguinte estrutura no Maverik:

- É verificado se há algum evento de interação;
- após isto, começa a criação de uma nova janela;
- o próximo passo é a execução do restante do programa, com a renderização dos objetos definidos;
- Por último a aplicação informa que a janela está completa e pronta para ser exibida.

A forma a qual um objeto é renderizado é controlada pelos parâmetros de superfície chamados *surface parameters* (cor, textura, luminosidade, etc.), que são indexados em uma paleta de renderização. Uma paleta contém uma especificação do ambiente de luz, uma tabela de luminosidade, uma tabela de cor, uma de material, uma de textura e uma de fonte. Ao se criar uma janela associa-se automaticamente à mesma uma paleta. Porém nada impede que o programador crie uma nova

paleta usando funções já existentes no Maverik. Os parâmetros de superfície de um objeto se referem às entradas da paleta associada à janela na qual o objeto é renderizado. São eles:

- modo de renderização: especifica como a cor, textura e material do objeto serão renderizados. Maverik tem quatro modos de renderização:
  - MAV_COLOUR: o objeto tem uma cor uniforme, obtida da tabela de cor da paleta.
  - MAV_MATERIAL: o objeto tem propriedades de reflexão ambiente, difusa e especular do material, obtido da tabela de material da paleta.
  - MAV_TEXTURE: o objeto tem uma textura mapeada na sua superfície, obtida da tabela de textura da paleta.
  - MAV_BLENDED_TEXTURE: o objeto tem uma cor que é a interpolação entre as cores do material e da textura, regida pelo valor de alfa da textura. Se alfa = 0, cor = material; se alfa = 1, cor = textura.
- cor: indica a cor do objeto
- material: indica o material do objeto
- textura: indica a textura do objeto. Maverik suporta apenas arquivos de texturas com extensão PPM e PNG.

Como descrito no guia de programação do Maverik [GNU Maverik (2002)], seu modelo de visão é baseado no modelo de câmera sintética. A aplicação define:

- Um ponto de vista (*eyepoint*): a posição da câmera nas coordenadas do mundo,
- uma direção de visão (*view direction*): um vetor normalizado indicando a direção de visão para a câmera,
- view up:um vetor que aponta a direção para cima da câmera (observador).

O padrão de Maverik para o modelo de visão é a câmera no eixo z positivo, olhando para baixo no eixo z, em direção a origem e direção paralela ao eixo y do mundo. Uma aplicação pode ter mais de uma câmera. Para isto é necessário criar a função que define cada câmera e usar os eventos do teclado para mudar de câmera. As coordenadas do mundo em Maverik podem ser visualizadas conforme a figura 4.1

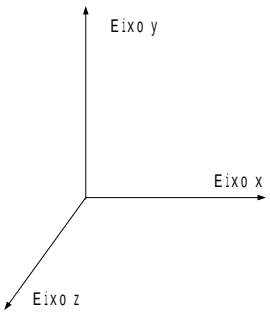

Figura 4.1: Sistema de coordenadas do mundo em Maverik, visto conforme a posição inicial da câmera

Uma aplicação define as ações que devem ser tomadas, quando eventos do mouse e do teclado ocorrem, usando um mecanismo de *callback*. Eventos do mouse são definidos para um botão específico, enquanto que eventos do teclado são definidos para qualquer tecla. Maverik possue uma função de navegação padrão para o mouse. Este padrão comporta da seguinte maneira: quando o botão da esquerda é pressionado há a movimentação para frente e para trás, além da rotação no eixo y. Quando o botão da direita é pressionado há a movimentação para cima, para baixo, para esquerda e para a direita. Maverik também possue uma função de navegação padrão para o teclado, a qual se comporta da seguinte forma:

- cursores: anda para frente, trás e rotaciona em y,
- teclas *page up/dow*: anda para cima e para baixo,
- teclas *alt-Page up/down*: rotaciona no eixo x,
- alt-seta esquerda/direita: anda para esquerda e para a direita, respectivamente.

4.1.1 A aplicação desenvolvida

Para a criação do parque foram utilizados os objetos padrão de Maverik. Foi utilizada a idéia de objetos compostos para se criar a árvore e um loop para criar várias árvores iguais. Para criar um objeto composto, define-se os objetos a serem usados (com suas características, como: tamanho, cor, luminosidade, localização, etc.), insere-os em um SMS através da função **mav_SMSObjectAdd** ("nome do

sms", **mav_objectNew** (classe do objeto, nome do objeto)) e define-se a matriz de transformação comum a todos. Através da função **mav_SMSObjectAdd** ("sms a ser renderizado", **mav_objectNew**(**mav_class_SMSObj**, "sms no qual estão os objetos")) adiciona-se o SMSObj (que é um objeto padrão do Maverik) em um SMS para ser renderizado (nota-se que a **função mav_objectNew** declara o SMS no qual estão os objetos como sendo da classe SMSObj).

A cada interação do *loop* de criação da árvore é definida uma árvore em um lugar distinto. Esta árvore é incluida em um SMS para ser renderizada. O código da criação das árvores pode ser visto em 4.2. Além das árvores, a criação dos bancos seguiu esta linha de raciocínio.

Para que a prancha da gangorra se movesse conforme uma gangorra real, foi implementado na função principal um código (figura 4.3) que a cada vez que a janela é redesenhada, renderiza o assento da gangorra com uma inclinação diferente no eixo X. Desta forma é representado o movimento de "sobe e desce"da gangorra.

A colisão com o solo foi tratada através de uma função que nada mais é que um condicional, ou seja, se o parâmetro do atributo **eye** do parâmetro de visão que indica a altura em que a câmera se encontra for menor ou igual a um dado valor, este parâmetro recebe um valor padrão, fazendo com que a câmera fique parada no chão do ambiente.

A detecção de colisão com outros objetos é feita através da intersecção de todos os SMSs da janela com a linha que junta o ponto de vista antes e após da navegação ter ocorrido. Se qualquer objeto intercepta esta linha e a distância até ele é menor que a distância viajada, então ocorre a colisão e os parâmetros de visão não são modificados. É facil notar que este tipo de detecção só ocorre quando o ponto de vista, ao ser movimentado, atinge um objeto. De acordo com o guia de programação [GNU Maverik (2002)], a detecção de colisão é implementada apenas em uma função de navegação. Logo, detecção de colisões utilizando espaço de volumes representando corpos de usuários deve ser realizada através de funções feitas pelo programador.

Antes de qualquer ação da aplicação, o Maverik deve ser inicializado. Isto é feito através de duas funções: **mav_initialise** ou **mav_initialiseNoArgs**. As duas funções são praticamente iguais, diferenciando-se, porém, na maneira que se toma os argumentos que serão usados para controlar o processo de inicialização. Após isto, os objetos são definidos, incluídos em um SMS e renderizados através da função **mav_SMSDisplay("janela na qual o objeto será renderizado", "nome do objeto")**. Por padrão uma janela que ocupa um quarto da tela é aberta e posicionada no quadrante inferior esquerdo dessa tela. Os resultados dessa implementação

```
void defCaule(MAV_cylinder* c) //Definir o caule
{
 c->radius= 0.2;
 c->height= 6;
 c->endcap= 10;
 c->nverts= 10;
 c->matrix= mav_matrixSet(0,270,0, -20,1,0);
 c->sp= mav_surfaceParamsNew(MAV_COLOUR, 1,1,1);
}
void defCopa(MAV_cone* copa) //Definir a copa
{
 MAV_surfaceParams sp;
 MAV_palette* paleta;
 copa->rt=0;
 copa->rb=2;
 copa->height= 3;
 copa->endcap= 100;
 copa->nverts= 100;
 copa->matrix= mav_matrixSet(0,270,0, -20,4,0);
 if (mav_paletteTextureSet(mav_palette_default,
    TEX_TREE1, "copa.ppm"))
  copa->sp= mav_surfaceParamsNew(MAV_TEXTURE,2,2,TEX_TREE1);
 mav_paletteTextureFree(mav_palette_default,TEX_TREE1);
}
void defArvore(MAV_SMS *arvore, MAV_SMSObj *conjArvores)
{/* Definicao da arvore */
 conjArvores->sms=arvore;
 conjArvores->matrix=mav_matrixSet(0,0,0, -20+mav_random()*70,
                                   0,-20+mav_random()*100);
}
/*NA FUNCAO MAIN*/
defCaule(&caule);
defCopa(&folhas);
arvore= mav_SMSObjListNew();//Criar os SMS's
conjArvores= mav_SMSObjListNew();
mav_SMSObjectAdd(arvore,mav_objectNew(mav_class_cone, &folhas));
mav_SMSObjectAdd(arvore,mav_objectNew(
                        mav_class_cylinder,&caule));
for(quantidadeArvores=0;quantidadeArvores<80;quantidadeArvores++)
{
 defArvore(arvore, &arvores[quantidadeArvores]);
 mav_SMSObjectAdd(conjArvores,mav_objectNew(mav_class_SMSObj,
                &arvores[quantidadeArvores]));
}
```

Figura 4.2: Código de criação das árvores em Maverik

```
/*-------------- Faz a gangorra balancar -------------------*/
/*verifica o valor de rx, ry, rz, ou seja,
  roll, pitch e yaw respectivamante*/
mav_matrixRPYGet(assentoGangorra.matrix, &rx, &ry, &rz);

if (verificaVerdade == MAV_FALSE)
{
  while(rx <= 25 && rx >= -25)
  {
    assentoGangorra.matrix=mav_matrixSet(rx,0,0, -51,-1.5,7);
    drawFrame(groundPlane, parque, conjArvores,
              conjBancos, gangorra);
    rx-=5;
  }
  verificaVerdade = MAV_TRUE;
  }
else
{
  while (rx >= -25 && rx<25)
  {
    assentoGangorra.matrix=mav_matrixSet(rx,0,0, -51,-1.5,7);
    drawFrame(groundPlane, parque, conjArvores,
              conjBancos, gangorra);
    rx+=5;
}
verificaVerdade = MAV_FALSE;
}
```

Figura 4.3: Código que faz a gangorra balançar - Maverik

podem ser vistos nas figuras 4.4 e 4.5.

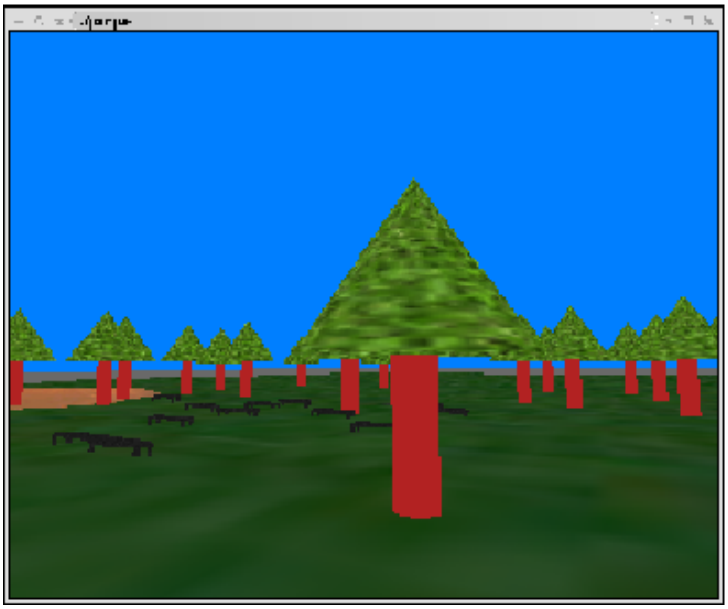

Figura 4.4: ambiente criado no maverik

4.1.2 Conclusões sobre a biblioteca Maverik

Utilizando o Maverik pôde-se observar que, baseando-se nos aspectos citados no referencial teórico (facilidade de uso, facilidade de obtenção, documentação disponível, eficiência do programa produzido, adequação aos princípios do estilo de POO, tamanho do programa produzido, nível de abstração e abrangência das funções e métodos implementados, recursos disponíveis), percebeu-se que esta é uma biblioteca de fácil utilização, pois a abstração de seus metodos é de alto nível e sua obtenção é fácil, sendo feita através do endereco `http://aig.cs.man.ac.uk/maverik/download.php` [visitado em 6/3/2002], baixando-se o arquivo disponível nesta.

A instalação também é fácil, bastando descompactar o arquivo de distribuição, ir para o diretório Maverik já criado e digitar **./setup; make**. Porém Maverik não completa a instalação. Para tal é preciso mover os arquivos gerados (diretório **lib**) para o diretório **/usr/lib**. Serão compilados a biblioteca maverik, os programas

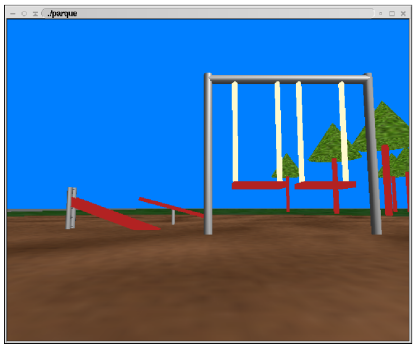

Figura 4.5: ambiente criado no maverik - parquinho

exemplos e qualquer programa de demonstração presente. O script **setup** gera um makefile apropriado para a máquina na qual a biblioteca está sendo instalada.

Com relação a documentação disponível, existem dois manuais que vem com a distribuição: um de especificação das funções e outro, que pode ser considerado um tutorial, chamado guia do programador [GNU Maverik (2002)]. A divisão desses tutoriais em níveis auxilia o desenvolvedor a entender melhor a biblioteca e assim assimilar o aprendizado de forma satisfatória, confirmando que Maverik é simples e fácil de usar. Além da documentação há um *e-mail* para que dúvidas possam ser sanadas (`maverik@aig.cs.man.ac.uk`).

Um ponto negativo do guia do programador é a existência de capítulos compostos apenas pelo título e a seguinte mensagem [GNU Maverik (2002)]:

"Please email maverik@aig.cs.man.ac.uk for advice on this topic."

A explicação dada por eles para este fato é a de que Maverik é um sistema grande e uma documentação completa tomaria muito tempo dos desenvolvedores da mesma. Por isso os esforços da documentação se concentraram no nível considerado para iniciantes. É em casos como este que a existência de um *e-mail* para que o programador possa tirar suas dúvidas é considerado de suma importância.

O programa produzido pode ser considerado eficiente, pois é veloz, conseguindo reagir rapidamente às ações do usuário, um dos requisitos a ser satisfeito por um sistema de RV. Para a verificação deste fato foi feito um teste no ambiente através da inclusão de cilindros (caules das árvores) a mais. O intervalo entre a apresentação das imagens começou a ser notado depois da inclusão de mil cilindros. Apesar do tratamento de colisão diminuir a velocidade da navegação, esta

continua a ser boa, o que pode ser explicado através do tipo de detecção de colisão realizado, pois é mais rápido ver se um objeto intercepta uma linha que ver se dois sólidos se interceptam.

Não há adequação aos princípios do paradigma de programação orientada a objetos, pois como a biblioteca foi feita para C, perde-se a idéia de orientação a objetos, o que acarreta alguns inconvenientes como a existência de um *struct* chamado MAV_object, que é simplesmente o encapsulamento, em uma estrutura de dados simples, de um ponteiro que indica os dados de um objeto e os métodos que agem sobre este objeto, e a existência de outro objeto chamado MAV_SMS, que tem definição análoga ao MAV_object, diferenciando-se porém na existência de um atributo que informa se o objeto é ou não selecionável através do *mouse* ou teclado. Percebe-se, então que o primeiro é um tipo especial do segundo, porém os dois são implementados devido ao fato de não haver as idéias de orientação a objeto (herança, uso.).

Maverik, como dito no referencial teórico, roda em várias plataformas e possui vários recursos disponíveis, tais como suporte para periféricos 3D (HMD, dispositivos de apresentação de imagens estéreo, dispositivos de posicionamento de cabeça e mãos, dispositivos de reconhecimento de linguagens, etc.[1]), tais como *Polhemus FASTRAK*[2] e *ISOTRAK II*[3], *Flock of birds*[4], *Spacetec SpaceBalls*[5], etc. Outro fator importante é o fato do programador poder usar tanto as características padrão desta bibloteca para toda a renderização e navegação de seu sistema como também definir as características da forma que ele quiser utilizando apenas funções implementadas na biblioteca.

O tamanho do programa gerado foi 25k Bytes e o espaço que a biblioteca ocupa em disco 2M Bytes. Este programa, mesmo sendo simples, gerou um ambiente com qualidade satisfatória, com um certo grau de realismo. Em ambientes mais complexos o grau de realismo pode ser tão grande que objetos sintéticos podem ser confundidos com objetos reais (por exemplo, vide figura 4.6).

Maverik também possue funções para a visualização de ambientes gerados com radiosidade (figura 4.7) (método de renderização baseado em uma análise detalhada das reflexões da luz em superfícies difusas. As imagens que resultam dessa renderização são caracterizadas por sombras suaves e graduais. É geralmente usada para renderizar imagens de interior, e pode alcançar resultados extremamente

[1] `http://student.dei.uc.pt/~assuncao/principal.html`
[2] `www.polhemus.com`
[3] `www.polhemus.com/isotrak.htm`
[4] `www.ascension-tech.com`
[5] `www.engmeth.com/products.htm`

Figura 4.6: nível de realismo de um ambiente criado no maverik

foto-realísticos para cenas que possuem muitas superfícies com reflexão difusa.[6]), aumentando assim o nível de realismo de um sistema e consequentemente sua qualidade. Além disto, a biblioteca também possue funções para a apresentação de imagens estéreo, o que pode ser considerada uma vantagem, haja visto que o programador terá apenas que usar esta funções, ao invés de implementá-las.

Figura 4.7: ambiente com radiosidade

Uma desvantagem do Maverik é que ele não tem suporte para áudio e vídeo. O desenvolvedor que quiser usar estes recursos tem que adicionar os mecanismos para isto. Além deste fato, há também o de Maverik ser desenvolvido para criar um ambiente mono-usuário. Desta forma não há como fazer ambientes que exijam a existência de mais de um usuário, pois não está incluída qualquer assistência para

[6]www.3dzone.com.br/radiosidade.html

executar ambientes multi-usuário.

A aceitação de Maverik por parte da comunidade desenvolvedora de ambientes de realidade virtual é grande, havendo até projetos de grande porte, como o REVEAL (reconstituição de cenas de crimes), TCI (centro de informação aos turistas) e *Molecular modeller* (modelador de moléculas). A galeria de projetos utilizadores do Maverik é extensa e pode ser vista em `http://aig.cs.man.ac.uk/gallery/`

4.2 PLIB

A biblioteca PLIB utiliza a idéia de árvore para o desenvolvimento do ambiente. Como nó raíz tem-se a cena (será o pai de todos os objetos a serem renderizados) e como nós filhos tem-se os objetos a serem renderizados. Na realidade, a árvore é montada da seguinte forma: cria-se o nó raiz (cena), cria-se o objeto a ser renderizado e aplica-se a ele o estado desejado (textura, cor, tamanho e luz), cria-se a matriz de transformação do mesmo, adiciona-se esta matriz ao nó raiz e, por fim, adiciona-se o objeto ao nó da matriz de transformação (vide figura 4.8).

Como dito anteriormente no referencial teórico, PLIB é um conjunto de bibliotecas semi autônomas (semi independentes). As principais são:

- Standart Geometry Library (SG): utilizada para criação e manipulação de vetores e matrizes, utilizados nas transformações dos objetos.
- Simple Scene Graph Library (SSG): Cuida da renderização da árvore de objetos. Possue todas as funções responsáveis pela criação dos nós e inclusão dos mesmos na árvore.
- SSG Auxiliary Library (SSGA): utilizada para a criação dos objetos. Possui função de criação de cubos, cilindros, etc.

Existem classes específicas na biblioteca **SSGA** para tratamento de cubos, cilindros, esferas, ondas de água, *spray* de partículas, fogo, entre outros. Todos esses objetos são desenhados a partir de uma quantidade de triângulos. Isto ocorre porque o polígono mais simples que existe é o triângulo (todo triângulo é necessariamente planar e convexo) e desenhar objetos a partir deles simplica algoritmos usados na renderização, tornando este processo mais eficiente. O número de triângulos de um objeto é especificado na sua construção. PLIB aceita importação de objetos de formato de arquivo VRML97, Lightwave e AC3D, além de outros mais. Assim como dito anteriormente, isto pode ser considerada uma vantagem,

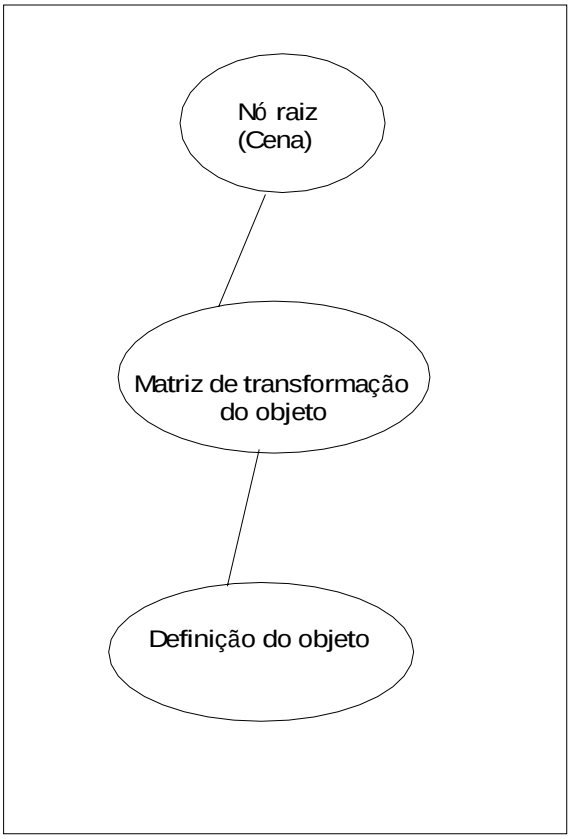

Figura 4.8: esquema de formação da árvore de renderização do PLIB

pois o programador pode reutilizar objetos feitos em outros formatos, poupando-lhe tempo de implementação. Com relação a objetos compostos por um conjunto de outros objetos, como PLIB segue a idéia dos objetos estarem dispostos em uma árvore, os nós (partes do objeto) que compõem o objeto principal devem ser filhos de um nó que descreverá o objeto em si. É este nó que deverá ser inserido no nó raiz.

A renderização do ambiente é feita da seguinte forma: primeiro todas as funções do GLUT (*GL Utility Toolkit*) (que é utilizado para criar a janela na qual o ambiente será renderizado) são inicializadas. Se estiver sendo utilizada a biblioteca SSG, esta também deve ser inicializada com uma função própria para isto, já

implementada. Após isto, as características dos objetos a serem renderizados são definidas (forma, cor, tamanho, localização no ambiente e textura.). Por fim tem-se a utilização de função específica do GLUT, responsável pelo *loop* de renderização, cuja execução causará no programa um loop de processamento de eventos. Se não há eventos para processar, o programa entra em um estágio de espera, com os gráficos na tela, até que o programa seja terminado (por exemplo, por um aperto de alguma tecla).

A biblioteca PLIB não tem um sistema pré-definido de movimentação da câmera através do *mouse*. Desejava-se implementar um que fosse como o do Maverik, com rotações relativas à câmera, dando assim liberdade de movimentos ao usuário. Existem duas funções para alterar a câmera: uma utiliza rotações que são umas relativas às outras, e a outra utiliza uma matriz de transformação geométrica. Como não há tutorial do PLIB, os exemplos que vem com a distribuição tratam a navegação de forma restrita (não com a liberdade que se desejava) e usuários mais experientes indicaram que nunca fora implementada uma navegação como a desejada, além de não saberem indicar um caminho para se fazer isto, a solução foi criar uma classe que tem como atributos a posição da câmera, as direções x, y e z da câmera e a matriz de transformação da mesma. Assim, a posição da câmera é guardada e pode ser utilizada nas tranformações vindouras. A classe desenvolvida tem métodos que implementam rotações e translações, relativas à posição da câmera, que calculam qual matriz de transformação será usada na função de PLIB que recebe uma matriz. Assim fica fácil fazer uma navegação como a existente em Maverik. Desta forma a cada modificação da câmera, sua matriz de transformação é modificada, e adicionada à câmera através da função que modifica a posição da câmera a partir da matriz de transformação geométrica, passada como parâmetro da função.

Para os eventos do mouse e teclado são utilizadas funções da biblioteca GLUT. A navegação é feita através das funções implementadas na classe que representa a câmera. Elas permitem que naveguemos livremente pelo ambiente. No caso da utilização do mouse para navegação, são utilizadas as funções do GLUT, implementadas para este fim, assim como no caso da navegação através do teclado. Em suma, a criação da janela de renderização (tamanho, localização) e as funções de navegação são todas provenientes do GLUT, porém existe uma biblioteca, chamada *PUI: A Picoscopic User Interface*, a qual faz parte da PLIB, que é um conjunto de funções de *widgets* que necessitam de GLUT, OpenGL e C++ para operarem. Fica, então, a critério do desenvolverdor qual usar para a criação da janela de renderização.

Testes de colisão em PLIB podem ser feitos através de funções já implementadas nesta biblioteca. Existem três funções que fazem este teste. Elas implementam três formas diferentes de testar colisão em uma base de dados:

- a função intercepta uma esfera contra o ambiente;
- a função intercepta uma linha vertical começando em um ponto dado pelo programador;
- a função intercepta um vetor arbitrário cuja direção é definida por um ponto dado pelo programador.

Em todas as funções acima a busca pela colisão começa no nó raiz e a base de dados é transformada pela matriz passada como parâmetro da função, antes do teste ser realizado. Como esta matriz é o inverso da matriz que descreve a localização do objeto de teste, o que ocorre na verdade é que a matriz inversa que descreve a localização do objeto passado como parâmetro na função (esfera, ponto, ou vetor) é aplicada ao ambiente, mudando-o de lugar e trazendo-o até o objeto de teste (esfera, ponto, ou vetor), ou seja, ao invés do objeto (esfera, ponto ou vetor) ir até o ambiente, é o ambiente que vai até ele.

4.2.1 A aplicação desenvolvida

A idéia de objetos compostos foi usada para a renderização dos bancos. Foi criado um vetor de 25 posições do tipo **ssgTransform**. Estas posições indicam onde ficam os bancos no ambiente. Um *loop* percorre este vetor adicionando um nó *ssgBranch*, que define o banco (todas os objetos que descrevem o banco estão guardados neste tipo de nó), em cada posição do mesmo. Cada elemento deste vetor é adicionado ao nó raiz. Como já se tem conhecimento de que a biblioteca VRJUGGLER não aceita importação de objetos dos tipos adicionados na aplicação realizada utilizando-se Maverik, não houve importação de objetos em PLIB, com o intuito de que os ambientes ficassem com o mesmo número de objetos para uma comparação melhor. Para a prancha da gangorra se mexer foi implementado na função **update_motion()** o trecho de código presente na figura 4.9. A cada vez que a janela é redesenhada, o assento da gangorra aparece em uma nova posição devido a rotação aplicada a sua matriz de tranformação.

A posição default da câmera é no ponto (0,0,0), com a direção de x (1,0,0), direção de y (0,1,0) e de z (0,0,1). A mesma sofre uma rotação de 90 graus em x e logo após uma rotação de 180 graus em z para que o parque possa ser enxergado

```
sgCoord tptpos; //indica as coordenadas do assento da gangorra
static float frameno = 0;

frameno+=0.1;
if (decrementando)
{
  sgSetCoord ( & tptpos, -48.5, 0.15, 9.25, rx, 0.0f, 0.0f );
  assentoGang  -> setTransform ( & tptpos );
  rx -= 0.1;
  if (rx < -25.0)
  {
    decrementando = false;
  }

}
else
{
  sgSetCoord ( & tptpos, -48.5, 0.15, 9.25, rx, 0.0f, 0.0f );
  assentoGang  -> setTransform ( & tptpos );
  rx += 0.1;
  if (rx > 25.0) decrementando = true;
}
```

Figura 4.9: Código que faz a gangorra balançar - PLIB

e também para que a câmera inicial fique na mesma posição que a câmera *default* de Maverik. As coordenadas do mundo em PLIB podem ser vistas na figura 4.10.

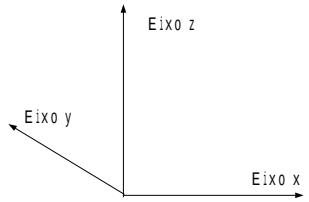

Figura 4.10: Sistema de coordenadas do mundo em PLIB vista pela posição padrão inicial da câmera

Para a navegação no ambiente, a classe câmera tem implementadas as funções responsáveis por fazer a câmera ir para frente, para trás, cima, baixo, esquerda e direita, além das três rotações necessárias (virar para esquerda/direita, virar para cima/baixo, "tombar"para esquerda/direita). Para mover a câmera, então, é só usar as funções implementadas na classe **clCamera**. No caso da utilização do mouse para navegação, são utilizadas as funções **static void mousefn (int button, int upDown, int x, int y)** que é acionada cada vez que um botão do mouse é pressionado ou solto e **static void motionfn (int x, int y)** que é acionada quando o botão do mouse é apertado e o mouse se move. Esta funções são inicializadas com as seguintes funções do GLUT: **glutMouseFunc(mousefn)** e **glutMotionFunc (motionfn)**. Para que a câmera continue a se mover, mesmo com o *mouse* estacionado e algum dos seus botões acionado, foi implementado na função **update_motion()** o trecho de código da figura 4.11. Este código faz com que a posição da câmera seja modificada se algum botão do mouse estiver pressionado e tiver ocorrido algum deslocamento do mesmo anteriormente (com o botão acionado). No caso da navegação através do teclado, foi implementada a **função static void specialfn (int k, int x, int y)**, a qual verifica se há algum botão pressionado (no caso, as teclas que indicam esquerda, direita, cima e baixo, representadas por flechas no teclado, *Page up* e *Page down*) e, se hover, atribui seu valor ao parâmetro **k**. Os parâmetros *x* e *y* desempenham o mesmo papel que nas funções de navegação do mouse. A função **specialfn** verifica então qual o botão apertado (através do parâmetro **k**) e, a partir disto, posiciona a câmera, utilizando as funções implementadas na classe clCamera, de acordo com **x** ou **y**.

Para ir direto a determinados locais, como acontece na aplicação desenvolvida com Maverik, utiliza-se a mesma idéia usada no Maverik: foi implmentada uma

```
if(apertado && esquerdo)
{
     camera.yaw(andarXQuant);
     camera.forward(andarYQuant);
}

if (apertado && direito)
{
     camera.pitch(andarYQuant);
     camera.roll(andarXQuant);
}
posicionarCamera();
```

Figura 4.11: Trecho de código da navegação

função chamada **static void keyboard (unsigned char key, int, int)** a qual percebe se uma determinada tecla foi acionada e procura se há algum comando a ser executado se a tecla percebida fosse acionada.

A colisão entre objetos foi testada através da função **int ssgIsect (ssgRoot *root, sgSphere *s, sgMat4 m, ssgHit **results)** que intercepta uma esfera contra o ambiente. Para isso foi criada uma esfera que acompanhará a câmera. Cada vez que o ambiente for redesenhado a posição da câmera é adquirida e faz-se o teste da colisão. Se não houver colisão, a navegação ocorre normalmente, porém se houver colisão a câmera é posicionada usando-se a matriz da câmera anterior à colisão. Para melhor entendimento, vide figura 4.12. A colisão entre o solo também é detectada com a função acima, pois a esfera colidirá em qualquer direção e não como no Maverik, que utiliza uma linha para testes de colisão (apenas uma direção). Não foi percebida dimiuição significativa na velocidade da navegação com o teste de colisões, o que é considerado uma vantagem neste caso, já que o teste está sendo feito entre sólidos. Os resultados da implementação podem ser vistos nas figuras 4.13 e 4.14.

4.2.2 Considerações sobre a biblioteca PLIB

A bibllioteca PLIB é de fácil obtenção, podendo ser adquirida no endereço `http://plib.sourceforge.net/download.html` [visitado em 6/2/2003]. Para a sua instalação, é exigido que já se tenha instalado o OpenGl e o Mesa, os quais também são facilmente adquiridos na *internet*. Após seu *download*, basta dar os segintes comandos (como *root*):

```
static void redraw ()
{
  sgMat4 mat; sgMat4 matInv;
  ssgHit *results; sgMat4 matTrans;
  static clCamera camAnterior;

  camera.GetMatrix(mat); //pega a matriz da camera
  sgMakeIdentMat4(matTrans);

  //Preenche a matriz de translacao
  matTrans[3][0] = mat [3][0];
  matTrans[3][1] = mat [3][1];
  matTrans[3][2] = mat [3][2];

  //inverte-se a matriz de translacao da camera
  //Usa-se a matriz de translacao pois esta e a unica
  //caracteristica que interessa na colisao. Ou seja,
  //nao intersa para onde a camera esta olhando, e sim
  //onde ela esta.
  sgInvertMat4 ( matInv, matTrans );
  //Ocorreu colisao!
  if (ssgIsect (parque, &esfera, matInv, &results))
  {
    //atribui a camera a posicao da camera anterior
    camera = camAnterior;
    posicionarCamera();
  }
  else
  {
    //Guarda a posicao da camera
    camAnterior = camera; ;
  }
}
```

Figura 4.12: Teste de colisão

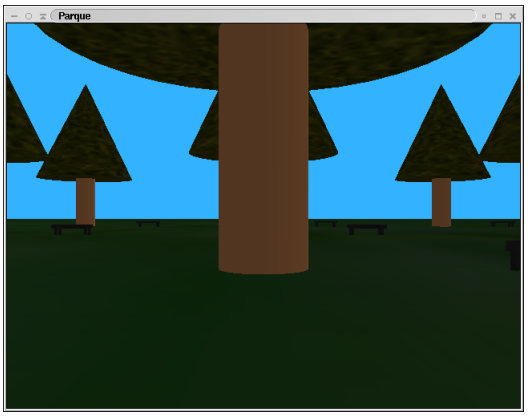

Figura 4.13: ambiente criado no PLIB

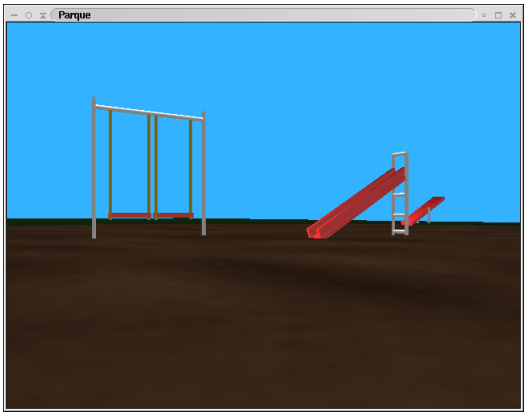

Figura 4.14: ambiente criado no PLIB - parquinho

tar xzf plib-1.6.0.tar.gz

cd plib-1.6.0

./configure ; make ; make install

Existe um pacote separado que traz exemplos de programas feitos usando-se esta biblioteca. Ele também deve ser baixado para o aprendizado da mesma, pois não existem tutoriais de PLIB (como dito anteriormente). Isto é um ponto bastante negativo para o aprendizado desta, haja visto que os exemplos também

não trazem comentários das funções e o aprendiz tem que "advinhar"o que muitas funções significam. Há uma lista de discussão dos usuários, onde se pode tirar dúvidas (plib-users@lists.sourceforge.net), porém, muitas vezes a resposta demora a chegar ou não chega.

Outro ponto do PLIB é que o úsuario tem que estar familiarizado com a biblioteca OpenGL, já que muitas funções usadas são dela. É difícil aprender PLIB, haja visto que não há tutorial, os exemplos são mal comentados, quando o são, e a lista de discussão às vezes não retorna resposta para o problema. Em contra-partida PLIB já tem funções complexas implementadas, como, por exemplo:

- Teste de colisão de volumes: muito importante para detectar interceptação entre objetos.
- *Spray* de partículas: útil na criação de objetos, como a água saindo do bico da chaleira na figura 4.15.
- Fogo: simula fogo, como pode ser visto na figura 4.15[7].

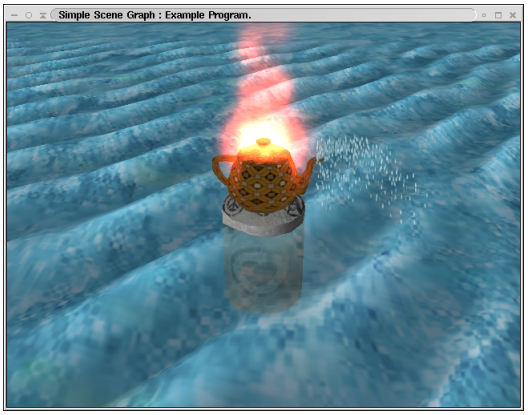

Figura 4.15: Utilização do *spray* de partículas e do fogo

- Arranjo de mola e amortecedor: utiliza uma composição de molas e amortecedores para criar elementos deformáveis, os quais irão compor um objeto.

O tamanho do executável gerado é de 2,3M, muito maior que o executável da implementação em Maverik, porém, dentro de programas de realidade virtual, este

[7] disponível com a distribuição da biblioteca

tamanho ainda é aceitável. A biblioteca ocupa, em disco, 16M. Apesar de algumas funções implementadas não produzirem o efeito desejado em algumas ocasiões, como, por exemplo a mudança da câmera de lugar no programa implementado, utilizando-se um sgCoord, no geral PLIB tem muitas funções já implementadas, como as citadas acima, as quais facilitam o trabalho do programador. O programa gerado é bastante eficiente, já que responde rapidamente às ações do usuário, mesmo possuindo teste de colisão entre volumes. Para o teste de velocidade foi utilizada a mesma idéia do teste de velocidade do Maverik. Somente após a inclusão de dois mil e quinhentos cilindros (caules de árvores) é que começa a se notar o intervalo entre os *frames*. O nível de qualidade das aplicações em PLIB é boa. Porém, tudo depende do realismo que o desenvolvedor consegue dar a sua aplicação. Este realismo pode ser dado, dentre outros fatores, através da iluminação. É fácil perceber a diferença de realismo entre as figuras 4.16 e 4.17. A única coisa modificada no programa que gera esse ambiente foi a iluminação.

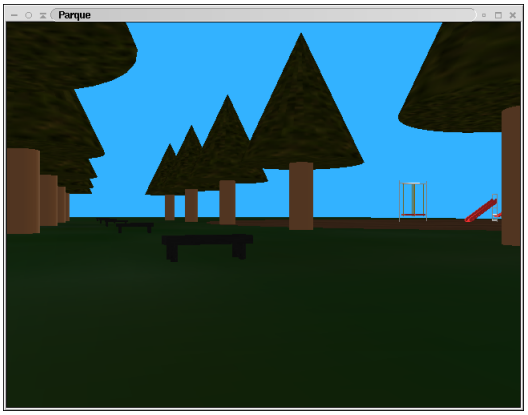

Figura 4.16: Ambiente gerado com uma certa iluminação no PLIB

O nível de adequação à orientação a objetos é impressionante. Como o ambiente é sempre visto como uma árvore composta por objetos, algumas vezes compostos por outros objetos (como o caso dos bancos), cada qual com a sua matriz de transformação e características como cor, textura e iluminação, é fácil verificar as idéias contidas na orientação a objetos (herança, polimorfismo, entre outras idéias.). Logo não há o problema, já detectado no Maverik, de existirem objetos que são tipos específicos de outros, mas, ao invés de herdarem as características do objeto mais geral, são novamente implementados de forma totalmente independente

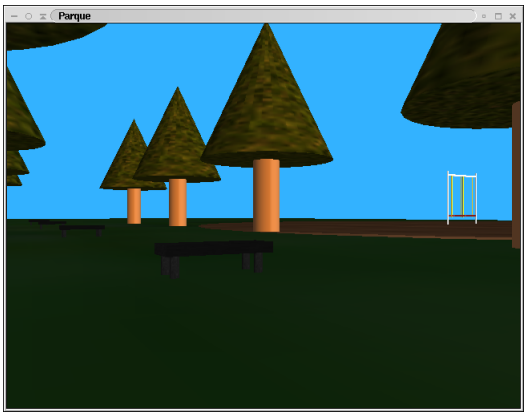

Figura 4.17: Mesmo ambiente gerado com outro tipo de iluminação

do outro. PLIB ainda possue funções para utilização de *joystick*, música e também para a criação de jogos para a rede.

A aceitação de PLIB por parte da comunidade desenvolvedora de jogos é grande. Fora desta área não foi encontrado nenhum projeto envolvendo PLIB. Podemos citar, dentre os vários projetos, *FDS - Flight Dynamic Simulator*, um simulador de vôo, *TuxKart*, um jogo de corrida de *kart* e *flightgear*, simulador de vôo. Estes e outros projetos podem ser encontrados em `http://plib.sourceforge.net/` [visitado em 27 de Maio de 2003].

4.3 VRJuggler

A biblioteca VRJuggler é um *framework* e um conjunto de classes em C++ para desenvolvimento de ambientes virtuais. Ela foi desenvolvida para permitir ao desenvolvedor acesso direto à interface do programa para o controle máximo da aplicação, enquanto ainda fornece uma visão da renderização fácil de se entender, além da facilidade em conectar novos dispositivos de entrada e saída. VRJuggler é desenvolvida para prover um nível de abstração em cima da computação distribuída, compartilhamento de memória, multiprocessamento e dispositivos I/O, ou seja, o programador não precisa se preocupar com estes detalhes, pois VRJuggler já o faz. Nesta biblioteca todas as aplicações são escritas como objetos a serem identificados pelo *kernel*. Assim, como toda a aplicação em VRJuggler é formada por objetos, desenvolvedores não precisam escrever a função *main()* da forma

```
#include <simpleApp.h>
#include <Kernel/vjKernel.h>

int main()
{
     char key;
   // alocar o objeto kernel e o objeto aplicacao
   vjKernel* kernel = vjKernel::instance();
   simpleApp* application = new simpleApp();
   kernel->loadConfigFile();
   kernel->start();
   kernel->setApplication(application);

   while(1)
   {
      usleep(250000);
   }
}
```

Figura 4.18: implementação básica da função main()

tradicional. Ao invés disto, é necessário criar uma aplicação que implementa um conjunto de interfaces pré-definidas. O *kernel* controla o tempo de processamento da chamada dos métodos de implementação da interface dos objetos. Devido ao fato do *kernel* ter informações sobre os recursos necessários da aplicação, é mantida uma agenda (*schedule*) que define quando a aplicação entra no tempo de processamento. Esta é a base para manter coerência no sistema. A implementação da função *main* no VRJuggler pode ser vista na figura 4.18.

Como toda a configuração é feita através de arquivos independentes do arquivo da aplicação, a aplicação pode rodar em qualquer máquina sem mudanças em seu programa, e sim somente no arquivo de configuração. Devido a este fato também, dispositivos novos podem ser adicionados à aplicação com certa facilidade.

As funcionalidades do VRJuggler estão divididas em componentes chamados *manager*. Cada *manager* encapsula um conjunto específico de detalhes do sistema. Os *managers* existentes são:

- *Input Manager*: Todos os dispositivos de entrada do sistema são controlados por este *manager*. Quando uma aplicação requer acesso a um dispositivo em particular, ela recebe um *Proxy*, que é um intermediário entre este dispositivo e a aplicação. A aplicação "conversa"com o proxy e este mantém os

dados para o processamento do dispositivo.

- *Output Manager*: Este *manager* trabalha da mesma forma que o *Input manager*, exceto pelo fato dos dados serem enviados para um dispositivo de saída.

- *Draw managers*: Há muitos *Draw managers*. Cada um encapsula um API específico. O *Draw manager* engloba vários detalhes das operações gráficas, tais como parâmetros de visão, funções de renderização definidas na aplicação e engloba detalhes como visão estéreo, *buffer swapping*, etc.

- *Display Manager*: Encapsula todas as informações sobre as janelas, a renderização e **API**. Assim como o *Draw manager*, existem vários tipos de *Display Manager*, mas apenas um é instanciado por vez. Ele encapsula informações como tamanho, localização, pipeline e parâmetros de visão. Ele usa estas informações para criar janelas. A partir daí o controle é passado para o *draw manager*, o qual cria a renderização.

- *Network Manager*: É uma interface genérica, orientada a objeto, para a abstração da rede do sistema. Ele provê o sistema com toda a rede necessária para o uso da aplicação e do *Kernel*.

- *Configuration Manager*: Gerencia toda a parte de configuração, tal como nome dos dispositivos usados, modo do vídeo, localização das janelas, entre outros. As aplicações podem usar este gerenciador para definir e armazenar suas próprias informações de configuração.

- *Environment Manager*: É o controle do sistema em tempo de execução. Consiste em dois componentes: um objeto similar aos outros gerenciadores e uma interface gráfica que permite que usuários examinem e controlem a aplicação.

Um conjunto de objetos de baixo nível formam a fundação na qual o *kernel* e os gerenciadores (*managers*) são construídos. Estes objetos controlam o gerenciamento de processos, sincronização e gerenciamento de memória. Aplicar o VRJuggler a uma nova arquitetura consiste, então, em reescrever estas classes de baixo nível para se usar a configuração da nova arquitetura. O *Kernel* liga todos os componentes do VRJuggler. Ele controla todo o sistema em tempo de execução e quebra toda a comunicação entre os vários gerenciadores. A aplicação "conversa"diretamente com o *kernel*, tendo acesso ao *Draw manager*, dispositivos

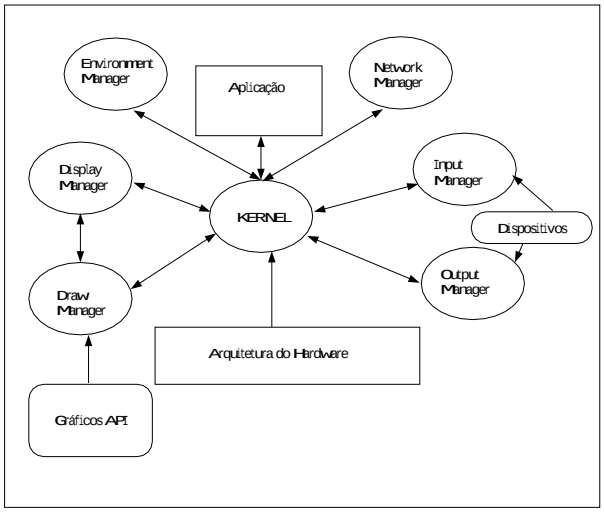

Figura 4.19: Descrição do VRJuggler

controlados pelo *Input manager* e *Output manager* e informações de configuração armazenadas no *Configuration manager* somente através do *Kernel*. A descrição de todos os componentes do VRJuggler pode ser vista na figura 4.19.

4.3.1 A aplicação desenvolvida

A classe da aplicação deve sempre herdar da classe vjGlApp, já implementada na biblioteca. A implementação base da classe aplicação deve definir as seguintes funções:

- init() - é chamada pelo *kernel* para inicializar qualquer dado da aplicação. Quando o *kernel* se prepara para iniciar uma nova aplicação, ele primeiro chama esta função para sinalizar à aplicação que ela está proxima de ser executada.

- preFrame() - é chamado quando o sistema está prestes a disparar o desenho.

- intraFrame() - o código implementado neste método é executado em paralelo com o método de renderização, ou seja, ele executa enquanto o frame está sendo desenhado.

```
// --- Bancos do parque: utilizacao de display list --- //
     glMatrixMode(GL_PROJECTION);
     glPushMatrix();
     glMultMatrixf(banco.getFloatPtr());
     // Cria a display list para os bancos
     glNewList(1, GL_COMPILE);
     for ( float x = 0.0; x < 22.0; x += 7.0 )
     {
        for ( float  y = 0.0; y < 22.0; y += 6.0)
        {
             glPushMatrix();
             glTranslatef(x,0.0,y+5);
             glEnable(GL_TEXTURE_2D);
             glBindTexture ( GL_TEXTURE_2D, idsTexturas[1]);
             drawBanco(banco);
             glPopMatrix();
             glDisable(GL_TEXTURE_2D);
        }
     }
     // Fim da lista
     glEndList();

     glCallList(1);
```

Figura 4.20: Trecho de código dos bancos

- postFrame() - é disponibilizado para processamento no fim do *loop* do *kernel*.

A construção de objetos compostos no VRJuggler é feita através de *display lists* do OpenGL. Cria-se uma *display list* através do comando **glNewList("inteiro identificador da lista", GL_ COMPILE)** e adiciona a ela os objetos constituintes do objeto final. Após isto termina-se a lista através do comando **glEndList()** e chama-se a *display list* através do comando **glCallList("identificador da lista")**. Esta idéia foi usada para a renderização dos bancos (4.20).

Para a prancha da gangorra se mexer foi criada uma *display list* para o assento da gangorra e implementada na função **postFrame()** uma forma de gerenciar a posição do assento da gangorra. Desta forma, a cada vez que a janela é redesenhada, o assento da gangorra aparece em uma nova posição devido a rotação aplicada a sua matriz de tranformação. Esta implementação pode ser vista na figura 4.21.

```
// Cria a display list para a gangorra
if(glIsList(2))
{
     glDeleteLists(2,1);
}
glNewList(2, GL_COMPILE);
glPushMatrix();
glTranslatef(-14.0, 5.7, 5.0);
glRotatef(rx, 0, 0, 1);
drawCube(-1.0,1.0, 0.0, 0.1, 0.0, 0.5 );
glPopMatrix();
// Fim da lista
glEndList();

glCallList(2);

void simpleApp::postFrame()
{
  //angulo da rotacao da gangorra esta decrementando
  static bool decrementando;
  float revs_per_second = 1.0f;
  float degs_per_revolution = 25.0f;
  float degs_per_second = degs_per_revolution * revs_per_second;
  timer.stopTiming();
  timer.startTiming();
  if (decrementando)
  {
     rx += timer.getLastTiming() * degs_per_second-5;
     if (rx < -25.0) decrementando = false;
  }
  else
  {
     rx += timer.getLastTiming() * degs_per_second+5;
     if (rx > 25.0) decrementando = true;
  }
}
```

Figura 4.21: Código da movimentação da gangorra - VRJuggler

Dependendo dos arquivos de configuração usados, são abertas janelas que gerenciam a câmera e outros dispositivos. No caso da aplicação foi utilizado apenas um arquivo de configuração, que já veio com a biblioteca, o qual abre uma janela, além da janela da aplicação, a qual inicializa a câmera e todos os outros dispositivos necessários. Não existe nenhum arquivo de configuração em que todo o gerenciamento e a renderização são feitos em uma janela só. Porém a nova versão desta biblioteca, ainda não finalizada, provê este arquivo de configuração. A câmera é inicializada no arquivo de configuração na mesma posição que o seu *proxy*. Como os dispositivos nunca são acessados diretamente pela aplicação, o acesso a mesma deve ser feito através de *proxies*, que nada mais é que um intermediário que passa as informações entre as duas partes. No caso de se saber a posição da câmera, basta usar a função **getData()**, que retorna a matriz que define a câmera (posição e rotações). Os eventos do mouse, assim como os do teclado, são definidos no arquivo de configuração. Por causa disso a navegação também é feita através do arquivo de configuração.

De acordo com Patrick Hartling[8], é melhor obter a posição da câmera pelo proxy *head* existente no arquivo de configuração. Se a configuração for para um ambiente imersivo, a câmera será os olhos do usuário, e se pode concluir que o corpo e a cabeça estão no mesmo plano, trabalhando-se, portanto, apenas com a cabeça. O *proxy head* usado para se obter a posição da câmera não pode ser usado para mudar esta posição, pois é apenas de leitura. A mudança dessa posição talvez possa ser feita através da configuração de um dispositivo no arquivo de configuração chamado *vjDummyPosition*, cuja matriz pode ser manipulada pela aplicação. Porém, isto só serve para o modo *simulator*, ou seja, ambientes não imersivos.

Como não existe tutorial para a mudança de arquivos de configuração e os mesmos não possuem qualquer comentário, foi possível obter a posição da câmera e verificar se há colisão com os objetos do ambiente. Porém, fazer com que a câmera não ultrapasse estes objetos foi um objetivo não alcançado, haja vista a dificuldade em se entender os arquivos de configuração. Os resultados da implementação podem ser vistos nas figuras 4.22 e 4.23.

A aceitação de VRJuggler por parte da comunidade desenvolvedora de ambientes virtuais não é tão grande. Os projetos, que utilizam esta biblioteca, encontrados são: *CuevaDeFuego*, que mostra o potencial de visualização foto-realística de uma estrutura geológica utilizando ambientes sintéticos, *CFD in VR*, que trabalha

[8]HARTLING, P. (patrick@137.org). *Doubts about collision test*. E-mail para SCHNEIDER, A. R. A. (andreia@comp.ufla.br). 5 de Maio de 2003

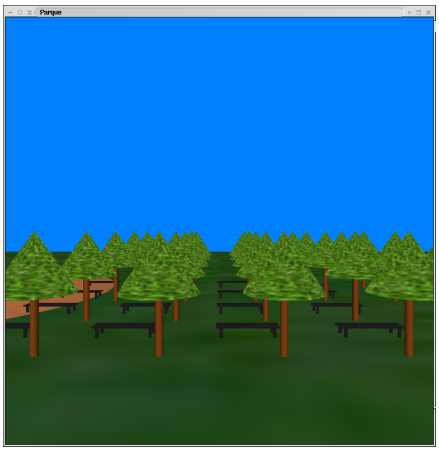

Figura 4.22: ambiente criado no VRJuggler

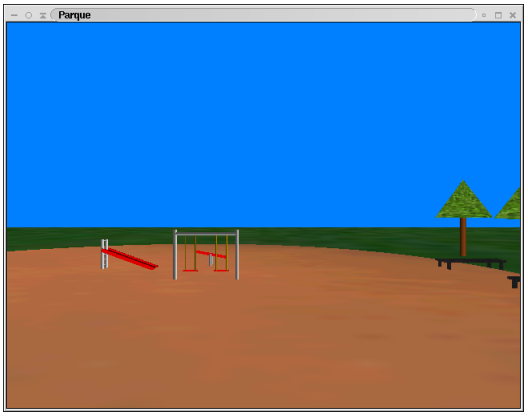

Figura 4.23: ambiente criado no VRJuggler - parquinho

com dinâmica de fluídos, *Cave Quake]I[Arena*, criação de um ambiente próprio para cavernas digitais e *MetVR*, utilizado para visualização meteorológica e oceanográfica pela Universidade do Estado do Mississipi (EUA). Todos estes projetos podem ser encontrados em: `http://www.vrjuggler.org/gallery.php`. [Visitado em 27 de Maio de 2003]. Vale ressaltar que a única caverna digital da América Latina, localizada na Politécnica da USP, utiliza VRJuggler para criar

seus ambientes.

4.3.2 Considerações sobre a biblioteca VRJuggler

A biblioteca VRJuggler, asim como as anteriores, é de fácil obtenção, podendo ser baixada através da página `http://www.vrjuggler.org/download.main.php` [visitado em 30/01/2003]. Para instalar, como o sistema operacional da máquina utilizada não é RH 7.2, o código fonte da biblioteca teve que ser baixado e compilado novamente. A variável VJ_BASE_DIR deve ser inserida no arquivo de inicialização do SHELL (no caso .bashrc). Para se compilar um programa basta se ter um makefile para o mesmo. No caso aproveitou-se um makefile já existente de outra aplicação. Para executar um programa deve ser passado o nome do executável mais os arquivos de configuração necessários. Como, por exemplo:

> ./torus /home/andreia/PO/share/Data/configFiles/sim.base.config
>
> /home/andreia/PO/share/Data/configFiles/sim.displays.config
>
> /home/andreia/PO/share/Data/configFiles/sim.wand.mixin.config

Como dito antes, a navegação é definida no arquivo de configuração usado, o que pode ser considerada uma vantagem, pois não é necessário modificar o programa cada vez que se deseja uma navegação diferente. O que dificulta com relação aos arquivos de configuração é quando se torna necessário modificá-los. O desenvolvedor tem que entender a linguagem do mesmo, o que significa mais informações a guardar. O maior problema é a dificuldade em aprender a manipular os arquivos de configuração. Há uma interface gráfica feita em JAVA para tentar facilitar o aprendizado. Porém o desenvolvedor fica meio perdido face a tantas definições de novos objetos (proxies, vjdummyproxies, etc.). Sem saber o que significa cada um deles fica difícil modificar este arquivo de forma aceitável. Há um tutorial que explica como criar novos arquivos de configuração, porém são escritos em um nível muito baixo, deixando o desenvolvedor em dúvida em muitos casos, como por exemplo, "como criar um *vjDummyPosition* e associá-lo à câmera de maneira que se possa modificar sua posição?"

O tamanho do executável gerado é de 516K, muito maior que o executável da implementação em Maverik, porém menor que o do vrJuggler, entretanto a biblioteca ocupa, em disco, 44M. O programa gerado é eficiente, já que responde rapidamente às ações do usuário. No teste feito para se verificar a velocidade de interação, após a inserção 800 cilindros (caules) o intervalo da apresentação dos *frames* começou a ser notado, um valor relativamente baixo, se comparado

aos valores obtidos nas outras bibliotecas. Toda a programação é feita utilizando C++ e OpenGL, o que faz com que o desenvolvedor tenha um conhecimento desta linguagem e da biblioteca.

Capítulo 5

Considerações finais

5.1 Considerações gerais

Após o estudo das três bibliotecas pôde-se perceber que as três são de fácil obtenção e instalação, sendo que dentre elas a que se mostrou de mais fácil aprendizado e assimilação foi a Maverik.

Observou-se também a importância de uma documentação fácil de se entender e completa, facilitando assim o aprendizado. Neste quesito a biblioteca que mais chegou perto de uma documentação completa e fácil de se usar foi a Maverik, o que facilitou muito seu aprendizado. Em compensação, devido ao fato de PLIB não possuir uma documentação, apenas um arquivo com vários exemplos mal documentados, foi muito difícil o aprendizado da mesma. Apesar do VRJuggler possuir uma documentação vasta, ela não é tão simples como a do Maverik, fazendo com que alguns aspectos a serem estudados sejam de difícil assimilação.

Devido ao fato de VRJuggler se preocupar com o gerenciamento dos dispositivos, a programação do mesmo se torna mais fácil que a das outras bibliotecas, desde que o desenvolvedor saiba como implementar o arquivo de configuração, já que o mesmo não tem que se preocupar com detalhes de gerenciamento. Se o desenvolvedor souber manipular estes arquivos de configuração, basta que ele conheça a arquitetura do dispositivo que ele deseja usar e fazer um arquivo de configuração baseado na arquitetura deste dispositivo. Desta forma, qualquer dispositivo pode ser usado.

Não se tem a pretensão de, através deste trabalho, indicar uma biblioteca que seja sempre a melhor para qualquer ambiente a ser desenvolvido com qualquer dispositivo. O que se deseja é indicar, para determinados casos, qual é a melhor

biblioteca para se usar. Porém, nada impede que o desenvolvedor escolha outra biblioteca para usar. O que se deve ter em mente é que devem ser levadas em conta todas as características de um sistema (dispositivos a serem usados, sistemas operacionais em que deve rodar, conhecimento do desenvolvedor, entre outras características) para se escolher uma biblioteca que cobrirá todos estes itens. Entretanto o que se pode afirmar é que a biblioteca Maverik é indicada para ambientes mono-usuário, em que o fator realismo seja primordial, sem adição de dispositivos que não os que a mesma suporta por padrão, sem adição de som ou recursos de vídeo e sem detecção de colisão entre volumes de sólidos, devido ao fato da mesma não possuir funções de detecção de colisão entre sólidos, tão importante na implementação de um ambiente virtual, não sendo também criadora de ambientes multi-usuário. Um fato que deve ser acresentado é que o desenvolvedor pode fazer suas próprias funções para adicionar som e vídeo e para detectar colisão entre sólidos. O que ele deve observar é o que será mais vantajoso: fazer estas funções ou usar uma bibioteca que já tenha isto implementado.

Já a biblioteca PLIB é indicada para ambientes que utilizem efeitos especiais, tais como fogo, ondas e arranjo de mola e amortecedor. (o que explica o fato de ela ser tão popular entre os desenvolvedores de jogos), mas que sejam feitos para os dispositivos que já possuam funções implementadas, pois como não há documentação e os exemplos existentes são mal comentados. A utilização desta biblioteca só seria interessante quando for primordial o uso de funções consideradas de difícil implementação e já implementadas na própria biblioteca (como ssgaParticleSystem, que simula um *spray* de partículas).

VRJuggler, no entanto, pode ser considerada a biblioteca mais completa das três. Devido ao fato de VRJuggler se preocupar com o gerenciamento dos dispositivos, a programação do mesmo se torna mais fácil que a das outras bibliotecas. O que dificulta nela é a implementação do arquivo de configuração. Entretanto, se o desenvolvedor tiver um conhecimento profundo sobre seus arquivos de configuração, poderá criar seus próprios, podendo executar sua aplicação em qualquer arquitetura, sem modificar o código do programa. Como a maioria dos ambientes virtuais são implementados para serem executados em várias máquinas diferentes, isto é uma vantagem considerável. Uma desvantagem é que como a implementação dos objetos é em OpenGL, seus métodos são mais baixo nível que os das outras bibliotecas, sendo mais difícil a abstração. Desta forma, para aplicações que exijam novos dispositivos ou arquiteturas novas, ou que sejam executadas em várias máquinas diferentes, VRJuggler é uma boa indicação, pois o desenvolvedor tem apenas que criar um arquivo de configuração para as várias arquiteturas

e dispositivos e utilizar a mesma aplicação. O VRJuggler traz vários arquivos de configuração já prontos com a distribuição da biblioteca. Com isto a utilização de sistemas mais comuns (como cavernas digitais, pcs, etc.) ficam mais simples. Os resultados das comparações feitas podem ser vistos na tabela 5.1.

Tabela 5.1: Comparação entre as três bibliotecas

Critérios de comparação	$Maverik$	$PLIB$	$VRJuggler$
facilidade de uso	fácil	difícil	difícil de criar arq. de configuração
obtenção	fácil	fácil	fácil
documentação disponível	dois tutoriais	exemplos mal comentados	um tutorial
teste de eficiência (cilindros)	1000	2500	800
Orientação a objeto	não	sim	sim
tamanho do executável	25k	2.3M	516k
espaço ocupado em disco	2M	16M	44M
quantidade de projetos	32 (grandes projetos)	11 (jogos)	5

5.2 Considerações para o caso do LCC

No caso do **LCC** (Laboratório de Computação Científica da UFLA), os dispositivos utilizados serão uma estação gráfica com *display* estéreo, além de um *spaceball*. Maverik é a única biblioteca que possui suporte a *spaceball* já implementado, funções de visão estéreo fáceis de serem usadas, além de produzir executáveis de tamanho pequeno e ocupar um espaço considerado pequeno em disco. A primeira vista, no caso do **LCC**, Maverik parece ser a melhor biblioteca, mas tanto neste laboratório como em qualquer outro caso, a complexidade do ambiente a ser desenvolvido (objetos que o compõem, arquitetura usada, dispositivos de entrada e saída, distribuído ou não, etc.) deve ser analisada e, dependendo da mesma, a biblioteca mais indicada para este caso deve ser usada. Por exemplo, as vezes pode ser muito mais fácil criar um suporte para um dispositivo em uma determinada biblioteca, ao invés de implementar funções tidas como complexas, tais como jato de fogo e *spray* de água, já existentes em uma biblioteca que não suporta o dispositivo desejado. Desta forma, escolher uma biblioteca depende muito do ambiente a ser gerado e do tempo e disposição que o desenvolvedor tem para aprender a usar uma determinada biblioteca.

Referências Bibliográficas

[GNU Maverik (2002)] *Advanced Interfaces Group*,GNU Maverik, disponível em `http://aig.cs.man.ac.uk/maverik/`. Citada em 22 de Novembro de 2002

[BELL & FOGLER (1995)] BELL, J. T. & FOGLER, H.S. *The Investigation and Application of Virtual Reality as an Educational Tool*. In Proceedings of the American Society for Engineering Education Annual Conference. June 1995.

[Bierbaum (2000)] Bierbaum, A. D. *VR Juggler: A Virtual Platform for Virtual Reality Application Development*, 2000. Tese (mestrado em engenharia da computação)- Iowa State University

[Foley & DAM (1996)] FOLEY, J. D.; HUGHES, J. & DAM, A. van *Computer Graphics : Principles and Practice, Second Edition in C*. Addison-Wesley Pub Co. 1996.

[FSF.GNU (2002)] *FSF.GNU Library General Public license*, disponível em `http://www.gnu.org/copyleft/library.html`. Citada em Novembro de 2002

[FSF.GNU's not Unix (2002)] *FSF. GNU s not Unix! the GNU project and the Free Software Foundation (FSF)*, disponível em `http://www.gnu.org`. Citada em janeiro de 2002.

[Gomes (2001)] GOMES, R. S. *Pesquisa de Bibliotecas Multiplataforma para programas multimídia*, 2001. Monografia (conclusão de graduação em Ciência da Computação) - Departamento de Ciência da Computação, Universidade Federal de Lavras, Lavras/MG.

[Heller & BRENNAN (1994)] HELLER, D; FERGUSON, P. M. & BRENNAN, *D. Motif Programming Manual (The Definitive Guides to the X Window System, Volume 6A*. O Reilly & Associates. Pp 1016. 1994.

[Kirner & Ipólio (1996)] KIRNER, C.; ARAÚJO, R. B. & IPÓLIO, J. R. *Sistemas de Realidade Virtual: Aspectos, distribuição e Programação na Internet*. Grupo de Realidade Virtual. DC/UFSCar. Agosto de 1996

[Martins & Kirner (1997)] MARTINS, V. F. & KIRNER, T. G. *Processo de Desenvolvimento de Ambientes Virtuais: Definição e um Estudo de Caso*. In: Primeiro Workshop de Realidade Virtual. Novembro de 1997. São Carlos. Anais, 1997, p.119

[Open Source (2002)] *Open source*, disponível em `http://www.opensource.org`, citada em 25 de Novembro de 2002

[PLIB (2002)] *Steve's Portable Game Library*, disponível em `http://plib.sourceforge.net/`. Citada em 25 de Novembro de 2002

[Vince (1999)] VINCE, J. *Essencial Virtual Reality fast: how to understand the techniques and potential of virtual reality*. Springer.1999

[Vince (1995)] VINCE, J. *Virtual Reality systems*. Addison-Wesley Pub Co. 1995

[VR Juggler (2002)] *VR Juggler - Open Source Virtual Reality Tools*, disponível em `http://www.vrjuggler.org/`. Citada em 25 de Novembro de 2002

[Watt & Watt (1992)] WATT, A. H. & WATT, M. *Advanced Animation and Rendering Techniques: Theory and Practice*. Addison-Wesley Pub Co. 1992.