UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO

(Bacharelado)

PROTÓTIPO DE UM AMBIENTE VIRTUAL DISTRIBUÍDO MULTIUSUÁRIO

TRABALHO DE CONCLUSÃO DE CURSO SUBMETIDO À UNIVERSIDADE REGIONAL DE BLUMENAU PARA A OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA COM NOME EQUIVALENTE NO CURSO DE CIÊNCIAS DA COMPUTAÇÃO — BACHARELADO

LEONARDO WILLRICH

BLUMENAU, JUNHO/2002.

2002/1-46

PROTÓTIPO DE UM AMBIENTE VIRTUAL DISTRIBUÍDO MULTIUSUÁRIO

LEONARDO WILLRICH

ESTE TRABALHO DE CONCLUSÃO DE CURSO FOI JULGADO ADEQUADO PARA OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA DE TRABALHO DE CONCLUSÃO DE CURSO OBRIGATÓRIA PARA OBTENÇÃO DO TÍTULO DE:

BACHAREL EM CIÊNCIAS DA COMPUTAÇÃO

____

Prof. Dalton Solano dos Reis - Orientador na FURB

____

Prof. José Roque Voltolini da Silva - Coordenador do TCC

BANCA EXAMINADORA

____

Prof. Dalton Solano dos Reis

____

Prof. Paulo César Rodacki Gomes

____

Prof. Alexander Roberto Valdameri

AGRADECIMENTOS

Gostaria de agradecer a Deus por me dar oportunidade e a saúde necessária para a conclusão deste trabalho. Agradeço minha noiva, Karina Marquardt, por ter me apoiado e me dado incentivo para concluir o mesmo. Minha família, pois foram muitos importantes não só para a conclusão deste trabalho, mas também para a minha formação como acadêmico.

Quero também agradecer aos professores que me acompanharam durante a minha formação e também os monitores, estagiários e funcionários dos laboratórios da FURB, especialmente do laboratório PROTEM, que me deram a maior ajuda quando precisei configurar o ambiente para rodar o protótipo.

Agradeço de maneira muito especial o meu orientador, Dalton Solano dos Reis, que deu o maior apoio e me orientou de forma correta para a conclusão deste trabalho.

SUMÁRIO

RESUMO IX
ABSTRACT X
1 INTRODUÇÃO 1
1.1 CONTEXTUALIZAÇÃO / JUSTIFICATIVA 1
1.2 OBJETIVOS 3
1.3 ORGANIZAÇÃO DO TRABALHO 3
2 AMBIENTES VIRTUAIS DISTRIBUÍDOS 4
2.1 ASPECTOS GERAIS DE COMUNICAÇÃO DE UM AVD 5
2.2 COMUNICAÇÃO ENTRE OS USUÁRIOS DE UM AVD 8
2.3 HEARTBEATS 10
2.4 DISTRIBUTED INTERACTION SIMULATION (DIS) 12
2.5 DIS-JAVA-VRML 16
3 OPENGL 18
3.1 VANTAGENS E DESVANTAGENS 19
3.2 FUNCIONAMENTO 20
3.2.1 FUNCIONAMENTO DA OPENGL COM O WINDOWS 22
3.2.2 FUNCIONAMENTO DA OPENGL COM O AMBIENTE DE PROGRAMAÇÃO BORLAND DELPHI 22
3.3 COMANDOS BÁSICOS 23
3.4 DESENHANDO-SE COM A OPENGL 24
3.5 OPENGL COMO UMA MÁQUINA DE ESTADO 24
3.6 PERSPECTIVA 25
3.7 TESTE DE PROFUNDIDADE 26
3.8 TRANSFORMAÇÕES 26
3.8.1 EYE COORDINATES 27
3.8.2 VIEWING TRANSFORMATIONS 28
3.8.3 MODELING TRANSFORMATIONS 28
3.9 TEXTURAS 31
3.10CORES, LUZES E MATERIAIS 32
3.11O QUE MAIS É POSSÍVEL COM OPENGL 33
4 OBJETOS DISTRIBUÍDOS (CORBA) 34
4.1 CORBA 37
4.1.1 ARQUITETURA CORBA 38
4.1.1.1 Aplicações clientes 39
4.1.1.2 Aplicações servidoras 41
5 DESENVOLVIMENTO DO PROTÓTIPO 43
5.1 REQUISITOS IDENTIFICADOS 43
5.2 ESPECIFICAÇÃO E IMPLEMENTAÇÃO 43
5.2.1 DIAGRAMA DE CLASSES 44
5.2.2 CLASSE CENARIO 45
5.2.3 CLASSE PERSONAGEM E SUAS SUBCLASSES 51
5.2.4 CLASSE OPENGL 53
5.2.5 CLASSE TECLADO 53
5.2.6 CLASSE HEARTBEAT 55

5.2.7 CLASSE INFOAVD 55
5.2.8 CLASSES DIS E TRATARMOVIMENTOSCENARIO 56
5.2.9 CLASSES DE INTERFACE........ 60
5.3 FUNCIONAMENTO DO PROTÓTIPO 62
5.3.1 JANELA DE OPÇÃO 63
5.3.2 JANELA DE INFORMAÇÕES DO AVD 63
5.3.3 JANELA DO CENÁRIO 64
5.4 ANÁLISE DOS RESULTADOS 65
6 CONCLUSÕES 68
6.1 EXTENSÕES 68
ANEXO A: PRINCIPAIS MÉTODOS DA CLASSE CENARIO........ 70
ANEXO B: PASSOS A SEREM SEGUIDOS PARA A EXECUÇÃO DO PROTÓTIPO..... 74
REFERÊNCIAS BIBLIOGRÁFICAS 76

LISTA DE FIGURAS

FIGURA 2.1 - Modelo centralizado de comunicação 5
FIGURA 2.2 - Modelo de comunicação distribuído 6
FIGURA 2.3 - Mundo virtuais replicados e particionados........ 7
FIGURA 2.4 - Envio de mensagens utilizando *unicast*, *broadcast* e *multicast* 9
FIGURA 2.5 - Organização geral do projeto DIS-Java-VRML........ 17
FIGURA 3.1 – Implementação Genérica e Implementação de Hardware 21
FIGURA 3.2 – Projeção ortográfia e projeção em perspectiva........ 25
FIGURA 3.3 – Sistema eye coordinates........ 27
FIGURA 3.4 – Sistema de coordenadas rotacionados em relação a eye coordinates 28
FIGURA 3.5 – Transformações geométricas 29
FIGURA 3.6 – Ordem das transformações 30
FIGURA 4.1 - Troca de mensagens entre camadas do sistema........ 34
FIGURA 4.2 – Mapeamento da IDL para as linguagens de programação........ 38
FIGURA 4.3 - *Stubs* e *Skeletons* na estrutura CORBA........ 39
FIGURA 5.1 – Diagramas de classes do protótipo 45
FIGURA 5.2 – Cálculo da coordenada Y para quando o usuário estiver na rampa........ 47
FIGURA 5.3 – Fluxograma do método CriarCenario() 48
FIGURA 5.4 – Fluxograma do método Desenhar() 49
FIGURA 5.5 – Cálculo do novo ponto de visão........ 50
FIGURA 5.6 – Fluxograma especifiando a estrutura da classe Teclado........ 54
FIGURA 5.7 – Fluxograma especificando a estrutura da classe Heartbeat 55
FIGURA 5.8 – Tratamento dos PDU´s recebidos 59
FIGURA 5.9 – Troca de mensagens nas classes de interface 61
FIGURA 5.10 – Janela de opção........ 63
FIGURA 5.11 – Janela de informações do AVD 64
FIGURA 5.12 – Janela de interface........ 65

LISTA DE QUADROS

QUADRO 3.1 – Código fonte exemplificando a Unit de Interface 23
QUADRO 3.2 – Comandos básicos 23
QUADRO 3.3 – Comandos de desenho 24
QUADRO 3.4 – Comandos de estado 24
QUADRO 3.5 – Coordenadas de dois segmentos de reta paralelos.. 25
QUADRO 3.6 – Comando utilitário para definir a perspectiva da cena 26
QUADRO 3.7 – Comandos de transformações.. 30
QUADRO 3.8 – Comandos para utilização de texturas em polígonos 31
QUADRO 5.1 – Descrição dos métodos da classe Cenario 46
QUADRO 5.2 – Implementação do método Desenhar()... 51
QUADRO 5.3 – Método DesenharPersonagem() da classe PersonagemCubo........................ 52
QUADRO 5.4 – Descrição dos métodos da classe OpenGL.. 53
QUADRO 5.5 – Código fonte exemplificando a inicialização do objeto DIS......................... 57
QUADRO 5.6 – Código fonte dos métodos da classe DIS 57
QUADRO 5.7 – IDL dos objetos de interface... 62
QUADRO 6.1 – Implementação dos principais métodos da classe Cenario............................ 70

LISTA DE SIGLAS E ABREVIATURAS

2D	Duas Dimensões
3D	Três Dimensões
API	*Application Program Interface*
ARPA	*Advanced Research Projects Agency*
AVD	Ambiente Virtual Distribuído
CORBA	*Common Object Request Broker Architecture*
DIS	*Distributed Interactive Simulation*
DLL	*Dinamic-Link Library*
ESPDU	*Estity State Protocol Data Unit*
GDI	*Graphic Device Interface*
LAN	*Local Area Network*
NPSNET	*Naval Postgraduate School Network*
NURBS	*non-uniform rational B-spline*
OD	Objeto Distribuído
PDU	*Protocol Data Unit*
SIMNET	*Simulator Network*
UDP	*User Datagram Protocol*
VRTP	*Virtual Reality Transfer Protocol*
WAN	*Wide Area Network*
WIMP	*Window, Icon, Menu, Pointer*
WoW	*Window on World*

RESUMO

Este trabalho apresenta fatores relacionados à construção de ambientes virtuais que estejam distribuídos entre vários usuários. Mais especificamente, ele procura abordar as características relevantes a ambientes virtuais distribuídos. Entre estas, se destacam as técnicas e recursos de comunicação entre usuários e de locomoção dentro de um ambiente virtual. Para o emprego das técnicas de comunicação entre usuários foi utilizada a *Application Program Interface* (API) do DIS-Java-VRML, que foi desenvolvida na linguagem Java. Para o desenvolvimento da interface com o usuário foi utilizada a biblioteca gráfica OpenGL juntamente com a linguagem de programação Object Pascal. Também foi necessária a utilização de técnicas para comunicar objetos distribuídos. Isto se deve pela escolha das duas API´s a serem utilizadas (DIS-Java-VRML como suporte para comunicação entre usuários e a OpenGL para criação do ambiente virtual) que são implementadas em linguagens diferentes. A partir deste momento, passou-se a ter duas aplicações que fazem parte do mesmo sistema e que objetos implementados nestas aplicações necessitam trocar mensagens entre si. Por este motivo, optou-se pelo padrão CORBA para a implementação de objetos distribuídos.

ABSTRACT

This work presents factors relate to the construction of virtual environment that are distributed between several users. More specifically, it tries to approach the relevant characteristics to distributed virtual environments. Among these, it stands out the techniques and resources of communication between users and to move inside of a virtual environment. For use the techniques of communication between users, the Application Program Interface (API) of DIS-Java-VRML was used, and developed in Java language. For development of interface with the user the OpenGL library was used together with the Object Pascal language. It was also necessary the use of techniques for communication between distributed objects. This is because choose of this two APIs that will be use (DIS-Java-VRML for support of communication between users and the OpenGL for construction the virtual environments) that will be implemented in different languages. After this moment, the job has two applications that make part of the same system and that implemented object in these applications need send and receive messages between them. For this reason, was choosing for the CORBA standard for the implementation of the distributed objects.

1 INTRODUÇÃO

1.1 CONTEXTUALIZAÇÃO / JUSTIFICATIVA

Com o passar dos tempos a interface entre o usuário e os computadores passou a ser mais amigável de tal maneira que qualquer pessoa, independente do seu grau de instrução, pudesse conseguir atingir determinados objetivos com o uso de computadores. Isto se deve pelo fato da evolução que se teve com estudos na área de interfaces e também ao avanço e ao amadurecimento na construção de programas. E, com isso, objetivou-se tornar a interação do usuário mais natural possível com o computador.

A maioria das interfaces construída apresentava as informações aos usuários somente através de imagens em duas dimensões (2D) e geralmente através de um monitor. Como a interação do usuário com o mundo real é realizado em três dimensões (3D), começou-se a desenvolver aplicações que utilizassem imagens em 3D também. O objetivo é tornar a interface do usuário mais próxima da realidade. E com isso, começam a surgir aplicações que simulam a realidade do mundo real para se criar uma interface com os usuários. Essas aplicações passam a ser chamadas de ambientes virtuais.

Segundo Pinho (1999), um ambiente virtual é um cenário em três dimensões gerado computacionalmente através de técnicas de computação gráfica, procurando sempre representar a parte visual de um ambiente real ou imaginário. Geralmente, esses ambientes são caracterizados pela sua geração dinâmica e pela capacidade dos usuários interagirem com ele.

As primeiras interfaces que surgiram para os ambientes virtuais eram do tipo *Window*, *Icon*, *Menu*, *Pointer* (WIMP) (Pinho, 1999) ou *Window on World System* (WoW) (Isdale, 1993) e eram projetas em monitores comuns e utilizavam o teclado e o mouse como dispositivos de entrada. Com a evolução dos tempos começou-se o desenvolvimento de equipamentos mais específicos para estes ambientes virtuais, como capacetes, luvas e salas de projeção. Dessa forma a interação com os ambientes virtuais gerados por computador deixa de ser não imersiva, e passa a imergir o usuário dentro dos ambientes virtuais, procurando fazer com que o mesmo passe a se sentir dentro do próprio ambiente gerado. Dessas duas formas de interação com os mundos virtuais surge a primeira classificação de ambientes virtuais: os imersivos e não imersivos.

Uma outra característica também é atribuída a estes ambientes virtuais, que é a capacidade desses de suportarem um ou mais usuários e também permitirem que esses usuários compartilhem o mesmo ambiente virtual estando em computadores separados.

Com essa característica o ambiente virtual passa a ter uma série de fatores que devem ser considerados e levados em conta na hora de seu desenvolvimento. Segundo Macedonia (1997), os fatores são os seguintes:

a) limitação da largura de banda da rede sobre a qual o ambiente virtual está interconectado;
b) melhor forma de distribuir o mundo virtual (se de forma centralizada ou distribuída); como contornar os problemas de latência;
c) confiabilidade do ambiente virtual que trata as questões de sincronia e consistência entre os vários usuários participantes do mesmo.

Tomando-se em conta estes fatores, procurou-se desenvolver um protótipo de ambiente virtual multiusuário e distribuído, com uma interface não imersiva, permitindo implementar na prática as técnicas pesquisadas que tratam desses quatro fatores descritos.

Para a realização desse trabalho também foi necessário o estudo de duas API's: a API da biblioteca gráfica OpenGL (Silicon, 2001) e a API do DIS-Java-VRML (Web3D, 2001). A primeira foi utilizada para a construção do ambiente virtual e a segunda para a implementação dos mecanismos de controle da conexão entre os usuários do ambiente virtual.

Além do estudo das duas API´s também foi necessário o estudo de objetos distribuídos, pois as duas API´s serão implementadas em linguagens diferentes e existe a necessidade de troca de mensagens entre elas. Segundo Geyer (2000), o padrão *Common Object Request Broker Architecture* (CORBA) do grupo *Object Management Group* (OMG) propõe uma arquitetura de software para suportar objetos distribuídos e garantir a interoperabilidade entre diferentes plataformas de hardware e sistemas operacionais. Esta capacidade é obtida através do uso de uma interface comum e um mecanismo de passagem de informações implementado em diferentes linguagens de programação, entre elas o Borland Delphi e o Java.

Dessa forma, este trabalho permite que se tenha uma visão geral de algumas características importantes que precisam ser observadas na construção de ambientes virtuais distribuídos. Também verifica a viabilidade da utilização da biblioteca gráfica OpenGL, da API DIS-Java-VRML e do padrão CORBA para a implementação do protótipo.

Com isso, pôde-se estudar e aplicar de forma prática algumas técnicas já desenvolvidas para construção de ambientes virtuais multiusuário distribuídos e a utilidade dessas duas API's mencionadas no processo de construção do ambiente virtual e do controle das conexões entre os usuários.

1.2 OBJETIVOS

O objetivo principal do trabalho é implementar um protótipo de ambiente virtual distribuído sobre uma rede local, com suporte a multiusuários e com uma interface não imersiva.

Os objetivos específicos do trabalho são:

a) criar um ambiente virtual distribuído composto por um cenário e objetos simples;
b) permitir que este ambiente possibilite multiusuários interagirem com os objetos e entre os usuários.

1.3 ORGANIZAÇÃO DO TRABALHO

O trabalho está organizado conforme descrito abaixo.

O capítulo dois concentra-se na abordagem dos ambientes virtuais distribuídos, principais problemas encontrados na operacionalização do mesmo, algumas técnicas utilizadas para contornar esses problemas e também a API do DIS-JAVA-VRML.

O capítulo três apresenta uma visão geral da biblioteca gráfica OpenGL, mostrando o seu funcionamento, suas principais características e alguns de seus principais comandos.

O capítulo quatro apresenta os conceitos de objetos distribuídos e de CORBA.

O capítulo cinco apresenta o desenvolvimento do protótipo. Mas especificamente são apresentados a sua especificação, sua implementação e o seu funcionamento, demonstrando a aplicação de algumas técnicas pesquisadas nos capítulos anteriores. Neste capítulo também é apresentado as analises do resultado do trabalho.

O capítulo seis apresenta as conclusões alcançadas com o desenvolvimento e analise dos resultados deste trabalho. Também são apresentadas as extensões sugeridas para este trabalho.

2 AMBIENTES VIRTUAIS DISTRIBUÍDOS

Segundo Gossweiler (1994), ambientes virtuais são simulações de computador interativas que submergem os usuários a uma realidade acreditável. Pessoas movem-se ao redor, olham para diversas direções e manipulam objetos gráficos usando dispositvos de entrada que podem variar entre um simples teclado e o mouse até os mais exóticos dispositivos como os displays *head-mounted* e instrumentos de luva.

Uma aplicação ideal para ambientes virtuais é permitir que várias pessoas, ou jogadores, possam interagir dentro de uma simples simulação. Por exemplo: diversos estudantes em vários computadores conectados em uma rede (Gossweiler, 1994).

Com isso a forma de construção de ambiente virtual pode ser variada. Tendo-se aplicações para diversas áreas, como por exemplo, á área de educação, de simulação e de visualização científica, que podem estar rodando entre somente um usuário ou que podem estar interconectadas e distribuídos entre diversos usuários em uma rede qualquer.

As primeiras experiências com ambientes virtuais distribuídos sugiram a partir de aplicações com objetivos de simulações militares e também jogos com vários jogadores ao mesmo tempo. As simulações militares geralmente eram desenvolvidos por universidades dos Estados Unidas para o Departamento de Defesa, já as outras aplicações eram desenvolvidas por industrias que tinham interesses, principalmente em jogos multiusuários.

Segundo Eduardo (2001), entre os primeiros projetos de simulação desenvolvidos se destaca o *Simulator Network* (SIMNET). O SIMNET é um simulador de batalhas militares onde os integrantes do ambiente virtual assumem o controle de veículos militares como tanques, aviões e soldados para interagirem no ambiente virtual. Utilizando-se de redes de longa distância e *links* de satélite, o SIMNET permite a conexão e interação de centenas de usuários. Maiores informações sobre o SIMNET podem ser encontradas em Shaw (1993).

Outro projeto interessante que foi desenvolvido pela *Naval Postgraduate School* é o NPSNET. Este projeto também se refere a uma ambiente virtual que tem objetivo de realizar uma simulação militar e que tem como característica interconectar vários usuários, permitindo desta forma que a simulação aconteça com usuários de qualquer parte do mundo. Este trabalho foi apresentado na conferência de 1991 da Siggraph. Maiores informações podem ser encontradas em Macedônia (1995).

Com estes projetos tem-se a principal fonte de pesquisa para os principais aspectos que devem ser levados em conta para o desenvolvimento de ambiente virtual distribuído.

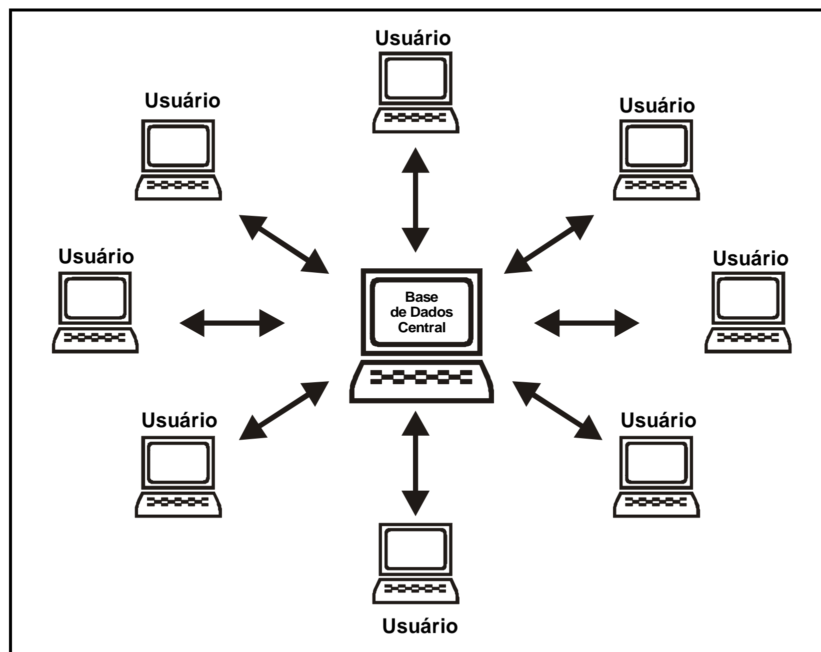
Entre estes aspectos estão relacionados fatores referentes à organização do modelo geral de distribuição do ambiente virtual, das técnicas a serem utilizadas e dos protocolos de comunicação mais recomendados até a atenção que se deve ser dada para as questões da utilização da largura de banda, latência e confiabilidade inerente à rede que está sendo utilizada para interconexão.

2.1 ASPECTOS GERAIS DE COMUNICAÇÃO DE UM AVD

Segundo Gossweiler (1994), existem basicamente dois modelos, que são os mais populares, para o desenvolvimento do AVD. Estes modelos são o modelo centralizado e o modelo distribuído.

No modelo centralizado existe um computador principal que tem a função de receber mensagens variadas de um usuário qualquer e repassá-las aos demais usuários conectados. Neste modelo o computador encarregado dessa função é conhecido também como computador central. Dessa forma, quando um usuário precisar enviar alguma mensagem aos outros usuários, ela obrigatoriamente tem que ser enviada ao computador central e, juntamente com a mensagem, deve existir a informação do destinatário da mesma. Essa forma de comunicação pode ser visualizada graficamente na fig. 2.1.

FIGURA 2.1 - MODELO CENTRALIZADO DE COMUNICAÇÃO



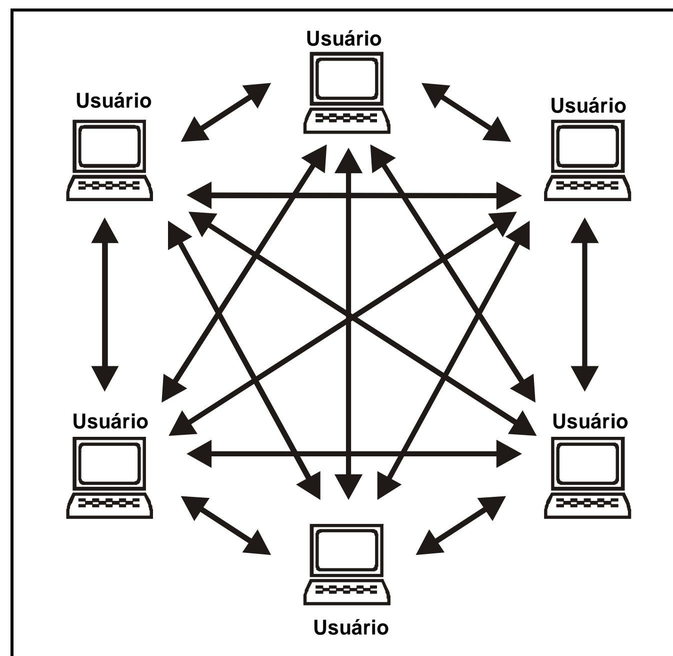
Fonte: adaptado de Gossweiler (1994)

Embora se pareça simples o desenvolvimento de AVD desse modelo, principalmente do mecanismo de controle e gerência de comunicação entre os usuários do AVD, existe o

problema da escalabilidade. A escalabilidade neste contexto se refere à capacidade do ambiente virtual suportar um número significativo de usuários. Pois neste modelo, quanto maior o número de usuários, maior será a quantidade de mensagens enviadas e recebidas pelo computador central, fazendo-se dessa maneira, o aumento no tráfego de mensagens com o computador central.

Já o modelo distribuído pode ser visto como uma maneira de resolver o problema da escalabilidade do modelo centralizado. Neste modelo, cada usuário é responsável pelo controle e gerenciamento da comunicação entre os demais usuários e, também é responsável pelo processamento gráfico da representação do AVD. Dessa forma, quando um usuário precisar enviar alguma mensagem, seja ela qual for, para os demais usuários, ele mesmo se encarrega de enviá-la para o destinatário. A fig. 2.2 ilustra esse modelo de comunicação.

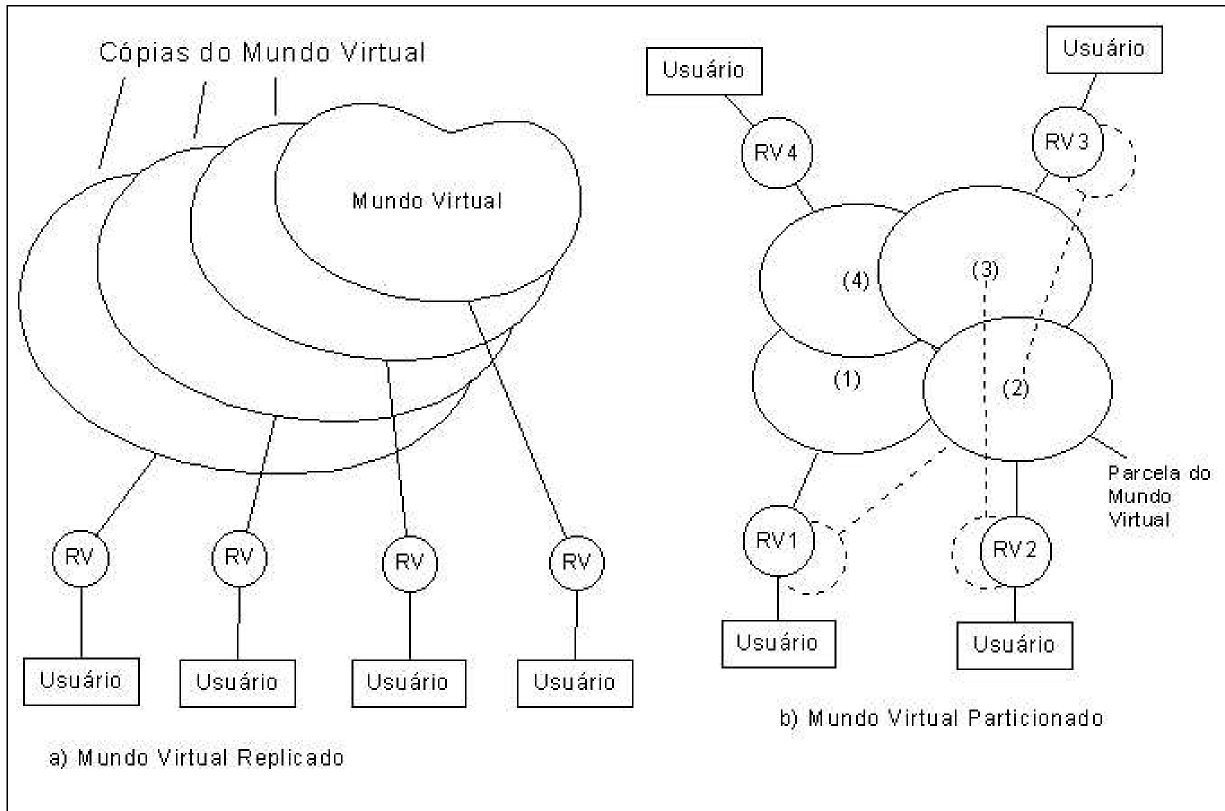
FIGURA 2.2 - MODELO DE COMUNICAÇÃO DISTRIBUÍDO



Fonte: adaptado de Gossweiler (1994)

Segundo Kirner (2000) no modelo distribuído o mundo virtual pode ser replicado para mundos pequenos ou particionado para mundos virtuais de grande porte, conforme a fig. 2.3.

FIGURA 2.3 - MUNDO VIRTUAIS REPLICADOS E PARTICIONADOS



Fonte: Kirner (2000)

Num sistema replicado com `n` usuários, quando um usuário fizer qualquer alteração no mundo virtual, isto deverá ser comunicado para todas as `n-1` versões do mundo virtual. Dessa maneira, as mensagens transmitidas sempre serão difundidas no ambiente virtual por completo, ou seja, um transmite e os demais recebem a mesma mensagem.

Já num sistema particionado com `n` usuários, a situação passa a ser mais complexa, uma vez que o mundo é divido em várias outras partes e cada usuário ficará encarregado de uma delas. Como o usuário é livre para navegar no mundo virtual, é possível que o usuário transite de uma região para outra, de forma que no momento da mudança o mundo virtual deverá ser trocado por uma réplica da nova região onde ele se encontra. Desta maneira, as transmissões sempre se realizaram em grupos que estarão definidos de acordo com as regiões que cada usuário fará parte.

Apesar do modelo distribuído resolver o problema da escalabilidade do ambiente virtual que fora encontrado no modelo centralizado, ele não é o modelo ideal ainda. Como neste modelo cada usuário fica responsável por enviar e receber mensagens do ambiente virtual para os outros usuários, ou pelo menos a um grupo de usuários, passa-se então a ter um número maior de mensagens geradas. E, além deste problema, o número de conexões também cresce, já que cada usuário precisa usar um canal de comunicação com os demais usuários.

Dessa maneira pode-se dizer que nenhum modelo é considerado como ideal. E, para cada ambiente virtual é importante observar-se os seguintes aspectos relacionados ao modelo de distribuição: a utilização da largura de banda (*bandwidth*), a latência (*latency*) e a confiabilidade ou garantia (*reability*) de que as mensagens serão recebidas por todos os usuários do AVD. Para maiores informações pode-se consultar em Macedonia (1995), Macedonia (1997), Stevens (1995), Szwarcman (1999) e Wloka (1995).

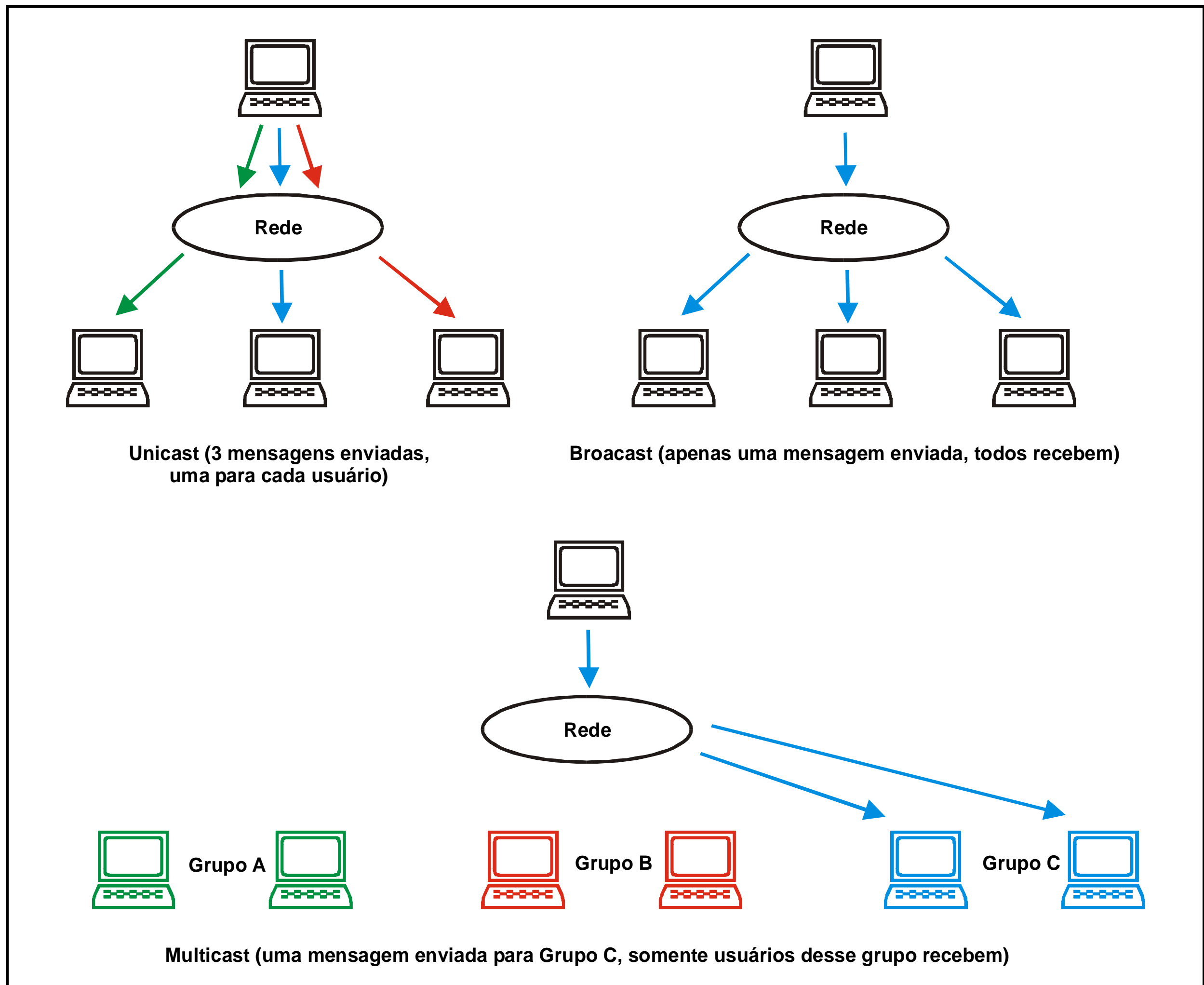
2.2 COMUNICAÇÃO ENTRE OS USUÁRIOS DE UM AVD

Segundo Eduardo (2001), há três formas básicas de comunicação entre os usuários de um AVD:

a) ***unicast:*** cada mensagem enviada para a rede deve conter o destinatário, que é único para cada mensagem enviada. Dessa forma, se o usuário fizer alguma alteração no ambiente virtual que necessite enviar alguma notificação desta para os outros usuários, o usuário terá que enviar está mensagem `n-1` vezes onde o `n` corresponde ao número de usuários existentes no ambiente virtual. Esta forma também é conhecida como ponto-a-ponto (*point-to-point*) devido a comunicação envolver somente dois usuários. É eficiente quando utilizado juntamente com o modelo centralizado;
b) ***broadcast:*** desta maneira será enviada somente uma mensagem e, através de um canal comum para os outros usuários, todos os usuários a receberam. Assim, a cada alteração do usuário que necessite enviar mensagens ao ambiente virtual, somente uma mensagem será enviada. Já esta forma de comunicação é mais eficiente junto com a utilização do modelo distribuído;
c) ***multicast:*** esta maneira de comunicação é semelhante a *broadcast*, só que ao invés de enviar mensagens para todos, são criados grupos que podem ser divididos por determinadas características e as mensagens são enviadas um determinado grupo. Esta forma de comunicação pode ser bem aproveitada para ambientes virtuais distribuídos e replicados.

A fig. 2.4 ilustra esses três métodos de comunicação.

FIGURA 2.4 - ENVIO DE MENSAGENS UTILIZANDO *UNICAST*, *BROADCAST* E *MULTICAST*



Fonte: Eduardo (2001)

Segundo Eduardo (2001), existem vantagens e desvantagens em cada um destes métodos de comunicação em relação ao desenvolvimento de ambientes virtuais.

No método *unicast*, se tem como vantagem aparente à sincronia das atualizações entre os usuários, visto que cada usuário tem que estabelecer uma conexão com os demais usuários. Porém, a desvantagem deste método, é que isso resultaria um número muito auto de conexões, tornado dessa forma muito difícil o gerenciamento e o controle das mesmas, principalmente com relação à criação de novas conexões para cada usuário novo no ambiente virtual. Além dessa desvantagem, também se pode citar o número elevado de mensagens geradas por este método. Segundo Macedonia (1997), este método requer o estabelecimento de uma conexão ou o mapeamento do caminho de cada usuário para os outros usuários, que acaba resultando num total de `n*(n-1)` conexões no ambiente virtual como um todo. Sendo que o `n` descrito anteriormente representa cada usuário do ambiente virtual.

Já o método *broadcast* apresenta-se como uma solução para o problema do alto número de conexões e mensagens geradas pelo *unicast*. Neste método não é necessário que exista uma conexão entre o usuário e os demais usuários do ambiente virtual. Ele se define pelo fato de que uma mensagem é enviada e de que todos os outros usuários pertencentes da mesma rede possam recebê-la. Para isso se tornar possível, deve existir um canal comum e de conhecimento de todos os outros usuários. Dessa maneira o usuário deixa de transmitir `(n-1)` vezes cada mensagem enviada ao ambiente e somente transmite uma vez só. Este método, segundo Eduardo (2001), é útil principalmente sobre AVD´s desenvolvidos para funcionarem sobre LAN´s.

Segundo Gossweiler (1994), o método *broadcast* apresenta como desvantagem o uso na maioria dos casos o protocolo UDP *broadcast*, já o método *unicast* geralmente faz uso do protocolo TCP *unicast*. E, a principal diferença entre estes dois protocolos é o confinamento do protocolo UPD sobre a rede local. Dessa forma o funcionamento de um AVD que utilize o protocolo UDP fica limitado somente a LAN´s.

Mas, como vantagem apontado por Gossweiler (1994), pode-se citar a facilidade e o tempo que se leva para a implementação de um AVD com o protocolo UDP comparado com a utilização do protocolo TCP.

Segundo Eduardo (2001), para se resolver o problema do método *broadcast* em relação ao seu confinamento a redes locais, pode-se optar pelo método *multicast*. Neste tem-se o mesmo principio do *broadcast*, mas ao invés de todos receberem a mensagem enviada, a mesma é endereçada para um determinado grupo e somente aquele grupo passa a recebe-la.

Para maiores informações e detalhamentos sobre este métodos e os protocolos mais utilizados pode-se consultar em: Eduardo (2001), Gossweiler (1994), Macedonia (1995), Macedonia (1997) e Stytz (1996).

Na seção seguinte será apresentada a técnica, que junto com o método de *broadcast*, é essencial para o bom funcionamento do AVD, principalmente para novos usuários que precisam receber mensagens de atualizações para montar a representação do seu ambiente virtual, inclusive de objetos de baixa freqüência de atualizações.

2.3 *HEARTBEATS*

Heatbeats nada mais é do que o envio de mensagens periódicas com o estado do usuário para os demais usuários. Esse método é utilizado para informar novos usuários sobre

o seu atual estado e também para compensar a perda de algum pacote de estado que foi enviado por algum usuário e que por algum motivo não tenha sido recebida por outro, deixando desta forma o ambiente virtual inconsistente. Isto se faz necessário pois podem existir situações que usuários não interajam com o sistema, fazendo desta maneira que fiquem um determinado tempo sem que enviem mensagens de atualizações de estado. Então para que os novos usuários possam detectar e posicionar os usuários já existentes em seu mundo virtual eles recebem periodicamente as mensagens dos mesmos.

Por exemplo, pode-se citar um ambiente virtual de simulação de guerra, onde se pode ter um veículo ou algum soldado que está em algum ponto estratégico somente fazendo guarda. Neste momento o usuário somente fica observando o ambiente virtual. Supondo que neste instante entre um segundo usuário, como o veículo ou o soldado não está interagindo como o ambiente, nenhuma mensagem de estado do mesmo é enviada aos demais usuários, e o novo usuário não ficará sabendo de sua existência até ele interagir com o ambiente virtual.

Agora, se algum algoritmo de *heatbeat* estiver sendo executado com um intervalo de tempo determinado em cinco e cinco segundos por exemplo, este problema não ocorrerá, pois o novo usuário iria ficar no máximo cinco segundos, se não houvesse nenhum problema na transmissão da mensagem, sem saber da existência do veículo ou do soldado em guarda.

Este método também é interessante para consistir se um usuário está ativo ou se, por algum motivo qualquer, ele tenha saído do ambiente virtual sem um aviso prévio. Pois se os usuários pararem de receber mensagens de estado com informações de um determinado usuário, isto significa que o mesmo não está mais presente e que a representação do mesmo não é mais verdadeira e já pode ser retirada do ambiente virtual.

Segundo Macedonia (1995), a utilização deste método não representa uma das melhores soluções, visto que a obrigatoriedade de cada objeto do ambiente virtual em enviar uma mensagem de atualização, num determinado intervalo de tempo, representa um consumo significativo na largura de banda. Outras soluções também são apontadas, principalmente para resolver o problema da atualização para os novos usuários que entram no ambiente virtual.

Uma outra maneira de se solucionar este problema, sem que fosse pela utilização dessa técnica, seria, segundo Eduardo (2001), determinar um computador central o qual ficaria responsável por manter uma representação completa do ambiente virtual. Este computador seria uma fonte de pesquisa para cada usuário que entra no ambiente virtual, fornecendo a ele todas as informações dos usuários já existentes no AVD. Dessa forma, o novo usuário teria de maneira segura e rápida toda a representação do ambiente virtual em sua estação, mas em

contrapartida, esta forma estaria centralizando uma informação num computador, que, se por algum motivo deixasse de funcionar, deixaria o ambiente virtual inconsistente para novos usuários.

Como forma de se resolver este problema, ter-se-ia como solução determinar como o computador encarregado por realizar esta tarefa o computador que há mais tempo está no ambiente virtual. Dessa maneira, se o mesmo parasse de funcionar, a tarefa de atualizar novos usuários seria transferida para o usuário mais antigo do AVD.

2.4 DISTRIBUTED INTERACTION SIMULATION (DIS)

Em 1983 a *Advanced Research Projects Agency* (ARPA) financiou o programa *Simulation Network* (SIMNET) para criar uma nova tecnologia para expandir a tarefa de realizar simulações interconectadas. SIMNET foi um sucesso, produzindo até 300 simulações interconectadas com a tecnologia que foi desenvolvida dentro do DIS. No passado, DIS e SIMNET não tinham seus conceitos bem definidos, mas as ambições do DIS eram consideravelmente maiores do que as do SIMNET. Os dispositivos do SIMNET usavam a mesma tecnologia enquando o DIS utilizava diversas tecnologias (Hardt, 1998).

Segundo Gossweiler (1994), o surgimento do protocolo DIS está fortemente ligada às experiências obtidas no desenvolvimento do projeto SIMNET. Segundo Macedonia (1995) o projeto SIMNET e DIS são os projetos mais antigos que objetivaram a construção de ambientes virtuais caracterizados pelo fato de suportarem vários usuários ao mesmo tempo. Portanto, a importância deste projeto juntamente com o protocolo DIS é extremamente fundamental para a fundamentação cientifica para o desenvolvimento de ambientes virtuais distribuídos. A seguir segue um breve resumo do projeto SIMNET, conforme Macedonia (1995) e Eduardo (2001).

O *Simulator Network* (SIMNET) é um sistema de treinamento militar distribuído originalmente desenvolvido pela Agência de Projetos de Pesquisa Avançados (ARPA) e pelo exército dos Estados Unidos. Seu principal objetivo é fornecer um ambiente virtual para simulação e treinamento de combates que ajudassem no ensino e prática de técnicas de organização e batalha em conjunto (Mecedonia, 1995).

Segundo Eduardo (2001), o projeto SIMNET foi desenvolvido a partir cinco princípios básicos:

a) no sistema não existe um computador central que fica responsável por notificar a ocorrência de eventos no ambiente virtual;
b) cada computador que participa da simulação é autônomo, mantendo sua própria representação do ambiente virtual, bem como o estado de todos os objetos que o habitam;
c) cada novo usuário que começa a fazer parte do ambiente virtual entra com seus próprios recursos de processamento;
d) cada usuário comunica somente as alterações de estado que ocorrem nos objetos;
e) algoritmos de *Dead Reckoning*[1] são utilizados para reduzir o tráfego de mensagens de atualização.

Estes princípios foram utilizados com o intuito de solucionar, ou ao menos amenizar, os vários problemas relacionados ao AVD´s nos quais foram abordados em seções anteriores.

O protocolo de simulação utilizado no SIMNET é uma das características mais importantes da origem do DIS. Esse protocolo consiste de um determinado número de *Protocol Data Units* (PDU's) que são os responsáveis por informar os estados dos objetos do ambiente virtual ou da ocorrência de eventos ocorridos no mesmo. No SIMNET, por exemplo, há um tipo de PDU responsável por comunicar a aparência de um veículo, estando contido nesse PDU informações como a localização do veículo, orientação, situação, etc. Outros exemplos de PDU's que informam acerca de eventos incluem os de disparo, colisão, explosão e outros associados com eventos comuns a um campo de batalha (Eduardo, 2001). A tabela 2.1 mostra a estrutura típica de um PDU dessa natureza.

[1] *Dead Reckoning* é uma técnica que visa em reduzir o tráfico de pacotes na rede através de algoritmos (Eduardo, 2001).

TABELA 2.1 - EXEMPLO DA ESTRUTURA DE UM PDU

Tamanho Campo (Bytes)	Campos do PDU de aparência do veículo	
6	ID do veículo	*Site*, *host*, veículo
1	Classe do Veículo	Tanque, simples, parado, irrelevante
1	ID da força	
8	Disfarces	Tipo objeto – diferençável de outros
24	Coordenadas no ambiente	Localização: x, y, z
36	Matriz de rotação	
4	Aparência	
12	Marcações	Campo de texto
4	*Timestamp*	
32	Permissões	
2	Velocidade da máquina	
2		Bits indicadores de parada
24	Variáveis de aparência do veículo	Vetor de velocidade, elevação da arma

Fonte: Adaptado de Macedonia (1995).

Para Eduardo (2001), a forma como os dados contidos nesses PDU's estavam estruturados era algo condizente com as necessidades de um ambiente de simulação do exército dos Estados Unidos e, apesar de ter se tornado um padrão de fato, nada impedia que outros implementassem PDU's divergentes desses padrões.

Portanto, esta divergência que poderia existir entre as implementações do protocolo DIS, é que levaram a padronização do protocolo DIS.

Segundo Macedonia (1995), o protocolo DIS é um grupo de padrões, definido pelo Departamento de Defesa dos Estados Unidos e indústrias, preocupados em determinar uma arquitetura de comunicação comum, definindo "o formato e o conteúdo dos dados comunicados; informações a respeito dos objetos do mundo virtual e sua interação; gerenciamento da simulação; medidas de performance; comunicações de rádio; segurança; fidelidade; controle de exercícios, etc".

Pelo fato do DIS ter originado do projeto SIMNET, é certo que herdou características do mesmo. Uma dessas características que se pode citar é o reaproveitamento dos PDU´s. A utilização de PDU´s já tinha sido consagrada pelo projeto SIMNET é era algo que não poderia ser esquecido no processo de padronização do protocolo DIS. No padrão definido na IEEE 1278-1993 (IEEE, 1993) existem vinte e sete tipos de PDU´s diferentes. Entre os principais

PDU´s se destaca o PDU da classe *Entity State* PDU (ESPDU). Este PDU tem como função levar a informação do estado atual de um objeto para os demais usuários do ambiente virtual. Entre estas informações destacam-se informações como sua posição, orientação, velocidade e aparência. Cada tipo de PDU possui informações em comuns com outros PDU´s e informações que são específicas a seu objetivo, gerando desta forma, um conjunto de classes nas quais um PDU pode herdar características de outro PDU mais genérico.

A tabela 2.2 apresenta a estrutura de um ESPDU.

TABELA 2.2 - EXEMPLO DA ESTRUTURA DE UM PDU DE ESTADO DE OBJETO

Tamanho Campo (bytes)	**Campos do PDU de Estado do Objeto**	
12	Cabeçalho do PDU	Versão protocolo, ID do exercício, tipo do PDU, *timestamp*, tamanho em bytes
6	ID do objeto	*Site*, aplicação, objeto
1	ID da força	
1	Número de parâmetro de articulação	
8	Tipo objeto	Tipo objeto, domínio, país, categoria, subcategoria, dados específicos, extras
12	Tipo objeto alternativo	Mesmo tipo da informação acima
12	Velocidade linear	X, Y e Z (componentes de 32 bits)
24	Localização	X, Y e Z (componentes de 64 bits)
12	Orientação	Psi, Theta, Phi (componentes de 32 bits)
4	Aparência	
40	Parâmetros de *Dead Reckoning*	Algoritmo, outros parâmetros, aceleração linear do objeto, velocidade angular do objeto
12	Marcações do objeto	
4	Permissões	32 Campos booleanos
N * 16	Parâmetros de articulação	Mudança, ID, tipo de parâmetro, valor do parâmetro

Fonte: Adaptado de Macedonia (1995).

O protocolo DIS foi padronizado com o número IEEE 1278-1993 no ano de 1993. Esta versão já passou por revisões diversas resultando no padrão IEEE 1278.1-1995 (IEEE, 1995a) e também em padrões adicionais da mesma família: IEEE 1278.2-1995 (IEEE, 1995b), IEEE 1278.3-1996 (IEEE, 1996), IEEE 1278.4-1997 (IEEE, 1997) e IEEE 1278.1a-1998 (IEEE, 1998).

2.5 DIS-JAVA-VRML

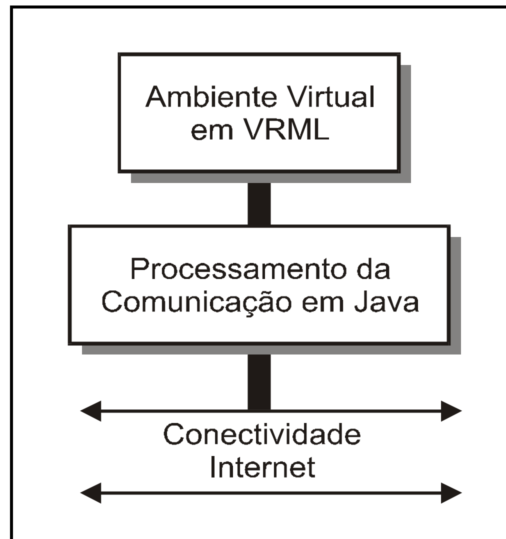
Segundo Web3d (2001), DIS-JAVA-VRML é uma API de código fonte aberto e gratuita sob os termos da *General Public License* (GNU), escrita em JAVA e integere com DIS e VRML. Foi desenvolvida em 1997 pelo grupo de trabalho DIS-JAVA-VRML, que acabaram resolvendo nomear a API com o mesmo nome do grupo.

Um dos primeiros ambientes virtuais distribuídos foi o NPSNET. Ele foi o primeiro a utilizar multicast DIS, foi escrito em C++ e usou como interface gráfica a API da *Silicon Graphics incorporation* (SGI). Geralmente estes ambientes virtuais utilizavam recusos muitos caros e precisam de soluções próprias de software, esse fator representava uma certa barreira que dificultava o crescimento na área de desenvolvimento de ambientes virtuais distribuídos.

Porém, segundo Eduardo (2001), com a evolução dos computadores o processamento foi ficando cada vez mais rápido e o custo dos computadores passou-se a ser cada vez menor. Isto então foi motivo para a utilização de softwares mais sofisticados, deixando de se preocupar então com o custo de processamento dos softwares. Com esta visão, duas novas linguagens prosperavam entre outras, criando novas expectativas para a criação de ambientes virtual. Estas linguagens eram o Java e o VRML. Juntas, essas duas linguagens criaram uma situação ideal para a construção de ambientes virtuais portáveis, capazes de serem executados tanto em equipamentos com alto poder de processamento, quanto em equipamentos com poder de processamento mais modesto. Esta visão portanto foi a maior motivação do grupo Web3D para a criação da API DIS-Java-VRML com estas duas linguagens.

A API do DIS-Java-VRML agrega classes que implementam o protocolo DIS, descrito na seção anterior, bem como classes que controlam a comunicação com a rede e com os cenários feitos em VRML. O VRML por sua vez é utilizado para a modelagem e renderização do ambiente virtual. A fig. 2.5 ilustra a organização geral do projeto DIS-Java-VRML.

FIGURA 2.5 - ORGANIZAÇÃO GERAL DO PROJETO DIS-JAVA-VRML



Fonte: Eduardo (2001)

DIS-Java-VRML foi desenvolvido em duas camadas separadas, a *virtual reality transfer protocol* (vrtp) que é responsável pelas transmissões dos PDU´s e a camada que fornece a interface gráfica, a VRML. O Java foi utilizado como a linguagem de implementação do processamento da comunicação do ambiente virtual distribuído e também ficou responsável pela geração da montagem de *scripts* para a interface VRML.

3 OPENGL

Segundo Jacobs (1999), OpenGL, ou *Open Graphics Library*, é uma biblioteca padronizada e comercial para comandos gráficos, que possibilita aos desenvolvedores produzirem animações 3D com simples linhas de comando. Já segundo Woo (1999), OpenGL é "uma interface de *software* para *hardware* gráfico". Essencialmente, é uma biblioteca gráfica e de modelagem 3D extremamente portável e muito rápida. Foi desenvolvida pela Silicon Graphics, Incorporate (SGI), um reconhecido líder mundial em computação gráfica e animação.

Ela consiste de aproximadamente 150 comandos usados para definir objetos e operações para se criar aplicações em 3D. OpenGL, que significa *Open Graphics Library* ou Biblioteca Gráfica Aberta, foi desenhada para se manter independente da plataforma e do sistema operacional onde esta rodando a aplicação e por isso não tem comandos para o gerenciamento de janelas e entrada de dados. Ela também não tem comandos em alto nível para descrever objetos. Dessa maneira, tem-se que construir os objetos a partir de entidades primitivas como pontos, linhas e polígonos. Para resolver isso, existe a *OpenGL Utility Library* (GLU), ou Biblioteca de Utilidades da OpenGL, que provê várias funções como superfícies quadráticas, *non-uniform rational B-spline* (NURBS) e outras. A GLU existe em todas as implementações da OpenGL, segundo Molofee (2002).

OpenGL é uma biblioteca 3D essencial, que aproveita toda a capacidade de aceleração do hardware para impulsionar jogos dentro da realidade e qualidade profissional de renderização, mapeamento de texturas e efeitos especiais. Quase todos os principais jogos, como por exemplo Quake III, Half-Life, MDK2, Baldurs Gate, Decent 3 e Madden NFL 2001, requerem OpenGL para aceleração do hardware (Silicon, 2001).

Segundo Wright (2000), uma implementação de OpenGL pode ser uma biblioteca de *software* que cria imagens tridimensionais em resposta a chamadas de funções OpenGL ou um controlador de dispositivo (normalmente uma placa de vídeo) que faz o mesmo. Ou seja, existem dois tipos de implementação de OpenGL, uma baseada em *hardware*, outra em *software*. A implementação OpenGL que acompanha os sistemas operacionais Windows NT 3.1 em diante, o Windows 95 em diante e o Windows 2000 é uma típica implementação de *software*. Quando se instala uma placa de vídeo aceleradora 3D que implemente no seu controlador (*driver*) a biblioteca OpenGL, esta é uma implementação dita em *hardware*.

As funções da OpenGL, agora chamadas de comandos, são desenvolvidas para criar gráficos em 2D e 3D sendo com mais ênfase em gráficos 3D. O programa é completamente funcional, incluindo basicamente tudo o que o usuário precisa para gráficos 3D. Ela inclui modelagem 3D, transformações, cores, iluminação, degrade, mapa de textura, curvas *non-uniform rational B-spline* (NURBS) e superfícies, efeito de neblina, união e outras (Silicon, 2001).

A OpenGL não é a única biblioteca gráfica no mercado, existem outras como a DirectX. Segundo Microsoft (2002), o DirectX é um conjunto de bibliotecas de baixo nível para a criação de jogos e outras aplicações multimídia. Inclui suporte para gráficos 2D e 3D, efeitos sonoros e música, dispositivos de entrada e suporte para aplicações em rede como jogos para vários jogadores simultâneos. Pode-se ver que umas das vantagens do DirectX é o tratamento mais especifico de dispositivos de entradas e saídas, como o mouse, teclado, placa de som e outros que possam a se utilizados pelo usuário. Já a biblioteca OpenGL não dá suporte a estes, mas em compensação, a biblioteca OpenGL foi desenvolvida para obter-se independência de software e hardware, o que já não acontece com a DirectX, pois ela somente foi desenvolvida para plataforma Windows. Na seção seguinte será tratada justamente as vantagens e as desvantagens da OpenGL.

3.1 VANTAGENS E DESVANTAGENS

A biblioteca gráfica OpenGL traz uma série de vantagens e desvantagens que devem ser analisadas antes de qualquer decisão da utilização da mesma para o desenvolvimento de projetos que utilizem interface gráfica.

Abaixo pode-se citar algumas das principais vantagens e desvantagens segundo Silicon (2001):

a) padrão de indústria: um consórcio independente, a comissão de diretores de pesquisa da arquitetura OpenGL, guia a especificação do OpenGL. Com amplo apoio da indústria, OpenGL é o único padrão gráfico multiplataforma verdadeiramente aberto, independente de fornecedores;

b) estável: implementações com OpenGL têm estado disponíveis desde 1992 em uma grande variedade de plataformas. Inclusões em sua especificação são bem controladas, e atualizações propostas são anunciadas a tempo para que os desenvolvedores adotem as mudanças. Exigências de compatibilidade asseguram as aplicações existentes a não ficarem obsoletas;

c) fidedigno e portável: todas as aplicações OpenGL produzem exibições visuais consistentes em qualquer implementação, apesar do sistema operacional ou da compatibilidade de hardware;

d) desenvolvendo-se: por causa de seu completo design e sua visão do futuro, OpenGL permite que novas inovações de hardware sejam acessíveis através da API pelo mecanismo de extensão do OpenGL. Deste modo, inovações aparecem na API na hora certa e deixam os desenvolvedores de aplicações e os fornecedores de hardware incorporar novas características nos seus ciclos normais de lançamento dos produtos;

e) escalabilidade: aplicações baseadas na OpenGL podem ser executadas em sistemas que variam de produtos eletrônicos a PC´s, estações de trabalho, e supercomputadores. Como resultado, aplicações podem escalar para qualquer classe de máquina que o desenvolvedor deseja atingir;

f) fácil de usar: OpenGL é bem estruturado, com um desenho intuitivo e comandos lógicos. Rotinas eficientes de OpenGL tipicamente resultam em aplicações com menos linhas de código que as aplicações geradas utilizando outros pacotes ou bibliotecas gráficas. Além disso, os *drivers* OpenGL encapsulam informação sobre o hardware subjacente, livrando o desenvolvedor da aplicação de ter que projetá-la para características de hardware específicas;

g) bem documentado: foram publicados numerosos livros sobre OpenGL, e uma grande amostra de código está prontamente disponível, fazendo as informações sobre OpenGL baratas e fáceis de se obter.

3.2 FUNCIONAMENTO

OpenGL é uma biblioteca gráfica orientada a procedimento. Ao invés de descrever uma cena e como ela deve parecer-se, o programador na verdade descreve os passos necessários para alcançar certa aparência ou efeito. Estes "passos" envolvem chamadas à comandos da biblioteca OpenGL. Estes comandos são usados para desenhar primitivas gráficas como pontos, linhas e polígonos em três dimensões. Além disso, através das bibliotecas de utilidades como a GLU, OpenGL pode também suportar iluminação e sombreamento, mapeamento de texturas, *blending* (combinação), transparência, animação e muitos outros efeitos especiais e capacidades. O sombreamento citado acima não se refere a

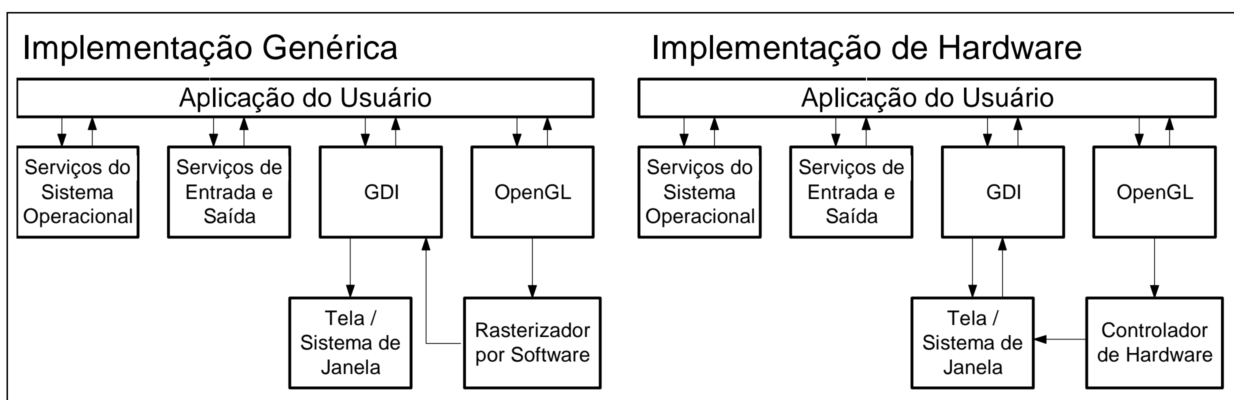
sombras projetas por um objeto em outro, e sim se refere ao fato de um objeto ser mais iluminado no lado próximo a uma fonte de luz, enquanto o outro lado é mais escuro. Embora possível de implementação de diversas formas, não existe uma função OpenGL nativa para sombras, nem recursos para as luzes projetarem sombras automaticamente.

As cenas são construídas pelo programador para satisfazer suas próprias necessidades de alto nível programando-as utilizando comandos OpenGL de baixo nível. O mesmo ocorre com texturas. OpenGL não tem nenhum formato de textura nativo. Entretanto, é extremamente simples escrever um módulo em Object Pascal que leia um arquivo com formato *bitmap*, por exemplo.

Segundo Wright (2000), as implementações OpenGL podem ser dividas em implementações genéricas e implementações de Hardware. Uma implementação genérica é uma implementação das funções OpenGL por software. Tecnicamente, uma implementação genérica pode funcionar em qualquer lugar onde o sistema tiver capacidade de mostrar a imagem gerada. Já uma implementação OpenGL em hardware normalmente é um controlador para uma placa de vídeo aceleradora 3D.

Pode-se diferenciar as duas implementações pela forma de retornarem os gráficos. Aplicações Windows, por exemplo, que desejam criar uma saída na tela normalmente chamam uma biblioteca do Windows chamada *Graphics Device Interface* (GDI). A GDI contém funções para escrever texto numa janela e desenhar linhas 2D. Então quando a implementação é genérica, a camada da OpenGL faz o processamento e envia os gráficos processados para a GDI. Já na implementação por hardware a saída é feita diretamente para o controlador de hardware. Na fig. 3.1 tem-se exemplificado o relacionamento das implementações genéricas e de hardware com a aplicação do usuário.

FIGURA 3.1 – IMPLEMENTAÇÃO GENÉRICA E IMPLEMENTAÇÃO DE HARDWARE



Fonte: Adaptado de Wright (2000)

Agora que se tem uma visão geral do funcionamento da OpenGL, nas seções seguintes é apresentado como é o funcionamento especificamente no Windows e no ambiente de programação Borland Delphi.

3.2.1 FUNCIONAMENTO DA OPENGL COM O WINDOWS

Segundo Jacobs (1999), OpenGL está somente disponível para versões 32-bit do Windows, como o NT e versões do 95 ou superiores. No Windows a OpenGL é implementada nas seguintes *Dinamic-Link Library* (DLL):

a) OpenGL32.dll – Implementação das principais funções ou funções primitivas da OpenGL;

b) Glu32.dll – Implementações de algumas funções utilitárias aos desenvolvedores de software.

Estas duas DLL´s contém um conjunto de rotinas na qual podem produzir uma cena gráfica e animações com alto grau de realismo. Nas versões do Windows NT, Windows98 e OEM do Windows 95 estas DLL´s já fazem parte do pacote, mas para versões mais antigas do Windows 95 é necessário fazer o *download* das mesmas pela internet.

3.2.2 FUNCIONAMENTO DA OPENGL COM O AMBIENTE DE PROGRAMAÇÃO BORLAND DELPHI

Para usar a OpenGL juntamente com o Borland Delphi é necessário que alguns pré-requisitos estejam de acordo, caso contrário a utilização não será possível. Um dos requisitos é possuir uma versão 32 bits do Delphi. Está versão pode ser qualquer uma a partir da versão 2 do Delphi. Outro requisito é as *units* de interface. Estas *units* são utilizadas para fazer as declarações de interfaceamento das rotinas das DLL´s do OpenGL. Ela deve ser declarada na clausura *uses* da *unit* que irá fazer as chamadas a comandos da OpenGL. No quadro 3.1 tem-se o exemplo de como funciona a importação de funções de DLL´s para uma unit do Delphi.

QUADRO 3.1 – CÓDIGO FONTE EXEMPLIFICANDO A UNIT DE INTERFACE

```
Unit OpenGL;

Uses
  Windows;

Interface
Const
  ...

  procedure glViewport (x,y: GLint; width, height: GLsizei); stdcall;

Implementation
  procedure glViewport; external ‘opengl32.dll’;
End.
```

Note que após a declaração do comando existe a chamada a diretiva *stdcall*. A *stdcall* faz parte de um conjunto de diretivas que definem a maneira de como serão passados os parâmetros da rotina. No caso da *stdcall*, os parâmetros serão passados da direita para esquerda, a rotina mesmo se encarrega de limpar a pilha e não serão utilizados os registradores para armazenar os parâmetros.

Um outro ponto importante que se torna um requisito para o desenvolvimento de aplicações com OpenGL é o conhecimento de desenvolvimento de aplicações com o Delphi.

3.3 COMANDOS BÁSICOS

Pode-se fazer a citação de alguns comandos que são básicos para a utilização e a criação de cenas na OpenGL:

a) viewport: usado para fazer uma conexão com as coordenadas do sistema (Windows) com as coordenadas usadas pela OpenGL;

b) projeção ortográfica: define qual é a região no espaço na qual os objetos poderão ser visualizados.

No quadro 3.2 pode-se ver a interface destes dois comandos.

QUADRO 3.2 – COMANDOS BÁSICOS

```
procedure glViewport (x,y: GLint; width, height: GLsizei); stdcall;
procedure glOrtho (left, right, bottom, top, zNear, zFar: GLdouble);
stdcall;
```

3.4 DESENHANDO-SE COM A OPENGL

Os comandos de desenho da OpenGL ou de inserção de vértices devem sempre estar entre os comandos *glBegin* e *glEnd*:

a) *glBegin*: o parâmetro define a forma do desenho. Como por exemplo: linha, quadrado e um ponto;

b) *glEnd*: finaliza o comando *glBegin*.

Entre estes dois comandos existe a lista de vértices e os possíveis comandos relacionados com os vértices. O comando utilizado para inserir vértices é o *glVertex* e tem algumas derivações com alteração no parâmetro passado para o mesmo. No quadro 3.3 pode-se ver a interface dos comandos descritos acima.

QUADRO 3.3 – COMANDOS DE DESENHO

```
procedure glBegin (mode:GLenum); stdcall;
procedure glEnd; stdcall;
procedure glVertex2f (x,y: GLfloat); stdcall;
procedure glVertex3f (x,y,z: GLfloat); stdcall;
procedure glVertex4f (x,y,z,w: GLfloat); stdcall;
```

3.5 OPENGL COMO UMA MÁQUINA DE ESTADO

OpenGL é uma máquina de estado. É possível informar vários estados ou modos a atributos, que estes permaneceram com seu estado até que algum comando os mude. Por exemplo, pode-se selecionar a cor corrente para branco, vermelho ou qualquer outra cor, e depois disso, todos objetos desenhados serão apresentados com a cor atual até que a mesma seja alterada.

Pode-se habilitar e desabilitar diversas variáveis de estado através dos comandos *glEnable* ou *glDisable*. Além destes comandos existem também outros três comandos que auxiliam o programador. Um deles é *glIsEnabled*, que tem a função de testar se uma determinada variável está ativada ou não. E os outros dois comandos, *glPushAttrib* e *glPopAttrib*, têm a função de salvar os estados da variável e depois restaurar, isto no conceito de uma pilha. No quadro 3.4 pode-se ver a interface dos comandos descritos acima.

QUADRO 3.4 – COMANDOS DE ESTADO

```
procedure glEnable (cap: Glenum); stdcall;
procedure glDisable (cap: GLenum); stdcall;
function  glIsEnabled (cap: GLenum): GLBoolean; stdcall;
procedure glPushAttrib(mask: GLbitfield); stdcall;
procedure glPopAttrib; stdcall;
```

3.6 PERSPECTIVA

Projeção ortográfica ou paralela pode ser útil para diversos propósitos, mas quando objetivo é realismo em gráficos, a projeção em perspectiva é mais apropriada.

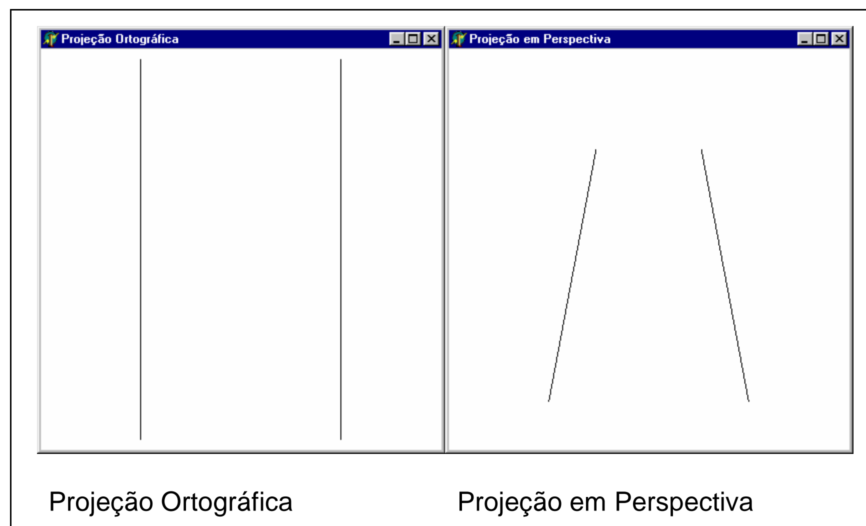
Para configurar a projeção em perspectiva em OpenGL, troca-se o comando de projeção ortográfica, o *glOrtho*, pelo comando *glFrustum*, que criará uma limitação no espaço onde os objetos poderão ser desenhados e observados.

No quadro 3.5 segue as coordenadas de dois segmentos de retas paralelos tomados como exemplo. A fig. 3.2 mostra a diferença de uma cena desenhada com os dois segmentos de reta com projeção ortográfica e outra com a projeção em perspectiva.

QUADRO 3.5 – COORDENADAS DE DOIS SEGMENTOS DE RETA PARALELOS

```
glBegin(GL_LINES);
  glVertex3f(-0.5,-0.95,1.0);
  glVertex3f(-0.5,0.95,-1.0);
  glVertex3f(0.5,-0.95,1.0);
  glVertex3f(0.5,0.95,-1.0);
glEnd;
```

FIGURA 3.2 – PROJEÇÃO ORTOGRÁFIA E PROJEÇÃO EM PERSPECTIVA



Existe um comando utilitário que faz parte da biblioteca de utilitários (*Glu32.dll*) que é parecido com o *glFrustum*, mas que especifica-se de uma maneira diferente. Este comando é o *gluPerspective*. Este comando é limitado a criar espaços que são simétricos em ambos os eixos (X e Y). O Quadro 3.6 mostra a sua interface e comenta sobre seus parâmetros.

QUADRO 3.6 – COMANDO UTILITÁRIO PARA DEFINIR A PERSPECTIVA DA CENA

```
procedure gluPerspective(fovy, aspect, zNear, zFar: GLdouble); stdcall;
```

Onde:

Fovy: é o ângulo do campo de visão sobre o plano Z.

Aspect: é o *aspect ratio* do campo de visão.

Near e Far: são as distancias entre o ponto de visão (câmara) e o plano de fundo.

3.7 TESTE DE PROFUNDIDADE

O teste de profundidade tem o objetivo de desenhar os outros objetos que estão à frente de outros objetos, desconsiderando a ordem na qual foram informados os seus vértices.

Dessa forma, toda vez que um *pixel* for desenhado na tela, será feito um teste com um *buffer* de profundidade calculado previamente. Este *buffer* vai variar de acordo com o ângulo de visão da cena. Para verificar se o teste de profundidade está ativou ou não e para configurar alguns parâmetros do teste existem varias funções. Estas funções podem ser selecionadas utilizando-se do comando *glDelphFunc* e como parâmetro deste comando deve-se passar a função que se deseja utilizar para fazer o teste. Para maiores informações consultar Wright (2000).

O teste de profundidade pode ser habilitado e desabilitado chamando-se os comandos *glEnable* e *glDisable* passando como parâmetro a constante *GL_DEPTH_TEST*.

3.8 TRANSFORMAÇÕES

As transformações possibilitam que coordenadas 3D sejam projetadas para dentro do monitor, que no caso é um periférico com coordenadas 2D. Também permitem girar objetos ao redor, movê-los e até mesmo fazer com que estiquem, encolham, e deformem.

Quando se especifica um vértice para ser visualizado na tela, três transformações ocorrem entre a especificação e a sua visualização. As transformações são de visualização, modelagem e de projeção. Na tabela 3.1 pode-se ver um resumo das transformações:

TABELA 3.1 – RESUMO DAS TRANSFORMAÇÕES DA OPENGL

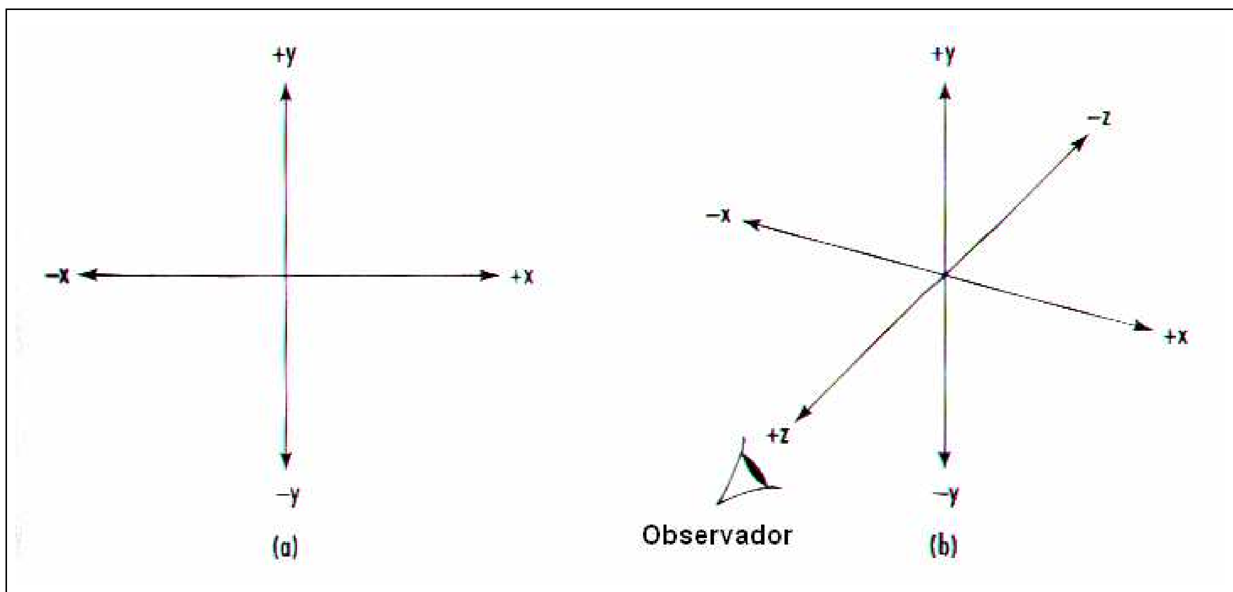
Transformação	Descrição
Viewing	Especifica o local do ponto de visão ou da câmera
Modeling	Move os objetos ao redor da cena
Modelview	Descreve a dualidade das transformações viewing e modeling
Projection	Recorda e redimensiona o volume que esta sendo visto
Viewport	Aplica a escala da saída final para a janela

3.8.1 *EYE COORDINATES*

Segundo Wright (2000), *Eye Coordinates* é o ponto de vista do observador, indiferente de qualquer transformação que pode ocorrer, pode-se dizer que é a coordenada absoluta da tela. Dessa forma, *eye coordinates* não é a coordenada real, mas representa um sistema de coordenada virtual fixada que é usado com uma armação de referencia.

Na fig. 3.3 pode-se ver o sistema de *eye coordinates* sobre dois pontos de vista. O primeiro é perpendicular ao monitor e o segundo foi rotacionado para uma melhor visualização do eixo Z:

FIGURA 3.3 – SISTEMA EYE COORDINATES

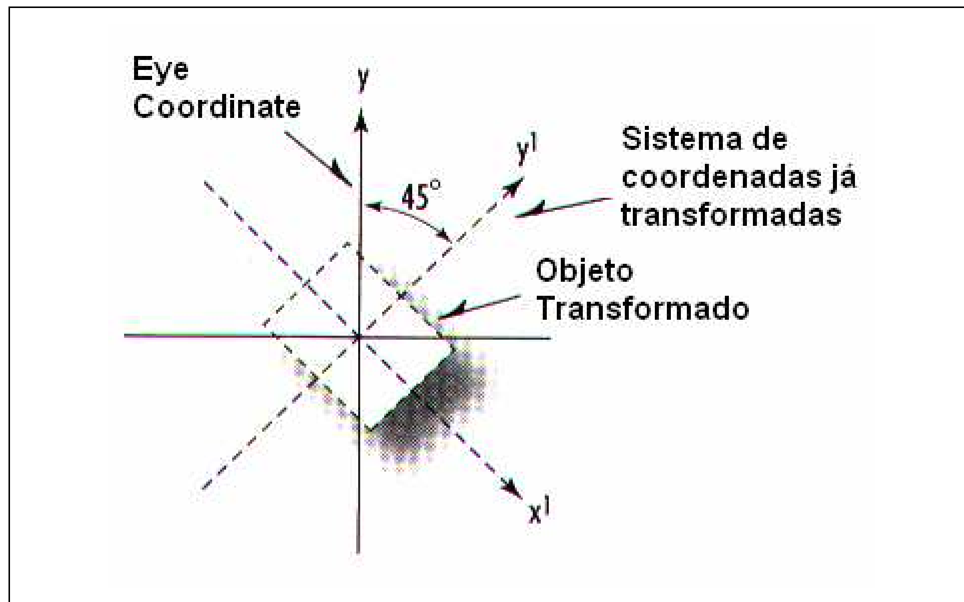


Fonte: Adaptado de Wright (2000).

Quando se desenha em 3D com OpenGL, usa-se o plano cartesiano como sistema de coordenadas. Sobre a ausência de qualquer transformação, o sistema usado é idêntico ao sistema *eye coordinates*. Todas as transformações mudam o sistema de coordenada corrente com respeito a sistema *eye coordinates*.

A fig. 3.4 mostra um exemplo com coordenadas 2D com o sistema de coordenadas rotacionado com quarenta e cinco graus no sentido horário baseado na *eye coordinates*.

FIGURA 3.4 – SISTEMA DE COORDENADAS ROTACIONADOS EM RELAÇÃO A EYE COORDINATES



Fonte: Adaptado de Wright (2000)

3.8.2 *VIEWING TRANSFORMATIONS*

A *Viewing transformations* é a primeira a ser aplicada na cena. Ela é usada para determinar o ponto de observação da cena. Por padrão, o ponto de observação de origem é (0,0,0), olhando-se sobre o eixo Z no sentido negativo. Este ponto de observação é movido relativamente com o sistema *eye coordinates* para providenciar um especifico ponto de visualização da cena. Quando o ponto de observação está localizado na origem, objetos desenhados com a coordenada Z positiva ficam atrás do observador (Wright, 2000).

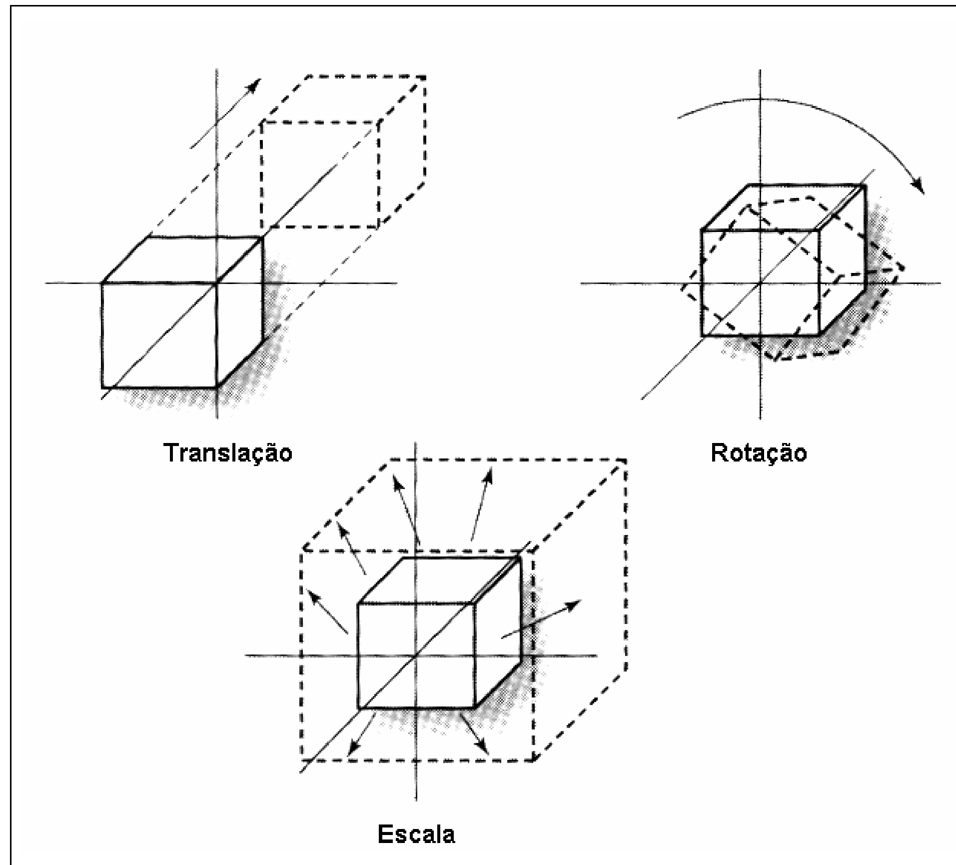
Dessa forma, pode-se dizer que quando determina-se a transformação *viewing*, está se determinando a posição da câmera na cena. Nesta transformação pode ser alteradas a posição e a direção do ponto de vista do usuário.

A transformação *viewing* deve ser realizada antes de qualquer outra transformação. Isto porque ela move o sistema corrente de coordenada que se esta trabalhando em respeito com o sistema *eye coordinates*. Então, todas as transformações realizadas depois desta, ocorrem baseadas sobre o novo sistema de coordenada modificada.

3.8.3 *MODELING TRANSFORMATIONS*

São usadas para manipular seu modelo e os objetos. Esta transformação move para outros lugares, rotaciona e aplica escala a objetos da cena. A fig. 3.5 ilustra estas três transformações que podem ser aplicadas nos objetos.

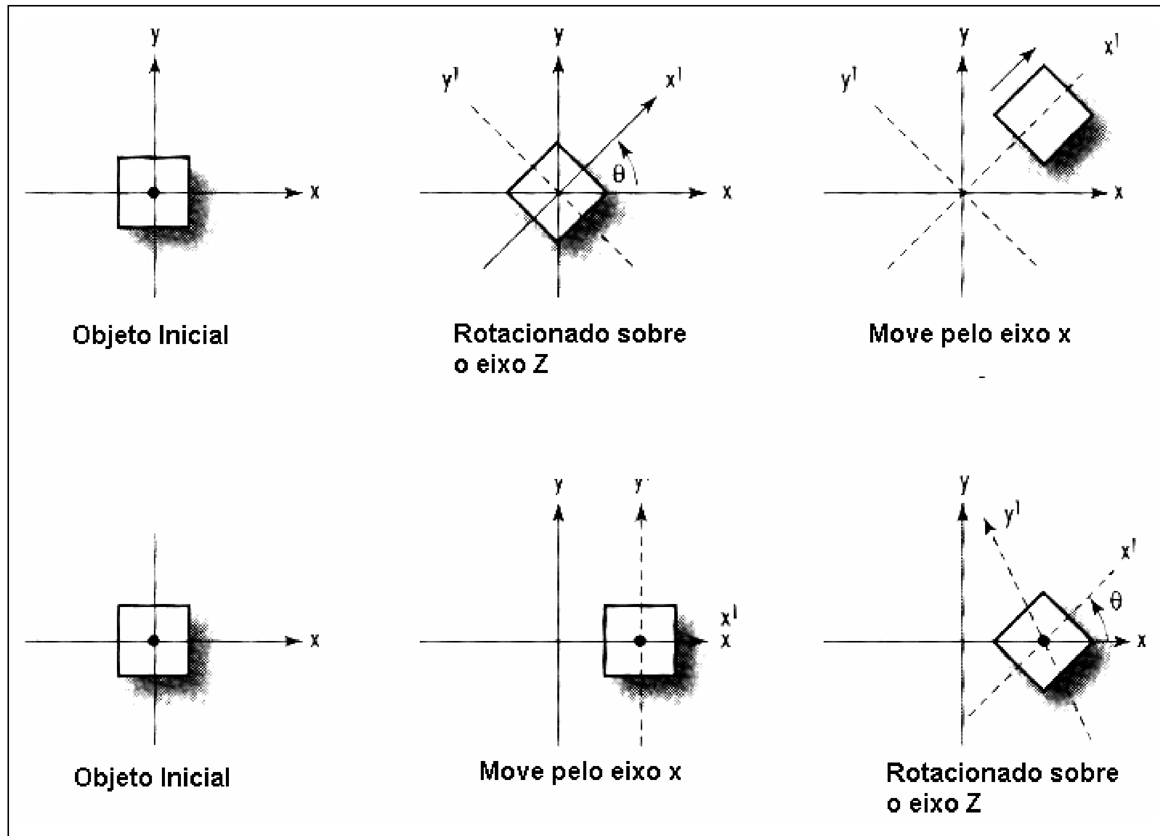
FIGURA 3.5 – TRANSFORMAÇÕES GEOMÉTRICAS



Fonte: Adaptado de Wright (2000)

A aparência final da cena vai depender diretamente da ordem das transformações aplicadas. Isto ocorre particularmente por causa da translação e da rotação. A fig. 3.6 ilustra esta diferença com transformações em ordens diferentes.

FIGURA 3.6 – ORDEM DAS TRANSFORMAÇÕES



Fonte: Adaptado de Wright (2000)

Para este trabalho, as transformações mais relevantes foram estas comentadas acima. Para obter mais informações sobre as transformações *mdelview*, *projection* e *viewport* consultar Wright (2000).

Com isso, pode-se concluir que a OpenGL permite realizar as três transformações básicas da geometria através de comandos, que são: rotação, translação e a escala. No quadro 3.7 pode-se ver as suas declarações.

QUADRO 3.7 – COMANDOS DE TRANSFORMAÇÕES

Comandos de Rotação:

```
Procedure glRotated(angle:GLdouble; x:GLdouble; y:GLdouble; z:GLdouble);
stdcall;
Procedure glRotatef(angle: GLfloat; x: GLfloat; y: GLfloat; z: GLfloat);
stdcall;
```

Comandos de Translação:

```
Procedure glTranslated(x: GLdouble; y: GLdouble; z: GLdouble); stdcall;
Procedure glTranslatef(x: GLfloat; y: GLfloat; z: GLfloat); stdcall;
```

Comandos de Escala:

```
Procedure glScaled(x: GLdouble; y: GLdouble; z: GLdouble); stdcall;
Procedure glScalef(x: GLfloat; y: GLfloat; z: GLfloat); stdcall;
```

3.9 TEXTURAS

Segundo Wright (2000), a técnica de mapeamento de texturas é provavelmente o avanço mais significativo para a computação gráficas nos últimos anos. OpenGL prove uma série de funções para se realizar o mapeamento de imagens de textura para dentro de polígonos pertencentes à cena. Jogos como Quake são exemplos, eles usam o mapeamento de texturas para gerar imagens reais de salas, monstros e outros objetos reais. Toda textura é uma imagem gráfica retangular de algum tipo, que é representada dentro de um vetor de *bytes* que pode ser de uma dimensão, duas dimensões ou até três dimensões.

Uma textura de 1D é uma imagem com largura mas sem altura, ou vice versa. Texturas 1D são simples *pixeis* largos ou altos. Já uma textura 2D é uma imagem que é mais de um *pixel* largo ou alto e é geralmente carregado de um arquivo *bitmap* com formato do Windows. Para este trabalho serão utilizadas as texturas 2D.

Naturalmente deve-se primeiramente definir a imagem de textura antes de aplicar texturas em polígonos. Para definir as imagens que representaram as texturas seguem as mesmas regras de armazenamento que a de um *bitmap*. Para maiores informações sobre estas regras consultar Wright (2000).

Depois de carregar a imagem que irá representar a textura armazenada em memória, pode-se definir a textura nos polígonos. Para isso OpenGL faz uso de do comando *glTexImage2D*. Este comando serve para definir somente imagens 2D, para outras dimensões se tem os comandos *glTextImage1D* e *glTextImage3D*.

Para utilizar texturas são necessários ainda mais dois passos, um deles é habilitar a variável GL_TEXTURE_2D, isso para texturas 2D, com o comando *glEnabled* visto na seção anterior. O outro passo e mapear a mesma nos polígonos no qual conterão a textura. Para isso é utilizado o comando *glTexCoord2f*. Este comando também somente é utilizado para texturas 2D.

No quadro 3.8 pode-se ver a declaração da interface do comando que define uma textura e do comando que realiza o mapeamento da textura no polígono.

QUADRO 3.8 – COMANDOS PARA UTILIZAÇÃO DE TEXTURAS EM POLÍGONOS

```
procedure glTexImage2D(target:GLenum; level, components:GLint; width,
height:GLsizei; border: GLint; format, _type: GLenum; pixels: Pointer);
stdcall;
procedure glTexCoord2f (s,t: GLfloat); stdcall;
```

Existem ainda vários detalhes a serem estudados em texturas, mas, como o objetivo do trabalho não é este, somente foi apresentado o básico para utilizar texturas nos polígonos de uma cena. Para maiores informações consultar Jacobs (1999), Molofee (2002), Wright (2000) e Woo (1999).

3.10 CORES, LUZES E MATERIAIS

A OpenGL adotou padrão RGB para se definir uma cor. Dessa forma especifica-se a intensidade de cada cor do RGB, que no caso são vermelho, verde e azul, e a soma dos mesmos resultará na cor final. Para selecionar a cor em OpenGL se utiliza o comando *glColor<x><t>*, onde o *<x>* representa o número de parâmetros que pode ser 3, para três parâmetros representando o vermelho, o verde e o azul, ou 4 para quatro parâmetros incluído-se o componente *alpha*, que especifica a translucidez da cor. Já o *<t>* especifica o tipo da informação que será passada como parâmetro, podendo ser *b*, *d*, *f*, *i*, *s*, *ub*, *ui* ou *us*, isto respectivamente representa os tipos *byte*, *double*, *float*, *interger*, *short*, *unsigned byte*, *unsigned interger* e o *unsigned short*.

Quanto as cores, ainda existe a opção de se fazer um degrade com as cores, isto se habilitando a opção de *shading*. Para se habilitar esta opção se utiliza o comando *glShadeModel*, passando-se como parâmetro a forma como se quer o degrade.

Agora, no mundo real os objetos não possuem cores sólidas ou em degrades baseadas somente sobre seus valores em RGB. Na verdade a cor deles é formada pela reflexão da luz ambiente ou alguma luz artificial. Dessa forma, pode-se classificar as luzes em três tipos: a luz ambiente, a luz difusa e a luz specular.

Cada uma delas tem um efeito diferente sobre as cores dos objetos. Também é possível aplicar a influencia de todas elas sobre um objeto, não ficando restrito a utilização de somente um tipo.

A luz é somente uma parte da equação. No mundo real, os objetos não têm cores por si próprios. Isso se deve por que cada objeto, ao ser refletido, emite um onda de luz. Por exemplo, uma bola azul, ao ser iluminada por uma luz ela irá refletir mais a cor azul e as outras cores serão absorvidas. Neste caso o azul é o que será visto pelo observador. Geralmente as cenas do mundo real são iluminadas por uma luz branca contendo a mistura de todas as cores. Sobre uma luz branca, os objetos apareceram com suas próprias cores ou cores naturais. Porém, isto nem sempre é assim, imagina-se a mesma bola azul num quarto escuro com somente uma luz amarela. A bola aparecerá preta para o observador porque todas as

luzes amarelas serão absorvidas e o azul não é refletido pela ausência do azul na luz. Para isso é possível definir qual o material do objeto.

O material irá representar qual a luz que ele mais reflete. Para selecionar as propriedades do material corrente utiliza-se o comando *glMaterialfv*. Também é necessário habilitar a utilização de material, com o comando *glEnable* passando-se como parâmetro a variável *GL_COLOR_MATERIAL*, e definir qual a cor que o material irá refletir utilizando-se o comando *glColorMaterial*.

Para maiores informações sobre o assunto consultar Wright (2000).

3.11 O QUE MAIS É POSSÍVEL COM OPENGL

Ainda existem vários outros recursos que a OpenGL fornece mais que não serão relevantes para o objetivo principal deste trabalho. Segue abaixo uma lista de mais alguns recursos que poderão ser encontrados na OpenGL:

a) interações: é possível nomear objetos para sua seleção posteriormente. Isto é importante quando o usuário quer selecionar um objeto com um click do mouse, por exemplo;

b) comandos para criação de triângulos: OpenGL fornece recursos que facilitam a criação de triângulos sem que haja necessidade de informar todos os vértices do mesmo;

c) animações: existem várias técnicas para desenvolver animações com OpenGL, entre elas pode-se citar a animação em tempo-real, onde a animação tem que ter a mesma performance independente do hardware utilizado;

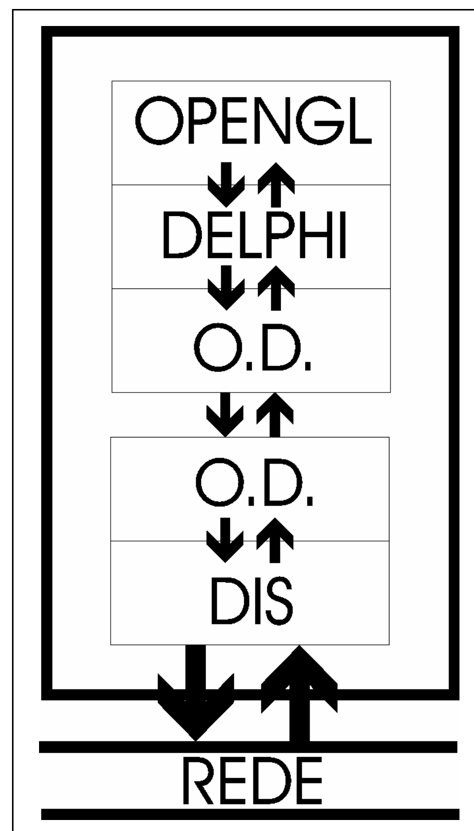
d) efeitos especiais: OpenGL traz uma série de efeitos que podem ser aplicados na cena. Entre elas pode-se citar a transparências em objetos e imagens, o *Anti-aliasing* e névoa;

e) desenho de imagens: OpenGL também permite criar desenhos em 2D, podendo-se criar *pixels*, *bitmaps*, *fonts* e imagens raster.

4 OBJETOS DISTRIBUÍDOS (CORBA)

Visto os tópicos abordados anteriormente, teve-se a idéia de implementar um ambiente virtual distribuído fazendo-se uso da camada vrtp para realizar a transmissão de PDU´s pela rede e substituindo-se a camada de interface VRML, vista na seção 2.5, pela interface OpenGL. Dessa maneira, ter-se-ia o vrtp implementado na linguagem Java e a interface gráfica implementada em Object Pascal juntamente com a OpenGL no ambiente de programação o Delphi. A fig. 4.1 representa o fluxo de dados através de troca de mensagens e as camadas do sistema proposto.

FIGURA 4.1 - TROCA DE MENSAGENS ENTRE CAMADAS DO SISTEMA



Com a utilização de duas linguagens, Object Pascal e Java, passa-se a ter duas aplicações que fazem parte do mesmo sistema. Para que os objetos, que estão distribuídos agora entre as aplicações, possam se comunicar através de mensagem, é necessário que exista alguma tecnologia para criar-se uma ponte entre eles, caso contrário, não seria possível a utilização desta proposta. Por este motivo se tem em cada módulo do sistema uma camada chamada de Objetos Distribuídos (OD).

Segundo Jacobsen (2000), objetos distribuídos são estruturas que executam uma determinada tarefa. Sua principal característica refere-se à sua localização, eles podem ser abrigados todos em uma única máquina ou distribuídos em máquinas distintas de uma rede de computadores LAN, WAN, ou até mesmo na internet. Em conjunto, esses objetos são capazes de executar funções para um sistema distribuído.

Para se entender melhor como os objetos distribuídos funcionam é importante conhecer o conceito de orientação a objetos. Segundo Capeletto (1999), pode-se criar conceitos básicos de programação orientada a objetos através de algumas idéias fundamentais. Abaixo segue as principais idéias que suportam a tecnologia baseada em objetos:

a) classe: é um conjunto de informações que compartilham estruturas comuns e comportamento idêntico. É uma abstração, que representa uma idéia ou noção geral de um conjunto de objetos similares. Uma classe é uma implementação de um tipo de objeto. Ela tem uma estrutura de dados e métodos que especificam as operações que podem ser feitas com aquela estrutura de dados;

b) objeto: é qualquer coisa, real ou abstrata, sobre a qual se armazenam dados e operações que manipulam os dados. Um objeto corresponde a uma concepção, abstração ou coisa que pode ser identificada distintamente. Durante a análise, objetos tem atributos e podem ser envolvidos em relacionamentos com outros objetos. Durante o projeto, a noção de objeto é estendida pela introdução de métodos e atributos de objetos. Na fase de implementação a noção de objeto é determinada pela linguagem de programação. Objeto é uma instância de uma classe;

c) métodos: especificam a maneira como as operações são codificadas no software. São códigos para implementação em uma classe, ou operação interna, ou seja, o processo de desenvolvimento;

d) solicitações: uma solicitação pede que uma operação especificada seja ativada, usando um ou mais objetos como parâmetros. É uma mensagem, que é recebida por diferentes objetos, solicitada pelos métodos do objeto remetente;

e) encapsulamento: é o resultado (ou o ato) de ocultar, do usuário, os detalhes da implementação de um objeto;

f) polimorfismo: é a habilidade de duas ou mais classes responderem à mesma solicitação, cada uma a seu modo. Métodos que utilizam o polimorfismo usam a mesma expressão para denotar diferentes operações;

g) herança: Classes podem ser organizadas em hierarquias, onde classes mais concretas herdam atributos e operações de classes mais abstratas (super classes). Classes que são herdadas, mantém as características das classes ancestrais, e incluem novos operadores e propriedades, sendo assim especializadas.

Além da orientação a objetos, pode-se pensar também nas aplicações distribuídas. Hoje em dia é muito comum o sistema estar distribuído em varias aplicações que podem estar numa mesma máquina ou então em uma rede de computadores.

Segundo Mainetti Junior (1997), a origem dos Objetos Distribuídos (OD) se deu a união dessas duas tecnologias, a orientação a objetos e as aplicações distribuídas. Com isso, pode-se concluir que objetos distribuídos são definidos pela utilização da tecnologia de orientação a objetos em um ambiente distribuído. Como estas duas tecnologias já estão se transformando em realidade atual na grande maioria das empresas, será uma evolução natural. OD começou a sua trajetória através do *middleware*, que é a parte de software responsável pela intercomunicação entre os vários componentes distribuídos em uma rede, mas hoje está invadindo todas as áreas, inclusive a Internet.

Conforme Jacobsen (2000), os padrões de objetos distribuídos mais conhecidos são o *Distributed Componet Obect Mode* (DCOM) e o *Common Object Resquest Broker Arquiteture* (CORBA).

O padrão DCOM foi desenvolvido pela Microsoft, que já há alguns anos vem promovendo o *Object Linking and Embedding* (OLE) como uma de suas principais tecnologias para desenvolvimento e intergração de aplicações. A camada base para o OLE é conhecida como *Component Object Model* (COM). Até então a Microsoft não tinha uma solução para o uso de objetos em um ambiente distribuído. Então com o lançamento do Windows NT 4.0 no início de 1997, a Microsoft introduziu o DCOM que é uma implementação do COM contemplando características distribuídas de aplicações desenvolvidas em uma rede de computadores com Windows NT. Não se pode ignorar a influência da Microsoft em praticamente todas as áreas da computação, e não também não se pode ignorar o fato de que a presença de um modelo de objetos distribuídos embutido no Windows NT irá promover ainda mais esta área. Mas o uso de DCOM ainda se limita ao ambiente Windows (DCOM ainda não está disponível para Unix, Macintosh, OS/2 e outros

ambientes) e a instalações que disponham de apenas Windows NT versão 4.0 ou superior (Mainetti Junior, 1997).

Já o CORBA, que é uma tecnologia desenvolvida pela *Object Management Group* (OMG), tem o mesmo princípio que o DCOM, porém não tem limitação a diversidade de plataformas. Na seção seguinte será realizado o estudo mais aprofundado deste padrão.

4.1 CORBA

Segundo Mainetti Junior (1997), CORBA é um padrão que define como objetos devem interoperar em um ambiente distribuído. Uma das características mais importantes deste padrão é a existência de uma linguagem para a definição de interfaces, chamada de *Interface Definition Language* (IDL), que é uma linguagem, também padronizada pela OMG, independente de arquitetura, que permite se especificar as interfaces dos objetos distribuídos de uma forma que todos possam requisitar serviços a eles.

A arquitetura CORBA define e implementa a estrutura necessária à comunicação entre aplicações distribuídas em diferentes plataformas, sistemas operacionais e linguagens de programação. Com CORBA, uma aplicação cliente não precisa conhecer os detalhes de implementação do objeto que obterá de um servidor. Esta capacidade é fornecida pela utilização de uma interface comum, compartilhada para a passagem de informações (Geyer, 2000).

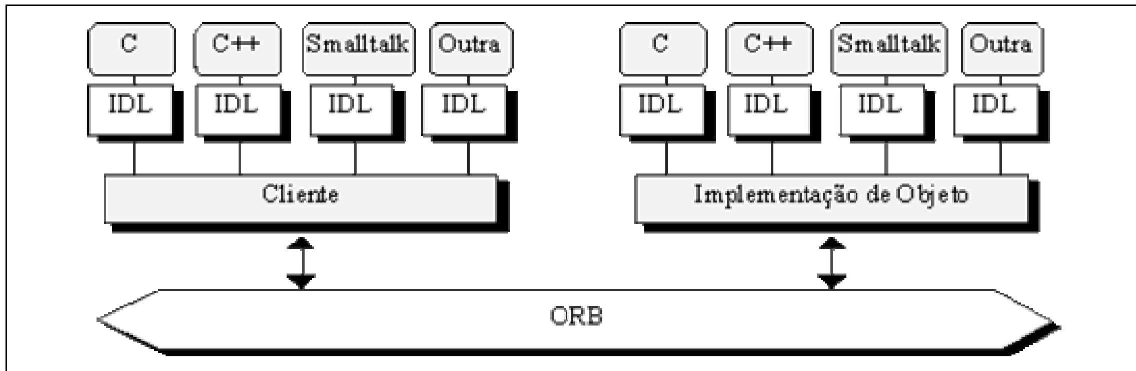
Há vários fatores que destacam o CORBA das outras tecnologias de distribuição. O principal deles é que CORBA é um padrão aberto, isto é, sua especificação está constantemente sendo revisada e atualizada pela OMG.

O elemento chave da tecnologia CORBA é o *Object Request Broker* (ORB), que gerencia o acesso dos objetos em uma aplicação, comunica estes objetos com outros, monitora suas funções, descobre suas localizações e controla a comunicação com outros ORB´s. Basicamente, o ORB é o principal mecanismo para simplificar o desenvolvimento de aplicações padrão CORBA. A simplificação é o resultado de três características: independência de localização e interoperabilidade entre plataformas e linguagens. Independência de localização significa que um ORB trata todos os objetos como se fossem locais, mesmo que estejam em sistemas remotos. Interoperabilidade entre plataformas significa que objetos criados em uma plataforma de hardware/software podem executar em qualquer outra plataforma que suporte CORBA. Por fim, interoperabilidade entre linguagens significa que objetos escritos em uma linguagem podem interagir com aplicações escritas em

outra linguagem, graças a IDL. A IDL é uma linguagem que define as interfaces dos objetos, mas não suas implementações. Os objetos podem ser escritos em qualquer linguagem (C,C++, Java, Delphi) devido a esta facilidade do padrão CORBA.

A fig. 4.2 ilustra o papel da IDL no padrão CORBA.

FIGURA 4.2 – MAPEAMENTO DA IDL PARA AS LINGUAGENS DE PROGRAMAÇÃO



Fonte: Capeletto (1999)

CORBA também inclui mecanismos para comunicação entre objetos através de uma rede. O protocolo *General Inter-ORB Protocol* (GIOP) especifica formatos de mensagens e representações de dados que garantam a interoperabilidade entre ORBs. O protocolo *Inter-ORB Protocol* (IIOP) define detalhes específicos para usar GIOP sob TCP/IP.

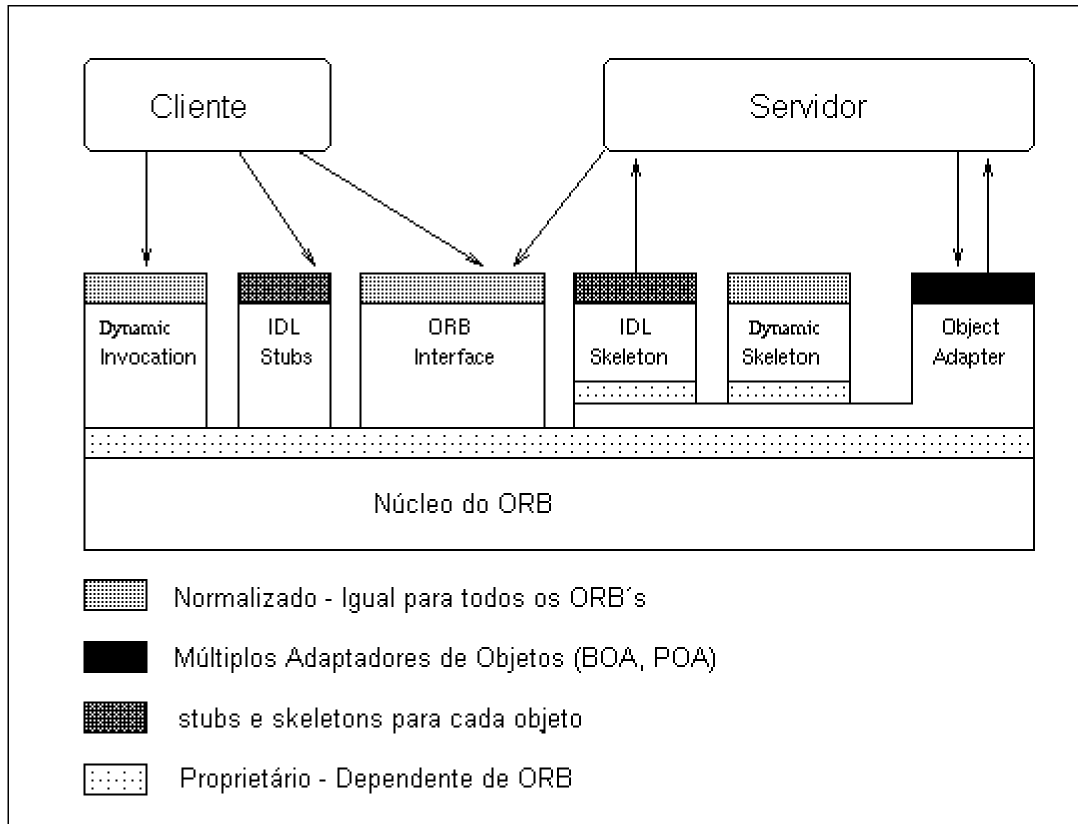
4.1.1 ARQUITETURA CORBA

Para se entender melhor a arquitetura CORBA, divide-se as aplicações em duas partes: aplicações servidoras e aplicações clientes. As aplicações servidoras vão ser as aplicações que forneceram serviços através de objetos CORBA e as aplicações clientes serão as aplicações que requisitaram algum serviço para os objetos CORBA.

Na arquitetura CORBA se encontra as *stubs* e *skeletons*. Elas são responsáveis pelo “empacotamento” dos dados. As *stubs* estão localizadas entre o ORB e o cliente, e gerenciam as mensagens enviadas do cliente para o ORB. Os *skeletons* estão entre o servidor e o ORB, e repassam as mensagens recebidas pelo ORB até o servidor.

A fig. 4.3 mostra o posicionamento das *stubs* e *skeletons* na arquitetura CORBA.

FIGURA 4.3 - *STUBS* E *SKELETONS* NA ESTRUTURA CORBA



Fonte: Capeletto (1999)

Nas próximas seções serão vistas as arquiteturas que se encontram nas aplicações clientes e servidoras.

4.1.1.1 Aplicações clientes

Uma requisição é passada do cliente, através de sua *stub* IDL, para o ORB, até o *skeleton* do objeto alvo, e finalmente recebido pela implementação do objeto. Lá será executada e seu resultado será retornado pela rota de retorno correspondente. A definição das interfaces em IDL permite que os ORB´s manipulem todos os detalhes da passagem das requisições/respostas, incluindo o formato das transações quando o cliente e o objeto destino estão em diferentes sistemas.

O cliente inicia as requisições, que podem ser passadas para o ORB através de uma *stub* IDL estática *Static Invocation Interface* (SII) ou através de *Dynamic Invocation Interface* (DII). Há muitas diferenças entre a SII e a DII; a maior de todas é que a DII permite adiar a seleção do tipo de objeto e sua operação para que esta seja feita em tempo de execução, enquanto a SII requer que essa seleção seja feita em tempo de compilação. Ou seja, a DII pode ser chamada de seleção dinâmica, enquanto a SII se caracteriza pela seleção estática. Mas existem outras diferenças, numa conseqüência da vinculação dinâmica, a DII

não consegue checar se os argumentos estão sendo passados corretamente em tempo de compilação como a SII permite.

4.1.1.1.1 Estrutura do Cliente e IDL *Stubs*

As *stubs* são geradas pelo compilador que acompanha cada ORB, ou seja, o código gerado através de uma definição IDL pode mudar de fornecedor para fornecedor.

Como em tempo de projeto a IDL pode mudar, pode-se gerar as *stubs* a qualquer momento, bastando para isso compilá-las através de um compilador IDL.

A interface cliente-*stub* é padronizada pela OMG para cada linguagem já homologada. Isso significa que o código gerado, pode ser portado de um ORB para outro, desde que na mesma linguagem. Em contraste, a interface *stub*-ORB é proprietária, ou seja, cada fornecedor de ORB tem sua própria definição, e não há maneira de compartilhar os códigos entre ORB´s. Então, compiladores IDL e ORB´s são vendidos em conjunto, como o pacote Visibroker que será usado neste trabalho. Não se pode utilizar um compilador IDL de um fornecedor e o ORB de outro, pois ambas as especificações são proprietárias.

Também faz parte da estrutura das aplicações cliente a *Dynamic Invocation Interface* (DII), ela dá ao cliente, a possibilidade de, a qualquer momento, chamar qualquer operação em qualquer objeto que pode ser acessado pela rede. Isso inclui objetos que não tem *stub* definida e novos objetos adicionados à rede ou descobertos através de serviços de busca. Para cada objeto, com ou sem *stub*, a DII permite requisições assíncronas, ou seja, o controle é retornado ao cliente tão logo a requisição foi efetuada.

As implementações de objetos não têm conhecimento se as requisições que estão recebendo do ORB estão sendo recebidas através de DII ou SII. Por isso o ORB é responsável por preparar requisições dinâmicas para que elas tenham a mesma forma de uma requisição estática antes de transmitir para a implementação do objeto. A escolha é feita pelo cliente, e o trabalho é completado pelo ORB. Programadores de objetos não precisam fazer nada a mais para que seus objetos estejam preparados para requisições via DII.

Há quatro passos para uma requisição dinâmica:

a) identificar o objeto a ser requisitado;

b) buscar na *Interface Repository* (IR) sua interface;

c) construir a requisição;

d) efetuar a requisição e esperar o resultado.

A interface do objeto alvo é encontrada na *Interface Repository* (IR) e através de uma operação do ORB: get_interface. Esta operação retorna uma referência para o objeto que será usado para montar a requisição.

A IR é um banco de dados *on-line* de informações sobre os tipos de objetos do ORB. Ela é crucial para a operação do CORBA. A especificação do CORBA exige que cada ORB suporte e implemente a interface IR, permitindo que se conheça como as definições IDL são gravadas, modificadas e recuperadas. Essas informações podem ser usadas para vários propósitos; o manual do CORBA aponta três maneiras que o ORB pode utilizá-las diretamente:

a) para permitir a interoperabilidade entre diferentes implementações de ORB´s;

b) para permitir checagem de tipos;

c) para checar a correta herança entre as interfaces.

Mas estas informações também podem ser úteis para:

a) gerenciar a instalação e distribuição das definições de interfaces pela rede;

b) durante o processo de desenvolvimento, permitir a visualização e alteração das interfaces e outras informações armazenadas na IDL;

c) gerar *stubs* e *skeletons* diretamente da IR, já que todas as informações da IDL estão disponíveis na IR.

Uma IR particular pode ser compartilhada para mais de um ORB, e, um ORB pode acessar mais de uma IR.

4.1.1.2Aplicações servidoras

Nas aplicações clientes, o máximo que se conhece das implementações dos objetos são suas definições IDL, uma para cada objeto. O cliente não consegue visualizar um servidor no outro lado do ORB. Mas no lado da implementação do objeto a estrutura da implementação é visível, e os adaptadores de objeto (*Object Adapter*) são equipados com mecanismo para localizar e repassar as referências do objeto alvo para o ORB.

Uns dos adaptadores de objeto conhecidos é o *Basic Object Adapter* (BOA). Ele é o responsável em proporcionar aos objetos e servidores as funcionalidades básicas nas configurações do ORB. O BOA oferece geração e interpretação de referências de objetos,

ativação e desativação de implementações de objetos, ativação e desativação de objetos individuais, invocação de métodos via *skeletons*, que podem ser estáticos ou dinâmicos.

Os *skeletons* estáticos de IDL funcionam nas aplicações servidoras assim como as *stubs* funcionam nas aplicações clientes. Eles se conectam ao servidor via o mapeamento padronizado de sua linguagem, e se conectam ao adaptador de objetos via uma interface proprietária de cada fornecedor. As requisições passam pelos *skeletons*, são encaminhadas ao adaptador de objetos, e finalmente são executados na implementação do objeto. Depois de executados, seguem a rota de retorno.

Os *skeletons* dinâmicos, conhecidos como *Dynamic Skeleton Interface* (DSI), permitem aos servidores despachar operações de objetos que não foram definidos estaticamente em tempo de compilação, ou seja, objetos que tem seu estado e comportamento mudado durante a execução.

Estes objetos não são gerados a partir de uma definição IDL. A DSI permite a um objeto o registro durante o tempo de execução, a recepção de requisições dos clientes e a resposta a essas solicitações sem passar pelas classes *skeletons*.

Do ponto de vista do cliente, um objeto implementado com DSI tem comportamento idêntico a qualquer outro objeto implementado com *skeletons* estáticos. Portanto, não é necessária qualquer modificação ou preocupação por parte dos clientes.

Para maiores informações sobre CORBA é possível consultar Capeletto (1999), Geyer (2000), Jacobsen (2000) e Mainetti Junior (1997).

5 DESENVOLVIMENTO DO PROTÓTIPO

Com os assuntos abordados nos capítulos anteriores, pôde-se desenvolver um protótipo de ambiente virtual distribuído multiusuário composto de um cenário e de personagens simples. A principal preocupação desse protótipo era possibilitar a participação de mais de um usuário, fazer com que houvesse um grau de sincronia aceitável entre os mesmos e a visualização do ambiente virtual através da biblioteca gráfica OpenGL.

A seção 5.1 apresenta os requisitos identificados para a construção do protótipo. Na seção 5.2, tem-se a especificação e implementação. A seção 5.3 apresenta a interface e utilização do protótipo e na seção 5.4 é feita uma análise dos resultados alcançados e limitações encontradas.

5.1 REQUISITOS IDENTIFICADOS

Para o desenvolvimento do protótipo observou-se os seguintes requisitos básicos:

a) utilização de um modelo distribuído para comunicação entre usuários no AVD;
b) envio de mensagens através de *broadcast* UDP, já que o funcionamento do protótipo está limitado a LAN´s;
c) utilização de um protocolo de comunicação baseado no DIS, através do uso da API do DIS-JAVA-VRML que implementa os PDU´s do DIS;
d) comunicação entre objetos distribuídos em aplicações baseado no padrão CORBA;
e) criação do cenário e dos personagens com a biblioteca gráfica OpenGL;
f) tratamento para entradas de dados no AVD, visto que a OpenGL não possui tratamentos para dispositivos de entrada de dados;
g) utilização da técnica de *heartbeats*, para possibilitar que usuários novos tenham conhecimento de todos os usuários no AVD.

5.2 ESPECIFICAÇÃO E IMPLEMENTAÇÃO

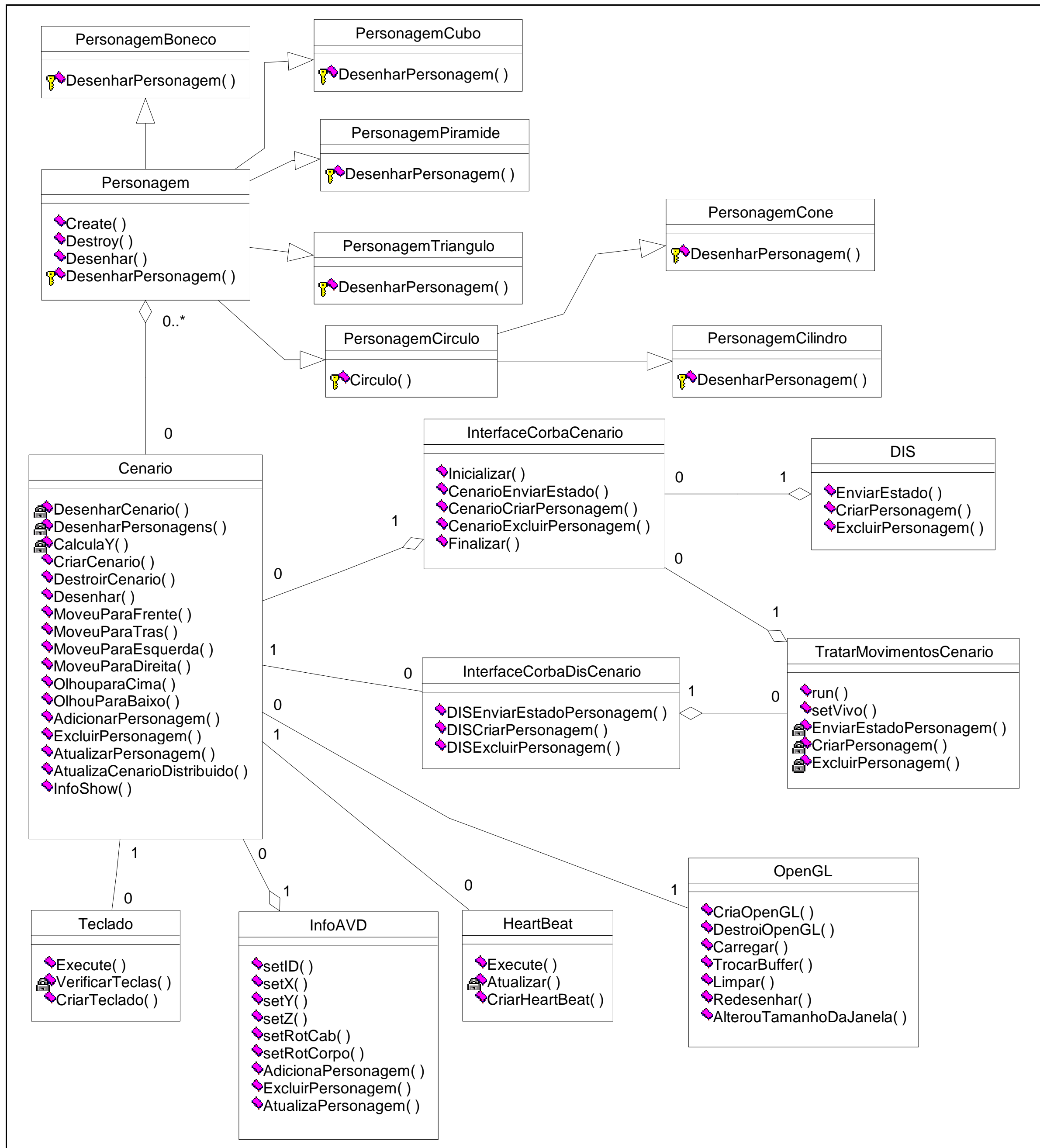
O protótipo foi especificado fazendo-se uso do padrão UML utilizando-se o diagrama de classes e através de fluxogramas para representar a funcionalidade das principais classes, e foi implementado em duas linguagens, Object Pascal e Java. Mas especificamente, foi utilizado o ambiente de programação Borland Delphi 5 para implementações na linguagem

Object Pascal e para implementar na linguagem Java foi utilizado o JSDK 1.3. A versão da API da OpenGL utilizada para a criação e manipulação do ambiente virtual foi a 1.2 e a API DIS-JAVA-VRML utilizada para o controle, envio e recebimento de PDU's, na época em que foi escrito esse trabalho, ainda não possuía um número de versão atribuído. Para implementar objetos distribuídos no padrão CORBA foi necessário a utilização do software VisiBroker versão 4.5.1 produzido pela Borland.

5.2.1 DIAGRAMA DE CLASSES

O protótipo é formado por diversas classes e cada uma delas fornece um serviço para outras classes que por sua vez também fornecem serviços. Para iniciar o estudo da implementação do protótipo, tem-se a especificação do diagrama de classes no qual o sistema é formado. Cada classe conterá uma funcionalidade única e terá sua implementação encapsulada, fazendo, desta forma, que outras classes somente acessem sua interface publicada. Na fig. 5.1 pode se ver o diagrama de classes do protótipo.

FIGURA 5.1 – DIAGRAMAS DE CLASSES DO PROTÓTIPO



Nas próximas seções serão vistas as principais características das classes que fazem parte do diagrama de classes.

5.2.2 CLASSE CENARIO

Esta classe é responsável pelo gerenciamento do cenário. Nela estão implementados os métodos responsáveis pela movimentação do usuário, como por exemplo o método `MoveuParaFrente()`, os métodos responsável pelo controle de personagens inseridos por outros usuários do AVD, como por exemplo o método `AdicionarPersonagem()` e

também é responsável pela criação da interface com o usuário, desenhando o cenário e os personagens existentes. Cada usuário terá a instância de um objeto do tipo desta classe em seu ambiente para fazer o controle do cenário.

No quadro 5.1 pode se ver a descrição e a funcionalidade de cada método que faz parte desta classe.

QUADRO 5.1 – DESCRIÇÃO DOS MÉTODOS DA CLASSE CENARIO

Métodos privados:

`DesenharCenario():` Desenha a parte estática cenário, desenhando um piso inferior e superior, a rampa, as paredes, o teto, aplica a textura aos mesmos e também aplica o efeito de iluminação;

`DesenharPersonagem():` Percorre a lista de personagens e manda uma mensagem para cada um se desenhar no cenário;

`CalculaY():` Calcula a coordenada Y de acordo com a posição X e Z do usuário no AVD. Este método se faz necessário pois como o cenário é composto por dois pisos e mais uma rampa que dá acesso aos mesmos e é preciso saber qual a elevação do usuário quando ele esta com um determinado estado.

Métodos públicos:

`CriaCenario(prForm:TForm; prOpenGL:TOpenGL; iTpPersonagem:Integer):` método utilizado para criar o cenário. Nele são passados como parâmetros a janela onde será desenhado, o objeto OpenGL e o tipo do personagem que o usuário escolheu na janela de opções inicial;

`DestroiCenario():` método utilizado para destruir o cenário. Nele serão destruído todos os objetos agregados ao cenário, como por exemplo os personagens;

`Desenhar():` método utilizado para redesenhar o cenário por completo, ou seja, a parte do cenário estática e a parte variável que seriam os demais usuários;

`MoveuParaFrente():` utilizado para mover a câmara (ponto de visão do usuário) para frente;

`MoveuParaTras():`utilizado para mover a câmara (ponto de visão do usuário) para traz;

`MoveuParaEsquerda():`utilizado para mover a câmara (ponto de visão do usuário) para esquerda;

`MoveuParaDireita():` utilizado para mover a câmara (ponto de visão do usuário) para direita;

`OlhouParaCima():` utilizado para mover a câmara (ponto de visão do usuário) para cima;

`OlhouParaBaixo():` utilizado para mover a câmara (ponto de visão do usuário) para baixo;

`AdiconarPersonagem(prID:Integer; prTipo:TTipoPersonagem; prRotacaoCorpo, prPosicaoX, prPosicaoY, prPosicaoZ:GLfloat):` utilizado para adicionar um novo personagem que acabou de entrar no AVD. Neste método são passados como parâmetro a identificação, o tipo e o estado do novo personagem;

`ExcluirPersonagem(prID:Integer):` Exclui um personagem que saiu do AVD;

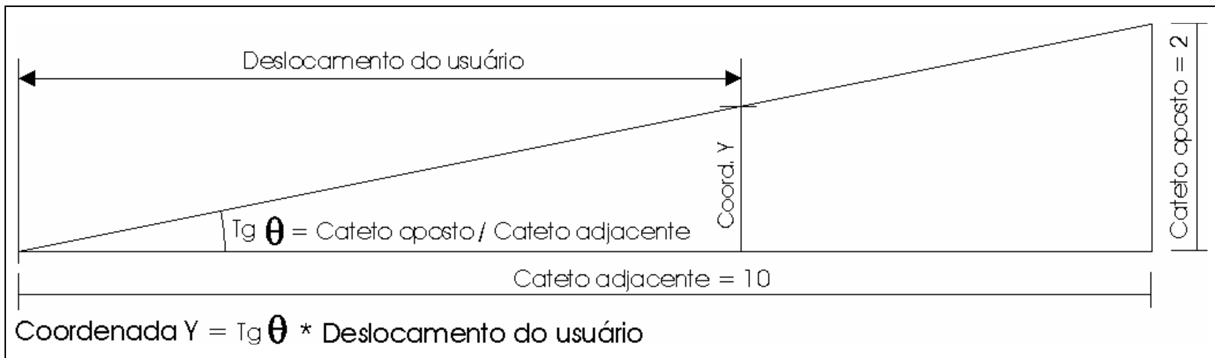
`AtualizaPersonagem(prID:Integer; prTipo:TTipoPersonagem; prRotacaoCorpo, prPosicaoX, prPosicaoY, prPosicaoZ:GLfloat):` método utilizado para informar estado do personagem, que é enviado a cada alteração que cada usuário distribuído realiza no AVD ou então quando é disparado pelo algoritmo de *heartbeat*. Se por acaso não for encontrado o usuário identificado pelo parâmetro prID, será adiciona como um novo usuário na lista de personagens;

`AtualizaCenarioDistribuido(prRedesenhar:Boolean):` Este método sempre deve ser chamado quando o usuário interagem com o cenário. Ele serve para redesenhar a cena e também atualizar outros AVD.

`InfoShow():Boolean:` Retorna verdadeiro se a janela de informação estiver ativa e falso se por acaso o usuário a fechou.

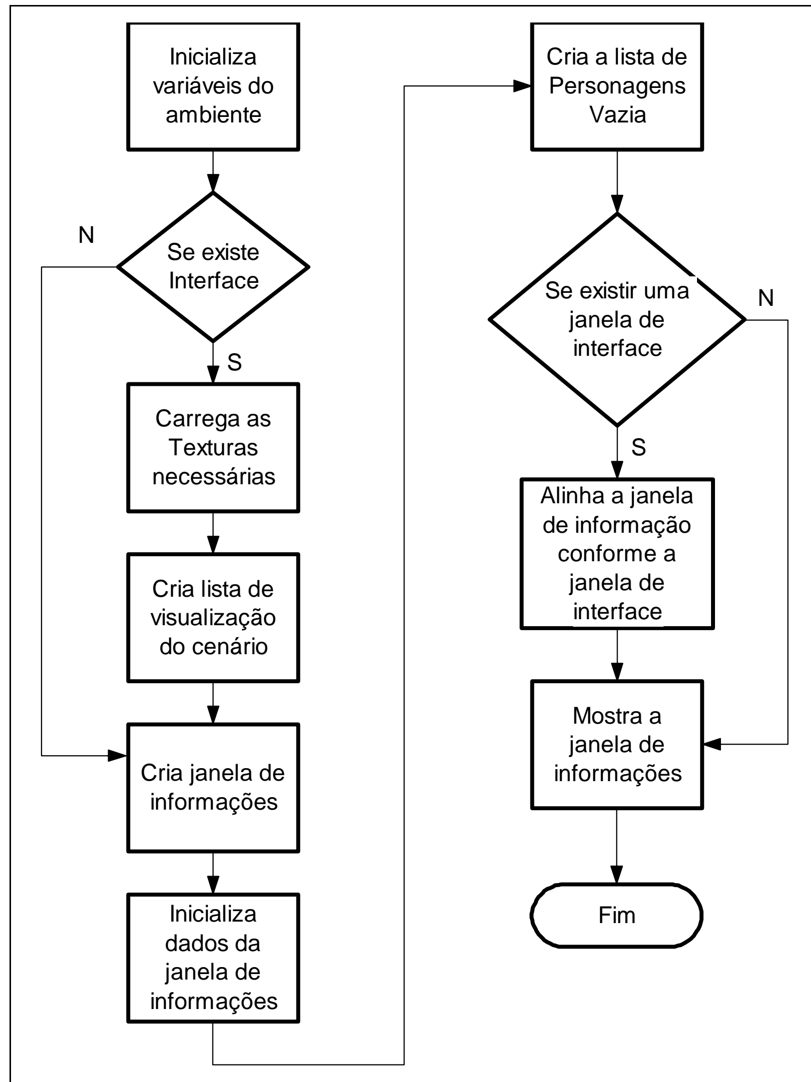
A especificação do método `CalculaY()` é bem simples, visto que o cenário é estático e composto de dois pisos, um inferior e outro superior, e uma rampa que dá o acesso a estes dois pisos. Então, se o usuário estiver no piso inferior ou superior, o seu Y será fixo, caso contrário, se o usuário estiver na rampa, será necessário fazer o cálculo de acordo com a distância de sua posição com o inicio da rampa do usuário. A fig. 5.2 exemplifica o cálculo da coordenada Y para o usuário na rampa.

FIGURA 5.2 – CÁLCULO DA COORDENADA Y PARA QUANDO O USUÁRIO ESTIVER NA RAMPA



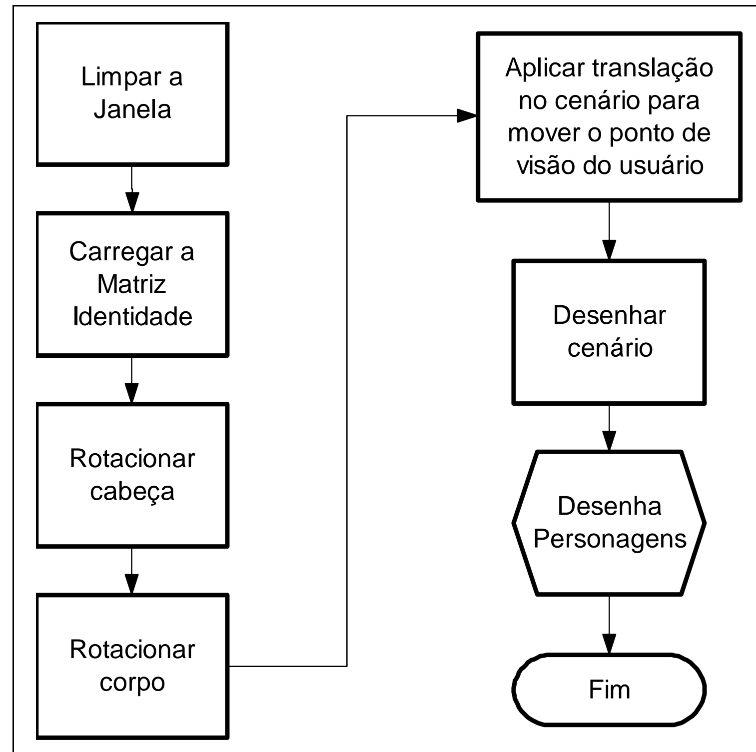
A criação do cenário pode ser feita de duas formas: uma com a interface e outra sem interface. Quando o cenário não possuir interface, somente a janela de informações dos usuários distribuídos será visualizada, caso contrário, o cenário será visualizado através de uma janela. Para definir que o cenário não utiliza interface basta passar o parâmetro *prOpenGL* do método `CriaCenario()` nulo. A fig. 5.3 mostra a especificação do método `CriarCenario()` através de um fluxograma.

FIGURA 5.3 – FLUXOGRAMA DO MÉTODO CRIARCENARIO()



O método `Desenhar()` consiste em limpar a tela e desenhar todo o cenário novamente. Ele deve ser chamado a cada vez que o cenário sofre alguma alteração. A fig. 5.4 demonstra com o fluxograma a especificação do método `Desenhar()`.

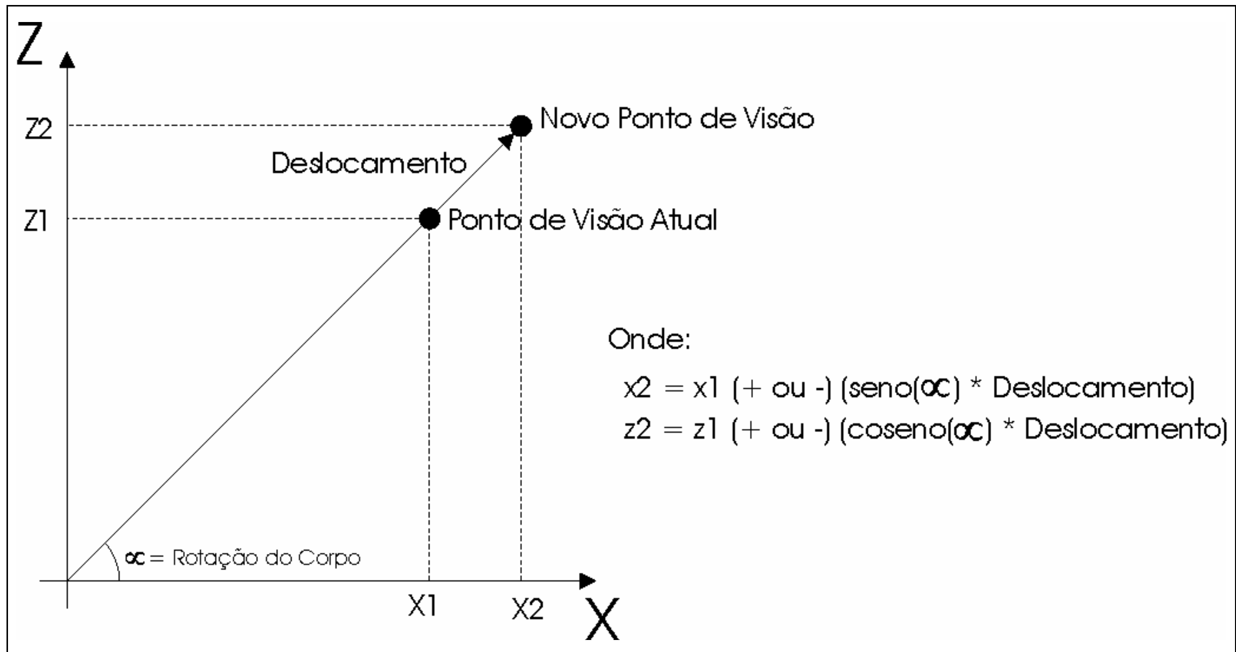
FIGURA 5.4 – FLUXOGRAMA DO MÉTODO DESENHAR()



A especificação dos métodos `MoveuParaEsquerda()`, `MoveuParaDireita()`, `OlhouParaCima()` e `OlhouParaBaixo()` é bem simples, basta somar ou diminuir das variáveis que armazenam os dados de rotação do corpo e de rotação da cabeça uma constante que representa a velocidade do movimento. Agora para especificação dos métodos `MoveuParaFrente()` e `MoveuParaTraz()` tem que se preocupar com alguns detalhes. Um deles é o cálculo das novas coordenadas. Se o usuário estiver com o corpo virado para frente ou para traz ou para o lado direito ou para o lado esquerdo não se encontra problemas, pois simplesmente seria necessário somar ou diminuir as variáveis que armazenam os dados da posição X e da posição Z da visualização do usuário. Mas como também existe a possibilidade do usuário se descolar diagonalmente se faz necessário recorrer a um cálculo para saber quais as novas coordenadas para a posição X e para a posição Z que representam o ponto de visão do usuário.

Sabendo-se o valor do deslocamento do usuário para cada interação realizada no cenário, ângulo de rotação do corpo e as coordenadas anteriores do ponto de visão do usuário, pode-se calcular as novas coordenadas através do cálculo ilustrado pela fig. 5.5.

FIGURA 5.5 – CÁLCULO DO NOVO PONTO DE VISÃO



O x2 e z2 será o novo ponto de visão do usuário. Dessa forma, não importa qual for a direção do movimento, a distância do deslocamento sempre será a mesma.

O outro ponto que se tem estar atento é ao limite do cenário. Como o cenário é estático, pode-se simplesmente testar se as novas coordenadas ultrapassam os limites do cenário, caso ultrapassem, as coordenadas antigas são restauradas, caso contrário, as novas são mantidas. Um detalhe importante é que não foi considerada a colisão com outros usuários no cenário. Nestes casos, um personagem irá fundir-se ao outro no caso de uma colisão.

O método `AtualizaPersonagem()` é especificado da seguinte maneira: primeiro é feito uma busca seqüencial numa lista de personagens pelo personagem passado como parâmetro. Depois se o mesmo foi encontrado, é atribuído seu novo estado e atualizado a janela de informações dos usuários do AVD, caso contrário, é adicionado um personagem novo ao cenário. Este método sempre será invocado quando algum PDU de estado é recebido de algum usuário do AVD.

Já o método `AtualizaPersonagemDistribuido()` é especificado de maneira diferente, primeiro ele atualiza a janela de informações, depois ele faz uma chamada ao método da classe DIS que é responsável pelo envio de um PDU de estado para o AVD. Depois disto, se o parâmetro *prDesenhar* estiver como verdadeiro e se existir interface, é feita a atualização da interface. O parâmetro *prDesenhar* é necessário porque este método é

chamado dentro da implementação da técnica de *heartbeat*, e nesse caso não é necessário redesenhar a cena novamente.

No anexo A pode-se ver a implementação do método privado `CalculaY()` e dos métodos públicos `CriaCenário()`, `Desenhar()`, `MoveuParaFrente()`, `AtualizarPersonagem()` e `AtualizaCenarioDistribuido()`, que são os principais métodos da classe Cenario.

5.2.3 CLASSE PERSONAGEM E SUAS SUBCLASSES

A classe Personagem é uma classe abstrata. Ela possui as principais funcionalidades que são em comuns entre todas as suas classes herdadas. Cada subclasse irá representar um personagem do cenário e a principal diferença de uma subclasse da outra é a forma de como será desenhado o personagem. Visto isto, teve-se a idéia de usar o polimorfismo, ou seja, várias formas de se escrever o mesmo método, para especificar os personagens.

O método `Desenhar()` da classe Personagem é responsável pelo desenho que representará o personagem do ambiente virtual. Ele consiste em aplicar uma translação, depois rotacionar o cenário e, em seguida, desenhar o personagem. Mas antes de aplicar estas transformações ao cenário, é necessário salvar o estado atual do cenário para que após estas transformações possa se restaurar o estado original do cenário. Isto se faz necessário para que outros personagens possam ser inseridos também. O quadro 5.2 mostra o código fonte da implementação deste método.

QUADRO 5.2 – IMPLEMENTAÇÃO DO MÉTODO DESENHAR()

```
procedure TPersonagem.Desenhar;
begin
  {Salva a matriz com o seu estado atual}
  glPushMatrix;

  {aplica o deslocamento do usuário fazendo uma translação}
  glTranslatef(PosicaoX, 0.2+PosicaoY, PosicaoZ);
  {aplica a rotacao no corpo do usuário}
  glRotatef(RotacaoCorpo,0,1.0,0);
  {aplica uma escala para padronizar os personagens}
  glScalef(0.2,0.2,0.2);

  {Desenha o personagem, este método é obrigado a estar implementado em
sua subclasse, caso contrário, será gerado uma excessão}
  DesenharPersonagem;

  {Restaura a matriz}
  glPopMatrix;
end;
```

O método `DesenharPersonagem()` é abstrato na classe Personagem é deve ser implementado em suas subclasses de acordo com a aparência de cada personagem. A única subclasse que não implementa este método é a PersonagemCirculo, isto porque ela é abstrata também e serve como super classe para as classes PersonagemCone e PersonagemCilindro. Isto foi especificado desta forma porque existe um método que é comum para estas duas classes, desta maneira estas passam a ser subclasses da classe PersonagemCirculo, que por sua vez é subclasse da classe Personagem, e passam a implementar o método `DesenharPersonagem()`. O quadro 5.3 exemplifica a implementação do método `DesenharPersonagem()` na classe PersonagemCubo.

QUADRO 5.3 – MÉTODO DESENHARPERSONAGEM() DA CLASSE PERSONAGEMCUBO

```
procedure TPersonagemCubo.DesenharPersonagem;
begin
  glBegin(GL_POLYGON);
    glColor3f(1,1,1);
    glVertex3f(1.0, 1.0, 1.0);
    glColor3f(0,1,1);
    glVertex3f(-1.0, 1.0, 1.0);
    glColor3f(0,0,1);
    glVertex3f(-1.0, -1.0, 1.0);
    glColor3f(1,0,1);
    glVertex3f(1.0, -1.0, 1.0);
  glEnd;

  ...

  glBegin(GL_POLYGON);
    glColor3f(0,1,0);
    glVertex3f(-1.0, 1.0, -1.0);
    glColor3f(0,1,1);
    glVertex3f(-1.0, 1.0, 1.0);
    glColor3f(1,1,1);
    glVertex3f(1.0, 1.0, 1.0);
    glColor3f(1,1,0);
    glVertex3f(1.0, 1.0, -1.0);
  glEnd;

  glBegin(GL_POLYGON);
    glColor3f(0,0,0);
    glVertex3f(-1.0, -1.0, -1.0);
    glColor3f(1,0,0);
    glVertex3f(1.0, -1.0, -1.0);
    glColor3f(1,0,1);
    glVertex3f(1.0, -1.0, 1.0);
    glColor3f(0,0,1);
    glVertex3f(-1.0, -1.0, 1.0);
  glEnd;
end;
```

5.2.4 CLASSE OPENGL

Esta classe encapsula algumas implementações necessárias para o funcionamento da biblioteca gráfica OpenGL. Entre elas pode-se citar a configuração de ambiente ativando e desativando variáveis como por exemplo a do teste de profundidade.

O quadro 5.4 descreve a funcionalidade e as principais características de cada método desta classe.

QUADRO 5.4 – DESCRIÇÃO DOS MÉTODOS DA CLASSE OPENGL

`CriaOpenGL(iForm:TForm)`: Este método cria o objeto OpenGL e inicializa as variáveis privadas deste objeto.

`DestroiOpenGL()`: Este método finaliza o objeto OpenGL.

`Carregar()`: Este método configura a janela passada como parâmetro no método CriaOpenGL para visualizar a cena criada com OpenGL através de comandos específicos para a implementação da OpenGL com o windows. Este método deve sempre ser chamado logo após a criação do objeto OpenGL e da atribuição de suas propriedades.

`TrocarBuffer()`: executa o comando *SwapBuffer* da OpenGL. Este comando serve para trocar o *buffer* de visualização e sempre deve ser chamado logo após uma alteração na cena.

`Limpar()`: método utilizado para limpar a janela. Sempre deve ser chamado antes de desenhar o cenário.

`Redesenhar()`: método utilizado para redesenhar a janela. Deve ser chamado quando existe alguma alteração no cenário por parte de um usuário distribuído.

`AlterouTamanhoDaJanela(): boolean`: Este método refaz a configuração da janela quando algum redimensionamento é feito na mesma. Este método retorna verdadeiro sempre quando é necessário redesenhar a janela.

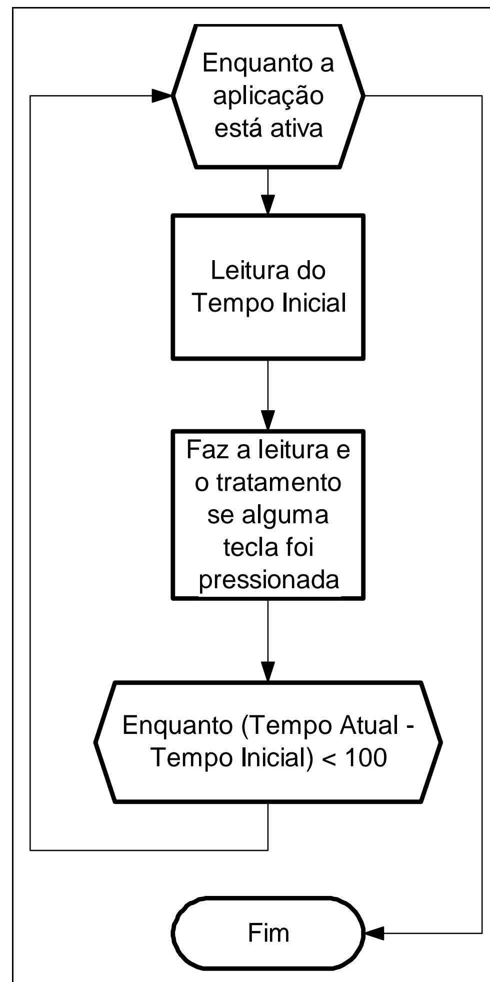
`CarregaTextura(var Textura:GLuint; prArqImagem:String)`: Este método é utilizado para ler um arquivo do formato bitmap e carregar na memória.

5.2.5 CLASSE TECLADO

A função desta classe é fazer a leitura de teclas pressionadas pelo usuário e fazer o seu tratamento. Ela deve estar funcionando como um processo paralelo e o processamento das teclas devem estar dentro de um laço de repetição que só será terminado quando a aplicação se encerar. Por este motivo ela deve herdar características de uma classe que tem como função criar processamentos paralelos na aplicação. Na linguagem Object Pascal existe uma classe que implementa esta funcionalidade chamada de *TThread*, portanto na implementação esta classe deverá ser uma subclasse desta.

Um outro ponto que deve ser especificado para esta classe é a questão do processamento das teclas em tempo real, ou seja, o tempo que se deve fazer a leitura e o tratamento das teclas deve ser igual em todos os usuários, independente da velocidade de processamento do computador que o usuário esta usando. Para isso, é arbitrado um tempo que definirá o número de quadros por segundos de acordo com as teclas pressionadas pelo usuário. A fig. 5.6 mostra um fluxograma especificando a implementação desta classe.

FIGURA 5.6 – FLUXOGRAMA ESPECIFIANDO A ESTRUTURA DA CLASSE TECLADO



O tratamento das teclas é feito de maneira bem simples, primeiro é verificado se uma determinada tecla foi pressionada e, caso ela tenha sido, é enviado uma mensagem para o cenário de acordo com a tecla pressionada. Por exemplo, se a tecla pressionada foi a seta para cima, então é feita a chamada do método `MoveuParaFrente()` do objeto que representa uma instância da classe Cenario na aplicação.

Ao final deste tratamento, caso alguma modificação tenha acontecido no cenário, existe a necessidade de enviar uma mensagem ao cenário solicitando a atualização do cenário.

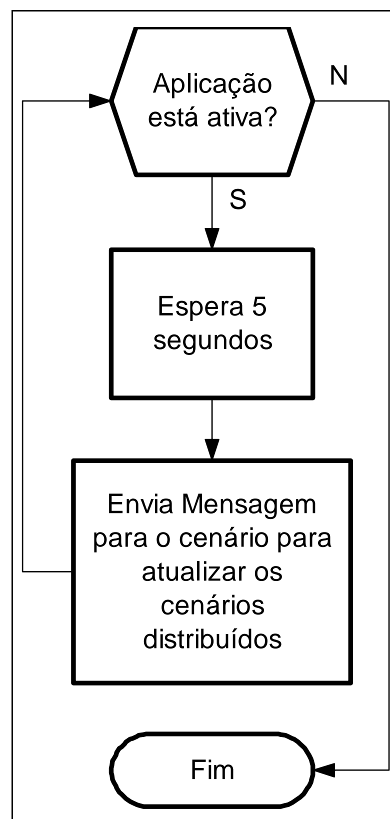
Esta mensagem pode ser passada através da chamada de um método da classe Cenario chamado `AtualizaCenarioDistribuido()`.

5.2.6 CLASSE HEARTBEAT

Esta classe implementa a técnica de *heartbeat*. Ela tem como objetivo enviar uma mensagem para o cenário de tempo em tempo solicitando a atualização dos cenários distribuídos através do método `AtualizaCenarioDistribuido()`. Assim como a classe Teclado, esta classe também deve ser executada em um processo paralelo e ficar dentro de um laço de repetição que só será finalizado ao término da aplicação.

Sua especificação é bem simples e sua estrutura pode ser visualizada melhor através do fluxograma representado na fig. 5.7.

FIGURA 5.7 – FLUXOGRAMA ESPECIFICANDO A ESTRUTURA DA CLASSE HEARTBEAT



5.2.7 CLASSE INFOAVD

Esta classe tem como objetivo apresentar uma janela com informações do AVD. Estas informações compreendem de informações do usuário no cenário, tal como, identificação, rotação do corpo, rotação da cabeça e as coordenada X, Y e Z atuais do usuário. E também de

informações de outros usuários, contendo a identificação, o tipo do personagem, as suas coordenadas e a rotação do seu corpo.

Sua especificação é bem simples, pois seus métodos somente têm a função de atualizar os dados apresentados nesta janela.

5.2.8 CLASSES DIS E TRATARMOVIMENTOSCENARIO

Estas duas classes são responsáveis pela comunicação entre os usuários. Elas fazem uso de alguns objetos fornecidos pela API DIS-Java-VRML. Objetos dessa classe sempre devem de ser instanciados quando o usuário entrar no AVD e finalizados quando o usuário sair.

A classe DIS será a responsável pelo envio de PDU´s para os demais usuários do AVD e a classe TratarMovimentosCenario será a responsável por ler e tratar estes PDU´s que foram enviados pela classe DIS de outros usuários. Um detalhe importante que se deve levar em conta é que não só PDU´s de outros usuários serão recebidos pela aplicação, mas também os PDU´s enviados pela própria aplicação. Então se deve sempre verificar quem é o proprietário do PDU e, caso seja a própria aplicação, se deve descartá-lo.

Dessa forma, ao se instanciar um objeto da classe DIS, será obtido a identificação do usuário através de um algoritmo que consiste em um somatório dos valores de cada letra que forma o nome do computador do usuário. Depois disto, será instanciado um objeto da classe TratarMovimentosCenario para ler os PDU´s recebidos. Ao finalizar esta classe, ela irá finalizar o objeto da classe TratarMovimentosCenario.

O quadro 5.5 mostra o código fonte exemplificando a inicialização do objeto DIS.

QUADRO 5.5 – CÓDIGO FONTE EXEMPLIFICANDO A INICIALIZAÇÃO DO OBJETO DIS

```
Public DIS()
{
   int temp2 = 0;
   //atribui um número de identificação pra aplicação
   //de acordo com o nome do host
   try {
      InetAddress end = InetAddress.getLocalHost();
      String temp = end.getHostName();
      for (int i = 0; i < temp.length(); i++) {
         Character ch = new Character(temp.charAt(i));
         temp2 = temp2 + ch.getNumericValue(ch.charValue());
      }
   } catch (UnknownHostException uhe) {
      System.out.println("AmbienteVirtual:constructor(): Impossível
determinar IP do host local");
   }
   appId = (short)temp2;
      // Cria a Classe responsavel por receber PDU´s de outros usuários
      tm = new TrataMovimentosCenario(appId);
   tm.start();
}
```

Os métodos EnviarEstado(), CriarPersonagem() e ExcluirPersonagem() da classe DIS têm como função empacotar as informações do usuário dentro de um respectivo PDU e enviar pela rede. Cada PDU conterá informações diferentes, mas todos eles conterão a identificação da aplicação. O quadro 5.6 exemplifica estes métodos com o código fonte.

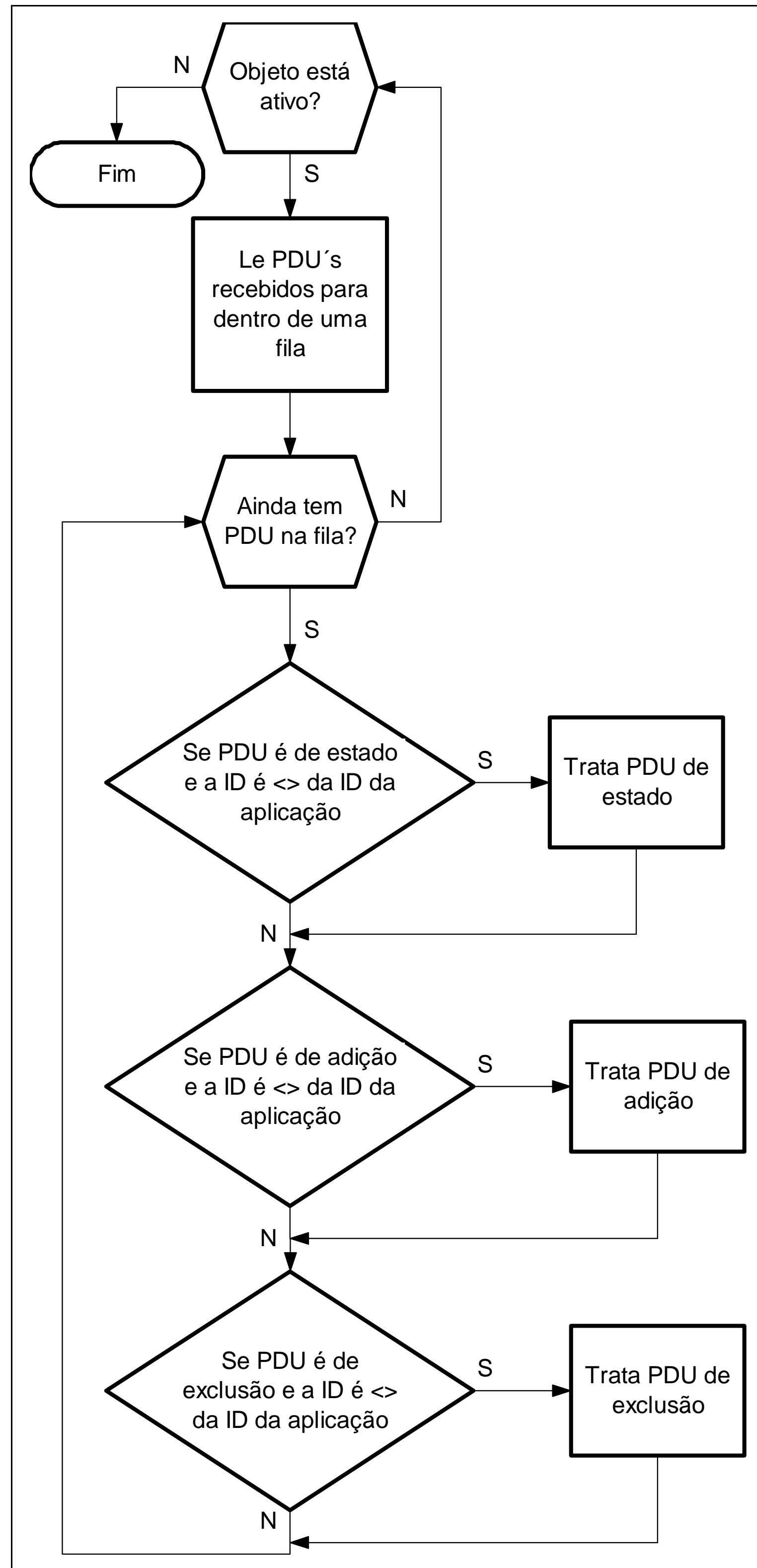
QUADRO 5.6 – CÓDIGO FONTE DOS MÉTODOS DA CLASSE DIS

```
// Metodos chamado pela classe Cenario
public short CriarPersonagem(int tpEntidade)
{
   //cria e envia um CEPDU aos demais usuário a fim de informar
   //acerca da entrada desse novo usuário

   System.out.println("Vai criar o Personagem com a ID = "+appId);

   CreateEntityPdu cePDU = new CreateEntityPdu();
   cePDU.setOriginatingEntityID(new EntityID(siteId,appId,0));
   cePDU.setRequestID(tpEntidade);
   bsb.setTimeToLive(15);
   Thread enviadorPDU = new Thread(bsb);
   cePDU.setExemplar(cePDU);
   bsb.sendPdu(cePDU.getExemplar(), endBroadCast, 8133);

   tpPersonagem = tpEntidade;

   return appId;
}

public void ExcluirPersonagem()
{
   System.out.println("Vai excluir o Personagem com a ID = "+appId);

   RemoveEntityPdu rePDU = new RemoveEntityPdu();
   rePDU.setOriginatingEntityID(new EntityID(siteId,appId,1));
```

```
   bsb.setTimeToLive(15);
   Thread enviadorPDU = new Thread(bsb);
   rePDU.setExemplar(rePDU);
   bsb.sendPdu(rePDU.getExemplar(), endBroadCast, 8133);
}

public void EnviarEstado(double prX, double prY, double prZ, float
prRotacao)
{

   System.out.println("Vai enviar o Estado do Personagem com a ID =
"+appId);

   EntityStatePdu esPDU = new EntityStatePdu();

   esPDU.setEntityID(siteId,appId,(short)1);
   esPDU.setEntityAppearance(tpPersonagem);
   //insere as informações que determinam as novas coordenadas dessa
entidade
   esPDU.setEntityLocationX(prX);
   esPDU.setEntityLocationY(prY);
   esPDU.setEntityLocationZ(prZ);
   esPDU.setEntityOrientationPhi(prRotacao);
   esPDU.setExemplar(esPDU);
   bsb.setTimeToLive(15);
   Thread enviadorPDU = new Thread(bsb);
   //envia ESPDU aos demais usuários
   bsb.sendPdu(esPDU.getExemplar(), endBroadCast, 8133);
}
```

Já a classe TratarMovimentosCenario deve ser instanciada como uma *Thread*, fazendo com que seja criado um processamento paralelo e independente da aplicação. Esta *Thread* deve executar um laço de repetição que somente será finalizado quando o objeto for finalizado. Dentro deste laço são lidos os PDU´s recebidos de outros usuários e são tratados de acordo com o tipo do PDU. Para cada tipo de PDU existe um método para desempacotar o PDU e enviar a informação para a aplicação desenvolvida em Object Pascal. A fig. 5.8 apresenta um fluxograma especificando o tratamento dos PDU´s recebidos.

FIGURA 5.8 – TRATAMENTO DOS PDU´S RECEBIDOS



5.2.9 CLASSES DE INTERFACE

Estas classes têm como objetivo criar uma interface entre o cenário que será implementado na linguagem Object Pascal e as classes criadas na linguagem Java que fornecem serviços de comunicação entre os usuários distribuídos. Os objetos instanciados através destas classes deverão estar disponíveis para que outras aplicações tenham acesso aos seus serviços.

A classe InterfaceCorbaCenario deverá ser implementada na linguagem Java e deve ser instanciada por um objeto logo no inicio da aplicação Java e ficar disponível para que a aplicação implementada em Object Pascal consiga obter um ponteiro para este objeto. Dessa forma, quando a aplicação Object Pascal for iniciada, ela deve de obter um ponteiro para o objeto da classe InterfaceCorbaCenario que foi instanciado pela aplicação Java.

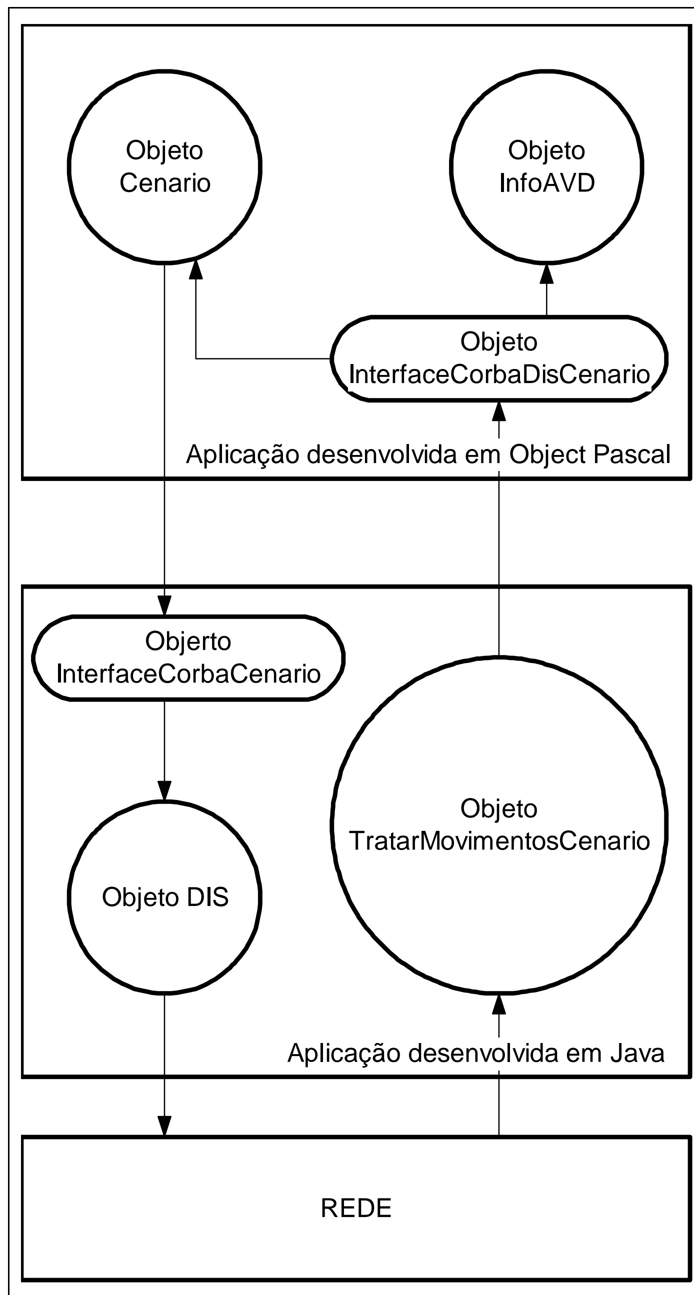
Os métodos `Inicializar()` e `Finalizar()` da classe InterfaceCorbaCenario servem para iniciar e finalizar as classes DIS e TrataMovimentosCenario. Já os métodos `CenarioEnviarEstado()`, `CenarioCriarPersonagem()` e `CenarioExcluirPersonagem()` servem para passar mensagens ao objeto DIS. Dessa forma, estes últimos métodos só poderão ser chamados depois da chamada do método `Inicializar()` e antes da chamada do método `Finalizar()`, caso contrário, uma exceção será gerada, pois o objeto DIS não foi instanciado e a variável que aponta para o objeto conterá um apontamento inválido.

Já a classe InterfaceCorbaDisCenario é implementada na linguagem Object Pascal e deverá ficar disponível para que a aplicação Java possa acessá-lo a qualquer momento. Mas especificamente, quem conterá um ponteiro para este objeto é a classe TrataMovimentosCenario, que, a cada PDU recebido, será enviado uma mensagem ao Objeto da classe InterfaceCorbaDisCenario de acordo com o tipo do PDU que por sua vez pode ser um PDU de estado, um PDU para criar um personagem ou um PDU para excluir um personagem.

Os métodos da classe InterfaceCorbaDisCenario deverão passar mensagens correspondentes ao método para o cenário, se estiver instanciado ou para a janela de informações dos usuários caso não exista uma instância do cenário.

A fig. 5.9 apresenta o fluxo esquematizado da troca de mensagens com as classes de interface.

FIGURA 5.9 – TROCA DE MENSAGENS NAS CLASSES DE INTERFACE



Observa-se que quando há necessidade de trocar mensagens entre as aplicações, se faz o uso dos objetos utilizados como interface.

Como estas interfaces são implementadas como objetos distribuídos com o padrão CORBA, se faz necessário a criação da sua interface através da linguagem IDL. O quadro 5.7 exemplifica a IDL dos dois objetos de interface do protótipo.

QUADRO 5.7 – IDL DOS OBJETOS DE INTERFACE

```
// IDL do Objeto InterfaceCorbaCenario
module ServerDIS {
  interface CorbaCenario {
    void inicializar();
    void CenarioEnviarEstado(in double prX, in double prY, in double prZ,
                             in float Rotacao);
    long CenarioCriarPersonagem(in long prTipo);
    void CenarioExcluirPersonagem();
    void finalizar();
  };
};

// IDL do Objeto InterfaceCorbaDisCenario
module Prototipo
{
  interface ICorbaDisCenario;


  interface ICorbaDisCenario
  {
    void DISEnviarEstado(in short prID, in long prTpPersonagem, in double
                         prX, in double prY, in double prZ,
                         in float Rotacao);
    void DISCriarPersonagem(in short prID, in long prTpPersonagem);
    void DISExcluirPersonagem(in short prID);
  };

  interface CorbaDisCenarioFactory
  {
    ICorbaDisCenario CreateInstance(in string InstanceName);
  };
};
```

5.3 FUNCIONAMENTO DO PROTÓTIPO

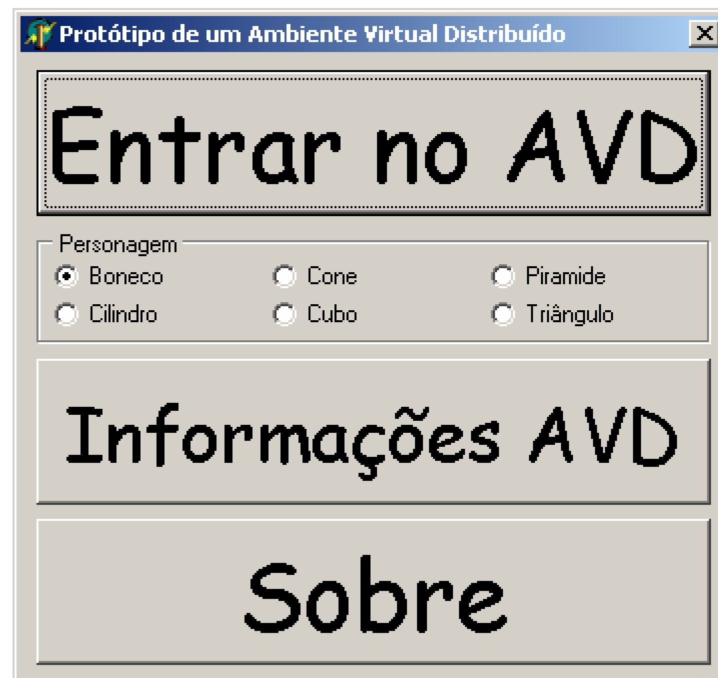
Nesta seção será apresentada a interface do protótipo, assim como seu funcionamento. Fazem parte do protótipo duas aplicações, mas somente uma possui interface e permite a interação do usuário enquanto a outra aplicação fica processando em segundo plano. Além destas duas aplicações, existem também mais outras duas aplicações que funcionam como agentes para possibilitar a comunicação entre os objetos distribuídos e também devem estar executando em segundo plano. O anexo B apresenta os passos necessários para execução das aplicações que o protótipo necessita.

A interface do protótipo é bem simples, ela está dividida em três janelas. A primeira é uma janela de opções, onde o usuário irá selecionar a opção desejada e as outras duas são a janela que irá representar graficamente o cenário e a janela que trará informações do usuário e dos demais personagens presentes no AVD.

5.3.1 JANELA DE OPÇÃO

Esta janela é a primeira a ser apresentada quando iniciada a aplicação. Nela o usuário poderá escolher qual o personagem que ele deseja usar e entrar no AVD, poderá só obter informações do AVD sem interagir e visualizar o cenário, e também terá a opção visualizar informações sobre o protótipo. A escolha do personagem não influenciará em nenhum ponto para o usuário, mas definirá qual a aparência do mesmo para os outros AVD. A fig. 5.10 mostra a janela de opções.

FIGURA 5.10 – JANELA DE OPÇÃO



5.3.2 JANELA DE INFORMAÇÕES DO AVD

A função dessa janela é fornecer informações do AVD para o usuário. Na parte superior é mostrada a identificação do usuário, as coordenadas X, Y e Z do usuário no cenário e a rotação do corpo e da cabeça do usuário. Na parte inferior da janela é exibida uma tabela com as informações dos usuários existentes no AVD onde nas colunas da tabela é exibida a identificação, as coordenadas e a rotação do usuário em relação ao cenário.

Nesta janela o usuário não tem nenhuma opção para interagir com o protótipo, somente é uma janela informativa. A fig. 5.11 apresenta a janela de informações do AVD.

FIGURA 5.11 – JANELA DE INFORMAÇÕES DO AVD

Informações do AVD

Identificação: 156
Rotação da Cabeça: 0,0000000000
Rotação do Corpo: 324,0000000000
Coordenada X: -0,8816778660
Coordenada Y: 0,0000000000
Coordenada Z: 1,2135254145

ID	Tipo	Coord. X	Coord. Y	Coord. Z	Rotação Corpo
130	Boneco	0,0000000000	0,0000000000	0,0000000000	172,0000000000

5.3.3 JANELA DO CENÁRIO

Esta é a janela onde será desenhado o cenário. Nela o usuário poderá visualizar todo o cenário no seu estado atual. Nota-se que o ambiente é estático e os personagens que estão participando do cenário são dinâmicos e podem variar de posição de acordo com a interação de cada usuário com o seu ambiente virtual.

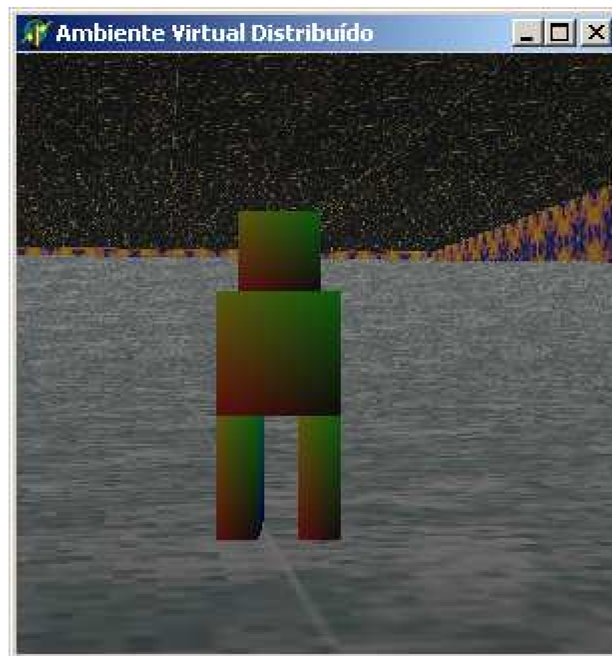
Para interagir com o cenário, o usuário deve fazer uso do teclado. As teclas de navegação atribuídas às interações deste protótipo podem ser vistas na tabela 5.1.

TABELA 5.1 - TECLAS PARA A NAVEGAÇÃO NO AMBIENTE VIRTUAL

Tecla	Efeito
Seta para cima	Desloca câmara para frente
Seta para baixo	Desloca câmara para trás
Seta para esquerda	Rotaciona câmara para a esquerda
Seta para direita	Rotaciona câmara para a direita
Page down	Rotaciona câmara para cima
Page up	Rotaciona câmara para baixo

A fig. 5.12 demonstra esta janela como usuário focando o fundo do cenário e logo a frente do mesmo está localizado um personagem que representa outro usuário do AVD.

FIGURA 5.12 – JANELA DE INTERFACE



5.4 ANÁLISE DOS RESULTADOS

Com o desenvolvimento deste protótipo pôde-se estabelecer aspectos negativos e positivos em relação ao objetivo deste trabalho, que é desenvolver um ambiente virtual distribuído multiusuário, com o uso das tecnologias escolhidas. Para se obter uma melhor análise dos resultados, o desenvolvimento do protótipo será dividido pelas tecnologias utilizadas, que no caso foram a biblioteca gráfica OpenGL para implementar a interface gráfica, alguns objetos que foram implementados pela API do DIS-Java-VRML para realizar a comunicação entre usuários distribuídos, e o CORBA que foi utilizado para realizar a troca de mensagens dos objetos distribuídos na camada de interface entre as duas aplicações.

A utilização da biblioteca gráfica OpenGL permitiu a abstração na criação do cenário. Isto foi possível porque ela especifica e implementa varias técnicas de realismo, como por exemplo, a aplicação de texturas em polígonos do cenário. A OpenGL também deu suporte com comandos de alto nível para as transformações geométricas que foram necessárias no protótipo, como por exemplo, translação, rotação e escala.

Outro ponto positivo do uso da OpenGL foi a vantagem dela ser uma biblioteca com o padrão bem difinido pela *Silicon Graphics Inc.*, que é a empresa criadora do padrão, e ser de código fonte aberto. Além disto ela é especificada como uma biblioteca multiplataforma, mas como ela foi implementada no protótipo pelo ambiente de programação Delphi, ficou vinculada ao sistema operacional Windows, deixando desta forma de ser multiplataforma por

causa da linguagem utilizada. Mas se um trecho do código fonte referente a utilização da OpenGL for retirado de um código fonte da implementação, o mesmo deve de funcionar em outra linguagem pois os nomes dos comandos não mudam de uma implementação para outra.

Uns dos aspectos que se pode definir como negativo para criação de cenários, os mais complexos principalmente, é que a criação das cenas em OpenGL são geradas através de primitivas gráficas, como por exemplo pontos, segmentos de retas e polígonos. Já em outras bibliotecas gráficas existentes no mercado encontram-se mais facilidades por existirem objetos já pré-definidos na biblioteca. Mas existem diversas bibliotecas comerciais utilitárias que fornecem formas de objetos já pré-definidas para facilitar a criação de cenas com a OpenGL.

A utilização da API do DIS-Java-VRML abstraiu bastante a implementação de mecanismos para enviar PDU´s pela rede. Um dos pontos importante da utilização é que além dos mecanismos de envio de dados pela rede ela também implementa os vinte e setes PDU´s especificados pelo DIS em forma de classes, dessa forma a API se tornou uma ferramenta bem aproveitada para atingir o objetivo do trabalho.

Uma das coisas que facilitou a implementação do mecanismo para gerenciar o AVD foi a utilização do protocolo UDP, que é um protocolo não orientado a conexão. Dessa forma, sempre que um usuário tinha algum PDU para ser enviado, ele era enviado em *broadcast* para todos os outros usuários e não ficava no aguardo de uma resposta se cada usuário recebeu ou não o pacote transmitido. Dessa forma não se precisou desenvolver um gerenciador central para o AVD.

Um aspecto negativo da utilização do protocolo UDP é o tráfego gerado sobre a rede, pois o número de pacotes que são enviados pela mesma é alto, fazendo-se desta forma a performance da rede não ser das melhores. Como geralmente as redes locais, no caso as LAN´s, operam com uma taxa de transmissão alta, este fator não influenciou tanto no desempenho do AVD, mas para redes com taxa de transmissão mais baixa, este seria um ponto crítico a ser revisto.

A técnica de *heartbeat* utilizada foi essencial para a utilização do envio dos pacotes utilizando-se o protocolo UDP. Esta técnica também influenciou no tráfico gerado na rede, pois a cada cinco segundos era enviado um PDU de estado do usuário através do protocolo UDP.

Outra vantagem de se utilizar a API do DIS-Java-VRML é que seu código fonte é aberto e sua implementação segue o padrão especificado pelo DIS. Dessa forma, se fosse

necessário fazer algum ajuste no código fonte, para corrigir ou adequar-se a alguma situação, seria fácil.

Apesar de todos os pontos positivos que a API do DIS-Java-VRML apresenta para o desenvolvimento de um AVD, ela foi desenvolvida especificamente para simulações militar, isto porque ela seguiu a especificação do DIS. Dessa forma ela não seria bem aproveita em qualquer AVD, pois dependendo da finalidade do AVD, ela não seria aplicada ou não atenderia plenamente a especificação do AVD.

A utilização do CORBA para a criação de objetos distribuídos atendeu plenamente a necessidade que se tinha. Ele facilitou em muito a troca de mensagens entre as aplicações desenvolvidas nas linguagens Object Pascal e Java. O CORBA é um padrão bem definido e bem especificado pela OMG, dessa forma ele é confiável e bem documentado.

Infelizmente a utilização de objetos distribuídos junto com o padrão CORBA apresentou alguns aspectos negativos. Entre eles se pode citar o custo de processamento, que é essencial para a computação gráfica envolvida no desenvolvimento. Outro aspecto negativo que se pode citar também é a configuração do ambiente, que no caso é muito complicada, tendo que deixar rodando em paralelo duas aplicações, uma sendo o VisiBroker Smart Agent e a outra o Irep, que é um repositório de interfaces utilizado para declarar a interface do objeto distribuído.

Como um todo se pode ver que para cada recurso utilizado para atingir o objetivo deste trabalho encontraram-se aspectos positivos e negativos. É certo que a combinação de recursos utilizados neste trabalho não são estáticos, podendo ser substituído por outras tecnologias, como por exemplo, o CORBA poderia ser substituído pelo DCOM e, ao invés de ter utilizado a OpenGL, poderia ter sido utilizado o DirectX para implementar a interface gráfica. Mas estas escolhas foram pessoais e a idéia era justamente poder testar e analisar os resultados obtidos.

O capítulo seguinte apresenta as conclusões alcançadas ao longo do desenvolvimento desse trabalho, bem como as possíveis extensões que podem enriquecer ainda mais o conteúdo e protótipo resultante desse trabalho.

6 CONCLUSÕES

Com o término do desenvolvimento do protótipo, pode-se concluir que é viável a construção de um ambiente virtual distribuído utilizando-se a biblioteca gráfica OpenGL para construção da interface gráfica, que no caso do trabalho era composta de um cenário estático com os seus personagens se movendo de forma dinâmica de acordo com a interação de cada usuário distribuído. A utilização da API do DIS-Java-VRML, que implementa as especificações do DIS e foi utilizado para transmitir protocolos UDP para enviar diferentes PDU´s para outros usuários. E também a utilização do padrão CORBA, que serviu como uma ponte entre duas aplicações para possibilitar a chamada de mensagens entre um objeto e outro.

O desenvolvimento do protótipo trouxe muita experiência na criação da interface gráfica com OpenGL e na utilização de objetos distribuídos para criar um protótipo na área de computação gráfica e mais especificamente em realidade virtual.

O modelo escolhido para o desenvolvimento do AVD, que foi o modelo distribuído, facilitou em muito o desenvolvimento do protótipo, pois não foi necessário o desenvolvimento de um gerenciador para centralizar e controlar o AVD.

A utilização da API do DIS-Java-VRML já era conhecida, pois com o trabalho desenvolvido por Eduardo (2001), a API passou a ser mais bem definida e sua utilização deixou de ser tão relevante ao trabalho. A utilização da API talvez tivesse relevância se fossem utilizados outros protocolos além do UDP para comunicação, como por exemplo, um protocolo orientado a conexão como o TCP.

6.1 EXTENSÕES

As possíveis extensões que podem ser feitas a partir desse trabalho estão enumeradas a seguir:

a) melhorar a aparência do ambiente virtual, incluindo no mesmo elementos adicionais que o tornem mais real, o que resultaria no estudo da utilização de recursos mais avançados na API da OpenGL;

b) permitir que os usuários possam de alguma forma interagir com outros elementos do ambiente virtual, como por exemplo objetos a sua volta;

c) melhorar o processo de comunicação entre os ambientes, bem como o controle do mesmo, fazendo uso de outros tipos de PDU's que não foram utilizados no desenvolvimento desse protótipo;

d) fazer com que o ambiente funcione também em WAN's. Isso é interessante principalmente porque levaria ao estudo da aplicação de *multicast* e técnicas adicionais relacionadas à ambientes virtuais distribuídos de grande escala como áreas de interesse;

e) implantar técnicas de tratamento de colisão entre o usuário e outros usuário e, se caso existir algum objeto na cena, com o mesmo também;

f) melhorar a peformance do AVD, como por exemplo, aplicando algoritmos de *Dead Reckoning* para dimininuir o número de PDU´s envidados pela aplicação;

g) utilizar uma outra API que tenha como função a comunicação entre os usuários, fazendo uso da interface gráfica deste trabalho e utilizando a camada de objetos distribuídos para trocar mensagens entre a nova API e a interface gráfica;

h) Implementar a especificação do DIS conforme a IEEE na linguagem Object Pascal, tendo-se como exemplo a própria implementação da API DIS-Java-VRML, visto que o seu código fonte é aberto.

ANEXO A: PRINCIPAIS MÉTODOS DA CLASSE CENARIO

O quadro 6.1 apresentado abaixo exemplifica a implementação dos principais métodos apresentados pela especificação da classe cenário, que é a principal classe do protótipo.

QUADRO 6.1 – IMPLEMENTAÇÃO DOS PRINCIPAIS MÉTODOS DA CLASSE CENARIO

```
// Método CalculaY()
function TCenario.CalculaY:GLFloat;
begin
  {Verifica se está no primeiro Piso}
  if (PosicaoZ >= -10) and (PosicaoZ <= 10) then
    Result := 0
  else {Verifica se esta no piso superior}
  if (PosicaoZ <= -20) then
    Result := 2
  else {senão está na rampa}
    {Calculo = Coef. Angular * Posicao do usuário na rampa}
    {Coef. Angular = Cateto Oposto / Cateto Adjecente}
    Result := -2/10 * (PosicaoZ + 10);
end;

// Método CriaCenario()
constructor TCenario.CriaCenario(prForm:TForm; prOpenGL:TOpenGL;
iTpPersonagem:Integer);
begin
  inherited Create;

  {Inicializa variáveis}
  PodeMatar := True;
  OpenGL := prOpenGL;
  PosicaoX := 0;
  PosicaoZ := 0;
  PosicaoY := CalculaY;

  if OpenGL <> nil then
  begin
    {Carrega a Textura}
    OpenGL.CarregaTextura(TextParede,'parede.bmp');
    OpenGL.CarregaTextura(TextChao,'chao.bmp');
    OpenGL.CarregaTextura(TextTeto,'teto.bmp');

    {Cria o ponteiro para a lista vazia}
    Cenario := glGenLists(1);
    {Desenha o Cenario na Lista}
    glNewList(Cenario,GL_COMPILE_AND_EXECUTE);
      DesenharCenario;
    glEndList;
  end;

  {Cria a Janela de Informações}
  InfoAVD := TFrmInfoAVD.Create(Application);

  InfoAVD.setID(0);
  InfoAVD.setX(PosicaoX);
  InfoAVD.setY(PosicaoY);
  InfoAVD.setZ(PosicaoZ);
```

```
  InfoAVD.setRotCab(RotacaoCabeca);
  InfoAVD.setRotCorpo(RotacaoCorpo);

  {Cria a Lista de Personagens}
  Personagens := TCollection.Create(TNodoLista);

  // Cria o ponteiro para o Objeto CorbaCenario
  CorbaInitialize;
  CorbaCenario := ORB.Bind('IDL:ServerDIS/CorbaCenario:1.0');
  // Inicializa as Classes que tratam o DIS
  CorbaCenario.inicializar;

  // Cria o Personagem em outros Ambientes Virtuais
  if OpenGL <> nil then
    InfoAVD.setID(CorbaCenario.CenarioCriarPersonagem(iTpPersonagem));

  if prForm <> nil then
  begin
    InfoAVD.Left := (Screen.Width div 2) - (InfoAVD.Width div 2);
    InfoAVD.Top  := (Screen.Height div 2) - ((InfoAVD.Height +
prForm.Height) div 2) + prForm.Height;
    prForm.Left := (Screen.Width div 2) - (prForm.Width div 2);
    prForm.Top  := (Screen.Height div 2) - ((InfoAVD.Height +
prForm.Height) div 2);
  end;
  InfoAVD.Show;
end;

// Método Desenhar
procedure TCenario.Desenhar;
begin
  if OpenGL = nil then
    exit;

  {Limpa a Tela e restaura matriz original}
  glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);
  glLoadIdentity();

  {Rotaciona a cena para o ângulo de visão do usuário}
  glRotatef(RotacaoCabeca,1.0,0,0);
  glRotatef(360 - RotacaoCorpo,0,1.0,0);

  {Move o personagem para seu ponto fazendo uma translação sobre
   o eixo X e Z}
  glTranslatef(-PosicaoX, -Passo - 0.75 - PosicaoY, -PosicaoZ);

  {Chama a lista pré-compilada do cenário para desenha-la}
  glCallList(Cenario);

  {Desenha os personagens pertencentes ao cenário}
  DesenharPersonagens;
end;

// Método MoveuParaFrente()
procedure TCenario.MoveuParaFrente;
var
  PosicaoXAnt,
  PosicaoZAnt:GLFloat;
begin

  {Salva posições anteriores, no caso de passar dos limites é restaurado a
posição anterior}
```

```
  PosicaoXAnt := PosicaoX;
  PosicaoZAnt := PosicaoZ;

  {Calcula as novas posições, considerando que o personagem possa estar em
uma diagonal em relação ao eixo X e Z}
  PosicaoX := PosicaoX - sin(Diagonal*PiSobre180) * 0.3;
  PosicaoZ := PosicaoZ - cos(Diagonal*PiSobre180) * 0.3;

  {Verifica se a nova posição não extrapolou os limites}
  if (PosicaoX > LimiteMaxX) or (PosicaoX < LimiteMinX) then
    PosicaoX := PosicaoXAnt;
  if (PosicaoZ > LimiteMaxZ) or (PosicaoZ < LimiteMinZ) then
    PosicaoZ := PosicaoZAnt;

  {Calcula a nova posição do Y em relação as coordenadas X e Z}
  PosicaoY := CalculaY;

  {Calcula as variáveis utilizadas para o efeito da caminhada}
  if (AngPasso = 360) then
    AngPasso := 0
  else
    AngPasso := AngPasso + 10;
  Passo := sin(AngPasso * PiSobre180)/20.0;
end;

// Método AtualizarPersonagem()
procedure TCenario.AtualizaPersonagem(prID:Integer;
prTipo:TTipoPersonagem; prRotacaoCorpo, prPosicaoX, prPosicaoY,
prPosicaoZ:GLfloat);
var
  i:integer;
  achou:Boolean;
begin
  {Flag para sinalizar as threads instanciadas que ainda existe processo
corrente}
  PodeMatar := False;

  {Flag para indicar se localizou o usuário}
  achou := False;

  {Faz uma busca sequencial para achar o personagem pelo ID}
  for i := 0 to Personagens.Count - 1 do
    {Faz o teste para ver se a identificação é a mesma}
    if TNodoLista(Personagens.Items[i]).Personagem.ID = prID then
    begin
      with TNodoLista(Personagens.Items[i]).Personagem do
      begin
        {Se achou atualiza o flag e atualiza o estado do personagem}
        Achou := True;
        RotacaoCorpo := prRotacaoCorpo;
        PosicaoX := prPosicaoX;
        PosicaoY := prPosicaoY;
        PosicaoZ := prPosicaoZ;
        {Atualiza a janela contento as informações dos usuários
distribuídos}
        if Assigned(InfoAVD) then
          if InfoAVD.Visible then

InfoAVD.AtualizarPersonagem(prId,RotacaoCorpo,prPosicaoX,prPosicaoY,prPosi
caoZ);
      end;
      Break;
```

```
    end;

  {Se não achou o personagem na lista então adiciona}
  if not Achou then

AdiconarPersonagem(prId,prTipo,RotacaoCorpo,prPosicaoX,prPosicaoY,prPosica
oZ);

  {Atualiza o flag}
  PodeMatar := True;

  {Resedenha o cenário por completo}
  if OpenGL <> nil then
    OpenGL.Redesenhar;
end;

// Método AtualizaCenarioDistribuido()
procedure TCenario.AtualizaCenarioDistribuido(prRedesenhar:Boolean);
var
  Coord:array[1..3] of Double;
begin
  PodeMatar := False;

  {Atualiza a janela contento as informações dos usuários distribuídos}
  if Assigned(InfoAVD) then
    if InfoAVD.Visible then
    begin
      InfoAVD.setX(PosicaoX);
      InfoAVD.setY(PosicaoY);
      InfoAVD.setZ(PosicaoZ);
      InfoAVD.setRotCab(RotacaoCabeca);
      InfoAVD.setRotCorpo(RotacaoCorpo);
    end;

  // Envia Estado para outros usuários
  Coord[1] := PosicaoX;
  Coord[2] := PosicaoY;
  Coord[3] := PosicaoZ;

CorbaCenario.CenarioEnviarEstado(Coord[1],Coord[2],Coord[3],RotacaoCorpo);

  // Desenha no próprio cenário
  if (prRedesenhar) and (OpenGL <> nil) then
  begin
    Desenhar;
    OpenGL.TrocarBuffer;
  end;

  PodeMatar := True;
End;
```

ANEXO B: PASSOS A SEREM SEGUIDOS PARA A EXECUÇÃO DO PROTÓTIPO

Para que seja possível executar o protótipo, ou recompilar os fontes do mesmo, é necessário que os seguintes passos sejam seguidos:

a) ter instalado a plataforma de desenvolvimento Java, no caso o JSDK versão 1.3, ou no mínimo, o ambiente de exeução Java correspondente a essa versão (JRE 1.3). Pode-se obter a instalação na *home page* da Sun, no endereço http://java.sun.com/products/. Em "*Product Shortcuts*" selecione a versão desejada;

b) ter instalado o ambiente de programação Borland Delphi 5 com suporte a tecnologia CORBA;

c) ter instalado o arquivo do tipo **`jar`** correspondentes à API do DIS-Java-VRML no diretório **`C:\vrtp\`**. Para adquirir as classes implementadas da API acesse o endereço http://www.web3d.org/WorkingGroups/vrtp/dis-java-vrml/download.html e baixe o arquivo **`dis-java-vrml.tar.gz`** ou **`dis-java-vrml.zip`**, descompacte-o num novo diretório qualquer e mova o arquivo **`dis-java-vrml.jar`** que estará nesse diretório onde foi descompactado, para o diretório **`c:\vrtp`**. Depois disto, é necessário configurar a variável de ambiente **`CLASSPATH`** com o seguinte valor: **`c:\vrtp\dis-java-vrml.jar;.;`**. Para maiores informações sobre a instalação, consultar em Web3D (2001);

d) ter instalado o VisiBroker for Java versão 4.5.1. Um detalhe importante na instalação do mesmo é o seu diretório de instalação, pois de maneira nenhuma ele deverá ser instalado no mesmo diretório onde foi instalado o VisiBroker que foi instalado pela instalação do Delphi. Para obter-se uma versão demostrativa do software, pode se acessar o endereço http://www.borland.com;

e) ter instalado a API do OpenGL. Geralmente está API vem junto com a própria instalação do Windows. Mas, dependendo de como foi instalado o mesmo, pode existir a necessidade de instala-la. Para isso, pode-se adquirir a instalação da mesma no endereço http://www.opengl.org.

Depois destes softwares estarem instalados e configurados corretamente, já é possivel a execução do protótipo. Abaixo segue uma explicação de como e quais aplicações devem ser iniciadas para que o protótipo funcione de maneira correta. Nota-se que tem-se duas

aplicações que funcionam como agentes e que não foram implementadas por este trabalho, mas que se fazem necessárias para o funcionamento do mesmo.

A primeira aplicação a ser iniciada é o VisiBroker Smart Agent. Este software implementa o ORB para que os objetos possam se comunicar. Depois dele ser iniciado ele permanecerá em forma de ícone no canto direito da barra de ferramentas do windows. Este software faz parte de um conjunto de softwares que acompanham o VisiBroker for Java, que é um produto da Borland.

Depois disto, deve se publicar no Repositório de Interface a IDL do objeto distribuído desenvolvido em Java. Para isto, deve-se chamar o aplicativo **irep** através da seguinte linha de comando no diretório **prototipo\dis**: **irep –console serverdis serverdis.idl**. Isto se faz necessário porque foi implementado em Object Pascal um cliente para objeto distribuído que foi iniciado na aplicação Java, e este cliente conecta-se ao objeto distribuído de maneira dinâmica, ou seja, a aplicação só ficará conhecendo a interface do objeto distribuído em tempo de execução. Nota-se que tem-se dois aplicativos com o nome de **irep**, um deles pertence ao pacote do VisiBroker for Java e o outro faz parte do VisiBroker instalado pelo Delphi. Para funcionamento correto é necessário que seja executado o **irep** do VibiBroker instalado pelo Delphi.

Depois de ter iniciado o VisiBroker Smart Agent e publicado a IDL, deve ser iniciada a aplicação implementada em Java. Esta aplicação instanciará um objeto da classe InterfaceCorbaCenario e ficará aguardando a iniciação da aplicação implementada em Object Pascal. Para execução da mesma é necessário executar a seguinte linha de comando no diretório **prototipo\dis\vbj DIS**.

E por último inicia-se a aplicação implementada em Object Pascal. Esta aplicação é a que fornecerá toda a interface com o usuário, as demais somente estão funcionando em segundo plano para usuário. Para executar esta aplicação executa-se a seguinte linha de comando: **prototipo\prototipo.exe**.

Revisando os passos para executar as aplicações são os seguintes:

a) Iniciar o ORB Smart Agent;

b) Iniciar a aplicação Java;

c) Executar o Interface Repository e carregar a IDL do objeto distríbuido;

d) Executar a aplicação Object Pascal.

REFERÊNCIAS BIBLIOGRÁFICAS

CAPELETTO, Johni Jeferson. **Comunicacao entre objetos distribuidos utilizando a tecnologia Corba (Common Object Request Broker Architecture).** 1999. 60f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

EDUARDO, Vandeir. **Protótipo de um ambiente virtual distribuído multiusuário.** 2001. 108f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

GEYER, Cláudio Fernado Resin. **Usando Borland Delphi para imprlementar aplicações CORBA.** Porto Alegre, 2000. Disponível em: <http://www.inf.ufrgs.br/procpar/disc/dsitec08/trabalhos/sem2000-1/cb_svd/>. Acesso em 09 jun. 2002.

GOSSWEILER, R. *et al*. **An introductory tutorial for developing multi-user virtual environments.** Virginia: [s.n.], [1994]. Disponível em: <http://citeseer.nj.nec.com/gossweiler94introductory.html>. Acesso em: 09 jun. 2002.

HARDT, James; WHITE, Kevin. **Distributed interactive simulation (DIS)**. Florida, 1998. Disponível em: <http://www-ece.engr.ucf.edu/~jza/classes/4781/DIS/project.html>. Acesso em 09 jun. 2002.

IEEE, Institute of Electrical and Electronics Engineers. **IEEE Std 1278: Standard for information technology, protocols for distributed interactive simulation.** [S.l], 1993.

______. IEEE Std 1278.1: Standard for distributed interactive simulation: application protocols. [S.l], 1995a.

______. IEEE Std 1278.1a: Standard for distributed interactive simulation: supplement to IEEE Std 1278.1-1995. [S.l], 1998.

______. IEEE Std 1278.2: Standard for distributed interactive simulation: communication services and profiles. [S.l], 1995b.

______. **IEEE Std 1278.3: Recommended practice for distributed interactive simulation: exercise management and feedback.** [S.l], 1996.

______. **IEEE Std 1278.4: Trial-use recommended practice for distributed interactive simulation: verification, validation, and accreditation.** [S.l], 1997.

ISDALE, Jerry. **What is virtual reality?** Ontario: [s.n.], 1993. Disponível em: <ftp://sunee.uwaterloo.ca/pub/vr/documents/whatisvr.txt>. Acesso em: 08 nov. 2000.

JACOBS, Jon Q. Delphi developer´s: **guide to OpenGL.** Texas : Wordware, c1999.

JACOBSEN, Douglas Thomas. **Sistema de apoio ao coordenador do simulador de empresas virtual utilizando a tecnologia CORBA.** 2000. 55f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

KIRNER, Cláudio; PINHO, Márcio S. **Uma introdução à realidade virtual,** São Carlos, [2000?]. Disponível em: <http://grv.inf.pucrs.br/Pagina/TutRV/tutrv.htm>. Acesso em: 09 jun. 2002.

MACEDONIA, Michael. **A network software architecture for large scale virtual environments.** 1995. 200 p. Dissertação de doutorado (Doutor de filosofia em ciências da computação), Naval Postgraduate School, Monterey.

MACEDONIA, Michael.; ZYDA, Michael. A Taxonomy for networked virtual environments. **IEEE Multimedia**, [S.l.], v. 4, n. 1, p. 48-56, jan./mar. 1997.

MAINETTI Junior, Sergio. **Objetos Distribuídos,** Curtiba, 1997. Disponível em: <http://www.visionnaire.com.br/downloads/artig-od.pdf>. Acesso em: 09 jun. 2002.

MICROSOFT. **Welcome to the Microsoft Corporate web site**, [S.l.] mar. [2002]. Disponível em: <http://www.microsoft.com >. Acesso em: 09 jun. 2002.

MOLOFEE, Jeff. **OpenGL Tutorials**. Disponível em: <http://nehe.gamedev.net>. Acesso em: 09 jun. 2002.

PINHO, Márcio S. *et al*. **Um modelo de interface para navegação em mundos virtuais.** Porto Alegre: [s.n.], 1999. Disponível em:

<http://grv.inf.pucrs.br/Pagina/Publicacoes/Bike/Portugues/Bike.htm>. Acesso em: 09 jun. 2002.

SHAW, Chris.; GREEN, Mark. The MR toolkit peers package and experiment. **In:** IEEE Virtual Reality Annual International Symposium (VRAIS 93), Washington, p. 463-469, set. 1993.

SILICON GRAPHICS INC. **OpenGL - high performance 2D/3D graphics.** [S.l.][2001?]. Disponível em: <http://www.opengl.org/>. Acesso em: 09 jun. 2002.

STEVENS, W. Richard. **TCP/IP illustrated**. Reading: Addison-Wesley, v. 1, 1995.

STYTZ, Martin R. Distributed virtual environments. **IEEE computer graphics and applications**, Los Alamitos, n. 3, p. 19-31, maio/jun. 1996.

SZWARCMAN, Dilza.; FEIJÓ, Bruno.; COSTA, Mônica. **A framework for networked emotional characters.** [S.l.: s.n.], [1999]. Disponível em: <ftp://ftp.inf.puc-rio.br/pub/docs/techreports/99_23_szwarcman.pdf>. Acesso em: 02 mar. 2002.

WEB3D, Web3D Consortium. **Distributed Interactive Simulation DIS-Java-VRML Working Group.** [S.l.: s.n.], [2001?]. Disponível em: <http://www.web3d.org/WorkingGroups/vrtp/dis-java-vrml/>. Acesso em: 09 jun. 2002.

WLOKA, Mathias M. Lag in multiprocessor VR. **PRESENCE: Teleoperators and Virtual Environments**, [s.l.], v. 4, p. 50-63, 1995.

WRIGHT, Richard S, SWEET, Michael. OpenGL SuperBible. 2.ed. Indianopolis : Waite Group Press, c2000. xxiii, 696p.

WOO, Mason; NEIDER, Jackie; DAVIS, Tom; Shreiner, Dave. OpenGL programming guide: **the official guide to learning OpenGL**, version 1.2. 3.ed. Boston : Addison Wesley, 1999.