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

VANDEIR EDUARDO

BLUMENAU, JUNHO/2001.

2001/1-67

PROTÓTIPO DE UM AMBIENTE VIRTUAL DISTRIBUÍDO MULTIUSUÁRIO

VANDEIR EDUARDO

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 Cesar Rodacki Gomes

Prof. Maurício Capobianco Lopes

AGRADECIMENTOS

Agradeço em primeira instância a Deus que me abençoou com o dom da dedicação aos estudos e pelo sentimento de prazer pela busca constante do conhecimento. Em segunda instância, mas não menos importante, agradeço a minha amada mãe: alicerce de nossa família e da minha vida.

Agradecimentos merecidos também vão para Luciano Kienolt e meu orientador professor Dalton Solano dos Reis, cuja força e paciência dispensadas, foram de grande importância no período em que estive dedicado ao desenvolvimento desse trabalho.

A todos vocês obrigado de coração!

SUMÁRIO

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE SIGLAS E ABREVIATURAS

2D	Duas Dimensões
3D	Três Dimensões
API	*Application Program Interface*
ARPA	*Advanced Research Projects Agency*
AVD	Ambiente Virtual Distribuído
bps	bits por segundo
DIS	*Distributed Interactive Simulation*
ESPDU	*Estity State Protocol Data Unit*
IP	*Internet Protocol*
LAN	*Local Area Network*
ms	milisegundos
NPSNET	*Naval Postgraduate School Network*
PDU	*Protocol Data Unit*
SIMNET	*Simulator Network*
TCP	*Transport Control Protocol*
UDP	*User Datagram Protocol*
WAN	*Wide Area Network*
WIMP	*Window, Icon, Menu, Pointer*
WoW	*Window on World*

RESUMO

Este trabalho apresenta fatores relacionados à construção de mundos virtuais. Mais especificamente, ele procura abordar as características relevantes a ambientes virtuais distribuídos, concentrando-se principalmente nas técnicas e recursos utilizados para manter a sincronia e consistência dos ambientes virtuais. Também aborda técnicas que procuram aproveitar de forma mais otimizada os recursos da rede, numa tentativa de contornar os problemas de utilização de largura de banda (*bandwidth*), distribuição (*distribution*), latência (*latency*) e confiabilidade (*reliability*). Este trabalho também demonstra como a utilização da *Application Program Interface* (API) do Java3D e do DIS-Java-VRML contribuíram, respectivamente, para a construção do mundo virtual e para a implementação dos mecanismos responsáveis pelo controle da comunicação entre os usuários.

ABSTRACT

This work presents factors related to the construction of virtual environments. More specifically, it tries to approach the important characteristics to distributed virtual environments, concentrating mainly on the techniques used to maintain the synchronism and consistence of the virtual environments. It also approaches techniques that try to optimize use of network resources, in an attempt of outlining the problems of bandwidth, distribution, latency and reliability. This work also demonstrates as the use of Application Program Interface (API) of Java3D and DIS-Java-VRML contributed, respectively, for the construction of the virtual environment and programming of the mechanisms for the control of the communication among the users.

1 INTRODUÇÃO

1.1 CONTEXTUALIZAÇÃO / JUSTIFICATIVA

Com o avanço e conseqüente amadurecimento na construção de programas, um item que começou a receber maior atenção e dedicação foi a interface entre o usuário e o programa desenvolvido. Um dos objetivos principais disso é justamente procurar tornar a interação do usuário a mais natural e intuitiva possível.

A maioria das interfaces construídas se limitava a apresentar as informações aos usuários somente através de imagens em duas dimensões (2D). Como o mundo em que vivemos nos é apresentado em três dimensões (3D), começou-se a desenvolver aplicações que utilizassem imagens em 3D, numa tentativa de tornar a interface do usuário mais próxima da realidade.

Entre essas aplicações desenvolvidas, que faziam uso de imagens em 3D, apareceram as primeiras que procuravam simular um ambiente virtual.

Segundo Pinho (1999), um ambiente virtual é um cenário em 3D gerado por computador, através de técnicas de computação gráfica, procurando representar um ambiente real ou fictício. Geralmente, esses ambientes são caracterizados pela sua geração dinâmica e pela capacidade dos usuários interagirem com ele.

A princípio, a interface desses ambientes virtuais era do tipo *Window*, *Icon*, *Menu*, *Pointer* (WIMP) (Pinho, 1999) ou *Window on World System* (WoW) (Isdale, 1993), nas quais a interação com as imagens do ambiente virtual era feita através do uso de monitores comuns e mouse.

Numa fase seguinte, com o desenvolvimento de equipamentos como capacetes, luvas e salas de projeção, a interação com os ambientes virtuais gerados por computador deixa de ser não imersiva, procurando fazer com que o usuário passe a se sentir dentro do próprio ambiente gerado, caracterizando a imersão no mundo virtual.

Dessas duas formas de interação com os mundos virtuais surge a primeira classificação de ambientes virtuais: os imersivos e não imersivos.

A próxima classificação provém da capacidade desses ambientes suportarem um ou mais usuários e também permitirem que esses usuários compartilhem o mesmo ambiente virtual estando em computadores separados.

Essa segunda característica traz uma série de detalhes que precisam ser observados e que serão fatores importantes para o desempenho adequado do ambiente virtual. Entre eles, segundo Macedonia (1997), estão a limitação imposta pela largura de banda da rede sobre a qual o ambiente virtual está interconectado, qual a melhor forma de distribuir o mundo virtual (se de forma centralizada ou distribuída), como contornar os problemas de latência e a confiabilidade do ambiente virtual que trata as questões de sincronia e consistência entre os vários usuários participantes do mesmo.

Concentrando-se principalmente nesses 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 específicas da linguagem Java: a API do Java3D (Sun, 2000a) e a API do DIS-Java-VRML (Web3D, 2001a). 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.

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 e como a utilização da API do Java3D e do DIS-Java-VRML foram utilizadas para a implementação do protótipo.

Com isso, pôde-se estudar e aplicar de forma prática algumas técnicas já desenvolvidas que procuram contornar esses quatro fatores importantes dos 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) avaliar o funcionamento da API do Java3D e do DIS-Java-VRML para a construção de ambientes virtuais distribuídos;

b) pesquisar as principais características que devem ser observadas no processo de implementação de ambientes virtuais distribuídos.

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 e algumas técnicas utilizadas para contornar esses problemas.

O capítulo três apresenta as principais características da API do Java3D e o quatro as do DIS-Java-VRML.

O capítulo cinco apresenta a especificação, implementação e o funcionamento do protótipo de ambiente virtual multiusuário distribuído, demonstrando a aplicação de algumas técnicas pesquisadas no capítulo dois.

No capítulo seis são apresentadas as conclusões provenientes da execução desse trabalho, bem como as possíveis extensões que dele podem ser desenvolvidas.

2 AMBIENTES VIRTUAIS DISTRIBUÍDOS

Ao mesmo tempo em que os ambientes virtuais podem ser aplicados nas mais variadas áreas (educação, simulação, visualização científica, jogos), também a sua forma de construção pode variar bastante. Os ambientes desenvolvidos podem variar desde ambientes que suportam somente uma pessoa, sendo executados em um único computador, até ambientes multiusuários, onde cada usuário compartilha o ambiente virtual através de computadores interconectados.

As primeiras experiências com ambientes virtuais distribuídos estão associadas a aplicações de simulações militares feitas por universidades dos Estados Unidos para o Departamento de Defesa e a alguns esforços em desenvolver jogos multiusuários.

Entre os primeiros projetos de simulação desenvolvidos figura 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) e em Locke (1994).

Outro projeto interessante, desenvolvido pela *Naval Postgraduate School*, é o NPSNET. Os primeiros resultados obtidos desse projeto datam de 1991 e foram apresentados na conferência de 1991 da Siggraph. O NPSNET também figura entre os ambientes virtuais que tem por objetivo interconectar vários usuários, permitindo a simulação de batalhas militares. Atualmente, o NPSNET encontra-se na versão cinco, sendo que sua construção e evolução resultaram em publicações valiosas na área de desenvolvimento de ambientes virtuais distribuídos. Para maiores informações, umas das melhores fontes de pesquisa é a própria página na internet do grupo de pesquisa NPSNET (Npsnet, 2000).

São justamente as publicações resultantes desses projetos que servem como fonte de pesquisa para os principais aspectos que devem ser observados ao se construir um ambiente virtual distribuído (AVD). Esses aspectos vão desde a organização do modelo geral de comunicação (centralizado ou distribuído), protocolos de comunicação atualmente disponíveis mais indicados para serem utilizados, até a atenção que deve ser dada às questões de utilização da largura de banda, latência e confiabilidade inerente a rede que está sendo utilizada para a interconexão. Esses aspectos serão abordados em seguida.

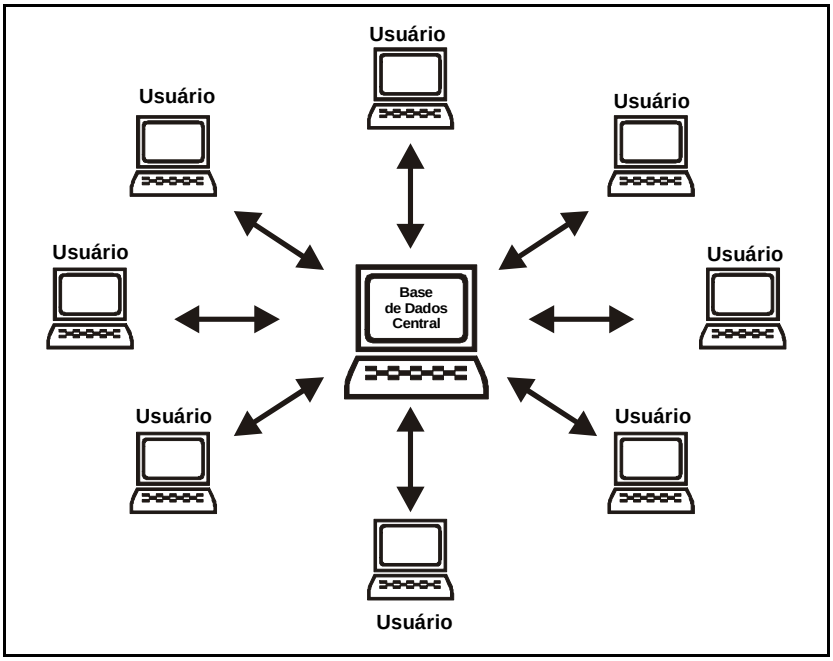
2.1 MODELOS GERAIS DE COMUNICAÇÃO DE UM AVD

Segundo Gossweiler (1994) os dois métodos mais populares de implementar um AVD são o modelo centralizado e o modelo distribuído.

No modelo centralizado, um computador principal recebe todos os dados provenientes dos usuários conectados ao mundo virtual, aplica essas informações ao ambiente virtual armazenado no computador central e devolve, para cada um dos usuários conectados, o resultado das alterações feitas sobre o mundo virtual, atualizando-os.

Essa forma de comunicação pode ser visualizada graficamente na fig. 2.1.

FIGURA 2.1 - MODELO CENTRALIZADO DE COMUNICAÇÃO



Fonte: adaptado de Gossweiler (1994)

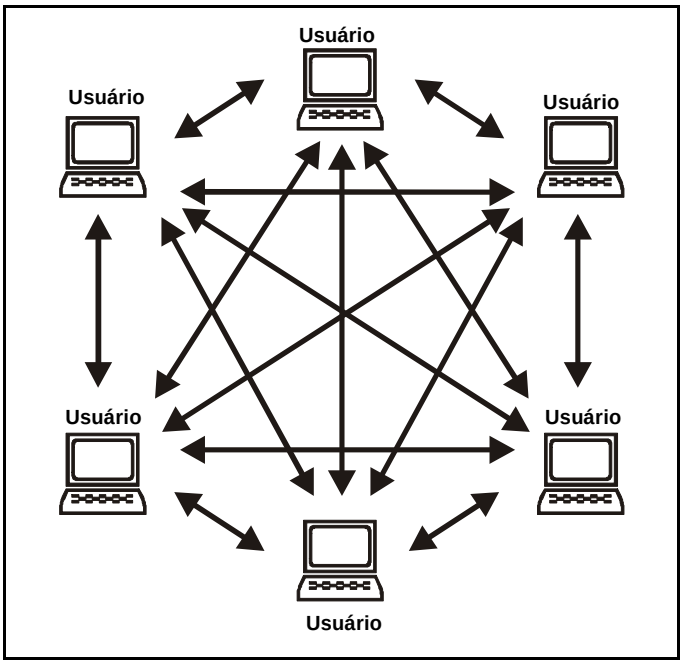
Apesar desse modelo apresentar facilidades com relação à implementação dos mecanismos de controle da comunicação entre os usuários do AVD, existe um problema sério de escalabilidade. A escalabilidade, aqui abordada, refere-se à capacidade do ambiente virtual crescer, no sentido de suportar o aumento significativo no número de usuários. À medida que a quantidade de usuários do AVD cresce, maior se torna o tráfego de mensagens com o computador central, fazendo com que a velocidade de acesso caia devido ao excesso de mensagens as quais o computador central precisa receber, processar e devolver.

Como forma de resolver essa limitação de escalabilidade do modelo de comunicação centralizado, tem-se o modelo distribuído.

Nesse modelo, cada usuário do AVD mantém uma cópia própria do ambiente virtual, realizando, ele mesmo, as tarefas de renderização, computação e animação dos objetos do ambiente virtual. Dessa forma, quando um usuário realiza alguma modificação sobre o ambiente virtual, ele se encarrega de comunicar a todos os outros usuários sobre as alterações realizadas.

A fig. 2.2 ilustra esse modelo de comunicação.

FIGURA 2.2 - MODELO DE COMUNICAÇÃO DISTRIBUÍDO



Fonte: adaptado de Gossweiler (1994)

Porém, ao mesmo tempo em que esse modelo de comunicação resolve o problema de congestionamento de mensagens sobre um único computador, que é o caso do modelo centralizado, ele também cria outro. Como no modelo distribuído cada usuário fica encarregado de comunicar aos demais usuários sobre as mudanças realizadas sobre o ambiente virtual, um número maior de mensagens é gerado. Além desse problema, o número de conexões também cresce, já que cada usuário precisa usar um canal de comunicação com cada um dos demais usuários.

Assim, seja qual for o modelo de comunicação utilizado, é importante observar que nenhum deles apresenta-se como o ideal. Por isso, o estudo de três fatores diretamente ligados a esses aspectos é importante: 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.

2.2 LARGURA DE BANDA (*BANDWIDTH*)

Segundo Szwarcman (1999), a largura de banda de uma rede pode ser definida como a capacidade máxima de se transferir uma quantidade determinada de informações num determinado período, geralmente expressa em bits por segundo (bps).

Num AVD é a largura de banda da rede utilizada para interconectar os usuários que determinará o tamanho e a riqueza de detalhes do ambiente virtual. Entre esses fatores, o número de usuários é que irá exercer maior influência com relação a utilização da largura de banda. Logicamente, à medida que o número de usuários do AVD aumenta, também aumenta o tráfego gerado na rede pelas constantes mensagens de atualização geradas.

Em redes locais (*Local Area Network* – LAN), conforme Macedônia (1997), pelo fato da maioria operar a 10 Mbps e o número de usuários ser relativamente limitado, a largura de banda disponível é geralmente suficiente. Porém, quando se interconecta os usuários de um AVD sobre uma rede de longa distância (*Wide Area Netword* – WAN), o quadro se torna diferente.

Isto porque a partir do momento em que se utiliza uma WAN para interconectar os usuários do ambiente virtual, geralmente larguras de banda mais modestas estão disponíveis, algo em torno 1,5 Mbps. Levando-se em consideração que numa WAN o número de usuários interconectados pode ser bem maior, nem sempre essa largura de banda é suficiente para garantir um desempenho satisfatório do ambiente virtual.

Num modelo de comunicação distribuído não é difícil perceber como o aumento do número de usuários aumenta consideravelmente o uso da largura de banda. Como nesse modelo cada modificação no ambiente virtual feita por um usuário precisa ser comunicada a todos os outros usuários, isto significa que para um número N de usuários, N-1 mensagens precisam ser enviadas a cada alteração.

2.3 LATÊNCIA (*LATENCY*)

Entre algumas definições da palavra latência dadas por Ferreira (1980), a mais apropriada é aquela que associa latência ao tempo decorrido entre a aplicação de um estímulo e a resposta provocada. No contexto das redes de computadores, pode-se dizer que latência é o período de tempo decorrido, medido em unidades de tempo, geralmente milisegundos (ms), entre o envio de um pacote de dados e o seu recebimento no computador de destino.

Segundo Macedonia (1997), é o controle da latência que determinará se a interação e dinâmica do ambiente virtual serão satisfatórias para os usuários do ambiente virtual. Se o objetivo do ambiente virtual é simular um mundo real, este precisa operar em tempo real, considerando-se a percepção humana.

Wloka (1995) diz que um AVD precisa garantir que a entrega de pacotes tenha uma latência mínima para permitir que as imagens do ambiente virtual sejam geradas, para cada um dos usuários, numa freqüência suficiente para garantir a ilusão de realidade. Os valores ideais dessa freqüência variam numa faixa de 30 a 60 quadros por segundo.

Mais uma vez, em se tratando de LAN's, a latência não apresenta maiores problemas. O desafio se encontra nas WAN's, onde o longo caminho que um pacote precisa percorrer até chegar ao seu destino, passando por vários equipamentos que interconectam redes, contribui significativamente para o aumento da latência.

Porém, é importante lembrar que não só a latência pode comprometer a ilusão de realidade de um ambiente virtual. Itens como a interface de rede, memória e capacidade de processamento de cada um dos computadores conectados ao ambiente virtual também são importantes para garantir uma freqüência satisfatória de quadros por segundo.

2.4 CONFIABILIDADE (*REABILITY*)

A confiabilidade, como o próprio nome diz, está relacionada à garantia de que uma mensagem enviada por um usuário do ambiente virtual, com o objetivo de informar sobre uma determinada atualização feita sobre o ambiente, seja recebida por todos os demais usuários.

Porém, segundo Macedônia (1997), a confiabilidade nas comunicações de um AVD significa fazer uso de protocolos de comunicação orientados a conexão, como o *Transport Control Protocol* (TCP), que utilizam mecanismos de confirmação de recebimento e recuperação quando da ocorrência de erros ou não recebimento de algum pacote.

O problema da utilização de protocolos como o TCP é que esses mecanismos mencionados em conjunto com outros que controlam o fluxo de pacotes causam um atraso inaceitável na comunicação entre os usuários, principalmente no caso de usuários interconectados através de WAN's.

Esse atraso é inaceitável, segundo Macedônia (1997), pois em simulações de tempo real é impossível voltar no tempo. Por exemplo, quando um pacote se perde, seja por algum problema de roteamento, o computador de destino notifica o computador de envio,

possivelmente invalidando os pacotes enviados posteriormente a este que se perdeu e obrigando o computador de envio a retransmitir o pacote que se perdeu. No caso, por exemplo, desta comunicação estar transmitindo as informações de atualização do movimento de algum personagem do ambiente virtual, a atualização desse movimento, no computador de destino, ficaria seriamente comprometida.

Nesse caso, pode-se perceber que o uso de protocolos orientados a conexão não podem ser utilizados em todas as situações de comunicação entre os usuários. Torna-se necessário também a utilização de protocolos como o *User Datagram Protocol* (UDP), o qual não é orientado a conexão, conseqüentemente não garantindo que um determinado pacote seja entregue, porém agilizando a entrega dos mesmos, já que nesse protocolo não há mecanismos de controle de erro e recebimento.

Para maiores informações sobre os protocolos TCP e UDP recomenda-se uma consulta a Stevens (1995).

Mas a utilização de um protocolo não orientado a conexão não resolve a situação. É preciso analisar para cada situação qual o impacto do não recebimento de algum pacote, ou seja, como isso afetaria o desempenho normal do AVD. Justamente nessa parte é que entram algumas técnicas que permitem controlar a perda de pacotes, ou fazer com que elas não tenham um impacto desastroso sobre o AVD.

2.5 TÉCNICAS E PROTOCOLOS ASSOCIADOS AOS AVD'S

Essas questões levantadas até agora, resultante das pesquisas e projetos na área de AVD's, levaram os pesquisadores, à medida que iam amadurecendo seus projetos, a desenvolver técnicas e utilizar diferentes protocolos, sempre com o objetivo de otimizar, o máximo possível, o processo de comunicação entre os usuários de um AVD.

São essas técnicas e protocolos que serão abordados nas seções seguintes.

2.5.1 *UNICAST*, *BROADCAST* E *MULTICAST*

Segundo Stytz (1996), há três formas básicas de comunicação entre os usuários de um AVD:

a) ***unicast:*** neste método, quando um usuário do ambiente virtual faz alguma modificação, ele precisa mandar uma mensagem correspondente a esta modificação para cada um dos outros usuários do ambiente virtual. Esse tipo

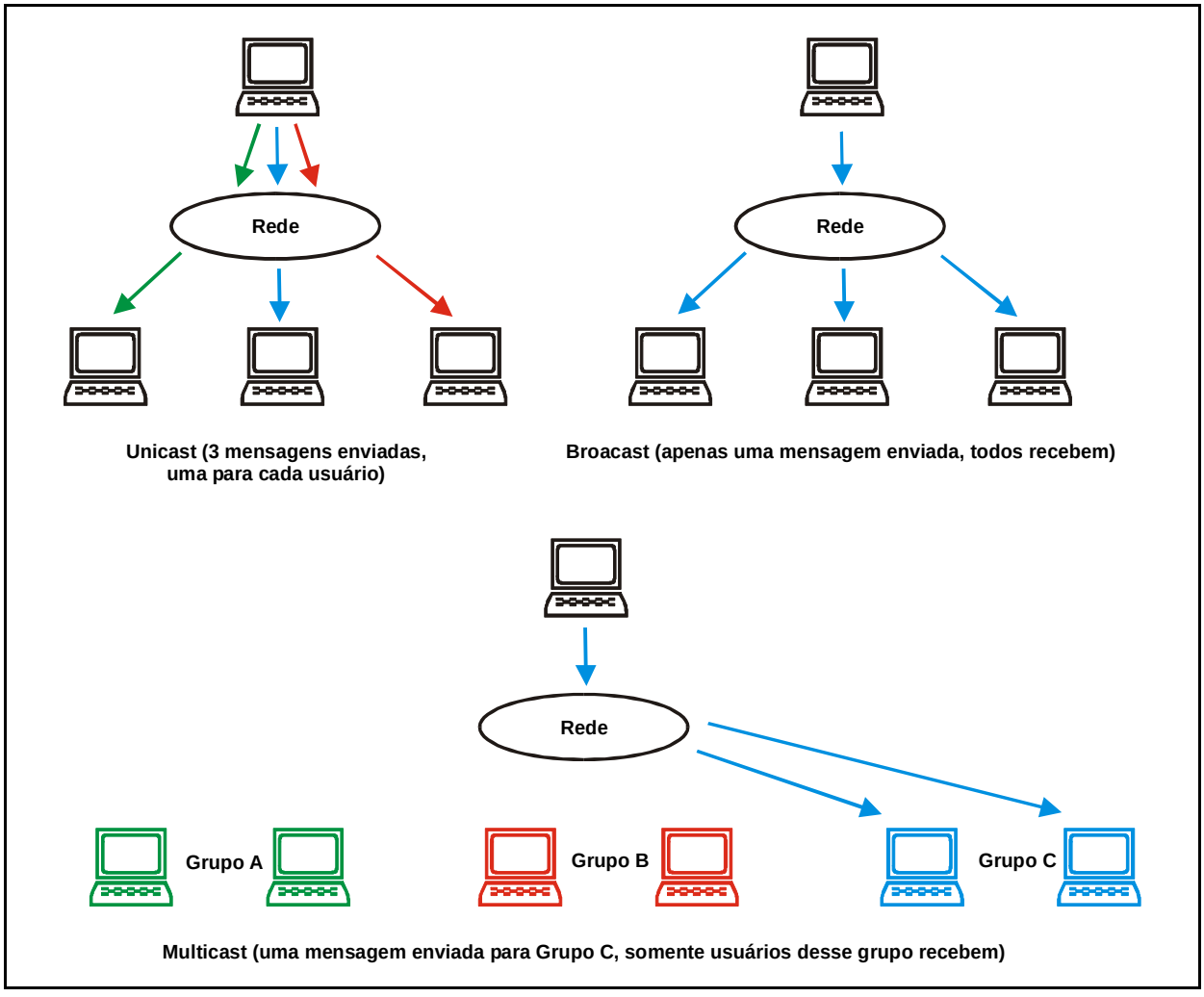
de comunicação também é conhecida como comunicação ponto-a-ponto (*point-to-point*);

b) ***broadcast:*** neste método, ao invés de ser enviada uma mensagem para cada um dos usuários quando ocorre alguma alteração no ambiente virtual, somente uma única mensagem é enviada, sendo que todos os outros usuários "ouvem" essa mensagem;

c) ***multicast:*** neste método, um conceito de canal é adotado. Cada usuário do ambiente virtual "ouve" somente em um ou mais canais do ambiente virtual. Dessa forma, quando um usuário envia uma mensagem, ele a envia para um canal específico do qual faz parte e somente os usuários que estiverem participando do mesmo canal "ouvirão" essa mensagem.

A fig. 2.3 ilustra esses três métodos de comunicação.

FIGURA 2.3 - ENVIO DE MENSAGENS UTILIZANDO *UNICAST*, *BROADCAST* E *MULTICAST*



Cada um desses métodos de comunicação apresenta suas vantagens e desvantagens, de acordo com Macedônia (1997).

No caso do *unicast*, pelo fato de que nesse método é necessário que cada usuário estabeleça uma conexão com os demais usuários, a vantagem aparente é que a sincronia das atualizações entre os usuários será elevada. Porém, para um número elevado de usuários, isso resultaria em um número muito alto de conexões, tornando difícil o gerenciamento das mesmas, principalmente com relação à criação de novas conexões para cada novo usuário que ingressasse no ambiente virtual. Outra desvantagem é o número elevado de mensagens geradas, pois cada atualização no ambiente virtual resulta no envio de uma mensagem para cada um dos outros usuários.

Como forma de solução para o problema do alto número de conexões e mensagens geradas no *unicast*, tem-se o *broadcast*. No caso do *broadcast*, não é necessário que haja uma conexão entre cada um dos usuários do ambiente virtual. Basta que todos os usuários de um mesmo ambiente "concordem" em se comunicar através de um canal em comum. Com isso, o problema do alto número de mensagens geradas é amenizado, pois cada atualização no ambiente virtual feita por um usuário resulta em somente uma mensagem, a qual será "ouvida" por todos os demais usuários. A utilização do *broadcast* é útil principalmente em AVD's sobre LAN's.

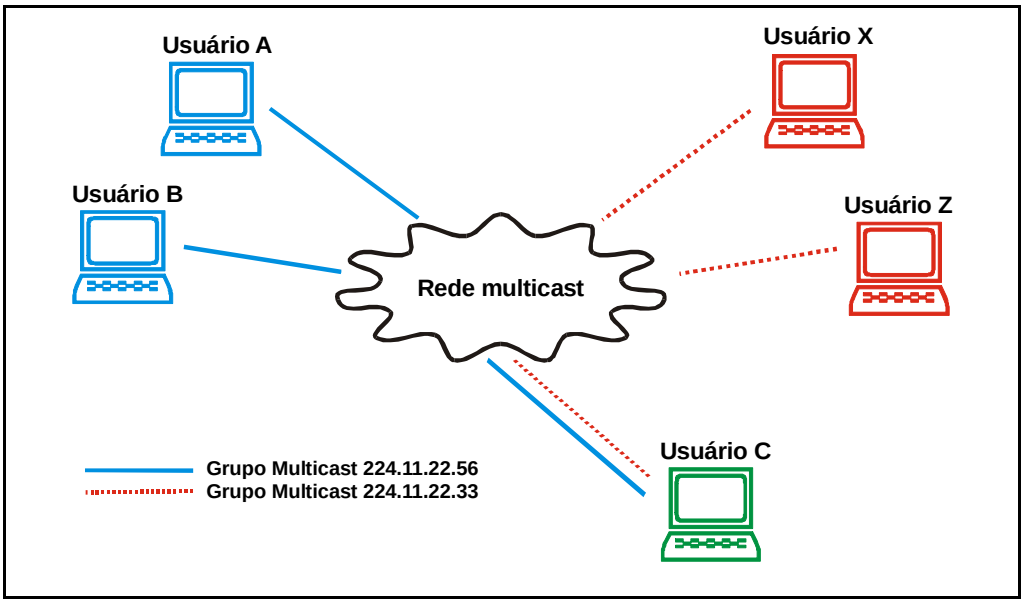
Uma das desvantagens do *broadcast*, segundo Gossweiler (1994), é que nesse método de comunicação geralmente se faz uso de UDP *broadcast*, onde as mensagens são enviadas para um endereço de *broadcast* da rede local, através do protocolo UDP. Dessa forma essas mensagens ficam confinadas à rede local, limitando o funcionamento de um AVD, que utilize UDP *broadcast*, às LAN´s. Para AVD's sobre WAN's a utilização de tal método não seria possível. Nesse caso, tem-se como opção o *multicast*.

Conforme Brutzmann (1995), no *multicast*, ou melhor, no IP *multicast* a transmissão de datagramas IP's pode ser feita para um número ilimitado de *hosts* compatíveis com *multicast*, os quais estão conectados a internet através de roteadores também compatíveis com *multicast*.

Os diferentes canais aos quais os usuários de um AVD estariam associados poderiam ser definidos através da utilização de um endereço IP classe D, correspondente aos endereços de 224.0.0.0 até 239.255.255.255. Dessa forma, um *host* pode escolher se juntar ou deixar um grupo *multicast* ao informar ao roteador, ao qual este *host* se conecta a internet, sobre quais grupos *multicast* ele tem interesse. A partir daí, esse roteador somente repassaria as mensagens dos grupos aos quais há algum *host* cadastrado. Para maiores informações sobre IP *multicast* pode-se consultar Comer (1995).

A idéia de se utilizar grupos de IP *multicast* torna-se ainda mais interessante para se dividir o ambiente virtual, ou criar as chamadas áreas de interesse (Zyda, 1995). No caso da divisão do ambiente virtual, pode-se atribuir um endereço IP *multicast* para cada uma das áreas divididas. Nesse caso, ao entrar em uma determinada área, um usuário passaria a fazer parte de um grupo *multicast* específico, enviando e recebendo somente as atualizações do ambiente virtual referentes aquele grupo. A fig. 2.4 ajuda a ilustrar um exemplo.

FIGURA 2.4 - EXEMPLO DE AGRUPAMENTO DE USUÁRIOS ATRAVÉS DE *MULTICAST*



Fonte: Adaptado de Zyda (1995).

Nesse exemplo, há cinco usuários distintos fazendo parte de um ambiente virtual. A área do ambiente virtual na qual se encontram os usuários **A** e **B** corresponde ao grupo *multicast* 224.11.22.56 e a área na qual estão os usuários **X** e **Z** corresponde ao grupo *multicast* 224.11.22.33. Já o usuário **C** se encontra numa área comum a essas duas áreas.

Qualquer atualização feita pelo usuário **A** no ambiente virtual, ao contrário de ser enviada para todos os outros usuários do ambiente virtual, fica confinada ao grupo *multicast* correspondente a área em que ele se encontra. Dessa forma, somente os usuários **B** e **C** são comunicados dessa atualização. Da mesma forma que qualquer alteração feita, por exemplo, pelo usuário **Z** somente será comunicada aos usuários **X** e **C**. No caso de ser o usuário **C** a fazer alguma alteração, daí sim todos serão comunicados, tendo-se em vista que ele ocupa uma área comum a todos, estando associado aos dois grupos *multicast*.

Dessa forma, as questões levantadas na seção 2.2, que tratam da utilização da largura de banda, são resolvidas de acordo com a escolha certa entre esses três métodos de

comunicação, estando fortemente ligada a abrangência (LAN ou WAN) e ao número de usuários que o ambiente virtual terá.

2.5.2 ALGORITMOS DE *DEAD RECKONING*

Os algoritmos de *Dead Reckoning*, no contexto de AVD's, são utilizados com o objetivo de reduzir substancialmente o número de mensagens geradas por cada um dos usuários do ambiente virtual. Numa situação normal, sem a utilização de algoritmos dessa natureza, quando um usuário, por exemplo, movimenta seu personagem no ambiente virtual, a cada nova posição ocupada pelo personagem, uma nova mensagem precisa ser enviada para comunicar aos demais usuários dessa mudança de posição. Numa situação onde vários usuários, que ocupam o ambiente virtual, estão se movimentando simultaneamente, fica fácil perceber a quantidade de mensagens geradas para que todos os usuários se mantenham atualizados acerca da posição dos demais.

Basicamente, segundo Gossweiler (1994), os algoritmos de *Dead Reckoning* fazem uma abordagem diferente com relação a atualização de posição. Quando um usuário se movimenta no ambiente virtual, ao contrário de serem enviadas várias mensagens informando as posições que o usuário vai ocupando, apenas, por exemplo, a direção e velocidade do mesmo são informadas. Estando de posse dessa informação, cada computador conectado ao ambiente virtual é capaz de calcular a nova posição que o usuário vai ocupar.

Dessa forma, cria-se um conceito de objeto "vivo" e "fantasma". No caso, por exemplo, de um usuário **A** no computador **A** começar a movimentar algum objeto, esse objeto no computador **A** é considerado o objeto vivo. Nos demais computadores esse objeto é considerado um objeto fantasma.

Ao iniciar a movimentação do objeto, esse usuário envia as informações de direção e velocidade, referentes a esse movimento, para os demais usuários. Esses, ao receberem essas informações, através do algoritmo de *Dead Reckoning*, conseguem determinar o movimento que o objeto no computador **A** está realizando, possibilitando que o mesmo seja representado em suas respectivas representações do ambiente virtual.

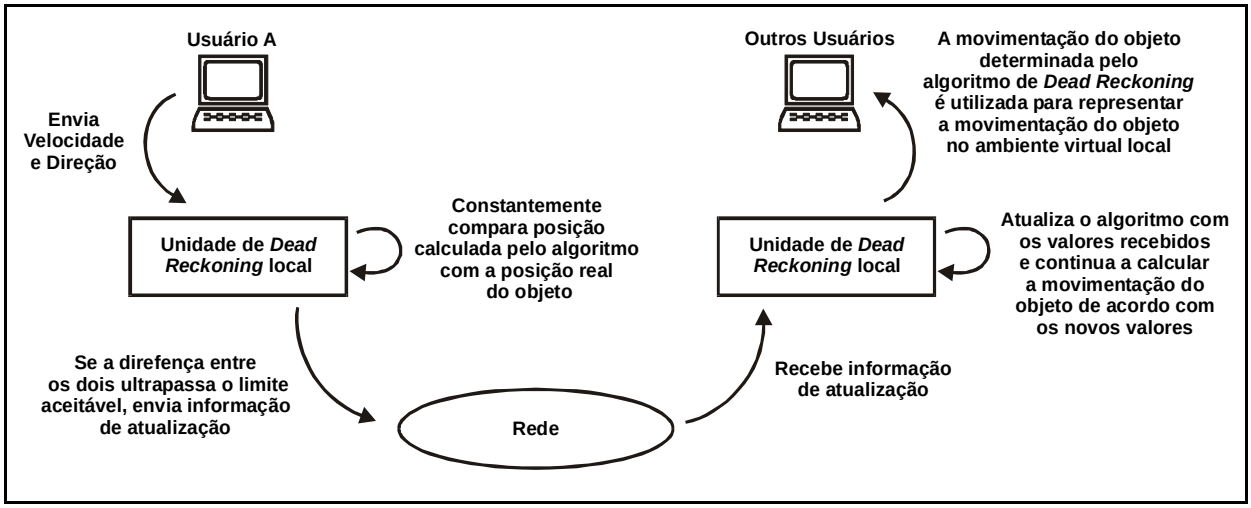
Porém, o algoritmo de *Dead Reckoning* também é executado no computador **A** com o objetivo de determinar como a movimentação do objeto está sendo feita nos demais usuários. A movimentação real do objeto no computador **A** é constantemente comparada com a movimentação determinada pelo algoritmo de *Dead Reckoning* local. Caso a movimentação

do objeto no computador **A** fuja do determinado pelo algoritmo de *Dead Reckoning* local, daí sim uma mensagem é enviada para todos os outros usuários, atualizando-os acerca da nova direção e velocidade. Os algoritmos de *Dead Reckoning* nos demais usuários passam, então, a determinar o movimento do objeto nos demais usuários com as informações atualizadas.

A mínima diferença entre a movimentação real do objeto e a determinada pelo algoritmo de *Dead Reckoning* não significa que isso implica numa atualização obrigatória. Pode-se especificar um intervalo aceitável de desvio. Enquanto esse desvio gerado pelo algoritmo de *Dead Reckoning* estiver dentro desse intervalo, não há a necessidade de se enviar uma mensagem de atualização. Somente quando o desvio ultrapassa esse intervalo aceitável é que a mensagem de atualização é enviada.

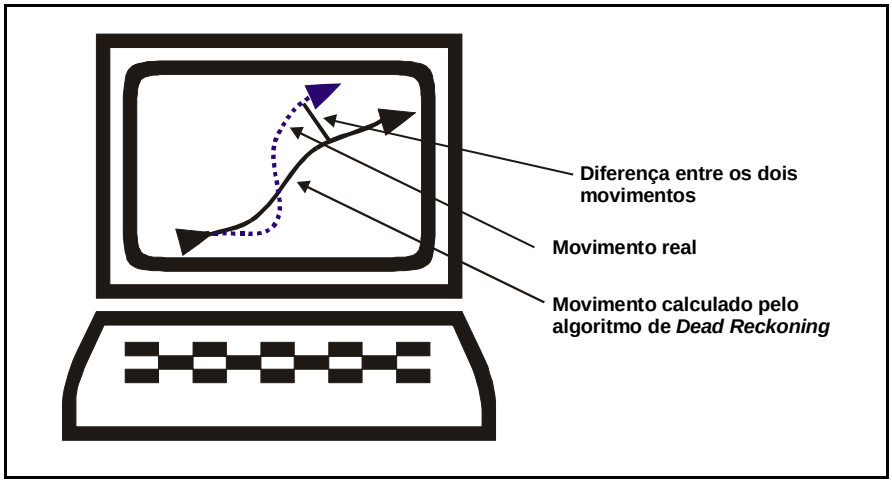
A fig. 2.5 ilustra o processo de atualização do ambiente virtual para os demais usuários, utilizando-se o algoritmo de *Dead Reckoning*.

FIGURA 2.5 - PROCESSO DE ATUALIZAÇÃO DO AMBIENTE VIRTUAL UTILIZANDO *DEAD RECKONING*



A fig. 2.6 mostra um exemplo de diferença gerada entre o movimento real de um objeto e o movimento determinado pelo algoritmo de *Dead Reckoning*.

FIGURA 2.6 - DIFERENÇA DE MOVIMENTO REAL E CALCULADO PELO ALGORITMO DE *DEAD RECKONING*



Fonte: Adaptado de Gossweiler (1994).

Com relação aos problemas descritos nas seções anteriores, relacionados com a rede, os algoritmos de *Dead Reckoning* auxiliam a contornar as questões de utilização da largura de banda e latência (seções 2.2 e 2.3). A utilização mais racional da largura de banda é evidente pois o número de mensagens de atualização geradas reduz consideravelmente com a utilização de algoritmos de *Dead Reckoning*.

Já em relação a latência, os algoritmos de *Dead Reckoning* ajudam no sentido de manter uma movimentação constante dos objetos quando ocorre algum atraso maior. Como esse algoritmo é executado em cada um dos usuários, sendo responsável por determinar a movimentação dos objetos, quando alguma mensagem de atualização sofre algum tipo de atraso significativo, esses objetos não interrompem sua movimentação, já que quem determina o seu movimento é o algoritmo de *Dead Reckoning* local. Logicamente, esse movimento pode não estar totalmente condizente com o movimento real, porém, assim que a mensagem de atualização for recebida, o movimento será corrigido.

Os algoritmos de *Dead Reckoning* são o coração dos mecanismos mais populares de simulação, estando fortemente ligados a protocolos como o *Distributed Interactive Simulation* (DIS), que será abordado na próxima seção e estando presente em AVD's como o SIMNET (Shaw 1993, Locke 1994) e NPSNET (Npsnet, 2000).

2.5.3 *DISTRIBUTED INTERACTION SIMULATION* (DIS)

A origem do DIS está fortemente ligada às experiências obtidas no desenvolvimento do SIMNET (Gossweiler, 1994). Também, o SIMNET e DIS representam o maior e mais antigo esforço na área de desenvolvimento de ambientes virtuais distribuídos (Macedônia,

1995). Portanto, torna-se interessante dar uma visão geral a respeito da origem e desenvolvimento do SIMNET, para num passo seguinte ser abordado o DIS propriamente dito. Essa visão, baseada numa compilação de Macedônia (1995), será exposta a seguir.

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

A forma como o SIMNET foi desenvolvido está baseada em 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* são utilizados para reduzir o tráfego de mensagens de atualização.

Cada um desses princípios ou aspectos do SIMNET são utilizados justamente com o objetivo de contornar ou amenizar os vários problemas inerentes aos AVD's, os quais foram abordados nas seções anteriores.

Uma das características mais importantes, que está fortemente ligada à definição futura do DIS, é o protocolo de simulação utilizado no SIMNET. Esse protocolo consiste de um determinado número de *Protocol Data Units* (PDU's) responsáveis por informar acerca dos estados dos objetos do ambiente virtual ou da ocorrência de eventos. 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.

A tabela 2.1 mostra a estrutura típica de um PDU dessa natureza.

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 Macedônia (1995).

Porém, 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.

A preocupação com essas possíveis divergências entre padrões, que poderiam acontecer, foi que levou a especificação do protocolo DIS.

Segundo Macedônia (1995), o protocolo DIS é um grupo de padrões, definido pelo Departamento de Defesa dos Estados Unidos e indústrias interessadas, 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".

Uma das principais características herdadas do SIMNET no DIS foi a utilização de PDU's para a comunicação de estados e eventos no ambiente virtual. No seu padrão original definido na IEEE 1278-1993 (IEEE, 1993), existem vinte e sete tipos diferentes de PDU's. Um tipo essencial de PDU é o *Entity State* PDU (ESPDU), o qual é utilizado para enviar informações sobre o estado atual de um objeto no ambiente virtual, incluindo informações

como sua posição, orientação, velocidade e aparência. Macedônia (1995) cita outros exemplos de PDU's como os PDU's de disparo que contém informações, por exemplo, sobre o tipo da arma que ocasionou o disparo e PDU's de detonação que informam quando algum armamento de caráter explosivo é detonado ou quando algum objeto é destruído.

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 Macedônia (1995).

Desde a sua primeira versão de 1993, o padrão IEEE 1278-1993 já passou por uma revisão resultando na 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).

2.5.4 HEARTBEATS

Além da principal utilidade dos PDU's de manter todos os usuários informados a respeito de eventos ocorridos no ambiente virtual, eles também são utilizados num processo chamado de "*heartbeating*". Esse processo força cada objeto do mundo virtual a enviar uma mensagem de atualização, agora chamada de PDU, num intervalo de tempo determinado, por exemplo de cinco em cinco segundos.

No caso de ambientes virtuais que utilizem um sistema de comunicação não confiável, esse processo ajuda a notificar o estado dos objetos do mundo virtual no caso de algum usuário não ter recebido um PDU de atualização anterior. Também, o *heartbeating* é importante para novos usuários que estejam entrando no ambiente virtual, já que é a partir desses PDU's que ele poderá construir sua própria representação do ambiente virtual.

O processo de *hearbeating* também se apresenta como uma solução no caso de objetos com uma freqüência de atualização muito baixa. Imagine um veículo de batalha que tenha sofrido um dano muito significativo, o qual o impossibilitasse de continuar a se mover. A partir desse momento, já que esse veículo não se movimenta mais, ele conseqüentemente deixaria de enviar PDU's de atualização, porém a sua representação no ambiente virtual deveria continuar para todos os usuários, mesmo que fosse a de um veículo estático.

No caso dos usuários que estivessem participando do ambiente virtual no momento da parada do veículo, isso não representaria problemas, já que esses simplesmente parariam de movimentar suas próprias representações do veículo. Porém, para usuários que entrassem no ambiente virtual após a parada do veículo, já que esse não enviaria mais PDU's de atualização, esses novos usuários nem sequer teriam idéia da existência desse veículo, conseqüentemente não o representando no seu ambiente virtual.

Com a utilização do *heartbeating* esse problema é solucionado pois mesmo que um objeto não necessite enviar PDU's de atualização, por não haver mudanças no seu estado, ele obrigatoriamente, no intervalo definido para o *heartbeating*, envia um PDU informando acerca do seu estado atual.

Porém, a utilização desse método, segundo Macedônia (1995), não representa uma das melhores soluções já que a obrigatoriedade de cada objeto do ambiente virtual em enviar um PDU 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 das soluções seria determinar um computador central o qual ficaria responsável por manter uma representação completa do ambiente virtual. Esse computador seria uma fonte de pesquisa para cada novo usuário que entrasse no ambiente virtual, fornecendo as informações necessárias para que esse novo usuário montasse a sua própria representação do ambiente virtual. Após esse processo inicial, esse usuário passaria a se comportar como os demais usuários, não ficando mais dependente desse computador.

Porém, manter um computador central para esse serviço é indesejável, já que uma possível falha do mesmo poderia comprometer a entrada de novos usuários no ambiente virtual. Dessa forma, poder-se-ia determinar que o computador responsável por esse serviço seria o do usuário “mais antigo” do ambiente virtual. Isso implica em cada usuário do ambiente virtual manter a informação de quanto tempo está no ambiente virtual e se auto eleger para tal função caso não detecte nenhum usuário a mais tempo no ambiente virtual do que o seu.

Nesse capítulo, procurou-se abordar de forma geral os aspectos inerentes aos AVD’s. Técnicas que consideram esses aspectos também foram vistas, principalmente aquelas que procuram contornar ou resolver os problemas que surgem ao se considerar esses aspectos. Também foi abordado o protocolo DIS, dando uma visão geral de sua estrutura e utilidade.

O próximo capítulo apresenta uma visão geral da API do Java3D, a qual será utilizada para a construção do ambiente virtual.

3 JAVA3D

A popularização e o grande desenvolvimento da Internet fez com que um ponto importante passasse a ser considerado ao se desenvolver um software: a sua integração com a Internet. Porém, é importante lembrar que cada um desses usuários que utilizam a Internet possuem o seu próprio tipo de computador (Macintosh, IBM-PC compatíveis, RISC´s, etc) e também o seu próprio sistema operacional (Mac OS, Windows, variantes do Unix, OS 2, etc). Portanto a interoperabilidade do software também se tornou algo muito importante. E além desses dois pontos, existe a constante procura pelo desenvolvimento de softwares com interfaces cada vez mais amigáveis, o que implica, geralmente, na utilização de recursos multimídia tais como áudio, vídeo e gráficos 3D.

Com relação ao problema da interoperabilidade, a linguagem de programação Java, com o seu modelo “escreva uma vez, execute em qualquer lugar”, trouxe uma série de facilidades com relação ao desenvolvimento de programas que necessitem de interoperabilidade. Com relação aos aspectos de interconexão com a internet e utilização de recursos multimídia, a equipe JavaSoft, em conjunto com outras empresas, que são responsáveis pelo desenvolvimento da linguagem Java, passaram a desenvolver um conjunto de API’s que procurasse facilitar a integração desses aspectos no desenvolvimento de softwares. Esse conjunto de API´s ficou conhecido como “*The Java Media Family*” (Sun, 2000b).

Basicamente, esse conjunto é formando pelas seguintes API´s, segundo Sun (2000b):

a) ***Java 2D API:*** provê um conjunto de classes para se trabalhar com imagens (rasters ou vetoriais), linhas, cores, transformações, composição, etc;
b) ***Java Media Framework Specification:*** especifica uma arquitetura unificada, protocolo de mensagens e interface de programação para jogos e conferências multimídia. Inclui três API´s separadas (*Java Media Player*, *Java Media Capture* e *Java Media Conference*);
c) ***Java Collaboration API:*** oferece um conjunto de classes que permite a interação e comunicação entre vários usuários interconectados;
d) ***Java Speech API:*** provê um conjunto de classes utilizadas para reconhecimento e síntese de voz;
e) ***Java Telephony API:*** integra telefones e computadores, oferecendo mecanismos para controle de chamadas;

f) ***Java Animation:*** utilizada para movimentação e transformação de objetos em 2D;

g) ***Java 3D API:*** conjunto de classes para a construção e manipulação de imagens em 3D.

Como se pode observar, entre as API´s que compõem o *Java Media Family*, encontra-se a do Java3D, justamente voltada para a construção e manipulação de imagens em 3D.

Assim como outras bibliotecas gráficas, a API do Java3D é uma interface para se desenvolver softwares que tenham por objetivo exibir cenários e gráficos em 3D, permitindo que haja interação com os mesmos. Basicamente, a API provê um conjunto de classes que permite criar e manipular estruturas gráficas em 3D, fazendo com que o processo de implementação de softwares que necessitem de tais características, seja em "alto nível".

Cada objeto que fará parte do mundo virtual a ser criado, será derivado de uma das classes do Java3D. A API, que no período em que foi escrito esse trabalho se encontrava na versão 1.2.1 beta 2, possui em média mais de 100 classes que estão incluídas no pacote `javax.media.j3d` e que constituem o núcleo do Java3D. Outros pacotes, como o `com.sun.j3d.utils`, `java.awt` e `javax.vecmath` também auxiliam na criação e manipulação de objetos tridimensionais. O pacote `com.sun.j3d.utils` fornece classes que simplificam bastante a criação de objetos simples, o pacote `java.awt` fornece as classes básicas para a construção de interfaces gráficas e o pacote `javax.vecmath` fornece as classes que permitem realizar as operações matemáticas sobre pontos, vetores, matrizes e outros objetos matemáticos (Sun, 2000b).

Nas seções seguintes desse capítulo é apresentada a descrição do grafo de cena utilizado para especificar os ambientes virtuais criados através da API do Java3D e na seqüência é feita uma abordagem das principais classes da API do Java3D utilizadas para a construção de ambientes virtuais simples, para a manipulação de objetos em 3D e a interação com os mesmos. O conteúdo dessas seções está baseado em Nardeau (1999), Sowizral (2000) e Sun (2000abc).

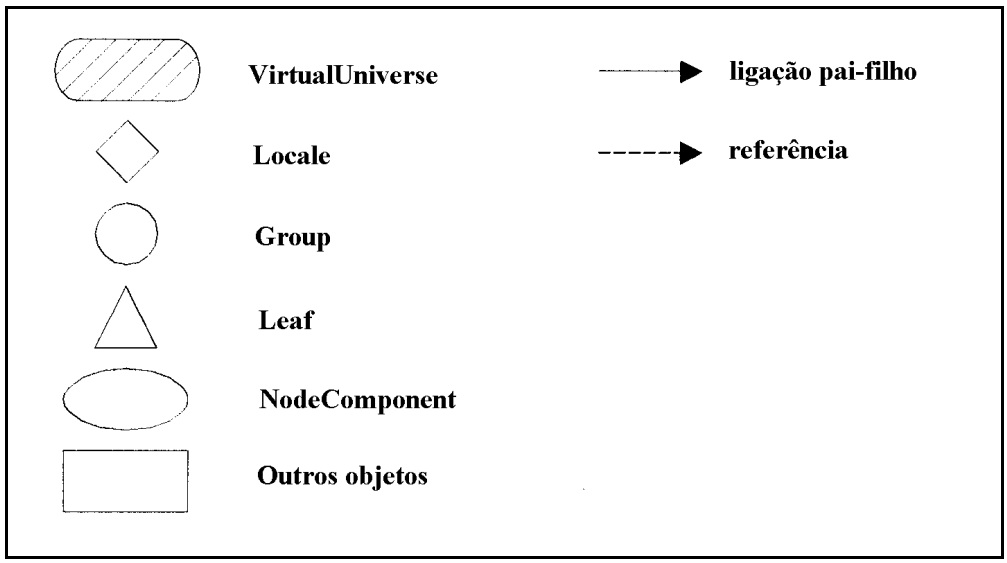
3.1 O GRAFO DE CENA

O grafo de cena, dentro do contexto do Java3D, é uma estrutura de dados composta por nós e arcos. Cada nó do grafo representa uma instância de uma classe da API do Java3D e cada arco a relação que existe entre os objetos instanciados dessas classes. De forma geral, o

grafo de cena serve como uma especificação ou documentação que permite representar graficamente as informações referentes a geometria, sons, luzes, localização, orientação e aparência dos objetos do ambiente virtual.

A fig. 3.1 mostra os principais símbolos utilizados para a construção de um grafo de cena.

FIGURA 3.1 - SÍMBOLOS DE UM GRAFO DE CENA



Fonte: Adaptado de Sun (2000c)

Em seguida, tem-se uma breve explicação de cada componente do grafo de cena:

a) **`VirtualUniverse`/`Locale`:** cada grafo de cena possui como nó raiz um único `VirtualUniverse` e esse por sua vez, geralmente possui um objeto do tipo `Locale`, que fornece um ponto de referência dentro do mundo virtual. *`VirtualUniverse`* pode ser a representação da instância de um objeto da classe `VirtualUniverse` e `Locale` a representação da instância de um objeto da classe `Locale`;

b) **`Group`*:*** os nós do tipo `Group` são representações de instâncias de classes como a `BranchGroup` e `TransformGroup`, derivadas da própria classe `Group`. Essas classes são responsáveis por controlar as informações de localização e orientação dos objetos visuais;

c) **`Leaf`:** já os nós do tipo `Leaf` são representações de instâncias de classes como a `Shape3D`, `Ligth`, `Behavior` e `Sound`, derivadas da própria classe `Leaf`. Essas classes são responsáveis por controlar as informações de forma, som, luzes e comportamento dos objetos visuais;

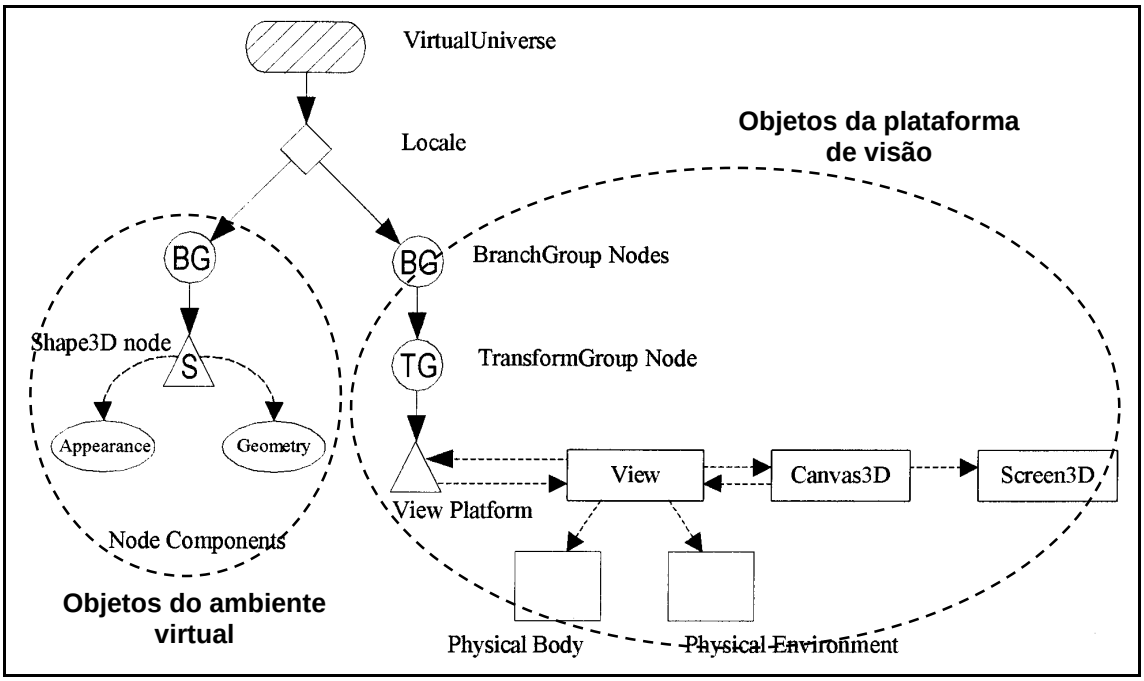
d) **`NodeComponent:`** os nós do tipo `NodeComponent` são representações de instâncias de subclasses da classe `NodeComponent` e especificam as características de geometria, aparência, textura e material dos objetos visuais;

e) **seta cheia:** demonstram a relação entre os objetos dentro do grafo de cena;

f) **seta tracejada:** referenciam objetos a `NodeComponents`, sendo estes os únicos objetos que podem ser referenciados por mais de um objeto.

A fig. 3.2 mostra uma visão geral de um típico grafo de cena.

FIGURA 3.2 - EXEMPLO DE UM GRAFO DE CENA



Fonte: (Sun, 2000c)

O grafo de cena, dessa forma, é construído ao se acrescentar nós que representam cada um dos objetos visuais que serão adicionados ao ambiente virtual, bem como os objetos responsáveis pelas suas características como comportamento, aparência, geometria, localização, transformações, etc.

As setas cheias são utilizadas para indicar uma relação direta entre nó pai e nó filho, não sendo permitido que um nó filho tenha mais que um nó pai. As setas tracejadas são utilizadas para fazer referência a outros nós que representam instâncias de objetos derivados da classe `NodeComponent`.

Os nós derivados dessa classe, responsáveis por representar instâncias de classes que definem a geometria, aparência, textura e material dos objetos visuais, podem possuir mais

que um nó pai. Isto porque essas características podem ser comuns a mais de um objeto visual do ambiente virtual.

Como pode ser observado na fig. 3.2, além de ser utilizado para representar os objetos que compõem o mundo virtual, o grafo de cena também pode ser utilizado para representar os objetos que controlam o mecanismo de visão do mundo virtual. Dessa forma, geralmente um grafo de cena possui um ou mais `BranchGroup's` de conteúdo, que especificam as informações dos objetos do mundo virtual, e um ou mais `BranchGroup's` de visão que especificam as informações das visões do mundo virtual. Tendo em vista que neste trabalho será utilizada a classe `SimpleUniverse`, que abstrai a construção do `BranchGroup` de visão, os detalhes inerentes a ele não serão abordados.

3.2 CLASSES BÁSICAS PARA A CONSTRUÇÃO RÁPIDA DE MUNDOS VIRTUAIS

A API completa do Java3D permite construir mundos virtuais com um grau de complexidade bastante elevado (Sun, 2000c). Isso inclui desde a definição da geometria do objeto, aparência (cor, textura, material), transformações sobre o objeto (escala, rotação e translação), definição de comportamento, animações, interação com o usuário, luzes, etc. Porém, para a construção de mundos virtuais simples, a API, juntamente com o pacote `javax.media.j3d.utils`, fornece classes que simplificam bastante esse processo.

3.2.1 UM MUNDO VIRTUAL SIMPLES

A construção de um mundo virtual normal, utilizando o Java3D, pode ser resumida nos seguintes passos:

a) instanciar um objeto da classe `VirtualUniverse`;
b) instanciar um objeto da classe `Locale`, adicionando-o ao objeto da classe `VirtualUniverse`;
c) construir um `BranchGroup` de visão;
d) instanciar um objeto da classe `View`;
e) instanciar um objeto da classe `ViewPlatform`;
f) instanciar um objeto da classe `PhysicalBody`;
g) instanciar um objeto da classe `PhysicalEnvironment`;

h) adicionar o objeto da classe `ViewPlatform`, `PhysicalBody`, `PhysicalEnvironment` e `Canvas3D` ao objeto da classe `View`;

i) construir um `BranchGroup` de conteúdo;

j) criar e adicionar objetos visuais ao `BranchGroup` de conteúdo;

k) compilar o `BranchGroup`;

l) adicionar os `BranchGroup's` de visão e conteúdo ao objeto da classe `Locale`.

Os passos correspondentes às letras de "c" até "h" são utilizados para instanciar e construir a plataforma de visão de forma manual. Os objetos das classes `View`, `ViewPlatform`, `PhysicalBody` e `PhysicalEnvironment` permitem controlar aspectos relacionados à plataforma de visão.

Porém, em se tratando de um mundo virtual simples, o pacote `javax.media.j3d.utils`, fornece uma das várias classes que simplificam a programação com o Java3D, abstraindo os passos de "b" a "h", criando um mundo virtual com um `BranchGroup` de visão padrão. Essa classe é a `SimpleUniverse`.

O quadro 3.1 mostra os principais construtores e métodos dessa classe.

QUADRO 3.1 - CONSTRUTORES E MÉTODOS DA CLASSE `SIMPLEUNIVERSE`

Construtores da classe SimpleUniverse:

SimpleUniverse() – cria um universo virtual simples

SimpleUniverse(Canvas3D canvas3D) – cria um universo virtual com uma referência a um objeto da classe Canvas3D, no qual será renderizado a imagem do mundo virtual.

Métodos da classe SimpleUniverse:

void addBranchGraph (BranchGroup bg) – utilizado para adicionar objetos da classe BranchGroup ao objeto da classe Locale instanciado pelo objeto SimpleUniverse.

ViewingPlatform getViewingPlatform() – utilizado para acessar o objeto da classe ViewingPlatform que foi instanciado pelo objeto SimpleUniverse. Geralmente esse método é utilizado em conjunto com o método setNominalViewingPlatform() para ajustar a localização da posição de visão.

Dessa forma, a construção de um mundo virtual com o uso da classe `SimpleUniverse`, simplifica o processo nos seguintes passos:

a) instanciar um objeto da classe `Canvas3D`;

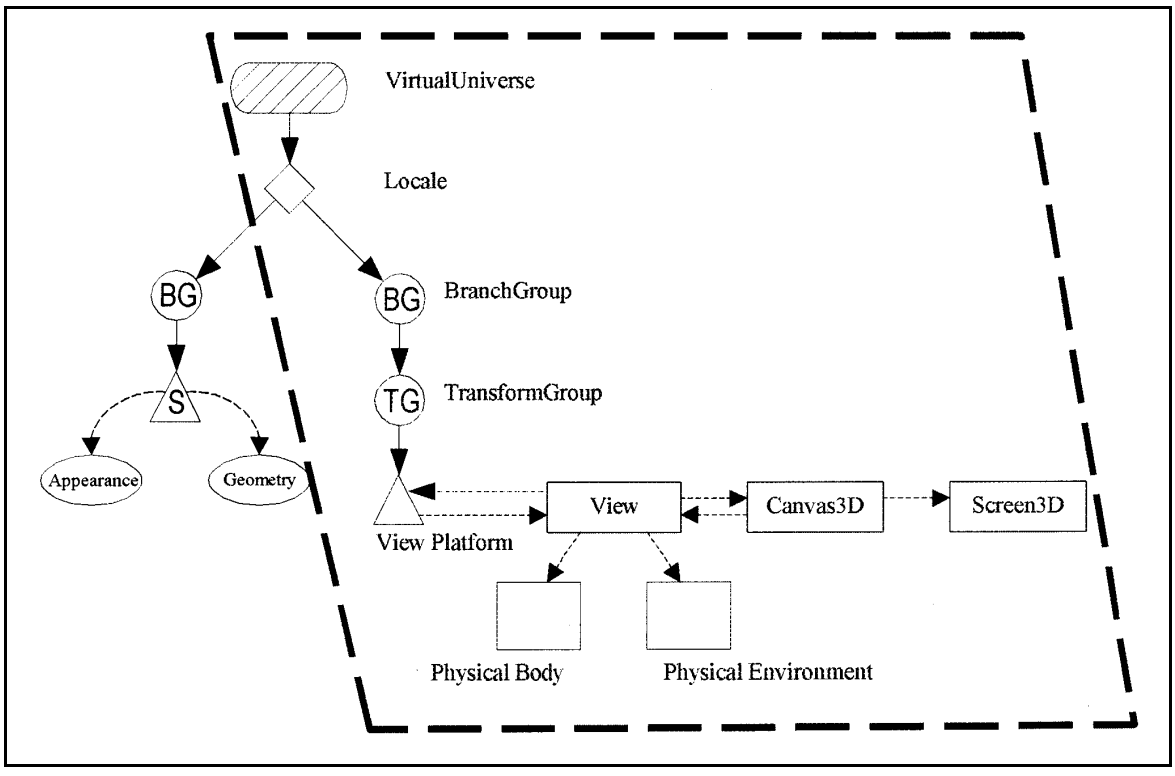
b) instanciar um objeto do tipo `SimpleUniverse`, o qual fará referência ao objeto `Canvas3D`;

c) otimizar o objeto `SimpleUniverse`;
d) construir um `BranchGroup` de conteúdo;
e) criar e adicionar objetos visuais ao `BranchGroup` de conteúdo;
f) compilar o `BranchGroup`;
g) adicionar o `BranchGroup` de conteúdo ao objeto da classe `SimpleUniverse`.

Como se pode notar, a utilização da classe `SimpleUniverse` abstrai toda a preocupação do programador com relação a criação do `BranchGroup` responsável pela visualização do mundo virtual, podendo esse se preocupar somente com a criação do `BranchGroup` de conteúdo, que diz respeito aos objetos que popularão o mundo virtual.

A linha tracejada da fig. 3.3 mostra toda a parte criada automaticamente ao se instanciar um objeto da classe `SimpleUniverse`.

FIGURA 3.3 - ABSTRAÇÃO DO BRANCHGROUP DE VISÃO NO SIMPLEUNIVERSE



Fonte: (Sun, 2000c)

3.2.2 FORMAS GEOMÉTRICAS BÁSICAS

Após ter sido criado um objeto da classe `SimpleUniverse`, o próximo passo do programador é começar a construir o `BranchGroup` de conteúdo. Para simplificar esse

passo, existem algumas classes dentro do pacote `com.sun.j3d.utils`, que auxiliam na construção de formas geométricas básicas como caixas, cones, cilindros e esferas. A construção dessas formas geométricas envolve, justamente, as classes `Box`, `Cone`, `Cylinder` e `Sphere`.

O quadro 3.2 mostra os principais construtores e métodos dessas classes.

QUADRO 3.2 - CONSTRUTORES E MÉTODOS DAS CLASSES `BOX`, `CONE`, `CYLINDER` E SPHERE

Construtores da classe Box

Box() – cria um objeto visual com o formato de uma caixa padrão de 2 metros de lagura, altura e pronfundidade, com sua origem nas coordenadas (0, 0, 0)

Box (float dimx, float dimy, float dimz, Appearance aparencia) – cria um objeto visual com o formato de uma caixa nas dimensões dimx, dimy e dimz informadas, atribuindo a aparência definida (a classe Appearance será descrita adiante)

Construtores da classe Cone

Cone() – cria um objeto visual com o formato de um cone com raio de 1 metro e altura de 2 metros.

Cone (float raio, float altura) – cria um objeto visual com o formato de um cone com o raio e altura informados

Construtores da classe Cylinder

Cylinder() – cria um objeto visual com o formato de um cilindro com raio de 1 metro e altura de 2 metros

Cylinder (float raio, float altura) – cria um objeto visual com o formato de um cilindro com o raio e altura informados

Cylinder (float raio, float altura, Appearance aparencia) – cria um objeto visual com o formato de um cilindro com o raio, altura e aparência informados.

Construtores da classe Sphere

Sphere() – cria um objeto visual com o formato de uma esfera com o raio de 1 metro.

Sphere (float raio) – cria um objeto visual com o formato de uma esfera e com o raio informado.

Sphere (float raio, Appearance aparencia) – cria um objeto visual com o formato de uma esfera e com o raio e aparência informados.

Métodos da classe Box, Cone, Cylinder e Sphere

Shape3D getShape (int id) – retorna um objeto da classe Shape3D que referencia uma das faces que constituem o objeto visual (por exemplo, um cilindro é formato por 3 faces, a face que forma a base, a face que forma o topo e a face lateral do cilindro), sendo que cada uma dessas faces possui a sua própria geometria.

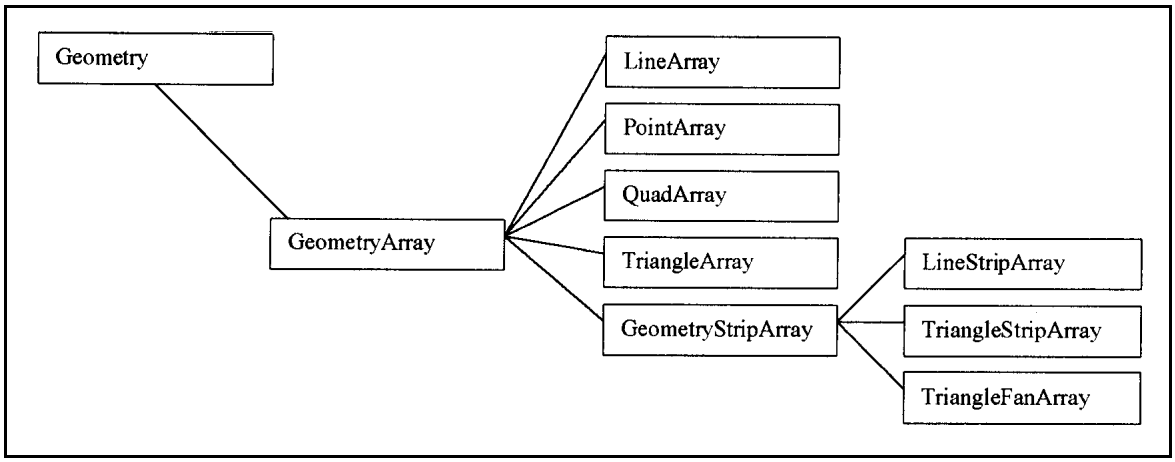
void setAppearance (Appearance aparencia) – associa um objeto da classe Appearance ao objeto visual.

Além dessas classes, há também uma classe chamada `ColorCube`, que permite criar um objeto visual no formato de um cubo com um tamanho padrão de 2 metros. Uma das diferenças da classe `ColorCube` e das classes `Box`, `Cone`, `Cylinder` e `Sphere` é que a classe `ColorCube` já tem uma aparência pré-definida, fazendo com que um objeto instanciado dessa classe apareça como um cubo onde cada uma das faces possui uma cor diferente. Já os objetos da classe `Box`, `Cone`, `Cylinder` e `Sphere` não possuem uma aparência definida. Com isso, a instância de objetos dessa classe resulta na criação de objetos visuais com todas as faces na cor branca. Porém, é possível associar uma aparência para cada uma das faces que formam esses objetos, possibilitando dessa forma definir cores e texturas individualmente para cada face.

Essas classes apresentadas são as que permitem construir formas geométricas da maneira mais rápida e simples possível. O Java3D possui um conjunto de classes bastante interessante que permite construir as mais variadas formas de objetos tridimensionais.

A fig. 3.4 mostra a estrutura das classes e subclasses que permitem construir objetos visuais de formas geométricas mais avançadas.

FIGURA 3.4 - CLASSES PARA A CONSTRUÇÃO DE FORMAS GEOMÉTRICAS AVANÇADAS



Fonte: (Sun, 2000c)

Sun (2000c) e Sowizral (2000) podem ser consultados para maiores detalhes na utilização dessas classes.

Utilizando-se dessas classes básicas já é possível criar um mundo virtual e colocar alguns objetos visuais nele. O quadro 3.3 apresenta o código fonte de uma aplicação que cria um mundo virtual baseado na classe `SimpleUniverse` e adiciona um objeto visual do tipo cone.

QUADRO 3.3 - EXEMPLO DE UTILIZAÇÃO DA CLASSE SIMPLEUNIVERSE E CONE

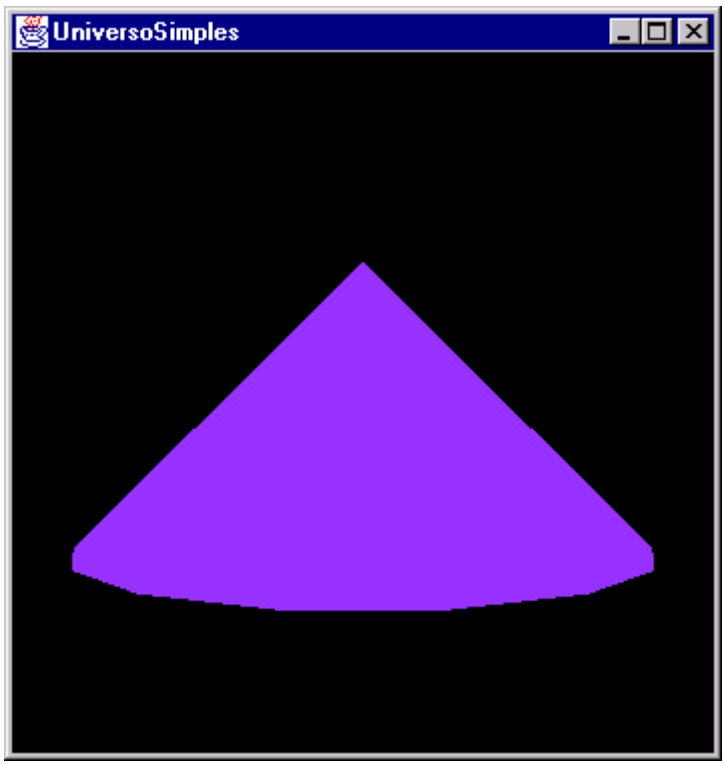
```
// Exemplo de SimpleUniverse e Cone

import java.applet.Applet; //fornece a classe Applet
// pacote awt fornece classes para
// construir e controlar elementos da interface gráfica
import java.awt.BorderLayout;
import java.awt.GraphicsConfiguration;
// pacote j3d.utils fornece classes que facilitam a construção
// de ambientes virtuais (SimpleUniverse, Cone, Box, Sphere, etc)
import com.sun.j3d.utils.applet.MainFrame;
import com.sun.j3d.utils.universe.*;
import com.sun.j3d.utils.geometry.*;
// pacote javax.media.j3d fornece as classes em geral do Java3D
import javax.media.j3d.*;

public class UniversoSimples extends Applet {
    public BranchGroup criaConteudoBrachGraph() {
  // Cria o BranchGroup  de conteúdo
  BranchGroup bg = new BranchGroup();
  // Cria um objeto visual do tipo cone
  Cone objCone = new Cone(0.8f, 0.8f);
  // Cria um objeto para setar uma cor para a aparência do cone
  ColoringAttributes cor =
     new ColoringAttributes(0.6f, 0.2f, 1.0f, 0);
  // Cria um objeto para especificar uma aparência para o cone
  Appearance aparencia = new Appearance();
  // Atribui a cor a aparência
  aparencia.setColoringAttributes(cor);
  // Atribui a aparência ao cone
  objCone.setAppearance(aparencia);
  // Adiciona o cone ao BranchGroup
  bg.addChild(objCone);

  // Chama o método compile() do BranchGroup com o
  // objetivo de otimizar os objetos do mundo virtual para
  // agiliar o processo de renderização. Entre as ações desse
  // método, estão a fusão de objetos de uma mesma num único
  // objeto, como por exemplo, objetos da classe TransformGroup
  // que agem sobre um outro objeto em comum
  bg.compile();

  return bg;
    }

    public UniversoSimples() {
  setLayout(new BorderLayout());
       GraphicsConfiguration config =
          SimpleUniverse.getPreferredConfiguration();
  // cria um objeto do tipo Canvas3D onde será projetada a
  // imagem do mundo virutal
  Canvas3D canvas3d = new Canvas3D(config);
  Add("Center", canvas3d);

  // Chama função que cria o conteúdo gráfico
  BranchGroup conteudoGrafico = criaConteudoBrachGraph();

  // Cria um objeto do tipo SimpleUniverse passando como
  // referência o objeto do tipo Canvas3D onde será
  // projetada a imagem do mundo virtual
  SimpleUniverse universo = new SimpleUniverse(canvas3d);
```

```
  // Desloca a plataforma de visão um pouco para trás
  // para que seja possível visualizar os objetos visuais
  universo.getViewingPlatform().setNominalViewingTransform();

  // Finalmente adiciona o BranchGroup de conteudo no
  // universo visual
  universo.addBranchGraph(conteudoGrafico);
    }

    public static void main(String[] args) {
  new MainFrame(new UniversoSimples(), 350, 350);
    }
}
```

Um detalhe que pode ser notado ao se analisar o código fonte é a utilização da classe `Appearance`. Como descrito anteriormente, a cada face de um objeto visual criado a partir das classes `Box`, `Cone`, `Cylinder` e `Sphere`, bem como outros objetos visuais mais complexos construídos através de outras classes, pode-se associar uma aparência específica para cada face. No exemplo mostrado, foi criado um objeto da classe `Appearance` e utilizado os métodos dessa classe para modificar os atributos de cor. Logo após, foi associado essa aparência ao cone.

A fig. 3.5 mostra o resultado gerado pela execução do programa.

FIGURA 3.5 - RESULTADO DO PROGRAMA UNIVERSO SIMPLES



Após essa abordagem da criação de um mundo virtual simples e como inserir objetos de forma básica, na seqüência serão abordadas algumas classes que permitem realizar transformações sobre os objetos visuais criados.

3.2.3 TRANSFORMAÇÕES SOBRE OBJETOS VISUAIS

Assim como existem classes no Java3D que permitem criar objetos visuais, também existem classes que permitem realizar as operações básicas de transformação sobre esses objetos. As principais classes utilizadas para isso são as classes `TransformGroup` e `Transform3D`.

O quadro 3.4 apresenta os principais construtores e métodos dessas classes:

QUADRO 3.4 - CONSTRUTORES E MÉTODOS DAS CLASSES TRANSFROMGROUP E TRANSFORM3D

Construtores da classe TransformGroup

TransformGroup() – cria e inicializa um objeto da classe TransformGroup usando uma matriz de transformação identidade.

TransformGroup (Transform3D transformacao) – cria e inicializa um objeto da classe TransformGroup associado as transformações definidas no objeto do tipo Transform3D informado.

Construtor da classe Transform3D

Tranform3D() – cria um objeto da classe Transform3D representado por uma matriz identidade (sem transformações)

Método da classe TransformGroup

void setTransform (Transform3D transformacao) – associa ao objeto da classe TransformGroup as transformações definidas no objeto do tipo Transform3D informado.

Métodos da classe Transform3D

void rotX (double angulo) – define uma transformação de rotação sobre o eixo x em sentido anti-horário de acordo com o ângulo informado em radianos.

void rotY (double angulo) – idem ao item anterior. Só que para o eixo Y.

void rotZ (double angulo) – idem ao item anterior. Só que para o eixo Z.

void setTranslation (Vector3f coordenada) – definine uma transformação de translação para as coordenadas informadas.

void setScale (double escala) – define uma transformação de escala para a escala informada.

void mul (Transform3D transformacao) – adiciona a transformação do objeto do tipo Transform3D a transformação já existente desse objeto.

Esses são só os construtores e métodos principais da classe `TransformGroup` e `Transform3D`. A classe `Transform3D`, principalmente, possui um número considerável de métodos adicionais. Para uma consulta detalhada veja Sun (2000c) e Sowizral (2000).

A aplicação de transformação sobre objetos visuais pode ser simplificada nos seguintes passos:

a) instanciar um ou mais objetos da classe `Transform3D`;
b) usar os métodos disponíveis na classe `Transform3D` para realizar as transformações desejadas;
c) caso seja utilizado mais que um objeto `Transform3D`, combinar as transformações desses objetos num único objeto `Transform3D`, ou adicionar um objeto no outro;
d) instanciar um ou mais objetos da classe `TransformGroup` referenciando os objetos `Transform3D` criados;
e) adicionar os objetos visuais que sofrerão as transformações nos objetos `TransformGroup`.

O fragmento de código mostrado no quadro 3.5 é uma adaptação feita no código mostrado no quadro 3.3, adicionando, justamente, uma transformação de translação e rotação sobre o objeto visual cone.

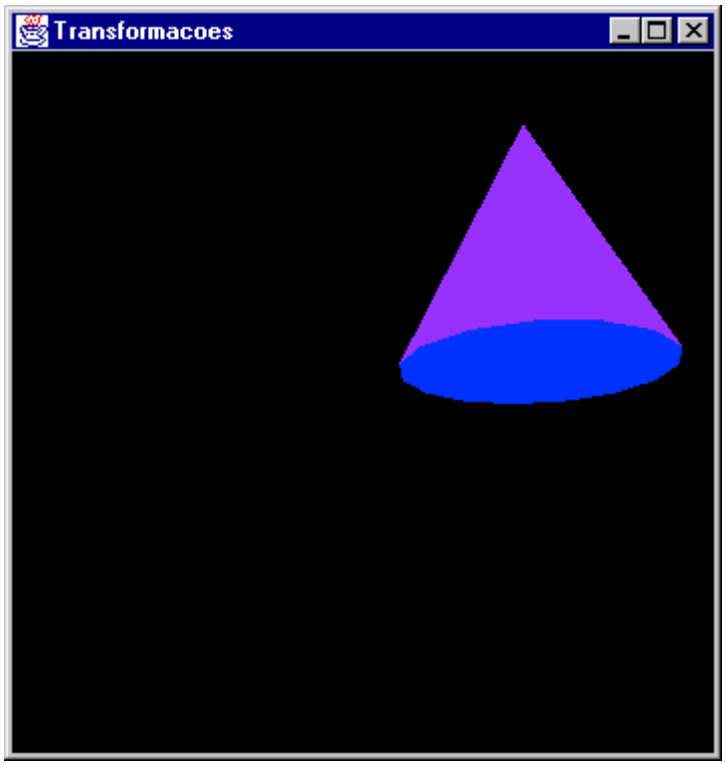
QUADRO 3.5 - FRAGMENTO DE CÓDIGO MOSTRANDO A UTILIZAÇÃO DAS CLASSES DE TRANSFORMAÇÃO

```
Public class Transformacoes extends Applet {
    Public BranchGroup criaConteudoBrachGraph() {
  // Cria o BranchGroup  de conteúdo
  BranchGroup bg = new BranchGroup();
  // Cria um objeto visual do tipo cone
  Cone objCone = new Cone(0.4f, 0.8f);
  // Cria um objeto para setar uma cor para a aparência do cone
  ColoringAttributes corCorpo =
     New ColoringAttributes(0.6f, 0.2f, 1.0f, 0);
  ColoringAttributes corBase =
     New ColoringAttributes(0.0f, 0.2f, 1.0f, 0);
  // Cria um objeto para especificar uma aparência para o cone
  Appearance aparenciaCorpo = new Appearance();
  Appearance aparenciaBase = new Appearance();
  // Atribui a cor a aparência
  aparenciaCorpo.setColoringAttributes(corCorpo);
  aparenciaBase.setColoringAttributes(corBase);
  // Atribui a aparência ao cone
  objCone.setAppearance(0, aparenciaCorpo);
  objCone.setAppearance(1, aparenciaBase);

  // Cria um objeto do tipo Transform3D para
  // realizar uma rotação sobre o eixo X do cone
```

```
Transform3D coneRotacionar = new Transform3D();
ConeRotacionar.rotX(-15*Math.PI/180);

// Cria um outro objeto do tipo Transform3D para
// realizar uma translação sobre o cone
Transform3D coneTranslar = new Transform3D();
ConeTranslar.setTranslation(new Vector3f(0.5f, 0.5f, 0.0f));
// Combina as transformações de Rotação e Translação
// no objeto coneRotacionar
coneRotacionar.mul(coneTranslar);
// Cria um objeto da classe TransformGroup referenciando
// o objeto coneRotacionar, onde estão definidas as
// transformações de rotação e translação
TransformGroup tg = new TransformGroup(coneRotacionar);

// Adiciona o cone ao TransformGroup
tg.addChild(objCone);
// Adiciona o TransformGroup ao BranchGroup de conteúdo
bg.addChild(tg);

// Chama o método compile() do BranchGroup com o objetivo
// de otimizar os objetos do mundo virtual para agilizar
// oprocesso de renderização
bg.compile();

return bg;
  }
```

A fig. 3.6 mostra o resultado das transformações realizadas sobre o cone.

FIGURA 3.6 - RESULTADO DO PROGRAMA TRANSFORMAÇÕES



Sabendo-se como criar um mundo virtual e populá-lo com objetos visuais, o próximo passo será mostrar algumas das formas disponíveis para interagir com os objetos do mundo virtual.

3.2.4 INTERAÇÃO COM O MUNDO VIRTUAL

No Java3D tanto os aspectos de interação e animação estão diretamente ligados a classe `Behavior` e suas subclasses. Esses tipos de classes oferecem mecanismos que permitem realizar modificações no grafo de cena em resposta a algum estímulo proveniente do usuário (movimentação do mouse, pressionamento de uma tecla), ou de acordo com a passagem do tempo, ou pela colisão de objetos, ou por qualquer outro estímulo definido e tratado pelo programa.

O Java3D também possui um conjunto de classes que permite tanto construir formas personalizadas de interagir com o mundo virtual, bem como classes com a maioria dos detalhes já definidos.

Dois conjuntos de classes se mostram interessantes no processo de interação com o usuário. Uma delas é a que permite interagir com o mundo virtual através do teclado, dando uma sensação de se estar caminhando no ambiente virtual. Essa classe é a `KeyNavigatorBehavior`. A outra permite utilizar o mouse para manipular objetos visuais do mundo virtual. Esse segundo conjunto é composto pelas classes `PickRotateBehavior`, `PickTranslateBehavior` e `PickZoomBehavior`.

Os passos básicos para se utilizar a classe `KeyNavigatorBehavior` são os seguintes:

a) instanciar um objeto da classe `KeyNavigatorBehavior`, informando sobre qual `TransformGroup` ele irá agir;
b) adicionar esse objeto ao grafo de cena;
c) indicar a região de atuação, fronteira (`Bounds` ou `BoundingLeaf`) para o objeto da classe `KeyNavigatorBehavior`.

O fragmento de código mostrado no quadro 3.6 mostra a criação de um mundo virtual contendo dois objetos visuais (cones) e a utilização da classe `KeyNavigatorBehavior` para navegar nesse mundo.

QUADRO 3.6 - UTILIZAÇÃO DA CLASSE KEYNAVIGATORBEHAVIOR

```
Public class NavegaUniverso extends Applet {
    public BranchGroup criaConteudoBrachGraph(SimpleUniverse universo,
     Canvas3D canvas3d) {
  // Cria o BranchGroup  de conteúdo
  BranchGroup bg = new BranchGroup();

  // Cria dois objetos visuais do tipo cone
  Cone objCone = new Cone(0.4f, 0.8f);
  Cone objCone2 = new Cone(0.3f, 0.6f);

  // Translada 0.8m pra trás o cone2
  Transform3D transCone2 = new Transform3D();
  TransCone2.setTranslation(new Vector3f(0.0f, 0.0f, -0.8f));
  TransformGroup tgCone2 = new TransformGroup(transCone2);
  tgCone2.addChild(objCone2);

  bg.addChild(tgCone2);

  // Cria um objeto para setar uma cor para a aparência do cone
  ColoringAttributes corCorpo =
    new ColoringAttributes(0.6f, 0.2f, 1.0f, 0);
  ColoringAttributes corBase =
    new ColoringAttributes(0.0f, 0.2f, 1.0f, 0);

  // Cria um objeto para especificar uma aparência para o cone
  Appearance aparenciaCorpo = new Appearance();
  Appearance aparenciaBase = new Appearance();

  // Atribui a cor a aparência
  aparenciaCorpo.setColoringAttributes(corCorpo);
  aparenciaBase.setColoringAttributes(corBase);

  // Atribui a aparência ao cone
  objCone.setAppearance(0, aparenciaCorpo);
  objCone.setAppearance(1, aparenciaBase);
  objCone2.setAppearance(0, aparenciaCorpo);
  objCone2.setAppearance(1, aparenciaBase);

  // Translada 0.5m pra esquerda e 0.5m pra baixo o cone1
  Transform3D transCone1 = new Transform3D();
  transCone1.setTranslation(new Vector3f(-0.5f,-0.5f, 0f));

  TransformGroup tg = new TransformGroup(transCone1);

  // Adiciona o cone ao TransformGroup
  tg.addChild(objCone);

  // Adiciona o TransformGroup ao BranchGroup de conteúdo
  bg.addChild(tg);

  // ** Pega o TransformGroup do BranchGroup de Visão do
  // ** SimpleUniverse
  TransformGroup brachViewTrans =
    Universo.getViewingPlatform().getViewPlatformTransform();

  // Cria o objeto da classe KeyNavigatorBehavior informando
  // sobre qual TransformGroup esse objeto vai agir
  // Nesse caso é justamente sobre o TransformGroup do
  // BranchGroup de visão do SimpleUniverse
  KeyNavigatorBehavior KeyNavBeh =
```

```
    new KeyNavigatorBehavior(brachViewTrans);
  KeyNavBeh.setSchedulingBounds
    (new BoundingSphere(new Point3d(), 1000.0));

  bg.addChild(keyNavBeh);

  // Chama o método compile() do BranchGroup com o objetivo
  // que otimizar os objetos do mundo virtual para agilizar o
  // processo de renderização
  bg.compile();

  return bg;
   }
```

Um ponto interessante do exemplo mostrado no quadro 3.6 (linha subseqüente às linhas de comentário com dois asteriscos "`// **`"), é a utilização dos métodos `getViewingPlatform` e `getViewPlatformTransform`. Nos passos para se utilizar a classe `KeyNavigatorBehavior`, é preciso indicar sobre qual `TransformGroup` ele irá agir e como se está utilizando um universo do tipo `SimpleUniverse` que cria automaticamente o `BranchGroup` de visão, esses métodos permitem que se tenha acesso ao `TransformGroup` do `BranchGroup` de visão, que é justamente onde o objeto da classe `KeyNavigatorBehavior` deve agir.

No caso da utilização das classes que permitem realizar interações através do mouse, os passos são os seguintes:

a) setar as permissões de leitura e escrita sobre os objetos da classe `TransformGroup` que irão sofrer a ação dos objetos das classes de interação com o mouse;
b) instanciar um objeto da classe de interação com o mouse (`PickRotateMouse`, `PickTranslateMouse` ou `PickZoomMouse`);
c) determinar sobre qual `TransformGroup` esses objetos irão agir;
d) indicar a região de ação, fronteira (`Bounds`, `BoundingLeaf`);
e) adicionar o objeto da classe de interação com o mouse no grafo de cena.

No fragmento de código mostrado no quadro 3.7, o mesmo universo usado no exemplo anterior é reutilizado, só que desta vez utilizam-se as classes que permitem realizar as operações de rotação, translação e escala sobre os objetos visuais do mundo virtual através do mouse. Pressionando-se o botão esquerdo do mouse sobre o objeto visual e arrastando, pode-se realizar a rotação do objeto sobre todos os seu eixos. Com o botão direito é possível

transladar o objeto e segurando-se a tecla Alt e pressionando-se o botão esquerdo do mouse pode-se transladar o objeto sobre o eixo Z.

QUADRO 3.7 - UTILIZAÇÃO DAS CLASSES PICKROTATEBEHAVIOR, PICKTRANSLATEBEHAVIOR E PICKZOOMBEHAVIOR

```
public BranchGroup criaConteudoBrachGraph(Canvas3D canvas3d) {
  // Cria o BranchGroup  de conteúdo
  BranchGroup bg = new BranchGroup();

  // Cria dois objetos visuais do tipo cone
  Cone objCone = new Cone(0.4f, 0.8f);
  Cone objCone2 = new Cone(0.3f, 0.6f);

  // Translada 0.8m pra trás o cone2
  Transform3D transCone2 = new Transform3D();
  transCone2.setTranslation(new Vector3f(0.0f, 0.0f, -0.8f));
  TransformGroup tgCone2 = new TransformGroup(transCone2);
  tgCone2.addChild(objCone2);

  // Adiciona o cone2 ao BranchGroup de conteúdo
  bg.addChild(tgCone2);

  // Cria um objeto para setar uma cor para a aparência do cone
  ColoringAttributes corCorpo =
    New ColoringAttributes(0.6f, 0.2f, 1.0f, 0);
  ColoringAttributes corBase =
    New ColoringAttributes(0.0f, 0.2f, 1.0f, 0);

  // Cria um objeto para especificar uma aparência para o cone
  Appearance aparenciaCorpo = new Appearance();
  Appearance aparenciaBase = new Appearance();

  // Atribui a cor a aparência
  aparenciaCorpo.setColoringAttributes(corCorpo);
  aparenciaBase.setColoringAttributes(corBase);

  // Atribui a aparência ao cone
  objCone.setAppearance(0, aparenciaCorpo);
  objCone.setAppearance(1, aparenciaBase);
  objCone2.setAppearance(0, aparenciaCorpo);
  objCone2.setAppearance(1, aparenciaBase);

  // Translada 0.5m pra esquerda e 0.5m pra baixo o cone1
  Transform3D transCone1 = new Transform3D();
  TransCone1.setTranslation(new Vector3f(-0.5f,-0.5f, 0f));

  TransformGroup tg = new TransformGroup(transCone1);

  // ** Seta as permissões de leitura, escrita e picking dos
  // ** TransformGroup's dos cones
  tg.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
  tg.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
  tg.setCapability(TransformGroup.ENABLE_PICK_REPORTING);
  tgCone2.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
  tgCone2.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
  tgCone2.setCapability(TransformGroup.ENABLE_PICK_REPORTING);

  // Adiciona o cone ao TransformGroup
  tg.addChild(objCone);
```

```
// Instancia os objetos que permitirão utilizar o mouse
// para rodar, translar e dar zoom nos objetos
// indicando o BranchGroup de conteúdo, o canvas onde a
// imagem do universo será renderizada e a fronteira
// de ação
BoundingSphere behaveBounds =
   New BoundingSphere(new Point3d(0,0,0), 10);
PickRotateBehavior pickRotate =
   New PickRotateBehavior(bg, canvas3d, behaveBounds);
PickTranslateBehavior pickTranslate =
   New PickTranslateBehavior(bg, canvas3d, behaveBounds);
PickZoomBehavior pickZoom =
   New PickZoomBehavior(bg, canvas3d, behaveBounds);

// Adicionar os objetos ao BranchGroup de conteúdo
bg.addChild(pickRotate);
bg.addChild(pickTranslate);
bg.addChild(pickZoom);

// Adiciona o TransformGroup ao BranchGroup de conteúdo
bg.addChild(tg);

// Chama o método compile() do BranchGroup com o objetivo de
// otimizar os objetos do mundo virtual para agilizar o processo
// de renderização
bg.compile();

return bg;
  }
```

Um ponto importante a se observar nesse fragmento de código (linhas subseqüentes às linhas de comentário indicadas por "`// **`") é a parte em que são definidas as permissões dos `TransformGroup's` do `BranchGroup` de conteúdo. Depois que o `BranchGroup` é adicionado ao objeto `Locale`, todos os objetos abaixo desse `BranchGroup` tornam-se "vivos", ou seja, eles começam a ser renderizados. Depois que os objetos visuais do mundo virtual se tornam vivos, certas características ou informações não podem mais ser acessadas ou modificadas, a não ser que certas permissões dos `TransformGroup's` aos quais esses objetos estão associados sejam modificadas. Para mais detalhes sobre as permissões que podem ser definidas sobre objetos da classe `TransformGroup` veja Sun (2000c) e Sowizral (2000).

O quadro 3.8 mostra os principais construtores e métodos das classes `PickRotateBehavior`, `PickTranslateBehavior` e `PickZoomBehavior`.

QUADRO 3.8 - CONSTRUTORES E MÉTODOS DAS CLASSES PICKROTATEBEHAVIOR, PICKTRANSLATEBEHAVIOR E PICKZOOMBEHAVIOR.

Os construtores dessas classes são semelhantes, portanto as palavras que contiverem Xxxxx devem ser substituídas pela transformação em questão (Rotate, Translate ou Zoom)

Construrotres das classes PickRotateBehavior, PickTranslateBehavior e PickZoomBehavior

PickXxxxxBehavior (BrachGroup bg, Canvas3D canvas3d, Bounds fronteira) – cria um objeto da classe PickXxxxxBehavior que aguarda pela interação do usuário através do mouse.

PickXxxxxBehavior (BrachGroup bg, Canvas3D canvas3d, Bounds fronteira, int modoMouseSobre) – idem ao anterior, só que nesse caso é possível definir o modo como se determina se o mouse está sob um objeto visual, se através da fronteira ou da geometria do mesmo.

Métodos das classes PickRotateBehavior, PickTranslateBehavior e PickZoomBehavior

void setPickMode (int modoMouseSobre) – define como deve ser determinado se o mouse está sobre um objeto visual, se através da geometria ou fronteira do mesmo.

void setupCallback (PickingCallback classeChamada) – determina a classe que deve ser chamada cada vez que uma interação com o mouse é feita.

Assim, esse capítulo apresentou um conjunto de classes e métodos que permite construir ambientes virtuais de forma simples é rápida, devido principalmente ao “alto nível” como a API do Java3D foi escrita e pode ser utilizada. Porém, é importante ressaltar que a API do Java3D é muito rica e possui inúmeras classes não abordadas aqui e que permitem trabalhar com outras características importantes na construção de um ambiente virtual. Uma fonte de referência indicada para uma abordagem completa da API do Java3D é a própria documentação da API (Sun, 2000a).

Até esse ponto, tem-se descrito os principais fatores que devem ser observados na construção de ambientes virtuais distribuídos e a API do Java3D para a construção de ambientes virtuais. No próximo capítulo, será abordada uma nova API, a do DIS-Java-VRML. As classes encontradas nessa API serão de grande ajuda para a construção dos mecanismos que irão controlar o processo de comunicação entre os usuários do ambiente virtual.

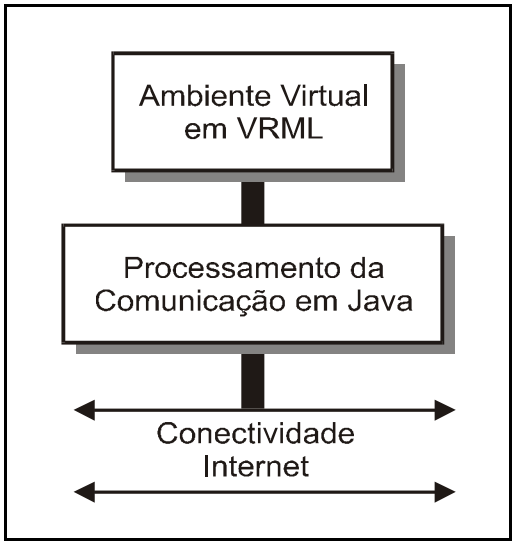
4 DIS-JAVA-VRML

Segundo Web3D (2001a), os primeiros ambientes virtuais distribuídos desenvolvidos, como o NPSNET, tinham como características comuns serem executados em equipamentos caros e desenvolvidos através de soluções próprias de software, geralmente feitas em C++ e fazendo uso de API's como da Silicon Graphics Incorporation (SGI). Esse fator representava uma certa barreira que dificultava o crescimento na área de desenvolvimento de ambientes virtuais distribuídos.

Porém, nos últimos anos, com o aumento significativo do poder de computação dos PC's, com uma redução significativa do seu custo, a necessidade de equipamentos caros deixou de ser um problema. No campo do software duas novas linguagens também surgiram criando novas expectativas: 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.

Com isso, o grupo de trabalho DIS-Java-VRML resolveu iniciar em 1997 o desenvolvimento de uma nova API, batizada com o mesmo nome do grupo. Essa API, de código fonte aberto e gratuita sob os termos da GNU *General Public License*, foi escrita na linguagem Java. Nela estão contidas classes que implementam o protocolo DIS, descrito na seção 2.5.3, 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. 4.1 ilustra a organização geral do projeto DIS-Java-VRML.

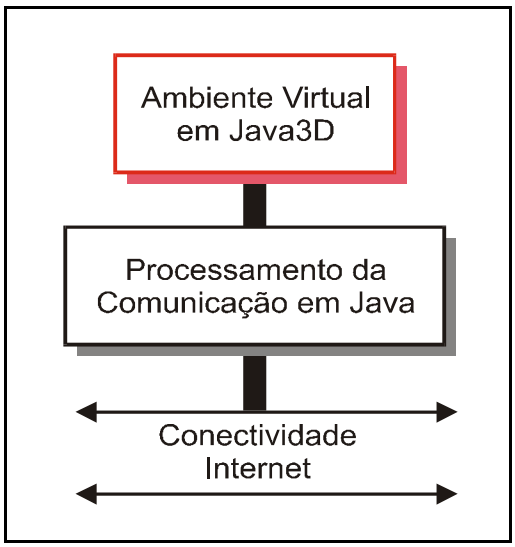
FIGURA 4.1 - ORGANIZAÇÃO GERAL DO PROJETO DIS-JAVA-VRML



Como nesse trabalho será utilizado a API do Java3D para a construção do ambiente virtual, somente as classes da API do DIS-Java-VRML que implementam o protocolo DIS e as classes que controlam o envio e recebimento de PDU's serão utilizadas. Dessa forma, as classes relacionadas com a interação das cenas em VRML não serão consideradas.

Portanto, a idéia é substituir a última camada, ilustrada na fig. 4.1, que representa o ambiente virtual desenvolvido em VRML, por outra camada que represente o ambiente virtual desenvolvido utilizando-se a API do Java3D, conforme ilustrado na fig. 4.2.

FIGURA 4.2 - SUBSTITUIÇÃO DA ÚLTIMA CAMADA VRML PELO JAVA3D



Nas próximas seções serão apresentadas somente as classes da API do DIS-Java-VRML que implementam alguns tipos de PDU's do protocolo DIS e as classes, das quais os objetos instanciados, permitem controlar o envio e recebimento de PDU's no ambiente virtual, conforme a documentação da API encontrada em Web3D (2001b). A implementação do ambiente virtual utilizando a API do Java3D, representada pela camada superior na fig. 4.2, bem como a interação da mesma com as camadas inferiores, será abordada no capítulo 5.

4.1 PRINCIPAIS CLASSES DA API DIS-JAVA-VRML QUE IMPLEMENTAM O PROTOCOLO DIS

Conforme descrito com maiores detalhes na seção 2.5.3, o protocolo DIS é formado por vários tipos de PDU's utilizados para informar sobre os eventos que podem ocorrer num ambiente virtual distribuído. Segundo Web3D (2001a), aproximadamente dois terços desses PDU's estão implementados na API do DIS-Java-VRML.

Como o número de tipos de PDU's é consideravelmente alto (cerca de 27) serão apresentadas somente as classes correspondentes a alguns tipos de PDU's, incluindo as classes herdeiras da classe `ProtocolDataUnit` (a classe `ProtocolDataUnit` é abstrata e serve como interface para suas subclasses) que são as classes `EntityStatePdu`, `CollisionPdu` e `FirePdu`. Também serão vistas as classes herdeiras da classe `SimulationManagementFamily` (também uma classe abstrata) que são as classes `CommentPdu`, `AcknowledgePdu`, `CreateEntityPdu`, `DataPdu`, `DataQueryPdu`, `SetDataPdu`, `RemoveEntityPdu`, `StartResumePdu` e `StopFreezePdu`.

Porém, antes de serem vistas as classes correspondentes aos principais tipos de PDU's, é interessante serem vistas as classes cujos objetos instanciados das mesmas permitem controlar o envio e recebimento de PDU's.

4.1.1 CLASSES BEHAVIORSTREAMBUFFERUDP E BEHAVIORSTREAMBUFFERTCP

Na API do DIS-Java-VRML se encontram duas classes cujos objetos instanciados das mesmas permitem enviar e receber PDU's num ambiente virtual. Essas duas classes são a `BehaviorStreamBufferUDP` e `BehaviorStreamBufferTCP`. Como o próprio nome demonstra, os objetos da classe `BehaviorStreamBufferUDP` são utilizados para enviar e receber PDU's sobre uma comunicação não orientada a conexão, enquanto os objetos da classe `BehaviorStreamBufferTCP` utilizam uma comunicação orientada a conexão. Maiores detalhes sobre a utilização de protocolos UDP e TCP em ambientes virtuais distribuídos podem ser vistos na seção 2.4.

O quadro 4.1 descreve os principais construtores e métodos dessas classes.

QUADRO 4.1 - PRINCIPAIS CONSTRUTORES E MÉTODOS DAS CLASSES BEHAVIORSTREAMBUFFERUDP E BEHAVIORSTREAMBUFFERTCP

Construtores da classe BehaviorStreamBufferUDP
BehaviorStreamBufferUDP () – instancia um objeto da classe BehaviorStreamBufferUDP que utilizará um modelo de comunicação unicast, utilizando-se uma porta de comunicação escolhida pelo sistema operacional.
BehaviorStreamBufferUDP (DatagramStreamBuffer datagramaParaComunicacao) – instancia um objeto da classe BehaviorStreamBufferUDP passando como parâmetro um objeto da classe DatagramStreamBuffer, sobre o qual será feita a comunicação.
BehaviorStreamBufferUDP (int portaConexao) – instancia um objeto da classe BehaviorStreamBufferUDP passando como parâmetro a porta na qual esse objeto fará a comunicação.

BehaviorStreamBufferUDP (java.lang.String enderecoMulticast, int portaConexao) – instancia um objeto da classe BehaviorStreamBufferUDP passando como parâmetro um endereço do tipo Multicast para o qual esse objeto enviará os PDU's e o número da porta na qual estabelecer a conexão.

Construtores da classe BehaviorStreamBufferTCP

BehaviorStreamBufferTCP (java.net.Socket sockerASerUtilizado) – instancia um objeto da classe BehaviorStreamBufferTCP passando como parâmetro um objeto da classe Socket, sobre o qual a comunicação será feita.

BehaviorStreamBufferTCP (java.lang.String enderecoConexao, int portaConexao) – instancia um objeto da classe BehaviorStreamBufferTCP passando como parâmetros o endereço do computador ao qual se deseja estabelecer uma conexão e a porta na qual essa comunicação será feita.

Principais métodos comuns às classes BehaviorStreamBufferTCP e BehaviorStreamBufferUDP

void cleanup () – esse método fecha todas as portas de comunicação utililizadas pelo objeto em questão.

java.util.Vector receivedPdus () – esse método retorna um objeto da classe Vector contendo todos os PDU's lidos desde a última vez em que um PDU lido pelo objeto foi recuperado.

void resumeReading () e **void suspendReading ()** – esses método fazem com que o objeto da classe específica (BehaviorStreamBufferTCP ou UDP) volte a ler ou pare, respectivamente, os pacote enviados para o computador.

void sendPdu (ProtocolDataUnit pduASerEnviado) – esse método é utilizado para enviar um PDU.

void sendPdu (ProtocolDataUnit pduASerEnviado, java.lang.Object enderecoDestino, java.lang.Objecto informacaoAdicional) – a finalidade desse método também é enviar um PDU, porém o parâmetro enderecoDestino permite especificar o endereço ou nome do host de destino desse PDU. O parâmetro informacaoAdicional pode ser nulo, ou se desejado o número da porta de conexão.

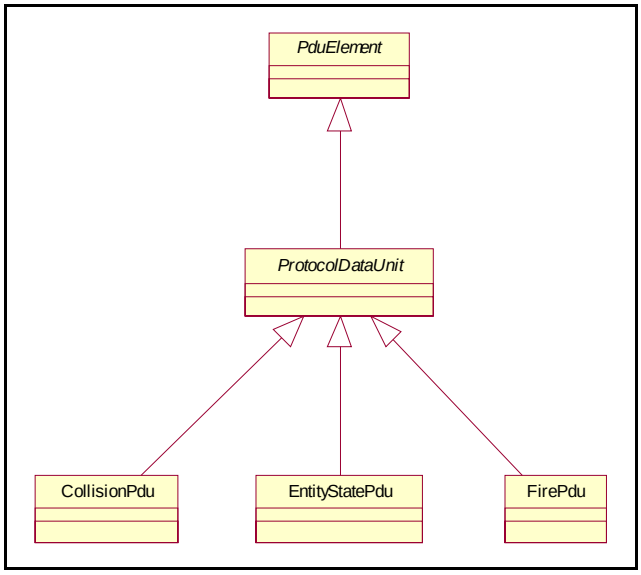
void setTimeToLive (int TTL) – método utilizado para definir o tempo de vida dos pacotes enviados por esse objeto. Os valores indicados na documentação da API são 15 para uma comunicação local, 63 para regional e 127 para global. Para maiores detalhes sobre TTL, Stevens (1995) pode ser consultado.

void shutdown () – método utilizado para terminar o loop interno principal do objeto, finalizando a Thread a qual esse objeto está associado.

4.1.2 PDU'S HERDEIROS DA CLASSE PROTOCOLDATAUNIT

A fig. 4.3 apresenta um diagrama de classes simplificado que ilustra as superclasses das classes `EntityStatePdu`, `CollisionPdu` e `FirePdu` (os atributos e métodos das classes não foram exibidos por motivos de clareza visual, tendo em vista que essas classes apresentam um número elevado de atributos e métodos).

FIGURA 4.3 - DIAGRAMA DE CLASSE ILUSTRANDO A HERANÇA DAS CLASSES COLLISIONPDU, ENTITYSTATEPDU E FIREPDU



Antes de serem detalhados os principais métodos das subclasses da classe `ProtocolDataUnit`, convém mostrar os principais métodos dessa classe, tendo em vista que todas as suas subclasses também herdam esses métodos. O quadro 4.2 apresenta esses métodos.

QUADRO 4.2 - PRINCIPAIS MÉTODOS DA CLASSE PROTOCOLDATAUNIT

UnsignedByte getExerciseID (), **void setExerciseID (int identificacaoSimulacao)** e **void setExeciseID (UnsignedByte identificacaoSimulacao)** - o primeiro método retorna um objeto da classe UnsignedByte que representa o número de identificação da simulação atribuída ao objeto dessa classe. O segundo e terceiro métodos permitem definir qual o número de identificação da simulação da qual esse objeto faz parte.

UnsignedByte getPduType () e **void setPduType (UnsignedByte tipoPdu)** – o primeiro método retorna um objeto da classe UnsignedByte que representa a qual tipo de PDU o objeto dessa classe pertence. O segundo permite definir a qual tipo de PDU o objeto dessa classe pertence. Os valores válidos para o parâmetro tipoPdu podem ser vistos na classe PduTypeField em Web3D (2001b).

UnsignedByte getProtocolFamily (), **void setProtocolFamily (int familiaDeProtocolo)** e **void setProtocolFamily(UnsignedByte familiaDeProtocolo)** – o primeiro método retorna um objeto da classe UnsignedByte que representa a qual família de protocolos o objeto dessa classe pertence. O segundo e terceito métodos permitem definir a qual família de protocolos o objeto dessa classe pertence. Os valores válidos para o parâmetro familiaDeProtocolo podem ser vistos na classe ProtocolFamilyField.

UnsignedByte getProtocolVersion (), **void setProtocolVersion (int versaoDoProtocolo)** e **void setProtocolVersion (UnsignedByte versaoDoProtocolo)** – idem aos métodos explicados anteriormente, só que esses métodos se referem a versão do protocolo utilizado. Os valores válidos para o parâmetro versaoDoProtocolo podem ser vistos na classe ProtocolVersionField.

void setSimulationStartTime (long horaInicioSimulacao) – método utilizado para definir no objeto dessa classe o horário de início da simulação.

void setTimeReceived (long horarioRecebimento) – método utilizado para definir no objeto dessa classe o horário em que ele foi recebido na aplicação.

4.1.2.1 Classe EntityStatePdu

Os objetos da classe `EntityStatePdu`, de acordo com o protocolo DIS, são utilizados para comunicar informações acerca do estado de uma entidade num ambiente virtual. Entre essas informações se encontram, por exemplo, a localização da entidade, sua velocidade linear e angular, orientação, como deve ser sua aparência, partes articuladas existentes na entidade, informações utilizadas nos algoritmos de *Dead Reckoning* (seção 2.5.2), etc. O quadro 4.3 apresenta os principais métodos dessa classe.

QUADRO 4.3 - PRINCIPAIS MÉTODOS DA CLASSE ENTITYSTATEPDU

void addArticulationParameter (ArticulationParameter parametroDeArticulacao) e **ArticulationParameter getArticulationParameterAt (int indicePosicao)** – o primeiro método adiciona um objeto da classe ArticulationParameter à lista de partes articuladas existentes no objeto dessa classe (um objeto da classe ArticulationParameter é responsável por conter informações sobre a parte articulada que será adicionada à entidade). O segundo método permite ter acesso as informações de uma parte articulada específica que foi adicionada à entidade.
UnsignedByte getDeadReckoningAlgorithm (), **void setDeadReckoningAlgorithm (int tipoAlgoritmoDeadReckoning)** e **void setDeadReckoningAlgorithm (UnsignedByte tipoAlgoritmoDeadReckoning)** – o primeiro método retorna um objeto da classe UnsignedByte que representa o tipo de algoritmo de *Dead Reckoning* que está sendo usado pelo objeto dessa classe. O segundo e terceiro métodos permitem especificar qual é o tipo de algoritmo de *Dead Reckoning* utilizado. Os valores válidos para o parâmetro tipoAlgoritmoDeadReckoning podem ser vistos na classe DeadReckoningAlgorithmField.
byte[] getDeadReckoningParameters () e **void setDeadReckoningParameters (byte[] parametrosAdicionaisDeadReckoning** – esses dois métodos permitem ter acesso e especificar, respectivamente, informações adicionais necessárias ao algoritmo de *Dead Reckoning*.
AngularVelocity getEntityAngularVelocity () e **void setEntityAngularVelocity (AngularVelocity velocidadeAngularEntidade)** – esses métodos permitem ter acesso e especificar, respectivamente, a velocidade angular da entidade. Esses métodos utilizam um objeto da classe AngularVelocity responsável por armazenar as informações de velocidade de rotação (em radianos por segundo) da entidade sobre cada um dos seus eixos (x, y e z).
UnsignedInt getEntityAppearance (), **void setEntityAppearance (int tipoAparenciaEntidade)** e **void SetEntityAppearance (UnsignedInt tipoAparenciaEntidade)** – o primeiro método retorna um objeto da classe UnsignedInt que representa a aparência da entidade quando representada no ambiente virtual. O segundo e terceito métodos permitem definir qual o tipo de aparência que a entidade deve ter. Os valores válidos para o parâmetro tipoAparenciaEntidade estão definidos nas várias classes que iniciam com o nome EntityAppearance.
EntityID getEntityID () e **void setEntityID (EntityID identificacaoEntidade)** – esses métodos permitem ter acesso e especificar, respectivamente, a identificação da entidade no ambiente virtual. O objeto da classe EntityID utilizado nesses métodos é responsável por armazenar as informações que vão identificar de forma única a entidade no ambiente virtual. Nesse objeto são armazenados três números que determinam a localidade a qual pertence a entidade (siteID), a aplicação em que está rodando a simulação (applicationID) e a própria entidade (entityID).
LinearAcceleration getEntityLinearAcceleration () e **void setEntityLinearAcceleration (LinearAcceleration velocidadeAceleracaoLinear)** – idem aos métodos que tratam a velocidade angular. Porém, nesse caso o objeto da classe LinearAcceleration é responsável por armazenar as informações de aceleração linear dos eixos x, y e z.
LinearVelocity getEntityLinearVelocity () e **void setEntityLinearVelocity (LinearVelocity**

velocidadeLinear) – idem aos métodos anteriores. Porém o objeto da classe LinearVelocity é responsável por armazenar as informações de velocidade linear dos eixos x, y e z.

WorldCoordinate getEntityLocation () e **void setEntityLocation (WorldCoordinate localizacaoEntidade)** – idem aos métodos anteriores. Porém o objeto da classe WorldCoordinate é responsável por armazenar as informações de localização da entidade no ambiente virtual, ou seja, as coordenadas x, y e z da entidade.

EulerAngle getEntityOrientation () e **void setEntityOrientation (EulerAngle angulosOrientacao)** – idem aos métodos anteriores. Porém o objeto da classe EulerAngle é responsável por armazenar as informações de orientação da entidade no ambiente virtual, ou seja o ângulo de rotação em torno do eixo x, y e z.

EntityType getEntityType () e **void setEntityType (EntityType tipoEntidade)** – idem aos métodos anteriores. Porém o objeto da classe EntityType é responsável por armazenar várias informações como o tipo da entidade, o país que desenvolveu a entidade, categoria, etc.

EntityStatePdu getExemplar () e **void setExemplar (EntityStatePdu entityStatePduModelo)** – o método setExemplar permite clonar um objeto da classe EntityStatePdu e manter uma referência interna a esse mesmo objeto para utilização futura. O método getExemplar permite ter acesso a essa cópia clonada.

UnsignedByte getForceID (), **void setForceID (int identificacaoGrupo)** e **void setForceID (UnsignedByte identificacaoGrupo)** – o primeiro método rotorna um objeto da classe UnsignedByte que representa a forma como a entidade é identificada com relação ao grupo, ou seja, se ela é aliada, inimiga, neutra, ou outra. O segundo e terceiro métodos permitem definir qual será a relação dessa entidade com os outros grupos. Os valores válidos para o parâmetro identificacaoGrupo estão definidos na classe ForceFieldID.

4.1.2.2 Classe CollisionPdu

Já os objetos da classe `CollisionPdu` são utilizados para comunicar as ocorrências de colisões ocorridas no ambiente virtual. Entre as informações mais relevantes contidas num objeto dessa classe estão a identificação da entidade que está emitindo esse PDU de colisão, a identificação da entidade que está colidindo, uma identificação de evento associada a colisão, o tipo de colisão que está ocorrendo, a velocidade em que a entidade estava na hora da colisão, a massa da entidade e a localização de onde ocorreu a colisão. O quadro 4.4 apresenta os principais métodos dessa classe.

QUADRO 4.4 - PRINCIPAIS MÉTODOS DA CLASSE `COLLISIONPDU`

EntityID getCollidingEntityID () e **void setCollidingEntityID (EntityID identificacaoEntidadeColidindo)** – o primeiro método retorna um objeto da classe EntityID que identifica a entidade que está colidindo. O segundo método permite especificar qual entidade está colidindo.

UnisignedByte getCollisionType () e **void setCollisionType (UnsignedByte tipoColisao)** – o primeiro e segundo métodos permitem ter acesso e definir qual o tipo de colisão a qual esse objeto está associado. Os valores válidos para o parâmetro tipoColisao podem ser vistos na classe CollisionTypeField.

EventID getEventID () e **void setEventID (EventID identificacaoEvento)** – o primeiro e segundo métodos permitem ter acesso e definir uma identificação de evento associada a essa colisão. O objeto da classe EventID utilizado nesses métodos contém informações referentes a localidade a qual pertence a entidade (siteID), a aplicação em que está rodando a simulação (applicationID) e um número que identifica

esse evento.

EntityID getIssuingEntityID () e **void setIssuingEntityID (EntityID identificacaoEntidade)** – esses métodos permitem ter acesso e definir a identificação da entidade que está emitindo esse PDU de colisão. Para maiores detalhes sobre o objeto EntityID utilizado nesses métodos veja o método get/setEntityID do quadro 4.3.

LinearVelocity getLinearVelocity () e **void setLinearVelocity (LinearVelocity velocidadeLinear)** – esses métodos permitem ter acesso e definir a velocidade linear dessa entidade no ato da colisão. Esses métodos seguem os mesmos princípios dos métodos get/setEntityLinearVelocity (quadro 4.3).

EntityCoordinate getLocation () e **void setLocation (EntityCoordinate coordenadasEntidade)** – esses métodos permitem ter acesso e definir a localização da entidade que está emitindo esse PDU de colisão. Esses métodos são semelhantes aos métodos get/setEntityLocation (quadro 4.3), com a diferença de que o objeto da classe EntityCoordinate armazena informações de localização da entidade relacionadas aos três eixos ortogonais, os quais tem sua origem no centro do envelope que envolve a entidade.

float getMass () e void **setMass (float massaEntidade)** – esses métodos permitem ter acesso e definir a massa da entidade que está emitindo esse PDU de colisão.

4.1.2.3 Classe FirePdu

Os objetos da classe `FirePdu` são utilizados para comunicar sobre um tiro que uma entidade esteja disparando. Entre as informações contidas num objeto dessa classe se encontram a identificação da entidade que está atirando, a identificação da entidade alvo desse tiro, a identificação do tiro que está sendo disparado, uma identificação de evento associada ao tiro, a localização do tiro no ambiente virtual, detalhes sobre o tiro, velocidade e o seu alcance. O quadro 4.5 apresenta os principais métodos dessa classe.

QUADRO 4.5 - PRINCIPAIS MÉTODOS DA CLASSE `FIREPDU`

BurstDescriptor getBurstDescriptor e **void setBurstDescriptor (BurstDescriptor detalhesTiro)** – esses métodos permitem ter acesso e definir várias informações associadas ao tiro. No objeto da classe BurstDescriptor estão informações que detalham, por exemplo, categoria do tiro, tipo de explosivo, quantidade dispara por tiro, taxa de disparo, etc. Para maiores detalhes consulte a classe BurstDescriptor.

EventID getEventID () e **void setEventID (EventID identificacaoEvento)** – idem aos métodos get/setEventID (quadro 4.4), porém nesse caso o evento está associado ao disparo e não à colisão.

EntityID getFiringEntityID () e **void setFiringEntityID (EntityID identificacaoEntidadeDisparando)** – esses métodos permitem ter acesso e definir a idenficação da entidade associada a esse PDU de disparo.

WorldCoordinate getLocationInWorldCoordinate () e **void setLocationInWorldCoordinate (WorldCoordinate localizacaoTiroNoAmbiente)** – esses métodos permitem ter acesso e definir a localização do disparo dentro do ambiente virtual.

EntityID getMunitionID () e **void setMunitionID (EntityID identifcacaoTiro)** – esses métodos permitem ter acesso e definir a identificação do tiro disparado ao qual esse PDU de disparo está associado.

float getRange () e **void setRange (float alcanceTiro)** – esses métodos permitem ter acesso e definir o alcance do tiro disparado.

LinearVelocity getVelocity () e **void setVelocity (LinearVelocity velocidadeTiro)** – esses métodos

permitem ter acesso e definir a velocidade do projétil que está sendo disparado.

EntityID getTargetEntityID () e **void setTargetEntityID (EntityID identificacaoEntidadeAlvo)** – esses métodos permitem ter acesso e definir a identificação da entidade alvo desse disparo.

Antes de ser abordado um outro grupo de classes, herdeiras da classe `SimulationManagementFamily`, o quadro 4.6 mostra trechos de código correspondentes a algumas funções que exemplificam a utilização de algumas dessas classes vistas. Explicações adicionais se encontram no próprio código, na forma de comentários precedidos por duas barras seguidas (//). A utilização de reticências (...) em duas linhas seguidas significa a omissão de linhas de código não importantes para a exemplificação.

QUADRO 4.6 - CÓDIGO-FONTE EXEMPLIFICANDO A UTILIZAÇÃO DAS CLASSES HERDEIRAS DA CLASSE `PROTOCOLDATAUNIT` E DA CLASSE `BEHAVIORSTREAMBUFFERUDP`

```
//Pacote que contém a definição das classes dos vários
//tipos de PDU (EntityStatePdu, FirePdu, CollisionPdu, etc)
//e as de comunicação de PDU's (BehaviorStreamBufferUDP)
import mil.navy.nps.dis.*;
//Pacote que contém classes que definem valores para
//alguns atributos enumerados das classes de PDU's
import mil.navy.nps.disEnumerations.*;
//Pacote que contém classes com a definição de alguns tipos
//de dados não suportados nativamente no Java (UnsignedByte,
//UnsignedInt, UnsignedShort) e que são usados
//nos vários tipos de PDU.
import mil.navy.nps.util.*;
import java.util.*;

//Classe utilizada para instanciar objetos que controlarão
//o recebimento de PDU's. Sendo essa classe herdeira da classe
//Thread, faz com que os objetos instanciados nela rodem como
//uma thread, não "travando" o ambiente virtual
class ReceptorPdus extends Thread {
  //controla execução da thread
  private boolean vivo = true;
  //Objeto BehavioStreamBufferUDP
  //usado na comunicação
  private BehaviorStreamBufferUDP bsb;
  //Variável que armazena temporariamente a identificação
  //da aplicação que está enviando o PDU
  private short tempId;
  //Variável que armazena a identificação da aplicação
  //local, que está recebendo o PDU
  private short appId;
  //Variável utilizada para manter referência
  //ao objeto que instancia um objeto dessa classe
  //permitindo invocar métodos do mesmo
  private AmbienteVirtual ambienteVirtual;

  ReceptorPdus (AmbienteVirtual ambVirtual) {
    ambienteVirtual = ambVirtual;
    appId = ambienteVirtual.getIdentificacaoAplicacao();
    //instancia um objeto da classe BehaviorStreamBufferUDP,
    //sendo que este objeto será utilizado para ficar recenbendo
    //PDU's na porta 8133
```

```
    bsb = new BehaviorStreamBufferUDP (8133);
    //instancia um novo objeto da classe Thread passando como
    //referência o objeto do tipo BehaviorStreamBufferUDP. A classe
    //BehaviorStreamBufferUDP implementa a classe Runnable, podendo dessa
    //forma rodar como uma Thread
    Thread leitorPDU = new Thread(bsb);
    //chama o método run() implementado na classe BehaviorStreamBufferUDP,
    //fazendo com que o objeto esteja pronto pra receber PDU's
    leitorPDU.start();
  }

  //método para controlar a execução da Thread
  public void setVivo (boolean continuaRodando) {
    vivo = continuaRodando;
  }

  public void run () {
    while (vivo) {
      //chama o método receivedPdus() do objeto bsb da classe
      //BehaviorStreamBufferUDP, retornando um vetor com todos os
      //PDU's já recebidos
      java.util.Vector vetorPDU = bsb.receivedPdus();
      //percorre esse vetor analizando cada um desses PDU's recebidos e
      //chamando os procedimentos necessários
      for (Enumeration enum = vetorPDU.elements();
                  enum.hasMoreElements();;) {
        //converte o objeto armazenado no vetor para um objeto da classe
        //ProtocolDataUnit, a classe mais genérica dos PDU's
        ProtocolDataUnit tempPDU = (ProtocolDataUnit) enum.nextElement();
        //através do dado retornado do método getPduType(), será possível
        //identificar de qual tipo é esse PDU que está sendo tratado
        UnsignedByte tipoPDU = tempPDU.getPduType();
        switch (tipoPDU.intValue()) {
          //caso seja um PDU do tipo EntityStatePdu
          case PduTypeField.ENTITYSTATE : {
            //converte ele pra classe específica
            EntityStatePdu tempESPDU = (EntityStatePdu) tempPDU;
            //pega o dado que identifica de qual aplicação partiu esse
            //PDU
            tempId =
               tempESPDU.getEntityID().getApplicationID().shortValue();
            //se for diferente da aplicação local esse PDU
            //deve ser tratado
            if (tempId != appId)
              //chama método responsável por tratar a informação
              //de atualização contida nesse PDU
              ambienteVirtual.atualizaEntidade(tempESPDU);
            break;
          }
          //caso seja um PDU do tipo FirePdu
          case PduTypeField.FIRE : {
            FirePdu tempFIPDU = (FirePdu) tempPDU;
            tempId =
        tempFIPDU.getFiringEntityID().getApplicationID().shortValue();
            if (tempId != appId)
              //chama método responsável por tratar a informação
              //de disparo contida nesse PDU
              ambienteVirtual.criaTiroRemoto(tempFIPDU);
            break;
          }
          //caso seja um PDU do tipo CollisionPDU
          case PduTypeField.COLLISION : {
```

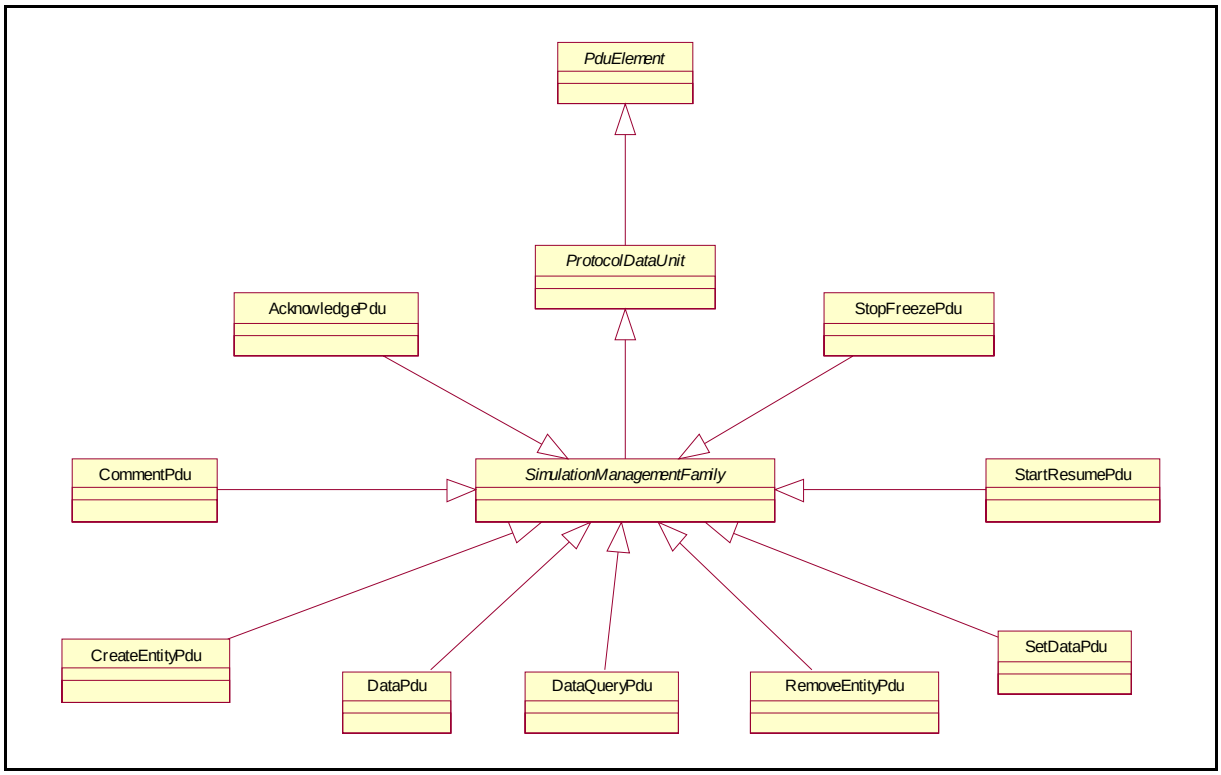
```
                CollisionPdu tempColiPDU = (CollisionPdu) tempPDU;
                tempId =
        tempColiPDU.getIssuingEntityID().getApplicationID().shortValue();
                if (tempId != appId)
                  //chama método responsável por tratar a informação
                  //de colisão contida nesse PDU
                  ambienteVirtual.trataColisao(tempColiPDU);
                break;
              }
            }
          }
          //Dorme 250 milisegundos
          try {
            Thread.sleep(250);
          } catch (Exception e) {
            System.out.println ("Problemas no sleep da Thread: " + e);
          }
        }
        //saindo do run() da thread dá um shutdown() no bsb para
        //interromper o recebimento de PDU's e depois um cleanup()
        //para fechar a comunicação
        bsb.shutdown();
        bsb.cleanup();
    }

}

class AmbienteVirtual {
    private ReceptorPdus receptorPdus;
    //...
    //...

    public void iniciaReceptorPdus() {
        //instancia objeto ReceptorPdus, passando como parâmetro
        //o objeto dessa própria classe (AmbienteVirtual) para que de lá
        //seja possível chamar os
        //métodos desse objeto daqui que tratarão dos PDU's (atualizaEntidade,
        //criaTiroRemoto, etc)
        receptorPdus = new ReceptorPdus(this);
        receptorPdus.start();
    }

    public void atualizaEntidade (EntityStatePdu ESPdu) {
        // Código responsável por atualizar a entidade relacionada
        // com Entity State Pdu passado como parâmetro
    }

    public void criaTiroRemoto (FirePdu FrPdu) {
        // Código responsável por tratar as informações sobre
        // o tiro informado no Fire Pdu
    }

    public void trataColisao (CollisionPdu ColiPdu) {
        // Código responsável por tratar a colisão informada
        // no Colision Pdu
    }

    //...
    //...

}
```

Na seqüência, tem-se a descrição de mais alguns tipos de PDU's, sendo esses herdeiros da classe `SimulationManagementFamily`.

4.1.3 PDU'S HERDEIROS DA CLASSE SIMULATIONMANAGEMENTFAMILY

A fig. 4.4 apresenta um diagrama de classes simplificado que ilustra as superclasses das classes `DataQueryPdu`, `DataPdu`, `SetDataPdu`, `CreateEntityPdu`, `RemoveEntityPdu`, `AcknowledgePdu`, `CommentPdu`, `StartResumePdu` e `StopFreesePdu`.

FIGURA 4.4 - DIAGRAMA DE CLASSES ILUSTRANDO AS PRINCIPAIS CLASSES HERDEIRAS DA CLASSE `SIMULATIONMANAGEMENTFAMILY`



Assim como foi interessante mostrar os principais métodos da classe `ProtocolDataUnit` na seção 4.1.2, aqui também se torna conveniente mostrar os métodos adicionais que a classe `SimulationManagementFamily` possui, pois todos os PDU's mencionados no parágrafo anterior são classes derivadas da classe `SimulationManagementFamily`. O quadro 4.7 mostra os principais métodos dessa classe.

QUADRO 4.7 - PRINCIPAIS MÉTODOS DA CLASSE SIMULATIONMANAGEMENTFAMILY

EntityID getOriginatingEntityID () e **void setOriginatingEntityID (EntityID identificacaoEntidadeOrigem)** – esses métodos permitem ter acesso e definir a identificação da entidade que está enviando esse PDU.
EntityID getReceivingEntityID () e **void setReceivingEntityID (EntityID identificacaoEntidadeDestino)** – idem aos métodos anteriores, porém nesses métodos é possível ter acesso e definir a identificação da entidade para a qual esse PDU se destina.

4.1.3.1 Classe DataQueryPdu

Os objetos dessa classe são utilizados para requisitar a outras entidades dados em geral. As informações contidas em objetos dessa classe são: identificação da entidade que está solicitando os dados, identificação da entidade da qual se quer obter os dados (esses dois atributos são herdados da superclasse `SimulationManagementFamily`, aliás todas as classes descritas nas sub seções da seção 4.1.2 herdam essa informação, portanto não serão mais mencionadas nas próximas classes apresentadas), número de identificação dessa requisição, uma lista fixa e variável que contém identificadores de informações a serem solicitadas. O quadro 4.8 descreve os principais métodos dessa classe.

QUADRO 4.8 - PRINCIPAIS MÉTODOS DA CLASSE DATAQUERYPDU

void addFixedDatumID (long identificacaoDadoTamanhoFixoRequisitado) e **void addVariableDatumID (long identificacaoDadoTamanhoVariavelRequisitado)** – o primeiro método permite adicionar um identificador na lista de identificadores de dados de tamanho fixo requisitados. Já o segundo adiciona um identificador na lista de identificadores de dados de tamanho variável requisitados.
void dropFixedDatumID () e **void dropVariableDatumID ()** – esses métodos permitem limpar as listas de identificadores de dados de tamanho fixo e variável, respectivamente.
long fixedDatumIDAt (int posicaoIdentificadorDadoTamanho Fixo) e **long variableDatumIDAt (int posicaoIdentificadorDadoTamanhoVariavel)** – esses métodos permitem ter acesso aos identificadores de dados requisitados numa posição específica da lista de identificadores de dados de tamanho fixo e variável, respectivamente.
int fixedDatumIDCount () e **variableDatumIDCount ()** – esses métodos retornam o tamanho da lista de identificadores de dados de tamanho fixo e variável, respectivamente.
DataQueryPdu getExemplar () e **void setExemplar (DataQueryPdu pduRequisaoDado)** – método com a mesma funcionalidade do método get/setExemplar apresentado no quadro 4.3.
UnsignedInt getRequestID () e **void setRequestID (long identificacaoRequisicao)** – esses métodos permitem ter acesso e definir um número que identifica a requisição de dado que está sendo feita.

4.1.3.2 Classes DataPdu e SetDataPdu

O objeto da classe `DataPdu` é enviado como resposta a uma solicitação feita por um objeto da classe `QueryDataPdu`. Nele, além da informação que identifica qual requisição ele está atendendo, há os dados referentes a solicitação feita.

Os mesmos métodos da classe `DataPdu` se aplicam à classe `SetDataPdu`. A diferença entre os objetos dessas classes é que um objeto da classe `DataPdu` geralmente é utilizado como resposta a uma solicitação de dados feita por uma outra entidade, enquanto que um objeto da classe `SetDataPdu` é utilizado para inicializar ou alterar uma entidade específica sem uma prévia solicitação. O quadro 4.9 descreve os principais métodos dessas classes.

QUADRO 4.9 - PRINCIPAIS MÉTODOS DAS CLASSES DATAPDU E SETDATAPDU

DatumSpecification getDatumInformation () e **void setDatumInformation (DatumSpecification especificacaoDados)** – esses métodos permitem ter acesso e definir um objeto da classe DatumSpecification associado a esse PDU. O objeto da classe DatumSpecification é responsável por armazenar duas listas de identificadores de dados requisitados, uma para dados de tamanho fixo e outra para dados de tamanho variável. É interessante observar que nessas listas são armazenados objetos da classe FixedDatum e Variable Datum, respectivamente. Por sua vez, os objetos da classe FixedDatum e VariableDatum são responsáveis por armazenar um número que identifica o tipo do dado e o valor desse dado em si. No caso do FixedDatum o atributo que armazena o valor do dado em si é do tipo int, já no VariableDatum esse atributo é do tipo array de long. Além disso o objeto da classe VariableDatum possui um atributo a mais que é o tamanho do dado). Os valores válidos para esse número que identifica o tipo de dado podem ser vistos na classe DatumIDField. **DataPdu getExemplar () e void setExemplar (DataPdu pduDados)** – idem aos métodos get/setExemplar (quadro 4.3). **UnsignedInt getRequestID ()** e **void setRequestID (long identificacaoRequisicao)** – esses métodos permitem ter acesso e definir um número que identifica a requisição a qual esse PDU está atendendo.

4.1.3.3 Classes CreateEntityPdu e RemoveEntityPdu

Como o próprio nome diz, os objetos dessas classes são utilizados para comunicar a criação ou remoção de uma entidade no ambiente virtual. As informações armazenadas nos objetos dessa classe são, conforme já mencionado anteriormente no que diz respeito aos objetos herdeiros da classe `SimulationManagementFamily`, as informações referentes às identificações das entidades de origem e destino e um número que identifica a requisição de criação ou remoção de entidade. O quadro 4.10 descreve os principais métodos dessas classes.

QUADRO 4.10 - PRINCIPAIS MÉTODOS DAS CLASSES CREATEENTITYPDU E REMOVEENTITYPDU

CreateEntityPdu getExemplar (), **void setExemplar (CreateEntityPdu pduCriacaoEntidade)** e **RemoveEntityPdu getExemplar ()**, **void setExemplar (RemoveEntityPdu pduRemocaoEntidade)** – mesma funcionalidade dos métodos get/setExemplar (quadro 4.3). Os métodos que fazem referência ao objeto da classe CreateEntityPdu, logicamente, se aplicam aos objetos da classe CreateEntityPdu e os que fazem referência ao objeto da classe RemoveEntityPdu, se aplicam aos objetos da classe RemoveEntityPdu. **UnsignedInt getRequestID ()**, **void setRequestID (UnsignedInt identificacaoRequisicao)** e **void setRequestID (long identificacaoRequisicao)** – mesma funcionalidade dos métodos get/setRequestID (quadro 4.9), porém esses identificadores de requisição se referem a criação ou remoção de entidade.

4.1.3.4 Classe AcknowledgePdu

Os objetos dessa classe têm por finalidade servir como resposta às requisições feitas pelo envio de objetos das classes `StartResumePdu`, `StopFreezePdu` e `Create`/`RemoveEntityPdu`, sendo essa resposta no sentido de aceitar ou não essa requisição. As informações contidas nos objetos dessa classe são: um *flag* que indica o tipo de requisição que o objeto dessa classe está respondendo, um *flag* indicando a aceitação da requisição e um número que identifica a qual requisição esse objeto está se referindo. O quadro 4.11 descreve os principais métodos dessa classe.

QUADRO 4.11 - PRINCIPAIS MÉTODOS DA CLASSE ACKNOWLEDGEPDU

UnsignedShort getAcknowledgeFlag (), **void setAcknowledgeFlag (UnsignedShort tipoRequisicaoRespondida)** e **void setAcknowledgeFlag (int tipoRequisicaoRespondida)** – esses métodos permitem ter acesso e definir qual o tipo de requisição esse objeto está respondendo, ou seja, se ele está respondendo a uma requisição de criar entidade, remover entidade, parar ou iniciar a simulação, etc. Os valores válidos para o parâmetro tipoRequisicaoRespondida podem ser vistos na classe AcknowledgeFlagField. **AcknowledgePdu getExemplar ()** e **void setExemplar (AcknowledgePdu pduResposta)** – mesma funcionalidade dos métodos get/setExemplar (quadro 4.3). **UnsignedInt getRequestID ()**, **void setRequestID (UnsignedInt identificacaoRequisicao)** e **void setRequestID (int identificacaoRequisicao)** – também possuem a mesma finalidade dos métodos get/setRequestID (quadro 4.9), porém nesse caso eles se referem a identificação da requisição a qual está sendo respondida. **UnsignedShort getResponseFlag ()**, **void setResponseFlag (UnsignedShort statusRequisicao)** e **void setResponse (int statusRequisicao)** – esses métodos permitem ter acesso e definir uma resposta à requisição feita, no sentido de aceitar ou não a requisição. Os valores válidos para o parâmetro statusRequisicao podem ser vistos na classe ResponseFlagField.

4.1.3.5 Classe CommentPdu

Utilizam-se os objetos dessa classe para enviar mensagens de qualquer natureza para outras entidades. A informação contida nos objetos dessa classe é justamente os dados da mensagem que se deseja enviar. O quadro 4.12 descreve os principais métodos dessa classe.

QUADRO 4.12 - PRINCIPAIS MÉTODOS DA CLASSE COMMENTPDU

DatumSpecification getDatumSpecification () e **void setDatumSpecification (DatumSpecification especificacaoDadosMensagem)** – esses métodos funcionam de maneira semelhante aos métodos das classes DataPdu e SetDataPdu. A diferença nesse caso é que o objeto da classe DatumSpecification mantém somente uma lista que identifica dados de tamanho variável, o que é condizente com o envio de mensagens arbitrárias de tamanho variável.
CommentPdu getExemplar () e **void setExemplar (CommentPdu pduMensagem)** – idem métodos get/setExemplar (quadro 4.3).

4.1.3.6 Classe StartResumePdu

Os objetos dessa classe são utilizados para comunicar o início ou reinício da simulação no ambiente virtual. As informações contidas nos objetos dessa classe são: horário do mundo real em que deverá iniciar ou reiniciar a simulação, horário do ambiente virtual em que deverá iniciar ou reiniciar a simulação e um número que identifica a requisição de início ou reinício da simulação. O quadro 4.13 descreve os principais métodos dessa classe.

QUADRO 4.13 - PRINCIPAIS MÉTODOS DA CLASSE STARTRESUMEPDU

StarResumePdu getExemplar () e **void setStartResumePdu (StartResumePdu pduInicioReinicio)** – idem aos métodos get/setExemplar (quadro 4.3).
ClockTime getRealWorldTime () e **void setRealWorldTime (ClockTime horarioMundoReal)** – esses métodos permitem ter acesso e definir um objeto da classe ClockTime que representa o horário no mundo real no qual a simulação deve iniciar ou reiniciar. No objeto da classe ClockTime são armazenadas a quantidade de horas decorridas desde 1 de janeiro de 1970 e a quantidade de tempo decorrido desde a última hora.
UnsignedInt getRequestID (), **void setRequestID (UnsignedInt identificacaoRequisicao)** e **void setRequestID (int identificacaoRequisicao)** – possuem a mesma finalidade dos métodos get/setRequestID (quadro 4.9), porém nesse caso eles se referem a identificação da requisição para iniciar ou reiniciar a simulação.
ClockTime getSimulationTime () e **void setSimulationTime (ClockTime horarioAmbienteVirtual)** – idem aos métodos get/setRealWorldTime, porém o objeto da classe ClockTime tratado nesses métodos se refere ao horário no ambiente virtual e não no mundo real.

4.1.3.7 Classe StopFreezePdu

A utilização de um objeto dessa classe é indicada para comunicar a parada ou congelamento de uma entidade ou exercício. Os objetos dessa classe têm a seguinte informação armazenada: qual o tipo de comportamento a ser assumido no ato da parada ou congelamento, o tipo da razão para essa parada ou congelamento e o horário do mundo real em que a entidade ou exercício deve ser parado ou congelado. O quadro 4.14 descreve os principais métodos dessa classe.

QUADRO 4.14 - PRINCIPAIS MÉTODOS DA CLASSE ST0PFREEZEPDU

StopFreezePdu getExemplar () e **void setStopFreezePdu (StopFreezePdu pduParadaCongelamento)** – idem aos métodos get/setExemplar (quadro 4.3).
UnsignedByte getFrozenBehavior (), **void setFrozenBehavior (UnsignedByte tipoComportamentoParado)** e **void setFrozenBehavior (int tipoComportamentoParado)** – esses métodos permitem ter acesso ou definir qual o tipo de comportamento que a entidade deve assumir no ato da parada ou congelamento da entidade ou exercício. Os valores válidos para o parâmetro tipoComportamentoParado podem ser visto na classe FrozenBehaviorField.
ClockTime getRealWorldTime () e **void setRealWorldTime (ClockTime horarioMundoReal)** – idem aos métodos get/setRealWorldTime (quadro 4.13). Porém nesse caso o objeto da clase ClockTime se refere a hora no mundo real em que a entidade deve parar ou congelar no exercício ou simulação.
UnsignedByte getReason (), **void setReason (UnsignedByte tipoMotivoParada)** e **void setReason (int tipoMotivoParada)** – esses métodos permitem ter acesso ou definir qual o tipo de motivo pelo qual a entidade ou exercício está sendo parado. Os valores válidos para o parâmetro tipoMotivoParada podem ser visto na classe ReasonField.
UnsignedInt getRequestID (), void setRequestID (UnsignedInt identificacaoRequisicao) e **void setRequestID (int identificacaoRequisicao)** – possuem a mesma finalidade dos métodos get/setRequestID (quadro 4.9), porém nesse caso eles se referem a identificação da requisição para parar ou congelar uma entidade ou simulação.

Com essa visão geral dos principais métodos e classes que são a implementação de alguns tipos de PDU's do protoco DIS, fica mais fácil perceber como a comunicação entre as entidades do ambiente virtual irá ocorrer. Dependendo da informação a ser comunicada, basta instanciar um objeto da classe correspondente ao PDU que será utilizado e transmitir esse PDU para os demais participantes do ambiente virtual.

O trecho de código descrito no quadro 4.6 apresentou exemplos de como utilizar o objeto da classe `BehaviorStreamBufferUDP` para receber os PDU's enviados por outros usuários do ambiente virtual. O quadro 4.15, a seguir, apresenta mais alguns trechos de código onde se pode ver a utilização de mais alguns tipos de PDU's e como utilizar o objeto da classe `BehaviorStreamBufferUDP` para enviar os PDU's. Explicações adicionais também se encontram inseridas no código.

QUADRO 4.15 - CÓDIGO-FONTE COM EXEMPLOS DE USO DAS CLASSES HERDEIRAS DA CLASSE SIMULATIONMANAGEMENTFAMILY E DA CLASSE BEHAVIORSTREAMBUFFERUDP

```
Import mil.navy.nps.dis.*;
import mil.navy.nps.disEnumerations.*;
import mil.navy.nps.util.*;
import java.util.*;

class AmbienteVirtualExemplo {
  //Objeto da classe BehaviorStreamBuffer utilizado para enviar os PDU's
  private BehaviorStreamBufferUDP distribuidorPdus;
  //...
  //...

  AmbienteVirtualExemplo () {
    //inicializa o distribuidorPdus, será utilizada
    //a porta 8134 para enviar os PDU's o TTL dos pacotes
    //enviados será de 15 hops
    distribuidorPdus = new BehaviorStreamBufferUDP (8134);
    distribuidorPdus.setTimeToLive(15);
    //...
    //...
  }

  //todas as informações definidas nesses PDU's que serão
  //enviados através do objeto distribuidorPdus serão importantes
  //para as outras aplicações que receberem esses PDU's e
  //através delas manter o ambiente virtual como um todo atualizado

  //função utilizada para enviar PDU's do tipo EntityStatePdu
  public void enviaInformacaoAtualizacao(
                          EntityID identificacaoEntidade,
                          WorldCoordinate localizacaoEntidade,
                          EulerAngle orientacaoEntidade,
                          AngularVelocity velocidadeAngular,
                          LinearAcceleration aceleracaoLinear,
                          LinearVelocity velocidadeLinear) {
    EntityStatePdu esPdu = new EntityStatePdu();
    //define a identificação da entidade a qual esse PDU
    //está relacionado
    esPdu.setEntityID(identificacaoEntidade);
    //define a atual localização da entidade a ser informada nesse PDU
    esPdu.setEntityLocation(localizacaoEntidade);
    //idem para a orientação, velocidade angular, aceleração linear e
    //velocidade linear
    esPdu.setEntityOrientation(orientacaoEntidade);
    esPdu.setEntityAngularVelocity(velocidadeAngular);
    esPdu.setEntityLinearAcceleration(aceleracaoLinear);
    esPdu.setEntityLinearVelocity(velocidadeLinear);
    //faz uma cópia de si mesmo
    esPdu.setExemplar(esPdu);
    Thread threadDistribuidoraPdus = new Thread(distribuidorPdus);
    //envia a cópia desse pdu para o endereço de broadcast contido
    //na variável enderecoBroadcast na porta 8133
    //a especificação de um endereço de broadcast como esse permite fazer
    //com que todas as aplicações que se encontram na mesma subrede desse
    //endereço e que estejam "ouvindo" na porta 8133 recebam o PDU
    distribuidorPdus.sendPdu(esPdu.getExemplar(),
                      enderecoBroadcast, 8133);
  }
  //função utilizada para enviar PDU's do tipo CreateEntityPdu
  public void enviaInformacaoCriacaoEntidade(
                             EntityID identificacaoEntidadeOrigem,
```

```
                            EntityID identificacaoEntidadeDestino,
                            UnsignedInt identificacaoRequisicao) {
    CreateEntityPdu cePdu = new CreateEntityPdu();
    cePdu.setOriginatingEntityID(identificacaoEntidadeOrigem);
    cePdu.setReceivingEntityID(identificacaoEntidadeDestino);
    //define uma identificação de requisição para esse PDU de criação.
    //Numa simulação onde haja uma máquina responsável por gerenciar
    //a simulação, esse número de requisição se torna útil para mais
    //tarde identificar sobre qual requisição um PDU do tipo
    //AcknowledgePdu se refere. Funciona dessa forma: a aplicação que
    //deseja criar, remover, parar, iniciar entidades ou a simulação,
    //precisa enviar um PDU correspondente para a máquina gerenciadora da
    //simulação. A máquina gerenciadora analisa essa requisição e envia um
    //AcknowledgePdu informando sobre o sucesso ou não dessa requisição.
    //Será através do RequestID que a aplicação vai saber sobre qual
    //requisição o AcknowledgePdu se refere
    cePdu.setRequestID(identificacaoRequisicao);
    cePdu.setExemplar(cePdu);
    Thread threadDistribuidoraPdus = new Thread(distribuidorPdus);
    distribuidorPdus.sendPdu(cePdu.getExemplar(),
                    enderecoBroadcast, 8133);
  }

  //função utilizada para enviar PDU's do tipo CommentPdu
  public void enviaComentario(
                    EntityID identificacaoEntidadeOrigem,
                    EntityID identificacaoEntidadeDestino,
                    String comentario) {
    //esse pré-processamento feito é necessário para transformar
    //a mensagem que se quer transmitir (String) num array de longs
    //No geral funciona assim:
    //1 - cria-se um objeto da classe VariableDatum;
    //2 - define-se um identificador pra mensagem (ver classe DatumField)
    //3 - define-se o tamanho dele, que será o próprio
    //    tamanho do array de longs
    //4 - define-se o valor dele, que será o próprio array de longs
    //5 - cria-se um objeto da classe DatumSpecification
    //6 - adiciona-se o objeto da classe VariableDatum nele
    //7 - e por último utiliza-se o método setDatumSpecification
    //do PDU de comentário, passando como parâmetro o objeto
    //da classe DatumSpecification
    byte comentarioByteArray[] = new byte[comentario.length()];
    long comentarioLongArray[] = new long[comentario.length()];
    comentarioByteArray = comentario.getBytes();
    for (int i = 0; i < comentario.length(); i++)
      comentarioLongArray[i] = (long) comentarioByteArray[i];
    CommentPdu cmPdu = new CommentPdu();
    cmPdu.setOriginatingEntityID(identificacaoEntidadeOrigem);
    cmPdu.setReceivingEntityID(identificacaoEntidadeDestino);
    // - 1 -
    VariableDatum dadoVariavel = new VariableDatum();
    // - 2 -
    dadoVariavel.setVariableDatumID(1);
    // - 3 -
    dadoVariavel.setVariableDatumLength(comentario.length());
    // - 4 -
    dadoVariavel.setVariableDatumValue(comentarioLongArray);
    // - 5 -
    DatumSpecification listaTiposDadosVariavel = new DatumSpecification();
    // - 6 -
    listaTiposDadosVariavel.addVariableDatum(dadoVariavel);
    // - 7 -
```

```
    cmPdu.setDatumSpecification(listaTiposDadosVariavel);
    cmPdu.setExemplar(cmPdu);
    Thread threadDistribuidoraPdus = new Thread(distribuidorPdus);
    distribuidorPdus.sendPdu(cmPdu.getExemplar(),
                     enderecoBroadcast, 8133);
  }

  //função utilizada para enviar PDU's do tipo FirePdu
  public void enviaInformacaoDisparo(
                       EntityID identificacaoEntidadeDisparando,
                       EntityID identificacaoEntidadeAlvo,
                       EntityID identificacaoTiro,
                       EventID identificacaoEvento,
                       WorldCoordinate localizacaoNoAmbiente,
                       BurstDescriptor descricaoTiro,
                       LinearVelocity velocidadeTiro) {
    FirePdu frPdu = new FirePdu();
    frPdu.setFiringEntityID(identificacaoEntidadeDisparando);
    frPdu.setTargetEntityID(identificacaoEntidadeAlvo);
    frPdu.setEventID(identificacaoEvento);
    frPdu.setLocationInWorldCoordinate(localizacaoNoAmbiente);
    //descricaoTiro da classe BurstDescriptor contém um série de
    //informações que especificam os detalhes do tiro (ver
    //classe BurstDescriptor)
    frPdu.setBurstDescriptor(descricaoTiro);
    frPdu.setVelocity(velocidadeTiro);
    Thread threadDistribuidoraPdus = new Thread(distribuidorPdus);
    distribuidorPdus.sendPdu(frPdu, enderecoBroadcast, 8133);
  }

  //função utilizada para enviar PDU's do tipo StopFreezePdu
  public void enviaInformacaoParada(
                       EntityID identificacaoEntidadeOrigem,
                       EntityID identificacaoEntidadeDestino,
                       UnsignedByte identificacaoMotivo,
                       UnsignedByte identificacaoComportamentoParado,
                       ClockTime horarioParada,
                       UnsignedInt identificacaoRequisicao) {
    StopFreezePdu sfPdu = new StopFreezePdu();
    sfPdu.setOriginatingEntityID(identificacaoEntidadeOrigem);
    sfPdu.setReceivingEntityID(identificacaoEntidadeDestino);
    //define-se a razão da parada (ver classe ReasonField, para os
    //valores válidos
    sfPdu.setReason(identificacaoMotivo);
    //define-se o tipo de comportamento que entidade/simulação
    //congelada deve assumir (ver classe FrozenBehaviorField
    //para os valores válidos
    sfPdu.setFrozenBehavior(identificacaoComportamentoParado);
    sfPdu.setRealWorldTime(horarioParada);
    sfPdu.setRequestID(identificacaoRequisicao);
    sfPdu.setExemplar(sfPdu);
    Thread threadDistribuidoraPdus = new Thread(distribuidorPdus);
    distribuidorPdus.sendPdu(sfPdu.getExemplar(),
                     enderecoBroadcast, 8133);
  }
  //...
  //...
}
```

Com essa abordagem geral da API do DIS-Java-VRML, pôde-se ver as principais classes correspondentes aos principais tipos de PDU’s do protocolo DIS que estão implementados nela. Adicionalmente, também foram apresentadas duas classes que permitem controlar o processo de envio e recebimento de PDU’s.

O capítulo seguinte apresenta a descrição do protótipo de ambiente virtual distribuído construído como resultado desse trabalho onde se procurou utilizar a API do Java3D para construir o ambiente virtual, em conjunto com a API do DIS-Java-VRML para realizar o processo de comunicação entre os usuários do ambiente virtual.

5 DESENVOLVIMENTO DO PROTÓTIPO

Com os assuntos abordados nos capítulos anteriores, pôde-se desenvolver um protótipo de ambiente virtual distribuído multiusuário. A principal preocupação desse protótipo era possibilitar a participação de mais de um usuário e fazer com que houvesse um grau de sincronia aceitável entre os mesmos.

A seção 5.1 apresenta os requisitos identificados para a construção do protótipo. Na seção 5.2, tem-se a especificação e implementação (optou-se por apresentar a especificação e implementação juntas para facilitar o entendimento das principais funções do protótipo). 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 a construção do protótipo foram considerados quatro requisitos básicos:

a) utilização de um modelo de comunicação distribuído;
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) utilização da técnica de *heartbeats*

5.2 ESPECIFICAÇÃO E IMPLEMENTAÇÃO

Seguindo a mesma linguagem em que foram implementadas as duas API's utilizadas nesse protótipo, foi utilizada a linguagem Java para o desenvolvimento do protótipo. Mais especificamente, foi utilizado o JSDK 1.3. A versão da API do Java3D utilizada para a criação e manipulação do ambiente virtual foi a 1.2.1 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.

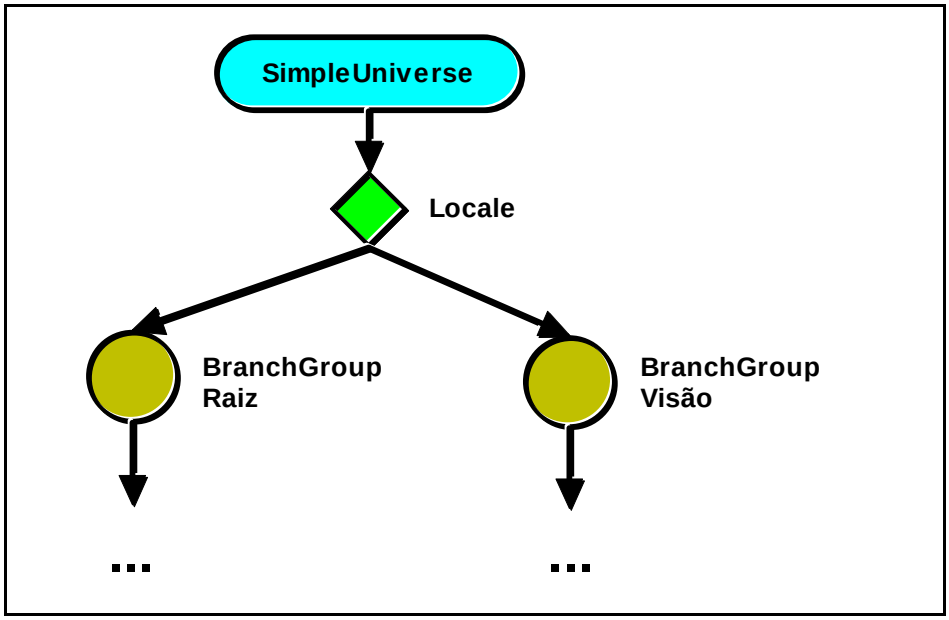
Como ponto de partida para a especificação do protótipo será apresentado o grafo de cena, explicado em maiores detalhes na seção 3.1, que representa a estrutura geral do ambiente virtual. Na seqüência, serão apresentados os fluxogramas que especificam as principais funções do protótipo, ao mesmo tempo em que, em alguns casos, serão

apresentados trechos de código para facilitar o entendimento e exemplificar como foi implementada a lógica especificada nos fluxogramas.

5.2.1 GRAFO DE CENA GERAL DO AMBIENTE VIRTUAL

A fig. 5.1 apresenta a estrutura geral (simplificada) do grafo de cena com os dois `BranchGroup's` principais do ambiente virtual: o `BranchGroup` raiz (correspondente ao `BranchGroup` de conteúdo) e o `BranchGroup` de visão. Do lado do `BranchGroup` raiz ficam todos os nós responsáveis por representar os objetos visuais que popularão o ambiente virtual. Do lado do `BranchGroup` de visão ficam todos os nós responsáveis por representar os objetos que controlam o mecanismo de visão do ambiente virtual. Para uma melhor visualização, o detalhamento dos dois `BranchGroup's` será apresentado nas figs. 5.2 e 5.3.

FIGURA 5.1 - BRANCHGROUP'S PRINCIPAIS DO AMBIENTE VIRTUAL



Também na implementação desse protótipo foi utilizada a classe `SimpleUniverse` para construir o ambiente virtual, que como já apresentada na seção 3.2.1, apresenta maior facilidade por já criar automaticamente toda uma estrutura pré-definida para o `BranchGroup` de visão, incluindo o objeto `Locale`, permitindo acrescentar posteriormente novas estruturas, na medida em que for necessário.

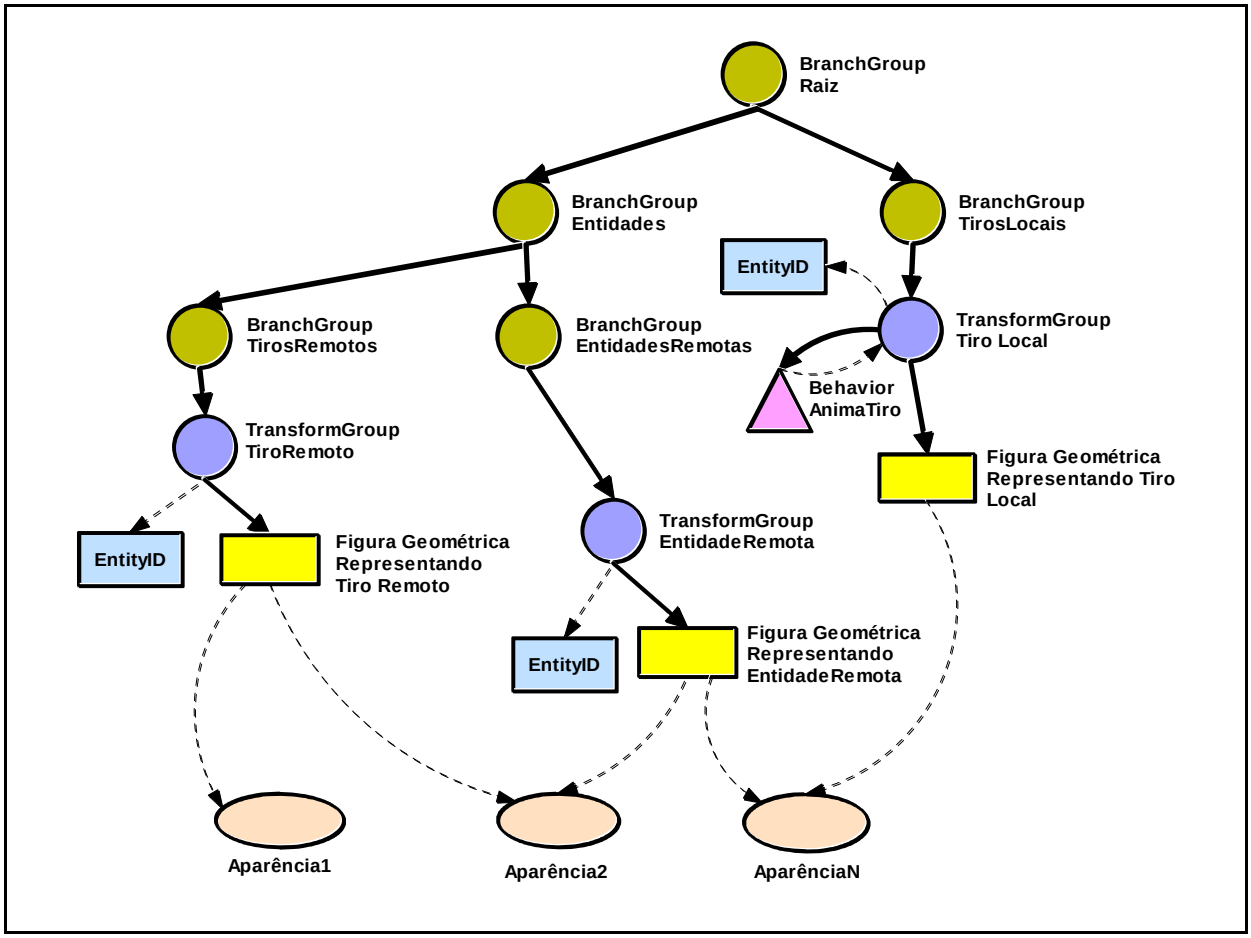
A fig. 5.2 apresenta mais detalhadamente o `BranchGroup` raiz. Os dois `BranchGroup's` diretamente relacionados com ele são o `BranchGroup` de entidades e o `BranchGroup` de tiros locais. Como o próprio nome sugere, será no `BranchGroup` de

tiros locais que serão adicionados nós que representarão os tiros disparados localmente pela entidade também local.

O `BranchGroup` de entidades se subdivide nos `BranchGroup's` de entidades remotas e tiros remotos. No `BranchGroup` de entidades remotas serão adicionados nós que representarão as entidades remotas que entraram no ambiente virtual, ou seja, o objeto visual que representa o personagem dos outros usuários. No `BranchGroup` de tiros remotos ficam os objetos que representam os tiros disparados por esses usuários remotos.

A inclusão dos demais nós em cada um dos `BranchGroup's` ficará mais clara à medida que forem explicadas as principais funções do protótipo responsáveis por controlar a entrada no ambiente virtual, entrada de novos usuários e disparos de tiros locais e remotos.

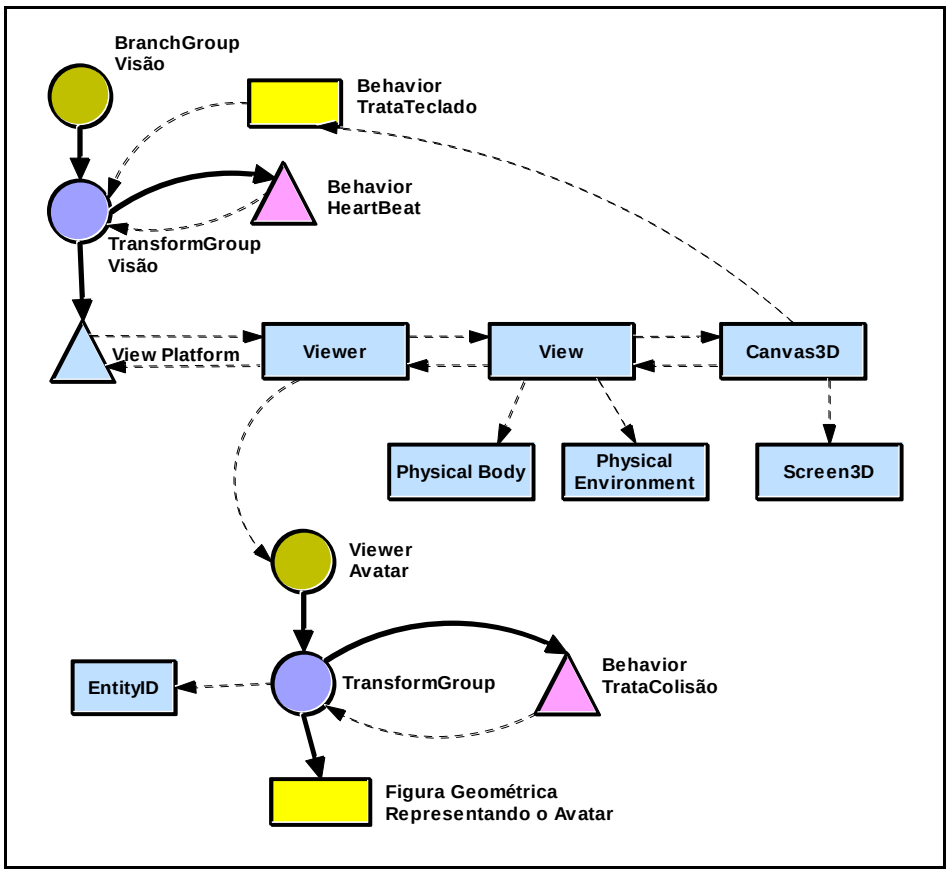
FIGURA 5.2 - DETALHAMENTO DO BRANCHGROUP RAIZ



O detalhamento do `BranchGroup` de visão aparece na fig. 5.3. Com exceção dos nós correspondentes ao `BehaviorTrataTeclado`, `BehaviorHeartBeat` e o subgrafo correspondente ao `ViewerAvatar`, todos os demais nós são criados automaticamente na criação do objeto da classe `SimpleUniverse`. A inclusão desses nós adicionais ao grafo de

cena também ficarão mais claros na explicação das classes `TrataTecladoBehavior` (seção 5.1.5), `HeartBeatBehavior` (seção 5.1.7) e da função “Entrar no ambiente virtual” (seção 5.1.2).

FIGURA 5.3 - DETALHAMENTO DO BRANCHGROUP DE VISÃO



Do ponto de vista gráfico, esses grafos de cena apresentados conseguem especificar a estrutura gráfica completa do protótipo. O quadro 5.1, a seguir, mostra um trecho de código onde é criada a estrutura principal do ambiente virtual. Nele está exemplificada a criação do objeto `SimpleUniverse`, bem como a criação de cada um dos `BranchGroup's` abaixo do `BranchGroup` raiz: `BranchGroup` de entidades, `BranchGroup` de tiros locais, `BranchGroup` de tiros remotos e o `BranchGroup` de entidades remotas.

QUADRO 5.1 - CÓDIGO EXEMPLIFICANDO A CRIAÇÃO DOS PRINCIPAIS BRANCHGROUP'S

```
...
...
//universo é um objeto da classe SimpleUniverse
universo = new SimpleUniverse(canvas3d);

//instancia cada um dos BranchGroups
bgRaiz = new BranchGroup();
bgEntidades = new BranchGroup();
bgTirosLocais = new BranchGroup();
bgTirosRemotos = new BranchGroup();
bgEntidadesRemotas = new BranchGroup();
```

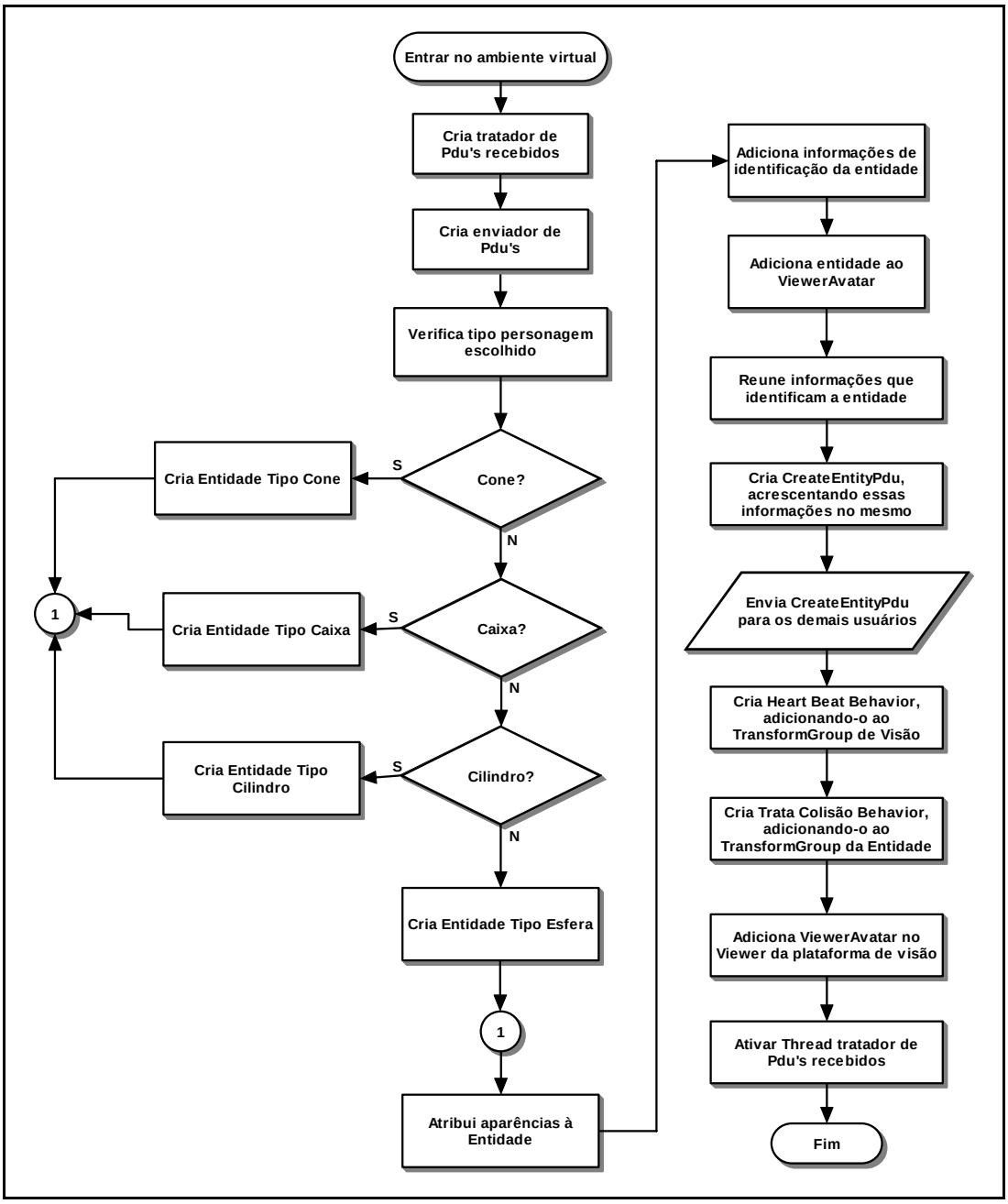
```
    //Modifica as permissões dos BranchGroup's instanciados
    //para permitir que estes possam ser desconectados
    //após terem sido conectados pela primeira vez
    //Isso é necessário pois a criação e eliminação dos objetos
    //visuais é feita dinamicamente e depois que um BranchGroup
    //é conectado não se pode alterá-lo a não ser que se
    //modifique essas permissões
    bgRaiz.setCapability(BranchGroup.ALLOW_DETACH);
    bgEntidades.setCapability(BranchGroup.ALLOW_DETACH);
    bgTirosLocais.setCapability(BranchGroup.ALLOW_DETACH);
    bgTirosRemotos.setCapability(BranchGroup.ALLOW_DETACH);
    bgEntidadesRemotas.setCapability(BranchGroup.ALLOWDETACH);

    //a permissão ALLOW_CHILDREN_READ está sendo setada
    //nesses BranchGroup's para que seja possível utilizar
    //determinados métodos que permitem ter acesso
    //aos objetos que estão abaixo dos mesmos
    bgEntidades.setCapability(BranchGroup.ALLOW_CHILDREN_READ);
    bgTirosLocais.setCapability(BranchGroup.ALLOW_CHILDREN_READ);
    bgTirosRemotos.setCapability(BranchGroup.ALLOW_CHILDREN_READ);
    bgEntidadesRemotas.setCapability(BranchGroup.ALLOW_CHILDREN_READ);

    //adiciona os BranchGroup's uns nos outros conforme
    //a estrutura do grafo de cena apresentado na fig. 5.2
    bgEntidades.addChild(bgTirosRemotos);
    bgEntidades.addChild(bgEntidadesRemotas);
    bgRaiz.addChild(bgEntidades);
    bgRaiz.addChild(bgTirosLocais);

    //Adiciona o BranchGroup principal no universo
    //A partir desse momento todos os objetos visuais
    //abaixo desse BranchGroup se tornam "vivos"
    //e passam a ser visualizados(renderizados)
    universo.addBranchGraph(bgRaiz);
    ...
    ...
```

Como já explicado anteriormente, os outros nós serão explicados posteriormente, quando forem apresentadas as especificações das funções diretamente relacionadas com esses nós. Também é importante lembrar que os trechos de código aqui apresentados foram simplificados ao máximo para dar maior clareza. Por esse motivo algumas linhas de código podem ter sido omitidas.

5.2.2 ENTRANDO NO AMBIENTE VIRTUAL

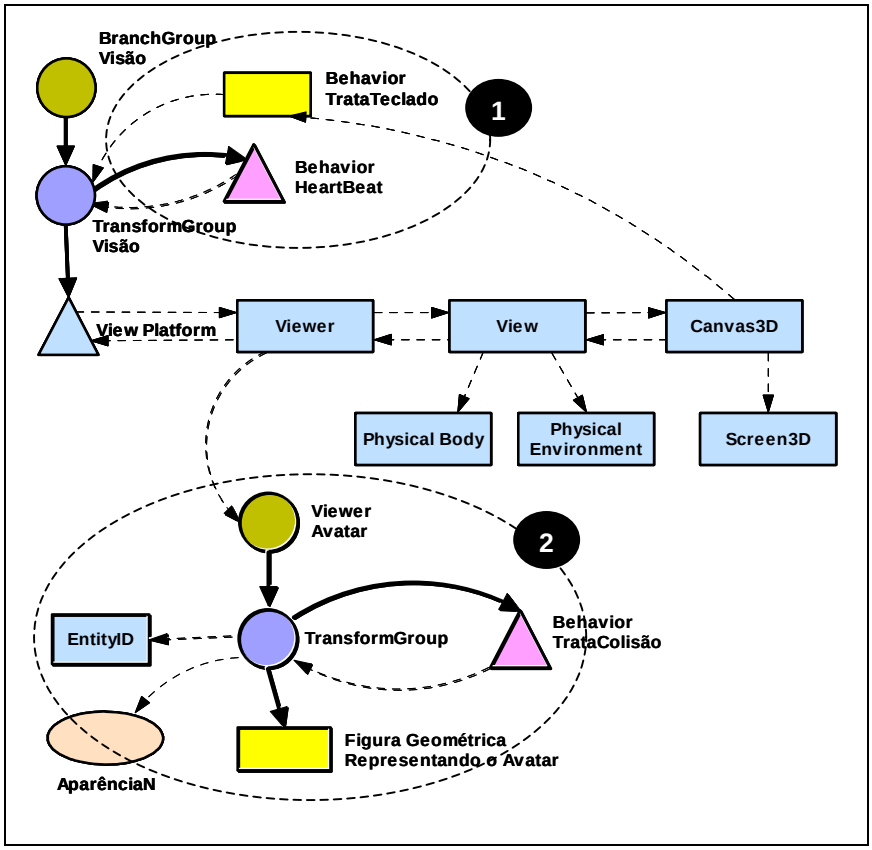
O processo “Entrar no ambiente virtual”, especificado no fluxograma da fig. 5.4, demonstra as principais tarefas executadas quando o usuário entra no ambiente virtual.

FIGURA 5.4 - PROCESSO ”ENTRAR NO AMBIENTE VIRTUAL”



As elipses 1 e 2 da fig. 5.5 indicam os nós do grafo de cena da plataforma de visão que que representam os objetos criados como conseqüência da execução desse processo. Os demais nós são criados automaticamente ao se criar um objeto da classe SimpleUniverse, conforme detalhado na seção 3.2.1.

FIGURA 5.5 - NÓS DO GRAFO DE CENA QUE REPRESENTAM OS OBJETOS CRIADOS PELO PROCESSO “ENTRAR NO AMBIENTE VIRTUAL”



Na seqüência o quadro 5.2 apresenta o trecho de código que exemplifica a implementação desse processo, no caso do usuário ter escolhido um personagem do tipo cone.

QUADRO 5.2 - CÓDIGO EXEMPLIFICANDO O PROCESSO DE ENTRADA NO AMBIENTE VIRTUAL

```
...
...
//o objeto tratadorPdu é da classe TratadorPdu
//A classe TratadorPdu, bem como sua lógica de funcionamento
//estão explicadas na seção 5.1.3
tratadorPdu = new TratadorPdu(getProto(), appId);

//o objeto bsb é da classe BehaviorStreamBufferUDP e será
//utilizado para enviar Pdu's para os demais usuários do
//ambiente virtual
bsb = new BehaviorStreamBufferUDP(8134);
bsb.setTimeToLive(15);
enviadorPDU = new Thread(bsb);
...
...
//caso o personagem escolhido seja um cone
Float raio = new Float(0.2f);
Float altura = new Float(0.35f);

//Instancia um objeto do tipo cone nas dimensões acima definidas
Cone objCone = new Cone(raio.floatValue(), altura.floatValue());

//Atribui uma aparência para cada uma das faces do cone
//apn são objetos da classe Appearance, utilizados para definir
```

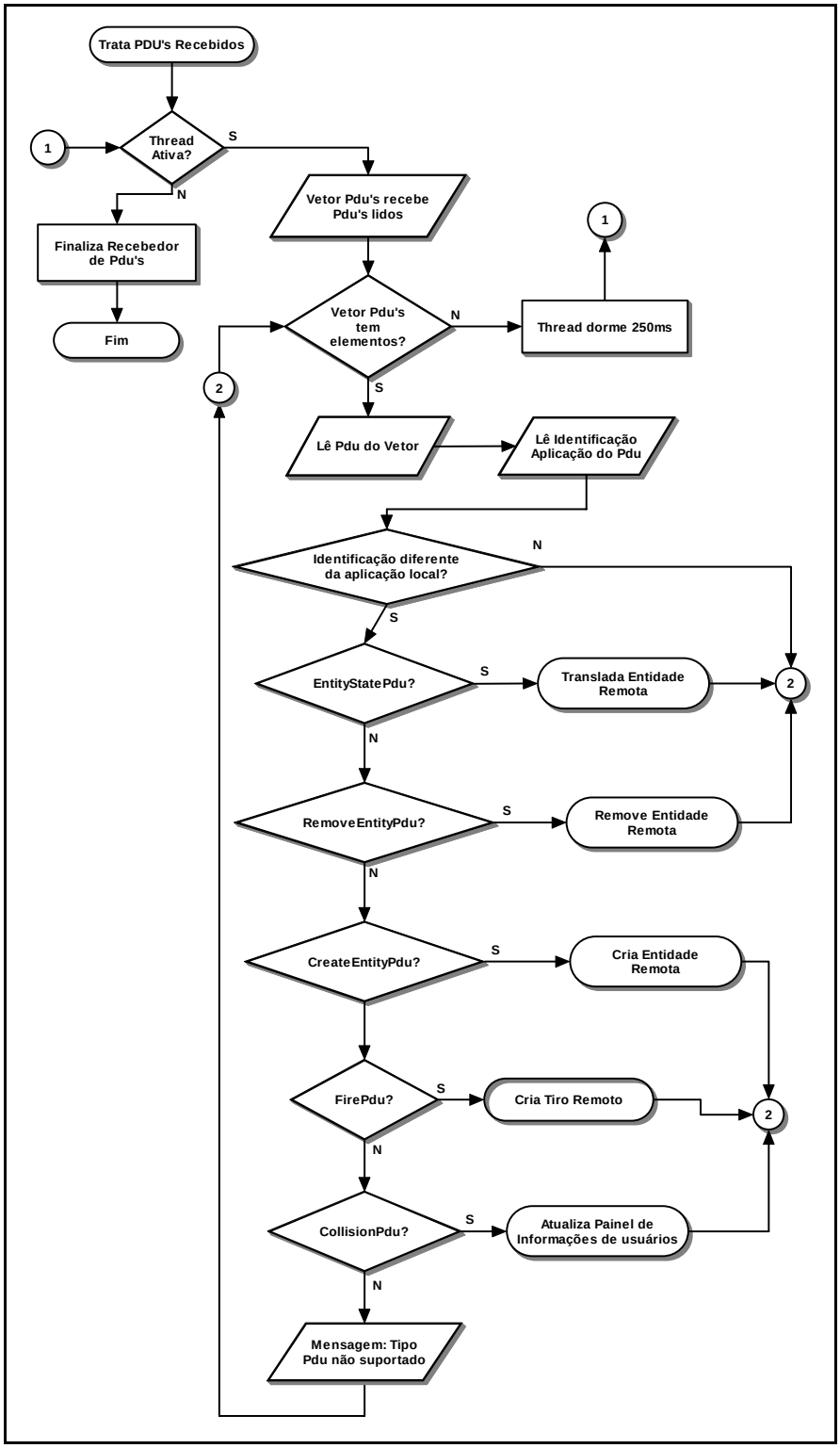
```
    //como será a aparência dos objetos visuais (cor, textura, etc)
    objCone.setAppearance(0, ap1);
    objCone.setAppearance(1, ap2);

    //Instancia um objeto do tipo TransformGroup que
    //será responsável por manter as informações das
    //transformações feitas sobre esse objeto
    TransformGroup tg = new TransformGroup();

    //Adiciona o objeto do tipo cone ao objeto do tipo
    //TransformGroup
    tg.addChild(objCone);

    //Instancia um objeto da classe EntityID e a referencia com o
    //TransformGroup do cone. Cada entidade do ambiente
    //virtual possui relacionada consigo um objeto dessa classe que a
    //a identifica de forma única no ambiente.
    EntityID entId = new EntityID(siteId, appId, (short)1);
    tg.setUserData(entId);

    //Instancia um objeto da classe ViewerAvatar a qual será
    //adicionado o TransformGroup do cone
    ViewerAvatar va = new ViewerAvatar();
    va.addChild(tg);

    //Instancia um objeto da classe CreateEntityPdu, o qual
    //será utilizado para informar acerca da entrada desse usuário
    //no ambiente virtual para os demais usuários do ambiente virtual
    CreateEntityPdu cePDU = new CreateEntityPdu();
    cePDU.setOriginatingEntityID(entId);
    cePDU.setExemplar(cePDU);
    bsb.sendPdu(cePDU.getExemplar(), enderecoBroadcast, 8133);

    //Instancia um objeto da classe HeartBeatBehavior. Essa classe
    //bem como a classe seguinte TrataColisaoBehavior serão vistas
    //nas seções seguintes. Porém é interessante observar que na criação
    //desse objeto está sendo passado como parâmetro uma referência
    //ao TransformGroup da plataforma de visão. Isso é importante pois
    //esse comportamento é responsável por informar o estado da entidade
    //de tempos em tempos, informação essa retirada justamente do
    //TransformGroup da plataforma de visão, que é onde essa entidade, que
    //está sendo criada nesse exemplo, será adicionada
    transformGroupDeVisao =
         universo.getViewingPlatform().getViewPlatformTransform();
    heartBeatBehavior = new HeartBeatBehavior(transformGroupDeVisao,
                                              getProto());
    heartBeatBehavior.setSchedulingBounds(new BoundingSphere());
    transformGroupDeVisao.addChild(heartBeatBehavior);

    //Cria tratador de colisões, passando como parâmetro uma referência
    //ao TransformGroup do cone que está sendo criado e posteriormente
    //adicionado esse objeto ao próprio TransformGroup
    TrataColisaoBehavior trataColisaoBehavior =
         new TrataColisaoBehavior(getProto(), tg);
    trataColisaoBehavior.setSchedulingBounds
         (new BoundingSphere(new Point3d(0, 0, 0), 2));
    tg.addChild(trataColisaoBehavior);

    //Adiciona o ViewerAvatar no Viewer da plataforma de visão
    universo.getViewer().setAvatar(va);

    //Ativa a thread tratadorPdu. A partir desse momento a aplicação
```

```
    //começa a receber e tratar os Pdu's enviados por outros usuários
    tratadorPdu.start();
    ...
    ...
```

5.2.3 TRATADOR DE PDU'S RECEBIDOS

Um objeto da classe `TratadorPdu` é responsável por ficar recebendo e processando todos os PDU's enviados por outros usuários. Essa classe foi implementada como uma *thread* para permitir que seu funcionamento não interfira no funcionamento da interface do usuário. A sua lógica de funcionamento está especificada no fluxograma da fig. 5.6. De forma resumida, fica-se esperando por PDU's que sejam enviados por outros usuários e assim que os recebe faz o processamento adequado de acordo com o tipo e origem do PDU.

FIGURA 5.6 - LÓGICA DO PROCESSAMENTO FEITO POR UM OBJETO DA CLASSE TRATAD0RPDU



O quadro 5.3 apresenta um trecho de código que exemplifica a implementação do bloco principal dessa classe.

QUADRO 5.3 - CÓDIGO EXEMPLIFICANDO A IMPLEMENTAÇÃO DO BLOCO PRINCIPAL DA CLASSE TRATADORPDU

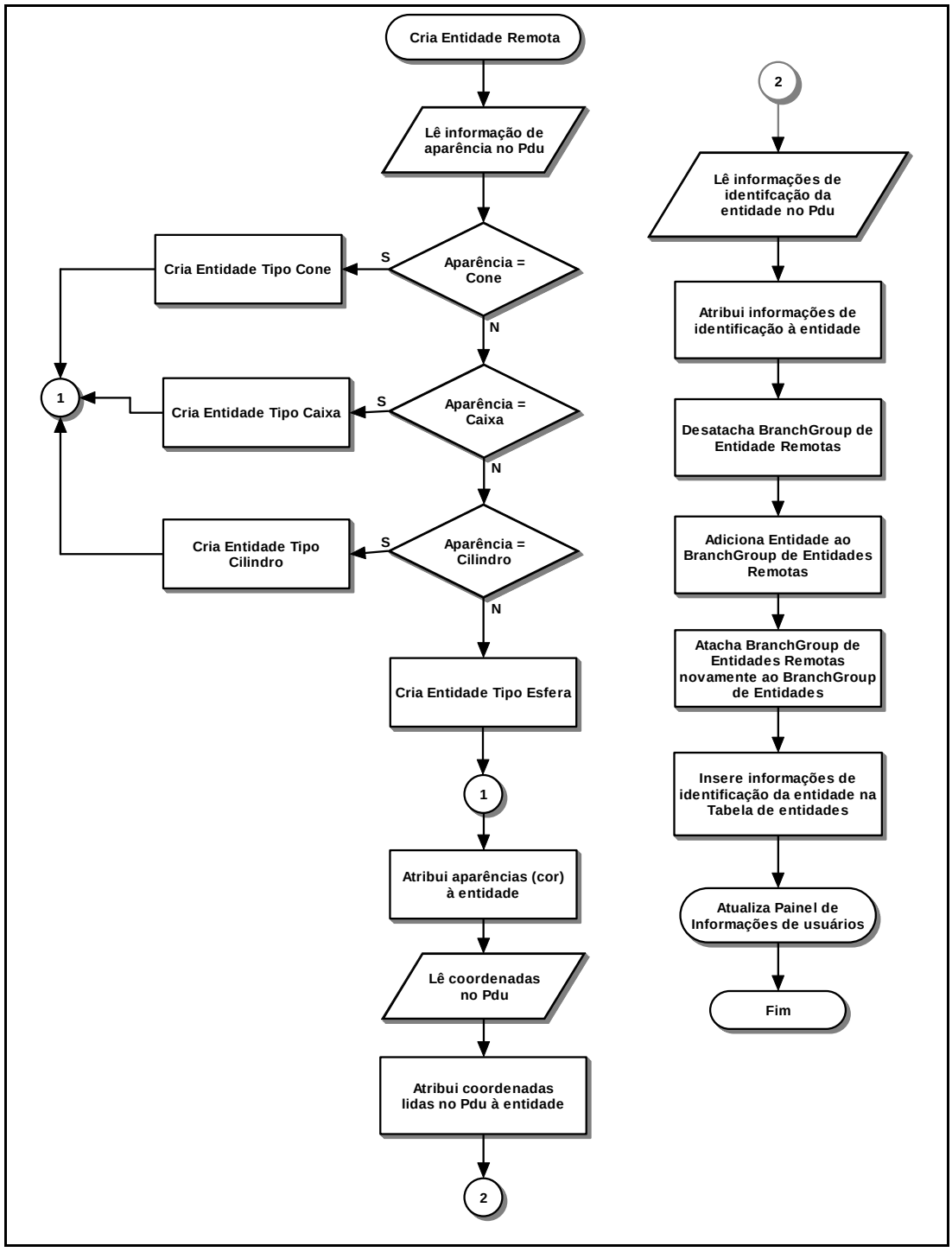
```
...
...
//enquanto a Thread estiver ativa
while (vivo) {
  //lê Pdu's recebidos do objeto bsb da classe BehaviorStreamBufferUDP
  //Esse objeto fica "ouvindo" todos os Pdu's enviados para a porta
  //8133
  java.util.Vector vetorPDU = bsb.receivedPdus();
  //para cada Pdu recebido
  for (Enumeration enum = vetorPDU.elements();
           enum.hasMoreElements();) {
    //Pega Pdu do vetor
    ProtocolDataUnit tempPDU = (ProtocolDataUnit) enum.nextElement();
    //Pega o tipo do Pdu
    UnsignedByte tipoPDU = tempPDU.getPduType();
    //Verifica de qual tipo ele é
    switch (tipoPDU.intValue()) {
      //caso seja um EntityStatePdu
      case PduTypeField.ENTITYSTATE : {
        //converte para um EntityStatePdu
        EntityStatePdu tempESPDU = (EntityStatePdu) tempPDU;
        //Pega identificação da aplicação que enviou o Pdu
        tempId =
          tempESPDU.getEntityID().getApplicationID().shortValue();
        //Se for um Pdu enviado por outra aplicação
        if (tempId != appId)
          transladaEntidadeRemota(tempESPDU);
        break;
      }
      //caso seja um RemoveEntityPdu
      case PduTypeField.REMOVEENTITY : {
        RemoveEntityPdu tempREPDU = (RemoveEntityPdu) tempPDU;
        tempId =
             tempREPDU.getOriginatingEntityID().
             getApplicationID().shortValue();
        if (tempId != appId)
          removeEntidadeRemota(tempREPDU);
        break;
      }
      //caso seja um CreateEntityPdu
      case PduTypeField.CREATEENTITY : {
        CreateEntityPdu tempCEPDU = (CreateEntityPdu) tempPDU;
        tempId =
             tempCEPDU.getOriginatingEntityID().
             getApplicationID().shortValue();
        if (tempId != appId))
          criaEntidadeRemota(tempCEPDU);
        break;
      }
      //caso seja um FirePdu
      case PduTypeField.FIRE : {
        FirePdu tempFIPDU = (FirePdu) tempPDU;
        tempId =
             tempFIPDU.getFiringEntityID().
             getApplicationID().shortValue();
        if (tempId != appId)
          criaTiroRemoto(tempFIPDU);
        break;
      }
      ...
```

```
            ...
            }
         }
      }
      //Após cada vetor de Pdu's recebidos processado, dorme 250ms
      try {
         Thread.sleep(250);
      } catch (Exception e) {
         System.out.println ("Problemas no sleep da Thread: " + e);
      }
   }
   //Após thread não estar mais ativa, fecha bsb "escutador" de Pdu's
   bsb.shutdown();
   bsb.cleanup();
}
...
...
```

5.2.4 CRIA ENTIDADE REMOTA

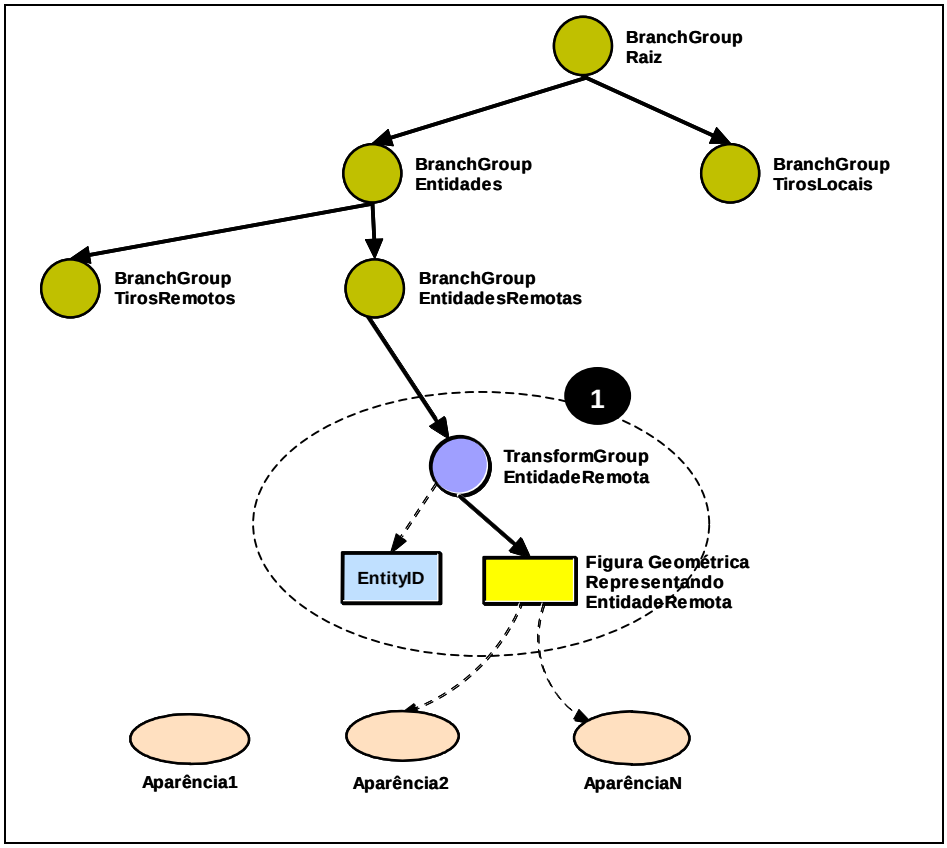
Quando um usuário entra no ambiente virtual, conforme visto no processo da seção 5.1.2, este envia para os demais usuários um `CreateEntityPdu` informando a respeito da criação de uma nova entidade no ambiente virtual. A finalidade do processo "Cria entidade remota" é justamente criar uma entidade nova no ambiente virtual, como conseqüência do recebimento de um `CreateEntityPdu`. A fig. 5.7 apresenta o fluxograma correspondente a esse processo.

FIGURA 5.7 - PROCESSO “CRIA ENTIDADE REMOTA”



A elipse 1 da fig. 5.8 indica os nós do grafo de cena que representam os objetos criados pela execução desse processo.

FIGURA 5.8 - NÓS DO GRAFO DE CENA QUE REPRESENTAM OS OBJETOS CRIADOS PELO PROCESSO “CRIA ENTIDADE REMOTA”.



Exemplificando o recebimento de um `CreateEntityPdu` informando acerca da criação de uma nova entidade, no caso tipo cone, tem-se a implementação do quadro 5.4.

QUADRO 5.4 - EXEMPLO DE IMPLEMENTAÇÃO DO PROCESSO “CRIA ENTIDADE TIPO CONE”

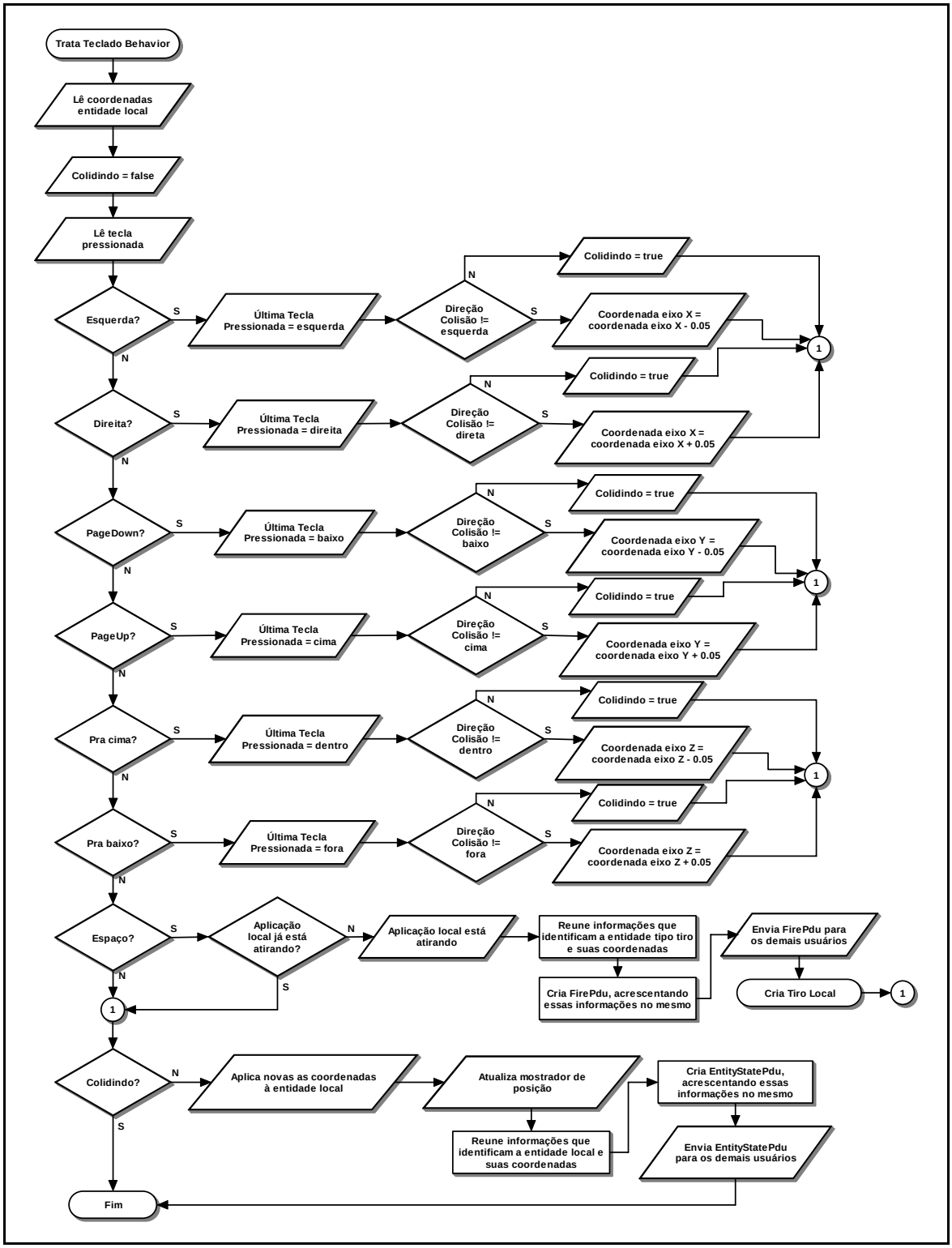
```
...
...
Float raio = new Float(0.2f);
Float altura = new Float(0.35f);
Cone objCone = new Cone(raio.floatValue(), altura.floatValue());
objCone.setAppearance(0, ap1);
objCone.setAppearance(1, ap2);
Transform3D t3d = new Transform3D();
t3d.setTranslation(new Vector3d(coordenadas[0], coordenadas[1],
  coordenadas[2]));
TransformGroup tg = new TransformGroup(t3d);
//adiciona o objeto Cone ao seu TransformGroup
tg.addChild(objCone);
//seta algumas permissões:
//ALLOW_TRANSFORM_WRITE e READ permitem que as informações
//de transformação do objeto sejam acessadas e modificadas
//mesmo após ele se tornar “vivo”
//ENABLE_COLLISION_REPORTING permite que esse TranformGroup
//seja reportado pelo método getTriggeringPath (quadro 5.8)
tg.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
tg.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
tg.setCapability(TransformGroup.ENABLE_COLLISION_REPORTING);
//Pega as informações que identificam a entidade
//e as acrecescenta ao TransformGroup dessa entidade
EntityID entId = new EntityID();
```

```
    entId = (EntityID)cePDU.getOriginatingEntityID().clone();
    tg.setUserData(entId);
    //desatacha BranchGroup de entidades remotas
    atachDetachBGRaiz("desatachar", bgEntidadesRemotas);
    //acrescesta identificação da entidade na tabela de entidades
    tabelaIdEntidades.add(entId);
    //adiciona o TransformGroup da entidade ao BranchGroup
    //de entidades remotas
    bgEntidadesRemotas.insertChild(tg, tabelaIdEntidades.size()-1);
    //atacha BranchGroup novamente
    atachDetachBGRaiz("atachar", bgEntidadesRemotas);
    //chama função que atualiza o painel mostrador de usuários
    atualizaTabelaEntidades(entId.getApplicationID().toString(),
         coordenadas, "Adicionar", "CEPDU");
}
...
...
```

5.2.5 TRATA TECLADO *BEHAVIOR*

A classe `TrataTecladoBehavior` é herdeira da classe `KeyAdapter` e implementa o método `keyPressed`. Dessa forma, um objeto dessa classe pode ser adicionado através do método `addKeyListner` a qualquer objeto da interface visual do Java do tipo `Container`. Como o objeto da classe `Canvas3D`, que é onde são apresentadas as imagens do ambiente virtual, é desse tipo, pode-se adicionar um objeto da classe `TrataTecladoBehavior` a um objeto da classe `Canvas3D` e sempre que uma tecla for pressionada, enquanto o foco estiver sobre o `Canvas3D`, o método `keyPressed` implementado nessa classe será chamado. A fig. 5.9 apresenta a lógica geral implementada no método `keyPressed` da classe `TrataTecladoBehavior` e o quadro 5.5 exemplifica a implementação do mesmo.

FIGURA 5.9 - LÓGICA DO MÉTODO KEYPRESSED IMPLEMENTADO NA CLASSE TRATATECLADOBEHAVIOR



QUADRO 5.5 - CÓDIGO EXEMPLIFICANDO O MÉTODO KEYPRESSED

```
public void keyPressed(KeyEvent k) {
  //Pega tecla pressionada
  int c = k.getKeyCode();
  tg.getTransform(t3d);
  t3d.get(v3d);
  //Pega coordenadas da entidade local
```

```
    v3d.get(coordenadas);
    boolean estaColidindo = false;
      //verifica o tipo de tecla pressionada
      switch (c) {
        //caso seta para esquerda
        case KeyEvent.VK_LEFT:
          //registra última tecla pressionada. Necessário para
          //tratamento de colisão
          setUltimaTeclaPressionada("esquerda");
          //Se a direção de colisão estiver na esquerda, não aceita
          //mais seta pra esquerda e seta colisão, caso contrário
          //decrementa 0.05m da coordenada do eixo X
          if (pai.getDirecaoColisao() != "esquerda")
            coordenadas[0] -= 0.05;
          else
            estaColidindo = true;
          break;
        ...
        ...
        //Idem para as teclas   seta direita  = + 0.05m eixo X
        //                      Page Down     = - 0.05m eixo Y
        //                      Page Up       = + 0.05m eixo Y
        //                      seta cima     = - 0.05m eixo Z
        //                      seta baixo    = + 0.05m eixo Z
        ...
        ...
        //caso barra de espaço
        case KeyEvent.VK_SPACE:
          //verifica se já não está atirando
          if (!getAtirando()) {
            //agora está atirando
            setAtirando(true);
            //função que cria um FirePdu e acrescenta nele
            //as informações que irão identificar quem está
            //disparando esse tiro e a própria identificação
            //desse tiro, pois o tiro nada mais é do que outra
            //entidade a ser representada no ambiente virtual.
            //Esse FirePdu é então enviado para todos os usuários
            disparaTiro(coordenadas);
          }
          break;
      }
      //caso não esteja colidindo
      if (!estaColidindo) {
        //atribui coordenadas alteradas em virtude do pressionamento
        //da tecla para a entidade local
        v3d.set(coordenadas);
        t3d.set(v3d);
        tg.setTransform(t3d);
        //função que cria um EntityStatePdu e acrescenta nele
        //as informações que irão identificar que entidade
        //é essa a qual corresponde esse EntityStatePdu
        //e as suas novas coordenadas. Esse EntityStatePdu
        //é então enviado para todos os usuários, para que os mesmos
        //possam atualizar a posição da entidade nas suas respectivas
        //representações do ambiente virtual
        informaAtualizacao(t3d, v3d);
        //função que atualiza o mostrador de posição da entidade local
        atualizaMostradorPosicao(coordenadas);
      }
    }
  }
```

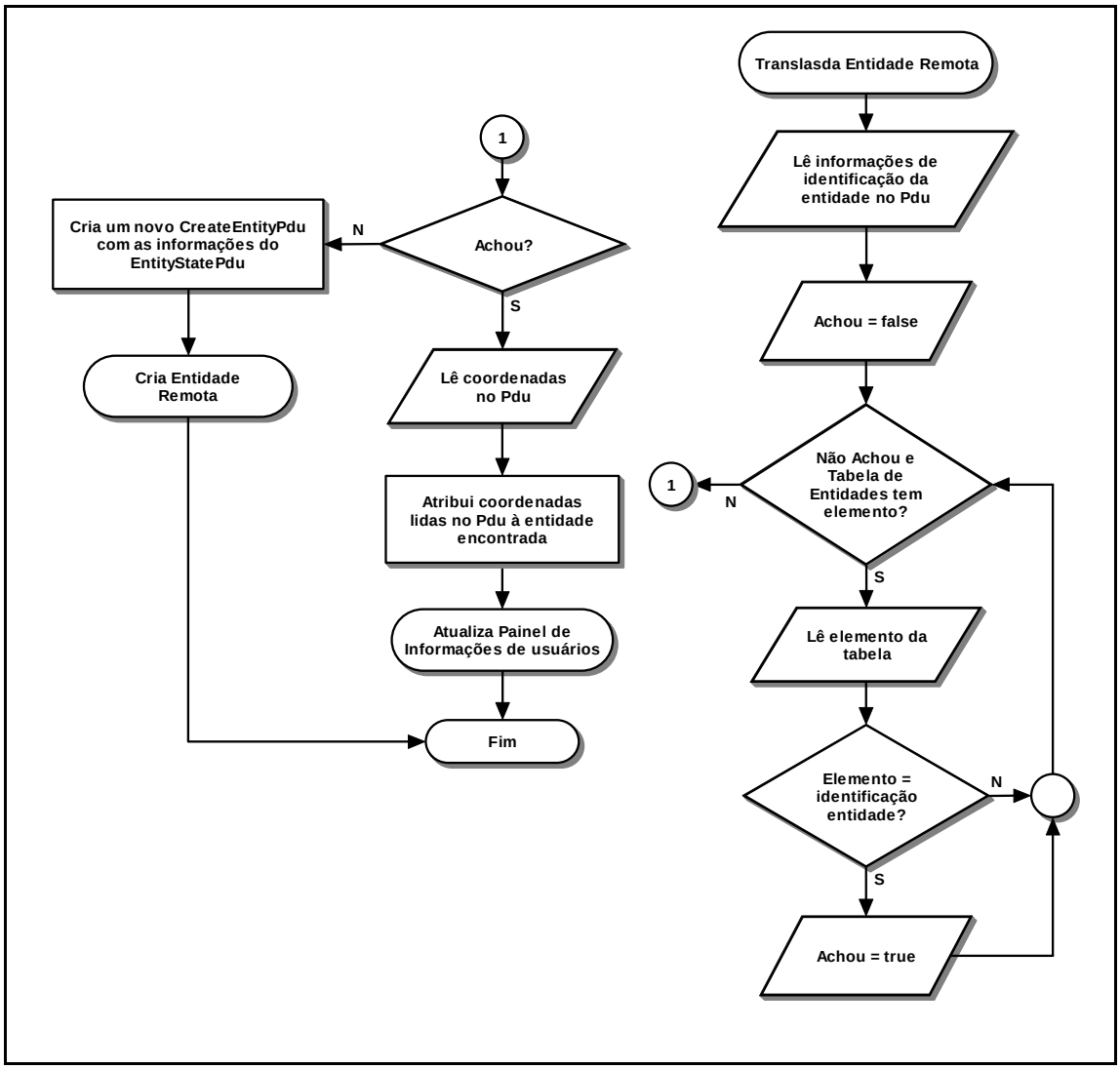
5.2.6 TRANSLADA ENTIDADE REMOTA

Conforme visto na classe `TrataTecladoBehavior`, sempre que um usuário pressionar as teclas seta para baixo, seta para cima, seta para direita, seta para esquerda, *page down* ou *page up*, ele estará realizando uma transformação de translação sobre a entidade a qual ele controla. Como forma de manter os demais usuários atualizados a respeito dessa atualização, um `EntityStatePdu` é criado a cada transformação de translação e enviado aos demais usuários. Esses, por sua vez, utilizam a função “Translada entidade remota” para encontrar a entidade referenciada pelo `EntityStatePdu` recebido e aplicar as novas informações de localização, também contidas no `EntityStatePdu`, a essa entidade.

Um detalhe importante deve ser observado nesse processo. Caso uma aplicação receba um `EntityStatePdu` com a identificação de uma entidade a qual essa aplicação ainda não criou, esse processo se encarrega de criar um `CreateEntityPdu` com base nas informações contidas no `EntityStatePdu` e chamar um método local para criar essa entidade ainda não criada no ambiente virtual.

Essa situação é muito comum no caso de usuários que entram em um ambiente virtual já populado por outros usuários. Logicamente, esse usuário que está entrando no ambiente virtual não vai receber um `CreateEntityPdu` dos demais usuários, pois esses somente o enviam quando entram no ambiente virtual. A única informação que esse novo usuário irá receber são os `EntityStatePdu's` referentes à atualização de posição desses usuários que já estavam no ambiente. Resumindo, caso um `EntityStatePdu` recebido faça referência a uma entidade ainda não representada no ambiente, ela é automaticamente criada. A fig. 5.10 apresenta o fluxograma desse processo. Na seqüência, o quadro 5.6 exemplifica a implementação do mesmo.

FIGURA 5.10 - PROCESSO “TRANSLADA ENTIDADE REMOTA”



QUADRO 5.6 - IMPLEMENTAÇÃO EXEMPLO DO PROCESSO “TRANSLADA ENTIDADE REMOTA”

```
...
...
void transladaEntidade(EntityStatePdu esPDU) {
  double[] coordenadas = new double[3];
  //lê coordenadas do EntityStatePdu
  coordenadas[0] = esPDU.getEntityLocationX();
  coordenadas[1] = esPDU.getEntityLocationY();
  coordenadas[2] = esPDU.getEntityLocationZ();
  //lê identificação da entidade
  long origSiteId = esPDU.getEntityID().getSiteID().longValue();
  long origAppId = esPDU.getEntityID().getApplicationID().longValue();
  long origEntityId = esPDU.getEntityID().getEntityID().longValue();
  int entityId = 0;
  boolean achou = false;
  EntityID entID = new EntityID();
  //percorre a tabela de entidade à procura da entidade
  //correspondente a essa do EntityStatePdu
  for (int i = 0; i < tabelaIdEntidades.size(); i++) {
    entID = (EntityID)tabelaIdEntidades.get(i);
    //se a entidade for igual
    if (((entID.getSiteID().longValue() == origSiteId) &&
      (entID.getApplicationID().longValue() == origAppId)) &&
```

```
          (entID.getEntityID().longValue() == origEntityId)) {
          entityId = i;
          achou = true;
        }
      }
      //caso não tenha achado a entidade é preciso criá-la primeiro
      if (!achou) {
        //cria um CreateEntityPdu para poder chamar o método
        //criaEntidadeRemota
        CreateEntityPdu cePDU = new CreateEntityPdu();
        //seta identificação da entidade a ser criada
        cePDU.setOriginatingEntityID(new
             EntityID((short)origSiteId,(short)origAppId,(short)1));
        criaEntidadeRemota(cePDU, coordenadas);
        return;
      }
      //acessa o TransformGroup da entidade encontrada
      tgAux = (TransformGroup) bgEntidades.getChild(entityId);
      tgAux.getTransform(t3dAux);
      t3dAux.setTranslation(new Vector3d(esPDU.getEntityLocationX(),
           esPDU.getEntityLocationY(), esPDU.getEntityLocationZ()));
      //seta as novas posições
      tgAux.setTransform(t3dAux);
      //chama função que atualiza o painel mostrador de usuários
      atualizaTabelaEntidades(esPDU.getEntityID().getApplicationID().
        toString(), coordenadas, "Atualizar", "ESPDU");
    }
    ...
    ...
```

5.2.7 *HEARTBEAT BEHAVIOR*

Os objetos da classe `HeartBeatBehavior`, seguindo a filosofia de funcionamento dos mecanismos de *heartbeat* em ambiente virtuais distribuídos, visto na seção 2.5.4, tem por finalidade ficar enviando um `EntityStatePdu` em um determinado intervalo de tempo, visando manter os novos usuários, que entram no ambiente virtual, informados a cerca da existência de entidades que não estejam enviando `EntityStatePdu's` pelo motivo de não estarem modificando suas posições.

Essa classe é herdeira da classe `Behavior` da API do Java3D e implementa os métodos `initialize` e `processStimulus`. No método `initialize` é definido o critério que dispara esse comportamento, no caso a passagem de 20 segundos de tempo, e no método `processStimulus` são implementadas as ações que devem ser executadas quando o comportamento é disparado.

É interessante lembrar que na criação de um objeto dessa classe, exemplificado nas linhas de código do quadro 5.2, é passado como parâmetro um objeto referente ao `TransformGroup` da plataforma de visão, do qual esse método irá retirar as informações

de localização da entidade para poder acrescentá-las ao `EntityStatePdu` e enviar para os demais usuários. A fig. 5.11 ilustra o funcionamento geral desse comportamento e na seqüência o quadro 5.7 exemplifica a implemetação do mesmo.

FIGURA 5.11 - LÓGICA DO COMPORTAMENTO IMPLEMENTADO NA CLASSE HEARTBEATBEHAVIOR

QUADRO 5.7 - EXEMPLO DE IMPLEMENTAÇÃO DOS MÉTODOS `INITIALIZE` E `PROCESSSTIMULUS` DA CLASSE `HEARTBEATBEHAVIOR`

```
...
...
//método initialize onde se define qual critério fará
//com que esse comportamento seja ativado, nesse caso,
//após a passagem de 20 segundos o método processStimulus
//dessa classe será chamado
public void initialize() {
   this.wakeupOn(new WakeupOnElapsedTime(20000));
}

//método processStimulus que é chamado sempre que esse
//comportamento for acionado
public void processStimulus (Enumeration criteria) {
   //o objeto tg aqui utilizado é inicializado no constructor
   //dessa classe com uma referência para o TransformGroup da
   //plataforma de visão, a qual é passada como parâmetro no
   //ato da criação do objeto da classe HearBeatBehavior, e do
   //qual será lida as informações de localização da entidade
   tg.getTransform(t3d);
   //coloca as informaçõeas que identificam a entidade no EstityStatePdu
   esPDU.setEntityID(siteId,appId,(short)1);
   t3d.get(v3f);
   //coloca as informações de localização da entidade
   esPDU.setEntityLocationX(v3f.x);
```

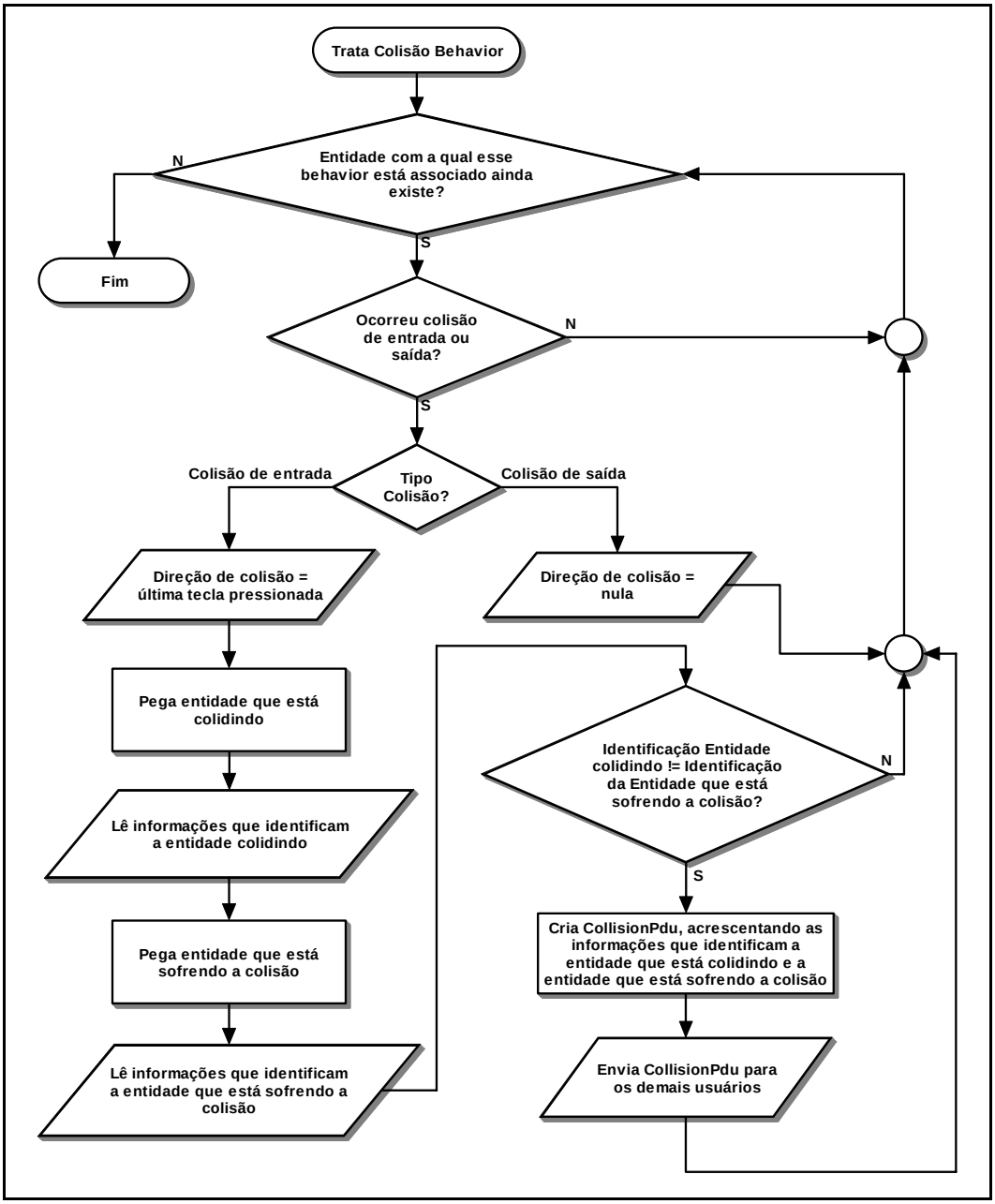
```
    esPDU.setEntityLocationY(v3f.y);
    esPDU.setEntityLocationZ(v3f.z);
    esPDU.setExemplar(esPDU);
    //envia esse EntityStatePdu para os demais usuários
    bsb.sendPdu(esPDU.getExemplar(), enderecoBroadcast, 8133);
    //define novamente que esse comportamento seja disparado
    //após a passagem de 20 segundos. Isso é necessário pois se isso
    //não for feito ao final do método processStimulus, o comportamento
    //será executado somente uma vez
    this.wakeupOn(new WakeupOnElapsedTime(20000));
  }
  ...
  ...
```

5.2.8 TRATA COLISÃO *BEHAVIOR*

O objeto dessa classe é utilizado para executar uma tarefa específica quando o objeto ao qual ele está associado entra ou sai de um estado de colisão com outro objeto. Essa classe é implementada de forma semelhante a classe `HeartBeatBehavior`. A principal diferença é na definição de quando esse comportamento é disparado.

A lógica implementada no método `processStimulus` permite tratar a colisão gerada pela movimentação da entidade no ambiente virtual, evitando que uma entidade passe por dentro da outra. No caso de colisões com tiros ela permite apenas reportar a ocorrência da colisão dos tiros com as entidades. A fig. 5.12 mostra o fluxograma referente à logica do comportamento implementado nessa classe.

FIGURA 5.12 - LÓGICA DO COMPORTAMENTO IMPLEMENTADO NA CLASSE TRATACOLISAOBEHAVIOR



O quadro 5.8 apresenta um exemplo de código implementando os métodos `initialize` e `processStimulus` dessa classe.

QUADRO 5.8 - IMPLEMENTAÇÃO DOS MÉTODOS INITIALIZE E PROCESSSTIMULUS DA CLASSE TRATACOLISAOBEHAVIOR

```
...
...
//o método initialize dessa classe define que a ativação
//desse comportamento ocorrerá baseado num conjunto de duas
//condições: quando o objeto com o qual esse comportamento
//está associado entra em estado de colisão com outro objeto
//ou sai desse estado de colisão. Os objetos da classe
//WakeupOnCollisionEntry e WakeupOnCollisionExit levam como parâmetro
//o TransformGroup da entidade com a qual esse comportamento
//está associado e a forma como será controlada a colisão. Nesse caso
```

```
  //a colisão será controlada analizando a geometria do objeto e não
  //o envelope que o envolve
  public void initialize() {
    WakeupCriterion[] wc = new WakeupCriterion[2];
    wc[1] = new WakeupOnCollisionEntry(tg,
                WakeupOnCollisionEntry.USE_GEOMETRY);
    wc[0] = new WakeupOnCollisionExit(tg,
                WakeupOnCollisionExit.USE_GEOMETRY);
    this.wakeupOn(new WakeupOr(wc));
  }

  //método chamado na ocorrência de uma entrada ou saída de um
  //estado de colisão
  public void processStimulus (Enumeration criteria) {
    //o parâmetro criteria traz um conjunto de objetos
    //referentes aos motivos pelos quais esse comportamento
    //foi disparado. Nesse caso só pode ser dois: entrada de colisão
    //(WakeupOnCollisionEntry) e saída de colisão (WakeuponCollisionExit)
    while (criteria.hasMoreElements()) {
      Object proximoElemento = criteria.nextElement();
      //se for entrada de colisão
      if (proximoElemento.getClass().getName().trim() ==
           "javax.media.j3d.WakeupOnCollisionEntry") {
        //seta como direção de colisão a última tecla pressionada
        //dessa forma futuros pressionamentos dessa tecla não
        //serão tratados, evitando com que a entidade mova-se nessa
        //direção
        setDirecaoColisao(getUltimaTeclaPressionada());
        WakeupOnCollisionEntry entradaColisao =
             (WakeupOnCollisionEntry)proximoElemento;
        //o método getTriggeringPath permite ter acesso ao ramo
        //da árvore de objetos correspondente ao objeto que está
        //colidindo. Dessa forma é possível acessar o mesmo e
        //consequentemente a identificação dessa entidade
        for (int i = 0; i <
             entradaColisao.getTriggeringPath().nodeCount(); i++) {
          Object noTemporario =
            entradaColisao.getTriggeringPath().getNode(i);
          //como os TraformGroup's das entidades inseridas no ambiente
          //virtual são os únicos que possuem as permissões de
          //ENABLE_COLLISION_REPORTING, ao se percorrer esse ramo
          //da árvore obtida com o getTriggeringPath, esses serão
          //os únicos que não terão uma referência null
          if (noTemporario != null) {
            TransformGroup tgTemp = (TransformGroup)noTemporario;
            //pega identificação da entidade associada com o
            //TransformGroup
            EntityID collidingEntity = (EntityID)tgTemp.getUserData();
            //pega identificação dessa própria entidade que está
            //sofrendo a colisão
            EntityID issuingEntity = (EntityID)tg.getUserData();
            //verifica se as entidades colidindo não pertencem
            //a mesma aplicação, caso que ocorre qdo uma entidade
            //dispara um tiro
            if (issuingEntity.getApplicationID().intValue() !=
              collidingEntity.getApplicationID().intValue())
              //chama função que cria um CollisionPdu, acrescentando
              //as identificações das entidades que estão colidindo,
              //enviando esse Pdu para os demais usuários
              informaColisao(issuingEntity, collidingEntity);
          }
        }
```

```
        //se não for entrada de colisão é saída de colisão
        //nesse caso define-se como sendo nula a direção de colisão
        //já que não está mais havendo colisão
        } else {
           setDirecaoColisao("nula");
        }
     }
     //ativa os disparadores de comportamento novamente
     WakeupCriterion[] wc = new WakeupCriterion[2];
     wc[1] = new WakeupOnCollisionEntry(tg,
          WakeupOnCollisionEntry.USE_GEOMETRY);
     wc[0] = new WakeupOnCollisionExit(tg,
          WakeupOnCollisionExit.USE_GEOMETRY);
     this.wakeupOn(new WakeupOr(wc));
  }
  ...
  ...
```

5.2.9 CRIA TIRO LOCAL E REMOTO

Os processos “Cria tiro local” e “Cria tiro remoto” são utilizados para criar entidades no ambiente virtual que representam os tiros disparados pelos usuários. O primeiro processo é chamado na própria aplicação do usuário que está disparando, enquanto que o segundo é chamado nas outras aplicações ao serem notificadas, através de um `FirePdu`, sobre a ocorrência de um disparo por outros usuários.

Os dois processos são bastante semelhantes, diferindo somente em alguns pontos. Em se tratando do processo “Cria tiro local”, à entidade que representa o tiro disparado é acrescentado o comportamento que anima o tiro no ambiente virtual (a classe `AnimaTiroBehavior` será vista nas seções seguintes), sendo que essa entidade é acrecescentada ao `BranchGroup` de tiros locais.

Já no caso do processo “Cria tiro remoto”, à entidade criada para representar um tiro disparado por outro usuário não é acrescentado o comportamento anima tiro, visto que a movimentação desse tiro será feita através do recebimento de `EntityStatePdu’s` por parte da aplicação do usuário que disparou o tiro. Também, essas entidades serão acrescentadas ao `BranchGroup` de tiros remotos. As figs. 5.13 e 5.14 ilustram os fluxogramas referentes a esses processos.

FIGURA 5.13 - FLUXOGRAMA DO PROCESSO “CRIA TIRO LOCAL”

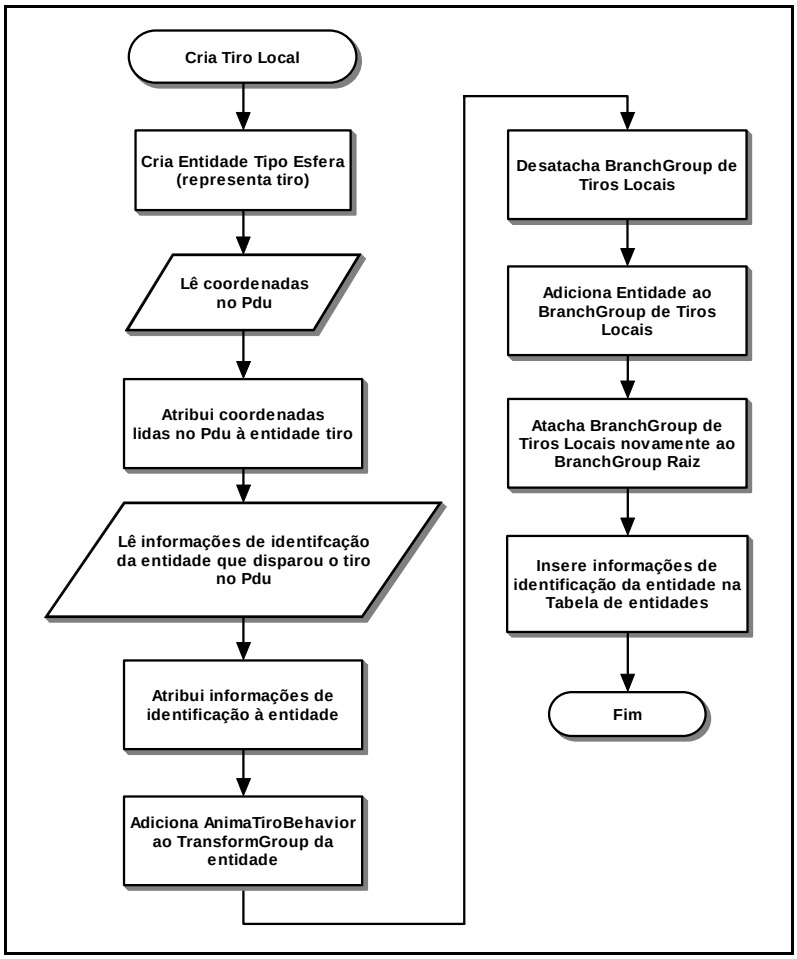
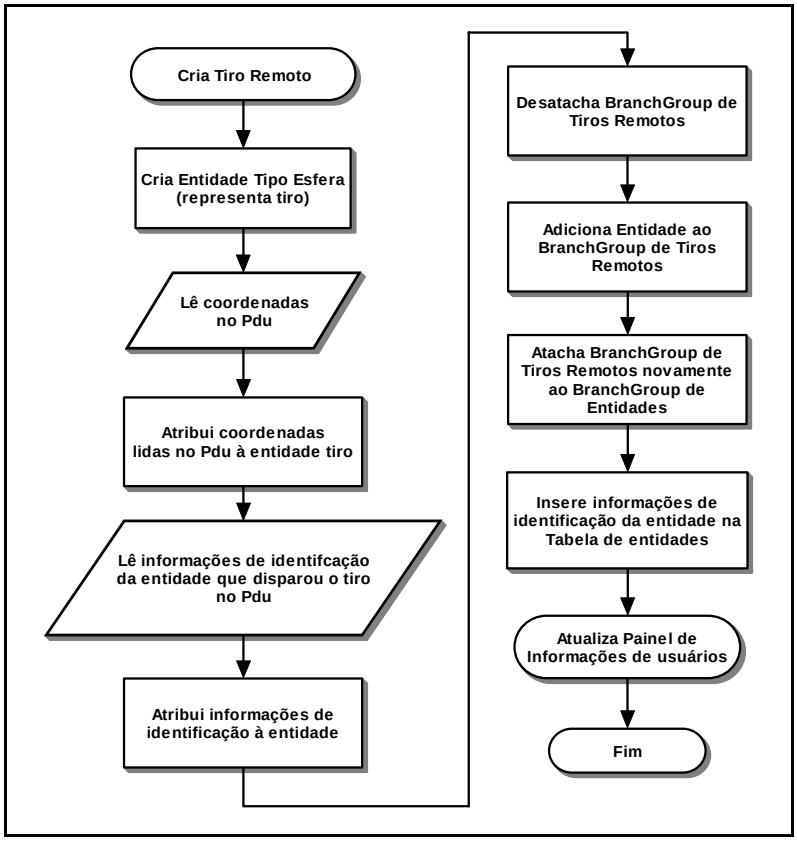
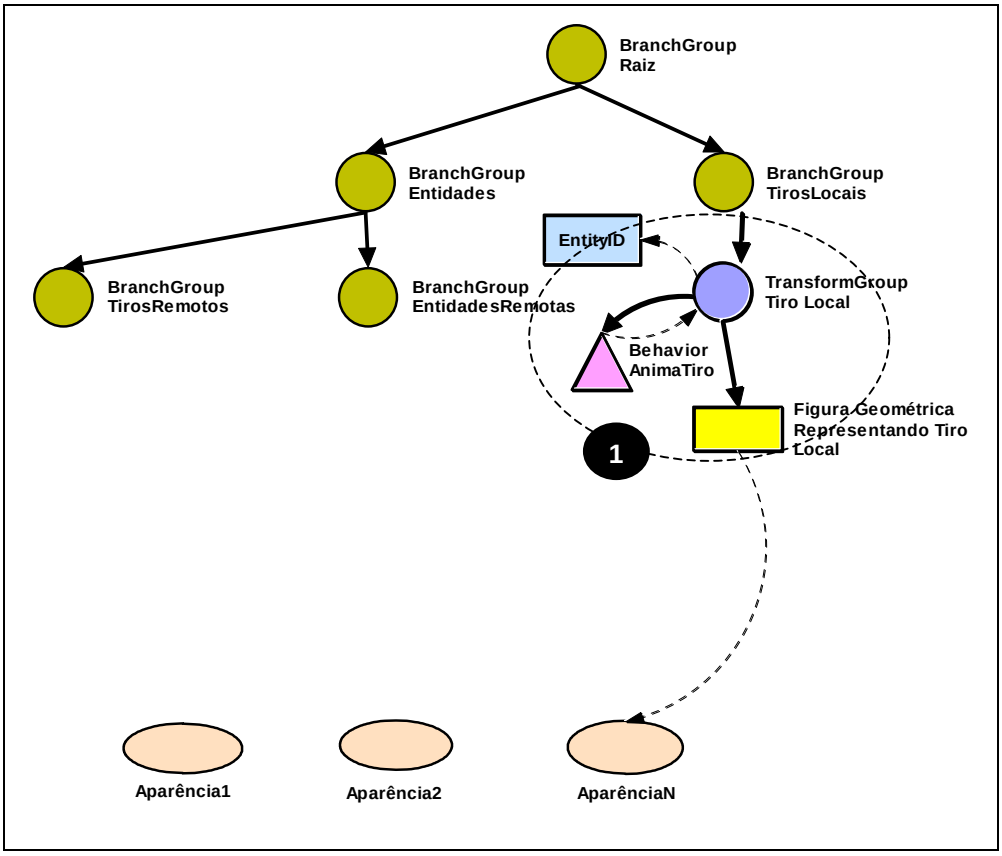


FIGURA 5.14 - FLUXOGRAMA DO PROCESSO “CRIA TIRO REMOTO”



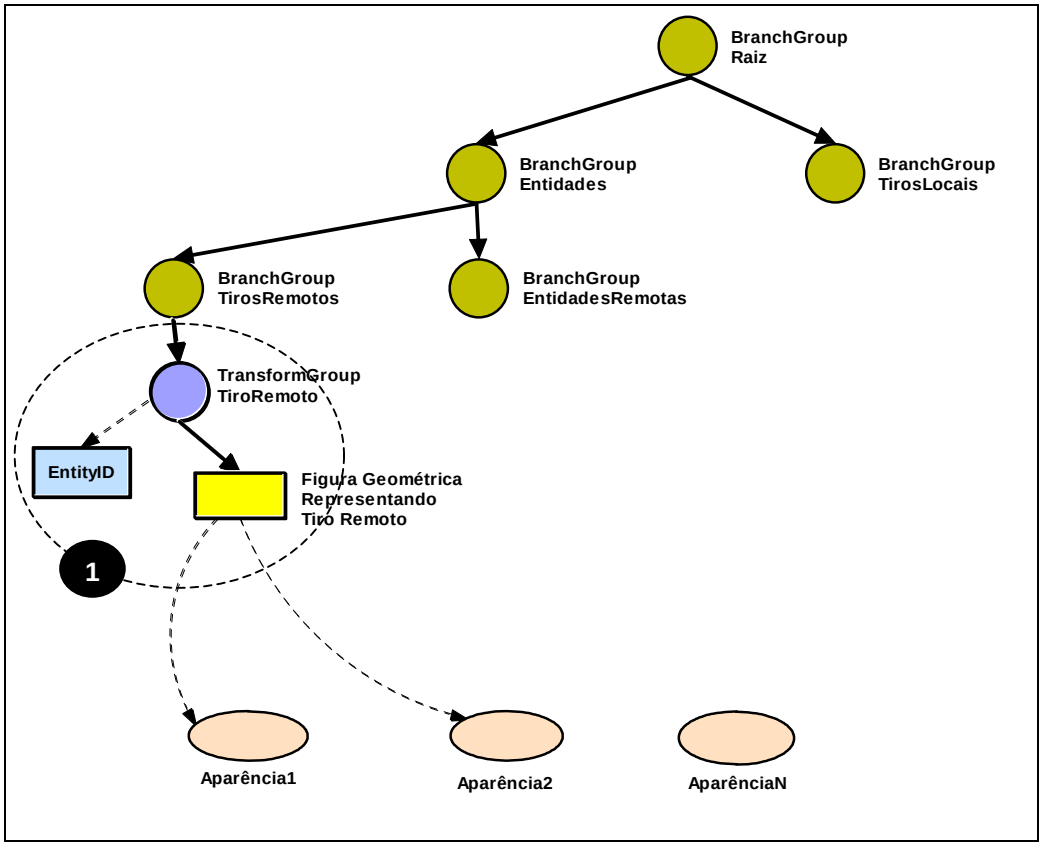
A elipse 1 da fig. 5.15 indica os nós do grafo de cena que representam os objetos criados pela execução do processo de criação de um tiro local.

FIGURA 5.15 - NÓS DO GRAFO DE CENA QUE REPRESENTAM OS OBJETOS CRIADOS PELO PROCESSO “CRIA TIRO LOCAL”



Na seqüência, a elipse 1 da fig. 5.16 indica os nós que representam os objetos criados pela execução do processo de criação de um tiro remoto.

FIGURA 5.16 - NÓS DO GRAFO DE CENA QUE REPRESENTAM OS OBJETOS CRIADOS PELO PROCESSO “CRIA TIRO REMOTO”



Já que os dois processos são muito semelhantes, no quadro 5.9 será apresentado somente a implementação do processo “Cria tiro local”. Para o processo “Cria tiro remoto” as únicas diferenças são que à entidade criada para representar o tiro disparado por outro usuário não é acrescentado o objeto da classe `AnimaTiroBehavior` e essa entidade é incluída no `BranchGroup` de tiros remotos.

QUADRO 5.9 - IMPLEMENTAÇÃO DO PROCESSSO “CRIA TIRO LOCAL”

```
...
...
void criaTiroLocal(FirePdu frPDU) {
  //uma esfera de raio 0.1m representa o tiro
  Float raio = new Float(0.1f);
  Sphere objEsfera = new Sphere(raio.floatValue(), ap6);
  //pega as coordenadas iniciais do tiro do FirePdu
  double tiroPosicaoX = frPDU.getLocationInWorldCoordinate().getX();
  double tiroPosicaoY = frPDU.getLocationInWorldCoordinate().getY();
  double tiroPosicaoZ = frPDU.getLocationInWorldCoordinate().getZ();
  Transform3D t3d = new Transform3D();
  //aplica coordenadas na entidade tiro
  t3d.setTranslation(new Vector3d(tiroPosicaoX, tiroPosicaoY,
       tiroPosicaoZ));
  TransformGroup tg = new TransformGroup(t3d);
  tg.addChild(objEsfera);
  tg.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
  tg.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
  tg.setCapability(TransformGroup.ENABLE_COLLISION_REPORTING);
```

```
    EntityID entId = new EntityID();
    entId = (EntityID)frPDU.getFiringEntityID().clone();
    //adiciona informação que identifica essa
    //entidade tipo tiro
    tg.setUserData(entId);
    //adiciona comportamento anima tiro behavior a essa entidade tipo
    //tiro, reponsável por relizar a movimentação da entidade, simulando
    //um tiro em movimento
    AnimaTiroBehavior animaTiroBehavior = new
         AnimaTiroBehavior(getProto(), tg);
    animaTiroBehavior.setSchedulingBounds(new BoundingSphere(new
         Point3d(tiroPosicaoX, tiroPosicaoY, tiroPosicaoZ), 0.2));
    //adiciona comportamento AnimaTiroBehavior ao TranformGroup
    //da entidade tipo tiro
    tg.addChild(animaTiroBehavior);
    //adiciona identificação da entidade na tabela de identificações
    //de entidades
    tabelaIdEntidades.add(entId);
    //desatacha BranchGroup de tiros locais, já que
    //não é possível adicionar objetos ao mesmo, estando esse vivo
    atachDetachBranchGroup("desatachar", bgTirosLocais);
    //adiciona entidade ao BrachGroup de tiros locais
    bgTirosLocais.insertChild(tg, tabelaIdEntidades.size()-1);
    //atacha BranchGruop novamente
    atachDetachBranchGroup("atachar", bgTirosLocais);
}
...
...
```

5.2.10 ANIMA TIRO *BEHAVIOR*

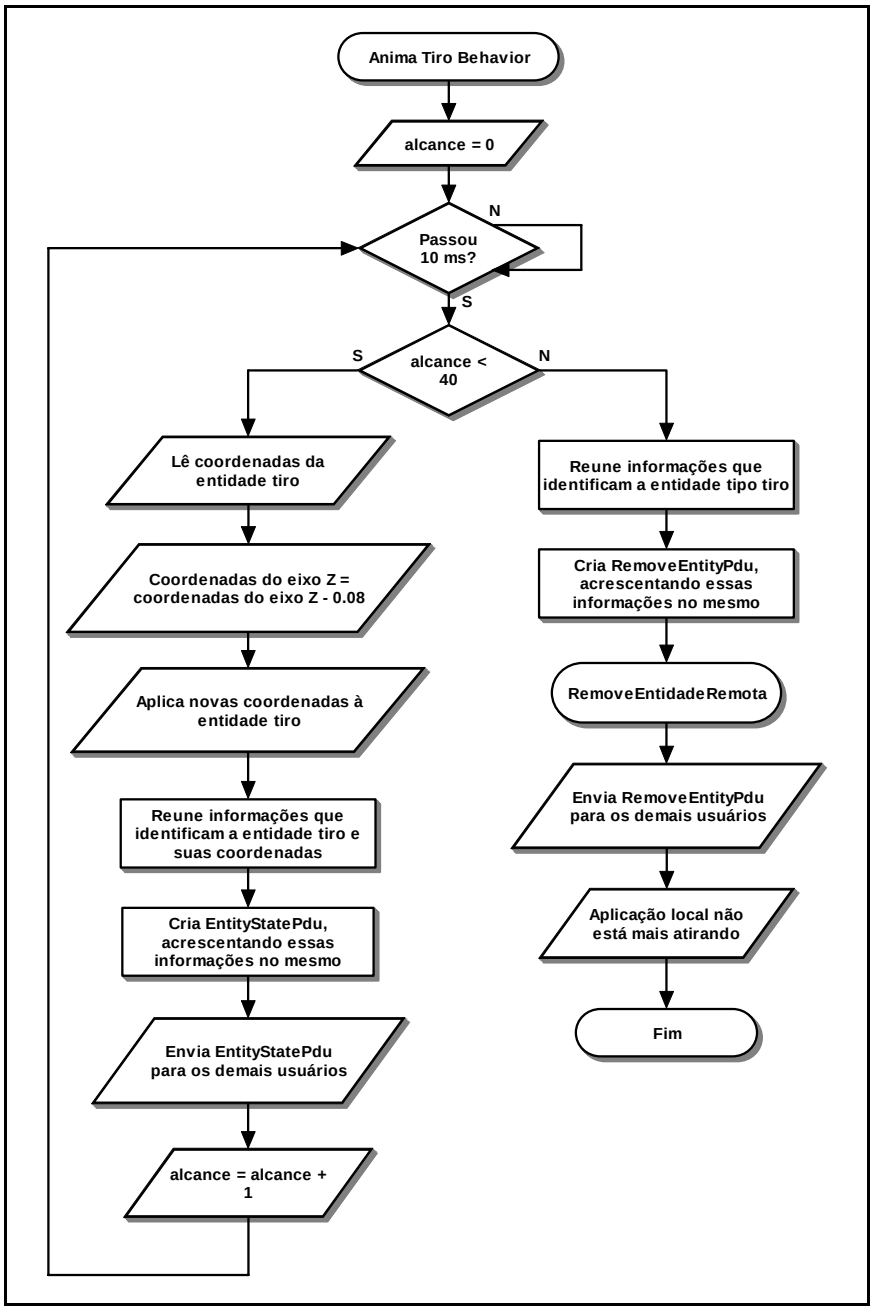
Os objetos instanciados da classe `AnimaTiroBehavior` tem por objetivo controlar a movimentação das entidades tipo tiro local criadas quando o usuário pressiona a barra de espaço. Conforme visto na seção anterior, ao se intanciar um objeto dessa classe se passa como um dos parâmetros uma referência a um objeto da classe `TransformGroup`, sobre o qual esse comportamento irá agir. Depois se adiciona esse comportamento ao `TransformGroup` da entidade que representa o tiro local.

O funcionamento desse comportamento é bem simples. Basicamente ele vai decrementando o valor da coordenada **`Z`** da entidade, num intervalo de tempo determinado, dando a impressão de que a entidade, representando o tiro, apresenta uma movimentação contínua para a frente. Enquanto está movimentando a entidade, a cada alteração da coordenada **`Z`**, um `EntityStatePdu` é criado e enviado para os demais usuários, para que o movimento da entidade tiro também aconteça nos demais usuários.

Esse comportamento se repete uma quantidade determinada de vezes. Após essa quantidade, a entidade tiro é eliminada localmente e um `RemoveEntityPdu` é criado e enviado aos demais usuários para que os mesmos também removam a entidade. Dessa forma

se cria uma espécie de alcance limitado para os tiros. Um *flag*, controlando se a entidade está atirando ou não, faz com que seja possível dar apenas um tiro por vez. A lógica de funcionamento do comportamento implementado nessa classe está ilustrada no fluxograma da fig. 5.17.

FIGURA 5.17 - FUNCIONAMENTO DO COMPORTAMENTO IMPLEMENTADO NA CLASSE `ANIMATIROBEHAVIOR`



O quadro 5.10 apresenta um exemplo de implementação dos métodos `initialize` e `processStimulus` desse comportamento.

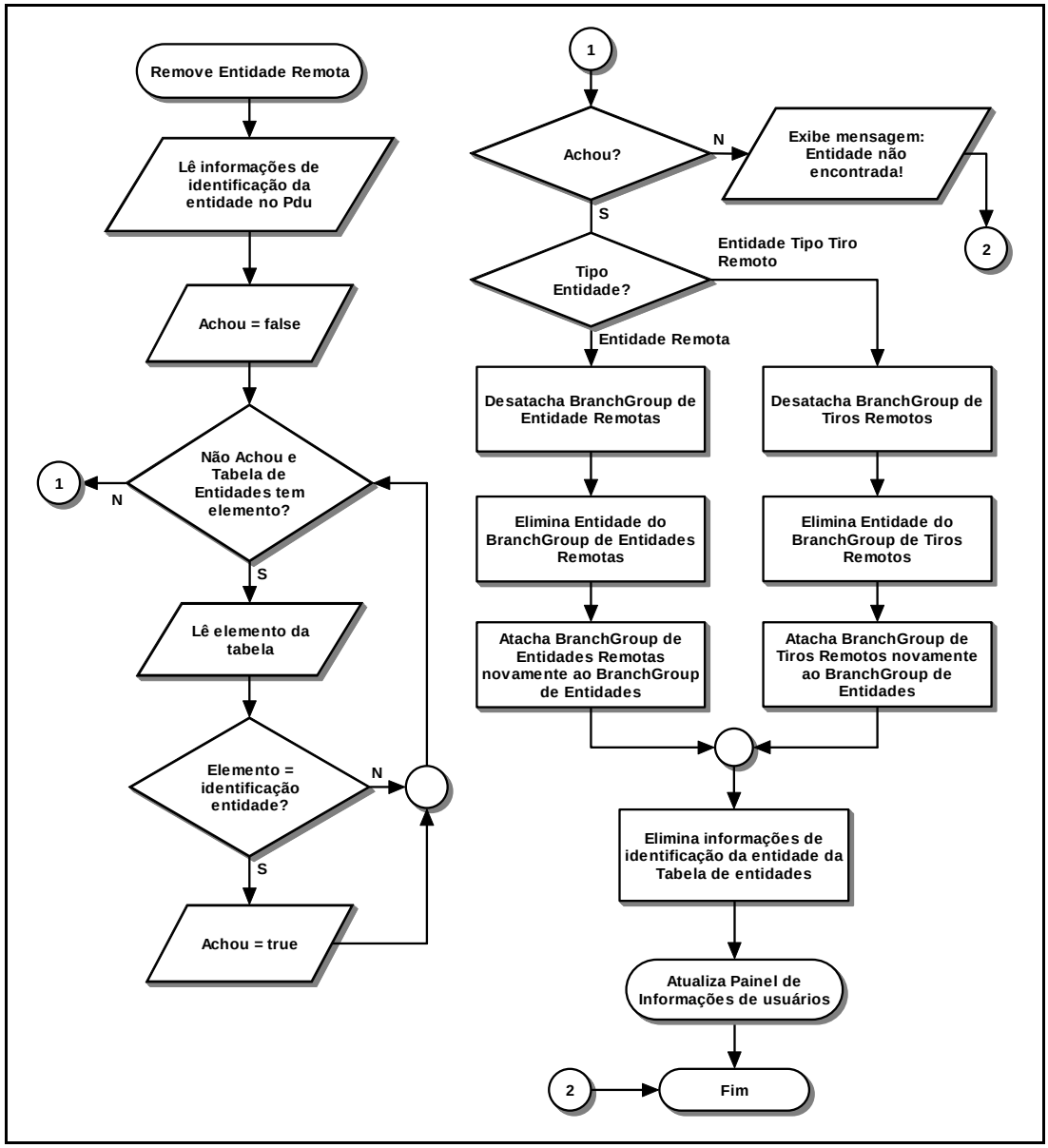
QUADRO 5.10 - IMPLEMENTAÇÃO DOS MÉTODOS INITIALIZE E PROCESSSTIMULUS DA CLASSE ANIMATIROBEHAVIOR

```
...
...
//define a condição de ativação desse comportamento
//no caso ele irá acontecer de tempos em tempos, mais
//precisamente num intervalo de tempo de 10ms
public void initialize() {
   this.wakeupOn(new WakeupOnElapsedTime(10));
}

//método processStimulus chamado a cada vez que o
//comportamento é executado
public void processStimulus (Enumeration criteria) {
   //verifica se já estourou o alcance do tiro
   if (range < 40) {
      //pega as coordenadas atuais do tiro
      tg.getTransform(t3d);
      t3d.get(v3d);
      v3d.get(coordenadas);
      //decrementa 0.08m da coordenada do eixo Z
      coordenadas[2] -= 0.08;
      v3d.set(coordenadas);
      t3d.set(v3d);
      //aplica coordenada alterada à entidade
      tg.setTransform(t3d);
      //chama função responsável por criar um EntityStatePdu
      //reunindo as informações que identificam essa entidade
      //tipo tiro e suas coordenadas, enviando-o para os demais
      //usuários
      informaAtualizacao(tg, v3d);
      //reativa condição de disparo
      this.wakeupOn(new WakeupOnElapsedTime(10));
      range++;
   //caso o range esteja estourado, deve-se remover
   //a entidade que representa o tiro, bem como criar
   //um RemoveEntityPdu para que os demais usuários sejam
   //informados e também removam a entidade
   } else {
        //cria RemoveEntityPdu
        RemoveEntityPdu rePDU = new RemoveEntityPdu();
        EntityID tempEntityID = (EntityID)tg.getUserData();
        //acrescenta informação que identifica a entidade a ser removida
        rePDU.setOriginatingEntityID(tempEntityID);
        rePDU.setExemplar(rePDU);
        //envia o RemoveEntityPdu para os demais usuários
        bsb2.sendPdu(rePDU.getExemplar(), enderecoBroadcast,
                     8133);
        //remove a entidade localmente
        removeEntidade(rePDU);
        range = 0;
        //usuário não está mais atirando
        setAtirando(false);
   }
}
...
...
```

5.2.11 REMOVE ENTIDADE REMOTA

O processo “Remove entidade remota” é utilizado, como o próprio nome sugere, para remover entidades remotas. Ele é executado como resultado do recebimento de um `RemoveEntityPdu`, por parte de outro usuário, informando a respeito da saída do usuário do ambiente virtual e consequentemente a eliminação da entidade que o representa no ambiente. Também o recebimento de um `RemoveEntityPdu` pode acontecer para informar a respeito da eliminação de entidades que representam tiros disparados por outros usuários. A fig. 5.18 ilustra o fluxograma desse processo.

FIGURA 5.18 - PROCESSO “REMOVE ENTIDADE REMOTA”



O quadro 5.11 apresenta um trecho de código exemplificando a implementação desse processo no caso da remoção de uma entidade do tipo remota.

QUADRO 5.11 - TRECHO DE CÓDIGO EXEMPLIFICANDO A REMOÇÃO DE UMA ENTIDADE DO TIPO REMOTA

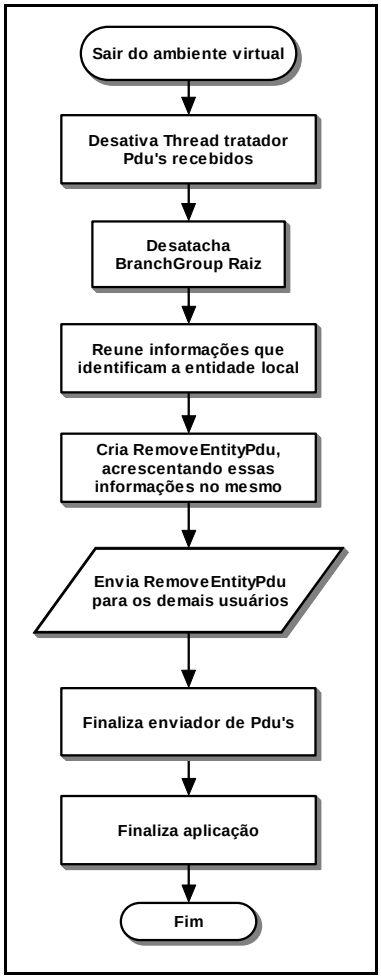
```
...
...
void removeEntidadeRemota(RemoveEntityPdu rePDU) {
  //lê as informações do RemoveEntityPdu que identificam
  //a entidade a ser removida
  long origSiteId =
    rePDU.getOriginatingEntityID().getSiteID().longValue();
  long origAppId =
    rePDU.getOriginatingEntityID().getApplicationID().longValue();
  long origEntityId =
    rePDU.getOriginatingEntityID().getEntityID().longValue();
  int entityId = 0;
  EntityID entID = new EntityID();
  boolean achou = false
  //percorre tabela de identificação de entidade
  //para achar entidade
  for (int i = 0; i < tabelaIdEntidades.size(); i++) {
    entID = (EntityID)tabelaIdEntidades.get(i);
    //se for essa a entidad a ser removida
    if (((entID.getSiteID().longValue() == origSiteId) &&
      (entID.getApplicationID().longValue() == origAppId)) &&
      (entID.getEntityID().longValue() == origEntityId)) {
      entityId = i;
      achou = true;
    }
  }
  ...
  ...
  //caso a entidade a ser removida seja do tipo
  //entidade remota e não tiro remoto
  //desatacha BranchGroup de entidades remotas
  atachDetachBGRaiz("desatachar", bgEntidadesRemotas);
  //remove entidade remota do BranchGroup
  bgEntidadesRemotas.removeChild(entityId);
  //remove identificação da entidade da tabela de entidades
  tabelaIdEntidades.remove(entityId);
  //atacha BranchGroup novamente
  atachDetachBGRaiz("atachar", bgEntidadesRemotas);
  //atualiza painel de informação de usuários
  atualizaTabelaEntidades(rePDU.getOriginatingEntityID().
    getApplicationID().toString(), coord, "Remover", "REPDU");
}
...
...
```

5.2.12 SAIR DO AMBIENTE VIRTUAL

O último processo que ainda possui relação direta com o controle do ambiente virtual é o de sair do ambiente virtual. Além de finalizar a *thread* responsável por processar os PDU's recebidos, bem como finalizar o objeto da classe `BehaviorStreamBufferUDP` utilizado para enviar PDU's, esse processo tem como principal função criar um `RemoveEntityPdu` e o enviar para os demais usuários, com a finalidade de informar sobre

a saída desse usuário do ambiente virtual, fazendo com que os demais usuários eliminem de suas representações do ambiente virtual a entidade que representa esse usuário. A fig. 5.19 apresenta o fluxograma correspondente a esse processo e o quadro 5.12 a implementação do mesmo.

FIGURA 5.19 - PROCESSO “SAIR DO AMBIENTE VIRTUAL”



QUADRO 5.12 - CÓDIGO EXEMPLIFICANDO O PROCESSO “SAIR DO AMBIENTE VIRTUAL”

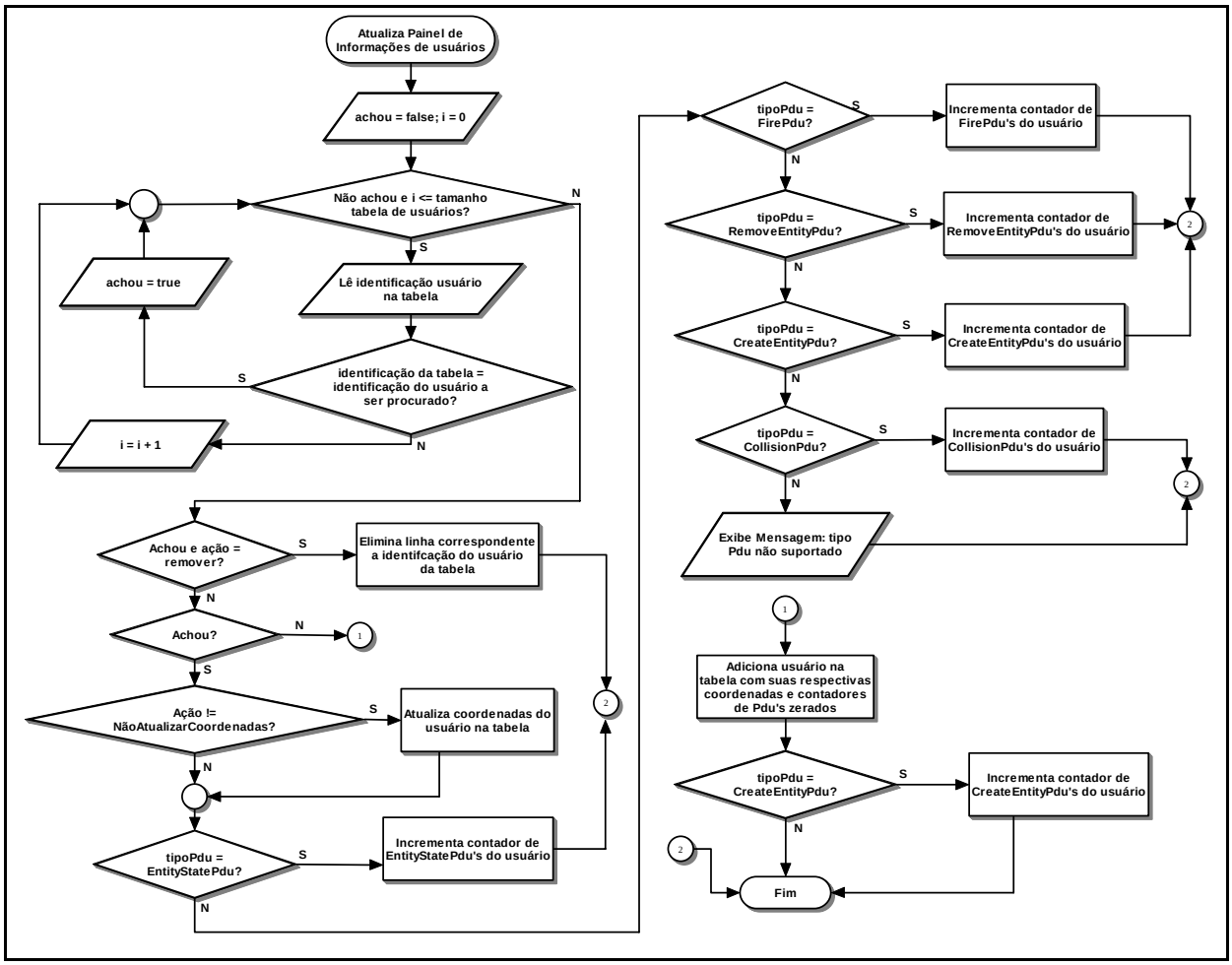
```
...
...
//para finalizar o loop run() da Thread
tratadorPdu.setVivo(false);
//desatacha o BranchGroup Raiz, dessa forma
//as imagens do ambiente páram de ser renderizadas no Canvas3D
atachDetachBGRaiz("desatachar", bgRaiz);
//cria RemoveEntityPdu para informar a respeito da saída
//do usuário
RemoveEntityPdu rePDU = new RemoveEntityPdu();
//reúne informação que identifica a entidade
rePDU.setOriginatingEntityID(new EntityID(siteId,appId,1));
rePDU.setExemplar(rePDU);
//envia RemoveEntityPdu
bsb.sendPdu(rePDU.getExemplar(), enderecoBroadcast, 8133);
//finaliza objeto BehaviorStreamBufferUDP utilizado
//para enviar PDU’s
bsb.shutdown();
```

```
    bsb.cleanup();
    ...
    ...
```

5.2.13 ATUALIZAR PAINEL DE INFORMAÇÕES DE USUÁRIOS

Por último, apenas como forma de complemento, já que esse processo não tem relação direta com o controle do ambiente virtual ou da comunicação entre os usuários, a fig. 5.20 apresenta o fluxograma do processo “Atualiza painel de informações de usuários”. A principal função dele é controlar a atualização da tabela de usuários, mostrada na interface gráfica, responsável por exibir informações sobre os usuários que estão participando do ambiente virtual.

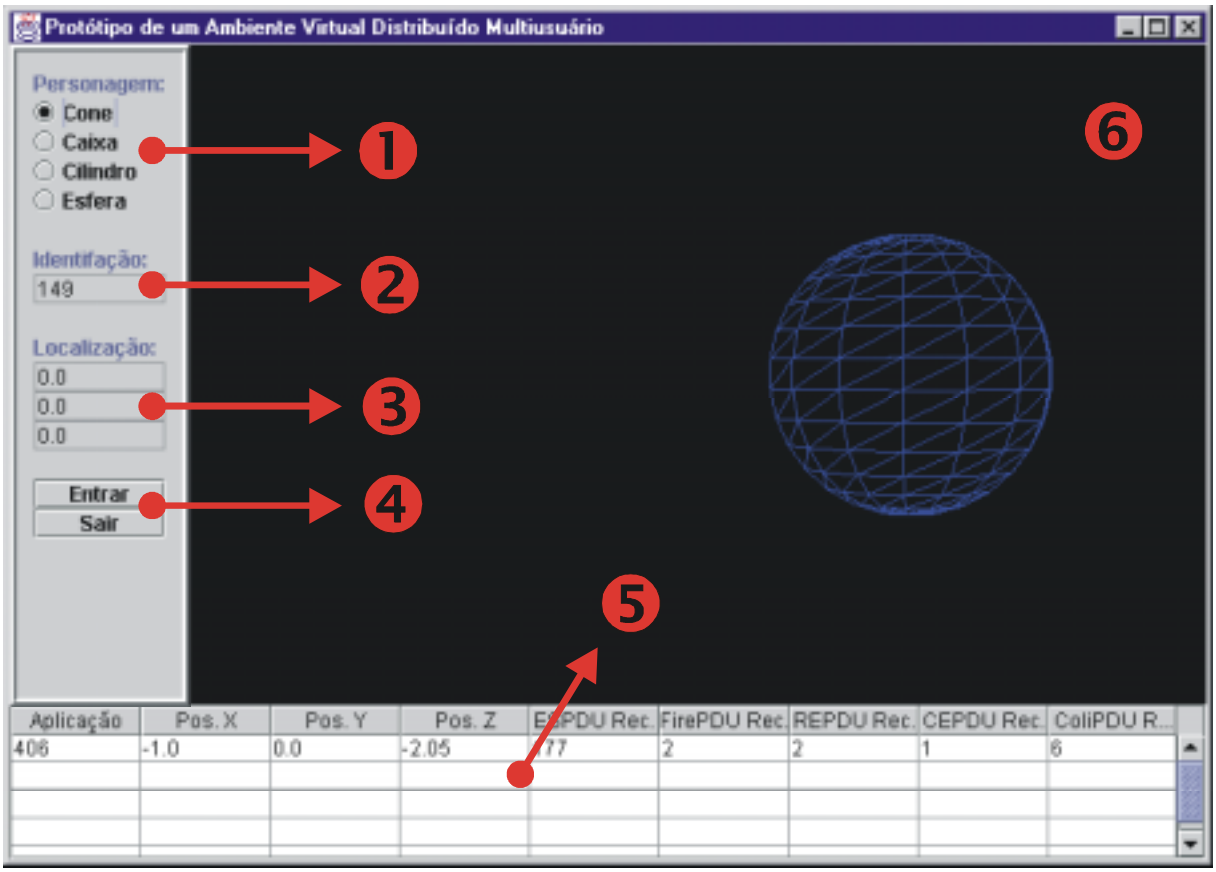
FIGURA 5.20 - PROCESSO “ATUALIZA PAINEL DE INFORMAÇÕES DE USUÁRIOS”



5.3 FUNCIONAMENTO DO PROTÓTIPO

Nesta seção será apresentada a interface principal do protótipo, bem como seu funcionamento em geral. A interface é bastante simples e possui basicamente seis pontos interessantes de serem vistos, conforme ilustrado na fig. 5.21: onde se escolhe o tipo de personagem que se quer entrar no ambiente virtual (1), um número que identifica essa aplicação no ambiente virtual (2), as coordenadas atuais referentes a localização do usuário (3), dois botões que permitem entrar e sair do ambiente virtual (4), uma tabela que exibe informações de outros usuários (5) e a região que exibe as imagens do ambiente virtual (6). Cada um desses itens será visto em maiores detalhes nas seções seguintes. A fig. 5.21 apresenta a interface do protótipo.

FIGURA 5.21 - INTERFACE DO PROTÓTIPO



5.3.1 ESCOLHENDO O PERSONAGEM

Antes de o usuário entrar no ambiente virtual, ele tem a opção de escolher qual forma geométrica representará o seu personagem no ambiente virtual. Conforme ilustrado na fig. 5.21 e indicado pela seta 1, há quatro opções disponíveis: cone, caixa, cilindro e esfera.

Na verdade, o tipo de personagem escolhido pelo usuário não mudará em nada para o usuário em si. A única diferença será que esse usuário, para os demais usuários, será visto de acordo com a forma geométrica escolhida. Isso porque a visão que o usuário tem do ambiente virtual será em primeira pessoa, conseqüentemente ele não vê a si mesmo.

Como o objetivo desse protótipo era atentar-se aos aspectos de distribuição e ser multiusuário, optou-se por escolher essas formas para simplificar a implementação do mesmo e também otimizar o desempenho do ambiente.

5.3.2 IDENTIFICAÇÃO DA APLICAÇÃO

A seta 2 da fig. 5.21 mostra um componente cuja função é exibir um número que identifica essa aplicação no ambiente virtual. Implementado no protótipo existe uma pequena rotina que gera esse número baseado no nome do computador do usuário.

Esse número, o número do *site* (no caso desse protótipo esse numero é fixo) e o número que identifica a entidade no ambiente virtual local são reunidos em um objeto da classe `EntityID` e atribuído a cada entidade criada, formando um identificador único em todo o ambiente virtual.

5.3.3 LOCALIZAÇÃO NO AMBIENTE VIRTUAL

Os componentes indicados pela seta 3 da fig. 5.21 têm por finalidade mostrar as coordenadas de localização nos eixos **X**, **Y** e **Z** do usuário no ambiente virtual. O primeiro exibe as coordenadas no eixo **X**, o segundo as coordenadas no eixo **Y** e terceiro as coordenadas no eixo **Z**.

Esses mostradores começam a ser atualizados assim que o usuário entra no ambiente virtual e inicia a navegação no mesmo através do uso do teclado. A unidade padrão utilizada para as coordenadas é o metro.

5.3.4 ENTRAR E SAIR DO AMBIENTE VIRTUAL

A seta 3 da fig. 5.21 indica os botões utilizados para entrar e sair, respectivamente, do ambiente virtual. A partir do momento em que o usuário clica no botão “Entrar”, uma notificação da entrada desse usuário no ambiente virtual é enviada para os demais usuários e esse usuário começa a ser representado nas demais aplicações dos outros usuários.

Também, a partir desse momento esse usuário começa a receber as informações dos outros usuários e as tratar da forma apropriada, seja criando a representação de outros usuários nessa aplicação, seja modificando as informações das entidades já criadas.

Já o botão “Sair” faz com que uma notificação de que esse usuário está saindo do ambiente virtual seja enviada para os demais usuários e esses por sua vez se encarreguem de eliminar a representação desse usuário de suas aplicações. Esse botão também faz com que essa aplicação seja finalizada.

5.3.5 TABELA DE USUÁRIOS

A tabela indicada pela seta 5 da fig. 5.21 tem por finalidade exibir informações referentes a outros usuários que também estão no ambiente virtual.

Quando um novo usuário entra no ambiente virtual ou a sua presença é detectada, uma nova linha é acrescentada à tabela contendo as informações do mesmo. De maneira análoga, a linha de um usuário que sair do ambiente virtual também é removida da tabela.

Essa tabela é composta por 9 colunas cuja finalidade é exibir as seguintes informações que aparecem na mesma ordem da tabela: número que identifica a aplicação do usuário remoto, coordenada atual no eixo **X**, **Y** e **Z** e quantidade de `EntityStatePdu’s`, `FirePdu’s`, `RemoveEntityPdu’s`, `CreateEntityPdu’s` e `CollisionPdu’s` enviados por esse usuário remoto.

Resumindo, essa tabela permite visualizar a localização de outros usuários que estão no ambiente virtual e informações estatísticas a respeito da quantidade de Pdu’s enviados pelos mesmos.

5.3.6 NAVEGAÇÃO E VISUALIZAÇÃO DO AMBIENTE VIRTUAL

O componente indicado pelo número 6 na fig. 5.21 é o responsável por exibir as imagens do ambiente virtual.

Nesse protótipo se optou por exibir as imagens do ambiente virtual no modo aramado, objetivando melhorar o desempenho de renderização das imagens e conseqüentemente tornar as animações mais rápidas.

Quando o foco da aplicação está sobre esse componente, é possível utilizar algumas teclas para navegar no ambiente virtual e até mesmo disparar tiros. A navegação no ambiente

virtual está limitada a movimentos para frente e para trás, correspondentes a alteração nas coordenadas do eixo **Z**, para esquerda e para direita, correspondentes a alteração nas coordenadas do eixo **X** e para cima e para baixo, correspondentes a alteração nas coordenadas no eixo **Y**.

Os comandos de navegação aceitos no protótipo podem ser vistos na tabela 5.1.

TABELA 5.1 - COMANDOS PARA A NAVEGAÇÃO NO AMBIENTE VIRTUAL

Tecla	Efeito	Observação
Seta para cima	Desloca entidade para o fundo do ambiente	Subtrai 0,05m da coordenada no eixo Z
Seta para baixo	Desloca entidade para fora do ambiente	Adiciona 0,05m na coordenada do eixo Z
Seta para esquerda	Desloca entidade para a esquerda	Subtrai 0,05m da coordenada no eixo X
Seta para direita	Desloca entidade para a direita	Adiciona 0,05m da coordenada no eixo X
Page down	Desloca entidade para baixo	Subtrai 0,05m da coordenada no eixo Y
Page up	Descola entidade para cima	Adiciona 0,05m na coordenada do eixo Y
Barra de espaços	Dispara tiro	Cria entidade tipo tiro

Como há um controle de colisão durante a navegação, implementado no protótipo, quando se está colidindo com outra entidade, não é possível continuar avançando na direção em que ocorreu a colisão. Ou seja, caso o usuário esteja se movimentando para frente e entre em estado de colisão com alguma entidade, não é possível continuar se movimentando para frente, somente quando sair do estado de colisão.

5.4 ANÁLISE DOS RESULTADOS

Os seguintes itens podem ser destacados como resultados alcançados e também limitações encontradas:

a) o protótipo resultante apresentou a funcionalidade desejada, permitindo ter um grau de sincronia aceitável entre os usuários;
b) mesmo em modo aramado e com um cenário gráfico muito simples, a renderização do AV é pesada em computadores AMD K6 475 Mhz e 128Mb de memória (os quais foram utilizados para os testes), fazendo com que o processo de navegação no AV não seja muito suave. Esse item, em particular, merece maiores estudos para avaliar melhor as causas dessa não suavidade;

c) a utilização de computadores de capacidade de processamento diferentes gera animações sem suavidade nos demais usuários. A capacidade maior de processamento leva a geração de PDU´s numa velocidade maior daquela possível de ser processada por máquinas com capacidade de processamento mais modesto;
d) por se utilizar de UDP *broadcast*, o funcionamento desse protótipo está limitado a redes locais, já que esse protocolo não é roteável;
e) o protocolo UDP é utilizado em todas as situações, o que pode levar a alguns problemas sérios de sincronia no AV caso algum pacote importante, como por exemplo um pacote contendo um `CreateEntityPdu` ou `RemoveEntityPdu` (que informam acerca da criação ou remoção de uma entidade, respectivamente) seja perdido;

Nesse capítulo foi apresentado a especificação e os detalhes de implementação do protótipo de ambiente virtual distribuído. Além da apresentação do grafo de cena que especifica a construção gráfica do ambiente virtual, também foram apresentados os principais processos que controlam o ambiente, os quais foram implementados procurando levar em consideração as características e detalhes inerentes aos ambientes virtuais distribuídos apresentados no capítulo dois. A especificação e implementação desses processos também ajudaram a mostrar como as API's do Java3D e DIS-Java-VRML foram utilizadas para a implementação dos mesmos.

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 resultantes desse trabalho.

6 CONCLUSÕES

No decorrer desse trabalho, pôde-se ver alguns aspectos importantes que devem ser observados ao se construir um ambiente virtual distribuído. A maior parte desses aspectos diz respeito às características da rede que irá interconectar os usuários desse ambiente virtual. Justamente por isso, foram apresentadas algumas técnicas, resultantes das pesquisas feitas nessa área, visando utilizar da maneira mais otimizada possível os recursos da rede e também contornar as limitações que a mesma impõe.

Como o objetivo do trabalho era construir um protótipo de ambiente virtual distribuído, além de terem sido apresentados os aspectos inerentes ao mesmo, também se mostrou necessário apresentar duas API's responsáveis por darem suporte à construção do ambiente virtual em si e dos mecanismos de controle da comunicação entre os usuários do ambiente virtual.

A API que deu suporte à construção do ambiente virtual, o que inclui toda a estrutura tridimensional do ambiente virtual e controle do mesmo, foi a API do Java3D. Por sua vez, a API do DIS-Java-VRML foi a responsável por dar suporte à construção dos mecanismos de controle da comunicação entre os usuários.

Tendo-se conhecimento dos principais aspectos relativos aos ambientes virtuais distribuídos, bem como às técnicas utilizadas para a construção do mesmo, em conjunto com a utilização das API's do Java3D e DIS-Java-VRML, foi possível construir o protótipo e alcançar os objetivos desse trabalho.

A utilização da API do Java3D para a construção do ambiente virtual se mostrou como uma ótima escolha. Isso é justificado pela simplicidade e rapidez com que a utilização da mesma permite construir ambientes virtuais. Essa API possui um conjunto de classes que permite construir formas geométricas e ambientes virtuais pré-definidos que simplificam bastante o processo de construção do ambiente. Ao mesmo tempo, também há classes que permitem construir ambientes virtuais mais complexos e personalizados.

Outros pontos positivos para o Java3D são a boa documentação existente da API, seguindo os padrões Javadoc, que ajudam bastante no processo de entendimento de como e quando utilizar as diversas classes e seus respectivos métodos, bem como o tutorial que ajuda a complementar, de forma prática, a documentação da API.

No caso da API do DIS-Java-VRML, apesar de ter sido desenvolvida para trabalhar em conjunto com ambientes virtuais construídos na linguagem VRML, a mesma foi bastante útil pois já possuía implementado, em forma de classes, todos os PDU's necessários no processo de comunicação do ambiente, sendo que essas mesmas classes puderam ser aproveitadas sem a necessidade de fazer qualquer alteração das mesmas. Essa API também foi muito útil ao fornecer algumas classes que permitiam controlar o envio e recebimento de PDU's, abstraindo todo o processo de controle, como por exemplo de serialização e deserialização de objetos, para envio através da rede.

Já com relação à documentação da API, também nos padrões do Javadoc, a mesma se encontra incompleta, deixando muito a desejar. Porém, um ponto que compensa isso é uma série de exemplos disponíveis que ajudam a dar uma boa idéia de como são utilizadas as classes dessa API. Outro ponto interessante dessa API é que ela possui o código fonte aberto, o que sem dúvida, contribui para um melhor entendimento do funcionamento interno das classes e, conseqüentemente, a melhor utilização das mesmas.

Com os objetivos alcançados, o protótipo resultante desse trabalho serve como uma contribuição significativa no sentido de demonstrar que também a API do Java3D pode ser utilizada em conjunto com outras API's, como é o caso do DIS-Java-VRML, para a construção de ambientes virtuais distribuídos.

Inclusive, uma das tarefas ainda em aberto no grupo de pesquisa do DIS-Java-VRML, conforme Web3D (2001a), é a adaptação dessa API para que a mesma possa também funcionar com o Java3D, algo em que esse trabalho pode contribuir bastante.

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 classes mais avançadas do Java3D;

b) permitir que os usuários possam de alguma forma interagir com outros elementos do ambiente virtual;

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 melhores de tratamento de colisão e algoritmos de *Dead Reckoning*;

f) criar uma API genérica, a qual conteria classes que permitiriam controlar a interface entre, por exemplo, a API do DIS-Java-VRML e os elementos gráficos do ambiente virtual criados através do Java3D.

ANEXO A: 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 a API do Java3D. Essa API também pode ser obtida no endereço indicado acima, basta em "*Product Shortcuts*" selecionar "Java 3D". Um item importante a ser observado na instalação do Java3D é quando, durante a instalação, é perguntado onde se quer instalar os arquivos do tipo **`jar`** que possuem as classes da API. Nesse ponto é importante indicar o diretório onde foi instalado o JSDK 1.3 ou o JRE 1.3. Dessa forma aos arquivos do tipo **`jar`** correspondentes a API serão colocados corretamente no subdiretório **`$JAVA_HOME\jre\lib\ext`**. Sendo que **`$JAVA_HOME`** corresponde ao diretório principal onde foi instalado o JSDK 1.3 ou o JRE 1.3;

c) ter instalado os arquivos do tipo **`jar`** correspondentes à API do DIS-Java-VRML no subdiretório **`$JAVA_HOME\jre\lib\ext`**. Acesse o endereço http://www.web3d.org/WorkingGroups/vrtp/dis-java-vrml/download.html e baixe o arquivo **`dis-java-vrml.tar.gz`** ou **`dis-java-vrml.zip`**, descompacte-o num novo diretório qualquer e mova os arquivos **`dis-java-vrml.jar`** e **`dis-java-vrml-Signed.jar`**, que estarão nesse diretório onde foi descompactado, para o subdiretório **`$JAVA_HOME\jre\lib\ext`**;

d) depois disso, basta acessar o diretório onde estão as classes correspondentes ao protótipo implementado nesse trabalho e digitar **`javac PrototipoAmbienteVirtual.java`**, caso se queira recompilar o programa, ou **`java PrototipoAmbienteVirtual`**, caso se queira executar diretamente o protótipo. É importante que o caminho **`$JAVA_HOME\bin`** esteja incluído na variável **`PATH`** do sistema operacional em uso.

REFERÊNCIAS BIBLIOGRÁFICAS

BRUTZMAN, Donald P.; MACEDONIA, Michael R.; ZYDA, Michel J. **Internetwork infrastructure requirements for virtual environments,** Monterey: [s.n.], 1995. Disponível em: <ftp://taurus.cs.nps.navy.mil/pub/auv/brutzman/nii_2000.txt>. Acesso em: 12 nov. 2000.

Comer, Douglas. **Internetworking with TCP/IP**. 3. ed. Englewood Cliffs, NJ: Prentice Hall, 1995.

FERREIRA, A. B. de Holanda. Dicionário da língua portuguesa. Rio de Janeiro: Nova Fronteira, 1980.

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: 22 jan. 2001.

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.

LOCKE, John**. An introduction to the internet networking environment and SIMNET/DIS.** Monterey: [s.n.], 1994. Disponível em: <http://web.nps.navy.mil/~code09/techreports.html>. Acesso em: 13 fev. 2001.

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.

NADEAU, David R.; SOWIZRAL, Henry A. Introduction do programming with Java 3D. In: Internation Conference in Computer Graphics and Interactive Techniques, 26., 1999. Los Angeles. **SIGGRAPH'99 Courses...** Los Angeles: [s.n.], 1999. Disponível em: <http://www.sdsc.edu/~nadeau/Courses/Siggraph99/>. Acesso em: 6 mar. 2001.

NPSNET, Naval Postgraduate School. **NPSNET Research Group.** Monterey: [s.n.], [2000?]. Disponível em: <http://www.npsnet.org/NPSNET-V/>. Acesso em: 13 nov. 2000.

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: 08 nov. 2000.

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.

SOWIZRAL, H. *et al.* **The Java 3D API specification.** 2. ed. Reading: Addison-Wesley, 2000.

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.

SUN, Sun Microsystems. **Java3D API.** [S.l.: s.n.], [2000a?]. Disponível em: <http://java.sun.com/products/java-media/3D/index.html>. Acesso em: 26 set. 2000.

_____. **The Java 3D API: Technical white paper.** [S.l.: s.n.], [2000b?]. Disponível em: <http://www.javasoft.com/products/java-media/3D/collateral/j3d_api/j3d_api_1.html>. Acesso em: 30 out. 2000.

_____. **Getting start with the java3d API.** [S.l.: s.n.], [2000c?]. Disponível em: <http://java.sun.com/products/java-media/3D/collateral/j3d_tut.zip>. Acesso em: 26 set. 2000.

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: 20 fev. 2001.

WEB3D, Web3D Consortium. **Distributed Interactive Simulation DIS-Java-VRML Working Group.** [S.l.: s.n.], [2001a?]. Disponível em: <http://www.web3d.org/WorkingGroups/vrtp/dis-java-vrml/>. Acesso em: 03 mar. 2001.

_____. **DIS-Java-VRML Javadoc.** [S.l.: s.n.], [2001b?]. Disponível em: <http://www.web3d.org/WorkingGroups/vrtp/javadoc/dis-java-vrml/index.html>. Acesso em: 03 mar. 2001.

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

ZYDA, Michael J. *et al*. Exploiting reality with multicast groups. **IEEE computer graphics And applications,** Los Alamitos, n. 5, p. 38-45, set./out. 1995.