Capítulo 1

Desenvolvimento Rápido de Aplicações de Realidade Virtual Utilizando *Software* Livre

Liliane S. Machado, Daniel F. L. Souza, Leandro C. Souza e Ronei M. Moraes

Abstract

The development of virtual reality (VR) systems can join several functionalities. Thus, the knowledge of specific techniques, as well as the learning of programming libraries of devices, and the integration of tasks demand time and can become complex the development process. The goal of this course is to present a set of libraries called CyberMed, set up to decrease the time of development of applications based on VR. It will be presented the concept used to develop the libraries, their functionalities and ways to expand them. Some functionalities of CyberMed can be highlighted: stereoscopic visualization, interaction through conventional and haptic devices, support to deformable models, integration of methods for user assessment and support to collision detection.

Resumo

O desenvolvimento de sistemas de realidade virtual (RV) pode englobar diferentes funcionalidades. Neste sentido, o aprendizado de técnicas específicas, bem como de bibliotecas de dispositivos, e a integração de todas as tarefas demandam tempo e podem tornar complexo o processo de desenvolvimento. O objetivo deste curso é apresentar um conjunto de bibliotecas livres, chamado CyberMed, que permite o rápido desenvolvimento de aplicações baseadas em RV. Pretende-se apresentar o conceito de desenvolvimento das bibliotecas, suas funcionalidades e modos de expandi-las. Dentre as funcionalidades do CyberMed destacam-se: visualização estéreo, interação através de dispositivos convencionais e hápticos, uso de modelos deformáveis, suporte à detecção de colisão e integração de métodos de avaliação online do usuário.

1.1. Introdução

Sistemas de Realidade Virtual (RV) utilizam dispositivos e plataformas computacionais que permitem criar e oferecer ambientes virtuais de simulação e visualização interativa. Esses sistemas podem ser classificados entre imersivos e não imersivos de acordo com o grau de envolvimento do(s) usuário(s) com a aplicação. Tal envolvimento é obtido a partir da exploração dos cinco sentidos do usuário durante a execução do ambiente de RV. Neste contexto, a visão é responsável por facilitar e ampliar o processo de assimilação conceitual. Aliada à interatividade, a visualização permite identificar, compreender e explorar ambientes tridimensionais utilizando ou não recursos de estereoscopia. Quanto à interação, há diferentes formas que podem ser oferecidas por dispositivos convencionais (mouse e teclado), de rastreamento e hápticos. O uso de tais dispositivos pode depender de conjuntos específicos de rotinas, fornecidos por seus fabricantes, integrados ao sistema de RV.

A visualização tridimensional é a característica mais comumente associada a sistemas de RV. Com o desenvolvimento de diferentes equipamentos e técnicas, é possível obter tal visualização utilizando tanto dispositivos populares como dispositivos profissionais, como óculos obturadores. Assim, seria adequado que tanto os diversos métodos de geração das imagens quanto os dispositivos pudessem ser suportados pelas APIs ou pacotes de desenvolvimento de RV, atendendo o diversificado público de desenvolvedores e usuários finais.

Sob o ponto de vista da plataforma física, sua configuração pode permitir o uso de modelos mais complexos, técnicas com alto custo computacional e suporte a dispositivos específicos. Tal fato pode demandar o uso de máquinas dedicadas. Por outro lado, a ausência de plataformas dedicadas não deve ser um impeditivo para o desenvolvimento e uso de sistemas de RV, visto que a tecnologia atual de acesso popular oferece meios de realizar operações consideradas complexas até pouco tempo atrás.

A principal dificuldade encontrada no desenvolvimento de aplicações de RV é certamente a integração de dispositivos e técnicas. Especificamente, a integração de dispositivos pode ser complexa quando são utilizados equipamentos não convencionais. Neste caso é necessário conhecer o padrão de funcionamento e programação das APIs ou bibliotecas fornecidas pelo fabricante para que estes possam ser integrados ao código geral da aplicação. Outros problemas relacionados à integração referem-se ao uso de técnicas de visualização, tratamento gráfico, identificação de colisões no ambiente virtual, deformação de objetos em tempo-real e monitoramento das ações do usuário. Tal conjunto de possibilidades torna complexo o processo de desenvolvimento além de demandar um módulo de gerenciamento de tarefas concorrentes.

Uma das qualidades relacionadas aos sistemas computacionais e, conseqüentemente, aos sistemas de RV, é a possibilidade de monitorar as interações do usuário com o sistema. Sob o ponto de vista da RV, tal monitoramento torna-se fundamental, pois pelo fato do ambiente visar o realismo, as interações correspondem às reações do usuário frente ao que ele compreende do ambiente virtual. Isto permite o uso das informações de interação [Sharabi 2007] para fins diversos que englobam desde a quantificação da compreensão, à usabilidade e à avaliação do usuário frente a uma tarefa no ambiente virtual [Machado 2000]. Apesar da sua importância, o suporte ao

monitoramento das ações do usuário não é uma funcionalidade comumente encontrada nos pacotes de desenvolvimento de sistemas de RV. Neste sentido, as avaliações são realizadas através de métodos conhecidos como *offline*, ou seja, as informações das ações do usuário são gravadas em vídeo ou armazenadas em um arquivo de dados e, posteriormente, são analisadas [Moraes 2004].

Para o desenvolvimento de aplicações de RV deve-se considerar as diversas finalidades e públicos envolvidos. Para o desenvolvimento de sistemas que envolvem apenas visualização interativa há uma série de ferramentas de desenvolvimento. Entretanto, quando o sistema precisa envolver deformação interativa e integrar algum dispositivo especial, a tarefa de desenvolvimento começa a tornar-se mais complexa. Sendo assim, observa-se uma série de dificuldades no desenvolvimento de sistemas de RV. Por exemplo, apesar dos diversos pacotes próprios para o desenvolvimento de aplicações de RV, poucos são aqueles que abordam a gama de dispositivos disponíveis no mercado. Problema similar ocorre com a integração de técnicas mais recentes, métodos de suporte a dispositivos de baixo custo, de colaboração, etc. De fato, não é possível prever todas as necessidades dos desenvolvedores em relação a dispositivos e métodos. Entretanto, é possível oferecer suporte a alguns e disponibilizar meios de expandir e complementar as ferramentas de desenvolvimento. Neste contexto, a sincronização das tarefas do sistema deve ser garantida pela ferramenta de desenvolvimento, acelerando este processo e livrando o programador de detalhar cada técnica empregada, ao mesmo tempo que oferece liberdade para incorporação ou especialização de processos.

Em geral, pacotes de desenvolvimento de sistemas de RV que suportam o us de técnicas variadas são proprietários e são compatíveis com conjuntos específicos de dispositivos. Dentre os pacotes gratuitos e de código aberto destacam-se o Chai3D (www.chai3d.org) e o CyberMed [Souza 2007a], cujo objetivo é oferecer suporte sincronizado ao uso de técnicas tradicionais utilizadas em RV, bem como permitir que expansões sejam incorporadas ao código da aplicação. Com isso, tais pacotes facilitam o desenvolvimento das aplicações sem que o programador fique restrito às suas funcionalidades. Dentre estes pacotes, o CyberMed destaca-se por ser baseado em um sistema operacional livre e oferecer suporte ao monitoramento das ações do usuário com métodos de avaliação *online*. Deste modo, o CyberMed está apto a permitir o monitoramento das ações do usuário e emitir um relatório sobre seu desempenho durante a interação com o sistema de RV. Tal característica é particularmente importante no desenvolvimento de simuladores voltados para o treinamento de atividades, sejam na área de educação, medicina ou engenharia, dentre outras.

Este tutorial descreve como agilizar o processo de desenvolvimento de sistemas de RV utilizando o CyberMed. Para isso serão apresentados alguns conceitos deste pacote, bem como sua arquitetura e funcionalidades que permitem sua utilização e expansão.

1.2. CyberMed

O CyberMed é um conjunto de bibliotecas voltadas ao desenvolvimento de sistemas de RV baseados em PCs [Souza 2007a]. Seu objetivo é dar suporte à criação de aplicações que envolvam os conceitos de RV, particularmente aquelas destinadas à simulação.

Inicialmente concebido para facilitar o desenvolvimento de simuladores para a área médica, seu uso mostrou-se amplo para conceber aplicações em diversas outras áreas. Além disso, como possui código aberto e é livre, permite a incorporação de métodos específicos e de suporte a dispositivos ainda não contemplados. Por ter sido concebido para sistemas baseados em computadores pessoais, o CyberMed oferece suporte a dispositivos de baixo custo e a dispositivos específicos, utilizados em aplicações geralmente profissionais. Esta qualidade, faz com que ele satisfaça uma gama considerável de programadores, permitindo-lhes selecionar os recursos disponíveis e adequados para o público usuário da aplicação final.

Toda a concepção do CyberMed baseou-se no conceito de *software* livre e aberto e buscou-se a não dependência de pacotes ou sistemas proprietários. Por esta razão, o CyberMed baseia-se no sistema operacional Linux e utiliza apenas linguagens abertas em seu código. Embora tenha sido desenvolvido de modo a permitir uma programação de alto nível, o modo de concepção do CyberMed permite expandir suas funcionalidades através de uma programação de mais baixo nível. Quanto à plataforma de desenvolvimento e execução dos sistemas e aplicações desenvolvidos com o CyberMed, esta dependerá das funcionalidades utilizadas e do grau de complexidade das operações escolhidas pelo programador. Assim, o uso de modelos tridimensionais complexos exigirá máquinas com maior capacidade de memória e processamento. De modo similar, uma aplicação composta pela visualização de modelos de menor complexidade utilizando um método de estereoscopia por anaglifo poderá ser executada em computadores populares.

O CyberMed está disponível para *download* através das páginas do projeto no sítio na Internet http://www.de.ufpb.br/~labteve/projetos/cybermed_p.html.

1.2.2. Arquitetura

A abordagem de modelagem escolhida no CyberMed foi orientada a objetos e foram definidos padrões a serem seguidos durante todo o desenvolvimento do projeto. A tríade Modelo/Visão/Controlador objetivou separar as camadas de modelagem, visualização de forma que estas pudessem ser trabalhadas separadamente, garantindo características de reuso ao *software* desenvolvido e uma maior agilidade no desenvolvimento. Como padrões de projeto foram utilizados a abordagem o *singleton* e o *Abstract Factory* [Gamma 2000]. O *singleton* tem como principal característica oferecer uma única instância de uma determinada classe, de forma que esta possa ser usada em todo o seu escopo, garantindo a unicidade da informção. Por outro lado, o *Abstract Factory* sugere a criação de interfaces para a criação de objetos com operações em comum, permitindo a fácil incorporação de novos objetos ao sistema que mudem apenas alguns métodos entre si.

Basicamente, o CyberMed divide-se em um conjunto de camadas que têm por finalidade prover uma série de serviços para os usuários das bibliotecas. Cada camada procura abstrair uma série de conceitos com o objetivo de facilitar a construção de aplicações baseadas em RV. A Figura 1 mostra o conjunto de camadas que compõe o CyberMed e como estas estão dispostas dentro do sistema. A Figura 2 apresenta a estrutura geral das dependências no sistema CyberMed, levando em conta suas

principais classes. Se relacionada à Figura 1, a classe CybParameters faz parte do *Core* e as demais classes da *Application Engine*.

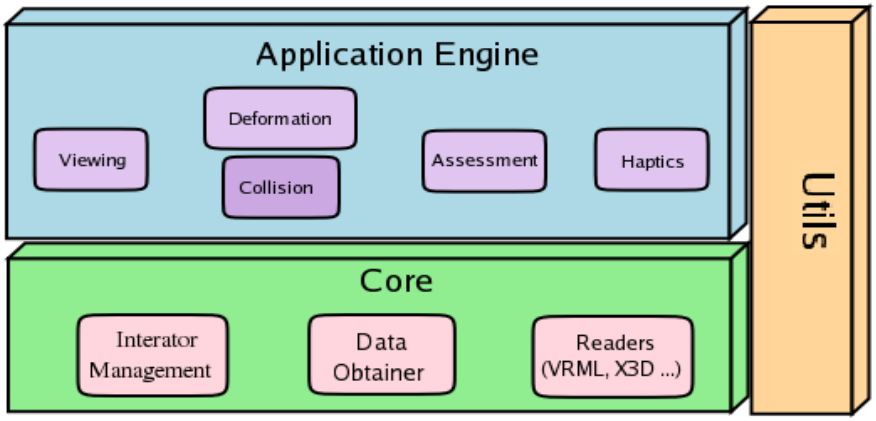

Figura 1: Arquitetura Geral do CyberMed.

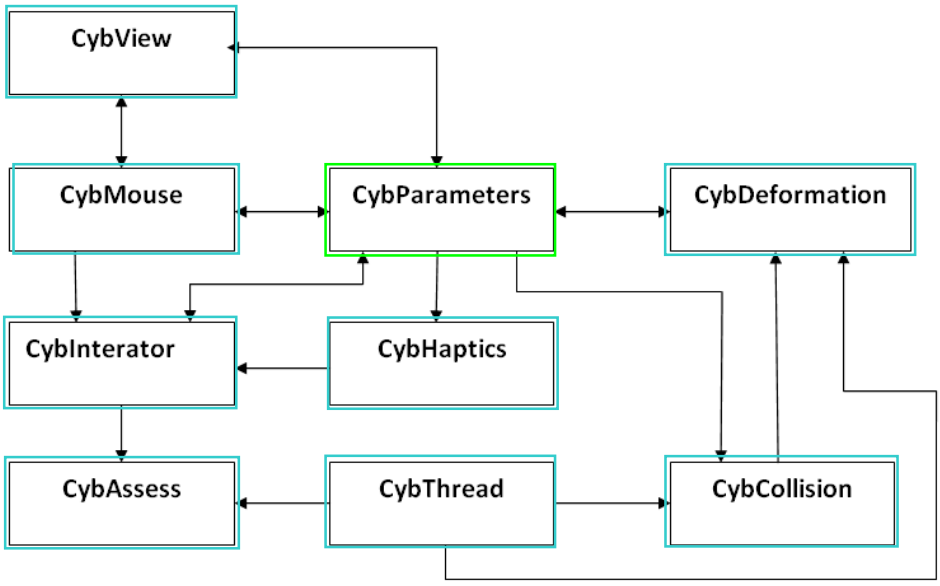

Figura 2: Diagrama geral de dependências do CyberMed. A cor verde indica a classe contida no *Core* do sistema e a cor azul as classes contidas na *Application Engine*.

A camada *Core* é responsável pelo controle dos estados internos do sistema. Algumas de suas funcionalidades estão ligadas a aquisição, cálculo, armazenamento e acesso aos dados do sistema. A aquisição de informações de modelos gráficos é feita através dos importadores de modelos (aqui chamados de *Readers*), que podem ser estendidos com a finalidade de integrar padrões diversos de modelos ao CyberMed. Outra responsabilidade desta camada é o gerenciamento dos interadores que, porventura, sejam integrados ao CyberMed.

A camada *Application Engine* tem por objetivo prover um conjunto de métodos para auxiliar o usuário na construção das aplicações. Esta camada oferece uma série de pacotes para a inclusão de visualização, colisão, deformação, avaliação e interação háptica. Por fim, a camada *Utils* possui uma série de mecanismos que auxiliam o desenvolvedor em tarefas como cálculo de matrizes, operações de transformação lineares, etc. Entre outras funcionalidades, a camada *Utils* também possui um conjunto de métodos para a construção de menus.

1.3. Estrutura de Dados

A classe de Estrutura de Dados é a provedora dos dados utilizados no CyberMed e está localizada na camada *Core*. Esta classe armazena informações originais de modelos tridimensionais, bem como cálculos diversos referentes a esses modelos, como normais e parâmetros de transformação espacial. Estas informações são centralizadas nesta classe e utilizada por todas as outras classes do conjunto de bibliotecas, de modo que não ocorra duplicidade ou recálculos desnecessários por cada classe individualmente. Como importantes características desta classe estão a otimização e controle no acesso às informações. A classe que agrupa e é o centro das informações de todo o sistema é a CybParameters. Ela é responsável por armazenar a estrutura dos objetos da cena e informações sobre estes (normais e transformações, por exemplo), centralizando informações sobre deformações e conectando a detecção de colisão à interação.

A leitura de objetos é feita a partir da classe CybDataObtainer, que recebe como parâmetros no construtor o número de objetos que serão carregados e o identificador do interador que será utilizado na simulação. O método utilizado para carregar a estrutura dos objetos é o método *load(int id, char* file)*, onde *id* representa o identificador do objeto e *file* o nome do arquivo VRML 2.0 que descreve o objeto. A Figura 3 mostra um exemplo de código de como isto é feito. Nesse caso, na linha 17 é solicitado o cálculo de parâmetros adicionais que são automaticamente armazenados na estrutura de dados dos objetos.

```
 1  int main(int argc, char** argv)
 2  {
 3  char *motor = "motor.wrl";     // object file
 4  char *alicate = "alicate.wrl"; // object file
 5  int numLayer = 2;              // number of objects
 6  int numInterator = 1;          // number of interators
 7
 8  //creates the data structure
 9  CybDataObtainer<cybTraits> data(numLayer, numInterator);
10
11  //loads model into the structure
12  data.loadModel(0, motor);
13  data.loadModel(1, alicate);
14
15  //calculates auxiliary parameters of the structure as
16  //neighbor points, normals, etc.
17  data.startParameters(numLayer);
18
19  return 0;
20  }
```

Figura 3: Exemplo de código para montar a estrutura capaz de armazenar um objeto adquirido de um arquivo VRML 2.0.

1.4. Visualização

As classes de visualização são responsáveis pela cena gráfica e seus objetos, bem como controlar todas as ações que podem ocorrer sobre eles [Cunha 2006]. Estas classes contêm a definição da cena gráfica e dos parâmetros ligados ao seu controle, bem como oferecem diferentes métodos de visualização que incluem: a visualização monoscópica (planar), a visualização estereoscópica por anaglifo e a visualização estereoscópica por obturação da luz. Elas foram desenvolvidas através de chamadas às funções OpenGL de forma que seu design procurou abstrair as dificuldades oferecidas pela API gráfica, englobando suas funcionalidades na perspectiva de orientação a objetos. Através desta abstração surgiu um conjunto de classes que atualmente compõe o módulo de visualização do CyberMed. Assim, uma aplicação que tenha por objetivo visualizar algum tipo de entidade pode ser construída de forma simplificada. A seqüência a seguir mostra os passos necessários para conceber tal aplicação:

1. Indica-se para a API através do *DataObtainer* o número de objetos que se deseja visualizar e o número de interadores que se deseja criar;
2. Ainda através do *DataObtainer*, carregam-se os modelos referentes aos objetos visualizados e referentes às representações gráficas dos interadores;
3. Inicializa-se os parâmetros de cada camada;
4. Chama-se o método *init* para iniciar a apresentação gráfica das imagens;

A Figura 4 apresenta o código da seqüência de passos descrita acima utilizando o CyberMed. O passo 1 é mostrado na linha 6. O passo 2 é feito na linha 15. O passo 3 é executado na linha 18. Por fim, o passo 4 é feito na linha 21. Seguindo estes passos básicos pode-se criar visualizadores diversos em poucos minutos. Como padrão, o CyberMed já habilita duas formas de interação: através do teclado e através do mouse. Desta forma, o código mostrado na Figura 4 cria um visualizador para objetos que já podem ser explorados através de comandos de mouse e teclado.

```
 1 char *fileName = "model.wrl"; //Model name
 2 int numLayer = 1; //Number of layers used in this application
 3 int numInterator = 0;
 4
 5 // Transfer the OF data to CybParameters structure
 6 CybDataObtainer<cybTraits> data(numLayer, numInterator);
 7
 8 // Monoscopic Visualization
 9 CybViewMono view;
10
11 //Access the parameters information of scene and graphical objects
12 CybParameters *cybCore = CybParameters::getInstance();
13
14 /*Load the model from VRML file (layerID, fileName)*/
15 data.loadModel(0, fileName);
16
17 /*Initialize the meshes parameters*/
18 data.startParameters(numLayer);
19
20 /*Initialize visualization*/
21 view.init();
```

Figura 4. Visualizador simplificado desenvolvido utilizando o CyberMed.

Outra característica importante do módulo de visualização é a capacidade de reutilizar o código para utilizar outros tipos diversos de visualização. O código

apresentado na Figura 4 gera uma cena monoscópica para o usuário. Caso o desenvolvedor estivesse interessado em oferecer algum outro tipo de método de visualização bastaria mudar uma única linha de código. Como exemplo, suponha que se queira utilizar visualização por anaglifo real. Neste caso, apenas a linha 9 da Figura 4 precisará ser modificada para:

```
CybViewAnaglyph view;
```

Informações referentes às características do objeto gráfico podem ser acessadas através da classe CybParameters, que faz parte da camada *Core* do CyberMed. Como citado na sessão que trata da arquitetura do CyberMed, o núcleo é responsável por armazenar o estado das entidades definidas pelo programador no contexto da aplicação. Outras informações de ambiente como iluminação e configuração de janelas, dentre outras, podem ser feitas através da própria classe de visualização. Como exemplo, a Figura 5 mostra que linhas de código deveriam ser inclusas antes da chamada do método *init()* para, por exemplo, modificar a cor do objeto e o nome da janela gráfica. Na linha 9 do exemplo modifica-se a cor do objeto gráfico com identificador 0. A linha 12 mostra como modificar o nome da janela de visualização.

```
1    ...
2    //Shutter Visualization
3    CybViewShutter view;
4
5    //Access the parameters information of scene and graphical objects
6    CybParameters *cybCore = CybParameters::getInstance();
7    ...
8    //Set the object color (layerID, r, g, b,a)
9    cybCore->setColor(0,1.0,1.0,0.7,0.5);
10
11   //Set the window name
12   view.setWindowName("Simple Visualization");
13   ...
14   /*Initialize visualization*/
15   view.init();
```

Figura 5: Exemplo de como manipular certas funcionalidades referentes à visualização.

Para a inserção de novos métodos de visualização é necessário herdar a classe CybView. Esta classe dispõe de métodos de controle e acesso à estrutura de dados e aos menus que podem ser associados ao sistema. O novo método de visualização deve estar contido dentro do método *void display()*. A Figura 6 apresenta os tipos de visualização disponíveis para utilização no sistema.

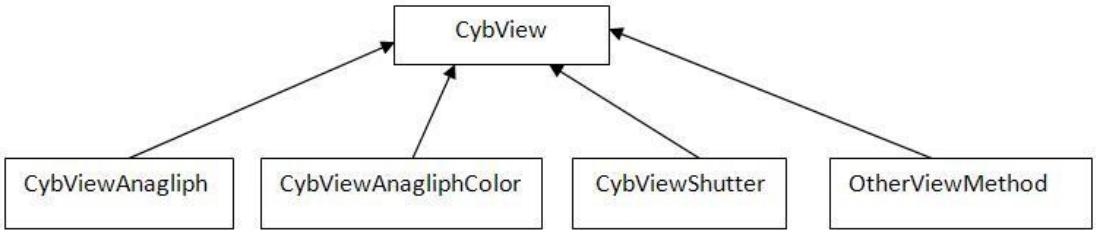

Figura 6. Estrutura de classes representando os tipos de visualização presentes no CyberMed.

1.5. Concorrência

A sincronização, decorrente da necessidade de concorrência entre as tarefas de uma aplicação de RV, desempenha papel fundamental na compatibilização e controle dos acessos às informações do CyberMed. Esta classe permite ordenar as tarefas presentes na aplicação de acordo com as funcionalidades utilizadas e estabelecer os critérios de concorrência. Isto porque as aplicações desenvolvidas com o CyberMed podem não utilizar determinadas funcionalidades, fato este que pode não dispensar a necessidade de sincronização entre as que estiverem presentes. A sincronização também é utilizada para estabelecer o ciclo de execução de cada tarefa, aumentando ou diminuindo o número de vezes por segundo que cada uma deve ser executada. Esta característica é essencial, pois permite adequar o ciclo das tarefas à aplicação, otimizando o seu desempenho.

A classe que implementa a funcionalidade de sincronização no sistema é a classe CybThread. Ela utiliza funções da biblioteca "pthread.h" para a manipulação das *threads* que conterão os sub-sistemas disponíveis no CyberMed. Entretanto, como a *pthread* é específica para o sistema operacional (SO) LINUX, as rotinas que utilizam tal biblioteca só podem ser compiladas neste SO.

Para se incorporar uma nova classe que possua propriedades de uma *thread* esta deve herdar da classe CybThread. A única exigência é que a nova classe implemente o método *void run()* (que é virtual puro e, caso não seja implementado, causará um erro de compilação). Para que a *thread* entre em execução, além da criação do objeto torna-se necessário chamar o método *init()*. Este método cria a *thread* e faz com que ela execute o método *run()* implementado na classe filha. Após criar e iniciar a execução da *thread*, torna-se necessária a chamada do método *join()* para que a *thread* não termine depois de sua primeira execução. Os principais métodos da classe CybThread são apresentados na Tabela 1.1.

A classe CybThread é uma classe independente do sistema. Existem sub-classes de CybThread especializadas em cada tarefa que o sistema propõe, tal como deformação e detecção de colisão, que facilitam esta incorporação de métodos específicos de sua área que necessitam ser processados em uma *thread* específica. Para implementar uma nova funcionalidade não prevista ainda no sistema, sugere-se utilizar a CybThread como superclasse e delegar responsabilidades relativas àquela aplicação específica ao longo da nova hierarquia.

Um exemplo de código para a utilização da classe CybThread pode ser dado pela Figura 7. Nesta figura é criada uma classe *Tarefa*, a partir de CybThread, que recebe um ponteiro como argumento e implementa o método *run()* (que é o único método que deve ser implementado, ou seja, o método que executará as ações da *thread*). Observa-se também na Figura 7 que o método *main()* compartilha a informação de uma única variável entre duas *threads* criadas a partir de objetos Tarefa (linhas 25 e 26). É interessante observar que no próprio método *main()* e no método *run()* foram utilizados métodos de controle de acesso para garantir a integridade da informação em ambas as *threads*. Tal controle é feito através dos métodos *lock()* e *unlock()*.

```
 1  #include “cybThread.h”
 2  #include <iostream>
 3  using namespace std;
 4  class Tarefa : public CybThread
 5  {
 6   int* info; // Pointer to a variable
 7   public:
 8   Exemplo(int* info)
 9    {
10     this->info = info;
11    }
12   void run()
13    {
14     cout << “hello”;
15     CybThread::lock(); //block other threads
16     //access information of info
17     cout << “this is a thread and its info ” << *(this->info) << endl;
18     CybThread::unlock(); // free other threads
19    }
20  };
21
22  int main()
23  {
24    int k = 0;
25    Tarefa* ex1 = new Tarefa(&k);
26    Tarefa* ex2 = new Tarefa(&k);
27
28    ex1->init(); // creates threads
29    ex2->init();
30    while(k < 10) { CybThread::lock();  k++; CybThread::unlock(); } //updates k
31    ex1->join();
32    ex2->join();
33
34    return 0;
35  };
```

Figura 7: Exemplo de criação de uma nova classe que se comporta como uma *thread* utilizando a classe CybThread.

As classes do sistema que utilizam a CybThread são a CybCollision, a CybAssess e a CybDeformation. Para cada objeto criado a partir destas classes haverá uma *thread* responsável por executar os seus métodos. As classes CybView e CybPhanton não a utilizam, pois apresentam um contexto de trabalho que cria suas próprias *threads*. A Figura 8 apresenta a hierarquia das classes dependentes da classe de concorrência.

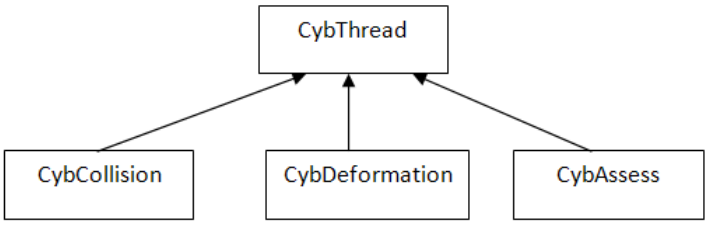

Figura 8. Hierarquia de dependências das classes de concorrência.

Tabela 1.1: Métodos da classe CybThread.

void destroy();	Método destrutor da *thread*.
bool isHabilitated();	Indica se o método *run()* está habilitado para ser executado ou não.
int getTime();	Retorna o tempo (em milissegundos) de espera para a rechamada do método *run()*.
void init();	Cria a estrutura da *thread* e habilita a execução do método *run()*.
void join();	Faz uma *thread* em questão esperar a execução das outras *threads*. Recomenda-se chamar este método depois da chamada do método *init()*.
static void lock();	Oferece exclusividade de execução a *thread* em questão e nenhuma outra *thread* será executada até que o método *unlock()* seja chamado. Recomenda-se utilizar este método antes de alteração de variáveis compartilhadas por duas ou mais *threads* a fim de se evitar uma inconsistência nos dados (vide *unlock()*).
void setTime(int time);	Retorna o tempo (ms) de espera para a rechamada do método *run()*.
static void sleep(int time);	Define um tempo de espera para a *thread*, em milissegundos.
void start();	Oferece permissão de execução do método *run()*.
void stop();	Retira a permissão de execução do método *run()*.
static void unlock();	Retira a exclusividade de execução de uma determinada *thread* (vide *lock()*).

1.6. Detecção de Colisão

A classe de detecção de colisão foi chamada CybCollision. Ela incorpora atributos inerentes a qualquer tipo de detecção de colisão que possa vir a ser incorporada ao Sistema CyberMed. A CybCollision associa ao método de colisão a camada com a qual ele está associado e verifica posições do interador. Além disso, ela herda uma classe de sincronização, mas não implementa o método *run()* que inicia o teste de colisão (o que deve ser feito pelos filhos que herdem dela). Ela é a classe base para os dois passos que incorporam uma detecção de colisão: a *Broad Phase* (localiza a região da colisão) e a *Narrow Phase* (a partir de uma região informa o ponto da colisão) [Luque 2005].

O CyberMed permite o uso de uma colisão do tipo *narrow* por objeto. Entretanto, este tipo de colisão pode ser utilizado em conjunto com um método tipo *broad*. Além disso, novos métodos podem ser adicionados associados à categoria de colisão ao qual se referem. Assim, as sub-classes da CybCollision foram chamadas de CybNarrowCollision e CybBroadCollision. A CybBroadCollision faz a ligação entre métodos da *broad phase* da colisão com uma *narrow phase* correspondente. Ela herda a CybCollision e possui uma lista que recebe triângulos como os elementos integrantes das áreas selecionadas, para uma detecção de colisão mais apurada. Não implementa nenhuma técnica de detecção de colisão, o que deve ser implementado em alguma classe

filha, mais especificamente no método *run()*, que preenche a lista dos triângulos para que o método correspondente à *narrow phase* possa ser executado. A CybNarrowCollision herda da classe CybCollision e associa características inerentes à *narrow phase*. Seus filhos devem implementar o método *run()* que executa a verificação de colisão. A CybNarrowCollision pode receber como parâmetros um identificado de objeto ou uma referência para CybBroadCollision que, neste caso, associa um objeto da *broad phase*. Ela também incorpora características bastante descritivas sobre a região do objeto em que houve colisão.

A implementação da classe geral de colisão, a CybCollision, preparou a base para a inserção e expansão de métodos de detecção de colisão no CyberMed. Inicialmente, foi adicionado um método na CybNarrowCollision para a verificação de colisão entre esferas e triângulos. Este método foi implementado na sub-classe CybSphereTriangle, que herda a classe CybNarrowCollision e seus atributos, podendo alterá-los. Esta sub-classe implementa o teste da colisão em seu método *run()*. Um exemplo de seu uso será feito em associação com a Deformação (vide tópico 1.7). A classe CybCollision se liga com a estrutura de dados a fim de monitorar a posição do dispositivo interador na cena. Ela também provê métodos para a conversão de coordenadas entre a cena e os objetos manipulados (que ficam estáticos na estrutura).

Uma característica importante da estrutura das classes de detecção de colisão é que qualquer CybNarrowColision pode se ligar a qualquer CybBroadCollision. A ligação entre elas ocorre por uma lista de triângulos, contida na CybBroadCollision, que será atualizada por seu método de detecção de colisão e averiguada pelo objeto filho de CybNarrowColision. Qualquer classe filha de CybBroadCollision deverá apenas implementar o método *run()* que conterá o método de detecção de colisão empregado, que deverá adicionar valores na lista listOfTriangles (do tipo ofList<cybMesh<cybTraits>::sCell *>), que representa um lista de triângulos repassados para algum filho de CybNarrowCollision. Mostra também que os filhos de CybNarrowCollision devem implementar além do método *void run()* (utilizado se não houver nenhum CybBroadCollision associado) o método *void collision (ofList<cybMesh<cybTraits>::sCell *>* list)* que será chamado caso algum CybBroadCollision seja associado. Neste caso, *list* será uma lista de triângulos previamente selecionada pelo método filho da CybBroadCollision. A Figura 9 é apresentado como desenvolver classes que estejam relacionadas entre si através da CybBroadCollision e da CybNarrowColision.

```
class CybSomeBroad : public  CybBroadCollision
{

public:

//constructor
CybSomeBroad(int layer) : CybBroadCollision(layer) {}

// thread method
void run()
{
//calling collision method and updating the structute
//ofList<cybMesh<cybTraits>::sCell *> listOfTriangles to store
//triangles list for narrow collision
}

};

class CybSphereTriangle : public CybNarrowCollision
```

```
{
private:
  float radius; //sphere radius

public:

  //constructor 1
  CybSphereTriangle(int layer) : CybNarrowCollision(layer)
  {
    …   //initialize attributes
  }

  //constructor 2
  CybSphereTriangle(CybBroadCollision* broadObject) :
  CybNarrowCollision(broadObject)
  {
   …   //initialize attributes
  }

  // method called if there is not broad associated
  void run();
  {
    // collision detection method applied over all object
  }

  void collision(ofList<cybMesh<cybTraits>::sCell *>* list)
  {
    // called if there is a broad. Access to triangles
    // can use the parameter list.
  }
};
```

Figura 9: Exemplo de criação de classes que implementem métodos de detecção de colisão utilizando as classes CybBroadCollision e CybNarrowCollision.

A Figura 10 mostra o diagrama de classes representando os tipos de colisão presentes no sistema CyberMed, como também alguns que podem ser incorporados, tais como AABB (*Axis-Aligned Bounding Boxes*), OBB (*Oriented Bounding Boxes*) [Bergen 2004].

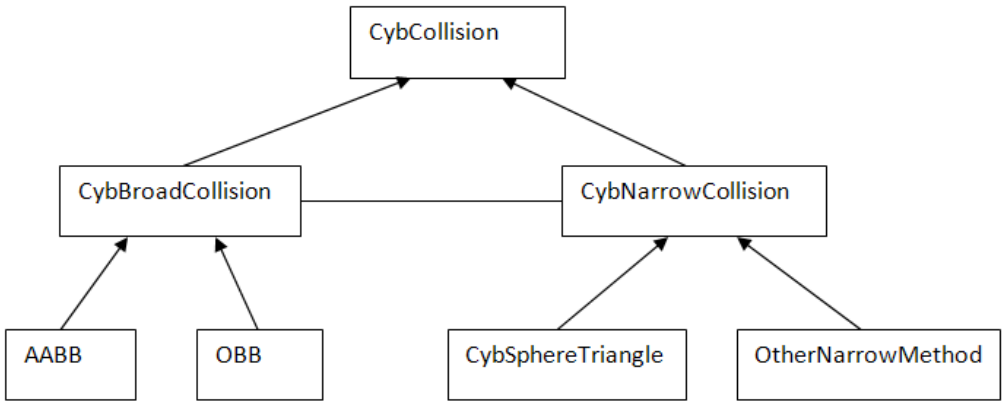

Figura 10: Diagrama de classes de detecção de colisão.

1.7. Deformação

A classe de deformação visa permitir que objetos manipulados pelo usuário apresentem características de reação ao contato. Neste caso, os objetos inseridos na cena podem reagir visualmente quando tocados pelo objeto interador. Existem diferentes métodos de deformação, mas alguns parâmetros comuns podem ser utilizados para definir algumas das propriedades associadas aos modelos. Estes parâmetros são utilizados durante a

simulação para o cálculo do reposicionamento dos pontos da malha que define o modelo tocado.

Uma vez que a deformação só é ativada quando ocorre um contato, a classe de detecção de colisão é fundamental para identificar tal ocorrência. Essa ocorrência deve fornecer à classe de deformação a posição do toque para que a região correspondente seja modificada. Quando a deformação for permanente, os novos pontos do objeto devem ser informados à ED para uso nas demais classes do CyberMed. Todo o processo precisa estar sincronizado utilizando um método de concorrência (*thread*) para que a visualização e a detecção de colisão utilizem os dados atualizados. Os principais métodos da classe de deformação são apresentados na Tabela 1.2.

Tabela 1.2: Métodos da classe de deformação que podem ser utilizados

CybNarrowCollision* getCollisionObject();	Retorna uma referência para o objeto de colisão associado.
float getDamping();	Retorna o amortecimento associado ao objeto em questão.
EnumDefMode getDeformationMode();	Retorna o modo de deformação. Pode retornar os valores DEF_GO_AND_BACK ou DEF_ONLY_GO.
float getError();	Retorna o erro aceitável que está associado aos cálculos realizados sobre o objeto em questão.
float getStep();	Retorna o passo da discretibilidade dos cálculos realizados.
float getStiffness();	Retorna a constante molar associada à malha do objeto.
void init();	Cria a *thread* do objeto em questão e do objeto de colisão associado.
void setError(float error);	Define o erro aceitável para os cálculos.
void setStep(float step);	Define o passo para a discretibilidade dos cálculos.

O sistema possui o método de deformação massa-mola [Marciel 2003] que, associado com métodos de detecção de colisão do tipo CybNarrowCollision, pode incorporar características de deformação aos objetos virtuais. Esta utilização pode ser dada sob dois contextos: no primeiro deles, apenas o filho de CybNarrowCollision irá verificar a detecção de colisão, já no segundo, são utilizados dois métodos de detecção agrupados: um filho de CybNarrowCollision e o um filho de CybBroadCollision. Na Figura 11 é apresentado um exemplo de código utilizando a deformação juntamente com a colisão.

```
int main()
{
  //load objects to the data structure

  // apply spring-mass deformation over object of layer 1,
  // uses sphere-triangle method for collision detection
  CybMassSpring* ms1 = new CybMassSpring(new CybSphereTriangle(1));

  // first test using a broad method,
  // if collision occurs the region is send to a narrow method
  // to identify the collision point.
  CybMassSpring* ms2 =
     new CybMassSpring(new CybSphereTriangle(new CybSomeBroadCollision(1)));
};
```

Figura 11: Código apresentando os dois apectos de utilização da deformação.

O diagrama das classes de deformação contidas no CyberMed é apresentado na Figura 12. Neste diagrama pode ser vista a classe de deformação massa-mola, cujo método já está implementado, e sua interação com o sistema de detecção de colisão. Nesta figura é possível observar também que o uso de deformação depende de algum método de detecção de colisão do tipo *narrow*.

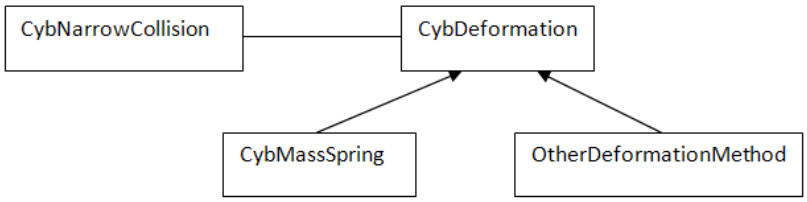

Figura 12: Diagrama de classes da Deformação do Sistema CyberMed.

1.8. Interação

A classe de Interador é responsável pelo gerenciamento e controle dos dados relacionados aos interadores utilizados no sistema. Neste caso, deve prever a utilização de dispositivos convencionais de interação, bem como suportar a adição de dispositivos específicos. Conforme previsto no projeto, o dispositivo apontador convencional (*mouse*) é suportado, bem como dispositivos de interação que oferecem retorno de força (dispositivos hápticos). Entretanto, a representação de um objeto interador é geralmente feita através de um objeto gráfico tridimensional. Por esta razão, o objeto interador utiliza uma malha de pontos de um modelo 3D, que é provida pela estrutura de dados. Os dados do interador também podem ser acessados pelas tarefas de detecção de colisão e avaliação para verificar o deslocamento do objeto manipulado pelo usuário na cena.

Através do interador tem-se o acesso às funcionalidades dos dispositivos integrados ao CyberMed. Quando, por exemplo, deseja-se acessar a posição do mouse ou qualquer outro dispositivo integrado ao sistema, invoca-se o objeto desta classe. Com a finalidade de referenciar unicamente as informações de interadores no CyberMed a classe de interação foi desenvolvida utilizando o padrão *singleton*, que garante a instanciação de um único objeto da classe. Outra funcionalidade desta classe é a configuração da representação visual do objeto interador. Através da classe é possível

definir a posição do objeto gráfico que representa o interador em função da posição pontual do mesmo dentro da cena.

A Figura 13 mostra um exemplo de uma aplicação onde se utiliza o mouse como objeto interador e configura-se o objeto gráfico em função da posição real do mouse. Na linha 16 indica-se a CybInterator o identificador do mouse. Na linha 19 indica-se informações referentes ao posicionamento da representação gráfica do interador na cena.

```
 1  // Transfer the OF data to CybParameters structure
 2  CybDataObtainer<cybTraits> data(numLayer, numInterator);
 3  // Anaglyph Visualization
 4  CybViewAnaglyph view;
 5
 6  //Access the parameters information of scene and graphical objects
 7  CybParameters *cybCore = CybParameters::getInstance();
 8
 9  /*Load the model from VRML file (layerID, fileName)*/
10  data.loadModel(0, fileName);
11
12  /*Initialize the meshes parameters*/
13  data.startParameters(numLayer);
14
15  /*Switcing to mouse interaction*/
16  integrator.setObjectType(0);
17
18  /*Configuring graphical representation of mouse */
19  Integrator.setInteratorAttribute(0, 0.0,0.0,-4.34);
20
21  /*Initialize visualization*/
22  view.init();
```

Figura 13: Aplicação onde o mouse é utilizado com objeto interador.

Atualmente, outra categoria de dispositivo suportada pelo CyberMed é a dos dispositivos hápticos. Particularmente, este suporte já é completo para dispositivo da família Phantom (www.sensable.com). Caso exista a necessidade de se manipular este dispositivo deve-se acessá-lo através da classe de interadores. As regras para configuração de sua representação visual são as mesmas de qualquer outro interador, precisando que o desenvolvedor apenas se preocupe com o identificador do dispositivo em questão. A Figura 14 mostra como acessar e configurar um dispositivo háptico através da classe CybInterator. Na linha 17 indica-se o identificador do dispositivo háptico, que utilizaremos para interação. Na linha 20 indicamos a posição relativa do objeto gráfico em função da posição do interador na cena. Por fim, na linha 23 acessamos a posição do dispositivo háptico.

Vale salientar que dispositivos que possuem um grau de complexidade mais elevado no que diz respeito à aquisição de informações são apenas instanciados na interator. Entretanto, suas próprias classes é que devem prover acesso a estas informações. Isto pode ser observado na linha 23 da Figura 14, onde é acessada a posição do dispositivo háptico através de sua instância.

```
// Transfer the OF data to CybParameters structure
CybDataObtainer<cybTraits> data(numLayer, numInterator);

// Anaglyph Visualization
CybViewAnaglyph view;

//Access the parameters information of scene and graphical objects
CybParameters *cybCore = CybParameters::getInstance();

/*Load the model from VRML file (layerID, fileName)*/
data.loadModel(0, fileName);

/*Initialize the meshes parameters*/
data.startParameters(numLayer);

/*Switching to haptic interaction*/
integrator.setObjectType(3);

/*Configuring graphical representation of haptic device */
integrator.setInteratorAttribute(0, 0.0,0.0,-4.34);

//Accessing information about haptic device position
interator.hapticDevice.getPosition(position);

/*Initialize visualization*/
view.init();
```

Figura 14: Aplicação que utiliza um dispositivo háptico como interador.

1.8.1. Sistemas Hápticos

Os seres humanos exploram objetos em muitas etapas. Primeiro fazem a varredura visual do ambiente com seus olhos para encontrar a posição do objeto, tocam então no objeto para sentir sua forma geral. Finalmente, é feita uma exploração manual mais cuidadosa para investigar as características de superfície do material que forma o objeto, sua maciez, textura e forma. Esta detecção manual é feita através de sistemas sensoriais táteis e cinestésicos que respondem a distribuição espacial e temporal das forças na mão do usuário [Basdogan 2004]. Quando estes objetos são tratados no contexto virtual, entra em cena a área de RV chamada de interação háptica.

O módulo *Haptics* é um conjunto de classes que compõe o CyberMed. O objetivo deste sub-sistema é oferecer uma interface mais intuitiva e abstrata aos usuários que utilizam o CyberMed, provendo uma séria de facilidades quanto ao acesso e integração de sistemas hápticos a aplicações baseadas em RV, bem como a criação e gerenciamento das rotinas de renderização háptica.

A CybHaptics foi desenvolvida para prover acesso alto nível para rotinas hápticas, como gerenciamento de dispositivos e renderização [Souza 2007b]. Contudo, seu design é flexível o bastante para permitir acesso baixo nível para programadores mais avançados. A CybHaptics possui uma implementação abstrata das funcionalidades mais comuns encontradas em dispositivos hápticos diversos. Para prover suporte a um dispositivo em particular o desenvolvedor deve herdar e implementar as funcionalidades das rotinas descritas por esta classe abstrata. Assim, nNovos dispositivos podem ser adicionados a CybHaptics através do mecanismo de herança. Isto irá permitir que não apenas que a integração de dispositivos dentro do módulo *Haptics* bem como, a todo o CyberMed. A Figura 15 mostra como esta integração pode ser feita para muitos dispositivos hápticos: uma classe usuário (HapticsDeviceInterface) deve herdar a classe abstrata Cybhaptics e prover uma interface entre as rotinas dos dispositivos e o

CyberMed. Algumas das funcionalidades oferecidas pela CybHaptics são: inicializar renderização háptica, atualizar cena háptica, acessar informações de posição e força do dispositivo, dentre outras.

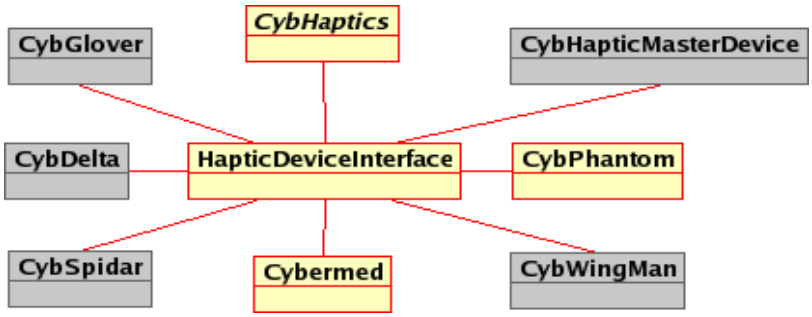

Figura 15: Integração de dispositivos hápticos ao CyberMed.

Atualmente o módulo Haptics oferece suporte para a família de dispositivos hápticos Phantom (www.sensable.com). A classe CybOpenHPhantom foi utilizada para integrar o sistema a dispositivos desta família. Este processo para integração seguiu o roteiro descrito nos paragráfos acima. Basicamente, ela provê uma fachada entre as rotinas de baixo nível do dispositivo háptico e o CyberMed através das regras descritas pela CybHaptics.

Outra classe do sistema Haptics, chamada de CybPhantom, provê o acesso de baixo nível às funcionalidades do dispositivo háptico. Esta classe foi implementada utilizando a API nativa do dispositivo háptico da família específica dos dispositivos em questão. Trabalhando em conjunto com a CybOpenHPhantom, estas classes são responsáveis pela interação háptica com estes dispositivos no CyberMed. Dentre suas funcionalidades pode-se destacar a possibilidade de toque em malhas complexas, a manipulação de propriedades materiais destas malhas e a inclusão de propriedades de ambiente. Quanto à inclusão de propriedades de ambiente, a CybPhantom atualmente oferece suporte à simulação de viscosidade. A Figura 16 mostra o diagrama de classes usado para integrar dispositivos da família Phantom ao CyberMed.

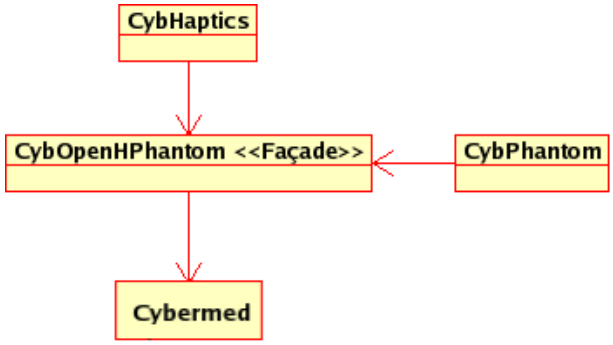

Figura 16: Classes que oferecem suporte ao dispositivo Phantom.

Para a inclusão de interação háptica nas aplicações desenvolvidas com o CyberMed basta apenas que o usuário possua o dispositivo háptico configurado em seu

PC e compile o CyberMed indicando que quer incluir o pacote Haptics. Caso o usuário deseje habilitar ou desabilitar o dispositivo háptico basta modificar este estado acessando uma instância da camada Core. Informações a respeito da posição do dispositivo na cena, forças aplicadas, torque e estados podem ser acessadas através da classe CybOpenHPhantom, no caso do uso da classe CybPhantom. A Figura 17 mostra trecho de código onde o usuário acessa informações sobre posição, força e torque do dispositivo. Nas linhas 12 e 16 o são acessadas informações sobre a posição do dispositivo háptico. Nas linhas 13 e 17 são acessadas informações sobre forças aplicadas pelo dispositivo háptico.

```
 1 //Requesting interator instance
 2 CybInterator interator = CybInterator::getInstance();
 3
 4 //Some declarations
 5 double *position = new double[3];
 6 double *force = new double[3];
 7 double *torque = new double[3];
 8 double *lastPosition = new double[3];
 9 double *lastForce = new double[3];
10
11 //Getting informations
12 interator.hapticDevice.getPosition(position);
13 interator.hapticDevice.getForce(force);
14 interator.hapticDevice.getTorque(torque);
15
16 interator.hapticDevice.getLastPosition(lastPosition);
17 interator.hapticDevice.getLastForce(lastForce);
```

Figura 17: Aquisição de informação sobre o dispositivo háptico.

Quanto à inclusão de propriedades materiais, esta é feita através da camada *Core.* Isto se deve ao fato de o CyberMed centralizar as informações de materiais dos objetos, uma vez que outros sub-sistemas, como o de deformação, também utilizam estas propriedades para seus cálculos internos. A Figura 18 mostra um trecho de código onde as propriedades materiais são configuradas para utilização com o dispositivo háptico. Na linha 8 cria-se as camadas para o contexto háptico, as informações referentes aos modelos são acessadas internamente através da estrutura de dados.. Na linha 11 é criado o contexto das propriedades materiais. Das linhas 15 a 23 são configurados os valores das propriedades materiais das camadas, bem como a face onde a propriedade será aplicada. Por fim, na linha 25 a renderização das propriedades é habilitada.

A inclusão de viscosidade é semelhante à inclusão de propriedades materiais. A diferença está no fato de a viscosidade ser gerenciada pelas próprias rotinas do sistema háptico, de forma que não é necessário executar chamadas à camada *Core*. A Figura 19 mostra exemplo de código necessário para a inclusão de viscosidade. Na linha 2 é acessada a instancia da classe viscosidade. Na linha 5 cria-se o contexto para a utilização de propriedades de ambiente. Da linha 7 a 10 configura-se as propriedades. Para finalizar, na linha 12 é habilitada a renderização da propriedade de ambiente.

```
 1 //Requesting interator instance
 2 CybInterator interator = CybInterator::getInstance
 3
 4 //Requesting core instance
 5 CybParameters *cybCore = CybParameters::getInstance();
 6
 7 //Creating the haptics layers
 8 interator.hapticDevice.createHapticLayers(numLayer,true);
 9
10 //Creating the material property context
11 interator.hapticDevice.createMaterialPropertyContext(numLayer, true);
12
13 //Configuring properties
14 cybCore->setMaterialPropertyValue(0, DAMPING, 0.1f);
15 cybCore->setMaterialFace(0, DAMPING, FRONT_AND_BACK);
16
17 cybCore->setMaterialPropertyValue(0, STATIC_FRICTION, 0.1f);
18 cybCore->setMaterialFace(0, STATIC_FRICTION, FRONT_AND_BACK);
19
20 cybCore->setMaterialPropertyValue(0, DYNAMIC_FRICTION, 0.9f);
21 cybCore->setMaterialFace(0, DYNAMIC_FRICTION, FRONT_AND_BACK);
22
23 //Enabling material property rendering
24 interator.hapticDevice.enableHapticMaterialProperty();
```

Figura 18: Configuração de propriedades materiais no CyberMed.

```
 1 //Requesting viscosity instance
 2 CybViscosity *viscosity = CybViscosity::getInstance(numLayer);
 3
 4 //Creating ambient property
 5 interator.hapticDevice.createAmbientProperty();
 6
 7 //Configuring ambient
 8 viscosity->initStatus(numLayer, HL_FRONT, 0.0, 0.4, 0,0,numLayer);
 9 viscosity->initStatus(0, HL_FRONT, 0.0, 0.2, 2.0,0,0);
10
11 //Enabling ambient property
12 interator.hapticDevice.enableAmbientProperty();
```

Figura 19: Configuração da propriedade material de ambiente.

1.9. Avaliação

O processo de avaliação do usuário depende da descrição, efetuada por um especialista, daquilo que é considerado certo, errado e suas variações para que seja constituído um modelo (estatístico [Moraes 2005a], baseado em regras [Machado 2000], cognitivo [Moraes 2005b], etc.) para realizar as futuras avaliações do usuário. Tais informações são compostas e armazenadas em classes ou bancos de desempenho. A forma como isto é feito depende do método de avaliação em uso e dos parâmetros que foram selecionados para serem monitorados durante a execução da aplicação. O conjunto de parâmetros definidos e o banco de desempenho serão utilizados para avaliar o usuário da aplicação. Desse modo, a classe de avaliação deve suportar a calibração das classes de desempenho, bem como avaliar um usuário de acordo com os dados de execução e os dados das classes de desempenho. Tais funcionalidades devem ser utilizadas de acordo com o propósito da aplicação: quando será utilizada para gerar as classes de avaliação e quando será utilizada para treinar um usuário.

Na prática a avaliação é feita como uma inferência sobre um conjunto de variáveis que assumem valores reais. Uma classe de avaliação determina a qualificação

de uma simulação, tal como excelente, bom, regular ou ruim. Assim, um sistema de avaliação deve mapear um conjunto de valores de um conjunto de variáveis para um conjunto de classes, previamente designadas. Isto é feito na etapa de treinamento do avaliador, onde um especialista treina o sistema, usando o simulador, que capta valores de variáveis que possam ser aferidas e as atribui a classes pré-definidas. Já na etapa de avaliação o usuário executa o treinamento e o sistema monitora as variáveis de interação, medindo a similaridade do seu conjunto aos modelos das classes armazenadas. Ao final do treinamento, o sistema de avaliação emite um relatório com essas similaridades ao conjunto de classes.

Atualmente, o CyberMed dispõe do método de avaliação feita pela técnica de Máxima Verossimilhança [Moraes 2005a] [Cunha 2007a]. Para se fazer a utilização deste método exite a classe CybAsses que já está ligada com o interador do sistema através da classe CybParameters. A CybAsses [Cunha 2007b] herda da CybThread, ou seja, implementa o método de avaliação dentro de uma *thread*.

Na etapa de treinamento do sistema, o especialista utiliza o sistema diversas vezes para gerar dados (advindos de diversas variáveis que possam ser aferidas no processo de treinamento) que o sistema armazena em arquivos. A partir destes dados, o sistema calibra o método de avaliação criando os seus parâmetros dos modelos. Na etapa de avaliação do treinamento, realizado pelo usuário, os modelos são utilizados para avaliar o treinamento em questão. O sistema não prevê um formato específico para o armazenamento dos parâmetros dos modelos, visto que eles podem ser diferentes para cada método incorporado.

A incorporação de novos métodos de avaliação pode ser feita independentemente do sistema. Como a avaliação pode ser ativada de várias formas, tal como: pelo tempo decorrido desde o início da simulação, ao clicar em uma opção do menu, ou com uma determinada ação, o programador fica livre para implementá-la do seu modo.

O uso do método de avaliação atualmente implementado é feito com o uso de um menu que, interligado com a classe de visualização, auxilia na ativação dos métodos de aprendizado das classes, bem como na identificação de cada uma delas. A classificação pode ser feita ao fim de um determinado treinamento. A classe CybAssessMenu implementa este menu e integra a avaliação com todo o sistema. Para se utilizar essa classe, um objeto seu deve ser passado como parâmetro para o objeto que representa a visualização corrente. A Figura 20 apresenta uma forma de utilização da avaliação.

```
int main()
{
  ...
  CybViewMono view(new CybAsssessMenu());
  view.init();
}
```

Figura 20: Ativação da avaliação com o sistema CyberMed.

1.10. Desenvolvendo uma aplicação completa

Um simulador completo, utilizando todas as funcionalidades do sistema CyberMed pode ser construído utilizando-se do código contido na Figura 21. Neste exemplo é gerada uma aplicação que utiliza o método de anaglifo real (linha 8) para visualização do objeto carregado (linha 14). Este objeto pode ser tocado com um interador padrão (pré-definido no CyberMed) e deformado com o método massa-mola, utilizando para detecção o método esfera-triângulo (linha 26). A coleta dos movimentos do interador háptico é realizada uma vez que o método de avaliação foi embutido no código (linha 8). Entretanto, ela dependerá da sua ativação em tempo de execução através de menu aberto com o mouse. O resultado desta aplicação pode ser observado na Figura 22.

```
 1  int main(int argc, char** argv)
 2  {
 3    char *fileName = "macaco.wrl"; // object file
 4    int numLayer = 1;              // number of objects
 5    int numInterator = 1;          // number of interators
 6
 7    // Anaglyph Visualization with assessment
 8    CybViewAnaglyph view(new CybAssessMenu());
 9
10    //Access the parameters information of scene and graphical objects
11    CybParameters *cybCore = CybParameters::getInstance();
12
13    /*Load the model from VRML file (layerID, fileName)*/
14    data.loadModel(0, fileName);
15
16    /*Initialize the meshes parameters*/
17    data.startParameters(numLayer);
18
19    /*Switching to haptic interaction*/
20    integrator.setObjectType(3);
21
22    /*Configuring graphical representation of haptic device */
23    integrator.setInteratorAttribute(0,0,0,0);
24
25    //apply deformation (with collision detection)
26    CybMassSpring *ms = new CybMassSpring ( new CybSphereTriangle(0));
27    ms->init();               // creates deformation thread
28
29    /*Initialize visualization*/
29    view.init();
30  }
```

Figura 21: Código de um simulador completo, integrando visualização, detecção de colisão, deformação e avaliação.

1.11. Expansões

A cada dia surgem novas técnicas e métodos para a exploração dos recursos computacionais e seu emprego na área de RV. Neste intuito, uma das características que se procurou incluir quando da concepção do CyberMed foi que o mesmo pudesse permitir expansões futuras. Uma série de estudos está sendo feita em áreas diversas para a inclusão de novas funcionalidades ao conjunto de bibliotecas. Pretende-se com isto prover uma gama maior de possibilidades para o usuário desenvolvedor. Atualmente, pesquisas na área de colaboração e rastreamento estão sendo executadas com a finalidade de introduzir estes conceitos ao sistema.

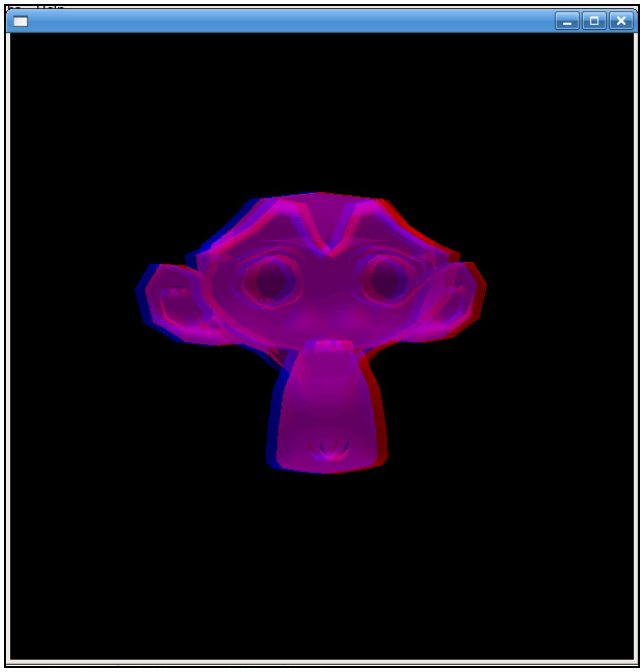

Figura 22: Janela gráfica com imagem da aplicação desenvolvida com o código da Figura 21.

1.11.1. Colaboração

A interação pode ser incorporada a uma simulação com apenas um usuário ou com múltiplos usuários o que, em geral, requer uma rede de comunicação (Internet ou uma rede dedicada). Quando há apenas um usuário, *o uso de sistemas hápticos pode* oferecer sensação de toque, tornando a interação com o ambiente mais realista, uma vez que o usuário pode perceber a forma geométrica dos objetos que compõem o ambiente e suas características materiais, como rigidez e textura. Entretanto, quando há vários usuários, além do toque também se torna necessário oferecer uma sensação de presença e de proximidade entre os usuários que interagem com o mesmo ambiente [Sallnäs 2000][Basdogan 2000]. A sensação de presença é o que faz o usuário se sentir no ambiente com o qual se está interagindo, seja de forma virtual ou remota. A sensação de proximidade provém do fato dos usuários poderem perceber as ações dos demais no ambiente compartilhado, podendo tocar e manipular os mesmos objetos, como em uma experiência real.

O suporte à colaboração está em desenvolvimento e deve integrar versões futuras do CyberMed, possibilitando que também aplicações colaborativas possam ser desenvolvidas de forma mais simplificada.

1.11.2. Rastreamento

Outra área interessante e que vêm sendo bastante explorada é a área de rastreamento espacial. Um dos fatores determinantes dentro da área de RV é a interatividade, sendo que esta pode se dar de várias formas. Porém, uma das maiores dificuldades em se ter interação com ambientes de RV é o fato de quão intuitivo é para o usuário a forma de interação. Dessa forma, a busca por novas formas de interação leva a pesquisa e

desenvolvimento de novas técnicas para que o usuário possa ter uma boa experiência com a aplicação.

Uma vez que o CyberMed é uma ferramenta que auxilia o usuário no desenvolvimento de aplicações baseadas em RV, a inclusão de interatividade através de sistemas de rastreamento o torna mais robusto, oferecendo ao usuário um número maior de funcionalidades para dar suporte a construção de simuladores. Dessa forma, um sistema de rastreamento está sendo concebido e será integrado em futuras versões do CyberMed.

1.12. Conclusões

Este tutorial apresentou uma breve introdução sobre as necessidades e dificuldades relacionadas ao desenvolvimento de aplicações de RV. Neste contexto, pode-se observar que as diversas necessidades demandadas por estes sistemas podem exigir a implementação de diversas técnicas e métodos dependentes de tecnologias além de um núcleo para sincronização das diferentes tarefas. Outro aspecto ainda pouco explorado em sistemas de RV é o monitoramento das ações do usuário cujas informações podem ser utilizadas para avaliar, qualificando ou quantificando, seu desempenho ou o grau de compreensão daquilo que o sistema de RV apresenta.

Visando agilizar o processo de desenvolvimento foi apresentado o uso do pacote CyberMed, cujas funcionalidades englobam visualização, interação, deformação interativa, detecção de colisão e avaliação do usuário, bem como oferece suporte a alguns dispositivos específicos de RV. Através de trechos de código foram demonstradas formas de utilizar tais funcionalidades, bem como expandí-las. Finalmente foi apresentada uma aplicação desenvolvida utilizando todas as funcionalidades disponibilizadas pelo CyberMed.

Uma das vantagens de utilizar o pacote apresentado é que, além dos benefícios relacionados ao desenvolvimento de sistemas de RV, seu código é livre e aberto, permitindo realizar expansões, adicionar novos métodos e funcionalidades.

Referências

Basdogan, C.; Ho, C.; Srinivasan, M. A.; Slater, M. (2000) “An Experimental Study on the Role of Touch in Shared Virtual Environments”. *ACM Transactions on Computer-Human Interaction*, **7**(4), p. 443-460.

Basdogan, C.; De, S., Kim; J., Muniyandi, M.; Srinivasan, M.A. (2004) “Haptics in Minimally Invasive Surgical Simulation and Training”, *IEEE Computer Graphics and Applications*, **24**(2), p. 56-64.

Bergen, G.V. (2004) Collision Detection in Interactive Environments 3D. Morgan Kaufman Publishers.

Cunha, I.L.L.; Moraes, R. M.; Machado, L. S. (2006). “CybView - Uma Classe para Visualização Interativa e Estereoscópica para Sistemas de Realidade Virtual”. *Anais do XIX Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI'2006) [cdrom]*. Outubro, Manaus, Brasil.

Cunha, I.L.L.; Moraes, R. M.; Machado, L. S. (2007a). "Performance Analysis of Assessment Using Maximum Likelihood for Virtual Reality Systems". *Proc. Int. Conference on Engineering and Computer Education (ICECE'2007),* p. 306-310.

Cunha, I.L.L.; Moraes, R. M.; Machado, L. S. (2007b). "CybAssess: A Class for Online Assessment of Users in a Virtual Reality System". *Proc. of Symposium on Virtual Reality (SVR'2007).* Maio/Junho, Petrópolis, Brasil.

Gamma, E. et al. (2000) Padrões de Projeto. Editora Bookman. Porto Alegre.

Luque, R.G. et al. (2005) "Broad-Phase Collision Detection Using Semi-Adjusting BSP-trees". *Proc.* 2005 *Symposium on Interactive 3D graphics and games*, p. 179-186.

Machado, L.S.; Moraes, R.M. and Zuffo, M.K. (2000). "A Fuzzy Rule-Based Evaluation for a Haptic and Stereo Simulator for Bone Marrow Harvest for Transplant". *Proc. of Phantom Users Group Workshop*.

Marciel, A.; Boulic, R.; Thalmann, D. (2003) "Deformable Tissue Parameterized by Properties of Real Biological Tissue", *Surgery Simulation and Soft Tissue Modeli ng: International Symposium*.

Moraes, R.M. e Machado, L.S. (2004) "Using Fuzzy Hidden Markov Models for Online Training Evaluation and Classification in Virtual Reality Simulators". *Int. Journal of General Systems*, **33**(2-3), p. 281-288.

Moraes, R.M.; Machado, L.S. (2005a) "Maximum Likelihood for On-line Evaluation of Training Based on Virtual Reality". *Proc. Global Congress on Eng. and Tech. Education*, p.299-302.

Moraes, R.M.; Machado, L.S. (2005b) "Evaluation System Based on EFuNN for On-line Training Evaluation in Virtual Reality". *Lecture Notes in Computer Science.* Berlim, v. 3773, p. 778-785.

Sallnãs, E.; Rassmus-Gröhn, K.; Sjöström, C. (2000) Supporting Presence in Collaborative Environment by Haptic Force Feedback. *ACM Transactions on Computar-Human Interaction*, **7**(4).

Sharabi, C. et al. (2007) "Immersidata Analysis: Four Case Studies". *IEEE Computer*, **40**(7), p. 45-52. 2007.

Souza, D.F.L.; Cunha, I.L.L.; Souza, L.C.; Moraes, R.M.; Machado, L.S. (2007a) "Development of a VR Simulator for Medical Training Using Free Tools: A Case Study". *Proc. SVR2007*, p. 100-105. Petrópolis.

Souza, D.F.L.; Moraes, R.M.; Machado, L.S. (2007b). "CybHaptics: A Class for Haptics Support in Virtual Reality Systems". *Proc. of Symposium on Virtual Reality (SVR'2007).* Petrópolis, Brasil.