PUC

Lúcia Teresa Schalcher da Fonseca

Uma Arquitetura para Construção de Ferramentas de Manipulação para Visualização Interativa de Dados Volumétricos

Dissertação de Mestrado

Departamento de Informática

Pontifícia Universidade Católica do Rio de Janeiro

Rio de Janeiro, 23 de setembro de 1997

Lúcia Teresa Schalcher da Fonseca

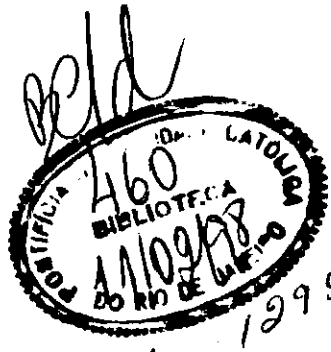
Uma Arquitetura para Construção de Ferramentas de Manipulação para Visualização Interativa de Dados Volumétricos

Dissertação apresentada ao Departamento de Informática da PUC-Rio como parte dos requisitos para obtenção do título de Mestre em Informática: Ciência da Computação

Orientador: Luiz Fernando Martha
Co-orientador: Marcelo Gatass

Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro

Rio de Janeiro, 23 de setembro de 1997



A Luiz Eduardo e Luiza.

Agradecimentos

A Petrobras, pela oportunidade de realização do curso de Mestrado e pelo suporte operacional a este trabalho.

A Luiz Fernando Martha, orientador da dissertação, pelo interesse, incentivo e colaborações.

Aos professores do curso de Mestrado, em especial a Marcelo Gatass, pelos ensinamentos.

Aos colegas de trabalho e amigos Luciano Pereira dos Reis, pelo incentivo e cooperação no desenvolvimento desta dissertação e Beatriz Castier, responsável, juntamente com Luciano, pelo desenvolvimento do trabalho que originou as idéias para a dissertação.

A todos os colegas da Petrobras, pelo apoio e incentivo, em especial a Flavio Yuan Gouveia, pelo apoio técnico, sempre que foi preciso.

A Luiz Eduardo, pela força e compreensão.

Resumo

Na área de exploração e produção (E&P) da indústria de petróleo, a utilização da visualização tridimensional representou uma grande melhoria no trabalho dos geocientistas. Essa tecnologia possibilita criar imagens 3D precisas de formações da subsuperfície e interpretar os dados de exploração e produção com maior acurácia e rapidez.

Na Petrobras, representações bidimensionais dos dados, como seções sísmicas, mapas de contorno, seções geológicas e *grids* de propriedades, ainda são muito utilizadas pelos geólogos, geofísicos e engenheiros de reservatório. Mas o interesse em trabalhar com os dados em uma forma tridimensional vem crescendo bastante e motivou o desenvolvimento de um *toolkit* orientado a objetos, para facilitar a construção de aplicações 3D interativas para visualização e interpretação de dados de exploração e produção.

Esta dissertação propõe a integração de interfaces de manipulação direta com as representações de dados implementadas no *toolkit*, na forma de *widgets* 3D (uma combinação de geometria e comportamento para controlar objetos da aplicação), de modo a possibilitar uma exploração mais interativa dos dados.

Uma arquitetura para a construção de *widgets* 3D (neste trabalho chamados de manipuladores) e um conjunto de manipuladores para interação com visualizações de dados sísmicos volumétricos são implementados. A metodologia de orientação a objetos também guiou o projeto e implementação da arquitetura e, assim, à medida que o *toolkit* for sendo estendido com novos dados e técnicas de visualização, novos manipuladores poderão ser incorporados.

Abstract

The use of three-dimensional visualization in the exploration and production area (E&P) of the petroleum industry represented a great improvement in the work of geoscientists. With this technology, precise 3D images of the subsurface formations can be created and the interpretation of exploration and production data can be done faster and more accurately.

In Petrobras, geologists, geophisycists and reservoir engineers still use two-dimensional data representations in their work, like seismic sections, contour maps, cross sections and grids of properties. But the interest in working with data in a three-dimensional way is increasing and have motivated the development of an object-oriented toolkit to facilitate the construction of interactive 3D applications for exploration and production data visualization and interpretation.

The integration of direct manipulation interfaces to the data representations implemented in the toolkit, to provide a more interactive data exploration, is the aim of this work. It proposes the use of widgets 3D, a combination of geometry and behavior to control application objects.

An architecture for the construction of widgets 3D, in this work called manipulators, and a set of manipulators for interaction with volumetric seismic data visualizations are implemented. The object-oriented methodology was used in the development of this architecture too. New manipulators can be incorporated as new data and visualization techniques extend the toolkit.

Sumário

Lista de Figuras

Lista de Tabelas

1. Introdução

Como em outros campos, a visualização tridimensional está transformando a indústria de exploração e produção de petróleo. À medida que as companhias de petróleo passam a adquirir e a interpretar dados 3D, o potencial da visualização por computador vai sendo melhor entendido e as necessidades para a continuidade do seu desenvolvimento vão sendo definidas de forma mais clara [Uchida (1994)]. Interfaces interativas para manipulação de representações de dados 3D são os objetos de estudo desta dissertação.

1.1 Análise de Dados Científicos

A análise de dados científicos é o processo de extrair de uma grande quantidade de dados, medidos ou calculados, regras ou parâmetros que caracterizem o fenômeno sob investigação. O que os cientistas querem é compreender o fenômeno e, nesse processo, utilizam imagens como produtos secundários.

Springmeyer et al. (1992) fizeram um estudo empírico sobre como os cientistas realizam o trabalho de análise, caracterizando como a visualização de dados se encaixa dentro do processo mais amplo de análise de dados científicos e sugerindo recomendações para o projeto de ferramentas de análise. Dentre as recomendações sugeridas, está a de facilitar a exploração ativa, ou seja, prover representações visuais qualitativas (para identificação de características dos dados) e permitir exploração quantitativa interativa (para determinar quantidades ou proporções das características identificadas) através de, por exemplo, comparação de resultados, recuperação de informações, aplicação de operações matemáticas e incorporação de resultados intermediários.

Tal estudo resultou em uma decomposição do processo de análise de dados científicos em dois ramos primários de atividade: **Investigação**, exploração dos dados para extrair informação ou confirmar resultados, e **Integração do *Insight***, assimilação do conhecimento resultante da análise, integrando-o à base de conhecimento e experiências do cientista.

A Investigação abrange atividades de organização dos dados, de escolha de representações, de interação com as representações dos dados e de aplicações matemáticas sobre os dados. Interagir com as representações dos dados envolve gerar as representações, examiná-las, aplicar ações que as altere, questionar, comparar resultados e classificar os dados.

1.2 Análise de Dados de Exploração e Produção

Geofísicos, geólogos e engenheiros de reservatório trabalham na busca e desenvolvimento de reservas de óleo e gás e, nesse trabalho, analisam dados de diversas fontes (dados de poços, de análise sísmica e de produção de petróleo).

Sem o uso de *hardware* e *software* 3D, a visualização e a interpretação de dados de exploração são tarefas muito mais trabalhosas. Dados de geofísica são plotados em seções sísmicas e, utilizando lápis colorido, os geofísicos pintam sobre elas **horizontes**[1] e **formações geológicas**[2]. Combinando-os com outros dados, eles tentam interpretar a geologia de uma porção da subsuperfície da terra. Para visualizar a interpretação, são utilizadas representações bidimensionais como o **mapa de contorno**, que representa a superfície de uma formação geológica em um plano horizontal através de linhas de contorno, e a **seção geológica**, que representa a subsuperfície em um plano de corte vertical. Com essas

[1]Superfícies que separam formações geológicas

[2]Regiões da subsuperfície classificadas de acordo com alguma característica geológica como tipo ou idade da rocha, conteúdo fossilífero ou tipo de fluido.

ferramentas, é mais difícil compreender a complexidade geométrica e espacial da subsuperfície. Outra característica do trabalho realizado desta forma é a independência do trabalho dos geofísicos, geológos e engenheiros, que interpretam os dados por eles coletados e resolvem os conflitos de interpretação em apresentações gerenciais.

O advento de estações de trabalho gráficas 3D e o aprimoramento das técnicas de visualização tridimensional estão facilitando a transição do processo de análise de dados científicos de 2D para 3D. Na área de exploração e produção de petróleo já se vê, como resultado, a maior integração entre os diferentes dados e a geração de modelos tridimensionais exatos da subsuperfície [Uchida (1994)].

Na Petrobras, a demanda por aplicações 3D na área de exploração e produção de óleo (**E&P**), envolvendo grande volume de dados, motivou o desenvolvimento de um *toolkit* orientado a objetos, para facilitar a construção dessas aplicações. A utilização desse *toolkit* permite o desenvolvimento das aplicações em um nível de abstração mais alto que o proporcionado com a utilização direta de bibliotecas padrões de visualização, como o *OpenGL* [Reis (1996)].

1.3 Tema e Propósito da Dissertação

A análise de dados científicos, como verificado, não é um processo passivo. Mesmo com poderosos efeitos visuais, não interessa aos analistas a visualização de representações inflexíveis. É necessário dar a eles maior controle sobre a exploração dos seus dados.

1.3.1 Objetivos

Esta dissertação tem como objetivo prover o *toolkit* para desenvolvimento de aplicações interativas na área de **E&P** de um ambiente extensível de exploração interativa

para ajudar geofísicos, geólogos e engenheiros da Petrobras no seu trabalho de análise. Uma arquitetura para construção de interfaces interativas de manipulação de representações de dados e um modelo de interação foram concebidos e desenvolvidos. Com o propósito de avaliar o projeto e a implementação da arquitetura e do modelo de interação, um protótipo, com um conjunto de interfaces interativas para manipulação de representações de dados volumétricos, foi produzido.

1.3.2 Aplicação Potencial: Análise de Dados Sísmicos Volumétricos

O protótipo desenvolvido utilizou dados sísmicos volumétricos para gerar as representações de dados para o conjunto de interfaces interativas implementado. Uma visão geral do dado sísmico é apresentada a seguir, salientando-se a sua importância para a busca de reservas de óleo e gás.

A gravação de reflexões sísmicas tem como objetivo básico medir o tempo necessário para que um sinal (onda sísmica), emitido por determinada fonte, reflita em uma ou mais descontinuidades do solo (superfícies que separam camadas de rochas não similares) e retorne a um ponto receptor (Figura 1-1).

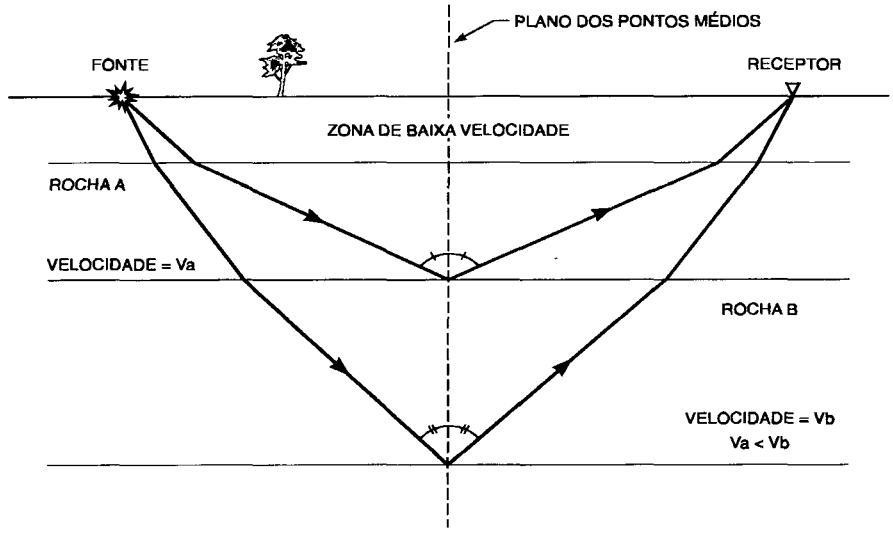


Figura 1-1 Reflexões em camadas geológicas (Adaptação de [Hatton (1986)]).

A aquisição de dados sísmicos é feita ao longo de uma determinada direção, chamada linha sísmica (Figura 1-2). Os dados coletados nessas linhas são tratados de forma a gerar visões de cortes nas camadas geológicas, denominados seções sísmicas (Figura 1-3).

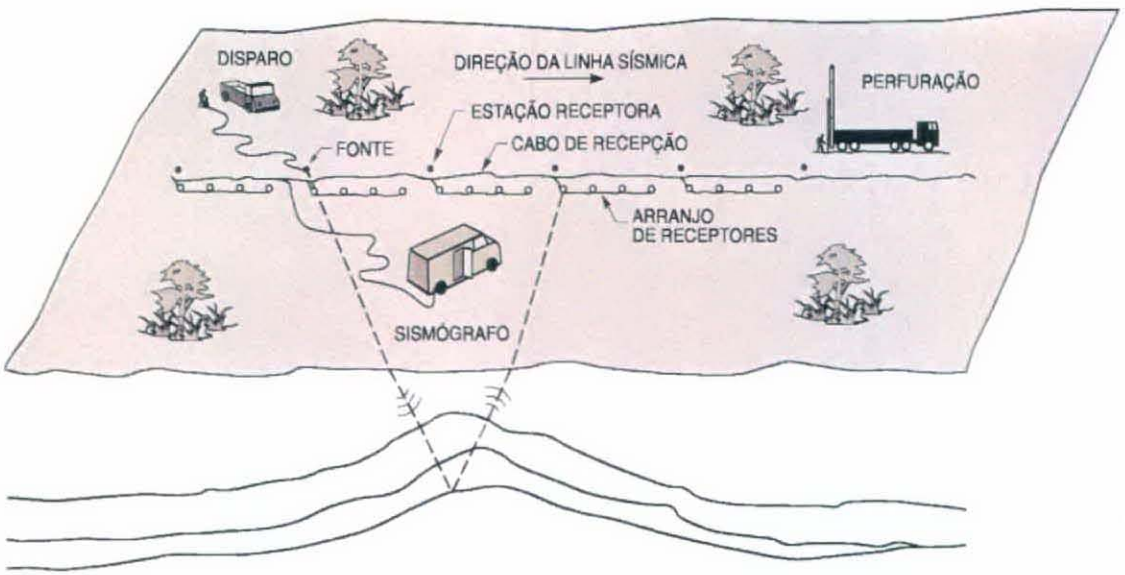


Figura 1-2 Disposição de fontes e receptores (Adaptação de [Robinson (1980)]).

Figura 1-3 Seção Sísmica.

Tradicionalmente, as linhas sísmicas eram dispostas paralelamente, em uma determinada região, a uma distância de aproximadamente 2 km entre linhas contíguas. A necessidade de estudar formações geológicas complexas fez com que o distanciamento entre linhas fosse bastante reduzido, resultando, em alguns casos, em aquisições cujo espaçamento entre linhas é da ordem de 20 metros. Este distanciamento permite que se obtenha uma visão volumétrica dos dados adquiridos em uma região.

Os valores escalares de amplitude dos sinais sísmicos obtidos são denominados amostras. Na visualização do dado sísmico 3D, as amostras são transformadas em primitivas coloridas (*pixels*, na visualização de fatias, ou *voxels*, na visualização volumétrica direta), para que as imagens 3D produzidas revelem as estruturas das formações geológicas, permitindo, ao intérprete, identificar potenciais reservatórios de óleo ou gás (Figura 1-4).

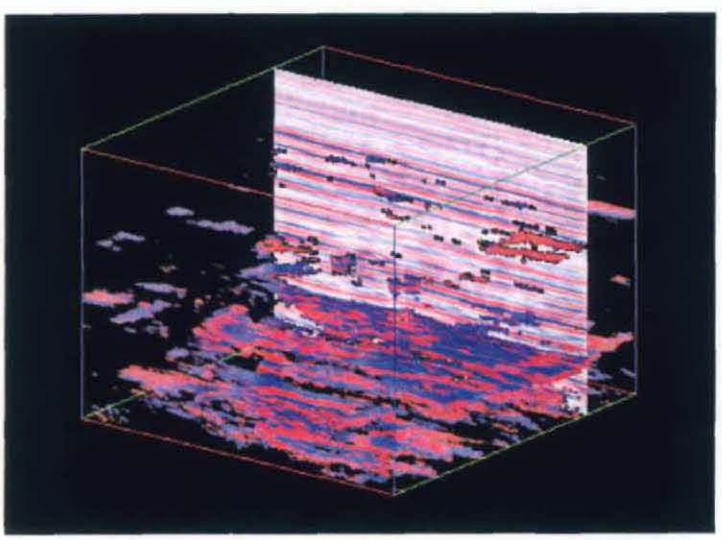


Figura 1-4 Visualização volumétrica e de uma fatia de um dado sísmico 3D.

O trabalho de interpretação requer recursos para realce de áreas de interesse, visualização de subconjuntos do dado original e extração de geometrias como planos de falhas e superfícies representando camadas geológicas, entre outros.

A arquitetura proposta neste trabalho define um conjunto de interfaces interativas para manipulação de representações de dados sísmicos volumétricos, baseado nessas necessidades.

1.3.3 Visão Geral do Trabalho

O capítulo 2 aborda a interação 3D, sua complexidade, o desenvolvimento de interfaces e técnicas, e apresenta soluções utilizadas em alguns ambientes tridimensionais. O capítulo 3 propõe uma arquitetura para construção de interfaces interativas, na forma de manipuladores, e um modelo de interação, a serem incorporados no *toolkit* para desenvolvimento de aplicacões 3D interativas na área de **E&P**. Nesse capítulo, uma visão geral do *toolkit* é apresentada. O capítulo 4 apresenta detalhes da implementação das classes definidas na arquitetura: suas funções, atributos e métodos. O protótipo com os manipuladores implementados é apresentado no capítulo 5. O capítulo 6 finaliza a dissertação, reunindo as conclusões e trabalhos futuros.

2. Interação 3D

Os programas aplicativos modernos de interface com o usuário são construídos utilizando-se ***widgets***, definidos como objetos com geometria e/ou comportamento usados para controlar a aplicação e seus objetos [Conner et al. (1992)].

Até recentemente, as aplicações 3D tiravam pouco partido da terceira dimensão, utilizando, predominantemente, *widgets* 2D. Em algumas aplicações, uma ou mais vistas 3D são apresentadas junto com um sistema hierárquico de menus, caixas de diálogos e *sliders* (potenciômetros virtuais). Outro exemplo de *widget* 2D, utilizado em ambientes de visualização modulares (**MVE**), é o editor de *network* (cadeia), uma interface de programação visual por manipulação direta para construir aplicações como cadeias executáveis de módulos (filtragem, mapeamento, geração de imagem, etc.). A interação direta com uma cena 3D englobava a visualização, seleção, translação e rotação, com a utilização de *widgets* 3D como: cursor 3D, translação gestual e esfera virtual.

Uma dificuldade para o projeto e implementação de interfaces 3D reside no fato da complexidade da interação 3D ser muito maior que a da interação 2D. Ela deve lidar com uma grande quantidade de primitivas, atributos e estilos de *rendering* (realização da imagem), com diferentes sistemas de coordenadas e com cálculos de projeções de visualização, determinação de visibilidade, iluminação e *shading* (sombreamento). Outra questão é que a interação 3D envolve mais graus de liberdade do que os especificados com dispositivos 2D de entrada, como o *mouse*.

Em sistemas de realidade virtual, a interação 3D é especialmente importante. Mas as dificuldades de entrada de dados e geração da imagem, nesses ambientes, têm levado a

pesquisa a se concentrar mais no desenvolvimento de novos dispositivos e técnicas para seu manuseio do que em técnicas de mais alto nível para interação 3D [Conner et al. (1992)].

Propostas de *toolkits* para interação 3D começaram a aparecer. Mas o fato dos diferentes ambientes de aplicações lidarem com diferentes tarefas de interação dificulta a criação de um *toolkit* genérico para criação de interfaces 3D, que teria que suportar uma variedade muito grande de técnicas de interação 3D.

Este capítulo aborda um pouco do que vem sendo feito, recentemente, em pesquisa e desenvolvimento de técnicas de interação 3D.

2.1 Integração entre a Aplicação e a Interface

As primeiras interfaces com o usuário foram projetadas por programadores de aplicação usando as mesmas ferramentas utilizadas para construir as aplicações. Mais recentemente, interfaces com o usuário têm sido construídas separadas da aplicação, com ferramentas genéricas de interface com o usuário. Essa separação produz interfaces consistentes e programas modulares, mas podem ser menos úteis do que seriam se fossem mais especializadas para a aplicação, especialmente para aplicações altamente interativas, como as aplicações 3D, que requerem uma comunicação intensa de dados entre elas e a interface.

Para aplicações 3D, a tendência atual é a interface e aplicação estarem fortemente integradas em um mesmo ambiente de desenvolvimento. Essa integração provê uma melhor comunicação entre elas, possibilitando melhor entrada interativa de dados e comunicação visual (*feedback*) semântica enquanto o usuário está interagindo.

2.2 Widgets *3D*

Widgets 3D encapsulam geometria 3D e comportamento para controlar ou exibir informações sobre objetos de aplicações 3D. Um *widget* 3D é visto como um objeto composto de uma ou mais objetos simples, chamados **primitivas**, ***handles*** (alças), ou ainda, ***draggers*** (manipuladores de arrasto), que possuem geometria e reagem a eventos de interação. As restrições e relacionamentos aplicados aos objetos simples definem o comportamento de um *widget* 3D.

Seja qual for a implementação utilizada na construção de um *widget* 3D, interagir com ele requer reconhecer qual de seus objetos simples foi selecionado para interação, interpretar seu movimento, de acordo com seu comportamento, manter os relacionamentos entre os objetos simples e alterar o objeto por ele controlado.

A implementação de técnicas de interação 3D por manipulação direta para dispositivos 2D, como o *mouse*, não é uma tarefa simples. Dispositivos de entrada com mais graus de liberdade podem aliviar o problema mas ainda apresentam problemas no uso (fadiga visual e muscular) e na resolução espacial [Herndon et al. (1994)] e, assim, soluções que façam uma ponte entre os dispositivos 2D e a interação 3D vêm sendo implementadas para aumentar a usabilidade das aplicações 3D.

As seções seguintes apresentam trabalhos realizados em centros de pesquisa de universidades e agências americanas e um produto comercial que propõem ferramentas para construção de interfaces tridimensionais.

2.3 Brown University Graphics Group

A **Universidade de Brown, USA**, vem trabalhando muito na especificação e projeto de *widgets* 3D, especialmente nas áreas de modelagem e animação. Um conjunto de critérios gerais para o projeto de *widgets* 3D é proposto [Snibbe et al. (1992)]:

- **Auto-revelação**

  Quando a geometria do *widget* indica seu comportamento.

- **Controle implícito de parâmetros versus controle explícito**

  Usualmente, um *widget* controla explicitamente um ou mais parâmetros de uma ação, através de *handles* (alças). Mas ele pode simplificar uma ação, reduzindo a complexidade da interface, controlando, implicitamente, outros parâmetros.

- **Graus de liberdade**

  Um ambiente 3D oferece vários graus de liberdade, que podem ajudar ou retardar uma operação particular. Um *widget* pode se tornar mais efetivo se os graus de liberdade desnecessários são removidos, através de restrições.

- **Uso pretendido**

  Widgets têm diferentes requisitos, de acordo com os propósitos de uso. Por exemplo, *widgets* para propósitos de engenharia devem ser mais precisos que *widgets* para propósitos artísticos.

O grupo propõe um *toolkit* interativo, orientado a objetos, para construção de interfaces e *widgets* tridimensionais, baseado na manipulação direta de primitivas visuais [Zeleznik et al. (1993)] [Stevens et al. (1994)].

As primitivas possuem uma representação geométrica e restrições que definem seu comportamento. Exemplos de **primitivas básicas** são: **ponto, vetor, plano** e **objeto gráfico** (cubos, esferas, CSGs, etc.). Um **objeto complexo** é construído a partir de associações (*links*) entre primitivas básicas e seu comportamento é definido pelas restrições estabelecidas para elas e entre elas. O usuário interage com uma primitiva básica, que trata o evento de interação de acordo com seu comportamento, atualiza sua geometria e os atributos necessários e faz com que todas as outras primitivas básicas a ela associadas se atualizem, de acordo com as restrições existentes entre elas.

Exemplos de *widgets* projetados pelo grupo são:

- *Racks*

  Widgets para aplicação de deformações em objetos 3D. Contêm *handles* para diferentes deformações, como *twisting* (torção), *tapering* (afilamento) e *bending* (flexão) [Snibbe et al. (1992)].

- Sombra interativa

  Projeção de um objeto 3D em um plano. O objeto 3D pode sofrer transformações no espaço pela utilização de uma variedade de técnicas de manipulação direta e sua sombra se movimenta de acordo com os resultados das transformações. Ou a sombra pode ser arrastada, resultando em um movimento do objeto. Este *widget* é útil tanto como uma informação adicional para a visualização do objeto 3D, quanto como uma ferramenta para restringir transformações sobre o objeto para um plano [Herndon et al. (1992)].

- *Widgets* para visualização interativa de campos escalares em ambientes virtuais

  Um trabalho em conjunto com o NASA Ames Research Group, no projeto de *widgets* para manipulação de isosuperfícies e planos de corte [Meyer et al. (1993)].

- *Widgets* para operações de modelagem com manipulação livre

  Um trabalho em colaboração com a Universidade de Utah, USA, no projeto de *widgets* para operações de *blending* (transição suave entre duas superfícies), *sweeping* (geração de curva, superfície ou volume como resultado do movimento de um objeto geométrico, como ponto, curva, superfície ou volume) e *warping* (deformação de uma superfície para criar saliências suaves) [Grimm et al. (1995)].

2.4 Open Inventor

Open Inventor, *software* padrão da *Silicon Graphics, Inc.*, é um *toolkit* orientado a objetos, independente de sistemas de janelas, utilizado para criar aplicações gráficas 3D interativas [Strauss et al. (1992)] [Wernecke (1994)]. Ele provê uma **base de dados de objetos** (nós) para compor cenas 3D, uma **biblioteca de componentes**, para sua utilização com sistemas de janelas específicos, e uma **API** para manipulação dos objetos e componentes. Provê, também, um mecanismo para modificar e estender o *toolkit* com novos objetos e operações, para atender novas necessidades.

Cenas 3D são descritas em um **grafo** (***scene graph***) composto de nós representando informações como *shapes* (formas), transformações geométricas, materiais, câmera e luzes. Nós podem ser associados em *node kits* (grupos de nós), com a finalidade de criar objetos de mais alto nível que encapsulem comportamentos específicos da aplicação.

A interação com os objetos da cena é proporcionada por nós especiais, como **manipuladores** e ***draggers***. Os manipuladores são subclasses dos nós editáveis, como os que representam tranformações geométricas e luzes, e substituem esses nós no grafo durante as interações. *Draggers* são objetos com geometria e comportamento específicos, que reagem a eventos de interação e que são utilizados pelos manipuladores para alterar seus atributos editáveis. No fim de uma interação, o grafo é restaurado com o nó original, que tem seus atributos atualizados com os valores correntes dos atributos do manipulador que o substituiu.

Um modelo semelhante é também utilizado na implementação dos *sensors*, em **VRML** 2.0, cuja arquitetura é fortemente baseada no *Open Inventor* [Hartman et al. (1996)].

Open Inventor é uma biblioteca de propósito geral. Estão disponibilizados manipuladores para editar nós como os que representam luzes e transformações geométricas. A utilização dessa biblioteca em sistemas de visualização científica requer a implementação de subclasses, para criar nós que representem as diversas técnicas de visualização, e de manipuladores para editá-las.

2.5 Interação 3D em Ambientes de Visualização Científica

Aplicações de visualização científica requerem técnicas de navegação e orientação (ações que alterem as representações dos dados). Lidar com conjuntos de dados muito grandes é outra necessidade.

Campos escalares 3D são exemplos de dados visualizados em ambientes de visualização científica. Um campo escalar 3D compreende uma função f definida sobre um espaço de coordenadas 3D. Uma representação típica para ele é a de pontos amostrados, em intervalos discretos, formando grades 3D.

Exemplos de técnicas de visualização para este tipo de dados são:

- **Planos de Corte** (fatias)

  Um plano de corte que atravessa o dado e define um mapeamento das amostras na superfície do corte. É uma técnica bem direta de inspeção de dados, onde um subconjunto do dado é visualizado e a exploração é feita através da movimentação do plano dentro do espaço do dado.

- **Isosuperfície**

  Uma superfície contínua que representa um valor escalar constante dentro de um campo escalar 3D, possibilitando uma visão global de um dado valor no campo. O mais comum é a isosuperfície ser gerada como uma superfície poliedral.

- ***Rendering* Direto do Volume**

  Técnica de *rendering* total do volume (campo escalar 3D), sem a conversão para representações intermediárias, como fatias e superfícies. Ela permite visualizar o interior do conjunto de dados, pela utilização de valores de opacidade nas funções de transferência, que transformam os valores dos dados para os valores dos *pixels*. É uma técnica de visualização cujos cálculos e *rendering* consomem muito tempo, dificultando a manipulação interativa.

Mesmo as técnicas de visualização de fatias e de isosuperfícies podem se tornar muito pesadas, quando aplicadas a conjuntos de dados grandes, comuns à maioria dos problemas científicos. O grande consumo de memória e tempo para os cálculos e a visualização restringe a interatividade da manipulação das representações. Em alguns casos, o recurso de não exibir os dados durante uma manipulação pode ser utilizado. Mas, para garantir um uso

mais efetivo das técnicas de interação, algum tipo de visualização deve ser mantido como *feedback*, utilizando-se meios para aumentar a velocidade do *rendering*, como **aproximações** e **multiresolução**.

Ambientes de visualização modulares, como o **AVS** [Upson et al. (1989)], baseados em paradigmas de *data-flow* e programação visual, são ferramentas muito utilizadas para visualização e análise de dados científicos. Mas o alto custo computacional associado aos sistemas *data-flow* resulta, em geral, em limitações na interatividade e na manipulação direta de dados. *Sliders* e entradas textuais são exemplos de ferramentas de manipulação utilizadas.

A utilização do *Open Inventor* para desenvolver aplicações 3D em ambientes de visualização científica é dificultada por algumas características que ele apresenta. O *Open Inventor* não permite acesso direto ao *framebuffer* e opera em modo *retained* (modo no qual uma imagem não existe somente como um desenho na tela, mas sim como um registro de primitivas e outras informações relacionadas, o que permite edições subsequentes e atualização da exibição sem sobrecarregar a aplicação), utilizando a estrutura de dados hierárquica (*scene graph*) para armazenar todas as informações necessárias ao *rendering* dos objetos da cena, como forma, tamanho, cor e localização espacial. Funcionar exclusivamente em modo *retained* limita a utilização do *Open Inventor* em aplicações envolvendo grande volume de dados. Não prover acesso direto ao *framebuffer* limita a implementação de métodos eficientes de *rendering* de volume. Uma abordagem possível de ser adotada é a de combinar comandos *OpenGL* com objetos e componentes *Inventor* pré-construídos.

Na Petrobras, a utilização do *Open Inventor* também é dificultada pelo custo da sua instalação, pois é um produto que tem que ser adquirido para os vários equipamentos e ambientes computacionais que se queira utilizar.

3. Arquitetura para Manipulação 3D

O *toolkit* para construção de aplicações 3D interativas de visualização científica que está sendo desenvolvido na Petrobras se compõe de uma biblioteca extensível de classes e métodos que provêem um conjunto de serviços para o domínio de aplicações de visualização e interpretação na área de exploração e produção de óleo (E&P).

O *toolkit* se baseia em idéias do *Open Inventor* e algumas de suas classes, como câmera, cena, luz e visualizador têm correspondência direta com classes dessa biblioteca. Classes como contexto gráfico, objeto gráfico, projeto e visualização, próprias do domínio de aplicações a que o *toolkit* se destina, foram introduzidas.

Este capítulo apresenta a proposta de extensão do *toolkit* com interfaces interativas para manipulação 3D. Inicialmente, é dada uma visão geral do *toolkit*, através da descrição de algumas classes base[3] e seus relacionamentos.

Neste trabalho, é utilizada uma notação para definição dos diagramas de classes de um projeto orientado a objetos, para mostrar as classes e seus relacionamentos em uma visão lógica do sistema [Booch (1994)], cuja simbologia está descrita na Figura 3-1.

---

[3]Uma classe base é a classe mais generalizada em uma estrutura de classes [Booch (1994)]. As aplicações têm, em geral, várias classes base.

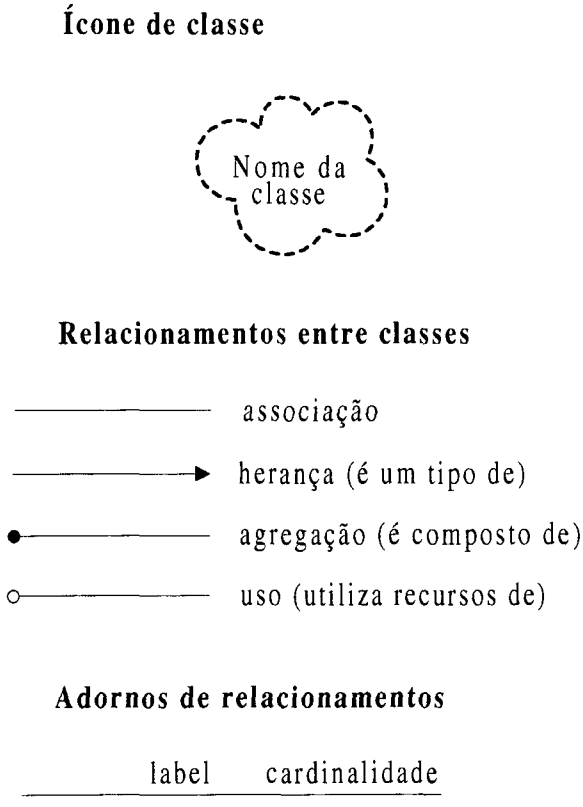


Figura 3-1 Simbologia do diagrama de classes.

3.1 Toolkit *para Desenvolvimento de Aplicações Interativas de Visualização Científica na Área de E&P*

A Figura 3-2 apresenta um diagrama de classes do *toolkit* existente.

A classe ***Graphical Object*** define o conjunto de objetos que representam os objetos matemáticos com os quais as aplicações de E&P lidam. São exemplos de objetos gráficos: pontos esparsos, linhas poligonais (para representação de poços), *grids* 2D (superfícies funcionais representando horizontes geológicos) e *grids* 3D (volumes com propriedades sísmicas ou de reservatórios).

A classe ***Visualization*** define objetos que representam formas de *rendering* dos objetos gráficos. São as visualizações que aparecem na tela e com elas o usuário quer

interagir. Um exemplo de visualização para superfícies funcionais é a visualização de contorno. Exemplos de visualizações para volumes são: a visualização de fatias, a visualização de isosuperfícies ou o *rendering* volumétrico direto.

Um conjunto de visualizações compõe uma cena 3D, objeto da classe ***Scene***. Para ser exibida, uma cena é associada a um visualizador.

Um visualizador, objeto da classe ***Viewer***, é um *widget* de um sistema de janelas que contém uma área de desenho, para o *rendering* da cena, e oferece uma interface com o usuário para controle da câmera, para definições como: estilo de desenho (sólido ou *wireframe*), tipo de *buffering* (*single* ou *double*) e modo de operação, e para execução de operações da aplicação.

A classe ***Editor*** define objetos utilizados para editar objetos de classes auxiliares, como ***Color Scale*** e ***Material Scale***.

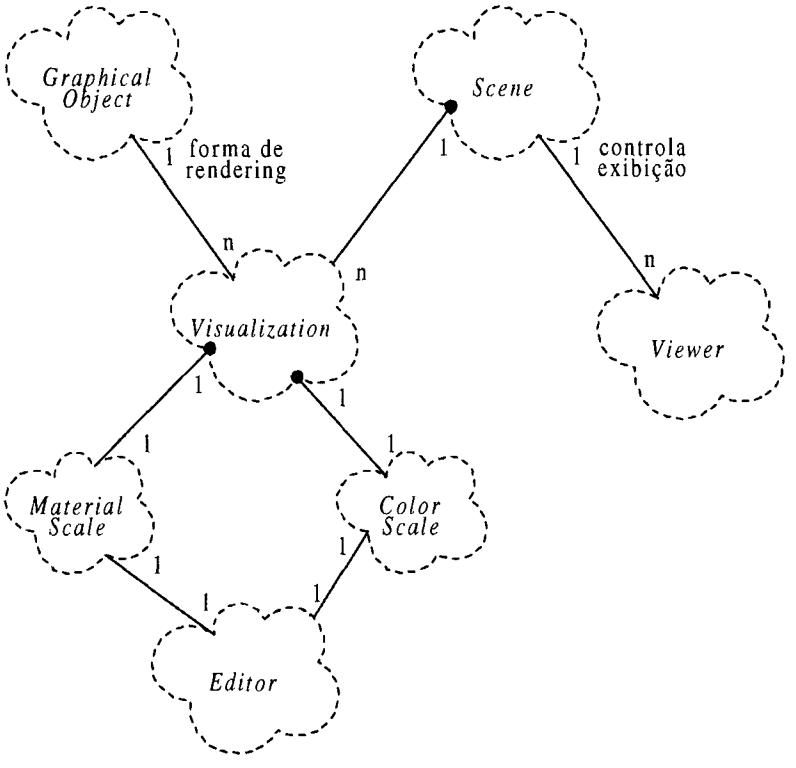


Figura 3-2 Diagrama de classes com algumas classes base do *toolkit*.

3.2 Extensão do Toolkit *para Manipulação 3D*

Este trabalho estende a funcionalidade do *toolkit* para desenvolvimento de aplicações interativas de visualização científica com operações de manipulação direta sobre as visualizações de uma cena através da utilização de *widgets* 3D, aqui chamados de manipuladores.

A arquitetura proposta acompanha o paradigma de orientação a objetos utilizado no desenvolvimento do *toolkit*. Ela define um conjunto de classes, métodos e relacionamentos, para implementação de um modelo de tratamento de eventos cuja idéia essencial é encapsular a lógica de processamento de eventos de manipulação direta na implementação dos manipuladores.

Essa é a idéia do modelo adotado pelo *Open Inventor*, de forma mais sofisticada, e também por algumas outras bibliotecas. Este trabalho identificou as classes essenciais para implementar o modelo de uma forma simples e extensível, para ser utilizado em aplicações baseadas em bibliotecas gráficas convencionais, como o *OpenGL*.

A arquitetura será apresentada através da descrição das classes base e objetos incorporados ao *toolkit* e do modelo de interação implementado. O diagrama de classes apresentado na Figura 3-3 ilustra as classes base e relacionamentos introduzidos no *toolkit* para as operações de manipulação 3D.

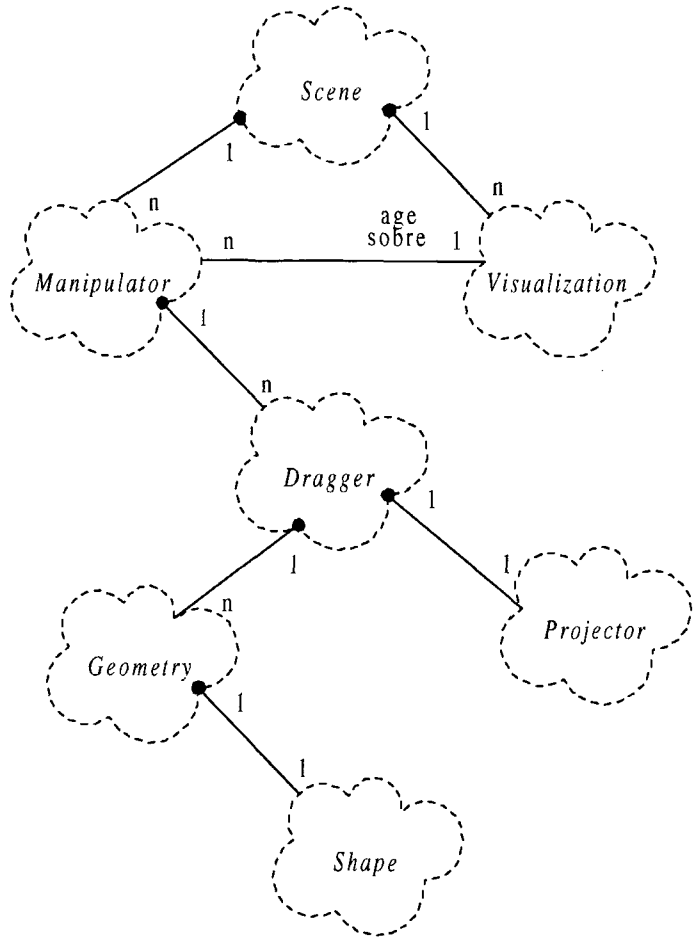


Figura 3-3 Diagrama de classes para manipulação 3D.

3.2.1 Manipuladores

A classe ***Manipulator*** define os objetos manipuladores. Um manipulador é um *widget* 3D que se apresenta como um objeto da cena, associado a uma visualização, e que pode ser editado, resultando em alterações na visualização ou retornando informações sobre ela.

O usuário utiliza o *mouse* para interagir com o manipulador que reage a eventos de botão pressionado / arrastado / solto. Esses eventos indicam, respectivamente, o início de uma interação, a interação propriamente dita e o fim da interação.

A visualização associa ao manipulador funções *callbacks* (funções dos clientes do manipulador), para que ele se comunique com ela quando reagir aos eventos de interação. A

função *callback* de início (*start callback*) estabelece uma comunicação entre o manipulador e a visualização quando ocorrem eventos de botão pressionado. A função *callback* de movimento (*motion callback*) estabelece uma comunicação quando ocorrem eventos de botão arrastado. E a função *callback* de fim (*finish callback*) estabelece uma comunicação quando ocorrem eventos de botão solto.

Um exemplo de manipulador, utilizado para movimentar uma visualização do tipo fatia de um volume, é apresentado de forma esquematizada na Figura 3-4 e será referenciado pelas seções seguintes para ilustrar a composição dos manipuladores.

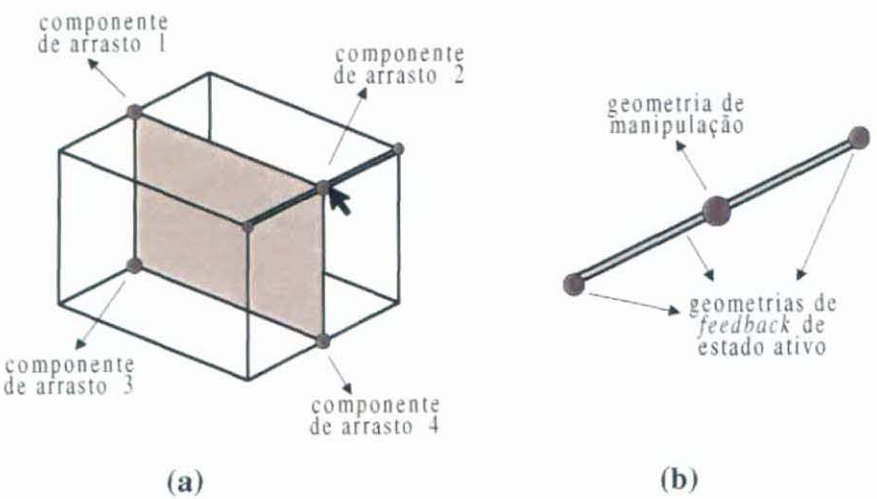


Figura 3-4 Esquema do manipulador de movimentação de fatia do volume (a) e detalhe da geometria do componente de arrasto ativo (b).

3.2.2 Componentes de Arrasto

A classe ***Dragger*** define os componentes de arrasto utilizados pelos manipuladores para compor sua geometria e suas edições complexas. São os componentes de arrasto que possuem geometria, que reagem diretamente aos eventos de interação com o usuário e que realizam edições simples como escala, rotação e translação.

Para compor transformações de translação, rotação e escala, foram identificados, no projeto de manipuladores, os seguintes tipos de movimento para os componentes de arrasto realizarem: translação sobre uma linha; translação sobre um plano; translação em um volume; e rotação sobre a superfície de um cilindro.

Utilizar um dispositivo 2D, como o *mouse*, requer a implementação de técnicas de interação menos naturais do que as permitidas por dispositivos 3D. Translações sobre linhas e planos e rotações sobre cilindros são movimentos que podem ser mapeados diretamente a partir de movimentos do *mouse*, como será visto na seção seguinte. Já translações no espaço envolvem mais graus de liberdade do que os que podem ser especificados com o *mouse* e têm que ser decompostos. Procura-se utilizar geometrias e dar *feedback* para indicar a execução desses movimentos como uma composição de movimentos diretamente mapeados.

Funções *callbacks* são associadas aos componentes de arrasto para que eles possam reagir aos eventos. A função *callback* de início (*start callback*) trata eventos de botão pressionado, preparando o componente de arrasto para uma interação. A função *callback* de movimento (*motion callback*) trata eventos de botão arrastado, interpretando o movimento do componente de arrasto. E a função *callback* de fim (*finish callback*) trata eventos de botão solto, encerrando a manipulação do componente de arrasto. Esses três tipos de função *callback* tratam a interação localmente. Existe ainda um quarto tipo de função *callback* que é associada a um componente de arrasto para que ele se comunique com seu manipulador passando informações sobre seu movimento. É a chamada função *callback* de valor alterado (*valuechanged callback*).

A Figura 3-4(a) ilustra o emprego de quatro componentes de arrasto na composição do manipulador para movimentação de fatia. Como o movimento da fatia é executado na

direção perpendicular ao seu plano, o movimento a ser realizado pelos componentes de arrasto é o de translação sobre uma linha.

3.2.3 Projetores

A classe ***Projector*** define os objetos que são utilizados pelos componentes de arrasto para mapear os movimentos do *mouse* sobre eles para os movimentos correspondentes em 3D.

O mecanismo para o mapeamento do movimento em 2D para 3D consiste em projetar a posição 2D normalizada do *mouse* como uma linha que parte do olho e atravessa a cena e calcular a interseção dessa linha com alguma forma geométrica (linha, plano, esfera, cilindro,...). Sucessivos mapeamentos da posição do *mouse* enquanto ele se desloca definem os movimentos do componente de arrasto em 3D.

Os componentes de arrasto que compõem o manipulador de movimentação de fatia utilizam projetores de linha para mapear seus movimentos.

3.2.4 Geometria

Os componentes de arrasto podem se apresentar com representações visuais diferentes em cada manipulador. Cabe aos manipuladores definir a aparência desejada seus componentes de arrasto.

A classe ***Geometry*** define os objetos que contêm as especificações geométricas dos componentes de arasto: posicionamento na cena (através de tranformações de translação e rotação), estilo de *rendering* (sólido ou *wireframe*), material e forma.

São três os tipos de geometria que podem ser definidos: a geometria de manipulação, a geometria de *feedback* e a geometria de *feedback* de estado ativo. Uma única geometria de manipulação é definida e é ela que é desenhada na operação de *picking* (seleção). Os outros dois tipos de geometria são utilizados para ajudar os componentes de arrasto a expressar sua função, podem ser definidos em quantidade qualquer e não são considerados para *picking*. A geometria de *feedback* é exibida junto com a geometria de manipulação e a geometria de *feedback* de estado ativo é exibida somente enquanto o componente de arrasto está ativo, ou seja, está sendo manipulado.

Na Figura 3-4(a), os três componentes de arrasto no estado inativo têm somente a geometria de manipulação exibida. O componente de arrasto selecionado para manipulação (componente de arrasto 2) tem exibidas, além da geometria de manipulação, três geometrias de *feedback* de estado ativo, uma para indicar a direção do movimento e duas para indicar os limites do movimento (Figura 3-4(b)).

3.2.5 Forma

A classe ***Shape*** define os modelos geométricos tridimensionais utilizados na definição da forma geométrica dos componentes de arrasto. Esferas, cilindros, cubos e cones são exemplos de modelos geométricos que ficam definidos a partir da especificação de parâmetros como raio, altura e largura. Modelos geométricos também podem ser definidos a partir da especificação de seus vértices.

A Figura 3-5 ilustra as formas geométricas definidas para as geometrias de manipulação e de *feedback* de um componente de arrasto do manipulador para movimentação de fatia.

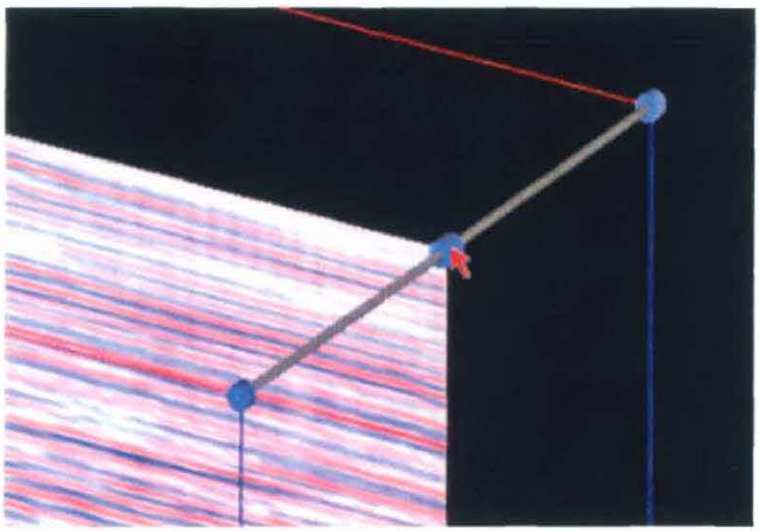


Figura 3-5 Geometria de um componente de arrasto do manipulador para movimentação de fatia.

3.2.6 Modelo de Interação

O modelo de interação será apresentado através da descrição dos objetos e relacionamentos envolvidos na manipulação 3D e da apresentação de diagramas de objetos.

Diagramas de objetos também fazem parte da notação de um projeto orientado a objetos e são utilizados para mostrar a existência de objetos e seus relacionamentos em uma visão lógica do sistema. Cada diagrama de objetos representa um *snapshot* em tempo de um fluxo de eventos sobre uma determinada configuração de objetos [Booch (1994)].

O diagrama de objetos utiliza os ícones mostrados na Figura 3-6.

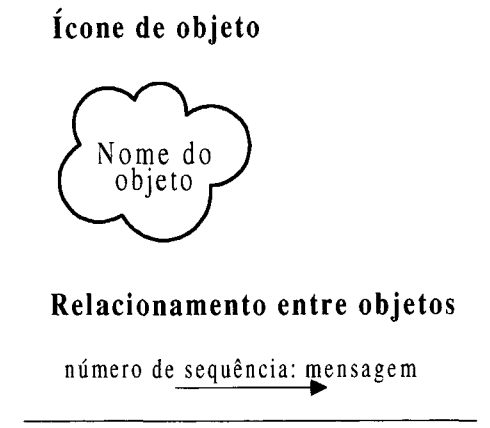


Figura 3-6 Simbologia do diagrama de objetos.

Um visualizador pode trabalhar em dois modos de operação. No modo de ***viewing***, o visualizador interpreta os eventos sobre a área de desenho para controlar a câmera associada. O outro modo que um visualizador pode estar é o modo de ***picking***, no qual os eventos que ocorrem sobre a área de desenho são interpretados como eventos de manipulação sobre algum objeto da cena. Nesse modo, o visualizador passa os eventos diretamente para a cena.

Ao receber um evento de botão do *mouse* pressionado, a cena utiliza o mecanismo de *picking* da biblioteca gráfica *OpenGL* para verificar se ele ocorreu sobre algum manipulador. Para que esse mecanismo possa ser utilizado, manipuladores e componentes de arrasto possuem um identificador de *picking*. Utilizar o mecanismo de *picking* consiste em redesenhar as geometrias de manipulação dos componentes de arrasto dos manipuladores da cena, no modo *selection* do *OpenGL*, restringindo o desenho à uma região próxima ao cursor. Nesse modo, as informações de desenho não são enviadas ao *framebuffer*, mas retornadas para a aplicação, na forma de registros contendo uma hierarquia de identificadores das primitivas que interceptam a região de desenho e os valores de z (profundidade no sistema de coordenadas da tela) das interseções. Analisando essas informações, a cena reconhece o

manipulador e o componente de arrasto sobre os quais o evento de início de interação ocorreu.

A cena guarda como um atributo seu o manipulador selecionado e passa para ele o evento e o identificador do componente de arrasto. O manipulador realiza as operações necessárias ao início de uma interação, como guardar como um atributo seu o componente de arrasto selecionado, e encaminha o evento para ele. O componente de arrasto se prepara para o início da interação executando operações como alterar sua geometria para indicar que está ativo e definir a posição inicial do movimento. Se necessário, o manipulador estabelece uma comunicação com a visualização ao qual está associado.

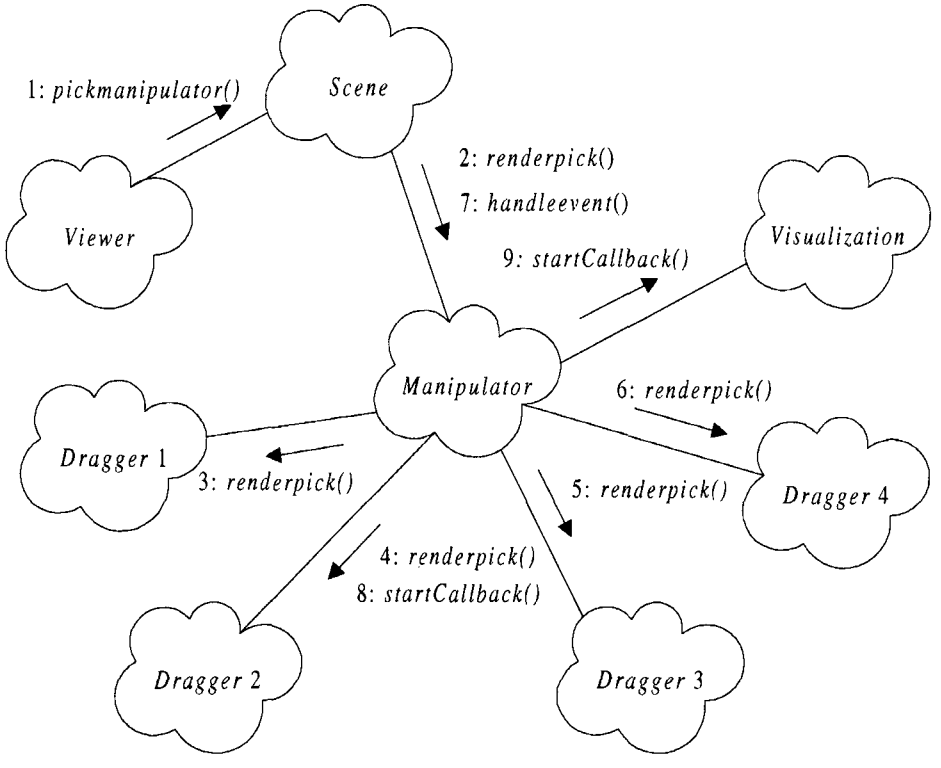


Figura 3-7 Diagrama de objetos que ilustra o tratamento do evento de início da interação.

O diagrama de objetos apresentado na Figura 3-7 mostra o fluxo de mensagens entre os objetos envolvidos no início da interação com o manipulador de movimentação de fatia. A

ordem da troca de mensagens segue a numeração mostrada na figura.

O mecanismo de *picking* só é necessário em um evento de botão pressionado. Os eventos de botão arrastado e de botão solto são passados diretamente para o manipulador que a cena identificou no início da interação e o manipulador, por sua vez, os passa diretamente para o componente de arrasto selecionado no início da interação.

Para eventos de botão arrastado, o componente de arrasto utiliza seu projetor para interpretar o movimento do *mouse* e informa o manipulador sobre seu movimento. O manipulador valida o movimento e se atualiza, informando aos seus componentes de arrasto se e como eles devem se atualizar e, se necessário, interage com a visualização ao qual está associado.

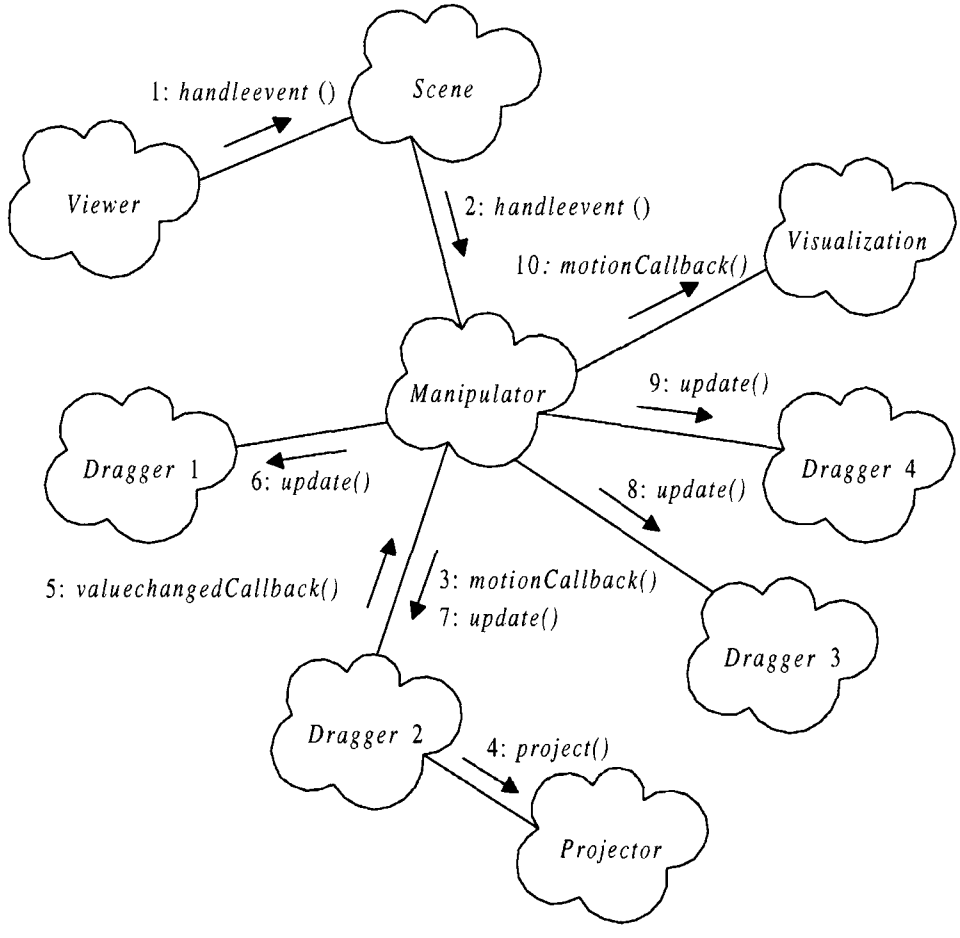


Figura 3-8 Diagrama de objetos que ilustra o tratamento de eventos de arrasto.

O fluxo de mensagens entre os objetos envolvidos em um evento de botão arrastado, para o manipulador de movimentação de fatia, é ilustrado no diagrama de objetos apresentado na Figura 3-8.

Um evento de botão do *mouse* solto encerra a interação. O componente de arrasto executa as operações necessárias ao fim da interação, como restaurar seu estado inativo, e o manipulador encerra a interação com a visualização.

A Figura 3-9 apresenta o diagrama de objetos com o fluxo de mensagens entre os objetos envolvidos na finalização da interação com o manipulador de movimentação de fatia.

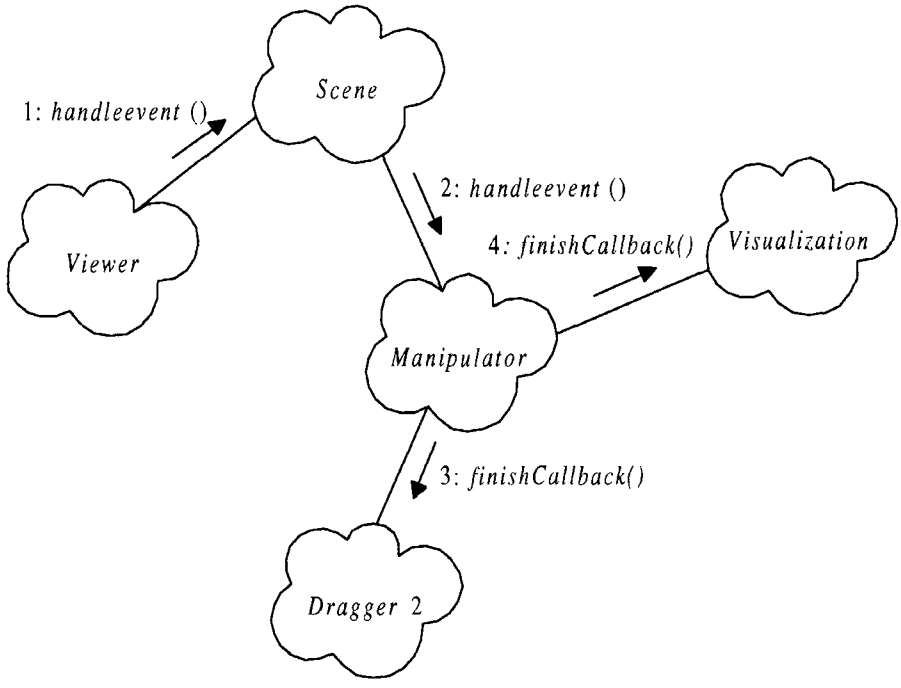


Figura 3-9 Diagrama de objetos que ilustra o tratamento do evento de finalização da interação.

4. Detalhamento da Arquitetura

No capítulo anterior, uma visão geral da arquitetura para manipulação 3D foi apresentada através da apresentação das classes base incorporadas ao *toolkit* para desenvolvimento de aplicações 3D interativas de visualização científica.

As classes base *Manipulator*, *Dragger*, *Projector* e *Shape* representam as categorias mais generalizadas na arquitetura para manipulação 3D proposta. As diferentes manipulações requeridas pelos vários tipos de visualizações e os diferentes movimentos e aparências requeridos para os componentes de arrasto resultam em especializações da estrutura e do comportamento dessas classes.

O objetivo, neste capítulo, é apresentar a implementação da arquitetura através da especificação mais completa das classes base *Shape*, *Geometry*, *Projector*, *Dragger* e *Manipulator* e das subclasses que as especializam. A especificação de cada classe abrange descrições de sua função e tabelas de atributos e métodos.

A biblioteca gráfica *OpenGL* já foi referenciada na apresentação do modelo de interação e volta a ser referenciada neste capítulo, pois é utilizada no *rendering* dos modelos tridimensionais.

4.1 *Classe* Shape

A classe *Shape* é a classe base que define os modelos tridimensionais utilizados para compor a forma geométrica dos componentes de arrasto.

Os modelos tridimensionais são definidos a partir da especificação de parâmetros ou de vértices e do estilo de *rendering*, sólido ou *wireframe*. Modelos definidos a partir de

parâmetros são desenhados centralizados na origem, sendo necessário definir transformações de rotação e translação para que eles sejam desenhados na posição desejada.

Para otimizar sucessivos *renderings*, os modelos são criados em *display lists* do *OpenGL* (grupos de comandos *OpenGL*, armazenados para posterior execução).

Dependentes dos modelos tridimensionais, os métodos da classe *Shape* são virtuais, definidos nas subclasses de especialização.

Tabela 4-1 Atributos da classe base *Shape*.

version	Versão para desenho: sólido ou *wireframe*.

Tabela 4-2 Métodos da classe base *Shape*.

getBoundingBox	Método virtual que calcula a caixa envolvente do modelo tridimensional. Definido pelas subclasses.
render	Método virtual que desenha o modelo tridimensional na versão corrente (sólido ou *wireframe*). Definido pelas subclasses.
setDimensions	Método virtual que estabelece as dimensões do modelo. Definido pelas subclasses.

Os modelos tridimensionais, identificados para implementação no projeto dos manipuladores e no desenvolvimento do protótipo, determinaram especializações para a classe *Shape* e são apresentados a seguir.

4.1.1 Classe *BoxShape*

A classe *BoxShape* é a subclasse de especialização da classe *Shape* que define o modelo tridimensional caixa.

Tabela 4-3 Atributos da classe *BoxShape*.

depth	Comprimento da caixa.
height	Altura da caixa.
width	Largura da caixa.

4.1.2 Classe *CubeShape*

A classe *CubeShape* é a subclasse de especialização da classe *Shape* que define o modelo tridimensional cubo.

Tabela 4-4 Atributos da classe *CubeShape*.

size	Tamanho das arestas do cubo.

4.1.3 Classe *CylinderShape*

A classe *CylinderShape* é a subclasse de especialização da classe *Shape* que define o modelo tridimensional cilindro. O cilindro é desenhado na origem, como todos os *shapes* definidos por parâmetros, e seu eixo corresponde ao eixo Y.

Tabela 4-5 Atributos da classe *CylinderShape*.

height	Altura do cilindro.
radius	Raio do cilindro.

4.1.4 Classe RectangleShape

A classe *RectangleShape* é a subclasse de especialização da classe *Shape* que define o modelo tridimensional retângulo. O retângulo é desenhado na origem, como todos os *shapes* definidos por parâmetros, sobre o plano perpendicular ao eixo Z.

Tabela 4-6 Atributos da classe *RectangleShape*.

height	Altura do retângulo.
width	Largura do retângulo.

4.1.5 Classe *SphereShape*

A classe *SphereShape* é a subclasse de especialização da classe *Shape* que define o modelo tridimensional esfera.

Tabela 4-7 Atributos da classe *SphereShape*.

radius	Raio da esfera.

4.1.6 Classe VertexBoxShape

A classe *VertexBoxShape* é a subclasse de especialização da classe *Shape* que define o modelo tridimensional de uma caixa, a partir da especificação de seus vértices.

Tabela 4-8 Atributos da classe *VertexBoxShape*.

vertices	Vértices da caixa.

4.2 Classe Geometry

Esta classe base define as especificações de posicionamento na cena, forma e material das geometrias de manipulação e de *feedback* dos componentes de arrasto.

Sua função, na operação de *rendering* de um componente de arrasto, é atualizar a matriz de modelagem e visualização com as transformações de rotação e translação, para posicionar a geometria, especificar as propriedades do material, para a versão sólida, ou a cor base, para a versão *wireframe*, e chamar o método de *rendering* do modelo tridimensional.

Tabela 4-9 Atributos da classe base *Geometry*.

rotation	Transformação de rotação para posicionamento da geometria na cena.
translation	Transformação de translação para posicionamento da geometria na cena.
ambient	Cor ambiente para a superfície do modelo tridimensional.
diffuse	Cor difusa para a superfície do modelo tridimensional, para a versão sólida, e cor base do modelo tridimensional, para a versão *wireframe*.
specular	Cor especular para a superfície do modelo tridimensional.
shininess	Coeficiente de especularidade.
shape	Identificador do modelo tridimensional definido para a geometria.

Tabela 4-10 Métodos da classe base *Geometry*.

fit	Estabelece as tranformações de rotação e translação, para posicionamento da geometria, e o modelo tridimensional.
render	Desenha a geometria a partir das especificações de posicionamento, material e modelo tridimensional.
setColor	Estabelece a cor base da geometria, para a versão *wireframe*.
setMaterial	Estabelece as propriedades do material da geometria, para a versão sólida.
setRotation	Estabelece a transformação de rotação da geometria.
setTranslation	Estabelece a transformação de translação da geometria.
updateRotation	Incorpora uma transformação de rotação à transformação de rotação corrente da geometria.
updateTranslation	Incorpora uma transformação de translação à transformação de translação corrente da geometria.

4.3 Classe Projector

A classe base *Projector* define os projetores a serem utilizados no mapeamento dos movimentos dos componentes de arrasto.

O mapeamento dos movimentos é feito em coordenadas normalizadas. No início de uma manipulação, a matriz de modelagem e visualização e a matriz de projeção correntes são armazenadas no projetor, assim como suas inversas. As matrizes são utilizadas para transformar os pontos que definem os projetores, de coordenadas do mundo para coordenadas normalizadas, e suas inversas, para transformar o resultado do mapeamento, de coordenadas normalizadas para coordenadas do mundo. A posição (x_d, y_d) do *mouse* é transformada para coordenadas normalizadas, de acordo com a *viewport* corrente, atribuindo-se o valor 0 para a coordenada z. A posição 3D correspondente é a interseção (ou o ponto mais próximo) entre a linha que passa por essa posição, com direção [0, 0, 1], e a função geométrica do projetor.

Tabela 4-11 Atributos da classe base *Projector*.

modelview	Matriz de modelagem e visualização corrente.
projection	Matriz de projeção corrente.
inverse_modelview	Inversa da matriz de modelagem e visualização corrente.
inverse_projection	Inversa da matriz de projeção corrente.

Tabela 4-12 Métodos classe base *Projector*.

normalize	Método virtual que calcula as coordenadas normalizadas do projetor, de acordo com o volume de visualização corrente. Definido pelas subclasses.
project	Método virtual que projeta a posição do *mouse* normalizada sobre uma função geométrica para definir uma posição 3D. Definido pelas subclasses.
setMatrices	Armazena a matriz de modelagem e visualização e a matriz de projeção correntes e calcula suas inversas, para serem utilizadas nas projeções das posições do *mouse*.

4.3.1 Classe *ProjectCylinder*

A classe *ProjectCylinder* é a subclasse de especialização da classe *Projector* que mapeia posições de um componente de arrasto que se movimenta sobre um cilindro, definindo rotações em torno do eixo do cilindro.

O cilindro de projeção é especificado pela definição do raio e da linha correspondente ao seu eixo. O mapeamento da posição do *mouse* requer calcular as interseções entre a linha que passa pela posição do *mouse* normalizada, com direção [0, 0, 1], e o cilindro de projeção normalizado (Figura 4-1).

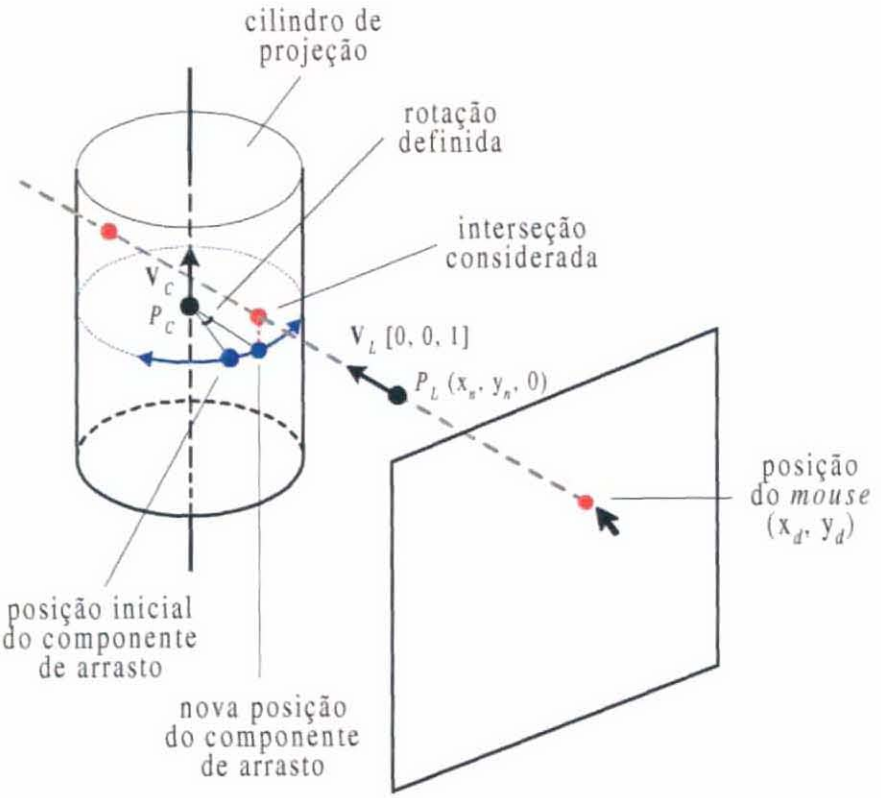


Figura 4-1 Mapeamento da posição do *mouse* sobre o cilindro de projeção e definição da rotação e da nova posição do componente de arrasto.

O algoritmo descrito em [Cychosz (1990)] é utilizado no cálculo das interseções. A linha é definida pelo ponto base P_L (posição do *mouse* normalizada) e o vetor direção unitário $\mathbf{V}_L$. O cilindro de projeção, pelo raio r, pelo vetor unitário $\mathbf{V}_C$ que define seu eixo, e por um ponto base P_C, localizado no eixo do cilindro. Se a linha intercepta o cilindro, os pontos de interseção são calculados.

O projetor de cilindro retorna o ponto 3D mapeado sobre o cilindro. Opcionalmente, ele pode devolver a rotação em torno de seu eixo definida por uma posição base (em geral, a posição corrente do componente de arrasto) e o ponto mapeado. Para que o projetor defina qual dos pontos de interseção será retornado ou utilizado para gerar uma rotação, uma orientação para o cilindro de projeção (em direção ao olho ou ao seu eixo Z) e a parte do

cilindro a ser utilizada na projeção (metade da frente ou de trás, em relação à sua orientação) são especificadas. Para calcular a rotação, a interseção considerada é projetada no plano perpendicular ao eixo do cilindro que passa pela posição corrente do componente de arrasto.

Tabela 4-13 Atributos da classe *ProjectCylinder*.

axis_line	Linha que define o eixo do cilindro de projeção.
radius	Raio do cilindro de projeção.
norm_unit_axis	Vetor direção unitário do eixo do cilindro de projeção normalizado.
norm_radius	Raio do cilindro de projeção normalizado.
norm_base_point	Ponto localizado no eixo do cilindro de projeção normalizado.
orient	Orientação do cilindro de projeção.
intersection_part	Parte do cilindro a ser interceptado na projeção (metade da frente ou de trás, em relação à sua orientação).

Tabela 4-14 Métodos da classe *ProjectCylinder*.

getCylinder	Recupera o cilindro de projeção corrente.
normalize	Calcula o eixo e o raio do cilindro de projeção, em coordenadas normalizadas, de acordo com o volume de visualização corrente.
project	Projeta a posição do *mouse* normalizada sobre o cilindro de projeção, definindo sua posição 3D.
getRotation	Projeta a posição do *mouse* normalizada sobre o cilindro de projeção, utilizando-a para definir uma rotação em torno do eixo do cilindro de projeção, a partir de uma posição base.
setCylinder	Especifica o cilindro de projeção.
setOrient	Estabelece a orientação do cilindro de projeção.
setPart	Estabelece a parte do cilindro a ser interterceptada na projeção.

4.3.2 Classe ProjectLine

Esta é a subclasse de especialização da classe *Projector* que mapeia posições de um componente de arrasto que se movimenta sobre uma linha.

No início de uma manipulação, a linha de projeção é transformada para coordenadas normalizadas. O mapeamento da posição do *mouse* para a posição correspondente na linha de projeção requer calcular a interseção, ou o ponto mais próximo, entre a linha que passa pela posição do *mouse* normalizada, com direção [0, 0, 1], e a linha de projeção normalizada (Figura 4-2).

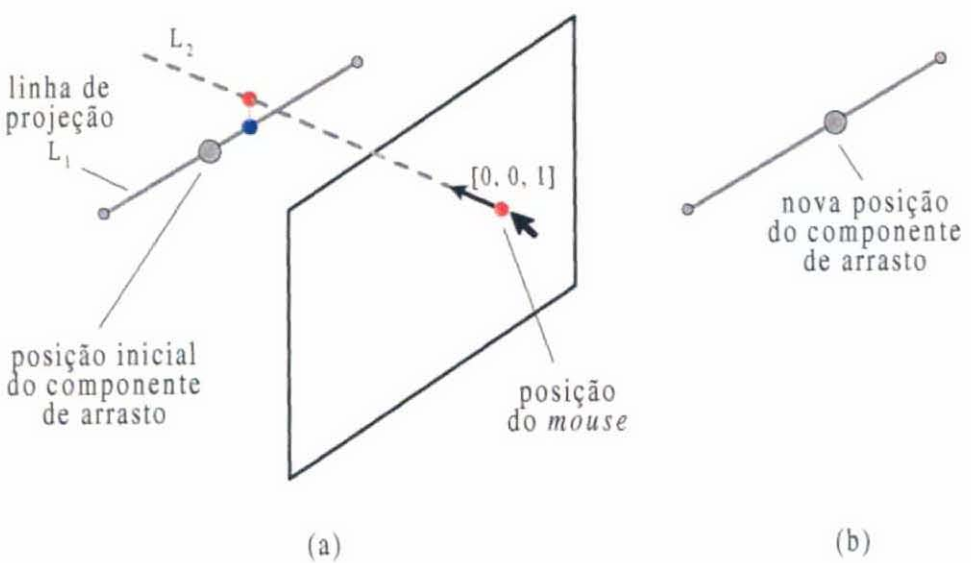


Figura 4-2 Mapeamento da posição do *mouse* sobre a linha de projeção (a), definindo a nova posição do componente de arrasto (b).

O algoritmo descrito em [Goldman (1990)] é utilizado. As duas linhas são definidas por um ponto P_i e um vetor direção unitário $\mathbf{V}_i$.

$$L_1(t) = P_1 + \mathbf{V}_1 t \qquad L_2(s) = P_2 + \mathbf{V}_2 s$$

A interseção ocorre em: $L_1(t) = L_2(s) \Rightarrow P_1 + \mathbf{V}_1 t = P_2 + \mathbf{V}_2 s$

Fazendo a linha de projeção corresponder a L_1, t é calculado. Se as linhas não se interceptam, t representa, em L_1, o ponto mais próximo entre as duas linhas (representado em azul, na Figura 4-2).

Tabela 4-15 Atributos da classe *ProjectLine*.

p1	Primeiro ponto que define a linha de projeção.
p2	Segundo ponto que define a linha de projeção.
norm_p1	Coordenadas normalizadas do primeiro ponto, em relação ao volume de visualização corrente.
norm_line	Linha de projeção em coordenadas normalizadas.
len_line	Comprimento da linha de projeção normalizada.
line_unit_vector	Vetor direção unitário da linha de projeção normalizada.

Tabela 4-16 Métodos da classe *ProjectLine*.

getLine	Recupera a linha de projeção corrente.
normalize	Calcula a linha de projeção em coordenadas normalizadas e seu vetor direção unitário, de acordo com o volume de visualização corrente.
project	Projeta a posição do *mouse* normalizada para a posição 3D sobre a linha de projeção.
setLine	Especifica a linha de projeção.

4.3.3 Classe *ProjectPlane*

Esta é a subclasse de especialização da classe *Projector* que mapeia posições de um componente de arrasto que se movimenta sobre um plano.

No início de uma manipulação, os pontos que definem o plano de projeção são transformados para coordenadas normalizadas, e o vetor normal **N** = [*A*, *B*, *C*] e o coeficiente *D* da equação do plano $Ax + By + Cz + D = 0$ são calculados.

O mapeamento da posição do *mouse* requer calcular a interseção entre a linha que passa pela posição do *mouse* normalizada, com direção [0, 0, 1], e o plano de projeção normalizado. Calcular esta interseção corresponde a substituir, na equação do plano, as coordenadas x e y pelas coordenadas (x, y) normalizadas da posição do *mouse*, obtendo-se a coordenada z normalizada.

Tabela 4-17 Atributos da classe *ProjectPlane*.

p1	Primeiro ponto que define o plano de projeção.
p2	Segundo ponto que define o plano de projeção.
p3	Terceiro ponto que define o plano de projeção.
D	Coeficiente *D* da equação do plano de projeção $Ax + By + Cz + D = 0$ (em coordenadas normalizadas).
N	Vetor [*A*, *B*, *C*], normal ao plano de projeção normalizado.

Tabela 4-18 Métodos da classe *ProjectPlane*.

getPlane	Recupera oplano de projeção corrente.
normalize	Calcula o coeficiente *D* e o vetor normal da equação do plano de projeção em coordenadas normalizadas, de acordo com o volume de visualização corrente.
project	Projeta a posição do *mouse* normalizada para a posição 3D sobre o plano de projeção.
setPlane	Especifica o plano de projeção.

4.4 Classe Dragger

A classe base *Dragger* define os objetos (componentes de arrasto) que compõem a geometria e o comportamento dos manipuladores.

Um componente de arrasto possui três tipos de geometria: de manipulação, de *feedback* e de *feedback* de estado ativo. O indicador de atividade de um componente de arrasto é utilizado para definir as geometrias a serem exibidas na operação de *rendering*.

Cada tipo de componente de arrasto trata diferentemente os movimentos do *mouse*, utilizando projetores para mapear os movimentos 2D em movimentos de um ponto sobre linhas 3D, planos 3D, esferas ou cilindros. A posição corrente de um componente de arrasto e a posição mapeada em um evento de arrasto são utilizadas para determinar a transformação (escala, rotação ou translação) equivalente a seu movimento, armazenada em um atributo que descreve o estado corrente do componente de arrasto (atributo de movimento).

Um movimento de rotação é especificado pelo vetor que define o eixo de rotação e pelo ângulo de rotação. Um movimento de translação é especificado por um vetor 3D.

Um componente de arrasto tem definido um identificador a ser utilizado na operação de *rendering* no modo de *picking*, para que o manipulador reconheça qual de seus componentes foi selecionado para interação.

Funções *callbacks* são definidas no componente de arrasto para tratar os eventos de interação com o *mouse*. As *callbacks* de início e de fim são definidas na classe base mas podem ser redefinidas pelas subclasses. A *callback* de movimento é definida pela subclasse que especializa o componente de arrasto, própria para o tipo de movimento que ele realiza. A *callback* de valor alterado é associada ao componente de arrasto pelo manipulador que o

emprega. Ela é executada quando o atributo de movimento é atualizado, para que o manipulador interprete e valide o movimento e se atualize.

A *callback* de início definida na classe base altera o estado do componente de arrasto para ativo, solicita a atualização do projetor com as matrizes de modelagem e visualização e de projeção correntes, e com o cálculo de suas inversas, solicita a transformação do projetor para coordenadas normalizadas e define o ponto inicial da manipulação. A *callback* de fim definida na classe base restaura o estado do componente de arrasto para inativo.

Componentes de arrasto simples possuem apenas uma geometria de manipulação e um atributo de movimento. Componentes de arrasto compostos utilizam vários componentes simples para formar sua geometria e consolidam seus movimentos para compor transformações complexas.

As geometrias dos componentes de arrasto simples são definidas pelos manipuladores ou pelos componentes compostos que os empregam.

Tabela 4-19 Atributos da classe base *Dragger*.

is_active	Indicador de atividade.
geometry	Geometria de manipulação.
feedback_geometry	Lista de geometrias de *feedback*.
active_feedback_geometry	Lista de geometrias de *feedback* de estado ativo.
current_position	Posição 3D corrente do componente de arrasto.
pickid	Identificador para operações de *picking*.

Tabela 4-20 Métodos da classe base *Dragger*.

addActiveFeedback	Adiciona uma geometria à lista de geometrias de *feedback* de estado ativo.
addFeedBack	Adiciona uma geometria à lista de geometrias de *feedback*.
getBoundingBox	Método virtual que calcula a caixa envolvente do componente de arrasto. Definido pelas subclasses.
getNormalizedPosition	Calcula a posição do *mouse* em coordenadas normalizadas.
getProjector	Método virtual que recupera os dados que definem o projetor utilizado pelo componente de arrasto. Definido pelas subclasses.
render	Desenha as geometrias do componente de arrasto. Pode ser redefinido por uma subclasse.
renderPick	Desenha a geometria de manipulação do componente de arrasto no modo de *picking* do *OpenGL*. Pode ser redefinido por uma subclasse.
setGeometry	Especifica a geometria de manipulação.
setProjector	Método virtual que especifica o projetor a ser utilizado pelo componente de arrasto. Definido pelas subclasses.
setTransformation	Estabelece uma transformação para posicionamento do componente de arrasto.
setFinishCb	Associa uma *callback* ao componente de arrasto para tratar o fim de uma manipulação.
setMotionCb	Associa uma *callback* ao componente de arrasto para tratar os movimentos de uma manipulação.
setStartCb	Associa uma *callback* ao componente de arrasto para tratar o início de uma manipulação.
setValueChangedCb	Associa uma *callback* ao componente de arrasto para informar o manipulador sobre um movimento da manipulação.

Tabela 4-21 *Callbacks* da classe base *Dragger*.

finishCallback	*Callback* chamada no fim da manipulação do componente de arrasto. Pode ser redefinida pelas subclasses.
motionCallback	*Callback* chamada após cada movimento do *mouse* durante a manipulação do componente de arrasto. Definida pelas subclasses.
startCallback	*Callback* chamada no início da manipulação do componente de arrasto. Pode ser redefinida pelas subclasses.
valueChangedCallback	*Callback* chamada após uma alteração do atributo de movimento do componente de arrasto, para que seu manipulador valide o movimento e se atualize. Definida pelo manipulador que utiliza o componente de arrasto.

4.4.1 Classe *TranslateLine*

Esta é a subclasse de especialização da classe *Dragger* que define o componente de arrasto simples que se movimenta sobre uma linha, produzindo transformações de translação.

O componente de arrasto *TranslateLine* utiliza o atributo de movimento translação, para armazenar seu estado corrente, e um projetor de linha, para mapear seus movimentos.

A *callback* de movimento definida na subclasse *TranslateLine* solicita ao projetor de linha o mapeamento da posição do *mouse* normalizada, define uma translação a partir da posição corrente do componente de arrasto e invoca a *callback* de valor alterado associada ao componente de arrasto.

O manipulador que emprega um componente de arrasto *TranslateLine* pode desejar limitar seu movimento. O componente de arrasto *TranslateLine* tem definido um método para restringir seu movimento ao segmento de reta definido pelos dois pontos da linha de

projeção, que consiste em utilizar a equação paramétrica da linha de projeção para definir o valor de t da nova posição do componente de arrasto.

$$P(t) = P_1 + t(P_2 - P_1)$$

- Se $0. \leq t \leq 1.$, a nova posição é aceita;
- Se $t < 0.$, o ponto P_1 é definido como a nova posição do componente de arrasto;
- Se $t > 1.$, o ponto P_2 é definido como a nova posição do componente de arrasto.

O manipulador fica responsável por qualquer outro tipo de limitação para o movimento do componente de arrasto *TranslateLine*.

Tabela 4-22 Atributos da classe *TranslateLine*.

translation	Translação definida por um evento de arrasto do componente de arrasto sobre a linha de projeção.
line_projector	Projetor de linha utilizado pelo componente de arrasto.

Tabela 4-23 Métodos da classe *TranslateLine*.

getBoundingBox	Calcula a caixa envolvente.
getProjector	Recupera o projetor de linha.
setProjector	Estabelece o projetor de linha.
validateTranslation	Valida o movimento do componente de arrasto, limitando-o ao segmento de reta definido pelos dois pontos da linha de projeção.

4.4.2 Classe TranslatePlane

A classe *TranslatePlane* é a subclasse de especialização da classe *Dragger* que define o componente de arrasto simples que se movimenta sobre um plano, produzindo transformações de translação.

O componente de arrasto *TranslatePlane* utiliza um projetor de plano, para mapear seus movimentos, e o atributo de movimento translação, para armazenar seu estado corrente.

A *callback* de movimento definida na subclasse *TranslatePlane* solicita ao projetor de plano o mapeamento da posição do *mouse* normalizada, define uma translação a partir da posição corrente do componente de arrasto e invoca a *callback* de valor alterado associada ao componente de arrasto.

O manipulador que emprega um componente de arrasto *TranslatePlane* pode desejar limitar seu movimento. O componente de arrasto *TranslatePlane* tem definido um método para restringir seu movimento ao quadrilátero definido pelos três pontos do plano de projeção. O método consiste em definir se a nova posição do componente de arrasto é ou não interior ao quadrilátero e, no caso de não ser interior, calcular a posição em que a linha definida pela posição corrente e pela nova posição corta o quadrilátero, definindo-a como a nova posição do componente de arrasto. O manipulador fica responsável por qualquer outro tipo de limitação para o movimento do componente de arrasto *TranslatePlane*.

Tabela 4-24 Atributos da classe *TranslatePlane*.

translation	Translação definida por um evento de arrasto do componente de arrasto sobre o plano de projeção.
plane_projector	Projetor de plano utilizado pelo componente de arrasto.

Tabela 4-25 Métodos da classe *TranslatePlane*.

getBoundingBox	Calcula a caixa envolvente.
getProjector	Recupera o projetor de plano.
setProjector	Estabelece o projetor de plano.
validateTranslation	Valida o movimento do componente de arrasto, limitando-o ao quadrilátero definido pelos pontos do plano de projeção.

4.4.3 Classe *RotateCylinder*

Esta é a subclasse de especialização da classe *Dragger* que define o componente de arrasto simples que se movimenta sobre a superfície de um cilindro, definindo transformações de rotação em torno de seu eixo.

O componente de arrasto *RotateCylinder* utiliza um projetor de cilindro, para mapear seus movimentos, e o atributo de movimento rotação, para armazenar seu estado corrente.

A *callback* de movimento definida na subclasse *RotateCylinder* solicita ao projetor de cilindro o mapeamento da posição do *mouse* normalizada e a definição da rotação, dada a posição corrente do componente de arrasto, e invoca a *callback* de valor alterado associada ao componente de arrasto.

O componente de arrasto *RotateCylinder* pré-define a orientação do projetor de cilindro e a parte a ser utilizada na projeção: em direção ao olho e a parte da frente, respectivamente. O manipulador que emprega um componente de arrasto *RotateCylinder* pode redefinir esses atributos.

Tabela 4-26 Atributos da classe *RotateCylinder*.

rotation	Rotação definida por um evento de arrasto do componente de arrasto sobre o cilindro de projeção.
cylinder_projector	Projetor de cilindro utilizado pelo componente de arrasto.

Tabela 4-27 Métodos da classe *RotateCylinder*.

getBoundingBox	Calcula a caixa envolvente.
getProjector	Recupera o projetor de cilindro.
setProjector	Estabelece o projetor de cilindro.

4.4.4 Classe *Box*

A classe *Box* é a subclasse de especialização da classe *Dragger* que define o componente de arrasto composto, com a forma de uma caixa, cujos componentes de arrasto simples produzem translações que alteram as dimensões e o posicionamento da caixa.

Dispondo de um dispositivo 3D de entrada, transladar um canto da caixa no espaço permitiria a alteração das três dimensões da caixa, assim como transladar uma aresta no espaço permitiria a alteração de duas dimensões. Com o *mouse*, isso não é possível. Foi necessário estabelecer esses comportamentos complexos a partir dos movimentos que podem ser mapeados diretamente (translações sobre linhas e planos). Uma opção para alterar uma dimensão poderia ser empurrar (ou puxar) uma face perpendicular às arestas que definem essa dimensão. Mas, além de só possibilitar a alteração de uma dimensão de cada vez, como seria feita a translação de toda a caixa? A solução adotada foi a de permitir movimentos sobre plano aos cantos da caixa. Como são três os planos adjacentes a um canto, foram definidos três componentes de arrasto sobre plano para cada canto. Seguindo o mesmo raciocínio,

foram definidos dois componentes de arrasto sobre linha para as arestas. Também a translação de toda a caixa foi decomposta, com as faces sendo utilizadas para transladar a caixa sobre seus planos.

A Figura 4-3 apresenta a composição de componentes de arrasto simples em cada face do componente de arrasto composto *Box*. Os componentes de arrasto posicionados nos cantos da face executam movimentos de translação sobre o plano da face e alteram duas dimensões da caixa ao mesmo tempo. Os componentes de arrasto posicionados no meio das arestas da face executam movimentos de translação sobre a linha definida pela sua posição e a posição do componente de arrasto da aresta oposta e alteram uma dimensão da caixa. A própria face é um componente de arrasto que se movimenta sobre plano, utilizado para transladar toda a caixa sobre o plano da face.

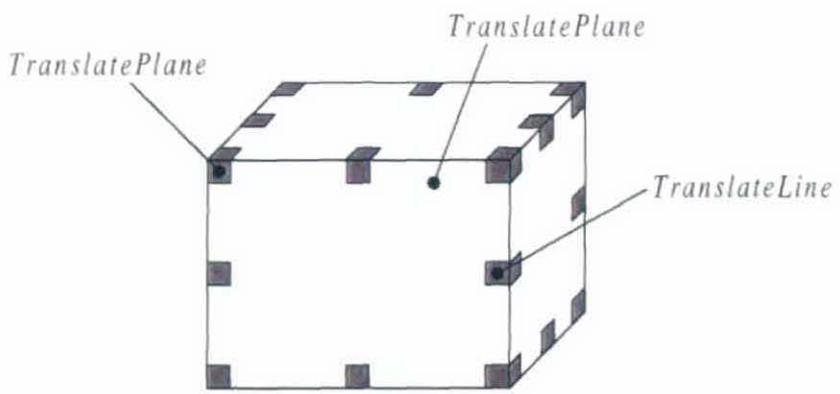


Figura 4-3 Composição de cada face do componente de arrasto *Box*.

A geometria de manipulação do componente de arrasto composto *Box* é definida pelos seus componentes de arrasto simples. A geometria de manipulação dos componentes de arrasto simples dos cantos (*corners*) e das arestas (*edges*) da caixa é o de um quadrado posicionado de forma a indicar a face sobre cujo plano eles se movimentam. Um

manipulador que empregue um componente de arrasto composto *Box* pode alterar os valores pré-definidos para o material e para o tamanho dos quadrados. A geometria de manipulação dos componentes de arrasto equivalentes às faces da caixa é invisível. A geometria de *feedback* de estado ativo do componente de arrasto composto *Box* é um contorno da caixa, indicando sua dimensão e posicionamento correntes.

O movimento de um componente de arrasto simples deve se refletir para os outros componentes, para que a forma de caixa seja mantida. Para isso, cada canto contém associações para os cantos, arestas e faces a ele ligados, nas três direções da caixa, cada aresta contém associações para os dois cantos a ela ligados e as *callbacks* originais que tratam os eventos de manipulação dos componentes de arrasto simples são redefinidas.

A *callback* de início definida no componente de arrasto *Box* substitui as *callbacks* de início definidas nos componentes de arrasto *TranslateLine* e *TranslatePlane*, para que também o componente de arrasto composto seja marcado como ativo, permitindo que sua geometria de *feedback* de estado ativo seja apresentada durante a manipulação.

As *callbacks* de movimento dos componentes de arrasto simples *TranslateLine* e *TranslatePlane* também são redefinidas no componente de arrasto *Box*. Além de mapear seus movimentos, elas atualizam a posição da geometria de todos os componentes simples afetados pelo movimento e as coordenadas dos vértices da caixa.

O componente de arrasto *Box* redefine as *callbacks* de fim dos componentes de arrasto *TranslateLine* e *TranslatePlane*. O movimento de um componente de arrasto simples não se reflete somente para a posição de outros componentes simples. Ele causa alterações também nos projetores a eles associados, que são atualizados pelas *callbacks* redefinidas.

Tabela 4-28 Atributos da classe *Box*.

corner[8]	Cantos da caixa.
edge[12]	Arestas da caixa.
face[6]	Faces da caixa.
box[8]	Coordenadas dos cantos da caixa.

Tabela 4-29 Métodos da classe *Box*.

getBoundingBox	Calcula a caixa envolvente do componente de arrasto *Box*.
render	Desenha as geometrias do componente de arrasto *Box*. Redefine o método da classe base.
renderPick	Desenha as geometrias de manipulação do componente de arrasto *Box* no modo de *picking*. Redefine o método da classe base.

4.4.5 Classe *TranslateSpace*

Esta é a subclasse de especialização da classe *Dragger* que define o componente de arrasto composto cujos componentes de arrasto simples produzem translações em três dimensões.

Como foi analisado na seção anterior, utilizar o *mouse* como dispositivo de interação 3D resultou na definição de translações no espaço como composições de movimentos diretamente mapeados (translações sobre linhas e planos).

O componente de arrasto composto *TranslateSpace* se compõe de três componentes de arrasto simples de movimento sobre linha e três componentes de arrasto simples de

movimento sobre plano, que se interceptam em um único ponto (posição corrente do componente de arrasto composto).

Os três componentes de arrasto simples que se movimentam sobre linha são sempre exibidos e são utilizados para transladar o componente de arrasto *TranslateSpace* nas direções dos eixos de seu espaço de coordenadas local.

Apenas um dos componentes de arrasto que se movimentam sobre plano fica disponível para manipulação. Inicialmente, o componente de arrasto disponível é o que se movimenta sobre o plano que passa pela posição do componente de arrasto *TranslateSpace* e é perpendicular ao eixo Z do seu espaço de coordenadas local. A tecla <Alt> é utilizada para alternar a disponibilidade para manipulação entre os três componentes.

O atributo de translação do componente de arrasto composto *TranslateSpace* acumula a translação total definida pela manipulação de um componente de arrasto simples. O movimento de um componente de arrasto simples do componente de arrasto *TranslateSpace* deve se refletir para os outros componentes simples. Para isso, as *callbacks* originais que tratam os eventos de manipulação dos componentes de arrasto simples são redefinidas.

A *callback* de início definida no componente de arrasto *TranslateSpace* substitui as *callbacks* de início definidas nos componentes de arrasto simples *TranslateLine* e *TranslatePlane*, para que também o componente de arrasto composto seja marcado como ativo, permitindo que sua geometria de *feedback* de estado ativo seja apresentada durante a manipulação.

As *callbacks* de movimento dos componentes de arrasto simples *TranslateLine* e *TranslatePlane* também são redefinidas no componente de arrasto *TranslateSpace*. Além de mapear seus movimentos, elas atualizam a posição das geometrias de manipulação e de

feedback de todos os componentes simples afetados pelo movimento e o atributo de translação do componente de arrasto *TranslateSpace*.

O componente de arrasto *TranslateSpace* redefine as *callbacks* de fim dos componentes de arrasto *TranslateLine* e *TranslatePlane*. O movimento de um componente de arrasto simples não se reflete somente para a posição de outros componentes simples. Ele causa alterações nos projetores a eles associados, que são atualizados pela *callbacks* redefinidas.

O manipulador que emprega um componente de arrasto *TranslateSpace* deve definir as geometrias de manipulação e de *feedback* dos componentes de arrasto simples.

Tabela 4-30 Atributos da classe *TranslateSpace*.

t_line[3]	Componentes de arrasto *TranslateLine*.
t_plane[3]	Componentes de arrasto *TranslatePlane*.
current_t_plane	Componente de arrasto *TranslatePlane* ativo.
translation	Translação total definida pela manipulação de um componente de arrasto simples do componente de arrasto *TranslateSpace*.

Tabela 4-31 Métodos da classe *TranslateSpace*.

activateNextTranslatePlane	Ativa próximo componente de arrasto *TranslatePlane*, desativando o corrente.
getBoundingBox	Calcula a caixa envolvente do componente de arrasto *TranslateSpace*.
render	Desenha as geometrias do componente de arrasto *TranslateSpace*.
renderPick	Desenha as geometrias de manipulação do componente de arrasto *TranslateSpace* no modo de *picking*.

4.5 Classe Manipulator

A classe *Manipulator* é a classe base que define os objetos que se apresentam em uma cena 3D, associados à visualizações, e que são utilizados para editá-las.

Um manipulador emprega componentes de arrasto simples ou compostos para compor sua geometria e seu comportamento. Os resultados das movimentações dos componentes de arrasto de um manipulador são interpretados e validados por ele e utilizados para definir uma edição sobre a visualização à qual está associado. As funções *callbacks* de um manipulador são utilizadas para que ele se comunique com a visualização à qual está associado, durante as manipulações.

Realizar o *rendering* de um manipulador corresponde a realizar o *rendering* dos componentes de arrasto que o compõem. Os manipuladores possuem um identificador, utilizado na operação de *rendering* no modo de *picking*, para que a cena possa reconhecer o manipulador selecionado para interação. Ao tratar um evento de início de uma manipulação, o manipulador reconhece o componente de arrasto simples selecionado para interação, armazenando-o como um atributo seu (*picked_dragger*), para que os eventos de manipulação subseqüentes sejam enviados diretamente para ele.

Tabela 4-32 Atributos da classe base *Manipulator*.

scene	Cena da qual o manipulador faz parte.
visualization	Visualização à qual o manipulador está associado.
bbox	Caixa envolvente do manipulador.
pickid	Identificador para operações de *picking*.
picked_dragger	Componente de arrasto selecionado para manipulação.

Tabela 4-33 Métodos da classe base *Manipulator*.

fit	Método virtual que estabelece a composição do manipulador. Definido pelas subclasses.
getBoundingBox	Método virtual que calcula a caixa envolvente do manipulador. Definido pelas subclasses.
handleEvent	Método virtual que recebe e encaminha para tratamento os eventos de manipulação com os componentes que compõem o manipulador. Definido pelas subclasses.
render	Método virtual que desenha a geometria do manipulador. Definido pelas subclasses.
renderPick	Método virtual que desenha a geometria do manipulador no modo de *picking*. Definido pelas subclasses.
setFinishCb	Associa uma *callback* ao manipulador para tratar o fim de uma manipulação.
setMotionCb	Associa uma *callback* ao manipulador para tratar os movimentos de uma manipulação.
setStartCb	Associa uma *callback* ao manipulador para tratar o início de uma manipulação.

Tabela 4-34 *Callbacks* da classe base *Manipulator*.

finishCallback	*Callback* chamada no fim da manipulação.
motionCallback	*Callback* chamada após cada movimento do *mouse*, durante a manipulação.
startCallback	*Callback* chamada no início da manipulação.

5. Protótipo de Manipulação 3D de Dados Volumétricos

Baseado nas necessidades requeridas pelo trabalho de interpretação, um conjunto de interfaces interativas (manipuladores) para manipulação de representações de dados sísmicos volumétricos está sendo definido.

O *toolkit* para construção de aplicações 3D interativas de visualização científica está sendo utilizado para desenvolver um visualizador 3D que permite a visualização dos objetos gráficos descritos na seção **3.1**. Este trabalho utiliza essa aplicação e incorpora, como um protótipo para avaliação, três manipuladores: para movimentar visualizações do tipo fatia nas direções do volume; para definir subvolumes; e para investigar dados do volume. As seções seguintes descrevem estes manipuladores.

O trabalho está sendo desenvolvido na gerência **E&P/GEREX/GETINF/GEDES**, na Petrobras. A estação utilizada é a **SGI Indigo^2 IMPACT**. A linguagem **C++** e ferramentas de modelagem e programação orientados a objetos estão sendo introduzidos na gerência. Por isto, a implementação utilizou a linguagem **C**, mas com uma disciplina de programação orientada a objetos que facilitará a futura conversão para o novo ambiente.

5.1 *Manipulador* Probe

Subclasse de especialização da classe base *Manipulator* que define o manipulador para investigação do volume, associado a uma visualização direta de volume para retornar informações sobre as amostras.

O manipulador para investigação de volume emprega o componente de arrasto composto *TranslateSpace* para compor sua geometria e seu comportamento. A posição inicial do manipulador corresponde à posição da amostra central do volume.

A geometria de manipulação definida para um componente de arrasto simples *TranslateLine* do componente de arrasto *TranslateSpace* é a de um cilindro orientado longitudinalmente a um dos eixos do volume, com altura igual à dimensão do volume nesse eixo. Dois cubos compõem a geometria de *feedback*, para indicar os limites do volume nessa direção. A Figura 5-1 mostra os componentes de arrasto no estado inativo. A posição onde as três geometrias de manipulação dos componentes de arrasto *TranslateLine* se interceptam corresponde à posição do manipulador.

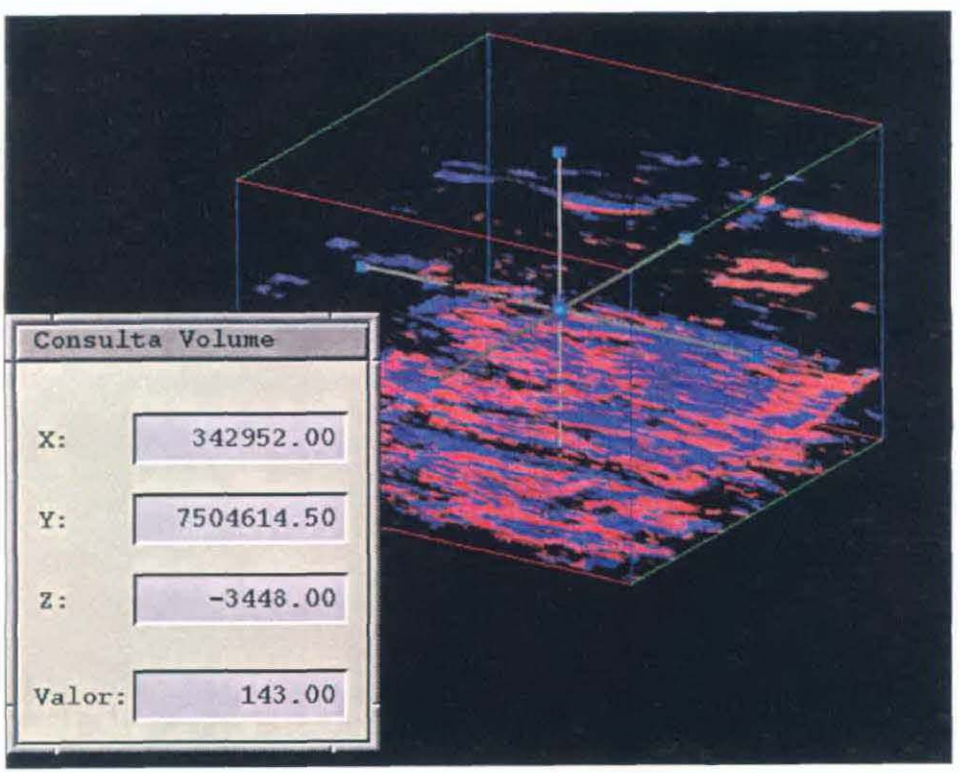


Figura 5-1 Manipulador para investigação do volume no estado inativo.

Em uma interação com um componente de arrasto *TranslateLine*, a geometria de *feedback* de estado ativo utilizada é a de um cilindro vermelho indicando a posição corrente do movimento, como ilustra a Figura 5-2.

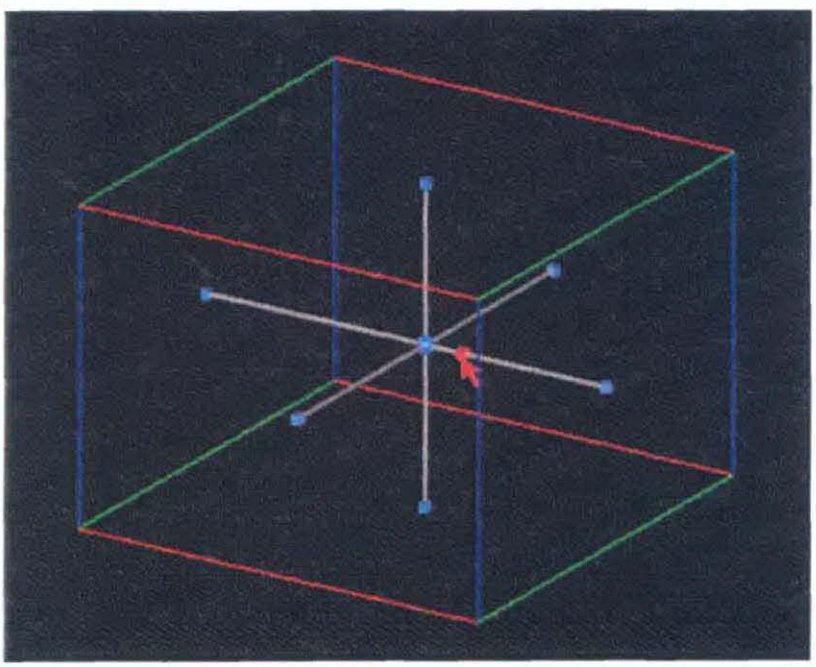


Figura 5-2 Interação com um componente de arrasto *TranslateLine*.

A geometria de manipulação definida para um componente de arrasto simples *TranslatePlane* do componente de arrasto *TranslateSpace* é uma esfera cujo centro corresponde à localização do manipulador. A Figura 5-1 ilustra a geometria e, conforme apresentado no capítulo anterior, no item **4.4.5**, apenas um dos componentes de arrasto *TranslatePlane* fica disponível para manipulação. A tecla <Alt> é utilizada para alternar a disponibilidade para manipulação entre os três componentes de arrasto. Em uma interação com um componente de arrasto *TranslatePlane*, a esfera passa a ser desenhada em vermelho (Figura 5-3).

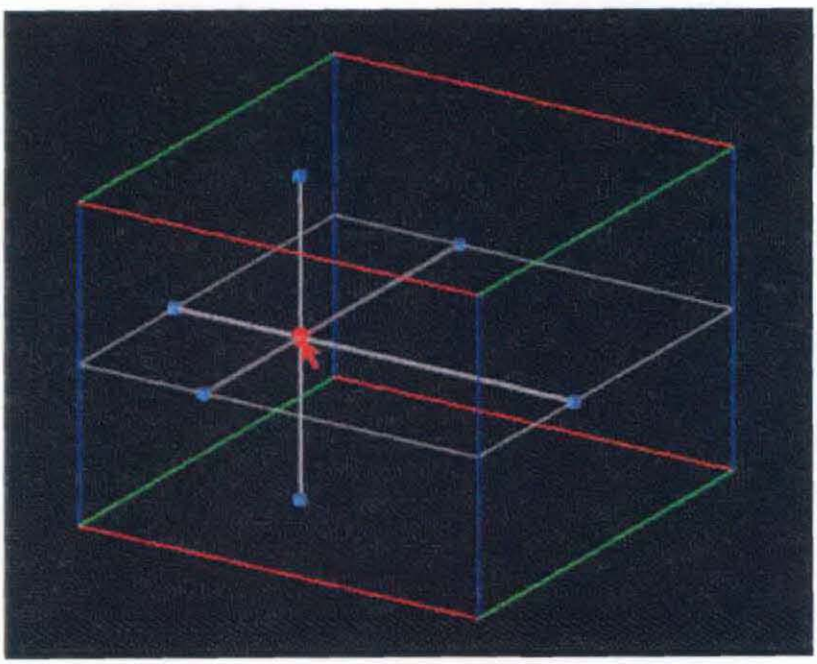


Figura 5-3 Interação com um componente de arrasto *TranslatePlane*.

Tabela 5-1 Atributos da classe *Probe*.

translatespace	Componente de arrasto *TranslateSpace* que compõe o manipulador.
position	Posição do manipulador no volume.
widgets	Identificadores dos *widgets* utilizados para apresentar as informações das amostras.

Tabela 5-2 Métodos da classe *Probe*.

fit	Estabelece a composição geométrica e o comportamento do manipulador.
getBoundingBox	Calcula a caixa envolvente.
handleEvent	Recebe e encaminha para tratamento os eventos de manipulação com os componentes de arrasto simples do manipulador.
render	Desenha as geometrias do manipulador.
renderPick	Desenha as geometrias de manipulação do manipulador no modo de *picking*.
translateLineFit	Especifica um componente de arrasto *TranslateLine* do componente de arrasto *TranslateSpace* do manipulador.
translatePlaneFit	Especifica um componente de arrasto *TranslatePlane* do componente de arrasto *TranslateSpace* do manipulador.

Tabela 5-3 *Callbacks* da classe *Probe*.

finishCallback	Não utilizada.
motionCallback	Recupera informação da amostra relativa à posição corrente.
startCallback	Não utilizada.

5.2 *Manipulador* SliceMover

Subclasse de especialização da classe base *Manipulator* que define o manipulador para movimentação de uma visualização do tipo fatia de volume.

O manipulador para movimentação de fatia de volume é composto de quatro componentes de arrasto simples *TranslateLine*, posicionados nos quatro cantos da fatia. As geometrias de manipulação são cilindros azuis orientados sobre as arestas do volume perpendiculares ao plano da fatia (Figura 5-4) e sugerem o comportamento de um *slider*,

indicando a possibilidade de movimentos 1D.

A posição corrente da visualização de fatia é armazenada no manipulador e definida relativamente em relação à extensão do volume na direção do movimento.

Um evento de botão do *mouse* apertado sobre um dos quatro componentes de arrasto *TranslateLine* inicia a movimentação da fatia e o componente de arrasto selecionado para interação (componente de arrasto ativo) exibe geometrias adicionais (de *feedback* de estado ativo) para indicar a direção e os limites para seu movimento (Figura 5-5). A geometria que indica a direção do movimento é a de um cilindro cinzento orientado sobre a aresta onde o componente de arrasto está posicionado, com raio menor que o do cilindro da geometria de manipulação, e com altura equivalente ao comprimento da aresta. Dois cilindros azuis localizados nos extremos da aresta, com mesmo raio que o cilindro de manipulação, mas com altura menor, indicam os limites do movimento.

Os movimentos de arrasto do *mouse* com o botão apertado determinam translações sobre a linha que define a direção do movimento do componente de arrasto *TranslateLine* ativo. O manipulador implementa a função *callback* de valor alterado, a ser invocada pelo componente de arrasto ativo quando o valor do seu atributo de translação for alterado. Essa função valida a translação, para garantir que o movimento fique restrito ao intervalo definido pelos vértices da aresta sobre a qual o componente de arrasto se movimenta, atualiza os demais componentes de arrasto do manipulador, mapeia a translação validada para uma posição relativa do componente de arrasto, em relação à dimensão da aresta, e atualiza a posição corrente da fatia, no manipulador.

De posse da nova posição da fatia, o manipulador invoca a sua função *callback* de movimento para passar essa informação para a visualização.

A sequência de figuras apresentada a seguir, Figura 5-4 à Figura 5-6, ilustra uma interação com o manipulador para movimentação de fatia.

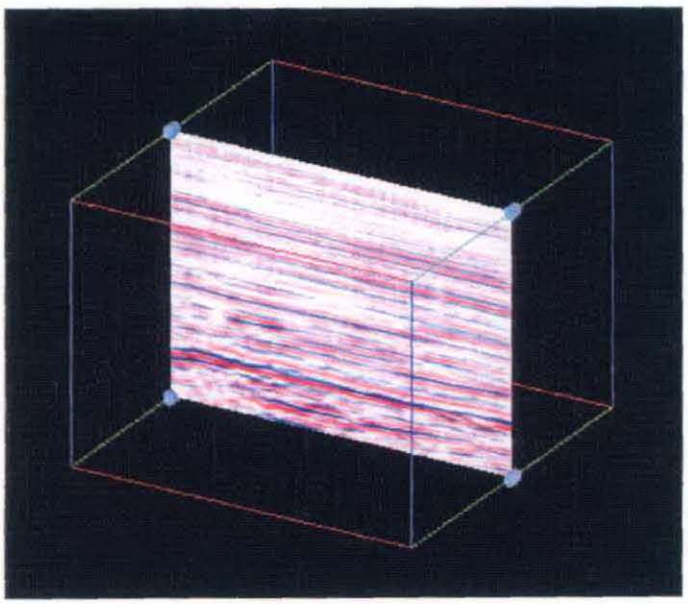


Figura 5-4 Posição corrente da fatia do volume e manipulador no estado inativo.

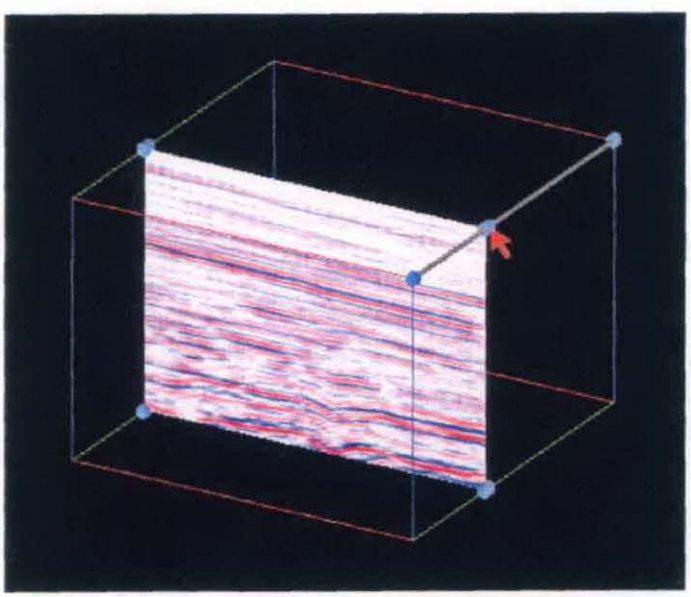


Figura 5-5 Interação com um componente de arrasto *TranslateLine*, que apresenta suas geometrias de *feedback* de estado ativo.

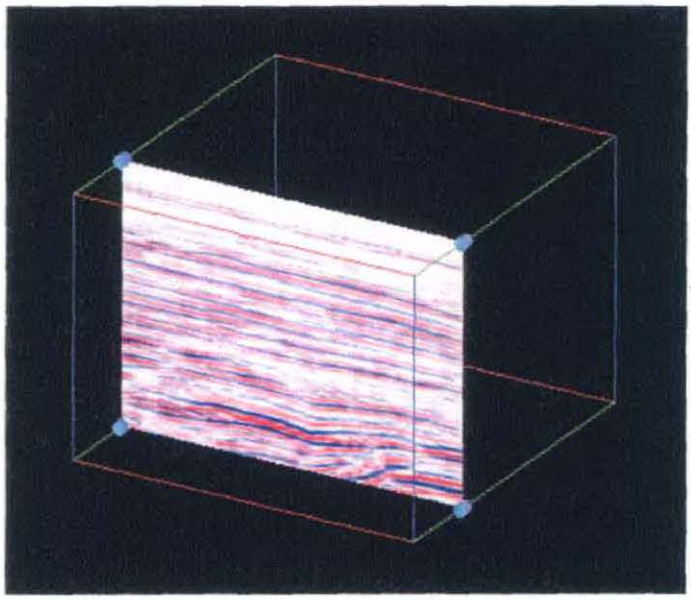


Figura 5-6 Manipulação encerrada. Visualização da fatia de volume em sua nova posição.

Tabela 5-4 Atributos da classe *SliceMover*.

t_line[4]	Componentes de arrasto *TranslateLine* que compõem o manipulador.
cut_position	Posição relativa da visualização de fatia à qual o manipulador está associado.

Tabela 5-5 Métodos da classe *SliceMover*.

fit	Estabelece a composição geométrica e o comportamento do manipulador.
getBoundingBox	Calcula a caixa envolvente.
handleEvent	Recebe e encaminha para tratamento os eventos de manipulação dos componentes de arrasto simples utilizados pelo manipulador.
render	Desenha as geometrias do manipulador.
renderPick	Desenha as geometrias de manipulação do manipulador no modo de *picking*.

Tabela 5-6 *Callbacks* da classe *SliceMover*.

finishCallback	Não utilizada.
motionCallback	Comunica à visualização de fatia associada sua nova posição relativa.
startCallback	Não utilizada.

5.3 Manipulador Subvolume

Subclasse de especialização da classe base Manipulator que define o manipulador que é associado a uma visualização direta de volume para definir um subconjunto do volume original (subvolume).

O manipulador para definição de subvolume emprega o componente de arrasto composto *Box* para compor sua geometria e seu comportamento. As coordenadas iniciais do subvolume, armazenadas no componente de arrasto *Box*, correspondem às coordenadas do volume original, como ilustra a Figura 5-7.

Como apresentado no capítulo anterior, no item **4.4.4**, a forma geométrica das geometrias de manipulação dos componentes de arrasto simples dos cantos e das arestas do componente de arrasto composto *Box* é a de um quadrado. O manipulador para definição de subvolume define a cor azul e calcula o tamanho do quadrado em função das dimensões do volume.

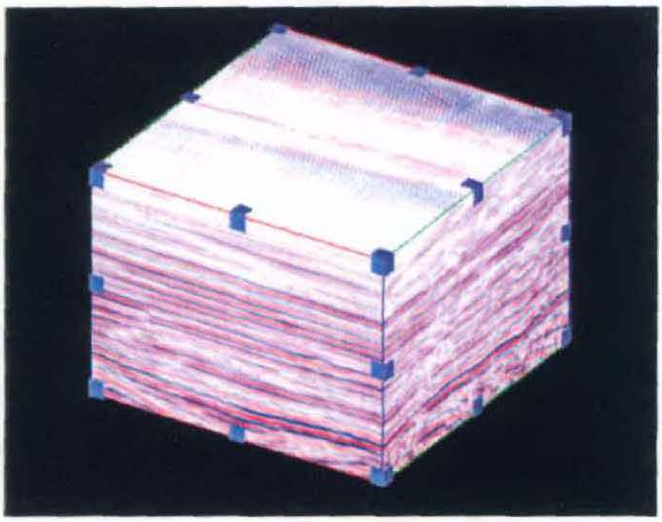


Figura 5-7 Manipulador *Subvolume* no estado inicial.

Um evento de botão do *mouse* apertado sobre um dos componentes de arrasto simples do componente de arrasto composto *Box* inicia uma edição das dimensões do manipulador. A manipulação de um componente de arrasto *TranslatePlane* de um canto altera duas dimensões do manipulador. A manipulação de um componente de arrasto *TranslateLine* de uma aresta altera uma dimensão do manipulador. A manipulação de um componente de arrasto *TranslatePlane* de uma face translada todo o manipulador sobre o plano da face. Durante uma manipulação, o *rendering* do volume não é realizado e a geometria de *feedback* de estado ativo utilizada é a de uma caixa em *wireframe* que indica a posição e as dimensões correntes do manipulador.

A cada manipulação, o componente de arrasto composto *Box* é responsável por manter os relacionamentos entre os componentes de arrasto simples que o compõem. Ao manipulador cabe garantir que o movimento não extrapole os limites do volume original e que o volume não seja invertido, o que é feito pela função *callback* de valor alterado nele implementada, a ser invocada pelo componente de arrasto ativo quando o valor do seu atributo de translação for alterado. Validada a translação de um componente de arrasto

simples, as coordenadas do manipulador são atualizadas.

No fim de uma manipulação, o manipulador invoca a função *callback* de fim para passar para a visualização as coordenadas a serem utilizadas na geração do subvolume.

As Figuras 5-8 e 5-9 ilustram uma interação com um componente de arrasto simples *TranslatePlane* de um canto (indicado pela seta).

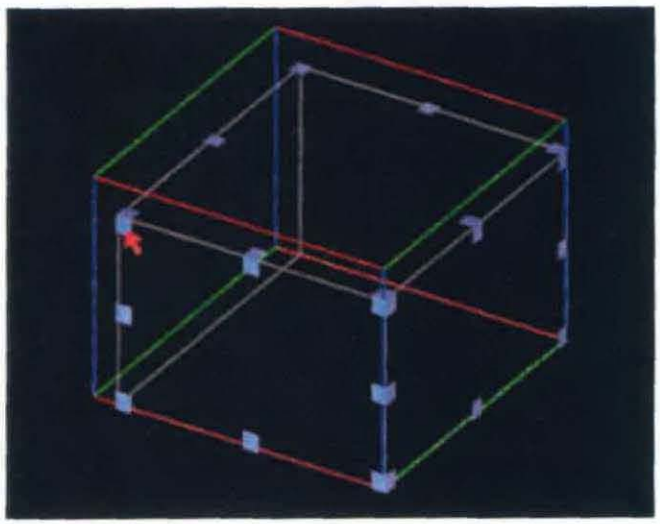


Figura 5-8 Interação com um componente de arrasto *TranslatePlane*.

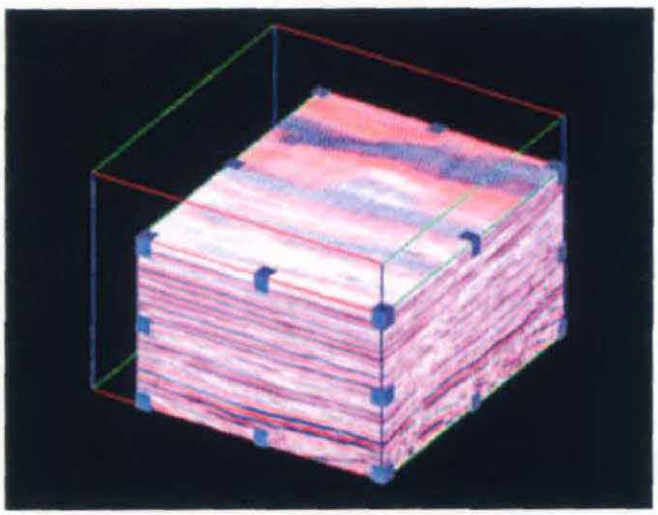


Figura 5-9 Finalização da manipulação do componente de arrasto *TranslatePlane* e visualização do subvolume gerado.

As Figuras 5-10 e 5-11 ilustram uma interação com um componente de arrasto simples *TranslateLine* (indicado pela seta).

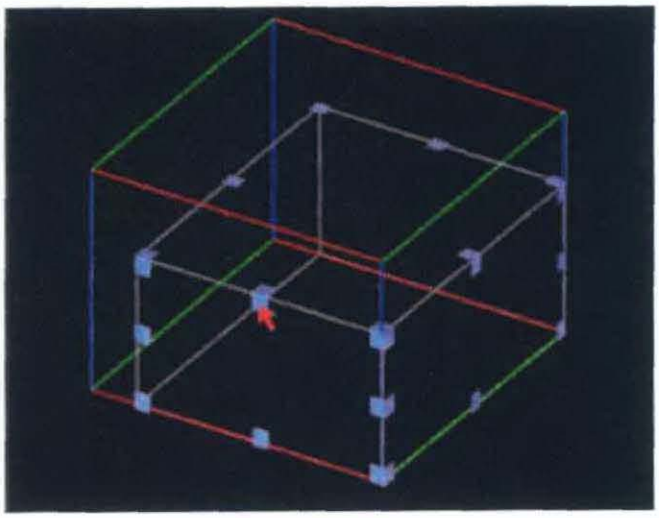


Figura 5-10 Interação com um componente de arrasto *TranslateLine*.

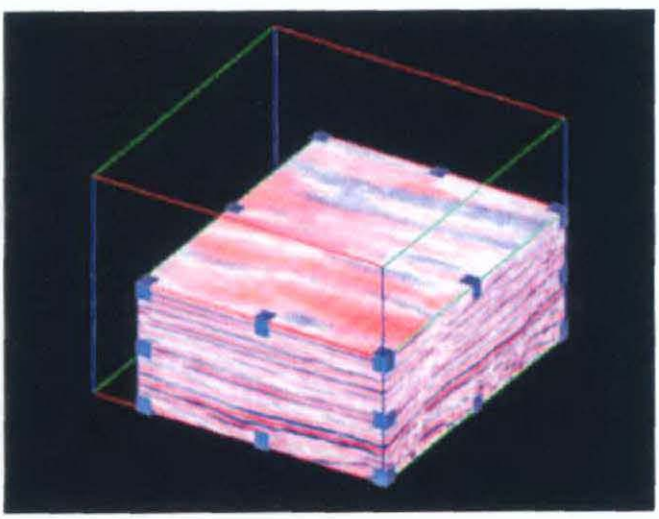


Figura 5-11 Finalização da manipulação do componente de arrasto *TranslateLine* e visualização do subvolume gerado.

As Figuras 5-12 e 5-13 ilustram uma interação com um componente de arrasto simples *TranslatePlane* de uma face (indicado pela seta).

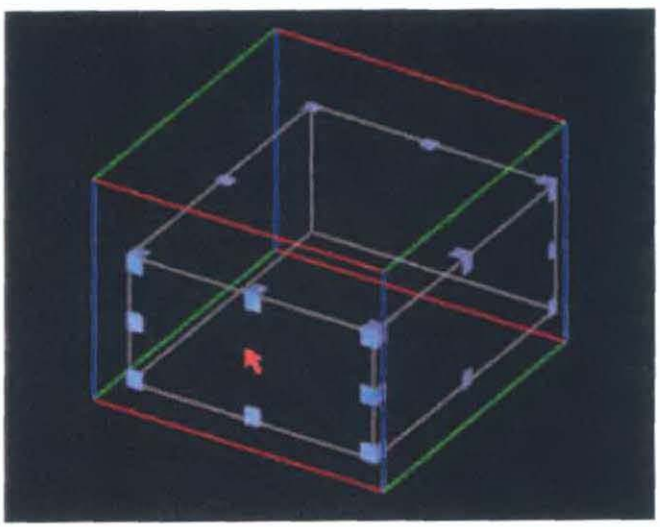


Figura 5-12 Interação com um componente de arrasto *TranslatePlane*.

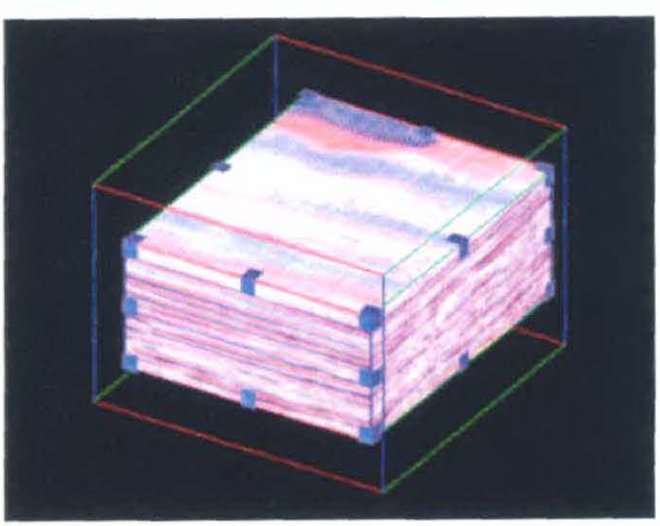


Figura 5-13 Finalização da manipulação do componente de arrasto *TranslatePlane* e visualização do subvolume gerado.

Tabela 5-7 Atributos da classe *Subvolume*.

box	Componente de arrasto *Box* que compõe o manipulador.

Tabela 5-8 Métodos da classe *Subvolume*.

fit	Estabelece a composição geométrica e o comportamento do manipulador.
getBoundingBox	Calcula a caixa envolvente.
handleEvent	Recebe e encaminha para tratamento os eventos de manipulação com os componentes de arrasto utilizados pelo manipulador.
render	Desenha as geometrias do manipulador.
renderPick	Desenha as geometrias de manipulação do manipulador no modo de *picking*.
translateLineFit	Especifica um componente de arrasto *TranslateLine* do componente de arrasto *Box* do manipulador.
translatePlaneFit	Especifica um componente de arrasto *TranslatePlane* do componente de arrasto *Box* do manipulador.

Tabela 5-9 *Callbacks* da classe *Subvolume*.

finishCallback	Comunica à visualização de volume as coordenadas a serem utilizadas na geração do subvolume.
motionCallback	Não utilizada.
startCallback	Não utilizada.

6. Conclusões e Trabalhos Futuros

Com a crescente disponibilidade de *hardware* e *software* 3D, aplicações gráficas tridimensionais vem sendo desenvolvidas, com grande aceitação por diversas comunidades, dentre elas, a da área de exploração e produção de petróleo.

Como foi analisado na seção **2.1**, uma tendência forte é a construção de interfaces 3D, ou seja, interfaces que coexistem, em um mesmo ambiente, com os objetos da aplicação que elas controlam.

O aproveitamento completo do poder das interfaces 3D requer o uso de dispositivos de entrada que ofereçam controle sobre os graus de liberdade introduzidos pelas aplicações 3D, tornando a interação mais natural. Como o alto custo e os problemas na operação destes disposisitivos ainda não permitem sua utilização em larga escala, soluções para interação 3D com a utilização do *mouse* ainda são necessárias.

Widgets já são bastante utilizados em interfaces 2D e, nos últimos anos, *widgets* 3D começaram a ser experimentados em aplicações 3D. O cursor 3D, a translação gestual e a esfera virtual estão entre os primeiros *widgets* 3D implementados.

Outras experiências vêm sendo feitas com *widgets* 3D, tanto com a utilização de dispositivos 2D, como em ambientes virtuais. Na composição de cenas 3D, eles são utilizados para localizar, orientar, transladar e escalar objetos. Em modelagem geométrica, exemplos de sua utilização são: aplicar deformações sobre objetos e gerar curvas, superfícies e volumes a partir do movimento de um objeto geométrico. Em visualização científica, experiências foram feitas em ambientes virtuais.

Baseado nessas experiências, este trabalho mostrou as possibilidades de utilização de *widgets* 3D em sistemas de visualização e interpretação de dados de exploração e produção de óleo, com a utilização do *mouse*. Com o objetivo de prover o *toolkit* para desenvolvimento de aplicações interativas na área de E&P (seção **3.1**) de um ambiente extensível de edição das representações de dados (visualizações) com as quais ele lida, uma arquitetura para construção de interfaces interativas de manipulação (manipuladores e componentes de arrasto) e um modelo de interação foram implementados.

A utilização do *mouse* impediu a maior naturalidade da manipulação com os componentes de arrasto. Translações sobre linhas e planos e rotações sobre cilindros foram considerados movimentos simples, pois puderam ser mapeados diretamente a partir dos movimentos do *mouse*. O manipulador para movimentação de fatia é um exemplo de manipulação simples. Já os movimentos no espaço tiveram que ser decompostos. Para manipuladores como o de definição de subvolume, foi necessário estabelecer seu comportamento complexo como uma composição de movimentos simples.

O modelo de interação implementado libera a aplicação do tratamento dos eventos de manipulação, pois as técnicas de interação foram encapsuladas nos manipuladores e componentes de arrasto. O mecanismo de associação de funções *callbacks* aos manipuladores, pelas visualizações, permite a particularização de sua utilização em diferentes aplicações.

O trabalho continuará com a avaliação, junto aos usuários, das técnicas de interação adotadas e com a implementação de novos manipuladores. O conjunto de manipuladores implementados no protótipo contemplou necessidades atuais de manipulação de dados volumétricos. O enfoque de orientação a objetos utilizado permite que novos manipuladores,

componentes de arrasto, projetores e formas geométricas sejam implementados, à medida que forem demandados pelas aplicações.

Este trabalho de avaliação já está sendo executado. Uma aplicação para estudo de velocidades a utilizar em um processo aplicado sobre dados sísmicos 3D, chamado migração sísmica, está sendo desenvolvida e está utilizando o manipulador para movimentação de fatia de volume na etapa de seleção das fatias do volume sísmico a serem analisadas.

Dois manipuladores já estão em fase de projeto. Um deles será utilizado para realçar áreas de interesse do volume, através da edição seletiva da cor e opacidade das amplitudes sísmicas. A idéia é que a resposta às edições seja agilizada pela seleção de um subconjunto do volume, para refletir as mudanças. Definidos os valores finais, todo o volume será atualizado. O outro manipulador será utilizado para definir direções arbitrárias para as visualizações do tipo fatia de volume e deverá explorar movimentos de rotação.

Outra linha de trabalho a ser atacada é a implementação de meios para aumentar a velocidade do *rendering* das visualizações, com o objetivo de manter um *feedback* durante as manipulações, sem restringir a interatividade. Multiresolução, refinamento incremental e aproximações são opções a serem utilizadas. O recurso de desenhar as visualizações parcialmente, durante sua manipulação, também poderá ser utilizado. O desenho é iniciado e, dependendo da velocidade das manipulações, ele fica mais, ou menos, completo.

Embora a motivação principal deste trabalho tenha sido proporcionar a visualização interativa de dados sísmicos volumétricos, espera-se que aplicações que lidem com outros tipos de representações se beneficiem da arquitetura de manipulação interativa proposta.

Referências Bibliográficas

G. Booch, *Object-oriented Analysis and Design with applications*, The Benjamin/Cummings Publishing Company, Inc., 2nd edition, 1994.

E. A. Bier, *Snap-Dragging in Three Dimensions*, Proceedings of the ACM SIGGRAPH, Computer Graphics, vol. 24, no. 4, pp. 193-204, 1990.

D. B. Conner, S. S. Snibbe, K. P. Herndon, D. C. Robbins, R. C. Zeleznik, A. van Dam, *Three-Dimensional Widgets*, Proceedings of the 1992 Workshop on Interactive 3D Graphics, pp. 183-188, 1992.

J. M. Cychosz, W. N. Waggenspack, Jr., *Intersecting a Ray with a Cylinder*, Graphics Gems IV, edited by Paul S. Heckbert, Academics Press Inc, 1994.

R. Goldman, *Intersection of Two Lines in Three-Space*, Graphics Gems, edited by Andrew Glassner, Academics Press Inc, 1990.

C. Grimm, D. Pugmire, M. Bloomenthal, J. Hughes, E. Cohen, *Visual Interfaces for Solid Modeling*, Proceedings of UIST '95, ACM Press, November, 1995.

J. Hartman, J. Wernecke, *The VRML 2.0 Handbook*, Addison Wesley, 1996.

L. Hatton, M. H. Worthington, J. Makin, *Seismic Data Processing: Thory and Practice*, Oxford, Blackwell Scientific Publications, 1986.

K. P. Herndon, R. C. Zeleznik, D. C. Robbins, D. B. Conner, S. S. Snibbe, A. van Dam, *Interactive Shadows*, Proceedings of the 1992 Symposium on User Interface Software and Technology, pp. 1-6, 1992.

K. P. Herndon, A. van Dam, M. Gleicher, *Workshop on the Challenges of 3D Interaction, SIGCHI Bulletin*, vol. 26, no. 4, pp. 1-9, 1994.

S. Houde, *Iterative Design of an Interface for Easy 3D Direct Manipulation*, Proceedings of CHI'92, pp. 135-142, 1992.

J. Neider, T. Davis, M. Woo, *OpenGL Programming Guide - The Official Guide to Learning OpenGL*, Release 1, Addison Wesley, 1991.

T. Meyer, A. Globus, *Direct Manipulation of Isosurfaces and Cutting Planes in Virtual Environments*, Brown University, Technical Report, 1993.

L. P. Reis, *A Toolkit fot Interactive Scientific Visualization*, Manuscrito, 1996.

E. Robinson, S. Treitel, *Geophysical Signal Analysis*, Englewoods Cliffs, Prentice-Hall, 1980.

S. S. Snibbe, K. P. Herndon, D. C. Robbins, D. B. Conner, A. van Dam, *Using Deformations to Explore 3D Widgets Design*, Computer Graphics (Proceedings SIGGRAPH'92) , vol. 26, no. 2, pp. 351-352, 1992.

M. P. Stevens, R. C. Zeleznik, J. F. Hughes, *An Architecture for an Extensible 3D Interface Toolkit*, Brown University, 1994.

W. Schroeder, H. Martin, B. Lorensen, *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*, Prentice Hall, 1996.

R. R. Springmeyer, M. M. Blattner, N. L. Max, *A Characterization of the Scientific Data Analysis Process*, Visualization, pp. 235-242, 1992.

P. S. Strauss, R. Carey, *An Object-Oriented 3D Graphics Toolkit*, Computer Graphics (Proceedings SIGGRAPH'92), vol. 26, no. 2, pp. 341-349, 1992.

R. Uchida, *Seeing Beneath the Surface*, Iris Universe, The Magazine of Visual Computing, no. 20, pp. 10-14, 1994.

C. Upson, T. Faulhaber, Jr., D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, A. van Dam, *The Application Visualization System: A Computational Environment for Scientific Visualization*, IEEE Computer Graphics & Applications, pp. 30-42, 1989.

D. Venolia, *Facile 3D Direct Manipulation*, Proceedings of INTERCHI'93, pp. 31-36, 1993.

J. Wernecke, *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor and The Inventor Toolmaker: Extending Open Inventor*, Release 2, Addison Wesley, 1994.

R. C. Zeleznik, K. P. Herndon, D. C. Robbins, N. Huang, T. Meyer, N. Parker, J. F. Hughes, *An Interactive 3D Toolkit for Constructing 3D Widgets*, Computer Graphics (SIGGRAPH'93 video paper), vol. 27, pp. 81-84, 1993.

"Uma Arquitetura para Construção de Ferramentas de Manipulação para Visualização Interativa de Dados Volumétricos". Dissertação de Mestrado apresentada por LÚCIA TERESA SCHALCHER DA FONSECA em 23 de setembro de 1997 ao Departamento de Informática da PUC-Rio e aprovada pela Comissão Julgadora, formada pelos seguintes professores:

Prof. Luiz Fernando Martha (Orientador)
Departamento de Engenharia Civil / PUC-Rio

Prof. Marcelo Gattass (Co-Orientador)
Departamento de Informática / PUC-Rio

Prof. Bruno Feijó
Departamento de Informática / PUC-Rio

Visto e permitida a impressão,

Rio de Janeiro, 06/11/97

Profa. Maria Augusta M. Davidovich
Coordenadora dos Programas de Pós-Graduação
do Centro Técnico Científico