Implementação de uma estrutura de dados para visualização científica

Carlos André Sanches de Souza

Orientadora: *Profa. Dra. Rosane Minghim*

Dissertação apresentada ao Instituto de Ciências Matemática e de Computação – ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências de Computação e Matemática Computacional.

USP – São Carlos
Fevereiro/2003

À minha família: minha mãe Vera,
meu pai José, meus irmãos Victor e
Leandra e minha namorada
Alessandra.

Agradecimentos

À Profa. Dra. Rosane Minghim pela paciência e dedicação durante sua orientação.

Aos professores doutores Antônio Castelo Filho, Maria Cristina Ferreira de Oliveira e Luis Gustavo Nonato e ao colega Helton Hideraldo Bísaro por estarem sempre dispostos a ajudar.

A todos os meus colegas do ICMC pelos momentos de descontração.

Aos meus pais por sempre terem respeitado e apoiado as minhas decisões

Aos meus irmãos.

À minha namorada Alessandra, pela paciência, compreensão, amor e carinho dedicados durante todos esses anos e que continuará assim por toda a vida.

A Capes pelo apoio financeiro.

A Deus pela saúde, força e oportunidade.

Sumário

LISTA DE FIGURAS	3
RESUMO.....	4
ABSTRACT	5
CAPÍTULO 1 – INTRODUÇÃO.....	6
1.1. CONSIDERAÇÕES INICIAIS	6
1.2. PROJETO DE PESQUISA.....	7
1.3. ORGANIZAÇÃO DA DISSERTAÇÃO.....	8
CAPÍTULO 2 – VISUALIZAÇÃO CIENTÍFICA E VTK	9
2.1. CONSIDERAÇÕES INICIAIS	9
2.2. VISÃO GERAL.....	9
2.3. CONJUNTOS DE DADOS.....	11
2.4. ORGANIZAÇÃO DO VOLUME DE DADOS	13
2.5. A BIBLIOTECA VTK	13
2.6. EXECUÇÃO DA REDE OU <i>PIPELINE</i> DE VISUALIZAÇÃO.....	15
2.6.1. <i>Execução Explícita</i>	16
2.6.2. <i>Execução Implícita</i>	17
2.7. MODELOS DE OBJETOS	18
2.8. A ESTRUTURA DE CLASSES DO VTK.....	19
CAPÍTULO 3 – CONCEITOS FUNDAMENTAIS E ESTRUTURAS DE DADOS TOPOLÓGICAS ..	23
3.1. CONSIDERAÇÕES INICIAIS	23
3.2. DEFINIÇÕES.....	23
3.2.1. <i>Variedade</i>	23
3.2.2. <i>Fecho Convexo</i>	24
3.2.3. <i>Decomposição Celular</i>	25
3.2.4. <i>Complexos Simpliciais</i>	26
3.2.5. <i>Estrutura de Dados</i>	26
3.3. ESTRUTURAS TOPOLÓGICAS	27
CAPÍTULO 4 – A ESTRUTURA SINGULAR HALF-FACE (SHF).....	32
4.1. CONSIDERAÇÕES INICIAIS	32
4.2. A ESTRUTURA SHF	32
4.3. OPERADORES E CONVERSORES	36
4.4. COLAGEM E DESCOLAGEM DE CÉLULAS	37
4.5. OPERADORES TOPOLÓGICOS	39
4.6. CONVERSORES	41
4.6.1. <i>Conversor para o Formato HLF</i>	41
4.6.2. <i>Conversor para o Formato Unstructured Grid</i>	43
4.6.3. <i>Conversor para o Formato 0-1</i>	45
4.6.4. <i>Conversores de Exportação</i>	45
CAPÍTULO 5 – IMPLEMENTAÇÕES REALIZADAS.....	47
5.1. CONSIDERAÇÕES INICIAIS	47
5.2. O CÓDIGO FONTE VTK.....	47
5.3. NOVAS CLASSES NO VTK.....	48
5.4. PRODUZINDO UM NOVO CONJUNTO DE DADOS	49
5.5. IMPLEMENTAÇÕES	53

5.6. FILTROS	54
5.7. FONTES	55
5.8. MAPEADORES	56
5.9. EXEMPLOS DE CÓDIGOS	58
CAPÍTULO 6 – APLICAÇÃO: RECONSTRUÇÃO 3D	63
6.1. CONSIDERAÇÕES INICIAIS	63
6.2. MÓDULO DE RECONSTRUÇÃO 3D	63
6.3. EXEMPLOS DE CÓDIGOS	71
CAPÍTULO 7 – CONCLUSÕES	74
7.1. CONSIDERAÇÕES INICIAIS	74
7.2. CONTRIBUIÇÕES DO TRABALHO	74
7.3. SUGESTÃO PARA TRABALHOS FUTUROS	75
APÊNDICE – DESCRIÇÃO DOS ARQUIVOS E DAS CLASSES	76
REFERÊNCIAS BIBLIOGRÁFICAS	86

Lista de Figuras

Figura 2.1: (a) Malha regular volumétrica, (b) malha de voxels, (c) malha de células.....	12
Figura 2.2: Tipos de malhas: (a) Cartesiana, (b) Regular, (c) Retilínea, (d) Estruturada, (e) Desestruturada.....	13
Figura 2.3: Modelo VTK. Objetos de processos A, B e C são objetos fonte, filtro e mapeador, respectivamente.....	14
Figura 2.4: Execução da rede. Ramificações paralelas não precisam executar se mudanças são locais para uma particular ramificação. Adaptada de [Sch98].....	16
Figura 2.5: Execução da rede explícita e implícita. Adaptada de [Sch98].....	17
Figura 2.6: Tipos de conjuntos de dados (datasets) do VTK. A “Malha Não Estruturada” consiste de todos os tipos de célula. Adaptada de [Sch98].....	20
Figura 2.7: Diagrama de objetos dos tipos de dados do VTK herdeiros de vtkDataSet [Sch98].....	21
Figura 3.1: (a) Variedade, (b) Não Variedade onde “P” é um ponto singular.	24
Figura 3.2. Representação Half-Edge.....	29
Figura 4.1: Visão geral da organização da estrutura de dados SHF. Adaptada de [Bís00].	33
Figura 4.2: Estrela de um vértice.	34
Figura 4.3: (a) Semi-face formada por 3 semi-arestas, (b) Mate de uma face. Adaptada de [Bís00].	34
Figura 4.4: (a) Mate-Célula, (b) Mate-Aresta. Adaptada de [Bís00].	35
Figura 4.5: Mate-aresta no bordo. Adaptada de [Bís00].	36
Figura 4.6: Mate-aresta no bordo de células vizinhas, compartilhando uma aresta singular.	38
Figura 4.7: Inserção de um tetraedro: (a) gera uma cavidade (M3EP), (b) gera um buraco (MVEH).....	40
Figura 4.8: Formato VTK: (a) conteúdo de um arquivo com 2 células e 10 faces, (b) exemplos de objetos com singularidades de aresta, (c) mais exemplos de objetos nesse formato.	42
Figura 4.9: Exemplo de um arquivo no formato unstructured grid.	44
Figura 4.10: Exemplo de um arquivo no formato 0-1.....	45
Figura 5.1: Classe responsável pelo sólido da estrutura SHF, “Csolid”, anexada ao VTK.	50
Figura 5.2 – Exemplo de uma célula hexaedro. A topologia é implicitamente definida pela ordem da lista de pontos. Adaptada de [Sch98].....	52
Figura 5.3: Visão geral da estrutura SHF incorporada ao VTK.	58
Figura 5.4: (a) Tetraedro gerado pelos dois primeiros algoritmos, (b) visão de arame do tetraedro, (c) sólido gerado pelo terceiro algoritmo com representação de escalares por cores.....	62
Figura 6.1: Tabela de formatos para arquivos PNM.....	64
Figura 6.2: Duas fatias vizinhas. Os pixels da imagem (valor 0) são representados por pontos pretos.	65
Figura 6.3: Exemplo de uma malha de cubos inserida entre duas fatias, com RefFactor igual a 5. A cada 5 pixels é colocado um vértice do cubo.....	66
Figura 6.4: Cubo formado por 6 tetraedros. As cores indicam cada lado do cubo.....	66
Figura 6.5: Coração reconstruído.....	67
Figura 6.6: (a) Reconstrução de um dente, (b) reconstrução de um mamão, (c) DVR dos dois objetos.....	68
Figura 6.7: Tabela de parâmetros e tipos de arquivo fonte.....	69
Figura 6.8: Mamão reconstruído sem cavidades na visão de arame.....	70
Figura 6.9: Módulos do reconstrutor.....	70
Figura 6.10: Coração após filtro de alisamento.	71

Resumo

Estruturas de dados volumétricas são de extrema utilidade em várias aplicações, e em particular na área de Visualização Científica. Essas estruturas são úteis em duas etapas do processo de visualização. A primeira é na representação de dados, isto é, de informações associadas aos valores, medidos ou simulados, os quais se deseja visualizar. A outra fase que necessita de uma estrutura de dados é a fase de exploração, ou seja, o modelo criado é usado tanto para exploração interativa quanto para a realização de simulações sobre o mesmo, por exemplo, numa cirurgia virtual.

Está em desenvolvimento no ICMC, uma estrutura de dados volumétrica, chamada Singular Half-Face, que tem como característica a modelagem explícita das singularidades presentes no modelo, além de outros elementos topológicos. Este trabalho de mestrado visa testar a viabilidade da estrutura num contexto de visualização em fluxo de dados, incorporando-a à biblioteca gráfica de visualização *Visualization ToolKit* (VTK), que possui definição extremamente pobre da topologia dos objetos que representa. Adicionando essa nova classe e realizando sobre ela técnicas convencionais de visualização e exploração de dados, é estudada sua capacidade de apoiar todas as fases do processo de visualização.

Abstract

Volumetric data structures are of extreme utility in various applications, and particularly in the area of Scientific Visualization. These structures are useful in two stages of the visualization process. The first one is in data representation, that is, in the organization of information associated with the values, measured or simulated, to be visualized. The other phase of the visualization data flow that needs a data structure is the exploration phase. It would be useful to have a model designed to be used both for interactive exploration and for simulation in a number of application, for example, in virtual surgery.

A volumetric data structure, named Singular Half-Face (SHF) is being developed at the ICMC, that has as its main characteristic the explicit modeling of singularities present in the model, besides other essential topological elements. This dissertation aims at testing the viability of this structure in a context of visualization in data flow, by incorporating it in the *Visualization ToolKit* (VTK) class library, whose data structures are extremely poor in definition of the topology of the objects that they represent. By adding SHF to this library and carrying out conventional visualization and data exploration on it, we wish to study its support to all the phases of the visualization process.

Capítulo 1 – Introdução

1.1. Considerações Iniciais

Na visualização científica aplicada a áreas como medicina existe uma forte tendência em trabalhar com imagens obtidas a partir de dados escalares, geradas por técnicas de *Rendering* Volumétrico Direto (*Direct Volume Rendering* - DVR). Esse tipo de técnica gera imagens diretamente a partir de conjuntos de dados provenientes de medições ou simulações, sem que nenhuma estrutura geométrica intermediária (como superfícies, sólidos, malhas) seja construída no processo. Uma desvantagem dessa técnica é que um grande conjunto de dados precisa ser percorrido para gerar cada imagem renderizada. A visualização por DVR tende, entretanto, a ser mais efetiva e poderosa se empregada em conjunto com as técnicas tradicionais que trabalham com reconstrução de superfícies usando primitivas poligonais (como triângulos), ou com reconstrução de volumes usando tetraedros.

Objetos reconstruídos se adaptam mais facilmente às filosofias de interação hoje presentes em ambientes tridimensionais, como aquelas advindas da Realidade Virtual, uma vez que atualmente é possível fazer *rendering* de malhas de primitivas em tempo real, mesmo em ambientes altamente interativos. Já no caso do DVR, tanto o aspecto da velocidade de apresentação das imagens quanto a ausência de uma abordagem consistente para apoiar a interação com o volume renderizado dificultam a integração desta técnica com a atual tecnologia de interface com o usuário. Adicionalmente, a ausência de um modelo geométrico dos objetos visualizados por DVR impede a utilização desses modelos em simulações numéricas, por exemplo.

Na linha de frente da integração entre DVR e reconstrução está um problema ainda sem solução, que é a obtenção de um modelo que possa ser usado na interação e exploração de dados, e ao mesmo tempo permita a indexação tanto dos elementos volumétricos da imagem resultante do DVR (unidades de volumes chamadas voxels), como dos elementos geométricos que compõem os objetos reconstruídos (tanto superficiais como volumétricos). Em outras palavras, quando submetido ao problema de integrar, em um sistema único e interativo, técnicas de *rendering* volumétrico direto e por superfície, o pesquisador se

depara com a ausência de um modelo específico para esse fim. Uma análise mais detalhada da situação nos leva a concluir que a falha se encontra na ausência de um modelo de dados apropriado, que considere tanto o conjunto de dados original, transformado em uma imagem volumétrica pelo método DVR, quanto as superfícies e volumes reconstruídos a partir dele. Ao ambiente 'ideal', que consegue integrar a apresentação pelas duas classes de técnicas diversas de mapeamento gráfico, é dado o nome de **ambiente híbrido de visualização**. Ao modelo de dados responsável por fornecer um arcabouço que possibilite a construção desse ambiente damos o nome de **modelos de dados híbrido**. A falta de um modelo ocorre, principalmente, porque a visualização, até há bem pouco tempo, era considerada uma etapa final do processo de entendimento dos dados, ao invés de uma etapa intermediária, tendo a interação com os modelos como objetivo.

1.2. Projeto de Pesquisa

No contexto da solução do problema de modelagem de dados em visualização híbrida, está em desenvolvimento no ICMC um projeto que envolve a descrição e implementação de representações de dados científicos para armazenamento e manipulação em memória, tanto aqueles advindos de simulações e medições, quanto os criados durante o processo de visualização, como geometria, topologia e informação de iluminação relativas ao volume de dados. Como ponto de partida para a implementação de tal modelo foi proposta por Bísaro uma estrutura de dados chamada Singular Half-Face (SHF) [Bis01], que tem a característica de modelar explicitamente, entre outros elementos topológicos, as singularidades presentes no modelo. Ela deve ser estendida, no futuro, para trabalhar com informações de *Rendering* Volumétrico.

Essa estrutura, implementada em C++ na forma de uma estrutura de dados e procedimentos para sua manipulação, ainda é limitada para representação de um modelo híbrido em visualização. É preciso aprimorar tanto a sua implementação quanto a sua usabilidade antes que ela possa evoluir para representar todos os elementos necessários em um ambiente híbrido. O propósito do presente trabalho de mestrado é realizar a próxima etapa de evolução da SHF, de forma a torná-la adequada para servir de base para vários outros trabalhos de pesquisa no ICMC. Pretende-se validar o uso da estrutura no contexto

da visualização científica, incorporando-a a biblioteca gráfica *Visualization ToolKit* (VTK) [Sch98][Sch00], que possui definição extremamente pobre da topologia dos objetos que representa, o que a torna limitada para a implementação de certas aplicações de visualização.

Assim, esta dissertação, além de apresentar a SHF e os conceitos relacionados com o seu desenvolvimento, apresenta a implementação da estrutura no ambiente VTK, discutindo aspectos importantes da extensão da biblioteca para manipular um novo conjunto de dados. Adicionalmente é apresentada uma aplicação da estrutura e de sua implementação para reconstrução de objetos 3D. Essa reconstrução também foi incorporada como um elemento da biblioteca VTK.

1.3. Organização da Dissertação

Esta dissertação está organizada da seguinte forma:

O Capítulo 2 apresenta uma visão geral sobre a área de visualização computacional, seguida por uma apresentação do software VTK.

O Capítulo 3 contém alguns conceitos básicos necessários para o entendimento da estrutura SHF. Também são revistas diversas estruturas de dados topológicas existentes anteriormente à SHF.

O Capítulo 4 apresenta uma visão geral sobre a estrutura SHF, descrevendo o funcionamento de seus operadores topológicos e dos conversores que importam dados de diferentes formatos para essa estrutura.

O Capítulo 5 descreve as implementações realizadas para incorporar a nova estrutura na biblioteca VTK.

O Capítulo 6 apresenta uma aplicação de reconstrução tridimensional desenvolvida para testar a nova estrutura no contexto da biblioteca.

Finalizando, o Capítulo 7 apresenta a conclusão deste trabalho.

Capítulo 2 – Visualização Científica e VTK

2.1. Considerações Iniciais

A visualização científica tem sido uma importante área de pesquisa nos últimos anos. A forma de visualização mais utilizada é a visualização volumétrica. Essa forma de visualização foi primeiramente apresentada em 1988, como uma técnica de visualização de superfícies a partir de dados volumétricos. Sua idéia era desenvolver uma visualização eficiente e precisa, em que se conseguisse “sintetizar” todas as informações contidas em um conjunto de dados volumétricos em uma única imagem, de forma que se tivesse a impressão de estar olhando para os dados reais [Sei96]. Com o passar do tempo, novas técnicas foram desenvolvidas para apoiar a enorme quantidade de informações que são geradas por diversas áreas da ciência e permitir a interação e a realização de simulações com esses dados.

Este capítulo apresenta uma visão geral sobre a área de visualização computacional, seguido por uma apresentação do VTK, a biblioteca de visualização utilizada neste trabalho. O objetivo é fornecer subsídios para utilização dos conceitos apresentados em capítulos subseqüentes.

2.2. Visão Geral

O ser humano possui uma alta capacidade de abstrair informações através de seu primeiro sentido: a visão. Uma imagem de dados consegue enviar ao cérebro humano grande quantidade de informações com eficiência superior a qualquer outro método de apresentação de dados.

Partindo desse princípio, podemos definir **visualização** como a transformação de dados ou informações em imagens gráficas, de forma a facilitar o seu entendimento.

A utilização do computador no processo de visualização tornou-se necessária, devido à grande quantidade de informações disponíveis atualmente para serem interpretadas e à escassez de tempo e de recursos humanos para fazê-la, o que deu início a

uma nova área da computação, denominada **visualização computacional**. Visualização de dados é outro termo utilizado para descrever a visualização computacional.

Muitas áreas de atuação humana que envolvem computação têm sido beneficiadas com a produção de visualizações, particularmente a área científica. A aplicação de técnicas gráficas para ampliar a capacidade de interpretação de dados científicos tem sido denominada **visualização científica** [Ear97]. Normalmente, essa área trabalha com conjuntos de dados obtidos através de simulações ou dispositivos de medição.

Uma das subáreas mais ativas da visualização científica, que tem tido um rápido crescimento, é a **visualização volumétrica**. A visualização volumétrica oferece técnicas para manipulação de dados presente em malhas multidimensionais e apresentação em um espaço tridimensional (3D) [Elv92]. Com a crescente utilização de dispositivos precisos, em especial na medicina, como a tomografia computadorizada (TC), ressonância magnética (RM), e outros, as pesquisas nessa área têm alcançado patamares progressivos de maturidade.

As técnicas de visualização volumétrica podem ser classificadas em dois grandes grupos: *Rendering Volumétrico Direto* (*Direct Volume Rendering* - DVR) e reconstrução. *Rendering* (ou Renderização) é o processo de geração de uma imagem a partir de uma descrição computacional de uma cena composta por um ou mais modelos.

Técnicas de ***Rendering Volumétrico Direto*** geram imagens tridimensionais dos objetos volumétricos diretamente a partir de dados, sem a utilização de uma estrutura geométrica intermediária. Para os algoritmos que trabalham com essas técnicas, o volume é representado por uma grade de paralelepípedos alinhados com os eixos, denominados *voxels* (abreviação de “*volume element*”, ou elemento do volume, que segue a mesma linha de abreviação de *pixel* para “*picture element*”, ou elemento da figura). Para cada *voxel*, é atribuído um valor de densidade do material correspondente. A essa densidade, é associada uma cor e uma opacidade, de acordo com uma tabela definida pelo usuário, o que permite a visualização de estruturas internas do volume. Um dos algoritmos mais utilizados nessa categoria é o *Ray Casting*.

A idéia básica do algoritmo de *Ray Casting* é o lançamento de raios que partem da posição do observador, através de cada *pixel* do plano de visualização, em direção ao

volume de dados. Para cada raio, integram-se as contribuições dos *voxels* interceptados por ele, determinando assim a cor do *pixel*.

As técnicas de **reconstrução** envolvem a extração e a representação de objetos presentes nos dados através do ajuste de primitivas geométricas sobre as informações de interesse do volume. O resultado é uma malha (superficial ou volumétrica) de elementos, que é posteriormente visualizada através da utilização de técnicas de computação gráfica (CG). Uma vantagem dessas técnicas é a possibilidade de identificação de estruturas nos processos de medições ou simulações. Elas também permitem análise em tempo real pois a renderização ser muito rápida, principalmente se feita por *hardware*. Entre as desvantagens, tem-se a perda da informação relativa ao conteúdo do conjunto de dados fora dos valores selecionados para o traçado. Além disso, nem sempre os dados podem ser descritos em termos de objetos, como é o caso dos entes amorfos (nuvens, fumaças, etc.). Quando a reconstrução é superficial também existe perda da informação de interior dos objetos de interesse.

Os métodos para o ajuste das primitivas geométricas incluem métodos de conexão de contornos e métodos de intersecção de *voxels*:

Métodos de conexão de contorno envolvem a identificação de contornos por um valor de interesse em fatias do volume de dados (planos que cortam o volume). Em seguida, as curvas em fatias adjacentes são conectadas em uma abordagem denominada tecelagem, triangulação ou tetraedralização.

Técnicas de intersecção de *voxels* operam em cada elemento do volume de dados procurando fazer uma aproximação do objeto de interesse naquele *voxel*. Essas aproximações são posteriormente conectadas para formar o modelo do objeto.

2.3. Conjuntos de Dados

Os dados em visualização científica podem ser definidos sobre domínios 2D, 3D e, genericamente, multidimensionais. Geralmente, um conjunto de dados possui uma certa organização como uma malha (ou grade) de células. As **células** também são elementos de volumes unitários, porém os dados estão posicionados nos seus vértices, enquanto que, nos

voxels, os dados estão armazenados em seu interior. Valores podem ser obtidos no interior de células através de interpolação. (figura 2.1).

Figura 2.1: (a) Malha regular volumétrica, (b) malha de voxels, (c) malha de células.

Os dados também podem se constituir de um conjunto de pontos esparsos ou malhas geométricas de outra natureza, como as descritas na seção 2.4. Além disso, as informações associadas a cada ponto da malha podem ser um ou mais valores escalares, vetoriais ou tensoriais [Min97].

Um conjunto de dados consiste de duas partes: uma estrutura de organização e atributos de dados complementares associados à estrutura. A estrutura tem duas partes: uma topologia e uma geometria. A topologia é um conjunto de propriedades invariantes sob certas transformações geométricas (como por exemplo, rotação e translação). A geometria é a instanciação da topologia, ou seja, a especificação da posição no espaço 3D. Por exemplo, quando declaramos que um polígono é um “triângulo”, estamos especificando a topologia. E quando fornecemos as coordenadas dos pontos, estamos especificando a geometria. Os atributos, por sua vez, são informações complementares que podem estar associadas tanto à geometria quanto à topologia [Sch98].

2.4. Organização do Volume de Dados

O volume de dados representa o trecho do espaço 3D onde se localiza o objeto, dado ou cena a ser renderizada. Uma malha faz referência à organização do volume de dados. Existem diferentes tipos de malhas: Em uma malha **cartesiana** todos os elementos são quadrados (no caso 2D) ou cubos idênticos (no caso 3D), alinhados aos eixos principais. É a forma ideal de estrutura, pois permite acesso mais rápido aos dados. Uma malha **regular** tem todos os seus elementos idênticos e alinhados aos eixos, mas esses elementos são retângulos ou paralelepípedos de dimensões constantes. Esse tipo de estrutura é muito comum em dados provenientes de Tomografia Computadorizada. Elementos de uma malha **retilínea** são quadriláteros ou hexaedros alinhados aos eixos, mas não necessariamente idênticos. Elementos de uma malha **estruturada** são quadriláteros ou hexaedros não alinhados aos eixos principais, como os que aparecem em grades esféricas ou curvilíneas. Uma malha **estruturada em blocos** é um conjunto de malhas estruturadas agrupadas. Uma malha **desestruturada** contém polígonos ou poliedros sem qualquer padrão explícito de conectividade. No caso 3D, as células podem ser tetraedros, hexaedros, pirâmides, etc. Uma malha **híbrida** é uma combinação de quaisquer dos tipos anteriores [Soa00] (figura 2.2).

Figura 2.2: Tipos de malhas: (a) Cartesiana, (b) Regular, (c) Retilínea, (d) Estruturada, (e) Desestruturada.

2.5. A Biblioteca VTK

O *Visualization ToolKit* é um sistema de software de código aberto para computação gráfica 3D, processamento de imagens, e visualização, usado por milhares de pesquisadores e desenvolvedores em todo o mundo. Ele consiste de uma biblioteca de classes em C++ e várias camadas de interfaces interpretadas, incluindo Tcl/Tk, Java, e

Python, e suporta uma ampla variedade de algoritmos de visualização para diferentes tipos de dados. Também são implementadas técnicas de modelagem avançadas como modelagem implícita, redução de polígonos, suavização de malhas e triangulação de Delaunay. O projeto e a implementação da biblioteca foram fortemente influenciados pelos princípios de orientação a objetos, e o software foi instalado e testado em quase todas as plataformas baseadas em Unix e Windows.

O modelo de programação do VTK adota o paradigma de fluxo de dados. Nesse paradigma, módulos são conectados para formar uma rede que descreve um canal (*pipeline*) de processamento de dados. Os módulos executam operações algorítmicas sobre os dados enquanto eles fluem pelo *pipeline*. No modelo VTK há dois tipos básicos de objetos: objetos de processos e objetos de dados. Objetos de processo são os módulos do *pipeline*, e objetos de dados representam os dados nos vários estágios no *pipeline*. Essencialmente, o desenvolvedor está ciente apenas dos objetos de processos, os objetos de dados “vivem” entre os objetos de processo.

Figura 2.3: Modelo VTK. Objetos de processos A, B e C são objetos fonte, filtro e mapeador, respectivamente.

Objetos de processo são de um dos três tipos: fontes, filtros e mapeadores (ver figura 2.3). Objetos fontes são encontrados no início do *pipeline*, e geram uma ou mais saídas de dados. Um objeto fonte pode ser um leitor para um tipo de arquivo particular ou pode gerar seu próprio dado, como o objeto *vtkSphereSource*, que cria uma representação (objeto de dados) de uma esfera poligonal. A saída desse objeto fonte pode então ser conectada à entrada de outro objeto de processo. O ato de conectar a entrada de um objeto de processo à saída de outro objeto de processo define como o *pipeline* está construído. Por exemplo, para conectar a saída do filtro A à entrada do filtro B, é usada a seguinte construção:

```
B->SetInput( A->GetOutput( ) );
```

Como conexões podem ser múltiplas (uma saída pode ser entrada de vários objetos), as conexões podem, de fato, se tornar uma rede (*network*) de visualização, ao invés de um *pipeline* simples.

Cada objeto de processo gera apenas uma saída.

O *pipeline* ou rede termina com mapeadores. Um mapeador mapeia a sua entrada para a tela (*render*). Nesse estágio do *pipeline*, a entrada para o mapeador é a saída de algum filtro que gera dados geométricos (*vtkPolyData*). Dados geométricos (*vtkPolyData*) são objetos de dados como qualquer outro que representam malhas planares irregulares. De fato, há filtros que trabalham com dados geométricos, como algoritmos de triangulação (*vtkTriangleFilter*).

O mapeador é associado a um objeto chamado *vtkActor*, que representa um objeto geométrico e seus atributos na cena. Informações em um *vtkActor* incluem os atributos de aparência do objeto (*vtkProperty*) e sua localização no espaço. Como resultado, o desenvolvedor instancia pelo menos um *vtkActor* para cada mapeador no *pipeline* do VTK, e um mapeador para cada objeto geométrico que deverá ser renderizado. O desenvolvedor ajusta cada atributo de cada *vtkActor* e adiciona cada *vtkActor* a uma janela de renderização, usada por uma instância da classe de *vtkRenderWindow*. A janela resultante exibe todos os atores a ela associados.

Uma das principais vantagens do VTK são seus recursos para interação com os gráficos 3D gerados. Tais recursos possibilitam que o usuário “navegue” através dessas imagens, e, conseqüentemente, através dos dados. A classe do VTK chamada *vtkRenderWindowInteractor* é a responsável por transformar eventos do *mouse* e teclado em modificações aos atores e câmeras que compõem a visualização. Operações como rotação, translação, escala e “trackball” estão implementadas, entre outras.

2.6. Execução da Rede ou *Pipeline* de Visualização

Para ser útil, uma rede de visualização precisa processar dados para gerar um resultado desejado. O processo completo de induzir cada objeto de processo a atuar é chamado de *execução* da rede.

Frequentemente a rede de visualização é executada mais de uma vez. Por exemplo, pode haver mudanças nos parâmetros ou na entrada de um objeto de processo. Isto é tipicamente devido à interação com o usuário: o usuário pode estar explorando ou metodicamente variando a entrada para observar resultados. Depois de uma ou mais mudanças no objeto de processo ou na sua entrada, é preciso executar a rede para gerar resultados atualizados.

Para uma alta performance, os objetos de processos na rede de visualização precisam executar *apenas* se uma mudança ocorrer em suas entradas. Em algumas redes, como mostrada na figura 2.4, podem existir ramificações paralelas que não precisam ser executadas se os objetos são modificados localmente em uma particular ramificação. Nessa figura, o objeto D e os objetos receptores E e F precisam executar porque um parâmetro de entrada de D foi alterado, e os objetos E e F dependem de D para suas entradas. Os outros objetos não precisam ser executados porque não há mudanças em suas entradas.

A execução da rede requer sincronização entre os objetos de processo. Há dois modos de sincronizar a execução: controle explícito ou implícito (figura 2.5).

Figura 2.4: Execução da rede. Ramificações paralelas não precisam executar se mudanças são locais para uma particular ramificação. Adaptada de [Sch98].

2.6.1. Execução Explícita

Controle explícito significa trilhar diretamente as mudanças na rede, e então controlar diretamente a execução dos objetos de processos baseados em uma análise de dependência explícita. A maior característica desse método é que um *executivo* centralizado é usado para coordenar a execução da rede. Esse executivo precisa trilhar mudanças nos

parâmetros e entrada de cada objeto, incluindo mudanças subseqüentes na topologia da rede. (figura 2.5(a)).

A vantagem desse método é que análise de sincronização e métodos de atualização são locais para um único objeto executivo. Também é possível criar gráficos de dependência e análise de performance do fluxo de dados cada vez que a saída é requerida. Essa capacidade é particularmente importante no caso de decompor a rede para computação paralela ou distribuir execução através de uma rede de computadores. A desvantagem do método explícito é que cada objeto de processo se torna dependente do executivo, já que o executivo precisa ser notificado de qualquer mudança.

Figura 2.5: Execução da rede explícita e implícita. Adaptada de [Sch98].

2.6.2. Execução Implícita

Controle implícito significa que um objeto de processo executa somente se sua entrada local ou parâmetros mudam (figura 2.5(b)). Controle implícito é implementado usando um processo de dois passos. Primeiro, quando a saída de um objeto é solicitada por um outro objeto, aquele objeto solicita seus objetos de entrada. Esse processo é recursivamente repetido até que os objetos fontes sejam encontrados. O objeto fonte então executa se eles foram alterados ou se suas entradas externas foram alterados. Então a

recursão segue onde cada objeto de processo examina suas entradas e determina se irá executar. Esse procedimento repete até o objeto que pediu inicialmente executa e termina o processo. Esses dois passos são chamados de passos de *atualização* e *execução*.

A principal vantagem do esquema de controle implícito é sua simplicidade. Cada objeto apenas precisa manter uma trilha de seu tempo de modificação interna. Quando a saída é requerida, o objeto compara seu tempo de modificação com os tempos de suas entradas, e executa se estiver fora da data. Além disso, objetos de processo precisam apenas saber sobre suas entradas diretas, e nenhum conhecimento global de outros objetos (como um executivo de rede) é necessário. A desvantagem do controle implícito é a dificuldade em distribuir a execução da rede para outros computadores ou implementar estratégia de execução sofisticada.

2.7. Modelos de Objetos

Os modelos de objetos nesta monografia utilizam a Técnica de Modelamento de Objetos (OMT) desenvolvida por Jim Rumbaugh e seus colegas [Rum91], a mesma utilizada no desenvolvimento do VTK. A OMT usa três modelos para especificar um projeto orientado a objeto: um modelo de objeto, um modelo dinâmico e um modelo funcional. Cada modelo descreve um aspecto diferente do sistema e cada um tem uma técnica de diagramação correspondente que ajuda analisar, projetar e implementar sistemas de softwares.

O modelo de objeto identifica cada objeto no sistema, suas propriedades e seus relacionamentos com outros objetos do sistema. Para a maioria dos sistemas de software, o modelo de objeto domina o projeto. O OMT usa retângulos para descrever classes de objetos e uma variedade de conectores para descrever herança e outras relações orientadas a objeto. O nome da classe ocupa o topo do retângulo. Uma linha separa o nome da classe da próxima seção que contém os atributos. Uma terceira seção descreve os métodos associados à classes. OMT representa herança com um triângulo, com a superclasse ligada ao ápice, e a subclasse ligada à base do triângulo.

2.8. A Estrutura de Classes do VTK

O VTK permite tanto a representação de objetos por DVR, onde os dados são armazenados como uma malha de pontos estruturados, como a modelagem e apresentação por *rendering* por superfície. Abaixo seguem algumas das classes presentes no VTK que representam os conjuntos de dados suportados pela biblioteca e seus mapeadores.

Todos os objetos criados no VTK, com poucas exceções, são especializações (subclasses) da classe abstrata *vtkObject* ou de um de seus filhos. Essa classe base fornece métodos de controle para cada objeto criado pela biblioteca, incluindo trilhagem do tempo de modificações, depuração e impressão.

Os conjuntos de dados de visualização suportados pelo VTK são mostrados na figura 2.6. Para a representação desses conjuntos de dados foi criada a classe genérica *vtkDataObject*. Objetos do tipo “conjunto de dados” (*datasets*) implementam os tipos de dados fundamentais tipicamente usados em visualização. Esses objetos de dados são entrada e saída de fontes, filtros e mapeadores. A classe abstrata *vtkDataSet* especifica uma interface que todas as classes derivadas precisam fornecer. Ela também provê métodos que fornecem informações sobre os dados, tais como o centro do objeto, caixa de bordo (limites espaciais do domínio de dados) e largura representativa (comprimento da diagonal da caixa de bordo). Um conjunto de dados no VTK consiste de uma estrutura definida por geometria e topologia, e de atributos de dados associados, sendo composto por um ou mais pontos e células. A figura 2.7 mostra o diagrama de classes do *vtkDataSet* e suas subclasses. O tipo de um conjunto de dados é determinado pela sua estrutura de organização, e especifica o relacionamento que as células e pontos têm entre si.

Para trabalhar com malhas do tipo cartesiana, o VTK possui uma classe chamada *vtkStructuredPoints*. A *vtkStructuredPoints* representa uma estrutura geométrica que é uma matriz de pontos topologicamente e geometricamente regular. Essa estrutura é utilizada para o armazenamento de figuras bidimensionais, como texturas, e dados volumétricos para a representação em DVR ou reconstrução. A classe *vtkRectilinearGrid* representa uma malha retilínea

a) Pontos Estruturados
(*vtkStructuredPoints*)

b) Malha Retilínea
(*vtkRectilinearGrid*)

c) Malha Estruturada
(*vtkStructuredGrid*)

d) Pontos Não Estruturados

e) Dado Poligonal
(*vtkPolyData*)

f) Malha Não Estruturada
(*vtkUnstructuredGrid*)

Figura 2.6: Tipos de conjuntos de dados (*datasets*) do VTK. A “Malha Não Estruturada” consiste de todos os tipos de célula. Adaptada de [Sch98].

A classe abstrata *vtkPointSet* especifica a interface para conjuntos de dados que usam explicitamente matrizes de pontos para representar geometria. Por exemplo, *vtkPolyData* e *vtkUnstructuredGrid* requerem matrizes de pontos para especificar as posições dos pontos, enquanto *vtkStructuredPoints* gera posições de pontos implicitamente. A *vtkStructuredGrid* representa uma malha estruturada. A classe *vtkUnstructuredGrid* consiste de uma combinação arbitrária de células 2D e 3D, além de pontos. Geometria e topologia são explicitamente definidas.

Figura 2.7: Diagrama de objetos dos tipos de dados do VTK herdeiros de *vtkDataSet* [Sch98].

A maioria das fontes do VTK geram objetos no formato “dados poligonais”, ou *PolyData*, que é um tipo de conjunto de dados que pode ser facilmente renderizado para bibliotecas gráficas. A classe *vtkPolyData* representa esse tipo de estrutura geométrica, consistindo de vértices, linhas, polígonos e faixa de triângulos. Ela é a estrutura utilizada durante o processo de renderização dos conjuntos de dados geométricos do VTK, que ocorre através da classe *vtkPolyDataMapper*.

A classe *vtkPolyDataMapper* é a classe que mapeia dados poligonais (*PolyData*) para primitivas gráficas renderizáveis. Ela é uma superclasse de mapeadores poligonais para dispositivos específicos, como a biblioteca OpenGL. O usuário não precisa selecionar qual dispositivo o VTK usará. Ele simplesmente aloca um mapeador de dados poligonais e o VTK escolhe o dispositivo adequado para a renderização.

Para os outros tipos de dados, é utilizada a classe *vtkDataSetMapper*, que é um mapeador de conjuntos de dados (*vtkDataSet* e todas classes derivadas) para primitivas gráficas. A classe *vtkDataSetMapper* não possui subclasses específicas. O processo de renderização desse mapeador converte o conjunto de dados (*DataSet*) para dados poligonais (*PolyData*) através de um filtro e renderiza utilizando o *vtkPolyDataMapper*. Uma exceção é o DVR, que possui mapeador próprio.

A classe abstrata *vtkDataSet* serviu de base para a inclusão da nova estrutura de dados SHF. O próximo capítulo apresenta alguns conceitos relacionados à Modelagem Geométrica necessários para o entendimento dessa estrutura de dados, além de outras estruturas de dados criadas antes da SHF.

Capítulo 3 – Conceitos Fundamentais e Estruturas de Dados Topológicas

3.1. Considerações Iniciais

Segundo Laszlo [Las96], a Computação Gráfica compreende métodos para modelar e renderizar cenas. A modelagem é usada para construir uma descrição da cena, e a natureza da cena pode ser variada: objetos normais em duas ou três dimensões, fenômenos naturais como nuvens ou árvores, páginas de texto, oceanos de números obtidos de dispositivos de imagem ou estudos de simulações, e muitos outros. Renderização, conforme mencionado no capítulo 2, é usada para transformar a descrição da cena em uma imagem ou animação.

A modelagem geométrica estuda técnicas para a construção de uma cena através da representação computacional de formas geométricas, como curvas, superfícies, sólidos e volumes, preocupando-se com seu tratamento (operações de interseção e união, por exemplo), recuperação de informações e sua apresentação visual.

Este capítulo tem a finalidade de introduzir alguns conceitos básicos relacionados à área de Modelagem Geométrica, como variedades, células, vértices, arestas, faces, e outros, necessários para o entendimento do restante do texto. Em seguida, são apresentadas algumas estruturas de dados topológicas existentes anteriormente à SHF.

3.2. Definições

3.2.1. Variedade

Sejam dois conjuntos abertos U e V e uma aplicação, $f : U \rightarrow V$, inversível. f é chamada de **homeomorfismo** se f e f^{-1} são contínuas. Se f é um homeomorfismo, é dito que o conjunto U é **homeomorfo** ao conjunto V .

Uma **variedade** (*manifold*) de dimensão k contida em $\mathbb{R}^n$ ($n \geq k$) é um conjunto de pontos em que a vizinhança, dentro desse conjunto, de qualquer um dos pontos, é equivalente topologicamente, isto é, homeomorfo, a um aberto de $\mathbb{R}^{k-1} \times \mathbb{R}^+$ [Gom98] (ver figura 3.1). Se $k=2$ tem-se uma **variedade bidimensional**.

Se existir pelo menos um ponto do conjunto acima que não satisfaz essa condição o conjunto é dito ser uma **não variedade** (*non manifold*) e o ponto que não satisfaz a condição é chamado de **ponto singular**.

Figura 3.1: (a) Variedade, (b) Não Variedade onde "P" é um ponto singular.

3.2.2. Fecho Convexo

Sejam P_0 e P_1 dois pontos distintos. O **fecho convexo** de P_0 e P_1 é o conjunto dos pontos dados por [Hof89]:

$$[P_0, P_1] = \{ \lambda P_0 + (1 - \lambda) P_1 \mid 0 \leq \lambda \leq 1 \}$$

Geometricamente, $[P_0, P_1]$ é um segmento de reta no espaço Euclidiano. Similarmente, o fecho convexo de três pontos P_0, P_1 e P_2 é o conjunto dos pontos tais que:

$$[P_0, P_1, P_2] = \{ (\lambda P_0 + (1 - \lambda) P_1) M + (1 - M) P_2 \mid 0 \leq \lambda, M \leq 1 \}$$

Se os pontos não são colineares, então $[P_0, P_1, P_2]$ é um triângulo com vértices P_0, P_1 e P_2 . O fecho convexo generalizado de $d + 1$ pontos é dado pelo conjunto dos pontos tais que:

$$[P_0, \dots, P_d] = \{ M q + (1 - M) P_d \mid q \in [P_0, \dots, P_{(d-1)}], 0 \leq M \leq 1 \}$$

3.2.3. Decomposição Celular

Uma **célula afim** em $\mathbb{R}^p$ é o fecho convexo de um número finito de pontos, ou seja, dados $v_0, \dots, v_n \in \mathbb{R}^p$ a célula afim σ pode ser escrita da seguinte forma [Cas92]:

$$v \in \sigma$$

$$v = \sum_{i=0}^n \lambda_i v_i \quad \text{onde} \quad \sum_{i=0}^n \lambda_i = 1 \quad \text{e} \quad \lambda_0, \dots, \lambda_n \geq 0.$$

A dimensão de uma célula afim σ é dada pela dimensão do espaço vetorial gerado pelos vetores $[v_1 - v_0, v_2 - v_0, \dots, v_n - v_0]$.

Uma **célula** de dimensão p em $\mathbb{R}^m$ é uma imagem homeomorfa de uma célula afim em $\mathbb{R}^p$.

Uma coleção C de células i -dimensionais ($i = 0, 1, \dots, n$) em $\mathbb{R}^m$ é uma **decomposição celular** de um conjunto $K \subset \mathbb{R}^m$ se as seguintes condições forem satisfeitas:

1. $K = \bigcup \{\sigma \in C\}$;
2. Se σ e $\tau \in C$ então $\sigma \cap \tau \in C$, sendo essa interseção vazia ou uma sub-célula de ambas;
3. Qualquer subconjunto compacto de $\mathbb{R}^m$ intercepta somente um conjunto finito de células.

Diz-se que uma célula σ é incidente a uma célula τ , quando ambas pertencem à mesma decomposição celular e também σ é uma sub-célula de τ ou vice-versa.

As células de dimensão 0, 1 e 2 de C são chamadas respectivamente de vértices, arestas e faces. O conjunto de células que contém um dado vértice v é chamado de **estrela** do vértice v .

Um **complexo celular** M mergulhado no espaço $\mathbb{R}^m$ é uma coleção finita de células $\{c^r\}$, onde cada c^r é uma célula afim de dimensão r , cujo bordo $\partial(c^r)$ de c^r é a união de subcélulas c^s com $s < r$, e tal que $c^r \cap c^s \neq \emptyset$. O complexo celular M é dito ser k -dimensional se k é a maior dimensão de suas células [Lop96].

3.2.4. Complexos Simpliciais

Um **d-simplexo** é uma célula convexa afim com dimensão d gerada por $d+1$ pontos afimemente independentes. Por exemplo:

- Um 0-simplexo é um ponto.
- Um 1-simplexo é uma aresta.
- Um 2-simplexo é um triângulo.
- Um 3-simplexo é um tetraedro.

O **bordo de um d-simplexo** S , denotado por ∂S , consiste de todos os $(d-k)$ -simplexos contidos em S , tal que $0 < k \leq d$. Todo simplexo no bordo de S é denominado face de S . Um k -simplexo no bordo é denominado k -face.

Um **complexo simplicial** C é um conjunto finito de simplexos satisfazendo as seguintes propriedades:

1. Seja S um simplexo em C e S' uma das faces de S , então S' está contida em C .
2. Sejam S_1 e S_2 dois simplexos de C , então a interseção de S_1 com S_2 , ou é vazia ou é uma face comum a S_1 e S_2 .

A dimensão de um complexo simplicial C é dada pela máxima dimensão de seus simplexos.

3.2.5. Estrutura de Dados

Um **grafo** é um conjunto finito não vazio de nós, denotado por $N(G)$, e um conjunto possivelmente vazio de linhas dirigidas que ligam dois nós, denotado por $L(G)$. Uma **estrutura** é um grafo onde o conjunto de nós $(N_1, \dots, N_n)$ e as linhas dirigidas $(L_1, \dots, L_n)$ são de diferentes classes. Do ponto de vista computacional, cada linha é um ponteiro, e cada nó ocupa um espaço de tamanho fixo na memória suficiente para conter todos os ponteiros referentes a ele, e também seus atributos.

3.3. Estruturas Topológicas

As definições que seguem nesta seção foram baseadas no trabalho de Soares [Soa99].

A modelagem geométrica identifica um conjunto de técnicas para manipular elementos geométricos como curvas e superfícies. Seu desenvolvimento está ligado a diversas aplicações na ciência e indústria. Existem várias técnicas para a modelagem desses objetos tridimensionais. Dentre as principais pode-se citar: a instanciação de primitivas, varredura (sweeping), Geometria Sólida Construtiva, modelagem implícita, modelagem paramétrica e a modelagem baseada em geometria fractal [Mor85]. A modelagem de sólidos surge como uma ferramenta de apoio, enfatizando a aplicabilidade dos modelos geométricos em contextos reais como medicina e manufatura. Em muitos casos, surge a necessidade de representar e manipular objetos sólidos.

De acordo com Requicha e Voelcker [Req83], o termo modelagem de sólidos envolve um corpo de teorias, técnicas e sistemas que focalizam uma representação de sólidos permitindo, pelo menos em princípio, que qualquer propriedade geométrica bem definida de qualquer sólido representável seja calculada. A principal importância dos sistemas de modelagem de sólidos é a sua capacidade de distinguir entre o interior, o exterior e a superfície de um objeto tridimensional, para o cálculo de propriedades dependentes dessa distinção.

A representação do sólido pela sua fronteira é bastante difundida. Nela, um sólido é descrito como uma coleção de faces, arestas e vértices que definem a sua superfície, e, para cada uma dessas entidades existem duas componentes: a parte geométrica da representação, que define a sua forma ou a localização no espaço; e a parte topológica, que descreve as relações de incidências e adjacências com os elementos que compõem o objeto. A técnica de **representação por fronteira** (“B-rep” – *Boundary Representation*) combina em uma estrutura de dados as informações topológicas e geométricas que descrevem a fronteira de um sólido, isto é, a superfície (normalmente fechada) que o define. A estrutura de dados capaz de armazenar as informações topológicas de um sólido ou objeto 3D é denominada *estrutura topológica*.

Em um modelo de representação por fronteira, um sólido consiste de uma coleção de faces agrupadas, e cada face é representada como um polígono, sendo que cada polígono é descrito geometricamente por uma seqüência de coordenadas dos vértices que o definem. Como algumas faces compartilham o mesmo vértice, pode ocorrer redundância de informação. Essa redundância é eliminada pela introdução de vértices como entidades independentes na estrutura de dados. Nesse caso, os identificadores de vértices são associados às faces, e as coordenadas associadas diretamente aos vértices.

Se o modelo de fronteira deve representar superfícies curvas, é interessante incluir também nós arestas explicitamente na estrutura de dados, tornando-a capaz de armazenar informações de curvas de interseção entre faces. Assim, cada face é representada em termos de uma seqüência fechada de arestas, chamada ciclo. A estrutura de dados atribui uma orientação para cada aresta. As faces também são orientadas seguindo o sentido horário para um observador olhando de fora do objeto. No caso de superfícies que são variedades, cada aresta ocorre em exatamente duas faces, uma vez orientação positiva, e uma vez na sua orientação oposta (negativa) da face.

A definição de variedade usada em alguns trabalhos anteriores à SHF é que variedade é uma superfície orientada sem bordo no R^3 . Para diferenciá-la da definição anterior, o termo será usado em itálico “*variedade*” quando estiver se referindo a essa última definição.

Um dos trabalhos precursores em modelos de representação de sólidos por fronteira baseado em arestas foi proposto por Baumgart [Bau75a][Bau75b], que apresenta a estrutura de dados *Winged-Edge*. Essa estrutura define entidades topológicas e um conjunto de operadores associados a fim de lidar com as construções livres de coordenadas na Modelagem de Sólidos. Uma visão geral de várias abordagens pode ser vista em Hoffman [Hof91].

O trabalho de Baumgart trata representações implícitas de *variedades*, ou seja, as células da decomposição celular que descreve a variedade são representadas implicitamente, e não explicitamente como nós (elementos) de uma estrutura de dados. Já a estrutura de dados Half-Edge, proposta por Mäntylä [Män88] e derivada da estrutura *Winged-Edge*, é um exemplo de representação explícita voltada para sólidos descritos por superfícies que são *variedades* bidimensionais sem bordo.

Essa estrutura interpreta cada aresta como composta de duas metades que correspondem aos sentidos positivos e negativos da aresta. A essas metades dá-se o nome de semi-arestas. A figura 3.2 (a) mostra a decomposição da aresta em semi-arestas.

Figura 3.2. Representação Half-Edge

A figura 3.2 (b) apresenta um diagrama da estrutura de dados Half-Edge, cujos elementos são:

- **Superfície:** É o nó raiz da estrutura, que dá acesso às faces, arestas e vértices do modelo por meio de ponteiros para três listas duplamente encadeadas.
- **Face:** Representa uma face planar do poliedro que define a superfície do sólido. Uma face pode possuir múltiplos bordos pela sua associação com uma lista de ciclos, em que cada ciclo representa uma curva de bordo poligonal da face. Todas as faces representam polígonos planares, um dos ciclos pode ser designado como a borda mais externa, enquanto os outros representam buracos de uma face.

- **Ciclo:** Descreve um bordo conectado como descrito acima. Ele tem um ponteiro para a face à qual pertence e um ponteiro para uma das semi-arestas que formam o bordo.
- **Semi-aresta:** Descreve um segmento de reta de um ciclo. Ele contém um ponteiro para o nó ciclo ao qual pertence e um outro para o vértice de onde se origina, obedecendo a orientação do ciclo. O vértice final de um segmento de linha é o vértice de início da próxima semi-aresta.
- **Aresta:** Foi incluído na estrutura para indicar diretamente como as faces se relacionam entre si. Esse relacionamento é fornecido pelas duas semi-arestas que formam as arestas do modelo.
- **Vértice:** Contém as coordenadas de um ponto em $\mathbb{R}^3$. Cada vértice contém um ponteiro para a semi-aresta da qual é origem.

É fundamental que a manipulação de uma estrutura de dados topológica não altere as condições topológicas do modelo sendo manipulado. Como as estruturas de dados são complexas, define-se um conjunto de operadores topológicos capazes de manipular as estruturas de dados obedecendo as condições de topologia. Para a manipulação da estrutura Half-Edge foram definidos os operadores de Euler, que foram derivados a partir da Lei de Euler generalizada para objetos poliedrais [Män88].

O trabalho de Mäntyla foi estendido por Lopes com a estrutura *Handle-Edge*, que possibilita a representação de variedades bidimensionais com ou sem bordo [Cas92]. Duas outras estruturas foram criadas a partir da *Handle-Edge*. A estrutura *Handle-Face*, proposta também por Lopes [Lop96][Lop97] para representar variedades tridimensionais com ou sem bordo, representa explicitamente a superfície de bordo do sólido, o que desempenha uma função essencial no controle da topologia. Essa estrutura utiliza o conceito de *faces orientadas*, ou *semi-faces*, para representar uma das duas orientações possíveis de cada face da variedade. A segunda é a estrutura *Handle-Cell*, proposta por Pesco [Pes97], que se baseia num modelo de espaços estratificados que permite representar não só superfícies (estratos de dimensão 2), como também curvas (estratos de dimensão 1) e pontos (estratos de dimensão 0) em um mesmo ambiente.

Algumas estruturas topológicas, como a *Handle-Edge* e a *Handle-Face*, são altamente eficientes e sofisticadas, mas representam apenas variedades de dimensões dois e três, respectivamente. A estrutura *Handle-Cell* é específica para representar "não variedades", porém permite apenas a representação de objetos com dimensão menor ou igual a dois. A proposta da SHF surgiu da necessidade de ter uma estrutura para "não variedades" capaz de representar objetos de dimensão três. Em visualização científica é necessário possuir uma estrutura com essa capacidade, pois muitos processos que criam objetos geométricos não podem garantir a geração de variedades. O próximo capítulo apresenta essa estrutura, seus operadores topológicos e conversores para importar e exportar dados.

Capítulo 4 – A Estrutura Singular Half-Face (SHF)

4.1. Considerações Iniciais

Este capítulo apresenta a estrutura de dados SHF, seus operadores topológicos e os conversores para importação e exportação de arquivos de dados, descrevendo o funcionamento de acordo com a proposta por Bísaro [Bís01].

4.2. A Estrutura SHF

A estrutura SHF representa objetos volumétricos enfatizando a velocidade na obtenção das relações de incidências e adjacências entre seus elementos topológicos, e também os custos de armazenamento, podendo representar objetos descritos por células não convexas e faces não planares.

Todos os nós da estrutura são mantidos em listas simplesmente encadeadas, sendo que somente a lista de semi-arestas é circular duplamente encadeada, pois se trata de um ciclo orientado e, dessa forma, a manipulação do elemento torna-se mais eficiente. A estrutura, ilustrada na figura 4.1, não representa explicitamente nem faces nem arestas, com o objetivo de reduzir os custos de armazenamento, já que essas informações podem ser obtidas examinando qualquer uma de suas semi-arestas ou semi-faces. No entanto, os elementos vértices singulares e arestas singulares, que poderiam ser obtidos percorrendo as listas de vértices e semi-arestas, foram incluídos para garantir velocidade no acesso a tais elementos.

Figura 4.1: Visão geral da organização da estrutura de dados SHF. Adaptada de [Bís00].

A estrutura contém os seguintes tipos de nós:

- **Sólido:** contém cinco ponteiros que apontam para as listas de nós, que descrevem vértices, faces de bordo, células, vértices singulares e arestas singulares, que compõem o sólido.
- **Célula:** contém ponteiros para a lista de semi-faces que definem essa célula e para o sólido ao qual a célula pertence.
- **Vértice:** contém ponteiros para a sua estrela (isto é, os elementos que o contém), e para o sólido ao qual o vértice pertence.

Nessa estrutura, a **estrela** de um vértice V é dada pelo conjunto de todas as arestas que contêm V . Por exemplo, na figura 4.2, a estrela do vértice $v5$ é formada pelas arestas: $a5$, $a6$, $a7$ e $a8$. Cada aresta é formada por, no mínimo, duas semi-arestas em sentidos contrários (figura 4.3a). Como a estrutura não representa arestas explicitamente, o nó estrela armazenará uma das semi-arestas que define a aresta.

Figura 4.2: Estrela de um vértice.

- **Componente de Bordo:** aponta para uma lista de bordos e para o sólido ao qual pertence. Um sólido pode conter mais do que um bordo. Por exemplo, um sólido com um cavidade interna contém dois bordos.
- **Face de Bordo:** aponta para uma lista das semi-faces que definem o bordo e para o componente ao qual a face de bordo pertence.
- **Semi-Face:** contém ponteiros para o sólido que a contém, para a lista de semi-arestas que compõem essa semi-face e para seu *mate*, isto é, a semi-face que está "colada" na semi-face em questão (figura 4.3(b)). Caso a semi-face seja de bordo, esse ponteiro é nulo.

Figura 4.3: (a) Semi-face formada por 3 semi-arestas, (b) Mate de uma face. Adaptada de [Bís00].

- **Semi-Aresta:** contém ponteiros para seu sólido, seu vértice inicial, sua *mate-célula*, que é a semi-aresta que compartilha a mesma aresta na mesma célula (figura 4.4(a)), e um ponteiro para sua *mate-aresta*, que é a semi-aresta que compartilha a mesma aresta, mas em uma célula vizinha (figura 4.4(b)). Além desses, o nó semi-aresta contém mais dois ponteiros: um para a próxima semi-aresta na semi-face e outro que aponta para a semi-aresta anterior. Ou seja, o vértice inicial da próxima semi-aresta da semi-aresta X corresponde ao vértice final da semi-aresta X, formando um ciclo.

Figura 4.4: (a) Mate-Célula, (b) Mate-Aresta. Adaptada de [Bís00].

Os ponteiros mate-aresta assumem um papel ligeiramente diferente nas faces de bordo, o de ligar arestas em faces de bordo vizinhas, fazendo com que o bordo fique totalmente conectado, sendo possível "caminhar" ao longo do bordo utilizando esses ponteiros (figura 4.5).

- **Estrela do Vértice:** possui apenas um ponteiro para a lista das semi-arestas que compõem a estrela de um vértice.
- **Vértice Singular:** uma lista cujos elementos contém um ponteiro para o seu sólido e outro para o vértice considerado singular.

Figura 4.5: Mate-aresta no bordo. Adaptada de [Bís00].

- **Aresta Singular:** uma lista cujos elementos contém um ponteiro para o seu sólido e outro para a aresta considerada singular.

Nos processos de reconstrução, tipicamente utilizados em visualização científica, é importante que seja possível armazenar informações variadas na estrutura. A estrutura SHF possui um campo do tipo “*void *data*” que permite armazenar qualquer tipo de informação como: variáveis de orientação, ponteiros, valores escalares, temperatura, energia, valores vetoriais como velocidade, etc.

4.3. Operadores e Conversores

A inserção de uma nova célula no complexo celular que descreve o sólido, e sua remoção devem garantir que a estrutura se mantenha topologicamente consistente. Com esse fim, foram definidos, inicialmente, operadores de colagem e descolagem de células. Esses operadores atualizam a estrutura sempre que uma célula é adicionada ou removida do sólido, mantendo corretas as relações de incidência e adjacência na estrutura de dados, além de alterar as listas de arestas singulares, vértices singulares e faces de bordo se necessárias.

Os sólidos que estão armazenados na estrutura podem ser salvos em arquivos de diversos formatos através de funções especiais chamadas conversores. Esses conversores

também fazem a leitura de dados, escritos em vários formatos, armazenando-os na estrutura. Esses conversores de leitura utilizam os operadores de colagem para incluir as novas células.

4.4. Colagem e Descolagem de Células

Várias são as ocasiões em que precisamos "colar" ou "descolar" células em um complexo celular representado pela estrutura de dados SHF. Esses operadores devem ser bastante gerais de forma que uma célula de qualquer formato ou em qualquer posição possa ser "colada" ou "descolada" de maneira correta.

O operador de colagem de células realiza a "colagem" de uma célula que já pertence à estrutura às demais células do sólido, ou seja, após inserir o novo elemento célula com suas semi-faces e semi-arestas na estrutura, chama-se o operador de colagem para atualizar informações como faces de bordo, estrelas dos vértices, vértices singulares, arestas singulares, *mates* e *mate-arestas*. Os ponteiros mate-células não precisam ser atualizados, já que eles não sofrem qualquer mudança com o processo de "colagem". Esses ponteiros devem, portanto, indicar corretamente suas "mate-células" antes de iniciar o processo de colagem (ou seja, a nova célula deve ter sido inserida corretamente na estrutura).

A função de colagem recebe um ponteiro para a célula a ser colada, e executa os seguintes passos:

1. Atualiza a estrela dos vértices: cada vértice da nova célula terá uma estrela definida por suas arestas e pelas arestas do sólido que a possui.
2. Identifica as faces coincidentes e atualiza seus *mates*, eliminando-a da lista de faces de bordo.
3. Identifica arestas coincidentes e atualiza suas *mate-arestas*. Para faces de bordo, identifica arestas vizinhas. No caso de arestas singulares, pode acontecer de três ou mais células estarem compartilhando uma mesma aresta (figura 4.6). Nesse caso, torna-se necessário calcular os ângulos entre as faces de bordo para saber quais são as faces de bordo vizinhas e, assim, descobrir quais são as semi-arestas vizinhas. Na figura 4.6, a *mate-aresta* da semi-aresta (1) da célula B pode

ser a semi-aresta da célula A ou a da célula C, já que ambas compartilham a mesma aresta e possuem mesma orientação. Nesse caso, como o ângulo formado entre a face da célula B e a face da célula A é menor que o formado entre as faces da célula B e da célula C, a mate-aresta é a semi-aresta da célula A, e o ponteiro correspondente apontará para ela.

Figura 4.6: Mate-aresta no bordo de células vizinhas, compartilhando uma aresta singular.

4. Insere as novas faces de bordo na lista correspondente. Para identificar quais faces devem ser inseridas, basta verificar quais faces não possuem mates, ou seja, aquelas para os quais o valor dos mates após o segundo passo ainda é nulo (NULL).
5. Verifica se a nova célula contém vértices ou arestas singulares, e acrescenta as nas listas de vértices singulares e de arestas singulares do sólido, respectivamente.

O operador de "descolagem" de células segue basicamente o mesmo padrão do operador de "colagem", porém em sentido contrário, com o cuidado de que as faces coincidentes da célula que está sendo retirada se tornarão faces de bordo.

A função de descolagem recebe o identificador da célula que será retirada e retorna o ponteiro para um novo sólido contendo apenas a célula retirada. Os seguintes passos são realizados por esse operador:

1. Cria um novo sólido S.
2. Retira a célula dada pelo identificador da lista de células do sólido antigo.
3. Atualiza a estrela dos vértices que não serão removidos do sólido.
4. Verifica se, após a retirada da célula, foi gerado ou eliminado algum vértice singular e atualiza a lista correspondente.
5. Descola as faces, ou seja, atualiza os mates das faces que serão separadas e acrescenta as na lista de faces de bordo.
6. Descola as arestas antes compartilhadas pela célula removida, ou seja, atualizar todas as mate-arestas.
7. Verifica se surgiu ou desapareceu alguma aresta singular, e atualiza a lista correspondente.
8. Associa a célula ao novo sólido. Os vértices, as semi-arestas e semi-faces da célula são copiados para o novo sólido e apagados do sólido antigo. Vértices que também pertencem ao sólido antigo são mantidos.

4.5. Operadores Topológicos

Os operadores de Morse para tetraedros (ou simplesmente operadores de Morse) são um conjunto de operadores topológicos para inclusão e exclusão de tetraedros num complexo simplicial, mantendo controle sobre o número de componentes conexas, buracos e cavidades (figura 4.7). Com relação a uma estrutura de dados topológica, os operadores de Morse possibilitam uma maneira mais robusta de controlar as relações de incidências e adjacências, uma vez que os elementos afetados pela adição ou remoção de um tetraedro estão completamente especificados em cada operador.

Figura 4.7: Inserção de um tetraedro: (a) gera uma cavidade (M3EP), (b) gera um buraco (MVEH).

Existem cinquenta e dois operadores de adição, separados em sete grupos, de acordo com as alterações provocadas na estrutura:

1. $\mathbf{TMO}_0 = \{MV, ME, M2E, M3E, MF, M2F, M3F, MEF\}$
2. $\mathbf{TMO}_{H0} = \{MC, M2VKC, M3VK2C, M4VK3C, MVEKC, M2VEK2C, MV2EKC, M2EKC, MVFKC\}$
3. $\mathbf{TMO}_{H1} = \{M2VH, M3V2H, M4V3H, MVEH, M2VE2H, MV2EH, M2EH, M3EKH, M4EKH, M5EK2H, M6EK3H, MVFH, M2EFKH, M3EFK2H, ME2FKH, MV3EHKH\}$
4. $\mathbf{TMO}_{H2} = \{M3EP, M4EP, M5E2P, M6E3P, M3EF2P, M2EFP, ME2FP, M4FKP\}$
5. $\mathbf{TMO}_{H0H1} = \{M3VHKC, M4VHK2C, M4V2HKC, M2VEHKC, MV3EKCH\}$
6. $\mathbf{TMO}_{H0H2} = \{MV3EPKC\}$
7. $\mathbf{TMO}_{H1H2} = \{MV3EHP, M5EPKH, M6EPK2H, M6E2PKH, M3EFPKH\}$

Todos os operadores dos grupos com H0 no nome alteram o número de componentes da estrutura. Os operadores dos grupos H1 alteram o número de buracos e os operadores dos grupos H2 alteram o número de cavidades da estrutura.

Os seguintes símbolos são usados para descrever as possíveis ações de cada operador sobre os elementos que eles alteram: M – *make* (incluir), K – *kill* (remover), H – *hole* (buraco), P – *pocket* (cavidade), C – *component* (componente), V – *vertex* (vértice identificado), E – *edge* (aresta identificada), F – *face* (face identificada). Números são utilizados para definir a quantidade de vezes que a ação é executada em cada elemento (o

número 1 é omitido para simplificar a notação). Esses símbolos, que dão nome aos operadores, dão uma sumária descrição de sua ação sobre um complexo simplicial. Por exemplo: o símbolo **MV2EH** nomeando um operador em **TMO_{HI}** indica que o operador identifica um vértice, duas arestas (totalizando três vértices) e inclui um buraco.

Operadores de remoção de tetraedros podem ser obtidos trocando os símbolos **M** por **K** (e vice-versa), na lista de operadores descritos acima.

4.6. Conversores

Não se conhece, em princípio, o formato dos dados a serem representadas na estrutura, e seria uma limitação muito grande carregar os dados a partir de um único formato. Por esse motivo, vários conversores foram implementados para gerar a representação SHF a partir de diferentes formatos de dados, visando dar maior flexibilidade à estrutura. Conversores de SHF para outros formatos também são ferramentas úteis. Alguns desses formatos estão descritos abaixo:

4.6.1. Conversor para o Formato HLF

O primeiro conversor carrega a representação de objetos descritos por uma sequência de células, faces e vértices em um formato denominado “formato HLF”. Esse formato é muito parecido com o formato "PolyData" utilizado pelo VTK, porém mais geral. A figura 4.8(a) apresenta o formato, que deve ser interpretado da seguinte forma:

O primeiro número que aparece no arquivo indica a quantidade de vértices do objeto representado. Logo abaixo são listadas as coordenadas espaciais (x,y,z) de cada vértice. O arquivo prossegue com um número indicando o número de faces do objeto. Cada face é definida por vários vértices, indicados pelo seu índice onde o número 0 corresponde ao primeiro vértice carregado. O número de vértices para cada face está indicado na frente dos índices. No exemplo da figura 4.8(a), o número 10 abaixo das coordenadas dos vértices indica que o objeto tem dez faces. Na primeira face, localizada na linha logo abaixo do número de faces, o primeiro número (“3”) indica que há três vértices na primeira face e os números 0, 1 e 2 indicam que a primeira face é formada pelos vértices 0 (primeiro vértice

gravado, cujas coordenadas nesse exemplo são 0.0, 0.0 e 0.0), 1 (segundo vértice) e 2 (terceiro vértice). Os próximos nove conjuntos de números indicam as próximas nove faces.

Em seguida, outro número aparece indicando a quantidade de células do objeto. As linhas seguintes informam os índices das faces que definem cada célula. Da mesma forma que na lista de faces, o número de faces de cada célula está indicado na frente dos índices. No exemplo da figura 4.8(a), o número 2 indica duas células no objeto e a primeira célula é formada por 5 faces com índices 0 (primeira face carregada) , 1, 2, 3 e 4.

A figuras 4.8(b) e 4.8(c) exibem alguns objetos que poderiam ser representados nesse formato, sendo que os dois objetos superiores contém singularidades de aresta. O objeto definido pelo arquivo exemplo da figura 4.8(a) corresponde ao primeiro objeto da figura 4.8(b), formado pelos dois prismas conectados por uma aresta singular.

Figura 4.8: Formato VTK: (a) conteúdo de um arquivo com 2 células e 10 faces, (b) exemplos de objetos com singularidades de aresta, (c) mais exemplos de objetos nesse formato.

Os passos para importar um arquivo HLF são:

1. Carregar a lista de vértices do sólido para a estrutura.
2. Carregar a lista de faces em uma variável auxiliar, necessária porque a estrutura não representa faces explicitamente. Dessa forma, cada face poderá gerar uma ou mais semi-faces.
3. Carregar a lista de células em outra variável auxiliar.
4. Criar a lista de células do sólido. Utilizando as variáveis auxiliares, é possível gerar todas as células do sólido com suas semi-faces e semi-arestas.
5. Atualizar as mate-células.
6. Colar as semi-faces, ou seja, atualizar as *mate* e as *mate-arestas* das faces que compartilham os mesmos vértices, e acrescentar na lista de faces de bordo as faces que não possuem mates.
7. Atualizar a orientação de todas as células: todas as semi-faces externas de cada célula devem possuir a mesma orientação, de modo que seu mate fique com a orientação contrária.
8. Gerar a estrela de cada vértice.
9. Verificar se existe algum vértice singular ou aresta singular e atualizar as respectivas listas.
10. Atualizar as mate-arestas das faces de bordo com as faces de bordo vizinhas. Para o caso de arestas singulares que compartilham mais de duas células, observar os ângulos formados entre as semi-faces para descobrir quais são as semi-arestas vizinhas.

4.6.2. Conversor para o Formato Unstructured Grid

Esse conversor lê arquivos salvos no formato *vtkUnstructuredGrid* (malha não estruturada) do VTK. Esse formato é bastante geral e permite objetos que não são necessariamente volumétricos, mas a implementação do conversor se restringe a objetos formados exclusivamente por células do tipo tetraedro. O arquivo no momento apresenta a seguinte forma: (figura 4.9).

Figura 4.9: Exemplo de um arquivo no formato unstructured grid.

O valor de “n” após o termo "POINTS" representa o número de vértices do sólido, "dataType" refere-se ao tipo de dados que é utilizado para representar as coordenadas dos vértices (inteiros ou reais). Os dados "px", "py" e "pz" correspondem às três coordenadas de cada vértice em R^3 . O valor de “m” indica o número de células do sólido e "size" informa a quantidade de memória ocupada pelas células, que corresponde a "5n" no caso de um sólido formado apenas por tetraedros, já que todas as células têm 4 vértices, mais a memória utilizada para indicar o número de vértices em cada célula.

Para cada célula, os valores de "i", "j", "k", "l" representam os índices dos vértices que a definem, e o número inicial (que nesse caso é quatro, pois estamos trabalhando somente com tetraedros) corresponde ao número de vértices da célula.

Note que, nesse arquivo, não são dadas as relações de adjacências entre células e seus elementos. Todas essas relações têm que ser construídas durante o processo de carregar o objeto para a estrutura SHF.

O procedimento realizado por esse conversor para importar os dados na estrutura segue os mesmos passos do conversor que importa dados no formato HLF. A única diferença está no processo de criação de células, já que esse formato não define explicitamente quais são as faces de cada célula: como esse conversor só importa tetraedros, para cada célula lida do arquivo, o conversor gera suas semi-faces e semi-arestas com base no índice dos vértices de cada célula.

4.6.3. Conversor para o Formato 0-1

Arquivos com o formato 0-1 são gerados por um algoritmo de reconstrução tridimensional a partir de seções planares desenvolvido por Nonato [Non98]. Esse algoritmo é baseado na triangulação de Delaunay 3D e gera um modelo volumétrico descrito por uma malha de tetraedros a partir das seções planares de um objeto. A saída do algoritmo é um arquivo de tetraedros no formato, descrito na figura 4.10.

Figura 4.10: Exemplo de um arquivo no formato 0-1.

Nesse formato, cada rótulo 1 representa o início de um novo tetraedro. Após cada rótulo, seguem as coordenadas dos vértices do tetraedro.

Para gerar o sólido no SHF, esse conversor carrega os primeiros quatro vértices na estrutura, cria um tetraedro, atualiza as informações das mate-células e executa o operador de colagem. Em seguida, carrega os próximos quatro vértices e continua o procedimento. Caso algum vértice já esteja na estrutura, ele não será armazenado novamente.

4.6.4. Conversores de Exportação

Além dos conversores de importação de dados, também foram criados conversores para salvar os dados da estrutura SHF em diversos formatos. Alguns desses conversores exportam todos os tetraedros da estrutura para arquivos cujos formatos já foram descritos nos conversores para leitura de dados, como os conversores para HLF, Unstructured Grid ou o formato 0-1. Também foram criados conversores que exportam somente o bordo de objetos formado por tetraedros, para arquivos no formato Unstructured Grid ou PolyData do VTK.

Esses conversores foram criados antes da inclusão da estrutura no VTK. O próximo capítulo descreve em detalhes como foi realizada a inclusão da SHF no VTK e outras adições na biblioteca.

Capítulo 5 – Implementações Realizadas

5.1. Considerações Iniciais

Neste capítulo, veremos as implementações realizadas na biblioteca VTK para a incorporação da estrutura SHF, como filtros, fontes e mapeadores. Aspectos gerais importantes para modificação do código VTK são discutidos. Como o código da estrutura de dados está em constante desenvolvimento pelo grupo de computação gráfica do ICMC, a sua incorporação na biblioteca procurou manter sempre o seu estágio de evolução atual até o momento da escrita desta dissertação. Dentre as modificações ocorridas durante este trabalho de mestrado, podemos citar sua nova programação totalmente orientada a objetos, criação dos operadores topológicos, remoção de códigos e métodos desnecessários ou com finalidades específicas para outras aplicações, criação de conversores, separação dos conversores da estrutura e correção de erros, entre outros. Os tópicos tratados neste capítulo são: obtenção e compilação do código, aspectos gerais de adição de novas classes ao VTK, aspectos gerais de adição de um novo conjunto de dados ao VTK e de adição da SHF, e descrição das diversas classes criadas, com exemplos de utilização.

5.2. O Código Fonte VTK

O código fonte do VTK pode ser adquirido através da página da Kitware [Kit02] ou pelo CD-ROM contido no livro [Sch98]. Esse código fonte contém todos os objetos da biblioteca, na forma de arquivos de classes C++, separados em diretórios de acordo com sua funcionalidade. Alguns desses diretórios são: *Common* (que contém os objetos base do VTK, como o *vtkObject* e os conjuntos de dados), *Filtering* (diretório de filtros), *Graphics* (para gráficos 3D), *Imaging* (para imagens bidimensionais), *IO* (com leitores e mapeadores de arquivos) e *Rendering* (contendo as ferramentas de renderização).

O processo de compilação necessita de uma plataforma de controle. A Kitware desenvolveu sua própria plataforma, chamada CMake [Cma02]. A compilação gera bibliotecas dinâmicas (bibliotecas DLL no Windows ou SO no Linux) utilizadas pelos usuários-programadores do VTK.

Para o projeto deste mestrado foram utilizados a versão 4.0 do VTK e o software Visual C++ 6.0 para Windows. As novas classes desenvolvidas também foram compiladas e testadas no ambiente Linux, sem qualquer problema de importação.

5.3. Novas Classes no VTK

Esta seção apresenta aspectos da estrutura de implementação do VTK, importantes para programadores que desejam modificar o seu código.

A criação de novas classes no VTK deve manter a correta execução da rede de visualização, também chamada *pipeline* de visualização. Para que isso ocorra, os seguintes aspectos precisam ser considerados durante a implementação dessas classes:

- **Contagem de Referência:** É uma técnica de gerenciamento de memória do VTK usada para reduzir pedidos de memória. Parte da memória (nesse caso objetos) pode ser referenciada por mais de um objeto. O objeto referenciado mantém um contador de referências para ele. O método *Register()* incrementa o contador enquanto o método *UnRegister()* decrementa. Se o contador retornar zero, o objeto se deleta, retornando a memória ocupada para o sistema.

Para que o mecanismo de contagem de referência trabalhe corretamente, é preciso criar e referenciar objetos com contagens de referência de forma apropriada. Quando esse objeto é criado, seu contador contém inicialmente o valor um. Quando o objeto é referenciado por outro objeto, seu contador é incrementado em um. Esse contador é decrementado quando os objetos que o estão utilizando são deletados ou deixam de usá-lo. Então, o correto procedimento ao trabalhar com um objeto com contador de referência é criá-lo com o método *New()*, designá-lo para outro objeto ou usá-lo como necessário, e então liberá-lo com o método *Delete()*.

- **Tempo de Modificação:** A rede de visualização mantém o controle do tempo de modificação dos objetos. No VTK, o tempo de modificação de objetos é um valor inteiro sem sinal. Durante a execução da rede, os tempos de modificação são comparados entre os objetos. Se a entrada para um objeto de processo foi modificada, ou o próprio objeto foi modificado, então o objeto de processo precisa ser reexecutado. Na maioria dos casos, o controle do tempo de modificação é

automático pelos métodos herdados da superclasse, ou usando as macros padrão do VTK *Set/Get*.

Toda subclasse de *vtkObject* herda o método padrão *GetMTime()*. Esse método virtual geralmente fornece o comportamento apropriado. Contudo, se um objeto A depende de um objeto B, e B precisa ser modificado independentemente de A, então A precisa implementar seu próprio método *GetMTime()*. Nesse método, ambos os tempos de modificação de A e B precisam ser comparados, e o maior valor retornado. Logicamente, se A depende de mais de um objeto, então uma comparação é necessária entre o tempo de modificação de cada um.

- **Atualização da Rede:** A técnica de execução implícita implementada no VTK requer repetidas atualizações da rede quando os objetos de processos são modificados ou os dados de entrada mudam. Esse processo é implementado usando dois métodos: *Execute()* e *Update()*. O método *Update()* compara o tempo de modificação dos objetos com o tempo de modificação de suas entradas, e, se necessário, executa (via *Execute()*) o objeto de processo. Na maioria dos casos esse procedimento é automático através dos métodos herdados das superclasses.

O método *Update()* precisa ser sobrecarregado se o objeto tem mais do que uma entrada, mais do que uma saída, ou não usa as variáveis de instância padrão *Input/Output*. O procedimento básico é usar um método *Update()* para cada entrada, e então comparar os tempos de modificação entre os objetos de entrada e o tempo de execução dos filtros. Se o tempo de modificação de qualquer objeto de entrada for maior do que o tempo de execução do objeto local, ou o tempo de modificação do objeto local for maior do que seu tempo de execução, então o objeto local precisa reexecutar. Além disso, se o objeto de processo local liberou dados para memória, ele precisa ser reexecutado.

5.4. Produzindo um Novo Conjunto de Dados

Produzir uma nova estrutura de dados no VTK implica em criar uma subclasse de *vtkDataSet*. Para isso, é preciso projetar o novo conjunto de dados de modo a satisfazer a

interface abstrata especificada por *vtkDataSet*. Para a nova estrutura, são necessários novos filtros, fontes e mapeadores apropriados.

A integração da estrutura SHF no VTK parte dessa superclasse (figura 5.1). A classe principal da estrutura responsável pelo armazenamento de todas as informações referentes ao sólido, chamada “*Csolid*”, foi modificada para ser a nova herdeira de *vtkDataSet*. *Csolid* foi apresentada na figura 4.1 como o nó Sólido. Detalhes dos componentes da classe podem ser vistos no apêndice.

Figura 5.1: Classe responsável pelo sólido da estrutura SHF, “*Csolid*”, anexada ao VTK.

Algumas diferenças foram encontradas entre os conjuntos de dados do VTK e as estruturas topológicas como a SHF, o que dificultou o reaproveitamento de boa parte dos métodos do VTK. A principal diferença está na forma de armazenamento, discutidas a seguir.

Na estrutura SHF, para cada novo dado é alocado um novo objeto, e esse é referenciado dentro de uma lista simplesmente encadeada. Dependendo da utilização desse dado, ele poderá ser referenciado dentro de um outro tipo de estrutura. Por exemplo, os dados referentes à semi-aresta são armazenados em listas circulares, representando um ciclo orientado da semi-face. Já o conversor de importação para o formato 0-1 utiliza uma árvore binária para armazenar os vértices que são carregados, antes de armazená-los no sólido. Nesse tipo de arquivo, cada vértice aparece várias vezes, tornando necessário várias buscas para saber se o vértice já foi carregado. O uso de uma árvore binária deixa o processo de busca mais eficiente.

Para armazenamento dos elementos de suas estruturas de informação, a ferramenta VTK utiliza matrizes de dados contínuas como base para a maioria de seus conjuntos de dados. Segundo Schroeder et. al [Sch98], matrizes contínuas podem ser criadas, deletadas e percorridas mais rapidamente do que estruturas de dados alternativas, como uma lista encadeada ou matriz de ponteiros para estruturas. Isso é verdade quando o único objetivo das estruturas é visualização, pois não existe alteração dos dados durante o processo. Esse não é o caso do objetivo da SHF. No VTK as informações são acessadas com base em um identificador, que é um índice inteiro para um objeto do tipo matriz. Devido às necessidades de indexação de listas e ponteiros pela SHF, essas matrizes não foram reutilizadas.

Outro problema encontrado está na definição das células que compõem as malhas de dados. A SHF define células como uma coleção explícita de semi-faces, e podem ter qualquer forma tridimensional, embora os operadores topológicos, por enquanto, trabalhem apenas com tetraedros. As células no VTK são definidas especificando um *tipo* em combinação com uma lista ordenada de pontos. Essa lista ordenada, frequentemente referida como a *lista de conectividade (connectivity list)*, combinada com a especificação do tipo, define a topologia da célula implicitamente. As coordenadas x-y-z dos pontos definem a geometria da célula.

Definição:
 Tipo: hexaedro
 Conectividade: (8,10,1,6,21,22,5,7)

Figura 5.2 – Exemplo de uma célula hexaedro. A topologia é implicitamente definida pela ordem da lista de pontos. Adaptada de [Sch98]

A figura 5.2 mostra um exemplo de célula, o hexaedro. A lista ordenada é uma sequência de identificadores de pontos que faz índice para uma lista de coordenadas de pontos. Além do hexaedro, o VTK permite representar células dos tipos: vértice, polivértice (uma coleção de vértices), linha, polilinha, triângulo, faixa de triângulos, quadrilátero, pixel, polígono, tetraedro e voxel (um hexaedro com restrições geométricas). Outros tipos de células podem ser adicionados ao VTK, desde que tenham um tipo e uma ordem de apresentação dos pontos. O mapeador se encarrega de traduzir a lista de pontos em imagens, de acordo com o tipo recebido. Para tipos novos, o mapeador precisa ser reimplementado.

Para a SHF, esse esquema de armazenamento por conjuntos lineares não pode ser mantido devido ao requisito de indexação. Como a superclasse *vtkDataSet* trabalha com esses tipos de células, alguns de seus métodos virtuais não puderam ser implementados corretamente na subclasse, uma vez que as células tratadas por esse método precisam ser necessariamente do tipo utilizado no VTK. Por exemplo, o método *GetCell()* recebe um identificador como parâmetro e retorna um ponteiro para um objeto do tipo *vtkCell*, que especifica o comportamento das células do VTK. Uma forma de aproveitar esse método seria converter a célula do SHF para uma célula do tipo aceito, e retornar o ponteiro para essa nova célula. Porém, o algoritmo só funcionaria para células do tipo tetraedro ou hexaedro, e não seria possível manipular essa célula, pois seus valores não pertenceriam mais à estrutura. Seria, portanto, utilizada somente para exibição ou cópia dos dados.

Um modo de resolver os problemas de comunicação entre os dois tipos de estruturas seria implementar cada nó da SHF como herdeiro de uma classe correspondente no VTK. A

classe que representa células na SHF seria uma sub-classe de *vtkCell* e a classe que representa seus vértices uma sub-classe de *vtkPoints*. Porém, devido à diferença de armazenamento das estruturas, acabaríamos adquirindo métodos e dados, presentes nessas superclasses, que não teriam utilidade na nossa estrutura. Esse procedimento seria inviável, pois aumentaria demais o espaço ocupado na memória. Como objetos volumétricos são geralmente formados por uma grande quantidade de células e vértices, a memória desperdiçada aumentaria muito.

Portanto, a nova estrutura se comunica com as tradicionais estruturas do VTK somente através de filtros, mapeadores de escrita e leitores de arquivos, possuindo sua própria estrutura de armazenamento, distinta do estilo VTK. A implementação de cada nó procurou manter a sua estrutura de classe original, com pequenas modificações, descritas na próxima seção.

5.5. Implementações

As novas classes do VTK precisam ser incluídas em algum diretório padrão do código fonte ou algum diretório local permitido pela biblioteca. As classes da estrutura SHF foram incluídas no diretório principal chamado *Common*. Em cada diretório da biblioteca há um arquivo chamado “*CMakeLists.txt*” que contém uma lista com todos arquivos que serão compilados nesse diretório (*SOURCE_FILES*). É necessário incluir os nomes das novas classes nessa lista para que elas sejam compiladas.

Em C++, para exportar todos os dados públicos e funções membros de uma classe para algum arquivo de biblioteca dinâmica DLL, uma palavra chave precisa aparecer do lado esquerdo do nome da classe. Geralmente, a palavra chave para exportação é “*__declspec(dllexport)*”. No VTK, essa palavra chave foi substituída por uma macro do próprio VTK, chamada: “*VTK_(nome do diretório)_EXPORT*”, onde “*nome do diretório*” corresponde ao diretório onde foi incluída a classe. No caso da estrutura SHF, a palavra chave foi “*VTK_COMMON_EXPORT*”.

Com exceção da classe principal da estrutura, que é “filha” da classe *vtkDataSet*, as outras classes da estrutura continuaram herdeiras de classes “*templates*” próprias responsáveis pela organização dentro das listas. Essas classes, junto com suas “*templates*”,

foram exportadas para as bibliotecas dinâmicas, permitindo sua utilização pelos mapeadores e filtros da estrutura dentro do VTK.

Durante a implementação, quando criamos uma classe herdeira de uma superclasse do VTK, é necessário adicionar uma “macro de tipo”, chamada *vtkTypeMacro*, no arquivo cabeçalho (arquivo com extensão “.h”). Essa macro de controle do VTK determina as relações entre suas classes e subclasses. Como a classe *Csolid* faz parte do VTK, precisou receber a macro *vtkTypeMacro(Csolid, vtkDataSet)*.

5.6. Filtros

Conforme apresentado no capítulo 2, os elementos do *pipeline* (fontes, filtros e mapeadores) podem ser conectados em uma variedade de formas para criar a rede de visualização. Contudo, há um aspecto importante que devemos considerar quando tentamos montar estas redes: o *tipo* dessas conexões.

Tipo, nesse caso, se refere o tipo de dados que os objetos de processos recebem como entrada ou geram como saída. Por exemplo, uma fonte de uma esfera pode gerar como saída uma representação poligonal, que pode ser utilizada como entrada em outros objetos que recebem poligonais. A entrada para um objeto de processo precisa ser especificada corretamente para a operação ter sucesso. Quando algum tipo de saída de um objeto não combina com a entrada de outro objeto, usa-se um filtro para fazer essa conversão.

Terminada a implementação da nova estrutura no VTK, foram criados alguns filtros de conversão entre a SHF e outros conjuntos de dados. Esses filtros permitem a utilização da estrutura na rede de visualização composta por objetos VTK.

O primeiro filtro desenvolvido foi o *vtkSHFBoundaryToPolyDataFilter*. Ele recebe como entrada os objetos de dados do tipo *Csolid*, classe principal da estrutura SHF. Seu objetivo é extrair o bordo do modelo representado pela SHF e devolvê-lo como um objeto de dados do VTK que armazena malha de células planares (*vtkPolyData*).

Esse objeto filtro foi criado como “filho” da superclasse *vtkPolyDataSource*. *vtkPolyDataSource* é uma classe abstrata para subclasses que geram dados poligonais. Além das macros de tipo (*vtkMacroType*) e de entrada de dados (*SetInput*), o filtro também

recebeu o método de execução na rede, chamado *Execute()*. O método é executado pelo VTK durante a renderização.

Dentro do método de execução foi implementado o algoritmo de conversão dos dados da nova estrutura para dados poligonais. A principal finalidade desse filtro é permitir a exibição dos sólidos armazenados na SHF pelo VTK através da rede. Portanto, somente os polígonos presentes no bordo do objeto são convertidos pelo filtro. A saída gerada pelo filtro pode ser ligada a um mapeador de polígonos ou a qualquer objeto que aceite dados poligonais como entrada.

Um outro filtro foi desenvolvido para conversão de todas as células da nova estrutura, chamado *vtkSHFToUnstructuredGridFilter*. Esse filtro converte somente células tetraédricas para células do tipo *VTK_TETRA* (células tetraédricas do VTK), que são armazenadas em uma malha não estruturada, do tipo *vtkUnstructuredGrid*. Dessa forma, foi utilizada como “pai” desse filtro a superclasse *vtkUnstructuredGridSource*. Essa superclasse cria uma malha não estruturada durante sua inicialização e possui macros de saídas apropriados (*GetOutput*).

Durante a execução, todos os vértices são armazenados em uma matriz de coordenadas, e cada célula, em uma matriz formada pelo identificador da célula e pelos identificadores dos quatro pontos que formam cada tetraedro.

5.7. Fontes

Objetos tipo fonte inicializam a rede de visualização. Eles não têm outros objetos como entrada, e freqüentemente recebe seus dados de arquivos externos ou os criam através de variáveis de instância. Fontes geram um ou mais conjuntos de dados como saídas.

Na estrutura, alguns exemplos desses objetos fontes são os objetos de importação de dados, chamados conversores. Para a inclusão desses conversores no VTK, foi criada uma classe abstrata, chamada *vtkSHFSource*, que gera dados do tipo *Csolid*. Esse fonte é herdeiro da classe *vtkSource*. *vtkSource* é uma classe abstrata que especifica o comportamento e a interface dos objetos fontes.

Com esse fonte foi possível criar um novo tipo de filtro de conversão do VTK, chamado *vtkUnstructuredGridToSHFFilter*. Esse filtro faz o inverso do filtro

vtkSHFToUnstructuredGridFilter, convertendo os dados do tipo malha não estruturada (*vtkUnstructuredGrid*) para a nova estrutura. A execução desse filtro trabalha apenas com as células tetraédricas da malha.

5.8. Mapeadores

Mapeadores são objetos de processos que terminam a rede de visualização, mapeando os dados de entrada para bibliotecas gráficas ou escrevendo os dados em discos.

O primeiro mapeador construído durante este trabalho utilizava o mesmo princípio do mapeador de conjunto de dados (*vtkDataSetMapper*), convertendo o bordo do objeto para dados poligonais e chamando o mapeador *vtkPolyDataMapper*. Dados poligonais fornecem uma boa interface para bibliotecas gráficas externas. Entretanto, há um custo na realização desse tipo de mapeamento, pois, além da perda de tempo em converter os dados duas vezes (de *Csolid* para *vtkPolyData* e desse para biblioteca gráfica), ele aloca mais memória para armazenar as primitivas do objeto antes de renderizar.

Um novo mapeador foi, em seguida, criado para exportar os dados da nova estrutura diretamente para a biblioteca gráfica OpenGL, a partir da superclasse *vtkMapper*. A classe *vtkMapper* é uma classe abstrata para especificar interface entre dados e primitivas gráficas. Esse novo mapeador é constituído de duas classes: uma classe abstrata geral *vtkSHFMapper*, que deve ser utilizada como superclasse por qualquer mapeador que trabalhar com dados do tipo *Csolid*; e uma subclasse específica para OpenGL, *vtkOpenGLSHFMapper*. O usuário não precisa saber qual biblioteca gráfica utilizar, ele simplesmente cria um objeto *vtkSHFMapper* e o VTK retorna o mapeador adequado. Essa escolha é feita no método *New()* da classe abstrata *vtkSHFMapper*. Entretanto, como só trabalhamos com OpenGL no projeto, o método *New()* simplesmente aloca um objeto *vtkOpenGLSHFMapper* e retorna para o usuário como se fosse um *vtkSHFMapper*.

A renderização no VTK ocorre através do método *Render()*, implementado tanto na classe *vtkSHFMapper*, como nas classes *vtkPolyDataMapper* e *vtkDataSetMapper*. A função principal desse método é invocar o método de renderização de suas subclasses, chamado *RenderPiece()*. Esse método deve ser implementado em todas as subclasses específicas de cada biblioteca, como a subclasse *vtkOpenGLSHFMapper*, e é o responsável

pela conversão das semi-faces de bordo da estrutura para uma lista de dados poligonais da biblioteca. O método virtual *ComputeBounds()* da classe *vtkDataSet*, implementado em todos os conjuntos de dados, calcula o limite espacial do domínio de dados e é utilizado pelo mapeador para definir o posicionamento do objeto na janela de visualização, precisando, portanto, estar funcionando corretamente.

Algumas opções de renderização foram sendo incrementadas no mapeador, permitindo a exibição do objeto na forma de arame (*wireframe*), o mapeamento das semi-faces do interior e a representação de dados escalares utilizando uma tabela de cores.

A representação por cores no mapeador da SHF é acionada pelo método *SetScalarModeToUsePointData()* da classe *vtkMapper*. O mapeador converte os valores escalares adicionados em cada vértice em um índice de cor da tabela *LookupTable* (*vtkLookupTable*) do VTK, e a cor correspondente é aplicada ao vértice durante a renderização. Essa tabela pode ser criada pelo usuário e adicionada ao mapeador com as cores desejadas. Caso nenhuma tabela seja adicionada, o mapeador criará a tabela com as cores padrão do VTK. Os dados escalares precisam estar armazenados no campo *data* de cada vértice (através do método *Set_data(void *)*).

A figura 5.3 traz uma visão geral da integração da estrutura na organização de classes do VTK.

Figura 5.3: Visão geral da estrutura SHF incorporada ao VTK.

5.9. Exemplos de Códigos

Seguem abaixo três códigos exemplos que demonstram a utilização da biblioteca VTK com a SHF. Os dois primeiros códigos exibem um mesmo tetraedro na tela. No primeiro, o tetraedro é criado na própria estrutura de dados. No segundo código, ele é criado em uma malha não estruturada e filtrado para a estrutura. As figuras 5.4(a) e 5.4 (b) mostram o tetraedro gerado pelos dois primeiros algoritmos e sua visão de arame. O terceiro exemplo demonstra o uso de escalares em um sólido formado por vários tetraedros, onde o valor escalar de cada vértice corresponde ao valor da sua coordenada X (figura 5.4(c)).

No próximo capítulo é apresentada uma aplicação desenvolvida durante este trabalho para reconstrução de objetos tridimensionais e armazenando o sólido resultante na estrutura SHF.

Criando um Tetraedro

```
#include "vtkRenderWindow.h"
#include "vtkRenderWindowInteractor.h"
#include "vtkSHFMapper.h"
#include "Cvertex.h"
#include "Ccell.h"
#include "Csolid.h"

// Esse programa cria um tetraedro na estrutura SHF e exibe-o na janela do VTK.
void main( )
{ // Coordenadas dos vertices do tetraedro
  int tetra[4]={0,1,2,3};
  int coords[4][3]={0,0,0},{10,0,0},{5,10,0},{5,5,10}};

  // Criacao do tetraedro
  // Criação do sólido
  Csolid *novo=Csolid::New();
  // Criação dos vértices
  for (int i=0; i<4; i++)
  { Cvertex *v=new Cvertex(coords[i][0],coords[i][1],coords[i][2]);
    // Especificação do identificador do vértice e adição ao sólido
    v->Set_id(tetra[i]);
    novo->Add_vtx(v);
  }
  // Criação da célula
  Ccell *c=new Ccell();
  c=novo->Make_Cell(tetra,0);

  // Colagem da célula
  // Atualização das estrelas dos vértices da nova célula
  novo->Set_StarVertex(c);
  // Utilização dos operadores topológicos para colagem
  novo->Set_Moperator(c);

  // Mapeador da SHF
  vtkSHFMapper *novoMapper=vtkSHFMapper::New();
  novoMapper->SetInput( novo );

  // Criação da cena
  // Ator
  vtkActor *novoActor=vtkActor::New();
  novoActor->SetMapper( novoMapper );
  // Cena
  vtkRenderer *ren1=vtkRenderer::New();
  ren1->SetBackground( 1.0, 1.0, 1.0 );
  ren1->AddActor( novoActor );
  // Janela de visualização
  vtkRenderWindow *renWin=vtkRenderWindow::New();
  renWin->AddRenderer( ren1 );
  renWin->SetSize( 300, 300 );

  // Render e interação
  vtkRenderWindowInteractor *iren=vtkRenderWindowInteractor::New();
  iren->SetRenderWindow(renWin);
  renWin->Render();
  iren->Start();

  // Liberação de memória
  novo->Delete();
  novoMapper->Delete();
  novoActor->Delete();
  ren1->Delete();
  renWin->Delete();
  iren->Delete();
}
```

Filtro de UnstructuredGrid para SHF

```
#include "vtkRenderWindow.h"
#include "vtkRenderWindowInteractor.h"
#include "vtkSHFMapper.h"
#include "vtkUnstructuredGrid.h"
#include "vtkPoints.h"
#include "vtkUnstructuredGridToSHFFilter.h"

// Esse programa cria um tetraedro em uma malha não estruturada e o converte para SHF.
void main( )
{ // Criação da UnstructuredGrid
  int tetra[4]={0,1,2,3};

  vtkUnstructuredGrid *vug=vtkUnstructuredGrid::New();
  vtkPoints *np=vtkPoints::New();

  np->InsertPoint(0,0,0,0);
  np->InsertPoint(1,10,0,0);
  np->InsertPoint(2,5,10,0);
  np->InsertPoint(3,5,5,10);
  vug->SetPoints(np);
  vug->InsertNextCell(VTK_TETRA,4,tetra);

  // Conversão para SHF
  vtkUnstructuredGridToSHFFilter *ugts=vtkUnstructuredGridToSHFFilter::New();
  ugts->SetInput(vug);

  // Mapeador da SHF
  vtkSHFMapper *novoMapper=vtkSHFMapper::New();
  novoMapper->SetInput( ugts->GetOutput() );

  // Criação da cena
  // Ator
  vtkActor *novoActor=vtkActor::New();
  novoActor->SetMapper( novoMapper );
  // Cena
  vtkRenderer *ren1=vtkRenderer::New();
  ren1->SetBackground( 1.0, 1.0, 1.0 );
  ren1->AddActor( novoActor );
  // Janela de Visualização
  vtkRenderWindow *renWin=vtkRenderWindow::New();
  renWin->AddRenderer( ren1 );
  renWin->SetSize( 300, 300 );

  // Render e interação
  vtkRenderWindowInteractor *iren=vtkRenderWindowInteractor::New();
  iren->SetRenderWindow(renWin);
  renWin->Render();
  iren->Start();

  // Liberação de memória
  vug->Delete();
  np->Delete();
  ugts->Delete();
  novoMapper->Delete();
  novoActor->Delete();
  ren1->Delete();
  renWin->Delete();
  iren->Delete();
}
```

Sólido colorido

```
#include "vtkRenderWindow.h"
#include "vtkRenderWindowInteractor.h"
#include "vtkSHFMapper.h"
#include "Cvertex.h"
#include "Ccell.h"
#include "Csolid.h"

// Esse programa cria um sólido na forma de um tetraedro, atribui um escalar para cada
// vértice com o valor de sua coordenada X e utiliza cores para representá-lo

// Procedimento que cria um dos lados (t) do sólido e armazena em *novo
void LadoSólido(int t,Csolid *novo)
{ // Coordenadas dos tetraedros criados
  int tetra[4];
  int coords[3][4][3]={{{0,0,0},{1,0,0},{1,1,0},{1,0,1}},{0,0,0},{0,1,0},{1,1,0},{1,0,1}},
  ,{{0,0,0},{1,0,0},{0,-1,1},{1,0,1}}};
  int cont=0,e=0,maxJ=10,i,j,k;

  // Criação dos tetraedros. A cada repetição é criado um no tetraedro no sólido.
  for (j=1; j<maxJ; j++)
  { e=0;
    for (k=0; k<j*2-1; k++)
    { for (i=0; i<4; i++)
      { // Criação do vértice. Cada lado do sólido (t) possui coordenadas diferentes.
        Cvertex *v;
        switch(t)
        { case 0 : v=new Cvertex(coords[e][i][0]+j,coords[e][i][1]+k/2,coords[e][i][2]);
            break;
          case 1 : v=new Cvertex(-coords[e][i][2]+maxJ,coords[e][i][1]+k/2,
            -coords[e][i][0]+maxJ-j); break;
          case 2 : v=new Cvertex(coords[e][i][0]+j,coords[e][i][2],coords[e][i][1]+k/2);
            break;
          case 3 : v=new Cvertex(coords[e][i][0]+j,coords[e][i][1]+j-1-k/2,
            coords[e][i][2]+k/2); break;
        }
        // Especificação do identificador do vértice
        v->Set_id(cont);
        // Atribuição de um escalar com o valor de sua coordenada X
        float *dado=new float(v->Get_x());
        v->Set_data(dado);
        tetra[i]=cont;
        cont++;
        // Adição do vértice no sólido
        novo->Add_vtx(v);
      }
      // Criação da célula
      Ccell *c=new Ccell();
      c=novo->Make_Cell(tetra,0);
      // Atualização da estrela dos vertices
      novo->Set_StarVertex(c);
      // Utilização dos operadores topológicos para colagem
      novo->Set_Moperator(c);
      if (t<3) e=(e+1)%2; else e=(e+2)%4;
    }
  }
}

// Programa principal da criação e exibição do sólido na janela do VTK.
void main( )
{
  // Criação do sólido.
  Csolid *novo=Csolid::New();
  LadoSólido(0,novo);
  LadoSólido(1,novo);
  LadoSólido(2,novo);
  LadoSólido(3,novo);
}
```



```

// Mapeador da SHF (com exibicao de cores)
vtkSHFMapper *novoMapper=vtkSHFMapper::New();
novoMapper->SetInput( novo );
novoMapper->SetScalarModeToUsePointData();

// Criação da cena
// Ator
vtkActor *novoActor=vtkActor::New();
novoActor->SetMapper( novoMapper );
// Cena
vtkRenderer *ren1=vtkRenderer::New();
ren1->SetBackground( 1.0, 1.0, 1.0 );
ren1->AddActor( novoActor );
// Janela
vtkRenderWindow *renWin=vtkRenderWindow::New();
renWin->AddRenderer( ren1 );
renWin->SetSize( 300, 300 );

// Render e interação
vtkRenderWindowInteractor *iren=vtkRenderWindowInteractor::New();
iren->SetRenderWindow(renWin);
renWin->Render();
iren->Start();

// Liberação da memória ocupada pelos esclacares
iterator<Cvertex> iv;
for (iv=novo->Begin_vertex(); iv!=novo->End_vertex(); ++iv)
{ void *f=iv->Get_data(); delete f; iv->Set_data(NULL); }

// Liberação da memória
novo->Delete();
novoMapper->Delete();
novoActor->Delete();
ren1->Delete();
renWin->Delete();
iren->Delete();
}

```

(a)

(b)

(c)

Figura 5.4: (a) Tetraedro gerado pelos dois primeiros algoritmos, (b) visão de arame do tetraedro, (c) sólido gerado pelo terceiro algoritmo com representação de escalares por cores.

Capítulo 6 – Aplicação: Reconstrução 3D

6.1. Considerações Iniciais

Veremos neste capítulo o uso da SHF anexada ao VTK em uma aplicação de reconstrução tridimensional.

6.2. Módulo de Reconstrução 3D

No presente projeto, foi criada uma aplicação para testar o funcionamento da nova estrutura do VTK e de seus operadores topológicos de colagem de células em uma aplicação real. Essa aplicação, que posteriormente foi adicionada como um novo módulo do VTK, chamado *vtkSHFReconstruction*, reconstrói objetos tridimensionais formados por uma seqüência de fatias. Cada fatia corresponde a uma imagem bidimensional de parte do sólido. Essas imagens podem vir de arquivos ou simuladas. O módulo trabalha como um filtro que converte um conjunto de malhas estruturadas (*vtkStructuredPointsCollection*) em objetos do tipo *Csolid*.

Os arquivos de imagens utilizados pelo reconstrutor geralmente são de dois tipos: vários arquivos com figuras bidimensionais representando cada fatia do sólido, ou um único arquivo contendo uma matriz tridimensional com todos os valores escalares do objeto.

O VTK possui leitores para diversos formatos de arquivos de imagens bidimensionais (por exemplo, para o formato BMP – *Windows Bitmap*), que geram como saída uma malha de pontos estruturados (*vtkStructuredPoints*). Uma malha de pontos estruturados no VTK pode ter duas ou três dimensões, cujos dados podem ser salvos ou lidos através de mapeadores de escrita ou leitores (*vtkStructuredPointsWriter* e *vtkStructuredPointsReader* respectivamente) específicos para essa malha.

Todas as malhas que serão utilizadas na reconstrução devem ser armazenadas em um objeto do tipo *vtkStructuredPointsCollection*, que será lido pelo reconstrutor. No caso do uso de uma matriz tridimensional, essa matriz também deve ser armazenada em um objeto *vtkStructuredPointsCollection*.

Formato de Arquivo de Entrada

Como o VTK continha apenas um leitor para arquivos PNM (*Portable Anymap Utilities*) [Rub00] no modo binário, foi criado um novo leitor para o modo ASCII. Esse leitor, subclasse de *vtkDataReader*, gera como saída uma malha estruturada. O formato PNM opera com os formatos PBM (*Portable Bitmap Utilities*), utilizado para imagens monocromáticas, PGM (*Portable Graymap Utilities*), que manipula imagens com tons de cinza, e PPM (*Portable Pixmap Utilities*) para imagens coloridas. Esses três formatos foram projetados para serem simples. O início é definido com um cabeçalho e os dados da figura seguem imediatamente depois. O cabeçalho do arquivo é sempre escrito em ASCII e os itens de dados são separados por espaços em branco. Os dados de informações dos pixels podem ser escritos em ASCII ou em formato binário.

Esse filtro trabalha apenas com imagens em tons de cinza. O cabeçalho para esse formato contém inicialmente a palavra chave “P2”, indicando o tipo de arquivo (ver figura 6.1), a largura em pixels, a altura em pixels e o valor correspondente à cor branca. A cor preta sempre corresponderá ao valor zero, e os valores intermediários aos tons de cinza.

P.Chave	Tipo	P.Chave	Tipo
P1	ASCII, monocromática	P4	Binário, monocromática
P2	ASCII, tons de cinza	P5	Binário, tons de cinza
P3	ASCII, colorida	P6	Binário, colorida

Figura 6.1: Tabela de formatos para arquivos PNM.

O filtro armazena os dados da figura em uma matriz de inteiros (*vtkIntArray*), que é passada para a malha estruturada através do método *GetPointData()* → *SetScalars(array)*. A dimensão da malha é indicada pelo método *SetDimensions(x,y,z)*.

Parâmetros

Além das imagens, o módulo de reconstrução precisa de parâmetros de controle, informados através dos comandos *Set* (por exemplo: *SetFinal(10)*). Esses parâmetros são:

o número da fatia inicial (*Initial*) e final (*Final*) que serão reconstruídas, um fator de refinamento (*RefFactor*), cujo valor inicial é 10, uma variável de escala (*ScaleZ*), com valor inicial igual a 1, e um intervalo de valores escalares que será aceito pelo reconstrutor (*ValueMin* e *ValueMax*). Esse intervalo deve conter os valores dos pixels da região da imagem que se deseja reconstruir. O valor inicial dos dois parâmetros é zero.

Algoritmo

Como se trata de uma nova classe do VTK, a execução do algoritmo de reconstrução ocorrerá somente durante o processo de renderização. Esse algoritmo trabalha com duas fatias vizinhas por vez, preenchendo com cubos e tetraedros os espaços entre elas. Cada fatia é afastada de sua vizinha no espaço R^3 por uma distância igual ao valor da variável *RefFactor*. O algoritmo interpreta como vizinha aquela fatia que estiver armazenada logo em seguida à outra no conjunto de pontos estruturados. O algoritmo também trabalha com malhas estruturadas tridimensionais. Nesse caso, cada fatia é representada pelos pontos no plano X-Y, e sua vizinha está no plano paralelo imediatamente superior no eixo Z (ver figura 6.2).

Figura 6.2: Duas fatias vizinhas. Os pixels da imagem (valor 0) são representados por pontos pretos.

Uma malha de cubos é gerada entre as fatias (figura 6.3). Cada cubo possui comprimento igual à distância entre as fatias (*RefFactor*) e é formado por seis tetraedros. A figura 6.4 mostra a posição de cada tetraedro dentro do cubo.

Figura 6.3: Exemplo de uma malha de cubos inserida entre duas fatias, com RefFactor igual a 5. A cada 5 pixels é colocado um vértice do cubo.

Figura 6.4: Cubo formado por 6 tetraedros. As cores indicam cada lado do cubo.

Sempre que o vértice de um tetraedro localiza-se próximo a um pixel de uma fatia cujo valor está dentro do intervalo de valores aceitos (*ValueMin* e *ValueMax*), esse tetraedro é “colado” ao sólido, e armazenado na estrutura SHF. Esse sólido é alocado antes da execução do reconstrutor pela classe pai *vtkSHFSource*. Caso o pixel esteja fora desse intervalo de valores, o tetraedro é eliminado.

Quanto menor o valor da variável de refinamento, maior a quantidade de detalhes no objeto, porém aumentará a memória utilizada devido ao grande aumento de tetraedros armazenados. Além disso, o objeto ficará menor pois diminuirá a distância entre as fatias. Essa distância pode ser controlada alterando o valor da variável de escala do eixo Z (*ScaleZ*). O valor da coordenada Z de cada vértice será multiplicado por essa escala durante a reconstrução.

Outra forma de indicar qual tetraedro pertence ao sólido é através da média aritmética dos valores dos pixels próximos aos quatro valores do tetraedro. Se a média estiver dentro do intervalo, o tetraedro será incluído. Para trabalhar com a média aritmética, foi criada uma nova variável de controle, que é habilitada pelo método *AverageOn()*.

Exemplo: Coração

A primeira reconstrução feita foi de um coração, cujas fatias estavam armazenadas em arquivos no formato PNM, no modo ASCII. As imagens carregadas continham apenas dois valores: 0 e 255. O coração está representado pelos pixels de valor 0. Portanto, o valor mínimo aceito pelo intervalo de escalares (*ValueMin*) será 0, e o valor máximo (*ValueMax*) pode ser qualquer valor menor do que 255, pois esse valor não pertence à imagem. A figura 6.5 mostra o exemplo do coração reconstruído com fator de refinamento 10, seguido por uma visão de arame desse último.

Figura 6.5: Coração reconstruído.

Outros Exemplos

Abaixo seguem outros exemplos reconstruídos. A figura 6.6(a) mostra um dente reconstruído e sua visão de arame. O intervalo de escalares utilizado reconstruiu apenas o dente, formando uma cavidade na região do nervo. A figura 6.6(b) mostra um mamão e sua visão de arame. Os valores escalares do caroço não foram utilizados, gerando uma

cavidade. A figura 6.6(c) exibe o dente e o mamão pela técnica de Rendering Volumétrico Direto do VTK. Esses dois objetos (mamão e dente) estavam armazenados em arquivos no formato malha estruturada tridimensional do VTK (*vtkStructuredPoints*). Seus parâmetros de controle estão listados na figura 6.7.

Figura 6.6: (a) Reconstrução de um dente, (b) reconstrução de um mamão, (c) DVR dos dois objetos.

Parâmetro \ Objeto	Coração	Mamão	Dente
ValueMin	0	34745	6310
ValueMax	10	48280	56610
RefFactor	10	3	10
Formato Arquivo	PNM	VTK	VTK

Figura 6.7: Tabela de parâmetros e tipos de arquivo fonte

Remoção de Cavidades

Dependendo do intervalo escolhido, podem surgir uma ou mais cavidades nos objetos reconstruídos. No caso da reconstrução do mamão, sete cavidades surgiram, incluindo a região do caroço. Para eliminar as cavidades pequenas sem alterar o intervalo escolhido, foi implementada no módulo uma opção de remover cavidades, preenchendo-a com tetraedros. Esse algoritmo é iniciado logo após a reconstrução, caso a variável *RemoveCavity* seja maior que 0. Essa variável especifica o número de cavidades a ser removida.

O algoritmo de eliminação primeiramente procura pela cavidade de menor volume. Cavidades são preenchidas da menor para a maior. Em seguida, o algoritmo começa a “colar” tetraedros nas paredes da cavidade, utilizando tetraedros de mesmo tamanho dos tetraedros utilizados na reconstrução, isto é, com altura, comprimento e largura iguais ao valor da variável *RefFactor*. Se, durante a colagem, o algoritmo gerar uma nova cavidade, a variável de controle será acrescida para que as duas cavidades sejam eliminadas.

A figura 6.8 mostra a imagem do mamão reconstruído, após eliminadas todas as cavidades, incluindo a cavidade formada pela extração do caroço. A primeira cavidade foi removida utilizando somente um tetraedro. A figura 6.9 apresenta um diagrama com os novos módulos na biblioteca.

Figura 6.8: Mamão reconstruído sem cavidades na visão de arame.

Figura 6.9: Módulos do reconstrutor.

Alisamento

Devido à aparência quadriculada de alguns objetos reconstruídos, foi adicionado um filtro no reconstrutor que faz um “alisamento” de suas componentes de bordo. Esse filtro altera o valor da coordenada de cada vértice do bordo com a média aritmética das coordenadas dos vértices do bordo ligados àquele através de uma semi-aresta. A média é calculada sempre com os valores antigos dos vértices, ou seja, os valores dos vértices só serão atualizados após a realização de todos os cálculos. A figura 6.10 mostra o exemplo do coração, reconstruído com fator de refinamento igual a 5 e escala igual a 2, após o uso do filtro.

Figura 6.10: Coração após filtro de alisamento.

6.3. Exemplos de Códigos

Nesta seção são apresentados alguns exemplos de códigos utilizados no reconstrutor. O primeiro é um procedimento que inclui um tetraedro no sólido. Ele é chamado sempre que os valores escalares dos pixels próximos a seus vértices estão dentro do intervalo aceito. O exemplo seguinte é um trecho do código de remoção de cavidades que identifica os limites espaciais de uma célula. Conhecendo esses limites, é possível determinar qual célula poderá ser colada a essa durante a remoção de cavidades. Esse exemplo demonstra como percorrer uma célula através de suas semi-arestas. O último exemplo mostra como usar o reconstrutor em um programa.

Incluir Tetraedro

```
// Procedimento que inclui um tetraedro no sólido utilizando uma matriz de coordenadas.

void IncludeTetra(int c[4][3],Csolid *s,int t[4],int colorido)
{ // Parâmetros:
  // c[4][3] - coordenadas do tetraedro
  // s - sólido
  // t[4] - escalares dos 4 vértices do tetraedro
  // colorido - se verdadeiro, escalares são anexados aos vértices

  int tetra[4],i;
  Cvertex *v,*vf;
  Ccell *cell;

  for (i=0;i<4; i++)
  { // Criação do vértice
    v = new Cvertex(c[i][0],c[i][1],c[i][2]);
    // Verificação da existência de um vértice com mesmas coordenadas
    vf = FindCoordinates(s->Get_lvertex(),v,NULL);
    if (vf!=NULL)
    { // Utilização do vértice antigo, caso existe um vértice com mesmas coordenadas
```

```

        tetra[i]=vf->Get_id();
        delete v;
    }
    else
    { // Inclusão do novo vértice, caso não existe outro com mesmas coordenadas
      // Especificação do identificador do vértice
      v->Set_id(s->Get_nvertex());
      if (colorido)
      { // Atribuição do escalar (t)
        float *a1=new float;
        *a1=(float)t[i];
        v->Set_data(a1);
      }
      // Adição do vértice no sólido
      s->Add_vtx(v);
      tetra[i] = v->Get_id();
    }
  }
  // Criação da nova célula
  cell = s->Make_Cell(tetra,0);
  // Atualização das estrelas dos vértices
  s->Set_StarVertex(cell);
  // Utilização dos operadores topológicos para colagem
  s->Set_Moperator(cell);
};

```

Limites de uma Célula

```

// Esse código calcula os limites espaciais de uma célula.
// Variáveis
double x[5],y[5],z[5];
double xmin,xmax,ymin,ymax,zmin,zmax;
int t=0;
// Iterator de semi-aresta, para percorrer a semi-face.
iterator<Chalfedge> he;
// Chalface *hf. Variável já declarada. Contém uma semi-face da célula analisada

// Atribuição de valores iniciais às variáveis limites
xmin=ymin=zmin=VTK_LARGE_FLOAT;
xmax=ymax=zmax=-VTK_LARGE_FLOAT;

// Repetição para cada semi-aresta da semi-face
for (he=hf->Begin_halfedge(); he!=hf->End_halfedge(); ++he)
{ // Capturação das coordenadas dos vértices
  x[t]=he->Get_vtx()->Get_x();
  y[t]=he->Get_vtx()->Get_y();
  z[t]=he->Get_vtx()->Get_z();
  // Atualização das variáveis limites
  if (x[t]<xmin) xmin=x[t];
  if (x[t]>xmax) xmax=x[t];
  if (y[t]<ymin) ymin=y[t];
  if (y[t]>ymax) ymax=y[t];
  if (z[t]<zmin) zmin=z[t];
  if (z[t]>zmax) zmax=z[t];
  t++;
}
// Capturação do último vértice da célula na aresta anterior da mate-célula.
he=hf->Begin_halfedge();
x[t]=he->Get_matecell()->m_pPrev->Get_vtx()->Get_x();
y[t]=he->Get_matecell()->m_pPrev->Get_vtx()->Get_y();
z[t]=he->Get_matecell()->m_pPrev->Get_vtx()->Get_z();
if (x[t]<xmin) xmin=x[t];
if (x[t]>xmax) xmax=x[t];
if (y[t]<ymin) ymin=y[t];
if (y[t]>ymax) ymax=y[t];
if (z[t]<zmin) zmin=z[t];
if (z[t]>zmax) zmax=z[t];

```

Usando o Reconstrutor

```
// Nome do arquivo
char nome[50];
sprintf(nome, "mamao.vtk");

// Leitura do arquivo contendo pontos estruturados
vtkStructuredPointsReader mamao=vtkStructuredPointsReader::New();
mamao->SetFileName(nome);

// Coleção de pontos estruturados (tipo utilizado pelo reconstrutor)
vtkStructuredPointsCollection *cole=vtkStructuredPointsCollection::New();
cole->AddItem(mamao->GetOutput());

// Criação e configuração do reconstrutor
vtkSHFReconstruction *hea=vtkSHFReconstruction::New();
hea->SetSlices(cole);
hea->SetValueMin(34745);
hea->SetValueMax(48280);
hea->SetRefFactor(3);
hea->AverageOff();
// Armazenamento dos escalares nos vértices
hea->ColorOn();
// Remoção de 6 cavidades
hea->SetRemoveCavity(6);

// Mapeador da SHF (com cores para os escalares)
vtkSHFMapper *novoMapper=vtkSHFMapper::New();
novoMapper->SetInput( hea->GetOutput() );
novoMapper->SetScalarModeToUsePointData();
```

O próximo capítulo apresenta as contribuições, conclusões e trabalhos futuros associados a esta dissertação.

Capítulo 7 – Conclusões

7.1. Considerações Iniciais

Este trabalho demonstrou a utilização da nova estrutura de dados topológica SHF em visualização científica, através da sua incorporação à biblioteca VTK, realizando sobre seus dados operações de renderização por superfície e visualização de dados escalares utilizando cores.

O trabalho envolveu a criação de novos módulos permitindo a utilização da SHF na rede de visualização do VTK. Os filtros implementados permitem que diversos tipos de dados da biblioteca sejam utilizados pela estrutura, e que os dados da estrutura sejam utilizados por outros módulos do VTK.

Uma aplicação de reconstrução tridimensional foi criada a fim de testar o funcionamento da nova estrutura na biblioteca. A implementação de um mapeador para OpenGL garantiu a visualização dos resultados na janela de visualização do VTK. Comparações entre os dados reconstruídos com essa aplicação e sua imagem exibida por DVR demonstraram o bom funcionamento de todos os módulos envolvidos no processo de reconstrução.

A utilização da estrutura em visualização científica mostrou-se viável, embora a biblioteca VTK não esteja preparada para suportar uma estrutura topológica projetada para modelagem. Seus objetos de dados são voltados apenas para visualização, utilizando matrizes para o armazenamento de seus conjuntos de dados, o que dificulta a incorporação das listas encadeadas da SHF.

7.2. Contribuições do Trabalho

Dentre as contribuições apresentadas por este trabalho, destacam-se:

- Participação na evolução da nova estrutura de dados SHF para uma estrutura orientada a objetos.
- Descrição do funcionamento e do modo de implementação da biblioteca de visualização VTK, focalizando nos conjuntos de dados suportados pela

biblioteca. Este trabalho serve como ponto de partida para que novos módulos sejam desenvolvidos e adicionados à biblioteca, visando prover novos recursos e permitir melhor desempenho.

- Incorporação da nova estrutura na biblioteca VTK, envolvendo a criação de diversos filtros, fontes e mapeadores apropriados.
- Criação de uma aplicação de reconstrução tridimensional utilizando a nova estrutura incorporada ao VTK, validando o uso da SHF em projetos de visualização.

7.3. Sugestão para Trabalhos Futuros

No âmbito do projeto *PowerVis*, em andamento no grupo de Computação Gráfica e Processamento de Imagens do ICMC-USP foram desenvolvidas técnicas para segmentação, reconstrução, visualização, interação e sonificação de dados, com acesso remoto pela internet (<http://www.lcad.icmc.sc.usp.br/~powervis>). O sistema *PowerVis* usa como base a biblioteca VTK, expandindo sua funcionalidade nos diversos aspectos do desenvolvimento do projeto. Todos os módulos do sistema, entretanto, utilizam as estruturas de dados do próprio VTK, que possuem definição pobre da topologia dos objetos que representam.

A substituição da biblioteca VTK, permitindo o uso da nova estrutura, deverá fazer com que muitas das operações que atualmente são realizadas no *PowerVis* atinjam níveis de eficiência maiores, em virtude das características de indexação da SHF. Para isso será necessário adaptar os módulos individuais do *PowerVis* para trabalhar com essa estrutura.

Também é possível revisar muitos dos objetos de processo do VTK para que eles trabalhem sobre a SHF ao invés dos tipos originários do VTK. Para ilustrar essa possibilidade, no momento está sob desenvolvimento o filtro `vtkGlyph3D` que funcione sobre a SHF.

Apêndice – Descrição dos arquivos e das Classes

Lista de arquivos:

Árvores e Listas:

BinTree.h
Iterators.h
List.h
ListCircle.h

Arquivos auxiliares:

Classes.cpp
Classes.h
Compare.cpp
Compare.h
Conversor.cpp
Conversor.h
HalfFace.h
HalfFaceGlobal.h
Operators.cpp
Operators.h
Utilities.h
Shfio.cpp
Shfio.h

Estrutura SHF:

Cboundarycp.cpp
Cboundarycp.h
Cboundaryface.cpp
Cboundaryface.h
Ccell.cpp
Ccell.h
Chalfedge.cpp
Chalfedge.h
Chalfface.cpp
Chalfface.h
Csingedge.cpp
Csingedge.h
Csingvertex.cpp
Csingvertex.h
Cvertex.cpp
Cvertex.h

Filtros VTK:

vtkSHFBoundaryToPolyDataFilter.cxx
vtkSHFBoundaryToPolyDataFilter.h
vtkSHFToUnstructuredGridFilter.cxx
vtkSHFToUnstructuredGridFilter.h
vtkUnstructuredGridToSHFFilter.cxx
vtkUnstructuredGridToSHFFilter.h

Mapeadores VTK:

vtkOpenGLSHFMapper.cxx
vtkOpenGLSHFMapper.h
vtkSHFMapper.cxx
vtkSHFMapper.h

Fontes VTK:

vtkPNMAsciiReader.cxx
vtkPNMAsciiReader.h
vtkSHFSource.cxx
vtkSHFSource.h
vtkSHFReconstruction.cxx
vtkSHFReconstruction.h

Csolid

Classe principal da estrutura de dados SHF, responsável pelo armazenamento de todas as informações referentes ao sólido. Representa o nó Sólido.

```
class Csolid : public vtkDataSet
```

Métodos:

int	GetNumberOfPoints()	Retorna o número de vértices do sólido
int	GetNumberOfCells()	Retorna o número de células do sólido
void	ComputeBounds()	Calcula a caixa de bordo (limites espaciais do domínio de dados)
void	GetScalarRange(float[2])	Pega a taxa dos dados escalares
int	Get_nvertex()	Retorna o número de vértices
void	Set_nvertex(int)	Altera o número de vértices
void	inc_nvertex(int=1)	Incrementa o número de vértices
void	dec_nvertex(int=1)	Decrementa o número de vértices
int	Get_nvertex()	Retorna o número de vértices
int	Get_ncav()	Retorna o número de cavidades
int	Get_ncc()	Retorna o número de componentes
int	Get_ncell()	Retorna o número de células
int	Get_nbcp()	Retorna o número de componentes de bordo
int	Get_nface()	Retorna o número de cavidades
int	Get_ncav()	Retorna o número de cavidades
Ccell	*Make_Cell(int*,int=1)	Cria um tetraedro com os índices dos vértices (int*).
void	Add_vtx(Cvertex*,int=0)	Adiciona um vértice ao sólido
void	Set_StarVertex(Ccell*)	Atualiza a estrela do vértice para uma célula nova
void	Set_Moperator(Ccell*)	Faz a colagem da nova célula utilizando operadores topológicos

Listas:

CLinkedList<Ccell>	Get_lcell()	Retorna a lista de células
CLinkedList<Cvertex>	Get_lvertex()	Retorna a lista de vértices
CLinkedList<Cboundarycp>	Get_lbcp()	Retorna a lista de componentes de bordo
CLinkedList<Csingvertex>	Get_lsingvertex()	Retorna a lista de vértices singulares
CLinkedList<Cedgevertex>	Get_lsingedge()	Retorna a lista de arestas singulares

Iterators:

iter_base<Ccell>	*Begin_cell()	Primeira célula da lista
iter_base<Ccell>	*End_cell()	Última célula da lista
iter_base<Cvertex>	*Begin_vertex()	Primeiro vértice da lista
iter_base<Cvertex>	*End_vertex()	Último vértice da lista
iter_base<Cboundarycp>	*Begin_boundarycp()	Primeiro componente de bordo da lista
iter_base<Cboundarycp>	*End_boundarycp()	Último componente de bordo da lista
iter_base<Csingvertex>	*Begin_singvertex()	Primeiro vértice singular da lista
iter_base<Csingvertex>	*End_singvertex()	Último vértice singular da lista
iter_base<Csingedge>	*Begin_singedge()	Primeira aresta singular da lista
iter_base<Csingedge>	*End_singedge()	Última aresta singular da lista

Cvertex

Representa vértices 3D na estrutura. Nó Vértices.

```
class Cvertex : public CLinkedListNode<Cvertex>
```

Definição:

typedef double real; “real” equivale a “double”

Construtor:

Cvertex(real x=0.0,real y=0.0,real z=0.0, Csolid* pai=NULL, int id=0,void*=NULL)

Entrada: coordenadas X-Y-Z, sólido ao qual pertence, identificador e dados adicionais

Métodos:

int	Get_id()	Retorna o identificador do vértice
void*	Get_data()	Retorna os dados anexado ao vértice
real	Get_x()	Retorna a coordenada X
real	Get_y()	Retorna a coordenada Y
real	Get_z()	Retorna a coordenada Z
Csolid*	Get_fsolid()	Retorna o sólido que contém esse vértice
Cvertex*	FindID(int)	Encontra um vértice pelo identificador
void	Set_id(int)	Altera o identificador do vértice
void	Set_x(real)	Altera a coordenada X
void	Set_y(real)	Altera a coordenada Y
void	Set_z(real)	Altera a coordenada Z
void	Set_data(void*)	Altera dados adicionais do vértice
void	Set_fsolid(Csolid*)	Indica o sólido ao qual pertence
void	Add_starvertex(Chalfedge*)	Adiciona uma semi-aresta na estrela do vértice

Lista:

CLinkedList<Cstarvertex> *Get_lstarvertex() Retorna a lista com a estrela do vértice

Iterators:

iter_base<Cstarvertex>	*Begin_starvertex()	Primeiro elemento de uma lista com a estrela do vértice
iter_base<Cstarvertex>	*End_starvertex()	Último elemento de uma lista com a estrela do vértice

Cstarvertex

Representa um elemento na estrela do vértice. Cada elemento é composto por uma semi-aresta e dados adicionais. Nó Estrelas.

```
class Cstarvertex : public CLinkedListNode<Cstarvertex>
```

Métodos:

void*	Get_data()	Retorna os dados anexado à estrela do vértice
void	Set_data(void*)	Altera dados adicionais da estrela do vértice
Chalfedge	*Get_lhalfedge()	Retorna a semi-aresta relacionada com esse elemento
void	Set_lhalfedge(Chalfedge*)	Especifica a semi-aresta relacionada a esse elemento

Ccell

Representa cada célula do sólido. Nó Células.

```
class Ccell : public CLinkedNode<Ccell>
```

Métodos:

void*	Get_data()	Retorna os dados anexado à célula
void	Set_data(void*)	Altera dados adicionais da célula
int	Get_id()	Retorna o identificador da célula
void	Set_id(int)	Altera o identificador da célula
Csolid*	Get_fsolid()	Retorna o sólido que contém essa célula
void	Set_fsolid(Csolid*)	Especifica o sólido que conterá essa célula
void	AddHalfFace(Chalfface*)	Adiciona uma semi-face na célula
int	Get_nface()	Retorna o número de faces da célula
Ccell*	FindID(int)	Encontra uma célula pelo identificador pedido

Lista:

CLinkedList<Chalfface>	*Get_lhalfface()	Retorna a lista de semi-faces da célula
------------------------	-------------------	---

Iterators:

iter_base<Chalfface>	*Begin_halfface()	Primeira semi-face da lista
iter_base<Chalfface>	*End_halfface()	Última semi-face da lista

Chalfface

Representa a semi-face de uma célula. Nó Semi-Faces.

```
class Chalfface : public CLinkedNode<Chalfface>
```

Métodos:

void*	Get_data()	Retorna os dados anexado à semi-face
void	Set_data(void*)	Altera dados adicionais da semi-face
int	Get_id()	Retorna o identificador da semi-face
void	Set_id(int)	Altera o identificador da semi-face
Ccell*	Get_fcell()	Retorna a célula que contém essa semi-face
void	Set_fcell(Ccell*)	Especifica a célula que conterá essa semi-face
real	Area()	Calcula a área de uma semi-face triangular
void	Add_halfedge(Chalfedge*)	Adiciona uma semi-aresta na semi-face
int	Get_nvertex()	Retorna o número de vértices da semi-face
Chalfface*	FindIdent(int)	Encontra uma semi-face pelo identificador na célula
Chalfface*	Get_mate()	Retorna o mate da semi-face
void	Set_mate(Chalfface*)	Especifica o mate da semi-face

Lista:

CCircularList<Chalfedge>	*Get_lhalfedge()	Retorna a lista de semi-arestas da semi-face
--------------------------	-------------------	--

Iterators:

iter_base<Chalfedge>	*Begin_halfedge()	Primeira semi-aresta adicionada na lista
iter_base<Chalfedge>	*End_halfedge()	Última semi-aresta adicionada na lista

Chalfedge

Representa a semi-aresta de uma semi-face. Nó Semi-Arestas.

```
class Chalfedge : public CCircularNode<Chalfedge>
```

Métodos:

void*	Get_data()	Retorna os dados anexado à semi-aresta
void	Set_data(void*)	Altera dados adicionais da semi-aresta
int	Get_id()	Retorna o identificador da semi-aresta
void	Set_id(int)	Altera o identificador da semi-aresta
Chalface*	Get_fhalfface()	Retorna a semi-face que contém essa semi-aresta
void	Set_fhalfface(Chalface*)	Especifica a semi-face que conterá essa semi-aresta
Cvertex*	Get_vtx()	Retorna o vértice da semi-aresta
void	Set_vtx(Cvertex*)	Especifica o vértice da semi-aresta
Chalfedge*	FindID(int)	Encontra uma semi-aresta pelo identificador na semi-face
Chalfedge*	Get_mateedge()	Retorna a mate-aresta da semi-aresta
Chalfedge*	Get_matecell()	Retorna a mate-célula da semi-aresta
void	Set_mateedge(Chalfedge*)	Especifica a mate-aresta da semi-aresta
void	Set_matecell(Chalfedge*)	Especifica a mate-célula da semi-aresta

Iterators:

iter_base<Chalfedge>	*Begin_staredge()	Primeiro elemento da estrela da aresta
iter_base<Chalfedge>	*End_staredge()	Último elemento da estrela da aresta

Cboundarycp

Representa os bordos da estrutura. Nó Componentes de Bordo.

```
class Cboundarycp : public CLinkedListNode<Cboundarycp>
```

Métodos:

int	Get_id()	Retorna o identificador do componente de bordo
void	Set_id(int)	Altera o identificador do componente de bordo
Csolid*	Get_fsolid()	Retorna o sólido que contém esse componente
void	Set_fsolid(Csolid*)	Especifica o sólido que conterá esse componente
int	Get_nbface()	Retorna o número de faces de bordo do componente
int	Get_nhole()	Retorna o número de buracos no bordo

Lista:

CLinkedList<Cboundaryface>	*Get_lbface()	Retorna a lista de faces de bordo do componente
----------------------------	----------------	---

Iterators:

iter_base<Cboundaryface>	*Begin_boundaryface()	Primeira face de bordo da lista
iter_base<Cboundaryface>	*End_boundaryface()	Última face de bordo da lista

Cboundaryface

Representa uma face de bordo de um componente. Nó Faces de Bordo.

```
class Cboundaryface : public CLinkedNode<Cboundaryface>
```

Métodos:

void*	Get_data()	Retorna os dados anexado à face de bordo
void	Set_data(void*)	Altera dados adicionais da face de bordo
int	Get_id()	Retorna o identificador da face de bordo
void	Set_id(int)	Altera o identificador da face de bordo
Cboundarycp*	Get_fcp()	Retorna o componente que contém essa face de bordo
void	Set_fcp(Cboundarycp*)	Especifica o componente que conterá essa face de bordo
Chalfface*	Get_lhalfface()	Retorna a semi-face considerada de bordo
void	Set_lhalfface(Chalfface*)	Especifica a semi-face de bordo

Csingvertex

Armazena um vértice singular. Nó Vértices-Singulares.

```
class Csingvertex : public CLinkedNode<Csingvertex>
```

Métodos:

void*	Get_data()	Retorna os dados anexado ao vértice singular
void	Set_data(void*)	Altera dados adicionais do vértice singular
int	Get_id()	Retorna o identificador do vértice singular
void	Set_id(int)	Altera o identificador do vértice singular
Csolid*	Get_fsolid()	Retorna o sólido que contém esse vértice singular
void	Set_fsolid(Csolid*)	Especifica o sólido que conterá esse vértice singular
Cvertex*	Get_vtx()	Retorna o vértice considerado singular
void	Set_vtx(Cvertex*)	Especifica o vértice singular

Csingedge

Armazena uma aresta singular. Nó Arestas-Singulares.

```
class Csingedge : public CLinkedNode<Csingedge>
```

Métodos:

void*	Get_data()	Retorna os dados anexado à aresta singular
void	Set_data(void*)	Altera dados adicionais da aresta singular
int	Get_id()	Retorna o identificador da aresta singular
void	Set_id(int)	Altera o identificador da aresta singular
Csolid*	Get_fsolid()	Retorna o sólido que contém essa aresta singular
void	Set_fsolid(Csolid*)	Especifica o sólido que conterá essa aresta singular
Chalfedge	*Get_lhalfedge()	Retorna a semi-aresta considerada singular
void	Set_lhalfedge(Chalfedge*)	Especifica a semi-aresta singular

SHF_reader

Leitor de arquivos utilizado pela estrutura. Conversores para importação.

```
class SHF_reader
```

Definição:

```
enum FILE_TYPE_R { VTK,TETRA,HLF,VTK_OPER,TETRA_OPER }
```

Tipos de arquivos: VTK – UnstructuredGrid, TETRA – 0-1, HLF – HLF,
OPER – Utiliza novos operadores topológicos

Construtor:

SHF_reader(char*,FILE_TYPE_R=VTK) Entrada: nome e tipo de arquivo

Métodos:

void	Set_FileName(char*)	Especifica o nome do arquivo
char*	Get_FileName()	Retorna o nome do arquivo especificado
void	Set_Filetype(FILE_TYPE_R)	Especifica o tipo de arquivo
Csolid*	Execute()	Faz a leitura e retorna o sólido

SHF_writer

Escritor de arquivos utilizado pela estrutura. Conversores para exportação.

```
class SHF_writer
```

Definição:

```
enum FILE_TYPE_W { vtk,tetra,hlf,VTK_B,POLYDATA_B }
```

Tipos de arquivos: vtk - UnstructuredGrid, tetra - 0-1, hlf - HLF, B - bordo, POLYDATA - PolyData

Construtor:

SHF_reader(char*=NULL,FILE_TYPE_W=vtk,Csolid*=NULL)

Entrada: nome e tipo de arquivo e sólido.

Métodos:

void	Set_FileName(char*)	Especifica o nome do arquivo
char*	Get_FileName()	Retorna o nome do arquivo especificado
void	Set_Filetype(FILE_TYPE_R)	Especifica o tipo de arquivo
void	Set_solid(Csolid*)	Indica o sólido que será salvo
void	Execute()	Salva o sólido indicado

vtkSHFMapper

Classe abstrata. Mapeador da estrutura SHF para primitivas gráficas.

```
class vtkSHFMapper : public vtkMapper
```

Métodos:

virtual void	RenderPiece(vtkRenderer*,vtkActor*)	Implementado pelas sub-classes
virtual void	Render(vtkRenderer*,vtkActor*)	Chama o RenderPiece
void	SetInput(Csolid*)	Especifica a entrada do mapeador
Csolid	*GetInput()	Retorna a entrada do mapeador
Void	SetScalarModeToUsePointData();	Exibe cores para representar os escalares

vtkOpenGLSHFMapper

Mapeador da estrutura SHF para OpenGL.

```
class vtkOpenGLSHFMapper : public vtkSHFMapper
```

Métodos:

virtual void	RenderPiece(vtkRenderer*,vtkActor*)	Faz a renderização
virtual void	Draw(vtkRenderer*,vtkActor*)	Método de desenho em OpenGL
void	ReleaseGraphicsResources(vtkWindow*)	Libera qualquer recurso gráfico utilizado

vtkSHFSource

Classe abstrata cujas subclasses geram dados na estrutura SHF.

```
class vtkSHFSource : public vtkSource
```

Métodos:

Csolid	*GetOutput()	Retorna a saída dessa fonte
void	SetOutput(Csolid*)	Retorna a saída dessa fonte

vtkUnstructuredGridToSHFFilter

Filtro que converte uma malha não-estruturada formada por tetraedro para um objeto do tipo *Csolid* (estrutura SHF).

```
class vtkUnstructuredGridToSHFFilter : public vtkSHFSource
```

Métodos:

```
void    SetInput(vtkUnstructuredGrid*)    Especifica a entrada do filtro
vtkUnstructuredGrid *GetInput( )          Retorna a entrada do filtro
```

vtkSHFToUnstructuredGridFilter

Filtro que converte os tetraedros armazenados na estrutura SHF para uma malha não-estruturada.

```
class vtkSHFToUnstructuredGridFilter : public vtkUnstructuredGridSource
```

Métodos:

```
void    SetInput(Csolid*)                Especifica a entrada do filtro
Csolid  *GetInput( )                     Retorna a entrada do filtro
```

vtkSHFBoundaryToPolyDataFilter

Filtro que converte o bordo dos objetos armazenados na estrutura SHF para dados poligonais.

```
class vtkSHFBoundaryToPolyDataFilter : public vtkPolyDataSource
```

Métodos:

```
void    SetInput(Csolid*)                Especifica a entrada do filtro
Csolid  *GetInput( )                     Retorna a entrada do filtro
```

vtkPNMAsciiReader

Leitor de arquivos PNM, em tons de cinza, no formato ASCII. A saída é um objeto de dados *vtkStructuredPoints*. Sua superclasse oferece métodos para controlar a leitura do arquivo de dados.

```
class vtkPNMAsciiReader : public vtkDataReader
```

Métodos:

vtkStructuredPoints	*GetOutput()	Retorna a saída desse leitor
void	SetOutput(vtkStructuredPoints*)	Recebe a saída desse leitor

vtkSHFReconstruction

Módulo que faz uma reconstrução tridimensional de objetos formados por uma sequência de fatias. O objeto gerado será armazenado na estrutura SHF.

```
class vtkSHFReconstruction : public vtkSHFSource
```

Métodos:

void	SetSlices(vtkStructuredPointsCollection*)	Especifica as fatias que serão utilizadas na reconstrução
vtkStructuredPointsCollection*	GetSlices()	Recebe as fatias que serão utilizadas na reconstrução
void	SetInitial(int)	Altera o número da fatia inicial
int	GetInitial()	Recebe o número da fatia inicial
void	SetFinal(int)	Altera o número da fatia final
int	GetFinal()	Recebe o número da fatia final
void	SetRefFactor(int)	Altera o fator de distância entre as fatias
int	GetRefFactor()	Recebe o fator de distância entre as fatias
void	SetScaleZ(double)	Altera a variável de escala
double	GetScaleZ()	Recebe o valor da variável de escala
void	SetValueMin(int)	Altera o valor mínimo do escalar aceito pelo reconstrutor
int	GetValueMin()	Recebe o valor mínimo do escalar aceito pelo reconstrutor
void	SetValueMax(int)	Altera o valor máximo do escalar aceito pelo reconstrutor
int	GetValueMax()	Recebe o valor máximo do escalar aceito pelo reconstrutor
void	SetRemoveCavity(int)	Especifica o número de cavidades que serão removidas
int	GetRemoveCavity()	Recebe o número de cavidades que serão removidas
void	ColorOn()	Habilita a inclusão dos valores escalares aos vértices
void	ColorOff()	Desabilita a inclusão dos valores escalares aos vértices
void	AverageOn()	A média dos escalares dos vértices será considerada na reconstrução
void	AverageOff()	Todos os escalares dos vértices serão considerados na reconstrução
void	SmoothOn()	Habilita o uso do filtro de alisamento
void	SmoothOff()	Desabilita o uso do filtro de alisamento

Referências Bibliográficas

- [Alv99] ALVES, A. D. Visualização na WWW: estudo e implementação para um domínio específico. Monografia de qualificação de mestrado - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 1999.
- [Bau75a] BAUMGART, B. G. Winged-Edge Polyhedron Representation for Computer Vision, *National Computer Conference*, p.589-596, 1975.
- [Bau75b] BAUMGART, B. G. A Polyhedron Representation for Computer Vision. *AFIPS Proceedings*, n.44, p.589-596, 1975.
- [Bis01] BÍSCARO, H. H. Singular Half-Face: Uma Estrutura de Dados Volumétrica. Dissertação de mestrado - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2001.
- [Cas92] CASTELO FILHO, A. Aproximações Adaptativas de Variedades Implícitas e com Aplicações à Modelagem Implícita e Equações Algébrico-Diferenciais. Tese de Doutorado - Departamento de Matemática, Pontifícia Universidade Católica do Rio de Janeiro, 1992.
- [Cma02] CMAKE, Cross-plataform Make. (World Wide Web, 12/02/03, <http://www.cmake.org>)
- [Ear97] EARNSHAW, R.A., VINCE, J., JONES, H., *Visualization and Modeling*, Academic Press, 1997.
- [Elv92] ELVINS, T. T. A Survey of Algorithms for Volume Visualization. *Computer Graphics*, v.26, n.3, p.194-201, 1992.
- [Gom98] GOMES, J., VELHO, L. *Computação Gráfica, Volume 1*. Série de Computação e Matemática, IMPA, 1998.
- [Hof89] HOFFMAN, C. M. *Geometric and Solid Modeling: An Introduction*. Morgan Kaufmann, 1989.
- [Hof91] HOFFMAN, C. M., VENECEK JR., G. Fundamental Techniques for Geometric and Solid Modeling, *Control and Dynamic Systems*, n.48, p.101-165, 1991.

- [Kit02] KITWARE, INC. *The Visualization Toolkit*. (World Wide Web, 12/02/03, <http://public.kitware.com/vtk>)
- [Las96] LASZLO, M. J. *Computational Geometric and Computer Graphics in C++*. Prentice Hall, 1996.
- [Lop96] LOPES, H. Algoritmos para Construção e Desconstrução de Variedades de Dimensões 2 e 3. Tese de doutorado - Departamento de Matemática, Pontifícia Universidade Católica do Rio de Janeiro, 1996.
- [Lop97] LOPES, H., TAVARES, G. Structural Operators for Modeling 3-Manifolds. *Symposium on Solid Modeling and Applications*, Atlanta, p.10-18, 1997.
- [Män88] MÄNTYLÄ, M. *An Introduction to Solid Modeling*. Computer Science Press, 1988.
- [Min97] MINGHIM, R., OLIVEIRA, M. C. F. Uma Introdução à Visualização Computacional. *XVI JAI – Jornadas de Atualização em Informática, XVII Congresso da Sociedade Brasileira de Computação*, p.85-131, 1997.
- [Mor85] MORTENSON, M. E. *Geometric Modeling*. John Wiley, 1985
- [Non98] NONATO, L. G. Reconstrução Volumétrica de Objetos definidos por seções Bidimensionais. Tese de doutorado - Departamento de Matemática, Pontifícia Universidade Católica do Rio de Janeiro, 1998.
- [Pes97] PESCO, S. Modelagem Geométrica com Espaços Poliedrais Singulares. Tese de doutorado - Departamento de Matemática, Pontifícia Universidade Católica do Rio de Janeiro, 1997.
- [Req83] REQUICHA, A. G., VOELCKER, H. B. Solid Modeling: current status and research directions. *IEEE computer graphics and application*, v.3, n.7, p.25-37, 1983.
- [Roc02] ROCHA, C. F. P. Visualização Volumétrica com Técnicas Distribuídas. Relatório de iniciação científica - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2002.
- [Rub00] RUBERT, C. Tratamento de Imagens Binárias por Morfologia Matemática. Trabalho de graduação - Departamento de Engenharias e Ciência da

Computação, Universidade Regional Integrada do Alto Uruguai e da Missões, 2000.

- [Rum91] RUMBAUGHT, J., BLAHA, M., PREMERLANI, W., EDDY, F., LORENSEN, W. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.
- [Sch00] SCHROEDER W. J., AVILA, L. S., MARTIM, K. M., HOFFMAN, W., CHARLES LAW, C. *The Visualization Toolkit - User's Guide*. Kitware Inc., 2000.
- [Sch98] SCHROEDER, W. J., MARTIM, K., LORENSEN, B. *The Visualization Toolkit - An Object-Oriented Approach to 3D Graphics*. 2.ed., Prentice-Hall, 1998.
- [Sei96] SEIXAS, R. B., GATTASS, M. Visualização Volumétrica em um Ambiente de Computação Distribuída. *Anais do IX SIBGRAPI*, p.15-22, 1996.
- [Soa99] SOARES, I. P. Monografia apresentada no curso de Seminários Avançados de Computação Gráfica - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 1999.
- [Soa00] SOARES, I. P. Visualização Volumétrica em Odontologia, Dissertação - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2000.