Visualização de Modelos de Sólidos Analisados pelo Método dos Elementos de Contorno

Raquel Marcia Müller

Dissertação apresentada ao Departamento de Computação e Estatística da Universidade Federal de Mato Grosso do Sul como parte dos requisitos para obtenção do título de **Mestre em Ciência da Computação.**

ORIENTADOR: Prof. Dr. Paulo Aristarco Pagliosa

Campo Grande - MS
2002

A meus pais.

Agradecimentos

Ao meu orientador Paulo Aristarco Pagliosa, pela confiança em mim depositada. Pelo tempo e dedicação destinados à minha orientação e pela paciência em trabalhar com uma orientanda que não morava na mesma cidade que ele e que não podia dedicar tempo integral aos trabalhos desenvolvidos durante o mestrado.

Ao amigo Roberto Murillo, que por muitas vezes compartilhou comigo os momentos de angústia e de alegria pelos quais passamos.

À minha família, especialmente à minha irmã Luciane e meu cunhado Flávio, que sempre me receberam de braços abertos em sua casa, me apoiando em tudo que precisei. Obrigada é pouco para dizer.

Aos meus alunos da UEMS e da UNIGRAN, que por algumas vezes tiveram que tolerar a minha ausência às aulas e ainda assim, sempre me trouxeram palavras de apoio.

Aos amigos do Departamento de Computação e Estatística da Universidade Federal de Mato Grosso do Sul, pela acolhida sempre calorosa com que fui recebida em todas as ocasiões em que lá estive.

Aos colegas professores da UEMS e da UNIGRAN, que não mediram esforços no sentido de me auxiliar nos momentos em que precisei.

Resumo

Müller, R.M. *Visualização de Modelos de Sólidos Analisados pelo Método dos Elementos de Contorno*. Campo Grande, 2002. Dissertação (Mestrado) - Universidade Federal de Mato Grosso do Sul.

O propósito deste trabalho é o desenvolvimento de técnicas de visualização de dados sobre o contorno e no domínio de modelos de sólidos elásticos analisados pelo Método dos Elementos de Contorno (MEC). O MEC tem-se constituído como importante método alternativo de análise numérica em Ciências e Engenharia, mas as resposta obtidas com o MEC nem sempre podem ser diretamente utilizadas, dependendo da implementação, para a visualização de campos escalares, vetoriais e tensoriais do domínio do corpo sendo analisado. O procedimento proposto para visualização de dados sobre o contorno e no domínio de um modelo de sólido é basedo na triangulação da superfície e do volume do sólido. O contorno é discretizado por elementos triangulares planos; o domínio por células internas tetraédricas. As funções de forma dos elementos de contorno e células internas são utilizadas para interpolação dos dados discretos resultantes da análise. Os principais objetivos desta pesquisa são a investigação de algoritmos de visualização escalar e vetorial e a aplicação destes algoritmos no desenvolvimento de uma biblioteca de classes denominada *Boundary Element Visualization Library* (BEVL). Tal biblioteca de classes poderá ser empregada na implementação de aplicações orientadas a objetos de visualização de dados escalares e vetoriais resultantes da análise elastostática de sólidos pelo MEC.

Palavras-chave: *visualização*, *método dos elementos de contorno*.

Abstract

Müller, R.M. *Visualização de Modelos de Sólidos Analisados pelo Método dos Elementos de Contorno*. Campo Grande, 2002. Dissertação (Mestrado) - Universidade Federal de Mato Grosso do Sul.

The purpose of this work is the development of techniques for visualization of both boundary and domain data resulting from the analysis of elastic solid models by using the Boundary Element Method (BEM). Although this one has become an important alternative numerical method in Science and Engineering, the analysis results given by the method do not directly allow, depending on its implementation, the visualization of scalar, vectorial, and tensorial fields in the domain of the body under consideration. The proposed approach to visualizing boundary and domain data of a solid model is based on triangulations of the surface and volume of the solid. The boundary is discretized by flat triangular elements; the domain by tetrahedral internal cells. The shape functions of the boundary elements and internal cells are employed to interpolate discrete data resulting from the analysis. The main objectives of this research are the study of scalar and vectorial visualization algorithms, and the application of them to the development of a class library called Boundary Element Visualization Library (BEVL). This one can be used for the implementation of object oriented applications for visualization of scalar and vectorial data resulting from BEM elastostatic analysis of solid models.

Keywords: *visualization*, *boundary element method*.

Conteúdo

Lista de Figuras

Lista de Programas

CAPÍTULO 1

Introdução

1.1 Modelagem, Análise Mecânica e Visualização de Sólidos

Seja um sólido qualquer, existente na natureza ou que deseja-se construir, submetido a um conjunto qualquer de ações tais como forças gravitacionais, variações de temperatura, ou impactos de outros corpos. *Qual é o comportamento mecânico do sólido? Como podemos prever este comportamento, e com que precisão e rapidez?* Respostas a estas perguntas são, do ponto de vista prático, fundamentais em muitos campos da Ciência e Engenharia, e só podem ser obtidas com o auxílio do computador.

A solução *exata* para a questão da predição do comportamento mecânico de um sólido, entretanto, não pode ser prontamente estabelecida, porque a natureza é complexa demais e seus fenômenos não podem ser totalmente compreendidos pela mente humana. O homem só pode lidar com a complexidade isolando mentalmente entidades concretas ou abstratas que, na realidade, nunca se encontram isoladas, dirigindo atenção apenas às propriedades mais importantes do problema de interesse e criando, em função destas propriedades, *representações* que definem idealmente a estrutura e o comportamento das entidades. Estas representações são chamadas de *modelos*.

MODELOS Modelos são representações das características principais de uma entidade concreta ou abstrata, construídas com o propósito de permitir a visualização e a compreensão da estrutura e do comportamento da entidade.

Na tentativa de compreender o comportamento mecânico de um sólido, PAGLIOSA [22] considera os tipos de modelos a seguir.

- *Modelos matemáticos.* O modelo matemático de um sólido é usualmente definido por um conjunto de equações diferenciais elaboradas a partir de hipóteses simplificadoras a respeito da natureza mecânica dos materiais que constituem o sólido. Tal conjunto de equações diferenciais, juntamente com as *condições de contorno* que garantem a unicidade de sua solução, representam o comportamento mecânico do sólido em relação à ações tais como forças gravitacionais, variações de temperatura, etc. As ações aplicadas a um sólido podem ser genericamente chamadas de *carregamentos*.

- *Modelos geométricos.* A geometria de um sólido pode ser exata ou aproximadamente definida por equações matemáticas ou hierarquias de elementos de curvas, superfícies ou volumes que, conectados entre si, descrevem a forma e as dimensões do sólido. Um modelo geométrico pode conter, também, a descrição das propriedades dos materiais que constituem o sólido e, eventualmente, as especificações das condições de contorno e das ações aplicadas ao sólido. Tanto as informações geométricas quanto as informações a respeito das propriedades materiais, condições de contorno e carregamentos são importantes no processo de resolução do modelo matemático do sólido.

- *Modelos de análise.* Para os casos mais gerais, soluções das equações diferenciais do modelo matemático de um sólido só podem ser obtidas por métodos computacionais numéricos tais como *Método dos Elementos Finitos* (MEF) e/ou *Método dos Elementos de Contorno* (MEC). Um modelo de análise de um sólido é definido por uma *malha* de elementos (finitos e/ou de contorno) resultante de uma *discretização* do volume e/ou da superfície do sólido, juntamente com a especificação das condições de contorno e das ações, ou carregamentos, aplicados ao sólido.

Segundo SCHROEDER e SHEPARD [27], análise é definida como a seguir.

ANÁLISE Processo que, a partir de um conjunto apropriado de manipulações, transforma informações de entrada de um determinado domínio físico em informações de saída que oferecem respostas a algumas questões de interesse no domínio considerado.

As informações de entrada são dadas no modelo geométrico construído para representar o sólido. As informações de saída são as respostas ao problema de predição do comportamento mecânico: valores dos deslocamentos e forças de superfície sobre o contorno do sólido e deslocamentos e tensões no domínio do sólido. O conjunto apropriado de transformações é definido pela metodologia numérica empregada na análise, por exemplo, MEC, MEF, ou acoplamento MEC/MEF.

O método computacional numérico mais amplamente utilizado em Ciências e Engenharia é o MEF [2, 25, 33, 34]. A técnica, matematicamente fundamentada em princípios variacionais, ou mais genericamente, em expressões de resíduos ponderados, consiste na divisão do volume, ou domínio, do sólido em um número finito de regiões ou *células* chamadas elementos finitos, as quais são interconectadas através de um número finito de pontos nodais chamados *nós*. A Figura 1.1 ilustra a divisão de um domínio bidimensional Ω (neste caso uma superfície, ao invés de um volume) em elementos finitos. O comportamento isolado de um elemento finito é especificado por um conjunto discreto de parâmetros normalmente associados às variáveis do problema físico em questão. O comportamento global do domínio, igualmente especificado em função de tais parâmetros, pode ser determinado pela solução de sistemas de equações algébricas definidas a partir das contribuições individuais de todos os elementos finitos.

A solução do sistema linear obtido a partir da formulação do MEF fornece respostas, para as variáveis que governam o problema em questão, nos pontos nodais do domínio discretizado. A partir dos valores nodais, podemos estabelecer o valor em *qualquer* ponto do domínio por interpolação dos valores nodais. Para tal, todo elemento finito é dotado de funções de interpolação que possibilitam a determinação do valor de uma grandeza, no interior ou contorno do elemento, a partir do valor (conhecido) dessa grandeza nos pontos nodais nos quais o elemento incide. Dado um ponto P qualquer do domínio Ω, no qual desejamos conhecer o valor de alguma grandeza resultante do processo de análise numérica, primeiramente determinamos em qual elemento finito o ponto está e, então, usamos as funções de interpolação do elemento para obter o valor desejado.

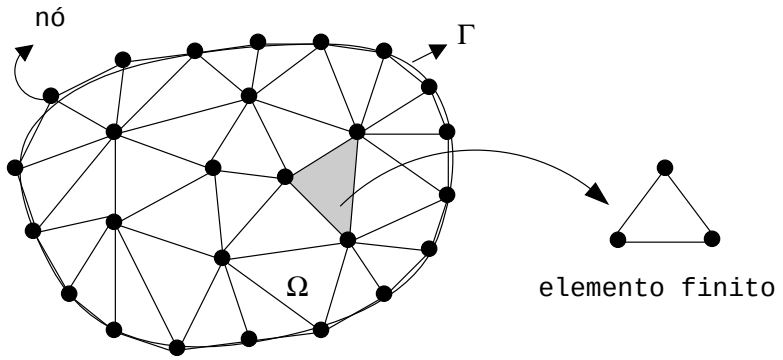

Figura 1.1: Domínio bidimensional discretizado em elementos finitos.

Uma técnica de análise numérica alternativa bastante interessante em Ciências e Engenharia é o MEC [2, 4, 25, 31]. O método baseia-se na transformação das equações diferenciais parciais que descrevem o comportamento mecânico no interior e na superfície, ou contorno, de um sólido em equações integrais definidas apenas em termos de valores no contorno. Em seguida, as equações integrais são resolvidas numericamente, a partir da discretização do contorno do sólido em células chamadas elementos de contorno, conectadas (como no caso dos elementos finitos) por pontos discretos denominados nós. Um elemento de contorno é um trecho de superfície do contorno definido geometricamente por uma seqüência ordenada dos nós nos quais o elemento incide. A Figura 1.2 ilustra a divisão do contorno Γ de um domínio bidimensional Ω em elementos de contorno (neste caso um trecho de curva, ao invés de um trecho de superfície). A figura mostra, também, um conjunto discreto de *pontos internos*, cuja funcionalidade será explicada posteriormente.

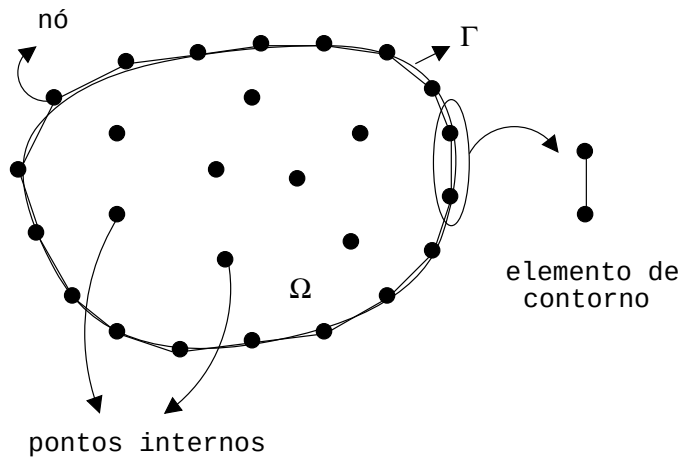

Figura 1.2: Domínio bidimensional discretizado em elementos de contorno.

O MEC, dependendo do problema físico, oferece algumas vantagens importantes em relação ao MEF. Umas das principais vantagens do MEC é a redução considerável do sistema de equações envolvidas (o método reduz a dimensionalidade do problema em um, pois as aproximações numéricas são realizadas apenas sobre o contorno do sólido em questão) e no volume de dados de entrada necessários para executar a análise numérica. Além disso, existem determinados tipos de problemas em mecânica computacional nos quais o MEC é mais convenientemente aplicado que o MEF, como por exemplo, em problemas de simulação em

geomecânica [31].[1] Os dados resultantes da análise numérica de um problema pelo MEC, diferentemente do MEF, fornecem respostas somente nos pontos nodais do contorno do sólido em questão. A partir dos valores dos pontos nodais, podemos obter o valor de uma grandeza em *qualquer* ponto do contorno do sólido, por interpolação. Para tal, da mesma forma que um elemento finito, um elemento de contorno é dotado de funções de interpolação. Para obtermos respostas em um ponto P do contorno do sólido, então, procedemos analogamente ao MEF. Primeiro determinamos sobre qual elemento de contorno P se encontra e, então, utilizamos os valores nodais e as funções de interpolação do elemento para calcular o valor desejado. Respostas no domínio do sólido podem ser obtidas, com o MEC, em pontos discretos internos (veja a Figura 1.2) através de fórmulas de integração numérica envolvendo os resultados obtidos no contorno [4].

Os processos de análise computacional numérica do comportamento mecânico de um sólido produzem, geralmente, uma grande quantidade de dados que impossibilita a interpretação imediata dos resultados da análise. Para oferecer suporte ao processo de interpretação dos resultados obtidos, podemos utilizar técnicas de *visualização*. O objetivo do processo de visualização é, efetivamente, possibilitar a interpretação da grande quantidade de dados resultante do processo de análise produzidos pelo MEF e/ou MEC. Existem várias definições para o termo *visualização* [5, 9, 12, 28]. Para nossos propósitos, assumiremos a definição dada por SCHROEDER [28].

VISUALIZAÇÃO Visualizar significa transformar dados extraídos de um modelo de objeto em imagens que eficientemente representam informações sobre a estrutura e o comportamento do objeto.

A visualização envolve, portanto, a *transformação* de dados e a *representação* de informações. Transformação é o processo de conversão de dados de sua forma original para primitivos gráficos [7] (pontos, linhas, textos, arcos de circunferência, malhas de polígonos) e, eventualmente, em imagens de computador. Representação inclui ambos, as estruturas de dados internas usadas para representar o dado e os primitivos gráficos usados para mostrar o dado.

Embora os métodos de visualização não discriminem a origem de dados, pudemos encontrar na literatura aplicações específicas de técnicas de visualização de dados resultantes da análise numérica de sólidos pelo MEF, por exemplo [8]. De fato, algumas técnicas de visualização de dados do contorno e/ou domínio de um sólido podem ser diretamente implementadas em sistemas computacionais de análise numérica pelo MEF porque tais dados podem ser diretamente determinados, em qualquer ponto do sólido, por interpolação dos dados nos pontos nodais.

Os dados resultantes da análise de um sólido pelo MEC, contudo, nos fornecem informações apenas sobre o contorno do sólido. Como dissemos anteriormente, informações em pontos internos discretos podem ser obtidos por integração. Assim como no MEF, gostaríamos de obter informações em qualquer região do domínio — porém sem ter de aplicar integração numérica, computacionalmente intensiva, a cada nova região — e, transformar tais dados em imagens que nos permitam uma compreensão mais imediata da distribuição dos resultados de análise no domínio do sólido. Além disso, há particularidades em elementos de contorno que, dependendo do modelo utilizado para representação dos dados de análise, impossibilitam a utilização direta dos dados (inclusive dados de contorno) resultantes da análise em algoritmos de visualização. No MEC, por exemplo, podemos representar descontinuidades

[1] Uma outra alternativa de análise numérica que vem sendo bastante pesquisada é o acoplamento do MEF e do MEC, como pode ser visto em [2, 25].

de forças de superfície em um ponto P do contorno no qual incidem elementos de contorno que definem, em P, condições de contorno naturais distintas. Isto significa que, no ponto de contorno P, poderemos ter forças de superfície distintas para cada elemento de contorno incidente em P. Tal situação não ocorre no MEF, onde todos os pontos nodais possuem valor único para um dado, independentemente dos elementos finitos que incidem no ponto.

Nas pesquisas realizadas para a elaboração desse trabalho, não encontramos na literatura qualquer menção à utilização do MEC da forma que estamos propondo. Esse fato nos motivou ainda mais na elaboração e apresentação dessa dissertação.

1.2 Objetivos

O objetivo geral desse trabalho é a investigação de algoritmos de visualização escalar e vetorial e a aplicação destes algoritmos no desenvolvimento de uma biblioteca de classes chamada *Boundary Element Visualization Library*, designada pelo acrônimo BEVL.[2] Tal biblioteca de classes será destinada ao desenvolvimento de aplicações orientadas a objetos de visualização de dados de contorno e de domínio, resultantes da análise numérica pelo MEC, em problemas estáticos de modelagem de sólidos.

Os objetivos específicos do trabalho são:

- Estudar e apresentar os fundamentos computacionais e matemáticos utilizados para o desenvolvimento das classes de objetos da biblioteca de classes;
- Descrever as técnicas desenvolvidas, bem como os atributos e métodos das classes de objetos da biblioteca de classes; e
- Exemplificar a utilização da biblioteca de classes.

Sintetizando, o que esperamos ao término desse trabalho é: dado o modelo de um sólido, e os resultados de análise do modelo pelo MEC, termos uma biblioteca de classes que nos permita construir aplicações de visualização de dados do contorno e domínio do modelo do sólido. A implementação em computador dessa biblioteca de classes será fundamentada na biblioteca de classes de OSW, desenvolvido por PAGLIOSA [22].

As contribuições esperadas com o trabalho são:

- Desenvolvimento de técnicas básicas para visualização de dados de domínio resultantes da análise numérica de modelos de sólidos pelo Método dos Elementos de Contorno;
- Revisão da biblioteca de classes e extensão dos recursos de análise e visualização de OSW.

1.3 OSW

OSW — *Object Structural Workbench* — é um *toolkit* destinado ao desenvolvimento de *aplicações de modelagem* em Ciências e Engenharia. Uma aplicação de modelagem, no contexto de OSW, é um programa orientado a objetos de análise e visualização de modelos estruturais. Em sistemas orientados a objetos, um *objeto* é definido como um conjunto de dados que representam a estrutura de uma entidade concreta ou abstrata e um conjunto de procedimentos que acessam esses dados e respondem sobre o comportamento da entidade em relação a eventos externos. O emprego da Programação Orientada a Objetos (POO) em OSW é adequada

[2] *Biblioteca*, em português, é o coletivo de livros e documentos. Em computação, é comum a utilização do termo para denotarmos uma coleção de componentes de software. É nesse contexto que o estamos empregando.

não somente porque a POO permite modelar problemas do mundo real tão próximo quanto possível da visão que temos desse mundo, ou porque pode-se escrever programas que são mais facilmente compreendidos e estendidos, mas também porque observa-se uma identidade dos conceitos de modelos e objetos, como dados anteriormente. Em OSW, as propriedadde da POO — encapsulamento, herança, polimorfismo e identidade — são diretamente empregadas na especificação de modelos estruturais.

Basicamente, OSW é constituído de:

- **Biblioteca de classes**. Um programa orientado a objetos é baseado em um modelo de computação definido em termos de objetos que se comunicam através do mecanismo de troca de mensagens. O tipo de um objeto, ou seja, sua estrutura e comportamento, é encapsulado em uma descrição de *classe* de objetos. Dizemos que objetos com estrutura e comportamento comuns pertencem à mesma classe de objetos. Os recursos de modelagem, análise e visualização de OSW são implementados em cerca de duas centenas de classes C++ que constituem a *biblioteca de classes* do *toolkit*, ou OCL (*OSW Class Library*).

- **Ambiente de desenvolvimento**. A utilização imediata da biblioteca de classes de OSW no desenvolvimento de programas de modelagem pode trazer algumas dificuldades, de início, porque, afinal de contas, há um número razoável de novos componentes que precisam ser compreendidos. O *ambiente de desenvolvimento* é um programa orientado a objetos que fornece uma interface de *programação visual* que, embora simples, pode auxiliar o programador na utilização das classes de OSW e na geração de código de uma aplicação de modelagem.

A versão atual da biblioteca de classes de OSW pode ser somente aplicada ao desenvolvimento de aplicações de modelagem simples, tais como programas para análise elastostática e visualização de modelos de cascas pelo MEF e sólidos (mas muito limitadamente) pelo MEC.

1.4 Da Modelagem à Visualização

Conforme discutido anteriormente, visualização é uma ferramenta necessária para auxiliar na compreensão da grande quantidade de informação que envolve o mundo dos computadores. De acordo com DEFANTI [5], o enorme volume de dados gerados por supercomputadores e até mesmo por microcomputadores, atualmente, torna impossível para usuários examinar quantitativamente mais que uma pequena parte de uma dada solução. No escopo deste trabalho, em particular, é praticamente impossível investigar a natureza qualitativa global dos dados resultantes da solução numérica.

Segundo SCHROEDER [28], existem dois pontos importantes envolvendo visualização. Primeiro, a visualização tira vantagem da habilidade natural do sistema de visão humano. Segundo, visualização oferece significantes vantagens financeiras. Nos mercados competitivos de hoje, o trabalho em conjunto de simulação e visualização pode reduzir o custo de um produto e seu tempo de lançamento no mercado [11, 21]. Técnicas de análise como MEF e MEC são usadas para simular a performance de um produto e visualização é usada para verificar os resultados da simulação.

O processo de visualização envolve várias etapas [9, 12, 13, 28]. Basicamente, pode-se identificar três fases principais [12]:

- *Fase de modelagem.* A modelagem envolve a transformação de um problema abstrato inicial em um problema matemático bem definido. O primeiro passo na fase de modelagem é desenvolver o modelo físico. Isso inclui selecionar o domínio da análise no

espaço, o tempo e as regras físicas relevantes a serem incluídas no modelo. O próximo passo é gerar uma idealização matemática do modelo físico. Isso envolve a formulação de equações diferenciais e as condições iniciais e de contorno do modelo matemático, bem como os objetivos da análise.

- *Fase de solução.* Algumas vezes pode-se calcular apenas soluções aproximadas para a idealização matemática. O objetivo da fase de solução é gerar uma solução aproximada confiável para o problema matemático responsável pela declaração dos objetivos da análise. O primeiro passo é formar um modelo matemático aproximado baseado no modelo matemático "exato". O próximo passo envolve modelagem geométrica e discretização geométrica do domínio do problema (por exemplo, geração de malhas no MEF) [27]. A fase de solução consiste, portanto, da transformação do problema matemático "exato" em um problema matemático aproximado, que pode ser resolvido com a utilização de um método numérico.

- *Fase de interpretação e avaliação.* Nesta fase, a solução deve ser interpretada, isto é, deve-se estabelecer uma explicação baseada em causalidades para os resultados da solução e produzir conclusões úteis. Visualização representa um importante papel na interpretação e avaliação.

Para melhor entendimento desse processo, utilizamos um modelo de fluxo de dados ou *diagrama de fluxo de dados* (DFD), também referenciado como *rede de visualização* ou *pipeline de visualização*, que descreve os passos de criação da visualização.

A Figura 1.3 mostra um exemplo de visualização da estrutura de uma esfera oca. O modelo geométrico foi gerado pelo processo de varredura rotacional de dois semicírculos concêntricos de raios iguais a 60cm e 80 cm.

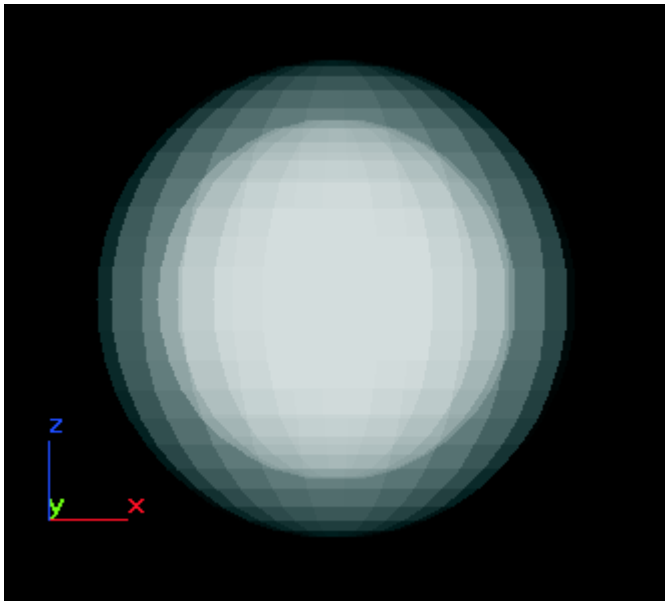

Figura 1.3: Visualização do modelo geométrico de uma esfera oca.

Os passos para a criação da visualização mostrada na Figura 1.3 podem ser vistos no DFD da Figura 1.4. Os blocos desenhados com duas linhas paralelas denotam *repositórios de dados*. Um repositório de dados é responsável pelo armazenamento das informações de um objeto que serão transformadas em imagens da estrutura e do comportamento do objeto.

Blocos retangulares representam *processos fontes* ou *processos sumidouros*. Um processo fonte, ou simplesmente *fonte*, é um processo produtor de um ou mais repositórios de dados. Um processo sumidouro é um processo consumidor de um ou mais repositórios de dados. Blocos retangulares com cantos ovalados representam *processos filtros*. Um processo filtro é um processo que transforma repositórios de dados em outros repositórios de dados. As setas indicam a direção do fluxo de dados.

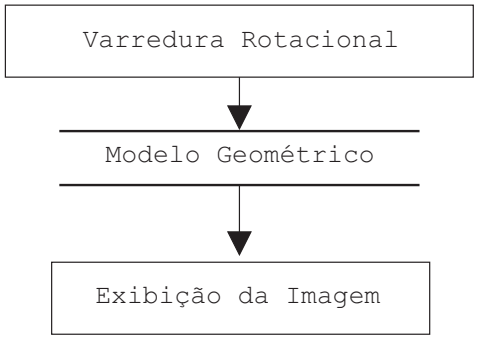

Figura 1.4: Diagrama de fluxo de dados para visualização da Figura 1.3.

No DFD mostrado na Figura 1.4, `Varredura Rotacional` é um processo fonte que toma como parâmetros de entrada uma ou mais curvas abertas ou fechadas, chamadas geratrizes, um eixo de rotação, e um ângulo de rotação. Este processo gera como saída o modelo geométrico resultante da rotação das geratrizes em torno do eixo de rotação, representado pelo repositório de dados `Modelo Geométrico`. O processo sumidouro `Exibição da Imagem` toma como entrada o modelo geométrico e produz como saída uma imagem do modelo, tal como ilustrado na Figura 1.3.

Vamos supor, agora, que a esfera seja constituída de um material elástico cujo módulo de elasticidade é igual a 1.000 kgf/cm^2 e cujo coeficiente de Poisson é igual a 0.3. Vamos supor, ainda, que a esfera sofra uma pressão unitária em seu interior e que seja engastada nas suas seções polares, como pode ser visto na Figura 1.5. Em vermelho vemos as seções engastadas e em azul a pressão interna. Para determinarmos numericamente o comportamento mecânico da esfera, precisamos, inicialmente, de uma malha de elementos discretos. A Figura 1.5 mostra uma malha de elementos de contorno triangulares para o modelo da Figura 1.3.

Os passos para a criação da visualização mostrada na Figura 1.5 podem ser vistos no DFD da Figura 1.6. O processo de transformação `Geração de Malhas` toma como entrada o modelo geométrico da esfera e parâmetros que controlam a aparência da malha e produz como saída a malha de elementos de contorno da esfera mostrada na Figura 1.5. A malha, denotada pelo repositório de dados `Malha`, possui 902 nós e 1.796 elementos de contorno. Tal como no DFD da Figura 1.4, o sumidouro `Exibição da Imagem` toma como entrada o modelo de análise da esfera, geometricamente definido pela malha de elementos, e produz como saída a imagem do modelo.

A Figura 1.7 mostra o mapa de cores do campo escalar correspondente ao deslocamento na direção y, para o modelo da Figura 1.5. Os passos para a criação da visualização mostrada na Figura 1.7 podem ser vistos no DFD da Figura 1.8. O processo rotulado com o nome `Análise` é um processo filtro que recebe como entrada a `Malha` e produz como saída o repositório de dados `Resultados da Análise`. Este repositório representa a mesma malha de elementos de contorno do repositório `Malha`, acrescida de valores para cada um dos 902 nós, resultantes do processo de `Análise`. `Geração do Mapa de Cores` é outro processo filtro que recebe como entrada a malha de elementos de contorno `Campo Escalar` (é a própria `Malha`, após

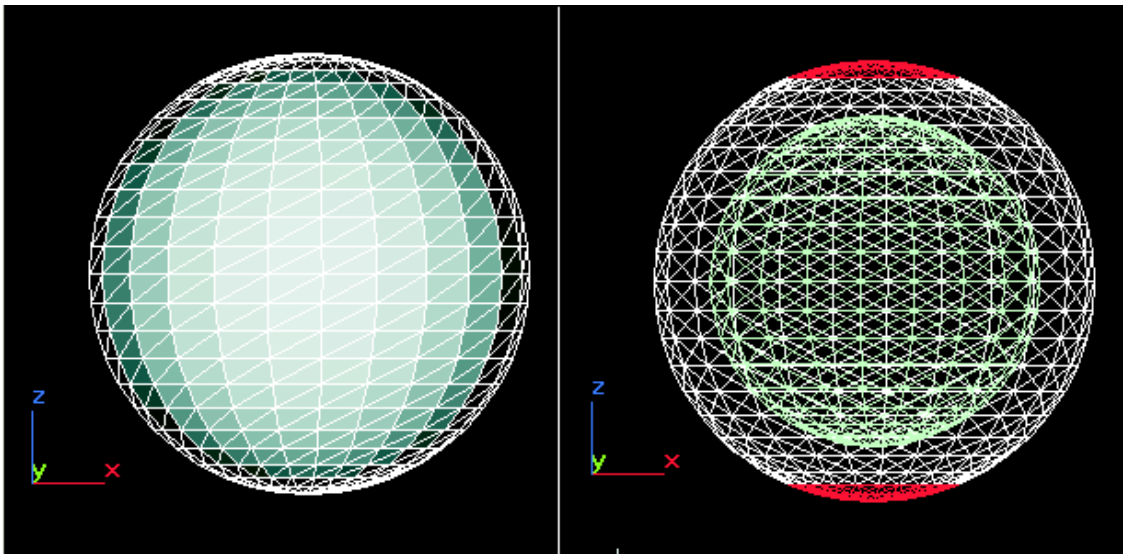

Figura 1.5: Malha de elementos de contorno da esfera.

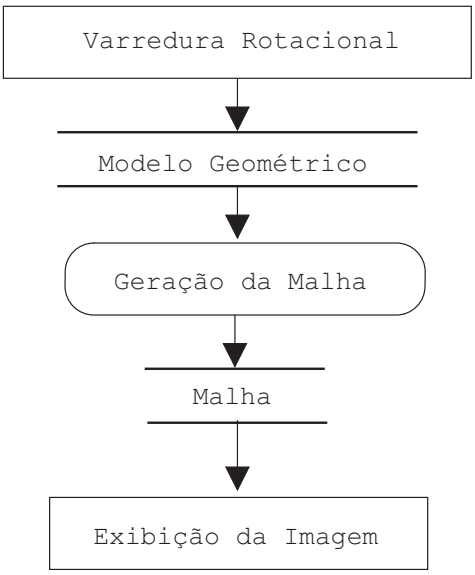

Figura 1.6: Diagrama de fluxo de dados para visualização da Figura 1.5.

passar pelo processo de `Extração do Campo Escalar`) e os parâmetros que definem os valores escalares das cores. A saída é um modelo gráfico que representa o mapa de cores, denotado, no DFD, pelo repositório de dados `Mapa de Cores`. Tal como visto anteriormente, o sumidouro `Exibição da Imagem` toma como entrada o modelo do mapa de cores, e produz como saída a imagem do modelo.

1.5 Passos para a Solução do Problema

A solução do problema proposto neste trabalho, envolve as seguintes etapas, descritas aqui de forma resumida. Estaremos considerando um sólido perfeitamente elástico definido por um domínio Ω de contorno Γ.

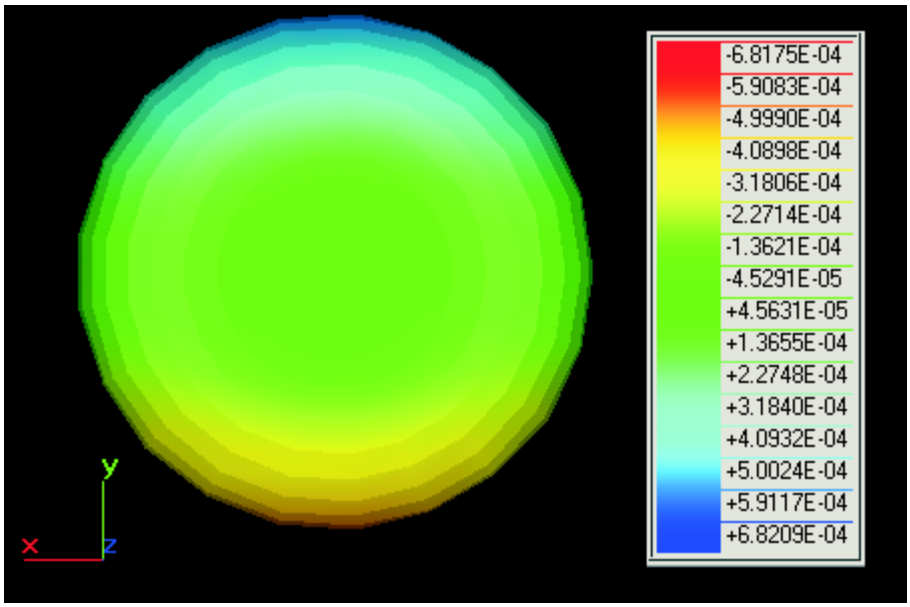

Figura 1.7: Exemplo de um mapa de cores para o modelo da Figura 1.3.

1. Discretizar o domínio Ω em células de domínio do tipo tetraedro. Os nós da malha de elementos de volume decorrentes desta discretização serão nós sobre o contorno Γ e, também, nós internos no domínio Ω.

2. Desenvolver um filtro de geração de malhas de elementos de contorno triangulares. Tal filtro tomará como entrada a malha de tetraedros gerada no passo anterior e produzirá, como saída, a malha de elementos de contorno triangulares. Após a aplicação do filtro, teremos um conjunto de dados constituído por duas malhas de elementos — uma malha de elementos de volume tetraédricos e uma malha de elementos de contorno triangulares — que usam a mesma coleção de nós. Os nós dos elementos de contorno serão rotulados como *nós de contorno*, e os demais nós serão rotulados como *nós internos*.

3. Resolver o modelo de análise pelo MEC. Utilizaremos, neste passo, o analisador desenvolvido em [22] e, atualmente, reescrito com base no analisador desenvolvido por Francisco Patrick Araújo Almeida, sob orientação do Dr. Humberto Breves Coda, do Grupo de Mecânica Computacional da SET-EESC-USP. Os resultados serão valores de deslocamentos e forças de superfície em todos os nós rotulados como nós de contorno no passo anterior. A partir dos valores de contorno, podemos computar, por integração, os valores de deslocamento e tensões internas em todos os nós rotulados como nós internos no passo 2. Com isso, serão conhecidos os valores em todos os nós do conjunto de dados resultante do passo 1. Conhecidos os valores nodais dos elementos de volume, podemos determinar, por interpolação, os valores em quaisquer pontos do domínio Ω do sólido, usando as funções de interpolação do tetraedro.

4. Utilizar os objetos da BEVL para visualização dos dados escalares e vetoriais de contorno e de domínio, obtidos no passo anterior.

1.6 Notação

Nesta seção será introduzida a notação matemática utilizada no decorrer do texto, de acordo com [22].

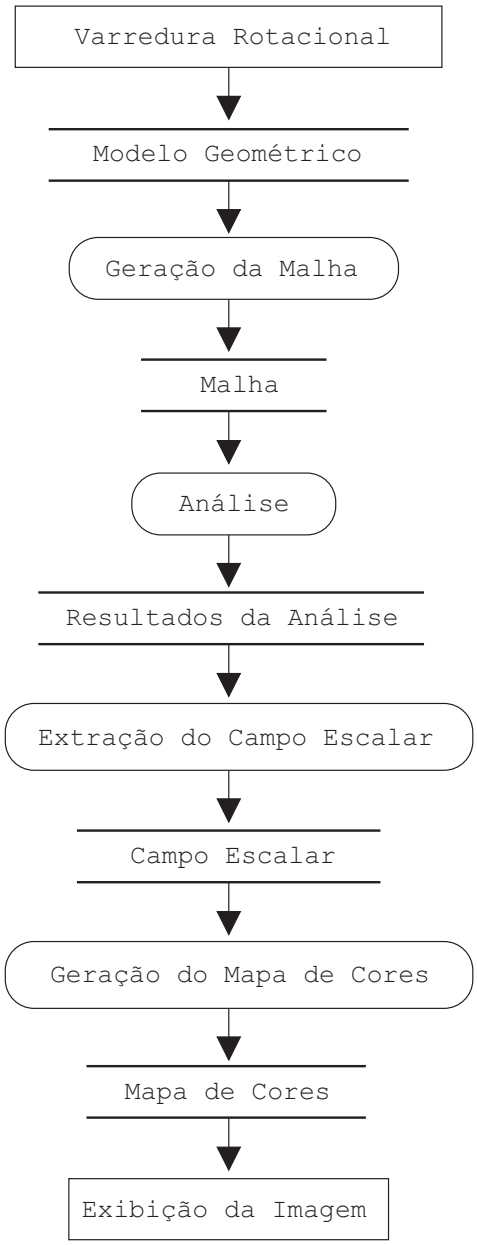

Figura 1.8: Diagrama de fluxo de dados para visualização da Figura 1.7.

Vetores, Tensores e Matrizes

Serão utilizadas letras minúsculas e maiúsculas em **sans serif negrito** para denotar, respectivamente, vetores e tensores no espaço Euclidiano tridimensional. Por exemplo, a equação

$$\mathbf{u} = \mathbf{T} \cdot \mathbf{v} \tag{1.1}$$

denota que, independente do sistema de coordenadas considerado, o vetor $\mathbf{u}$ é obtido do vetor $\mathbf{v}$ por uma *transformação linear* definida pelo tensor $\mathbf{T}$.

Serão utilizadas, também, letras gregas minúsculas para denotar alguns tensores. Consideraremos somente sistemas de coordenadas Cartesianas. Usaremos indistintamente as notações x, y, z e x_1, x_2, x_3 para referenciar os eixos coordenados. Os vetores ortonormais $\mathbf{i}$, $\mathbf{j}$, $\mathbf{k}$ ou $\mathbf{i}_1, \mathbf{i}_2, \mathbf{i}_3$, formam a *base* do sistema de coordenadas.

Os componentes de vetores e tensores em relação a um sistema de coordenadas podem ser convenientemente representados por matrizes. Serão usadas letras minúsculas e maiúsculas em **romano negrito** para denotar matrizes de componentes de vetores e tensores. Em coordenadas Cartesianas a expressão 1.1 é calculada como

$$\mathbf{u} = \mathbf{T}\mathbf{v}. \tag{1.2}$$

Para um tensor denotado por uma letra grega minúscula em negrito, por exemplo, $\boldsymbol{\sigma}$, será utilizada a notação $[\boldsymbol{\sigma}]$ para representar a matriz dos componentes Cartesianos do tensor.

Notação Tensorial Cartesiana

Em notação tensorial Cartesiana, ou *notação indicial*, os componentes Cartesianos (u_1, u_2, u_3) de um vetor $\mathbf{u}$ são denotados por u_i, $i = 1, 2, 3$. Se $\mathbf{n}$ é um vetor unitário na direção do vetor $\mathbf{u}$, os componentes de $\mathbf{n}$ são

$$n_i = \cos\alpha_i, \tag{1.3}$$

onde α_i são os ângulos entre $\mathbf{u}$ e os eixos x_i. Os componentes Cartesianos de $\mathbf{u}$, portanto, são dados pelas três equações

$$u_i = u\ n_i, \tag{1.4}$$

onde u, o módulo de $\mathbf{u}$, é

$$u^2 = u_1^2 + u_2^2 + u_3^2 = \sum_{i=1}^{3} u_i^2. \tag{1.5}$$

Na Equação (1.5) pode-se omitir o símbolo $\sum$ da fórmula porque, em notação tensorial Cartesiana, a ocorrência de dois índices repetidos em um *mesmo termo* representa somatório. A Equação (1.5) pode ser escrita como

$$u^2 = u_i u_i. \tag{1.6}$$

Para simplificar e abreviar algumas equações em notação indicial, podemos utilizar o *delta de Kronecker*, definido como

$$\delta_{ij} = \begin{cases} 1 & \text{se } i = j, \\ 0 & \text{se } i \neq j. \end{cases} \tag{1.7}$$

Como exemplo do uso do delta de Kronecker, a condição de ortogonalidade dos vetores unitários $\mathbf{i}_1, \mathbf{i}_2, \mathbf{i}_3$ pode ser expressa pela equação

$$\mathbf{i}_j \cdot \mathbf{i}_k = \delta_{jk}. \tag{1.8}$$

Em notação tensorial Cartesiana, denotaremos as derivadas parciais como

$$\begin{aligned} \frac{\partial u_i}{\partial x_j} &= u_{i,j}, \\ \frac{\partial^2 u_i}{\partial x_j \partial x_k} &= u_{i,jk}. \end{aligned} \tag{1.9}$$

1.7 Organização do Texto

O texto é organizado em um único volume, no qual apresentamos os fundamentos computacionais e matemáticos utilizados para o desenvolvimento das classes de objetos da BEVL. Apresentamos, também, a funcionalidade das classes principais e exemplos ilustrativos dos recursos de visualização implementados na aplicação de modelagem desenvolvida a partir da BEVL. Além desse capítulo, o volume é organizado em outros seis capítulos, cujos conteúdos são sumarizados a seguir.

Capítulo 2
Modelagem Geométrica
No Capítulo 2 introduziremos os diferentes tipos de modelos geométricos utilizados para representação dos repositórios de dados considerados no trabalho. Apresentaremos, também, a estrutura e funcionalidade das principais classes de objetos da OCL responsáveis pela implementação dos modelos geométricos de OSW.

Capítulo 3
Método dos Elementos de Contorno
No Capítulo 3 apresentaremos um resumo dos fundamentos do Método dos Elementos de Contorno, utilizado no trabalho para análise numérica de sólidos elastostáticos, bem como as classes de objetos da OCL envolvidas na implementação do MEC.

Capítulo 4
Algoritmos de Visualização
No Capítulo 4 descreveremos os algoritmos de visualização escalar e vetorial desenvolvidos no trabalho. As classes de ojetos da BEVL que implementam cada um dos algoritmos também serão descritas.

Capítulo 5
Solução do Problema
No Capítulo 5 descreveremos em detalhes os passos envolvidos na solução do problema de visualização de dados de contorno e domínio resultantes da análise numérica de sólidos elastostáticos pelo MEC, proposto neste trabalho, mostrando como as classes da BEVL são utilizadas em cada uma das etapas. Descreveremos, também, como OSW pode ser empregado no desenvolvimento de duas aplicações de modelagem e visualização para teste das classes de objetos da BEVL. Essas aplicações são denominadas BEG (*Boundary Element Generator) e BEV (*Boundary Element Viewer).

Capítulo 6
Exemplos de Visualização
No Capítulo 6 mostraremos alguns resultados de análise e de visualização obtidos com as aplicações construídas no Capítulo 5, ilustrados com figuras coloridas.

Capítulo 7
Conclusão
No Capítulo 7 finalizaremos com nossas conclusões, sugestões, dificuldades encontradas e contribuições pretendidas com o desenvolvimento do trabalho.

CAPÍTULO 2

Modelagem Geométrica

2.1 Introdução

Conforme visto no Capítulo 1, o processo de visualização inicia com a fase de modelagem, da qual a modelagem geométrica é uma das etapas. A modelagem geométrica envolve o uso do computador para auxiliar a criação, manipulação, manutenção e análise de representações da forma geométrica de objetos bi e tridimensionais [22]. Um modelo geométrico pode ser definido, então, como a representação das características geométricas de um objeto já existente que desejamos construir. Um modelo geométrico pode conter, ainda, outras informações ou *atributos* do objeto, tais como propriedades materiais, forças aplicadas ao objeto, etc.

SCHROEDER [26] define, mais formalmente, um modelo geométrico $\mathcal{M}$ como sendo a representação completa da forma, localização, topologia e outros atributos de um objeto, em termos de sua geometria $\mathcal{G}$, topologia $\mathcal{T}$ e atributos associados $\mathcal{A}$:

$$\mathcal{M} = \{\mathcal{G}, \mathcal{T}, \mathcal{A}\} . \tag{2.1}$$

A geometria $\mathcal{G}$ de um modelo $\mathcal{M}$ pode ser considerada como sendo o conjunto de informações que define, essencialmente, a forma geométrica de um objeto e a precisa localização espacial de todos os seus componentes. Se $\mathcal{M}$ é representado por vértices e faces, por exemplo, as coordenadas dos vértices e a equação da superfície das faces do modelo são informações a respeito de sua geometria $\mathcal{G}$.

Topologia é, segundo WEILER [32], um conjunto de informações invariantes sob determinadas transformações geométricas, tais como rotações e translações. Nesse texto utilizaremos o termo topologia como significado de *relações de adjacência* entre *elementos topológicos* de um modelo geométrico $\mathcal{M}$, tais como vértices, arestas e faces.[1] Uma relação de adjacência é a adjacência, em termos de proximidade física e ordem, de um grupo de elementos topológicos de um tipo em torno de um único elemento topológico de outro tipo [32]. Um exemplo de relação de adjacência pode ser expresso pela seguinte questão: *qual o conjunto de arestas incidentes em um vértice do modelo geométrico?* A topologia $\mathcal{T}$, nesse contexto, é o conjunto de informações armazenadas na estrutura de dados do modelo geométrico $\mathcal{M}$, a respeito de *conectividade* de seus componentes, que permite ou não responder diretamente a questões dessa natureza.

[1] Uma abordagem mais formal do conceito de topologia e sua aplicação em modelagem geométrica pode ser encontrada em [32].

Os atributos $\mathcal{A}$ de um modelo geométrico $\mathcal{M}$ são quaisquer outras informações associadas aos elementos topológicos de $\mathcal{M}$. Os atributos são geralmente categorizados de acordo com tipos específicos de dados, tais como dados escalares, vetoriais ou tensoriais. Funções de valores únicos como temperatura e pressão, por exemplo, são atributos escalares. Grandezas tais como deslocamentos e rotações são exemplos de atributos vetoriais, enquanto deformações e tensões são exemplos de atributos tensoriais. Algoritmos de visualização também são categorizados de acordo com o tipo de dados sobre os quais operam, como veremos no Capítulo 4.

Existe uma variedade de representações geométricas possíveis para um objeto, e a definição de uma representação depende de quais operações serão realizadas sobre o modelo. Esse trabalho trata, principalmente, de visualização de modelos de sólidos. Se estivéssemos considerando somente a necessidade de análise numérica pelo MEC, um modelo $\mathcal{M}$ com geometria $\mathcal{G}$ definida por um conjunto de nós e topologia $\mathcal{T}$ caracterizada por um conjunto de elementos, os quais representariam a superfície e o volume de um sólido, seria suficiente. Embora esse seja o foco principal, outras formas de representação geométrica, bem como as classes OSW de modelagem geométrica utilizadas no desenvolvimento da BEVL, serão introduzidas neste capítulo, ou porque são empregadas nos algoritmos de visualização ou porque são necessárias para uma melhor compreensão do contexto no qual o desenvolvimento da BEVL está inserido.

Na Seção 2.2, por exemplo, apresentamos resumidamente um tipo de modelo de sólidos cuja representação por fronteira é baseada na estrutura de dados semi-aresta proposta por MÄNTYLÄ [17]. Embora não utilizemos diretamente esta forma de representação na implementação dos algoritmos de visualização do Capítulo 4, a incluímos aqui porque, no contexto da análise de sólidos em OSW, um sólido não é primeiramente representado por um modelo de análise, ou seja, por uma malha de elementos de superfície e volume. Na verdade, um sólido é definido por um modelo por fronteira a partir do qual um ou mais modelos de análise são automaticamente criados por um filtro de geração de malhas tridimensionais [19]. A geometria e topologia de modelos de análise são definidas por modelos de decomposição por células descritos na Seção 2.3.

Dois outros tipos de modelos geométricos discutidos neste capítulo são *modelos gráficos* e *modelos de cascas*, Seção 2.4 e Seção 2.5, respectivamente. O primeiro é empregado na representação de primitivos gráficos (pontos, linhas, polígonos) resultantes da aplicação de certos algoritmos de visualização. O segundo é utilizado na representação de superfícies que definem ícones orientados, ou *glyphs*, os quais serão discutidos no Capítulo 4.

Na Seção 2.6, apresentamos os atributos e métodos principais dos objetos de OSW que implementam cada um dos tipos de modelos citados anteriormente. O objetivo é descrever os aspectos mais relevantes da estrutura e funcionalidade das classes que representam os modelos geométricos de OSW, principalmente os modelos de decomposição por células. Para algumas dessas classes, uma discussão mais pormenorizada sobre sua hierarquia de herança e seus métodos não será feita porque a implementação depende de classes acessórias que não são explicadas no texto (veja [22]). Processos de criação de modelos geométricos serão discutidos no Capítulo 4.

2.2 Modelos de Sólidos

A modelagem de sólidos [17] é um ramo da modelagem geométrica que enfatiza a aplicação geral de modelos a partir da criação de representações "completas" de objetos sólidos, isto é, representações que são adequadas para responder questões geométricas arbitrárias de maneira algorítmica, sem a ajuda do usuário. Um modelo de sólidos nos permite distinguir a região de espaço do interior de um volume, da região de espaço exterior, possibilitando a análise de propriedades de massa do objeto sendo representado.

Um modelo de sólidos é dito ser uma representação de variedade de dimensão 2, ou *2-manifold*. Em uma representação *2-manifold*, ou simplesmente *manifold*, cada ponto sobre a superfície de representação do sólido possui *vizinhança* homeomorfa a um disco aberto. Em outras palavras, a superfície, mesmo que exista no espaço tridimensional, é "plana" quando examinada próxima a uma área suficientemente pequena em torno de qualquer ponto dado.

Representação por fronteira, modelos de decomposição por células e geometria sólida construtiva (CSG), são alguns exemplos de formas de representação de sólidos. Modelos por fronteira, especificamente, expressam o conjunto de pontos de um sólido em termos de seu contorno, usualmente definido como uma superfície 2D, representada, geralmente, como uma coleção de faces. Faces, por sua vez, são representadas, também, em termos de seus contornos, definidos como curvas 1D. Modelos de contorno, ou *b-reps* (*boundary representations*), podem ser vistos como uma hierarquia de componentes tais como faces, arestas e vértices.

Em OSW um modelo de sólidos é representado por sua fronteira, empregando-se uma variação da *estrutura de dados semi-aresta* proposta por MÄNTYLÄ [17]. Os elementos da estrutura, brevemente comentados a seguir, constituem uma hierarquia conforme esquematizado na Figura 2.1.

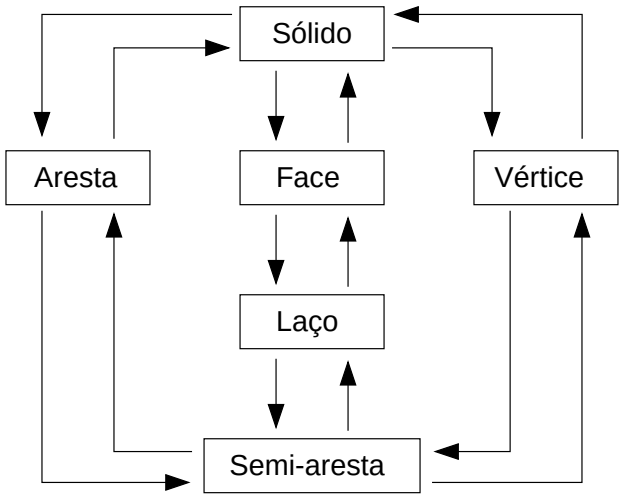

Figura 2.1: Estrutura de dados semi-aresta.

- **Face**. Uma face é um trecho limitado de superfície (plana) de um sólido, definida por um contorno externo e, opcionalmente, por contornos internos representando cavidades em seu interior. Um contorno de uma face, externo ou interno, é definido por um *laço*.
- **Laço**. Um laço é um contorno conectado de uma face, caracterizado por um ciclo de *semi-arestas*. O vetor normal à face, assumido apontar para o exterior do sólido, é definido pelo sentido das semi-arestas do laço externo da face. O sentido das semi-arestas dos laços internos, se existirem, definem normais apontando para o interior do sólido.
- **Semi-aresta**. Em uma representação de variedade de dimensão 2 exatamente duas faces incidem em qualquer aresta do modelo. Em outras palavras, uma aresta é "usada" exatamente por duas faces distintas. Uma semi-aresta representa um dos dois "usos" de uma aresta por uma face do modelo, um segmento orientado de linha de um laço que parte de um *vértice* e chega em outro. Uma semi-aresta mantém uma referência para o vértice do qual parte a semi-aresta; o vértice no qual a semi-aresta chega é dado pela próxima semi-aresta do ciclo de semi-arestas do laço.

- **Aresta**. Uma aresta é um segmento de linha do contorno de duas faces, definido por duas semi-arestas, cada uma pertencente ao laço de cada uma das faces.

- **Vértice**. Um vértice representa um ponto no espaço no qual incide uma ou mais arestas (ou seja, duas ou mais semi-arestas).

A escolha de uma estrutura de dados baseada na estrutura semi-aresta de Mäntylä se justifica por ser essa última uma estrutura de dados que responde facilmente a várias questões relativas à incidência de seus elementos topológicos, possibilitando com isso seu emprego eficiente em um esquema de geração automática de malhas tridimensionais sendo desenvolvido para OSW [19].

A manipulação da estrutura de dados semi-aresta é efetuada por procedimentos, ou operadores, responsáveis pela manutenção da *integridade topológica* do modelo, a qual é garantida pela fórmula de Euler-Poincaré

$$v - e + f = 2(s - h), \tag{2.2}$$

onde v é o número de vértices, e é o número de arestas, f é o número de faces, s é o número de superfícies conectadas e h é o *gênero* da superfície (número de cavidades). Tais procedimentos ou operadores, chamados *operadores de Euler* (definidos detalhamente por MÄNTYLÄ [16]), são usualmente identificados por mnemônicos que caracterizam sua função bem como os tipos de elementos topológicos sobre os quais atuam. Os principais operadores de Euler são sumarizados na Figura 2.2.

Mnemônico	**Significado**
MVFS	**M**ake **V**ertex **F**ace **S**olid
KVFS	**K**ill **V**ertex **F**ace **S**olid
MEV	**M**ake **E**dge **V**ertex
KEV	**K**ill **E**dge **V**ertex
MEF	**M**ake **E**dge **F**ace
KEF	**K**ill **E**dge **F**ace
MEKR	**M**ake **E**dge **K**ill **R**ing
KEMR	**K**ill **E**dge **M**ake **R**ing
MFKRH	**M**ake **F**ace **K**ill **R**ing **H**ole
KFMRH	**K**ill **F**ace **M**ake **R**ing **H**ole

Figura 2.2: Principais operadores de Euler.

Por exemplo, o operador MVFS cria uma instância da estrutura de dados semi-aresta que contém uma única face e um único vértice. Essa nova face possui um laço vazio, sem arestas (note que a fórmula de Euler-Poincaré, Equação (2.2), é satisfeita). Os demais operadores de Euler são detalhadamente explicados em [17].

A criação de modelos de sólidos a partir da utilização direta dos operadores de Euler é uma tarefa que, dependendo da complexidade da geometria do sólido, pode ser bastante complicada. Processos de varredura (translacional, rotacional, etc.) são exemplos de técnicas utilizadas para criação de modelos de sólidos primitivos, os quais podem ser combinados através de *operações booleanas regularizadas* [3]. Alguns desses processos, implementados em OSW, são citados no Capítulo 4.

2.3 Modelos de Decomposição por Células

Um modelo de decomposição por células é uma coleção de *vértices* e uma coleção de *células*. Um vértice é um ponto no espaço tridimensional com coordenadas tomadas em relação a um sistema global de coordenadas. A coleção de vértices do modelo define a geometria $\mathcal{G}$ do modelo. Uma célula é caracterizada por uma coleção ordenada dos vértices nos quais a célula incide, chamada de *lista de conectividade* da célula. A coleção de células do modelo define a topologia $\mathcal{T}$ do modelo.

Topologicamente, uma célula pode ter dimensão zero, um, dois ou três. Uma célula de dimensão topológica 0 é um único vértice no espaço; uma célula de dimensão topológica 1 é uma curva no espaço (uma linha); uma célula de dimensão topológica 2 é uma superfície no espaço (um triângulo, por exemplo); e, uma célula de dimensão topológica 3 é uma região do espaço (um tetraedro, por exemplo).

Geralmente, uma célula possui um *sistema de coordenadas normalizadas*, ou *sistema de coordenadas intrínsecas*. A dimensão de uma base desse sistema é igual à dimensão topológica da célula. (Um único vértice não possui sistema de coordenadas normalizadas.) Com um sistema de coordenadas normalizadas pode-se mais apropriadamente definir alguns atributos de uma célula e executar mais facilmente algumas operações numéricas (por exemplo, integrações evolvendo funções de interpolação). Uma célula de dimensão topológica 1 possui um sistema de coordenadas normalizadas definido por um eixo ξ (ou ξ_1). Uma célula de dimensão topológica 2 possui um sistema de coordenadas normalizadas definido por dois eixos ξ e η (ou ξ_1 e ξ_2). Uma célula de dimensão topológica 3 possui um sistema de coordenadas normalizadas definido por três eixos ξ, η e ζ (ou ξ_1, ξ_2 e ξ_3).

Uma operação importante sobre uma célula é, dadas as coordenadas normalizadas $\boldsymbol{\xi} = \xi_k$ (k variando de 1 até a dimensão topológica da célula), obter as coordenadas globais $\mathbf{x}(x_1, x_2, x_3)$. O mapeamento das coordenadas normalizadas para as coordenadas globais depende das coordenadas dos vértices da célula e da forma da célula, sendo definido por

$$\mathbf{x} = \sum_{i=1}^{r} N_i(\boldsymbol{\xi})\, \mathbf{x}_i, \tag{2.3}$$

onde $\mathbf{x}_i$ são as coordenadas globais do i-ésimo vértice da célula, N_i é a *função de forma*[2] associada ao i-ésimo vértice da célula e r é o número de vértices da célula.

Uma outra operação bastante importante sobre uma célula é a determinação da relação entre as derivadas de uma função qualquer $\mathbf{u}$ em coordenadas globais e as derivadas de $\mathbf{u}$ em coordenadas normalizadas:

$$\begin{bmatrix} \frac{\partial \mathbf{u}}{\partial \xi_1} \\ \frac{\partial \mathbf{u}}{\partial \xi_2} \\ \frac{\partial \mathbf{u}}{\partial \xi_3} \end{bmatrix} = \begin{bmatrix} \frac{\partial x_1}{\partial \xi_1} & \frac{\partial x_2}{\partial \xi_1} & \frac{\partial x_3}{\partial \xi_1} \\ \frac{\partial x_1}{\partial \xi_2} & \frac{\partial x_2}{\partial \xi_2} & \frac{\partial x_3}{\partial \xi_2} \\ \frac{\partial x_1}{\partial \xi_3} & \frac{\partial x_2}{\partial \xi_3} & \frac{\partial x_3}{\partial \xi_3} \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial \mathbf{u}}{\partial x_1} \\ \frac{\partial \mathbf{u}}{\partial x_2} \\ \frac{\partial \mathbf{u}}{\partial x_3} \end{bmatrix} = \mathbf{J} \cdot \begin{bmatrix} \frac{\partial \mathbf{u}}{\partial x_1} \\ \frac{\partial \mathbf{u}}{\partial x_2} \\ \frac{\partial \mathbf{u}}{\partial x_3} \end{bmatrix}, \tag{2.4}$$

onde $\mathbf{J}$ é a *matriz Jacobiana* de transformação entre o sistema de coordenadas globais e o sistema de coordenadas normalizadas. A relação inversa é expressa por:

$$\begin{bmatrix} \frac{\partial \mathbf{u}}{\partial x_1} \\ \frac{\partial \mathbf{u}}{\partial x_2} \\ \frac{\partial \mathbf{u}}{\partial x_3} \end{bmatrix} = \mathbf{J}^{-1} \cdot \begin{bmatrix} \frac{\partial \mathbf{u}}{\partial \xi_1} \\ \frac{\partial \mathbf{u}}{\partial \xi_2} \\ \frac{\partial \mathbf{u}}{\partial \xi_3} \end{bmatrix}. \tag{2.5}$$

[2]Para caracterizarmos totalmente a geometria de um elemento, utilizamos funções de interpolação que nos fornecem, a partir das coordenadas dos nós, as coordenadas de qualquer ponto sobre o elemento. Essas funções são chamadas *funções de forma*.

Tomando-se **r** no lugar de **u**, onde **r** é o vetor que define um ponto qualquer, Figura 2.3, podemos escrever a partir das relações anteriores o diferencial de volume como sendo [4]

$$d\Omega = \left| \frac{\partial \mathbf{r}}{\partial \xi_1} \times \frac{\partial \mathbf{r}}{\partial \xi_2} \cdot \frac{\partial \mathbf{r}}{\partial \xi_3} \right| \cdot d\xi_1 \cdot d\xi_2 \cdot d\xi_3 = |\mathbf{J}| \cdot d\xi_1 \cdot d\xi_2 \cdot d\xi_3, \tag{2.6}$$

e o diferencial de área $d\Gamma$ como

$$d\Gamma = \left| \frac{\partial \mathbf{r}}{\partial \xi_1} \times \frac{\partial \mathbf{r}}{\partial \xi_2} \right| \cdot d\xi_1 \cdot d\xi_2 = |\mathbf{G}| \cdot d\xi_1 \cdot d\xi_2. \tag{2.7}$$

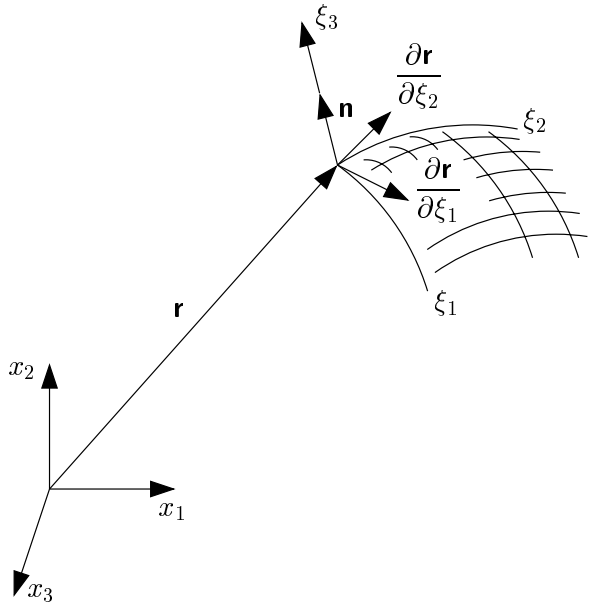

Figura 2.3: Sistemas de coordenadas globais e adimensionais.

As relações (2.6) e (2.7), dadas em função de elementos da matriz Jacobiana **J**, são importantes porque, conforme veremos no Capítulo 3, podem ser utilizados para integração numérica de termos de superfície e volume tais como

$$\int_\Gamma \mathbf{u}\, d\Gamma \quad \text{e} \quad \int_\Omega \mathbf{u}\, d\Omega, \tag{2.8}$$

os quais tornam-se, respectivamente

$$\int_\Gamma \mathbf{u}\ |\mathbf{G}|\ d\xi_1 d\xi_2 \quad \text{e} \quad \int_\Omega \mathbf{u}\ |\mathbf{J}|\ d\xi_1 d\xi_2 d\xi_3, \tag{2.9}$$

assumindo-se **u** uma função das coordenadas normalizadas $\boldsymbol{\xi}$.

A seguir apresentaremos as células utilizadas nos modelos do trabalho. As funções de forma e suas derivadas [4, 33] também são apresentadas.

- **Triângulo**. O triângulo é uma célula de dimensão topológica 2, definida por uma lista ordenada de três vértices. A ordem dos vértices especifica a direção da normal à superfície do triângulo, de acordo com a regra da mão direita. A Figura 2.4 mostra uma célula de dimensão topológica 2 triangular, cujas funções de forma são

$$\begin{aligned} N_1 &= 1 - \xi - \eta, \\ N_2 &= \xi, \\ N_3 &= \eta. \end{aligned} \tag{2.10}$$

As derivadas das funções de forma em relação às coordenadas normalizadas podem ser matricialmente organizadas como

$$\mathbf{D} = \begin{bmatrix} -1 & -1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}, \tag{2.11}$$

onde D_{ij} denota a derivada da função de forma N_i em relação à coordenadda ξ_j.

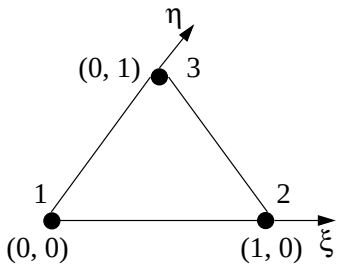

Figura 2.4: Triângulo.

- **Tetraedro**. O tetraedro é uma célula de dimensão topológica 3, definido por uma lista de quatro vértices não planares. O tetraedro possui seis arestas e quatro faces triangulares, como mostrado na Figura 2.5. As funções de forma do tetraedro são dadas por

$$\begin{aligned} N_1 &= 1 - \xi - \eta - \zeta, \\ N_2 &= \xi, \\ N_3 &= \eta, \\ N_4 &= \zeta, \end{aligned} \tag{2.12}$$

e suas derivadas pela matriz

$$\mathbf{D} = \begin{bmatrix} -1 & -1 & -1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{2.13}$$

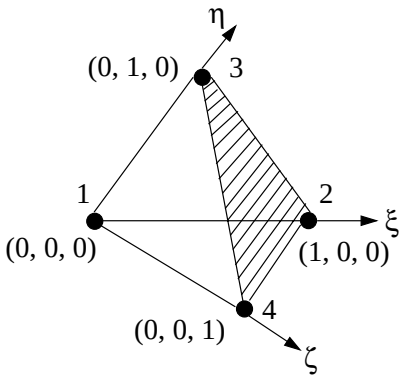

Figura 2.5: Tetraedro.

2.4 Modelos Gráficos

No Capítulo 1, definimos visualização como sendo a transformação de dados de um objeto em primitivos gráficos que, uma vez exibidos, representam algum tipo de informação a respeito do objeto. Um filtro de extração de isosuperfícies, por exemplo, pode transformar atributos escalares de um modelo em polígonos que representam tais isosuperfícies. Em OSW, coleções hierárquicas de primitivos gráficos são organizadas nos chamados *modelos gráficos*.

O uso de modelos gráficos se justifica por ser esta uma representação "econômica", que não contém explicitamente nenhuma informação a respeito das relações de adjacência entre seus componentes. O principal objetivo é prover um repositório de primitivos — usualmente resultantes da aplicação de certos algoritmos de visualização, conforme veremos no Capítulo 4 — que serão convertidos em imagens. A geometria $\mathcal{G}$ de um modelo gráfico é definida por uma coleção de pontos denominados *vértices*, os quais mantêm, além da posição espacial, atributos tais como cor, normal, valor escalar e valor vetorial.

Consideraremos os seguintes tipos de primitivos gráficos, Figura 2.6:

- **Ponto**. Um ponto é um primitivo definido por um único vértice. Um ponto pode ser desenhado em forma de cruz, "X", retângulo, triângulo, círculo, retângulo preenchido, triângulo preenchido e círculo preechido. (Os algoritmos da BEVL não geram primitivos do tipo ponto.)
- **Linha**. Uma linha é um primitivo definido por dois vértices. Linhas são utilizadas para representação de isolinhas.
- **Triângulo**. Um triângulo é um primitivo definido por três vértices coplanares.[3]
- **Polígono**. Um polígono é um primitivo definido por uma seqüência $v_1, v_2, \cdots, v_n$ de n vértices coplanares, sendo a i-aresta do polígono, $1 \leq i \leq n$, definida pelos vértices v_i e v_j, onde $j = i \mod n + 1$. Polígonos, bem como triângulos, são utilizados na representação de isosuperfícies.

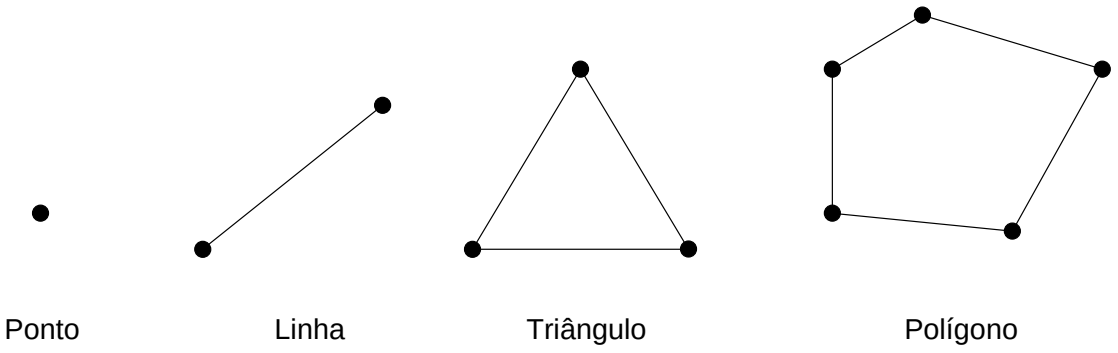

Figura 2.6: Tipos de primitivos de um modelo gráfico.

Em adição, desde que um modelo gráfico é uma coleção hierárquica de primitivos, um modelo gráfico é também um tipo de primitivo gráfico. Além da coleção de primitivos que o compõe, um modelo gráfico mantém ainda a coleção de vértices nos quais seus primitivos incidem. Dessa forma não ocorrem, em um modelo gráfico, vértices ocupando a mesma posição espacial. Ao invés disso, somente um vértice é mantido e "usado" por um ou mais primitivos. (Nenhuma informação a respeito dos "usos" de um vértice, contudo, é mantida na representação.)

[3] Um primitivo gráfico do tipo triângulo e uma célula triangular são objetos distintos.

2.5 Modelos de Cascas

Em OSW, objetos 3D constituídos de faces planares e arestas fio de arame e que, portanto, não necessariamente definem a superfície de variedade de dimensão 2 de um sólido, podem ser representados por *modelos de cascas*. Originalmente propostos para representação de estruturas laminares constituídas de barras e de superfícies (abertas ou fechadas) analisadas pelo MEF, modelos de cascas são usados, nesse trabalho, para representar superfícies que definem ícones orientados ou *glyphs*, tal como veremos no Capítulo 4.

Um modelo de casca é uma coleção hierárquica dos elementos topológicos ilustrados na Figura 2.7, e brevemente comentados a seguir. Maiores detalhes podem ser vistos em [22].

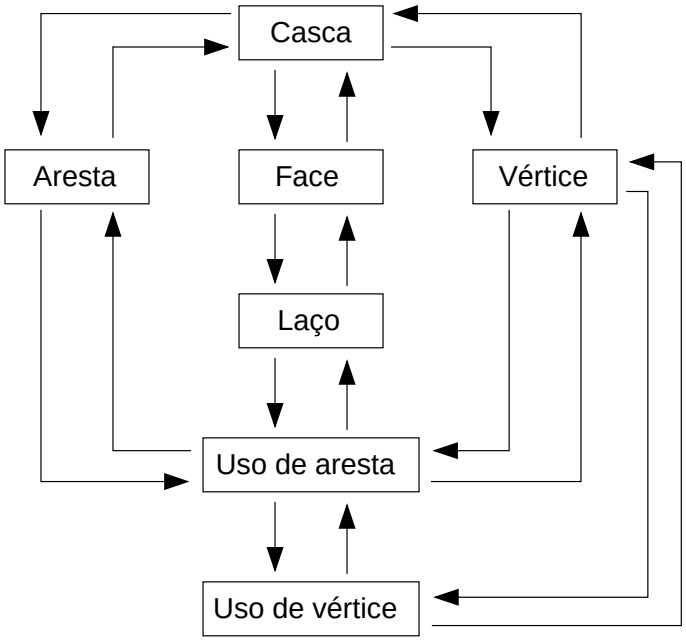

Figura 2.7: Estrutura de dados de um modelo de cascas.

- **Face**. Uma face de um modelo de casca é uma superfície planar definida por conjuntos de pontos conectados chamados laços. Uma face sempre possui pelo menos um laço que define seu contorno externo. Outros laços, se existirem, representam contornos de cavidades da face.
- **Laço**. Um laço é um contorno conectado definido por um ciclo de *usos de aresta*.
- **Uso de aresta**. Em uma representação não *manifold* pode-se ter uma, duas ou mais faces incidindo em uma única aresta. Em um modelo de casca, uma aresta é representada somente uma vez, mas mantém-se explicitamente, para cada face incidente na aresta, a informação que a face "usa" a aresta.
- **Aresta**. Uma aresta é um segmento do contorno de uma ou mais faces definido por dois vértices. Uma aresta mantém uma lista com todos os seus "usos".
- **Uso de vértice**. Pode-se ter várias arestas incidindo em um único vértice. O vértice é representado somente uma vez, mas mantém-se explicitamente todos os "usos" do vértice.

- **Vértice**. Um vértice representa um ponto no espaço no qual incide uma ou mais arestas. Um vértice mantém uma lista de todos os seus "usos".

A estrutura de dados de modelos de cascas permite responder de forma eficiente às mesmas questões topológicas sobre incidência feitas a um modelo de sólido. Contudo, devido ao fato de um modelo de cascas ser uma representação não *manifold* e, portanto, mais genérica que um modelo de sólido, as informações sobre adjacência são mantidas explicitamente na estrutura (através dos "usos" de arestas e vértices), o que leva a um maior consumo de memória.

A manipulação da estrutura de dados de modelos de cascas é efetuada por procedimentos, ou operadores, bastante simples chamados *operadores de modelagem de cascas*. Assim como os operadores de Euler, esses são identificados por mnemônicos que caracterizam sua função bem como os tipos de elementos topológicos sobre os quais atuam. Os principais operadores de modelagem de cascas são sumarizados na Figura 2.8. Por exemplo, o operador MEV cria uma nova aresta e um novo vértice incidentes em uma face da estrutura de dados de um modelo de casca. Os operadores de modelos de cascas são detalhadamente explicados em [22].

Mnemônico	**Significado**
MV	**M**ake **V**ertex
KV	**K**ill **V**ertex
MF	**M**ake **F**ace
KF	**K**ill **F**ace
ME	**M**ake **E**dge
MH	**M**ake **H**ole
MVF	**M**ake **V**ertex **F**ace
MEV	**M**ake **E**dge **V**ertex
MWE	**M**ake **W**ireframe **E**dge
KWE	**K**ill **W**ireframe **E**dge

Figura 2.8: Principais operadores de modelagem de cascas.

Tal como ocorre com modelos de sólidos, a criação de modelos de cascas a partir da utilização direta dos operadores de modelagem é uma tarefa que pode ser bastante complicada. Processos de varredura (translacional, rotacional, etc.) são técnicas que também podem ser utilizadas para criação de modelos de cascas (ver Capítulo 4).

2.6 Modelos Geométricos em OSW

Em OSW, um modelo é um objeto de uma classe derivada da classe abstrata `tModel`, a qual representa um modelo geométrico genérico manipulado por uma aplicação. Modelos de sólidos, de decomposição por células, gráficos e de cascas são todos objetos de classes derivadas de `tModel`. A Figura 2.9 mostra parte da hierarquia de classes de modelos geométricos de OSW, onde o símbolo $\triangle$ denota herança. Como vimos nas seções subseqüentes, porém, os modelos de OSW são bastante distintos entre si. Que tipo de generalização, então, `tModel` representa?

Todo modelo geométrico $\mathcal{M}$ de OSW é constituído por *coleções* de componentes agrupados de forma a definir a geometria $\mathcal{G}$, a topologia $\mathcal{T}$ e os atributos $\mathcal{A}$ do modelo. Um modelo de decomposição por células, por exemplo, é constituído de coleções de células e vértices; um modelo gráfico é uma coleção de primitivos; um modelo de sólidos é uma coleção de faces,

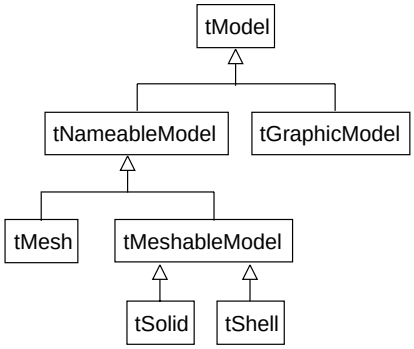

Figura 2.9: Parte das classes de modelos OSW.

vértices e arestas. Em programação orientada a objetos, um objeto que é constituído de coleções de outros objetos é chamado *contêiner*. Portanto, todo modelo OSW é um contêiner. Embora a *estrutura de dados* dos componentes dos modelos OSW sejam distintas, podemos tentar estabelecer seu *comportamento* comum.

Para isso, tomemos como exemplo o problema de visualização fio de arame ou com tonalização da estrutura de um modelo geométrico. Em OSW, os componentes responsáveis pela síntese de imagens de um modelo são objetos de classes derivadas da classe abstrata `tRenderer`, comentada no Capítulo 4. Sem entrar em detalhes sobre algoritmos de geração de imagens (veja [7, 24]), os sintetizadores OSW utilizados em aplicações baseadas na BEVL criam uma imagem de um modelo processando suas arestas, faces, ou ambos. Um modelo de decomposição por células, por exemplo, é definido por células e vértices, não possuindo faces ou arestas. Modelos de cascas e sólidos, por sua vez, são definidos diretamente por faces e arestas. Para conseguirmos visualizar as faces e/ou arestas de um modelo de decomposição por células, então, este deve "extraí-las" de suas células. Modelos de cascas e de sólidos não necessitam dessa operação.

Independente de sua representação, um modelo geométrico deve possibilitar que outros objetos tenham acesso a seus vértices, arestas e faces, mesmo que esses componentes não sejam diretamente armazenados no modelo. (Isso permite, por exemplo, que objetos sintetizadores de imagens funcionem para *qualquer* modelo de OSW.) O acesso aos objetos de um contêiner, em orientação a objetos, é executado por um objeto chamado *iterador*, responsável por "atravessar" um contêiner e disponibilizar os dados de seus componentes para outros objetos. Uma das características do comportamento comum de todo modelo geométrico OSW, definida na classe `tModel`, é justamente esse: fornecer iteradores para vértices, arestas e faces do modelo, objetos das classes `tVertexIterator`, `tEdgeIterator` e `tFaceIterator`, respectivamente. Parte da definição da classe `tModel` é listada no Programa 2.1.

A classe `tModel`, derivada da classe acessória `tSharedObject`, declara três métodos que retornam iteradores de vértices, arestas e faces de um modelo: `GetVertexIterator()`, `GetEdgeIterator()` e `GetFaceIterator()`, linhas 4, 8 e 12, respectivamente. `tModel` define também métodos virtuais para aplicação de transformações geométricas, linha 17, e extração de isopontos, isolinhas e isosuperfícies de um modelo, linha 18. Uma explicação mais detalhada dos iteradores de componentes de modelos geométricos será dada a seguir.

2.6.1 Iteradores de Componentes de Modelos Geométricos

Em OSW uma classe de iterador de objetos de uma classe `T` é definida através da classe paramétrica `tIterator<T>`. A definição parcial da classe e a implementação **`inline`** dos principais métodos são listados no Programa 2.2.

```
1    class tModel:  public tSharedObject
2    {
3       public:
4          tVertexIterator GetVertexIterator()
5          {
6             return tVertexIterator(MakeInternalVertexIterator());
7          }
8          tEdgeIterator GetEdgeIterator()
9          {
10            return tEdgeIterator(MakeInternalEdgeIterator());
11         }
12         tFaceIterator GetFaceIterator()
13         {
14            return tFaceIterator(MakeInternalFaceIterator());
15         }
16         ...
17         virtual void Transform(const t3DTransfMatrix&);
18         virtual void Contour(double, double*, tGraphicModel&) const;
19
20      private:
21         virtual tInternalVertexIterator* MakeInternalVertexIterator();
22         virtual tInternalEdgeIterator* MakeInternalEdgeIterator();
23         virtual tInternalFaceIterator* MakeInternalFaceIterator();
24   }; // tModel
```

Programa 2.1: Parte da classe `tModel`.

O método `Restart()`, linha 10, é responsável por iniciar o "atravessamento" da coleção de objetos da classe `T` do iterador, fazendo com que o objeto corrente do iterador seja o primeiro elemento da coleção. O método `Current()`, linha 14, retorna um ponteiro para o objeto corrente do iterador. O "atravessamento" da coleção é efetuado através dos métodos **`operator`** `++()`, linhas 23 e 29. Esses métodos retornam um ponteiro para o próximo elemento da coleção, o qual passa a ser o objeto corrente do iterador. O método **`operator int()`** permite verificar se o "atravessamento" chegou ao seu final. Note que

1. um iterador de componentes de modelos geométricos de OSW permite somente o "atravessamento" em uma direção, do primeiro ao último elemento da coleção;

2. uma classe de iterador não faz nenhuma suposição sobre a estrutura de dados utilizada na implementação de componentes da classe `T`. Essa independência é possível porque a implementação dos métodos de "atravessamento" de um iterador é efetuada por *iteradores internos*, explicados a seguir.

Um iterador interno é um objeto que "conhece" a estrutura de dados empregada na representação interna da coleção de componentes de um contêiner e, portanto, responsável pelo "atravessamento" efetivo da coleção. Cada contêiner de OSW contém um iterador interno específico para cada classe de componente armazenado ou que pode ser extraído de um contêiner. Uma classe de iterador interno de objetos de uma classe `T` é definida através da classe paramétrica `tInternalIterator<T>`, parcialmente listada no Programa 2.3.

A funcionalidade interna da classe é definida pelos métodos virtuais puros `MakeFirstObject()` e `MakeNext()`, declarados nas linhas 30 e 31, respectivamente. O primeiro retorna um ponteiro para o primeiro objeto da coleção. Dependendo da classe do modelo geométrico,

```
1   template<class T>
2   class tIterator
3   {
4     public:
5       tIterator();
6       tIterator(const tIterator&);
7       tIterator(tInternalIterator<T>*);
8       ...
9
10      void Restart()
11      {
12        InternalIterator->Start();
13      }
14      T* Current()
15      {
16        return InternalIterator->GetCurrent();
17      }
18      tIterator& operator =(const tIterator&);
19      operator int() const
20      {
21        return Current() != 0;
22      }
23      T* operator ++(int)
24      {
25        T* cur = Current();
26        InternalIterator->GetNext();
27        return cur;
28      }
29      T* operator ++()
30      {
31        return InternalIterator->GetNext();
32      }
33
34    private:
35      tInternalIterator<T>* InternalIterator;
36      static tInternalNullIterator<T> Null;
37  }; // tIterator
```

Programa 2.2: Classe paramétrica `tIterator<T>`.

esse objeto pode existir de fato ou ser temporariamente criado. A implementação de `MakeFirstObject()` de um iterador de faces de um modelo de decomposição por células, por exemplo, cria uma face temporária correspondente à primeira face da primeira célula (de dimensão topológica 2 ou 3) e retorna um ponteiro para a face criada. O segundo método retorna um ponteiro para o próximo objeto da coleção, se existir. Classes derivadas de `tInternalIterator<T>` devem sobrecarregar esses métodos. Classes geradas pela classe paramétrica `tInternallNullIterator<T>` (derivada de `tInternallIterator<T>`) definem versões de `MakeFirstObject()` e `MakeNext()` que retornam ponteiros iguais a zero (por padrão chamados iteradores nulos).

A partir da classe paramétrica `tIterator<T>`, discutida anteriormente, podemos definir as classes de iteradores e iteradores internos de vértices, arestas e faces como:

```
1    template<class T>
2    class tInternalIterator:  public tSharedObject
3    {
4       public:
5          void Start()
6          {
7             SetCurrent(MakeFirstObject());
8          }
9          T* GetCurrent()
10         {
11            return Object;
12         }
13         T* GetNext()
14         {
15            SetCurrent(MakeNext());
16            return Object;
17         }
18         ...
19
20      private:
21         T* Object;
22         bool DeleteObject;
23
24         void SetCurrent(T* object)
25         {
26            if (DeleteObject && Object)
27               delete Object;
28            Object = object;
29         }
30         virtual T* MakeFirstObject() = 0;
31         virtual T* MakeNext() = 0;
32   }; // tInternalIterator
```

Programa 2.3: Parte da classe paramétrica `tInternalIterator<T>`.

```
typedef tIterator<tVertex> tVertexIterator;
typedef tInternalIterator<tVertex> tInternalVertexIterator;
typedef tIterator<tEdge> tEdgeIterator;
typedef tInternalIterator<tEdge> tInternalEdgeIterator;
typedef tIterator<tFace> tFaceIterator;
typedef tInternalIterator<tFace> tInternalFaceIterator;
```

As classes `tVertex`, `tEdge` e `tFace` representam, respectivamente, vértices, arestas e faces genéricos que são compreendidos por todos os modelos geométricos de OSW e por outros objetos que precisam acessar seus componentes (tais como sintetizadores de imagens).

Um vértice de um modelo geométrico OSW é um objeto da classe `tVertex`, parcialmente listada no Programa 2.4. Um vértice é identificado por um número inteiro `Number`, declarado na linha 4. O ponto que define a posição espacial do vértice é mantido em `Position`, linha 5. (Em OSW, um ponto é definido por um vetor 3D, um objeto da classe acessória `t3DVector` [22].) Os demais atributos de um vértice (utilizados nos algoritmos do Capítulo 4) são comentados a seguir.

```
1    class tVertex
2    {
3       public:
4          int Number;
5          t3DVector Position;
6          tColor Color;
7          t3DVector Normal;
8          double Scalar;
9          t3DVector Vector;
10         ...
11   }; // tVertex
```

Programa 2.4: Parte da interface da classe `tVertex`.

- `Color`. O atributo `Color`, linha 6, armazena o valor da cor RGB do vértice. Em OSW, uma cor RGB é definida por um objeto da classe acessória `tColor`.
- `Normal`. Um vetor normal $\mathbf{n}$ é um versor utilizado para representar uma direção (um versor é um vetor de módulo unitário, ou seja, $|\mathbf{n}| = 1$). `Normal`, linha 7, é usado por sintetizadores de imagens para determinar as tonalizações nas faces que incidem no vértice.
- `Scalar`. O atributo `Scalar`, linha 8, armazena o valor de uma grandeza escalar qualquer atribuída ao vértice, tal como temperatura ou pressão.
- `Vector`. O atributo `Vector`, linha 10, armazena o valor de uma grandeza vetorial qualquer atribuída ao vértice, tal como deslocamento ou força de superfície.

Uma aresta genérica de um modelo de OSW é um segmento de linha definido por dois vértices, representado por um objeto da classe `tEdge`, Programa 2.5. Os atributos `V0` e `V1`, declarados nas linhas 6 e 7, identificam os vértices que formam a aresta. O construtor da classe, linha 4, toma como entrada dois vértices e os ajusta para `V0` e `V1`.

```
1    class tEdge
2    {
3       public:
4          tVertex(tVertex*, tVertex*);
5
6          tVertex* V0;
7          tVertex* V1;
8          ...
9    }; // tEdge
```

Programa 2.5: Parte da interface da classe `tEdge`.

Uma face genérica de um modelo de OSW é um objeto de uma classe derivada da classe abstrata `tFace`, listada no Programa 2.6.

Uma face genérica é capaz de fornecer, da mesma forma que os modelos geométricos, iteradores para seus vértices e arestas, através dos métodos `GetVertexIterator()`, linha 4, e `GetEdgeIterator()`, linha 8, respectivamente. Como discutido anteriormente, o "atravessamento" de um contêiner é implementado por iteradores internos. No caso de uma face, um iterador interno de vértices é obtido com o método virtual privado `MakeInternalVertexIterator()`, declarado na linha 18. O método virtual `MakeInternalEdgeIterator()`,

```
1   class tFace
2   {
3      public:
4         tVertexIterator GetVertexIterator()
5         {
6            return tVertexIterator(MakeInternalVertexIterator());
7         }
8         tEdgeIterator GetEdgeIterator()
9         {
10           return tEdgeIterator(MakeInternalEdgeIterator());
11        }
12
13        virtual t3DVector Normal() const = 0;
14        virtual tMaterial GetMaterial() const = 0;
15        virtual int GetNumber() const = 0;
16
17     private:
18        virtual tInternalVertexIterator* MakeInternalVertexIterator();
19        virtual tInternalEdgeIterator* MakeInternalEdgeIterator();
20  }; // tFace
```

Programa 2.6: Classe `tFace`.

linha 19, fornece um iterador interno para as arestas da face. As versões desses métodos em `tFace`, usualmente sobrecarregados em classes derivadas, retornam iteradores internos nulos de vértices e arestas.

Em adição, uma face deve ser capaz de fornecer seu vetor normal (somente faces planas são admitidas na versão atual de OSW), o material do qual é constituída (objeto da classe acessória `tMaterial`) e seu número (o qual pode ser usado como identificador da face). Essa funcionalidade é definida pelos métodos virtuais puros `Normal()`, `GetMaterial()` e `GetNumber()`, declarados nas linhas 13, 14 e 15, respectivamente. Esses métodos devem ser sobrecarregados em classes concretas derivadas de `tFace`.

Nas subseções seguintes apresentaremos uma visão geral das principais classes de OSW envolvidas na implementação dos modelos geométricos utilizados no desenvolvimento da BEVL.

2.6.2 Classes de Modelo de Sólido

Um modelo de sólido de variedade de dimensão 2 em OSW é um objeto da classe `tSolid`. A classe deriva de `tMeshableModel` e de `tNameableModel` (Figura 2.9), o que significa que um modelo de sólido pode ser transformado em um modelo de decomposição por células e ter um nome. A classe `tSolid`, parcialmente listada no Programa 2.7, fornece uma implementação orientada a objetos da estrutura de dados semi-aresta de Mäntylä e dos operadores de Euler. As classes internas `tVertex`, `tEdge`, `tFace`, `tLoop` e `tHalfEdge`, declaradas nas linhas 4 a 8, representam, respectivamente, vértices, arestas, faces, laços e semi-arestas da estrutura de dados semi-aresta. Os principais operadores de Euler, listados na Figura 2.2, são implementados pelos métodos declarados nas linhas 18 a 26.

A implementação da estrutura de dados de um modelo de sólido é baseada em listas lineares duplamente encadeadas de vértices, arestas e faces, definidas nas linhas 11 a 16. Em OSW uma classe de lista linear duplamente encadeada de uma classe `T` é definida através da classe paramétrica `tDoubleListImp<T>`, cuja definição parcial é listada no Programa 2.8.

```
1   class tSolid: public tMeshableModel
2   {
3     public:
4       class tVertex: public tMeshableModel::tVertex {...};
5       class tEdge {...};
6       class tFace: public ::tFace {...};
7       class tLoop {...};
8       class tHalfEdge {...};
9       ...
10    private:
11      typedef tDoubleListImp<tSolid::tVertex> tVertices;
12      typedef tDoubleListImp<tSolid::tEdge> tEdges;
13      typedef tDoubleListImp<tSolid::tFace> tFaces;
14      tVertices Vertices;
15      tEdges Edges;
16      tFaces Faces;
17
18      tHalfEdge* mvfs(int, int, const t3DVector&);
19      void lmev(tHalfEdge*, tHalfEdge*, int, const t3DVector&);
20      void lkev(tHalfEdge*, tHalfEdge*);
21      tFace* lmef(tHalfEdge*, tHalfEdge*, int);
22      void lkef(tHalfEdge*, tHalfEdge*);
23      tHalfEdge* lkemr(tHalfEdge*, tHalfEdge*);
24      void lmekr(tHalfEdge*, tHalfEdge*);
25      void lkfmrh(tFace*, tFace*);
26      tFace* lmfkrh(tLoop*, int);
27
28      tInternalVertexIterator* MakeInternalVertexIterator();
29      tInternalEdgeIterator* MakeInternalEdgeIterator();
30      tInternalFaceIterator* MakeInternalFaceIterator();
31      ...
32      friend class tSolidSweeper;
33  }; // tSolid
```

Programa 2.7: Parte da classe `tSolid`.

Uma lista duplamente encadeada é controlada por dois ponteiros `Head` e `Tail`, que armazenam o endereço do primeiro e do último elemento da lista, respectivamente. Uma lista duplamente encadeada está apta para:

- Adicionar elementos na lista. A classe possui três métodos que realizam a inserção de um elemento `e` na lista. O método `AddAtHead(e)`, linha 10, insere o elemento no início da lista. Os métodos `AddAtTail(e)`, linha 11, e `Add(e)`, linha 12, inserem o elemento no final da lista.
- Remover elementos da lista. O método `Remove(e)`, linha 16, remove o elemento `e` da lista.
- Fornecer o primeiro elemento da lista. O método `PeekHead()`, declarado na linha 18, retorna a "cabeça" da lista, isto é, o primeiro elemento da lista.
- Fornecer o último elemento da lista. O método `PeekTail()`, declarado na linha 19, retorna a "cauda" da lista, isto é, o último elemento da lista.

```
1    template<class T>
2    class tDoubleListImp
3    {
4       public:
5          tDoubleListImp():
6             Head(0), Tail(0), NumberOfElements(0)
7          {}
8          ...
9
10         void AddAtHead(T*);
11         void AddAtTail(T*);
12         void Add(T* e)
13         {
14            AddAtTail(e);
15         }
16         void Remove(T*);
17         void Flush();
18         T* PeekHead()const;
19         T* PeekTail()const;
20         uint GetNumberOfElements() const;
21         bool IsEmpty() const;
22
23      protected:
24         T* Head;
25         T* Tail;
26         uint NumberOfElements;
27   }; // tDoubleListImp
28   ...
```

Programa 2.8: Classe paramétrica `tDoubleListImp<T>`.

Uma lista duplamente encadeada é um contêiner. A classe de iterador de listas duplamente encadeadas de objetos de uma classe `T` é uma instância da classe paramétrica `tDoubleListIteratorImp<T>`. Os métodos da classe são similares àqueles dos iteradores de componentes de modelos geométricos, possibilitando o "atravessamento" da coleção de objetos do tipo `T` do contêiner.

2.6.3 Classes de Modelos de Decomposição por Células

Um modelo de decomposição por células em OSW é um objeto da classe `tMesh`, ou de uma classe derivada de `tMesh`. Modelos de decomposição por células são empregados na representação dos dados geométricos e de análise de malhas de elementos finitos e/ou de contorno. O Programa 2.9 mostra parte da classe `tMesh`.

A principal funcionalidade da classe `tMesh` é gerenciar a coleção de nós e de células do modelo. Um modelo de decomposição por células é capaz de:

- Fornecer o número de nós e células. Os métodos `GetNumberOfNodes()` e `GetNumberOfCells()`, linhas 4 e 5, retornam, respectivamente, a quantidade de nós e células que constituem o modelo.
- Fornecer nós e células específicos. O método `GetNote(i)`, declarado na linha 6, retorna um ponteiro para o `i`-ésimo nó do modelo. O método `GetCell(i)`, declarado na linha

```
1    class tMesh: public tNameableModel
2    {
3       public:
4          int GetNumberOfNodes() const;
5          int GetNumberOfCells() const;
6          tNode* GetNode(int) const;
7          tCell* GetCell(int) const;
8          tNodeIterator GetNodeIterator() const;
9          tCellIterator GetCellIterator() const;
10
11         void AddNode(tNode*);
12         void DeleteNode(tNode*);
13         void AddCell(tCell*);
14         void DeleteCell(tCell*);
15
16         void Transform(const t3DTransfMatrix&);
17         void Contour(double, double*, tGraphicModel&) const;
18         ...
19      private:
20         tInternalVertexIterator* MakeInternalVertexIterator();
21         tInternalEdgeIterator* MakeInternalEdgeIterator();
22         tInternalFaceIterator* MakeInternalFaceIterator();
23   }; // tMesh
```

Programa 2.9: Parte da classe `tMesh`.

7 retorna um ponteiro para a `i`-ésima célula do modelo. As classes `tNode` e `tCell` serão explicadas a seguir.

- Fornecer iteradores de nós e células. Um iterador de nós é um objeto da classe `tNodeIterator` e um iterator de células é um objeto da classe `tCellIterator`, responsáveis por "atravessar" a coleção de nós e células do modelo, respectivamente. Os métodos `GetNodeIterator()` e `GetCellIterator()`, declarados nas linhas 8 e 9, fornecem, respectivamente os iteradores de nós e células.

- Adicionar e remover nós. Os métodos `AddNode()` e `DeleteNode()`, declarados nas linhas 11 e 12, respectivamente, são responsáveis pela adição e remoção de nós do modelo.

- Adicionar e remover células. Células são adicionadas ou removidas do modelo pelos métodos `AddCell()` e `DeleteCell()`, declarados nas linhas 13 e 14, respectivamente.

Em adição, a classe sobrecarrega os métodos virtuais `Transform()` e `Contour()`, linhas 16 e 17, herdados de `tModel`. Os iteradores internos de vértices, arestas e faces de uma malha são fornecidos pelos métodos `MakeInternalVertexIterator()`, `MakeInternalEdgeIterator()` e `MakeInternalFaceIterator()`, declarados nas linhas 20, 21 e 22.

As coleções de nós e células de um modelo de decomposição por células são implementadas como listas duplamente encadeadas, cujas classes (e seus respectivos iteradores) são definidas como:

```
typedef tDoubleListImp<tNode> tNodes;
typedef tDoubleListIteratorImp<tNode> tNodeIterator;
typedef tDoubleListImp<tCell> tCells;
typedef tDoubleListIteratorImp<tCell> tCellIterator;
```

Células

Uma célula é um objeto de uma classe derivada da classe abstrata `tCell`. O Programa 2.10 mostra parte da interface de `tCell`.

```
1    class tCell
2    {
3       public:
4          int Number;
5          tMaterial Material;
6
7          tCell(int);
8
9          virtual int GetDimension() const = 0;
10         int GetNumberOfNodes() const;
11         virtual int GetNumberOfEdges() const = 0;
12         virtual int GetNumberOfFaces() const = 0;
13         tNode* GetNode(int) const;
14         virtual tEdge* GetEdge(int) const = 0;
15         virtual tFace* GetFace(int) const = 0;
16
17         void SetNode(int, tNode*);
18         virtual bool Check() const;
19         virtual void Contour(double, double*, tGraphicModel&) const= 0;
20
21         virtual void ComputeLocalSystem(tLocalSystem&) const;
22         virtual t3DVector ComputePositionAt(double []) const;
23         virtual tVector ComputeShapeFunctionsAt(double []) const = 0;
24         virtual tMatrix ComputeShapeFunctionDerivativesAt(double []) const = 0;
25         virtual tJacobianMatrix ComputeJacobianMatrix() const;
26         ...
27   }; // tCell
```

Programa 2.10: Parte da interface da classe `tCell`.

Cada célula de OSW é identificada por um número inteiro único, `Number`, linha 4 (usualmente definido pelo gerador do modelo de decomposição por células). O atributo `Material` (objeto da classe acessória `tMaterial`), declarado na linha 5, identifica o material da célula. O construtor da classe, linha 7, toma como entrada um número inteiro positivo, o número do nó no qual a nova célula incide.

A lista de incidência de uma célula é implementada com um arranjo de ponteiros para objetos da classe `tNode`, cuja dimensão é dada no construtor da classe `tCell`. O método `SetNode(i, node)`, linha 17, ajusta o ponteiro `node` para o `i`-ésimo nó da célula. `GetNode(i)`, linha 13, retorna um ponteiro para o `i`-ésimo nó da célula.

Em adição, uma célula é capaz de:

- Fornecer a sua dimensão topológica. Uma célula com dimensão topológica igual a 2 é uma superfície 3D (por exemplo, um triângulo); uma célula de dimensão topológica igual a 3 é uma região 3D (por exemplo, um tetraedro). Classes derivadas de `tCell` devem sobrecarregar o método virtual puro `GetDimension()`, linha 9, para fornecer a dimensão da célula.

- Verificar a si própria. O método virtual `Check()`, na linha 18, é responsável por verificar a integridade da célula. Essa operação é usualmente requerida como parte inicial do processo de análise numérica, explicado no Capítulo 3.

- Processar seu contornamento. Isso é feito em classes derivadas de `tCell` que sobrecarregam o método virtual puro `Contour()`, linha 19. Em células 2D, por exemplo, o método gera isolinhas correspondentes a determinados valores escalares; em células 3D, o método gera isosuperfícies.

- Calcular suas funções de interpolação. Os valores das funções de forma e de suas derivadas em um ponto interno da célula, vistos na Seção 2.3, são calculados através dos métodos virtuais puros `ComputeShapeFunctionsAt()` e `ComputeShapeFunctionDerivativesAt()`, declarados nas linhas 23 e 24, respectivamente, os quais devem ser sobrecarregados em classes derivadas de `tCell`. Como exemplo, o Programa 2.11 mostra o método `t4N3DCell::ComputeShapeFunctionsAt()`, responsável pelo cálculo das funções de forma para o tetraedro. A classe acessória `t4N3DCell`, derivada de `tCell`, define tetraedros como células de dimensão topológica 3 com 4 nós.

```
1   tVector
2   t4N3DCell::ComputeShapeFunctionsAt(double x[3]) const
3   {
4      tVector shapeF(4);
5
6      shapeF(0) = 1 - x[0] - x[1] - x[2];
7      shapeF(1) = x[0];
8      shapeF(2) = x[1];
9      shapeF(3) = x[2];
10     return shapeF;
11  }
```

Programa 2.11: Método `t4N3DCell::ComputeShapeFunctionsAt()`.

Matrizes Jacobianas e coordenadas globais de um dado ponto `x` são calculados pelos métodos `ComputeJacobianMatrix()`, linha 25, e `ComputePositionAt()`, linha 22, respectivamente. As classes acessórias `tVector` e `tMatrix` representam vetores e matrizes de números reais em OSW.

Nós

Um nó de um modelo de decomposição por células é um objeto da classe `tNode`, derivada da classe base `tVertex`. `tNode` é parcialmente descrito no Programa 2.12.

Um nó mantém seus *graus de liberdade* (DOFs), responsáveis pela manutenção dos componentes de deslocamento (e força de superfície, para nós de contorno), dados como condição de contorno ou determinados como resultado do processo de análise numérica do modelo. Um DOF é um objeto da classe acessória `tDOF`, a ser descrita no Capítulo 3. O construtor de `tNode`, linha 4, toma como entrada um inteiro não negativo, o número de DOFs do nó (um nó pode não ter DOFs, como é o caso de nós internos). O método `DOF(i)`, na linha 8, retorna um ponteiro para o `i`-ésimo DOF no nó. Um nó também deve ser capaz de responder se é um nó de contorno, ou não. O atributo `BoundaryNode`, linha 10, é ajustado com **`true`** se o nó pertence ao contorno do modelo, ou com **`false`**, caso contrário.

Um nó mantém, ainda, uma lista linear duplamente encadeada de ponteiros para células, cada elemento da lista contendo o endereço de uma célula que incide, ou "usa", o nó. Um

```
1    class tNode: public tVertex
2    {
3       public:
4          tNode(int = 0);
5
6          tNodeUse* GetUses() const;
7          int GetNumberOfDOFs() const;
8          tDOF* DOF(int);
9
10         bool BoundaryNode;
11         ...
12   }; // tNode
```

Programa 2.12: Parte da interface da classe `tNode`.

ponteiro para o primeiro elemento dessa lista é obtido através do método `GetUses()`, declarado na linha 6. Um "uso" de nó é representado por um objeto da classe `tNodeUse`, listada no Programa 2.13.

```
1    class tNodeUse
2    {
3       public:
4          tNodeUse(tCell*);
5
6          tCell* GetCell() const
7          {
8             return Cell;
9          }
10
11      private:
12         tCell* Cell;
13         DECLARE_DOUBLE_LIST_ELEMENT(tNodeUse);
14   }; // tNodeUse
```

Programa 2.13: Classe `tNodeUse`.

O construtor da classe, linha 4, toma como argumento um ponteiro para uma célula que "usa" o nó, o qual é mantido no atributo `Cell`, linha 12, e obtido através do método `GetCell()`, linha 6. (O macro `DECLARE_DOUBLE_LIST_ELEMENT(cls)` é utilizado para declarar que a classe `cls` é um elemento de uma lista linear duplamente encadeada. A expansão do macro define atributos privados `Next` e `Prev` e métodos públicos `GetNext()` e `GetPrev()` para obtê-los.) As relações de adjacência mantidas pelas listas de "usos" de nós de um modelo de decomposição por células são fundamentais para a implementação de vários dos filtros abordados no Capítulo 4.

2.6.4 Classes de Modelo Gráfico

Um modelo gráfico é um objeto da classe `tGraphicModel`, um contêiner de primitivos gráficos, tais como pontos, linhas, triângulos e polígonos. Um primitivo gráfico é um objeto de uma classe derivada da classe abstrata `tPrimitive`. Desde que um modelo gráfico *é* um primitivo, `tGraphicModel` é derivada de ambos `tModel` e `tPrimitive`. A Figura 2.10 mostra a hierarquia das classes de OSW utilizadas para a representação de modelos gráficos.

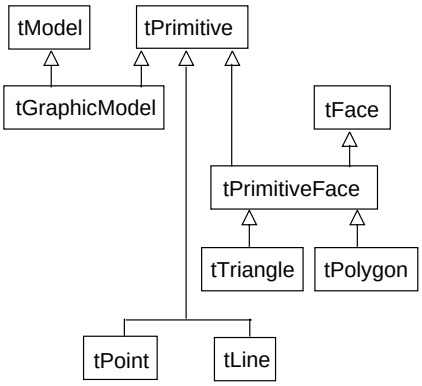

Figura 2.10: Hierarquia de classes de modelos gráficos em OSW.

Em OSW modelos gráficos são usados, por exemplo, para representar isopontos, isolinhas e isosuperfícies gerados por um filtro de contoramento. O Programa 2.14 mostra parte da interface de `tGraphicModel`.

```
1    class tGraphicModel: public tModel, tPrimitive
2    {
3       public:
4          void Transform(const t3DTransfMatrix&);
5          void Contour(double, double*, tGraphicModel&) const;
6          bool Render(tDC&, tRenderer&, tMapper*);
7
8          int GetNumberOfPrimitives() const;
9          tPrimitiveIterator GetPrimitiveIterator();
10         tVertex* GetPrimitiveVertex(const t3DVector&);
11         void InitPointLocator(const tBoundingBox&);
12         void DeletePointLocator();
13         ...
14   }; // tGraphicModel
```

Programa 2.14: Parte da interface da classe `tGraphicModel`.

Um modelo gráfico é capaz de, entre outros, fornecer o número de primitivos que o compõem, método `GetNumberOfPrimitives()`, linha 8, e fornecer o iterador de seus primitivos, método `GetPrimitiveIterator()`, linha 9. (Note, no Programa 2.14, que não há um método público para adição de primitivos à coleção de primitivos de um modelo gráfico. Essa operação é efetuada internamente durante a construção de um primitivo gráfico [22].) A classe `tGraphicModel` sobrecarrega os métodos `Transform()` e `Contour()`, herdados de `tModel`.

Internamente, além da coleção de primitivos, implementada como uma lista linear duplamente encadeada, um modelo gráfico mantém a lista dos vértices utilizados pelos seus primitivos. Sempre que um novo primitivo é construído e adicionado a um modelo gráfico, a lista de vértices do modelo é verificada. Se algum dos vértices do primitivo sendo construído tem coordenadas iguais ou "muito" próximas a algum dos vértices da lista de vértices do modelo, então o vértice já pertencente à lista é utilizado. Caso contrário, o novo vértice é adicionado à lista de vértices do modelo. O método `InitPointLocator(b)`, linha 11, cria

e inicia um objeto da classe acessória `tPointLocator`, responsável pelo gerenciamento eficiente da lista de vértices do modelo gráfico. Utilizando uma técnica de subdivisão espacial baseada em *octree*, um objeto `tPointLocator` verifica um dado ponto p em 3D, se algum ponto coincidente ou "muito" próximo a p existe na lista de vértices do modelo. Com isso, elimina-se a replicação de vértices em um modelo gráfico.

Os modelos gráficos resultantes dos filtros da BEVL são basicamente constituídos de triângulos e polígonos. Um triângulo é um objeto da classe `tTriangle`, Programa 2.15, derivada de `tPrimitiveFace`, a qual, por sua vez, é derivada de `tFace` e `tPrimitive`. `tPrimitiveFace` mantém o número e o material de um triângulo ou polígono.

```
1    class tTriangle: public tPrimitiveFace
2    {
3       public:
4          tTriangle(tGraphicModel&, int, const t3DVector&,
5             const t3DVector&, const t3DVector&);
6          t3DVector Normal() const;
7          void Contour(double, double*, tGraphicModel&) const;
8          tVertex* GetVertex(int i) const;
9          ...
10   }; // tTriangle
```

Programa 2.15: Parte da interface da classe `tTriangle`.

A chamada ao construtor da classe, `tTriangle(g, n, p0, p1, p2)`, linha 4, constrói um novo triângulo, identificado pelo número `n` e definido pelos pontos `p0`, `p1` e `p2`, e o adiciona à coleção de primitivos do modelo gráfico `g`. A classe `tTriangle` sobrecarrega os métodos `Normal()` e `Contour()`, herdados de `tFace` (Programa 2.6) e `tPrimitive` (não mostrada aqui), respectivamente.

Um primitivo do tipo polígono é um objeto da classe `tPolygon`, derivada de `tPrimitiveFace`. A classe mantém, internamente, a lista de vértices que define o polígono. Parte da interface de `tPolygon` pode ser vista no Programa 2.16.

```
1    class tPolygon: public tPrimitiveFace
2    {
3       public:
4          tPolygon(tGraphicModel&, int);
5          t3DVector Normal() const;
6          void Contour(double, double*, tGraphicModel&) const;
7          ::tVertex* mv(const t3DVector&);
8          int GetNumberOfVertices() const;
9          ...
10   }; // tPolygon
```

Programa 2.16: Parte da interface da classe `tPolygon`.

A exemplo de `tTriangle`, a classe sobrecarrega os métodos `Normal()` e `Contour()`, linhas 5 e 6, herdados de `tFace` e `tPrimitive`, respectivamente. O método `mv()`, declarado na linha 7, cria um novo vértice para o polígono, inserindo-o na lista de vértices do polígono.

2.6.5 Classes de Modelo de Casca

Em OSW, um modelo de casca é um objeto da classe `tShell`, cuja interface é parcialmente mostrada no Programa 2.17.

```
1    class tShell: public tMeshableModel
2    {
3       public:
4          class tVertex: public tMeshableModel::tVertex {...};
5          class tEdge {...};
6          class tFace: public ::tFace {...};
7          class tWireframe {...};
8          class tLoop {...};
9          class tVertexUse {...};
10         class tEdgeUse {...};
11
12         void Transform(const t3DTransfMatrix&);
13
14         int GetNumberOfVertices() const;
15         int GetNumberOfEdges() const;
16         int GetNumberOfFaces() const;
17         int GetNumberOfWireframes() const;
18         tVertex* GetVertex(int);
19         tFace* GetFace(int);
20         tWireframe* GetWireframe(int);
21
22         tVertex* mv(int, const t3DVector&);
23         void kv(tVertex*);
24         tEdgeUse* mf(int, tVertex*);
25         void kf(tFace*);
26         tEdgeUse* me(tVertex*, tEdgeUse*);
27         tEdgeUse* mh(tVertex*, tFace*);
28         tEdgeUse* mvf(int, int, const t3DVector&);
29         tEdgeUse* mev(int, const t3DVector&, tEdgeUse*);
30         tEdgeUse* mwf(int, tVertex*);
31         void kwf(tWireframe*);
32         tEdgeUse* mwe(tVertex*, tEdgeUse*);
33         ...
34   }; // tShell
```

Programa 2.17: Parte da interface da classe `tShell`.

`tShell` deriva de `tMeshableModel`, a qual por sua vez, deriva de `tNameableModel`. Isso significa que um modelo de casca pode ser transformado em uma malha e ter um nome. A classe sobrecarrega o método virtual `Transform()`, linha 12, herdado de `tModel`.

Um modelo de casca está apto para:

- Retornar seu número de vértices, arestas e faces. Isso é realizado pelos métodos `GetNumberOfVértices()`, `GetNumberOfEdges()` e `GetNumberOfFaces()`, nas linhas 6, 7 e 8, respectivamente. Como ocorre com modelos de sólidos, as coleções de vértices, arestas e faces de modelos de cascas são implementadas por listas lineares duplamente encadeadas.

- Retornar um vértice ou face específicos. `GetVertex(i)`, linha 10, retorna um ponteiro para o `i`-ésimo vértice da casca. `GetFace(i)`, linha 11, retorna um ponteiro para a `i`-ésima face da casca.

- Realizar operações de inserção e remoção de vértices, arestas e faces. Essas operações são implementadas pelos operadores de modelagem de cascas, listados na Figura 2.8, métodos declarados nas linhas 14 a 24. Por exemplo, `mv()` e `kv()`, linhas 14 e 15, fazem a inserção e remoção de vértices no modelo, respectivamente. Faces são inseridas e removidas por `mf()` e `kf()`, linhas 16 e 17. A classe acessória `tEdgeUse` representa os usos de aresta de um modelo de casca.

2.7 Sumário

Nesse capítulo introduzimos os principais tipos de modelos geométricos de OSW envolvidos ou relacionados com a implementação da BEVL: modelos de sólidos, modelos de decomposição por células, modelos gráficos e modelos de cascas. Discutimos, também, as classes da OCL correspondentes aos diferentes modelos geométricos e seus componentes.

Um modelo de sólido OSW é um objeto da classe `tSolid`. A representação de variedade de dimensão 2 é baseada na estrutura de dados semi-aresta proposta por MÄNTYLA [17]. A classe declara métodos privados que implementam os operadores de Euler. A manipulação de tais métodos é efetuada, usualmente, por objetos que geram modelos de sólidos através de processos simples de varredura translacional, translacional cônica e rotacional.

Um modelo de decomposição por células é um objeto da classe `tMesh`, definido por listas lineares duplamente encadeadas de nós e células. Um nó é um objeto da classe `tNode`, derivada de `tVertex`. Um nó mantém seus graus de liberdade (objetos da classe `tDOF`) e uma lista duplamente encadeada de ponteiros para as células que nele incidem (um "uso de nó é um objeto da classe `tNodeUse`). Uma célula é um objeto de uma classe derivada da classe abstrata `tCell`, a qual declara, entre outros, métodos para computação de funções de forma e contornamento. Modelos de decomposição por células são utilizados no trabalho para representação de malhas de elementos de contorno.

Um modelo gráfico é um objeto da classe `tGraphicModel`, a qual mantém uma lista linear duplamente encadeada de primitivos gráficos tais como pontos, linhas, triângulos, polígonos e outros modelos gráficos. A classe encapsula, ainda, uma lista linear simplesmente encadeada de vértices usados pelos primitivos de um modelo gráfico, manipulada por um objeto da classe `tPointLocator`. Modelos gráficos são usados no trabalho principlamente para representação de isolinhas e isosuperfícies geradas pelos algoritmos de visualização.

Um modelo de casca é um objeto da classe `tShell`. A representação, não *manifold*, é baseada em uma estrutura de dados proposta por PAGLIOSA [22]. A exemplo dos modelos de sólidos, a manipulação dos operadores de modelagem de cascas, embora possível diretamente, é usualmente efetuada por processos de geração de cascas por varredura. Modelos de cascas são usados no trabalho para representação da geometria de ícones orientados empregados para visualização de campos vetoriais.

Embora as representações dos modelos geométricos comentados ao longo do capítulo sejam distintas, todo modelo OSW é capaz de fornecer iteradores internos de componentes topológicos tais como vértices, arestas e faces, objetos de classes derivadas das classes abstratas `tInternalVertexIterator`, `tInternalEdgeIterator` e `tInternalFaceIterator`, respectivamente. Isso define uma interface através da qual outros objetos de OSW, tais como sintetizadores de imagens, possam manipular, indistintamente, qualquer modelo geométrico.

Para a implementação dos algoritmos descritos nos capítulos subseqüentes, as seguintes atividades foram realizadas:

- Revisão das classes de OSW `tCell`, `tPrimitive` e suas classes derivadas, e `tGraphicModel`;
- Criação das classes `t4N3DCell` e `tPointLocator`.

CAPÍTULO 3

Método dos Elementos de Contorno

3.1 Introdução

Neste capítulo, abordaremos de forma resumida o Método dos Elementos de Contorno (MEC), o método de análise numérica empregado no trabalho para análise de modelos de sólidos. A formulação do MEC aplicada ao problema de análise elastostática de sólidos é baseada na discretização do contorno do corpo em elementos de superfície denominados *elementos de contorno*, sobre os quais são aproximados, em termos de funções interpoladoras e parâmetros nodais, deslocamentos e forças de superfície.

Nosso objetivo, nesse capítulo, não é o de apresentar uma descrição detalhada da formulação nem da implementação do MEC. Ao invés disso, iremos contextualizar a formulação do método, apresentando apenas os conceitos necessários ao seu entendimento e a sua utilização para a construção das classes de objetos da BEVL. A exata compreensão dos princípios envolvidos na formulação do MEC requer conhecimentos de algumas áreas de matemática e mecânica que estão fora do escopo desse trabalho. As equações aqui apresentadas podem ser encontradas em bons textos de mecânica [18].

Discutiremos os passos básicos do método dos elementos de contorno, partindo das equações integrais de contorno até a obtenção das equações algébricas que nos permitirão determinar deslocamentos e forças de superfície no contorno e deslocamentos e tensões em pontos discretos no domínio do sólido em questão. Na Seção 3.2, apresentamos os princípios fundamentais e as equações diferenciais do modelo matemático de sólidos elastostáticos. A partir dessas equações diferenciais, a formulação do método dos elementos de contorno é introduzida na Seção 3.3, sendo os passos envolvidos na solução computacional do MEC sumarizados na Seção 3.4. Na Seção 3.5, introduzimos as classes de OSW que implementam a análise numérica de sólidos elastostáticos pelo MEC e discutimos aspectos gerais da implementação do método dos elementos de contorno.

3.2 Princípios Fundamentais

Consideremos um sólido definido por um domínio Ω e um contorno Γ, ao qual são aplicados forças de volume e de superfície. O equilíbrio estático do sólido, escrito em termos de tensões e forças de volume, é dado, em notação indicial, pela seguinte equação diferencial [22]:

$$\sigma_{ji,j} + b_i = 0, \tag{3.1}$$

onde σ_{ji} representa o tensor de tensões e b_i representa a força de volume, por unidade de volume. Se, em adição, o sólido apresenta pequenos deslocamentos e deformações, podemos escrever

$$\epsilon_{ij} = \frac{1}{2}(u_{i,j} + u_{j,i}), \tag{3.2}$$

onde ϵ_{ij} representa o tensor de (pequenas) deformações e u_i é o campo de deslocamentos. Se assumirmos, ainda, que o material do sólido é homogêneo e perfeitamente elástico[1], sua relação constitutiva é expressa pela Lei de Hooke

$$\sigma_{ij} = C_{ijlm}\epsilon_{lm}. \tag{3.3}$$

Para materiais isotrópicos o tensor de módulos elásticos C_{ijlm} é

$$C_{ijlm} = \lambda\delta_{ij}\delta_{lm} + \mu(\delta_{il}\delta_{jm} + \delta_{im}\delta_{jl}), \tag{3.4}$$

onde λ e μ são as constantes de Lame e δ_{ij} é o delta de Kronecker. Substituindo a Equação (3.4) na Equação (3.3), a versão isotrópica da Lei de Hooke fica

$$\sigma_{ij} = \lambda\epsilon_{kk}\delta_{ij} + 2\mu\epsilon_{ij}. \tag{3.5}$$

As constantes de Lame podem ser relacionadas ao módulo de elasticidade transversal E, coeficiente de Poisson ν e módulo de elasticidade transversal G na forma

$$G = \mu = \frac{E}{2(1+\nu)} \qquad \text{e} \qquad \lambda = \frac{\nu\, E}{(1+\nu)(1-\nu)}. \tag{3.6}$$

Considerando a Equação (3.6) e substituindo a Equação (3.2) na Equação (3.5) e a equação resultante na Equação (3.1), obtemos

$$G\ u_{i,jj} + \frac{G}{1-2\nu}\ u_{j,ij} + b_i = 0. \tag{3.7}$$

A Equação (3.7) é conhecida como equação de Navier-Cauchy da elasticidade, e expressa o equilíbrio estático de um sólido em função do campo de deslocamentos u_i e da força de volume b_i aplicadas ao sólido.

A exata compreensão da Equação (3.7) requer conhecimentos sobre cálculo tensorial e mecânica do contínuo que estão além do escopo deste texto. Sem entrarmos em detalhes, enunciaremos que a unicidade da solução da Equação (3.7) deve satisfazer duas espécies de condições de contorno: deslocamentos prescritos (condições de contorno essenciais)

$$u_i = \overline{u}_i \qquad \text{em } \Gamma_1 \tag{3.8}$$

e forças de superfície prescritas (condições de contorno naturais)

$$p_i = \sigma_{ji}n_j = \overline{p}_i \qquad \text{em } \Gamma_2, \tag{3.9}$$

[1] Os conceitos de continuidade, homogeneidade, isotropia e elasticidade são dados em [22].

onde $\Gamma = \Gamma_1 + \Gamma_2$ é a superfície do contorno do sólido e n_j é a normal externa a Γ_2. A Equação (3.1), derivada da teoria da mecânica do contínuo, admitida as hipóteses simplificadoras listadas anteriormente, juntamente com as condições de contorno dadas pela Equação (3.8) e pela Equação (3.9), caracteriza o modelo matemático de um sólido elástico.

Para geometria e condições de contorno quaisquer, a solução do modelo matemático só pode ser obtida com o emprego de métodos numéricos, tais como o MEC e o MEF. Uma solução aproximada para o modelo matemático de sólidos elásticos é determinada, com o MEC, a partir da discretização do contorno Γ do sólido em células especializadas chamadas *elementos de contorno*. O contorno Γ do sólido passa a ser representado, portanto, por um *modelo de decomposição por células*. O comportamento de um elemento de contorno é definido por funções interpoladoras que descrevem as variações dos deslocamentos, deformações e tensões sobre o elemento, a partir dos valores nodais dessas grandezas (estas funções interpoladoras, para algumas classes de elementos, são as próprias funções de forma da célula que define a geometria do elemento de contorno). O comportamento de todo o contorno do sólido é determinado a partir das contribuições individuais de todos os elementos de contorno.

3.3 Formulação do Método dos Elementos de Contorno

A formulação do Método dos Elementos de Contorno, aplicada à análise elastostática de um sólido, pode ser derivada de expressões de *resíduos ponderados* para a equação diferencial que governa o comportamento do sólido, Equação (3.7). A matemática sobre a qual está fundamentada a técnica de resíduos ponderados, bem como a técnica propriamente dita, estão, mais uma vez, além do escopo deste texto, podendo ser encontradas em [22].

O emprego da técnica para o MEC permite que a equação diferencial seja transformada, primeiro, em uma equação integral de domínio. Isso pode ser feito multiplicando-se a Equação (3.1) por um campo válido de deslocamentos arbitrários u^*_{ki} (a função ponderadora u^*_{ki} é chamada *solução fundamental*):

$$(\sigma_{ji,j} + b_i)u^*_{ki} = 0. \tag{3.10}$$

A solução fundamental u^*_{ki} pode ser vista como o campo de deslocamentos sobre um domínio Ω^* com contorno Γ^* — o qual pode ser infinito —, com as mesmas propriedades materiais e que contém o sólido $\Omega+\Gamma$, Figura 3.1. Essa nova região é considerada em estado de equilíbrio quando submetida a forças de volume correspondendo a pontos de carregamentos positivos e unitários aplicados em um ponto $p \in \Omega^*$, em que cada uma das três direções ortogonais é dada pelo vetor unitário $\mathbf{i}_k$. Isso pode ser representado pela equação

$$b^*_{ki} = -\sigma^*_{kji,j} = \delta(p,q)\ \delta_{ki}, \tag{3.11}$$

onde $\delta(p,q)$ é a função delta de Dirac, $q \in \Omega^*$ é um ponto qualquer do domínio e p é o ponto de aplicação da função delta de Dirac, chamado de *ponto fonte*. Para um domínio infinito Ω^*, a solução da Equação (3.7) é a solução fundamental de Kelvin,

$$u^*_{ki} = \frac{1}{16\pi(1-\nu)\ Gr}\left\{(3-4\nu)\ \delta_{ki} + r_{,k}r_{,i}\right\}, \tag{3.12}$$

onde r representa a distância entre os pontos p e q. A expressão para forças de superfíce é:

$$u^*_{ki} = \frac{-1}{8\pi(1-\nu)\ G}\left\{(3-4\nu)\ \ln(r)\ \delta_{ki} - r_{,k}r_{,i}\right\}. \tag{3.13}$$

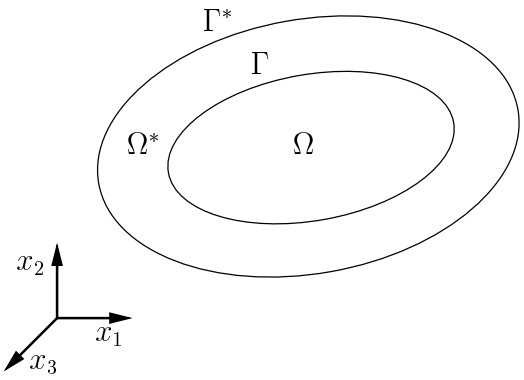

Figura 3.1: Domínio infinito Ω^* contendo $\Omega + \Gamma$.

A equação integral de domínio é obtida, como usual em resíduos ponderados, integrando-se a Equação (3.10) sobre todo o domínio Ω:

$$\int_\Omega \sigma_{ji,j} u^*_{ki} \, \mathrm{d}\Omega + \int_\Omega b_i u^*_{ki} \, \mathrm{d}\Omega = 0. \tag{3.14}$$

A Equação (3.14) pode ser transformada em uma equação integral de contorno como a seguir. Aplicando o teorema da divergência no primeiro termo da equação e substituindo o tensor de tensões, assumido simétrico, por forças de superfície ao longo do contorno, de acordo com a formulação de Cauchy ($p_i = \sigma_{ji} n_j$), obtemos

$$\int_\Gamma p_i u^*_{ki} \, \mathrm{d}\Gamma + \int_\Omega b_i u^*_{ki} \, \mathrm{d}\Omega = \int_\Omega \sigma_{ij} \epsilon^*_{kij} \, \mathrm{d}\Omega. \tag{3.15}$$

Considerando a simetria do tensor módulo elástico C_{ijlm} e aplicando novamente o teorema da divergência no último termo da Equação (3.15), a seguinte equação integral é derivada [4]:

$$\int_\Gamma p_i u^*_{ki} \, \mathrm{d}\Gamma + \int_\Omega b_i u^*_{ki} \, \mathrm{d}\Omega = \int_\Gamma p^*_{ki} u_i \, \mathrm{d}\Gamma - \int_\Omega \sigma^*_{kji,j} u_i \, \mathrm{d}\Omega. \tag{3.16}$$

Substituindo a Equação (3.11) na última integral da Equação (3.16), obtemos a conhecida identidade Somigliana para deslocamentos:

$$u_i(p) + \int_\Gamma p^*_{ki} u_i \, \mathrm{d}\Gamma = \int_\Gamma p_i u^*_{ki} \, \mathrm{d}\Gamma + \int_\Omega b_i u^*_{ki} \, \mathrm{d}\Omega. \tag{3.17}$$

A Equação (3.17) permite a determinação dos deslocamentos em um ponto p de Ω a partir dos valores dos deslocamentos e forças de superfície no contorno Γ e das forças de volume no domínio Ω.

Como é comum em formulações para o MEC, a seguinte equação integral para deslocamentos sobre qualquer ponto p pode ser generalizada da Equação (3.17) [4, 14]:

$$c_{ki}(p) u_i(p) + \int_\Gamma p^*_{ki} u_i \, \mathrm{d}\Gamma = \int_\Gamma p_i u^*_{ki} \, \mathrm{d}\Gamma + \int_\Omega b_i u^*_{ki} \, \mathrm{d}\Omega, \tag{3.18}$$

onde o termo independente $c_{ki}(p)$ é igual a δ_{ki}, quando o carregamento unitário é aplicado em $p \in \Omega$, e depende da geometria do contorno se o ponto fonte $p \in \Gamma$.

O estado de tensões em qualquer ponto $p \in \Omega$ pode ser obtido pela combinação das derivadas da Equação (3.17) considerando as relações deformação-deslocamento e a Lei de Hooke, Equações (3.2) e (3.5), respectivamente. A equação resultante é:

$$\sigma_{ij}(p) = \int_\Gamma D^*_{ijk} p_k \, \mathrm{d}\Gamma - \int_\Gamma S^*_{ijk} u_k \, \mathrm{d}\Gamma + \int_\Omega D^*_{ijk} b_k \, \mathrm{d}\Omega, \tag{3.19}$$

onde, para a solução fundamental de Kelvin, os termos D^*_{ijk} e S^*_{ijk} são dados por:

$$D_{ijk} = -\,\sigma^*_{ijk}, \tag{3.20}$$

$$\begin{aligned} S_{ijk} =& \frac{G}{4\pi(1-\nu)r^3}\Big\{3\frac{\partial r}{\partial n}[(1-2\nu)\delta_{ij}\,r_{,k} + \nu(\delta_{ik}\,r_{,j} + \delta_{jk}\,r_{,i}) - 5r_{,i}r_{,j}r_{,k}] \\ & 3\nu(n_i\,r_{,j}\,r_{,k} + n_j\,r_{,i}\,r_{,k}) + (1-2\nu)(3n_k\,r_i\,r_j + n_j\,\delta_{ik} + n_i\,\delta_{jk}) \\ & -(1-4\nu)n_k\,\delta_{ij}\Big\}. \end{aligned} \tag{3.21}$$

A Equação (3.18) é a equação integral de contorno para o problema de sólidos elastostáticos e o ponto inicial para o método dos elementos de contorno. Essa equação pode ser expressa na forma matricial como

$$\mathbf{c}(p)\mathbf{u}(p) + \int_\Gamma \mathbf{p}^*\mathbf{u}\;\mathrm{d}\Gamma = \int_\Gamma \mathbf{u}^*\mathbf{p}\;\mathrm{d}\Gamma + \int_\Omega \mathbf{u}^*\mathbf{b}\;\mathrm{d}\Omega. \tag{3.22}$$

A adoção de funções aproximadoras para deslocamentos e forças de volume e de superfície, permite que a equação integral de contorno seja transformada em um sistema linear de equações. Para isso, a superfície Γ e o volume Ω do sólido a ser analisado são discretizados em uma série de n_e elementos de contorno e n_c células internas, respectivamente, como ilustrado na Figura 1.2. Agora, a Equação (3.22) pode ser escrita como

$$\mathbf{c}(p)\mathbf{u}(p) + \sum_{e=1}^{n_e}\int_{\Gamma_e} \mathbf{p}^*\mathbf{u}_e\;\mathrm{d}\Gamma = \sum_{e=1}^{n_e}\int_{\Gamma_e} \mathbf{u}^*\mathbf{p}_e\;\mathrm{d}\Gamma + \sum_{c=1}^{n_c}\int_{\Omega_c} \mathbf{u}^*\mathbf{b}_c\;\mathrm{d}\Omega, \tag{3.23}$$

onde $\mathbf{u}_e$ e $\mathbf{p}_e$ são, respectivamente, deslocamento e força de superfície em um ponto em Γ_e e $\mathbf{b}_c$ é a força de volume em um ponto interno a Ω_c.

Sobre cada elemento de contorno Γ_e o campo deslocamento e as forças de superfície são aproximadas por interpolações paramétricas, como segue:

$$\begin{aligned} \mathbf{u}_e = \mathbf{N}_e\mathbf{U}_e = \begin{bmatrix}\mathbf{N}_e^1 & \mathbf{N}_e^2 & \cdots & \mathbf{N}_e^n\end{bmatrix}\begin{bmatrix}\mathbf{u}_e^1\\ \mathbf{u}_e^2\\ \cdots\\ \mathbf{u}_e^n\end{bmatrix} = \sum_{i=1}^{n}\mathbf{N}_e^i\mathbf{u}_e^i, \\ \mathbf{p}_e = \mathbf{N}_e\mathbf{P}_e = \begin{bmatrix}\mathbf{N}_e^1 & \mathbf{N}_e^2 & \cdots & \mathbf{N}_e^n\end{bmatrix}\begin{bmatrix}\mathbf{p}_e^1\\ \mathbf{p}_e^2\\ \cdots\\ \mathbf{p}_e^n\end{bmatrix} = \sum_{i=1}^{n}\mathbf{N}_e^i\mathbf{p}_e^i, \end{aligned} \tag{3.24}$$

onde n é o número de nós do elemento e $\mathbf{u}_e^i$, $\mathbf{p}_e^i$ são, respectivamente, o vetor de componentes de deslocamento e o vetor de componentes de forças de superfície no nó i. Nesse trabalho consideramos que a matriz de função de interpolação $\mathbf{N}_e^i$ associada ao nó i é dada por

$$\mathbf{N}_e^i = N_e^i(\xi,\eta)\;\mathbf{I}, \tag{3.25}$$

onde N_e^i, a função de forma do elemento e para o nó i, é uma função escalar de coordenadas dimensionais ξ e η, tal como visto no Capítulo 2.

De modo similar, a força de volume em um ponto qualquer em cada célula interna pode ser aproximada como

$$\mathbf{b}_c = \mathbf{N}_c\mathbf{B}_c = \begin{bmatrix}\mathbf{N}_c^1 & \mathbf{N}_c^2 & \cdots & \mathbf{N}_c^m\end{bmatrix}\begin{bmatrix}\mathbf{b}_c^1\\ \mathbf{b}_c^2\\ \cdots\\ \mathbf{b}_c^m\end{bmatrix} = \sum_{i=1}^{m}\mathbf{N}_c^i\mathbf{b}_c^i, \tag{3.26}$$

onde m é o número de nós da célula, $\mathbf{b}_c^i$ é a força nodal na célula i e $\mathbf{N}_c^i$ é a matriz de função de forma da célula associada à célula i.

Substituindo as aproximações dadas pelas equações (3.24) e (3.26) na Equação (3.23), obtemos

$$\mathbf{c}(p)\mathbf{u}(p) + \sum_{e=1}^{n_e}\int_{\Gamma_e}\mathbf{p}^*\mathbf{N}_e\ \mathrm{d}\Gamma\,\mathbf{U}_e = \sum_{e=1}^{n_e}\int_{\Gamma_e}\mathbf{u}^*\mathbf{N}_e\ \mathrm{d}\Gamma\,\mathbf{P}_e + \sum_{c=1}^{n_c}\int_{\Omega_c}\mathbf{u}^*\mathbf{N}_c\ \mathrm{d}\Omega\,\mathbf{B}_c, \tag{3.27}$$

a qual pode ser escrita na forma

$$\mathbf{c}(p)\mathbf{u}(p) + \sum_{e=1}^{n_e}\hat{\mathbf{H}}_e\mathbf{U}_e = \sum_{e=1}^{n_e}\mathbf{G}_e\mathbf{P}_e + \sum_{c=1}^{n_c}\mathbf{D}_c\mathbf{B}_c, \tag{3.28}$$

onde

$$\begin{aligned}\hat{\mathbf{H}}_e &= \int_{\Gamma_e}\mathbf{p}^*\mathbf{N}_e\ \mathrm{d}\Gamma,\\ \mathbf{G}_e &= \int_{\Gamma_e}\mathbf{u}^*\mathbf{N}_e\ \mathrm{d}\Gamma,\end{aligned} \tag{3.29}$$

são as chamadas *matrizes de influência* do e-ésimo elemento de contorno para o ponto fonte p, e

$$\mathbf{D}_c = \int_{\Omega_c}\mathbf{u}^*\mathbf{N}_c\ \mathrm{d}\Omega. \tag{3.30}$$

Após adotarmos o número de pontos fonte igual ao número de nós de contorno e apresentarmos todas as integrais espaciais, o seguinte sistema de equações é encontrado [4]:

$$\mathbf{CU} + \hat{\mathbf{H}}\mathbf{U} = \mathbf{GP} + \mathbf{DB}. \tag{3.31}$$

Adicionando as matrizes $\mathbf{C}$ e $\hat{\mathbf{H}}$ obtemos

$$\mathbf{HU} = \mathbf{GP} + \mathbf{DB}. \tag{3.32}$$

(Ao invés de $\hat{\mathbf{H}}_e$, usaremos $\mathbf{H}_e$ como uma das matrizes de influência do e-ésimo elemento.) Agora podemos aplicar as condições de contorno, equações (3.8) e (3.9), na Equação (3.32). O sistema linear pode ser reordenado de tal modo que as incógnitas (deslocamentos e forças de superfície) sejam escritas do lado esquerdo, resultando

$$\mathbf{AX} = \mathbf{F}. \tag{3.33}$$

A solução do sistema expresso pela Equação (3.33) fornece os valores dos deslocamentos e forças de superfície incógnitos sobre o contorno do sólido, a partir dos quais pode-se determinar os valores no domínio. Os valores em qualquer ponto interno podem ser calculados a partir dos valores de contorno obtidos através da integração numérica das equações (3.17) e (3.19). Com isso, serão conhecidos os valores em todos os nós dos elementos de volume, a partir dos quais pode-se determinar, por interpolação, os valores em quaisquer pontos do domínio Ω do sólido, usando as funções de interpolação das células internas.

3.4 Esquema Computacional

A implementação do MEC pode ser sumarizada nos passos a seguir.

Passo 1 **Discretização do sólido**. O contorno Γ é discretizado em uma série de $\mathbf{N}_e$ elementos sobre os quais são interpolados, em termos de funções aproximadoras e valores nodais, deslocamentos e forças de superfície. Esse passo é chamado de pré-processamento. Os elementos de contorno considerados no trabalho são triângulos planares definidos por 3 nós, cujas funções de forma são dadas pela Equação (2.10). Se forças de volume foram aplicadas ao sólido, então o domínio também é discretizado em um conjunto $\mathbf{N}_c$ de células internas. As células internas adotadas serão tetraedros com faces planares, definidos por 4 nós, cujas funções de forma são dadas pela Equação (2.12). Por fim, em função das condições de contorno aplicadas ao sólido, nós múltiplos são criados se, para determinado nó de contorno, pode haver descontinuidade de forças de superfícies nos elementos nele incidentes. Nós múltiplos são nós de contorno com as mesmas coordenadas espaciais, mesmo deslocamento, mas forças de superfície distintas, como exemplificado pelos nós duplos l e m da Figura 3.2.

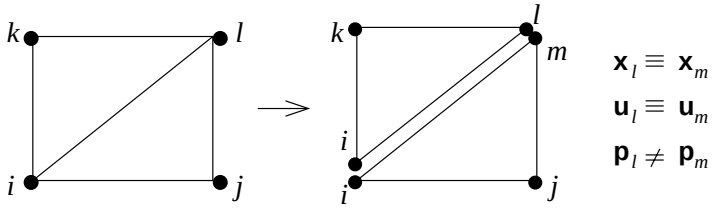

Figura 3.2: Conceito de nó multiplo ilustrado para elementos triangulares.

Passo 2 **Computação das contribuições de contorno**. As matrizes de influência $\mathbf{H}_e$ e $\mathbf{G}_e$ são determinadas para cada elemento e, usualmente, através de esquemas numéricos de quadratura. Note que o emprego da solução fundamental de Kelvin, equações (3.12) e (3.13), introduz singularidades nos núcleos das integrais (3.29) se o ponto fonte pertence ao elemento sendo integrado. Além disso, na prática, pontos fonte não pertencentes a um elemento, mas muito "próximos" deste, resultam em problemas de instabilidade numérica. Para estes casos, técnicas especiais de integração singulares e quase singulares devem ser empregadas [30].

Passo 3 **Montagem do sistema linear**. A partir das contribuições individuais de todos os $\mathbf{N}_e$ elementos de contorno e $\mathbf{N}_c$ células internas (se houverem) do modelo, o sistema linear expresso pela Equação (3.32) é formado.

Passo 4 **Introdução das condições de contorno**. O sistema linear da Equação (3.32), após a introdução das condições de contorno, é reescrito na forma da Equação (3.33).

Passo 5 **Solução do sistema linear**. A solução do sistema linear fornece os deslocamentos e forças de superfície nos pontos nodais e direções onde esses valores não foram prescritos. Conhecidos os valores nodais de deslocamentos e forças de superfície, podemos determinar, por interpolação, os deslocamentos e forças de superfície em qualquer ponto do contorno.

Passo 6 **Computação de valores no domínio**. Uma vez obtidos deslocamentos e forças de superfície no contorno do sólido, pode-se determinar, numericamente, deslocamentos e tensões em pontos *discretos* no domínio Ω do sólido. Os deslocamentos são calculados com a Equação (3.18) e as tensões com a Equação (3.19). Os termos de domínio, em ambas as equações, podem ser computados com auxílio de células internas ou transformação do termo em integral de contorno[2]. Usaremos células internas porque:

- os nós internos para os quais os valores de domínio serão calculados são os próprios nós resultantes da discretização do domínio;
- células internas provêem funções de interpolação necessárias à visualização dos valores de domínio em qualquer ponto interno. Não obstante, isso requerirá a geração de malhas de tetraedros, como será visto no Capítulo 5.

3.5 Elementos de Contorno em OSW

Em OSW, um elemento de contorno é uma célula de um modelo de decomposição por células especializada. A Figura 3.3 mostra parte da hierarquia das classes de OSW utilizadas para a modelagem de elementos de contorno (o símbolo ▲ denota herança virtual).

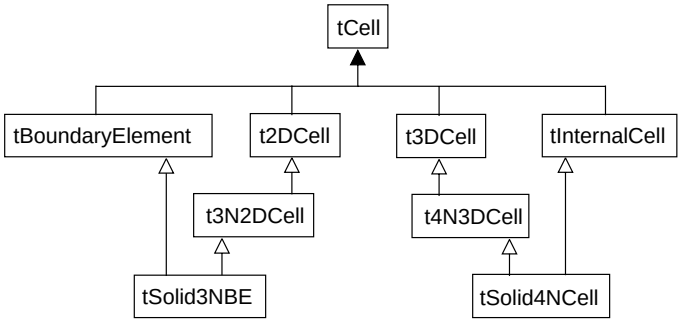

Figura 3.3: Parte das classes de elementos e células em OSW.

A implementação dos elementos de contorno em OSW mostrou ser mais conveniente manter elementos de contorno e células internas em conjuntos distintos, como pode-se observar na Figura 3.3. Para que isso se tornasse possível, foi criada uma nova classe, denominada `tBEMesh`, para gerenciar exclusivamente o conjunto de células internas. Os principais métodos da interface da classe podem ser vistos no Programa 3.1. Note que um objeto da classe `tBEMesh` possui duas coleções de células: uma herdada de `tMesh` e outra de células internas.

3.5.1 Classes de Elementos de Contorno e Células Internas

A Figura 3.3 mostra classes abstratas derivadas virtualmente de `tCell`. As classes `t2DCell` e `t3DCell` representam células 2D e 3D genéricas, respectivamente. Um triângulo é geometricamente definido como uma célula de dimensão topológica 2 com três nós, objeto da classe `t3N2DCell`. Um tetraedro é uma célula de dimensão topológica 3 com quatro nós, objeto da classe `t4N3DCell`.

[2]Na verdade, esse termo de domínio pode ser transformado em uma integral de contorno para certos tipos de forças de volume, o que eliminaria a necessidade de discretização do domínio do sólido [4].

```
1    typedef tDoubleListImp<tInternalCell> tInternallCell;
2    typedef tDoubleListImpIterator<tInternalCell> tInternallCellIterator;
3
4    class tBEMesh: public tMesh
5    {
6       public:
7          int GetNumberOfInternallCells() const;
8          tInternalCell* GetInternalCell(int);
9          tInternalCellIterator GetInternalCellIterator();
10         void AddInternalCell(tInternalCell*);
11         void DeleteInternalCell(tInternalCell*);
12         ...
13   }; // tBEMesh
```

Programa 3.1: Parte da interface da classe `tBEMesh`.

Elementos de Contorno

A nova classe `tBoundaryElement`, parcialmente listada no Programa 3.2, define a estrutura e o comportamento genérico de um elemento de contorno em OSW. O construtor na linha 4 toma como parâmetro o número de nós do elemento, o qual é utilizado para construção da classe base virtual `tCell`.

A principal funcionalidade da classe é dada pelo método virtual `ComputeInfluenceMatrices(p)`, linha 5, responsável por calcular, para um ponto fonte `p`, as matrizes de influência `H` e `G`, declaradas nas linhas 14 e 15, respectivamente. Inicialmente o método determina a posição do ponto `p` em relação ao elemento. (Um ponto fonte é um objeto da classe acessória `tSourcePoint`, não mostrada aqui). Há três possibilidades:

- O ponto fonte está sobre o elemento (incluindo arestas e vértices do contorno do elemento). Nesse caso, o ponto `p` é dito singular, e as matrizes de influência são calculadas pelo método virtual puro `ComputeSingularIntegrals(p)`, linha 19, o qual deve ser sobrecarregado em classes concretas derivadas de `tBoundaryElement`.

- O ponto fonte está fora, porém "muito próximo" ao elemento. As matrizes de influência são calculadas pelo método virtual `ComputeQuasiSingularIntegrals(p)`, linha 21. A implementação do método em `tBoundaryElement` calcula as integrais usando quadratura de Hammer com subdivisão do elemento de acordo com seu "tamanho" e sua "distância" em relação ao ponto fonte `p` [30]. O método virtual `ComputeDistanceFrom(p)`, linha 10, retorna como "distância", a menor das distâncias entre `p` e cada um dos nós do elemento. O "tamanho" considerado é o tamanho médio de todas as arestas do elemento, determinado pelo método virtual `ComputeSize()`, linha 11. Os valores fundamentais $\mathbf{u}^*$ e $\mathbf{p}^*$ devem ser fornecidos por classes derivadas de `tBoundaryElement`, através da sobrecarga dos métodos virtuais puros `ComputeFundamentalU(r)` e `ComputeFundamentalP(r)`, linhas 17 e 18, onde `r` é o vetor de `p` para o ponto `q`.

- O ponto fonte está fora e não "muito próximo do elemento". As matrizes de influência são calculadas pelo método virtual `ComputeNonSingularIntegrals(p)`, linha 20. A implementação é análoga a do caso quase singular, ou seja, baseada em quadratura de Hammer.[3]

[3] A integração numérica é um dos aspectos mais importantes do MEC e um dos mais difíceis de implementar. Uma discussão pormenorizada da implementação em OSW está fora do escopo desse texto.

```
1    class tBoundaryElement: virtual public tCell
2    {
3       public:
4          tBoundaryElement(int);
5          virtual void ComputeInfluenceMatrices(tSourcePoint&);
6          virtual void Init();
7          virtual void Terminate();
8          tMatrix GetHMatrix();
9          tMatrix GetGMatrix();
10         virtual double ComputeDistanceFrom(tSourcePoint&) const;
11         virtual double ComputeSize() const;
12         ...
13      protected:
14         tMatrix H;
15         tMatrix G;
16         ...
17         virtual tMatrix ComputeFundamentalU(const t3DVector&) = 0;
18         virtual tMatrix ComputeFundamentalP(const t3DVector&) = 0;
19         virtual void ComputeSingularIntegrals(tSourcePoint&) = 0;
20         virtual void ComputeNonSingularIntegrals(tSourcePoint&);
21         virtual void ComputeQuasiSingularIntegrals(tSourcePoint&);
22         ...
23   }; // tBoundaryElement
```

Programa 3.2: Parte da interface da classe `tBoundaryElement`.

Um objeto da classe `tSolid3NBE` é um elemento de contorno de dimensão topológica 2 com três nós, para análise de sólidos elastostáticos usando a solução fundamental de Kelvin. A classe deriva de `t3N2DCell` e `tBoundaryElement` e, virtualmente, de `tCell`. O Programa 3.3 mostra a implementação do construtor da classe, o qual toma como parâmetro um vetor `nodes` de três ponteiros `tSolidBENode`, usados para ajustar os nós do elemento.

```
1    class tSolid3NBE: public t3N2DCell, public tBoundaryElement
2    {
3       public:
4          tSolid3NBE (tSolidBENode* nodes[3]):
5             tCell(3), t3N2DCell(3), tBoundaryElement(3)
6          {
7             for (int i = 0; i < 3; i++)
8                SetNode(i,nodes[i]);
9          }
10         ...
11   }; // tSolid3NBE
```

Programa 3.3: Parte da interface da classe `tSolid3NBE`.

A classe `tSolidBENode` é derivada de `tSolidInternalNode`, que por sua vez, é derivada de `tNode`, como podemos ver no Programa 3.4. `tSolidInternalNode` especializa a classe `tNode` pela adição de métodos para obter e ajustar o deslocamento `U` e a força de volume `B` de um nó interno de um sólido, linhas 4 a 7. `tSolidBENode`, em adição, declara nas linhas 15 e 16 métodos para obter e ajustar a tração `P` de um nó do contorno (um nó de contorno é um nó interno que foi "jogado" para o contorno do sólido).

```
1    class tSolidInternalNode: public tNode
2    {
3       public:
4          double U(int) const;
5          double U(int);
6          double B(int) const;
7          double B(int);
8          ...
9    }; // tSolidInternalNode
10
11   class tSolidBENode: public tSolidInternalNode
12   {
13      public:
14         tSolidBENode(int);
15         double P(int) const;
16         double& P(int);
17         ...
18   }; // tSolidBENode
```

Programa 3.4: Parte da interface da classe `tSolidBENode`.

Células Internas

Uma célula interna é um objeto de uma classe derivada da classe abstrata `tInternalCell`, parcialmente listada no Programa 3.5. Uma célula interna é uma célula especializada capaz de calcular seu termo de domínio `D` em um dado ponto fonte `p`, declarado na linha 11. Isso é feito através do método virtual `ComputeDomainTerm(p)`, linha 5. Classes concretas derivadas de `tInternalCell` devem sobrecarregar o método virtual puro `ComputeFundamentalU(r)`, linha 13. A nova classe `tSolid4NCell`, Figura 3.3, sobrecarrega o método `ComputeDomainTerm(p)`, linha 5, para avaliar a Equação (3.30). A classe deriva de `t4N3DCell` e `tInternalCell`.

```
1    class tInternalCell: virtual public tCell
2    {
3       public:
4          tInternalCells(int);
5          virtual void ComputeDomainTerm(tSourcePoint&);
6          virtual void Init();
7          virtual void Terminate();
8          tMatrix GetDMatrix();
9          ...
10      protected:
11         tMatrix D;
12         ...
13         virtual tMatrix ComputeFundamentalU(const t3DVector&) = 0;
14         ...
15   }; // tInternalCell
```

Programa 3.5: Parte da interface da classe `tInternalCell`.

3.5.2 Classes de Analisadores

Um processo de análise numérica em OSW é um objeto de uma classe derivada da classe abstrata `tSolver`, cuja interface principal pode ser vista no Programa 3.6.

```
1    class tSolver: public tErrorHandler, tThread
2    {
3       public:
4          tSolver(tMesh&);
5          ...
6       protected:
7          tMesh* Mesh;
8          tLinearSystem* LS;
9          ...
10         virtual void Init();
11         virtual void Check();
12         virtual tLinearSystem* CreateLinearSystem();
13         virtual void AssembleSystem() = 0;
14         virtual void Terminate();
15         virtual void Resume(tXSolver&);
16         virtual bool CheckCell(tCell*);
17         ...
18      private:
19         void Run();
20         ...
21   }; // tSolver
```

Programa 3.6: Parte da interface da classe `tSolver`.

A classe `tSolver` deriva das classes acessórias `tErrorHandler` e `tThread`. O construtor da classe, linha 4, toma como parâmetro uma referência a uma malha, o modelo de análise de entrada do analisador. `tSolver` possui métodos para:

- Gerenciar um ponteiro para o modelo de análise de entrada e um ponteiro para o sistema linear mantido pelo analisador. Esses ponteiros são representados, respectivamente, pelos atributos `Mesh` e `LS`, declarados nas linhas 7 e 8.
- Iniciar o processo de análise. O processo é iniciado através do método `Init()`, linha 10.
- Verificar o modelo de análise da entrada. O método `Check()`, linha 11, realiza essa verificação.
- Criar e montar o sistema linear do analisador. A criação do sistema linear é realizada pelo método `CreateLinearSystem()`, linha 12. O método `AssembleSystem()`, linha 13, é responsável pela montagem do sistema linear.
- Terminar o processo de análise. O processo de análise finaliza com a chamada ao método `Terminate()`, declarado na linha 14.

O processo de análise inicia quando o objeto analisador recebe a mensagem `Start()`. O método correspondente (declarado em `tThread`) chama o método virtual `Run()` (derivado de `tThread`), na linha 19. A implementação do método pode ser vista no Programa 3.7 e será comentada a seguir.

```
1   void tSolver::Run()
2   {
3      try
4      {
5         Init();
6         AssembleSystem();
7         LS->Solve();
8         Terminate();
9      }
10     catch (tXSolver& x)
11     {
12        Resume(x);
13     }
14  }
```

Programa 3.7: Programa que mostra a execução de tSolver.

A funcionalidade de um analisador MEC genérico de OSW é definida na nova classe abstrata `tBESolver`, derivada de `tSolver` e cuja interface parcial pode ser vista no Programa 3.8. Um analisador MEC específico para o problema de sólidos elastostáticos é um objeto da classe `tSolidBESolver`, a qual não será descrita aqui. Maiores detalhes podem ser vistos em [22].

```
1   class tBESolver: public tSolver
2   {
3      public:
4         tBESolver(tBEMesh&);
5         virtual int GetNumberOfDOFsPerNode() const = 0;
6         ...
7      protected:
8         void Check();
9         void AssembleSystem();
10        void Terminate();
11        virtual bool CheckInternalCell(tInternalCell*);
12        virtual tSourcePointIterator GetSourcePointIterator();
13        virtual void Assemble(tSourcePoint&, tBoundaryElement&);
14        virtual void Assemble(tSourcePoint&, tInternalCell&);
15        virtual void ComputeBoundaryValues();
16        virtual void ComputeDomainValues();
17        ...
18  }; // tBESolver
```

Programa 3.8: Parte da interface da classe `tBESolver`.

3.5.3 Fases do Processo de Análise

Nesta seção, apresentamos uma descrição da implementação dos principais passos do processo de análise numérica pelo MEC.

Iniciando a Análise

No Programa 3.7, `tSolver::Run()` envia a mensagem `Init()` para o analisador começar o processo de análise. Em resposta, o método `tSolver::Init()` executa o método `Check()`. O método `tSolver::Check()` é declarado como virtual no Programa 3.6 e sobrecarregado na classe derivada `tBESolver`. `tBESolver::Check()`, declarado no Programa 3.8, linha 8, é responsável por verificar a integridade do modelo de análise da entrada. O método:

- Verifica o contorno pela chamada ao método herdado `tSolver::Check()`, o qual "atravessa" a coleção de elementos do modelo de análise e, para cada elemento de contorno apontado por `e`, envia ao analisador a mensagem `CheckCell(e)`. O método `CheckCell(e)` é sobrecarregado na classe derivada `tSolidBESolver`. `tSolidBESolver::CheckCell(e)`, verifica se `*e` é uma instância da classe `tSolid3NBE` e, depois, envia a mensagem `Check()` para `*e`, delegando ao elemento a responsabilidade de verificar a si mesmo. Se o elemento responde `true` a análise prossegue. Caso contrário, `tSolver::Check()` gera uma exceção da classe acessória `tXSolver`. O método `tSolver::Run()` captura a exceção e a trata com uma chamada ao método virtual `Resume()`. `tSolver::Resume()` mostra a mensagem de erro gerada pela exceção e aborta a análise.

- Verifica o domínio, "atravessando" a coleção de células internas do modelo de entrada e, para cada célula apontada por `c` envia a mensagem `CheckInternalCell(c)` para o analisador. O método é declarado como virtual em `tBESolver` (Programa 3.8) e sobrecarregado na classe derivada `tSolidBESolver`. O método `tSolidBESolver::CheckInternalCell(c)` verifica se `*c` é uma instância de `tSolidInternalCell` e, depois, envia a mensagem `Check()` para `*c`. Em caso de erro, uma exceção da classe `tXSolver` é gerada.

Após o modelo de análise ter sido verificado, `tSolver::Init()` envia a mensagem `CreateLinearSystem()` para o analisador. Em resposta, o analisador executa o método correspondente, linha 12 do Programa 3.6, o qual cria uma instância da classe acessória `tFullSystem`, derivada de `tLinearSystem`. Um ponteiro para o objeto é salvo no atributo `LS` da classe `tSolver`.

Montando o Sistema Linear

No Programa 3.7, `tSolver::Run()` envia a mensagem `AssembleSystem()` para o analisador. Em resposta, este executa o método `tBESolver::AssembleSystem()`, declarado na linha 9 do Programa 3.8, cuja implementação pode ser vista no Programa 3.9.

O método `tBESolver::AssembleSystem()`:

- Envia, na linha 4, a mensagem `GetSourcePointIterator()` para o analisador, para obter o iterador de pontos fonte `pit`. O método é declarado como virtual, na classe `tBESolver`. Um objeto `tSourcePointIterator` [22], associa um ponto fonte a cada nó do contorno do modelo de análise da entrada.

- Monta, para cada ponto fonte `*p`, declarado na linha 7, as linhas correspondentes ao sistema linear. Isso é feito através da obtenção das contribuições de todos os elementos de contorno e de todas as células internas do modelo de análise. Em um primeiro passo, linhas 8 a 14, o método envia a mensagem `GetCellIterator()` para a malha, para obter um iterador de elementos de contorno. Então, para cada elemento de contorno `*e`, o método envia a mensagem `ComputeInfluenceMatrices(*p)`. A seguir, o método

```
1    void tBESolver::AssembleSystem()
2    {
3       tBEMesh* mesh = (tBEMesh*)Mesh;
4       tSourcePointIterator pit = GetSourcePointIterator();
5       while (pit)
6       {
7          tSourcePoint* p = pit++;
8          for (tCellIterator cit = mesh->GetCellIterator(); cit;)
9          {
10            tBoundaryElement* e = (tBoundaryElement*)cit++;
11            e->ComputeInfluenceMatrices(*p);
12            Assemble(*p,*e);
13            e->Terminate();
14         }
15         tInternalCellIterator cit = mesh->GetInternalCellIterator();
16         while (cit)
17         {
18            tInternalCell* c = cit++;
19            c->ComputeDomainTerm(*p);
20            Assemble(*p,*c);
21            c->Terminate();
22         }
23      }
24   }
```

Programa 3.9: Montando o sistema linear.

envia para o analisador a mensagem `Assemble(*p,*e)`, para inserir no sistema linear as matrizes de influência do elemento `*e`. Na linha 13, o método envia a mensagem `Terminate()` para o elemento `*e`. Em resposta, o elemento libera suas matrizes de influência. Em um segundo passo, da linha 15 a 22, operações similares são realizadas para as células internas do modelo de análise. A mensagem `Assemble(*p,*c)` armazena no sistema linear do analisador o termo de domínio `*c` da célula interna. `tBESolver::Assemble(*p,*e)`, que adiciona em `*LS` as contribuições das matrizes de influência de `*e` devido ao ponto fonte `*p`, pode ser vista no Programa 3.10.

Resolvendo o Sistema Linear e Finalizando a Análise

No Programa 3.7, `tSolver::Run()` envia para o sistema linear apontado por `LS` a mensagem `Solve()`. Em resposta, o sistema linear resolve a si próprio. Se o sistema é singular, o método gera uma exceção da classe `tXSolver`. Finalizando a análise, `tSolver::Run()` envia a mensagem `Terminate()` para o analisador. No Programa 3.8, `tBESolver::Terminate()` termina a análise enviando as mensagens `ComputeBoundaryValues()` e `ComputeDomainValues()` para o analisador. Esse, em resposta, executa os métodos:

- `tSolidBESolver::ComputeBoundaryValues()`, responsável por mover, para cada nó do contorno do modelo de análise, os elementos do vetor solução do sistema linear e valores prescritos de DOF´s para o atributo correspondente (deslocamento ou tração).
- `tSolidBESolver::ComputeDomainValues()`, o qual calcula, para cada nó interno do modelo de análise, o deslocamento e tensões dados pela Equação (3.17) e pela Equação (3.19), respectivamente.

```
1    void tBESolver::Assemble(tSourcePoint& p, tBoundaryElement& e)
2    {
3       tMatrix A = ((tFullSystem*)LS)->GetMatrix();
4       tVector B = ((tFullSystem*)LS)->GetVector();
5       tMatrix H = be.GetHMatrix();
6       tMatrix G = be.GetGMatrix();
7       int ndofs = GetNumberOfDOFsPerNode();
8       int n = e.GetNumberOfNodes();
9       for (int i = 0,r = p.GetNumber()*ndofs; i < ndofs; i++,r++)
10      {
11         for (int k = 0; k < n; k++)
12         {
13            tNode* node = e.GetNode(k);
14            int c = node->GetNumber()*ndofs;
15            for (int kd = k*ndofs,j = 0; j < ndofs; j++,kd++,c++)
16            {
17               tDOF* dof = node->GetDOF(j);
18               if (dof->IsFixed())
19               {
20                  A(r,c) += H(i,kd);
21                  B(r) += G(i,kd)*dof->GetPrescribedValue();
22               }
23               else
24               {
25                  A(r,c) -= G(i,kd);
26                  B(r) -= H(i,kd)*dof->GetPrescribedValue();
27               }
28            }
29         }
30      }
31   }
```

Programa 3.10: Montando condições elementares.

3.6 Sumário

Nesse capítulo, apresentamos o desenvolvimento da formulação para o MEC a partir de representações integrais das equações diferenciais do modelo matemático de sólidos elastostáticos. Consideramos um sólido de domínio Ω e contorno Γ, escrevendo as equações integrais de contorno para deslocamentos de pontos localizados no seu interior e sobre o seu contorno. Dividindo o contorno em um modelo de elementos de contorno, as equações integrais foram representadas por somas de integrais sobre as superfícies de todos os elementos de contorno. O resultado foi um sistema linear obtido pela aplicação das equações integrais discretizadas em N pontos fonte distintos, localizados no exterior ou no contorno Γ, onde N representa o número de nós do modelo de análise. A solução do modelo fornece os deslocamentos e forças de superfície incógnitos, a partir dos quais podem ser determinados os valores de domínio.

As classes de OSW que implementam a análise numérica de sólidos elastostáticos pelo MEC foram introduzidas e seus aspectos gerais discutidos. O propósito não foi descrever com detalhes a implementação, nem tão pouco a formulação do MEC, mas apresentar conceitos e classes cujas nomenclaturas serão utilizadas nos resultados dos algoritmos de visualização da BEVL.

Nessa fase de estudo do MEC, nossa principal atividade foi a revisão de todos os componentes OSW que implementam o MEC, para que pudessem ser utilizados com sucesso no teste das classes da BEVL.

CAPÍTULO 4

Algoritmos de Visualização

4.1 Introdução

No Capítulo 1, vimos que a visualização trata de transformação e representação de dados. Os algoritmos que transformam dados são o coração da visualização de dados. Para descrever as várias transformações possíveis, é preciso categorizar os algoritmos de acordo com a *estrutura* e o *tipo* de transformação. Estrutura é definida como sendo os efeitos que a transformação tem na topologia e geometria de um modelo genérico. O tipo de transformação faz referência ao tipo de modelo sobre o qual o algoritmo opera.

Transformações estruturais podem ser classificadas de quatro maneiras [28], dependendo dos efeitos na geometria, topologia e atributos do modelo utilizado.

- *Transformações geométricas* alteram a geometria do modelo, mas não mudam sua topologia. Por exemplo, a aplicação de operações de translação, rotação e/ou escala em um modelo, equivale a aplicação de uma transformação geométrica no modelo.
- *Transformações topológicas* alteram a topologia do modelo, mas não alteram sua geometria. Por exemplo, a conversão de um modelo de sólido em um modelo de decomposição por células, altera a topologia, pois as informações sobre a conectividade de vértices, arestas e faces são distintas nessas representações. Algumas vezes, no entanto, transformações na topologia também implicam na transformação da geometria.
- *Transformações de atributos* convertem dados de atributos de uma forma para outra, ou criam novos atributos a partir dos atributos do modelo, sem modificar sua estrutura.
- *Combinações de transformações* alteram, conjuntamente, a geometria, a topologia e os atributos de um modelo.

Os algoritmos também podem ser classificados de acordo com o tipo de dado sobre os quais operam, ou de acordo com o tipo de dado que geram.

- *Algoritmos escalares* operam sobre dados escalares. Por exemplo, a geração de mapa de cores de um campo escalar correspondente à pressão aplicada em uma determinada região da superfície de um modelo.
- *Algoritmos vetoriais* operam sobre dados vetoriais. Por exemplo, ícones orientados para mostrar deslocamentos e forças de superfície do contorno de um sólido.

- *Algoritmos tensoriais* operam sobre dados tensoriais. Neste trabalho, não abordaremos algoritmos tensoriais.

- *Algoritmos de modelagem* geram conjuntos de dados topológicos ou geométricos, ou superfícies normais, ou dados de textura. Algoritmos de modelagem têm a tendência de ser uma categoria que agrupa algoritmos que não se enquadram em nenhuma das categorias mencionadas anteriormente. Por exemplo, uma combinação de um algoritmo escalar/vetorial.

Neste capítulo, discutiremos os algoritmos de transformações de dados em imagens de computador. Na Seção 4.2 apresentamos os algoritmos empregados na visualização de campos escalares. Os algoritmos para visualização de campos vetoriais são descritos na Seção 4.3. Na Seção 4.4, conceituamos a geração de superfícies de corte, utilizadas para visualização de dados escalares e vetoriais de domínio de um modelo de decomposição por células. A implementação dos algoritmos de visualização de OSW, bem como as classes e objetos envolvidos no processo de visualização são descritos na Seção 4.5.

4.2 Algoritmos Escalares

A visualização de escalares é a forma mais simples e comum de visualização de dados, existindo para isso diferentes algoritmos [10, 28] e aplicações [1, 20]. Nesta seção, serão considerados dois algoritmos de visualização de escalares: mapa de cores e contornamento (*contouring*).

4.2.1 Mapas de Cores

O mapeamento de cores é uma técnica comum de visualização de escalares que consiste na associação de cores a dados escalares de um modelo. Neste trabalho consideramos que atributos escalares de um modelo são armazenados nos vértices do modelo. Uma vez efetuado o mapeamento, portanto, o mapa de cores é visualizado como resultado direto da síntese da imagem do modelo com tonalização de Gouraud [24]. As cores dos pixels da imagem correspondentes aos pontos visíveis de uma face do modelo são determinadas por interpolação bilinear das cores dos vértices da face.

O mapeamento escalar é implementado utilizando-se uma *tabela de cores* com n entradas. Cada entrada da tabela de cores contém os componentes de uma cor, dado em algum modelo de cores, usualmente RGB ou HSV [7]. Associado com a tabela de cores, há uma faixa de valores escalares definida por um valor mínimo s_{min} e um valor máximo s_{max}. A cor associada a um escalar s é a cor da i-ésima entrada da tabela de cores, onde

$$i = \begin{cases} 0 & \text{se } s < s_{\min}, \\ n-1 & \text{se } s > s_{\max}, \\ \dfrac{s - s_{\min}}{s_{\max} - s_{\min}} & \text{se } s_{\min} \leq s \leq s_{\max}. \end{cases} \tag{4.1}$$

O mapa de cores mostrado na Figura 1.7 utiliza uma tabela de cores variando do vermelho ao azul, com o valor mínimo de escalar igual a $-6,8175^{-4}$ m, e valor máximo de escalar igual a $+6,8209^{-4}$ m.

4.2.2 Contornamento

Uma extensão natural do mapa de cores é o contornamento (*contouring*). Quando vemos uma superfície colorida, podemos visualmente separar áreas com cores similares em regiões

distintas, como se estivéssemos construindo o contorno entre essas regiões. Esse contorno corresponde a linhas (2D) ou superfícies (3D) de valor escalar constante. Contornos tridimensionais são chamados *isosuperfícies* e podem ser aproximados geometricamente por polígonos planares. Contornos bidimensionais são chamados *isolinhas*, as quais podem ser aproximadas por linhas retas. Um exemplo de isolinhas são os contornos que marcam níveis de pressão aplicados a uma superfície. Um exemplo de isosuperfícies são superfícies contendo pontos do domínio de um sólido sob mesma tensão.

O algoritmo "marchando" em cubos (*marching cubes*) desenvolvido por LORENSEN e CLIME em 1987 [15], é uma técnica simples e elegante para a criação de isosuperfícies. Para explicar essa técnica, será introduzido o algoritmo "marchando" em quadrados (*marching squares*), uma técnica bidimensional análoga.

Considere a malha 2D esquematizada na Figura 4.1. Os valores escalares dos vértices são mostrados próximos aos nós que definem a malha. O processo de geração de isolinhas começa pela seleção de um escalar, ou valor de "contorno", que corresponde ao valor das isolinhas geradas. Para gerar os contornos, utiliza-se uma função de interpolação linear sobre as arestas dos quadrados. Se uma aresta é formada pelos valores escalares 0 e 10, por exemplo, e tentarmos gerar uma linha de contorno de valor 5, a interpolação calcula que o contorno passa através do ponto médio da aresta. Uma vez gerados os pontos das isolinhas sobre as arestas em todos os quadrados, esses pontos são conectados para formar isolinhas. A ligação desses pontos pode ser feita usando-se uma técnica 2D de divisão e conquista, que trata cada célula do modelo separadamente, chamada "marchando" em quadrados (*marching squares*).

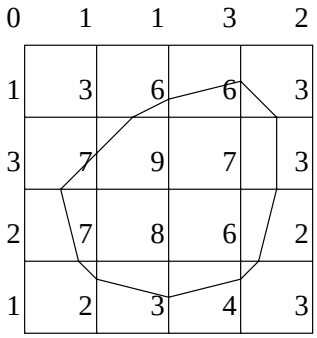

Figura 4.1: Isolinha em uma malha estruturada 2D.

A idéia da técnica "marchando" em quadrados é que uma isolinha pode passar através de uma célula por um número finito de maneiras. Uma *tabela de casos* é construída para enumerar todos os *estados* topológicos possíveis de um quadrado, dadas as combinações dos valores escalares nos vértices do quadrado. O número de estados topológicos depende do número de vértices da célula e do número de relações dentro/fora que um vértice pode ter em relação ao valor da isolinha. Um vértice é considerado dentro de uma isolinha se seu valor escalar é maior que o valor escalar da isolinha. Vértices com valor escalar menor que o valor da isolinha são ditos fora da isolinha. Por exemplo, se a célula tem 4 vértices e cada vértice pode estar dentro ou fora da isolinha, há $2^4 = 16$ maneiras possíveis das isolinhas passarem pela célula, representadas com 4 *bits* de índice para a tabela de casos.

O algoritmo pode ser descrito pelos passos a seguir.

Passo 1 Selecione uma célula.

Passo 2 Calcule o estado dentro/fora de cada vértice da célula.

Passo 3 Crie um índice armazenando o estado binário (0 ou 1) de cada vértice em um *bit* separado.

Passo 4 Use o índice para localizar o estado topológico da célula na tabela de casos.

Passo 5 Calcule o ponto de interseção da isolinha para cada aresta na tabela de casos.

A Figura 4.2 ilustra as 16 combinações possíveis para um quadrado.

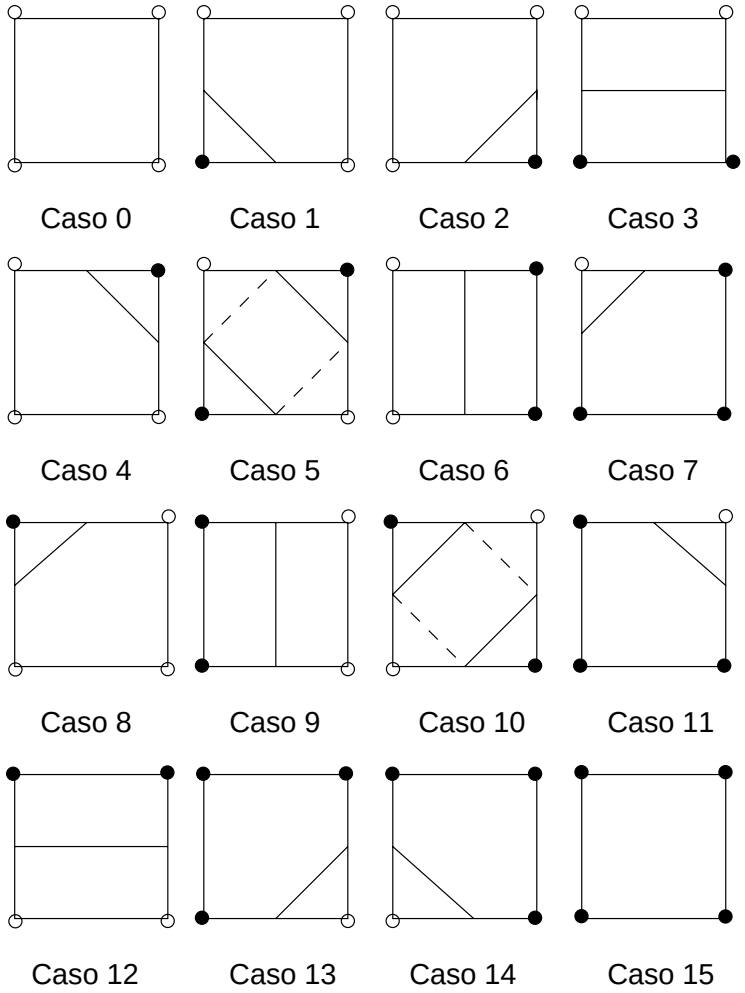

Figura 4.2: Casos do algoritmo "marchando" em quadrados.

O algoritmo "marchando" em quadrados pode ser generalizado para qualquer tipo de célula. O algoritmo "marchando" em cubos é a generalização do algoritmo para células cúbicas e o algoritmo "marchando" em tetraedros é a generalização do algoritmo para células tetraédricas. A Figura 4.3 mostra as 16 combinações possíveis para células tetraédricas.

A Figura 4.4 mostra, além do mapa de cores, as isolinhas correspondentes ao deslocamento na direção y, obtidos da análise da esfera da Figura 1.3. Note que as isolinhas são o "contorno" do mapa de cores.

4.3 Algoritmos Vetoriais

No Capítulo 2, vimos que dados vetoriais representam grandezas com direção e magnitude resultantes, por exemplo, do estudo de fluxo de fluídos [9, 13], ou de campos de deslocamentos

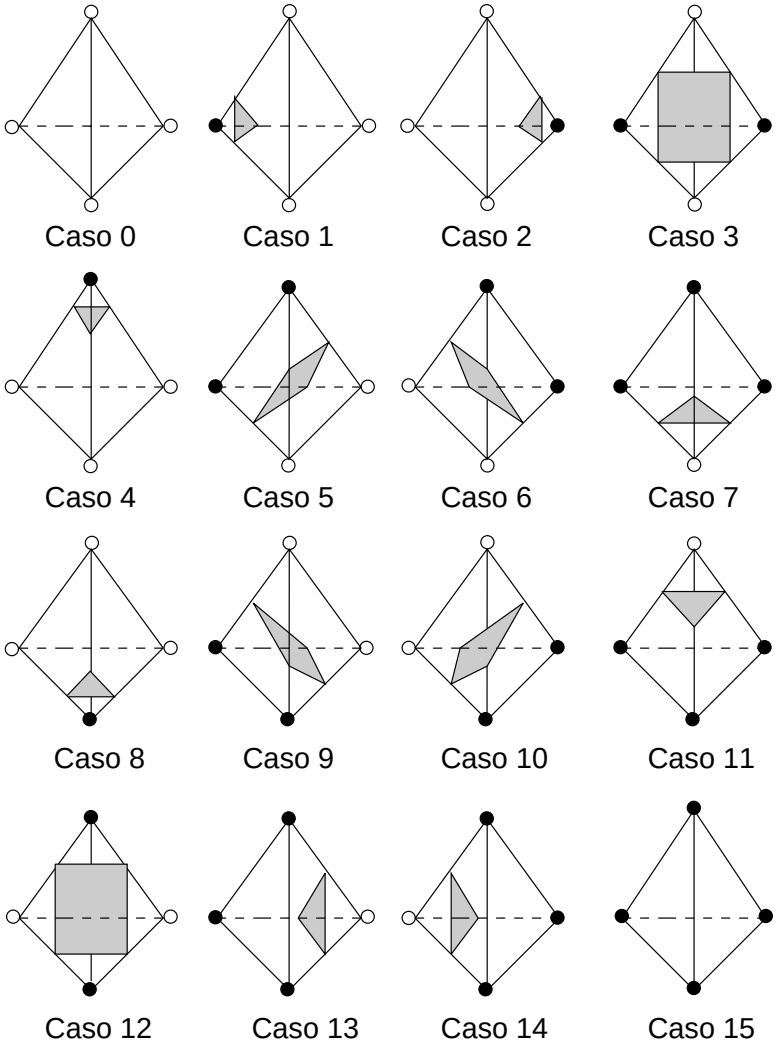

Figura 4.3: Casos do algoritmo "marchando" em tetraedros.

e forças de superfície. Nessa seção, abordaremos os algoritmos de deformação (*warping*), utilizado para visualização da estrutura deformada de um modelo, e ícones orientados (*glyphs*), utilizados para visualização da direção de deslocamentos e forças de superfície de um modelo.

4.3.1 Deformação

Os dados vetoriais mais interessantes para os propósitos desse trabalho são os deslocamentos de um sólido. Uma técnica eficiente para mostrar tais dados vetoriais é "deformar" a geometria do sólido, de acordo com o campo vetorial. Essa técnica é chamada deformação (*warping*).

Considere, então, um modelo de decomposição por células cujos nós contêm o deslocamento resultante da análise numérica pelo MEC (os componentes do deslocamento de um nó são armazenados nos graus de liberdade do nó). Primeiramente o algoritmo "atravessa" a coleção de vértices do modelo de decomposição por células e extrai, de cada nó o vetor de interesse, nesse caso o deslocamento $\mathbf{u}$ do nó. Em seguida, o algoritmo aplica uma transformação de escala no campo vetorial, cujo objetivo é controlar a distorção geométrica do modelo. Deformações muito pequenas podem não ser visíveis e deformações muito grandes

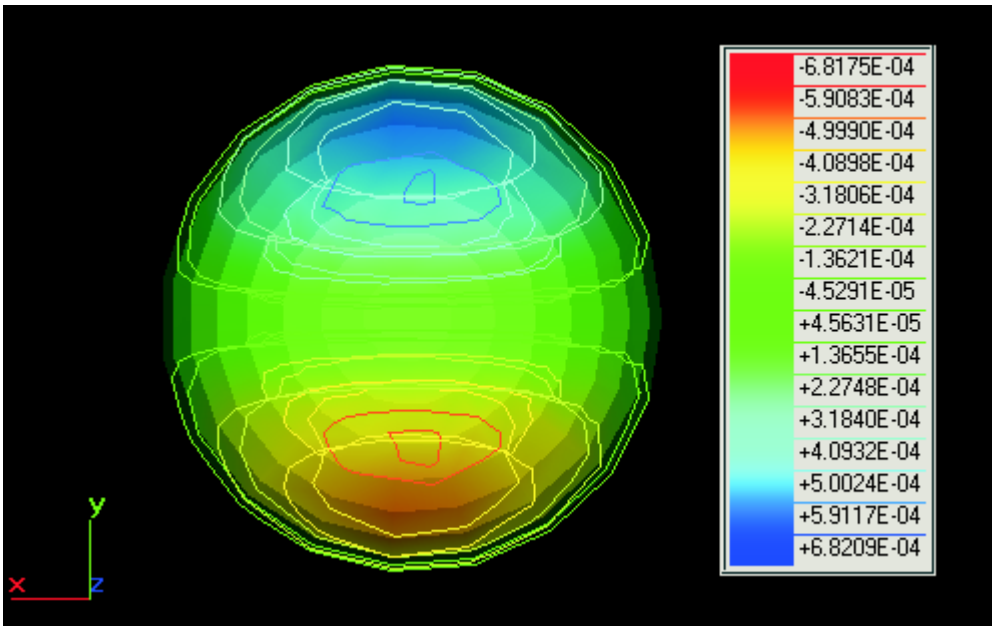

Figura 4.4: Exemplo de isolinhas e mapa de cores para o modelo da Figura 1.3.

podem fazer a estrutura "virar de dentro para fora". As coordenadas de um vértice da estrutura deformada são obtidas pela adição do vetor **u** do vértice com as coordenadas originais do vértice. Finalmente, a estrutura deformada é visualizada, como no exemplo da Figura 4.5. À esquerda vemos a malha original e à direita a malha com escala de deformação igual a 200.

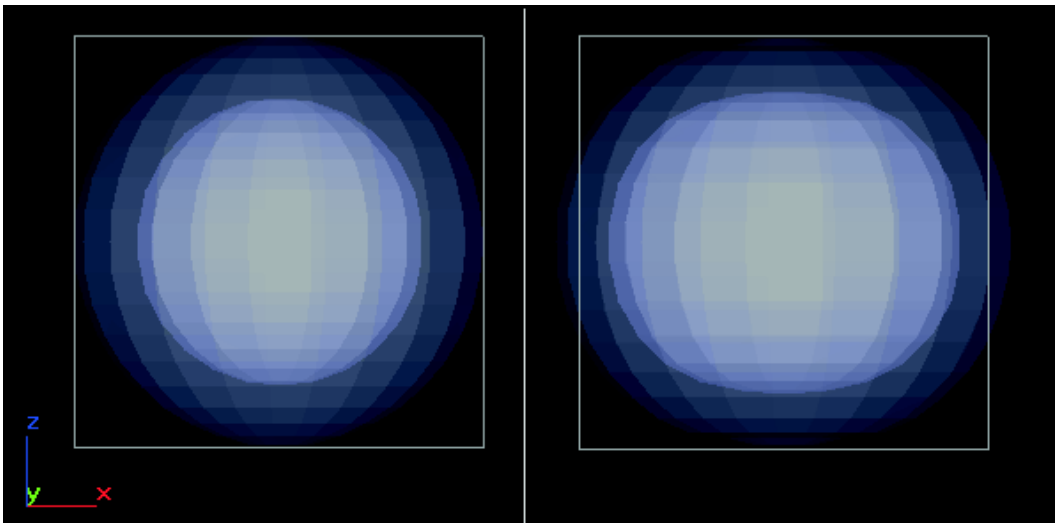

Figura 4.5: Estrutura deformada do modelo da Figura 1.3.

4.3.2 Ícones Orientados

Ícones (*glyphs*) [6] representam uma técnica versátil para visualização de dados de todos os tipos. Um ícone é um componente de um sistema de visualização, definido por um conjunto

de dados ou uma imagem, que pode ser parametrizado por vários dados de entrada. Um ícone pode orientar, aplicar escala, transladar, deformar ou alterar de qualquer forma sua aparência, em resposta a seus dados de entrada.

De acordo com SCHROEDER [28], ícones representam o resultado fundamental do processo de visualização. Além disso, todas as técnicas de visualização apresentadas até agora podem ser tratadas como representações concretas de uma classe abstrada de ícone. Por exemplo, isosuperfícies podem ser consideradas, topologicamente, um ícone bidimensional para dados escalares. Em seu trabalho, DELMARCELLE e HESSELINK [6] classificam ícones de acordo com uma das três categorias:

- *Ícones elementares* representam seus dados através da extensão de seu domínio espacial. Por exemplo, uma seta orientada pode ser usada para representar a normal a uma superfície.
- *Ícones locais* representam informações elementares em adição a uma distribuição dos valores em torno do domínio espacial. Um vetor normal à superfície colorido por curvas locais é um exemplo de um ícone local.
- *Ícones globais* mostram a estrutura de um conjunto de dados completo. Uma isosuperfície é um exemplo de um ícone global.

Esse esquema de classificação pode ser estendido para outras técnicas de visualização, tais como algoritmos vetoriais e tensoriais [28]. Na BEVL, utilizaremos ícones orientados 3D para visualização de dados vetoriais,[1] como a direção da normal em um ponto sobre uma superfície. De forma similar, pode-se usar ícones orientados para visualização de vetores de deslocamento ou de forças de superfície no contorno de um sólido. A Figura 4.6 mostra ícones orientados, em forma de cones utilizados para visualização dos deslocamentos nodais da esfera da Figura 1.3. A direção de cada cone é igual a direção do deslocamento nodal correspondente. A altura de cada cone é proporcional à magnitude do deslocamento nodal correspondente.

4.4 Geração de Superfícies de Corte

Muitas vezes deseja-se cortar um modelo com uma superfície e mostrar os valores interpolados, escalares ou vetoriais, na seção de corte definida pela superfície. Esse é justamente o caso de visualização de dados no interior de um sólido analisado pelo MEC, conforme proposto nesse trabalho. Essa técnica é conhecida como incisão de dados (ou *data cutting* ou simplesmente *cutting*) [28]. Essa técnica trabalha com duas informações: uma definição para a superfície de corte e um modelo para cortar. Assumiremos, neste trabalho, que a superfície de corte é definida através de uma *função implícita* do tipo

$$f^{-1}(0) = (x, y, z) \in \mathbb{R}^3 : f(x, y, z) = 0, \tag{4.2}$$

onde $f : \mathbb{R}^3 \rightarrow \mathbb{R}$ é de classe C^1.

Funções implícitas desse tipo possuem duas propriedades importantes:

- Descrição geométrica simples, permitindo a descrição de formas geométricas tais como planos, esferas, cilindros, cones, elipsóides e quádricas.

[1] SCHROEDER [28] classifica algoritmos que geram ícones como sendo algoritmos de modelagem. Como estamos utilizando ícones para visualização de vetores, abordaremos o tema nessa seção.

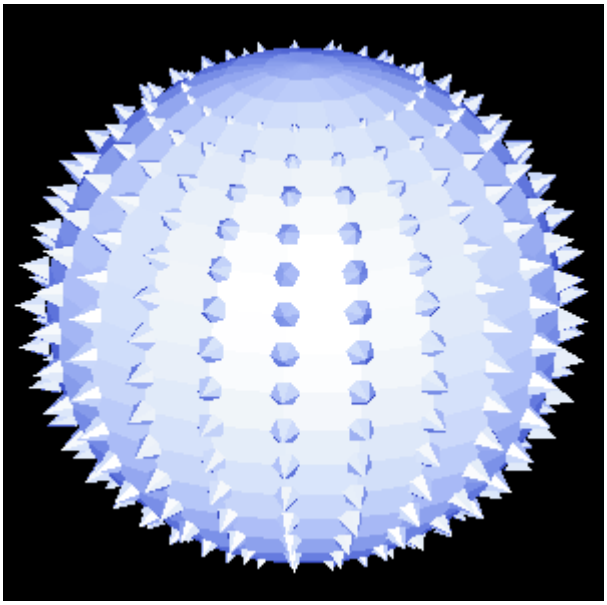

Figura 4.6: Ícones orientados para visualização de deslocamentos.

- Separação do espaço Euclidiano 3D em três regiões distintas: interna à superfície ($f(x, y, z) < 0$), sobre a superfície ($f(x, y, z) = 0$) e externa à superfície ($f(x, y, z) > 0$).

Um exemplo de função implícita é a equação da esfera centrada na origem de raio R

$$f(x, y, z) = x^2 + y^2 + z^2 - R^2, \tag{4.3}$$

a qual define as três regiões: na superfície da esfera, interna à esfera e externa à esfera. Qualquer ponto pode ser classificado dessa forma pela avaliação da Equação (4.3). A Figura 4.7 mostra uma representação bidimensional para a esfera dada pela Equação (4.3).

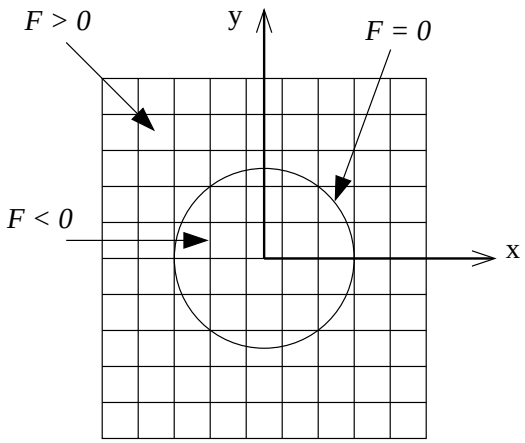

Figura 4.7: Representação bidimensional de uma esfera definida por função implícita.

Uma das características de funções implícitas é converter uma posição em $\mathbb{R}^3$ em um valor escalar. Podemos utilizar essa propriedade em combinação com um algoritmo de contornamento "marchando" em células para gerar superfícies de corte.

Dada a função implícita que define a superfície de corte, o algoritmo de incisão trabalha como segue. Para cada célula, valores são gerados pela avaliação de $f(x, y, z)$ para cada nó da célula. Se todos os nós estimam valores positivos ou negativos, a superfície não corta a célula. No entanto, se os nós estimam valores positivos e negativos, a superfície passa através da célula. Podemos utilizar uma operação de contornamento da célula para gerar a isosuperfície $f(x, y, z) = 0$. Atributos de valores dados podem ser calculados pela interpolação ao longo das arestas de corte.

A Figura 4.8 mostra uma superfície de corte gerada pelo método de incisão de dados na esfera da Figura 1.3. A figura mostra, também, mapa de cores e isolinhas sobre a superfície de corte, para componentes de deslocamentos na direção y.

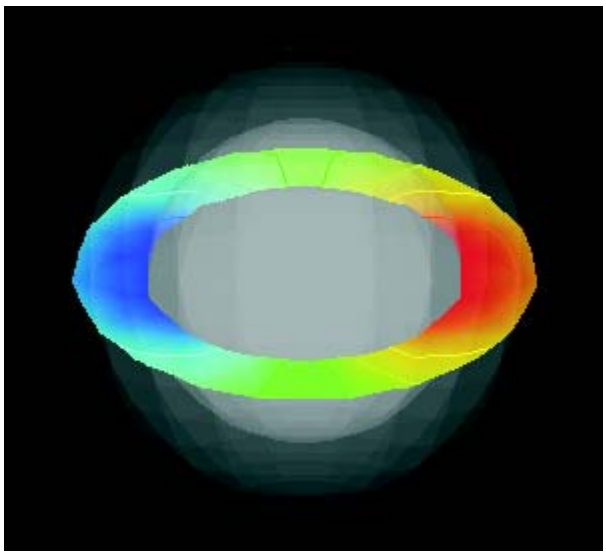

Figura 4.8: Incisão de dados.

4.5 Visualização em OSW

Conforme visto no Capítulo 1, um fluxo de visualização é definido por repositórios de dados e por processos que produzem, transformam e consomem dados. Em OSW, repositórios de dados são representados pelos diferentes tipos de modelos geométricos descritos no Capítulo 2. Nesta seção, introduziremos as principais classes de OSW de fontes, filtros e sumidouros de dados, utilizadas na implementação da BEVL. Iremos enfatizar as classes de objetos empregadas na implementação dos algoritmos escalares, vetoriais e de visualização em superfícies de corte, abordados neste capítulo. Outros tipos de fontes e de filtros, específicos da aplicação de modelagem de sólidos pelo MEC, serão abordados no Capítulo 5.

4.5.1 Classes de Fontes

O Capítulo 1 apresentou um fonte de dados como sendo um objeto que produz modelos. Em OSW, uma classe de fonte de dados de um tipo `tOutput` é definida através da classe paramétrica `tSource<tOutput>`, mostrada no Programa 4.1.

Um fonte de dados será, então, uma instância da classe `tSource<Output>`, onde o parâmetro `tOutput` representa o tipo de repositório de dados, ou modelo, que será gerado. O método `Execute()`, declarado na linha 9 e listado no Programa 4.2, inicia o processo

```
1   template <class tOutput>
2   class tSource
3   {
4     public:
5       tSource():  Output(0)
6       {}
7       virtual ~tSource();
8
9       virtual void Execute();
10      tOutput* GetOutput();
11
12    protected:
13      tOutput* Output;
14      virtual tOutput* ConstructOutput() const = 0;
15
16    private:
17      virtual void Run() = 0;
18  }; // tSource
```

Programa 4.1: Classe paramétrica `tSource<tOutput>`.

de criação do modelo a ser gerado, enviando as mensagens `ConstructOutput()` e `Run()` ao fonte. A primeira é responsável por construir uma instância do modelo de saída. A segunda executa o processo específico de geração dos dados do repositório. Classes derivadas de `tSource<tOutput>` devem sobrecarregar os métodos virtuais `Run()` e `ConstructOutput()`.

```
1   template <class tOutput>
2   void
3   tSource<tOutput>::Execute()
4   {
5     Delete(Output);
6     Output = ConstructOutput();
7     Run();
8   }
```

Programa 4.2: Método `tSource<tOutput>::Execute()`.

OSW possui quatro classes de fontes de dados principais, comentadas a seguir.

- `tShellSource`. Um objeto de uma classe derivada da classe abstrata `tShellSource`, cuja hierarquia de classes é mostrada na Figura 4.9, é um fonte genérico de modelos de cascas. Há dois tipos de fontes de modelos de cascas em OSW: leitores e varredores (*sweepers*). Um leitor é um objeto da classe `tShellReader`, o qual gera um modelo de cascas a partir de dados de um arquivo texto (a gramática de descrição de modelos de sólidos pode ser encontrada em [22]). Um varredor é um objeto da classe `tShellSweeper`, o qual gera modelos de cascas através de processos de varredura translacional, translacional cônica e rotacional. Um modelo de cascas em forma de cone, por exemplo, pode ser gerado por varredura translacional cônica de um círculo (na verdade, de um polígono regular). O processo consiste em convergir cada ponto do círculo — a curva geratriz — a um ponto fixo, o ápice do cone. Todos os pontos resultantes ao longo dessa translação definem a superfície lateral do cone. Modelos de cascas são utilizados

na BEVL para definição da geometria de ícones orientados (*glyphs*). Os exemplos do Capítulo 6 ilustram ícones em forma de cone, cuja geometria foi gerada por um fonte da classe `tConeShellSource`. O objeto utiliza um objeto `tShellSweeper` para gerar, por padrão, um cone com 6 faces laterais.

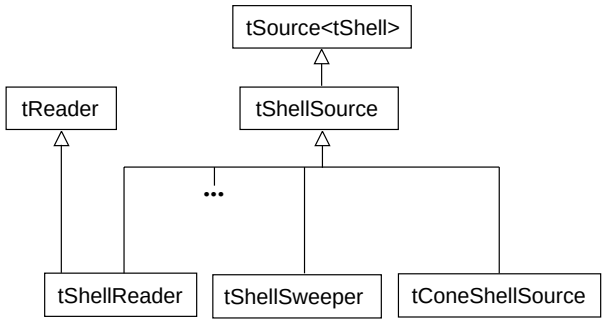

Figura 4.9: Hierarquia de classes de fontes de modelos de cascas.

- `tSolidSource`. Um objeto de uma classe derivada da classe abstrata `tSolidSource` é um fonte genérico de modelos de sólidos. Existem dois tipos de fontes de modelos de sólidos em OSW: leitores e varredores. De modo análogo ao que ocorre com modelos de cascas, um leitor é um objeto da classe `tSolidReader`, o qual gera modelos de sólidos a partir de dados de um arquivo texto. Um varredor é um objeto da classe `tSolidSweeper`, o qual gera modelos de sólidos através de processos de varredura.

- `tMeshSource`. Um objeto de uma classe derivada da classe abstrata `tMeshSource` é um fonte genérico de modelos de decomposição por células. Um dos tipos de fonte de modelos de decomposição por células em OSW é chamado de leitor. Um leitor é um objeto da classe `tMeshReader`, o qual gera modelos de decomposição por células a partir de dados de um arquivo texto. No Capítulo 5 descreveremos a classe `t3DDelaunay`, outro tipo de fonte de modelos de decomposição por células desenvolvido no trabalho. Um objeto `t3DDelaunay` gera modelos de decomposição por células através de um processo baseado na triangulação de Delaunay [26].

- `tPolySource`. Um objeto de uma classe derivada da classe abstrata `tPolySource` é um fonte genérico de modelos gráficos. A exemplo dos fontes de dados comentados anteriormente, um leitor de modelos gráficos, objeto da classe `tPolyReader`, gera modelos gráficos a partir de dados de um arquivo texto. Um objeto da classe `t3DGlyphSource`, derivada de `tPolySource`, é outro tipo de fonte de modelos gráficos. O objeto é utilizado na BEVL, conforme explicado no Capítulo 5, para a criação de ícones orientados em forma de cone.

4.5.2 Classes de Filtros

Filtros de dados são processos de transformação de modelos, isto é, um filtro é um processo que transforma um modelo de entrada em um outro modelo de saída. Em OSW, uma classe de filtro que toma como entrada um modelo do tipo `tInput` e gera como saída um modelo do tipo `tOutput`, é definida através da classe paramétrica `tFilter<tInput, tOutput>`, cuja interface pode ser vista no Programa 4.3. Note que a classe deriva de `tSource<tOutput>`, ou seja, um filtro que gera um modelo do tipo `tOutput` é um fonte desse tipo de modelo.

Um filtro é capaz de obter o modelo de entrada, através do método `GetInput()`, declarado na linha 10 e, também, ajustar o modelo de entrada, através do método `SetInput()`,

```
1    template <class tInput, class tOutput>
2    class tFilter:  public tSource<tOutput>
3    {
4       public:
5          tFilter():  Input(0)
6          {}
7
8          virtual ~tFilter();
9
10         tInput* GetInput();
11         tInput SetInput(tInput*);
12         virtual void Execute();
13
14      protected:
15         tInput* Input;
16   }; // tFilter
```

Programa 4.3: Classe paramétrica `tFilter<tInput, tOutput>`.

declarado na linha 11. O método `Execute()`, linha 12, é responsável por iniciar a execução do filtro, chamando o método `Run()`, próprio de cada filtro. Após verificar se o modelo de entrada existe, o método executa `tSource<tOutput>::Execute()`.

Na BEVL, foram desenvolvidos três tipos principais de filtros: filtros que geram modelos gráficos, filtros que geram modelos de decomposição por células e filtros que geram modelos geométricos de qualquer tipo. A hierarquia das classes de objetos que geram modelos gráficos, explicada a seguir, pode ser vista na Figura 4.10.

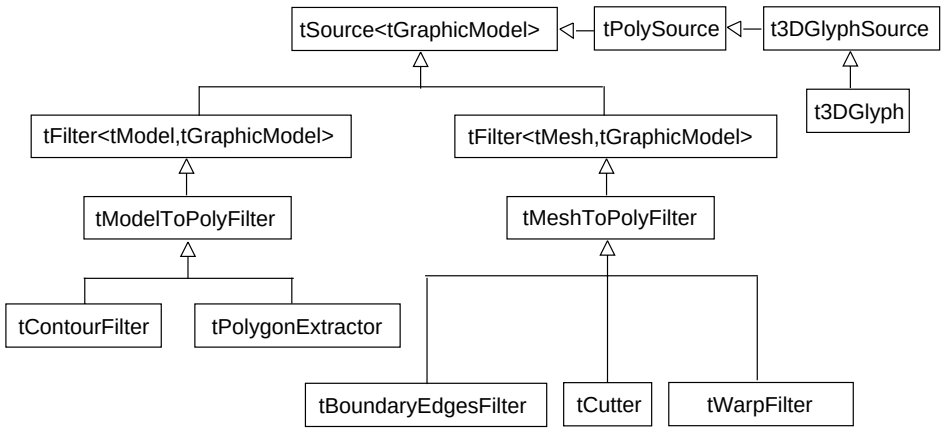

Figura 4.10: Hierarquia de classes de filtros que geram modelos gráficos.

1. `tModelToPolyFilter`. Esse tipo de filtro toma como entrada um modelo geométrico qualquer e gera como saída um modelo gráfico. Na BEVL são usados:

   - `tContourFilter`. Um objeto da classe `tContourFilter` é um filtro de contorno genérico, responsável pela geração de isolinhas e/ou isosuperfícies dos modelos utilizados na BEVL. A classe `tContourFilter` será comentada na Seção 4.5.4.
   - `tPolygonExtractor`. Um objeto da classe `tPolygonExtractor` é um filtro responsável pela "extração" de polígonos de um modelo geométrico qualquer. Por

exemplo, pode-se utilizar o filtro para extração das faces de contorno de um modelo de decomposição por células.

2. `tMeshToPolyFilter`. Esse tipo de filtro toma como entrada um modelo de decomposição por células e gera como saída um modelo gráfico. Na BEVL são usados:

   - `tBoundaryEdgesFilter`. Um objeto da classe `tBoundaryEdgesFilter` é um filtro que toma como entrada um modelo de decomposição por células e transforma o modelo em um modelo gráfico contendo as arestas de contorno do modelo de entrada (arestas de contorno, nesse contexto, são arestas usadas por só e somente só uma célula).
   - `tCutter`. Um objeto da classe `tCutter` é um filtro utilizado para a geração de seções de corte definidas por uma superfície implícita. A classe será discutida na Seção 4.5.6.
   - `tWarpFilter`. Um objeto da classe `tWarpFilter` é um filtro que toma como entrada um modelo de decomposição por células e gera como saída um modelo gráfico que representa a malha deformada. A classe `tWarpFilter` será comentada na Seção 4.5.5.

3. `t3DGlyph`. Um objeto da classe `t3DGlyph` é um filtro que toma como entrada um modelo de decomposição por células e gera como saída um modelo gráfico contendo, em cada vértice, um ícone (um cone) que representa vetores de deslocamento ou forças de superfícies no contorno do modelo. A classe será comentada na Seção 4.5.5.

A hierarquia de filtros que geram modelos de decomposição por células pode ser vista na Figura 4.11. Os filtros são descritos a seguir.

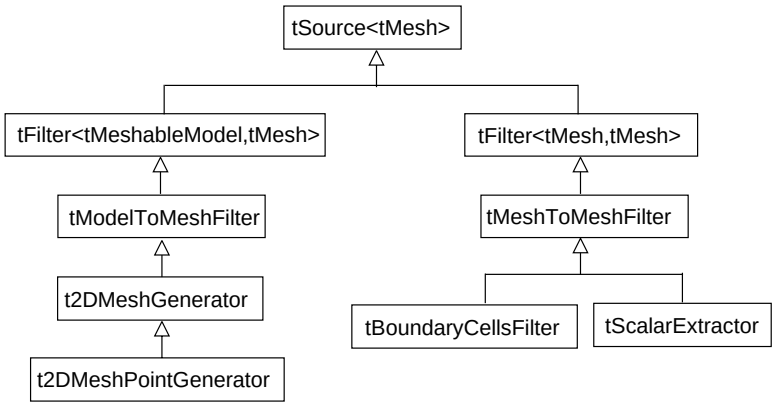

Figura 4.11: Hierarquia de filtros que geram modelos de decomposição por células.

1. `tModelToMeshFilter`. O filtro toma como entrada um modelo geométrico qualquer e gera como saída um modelo de decomposição por células. Um objeto da classe `t2DMeshPointGenerator`, por exemplo, toma como entrada um modelo geométrico qualquer e gera como saída uma malha correspondente à triangulação 2D do modelo geométrico. Uma aplicação desse filtro é mostrada na Seção 5.3.3.

2. `tMeshToMeshFilter`. Esse tipo de filtro toma como entrada um modelo de decomposição por células e pode gerar como saída o mesmo modelo ou um novo modelo de decomposição por células. Na BEVL são usados:

- `tBoundaryCellsFilter`. Um objeto da classe `tBoundaryCellsFilter` é um filtro utilizado na geração da malha de elementos de contorno. O filtro toma como entrada um modelo de decomposição por células e produz como saída, um modelo de decomposição por células contendo as células de contorno das células do modelo de entrada (o modelo de saída pode ser o mesmo modelo de entrada ou uma nova malha). Na Seção 5.4, ilustramos a utilização do filtro na geração de uma malha de elementos de contorno a partir de uma malha de elementos de volume.
- `tScalarExtractor`. Um objeto da classe `tScalarExtractor` é um filtro que toma como entrada um modelo de decomposição por células e gera como saída o mesmo modelo de entrada. O filtro "extrai" os valores (escalares) dos graus de liberdade nodais do modelo para visualização, sendo utilizado na implementação do mapa de cores e do contornamento, explicados na Seção 4.5.4.

Filtros que transformam modelos geométricos de qualquer tipo em modelos geométricos de qualquer tipo são objetos de uma classe derivada da classe `tModelToModelFilter`, como pode ser visto na Figura 4.12. Um exemplo é o filtro da classe `tImplicitSetter`, derivada de `tModelToModelFilter`, utilizada na BEVL para ajustar o escalar de cada vértice de um modelo de acordo com o valor de uma função implícita cujos argumentos são as coordenadas do vértice. O Capítulo 6 demonstra a utilização do filtro.

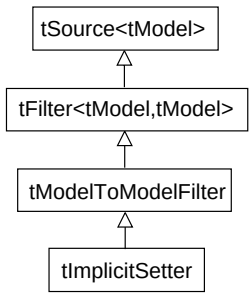

Figura 4.12: Hierarquia de filtros que geram modelos geométricos.

4.5.3 Classes de Sumidouros

Um processo sumidouro é um processo consumidor de um ou mais repositórios de dados. Em OSW, existem três tipos principais de processos sumidouros: *mapeadores*, *sintetizadores de imagens* e *escritores*, os quais serão comentados a seguir.

Um sintetizador de imagens OSW é um objeto de uma classe derivada da classe abstrata `tRenderer`. O sintetizador mais comumente empregado em aplicações de modelagem OSW é um objeto da classe `tScanner`, derivada de `tRenderer`. Esse sintetizador permite a geração de imagens de qualquer um dos modelos do Capítulo 2 em cinco modos distintos: fio-de-arame, linhas escondidas, superfícies escondidas com tonalização constante e de Gouraud e, transparência, empregando o algoritmo de linha de rastreio (*scanline*) [24]. (As classes `tRenderer` e `tScanner` não serão descritas aqui). Nos modos superfície escondida com tonalização de Gouraud, especificamente, a cor de cada ponto da superfície visível de um modelo é determinada por um objeto `tScanner` em função das luzes da cena e das cores dos vértices que definem a superfície. Normalmente, a cor de um vértice é função das propriedades materiais da superfície.

Um mapeador é um "parceiro" de um sintetizador de imagem, responsável pela definição das cores dos vértices das superfícies de um modelo OSW, a partir do qual o sintetizador

gerará uma imagem. Um mapeador é um objeto da classe `tMapper`, cuja interface parcial é mostrada no Programa 4.4.

```
1    class tMapper
2    {
3       public:
4          tMapper(tModel*);
5          tLookupTable* GetLookupTable();
6          void SetModel(tModel*);
7          void SetLookupTable(tLookupTable*);
8          void UseScalars(bool);
9          virtual void Render();
10         ...
11   }; // tMapper
```

Programa 4.4: Parte da interface da classe `tMapper`.

Um mapeador toma como entrada qualquer modelo geométrico, como mostra o construtor da classe, na linha 4. O objeto mais importante de um mapeador é sua *tabela de cores*, instância da classe `tLookupTable`, comentada na Seção 4.5.4. A tabela de cores é utilizada pelo mapeador para atribuir cores aos vértices do modelo de entrada (a cor é armazenada no atributo `Color` da classe `tVertex`, Programa 2.4), a partir das quais o sintetizador da cena sintetiza uma imagem colorida do modelo (método `Render()`, declarado na linha 9). A classe `tMapper` permite a alteração da tabela de cores do mapeador, através do método `SetLookupTable()`, declarado na linha 7, bem como a solicitação ao mapeador para que não utilize a tabela de cores, método `UseScalars()`, linha 8.

Um escritor é um objeto de uma classe derivada da classe abstrata `tWriter`. Um escritor é responsável por escrever em um arquivo texto, as informações que descrevem determinado modelo geométrico, utilizando a mesma gramática do leitor correspondente. Há quatro classes de escritores em OSW: `tMeshWriter`, `tPolyWriter`, `tSolidWriter` e `tSolidWriter`, não comentadas aqui.

4.5.4 Visualização de Escalares

Nesta seção, serão apresentadas as classes envolvidas na implementação dos algoritmos escalares da BEVL, discutidos na Seção 4.2. Para que seja possível a visualização de escalares, o atributo `Scalar` da classe `tVertex`, classe base dos vértices dos modelos de OSW, Programa 2.4, deve armazenar o valor do escalar que será visualizado.

Para um modelo analisado pelo MEC, os escalares de interesse são componentes de deslocamento ou força de superfície em pontos do contorno do modelo, ou componentes de deslocamento ou tensão em pontos do domínio do modelo. Para um nó de contorno, tais componentes estão armazenados, como resultado da análise numérica, nos graus de liberdade do nó. A visualização de escalares, nesse caso, utiliza um filtro de extração de escalares da classe `tScalarExtractor`, listada no Programa 4.5. O filtro toma como entrada um modelo de decomposição por células e gera como saída o mesmo modelo de entrada.

O método público `SelectField()`, linha 7, define qual o campo escalar será extraído dos nós de contorno do modelo de entrada. O primeiro argumento é o número do grau de liberdade, podendo assumir, no caso tridimensional, os valores 0, 1 e 2, os quais correspondem aos componentes x, y e z, respectivamente. O segundo argumento é 0 para deslocamento ou 1 para força de supefície. Os valores dos argumentos do método são armazenados nos atributos `DOFNumber` e `VarNumber`, declarados nas linhas 11 e 12, respectivamente. Após a execução

```
1   class tScalarExtractor:  public tMeshToMeshFilter
2   {
3     public:
4       tScalarExtractor();
5
6       double GetScalarRange(double[2]) const;
7       void SelectField(int, int);
8
9     private:
10      double Range[2];
11      int DOFNumber;
12      int VarNumber;
13
14      void Run();
15  }; // tScalarExtractor
```

Programa 4.5: Classe `tScalarExtractor`.

do filtro, os valores mínimo e máximo dos escalares extraídos são armazenados no atributo `Range`, declarado na linha 10, e obtido com o método `GetScalarRange()`, declarado na linha 6. O método `Run()`, linha 14, "atravessa" a coleção de nós de contorno do modelo de entrada e, para cada nó, armazena o valor do escalar a ser visualizado no atributo `Scalar` do nó. A implementação do método é listada no Programa 4.6.

```
1   void
2   tScalarExtractor::Run()
3   {
4     Range[0] = +D_HUGE;
5     Range[1] = -D_HUGE;
6     for (tNodeIterator nit(Input->GetNodeIterator()); nit; ++nit)
7     {
8       double scalar = (nit.Current()->DOF(DOFNumber))[VarNumber];
9       if (scalar < Range[0])
10        Range[0] = scalar;
11      if (scalar > Range[1])
12        Range[1] = scalar;
13      nit.Current()->Scalar = scalar;
14    }
15    if (IsEqual(Range[0], Range[1]))
16    {
17      Range[0] = -D_HUGE;
18      Range[1] = +D_HUGE;
19    }
20  }
```

Programa 4.6: Método `tScalarExtractor::Run()`.

Mapa de Cores

Como visto anteriormente, mapeamento de cores é uma técnica de visualização de escalares, que mapeia dados escalares em cores. Esse mapeamento é realizado com a utilização de um mapeador, um objeto da classe `tMapper`. O mapeamento escalar é implementado utilizando-se uma tabela de cores com n entradas. Essa tabela de cores é um objeto da classe `tLookupTable`, Programa 4.7. O construtor da classe toma como entrada o número de cores da tabela. O construtor inicializa 256 cores, variando do azul até o vermelho.

```
1    class tLookupTable: public tSharedObject
2    {
3       public:
4          ...
5          void SetScalarRange(double, double);
6          double GetMinScalar() const;
7          double GetMaxScalar() const;
8          void SetHueRange(double, double);
9          double GetMinHue() const;
10         double GetMaxHue() const;
11         void SetSaturationRange(double, double);
12         double GetMinSaturation() const;
13         double GetMaxSaturation() const;
14         void SetValueRange(double, double);
15         double GetMinValue() const;
16         double GetMaxValue() const;
17         int GetNumberOfColors() const;
18         ...
19   }; // tLookupTable
```

Programa 4.7: Parte da interface da classe `tLookupTable`.

O método `SetScalarRange()`, linha 5, ajusta o intervalo de escalares da tabela. Os métodos `GetMinScalar()` e `GetMaxScalar()`, declarados nas linhas 6 e 7 retornam o menor e o maior valor escalar da tabela, respectivamente. `SetHueRange()`, linha 8, ajusta o intervalo de matiz das cores da tabela. Os métodos `GetMinHue()` e `GetMaxHue()`, linhas 9 e 10, retornam o menor e o maior valor de matiz das cores da tabela, respectivamente. O intervalo de saturação das cores da tabela é ajustado pelo método `SetSaturationRange()`, linha 11. Os métodos `GetMinSaturation()` e `GetMaxSaturation()`, declarados nas linhas 12 e 13, retornam o menor e o maior valor de saturação das cores da tabela, respectivamente. O método `SetValueRange()`, linha 14, ajusta o intervalo de valor das cores da tabela. Os métodos `GetMinValue()` e `GetMaxValue()`, linhas 15 e 16, retornam o menor e o maior valor do atributo valor das cores da tabela, respectivamente. O método `GetNumberOfColors()`, linha 17, retorna o número de cores da tabela.

A geração do mapa de cores ocorre a partir dos valores armazenados em cada campo `Scalar` de cada vértice do modelo de entrada. A cada vértice do modelo é, então, associada uma cor da tabela de cores. O mapa de cores sobre cada célula do modelo é gerado por interpolação bilinear das cores nos vértices da célula através do processo de tonalização de Gouraud [22]. O resultado pode ser visto na Figura 1.7.

Contornamento

O contornamento é uma extensão natural do mapa de cores e é implementado com a utilização de um filtro de contorno, um objeto da classe `tContourFilter`, Programa 4.8.

```
1    class tContourFilter: public tModelToPolyFilter
2    {
3       public:
4          tContourFilter();
5          void SetValue(int, double);
6          void GenerateValues(int, double, double);
7          ...
8       protected:
9          void Run();
10         ...
11   }; // tContourFilter
```

Programa 4.8: Parte da interface da classe `tContourFilter`.

`tContourFilter` toma como entrada um modelo geométrico qualquer e gera como saída um modelo gráfico, representando as isolinhas ou isosuperfícies geradas a partir dos valores escalares dos vértices do modelo de entrada, através do algoritmo "marchando em células". O método `SetValue()`, linha 5, ajusta o valor de contorno a ser utilizado e o método `GenerateValues()`, linha 6, ajusta a quantidade de isolinhas (ou isosuperfícies) a serem geradas e os valores a serem considerados. O método `Run()`, linha 9, executa o contornamento. A implementação do método é mostrada no Programa 4.9.

Para um modelo de entrada qualquer, `tContourFilter::Run()` envia, para cada valor escalar de contornamento, a mensagem `Contour()` para o modelo. No caso específico do modelo de entrada ser um modelo de decomposição por células (verificado pelo **`dynamic_cast`** da linha 7), o método faz um pouco melhor, enviando, para cada valor escalar de contornamento, a mensagem `Contour()` para cada célula do modelo de entrada.

O Programa 4.15 mostra, como exemplo, a tabela de casos construída para enumerar todos os estados topológicos possíveis de um tetraedro. A implementação do algoritmo de contornamento para um tetraedro é mostrada no Programa 4.16. (Ambos os programas são listados no final deste capítulo.)

4.5.5 Visualização de Vetores

Nesta seção, apresentamos as classes envolvidas na implementação do algoritmo de deformação e no algoritmo de geração de ícones orientados descritos na Seção 4.3.

Deformação

Conforme descrito anteriormente, o processo de deformação trabalha com os vértices de um modelo de decomposição por células, extraindo, de cada vértice, um vetor de interesse, o qual sofre uma transformação de escala, gerando um modelo deformado para visualização. Esse processo de transformação é realizado utilizando-se um objeto da classe `tWarpFilter`, mostrada no Programa 4.10.

Um objeto da classe `tWarpFilter` é um filtro que toma como entrada um modelo de decomposição por células e produz como saída um modelo gráfico que representa a estrutura deformada do modelo de entrada. O método `Run()`, declarado na linha 14, é responsável pela

```
1    void
2    tContourFilter::Run()
3    {
4       tBoundingBox boundingBox(*Input);
5       Output->InitPointLocator(boundingBox);
6
7       if (dynamic_cast<tMesh*>(Input))
8       {
9          for (tCellIterator cit(((tMesh*)Input)->GetCellIterator()); cit;)
10         {
11            double cellScalars[20];
12            tCell* cell = cit++;
13
14            for (int i = 0; i < cell->GetNumberOfNodes(); i++)
15               cellScalars[i] = cell->GetNode(i)->Scalar;
16            for (int i = 0; i < NumberOfValues; i++)
17               cell->Contour(Values[i], cellScalars, *Output);
18         }
19      }
20      else
21         for (int i = 0; i < NumberOfValues; i++)
22            Input->Contour(Values[i], 0, *Output);
23      Output->DeletePointLocator();
24   }
```

Programa 4.9: Método `tContourFilter::Run()`.

execução do filtro. O método percorre todos os vértices do modelo de entrada, aplicando em cada vértice uma transformação de escala `S`, sobre o deslocamento **u** do vértice, que faz com que o modelo de saída fique com a aparência deformada.

Ícones Orientados

Como descrito anteriormente, um ícone orientado é um componente de um sistema de visualização, definido por um conjunto de dados ou uma imagem, que pode ser parametrizado por vários dados de entrada. Na BEVL, um ícone orientado é um objeto da classe `t3DGlyph`, derivada da classe `t3DGlyphSource`. Parte da interface da classe `t3DGlyph` pode ser vista no Programa 4.11.

Um ícone orientado é gerado por um filtro que recebe como entrada dois modelos quaisquer e gera como saída um modelo gráfico. O primeiro modelo de entrada, ajustado com o método `SetInput()`, linha 8, é o modelo que fornecerá os pontos e os vetores a partir dos quais os ícones serão gerados. O segundo modelo de entrada, ajustado com o método `SetSource()` (declarado na classe base `t3DGlyphSource` e não mostrado aqui), define a geometria dos ícones. Se não especificado, o filtro considera como segundo modelo uma casca cônica gerada por um filtro da classe `tConeShellSweeper`. `GetInput()`, declarado na linha 7, fornece o primeiro modelo de entrada. O método `Execute()`, declarado na linha 9, executa o filtro, extraindo os pontos do modelo de entrada e gerando, sobre cada um dos pontos, o modelo geométrico que representa a direção da informação que se deseja visualizar (normal, deslocamento, tração).

Um ícone orientado pode, ainda, orientar, aplicar escala, transladar, deformar ou alterar

```
1   class tWarpFilter: public tMeshToPolyFilter
2   {
3     public:
4       tWarpFilter();
5
6       void SetScale(double);
7       double GetScale() const;
8
9     private:
10      double S;
11
12      tGraphicModel* ConstructOutput() const;
13      void Run();
14  }; // tWarpFilter
```

Programa 4.10: Classe `tWarpFilter`.

```
1   class t3DGlyph: public t3DGlyphSource
2   {
3     public:
4       t3DGlyph():  Input(0) {}
5       ~t3DGlyph();
6
7       tModel* GetInput();
8       void SetInput(tModel*);
9       void Execute();
10    ...
11  }; // t3DGlyph
```

Programa 4.11: Parte da interface da classe `t3DGlyph`.

de qualquer forma sua aparência, em resposta a seus dados de entrada. Isso é possível com a aplicação de métodos herdados da classe `t3DGlyphSource`:

- `GetSource()`. O método `GetSource()` fornece o modelo geométrico que será utilizado para representar o ícone orientado. Na BEVL, utilizamos pequenos cones orientados para representar ícones orientados.
- `SetSource()`. O método `SetSource()` ajusta o modelo geométrico de representação do ícone orientado, conforme discutido anteriormente.
- `GetMinRange()` e `GetMaxRange()`. Os métodos `GetMinRange()` e `GetMaxRange()` fornecem os valores mínimo e máximo de variação da informação que o ícone orientado representa, respectivamente.
- `GetScaleFactor()` e `SetScaleFactor()`. O fator de escala a ser aplicado é manipulado pelos métodos `GetScaleFactor()` e `SetScaleFactor()`. `GetScaleFactor()` fornece o fator de escala aplicado e `SetScaleFactor()` ajusta o fator de escala com um novo valor.

4.5.6 Visualização de Dados em Superfícies de Corte

Conforme visto anteriormente, adotaremos funções implícitas como da Equação (4.2) para definição de superfícies de corte. Em OSW, uma função implícita é um objeto de uma classe derivada da classe abstrata `tImplicitFunction`, Programa 4.12. Classes concretas de funções implícitas devem sobrecarregar os métodos virtuais puros `EvaluateFunction()` e `EvaluateGradient()`, linhas 6 e 7, responsáveis pela avaliação da função e do gradiente da função em um ponto `p`, respectivamente.

```
1    class tImplicitFunction: public tModel
2    {
3       public:
4          double FunctionValue(const t3DVector&) const;
5          t3DVector FunctionGradient(const t3DVector&) const;
6          double EvaluateFunction(const t3DVector&) const = 0;
7          t3DVector EvaluateGradient(const t3DVector&) const = 0;
8       protected:
9          t3DTransfMatrix T;
10   }; // tImplicitFunction
```

Programa 4.12: Parte da interface da classe `tImplicitFunction`.

A Figura 4.13 mostra parte da hierarquia de classes de funções implícitas de quádricas simples. Um cilindro é um objeto da classe `tCylinder`; um cone é um objeto da classe `tCone`; um plano é um objeto da classe `tPlane`; uma esfera é um objeto da classe `tSphere`. Todas essas classes são derivadas de `tImplicitFunction` e todas sobrecarregam os métodos `EvaluateFunction()` e `EvaluateGradient()`.

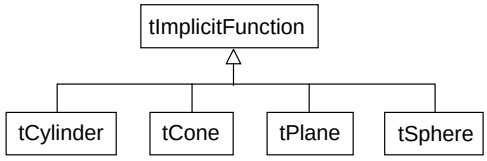

Figura 4.13: Hierarquia de classes de funções implícitas de quádricas simples.

Funções implícitas também podem ser combinadas para criar modelos mais complexos através de operações booleanas. Para implementar combinações de funções implícitas foi criada a classe `tImplictBoolean`, da qual derivam `tImplicitUnion`, `tImplicitIntersect` e `tImplicitDiff`, as quais implementam as operações de união, intersecção e diferença de superfícies definidas por funções implícitas, respectivamente. A geração de uma superfície de corte é executada por um objeto da classe `tCutter`, listada no Programa 4.13.

Um objeto da classe `tCutter` é um filtro que toma como entrada um modelo de decomposição por células e gera como saída um modelo gráfico. O construtor da classe, declarado na linha 4, ajusta a definição para a superfície utilizada para cortar o modelo (a superfície é definida por uma função implícita `F`), declarada na linha 8. O modelo a ser "cortado" é definido pela entrada do filtro, nesse caso um modelo de decomposição por células.

O método `Run()`, declarado na linha 10, executa o filtro. O Programa 4.14 mostra a implementação do método, que gera valores para todos os nós do modelo de decomposição por células de entrada, através da avaliação da função implícita no nó. Uma vez computados os valores nodais da função implícita, a superfície é gerada através do contornamento de cada célula do modelo de entrada, no escalar 0. Isso é feito invocando-se o método `Contour()` de

```
1    class tCutter: public tMeshToPolyFilter
2    {
3       public:
4          tCutter (tImplicitFunction&);
5          ~tCutter();
6
7       private:
8          tImplicitFunction* F;
9
10         void Run();
11   }; // tCutter
```

Programa 4.13: Classe `tCutter`.

cada célula, linha 19. Uma vez gerada a superfície de corte, seus dados escalares e vetoriais podem ser visualizados aplicando-se os algoritmos descritos na Seção 4.2 e na Seção 4.3.

```
1    void
2    tCutter::Run()
3    {
4       tFixedArray <double> fValues(Input->GetNumberOfNodes());
5       double scalars[20];
6
7       Input->RenumerateNodes(true);
8       for (tNodeIterator nit(Input->GetNodeIterator()); nit; nit++)
9       {
10         tNode* node = nit.Current();
11         fValues[node->Number] = F->EvaluateFunction(node->Position);
12      }
13      Output->InitPointLocator(tBoundingBox(*Input));
14      for (tCellIterator cit(Input->GetCellIterator()); cit; cit++)
15      {
16         tCell* cell = cit.Current();
17         for (int i = 0; i < cell->GetNumberOfNodes(); i++)
18            scalars[i] = fValues[cell->GetNode(i)->Number];
19         cell->Contour(0, scalars, *Output);
20      }
21      Output->DeletePointLocator();
22   }
```

Programa 4.14: Método `tCutter::Run()`.

4.6 Sumário

Visualização é a transformação de dados de um modelo de objeto em imagens que representam a estrutura e o comportamento do objeto. Nesse capítulo, descrevemos os algoritmos de visualização da BEVL, bem como as classes de objetos que os implementam.

Algoritmos escalares transformam escalares associados com algum componente de um modelo, geralmente os vértices do modelo. Apresentamos dois algoritmos escalares, o mapa de cores e o contornamento. O mapa de cores mapeia dados escalares em cores e o contornamento

```
struct tetrahedra_EDGE_CASES
{
   EDGE_LIST edges[5];
};

static tetrahedra_EDGE_CASES tetrahedra_EdgeCases[] =
{
   {{-1, -1, -1, -1, -1}}
   {{ 0, 1, 2, -1, -1}}
   {{ 0, 4, 3, -1, -1}}
   {{ 1, 2, 4, 3, -1}}
   {{ 1, 3, 5, -1, -1}}
   {{ 0, 2, 5, 3, -1}}
   {{ 0, 4, 5, 1, -1}}
   {{ 2, 4, 5, -1, -1}}
   {{ 2, 5, 4, -1, -1}}
   {{ 0, 1, 5, 4, -1}}
   {{ 0, 2, 5, 3, -1}}
   {{ 1, 5, 3, -1, -1}}
   {{ 1, 3, 4, 2, -1}}
   {{ 0, 3, 4, -1, -1}}
   {{ 0, 2, 1, -1, -1}}
   {{-1, -1, -1, -1, -1}}
};
```

Programa 4.15: Tabela de casos para um tetraedro.

gera isolinhas (ou isosuperfícies) para uma faixa de valores escalares.

Algoritmos vetoriais transformam dados vetoriais que representam direção e magnitude resultantes, por exemplo, de campos de deslocamentos gerados pela análise de um sólido pelo MEC. Apresentamos duas técnicas para visualização de dados vetoriais. O algoritmo de deformação "deforma" a geometria de um modelo, de acordo com o campo vetorial considerado. Ícones orientados representam uma técnica versátil para visualização de dados de todos os tipos, sendo utilizados para visualização de dados vetoriais, tais como normais e deslocamentos.

O algoritmo de incisão de dados corta um modelo com uma superfície (definida por uma função implícita) e mostra os valores interpolados, escalares ou vetoriais, na seção de corte definida pela superfície.

As classes de OSW que implementam os algoritmos de visualização foram introduzidas e seus aspectos gerais discutidos. Para a implementação dos algoritmos de visualização da BEVL realizamos as seguintes atividades:

- Revisão das classes de OSW `tScanner`, `tContourFilter` e de todas as classes bases de fontes e filtros;
- Criação das classes `t3DGlyph`, `tCutter`, `tImplicitFunction` e suas derivadas, `tWarpFilter`, `tBoundaryCellsFilter` e `tContourFilter`.

```
void t4N3DCell::Contour(double value, double* scalars, tGraphicModel& model)
{
   static int CASE_MASK[4] = {1, 2, 4, 8};
   int index = 0, count = 0;
   tetrahedra_EDGE_CASES* edgeCase;
   EDGE_LIST* edge;
   t3DVector p[4], t3DVector v[4];
   double s[4];

   for (int i = 0; i < 4; i++)
      if (scalars[i] >= value) index |= CASE_MASK[i];
   edgeCase = tetrahedra_EdgeCases + index;
   if (*(edge = edgeCase->edges) > -1)
   {
      do
      {
         tEdge* e = GetEdge(*edge);
         t3DVector dp = e->V1->Position - e->V0->Position;
         double ds = e->V1->Scalar - e->V0->Scalar;
         t3DVector dv = e->V1->Vector - e->V0->Vector;
         int v0 = GetEdgeV0Index(*edge);
         int v1 = GetEdgeV1Index(*edge);
         double t = (value - scalars[v0]) / (scalars[v1] - scalars[v0]);
         s[count] = e->V0->Scalar + t * ds;
         v[count] = e->V0->Vector + t * dv;
         p[count++] = e->V0->Position + t * dp;
      } while (*(++edge) > -1);
      if (count < 3) return;
      t3DVector d0 = p[1] - p[0];
      t3DVector d1 = p[2] - p[1];
      if (d1.Cross(d0).IsNull()) return;
      if (count == 4)
      {
         tPolygon* poly = new tPolygon(model, fid++);
         for (int i = 0; i < 4; i++)
         {
            tVertex* pv = poly->mv(p[i]);
            pv->Scalar = s[i];
            pv->Vector = v[i];
         }
         return;
      }
      tTriangle* tri = new tTriangle(model, fid++, p[0], p[1], p[2]);
      for (int i = 0; i < 3; i++)
      {
         tri->GetVertex(i)->Scalar = s[i];
         tri->GetVertex(i)->Vector = v[i];
      }
   }
}
```

Programa 4.16: Método `t4N3DCell::Contour()`.

CAPÍTULO 5

Solução do Problema

5.1 Introdução

No Capítulo 1, definimos o problema fundamental desse trabalho, o desenvolvimento de classes de objetos para a visualização de dados resultantes da análise numérica de sólidos através do MEC. A solução do problema envolve as seguintes etapas, descritas de forma resumida. Estaremos considerando um sólido definido por um domínio Ω de contorno Γ.

1. Discretização do domínio Ω em células de domínio do tipo tetraedros. A malha de elementos de volume decorrentes desta discretização conterá nós sobre o contorno Γ e nós internos no domínio Ω. Um filtro de geração de malhas tridimensionais que toma como entrada um modelo de sólidos de variedade de dimensão 2 e produz como saída uma malha de tetraedros está sendo desenvolvido para esse propósito [19]. Devido à indisponibilidade desse gerador, escrevemos nosso próprio gerador, mais simples, mas que possibilita a geração de dados para os testes da BEVL.

2. Desenvolvimento de um filtro de geração de malhas de elementos de contorno triangulares. Tal filtro tomará como entrada a malha de tetraedros gerada no passo anterior e produzirá, como saída, a malha de elementos de contorno triangulares. Os elementos de contorno triangulares gerados pelo filtro serão definidos pelas faces (triangulares) sem vizinhança dos tetraedros. Uma face sem vizinhança é uma face na qual incide só e somente só um tetraedro. Após a aplicação do filtro, teremos um conjunto de dados constituído por duas malhas de elementos — uma malha de elementos de volume tetraédricos e uma malha de elementos de contorno triangulares — que usam a mesma coleção de nós. Os nós dos elementos de contorno serão rotulados como *nós de contorno*, e os demais nós serão rotulados como *nós internos*.

3. Aplicação dos passos listados na Seção 3.4 para resolver o modelo pelo MEC. Utilizamos, neste passo, o analisador desenvolvido em [22]. Os resultados serão valores de deslocamentos e forças de superfície em todos os nós rotulados como nós de contorno no passo anterior. Conhecidos os valores nodais dos elementos de contorno, podemos determinar, por interpolação, os valores em quaisquer pontos do contorno Γ do sólido. A partir dos valores de contorno obtidos computar, por integração, os valores de deslocamento e tensões internas em todos os nós rotulados como nós internos no passo 2. Com isso, serão conhecidos os valores em todos os nós do conjunto de dados resultante

do passo 1. Conhecidos os valores nodais dos elementos de volume, podemos determinar, por interpolação, os valores em quaisquer pontos do domínio Ω do sólido, usando as funções de interpolação do tetraedro.

4. Utilização dos objetos da BEVL para visualização dos dados escalares (mapa de cores, isolinhas e isosuperfícies) e vetoriais (malha deformada e ícones orientados) de contorno e de domínio, obtidos no passo anterior. Os dados de domínio serão visualizados em seções definidas por superfícies de corte geradas através do método de incisão de dados, descrito na Seção 4.4.

Neste capítulo, descreveremos como empregamos OSW no desenvolvimento de duas aplicações de modelagem para teste das classes de objetos da BEVL, utilizadas em cada uma das etapas mencionadas anteriormente, cumpridas para solucionar o problema proposto. Começaremos na Seção 5.2, apresentando uma visão geral dos principais componentes da arquitetura de uma aplicação OSW. Os componentes específicos das aplicações desenvolvidas, denominadas BEG (*Boundary Element Generator*) e BEV (*Boundary Element Viewer*) também serão introduzidos na Seção 5.2, e empregados nas seções seguintes. Utilizamos a BEG para a geração das malhas de elementos de volume e de contorno e para a determinação das condições de contorno a serem empregadas às malhas. A BEV permite a visualização dos resultados da análise dos modelos, com a utilização dos algoritmos de visualização. Na Seção 5.3, apresentamos a solução da primeira etapa, a discretização do domínio do sólido. A segunda etapa, a geração da malha de elementos de contorno triangulares é comentada na Seção 5.4. A terceira etapa, a solução do modelo pelo MEC é feita por outra aplicação, sem interface gráfica, e não será discutida neste capítulo. As classes de analisadores e o processo de análise foram apresentados no Capítulo 3. A quarta e última etapa, a visualização dos dados, é descrita na Seção 5.5.

5.2 As Aplicações de Modelagem para Teste da BEVL

Um aplicação de modelagem de OSW é um programa para análise e/ou visualização de modelos estruturais. Tipicamente, as tarefas executadas por uma aplicação de modelagem (não necessariamente todas) são:

1. Criação, modificação e manutenção de modelos, pela adição, remoção e alteração dos seus dados.

2. "Atravessamento" de modelos para extração de dados geométricos para síntese de imagens.

3. "Atravessamento" de modelos para extração de dados para análise numérica.

4. Apresentação de ambos os dados, isto é, síntese de imagem de um modelo geométrico e saída do processo de análise.

5. Apresentação de elementos de interface homem-computador, tais como menus e caixas de diálogo.

Alguns componentes de uma aplicação de modelagem foram introduzidos nos capítulos anteriores, bem como as classes de objetos correspondentes a tais componentes. No Capítulo 2, vimos os componentes principais de uma aplicação de modelagem, os modelos geométricos de OSW. Apresentamos também os iteradores de componentes topológicos dos modelos geométricos. No Capítulo 3, introduzimos elementos de contorno, células internas e analisadores

MEC. Há muitos outros tipos de elementos e analisadores disponíveis na OCL, mas não relevantes para o trabalho. No Capítulo 4, vimos fontes de dados, filtros, sintetizadores de imagens e mapeadores. Nesta seção, apresentaremos uma visão geral das classes correspondentes à aplicação propriamente dita, ao documento da aplicação, suas cenas e vistas, tal como ilustrado no diagrama de classes da Figura 5.1. Maiores detalhes podem ser vistos em [22].

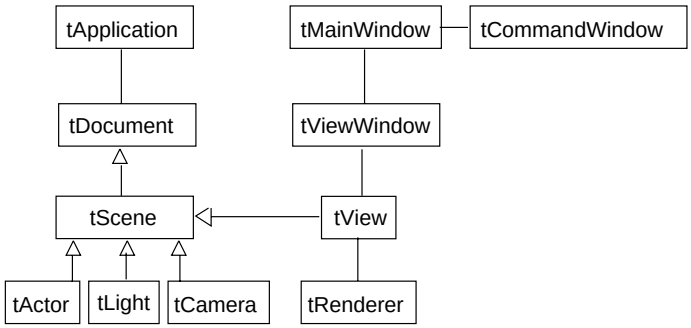

Figura 5.1: Parte da hierarquia de classes de uma aplicação de OSW.

Uma aplicação de modelagem de OSW é um objeto de uma classe derivada da classe `tApplication`, a qual gerencia uma base de dados chamada *documento* da aplicação. Os métodos de `tApplication` permitem inicializar a aplicação e sua janela principal e criar, abrir, salvar e fechar um documento de aplicação genérico. Classes derivadas de `tApplication` devem sobrecarregar o método virtual `ConstructDoc()`, para construir a base de dados particular da aplicação.

Uma base de dados de uma aplicação é um objeto de uma classe derivada da classe `tDocument`. Um documento da aplicação armazena, de forma persistente, os modelos e cenas manipulados pela aplicação. A interface de `tDocument` oferece métodos para abrir, gravar e fechar a si próprio, bem como para adicionar e remover modelos e cenas no documento.

Um objeto da classe `tScene` representa uma cena de um documento da aplicação, uma coleção de atores, luzes e câmeras. A classe contém métodos para adicionar atores, luzes e câmeras em uma cena, bem como remover atores, luzes e câmeras de uma cena. Além disso, `tScene` oferece métodos para obter os iteradores de atores, luzes e câmeras, entre outros. Um ator de uma cena é um objeto da classe `tActor`. Uma luz da cena é um objeto da classe derivada da classe abstrata `tLight`. Existem dois tipos de luz em OSW: *luzes puntuais* (objetos da classe `tPunctualLight`) e *luzes direcionais* (objetos da classe `tDirectionaLight`). Uma câmera da cena é um objeto da classe `tCamera`. Classes derivadas de `tScene` devem sobrecarregar o método virtual `ConstructView()`, para construir suas vistas particulares.

Uma vista é um objeto de uma classe derivada da classe `tView`. Uma vista de uma cena tem dois objetivos: permitir ao usuário interagir com os componentes da cena e visualizar imagens da cena. Para o primeiro, a classe `tView` declara métodos através dos quais os usuários podem obter, interativamente, pontos, distâncias, ângulos e cores, entre outros, das janelas de vista, bem como selecionar atores, luzes e câmeras. Para o segundo, cada vista de OSW possui um objeto de uma classe derivada da classe `tRenderer`, responsável pela construção de imagens da cena (Capítulo 4). A funcionalidade de uma vista particular é definida por *comandos* de vista. Classes derivadas de `tView` podem definir seus próprios comandos. Classes derivadas podem, também, sobrecarregar o método virtual `ConstructRenderer()`, para construir seus próprios objetos de síntese de imagens.

As classes de janelas de OSW representam os elementos de interface homem-computador de uma aplicação de modelagem. As principais classes de janela de OSW são:

- `tMainWindow`. Um objeto da classe `tMainWindow` representa a janela principal da aplicação. A janela principal é a janela pai de todas as outras janelas da aplicação.
- `tViewWindow`. Um objeto da classe `tViewWindow` é uma janela-filha que contém uma vista da cena. A funcionalidade de uma janela de vista é definida por comandos de um objeto de uma classe derivada da classe `tView`.
- `tCommandWindow`. A janela de comandos é a janela da aplicação na qual os usuários digitam comandos e seus parâmetros. A janela de comandos é um objeto da classe `tCommandWindow`.

Um guia completo para utilização de OSW na implementação de algoritmos de modelagem pode ser encontrado em [23].

5.2.1 BEG - *Boundary Element Generator*

BEG - *Boundary Element Generator* é a aplicação responsável pela geração dos modelos de análise de sólidos elásticos. A aplicação é um objeto da classe `tMecApp`, derivada de `tApplication`. O documento da aplicação é um objeto da classe `tMecDoc`, derivada de `tDocument`. O documento mantém somente uma cena, um objeto da classe `tMecScene`, derivada da classe `tScene`. As vistas da cena da BEG são objetos da classe `tMecView`, derivada da classe `tMeshView`, a qual, por sua vez é derivada virtualmente de `tView`. A classe acessória `tMeshView` é uma classe especial para a construção de vistas de modelos de decomposição por células [22].

A BEG é constituída de uma janela principal, que contém quatro janelas-filha. Cada janela-filha exibe uma vista da cena, tomada por uma câmera virtual (objeto da classe acessória `tCamera`) cujos parâmetros podem ser alteradas pelo usuário, através de comandos herdados da classe `tView`. Como toda aplicação OSW, a BEG também possui janelas de comandos para iteração do usuário com a aplicação, através da digitação de comandos de vista e seus parâmetros. A interface da BEG é mostrada na Figura 5.2.

Com a utilização da BEG, é possível gerar malhas de elementos de contorno triangulares e células internas tetraédricas na forma de paralelepípedos, cilindros, domos e esferas. Nos modelos criados podem ser especificadas as condições de contorno, deslocamentos ou forças de superfície prescritas. A aplicação gera um arquivo de saída, com extensão `.ent`, o qual contém uma descrição completa do modelo criado. Esse arquivo é lido pelo analisador MEC, que realiza a análise e determina os valores de deslocamentos e forças de superfície e tensões visualizados na BEV. Nas seções seguintes, faremos referências a alguns dos métodos da aplicação, mas sem entrar em detalhes da implementação. Uma descrição pormenorizada da BEG (e da BEV), mereceria um texto exclusivo, o que não é o objetivo deste trabalho.

5.2.2 BEV - *Boundary Element Viewer*

BEV - *Boundary Element Visualization* é a aplicação responsável pela visualização dos resultados da análise dos modelos de sólidos gerados pelo analisador MEC. A aplicação BEV é um objeto da classe `tResApp`, derivada de `tApplication`. O documento da aplicação é um objeto da classe `tResDoc`, derivada de `tDocument`. O documento mantém duas cenas, uma para visualização de dados de contorno e outra para visualização de dados de domínio. ambas derivadas de `tResScene`, a qual, por sua vez, é derivada de `tScene`. A primeira é um objeto da classe `tBoundaryScene`; a segunda é um objeto da clase `tDomainScene`.

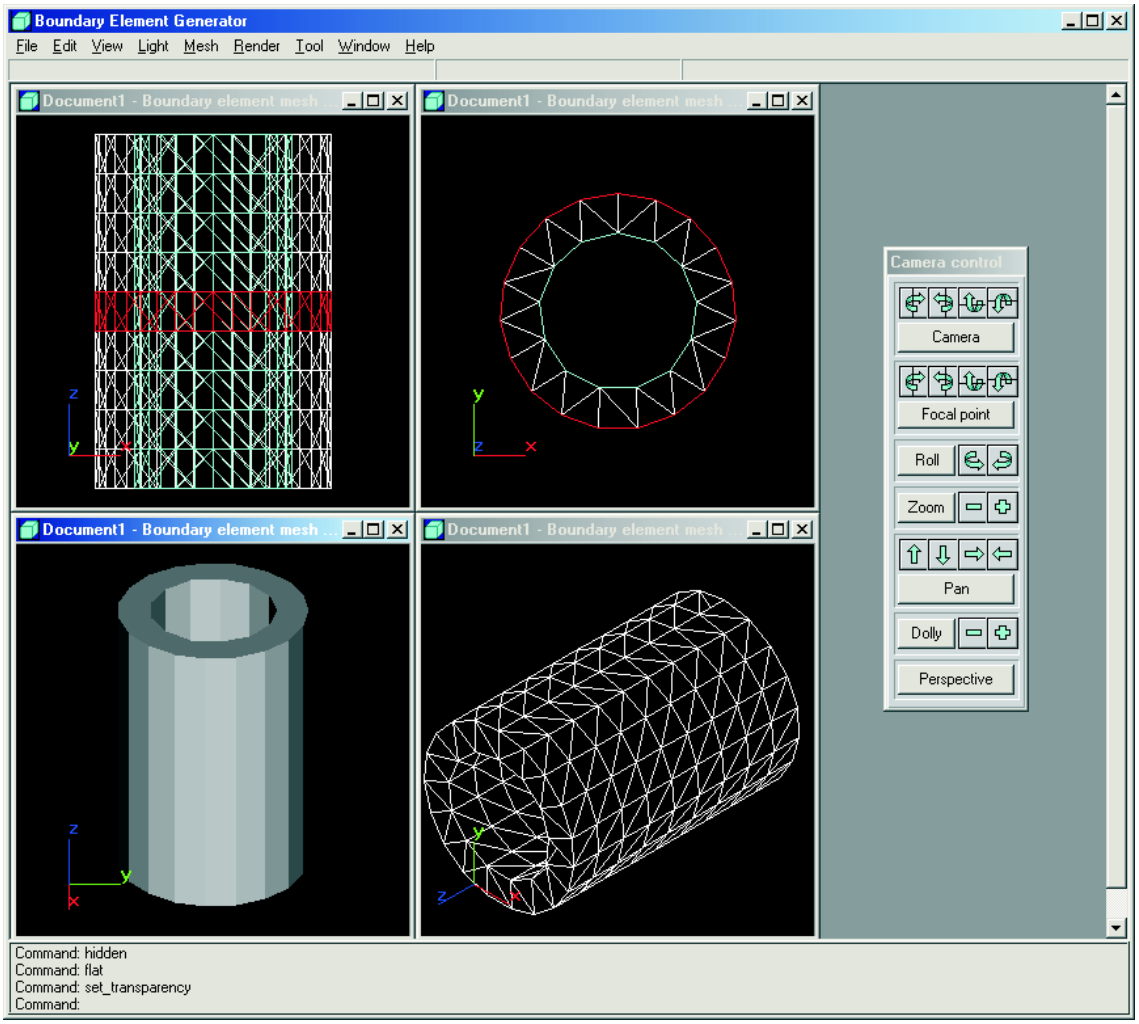

Figura 5.2: Interface da BEG.

Uma vista da BEV é um objeto cuja classe deriva de `tResView`. A classe é derivada de `tColorMapView`, a qual, por sua vez, é derivada virtualmente de `tView`. A classe acessória `tColorMapView` é uma classe especial para a construção de vistas para a visualização de atores que utilizam mapa de cores [22]. Há duas classes de vistas derivadas de `tResView`: `tBoundaryResView`, responsável pela interação com a cena para visualização de dados de contorno, e `tDomainView`, responsável pela interação com a cena para visualização de dados de domínio.

A BEV possui uma janela principal, que contém duas janelas-filha. Cada janela-filha representa uma vista da cena, uma vista de contorno e uma vista de domínio. Assim como a BEG, a BEV possui janelas de comandos, para iteração do usuário com a aplicação, através da digitação de comandos e parâmetros. A interface da BEV é mostrada na Figura 5.3.

Para gerar as imagens dos modelos de sólidos, a BEV faz a leitura de um arquivo, com extensão `.sol`, resultante do processo de análise do modelo pelo analisador MEC. Esse arquivo contém todas as informações de geometria e topologia do modelo, bem como os valores de deslocamentos e forças de superfície aplicados sobre o modelo. Com a utilização da BEV, é possível a visualização dos resultados da análise através da aplicação dos algoritmos de visualização descritos no Capítulo 4, bem como, a visualização dos valores em seções do modelo, com a utilização de superfícies de corte.

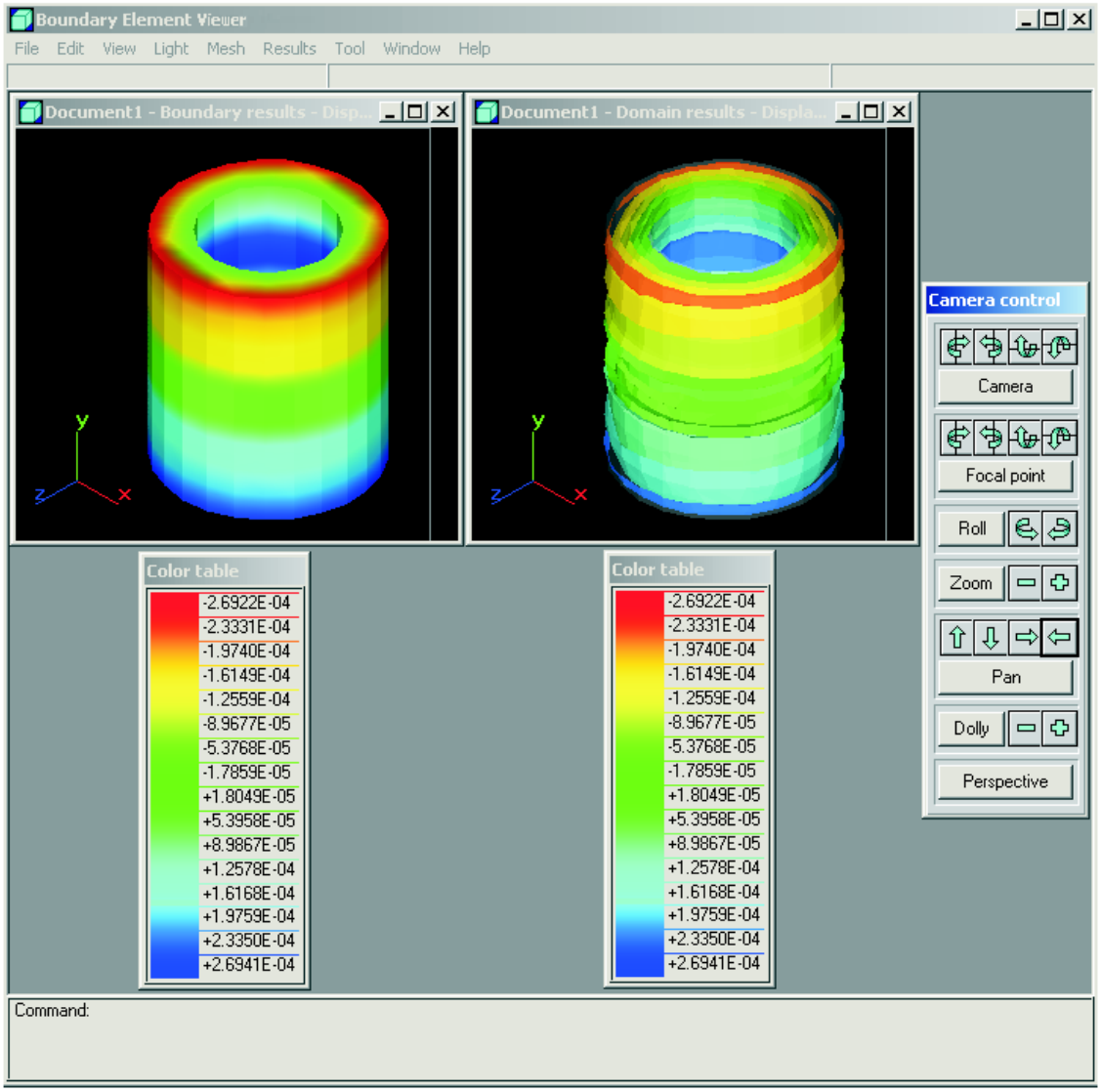

Figura 5.3: Interface da BEV.

5.3 Discretização do Domínio do Sólido

O primeiro passo a ser executado para obtermos a visualização de sólidos pelo MEC, é a discretização do domínio Ω considerado em células de domínio do tipo tetraedros. Os nós da malha de elementos de volume decorrentes desta discretização serão, então, nós sobre o contorno Γ e, também, nós internos no domínio Ω.

Para realizar a discretização do domínio, gostaríamos de utilizar as técnicas de modelagem geométrica e o filtro de geração de malhas a serem desenvolvidos em [19]. Como essas técnicas e o filtro ainda não estavam disponíveis, utilizamos outra técnica para gerar a malha de elementos de volume, a *triangulação de Delaunay*, a qual apresentamos a seguir.

A fim de ilustrarmos o processo de triangulação de Delaunay, iremos considerar um conjunto finito de pontos, ou vértices, $V = \{v_0, v_1, v_2, \ldots, v_n\}$, $v_i \in \mathbb{R}^2$, $0 \leq i \leq n$. Uma *triangulação* de V é um conjunto de triângulos $T = \{t_0, t_1, t_2, \ldots, t_m\}$ tal que

(i) todo ponto $v_i \in V$, $0 \leq i \leq n$, é um vértice de um triângulo $t_k \in T$, $0 \leq k \leq m$;

(ii) para todo $k \neq l$, $0 \leq k, l \leq m$, $\text{interior}(t_k) \bigcap \text{interior}(t_l) = \emptyset$; e

(iii) $\bigcup_{k=0}^{m} t_k = \text{envolv}(V)$,

onde $\text{interior}(t_k)$ denota o interior do k-ésimo triângulo de T e $\text{envolv}(V)$ denota a *envoltória convexa* de V. (A envoltória convexa de um conjunto V de pontos é o menor polígono convexo P para o qual cada ponto em V está ou no contorno ou no interior de P.)

Sejam v_i e v_j, $0 \leq i, j \leq n$, dois vértices quaisquer de V, e v_{ij} a aresta formada pelos vértices i e j. Um círculo circunscrito C da aresta v_{ij} é qualquer círculo que passa por v_i e por v_j (a aresta v_{ij} possui um número infinito de círculos circunscritos). Um círculo circunscrito C de v_{ij} é dito *vazio* se não contiver qualquer vértice de V em seu interior (os vértices v_i e v_j estão *sobre* C). Uma *triangulação de Delaunay* [29] de V é um grafo D, formado pelos vértices de V, tal que para toda aresta v_{ij} de D existe um círculo circunscrito vazio de v_{ij}. Toda aresta v_{ij} que satisfaz tal propriedade é dita de *Delaunay*.

SHEWCHUK [29] prova que o conjunto de arestas de Delaunay de um conjunto de vértices V forma uma triangulação de V. A prova é baseada no conceito de *triângulo de Delaunay*: um triângulo t_k de T é dito de Delaunay se e somente se o círculo circunscrito passando pelos três vértices de t_k é vazio. Tal característica de triângulos de Delaunay é chamada *propriedade do círculo circunscrito vazio*, ilustrada na Figura 5.4. A partir desta propriedade pode-se mostrar que, se todos os triângulos de T forem de Delaunay, então todas as arestas de T são de Delaunay, e vice-versa [29].

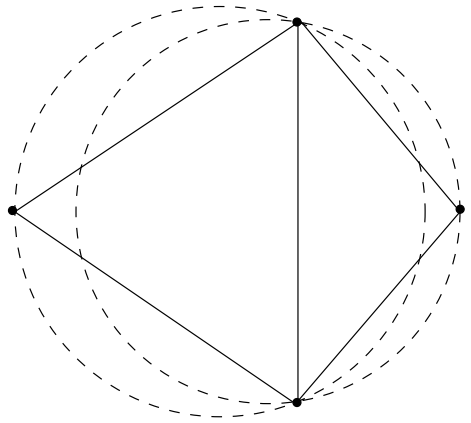

Figura 5.4: Triangulação de Delaunay.

A triangulação de Delaunay de um conjunto de pontos $V \in \mathbb{R}^d$, para $d \geq 2$, é uma generalização da triangulação de Delaunay bidimensional. Uma *triangulação* de V é um conjunto de d-simplexos (por exemplo tetraedros para $d = 3$) $T = \{s_0, s_1, s_2, \ldots, s_m\}$ tal que

(i) todo ponto $v_i \in V$, $0 \leq i \leq n$, é um vértice de um d-simplexo $s_k \in T$, $0 \leq k \leq m$;

(ii) para todo $k \neq l$, $0 \leq k, l \leq m$, $\text{interior}(s_k) \bigcap \text{interior}(s_l) = \emptyset$; e

(iii) $\bigcup_{k=0}^{m} s_k = \text{envolv}(V)$.

Seja s um k-simplexo (para qualquer k) cujos vértices estão em V. Seja S uma esfera d-dimensional em $\mathbb{R}^d$; S é uma *esfera circunscrita* de s se S incide sobre todos os vértices de s. Se $s = d$ então s tem uma única esfera circunscrita; senão s tem infinitas esferas circunscritas. O simplexo s é dito ser de Delaunay se e somente se existe uma esfera circunscrita *vazia* de

s. O simplexo é dito *fortemente* de Delaunay se existe uma esfera circunscrita S de s tal que nenhum vértice de V esteja no interior ou no contorno de S, exceto os vértices de s.

5.3.1 O Algoritmo de Inserção

Há vários algoritmos para geração da triangulação de Delaunay. Para nossos propósitos adotamos o algoritmo sugerido por SCHROEDER [26], chamado de chamado de *algoritmo de inserção*. O algoritmo se baseia na inserção incremental de pontos em uma triangulação de Delaunay pré-existente.

A computação fundamental do algoritmo de inserção é baseada no teste da hiperesfera: o teste da esfera $TE(p_i)$ de um ponto p_i para um simplexo s_i^n, com circuncentro x_c^n e raio r_i é

$$TE(p_i) = \begin{cases} \text{“}in\text{”} & \text{se } ||p_i - x_c^n|| \leq r_i \\ \text{“}out\text{”} & \text{caso contrário.} \end{cases} \tag{5.1}$$

O teste da hiperesfera determina quando um ponto pode ser considerado dentro (“*in*”) ou fora (“*out*”) do contorno da hiperesfera circunscrita c_i^n.

A seguir, descrevemos os principais passos do algoritmo de inserção.

Passo 1 Introduzir pontos P_B para formar uma triangulação inicial de Delaunay, que contorna o conjunto de pontos P.

Passo 2 Selecionar um ponto p_i de P e inserí-lo na triangulação. Os simplexos s_i^n de T_D^n, cujo contorno da hiperesfera contém o ponto, serão removidos. Isso forma um poliedro de inserção n-dimensional $Q(p_i)$. Uma nova triangulação é, então formada, através da ligação dos pontos do poliedro de inserção com o ponto p_i. O processo continua para os pontos restantes de P.

Passo 3 Ao final do passo anterior, todos os pontos terão sido inseridos na triangulação. Então, os simplexos conectando os pontos P_B podem ser removidos, resultando na triangulação final. Esse processo pode resultar em uma triangulação não convexa, cujo tratamento é detalhado em [26].

Para um conjunto de pontos aleatórios, a triangulação de Delaunay é única. No entanto, quando computações com precisão aritmética finita são realizadas ou quando conjuntos de pontos não aleatórios são utilizados, casos *degenerados* podem ocorrer. A solução de casos degenerados requer uma decisão de qual c_i^n de um ponto particular deve ser considerada como dentro ou fora do ponto. A escolha deve ser feita de modo a preservar as propriedades da triangulação de Delaunay, vistas na Seção 5.3.

Na prática, para verificarmos se um ponto p_i está dentro de uma circunsfera c_i^n, de raio r e circuncentro x_c, pode ser feita baseada em um *fator de tolerância* ϵ:

$$p_i \in c_i^n \text{ quando } ||p_1 - x_c|| \leq (1 - \epsilon)r. \tag{5.2}$$

Escolher $\epsilon > 0$ é equivalente a classificar p_i fora de todo c_i^n degenerado, enquanto que a escolha de $\epsilon < 0$ classifica p_i dentro de todo c_i^n degenerado. Geralmente, ϵ é tomado como $\epsilon > 0$, por exemplo, $\epsilon = 0.001$. Isso minimiza a remoção de simplexos e provê um mecanismo simples para controlar triangulação de malhas por ordenação de pontos. Maiores detalhes pode ser vistos em [26].

5.3.2 Implementação da Triangulação de Delaunay

A implementação da triangulação de Delaunay foi feita seguindo os passos do algoritmo de inserção, descritos anteriormente. A classe `t3DDelaunay`, Programa 5.1, é uma instância da classe `tMeshSource`, sendo, portanto, um fonte de dados.

```
1    class t3DDelaunay: public tMeshSource
2    {
3       public:
4          void SetInputPoints(tPoints*);
5          void Execute();
6          ...
7       private:
8          void Run();
9          void InitPointInsertion();
10         void InsertTetra(tTetrahedron*);
11         void InsertPoint(const t3DVector&);
12         tTriangles* FindEnclosingFaces(const t3DVector&);
13         tTetrahedron* FindTetra(const t3DVector&);
14         bool InSphere(const t3DVector&, tTetrahedron*);
15         ...
16   }; // t3DDelaunay
```

Programa 5.1: Parte da classe `t3DDelaunay`.

A triangulação inicia com a chamada ao método `Execute()`, declarado na linha 5 e herdado da classe base `tMeshSource`. Após construir uma instância de `tMesh`, a saída do filtro, o método envia a mensagem `Run()` ao filtro, Programa 5.2, o qual executa os três passos do algoritmo de inserção. Os pontos a serem inseridos na triangulação são dados pelo método `SetInputPoints(points)`, linha 4 do Programa 5.1, onde `points` é um ponteiro para um objeto da classe acessória `tPoints`. (`tPoints` representa uma lista encadeada de pontos 3D).

```
1    void
2    t3DDelaunay::Run()
3    {
4       InitPointInsertion();
5       for (tPointIterator pit = InputPoints->GetPointIterator(); pit; pit++)
6          InsertPoint(*pit.Current());
7       EndPointInsertion();
8    }
```

Programa 5.2: Parte do método `t3DDelaunay::Run()`.

A execução do primeiro passo do algoritmo, Programa 5.2, se dá com a chamada ao método `InitPointInsertion()`, linha 4, o qual cria a triangulação inicial, definida por um octaedro formado por seis pontos e quatro tetraedros. Criada a triangulação inicial, o próximo passo do algoritmo realiza a inserção de todos os pontos do conjunto dentro da triangulação, linhas 5 e 6, fazendo a remoção de simplexos de acordo com o critério da hiperesfera contida. Essa tarefa é executada pelo método `InsertPoint()`, linha 6.

O método `InsertPoint()` irá inserir cada ponto p_i na triangulação, desde que esse satisfaça o critério de Delaunay. Isso é verificado pelo método `FindEnclosingFaces()`,

que retorna uma lista de faces que encapsulam o ponto p_i, de acordo com o critério da hiperesfera contida. O teste da esfera é feito pelo método `InSphere()`. As faces retornadas pelo método `FindEnclosingFaces()` formarão uma nova triangulação, juntamente com o ponto p_i. O terceiro e último passo do algoritmo é executado pela chamada ao método `EndPointInsertion()`, linha 7, onde os pontos do octaedro inicial são removidos, resultando na triangulação final. O surgimento de casos degenerados foi tratado utilizando-se o fator de tolerância $\epsilon = 0.001$.

5.3.3 Geração da Malha de Elementos de Volume

A criação da malha de elementos de volume é realizada em três passos. No primeiro passo, os pontos a serem inseridos na triangulação são gerados e armazenados em um objeto da classe `tPoints`. No segundo passo, os pontos são fornecidos a um objeto da classe `t3DDelaunay`, o qual gerará a malha de tetraedros, cujo contorno é a envoltória convexa do conjunto de pontos criados no primeiro passo. O terceiro passo consiste na classificação e eliminação dos tetraedros que não pertencem ao volume. Por exemplo, tetraedros na cavidade de um cilindro vazado devem ser eliminados da malha final. Para ilustrar o processo, considere o método `tMecView::Extrude()`, Programa 5.3, o qual retorna um ponteiro para uma malha. O método toma como argumentos um ponteiro para uma casca, uma altura, o tamanho do tetraedro desejado e um ponteiro para um objeto da classe acessória `tMeshClassifier`.

```
1    tMesh*
2    tMecView::Extrude(tShell* shell, double height, double size,
3       tMeshClassifier* mc)
4    {
5       t2DMeshPointGenerator mg;
6
7       mg.SetInput(shell);
8       mg.SetElementSize(size);
9       mg.Execute();
10      if (mg.GetOutput())
11      {
12         tMesh* mesh = mg.GetOutput();
13         tPoints points;
14         int nh = nint(height / size);
15         t3DVector dh = Camera->GetViewPlaneNormal() * (height / nh);
16
17         for (tNodeIterator nit(mesh->GetNodeIterator()); nit;)
18         {
19            t3DVector p = nit++->Position;
20
21            for (int i = 0; i < nh; i++)
22               points.Insert(p + dh * i);
23         }
24         if ((mesh = Triangulate(&points, mc)) != 0)
25            return mesh;
26      }
27      return 0;
28   }
```

Programa 5.3: Método `tMecView::Extrude()`.

O método cria uma malha através de varredura translacional da casca `shell` em uma direção dada pelo vetor normal da câmera da vista e altura `height`. Inicialmente, o método cria o conjunto de pontos correspondente à "fatia" da base da malha, usando para isso um filtro da classe `t2DMeshPointGenerator`. Esse filtro toma como entrada uma casca e gera como saída uma malha correspondente a triangulação 2D da casca, linhas 7 a 15. Os pontos das demais "fatias" são determinados pela translação dos pontos da fatia inicial, linhas 17 a 23. A malha final de tetraedros é gerada com uma chamada ao método `tMecView::Triangulate()`, Programa 5.4.

```
1    tMesh*
2    tMecView::Triangulate(tPoints* points, tMeshClassifier* mc)
3    {
4       t3DDelaunay df;
5
6       df.SetInputPoints(points);
7       df.Execute();
8
9       tMesh* mesh = df.GetOutput();
10      ...
11   }
```

Programa 5.4: Método `tMecView::Triangulate()`.

Na linha 4 construímos um objeto da classe `t3DDelaunay`, um processo fonte, que gera malhas de elementos tetraédricos. Na linha 6, enviamos a mensagem `SetInputPoints()` ao objeto `df`, que ajusta os pontos armazenados em `points` para a entrada do processo. Na linha 7, enviamos a mensagem `Execute()` ao objeto `cf`. O método `Execute()` da classe `t3DDelaunay` faz uma chamada ao método `t3DDelaunay::Run()`, que realiza a triangulação de Delaunay, conforme visto na Seção 5.3.2. Na linha 9, ajustamos a saída do processo para a variável `mesh`, um ponteiro para a malha de elementos de volume gerada.

5.4 Geração da Malha de Elementos de Contorno

Após termos realizado a discretização do domínio considerado em células tetraédricas, temos uma malha de elementos de volume, cujos nós pertencem tanto ao contorno quanto ao volume do domínio. O próximo passo é gerar os elementos de contorno da malha. Para isso, desenvolvemos uma classe base de filtros de geração de malhas de elementos de contorno triangulares chamada `tBoundaryCellsFilter`, cuja definição parcial pode ser vista no Programa 5.5.

Um objeto da classe `tBoundaryCellsFilter` derivada da classe `tMeshToMeshFilter` toma como entrada um modelo de decomposição por células e produz como saída, outro modelo de decomposição por células contendo as células de contorno das células do modelo de entrada (opcionalmente, as células do modelo de entrada podem ser clonadas na malha de saída). Assim, a saída do filtro conterá todas as células de dimensão topológica 2 e dimensão topológica 1 correspondentes ao contorno das células de dimensão topológica 3 e 2 da malha de entrada, respectivamente.

No Programa 5.5, somente os métodos de extração do contorno bidimensional são listados. `Extract2DBoundary(cell)`, linha 4, identifica todas as faces de contorno da célula volumétrica `cell` e, para tais faces, constrói uma célula 2D correspondente, através do método `Make2DCell(n)`, linha 5, onde `n` é o número de nós. (Uma face de contorno é aquela na qual incide somente uma célula volumétrica). `tBoundaryCellsFilter::Make2DCell(n)`

```
1    class tBoundaryCellsFilter: public tMeshToMeshFilter
2    {
3       protected:
4          virtual void Extract2DBoundary(t3DCell*);
5          virtual t2DCell* Make2DCell(int) const;
6          tMesh* ConstructOutput() const;
7
8       private:
9          void Run();
10         ...
11   }; // tBoundaryCellsFilter
```

Programa 5.5: Parte da classe `tBoundaryCellsFilter`.

constrói um objeto da classe `t3N2DCell` para `n = 3`. No entanto, estamos interessados na geração de malhas de elementos de contorno triangulares.

Uma vez gerada a malha de elementos de contorno triangulares e células internas tetraédricas, antes de analisarmos o modelo, devemos impor as condições de contorno, ou seja, deslocamentos ou forças de superfície prescritas. A classe `tMecView` da BEG define um conjunto de comandos que permite ao usuário selecionar elementos de contorno nos quais deslocamentos serão restringidos ou forças de superfície serão aplicadas. A Figura 5.5 ilustra a execução do comando `Fix()` de `tMecView` durante a seleção dos elementos de contorno restringidos através de um cursor em forma de cursor elástico (a). Em (b) podemos ver em vermelho, os elementos de contorno que foram restringidos. Não discutiremos aqui a implementação desses comandos, pois envolvem a descrição de classes da OCL que estão fora do escopo desse texto.

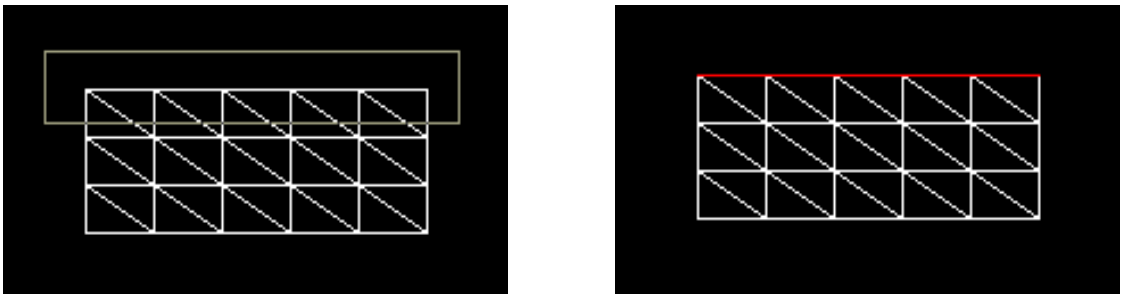

Figura 5.5: Execução do comando `tMecView::Fix()`.

Após a especificação das condições de contorno, a malha a ser submetida ao analisador `tSolidBESolver` será filtrada por um objeto da classe `tMultiNodesFilter`. Esse filtro toma como entrada uma `tBEMesh` e produz como saída uma `tBEMesh` com nós múltiplos, necessários para representação de descontinuidades de forças de superfície em nós de contorno, conforme visto no Capítulo 3.

5.5 Visualização de Dados

No Capítulo 4, descrevemos as classes da BEVL envolvidas na implementação dos algoritmos de visualização. Nesta seção, ilustraremos a utilização dos objetos da BEVL para visuali-

zação dos dados escalares (mapa de cores e contornamento) e vetoriais (deformação e ícones orientados) de contorno e de domínio, obtidos através da aplicação do MEC nos modelos considerados. Os dados de domínio serão visualizados em seções definidas por superfícies de corte geradas através do método de incisão de dados, descrito na Seção 4.4.

5.5.1 Visualizando Dados Escalares

As classes desenvolvidas na BEVL permitem a visualização de dados escalares através de duas técnicas, mapa de cores e contornamento. A seguir, descrevemos como utilizar os objetos das classes da BEVL, para visualizarmos os dados escalares de contorno e de domínio utilizando essas duas técnicas.

Visualizando Mapa de Cores

Mapeamento de cores é uma técnica de visualização de dados escalares, que mapeia dados escalares em cores (Capítulo 4). O processo de geração do mapa de cores ocorre como exemplificado com o método `tResView::CmColorMap()` da BEG, Programa 5.6.

```
1    void
2    tResView::CmColorMap()
3    {
4       ColorMapFlag ^= true;
5       Invalidate(false);
6    }
```

Programa 5.6: Visualizando mapa de cores.

O mapa de cores é gerado pelo *scanner* da vista e controlado por uma *flag*, denominada `ColorMapFlag`. O método `CmColorMap()`, ajusta `ColorMapFlag` para **`true`**, indicando que o mapa de cores está ativo. O *scanner*, então, gera uma imagem com o mapa de cores, aplicando uma tonalização no modelo de acordo com Gouraud [22]. Isso é feito através da interpolação bilinear das cores dos vértices dos polígonos do modelo a ser tonalizado (os polígonos de um modelo são extraídos com a solicitação de um iterador de faces do modelo, objeto da classe `tFaceIterator`, discutida no Capítulo 2).

O Programa 5.7 mostra parte da implementação do método `tResView::Paint()` da BEV, responsável pela geração de imagens da malha de contorno. Para cada ator da cena, o mapeador do ator, objeto da classe `tMapper`, é obtido na linha 6. Se o modelo associado ao mapeador for a malha de elementos de contorno, a mensagem `UseScalars(ColorMapFlag)` é enviada ao mapeador, linha 9. (A *flag* `ColorMapFlag` é ajustada no método `tResView::CmColorMap()`, Programa 5.6). Isso indica que o mapeador deve utilizar, em função de `ColorMapFlag`, ou os escalares dos vértices ou o material do modelo para determinar as cores da imagem a ser gerada. Quando a *flag* do mapa de cores não está ativa (`ColorMapFlag` é **`false`**), as cores utilizadas na geração da imagem são determinadas em função do material de cada polígono do modelo.

Visualizando Isolinhas e Isosuperfícies

Como vimos no Capítulo 4, a técnica de contornamento é implementada com a utilização de um filtro, um objeto da classe `tContourFilter`. O método `tResView::Contour()`, listado no Programa 5.8, ilustra como utilizamos as classes da BEVL para visualizarmos isolinhas ou isosuperfícies em um modelo.

```
1    void
2    tResView::Paint(TDC& dc, bool erase, TRect& rect)
3    {
4       for (tActorIterator ait(Scene->GetActorIterator()); ait; ++ait)
5       {
6          tMapper* mapper = ait.Current()->GetMapper();
7
8          if (mapper->GetModel() == GetScene()->GetMesh())
9             mapper->UseScalars(ColorMapFlag);
10         mapper->SetLookupTable(ColorMapFrame->GetLookupTable());
11      }
12      ((tScanner*)Renderer)->SetRenderMode(tScanner::Gouraud);
13      ...
14   }
```

Programa 5.7: Método `tResView::Paint()` da BEV.

```
1    void
2    tResView::Contour()
3    {
4       tContourFilter cf;
5       tModel* model = GetScene()->GetCutSurface();
6
7       if (!model)
8          model = GetScene()->GetMesh();
9
10      cf.GenerateValues(ContourParms.Levels, ContourParms.MinValue,
11         ContourParms.MaxValue);
12      cf.SetInput(model);
13      cf.Execute();
14      GetScene()->SetContour(cf.GetOutput());
15   }
```

Programa 5.8: Método `tResView::Contour()` da BEV.

Antes de detalharmos a implementação do método `Contour()`, é importante esclarecer como estamos utilizando o método na BEV. Na visualização de dados de contorno, `Contour()` atua sobre um modelo de decomposição por células, considerando apenas as células de contorno, gerando isolinhas. Na visualização de dados de domínio, duas situações podem ocorrer. Na primeira, podemos visualizar os dados de domínio considerando o modelo de decomposição por células; nesse caso a execução do método `Contour()` resultará na visualização de isosuperfícies. Na segunda, podemos visualizar os dados de domínio apenas em uma seção do modelo, definida por uma superfície de corte. Nesse caso, o método `Contour()` considera apenas a superfície, gerando isolinhas. Em resumo, se houver uma superfície de corte no modelo de decomposição por células, o método tomará como modelo a ser considerado a superfície de corte; caso contrário, o método considera sempre o modelo de decomposição por células.

Na linha 4 do Programa 5.8, construímos um objeto da classe `tContourFilter`. Na linha 5 executamos `tResDoc::GetCutSurface()`. A função obtém um ponteiro para o modelo gráfico que representa as superfícies de corte da malha da cena da vista. Se a cena não "possui" superfícies de corte, o método `tResDoc::GetCutSurface()` retorna um ponteiro nulo.

Por isso, na linha 7, testamos se a obtenção do modelo foi bem sucedida. Se não tivermos um modelo gráfico, ajustamos a malha da cena da vista como sendo o modelo considerado, através da execução de `tResDoc::GetMesh()`, linha 8. Neste caso não há a necessidade de testar se a obtenção do modelo foi bem sucedida, pois sempre haverá uma malha sendo visualizada. A execução do filtro começa na linha 10, com o envio da mensagem `GenerateValues()` ao objeto `cf`. O método `GenerateValues()` ajusta os parâmetros utilizados pelo contornamento: o número de níveis de isolinhas ou isosuperfícies a serem gerados e os valores mínimo e máximo dos dados escalares considerados. Na linha 12, enviamos a mensagem `SetInput()` ao objeto `cf`, que ajusta o modelo da vista da cena como o modelo de entrada do filtro. Na linha 13, enviamos a mensagem `Execute()` ao objeto `cf`. O método `Execute()` da classe `tContourFilter` faz uma chamada ao método `Run()`, o qual envia uma mensagem ao modelo solicitando um iterador de células, no caso de um modelo de decomposição por células. De posse do iterador, o filtro envia para cada célula do modelo a mensagem `Contour()`. No caso de um modelo gráfico, o filtro envia a mensagem `Contour()` ao próprio modelo, para que este se contorne. Na linha 14, ajustamos a saída do filtro para a vista da cena, o modelo gráfico contendo as isolinhas ou isosuperfícies geradas.

5.5.2 Visualizando Dados Vetoriais

As classes desenvolvidas na BEVL permitem a visualização de dados vetoriais através de duas técnicas, deformação e ícones orientados. A seguir, descrevemos como utilizar os objetos das classes da BEVL, para visualizarmos os dados vetoriais de contorno e de domínio utilizando essas duas técnicas.

Visualizando Deformação

A técnica de deformação trabalha com os vértices de um modelo genérico, extraindo, de cada vértice, um vetor de interesse, o qual sofre uma transformação de escala, gerando um modelo deformado para visualização. O Programa 5.9 ilustra a execução do método `Warp()`, membro de `tResView`.

```
1    void
2    tResView::Warp(double scale)
3    {
4       tModel* model = GetScene()->GetCutSurface();
5       if (!model)
6          model = GetScene()->GetMesh();
7
8       tWarpFilter wf(scale);
9       wf.SetInput(model);
10      wf.Execute();
11      if (wf.GetOutput() != 0)
12         GetScene()->SetWarp(wf.GetOutput());
13
14      if (ContourFlag)
15         Contour();
16   }
```

Programa 5.9: Método `tResView::Warp()` da BEV.

Assim como ocorre na aplicação da técnica de contornamento, se houver uma superfície de corte no modelo de decomposição por células, o método tomará como modelo a ser considerado

a superfície de corte; caso contrário, o método considera sempre o modelo de decomposição por células. Na linha 4, executamos `tResDoc::GetCutSurface()`, que obtém um ponteiro para o modelo corrente da cena da vista, uma superfície de corte. Na linha 5, testamos se a obtenção do modelo foi bem sucedida. Se não tivermos uma superfície de corte, ajustamos o modelo de decomposição por células da cena da vista como sendo o modelo considerado, através da execução de `tResDoc::GetMesh()`, linha 6. Na linha 8, construímos um objeto da classe `tWarpFilter`. O argumento `scale` ajusta a escala que será aplicada na deformação do modelo. A execução do filtro inicia na linha 9, com o envio da mensagem `SetInput()` ao objeto `wf`, o qual ajusta o modelo da vista da cena como o modelo de entrada do filtro. Na linha 10, enviamos a mensagem `Execute()` ao objeto `wf`. O método `Execute()` da classe `tWarpFilter` faz uma chamada ao método `Run()` próprio do modelo, o qual envia uma mensagem ao modelo solicitando um iterador de vértices. Então, para cada vértice é extraído o vetor de interesse, o qual sofre uma transformação de escala determinada pelo atributo `scale`. Na linha 12, ajustamos a saída do filtro para a vista da cena, o modelo gráfico deformado.

Visualizando Ícones Orientados

Na BEVL, ícones são gerados por um processo fonte que toma como entrada um modelo genérico e gera, como saída, modelos gráficos na forma de cones, que representam alguma informação relevante do modelo de entrada (deslocamentos, por exemplo). Ilustramos a visualização de ícones orientados com a implementação do método `CmShowGlyphs()`, membro de `tResView`, listado no Programa 5.10.

Analogamente ao que já foi citado nas técnicas de contornamento e deformação, o método que implementa a visualização de ícones orientados considera a superfície de corte no modelo de decomposição por células, se esta existir; caso contrário, o método considera sempre o modelo de decomposição por células da cena da vista. A visualização de ícones orientados é controlada por uma *flag*, `GlyphsFlag`, a qual deve ser ajustada para **`true`**, para tornar os *glyphs* ativos. Na linha 4, verificamos o *status* de `GlyphsFlag`. Se for **`true`**, nas linhas 6 e 7, executamos, respectivamente, `tResDoc::GetMesh()` e `tResDoc::GetCutSurface()`, para obtermos o modelo da cena da vista, um modelo de decomposição por células, uma superfície de corte, ou ambos. Nas linhas 8 a 19 testamos o valor do argumento `id`, que contém um valor correspondente à opção de visualização selecionada pelo usuário no menu da aplicação. É possível visualizar ícones orientados representando deslocamentos ou forças de superfície aplicadas ao modelo. Dependendo da escolha do usuário, os métodos `ExtractDisplacements()`, linha 12, ou `ExtractTractions()`, linha 18, serão executados para extrair do modelo o vetor de interesse. Na linha 20, construímos um objeto da classe `t3DGlyph`. A execução do processo inicia na linha 22, com o envio da mensagem `SetInput()` ao objeto `gf`. O método `SetInput()` ajusta o modelo de entrada do processo, um modelo de decomposição por células ou uma superfície de corte. Na linha 23, enviamos ao objeto `gf` a mensagem `SetScaleFactor()`, que ajusta o fator de escala a ser aplicado na criação do ícone orientado. Como convenção, utilizamos o valor da escala aplicada na técnica de deformação, armazenado no atributo `WarpScale`. Na linha 24, enviamos a mensagem `Execute()` ao objeto `gf`. O método `Execute()` da classe `t3DGlyph` faz uma chamada ao método `Run()` próprio do modelo, o qual envia uma mensagem ao modelo solicitando um iterador de vértices, sobre os quais serão criados os cones, representando alguma grandeza de interesse (deslocamentos ou forças de superfície). Na linha 26, ajustamos a saída do processo para a vista da cena, os modelos gráficos em forma de cones.

```
1    void
2    tResView::CmShowGlyphs(uint id)
3    {
4       if (GlyphsFlag ^= true)
5       {
6          tMesh* mesh = GetScene()->GetMesh();
7          tGraphicModel* surface = GetScene()->GetCutSurface();
8          switch (id)
9          {
10            case 638:
11               if (!surface)
12                  ExtractDisplacements(mesh);
13               break;
14
15            case 639:
16               if (surface)
17                  return;
18               ExtractTractions(mesh);
19         }
20         t3DGlyph gf;
21
22         gf.SetInput(surface ?  (tModel*)surface :  (tModel*)mesh);
23         gf.SetScaleFactor(WarpScale);
24         gf.Execute();
25         if (gf.GetOutput() != 0)
26            GetScene()->SetGlyphs(gf.GetOutput());
27      }
28      ...
29   }
```

Programa 5.10: Método `tResView::CmShowGlyphs()` da BEV.

Gerando Superfícies de Corte

Uma superfície de corte, na BEVL, permite a visualização de valores de domínio interpolados na seção definida pela superfície. A técnica de incisão de dados, Capítulo 4, permite cortar um modelo de decomposição por células com uma superfície e mostrar esses valores. A técnica é implementada pelo comando `CmCutter()`, listado no Programa 5.11. O comando é membro de `tDomainView`, que representa a janela de vista da aplicação, onde os valores de domínio do modelo podem visualizados.

Na BEVL, uma superfície de corte é aplicada sempre em um modelo de decomposição por células. Na linha 4, executamos `tResDoc::GetMesh()`, para obtermos um ponteiro para o modelo de decomposição por células da cena da vista. Na linha 5, testamos se a obtenção do modelo foi bem sucedida. Se não tivermos um modelo, não temos como aplicar a superfície de corte. Na linha 8, construímos um objeto da classe `tImplicitFunction`, uma função implícita que representa a superfície a ser utilizada no "corte" do modelo, nesse caso um plano. O método `GetPlane()`, retorna um ponteiro para um plano a ser utilizado no corte (interativamente definido pelo usuário). Se a construção do plano foi bem sucedida, na linha 11, construímos um objeto da classe `tCutter`, um filtro, ajustado com a superfície implícita (plano) representada por `f`. Na linha 13, enviamos a mensagem `SetInput()` para o objeto `c`, o qual ajusta a entrada do filtro para o modelo de decomposição por células, cujo endereço

```
1    void
2    tDomainView::CmCutter()
3    {
4       tMesh* mesh = GetScene()->GetMesh();
5       if (!mesh)
6          return;
7
8       tImplicitFunction* f = GetPlane();
9       if (f)
10      {
11         tCutter c(*f);
12
13         c.SetInput(mesh);
14         c.Execute();
15         if ((c.GetOutput()) != 0)
16         {
17            GetScene()->SetCutSurface(c.GetOutput());
18            ColorMapFlag = true;
19            ...
20         }
21         ...
22      }
23   }
```

Programa 5.11: Método `tResView::CmCutter()` da BEV.

está armazenado em `mesh`. Na linha 14, enviamos a mensagem `Execute()` ao objeto `c`. O método `Execute()` da classe `tCutter` faz uma chamada ao método `Run()` próprio do modelo, o qual envia uma mensagem ao modelo solicitando um iterador de nós, sobre os quais a função implícita será avaliada. Então, o filtro envia para cada célula do modelo a mensagem `Contour()`. Na linha 17, ajustamos a saída do processo para a vista da cena, a superfície de corte, em cuja seção, podem ser visualizados os valores de domínio, escalares e vetoriais.

5.6 Sumário

Nesse capítulo, descrevemos as etapas envolvidas na solução do problema de visualização de sólidos analisados pelo MEC.

Em um primeiro passo, discretizamos o domínio do modelo em células de domínio do tipo tetraedros utilizando a triangulação de Delaunay. Os nós da malha de elementos de volume resultante são nós sobre o contorno do modelo e, também, nós internos ao domínio do modelo.

Em um segundo passo, utilizamos um filtro de geração de malhas de elementos de contorno triangulares, que gera as células de contorno das células do modelo de entrada, a malha de elementos de volume criada no passo anterior. Os nós dos elementos de contorno são rotulados como nós de contorno, e os demais nós (do volume) são rotulados como nós internos.

Em um terceiro passo, submetemos nosso modelo para análise, utilizando um analisador MEC. Os resultados obtidos são valores de deslocamentos e forças de superfície em todos os nós de contorno. Conhecidos os valores nodais dos elementos de contorno, determinamos, por interpolação, os valores em quaisquer pontos do contorno do sólido.

No quarto e último passo, utilizamos os objetos da BEVL para visualização dos dados escalares (mapa de cores e contornamento) e vetoriais (deformação e ícones orientados) de contorno e de domínio obtidos. Os dados de domínio foram visualizados com a utilização de superfícies de corte geradas através do método de incisão de dados.

Para o cumprimento das etapas do problema de visualização proposto, realizamos as seguintes atividades:

- Criação da classe `t3DDelaunay`;
- Criação da classe `tBESolidFilter`;
- Criação da classe `tMultiNodesFilter`;
- Criação da classe `tMeshClassifier`;
- Criação da classe `tPoints`;
- Criação da classe `t2DMeshPointGenerator`.

CAPÍTULO 6

Exemplos de Visualização

6.1 Introdução

Nas páginas seguintes apresentamos figuras coloridas de alguns resultados obtidos pela utilização das classes de objetos da BEVL, para a modelagem, análise e visualização de modelos. As imagens foram capturadas diretamente das janelas de vistas gráficas da BEG e da BEV e impressas por um *software* de manipulação de imagens. A descrição dos exemplos que ilustram esse capítulo é apresentada a seguir. Procuramos gerar modelos que explorassem ao máximo os recursos da BEG e da BEV, porém encontramos dificuldades no processo de análise de alguns modelos, pois não tínhamos equipamento com memória suficiente para processar as informações desejadas.

6.2 Exemplos

- **Cubo com escalares de esfera.** As figuras coloridas 1 a 4 mostram imagens de um cubo cuja geometria foi definida como: centro $(0, 0, 0)$, orientação $(0, 0, 0)$ e escala $(1, 1, 1)$, com discretização $(1/10, 1/10, 1/10)$. Os valores escalares foram ajustados por uma esfera com centro igual ao centro do cubo.

- **Viga engastada em uma extremidade com carregamento vertical aplicado na extremidade livre.** As figuras coloridas 5 a 18 mostram imagens de uma viga cuja geometria é definida como: centro $(0, 0, 0)$, orientação $(0, 0, 0)$, escala $(0.2, 0.4, 1.6)$, com discretização $(1/2, 1/3, 1/10)$. O módulo de elasticidade é igual a 210.000 uf/ud^2 e coeficiente de Poisson igual a 0.3. A viga teve uma das extremidades fixa e foi aplicado um carregamento distribuído com vetor de carga $(0, -20, 0)$ na extremidade oposta (extremidade livre). As imagens de deformação utilizam escala 10.

- **Viga bi-apoiada com extremidades em balanço.** As figuras coloridas 19 a 23 mostram imagens de uma viga cuja geometria é definida como: centro $(0, 0, 0)$, orientação $(0, 0, 0)$, escala $(0.2, 0.4, 4)$ e discretização $(1/2, 1/3, 1/10)$. O módulo de elasticidade é igual a 210.000 uf/ud^2 e coeficiente de Poisson igual a 0.3. A viga foi fixada em dois pontos de sua base, ficando com as extremidades livres. Foi aplicado um carregamento horizontal e vertical ao longo da viga, com o vetor de carga distribuída $(-10, -20, 0)$. As imagens de deformação e ícones orientados utilizam escala 80.

- **Viga cilíndrica engastada em uma extremidade com carregamento vertical aplicado na extremidade livre.** As figuras coloridas 24 a 27 mostram imagens de uma viga cilíndrica, cuja geometria é definida como: centro $(0, 0, 0)$, raio 0.8, altura 2 e discretização 0.2. O módulo de elasticidade é igual a 210.000 uf/ud^2 e coeficiente de Poisson igual a 0.3. A viga foi fixada em uma das extremidades, ficando com a outra extremidades livre, onde foi aplicado um carregamento distribuído com vetor de carga $(20, 0, 0)$. As imagens de deformação e ícones orientados utilizam escala 40.

- **Cilindro vazado sob pressão interna.** As figuras coloridas 28 a 32 mostram imagens de um cilindro com a seguinte geometria: centro $(0, 0, 0)$, raio interno 0.5, raio externo 0.8, altura: 2 e discretização 0.2. O módulo de elasticidade é igual a 1.000 uf/ud^2 e coeficiente de Poisson igual a 0.3. O cilindro teve sua faixa central externa fixa e foi aplicada uma pressão de -1 em todas as células internas.

- **Esfera oca sob pressão interna, com apoio nos pólos.** As figuras coloridas 33 a 38 mostram imagens de uma esfera oca de geometria: centro $(0, 0, 0)$, raio interno 0.6, raio externo 0.8 e discretização 0.2. O módulo de elasticidade é igual a 1.000 uf/ud^2 e coeficiente de Poisson igual a 0.3. Foram fixadas as células externas nas faixas polares, sendo aplicada uma pressão de -1 em todas as células internas.

- **Esfera oca sob pressão interna com apoios na faixa central externa.** As figuras coloridas 39 a 42 mostram imagens de uma esfera oca com a seguinte geometria: centro $(0, 0, 0)$, raio interno 0.6, raio externo 0.8 e discretização 0.2. O módulo de elasticidade é igual a 1 uf/ud^2 e coeficiente de Poisson igual a 0.3. Foram fixadas as células externas na faixa central da esfera e aplicada uma pressão de -1 em todas as células internas. As imagens de deformação e ícones orientados utilizam escala 0.1.

- **Domo oco engastado na base, com carregamento vertical externo e pressão interna.** As figuras coloridas 43 a 49 mostram imagens de um domo oco com geometria: centro $(0, 0, 0)$, raio interno 0.6, raio externo 0.8 e discretização 0.2. O módulo de elasticidade é igual a 1.000 uf/ud^2 e coeficiente de Poisson igual a 0.3. Todas as células da base do domo foram fixadas e foi aplicada uma carga distribuída $(0, 0, -2)$ em células do topo do domo. Ainda, foi aplicada uma pressão de -1 em todas as células internas do domo.

CAPÍTULO 7

Conclusão

7.1 Discussão dos Resultados Obtidos

Neste trabalho, abordamos visualização de dados escalares e vetoriais sobre o contorno e no domínio de sólidos elásticos analisados pelo método dos elementos de contorno (MEC). No Capítulo 1 procuramos enfatizar que, em sistemas de mecânica computacional, visualização não é somente uma ferramenta conveniente porque permite uma compreensão mais imediata dos dados resultantes da análise numérica. Na verdade, visualização é parte essencial do sistema. Para comprovar isso, tomemos, do capítulo anterior, o exemplo do cilindro vazado submetido a uma pressão interna unitária mostrado nas Figuras Coloridas 28 a 32.

A malha do modelo de análise possui 892 elementos de contorno, 2.079 células internas e 684 nós, dos quais 540 são nós de contorno e 144 são nós internos. Considerando três componentes de deslocamento e três componentes de força de superfície em cada nó de contorno, e três componentes de deslocamento e seis componentes de tensão em cada nó interno, os dados resultantes da análise totalizam 4.536 números. Se impressos em formulários com 100 linhas por página, 6 números por linha, teríamos uma listagem com 8 páginas e muito trabalho para interpretação dos resultados. Observando a Figura Colorida 32, no entanto, imediatamente nos certificamos que o modelo parece se deformar como o esperado quando submetido a uma pressão interna, conforme nos mostram a estrutura deformada e os ícones de deslocamento. Isso pode ser notado sem que tenhamos que, em uma primeira etapa, interpretar quantitativamente os dados da análise. O fato da visualização ser essencial na compreensão dos números resultantes do processo de análise pode ser comprovado se considerarmos que, em comparação com casos práticos, onde os modelos possuem milhares ou até mesmo milhões de graus de liberdade, a quantidade de números no exemplo citado é ínfima (embora a análise tenha tomado duas horas de processamento quase exclusivo de um Pentium IV 1.8GHz com 256MB de memória).

Um dos problemas da visualização de dados resultantes de análise pelo MEC é que, não necessariamente, precisa haver uma discretização do domínio para que o método funcione, mesmo quando a integração de termos de domínio são requeridos. (Para certos tipos de forças de volume os termos de domínio podem ser transformados em termos de contorno [4].) Isso é justamente um dos atrativos do MEC em relação a outros métodos tais como o método dos elementos finitos (MEF): a redução da dimensionalidade do problema em 1, com conseqüente redução da quantidade de memória requerida para armazenamento do sistema de equações e do tempo de processamento. Respostas no domínio podem ser obtidas, então, em pontos

internos a partir dos valores determinados nos pontos de contorno. A diferença é que em pontos de contorno incidem elementos de contorno, os quais fornecem as funções espaciais necessárias à interpolação dos dados discretos resultantes da análise

Agora que temos tudo funcionando, a solução adotada para visualização de dados de domínio de um sólido parece ser óbvia: além de discretizar o contorno, discretizar também o domínio do sólido. As células internas, além de fornecerem os pontos internos e serem empregadas na computação dos termos de domínio, fornecem também as funções de interpolação de dados de domínio fundamentais para implementação de muitos dos algoritmos de visualização escalar e vetorial discutidos no texto.

Na verdade, não chegamos a considerar a implementação de outras possibilidades. A estratégia usada funciona até mesmo quando o modelo de análise não possui pontos internos, como no caso de sólidos esbeltos. Isso pode ser exemplificado pela esfera oca submetida à pressão interna mostrada na Figura Colorida 33. A malha possui 1.134 elementos de contorno, 1.659 células internas e 613 nós, todos de contorno. As Figuras Coloridas 35 a 38 ilustram a visualização de dados de domínio.

Uma suposta desvantagem da estratégia é que a discretização obrigatória do domínio anularia a economia de memória e velocidade de processamento em relação ao MEF, mas isso não é de todo verdadeiro. Diferente de elementos finitos de volume, células internas não possuem, por exemplo, matriz de rigidez local, nem para armazenar, nem para computar. A maior desvantagem, de fato, é que a discretização do sólido tem de ser efetuada, nos casos gerais, por processos automáticos de geração de malhas. Na indisponibilidade de um gerador de malhas 3D, desenvolvemos um para modelos simples, baseado na triangulação de Delaunay.

O objetivo geral do trabalho foi a implementação de uma biblioteca de classes denominada *Boundary Element Visualization Library*, ou BEVL, destinada ao desenvolvimento de aplicações orientadas a objetos de visualização de dados de contorno e de domínio resultantes da análise numérica elastostática de modelos de sólidos pelo MEC. A biblioteca é uma extensão de OSW — *Object Structural Workbench*, um *toolkit* para desenvolvimento de aplicações orientadas a objetos de modelagem, análise mecânica e visualização de modelos estruturais. Os objetivos específicos foram:

- *Estudar e apresentar os fundamentos computacionais e matemáticos utilizados para o desenvolvimento das classes de objetos da BEVL.* Esses estudos envolveram modelagem geométrica, uma introdução ao método dos elementos de contorno, visualização (principalmente), e programação orientada a objetos. Estudamos também as principais classes de objetos de OSW, utilizadas como fundação para implementação da BEVL. Ao longo do texto apresentamos os fundamentos estudados: no Capítulo 2, tratamos de modelagem geométrica; no Capítulo 3, dos princípios da formulação do MEC; e no Capítulo 4, dos algoritmos de visualização escalar e vetorial implementados na BEVL.

- *Descrever as técnicas desenvolvidas, bem como os atributos e métodos das classes de objetos da BEVL.* Do Capítulo 2 ao Capítulo 5, descrevemos em seus aspectos gerais, em seguida à teoria, as principais classes de objetos de OSW empregadas desde a modelagem até a visualização de modelos de sólidos analisados pelo MEC. Parte dessas classes são originais da OCL, muitas delas revisadas ao longo da implementação da BEVL. Os passos da solução proposta para visualização de dados de contorno e de domínio foram descritos em detalhes no Capítulo 5. Reconhecemos que uma descrição pormenorizada das classes de OSW que foram modificadas e das novas classes da BEVL é necessária, na forma de guia de referência. Esperamos, no entanto, que as descrições apresentadas tenham sido suficientes para o entendimento do trabalho realizado.

- *Exemplificar a utilização das classes de objetos da BEVL.* No Capítulo 5, mostramos como utilizar as principais classes da BEVL para a visualização de imagens de modelos de sólidos analisados pelo MEC, utilizando os algoritmos apresentados no Capítulo 4. Com a OCL e a BEVL, construímos duas aplicações, denominadas BEG e BEV. Com a primeira geramos malhas e especificamos condições de contorno de modelos de sólidos simples. Com a segunda, visualizamos dados escalares e vetoriais sobre o contorno e no domínio dos modelos de sólidos. O emprego dos conceitos de encapsulamento, herança e polimorfismo providos pela programação orientada a objetos possibilitou uma implementação mais eficiente (em termos de tempo de implementação e depuração) da BEG e da BEV, devido à reutilização de código das classes de objetos da OCL. Exemplos dos resultados obtidos foram mostrados nas figuras coloridas dos exemplos descritos no Capítulo 6.

Podemos afirmar, face aos resultados obtidos, que todos os objetivos propostos no trabalho foram plenamente atingidos.

7.2 Comparação com o VTK

Durante a implementação da BEVL, por vezes buscamos no VTK — *Visualization Toolkit*, desenvolvido por SCHROEDER e outros [28], sugestões para a solução de alguns problemas relacionados à visualização. A seguir, faremos algumas considerações a respeito das semelhanças e diferenças entre o VTK e OSW.

O VTK é um *toolkit* destinado especificamente à visualização, podendo ser aplicado na visualização de dados médicos, financeiros, ou resultantes de modelagem implícita, dinâmica de fluídos ou análise numérica pelo método dos elementos finitos, entre outros. O projeto das classes bases do VTK foi orientado nesse sentido, principalmente as classes de repositórios de dados, implementadas utilizando-se arranjos dinâmicos. Algumas vantagens de arranjos dinâmicos são: redução da fragmentação de memória ocasionada por sucessivas alocações e liberações dinâmicas de pequenos blocos de memória, e acesso a qualquer elemento em tempo constante. Não obstante, a expansão de um arranjo dinâmico, quando se fizer necessária, requer a duplicação dos elementos do arranjo. Devido à natureza dinâmica dos modelos geométricos apresentados no Capítulo 2 (não se conhece a priori o número de componentes de um modelo geométrico), preferiu-se utilizar listas lineares encadeadas e duplamente encadeadas para implementação de repositórios de dados em OSW.

Da mesma forma que o VTK, OSW também é destinado à visualização, compartilhando, por isso, conceitos e nomenclaturas em comum, tais como fontes, filtros e mapeadores. Além disso, algumas das técnicas de visualização utilizadas também são comuns: mapa de cores, contornamento, deformação, ícones orientados e incisão de dados. Ainda, algumas classes de objetos utilizados na BEVL foram inspirados em classes de objetos do VTK, como é o caso da classe `tPointLocator`. No entanto, OSW é destinado também à modelagem e à análise mecânica, e mais, ao desenvolvimento de aplicações de modelagem, análise e visualização de modelos estruturais. Isso conduziu a um projeto distinto de software.

Os modelos utilizados em OSW são modelos distintos daqueles empregados no VTK. VTK não tem modelos de sólidos, OSW sim. Não seria possível a geração de malhas de um modelo de sólidos facilmente utilizando a representação de dados do VTK. Os modelos do VTK assumem uma estrutura que consiste de células e pontos, onde as células especificam a topologia e os pontos a geometria do modelo. Em OSW, células são usadas somente em modelos de decomposição por células. Além disso, células OSW podem ser especializadas em elementos finitos ou de contorno. Diferente do VTK, OSW possui estruturas próprias para cada modelo considerado. Por exemplo, modelos de sólidos são baseados na estrutura de

dados semi-aresta de Mäntÿla, e modelos gráficos são representados por primitivos gráficos. Embora OSW e VTK possuam em comum alguns atributos utilizados em algoritmos de visualização, como escalares, normais e vetores, muitos atributos da visualização em OSW são resultantes do processo de análise. Esses atributos estão presentes nas células OSW (mais especificamente nos graus de liberdade de seus nós), mas não nas células VTK. Para utilizar OSW para modelagem e análise mecânica e o VTK para visualização seria necesário a construção de filtros que convertessem dados de uma representação para outra. No entanto, aplicações interativas baseadas em tais filtros poderiam resultar ineficientes.

Em relação aos recursos gráficos, o VTK utiliza a OpenGL para sintetizar suas imagens, transformando seus dados em primitivos da OpenGL. OSW possui seus próprios sintetizadores de imagens, os quais entendem diretamente a estrutura dos modelos OSW (por causa dos iteradores de componentes de modelos geométricos vistos no Capítulo 2). Os recursos da OpenGL tornam o VTK muito mais eficiente, nesse sentido, mas esses podem ser facilmente incorporados em OSW.

Em suma, embora muitos conceitos e algoritmos empregados em OSW para geração de imagens resultantes de uma aplicação de modelagem possam invocar similaridades com àqueles abordados no VTK, OSW engloba o VTK porque aborda também modelagem (não só geométrica, mas também mecânica) e análise. Os processos de geração e representação de dados geométricos e de análise são específicos o bastante em OSW para justificar o projeto de classes próprias de visualização, ao invés da utilização das do VTK.

7.3 Trabalhos Futuros

A primeira versão da BEVL permite a utilização da biblioteca no desenvolvimento de aplicações de dados escalares e vetoriais resultantes da análise elastostática de sólidos pelo MEC. Algumas sugestões para extensão da BEVL são dadas a seguir.

Inicialmente, outros algoritmos de visualização escalar e vetorial poderiam ser implementados. Por exemplo, para uma verificação quantitativa mais precisa dos dados resultantes de análise, seriam interessantes filtros geradores de gráficos Cartesianos bi e tridimensionais. Tais gráficos mostrariam valores, escalares ou vetoriais, obtidos em pontos definidos ao longo de curvas ou superfícies que interceptam o modelo de análise. Para isso, novas classes de objetos representando gráficos 2D e 3D deveriam ser criadas.

Outra possibilidade de extensão da BEVL é a consideração de algoritmos de visualização de tensores. Alguns exemplos, citados em [28], são *elipsóides de tensor* e *linhas de hiperfluxo*.

As maiores extensões na BEVL seriam àquelas provocadas pela tentativa de aplicar a biblioteca na visualização de dados resultantes da análise dinâmica. Modelagem dinâmica significa a utilização de modelos matemáticos e geométricos para visualização e compreensão da estrutura e do comportamento de objetos ao longo do tempo. Como conseqüência, os métodos computacionais numéricos de análise de tais modelos não fornecem somente um conjunto de resultados, como nos problemas estáticos, mas vários conjuntos de dados, um conjunto para cada instante de tempo do intervalo de tempo considerado. Ainda seria possível utilizar diretamente as classes da BEVL para visualização dos dados obtidos em cada intervalo de tempo. No entanto, devido ao caráter temporal dos resultados da modelagem dinâmica, seria mais conveniente a visualização de tais resultados ao longo do tempo de análise, o que conduz naturalmente ao emprego de técnicas de animação. O *sistema de animação* de uma aplicação de modelagem deveria ser capaz de armazenar uma animação resultante da simulação em meio persistente (usualmente disco rígido ou CD) e reproduzir a animação na velocidade adequada, além de representar, armazenar e manipular os atores, luzes, câmeras e ações envolvidos na geração da seqüência animada.

Referências Bibliográficas

[1] BAJAJ, C.; PASCUCCI, V.; SCHIKORE, D. Visualization of Scalar Topology for Structural Enhancement. *Proceeding of the IEEE Visualization*, p.51-58, 1998, Research Triangle Park, NC.

[2] BEER, G.; WATSON, J.O. *Introduction to Finite and Boundary Element Methods for Engineers*. John Wiley & Sons, 1992.

[3] BORIN, E.; COLOMBO, J.A.; SACCHI, R.P. *Um Sistema de Modelagem de Sólidos*, Campo Grande, 2000. Monografia (Projeto de Graduação) – Universidade Federal de Mato Grosso do Sul.

[4] BREBBIA, C.A.; TELLES, J.C.F; WROBEL, L.C. *Boundary Element Techniques: Theory and Applications in Engineering*. Springer-Verlag, 1984.

[5] DEFANTI, T.A.; BROWN, M.D.; McCORMICK, B.H. Visualization Expanding Scientific and Engineering Research Opportunities. *Visualization in Scientific Computing*, Nielson, G.M; Shriver, B., eds, IEEE Computer Society Press, Los Alamitos, CA, 1990, p.31-47.

[6] DELMARCELLE, T.; HESSELINK, L. A Unified Framework for Flow Visualization. In: GALLAGHER, R., ed. *Computer Visualization - Graphics Techniques for Scientific and Engineering Analysis*. CRC Press, 1995, p.129-170.

[7] FOLEY, J. et al. *Computer Graphics: Principles and Practice*. 2.ed. Addison-Wesley Publishing Company, 1992.

[8] GALLAGHER, R.S.; NAGTEGAAL, J.C. An Efficient 3-D Visualization Technique for Finite Element Models and Other Coarse Volumes. *Proceedings of SIGGRAPH'89, in Computer Graphics*, 23, 3, July 1989.

[9] GALLAGHER, R.S. Scientific Visualization: an Engineering Perspective. In: GALLAGHER, R., ed. *Computer Visualization - Graphics Techniques for Scientific and Engineering Analysis*. CRC Press, 1995, p.3-16.

[10] GALLAGHER, R.S. Scalar Visualization Techniques. In: GALLAGHER, R., ed. *Computer Visualization - Graphics Techniques for Scientific and Engineering Analysis*. CRC Press, 1995, p.89-127.

[11] GALLAGHER, R.S. Future Trends in Scientific Visualization. In: GALLAGHER, R., ed. *Computer Visualization - Graphics Techniques for Scientific and Engineering Analysis*. CRC Press, 1995, p.291-304.

[12] HABER, R.B.; McNABB, D.A. Visualization Idioms: A Conceptual Model for Scientific Visualization Systems, in *Visualization in Scientific Computing*, Nielson, G.M; Shriver, B., eds, IEEE Computer Society Press, Los Alamitos, CA, 1990, p.74-93.

[13] HELMAN, J.; HESSELINK, L. Representation and Display of Vector Field Topology in Fluid Flow Data Sets. *Visualization in Scientific Computing*, Nielson, G.M; Shriver, B., eds, IEEE Computer Society Press, Los Alamitos, CA, 1990, p.61-73.

[14] KANE, J.H. *Boundary Element Analysis in Engineering Continuum Mechanics.* Prentice-Hall International, 1994.

[15] LORENSEN, W.E.; CLINE, H.E. Marching Cubes: a High Resolution 3D Surface Construction Algorithm. *Computer Graphics*, v.21, n.3, p.163-, 1987.

[16] MÄNTYLÄ, M.; SULONEN, R. GWB – A Solid Modeler with Euler Operators. *IEEE Computer Graphics and Applications.* v.2, n.7, p.17-31, 1982.

[17] MÄNTYLÄ, M. *An Introduction to Solid Modeling.* Computer Science Press, 1988.

[18] MALVERN, L.E. *Introduction to the Mechanics of a Continuous Medium.* New Jersey, Prentice-Hall, 1969.

[19] COSTA JUNIOR, R.M.M. *Geração Automática de Malhas para Modelos Geométricos Tridimensionais com Casos de Carregamento e Condições de Contorno.* Campo Grande, 2000. Proposta de Dissertação (Mestrado) – Universidade Federal de Mato Grosso do Sul.

[20] NELSON, M.; CRAWFIS, R. Advances in Scientific Visualization. *Presented at Symposium on Electronic Imaging: Science and Technology*, San Jose, 1995.

[21] NOOR, A.K. Pathway to the Future of Simulation and Learning. In: TOPPING, B.H.V., ed. *Computational Mechanics for the Twenty-First Century*, Saxe-Coburg, 2000. p1-22.

[22] PAGLIOSA, P.A. *Um Sistema de Modelagem Estrutural Orientado a Objetos.* São Carlos, 1998. Tese (Doutorado) – Escola de Engenharia de São Carlos – USP.

[23] PAGLIOSA, P.A.; PAIVA, J.B. OSW: A Toolkit for Object Oriented Structural Modeling, in *Developments in Engineering Computational Technology*, Topping, B.H.V., ed, Civil-Comp Press, 151-166, 2001.

[24] ROGERS, D.F. *Procedural Elements for Computer Graphics.* McGraw-Hill, 1985.

[25] SCHNACK, E.; SZIKRAI, S.; TÜRKE, K. Domain Decomposition by Coupling of Finite Element Method and Boundary Element Method. In: TOPPING, B.H.V., ed. *Computational Mechanics: Techniques and Developments.* Civil-Comp Press, 2000, p.69-74.

[26] SCHROEDER, W.J. *Geometric Triangulations, with application to fully automatic 3-D mesh generation.* Nova York, 1991. Ph.D. Thesis – Rensselaer Polytechnic Institute.

[27] SCHROEDER, W.J.; SHEPARD, M.S. Analysis Data for Visualization. In: GALLAGHER, R., ed. *Computer Visualization - Graphics Techniques for Scientific and Engineering Analysis.* CRC Press, 1995, p.61-87.

[28] SCHROEDER, W.J.; MARTIN, K.; LORENSEN, B. *The Visualization Toolkit - An Object-Oriented Approach to 3D Graphics.* Prentice Hall, 1996.

[29] SHEWCHUK, R.J. *Lecture Notes on Delaunay Mesh Generation*. Department of Eletrical Engineering and Computer Science – University of California at Berkeley, 1999.

[30] SOUZA, V.J.B. *Algoritmos de Integração Eficientes para o Método dos Elementos de Contorno Tridimensional*. São Carlos, 2001. Dissertação (Mestrado) – Escola de Engenharia de São Carlos – USP.

[31] VENTURINI, W.S. *Boundary Element Method in Geomechanics*. Berlin, Springer-Verlag, 1983.

[32] WEILER, K.J. *Topological Structures for Geometric Modeling*. Nova York, 1986. Ph.D. Thesis Rensselaer Polytechnic Institute.

[33] ZIENKIEWICZ, O.C.; TAYLOR, R. *The Finite Element Method*, McGraw-Hill, v.1, 1994.

[34] ZIENKIEWICZ, O.C.; TAYLOR, R. *The Finite Element Method*, McGraw-Hill, v.2, 1994.