UM SISTEMA DE MODELAGEM ESTRUTURAL ORIENTADO A OBJETOS

Paulo Aristarco Pagliosa

Tese apresentada à Escola de Engenharia de São Carlos, da Universidade de São Paulo, como parte dos requisitos para obtenção do título de **Doutor em Engenharia de Estruturas**

ORIENTADOR: Prof. Dr. João Batista de Paiva

São Carlos
1998

Ficha catalográfica preparada pela Seção de Tratamento da Informação do Serviço de Biblioteca - EESC-USP

P138s

Pagliosa, Paulo Aristarco
Um sistema de modelagem estrutural orientado a objetos / Paulo Aristarco Pagliosa. -- São Carlos, 1998.

Tese (Doutorado) -- Escola de Engenharia de São Carlos-Universidade de São Paulo, 1998.
Área: Engenharia de Estruturas
Orientador: Prof. Dr. João Batista de Paiva

1. Programação orientada a objetos. 2. Método dos elementos finitos. 3. Método dos elementos de contorno. I. Título.

A meu filho, Lucas.

Agradecimentos

Ao meu orientador e amigo João Batista de Paiva. Muito obrigado pela disponibilidade em me orientar, de maneira tão generosa, nos estudos de assuntos tão interessantes. Se sua excelência em orientação tivesse se restringido somente à área técnica, ainda assim eu estaria cheio de motivos para dizer muito obrigado. Mas, além disso, Dr. Paiva me deu orientações de verdadeiro amigo. Agradeço, também, pela confiança em mim depositada. Espero que eu possa vir a ter muitas oportunidades de merecê-la.

Ao amigo Humberto Breves Coda. Em muitas situações interrompi o trabalho do Dr. Coda no SET para tirar dúvidas e trocar idéias sobre elementos finitos e de contorno. Em todas elas, sempre aprendi muito.

Aos colegas do Departamento de Computação e Estatística da Universidade Federal de Mato Grosso do Sul, pela infra-estrutura e apoio constantes. Em particular, aos amigos Marcelo F. Siqueira, Fábio H.V. Martinez e Ronaldo Alves Ferrerira. Marcelo possui um gosto por livros tão exagerado quanto sua generosidade em emprestá-los. Sobre minha mesa há mais livros de sua biblioteca particular do que meus próprios. Agradeço também pelo programa de impressão *Post-script* de modelos poliedrais, responsável por algumas das figuras impressas na primeira parte do texto. Com seu incentivo pessoal, Fábio tem tentado, com relativo sucesso, levantar meu ânimo por diversas vezes nos últimos tempos. Muito obrigado. Ronaldo, por sua vez, não mediu esforços para oferecer todas as condições necessárias para a conclusão do trabalho. Sou grato, igualmente, ao Prof. Celso Vitório Pierezan. Foi durante sua gestão como reitor da UFMS que tive oportunidade de me afastar de minhas atividades acadêmicas e me dedicar exclusivamente ao doutorado.

Aos amigos do Departamento de Engenharia de Estruturas da Escola de Engenharia de São Carlos, não somente por desempenharem suas atividades com máxima competência mas, sobretudo, pelas contínuas demonstrações de boa vontade. Quando estou no SET, sinto-me realmente em casa. Em especial, gostaria de agradecer a Maria Nadir Minatel, Rosi Aparecida Jordão Rodrigues, Rui Roberto Casale, Antônio Valdair Carneiro e Norberto Costardi.

Aos queridos amigos de São Carlos, por me cederem, gratuitamente, suas casas, algumas de suas preocupações e seus inesgotáveis sensos de humor. Obrigado por tudo.

À CAPES, pelas bolsas de estudos concedidas.

Conteúdo

Lista de Figuras

Lista de Programas

Resumo

Pagliosa, P.A. *Um Sistema de Modelagem Estrutural Orientado a Objetos*. São Carlos, 1998. Tese (Doutorado) - Escola de Engenharia de São Carlos, Universidade de São Paulo.

Em engenharia, modelos podem ser entendidos como representações das características principais de um objeto, criadas com o propósito de permitir a visualização e compreensão da estrutura e do comportamento do objeto, antes de sua construção. A estrutura de um objeto de engenharia pode ser definida por um *modelo geométrico* que descreve, exata ou aproximadamente, suas formas e dimensões materiais. O comportamento pode ser descrito por um conjunto de equações diferenciais de um *modelo matemático* que nos permite prever, sob certas condições, os efeitos de ações externas sobre o objeto. A solução do modelo matemático pode ser obtida pela análise computacional numérica de um *modelo mecânico* do objeto, através do método dos elementos finitos e/ou método dos elementos de contorno.

Nesse trabalho, apresentamos um Sistema de Modelagem Estrutural Orientado a Objetos denominado OSW — *Object Structural Workbench*, destinado ao desenvolvimento de programas de análise e visualização de modelos em engenharia de estruturas. Na primeira parte do texto, introduzimos os fundamentos utilizados no desenvolvimento do sistema. Na segunda parte, descrevemos como empregar as *bibliotecas de classes* de OSW na construção de um programa de modelagem e apresentamos alguns resultados obtidos com o sistema.

Palavras-chave: *modelagem*, *programação orientada a objetos*, *método dos elementos finitos*, *método dos elementos de contorno*, *computação gráfica*.

Abstract

Pagliosa, P.A. *Um Sistema de Modelagem Estrutural Orientado a Objetos*. São Carlos, 1998. Tese (Doutorado) - Escola de Engenharia de São Carlos, Universidade de São Paulo.

In engineering, models may be thought as representations for the main characteristics of an object. Such representations enable us to visualize and understand the object structure and behaviour before constructing the object itself. The engineering object structure can be defined by a *geometric model* which faithful or approximately describes the object shape and size. The object behaviour can be ruled by a differential equations set from a *mathematical model*, which enables us to predict the effects of external forces acting on the object. The solution for the mathematical model can be obtained by applying the method of finite elements or method of boundary elements to an object *mechanical model*.

In the text, we present an Object Oriented Structural Modeling System called OSW — *Object Structural Workbench*. The system has been designed to aid the development of computer programs for analysis and visualization of structural models. The text has been divided into two parts. At the first one, we introduce the mathematical and computational basis employed in OSW construction. At the second one, we describe how to use the OSW *class libraries* to develop our own structural modeling applications, and also we present some results from OSW.

Keywords: *modeling*, *object-oriented programming*, *finite element method*, *boundary element method*, *computer graphics*.

Parte Um

FUNDAMENTOS

CAPÍTULO 1

Introdução

1.1 Problema Fundamental

Qual é o comportamento de uma estrutura? Como podemos prever esse comportamento, e com que precisão e rapidez? Alguns dos motivos principais da *análise estrutural* em engenharia, relacionados a essas e a outras questões, podem ser objetivamente definidos a partir do seguinte problema, o qual chamaremos, para simples fim de denominação, de fundamental.

PROBLEMA FUNDAMENTAL Quais os estados de deslocamentos, deformações e tensões de um sólido submetido à ação de um sistema equilibrado de forças externas?

A resposta do problema fundamental é muito importante para o engenheiro de estruturas. O conhecimento das tensões e deformações em um sólido permite concluir sob quais condições uma estrutura pode ser seguramente construída ou não. Entretanto, a solução *exata* para essa questão 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 objetos que, na realidade, nunca se encontram isolados, dirigindo sua atenção apenas às propriedades mais importantes do problema de interesse e criando, em função dessas propriedades, representações que definem idealmente a estrutura e o comportamento dos objetos. Em engenharia, as representações de comportamento, especificamente, são elaboradas a partir de hipóteses, quase sempre simplificadoras, baseadas em observações que devem ser posteriormente comprovadas pela experimentação. Por indução, estabelecemos princípios, quantitativamente descritos por equações matemáticas, que são reunidos em uma teoria que explica, de forma mais ou menos aproximada, o comportamento dos objetos isolados.

O próprio problema fundamental só tem sentido se formulado à luz da teoria da mecânica do contínuo, pois é aí que os conceitos modernos de tensão e deformação, existentes somente na esfera da idéia pura, são estabelecidos e assumem importância real na análise de projetos de estruturas. A teoria é baseada na hipótese de que a

matéria é contínua, destituída de espaços vazios. Tal suposição permite a definição de tensão em um ponto — um lugar geométrico no espaço que não ocupa volume algum — como sendo um limite matemático, tal como a definição de derivada no cálculo diferencial. A mecânica do contínuo, portanto, não faz uma descrição precisa da natureza. A simplificação de continuidade, no entanto, não tira o mérito nem diminui a utilidade da teoria nos campos onde é convenientemente aplicada. Na verdade, as equações da mecânica do contínuo predizem um comportamento macroscópico dos materiais, nas condições como são utilizados na construção de estruturas, que, quantitativamente, está de acordo com os resultados obtidos pelas experiências. Felizmente, notícias de ruínas de estruturas, construídas com base no conceito matemático do *continuum*, não são comuns e nem freqüentes.

Outras suposições, impostas por limitações de processos e ferramentas de cálculo, ou por conveniências práticas de construção e utilização de estruturas — deslocamentos, deformações e rotações muito pequenos em relação às dimensões do objeto, por exemplo —, conduzem a teorias ainda mais simplificadas, mas que, quando formuladas consistente e logicamente, são tão respeitáveis quanto a teoria avançada. É o caso das teorias de vigas, placas e cascas, que dentro dos limites determinados pela experiência, podem ser adequadamente empregadas na análise de muitos problemas úteis em engenharia. Mas mesmo nessas teorias simplificadas, soluções analíticas "exatas" que respondem a pergunta do problema fundamental somente são definidas para alguns casos particulares de geometria e condições de contorno. Para os sistemas contínuos mais gerais, apenas soluções aproximadas são possíveis. E notemos que são soluções aproximadas de equações que descrevem aproximadamente o comportamento do objeto, baseadas em descrições incompletas, imprecisas e igualmente aproximadas da natureza.

Em qualquer caso, o problema fundamental pode ser "resolvido" numericamente em computador. De fato, os programas de computador para engenharia têm sido caracterizados, principalmente, por implementações de soluções numéricas de equações diferenciais que descrevem os principais aspectos comportamentais de uma variedade de problemas físicos. Essas implementações são baseadas em formulações derivadas de métodos de *discretização* do sistema contínuo, o qual passa a ser representado, dessa forma, por um conjunto finito de componentes sobre os quais são definidas aproximações para as variáveis que governam o problema em questão. A solução, assim obtida, deve ser tanto mais precisa quanto maior for o número de variáveis do sistema discreto.

O método numérico mais amplamente utilizado em engenharia de estruturas é o método dos elementos finitos (MEF) [17, 132, 133]. A técnica, matematicamente fundamentada em princípios variacionais, ou mais genericamente, em expressões de resíduos ponderados [37], consiste na divisão do *continuum* em um número finito de partes, ou *elementos*, cujo comportamento é especificado por um conjunto discreto de parâmetros normalmente associados às variáveis do problema físico. O comportamento global do domínio, igualmente especificado em função de tais parâmetros, pode ser determinado pela solução de sistemas de equações algébricas definidos a partir das contribuições individuais de todos os elementos finitos. Uma outra abordagem física mais direta e intuitiva do processo de discretização, permite a criação de uma analogia entre porções finitas de um domínio contínuo e elementos discretos reais. A denominação *elemento finito*, primeiramente utilizada por CLOUGH [22], é derivada desse procedimento mais intuitivo de *idealização estrutural* [91], no qual são baseados os métodos matriciais de análise de estruturas reticuladas [125, 126]; nesses casos, a aproximação de um

domínio como um conjunto de elementos lineares conduz a uma representação exata do sistema estrutural. Tentativas de extensão do método para o tratamento de problemas em mecânica dos sólidos foram apresentadas nos trabalhos de MCHENRY [76], HRENNIKOFF [51] e NEWMARK [78], que demostraram, ainda na década de 40, que "soluções razoavelmente boas para problemas elásticos contínuos podem ser obtidas através da substituição de pequenas porções do domínio analisado por um arranjo de barras elásticas simples." Mais tarde, no mesmo contexto, ARGYRIS [4] e TURNER *et alli* [119] mostraram que "uma substituição mais exata de propriedades pode ser obtida a partir de consideracões simplificadoras do comportamento de pequenas porções ou elementos do contínuo." A partir da década de 60, estudos detalhados dos princípios variacionais possibilitaram uma generalização do método. Desde então, o MEF tem sido empregado até mesmo em problemas onde um funcional não pode ser diretamente determinado; nesses casos, a formulação pode ser obtida a partir de métodos de resíduos ponderados aplicados às equações diferenciais do problema.

Uma importante técnica alternativa de análise numérica em mecânica do contínuo é o método dos elementos de contorno (MEC) [16, 18, 122]. O método baseia-se na transformação das equações diferenciais parciais que descrevem o comportamento de incógnitas, tais como deslocamentos e tensões, no interior e no contorno de um objeto em equações integrais definidas apenas em termos de valores no contorno, seguida da determinação de soluções numéricas dessas equações. Os valores das incógnitas em pontos internos, se necessários, são calculados a partir dos valores determinados no contorno. O método reduz a dimensionalidade do problema em um, porque as aproximações numéricas são realizadas apenas sobre os contornos, resultando em um sistema de equações menor que aquele obtido nos métodos de domínio. A primeira formulação dos métodos de contorno aplicada a problemas de elastostática foi proposta por KUPRADZE [63]. Contudo, pode-se considerar que o método "direto" de análise originou-se no trabalho de CRUSE e RIZZO [28] em elastostática. A denominação *elementos de contorno* é sugerida por BREBBIA [16] apenas em 1978, juntamente com as novas bases matemáticas do método. PAIVA [84] cita que "a partir da técnica dos resíduos ponderados, obteve-se uma maior generalização do MEC, sendo permitida sua associação com outros métodos numéricos. As pesquisas intensificaram-se e considerável progresso foi obtido na elaboração de novas formulações com aplicações no campo da engenharia", como podemos observar, por exemplo, na Escola de Engenharia de São Carlos, com os trabalhos de PAIVA [85], CODA [23] e VENTURINI [121].

Esse desenvolvimento acentuado da tecnologia de análise numérica em engenharia tem motivado a criação de programas de computador que podem ser aplicados à resolução de um número cada vez maior de estruturas complexas, utilizando tipos diversos e cada vez mais sofisticados de elementos estruturais. A "eficiência", em tais programas, é usualmente definida em termos da capacidade de armazenamento de grandes sistemas de equações na memória do computador e da precisão e rapidez do processamento numérico. Caracteristicamente, os sistemas de análise em engenharia têm sido codificados em FORTRAN, porque a linguagem é numericamente "eficiente."

A entrada dos dados de um programa de análise estrutural, geralmente organizados em arquivos de texto, consiste de uma quantidade considerável de informações relativas à definição da geometria, carregamentos, materiais e condições de contorno. A montagem manual desse arquivo pode se tornar uma tarefa muito complicada, dependendo da complexidade do problema, induzindo-nos a erros que são mais difíceis de detectar e corrigir. Além disso, uma discretização adequada da estrutura, funda-

mental para a qualidade da solução numérica, é em determinados casos, praticamente impossível de ser obtida sem o auxílio de um processo automático de geração de malhas [44, 50, 102, 116, 131]. Por isso, a utilização de componentes *pré-processadores* em sistemas de análise de estruturas, responsáveis pela discretização automática do domínio e verificação dos dados de entrada, não é somente conveniente, mas também essencial. Por outro lado, os resultados de saída tendem a produzir extensas listagens numéricas que não permitem ou até mesmo impossibilitam uma interpretação mais imediata dos valores obtidos na análise. Nesse caso, é vantajoso que as distribuições de grandezas tais como deslocamentos, tensões, forças e momentos possam ser representadas graficamente, por exemplo, como curvas ou superfícies de nível [48, 92] sobre o domínio considerado. Componentes *pós-processadores*, responsáveis pela organização dos arquivos de saída e *visualização* [41] dos resultados do processamento são igualmente essenciais em programas de engenharia. Nesse contexto, o desempenho e a aplicação do sistema não são definidos apenas pela eficiência da análise, mas também pela *interface* com o usuário. Programas que são complicados ou confusos de usar, ou que oferecem poucas possibilidades de entrada de dados e de visualização de resultados são, sob o ponto de vista do usuário, muito pouco úteis e interessantes [8]. Ao contrário, os sistemas que dispõe de ricos recursos de interação, principalmente interfaces gráficas com o usuário, são muito mais atrativos até mesmo quando possuem um campo mais restrito de aplicações.

O emprego de gráficos em programas de engenharia é bastante apropriado porque os objetos físicos de interesse possuem uma estrutura tridimensional que é muito bem definida geometricamente e que, portanto, admite uma representação gráfica natural. De fato, a *computação gráfica* [38] oferece um dos meios mais naturais de comunicação com o computador, pois a desenvolvida capacidade humana de reconhecimento de padrões permite perceber e processar dados gráficos rápida e eficientemente. A computação gráfica *interativa*, notadamente, possibilita o controle dinâmico e preciso do conteúdo, estrutura e aparência de objetos e suas imagens. Sistemas de *computer-aided design* (CAD), por exemplo, têm-se tornado, segundo MÄNTYLÄ [75], "quase que universalmente aceitos como ferramentas centrais para uma diversidade de aplicações em engenharia, impulsionando significativamente a produtividade em projetos e permitindo análises mais precisas e rigorosas, com redução de erros e aumento de qualidade de documentação." O sucesso dos programas de CAD, segue MÄNTYLÄ, "deve-se ao fato de que muitas tarefas importantes em projetos de engenharia são relacionadas, fundamentalmente, à forma geométrica dos objetos." Como resultado, a implementação de uma diversidade de técnicas de representação e utilização das informações geométricas de objetos em computadores constitui o núcleo principal dos sistemas de CAD correntes, os quais permitem a utilização de inúmeras ferramentas de visualização e de detalhamentos construtivos, bem como a produção de desenhos de engenharia e outras documentações de projeto.

Modelos e Aplicações de Modelagem

Até este ponto, procuramos justificar a necessidade da criação de *representações* que descrevem de forma mais ou menos simplificada a estrutura e o comportamento de um objeto físico que desejamos construir. Essas representações, especificadas com o propósito de permitir a visualização e a compreensão do objeto, são genericamente denominadas de *modelos*.

Consideraremos, para nossos propósitos, os seguintes tipos de modelos:

- *Modelos geométricos.* As formas e as dimensões dos objetos de engenharia podem ser representadas, exata ou aproximadamente, por superfícies de sólidos constituídos de materiais com propriedades físicas conhecidas, o que sugere o emprego natural da computação gráfica interativa nos processadores de entrada e saída dos programas de análise estrutural.

- *Modelos matemáticos.* O comportamento de uma estrutura em relação a ações externas é representado por equações diferenciais formuladas com base em hipóteses simplificadoras, mas que conduzem a previsões que dentro de certos limites, são comprovadas pela experimentação.

- *Modelos mecânicos.* A solução do modelo matemático só pode ser aproximadamente estabelecida, para os casos gerais, por processos computacionais numéricos que envolvem uma discretização do sistema contínuo. Os programas de computador para engenharia têm sido caracterizados por implementações dessas soluções.

A utilização de modelos, usual em ciências e engenharia, foi motivada nesse trabalho pela determinação de obtermos uma solução para o problema fundamental, o que nos conduziu à discussão dos métodos numéricos e dos programas computacionais de análise de estruturas. Devido ao fato desses programas de análise estrutural serem baseados em modelos, os chamaremos de *programas de modelagem*, ou *aplicações de modelagem*, e os distinguiremos como sendo constituídos de componentes responsáveis pelas seguintes operações:

- *Modelagem* (pré-processamento). O processo de análise estrutural começa com a construção de um modelo de sólido que descreve a geometria da estrutura, suas propriedades materiais, condições de contorno, condições iniciais e casos de carregamentos. Os dados do modelo geométrico, juntamente com as especificações dos tipos de elementos e da precisão desejada, são utilizados para a geração de malhas de elementos que definem os modelos mecânicos da estrutura.

- *Análise.* Análise em engenharia, segundo SHEPARD e SCHROEDER [104], é "o 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", nesse caso, são dadas no modelo mecânico construído na etapa de modelagem; as "informações de saída" são as respostas do problema fundamental: campos de deslocamentos, de deformações e de tensões da estrutura. O "conjunto apropriado de transformações" é definido pela metodologia numérica empregada na análise, por exemplo, método dos elementos finitos e método dos elementos de contorno.

- *Visualização* (pós-processamento). Visualizar significa transformar dados extraídos de um objeto em imagens que representam informações sobre o objeto [101]. Essas imagens não representam somente informações sobre a estrutura física ou geométrica dos objetos, mas podem representar também informações sobre os dados resultantes do processo de análise, tais como distribuições de campos escalares de um domínio nas formas de mapas de cores ou de isolinhas ou isosuperfícies.

Orientação a Objetos

Como podemos representar a estrutura e a funcionalidade de modelos em computador? O desenvolvimento de uma solução computacional para o problema fundamental de engenharia de estruturas nos conduz a outro problema, igualmente fundamental, de engenharia de *software* [90]: *como modelar objetos do mundo real em computador tão próximo quanto possível da visão que o homem tem desse mundo?* A solução, proposta nesse trabalho, é baseada na *programação orientada a objetos* (POO) [27, 111].

A orientação a objetos, segundo KHOSHAFIAN [59], é a "tecnologia de modelagem e desenvolvimento de sistemas que facilita a construção de programas complexos a partir de componentes individuais", ou *objetos*, os quais combinam *atributos* que representam a estrutura e *métodos* que representam o comportamento de uma entidade concreta ou abstrata. O procedimento, nesse caso, é idêntico àquele efetuado na modelagem do mundo físico: isolar objetos mais simples de descrever e entender. A vantagem é que a POO permite que essa maneira natural de analisar em termos de objetos seja estendida para as fases de projeto e implementação de programas, encorajando a reutilização de código e a clareza de programação, permitindo a criação de sistemas que são mais facilmente compreendidos e compartilhados com outras pessoas.

A programação orientada a objetos tem sido empregada apenas recentemente no desenvolvimento computacional de soluções numéricas de problemas de engenharia. Nos trabalhos de ZIMMERMANN *et alli* [33, 34] procura-se "demonstrar a utilidade deste paradigma de programação alternativo no campo dos elementos finitos", investigando-se "o potencial da POO na melhoria da legibilidade, modularidade e reutilização de código." Em DUBOIS-PÈLERIN e ZIMMERMANN [35] apresenta-se uma implementação em C++ do método dos elementos finitos aplicado à análise elástica linear de chapas e treliças planas submetidas a ações estáticas e dinâmicas; a conclusão é que C++ é uma "ferramenta adequada à prática de programação de elementos finitos" porque, em relação à versão FORTRAN do programa, "são verificadas performances excelentes na montagem do sistema linear de equações e na resolução deste sistema", com "melhorias muito significativas na clareza, capacidade de extensão e facilidade de depuração do código."

Em MENÉTREY e ZIMMERMANN [77], a POO é utilizada na análise não-linear física de estruturas planas pelo MEF, demonstrando-se que o emprego de objetos "pareceu adequado" neste caso, resultando "em um programa que pode ser facilmente estendido" com "redução de possíveis erros de programação." Outras soluções do MEF orientado a objetos, elaboradas com os mesmos propósitos de investigar a utilidade da POO em programas de análise numérica de engenharia, são apresentadas, para problemas estáticos, por GAJEWSKI [39], KONG e CHEN [61] e ZEGLINSKI e HAN [130]; para problemas dinâmicos, por PIDAPARTI e HUDLI [89]; e, para problemas de contato, por ULBIN, ZEN e FLASKER [120].

EYHERAMENDY e ZIMMERMANN [36], originalmente, num trabalho muito interessante no qual "os princípios da programação orientada a objetos são aplicados diretamente à sentença do problema em sua forma diferencial", vão muito além da implementação numérica do MEF, propondo um procedimento de geração automática de código onde "a derivação parte das equações diferenciais e passa pela forma fraca, forma de Galerkin e forma matricial, com o emprego de manipulações simbólicas" [29]. EYHERAMENDY e ZIMMERMANN concluem que o procedimento pode ser aplicado amplamente em "ferramentas de projeto auxiliado por computador destinadas a

computações numéricas" e que a orientação a objetos "parece conduzir a um modelo natural de descrição de problemas mecânicos."

Em outros trabalhos, ARRUDA *et alli* [5] discutem as vantagens do desenvolvimento de programas de engenharia orientados a objetos no ambiente gráfico do Windows [88]; JU e HOSAIN [56] empregam a orientação a objetos na análise de subestruturas [91] pelo MEF; BOMME e ZIMMERMANN [14] descrevem um esquema de integração de objetos com regras de inteligência artificial, com o propósito de "criar objetos inteligentes para uso na programação de elementos finitos orientados a objetos"; e GAJEWSKI e LOMPIES [40] aplicam a POO na implementação de algoritmos de redução da largura de banda de matrizes esparsas [25, 45, 106], mostrando a facilidade com que se pode "adicionar novas funcionalidades a um código do método dos elementos finitos já existente."

DEVLOO [31], em um estudo sobre a eficiência dos programas orientados a objetos em C++, mostra que "o uso indiscriminado de certas construções em POO realmente conduzem a uma redução (significativa) de eficiência", mas que, por outro lado, "se um programa orientado a objetos é escrito para eficiência máxima, sua performance é igual ou comparável à implementação FORTRAN equivalente", concluindo-se que a idéia de que a POO é inerentemente ineficiente é errada, sendo "conseqüência de uma compreensão deficiente da linguagem." Além disso, o conceito de eficiência, nesses casos, pode ser empregado de uma maneira mais ampla. De fato, uma linguagem de programação orientada a objetos também pode ser considerada eficiente se permite ao programador modelar um problema e projetar uma solução mais rapidamente, através do emprego de técnicas que facilitam o reaproveitamento e a extensão do código existente, com redução dos esforços de implementação e manutenção. MACKIE [71] conclui que, pelo fato dos programas de computador para engenharia "estarem se tornando cada vez mais complexos", há um "incentivo maior à modelagem de dados, em adição à implementação de algoritmos numéricos", sendo os "métodos orientados a objetos bastante adequados para esse desafio."

1.2 OSW

Nessa Seção, voltaremos nossa atenção à construção de aplicações de modelagem orientadas a objetos em C++ e introduziremos um sistema de modelagem estrutural que poderá nos auxiliar nessa tarefa. Denominaremos esse sistema de *Object Structural Workbench* e o designaremos, freqüentemente, pelo acrônimo OSW.

OSW OSW é um sistema de computador aplicado ao desenvolvimento de programas de modelagem estrutural orientados a objetos em C++.

O projeto e a implementação de OSW foram guiados por quatro propriedades que consideramos muito importantes em um programa de modelagem estrutural:

- *Funcionalidade.* A funcionalidade de um programa de modelagem estrutural é caracterizada pelos recursos de pré-processamento, análise e pós-processamento oferecidos pelo programa, definidos em função das necessidades de uma aplicação em particular. Aplicações de análise elastostática de cascas e de sólidos, por exemplo, podem requerer métodos distintos de modelagem geométrica, geração

de malhas, análise numérica e visualização dos resultados da análise. Dependendo dos propósitos das aplicações, podemos utilizar um esquema de representação de cascas bem mais simplificado que um esquema de representação de sólidos; da mesma forma, o método dos elementos finitos pode ser mais convenientemente aplicado à análise de cascas e o método dos elementos de contorno à análise de sólidos. Notemos, porém, que os recursos de modelagem, análise e visualização de um programa de modelagem não definem, somente, a funcionalidade específica de uma aplicação, mas também operações que são comuns a todos os programas de modelagem. Por exemplo, independentemente do método de análise numérica adotado, as incógnitas de *qualquer* problema são sempre determinadas a partir da resolução de sistemas de equações lineares. Naturalmente, métodos especializados de resolução de sistemas de matrizes esparsas, por exemplo, podem (e devem) ser empregados em determinadas aplicações. Idealmente, gostaríamos de escrever um programa de modelagem estrutural que pudesse ser aplicado a *todos* os problemas de engenharia de estruturas.[1] Felizmente, não é esse o caso. Isso nos leva à propriedade seguinte.

- *Extensibilidade.* Um programa de modelagem estrutural sempre pode crescer em todas as direções e sentidos, incorporando recursos e técnicas mais eficientes de representação e manipulação de objetos e de imagens de objetos, bem como novos processos de análise que permitem que o programa possa ser aplicado a um número maior de casos de modelagem de estruturas. Esse crescimento deve ser baseado, tanto quanto possível, no reaproveitamento do código já existente. Um programa de modelagem é um programa complexo e não podemos começar quase sempre do zero a cada nova aplicação. As extensões ou modificações de um componente do programa não deveriam implicar, também, em extensões ou modificações em outros componentes. Um dos principais motivos de construirmos programas orientados a objetos deve-se ao fato da POO nos fornecer mecanismos realmente muito adequados para extensão da funcionalidade de nossos programas de modelagem estrutural, como procuraremos enfatizar ao longo do texto.

- *Eficiência.* A eficiência é essencial em programas que envolvem processamento gráfico e numérico. OSW é um *toolkit* de desenvolvimento de programas de modelagem C++ porque C++ é uma linguagem compromissada, desde o projeto original de STROUSTRUP [110], com a eficiência, mesmo quando utilizamos suas construções orientadas a objetos. Os programas de modelagem C++ baseados em OSW executam no sistema operacional Windows NT. Escolhemos o Windows NT simplesmente porque é um sistema operacional *multithread* que oferece uma interface gráfica bastante conhecida. Além disso, seus esquemas de gerenciamento de memória virtual e de ligação dinâmica nos permitem escrever mais facilmente grandes programas de computador que demandam grandes quantidades de espaço.

- *Facilidade de uso.* Os programas de modelagem estrutural desenvolvidos em OSW possuem uma interface gráfica homem-computador baseadas em janelas do Windows NT. Essa interface é responsável pela entrada dos comandos do usuário e pela saída dos resultados do processamento dos comandos.

[1] Nessa eventualidade, os trabalhos de investigação em mecânica computacional estariam concluídos.

Defendemos o emprego da programação orientada a objetos em OSW não somente porque a POO nos permite modelar problemas do mundo real tão próximo quanto possível da visão que temos desse mundo, ou porque podemos escrever programas que podem ser mais facilmente compreendidos e estendidos, mas também porque observamos uma identidade dos conceitos de modelos e objetos, como dados anteriormente. Nesse trabalho, empregaremos as propriedadde da POO — encapsulamento, herança, polimorfismo e identidade — na especificação de modelos de estruturas.

Nosso sistema de modelagem é basicamente constituído de:

- **Bibliotecas 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 uma centena de classes C++ que podem ser utilizadas para a construção de programas de modelagem orientados a objetos, como veremos posteriormente. Essas classes são organizadas nas chamadas *bibliotecas de classes*[2] do sistema.

- **Ambiente de desenvolvimento**. A utilização imediata das bibliotecas 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.

Nossos objetivos, nesse trabalho, são:

- Apresentar os fundamentos matemáticos e computacionais utilizados no desenvolvimento das classes de objetos de OSW;

- Descrever em detalhes as bibliotecas de classes de objetos de OSW e discutir seus principais aspectos de implementação; e

- Exemplificar a utilização das classes de objetos de OSW na construção de aplicações de modelagem em engenharia. Inicialmente, descreveremos as classes específicas de duas aplicações OSW de análise e de visualização de modelos estruturais. Na primeira, analisaremos estruturas constituídas de cascas elásticas pelo método dos elementos finitos e, na segunda, sólidos elásticos pelo método dos elementos de contorno. Posteriormente, apresentaremos algumas figuras coloridas que ilustram os resultados obtidos.

[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.

1.3 Organização do Texto

O texto é dividido em duas partes. Na primeira parte, **FUNDAMENTOS**, apresentaremos os conceitos, teorias e métodos matemáticos, numéricos e computacionais mais importantes utilizados para o desenvolvimento de OSW.

Capítulo 2
O que são Modelos?
No Capítulo 2 aprofundaremos nossa discussão sobre modelos e sobre modelagem geométrica, matemática e mecânica de estruturas. Discutiremos, também, as características estruturais e funcionais dos sistemas compugráficos interativos de modelagem e proporemos um modelo para nossos programas de modelagem estrutural.

Capítulo 3
Modelos Geométricos
No Capítulo 3 introduziremos as estruturas de dados e as principais funções de modelagem empregadas nas representações geométricas de objetos em OSW. Ilustraremos o Capítulo com vários trechos de programas escritos em linguagem C. (No Capítulo 8 escreveremos as versões C++ desses programas e discutiremos suas diferenças.)

Capítulo 4
Modelos Matemáticos
No Capítulo 4 as equações diferenciais que descrevem os campos de deslocamento e de tensões dos objetos estruturais considerados no trabalho serão deduzidas. Apresentaremos um resumo da mecânica do contínuo aplicada ao estudo elastostático de sólidos e, posteriormente, particularizaremos as equações para os casos de membranas e placas.

Capítulo 5
Análise Numérica de Modelos Estruturais
No Capítulo 5 descreveremos os algoritmos das técnicas numéricas de resolução das equações diferenciais de nossos modelos matemáticos: o método dos elementos finitos e o método dos elementos de contorno. Derivaremos as formulações de ambas as técnicas do método dos resíduos ponderados.

Capítulo 6
Modelos Mecânicos
No Capítulo 6 abordaremos os elementos finitos e de contorno utilizados na modelagem das estruturas analisadas no trabalho. Discutiremos os tipos de condições de contorno e de distribuições de carregamentos aplicados aos elementos estruturais e os processos de geração de malhas de OSW.

Capítulo 7
Visualização
No Capítulo 7 resumiremos os algoritmos compugráficos de síntese de imagens de modelos geométricos. Descreveremos, também, as técnicas de visualização de campos escalares e de campos vetoriais implementadas em OSW.

Capítulo 8
O que são Objetos?
No Capítulo 8 apresentaremos as definições de objeto e das principais propriedades da POO. Embora esses conceitos sejam independentes de uma linguagem de programação específica, os exemplificaremos, em um enfoque bastante prático, com trechos de programas das bibliotecas de classes de OSW, escritos em C++.

Na segunda parte, **OBJECT STRUCTURAL WORKBENCH**, descreveremos em detalhes as classes C++ do Sistema de Modelagem Estrutural Orientado a Objetos proposto no trabalho, discutindo seus principais aspectos de implementação e exemplificando sua utilização no desenvolvimento de programas de modelagem em engenharia de esruturas.

Capítulo 9
Construíndo Aplicações em OSW
No Capítulo 9 começaremos apresentando uma visão geral de OSW. Mostraremos a organização e funcionalidade das bibliotecas de classes do sistema e definiremos os principais componentes dos programas de modelagem baseados em OSW. Em seguida, demonstraremos o emprego das classes de objetos do *toolkit* na construção de aplicações de modelagem em engenharia de estruturas, através do desenvolvimento de dois programas de análise estrutural. No primeiro, modelaremos estruturas constituídas de cascas elásticas pelo método dos elementos finitos e, no segundo, sólidos elásticos pelo método dos elementos de contorno.

Capítulo 10
As Bibliotecas de Classes
No Capítulo 10 descreveremos detalhadamente as definições de tipos, constantes, macros, funções globais e classes das bibliotecas de classes de OSW.

Capítulo 11
Exemplos
No Capítulo 11 mostraremos alguns exemplos de programas de modelagem desenvolvidos com as classes de objetos de OSW e alguns resultados de análise e de visualização obtidos com esses programas.

Capítulo 12
Conclusão
No Capítulo 12, por fim, apresentaremos nossas conclusões, sugestões, dificuldades encontradas e contribuições pretendidas com o desenvolvimento do trabalho.

1.4 Notação

Nessa Seção introduziremos a notação utilizada no decorrer do texto, concordando com BEER e WATSON [11] que uma "boa notação é difícil", especialmente em trabalhos "que tratam de dois diferentes métodos numéricos."

Vetores, tensores e matrizes Em nossa notação, usaremos 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, independentemente do sistema de coordenadas considerado, o vetor $\mathbf{u}$ é obtido do vetor $\mathbf{v}$ por uma *transformação linear* definida pelo tensor $\mathbf{T}$. Usaremos, também, letras gregas minúsculas em negrito para denotar alguns tensores. Por exemplo, $\boldsymbol{\sigma}$ é o tensor de tensões de Cauchy e $\boldsymbol{\epsilon}$ é o tensor de pequenas deformações elásticas.

Por simplicidade, consideraremos somente sistemas de coordenadas Cartesianas. No espaço, um sistema de coordenadas Cartesianas é definido por três eixos mutuamente ortogonais, tal como mostrado na Figura 1.1. 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.

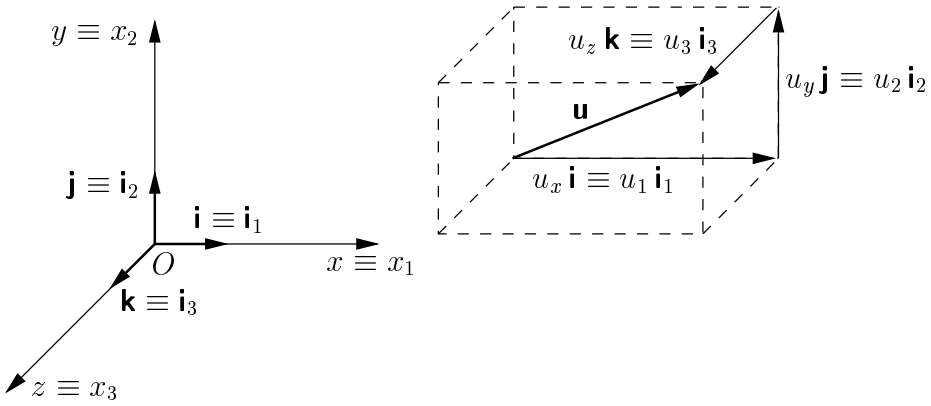


Figura 1.1: Sistema de coordenadas Cartesianas retangulares.

Os *componentes* de vetores e tensores em relação a um sistema de coordenadas podem ser convenientemente representados por matrizes. Usaremos letras minúsculas e maiúsculas em **romano negrito** para denotar matrizes de componentes de vetores e tensores. Por exemplo, as coordenadas Cartesianas (u_x, u_y, u_z) ou (u_1, u_2, u_3) de um vetor $\mathbf{u}$ podem ser organizadas na matriz coluna

$$\mathbf{u} = \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix} \qquad \text{ou} \qquad \mathbf{u} = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} . \tag{1.2}$$

Da mesma forma, os componentes Cartesianos de um tensor $\mathbf{T}$ de segunda ordem podem ser organizados matricialmente como

$$\mathbf{T} = \begin{bmatrix} T_{xx} & T_{xy} & T_{xz} \\ T_{yx} & T_{yy} & T_{yz} \\ T_{zx} & T_{zy} & T_{zz} \end{bmatrix} \qquad \text{ou} \qquad \mathbf{T} = \begin{bmatrix} T_{11} & T_{12} & T_{13} \\ T_{21} & T_{22} & T_{23} \\ T_{31} & T_{32} & T_{33} \end{bmatrix} . \tag{1.3}$$

Em coordenadas Cartesianas a expressão (1.1) é calculada como

$$\mathbf{u} = \mathbf{T}\,\mathbf{v}. \tag{1.4}$$

Observe, no entanto, que a representação $\mathbf{T} \cdot \mathbf{v}$ não é meramente outra notação para o produto matricial $\mathbf{Tv}$ da Equação (1.4). $\mathbf{T} \cdot \mathbf{v}$ simboliza uma *operação* que transforma um vetor em outro, enquanto $\mathbf{Tv}$ é o produto matemático das matrizes dos componentes de $\mathbf{T}$ e $\mathbf{v}$ em relação a um sistema de coordenadas Cartesianas.[3]

Para um tensor denotado por uma letra grega minúscula em negrito, por exemplo $\boldsymbol{\sigma}$, utilizaremos 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

$$n_i = \cos \alpha_i, \tag{1.5}$$

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

$$u_i = u n_i, \tag{1.6}$$

onde u, o módulo de $\mathbf{u}$, é

$$u^2 = u_1^2 + u_2^2 + u_3^2 = \sum_{i=1}^{3} u_i^2. \tag{1.7}$$

Na Equação (1.7) podemos omitir o símbolo Σ da fórmula porque, em notação tensorial Cartesiana, a ocorrência de dois índices repetidos em um *mesmo termo* representa somatório. Por exemplo,

$$\begin{aligned} u_i u_i &= u_1^2 + u_2^2 + u_3^2, \\ T_{kk} &= T_{11} + T_{22} + T_{33}. \end{aligned} \tag{1.8}$$

A Equação (1.7) pode agora ser escrita como

$$u^2 = u_i \, u_i. \tag{1.9}$$

Definiremos dois símbolos úteis que nos permitirão simplificar e abreviar equações em notação indicial. O símbolo de permutação e_{ijk} é definido como

$$e_{ijk} = \begin{cases} 0 & \text{quando quaisquer dois índices são iguais;} \\ +1 & \text{quando } i, j, k \text{ são } 1, 2, 3 \text{ ou uma permutação par de } 1, 2, 3; \\ -1 & \text{quando } i, j, k \text{ são uma permutação ímpar de } 1, 2, 3. \end{cases} \tag{1.10}$$

Usando o símbolo de permutação podemos escrever, por exemplo, o produto vetorial de dois vetores $\mathbf{u}$ e $\mathbf{v}$ como

$$\mathbf{u} \times \mathbf{v} = e_{ijk} u_j v_k \, \mathbf{i}_i. \tag{1.11}$$

[3] Em um sistema de coordenadas curvilíneas genérico é preciso algum cuidado no cálculo de $\mathbf{T} \cdot \mathbf{v}$ como um produto de matrizes. Por exemplo, o produto matricial de duas matrizes de componentes covariantes, em geral, não é igual a matriz de componentes covariantes de $\mathbf{u}$ (MALVERN [72]).

O *delta de Kronecker* é definido como

$$\delta_{ij} = \begin{cases} 1 & \text{se } i = j, \\ 0 & \text{se } i \neq j. \end{cases} \tag{1.12}$$

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

$$\mathbf{i}_j \cdot \mathbf{i}_k = \delta_{jk}. \tag{1.13}$$

Em notação tensorial Cartesiana, denotaremos as derivadas parciais como

$$\begin{aligned} \frac{\partial u_i}{\partial x_j} &= u_{i,j}, \\ \frac{\partial^2 u_i}{\partial x_j \partial x_k} &= u_{i,jk}. \end{aligned} \tag{1.14}$$

Gradientes e divergentes O gradiente de um escalar F é um vetor ∇F, ou $\operatorname{grad} F$, onde

$$\nabla = \frac{\partial}{\partial x_1}\mathbf{i}_1 + \frac{\partial}{\partial x_2}\mathbf{i}_3 + \frac{\partial}{\partial x_3}\mathbf{i}_3 = \mathbf{i}_k \frac{\partial}{\partial x_k}. \tag{1.15}$$

Portanto,

$$\operatorname{grad} F = \nabla F = \mathbf{i}_k F_{,k}. \tag{1.16}$$

O divergente de um vetor $\mathbf{u}$ é um escalar $\nabla \cdot \mathbf{u}$, ou $\operatorname{div} \mathbf{u}$, tal que

$$\operatorname{div} \mathbf{u} = \nabla \cdot \mathbf{u} = u_{k,k}. \tag{1.17}$$

O gradiente de um vetor $\mathbf{u}$ é um tensor de segunda ordem $\nabla\mathbf{u}$ tal que

$$\nabla\mathbf{u} = (\mathbf{i}_k \frac{\partial}{\partial x_k})(u_m \mathbf{i}_m). \tag{1.18}$$

Podemos escrever matricialmente os componentes Cartesianos do tensor (1.18) como

$$[\nabla\mathbf{u}] = \begin{bmatrix} \dfrac{\partial u_x}{\partial x} & \dfrac{\partial u_y}{\partial x} & \dfrac{\partial u_z}{\partial x} \\ \dfrac{\partial u_x}{\partial y} & \dfrac{\partial u_y}{\partial y} & \dfrac{\partial u_z}{\partial y} \\ \dfrac{\partial u_x}{\partial z} & \dfrac{\partial u_y}{\partial z} & \dfrac{\partial u_z}{\partial z} \end{bmatrix}. \tag{1.19}$$

O operador ∇ pode também aparecer à direita de um vetor $\mathbf{u}$. Nesse caso, utilizaremos o símbolo $\overleftarrow{\nabla}$ para denotar o tensor

$$\mathbf{u}\overleftarrow{\nabla} = (u_k \mathbf{i}_k)(\frac{\partial}{\partial x_m}\mathbf{i}_m) = u_{k,m}\mathbf{i}_k\mathbf{i}_m. \tag{1.20}$$

(A seta indica que o operador ∇ atua sobre a quantidade precedente. Quando o operador atua à direita, como na Equação (1.18), podemos omitir a seta.) Matricialmente, os componentes Cartesianos do tensor são

$$[\mathbf{u}\overleftarrow{\nabla}] = \begin{bmatrix} \frac{\partial u_x}{\partial x} & \frac{\partial u_x}{\partial y} & \frac{\partial u_x}{\partial z} \\ \frac{\partial u_y}{\partial x} & \frac{\partial u_y}{\partial y} & \frac{\partial u_y}{\partial z} \\ \frac{\partial u_z}{\partial x} & \frac{\partial u_z}{\partial y} & \frac{\partial u_z}{\partial z} \end{bmatrix}. \tag{1.21}$$

Finalmente, o divergente de um tensor de segunda ordem $\mathbf{T}$ é um vetor definido como

$$\begin{aligned} \overrightarrow{\nabla} \cdot \mathbf{T} &= (\frac{\partial}{\partial x_k}\mathbf{i}_k) \cdot (T_{pq}\,\mathbf{i}_p\mathbf{i}_q) = \frac{\partial T_{pq}}{\partial x_p}\mathbf{i}_q, \quad \text{ou} \\ \mathbf{T} \cdot \overleftarrow{\nabla} &= (T_{pq}\,\mathbf{i}_p\mathbf{i}_q) \cdot (\frac{\partial}{\partial x_k}\mathbf{i}_k) = \frac{\partial T_{pq}}{\partial x_q}\mathbf{i}_p, \end{aligned} \tag{1.22}$$

dependendo da ordem de atuação do operador ∇.

Programas Escreveremos trechos de programas em `courier`. Palavras reservadas de C e C++ serão escritas em **`courier negrito`**. Comentários aparecerão em *`courier itálico`*, como exemplificado no pequeno programa abaixo.

```
#include <stdio.h>
void main()
{
   // nosso primeiro programa
   puts("Bem-vindo a OSW! Boa leitura.");
}
```

CAPÍTULO 2

O que são Modelos?

2.1 Introdução

No Capítulo 1, procuramos justificar que os campos de deslocamentos, de deformações e de tensões de um sólido só podem ser aproximadamente determinados através de representações simplificadas da estrutura e do comportamento desse sólido. Chamamos essas representações simplificadas 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 literatura, encontramos conceitos similares para modelos. MÄNTYLÄ [74], por exemplo, define um modelo como sendo

> "um objeto construído artificialmente que torna a observação de outro objeto mais fácil. Para tornar a observação possível, *modelos físicos* de objetos tridimensionais tais como edifícios, naves e carros, usualmente compartilham as dimensões relativas e a aparência geral do objeto físico correspondente, mas não o tamanho. (...) *Modelos matemáticos*, amplamente utilizados em vários campos da ciência e engenharia, representam alguns dos aspectos comportamentais de fenômenos modelados em termos de dados numéricos e equações."

VAN DAM e SKILAR [38] afirmam que

> "modelos quantitativos comuns em ciências e engenharia são usualmente expressos como sistemas de equações e o modelador, experimentalmente, varia os valores das variáveis independentes, coeficientes e expoentes. Muitas vezes, os modelos simplificam a verdadeira estrutura ou funcionamento da entidade modelada, para tornar o modelo mais fácil de visualizar, ou para aqueles modelos representados por sistemas de equações, tornar o modelo tratável computacionalmente."

Nesse Capítulo faremos uma discussão generalizada sobre modelos em engenharia de estruturas, especificamente sobre *modelos computacionais*. Um modelo computacional é um modelo cuja estrutura pode ser armazenada em memória de computador e cujo comportamento pode ser determinado através de um programa de computador, ou *aplicação de modelagem*. Na Seção 2.2, trataremos da modelagem geométrica de estruturas. Na Seção 2.3, discutiremos o processo de modelagem matemática de um problema físico, particularmente o problema fundamental. Na Seção 2.4, abordaremos a modelagem mecânica. Na Seção 2.5, caracterizaremos a estrutura e a funcionalidade de um sistema computacional de modelagem.

Exemplo

Para ilustrarmos nossas discussões de modelagem geométrica, matemática e mecânica, vamos considerar, como exemplo, o sólido mostrado na Figura 2.1, definido por um volume Ω e uma superfície Γ. Nosso objetivo é determinar o comportamento do objeto antes de sua construção, ou seja, os deslocamentos, deformações e tensões em qualquer ponto do sólido, quando submetido aos carregamentos indicados na figura.

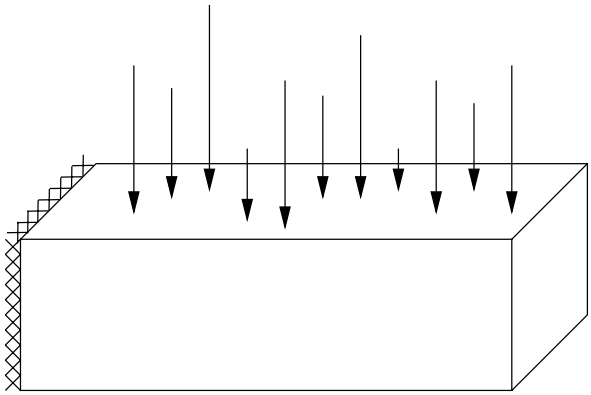


Figura 2.1: Qual é o comportamento desse sólido?

2.2 Modelagem Geométrica

A modelagem geométrica envolve o uso do computador para auxiliar a criação, manipulação, manutenção e análises de representações da forma geométrica de objetos bidimensionais e tridimensionais. Podemos distinguir diferentes formas de modelagem geométrica.

- *Modelagem fio-de-arame.* A representação de objetos é efetuada através de arestas e pontos sobre a superfície do objeto. A representação fio-de-arame do sólido da Figura 2.1 é definida por oito pontos, os "cantos" da estrutura, e por doze arestas retas que unem esses pontos. A representação não nos permite, por exemplo, a remoção das linhas escondidas da figura. Não podemos determinar quais arestas estão "escondidas" por alguma superfície porque, simplesmente, não há no modelo quaisquer informações sobre as superfícies que constituem o sólido.

- *Modelagem de superfícies.* A representação de objetos é baseada na descrição matemática da forma das superfícies dos objetos. Em um modelo de superfícies,

o sólido da Figura 2.1 é representado por seis faces planas, sendo cada face definida por quatro pontos. A representação permite o emprego de técnicas mais sofisticadas de geração de imagens, mas usualmente não oferece dados suficientes para verificação da integridade do modelo. Por exemplo, podemos afirmar, prontamente, que um conjunto de seis faces representam um objeto sólido?

- *Modelagem de sólidos.* A representação contém, explícita ou implicitamente, informações a respeito do fecho e conectividade dos volumes de formas sólidas. Um modelo de sólidos, diferentemente dos modelos de superfície, 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, tais como centro de gravidade e momentos de inércia. O modelo de sólidos do exemplo da Figura 2.1 pode ser definido por seis faces planas e oito vértices, como antes, *mais* um conjunto de informações sobre a adjacência desses componentes. Esse conjunto de informações é necessário para garantir, por exemplo, que em cada uma das doze arestas do modelo incidam sempre duas faces, de tal modo que o conjunto de faces do modelo represente a superfície Γ de Ω. Um modelo de sólidos é dito ser uma representação de variedade de dimensão 2, ou *2-manifold*. (Explicaremos o que é uma representação de variedade de dimensão 2 a seguir.)

- *Modelagem geométrica não-manifold.* Esse tipo de modelagem remove muitas das restrições associadas com a modelagem de sólidos *2-manifold* porque engloba todas as capacidades das três formas de modelagem vistas anteriormente em uma representação unificada, estendendo o domínio de representação de objetos.

Em uma representação de variedade de dimensão 2, ou *2-manifold* (*manifold* para abreviar), cada ponto sobre a superfície do sólido é bidimensional, ou seja, cada ponto tem uma vizinhança homeomórfica a um disco bidimensional. 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. O objeto de nosso exemplo é *manifold*. Usaremos o termo *não-manifold* em modelagem geométrica quando nos referirmos a situações topológicas que não são *manifold*, como ilustrado pelo sólido da Figura 2.2.

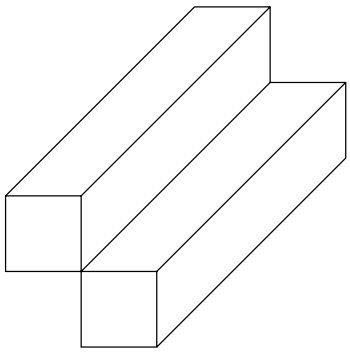


Figura 2.2: Exemplo de sólido *não-manifold*.

Observe que na aresta que une os dois blocos incidem quatro faces. Como uma aresta, teoricamente, só possui uma dimensão, esse sólido não pode existir no mundo real. No entanto, em muitos modelos de engenharia temos exatamente essa situação, como é o caso, por exemplo, de estruturas constituídas de pórticos e cascas delgadas.

2.2.1 Geometria e Topologia

Podemos considerar que a geometria representa essencialmente todas as informações a respeito da forma geométrica de um objeto e a precisa localização espacial de todos os seus componentes.

De acordo com WEILER [127], *topologia* é, "por definição, uma abstração, um subconjunto de todas as informações geométricas de um objeto. Mais formalmente, um conjunto de propriedades invariantes em relação a determinado conjunto de transformações geométricas." A invariância dessas propriedades em relação a transformações implica que as propriedades representadas pela topologia não incluem informações que possam ser modificadas pelas transformações. Portanto, não temos todas as informações geométricas na topologia. A topologia é uma informação geométrica incompleta que pode ser teoricamente derivada da especificação geométrica completa.

Consideremos o sólido prismático da Figura 2.1. Se aplicarmos ao sólido translações ou rotações arbitrárias, ou seja, movimentos de corpo rígido, o objeto mudará de posição, mas continuará sendo um prisma. Ainda teremos cada ponto sendo compartilhado por três arestas, cada aresta sendo compartilhada por duas faces, cada face conectada com outra de modo a formar a superfície externa do sólido. Dizemos que nosso objeto é topologicamente equivalente a um cubo, isto é, possui oito vértices, doze arestas e seis faces apropriadamente conectados.

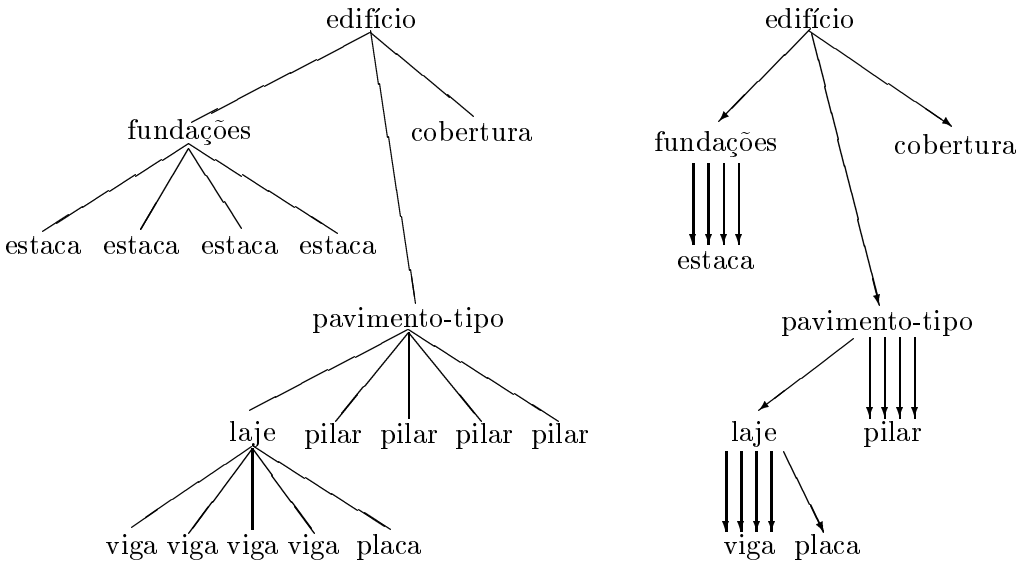
No contexto da modelagem geométrica, utilizaremos o termo topologia como significado de *relações de adjacência* entre elementos topológicos tais como vértices, arestas e faces. Uma relação de adjacência é a adjacência, em termos de proximidade física e ordem, de um grupo de elementos topológicos de um tipo ao redor de um único elemento topológico de outro tipo. Um exemplo de relação de adjacência é o grupo de arestas incidentes em um vértice de um modelo *manifold*.

A seguir, veremos algumas alternativas de modelagem geométrica de objetos, discutindo, com um pouco mais de detalhes, alguns aspectos de modelos de sólidos baseados em representações por fronteira. Antes, porém, veremos o que é uma hierarquia em modelos geométricos. No Capítulo 3 definiremos as representações geométricas de nossos modelos estruturais.

2.2.2 Hierarquias em Modelos Geométricos

Os modelos geométricos muitas vezes têm uma estrutura hierárquica induzida por um processo de construção *"bottom-up"*: componentes são usados como blocos básicos para criar entidades mais complexas, as quais, por sua vez, são usadas para construção de outras entidades, de nível ainda mais alto. Em grandes sistemas, no entanto, hierarquias raramente são construídas estritamente *"bottom-up"* ou *"top-down"*; o que interessa realmente é a hierarquia final e não o processo de construção.

No caso incomum em que cada objeto é incluído apenas uma vez em um objeto de nível mais alto, a hierarquia pode ser simbolizada como uma *árvore*, cujos nós representam os objetos e as arestas representam as relações de inclusão entre objetos. Nos casos mais comuns de objetos incluídos múltiplas vezes, a hierarquia é melhor simbolizada por um *grafo dirigido acíclico* (GDA). A estrutura simplificada de um edifício pode ser utilizada como exemplo de uma hierarquia de objetos. A Figura 2.3 mostra diagramaticamente a representação dos componentes da estrutura.



(a)Árvore (b)GDA

Figura 2.3: Hierarquia dos componentes de uma estrutura simples.

A estrutura do exemplo é composta pelas estruturas de fundações, pavimento-tipo e cobertura. As fundações, por sua vez, são definidas por quatro estacas, e o pavimento-tipo, por quatro pilares e uma laje, constituída por uma placa apoiada em quatro vigas.

Para simplificar a tarefa de construir objetos complexos (e seus modelos), comumente utilizamos componentes atômicos específicos à aplicação como blocos básicos de construção. Em 2D, estes componentes são, geralmente, desenhados a partir de representações das formas de conjuntos definidos de símbolos. Estas formas, por sua vez, são compostas de *primitivos geométricos* tais como linhas, retângulos, polígonos, arcos de elipse e de circunferência e assim por diante. Em 3D, os blocos básicos de construção podem ser formas tais como cilindros, paralelepípedos, esferas, pirâmides e superfícies de revolução. Estas formas 3D podem ser definidas em termos de primitivos baseadas em polígonos 3D; neste caso, superfícies curvas devem ser aproximadas por faces poligonais, com alguma perda de resolução. Alternativamente, em sistemas avançados de modelagem que manipulam diretamente superfícies e volumes genéricos, formas como superfícies paramétricas polinomiais e sólidos como cilindros, esferas e cones, são eles próprios primitivos geométricos, definidos analiticamente sem perda de resolução. VAN DAN, em FOLEY [38], usa o termo *objeto*[1]

> "para aqueles componentes 2D e 3D que são definidos em seus próprios sistemas de coordenadas de modelagem, em termos de primitivos geométricos e objetos de nível mais baixo e que, freqüentemente, possuem não apenas dados geométricos, mas também dados de aplicação associados."

Uma hierarquia, portanto, é criada para uma variedade de propósitos:

- construir objetos complexos de maneira modular, tipicamente por instâncias repetitivas de blocos básicos de construção, que variam em atributos geométricos e de aparência;

[1] Observe como o termo objeto é utilizado distintamente, dependendo do contexto e da disciplina.

- proporcionar economia de armazenamento, sendo suficiente armazenar apenas referências a objetos que são usados repetitivamente, ao invés da completa definição do objeto;

- permitir fácil propagação de modificações, uma vez que uma mudança na definição de algum bloco básico é automaticamente propagada para todos os objetos de mais alto nível que usam tal bloco.

Atualmente, vários sistemas de POO, tais como Smalltalk [47], estão cada vez mais sendo utilizados para codificar a hierarquia de modelos e armazenar informações de modelagem para os objetos geométricos em aplicativos compugráficos [75].

2.2.3 Modelagem de Sólidos

A modelagem de sólidos [74] 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.

O objetivo de aplicação geral separa a modelagem de sólidos de outros tipos de modelos geométricos, destinados a alguns propósitos especiais. *Modelos gráficos* destinam-se a descrever o desenho de um objeto ao invés do próprio objeto, constituindo-se em coleções não estruturadas de elementos de uma imagem, ou possuindo alguma estrutura interna para auxiliar, por exemplo, em operações de processamento de imagens. *Modelos de superfície* fornecem informações detalhadas sobre uma superfície curva, mas geralmente não possuem informações suficientes para determinação de todas as propriedades geométricas do objeto delimitado pela superfície.

A exigência de aplicação geral requer algumas propriedades dos modelos de sólidos, obtidas a partir de uma visão mais rigorosa de modelagem [96]. Essa visão é baseada na distinção entre três níveis de modelagem, conforme a Figura 2.4.

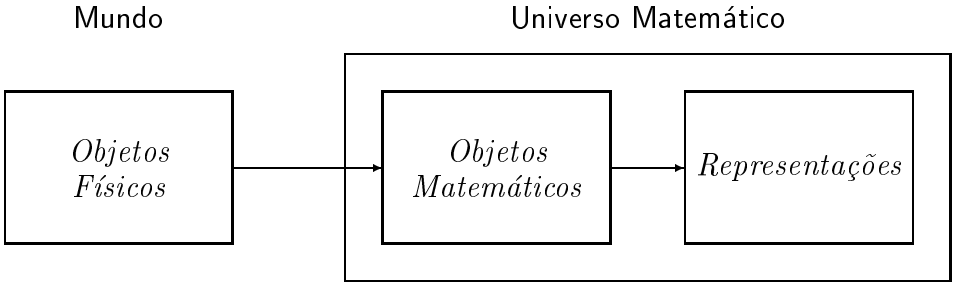


Figura 2.4: Uma visão em três níveis de modelagem.

1. *Objetos Físicos.* Através de modelos, intencionamos descobrir algumas coisas a respeito do mundo real tridimensional, embora não possamos perceber um objeto real em sua total complexidade e em seus detalhes microscópicos e muito menos representar todos seus aspectos em um computador.

2. *Objetos Matemáticos.* Devemos, portanto, adotar uma idealização apropriada dos objetos físicos reais de interesse. Estes objetos idealizados devem possuir uma conexão clara com o mundo real e serem simples o bastante tal que possamos atribuir uma representação computadorizada para eles.

3. *Representações.* O passo final da atividade de modelagem é atribuir ao objeto matemático uma representação que seja conveniente para manipulação computacional. Uma definição formal de representação, bem como maiores detalhes a respeito de modelos matemáticos de sólidos, suas características e propriedades, podem ser encontrados em MÄNTYLÄ [74].

REQUICHA [96] lista algumas propriedades desejáveis em esquemas de representação de sólidos:

- O *domínio de representação* necessita ser amplo o suficiente para permitir que um conjunto conveniente de objetos físicos possa ser representado.
- A representação precisa ser, idealmente, *não-ambígua*; não deve haver dúvida sobre o que está sendo representado e uma dada representação deve corresponder a um e somente um sólido. Uma representação não-ambígua é dita ser *completa*, ou integral.
- Uma representação é *única* se puder ser usada para codificar qualquer sólido de somente uma maneira. Se uma representação garante a imparidade, operações tais como o teste de igualdade de objetos são executadas mais facilmente.
- Uma representação *exata* permite que um objeto seja representado sem aproximações.
- Idealmente, um esquema de representação deve tornar impossível a criação de uma representação inválida (isto é, que não corresponda a um sólido), bem como fornecer ferramentas para criar adequadamente uma representação válida, tipicamente com a ajuda de um sistema interativo de modelagem de sólidos.
- Objetos sólidos devem manter *fecho* sob rotação e translação: a aplicação de tais operações sobre objetos válidos deve produzir apenas objetos válidos.
- A representação deve ser *compacta* para poupar espaço e, finalmente,
- Uma representação deve permitir o uso de algoritmos eficientes para geração de imagens e para computação das propriedades físicas e comportamentais desejadas.

Além das dificuldades impostas por tais propriedades, existem muitos aspectos práticos que tornam a construção de um modelador de sólidos uma tarefa não-trivial, como se pode observar pela análise dos componentes funcionais de um modelador, Figura 2.5.

Inicialmente, os objetos são descritos para o modelador em termos de uma linguagem de descrição, baseada nos conceitos de modelagem avaliados no modelador de sólidos, textualmente ou, de preferência, através de uma interface interativa gráfica com o usuário. As descrições dos objetos são, em seguida, traduzidas para se obter as representações internas armazenadas pelo modelador. A relação entre a linguagem de descrição e as representações internas não necessita ser direta: as representações internas podem empregar conceitos de modelagem diferentes daqueles da descrição original. Além disso, um modelador de sólidos pode dispor de diversas linguagens de descrição, dirigidas para diferentes categorias de usuários e aplicações.

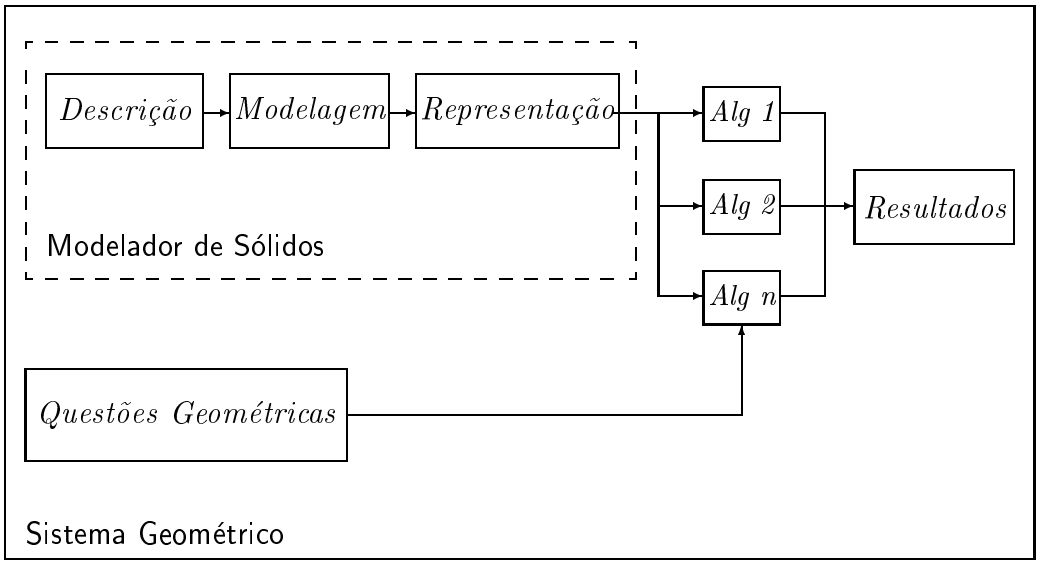


Figura 2.5: Componentes funcionais de um modelador de sólidos.

O modelador deve possuir interfaces para comunicação com outros módulos. Estas interfaces são usadas para transmitir informações para vários algoritmos ou, eventualmente, até mesmo os modelos sólidos completos para outros sistemas de análise. O modelador também deve incluir facilidades para armazenamento das descrições dos objetos e outros dados, em bases de dados permanentes.

Aplicação geral significa capacidade de fornecer, algoritmicamente, respostas para questões típicas em aplicações de engenharia, tais como:

1. Qual é a aparência do objeto?

2. Qual é o peso, área superficial, volume, etc. do objeto?

3. O objeto intercepta algum outro?

4. O objeto possui resistência suficiente para suportar determinado carregamento?

5. Como o objeto pode ser construído com certos processos construtivos disponíveis?

O resultado de uma questão geométrica pode ser uma imagem, um número simples ou uma constante *booleana*, bem como um outro modelo de sólido que representa o resultado de um cálculo, como a questão "qual é a deformação produzida por este carregamento aplicado ao objeto?" Claramente, é importante que um modelador geométrico inclua facilidades para modelar não apenas objetos físicos, mas também efeitos de processos físicos aplicados a eles. Naturalmente, o modelador deveria ser capaz de aplicar repetidamente tais operações aos resultados de operações anteriores, ou seja, as operações no modelador deveriam constituir um sistema fechado, garantindo a manutenção da exatidão dos modelos fundamentais.

A seguir, resumimos alguns esquemas de representação de sólidos mais utilizados atualmente. Formalismos e propriedades detalhadas sobre tais esquemas podem ser encontrados em MÄNTYLÄ [74].

Representação de Sólidos

Os esquemas de representação de sólidos podem ser divididos em três grandes classes:

- *Modelos de decomposição* representam o conjunto de pontos de um sólido como uma coleção de objetos simples, a partir de uma coleção fixa de tipos de objetos primitivos, combinados com uma operação simples de "colagem". *Instância de primitivos*, *enumeração exaustiva*, esquemas de *subdivisão de espaço* e *decomposição em células* são exemplos de modelos de decomposição.

- *Modelos construtivos* representam o conjunto de pontos de um sólido como uma combinação de conjuntos primitivos de pontos. Cada um destes primitivos é representado como uma instância de um tipo de sólido primitivo e os modelos incluem operações mais gerais de construção, tais como operações *booleanas* regularizadas [95]. Esse esquema de representação é denominado geometria construtiva de sólidos, ou *constructive solid geometry* (CSG) [94].

- *Modelos de contorno* representam 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. Portanto, modelos de contorno, ou *b-reps* (*boundary representations*), podem ser vistos como uma hierarquia de modelos.

Modelos de Contorno

Modelos de contorno representam objetos sólidos através da divisão de suas superfícies em uma coleção de *faces*. Freqüentemente, a divisão é efetuada de tal maneira que a forma de cada face possua uma representação matemática compacta [74], ou seja, que a face consista de uma única superfície planar, quadrática, toroidal ou paramétrica. Por sua vez, as curvas de contorno das faces são representadas por meio de uma divisão em *arestas*. Analogamente, desejamos também uma representação conveniente para as arestas, por exemplo, através de uma equação paramétrica. Os pontos de contorno da porção da curva que define uma aresta são denominados *vértices* da aresta.

As classes de objetos face, aresta e vértice, bem como as informações geométricas pertinentes a tais objetos, formam os constituintes básicos dos modelos de contorno. Além de informações geométricas como equações das faces e arestas, e coordenadas dos vértices, um modelo de contorno também deve representar as inter-relações entre faces, arestas e vértices. Comumente, o termo *geometria* de um modelo de contorno significa qualquer informação geométrica associada a uma entidade, enquanto *topologia* significa as interconexões de seus componentes.

Muitas alternativas de estruturas de dados capazes de representar a geometria e topologia de modelos de contorno têm sido propostas, dando origem a modelos de contorno dos tipos:

- *Baseados em polígonos*, também chamados *modelos poliedrais*. Nesses modelos, um sólido consiste de uma coleção de faces agrupadas, sendo as faces representadas por polígonos definidos pela seqüência de coordenadas de seus vértices. Esta representação é geralmente utilizada em meta-arquivos de sistemas gráficos.

- *Baseados em vértices.* Nos modelos baseados em polígonos a repetição de vértices em muitas faces significa a repetição de suas coordenadas. Esta redundância pode ser eliminada pela introdução dos vértices como entidades independentes na estrutura de dados do modelo. Neste caso, as faces são definidas como uma seqüência de *identificadores* (referências) de vértices.

- *Baseados em arestas.* Um modelo de contorno baseado em arestas representa o contorno de uma face em termos de uma seqüência fechada de arestas, ou *laços*, sendo cada aresta incluída explicitamente na estrutura de dados e definida por seus vértices e por sua orientação. Desta maneira, as faces são consistentemente orientadas e cada aresta ocorre exatamente em duas faces, em uma, orientada positivamente e, em outra, negativamente.

Em particular, a *estrutura de dados aresta alada* (*"winged-edge data structure"*) é um modelo de contorno baseado em arestas mais elaborado, acrescido de informações explícitas de vizinhança face-face. Essa estrutura, primeiramente introduzida por BAUMGART [9], facilita o processamento de algoritmos tais como remoção de linhas escondidas e *"shading"*. MÄNTYLÄ [74], por sua vez, apresenta uma estrutura de dados denominada *estrutura de dados semi-aresta* (*"half-edge data structure"*), uma variação da estrutura de dados aresta alada capaz de representar modelos não intuitivos necessários para o registro de todos os passos intermediários da seqüência de descrição de um modelo de sólido. Utilizaremos essa estrutura no Capítulo 3.

Validade de Modelos de Contorno Um modelo de contorno é *válido* se define o contorno de um sólido de variedade de dimensão 2. Os critérios de validade de um modelo de contorno incluem as seguintes condições:

1. O conjunto de faces de um modelo de contorno é topologicamente fechado;

2. As faces de um modelo não se interceptam, exceto em arestas e vértices comuns;

3. Os contornos das faces são superfícies únicas que não se interceptam.

A *integridade topológica* do modelo é necessária e suficientemente atendida [74] se a fórmula de Euler-Poincaré

$$v - e + f = 2(s - h) \tag{2.1}$$

é satisfeita para o modelo. Na Equação (2.1), 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 (*"shells"*) e h é o *gênero* da superfície (número de cavidades). Essa condição de integridade topológica pode ser garantida se exigirmos que cada aresta ocorra exatamente em duas faces. MÄNTYLÄ [73] apresenta um modelador de sólidos cuja estrutura baseada em arestas é construída a partir de operações elementares de tal forma que o modelo sempre satisfaça a Equação (2.1). Os comandos de tais operações são denominados *operadores de Euler* [74].

A *integridade geométrica* de um modelo de contorno, contudo, não pode ser garantida somente por meios estruturais, ou seja, é possível criarmos modelos inválidos a partir de informações geométricas inadequadas. Geralmente, o conjunto de mecanismos de descrição de sólidos oferecidos pelos modeladores, tais como *varredura de primitivos* (*"sweep primitive"* [38]) e *operações booleanas regularizadas*, garantem a validade geométrica (e topológica) dos objetos.

Propriedades dos Modelos de Contorno

Validade: o critério de validade é baseado em restrições topológicas e geométricas, como discutido anteriormente. Os testes para garantia de exatidão geométrica podem ser penalizados em velocidade, em projetos interativos.

Não-ambiguidade: modelos de contorno não são ambíguos.

Unicidade: modelos de contono não são únicos.

Concisão: os modelos podem se tornar bastante extensos, especialmente se objetos curvos são aproximados com modelos poliedrais.

Domínio de representação: depende da coleção de superfícies que podem ser usadas; modelos de contorno podem ser empregados para representar objetos de um domínio mais geral que CSG.

2.3 Modelagem Matemática

Matematicamente, o comportamento do sólido da Figura 2.1 pode ser expresso pela seguinte equação diferencial:

$$G\,\nabla^2\mathbf{u} + \frac{G}{1-2\nu}\,\nabla(\nabla\cdot\mathbf{u}) + \mathbf{b} = \mathbf{0}. \tag{2.2}$$

A Equação (2.2) é conhecida como Equação de Navier-Cauchy da elasticidade, e expressa o equilíbrio estático de um sólido em função do campo de deslocamentos $\mathbf{u}$ e das forças de volume $\mathbf{b}$ aplicadas ao sólido. As constantes G e ν são propriedades do material do objeto, módulo de elasticidade transversal e coeficiente de Poisson, respectivamente. No Capítulo 4 retornaremos a essa equação. Por ora, apenas consideremos que a unicidade da solução da Equação (2.2) depende das condições de contorno

$$\mathbf{p} = \overline{\mathbf{p}} \qquad \text{sobre } \Gamma_2, \text{ e} \tag{2.3}$$

$$\mathbf{u} = \overline{\mathbf{u}} \qquad \text{sobre } \Gamma_1, \tag{2.4}$$

onde $\overline{\mathbf{p}}$ são as forças de superfície e $\overline{\mathbf{u}}$ são os deslocamentos conhecidos na superfície do contorno $\Gamma = \Gamma_1 + \Gamma_2$ do sólido.

Se admitirmos que o comportamento da estrutura da Figura 2.1 seja definido pela Equação (2.2), então estaremos também admitindo as seguintes hipóteses simplificadoras:

- o material que constitui a estrutura é contínuo, homogêneo, isótropo e perfeitamente elástico, e
- os deslocamentos $\mathbf{u}$ e as deformações $\boldsymbol{\epsilon}$ são "pequenos."[2]

A equação diferencial de Navier é derivada da teoria da mecânica do contínuo. A teoria foi desenvolvida, primeiro, por *observação* dos fenômenos naturais. Na tentativa de compreender a complexidade da natureza, isolamos componentes individuais e

[2]No Capítulo 4 daremos uma noção mais precisa desses conceitos.

adotamos, como resultado desse processo de abstração, suposições aproximadas sobre o comportamento dos componentes. As *hipóteses* citadas acima são algumas dessas suposições, as quais devem ser sempre comprovadas pela *experimentação*. Posteriormente, estabelecemos *princípios* descritos por equações matemáticas que, finalmente, constituem a *teoria*.

Caracterizaremos um modelo matemático como sendo um conjunto de equações diferenciais que governam quantitativamente o comportamento de determinado objeto. No Capítulo 4 descreveremos os modelos matemáticos considerados no trabalho.

2.4 Modelagem Mecânica

A solução da Equação (2.2) com condições de contorno dadas pelas Equações (2.3) e (2.4) pode ser ser obtida numericamente em computador através de técnicas numéricas tais como o método dos elementos finitos (MEF) ou o método dos elementos de contorno (MEC). Na verdade, para geometrias e condições de contorno quaisquer, a solução *só* pode ser determinada com o emprego dos métodos numéricos. Em engenharia de estruturas, o MEF e o MEC são os métodos mais usualmente empregados. Consideremos, como ilustração para nosso exemplo, a utilização do método dos elementos de contorno.

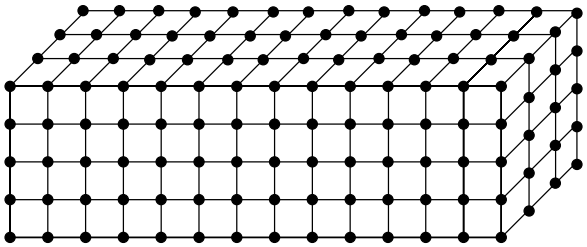


Figura 2.6: Discretização do sólido em elementos de contorno.

Uma solução aproximada é determinada, com o MEC, a partir da subdivisão do contorno do objeto em células, ou elementos, como ilustrado na Figura 2.6 para o sólido da Figura 2.1. Note que os elementos de contorno, nesse caso, possuem dimensionalidade topológica 2, ou seja, são elementos de superfície. A posição espacial de um elemento é definida pelas coordenadas de pontos discretos denominados *nós*. 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 denominadas *funções de forma*. O comportamento de um elemento é definido por funções interpoladoras que descrevem as variações dos deslocamentos, deformações e tensões sobre o elemento, a partir dos valores nodais dessas grandezas. O comportamento de todo o sólido é determinado a partir das contribuições individuais de todos os elementos. No Capítulo 5 trataremos do método dos elementos de contorno.

A definição do modelo mecânico começa com a especificação dos casos de carregamento, das condições iniciais e das condições de contorno impostos à estrutura, em adição às informações geométricas e topológicas do modelo geométrico. Depois disso, definimos os parâmetros necessários à discretização do objeto em uma malha de elementos, necessária para a etapa de análise. Apresentaremos as formulações dos modelos mecânicos de cascas elásticas e sólidos elásticos no Capítulo 6.

2.5 Sistemas Computacionais de Modelagem

Quase todos os sistemas gráficos interativos de modelagem podem ser conceitualmente descritos através de três componentes, esquematicamente ilustrados na Figura 2.7: o *programa de aplicação*, o *modelo de aplicação* e o *sistema gráfico*.

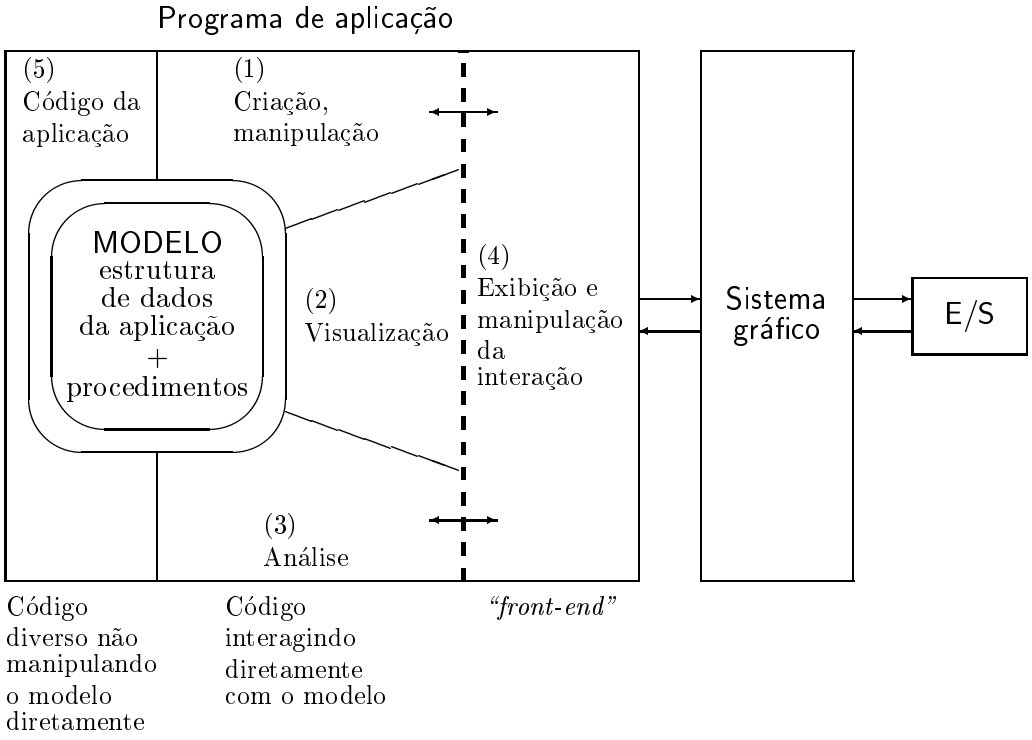


Figura 2.7: Componentes de um sistema gráfico de modelagem.

O programa de aplicação cria, armazena e recupera informações do modelo de aplicação, ou base de dados da aplicação, que representa os objetos que são manipulados pelo sistema. O programa de aplicação também captura os eventos de entrada do usuário e produz *visões* que são enviadas para o sistema gráfico, as quais contêm uma descrição geométrica detalhada do que será visto e atributos que especificam a aparência dos objetos. O sistema gráfico é um intermediário entre o programa de aplicação e o *hardware* gráfico, transformando os objetos do modelo de aplicação em visões do modelo e, simetricamente, transformando as ações do usuário em entradas para o programa de aplicação. O objetivo do projetista de um programa gráfico interativo é especificar quais classes de objetos serão representadas e como o usuário e o programa de aplicação interagirão para criar e modificar os modelos e suas visões.

De acordo com FOLEY *et alli* [38], um programa de modelagem compugráfica deve possuir, genericamente, capacidades para

1. Criar, modificar e armazenar o modelo, adicionando, eliminando e modificando suas informações.

2. Extrair do modelo informações para visualização.

3. Extrair do modelo informações usadas para análise de seu comportamento.

4. Permitir visualização, tanto de informações e de resultados do processamento de informações do modelo (*"rendering"* de um modelo geométrico ou resultados da análise estrutural, por exemplo) como de ferramentas da interface com o usuário (*menus* e caixas de diálogo, por exemplo), e

5. Executar diferentes tarefas da aplicação que não envolvem diretamente o modelo ou a visualização.

Na segunda parte do texto apresentaremos OSW, um *toolkit* orientado a objetos que pode nos auxiliar no desenvolvimento de sistemas gráficos de modelagem para Windows NT, com as características citadas acima.

2.6 Sumário

Nesse Capítulo discutimos generalidades sobre modelagem geométrica, matemática e mecânica de estruturas.

Modelos geométricos são representações de todas as informações que definem a forma e o posicionamento espacial de um objeto e de seus componentes, usualmente organizados em uma estrutura hierárquica. Topologia é um subconjunto das informações geométricas que representa propriedades invariantes em relação a transformações geométricas. Para nossos propósitos, topologia significa adjacência entre componentes tais como vértices, arestas e faces.

Modelos matemáticos são equações diferenciais que descrevem o comportamento de determinado objeto, obtidas a partir de hipóteses simplificadoras sobre o comportamento do objeto. Nossos modelos matemáticos serão baseados na teoria da mecânica do contínuo.

Modelos mecânicos são definidos geometricamente por um conjunto de elementos discretos do volume e/ou superfície do objeto original, mais um conjunto de condições de contorno e casos de carregamento. A partir de um modelo mecânico de uma estrutura podemos "resolver" numericamente as equações do modelo matemático. Usaremos o método dos elementos finitos e o método dos elementos de contorno.

CAPÍTULO 3

Modelos Geométricos

3.1 Introdução

Modelos geométricos são representações das características geométricas de um objeto que desejamos construir. Vimos, no Capítulo 2, que há uma variedade de representações possíveis, e que a definição de uma representação depende de quais operações serão realizadas sobre o modelo. Por exemplo, em um modelo de contorno, se necessitarmos perguntar muitas vezes quais são as faces que incidem em um determinado vértice, será mais eficiente, em termos de velocidade de processamento, armazenar diretamente essas informações de adjacência na estrutura de dados do modelo. Evidentemente, o armazenamento dessas informações diretamente na estrutura de dados do modelo resultará em uma representação que consome mais memória do computador. Esse conflito espaço *versus* tempo é uma constante nos projetos de modelos baseados em computador. Mais uma vez, o que decidirá a favor de uma representação, em detrimento de outra, são os propósitos do modelo.

Neste Capítulo discutiremos as técnicas de modelagem utilizadas na representação geométrica de nossos objetos estruturais. O propósito é descrever as estruturas de dados e as principais operações de construção e destruição dos componentes dos modelos geométricos empregados em OSW. Descreveremos, também, as principais operações de transformação de um modelo geométrico, iniciando na Seção 3.2 com a apresentação da formulação homogênea de algumas *transformações geométricas* bastante úteis: translações, transformações de escala, rotações e quaisquer composições dessas transformações. Algoritmos de visualização e de análise e operações tais como o armazenamento e recuperação dos dados dos modelos em arquivos serão vistos nos capítulos subseqüentes.

Começaremos nossas discussões sobre modelos geométricos com *modelos gráficos* bem simples, Seção 3.3. Um modelo gráfico pode ser utilizado para representar desenhos de objetos que podem ser descritos por figuras geométricas tais como pontos, linhas e polígonos. Chamaremos essas figuras geométricas de *primitivos*, e definiremos um modelo gráfico como uma coleção de primitivos que podem ser hierarquicamente agrupados, como descrito no Capítulo 2. Modelos gráficos serão úteis, por exemplo, na representação geométrica de isolinhas e isosuperfícies, como veremos no Capítulo 7. Um modelo gráfico deve ser utilizado somente quando o *desenho* de um objeto é rele-

vante. As informações armazenadas no modelo não são suficientes para operações mais complexas sobre o objeto. Se um conjunto de linhas do modelo representar as arestas de um sólido, por exemplo, não podemos determinar diretamente qual o volume ou a área superficial do sólido, ou quais são as faces que incidem em determinada aresta, pois não temos disponíveis quaisquer dados de adjacência no modelo.

Na Seção 3.4, proporemos um modelo geométrico que contém informações explícitas sobre determinadas relações de adjacência entre seus componentes. Utilizaremos esses modelos na representação de estruturas constituídas de cascas analisadas pelo método dos elementos finitos, conforme exemplificaremos no Capítulo 11. (Lembremos, do Capítulo 1, que nos propusemos a demonstrar a utilização de OSW com uma aplicação de análise de cascas pelo MEF.) Chamaremos esses modelos de *modelos de cascas*. Definiremos as informações de adjacência de um modelo de cascas em função das técnicas de geração de malhas e de visualização que apresentaremos no Capítulo 6 e no Capítulo 7, respectivamente.

Na Seção 3.5, descreveremos *modelos de sólidos* de variedade de dimensão 2, ou *2-manifold*, baseados em uma representação por fronteira introduzida por MÄNTYLÄ [74]. A representação é definida por uma estrutura de dados denominada *estrutura de dados semi-aresta* e por uma coleção de operadores, os *operadores de Euler*, responsáveis pela manipulação da estrutura de dados de tal forma que a integridade topológica do modelo seja mantida. Utilizaremos modelos de sólidos na na representação geométrica de estruturas analisadas pelo método dos elementos de contorno, conforme veremos no Capítulo 11. (No Capítulo 1, também propomos demonstrar a utilização de OSW com uma aplicação de análise elastoestática de sólidos pelo MEC.)

Na Seção 3.6, descreveremos os *modelos de decomposição por células* utilizados na representação de malhas de elementos finitos e de contorno. Um modelo de decomposição por células é definido por uma coleção de elementos (finitos ou de contorno) genericamente denominados de *células*, "colados" entre si em pontos discretos chamados *nós*. Nesse Capítulo, apresentaremos a geometria das células; no Capítulo 6 definiremos seu comportamento mecânico.

3.2 Transformações Geométricas

É usual, em muitas operações geométricas em computação gráfica, representarmos pontos em *coordenadas homogêneas* [38]. Um ponto do espaço Euclidiano $\mathbb{R}^3$ com coordenadas Cartesianas (x, y, z) é representado, em coordenadas homogêneas, por uma quádrupla $[X, Y, Z, W]$, com $W \neq 0$, tais que $x = X/W$, $y = Y/W$ e $z = Z/W$. Segue, dessa definição, que todas as coordenadas homogêneas $[Wx, Wy, Wz, W]$, com $W \neq 0$, são representações distintas do mesmo ponto (x, y, z). (Tomaremos, para nossos propósitos, $W = 1$.) As coordenadas homogêneas $[1, 2, 3, 1]$ e $[3, 6, 9, 3]$, por exemplo, representam o ponto de coordenadas Cartesianas $(1, 2, 3)$. Não consideraramos os pontos onde $W = 0$.[1]

[1] O conjunto de todas as quádruplas de números reais $[X, Y, Z, W]$, exceto a quádrupla $[0, 0, 0, 0]$, define os pontos do *espaço projetivo orientado* $\mathbb{T}_3$, sendo que $[X, Y, Z, W]$ e $[\alpha X, \alpha Y, \alpha Z, \alpha W]$, com $\alpha > 0$, denotam o mesmo ponto. Geometricamente, $\mathbb{T}_3$ consiste de duas cópias do espaço Euclidiano $\mathbb{R}^3$, denominadas de *aquém* e *além*, mais um *ponto no infinito* para toda direção de $\mathbb{R}^3$. A quádrupla $[xW, yW, zW, W]$ representa, se $W > 0$, o ponto de coordenadas Cartesianas (x, y, z) no aquém; se $W < 0$, a quádrupla representa o mesmo ponto no além; se $W = 0$, a quádrupla representa o ponto infinito na direção do vetor (x, y, z).

A propriedade mais importante das coordenadas homogêneas é que toda transformação geométrica que leva um ponto (x, y, z) do espaço em um outro ponto (x', y', z') do espaço pode ser representada por uma matriz de transformação $\mathbf{M}$ quatro por quatro tal que

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \mathbf{M} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}. \tag{3.1}$$

A seguir, apresentaremos as matrizes de transformações que utilizaremos para modelagem geométrica.

Translações

A translação de um ponto (x, y, z) por um vetor (t_x, t_y, t_z) resulta um ponto (x', y', z') tais que

$$\begin{aligned} x' &= x + t_x, \\ y' &= y + t_y, \\ z' &= z + t_z. \end{aligned} \tag{3.2}$$

Se utilizarmos as coordenadas homogêneas $[x, y, z, 1]$ e $[x', y', z', 1]$, a transformação fica definida pela Equação (3.1), sendo a matriz de transformação dada por

$$\mathbf{T}(t_x, t_y, t_z) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{3.3}$$

Transformações de Escala

A transformação de escala de um ponto em torno da origem do sistema de coordenadas é definida pela matriz de transformação

$$\mathbf{S}(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{3.4}$$

onde os parâmetros s_x, s_y e s_z, todos não-nulos, são os fatores de escala em relação aos eixos x, y e z, respectivamente. Na Figura 3.1 é ilustrada a transformação de escala dos vértices de um cubo unitário para $s_x = 2$, $s_y = 1/2$ e $s_z = 1$.

A matriz de transformação de escala em torno de um ponto qualquer $P(p_x, p_y, p_z)$ pode ser obtida a partir da *combinação* da seguinte seqüência de transformações:

1. Translação de P para a origem.
2. Transformação de escala em torno da origem.
3. Translação tal que o ponto na origem retorne a P.

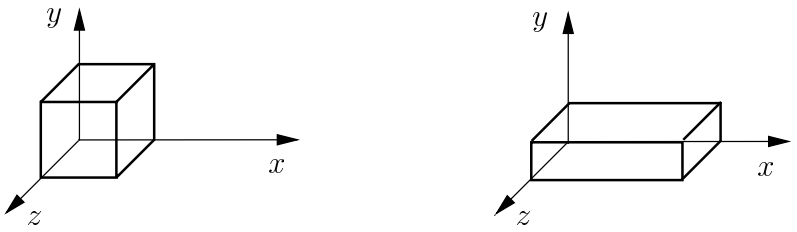


Figura 3.1: Transformação de escala em torno da origem.

Essa combinação de transformações, ilustrada na Figura 3.2 para o ponto $P(0,0,1)$ de um cubo unitário e $s_x = s_y = s_z = 2$, é expressa pelo produto das matrizes

$$\mathbf{T}(p_x, p_y, p_z) \cdot \mathbf{S}(s_x, s_y, s_z) \cdot \mathbf{T}(-p_x, -p_y, -p_z), \tag{3.5}$$

o qual define a matriz de escala em torno de um ponto qualquer

$$\mathbf{S}(P, s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & p_x(1-s_x) \\ 0 & s_y & 0 & p_y(1-s_y) \\ 0 & 0 & s_z & p_z(1-s_z) \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{3.6}$$

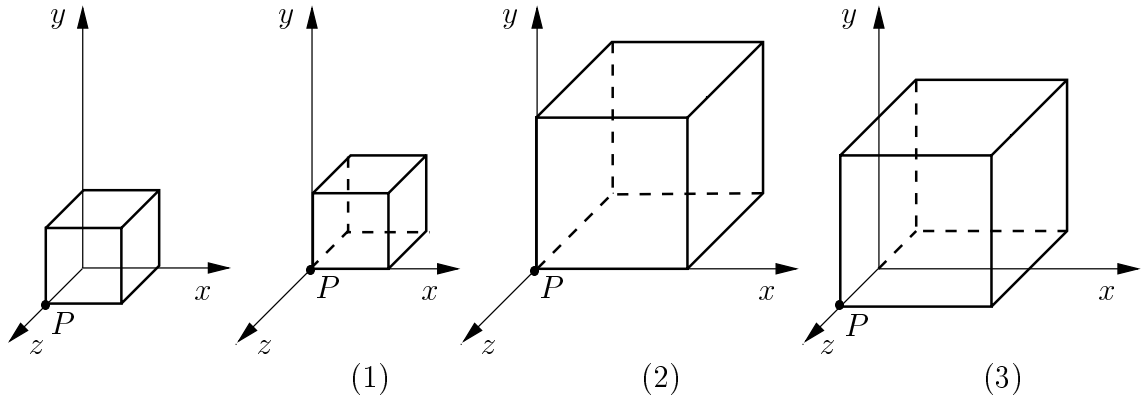


Figura 3.2: Transformação de escala em torno de um ponto qualquer.

Rotações

Antes de examinarmos a rotação de um ponto em torno de um eixo qualquer, consideraremos as rotações em torno dos eixos Cartesianos x, y e z. A matriz de rotação de um ponto em torno do eixo x é dada por [38]

$$\mathbf{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\operatorname{sen}\theta & 0 \\ 0 & \operatorname{sen}\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{3.7}$$

onde θ é o ângulo de rotação, considerado positivo se tomado de acordo com o sentido da regra da mão direita, aplicada no eixo de rotação. Na Figura 3.3(a) é ilustrada a rotação dos pontos de um cubo em torno do eixo x, para $\theta = 90^o$.

A rotação de um ângulo θ em torno do eixo y, ilustrada na Figura 3.3(b) para $\theta = 90^o$, é definida pela matriz de transformação

$$\mathbf{R}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \text{sen}\,\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\text{sen}\,\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{3.8}$$

Similarmente, a matriz de rotação de um ângulo θ em torno do eixo z, ilustrada na Figura 3.3(c) para $\theta = 90^o$, é dada por

$$\mathbf{R}_z(\theta) = \begin{bmatrix} \cos\theta & -\text{sen}\,\theta & 0 & 0 \\ \text{sen}\,\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.9}$$

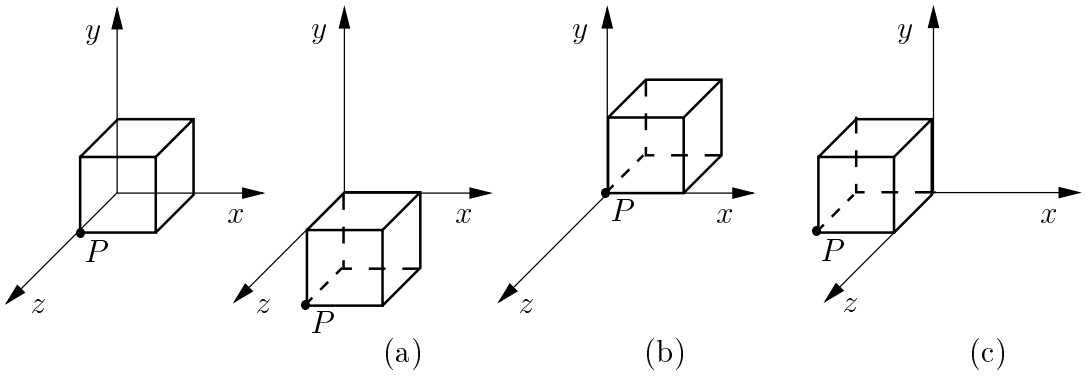


Figura 3.3: Rotações em torno dos eixos coordenados.

Consideremos, agora, a rotação de um ângulo θ em torno de um eixo qualquer que passa pela origem do sistema de coordenadas, definido por um vetor unitário $\mathbf{n}(n_x, n_y, n_z)$. Para obtermos a matriz de rotação utilizaremos, sem demonstrações, o conceito de *quaternion*, proposto por HAMILTON [49] em 1843.

Um quaternion $\mathbf{q}$ é uma estrutura algébrica constituída de duas partes: um escalar s e um vetor $\mathbf{r}(r_x, r_y, r_z)$, ou $\mathbf{q} = [s, \mathbf{r}]$. A multiplicação de dois quaternions $\mathbf{q}_1$ e $\mathbf{q}_2$ é definida como

$$\mathbf{q}_1\mathbf{q}_2 = [s_1, \mathbf{r}_1][s_2, \mathbf{r}_2] = [(s_1 s_2 - \mathbf{r}_1 \cdot \mathbf{r}_2), (s_1\mathbf{r}_2 + s_2\mathbf{r}_1 + \mathbf{r}_1 \times \mathbf{r}_2)]. \tag{3.10}$$

Um *quaternion unitário* é um quaternion onde $s^2 + r_x^2 + r_y^2 + r_z^2 = 1$. Uma rotação de um ângulo θ em torno do versor $\mathbf{n}$ é definida pelo quaternion unitário

$$\mathbf{q} = [s, \mathbf{r}] = [\cos(\theta/2), \text{sen}(\theta/2)\,\mathbf{n}]. \tag{3.11}$$

A rotação inversa $\mathbf{q}^{-1}$ é definida invertendo o sinal de s ou de $\mathbf{r}$ na Equação (3.11), mas não de ambos. Para rotacionar um ponto $P(x, y, z)$ por um quaternion $\mathbf{q}$, escrevemos o ponto P como o quaternion $\mathbf{p} = [0, (x, y, z)]$ e tomamos, de acordo com a regra definida na Equação (3.10), o produto

$$\mathbf{p}' = [0, (x', y', z')] = \mathbf{q}^{-1}\mathbf{p}\mathbf{q}, \tag{3.12}$$

onde $P'(x', y', z')$ é o ponto P rotacionado. Desenvolvendo a Equação (3.12), podemos escrever $P' = \mathbf{R}P$, onde $\mathbf{R}$ é uma matriz de rotação de um ângulo θ em torno de $\mathbf{n}$ cuja versão homogênea é dada por

$$\mathbf{R}(\mathbf{n}, \theta) = \begin{bmatrix} 1-2(r_y^2+r_z^2) & 2(r_x r_y - s r_z) & 2(r_x r_z + s r_y) & 0 \\ 2(r_x r_y + s r_z) & 1-2(r_x^2+r_z^2) & 2(r_y r_z - s r_x) & 0 \\ 2(r_x r_z - s r_y) & 2(r_y r_z - s r_x) & 1-2(r_x^2+r_y^2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{3.13}$$

com s e $\mathbf{r}$ definidos na Equação (3.11). Escolhemos a abordagem baseada em quaternions para derivar a matriz de rotação em torno de um eixo qualquer passando pela origem porque sua implementação em computador envolve um número menor de operações. (Veja, por exemplo, ROGERS [98] para uma derivação baseada em operações vetoriais.)[2]

A matriz de rotação de um ângulo θ em torno de um eixo qualquer $\mathbf{n}(n_x, n_y, n_z)$ passando por um ponto $P(p_x, p_y, p_z)$ distinto da origem é obtida, como no caso da escala em torno de um ponto qualquer, pela seguinte composição de transformações:

1. Translação de P para origem.
2. Rotação em torno do eixo $\mathbf{n}$ passando pela origem.
3. Translação tal que o ponto na origem retorne a P.

A matriz resultante é

$$\mathbf{R}(P, \mathbf{n}, \theta) = \begin{bmatrix} R_{11} & R_{12} & R_{13} & p_x - (R_{11}p_x + R_{12}p_y + R_{13}p_z) \\ R_{21} & R_{22} & R_{23} & p_y - (R_{21}p_x + R_{22}p_y + R_{23}p_z) \\ R_{31} & R_{32} & R_{33} & p_z - (R_{31}p_x + R_{32}p_y + R_{33}p_z) \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{3.14}$$

onde R_{ij}, $i = 1, 2, 3$, $j = 1, 2, 3$, é um elemento da matriz (3.13).

Uma outra rotação bastante útil é a *rotação de eixos*. Na Figura 3.4 são mostrados dois sistemas de coordenadas Cartesianas, ambos com origem no ponto O, definidos pelos eixos x_1, x_2, x_3 e u_1, u_2, u_3, respectivamente. Seja c_{ij} o cosseno diretor do eixo u_i em relação ao eixo x_j. A matriz de rotação que transforma as coordenadas x_1, x_2, x_3 nas coordenadas u_1, u_2, u_3 é [38, 101]

$$\mathbf{R}_c = \begin{bmatrix} c_{11} & c_{12} & c_{13} & 0 \\ c_{21} & c_{22} & c_{23} & 0 \\ c_{31} & c_{32} & c_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{3.15}$$

Inversamente, $\mathbf{R}_c^{-1} = \mathbf{R}_c^T$ ($\mathbf{R}_c$ é uma matriz ortogonal) transforma as coordenadas u_1, u_2, u_3 nas coordenadas x_1, x_2, x_3.

A Equação (3.1) pode ser facilmente implementada em C. Utilizaremos a estrutura `t3DVector` do Programa 3.1 para representar as coordenadas Cartesianas de um ponto no espaço.

[2] O resultado obtido é

$$\begin{bmatrix} n_x^2+(1-n_x^2)\cos\theta & n_x n_y(1-\cos\theta) - n_z \operatorname{sen}\theta & n_z n_x(1-\cos\theta) + n_y \operatorname{sen}\theta & 0 \\ n_x n_y(1-\cos\theta) + n_z \operatorname{sen}\theta & n_y^2+(1-n_y^2)\cos\theta & n_y n_z(1-\cos\theta) - n_x \operatorname{sen}\theta & 0 \\ n_z n_x(1-\cos\theta) - n_y \operatorname{sen}\theta & n_y n_z(1-\cos\theta) + n_x \operatorname{sen}\theta & n_z^2+(1-n_z^2)\cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

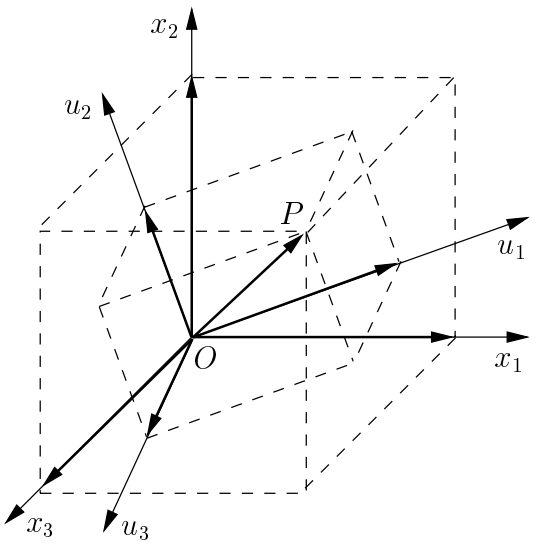


Figura 3.4: Rotação de eixos.

```
struct t3DVector
{
  double x;
  double y;
  double z;
}; // t3DVector
```

Programa 3.1: Definição de ponto no espaço.

A função de transformação é mostrada no Programa 3.2. Note que quaisquer composições das matrizes de transformação apresentadas anteriormente possuem a forma

$$\begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{3.16}$$

onde r_{ij} são termos que combinam rotação e transformação de escala e t_i são termos de translação.

```
typedef double t3DTransfMatrix[4][4];
void TransformPoint(t3DTransfMatrix m, t3DVector* v)
{
  double x = v->x;
  double y = v->y;
  double z = v->z;

  v->x = m[0][0]*x + m[0][1]*y + m[0][2]*z + m[0][3];
  v->y = m[1][0]*x + m[1][1]*y + m[1][2]*z + m[1][3];
  v->z = m[2][0]*x + m[2][1]*y + m[2][2]*z + m[2][3];
}
```

Programa 3.2: Transformação geométrica de um ponto no espaço.

3.3 Modelos Gráficos

Um modelo gráfico é uma coleção de *primitivos gráficos* e de *operadores de modelagem*. Um primitivo gráfico pode ser, por exemplo, um ponto, uma linha ou um polígono no plano ou no espaço (consideraremos somente primitivos no espaço). Os operadores de modelagem são os procedimentos responsáveis pela criação, inicialização e destruição dos primitivos de um modelo gráfico.

Notação

Utilizaremos uma notação gráfica para descrevermos modelos e seus componentes. Chamaremos essa notação gráfica de *diagrama de objetos* [100]. O *tipo* de um modelo ou componente será representado, em um diagrama de objetos, por um retângulo rotulado com o nome do tipo, em negrito. Os *atributos* do tipo, se especificados, serão descritos dentro do retângulo, abaixo do rótulo e separados deste por uma linha horizontal. Opcionalmente, poderemos descrever também o tipo do atributo (utilizaremos, nesse caso, os nomes dos tipos primitivos da linguagem C). Uma *coleção* de componentes de um modelo será representada por uma linha com um pequeno losango, tal como ilustrado na Figura 3.5. A linha sai do retângulo que representa o tipo do modelo e chega no retângulo que representa o tipo do componente.

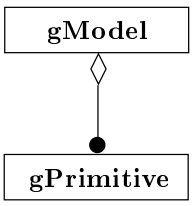


Figura 3.5: Diagrama de objetos: modelo gráfico.

O diagrama da Figura 3.5 nos diz que "**`gModel`** é definido como uma coleção de zero ou mais **`gPrimitive`**." (Utilizaremos o prefixo "`g`" para denotar que o tipo é referente a um modelo gráfico.) Note que a notação não define a *implementação* da coleção, mas somente que entre **`gModel`** e **`gPrimitive`** há uma *associação* do tipo coleção (outros tipos de associações serão definidos posteriormente.)

O Programa 3.3 mostra a implementação C do tipo **`gPrimitive`** (escolhemos implementar a coleção de um modelo gráfico como uma lista ligada duplamente encadeada de primitivos gráficos). A estrutura `gPrimitive` contém o tipo `Type` do primitivo, um ponteiro `Parent` que identifica o modelo "pai" do qual o primitivo faz parte e dois ponteiros para os elementos posterior e anterior da lista. (No diagrama da Figura 3.5 não especificamos nenhum atributo chamado `Type` para **`gPrimitive`** porque o tipo do primitivo *não é* um atributo de **`gPrimitive`**; sua definição no Programa 3.3 é detalhe de programação, e não de análise do problema.) Embora a estrutura de dados utilize dois ponteiros para cada primitivo, a implementação é justificada porque, em um ambiente interativo, pode haver um grande número de adições e remoções de elementos na coleção. Note que `gPrimitive` define a estrutura de dados comum a qualquer primitivo gráfico, ou seja, representa um primitivo *genérico* de um modelo gráfico.

```
enum gPrimitiveType
{
   gPOINT,
   gLINE,
   gPOLYLINE,
   gPOLYGON,
   gMODEL
}; // gPrimitiveType

struct gPrimitive
{
   gPrimitiveType Type;
   gModel*        Parent;
   gPrimitive*    Next;
   gPrimitive*    Previous;
}; // gPrimitive
```

Programa 3.3: Definição de primitivo gráfico.

A partir do Programa 3.3 poderíamos implementar o tipo **gModel** como sendo a estrutura

```
struct gModel
{
   gPrimitive* Primitives;
}; // gModel
```

onde `Primitives` é um ponteiro para o nó inicial da lista de primitivos do modelo. Porém, antes de implementarmos o tipo **gModel**, complementaremos o diagrama de objetos da Figura 3.5 com os tipos de primitivos de um modelo gráfico.

3.3.1 Primitivos Gráficos

Em sistemas de CAD como o AutoCAD, por exemplo, há uma grande variedade de primitivos gráficos, tais como pontos, linhas, textos, arcos de circunferência, *splines* e malhas de polígonos. Inicialmente, consideraremos os seguintes primitivos para nossos modelos gráficos: *ponto*, *linha*, *polilinha* e *polígono*, mostrados na Figura 3.6.

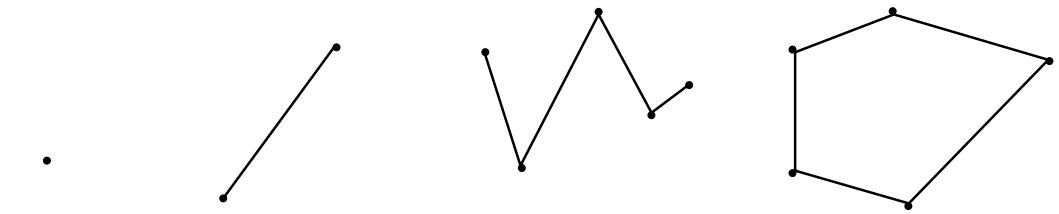


Figura 3.6: Tipos de primitivos de um modelo gráfico.

Um primitivo é geometricamente especificado por um ou mais pontos denominados *vértices*. Uma linha, por exemplo, contém dois vértices, uma polilinha contém dois ou mais vértices e um ponto contém somente um vértice. Um vértice mantém sua posição

espacial e atributos próprios que dependem da aplicação de modelagem. Esses atributos podem ser grandezas escalares, vetores, tensores ou parâmetros que controlam a aparência do primitivo ao qual o nó pertence (a cor do vértice, por exemplo). A posição espacial de um vértice é definida pelas coordenadas (x, y, z) de um vetor no espaço tridimensional, tomadas em relação a um sistema de coordenadas Cartesianas denominado *sistema global de coordenadas*, ou *sistema mundial de coordenadas*, designado pelo acrônimo WC.[3] Na Figura 3.7 é mostrado o diagrama de objetos para o tipo **gVertex**. (Os atributos dependentes da aplicação não foram especificados no diagrama.)

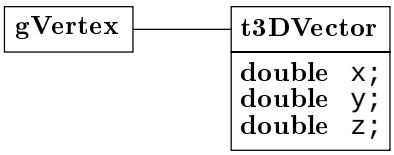


Figura 3.7: Diagrama de objetos: vértice de um primitivo gráfico.

Note, na Figura 3.7, que as coordenadas (x, y, z) de um vértice não foram definidas como um atributo do tipo **gVertex**, mas sim pelo tipo **t3DVector** implementado no Programa 3.1. A linha unindo os retângulos representa uma associação um para um; nesse exemplo, uma associação do tipo *um tem um*, ou seja, *um* **gVertex** *tem um* **t3DVector**. A implementação C do tipo **gVertex** é apresentada no Programa 3.4.

```
struct gVertex
{
  t3DVector Position;
  // atributos dependentes da aplicação
  ...
}; // gVertex
```

Programa 3.4: Definição de vértice de um modelo gráfico.

Os primitivos gráficos são *especializações* do tipo genérico **gPrimitive**. Uma especialização é representada em um diagrama de objetos por uma linha com um pequeno triângulo, como ilustrado na Figura 3.8. O diagrama da figura nos diz que os tipos "**gPoint**, **gLine**, **gPolyline**, **gPolygon** e **gModel** são especializações do tipo **gPrimitive**", ou seja, contém, além de dados específicos (descritos a seguir), também a estrutura de dados de **gPrimitive**.

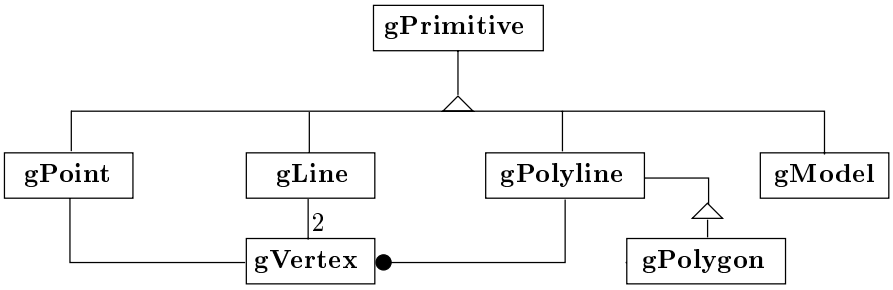


Figura 3.8: Diagrama de objetos: primitivos gráficos.

[3] *World coordinates*, em inglês.

Observe, no diagrama da Figura 3.8, que consideramos, além de ponto, linha, polilinha e polígono, também modelo gráfico como especialização de primitivo. Essa definição recursiva de modelo — um modelo é uma coleção de primitivos e um primitivo pode ser um modelo — possibilita a construção de *hierarquias* em nossos modelos gráficos, tal como discutido no Capítulo 2. A implementação de modelo gráfico é apresentada no Programa 3.5. O tipo **`gModel`** é definido como uma estrutura que contém, inicialmente, os mesmos dados de um primitivo gráfico genérico: `Type`, `Parent`, `Next` e `Previous`. (Afinal, um modelo gráfico *é* um primitivo.) Adicionalmente, a estrutura contém um ponteiro `Primitives` que representa o nó inicial da lista de primitivos "filhos" do modelo.

```
struct gModel
{
   gPrimitiveType Type;
   gModel*        Parent;
   gPrimitive*    Next;
   gPrimitive*    Previous;
   gPrimitive*    Primitives;
}; // gModel
```

Programa 3.5: Definição de modelo gráfico.

Podemos definir, agora, as funções de adição de um primitivo à lista de primitivos de um modelo e de remoção de um primitivo da lista de primitivos de um modelo. Essas funções são implementados em C no Programa 3.6. (Estamos admitindo, na função `gRemovePrimitive(model, prim)`, que o primitivo `prim` *pertence* à lista de primitivos de `model`, tendo sido anteriormente adicionado através da função `gAddPrimitive(model, prim)`.) Utilizaremos essas funções na especificação dos operadores de modelagem de um modelo gráfico.

Ponto

Um ponto é um primitivo definido por um único vértice. A implementação C do primitivo ponto é apresentada no Programa 3.7 e deriva diretamente do diagrama de objetos da Figura 3.8. Note que `gPoint` contém a estrutura de dados comum de todo primitivo gráfico, definida no Programa 3.3, mais a estrutura de dados *específica* de um ponto.

Linha

Uma linha é um primitivo definido por dois vértices. Observe, no diagrama de objetos para o tipo **`gLine`**, mostrado na Figura 3.8, o rótulo "2" na extremidade da linha próxima ao retângulo do tipo **`gVertex`**. Usaremos essa notação em uma associação de um diagrama de objetos para denotar a *multiplicidade* um para n da associação; nesse exemplo, uma associação do tipo *um tem n* (no caso, *um tem 2*). Ou seja, *um* **`gLine`** *tem 2* **`gVertex`**.

A implementação do tipo **`gLine`** é mostrada no Programa 3.8. Assim como na definição do primitivo ponto, `gLine` contém a estrutura de dados comum a todo primitivo gráfico, mais a estrutura de dados específica de uma linha.

```
void gAddPrimitive(gModel* model, gPrimitive* prim)
{
  prim->Next = model->Primitives;
  prim->Previous = 0;
  if (model->Primitives)
    model->Primitives->Previous = prim;
  model->Primitives = prim;
  prim->Parent = model;
}

void gRemovePrimitive(gModel* model, gPrimitive* prim)
{
  if (prim->Next)
    prim->Next->Previous = prim->Previous;
  if (prim->Previous)
    prim->Previous->Next = prim->Next;
  if (model->Primitives == prim)
    model->Primitives = 0;
  prim->Parent = 0;
}
```

Programa 3.6: Adição e remoção de primitivos de um modelo gráfico.

```
struct gPoint
{
  gPrimitiveType Type;
  gModel*        Parent;
  gPrimitive*    Next;
  gPrimitive*    Previous;
  gVertex        Vertex;
}; // gPoint
```

Programa 3.7: Definição de ponto.

```
struct gLine
{
  gPrimitiveType Type;
  gModel*        Parent;
  gPrimitive*    Next;
  gPrimitive*    Previous;
  gVertex        V1;
  gVertex        V2;
}; // gLine
```

Programa 3.8: Definição de linha.

Polilinha

Uma polilinha é um primitivo definido por uma coleção de dois ou mais vértices, conforme especificado no diagrama de objetos do tipo **gPolyline**, Figura 3.8. Os vértices de uma polilinha não pertencem necessariamente ao mesmo plano. Dois vértices consecutivos definem um segmento de linha da polilinha.

A implementação do tipo **gPolyline** é apresentada no Programa 3.9. A estrutura `gPolyline` contém, em adição aos dados comuns de um primitivo gráfico, um ponteiro `Vertices` para um vetor de `NumberOfVertices` vértices. Estamos admitindo, ao escolhermos um vetor para armazenar os vértices da polilinha, que o número de vértices, embora "ilimitado", seja conhecido no momento da construção da polilinha e constante até que a polilinha seja destruída. Embora essas restrições possam parecer limitantes, a implementação da polilinha fica bastante simples e compacta e, mais importante, atende aos propósitos do modelo. Em contrapartida, podemos acessar qualquer vértice em tempo constante, independente de sua posição, ao contrário de uma implementação em lista ligada.

```
struct gPolyline
{
   gPrimitiveType Type;
   gModel*        Parent;
   gPrimitive*    Next;
   gPrimitive*    Previous;
   int            NumberOfVertices;
   gVertex*       Vertices;
}; // gPolyline
```

Programa 3.9: Definição de polilinha.

Polígono

Um polígono é um primitivo definido por uma coleção de vértices, tal como a polilinha. De fato, como podemos observar no diagrama de objetos da Figura 3.8, o tipo **gPolygon** é uma especialização do tipo **gPolyline** (e, conseqüentemente, do tipo **gPrimitive**). Portanto, um polígono é uma polilinha (fechada e com no mínimo três vértices, obviamente). No entanto, diferentemente da polilinha, tomaremos cuidado para que os vértices de um polígono sejam sempre coplanares. O tipo **gPolygon** é implementado do Programa 3.10.

3.3.2 Operadores de Modelagem

Os operadores de modelagem são procedimentos, implementados como funções C, utilizados para a criação e destruição de um modelo gráfico. A partir dos operadores de modelagem, brevemente discutidos a seguir, podemos escrever outras funções de "mais alto nível" para criar modelos gráficos mais complexos.

`gModel* gNewModel(gModel* parent);`
Cria um novo modelo gráfico e adiciona o modelo na lista de primitivos de `parent`, se `parent` for diferente de zero. A adição é efetuada pela função `gAddPrimitive()`.

```
struct gPolygon
{
  gPrimitiveType Type;
  gModel*        Parent;
  gPrimitive*    Next;
  gPrimitive*    Previous;
  int            NumberOfVertices;
  gVertex*       Vertices;
}; // gPolygon
```

Programa 3.10: Definição de polígono.

```
gModel* gNewInstance(gModel* parent,
  gModel* model, t3DTransfMatrix m);
```
Cria um novo modelo gráfico igual a uma cópia do modelo `model`, mas cujos vértices são transformados pela matrix de transformação `m`, conforme a função do Programa 3.2. O novo modelo é adicionado na lista de primitivos de `parent`.

```
gPoint* gNewPoint(gModel* parent, gVertex* v);
```
Cria um novo ponto definido pelo vértice `v` e adiciona o ponto na lista de primitivos de `parent`.

```
gLine* gNewLine(gModel* parent, gVertex* v1, gVertex* v2);
```
Cria uma nova linha definida pelos vértices `v1` e `v2` e adiciona a linha na lista de primitivos de `parent`.

```
gPolyline* gNewPolyline(gModel* parent, int n, gVertex* v);
```
Cria uma nova polilinha definida pelo vetor de `n` vértices `v` e adiciona a polilinha na lista de primitivos de `parent`.

```
gPolygon* gNewPolygon(gModel* parent, int n, gVertex* v);
```
Cria um novo polígono definido pelo vetor de `n` vértices `v` e adiciona o polígono na lista de primitivos de `parent`.

```
void gDeleteModel(gModel* model);
```
Remove o modelo `model` da lista de primitivos de seu modelo pai, se o pai for diferente de zero, e destrói `model` e todos os seus primitivos (veja os operadores a seguir). A remoção é efetuada pela função `gRemovePrimitive()` do Programa 3.6.

```
void gDeletePoint(gPoint* point);
```
Remove o ponto `point` da lista de primitivos de seu modelo pai e destrói o ponto. Nesse caso, "destruir" significa liberar a área de memória utilizada pelo primitivo.

```
void gDeleteLine(gLine* line);
```
Remove a linha `line` da lista de primitivos de seu modelo pai e destrói a linha.

```
void gDeletePolyline(gPolyline* pline);
```
Remove a polilinha `pline` da lista de primitivos de seu modelo pai e destrói a linha.

Nesse caso, "destruir" significa liberar a área de memória utilizada pelo primitivo mais a área de memória utilizada para armazenamento dos vértices da polilinha.

`void gDeletePolygon(gPolygon* poly);`
Remove o polígono `poly` da lista de primitivos de seu modelo pai e destrói o polígono, da mesma forma que a polilinha.

Consideremos a implementação do operador de modelagem `gDeleteModel()`, Programa 3.11. A função "atravessa" a lista dos primitivos do modelo e, baseado no tipo de cada primitivo, chama a função de destruição correspondente. A decisão de qual função chamar é tomada em uma sentença **`switch`**.

```
void gDeleteModel(gModel* model)
{
  while (Primitives)
    switch (Primitives->Type)
    {
      case gMODEL:
        gDeleteModel(Primitives);
        break;
      case gPOINT:
        gDeletePoint(Primitives);
        break;
      ...
    }
}
```

Programa 3.11: Destruição de modelo gráfico.

Vamos supor, agora, que inventássemos um novo primitivo gráfico, por exemplo, um arco de circunferência `gArc`. Primeiramente, teríamos que definir a estrutura `gArc` com os dados comuns a todo primitivo gráfico, tal como fizemos para ponto, linha, etc., seguidos, é claro, dos dados específicos de uma arco (centro, raio, etc.). Segundo, teríamos que escrever operadores de modelagem próprios para `gArc`. Não há outra alternativa, pois `gArc` é um tipo novo de primitivo. Terceiro, teríamos que acrescentar um novo **`case`** à função `DeleteModel()` — e a quaisquer outras funções que "atravessam" a lista de primitivos de um modelo e realizam alguma operação sobre cada primitivo. Isso significa que teríamos de alterar um programa supostamente correto para cada novo tipo de primitivo. Como podemos evitar isso?

Consideremos o Programa 3.12. Note que redefinimos `gPrimitiveType`. No Programa 3.3, `gPrimitiveType` era um inteiro; agora, é uma estrutura que contém um ponteiro de caracteres `Name` para o nome do tipo de primitivo e um ponteiro `Delete` para uma função que toma como parâmetro um ponteiro para um primitivo. Redefinimos, também, `gPrimitive`, em função de `gPrimitiveType`. O Programa 3.13 mostra a estrutura `gPrimitiveType` para o caso da linha e a implementação da função `gNewLine()`.

Se tivermos uma estrutura desse tipo para cada primitivo do modelo, podemos escrever novamente a função `DeleteModel()` como mostrado no Programa 3.14. Essa função *não necessita de alterações* para um novo primitivo. Resolvemos o problema,

```
struct gPrimitiveType
{
  char* Name;
  void  (*Delete)(gPrimitive*);
}; // gPrimitiveType

struct gPrimitive
{
  gPrimitiveType* Type;
  gModel*         Parent;
  gPrimitive*     Next;
  gPrimitive*     Previous;
}; // gPrimitive
```

Programa 3.12: Outra definição de primitivo gráfico.

```
gPrimitiveType gLineClass = { "gLine", gDeleteLine };

gLine* gNewLine(gModel* parent, gVertex* v1, gVertex* v2)
{
  gLine* line = malloc(sizeof(gLine));

  line->Type = &gLineClass;
  line->V1 = *v1;
  line->V2 = *v2;
  gAddPrimitive(parent, (gPrimitive*)line);
  return line;
}
```

Programa 3.13: Instância do primitivo linha.

mas o processo é um pouco confuso e trabalhoso. No Capítulo 8 faremos muito melhor que isso, mas já estamos programando orientado a objetos.

```
void gDeleteModel(gModel* model)
{
  while (Primitives)
    (*Primitives->Type->Delete)(Primitives);
}
```

Programa 3.14: Outra versão para destruição de modelo gráfico.

3.4 Modelos de Cascas

Consideremos o modelo gráfico M da Figura 3.9, constituído por dois polígonos P1 e P2, os quais, *no desenho*, compartilham a aresta E. Dizemos *no desenho* porque, em um modelo gráfico, cada primitivo é definido em termos de seu próprio conjunto de vértices e, portanto, não podemos afirmar, *somente* a partir dos dados armazenados no modelo, que a aresta E seja comum a P1 e P2. De fato, se necessitarmos saber

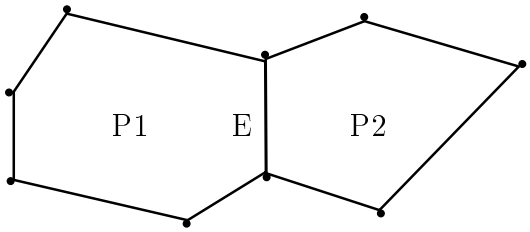


Figura 3.9: Dois polígonos de um modelo gráfico M.

se P1 e P2 compartilham alguma aresta, devemos *comparar* cada par de vértices de P1 com cada par de vértices de P2. Naturalmente, dependendo da aplicação, esse esquema pode se tornar bastante ineficiente em termos computacionais. Além disso, se alterarmos a posição de um dos vértices da aresta "compartilhada", obteríamos o resultado mostrado na Figura 3.10(a), ao invés daquele mostrado na Figura 3.10(b), o que, também dependendo da aplicação, poderia não ser o resultado desejado. Por exemplo, se utilizássemos o modelo M para representar a geometria de uma estrutura definida por duas cascas (os polígonos P1 e P2), certamente gostaríamos de obter o resultado da Figura 3.10(b).

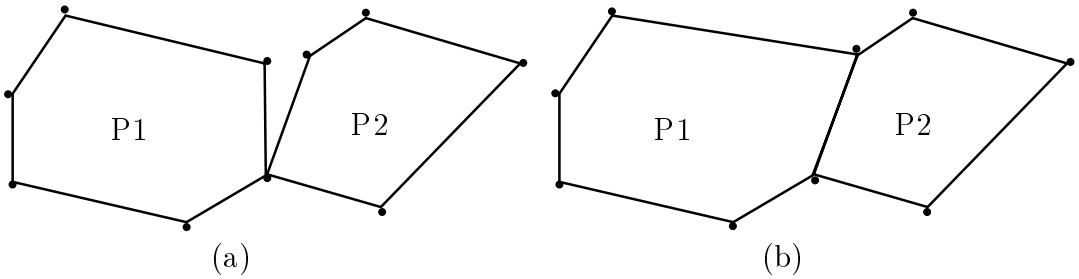


Figura 3.10: Alteração da posição de um dos vértices de M.

Vamos, agora, sofisticar um pouco nossos modelos gráficos, acrescentando à estrutura de dados informações explícitas sobre determinadas *relações de adjacência* entre os elementos que constituem o modelo. Chamaremos esse novo modelo geométrico *não-manifold* de *modelo de cascas* e o utilizaremos para representar estruturas constituídas de cascas delgadas. Definiremos um modelo de cascas como sendo uma coleção de *faces*, *arestas* e *vértices*, como mostrado no diagrama de objetos da Figura 3.11 (usaremos prefixo `b` para modelos de cascas).

O Programa 3.15 mostra a implementação C dos tipos do diagrama de objetos da Figura 3.11. Mais uma vez, adotamos listas ligadas duplamente encadeadas para as coleções de componentes do modelo. Vamos comentar brevemente cada um dos componentes do modelo.

Face. Uma face de um modelo de cascas 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. Os demais laços, se existirem, representam contornos de "buracos" da face. Além dos ponteiros para os elementos posterior e anterior da lista de faces, a estrutura `bFace` contém um ponteiro `Parent` para o modelo ao qual a face pertence, um ponteiro `Loops` para a lista de laços da face e um ponteiro `OuterLoop` que identifica o laço do contorno externo.

```
struct bModel
{
  bFace*   Faces;
  bEdge*   Edges;
  bVertex* Vertices;
}; // bModel

struct bFace
{
  bModel* Parent;
  bLoop*  OuterLoop;
  bLoop*  Loops;
  bFace*  Next;
  bFace*  Previous;
}; // bFace

struct bLoop
{
  bFace*    Face;
  bEdgeUse* FirstEdgeUse;
  bLoop*    Next;
  bLoop*    Previous;
}; // bLoop

struct bEdgeUse
{
  bLoop*    Loop;
  bEdge*    Edge;
  bVertex*  Vertex;
  bEdgeUse* Next;
  bEdgeUse* Previous;
}; // bEdgeUse

struct bVertex
{
  t3DVector   Position;
  bVertexUse* VertexUses;
  bEdgeUse*   Next;
  bEdgeUse*   Previous;
}; // bVertex

struct bVertexUse
{
  bEdgeUse*   EdgeUse;
  bVertexUse* Next;
  bVertexUse* Previous;
}; // bVertexUse
```

Programa 3.15: Definição de modelo de cascas.

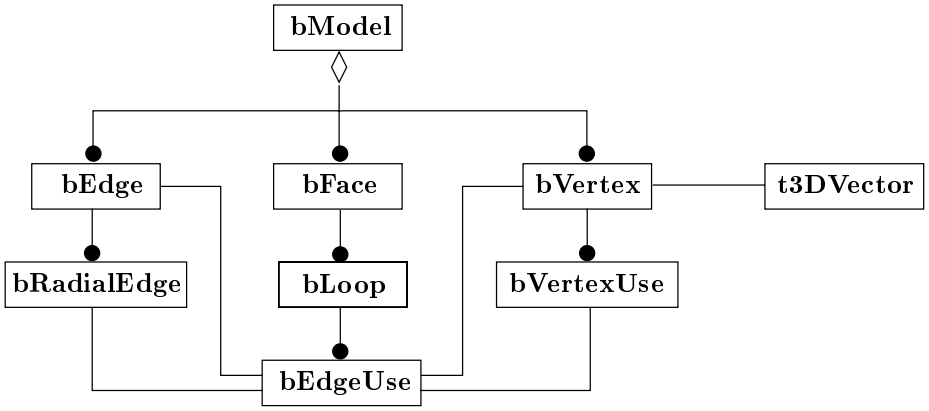


Figura 3.11: Diagrama de objetos: modelo de cascas.

```
struct bEdge
{
  bVertex*     V1;
  bVertex*     V2;
  bRadialEdge* RadialEdges;
  bEdge*       Next;
  bEdge*       Previous;
}; // bEdge

struct bRadialEdge
{
  bEdgeUse*    EdgeUse;
  bRadialEdge* Next;
  bRadialEdge* Previous;
}; // bRadialEdge
```

Programa 3.16: Definição de modelo de cascas. (cont.)

Laço. Um laço é um contorno conectado definido por um ciclo de *usos de aresta* (explicado a seguir). `bLoop` contém um ponteiro `Face` para a face a qual o laço pertence, um ponteiro `FirstEdgeUse` para a lista ligada circular duplamente encadeada de usos de aresta do laço e os ponteiros para os elementos posterior e anterior da lista de laços da face. Cada par (uso de aresta, uso de aresta próximo) define uma aresta da face `Face`.

Uso de aresta. Em uma representação *não-manifold* podemos ter uma, duas ou mais faces incidindo em uma única aresta. Representaremos a aresta somente uma vez, mas manteremos explicitamente, para cada face incidente na aresta, a informação que a face "usa" a aresta. Essa informação é implementada por `bEdgeUse`, o qual contém um ponteiro `Loop` para o laço da face que "usa" a aresta, um ponteiro `Vertex` para o vértice do qual parte o "uso" e um ponteiro `Edge` para a aresta na qual a face que contém o "uso" incide.

Vértice. `bVertex` contém um vetor `Position` que define a localização do vértice, um ponteiro `VertexUses` para a lista de usos de vértice (explicado a seguir) e os ponteiros para os elementos posterior e anterior da lista de vértices.

Uso de vértice. Assim como no caso de várias faces incidindo em uma única aresta, podemos ter também várias arestas incidindo em um único vértice. Faremos como anteriormente: representaremos o vértice somente uma vez, mas recordaremos explicitamente todos os "usos" do vértice em um `bVertexUse`. A estrutura contém um ponteiro `EdgeUse` para o uso de aresta que "usa" o vértice e os ponteiros para os elementos posterior e anterior da lista de "usos" de vértice do vértice.

Aresta. Uma aresta é um segmento do contorno de uma ou mais faces definido por dois vértices. A estrutura `bEdge` mantém dois ponteiros `V1` e `V2` que identificam seus vértices, um ponteiro `RadialEdges` para a lista de *arestas radiais* (explicado a seguir) e os ponteiros para os elementos posterior e anterior da lista de arestas do modelo.

Aresta radial. `bRadialEdge` contém um ponteiro `EdgeUse` para um uso de aresta e os ponteiros para os elementos posterior e anterior da lista de arestas radiais de uma aresta. O objetivo de uma aresta radial é definir explicitamente que determinada aresta é "usada" por um laço de uma face.

Observe cuidadosamente a estrutura de dados de um modelo de cascas (admitimos que pode ser um pouco complicado) e verifique que uma face "conhece" suas arestas e seus vértices, uma aresta "conhece" suas faces e seus vértices e um vértice "conhece" suas faces e suas arestas. Representamos todas essas informações de adjacência com um custo adicional de memória e de complexidade dos operadores que manipulam a estrutura do Programa 3.15. Inventamos essa representação porque sua utilização simplifica o processo automático de geração de malhas, conforme resumiremos no Capítulo 6.

Descreveremos os operadores de modelagem de um modelo de cascas na segunda parte do texto, Capítulo 10 (veja a classe C++ `tShell`). Será muito mais fácil do que escrevermos uma versão em C somente para comparar as implementações. A Figura 3.12 mostra um exemplo de casca cilíndrica gerada por varredura translacional (veja a próxima seção) a partir dos operadores de modelagem.

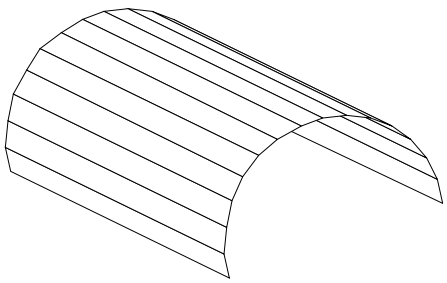


Figura 3.12: Casca cilíndrica.

3.5 Modelos de Sólidos

Definiremos um modelo de sólidos como sendo uma coleção de faces, arestas e vértices, da mesma forma que fizemos para modelos de cascas. Porém, diferentemente de uma casca, as faces de um sólido devem ser conectadas de modo a formar a superfície

2-manifold de contorno do volume do sólido. (No Capítulo 2 discutimos os modelos de contorno.) O diagrama de objetos de um modelo de sólidos é mostrado na Figura 3.13.

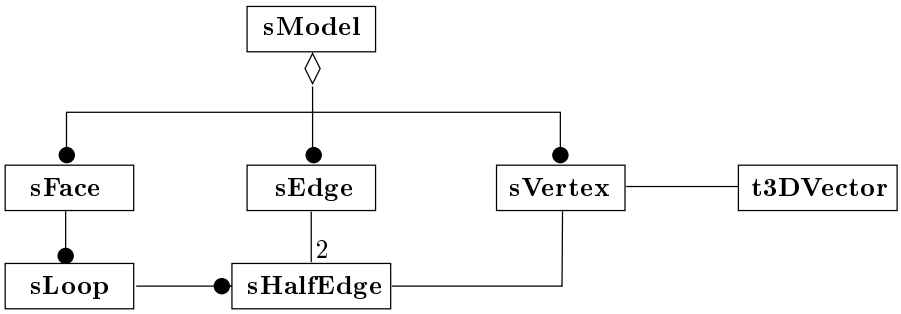


Figura 3.13: Diagrama de objetos: modelo de sólidos

Adotamos a estrutura de dados do Programa 3.17 para implementarmos os componentes de um modelo de sólidos. A estrutura, inventada por MÄNTYLÄ [74] e chamada *estrutura de dados semi-aresta* é, sob alguns aspectos, similar à estrutura de dados de nossos modelos de cascas.[4] No entanto, os operadores de modelagem de sólidos, os operadores de Euler, são definidos de tal forma que sua aplicação sempre mantém a integridade topológica do modelo, definida pela fórmula de Euler-Poincaré, Equação (2.1). Não discutiremos os operadores de Euler aqui. A implementação, baseada na teoria de *modelos planares*, é detalhadamente apresentada por MÄNTYLÄ.

Comparemos as estruturas de dados de um modelo de cascas e um modelo de sólidos. `sFace` é similar a `bFace` e representa uma face plana da superfície de um sólido, definida por um contorno externo, podendo conter ou não cavidades em seu interior. Um `sLoop` também é similar a `bLoop`. As diferenças começam na definição da estrutura `sHalfEdge`, a qual representa o uso de uma aresta por uma face do sólido. Note, contudo, que uma aresta *manifold* possui somente (e sempre) dois usos, ou seja, só podem incidir duas faces em uma aresta de um sólido ("meia" aresta em cada face). Um `sHalfEdge` mantém um ponteiro `Edge` para a aresta da qual a "semi-aresta" faz parte, um ponteiro `Vertex` para o vértice do qual parte a "semi-aresta", um ponteiro `Loop` para o laço que contém a "semi-aresta" e os ponteiros para os elementos posterior e anterior da lista ligada circular duplamente encadeada de "semi-arestas" do laço. A estrutura `sEdge` mantém diretamente dois ponteiros para cada uma das duas "semi-arestas" da aresta, não sendo necessária a definição de uma lista de "arestas radiais", como fizemos no caso da estrutura `bEdge`. Da mesma forma, todas as relações de adjacência ao redor de um vértice `sVertex` podem ser determinadas somente através de um único ponteiro `HalfEdge` para uma das "semi-arestas" que incidem no vértice (MÄNTYLÄ descreve e implementa todos os detalhes).

A Figura 3.14 mostra três exemplos de primitivos sólidos. Os objetos foram gerados por processos de *varredura* responsáveis pela aplicação adequada dos (complicados) operadores de Euler de baixo nível. Os tipos de varredura, ou *sweep*, implementados em OSW são descritos a seguir.

[4] A estrutura de um modelo de sólidos é mais simples porque a representação é *manifold*: em uma aresta incidem sempre duas faces, etc.

```
struct sModel
{
  sFace*   Faces;
  sEdge*   Edges;
  sVertex* Vertices;
}; // sModel

struct sFace
{
  sModel* Solid;
  sLoop*  OuterLoop;
  sLoop*  Loops;
  sFace*  Next;
  sFace*  Previous;
}; // sFace

struct sLoop
{
  sHalfEdge* HalfEdge;
  sFace*     Face;
  sLoop*     Next;
  sLoop*     Previous;
}; // sLoop

struct sHalfEdge
{
  sEdge*     Edge;
  sVertex*   Vertex;
  sLoop*     Loop;
  sHalfEdge* Next;
  sHalfEdge* Previous;
}; // sHalfEdge
```

Programa 3.17: Estrutura de dados semi-aresta.

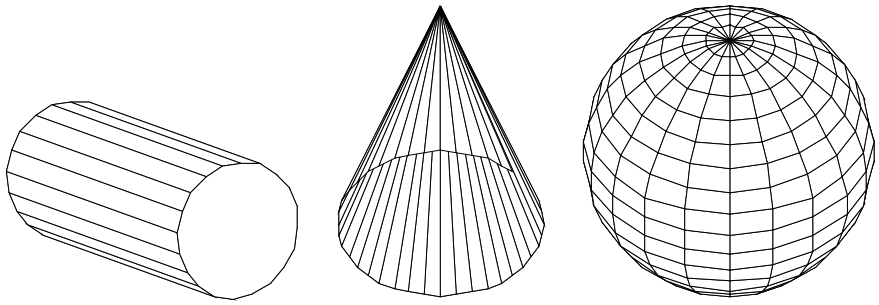
Varredura translacional. Os pontos de uma polilinha (fechada ou aberta) "escorregam" em um eixo de determinada altura. O conjunto de todos os pontos gerados durante o "escorregamento" definem o objeto, como mostra o cilindro da Figura 3.14(a).

Varredura translacional cônica. Os pontos de uma polilinha aberta ou fechada "escorregam" convergindo para determinado ponto, como mostrado no cone da Figura 3.14(b).

Varredura rotacional. Os pontos de uma polilinha aberta ou fechada "escorregam" ao redor de determinado eixo. A esfera da Figura 3.14(c) foi obtida a partir da rotação de 360^o de meia "circunferência" (na verdade, meio polígono regular com 20 pontos) em torno de seu diâmetro.

```
struct sEdge
{
  sHalfEdge* He1;
  sHalfEdge* He2;
  sEdge*     Next;
  sEdge*     Previous;
}; // sEdge

struct sVertex
{
  t3DVector  Position;
  sHalfEdge* HalfEdge;
  sVertex*   Next;
  sVertex*   Previous;
}; // sVertex
```

Programa 3.18: Estrutura de dados semi-aresta. (cont.)



(a) Cilindro (b) Cone (c) Esfera

Figura 3.14: Primitivos sólidos.

3.6 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 ao sistema global de coordenadas. A coleção de vértices do modelo define a *geometria* do modelo. Um vértice possui um *identificador global*, usualmente um número inteiro que define a posição ou a ordem do vértice na coleção. Uma célula é caracterizada por uma coleção ordenada de identificadores dos vértices nos quais a célula incide, chamada de *lista de conectividade* da célula. A coleção de células do modelo define a *topologia* do modelo. O diagrama de objetos de um modelo de decomposição por células é mostrado na Figura 3.15.

O Programa 3.19 ilustra a implementação de um modelo de decomposição por células. (Mais uma vez utilizamos listas ligadas duplamente encadeadas para células e vértices do modelo). Da mesma forma como fizemos para modelos gráficos, não definiremos o tipo de uma célula somente por um número inteiro; ao invés disso, utilizaremos um ponteiro para uma estrutura `cCellType`, a qual mantém, nesse exemplo, somente o nome do tipo da célula e um ponteiro para uma função que "destrói" o componente.

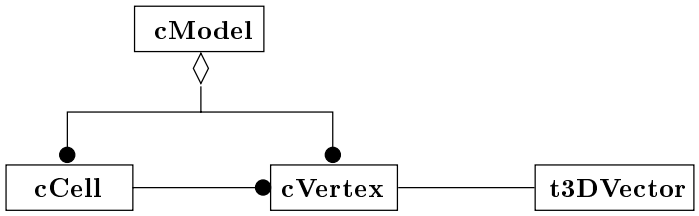


Figura 3.15: Diagrama de objetos: modelo de decomposição por células.

A lista de conectividade de uma célula `cCell` genérica é implementada por um vetor `Vertices` de `NumberOfVertices` ponteiros para os vértices nos quais a célula incide. A estrutura `cVertexUse` informa qual é a célula `Cell` que "usa" o vértice. Essa informação será útil em alguns algoritmos de visualização.

3.6.1 Tipos de Células

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 célula de dimensão topológica 2 é uma superfície no espaço e uma célula de dimensão topológica 3 é uma região do espaço. (Fisicamente, uma célula sempre é tridimensional.)

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.) Como veremos no Capítulo 5, com um sistema de coordenadas normalizadas podemos mais apropriadamente definir alguns atributos de uma célula (por exemplo, a rigidez de um elemento finito) e executar mais facilmente algumas operações numéricas (por exemplo, a integração de 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

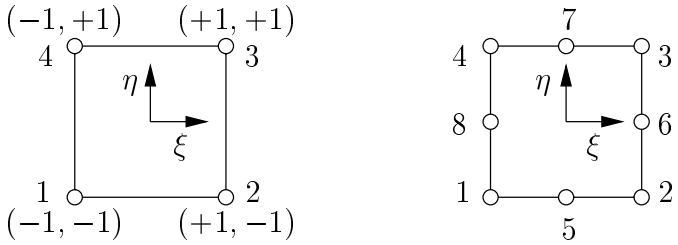
$$\mathbf{x} = \sum_{i=1}^{r} N_i(\boldsymbol{\xi})\, \mathbf{x}_i, \tag{3.17}$$

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

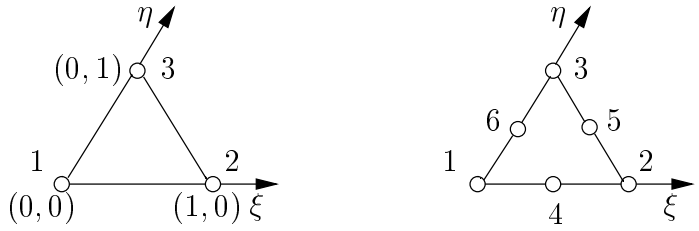
A Figura 3.16 mostra as células de dimensão topológica 2 utilizadas nos modelos do trabalho. As funções de forma, comumente encontradas na literatura [18, 132], são

```
struct cCellType
{
  char* Name;
  void  (*Delete)(cCell*);
}; // cCellType

struct cVertexUse
{
  cCell*     Cell;
  cVertexUse* Next;
  cVertexUse* Previous;
}; // cVertexUse

struct cVertex
{
  t3DVector   Position;
  cVertexUse* Uses;
  cVertex*    Next;
  cVertex*    Previous;
  // atributos dependentes da aplicação
  // veja Capítulo 6 e Capítulo 7
  ...
}; // cVertex

typedef cVertex* cVertexArray;
struct cCell
{
  cCellType*    Type;
  int           NumberOfVertices;
  cVertexArray* Vertices;
  cCell*        Next;
  cCell*        Previous;
}; // cCell

struct cModel
{
  int      NumberOfVertices;
  cVertex* Vertices;
  int      NumberOfCells;
  cCell*   Cells;
}; // cModel
```

Programa 3.19: Definição de célula e de modelo de decomposição por células.



(a) Quadriláteros



(b) Triângulos

Figura 3.16: Células de dimensão topológica 2.

dadas, para o quadrilátero com quatro vértices da Figura 3.16(a), por

$$\begin{aligned} N_1 &= \frac{1}{4}(1-\xi)(1-\eta), \\ N_2 &= \frac{1}{4}(1+\xi)(1-\eta), \\ N_3 &= \frac{1}{4}(1+\xi)(1+\eta), \\ N_4 &= \frac{1}{4}(1-\xi)(1+\eta), \end{aligned} \tag{3.18}$$

e, para o quadrilátero com oito vértices, por

$$\begin{aligned} N_1 &= \frac{1}{4}(1-\xi)(1-\eta)(-1-\xi-\eta), \\ N_2 &= \frac{1}{4}(1+\xi)(1-\eta)(-1+\xi-\eta), \\ N_3 &= \frac{1}{4}(1+\xi)(1+\eta)(-1+\xi+\eta), \\ N_4 &= \frac{1}{4}(1-\xi)(1+\eta)(-1-\xi+\eta), \\ N_5 &= \frac{1}{2}(1-\xi^2)(1-\eta), \\ N_6 &= \frac{1}{2}(1-\eta^2)(1+\xi), \\ N_7 &= \frac{1}{2}(1-\xi^2)(1+\eta), \\ N_8 &= \frac{1}{2}(1-\eta^2)(1-\xi). \end{aligned} \tag{3.19}$$

Para o triângulo com três vértices da Figura 3.16(b), as funções de forma são dadas por

$$\begin{aligned} N_1 &= 1 - \xi - \eta, \\ N_2 &= \xi, \\ N_3 &= \eta, \end{aligned} \tag{3.20}$$

e, para o triângulo com seis vértices, por

$$\begin{aligned} N_1 &= 2(1 - \xi - \eta)(\frac{1}{2} - \xi - \eta), \\ N_2 &= \xi(2\xi - 1), \\ N_3 &= \eta(2\eta - 1), \\ N_4 &= 4\xi\eta, \\ N_5 &= 4\eta(1 - \xi - \eta), \\ N_6 &= 4\xi(1 - \xi - \eta). \end{aligned} \tag{3.21}$$

3.7 Sumário

Nesse Capítulo descrevemos as estruturas de dados utilizadas na representação geométrica de objetos em OSW: modelos gráficos, modelos de cascas, modelos de sólidos e modelos de decomposição por células.

Um modelo gráfico é uma coleção hierárquica de primitivos gráficos. Um primitivo gráfico pode ser um ponto, uma linha, uma polilinha, um polígono ou outro modelo gráfico. Um primitivo é definido geometricamente por um ou mais vértices, os quais mantém a posição do primitivo no espaço e os atributos que controlam sua aparência ou são dependentes da aplicação.

Um modelo de cascas é uma coleção *não-manifold* de faces planas, arestas e vértices. Uma face é definida por um ou mais laços. Um laço é uma seqüência cíclica de usos de arestas. Um uso de aresta é definido pelo vértice do qual parte o uso. Um vértice é definido por um vetor responsável pela localização espacial do vértice e por uma lista de usos de vértices. Usos de arestas e usos de vértices contêm as principais informações relativas à adjacência dos componentes topológicos do modelo.

Um modelo de sólidos é uma coleção de vértices, arestas e faces planas que representam a superfície *manifold* de um sólido. Adotamos a estrutura de dados proposta por MÄNTYLÄ [74] para implementarmos modelos de sólidos.

Um modelo de decomposição por células é uma coleção de células "coladas" em vértices discretos comumente denominados nós. Apresentamos as funções de forma de alguns dos tipos de células bidimensionais utilizadas em OSW.

CAPÍTULO 4

Modelos Matemáticos

4.1 Introdução

No Capítulo 2 vimos que um modelo matemático é definido em termos de equações diferenciais que descrevem quantitativamente o comportamento de um determinado objeto, obtidas a partir de hipóteses simplificadoras sobre o comportamento do objeto. Neste Capítulo, desenvolveremos alguns modelos matemáticos baseados na teoria da mecânica do contínuo para o problema fundamental descrito no Capítulo 1. As equações da teoria nos permitirão predizer quais os campos de deslocamentos, deformações e tensões de um corpo sólido submetido à ação estática de forças externas.

A formulação de nossos modelos matemáticos será primeiramente fundamentada na suposição de que a matéria que compõe os corpos sólidos seja destituída de espaços vazios. Admitiremos, também, que todas as funções matemáticas utilizadas na teoria sejam contínuas, exceto, possivelmente, em número finito de pontos ou em superfícies internas que separam regiões de continuidade. Denominaremos este material hipotético de meio contínuo, ou *continuum*. A hipótese da continuidade é a primeira simplificação considerada na modelagem matemática do problema fundamental.

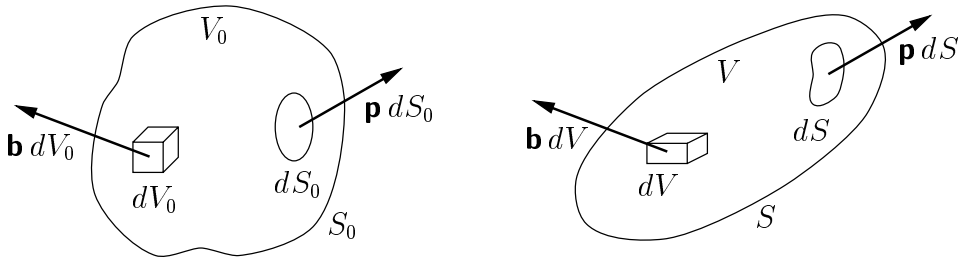
CONTINUIDADE Um material é contínuo se preenche completamente o espaço que ocupa, sem vazios, e suas propriedades puderem ser descritas por funções contínuas.

O conceito de meio contínuo nos permite definir *tensão* em um ponto, um lugar geométrico no espaço que não ocupa volume algum. Definiremos tensão na Seção 4.2, fundamentados apenas na hipótese da continuidade. Na Seção 4.3 apresentaremos um resumo da cinemática da partícula e os conceitos de *deformações*, os quais são baseados no deslocamento relativo entre as partículas que constituem um corpo. Posteriormente, restringiremos nossa discussão somente a sólidos que apresentam pequenos deslocamentos e pequenas deformações. Na Seção 4.4 derivaremos as equações de equilíbrio estático de um sólido, escritas em termos de tensões e de forças externas aplicadas no volume do corpo. Na Seção 4.5 definiremos as relações entre tensões e deformações em sólidos supostos idealmente elásticos, conhecidas como lei de Hooke generalizada.

Nosso primeiro modelo matemático para o problema fundamental será formulado na Seção 4.6, a partir das equações de equilíbrio, das equações de deformação e da lei de Hooke generalizada. Apresentaremos uma solução analítica do modelo para corpos de dimensão infinita sujeitos a um carregamento unitário concentrado, denominada solução fundamental de Kelvin. Essa solução será útil no Capítulo 5. Na Seção 4.7 definiremos as equações dos modelos simplificados de membranas e placas delgadas que utilizaremos na análise numérica de cascas.

4.2 Tensões

Consideremos um corpo contínuo qualquer que, em um determinado instante $t_0 = 0$ de tempo, ocupa um volume V_0 no espaço, delimitado por uma superfície S_0, e sobre o qual atuam forças externas como mostrado na Figura 4.1(a). Devido à atuação das forças externas, as partículas do corpo sofrem uma modificação de sua posição inicial e passam a ocupar, em um instante qualquer t após a aplicação das forças, novas posições que definem a *configuração deformada* do corpo. Na configuração deformada, conforme indicado na Figura 4.1(b), o corpo ocupa um volume V, delimitado por uma superfície S. Vamos supor que, nessa nova configuração, o corpo se encontre em equilíbrio estático. Começaremos nossa discussão sobre tensão classificando as forças que atuam sobre o corpo em duas categorias: forças de volume e forças de superfície.



(a) $t_0 = 0$ (b) t

Figura 4.1: Corpo em equilíbrio estático.

Forças de volume são forças que atuam sobre elementos de volume ou de massa do interior de um corpo, como, por exemplo, a força da gravidade. Denotaremos a força de volume por unidade de volume que atua sobre um elemento de volume dV do domínio de um corpo por $\mathbf{b}$. Em um instante qualquer de tempo, a força de volume que atua sobre um volume finito V de um corpo será

$$\int_V \mathbf{b}\, dV = \mathbf{i}_k \int_V b_k\, dV. \tag{4.1}$$

Um dos postulados fundamentais da mecânica do contínuo, baseado unicamente na hipótese da continuidade, é que o limite

$$\lim_{\Delta V \to 0} \frac{\Delta \mathbf{B}}{\Delta V} = \mathbf{b}, \tag{4.2}$$

onde $\Delta\mathbf{B}$ é a força total que atua sobre o elemento de volume ΔV, existe para cada ponto do domínio e é independente da geometria do elemento de volume considerado

no ponto, ou seja, podemos considerar, por exemplo, elementos de esfera, cubo ou qualquer outro volume cuja dimensão tenda a zero.

Forças de superfície são forças de contato que atuam sobre elementos de superfície do contorno de um corpo. Denotaremos a força de superfície por unidade de área que atua sobre um elemento infinitesimal dS do contorno de um corpo $\mathbf{p}$. Em um instante qualquer de tempo, a força de superfície que atua sobre uma porção finita S da superfície de um corpo será

$$\int_S \mathbf{p}\, dS = \mathbf{i}_k \int_S p_k\, dS. \tag{4.3}$$

Um postulado similar àquele definido pela Equação (4.2) pode ser derivado da definição de força de superfície. Consideremos uma força $\Delta\mathbf{P}$ atuando sobre o elemento ΔS da superfície S de um volume V, como mostrado na Figura 4.2. A superfície S separa

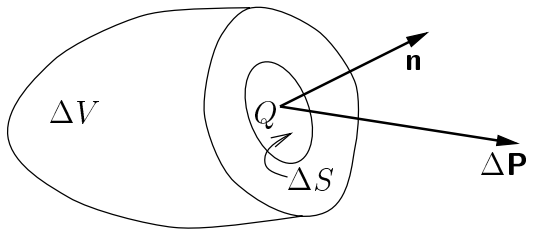


Figura 4.2: Força $\Delta\mathbf{P}$ na superfície ΔS.

duas porções de um corpo seccionado por um plano definido pelo vetor normal $\mathbf{n}$ e pelo ponto Q, sendo o volume ΔV uma dessas porções, tomada arbitrariamente. A força $\Delta\boldsymbol{P}$ representa a *ação interna* exercida entre as duas porções do corpo. O postulado é que o limite

$$\lim_{\Delta S \to 0} \frac{\Delta \mathbf{P}}{\Delta S} = \mathbf{p} \tag{4.4}$$

existe sobre a superfície S, no ponto Q, e é independente do elemento de superfície considerado. A força de superfície $\mathbf{p}$, ou *tensão*, define a intensidade da resultante das forças (supostas continuamente distribuídas na superfície) por unidade de área que atuam em um plano com normal $\mathbf{n}$, no ponto Q. A determinação do limite (4.4) é baseada na suposição de que este limite é dependente da orientação da normal $\mathbf{n}$ da superfície que passa pelo ponto Q, como ilustrado na Figura 4.3.

A expressão para o vetor de tensão em um ponto Q da superfície de um corpo, segundo uma direção qualquer definida por uma normal $\mathbf{n}$, pode ser obtida a partir dos vetores de tensão, no ponto Q, sobre planos perpendiculares aos eixos coordenados. Os componentes Cartesianos desses vetores definem os componentes Cartesianos do *tensor de tensões de Cauchy* $\boldsymbol{\sigma}$ no ponto Q, os quais podem ser organizados matricialmente como

$$[\boldsymbol{\sigma}] = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ \sigma_{yx} & \sigma_{yy} & \sigma_{yz} \\ \sigma_{zx} & \sigma_{zy} & \sigma_{zz} \end{bmatrix} \quad \text{ou} \quad [\boldsymbol{\sigma}] = \begin{bmatrix} \sigma_{11} & \sigma_{12} & \sigma_{13} \\ \sigma_{21} & \sigma_{22} & \sigma_{23} \\ \sigma_{31} & \sigma_{32} & \sigma_{33} \end{bmatrix}, \tag{4.5}$$

onde $\sigma_{ij} = p_j^{(i)}$. A notação para distinguir os nove componentes de $\boldsymbol{\sigma}$ é mostrada na Figura 4.4. O primeiro índice indica o plano de atuação e o segundo índice indica

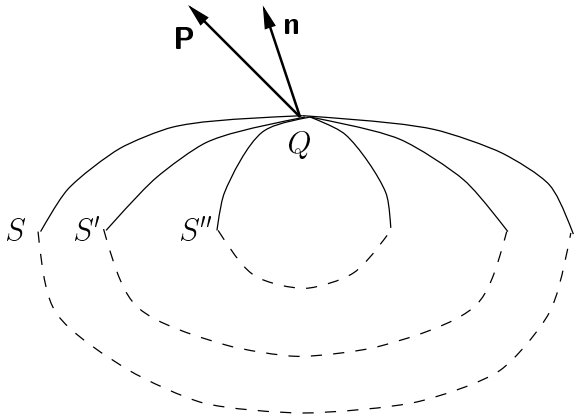


Figura 4.3: A tensão em um ponto depende da normal ao ponto.

a direção de atuação, considerada positiva no sentido positivo do eixo coordenado correspondente. Os componentes σ_{ii} representam os componentes *normais* de tensão; os componentes σ_{ij}, $i \neq j$, representam os componentes de *cisalhamento* de tensão, ou cortantes.

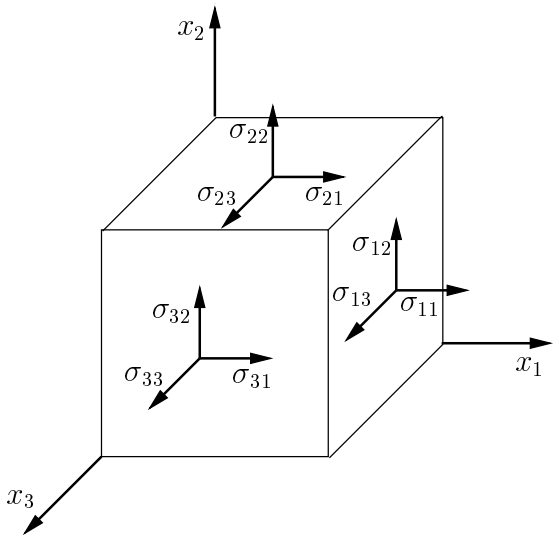


Figura 4.4: Componentes Cartesianos do tensor de tensões de Cauchy.

O tensor de tensões de Cauchy, quando não há momentos de volume [72], é simétrico, ou seja,

$$\sigma_{ij} = \sigma_{ji}. \tag{4.6}$$

A expressão (4.6) pode ser obtida a partir da consideração de equilíbrio de um paralelepípedo elementar definido em torno de um ponto Q, onde as forças de volume são desconsideradas porque representam um infinitésimo de ordem superior em relação às forças de superfície [117]. Os momentos devidos à não uniformidade de distribuição das tensões normais também são desconsiderados, por representarem infinitésimos de ordem superior em relação àqueles devidos às tensões cortantes.

Podemos obter o vetor de tensão sobre um plano arbitrário passando em um ponto Q da superfície de um corpo como segue. Se o corpo está em equilíbrio, qualquer

porção desse corpo também deve estar em equilíbrio. Consideremos, então, um tetraedro "muito pequeno" extraído do corpo em consideração, ao redor do ponto Q, como mostrado na Figura 4.5. Este tetraedro é denominado *tetraedro de Cauchy*, sendo composto por três faces perpendiculares aos eixos coordenados, cujas normais apontam no sentido negativo desses eixos, e por uma face inclinada, definida por um vetor normal $\mathbf{n}$. O equilíbrio em cada direção pode ser expresso a partir das forças que atuam no

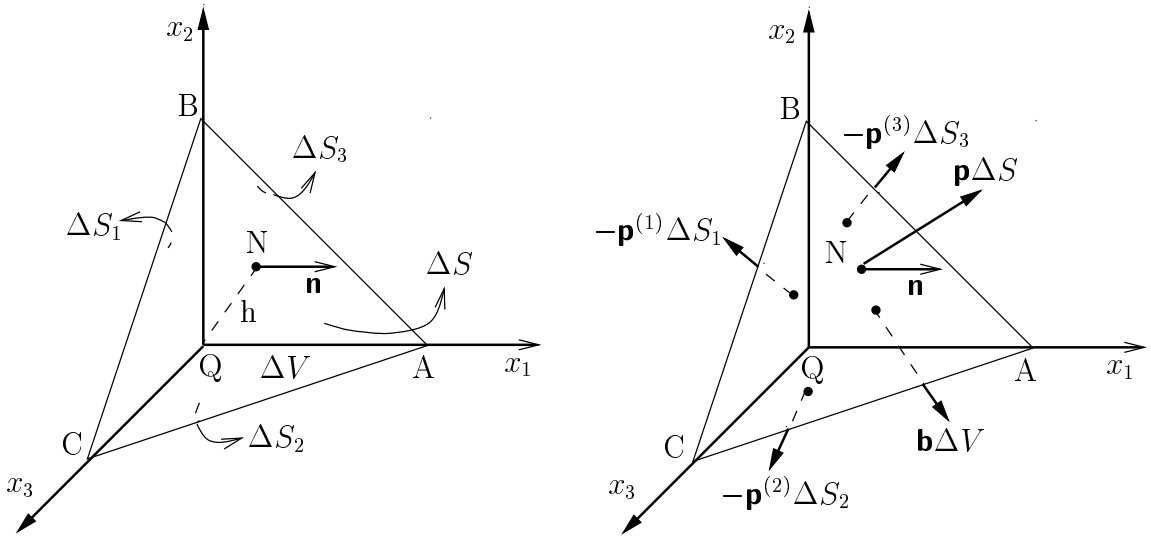


Figura 4.5: Tetraedro de Cauchy.

tetraedro,[1] Figura 4.5(b). No limite, quando o tamanho do tetraedro tende a zero, sua altura h também tende a zero, Figura 4.5(a), o efeito da força de volume anula-se e obtemos [57, 72, 117]

$$\mathbf{p} = \mathbf{p}^{(1)}\, n_1 + \mathbf{p}^{(2)}\, n_2 + \mathbf{p}^{(3)}\, n_3, \tag{4.7}$$

ou

$$\begin{aligned} &\mathbf{p} = \mathbf{n} \cdot \boldsymbol{\sigma}, \\ &p_i = \sigma_{ji}\, n_j \quad \text{em notação indicial.} \end{aligned} \tag{4.8}$$

A Equação (4.8) é chamada *transformação de tensão de Cauchy*, e relaciona os componentes do vetor de tensão em um ponto de um plano de normal $\mathbf{n}$ com os componentes do tensor de tensões no ponto.

4.2.1 Tensões Principais

Conhecido o estado de tensões em determinado ponto, é sempre possível escolhermos um conjunto especial de eixos que passam através do ponto tal que os componentes de cisalhamento do tensor de tensões, tomado em relação ao sistema de coordenadas definido por esses eixos, sejam nulos. Esses eixos são denominados *eixos principais* ou *direções principais*. O vetor de tensão sobre um plano perpendicular a um eixo principal é normal ao plano. Os três planos que passam através do ponto, perpendiculares aos

[1] Em uma discussão mais completa devemos considerar, também, o equilíbrio dos momentos das forças atuantes no tetraedro. Veja, por exemplo, MALVERN [72, página 213].

três eixos principais, são chamados *planos principais* e os componentes normais de tensão sobre os planos principais, denotados por σ_1, σ_2 e σ_3, são chamados *tensões principais*. As direções e tensões principais em um ponto Q de um corpo podem ser determinadas como segue.

Seja $\mathbf{v}$ um vetor unitário em uma das direções principais, ainda desconhecidas, e $\boldsymbol{\sigma}$ o tensor de tensões de Cauchy em Q. Como não há tensões de cisalhamento em um plano perpendicular a $\mathbf{v}$, o vetor de tensão $\mathbf{p}$ é paralelo a $\mathbf{v}$, ou seja,

$$p_i = \lambda v_i. \tag{4.9}$$

Usando a transformação de tensão de Cauchy, Equação (4.8), em sua forma indicial, e a expressão (1.12) do delta de Kronecker na Equação (4.9), obtemos

$$(\sigma_{ji} - \lambda\delta_{ji})v_j = 0, \tag{4.10}$$

ou, matricialmente,

$$\mathbf{v}([\boldsymbol{\sigma}] - \lambda\mathbf{I}) = \mathbf{0}, \tag{4.11}$$

onde $\mathbf{I}$ é a matriz identidade. A Equação (4.10) representa um sistema de três equações lineares homogêneas cujas incógnitas são os cossenos diretores v_1, v_2 e v_3 de $\mathbf{v}$, os quais também devem satisfazer

$$v_i v_i = 1. \tag{4.12}$$

O sistema (4.11) admite solução não-trivial se o determinante da matriz dos coeficientes é zero,

$$|\sigma_{ji} - \lambda\delta_{ji}| = 0, . \tag{4.13}$$

As raízes da Equação (4.13), denominada *equação característica*, são as tensões principais procuradas e correspondem aos *autovalores* da matriz $[\boldsymbol{\sigma}]$. Os *autovetores* de $[\boldsymbol{\sigma}]$ correspondem às direções principais em Q.

As tensões principais são quantidades físicas que não dependem do sistema de coordenadas no qual os componentes de tensão foram inicialmente tomados, sendo, portanto, *invariantes* do estado de tensão.

4.3 Deformações

Na Seção 4.2, vimos que, quando submetido a um sistema de forças de volume e de superfície, um corpo assume uma configuração deformada caracterizada por uma modificação da posição inicial das partículas que o constituem. A modificação da posição de uma partícula é denominada *deslocamento* da partícula. A descrição da *deformação* de um corpo é definida em termos do *campo de deslocamentos* associado a cada partícula do corpo em um instante de tempo t. Essa descrição, baseada na cinemática clássica não relativística, requer o conhecimento de onde o corpo *está* no instante t e de onde o corpo *estava* antes da deformação, no instante $t_0 = 0$.

De acordo com TRUESDELL [118], quatro tipos de descrição do movimento de um *continuum*, enumeradas a seguir, são comumente empregadas.

1. *Descrição material.* As variáveis independentes são a partícula X e o tempo t. A equação $\mathbf{x} = \mathbf{x}(X, t)$ define simbolicamente a posição $\mathbf{x}$ ocupada pela partícula X no tempo t.

2. *Descrição referencial.* As variáveis independentes são a posição $\mathbf{X}$ da partícula X em uma configuração de referência e o tempo t. Quando a configuração de referência é a configuração inicial em $t_0 = 0$ (correspondente ao estado anterior à aplicação das forças de volume e de superfície, ou ao estado indeformado), a descrição referencial é também chamada de *descrição Lagrangeana* do movimento.[2]

3. *Descrição espacial.* As variáveis independentes são a *posição atual* $\mathbf{x}$ ocupada pela partícula X no tempo t e o tempo t. A descrição espacial é também chamada de *descrição Euleriana* do movimento.

4. *Descrição relativa.* As variáveis independentes são a posição atual $\mathbf{x}$ da partícula X e um tempo variável τ (definido quando a partícula ocupava outra posição diferente de $\mathbf{x}$). (Não utilizaremos a descrição relativa em nossas discussões sobre deformação. Veja, por exemplo, MALVERN [72] para maiores detalhes.)

As descrições do movimento mais interessantes para os nossos propósitos são as descrições referencial e espacial. Consideraremos como configuração de referência a configuração definida no tempo $t_0 = 0$ (descrição Lagrangeana do movimento) e tomaremos a posição de referência $\mathbf{X}$ da partícula X como um identificador da partícula. Portanto, o termo "$\mathbf{x}$ é o lugar ocupado no tempo t pela partícula $\mathbf{X}$" significa, na verdade, que "$\mathbf{x}$ é o lugar ocupado no tempo t pela partícula que ocupa a posição $\mathbf{X}$ na configuração de referência". Simbolicamente,

$$\mathbf{x} = \mathbf{x}(\mathbf{X}, t) = x_i(X_1, X_2, X_3, t), \tag{4.14}$$

onde $\mathbf{X} = (X_1, X_2, X_3)$ são as *coordenadas materiais* da partícula.

Na Equação (4.14), a posição onde a partícula *está* é definida em termos de onde a partícula *estava*. Em uma descrição Euleriana o movimento é descrito em termos da posição no espaço ocupado pela partícula X no tempo t,

$$\mathbf{X} = \mathbf{X}(\mathbf{x}, t) = X_i(x_1, x_2, x_3, t), \tag{4.15}$$

onde $\mathbf{x} = (x_1, x_2, x_3)$ são as *coordenadas espaciais* da partícula. Na Equação (4.15) a posição onde a partícula *estava* é definida em termos de onde a partícula *está*. Embora as coordenadas espacial e material possam ser medidas em relação a sistemas de coordenadas distintos, usualmente o mesmo sistema de referência para $\mathbf{x}$ e $\mathbf{X}$ é utilizado.

As definições de deformação são baseadas em medidas quantitativas de certos tipos de deslocamentos relativos entre partículas vizinhas. Consideremos, então, uma partícula X que ocupa uma posição $\mathbf{X}$ em relação à configuração de referência, Figura 4.6. Portanto, $\mathbf{X}$ é a posição onde a partícula X *estava* em $t_0 = 0$. Após a aplicação de forças externas, a partícula passa a ocupar a posição $\mathbf{x}$. Portanto, $\mathbf{x}$ é a posição onde a partícula X *está* no instante t. $\mathbf{u}$ é o deslocamento da partícula X de sua posição inicial $\mathbf{X}$ para a posição corrente $\mathbf{x}$.

[2] A descrição material, em termos da partícula X, é muito confundida com a descrição referencial, em termos da posição $\mathbf{X}$ ocupada pela partícula X em uma configuração de referência.

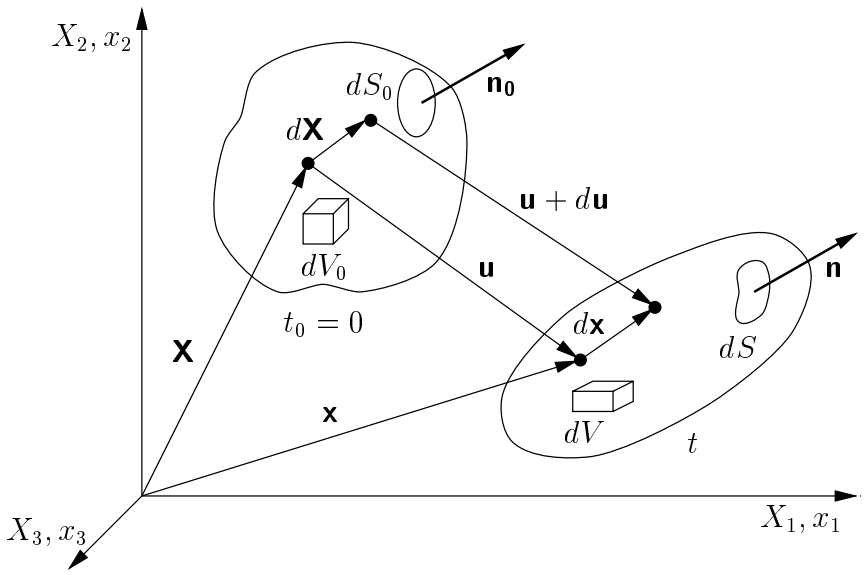


Figura 4.6: Cinemática da partícula.

As deformações podem ser definidas em termos da configuração não deformada, usualmente chamada *formulação Lagrangeana*, e em termos da configuração deformada, chamada *formulação Eulereana*. Usando a Equação (4.14), a relação entre $d\mathbf{x}$ e $d\mathbf{X}$ pode ser escrita como

$$d\mathbf{x} = \mathbf{F}\, d\mathbf{X} = \frac{\partial \mathbf{x}}{\partial \mathbf{X}}\, d\mathbf{X}, \tag{4.16}$$

onde $F_{ij} = \frac{\partial x_i}{\partial X_j}$ são os componentes do *tensor de gradientes de deformação* $\mathbf{F}$ em relação à configuração não deformada. O tensor $\mathbf{F}$ relaciona uma "fibra" $d\mathbf{X}$ do material antes da deformação com uma "fibra" $d\mathbf{x}$ depois da deformação. (Embora os componentes do tensor de gradientes de deformação sejam finitos, considera-se, na Equação (4.16), a deformação de uma vizinhança infinitesimal da partícula X. Usualmente, podemos interpretar $d\mathbf{x}$ como a posição ocupada pelo material deformado $d\mathbf{X}$).

A determinação do *campo de deformações* de um corpo pode ser efetuada a partir da diferença dos quadrados dos comprimentos infinitesimais das fibras antes e depois da deformação, Figura 4.7.

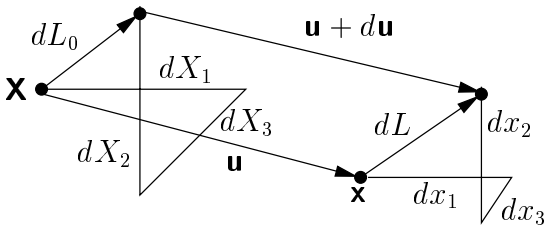


Figura 4.7: Deformação de um elemento de linha infinitesimal.

Na formulação Lagrangeana as seguintes relações são válidas [72]:

$$\begin{aligned} dL^2 - dL_0^2 &= 2d\mathbf{X} \cdot \mathbf{E} \cdot d\mathbf{X}, \quad \text{ou} \\ dL^2 - dL_0^2 &= 2dX_i\, E_{ij}\, dX_j. \end{aligned} \tag{4.17}$$

Na formulação Euleriana, temos

$$\begin{aligned} dL^2 - dL_0^2 &= 2d\mathbf{x} \cdot \mathbf{E}^* \cdot d\mathbf{x}, \quad \text{ou} \\ dL^2 - dL_0^2 &= 2dx_i\, E^*_{ij}\, dx_j. \end{aligned} \tag{4.18}$$

Nas equações acima, $\mathbf{E}$ é o *tensor de deformações de Green-Lagrange* e $\mathbf{E}^*$ é o *tensor de deformações de Almansi*, ou Euleriano. Esses tensores são definidos, respectivamente, pelas expressões (veja, por exemplo, KANE [57, página 110])

$$E_{ij} = \frac{1}{2}\left[\frac{\partial u_i}{\partial X_j} + \frac{\partial u_j}{\partial X_i} + \frac{\partial u_k}{\partial X_i}\frac{\partial u_k}{\partial X_j}\right] \tag{4.19}$$

e

$$E^*_{ij} = \frac{1}{2}\left[\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} - \frac{\partial u_k}{\partial x_i}\frac{\partial u_k}{\partial x_j}\right]. \tag{4.20}$$

A partir da Equação (4.16) e de sua relação inversa podemos escrever os tensores de deformação $\mathbf{E}$ e $\mathbf{E}^*$ como

$$\mathbf{E} = \frac{1}{2}[\mathbf{F}^T \cdot \mathbf{F} - \mathbf{1}] \quad \text{e} \quad \mathbf{E}^* = \frac{1}{2}[\mathbf{1} - (\mathbf{F}^{-1})^T \cdot \mathbf{F}^{-1}], \tag{4.21}$$

com componentes Cartesianos dados por

$$E_{ij} = \frac{1}{2}\left[\frac{\partial x_k}{\partial X_i}\frac{\partial x_k}{\partial X_j} - \delta_{ij}\right] \quad \text{e} \quad E^*_{ij} = \frac{1}{2}\left[\delta_{ij} - \frac{\partial X_k}{\partial x_i}\frac{\partial X_k}{\partial x_j}\right]. \tag{4.22}$$

A única suposição que admitimos até o momento, e a partir da qual as definições apresentadas nesse Capítulo foram fundamentadas, foi a hipótese da continuidade. Portanto, as equações de tensões e deformações desenvolvidas até aqui são gerais e independentes do meio contínuo considerado. Introduziremos agora uma nova hipótese.

PEQUENOS DESLOCAMENTOS, PEQUENAS DEFORMAÇÕES. Quando submetidos a forças de volume e de superfície, os corpos apresentarão pequenos deslocamentos e pequenas deformações.

Esta hipótese, determinada na prática pelas condições de utilização de estruturas civis, simplifica consideravalmente nossos modelos matemáticos. Se as derivadas parciais dos deslocamentos u_i em relação às coordenadas materiais X_i são todas pequenas quando comparadas com a unidade, os quadrados e produtos dessas derivadas podem ser desconsiderados quando comparados com os termos lineares. Considerando a hipótese anterior, as Equações (4.19) e (4.20) podem ser simplificadamente escritas como

$$\epsilon_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial X_j} + \frac{\partial u_j}{\partial X_i}\right) \tag{4.23}$$

e

$$\epsilon^*_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right), \tag{4.24}$$

onde E foi trocado por ϵ para denotar pequenas deformações. Além disso, podemos ignorar as distinções entre as formulações Lagrangeana e Euleriana e escrever o tensor de pequenas deformações como

$$\epsilon_{ij} = \frac{1}{2}\,(u_{i,j} + u_{j,i}). \tag{4.25}$$

Na teoria linearizada da elasticidade, a Equação (4.25) é usualmente obtida como segue. Sejam duas partículas vizinhas X_P e X_Q, mostradas na Figura 4.8. O (pequeno) *deslocamento relativo* de X_Q em relação a X_P é

$$d\mathbf{u} = \mathbf{u}_{X_Q} - \mathbf{u}_{X_P}. \tag{4.26}$$

O *deslocamento relativo unitário* é $d\mathbf{u}/dL$, onde dL é o comprimento da "fibra" in-

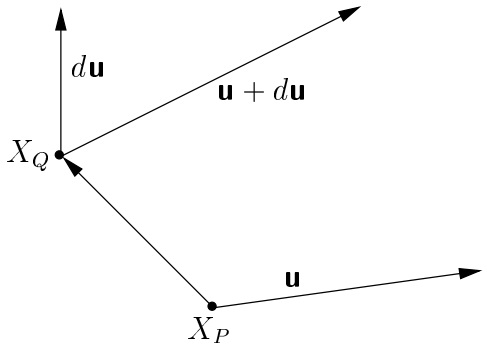


Figura 4.8: Deslocamento relativo entre duas partículas.

finitesimal representada pelo vetor $d\mathbf{X} = \mathbf{X}_P\mathbf{X}_Q$. Os componentes retangulares de $d\mathbf{u}$ são

$$\frac{du_i}{dL} = \frac{\partial u_i}{\partial X_j}\frac{dX_j}{dL} \tag{4.27}$$

(somatório em j), ou

$$\begin{bmatrix} \dfrac{du_x}{dL} \\ \dfrac{du_y}{dL} \\ \dfrac{du_z}{dL} \end{bmatrix} = \begin{bmatrix} \dfrac{\partial u_x}{\partial X} & \dfrac{\partial u_x}{\partial Y} & \dfrac{\partial u_x}{\partial Z} \\ \dfrac{\partial u_y}{\partial X} & \dfrac{\partial u_y}{\partial Y} & \dfrac{\partial u_y}{\partial Z} \\ \dfrac{\partial u_z}{\partial X} & \dfrac{\partial u_z}{\partial Y} & \dfrac{\partial u_z}{\partial Z} \end{bmatrix} \begin{bmatrix} \dfrac{dX}{dL} \\ \dfrac{dY}{dL} \\ \dfrac{dZ}{dL} \end{bmatrix}. \tag{4.28}$$

A Equação (4.28) pode ser escrita como

$$\frac{d\mathbf{u}}{dL} = \mathbf{u}\overleftarrow{\nabla}\cdot\mathbf{n} \qquad \text{ou} \qquad \frac{d\mathbf{u}}{dL} = \mathbf{J}_u\,\mathbf{n}, \tag{4.29}$$

onde $\mathbf{n}$ é o vetor unitário na direção de $d\mathbf{X}$. A matriz quadrada $\mathbf{J}_u$ é chamada *matriz de deslocamento relativo unitário*. Podemos escrever $\mathbf{J}_u$ como a soma de duas matrizes, uma simétrica e outra anti-simétrica. Chamaremos a parte simétrica de $\mathbf{J}_u$ de $[\boldsymbol{\epsilon}]$ e a

parte anti-simétrica de $[\boldsymbol{\omega}]$. As expressões são

$$[\boldsymbol{\epsilon}] = \begin{bmatrix} \dfrac{\partial u_x}{\partial X} & \dfrac{1}{2}\left(\dfrac{\partial u_x}{\partial Y} + \dfrac{\partial u_y}{\partial X}\right) & \dfrac{1}{2}\left(\dfrac{\partial u_x}{\partial Z} + \dfrac{\partial u_z}{\partial X}\right) \\ \dfrac{1}{2}\left(\dfrac{\partial u_y}{\partial X} + \dfrac{\partial u_x}{\partial Y}\right) & \dfrac{\partial u_y}{\partial Y} & \dfrac{1}{2}\left(\dfrac{\partial u_y}{\partial Z} + \dfrac{\partial u_z}{\partial Y}\right) \\ \dfrac{1}{2}\left(\dfrac{\partial u_z}{\partial X} + \dfrac{\partial u_x}{\partial Z}\right) & \dfrac{1}{2}\left(\dfrac{\partial u_z}{\partial Y} + \dfrac{\partial u_y}{\partial Z}\right) & \dfrac{\partial u_z}{\partial Z} \end{bmatrix}$$

$$\boldsymbol{\epsilon} = \frac{1}{2}\left(\mathbf{u}\overleftarrow{\nabla} + \overrightarrow{\nabla}\mathbf{u}\right), \tag{4.30}$$

e

$$[\boldsymbol{\omega}] = \begin{bmatrix} 0 & \dfrac{1}{2}\left(\dfrac{\partial u_x}{\partial Y} - \dfrac{\partial u_y}{\partial X}\right) & \dfrac{1}{2}\left(\dfrac{\partial u_x}{\partial Z} - \dfrac{\partial u_z}{\partial X}\right) \\ \dfrac{1}{2}\left(\dfrac{\partial u_y}{\partial X} - \dfrac{\partial u_x}{\partial Y}\right) & 0 & \dfrac{1}{2}\left(\dfrac{\partial u_y}{\partial Z} - \dfrac{\partial u_z}{\partial Y}\right) \\ \dfrac{1}{2}\left(\dfrac{\partial u_z}{\partial X} - \dfrac{\partial u_x}{\partial Z}\right) & \dfrac{1}{2}\left(\dfrac{\partial u_z}{\partial Y} - \dfrac{\partial u_y}{\partial Z}\right) & 0 \end{bmatrix}$$

$$\boldsymbol{\omega} = \frac{1}{2}\left(\mathbf{u}\overleftarrow{\nabla} - \overrightarrow{\nabla}\mathbf{u}\right). \tag{4.31}$$

O tensor $\boldsymbol{\epsilon}$ é o tensor de pequenas deformações, como antes. O tensor $\boldsymbol{\omega}$ é o tensor de pequenas rotações do campo de deslocamentos infinitesimais $\mathbf{u}$.

4.4 Equações de Equilíbrio

O conceito de equilíbrio utilizado para derivar a transformação de tensão de Cauchy, Equação (4.8), também pode ser aplicado para derivar uma relação fundamental entre as derivadas espaciais dos componentes de tensão. Considerando o equilíbrio estático do corpo da Figura 4.1, podemos escrever, a partir das Equações (4.1) e (4.3),

$$\int_V \mathbf{b}\, dV + \int_S \mathbf{p}\, dS = 0. \tag{4.32}$$

Podemos transformar a integral de superfície da Equação (4.32) em uma integral de volume, primeiro aplicando a transformação de tensão de Cauchy e, segundo, utilizando o teorema da divergência,

$$\int_S p_i\, dS = \int_S \sigma_{ji} n_j\, dS = \int_V \frac{\partial \sigma_{ji}}{\partial x_j}\, dV. \tag{4.33}$$

Combinando as Equações (4.32) e (4.33), obtemos a *equação de equilíbrio*

$$\int_V \frac{\partial \sigma_{ji}}{\partial x_j}\, dV + \int_V b_i\, dV = 0, \tag{4.34}$$

ou

$$\frac{\partial \sigma_{ji}}{\partial x_j} + b_i = 0. \tag{4.35}$$

(Essa equação é uma expressão particular, para o caso de equilíbrio estático, do princípio mais geral de conservação de *momentum*, do qual resultam as equações de movimento de Cauchy [72, página 214] $\frac{\partial \sigma_{ji}}{\partial x_j} + b_i = \rho \ddot{u}_i$, onde ρ é a massa do corpo e $\mathbf{u}$ é o vetor de deslocamentos.)

Note que a Equação (4.35) expressa o equilíbrio do corpo na configuração deformada em função de variáveis tomadas nessa configuração. (Estamos utilizando o tensor de tensões de Cauchy.) Esse fato pode nos trazer algumas complicações porque, na configuração deformada (após a aplicação das forças externas), não conhecemos, por exemplo, o volume do corpo. Podemos escrever a equação de equilíbrio na configuração deformada em função de variáveis tomadas na configuração de referência como

$$\overrightarrow{\nabla} \cdot [\mathbf{T} \cdot \mathbf{F}^T] + \mathbf{b} = \mathbf{0}, \tag{4.36}$$

onde $\mathbf{T}$ é o (simétrico) *segundo tensor de Piola-Kirchhoff*, dado por [72]

$$\mathbf{T} = \mathbf{F}^{-1} \cdot \boldsymbol{\sigma} \cdot (\mathbf{F}^{-1})^T. \tag{4.37}$$

Felizmente, como estamos admitindo pequenos deslocamentos e pequenos gradientes de deformação, podemos ignorar as diferenças de volume entre a configuração de referência e a configuração deformada e expressar o equilíbrio através da Equação (4.35).

4.5 Elasticidade Tridimensional

Na Seção 4.2 e na Seção 4.3 consideramos as descrições matemáticas de tensão e de deformação, respectivamente, determinadas a partir da hipótese básica da continuidade e aplicadas, de forma geral, a qualquer material suposto contínuo. Nesta seção, consideraremos as relações entre tensões e deformações que caracterizam o comportamento de um material específico. Essas relações são denominadas de *relações constitutivas*, ou equações constitutivas. Apenas os materiais elásticos serão abordados aqui. Além disso, admitiremos que esses materiais são *homogênos*. Essa nova hipótese a respeito das propriedades dos materiais é definida como a seguir.

HOMOGENEIDADE Um material homogêneo tem propriedades idênticas em todos os pontos.

Um material é chamado *idealmente elástico* quando um corpo formado pelo material recupera completamente sua forma original, após a remoção das forças causadoras da deformação, e quando há uma relação unívoca entre o estado de tensão e de deformação, para uma dada temperatura.[3]

As relações constitutivas elásticas, ou lei de Hooke generalizada, são nove equações que expressam os componentes de tensão como funções lineares homogêneas dos nove componentes de deformação

$$T_{ij} = C_{ijrs}\, E_{rs}, \tag{4.38}$$

[3] Assumiremos que a dependência da temperatura possa ser negligenciada ou que a variação seja suficientemente pequena durante a deformação.

se o tensor de Almansis $\mathbf{E}$ e o segundo tensor de Piola-Kirchhoff $\mathbf{T}$ são tomados como variáveis referidas à configuração de referência. Se admitirmos a hipótese de pequenos deslocamentos, pequenas deformações e pequenas rotações, definida na Seção 4.3, nenhuma distinção precisa ser tomada entre as coordenadas materiais e espaciais, e a Equação (4.38) pode ser escrita em função do tensor de Cauchy $\boldsymbol{\sigma}$ e o tensor de pequenas deformações $\boldsymbol{\epsilon}$ como

$$\sigma_{ij} = C_{ijrs}\,\epsilon_{rs}. \tag{4.39}$$

O tensor de quarta ordem $\mathbf{C}$, *tensor de módulos elásticos*, possui 81 componentes, mas como, na Equação (4.39), $\boldsymbol{\sigma}$ e $\boldsymbol{\epsilon}$ são simétricos, podemos supor, sem perda de generalidade, que $C_{ijrs} = C_{jirs}$ e $C_{ijrs} = C_{ijsr}$. Nesse caso, somente 36 componentes de $\mathbf{C}$ são independentes, e a Equação (4.39) pode ser matricialmente escrita como

$$\begin{bmatrix}\sigma_{11}\\ \sigma_{22}\\ \sigma_{33}\\ \sigma_{23}\\ \sigma_{31}\\ \sigma_{12}\end{bmatrix} = \begin{bmatrix}C_{11} & C_{12} & C_{13} & C_{14} & C_{15} & C_{16}\\ C_{21} & C_{22} & C_{23} & C_{24} & C_{25} & C_{26}\\ C_{31} & C_{32} & C_{33} & C_{34} & C_{35} & C_{36}\\ C_{41} & C_{42} & C_{43} & C_{44} & C_{45} & C_{46}\\ C_{51} & C_{52} & C_{53} & C_{54} & C_{55} & C_{56}\\ C_{61} & C_{62} & C_{63} & C_{64} & C_{65} & C_{66}\end{bmatrix} \begin{bmatrix}\epsilon_{11}\\ \epsilon_{22}\\ \epsilon_{33}\\ 2\epsilon_{23}\\ 2\epsilon_{31}\\ 2\epsilon_{12}\end{bmatrix}. \tag{4.40}$$

Além das suposições a respeito da continuidade e homogeneidade do *continuum*, admitidas até agora, particularizaremos nossas equações constitutivas para o caso de materiais *isótropos*.

ISOTROPIA Um material é isótropo em relação a determinadas propriedades se estas propriedades forem as mesmas em todas as direções.

Com a consideração da hipótese da isotropia, o tensor de módulos elásticos $\mathbf{C}$ é um tensor isotrópico de quarta ordem definido como (veja MALVERN [72, página 277])

$$C_{ijrs} = \lambda\,\delta_{ij}\delta_{rs} + \mu\,(\delta_{ir}\delta_{js} + \delta_{is}\delta_{jr}), \tag{4.41}$$

onde λ e μ são as constantes de Lamé e δ_{ij} é o delta de Kronecker. A versão isotrópica da lei de Hooke, Equação (4.39), fica,

$$\sigma_{ij} = \lambda\,\epsilon_{kk}\,\delta_{ij} + 2\mu\,\epsilon_{ij}. \tag{4.42}$$

A relação inversa da Equação (4.39), a qual expressa deformações em função de tensões, é dada por

$$\epsilon_{ij} = -\frac{\lambda\,\delta_{ij}}{2\mu\,(3\lambda + 2\mu)}\sigma_{kk} + \frac{1}{2\mu}\sigma_{ij}. \tag{4.43}$$

As constantes de Lamé são relacionadas com o módulo de cisalhamento G, o módulo de elasticidade E (ou módulo de Young) e o coeficiente de Poisson ν como segue:

$$G = \mu = \frac{E}{2(1+\nu)} \qquad \text{e} \qquad \lambda = \frac{\nu\,E}{(1+\nu)(1-\nu)}, \tag{4.44}$$

ou

$$E = \frac{\mu\,(3\lambda + 2\mu)}{\lambda + \mu} \qquad \text{e} \qquad \nu = \frac{\lambda}{2(\lambda + \mu)}. \tag{4.45}$$

Usando as identidades (4.44), a Equação (4.43) é escrita como

$$\epsilon_{ij} = -\frac{\nu}{E}\,\sigma_{kk}\,\delta_{ij} + \frac{1+\nu}{E}\,\sigma_{ij}. \tag{4.46}$$

Matricialmente, podemos organizar a forma isotrópica da Equação (4.40) como

$$[\boldsymbol{\sigma}] = \mathbf{C}\,[\boldsymbol{\epsilon}], \tag{4.47}$$

onde

$$[\boldsymbol{\sigma}] = \begin{bmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{zz} \\ \sigma_{xy} \\ \sigma_{yz} \\ \sigma_{zx} \end{bmatrix}, \qquad [\boldsymbol{\epsilon}] = \begin{bmatrix} \epsilon_{xx} \\ \epsilon_{yy} \\ \epsilon_{zz} \\ \epsilon_{xy} \\ \epsilon_{yz} \\ \epsilon_{zx} \end{bmatrix} \tag{4.48}$$

e

$$\mathbf{C} = \frac{E}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1-\nu & \nu & \nu & 0 & 0 & 0 \\ \nu & 1-\nu & \nu & 0 & 0 & 0 \\ \nu & \nu & 1-\nu & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1-2\nu}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1-2\nu}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1-2\nu}{2} \end{bmatrix}. \tag{4.49}$$

Inversamente,

$$[\boldsymbol{\epsilon}] = \mathbf{C}^{-1}\,[\boldsymbol{\sigma}], \tag{4.50}$$

sendo

$$\mathbf{C}^{-1} = \frac{1}{E} \begin{bmatrix} 1 & -\nu & -\nu & 0 & 0 & 0 \\ -\nu & 1 & -\nu & 0 & 0 & 0 \\ -\nu & -\nu & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2(1+\nu) & 0 & 0 \\ 0 & 0 & 0 & 0 & 2(1+\nu) & 0 \\ 0 & 0 & 0 & 0 & 0 & 2(1+\nu) \end{bmatrix}. \tag{4.51}$$

4.6 Modelo Matemático Geral

Organizaremos, nessa Seção, as equações de campo da teoria da elasticidade linear isotrópica que constituem o modelo matemático do problema fundamental definido no Capítulo 1. A formulação usa a descrição referencial Lagrangeana em termos das coordenadas materiais $\mathbf{X}$ de uma partícula na configuração não deformada. O modelo é baseado nas hipóteses a seguir.

- Os componentes do gradiente de deslocamentos são pequenos quando comparados com a unidade;
- A lei de Hooke generalizada é a equação constitutiva do material; e
- As equações de equilíbrio são satisfeitas na configuração de referência não deformada.

As equações do modelo são:

$$\text{6 equações deformação-deslocamento} \quad \epsilon_{ij} = \frac{1}{2}(u_{i,j} + u_{j,i}), \tag{4.52}$$

$$\text{6 equações constitutivas} \quad \sigma_{ij} = \lambda \epsilon_{kk} \delta_{ij} + 2\mu \epsilon_{ij}, \text{ e} \tag{4.53}$$

$$\text{3 equações de equilíbrio} \quad \sigma_{ji,j} + b_i = 0, \tag{4.54}$$

em um total de 15 equações para 3 deslocamentos, 6 deformações e 6 tensões. As Equações (4.52), (4.53) e (4.54) representam a sentença matemática do problema fundamental e, uma vez, resolvidas, nos fornecem os campos de deslocamentos, tensões e deformações de um sólido submetido à ação de um sistema equilibrado de forças externas (admitidas as hipóteses anteriores).

Podemos ter as seguintes *condições de contorno* para as equações de campo do modelo:

1. *Deslocamentos prescritos*, com os três deslocamentos u_i conhecidos no contorno do corpo.
2. *Forças de superfície prescritas*, com os três componentes de tensão $p_i = \sigma_{ji} n_j$ conhecidos no ponto onde o contorno do corpo possui normal $\mathbf{n}$.
3. *Condições de contorno mistas*, com (a)deslocamentos prescritos sobre uma parte do contorno e forças de superfície prescritas no restante do contorno, ou (b)u_i ou p_i prescrito em cada ponto do contorno, mas não ambos.

Equação de Navier A solução de um problema envolvendo 15 equações para 15 incógnitas não é uma tarefa fácil. Há várias maneiras de reformularmos o modelo matemático do problema fundamental com o propósito de obtermos um número menor de incógnitas e um número menor de equações. O método mais direto consiste em substituir a Equação (4.52) em (4.53) para obtermos as tensões em termos dos gradientes de deslocamentos, e, então, substituir o resultado na Equação (4.54) para obtermos três equações diferenciais parciais de segunda ordem para os três componentes de deslocamento:

$$G\, u_{j,kk} + \frac{G}{1-2\nu}\, u_{k,jk} + b_j = 0, \tag{4.55}$$

ou, vetorialmente,

$$G\, \nabla^2 \mathbf{u} + \frac{G}{1-2\nu}\, \nabla(\nabla \cdot \mathbf{u}) + \mathbf{b} = \mathbf{0}, \tag{4.56}$$

onde $\mathbf{0}$ é o vetor nulo. A Equação (4.55), ou sua forma vetorial (4.56), é conhecida como *equação de Navier da elasticidade*.[4] Substituindo a Equação (4.52) em (4.53) e o resultado na transformação de tensão de Cauchy (4.8), obtemos uma expressão para as forças de superfície prescritas $\overline{p}_i$ em função dos componentes de deslocamento:

$$\frac{2G\nu}{1-2\nu}\, u_{k,k}\, n_i + G\,(u_{i,j} + u_{j,i})n_j = \overline{p}_i, \tag{4.57}$$

ou, vetorialmente,

$$\frac{2G\nu}{1-2\nu}\,(\nabla \cdot \mathbf{u})\mathbf{n} + G\,(\mathbf{u}\overleftarrow{\nabla} + \overrightarrow{\nabla}\mathbf{u}) \cdot \mathbf{n} = \overline{\mathbf{p}}. \tag{4.58}$$

4.6.1 Solução Fundamental de Kelvin

A solução do modelo matemático do problema fundamental, como já sabemos, só pode ser *analiticamente* obtida para alguns casos particulares de geometria, carregamentos e condições de contorno. Apresentaremos, nessa Seção, a *solução fundamental de Kelvin* para o problema fundamental de um sólido infinito sujeito a uma força concentrada unitária aplicada em um ponto qualquer do sólido. Embora possa parecer somente um exercício teórico, utilizaremos a solução fundamental de Kelvin na formulação do método dos elementos de contorno, conforme veremos no Capítulo 5.

Uma força concentrada em um ponto p (certamente uma anomalia em mecânica do contínuo) pode ser considerada, por exemplo, como o caso limite de uma força de volume atuando em uma esfera centrada em p quando o raio da esfera tende a zero e a intensidade da força de volume aumenta de tal forma que sua resultante permaneça constante. Matematicamente, descreveremos essa abstração através da *função delta de Dirac*, a qual pode ser definida como um limite da função pulso retangular unitário [57] mostrada na Figura 4.9. A função $F(x, \xi, \varepsilon)$ centrada em ξ de largura ε vale

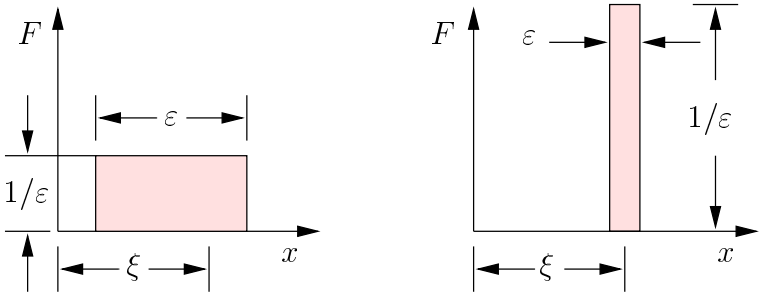


Figura 4.9: Função pulso retangular unitário.

$$F(x, \xi, \varepsilon) = \begin{cases} 0 & \text{se } x < \xi - \frac{\varepsilon}{2} \vee x > \xi + \frac{\varepsilon}{2}, \\ \frac{1}{\varepsilon} & \text{se } \xi - \frac{\varepsilon}{2} \leq x \leq \xi + \frac{\varepsilon}{2} \end{cases} \tag{4.59}$$

e sua integral sobre o domínio x é igual a unidade. A função delta de Dirac $\delta(x - \xi)$ é o limite

$$\delta(x - \xi) = \lim_{\varepsilon \to 0} F(x, \xi, \varepsilon) = \begin{cases} 0 & \text{se } x \neq \xi, \\ \infty & \text{se } x = \xi, \end{cases} \tag{4.60}$$

[4] Utilizamos a equação de Navier no exemplo do Capítulo 2.

sendo

$$\int_{-\infty}^{+\infty} \delta(x-\xi)\,dx = 1. \tag{4.61}$$

A Equações (4.60) nos mostra que, à medida que a região ε sobre a qual a função atua torna-se menor, sua intensidade aumenta de tal forma que a resultante no domínio x, Equação (4.61), permaneça constante. Uma propriedade muito útil da função delta de Dirac, conhecida como *propriedade de seleção*, é que

$$\int_{-\infty}^{+\infty} g(x)\delta(x-\xi)\,dx = f(\xi), \tag{4.62}$$

ou seja, o produto da função delta de Dirac e outra função g, integrado sobre o domínio sobre o qual a função delta de Dirac atua, é igual ao valor da função g no ponto de aplicação ξ da função delta de Dirac. Utilizaremos, ainda, as notações $\delta(\xi, x)$ ou $\delta(p, q)$ para a função delta de Dirac, onde q é um ponto qualquer de um domínio e p é o ponto de aplicação da função delta de Dirac (entendido como ponto de aplicação da carga unitária concentrada). Chamaremos o ponto p, pertencente ao domínio, de *ponto fonte*.

Seja, então, um sólido elástico definido por um volume infinito V^* e, conseqüentemente, uma superfície S^* no infinito. Após a aplicação de uma força unitária

$$\mathbf{b}_i^* = \delta(p, q)\,\mathbf{i}_i \tag{4.63}$$

em um ponto fonte $p \in V^*$, o sólido hipotético assume um estado de equilíbrio definido pelos deslocamentos $\mathbf{u}_i^*$ e forças de superfície $\mathbf{p}_i^*$ em S^*, de acordo com as equações (4.56) e (4.58), onde o índice i indica a direção de aplicação da força unitária. Para $i = 1, 2, 3$ temos

$$\mathbf{u}^* = \begin{bmatrix} u_{11}^* & u_{12}^* & u_{13}^* \\ u_{21}^* & u_{22}^* & u_{23}^* \\ u_{31}^* & u_{32}^* & u_{33}^* \end{bmatrix} \quad \text{ou} \quad u_{ij}^*, \tag{4.64}$$

$$\mathbf{p}^* = \begin{bmatrix} p_{11}^* & p_{12}^* & p_{13}^* \\ p_{21}^* & p_{22}^* & p_{23}^* \\ p_{31}^* & p_{32}^* & p_{33}^* \end{bmatrix} \quad \text{ou} \quad p_{ij}^*. \tag{4.65}$$

Nessas equações, $u_{ij}^*(p, q)$ e $p_{ij}^*(p, q)$ representam, respectivamente, os deslocamentos e forças de superfície na direção j no ponto q, correspondentes a uma força concentrada unitária atuando na direção i aplicada no ponto p, como indicado na Figura 4.10. A função $u_{ij}^*(p, q)$, *solução fundamental de Kelvin* para deslocamentos, é[5]

$$u_{ij}^*(p, q) = \frac{1}{16\pi(1-\nu)G\,r}\Big\{(3-4\nu)\,\delta_{ij} + r_{,i}\,r_{,j}\Big\}, \tag{4.66}$$

onde r é a distância entre p e q. A partir da Equação (4.66) e das relações deformação-deslocamento (4.52), as deformações em qualquer ponto q causadas por uma carga concentrada unitária aplicada em p na direção i são

$$\epsilon_{ijk}^*(p, q) \;=\; -\frac{1}{16\pi(1-\nu)Gr^2}\Big\{(1-2\nu)(r_{,k}\delta_{ij} + r_{,j}\delta_{ik}) - r_{,i}\delta_{jk} + 3r_{,i}r_{,j}r,k\Big\}, \tag{4.67}$$

[5]KANE [57, página 132] mostra detalhadamente o desenvolvimento da solução fundamental de Kelvin.

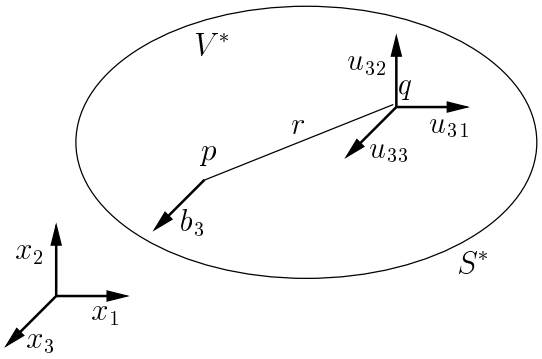


Figura 4.10: Forças e deslocamentos em V^*.

as quais, considerando a lei de Hooke (4.53), definem as tensões em qualquer ponto q causadas por uma carga concentrada unitária aplicada em p na direção i

$$\sigma^*_{ijk}(p,q) \;=\; -\frac{1}{8\pi(1-\nu)r^2}\Big\{(1\,-\,2\nu)(r_{,k}\delta_{ij}\,+\,r_{,j}\delta_{ki}\,-\,r_{,i}\delta_{jk})\,+\,3r_{,i}r_{,j}r_{,k}\Big\}. \quad (4.68)$$

Aplicando a transformação de tensão de Cauchy (4.8) na Equação (4.68), temos, finalmente

$$p^*_{ij}(p,q) \;=\; -\frac{1}{8\pi(1-\nu)r^2}\Big\{[(1-2\nu)\delta_{ij}\,+3r_{,i}r_{,j}]r_{,k}n_k\,-\,(1-2\nu)(r_{,i}n_j\,-\,r_{,j}n_i)\Big\}. \quad (4.69)$$

4.7 Modelos Simplificados

Em muitas aplicações práticas de engenharia empregamos elementos estruturais que possuem uma ou duas dimensões pequenas, quando comparadas com as demais dimensões do objeto. Exemplos bastante comuns são as vigas e cascas de um edifício. Nesses casos, podemos obter informações úteis sobre o comportamento da estrutura substituindo o modelo matemático geral, difícil de resolver, por modelos baseados em teorias simplificadas unidimensionais ou bidimensionais. A seguir, veremos dois exemplos de modelos simplificados: membranas e placas. Uma membrana possui comportamento caracterizado por uma distribuição bidimensioanal de tensões denominada *estado plano de tensões*. Em uma placa, ao invés de resolvermos o problema fundamental em termos de campos tridimensionais de tensões e deformações, o resolvemos em termos de certas *resultantes de tensão* e de certos deslocamentos e rotações associadas a essas resultantes. Utilizaremos esses elementos estruturais no modelo mecânico de uma casca, Capítulo 6.

4.7.1 Membranas

Uma *membrana* é um objeto geometricamente definido por duas superfícies planas, tal como ilustrado na Figura 4.11. A distância entre as superfícies, ou espessura da membrana, é admitida pequena quando comparada com as demais dimensões do objeto. O plano equidistante às duas superfícies que definem a membrana é denominado plano médio da membrana. Nos referiremos ao plano médio simplesmente como o "plano"

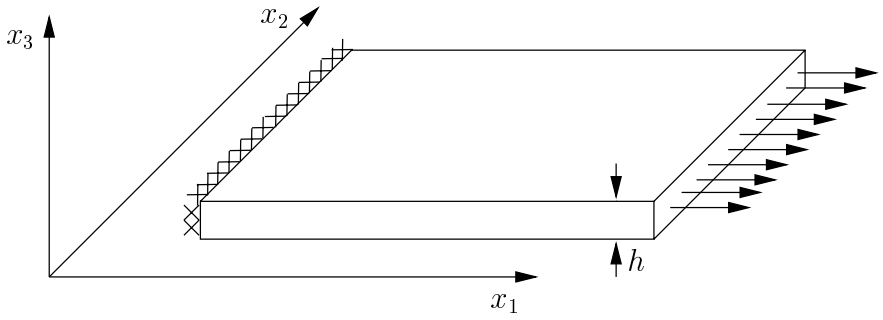


Figura 4.11: Geometria e carregamento de uma membrana.

da membrana. Admitiremos, ainda, que os carregamentos sejam sempre paralelos ao plano, aplicados no contorno e uniformemente distribuídos ao longo da espessura.

O modelo matemático de uma membrana é uma simplificação do modelo geral apresentado na Seção 4.6, caracterizado por uma distribuição bidimensional de tensões definida a partir da suposição de que

$$\sigma_{i3} = \sigma_{3i} = 0, \tag{4.70}$$

sendo a direção 3 perpendicular ao plano da membrana. Podemos supor, também, que as tensões sejam constantes ao longo da espessura da membrana e que todas as hipóteses da elasticidade linear, definidas anteriormente, sejam válidas.[6] O estado de tensões em um ponto qualquer do plano da membrana é então especificado somente pelos componentes σ_{11}, σ_{22} e $\sigma_{12} = \sigma_{21}$ do tensor de tensões e denominado *estado plano de tensões*.

Note que o objeto é livre para contrair ou expandir na direção 3. A partir da condição (4.70), podemos determinar o componente de deformação fora do plano, ϵ_{33}. Usando a lei de Hooke obtemos

$$\epsilon_{33} = -\frac{\lambda}{\lambda + 2\mu}\,\epsilon_{ll}, \quad l = 1, 2. \tag{4.71}$$

(O componente de deformação fora do plano é obtido em função dos componentes de deformação no plano.) A lei de Hooke isótropa para o estado plano de tensões é definida subtituindo a Equação (4.71) na expressão (4.42), resultando

$$\sigma_{ij} = \delta_{ij}\frac{2\lambda\mu}{\lambda + 2\mu}\,\epsilon_{kk} + 2\mu\,\epsilon_{ij}, \quad i, j, k = 1, 2. \tag{4.72}$$

Matricialmente, temos uma particularização da Equação (4.49), onde

$$[\boldsymbol{\sigma}] = \begin{bmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{xy} \end{bmatrix}, \qquad [\boldsymbol{\epsilon}] = \begin{bmatrix} \epsilon_{xx} \\ \epsilon_{yy} \\ \epsilon_{xy} \end{bmatrix} \tag{4.73}$$

e

$$\mathbf{C} = \frac{E}{1 - \nu^2}\begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{bmatrix}. \tag{4.74}$$

[6] A hipótese (4.70) viola algumas das condições de compatibilidade e a variação de tensões ao longo da espessura ocorre, mas podem ser ignoradas para membranas "suficientemente delgadas". TIMOSHENKO [117, página 267] comenta criticamente essa suposição.

Na relação inversa, Equação (4.51), usamos

$$\mathbf{C}^{-1} = \frac{1}{E}\begin{bmatrix} 1 & -\nu & 0 \\ -\nu & 1 & 0 \\ 0 & 0 & 2(1+\nu) \end{bmatrix}. \tag{4.75}$$

4.7.2 Placas

Uma *placa* é um objeto definido geometricamente da mesma forma que uma membrana, ou seja, por duas superfícies planas tal que a distância entre as superfícies, ou espessura da placa, seja pequena em relação às demais dimensões do objeto, Figura 4.12. Tal como na membrana, a superfície equidistante às duas superfícies que definem a placa é denominada plano médio da placa. O carregamento g será sempre aplicado transversal ao plano médio.

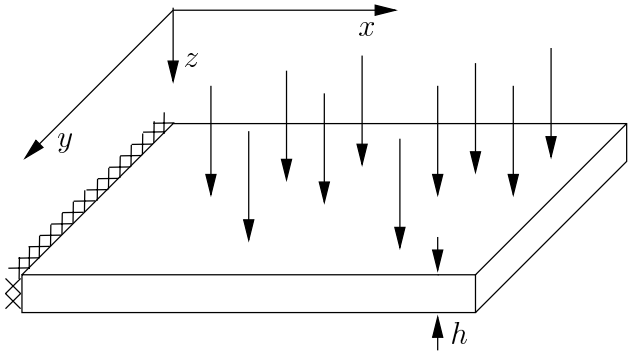


Figura 4.12: Geometria e carregamento de uma placa.

.

A Figura 4.13 mostra o estado de tensão de um elemento de placa definido por planos paralelos aos planos xz e yz. As resultantes da tensão normal σ_{xx} por unidade de área na face x positiva são a *força normal* N_x por unidade de comprimento ao longo da linha AB e o *momento fletor* M_x por unidade de comprimento ao longo da linha AB,

$$N_x = \int_{-h/2}^{h/2} \sigma_{xx}\, dz, \tag{4.76a}$$

$$M_x = \int_{-h/2}^{h/2} z\sigma_{xx}\, dz. \tag{4.76b}$$

As distribuições lineares de força N_x e de momento M_x são estaticamente equivalentes à distribuição superficial de σ_{xx}. Analogamente, temos, na face y positiva,

$$N_y = \int_{-h/2}^{h/2} \sigma_{yy}\, dz, \tag{4.76c}$$

$$M_y = \int_{-h/2}^{h/2} z\sigma_{yy}\, dz. \tag{4.76d}$$

A distribuição da tensão de cisalhamento vertical τ_{xz} por unidade de área é estaticamente equivalente à *força cortante vertical* Q_x por unidade de comprimento ao longo

de AB,

$$Q_x = \int_{-h/2}^{h/2} \tau_{xz}\, dz. \tag{4.76e}$$

Analogamente, em y,

$$Q_y = \int_{-h/2}^{h/2} \tau_{yz}\, dz. \tag{4.76f}$$

As resultantes da tensão de cisalhamento horizontal $\tau_{xy} = \tau_{yx}$ por unidade de área são a *força cortante horizontal* Q_{xy} por unidade de comprimento e o *momento torsor* M_{xy} por unidade de comprimento,

$$N_{xy} = N_{yx} = \int_{-h/2}^{h/2} \tau_{xy}\, dz, \tag{4.76g}$$

$$M_{xy} = M_{yx} = \int_{-h/2}^{h/2} z\tau_{xy}\, dz. \tag{4.76h}$$

A convenção de sinais positivos das resultantes de tensão em um elemento de placa é mostrada na Figura 4.14.

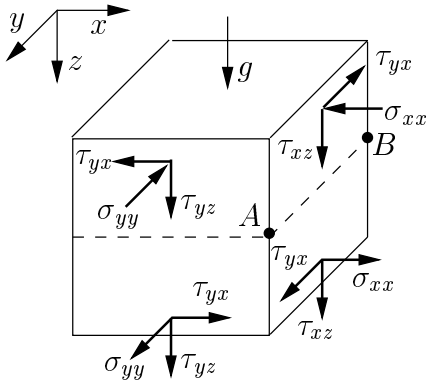


Figura 4.13: Tensões em um elemento de placa.

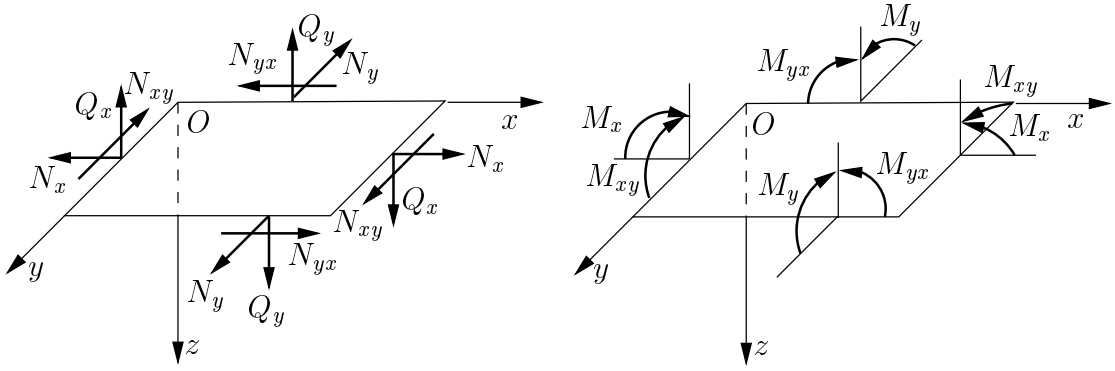


Figura 4.14: Resultantes de tensão em um elemento de placa.

Teoria de Mindlin A primeira e mais importante hipótese do modelo matemático de uma placa de Mindlin é que, durante o processo de deformação, seções planas normais ao plano médio da placa permanecem planas. A segunda hipótese é que as tensões na direção normal ao plano médio são pequenas e podem ser desprezadas. (Se considerarmos a placa constituída de "lâminas", estaremos supondo, com essa hipótese, um estado plano de tensões em cada "lâmina".) Na teoria simplificada de Mindlin, admitimos ainda que a placa seja suficientemente rígida em seu plano médio de tal modo que as *forças de membrana* por unidade de comprimento N_x, N_y e N_{xy}, atuando no plano médio, possam ser desprezadas. Com essas hipóteses, os componentes de deslocamento de um ponto da placa de coordenadas x, y, z, Figura 4.15, são

$$u = z\theta_x(x,y), \qquad v = z\theta_y(x,y), \text{ e} \qquad w = w(x,y), \tag{4.77}$$

onde w é o deslocamento transversal e θ_x e θ_y são as rotações da normal ao plano médio indeformado em relação aos planos xz e yz, respectivamente. A partir das relações

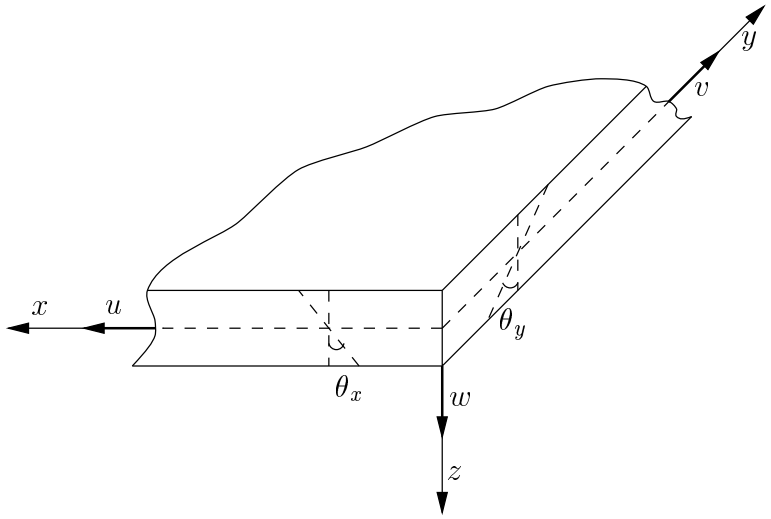


Figura 4.15: Deslocamentos e rotações em uma placa.

deformação-deslocamento (4.52) podemos escrever matricialmente as deformações do ponto, as quais podem ser convenientemente separadas em dois grupos: as deformações de flexão (lineares na espessura)

$$[\boldsymbol{\epsilon}] = \begin{bmatrix} \epsilon_{xx} \\ \epsilon_{yy} \\ \epsilon_{xy} \end{bmatrix} = z \begin{bmatrix} \dfrac{\partial}{\partial x} & 0 \\ 0 & \dfrac{\partial}{\partial y} \\ \dfrac{\partial}{\partial y} & \dfrac{\partial}{\partial x} \end{bmatrix} \begin{bmatrix} \theta_x \\ \theta_y \end{bmatrix} = z\mathbf{L}[\boldsymbol{\theta}] \tag{4.78}$$

e as deformações de cisalhamento transversal (constantes na espessura)

$$[\boldsymbol{\gamma}] = \begin{bmatrix} \gamma_{xz} \\ \gamma_{yz} \end{bmatrix} = \begin{bmatrix} \dfrac{\partial w}{\partial x} \\ \dfrac{\partial w}{\partial y} \end{bmatrix} + \begin{bmatrix} \theta_x \\ \theta_y \end{bmatrix} = [\boldsymbol{\nabla}]w + [\boldsymbol{\theta}]. \tag{4.79}$$

Considerando as relações constitutivas (4.53), podemos escrever (suposto o estado plano de tensões em cada lâmina, material isótropo, etc.) as tensões de flexão

$$[\boldsymbol{\sigma}] = z\mathbf{C}\mathbf{L}[\boldsymbol{\theta}], \tag{4.80}$$

com $\mathbf{C}$ dado pela Equação (4.74), e as tensões de cisalhamento transversal

$$[\boldsymbol{\tau}] = G([\boldsymbol{\nabla}]w + [\boldsymbol{\theta}]). \tag{4.81}$$

Substituindo os componentes de tensão nas Equações (4.76a) de resultantes de tensão correspondentes (desprezando as forças de membrana), obtemos as expressões matriciais para os momentos

$$\mathbf{M} = \begin{bmatrix} M_x \\ M_y \\ M_{xy} \end{bmatrix} = \mathbf{D}\mathbf{L}[\boldsymbol{\theta}], \tag{4.82}$$

onde

$$\mathbf{D} = \frac{Eh^3}{12(1-\nu^2)} \begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{bmatrix}, \tag{4.83}$$

e para as forças cortantes

$$\mathbf{Q} = \begin{bmatrix} Q_x \\ Q_y \end{bmatrix} = \beta G t([\boldsymbol{\nabla}]w + [\boldsymbol{\theta}]). \tag{4.84}$$

(A introdução da constante β deve-se ao fato das tensões tangenciais não serem constantes ao longo da espessura. Veja, por exemplo, TIMOSHENKO [117].)

As equações que governam o comportamento de placas são completadas com as relações de equilíbrio

$$\begin{bmatrix} \dfrac{\partial}{\partial x} & 0 & \dfrac{\partial}{\partial y} \\ 0 & \dfrac{\partial}{\partial y} & \dfrac{\partial}{\partial x} \end{bmatrix} \begin{bmatrix} M_x \\ M_y \\ M_{xy} \end{bmatrix} + \begin{bmatrix} Q_x \\ Q_y \end{bmatrix} \equiv \mathbf{L}^T\mathbf{M} + \mathbf{Q} = \mathbf{0}, \tag{4.85}$$

e

$$\begin{bmatrix} \dfrac{\partial}{\partial x} & \dfrac{\partial}{\partial y} \end{bmatrix} \begin{bmatrix} Q_x \\ Q_y \end{bmatrix} - g \equiv [\boldsymbol{\nabla}^T]\mathbf{Q} - g = 0. \tag{4.86}$$

Teoria de Kirchhoff Consideraremos, por fim, a seguinte hipótese adicional a respeito do comportamento de placas delgadas na flexão.

PLACAS DELGADAS Seções planas normais ao plano médio da placa, durante o processo de deformação, permanecem planas e normais ao plano médio deformado.

Com essa hipótese, estamos negligenciando as deformações de cisalhamento transversal e adotando $G = \infty$. A Equação (4.84) fica, portanto,

$$[\boldsymbol{\nabla}]w + [\boldsymbol{\theta}] = \mathbf{0}. \tag{4.87}$$

Então, temos que

$$\theta_x = -\frac{\partial w}{\partial x}, \qquad \theta_y = -\frac{\partial w}{\partial y}. \tag{4.88}$$

4.8 Sumário

Nesse Capítulo desenvolvemos os modelos matemáticos que governam o comportamento mecânico dos sólidos considerados no trabalho, com base na teoria da mecânica do contínuo. As equações dos modelos matemáticos foram deduzidas a partir de hipóteses simplificadoras sobre os materiais que constituem nossos objetos estruturais.

Primeiramente consideramos a matéria como sendo contínua, destituída de espaços vazios. Com base nessa hipótese, apresentamos os conceitos de tensão e deformação. Em seguida, simplificamos as expressões dos campos de deformações, supondo "pequenos" deslocamentos e "pequenos" gradientes de deformações. Deduzimos, então, as equações de equilíbrio estático em função do tensor de tensões de Cauchy.

Posteriormente, consideramos somente materiais homogênos, isótropos e elásticos e derivamos as relações constitutivas da elasticidade, a lei de Hooke generalizada. A partir das equações de equilíbrio, das equações deformação-deslocamento e das expressões da lei de Hooke, definimos nosso modelo matemático geral. Compactamente, utilizamos a equação de Navier para sumarizar as equações de campo. Apresentamos, também, uma solução analítica do modelo matemático geral, a solução fundamental de Kelvin.

Finalmente, particularizamos resumidamente as equações de nosso modelo matemático geral para os casos de membranas e placas delgadas elásticas.

CAPÍTULO 5

Análise Numérica de Modelos Estruturais

5.1 Introdução

No Capítulo 4 deduzimos um modelo matemático para sólidos contínuos, homogêneos, isótropos e perfeitamente elásticos que supostamente apresentam pequenos deslocamentos e pequenos gradientes de deformação quando submetidos a forças de volume e de superfície estaticamente aplicadas. Discutimos que as equações diferenciais do modelo matemático só podem ser resolvidas, para os casos mais gerais de geometria, carregamentos e condições de contorno, através do emprego de métodos numéricos aproximados.

Neste Capítulo, abordaremos o método dos elementos finitos e o método dos elementos de contorno, importantes técnicas de análise numérica em engenharia de estruturas. O método dos elementos finitos é baseado na representação do domínio e do contorno de um corpo em subregiões denominadas *elementos finitos*, sobre as quais são escritas aproximações para os campos de deslocamentos em termos de funções interpoladoras e de parâmetros nodais, quase todos incógnitos. O método dos elementos de contorno, por outro lado, é baseado na discretização do contorno de um corpo, somente, 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. Derivaremos as formulações para ambos os processos do método mais geral de *resíduos ponderados*, abordado na Seção 5.2. Nosso objetivo é escrever a forma fraca e a sentença inversa de resíduos ponderados para o problema fundamental. A partir da forma fraca desenvolveremos, na Seção 5.3, a expressão do *princípio dos trabalhos virtuais* (PTV), usualmente tomada como ponto de partida para o modelo de elementos finitos denominado *método dos deslocamentos*. Resumiremos o algoritmo do método dos deslocamentos na Seção 5.4. Da sentença inversa de resíduos ponderados para o problema fundamental escreveremos, na Seção 5.5, as *equações integrais de contorno*, a partir das quais discutiremos os passos básicos do método dos elementos de contorno, Seção 5.6. As implementações (orientadas a objetos) dos métodos serão vistas na segunda parte do texto.

5.2 Resíduos Ponderados

Seja um corpo definido por um domínio Ω (volume, superfície, etc.) delimitado por um contorno Γ de normal $\mathbf{n}$. Consideremos que um determinado problema físico seja governado por um conjunto de equações diferenciais representadas por um operador $\mathcal{L}$. $\mathcal{L}$ é definido como um processo que aplicado a uma função $\mathbf{u}$ produz uma outra função $\mathbf{f}$, tal que

$$\mathcal{L}(\mathbf{u}) = \mathbf{f} \qquad \text{em } \Omega. \tag{5.1}$$

$\mathcal{L}$ pode denotar apenas uma equação diferencial, ou um conjunto de equações simultâneas. A variável $\mathbf{u}$ pode representar uma função escalar ou uma função vetorial no plano ou no espaço. Por exemplo, na equação de Navier em sua forma vetorial,

$$G\,\nabla^2\mathbf{u} + \frac{G}{1-2\nu}\,\nabla(\nabla\cdot\mathbf{u}) + \mathbf{b} = \mathbf{0}, \tag{4.56-repetida}$$

o operador $\mathcal{L}$ é definido como

$$\mathcal{L}(\mathbf{u}) = G\,\nabla^2() + \frac{G}{1-2\nu}\,\nabla\left[\nabla\cdot()\right], \tag{5.2}$$

representando um conjunto de três equações diferenciais parciais de segunda ordem.

A versão homogênea da Equação (5.1) é

$$\mathcal{L}(\mathbf{u}) = \mathbf{0} \qquad \text{em } \Omega. \tag{5.3}$$

Seja agora uma função $\mathbf{w}$. Podemos definir um produto interno tal que

$$\int_\Omega \mathcal{L}(\mathbf{u})\,\mathbf{w}\,d\Omega = 0. \tag{5.4}$$

O produto interno (5.4) pode ser integrado por partes até que todas as derivadas em $\mathbf{u}$ sejam eliminadas. Em geral, podemos escrever [18, página 3]

$$\int_\Omega \mathcal{L}(\mathbf{u})\,\mathbf{w}\,d\Omega = \int_\Omega \mathbf{u}\,\mathcal{L}^*(\mathbf{w})\,d\Omega + \int_\Gamma \left[\mathcal{S}^*(\mathbf{w})\mathcal{G}(\mathbf{u}) - \mathcal{G}^*(\mathbf{w})\mathcal{S}(\mathbf{u})\right]\,d\Gamma. \tag{5.5}$$

A Equação (5.5) é a forma transposta (ou forma inversa) do produto interno. O operador $\mathcal{L}^*$ é chamado de adjunto de $\mathcal{L}$, e se $\mathcal{L}^* = \mathcal{L}$, então $\mathcal{L}$ é dito ser auto-adjunto (neste caso, $\mathcal{S}^* = \mathcal{S}$ e $\mathcal{G}^* = \mathcal{G}$). Se $\mathcal{L}$ é auto-adjunto, a integração por partes produz dois tipos diferentes de condições de contorno, definidas pelos conjuntos

$$\begin{aligned} &\mathcal{S}(\mathbf{u}) \qquad \text{prescrito em } \Gamma_1, \text{ e} \\ &\mathcal{G}(\mathbf{u}) \qquad \text{prescrito em } \Gamma_2, \end{aligned} \tag{5.6}$$

onde $\mathcal{S}$ são as condições de contorno *essenciais* e $\mathcal{G}$ são as condições de contorno *naturais* do problema. Γ_1 e Γ_2, $\Gamma = \Gamma_1 + \Gamma_2$, são porções complementares do contorno. Podemos especificar ambos os tipos de condições sobre o contorno Γ de Ω, mas um certo número de condições essenciais *devem* ser prescritas em alguns pontos para que a solução da

Equação (5.1) seja única. Por exemplo, para o problema elastostático modelado no Capítulo 4, as condições de contorno essenciais e naturais são[1]

$$\begin{aligned} &\mathbf{u} = \overline{\mathbf{u}}, \text{ e} \\ &\mathbf{p} = \overline{\mathbf{p}} = \frac{2G\nu}{1-2\nu}(\nabla \cdot \mathbf{u})\mathbf{n} + G\,(\mathbf{u}\overleftarrow{\nabla} + \overrightarrow{\nabla}\mathbf{u}) \cdot \mathbf{n}, \end{aligned} \tag{5.7}$$

deslocamentos prescritos em Γ_1 e forças de superfície prescritas em Γ_2, respectivamente. Nesse caso, os operadores $\mathcal{S}$ e $\mathcal{G}$ são

$$\begin{aligned} &\mathcal{S}(\mathbf{u}) = (), \text{ e} \\ &\mathcal{G}(\mathbf{u}) = \frac{2G\nu}{1-2\nu}[\nabla \cdot ()]\,\mathbf{n} + G\left[()\overleftarrow{\nabla} + \overrightarrow{\nabla}()\right] \cdot \mathbf{n}. \end{aligned} \tag{5.8}$$

(Lembremos que essa expressão para $\mathcal{G}$ corresponde à equação de Navier para tensões no contorno. Notemos, também, que Γ_1 e Γ_2 nem sempre representam pontos distintos: em um mesmo ponto de contorno podemos prescrever deslocamentos e forças de superfície, desde que em direções diferentes.)

Consideremos, agora, uma função $\mathbf{u}_0$ que seja a solução exata do modelo matemático do problema físico governado pela Equação (5.1). Ou seja, $\mathbf{u}_0$ satisfaz *exatamente* a equação

$$\mathcal{L}(\mathbf{u}_0) = \mathbf{f} \qquad \text{em } \Omega \tag{5.9}$$

e as condições de contorno

$$\begin{aligned} &\mathcal{S}(\mathbf{u}_0) = \mathbf{s} \qquad \text{em } \Gamma_1, \text{ e} \\ &\mathcal{G}(\mathbf{u}_0) = \mathbf{g} \qquad \text{em } \Gamma_2. \end{aligned} \tag{5.10}$$

Geralmente, a expressão analítica de $\mathbf{u}_0$ é muito difícil de ser determinada. Escreveremos, então, uma aproximação para $\mathbf{u}_0$ tal que

$$\mathbf{u}_0 \approx \mathbf{u} = \sum_{k=1}^{n} \boldsymbol{\phi}_k\, \boldsymbol{\alpha}_k, \tag{5.11}$$

onde $\boldsymbol{\alpha}_k$ são parâmetros, quase todos incógnitos, e $\boldsymbol{\phi}_k$ são funcões linearmente independentes tomadas de um conjunto completo de funções (as quais devem satisfazer certas condições de *admissibilidade*)

$$\boldsymbol{\phi}_1(\mathbf{x}),\ \boldsymbol{\phi}_2(\mathbf{x}),\ \ldots,\ \boldsymbol{\phi}_n(\mathbf{x}), \tag{5.12}$$

sendo $\mathbf{x}$ as coordenadas espaciais no domínio Ω. A seqüência de funções (5.12) é linearmente independente se

$$a_1\boldsymbol{\phi}_1 + a_2\boldsymbol{\phi}_2 + \cdots + a_n\boldsymbol{\phi}_n = \mathbf{0} \tag{5.13}$$

é satisfeita somente quando todos a_k são nulos.[2]

[1] O operador $\mathcal{L}$ da equação de Navier é auto-adjunto. O método dos resíduos ponderados, contudo, pode ser aplicado da mesma forma para operadores que não são auto-adjuntos.

[2] Veja BREBBIA [18, página 8] para uma definição de seqüência completa de funções linearmente independentes.

Admitiremos, inicialmente, que a função aproximadora definida pela Equação (5.11) satisfaz exatamente *todas* as condições de contorno (5.10) do problema e possua ordem de continuidade tal que $\mathcal{L}(\mathbf{u}) \neq \mathbf{0}$. A substituição da Equação (5.11) na Equação (5.9) produz uma funcão de erro ou *resíduo* $\mathbf{R}$ sobre o domínio tal que

$$\mathcal{L}(\mathbf{u}) - \mathbf{f} = \mathbf{R} \neq \mathbf{0} \qquad \text{em } \Omega. \tag{5.14}$$

Nosso propósito é tornar o resíduo definido na Equação (5.14) tão "pequeno" quanto possível. Faremos isso distribuindo ponderadamente o resíduo $\mathbf{R}$ sobre o domínio Ω, como a seguir.

Seja uma função $\mathbf{w}$ tal que

$$\mathbf{w} = \boldsymbol{\psi}_1\boldsymbol{\beta}_1 + \boldsymbol{\psi}_2\boldsymbol{\beta}_2 + \boldsymbol{\psi}_3\boldsymbol{\beta}_3 + \cdots, \tag{5.15}$$

onde $\boldsymbol{\beta}_k$ são parâmetros arbitrários e $\boldsymbol{\psi}_k$ são funções linearmente independentes tomadas de um outro conjunto

$$\boldsymbol{\psi}_1,\ \boldsymbol{\psi}_2,\ \ldots,\ \boldsymbol{\psi}_n.$$

O erro $\mathbf{R}$ pode ser *distribuído* em Ω tomando-se o produto interno

$$\int_\Omega \mathbf{R}\,\mathbf{w}\,d\Omega = 0, \tag{5.16}$$

onde a função $\mathbf{w}$ definida pela Equação (5.15) é a *função ponderadora* dessa distribuição. Como os parâmetros $\boldsymbol{\beta}_k$ são arbitrários, temos, da Equação (5.16), que

$$\int_\Omega \mathbf{R}\,\boldsymbol{\psi}_i\,d\Omega = 0 \qquad \text{para } i = 1, 2, \ldots, n. \tag{5.17}$$

Se a função aproximadora $\mathbf{u}$ não satisfaz identicamente as condições de contorno, ou seja

$$\begin{aligned} \mathcal{S}(\mathbf{u}) - \mathbf{s} &= \mathbf{R}_1 \neq \mathbf{0} \qquad \text{em } \Gamma_1, \text{ e} \\ \mathcal{G}(\mathbf{u}) - \mathbf{g} &= \mathbf{R}_2 \neq \mathbf{0} \qquad \text{em } \Gamma_2, \end{aligned} \tag{5.18}$$

então os resíduos $\mathbf{R}_1$ e $\mathbf{R}_2$ sobre o contorno também devem ser distribuídos em Γ_1 e Γ_2 de acordo com a função ponderadora $\mathbf{w}$. Para um operador $\mathcal{L}$ auto-adjunto, a expressão de resíduos ponderados, nesse caso, pode ser escrita como

$$\int_\Omega \mathbf{R}\,\mathbf{w}\,d\Omega = \int_{\Gamma_2} \mathbf{R}_2\,\mathcal{S}^*(\mathbf{w})\,d\Gamma - \int_{\Gamma_1} \mathbf{R}_1\,\mathcal{G}^*(\mathbf{w})\,d\Gamma. \tag{5.19}$$

(A expressão de resíduos ponderados para o problema fundamental será desenvolvida na próxima Seção.) Notemos que as Equações (5.16) e (5.19) são representações integrais que envolvem a solução aproximada das equações diferenciais do problema em questão. Essas *formas integrais* possibilitam que, a partir de uma divisão do domínio Ω e/ou do contorno Γ em *elementos* discretos, a solução aproximada seja obtida a partir das contribuições individuais de todos os elementos. A técnica geral, baseada na Equação (5.16) e em sua extensão (5.19), é denominada *método dos resíduos ponderados*. Dependendo da escolha da função ponderadora, a técnica resulta em diferentes métodos numéricos. Vejamos alguns exemplos.

Método da colocação de pontos Neste método, a função de erro, Equação (5.14), é satisfeita apenas em alguns pontos escolhidos sobre o domínio Ω. Essa condição pode ser expressa na Equação (5.16) de resíduos ponderados tomando-se como função ponderadora

$$\mathbf{w} = \delta(\mathbf{x}_1, \mathbf{x})\boldsymbol{\beta}_1 + \delta(\mathbf{x}_2, \mathbf{x})\boldsymbol{\beta}_2 + \cdots + \delta(\mathbf{x}_n, \mathbf{x})\boldsymbol{\beta}_n = \sum_{i=1}^{n} \delta(\mathbf{x}_i, \mathbf{x})\boldsymbol{\beta}_i, \tag{5.20}$$

onde n é o número de pontos escolhidos, os *pontos de colocação* (usualmente distribuídos de maneira uniforme em Ω), e $\delta(\mathbf{x}_i, \mathbf{x})$ é a função delta de Dirac, definida na Seção 4.6. Em princípio, o número de pontos de colocação é igual ao número de incógnitas $\boldsymbol{\alpha}_i$ do problema, Equação (5.11).

Método da colocação de subregiões Neste método, a expressão de resíduos ponderados, Equação (5.16) é satisfeita sobre diferentes regiões Ω_i do domínio, ou seja,

$$\int_{\Omega_i} \mathbf{R}\, d\Omega = 0 \qquad \text{em } \Omega_i. \tag{5.21}$$

A função ponderadora, nesse caso, é $\mathbf{w} = \mathbf{1}$, a função identidade.

Método de Galerkin No método de Galerkin, as funções ponderadoras pertencem ao mesmo conjunto das funções aproximadoras, ou seja,

$$\mathbf{w} = \boldsymbol{\phi}_1\boldsymbol{\beta}_1 + \boldsymbol{\phi}_2\boldsymbol{\beta}_2 + \cdots + \boldsymbol{\phi}_n\boldsymbol{\beta}_n, \tag{5.22}$$

onde $\boldsymbol{\phi}_i$ é uma função linearmente independente tomada do conjunto completo (5.12). Como as mesmas funções são utilizadas para $\mathbf{u}$ e $\mathbf{w}$ e os parâmetros $\boldsymbol{\beta}_i$ são arbitrários, podemos escrever $\mathbf{w}$ como uma *variação* de $\mathbf{u}$, isto é,

$$\mathbf{w} = \delta\mathbf{u} = \boldsymbol{\phi}_1\,\delta\boldsymbol{\alpha}_1 + \boldsymbol{\phi}_2\,\delta\boldsymbol{\alpha}_2 + \boldsymbol{\phi}_3\,\delta\boldsymbol{\alpha}_3 + \cdots, \tag{5.23}$$

onde $\delta\boldsymbol{\alpha}_i \equiv \boldsymbol{\beta}_i$. Para um operador $\mathcal{L}$ linear, a Equação (5.16) produz um sistema de equações algébricas a partir do qual os parâmetros $\boldsymbol{\alpha}_i$ podem ser determinados, como veremos posteriormente. Na prática, como decorrência das funções aproximadora e ponderadora serem as mesmas, o sistema algébrico derivado da Equação (5.16) possui, geralmente (mas nem sempre), coeficientes simétricos.

5.2.1 Resíduos Ponderados para o Problema Fundamental

Nesta Seção escreveremos as expressões de resíduos ponderados para o problema elastostático modelado no Capítulo 4. Utilizaremos, ao invés da equação de Navier, a equação de equilíbrio estático em sua forma indicial,

$$\sigma_{jk,j} + b_k = 0 \qquad \text{em } \Omega, \tag{5.24}$$

juntamente com as condições de contorno

$$u_k = \overline{u}_k \qquad \text{em } \Gamma_1, \text{ e} \tag{5.25}$$

$$p_k = \overline{p}_k \qquad \text{em } \Gamma_2. \tag{5.26}$$

Sentença original Seja o campo de deslocamentos $\mathbf{u}_0$ a solução exata do problema, a qual será aproximada no domínio Ω por uma função $\mathbf{u}$ (ou, na forma indicial, u_k). Ainda estamos admitindo que a função aproximadora do campo de deslocamentos $\mathbf{u}$ satisfaz as condições de contorno essenciais e naturais do problema, mas não satisfaz a equação de equilíbrio (5.24). Temos, então, um resíduo

$$\sigma_{jk,j} + b_k \neq 0 \qquad \text{em } \Omega, \tag{5.27}$$

o qual pode ser distribuído no domínio fazendo-se

$$\int_\Omega (\sigma_{jk,j} + b_k)\, u_k^*\, d\Omega = 0, \tag{5.28}$$

onde a função ponderadora $\mathbf{w} = \mathbf{u}^*$ representa um campo de deslocamentos qualquer sobre um domínio Ω^* qualquer. Assumiremos que os gradientes do campo de deslocamentos ponderador $\mathbf{u}^*$ sejam pequenos em relação à unidade (ou seja, as relações deslocamento-deformação (4.70) são válidas para $\mathbf{u}^*$) e que as equações constitutivas elásticas (4.74) sejam satisfeitas em Ω^*.

A Equação (5.28) pode ser generalizada para o caso da função aproximadora u_k não satisfazer as condições de contorno (5.25) e (5.26). Nesse caso,

$$\begin{aligned} u_k - \overline{u}_k &\neq 0 \qquad \text{em } \Gamma_1, \text{ e} \\ p_k - \overline{p}_k &\neq 0 \qquad \text{em } \Gamma_2 \end{aligned} \tag{5.29}$$

são os erros, ou resíduos, cometidos ao aproximarmos as condições de contorno essenciais e naturais, respectivamente. Para obtermos uma expressão que relacione os erros de domínio e de contorno, vamos integrar por partes[3] a Equação (5.28),

$$-\int_\Omega \sigma_{jk}\, \epsilon_{jk}^*\, d\Omega + \int_\Omega b_k\, u_k^*\, d\Omega = -\int_\Gamma p_k\, u_k^*\, d\Gamma, \tag{5.30}$$

onde ϵ_{jk}^* é o campo de deformações associado à função ponderadora u_k^*. Devido à simetria do tensor de módulos elásticos C_{jkli}, o primeiro termo da Equação (5.30) pode ser escrito como (veja, por exemplo, BREBBIA [18, página 184])

$$\int_\Omega \sigma_{jk}\, \epsilon_{jk}^*\, d\Omega = \int_\Omega \sigma_{jk}^*\, \epsilon_{jk}\, d\Omega. \tag{5.31}$$

Integrando-se por partes o termo de domínio da Equação (5.30) e levando-se em consideração a reciprocidade (5.31), obtemos a forma transposta da Equação (5.28),[4]

$$\int_\Omega \sigma_{jk,j}^*\, u_k\, d\Omega + \int_\Omega b_k\, u_k^*\, d\Omega = -\int_\Gamma p_k\, u_k^*\, d\Gamma + \int_\Gamma u_k\, p_k^*\, d\Gamma. \tag{5.32}$$

Agora podemos substituir as condições de contorno (5.25) e (5.26) na Equação (5.32),

$$\int_\Omega \sigma_{jk,j}^*\, u_k\, d\Omega + \int_\Omega b_k\, u_k^*\, d\Omega \;=\; -\int_{\Gamma_1} p_k\, u_k^*\, d\Gamma - \int_{\Gamma_2} \overline{p}_k\, u_k^*\, d\Gamma + \int_{\Gamma_1} \overline{u}_k\, p_k^*\, d\Gamma + \int_{\Gamma_2} u_k\, p_k^*\, d\Gamma. \tag{5.33}$$

[3] Podemos utilizar, por exemplo, o teorema da divergência.

[4] De acordo com KANE [57, página 142], pode-se mostrar que a identidade (5.31) é verdadeira também para outros materiais e não somente para os elástico-lineares isotrópicos.

Ao substituirmos as condições de contorno na Equação (5.32), estamos fazendo aproximações que podem ser melhor compreendidas se recuperarmos a Equação (5.28) a partir da Equação (5.33). Integrando-se por partes duas vezes o primeiro termo da Equação (5.33), obtemos

$$\int_{\Omega} (\sigma_{jk,j} + b_k)\, u_k^*\, d\Omega = \int_{\Gamma_2} (p_k - \overline{p}_k)\, u_k^*\, d\Gamma - \int_{\Gamma_1} (u_k - \overline{u}_k)\, p_k^*\, d\Gamma. \tag{5.34}$$

A Equação (5.34) é a *sentença original* de resíduos ponderados para o problema elastostático, supondo que a função $\mathbf{u}$ aproxima a equação diferencial em Ω e as condições de contorno em Γ.

Forma fraca Na expressão de resíduos ponderados (5.19), quais são as restrições impostas à função aproximadora $\mathbf{u}$ tal que o valor da integral seja finito? A resposta, para um problema genérico, depende da ordem de diferenciação dos operadores $\mathcal{L}$, $\mathcal{S}$ e $\mathcal{G}$. Se derivadas de ordem n ocorrem em $\mathcal{L}(\mathbf{u})$, por exemplo, então a função $\mathbf{u}$ deve ser tal que suas $n-1$ derivadas sejam contínuas em Ω, ou seja, $\mathbf{u}$ deve ter continuidade[5] C^{n-1}. Para o problema fundamental, de fato, admitimos até esse ponto que a função aproximadora $\mathbf{u}$ possui ordem de continuidade tal que $\sigma_{jk,j} \neq 0$ em Ω (desde que $b_k \neq 0$). Em muitos casos é possível, e vantajoso, reduzir a ordem de continuidade requerida para a função $\mathbf{u}$ através de uma integração por partes da Equação (5.34),

$$\int_{\Omega} \sigma_{jk}\, \epsilon_{jk}^*\, d\Omega - \int_{\Omega} b_k\, u_k^*\, d\Omega = \int_{\Gamma_2} \overline{p}_k\, u_k^*\, d\Gamma + \int_{\Gamma_1} p_k\, u_k^*\, d\Gamma + \int_{\Gamma_1} (u_k - \overline{u}_k)\, p_k^*\, d\Gamma. \tag{5.35}$$

Na Equação (5.35), uma ordem menor de continuidade para $\mathbf{u}$ é requerida, ao custo de uma ordem maior para a função ponderadora $\mathbf{w} = \mathbf{u}^*$. A Equação (5.35) é a *forma fraca* de resíduos ponderados para o problema elastostático, supondo que a função $\mathbf{u}$ aproxima a equação diferencial em Ω e as condições de contorno em Γ. Derivaremos a formulação do método dos elementos finitos a partir dessa equação.

Sentença inversa Integrando-se por partes o primeiro termo da Equação (5.35), obtemos

$$\int_{\Omega} \sigma_{jk,j}^*\, u_k\, d\Omega + \int_{\Omega} b_k\, u_k^*\, d\Omega = -\int_{\Gamma_2} \overline{p}_k\, u_k^*\, d\Gamma - \int_{\Gamma_1} p_k\, u_k^*\, d\Gamma + \int_{\Gamma_2} u_k\, p_k^*\, d\Gamma + \int_{\Gamma_1} \overline{u}_k\, p_k^*\, d\Gamma. \tag{5.36}$$

A Equação (5.36) é a *sentença inversa* de resíduos ponderados para o problema elastostático, supondo que a função $\mathbf{u}$ aproxima a equação diferencial em Ω e as condições de contorno em Γ. Derivaremos a formulação do método dos elementos de contorno a partir dessa equação.

[5] Veja BREBBIA [18, página 25] e ZIENKIEWICZ [132, página 211] para uma discussão sobre a continuidade das funções aproximadoras.

5.3 Princípio dos Trabalhos Virtuais

Seja a forma fraca do problema elastostático, Equação (5.35). Se admitirmos que a função aproximadora u_k satisfaz as condições de contorno essenciais (5.25), isto é, se $u_k \equiv \overline{u}_k$ em Γ_1, podemos escrever

$$\int_\Omega \sigma_{jk}\,\epsilon^*_{jk}\,d\Omega - \int_\Omega b_k\,u^*_k\,d\Omega = \int_{\Gamma_2} \overline{p}_k\,u^*_k\,d\Gamma + \int_{\Gamma_1} p_k\,u^*_k\,d\Gamma. \tag{5.37}$$

Consideremos, agora, uma função ponderadora $u^*_k = \delta u_k$, onde a variação δu_k é um campo de deslocamentos infinitesimais definido a partir do mesmo conjunto de funções linearmente independentes utilizado na definição de u_k (método de Galarkin). Vamos supor que esse campo de deslocamentos satisfaça a versão homogênea das condições de contorno essenciais, ou seja, $u^*_k = \delta u_k \equiv 0$ em Γ_1. Então, a Equação (5.37) se reduz a

$$\int_\Omega \sigma_{jk}\,\delta\epsilon_{jk}\,d\Omega = \int_\Omega b_k\,\delta u_k\,d\Omega + \int_{\Gamma_2} \overline{p}_k\,\delta u_k\,d\Gamma. \tag{5.38}$$

A Equação (5.38) é a expressão do *princípio dos trabalhos virtuais* (PTV). Se considerarmos que os deslocamentos $\delta\mathbf{u}$ são *deslocamentos virtuais*[6] aplicados a uma configuração de equilíbrio de um corpo, então podemos interpretar o lado direito da Equação (5.38) como o trabalho externo δW_{ext} realizado pelas forças de volume e de superfície atuando no domínio e no contorno do corpo durante os deslocamentos imaginários $\delta\mathbf{u}$. O lado esquerdo da equação pode ser interpretado como a energia de deformação δU armazenada no corpo durante os deslocamentos imaginários $\delta\mathbf{u}$. Na configuração de equilíbrio, $\delta U = \delta W_{\text{ext}}$. A função ponderadora $\delta\mathbf{u}$ deve satisfazer a versão homogênea das condições de contorno essenciais porque os deslocamentos virtuais devem ser *cinematicamente admissíveis*, ou seja, devem satisfazer quaisquer deslocamentos prescritos. (Desde que os deslocamentos virtuais são deslocamentos *adicionais* impostos a uma configuração de equilíbrio de um corpo, um componente de deslocamento virtual deve ser nulo sempre que o deslocamento atual seja prescrito por uma condição de contorno.)

Princípio dos Trabalhos Virtuais Se $\delta U = \delta W_{\text{ext}}$ para qualquer campo de deslocamentos virtuais cinematicamente admissíveis, então o estado de tensões satisfaz as condições de equilíbrio do corpo.

Embora a expressão (5.38) possa ser derivada da aplicação de métodos variacionais em mecânica dos sólidos, foi escrita, aqui, simplesmente como uma forma fraca da equação de equilíbrio (5.24), supondo uma função aproximadora satisfazendo as condições de contorno essenciais e uma função ponderadora de Galerkin satisfazendo a versão homogênea das condições essenciais.

[6] Note que, ao afirmarmos que os deslocamentos virtuais $\delta\mathbf{u}$ são infinitesimais, não estamos impondo restrição alguma sobre a magnitude dos deslocamentos atuais $\mathbf{u}$. Portanto, o princípio dos trabalhos virtuais pode ser aplicado em problemas onde os deslocamentos, rotações e deformações não sejam "pequenos".

5.4 Método dos Elementos Finitos

Seja um sólido contínuo, homogêneo, isotrópico e elástico, definido por um volume Ω e uma superfície Γ. Vamos supor que, devido à ação de forças externas aplicadas no volume Ω e na superfície Γ, o sólido assuma uma configuração de equilíbrio cinematicamente caracterizada por campos de pequenos deslocamentos e pequenas deformações.

O método dos elementos finitos é baseado na subdivisão do domínio Ω em um conjunto de células, os *elementos finitos*, os quais são conectados em pontos discretos chamados *nós*. As coordenadas de um nó são tomadas em relação a um sistema de cooordenadas Cartesianas que denominaremos de *sistema global de coordenadas*. Um elemento finito é uma subregião do domínio definida geometricamente por uma seqüência ordenada dos nós nos quais o elemento incide, chamada *lista de incidência* do elemento. Note que o domínio Ω pode ser *geometricamente* representado por um modelo de decomposição por células, tal como visto na Seção 3.6. Os deslocamentos, deformações e tensões em qualquer ponto do domínio podem ser determinados, por interpolação, a partir dos deslocamentos, deformações e tensões nos pontos discretos do modelo.

Utilizaremos o princípio dos trabalhos virtuais, Equação (5.38), para expressar matematicamente o equilíbrio do sólido. Será mais conveniente escrevermos o PTV na forma matricial

$$\int_{\Omega} [\delta\boldsymbol{\epsilon}]^T [\boldsymbol{\sigma}]\, d\Omega = \int_{\Omega} [\delta\mathbf{u}]^T \mathbf{b}\, d\Omega + \int_{\Gamma_2} [\delta\mathbf{u}]^T \mathbf{p}\, d\Gamma, \tag{5.39}$$

sendo $[\boldsymbol{\sigma}]$ e $[\boldsymbol{\epsilon}]$, respectivamente, vetores de componentes de tensão e de deformação dados, no caso tridimensional, pelas expressões (4.47).

Com a subdivisão do domínio do sólido em NE elementos finitos, podemos escrever a Equação (5.39) como somas de integrais sobre o volume e a superfície de cada elemento finito do modelo:

$$\sum_{e=1}^{\mathrm{NE}} \left\{ \int_{\Omega^{(e)}} \left[\delta\boldsymbol{\epsilon}^{(e)}\right]^T \left[\boldsymbol{\sigma}^{(e)}\right] d\Omega - \int_{\Omega^{(e)}} \left[\delta\mathbf{u}^{(e)}\right]^T \mathbf{b}^{(e)} d\Omega - \int_{\Gamma_2^{(e)}} \left[\delta\mathbf{u}^{(e)}\right]^T \mathbf{p}^{(e)} d\Gamma \right\} = 0, \tag{5.40}$$

onde, para um elemento finito e, $\mathbf{u}^{(e)}$ são os componentes Cartesianos dos deslocamentos em um ponto qualquer q de e, $[\boldsymbol{\sigma}^{(e)}]$ são os componentes de tensão em q, $\mathbf{b}^{(e)}$ e $\mathbf{p}^{(e)}$ são, respectivamente, as forças de volume e de superfície em q e $[\delta\boldsymbol{\epsilon}^{(e)}]$ são os componentes Cartesianos das deformações "virtuais" em q.

Vamos escrever, agora, uma função aproximadora para o campo de deslocamentos $\mathbf{u}^{(e)}$ de cada elemento finito do modelo. Consideremos, então, um elemento finito e definido por n nós. Seja q um ponto qualquer de e, cujas coordenadas $\boldsymbol{\xi}$ são tomadas em relação a um *sistema de coordenadas normalizadas* definido no Capítulo 3. Usando a Equação (5.11), o vetor de deslocamento $\mathbf{u}^{(e)}$ de q pode ser aproximado por

$$\mathbf{u}^{(e)} = \sum_{i=1}^{n} \boldsymbol{\phi}_i^{(e)}\, \boldsymbol{\alpha}_i^{(e)}, \tag{5.41}$$

onde $\boldsymbol{\phi}_i^{(e)}$ são funções da posição $\boldsymbol{\xi}$ (linearmente independentes tomadas de um conjunto completo, etc.) e $\boldsymbol{\alpha}_i^{(e)}$ são *parâmetros generalizados*, quase todos incógnitos, associados

ao elemento e. Matricialmente, a Equação (5.41) fica

$$\mathbf{u}^{(e)} = \left[\boldsymbol{\phi}_1^{(e)} \quad \boldsymbol{\phi}_2^{(e)} \quad \cdots \quad \boldsymbol{\phi}_n^{(e)}\right] \begin{bmatrix} \boldsymbol{\alpha}_1^{(e)} \\ \boldsymbol{\alpha}_2^{(e)} \\ \cdots \\ \boldsymbol{\alpha}_n^{(e)} \end{bmatrix} = \mathbf{A}^{(e)} \left[\boldsymbol{\alpha}^{(e)}\right]. \tag{5.42}$$

Podemos escrever a Equação (5.50) para os deslocamentos de cada um dos n nós do elemento e organizar na forma matricial

$$\mathbf{U}_{(e)} = \begin{bmatrix} \mathbf{u}_1^{(e)} \\ \mathbf{u}_2^{(e)} \\ \cdots \\ \mathbf{u}_n^{(e)} \end{bmatrix} = \begin{bmatrix} \mathbf{A}_1^{(e)} \\ \mathbf{A}_2^{(e)} \\ \cdots \\ \mathbf{A}_n^{(e)} \end{bmatrix} \left[\boldsymbol{\alpha}^{(e)}\right] = \left[\boldsymbol{\Phi}^{(e)}\right] \left[\boldsymbol{\alpha}^{(e)}\right], \tag{5.43}$$

onde $\mathbf{u}_i^{(e)}$ é o vetor de deslocamento do nó i e $\mathbf{U}_{(e)}$ é o vetor dos deslocamentos de todos os nós do elemento e. Se escolhermos as funções $\boldsymbol{\phi}_i^{(e)}$ de tal forma que a matriz $\left[\boldsymbol{\Phi}^{(e)}\right]$ não seja singular, obteremos, da Equação (5.43),

$$\left[\boldsymbol{\alpha}^{(e)}\right] = \left[\boldsymbol{\Phi}^{(e)}\right]^{-1} \mathbf{U}^{(e)}. \tag{5.44}$$

Substituindo a Equação (5.44) na (5.42), temos que

$$\mathbf{u}^{(e)} = \mathbf{A}\left[\boldsymbol{\Phi}^{(e)}\right]^{-1} \mathbf{U}^{(e)} = \mathbf{N}^{(e)}\,\mathbf{U}^{(e)} = \left[\mathbf{N}_1^{(e)} \quad \mathbf{N}_2^{(e)} \quad \cdots \quad \mathbf{N}_n^{(e)}\right] \begin{bmatrix} \mathbf{u}_1^{(e)} \\ \mathbf{u}_2^{(e)} \\ \cdots \\ \mathbf{u}_n^{(e)} \end{bmatrix}, \tag{5.45}$$

ou

$$\mathbf{u}^{(e)} = \sum_{i=1}^{n} \mathbf{N}_i^{(e)} \mathbf{u}_i^{(e)}, \tag{5.46}$$

onde $\mathbf{N}_i^{(e)}$ é uma função da posição $\boldsymbol{\xi}$ associada ao nó i do elemento e a matriz $\mathbf{N}^{(e)}$ é a matriz de funcões de interpolação do elemento. A Equação (5.46) nos mostra que podemos definir uma função aproximadora para o deslocamento $\mathbf{u}^{(e)}$ de um ponto qualquer de um elemento finito e em termos dos deslocamentos nodais $\mathbf{U}^{(e)}$ de e, quase todos incógnitos, os quais são, sem dúvida, fisicamente mais significativos que os parâmetros generalizados $\boldsymbol{\alpha}^{(e)}$. As funções de interpolação devem ser tomadas tais que a Equação (5.46) seja satisfeita para os pontos nodais do elemento. Se todos os componentes do vetor de deslocamento são interpolados identicamente, podemos escrever [132]

$$\mathbf{N}_i^{(e)} = N_i^{(e)} \mathbf{I}, \tag{5.47}$$

onde $N_i^{(e)}$ é uma função escalar da posição $\boldsymbol{\xi}$, associada ao nó i do elemento e. Nesse caso, a Equação (5.46) fica

$$\mathbf{u}^{(e)} = \sum_{i=1}^{n} N_i^{(e)} \mathbf{u}_i^{(e)}. \tag{5.48}$$

Notemos que essa equação é análoga à Equação (3.17) de interpolação das coordenadas de um ponto qualquer de uma célula, tal como definida no Capítulo 3. Em geral, podemos utilizar expressões do tipo da Equação (5.48) para interpolar as grandezas mecânicas e geométricas sobre um elemento finito (ou de contorno),[7] conhecidos os valores nodais. Se adotarmos as mesmas funcões interpoladoras nas Equações (3.17) e (5.48), ou seja, se utilizarmos as mesmas funções para interpolar grandezas geométricas e mecânicas em um elemento, diremos que o elemento é *isoparamétrico*, e chamaremos genericamente as funções de interpolação de *funções de forma* do elemento.

A partir da Equação (5.46), temos que o vetor de componentes de pequenas deformações em um ponto qualquer q de um elemento finito e é dado por

$$\left[\boldsymbol{\epsilon}^{(e)}\right] = \mathbf{B}^{(e)}\,\mathbf{U}^{(e)} = \sum_{i=1}^{n} \mathbf{B}_i^{(e)} \mathbf{u}_i^{(e)}, \tag{5.49}$$

onde $\mathbf{B}^{(e)}$ é a *matriz de deformação-deslocamento* de e e $\mathbf{B}_i^{(e)}$ é a matriz deformação-deslocamento associada ao nó i do elemento e, definida, no caso geral tridimensional, como

$$\mathbf{B}_i^{(e)} = \begin{bmatrix} \dfrac{\partial N_i^{(e)}}{\partial x} & 0 & 0 \\ 0 & \dfrac{\partial N_i^{(e)}}{\partial y} & 0 \\ 0 & 0 & \dfrac{\partial N_i^{(e)}}{\partial z} \\ \dfrac{\partial N_i^{(e)}}{\partial y} & \dfrac{\partial N_i^{(e)}}{\partial x} & 0 \\ 0 & \dfrac{\partial N_i^{(e)}}{\partial z} & \dfrac{\partial N_i^{(e)}}{\partial y} \\ \dfrac{\partial N_i^{(e)}}{\partial z} & 0 & \dfrac{\partial N_i^{(e)}}{\partial x} \end{bmatrix}. \tag{5.50}$$

O vetor de componentes de tensão em um ponto qualquer de um elemento finito e, considerando as relações constitutivas elásticas e a Equação (5.42), é

$$\left[\boldsymbol{\sigma}^{(e)}\right] = \mathbf{C}^{(e)}\,\mathbf{B}^{(e)} = \mathbf{C}^{(e)} \left(\sum_{j=1}^{n} \mathbf{B}_j^{(e)} \mathbf{u}_j^{(e)} \right), \tag{5.51}$$

onde $\mathbf{C}^{(e)}$, a *matriz constitutiva* do elemento e, para o caso tridimensional, é dada pela Equação (4.49).

As versões "virtuais" do deslocamento e das deformações no ponto q do elemento e podem agora ser definidas como

$$\delta\mathbf{u}^{(e)} = \mathbf{N}^{(e)}\,\mathbf{U}(e) = \sum_{i=1}^{n} \mathbf{N}_i^{(e)}\,\delta\mathbf{u}_i \quad \text{e} \quad \left[\delta\boldsymbol{\epsilon}^{(e)}\right] = \mathbf{B}^{(e)}\,\delta\mathbf{U}^{(e)} = \sum_{i=1}^{n} \mathbf{B}_i^{(e)}\,\delta\mathbf{u}_i^{(e)}, \tag{5.52}$$

[7] Genericamente, sobre uma célula de um modelo de decomposição por células.

onde $\delta\mathbf{U}^{(e)}$ é o vetor de deslocamentos nodais de todos os nós de e. Substituindo as Equações (5.52) e a Equação (5.51) na expressão (5.39), obtemos, para o elemento e

$$\left[\delta\mathbf{U}^{(e)}\right]^T \left\{ \int_{\Omega^{(e)}} \left[\mathbf{B}^{(e)}\right]^T \mathbf{C}^{(e)}\mathbf{B}^{(e)}\mathbf{U}^{(e)} d\Omega - \int_{\Omega^{(e)}} \left[\mathbf{N}^{(e)}\right]^T \mathbf{b}^{(e)} d\Omega - \int_{\Gamma_2^{(e)}} \left[\mathbf{N}^{(e)}\right]^T \mathbf{p}^{(e)} d\Gamma \right\} = 0. \tag{5.53}$$

Como os deslocamentos virtuais $\delta\mathbf{U}^{(e)}$ são arbitrários, temos que

$$\int_{\Omega^{(e)}} \left[\mathbf{B}^{(e)}\right]^T \mathbf{C}^{(e)}\mathbf{B}^{(e)}\mathbf{U}^{(e)} d\Omega - \int_{\Omega^{(e)}} \left[\mathbf{N}^{(e)}\right]^T \mathbf{b}^{(e)} d\Omega - \int_{\Gamma_2^{(e)}} \left[\mathbf{N}^{(e)}\right]^T \mathbf{p}^{(e)} d\Gamma = 0. \tag{5.54}$$

A Equação (5.54) pode ser colocada na forma

$$\mathbf{K}^{(e)}\mathbf{U}^{(e)} = \mathbf{F}^{(e)}, \tag{5.55}$$

onde $\mathbf{K}^{(e)}$ é a *matriz de rigidez* do elemento e e $\mathbf{F}^{(e)}$ é o *vetor de carregamentos nodais equivalentes* do elemento e, definidos, respectivamente, como

$$\mathbf{K}^{(e)} = \int_{\Omega^{(e)}} \left[\mathbf{B}^{(e)}\right]^T \mathbf{C}^{(e)}\mathbf{B}^{(e)} d\Omega, \tag{5.56}$$

$$\mathbf{F}^{(e)} = \int_{\Omega^{(e)}} \left[\mathbf{N}^{(e)}\right]^T \mathbf{b}^{(e)} d\Omega + \int_{\Gamma_2^{(e)}} \left[\mathbf{N}^{(e)}\right]^T \mathbf{p}^{(e)} d\Gamma. \tag{5.57}$$

A partir das relações anteriores a Equação (5.40) pode ser escrita como

$$\sum_{e=1}^{\mathrm{NE}} \mathbf{K}^{(e)}\,\mathbf{U}^{(e)} = \sum_{e=1}^{\mathrm{NE}} \mathbf{F}^{(e)}, \tag{5.58}$$

ou

$$\mathbf{K}\,\mathbf{U} = \mathbf{F}, \tag{5.59}$$

onde $\mathbf{K}$ é a *matriz de rigidez global* do modelo, $\mathbf{F}$ é o *vetor de carregamentos nodais equivalentes* do modelo e $\mathbf{U}$ é o vetor de incógnitas nodais de todos os nós do elemento. Os somatórios da Equação (5.58) são efetuados através da adição das contribuições de cada elemento de tal forma que $\mathbf{K}$, $\mathbf{U}$ e $\mathbf{F}$ sejam organizados em partições nodais.

A solução do sistema linear (5.59), após a introdução das condições de contorno, fornece os deslocamentos incógnitos do problema, a partir dos quais podemos determinar as deformações e tensões em todos os nós do modelo e, por interpolação, em qualquer ponto do domínio. Devido ao fato dos deslocamentos nodais serem incógnitos, a formulação do método dos elementos finitos discutida aqui é chamada de *método dos deslocamentos*. A seguir, resumiremos brevemente o esquema computacional do método dos deslocamentos. Maiores detalhes são facilmente encontrados na literatura.

5.4.1 Esquema Computacional

Passo 1 **Discretização do domínio**. O domínio Ω é subdividido em NE elementos finitos sobre os quais são aproximados, em termos de funções de interpolação e de parâmetros nodais, o campo de deslocamentos. Essa etapa é chamada *pré-processamento*.

Passo 2 **Computação das contribuições dos elementos**. A matriz de rigidez $\mathbf{K}^{(e)}$ e os carregamentos nodais $\mathbf{F}^{(e)}$ são determinados para cada elemento finito e.

Passo 3 **Montagem do sistema linear**. O lado esquerdo $\mathbf{K}$ e o lado direito $\mathbf{F}$ do sistema (5.59) são montados a partir das contribuições de todos os NE elementos finitos do modelo.

Passo 4 **Introdução das condições de contorno**. A matriz $\mathbf{K}$, originalmente singular, é transformada em uma matriz regular com a consideração de um número apropriado de condições de contorno essenciais que definem a *vinculação* do sólido.

Passo 5 **Solução do sistema linear**. A solução do sistema fornece os deslocamentos incógnitos nos pontos e direções onde esses valores não são prescritos.

Passo 6 **Computação de valores no domínio**. Conhecidos os deslocamentos nodais $\mathbf{U}$ do sólido, podemos determinar os deslocamentos, deformações e tensões em qualquer ponto do domínio. Essa etapa é chamada de *pós-procesamento*.

5.5 Equações Integrais de Contorno

Consideremos, agora, a forma inversa do problema elastostático, Equação (5.36), e uma função ponderadora $\mathbf{u}^*$ que satisfaz as equações diferenciais do modelo matemático em Ω^*. Na Seção 4.6 vimos uma função desse tipo, a solução fundamental de Kelvin, a qual satisfaz as condições de equilíbrio

$$\sigma^*_{ijk,j}(p,q) + \delta(p,q)\,\delta_{ik} = 0 \tag{5.60}$$

em um sólido elástico infinito, onde $\delta(p,q)$ é a função delta de Dirac, δ_{ik} é o delta de Kronecker e

$$b^*_{ik}(p,q) = \delta(p,q)\,\delta_{ik} \tag{5.61}$$

representa uma força concentrada unitária aplicada em um ponto fonte p de Ω^*, na direção i. Vamos supor que $\Omega + \Gamma$ seja uma subregião finita do meio infinito Ω^*, como mostrado na Figura 5.1, e que $p \in \Omega$.

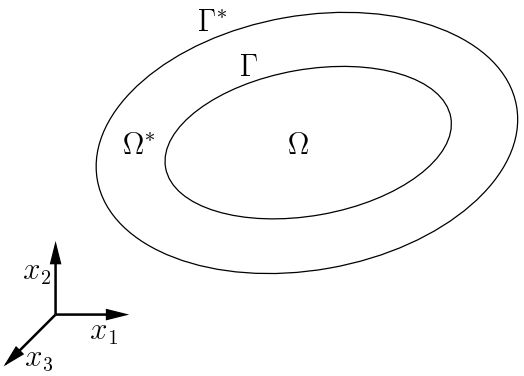


Figura 5.1: Domínio infinito Ω^* contendo $\Omega + \Gamma$.

Identidade Somigliana

Escrevendo a forma inversa para cada direção i (com $u_k^* = u_{ik}^*$ e $p_k^* = p_{ik}^*$), temos

$$\int_\Omega \sigma_{ijk,j}^*(p,q)\, u_k(q)\, d\Omega + \int_\Omega u_{ik}^*(p,q)\, b_k(q)\, d\Omega = - \int_\Gamma u_{ik}^*(p,q)\, p_k(q)\, d\Gamma + \int_\Gamma p_{ik}^*(p,q)\, u_k(q)\, d\Gamma, \quad (5.62)$$

onde, para abreviar a notação, nenhuma distinção foi feita entre u_k e $\overline{u}_k$ e p_k e $\overline{p}_k$. Usando a Equação (5.60) na primeira integral da Equação (5.62) obtemos

$$\int_\Omega \delta(p,q)\, \delta_{ik}\, u_k(q)\, d\Omega = \int_\Gamma u_{ik}^*(p,q)\, p_k(q)\, d\Gamma - \int_\Gamma p_{ik}^*(p,q)\, u_k(q)\, d\Gamma + \int_\Omega u_{ik}^*(p,q)\, b_k(q)\, d\Omega. \quad (5.63)$$

Aplicando a propriedade de seleção da função delta de Dirac, Equação (4.62), generalizada para o caso de um domínio qualquer Ω,

$$\int_\Omega g(q)\, \delta(p,q)\, d\Omega = \begin{cases} g(p) & \text{se } p \in \Omega, \\ 0 & \text{se } p \notin \{\Omega + \Gamma\}, \end{cases} \quad (5.64)$$

o primeiro termo da Equação (5.63), para pontos localizados no domínio, pode ser escrito como

$$\int_\Omega \delta(p,q)\, \delta_{ik}\, u_k(q)\, d\Omega = u_i(p). \quad (5.65)$$

Logo, a Equação (5.63) fica

$$u_i(p) = \int_\Gamma u_{ik}^*(p,q)\, p_k(q)\, d\Gamma - \int_\Gamma p_{ik}^*(p,q)\, u_k(q)\, d\Gamma + \int_\Omega u_{ik}^*(p,q)\, b_k(q)\, d\Omega. \quad (5.66)$$

A Equação (5.66), conhecida como *identidade Somigliana*, 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 em Ω (sempre prescritas). Originalmente, a identidade Somigliana é derivada da relação de reciprocidade (5.31),

$$\int_{\Omega} \sigma_{jk}(q)\,\epsilon^{*}_{ijk}(p,q)\,d\Omega = \int_{\Omega} \sigma^{*}_{ijk}(p,q)\,\epsilon_{jk}(q)\,d\Omega, \tag{5.67}$$

considerando os estados de equilíbrio em Ω e Ω^{*}, definidos, respectivamente, por u_k, p_k, σ_{jk}, ϵ_{jk} e por u^{*}_{ik}, p^{*}_{ik}, σ^{*}_{ijk}, ϵ^{*}_{ijk}. Integrando por partes os dois lados da Equação (5.67), obtemos a equação do *teorema de reciprocidade de Betti*

$$\int_{\Omega} b^{*}_{ik}(p,q)\,u_k(q)\,d\Omega + \int_{\Gamma} p^{*}_{ik}(p,q)\,u_k(q)\,d\Gamma = \int_{\Omega} b_k(q)\,u^{*}_{ik}(p,q)\,d\Omega + \int_{\Gamma} p_k(q)\,u^{*}_{ik}(p,q)\,d\Gamma, \tag{5.68}$$

o qual expressa que o trabalho realizado pelo sistema de forças de volume b^{*}_{ik} e de superfície p^{*}_{ik} sobre os deslocamentos u_k é igual ao trabalho realizado pelo sistema de forças de volume b_k e de superfície p_k sobre os deslocamentos u^{*}_{ik}. Notemos que o primeiro termo da Equação (5.68) é igual ao primeiro termo da Equação (5.62) com sinal trocado,

$$\int_{\Omega} \sigma^{*}_{ijk,j}(p,q)\,u_k(q)\,d\Omega = -\int_{\Omega} b^{*}_{ik}(p,q)\,u_k(q)\,d\Omega, \tag{5.69}$$

e a identidade Somigliana é escrita substituindo-se, primeiro, a Equação (5.61) e, depois, a Equação (5.65) na expressão do teorema da reciprocidade de Betti.

Equação integral de contorno para pontos externos A identidade Somigliana é a equação integral de contorno para os deslocamentos em um ponto $p \in \Omega$. Se o ponto fonte p gerador da solução fundamental de Kelvin é externo ao contorno do sólido, ou seja, se $p \notin \{\Omega + \Gamma\}$, a aplicação da propriedade de seleção (5.64) à identidade Somigliana resulta na equação integral de contorno para os deslocamentos de pontos externos,

$$0 = \int_{\Gamma} u^{*}_{ik}(p,q)\,p_k(q)\,d\Gamma - \int_{\Gamma} p^{*}_{ik}(p,q)\,u_k(q)\,d\Gamma + \int_{\Omega} u^{*}_{ik}(p,q)\,b_k(q)\,d\Omega. \tag{5.70}$$

Equação integral de contorno para pontos de contorno Consideremos, agora, que o ponto fonte p esteja sobre o contorno do sólido, ou seja, $p \in \Gamma$. Nesse caso, como podemos observar nas expressões (4.66) e (4.67), as integrais de contorno da identidade Somigliana apresentam singularidades $0(1/r)$ e $0(1/r^2)$ em p, respectivamente. Para tratar essas singularidades, vamos supor que o sólido possa ser representado como mostrado na Figura 5.2, onde ao domínio Ω foi adicionada parte de uma esfera infinitesimal

Ω_ε de raio ε, centrada em p. A identidade Somigliana, Equação (5.66), pode ser escrita para o novo domínio como

$$u_i(p) = \int_{\Gamma-\overline{\Gamma}_\varepsilon+\Gamma_\varepsilon} u^*_{ik}(p,q)\, p_k(q)\, d\Gamma - \int_{\Gamma-\overline{\Gamma}_\varepsilon+\Gamma_\varepsilon} p^*_{ik}(p,q)\, u_k(q)\, d\Gamma + \int_{\Omega+\Omega_\varepsilon} u^*_{ik}(p,q)\, b_k(q)\, d\Omega. \quad (5.71)$$

A equação integral para deslocamentos de pontos sobre o contorno é obtida tomando-se, na Equação (5.71), o limite para ε tendendo a zero [18, 57, 122],

$$c_{ik}(p)\, u_i(p) + \int_\Gamma p^*_{ik}(p,q)\, u_k(q)\, d\Gamma = \int_\Gamma u^*_{ik}(p,q)\, p_k(q)\, d\Gamma + \int_\Omega u^*_{ik}(p,q)\, b_k(q)\, d\Omega, \quad (5.72)$$

onde a integral do lado esquerdo deve ser interpretada no sentido do valor principal de Cauchy. O tensor c_{ik} vale

$$c_{ik}(p) = \delta_{ik} + \lim_{\varepsilon\to 0} \int_{\Gamma_\varepsilon} p^*_{ik}(p,q)\, d\Gamma. \quad (5.73)$$

Se o plano tangente em p é contínuo, $c_{ik} = \delta_{ik}/2$ [18]; caso contrário, expressões analíticas de c_{ik}, para o caso tridimensional, podem ser escontradas em CODA [23].

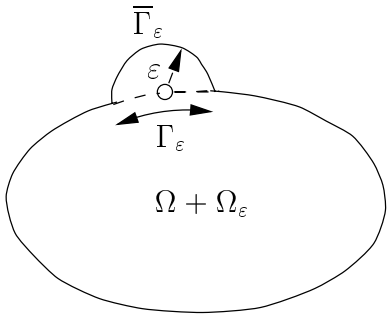


Figura 5.2: Ponto singular sobre o contorno acrescido de Ω_ε.

As equações integrais de contorno para pontos internos, pontos externos e pontos de contorno apresentadas anteriormente podem ser unicamente representadas pela Equação (5.72), se considerarmos, para pontos externos, $c_{ik} = 0$, para pontos internos, $c_{ik} = \delta_{ik}$, e para pontos de contorno, c_{ik} como dado pela Equação (5.73).

Tensões em pontos internos A equação integral para tensões em um ponto $p \in \Omega$ é derivada da Equação (5.66), considerando as relações deformação-deslocamento e a lei de Hooke,

$$\sigma_{ij}(p) = \int_\Gamma u^*_{ijk}(p,q)\, p_k(q)\, d\Gamma - \int_\Gamma p^*_{ijk}(p,q)\, u_k(q)\, d\Gamma + \int_\Omega u^*_{ijk}(p,q)\, b_k(q)\, d\Omega, \quad (5.74)$$

onde os tensores u^*_{ijk} e p^*_{ijk}, para a solução fundamental de Kelvin, são

$$u^*_{ijk} = -\,\sigma^*_{ijk}, \tag{5.75}$$

$$\begin{aligned} p^*_{ijk} =& \frac{G}{4\pi(1-\nu)r^3}\Bigg\{3\frac{\partial r}{\partial n}[(1-2\nu)\delta_{ij}\,r_{,k} + \nu(\delta_{ik}\,r_{,j} + \delta_{jk}\,r_{,i}) - 5r_{,i}r_{,j}r_{,k}] \\ & 3\nu(n_i\,r_{,j}\,r_{,k} + n_j\,r_{,i}\,r_{,k}) + (1-2\nu)(3n_k\,r_i\,r_j + n_j\,\delta_{ik} + n_i\,\delta_{jk}) \\ & -(1-4\nu)n_k\,\delta_{ij}\Bigg\}. \end{aligned} \tag{5.76}$$

Uma vez conhecidas as tensões em um ponto interno, as deformações podem ser determinadas através das relações constitutivas.

5.6 Método dos Elementos de Contorno

Seja um sólido contínuo, homogêneo, isotrópico e elástico, definido por um domínio Ω e um contorno Γ. No método dos elementos de contorno, a discretização do sólido consiste na subdivisão de seu contorno Γ em um conjunto de células, os *elementos de contorno*, os quais são conectados em pontos discretos chamados *nós*. As coordenadas de um nó são tomadas em relação ao sistema global de coordenadas. Um elemento de contorno é um trecho de superfície do contorno definido geometricamente por uma seqüência ordenada dos nós nos quais o elemento incide, exatamente como os elementos finitos. Da mesma forma, o contorno Γ pode ser *geometricamente* representado por um modelo de decomposição por células.

A formulação do método é baseada nas equações integrais de contorno, definidas na Seção 5.5. Não consideraremos, por ora, as forças de volume. Temos, então,

$$c_{ik}(p)\,u_i(p) + \int_\Gamma p^*_{ik}(p,q)\,u_k(q)\,d\Gamma = \int_\Gamma u^*_{ik}(p,q)\,p_k(q)\,d\Gamma. \tag{5.77}$$

A Equação anterior pode ser expressa matricialmente como

$$\mathbf{c}(p)\,\mathbf{u}(p) + \int_\Gamma \mathbf{p}^*\,\mathbf{u}\,d\Gamma = \int_\Gamma \mathbf{u}^*\,\mathbf{p}\,d\Gamma, \tag{5.78}$$

onde $\mathbf{u}^*$ e $\mathbf{p}^*$ são dados pelas Equações (4.65) e (4.66), respectivamente. Com a subdivisão do contorno em NE elementos de contorno, a Equação (5.78) pode ser escrita como somas de integrais sobre cada elemento de contorno do modelo:

$$\mathbf{c}(p)\,\mathbf{u}(p) + \sum_{e=1}^{\mathrm{NE}} \left(\int_{\Gamma^{(e)}} \mathbf{p}^*\,\mathbf{u}^{(e)} d\Gamma\right) = \sum_{e=1}^{\mathrm{NE}} \left(\int_\Gamma \mathbf{u}^*\,\mathbf{p}^{(e)}\,d\Gamma\right), \tag{5.79}$$

onde $\mathbf{u}^{(e)}$ e $\mathbf{p}^{(e)}$ são, respectivamente, os deslocamentos e as forças de superfície em um ponto qualquer q sobre a superfície $\Gamma^{(e)}$ do elemento e. Analogamente aos elementos finitos, vamos escrever funções aproximadoras para os deslocamentos e forças de superfície de cada elemento de contorno do modelo. Para um elemento e definido por n

nós, temos

$$\begin{aligned}
\mathbf{u}^{(e)} &= \mathbf{N}^{(e)}\,\mathbf{U}^{(e)} = \begin{bmatrix}\mathbf{N}_1^{(e)} & \mathbf{N}_2^{(e)} & \cdots & \mathbf{N}_n^{(e)}\end{bmatrix}\begin{bmatrix}\mathbf{u}_1^{(e)}\\ \mathbf{u}_2^{(e)}\\ \cdots\\ \mathbf{u}_n^{(e)}\end{bmatrix} = \sum_{i=1}^{n}\mathbf{N}_i^{(e)}\,\mathbf{u}_i^{(e)},\\
\mathbf{p}^{(e)} &= \mathbf{N}^{(e)}\,\mathbf{P}^{(e)} = \begin{bmatrix}\mathbf{N}_1^{(e)} & \mathbf{N}_2^{(e)} & \cdots & \mathbf{N}_n^{(e)}\end{bmatrix}\begin{bmatrix}\mathbf{p}_1^{(e)}\\ \mathbf{p}_2^{(e)}\\ \cdots\\ \mathbf{p}_n^{(e)}\end{bmatrix} = \sum_{i=1}^{n}\mathbf{N}_i^{(e)}\,\mathbf{p}_i^{(e)},
\end{aligned} \tag{5.80}$$

onde $\mathbf{u}_i^{(e)}$, $\mathbf{p}_i^{(e)}$ e $\mathbf{N}_i^{(e)}$ são, respectivamente, o vetor de componentes de deslocamento, o vetor de componentes de forças de superfície e a matriz de interpolação do i-ésimo nó do elemento e. Substituindo as Equações (5.80) na expressão (5.79), obtemos

$$\mathbf{c}(p)\,\mathbf{u}(p) + \sum_{e=1}^{\mathrm{NE}}\left(\int_{\Gamma^{(e)}}\mathbf{p}^*\,\mathbf{N}^{(e)}d\Gamma\right)\mathbf{U}^{(e)} = \sum_{e=1}^{\mathrm{NE}}\left(\int_{\Gamma}\mathbf{u}^*\,\mathbf{N}^{(e)}\,d\Gamma\right)\mathbf{P}^{(e)}. \tag{5.81}$$

A Equação (5.81) pode ser colocada na forma

$$\mathbf{c}(p)\,\mathbf{u}(p) + \sum_{e=1}^{\mathrm{NE}}\mathbf{H}^{(e)}\,\mathbf{U}^{(e)} = \sum_{e=1}^{\mathrm{NE}}\mathbf{G}^{(e)}\,\mathbf{P}^{(e)}, \tag{5.82}$$

onde

$$\begin{aligned}
\mathbf{H}^{(e)} &= \int_{\Gamma^{(e)}}\mathbf{p}^*\,\mathbf{N}^{(e)}d\Gamma \quad \text{e}\\
\mathbf{G}^{(e)} &= \int_{\Gamma^{(e)}}\mathbf{u}^*\,\mathbf{N}^{(e)}d\Gamma
\end{aligned} \tag{5.83}$$

são as *matrizes de influência* do elemento e. Para um modelo com N nós, o desenvolvimento das integrais da expressão (5.82), aplicada a um ponto fonte p qualquer no contorno ou localizado fora do sólido, produz uma equação do tipo

$$\begin{bmatrix}\mathbf{H}_{i1} & \mathbf{H}_{i2} & \cdots & \mathbf{H}_{i\mathrm{N}}\end{bmatrix}\begin{bmatrix}\mathbf{u}_1\\ \mathbf{u}_2\\ \cdots\\ \mathbf{u}_{\mathrm{N}}\end{bmatrix} = \begin{bmatrix}\mathbf{G}_{i1} & \mathbf{G}_{i2} & \cdots & \mathbf{G}_{i\mathrm{N}}\end{bmatrix}\begin{bmatrix}\mathbf{p}_1\\ \mathbf{p}_2\\ \cdots\\ \mathbf{p}_{\mathrm{N}}\end{bmatrix}, \tag{5.84}$$

onde $\mathbf{u}_i$ e $\mathbf{p}_i$ são, respectivamente, deslocamento e forças de superfície no i-ésimo nó do modelo. (No caso tridimensional, $\mathbf{H}_{ij}$ e $\mathbf{G}_{ij}$ são submatrizes 3×3.) Para determinarmos uma solução para o problema, devemos escrever um sistema de N equações do tipo (5.84) (no caso tridimensional, com $3\times\mathrm{N}$ incógnitas), ou seja, devemos aplicar a Equação (5.82) em N pontos fontes distintos. Dessa forma, obteremos

$$\begin{bmatrix}\mathbf{H}_{11} & \mathbf{H}_{12} & \cdots & \mathbf{H}_{1\mathrm{N}}\\ \mathbf{H}_{21} & \mathbf{H}_{22} & \cdots & \mathbf{H}_{2\mathrm{N}}\\ \cdots & & & \\ \mathbf{H}_{\mathrm{N}1} & \mathbf{H}_{\mathrm{N}2} & \cdots & \mathbf{H}_{\mathrm{NN}}\end{bmatrix}\begin{bmatrix}\mathbf{u}_1\\ \mathbf{u}_2\\ \cdots\\ \mathbf{u}_{\mathrm{N}}\end{bmatrix} = \begin{bmatrix}\mathbf{G}_{11} & \mathbf{G}_{12} & \cdots & \mathbf{G}_{1\mathrm{N}}\\ \mathbf{G}_{21} & \mathbf{G}_{22} & \cdots & \mathbf{G}_{2\mathrm{N}}\\ \cdots & & & \\ \mathbf{G}_{\mathrm{N}1} & \mathbf{G}_{\mathrm{N}2} & \cdots & \mathbf{G}_{\mathrm{NN}}\end{bmatrix}\begin{bmatrix}\mathbf{p}_1\\ \mathbf{p}_2\\ \cdots\\ \mathbf{p}_{\mathrm{N}}\end{bmatrix}, \tag{5.85}$$

ou

$$\mathbf{H}\,\mathbf{U} = \mathbf{G}\,\mathbf{P}. \tag{5.86}$$

É usual, no método dos elementos de contorno, considerarmos como pontos fontes pontos externos ou os próprios nós de contorno do modelo. Para um ponto p externo, temos, na Equação (5.82), $\mathbf{c}(p) = \mathbf{0}$. Se o ponto fonte for o j-ésimo nó do modelo, o elemento $\mathbf{H}_{jj}$ da diagonal do lado esquerdo do sistema (5.85) será acrescido do termo $\mathbf{c}_j$ (determinado explicitamente ou através de considerações de *movimento de corpo rígido* [18, 122]).

Agora podemos aplicar as condições de contorno $\mathbf{u}_i = \overline{\mathbf{u}}_i$ em Γ_1 e $\mathbf{p}_i = \overline{\mathbf{p}}_i$ em Γ_2 no sistema linear (5.86). Se os deslocamentos são conhecidos, podemos encontrar as forças de superfície e vice-versa. Isso implica que o sistema pode ser reordenado tal que as incógnitas sejam escritas no lado esquerdo, resultando

$$\mathbf{A}\,\mathbf{X} = \mathbf{F}. \tag{5.87}$$

A solução do sistema (5.87) fornece os valores dos deslocamentos e forças de superfície incógnitos sobre o contorno do sólido, a partir dos quais podemos determinar os valores no domínio. A seguir, sumarizamos os passos computacionais do método dos elementos de contorno.

5.6.1 Esquema Computacional

Passo 1 **Discretização do contorno**. O contorno Γ é discretizado em uma série de NE elementos sobre os quais são interpolados, em termos de funções aproximadoras e valores nodais, deslocamentos e forças de superfície. Como no caso dos elementos finitos, esse passo é chamado de pré-processamento.

Passo 2 **Computação das contribuições de contorno**. A Equação (5.82) é aplicada a N pontos distintos. As matrizes de influência $\mathbf{H}^{(e)}$ e $\mathbf{G}^{(e)}$ são determinadas para cada elemento e, usualmente através de esquemas numéricos de quadratura.

Passo 3 **Montagem do sistema linear**. A partir das contribuições individuais de todos os NE elementos de contorno do modelo o sistema linear (5.86) é formado.

Passo 4 **Introdução das condições de contorno**. O sistema linear (5.86), após a introdução das condições de contorno, é reescrito na forma da Equação (5.87).

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.

Passo 6 **Computação de valores no domínio**. 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. Deslocamentos e tensões no interior do corpo podem ser computados através da identidade Somigliana e da Equação (5.74), respectivamente. É a etapa de pós-processamento.

5.7 Sumário

Nesse Capítulo vimos os passos básicos do método dos elementos finitos e do método dos elementos de contorno. Desenvolvemos as formulações de ambos os métodos a partir de representações integrais das equações diferenciais que governam o problema fundamental, obtidas a partir do método mais geral de resíduos ponderados.

A formulação do método dos elementos finitos foi derivada da expressão integral do princípio dos trabalhos virtuais (PTV). Escrevemos o PTV como uma forma fraca da equação de equilíbrio, supondo uma função aproximadora satisfazendo as condições de contorno essenciais e uma função ponderadora de Galerkin satisfazendo a versão homogênea das condições essenciais. Com a subdivisão do domínio Ω e do contorno Γ de um corpo em um modelo de elementos finitos, a integral do PTV pôde ser representada por somas de integrais sobre todos os elementos finitos. O resultado foi um sistema linear cuja solução fornece os deslocamentos nodais incógnitos do modelo, a partir dos quais podemos determinar os valores no domínio.

A formulação do método dos elementos de contorno foi derivada da sentença inversa da expressão de resíduos ponderados para o problema fundamental, da qual escrevemos as equações integrais de contorno para deslocamentos de pontos localizados no interior, no exterior e sobre o contorno Γ de um corpo de domínio Ω. Com a subdivisão do 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, sendo N o número de nós do modelo. A solução do sistema fornece os deslocamentos e forças de superfície incógnitos, a partir dos quais podem ser determinados os valores de domínio.

CAPÍTULO 6

Modelos Mecânicos

6.1 Introdução

No Capítulo 5 apresentamos as formulações dos métodos de análise numérica que utilizaremos na resolução do modelo matemático do problema fundamental: método dos elementos finitos e método dos elementos de contorno. Vimos que o método dos elementos finitos é baseado na discretização do domínio e do contorno do corpo em consideração, e que o método dos elementos de contorno é baseado na discretização somente do contorno do corpo. Essas discretizações consistem na divisão do domínio e do contorno do corpo, ou somente do contorno, em elementos finitos ou de contorno. Uma *malha* de elementos finitos ou de contorno, juntamente com a definição das propriedades materiais, dos casos de carregamento e das condições de contorno do problema, definem o que chamaremos de *modelo mecânico*, ou *modelo de análise* do problema.

Nesse Capítulo apresentaremos as formulações dos elementos finitos e de contorno empregados, respectivamente, na modelagem mecânica de cascas delgadas elásticas pelo método dos elementos finitos e de sólidos elásticos pelo método dos elementos de contorno. Não temos como objetivo desenvolver um novo elemento estrutural, tão pouco efetuar testes comparativos de desempenho dos elementos aqui considerados com outros tipos de elementos estruturais. Por isso, apresentaremos as formulações diretamente, sem demonstrações, tal como dadas na literatura. Maiores detalhes a respeito dos elementos podem ser encontrados nas referências bibliográficas indicadas.

De acordo com BATOZ, BATHE e HO [10], há essencialmente três procedimentos que podem ser seguidos para o desenvolvimento de um elemento finito de casca: (1)uma teoria particular de cascas é usada; (2)as equações da mecânica do contínuo são usadas e discretizadas; e (3)as matrizes de rigidez de dois elementos, um de placa e um de membrana, são combinadas e montadas em relação a um sistema global de coordenadas. Seguiremos a última estratégia e apresentaremos, na Seção 6.2, os elementos finitos de placa e de membrana utilizados na composição do elemento finito de casca delgada. Escolhemos como elemento de placa o DKT, acrônimo de *Discrete Kirchhoff Theory*, e como elemento de membrana, um elemento com liberdades rotacionais baseado na *formulação livre* de BERGAN e NYGARD [13]. Nossa escolha foi fundamentada nas comprovações de eficiência numérica desses elementos, dadas, respectivamente, nos trabalhos de CARRIJO [20] e PELETEIRO [86]. Além disso, a consideração de liberdades

rotacionais no elemento de membrana evita a necessidade de incluirmos na formulação a chamada "rigidez rotacional fictícia" na direção z (veja ZIENKIEWICZ [133, página 114]). Os elementos de placa e de membrana, e consequentemente o elemento de casca resultante da composição de ambos, são triangulares. Elementos quadrilaterais podem ser obtidos da união de quatro elementos triangulares, com posterior condensação estática dos parâmetros internos (veja, por exemplo, PELETEIRO [86, página 57]). Consideraremos, no entanto, somente os elementos triangulares.

Na Seção 6.3 apresentaremos os elementos de contorno utilizados na modelagem mecânica das superfícies de contorno de um sólido. Inspirados nos resultados obtidos por CODA [23], escolhemos os elementos quadrilaterais isoparamétricos contínuos linear e quadrático, definidos, respectivamente, por quatro e por oito nós. Mostraremos, também, como derivar as funções de forma dos elementos descontínuos de quatro e oito nós, úteis na simulação de descontinuidades de forças de superfície sobre o contorno do sólido.

Na Seção 6.4 descreveremos os passos básicos do processo de discretização das faces de um modelo geométrico de cascas ou de sólidos, definidos no Capítulo 3. O algoritmo utilizado é uma extensão, para o caso de faces no espaço, do algoritmo de geração de malhas planas de forma livre proposto por SEZER e ZEID [102].

6.2 Elementos Finitos

Nessa Seção apresentaremos a formulação do elemento finito de casca delgada resultante da combinação do elemento triangular de placa DKT e do elemento triangular de membrana com liberdades rotacionais. Vejamos, primeiro, esses dois elementos separadamente.

6.2.1 Elemento Finito de Placa

O elemento finito DKT é um elemento definido por três nós e nove graus de liberdade, três graus de liberdade por nó, como indicado na Figura 6.1. Os graus de liberdade nodais são o deslocamento transversal w e as rotações θ_x e θ_y, os quais podem ser matricialmente organizados como

$$\mathbf{U}_{\mathrm{P}} = \begin{bmatrix} w_1 & \theta_{x1} & \theta_{y1} & w_2 & \theta_{x2} & \theta_{y2} & w_3 & \theta_{x3} & \theta_{y3} \end{bmatrix}^T . \tag{6.1}$$

Os esforços nodais correspondentes aos graus de liberdade são a força vertical F_z e os momentos fletores M_x e M_y. Matricialmente, temos

$$\mathbf{F}_{\mathrm{P}} = \begin{bmatrix} F_{z1} & M_{x1} & M_{y1} & F_{z2} & M_{x2} & M_{y2} & F_{z3} & M_{x3} & M_{y3} \end{bmatrix}^T . \tag{6.2}$$

A geometria do elemento é mostrada na Figura 6.2. Definiremos as seguintes relações geométricas para o triângulo da figura:

$$\begin{aligned} x_{ij} &= x_i - x_j, \\ y_{ij} &= y_i - y_j, \\ l_{ij} &= (x_{ij}^2 + y_{ij}^2)^{1/2}, \\ x_m &= (x_i + x_j)/2, \\ y_m &= (y_i + y_j)/2, \end{aligned} \tag{6.3}$$

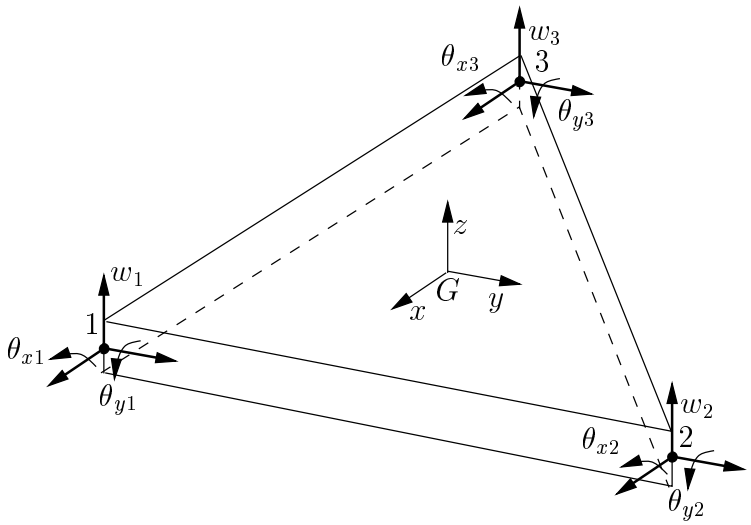


Figura 6.1: Elemento de placa: graus de liberdade em coordenadas locais.

onde $i = 1, 2, 3$ representa um vértice de canto do triângulo, $m = 4, 5, 6$ representa o vértice médio do lado ij e l_{ij} é o comprimento do lado ij.

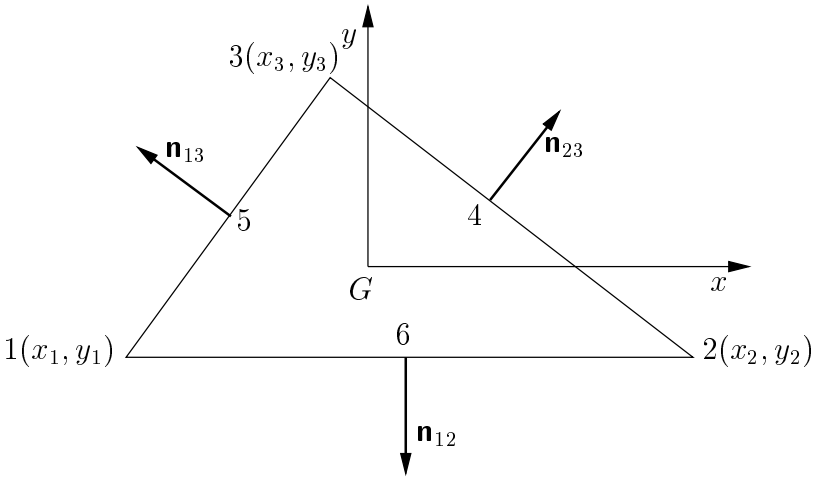


Figura 6.2: Geometria do elemento de placa.

A matriz de rigidez do elemento DKT é desenvolvida por BATOZ, BATHE e HO [10] a partir das seguintes suposições:

- As rotações θ_x e θ_y variam quadraticamente sobre o elemento:

$$\theta_x = \sum_{i=1}^{6} N_i\, \theta_{xi} \qquad \text{e} \qquad \theta_y = \sum_{i=1}^{6} N_i\, \theta_{yi}, \tag{6.4}$$

  onde θ_{xi} e θ_{yi} são os valores nodais das rotações nos cantos e nos vértices médios do elemento e N_i são as funções de forma definidas pela Equação (3.21).

- A hipótese de Kirchhoff, definida no Capítulo 4, é imposta nos vértices de canto

$$\begin{bmatrix} \theta_x + w_{,x} \\ \theta_y + w_{,y} \end{bmatrix} = \mathbf{0} \tag{6.5}$$

e nos vértices médios do elemento,

$$\theta_{s_m} + w_{,s_m} = 0, \qquad m = 4, 5, 6. \tag{6.6}$$

- A variação de w é cúbica nos lados do elemento:

$$w_{,s_m} = -\frac{3}{2l_{ij}} w_i - \frac{1}{4} w_{,s_i} + \frac{3}{2l_{ij}} w_j - \frac{1}{4} w_{,s_j}. \tag{6.7}$$

- Uma variação linear de θ_n é imposta nos lados do elemento:

$$\theta_{nm} = \frac{1}{2}(\theta_{ni} + \theta_{nj}), \tag{6.8}$$

onde $m = 4, 5, 6$ denota o vértice médio dos lados 23, 31 e 12, respectivamente.

A matriz de rigidez do elemento DTK é definida como

$$\mathbf{K}_{\mathrm{P}} = 2A \int_0^1 \int_0^{1-\eta} \mathbf{B}_P^T \, \mathbf{D}_P \, \mathbf{B}_p \, d\xi d\eta, \tag{6.9}$$

onde A é a área do elemento (a integral é tomada em relação ao sistema de coordenadas adimensionais da célula triangular). A matriz constitutiva da placa foi definida no Capítulo 4 como

$$\mathbf{D}_{\mathrm{P}} = \frac{Eh^3}{12(1-\nu^2)} \begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1}{2}(1-\nu) \end{bmatrix}. \tag{6.10}$$

A partir das hipóteses anteriores, BATOZ, BATHE e HO definem a matriz $\mathbf{B}_{\mathrm{P}}$ como sendo

$$\mathbf{B}_{\mathrm{P}}(\xi, \eta) = \frac{1}{2A} \begin{bmatrix} +y_{31}\mathbf{H}_{x,\xi}^T + y_{12}\mathbf{H}_{x,\eta}^T \\ -x_{31}\mathbf{H}_{y,\xi}^T - x_{12}\mathbf{H}_{y,\eta}^T \\ -x_{31}\mathbf{H}_{x,\xi}^T - x_{12}\mathbf{H}_{x,\eta}^T + x_{31}\mathbf{H}_{y,\xi}^T + y_{12}\mathbf{H}_{y,\eta}^T \end{bmatrix}, \tag{6.11}$$

onde as matrizes de funções de forma $\mathbf{H}_x$ e $\mathbf{H}_y$ possuem 9 componentes cada, sendo

$$\begin{aligned} H_{x1} &= 1.5(a_6 N_6 - a_5 N_5), \\ H_{x2} &= b_5 N_5 + b_6 N_6, \\ H_{x3} &= N_1 - c_5 N_5 - c_6 N_6, \\ H_{y1} &= 1.5(d_6 N_6 - d_5 N_5), \\ H_{y2} &= -N_1 + e_5 N_5 + e_6 N_6, \\ H_{y3} &= -H_{x2}. \end{aligned} \tag{6.12}$$

As funções H_{x4}, H_{x5}, H_{x6}, H_{y4}, H_{y5} e H_{y6} são obtidas das expressões (6.12) trocando-se N_1 por N_2 e os índices 6 e 5 por 4 e 6, respectivamente. As funções H_{x7}, H_{x8}, H_{x9},

H_{y7}, H_{y8} e H_{y9} são obtidas das expressões (6.12) trocando-se N_1 por N_3 e os índices 6 e 5 por 5 e 4, respectivamente. Além disso, temos

$$\begin{aligned} a_m &= -x_{ij}/l_{ij}^2, \\ b_m &= \frac{3}{4}x_{ij}y_{ij}/l_{ij}^2, \\ c_m &= (\frac{1}{4}x_{ij}^2 - \frac{1}{2}y_{ij}^2)/l_{ij}^2, \\ d_m &= -y_{ij}/l_{ij}^2, \\ e_m &= (\frac{1}{4}y_{ij}^2 - \frac{1}{2}x_{ij}^2)/l_{ij}^2, \end{aligned} \tag{6.13}$$

onde $m = 4, 5, 6$ para os lados $ij = 23, 31, 12$, respectivamente. As derivadas de $\mathbf{H}_x$ em relação às coordenadas adimensionais ξ e η valem, respectivamente,

$$\mathbf{H}_{x,\xi} = \begin{bmatrix} P_6(1-2\xi) + (P_5+P_6)\eta \\ q_6(1-2\xi) - (q_5+q_6)\eta \\ -4 + 6(\xi+\eta) + r_6(1-2\xi) - (r_5+r_6)\eta \\ -P_6(1-2\xi) + (P_6+P_4)\eta \\ q_6(1-2\xi) + (q_4+q_6)\eta \\ -2 + 6\xi + r_6(1-2\xi) + (r_4-r_6)\eta \\ -\eta(P_4+P_5) \\ \eta(q_4-q_5) \\ \eta(r_4-r_5) \end{bmatrix} \tag{6.14}$$

e

$$\mathbf{H}_{x,\eta} = \begin{bmatrix} -P_5(1-2\eta) + (P_5-P_6)\xi \\ q_5(1-2\eta) - (q_5+q_6)\eta \\ -4 + 6(\xi+\eta) + r_5(1-2\eta) - (r_5+r_6)\xi \\ \xi(P_4+P_6) \\ \xi(q_4-q_6) \\ \xi(r_4-r_6)P_5(1-2\eta) - (P_4+P_5)\xi \\ q_5(1-2\eta) + (q_4-q_5)\xi \\ -2 + 6\xi + r_5(1-2\eta) + (r_4-r_5)\eta \end{bmatrix}. \tag{6.15}$$

As derivadas de $\mathbf{H}_y$ em relação às coordenadas adimensionais ξ e η valem, respectivamente,

$$\mathbf{H}_{y,\xi} = \begin{bmatrix} t_6(1-2\xi) + (t_5-t_6)\eta \\ 1 + r_6(1-2\xi) - (r_5+r_6)\eta \\ -q_6(1-2\xi) + (q_5+q_6)\eta \\ -t_6(1-2\xi) + (t_6+t_4)\eta \\ -1 + r_6(1-2\xi) + (r_4-r_6)\eta \\ -q_6(1-2\xi) - (q_4-q_6)\eta \\ -\eta(t_4+t_5) \\ \eta(r_4-r_5) \\ -\eta(q_4-q_5) \end{bmatrix} \tag{6.16}$$

e

$$\mathbf{H}_{y,\eta} = \begin{bmatrix} -t_5(1-2\eta) + (t_5 - t_6)\xi \\ 1 + r_5(1-2\eta) - (r_5 + r_6)\eta \\ -q_5(1-2\eta) + (q_5 + q_6)\xi \\ \xi(t_4 + t_6) \\ \xi(r_4 - r_6) \\ -\xi(q_4 - q_6)t_5(1-2\eta) - (t_4 + t_5)\xi \\ -1 + r_5(1-2\eta) + (r_4 - r_5)\xi \\ -q_5(1-2\eta) - (q_4 - q_5)\eta \end{bmatrix}. \tag{6.17}$$

Nas equações acima, temos

$$\begin{aligned} P_m &= -6x_{ij}/l_{ij}^2 = 6a_m, \\ r_m &= -6y_{ij}/l_{ij}^2 = 6d_m, \\ q_m &= 3x_{ij}y_{ij}/l_{ij}^2, \\ t_m &= 3y_{ij}^2/l_{ij}^2. \end{aligned} \tag{6.18}$$

A partir das equações anteriores, a matriz de rigidez do elemento DKT em relação ao sistema de coordenadas locais da Figura 6.1 pode ser obtida por integração numérica da Equação (6.9) (veja, por exemplo, ZIENKIEWICZ [132, página 175]).

O carregamento considerado sobre um elemento de placa é a carga uniformemente distribuída g, aplicada na direção do eixo z do sistema de coordenadas locais do elemento. O vetor de esforços nodais equivalentes correspondente ao carregamento distribuído g é

$$\mathbf{F}_{\mathrm{P}} = \frac{A}{3}\begin{bmatrix} g & 0 & 0 & g & 0 & 0 & g & 0 & 0 \end{bmatrix}^T. \tag{6.19}$$

6.2.2 Elemento Finito de Membrana

A seguir apresentaremos a matriz de rigidez e o vetor de esforços nodais equivalentes do elemento finito de membrana com liberdades rotacionais que utilizamos na modelagem mecânica de cascas. O desenvolvimento do elemento é baseado na formulação livre de BERGAN e NYGARD [13]. Não efetuaremos nenhuma descrição da formulação livre aqui, apenas apresentaremos as expressões necessárias para o cálculo das contribuições do elemento de membrana. Para maiores detalhes sobre o desenvolvimento do elemento, veja o artigo de BERGAN E FELIPPA [12] e a dissertação de PELETEIRO [86].

A geometria e os graus de liberdade de um elemento individual são mostrados na Figura 6.3. (Note que o sistema de coordenadas adimensionais possui origem no centróide do triângulo, diferentemente do sistema de coordenadas adimensionais da célula triangular da Figura 3.16. Apesar da distinção, usaremos esse sistema no elemento de membrana, de acordo com o desenvolvimento de BERGAN e FELIPPA.)

Os graus de liberdade nodais do elemento de membrana são os deslocamentos u e v no plano e a rotação θ_z, organizados matricialmente como

$$\mathbf{U}_{\mathrm{M}} = \begin{bmatrix} u_1 & v_1 & \theta_{z1} & u_2 & v_2 & \theta_{z2} & u_3 & v_3 & \theta_{z3} \end{bmatrix}^T, \tag{6.20}$$

onde a rotação θ_{zi} é definida como

$$\theta_{zi} = \frac{1}{2}\left(\frac{\partial v_i}{\partial x} - \frac{\partial u_i}{\partial y}\right). \tag{6.21}$$

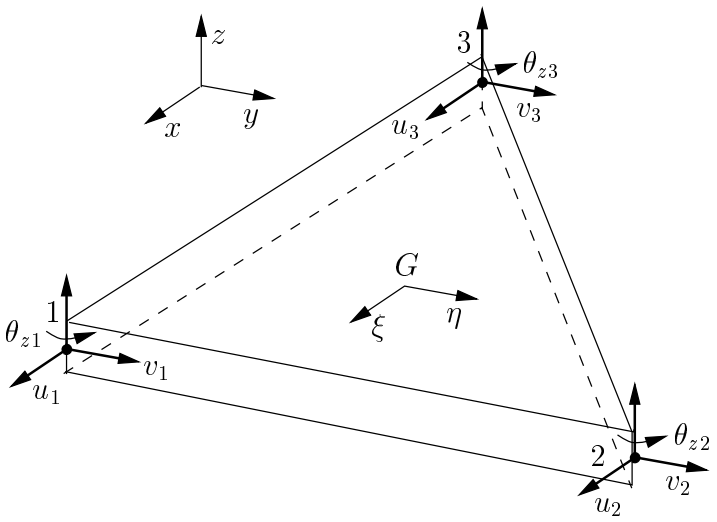


Figura 6.3: Elemento de membrana: geometria e graus de liberdade.

Os esforços nodais equivalentes são as forças F_x e F_y e o momento M_z. Matricialmente,

$$\mathbf{F}_{\mathrm{M}} = \begin{bmatrix} F_{x1} & F_{y1} & M_{z1} & F_{x2} & F_{y2} & M_{z2} & F_{x3} & F_{y3} & M_{z3} \end{bmatrix}^T. \tag{6.22}$$

Definiremos as seguintes relações geométricas para o triângulo da Figura 6.3:

$$\begin{aligned}
a_{1i} &= -\frac{1}{2} s_i c_i^2, \\
a_{2i} &= +c_i^3, \\
a_{3i} &= +\frac{1}{2} s_i^3 + s_i c_i^2, \\
b_{1i} &= -s_i^2 c_i - \frac{1}{2} c_i^3, \\
b_{2i} &= -s_i^3, \\
b_{3i} &= +\frac{1}{2} s_i^2 c_i, \\
s_i &= (y_m - y_i)/l_{mi}, \\
c_i &= (x_m - x_i)/l_{mi}, \\
x_m &= \frac{1}{2}(x_j + x_k), \\
y_m &= \frac{1}{2}(y_j + y_k), \\
l_{mi} &= [(x_m - x_i)^2 + (y_m - y_i)^2]^{1/2}, \\
\lambda &= \frac{1}{\sqrt{A}}.
\end{aligned} \tag{6.23}$$

Nas equações acima, i, j, k denotam permutações cíclicas positivas de $1, 2, 3$; por exemplo, $i = 2$, $j = 3$, $k = 1$.

A matriz de rigidez do elemento de membrana, de acordo com a formulação livre, é formada pela superposição da *matriz de rigidez de modos básicos* $\mathbf{K}_b$ e da *matriz de rigidez generalizada de alta ordem* $\mathbf{K}_h$:

$$\mathbf{K}_{\mathrm{M}} = \mathbf{K}_b + \beta\, \mathbf{K}_h, \tag{6.24}$$

sendo β um dos dois *parâmetros livres* da formulação. (BERGAN e FELIPPA sugerem $\beta = 0.5$.) A matriz de rigidez de modos básicos é definida como

$$\mathbf{K}_b = \frac{1}{A}\,\mathbf{L}\,\mathbf{D}_{\mathrm{M}}\,\mathbf{L}^T, \tag{6.25}$$

sendo $\mathbf{D}_{\mathrm{M}}$, a matriz constitutiva que relaciona forças de membrana por unidade de comprimento com deformações de membrana, dada por

$$\mathbf{D}_{\mathrm{M}} = \frac{Eh}{1-\nu^2}\begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1}{2}(1-\nu) \end{bmatrix}. \tag{6.26}$$

A matriz $\mathbf{L}$, chamada de matriz "amontoadora", expressa a relação entre os esforços nodais equivalentes do elemento e as forças de membrana por unidade de comprimento, sendo definida como

$$\mathbf{L} = \begin{bmatrix} \mathbf{L}_1 \\ \mathbf{L}_2 \\ \mathbf{L}_3 \end{bmatrix}, \tag{6.27}$$

onde

$$\mathbf{L}_j = \frac{1}{2}\begin{bmatrix} y_{ki} & 0 & x_{ik} \\ 0 & x_{ik} & y_{ki} \\ \frac{1}{6}\alpha(y_{ji}^2 - y_{kj}^2) & \frac{1}{6}\alpha(x_{ij}^2 - x_{lk}^2) & \frac{1}{3}\alpha(x_{ij}y_{ji} - x_{jk}y_{kj}) \end{bmatrix}. \tag{6.28}$$

O número α é o segundo parâmetro livre da formulação. Se $\alpha = 0$, $\mathbf{L}$ resulta na matriz de "amontoamento" do triângulo de deformação constante (CST). (BERGAN e FELIPPA sugerem $\alpha = 1.5$.)

A matriz de rigidez generalizada de alta ordem é dada por

$$\mathbf{K}_h = \mathbf{H}_h^T\,\mathbf{K}_{qh}\,\mathbf{H}_h, \tag{6.29}$$

onde

$$\mathbf{K}_{qh} = \int_A \mathbf{B}_h^T\,\mathbf{D}_{\mathrm{M}}\,\mathbf{B}_h\,dA. \tag{6.30}$$

Na equação acima, a matriz $\mathbf{B}_h$ vale

$$\mathbf{B}_h = \lambda\begin{bmatrix} 2a_{11}\xi + a_{21}\eta & 2a_{12}\xi + a_{22}\eta & 2a_{13}\xi + a_{23}\eta \\ b_{21}\xi + 2b_{31}\eta & b_{22}\xi + 2b_{32}\eta & b_{23}\xi + 2b_{33}\eta \\ -4b_{31}\xi - 4a_{11}\eta & -4b_{32}\xi - 4a_{12}\eta & -4b_{33}\xi - 4a_{13}\eta \end{bmatrix}. \tag{6.31}$$

Denotando a i-ésima coluna de $\mathbf{B}_h$ por

$$\mathbf{B}_{hi} = \xi\mathbf{B}_{\xi i} + \eta\mathbf{B}_{\eta i}, \tag{6.32}$$

onde $\mathbf{B}_\xi$ e $\mathbf{B}_\eta$ são os termos dos coeficientes de $\mathbf{B}_h$ associados com ξ e η, respectivamente, e assumindo que $\mathbf{D}_{\mathrm{M}}$ seja constante no elemento, o termo (i,j) de $\mathbf{K}_{qh}$ pode ser expresso, segundo BERGAN e FELIPPA, como

$$K_{qhij} = J_{\xi\xi}\mathbf{B}_{\xi i}^T\mathbf{D}_{\mathrm{M}}\mathbf{B}_{\xi j} + J_{\xi\eta}(\mathbf{B}_{\xi i}^T\mathbf{D}_{\mathrm{M}}\mathbf{B}_{\eta j} + \mathbf{B}_{\eta i}^T\mathbf{D}_{\mathrm{M}}\mathbf{B}_{\xi j}) + J_{\eta\eta}\mathbf{B}_{\eta i}^T\mathbf{D}_{\mathrm{M}}\mathbf{B}_{\eta j}, \tag{6.33}$$

onde

$$J_{\xi\xi} = \int_A \xi^2\, dA = -\frac{1}{6}A(\xi_1\xi_2 + \xi_2\xi_3 - \xi_3\xi_1),$$
$$J_{\xi\eta} = \int_A \xi\eta\, dA = \frac{1}{12}A(\xi_1\eta_2 + \xi_2\eta_3 - \xi_3\eta_1), \tag{6.34}$$
$$J_{\eta\eta} = \int_A \eta^2\, dA = -\frac{1}{6}A(\eta_1\eta_2 + \eta_2\eta_3 - \eta_3\eta_1).$$

A matriz $\mathbf{H}_h$ é tal que

$$\mathbf{H} = \begin{bmatrix}[\mathbf{H}_{rc}]_{6\times 9} & [\mathbf{H}_h]_{3\times 9}\end{bmatrix}_{9\times 9} = \mathbf{G}^{-1} = \begin{bmatrix}[\mathbf{G}_{rc}]_{9\times 6} & [\mathbf{G}_h]_{9\times 3}\end{bmatrix}^{-1}_{9\times 9}, \tag{6.35}$$

onde $\mathbf{G}$ é a matriz que relaciona os graus de liberdade nodais com o conjunto de modos de corpo rígido, de deformação constante e de alta ordem do elemento. A submatriz que relaciona os graus de liberdade nodais com os modos de corpo rígido e de deformação constante é

$$\mathbf{G}_{rc} = \begin{bmatrix} 1 & 0 & -\eta_1 & \xi_1 & 0 & \eta_1 \\ 0 & 1 & \xi_1 & 0 & \eta_1 & \xi_1 \\ 0 & 0 & \lambda & 0 & 0 & 0 \\ 1 & 0 & -\eta_2 & \xi_2 & 0 & \eta_2 \\ 0 & 1 & \xi_2 & 0 & \eta_2 & \xi_2 \\ 0 & 0 & \lambda & 0 & 0 & 0 \\ 1 & 0 & -\eta_3 & \xi_3 & 0 & \eta_3 \\ 0 & 1 & \xi_3 & 0 & \eta_3 & \xi_3 \\ 0 & 0 & \lambda & 0 & 0 & 0 \end{bmatrix}. \tag{6.36}$$

A submatriz que relaciona os graus de liberdade nodais com os modos de alta ordem é

$$\mathbf{G}_h = \begin{bmatrix} \mathbf{G}_{h11} & \mathbf{G}_{h12} & \mathbf{G}_{h13} \\ \mathbf{G}_{h21} & \mathbf{G}_{h22} & \mathbf{G}_{h23} \\ \mathbf{G}_{h31} & \mathbf{G}_{h32} & \mathbf{G}_{h33} \end{bmatrix}, \tag{6.37}$$

sendo

$$\mathbf{G}_{hij} = \begin{bmatrix} a_{1j}\xi_i^2 + a_{2j}\xi_i\eta_i + a_{3j}\eta_i^2 \\ b_{1j}\xi_i^2 + b_{2j}\xi_i\eta_i + b_{3j}\eta_i^2 \\ -\lambda(c_j\xi_i + s_j\eta_i) \end{bmatrix}. \tag{6.38}$$

A submatriz $\mathbf{H}_h$, necessária para o cálculo da matriz de rigidez de alta ordem, é obtida numericamente pela inversão da matriz $\mathbf{G}$.

Os carregamentos considerados em um elemento de membrana são as cargas distribuídas constantes t_n e t_s, aplicadas, respectivamente, na direção normal e na direção tangencial do lado ij. O vetor de esforços nodais equivalentes correspondente aos carregamentos da membrana é [86, página 71]

$$\mathbf{F}_{\mathrm{M}} = \begin{bmatrix} \frac{1}{2}\, l_{ji}t_s \\ \frac{1}{2}\, l_{ji}t_n \\ \frac{\alpha}{12}\, l_{ji}^2 t_n \\ \frac{1}{2}\, l_{ji}t_s \\ \frac{1}{2}\, l_{ji}t_n \\ -\frac{\alpha}{12}\, l_{ji}^2 t_n \end{bmatrix}. \tag{6.39}$$

6.2.3 Elemento Finito de Casca

O elemento finito de casca é formado pela composição do elemento finito de placa e o elemento finito de membrana com liberdades rotacionais. O elemento resultante possui três nós e 18 graus de liberdade nodais, 6 graus de liberdade por nó, conforme esquematizado na Figura 6.4.

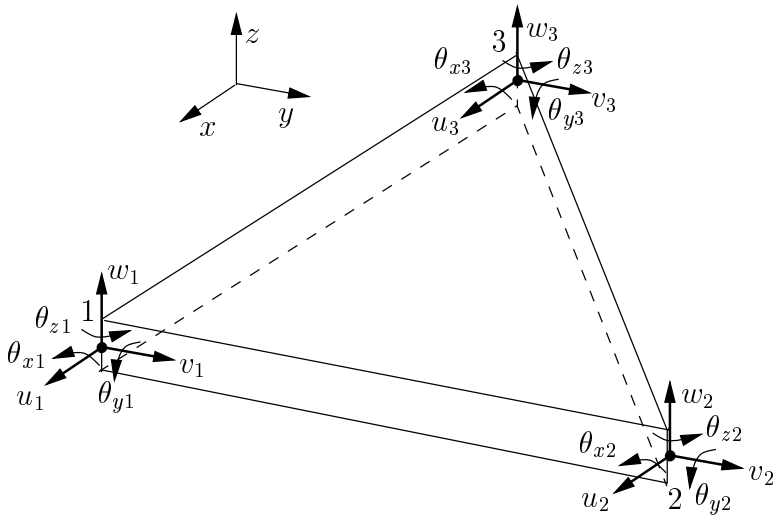


Figura 6.4: Elemento finito de casca.

Para um nó i do elemento, os graus de liberdade e os esforços equivalentes são, respectivamente

$$\mathbf{u}_{\mathrm{C}i} = \begin{bmatrix} u_i \\ v_i \\ w_i \\ \theta_{xi} \\ \theta_{yi} \\ \theta_{zi} \end{bmatrix} \qquad \text{e} \qquad \mathbf{f}_{\mathrm{C}_i} = \begin{bmatrix} F_{xi} \\ F_{yi} \\ F_{zi} \\ M_{xi} \\ M_{yi} \\ M_{zi} \end{bmatrix} . \tag{6.40}$$

A matriz de rigidez do elemento de casca é definida pelo acoplamento da matriz de rigidez do elemento de placa $\mathbf{K}_{\mathrm{P}}$ e da matriz de rigidez do elemento de membrana $\mathbf{K}_{\mathrm{M}}$. Esse acoplamento pode ser representado simbolicamente pela expressão

$$\mathbf{K}_{\mathrm{C}} = \mathbf{K}_{\mathrm{P}} + \mathbf{K}_{\mathrm{M}}, \tag{6.41}$$

sendo uma submatriz da matriz de rigidez do elemento de casca dada por

$$\mathbf{K}_{\mathrm{C}_{rs}} = \begin{bmatrix} K_{\mathrm{M}_{il}} & K_{\mathrm{M}_{im}} & 0 & 0 & 0 & K_{\mathrm{M}_{in}} \\ K_{\mathrm{M}_{jl}} & K_{\mathrm{M}_{jm}} & 0 & 0 & 0 & K_{\mathrm{M}_{jn}} \\ 0 & 0 & K_{\mathrm{P}_{il}} & K_{\mathrm{P}_{im}} & K_{\mathrm{P}_{in}} & 0 \\ 0 & 0 & K_{\mathrm{P}_{jl}} & K_{\mathrm{P}_{jm}} & K_{\mathrm{P}_{jn}} & 0 \\ 0 & 0 & K_{\mathrm{P}_{kl}} & K_{\mathrm{P}_{km}} & K_{\mathrm{P}_{kn}} & 0 \\ K_{\mathrm{M}_{kl}} & K_{\mathrm{M}_{km}} & 0 & 0 & 0 & K_{\mathrm{M}_{kn}} \end{bmatrix}, \tag{6.42}$$

$r = 1, 2, 3$, $s = 1, 2, 3$, $i = 3r - 2$, $j = 3r - 1$, $k = 3r$, $l = 3s - 2$, $m = 3s - 1$ e $n = 3s$. A matriz de rigidez (6.41) é definida em relação ao sistema local de coordenadas do

elemento de casca, mostrado na Figura 6.4. A transformação da matriz de rigidez $\mathbf{K}_{\mathrm{C}}$ para o sistema global de coordenadas é dada por

$$\mathbf{K}_{\mathrm{CG}} = \mathbf{T}^T \mathbf{K}_{\mathrm{C}} \mathbf{T}, \tag{6.43}$$

onde $\mathbf{K}_{\mathrm{CG}}$ é a matriz de rigidez do elemento no sistema global de coordenadas e, como mostrado em ZIENKIEWICZ [133, página 108],

$$\mathbf{T} = \begin{bmatrix} \mathbf{T}_0 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{T}_0 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{T}_0 \end{bmatrix}, \tag{6.44}$$

sendo

$$\mathbf{T}_0 = \begin{bmatrix} \mathbf{R}_c & \mathbf{0} \\ \mathbf{0} & \mathbf{R}_c \end{bmatrix}. \tag{6.45}$$

Na equação acima, $\mathbf{R}_c$ é a matriz de rotação de eixos definida no Capítulo 3 (na Seção 6.4 veremos como determinar os componentes de $\mathbf{R}_c$). Das Equações (6.42), (6.43) e (6.44), podemos escrever

$$\mathbf{K}_{\mathrm{CG}_{rs}} = \mathbf{T}_0^T \mathbf{K}_{\mathrm{C}_{rs}} \mathbf{T}_0. \tag{6.46}$$

6.3 Elementos de Contorno

Nessa Seção definiremos os elementos de contorno empregados na modelagem mecânica das superfícies de sólidos elásticos. Os graus de liberdade de um nó i de um elemento de contorno são os componentes de deslocamento

$$\mathbf{u}_i = \begin{bmatrix} u_{1i} & u_{2i} & u_{3i} \end{bmatrix}, \tag{6.47}$$

aos quais são associadas as forças de superfície

$$\mathbf{p}_i = \begin{bmatrix} p_{1i} & p_{2i} & p_{3i} \end{bmatrix}. \tag{6.48}$$

Tomaremos como funções interpoladoras para deslocamentos e forças de superfície sobre um elemento as mesmas funções de interpolação geométrica, ou seja, consideraremos os elementos como sendo isoparamétricos. Tipicamente, para um elemento definido por n nós, os deslocamentos e as forças de superfície sobre um ponto qualquer q do elemento são aproximados por

$$\begin{aligned} \mathbf{u} = \mathbf{N}\,\mathbf{U} &= \sum_{i=1}^{n} \mathbf{N}_i\,\mathbf{u}_i, \\ \mathbf{p} = \mathbf{N}\,\mathbf{P} &= \sum_{i=1}^{n} \mathbf{N}_i\,\mathbf{p}_i, \end{aligned} \tag{6.49}$$

onde $\mathbf{N}_i$, $\mathbf{u}_i$ e $\mathbf{p}_i$ são, respectivamente, as funções de forma, os deslocamentos e as forças de superfície do nó i. A partir das Equações (6.49), definimos as matrizes de influência

de um elemento e para um ponto fonte p, conforme vimos no Capítulo 5, como

$$\begin{aligned} \mathbf{H}^{(e)} &= \int_{\Gamma} \mathbf{p}^* \, \mathbf{N} \, d\Gamma \quad \text{e} \\ \mathbf{G}^{(e)} &= \int_{\Gamma} \mathbf{u}^* \, \mathbf{N} \, d\Gamma, \end{aligned} \tag{6.50}$$

onde os componentes de $\mathbf{p}$ e $\mathbf{u}$ vêm da solução fundamental de Kelvin. Usualmente, as integrais das Equações (6.50) são calculadas numericamente, através de um processo de quadratura, por exemplo, a quadratura Gaussiana. Nesse caso, as expressões são escritas como

$$\begin{aligned} \mathbf{H}^{(e)} &= \sum_{g=1}^{\mathrm{NG}} \left(\mathbf{p}^* \, \mathbf{N}(\xi, \eta) \, |\mathbf{G}|_g \, w_g \right), \\ \mathbf{G}^{(e)} &= \sum_{g=1}^{\mathrm{NG}} \left(\mathbf{u}^* \, \mathbf{N}(\xi, \eta) \, |\mathbf{G}|_g \, w_g \right), \end{aligned} \tag{6.51}$$

onde NG é o número de pontos de Gauss, ξ e η são as coordenadas do g-ésimo ponto de Gauss, w_g são os pesos de Gauss e $|\mathbf{G}|$ é o *Jacobiano* da transformação de coordenadas adimensionais para coordenadas globais, definido como

$$|\mathbf{G}| = (g_1^2 + g_2^2 + g_3^2)^{1/2}. \tag{6.52}$$

Na equação anterior, temos

$$\begin{aligned} g_1 &= \frac{\partial x_2}{\partial \xi} \frac{\partial x_3}{\partial \eta} - \frac{\partial x_2}{\partial \eta} \frac{\partial x_3}{\partial \xi}, \\ g_2 &= \frac{\partial x_3}{\partial \xi} \frac{\partial x_1}{\partial \eta} - \frac{\partial x_3}{\partial \eta} \frac{\partial x_1}{\partial \xi}, \\ g_3 &= \frac{\partial x_1}{\partial \xi} \frac{\partial x_2}{\partial \eta} - \frac{\partial x_1}{\partial \eta} \frac{\partial x_2}{\partial \xi}. \end{aligned} \tag{6.53}$$

Note que se tomarmos um fonte p pertencente ao contorno do elemento e, estaremos introduzindo singularidades nas integrais das Equações (6.50). Em nossa implementação contornamos esse problema adotando pontos fonte no exterior do sólido. Dessa forma, não necessitamos de quaisquer expressões analíticas ou do emprego de métodos especiais de quadratura.

A seguir, apresentaremos as funções de forma para os elementos quadrilateral linear e quadrilateral quadrático.

6.3.1 Elemento Quadrilateral Linear

O elemento quadrilateral linear é definido por 4 nós, como mostrado na Figura 3.16. Os deslocamentos e forças de superfície sobre o elemento são interpolados como

$$\begin{aligned} \mathbf{u} &= \mathbf{N}\,\mathbf{U} = \sum_{i=1}^{4} N_i \, \mathbf{u}_i, \\ \mathbf{p} &= \mathbf{N}\,\mathbf{U} = \sum_{i=1}^{4} N_i \, \mathbf{p}_i, \end{aligned} \tag{6.54}$$

onde N_i são as funções de forma definidas pelas Equações (3.18).

Elemento Descontínuo

Consideraremos, agora, o elemento descontínuo mostrado na Figura 6.5. (Podemos utilizar esse elemento na representação de descontinuidades de forças de superfície no contorno do sólido em análise.) A matriz de funções de forma do elemento pode ser determinada como segue.

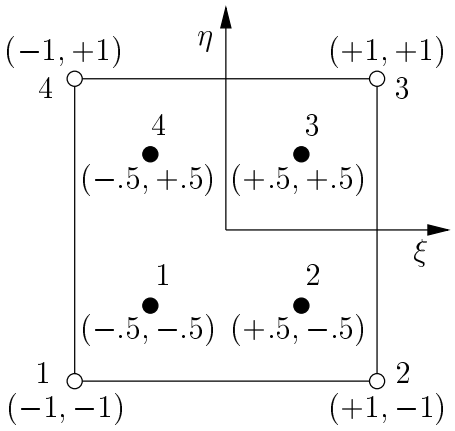


Figura 6.5: Elemento quadrilateral linear descontínuo.

Seja $\mathbf{v}_j$ qualquer quantidade (por exemplo, o deslocamento) na posição do *novo* vértice j. Podemos escrever

$$\mathbf{v}_j = \mathbf{N}\,\mathbf{U} = \sum_{i=1}^{4} N_i(\xi_j, \eta_j)\,\mathbf{u}_i, \tag{6.55}$$

onde $\mathbf{u}_i$ é a quantidade no vértice *padrão* i. Para $j = 1, 2, 3, 4$, temos

$$\begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \\ \mathbf{v}_3 \\ \mathbf{v}_4 \end{bmatrix} = \begin{bmatrix} N_1(\xi_1,\eta_1) & N_2(\xi_1,\eta_1) & N_3(\xi_1,\eta_1) & N_4(\xi_1,\eta_1) \\ N_1(\xi_2,\eta_2) & N_2(\xi_2,\eta_2) & N_3(\xi_2,\eta_2) & N_4(\xi_2,\eta_2) \\ N_1(\xi_3,\eta_3) & N_2(\xi_3,\eta_3) & N_3(\xi_3,\eta_3) & N_4(\xi_3,\eta_3) \\ N_1(\xi_4,\eta_4) & N_2(\xi_4,\eta_4) & N_3(\xi_4,\eta_4) & N_4(\xi_4,\eta_4) \end{bmatrix} \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \\ \mathbf{u}_3 \\ \mathbf{u}_4 \end{bmatrix}, \tag{6.56}$$

ou

$$\mathbf{V} = \mathbf{Q}\,\mathbf{U}. \tag{6.57}$$

Da Equação (6.57), obtemos

$$\mathbf{U} = \mathbf{Q}^{-1}\,\mathbf{V}, \tag{6.58}$$

o que nos permite escrever, a partir da Equação (6.54),

$$\mathbf{u} = \mathbf{N}\mathbf{Q}^{-1}\,\mathbf{V}. \tag{6.59}$$

Na equação acima, a matriz $\mathbf{N}\mathbf{Q}^{-1}$ é a matriz de funções de forma do elemento descontínuo. Para os pontos j definidos na Figura 6.5 (denotados pelo símbolo $\bullet$), temos

$$\mathbf{Q}_4 = \frac{1}{16}\begin{bmatrix} 9 & 3 & 1 & 3 \\ 3 & 9 & 3 & 1 \\ 1 & 3 & 9 & 3 \\ 3 & 1 & 3 & 9 \end{bmatrix}, \tag{6.60}$$

cuja inversa é

$$\mathbf{Q}_4^{-1} = \frac{1}{4}\begin{bmatrix} 9 & -3 & 1 & -3 \\ -3 & 9 & -3 & 1 \\ 1 & -3 & 9 & -3 \\ -3 & 1 & -3 & 9 \end{bmatrix}. \tag{6.61}$$

6.3.2 Elemento Quadrilateral Quadrático

O elemento quadrilateral quadrático é definido por 8 nós, como mostrado na Figura 3.16. Os deslocamentos e forças de superfície sobre o elemento são interpolados como

$$\begin{aligned} \mathbf{u} &= \mathbf{N}\,\mathbf{U} = \sum_{i=1}^{8} N_i\,\mathbf{u}_i, \\ \mathbf{p} &= \mathbf{N}\,\mathbf{P} = \sum_{i=1}^{8} N_i\,\mathbf{p}_i, \end{aligned} \tag{6.62}$$

onde N_i são as funções de forma definidas pelas Equações (3.19).

Elemento Descontínuo

O elemento quadrilateral quadrático descontínuo é mostrado na Figura 6.6. Adotando o mesmo procedimento do elemento quadrilateral linear descontínuo, obtemos

$$\mathbf{Q}_8 = \frac{1}{6}\begin{bmatrix} 25/18 & -5/6 & -7/18 & -5/6 & 25/9 & 5/9 & 5/9 & 25/9 \\ -5/6 & 25/18 & -5/6 & -7/18 & 25/9 & 25/9 & 5/9 & 5/9 \\ -7/18 & -5/6 & 25/18 & -5/6 & 5/9 & 25/9 & 25/9 & 5/9 \\ -5/6 & -7/18 & -5/6 & 25/18 & 5/9 & 5/9 & 25/9 & 25/9 \\ -5/6 & -5/6 & -5/6 & -5/6 & 5 & 5/3 & 1 & 5/3 \\ -5/6 & -5/6 & -5/6 & -5/6 & 5/3 & 5 & 5/3 & 1 \\ -5/6 & -5/6 & -5/6 & -5/6 & 1 & 5/3 & 5 & 5/3 \\ -5/6 & -5/6 & -5/6 & -5/6 & 5/3 & 1 & 5/3 & 5 \end{bmatrix}, \tag{6.63}$$

cuja inversa é

$$\mathbf{Q}_8^{-1} = \frac{1}{4}\begin{bmatrix} 25/2 & 5/4 & -1 & 5/4 & -25/4 & 5/4 & 5/4 & -25/4 \\ 5/4 & 25/2 & 5/4 & -1 & -25/4 & -25/4 & 5/4 & 5/4 \\ -1 & 5/4 & 25/2 & 5/4 & 5/4 & -25/4 & -25/4 & 5/4 \\ 5/4 & -1 & 5/4 & 25/2 & 5/4 & 5/4 & -25/4 & -25/4 \\ 5/4 & 5/4 & 5/4 & 5/4 & 5 & -5/2 & -1 & -5/2 \\ 5/4 & 5/4 & 5/4 & 5/4 & -5/2 & 5 & -5/2 & -1 \\ 5/4 & 5/4 & 5/4 & 5/4 & -1 & -5/2 & 5 & -5/2 \\ 5/4 & 5/4 & 5/4 & 5/4 & -5/2 & -1 & -5/2 & 5 \end{bmatrix}. \tag{6.64}$$

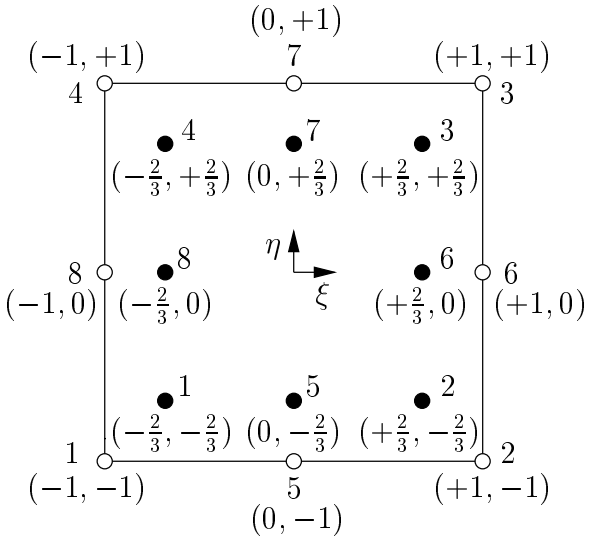


Figura 6.6: Elemento quadrilateral quadrático descontínuo.

6.4 Geração de Malhas

No Capítulo 3 descrevemos duas classes de modelos que utilizamos na representação geométrica de estruturas: modelos de cascas e modelos de sólidos. Nessa Seção trataremos da transformação de um modelo geométrico de cascas ou de sólidos em um modelo de decomposição por células que representa a malha de elementos finitos ou de contorno de um modelo mecânico. No Capítulo 3 definimos a geometria dos modelos de decomposição por células; nesse Capítulo estenderemos a definição com a inclusão de dados mecânicos no modelo, como mostrado no Programa 6.1.

Note, no programa, que acrescentamos a um vértice `cVertex` um identificador `Number` e um vetor `DOFs` de `NumberOfDOFs` graus de liberdade. Um grau de liberdade genérico é representado pela estrutura `tDOF`. `tDOF` mantém o valor `u` incógnito ou prescrito e o valor `p` corresponente, prescrito ou incógnito (por exemplo, θ_x e M_x), bem como o número `EquationNumber` da equação do sistema linear associada ao grau de liberdade e o tipo `BCType` de condição de contorno (essencial ou natural.) À estrutura `cCell` acresentamos o identificador `Number` e um ponteiro `Material` para as características mecânicas do material (contínuo, homogêneo, isótropo, etc.) da célula, representado pela estrutura `tMaterial`. Podemos voltar, agora, à geração de malhas.

Há uma variedade de algoritmos de geração de malhas para domínios bidimensionais e tridimensionais, estruturadas e não estruturadas. THACKER [114] apresenta uma revisão bibliográfica de algumas técnicas de geração de malhas irregulares publicadas até 1980. Mais recentemente, só para citar alguns exemplos, LÖHNER e PARIKH [68] mostram um método de geração de malhas tridimensionais baseado na técnica de *avanço de fronte*; BURATYNSKI [19] descreve uma gerador de malhas inspirado em *octrees*; e SHEPARD e GEORGES [103] propõem um algoritmo de geração de malhas tridimensionais baseado em *octrees* finitas. Nosso gerador de malhas foi desenvolvido a partir das seguintes considerações:

- Os modelos mecânicos são utilizados para análise de estruturas constituídas de cascas delgadas pelo método dos elementos finitos e de sólidos pelo método dos elementos de contorno. As malhas são formadas de elementos triangulares ou quadrilaterais definidos na Seção 6.2 e Seção 6.3.

```
struct tDOF
{
  double u;
  double p;
  int    EquationNumber;
  int    BCType;
}; // tDOF

struct cVertex
{
  t3DVector   Position;
  cVertexUse* Uses;
  cVertex*    Next;
  cVertex*    Previous;
  int         Number;
  int         NumberOfDOFs;
  tDOF*       DOFs;
}; // cVertex

struct tMaterial
{
  double E;
  double G;
  double Poisson;
}; // tMaterial

struct cCell
{
  cCellType*    Type;
  int           NumberOfVertices;
  cVertexArray* Vertices;
  cCell*        Next;
  cCell*        Previous;
  int           Number;
  tMaterial*    Material;
}; // cCell
```

Programa 6.1: Definição de vértice e célula de um modelo mecânico.

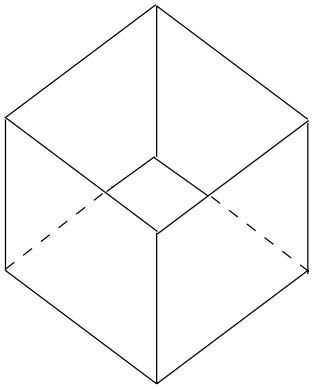


Figura 6.7: Modelo geométrico de uma casca.

- Os modelos geométricos a partir dos quais as malhas serão geradas são definidos por faces planares. *Geometricamente*, os elementos finitos e de contorno também serão planares.

Tomaremos como exemplo para ilustrar os passos do algoritmo a casca mostrada na Figura 6.7. (O processo se aplica igualmente no caso de modelos de sólidos.)

O algoritmo de geração de malhas tem como entrada o modelo geométrico da casca e certos parâmetros que definem o aspecto da malha gerada. Consideraremos, por ora, somente dois parâmetros: o tamanho d do elemento e o tipo de elemento, triangular (3 nós) ou quadrilateral (4 nós). O tamanho d do elemento é o comprimento esperado de qualquer lado de qualquer elemento da malha. A saída do algoritmo é um modelo de decomposição por células, a malha de elementos do modelo mecânico. Antes de descrevermos o algoritmo, vamos definir um modelo auxiliar chamado *modelo de contornos de faces*. Um modelo de contornos de faces, como mostrado no diagrama de objetos da Figura 6.8, é uma coleção de *contornos de faces*. A implementação C do diagrama é apresentada no Programa 6.2. Escolhemos, uma vez mais, implementar a coleção de contornos de faces do modelo com uma lista ligada duplamente encadeada. Vejamos cada um dos componentes do modelo.

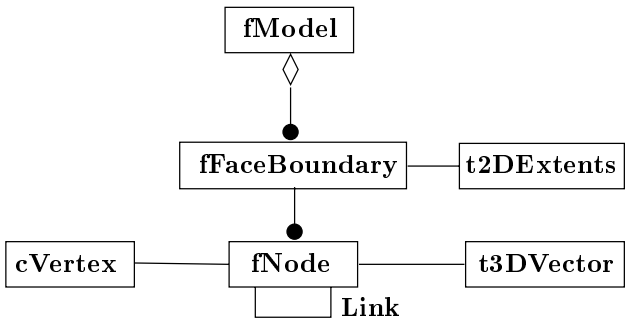


Figura 6.8: Diagrama de objetos: modelo de contornos de faces.

Contorno de face. Um contorno de face é uma seqüência cíclica de *vértices de contorno* que definem o contorno de um trecho de superfície planar, similar a um laço de uma face de um modelo de cascas ou de um modelo de sólidos. Porém, diferentemente de um modelo de cascas ou de sólidos, exigiremos que a superfície representada por um contorno de face seja convexa e simplesmente conectada.

```
struct t2DExtents
{
  t3DVector P1;
  t3DVector P2;
}; // t2DExtents

struct fNode
{
  fNode*    Link;
  cVertex*  MeshNode;
  t3DVector Position;
  fNode*    Next;
  fNode*    Previous;
}; // fNode

struct fFaceBoundary
{
  fNode*         FirstNode;
  int            NumberOfNodes;
  t2DExtents     Extents;
  fFaceBoundary* Next;
  fFaceBoundary* Previous;
}; // fFaceBoundary

struct fModel
{
  fFaceBoundary* Faces;
}; // fModel
```

Programa 6.2: Definição de modelo de contornos de faces.

A estrutura `fBoundaryFace` contém um ponteiro `FirstNode` para o primeiro vértice de contorno, um contador `NumberOfNodes` do número de vértices do contorno da face, uma estrutura `Extents` que define as coordenadas do retângulo envolvente do contorno, em relação a um sistema local de coordenadas, e os ponteiros para os elementos posterior e anterior na lista de contornos de faces do modelo.

Vértice de contorno. Um vértice de contorno representa um ponto de um contorno de face de um modelo de contornos de faces. Dois vértices de contorno consecutivos definem um segmento de um contorno de face. Os campos da estrutura `fNode` serão explicados na descrição do algoritmo de geração de malhas.

Os passos do algoritmo de discretização da casca da Figura 6.7 são descritos a seguir.

Passo 1 **Geração do modelo de contornos de faces**. Cada aresta do modelo de cascas é subdividida tal que, para quaisquer dois pontos consecutivos com coordenadas P_i e P_j sobre qualquer aresta de qualquer face da casca tenhamos

$$\| P_j - P_i \| = d, \tag{6.65}$$

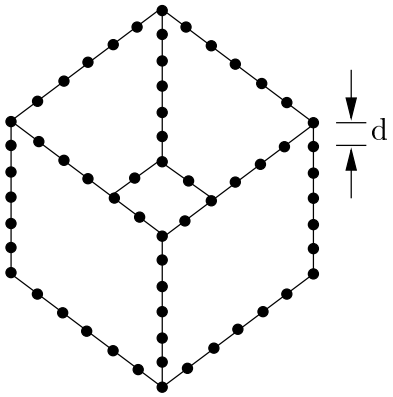


Figura 6.9: Modelo de contornos de faces intermediário.

onde d é o tamanho do elemento definido anteriormente. A subdivisão das arestas não implica em modificações no modelo de cascas original; ao invés disso, usaremos um modelo de contorno de faces para armazenar as informações resultantes desse passo. Um modelo de contorno de faces, portanto, é um modelo intermediário entre o modelo de cascas (ou de sólidos) de entrada e o modelo de decomposição por células de saída. A Figura 6.9 mostra o modelo de contornos de faces para a casca da Figura 6.7.

Em cada aresta e do modelo de cascas, o número n_e de pontos de subdivisão é dado por

$$n_e = \text{nint}\left(\frac{\| P_{e2} - P_{e1} \|}{d}\right), \tag{6.66}$$

onde nint é a função inteiro mais próximo e P_{e1} e P_{e2} são as coordenadas dos vértices da aresta e. Podemos ajustar o tamanho de cada segmento para a aresta e em função do número de pontos da subdivisão. Temos, então,

$$\mathbf{d}_e = \frac{P_{e2} - P_{e1}}{n_e}. \tag{6.67}$$

As coordenadas P_{ei} de um ponto i sobre a aresta e (armazenadas no vetor `Position` da estrutura `fNode` do Programa 6.2) são definidas parametricamente como

$$P_{ei} = P_1 + i\,\mathbf{d}_e, \qquad 0 \leq i \leq n_e. \tag{6.68}$$

A construção do modelo de contornos de faces de uma casca é executada pela função

```
fModel* bSplitEdges(bModel* shell, double size);
```

a qual toma como argumentos um ponteiro `shell` para uma casca e o tamanho `size` do elemento e cria o modelo de contornos de faces para a casca. Não discutiremos maiores detalhes sobre `bSplitEdges()` aqui, mas sua implementação foi bastante simplificada por causa da representação explícita das relações de adjacência dos elementos de um modelo de cascas, definida no Capítulo 3. Similarmente temos, para um modelo de sólidos, a função

```
fModel* bSplitEdges(sModel* solid, double size);
```

a qual toma como argumentos um ponteiro `solid` para um sólido e o tamanho `size` do elemento e cria o modelo de contornos de faces para o sólido. (Veja as classes `tShell` e `tSolid` no Capítulo 10.)

Passo 2 **Criação da malha de elementos**. Após a geração do modelo de contornos de faces intermediário, criamos um novo modelo de decomposição por células (cuja estrutura de dados foi definida no Capítulo 3), inicialmente vazio. Para cada vértice de contorno do modelo de contornos de faces (estrutura `fNode`), criamos um vértice correspondente no modelo de decomposição por células (estrutura `cVertex`). O endereço do vértice criado é armazenado no ponteiro `MeshNode` da estrutura `fNode`. Nesse ponto, temos uma malha com número de nós igual ao número de nós do modelo intermediário.

Passo 3 **Discretização dos contornos de faces**. Para cada contorno de face B do modelo de contornos de faces intermediário, executaremos os passos 4 a 7 a seguir. Esses passos são responsáveis pela discretização de um contorno de face B em elementos triangulares ou quadrilaterais. O algoritmo utilizado é uma extensão, para o caso tridimensional, do algoritmo de geração de malhas de forma livre para regiões planares de SEZER e ZEID [102].

Passo 4 **Rotação do contorno de face para o sistema local**. Um sistema de coordenadas locais é definido para o contorno de face B. Consideraremos um dos eixos do sistema local como sendo definido por um vetor unitário $\mathbf{u}_1$ tomado no sentido do maior lado de B (antes da subdivisão das arestas). Um segundo eixo $\mathbf{u}_3$ é dado pela normal $\mathbf{n}$ de B. O versor que define o terceiro eixo é obtido diretamente do produto vetorial ($\mathbf{u}_3$ e $\mathbf{u}_1$ são versores)

$$\mathbf{u}_2 = \mathbf{u}_3 \times \mathbf{u}_1. \tag{6.69}$$

Com os componentes de $\mathbf{u}_1$, $\mathbf{u}_2$ e $\mathbf{u}_3$ podemos montar a matriz de rotação de eixos $\mathbf{R}_c$, Equação (3.16). Com a matriz $\mathbf{R}_c$, levamos as coordenadas `Position` de cada vértice de contorno de B para o sistema local de coordenadas. Nesse ponto, temos o contorno de face B paralelo ao plano xy do sistema de coordenadas globais, como ilustrado na Figura 6.10.

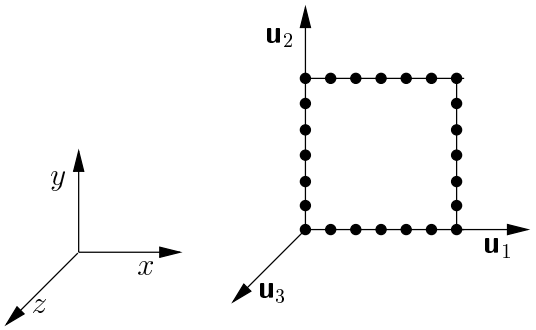


Figura 6.10: Contorno de face paralela ao plano xy.

.

Passo 5 **Geração do contorno de face interno**. As coordenadas P_{r1} e P_{r2} do retângulo envolvente `Extents` de B são determinadas. Em seguida, traçamos n_r "raios" paralelos ao eixo x, como mostrado na Figura 6.11, onde

$$n_r = \text{nint}\left(\frac{P_{r2_y} - P_{r1_y}}{d}\right). \tag{6.70}$$

A distância d_r entre cada "raio" é dada por

$$d_r = \frac{P_{r2_y} - P_{r1_y}}{n_r}. \tag{6.71}$$

Portanto, a equação de um raio r é

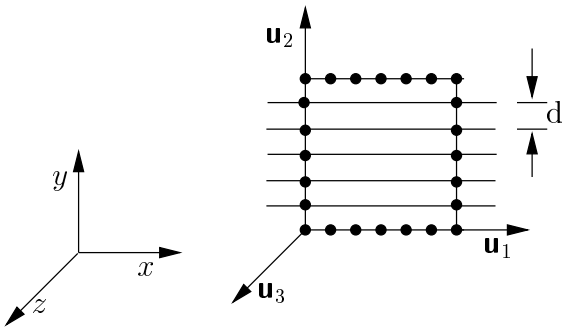


Figura 6.11: Traçado de "raios" sobre um contorno de face.

$$y_r = P_{r1_y} + r\, d_r, \qquad 0 \leq r \leq n_r. \tag{6.72}$$

Para cada raio r, $r \neq 0$ e $r \neq n_r$, determinamos os pontos de intersecção P_{ra} e P_{rb} de r com o contorno de face B. As coordenadas (x, y) do ponto de intersecção do raio r com um segmento definido por dois vértices de contorno consecutivos de B são dadas por

$$x = P_{1_x} + \frac{(P_{2_x} - P_{1_x})(r_y - P_{1_y})}{(P_{2_y} - P_{1_y})}, \quad y = r_y, \tag{6.73}$$

onde P_1 e P_2 são as coordenadas dos vértices consecutivos. O ponto de intersecção está sobre o segmento se $(P_{2_y} - r_y)(P_{1_y} - r_y) \leq 0$. (Sempre teremos dois pontos porque estamos admitindo somente superfícies conexas simplesmente conectadas.)

Uma vez determinados os pontos P_{ra} e P_{rb}, *deslocamos* esses pontos para o interior do contorno de face B, sobre o "raio" r e de uma distância d, como mostrado na Figura 6.12. Em seguida, para cada ponto "deslocado" D, verificamos se a distância entre D e qualquer vértice do contorno de face B não é menor que αd, onde $0 < \alpha \leq 1$ (SEZER e ZEID sugerem $\alpha = 0.5$). Se a distância for menor que αd, o ponto D é desconsiderado. Em seguida, *criamos um novo contorno de face* com todos os pontos D remanescentes desse processo de filtragem. Chamaremos esse novo contorno de face de O e subdiviremos cada aresta formada por dois vértices de contorno consecutivos de O como no passo 1. Repare que agora temos dois contornos de faces:

o contorno externo B e o contorno "deslocado" O. Para cada vértice de O criamos um vértice correspondente no modelo de decomposição de célula e armazenamos o endereço do vértice criado em `MeshNode`, como anteriormente. (As coordenadas de `MeshNode` são transformadas para o sistema global de coordenadas.) O resultado é ilustrado na Figura 6.12.

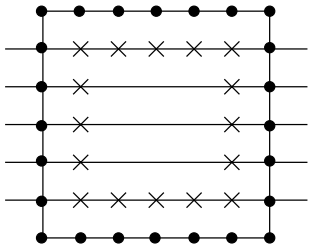
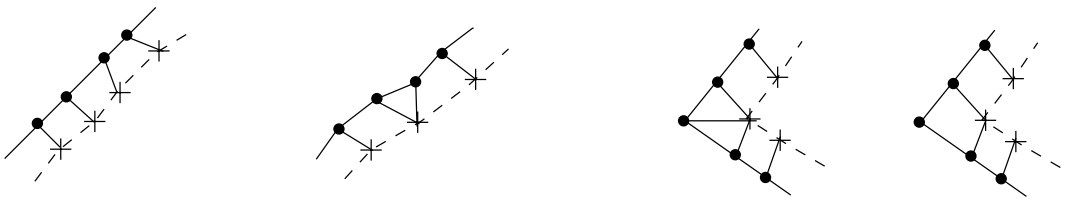


Figura 6.12: Novo contorno de face interno.

Passo 6 **Geração de células**. Uma nova célula da malha é formada pela conexão dos vértices do contorno de face externo B e do contorno de face interna O. Inicialmente, determinamos, para cada vértice V_B de B, qual é o vértice V_O de O mais próximo de V_B. Uma vez determinado V_O, ajustamos o ponteiro `Link` de V_B com o endereço de V_O. O ponteiro `Link` define, portanto, o vértice V_O de conexão do vértice V_B. A geração de elementos é definida por três regras, ilustradas na Figura 6.13.



(a) Regra dois/dois (b) Regra dois/um (c) Regra três/um

Figura 6.13: Regras de geração de células.

1. Regra dois/dois: dois vértices consecutivos do contorno externo B são conectados a dois vértices consecutivos de contorno interno O para formar um elemento quadrilateral.
2. Regra dois/um: dois vértices consecutivos do contorno externo B são conectados a um único vértice do contorno interno O para formar um elemento triangular.
3. Regra três/um: três vértices do contorno externo B são conectados a um único vértice do contorno interno O. Nesse caso, a conexão entre o segundo vértice de B e o vértice de O pode ser eliminada para formar um elemento quadrilateral ou ser mantida para formar dois elementos triangulares. O critério de eliminação é estabelecido por SEZER e ZEID em função dos ângulos internos do (possível) elemento quadrilateral, considerados aceitáveis se estiverem entre $90^o \pm 20^o$.

A lista de incidência dos elementos gerados por uma das três regras acima é definida pelo ponteiro `MeshNode` de cada vértice de contorno envolvido. Cada novo elemento gerado é, então, adiciondo à lista de elementos da malha (ou, equivalentemente, cada célula é adicionda à lista de células do modelo de decomposicão por células), como ilustrado na Figura 6.14.

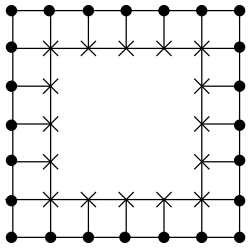


Figura 6.14: Geração de células.

Passo 7 **Geração interativa de vértices e células**. Concluída a geração dos elementos da "lâmina" definida pelos contornos de faces B e O, consideramos o contorno de face interna O como sendo o contorno de face externa B e repetimos as passos 5 e 6, desde que o número de vértices `NumberOfVertices` de O seja maior que quatro. Se o número de vértices de O for menor ou igual a 4, interrompemos o processo interativo. Se restarem quatro vértices em O, geramos um último elemento quadrilateral. Se restarem três vértives em O, geramos um último elemento triangular. Passamos, então, para o processamento do próximo contorno de face do modelo intermediário.

A malha resultante do algoritmo descrito acima pode conter elementos triangulares e quadrilaterais. Se quisermos uma malha somente com triângulos, temos que criar, para cada célula quadrilateral do modelo, se houver, duas células triangulares e, posteriormente, destruir a célula quadrilateral. (A célula quadrilateral é divida na diagonal menor.) Se quisermos uma malha somente com quadriláteros, temos que criar, para cada célula triangular ou quadrilateral do modelo, quatro células quadrilaterais, conforme mostrado na Figura 6.15. Posteriormente, a célula triangular ou quadrilateral é destruída. Note, na figura, que o tamanho do elemento fica, como conseqüência da subdivisão, dividido por dois. O gerador de malhas, nesse caso, multiplica automaticamente por dois o tamanho de elemento d antes de proceder à geração dos vértices. A Figura 6.16 mostra uma malha de quadriláteros para a casca da Figura 6.7.

(a) Elementos triangulares (b) Elementos quadrilaterais

Figura 6.15: Subdivisão de uma célula em quadriláteros.

A parte final do processo de geração de malhas é a *suavização* da malha. A suavização, ou *smoothing*, consiste no reposicionamento dos vértices "interiores" do modelo

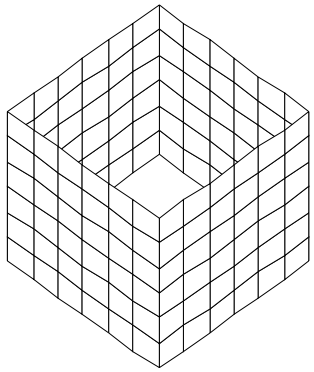


Figura 6.16: Malha de quadriláteros da casca.

de decomposição por células de acordo com determinado critério de suavização. (Estamos chamando de vértices "interiores" os vértices que não são resultantes da subdivisão de uma aresta do modelo geométrico original.) Adotamos como critério de suavização o método do centróide proposto por SEZER e ZEID, resumido a seguir.

Seja V_I um vértice "interior" da malha no qual incidem m células. Para cada célula k incidente em V_I determinamos as coordenadas do centróide P_{ck} e a área A_k da célula. (A determinação do centróide e da área de triângulos e quadriláteros não apresenta quaisquer dificuldades.) As novas coordenadas do vértice V_I são as coordenadas do centróide combinado de todas as m células incidindo em V_I, dadas pela expressão

$$P_c = \left(\sum_{k=1}^{m} P_{ck}\, A_k \right) / \left(\sum_{k=1}^{m} A_k \right) . \tag{6.74}$$

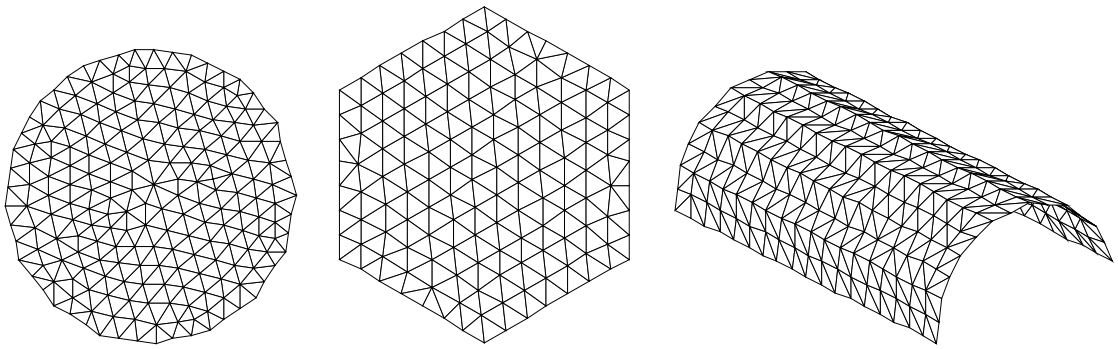
O método pode ser implementado facilmente se soubermos quais são as células que incidem em um vértice do modelo. Em nossos modelos de decomposição por células, essa informação é armazenada diretamente em uma estrutura do tipo `cVertexUse`, tal como definimos no Capítulo 3.

Alguns exemplos do processo de geração de malhas descrito nessa Seção são mostrados na Figura 6.17, para superfícies convexas simplesmente conectadas (SEZER e ZEID descrevem com detalhes como transformar uma região côncava em regiões convexas e uma região multiplamente conectada em regiões simplesmente conectadas.) Alternativamente, usaremos também os métodos de varredura discutidos no Capítulo 3 para os casos onde esses métodos forem mais apropriados que a técnica de avanço de fronte, por exemplo, para a geração de uma malha sobre a superfície de uma esfera.

6.5 Sumário

Nesse Capítulo apresentamos as formulações dos elementos finitos e de contorno empregados na modelagem mecânica das estruturas analisadas no trabalho. As hipóteses admitidas no Capítulo 4 continuam válidas, ou seja, consideramos somente estruturas constituídas de materiais contínuos, homogêneos, isótropos e elástico lineares.

O elemento finito de casca utilizado é uma composição do elemento finito de placa DKT e do elemento de membrana com liberdades rotacionais derivado da formulação livre. O elemento de casca possui 3 nós e 18 graus de liberdade. Em cada nó, temos 3 rotações e 3 deslocamentos e, em correspondência, 3 momentos e 3 forças.



(a) Casca circular (b) Casca hexagonal (c) Casca cilíndrica

Figura 6.17: Exemplos de geração de malhas simples.

Os elementos de contorno considerados são quadrilaterias isoparamétricos com 4 e 8 nós. Apresentamos, também, as funções de forma para elementos descontínuos, úteis no tratamento de descontinuidades de forças de superfície em pontos de canto.

Os modelos mecânicos são geometricamente representados por malhas de elementos triangulares e quadrilaterais geradas a partir do modelo geométrico da estrutura. Nosso algoritmo de geração de malhas é uma extensão, para o caso tridimensional, do algoritmo de geração de malhas de forma livre de regiões planares de SEZER e ZEID.

CAPÍTULO 7

Visualização

7.1 Introdução

No Capítulo 1 dissemos que os métodos computacionais de solução do problema fundamental, dependendo da complexidade da estrutura, podem produzir extensas listagens numéricas que impedem ou até mesmo impossibilitam uma compreensão mais imediata dos resultados da análise. As distribuições dos campos de deslocamentos, deformações ou tensões de uma estrutura podem ser mais facilmente compreendidas através de *imagens* que representam essas distribuições, tais como mapas de cores ou isolinhas, por exemplo. Além disso, os objetos de engenharia possuem uma estrutura muito bem definida geometricamente, o que sugere naturalmente o emprego da computação gráfica para *visualização* dessa estrutura.

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. Nos capítulos anteriores vimos algumas formas de representação de informações de um objeto estrutural. No Capítulo 3 definimos os modelos geométricos de cascas e de sólidos, cujas informações serão utilizadas para sintetizar imagens da estrutura do objeto. No Capítulo 6 estendemos a representação de informações geométricas do objeto com atributos caracteristicamente mecânicos, tais como propriedades do material, forças e deslocamentos, os quais serão utilizados para sintetizar imagens do comportamento do objeto.

Nesse Capítulo discutiremos os algoritmos de transformações de dados de um modelo estrutural em primitivos gráficos e, posteriormente, em imagens de computador. Começaremos, na Seção 7.2, definindo quais são os processos de transformação de dados de modelos em primitivos gráficos. Veremos o que são *fontes*, *filtros* e *mapeadores* e como esses objetos podem ser empregados na construção de um modelo de visualização.

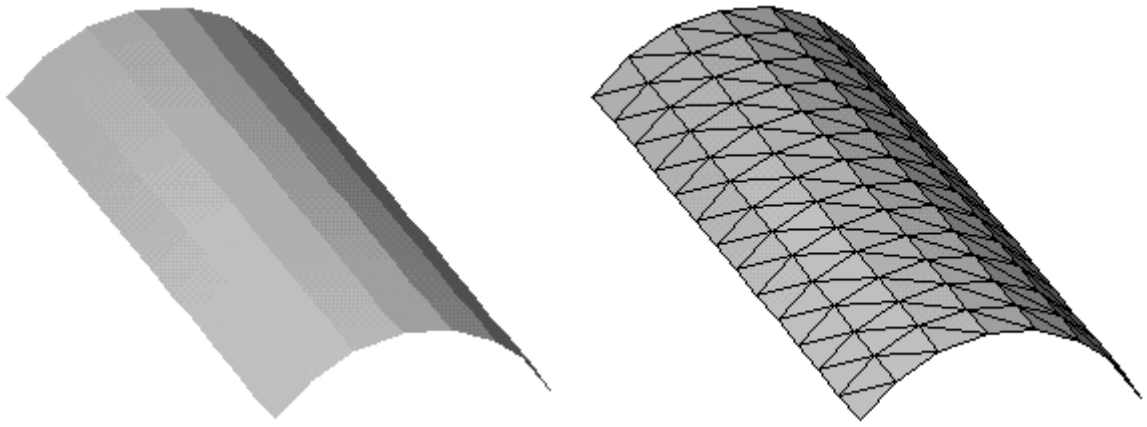
A transformação de primitivos gráficos em imagens de computador é objeto de estudo da *computação gráfica*, a fundação da visualização. Na Seção 7.3 introduziremos

alguns conceitos de computação gráfica necessários à leitura das classes do Capítulo 10. Infelizmente, não descreveremos todos os métodos compugráficos implementados nas classes de OSW porque a descrição desses métodos poderia ser extensa o bastante para ocupar um outro volume. Definiremos uma *cena* como sendo constituída de *atores*, *luzes* e *câmeras* e veremos os princípios elementares de síntese de imagens de uma cena.

Na Seção 7.4 descreveremos os algoritmos empregados na visualização de campos escalares e vetoriais de nossos modelos estruturais.

7.2 Da Modelagem à Visualização

Para ilustrarmos essa Seção, tomemos como exemplo a análise de uma casca cilíndrica pelo método dos elementos finitos. Inicialmente, vamos criar um modelo geométrico para a casca. Utilizaremos, nesse caso, um modelo de cascas tal como visto no Capítulo 3. A Figura 7.1(a) mostra uma imagem com "superfícies escondidas" do modelo geométrico da casca. Em seguida, precisamos criar uma malha de elementos finitos para a análise. A Figura 7.1(b) ilustra uma discretização do modelo geométrico de acordo com o método de geração de malhas descrito no Capítulo 6. Note que temos dois modelos na Figura 7.1: um modelo de cascas e um modelo de decomposição por células. Embora um modelo de decomposição por células *seja* um modelo geométrico, escolhemos representar a geometria da estrutura por um modelo de cascas e *derivar* a geometria do modelo mecânico a partir do modelo de cascas. Com isso, podemos utilizar o modelo de cascas original para derivar malhas distintas de elementos e, posteriormente, comparar os resultados de análise dessas malhas. (Se quiséssemos, poderíamos criar diretamente o modelo de decomposição por células da casca, por exemplo, através do processo de varredura translacional de um arco de circunferência).

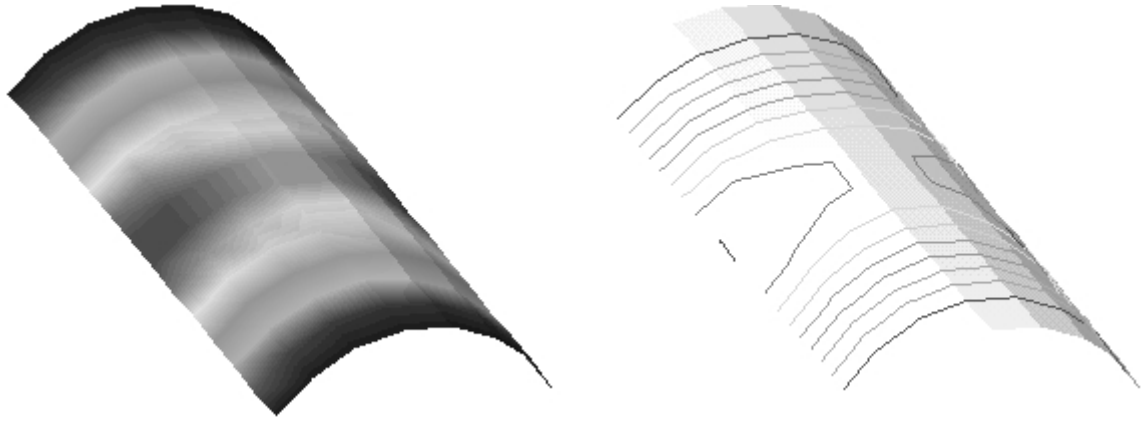


(a) Modelo geométrico (b) Modelo mecânico

Figura 7.1: Modelos geométrico e mecânico de uma casca.

As imagens da Figura 7.1 são exemplos de visualização da *estrutura* de um modelo e, embora simples, servem para demonstrar a utilidade da visualização. A corretude do modelo de decomposição por células da casca pode ser muito mais facilmente percebida pela observação direta da imagem do modelo do que pela conferência da listagem numérica dos 112 vértices e 182 células, resultantes do processo de geração da malha.

Após a descrição das condições de contorno e dos carregamentos, o modelo mecânico da casca é analisado pelo método dos elementos finitos. Dois exemplos de visualização dos resultados da análise são mostrados na Figura 7.2. A Figura 7.2(a) mostra o *mapa de cores* correspondente à distribuição de um campo escalar sobre a superfície da casca. (Note que o problema fundamental é definido em termos de vetores e tensores, não de escalares. Nesse caso, consideramos o escalar como sendo um componente de um vetor ou tensor, por exemplo, o deslocamento na direção y.) A Figura 7.2(b) mostra algumas isolinhas do campo escalar. As imagens ilustram a visualização do *comportamento* de um modelo. Mais uma vez, podemos utilizar a visualização para perceber mais imediatamente a distribuição dos resultados da análise da casca.



(a) Mapa de cores (b) Isolinhas

Figura 7.2: Exemplos de visualização de uma casca.

Introduziremos uma notação gráfica para descrever os passos de criação da visualização chamada *diagrama de fluxo de dados* (DFD). A Figura 7.3 mostra o diagrama de fluxo de dados para os exemplos de visualização da Figura 7.1 e da Figura 7.2. 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. Em OSW, um repositório de dados é um modelo geométrico ou mecânico de uma estrutura. Os retângulos em um DFD denotam *processos fontes* e *processos sumidouros* de dados. Um processo 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. Os retângulos com cantos arredendados denotam processos de transformação de repositórios de dados em outros repositórios de dados. As setas do DFD indicam a direção do fluxo de dados.

No DFD da Figura 7.3, temos um processo fonte rotulado com o nome `Varredura Translacional`. O processo toma como parâmetros de entrada uma curva e um caminho e produz como saída o modelo geométrico resultante do "escorregamento" da curva ao longo do caminho. (Note: o processo possui parâmetros de entrada, mas esses parâmetros não são repositórios de dados. Somente a saída de um processo fonte é um repositório de dados.) Chamaremos um processo fonte simplesmente de *fonte*. Se o fonte gerar repositórios de dados a partir de arquivos armazenados em memória secundária (disco rígido, por exemplo), chamaremos o fonte de *leitor*. Usamos

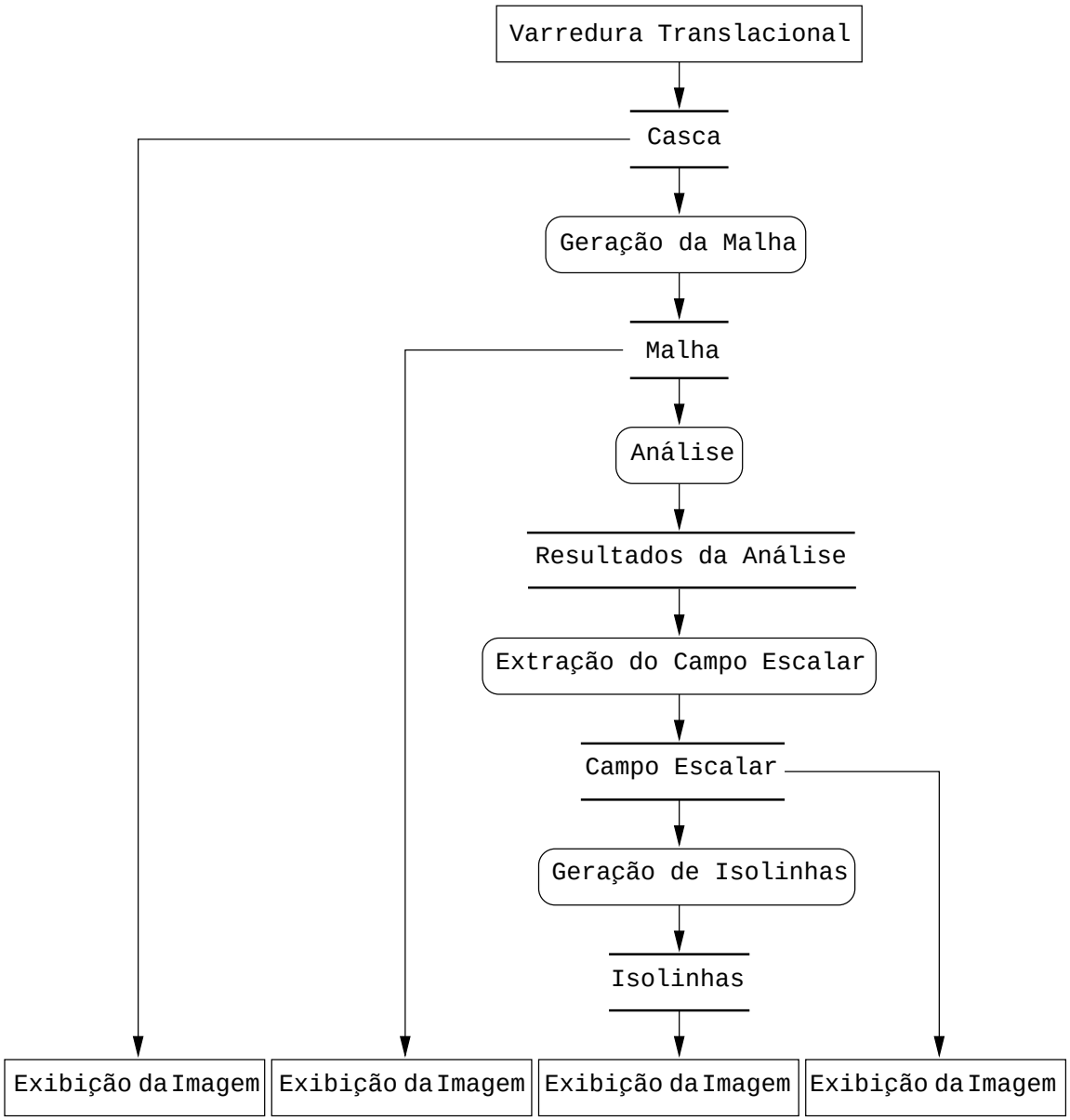


Figura 7.3: Diagrama de fluxo de dados.

o fonte `Varredura Translacional` para gerar o modelo de cascas da Figura 7.1(a), denotado, no DFD, pelo repositório de dados rotulado com o nome `Casca`.

Seguindo a direção do fluxo de dados, encontramos o processo de transformação de dados rotulado com o nome `Geração da Malha`. O processo toma como entrada o modelo geométrico da casca e parâmetros que controlam a aparência da malha (tipo de elemento e tamanho do elemento, conforme vimos no Capítulo 6). A saída do processo é o modelo de decomposição por células da Figura 7.1(b), denotado no diagrama pelo repositório de dados `Malha`. (Note: o processo toma como entrada, além dos parâmetros de aspecto da malha, um repositório de dados e transforma o respositório de entrada em um outro repositório de dados.) Chamaremos um processo de transformação de dados de *filtro*. O processo rotulado com o nome `Geração de Isolinhas` é outro exemplo de filtro. A entrada do processo é o modelo de decomposição por células `Campo Escalar` (é a própria `Malha`, após passar pelos processos de `Análise` e `Extração do Campo Escalar`) e os parâmetros que definem os valores

escalares das isolinhas; a saída é um modelo gráfico que contém primitivos `gLine` (veja o Capítulo 3) que representam as isolinhas, denotado, no DFD, pelo repositório de dados `Isolinhas`.

Finalizando o fluxo de dados do diagrama da Figura 7.3, temos os processos sumidouros rotulados com o nome `Exibição da Imagem`. Um processo `Exibição da Imagem` toma como entrada um modelo geométrico qualquer e produz como saída uma imagem do modelo. (Note: a saída do processo é uma imagem, não um outro repositório de dados.) Chamaremos um processo sumidouro de *mapeador*. No caso particular do processo sumidouro escrever os dados dos repositórios de dados de entrada em arquivos armazenados em memória secundária, chamaremos o processo sumidouro de *escritor*.

O fluxo de dados representado no DFD da Figura 7.3 é chamado de *rede de visualização* (*visualization pipeline*). SCHROEDER, MARTIN E LORENSEN [101] discutem detalhadamente as estratégias de implementação de redes de visualização. Nosso propósito foi somente introduzir os conceitos de fontes, filtros e mapeadores (daremos alguns exemplos na Seção 7.4.) No Capítulo 9 apresentaremos os principais processos de visualização de OSW e mostraremos como utilizá-los na contrução de uma aplicação orientada a objetos de modelagem estrutural.

7.3 Computação Gráfica Tridimensional

Os algoritmos de visualização transformam dados sobre a estrutura e o comportamento de um modelo em primitivos gráficos. Com a *computação gráfica*, transformamos esses primitivos gráficos em imagens. Objetivamente, computação gráfica é a *síntese* de imagens em computador.

Definiremos uma *imagem* como sendo uma coleção discreta de pontos coloridos denominados *pixels*, organizados em um arranjo retangular chamado *mapa de pixels*. (Achamos o termo mais adequado que *mapa de bits*, ou *bitmaps*, porque um *pixel*, geralmente, possui mais de um *bit*; no entanto, o uso do termo está tão consagrado que o utilizaremos como sinônimo de mapa de *pixels*.) Nosso problema consiste em determinar a cor correta de cada *pixel* de uma imagem.

Para descrevermos o processo de síntese de imagens, ou *rendering*, consideremos um ambiente definido por alguns objetos, uma fonte de luz e um observador, conforme o esquema ilustrado na Figura 7.4. Vamos admitir que a luz esteja apagada e que não haja quaisquer outras fontes luminosas no ambiente. O que o observador veria, nesse caso? Sem luz, absolutamente nada. Agora, acendamos a luz. A fonte luminosa passa a emitir, em todas as direções, partículas chamadas *fótons* (ou, dualmente, ondas de luz). Vamos seguir as trajetórias, ou os *raios*, de alguns desses fótons.

Observe, na figura, o raio A. Esse raio não intercepta objeto algum do ambiente e, portanto, não contribui para a formação da imagem do observador (representado pela câmera da figura). O raio B, por outro lado, intercepta um objeto do ambiente. De acordo com as leis da Física Ótica, no ponto de intersecção o objeto absorve uma parcela da energia luminosa do raio B, refrata outra parcela e reflete uma outra parte. A quantidade de energia resultante na absorção, refração e reflexão é determinada em função da energia do raio B e das propriedades materiais do objeto no ponto de intersecção. Vamos supor que o objeto seja absolutamente translúcido; nesse caso, não há refração alguma. Vamos supor, também, que a superfície do sólido seja polida no ponto de intersecção; nesse caso, uma parcela do raio B é refletida pelo objeto, conforme

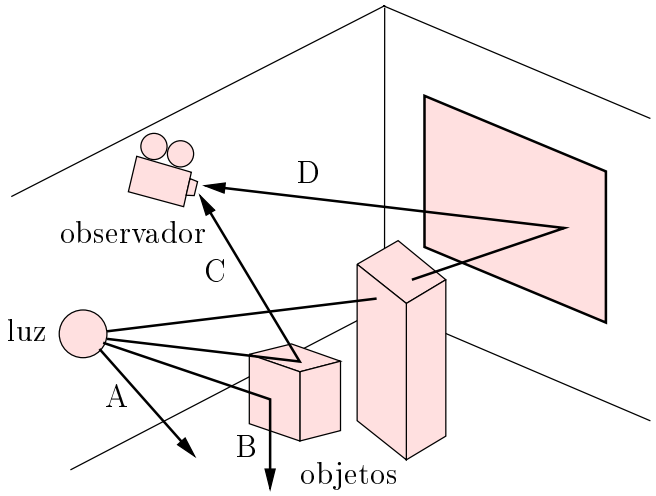


Figura 7.4: Ambiente: objetos, fonte de luz e observador.

mostrado na Figura 7.4. Note, no entanto, que o raio B, da mesma forma que o raio A, não atinge a posição do observador que, por enquanto, continua sem observar nada.

Vejamos, agora, o raio C. Como o raio B, C intercepta o mesmo objeto do ambiente e, como antes, é absorvido e refletido (estamos admitindo que o objeto seja translúcido). Porém, desta vez, o raio C atinge a posição do observador. O que o observador vê é a *cor* associada à energia do raio C, digamos, verde. Finalmente, luz. O raio D também atinge a posição do observador, mas percorre uma trajetória diferente: primeiro refrata em um objeto transparente e depois reflete em um objeto polido. Vamos supor que a cor associada à energia do raio D seja vermelha. Se os raios C e D atingirem a posição do observador no mesmo ponto, a cor percebida será correspondente à energia somada desses dois raios, nesse caso, amarela. A imagem vista pelo observador é formada pela contribuição de todos os raios luminosos que partem da fonte de luz e atingem o olho do observador, após interagirem com os objetos do ambiente.

Uma fonte de luz emite uma infinidade de raios em todas as direções. Dependendo das dimensões do ambiente considerado, apenas uns poucos raios (talvez alguns milhões) interceptam algum objeto e atingem o olho do observador, contribuindo para a formação da imagem. Esse processo de formação de imagens é denominado *traçado de raios progressivo* porque seguimos a trajetória dos raios de luz desde sua emissão na fonte até sua chegada no observador. Para tratar computacionalmente o problema, ao invés de seguirmos a trajetória do raio desde a fonte até o observador, faremos o caminho inverso: seguiremos o raio desde o observador até a fonte de luz. (Se o raio atingiu o observador é porque, evidentemente, partiu de alguma fonte de luz.) Com isso, estaremos considerando somente os raios de luz que efetivamente contribuem para a formação da imagem. Esse processo de *rendering* é denominado *traçado de raios regressivo*. Designaremos o traçado de raios regressivo simplesmente como traçado de raios, ou *ray tracing*.

O algoritmo de traçado de raios toma como entrada um ambiente definido por objetos, fontes de luz e um observador e produz como saída uma imagem com $m \times n$ *pixels*. Os passos do algoritmo de traçado de raios são sumarizados a seguir.

Passo 1 Para cada *pixel* (i, j) da imagem, $1 \leq i \leq m, 1 \leq j \leq n$, "traçamos" um raio r que parte da posição do observador, em direção ao centro do elemento de

área representado pelo *pixel* (i, j), conforme esquematizado na Figura 7.5. O raio r é chamado *raio de pixel*.

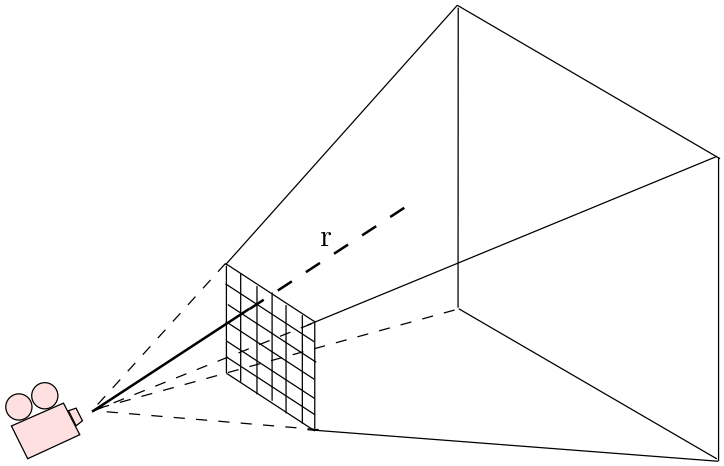


Figura 7.5: "Traçando" um raio de *pixel*.

Passo 2 Determinamos o objeto O mais próximo do observador que é interceptado pelo raio r. Se o raio não interceptar nenhum objeto, então o observador vê a *cor de fundo* do ambiente.

Passo 3 Para o objeto O, computamos o ponto de intersecção P do objeto com o raio r. Calculamos, então, a contribuição de todas as fontes de luz para a iluminação de P. O cálculo é baseado no "traçado" de raios do ponto P até cada uma das fontes de luz do ambiente, como ilustrado na Figura 7.6. Note que estamos seguindo a trajetória inversa de todos os raios luminosos que atingem o observador e que passam pelo *pixel* (i, j). Esses raios são chamados *raios de sombra*, ou *raios de iluminação*. A energia luminosa que incide em P é determinada por um *modelo de iluminação* que considera a interação de cada raio com o material do objeto em P (posteriormente definiremos um modelo de iluminação). Essa energia luminosa define a parcela da cor do *pixel* devida à iluminação direta das fontes de luz. Se os raios de sombra interceptarem algum objeto, então as fontes de luz não iluminam diretamente o ponto P.

Passo 4 Se o objeto O for reflexivo em P, "traçamos" um novo raio partindo de P na direção de reflexão. Chamamos esse raio de *raio de reflexão*. A cor do raio de reflexão, definida pela execução recursiva do algoritmo, é adicionada à cor determinada no passo anterior.

Passo 5 Se o objeto O for transparente em P, "traçamos" um novo raio partindo de P na direção de refração. Chamamos esse raio de *raio de transparência*. A cor do raio de transparência, também definida pela execução recursiva do algoritmo, é adicionada à cor determinada nos passos anteriores. A cor resultante é a cor do *pixel* (i, j).

Talvez possamos até ter compreendido as idéias do princípio do traçado de raio com o esboço dado acima, mas a descrição do algoritmo está muito distante de sua

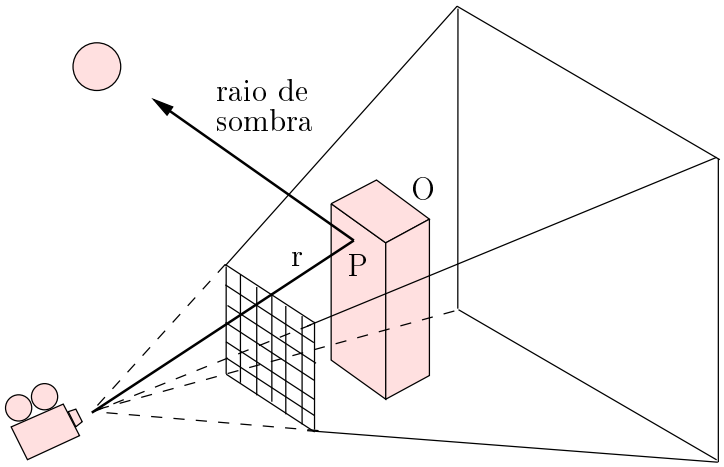


Figura 7.6: "Traçando" um raio de sombra.

implementação. Note a complexidade: precisamos traçar raios na direção correta, encontrar a intersecção dos raios com o objeto mais próximo do observador, determinar a cor no ponto de intersecção e, dependendo das características do material, "reflexivo" ou "transparente", traçar mais raios novamente. Conceitualmente, no entanto, o algoritmo de traçado de raios não é muito complicado. As imagens obtidas com o *ray tracing* podem ter "realidade de fotografia" (principalmente se os materiais dos objetos do ambiente forem reflexivos) porque os efeitos da iluminação direta, reflexão e refração são naturalmente tratados pelo método, diferentemente de outros processos de geração de imagens.

O traçado de raios possui seus problemas, contudo. O traçado de raios é relativamente lento se comparado a outras técnicas. Além disso, por ser um processo de amostragem de pontos, as imagens geradas apresentam o que chamamos de *aliasing* [38]. Dependendo dos propósitos da imagem, podemos empregar técnicas mais simples de *rendering*, tais como o algoritmo de *scanlines* com tonalização de Gouraud ou Phong [99]. Não entraremos em maiores detalhes a respeito de traçado de raios nesse Capítulo (veja a classe `tRayTracer` no Capítulo 10). GLASSNER [46] apresenta uma ótima introdução sobre a teoria e prática do *ray tracing*, incluindo algoritmos de intersecção e iluminação e técnicas de aceleração e *antialiasing*. Implementações em C podem ser encontradas em WATKINS, COY e FINLAY [124] e LINDLEY [66].

Cena

Chamaremos o ambiente de entrada do processo de *rendering* de *cena*. Denominaremos os objetos do ambiente de *atores* e definiremos as vistas do observador através de um conjunto de *câmeras* "virtuais". Uma cena, portanto, é uma coleção de atores, luzes e câmeras que definem um ambiente a partir do qual sintetizaremos uma imagem. A imagem é obtida pela determinação da cor de cada *pixel*, função da vista tomada por uma câmera e da interação da luz emitida pelas fontes de luz com os materiais que constituem as superfícies dos atores. O diagrama de objetos de uma cena é mostrado na Figura 7.7 e seus componentes comentados a seguir.

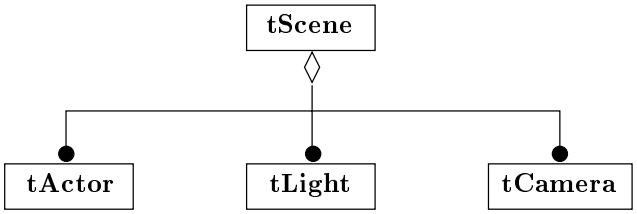


Figura 7.7: Diagrama de objetos: cena.

Modelos de Cores

Em computação gráfica, uma cor é especificada através de um *modelo de cores*. Usamos dois modelos de descrição de cores em OSW: o modelo RGB, empregado nos monitores de vídeo coloridos, e o modelo HSV.

O modelo RGB representa uma cor em termos das intensidades de vermelho, verde e azul da cor. Geometricamente, o modelo é definido em um sistema de coordenadas Cartesianas pelo cubo unitário ilustrado na Figura 7.8. O modelo RGB é *aditivo*, ou seja, o resultado da combinação de duas ou mais cores é obtido pela soma das contribuições individuais dos componentes de cada uma das cores. Por exemplo, a combinação da cor azul com a cor verde resulta na cor ciano. O Programa 7.1 apresenta a implementação de uma cor no modelo RGB. Utilizaremos a estrutura `tColor` para especificar as cores dos *pixels* de uma imagem.

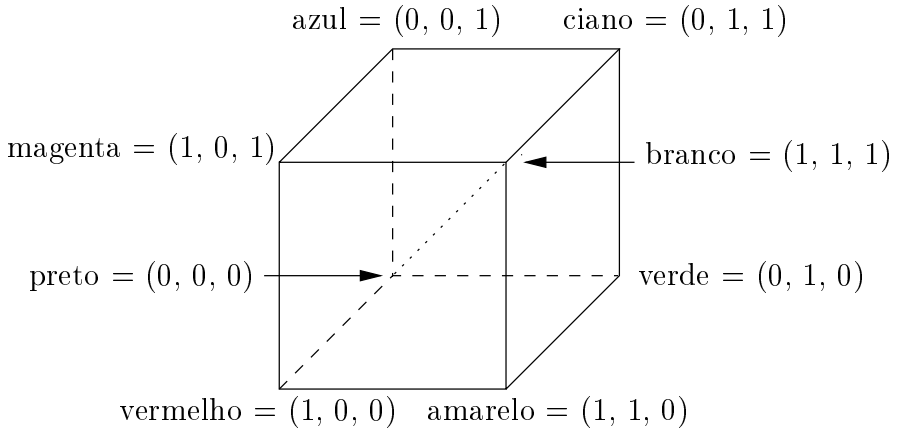


Figura 7.8: Modelo de cores RGB.

```
struct tColor
{
  double r;
  double g;
  double b;
}; // tColor
```

Programa 7.1: Definição de cor RGB.

O sistema HSV representa uma cor baseado na matiz, saturação e valor da cor (ou *hue*, *saturation* e *value*). O modelo HSV é geometricamente definido em um sistema de coordenadas cilíndricas por um pirâmide de base hexagonal, como ilustrado na

Figura 7.9. A matiz H da cor é medida pelo ângulo em torno do eixo vertical da pirâmide, com vermelho em 0^o, verde em 120^o, etc. A saturação S é a distância variável de 0, no centro da base da pirâmide, até 1, nas faces triangulares da pirâmide. O valor V é a altura variável de 0, no vértice da pirâmide, até 1, no centro da base. O ponto $V = 0$ é preto (nesse ponto, H e S são irrelevantes). O ponto $S = 0$, $V = 1$ é branco. Usaremos o modelo HSV para especificar as cores de um *mapa de cores* (veja a Seção 7.4). FOLEY [38, página 584] discute minuciosamente os modelos de cores e os algoritmos de conversão RGB-HSV e HSV-RGB.

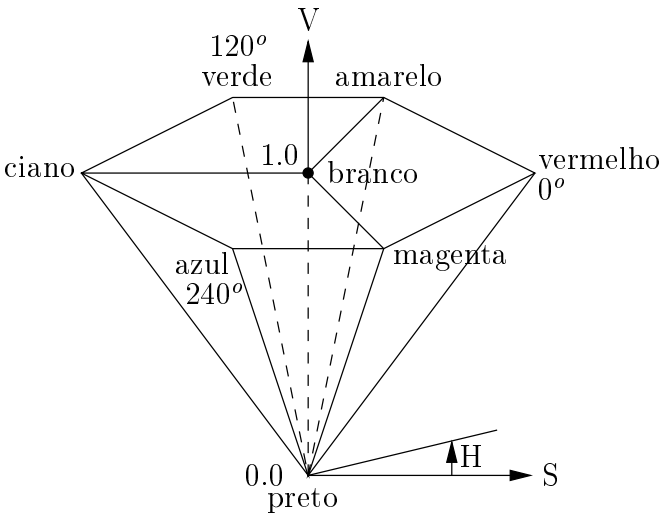


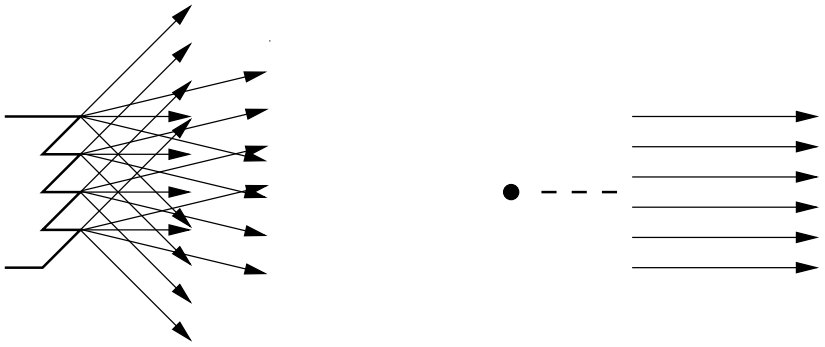
Figura 7.9: Modelo de cores HSV.

Luzes

Na descrição do processo de *rendering* apresentado anteriormente, consideramos uma fonte de luz emitindo fótons de diferentes intensidades em todas as direções, como ilustrado na Figura 7.10(a). Chamaremos esse tipo de fonte de luz de *luz puntual*. Usaremos luzes puntuais em uma cena para simular fontes de luz próximas aos objetos da cena. A Figura 7.10(b) mostra outro tipo de fonte de luz utilizada em computação gráfica chamada *luz direcional*. Uma fonte de luz direcional está localizada tão distante dos objetos da cena que consideramos que os raios de luz emitidos pela fonte chegam à cena paralelos entre si. Usaremos luzes direcionais para simular a luz do sol, por exemplo. (É claro que, em proporções astronômicas, o sol é uma fonte de luz puntual.)

O Programa 7.2 apresenta a definição C de uma fonte de luz de uma cena. Note que implementamos a coleção como uma lista ligada duplamente encadeada de luzes. A estrutura `tLight` mantém o tipo `Type` de luz (puntual ou direcional), a posição (ou direção) `Position`, a cor `Color`, a *flag* `Switch` que indica se a luz está ligada ou desligada (em nossas cenas, podemos desligar o "sol") e os ponteiros para os elementos posterior e anterior na lista de luzes da cena.

A interação da luz com a superfície de um objeto pode ser teoricamente dividida em quatro classes, chamadas *modos de transporte de luz*: *reflexão difusa*, *reflexão especular*, *transmissão difusa* e *transmissão especular*. Quando um raio de luz incide sobre uma superfície, sofre mudanças de direção e cor como resultado desses quatro efeitos. A influência de cada modo de transporte de luz na cor final do raio de luz é função,



(a) Puntual (b) Direcional

Figura 7.10: Tipos de luz de uma cena.

```
enum tLightType { lPUNCTUAL, lDIRECTIONAL };
struct tLight
{
   tLightType Type;
   t3DVector  Position;
   tColor     Color;
   bool       Switch;
   tLight*    Next;
   tLight*    Previuos;
}; // tLight
```

Programa 7.2: Definição de luz.

principalmente, das características materiais da superfície na qual o raio incide. Em computação gráfica, a determinação quantitativa da interação da luz com a superfície de um objeto é usualmente baseada em modelos matemáticos mais simplificados que os complexos modelos derivados da Física Ótica. Chamaremos esses modelos simplificados de *modelos de iluminação* e veremos, a seguir, as equações do modelo de iluminação utilizado em OSW, o modelo de iluminação de Phong [99].

Modelo de iluminação de Phong A *luz ambiente* é uma fonte de luz fictícia que representa o produto das múltiplas reflexões de luz de todas superfícies que constituem o ambiente. A equação correspodente à contribuição da luz ambiente na iluminação de um ponto P sobre a superfície de um objeto é

$$I_P = I_a \, k_a \, O_a, \tag{7.1}$$

onde I_a é a intensidade da luz ambiente (definida por uma cor RGB), assumida constante para todos os objetos, k_a, $0 \leq k_a \leq 1$, é o *coeficiente de reflexão difusa da luz ambiente* do objeto no ponto P e O_a é cor ambiente do objeto. (Alguns autores apresentam uma versão mais simples da Equação (7.1), com a omissão do termo O_a [38, 99]. Preferimos incluir a cor ambiente do objeto no modelo. Para desconsiderá-la, basta tomarmos O_a como sendo igual a cor branca.)

A contribuição da reflexão difusa na iluminação do ponto P é definida pela *lei dos cossenos de Lambert*. Para uma dada superfície, a intensidade de luz refletida difusamente depende apenas do ângulo θ entre a direção $\mathbf{L}_i$ e a normal $\mathbf{N}$ em P, Figura 7.11.

Para uma fonte de luz i com intensidade I_i, a equação da reflexão difusa é

$$I_P = I_i\, k_d\, O_d \cos\theta, \qquad 0^o < \theta \leq 90^o, \tag{7.2}$$

onde k_d, $0 \leq k_d \leq 1$, é o *coeficiente de reflexão difusa* do material no ponto P e O_d é a cor de reflexão difusa do objeto. Se os vetores $\mathbf{L}_i$ e $\mathbf{N}$ forem unitários, podemos escrever a Equação (7.2) como

$$I_P = I_i\, k_d\, O_d\, (\mathbf{N} \cdot \mathbf{L}_i). \tag{7.3}$$

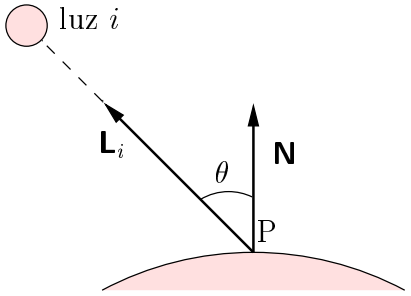


Figura 7.11: Reflexão difusa.

A contribuição da reflexão especular de uma fonte de luz i com intensidade I_i na iluminação do ponto P é definida pela expressão [99]

$$I_P = I_i\, k_s\, O_s\, (\mathbf{R}_i \cdot \mathbf{V})^{n_s}, \tag{7.4}$$

onde k_s, $0 \leq k_s \leq 1$, é o *coeficiente de reflexão especular* do material no ponto P, O_s é a cor de reflexão especular do objeto e n_s é o *expoente de reflexão especular* do objeto. Os valores de n_s variam de 1 até algumas centenas, dependendo da superfície. Valores pequenos de n_s caracterizam superfícies opacas e não metálicas; valores grandes de n_s caracterizam superfícies especulares. Os vetores $\mathbf{R}_i$ e $\mathbf{V}$, considerados unitários, são mostrados na Figura 7.12. A direção de reflexão $\mathbf{R}_i$ é dada por [38, 46]

$$\mathbf{R}_i = 2\mathbf{N}\, (\mathbf{N} \cdot \mathbf{L}_i) - \mathbf{L}_i. \tag{7.5}$$

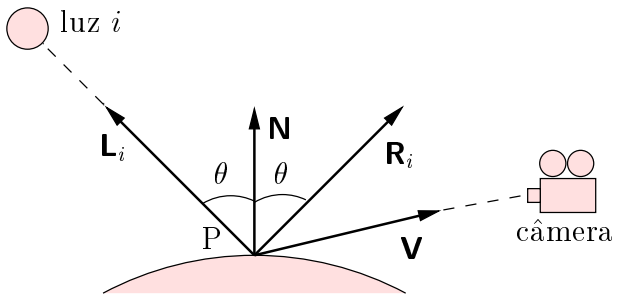


Figura 7.12: Reflexão especular.

Se tivermos NL fontes de luz na cena, a iluminação no ponto P é definida por

$$I_P = I_a\, k_a\, O_a + \sum_{i=1}^{\mathrm{NL}} I_i \left[k_d\, O_d\, (\mathbf{N} \cdot \mathbf{L}_i) + k_s\, O_s\, (\mathbf{R}_i \cdot \mathbf{V})^{n_s} \right]. \tag{7.6}$$

Atores

Um ator é um objeto da cena no qual incidem os raios luminosos das fontes de luz. A imagem da cena é obtida da interação da luz com a superfície dos atores. A definição C de um ator é apresentada no Programa 7.3. A estrutura `tActor` mantém o ponteiro `Mapper` para um mapeador, a matriz de transformação geométrica `Matrix` que define a posição, a orientação e a escala do ator na cena, a *flag* `Visible` que indica se o ator é visível ou não e os ponteiros para os elementos posterior e anterior na lista de atores da cena.

```
struct tActor
{
   tMapper*        Mapper;
   t3DTransfMatrix Matrix;
   bool            Visible;
   tActor*         Next;
   tActor*         Previous;
}; // tActor
```

Programa 7.3: Definição de ator.

Observando o Programa 7.3, podemos perguntar: se um ator é um objeto de uma cena, onde, em `tActor`, armazenamos o modelo que descreve a estrutura e o comportamento do objeto? A resposta é: em `Mapper`. (Lembremos que um mapeador é um processo consumidor que toma como entrada um repositório de dados.) `Mapper` é o responsável pela transformação dos dados estruturais e comportamentais do modelo nos primitivos gráficos que constituirão a imagem da cena. Definiremos a implementação de um mapeador no Capítulo 10 (veja a classe `tMapper`).

Câmeras

Uma câmera define a porção da cena que será visualizada pelo observador. A Figura 7.13 mostra os parâmetros de uma câmera. A *posição* é o ponto onde a câmera está, ou seja, a posição do observador. O *ponto focal* é o ponto para o qual a câmera está apontando. As coordenadas da posição e o do ponto focal são tomadas em relação ao sistema global de coordenadas (WC). O vetor formado pela diferença do ponto focal e da posição define a *direção de projeção* da câmera. O *plano de projeção*, ou *plano de vista*, é o plano perpendicular à direção de projeção que contém o ponto focal. O vetor unitário normal ao plano de vista é chamado VPN (*view plane normal*).

A *janela* é a porção retangular sobre o plano de vista que define a altura H e a largura W da imagem tomada pela câmera. A janela é orientada em relação a um sistema de coordenadas Cartesianas denominado *sistema de coordenadas de vista*, ou VRC (*view reference system*). A origem do VRC é o ponto focal e o eixo $\mathbf{n}$ do VRC, normal ao plano de projeção, é dado pelo VPN. Para definirmos os outros dois eixos, usamos um vetor chamado "para cima", tomado em relação ao WC. A projeção paralela ao VPN do vetor "para cima" no plano de projeção, define o eixo $\mathbf{v}$. O eixo $\mathbf{u}$ é dado diretamente pelo produto vetorial

$$\mathbf{u} = \mathbf{n} \times \mathbf{v}. \tag{7.7}$$

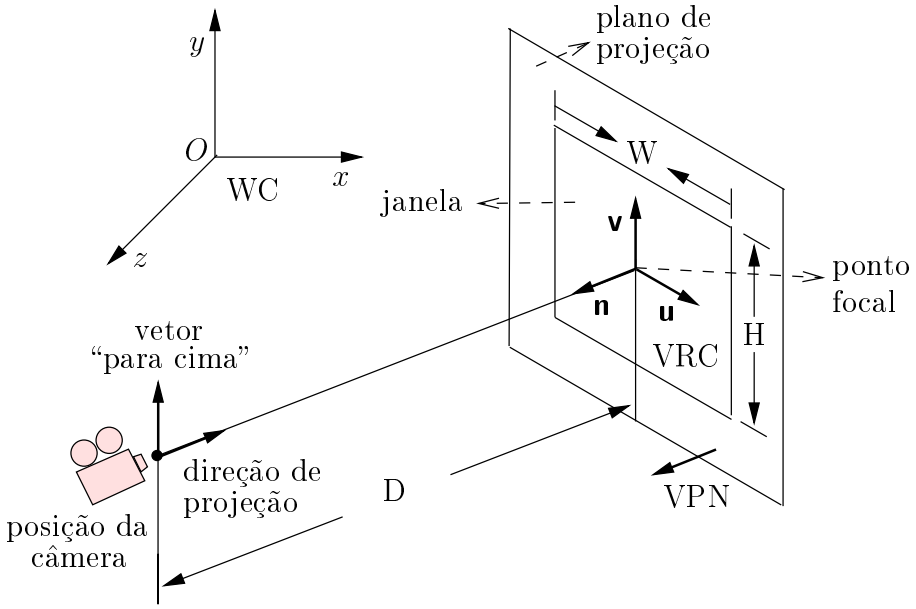


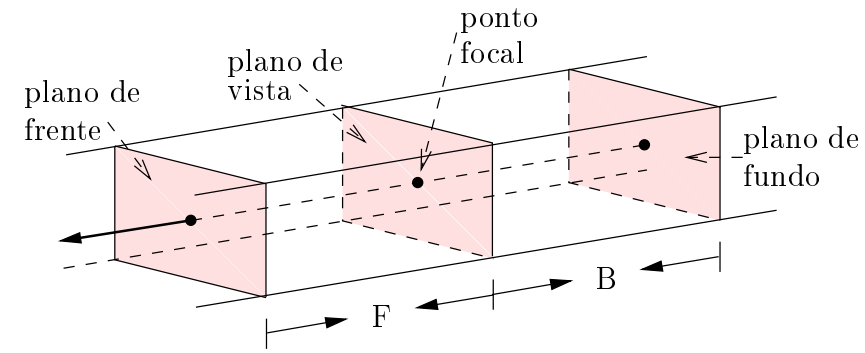
Figura 7.13: Parâmetros de uma câmera.

Para definirmos a porção da cena observada pela câmera, chamada *volume de vista*, devemos especificar qual o tipo de projeção utilizada. Consideraremos somente dois tipos de projeção: a projeção paralela ortográfica e a projeção perspectiva. Em uma projeção paralela, a posição da câmera é utilizada apenas para especificar, juntamente com o ponto focal, a direção de projeção. Em uma projeção paralela, supomos que o observador está no "infinito"; dessa forma, os raios de luz que chegam ao observador são paralelos. O volume de vista, nesse caso, é um paralelepípedo infinito, tal como mostrado na Figura 7.14(a). Na projeção perspectiva, o observador situa-se realmente na posição da câmera; os raios que chegam ao observador "passam" pela janela e convergem para a posição da câmera, também chamada, nesse caso, de *centro de projeção*. O volume de vista é um tronco de pirâmide semi-infinito, com vértice no centro de projeção, Figura 7.14(b). Na projeção perspectiva, a largura W da janela e a distância D entre a posição e o ponto focal definem o *ângulo de vista* θ da câmera, dado por

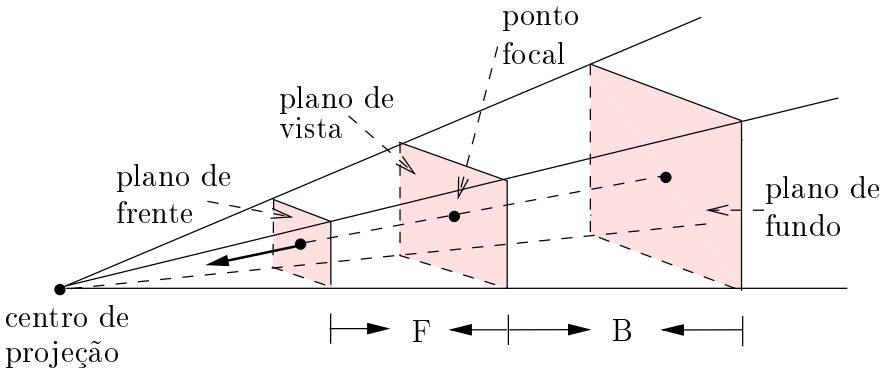
$$\theta = 2 \arctan\left(\frac{W}{2D}\right). \tag{7.8}$$

Ambos os volumes de vista podem ser delimitados por dois planos, paralelos ao plano de projeção, chamados *planos de recorte*. A posição dos planos é definida pelas distâncias F (plano de frente) e B (plano de fundo), tomadas sobre a direção de projeção a partir da origem do VRC.

A implementação C de uma câmera é mostrada no Programa 7.4. A estrutura `tCamera` mantém o tipo `ProjectionType` de projeção da câmera, a posição `Position`, o ponto focal `FocalPoint`, o vetor "para cima" `ViewUp`, a distância `Distance` entre a posição e o ponto focal, a largura `Width` da janela e a razão de aspecto `AspectRatio` da janela (a razão de aspecto é a razão W/H entre a largura e a altura da janela). Os demais atributos definem as distâncias `ClipDist` dos planos de recorte, as *flags* `ClipFlag`, indicando se os planos de recorte estão ativos ou não, uma matriz de visualização `ViewMatrix` (explicada a seguir) e os ponteiros para os elementos posterior e anterior na lista de câmeras da cena.



(a) Projeção paralela



(b) Projeção perspectiva

Figura 7.14: Volumes de vista.

Apenas para ilustrar o funcionamento da câmera, vamos descrever o processo de *rendering* de uma imagem fio-de-arame de um modelo de cascas. Uma imagem fio-de-arame é o tipo mais simples de imagem de um modelo implementado em OSW. Por isso mesmo, é a mais rápida de ser gerada. Usaremos imagens fio-de-arame para posicionar os objetos na cena, antes de solicitarmos uma imagem com luz. Os passos do *rendering* fio-de-arame são enumerados a seguir.

Passo 1 Para cada aresta e do modelo de cascas, definida pelos vértices de coordenadas P_1 e P_2, executaremos os passos 2 até 5.

Passo 2 **Normalização**. Primeiramente, os pontos P_1 e P_2 da aresta e são *normalizados*. Normalizar um ponto significa transformar suas coordenadas, dadas no WC, para as coordenadas do VRC. Além disso, efetuaremos a transformação das coordenadas do volume de vista para um *volume de vista canônico*. Dois volumes de vista canônicos são apresentados na Figura 7.15, para as projeções paralela e perspectiva. A função de um volume de vista canônico é facilitar a operação seguinte, chamada *recorte*. A transformação de normalização para a projeção paralela é definida por uma matriz de transformação obtida pela composição das seguintes transformações:

1. Translação do ponto focal para o origem do sistema global.
2. Rotação do sistema de coordenadas de vista tal que seus eixos se alinhem com os eixos do sistema global.

```
enum tProjectionType { pPARALLEL, pPERSPECTIVE };
struct tCamera
{
  tProjectionType ProjectionType;
  t3DVector       Position;
  t3DVector       FocalPoint;
  t3DVector       ViewUp;
  double          Distance;
  double          Width;
  double          AspectRatio;
  double          ClipDist[2];
  bool            ClipFlag[2];
  t3DTransfMatrix ViewMatrix;
  tCamera*        Next;
  tCamera*        Previous;
}; // tCamera
```

Programa 7.4: Definição de câmera.

3. Transformação de escala do volume de vista para o volume de vista canônico da projeção paralela.

A transformação de normalização para a projeção perspectiva é definida por uma matriz de transformação obtida pela composição das seguintes transformações:

1. Translação do ponto focal para o origem do sistema global.
2. Rotação do sistema de coordenadas de vista tal que seus eixos se alinhem com os eixos do sistema global.
3. Translação da posição da câmera para a origem do sistema global.
4. Transformação de escala do volume de vista para o volume de vista canônico da projeção perspectiva.

A matriz de normalização resultante, armazenada em `ViewMatrix`, é dada por

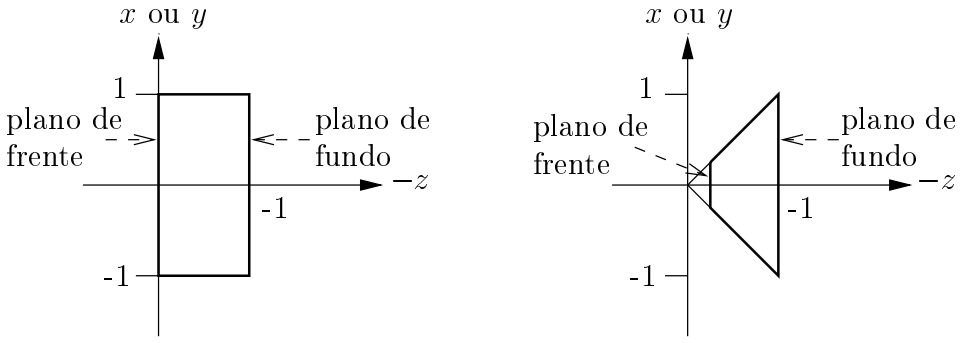
$$\begin{bmatrix} u_x s_x & u_y s_x & u_z s_x & -\mathbf{p}\cdot\mathbf{u} \\ v_x s_y & v_y s_y & v_z s_y & -\mathbf{p}\cdot\mathbf{v} \\ n_x & n_y & n_z & -\mathbf{p}\cdot\mathbf{n} + t_z \end{bmatrix}, \tag{7.9}$$

onde $\mathbf{p}$ é o ponto focal. Para a projeção paralela, temos

$$\begin{aligned} s_x &= \frac{2}{W}, \\ s_y &= \frac{2}{H}, \\ t_z &= 0. \end{aligned} \tag{7.10}$$

e, para a projeção perspectiva,

$$\begin{aligned} s_x &= \frac{2D}{W}, \\ s_y &= \frac{2D}{H}, \\ t_z &= D. \end{aligned} \tag{7.11}$$



(a) Projeção paralela (b) Projeção perspectiva

Figura 7.15: Volumes de vista canônicos.

Passo 3 **Recorte**. O recorte consiste na eliminação das porções do objeto que são invisíveis ao observador, ou seja, que não estão dentro do volume de vista. No caso fio-de-arame, estamos interessados no *recorte de linhas* [65]. A aresta e pode estar totalmente contida, totalmente fora ou interceptar o volume de vista em um ou dois pontos. Porém, ao invés de desenvolvermos uma função que efetue o recorte de linhas em um volume de vista genérico, escreveremos funções especializadas que efetuam o recorte em volumes de vista particulares, usualmente definidos por planos que reduzem ou simplificam os cálculos de intersecção. Esses volumes de vista são os volumes de vista canônicos. (Não apresentaremos as funções de recorte de linhas aqui.)

Passo 4 **Projeção**. Os pontos de coordenadas P_1' e P_2' que definem a porção da aresta e resultante do recorte, são *projetados* no plano de projeção. A matriz de transformação da projeção paralela é definida como

$$\mathbf{P}_{\text{par}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{7.12}$$

A matriz de transformação da projeção perspectiva é

$$\mathbf{P}_{\text{per}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/D & 0 \end{bmatrix}, \tag{7.13}$$

onde D é a distância entre a posição da câmera e o ponto focal. Note que as matrizes de transformação de projeção não são inversíveis. (Podemos eliminar uma das dimensões de um ponto no espaço e levá-lo para o plano, mas não podemos levar um ponto projetado no plano à sua posição original no espaço.) A função C de implementação da projeção de um ponto é mostrada no Programa 7.5.

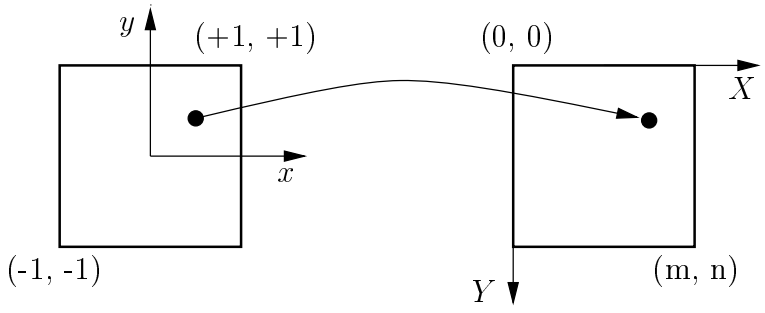
```
void Project(t3DVector* point, tCamera* camera)
{
  if (camera->ProjectionType == pPERSPECTIVE)
  {
    double z = camera->Distance / point->z;

    point->x *= z;
    point->y *= z;
  }
}
```

Programa 7.5: Projeção de um ponto.

Passo 5 **Mapeamento**. As coordenadas (x_1, y_1) e (x_2, y_2) das projeções dos pontos P_1' e P_2' da aresta e são, finalmente, *mapeados* para os pontos (X_1, Y_1) e (X_2, Y_2) da imagem, Figura 7.16, de acordo com as expressões

$$\begin{aligned} X &= \frac{m}{2}(1+x), \\ Y &= \frac{n}{2}(1-y), \end{aligned} \tag{7.14}$$

onde m e n são, respectivamente, a largura e a altura da imagem. Após o mapeamento dos pontos extremos da aresta, uma linha entre os dois pontos é desenhada na imagem.



(a) Janela (b) Imagem

Figura 7.16: Mapeamento janela-imagem.

Operações com a câmera Podemos especificar a vista de uma câmera ajustando diretamente os parâmetros da Figura 7.13 e calculando a `ViewMatrix` correspondente. Existem algumas operações, contudo, que tornam essa tarefa mais fácil e intuitiva. As operações implementadas em OSW são listadas a seguir.

Elevação. Rotação da câmera em torno do eixo definido pelo produto vetorial do VPN e do vetor "para cima", com centro no ponto focal.

Azimute. Rotação da câmera em torno do vetor "para cima", com centro no ponto focal.

Rolagem. Rotação da câmera em torno do VPN.

Guinada. Rotação do ponto focal em torno do eixo definido pelo produto vetorial do vetor "para cima" e do VPN, com centro na posição da câmera.

Arfagem. Rotação do ponto focal em torno do vetor "para cima" e do VPN, com centro na posição da câmera.

Zoom. Mudança das dimensões da janela de vista. Para maiores detalhes, veja a classe `tCamera` no Capítulo 10.

Encerraremos essa Seção com a definição da estrutura `tScene` mostrada no Programa 7.6. `tScene` contém os componentes genéricos (atores, luzes e câmeras) de uma base de dados de uma aplicação de modelagem em OSW, conforme veremos no Capítulo 9.

```
struct tScene
{
  tActor*  Actors;
  tLight*  Lights;
  tCamera* Cameras;
  tColor   AmbientLight;
  tColor   BkgndColor;
}; // tScene
```

Programa 7.6: Definição de cena.

7.4 Algoritmos de Visualização

Podemos classificar os algoritmos de transformação de acordo com a estrutura da transformação:

- *Transformações geométricas.* Uma transformação geométrica altera a geometria do modelo, mas não altera a topologia do modelo. Translação, rotação e transformação de escala, ou quaisquer combinações dessas transformações, são exemplos de transformações geométricas. As transformações geométricas foram discutidas no Capítulo 3.

- *Transformações topológicas.* Uma transformação topológica altera a topologia do modelo, mas não altera a geometria do modelo. 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. (Muitas vezes, porém, transformações na topologia também implicam na transformação da geometria.)

- *Transformações de atributos.* Uma transformação de atributos converte dados de atributos de uma forma para outra, ou cria novos atributos a partir dos atributos do modelo, sem modificar a estrutura (geometria + topologia) do modelo. Podemos considerar que os processos de análise numérica vistos no Capítulo 5 são algoritmos de transformações de atributos, porque criam novos atributos (deformações e tensões) a partir de outros (deslocamentos e forças de volume e superfície).

- *Combinações de transformações.* Podemos definir transformações que alteram, conjuntamente, a geometria, a topologia e os atributos de um modelo.

Como exemplo de um algoritmo de transformação topológica, consideremos as isolinhas mostradas na Figura 7.17. Observe que, diferentemente da Figura 7.2(b), a imagem exibe o desenho das *arestas de contorno* do modelo. Uma aresta de contorno é uma aresta que pertence a uma única célula do modelo. Note, porém, na estrutura `cModel` do Capítulo 3, que não representamos no modelo quaisquer informações a respeito da adjacência das arestas de uma célula. Portanto, devemos *computar* essas informações de adjacência a partir dos usos de vértices. A função a seguir determina o número de células vizinhas da célula `cell` na aresta definida pelos vértices `n0` e `n1` de `cell`:

```
int CountNeighbors(cCell* cell, cVertex* n0, cVertex* n1)
{
  int count = 0;

  for (cNodeUse* u0 = n0->Uses; u0; u0 = u0->Next)
    if (u0->Cell != cell)
      for (tNodeUse* u1 = n1->Uses; u1; u1 = u1->Next)
        if (u1->Cell == u0->Cell)
          ++count;
  return count;
}
```

Se o número de células vizinhas for igual a zero, então a aresta `n0-n1` é uma aresta de contorno.

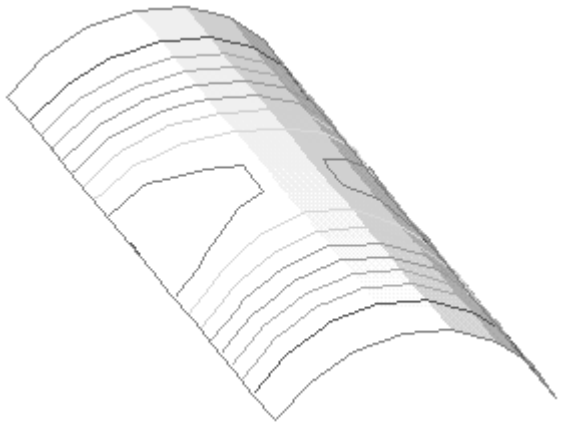


Figura 7.17: Exemplo de transformação topológica.

As arestas de contorno da Figura 7.17 foram obtidas por um *filtro de arestas de contorno*, um processo de transformação topológica de dados que toma como entrada

um modelo de decomposição por células e produz como saída um modelo gráfico que contém as linhas que representam as arestas de contorno do modelo de decomposição por células (veja a classe `tBoundaryEdgesFilter` no Capítulo 10). O filtro não altera a geometria da malha.

Os algoritmos de transformação também podem ser classificados de acordo com o tipo dos atributos do modelo de entrada e o tipo de saída que produzem. Podemos distinguir os algoritmos a seguir.

- *Algoritmos escalares.* Os algoritmos escalares operam sobre dados escalares, tais como pressão, temperatura, etc. Embora o problema fundamental seja definido em termos de vetores (deslocamentos, etc.) e tensores (deformações, etc.), podemos empregar os algoritmos escalares para os componentes de um vetor ou tensor. Abordaremos, a seguir, dois algoritmos escalares: mapa de cores e *contouring*.

- *Algoritmos vetoriais.* Os algoritmos vetoriais operam sobre dados vetoriais. Veremos um tipo bastante simples de algoritmo vetorial que nos permitirá visualizar, após a aplicação dos carregamentos externos, os deslocamentos da estrutura. Essa técnica é chamada de *warping*.

- *Algoritmos tensoriais.* Os algoritmos tensoriais operam sobre dados tensoriais (podemos utilizar ícones orientados para mostrar os componentes de tensão e deformação de um sólido). Não abordaremos aqui os algoritmos de visualização de tensores.

7.4.1 Visualização de Escalares

No DFD da Figura 7.3 apresentamos um filtro chamado `Geração de Isolinhas`. O repositório de dados de entrada do filtro é um modelo de decomposição por células e o repositório de dados de saída é um modelo gráfico que contém as isolinhas correspondentes a uma faixa determinada de valores escalares (veja a Figura 7.2). Antes de descrevermos o algoritmo de geração de isolinhas, vamos adicionar mais um atributo aos vértices de um modelo mecânico. O Programa 7.7 mostra a estrutura `cVertex`, definida primeiramente no Programa 3.19 e modificada no Programa 6.1. Observe que acrescentamos à estrutura o número real `Scalar`. Usaremos `Scalar` para armazenar o valor do escalar que iremos visualizar.

```
struct cVertex
{
  t3DVector   Position;
  cVertexUse* Uses;
  cVertex*    Next;
  cVertex*    Previous;
  int         Number;
  int         NumberOfDOFs;
  tDOF*       DOFs;
  double      Scalar;
}; // cVertex
```

Programa 7.7: Redefinição de vértice de um modelo mecânico.

Por exemplo, vamos supor que, para a casca da Figura 7.1, queiramos visualizar o campo de deslocamentos na direção y. O filtro `Extração de Escalares` "atravessa" todos os vértices do modelo de decomposição de células e armazena o valor `DOFs[1].u` (o deslocamento na direção y) em `Scalar`. Veremos a seguir como essa informação pode ser útil.

Mapas de Cores

O *mapeamento de cores* (*color mapping*) é uma técnica de visualização de escalares baseada no mapeamento dos dados escalares de um modelo em cores, as quais são exibidas no sistema gráfico.

A implementação do mapeamento de cores usa uma *tabela de cores* com n entradas, tal como mostrado na Figura 7.18. Cada entrada da tabela de cores contém os componentes de uma cor, dados em algum modelo de cores, tipicamente o modelo RGB. 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

$$i = \begin{cases} 0 & \text{se } s < s_{\min}, \\ n-1 & \text{se } s > s_{\max}, \\ \dfrac{s - s_{\min}}{s_{\max} - s_{\min}} & \text{se } s_{\min} \leq s \leq s_{\max}. \end{cases} \tag{7.15}$$

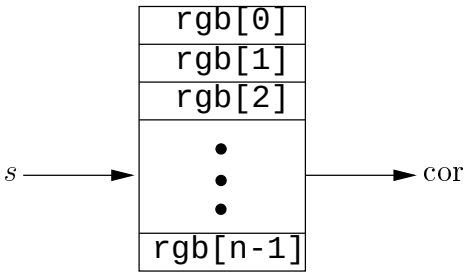


Figura 7.18: Mapeamento de cores: a tabela de cores.

Consideremos o exemplo dado anteriormente. Após a aplicação do filtro `Extração de Escalares`, o campo `Scalar` de cada vértice do modelo de decomposição por células da casca conterá o valor do deslocamento em y daquele vértice. O filtro determina, também, o deslocamento mínimo $s_{\min}$ e o deslocamento máximo $s_{\max}$ no modelo. Dada uma tabela de cores, podemos associar ao escalar `Scalar` de cada um dos vértices do modelo uma cor da tabela, de acordo com a Equação (7.15). Nesse ponto, temos uma cor associada a cada vértice do modelo. O mapa de cores sobre cada célula do modelo é gerado por interpolação das cores nos vértices da célula, através do processo de tonalização de Gouraud [21]. (A cor no vértice é utilizada como cor de reflexão difusa O_d na Equação (7.2).) O resultado é mostrado na Figura 7.2(a).

Isolinhas

O *contouring* é uma extensão natural do mapa de cores. Repare, na Figura 7.2, que as linhas de nível definem o "contorno" de algumas regiões coloridas do mapa de cores. "Contornos" tridimensionais são chamados de isosuperfícies, as quais podem ser

geometricamente aproximadas por polígonos. "Contornos" bidimensionais são chamados de isolinhas, as quais podem ser aproximadas por linhas retas. Os "contornos" unidimensionais são chamados isopontos. Construímos modelos gráficos no Capítulo 3 principalmente para representar isosuperfícies, isolinhas e isopontos.

Marchando em células LORENSEN [69] desenvolveu uma técnica simples e elegante para determinar isosuperfícies, denominada de *marching cubes*. Para explicar a técnica, usaremos sua versão bidimensional, chamada de *marching squares*. Posteriormente, generalizaremos a técnica para qualquer tipo de célula de um modelo de decomposição por células.

Consideremos a malha 2D esquematizada na Figura 7.19. 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. Vamos supor que esse valor seja igual a 5. Para descobrirmos os pontos sobre as células que possuem o valor 5, devemos utilizar alguma função de interpolação. Poderíamos pensar, naturalmente, em usar as funções de forma da célula. No entanto, seremos simplistas e usaremos uma função de interpolação linear sobre as arestas dos quadrados, tal como no processo de tonalização de Gouraud. (Sempre podemos dividir a malha em um número maior de células, se for o caso.) Se uma aresta de um quadrado possui em seus vértices os valores escalares 0 e 10, por exemplo, então uma isolinha passa pelo ponto médio da aresta. Uma vez gerados os pontos das isolinhas sobre as arestas em todas os quadrados, podemos conectar esses pontos para formar as isolinhas.

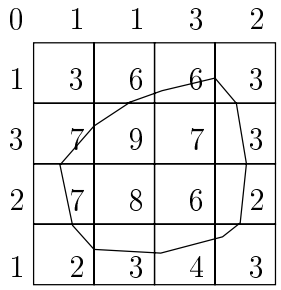


Figura 7.19: Isolinha em uma malha estruturada 2D.

A técnica "marchando em quadrados" é uma estratégia de divisão e conquista que trata cada célula do modelo separadamente. O primeiro passo do algoritmo é determinar se o valor escalar de cada vértice de um quadrado q é maior ou menor que o valor da isolinha (5, em nosso exemplo). A cada vértice de q associamos um valor binário (0 ou 1), dependendo do resultado da comparação. Se o valor escalar do vértice for menor que 5, o *bit* associado ao vértice será 0. Se o valor escalar do vértice for maior que 5, o *bit* associado ao vértice será 1. Esses quatro *bits* são organizados em um número binário que define o *estado topológico* do quadrado. Para uma célula genérica, o número de estados topológicos depende do número de vértices da célula e do número de relações lado de dentro/lado de fora que um vértice pode ter em relação ao valor da isolinha. Um vértice é considerado do lado de dentro de uma isolinha se o valor de seu escalar é maior que o valor do escalar da isolinha (*bit* igual a 1). Se o valor do escalar de um vértice é menor que o valor do escalar da isolinha, então o vértice é considerado do lado de fora da isolinha (*bit* igual a 0). No caso de um quadrado, como a célula

tem quatro vértices e cada vértice pode estar do lado de dentro ou do lado de fora de uma isolinha, há somente $2^4 = 16$ maneiras da isolinha passar através da célula. Essas maneiras são mostradas na Figura 7.20.

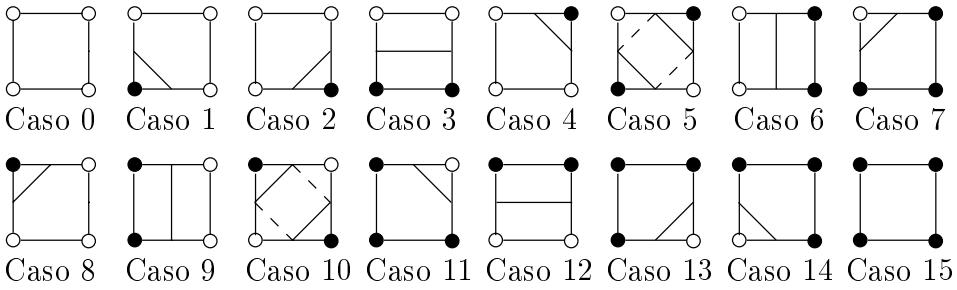


Figura 7.20: Marchando em células: os 16 casos de um quadrado.

Os 16 casos ilustrados na Figura 7.20 podem ser codificados em uma *tabela de casos*, onde cada entrada da tabela define (a)quantos segmentos da isolinha passam pelo quadrado e (b)em quais arestas do quadrado os vértices desses segmentos estão contidos. A entrada da tabela de casos correspondente ao estado topológico 0010, por exemplo, informa que somente um segmento da isolinha passa pelo quadrado e que os vértices desse segmento estão na aresta inferior e na aresta direita do quadrado (veja o caso 2 na Figura 7.20). Note que se os *bits* associados aos vértices do quadrado q forem todos iguais a 0 ou todos iguais a 1, a isolinha não passa pelo quadrado (casos 0 e 15 na Figura 7.20). Note, também, que os casos 5 e 10 são ambíguos. GALLAGHER [42] discute o tratamento de ambiguidades em *marching cubes*.

O estado topológico do quadrado q é usado como índice da tabela de casos do quadrado. Uma vez selecionado o caso, as localizações das intersecções da isolinha com as arestas de q são calculadas por interpolação linear. O algoritmo processa o quadrado q e então *marcha* para o próximo quadrado. Depois que todos os quadrados forem visitados, o "contorno" para o valor 5 estará completo.

Podemos utilizar o algoritmo para qualquer tipo de célula, se construirmos a tabela de estados topológicos para a célula. Os passos do algoritmo são sumarizados a seguir.

Passo 1 Para cada célula c de um modelo de decomposição por células, executamos os passos 2 até 5.

Passo 2 Calculamos o estado de cada vértice v da célula c. O estado do vértice v é definido por um dígito binário (0 ou 1) que identifica se o valor do escalar de v é maior ou menor que o valor escalar da isolinha (ou isosuperfície ou isoponto). Para uma célula c com n vétices, teremos n dígitos binários.

Passo 3 Criamos um índice de n dígitos binários, cada *bit* do índice correspondente ao estado de cada um dos n vértices de c.

Passo 4 Usamos o índice para determinar o estado topológico da célula c em uma tabela de casos.

Passo 5 Calculamos os pontos de intersecção da isolinha (ou isosuperfície ou isoponto) com cada uma das arestas da tabela de casos.

O Programa 7.8 mostra a tabela de casos para a célula quadrilateral com 4 vértices. A implementação do algoritmo é apresentada no Programa 7.9. As funções auxiliares `AddVect()`, `SubVect()` e `MulVect()` efetuam, respectivamente, a adição de dois vetores, a subtração de dois vetores e a multiplicação de um vetor por um escalar.

```
struct quad_LINE_CASES
{
  int edges[5];
}; // quad_LINE_CASES

static quad_LINE_CASES quad_LineCases[] =
{
  {{-1, -1, -1, -1, -1}},
  {{0, 3, -1, -1, -1}},
  {{1, 0, -1, -1, -1}},
  {{1, 3, -1, -1, -1}},
  {{2, 1, -1, -1, -1}},
  {{0, 3, 2, 1, -1}},
  {{2, 0, -1, -1, -1}},
  {{2, 3, -1, -1, -1}},
  {{3, 2, -1, -1, -1}},
  {{0, 2, -1, -1, -1}},
  {{1, 0, 3, 2, -1}},
  {{1, 2, -1, -1, -1}},
  {{3, 1, -1, -1, -1}},
  {{0, 1, -1, -1, -1}},
  {{3, 0, -1, -1, -1}},
  {{-1, -1, -1, -1, -1}}
};
```

Programa 7.8: Tabela de casos da célula quadrilateral.

7.4.2 Visualização de Vetores

Utilizaremos uma técnica bastante simples de visualização de vetores chamada *warping*. A técnica, usualmente empregada em engenharia, consiste na exibição da malha deformada de um modelo mecânico, como conseqüência da aplicação de esforços externos na estrutura.

Primeiramente, o algoritmo "atravessa" a coleção de vértices do modelo de decomposição por células e extrai, de cada vértice, o vetor de interesse, tipicamente o deslocamento **u** do vértice. (Poderíamos adicionar um vetor `Vector` à estrutura `cVertex` do Programa 7.7, tal como fizemos com o escalar `Scalar`.) 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 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

```
void Contour(cCell* cell, double value, gModel* model)
{
  static int CASE_MASK[4] = {1,2,4,8};
  static int edges[4][2] = {{0,1}, {1,2}, {2,3}, {3,0}};
  int i;
  int index;
  quad_LINE_CASES* lineCase;
  int* edge;
  t3DVector p[2];
  cVertexArray nodes = cell->Vertices;

  for (i = 0, index = 0; i < 4; i++)
    if (nodes[i]->Scalar >= value)
      index |= CASE_MASK[i];
  lineCase = quad_LineCases + index;
  edge = lineCase->edges;
  for (; edge[0] > -1; edge += 2)
  {
    for (i = 0; i < 2; i++)
    {
      int* vert = edges[edge[i]];
      t3DVector dp = SubVect(nodes[vert[1]]->Position -
        nodes[vert[0]]->Position);
      double t = (value - nodes[vert[0]]->Scalar) /
        (nodes[vert[1]]->Scalar - nodes[vert[0]]->Scalar);

      p[i] = AddVect(nodes[vert[0]]->Position, MulVect(dp, t));
      gNewLine(model, gNewVertex(p[0]), gNewVertex(p[1]));
    }
  }
}
```

Programa 7.9: Isolinhas da célula quadrilateral.

é visualizada, como exemplificado na Figura 7.21 (os deslocamentos foram aumentados 10 vezes).

7.5 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, definimos alguns processos de transformação de dados de um modelo. Um fonte é um processo produtor de modelos; um filtro é um processo que transforma um ou mais modelos em outros modelos; um mapeador é um processo consumidor de modelos. A conexão de fontes, filtros, mapeadores e modelos define o fluxo de dados desde a modelagem até a visualização de um objeto, conhecido como rede de visualização.

A visualização transforma dados em primitivos gráficos e a computação gráfica, fundação da visualização, transforma primitivos gráficos em imagens. Definimos com-

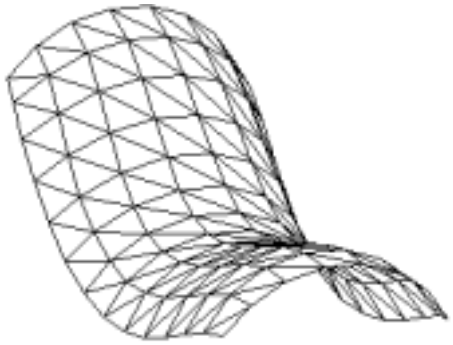


Figura 7.21: Exemplo da estrutura deformada de uma casca.

putação gráfica como sendo a síntese de imagens em computador, ou *rendering*. As técnicas de *rendering* tridimensional simulam a interação de câmeras e de luzes com os materiais que constituem as superfícies dos objetos de um ambiente. Chamamos esses objetos de atores. A coleção de atores, luzes e câmeras que definem um ambiente é denominada cena.

A especificação de cores em computação gráfica é efetuada através de um modelo de cores. Utilizamos dois tipos de modelos de cores: o modelo RGB (vermelho, verde, azul) e o modelo HSV (matiz, saturação e valor).

A interação da luz com a superfície de um ator é matematicamente definida por um modelo de iluminação. O modelo de iluminação de Phong considera as contribuições da luz ambiente, da reflexão difusa e da reflexão especular para a iluminação de um ponto da superfície de um ator.

Os algoritmos de visualização podem transformar a geometria, a topologia e/ou os atributos de um modelo. Os algoritmos de transformação também podem ser classificados de acordo com o tipo de dados sobre os quais operam. Algoritmos escalares operam sobre dados escalares, algoritmos vetoriais operam sobre dados vetoriais e algoritmos tensoriais operam sobre dados tensoriais. Vimos dois tipos de algoritmos de visualização de escalares: mapa de cores e *contouring*. Implementamos o *contouring* utilizando uma técnica de divisão e conquista chamada "marchando em células". Vimos, por fim, um método bastantes simples de visualização de uma estrutura deformada denomindo *warping*.

CAPÍTULO 8

O que são Objetos?

8.1 Introdução

Em sistemas de programação estruturada, dados e procedimentos são tratados como entidades separadas, sendo o programador responsável pela aplicação de procedimentos ativos a estruturas de dados passivas e, freqüentemente, pela garantia da adequação entre procedimentos e tipos de dados aos quais esses procedimentos são aplicados. Em programas orientados a objetos, ao contrário, a modelagem do domínio de aplicação é feita de acordo com uma visão tão próxima quanto possível do mundo real, através de *objetos* que se comunicam por troca de *mensagens*.

OBJETOS Objetos são um conjunto de dados mais um conjunto de procedimetos que representam a estrutura e o comportamento de uma entidade concreta ou abstrata. Os dados, ou atributos, são informações que descrevem a estrutura. Os procedimentos, ou métodos, são operações pré-definidas que acessam os atributos e respondem sobre o comportamento do objeto.

Estacas, sapatas, pilares, vigas, lages e alvenarias são, sem dúvida, objetos, na essência da palavra. Possuem uma coleção de *atributos* (material, características mecânicas, dimensões, custos, métodos construtivos) expressos através de números e textos descritivos, bem como de *métodos* próprios que manipulam e fornecem respostas a partir dos valores desses atributos (deslocamentos, tensões, reações, custo total). Entretanto, entidades não tão concretas, tais como elementos finitos de um domínio discretizado, por exemplo, podem ser (e de fato são) tratados como objetos. Nesse caso, o estado interno é representado por valores de variáveis como número e coordenadas dos nós do elemento e propriedades mecânicas específicas; o comportamento é ditado por métodos que informam a área, o volume ou a contribuição do elemento na resolução do modelo matemático. Equações diferenciais, sistemas lineares e quadraturas Gaussianas são, igualmente, sob a ótica da POO, todos objetos. Na verdade, nada impede que a essência de um objeto seja o *processamento*, ao invés da estruturação de dados; um *objeto*, em princípio, pode ser o *modelo* de qualquer coisa, desde um processo puro (análise numérica de algum domínio submetido a carregamentos estaticamente

aplicados, por exemplo) até um dado puro (definição da geometria e da topologia de algum sólido, por exemplo).

A *orientação a objetos* é a técnica de desenvolvimento (engenharia) e modelagem de sistemas que facilita a construção de programas complexos a partir de componentes individuais [59], através de ferramentas e conceitos que nos permitem modelar o mundo real tão próximo quanto possível da visão que temos desse mundo. De acordo com LEDBETTER e COX [64],

> "(..) a programação orientada a objetos permite uma representação mais direta do mundo real no código. Como resultado, a transformação radical normal dos requisitos do sistema (definidos em termos do usuário) em especificações do sistema (definidos em termos computacionais) é amplamente reduzida."

Nesse Capítulo discutiremos os conceitos básicos e as propriedades mais importantes da programação orientada a objetos. Esses conceitos são independentes de uma linguagem de programação em particular, orientada a objetos ou não. (Podemos programar orientado a objetos sem utilizar, necessariamente, uma linguagem orientada a objetos.) No entanto, guiaremos nossas discussões com exemplos de trechos de programas em C++ [110], a linguagem na qual escrevemos OSW. Nosso objetivo é introduzir os termos mais comumente empregados em orientação a objetos necessários ao entendimento da segunda parte do texto. Na Seção 8.2 apresentaremos as definições de classe de objetos, instância e acoplamento mensagem/método. Na Seção 8.3 discutiremos os principais mecanismos da programação orientada a objetos: *encapsulamento*, *herança* e *polimorfimo*.

8.2 Conceitos Básicos

Nessa Secão introduziremos alguns conceitos básicos sobre objetos e sobre programação orientada a objetos.

8.2.1 Classe

Na Seção anterior definimos *objeto* como sendo um conjunto de dados e de procedimentos que representam a estrutura e o comportamento de uma entidade concreta ou abstrata. Chamamos o conjunto de dados de *atributos* do objeto e o conjunto de procedimentos de *métodos* do objeto. Diremos que objetos com a mesma estrutura e o mesmo comportamento pertencem a uma mesma *classe* de objetos. Uma classe, portanto, é uma descrição dos atributos e dos métodos de determinado *tipo* de objeto.

Em C++, uma classe é definida pela palavra reservada **`class`**, como exemplificado no Programa 8.1.

A classe `t3DVector` corresponde à estrutura `t3DVector` definida no Programa 3.1.[1] Note, no entanto, que acrescentamos um conjunto de procedimentos que nos fornecem respostas sobre algumas questões pertinentes a um vetor no espaço. Por exemplo, qual é o comprimento de um vetor, qual é o resultado do produto interno de dois vetores, etc. Declaramos alguns desses procedimentos, aparentemente, como funções

[1] Em C++ uma estrutura é um caso particular de classe.

```
1   class t3DVector
2   {
3     public:
4       t3DVector();
5       t3DVector(const t3DVector&);
6       t3DVector(double, double, double);
7
8       void             SetCoordinates(const t3DVector&);
9       void             SetCoordinates(double, double, double);
10
11      t3DVector&       operator =(const t3DVector&);
12      t3DVector        operator +(const t3DVector&) const;
13      t3DVector        operator -(const t3DVector&) const;
14      t3DVector        operator *(double) const;
15      friend t3DVector operator *(double, const t3DVector&);
16      double           operator *(const t3DVector&) const;
17      t3DVector&       operator +=(const t3DVector&);
18      t3DVector&       operator -=(const t3DVector&);
19      t3DVector&       operator *=(const t3DVector&);
20      bool             operator ==(const t3DVector&) const;
21      bool             operator !=(const t3DVector&) const;
22
23      bool             IsNull()const;
24      double           Length() const;
25      double           Inner(const t3DVector&) const;
26      t3DVector        Cross(const t3DVector&) const;
27      t3DVector        Versor() const;
28      t3DVector&       Normalize();
29
30      double           x;
31      double           y;
32      double           z;
33
34      static const t3DVector XAxis;
35      static const t3DVector YAxis;
36      static const t3DVector ZAxis;
37
38  }; // t3DVector
```

Programa 8.1: Classe de vetor 3D.

comuns do C, porém embutidos na definição da classe.[2] Outros, diferentemente do C, tomam o nome de certos operadores da linguagem. Em C++, esses procedimentos, ou métodos, são chamados de *funções membro*. Os atributos, ou *variáveis de instância* são denominados, em C++, de *dados membro*. Comentaremos o Programa 8.1 à medida que introduzirmos outros conceitos.

8.2.2 Instância

Uma *instância* é um objeto gerado por uma classe. Podemos dizer, por outro lado, que um objeto é uma instância de uma classe. Em C++, podemos instanciar um objeto da mesma forma que o fazemos para os tipos primitivos da linguagem. Para criar uma instância de um `t3DVector`, por exemplo, declaramos, simplesmente

```
t3DVector v;
```

Observe: `t3DVector` é a classe e `v` é uma instância, um objeto da classe `t3DVector`. Podemos criar uma instância de uma classe dinamicamente, usando o operador C++ **`new`**:

```
t3DVector* p = new t3DVector;
```

Note que `p` *não* é um objeto da classe `t3DVector`, mas sim um *ponteiro* para um objeto da classe `t3DVector`. O objeto propriamente dito, `*p`, está na área de memória de alocação dinâmica, o *heap* do sistema. (O operador **`new`** é, sob alguns aspectos, similar à função `malloc`, porém muito superior, como veremos posteriormente.)

O conteúdo de um objeto, ou seu *estado interno*, é definido por suas variáveis de instância, declaradas na definição de classe. Os vetores `v` e `*p` possuem estados internos (distintos) caracterizados pelas variáveis `x`, `y` e `z` (linhas 30-32 do Programa 8.1), as quais representam suas coordenadas Cartesianas. Podemos ter, também, variáveis que não pertencem a qualquer instância de uma classe, mas que são *compartilhadas* por todas as instâncias da classe. Em Smalltalk [47], essas variáveis pertencem à classe, e não às instâncias da classe e, por isso, são denominadas *variáveis de classe*. Em C++, uma variável de classe é declarada com o especificador de tipo **`static`**, como nas linhas 34-36 do Programa 8.1.

8.2.3 Mensagem

Vamos supor que `s` seja uma estrutura tal como definida no Programa 3.1. Se quisés-semos acessar um dos campos de `s`, por exemplo, a coordenada `x`, escreveríamos `s.x`. A mesma regra sintática vale em C++. Adicionalmente, podemos "acessar" também as funcões definidas na classe:

```
v.Normalize();
p->SetCoordinates(1,1,1);
```

No exemplo acima, estamos "acessando" o método `Normalize` do vetor `v` e o método `SetCoordinates` do vetor apontado por `p`. Semanticamente, porém, o significado é outro. O que estamos fazendo, na verdade, é enviando uma *mensagem* para os objetos `v` e `*p`, uma solicitação para que esses objetos executem alguma operação. O programa

[2] Em seguida, veremos como implementamos o código dos procedimentos.

anterior quer dizer, em C++, "`v`, por favor, normalize-se" e "`*p`, por favor, ajuste suas coordenadas para $(1, 1, 1)$." Quando enviamos uma mensagem a um objeto, dizemos ao objeto *o que fazer*, e não *como fazer*.

Uma mensagem é definida por um *receptor*, por um *seletor* e por um conjunto, eventualmente vazio, de *parâmetros*. Na segunda linha do exemplo acima, `*p` é o receptor da mensagem, `SetCoordinates` é o seletor da mensagem e `1,1,1` são os parâmetros da mensagem.

Quando um objeto recebe uma mensagem, o método correspondente à mensagem é executado. Em C++, o "método correspondente a uma mensagem" é a função membro, declarada na definição da classe do objeto, que possui o mesmo nome do seletor da mensagem e o mesmo número e tipo de parâmetros da mensagem. (Note que podemos ter métodos com o mesmo nome, desde que com tipos e/ou número distintos de parâmetros, como nas linhas 8-9 do Programa 8.1.) Chamamos essa operação de *acoplamento mensagem/método*. Portanto, em resposta à mensagem `SetCoordinates(1,1,1)`, o vetor `*p` "acopla" e executa o método `SetCoordinates` declarado na linha 9 do programa. O valor retornado por um objeto, como resultado do processamento de uma mensagem, é o próprio valor da função membro correspondente (eventualmente, **`void`**).

Em um programa orientado a objetos, a computação é realizada por objetos que trocam mensagens entre si. Consideraremos somente o modelo *passivo* de objetos, isto é, um objeto que envia uma mensagem para um outro objeto não executa mais nada enquanto não obtiver, da parte do receptor, resposta da mensagem enviada.

8.2.4 Construção e Destruição de Objetos em C++

Imediatamente após a construção de um objeto, seu estado interno deve ser inicializado. Em C++, isso é feito pela execução de métodos especiais chamados *construtores*. Um construtor em C++ é uma função sem tipo de retorno (nem mesmo **`void`**) cujo nome é idêntico ao nome da classe. A classe `t3DVector`, como podemos observar nas linhas 4-6 do Programa 8.1, possui três construtores. O primeiro não toma quaisquer parâmetros e é chamado de construtor *default*. O segundo toma como parâmetro uma *referência* para um objeto constante da classe `t3DVector`; esse construtor é chamado *construtor de cópia* e é utilizado para inicializar o estado interno do objeto sendo construído com uma cópia do estado interno do objeto passado como parâmetro. O terceiro toma como parâmetros três números reais correspondentes às coordenadas do vetor sendo construído.

No momento da declaração de um objeto em C++, estática ou dinamicamente como visto acima, podemos enviar uma mensagem ao novo objeto solicitando sua inicialização. Por exemplo,

```
t3DVector u(2,2,2);
t3DVector w(u);
```

Estamos dizendo, acima, "`u`, por favor, inicialize-se com as coordenadas $(2, 2, 2)$" e "`w`, por favor, inicialize-se com uma cópia de `u`." (Se não enviarmos mensagem alguma na construção de um novo objeto, o compilador automaticamente chama o construtor *default* da classe, se houver. Se nenhum construtor *default* for declarado, o compilador fornece um, a menos que um outro tipo de construtor tenha sido definido.)

C++ é uma linguagem que não possui "coleta de lixo" automática. Por isso, todo objeto que utilizar memória dinâmica deve liberar a memória utilizada antes de ser destruído. Em C++, há métodos especiais, chamados de *destrutores*, que podem ser definidos para executar essa tarefa. Um destrutor, se declarado em uma classe, é chamado automatica e imediatamente antes da destruição de um objeto da classe. (Se não há destrutor, o compilador fornece um.) Um destrutor é definido em uma classe C++ por uma função com o mesmo nome da classe precedido pelo símbolo ~, sem argumentos e sem tipo de retorno. A classe `t3DVector` não possui destrutor (veja a próxima Seção).

O tempo de vida de um objeto em um programa C++ é definido pelas regras de escopo da linguagem, tal como no C.[3] Um objeto criado dentro de um bloco será destruído ao final do bloco (a mensagem é enviada automaticamente pelo compilador). A única exceção é para aqueles objetos criados dinamicamente. Nesse caso, devemos explicitamente invocar o destrutor do objeto, utilizando o operador **`delete`**. Para o vetor `*p`, escrevemos

```
delete p;
```

(O operador **`delete`** é, sob alguns aspectos, similar à função `free`, porém, da mesma forma que o operador **`new`**, muito superior. Um dos motivos é que **`delete`** não somente libera memória, mas envia uma mensagem ao objeto solicitando, e ao mesmo tempo permitindo, que este faça sua própria "limpeza".)

8.3 Propriedades da Orientação a Objetos

Por que precisamos de uma classe de objetos? Poderíamos argumentar que a classe `t3DVector`, por exemplo, nos permite modularizar um pouco mais o código, porque estamos definindo, em uma única unidade, as estruturas de dados e os procedimentos que manipulam essas estruturas. Mas, se o benefício fosse somente esse, poderíamos conseguí-lo com uma boa prática de programação em C. Antes de prosseguirmos a discussão, vejamos dois conceitos relacionados à programação orientada a objetos:

Abstração. A *abstração* é um dos mecanismos mais importantes utilizados pela mente humana para compreender o mundo real. Ao analisarmos um problema complexo, naturalmente separamos objetos que, na realidade, nunca se encontram isolados, caracterizando-os a partir de certas propriedades que definem aproximadamente seus principais aspectos estruturais e comportamentais. A abstração sempre deve ter algum propósito, porque é o propósito que permite determinar quais informações serão representadas e que operações serão executadas pelos objetos.

Tipos abstratos de dados. *Tipo abstrato de dados* é o mecanismo através do qual uma linguagem de programação orientada a objetos fornece suporte à especificação das informações de um objeto e das operações executadas por um objeto, de forma similar ao tipo de dados de uma linguagem estruturada. A diferença é que um tipo abstrato de dados *esconde* as informações sobre a estrutura do

[3] C++ não admite diretamente o conceito de *persistência*, e por isso não a discutiremos aqui. Um objeto persistente é um objeto que sobrevive à execução de um programa, sendo, portanto, armazenado em uma base de dados (orientada a objetos) [58, 108].

objeto de observadores externos. Com isso, as abstrações de um sistema são definidas, na prática, por duas partes: uma *interface* que descreve *quais* operações podem ser executadas e uma *implementação* que determina *como* as operações são executadas. A *classe* é a construção de linguagem mais comumente utilizada para implementar um tipo abstrato de dados em linguagens de programação orientadas a objetos.

De acordo com a definição de tipo abstrato de dados, as variáveis de instância declaradas em uma definição de classe deveriam ser "escondidas" de um observador externo, sendo acessíveis somente aos próprios métodos da classe. Afinal, as variáveis de instância definem o estado *interno* do objeto. Em C++, porém, somos livres para controlar a visibilidade de qualquer membro de uma classe, seja dado ou função. Observe, na linha 3 do Programa 8.1, o uso da palavra reservada **`public`**. Significa que, daquele ponto em diante, todos os membros declarados na classe são públicos, podendo ser diretamente acessados. Para um vetor `v`, poderíamos escrever, corretamente, `v.x` para obter (ou alterar) o valor da coordenada `x` de `v`. Para os defensores "puristas" da orientação a objetos, essa possibilidade é inaceitável e, mesmo em C++, é fortemente não recomendável. Vejamos por quê.

8.3.1 Encapsulamento

Consideremos a definição (parcial) da classe `gPrimitive`, Programa 8.2. Essa classe representa um tipo genérico de primitivo de um modelo gráfico, tal como caracterizado no Capítulo 3.

```
1   class gPrimitive
2   {
3     public:
4       gPrimitive(gModel*);
5       virtual ~gPrimitive();
6
7       gModel*      GetParent();
8       ...
9
10    protected:
11      gModel*      Parent;
12      gPrimitive*  Next;
13      gPrimitive*  Previous;
14
15      friend       gModel;
16
17  }; // gPrimitive
```

Programa 8.2: Definição C++ de primitivo gráfico.

Observe que temos duas partes bem distintas na classe `gPrimitive`. A primeira parte começa na linha 3 com a palavra reservada **`public`** e vai até a linha 9. Declaramos na linha 4 um construtor que toma como argumento um ponteiro para `gModel`, um destrutor virtual na linha 5 (explicaremos o significado de "virtual" posteriormente) e, na linha 7, um método `GetParent` que nos informa qual é o modelo "pai" do primitivo.

Note que só temos métodos públicos na primeira parte da classe. Chamaremos essa parte de *interface*, ou *protocolo* da classe. É na interface que definimos quais mensagens podemos enviar às instâncias de uma classe, ou seja, *quais* são as operações que podemos solicitar aos objetos de uma classe.

A segunda parte começa na linha 10 com a palavra reservada **`protected`** e vai até a linha 15. Usamos **`protected`** para dizer que todos os membros declarados daquele ponto em diante, dados ou funcões, são *protegidos*, ou escondidos, de outros objetos que não pertencem à classe `gPrimitive`. (Em C++, podemos declarar atributos e métodos *privados*, usando a palavra reservada **`private`**.)[4] Como no Capítulo 3, escolhemos implementar a coleção de primitivos de um modelo gráfico como uma lista ligada duplamente encadeada de primitivos. Na linha 11 declaramos um ponteiro `Parent` para o modelo "pai" do qual o primitivo faz parte; nas linhas 12 e 13 declaramos os ponteiros `Next` e `Previous` para os elementos posterior e anterior da lista, respectivamente. Note que o método `GetParent` é realmente necessário se tivermos necessidade de obter o modelo "pai" de um primitivo, uma vez que não podemos acessar diretamente o atributo protegido `Parent`.

Via de regra, quando definirmos uma classe, declararemos como públicos somente os métodos da interface da classe. Os métodos que executam operações internas sobre os dados, se houverem, e os próprios dados, serão "escondidos" na definição de classe. Chamamos essa propriedade da orientação a objetos de *encapsulamento*. O objetivo do encapsulamento é separar o usuário do objeto do programador do objeto. O benefício óbvio é que podemos alterar a implementação de um método ou a estrutura de dados "escondidos" de um objeto sem afetar as aplicações que dele se utilizam. Nesse sentido, a declaração das variáveis de instância na seção pública da classe `t3DVector` é um mau exemplo. (É mais compacto e menos confuso, no entanto, que a utilização de métodos tais como `GetX()`, `GetY()` e `GetZ()`. Além disso, não acreditamos que a quebra de encapsulamento em `t3DVector` possa vir a introduzir maiores complicações no desenvolvimento de programas que utilizam a classe.)

Imediatamente surge uma questão: "para acessar um simples dado de um objeto, então, sempre será preciso enviar uma mensagem ao objeto, o que corresponde à execução de uma função. Isso não torna o sistema ineficiente?" A resposta é sim. Em C++, porém, podemos contornar o problema definindo uma função como sendo **`inline`**:

```
inline gModel*
gPrimitive::GetParent()
{
  return Parent;
}
```

(O exemplo mostra a implementação da função `gPrimitive::GetParent()`. O nome da função dever ser precedido, no caso do corpo da função ser definido fora da classe, do nome da classe seguido do operador `::`, para indicar que `GetParent` é membro de `gPrimitive`.) Uma chamada a uma função **`inline`** é resolvida, pelo compilador, por uma expansão "em linha" do corpo da função no ponto de chamada. No caso do exemplo acima, é exatamente o que queremos. (Devemos usar a construção **`inline`**

[4] A diferença entre membros protegidos e privados de uma classe é que os privados não são visíveis nem mesmo para os objetos das classes derivadas (veja herança para uma definição de classe derivada).

com critério. Para funções maiores que são chamadas em muitos pontos distintos do programa podemos ter um acréscimo significativo no tamanho do código final.)

C++ permite ainda um outro mecanismo para a quebra de encapsulamento dos dados de uma classe, mostrado na linha 15 do Programa 8.2 e igualmente não recomendável: classes (ou funções) amigas. Uma classe `A` **`friend`** de uma outra classe `B` tem permissão, dada pela própria classe `B`, para acessar diretamente o estado interno de qualquer instância de `B`. A classe `gModel` é amiga de `gPrimitive` porque uma instância de `gModel` contém uma coleção de objetos da classe `gPrimitive`. Freqüentemente, um objeto `gModel` necessita "atravessar" a coleção de primitivos, mas, para isso, teríamos que declarar, em `gPrimitive`, métodos para obtenção dos ponteiros protegidos `Next` e `Previous`. Optamos por definir **`friend`** `gModel`.

8.3.2 Herança

No Capítulo 3 definimos alguns tipos simples de primitivos gráficos como *especializações* do tipo genérico `gPrimitive`. Uma modelo gráfico, por exemplo, é um primitivo que contém uma coleção hierárquica de outros primitivos. No Programa 3.5 implementamos a estrutura `gModel` como sendo constituída dos dados comuns de um primitivo genérico, declarados no Programa 3.3, mais os dados específicos de um modelo gráfico. Em C++, podemos dizer explicitamente que uma classe é uma especialização de outra, como mostrado no Programa 8.3.

```
1   class gModel:public gPrimitive
2   {
3     public:
4       gModel(gModel*);
5       ~gModel();
6
7       void        Add(gPrimitive*);
8       void        Remove(gPrimitive*);
9       ...
10
11    protected:
12      gPrimitive* Primitives;
13
14  }; // gModel
```

Programa 8.3: Definição C++ de modelo gráfico.

Na linha 1 do programa, estamos declarando que a classe `gModel` *deriva* da classe `gPrimitive`, ou seja, além de atributos e métodos próprios, contém todos os atributos e métodos de `gPrimitive`, sem que, com isso, tenhamos que definí-los novamente em *gModel*. Essa propriedade é chamada de *herança*.

A herança é a principal característica de distinção entre um sistema de programação orientada a objetos e outros sistemas de programação. Classes são inseridas em uma *hierarquia* de especializações de tal forma que uma classe mais especializada *herda* todas as propriedades da classe mais geral a qual é subordinada na hierarquia. A classe mais

geral é denominada *superclasse* (*classe base*, em C++) e a classe mais especializada é denominada *subclasse* (*classe derivada*, em C++).[5]

O principal benefício proporcionado pelo mecanismo de herança é a *reutilização de código*. Por exemplo, suponha que `model` seja instância de `gModel`. É correto escrevermos

```
model.GetParent();
```

Observe, porém, que `GetParent` não foi redefinida em `gModel`, mas sim herdada da classe base `gPrimitive`.

O Programa 8.4 mostra a implementação dos métodos de adição e remoção de um primitivo à lista de primitivos de um modelo gráfico. Diferentemente do Programa 3.6, essas funcões são agora encapsuladas na classe `gModel`. O ponteiro `Primitives` pode apontar para qualquer objeto cuja classe é derivada de `gPrimitive`. O Programa 8.5 mostra como esses métodos são utilizados pelo construtor e destrutor da classe `gPrimitive`. (A palavra reservada **`this`** representa um ponteiro para o objeto receptor da mensagem.)

```
1   void
2   gModel::Add(gPrimitive* prim)
3   {
4     prim->Next = Primitives;
5     prim->Previous = 0;
6     if (Primitives)
7       Primitives->Previous = prim;
8     Primitives = prim;
9     prim->Parent = this;
10  }
11
12  void
13  gModel::Remove(gPrimitive* prim)
14  {
15    if (prim->Next)
16      prim->Next->Previous = prim->Previous;
17    if (prim->Previous)
18      prim->Previous->Next = prim->Next;
19    if (Primitives == prim)
20      Primitives = 0;
21    prim->Parent = 0;
22  }
```

Programa 8.4: Adição e remoção C++ de primitivos de um modelo gráfico.

O Programa 8.6 mostra outro exemplo de herança, a classe `gLine` derivada de `gPrimitive`. A implementação do construtor da classe `gLine` é apresentada no Programa 8.7. Note, na linha 2 do programa, a sintaxe para inicializarmos a porção `gPrimitive` de `gLine`. Observe que não precisamos nos preocupar com a adição da linha na lista de primitivos do modelo. Essa tarefa é executada durante a inicialização

[5] O conceito de *herança múltipla* é suportado em C++: mais que uma classe base pode contribuir para a estrutura e comportamento de instâncias de uma classe derivada.

```
1   gPrimitive::gPrimitive(gModel* parent)
2   {
3     if (parent)
4       parent->Add(this);
5   }
6
7   gPrimitive::~gPrimitive()
8   {
9     if (Parent)
10      Parent->Remove(this);
11  }
```

Programa 8.5: Construtor e destrutor de primitivo gráfico.

da porção `gPrimitive` da linha. Não é necessário, também, nenhuma conversão de tipos, como fizemos na função `gNewLine()` do Capítulo 3. De fato, uma linha *é* um primitivo gráfico.

```
1   class gLine:public gPrimitive
2   {
3     public:
4       gLine(gModel*, gVertex&, gVertex&);
5
6       gVertex& From();
7       gVertex& To();
8       ...
9
10    protected:
11      gVertex  V1;
12      gVertex  V2;
13
14  }; // gLine
```

Programa 8.6: Definição C++ de linha.

```
1   gLine::gLine(gModel* parent, gVertex& v1, gVertex& v2):
2     gPrimitive(parent)
3   {
4     V1 = v1;
5     V2 = v2;
6   }
```

Programa 8.7: Construtor do primitivo linha.

Nesse ponto é conveniente ressaltarmos a distinção entre *hierarquia de herança* de classes (mecanismo de criação de classes de objetos que compartilham propriedades com outras classes de objetos similares) e *hierarquia de partes* de objetos (mecanismo de criação de objetos compostos de outros objetos). A primeira é representada, em um diagrama de objetos, por associacões do tipo *especialização* e a última por associações do tipo *agregação*. (Compare os diagramas da Figura 3.5 e da Figura 3.8.)

8.3.3 Polimorfismo

O *polimorfismo* é mais conhecido em programação como a propriedade de um procedimento executar com um número e tipo variado de parâmetros, ou de um símbolo representar operações distintas, de acordo com os tipos dos operandos. Em Pascal, por exemplo, o símbolo '+' pode denotar soma de inteiros, soma de reais, união de conjuntos ou concatenação de cadeias de caracteres. Essa facilidade, conhecida como *sobrecarga*, caracteriza o que se costuma chamar *acoplamento estático* (ou junção anterior), só possível nos casos onde os tipos de operandos ou parâmetros são conhecidos em tempo de compilação.

Voltemos ao Programa 8.1. A classe `t3DVector` apresenta alguns casos de sobrecarga. Iniciando na linha 11 até a linha 21, vemos a sobrecarga de diversos operadores do C++. Com esses operadores, podemos escrever trechos de programa contendo expressões aritméticas envolvendo vetores:

```
t3DVector a(1,1,1);
t3DVector b(2,2,2);
t3DVector c;

c = (a + b) * (a * b);
```

No exemplo acima, `c` recebe o resultado da multiplicação do vetor soma `a + b` pelo escalar resultante do produto interno `a * b`. O código é bastante claro. Pode não ser óbvio (e talvez a intenção de STROUSTRUP [110], o inventor de C++, tenha sido exatamente essa) que o vetor `c` possui um estado interno definido pela seguinte sequência de execução de mensagens:

```
c.operator =((a.operator +(b)).operator *(a.operator *(b)));
```

Nas linhas 8 e 9 do Programa 8.1 temos outro caso de sobrecarga, dessa vez de identificadores. Podemos escrever:

```
v.SetCoordinates(a);
c.SetCoordinates(3,3,3);
```

O compilador gera código corretamente porque conhece, em tempo de compilação, o tipo do receptor e tipo dos parâmetros da mensagem.

A sobrecarga de operadores e identificadores de métodos pode nos ajudar a tornar um programa mais claro e, portanto, mais fácil de compreender e compartilhar com outros desenvolvedores. Contudo, a característica mais notável do polimorfismo em sistemas de programação orientada a objetos é o *acoplamento dinâmico* (ou junção posterior). Nesse caso, o tipo do objeto só é conhecido durante a execução do programa e, portanto, o compilador não pode decidir, em tempo de compilação, qual método de qual classe acoplar à mensagem.

Acoplamento dinâmico e herança são conceitos associados. Tomemos como exemplo o Programa 8.8, no qual implementamos o destrutor de um modelo gráfico.

O método é bastante simples. Na linha 3 testamos se há primitivos na lista de primitivos do modelo. Se houver, invocamos o destrutor para o primitivo apontado por `Primitives` na linha 4. O que acontece, então? Vamos supor que o primitivo apontado por `Primitives` seja uma linha. Na definição da classe `gLine`, Programa 8.6, não notamos a declaração de um destrutor. Sem problemas, pois `gLine` deriva de

```
1   gModel::~gModel()
2   {
3     while (Primitives)
4       delete Primitives;
5   }
```

Programa 8.8: Destrutor de modelo gráfico.

`gPrimitive` e em `gPrimitive`, Programa 8.5, temos um destrutor (caso contrário o compilador forneceria um). A linha usa, portanto, a implementação do destrutor declarado na classe base `gPrimitive`, o qual envia uma mensagem para o modelo "pai" solicitando a remoção do primitivo da lista de primitivos do modelo. O modelo "pai", em resposta à mensagem, executa o método `gModel::Remove()`. Como resultado, a linha é removida da lista de primitivos do modelo e o ponteiro `Primitive` passa a apontar para o próximo primitivo da lista (verifique no Programa 8.4). A seguir, o operador **`delete`** completa sua função e libera a área de memória dinâmica ocupada pela linha (supomos que os primitivos sejam criados com o operador **`new`**).

Vamos imaginar, agora, que o próximo primitivo apontado por `Primitives` seja outro modelo gráfico. Na classe `gModel` definimos um destrutor (que, nesse instante, está sendo executado para alguma instância da classe). Se quisermos destruir o modelo `Primitives` corretamente, temos que invocar `gModel::~gModel()` (o qual automaticamente invocará `gPrimitive::~gPrimitive()`. Note, porém, que `Primitives` é um ponteiro da classe `gPrimitive`. Em tempo de compilação, o compilador não tem como determinar qual será a classe do objeto para o qual, durante a execução do programa, `Primitives` apontará (pode ser qualquer objeto cuja classe seja derivada direta ou indiretamente de `gPrimitive`). Portanto, a decisão de qual método invocar deve ser tomada em tempo de *execução*, em função do tipo de objeto apontado por `Primitives`. Isso é o que chamamos de acoplamento dinâmico.

Em C++, a especificação de quais métodos de uma classe poderão ser acoplados dinamicamente é feita seletivamente, com o uso do especificador de tipo **`virtual`**. Na linha 5 do Programa 8.2 declaramos o destrutor de `gPrimitive` como **`virtual`**. Um método declarado como **`virtual`** em uma classe base pode ser *sobrecarregado* nas classes derivadas, possibilitando que uma especialização forneça sua própria versão para o método e, mais importante, que essa versão possa ser acoplada em tempo de execução.

O polimorfismo em programação orientada a objetos elimina a necessidade do uso de construções **`switch-case`**. O código do Programa 8.8 é muito claro e legível e funciona, sem modificações, para qualquer tipo de primitivo gráfico cuja classe seja derivada da classe `gPrimitive`.

Há um custo para o polimorfismo, no entanto. Em C++, temos, para cada classe polimórfica, uma *tabela de funções virtuais* (cujas entradas contêm os endereços de todas as funções virtuais declaradas na classe) e, para cada instância de uma classe polimórfica, um *ponteiro para a tabela de funções virtuais* da classe. No Capítulo 3 construímos nossa própria tabela de funções virtuais para um primitivo gráfico genérico, a estrutura `gPrimitiveType` do Programa 3.12. Em C++, o compilador faz todo o trabalho para nós.

8.4 Sumário

Nesse Capítulo apresentamos os principais conceitos e propriedades da programação orientada a objetos.

Definimos objeto como sendo uma coleção de atributos e de métodos que caracterizam a estrutura e o comportamento de uma entidade concreta ou abstrata. Objetos com a mesma estrutura e o mesmo comportamento pertencem a mesma classe de objetos. Um objeto é uma instância de uma classe.

Em um modelo de computação orientado a objetos, a execução de um programa é dada em termos de objetos que se comunicam através do mecanismo de troca de mensagens. Uma mensagem é definida por um objeto receptor, um seletor da mensagem e um conjunto de parâmetros. A execução de um método de uma classe, por parte de um objeto da classe que recebe uma mensagem, é chamado acoplamento mensagem/método.

Em uma definição de classe, os atributos que caracterizam o estado interno de um objeto são encapsulados, sendo acessíveis somente aos métodos da classe. Dessa forma, eventuais alterações na implementação de uma classe não acarretam, necesariamente, alterações nos programas que utilizam a classe (desde que a interface da classe não seja modificada).

O mecanismo de herança permite a construção de hierarquias de generalizações/especializações em um sistema orientado a objetos, sendo que uma classe mais especializada herda os atributos e os métodos da(s) classe(s) mais genérica(s). A(s) classe(s) mais genérica(s) é(são) denominada(s) classe(s) base(s), ou superclasse(s), e a classe mais especializada é chamada classe derivada, ou subclasse. A herança possibilita, muito acentuadamente, a reutilização de código, e juntamente com o polimorfismo, são responsáveis pela "mágica" dos programas orientados a objetos.

Parte Dois

OBJECT STRUCTURAL WORKBENCH

CAPÍTULO 9

Construindo Aplicações com OSW

9.1 Introdução

Na primeira parte do texto apresentamos os fundamentos matemáticos e computacionais utilizados no desenvolvimento de nosso Modelador Estrutural Orientado a Objetos. Nessa segunda parte, esses fundamentos serão colocados todos juntos na especificação das *bibliotecas de classes* de OSW — *Object Structural Workbench*. OSW é um *toolkit* destinado ao desenvolvimento orientado a objetos de programas de análise e de visualização de modelos estruturais. Chamaremos esses programas de análise e de visualização de *programas de modelagem*, ou *aplicações de modelagem*. Nesse Capítulo, apresentaremos uma visão geral das classes de OSW e mostraremos como utilizá-las para a construção de uma aplicação de modelagem. No Capítulo 10, descreveremos as interfaces pública e protegida das principais classes de objetos do *toolkit*.

Nos referiremos às aplicações de modelagem construídas com as classes de OSW como aplicações OSW, ou aplicações em OSW, ou, ainda, como aplicações baseadas em OSW. Uma aplicação OSW é um programa para Windows NT [97], escrito em C++. Na Seção 9.2 definiremos as características, os principais elementos de interface e os tipos de arquivos que fazem parte do projeto de uma aplicação OSW. Na Seção 9.3, apresentaremos uma classificação das classes mais importantes do sistema.

Nas seções seguintes, tentaremos explicar, pelo menos em seus aspectos gerais, como construir uma aplicação de modelagem com as classes de objetos de OSW. Escolhemos, como exemplo, uma aplicação de análise e de visualização de cascas elásticas pelo método dos elementos finitos chamada OSW-Shell. A princípio, a tarefa pode parecer complicada porque as bibliotecas de classes de OSW possuem cerca de uma centena de classes de objetos. Nesse Capítulo, faremos referência a mais de oitenta dessas classes. Além disso, os arquivos `.h` e `.cpp` de OSW-Shell somam mais de 300KB de código-fonte. Sugerimos que o material apresentado nesse Capítulo seja lido conjuntamente com as descrições do Capítulo 10.[1]

[1] OSW está disponível no Departamento de Engenharia de Estruturas da EESC-USP.

9.2 O que é uma Aplicação de Modelagem?

No Capítulo 2 vimos que um sistema gráfico de modelagem pode ser conceitualmente descrito por três componentes: o programa de aplicação, a base de dados da aplicação e o sistema gráfico. A função do programa de aplicação é criar, armazenar e recuperar informações da base de dados da aplicação, além de capturar os eventos de entrada do usuário e produzir visões dos objetos da base de dados da aplicação. Essas visões contêm a descrição geométrica do objeto a ser visualizado e atributos que controlam sua aparência. O sistema gráfico transforma as descrições geométricas e de aparência dos objetos da base de dados da aplicação em imagens desses objetos, as quais são enviadas ao *hardware* gráfico.

Denominaremos o programa de aplicação de um sistema gráfico de modelagem de *aplicação de modelagem*. Uma aplicação de modelagem, portanto, é um dos componentes de um sistema gráfico de modelagem. As aplicações de modelagem baseadas em OSW são programas de 32 bits que executam no sistema operacional Windows NT, o qual fornece, também, o sistema gráfico descrito anteriormente. A Figura 9.1 mostra a interface de um programa construído com as classes de OSW. Vejamos os principais elementos da interface.

Janela principal. O principal elemento de interface de uma aplicação de modelagem (e da maioria das aplicações para Windows) é a janela principal da aplicação. A janela principal contém a barra de título da aplicação, a barra de menus e a *área cliente*. Na área cliente da janela principal são exibidas todas as demais janelas da interface da aplicação. Essas janelas são chamadas, no jargão do Windows, *janelas-filhas* da janela principal, discutidas a seguir.

Barra de *status*. A barra de *status* é uma janela-filha da janela principal localizada, inicialmente, logo abaixo da barra de menus. A barra de *status* pode ser utilizada para exibição de algumas informações do estado da aplicação durante sua execução. Por exemplo, o nome do comando sendo executado e as coordenadas do cursor (veja a Seção 9.8) são mostradas na barra de *status*.

Janela de comandos. A maioria dos comandos de uma aplicação podem ser executados a partir de sua seleção no menu da aplicação. No entanto, é desejável, em programas gráficos, que o usuário tenha a possibilidade de fornecer o comando e os parâmetros do comando textualmente. Uma vez familiarizado com a sintaxe dos comandos, essa opção pode ser mais ágil que a navegação nos menus. Além disso, um comando pode requerer os parâmetros para sua execução através de mensagens ao usuário. Essas mensagens são exibidas na janela de comandos. Inicialmente, a janela de comandos de uma aplicação é localizada na parte inferior da janela principal. O objetivo é evitar o congestionamento da parte central da área cliente da janela principal. A interface de um programa gráfico deve dirigir a atenção do usuário aos modelos sendo contruídos e às imagens dos modelos, e não à própria interface.

Janela de resultados de análise. As janelas de resultado de análise são janelas-filhas da janela principal que exibem, em formato de planilha, os resultados da análise numérica de um modelo mecânico pertencente à base de dados da aplicação. Uma janela de resultados de análise é definida por uma coleção de *colunas*. Cada coluna da janela exibe uma informação específica do modelo mecânico,

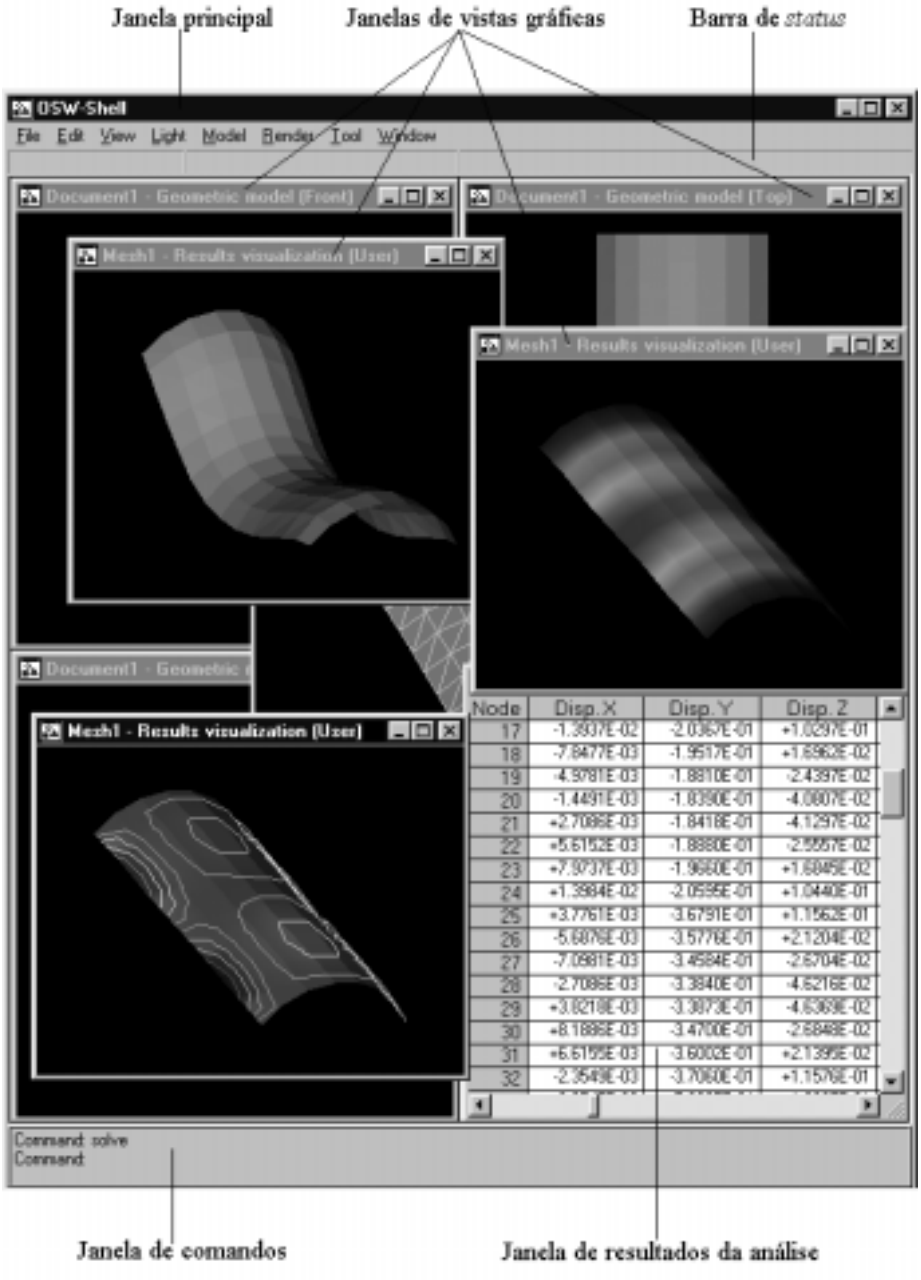


Figura 9.1: Interface de uma aplicação de modelagem em OSW.

por exemplo, as coordenadas dos vértices ou os deslocamentos transversais dos vértices. As informações do i-ésimo vértice do modelo mecânico são exibidas na i-ésima linha da janela. A intersecção de uma linha com uma coluna define uma *célula* da janela.

Janelas de vistas gráficas. O restante da área cliente da janela principal é ocupada pelas janelas de vistas gráficas da aplicação. As janelas de vistas mostram as vistas gráficas tridimensionais dos objetos da base de dados da aplicação e são responsáveis pela execução dos comandos do usuário (veja a Seção 9.8). Podemos ter várias janelas de vistas abertas ao mesmo tempo, mostrando vistas distintas dos mesmos objetos ou vistas distintas de objetos distintos.

A construção de um programa para Windows não é uma tarefa das mais imediatas. Precisamos aprender como criar e destruir janelas, como desenhar em janelas, como o Windows captura os eventos de entrada do usuário, etc. Temos de dominar as funções básicas da API (*Application Programming Interface*) do Windows, tais como `GetMessage()`, `DispatchMessage()`, `TranslateMessage()`, `RegisterClass()`, `CreateWindow()`, `DestroyWindow()`, `BeginPaint()`, `SelectObject()`, `MoveTo()`, `LineTo()`, `EndPaint()`, etc. Vamos simplificar o processo. Utilizaremos como fundação para a construção dos elementos de interface de nossos programas de aplicação a *Object Windows Library*, ou OWL.[2] As janelas, caixas de diálogo, botões e menus de uma aplicação OSW são objetos cujas classes são derivadas das classes de objetos da OWL. No Capítulo 10 faremos algumas referências a algumas classes da OWL, mas não discutiremos essas classes no texto. Os manuais da OWL e a documentação *on-line* do *software* são bastante detalhados.

Ao utilizarmos a OWL, estamos sendo coerentes e nos aproveitando dos benefícios da programação orientada a objetos: ao invés de inventarmos novamente tudo que precisamos, vamos reaproveitar código e especializar objetos. Contudo, dominar os componentes de uma biblioteca de classes constituída de dezenas de classes para "facilitar" o trabalho de construção de uma aplicação é uma tarefa, inicialmente, tão complicada quanto aprender a dominar o Windows sem a OWL (na verdade, precisamos conhecer o Windows com ou sem a OWL). À medida que formos ficando mais familiarizados com suas classes, no entanto, realmente constatamos que a biblioteca facilita o trabalho de escrever uma aplicação para Windows. (Com as bibliotecas de OSW talvez não seja diferente: a princípio, compreender algumas classes e suas relações pode ser algo confuso, mas, posteriormente, esperamos que a utilização do *toolkit* possa se mostrar um pouco mais atrativa.)

9.2.1 Montando o Projeto de uma Aplicação

Uma aplicação C++ para Windows, tipicamente, é constituída dos seguintes tipos de arquivos:

`.cpp` Os arquivos `.cpp` contêm o código-fonte da aplicação. Definiremos, em um arquivo `.cpp`, a implementação das funções e classes de objetos de uma aplicação. As declarações de classes e macros e a implementação de funções **`inline`** e de classes paramétricas (**`template`**) serão feitas em arquivos de cabeçalho, os quais possuem a extensão `.h`. Os arquivos `.cpp` incluem os arquivos de cabeçalho através da diretiva de pré-processamento do C (e C++) `#include`. Em um dos arquivos `.cpp` da aplicação, devemos definir a função principal da aplicação (veja a Seção 9.4).

`.rc` Ícones, menus, cursores, barras de rolagem, e caixas de diálogo são exemplos de *recursos* comuns à maioria dos programas para Windows. A definição dos recursos de uma aplicação, se a aplicação utilizar quaisquer recursos, é feita em arquivos do tipo `.rc`. As aplicações OSW utilizam pelo menos um arquivo de recursos chamado `osw.rc`.

`.lib` Um arquivo `.lib` é uma biblioteca de ligação estática, contendo código já compilado que será ligado ao código dos arquivos `.cpp` para formação do

[2] OWL é marca registrada da Borland International.

arquivo `.exe`. O Windows também permite a *ligação dinâmica* de arquivos do tipo `.dll` [88]. (O código-objeto das classes de OSW está contido na biblioteca de ligação dinâmica `osw.dll`.) Uma aplicação em OSW utiliza, no mínimo, uma biblioteca de ligação estática chamada `osw.lib`.

`.def` Um arquivo `.def` é um arquivo de definições do Windows e contém algumas informações a respeito do programa executável. Definimos um arquivo `.def` chamado `osw.def` para ser utilizado pelas aplicações OSW.

O programa executável (`.exe`) é gerado com a compilação dos arquivos `.cpp`, ligação estática dos arquivos `.lib` e montagem dos recursos definidos nos arquivos `.rc`. Todos esses arquivos, mais o arquivo `.def`, constituem o que chamamos de *projeto* de uma aplicação.

Utilizamos o ambiente de desenvolvimento integrado (IDE — *integrated development environment*) do Borland C++ 5.0 para gerar o código executável de uma aplicação porque o IDE fornece alguns recursos visuais que facilitam a definição dos componentes do projeto de uma aplicação. Não entraremos em detalhes de como utilizar o IDE. Qualquer outro sistema de compilação pode ser utilizado para a geração do código executável.

9.2.2 Aplicações de Modelagem Orientadas a Objetos

Uma aplicação de modelagem OSW é definida por objetos. (Utilizamos o termo livremente no texto, interpretando seu significado de acordo com a disciplina, mas agora estamos nos referindo a objetos do ponto de vista da programação orientada a objetos.) A janela principal, a janela de comandos, as janelas de vistas, a base de dados da aplicação e a própria aplicação são objetos, instâncias de uma classe. Conhecer a funcionalidade e o relacionamento das classes de objetos de OSW é essencial para construirmos uma aplicação de modelagem orientada a objetos.

Basicamente, podemos utilizar uma classe de três maneiras:

1. *Construindo uma instância da classe.* No Capítulo 8 ilustramos como construir objetos de uma classe em C++, estática ou dinamicamente. Para algumas classes de OSW, isso é tudo que temos a fazer. Objetos da classe `tCamera`, por exemplo, podem ser construídos e utilizados prontamente em uma aplicação de modelagem.

2. *Derivando uma nova classe da classe.* Outras classes de OSW, contudo, podem ou devem ser especializadas em classes derivadas, as quais herdam os atributos e os métodos das classes bases (em C++ a visibilidade dos atributos e métodos herdados por uma classe derivada é definida pelas palavras reservadas **`public`**, **`protected`** e **`private`**). Uma classe derivada pode especializar a classe base (1)adicionando novos atributos, (2)adicionando novos métodos e (3)*sobrecarregando* métodos virtuais das classes bases. Um objeto da classe `tFiniteElement`, por exemplo, representa um elemento finito genérico de um modelo de decomposição por células. A interface da classe define métodos virtuais para o cálculo da matriz de rigidez e do vetor de esforços nodais equivalentes de um elemento finito (veja a Seção 9.12). A classe `tFE3NShell` é uma classe derivada de `tFiniteElement`. `tFE3NShell` sobrecarrega os métodos de cálculo da matriz de rigidez e do vetor de esforços equivalentes da classe base e adiciona atributos e

métodos específicos de um elemento de casca (espessura e rotação da matriz de rigidez para o sistema global, por exemplo).

3. *Associando a classe a outra classe.* Além da especialização, há outros tipos de relacionamento entre classes. Um objeto de uma classe pode conter um objeto de uma outra classe ou compartilhar um objeto de uma outra classe. Um objeto da classe `tMesh`, por exemplo, contém uma coleção de objetos da classe `tCell`. Cada objeto da classe `tCell` compartilha um objeto da classe `tMaterial`.

Antes de descrevermos os passos de criação de uma aplicação, apresentaremos uma visão geral das classes de OSW.

9.3 Visão Geral das Classes de OSW

Nessa Seção apresentaremos uma descrição sucinta das principais classes de objetos de OSW. Para maior clareza, as classes foram separadas em grupos. Tentamos classificar as classes de acordo com sua funcionalidade, mas há classes que podem pertencer a mais de um grupo.

Classes de Aplicações

Uma aplicação de modelagem OSW é um objeto de uma classe derivada da classe `tApplication`. `tApplication` oferece métodos virtuais para inicialização da aplicação e da janela principal da aplicação e gerenciamento da base de dados da aplicação.

Classes de Janelas

Os objetos das classes de janelas de OSW representam os elementos de interface de uma aplicação de modelagem, conforme visto na Seção anterior. As principais classes de janela de OSW são:

- `tMainWindow`. Um objeto da classe `tMainWindow` representa a janela principal da aplicação. Normalmente, não precisamos especializar a funcionalidade de `tMainWindow`.

- `tStatusBar`. Um objeto da classe `tStatusBar` representa a barra de *status* da aplicação. A interface da classe define métodos para adição e remoção de *gadgets* na barra de *status* da aplicação. A barra de *status* padrão contém dois *gadgets* nos quais são exibidos o nome do comando sendo executado e as coordenadas do cursor.

- `tCommandWindow`. A janela de comandos é a janela da aplicação na qual são digitados os comandos e seus parâmetros e exibidas as mensagens dos comandos executados pela aplicação. A janela de comandos é um objeto da classe `tCommandWindow`.

- `tDataWindow`. As janelas de dados são objetos da classe `tDataWindow` (as janelas de resultados de análise são janelas de dados). A interface de `tDataWindow` define métodos de adição e remoção de colunas na janela de dados. Cada coluna de uma janela de dados é um objeto da classe `tColumn` responsável pela exibição de um tipo de informação na janela.

- `tViewWindow`. Um objeto da classe `tViewWindow` é uma janela-filha MDI (*Multiple Document Interface*) do Windows que contém em sua área cliente uma vista gráfica dos objetos da base de dados da aplicação. A funcionalidade de uma vista é definida por um objeto de uma classe derivada da classe `tView`.

- `tLookupTableWindow`. Um objeto da classe `tLookupTableWindow` representa uma janela de exibição da tabela de cores utilizada em um mapa de cores. Usualmente, não é preciso derivar novas classes de `tLookupTableWindow`.

Uma aplicação OSW pode utilizar *caixas de diálogo* pré-definidas nas bibliotecas de classes. Alguns exemplos são:

- `tColorDialog`. Uma caixa de diálogo de cores permite a seleção de uma cor definida no modelo de cores RGB. A cor selecionada pode, por exemplo, ser a cor de uma luz de uma cena.

- `tCreateViewDialog`. Uma caixa de diálogo de criação de vista permite o ajuste dos parâmetros da câmera de uma vista que desejamos criar.

- `tOpenFileDialog` e `tSaveFileDialog`. Essas caixas de diálogo possibilitam a seleção de um nome de arquivo manipulado por uma aplicação.

- `tInputDialog`. Uma caixa de diálogo de entrada contém um comando de edição que permite a entrada de expressões matemáticas representando vetores, números reais ou números inteiros.

Classes de Documento de uma Aplicação

Uma base de dados de uma aplicação é um objeto de uma classe derivada da classe `tDocument`. `tDocument` oferece métodos virtuais para recuperação e armazenamento de objetos de uma base de dados da aplicação. As bases de dados de uma aplicação OSW são definidas em termos de coleções de *modelos* e de *cenas*.

Classes de Modelos

Um modelo é um objeto de uma classe derivada da classe `tModel`. A interface de `tModel` oferece métodos virtuais puros para acesso aos componentes topológicos necessários à visualização da geometria de um modelo. Um método virtual puro **`void method()`** é definido pela seguinte sintaxe:

```
...
virtual void method() = 0;
...
```

Uma classe que possui métodos virtuais puros é chamada *classe abstrata*. As classes derivadas de uma classe abstrata *devem* sobrecarregar os métodos virtuais puros, senão também serão abstratas. Uma classe abstrata não possui instâncias, ou seja, não podemos construir objetos de uma classe abstrata. `tModel` é abstrata.

As principais classes (não-abstratas) de modelos de OSW são:

- `tGraphicModel`. Um objeto da `tGraphicModel` é um modelo gráfico tal como definido no Capítulo 3 (a classe é uma extensão orientada a objetos da estrutura `gModel`). Um primitivo de um modelo gráfico é um objeto de uma classe derivada da classe `tPrimitive`. `tGraphicModel` é derivada da classe abstrata `tModel` e de `tPrimitive` (um modelo gráfico *é* um modelo e, ao mesmo tempo, um primitivo gráfico).

- `tShell`. Um objeto da classe `tShell` é um modelo geométrico de cascas (a classe é uma extensão da estrutura `bModel`).

- `tSolid`. Um objeto da classe `tSolid` é um modelo geométrico de sólidos (a classe é uma extensão da estrutura `sModel`).

- `tMesh`. Um objeto da classe `tMesh` é um modelo de decomposição por células (a classe é uma extensão orientada a objetos da estrutura `cModel`). Um vértice de um modelo de decomposição por células é um objeto da classe `tNode`. Uma célula do modelo é um objeto de uma classe derivada da classe abstrata `tCell`.

Classes de Cenas de um Documento

Uma cena é um objeto da classe `tScene`. A interface da classe oferece métodos para adição e remoção de *atores*, *luzes* e *câmeras* de uma cena. Atores de uma cena são objetos da classe `tView`, luzes são objetos da classe `tLight` e câmeras são objetos da classe `tCamera`.

Classe de Vistas de uma Cena

Uma cena possui uma coleção de *vistas*. Uma vista é um objeto de uma classe derivada da classe `tView`, responsável pelo processamento de comandos de entrada e exibição, em uma janela de vista, de uma imagem da cena. A interface de `tView` define métodos de entrada de argumentos de comandos, tais como pontos, ângulos e distâncias e de controle de parâmetros de visualização. Uma classe derivada de `tView` deve definir seu conjunto próprio de comandos.

Classes de *Renderer*

A imagem da cena é gerada por um objeto de uma classe derivada da classe abstrata `tRenderer`. Há duas classes (não-abstratas) de *rendering* em OSW:

- `tScanner`. Um objeto da classe `tScanner` é um *renderer* baseado no algoritmo de linhas de varredura (*scanlines*). A interface de `tScanner` define métodos para síntese de imagens fio-de-arame, remoção de superfícies escondidas e tonalização "*flat*" ou de Gouraud de modelos geométricos.

- `tRayTracer`. Um objeto da classe `tRayTracer` é um *renderer* baseado no algoritmo de traçado de raios descrito no Capítulo 7.

Classes de Fontes

Fontes são processos geradores de modelos. Em OSW, um processo fonte é um objeto de uma classe derivada da classe abstrata `tSource`. Objetos das classes `tCone`, `tSphere`, `tCube`, `tCylinder` e `tSolidSweeper` são fontes de modelos de sólidos. Objetos das classes `tCylindricalShell` e `tShellSweeper` são fontes de modelos de cascas. Objetos da classe `tMeshSweeper` são fontes de modelos de decomposição por células. Objetos da classe `tSolidReader` são leitores de modelos de sólidos; objetos da classe `tShellReader` são leitores de modelos de cascas; e objetos da classe `tMeshReader` são leitores de modelos de decomposição por células.

Classes de Filtros

Filtros são processos de transformação de modelos. Em OSW, um filtro é um objeto de uma classe derivada da classe abstrata `tFilter`. Os principais filtros de OSW são:

- `t2DMeshGenerator`. Um objeto da classe `t2DMeshGenerator` é um filtro que toma como entrada um modelo de cascas ou de sólidos e transforma o modelo em um modelo de decomposição por células. As células do modelo de saída são triângulos ou quadriláteros (dimensão topológica 2).

- `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, como exemplificado no Capítulo 7.

- `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. `tScalarExtractor` é um filtro de transformação de atributos que "extrai" os valores (escalares) dos graus de liberdade nodais do modelo para visualização.

- `tContourFilter`. Um objeto da classe `tContourFilter` é um filtro que toma como entrada um modelo de decomposição por células e gera como saída um modelo gráfico contendo os isopontos, isolinhas e isosuperfícies do modelo de entrada.

- `tVectorExtractor`. Um objeto da classe `tVectorExtractor` é um filtro que toma como entrada um modelo de decomposição por células e gera como saída o mesmo modelo de entrada. `tVectorExtractor` é um filtro de transformação de atributos que "extrai" os valores (vetoriais) dos graus de liberdade nodais do modelo para visualização.

- `tWarpFilter`. 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.

Classes de Mapeadores

Mapeadores são processos sumidouros de modelos. Em OSW, um mapeador é um objeto da classe `tMapper`. Um mapeador compartilha com outros mapeadores um

objeto da classe `tLookupTable`, uma *tabela de cores* a partir da qual será gerado o mapa de cores dos modelos de entrada do mapeador. Todo ator de uma cena possui um mapeador que controla a geração de imagens do modelo geométrico do ator.

Classes de Elementos Estruturais

Um elemento finito é um objeto de uma classe derivada da classe `tFiniteElement`. A interface da classe define métodos virtuais para o cálculo da matriz de rigidez de um elemento e para o cálculo do vetor de esforços nodais equivalentes de um elemento finito genérico. `tFiniteElement` deriva virtualmente da classe abstrata `tCell`.[3] Um elemento finito de casca é um objeto da classe `tFE3NShell`, derivada das classes `tFiniteElement` e `t3N2DCell`. Devido ao mecanismo de derivação virtual, os objetos da classe `tFE3NShell` possuem somente uma cópia de `tCell`.

Um elemento de contorno é um objeto de uma classe derivada da classe base `tBoundaryElement`. A interface da classe define métodos virtuais para o cálculo das matrizes de influência de um elemento de contorno genérico. Como `tFiniteElement`, `tBoundaryElement` é derivada virtualmente de `tCell`. Os elementos de contorno com 4 e 8 nós definidos no Capítulo 6 são, respectivamente, objetos das classes `tBE4NQuad` e `tBE8NQuad`.

Classes de Analisadores

Um processo de análise numérica em OSW é representado por um objeto de uma classe derivada da classe abstrata `tSolver`. A interface de `tSolver` define métodos virtuais para inicialização do processo de análise, verificação dos dados do modelo mecânico, construção do sistema de equações lineares (objeto de uma classe derivada da classe abstrata `tLinearSystem`), montagem do sistema de equações e término do processo de análise. Há duas classes (não-abstratas) de analisadores em OSW:

- `tFESolver`. Um objeto da classe `tFESolver` é um analisador de estruturas tridimensionais elastostáticas pelo método dos elementos finitos.
- `tBESolver`. Um objeto da classe `tBESolver` é um analisador de estruturas tridimensionais elastostáticas pelo método dos elementos de contorno.

Classificamos como "classes auxiliares" outras classes que não se enquadram nos grupos apresentados anteriormente. Descreveremos essas e as demais classes de OSW no Capítulo 10.

9.4 Definindo a Aplicação

Uma aplicação de modelagem OSW é um objeto da classe `tApplication` (ou, mais comumente, um objeto de uma classe derivada de `tApplication`). Uma aplicação de modelagem é responsável pela criação, armazenamento e recuperação de informações na base de dados da aplicação. Uma aplicação genérica efetua algumas tarefas relacionadas à criação e manutenção de uma base de dados genérica, mas cada aplicação em particular deve conhecer sua base de dados específica.

[3] Veja STROUSTRUP [110, página 206] para uma discussão sobre classes bases virtuais.

Para construirmos uma aplicação de modelagem, devemos:

Passo 1 Criar um arquivo `.cpp` e incluir o arquivo de cabeçalho `oswapp.h`, o qual contém a definição da classe `tApplication`.

Passo 2 Definir no arquivo `.cpp` a função principal da aplicação, chamada `OSWMain()`. Na função principal, construir um objeto da classe `tApplication`.

Passo 3 Enviar a mensagem `Run()` ao objeto de aplicação.

OSW-Shell

Para ilustrar o processo de criação de uma aplicação OSW, consideraremos, como exemplo, um programa de análise de cascas elásticas pelo método dos elementos finitos e de visualização dos modelos e dos resultados de análise. Denominaremos o programa de OSW-Shell. Vejamos, resumidamente, a sentença do problema.

> As cascas a serem analisadas podem ser definidas por modelos geométricos de cascas, a partir dos quais serão derivados os modelos mecânicos de análise pelo método dos elementos finitos. O modelo mecânico pode também ser definido diretamente, sem necessidade de derivar de um modelo geométrico. As definições dos modelos podem ser feitas interativamente nas janelas de vistas gráficas da aplicação ou podem ser dadas por comandos escritos em um arquivo texto. Após a análise da estrutura, os resultados devem ser visualizados com mapa de cores e isolinhas dos componentes de deslocamentos e esforços, e com exibição da estrutura deformada. Os dados dos modelos podem ser armazenados e recuperados de arquivos em disco.

Os arquivos necessários para geração do arquivo `.exe` da aplicação são mostrados na janela do gerente de projeto do IDE, Figura 9.2. Os arquivos `osw.lib`, `osw.rc` e `osw.def` já foram comentados anteriormente. O arquivo `shellapp.cpp`, mostrado no Programa 9.1, contém a função principal da aplicação.

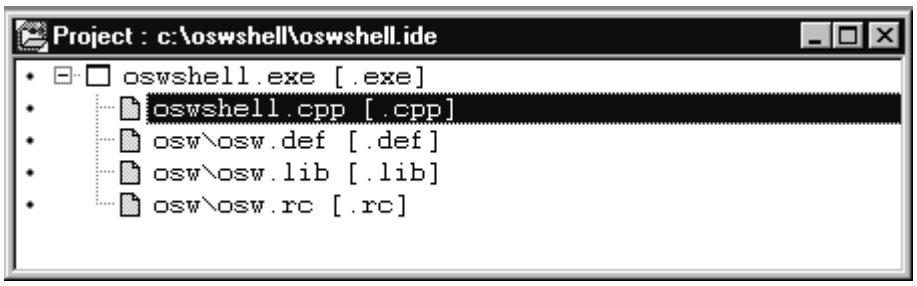


Figura 9.2: Projeto de uma aplicação mínima OSW.

```
1   #include "oswapp.h"
2   int OSWMain(int, char**)
3   {
4     return tApplication("OSW-Shell").Run();
5   }
```

Programa 9.1: Construindo uma aplicação em OSW.

Na linha 1 incluímos o arquivo de cabeçalho `oswapp.h`. A função principal da aplicação, definida na linha 2, toma como argumentos um inteiro e um vetor de ponteiros de caracteres que representam, respectivamente, o número e o nome dos parâmetros passados à aplicação na linha de comando. Por enquanto, não utilizaremos esses argumentos. Na linha 4 construímos um objeto da classe `tApplication`. O construtor da classe toma como argumento um ponteiro de caracteres que define o nome da aplicação. Em seguida, enviamos a mensagem `Run()` para a aplicação. O método `Run()` inicializa a janela principal da aplicação e deixa a aplicação pronta para receber os comandos do usuário. `Run()` retorna um inteiro que indica o *status* da execução da aplicação. Esse valor é utilizado como valor de retorno da função principal. O resultado da execução da aplicação é mostrado na Figura 9.3.

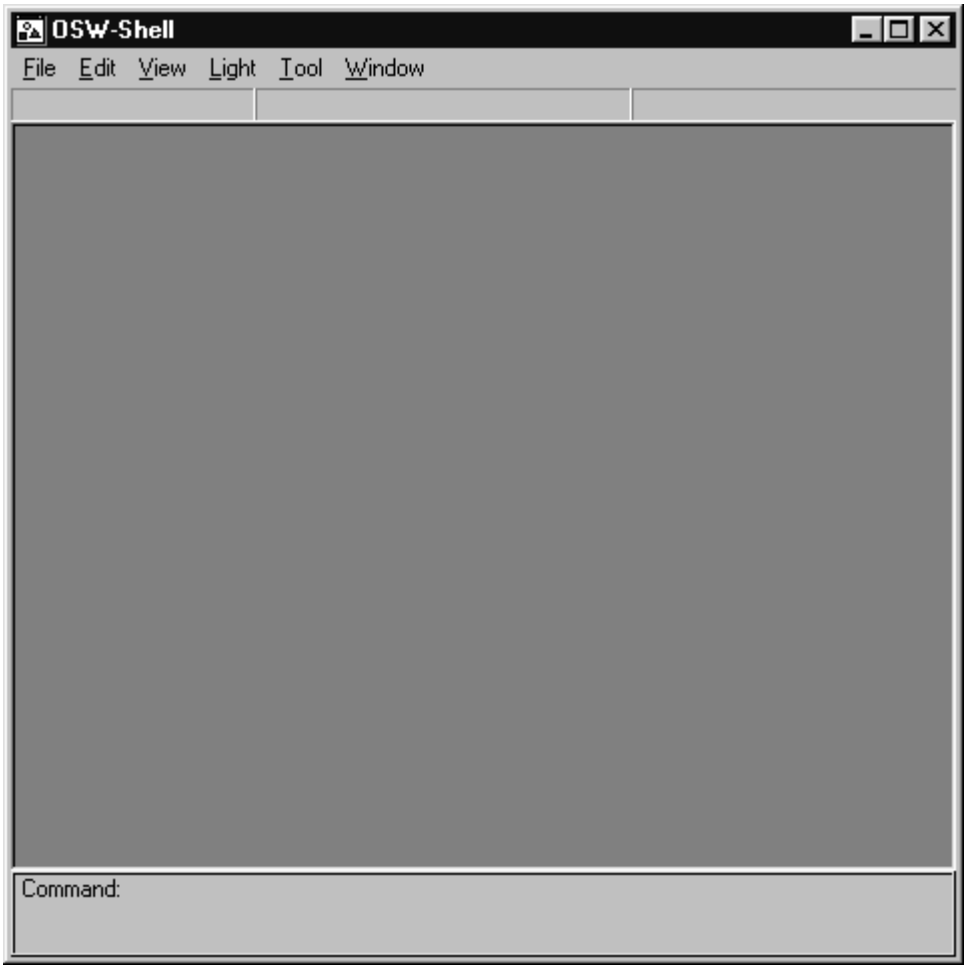


Figura 9.3: Construindo uma aplicação OSW.

Nossa primeira aplicação foi simples de construir. Com apenas uma linha de código na função principal temos uma programa com menu, barra de *status* e uma janela de comandos. No menu, temos comandos para criar janelas de vistas gráficas de uma cena; criar, modificar e eliminar luzes de uma cena; acessar ferramentas comuns tais como o *editor de materiais* (descreveremos a classe `tMaterialEditor` no Capítulo 10); e criar arquivos, abrir arquivos já existentes, salvar arquivos e encerrar a execução da aplicação. Mas quais tipos de arquivos? Qual é, exatamente, a funcionalidade de nossa primeira aplicação?

Quando enviamos a mensagem `Run()` a um objeto de aplicação, o objeto executa o método `InitApplication()`, responsável pela inicialização da aplicação. O método é declarado como virtual na classe `tApplication` e, portanto, pode ser sobrecarregado em uma classe derivada de `tApplication`. Por exemplo, podemos reescrever `InitApplication()` em uma classe derivada para tratarmos os parâmetros da linha de comando, passados como argumentos da função principal da aplicação (faremos isso a seguir). O método `InitApplication()`, por sua vez, executa o método virtual `InitMainWindow()`, responsável pela inicialização da janela principal da aplicação (se sobrecarregarmos o método de inicialização da aplicação, devemos executar `tApplication::InitApplication()` para inicializar a janela principal.) Normalmente, não precisamos nos preocupar com `InitMainWindow()`, a não ser que queiramos modificar a janela principal da aplicação.

Para definirmos uma aplicação de modelagem específica, devemos:

Passo 1 Declarar uma classe derivada da classe `tApplication`.

Passo 2 Sobrecarregar, na nova classe, o método virtual `ConstructDoc()`, responsável pela construção da base de dados manipulada pela aplicação. *Podemos* sobrecarregar também outros métodos virtuais de `tApplication`, mas *devemos* sobrecarregar `ConstructDoc()`.

O Programa 9.2 mostra o arquivo de cabeçalho `shellapp.h`, o qual contém a definição da classe `tShellApp`, derivada de `tApplication`.

```
1   #include "oswapp.h"
2   class tShellApp:public tApplication
3   {
4     public:
5       // Constructor
6       tShellApp(int argc = 0, const char** argv = 0):
7         tApplication("OSW-Shell", argc, argv),
8         DocPathData("Shell files (*.shl)|*.shl")
9       {}
10
11    private:
12      void       InitApplication();
13      tDocument* ConstructDoc(tApplication&);
14
15  }; // tShellApp
```

Programa 9.2: Definindo a classe de aplicação de cascas.

Na linha 6 do programa declaramos o construtor da classe `tShellApp`. Os parâmetros do construtor são um inteiro `argc` e um vetor de cadeias de caracteres `argv`. Na linha 7 inicializamos a classe base `tApplication` com o nome da aplicação e com `argc` e `argv`. Na linha 8 inicializamos o atributo da aplicação chamado `DocPathData` (veja a classe `tDocPathData` no Capítulo 10), herdado de `tApplication`. Nas linhas 12 e 13, respectivamente, declaramos os métodos `InitApplication()` e `ConstructDoc()`, ambos herdados de `tApplication`. A implementação da classe é feita no arquivo modificado `shellapp.cpp`, apresentado no Programa 9.3.

```
1   #include "shellapp.h"
2   void
3   tShellApp::InitApplication()
4   {
5     tApplication::InitApplication();
6     if (CmdCount > 1)
7       InitDoc(ConstructDoc(), CmdName[1], dfOpenDoc);
8   }
9
10  tDocument*
11  tShellApp::ConstructDoc(tApplication& app)
12  {
13    return new tShellDoc(app);
14  }
15
16  int OSWMain(int argc, char** argv)
17  {
18    return tShellApp(argc, argv).Run();
19  }
```

Programa 9.3: Construindo uma aplicação de cascas.

O método `tShellApp::InitApplication()`, definido a partir da linha 2 do Programa 9.3, verifica se há mais de um parâmetro na linha de comando (o primeiro parâmetro sempre é o nome da aplicação). Se o teste for positivo, o método executa o método virtual `InitDoc()`, herdado de `tApplication`, responsável pela inicialização da base de dados da aplicação, nesse caso, um objeto da classe `tShellDoc` (definiremos `tShellDoc` a seguir). Os atributos `CmdCount` e `CmdName` são herdados de `tApplication`. Na linha 5 executamos `tApplication::InitApplication()`, responsável pela inicialização da janela principal da aplicação.

O método `ConstructDoc()`, implementado a partir da linha 10, é o "construtor virtual" da base de dados de `tShellApp`. `tApplication::ConstructDoc()` simplesmente retorna um ponteiro nulo. Em `tShellApp`, sobrecarregamos o método para construir um documento do tipo `tShellDoc`. Agora que nossa aplicação já sabe como construir seu objeto de base de dados, podemos criar, abrir e fechar documentos da aplicação. Antes, porém, vamos definir a classe `tShellDoc`.

9.5 Definindo o Documento da Aplicação

Uma base de dados genérica de uma aplicação de modelagem é representada por um objeto da classe `tDocument`. A interface da classe `tDocument` contém um conjunto de métodos virtuais que define a funcionalidade comum a toda base de dados em OSW. Um documento armazena todos os *modelos* criados e manipulados pela aplicação. Um documento mantém, ainda, todas as *cenas* criadas e manipuladas pela aplicação. Definimos uma cena no Capítulo 7 como sendo uma coleção de *atores*, *luzes* e *câmeras*. Um documento pode conter várias cenas, cada cena com seu conjunto próprio de atores, luzes e câmeras. Os atores de uma cena compartilham os modelos armazenados no documento.

Para definir um documento específico de uma aplicação, devemos:

Passo 1 Declarar um classe derivada da classe `tDocument`.

Passo 2 Sobrecarregar, na nova classe, o método virtual `ConstructScenes()`, responsável pela inicialização das cenas do documento. Podemos sobrecarregar outros métodos virtuais de `tDocument`.

A declaração da classe `tShellDoc` é feita no arquivo de cabeçalho `shelldoc.h`, mostrado no Programa 9.4.

```
1   #include "oswdoc.h"
2   class tShellDoc:public tDocument
3   {
4     public:
5       // Constructor
6       tShellDoc(tApplication& app):
7         tDocument(app),
8         Shell(0)
9       {}
10      tShell* GetShell();
11      void    SetShell(tShell*);
12
13    private:
14      void    ConstructScenes(tDocument&);
15      bool    Open();
16      bool    Commit();
17
18      tShell* Shell;
19
20  }; // tShellDoc
21
22  inline tShell*
23  tShellDoc::GetShell()
24  {
25    return Shell;
26  }
```

Programa 9.4: Definindo o documento da aplicação de cascas.

Na linha 1 incluímos o arquivo de cabeçalho `oswdoc.h`, o qual contém a definição da classe base `tDocument`. Na linha 2, a classe `tShellDoc`, derivada de `tDocument`, é declarada. Na linha 6 definimos o construtor da classe. O construtor toma como parâmetro uma referência `app` para a aplicação que manipula o documento. Usamos `app` para inicializar `tDocument`. Na linha 8 inicializamos o modelo geométrico de cascas manipulado pelo documento, declarado na linha 18. (Embora o documento possua uma lista de modelos, utilizaremos o ponteiro `Shell` para acessar imediatamente o modelo geométrico da casca.) Na linha 14 declaramos o método `ConstructScenes()`, herdado de `tDocument`. A implementação do método é apresentada no Programa 9.5. (Para poupar espaço, não apresentaremos as implementações dos métodos `SetShell()`, `Open()` e `Commit()`, linhas 11, 15 e 16, respectivamente.)

```
1   #include "shelldoc.h"
2   void
3   tShellDoc::ConstructScenes(tDocument& doc)
4   {
5     tGeoScene* gs = new tGeoScene((tShellDoc&)doc);
6
7     if (Shell)
8       gs->AddActor(new tActor(Shell));
9   }
```

Programa 9.5: Inicializando as cenas do documento.

Na linha 1 do Programa 9.5 incluímos o arquivo de cabeçalho `shelldoc.h`, definido no Programa 9.4. A implementação de `tShellDoc::ConstructScenes()` começa na linha 2. O método constrói uma cena da classe `tGeoScene` (definida a seguir). Se o modelo geométrico da casca existir, um ator é construído e adicionado à cena de modelo geométrico do documento. Agora que o documento já sabe como inicializar suas cenas, vejamos como usar a base de dados da aplicação.

9.5.1 Criando um Novo Documento

Para criarmos um novo documento, enviamos a mensagem `CmFileNew()` ao objeto de aplicação. O método virtual `tApplication::CmFileNew()` executa outro método virtual da classe `tApplication` chamado `CreateDoc()`, passando como parâmetro para o método a constante `dfNewDoc`. `CreateDoc()` verifica se existe um documento aberto e fecha esse documento. Em seguida, executa o método virtual `ConstructDoc()` (apresentamos a versão para a classe `tShellApp` no Programa 9.3), responsável pela construção de um novo documento da aplicação. Finalmente, o novo documento é inicializado pelo método virtual `InitDoc()`, o qual executa `ConstructScenes()`. Podemos especializar todo esse comportamento, mas usualmente deixaremos isso por conta dos métodos de `tApplication`.

9.5.2 Salvando um Documento

Para armazenarmos um documento em disco, enviamos a mensagem `CmFileSave()` ao objeto de aplicação. O método virtual `tApplication::CmFileSave()` verifica se o documento foi alterado desde a última execução do método, enviando a mensagem `IsDirty()` para o documento. Se a execução do método virtual `IsDirty()` retornar `true`, `CmFileSave()` envia a mensagem `Commit()` ao documento. O método virtual `tDocument::Commit()` cria um *stream* persistente de escrita e "atravessa" a coleção de modelos e de cenas do documento da aplicação, enviando a mensagem `Write()` para cada objeto do documento. `Write()` toma como parâmetro o *stream* persistente. O método `tShellDoc::Commit()` executa `tDocument::Commit()`. Sobrecarregamos o método somente para escrita do ponteiro `Shell`, declarado no Programa 9.4.

As bases de dados de uma aplicação OSW não são muito sofisticadas. Não tivemos pretensão de construir nada parecido com um banco de dados, muito menos com um banco de dados orientado a objetos. Para nossos propósitos, os *streams* persistentes da biblioteca de classes do Borland C++ (classes `ipstream` e `opstream`) se mostraram

bastante adequados. Os objetos persistentes de OSW são derivados da classe `TStreamableBase`, definida no arquivo de cabeçalho `objstrm.h`. Para maiores detalhes, consulte o manual da biblioteca de classes do Borland C++ ou a ajuda *on-line* do *software*.

9.5.3 Abrindo um Documento

Para recuperarmos os dados de um documento armazenado em disco, enviamos a mensagem `CmFileOpen()` ao objeto de aplicação. `tApplication::CmFileOpen()` executa o método virtual `CreateDoc()`, passando como parâmetro a constante `dfOpenDoc`. `tApplication::CreateDoc()` abre uma caixa de diálogo solicitando ao usuário o nome do documento (a extensão padrão do arquivo é definida pelo atributo `DocPathData`, linha 8 do Programa 9.2). Após a seleção do nome do arquivo, `CreateDoc()` solicita a construção de um novo documento (método `ConstructDoc()`), e inicializa o novo documento com o nome selecionado, como fizemos na linha 7 do Programa 9.2. `InitDoc()`, por sua vez, envia as mensagens `Open()` e `ConstructScenes()` ao novo documento.

O método `Open()` é declarado como virtual em `tDocument` e, portanto, pode ser sobrecarregado em uma classe derivada. `tDocument::Open()` associa um *stream* persistente de leitura ao arquivo do documento, responsável pela construção dos objetos armazenados no arquivo e pelo envio da mensagem `Read()` a cada objeto construído. O método `tShellDoc::Open()` executa `tDocument::Open()`. Sobrecarregamos o método somente para leitura do ponteiro `Shell`, declarado no Programa 9.4.

9.5.4 Fechando um Documento

Para destruirmos um documento, enviamos a mensagem `CmFileClose()` ao objeto da aplicação. O método virtual `tApplication::CmFileClose()` executa o método virtual `CanClose()`, declarado em `tApplication`. `tApplication::CanClose()` verifica se o objeto de documento da aplicação existe e, se existir, envia a mensagem `CanClose()` ao documento.

O método `CanClose()` é declarado como virtual em `tDocument` e, portanto, pode ser sobrecarregado em uma classe derivada. `tDocument::CanClose()` executa o método virtual `IsDirty()`. Se o método retornar **`true`**, `CanClose()` abre uma caixa de diálogo solicitando ao usuário se o documento deve ser salvo ou se a operação deve ser cancelada. Se o usuário cancelar a operação, `CanClose()` retorna **`false`**. Caso contrário, `CanClose()` envia a mensagem `CmFileSave()` a seu objeto de aplicação e retorna **`true`**. Não sobrecarregaremos quaisquer desses métodos virtuais.

9.5.5 Adicionando e Removendo Modelos do Documento

A adição de um modelo a um documento é executada pelo método `AddModel()`, declarado na classe `tDocument`. O método toma como parâmetro um ponteiro para o modelo.

```
tDocument* doc;
tModel* pmodel;
...
doc->AddModel(pmodel);
```

O modelo adicionado ao documento pode ser utilizado por diversos atores e ser exibido em várias cenas do documento, ou não ser exibido em cena alguma. Note: o modelo é propriedade do documento, mas pode ser *compartilhado* por outros objetos do sistema, tais como atores.

Um modelo é removido do documento pelo método `DeleteModel()`, declarado na classe `tDocument`. Como no caso da adição, o método toma como parâmetro um ponteiro para o modelo.

```
tDocument* doc;
tModel* pmodel;
...
doc->DeleteModel(pmodel);
```

C++ é uma linguagem que não possui recursos de "coleta de lixo" automática (*garbage collection*). O método `DeleteModel()` deve invocar o destrutor de `*pmodel`. Caso contrário, o modelo continuará a existir no *heap* do sistema, mas será inacessível pela aplicação. Um desperdício de memória. No entanto, permitimos que outros objetos (atores, por exemplo), compartilhem o modelo. Se o modelo estiver sendo utilizado por outro objeto e for destruído, teremos ponteiros apontando para o modelo que já não existe mais (*dangling pointers*).

A classe `tModel` é derivada da classe `tObjectBody`. Um corpo de objeto possui um *contador de referência* que é incrementado de um sempre que o objeto for compartilhado por outro objeto. Quando um objeto não usa mais um corpo de objeto, decrementa seu contador de referência de um. Somente quando o contador de referência do corpo de objeto for igual a zero é que seu destrutor será invocado. A função `DeleteModel()` remove o modelo da lista de modelos do documento e envia a mensagem `Delete()` ao modelo. O método `tObjectBody::Delete()` decrementa o contador de referência do objeto e, se o contador for igual a zero, executa o destrutor do objeto. Muitas outras classes de objetos de OSW são derivadas de `tObjectBody` (veja as classes `tMatrix` e `tVector` no Capítulo 10).

9.5.6 Adicionando e Removendo Cenas ao Documento

A adição de uma cena ao documento é efetuada pela execução do método `AddScene()`, declarado na classe `tDocument`. O método toma como parâmetro um ponteiro para a cena.

```
tDocument* doc;
tScene* pscene;
...
doc->AddScene(pscene);
```

Quando construímos uma cena, não nos preocupamos em adicioná-la a um documento porque o construtor da classe `tScene` realiza essa tarefa. `tScene::tScene(doc)` envia a mensagem `AddScenes()` ao documento `doc` passado como parâmetro do construtor.

A cena é removida do documento pelo método `DeleteScene()`, definido na classe `tDocument`. Como no caso da adição, o método toma como parâmetro um ponteiro para a cena.

```
tDocument* doc;
tScene* pscene;
...
doc->DeleteScene(pscene);
```

9.6 Definindo os Modelos do Documento

Um modelo é um objeto de uma classe derivada da classe `tModel`. A classe `tModel` representa um modelo genérico da base de dados da aplicação. Um modelo de sólidos e um modelo de cascas, por exemplo, são objetos de classes derivadas de `tModel`. Mas, como vimos, no Capítulo 3, os modelos de OSW são bastante distintos entre si. Que tipo de generalização, então, `tModel` representa?

Todos os modelos de OSW são definidos por *coleções* de componentes. Um modelo de decomposição por células é uma coleção de vértices e células, um modelo gráfico é uma coleção de primitivos e um modelo de sólidos é uma coleção de faces, vértices e arestas. Em programação orientada a objetos, um objeto que representa uma coleção de outros objetos é chamado de *container*. Nossos modelos são, portanto, *containers*. (Documentos e cenas também são *containers*.) Embora a *estrutura de dados* dos componentes dos modelos sejam distintas, podemos estabelecer seu *comportamento* comum. Um modelo, independente de sua representação, deve ser capaz de armazenar e recuperar seus componentes da base de dados da aplicação, por exemplo. Um modelo também deve permitir que outros objetos possam ter acesso a seus componentes. Um objeto da classe `tScanner`, por exemplo, necessita conhecer quais são as faces de um modelo para poder gerar uma imagem. Temos alguns problemas aqui que merecem ser discutidos.

Vimos, no Capítulo 8, que uma das principais propriedades da programação orientada a objetos é o *encapsulamento*. Uma definição de classe de objeto deve *esconder* a estrutura de dados do objeto de outros objetos do sistema, permitindo seu acesso somente a partir dos métodos públicos da interface da classe. Um objeto da classe `tScanner`, retomando o exemplo anterior, precisa acessar as faces de um modelo, mas não pode (ou, pelo menos, não poderia) fazê-lo diretamente. Essas informações fazem parte do estado interno do modelo e só podem ser acessadas através de métodos públicos da classe do modelo. Se esses métodos não forem definidos, `tScanner` não poderá (ou não poderia) gerar uma imagem do modelo. Discutimos as vantagens do encapsulamento no Capítulo 8: se modificarmos a implementação do modelo não precisaremos reescrever `tScanner`.

Mas há um outro ponto. `tScanner` só sabe gerar imagens a partir de faces, arestas e vértices. Um modelo de decomposição por células, contudo, não possui faces ou arestas, somente células e vértices. Um modelo de decomposição por células deve *extrair* as faces de suas células (daquelas que possuem dimensão topológica 2 ou 3) e fornecer essas faces a `tScanner`. Modelos de cascas e de sólidos, por outro lado, são definidos diretamente por faces. Nesse caso, nenhuma "extração" de faces seria necessária, a não ser que a estrutura de dados das faces compreendidas por `tScanner` seja diferente da estrutura de dados das faces dos modelos de cascas ou de sólidos.

Para resolvermos o problema, criamos vértices, arestas e faces que são compreendidos por modelos e por *scanners*. Esses objetos são instâncias, respectivamente, das classes `tVertex`, `tEdge` e `tFace`. Se um modelo quiser ter uma imagem gerada por

um `tScanner`, deverá processar mensagens que retornem seus vértices, arestas e faces. Dessa forma, construímos somente uma classe `tScanner` que funciona para *qualquer* modelo de OSW. Novamente, temos um conflito espaço *versus* tempo: `tScanner` funciona com todas as classes de modelos, mas depende da execução de métodos (virtuais) para ter acesso às informações dos modelos.

Há uma outra alternativa: ao invés de `tScanner` enviar mensagens a um modelo solicitando suas faces, o modelo envia mensagens a `tScanner` solicitando o *rendering* de seus componentes. É uma decisão de projeto: ou todo modelo é capaz de fornecer seus vértices, arestas e faces a `tScanner`, ou todo modelo conhece profundamente o funcionamento de `tScanner`. Infelizmente, essa segunda estratégia não funciona para outros tipos de *renderers*. Um objeto da classe `tRayTracer`, por exemplo, necessita três informações de um modelo: o ponto de intersecção do modelo com um raio, a normal no ponto de intersecção e o material no ponto de intersecção. Não faz sentido um modelo enviar mensagens a um `tRayTracer` solicitando alguma coisa. Adotamos a primeira alternativa.

O acesso aos objetos de um *container* é executado por um outro objeto chamado *iterator*. Para uma classe de modelo, definiremos três classes de *iterators*: um *iterator* de vértices, um *iterator* de arestas e um *iterator* de faces. Essas classes são derivadas, respectivamente, das classes bases `tVertexIterator`, `tEdgeIterator` e `tFaceIterator` (veja o Capítulo 10). A classe abstrata `tModel` declara três métodos virtuais que retornam cada um dos três *iterators* do modelo: `GetVertexIterator()`, `GetEdgeIterator()` e `GetFaceIterator()`. As classes `tGraphicModel`, `tShell`, `tSolid` e `tMesh` implementam versões próprias desses métodos.

Para definirmos um modelo em OSW, devemos:

Passo 1 Declarar uma classe derivada da classe `tModel`.

Passo 2 Sobrecarregar, na nova classe, os métodos virtuais `GetVertexIterator()`, `GetEdgeIterator()` e `GetFaceIterator()`. Como discutido anteriormente, esses métodos retornam *iterators* que permitem que outros objetos acessem, respectivamente, os vértices, arestas e faces do modelo.

Passo 3 Adicionar, na nova classe, a macro `DECLARE_STREAMABLE`. Com isso, estamos declarando que o modelo pode ser recuperado de e armazenado em um *stream* persistente.

Passo 4 Adicionar, em um arquivo `.cpp`, a macro `IMPLEMENT_STREAMABLEx` e sobrecarregar os métodos virtuais `Read()` e `Write()`, responsáveis, respectivamente, pela recuperação e armazenamento do estado interno do modelo em um *stream* persistente. Veja o guia da biblioteca de classes do Borland C++ para detalhes de utilização de *streams* persistentes.

Podemos, ainda:

Passo 5 Sobrecarregar, na nova classe, o método virtual `Transform()`. As transformações geométricas são transformações básicas em modelagem geométrica. Nesse caso, ao invés de construirmos um filtro de transformação geométrica, preferimos implementar o processo como um método de classe de um modelo (na Seção 9.10 discutimos alguns aspectos de implementação de filtros). O método toma como parâmetro uma referência para um objeto da

classe `t3DTransfMatrix`, o qual representa uma matriz de transformação geométrica. `tModel::Transform()` executa `GetVertexIterator()` para obter um *iterator* de vértices e, então, transforma as coordenadas de cada vértice retornado pelo *iterator*. Podemos sobrecarregar o método para efetuar a transformação do modelo mais eficientemente.

Passo 6 Sobrecarregar, na nova classe, o método virtual `Intersect()`. O método toma como parâmetros uma referência para um objeto da classe `tRay` e uma referência para um objeto da classe `tIntersectInfo` (veja o Capítulo 10) e retorna um **`bool`**. O método é utilizado por um `tRayTracer` para obter informações de intersecção de um raio com o modelo. `tModel::Intersect()` retorna **`false`**. Se quisermos obter imagens "raio traçadas" do modelo, temos que sobrecarregar o método.

9.7 Definindo as Cenas do Documento

Um objeto da classe `tScene` é uma cena genérica de um documento de uma aplicação. Podemos adicionar e remover atores, luzes e câmeras de uma cena. Uma cena contém, ainda, uma coleção de *vistas*. Uma vista de uma cena é responsável pela exibição de uma imagem da cena e pela execução dos comandos de manipulação dos objetos da cena. Definiremos as vistas de uma cena na próxima seção.

Para definir uma cena, devemos:

Passo 1 Declarar uma classe derivada da classe `tScene`.

Passo 2 Sobrecarregar, na nova classe, o método virtual `ConstructView()`, responsável pela construção da vista específica da cena. *Podemos* sobrecarregar outros métodos virtuais de `tScene`, mas *devemos* sobrecarregar `ConstructView()`.

Passo 3 Adicionar, na nova classe, a macro `DECLARE_STREAMABLE`, ou, se a nova classe não possuir atributos adicionais, a macro `DECLARE_STREAMABLE_FROM_BASE`. Com isso, estamos declarando, da mesma forma que fizemos para um modelo, que a cena pode ser recuperada de e armazenada em um *stream* persistente.

Passo 4 Adicionar, em um arquivo `.cpp`, a macro `IMPLEMENT_STREAMABLEx` (nesse caso, devemos sobrecarregar também os métodos virtuais de armazenamento e recuperação da cena, `Read()` e `Write()`), ou, se a classe não possuir atributos adicionais, a macro `DECLARE_STREAMABLE_FROM_BASE`.

Nossa aplicação de análise e visualização de cascas manipula diversos tipos de modelos. Inicialmente, temos o modelo geométrico da casca, do qual é derivado o modelo mecânico para análise. Após a análise, extraímos campos escalares e vetoriais do modelo mecânico para visualização e geramos modelos gráficos. Para visualizarmos cada um desses modelos, criamos um ator para o modelo. Mas será conveniente definirmos cenas distintas para cada caso. Poderia ser confuso incluir na mesma cena o modelo geométrico e mecânico da casca, por exemplo. Nossa aplicação de análise e visualização de cascas manipula três classes de cena: `tGeoScene`, `tMecScene` e `tPosScene`. Apresentaremos aqui somente a classe de cena do modelo geométrico da casca, declarada no arquivo `shellscene.h` mostrado no Programa 9.6.

```
1   #include "scene.h"
2   class tGeoScene:public tScene
3   {
4     public:
5       // Constructor
6       tGeoScene(tShellDoc& doc):
7         tScene(doc)
8       {}
9       tShellDoc& GetDoc()
10      {
11        return *(static_cast<tShellDoc*>(Doc));
12      }
13
14    private:
15      void   InitDefaultViews();
16      tView* ConstructView(tScene&, tCamera*);
17      DECLARE_STREAMABLE_FROM_BASE(IMPEXPMACRO,tGeoScene,1);
18
19  }; // tGeoScene
```

Programa 9.6: Definindo a cena do modelo geométrico de uma casca.

Na linha 1 incluímos o arquivo de cabeçalho `scene.h`, o qual contém a definição da classe base `tScene`. Na linha 2 declaramos a classe `tGeoScene`, derivada de `tScene`. O construtor da classe é declarado na linha 6. O construtor de `tGeoScene` toma como parâmetro uma referência `doc` para o documento ao qual a cena pertence. Usamos `doc` para inicializar a classe base `tScene`. Na linha 9 declaramos o método **`inline`** `GetDoc()`, o qual retorna uma referência para o documento da cena. (`Doc` é declarado em `tScene` como sendo um ponteiro de um objeto da classe `tDocument`. Por isso efetuamos a conversão de tipos.) Nas linhas 15 e 16, respectivamente, declaramos os métodos `InitDefaultViews()` e `ConstructView()`, herdados de `tScene`. Na linha 17 adicionamos a macro `DECLARE_STREAMABLE_FROM_BASE`. A implementação da classe é mostrada no Programa 9.7.

Na linha 1 do Programa 9.7, incluímos o arquivo de cabeçalho `shellscene.h`, definido no Programa 9.6. Adicionamos a macro `IMPLEMENT_STREAMABLE_FROM_BASE` na linha 2. A implementação do método virtual de inicialização das vistas da cena começa na linha 4. `tGeoScene::InitDefaultViews()` constrói quatro vistas da classe `tGeomView` (definida na próxima seção). O resultado da execução do método é ilustrado na Figura 9.4.

9.7.1 Adicionando e Removendo Atores de uma Cena

A adição de um ator a uma cena é efetuada pela execução do método `AddActor()`, declarado na classe `tScene`. O método toma como parâmetro um ponteiro para o ator.

```
tScene* scene;
tActor* actor;
...
scene->AddActor(actor);
```

```
1   #include "shellscene.h"
2   IMPLEMENT_STREAMABLE_FROM_BASE(tGeoScene, tScene);
3
4   void
5   tGeoScene::InitDefaultViews()
6   {
7     CreateView(vfFrontView);
8     CreateView(vfTopView);
9     CreateView(vfRightView);
10    CreateView(vfIsometricView);
11  }
12
13  tView*
14  tGeoScene::ConstructView(tScene& scene, tCamera* camera)()
15  {
16    return new tGeoView((tGeoScene&)scene, camera);
17  }
```

Programa 9.7: Inicializando as cenas do documento.

Para construirmos um ator para um modelo de sólidos, por exemplo, podemos escrever:

```
tActor* actor = new tActor(new tSolid);
```

O construtor da classe `tActor` toma como parâmetro um ponteiro para um objeto da classe `tModel` (`tSolid` é um `tModel`) e constrói automaticamente um mapeador para o modelo (veja a Seção 9.11).

Um ator é removido da cena pelo método `DeleteActor()`, definido na classe `tScene`. Como no caso da adição, o método toma como parâmetro um ponteiro para o ator.

```
tScene* scene;
tActor* actor;
...
scene->DeleteActor(actor);
```

Podemos remover um ator de uma cena enviando a mensagem `CmDeleteActor()` a algum objeto de vista de cena. O método virtual `tView::CmDeleteActor()` é um comando padrão de uma vista de uma cena que permite ao usuário selecionar o(s) ator(es) que será(ão) eliminado(s). O método executa `DeleteActor()` (na próxima seção veremos como executar comandos de uma vista de uma cena.)

9.7.2 Adicionando e Removendo Luzes de uma Cena

A adição de uma luz a uma cena é efetuada pela execução do método `AddLight()`, declarado na classe `tScene`. O método toma como parâmetro um ponteiro para a luz.

```
tScene* scene;
tLight* light;
...
scene->AddLight(light);
```

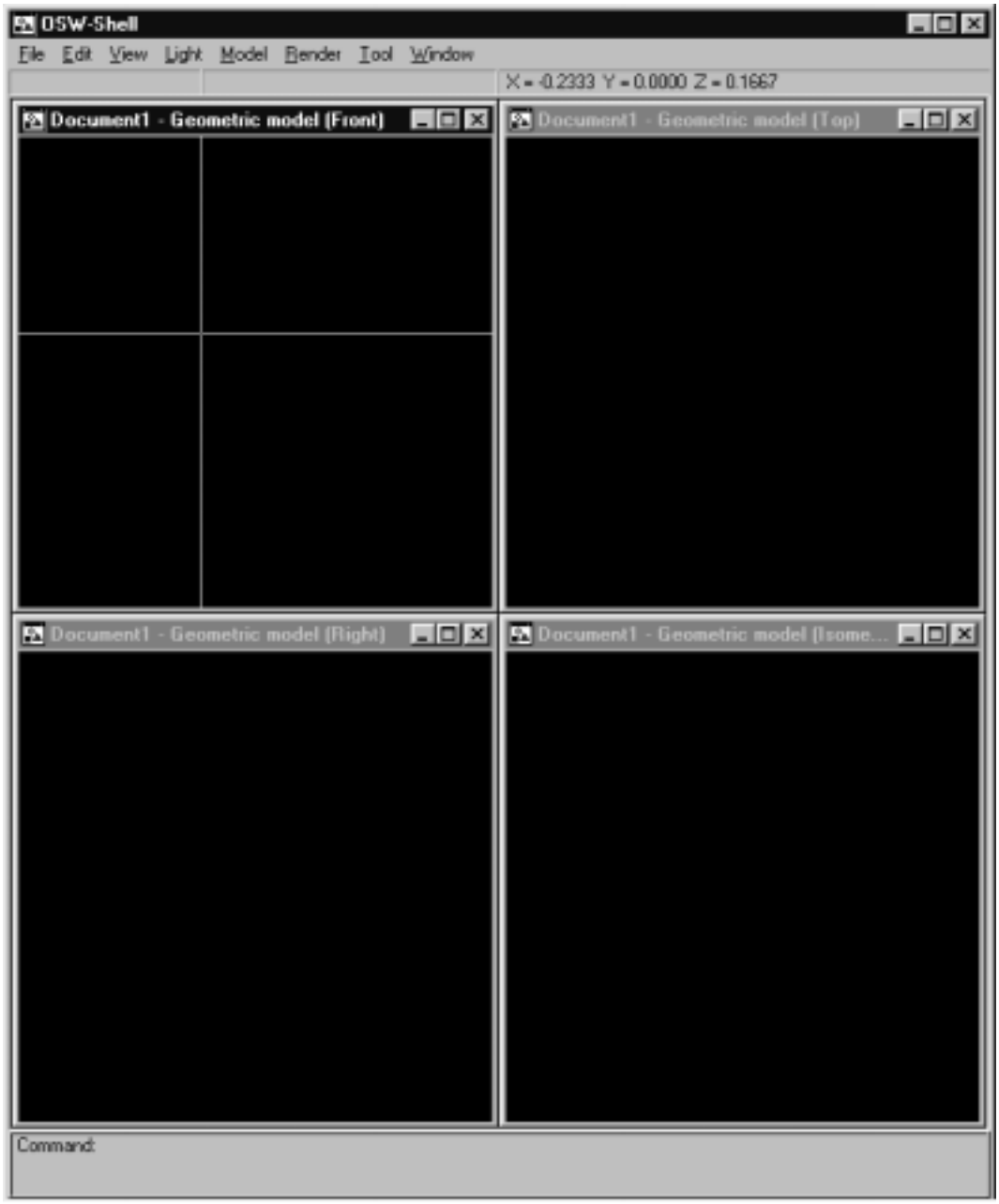


Figura 9.4: Vistas da cena do modelo geométrico de uma casca.

Podemos criar uma luz de uma cena enviando a mensagem `CmCreateLight()` a algum objeto de vista da cena. O método virtual `tView::CmCreateLight()` é um comando padrão de uma vista de uma cena que permite ao usuário selecionar o tipo da fonte de luz desejada (puntual ou direcional), posicionar a fonte de luz na cena e escolher a cor da luz. `CmCreateLight()` executa o método `AddLight()` para adicionar a luz na cena.

Uma luz é removida da cena pelo método `DeleteLight()`, definido na classe `tScene`. Como no caso da adição, o método toma como parâmetro um ponteiro para a luz.

```
tScene* scene;
tLight* light;
...
scene->DeleteLight(light);
```

Podemos remover uma luz de uma cena enviando a mensagem `CmDeleteLight()` a algum objeto de vista de cena. O método virtual `tView::CmDeleteLight()` é um comando padrão de uma vista de uma cena que permite ao usuário selecionar a(s) luz(es) que será(ão) eliminada(s). O método executa `DeleteLight()`.

9.7.3 Adicionando e Removendo Câmeras de uma Cena

A adição de uma câmera a uma cena e a remoção de uma câmera de uma cena é efetuada da mesma forma que fizemos para atores e luzes. Para adicionarmos uma câmera, usamos o método `AddCamera()`; para removermos uma câmera, usamos o método `DeleteCamera()`. Ambos os métodos são definidos na classe `tScene` e tomam como parâmetro um ponteiro para um objeto da classe `tCamera`. Analogamente, temos definidos na classe `tView` os métodos virtuais `CmCreateCamera()` e `CmDeleteCamera()`, os quais podem ser utilizados para criação e remoção de uma câmera de uma cena.

9.7.4 Adicionado e Removendo Vistas de uma Cena

Para adicionarmos uma vista a uma cena, enviamos a mensagem `CreateView()` ao objeto de cena. O método virtual `tScene::CreateView()` toma com parâmetro um **`long`** que define o tipo de vista que será criado. Os valores do parâmetro podem ser as constantes `vfFrontView`, `vfTopView`, `vfRightView`, `vfBackView`, `vfLeftView`, `vfBottomView` e `vfUserView`, definidas na classe `tScene`. `CreateView()` constrói um objeto da classe `tCamera` de acordo com o valor do parâmetro e executa o método virtual `ConstructView()`, passando como parâmetros uma referência para a cena e um ponteiro para o objeto da classe `tCamera` (o método `tGeoScene::ConstructView()` foi implementado no Programa 9.7). Por exemplo,

```
tScene* scene;
...
scene->CreateView(tScene::vfTopView);
```

cria uma vista frontal da cena. Se a constante `vfUserView` for usada, o método `CreateView()` abre uma caixa de diálogo permitindo ao usuário ajustar os parâmetros da câmera (posição, ponto focal, etc.).

Removemos uma vista da cena simplesmente fechando a janela de vista. Veremos um pouco mais sobre vistas a seguir.

9.8 Definindo as Vistas das Cenas do Documento

Um objeto da classe `tView` é um objeto de interface homem-documento. Na Figura 9.1 temos alguns exemplos de vistas de cenas de um documento, as quais exibem o modelo geométrico de uma casca, a malha de elemetos finitos, um mapa de cores e a estrutura deformada da casca. Cada uma dessas vistas está associada a uma janela-filha da janela principal da aplicação, chamada *janela de vista*. Uma aplicação pode possuir várias

janelas de vistas de cenas na área cliente da janela principal. No entanto, somente uma das janelas de vistas estará *ativa*, em determinado momento.

Uma vista exibe uma imagem de uma cena de um documento. Portanto, uma vista deve conhecer a cena a partir da qual a imagem será gerada. A imagem exibida pela vista deve ser tomada por alguma câmera, pertencente ou não à coleção de câmeras da cena. Uma câmera é um objeto da classe `tCamera`, a qual define métodos para ajuste dos parâmetros definidos no Capítulo 7. A imagem de uma cena exibida em uma vista é gerada por um objeto derivado da classe abstrata `tRenderer`.

Observe a janela de vista da Figura 9.5. A vista exibe a cena do modelo mecânico de uma casca. A imagem foi capturada durante a execução do comando `fix`. O comando `fix` impõe as condições de contorno essenciais a um conjunto de vértices do modelo mecânico. A execução do comando inicia com a especificação de quais graus de liberdade serão restringidos (por exemplo, os componentes de deslocamento u, v e w). Em seguida, os vértices do modelo aos quais as restrições serão impostas devem ser selecionados. A seleção é efetuada na vista da cena do modelo, através do retângulo *elástico* mostrado na figura. Todos os vértices contidos no interior do retângulo serão selecionados e terão os deslocamentos u, v e z restringidos. Portanto, uma vista não exibe somente a imagem de uma cena, mas possibilita que, através da imagem, o usuário manipule os objetos da cena. Essa manipulação é definida nos *comandos* da vista.

Classes derivadas da classe base `tView` podem (e devem) definir seu próprio conjunto de comandos. Para exemplificar, consideremos uma vista de uma cena do modelo geométrico de uma casca e uma vista de uma cena do modelo mecânico de uma casca. Ambas as vistas possuem estrutura e funcionalidade comuns. Por exemplo, podemos solicitar, a ambas, a visualização fio-de-arame do modelo, a criação de uma luz ou a alteração dos parâmetros de uma câmera. Mas a funcionalidade *específica* de cada uma das vistas é distinta. Para a primeira, podemos solicitar, por exemplo, a geração da malha do modelo mecânico; para a segunda, a análise numérica e a geração de isolinhas.

Para definirmos uma vista de uma cena, devemos:

Passo 1 Declarar uma classe derivada da classe `tView`.

Passo 2 Declarar e implementar os comandos da nova classe, se quisermos que a vista responda a comandos próprios (se não quisermos, não há necessidade de definirmos uma nova classe de vista).

9.8.1 Definindo os Comandos de uma Vista

Uma vista possui uma *tabela de comandos* que define quais são os comandos que podem ser executados pela vista. Esses comandos podem criar, alterar ou eliminar atores, luzes ou câmeras da cena da vista (vimos alguns exemplos na seção anterior), ou criar, alterar e eliminar novas vistas da cena. Um comando é definido por um nome, um identificador numérico e um método da classe de vista que implementa o comando.

Para definir um comando de uma vista, é preciso:

Passo 1 Declarar o método correspondente na definição de classe da vista.

Passo 2 Adicionar, na definição de classe da vista, a macro `DECLARE_COMMAND_TABLE`, a qual declara a tabela de comandos da classe. (A macro possui assinatura

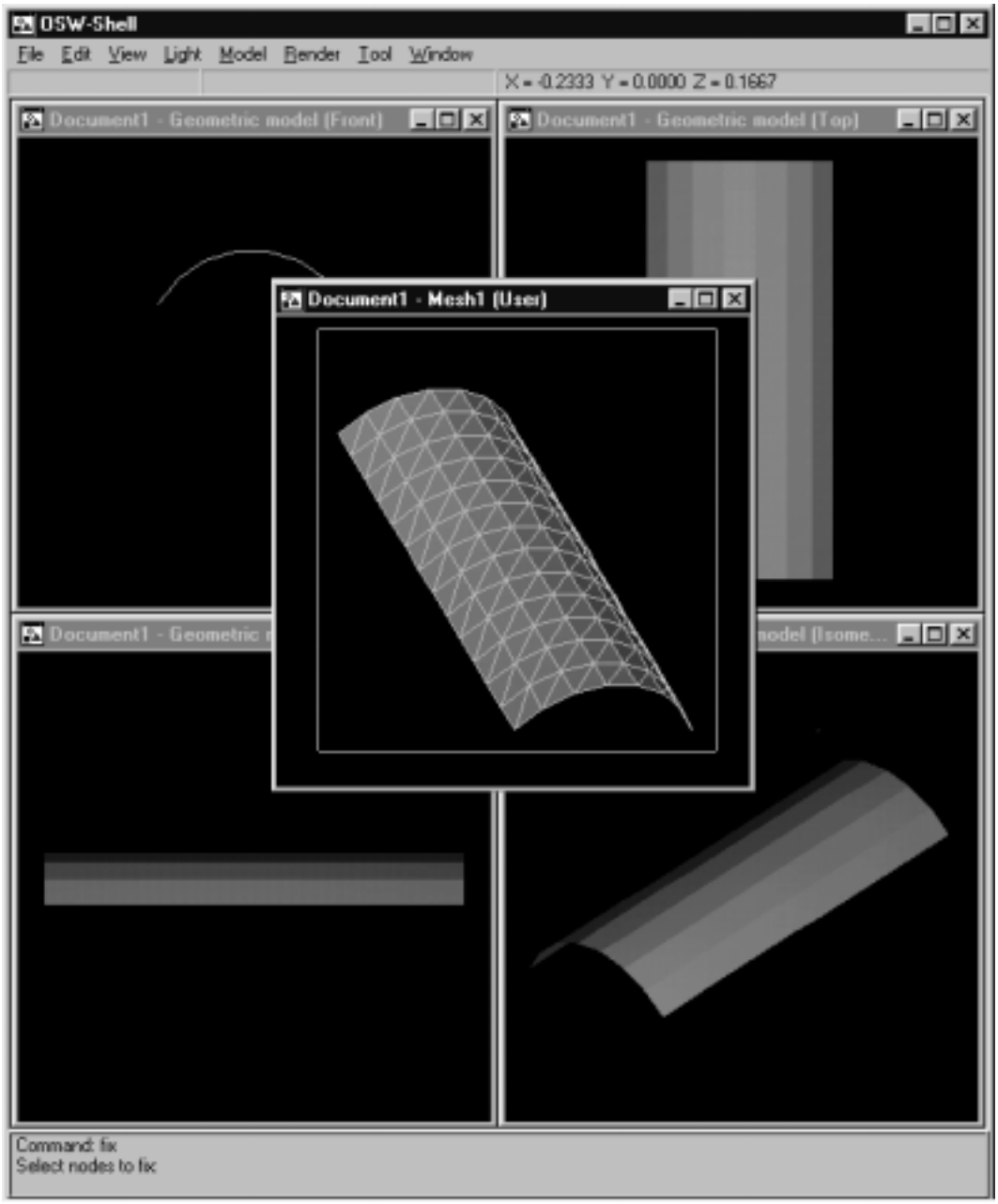


Figura 9.5: Execução de um comando de uma vista de cena.

similar a macro da OWL chamada `DECLARE_RESPONSE_TABLE`, mas as funcionalidades de ambas são bastante distintas.)

Passo 3 Em um arquivo `.cpp`, definir a tabela de comandos da classe.

Passo 4 Implementar o método correspondente ao comando.

Passo 5 Opcionalmente, incluir no arquivo de recursos um ítem de menu correspondente ao comando criado.

O arquivo `shellview.h`, mostrado no Programa 9.8, contém a definição (parcial) da classe `tGeoView`. Utilizaremos a classe para exemplificar a definição dos comandos de uma vista.

```
1   #include "shellscene.h"
2   #include "shellapp.rh"
3   class tGeoView:public tView
4   {
5     public:
6       // Constructor
7       tGeoView(tGeoScene& geoScene, tCamera* camera):
8         tView(geoScene, camera)
9       {
10        SetViewMenu(SHELL_MENU);
11      }
12      tGeoScene& GetScene()
13      {
14        return *(static_cast<tGeoScene*>(Scene));
15      }
16
17    private:
18      const char* GetViewName();
19      void        CmGenerateMesh();
20      void        CmRotate();
21      void        CmReadShell();
22      ...
23      DECLARE_COMMAND_TABLE(tGeoView);
24
25  }; // tGeoView
```

Programa 9.8: Definição da vista de uma cena da aplicação de cascas.

Na linha 1 do programa incluímos o arquivo de cabeçalho `shellscene.h`, definido no Programa 9.6 (`shellscene.h` inclui `scene.h`, o qual, por sua vez, inclui `view.h`). Na linha 2 incluímos o arquivo de cabeçalho de recursos `shellapp.rh` (a constante `SHELL_MENU`, linha 10, é definida nesse arquivo). Na linha 3 declaramos a classe `tGeoView`, derivada de `tView`. Na linha 7 declaramos o construtor da classe. Os parâmetros do construtor são uma referência `geoScene` para a cena proprietária da vista e um ponteiro `camera` para um objeto da classe `tCamera` (construído pelo método `tScene::CreateView()`). Usamos `geoScene` e `camera` para inicializar `tView`. Na linha 12 declaramos o método `GetScene()`, o qual retorna uma referência para a cena da vista. Nas linhas 18, 19, 20 e 21, respectivamente, declaramos os métodos privados `GetViewName()` (herdado de `tView`), `CmGenerateMesh()`, `CmRotate()` e `CmReadShell()`. Na linha 23, declaramos que a classe `tGeoView` possui uma tabela de comandos. A declaração é feita através da macro `DECLARE_COMMAND_TABLE`. O parâmetro da macro é o nome da classe. Para definirmos a tabela de comandos de uma vista, devemos:

Passo 1 Adicionar, em um arquivo `.cpp`, a macro `DEFINE_COMMAND_TABLEx`.

Passo 2 Definir os comandos da tabela, usando as macros `COMMAND` e `COMMAND_AND_ID`.

Passo 3 Finalizar a definição da tabela com a macro `END_COMMAND_TABLE`.

A implementação da classe é feita no arquivo `shellview.cpp`, mostrado parcialmente no Programa 9.9.

```
1   #include "shellview.h"
2   DEFINE_COMMAND_TABLE1(tGeoView, tView)
3     ...
4     COMMAND("generate_mesh", 100, CmGenerateMesh),
5     COMMAND("rotate", 101, CmRotate),
6     COMMAND("read_shell", 102, CmReadShell),
7   END_COMMAND_TABLE;
8
9   void
10  tGeoView::CmGenerateMesh()
11  {
12    tShell* shell = GetScene().GetDoc().GetShell();
13
14    if (shell)
15    {
16      tFEShellMeshGenerator mg;
17
18      mg.SetInput(shell);
19      mg.Execute();
20      tMesh* mesh = mg.GetOutput();
21      if (mesh)
22      {
23        tMecScene* mecScene = new tMecScene(Scene->Doc);
24
25        mecScene->InitDefaultViews();
26        mecScene->AddActor(new tActor(mesh));
27      }
28    }
29  }
```

Programa 9.9: Implementação de um comando de uma vista.

Na linha 1 incluímos o arquivo de cabeçalho `shellview.h`, o qual contém a definição da classe `tGeoView`. Na linha 2 iniciamos a definição da tabela de comandos de `tGeoView`, com a macro `DEFINE_COMMAND_TABLE1`. Os parâmetros da macro são o nome da classe a qual a tabela pertence e o nome da classe base, `tGeoView` e `tView`, respectivamente. Nas linhas 4, 5 e 6 definimos, respectivamente, os comandos `generate_mesh`, `rotate` e `read_shell` na tabela de comandos da vista. A macro `COMMAND` toma três parâmetros: o nome do comando, o identificador numérico do comando e o nome do método de `tGeoView` que implementa o comando. (A vírgula após a macro `COMMAND` é obrigatória.) Na linha 7, a macro `END_COMMAND_TABLE` é utilizada para finalizar a definição da tabela de comandos da classe. Os métodos de implementação dos comandos não tomam quaisquer argumentos.

A implementação de `CmGenerateMesh()` começa na linha 9 do Programa 9.9. Primeiramente, verificamos se o modelo geométrico da casca existe no documento. Se o modelo existir, construímos um filtro da classe `tFEShellMeshGenerator` (um fil-

tro é um processo que toma como entrada um ou mais modelos e produz como saída um ou mais modelos, como vimos no Capítulo 7). O filtro da classe `tFEShellMeshGenerator`, derivada da classe `t2DMeshGenerator`, toma como entrada um modelo geométrico de cascas ou de sólidos e produz como saída um modelo de decomposição por células triangulares ou quadrilaterais (na Seção 9.10 abordaremos a utilização de filtros). A entrada do filtro é definida pela mensagem `SetInput()`, linha 18. Na linha 19, o filtro é executado. O modelo de saída do filtro, uma malha de elementos da classe `tMesh`, é obtido pela mensagem `GetOutput()`, linha 20. Se a execução do filtro foi bem sucedida, construímos na linha 23 uma nova cena de modelos mecânicos, inicializada na linha 25. Na linha 26, criamos um novo ator e passamos como parâmetro do construtor o ponteiro da malha gerada pelo filtro. Finalmente, adicionamos o ator à coleção de atores da cena de modelos mecânicos.

O comando `generate_mesh` pode ser digitado na janela de comandos da aplicação. Quando isso é feito, ocorre o seguinte:

- Após a entrada do nome do comando na janela de comandos, a aplicação envia uma mensagem para a janela de vista ativa, se houver, informando que o comando `generate_mesh` foi solicitado.

- Em resposta à mensagem, a vista procura em sua tabela de comandos o comando cujo nome é `generate_mesh`. Se o comando for encontrado, a vista executa o método correspondente ao comando. Se o comando não for encontrado, a busca continuará na tabela de comandos da sua classe base (segundo parâmetro da macro `DEFINE_RESPONSE_TABLE`). Se nenhuma vista puder responder ao comando, uma mensagem de erro é exibida na janela de comandos.

Podemos, ainda, criar ítens de menu para os comandos. O Programa 9.10 mostra um pequeno trecho do arquivo `shellapp.rc`. O arquivo faz parte do projeto da aplicação e contém as definições dos recursos utilizados em OSW-Shell.

```
1   #include "shellapp.rh"
2   ...
3   POPUP SHELL_MENU
4   {
5      ...
6      MENUITEM "Read &shell...", 102
7      MENUITEM "&Rotate", 101
8      MENUITEM SEPERATOR
9      MENUITEM "&Generate mesh", 100
10  }
11  ...
```

Programa 9.10: Criando ítens de menu para os comandos.

Na linha 3 criamos um menu *popup* identificado por `SHELL_MENU` (definido em `shellapp.rh`, incluído na linha 1). Nas linha 6, 7 e 9 definimos os ítens de menu para os comandos `read_shell`, `rotate` e `generate_mesh`, respectivamente. Um ítem de menu é definido por uma cadeia de caracteres (o texto do ítem de menu) e por um identificador numérico. Quando o ítem for selecionado, o Windows envia uma mensagem do tipo `WM_COMMAND` para a janela de vista ativa, sendo o identificador numérico passado

como parâmetro da mensagem. Note que os identificadores numéricos são os mesmos utilizados nas definições dos comandos, Programa 9.9. Em reposta à mensagem, a janela de vista efetua uma busca em sua tabela de comandos como anteriormente, só que desta vez utilizando o identificador numérico do comando, ao invés do nome. No construtor de `tGeoView`, linha 10 do Programa 9.8, usamos o método `SetViewMenu()` para ajustar o menu das vistas de cena do modelo geométrico da aplicação de casca.

9.8.2 Executando os Comandos de um Vista

A janela de vista ativa contém o *cursor*, um elemento de interface que indica qual a posição corrente na vista. Há vários tipos de cursores pré-definidos em OSW, exibidos em função do tipo de operação sendo executada pela vista. Quando o ponteiro do *mouse* passa sobre a janela de vista ativa, o cursor é exibido. A vista possui um *plano de trabalho* que define a posição espacial do cursor. Usamos o plano de trabalho para determinar a altura do cursor (lembremos que a transformação de projeção não é inversível, ou seja, não podemos determinar as coordenadas espaciais de um ponto projetado em um plano). Vejamos algumas utilidades do cursor.

Funções de Entrada de uma Vista

O trecho de código do Programa 9.11 mostra a implementação do comando `rotate` de rotação do modelo geométrico de uma casca. Vamos comentar mais detalhadamente o programa.

```
1   void
2   tGeoView::CmRotate()
3   {
4     tShell* shell = GetScene().GetDoc().GetShell();
5
6     if (shell)
7     {
8       t3DVector bp;
9       double angle;
10
11      if (GetPoint(bp, "Enter the rotation base point:"))
12        if (GetAngle(angle, "Enter the rotation angle:"))
13        {
14          t3DTransfMatrix r;
15
16          r.SetRotation(bp, Camera->GetVPN(), angle);
17          shell->Transform(r);
18          Scene->Regenerate();
19        }
20    }
21  }
```

Programa 9.11: Outro exemplo de comando de uma vista.

Linha 6 Primeiro, verificamos se o modelo geométrico existe. Como estamos admitindo somente um modelo na cena, a implementação do comando é mais simples. Se tivéssemos vários objetos na cena, precisaríamos *selecionar* quais objetos gostaríamos de aplicar a transformação. Essa operação de entrada é chamada, em computação gráfica, de *pick*.

Linha 11 Se o modelo geométrico existir, perguntamos qual é o ponto base da rotação. A pergunta é feita através do método de `tView` chamado `GetPoint()`. Os parâmetros do método são uma referência para um objeto da classe `t3DVector` e um ponteiro de caracteres. O primeiro parâmetro conterá o ponto desejado, se `GetPoint()` retornar **`true`**. O segundo parâmetro define a mensagem exibida pelo método na janela de comandos da aplicação. As coordenadas do ponto podem ser definidas de duas maneiras: digitando-se diretamente três expressões separadas por vírgulas na janela de comandos, logo após a mensagem exibida pelo método (por exemplo, `1,1,1`), ou movendo-se o cursor para a posição desejada (as coordernadas do cursor são mostradas na barra de *status* da aplicação). O método `GetPoint()` implementa o dispositivo lógico de entrada chamado *locator*. A entrada pode ser cancelada, digitando-se CTRL+C ou pressionado-se o botão direito do mouse. Nesse caso, a mensagem apropriada é exibida na janela de comandos e a função retorna **`false`**. Esse é o único cuidado que deve ser tomado pelo programador do comando: testar o valor de retorno de uma função de entrada da vista.

Linha 12 Perguntamos, agora, qual é o valor do ângulo de rotação (o eixo de rotação é o VPN da câmera da vista). A pergunta é feita através do método `GetAngle()`. Os parâmetros do método são uma referência para um **`double`** e um ponteiro de caracteres. O primeiro parâmetro conterá o valor do ângulo em radianos, se a função retornar **`true`**, e o segundo define a mensagem exibida na janela de comandos. O valor do ângulo pode ser dado diretamente por uma expressão digitada na janela de comandos ou pelo cursor (o ângulo é definido como sendo o ângulo formado pelo vetor ponto base de rotação-posição corrente do cursor e o eixo **`u`** do VRC da vista). O método `GetAngle()`, juntamente com o método `GetDistance()`, implementam o dispositivo lógico de entrada chamado *valuator*. O cursor toma a forma de uma linha elástica quando um desses dois métodos estiver sendo executado.

Linha 14 Construímos um objeto da classe `t3DTransfMatrix`. O objeto representa uma matriz de transformação geométrica, tal como definida no Capítulo 3. Enviamos a mensagem `SetRotation()` para a matriz. Em resposta, a matriz ajusta seus elementos para efetuar a transformação de rotação de `angle` radianos em torno de `Camera->GetVPN()`, com centro no ponto `bp`.

Linha 17 Enviamos uma mensagem `Transform()` para o modelo geométrico da casca. O parâmetro da mensagem é uma referência para a matriz de transformação definida na linha 14. Em resposta, o modelo transforma todos os seus vértices.

Linha 18 Finalmente, enviamos a mensagem `Regenerate()` para a cena da vista. O método virtual `Regenerate()` da classe `tScene` envia uma mensagem

`Regenerate()` para todas as vistas da cena (inclusive para a vista ativa). O método virtual `Regenerate()` da classe `tView` envia a mensagem `Run()` para o *renderer* da vista. Em resposta, o *renderer* gera uma nova imagem da cena, exibida na janela de vista. No final da execução do comando, todas as janelas de vistas da cena mostrarão o modelo rotacionado.

9.9 Usando Fontes

No Capítulo 7 vimos que fontes são processos geradores de dados. Vamos exemplificar a utilização de um fonte com a implementação do comando `read_shell`, definido no Programa 9.9. A implementação é mostrada no Programa 9.12.

```
1   void
2   tGeoView::CmReadShell()
3   {
4     char fileName[256];
5     if (GetFileName(fileName, 256))
6     {
7       tShellReader sr(fileName);
8       sr.Execute();
9       tShell* newShell = sr.GetOutput();
10
11      if (newShell)
12        GetScene().GetDoc().SetShell(newShell);
13    }
14  }
```

Programa 9.12: Exemplo de utilização de um fonte.

Na linha 5 executamos `GetFileName()`. A função exibe uma caixa de diálogo solicitando ao usuário o nome de um arquivo, o qual será armazenado em `fileName`. Se `GetFileName()` retornar verdadeiro, construímos, na linha 7, um objeto da classe `tShellReader`, um *leitor* de modelos de cascas, passando como parâmetro o nome do arquivo obtido na linha 5. Em seguida, enviamos a mensagem `Execute()` ao objeto. O método `tShellReader::Execute()` abre o arquivo passado no construtor e inicia sua leitura. O arquivo de entrada de `tShellReader` é um arquivo texto que contém a descrição de uma casca de acordo com as regras da seguinte *gramática* [2] (os símbolos em **`negrito`** são terminais):

```
model ::= <transf_def> shell_def
transf_def ::= transform '{' transf_list '}'
transf_list ::= transf
  | transf_list transf
transf ::= translate point
  | scale point
  | rotate point float
shell_def ::= shell name '{' shell_body '}'
shell_body ::= vertex_def face_def_list
vertex_def ::= vertices '{' vertex_list '}'
vertex_list ::= vertex
```

```
    | vertex_list vertex
  vertex ::= integer point
  face_def_list ::= face_def
    | face_def_list face_def
  face_def ::= face '{' loop_def_list '}'
  loop_def_list ::= loop_def
    | loop_def_list loop_def
  loop_def ::= loop '{' vertex_range_list '}'
  vertex_range_list ::= vertex_range
    | vertex_range_list ',' vertex_range
  vertex_range ::= integer
    | integer to integer
  point ::= '<'float','float',' float'>'
```

Na linha 9 obtemos, com a mensagem `GetOutput()`, um ponteiro para o modelo de cascas gerado pelo fonte. Se o arquivo de entrada possuir erros de sintaxe, o método `tShellReader::GetOutput()` retorna um ponteiro nulo. Por isso, na linha 11, testamos se a execução do processo de leitura foi bem sucedida. Se tivermos um modelo de cascas, enviamos a mensagem `SetShell()` ao documento da cena da vista. O método `tShellDoc::SetShell()`, declarado na linha 11 do Programa 9.4, destrói o modelo de cascas do documento, se houver um, e torna `shell` o modelo corrente. Em seguida, envia a mensagem `Regenerate()` para as cenas do documento. No final, as vistas conterão a imagem do modelo construído pelo leitor da classe `tShellReader`.

9.10 Usando Filtros

No Capítulo 7 vimos que filtros são processos de transformação de dados em um diagrama de fluxo de dados. O comando `generate_mesh`, definido na Seção 9.8, utiliza um filtro da classe `t2DMeshGenerator`. O objeto representa um processo que transforma um modelo geométrico de cascas ou de sólidos em um modelo de decomposição por células. (No Capítulo 8 comentamos que um objeto pode representar um processo puro, um dado puro ou, mais comumente, combinações entre ambos.) `t2DMeshGenerator` executa seu trabalho com a colaboração do modelo geométrico de entrada. No Capítulo 6 fizemos uma referência à função `bSplitEdges()`, responsável pela geração do modelo de contornos de faces de uma casca. Na implementação orientada a objetos, a função é o método virtual `SplitEdges()`, definido na classe `tShell` (versão C++ da estrutura `bModel` do Capítulo 3). Durante a transformação do modelo de cascas em uma malha de elementos, o filtro `t2DMeshGenerator` envia uma mensagem `SplitEdges()` para o modelo de entrada, solicitando a geração do modelo de contornos de faces intermediário. O método `SplitEdges()` foi implementado na classe `tShell` porque não exigimos que um filtro gerador de malhas compreendesse a (complicada) estrutura interna de um modelo de cascas. Novamente, tinhamos duas escolhas: ou o modelo geométrico conheceria algo a respeito do processo de geração de malhas ou forneceria métodos para o gerador de malhas acessar suas informações. Nesse caso, a primeira estratégia pareceu mais atrativa, resultando em uma interface mais simples para a classe `tShell`. Os filtros de OSW são objetos que trabalham em parceria com os modelos de entrada e saída. Justificaremos sucintamente por quê.

Poderíamos implementar um filtro como um método do modelo a ser transformado. Por exemplo, a classe `tShell` poderia ter um método chamado `GenerateMesh()` que

geraria a malha de elementos da casca. Com essa alternativa, um repositório de dados e seus processos de transformação formariam um único objeto. De fato, definimos um objeto como sendo uma combinação de estruturas de dados e de procedimentos que manipulam as estruturas de dados. A vantagem é que os processos de transformação de dados possuem, nesse caso, acesso total aos dados, resultando em uma computação mais eficiente. Em contrapartida, devemos duplicar a implementação dos algoritmos de transformação em cada classe de modelo. (O método `GenerateMesh()` deve ser duplicado na classe `tSolid`, por exemplo.)

Alternativamente, poderíamos implementar filtros e modelos como objetos separados, com os modelos fornecendo somente funções de acesso aos dados e com os filtros executando as operações de transformações de dados. O código resultante pode ser mais simples e modular, e mais fácil de entender, manter e estender. No entanto, a interface entre modelos e filtros deve ser mais formal e cuidadosamente projetada para garantir performance e flexibilidade adequadas. Escolhemos uma terceira alternativa: implementar filtros e modelos como objetos separados, mas cooperando entre si. Modelos não fornecem somente métodos de acesso aos dados, mas executam também algumas operações de transformação. Consideremos, para exemplificar, os comandos `contourxx`, executados pelas vistas das cenas do modelo mecânico de uma casca. A implementação é apresentada no Programa 9.13. A seguir, comentamos alguns trechos do programa.

Linha 5-10 Usamos a macro `COMMAND_AND_ID` para definir os comandos. Como `COMMAND`, a macro toma três parâmetros: o nome do comando, o identificador numérico do comando e o método de classe que implementa o comando. Diferente de `COMMAND`, o método de implementação do comando toma como parâmetro o identificador numérico do comando. Com isso, podemos declarar comandos diferentes na tabela de comandos da vista que são implementados pelo mesmo método da classe.

Linha 13 Os comandos são implementados pelo método `CmContour` da classe `tMecView` (não apresentamos a definição de `tMecView`). O parâmetro do método identifica qual dos 6 comandos definidos na tabela de comandos da classe foi selecionado. (Observe, na tabela de comandos, que os 6 comandos são implementados pelo mesmo método.) Usaremos o identificador do comando para definir qual campo escalar deve ser extraído do modelo (deslocamentos u, v, w ou rotações θ_x, θ_y, θ_z).

Linha 21 Se a malha de elementos existir, construímos um objeto da classe `tScalarExtractor`. O objeto é um filtro que toma como entrada um modelo de decomposição por células e gera como saída o mesmo modelo de entrada. A função do filtro é simples: extrai o valor do grau de liberdade número `dof` de cada vértice do modelo e armazena no campo `Scalar` do vértice, tal como discutido no Capítulo 7. O filtro também determina os valores mínimo e máximo do campo escalar.

Linha 22 Construímos um objeto da classe `tContourFilter`. O objeto é um filtro que toma como entrada um modelo de decomposição por células e produz como saída um modelo gráfico que contém os isopontos, as isolinhas e as isosuperfícies para determinada faixa de valores escalares (no caso de uma casca, somente isolinhas).

```
1   #include "shellview.h"
2
3   DEFINE_COMMAND_TABLE1(tMecView, tView)
4      ...
5      COMMAND_AND_ID("contourUx", 200, CmContour),
6      COMMAND_AND_ID("contourUy", 201, CmContour),
7      COMMAND_AND_ID("contourUz", 202, CmContour),
8      COMMAND_AND_ID("contourRx", 203, CmContour),
9      COMMAND_AND_ID("contourRy", 204, CmContour),
10     COMMAND_AND_ID("contourRz", 205, CmContour),
11  END_COMMAND_TABLE;
12
13  void
14  tMecView::CmContour(int id)
15  {
16     tMesh* mesh = GetScene().GetMesh();
17
18     if (mesh)
19     {
20        int dof = id - 200;
21        tScalarExtractor se;
22        tContourFilter cf;
23
24        se.SetInput(mesh);
25        se.ExtractField(dof, true);
26        se.Execute();
27        cf.SetInput(se.GetOutput();
28        cf.GenerateValues(10,
29           se.GetMinScalar(), se.GetMaxScalar());
30        cf.Execute();
31
32        tGraphicModel* contour = cf.GetOutput();
33
34        if (contour)
35        {
36           tPosScene* posScene = new tPosScene(Scene->Doc);
37
38           posScene->InitDefaultViews();
39           pecScene->AddActor(new tActor(contour));
40        }
41     }
42  }
```

Programa 9.13: Implementação dos comandos de geração de isolinhas.

Linha 30 Enviamos a mensagem `Execute()` ao objeto `cf`. O método `Execute()` da classe `tContourFilter` envia uma mensagem ao modelo de decomposição por células solicitando um *iterator* para a coleção de células do modelo. De posse do *iterator*, o filtro envia para cada célula do modelo a mensagem `Contour()`. O método é declarado como virtual puro na classe abstrata `tCell` e, portanto, deve ser sobrecarregado em todas as classes (não-abstratas) derivadas de `tCell`. (Uma versão C para células quadrilaterais é apresentada no Programa 7.9.) O método `Execute()` é implementado no Programa 9.14. Note a cooperação entre as classes `tContourFilter` e as classes derivadas de `tCell`.

```
1   void
2   tContourFilter::Execute()
3   {
4     if (!Input)
5       return;
6     if (Output)
7       Output->Delete();
8     Output = new tGraphicModel(0);
9
10    tCellIterator cit = Input->GetCellIterator();
11
12    while (cit)
13    {
14      tCell* cell = cit++;
15
16      for (int i = 0; i < NumberOfValues; i++)
17        cell->Contour(Values[i], output);
18    }
19  }
```

Programa 9.14: Implementação do filtro de geração de isolinhas.

9.11 Usando Mapeadores

Vimos no Capítulo 7 que um mapeador é um processo sumidouro de dados. Em OSW, um mapeador é um objeto de uma classe `tMapper`. Quando construímos um ator, um mapeador é automaticamente criado para o ator. Um mapeador toma como entrada qualquer modelo geométrico de OSW.

O objeto mais importante de um mapeador é sua tabela de cores, instância da classe `tLookupTable`. A tabela de cores é usada pelo mapeador para atribuir cores aos vértices do modelo geométrico de entrada, a partir das quais o *renderer* da cena sintetiza uma imagem colorida do modelo. Podemos alterar a tabela de cores de um mapeador ou solicitar ao mapeador que não use a tabela de cores. Nesse caso, as cores dos vértices do modelo serão determinadas em função das propriedades dos materiais que constituem a superfície do modelo, de acordo com o modelo de iluminação definido no Capítulo 7. O Programa 9.15 ilustra a utilização de um mapeador.

Nas linhas 1 e 2 declaramos um ponteiro para um ator e um mapeador, respectivamente. Na linha 3, construímos dinamicamente um objeto da classe `tLookupTable`.

```
1   tActor* actor;
2   tMapper* mapper;
3   tLookupTable* colorTable = new tLookupTable(256);
4   ...
5   mapper = actor->GetMapper();
6   mapper->SetLookupTable(colorTable);
7   mapper->UseScalars(true);
8   colorTable->Delete();
9   ...
```

Programa 9.15: Usando um mapeador.

O parâmetro do construtor é um inteiro que define o número de cores da tabela. O construtor inicializa 256 cores, variáveis do azul até o vermelho. `tLookupTable` oferece métodos que nos permitem especificar, no modelo de cores HSV definido no Capítulo 7, a faixa de cores de uma tabela de cores (veja a classe `tLookupTable` no Capítulo 10).

Vamos supor que entre as linhas 3 e 5 o ator seja construído. Na linha 5, enviamos a mensagem `GetMapper()` ao ator, a qual retorna o mapeador de `actor`. Na linha 6 enviamos a mensagem `SetLookupTable` ao objeto `mapper`. O método toma como parâmetro um ponteiro para a nova tabela de cores usada pelo mapeador, `colorTable`. Na linha 7 ordenamos que o mapeador utilize os escalares do modelo geométrico para determinar as cores dos vértices do modelo. Essas cores são mapeadas de `colorTable`, de acordo com o valor do escalar do vértice.

Na linha 8, enviamos a mensagem `Delete()` ao objeto `colorTable`, solicitando que a tabela seja eliminada. Da mesma forma que os modelos discutidos na Seção 9.6, a classe `tLookupTable` é derivada de `tObjectBody`. Quando construímos o objeto, na linha 3, seu contador de referência passou a ter o valor 1. `tMapper::SetLookupTable()`, na linha 6, incrementa o contador de referência de `colorTable`, pois agora a tabela é usada por `mapper`. O método `Delete()` decrementa o contador de referência de `colorTable`, mas o objeto não é destruído porque está sendo utilizado por `mapper`. Uma tabela de cores pode ser compartilhada por vários atores de uma cena.

9.12 Definindo Elementos Finitos

Um elemento finito é um objeto de uma classe derivada da classe `tFiniteElement`. Um elemento finito genérico é responsável pela computação de sua matriz de rigidez e pela computação de seu vetor de esforços equivalentes (estamos considerando a aplicação do elemento somente em problemas elastostáticos). Para definirmos um elemento finito, devemos:

Passo 1 Declarar uma classe derivada da classe `tFiniteElement`.

Passo 2 Associar uma classe de célula a nova classe de elemento finito.

Passo 3 Sobrecarregar os métodos virtuais de cálculo da matriz de rigidez e do vetor de esforços equivalentes, declarados na classe `tFiniteElement`.

O Programa 9.16 mostra o arquivo de cabeçalho `fe3nshell.h`, o qual contém a definição da classe do elemento finito de casca, chamada `tFE3NShell`.

```
1   #include "fem.h"
2   class tFE3NShell:virtual public tFiniteElement, t3N2DCell
3   {
4     public:
5       // Constructor
6       tFE3NShell();
7
8       double  GetTickness() const;
9       void    SetTickness(double);
10
11    protected:
12      tMatrix ComputeStiffnessMatrix();
13      tVector ComputeLoadVector();
14
15      double  Tickness;
16
17  }; // tFE3NShell
```

Programa 9.16: Definindo o elemento finito de casca.

Na linha 1 incluímos o arquivo de cabeçalho `fem.h`, o qual contém a definição da classe `tFiniteElement`. Na linha 2 declaramos a classe `tFE3NShell`. A classe deriva virtualmente de `tFiniteElement` porque `tFiniteElement` deriva virtualmente de `tCell` (um elemento finito *é* uma célula de um modelo de decomposição por células que possui um comportamento especializado). Mas a classe `tCell` (veja o Capítulo 10) também é uma classe abstrata. Além disso, um elemento finito de casca *não é* uma célula genérica, mas sim uma célula de dimensão topológica 2, definida por 3 nós. Por isso, a classe `tFE3NShell` deriva de `t3N2DCell`. Contudo, `t3N2DCell` também deriva (virtualmente) de `tCell` (é claro que uma célula de dimensão topológica 2 com 3 nós *é* uma célula). Se não derivássemos `tFE3NShell` virtualmente de `tFiniteElement`, um elemento finito de casca teria *duas* células: uma herdada de `tFiniteElement` e outra herdada de `t3N2DCell`. Em determinadas situações é exatamente isso que queremos, mas não nesse caso: um elemento finito de casca é uma *única* célula do modelo de decomposição por células. Usamos herança múltipla para associarmos uma célula ao elemento finito. STROUSTRUP [110] discute com detalhes o que são e como usar classes bases virtuais (é o próprio inventor do engenho).

Nas linhas 8 e 9 declaramos os métodos `GetTickness()` e `SetTickness()`, respectivamente. Métodos desse tipo, conhecidos como *getters* e *setters*, são responsáveis pela leitura e escrita de atributos protegidos ou privados do objeto, nesse caso, a espessura `Tickness` do elemento, declarada na linha 15.

Na linha 12 declaramos o método `ComputeStiffnessMatrix()`, embora o método não seja declarado como virtual puro na classe base `tFiniteElement` (veja a classe `tFiniteElement` no Capítulo 10). O método calcula a matriz de rigidez do elemento de casca em coordenadas locais, de acordo com as equações apresentadas no Capítulo 6, e transforma a matriz para o sistema global de coordenadas. Na linha 13 declaramos o método `ComputeLoadVector()`, herdado de `tFiniteElement`. O método considera somente o carregamento distribuído por metro quadrado de área correspondente ao peso próprio da estrutura. Isso é tudo que temos a fazer com um elemento finito.

9.13 Definindo Analisadores

Um analisador é um objeto de uma classe derivada da classe abstrata `tSolver`. A interface pública de `tSolver` contém métodos virtuais responsáveis pelas seguintes tarefas relacionadas à análise numérica de um modelo:

1. Início do processo de análise.
2. Verificação da integridade dos dados do modelo mecânico.
3. Montagem do sistema de equações lineares.
4. Resolução do sistema de equações lineares.
5. Término do processo de análise.

Para analisar um modelo mecânico, devemos:

Passo 1 Construir um objeto de uma classe derivada da classe `tSolver`. O construtor da classe derivada deve tomar como parâmetro uma referência para o modelo mecânico, utilizada para inicializar a classe base `tSolver`.

Passo 2 Enviar a mensagem `Run()` ao objeto construído.

Por exemplo:

```
tMesh* mesh = new tMesh;
...
tSolver* solver = new tFESolver(*mesh);
solver->Run();
```

Nesse exemplo, construímos um objeto da classe `tFESolver`, derivada de `tSolver`. O objeto é um analisador de estruturas elastostáticas pelo método dos elementos finitos. Para definirmos um analisador tal como `tFESolver`, devemos:

Passo 1 Derivar uma classe da classe abstrata `tSolver`.

Passo 2 Sobrecarregar, na nova classe, o método virtual `AssembleSystem()`, herdado de `tSolver`. *Podemos* sobrecarregar outros métodos virtuais de `tSolver`, mas *devemos* sobrecarregar `AssembleSystem()`.

A definição da classe `tFESolver` é apresentada no Programa 9.17.

Na linha 1 incluímos o arquivo de cabeçalho `solver.h`, o qual contém a definição da classe base `tSolver`. Na linha 2 declaramos `tFESolver`, derivada de `tSolver`. Na linha 6 declaramos o construtor da classe. O construtor toma como parâmetro uma referência `mesh` para malha do modelo mecânico. Usamos `mesh` para inicializar `tSolver`. Na linha 12 declaramos o método `AssembleSystem()`. Em `tSolver`, o método é declarado como virtual puro porque um analisador genérico não sabe como montar o sistema de equações lineares. A montagem do sistema depende do método particular de análise, como visto no Capítulo 5. Nas linhas 11 e 13 sobrecarregamos, respectivamente, os métodos `ConstructLinearSystem()` e `Terminate()`, ambos herdados de `tSolver`. O primeiro é responsável pela construção do sistema linear do analisador. `tSolver::ConstructLinearSystem()` constrói um sistema da classe `tFullSystem`.

```
1   #include "solver.h"
2   class tFESolver:public tSolver
3   {
4     public:
5       // Constructor
6       tFESolver(tMesh& mesh):
7         tSolver(mesh)
8       {}
9
10    protected:
11      tLinearSystem* ConstructLinearSystem();
12      void           AssembleSystem();
13      void           Terminate();
14
15  }; // tFESolver
```

Programa 9.17: Definindo o analisador MEF.

O objeto é um sistema linear de equações com matriz completa. Sobrecarregamos o método para construirmos um sistema da classe `tBandSystem`, cuja matriz é simétrica e armazenada em banda. O segundo método termina o processo de análise. Sobrecarregamos `Terminate()` para calcularmos os esforços nodais do modelo. Vejamos o que acontece quando um objeto da classe `tFESolver` executa o método virtual `Run()`. A implementação de `tSolver::Run()` é mostrada no Programa 9.18.

```
1   void
2   tSolver::Run()
3   {
4     try
5     {
6       Init();
7       AssembleSystem();
8       LS->Solve();
9       Terminate();
10    }
11    catch (tXSolver& x)
12    {
13      Resume(x);
14    }
15  }
```

Programa 9.18: Analisando o modelo mecânico.

9.13.1 Iniciando a Análise

`tSolver::Run()` começa executando o método virtual `Init()` (linha 6 do Programa 9.18), declarado em `tSolver`. O método é responsável pela inicialização do processo de análise. `tSolver::Init()` executa o método virtual `Check()`, responsável pela verificação da integridade dos dados do modelo mecânico. `tSolver::Check()`

envia a mensagem `Check()` para todas as células do modelo mecânico. Isso significa que o analisador pergunta, para cada célula do modelo: "célula, tudo bem com você?" Se todas as células responderem positivamente, a análise prossegue. Caso contrário, uma exceção da classe `tXSolver` é gerada (veja a classe `tXSolver` no Capítulo 10). `tSolver::Run()` trata a exceção executando o método virtual `Resume()` (linha 13 do Programa 9.18). `tSolver::Resume()` simplesmente exibe a mensagem de erro definida no objeto da classe `tXSolver`. As classes derivadas de `tSolver` podem especializar o tratamento de erros de análise sobrecarregando `Resume()`.

Após a execução do método `Check()`, `tSolver::Init()` executa o método virtual `ConstructLinearSystem()`. Sobrecarregamos o método em `tFESolver` para construirmos um sistema de equações com a matriz simétrica armazenada em banda, como discutido anteriormente. Se sobrecarregarmos `Init()` em uma classe derivada, devemos executar `Check()` e `ConstructLinearSystem()`.

9.13.2 Montando o Sistema de Equações

Depois de inicializar a análise, `tSolver::Run()` executa `AssembleSystem()` (linha 7 do Programa 9.18), responsável pela montagem do sistema de equações lineares construído com `ConstructLinearSystem()`. `tFESolver::AssembleSystem()` é apresentado no Programa 9.19 (não consideramos as forças aplicadas diretamente nos vértices do modelo estrutural).

```
1   #include "fesolver.h"
2   void
3   tFESolver::AssembleSystem()
4   {
5     tCellIterator cells = Mesh->GetCellsIterator();
6
7     while (cells)
8     {
9       tFiniteElement* fe = (tFiniteElement*)(cells++);
10      tLocationArray* la = fe->GetLocationArray();
11
12      LS->Assemble(fe->GetStiffnessMatrix(), *la);
13      LS->Assemble(fe->GetLoadVector(), *la);
14    }
15  }
```

Programa 9.19: Montando o sistema no analisador MEF.

Na linha 1 incluímos o arquivo de cabeçalho `fesolver.h`, o qual contém a definição da classe `tFESolver`. A montagem do sistema começa na linha 2. Inicialmente, o método envia uma mensagem ao modelo de decomposição por células solicitando um *iterator* de células. Para cada célula do modelo de decomposição por células, ou seja, para cada elemento finito do modelo mecânico, o método envia as mensagens `GetLocationArray()`, `GetStiffnessMatrix()` e `GetLoadVector()`. O método `GetLocationArray()` retorna um ponteiro para o vetor de localização do elemento. A função do vetor de localização é identificar, para cada grau de liberdade do elemento (18, no caso do elemento de casca), qual é a equação correspondente no sistema

linear. Se um grau de liberdade tiver uma condição de contorno do tipo essencial (por exemplo, se o deslocamento na direção x de um dos nós do elemento for prescrito), então o grau de liberdade não é incógnita do problema. Nesse caso, não haverá uma equação correspondente no sistema de equações (veja a classe `tLocationArray` no Capítulo 10). O método `GetStiffnessMatrix()` retorna a matriz de rigidez do elemento. Se a matriz ainda não foi calculada, o método executa o método virtual `ComputeStiffnessMatrix()`, declarado na linha 12 do Programa 9.16. O método `GetLoadVector()` retorna o vetor de esforços nodais equivalentes do elemento. Se o vetor ainda não foi calculado, o método executa o método virtual `ComputeLoadVector()`, declarado na linha 13 do Programa 9.16.

Em seguida, `tFESolver::AssembleSystem()` envia ao sistema linear, atributo `*LS` herdado de `tSolver`, mensagens solicitando a adição das contribuições do elemento finito aos lados esquerdo e direito, linhas 12 e 13 do Programa 9.19, respectivamente. Note que delegamos ao próprio sistema linear a tarefa de adicionar as contribuições de um elemento, através dos métodos virtuais `Assemble()`, declarados como virtuais puros na classe abstrata `tLinearSystem`. Se quisermos empregar um outro tipo de sistema linear, por exemplo, um sistema com a matriz armazenada em perfil, podemos definir uma classe derivada de `tLinearSystem` e sobrecarregar os método virtuais `Assemble()` (veja a classe `tLinearSystem` no Capítulo 10). Para utilizar o sistema linear em `tFESolver`, basta reescrever o método virtual `ConstructLinearSystem()`. O restante do código funciona sem alterações.

9.13.3 Resolvendo o Sistema de Equações

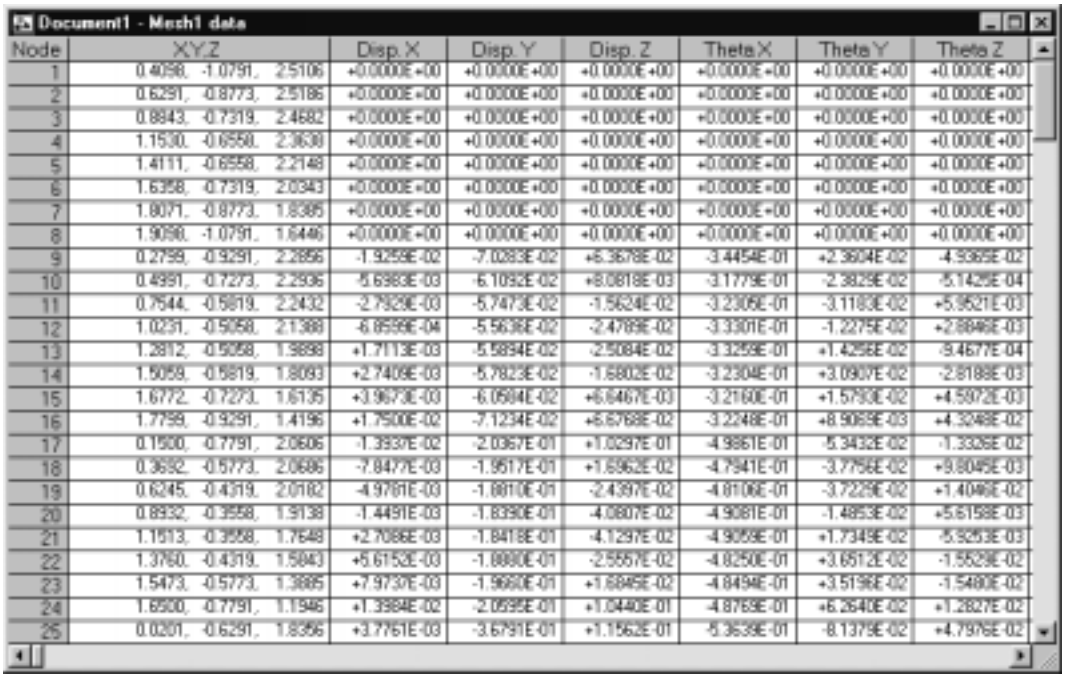
Uma vez montado o sistema de equações, `tSolver::Run()` envia a mensagem `Solve()` a `*LS`, linha 8 do Programa 9.18. O método, declarado como virtual puro na classe base `tLinearSystem()`, é responsável pela resolução numérica do sistema de equações [1]. Se o sistema for singular, o método gera uma exceção da classe `tXSolver`. `tSolve` deve ser sobrecarregado nas classes derivadas de `tLinearSystem` (veja as classes `tFullSystem` e `tBandSystem` no Capítulo 10).

9.13.4 Terminando a Análise

Finalizando o processo de análise numérica, `tSolve::Run()` executa o método virtual `Terminate()`, linha 9 do Programa 9.18. `tFESolver::Terminate:()`, declarado na linha 13 do Programa 9.17, transfere os elementos do vetor solução de `*LS` para os graus de liberdade correspondentes no modelo mecânico. Em seguida, calcula os esforços nos vértices do elemento. Finalmente, temos o modelo pronto para o processo de visualização. No Capítulo 11 mostraremos alguns exemplos. Os resultados da análise são mostrados na Figura 9.6.

9.14 Sumário

Nesse Capítulo definimos o que é uma aplicação de modelagem orientada a objetos e mostramos como utilizar algumas classes de objetos de OSW na construção de um programa de análise e visualização de modelos de cascas.



Document1 - Mesh1 data

Node	XYZ	Disp.X	Disp.Y	Disp.Z	ThetaX	ThetaY	ThetaZ
1	0.4098, -1.0791, 2.5106	+0.0000E+00	+0.0000E+00	+0.0000E+00	+0.0000E+00	+0.0000E+00	+0.0000E+00
2	0.6291, -0.8773, 2.5186	+0.0000E+00	+0.0000E+00	+0.0000E+00	+0.0000E+00	+0.0000E+00	+0.0000E+00
3	0.8843, -0.7319, 2.4682	+0.0000E+00	+0.0000E+00	+0.0000E+00	+0.0000E+00	+0.0000E+00	+0.0000E+00
4	1.1530, -0.6558, 2.3639	+0.0000E+00	+0.0000E+00	+0.0000E+00	+0.0000E+00	+0.0000E+00	+0.0000E+00
5	1.4111, -0.6558, 2.2148	+0.0000E+00	+0.0000E+00	+0.0000E+00	+0.0000E+00	+0.0000E+00	+0.0000E+00
6	1.6358, -0.7319, 2.0343	+0.0000E+00	+0.0000E+00	+0.0000E+00	+0.0000E+00	+0.0000E+00	+0.0000E+00
7	1.8071, -0.8773, 1.8385	+0.0000E+00	+0.0000E+00	+0.0000E+00	+0.0000E+00	+0.0000E+00	+0.0000E+00
8	1.9098, -1.0791, 1.6446	+0.0000E+00	+0.0000E+00	+0.0000E+00	+0.0000E+00	+0.0000E+00	+0.0000E+00
9	0.2759, -0.9291, 2.2856	-1.9259E-02	-7.0283E-02	+6.3678E-02	-3.4454E-01	+2.3604E-02	-4.9365E-02
10	0.4991, -0.7273, 2.2936	-5.6383E-03	-6.1092E-02	+8.0818E-03	-3.1779E-01	-2.3829E-02	-5.1425E-04
11	0.7544, -0.5819, 2.2432	-2.7929E-03	-5.7473E-02	-1.5624E-02	-3.2305E-01	-3.1183E-02	+5.9521E-03
12	1.0231, -0.5058, 2.1388	-6.8559E-04	-5.5639E-02	-2.4769E-02	-3.3301E-01	-1.2275E-02	+2.8848E-03
13	1.2812, -0.5058, 1.9898	+1.7113E-03	-5.5894E-02	-2.5084E-02	-3.3259E-01	+1.4256E-02	-9.4677E-04
14	1.5059, -0.5819, 1.8093	+2.7409E-03	-5.7823E-02	-1.6802E-02	-3.2304E-01	+3.0907E-02	-2.8188E-03
15	1.6772, -0.7273, 1.6135	+3.9878E-03	-6.0584E-02	+6.8467E-03	-3.2160E-01	+1.5793E-02	+4.5972E-03
16	1.7759, -0.9291, 1.4196	+1.7500E-02	-7.1234E-02	+6.6768E-02	-3.2248E-01	+8.9069E-03	+4.3248E-02
17	0.1500, -0.7791, 2.0606	-1.3937E-02	-2.0367E-01	+1.0297E-01	-4.9861E-01	-5.3432E-02	-1.3326E-02
18	0.3692, -0.5773, 2.0686	-7.8477E-03	-1.9517E-01	+1.6962E-02	-4.7941E-01	-3.7756E-02	+9.8049E-03
19	0.6245, -0.4319, 2.0182	-4.9761E-03	-1.8810E-01	-2.4397E-02	-4.8106E-01	-3.7229E-02	+1.4046E-02
20	0.8932, -0.3558, 1.9138	-1.4491E-03	-1.8390E-01	-4.0807E-02	-4.9081E-01	-1.4853E-02	+5.6158E-03
21	1.1513, -0.3558, 1.7648	+2.7088E-03	-1.8418E-01	-4.1297E-02	-4.9059E-01	+1.7345E-02	-5.9253E-03
22	1.3760, -0.4319, 1.5843	+5.6152E-03	-1.8880E-01	-2.5557E-02	-4.8250E-01	+3.6512E-02	-1.5529E-02
23	1.5473, -0.5773, 1.3885	+7.9737E-03	-1.9660E-01	+1.6845E-02	-4.8494E-01	+3.5196E-02	-1.5480E-02
24	1.6500, -0.7791, 1.1946	+1.3984E-02	-2.0595E-01	+1.0440E-01	-4.8769E-01	+6.2640E-02	+1.2827E-02
25	0.0201, -0.6291, 1.8356	+3.7761E-03	-3.6791E-01	+1.1562E-01	-5.3639E-01	-8.1379E-02	+4.7976E-02

Figura 9.6: Janela de resultados da análise.

Uma aplicação é um objeto de uma classe derivada de `tApplication`. Um objeto de aplicação é responsável pela inicialização da aplicação, inicialização da janela principal da aplicação e manipulação da base de dados da aplicação.

A base de dados da aplicação é um objeto de uma classe derivada de `tDocument`. Um documento é uma coleção de modelos e de cenas persistentes. Um modelo é um objeto de uma classe derivada da classe abstrata `tModel`. Uma cena é um objeto de uma classe derivada da classe `tScene`.

Uma cena contém coleções de atores, luzes e câmeras. Um ator é um objeto da classe `tActor`; uma luz é um objeto de uma classe derivada de `tLight` e uma câmera é um objeto da classe `tCamera`. Uma cena possui, ainda, uma coleção de elementos de interface chamados vistas.

Uma vista é um objeto de uma classe derivada da classe `tView`. Uma vista possui uma tabela de comandos e um objeto de uma classe derivada da classe abstrata `tRenderer`, responsável pela síntese de imagens da cena da vista. Podemos definir comandos para criação de modelos, transformação de modelos e visualização de modelos. Criamos modelos usando objetos de classes derivadas da classe abstrata `tSource`. Transformamos modelos usando objetos de classes derivadas da classe abstrata `tFilter`. A transformação de dados em imagem é comandada por objetos da classe `tMapper`.

Um elemento finito é um objeto de uma classe derivada de `tFiniteElement`. Um elemento finito genérico deve ser capaz de calcular sua matriz de rigidez e seu vetor de esforços equivalentes. A matriz de rigidez e o vetor de esforços nodais de um elemento finito são usados por um analisador MEF, um objeto da classe `tFESolver`, para montagem e resolução do sistema de equações lineares do problema.

CAPÍTULO 10

As Bibliotecas de Classes

10.1 Funções Globais de OSW

Nessa Seção apresentamos, em ordem alfabética, as funções globais de OSW.

Função IsEqual — pmath.h

```
inline bool IsEqual(double a, double b);
```
Retorna **true** se os números reais `a` e `b` são iguais.

Função IsGreater — pmath.h

```
inline bool IsGreater(double a, double b);
```
Retorna **true** se o número real `a` é maior que o número real `b`; retorna **false** caso contrário.

Função IsGreaterEqual — pmath.h

```
inline bool IsGreaterEqual(double a, double b);
```
Retorna **true** se o número real `a` é maior ou igual ao número real `b`; retorna **false** caso contrário.

Veja também `::IsGreater`, `::IsEqual`

Função IsLesser — pmath.h

```
inline bool IsLesser(double a, double b);
```
Retorna **true** se o número real `a` é menor que o número real `b`; retorna **false** caso contrário.

Função IsLesserEqual — pmath.h

```
inline bool IsLesserEqual(double a, double b);
```
Retorna **true** se o número real `a` é menor ou igual ao número real `b`; retorna **false** caso contrário.

Veja também `::IsLesser`, `::IsEqual`

Função IsNegative — pmath.h

```
inline bool IsNegative(double a);
```
Retorna **true** se o número real `a` é menor que zero; retorna **false** caso contrário.

Função IsPositive — pmath.h

```
inline bool IsPositive(double a);
```
Retorna **true** se o número real `a` é maior que zero; retorna **false** caso contrário.

Função IsZero — pmath.h

```
inline bool IsZero(double a);
```
Retorna **true** se o número real `a` é igual a zero.

Função ToDegrees — pmath.h

```
inline double ToDegress(double a);
```
Retorna o valor de `a` em graus, no intervalo $[0, 360]$.

Função ToRadians — pmath.h

```
inline double ToRadians(double a);
```
Retorna o valor de `a` em radianos, no intervalo $[0, 2\pi]$.

10.2 Macros de OSW

Nessa Seção apresentamos, em ordem alfabética, os macros de OSW. Um macro (ou uma macro) é um identificador definido pela diretiva de pré-processamento `#define`. Um macro pode ter argumentos e um corpo, como mostrado no exemplo a seguir:

```
#define max(a,b) ((a) > (b) ?  (a) :  (b))
```

Durante o pré-processamento, o pré-processador C++ *espande* cada ocorrência do identificador `max` pelo corpo correspondente. (Em C++, a utilização de macros como no exemplo acima é obsoleta, sendo preferido o emprego de funções **inline**. A vantagem é que, no caso de uma função **inline**, o compilador verifica o tipo dos parâmetros e avalia seus valores antes da expansão da função.)

Macro COMMAND — cmdhandl.h

O macro `COMMAND` é utilizado para definição de um comando em uma tabela de comandos de uma vista, como discutido no Capítulo 9. O macro possui três parâmetros: um número inteiro identificador do comando, uma cadeia de caracteres que contém o nome do comando e o nome do método da classe de vista que implementa o comando.

Exemplo

O exemplo a seguir ilustra a declaração, em um arquivo de cabeçalhos `myview.h`, da tabela de comandos de uma classe de vista genérica chamada `tMyView`. A definição da tabela é feita no arquivo `myview.cpp`.

```
// arquivo myview.h
#include "view.h"

class tMyView:  public tView
{
   public:
      ...
      void CmCommandA();
      void CmCommandB();
      void CmCommandX(int);
      ...
      DECLARE_COMMAND_TABLE(tMyView);
}; // tMyView

// arquivo myview.cpp
#include "myview.h"

DEFINE_COMMAND_TABLE1(tMyView, tView)
   COMMAND(201, "Command_1", CmCommandA),
   COMMAND(202, "Command_2", CmCommandB),
   COMMAND_AND_ID(203, "Command_3", CmCommandX),
   COMMAND_AND_ID(204, "Command_4", CmCommandX),
END_COMMAND_TABLE;
...
```

O macro `DECLARE_COMMAND_TABLE` declara que a classe `tMyView`, derivada de `tView`, possui uma tabela de comandos. O macro `DEFINE_RESPONSE_TABLE` define a tabela de comandos da classe `tMyView`. O macro define, ainda, que a tabela de comandos de `tMyView` é uma extensão da tabela de comandos da classe base `tView`. Os comandos da tabela podem ser definidos com os macros `COMMAND` e `COMMAND_AND_ID`. Note que os métodos de classe que implementam comandos definidos com o macro `COMMAND` não tomam quaisquer argumentos.

Veja também Macro `COMMAND_AND_ID`

Macro COMMAND_AND_ID **cmdhandl.h**

Além do macro `COMMAND`, podemos utilizar também o macro `COMMAND_AND_ID` para definirmos um comando de uma tabela de comandos de uma vista. O macro toma três argumentos: um número inteiro identificador do comando, uma cadeia de caracteres que contém o nome do comando e o método da classe de vista que implementa o comando. Diferentemente do método que implementa um comando definido com o macro `COMMAND`, o método que implementa um comando definido com o macro `COMMAND_AND_ID` toma como parâmetro um número inteiro igual ao identificador do comando. Com isso, podemos utilizar o mesmo método para implementar comandos distintos (veja o exemplo do macro `COMMAND`).

Veja também Macro COMMAND

Macro DECLARE_COMMAND_TABLE cmdhandl.h

O macro DECLARE_COMMAND_TABLE declara que uma classe derivada da classe tView possui uma tabela de comandos. O macro toma como argumento o nome da classe e deve ser utilizado dentro da declaração da classe de vista (veja o exemplo do macro COMMAND).

Veja também Macro DEFINE_COMMAND_TABLEx

Macro DECLARE_ERROR_MESSAGE_TABLE errhandl.h

O macro DECLARE_ERROR_MESSAGE_TABLE, quando utilizado dentro da definição de uma classe derivada de tErrorHandler, declara que a classe possui uma tabela de mensagens de erro. As classes OSW derivadas de tErrorHandler são, usualmente, analisadores sintáticos de arquivos de entrada. A classe tMeshReader é um exemplo. Um objeto da classe tMeshReader é um leitor de modelos de decomposição por células (vimos a definição de *leitor* no Capítulo 7). As mensagens dos erros occorridos durante a leitura do modelo de decomposição por células, se houverem, são definidas em uma tabela de tMeshReader declarada com o macro DECLARE_ERROR_MESSAGE. O macro toma como parâmetro o nome da classe.

Veja também Macro DEFINE_ERROR_MESSAGE_TABLE

Macro DECLARE_KEYWORD_TABLE reader.h

O macro DECLARE_KEYWORD_TABLE, quando utilizado dentro da definição de uma classe derivada de tReader, declara que a classe possui uma tabela de palavras reservadas. Cada objeto leitor de OSW (vimos a definição de *leitor* no Capítulo 7) possui sua própria tabela de palavras reservadas. O macro toma como parâmetro o nome da classe.

Veja também Macro DEFINE_KEYWORD_TABLE

Macro DEFINE_COMMAND_TABLEx cmdhandl.h

O macro DEFINE_COMMAND_TABLEx define a tabela de comandos de uma classe derivada da classe tView (as entradas da tabela de comandos são definidas com os macros COMMAND e COMMAND_AND_ID). x pode ser 1, 2 ou 3. O macro toma x+1 parâmetros: o nome da classe de vista e x nomes de classes bases da classe de vista. As classes bases devem ser derivadas da classe tView, se a classe não derivar diretamente de tView.

Veja também Macro END_COMMAND_TABLE

Exemplo

No exemplo a seguir declaramos duas classes derivadas de tView, tBase1 e tBase2, cada uma com sua própria tabela de comandos. A seguir, declaramos a classe tView12, derivada de tBase1 e tBase2 (note a herança virtual e a herança múltipla).

```
// arquivo view12.h
#include "view.h"

class tBase1:  virtual public tView
{
  public:
     ...
     void CmCommandA();
     void CmCommandB();
     ...
     DECLARE_COMMAND_TABLE(tBase1);
}; // tBase1

class tBase2:  virtual public tView
{
  public:
     ...
     void CmCommandC();
     void CmCommandD();
     ...
     DECLARE_COMMAND_TABLE(tBase2);
}; // tBase2

class tView12:  public tBase1, public tBase2
{
  public:
     ...
     void CmCommandE();
     void CmCommandF();
     ...
     DECLARE_COMMAND_TABLE(tView12);
}; // tView12
// arquivo view12.cpp
#include "view12.h"

DEFINE_COMMAND_TABLE1(tBase1, tView)
  COMMAND(100, "CommandA", CmCommandA),
  COMMAND(101, "CommandB", CmCommandB),
END_COMMAND_TABLE;

DEFINE_COMMAND_TABLE1(tBase2, tView)
  COMMAND(200, "CommandC", CmCommandC),
  COMMAND(201, "CommandD", CmCommandD),
END_COMMAND_TABLE;

DEFINE_COMMAND_TABLE2(tView12, tBase1, tBase2)
  COMMAND(300, "CommandE", CmCommandE),
  COMMAND(301, "CommandF", CmCommandF),
END_COMMAND_TABLE;
```

A definição da tabela de comandos da classe de vista `tView12` nos diz que os objetos da classe podem responder aos comandos definidos na tabela da classe (`CommandE` e `CommandF`) *e* aos comandos definidos nas classes bases (ambas derivadas de `tView`) `tBase1` e `tBase2`.

Macro DEFINE_ERROR_MESSAGE_TABLE — errhandl.h

O macro `DEFINE_ERROR_MESSAGE_TABLE` define a tabela de mensagens de erros de uma classe derivada da classe `tErrorHandler`. O macro toma como parâmetros o nome da classe e o nome de uma classe base, a qual deve ser `tErrorHandler` ou uma classe derivada direta ou indiretamente de `tErrorHandler`. As entradas da tabela são definidas com o macro `ERROR_MESSAGE`.

Veja também Macro `DECLARE_ERROR_MESSAGE_TABLE`

Macro DEFINE_KEYWORD_TABLE — reader.h

O macro `DEFINE_KEYWORD_TABLE` define a tabela de palavras reservadas de uma classe derivada da classe `tReader`. O macro toma como parâmetros o nome da classe e o nome de uma classe base, a qual deve ser `tReader` ou uma classe derivada direta ou indiretamente de `tReader`. As entradas da tabela são definidas com o macro `KEYWORD`.

Veja também Macro `DECLARE_KEYWORD_TABLE`

Macro END_COMMAND_TABLE — cmdhandl.h

O macro `END_COMMAND_TABLE` é utilizado ao final da definição da tabela de comandos de uma classe de vista (veja o exemplo do macro `COMMAND`).

Veja também Macro `DEFINE_COMMAND_TABLE`

Macro END_ERROR_MESSAGE_TABLE — cmdhandl.h

O macro `END_ERROR_MESSAGE_TABLE` é utilizado ao final da definição da tabela de mensagens de erro de uma classe de manipulação de erros (veja o exemplo do macro `ERROR_MESSAGE`).

Veja também Macro `DEFINE_ERROR_MESSAGE_TABLE`

Macro END_KEYWORD_TABLE — reader.h

O macro `END_KEYWORD_TABLE` é utilizado ao final da definição da tabela de palavras reservadas de uma classe de leitor (veja o exemplo do macro `ERROR_MESSAGE`).

Veja também Macro `DEFINE_KEYWORD_TABLE`

Macro ERROR_MESSAGE — errhandl.h

O macro `ERROR_MESSAGE` é utilizado na definição de uma mensagem de erro de uma tabela de mensagens de erros de uma classe de manipulação de erros. O macro toma como parâmetros um inteiro identificador do código de erro associado à mensagem e uma cadeia de caracteres que contém a mensagem.

Exemplo

No exemplo a seguir ilustramos a utilização dos macros de declaração e definição das tabelas de mensagens de erro e de palavras reservadas. A classe `tMyReader` é derivada de `tReader`.

```
// arquivo myread.h
#include "reader.h"

class tMyReader:  public tReader
{
   public:
      ...
      enum
      {
         CODE1 = tReader::LastErrorCode,
         CODE2,
         LastErrorCode
      }; //
      enum
      {
         KEYW1 = tReader::LastKeyword,
         KEYW2,
         LastErrorCode
      }; //
      ...
      DECLARE_ERROR_MESSAGE_TABLE(tMyReader);
      DECLARE_KEYWORD_TABLE(tMyReader);
      ...
}; // tMyReader

// arquivo myread.cpp
#include "myread.h"

DEFINE_ERROR_MESSAGE_TABLE(tMyReader, tReader)
   ERROR_MESSAGE(CODE1, "Error 1"),
   ERROR_MESSAGE(CODE2, "Error 2"),
END_ERROR_MESSAGE_TABLE;

DEFINE_KEYWORD_TABLE(tMyReader, tReader)
   KEYWORD(KEYW1, "keyw1"),
   KEYWORD(KEYW2, "keyw2"),
END_KEYWORD_TABLE;
```

Macro KEYWORD **reader.h**

O macro `KEYWORD` é utilizado na definição de uma palavra reservada de uma tabela de palavras reservadas de uma classe de leitor. O macro toma como parâmetros um inteiro identificador do *token* associado à palavra reservada e uma cadeia de caracteres que contém a palavra reservada. A utilização do macro é ilustrada no exemplo do macro `ERROR_MESSAGE`.

10.3 Classes de OSW

Nesta Seção são apresentamos, em ordem alfabética, as classes de OSW. Para cada classe, são mostradas as definições de tipo, as constantes, os atributos e os métodos públicos e protegidos da classe. São destacadas, também, a sobrecarga dos operadores públicos e protegidos e os operadores e funções "amigas" da classe.

Classe t2DCell — 2dcell.h

A classe `t2DCell` é uma classe abstrata que representa a estrutura e o comportamento genéricos de uma célula de dimensão topológica 2 de um modelo de decomposição por células. A classe é derivada da classe abstrata `tFace` e derivada virtualmente da classe abstrata `tCell`.

Veja também Classes `tCell` e `tFace`

Construtor Público

Construtor
```
inline t2DCell(int n);
```
Inicializa a classe base `tCell`. O parâmetro `n` é o número de nós da célula.

Veja também Classe `tCell`

Métodos Públicos

Area
```
virtual double Area() const;
```
Retorna a área da superfície de **this**.

ComputeLocalSystem
```
virtual void ComputeLocalSystem(tLocalSystem& ls) const;
```
Determina a origem e os eixos Cartesianos do sistema local de **this**, armazenando o resultado em `ls`.

Veja também Classe `tLocalSystem`

GetDimension
```
int GetDimension() const;
```
Retorna 2, a dimensão topológica da célula. O método é declarado como virtual puro na classe base `tCell`.

GetEdge
```
tEdge* GetEdge(int i) const;
```
Retorna um ponteiro para a `i`-ésima aresta da célula. O método é declarado como virtual puro na classe base `tCell`.

Veja também Classe `tEdge`

GetFace
```
tFace* GetFace(int) const;
```
Retorna **this**. O método é declarado como virtual puro na classe base `tCell`.

GetMaterial
`tMaterial GetMaterial() const;`
Retorna o atributo `Material`, herdado da classe base `tCell`, o material da superfície da célula. O método é declarado como virtual puro na classe base `tFace`. O material de uma face é suposto constante em toda sua superfície.

Veja também Classe `tMaterial`

GetNumber
`int GetNumber() const;`
Retorna o atributo `Number`, herdado de `tCell`, correspondente ao identificador da célula no modelo de decomposição por células. O método é declarado como virtual puro na classe base `tFace`.

Veja também Classe `tFace`

GetNumberOfEdges
`int GetNumberOfEdges() const;`
Retorna o número de arestas da célula. O número de arestas de uma célula de dimensão topológica 2 é igual ao número de nós que definem a célula. O método é declarado como virtual puro na classe base `tCell`.

GetNumberOfFaces
`int GetNumberOfFaces() const;`
Retorna 1, o número de faces da célula. O método é declarado como virtual puro na classe `tCell`.

NormalAt
`t3DVector NormalAt(const t3DVector& p) const;`
Retorna o vetor normal à superfície da célula no ponto p. O método é declarado como virtual puro na classe base `tFace`.

Classe t2DExtents — faceboun.h

Um objeto da classe `t2DExtents` representa o retângulo envolvente de um objeto de dimensão topológica 2. As coordenadas dos pontos que definem o retângulo envolvente podem ser tomadas em relação a qualquer sistema de coordenadas Cartesianas.

Atributos Públicos

P1
`t3DVector P1;`
Coordenadas do canto inferior esquerdo de **this**.

P2
`t3DVector P2;`
Coordenadas do canto superior direito de **this**.

Construtores Públicos

Construtores
`inline t2DExtents();`
Construtor *default*. Inicializa **this** com um retângulo vazio.

inline t2DExtentes(**const** t2DExtents& e);
Construtor de cópia. Inicializa P1 e P2 com uma cópia de e.P1 e e.P2, respectivamente.

inline t2DExtents(**const** t3DVector& p1, **const** t3DVector& p2);
Inicializa P1 com p1 e P2 com p2.

inline t2DExtents(**double** x1, **double** y1, **double** x2, **double** y2);
Inicializa P1 com (x1,y1,0) e P2 com (x2,y2,0).

Métodos Públicos

Height
inline double Height() **const**;
Retorna a altura de **this**.

Width
inline double Width() **const**;
Retorna a largura de **this**.

Operador Público

operator <<
t2DExtents& **operator** <<(**const** t3DVector& p);
Ajusta as coordenadas de P1 e P2, se necessário, de modo que **this** envolva o ponto p. Retorna uma referência para **this**.

Classe t2DMeshGenerator — 2dmeshge.h

A classe abstrata t2DMeshGenerator encapsula a estrutura e a funcionalidade comuns de um filtro de geração de malhas de elementos de dimensão topológica 2 (definimos *filtros* no Capítulo 7). O filtro toma como entrada um objeto de uma classe derivada da classe abstrata tMeshableModel (as classes de modelos de cascas e de sólidos são derivadas de tMeshableModel) e produz como saída um objeto de uma classe derivada da classe tMesh.

Veja também Classes tMeshableModel e tMesh

Construtor Público

Construtor
inline t2DMeshGenerator();
Construtor *default*. Inicializa os atributos Input, Output e ElementSize do filtro.

Métodos Públicos

Execute
void Execute();
Executa o filtro. Transforma o modelo de entrada no modelo de saída. Se ocorrer algum erro durante o processo de geração da malha, uma exceção da classe xmsg é gerada.

Veja também Classe xmsg (C++)

GetElementSize
```
inline double GetElementSize() const;
```
Retorna o tamanho do elemento do modelo de saída de **this**.

GetInput
```
inline tMeshableModel* GetInput();
```
Retorna um ponteiro para o modelo de entrada de **this**.

GetOutput
```
inline tMesh* GetOutput();
```
Retorna um ponteiro para o modelo de saída de **this**.

SetElementSize
```
void SetElementSize(double esize);
```
Ajusta o tamanho do elemento do modelo de saída de **this** com o parâmetro `esize`.

SetInput
```
inline void SetInput(tMeshableModel* input);
```
Ajusta o modelo de entrada de **this** com o parâmetro `input`.

Atributos Protegidos

ElementSize
```
double ElementSize;
```
Tamanho do elemento do modelo de saída de **this**.

Input
```
tMeshableModel* Input;
```
Modelo de entrada de **this**.

Veja também Classe `tMeshableModel`

Output
```
tMesh* Output;
```
Modelo de saída de **this**.

Veja também Classe `tMesh`

Métodos Protegidos

CanMakeNodeAt
```
virtual bool CanMakeNodeAt(tBoundaryFace& bf, double x, double y);
```
Retorna **true** se um nó com coordenadas locais `x`,`y` puder ser construído durante a discretização da face de contorno `bf`; retorna **false** caso contrário. Um nó poderá ser construído se satisfizer algum critério de distância mínima em relação aos outros nós da face de contorno. O método pode ser sobrecarregado em classes derivadas de `t2DMeshGenerator` que implementam critérios específicos.

GenerateCells
```
virtual void GenerateCells(tBoundaryFace& bf);
```
Gera as células do modelo de saída de **this** sobre a face de contorno `bf`, a partir do fronte inicial e dos nós internos da face de contorno, como descrito no Capítulo 6.

Veja também `MakeInitialFront()`, `GenerateInternalNodes()`

GenerateInternalNodes
`virtual void GenerateInternalNodes(tBoundaryFace& bf);`
Gera os nós internos da face de contorno `bf`. Os nós gerados pelo método serão utilizados pelo método `GenerateCells()` para geração das células sobre a face de contorno `bf`. Executa o método virtual `CanMakeNodeAt()`.

Veja também `CanMakeNodeAt()`, `GenerateCells()`

MakeCell
`virtual t2DCell* MakeCell() = 0;`
Método virtual puro de construção dos elementos de dimensão topológica 2 do modelo de saída do filtro. O método retorna um ponteiro para o objeto contruído e deve ser sobrecarregado em classes concretas derivadas de `t2DMeshGenerator`.

Veja também Classe `t2DCell`

MakeInitialFront
`virtual void MakeInitialFront(tBoundaryFace& bf) = 0;`
Método virtual puro de construção do fronte inicial da face de contorno `bf`. O método deve ser sobrecarregado em classes concretas derivadas de `t2DMeshGenerator`.

Veja também Classe `tBoundaryFace`

MakeMesh
`virtual tMesh* MakeMesh();`
Método virtual de construção do modelo de saída do filtro. Em `t2DMeshGenerator` o método constrói um objeto da classe `tMesh`. Classes derivadas de `t2DMeshGenerator` podem sobrecarregar o método para construir objetos de classes derivadas de `tMesh`. O método retorna um ponteiro para o objeto construído.

Veja também Classe `tMesh`

MakeNode
`virtual tNode* MakeNode() = 0;`
Método virtual puro de construção dos nós dos elementos do modelo de saída do filtro. O método retorna um ponteiro para o objeto contruído e deve ser sobrecarregado em classes concretas derivadas de `t2DMeshGenerator`.

Veja também Classe `tNode`

SmoothMesh
`virtual void SmoothMesh();`
Suaviza a malha, de acordo com o método descrito no Capítulo 6. Pode ser sobrecarregado em classes derivadas de `t2DMeshGenerator`.

SplitFace
`virtual void SpliteFace(tBoundaryFace& bf);`
Gera as células do modelo de saída para a face de contorno `bf`. O método executa os métodos virtuais `GenerateInternalNodes()` e `GenerateCells()`.

Veja também `GenerateInternalNodes()`, `GenerateCells()`

Exemplo

A função a seguir ilustra a utilização de um filtro de geração de malhas da classe `tFEShellMeshGenerator`, diretamente derivada de `t2DMeshGenerator`. O filtro gera uma malha de elementos finitos triangulares de casca a partir de um modelo geométrico de cascas. O elemento finito de casca foi definido no Capítulo 6 e o modelo geométrico de cascas foi definido no Capítulo 3.

```
tMesh* GenMesh(tShell* shell, double elementSize)
{
  tFEShellMeshGenerator mg;

  mg.SetInput(shell);
  mg.SetElementSize(elementSize);
  mg.Execute();
  return mg.GetOutput();
}
```

Veja também Classes `tFEShellMeshGenerator` e `tShell`

Classe t3DTransfMatrix **3dtransf.h**

Os objetos da classe `t3DTransfMatrix` representam matrizes de transformações homogêneas 3D, como visto no Capítulo 7. Os métodos da classe `t3DTransfMatrix` implementam as operações de composição de transformações e de transformação de pontos no espaço 3D.

Construtores Públicos

`inline t3DTransfMatrix();`
Construtor *default*.

`t3DTransfMatrix(const t3DTransfMatrix& mat);`
Construtor de cópia. Inicializa os elementos de **this** com os elementos correpondentes de `mat`.

Métodos Públicos

ComposeInPlaceWith
`t3DTransfMatrix& ComposeInPlaceWith(const t3DTransfMatrix& mat);`
Compõe a transformação definida em **this** com a transformação definida em `mat` e armazena o resultado em **this**. Retorna uma referência para **this**.

ComposeWith
`t3DTransfMatrix ComposeWith(const t3DTransfMatrix& mat) const;`
Retorna a matriz de transformação resultante da composição de **this** com `mat`.

Inverse
`t3DTransfMatrix Inverse() const;`
Retorna a matriz de transformação inversa de **this**. Se **this** for singular, uma exceção da classe `tXMath` é gerada.

Veja também Classe `tXMath`

InverseInPlace
```
t3DTransfMatrix& InverseInPlace();
```
Inverte **this** e armazena o resultado em **this**; retorna uma referência para **this**. Se **this** for singular, uma exceção da classe `tXMath` é gerada.

Veja também Classe `tXMath`

SetColumn
```
void SetColumn(int c, double a, double b, double c, double d);
```
Ajusta os elementos da coluna `c` com os valores `a`, `b`, `c` e `d`.

SetIdentity
```
void SetIdentity();
```
Torna **this** a matriz identidade.

SetRotation
```
void SetRotation(const t3DVector& org, const t3DVector& dir,
  double angle);
```
Torna **this** a matriz de rotação de `angle` radianos em torno do eixo definido pela direção `dir` e pela origem `org`.

SetRotationX
```
void SetRotationX(double angle);
```
Torna **this** a matriz de rotação de `angle` radianos em torno do eixo X.

SetRotationY
```
void SetRotationY(double angle);
```
Torna **this** a matriz de rotação de `angle` radianos em torno do eixo Y.

SetRotationZ
```
void SetRotationZ(double angle);
```
Torna **this** a matriz de rotação de `angle` radianos em torno do eixo Z.

SetRow
```
void SetRow(int r, double a, double b, double c, double d);
```
Ajusta os elementos da linha `r` com os valores `a`, `b`, `c` e `d`.

SetScale
```
void SetScale(const t3DVector& base, const t3DVector& s);
```
Torna **this** a matriz de transformação de escala definida pelas coordenadas de `s`, com base no ponto `base`.

SetTranslation
```
void SetTranslation(const t3DVector& t);
```
Torna **this** a matriz de translação definida pelas coordenadas de `t`.

Transform
```
t3DVector Transform(const t3DVector& v) const;
```
Retorna o ponto resultante da aplicação da transformação definida por **this** no ponto `v`, sem alterar `v`.

Veja também Classe `t3DVector`

TransformInPlace
`t3DVector& TransformInPlace(t3DVector& v) const;`
Aplica a transformação definida por **this** no próprio ponto `v`. Retorna uma referência para `v`.

Veja também Classe `t3DVector`

Operadores Públicos

operator =
`t3DTransfMatrix& operator =(const t3DTransfMatrix& mat);`
Operador de cópia. Ajusta os elementos de **this** com os valores dos elementos de `mat`.

operator *
`t3DTransfMatrix operator *(const t3DTransfMatrix& mat) const;`
Retorna a matriz resultante da composição de **this** com `mat`. Sinônimo do método `ComposeWith`.

Veja também `t3DTransfMatrix::ComposeWith`

`t3DVector operator *(const t3DVector& v) const;`
Retorna o ponto resultante da aplicação da transformação definida por **this** no ponto `v`, sem alterar `v`. Sinônimo do método `Transform`.

Veja também `t3DTransfMatrix::Transform`

Classe t3DVector — 3dvector.h

Os objetos da classe `t3DVector` representam vetores no espaço 3D, com coordenadas (x, y, z) tomadas em relação a algum sistema global ou local de coordenadas cartesianas.

Construtores Públicos

`inline t3DVector();`
Construtor *default*.

`inline t3DVector(const t3DVector& v);`
Construtor de cópia. Inicializa as coordenadas do vetor com os valores das coordenadas do vetor `v`.

`inline t3DVector(double x, double y, double z);`
Inicializa as coordenadas do vetor com os valores dos parâmetros `x`, `y` e `z`.

Atributos Públicos

x
`double x;`
Coordenada x de **this**.

y
`double y;`
Coordenada y de **this**.

z
`double z;`
Coordenada z de **this**.

Métodos Públicos

Cross
`t3DVector Cross(const t3DVector& v) const;`
Retorna o vetor resultante do produto vetorial de **this** com o vetor v.

Inner
`inline double Inner(const t3DVector& v) const;`
Retorna o escalar resultante do produto interno de **this** com o vetor `v`.

IsNull
`inline bool IsNull() const;`
Retorna **true** se **this** é o vetor nulo, ou seja, se as coordenadas de **this** são todas iguais a zero; retorna **false** caso contrário.

Length
`double Length() const;`
Retorna o módulo de **this**.

Normalize
`t3DVector& Normalize();`
Normaliza **this**, ou seja, armazena em **this** o produto escalar de **this** com o inverso do módulo de this. Se **this** é o vetor nulo, uma exceção do tipo `tXMath` é gerada.

Veja também `t3DVector::Versor`, classe `tXMath`

SetCoordinates
`inline void SetCoordinates(const t3DVector& v);`
Ajusta as coordenadas de **this** com os valores das coordenadas do vetor `v`.

`inline void SetCoordinates(double x, double y, double z);`
Ajusta as coordenadas de **this** com os valores dos parâmetros `x`, `y` e `z`.

Versor
`t3DVector Versor() const;`
Retorna o vetor unitário resultante do produto escalar de **this** com o inverso do módulo de **this**. Se **this** é o vetor nulo, uma exceção do tipo `tXMath` é gerada.

Veja também `t3DVector::Normalize`, classe `tXMath`

Operadores Públicos

operator =
`inline t3DVector& operator =(const t3DVector&);`
Copia as coordenadas do vetor `v` em **this**. Retorna uma referência para **this**.

operator +
`inline t3DVector operator +(const t3DVector& v) const;`
Retorna o vetor resultante da soma de **this** com o vetor `v`.

operator –
```
inline t3DVector operator -(const t3DVector& v) const;
```
Retorna o vetor resultante da subtração de **this** com o vetor `v`.

operator *
```
inline t3DVector operator *(double s) const;
```
Retorna o vetor resultante do produto de **this** com o escalar `s`.

```
inline double operator *(const t3DVector& v) const;
```
Retorna o escalar resultante do produto interno de **this** com o vetor `v`.

Veja também `t3DVector::Inner`

operator +=
```
inline t3DVector& operator +=(const t3DVector& v);
```
Soma o vetor `v` com **this** e armazena o vetor resultante em **this**. Retorna uma referência para **this**.

operator –=
```
inline t3DVector& operator -=(const t3DVector& v);
```
Subtrai o vetor `v` de **this** e armazena o vetor resultante em **this**. Retorna uma referência para **this**.

operator *=
```
inline t3DVector& operator *=(double s);
```
Multiplica **this** com o escalar `s` e armazena o vetor resultante em **this**. Retorna uma referência para **this**.

operator ==
```
inline bool operator ==(const t3DVector& v) const;
```
Retorna **true** se as coordenadas de **this** e do vetor `v` são iguais; retorna **false** caso contrário.

Veja também `::IsEqual`

operator !=
```
inline bool operator !=(constt3DVector& v) const;
```
Retorna **false** se as coordenadas de **this** e do vetor `v` são iguais; retorna **true** caso contrário.

Classe t3N2DCell — 2dcell.h

Objetos da classe `t3N2DCell` representam células de dimensão topológica 2 com três nós. A classe `t3N2DShell`, derivada de `t2DCell`, é a classe base do elemento de casca triangular de OSW.

Veja também Classe `t2DCell`

Construtores Públicos

Construtores
```
t3N2DCell();
```
Construtor *default*. Inicializa os objetos bases `tCell` e `t2DCell`.

`t3N2DCell(tNode* n1, tNode* n2, tNode* n2);`
Inicializa as classes bases `tCell` e `t2DCell` e ajusta os nós de **this** com os parâmetros `n1`, `n2` e `n3`.

Veja também Classe `tNode`

Métodos Públicos

Area
`double Area() const;`
Retorna a área da superfície de **this**. O método, declarado como virtual na clase base `t2DCell`, é sobrecarregado porque a área de um triângulo pode ser *diretamente* determinada a partir das coordenadas de seus nós.

Contour
`void Contour(double s, tGraphicModel& gm);`
Determina a isolinha de **this** correpondente ao escalar `s`, tal como visto no Capítulo 7, e adiciona a isolinha no modelo gráfico `gm`. O método é declarado como virtual puro na classe base `tCell`.

Veja também Classe `tGraphicModel`

Métodos Protegidos

ComputeShapeDerivativesAt
`tMatrix ComputeShapeDerivativesAt(double x[]) const;`
Retorna a matrix das derivadas das funções de forma de **this** no ponto de coordenadas adimensionais `x[0]`, `x[1]`.

Veja também Classe `tMatrix`

ComputeShapeFunctionsAt
`tVector ComputeShapeFunctionsAt(double x[]) const;`
Retorna o vetor das funções de forma de **this** no ponto de coordenadas adimensionais `x[0]`, `x[1]`. As funções de forma da célula triangular linear são dadas pelas Equações (3.20).

Veja também Classe `tVector`

Classe t3NShell — 3nshell.h

Um objeto da classe `t3NShell` representa um elemento finito triangular de casca, tal como definido no Capítulo 6. A classe, derivada da classe `t3N2DCell` e da classe `tFiniteElement`, implementa métodos específicos de cálculo da matriz de rigidez e do vetor de esforços equivalentes de um elemento finito de casca.

Veja também Classes `t3N2DCell`, `tFiniteElement`

Construtores Públicos

Construtores
`inline t3NShell();`
Construtor *default*. Inicializa os objetos base `t3N2DCell` e `tFiniteElement`.

inline t3NShell(tNode* n1, tNode* n2, tNode* n3);
Inicializa os objetos base t3N2DCell e tFiniteElement com os parâmetros n1, n2 e n3.

Atributo Público

Tickness
double Tickness;
Espessura de **this**.

Método Público

ComputeDisplacementVector
tVector ComputeDisplacementVector();
Calcula o vetor de deslocamentos nodais de **this**.

Veja também Classe tVector

Métodos Protegidos

ComputeLoadVector
tVector ComputeLoadVector();
Calcula o vetor de esforços nodais equivalentes de **this**.

Veja também Classes tVector, tFiniteElement

ComputeStiffnessMatrix
tMatrix ComputeStiffnessMatrix();
Calcula a matriz de rigidez de **this**, tal como visto no Capítulo 6.

Veja também Classes tMatrix, tFiniteElement

Classe t4N2DCell **2dcell.h**

Objetos da classe t4N2DCell representam células de dimensão topológica 2 com quatro nós. A classe t4N2DShell, derivada de t2DCell, é a classe base do elemento de contorno quadrilateral com aproximação linear.

Veja também Classe t2DCell

Construtores Públicos

Construtores
t4N2DShell();
Construtor *default*. Inicializa as classes bases tCell e t2DCell.

t4N2DCell(tNode* n1, tNode* n2, tNode* n2, tNode* n3);
Inicializa as classes bases tCell e t2DCell e ajusta os nós de **this** com os parâmetros n1, n2, n3 e n4.

Veja também Classe tNode

Métodos Públicos

Contour
```
void Contour(double s, tGraphicModel& gm);
```
Determina a isolinha de **this** correspondente ao escalar `s`, tal como visto no Capítulo 7, e adiciona a isolinha no modelo gráfico `gm`. O método é declarado como virtual puro na classe base `tCell`.

Veja também Classe `tGraphicModel`

Métodos Protegidos

ComputeShapeDerivativesAt
```
tMatrix ComputeShapeDerivativesAt(double x[]) const;
```
Retorna a matrix das derivadas das funções de forma de **this** no ponto de coordenadas adimensionais `x[0]`, `x[1]`.

Veja também Classe `tMatrix`

ComputeShapeFunctionsAt
```
tVector ComputeShapeFunctionsAt(double x[]) const;
```
Retorna a vetor das funções de forma de **this** no ponto de coordenadas adimensionais `x[0]`, `x[1]`. As funções de forma da célula quadrilateral linear são dadas pelas Equações (3.18).

Veja também Classe `tVector`

Classe tActor — actor.h

Um objeto da classe `tActor` representa um objeto da imagem de uma cena. `tActor` oferece métodos para posicionamento, escala e orientação do ator na cena. Um ator contém, ainda, uma referência para um objeto da classe `tMapper`, responsável pela aparência do ator na imagem da cena.

Veja também Classe `tMapper`, classe `tScene`

Construtor e Destrutor Públicos

Construtor
```
tActor(tModel* model);
```
Constrói o ator para o modelo `model`. O modelo, usualmente, pertence à coleção de modelos do documento da cena da qual **this** faz parte. O método automaticamente cria um mapeador para o ator. O mapeador criado usa uma tabela de cores *default* para determinar as cores dos vértices do modelo. O mapeador do ator pode ser modificado pelo método `SetMapper`. Se quisermos visualizar um modelo em um cena, devemos criar um ator para ele.

Veja também Classe `tModel`, `tActor::SetMapper`

Destrutor
```
~tActor();
```
Destrói o mapeador de **this**.

Métodos Públicos

GetMapper
```
inline tMapper* GetMapper();
```
Retorna um ponteiro para o mapeador de **this**.

GetPickable
```
inline bool GetPickable() const;
```
Retorna o valor da *flag* de seleção do ator.

Veja também `SetPickable`

GetVisibility
```
inline bool GetVisibility() const;
```
Retorna o valor da *flag* de visibilidade do ator.

Veja também `SetVisibility`

SetMapper
```
void SetMapper(tMapper* mapper);
```
Se `mapper` for diferente de zero, destrói o mapeador de **this** e torna `*mapper` o novo mapeador de **this**.

SetOrientation
```
void SetOrientation(const t3DVector& o);
```
Ajusta a orientação de **this**. Os componentes `o.x`, `o.y` e `o.z` definem os ângulos de rotação do ator em torno dos eixos Cartesianos correspondentes.

SetOrigin
```
void SetOrigin(const t3DVector& o);
```
Ajusta a origem de **this** para o ponto de coordenadas `o`. A origem de um ator é o ponto em torno do qual são efetuadas as rotações que definem a orientação do ator.

SetPickable
```
void SetPickable(bool pickable);
```
Ajusta a *flag* de seleção do ator. Se `pickable` for **true**, o ator pode ser selecionado durante a execução de um comando de uma vista da cena.

Veja também `GetPickable`

SetPosition
```
void SetPosition(const t3DVector& p);
```
Ajusta a posição de **this** para o ponto de coordenadas `p`.

SetScale
```
void SetScale(const t3DVector& s);
```
Ajusta os fatores de escala `s.x`, `s.y` e `s.z` de **this**.

SetVisibility
```
void SetVisibility(bool visibility);
```
Ajusta a *flag* de visibilidade do ator. Se `visibility` for **true**, o ator é exibido na cena; caso contrário, nenhuma imagem do ator será gerada.

Veja também `GetVisibility`

Classe tApplication — oswapp.h

Um objeto da classe `tApplication` representa uma aplicação genérica de modelagem OSW. A interface da classe define métodos virtuais para inicialização da janela principal e gerenciamento da base de dados da aplicação. As classes derivadas de `tApplication` devem ser responsáveis pela construção do objeto de documento manipulado pela aplicação. A classe é derivada da classe OWL `TApplication`.

Veja também Classe `tDocument`, classe `tMainWindow`

Construtor e Destrutor Públicos

Construtor
```
tApplication(const CHAR* name = 0, int argc = 0, char** argv = 0);
```
Inicializa a classe base com o parâmetro `name` e os atributos protegidos `CmdCount` e `CmdName` com `argc` e `argv`, respectivamente.

Destrutor
```
virtual ~tApplication();
```
Destrutor virtual. Sem funcionalidade específica em `tApplication`.

Métodos Públicos

CmFileClose
```
virtual void CmFileClose();
```
Se existir o documento de aplicação, verifica se o documento está aberto. Envia a mensagem `CanClose()` ao documento. Se a mensagem retornar **true**, envia a mensagem `Close()`.

Veja também Classe `tDocument`

CmFileNew
```
virtual void CmFileNew();
```
Executa o método `CreateDoc()`, passando como argumento a constante `dfNewDoc`.

Veja também `CreateDoc`

CmFileOpen
```
virtual void CmFileOpen();
```
Executa o método `CreateDoc()`, passando como argumento a constante `dfOpenDoc`.

CmFileSave
```
virtual void CmFileSave();
```
Se exisitir o documento da aplicação, verifica se o documento foi alterado. Envia a mensagem `Commit()` ao documento.

CmFileSaveAs
```
virtual void CmFileSaveAs();
```
Se existir o documento da aplicação, abre uma caixa de diálogo solicitando um nome de arquivo. Envia a mensagem `SetDocPath()` ao documento e, em seguida, a mensagem `Commit()`.

GetDocument
```
inline tDocument* GetDocument();
```
Retorna um ponteiro para o documento manipulado pela aplicação.

InitApplication
```
virtual void InitApplication();
```
Inicializa a aplicação. Executa o método de inicialização da janela principal da aplicação, `InitMainWindow`.

InitMainWindow
```
virtual void InitMainWindow();
```
Inicializa os elementos de interface da aplicação: a janela principal, a barra de *status* e a janela de comandos.

Run
```
virtual int Run();
```
Executa a aplicação.

Atributos Protegidos

CmdCount
```
int CmdCount;
```
Número dos argumentos na linha de comandos passados a **this**.

CmdName
```
char** CmdName;
```
Ponteiros para as cadeias de caracteres dos argumentos na linha de comandos passados a **this**.

Doc
```
tDocument* Doc;
```
Ponteiro para o documento manipulado pela aplicação.

DocPathData
```
tDocPathData DocPathData;
```
Informações sobre o tipo de arquivo de documento manipulado pela aplicação.

Veja também Classe `tDocPathData`

Métodos Protegidos

ConstructDoc
```
virtual tDocument* ConstructDoc(tApplication& app);
```
Construtor "virtual" do documento da aplicação. Retorna zero.

CreateDoc
```
virtual tDocument* CreateDoc(long flags);
```
Se existir o documento da aplicação, fecha o documento e executa o método virtual `ConstructDoc()`. Se `flags` for igual a constante `dfOpenDoc`, abre uma caixa de diálogo solicitando o nome do arquivo. Executa o método de inicialização do documento, `InitDoc`.

InitDoc
```
virtual tDocument* InitDoc(tDocument* doc,
  const char* path, long flags);
```
Inicializa o documento `doc`. Se `flags` for igual a `dfOpenDoc`, envia as mensagens `SetDocPath()` e `Open()` a `doc`. Por fim, envia a mensagem `ConstructScenes()` ao documento `doc`.

Classe tBandSystem — bandsystem.h

Um objeto da classe `tBandSysytem` é um sistema linear com matriz de coeficientes simétrica e armazenada em banda. A interface da classe sobrecarrega os métodos virtuais de solução do sistema e montagem da matriz de coeficientes e do vetor de termos independentes, herdados da classe base `tLinearSystem`.

Construtor Público

Construtor
```
tBandSystem(int m, int n);
```
Inicializa a classe base `tLinearSystem` com os parâmetros `m` e `n`.

Métodos Públicos

Assemble
```
void Assemble(const tMatrix& m, tLocationArray* l);
```
Adiciona a matriz `m` à matriz de coeficientes de **this**, de acordo com os números de equações contidas em `*l`.

```
void Assemble(const tVector& v, tLocationArray* l) = 0;
```
Adiciona o vetor `v` ao vetor de termos independentes de **this**, de acordo com os números de equações contidas em `*l`.

GetSemibandWidth
```
inline int GetSemibandWidth() const;
```
Retorna a largura da semibanda da matriz de coeficientes de **this**.

Solve
```
void Solve();
```
Calcula o vetor solução de **this**.

Classe tBE4NQuad — bequad.h

Um objeto da classe `tBE4NQuad` representa um elemento de contorno quadrangular plano, tal como definido no Capítulo 6. A classe deriva das classes `t4N2DCell` e `tBoundaryElement`.

Veja também Classes `t4N2DCell` , `tBoundaryElement`

Construtor Público

Construtor
```
tBE4NQuad();
```
Inicializa as classes base `t4N2DCell` e `tBoundaryElement`.

Métodos Públicos

ComputeDistanceFrom
`double ComputeDistanceFrom(tSourcePoint& sp) const;`
Retorna a menor distância entre o ponto fonte `sp` e os nós de **this**. O método é declarado como virtual puro na classe base `tBoundaryElement`.

ComputeSize
`double ComputeSize() const;`
Retorna o perímetro de **this**. O método é declarado como virtual puro na classe base `tBoundaryElement`.

NormalAt
`t3DVector NormalAt(double x[]) const;`
Retorna o vetor normal a **this**, no ponto de coordenadas adimensionais `x[0]`, `x[1]`. O método é declarado como virtual puro na classe base `tBoundaryElement`.

Veja também Classe `t3DVector`

Classe tBESolidMeshGenerator 2dmeshge.h

Um objeto da classe `tBESolidMeshGenerator` é um filtro de geração de malhas de elementos de contorno quadrangulares planos definidos no Capítulo 6. A entrada do filtro é um modelo da classe `tSolid`; a saída do filtro é um modelo da classe `tSolidMesh`. A classe deriva diretamente de `t2DMeshGenerator`.

Veja também Classes `t2DMeshGenerator`, `tShell`, `tShellMesh`

Métodos Públicos

GetInput
`inline tSolid* GetInput();`
Retorna um ponteiro para o modelo de entrada de **this**.

SetInput
`void SetInput(tSolid* solid);`
Torna `solid` o modelo de entrada de **this**.

Métodos Protegidos

MakeCell
`t2DCell* MakeCell();`
Constrói uma célula da classe `tBE4NQuad`, derivada de `t2DCell`, e retorna um ponteiro para a célula. O método é declarado como virtual puro na classe `t2DMeshGenerator`.

Veja também Classe `tBE4NQuad`

MakeInitialFront
`void MakeInitialFront(tBoundaryFace& bf);`
Constrói o fronte inicial da face de contorno `bf`. O método é declarado como virtual puro na classe base `t2DMeshGenerator`.

Veja também Classe `tBoundaryFace`

MakeMesh
```
tMesh* MakeMesh(const char* name);
```
Constrói um modelo de decomposição por células da classe `tSolidMesh` e retorna um ponteiro para o modelo.

MakeNode
```
tNode* MakeNode();
```
Constrói um nó da classe `t3FNode`, derivado de `tNode`, e retorna um ponteiro para o nó. O método é declarado como virtual puro na classe base `t2DMeshGenerator`.

Veja também Definição de tipo `t3FNode`

Classe tBESolver — besolver.h

Um objeto da classe `tBESolver` representa um analisador de estruturas elastostáticas pelo método dos elementos de contorno. A classe é derivada de `tSolver` e sua interface define métodos próprios de construção e montagem do sistema de equações lineares e de término do processo de análise.

Veja também Classe `tSolver`

Construtor Público

Construtor
```
tBESolver(tMesh& mesh);
```
Inicializa a classe base `tSolver` com `mesh`.

Métodos Protegidos

AssembleSystem
```
void AssembleSystem();
```
Montagem do sistema de equações lineares. Solicita a todos os elementos de contorno do modelo de entrada, para cada ponto fonte, a contribuição do elemento para o lados esquerdo e direito do sistema de equações de **this**, ou seja, as matrizes de influência **H** e **G** do elemento. Em seguida, solicita ao sistema linear a adição das matrizes de influência aos lados esquerdo e direito do sistema. Os pontos fonte de **this** são obtidos de um *iterator* de pontos fontes, através do método `GetSourcePointIterator()`.

Veja também `GetSourcePointIterator`, classe `tBoundaryElement`

ConstructLinearSystem
```
tLinearSystem* ConstructLinearSystem();
```
Constrói um sistema linear com matriz cheia, objeto da classe `tFullSystem`.

Veja também Classe `tFullSystem`

Método Protegido

GetSourcePointIterator
```
tSourcePointIterator GetSourcePointIterator();
```
Retorna um *iterator* de pontos fonte de **this**. Em `tBESolver`, os pontos fonte, dados por um *iterator* interno, são pontos externos cuja localização é determinada em função da posição dos nós e das normais dos elementos de contorno do modelo de entrada de

this. Classes derivadas de `tBESolver` podem utilizar seus próprios *iterators* internos de pontos fonte.

Veja também Classes `tSourcePoint` , `tInternalIterator`

Classe tBoundaryEdgesFilter bedges.h

Um objeto da classe `tBoundaryEdgesFilter` representa um filtro de extração das *arestas de contorno* de um modelo de decomposição por células. Uma aresta de contorno é uma aresta que pertence somente a uma célula do modelo de entrada. A saída do filtro é um modelo gráfico contendo as arestas de contorno do modelo de entrada.

Veja também Classe `tMesh`, classe `tGraphicModel`

Construtor Público

Construtor
`tBoundaryEdgesFilter();`
Construtor *default*.

Métodos Públicos

Execute
`tGraphicModel* Execute();`
Executa o filtro, se existir o modelo de entrada. Retorna um ponteiro para o modelo gráfico resultante do processo.

SetInput
`void SetInput(tMesh* mesh);`
Torna `mesh` o modelo de entrada de **this**.

Classe tBoundaryElement belement.h

A classe abstrata `tBoundaryElement` representa o comportamento de um elemento de contorno genérico de um modelo mecânico. A interface da classe define métodos virtuais para o cálculo das matrizes de influência de um elemento de contorno. `tBoundaryElement` deriva virtualmente da classe `tCell`.

Veja também Classe `tCell`

Construtor e Destrutor Públicos

Construtor
`tBoundaryElement(int n);`
Inicializa a classe base virtual `tCell` com o parâmetro `n`.

Destrutor
`~tBoundaryElement();`
Executa o método `Terminate()`.

Métodos Públicos

ComputeDistanceFrom
`virtual double ComputeDistanceFrom(tSourcePoint& sp) const = 0;`
Retorna um valor correspondente a menor distância entre o ponto fonte `sp` e **this**. Esse valor é utilizado para determinação do número de pontos de Gauss utilizados nas integrações numéricas do elemento. Classes derivadas de `tBoundaryElement` devem sobrecarregar o método.

ComputeElementContributions
`virtual void ComputeElementContributions(tSourcePoint& sp);`
Calcula as matrizes de influência de **this** para o ponto fonte `sp`.

Veja também `H`, `G`

ComputeSize
`virtual double ComputeSize() const = 0;`
Retorna um valor correspondente ao tamanho de **this**. Esse valor é utilizado para determinação do número de pontos de Gauss utilizados nas integrações numéricas do elemento. Classes derivadas de `tBoundaryElement` devem sobrecarregar o método.

GetGMatrix
`inline tMatrix GetGMatrix();`
Retorna a matriz **G** de **this**.

GetHMatrix
`inline tMatrix GetHMatrix();`
Retorna a matriz **H** de **this**.

NormalAt
`virtual t3DVector NormalAt(double x[]) const = 0;`
Retorna o vetor normal a **this**, no ponto de coordenadas adimensionais `x`. Classes derivadas de `tBoundaryElement` devem sobrecarregar o método.

Veja também Classe `t3DVector`

Terminate
`virtual void Terminate();`
Destrói as matrizes de influência de **this**.

Atributos Protegidos

G
`tMatrix G;`
Matriz **G** de **this**.

Veja também Classe `tMatrix`

H
`tMatrix H;`
Matriz **H** de **this**.

Veja também Classe `tMatrix`

p
```
static double p[3][3];
```
Matriz de soluções fundamentais $\mathbf{p}^*$ de **this**.

u
```
static double u[3][3];
```
Matriz de soluções fundamentais $\mathbf{u}^*$ de **this**.

Métodos Protegidos

ComputeFundamentalSolutions
```
virtual void ComputeFundamentalSolutions(t3DVector& s,
  t3DVector& x);
```
Calcula as soluções fundamentais de Kelvin no ponto `x`. `s` é a posição do ponto fonte. O método armazena os resultados do cálculo nas matrizes `u` e `p`, as quais representam, respectivamente, $\mathbf{u}^*$ e $\mathbf{p}^*$ do elemento. Como o método é virtual, outras soluções fundamentais podem ser implementadas em classes derivadas de `tBoundaryElement`.

Veja também `u`, `p`

ComputeOutsideIntegration
```
virtual void ComputeOutsideIntegration(tSourcePoint& sp,int rule);
```
Efetua as integrações numéricas necessárias ao cálculo das matrizes de influência do elemento. `sp` é o ponto fonte, considerado externo, e `rule` é o número de pontos de Gauss utilizados nas integrações.

Veja também Classe `tSourcePoint`

Classe tBoundaryFace — faceboun.h

Um objeto da classe `tBoundaryFace` representa uma face de contorno de um modelo de faces de contorno, tal como visto no Capítulo 6. Uma face de contorno mantém a lista de nós internos gerados sobre a face, a lista de arestas do fronte da face, o sistema de coordenadas locais da face e as coordenadas dos cantos do retângulo envolvente da face, tomadas em relação ao sistema de coordenadas locais da face. Objetos de classes derivadas da classe `t2DMeshGenerator` criam faces de contorno durante o processo de geração de malhas de dimensão topológica 2.

Veja também Classe `t2DMeshGenerator`

Métodos Públicos

ComputeExtents
```
void ComputeExtents();
```
Determina as coordenadas do retângulo envolvente de **this**.

Veja também Classe `t2DExtents`

FindOptimalNode
```
virtual tNode* FindOptimalNode(tNode* a, tNode* b);
```
Retorna um ponteiro para o nó de **this** que forma o melhor elemento tringular com os nós `a` e `b`. Durante a geração de malhas de dimensão topológica 2, um objeto de uma classe derivada de `t2DMeshGenerator` envia essa mensagem para a face de

contorno sobre a qual estão sendo geradas as células da malha. Classes derivadas de `tBoundaryFace` podem sobrecarregar o método para a implementação de critérios específicos de escolha do nó ótimo.

Veja também Classe `t2DMeshGenerator`

GetExtents
```
t2DExtents& GetExtents();
```
Retorna uma referência para o retângulo envolvente de **this**.

Veja também `ComputeExtents()`

KillEdge
```
void KillEdge(tEdge* f_edge);
```
Remove a aresta `f_edge` da lista de arestas do fronte de **this** e destrói `f_edge`.

Veja também Classe `tBoundaryFace::tEdge`

KillNode
```
void KillNode(tNode* f_node);
```
Remove o nó `f_node` da lista de nós de **this** e destrói `f_node`, se `f_node` não pertencer a nenhuma aresta do fronte de **this**.

Veja também Classe `tBoundaryFace::tNode`

MakeEdge
```
tEdge* MakeEdge(tNode* f_n1, tNode* f_n2);
```
Constrói uma aresta do fronte de **this**, do nó `f_n1` ao nó `f_n2` da face de contorno. Adiciona a aresta à lista de arestas de **this** e retorna um ponteiro para a aresta criada.

Veja também Classe `tBoundaryFace::tEdge`

MakeNode
```
tNode* MakeNode(::tNode* node);
```
Constrói um nó da face de contorno, o qual está associado ao nó `node` de um modelo de decomposição por células. Adiciona o nó à lista de nós de **this** e retorna um ponteiro para o nó criado.

Veja também Classe `tBoundaryFace::tNode`

PeekEdge
```
tEdge* PeekEdge(tNode* f_n1, tNode* f_n2);
```
Retorna um ponteiro para a aresta do fronte, se existir, do nó `f_n1` ao nó `f_n2`; retorna 0 caso contrário.

PeekFrontEdge
```
inline tEdge* PeekFrontEdge();
```
Retorna um ponteiro para a próxima aresta do fronte de **this**.

Atributos Públicos

LS
```
tLocalSystem LS;
```
Sistema Cartesiano de coordenadas locais de **this**.

Veja também Classe `tLocalSystem`

Classe tCamera — camera.h

Um objeto da classe `tCamera` é uma câmera virtual utilizada no processo de síntese de imagens tridimensionais. A classe define métodos para posicionar e orientar uma câmera como definido no Capítulo 7. Uma câmera é um dos objetos de uma cena, juntamente com atores e luzes. Um objeto da classe `tRenderer` usa uma câmera para capturar a porção da cena que fará parte da imagem sintetizada.

Veja também Classe `tRenderer`, classe `tScene`

Enumeração Pública

tProjectionType
`enum tProjectionType;`
Tipos de projeção de uma câmera: `Parallel` ou `Perspective`.

Construtores Públicos

Construtores
`tCamera();`
Construtor *default*. Constrói uma câmera posicionada em $(0, 0, 1)$, com ponto focal em $(0, 0, 0)$, vetor "para cima" $(0, 1, 0)$, ângulo de vista de 30^0, razão de aspecto igual a 1, tipo de projeção paralela.

`tCamera(const tCamera& camera);`
Construtor de cópia. Constrói **this** com uma cópia de `camera`.

Métodos Públicos

Azimuth
`void Azimuth(double angle);`
Rotaciona a posição da câmera do ângulo `angle` em torno do vetor "para cima" centrado no ponto focal.

Elevation
`void Elevation(double angle);`
Rotaciona a posição da câmera do ângulo `angle` em torno do vetor resultante do produto vetorial do VPN com o vetor "para cima" centrado no ponto focal.

GetAspectRatio
`double GetAspectRatio() const;`
Retorna a razão de aspecto de **this**. A razão de aspecto é a razão entre a largura e a altura da imagem captada pela câmera.

GetFocalPoint
`const t3DVector& GetFocalPoint() const;`
Retorna o ponto focal de **this**.

GetPosition
`const t3DVector& GetPosition() const;`
Retorna a posição de **this**.

GetProjection
```
tProjectionType GetProjectionType() const;
```
Retorna o tipo de projeção efetuada por **this**.

GetViewAngle
```
double GetViewAngle() const;
```
Retorna o ângulo de vista de **this**.

GetViewUp
```
const t3DVector& GetViewUp() const;
```
Retorna o vetor "para cima" de **this**.

GetVPN
```
const t3DVector& GetVPN() const;
```
Retorna o vetor normal ao plano de vista de **this**.

Pitch
```
void Pitch(double angle);
```
Rotaciona o ponto focal do ângulo `angle` em torno do vetor resultante do produto vetorial do "para cima" com o VPN centrado na posição da câmera.

Roll
```
void Pitch(double angle);
```
Rotaciona a câmera do ângulo `angle` em torno do VPN.

SetAspectRatio
```
void SetAspectRatio(double aspectRatio);
```
Ajusta o razão de aspecto de **this** com o valor do parâmetro `aspectRatio`. A razão de aspecto é a razão entre a largura e a altura da imagem captada pela câmera.

SetFocalPoint
```
void SetFocalPoint(const t3DVector& focalPoint);
```
Ajusta o ponto focal de **this** com as coordenadas de `focalPoint`, dadas em relação ao sistema global, se `focalPoint` for diferente da posição da câmera.

```
void SetFocalPoint(double x, double y, double z);
```
Ajusta o ponto focal de **this** com as coordenadas globais `x`, `y`, `z`.

SetPosition
```
void SetPosition(const t3DVector& position);
```
Ajusta a posição de **this** com as coordenadas de `position`, dadas em relação ao sistema global, se `position` for diferente do ponto focal.

```
void SetPosition(double x, double y, double z);
```
Ajusta a posição de **this** com as coordenadas globais `x`, `y`, `z`.

SetProjection
```
void SetProjection(tProjectionType ptype);
```
Ajusta o tipo de projeção efetuada por **this**.

SetViewAngle
```
void SetViewAngle(double angle);
```
Ajusta o ângulo de vista de **this** com o valor de `angle`, se $1^0 \leq$ `angle` $\leq 179^0$.

SetViewUp
```
void SetViewUp(const t3DVector& vup);
```
Ajusta o vetor "para cima" de **this** com as coordenadas de `vup`, dadas em relação ao sistema global, se `vup` não for coplanar com a direção de projeção.

```
void SetViewUp(double x, double y, double z);
```
Ajusta o vetor "para cima" de **this** com as coordenadas globais `x`, `y`, `z`.

Switch
```
void Switch(bool on);
```
"Liga" **this**, se `on` for **true**; "desliga" caso contrário.

Yaw
```
void Yaw(double angle);
```
Rotaciona o ponto focal do ângulo `angle` em torno do vetor "para cima" centrado na posição da câmera.

Zoom
```
void Zoom(double s);
```
Divide as dimensões da janela de vista pelo valor de `s`. `s`≥ 1 caracteriza um *zoom-in*; `s`≥ 1 caracteriza um *zoom-out*.

Classe tCameraDialog — viewdlg.h

Um objeto da classe `tCameraDialog` é uma caixa de diálogo que permite o ajuste dos parâmetros da câmera de uma vista de uma aplicação OSW. A caixa de diálogo possui botões que executam as operações de azimute, elevação, guinada, arfagem, rolagem, *zoom*, *pan* e *dolly*, discutidas no Capítulo 7, além de controles de especificação do tipo de projeção e das coordenadas da posição e do ponto focal da câmera.

Veja também Classe `tCamera`

Construtor Público

Construtor
```
tCameraDialog(TWindow* parent);
```
Inicializa o objeto base `TDialog` com `parent` e constrói os botões da caixa de diálogo. Não precisamos explicitamente criar uma caixa de diálogo da classe `tCameraDialog`; a construção do objeto é efetuada automaticamente pela aplicação.

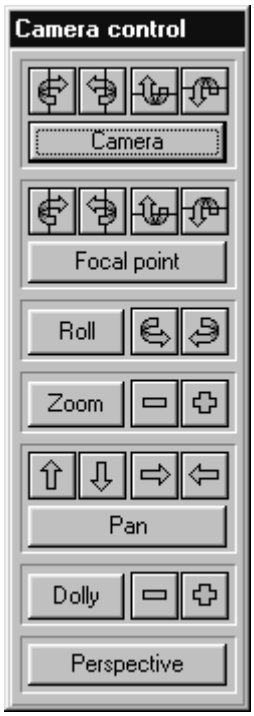
Veja também Classe `TDialog` (OWL)

Métodos Públicos

GetView
```
inline tView* GetView();
```
Retorna um ponteiro para a vista que contém a câmera manipulada por **this**.

Veja também Classe `tView`

SetView
```
void SetView(tView* view);
```
Torna `view` a vista que contém a câmera manipulada por **this**.



Classe tCell **cell.h**

A classe `tCell` é uma classe abstrata que representa uma célula genérica de um modelo de decomposição por células, como visto no Capítulo 3. A interface da classe contém métodos virtuais puros para cálculo das funções de forma de uma célula e suas derivadas e para determinação dos isopontos, isolinhas ou isosuperfícies de uma célula. A classe `tCell` é classe base virtual das classes de elementos finitos e de elementos de contorno.

Veja também Classes `tMesh`, `tFiniteElement` e `tBoundaryElement`

Construtor e Destrutor Públicos

Construtor
`tCell(int n);`
Constrói uma célula com `n` nós.

Veja também Classe `tNode`

Destrutor
`virtual ~tCell();`
Destrutor virtual.

Métodos Públicos

ComputeJacobianMatrixAt
`tJacobianMatrix ComputeJacobianMatrixAt(double xi[]) const;`
Retorna a matriz jacobiana da célula no ponto de coordenadas adimensionais `xi`. A dimensão da matriz jacobiana é $k \times 3$, onde k é a dimensão topológica da célula. A matriz jacobiana é utilizada na integração numérica de funções sobre o domínio da célula, como visto no Capítulo 6.

Veja também Classe `tJacobianMatrix`

ComputePositionAt
```
virtual t3DVector ComputePositionAt(double xi[]) const;
```
Retorna o vetor que contém as coordenadas globais x, y, z de um ponto da célula definido pelas coordenadas adimensionais `xi`, de acordo com a Equação (3.17) do Capítulo 3. O método é declarado como virtual e pode, portanto, ser sobrecarregado nas classes derivadas de `tCell`.

Veja também Classe `t3DVector`, `tCell::ComputeShapeFunctionsAt`

ComputeShapeDerivativesAt
```
virtual tMatrix ComputeShapeDerivativesAt(double xi[]) const= 0;
```
Retorna uma matriz com os valores das derivadas das funcões de forma da célula no ponto de coordenadas adimensionais `xi`. O número de linhas da matriz é igual ao número de nós da célula; o número de colunas da matriz é igual à dimensão topológica da célula.

Veja também Classe `tMatrix`

ComputeShapeFunctionsAt
```
virtual tVector ComputeShapeFunctionsAt(double xi[]) const = 0;
```
Retorna um vetor com os valores das funcões de forma da célula no ponto de coordenadas adimensionais `xi`. A dimensão do vetor é igual ao número de nós da célula. O método é virtual puro e *deve* ser sobrecarregado nas classes derivadas de `tCell`.

Veja também Classe `tVector`

Contour
```
virtual void Contour(double value, tGraphicModel* model) = 0;
```
Gera os isopontos, isolinhas ou isosuperfícies de **this** para o valor `value`. Os primitivos gráficos são colocados em `model`.

GetDimension
```
virtual int GetDimension() const = 0;
```
Retorna a dimensão topológica da célula. Uma célula de dimensão topológica igual a 1 é uma segmento de curva; uma célula de dimensão topológica igual a 2 é uma porção de superfície; e uma célula de dimensão topológica igual a 3 é uma região de espaço. A dimensão *física* de uma célula é sempre igual a 3.

GetMaterial
```
tMaterial GetMaterial() const;
```
Retorna o material da célula.

Veja também Classe `tMaterial`

GetNode
```
inline const tNode GetNode(int i) const;
```
Retorna um ponteiro do tipo *rvalue* para o `i`-ésimo nó da célula. Um *rvalue* pode aparecer somente no lado direito de uma expressão de atribuição.

Veja também Classe `tNode`

GetNumberOfNodes
```
inline int GetNumberOfNodes() const
```

Retorna o número de nós da célula.

SetMaterial
`void SetMaterial(tMaterial mat);`
Torna `mat` o novo material da célula.

Veja também Classe `tMaterial`

Atributos Protegidos

Material
`tMaterial Material;`
Material de **this**.

Veja também Classe `tMaterial`

Next
`tCell* Next;`
Ponteiro para a próxima célula na lista de células do modelo mecânico de **this**.

Nodes
`tFixedArray<tNode*> Nodes;`
Vetor de ponteiros de nós de **this**.

Previous
`tCell* Previous;`
Ponteiro para a célula anterior na lista de células do modelo mecânico de **this**.

Métodos Protegidos

SetNode
`void SetNode(int i, tNode* node);`
Torna o i-ésimo nó de **this** igual a `node`.

Classe tColor — color.h

Um objeto da clase `tColor` é uma cor no modelo RGB.

Construtores Públicos

`inline tColor()`
Construtor *default*.

`inline tColor(const tColor& color);`
Construtor de cópia. Inicializa os componentes RGB com os componentes correspondentes de `color`.

`tColor(double fr, double fg, double fb);`
Inicializa os componentes RGB com os valores dos parâmetros `fr`, `fg` e `fb`. Os valores dados devem estar entre 0 e 1.

`tColor(int ir, int ig, int ib);`
Inicializa os componentes RGB com os valores dos parâmetros `ir`, `ig` e `ib`. Os valores dados devem estar entre 0 e 255.

Atributos Públicos

b
```
double b;
```
Componente azul da cor.

g
```
double g;
```
Componente verde da cor.

r
```
double r;
```
Componente vermelho da cor.

Métodos Públicos

SetRGB
```
inline tColor& SetRGB(const tColor& color);
```
Ajusta os componentes RGB com os componentes correspondentes de `color`; retorna uma referência para **this**.

```
tColor& SetRGB(double fr, double fg, double fb);
```
Ajusta os componentes RGB com os valores dos parâmetros `fr`, `fg` e `fb`. Os valores dados devem estar entre 0 e 1. Retorna uma referência para **this**.

```
tColor& SetRGB(int ir, int ig, int ib);
```
Ajusta os componentes RGB com os valores dos parâmetros `ir`, `ig` e `ib`. Os valores dados devem estar entre 0 e 255. Retorna uma referência para **this**.

Operadores Públicos

operator =
```
inline tColor& operator =(const tColor& color);
```
Ajusta os componentes RGB com os valores dos componentes de `color`; retorna uma referência para **this**.

operator +
```
tColor operator +(const tColor& color) const;
```
Retorna uma nova cor resultante da adição dos componentes de **this** com os componentes correspondentes de `color`.

operator +=
```
tColor& operator +=(const tColor& color);
```
Adiciona os componentes RGB de **this** com os componentes correspondentes de `color`; retorna uma referência para **this**.

operator *
```
tColor operator *(double scale) const;
```
Retorna uma nova cor resultante da multiplicação dos componentes de **this** com `scale`.

operator *=
```
tColor& operator *=(double scale);
```

Multiplica os componentes RGB de **this** com `scale`; retorna uma referência para **this**.

operator ==
```
inline bool operator ==(const tColor& color) const;
```
Retorna **true** se as componentes RGB de **this** são iguais às componentes RGB correspondentes de `color`; retorna **false** caso contrário.

operator !=
```
inline bool operator !=(const tColor& color) const;
```
Retorna **true** se uma das componentes RGB de **this** é diferente da componente RGB correspondente de `color`; retorna **false** caso contrário.

Classe tColorDialog — colordlg.h

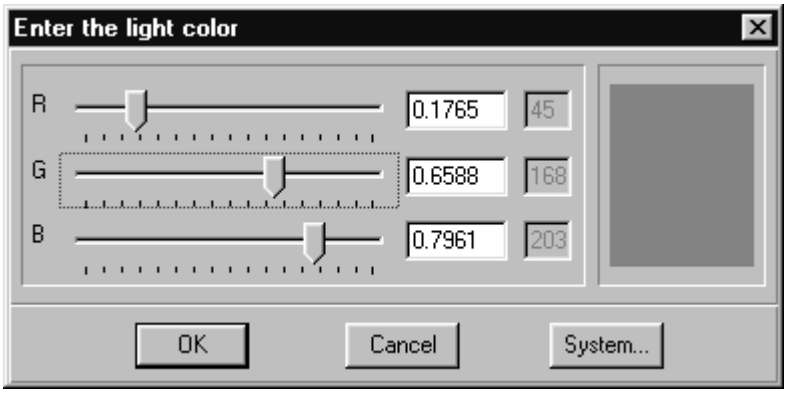
Um objeto da classe `tColorDialog` é um caixa de diálogo que permite a seleção de uma cor no modelo de cores RGB. A cor pode ser definida ajustando-se as componentes R, G e B ou a partir de um conjunto pré-definido de cores de uma biblioteca de cores. `tColorDialog` é derivada da classe `TDialog`.

Veja também Classe `TDialog` (OWL), classe `tColor`

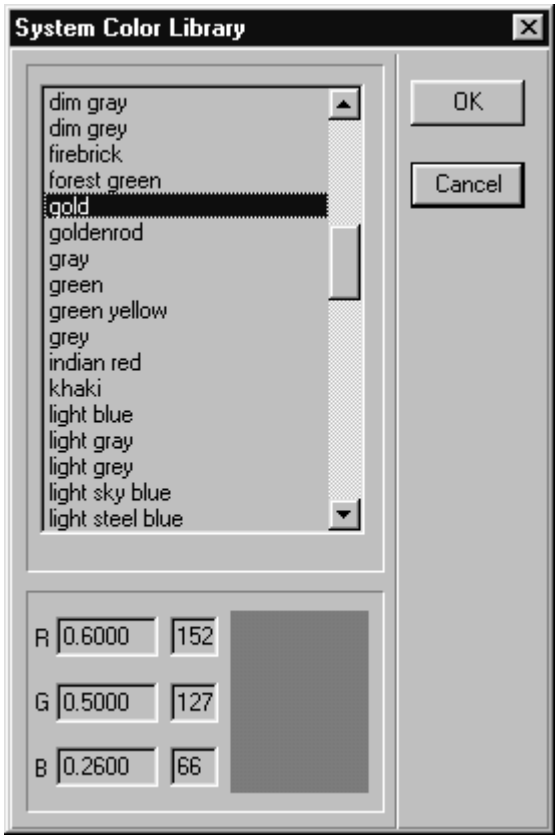
Exemplo

O seguinte trecho de código define a cor da luz ambiente de um estúdio.

```
// Ambient é a cor da luz ambiente, inicialmente branca
tColor Ambient(255, 255, 255);
//
// Agora vamos executar o diálogo
//
tColorDialog(0, "Ambient light color", Ambient).Execute();
```



O método `Execute` é implementado na classe base `TDialog`. A cor pode ser selecionada, também, a partir do conjunto de cores da biblioteca de cores do sistema (botão System), como mostrado na figura a seguir.



Construtor e Destrutor Públicos

Construtor
`tColorDialog(TWindow* parent, const char* title, tColor& color);`
Constrói um objeto `tColorDialog` filho da janela `parent` e com título `title`. O parâmetro `color` é a cor que será definida no diálogo.

Veja também Classe `TWindow`

Destrutor
`~tColorDialog();`
Destrói o `tColorDialog`.

Classe tColorMapFrame — cmapview.h

Um objeto da classe `tColorMapFrame` é uma janela que exibe em sua área cliente uma tabela de cores, definida no Capítulo 7.

Construtor Público

Construtor
`tColorMapFrame(TWindow* parent, tLookupTable* lt = 0);`
Inicializa o objeto base `TFrameWindow` com `parent`. Ajusta a tabela de cores exibida por **this** com `*lt`. Se `lt` for nulo, cria uma nova tabela de cores *default* para this.

Veja também Classes `TFrameWindow` (OWL), `tLookupTable`

Métodos Públicos

GetLookupTable
```
inline tLookupTable* GetLookuTable();
```
Retorna um ponteiro para a tabela de cores exibida por **this**.

SetLookupTable
```
void SetLookupTable(tLookupTable* lt);
```
Torna `lt` a tabela de cores exibida por **this**.

Classe tColorMapView — cmapview.h

Um objeto da classe `tColorMapView` representa uma vista que contém uma janela de tabela de cores, objeto da classe `tColorMapFrame`. A classe deriva virtualmente de `tView` e define comandos que permitem o controle da exibição da janela de tabela de cores. Classes de vistas que apresentam resultados de visualização de escalares, tais como mapas de cores e isolinhas, podem derivar de `tColorMapView`, se quisermos mostrar a tabela de cores correspondente aos escalares (veja a classe `tResView` no Capítulo 11).

Veja também Classes `tColorMapFrame`, `tView`

Construtor Público

Construtor
```
inline tColorMapView(tScene& scene, tCamera* camera);
```
Inicializa o objeto base `tView` com os parâmetros `scene` e `camera`. Constrói a janela de tabela de cores de **this**.

Atributo Protegido

ColorMapFrame
```
tColorMapFrame* ColorMapFrame
```
Ponteiro para a tabela de janela de cores de **this**.

Comandos

set_color_range
Executa uma caixa de diálogo solicitando a entrada de duas cores correspondentes às cores associadas ao menor e ao maior escalares da tabela de cores da janela de tabela de cores exibida por **this**. Ajusta o intervalo de cores da tabela de cores da janela de tabelas de cores exibida de **this** com as cores dadas.

Veja também Classe `tLookupTable`

set_scalar_range
Executa uma caixa de diálogo solicitando dois números reais correspondentes ao menor e ao maior escalares da tabela de cores da janela de tabela de cores exibida por **this**. Ajusta o menor e o maior escalares da tabela de cores da janela de tabelas de cores de **this** com os valores dados.

Veja também Classe `tLookupTable`

show_color_map_frame
Exibe ou esconde a janela de tabela de cores de **this**.

Classe tColumn datawin.h

A classe `tColumn` é uma classe abstrata que descreve a estrutura e funcionalidade de uma coluna de uma janela de resultados de análise. Uma coluna de uma janela de resultados de análise é responsável pela exibição de determinado atributo dos nós de um modelo de decomposição por células. A classe declara atributos tais como o título e largura da coluna e uma interface com um método virtual puro para exibição dos dados da coluna.

Veja também Classe `tDataWindow`

Construtor Público

Construtor
```
tColumn(uint id, const char* title, uint w, uint style);
```
Inicializa o identificador, o título, a largura e o estilo da coluna com `id`, `title`, `w` e `style`, respectivamente.

Métodos Públicos

GetTitle
```
const char* GetTitle() const;
```
Retorna um ponteiro para a cadeia de caracteres que contém o título de **this**.

Print
```
virtual const char* Print(tNode& node) = 0;
```
Retorna um ponteiro para a cadeia de caracteres que contém os dados formatados correspondentes ao atributo de `node` exibido por **this**. O método deve ser sobrecarregado nas classes concretas derivadas de `tColumn`.

Veja também Classe `tNode`

Atributos Protegidos

Id
```
uint Id;
```
Identificador de **this**.

Style
```
uint Style;
```
Estilo de exibição de **this**. Definido para uso futuro.

Title
```
const char* Title;
```
Ponteiro para a cadeia de caracteres que contém o título de **this**.

W
```
uint W;
```
Largura de **this** (em *pixels*).

X
`uint X;`
Coluna inicial de **this**, relativa à origem da área cliente da janela de resultados de análise de **this**.

Veja também Classe `tDataWindow`

Classe tContourFilter — contour.h

Um objeto da classe `tContourFilter` representa um filtro de geração de isopontos, isolinhas e isosuperfícies de um modelo mecânico. A entrada do filtro é a malha de elementos do modelo mecânico. A saída do filtro é um modelo gráfico contendo os isopontos, isolinhas e isosuperfícies do modelo de entrada. O filtro possui um vetor de valores escalares a partir dos quais os "contornos" serão gerados.

Veja também Classe `tMesh`, classe `tGraphicModel`

Construtor Público

Construtor
`tContourFilter();`
Construtor *default*.

Métodos Públicos

Execute
`tGraphicModel* Execute();`
Executa o filtro, se existir o modelo de entrada. Retorna um ponteiro para o modelo gráfico resultante do processo.

GenerateValues
`void GenerateValues(int n, double min, double max);`
Cria o vetor de valores escalares de **this**. O vetor possui `n` entradas, sendo `min` o valor mínimo e `max` o valor máximo dos escalares.

SetInput
`void SetInput(tMesh* mesh);`
Torna `mesh` o modelo de entrada de **this**.

SetValue
`void SetValue(int i, double value);`
Ajusta a i-ésima entrada do vetor de escalares de **this** com o valor `value`.

Classe tDataWindow — datawin.h

Um objeto da classe `tDataWindow` representa uma janela de exibição de dados numéricos correspondentes aos resultados da análise de um modelo mecânico de uma estrutura. A janela contém uma lista de colunas, sendo cada coluna um objeto de uma classe derivada da classe `tColumn` responsável pela exibição de determinado atributo dos nós do modelo mecânico. A interface da classe é definida por métodos que permitem a adição de colunas à lista de colunas da janela e remoção de colunas da lista de colunas da janela.

Veja também Classe `tColumn`

Enumeração Pública

tPlacement

Valor	*Significado*
`After`	Coluna adicionada após outra coluna
`Before`	Coluna adicionana antes de outra coluna

Veja também `Insert()`

Construtor e Destrutor Públicos

Construtor

```
tDataWindow(tMesh& mesh);
```

Inicializa o objeto base `TWindow`. `mesh` é a malha do modelo mecânico cujos dados são exibidos por **this**.

Veja também Classes `tMesh`, `TWindow` (OWL)

Destrutor

```
virtual ~tDataWindow();
```

Destrói a lista de colunas de **this**.

Métodos Públicos

Insert

```
void Insert(tColumn* column,
  tPlacement p = After, tColumn* sibling = 0);
```

Adiciona a coluna `column` à lista de colunas de **this**. A coluna `column` é adicionada antes ou depois da coluna `sibling`, dependendo do valor do parâmetro `p`. Se `sibling` for nulo, `column` é adicionada no final da lista de colunas de **this**.

Veja também `Remove()`

Remove

```
void Remove(tColumn* column);
```

Remove `column` da lista de colunas de **this**. `column` deve ter sido previamente adicionada à lista de colunas de **this** com o método `Insert()`.

Veja também `Insert()`

Atributos Protegidos

Columns

```
tColumn* Columns;
```

Ponteiro para a primeira coluna da lista de colunas de **this**.

Current

```
tColumn* Current;
```

Ponteiro para a coluna corrente de **this**.

CurrenRow
`uint CurrentRow;`
Inteiro que identifica a linha corrente de **this**.

Mesh
`tMesh* Mesh;`
Ponteiro para a malha do modelo mecânico cujos dados são exibidos por **this**.

NumberOfColumns
`uint NumberOfColumns;`
Número de colunas da lista de colunas de **this**.

NumberOfRows
`uint NumberOfRows;`
Número de linhas de **this**, igual ao número de nós da malha do modelo mecânico cujos dados são exibidos por **this**.

Classe tDCObject — dfile.h

A classe `tDCObject` é uma classe abstrata que descreve a funcionalidade de um objeto DC genérico de OSW. Um objeto DC é um objeto gráfico exibido em alguma janela de vista que representa um outro objeto. Uma fonte de luz, por exemplo, é representada em uma vista por um objeto DC definido por 4 pequenas linhas amarelas, uma horizontal, uma vertical e duas diagonais, que se cruzam no ponto correspondente à posição da luz. Um objeto DC pode ser desenhado e selecionado.

Destrutor Público

Destrutor
`virtual ~tDCObject();`
Destrutor virtual. Sem funcionalidade.

Métodos Públicos

Draw
`virtual void Draw(TDC& dc) const = 0;`
Método virtual puro de desenho do objeto DC. Desenha **this** no dispositivo de contexto `dc`. Deve ser sobrecarregado nas classes concretas derivadas de `tDCObject`.

Veja também Classe `TDC` (OWL)

GetColor
`virtual TColor GetColor() const;`
Método virtual que retorna a cor do objeto DC.

Veja também Classe `TColor` (OWL)

Pick
`virtual bool Pick(const tViewport& vport, int mode) const = 0;`
Método virtual puro de seleção do objeto DC. Retorna **true** se **this** é interceptado pelo retângulo (`mode == 0`) ou totalmente contido no retângulo (`mode == 1`) definido por `vport`; retorna **false** caso contrário.

Veja também Classe `tViewport`

Classe tDCObjectT — dfile.h

```
template <class T> class tDCObjectT;
```
A classe paramétrica abstrata `tDCObjectT`, derivada de `tDCObject`, associa um objeto da classe `T` a um objeto DC.

Veja também Classe `tDCObject`

Construtor Público

Construtor
```
inline tDCObjectT(const T& object);
```
Inicializa o objeto associado a **this** com `object`.

Método Público

GetObject
```
inline T* GetObject();
```
Retorna um ponteiro para o objeto associado a **this**.

Classe tDisplayFile — dfile.h

```
template <class T> class tDisplayFile;
```
A classe paramétrica `tDisplayFile` representa um arquivo de imagens da classe `T`. Um arquivo de imagens é um *container* de objetos DC pertencente a uma vista de uma cena, sendo cada objeto DC associado a um objeto da classe `T` pertencente à cena. Um arquivo de imagens é responsável pelo desenho de seus objetos DC e pela seleção de seus objetos DC na vista proprietária. Como cada objeto DC está associado a um objeto da classe `T`, ao selecionarmos o objeto DC estamos selecionando, também, o objeto da classe `T` associado. Utilizamos os objetos DC e os arquivos de imagems para implementar os mecanismos de seleção de objetos nas janelas de vistas de uma cena.

Veja também Classe `tDCObjectT`

Construtor Público

Construtor
```
tDisplayFile(tRenderer& renderer);
```
Inicializa o arquivo de imagens. Utiliza os parâmetros de projeção e mapeamento de `renderer` para geração dos objetos DC de **this**.

Métodos Públicos

Add
```
tDCObjectT<T>* Add(const T& t);
```
Cria um objeto DC associado ao objeto `t` e adiciona o objeto DC à coleção de objetos DC de **this**. Retorna um ponteiro para o objeto DC.

Delete
```
virtual void Delete(T& t);
```
Se existir algum objeto DC associado ao objeto `T` na coleção de objetos DC de **this**, remove e destrói o objeto DC.

DeleteAll
```
inline void DeleteAll();
```
Destrói todos os objetos DC de **this**.

Draw
```
virtual void Draw(TDC& dc) const;
```
Desenha todos os objetos DC de **this** no dispositivo de contexto `dc`.

Veja também Classe `TDC` (OWL)

Pick
```
virtual void Pick(tPickInfoT<T>& info);
```
Seleciona os objetos DC de **this** de acordo com as informações contidas em `info`.

Veja também `tDCObject::Draw()`, classe `tPickInfoT`

PickSingle
```
virtual T* PickSingle(const tViewport& vport, int mode) const;
```
Seleciona um único objeto DC com os parâmetros `vport` e `mode`. Se um objeto DC foi selecionado, retorna um ponteiro para o objeto da classe `T` associado; retorna 0 caso contrário.

Veja também `tDCObject::Pick()`

Classe tDocPath — oswdoc.h

Um objeto da classe `tDocPath` encapsula os dados que definem o filtro e a extensão dos nomes de arquivos de uma classe de documentos de uma aplicação OSW.

Veja também Classe `tDocument`

Construtor Público

Construtor
```
inline tDocPath(const char* filter, const char* defaultExt);
```
Inicializa o objeto base `TOpenSaveDialog::TData` com os parâmetros `filter` e `defaultExt`.

Veja também Classe `TOpenSaveDialog::TData` (OWL)

Classe tDocument — oswdoc.h

Um objeto da classe `tDocument` representa uma base de dados genérica de uma aplicação OSW. Um base de dados é definida por uma coleção de modelos e uma coleção de cenas. A interface da classe contém métodos de gerenciamento das coleções de modelos e cenas de um documento. As classes derivadas de `tDocument` devem ser responsáveis pela inicialização das cenas manipuladas pelo documento.

Veja também Classe `tModel`, classe `tScene`

Construtor e Destrutor Públicos

Construtor
`tDocument(tApplication& app);`
Construtor do documento. `app` é a aplicação que manipula **this**.

Destrutor
`virtual ~tDocument();`
Destrutor virtual. Destrói as listas de modelos e de cenas de **this**.

Métodos Públicos

AddModel
`void AddModel(tModel* model);`
Adiciona o modelo `model` na lista de modelos de **this**.

AddScene
`void AddScene(tScene* scene);`
Adiciona a cena `scene` na lista de cenas de **this**.

CanClose
`virtual bool CanClose();`
Se `IsDirty()` retornar **true**, abre uma caixa de diálogo informando que o documento foi alterado e confirmando se **this** deve ser salvo. Se o resultado da consulta for negativo, o método retorna **false**; caso contrário, executa o método `Save()` e retorna **true**. Se `IsDirty()` retornar **false**, retorna **true**.

Close
`virtual bool Close();`
Fecha o documento e destrói todas as janelas de vistas das cenas de **this**. Retorna **true** se a operação foi bem sucedida; caso contrário, retorna **false**.

Commit
`virtual bool Commit(boolforce);`
Armazena em meio persistente os modelos e as cenas do documento, se `IsDirty()` retornar **true**, ou se `IsDirty()` retornar **false** e `force` for **true**.

DeleteAll
`void DeleteAll();`
Remove todos os componentes das listas de modelos de **this**. Envia a mensagem `DeleteAll()` a todas as cenas do documento.

DeleteModel
`void DeleteModel(tModel* model);`
Remove o modelo `model` da lista de modelos de **this** e destrói `model`. O modelo só é realmente destruído se seu contador de referência for igual a zero.

DeleteScene
`void DeleteScene(tScene* scene);`
Remove a cena `scene` da lista de cenas de **this** e destrói `scene`.

GetApplication
`inline tApplication* GetApplication();`

Retorna um ponteiro para a aplicação que manipula **this**.

GetDocPath
```
inline const char* GetDocPath();
```
Retorna o nome de caminho de **this**.

IsDirty
```
virtual bool IsDirty();
```
Retorna o valor de `DirtyFlag`.

IsOpen
```
virtual bool IsOpen();
```
Retorna **true** se **this** está aberto.

Open
```
virtual bool Open();
```
Se `IsOpen()` retornar **true**, executa `Close()`. Em seguida, abre o documento.

SetDocPath
```
void SetDocPath(const char* docPath);
```
Torna `docPath` o novo nome de caminho de **this**.

Atributos Protegidos

Application
```
tApplication* Application;
```
Ponteiro para a aplicação que manipula **this**.

DirtyFlag
```
bool DirtyFlag;
```
Flag que indica se o documento foi alterado. Utilizada pelo método `IsDirty()`.

DocPath
```
char* DocPath;
```
Nome de caminho de **this**.

Métodos Protegidos

ConstructScenes
```
virtual void ConstructScenes();
```
Construtor "virtual" das cenas do documento. Sem funcionalidade em `tDocument`.

Classe tDOF — dof.h

A classe `tDOF` representa um grau de liberdade de um vértice de um modelo mecânico.

Atributos Públicos

EquationNumber
```
int EquationNumber;
```
Número da equação de **this** no sistema de equações lineares de um objeto derivado da classe `tSolver`. Se **this** for restringido, seu número de equação é igual a -1.

Restraint
`bool Fixed;`
Flag que indica se **this** é restringido.

P
`double P;`
Valor do esforço associado a **this**.

U
`double U;`
Valor de **this**.

Classe tDoubleList dlist.h

`template <class T> class tDoubleList;`
A classe paramétrica `tDoubleList` descreve a estrutura e funcionalidade de um *container* de objetos da classe `T`, implementado como uma lista duplamente encadeada de elementos. A interface da classe define métodos de adição e remoção de elementos; a iteração sobre os elementos de uma lista é efetuada por *iterators* da classe paramétrica `tDoubleListIterator` (*containers* e *iterators* foram discutidos no Capítulo 9).

Veja também Classe `tDoubleListIterator`

Definição de Tipo Pública

tIterFunc
`typedef void (*tIterFunc)(T*);`
Tipo correspondente a um ponteiro para uma função que toma como parâmetro um ponteiro para um objeto da classe `T` e não retorna nada.

Construtor e Destrutor Públicos

Construtor
`inline tDoubleList();`
Construtor *default*. Inicializa **this** como uma lista vazia.

Destrutor
`inline ~tDoubleList();`
Executa o método `Flush()` para esvaziar **this**.

Veja também `Flush()`

Métodos Públicos

Flush
`void Flush();`
Esvazia a lista, destruindo todos os elementos contidos em **this**.

ForEach
`void ForEach(tIterFunc func);`
Executa a função `func` para cada elemento de **this**. O elemento é passado como parâmetro para `func`.

Veja também `tIterFunc`

GetNumberOfObjects
`int GetNumberOfObjects() const;`
Retorna o número de elementos de **this**.

Insert
`void Insert(T* node);`
Adiciona o elemento `node` a **this**.

IsEmpty
`bool IsEmpty() const;`
Retorna **true** se **this** está vazia; retorna **false** caso contrário.

PeekHead
`T* PeekHead();`
Retorna um ponteiro para o primeiro elemento de **this**.

Remove
`void Remove(T* node);`
Remove o elemento `node` de **this**.

Atributos Protegidos

Head
`T* Head;`
Ponteiro para o primeiro elemento de **this**.

NumberOfObjects
`int NumberOfObjects;`
Número de elementos de **this**.

Classe tDoubleListIterator — dlist.h

`template <class T> class tDoubleListIterator;`
Um objeto da classe paramétrica `tDoubleListIterator` representa um *iterator* de `containers` da classe paramétrica `tDoubleList`. A interface da classe define métodos **inline** que permitem a iteração sobre todos os elementos de uma lista duplamente encadeada (*containers* e *iterators* foram discutidos no Capítulo 9).

Veja também Classe `tDoubleList`

Construtor Público

Construtor
`inline tDoubleListIterator(tDoubleList<T>& list);`
Inicializa o *iterator* com a lista `list`.

Métodos Públicos

Current
`inline T* Current();`
Retorna um ponteiro para o elemento corrente da lista de **this**.

Restart
```
inline void Restart();
```
Reinicializa **this**. O ponteiro para o elemento corrente passa a ser o ponteiro para o primeiro elemento da lista de **this**.

Operadores Públicos

operator int
```
inline operator int();
```
Retorna um inteiro diferente de zero se o ponteiro para o elemento corrente da lista de **this** for não nulo; retorna zero caso contrário.

operator ++
```
inline T* operator ++();
```
Pré-incremento. Ajusta o ponteiro para o elemento corrente da lista de **this** com o endereço do próximo elemento da lista e retorna o ponteiro para o elemento corrente *após* o ajuste.

```
inline T* operator ++(int);
```
Pós-incremento. Ajusta o ponteiro para o elemento corrente da lista de **this** com o endereço do próximo elemento da lista e retorna o ponteiro para o elemento corrente *antes* do ajuste.

operator −−
```
inline T* operator --();
```
Pré-decremento. Ajusta o ponteiro para o elemento corrente da lista de **this** com o endereço do elemento anterior da lista e retorna o ponteiro para o elemento corrente *após* o ajuste.

```
inline T* operator --(int);
```
Pós-decremento. Ajusta o ponteiro para o elemento corrente da lista de **this** com o endereço do elemento anterior da lista e retorna o ponteiro para o elemento corrente *antes* do ajuste.

Classe tEdge **model.h**

Um objeto da classe `tEdge` representa uma aresta global de OSW. Uma aresta global é uma aresta definida por dois vértices, somente, sem quaisquer outras informações topológicas características de um modelo geométrico específico. Um *renderer* da classe `tRenderer` (ou de uma classe derivada de `tRenderer`) utiliza arestas globais para gerar imagens fio-de-arame de um modelo. Por esse motivo, os modelos geométricos de OSW implementam *iterators* próprios de arestas globais, conforme discutimos no Capítulo 9.

Veja também Classes `tModel`, `tRenderer`

Atributos Públicos

V0
```
tVertex* V0;
```
Ponteiro para o primeiro vértice de **this**.

Veja também Classe `tVertex`

V1
`tVertex* V1;`
Ponteiro para o segundo vértice de **this**.

Veja também Classe `tVertex`

Construtores Públicos

Construtores
`inline tEdge();`
Construtor *default*. Sem funcionalidade.

`inline tEdge(tVertex* v0, tVertex* v1);`
Inicializa `V0` com `v0` e `V1` com `v1`.

Métodos Públicos

Length
`double Length() const;`
Retorna o comprimento de **this**, ou seja, a distância entre `V0` e `V1`.

Vector
`t3DVector Vector() const;`
Retorna o vetor obtido pela diferença puntual dos pontos que definem as posições de `V1` e `V0`.

Veja também Classe `t3DVector`

Classe tEnvironment — environm.h

Um objeto da classe `tEnvironment` encapsula as informações sobre as cores e fontes utilizadas nos diversos elementos de interface de uma aplicação de modelagem. Os métodos públicos de `tEnvironment` possibilitam a modificação das cores e fontes do ambiente de uma aplicação.

Veja também Classe `tApplication`

Construtor e Destrutor Públicos

Construtor
`tEnvironment(const char* envFileName = 0);`
Construtor *default*. Inicializa as cores e fontes do ambiente a partir das informações do arquivo de ambiente definido pelo parâmetro `envFileName`. Se `envFileName` não for definido, as cores e fontes são inicializadas a partir dos valores *default* do sistema.

Destrutor
`~tEnvironment();`
Normalmente chamado ao término de execução de uma aplicação de modelagem, destrói as tabelas internas de fontes e cores do ambiente da aplicação.

Métodos Públicos

GetCommandBkColor
```
inline TColor GetCommandBkColor() const;
```
Retorna a cor de fundo da janela de comando de uma aplicação.

Veja também Classe `TColor` (OWL)

GetTokenBkColor
```
inline TColor GetTokenBkColor(tTokenType tokenType) const;
```
Retorna a cor de fundo dos *tokens* do tipo `tokenType`.

Veja também Constantes `tTokenType`

GetTokenFgColor
```
inline TColor GetTokenFgColor(tTokenType tokenType) const;
```
Retorna a cor de frente dos *tokens* do tipo `tokenType`.

GetTokenFont
```
inline TFont& GetTokenFont(tTokenType tokenType) const;
```
Retorna uma referência para a fonte dos *tokens* do tipo `tTokenType`.

Veja também Classe `TFont` (OWL)

GetViewBkColor
```
inline TColor GetViewBkColor() const;
```
Retorna a cor de fundo das janelas de vista de uma aplicação.

GetViewCursorColor
```
inline TColor GetViewCursorColor() const;
```
Retorna a cor do cursor utilizado nas janelas de vista de uma aplicação.

Classe tFace **model.h**

A classe abstrata `tFace` representa a funcionalidade de uma face global de OSW. Uma face global é uma face que não possui informações geométricas e/ou topológicas características de um modelo geométrico específico. Um *renderer* da classe `tScanner` (ou de uma classe derivada de `tScanner`) utiliza faces globais para gerar imagens com remoção de superfícies escondidas de um modelo. Por esse motivo, os modelos geométricos de OSW implementam *iterators* próprios de faces globais, conforme discutimos no Capítulo 9.

Veja também Classes `tModel`, `tScanner`

Métodos Públicos

GetEdgeIterator
```
inline tEdgeIterator GetEdgeIterator();
```
Retorna um *iterator* de arestas globais de **this**. O *iterator* é um objeto da classe derivada de `tInternalIterator`.

Veja também Classes `tEdge`, `tInternalIterator`

GetMaterial
```
virtual tMaterial GetMaterial() const = 0;
```
Método virtual puro que retorna o material de **this**. O material é suposto constante em toda a superfície da face. O método deve ser sobrecarregado em classes concretas derivadas de `tFace`.

GetNumber
```
virtual int GetNumber() const = 0;
```
Método virtual puro que retorna um inteiro identificador da face. O método deve ser sobrecarregado em classes concretas derivadas de `tFace`.

GetVertexIterator
```
inline tVertexIterator GetVertexIterator();
```
Retorna um *iterator* de vértices de **this**. O *iterator* é um objeto da classe derivada de `tInternalIterator`.

Veja também Classes `tVertex`, `tInternalIterator`

Classe tFEShellMeshGenerator — 2dmeshge.h

Um objeto da classe `tFEShellMeshGenerator` é um filtro de geração de malhas de elementos finitos de casca definidos no Capítulo 6. A entrada do filtro é um modelo da classe `tShell`; a saída do filtro é um modelo da classe `tShellMesh`. A classe deriva diretamente de `t2DMeshGenerator`.

Veja também Classes `t2DMeshGenerator`, `tShell`, `tShellMesh`

Métodos Públicos

GetInput
```
inline tShell* GetInput();
```
Retorna um ponteiro para o modelo de entrada de **this**.

SetInput
```
void SetInput(tShell* shell);
```
Torna `shell` o modelo de entrada de **this**.

Métodos Protegidos

MakeCell
```
t2DCell* MakeCell();
```
Constrói uma célula da classe `t3NShell`, derivada de `t2DCell`, e retorna um ponteiro para a célula. O método é declarado como virtual puro na classe `t2DMeshGenerator`.

Veja também Classe `t3NShell`

MakeInitialFront
```
void MakeInitialFront(tBoundaryFace& bf);
```
Constrói o fronte inicial da face de contorno `bf`. O método é declarado como virtual puro na classe base `t2DMeshGenerator`.

Veja também Classe `tBoundaryFace`

MakeMesh
```
tMesh* MakeMesh(const char* name);
```
Constrói um modelo de decomposição por células da classe `tShellMesh` e retorna um ponteiro para o modelo.

MakeNode
```
tNode* MakeNode();
```
Constrói um nó da classe `t6FNode`, derivado de `tNode`, e retorna um ponteiro para o nó. O método é declarado como virtual puro na classe base `t2DMeshGenerator`.

Veja também Definição de tipo `t6FNode`

Classe tFESolver — fesolver.h

Um objeto da classe `tFESolver` representa um analisador de estruturas elastostáticas pelo método dos elementos finitos. A classe é derivada de `tSolver` e sua interface define métodos próprios de construção e montagem do sistema de equações lineares e de término do processo de análise.

Veja também Classe `tSolver`

Construtor Público

Construtor
```
tFESolver(tMesh& mesh);
```
Inicializa a classe base `tSolver` com `mesh`.

Métodos Protegidos

AssembleSystem
```
void AssembleSystem();
```
Montagem do sistema de equações lineares. Solicita a todos os elementos finitos do modelo de entrada a matriz de rigidez, o vetor de esforços nodais equivalentes e o vetor de localização do elemento. Solicita ao sistema linear a adição da matriz de rigidez e do vetor de esforços do elemento aos lados esquerdo e direito do sistema, respectivamente.

Veja também Classe `tFiniteElement`

ConstructLinearSystem
```
tLinearSystem* ConstructLinearSystem();
```
Constrói um sistema linear com matriz simétrica armazenada em banda, objeto da classe `tBandSystem`.

Veja também Classe `tBandSystem`

Terminate
```
void Terminate();
```
Executa `tSolver::Terminate()` e calcula os esforços nos vértices do modelo mecânico de entrada.

Veja também Classe `tSolver`

Classe tFile — pfile.h

Um objeto da classe `tFile` representa um arquivo genérico, usualmente manipulado por leitores de OSW, objetos de classes derivadas da classe `tReader` (definimos *leitores* no Capítulo 7).

Veja também Classe `tReader`

Construtores e Destrutor Públicos

Construtores
```
inline tFile();
```
Construtor *default*. Inicializa **this** como sendo um arquivo fechado.

```
inline tFile(const char* name, uint mode, uint perm);
```
Executa o método `Open()`.

Veja também `Open()`

Destrutor
```
inline ~tFile();
```
Inicializa **this** como sendo um arquivo fechado e executa o método `Close()`.

Veja também `Close()`

Métodos Públicos

Close
```
bool Close();
```
Se **this** está aberto, fecha **this**.

IsOpen
```
inline bool IsOpen() const;
```
Retorna **true** se **this** está aberto; retorna **false** caso contrário.

Length
```
inline long Length() const;
```
Retorna o número de bytes de **this**.

```
inline void Length(long len);
```
Ajusta o tamanho de **this** com `len` bytes, truncando ou espandindo o arquivo.

Open
```
bool Open(const char* name, uint mode, uint perm);
```
Executa o método `Close()` e abre **this** com o nome `name`, com mode `mode` e com a permissão `perm`. As *flags* de modo e permissão são definidas nos arquivos de cabeçalho do C++ `io.h` e `fcntl.h`.

Veja também `Close()`

Position
```
inline long Position() const;
```
Retorna a posição corrente de leitura/escrita de **this**.

Read
```
inline int Read(void* buf, int len);
```
Efetua a leitura de `len` bytes de **this** e armazena os dados em `buf`. Retorna o número de bytes lidos ou -1, em caso de erro.

Veja também `Write()`

Seek
```
inline long Seek(long offset, int origin);
```
Posiciona a posição corrente de leitura/escrita de **this** no deslocamento `offset`, relativo à origem `origin`, e retorna a posição corrente. Os valores possíveis de `origin` são definidos no arquivo de cabeçalho C++ `io.h`.

Write
```
inline int Write(const void* buf, int len);
```
Efetua a escrita em **this** de `len` bytes contidos em `buf`. Retorna o número de bytes escritos ou -1, em caso de erro.

Veja também `Read()`

Classe tFiniteElement — felement.h

A classe abstrata `tFiniteElement` representa o comportamento de um elemento finito genérico de um modelo mecânico. A interface da classe define métodos virtuais para o cálculo da matriz de rigidez e do vetor de esforços nodais equivalentes. `tFiniteElement` deriva virtualmente da classe `tCell`.

Veja também Classe `tCell`

Construtor e Destrutor Públicos

Construtor
```
tFiniteElement(int n);
```
Inicializa a classe base virtual `tCell` com o parâmetro `n`.

Destrutor
```
~tFiniteElement();
```
Executa o método `Terminate()`.

Métodos Públicos

GetLoadVector
```
tVector GetLoadVector();
```
Retorna o vetor de esforços nodais equivalentes de **this**. Se o vetor ainda não foi calculado, executa o método virtual `ComputeLoadVector()`.

GetLocationArray
```
tLocationArray* GetLocationArray();
```
Retorna o vetor de localização de **this**, tal como discutido no Capítulo 9.

GetStiffnessMatrix
```
tMatrix GetStiffnessMatrix();
```

Retorna a matriz de rigidez de **this**. Se a matriz ainda não foi calculada, executa o método virtual `ComputeStiffnessMatrix()`.

Terminate
```
virtual void Terminate();
```
Destrói a matriz de rigidez e o vetor de esforços nodais de **this**.

Atributos Protegidos

LoadVector
```
tVector LoadVector;
```
Vetor de esforços nodais de **this**.

Veja também Classe `tVector`

LocationArray
```
tLocationArray* LocationArray;
```
Ponteiro para o vetor de localização de **this**.

StiffnessMatrix
```
tMatrix StiffnessMatrix;
```
Matriz de rigidez de **this**.

Veja também Classe `tMatrix`

Métodos Protegidos

ComputeLoadVector
```
virtual tVector ComputeLoadVector() = 0;
```
Calcula o vetor de esforços nodais equivalentes de **this**.

ComputeStiffnessMatrix
```
virtual tMatrix ComputeStiffnessMatrix() = 0;
```
Calcula a matriz de rigidez de **this**.

MakeLocationArray
```
virtual tLocationArray* MakeLocationArray();
```
Monta o vetor de localização de **this**.

Veja também Classe `tLocationArray`

Classe tFixedArray — array.h

```
template <class T> class tFixedArray;
```
A classe paramétrica `tFixedArray` é uma classe auxiliar simples que descreve a estrutura e funcionalidade de um *container* de objetos da classe `T`, implementado como um arranjo unidimensional com um número fixo e constante de elementos. A iteração sobre os elementos de uma arranjo pode ser efetuada por *iterators* da classe paramétrica `tFixedArrayIterator` (*containers* e *iterators* foram discutidos no Capítulo 9).

Veja também Classe `tFixedArrayIterator`

Construtor e Destrutor Públicos

Construtor
`tFixedArray(int n);`
Inicializa o arranjo com `n` entradas da classe `T`.

Destrutor
`~tFixedArray();`
Destrói as entradas de **this**.

Método Público

GetNumberOfObjects
`inline GetNumberOfObjects() const;`
Retorna o número de entradas de **this**.

Operador Público

operador []
`inline const T& operator [](int i) const;`
Retorna uma referência do tipo *rvalue* para o objeto da `i`-ésima entrada de **this**. Um *rvalue* pode aparecer somente no lado direito de uma expressão de atribuição.

`inline T& operator [](int i);`
Retorna uma referência do tipo *lvalue* para o objeto da `i`-ésima entrada de **this**. Um *lvalue* pode aparecer no lado esquerdo de uma expressão de atribuição.

Classe tFixedArrayIterator — array.h

`template <class T> class tFixedArrayIterator;`
Um objeto da classe paramétrica `tFixedArrayIterator` representa um *iterator* de *containers* da classe paramétrica `tFixedArray` (*containers* e *iterators* foram discutidos no Capítulo 9).

Construtor Público

Construtor
`inline tFixedArrayIterator(tFixedArray<T>& array);`
Inicializa o *iterator* com o arranjo `array`.

Métodos Públicos

Current
`inline T& Current();`
Retorna uma referência para o elemento da posição corrente do arranjo de **this**.

Restart
`inline void Restart();`
Reinicializa **this**. A posição corrente do arranjo de **this** passa a ser zero.

Operadores Públicos

operator int
```
inline operator int();
```
Retorna um inteiro diferente de zero se a posição corrente do arranjo de **this** for menor que o número de entradas do arranjo de **this**; retorna zero caso contrário.

operator ++
```
inline T& operator ++();
```
Pré-incremento. Incrementa a posição corrente do arranjo de **this** e retorna uma referência para o elemento da posição corrente *após* o incremento.

```
inline T& operator ++(int);
```
Pós-incremento. Incrementa a posição corrente do arranjo de **this** e retorna uma referência para o elemento da posição corrente *antes* do incremento.

operator −−
```
inline T& operator --();
```
Pré-decremento. Decrementa a posição corrente do arranjo de **this** e retorna uma referência para o elemento da posição corrente *após* o incremento.

```
inline T& operator --(int);
```
Pós-decremento. Decrementa a posição corrente do arranjo de **this** e retorna uma referência para o elemento da posição corrente *antes* do incremento.

Classe tFullSystem — fullsystem.h

Um objeto da classe `tFullSystem` é um sistema linear com matriz completa. A interface da classe sobrecarrega os métodos virtuais de solução do sistema e montagem da matriz de coeficientes e do vetor de termos independentes, herdados da classe base `tLinearSystem`.

Construtor Público

Construtor
```
tFullSystem(int m);
```
Inicializa a classe base `tLinearSystem` com o parâmetro `m`, dimensão de **this**.

Métodos Públicos

Assemble
```
void Assemble(const tMatrix& m, tLocationArray* l);
```
Adiciona a matriz `m` à matriz de coeficientes de **this**, de acordo com os números de equações contidas em `*l`.

```
void Assemble(const tVector& v, tLocationArray* l) = 0;
```
Adiciona o vetor `v` ao vetor de termos independentes de **this**, de acordo com os números de equações contidas em `*l`.

Solve
```
void Solve();
```
Calcula o vetor solução de **this**.

Classe tGraphicModel gmodel.h

Um objeto da classe `tGraphicModel` é uma coleção de primitivos gráficos, tal como definido no Capítulo 3. A classe deriva da classe abstrata `tModel` e da classe `tPrimitive`. A interface de `tGraphicModel` define métodos para adição e remoção de primitivos na coleção de primitivos de um modelo gráfico.

Construtor e Destrutor Públicos

Construtor
```
tGraphicalModel(tGraphicalModel* parent = 0);
```
Construtor *default*. Adiciona **this** na lista de primitivos de `parent`, se `parent` é diferente de zero.

Destrutor
```
~tGraphicalModel();
```
Destrói todos os componentes da lista de primitivos de **this**.

Métodos Públicos

AddPrimitive
```
void AddPrimitive(tPrimitive* prim);
```
Adiciona `prim` na lista de primitivos de **this**.

DeletePrimitive
```
void DeletePrimitive(tPrimitive* prim);
```
Remove `prim` da lista de primitivos de **this** e destrói o primitivo.

GetEdgeIterator
```
tEdgeIterator GetEdgeIterator();
```
Retorna um *iterator* para a coleção de arestas de **this**.

Veja também Tipo `tEdgeIterator`

GetFaceIterator
```
tFaceIterator GetFaceIterator();
```
Retorna um *iterator* para a coleção de faces de **this**.

Veja também Tipo `tFaceIterator`

GetNumberOfChilds
```
inline int GetNumberOfChilds() const;
```
Retorna o número de primitivos da lista de primitivos de **this**.

GetNumberOfEdges
```
int GetNumberOfEdges() const;
```
Retorna o número de arestas de **this**.

GetNumberOfFaces
```
int GetNumberOfFaces() const;
```
Retorna o número de faces de **this**.

GetNumberOfVertices
```
int GetNumberOfVertices() const;
```

Retorna o número de vértices de **this**.

GetPrimitiveIterator
```
tPrimitiveIterator GetPrimitiveIterator();
```
Retorna um *iterator* para os primitivos de **this**.

Veja também Tipo `tPrimitiveIterator`

GetVertexIterator
```
tVertexIterator GetVertexIterator();
```
Retorna um *iterator* para a coleção de vértices de **this**.

Veja também Tipo `tVertexIterator`

Transform
```
void Transform(const t3DTransfMatrix& m);
```
Transforma **this**. Envia a mensagem `Transform()` a todos os primitivos do modelo, passando `m` como parâmetro.

Classe tInputDialog — input.h

Um objeto da classe `tInputDialog` é uma caixa de diálogo que permite a entrada de expressões matemáticas simples que definem um número real ou as coordenadas de um vetor no espaço. A classe é derivada da classe `TDialog`.

Veja também Classe `TDialog` (OWL)

Construtores Públicos

Construtores
```
tInputDialog(TWindow* parent, const char* title, double& f,
  tInputRestraint ir = irNone, tWriteFormat wf = wfDefault);
```
Constrói o objeto base `TDialog` com os parâmetros `parent` e `title`. Inicializa o diálogo para receber o número real `f`, no formato de escrita `wf` e sujeito às restrições `ir`.

```
tInputDialog(TWindow* parent, const char* title, t3DVector& p,
  tInputRestraint ir = irNone, tWriteFormat wf = wfDefault);
```
Constrói o objeto base `TDialog` com os parâmetros `parent` e `title`. Inicializa o diálogo para receber as coordenadas do vetor 3D `p`, no formato de escrita `wf` e sujeito às restrições `ir`.

Veja também Classe `t3DVector`

Classe tInternalIterator model.h

```
template <class T> class tInternalIterator;
```
A classe paramétrica `tInternalIterator`, derivada de `tObjectBody`, é uma classe abstrata base que descreve a estrutura e funcionalidade de um *iterator* genérico de componentes de um modelo geométrico. A classe declara o métodos virtuais puros privados `MakeFirstObject()` e `MakeNext()`, os quais implementam, nas classes concretas derivadas de `tInternalIterator`, a funcionalidade específica do *iterator*.

Veja também Classes `tObjectBody`, `tIterator`

Métodos Públicos

GetCurrent
```
inline T* GetCurrent();
```
Retorna o ponteiro para o objeto corrente de **this**.

GetNext
```
inline T* GetNext();
```
Ajusta o ponteiro para o objeto corrente de **this** para o endereço do próximo objeto do *container* de **this**, determinado com a execução do método virtual privado `MakeNext()`. Retorna o ponteiro para o objeto corrente.

Start
```
inline T* Start();
```
Ajusta o ponteiro para o objeto corrente de **this** para o endereço do primeiro objeto do *container* de **this**, determinado com a execução do método virtual privado `MakeFirstObject()`. Retorna o ponteiro para o objeto corrente.

Construtor Protegido

Construtor
```
inline tInternalIterator(bool deleteObject = false);
```
Inicializa o *iterator*. Se a *flag* `deleteObject` for igual a **true**, o objeto corrente deve ser destruído a cada atualização do ponteiro para o objeto corrente de **this**. Se `deleteObject` for igual a **false**, o objeto corrente é propriedade do *container* de **this** e não deve ser destruído.

Classe tIntersectInfo raytrace.h

Um objeto da classe `tIntersectInfo` contém as informações de intersecção de um raio com um modelo geométrico de uma cena. Essas informações são utilizadas por um *renderer* da classe `tRayTracer` durante o processo de geração de uma imagem da cena.

Atributos Públicos

Enter
```
bool Enter;
```
Flag que indica se o raio está entrando ou saindo do modelo.

Material
`tMaterial Material;`
Material do modelo interceptado pelo raio, no ponto de intersecção.

Model
`tModel* Model;`
Ponteiro para o modelo interceptado pelo raio.

t
`double t;`
Distância entre o ponto de origem do raio e o ponto de intersecção do raio com o modelo.

Classe tIterator — model.h

`template <class T> class tIterator;`
O objetvo da classe paramétrica `tIterator` é descrever uma interface comum para os *iterators* de componentes de um modelo geométrico. A funcionalidade específica de um *iterator*, de fato, é definida por um objeto derivado da classe paramétrica abstrata `tInternalIterator`.

Veja também Classe paramétrica `tInternalIterator`

Construtores e Destrutor Públicos

Construtores
`inline tIterator(const tIterator<T>& it);`
Construtor de cópia. Inicializa os atributos de **this** com uma cópia do tipo *shallow-copy* dos atributos de `it`.

`inline tIterator(tInternalIterator<T>* internalIt);`
Inicializa o *iterator* interno de **this** com `internalIt`.

Destrutor
`inline ~tIterator();`
Destrói o *iterator* interno de **this**.

Métodos Públicos

Current
`inline T* Current();`
Retorna o ponteiro para o objeto corrente do *iterator* interno de **this**.

Restart
`inline T* Restart();`
Reinicializa o *iterator*.

Operadores Públicos

operator int
`inline operator int() const;`
Retorna um inteiro diferente de zero se o ponteiro para o objeto corrente do *iterator* interno de **this** for não nulo; retorna zero caso contrário.

operator ++
`inline T* operator ++();`
Pré-incremento. Ajusta o ponteiro do objeto corrente do *iterator* interno de **this** com o endereço do próximo objeto do container e retorna o ponteiro *após* o ajuste.

`inline T* operator ++(int);`
Pós-incremento. Ajusta o ponteiro do objeto corrente do *iterator* interno de **this** com o endereço do próximo objeto do container e retorna o ponteiro *antes* do ajuste.

Classe tJacobianMatrix — jacobian.h

Um objeto da classe `tJacobianMatrix` representa a matriz Jacobiana de uma célula de um modelo de decomposição por células.

Construtores e Destrutor Públicos

Construtores
`tJacobianMatrix(const tCell& cell, double xi[]);`
Constrói a matriz Jacobiana para a célula `cell` no ponto definido pelas coordenadas adimensionais `xi`.

Veja também Classe `tCell`

`tJacobianMatrix(const tJacobianMatrix& jacobianMatrix);`
Construtor de cópia. Constrói a matriz Jacobiana com o corpo de `jacobianMatrix`.

Destrutor
`~tJacobianMatrix();`
Decrementa o contador de referência do corpo da matriz. Se o contador de referência for igual a zero, invoca o destrutor do corpo.

Métodos Públicos

Jacobian
`double Jacobian() const;`
Retorna o "Jacobiano" de **this**.

RecomputeAt
`void RecomputeAt(double xi[])`
Calcula os elementos da matriz Jacobiana no ponto definido pelas coordenadas adimensionais `xi`, para a célula dada no construtor de **this**.

Operadores Públicos

operator ()
`double operator ()(int i, int j) const;`
Retorna o elemento da linha `i` e coluna `j` de **this**. O número de colunas da matriz sempre é 3 e o número de linhas é igual à dimensão topológica da célula dada no contrutor de **this**.

Classe tLight — light.h

A classe `tLight` representa uma luz genérica de uma cena. A interface da classe define métodos de posicionamento e ajuste da cor de uma luz.

Construtores Públicos

Construtores
```
tLight(const t3DVector& p);
```
Inicializa a luz na posição definida por `p`, com a cor branca.

```
tLight(const t3DVector& p, const tColor& c);
```
Inicializa a luz na posição definida por `p`, com a cor `c`.

Métodos Públicos

GetColor
```
inline const tColor& GetColor() const;
```
Retorna uma referência para a cor de **this**.

GetPosition
```
inline const t3DVector& GetPosition() const;
```
Retorna uma referência para a posição de **this**.

SetColor
```
void SetColor(const tColor& c);
```
Ajusta a cor de **this** com `c`.

SetPosition
```
void SetPosition(const t3DVector& p);
```
Ajusta a posição de **this** com as coordenadas de `p`.

Classe tLine — gmodel.h

Um objeto da classe `tLine` é um primitivo gráfico definido por dois vértices, conforme visto no Capítulo 3. Utilizamos objetos da classe `tLine` para representar as isolinhas extraídas de um modelo mecânico por um filtro da classe `tContourFilter`. A classe deriva da classe `tPrimitive`.

Veja também Classes `tPrimitive`, `tContourFilter`

Atributos Públicos

V0
```
tVertex V0;
```
Primeiro vértice de **this**.

V1
```
tVertex V1;
```
Segundo vértice de **this**.

Construtores Públicos

Construtores
```
inline tLine(tGraphicModel* parent);
```
Inicializa o objeto base `tPrimitive` com `parent`.

Veja também Classe `tGraphicModel`

```
inline tLine(tGraphicModel* parent,
  const tVertex& v0, const tVertex& v1);
```
Inicializa o objeto base `tPrimitive` com `parent` e os vértices `V0` e `V1` com `v0` e `v1`, respectivamente.

Veja também Classe `tGraphicModel`

Método Público

Transform
```
void Transform(const t3DTransfMatrix& m);
```
Transforma os vértices de **this** com a transformação `m`.

Classe tLinearSystem — linearsystem.h

A classe abstrata `tLinearSystem` representa um sistema genérico de equações lineares. A interface da classe define métodos virtuais para resolução do sistema e montagem da matriz de coeficientes e do vetor de termos independentes.

Veja também Classe `tMatrix`, classe `tVector`

Métodos Públicos

Assemble
```
virtual void Assemble(const tMatrix& m, tLocationArray* l) = 0;
```
Adiciona a matriz `m` à matriz de coeficientes de **this**, de acordo com os números de equações contidas em `*l`.

```
virtual void Assemble(const tVector& v, tLocationArray* l) = 0;
```
Adiciona o vetor `v` ao vetor de termos independentes de **this**, de acordo com os números de equações contidas em `*l`.

GetDimension
```
int GetDimension() const;
```
Retorna a dimensão de **this**.

Solve
```
virtual void Solve() = 0;
```
Método virtual puro de solução de **this**. Deve ser sobrecarregado nas classes derivadas.

X
```
double X(int i) const;
```
Retorna o `i`-ésimo elemento do vetor solução de **this**, após a execução do método `Solve()`.

Veja também `Solve`

Construtores Protegidos

Construtores
`tLinearSystem(int m, int n);`
Constrói a matriz de coeficientes de **this** com `m` linhas e `n` colunas. Constrói o vetor de termos independentes com `m` elementos.

`tLinearSystem(tMatrix& A, tVector& B);`
Constrói a matriz de coeficientes de **this** com uma cópia da matriz `A`. Constrói o vetor de termos independentes com uma cópia do vetor `B`.

Atributos Protegidos

A
`tMatrix A;`
Matriz de coeficientes de **this**.

B
`tMatrix B;`
Vetor de termos independentes de **this**. Após a execução do método `Solve()`, contém os elementos do vetor solução de **this**.

Veja também `Solve`

Classe tLocalSystem — 3dtransf.h

Um objeto da classe `tLocalSystem` representa o sistema Cartesiano de coordenadas locais de um objeto de dimensão topológica 2. Um sistema local é definido por um ponto de origem e por três versores linearmente independentes, cujas coordenadas são tomadas em relação ao sistema global.

Atributos Públicos

N
`t3DVector N;`
Coordenadas do versor correspondente ao eixo z de **this**.

O
`t3DVector O;`
Coordenadas da origem de **this**.

U
`t3DVector U;`
Coordenadas do versor correspondente ao eixo x de **this**.

V
`t3DVector V;`
Coordenadas do versor correspondente ao eixo y de **this**.

Construtores Públicos

Construtores
`inline tLocalSystem();`
Construtor *default*. Sem funcionalidade.

`tLocalSystem(const t3DVector& normal, const t3DVector& up);`
Inicializa a origem `O` do sistema local com $(0, 0, 0)$. Os versores `U`, `V` e `N` são inicializados com as coordenadas dos versores do VRC definido por `normal` e `up`, tal como visto no Capítulo 7.

Métodos Públicos

GlobalToLocal
`t3DVector GlobalToLocal(const t3DVector& p) const;`
Retorna o vetor que contém as coordenadas globais do ponto `p`, dado em coordenadas locais.

`inline t3DVector GlobalToLocal(double x, double y, double z);`
Retorna o vetor que contém as coordenadas globais do ponto em coordenadas locais `x`,`y`,`z`.

LocalToGlobal
`t3DVector LocalToGlobal(const t3DVector& p) const;`
Retorna o vetor que contém as coordenadas locais do ponto `p`, dado em coordenadas globais.

`inline t3DVector LocalToGlobal(double x, double y, double z);`
Retorna o vetor que contém as coordenadas locais do ponto em coordenadas globais `x`,`y`,`z`.

Classe tLookupTable — looktab.h

Um objeto da classe `tLookupTable` representa uma tabela de cores utilizada por uma objeto da classe `tMapper`. A interface da classe possui métodos para ajuste da matiz, saturação e valor (modelo HSV) das cores representadas na tabela. A classe `tLookupTable` é derivada da classe `tObjectBody`.

Veja também Classe `tObjectBody`

Construtor e Destrutor Públicos

Construtor
`tLookupTable(int n = 256);`
Construtor *default*. Constrói a tabela com 256 cores, variando do vermelho até o azul.

Destrutor
`~tLookupTable();`
Destrutor de **this**.

Métodos Públicos

GetMinHue
```
inline double GetMinHue() const;
```
Retorna o menor valor de matiz das cores da tabela.

GetMinSaturation
```
inline double GetMinSaturation() const;
```
Retorna o menor valor de saturação das cores da tabela.

GetMinScalar
```
inline double GetMinScalar() const;
```
Retorna o menor valor escalar da tabela, usado no mapeamento de cores.

GetMinValue
```
inline double GetMinValue() const;
```
Retorna o menor valor do atributo valor das cores da tabela.

GetMaxHue
```
inline double GetMaxHue() const;
```
Retorna o maior valor de matiz das cores da tabela.

GetMaxSaturation
```
inline double GetMaxSaturation() const;
```
Retorna o maior valor de saturação das cores da tabela.

GetMaxScalar
```
inline double GetMaxScalar() const;
```
Retorna o maior valor escalar da tabela, usado no mapeamento de cores.

GetMaxValue
```
inline double GetMaxValue() const;
```
Retorna o maior valor do atributo valor das cores da tabela.

GetNumberOfColors
```
inline int GetNumberOfColors() const;
```
Retorna o número de cores da tabela.

MapColor
```
const tColor& MapColor(double scalar) const;
```
Mapeamento de cores. Retorna a cor RGB associada ao escalar `scalar`.

SetHueRange
```
void SetHueRange(double min, double max);
```
Ajusta o intervalo de matiz das cores da tabela.

SetSaturationRange
```
void SetSaturationRange(double min, double max);
```
Ajusta o intervalo de saturação das cores da tabela.

SetScalarRange
```
void SetScalarRange(double min, double max);
```
Ajusta o intervalo de escalares da tabela, usados no mapeamento de cores.

SetValueRange
```
void SetValueRange(double min, double max);
```
Ajusta o intervalo de valor das cores da tabela.

Operadores Públicos

operator []
```
inline const tColor& operator [](int i) const;
```
Retorna um *rvalue* correspondente a `i`-ésima cor da tabela. Um *rvalue* pode aparecer somente no lado direito de uma expressão de atribuição.

```
inline tColor& operator [](int i);
```
Retorna um *lvalue* correspondente a `i`-ésima cor da tabela. Um *lvalue* pode aparecer no lado esquerdo de uma expressão de atribuição.

Classe tMainWindow — oswwin.h

Um objeto da classe `tMainWindow` representa a janela principal de uma aplicação de modelagem OSW, como visto no Capítulo 9. A classe deriva diretamente da classe `TDecoratedFrameWindow`.

Veja também Classe `TDecoratedFrameWindow` (OWL)

Construtor Público

Construtor
```
tMainWindow(const char* title, TResId menuId);
```
Inicializa o objeto base `TDecoratedFrameWindow` com os parâmetros `title` e `menuId`. Constrói a barra de *status* e a janela de comando da aplicação. A janela principal é automaticamente construída pelo objeto de aplicação.

Veja também Classe `tApplication`

Classe tMapper — mapper.h

Um objeto da classe `tMapper` representa um processo sumidouro de dados em um diagrama de fluxo de dados, conforme definido no Capítulo 7. Um mapeador mantém uma referência para um objeto da classe `tLookupTable`, a tabela de cores utilizada pelo mapeador para definição das cores dos vértices do modelo de entrada do processo. A classe `tMapper` é derivada da classe `tObjectBody`.

Veja também Classe `tLookupTable`, classe `tObjectBody`

Construtor e Destrutor Públicos

Construtor
```
tMapper(tLookupTable* table = 0);
```
Construtor `default`. Se o `table` for igual a zero, **this** utiliza uma tabela de cores padrão da classe `tMapper`

Destrutor
`~tMapper()();`
Destrói a tabela de cores e o modelo mapeado por **this**, se houver. Os objetos só serão realmente destruídos se seu contador de referência for igual a zero.

Métodos Públicos

GetInput
```
inline tModel* GetInput();
```
Retorna um ponteiro para o modelo mapeado por **this**.

SetInput
```
void SetInput(tModel* model);
```
Destrói o modelo mapeado por **this**, se houver, e torna `model` o novo modelo de **this**.

SetLookupTable
```
void SetLookupTable(tLookupTable* table);
```
Destrói a tabela de cores de **this** e torna `table` a nova tabela de cores. A tabela de cores só é realmente destruída se seu contador de referência for igual a zero.

SetScalarRange
```
void SetScalarRange(double min, double max);
```
Ajusta os valores mínimo e máximo a partir dos quais os escalares dos vértices do modelo de entrada de **this** serão mapeados em cores.

UseScalars
```
void UseScalars(bool use);
```
Se `use` for igual a **true**, **this** determina a cor de um vértice do modelo de entrada, se houver, a partir do valor escalar do vértice e da tabela de cores de **this**. Se `use` for **false**, a cor é determinada pelas propriedades da superfície do modelo de entrada.

Render
```
virtual void Render(tRenderer& renderer);
```
Início do processo de mapeamento do modelo de entrada de **this**. `renderer` é o objeto responsável pela geração da imagem da cena.

Veja também Classe `tRenderer`

Atributos Protegidos

Input
```
tModel* Input;
```
Modelo de entrada de **this**.

tLookupTable*
```
tLookupTable* Table;
```
Tabela de cores de **this**.

Classe tMaterial — material.h

Um objeto da classe `tMaterial` é um manipulador de propriedades do material de um modelo. As propriedades de um material incluem as características mecânicas, tais como módulo de elasticidade e coeficiente de Poisson, e as constantes utilizadas no

modelo de iluminação definido no Capítulo 7, tais como cor e coeficiente de reflexão difusa do objeto. Os objetos da classe `tMaterial` contém um ponteiro para um objeto da classe `tMaterialBody`, derivada de `tObjectBody`.

Veja também Classe `tMatrix`

Enumeração Pública

tProperty
```
enum tProperty;
```
Enumera as propriedades de um material: `E` (módulo de elasticidade), `G` (módulo de elasticidade transversal), `Poisson` (coeficiente de Poisson), `Od` (cor de reflexão difusa), `Kd` (coeficiente de reflexão difusa), `Oa` (cor de luz ambiente), `Ka` (coeficiente de reflexão difusa da luz ambiente).

Construtores e Destrutor Público

Construtores
```
tMaterial();
```
Construtor de material *default*.

```
tMaterial(const tMaterial& m, int cpType = tObject::ShallowCopy);
```
Construtor de cópia. Torna **this** uma cópia de `m`. O tipo de cópia é especificado pelo parâmetro `cpType`: se `cpType` for igual a `tObject::DeepCopy`, um novo corpo de material é construído e seus elementos são inicializados com os valores dos elementos de `m`; se `cpType` for igual a `tObject::ShallowCopy`, um novo corpo de vetor não é criado; nesse caso, o corpo de `m` é compartilhado.

Destrutor
```
~tMaterial();
```
Se o corpo do material for utilizado somente por **this**, destrói o corpo do material.

Métodos Públicos

GetProperty
```
inline double GetProperty(tProperty p) const;
```
Retorna o valor da propriedade `p` de this.

SetProperty
```
inline void SetProperty(tProperty p, double value);
```
Ajusta o valor da propriedade `p` de **this** com o valor `value`.

Operadores Públicos

operator =
```
tMaterial operator =(const tMaterial m);
```
Operador de cópia (*shallow-copy*). Torna **this** uma cópia de `m` e retorna **this**.

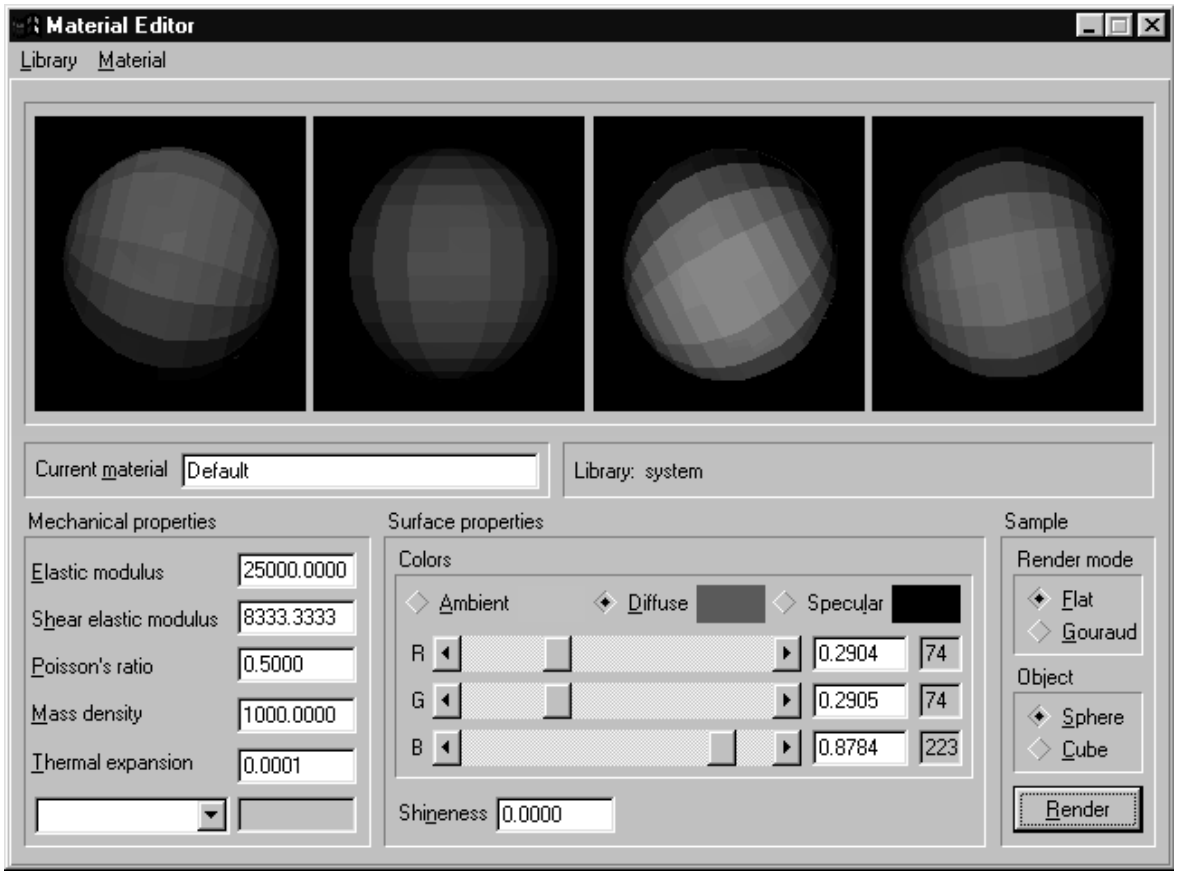
operator ==
```
bool operator ==(const tMaterial m) const;
```
Operador de comparação (*shallow-equal*). Retorna **true** se **this** e `m` possuírem o mesmo corpo de material; retorna **false** caso contrário.

operator !=
```
bool operator !=(const tMaterial m) const;
```
Retorna **true** se **this** e `m` possuírem corpos de material diferentes; retorna **false** caso contrário.

Classe tMaterialEditor — materdlg.h

Um objeto da classe tMaterialEditor é uma caixa de diálogo que permite a especificação das propriedades mecânicas e de superfície de determinado material, vistas, respectivamente, nos Capítulos 6 e 7. Uma vez definidas as propriedades de um material, podemos armazenar o material em *bibliotecas de materiais* e, posteriormente, recuperar seus dados para uso em outras aplicações de modelagem. A classe é derivada da classe `TDialog`.

Veja também Classe `TDialog` (OWL)



Construtor e Destrutor Públicos

Construtor
```
tMaterialEditor(TWindow* parent);
```
Inicializa o objeto base `TDialog` com `parent` e constrói os controles de edição das propriedades de **this**. A construção de um editor de materiais é efetuada automaticamente pela aplicação OSW.

Veja também Classe `tApplication`

Destrutor
`~tMaterialEditor();`
Destrói os objetos privados de **this**.

Classe tMatrix **matrix.h**

Um objeto da classe `tMatrix` é um manipulador de matrizes de números reais. Em OSW, uma matriz pode ser compartilhada por diversos objetos ou por diversas funções ao mesmo tempo. Cada objeto ou função que compartilha a matriz não possui uma cópia distinta da matriz; ao invés disso, o objeto ou função mantém um *manipulador* que lhe garante acesso aos dados e às operações efetuadas com a matriz. Esse manipulador é um objeto da classe `tMatrix`. A matriz, propriamente, é definida por um objeto da classe `tMatrixBody`, o *corpo* da matriz, a qual é derivada da classe `tObjectBody`.

Exemplo

```
tMatrix f()
{
  tMatrix a(100, 100); // a é matriz com 10000 elementos
  tMatrix b = a;

  for (int i = 0; i < 100; i++)
    a[i,i] = i * i;
  tMatrix c = a.Inverse() * b;
  return c;
}
```

A função `f` retorna a matriz resultante do produto da matriz `a` com a inversa da matriz `b`. Mas, ao construirmos a matriz `b` como uma cópia de `a`, não estamos criando uma nova cópia de todos os 10000 elementos de `a`; na verdade, estamos somente afirmando que `b` e `a` são iguais. De fato, ao alterarmos os elementos da diagonal de `a`, também o estamos fazendo em `b` e, portanto, a matriz `c` é a matriz identidade.

Ao término da função, a matriz `c` é retornada. Isso significa que um objeto temporário do tipo `tMatrix` é criado e o construtor de cópia de `tMatrix` é chamado para inicializar o temporário a partir do objeto `c`. Finalmente, o destrutor para `c` é invocado. Se o corpo das matrizes não fossem compartilhados, essas operações envolveriam, nesse exemplo, a transferência de 10000 objetos do tipo **`double`**. Se uma cópia não-compartilhada de uma matriz for necessária, podemos fazer:

```
tMatrix d(100, 100);
tMatrix e(a, tObject::DeepCopy);
```

A matriz `e` é uma cópia *profunda* de `a`, ou seja, possui 10000 elementos idênticos aos elementos de `a`, mas `d` e `a` *não* são as mesmas matrizes. Se alterarmos algum elemento de `d`, por exemplo, a modificação não afetará a matriz `a`.

Construtores e Destrutor Públicos

Construtores
`tMatrix(uint m, uint n);`
Constrói um corpo de matriz com `m` linhas e `n` colunas e inicializa todos os elementos com zero.

`tMatrix(const tMatrix m, int cpType = tObject::ShallowCopy);`
Construtor de cópia. O tipo de cópia é especificado pelo parâmetro `cpType`: se `cpType` for igual a `tObject::DeepCopy`, um novo corpo de matriz é construído e seus elementos são inicializados com os valores dos elementos de `m`; se `cpType` for igual a `tObject::ShallowCopy`, um novo corpo de matriz não é criado; nesse caso, o corpo de `m` é compartilhado.

Destrutor
`~tMatrix();`
Se o corpo da matriz for utilizado somente por **this**, destrói o corpo da matriz.

Veja também Classe `tObjectBody`

Métodos Públicos

Copy
`tMatrix Copy(const tMatrix m, int cpType = tObject::ShallowCopy);`
Cópia de matrizes. Torna **this** uma cópia de `m` e retorna **this**. O tipo de cópia é especificado pelo parâmetro `cpType`: se `cpType` for igual a `tObject::DeepCopy`, um novo corpo de matriz é construído e seus elementos são inicializados com os valores dos elementos de `m`; se `cpType` for igual a `tObject::ShallowCopy`, um novo corpo de matriz não é criado; nesse caso, o corpo de `m` é compartilhado.

GetNumberOfColumns
`inline int GetNumberOfColumns() const;`
Retorna o número de colunas da matriz.

GetNumberOfRows
`inline int GetNumberOfRows() const;`
Retorna o número de linhas da matriz.

IsEqual
`bool IsEqual(const tMatrix m, int eqType = tObject::ShallowEqual);`
Comparação de matrizes. Retorna **true** se **this** e `m` são iguais; retorna **false** caso contrário. O tipo de identidade é especificado pelo parâmetro `eqType`: se `eqType` for igual a `tObject::ShallowEqual`, as matrizes são iguais se seus corpos forem os mesmos; se `eqType` for igual a `tObject::DeepEqual`, as matrizes são iguais se os elementos de seus corpos forem iguais.

IsOrthogonal
`bool IsOrthogonal() const;`
Retorna **true** se a matriz for ortogonal; retorna **false** caso contrário.

IsSimmetrical
`bool IsSimmetrical() const;`
Retorna **true** se a matriz for simétrica; retorna **false** caso contrário.

IsSquare
`inline bool IsSquare() const;`
Retorna **true** se a matriz for quadrada; retorna **false** caso contrário.

Operadores Públicos

operator =
`tMatrix operator =(const tMatrix m);`
Operador de cópia (*shallow-copy*). Torna **this** uma cópia de `m` e retorna **this**.

Veja também `tMatrix::Copy`

operator +
`tMatrix operator +(const tMatrix m) const;`
Retorna a matriz resultante da soma de **this** com `m`. Se **this** e `m` não possuírem a mesma ordem, uma exceção da classe `tXMath` é gerada.

Veja também Classe `tXMath`

operator –
`tMatrix operator -(const tMatrix m) const;`
Retorna a matriz resultante da subtração de **this** com `m`. Se **this** e `m` não possuírem a mesma ordem, uma exceção da classe `tXMath` é gerada.

Veja também Classe `tXMath`

operator *
`tMatrix operator *(const tMatrix m) const;`
Retorna a matriz resultante da multiplicação de **this** com a matriz `m`. Se o número de colunas de **this** for diferente do número de linhas de `m`, uma exceção da classe `tXMath` é gerada.

Veja também Classe `tXMath`

`tVector operator *(const tVector v) const;`
Retorna o vetor resultante da multiplicação de **this** com o vetor `v`. Se o número de colunas de **this** for diferente do número de elementos de `v`, uma exceção da classe `tXMath` é gerada.

Veja também Classe `tVector`, classe `tXMath`

`tMatrix operator *(double s) const;`
Retorna a matriz resultante da multiplicação de **this** com o escalar `s`.

operator +=
`tMatrix operator +=(const tMatrix m);`
Soma **this** com a matriz `m`, armazena o resultado em **this** e retorna **this**. Se **this** e `m` não possuírem a mesma ordem, uma exceção da classe `tXMath` é gerada.

Veja também Classe `tXMath`

operator -=
`tMatrix operator -=(const tMatrix m);`
Subtrai a matriz `m` de **this**, armazena o resultado em **this** e retorna **this**. Se **this** e `m` não possuírem a mesma ordem, uma exceção da classe `tXMath` é gerada.

Veja também **operator -**

operator *=
```
tMatrix operator *=(double s);
```
Multiplica **this** pelo escalar `s`, armazena o resultado em **this** e retorna **this**.

Veja também `operator *`

operator ==
```
bool operator ==(const tMatrix m) const;
```
Operador de comparação (*shallow-equal*). Retorna **true** se **this** e `m` possuírem o mesmo corpo de matriz; retorna **false** caso contrário.

Veja também `tMatrix::IsEqual`

operator !=
```
bool operator !=(const tMatrix m) const;
```
Retorna **true** se **this** e `m` possuírem corpos de matriz diferentes; retorna **false** caso contrário.

Veja também `tVector::IsEqual`

operator ()
```
double operator ()(int i, int j) const;
```
Retorna, como um *rvalue*, o elemento da linha `i` e da linha `j` de **this**. Um *rvalue* é um valor que pode aparecer somente no lado direito de uma expressão de atribuição.

```
double& operator ()(int i, int j);
```
Retorna, como um *lvalue*, o elemento da linha `i` e da linha `j` de **this**. Um *lvalue* é um valor que pode aparecer no lado esquerdo de uma expressão de atribuição.

Classe tMesh — mesh.h

Um objeto da classe `tMesh` é um modelo de decomposição por células genérico, definido por uma coleção de vértices, ou nós, e por uma coleção de células. A interface da classe sobrecarrega os métodos virtuais da classe base `tModel` responsáveis pelo fornecimento de *iterators* de vértices, arestas e faces do modelo. Oferece, ainda, um método que permite o acesso de outros objetos às células do modelo.

Veja também Classe `tModel`

Construtor e Destrutor Públicos

Construtor
```
tMesh();
```
Construtor *default*. Inicializa a classe base `tModel`.

Destrutor
```
~tMesh();
```
Destrói os elementos da lista de células e de nós de **this**.

Métodos Públicos

GetCellIterator
```
tCellIterator GetCellIterator();
```
Retorna um *iterator* para a lista de células de **this**.

Veja também Classe `tCellIterator`

GetEdgeIterator
`tEdgeIterator GetEdgeIterator();`
Retorna um *iterator* para a coleção de arestas de **this**.

GetFaceIterator
`tFaceIterator GetFaceIterator();`
Retorna um *iterator* para a coleção de faces de **this**.

GetMaterialAt
`tMaterial GetMaterialAt(const t3DVector& p) const;`
Retorna o material da superfície de **this** no ponto `p`.

Veja também Classe `tMaterial`

GetNumberOfCells
`int GetNumberOfCells() const;`
Retorna o número de células de **this**.

GetNumberOfVertices
`int GetNumberOfVertices() const;`
Retorna o número de vértices de **this**.

GetVertexIterator
`tVertexIterator GetVertexIterator();`
Retorna um *iterator* para a coleção de vértices de **this**.

Transform
`void Transform(const t3DTransfMatrix& m);`
Transforma a coleção de vértices de **this**, usando a matriz de transformação geométrica `m`.

Veja também Classe `t3DTransfMatrix`

Classe tMeshableModel — model.h

Um objeto da classe `tMeshableModel`, derivada de `tNameableModel`, representa um modelo geométrico genérico de entrada de um filtro de uma classe derivada da classe abstrata `t2DMeshGenerator`. A classe define um método virtual responsável pela divisão das arestas do modelo. O método é executado pelo filtro durante o processo de geração de malhas.

Veja também Classes `t2DMeshGenerator`, `tNameableModel`

Construtores Públicos

Construtores
`inline tMeshableModel();`
Construtor *default*. Sem funcionalidade.

`inline tMeshableModel(const char* name);`
Inicializa o objeto base `tNameableModel` com o parâmetro `name`.

Método Público

SplitEdges
```
virtual void SplitEdges(double size);
```
Divide as arestas de **this** em arestas do tamanho `size`.

Classe tMeshReader — meshread.h

A classe abstrata `tMeshReader`, derivada de `tPolyReader`, descreve a estrutura e funcionalidade de um leitor OSW de modelos de decomposição por células (vimos o conceito de *leitores* no Capítulo 7). A classe declara métodos virtuais puros de construção dos nós e das células de um modelo de decomposição por células.

Veja também Classe `tPolyReader`

Construtor Público

Construtor
```
inline tMeshReader(const char* fileName);
```
Inicializa o objeto base `tPolyReader` com o parâmetro `fileName`.

Método Público

GetOutput
```
inline tMesh* GetOutput();
```
Retorna um ponteiro para o modelo de saída de **this**.

Métodos Protegidos

MakeCell
```
virtual tCell* MakeCell(const char* clsName) = 0;
```
Método virtual puro de construção das células do modelo de saída de **this**. Constrói uma célula da classe cujo nome é dado por `clsName` e retorna um ponteiro para a célula. Deve ser sobrecarregado nas classes concretas derivadas de `tMeshReader`.

Veja também Classe `tCell`

MakeMesh
```
virtual tMesh* MakeMesh(const char*);
```
Constrói uma malha da classe `tMesh` e retorna um ponteiro para a malha. Pode ser sobrecarregado nas classes derivadas de `tMeshReader`.

Veja também Classe `tMesh`

MakeNode
```
virtual tNode* MakeNode(const char* clsName) = 0;
```
Método virtual puro de construção dos nós do modelo de saída de **this**. Constrói uma nó da classe cujo nome é dado por `clsName` e retorna um ponteiro para o nó. Deve ser sobrecarregado nas classes concretas derivadas de `tMeshReader`.

Veja também Classe `tNode`

Classe tMeshView meshview.h

Um objeto da classe `tMeshView` representa uma vista OSW na qual podemos efetuar a seleção múltipla ou simples de nós ou células de um modelo de decomposição por células. (A classe `tMecView`, descrita no Capítulo 11, é um exemplo de classe derivada de `tMeshView`. Os métodos de seleção de nós e células, herdados de `tMeshView`, são usados na implementação dos comandos de especificação dos vínculos dos carregamentos da estrutura.) `tMeshView` é derivada virtualmente de `tView`.

Veja também Classe `tView`

Construtor Público

Construtor
```
inline tMeshView(tScene& scene, tCamera* camera);
```
Inicializa o objeto base `tView` com os parâmetros `scene` e `camera`.

Métodos Protegidos

PaintFixedNodes
```
virtual void PaintFixedNodes(TDC& dc, tNodePickList& list);
```
Método virtual de desenho, no dispositivo de contexto `dc`, dos ícones correspondentes a um conjunto de nós selecionados dado em `list`. Assume-se que os nós foram selecionados para aplicação de condições de contorno.

PaintLoads
```
virtual void PaintLoads(TDC& dc, tCellPickList& list);
```
Método virtual de desenho, no dispositivo de contexto `dc`, dos ícones correspondentes a um conjunto de células selecionadas dado em `list`. Assume-se que as células foram selecionados para aplicação de carregamentos.

```
virtual void PaintLoads(TDC& dc, tNodePickList& list);
```
Método virtual de desenho, no dispositivo de contexto `dc`, dos ícones correspondentes a um conjunto de nós selecionados dado em `list`. Assume-se que os nós foram selecionados para aplicação de carregamentos.

SelectCells
```
bool SelectCells(tMesh& mesh, tCellPickInfo& info, const CHAR* msg);
```
Método de seleção múltipla de células da malha `mesh`.

SelectNodes
```
bool SelectNodes(tMesh& mesh, tNodePickInfo& info, const CHAR* msg);
```
Método de seleção múltipla de nós da malha `mesh`.

SelectSingleCell
```
tCell* SelectSingleCell(tMesh& mesh, const CHAR* msg);
```
Método de seleção simples de uma célula da malha `mesh`. Retorna um ponteiro para a célula selecionada, ou 0, se célula alguma foi selecionada.

SelectSingleNode
```
tNode* SelectSingleNode(tMesh& mesh, const CHAR* msg);
```
Método de seleção simples de um nó da malha `mesh`. Retorna um ponteiro para o nó selecionado, ou 0, se nó algum foi selecionado.

Classe tMeshWriter — meshwrit.h

Um objeto da classe `tMeshWriter` é um escritor de modelos de decomposição por células definidos no Capítulo 3 (vimos o conceito de *escritor* no Capítulo 7). O modelo de entrada do escritor é um objeto de uma classe derivada da classe `tMesh`. A classe é derivada da classe abstrata `tWriter`. O método de escrita do modelo de entrada do escritor é implementado pelo método privado `Run()`, declarado como virtual puro na classe base `tWriter`.

Veja também Classes `tMesh`, `tWriter`

Construtor Público

Construtor
```
inline MeshWriter(const char* fileName);
```
Inicializa o objeto base `tWriter` com `fileName`.

Métodos Públicos

GetInput
```
inline tMesh* GetInput();
```
Retorna um ponteiro para o modelo de entrada de **this**.

SetInput
```
void SetInput(tMesh* mesh);
```
Torna **this** o modelo de entrada de **this**.

Atributo Protegido

Mesh
```
tMesh* Mesh;
```
Ponteiro para o modelo de entrada de **this**.

Classe tModel — model.h

A classe `tModel` é uma classe abstrata a partir da qual todas as outras classes de modelos de OSW são derivadas. A interface de `tModel` oferece métodos virtuais utilizados por outros objetos para acesso às coleções de vértices, arestas e faces de um modelo, como discutido no Capítulo 9. As classes derivadas de `tModel` devem sobrecarregar esses métodos e implementar *iterators* próprios de vértices, arestas e faces. `tModel` é derivada de `tObjectBody` e de `TStreamableBase`.

Veja também Classe `tObjectBody`, classe `TStreamableBase` (classlib)

Construtor e Destrutor Públicos

Construtor
```
tModel();
```
Construtor *default*. Inicializa a classe base `tObjectBody`.

Destrutor
```
virtual ~tModel();
```
Destrutor virtual.

Métodos Públicos

GetEdgeIterator
```
virtual tEdgeIterator GetEdgeIterator() = 0;
```
Retorna um *iterator* para a coleção de arestas de **this**.

Veja também Classe `tEdgeIterator`

GetFaceIterator
```
virtual tFaceIterator GetFaceIterator() = 0;
```
Retorna um *iterator* para a coleção de faces de **this**.

Veja também Classe `tFaceIterator`

GetMaterialAt
```
virtual tMaterial GetMaterialAt(const t3DVector& p) const;
```
Retorna o material da superfície de **this** no ponto `p`. Em `tModel`, o método retorna o material *default*.

Veja também Classe `tMaterial`

GetNumberOfEdges
```
virtual int GetNumberOfEdges() const;
```
Retorna o número de arestas de **this**. Em `tModel`, o método retorna 0.

GetNumberOfFaces
```
virtual int GetNumberOfFaces() const;
```
Retorna o número de faces de **this**. Em `tModel`, o método retorna 0.

GetNumberOfVertices
```
virtual int GetNumberOfVertices() const;
```
Retorna o número de vértices de **this**. Em `tModel`, o método retorna 0.

GetVertexIterator
```
virtual tVertexIterator GetVertexIterator() = 0;
```
Retorna um *iterator* para a coleção de vértices de **this**.

Veja também Classe `tVertexIterator`

Intersect
```
virtual bool Intersect(const tRay& r, tIntersectInfo& i) const;
```
Calcula as informações de intersecção de **this** com o raio `r`. Essas informações, armazenadas no objeto `i`, são as coordenadas do ponto de intersecção, a normal no ponto de intersecção e o material de **this** no ponto de intersecção. Se o raio `r` interceptar o objeto, retorna **true**; caso contrário, retorna **false**. Em `tModel`, o método retorna **false**.

Veja também Classe `tRay`, classe `tIntersectInfo`

Transform
```
virtual void Transform(const t3DTransfMatrix& m);
```
Transforma a coleção de vértices de **this**, usando a matriz de transformação geométrica `m`.

Veja também Classe `t3DTransfMatrix`

Classe tNameableModel — model.h

Um objeto da classe `tNameableModel` é um modelo genérico para o qual podemos atribuir um nome. A classe é derivada de `tNameableObject` e derivada virtualmente de `tModel`.

Veja também Classes `tNameableObject`, `tModel`

Construtores Públicos

Construtores
```
inline tNameableModel();
```
Costrutor *default*. Sem funcionalidade.

```
inline tNameableModel(const char* name);
```
Inicializa o objeto base `tNameableObject` com o parâmetro `name`.

Classe tNameableObject — obody.h

Um objeto da classe `tNameableObject` é um corpo de objeto OSW para o qual podemos atribuir um nome. A classe é derivada diretamente de `tObjectBody`.

Veja também Classe `tObjectBody`

Construtores e Destrutor Públicos

Construtores
```
inline tNameableObject();
```
Inicializa o nome do objeto com uma cadeia de caracteres nula.

```
tNameableObject(const char* name);
```
Inicializa o nome do objeto com uma cópia da cadeia de caracteres `name`.

Destrutor
```
~tNameableObject();
```
Libera a memória utilizada para o armazenamento da cadeia de caracteres que contém o nome de **this**.

Métodos Públicos

GetName
```
inline const char* GetName() const;
```
Retorna um ponteiro para a cadeia de caracteres que contém o nome de **this**.

SetName
```
void SetName(const char* name);
```
Torna uma cópia de `name` a cadeia de caracteres que contém o nome de **this**.

Classe tNode — node.h

A classe `tNode`, derivada de `tVertex`, representa um nó genérico de um modelo de decomposição por células, como visto no Capítulo 3. Um nó é responsável pelo posicionamento espacial do modelo e pela manutenção de seus graus de liberdade.

Construtor Público

Construtor
`tNode(int n);`
Constrói o nó com `n` graus de liberdade.

Métodos Públicos

GetNumberOfDOFs
`int GetNumberOfDOFs() const`
Retorna o número de graus de liberadade associados ao nó.

GetDOF
`inline const tDOF& GetDOF(int i) const;`
Retorna uma referência do tipo *rvalue* para o `i`-ésimo grau de liberdade associado ao nó. Um *rvalue* pode aparecer somente no lado direito de uma operação de atribuição.

Veja também Classe `tDOF`

GetScalar
`inline double GetScalar() const;`
Retorna o atributo escalar associado ao nó. Esse atributo é utilizado no mapeamento de cores do modelo e na geração de isopontos, isolinhas ou isosuperfícies.

GetVector
`inline const t3DVector& GetVector() const;`
Retorna o atributo vetorial associado ao nó. Esse atributo é utilizado na determinação da estrutura deformada do modelo.

SetDOF
`inline tDOF& SetDOF(int i) const;`
Retorna uma referência do tipo *lvalue* para o `i`-ésimo grau da liberdade associado ao nó. Um *lvalue* aparece no lado esquerdo de uma expressão de atribuição. O método é utilizado, portanto, para atribuir um grau de liberdade para o nó.

Veja também Classe `tDOF`

SetScalar
`inline SetScalar(double s);`
Ajusta o atributo escalar de **this** com o escalar `s`.

SetVector
`inline SetVector(const t3DVector& v);`
Ajusta o atributo escalar de **this** com o vetor `v`.

Atributos Protegidos

DOFs
`tFixedArray<tDOF*> DOFs;`
Vetor de ponteiros para os graus de liberdade do nó.

Next
`tNode* Next;`
Ponteiro para o próximo nó da lista de nós do modelo mecânico de **this**.

NodeUses
`tNodeUse* NodeUses;`
Ponteiro para a lista de usos de nó de **this**.

Previous
`tNode* Next;`
Ponteiro para o nó anterior da lista de nós do modelo mecânico de **this**.

Scalar
`double Scalar;`
Atributo escalar de **this**.

Vector
`t3DVector Vector;`
Atributo vetorial de **this**.

Métodos Protegidos

MakeNodeUse
`void MakeNodeUse(tCell* cell);`
Adiciona à lista de usos do nó um novo elemento que indica que a célula `cell` usa o nó. Como visto no Capítulo 3, um nó de um modelo de decomposição por células mantém uma lista de elementos que indicam quais são as células que incidem no nó. O método é utilizado pela classe `tCell` e, em geral, não precisa ser invocado diretamente.

Veja também Classe `tCell`, `tNode::KillNodeUse`

KillNodeUse
`void KillNodeUse(tCell* cell);`
Remove da lista de usos do nó o elemento que indica que a célula `cell` usa o nó, se esse elemento existir. O método é utilizado pelo destrutor da classe `tCell` e, em geral, não precisa ser invocado diretamente.

Veja também Classe `tCell`, `tNode::MakeNodeUse`

Classe tNodeT — node.h

`template <int n> class tNodeT;`
A classe paramétrica `tNodeT`, derivada da classe `tNode`, representa um nó de um modelo de decomposição por células com `n` graus de liberdade.

Veja também Classe `tNode`

Construtor Público

tNodeT
`tNodeT();`
Construtor *default* do nó.

Classe tNodeUse — mesh.h

Um objeto da classe `tNodeUse` representa um uso de nó de um modelo de decomposição por células, como visto no Capítulo 3.

Atributo Público

Cell
`tCell* Cell;`
Ponteiro para a célula que usa **this**.

Veja também Classe `tCell`

Construtor Público

Construtor
`inline tNodeUse(tCell* cell);`
Inicializa `Cell` com o parâmetro `cell`.

Classe tObjectBody — obody.h

Um objeto da classe `tObjectBody` representa um corpo de objeto OSW. Um corpo de objeto é um objeto que pode ser compartilhado por outros objetos. Um corpo de objeto mantém um *contador de referência* que indica o número de objetos que estão compartilhando o corpo. Quando um objeto compartilha um corpo de objeto, o contador de referência do corpo deve ser incrementado. Quando um objeto não necessita mais compartilhar um corpo de objeto, o contador de referência do corpo deve ser decrementado. Um corpo de objeto somente deve ser destruído quando seu contador de referência for igual a zero. Classes de OSW tais como `tMatrix`, `tVector` e `tMaterial` *possuem* corpos de objetos; outras classes tais como `tModel` e `tLookupTable` *são* corpos de objetos.

Operadores Públicos

operator ++
`inline int operator ++();`
Pré-incremento. Incrementa o contador de referência de **this** e retorna o valor do contador de referência *após* o incremento.

`inline int operator ++(int);`
Pós-incremento. Incrementa o contador de referência de **this** e retorna o valor do contador de referência *antes* do incremento.

operator −−
`inline int operator --();`
Pré-decremento. Decrementa o contador de referência de **this** e retorna o valor do contador de referência *após* o incremento.

`inline int operator --(int);`
Pós-decremento. Decrementa o contador de referência de **this** e retorna o valor do contador de referência *antes* do incremento.

Construtor Protegido

Construtor
`inline tObjectBody();`
Inicializa o contador de referência de **this** com 1.

Classe tPickableObject dfile.h

A classe abstrata `tPickableObject`, derivada de `tNameableObject`, descreve a funcionalidade dos objetos OSW que podem ser selecionados em uma janela de vista de uma cena. A classe declara um método virtual puro responsável pela criação do objeto DC associado. A classe `tLight`, por exemplo, é derivada de `tPickableObject`.

Veja também Classes `tNameableObject`, `tDCObjectDC`

Construtores Públicos

Construtores
```
inline tPickableObject();
```
Construtor *default*. Sem funcionalidade.

```
inline tPickableObject(const char* name);
```
Inicializa o objeto base `tNameableObject` com `name`.

Métodos Públicos

IsSelected
```
inline bool IsSelected() const;
```
Retorna **true** se **this** está selecionado; retorna **false** caso contrário.

MakeDCObject
```
virtual tDCObject* MakeDCObject(const tRenderer& r) const = 0;
```
Método virtual puro de construção do objeto DC associado a **this**. Deve ser sobrecarregado nas classes concretas derivadas de `tPickableObject`

Veja também Classe `tDCObject`

Select
```
inline void Select(bool selected = true);
```
Ajusta a *flag* de seleção de **this** com o parâmetro `selected`.

Atributo Protegido

Selected
```
bool Selected;
```
Flag de seleção de **this**.

Classe tPickInfo dfile.h

Um objeto da classe `tPickInfo` contém as informações genéricas de um processo de seleção múltipla de objetos em uma janela de vista.

Atributos Públicos

Duplicated
```
int Duplicated;
```
Número de objetos selecionados mais de uma vez no processo de seleção.

Found
`int Found;`
Número de objetos encontrados.

PickMode
`int PickMode;`
Modo de seleção. Para uso futuro.

Removed
`int Removed;`
Número de objetos removidos no processo de seleção. Para uso futuro.

Viewport
`tViewport Viewport;`
Retângulo de seleção.

Veja também Classe `tViewport`

Classe tPickInfoT — dfile.h

`template <class T> class tPickInfoT;`
A classe paramétrica `tPickInfoT`, derivada de `tPickInfo`, adiciona uma lista de seleção de objetos da classe `T` à estrutura da classe base. A lista mantém ponteiros para os objetos selecionados durante o processo de seleção.

Veja também Classe `tPickInfo`

Construtor Público

Construtor
`inline tPickInfoT(tPickList<T>& list);`
Inicializa a lista de seleção de **this** com `list`.

Veja também Classe paramétrica `tPickList`

Método Público

GetList
`inline tPickList<T>* GetList();`
Retorna um ponteiro para a lista de seleção de **this**.

Veja também Classe paramétrica `tPickList`

Classe tPickList — dfile.h

`template <class T> class tPickList;`
Um objeto da classe paramétrica `tPickList` representa uma lista de seleção de objetos da classe `T`.

Veja também Classe paramétrica `tPickInfoT`

Construtor e Destrutor Públicos

Construtor
```
inline tPickList();
```
Inicializa a lista de seleção como uma lista vazia.

Destrutor
```
inline ~tPickList();
```
Executa o método `Flush()`.

Veja também `Flush()`

Métodos Públicos

Flush
```
void Flush();
```
Esvazia **this**, destruindo todos os seus elementos.

GetNumberOfItems
```
inline int GetNumberOfItems() const;
```
Retorna o número de ítens selecionados de **this**.

Insert
```
bool Insert(T* object);
```
Adiciona `object` em **this**.

IsEmpty
```
inline bool IsEmpty() const;
```
Retorna **true** se há ítens selecionados em **this**; retorna **false** caso contrário.

Remove
```
bool Remove(T* object);
```
Remove `object` de **this**.

Classe tPolyReader — polyread.h

A classe abstrata `tPolyRead`, derivada de `tReader`, descreve a estrutura e funcionalidade de um leitor OSW de modelos geométricos genéricos (vimos o conceito de *leitor* no Capítulo 7). A classe declara métodos que possibilitam o especificação de transformações geométricas e descrição de materiais nos arquivos de entrada do leitor, as quais podem ser aplicadas aos modelos geométricos construídos por objetos de classes derivadas de `tPolyReader`.

Veja também Classe `tReader`

Construtor Público

Construtor
```
inline tPolyReader(const char* fileName);
```
Inicializa o objeto base `tReader` com o parâmetro `fileName`.

Atributos Protegidos

Material
`tMaterial Material;`
Material corrente de **this**.

Veja também Classe `tMaterial`

MTM
`t3DTransfMatrix MTM;`
Matriz de transformação geométrica corrente de **this**.

Veja também Classe `t3DTransfMatrix`

Métodos Protegidos

MatchMaterial
`void MatchMaterial();`
Método que implementa o *parsing* da descrição do material corrente de **this**. Se o método não detectar quaisquer erros, os atributos de `Material` são ajustados de acordo com a descrição do material.

MatchTransform
`void MatchTransform();`
Método que implementa o *parsing* da especificação da transformação geométrica corrente de **this**. Se o método não detectar quaisquer erros, a matriz de transformação `MTM` é ajustada de acordo com a especificação da transformação.

Classe tPolyView — polyview.h

Um objeto da classe `tPolyView` representa uma vista OSW na qual podemos efetuar a seleção simples de uma face ou de um vértice de um modelo geométrico. (A classe `tGeoView`, descrita no Capítulo 11, é um exemplo de classe derivada de `tPolyView`. Os métodos de seleção de vértice e face herdados de `tMeshView` são usados na implementação dos comandos de edição dos vértices e faces de um modelo geométrico de cascas.) `tPolyView` é derivada virtualmente de `tView`.

Veja também Classe `tView`

Construtor Público

Construtor
`inline tPolyView(tScene& scene, tCamera* camera);`
Inicializa o objeto base `tView` com os parâmetros `scene` e `camera`.

Métodos Protegidos

SelectSingleFace
`tFace* SelectSingleFace(tModel& model, const char* msg);`
Método de seleção simples de uma face de `model`. Retorna um ponteiro para a face selecionada, ou 0, se face alguma foi selecionada.

SelectSingleVertex
`tVertex* SelectSingleVertex(tModel& model, const char* msg);`
Método de seleção simples de um vértice de `model`. Retorna um ponteiro para o vértice selecionado, ou 0, se vértice algum foi selecionado.

Classe tPrimitive — gmodel.h

A classe `tPrimitive` descreve a estrutura e funcionalidade genéricas de um primitivo gráfico definido no Capítulo 3. A classe deriva virtualmente de `tModel`.

Veja também Classe `tModel`.

Construtor e Destrutor Público

Construtor
`tPrimitive(tGraphicModel* parent);`
Se `parent` for diferente de zero, envia a mensagem `AddPrimitive(this)` a `*parent`.

Veja também `tGraphicModel::AddPrimitive()`

Destrutor
`virtual ~tPrimitive();`
Se o atributo protegido `Parent` for diferente de zero, envia a `*Parent` a mensagem `RemovePrimitive(this)`.

Veja também `tGraphicModel::AddPrimitive()`

Método Público

Delete
`virtual void Delete();`
Destrutor virtual condicional de um primitivo gráfico. Em `tPrimitive`, simplemente `delete this`.

Atributo Protegido

Parent
`tGraphicModel* Parent;`
Ponteiro para o modelo gráfico ao qual **this** pertence.

Veja também Classe `tGraphicModel`

Classe tRay — raytrace.h

Um objeto da classe `tRay` contém os dados que definem um raio de luz manipulado por um *renderer* da classe `tRayTracer`.

Veja também Classe `tRayTracer`

Atributos Públicos

D
`t3DVector D;`
Vetor que define a direção de **this**.

Veja também Classe `t3DVector`

P
`t3DVector P;`
Vetor que define o ponto de origem de **this**.

Veja também Classe `tRayTracer`

Classe tRayTracer — raytrace.h

Um objeto da classe `tRayTracer` é um *renderer* que gera imagens de uma cena através da técnica de *traçado de raios* discutida sucintamente no Capítulo 7. Os métodos que implementam a técnica (`Screen()`, `Scan()`, `Trace()`, `Shade()`, `Intersect()`, `Shadow()`) são privados e não serão descritos aqui. A classe é derivada de `tRenderer`.

Veja também Classe `tRenderer`

Construtor Público

Construtor
`tRayTracer(tCamera* camera = 0);`
Inicializa o objeto base `tRenderer` com o parâmetro `camera` e os demais atributos internos de **this**.

Veja também Classe `tCamera`

Método Público

Render
`void Render(TDC& dc);`
Gera a imagem da cena de **this** e exibe a imagem no dispositivo de contexto `dc`. Executa o método privado `Screen()`.

Veja também Classes `TDC` (OWL), `tScene`

Classe tReader — reader.h

A classe abstrata `tReader` descreve a estrutura e funcionalidade de um leitor OSW genérico (definimos *leitor* no Capítulo 7). Um leitor OSW é um analisador de predicados recursivo que constrói modelos geométricos a partir de descrições contidas em arquivos de texto. Essas descrições devem obedecer a determinadas regras sintáticas especificadas por uma gramática livre de contexto do tipo LL(1). Leitores de classes distintas derivadas de `tReader` podem possuir gramáticas próprias, mas os atributos e métodos declarados em `tReader` definem a estrutura e o comportamento comum de um analisador de predicados de um leitor OSW. A classe declara o método virtual puro privado `Run()`, o qual deve ser sobrecarregado nas classes concretas derivadas de `tReader` para fornecer o comportamento específico do leitor.

Construtor e Destrutor Públicos

Construtor
`tReader(const char* fileName);`
Inicializa o nome do arquivo fonte do leitor com `fileName`.

Destrutor
`virtual~tReader();`
Destrutor virtual. Libera a memória utilizada para armazenar o nome do arquivo fonte de **this**.

Método Público

Execute
`void Execute();`
Executa o leitor. Abre o arquivo fonte de **this** com o nome de arquivo dado no construtor. Executa o método virtual privado `Run()` e fecha o arquivo fonte de **this**.

Atributos Protegidos

File
`tFile File;`
Arquivo fonte de **this**.

FileName
`char* FileName;`
Ponteiro para a cadeia de caracteres que contém o nome do arquivo fonte de **this**.

LineNumber
`int LineNumber;`
Número da linha corrente do arquivo fonte de **this**.

lToken
`int lToken;`
Identificador do tipo do *token* corrente de **this**.

lValue
`tValue lValue;`
União que armazena os atributos do *token* corrente de **this**.

Métodos Protegidos

FindErrorMessage
`virtual char* FindErrorMessage(int code);`
Método virtual que retorna um ponteiro para a cadeia de caracteres correpondente ao código de erro `code`.

FindKeyword
`virtual int FindKeyword(char* keyw);`
Método virtual que retorna o identificador do *token* correspondente à palavra reservada definida pela cadeia de caracteres `keyw`.

Match
`void Match(int token);`
Se `token` é igual ao tipo do símbolo de entrada esperado, captura o próximo *token* do arquivo fonte de **this**, armazenando seu tipo em `lToken` e seus atributos em `lValue`.

MatchColor
`tColor MatchColor();`
Método que implementa o *parsing* da descrição de uma cor no arquivo fonte de **this**. Se o método não detectar quaisquer erros, retorna a cor lida.

Veja também Classe `tColor`

MatchInteger
`int MatchInteger();`
Método que implementa o *parsing* da descrição de um inteiro no arquivo fonte de **this**. Se o método não detectar quaisquer erros, retorna o inteiro lido.

MatchFloat
`double MatchFloat();`
Método que implementa o *parsing* da descrição de um número real no arquivo fonte de **this**. Se o método não detectar quaisquer erros, retorna o número real lido.

MatchPoint
`t3DVector MatchPoint();`
Método que implementa o *parsing* da descrição de um vetor no arquivo fonte de **this**. Se o método não detectar quaisquer erros, retorna o vetor lido.

Veja também Classe `t3DVector`

NextToken
`virtual int NextToken();`
Retorna o próximo *token* do arquivo fonte de **this**.

Classe tRenderer — renderer.h

Um *renderer* é um objeto responsável pela geração da imagem de uma cena. A classe abstrata `tRenderer` mantém um ponteiro para a cena a partir da qual a imagem será gerada e um ponteiro para a câmera utilizada pelo *renderer*. A interface da classe define métodos de transformação de coordenadas entre o sistema global de coordenadas e o sistema de vista e de ajuste dos parâmetros de *rendering*.

Veja também Classe `tScene`, classe `tCamera`

Construtor e Destrutor Públicos

Construtor
`tRenderer(tCamera* camera = 0);`
Construtor *default*. `camera` é a câmera utilizada por **this**. Se `camera` for igual a zero, uma câmera *defautl* é construída para ser utilizada por **this**.

Destrutor
`virtual ~tRenderer();`
Destrutor virtual do *renderer*.

Métodos Públicos

BackToWorld
`t3DVector BackToWorld(const TPoint& p);`

Converte as coordenadas de imagem do ponto `p` para coordenadas globais. Utiliza a distância do plano de trabalho para determinar as coordenadas globais do ponto.

Veja também `Map`, `WorkPlaneDistance`

GetCamera
```
inline tCamera* GetCamera();
```
Retorna um ponteiro para a câmera utilizada por **this**.

GetScene
```
inline tScene* GetScene();
```
Retorna um ponteiro para a cena de **this**.

GetViewport
```
inline const tViewport& GetViewport() const;
```
Retorna uma referência para a *viewport* de **this**. A *viewport* define as dimensões da imagem gerada pelo *renderer*.

Veja também Classe `tViewport`

Map
```
TPoint Map(const t3DVector& p);
```
Converte as coordenadas globais do ponto `p` para coordenadas de imagem.

Render
```
virtual void Render(TDC& dc) = 0;
```
Gera a imagem da cena de **this** e exibe a imagem no dispositivo de contexto `dc`. Um objeto de uma classe derivada da classe `TDC` pode representar a área cliente de uma janela de vista gráfica ou um mapa de *pixels* em memória. As classes derivadas de `tRenderer` devem sobrecarregar esse método.

Veja também Classe `TDC` (OWL)

SetCamera
```
SetCamera(tCamera* camera);
```
Torna `camera` a nova câmera utilizada por **this**.

SetScene
```
SetScene(tScene* scene);
```
Torna `scene` a nova cena de **this**. A cena de um *renderer* é a cena a partir da qual o objeto gera uma imagem.

SetViewport
```
void SetViewport(const tViewport& vport);
```
Ajusta os pontos de canto da *viewport* de **this** com uma cópia dos pontos de canto de `vport`.

```
void SetViewport(const TPoint& p1, TPoint& p2);
```
Ajusta os pontos de canto da *viewport* de **this** com as coordenadas dos pontos `p1` e `p2`.

Veja também Classe `TPoint` (OWL)

SetWorkPlaneDistance
`void SetWorkPlaneDistance(double d);`
Ajusta a distância do plano de trabalho para o valor `d`.

Veja também `WorkPlaneDistance`

Atributos Protegidos

Camera
`tCamera* Camera;`
Ponteiro para a câmera utilizada por **this**.

Scene
`tScene* Scene;`
Ponteiro da cena a partir da qual **this** gera uma imagem.

Viewport
`tViewport Viewport;`
Viewport que define as dimensões das imagens geradas por **this**.

WorkPlaneDistance
`double WorkPlaneDistance;`
Distância do plano de trabalho de **this**. O plano de trabalho é um plano paralelo ao plano de projeção, situado a uma distância d desse plano, tomada sobre o VPN e em relação à origem do VRC.

Classe tScalarExtractor — contour.h

Um objeto da classe `tScalarExtractor` é um filtro de extração de um campo escalar de um modelo mecânico. A entrada do filtro é um modelo de decomposição por células; a saída do filtro é o mesmo modelo de decomposição por células de entrada. O filtro não transforma a geometria ou a topologia do modelo de entrada; somente extrai o valor escalar de um dos graus de liberdade dos nós do modelo de entrada e armazena a informação no atributo `Scalar` do próprio nó. Um objeto da classe `tScalarExtractor` é um filtro de transformação de atributos, como visto no Capítulo 7.

Veja também `tVertex::Scalar`, classes `tNode`, `tMesh`

Construtor Público

Construtor
`inline tScalarExtractor();`
Construtor *default*. Inicializa a entrada e a saída do filtro.

Métodos Públicos

Execute
`void Execute();`
Executa o filtro.

GetInput
`inline tMesh* GetInput();`
Retorna um ponteiro para o modelo mecânico de entrada de **this**.

GetOutput
```
inline tMesh* GetOutput();
```
Retorna um ponteiro para o modelo mecânico de saída **this**.

SelectField
```
virtual void SelectField(int dof, bool u = true);
```
Seleciona o campo escalar do grau de liberadade de número `dof` dos nós do modelo mecânico de entrada de **this**. Se a *flag* `u` for igual a **false**, o campo escalar é definido pelo esforço `P` do grau de liberdade `dof`.

Veja também Classe `tDOF`

SetInput
```
void SetInput(tMesh* mesh);
```
Torna `mesh` o modelo de entrada de **this**.

Classe tScanner — scanner.h

Um objeto da classe `tScanner` é um *renderer* que usa o algoritmo de linhas de varredura para gerar imagens de modelos poliedrais. É o *renderer default* de uma vista de uma cena. `tScanner` é derivada de `tRenderer`.

Veja também Classe `tRenderer`

Enumeração Pública

tRenderMode
```
enum tRenderMode;
```
Valores dos modos de *rendering* de objeto da classe `tScanner`.

Veja também `RenderMode`

Construtor e Destrutor Públicos

Construtor
```
tScanner(tCamera* camera = 0);
```
Construtor *default*. Inicializa a classe base `tRenderer` com `camera`.

Destrutor
```
~tRenderer();
```
Destrutor de **this**.

Métodos Públicos

GetEdgeVisibility
```
inline bool GetEdgeVisibility() const;
```
Retorna o valor da *flag* de visibilidade de arestas de **this**. Se o valor da *flag* de visibilidade de arestas for igual a **true**, a imagem gerada por **this** conterá os fios-de-arame dos modelos da cena.

Veja também `SetEdgeVisibility`

GetRenderMode
```
inline tRenderMode GetRenderMode() const;
```
Retorna o modo de *rendering* de **this**.

Veja também `tScanner::tRenderMode`

Render
```
void Render(TDC& dc);
```
Gera a imagem da cena de **this** e exibe a imagem no dispositivo de contexto `dc`.

SetEdgeVisibility
```
void SetEdgeVisibility(bool visible);
```
Ajusta o valor da *flag* de visibilidade de arestas de **this** com o parâmetro `visible`. Se `visible` for igual a **true**, as arestas dos modelos da cena serão exibidas na imagem gerada por **this**.

Veja também `GetEdgeVisibility`

SetRenderMode
```
void SetRenderMode(tRenderMode mode);
```
Ajusta o modo de *rendering* de **this**. Um objeto da classe `tScanner` pode gerar imagens fio-de-arame, com remoção de linhas escondidas e com tonalização "*flat*" e de Gourard.

Veja também `tScanner::tRenderMode`

Atributos Protegidos

EdgeVisibilityFlag
```
bool EdgeVisibilityFlag;
```
Flag de visibilidade de arestas de **this**.

RenderMode
```
tRenderMode RenderMode;
```
Modo de *rendering* de **this**. O modo de *rendering* define o tipo da imagem gerada pelo *renderer*: imagem fio-de-arame (`Wireframe`), com linhas escondidas (`HiddenLines`), com tonalização "*flat*" (`Flat`) ou com tonalização de Gourard (`Gourard`).

Classe tScene **scene.h**

Um objeto da classe `tScene` representa uma coleção de atores, luzes e câmeras que definem um ambiente a partir do qual será sintetizada uma imagem. A imagem é tomada por uma das câmeras da cena, em função da interação das luzes da cena com os materiais das superfícies dos atores da cena. As classes derivadas de `tScene` são responsáveis pela construção das vistas gráficas manipuladas pela cena.

Veja também Classes `tActor`, `tLight`, `tCamera` e `tRenderer`

Construtores e Destrutor Públicos

Construtores
```
tScene(tDocument& doc);
```
Inicializa as listas de atores, luzes e câmeras da cena do documento `doc`. A luz ambiente de **this** é ajustada com a cor branco. A cor de fundo é preto.

`tScene(const tScene& scene);`
Construtor de cópia. Cria uma cena que contém os mesmos atores, as mesmas luzes, as mesmas câmeras, a mesma luz ambiente e o mesmo fundo de `scene`.

Destrutor
`virtual ~tScene();`
Destrutor virtual. Executa o método `DeleteAll()`.

Métodos Públicos

AddActor
`void AddActor(tActor* actor);`
Adiciona o ator `*actor` na lista de atores de **this**.

AddCamera
`void AddCamera(tCamera* camera);`
Adiciona a câmera `camera` na lista de cameras de **this**.

AddLight
`void AddLight(tLight* light);`
Adiciona a luz `light` na lista de luzes de **this**.

CreateView
`virtual tView* CreateView(long flags);`
Cria uma nova câmera, de acordo com o valor de `flags`. Executa `ConstructView()`, passando um ponteiro para a câmera criada como argumento. Se `flags` for igual a `vfUserView`, abre uma caixa de diálogo solicitando os parâmetros de ajuste de câmera.

DeleteActor
`void DeleteActor(tActor* actor);`
Remove o ator `actor` da lista de atores de **this** e destrói o ator.

DeleteAll
`void DeleteAll();`
Destrói todos os componentes das lista de atores, de luzes e de câmeras da cena.

DeleteCamera
`void DeleteCamera(tCamera* camera);`
Remove a câmera `camera` da lista de câmeras de **this** e destrói a câmera.

DeleteLight
`void DeleteLight(tLight* light);`
Remove a luz `light` da lista de luzes de **this** e destrói a luz.

GetActorIterator
`inline tActorIterator GetActorIterator();`
Retorna um *iterator* da lista de atores de **this**.

GetAmbientLightColor
`inline tColor GetAmbientLightColor() const;`
Retorna os componentes RGB da luz ambiente da cena.

GetBkgndColor
```
inline tColor GetBkgndColor() const;
```
Retorna os componentes RGB da luz de fundo da cena.

GetCameraIterator
```
inline tCameraIterator GetCameraIterator();
```
Retorna um *iterator* da lista de câmeras de **this**.

GetDocument
```
inline tDocument* GetDocument();
```
Retorna um ponteiro para o documento de **this**.

GetLightIterator
```
inline tLightIterator GetLightIterator();
```
Retorna um *iterator* da lista de luzes de **this**.

SetAmbientLight
```
inline void SetAmbientLightColor(const tColor& color);
```
Ajusta os componentes RGB da luz ambiente da cena com os componentes RGB de `color`.

SetBkgndColor
```
inline void SetBkgndColor(const tColor& color);
```
Ajusta os componentes RGB da cor de fundo da cena com os componentes RGB de `color`.

Atributos Protegidos

Document
```
tDocument* Document;
```
Ponteiro para o documento ao qual a cena pertence.

Métodos Protegidos

ConstructView
```
virtual tView* ConstructView(tScene& scene, tCamera* camera);
```
Construtor "virtual" das vistas da cena. Constrói uma vista pertencente à cena `scene`, cuja imagem será tomada pela câmera `camera`.

Classe tShell — shell.h

Um objeto da classe `tShell` representa um modelo de cascas, tal como definido no Capítulo 3. A interface da classe define métodos responsáveis pelas operações de modelagem de um modelo de cascas. `tShell` é derivada da classe base `tMeshableModel`.

Veja também Classe `tMeshableModel`

Construtores e Destrutor Públicos

Construtores
```
tShell();
```
Construtor *default*.

```
tShell(const char* name);
```
Inicializa a classe base `tMeshableModel` com `name`.

Destrutor
```
~tShell();
```
Destrói todas as listas dos elementos topológicos de **this**.

Métodos Públicos

GetEdgeIterator
```
tEdgeIterator GetEdgeIterator();
```
Retorna um *iterator* para a lista de arestas de **this**.

Veja também Classe `tEdgeIterator`

GetFaceIterator
```
tFaceIterator GetFaceIterator();
```
Retorna um *iterator* para a lista de faces de **this**.

Veja também Classe `tFaceIterator`

GetNumberOfEdges
```
inline int GetNumberOfEdges() const;
```
Retorna o número de arestas de **this**.

GetNumberOfFaces
```
inline int GetNumberOfFaces() const;
```
Retorna o número de faces de **this**.

GetNumberOfVertices
```
inline int GetNumberOfVertices() const;
```
Retorna o número de vértices de **this**.

GetVertexIterator
```
tVertexIterator GetVertexIterator();
```
Retorna um *iterator* para a lista de vértices de **this**.

Veja também Classe `tVertexIterator`

ke
```
void ke(tEdgeUse* eu);
```
Remove da lista de arestas de **this** a aresta definida pelo uso de aresta `eu`, e todos os seus "usos".

kv
```
void kv(tVertex* v);
```
Remove o vértice `v` da lista de vértices de **this**, e todos os elementos topológicos que usam `v`.

me
```
tEdgeUse* me(tVertex* v, tEdgeUse* eu);
```
Cria um nova aresta de **this** e seus "usos", partindo do vértice `v`. Insere um uso de aresta na lista de usos de aresta do laço de `eu` e retorna um ponteiro para esse uso de aresta.

mf
```
tEdgeUse* mf(tVertex* v);
```
Cria um novo uso de aresta, partindo do vértice `v`, um novo laço e uma nova face de **this**. Retorna um ponteiro para o novo uso de aresta criado.

mv
```
tVertex* mv(const t3DVector& p);
```
Cria um novo vértice de **this**, com coordenadas definidas por `p`. Retorna um ponteiro para o vértice criado.

SplitEdges
```
tBoundaryFace* SplitEdges(double size) const;
```
Gera um modelo de contornos de faces de **this**, conforme descrito no Capítulo 6. Retorna um ponteiro para o modelo gerado.

Veja também Classe `tBoundaryFace`

Transform
```
void Transform(const t3DTransfMatrix& m);
```
Transforma todos os vértices da lista de vértices de **this**.

Classe tShell::tEdge — shell.h

Um objeto da classe `tShell::tEdge` representa uma aresta de um modelo de cascas, tal como definida no Capítulo 3.

Atributos Públicos

RadialEdges
```
tRadialEdge* RadialEdges;
```
Ponteiro para o primeiro elemento da lista de arestas radiais de **this**.

Veja também Classe `tShell::tRadialEdge`

V1
```
tVertex* V1;
```
Ponteiro para o primeiro vértice de **this**.

Veja também Classe `tShell::tVertex`

V2
```
tVertex* V2;
```
Ponteiro para o segundo vértice de **this**.

Veja também Classe `tShell::tVertex`

Construtor e Destrutor Públicos

Construtor
```
tEdge(tEdgeUse* edgeUse);
```
Inicializa `V1` com `edge->Vertex` e `V2` com `edge->Next->Vertex`. Inicializa a lista de arestas radiais de **this** como uma lista vazia.

Veja também Classes `tShell::tEdgeUse`, `tShell::tRadialEdge`

Destrutor
`~tEdge();`
Destrói a lista de arestas radiais de **this**.

Veja também Classe `tShell::tRadialEdge`

Classe tShell::tEdgeUse — shell.h

Um objeto da classe `tShell::tEdgeUse` representa um uso de aresta de um modelo de cascas, tal como definido no Capítulo 3.

Atributos Públicos

Edge
`tEdge* Edge;`
Aresta a qual **this** pertence.

Veja também `tShell::tEdge`

Loop
`tLoop* Loop;`
Laço da face a qual **this** pertence.

Veja também `tShell::tLoop`

Vertex
`tVertex* Vertex;`
Ponteiro para o vértice do qual parte **this**.

Veja também `tShell::tVertex`

Construtor e Destrutor Público

Construtor
`inline tEdgeUse(tVertex* vertex);`
Envia a mensagem `MakeVertexUse(this)` para o vértice `*vertex`. `*vertex` é o vértice do qual parte o uso de aresta.

Veja também `tShell::tVertex::MakeVertexUse()`

Destrutor
`~tEdgeUse();`
Envia a mensagem `KillVertexUse(this)` para o vértice do qual parte **this**.

Classe tShell::tFace — shell.h

Um objeto da classe `tShell::tFace` representa uma face de um modelo de cascas, tal como definida no Capítulo 3. A classe deriva da classe abstrata `::tFace` de faces globais de OSW.

Veja também Classe `::tFace`

Atributos Públicos

Loops
`tLoop* Loops;`
Ponteiro para o primeiro elemento da lista de laços de **this**.

Veja também Classe `tShell::tLoop`

Material
`tMaterial Material;`
Material de **this**.

Number
`int Number;`
Inteiro identificador de **this**.

OuterLoop
`tLoop* OuterLoop;`
Ponteiro para o laço externo de **this**. `OuterLoop` é um elemento de `Loops`.

Veja também `tShell::tFace::Loops`

Shell
`tShell* Shell;`
Ponteiro para a casca a qual **this** pertence.

Construtores e Destrutor Públicos

Construtores
`inline tFace(int fid, tShell* shell);`
Inicializa os atributos `Number` e `Shell` com os parâmetros `fid` e `shell`, respectivamente. Inicializa a lista de laços de **this** como uma lista vazia.

Veja também `tShell::tLoop`

`tFace(const tFace& face, tShell& shell);`
Inicializa `Shell` com `&shell` e os demais atributos de **this** com uma cópia dos atributos de `face`. Inicializa a lista de laços de **this** como uma lista vazia.

Veja também `tShell::tLoop`

Destrutor
`~tFace();`
Destrói os laços da lista de laços de **this**.

Veja também `tShell::tLoop`

Métodos Públicos

AddLoop
`void AddLoop(tLoop* loop);`
Adiciona o laço `loop` à lista de laços de **this**.

GetMaterial
`tMaterial GetMaterial() const;`

Retorna o material da superfície de **this**. O método é declarado como virtual puro na classe base `::tFace`.

GetNumber
```
int GetNumber() const;
```
Retorna o inteiro identificador de **this**. O método é declarado como virtual puro na classe base `::tFace`.

NormalAt
```
t3DVector Normal(const t3DVector&) const;
```
Retorna o vetor normal à superfície de **this**. O método é declarado como virtual puro na classe base `::tFace`.

Veja também Classe `t3DVector`

RemoveLoop
```
void RemoveLoop(tLoop* loop);
```
Remove o laço `loop` da lista de laços de **this**.

Classe tShell::tLoop — shell.h

Um objeto da classe `tShell::tLoop` representa um laço de uma face de um modelo de cascas, como definido no Capítulo 3.

Atributos Públicos

Face
```
tFace* Face;
```
Ponteiro para a face a qual **this** pertence.

Veja também Classe `tFace`

FirstEdge
```
tEdgeUse* FirstEdge;
```
Ponteiro para o primeiro elemento da lista de usos de aresta de **this**.

Veja também Classe `tEdgeUse`

Construtor e Destrutor Públicos

Construtor
```
inline tLoop(tFace* face);
```
Envia a mensagem `AddLoop(this)` a `*face`. Inicializa a lista de usos de aresta de **this** como uma lista vazia.

Veja também `tShell::tFace::AddLoop()`

Destrutor
```
~tLoop();
```
Destrói todos os elementos da lista de usos de aresta de **this**. Envia a mensagem `RemoveLoop(this)` a `*Face`.

Veja também `tShell::tFace::RemoveLoop()`

Classe tShell::tRadialEdge shell.h

Um objeto da classe `tShell::tRadialEdge` representa uma aresta radial de um modelo de cascas, como definida no Capítulo 3.

Atributo Público

EdgeUse
`tEdgeUse* EdgeUse;`
Uso de aresta de **this**.

Veja também Classe `tShell::tEdgeUse`

Construtor Público

Construtor
`inline tRadialEdge(tEdgeUse* edgeUse);`
Inicializa o atributo `EdgeUse` com o parâmetro `edgeUse`.

Veja também Classe `tShell::tEdgeUse`

Classe tShell::tVertex shell.h

Um objeto da classe `tShell::tVertex` representa um vértice de um modelo de cascas, tal como definido no Capítulo 3. A classe deriva da classe `::tVertex` de vértices globais de OSW.

Veja também Classe `::tVertex`

Atributo Público

VertexUses
`tVertexUse* VertexUses;`
Ponteiro para o primeiro elemento da lista de usos de vértice de **this**.

Construtores e Destrutor Público

Construtores
`tVertex(int vid, const t3DVector& position);`
Inicializa o objeto base `::tVertex` com `position`. Ajusta o identificador do vértice com o parâmetro `vid` e a lista de usos de vértice do vértice como vazia.

Veja também Classe `tShell::tVertexUse`

`tVertex(const tVertex& v);`
Construtor de cópia. Inicializa os atributos do vértice com uma cópia dos atributos de `v`.

Destrutor
`~tVertex();`
Destrói a lista de usos de vértice de **this**.

Métodos Públicos

KillVertexUse
```
void KillVertexUse(tEdgeUse* edgeUse);
```
Remove da lista de usos de vértice de **this** o uso de vértice identificado pelo parâmetro `edgeUse`.

Veja também Classes `tShell::tEdgeUse`, `tShell::tVertexUse`

MakeVertexUse
```
void MakeVertexUse(tEdgeUse* edgeUse);
```
Constrói um uso de vértice identificado pelo parâmetro `edgeUse` e adiciona o uso de vértice à lista de usos de vértice de **this**.

Veja também Classes `tShell::tEdgeUse`, `tShell::tVertexUse`

Classe tShell::tVertexUse — shell.h

Um objeto da classe `tShell::tVertexUse` representa um uso de vértice de um modelo de cascas, como definido no Capítulo 3.

Atributo Público

EdgeUse
```
tEdgeUse* EdgeUse;
```
Uso de aresta cujo vértice de origem é **this**.

Veja também Classe `tShell::tEdgeUse`

Construtor Público

Construtor
```
inline tVertexUse(tVertexUse* edgeUse);
```
Inicializa o atributo `EdgeUse` com o parâmetro `edgeUse`.

Veja também Classe `tShell::tEdgeUse`

Classe tShellMesh — felement.h

Um objeto da classe `tShellMesh` é uma malha de elementos finitos de casca definidos no Capítulo 6. A classe deriva diretamente de `tMesh`.

Veja também Classe `tMesh`

Construtores Públicos

Construtores
```
inline tShellMesh();
```
Construtor *default*. Sem funcionalidade.

```
inline tShellMesh(const char* name);
```
Inicializa o objeto base `tMesh` com `name`.

Classe tShellReader polyread.h

Um objeto da classe `tShellReader` representa um leitor OSW de modelos geométricos de cascas (vimos o conceito de *leitores* no Capítulo 7). A classe é derivada da classe abstrata `tPolyReader`.

Veja também Classes `tPolyReader`, `tShell`

Construtor Público

Construtor
```
inline tShellReader(const char* fileName);
```
Inicializa o objeto base `tPolyReader` com o parâmetro `fileName`.

Método Público

GetOutput
```
inline tShell* GetOutput();
```
Retorna um ponteiro para o modelo de saída de **this**.

Classe tShellSweeper polysour.h

Um objeto da classe `tShellSwerper` é um fonte de modelos geométricos de cascas definidos no Capítulo 3 (vimos o conceito de *fontes* no Capítulo 7). Os métodos de geração dos modelos de cascas de `tShellSweeper` implementam os processos de varredura translacional e varredura rotacional descritos no Capítulo 3.

Definição de Tipo Pública

tPolyline
```
typedef tFixedArray<t3DVector> tPolyline;
```
`tPolyline` é um arranjo de vetores no espaço.

Veja também Classe `t3DVector`, classe paramétrica `tFixedArray`

Métodos Públicos

MakeBox
```
tShell* MakeBox(const t3DVector& center,
  const t3DVector& up, const t3DVector& normal,
  const t3DVector& dim);
```
Constrói por varredura translacional o modelo de cascas de uma caixa com dimensões dadas em `dim` e sistema local de coordenadas definido por `center` e pelos vetores `up` e `normal`.

Veja também `SweepByTranslation()`

MakeCylinder
```
tShell* MakeCylinder(const t3DVector& center,
  const t3DVector& firstp, const t3DVector& normal, double height);
```
Constrói por varredura translacional o modelo de cascas de uma cilindro de altura `height` e base definida por `center`, `firstp` e `normal`.

Veja também `SweepByTranslation()`

MakeCylindricalShell
```
tShell* MakeCylindricalShell(const t3DVector& center,
  const t3DVector& firstp, const t3DVector& normal,
  double height, double angle);
```
Constrói por varredura translacional o modelo de uma casca cilíndrica de altura `height` e arco geratriz definido por `center`, `firstp` e `normal`, com ângulo interno igual a `angle`.

Veja também `SweepByTranslation()`

MakeDome
```
tShell* MakeDome(const t3DVector& center,
  const t3DVector& firstp, const t3DVector& normal);
```
Constrói por varredura rotacional o modelo de cascas de um domo com centro em `center`, raio definido pela distância entre `firstp` e eixo polar `normal`.

Veja também `SweepByRotation()`

MakeSphere
```
tShell* MakeSphere(const t3DVector& center,
  const t3DVector& firstp, const t3DVector& normal);
```
Constrói por varredura rotacional o modelo de cascas de uma esfera com centro em `center`, raio definido pela distância entre `firstp` e eixo polar `normal`.

Veja também `SweepByRotation()`

SweepByTranslation
```
tShell* SweepByTranslation(const tPolyline& poly,
  const t3DVector& path);
```
Varredura translacional. Retorna um ponteiro para o modelo de cascas resultante do "escorregamento" da geratriz `poly` ao longo do caminho definido por `path`.

SweepByRotation
```
tShell* SweepByRotation(const tPolyline& poly,
  const t3DVector& org, const t3DVector& dir, double angle);
```
Varredura rotacional. Retorna um ponteiro para o modelo de cascas resultante da rotação de `angle` radianos da geratriz `poly` em torno do eixo definido pelos vetores `org` e `dir`.

Classe tShellWriter — polywrit.h

Um objeto da classe `tShellWriter` é um escritor de modelos geométricos de cascas da classe `tShell`, definidos no Capítulo 3 (vimos o conceito de *escritor* no Capítulo 7). A classe é derivada da classe abstrata `tWriter`. O método de escrita do modelo de entrada do escritor é implementado pelo método privado `Run()`, declarado como virtual puro na classe base `tWriter`.

Veja também Classes `tShell`, `tWriter`

Construtor Público

Construtor
```
inline ShellWriter(const char* fileName);
```

Inicializa o objeto base `tWriter` com `fileName`.

Métodos Públicos

GetInput
```
inline tShell* GetInput();
```
Retorna um ponteiro para o modelo de entrada de **this**.

SetInput
```
void SetInput(tShell* shell);
```
Torna `shell` o modelo de entrada de **this**.

Atributo Protegido

Input
```
tShell* Input;
```
Ponteiro para o modelo de entrada de **this**.

Classe tSolid **solid.h**

Um objeto da classe `tSolid` representa um modelo de sólidos, tal como definido no Capítulo 3. Em `tSolid`, ao contrário da classe `tShell`, as classes que definem os elementos topológicos de um modelo de sólidos (vértices, arestas, faces, etc.), bem como os métodos que implementam os operadores de Euler, são encapsulados na definição da classe. (Adotamos essa estratégia de proteção porque queremos garantir a integridade geométrica e topológica de um modelo de sólidos.) Os operadores de Euler são utilizados somente por objetos de classes "amigas" de `tSolid`, tais como `tSolidReader` e `tSolidSweeper`. `tSolid` é derivada da classe base `tMeshableModel`.

Veja também Classe `tMeshableModel`

Construtores e Destrutor Públicos

Construtores
```
tSolid();
```
Construtor *default*.

```
tSolidconst char* name;
```
Inicializa a classe base `tMeshableModel` com `name`.

Destrutor
```
~tSolid();
```
Destrói todos os elementos topológicos de **this**.

Métodos Públicos

GetEdgeIterator
```
tEdgeIterator GetEdgeIterator();
```
Retorna um *iterator* para as arestas de **this**.

Veja também Tipo `tEdgeIterator`

GetFaceIterator
`tFaceIterator GetFaceIterator();`
Retorna um *iterator* para as faces de **this**.

Veja também Tipo `tFaceIterator`

GetNumberOfEdges
`inline int GetNumberOfEdges() const;`
Retorna o número de arestas de **this**.

GetNumberOfFaces
`inline int GetNumberOfFaces() const;`
Retorna o número de faces de **this**.

GetNumberOfVertices
`inline int GetNumberOfVertices() const;`
Retorna o número de vértices de **this**.

GetVertexIterator
`tVertexIterator GetVertexIterator();`
Retorna um *iterator* para os vértices de **this**.

Veja também Tipo `tVertexIterator`

SplitEdges
`tBoundaryFace* SplitEdges(double size) const;`
Gera um modelo de contornos de faces de **this**, conforme descrito no Capítulo 6. Retorna um ponteiro para o modelo gerado.

Veja também Classe `tBoundaryFace`

Transform
`void Transform(const t3DTransfMatrix& m);`
Transforma todos os vértices de **this**, usando a matriz de transformação `m`.

Classe tSolidMesh — belement.h

Um objeto da classe `tSolidMesh` é uma malha de elementos de contorno de dimensão topológica 2, definidos no Capítulo 6. A classe deriva diretamente de `tMesh`.

Veja também Classe `tMesh`

Construtores Públicos

Construtores
`inline tSolidMesh();`
Construtor *default*. Sem funcionalidade.

`inline tSolidMesh(const char* name);`
Inicializa o objeto base `tMesh` com `name`.

Classe tSolidReader polyread.h

Um objeto da classe `tSolidReader` representa um leitor OSW de modelos geométricos de sólidos (vimos o conceito de *leitores* no Capítulo 7). A classe é derivada da classe abstrata `tPolyReader`.

Veja também Classes `tPolyReader`, `tSolid`

Construtor Público

Construtor
```
inline tSolidReader(const char* fileName);
```
Inicializa o objeto base `tPolyReader` com o parâmetro `fileName`.

Método Público

GetOutput
```
inline tSolid* GetOutput();
```
Retorna um ponteiro para o modelo de saída de **this**.

Classe tSolidSweeper polysour.h

Os métodos de geração dos modelos de cascas de `tSolidSweeper` implementam os processos de varredura translacional e varredura rotacional descritos no Capítulo 3.

Definição de Tipo Pública

tPolyline
```
typedef tFixedArray<t3DVector> tPolyline;
```
`tPolyline` é um arranjo de vetores no espaço.

Veja também Classe `t3DVector`, classe paramétrica `tFixedArray`

Métodos Públicos

MakeBox
```
tSolid* MakeBox(const t3DVector& center,
  const t3DVector& up, const t3DVector& normal,
  const t3DVector& dim);
```
Constrói por varredura translacional o modelo de sólidos de uma caixa com dimensões dadas em `dim` e sistema local de coordenadas definido por `center` e pelos vetores `up` e `normal`.

Veja também `SweepByTranslation()`

MakeCylinder
```
tSolid* MakeCylinder(const t3DVector& center,
  const t3DVector& firstp, const t3DVector& normal, double height);
```
Constrói por varredura translacional o modelo de sólidos de uma cilindro de altura `height` e base definida por `center`, `firstp` e `normal`.

Veja também `SweepByTranslation()`

MakeSphere
```
tSolid* MakeSphere(const t3DVector& center,
  const t3DVector& firstp, const t3DVector& normal);
```
Constrói por varredura rotacional o modelo de sólidos de uma esfera com centro em `center`, raio definido pela distância entre `firstp` e eixo polar `normal`.

Veja também `SweepByRotation()`

SweepByTranslation
```
tSolid* SweepByTranslation(const tPolyline& poly,
  const t3DVector& path);
```
Varredura translacional. Retorna um ponteiro para o modelo de sólidos resultante do "escorregamento" da geratriz `poly` ao longo do caminho definido por `path`.

SweepByRotation
```
tSolid* SweepByRotation(const tPolyline& poly,
  const t3DVector& org, const t3DVector& dir, double angle);
```
Varredura rotacional. Retorna um ponteiro para o modelo de sólidos resultante da rotação de `angle` radianos da geratriz `poly` em torno do eixo definido pelos vetores `org` e `dir`.

Classe tShellWriter — polywrit.h

Um objeto da classe `tShellWriter` é um escritor de modelos geométricos de sólidos da classe `tSolid`, definidos no Capítulo 3 (vimos o conceito de *escritor* no Capítulo 7). A classe é derivada da classe abstrata `tWriter`. O método de escrita do modelo de entrada do escritor é implementado pelo método privado `Run()`, declarado como virtual puro na classe base `tWriter`.

Veja também Classes `tSolid`, `tWriter`

Construtor Público

Construtor
```
inline SolidWriter(const char* fileName);
```
Inicializa o objeto base `tWriter` com `fileName`.

Métodos Públicos

GetInput
```
inline tSolid* GetInput();
```
Retorna um ponteiro para o modelo de entrada de **this**.

SetInput
```
void SetInput(tSolid* solid);
```
Torna `solid` o modelo de entrada de **this**.

Atributo Protegido

Input
```
tSolid* Input;
```
Ponteiro para o modelo de entrada de **this**.

Classe tSourcePoint belement.h

Um objeto da classe `tSourcePoint` representa um ponto fonte utilizado por um *solver* da classe `tBESolver`.

Veja também Classe `tBESolver`

Atributos Públicos

Number
```
int Number;
```
Número do ponto fonte.

Position
```
t3DVector Position;
```
Posição do ponto fonte.

Veja também Classe `t3DVector`.

Construtores Públicos

Construtores
```
tSourcePoint();
```
Construtor *default*.

```
tSourcePoint(int number, const t3DVector& pos);
```
Inicializa os atributos `Number` e `Position` de **this** com os parâmetros `number` e `pos`, respectivamente.

Classe tSolver solver.h

`tSolver` é uma classe abstrata que representa um processo genérico de análise numérica elastostática de estruturas. A interface da classe define métodos para inicialização do processo de análise, verificação dos dados do modelo mecânico, construção, montagem e resolução do sistema de equações lineares e término da análise. Classes derivadas de `tSolver` devem ser responsáveis pela mantagem do sistema de equações.

Veja também Classe `tLinearSystem`

Construtor e Destrutor Públicos

Construtor
```
tSolver(tMesh* mesh);
```
Inicializa o analisador. `mesh` é o modelo mecânico a ser analisado.

Destrutor
```
virtual ~tSolver();
```
Destrutor virtual. Destrói o sistema de equações de **this**.

Métodos Públicos

Run
```
virtual void Run();
```
Executa o processo de análise. Inicialmente, chama os métodos virtuais `Init()` e

`Check()`. Em seguida, executa `ConstructLinearSystem()` e `AssembleSystem()`. Por fim, chama o método `Terminate()`.

Atributos Protegidos

LS
```
tLinearSystem* LS;
```
Ponteiro para o sistema de equações lineares de **this**.

Mesh
```
tMesh* Mesh;
```
Ponteiro para o modelo mecânico de **this**.

NumberOfDOFs
```
int NumberOfDOFs;
```
Número total de graus de liberdade do modelo mecânico de **this**.

NumberOfFixs
```
int NumberOfFixs;
```
Número de graus de liberdade restringidos do modelo mecânico de **this**.

Métodos Protegidos

AssembleSystem
```
virtual void AssembleSystem() = 0;
```
Método virtual puro de montagem do sistema de equações lineares de **this**. Deve ser sobrecarregado nas classes derivadas de `tSolver`.

Check
```
virtual void Check(;
```
Verifica a integridade do modelo mecânico de entrada. Envia a mensagem `Check()` a todas as células do modelo mecânico.

ConstructLinearSystem
```
virtual tLinearSystem* ConstructLinearSystem();
```
Construtor "virtual" do sistema de equações lineares de **this**. Em `tSolver`, constrói um objeto da classe `tFullSystem`.

Veja também Classe `tFullSystem`

Init
```
virtual void Init();
```
Inicialização do processo de análise. Executa os métodos `Check()`, para verificação dos dados do modelo mecânico, e `ConstructLinearSystem()`, para construção do sistema de equações lineares.

Resume
```
virtual void Resume(tXSolver& x);
```
Tratamento de erros da análise. Qualquer um dos métodos de `tSolver`, ou das classes derivadas de `tSolver`, podem gerar uma exceção da classe `tXSolver`, se ocorrer um erro na análise. Quando isso acontece, o método `Resume()` é executado. Em `tSolver`, o método exibe a mensagem de erro de `x` e interrompe a análise.

Terminate
```
virtual void Terminate();
```
Termina o processo de análise. Transfere a solução do sistema de equações de **this** para os graus de liberdade correspondentes no modelo mecânico de entrada.

Classe tVector — matrix.h

Um objeto da classe `tVector` é um manipulador de vetores de números reais.

Construtores e Destrutor Públicos

Construtores
```
tVector(uint n);
```
Constrói um corpo de vetor com `n` elementos inicializados com zero.

```
tVector(const tVector v, int cpType = tObject::ShallowCopy);
```
Construtor de cópia. Torna **this** uma cópia de `v`. O tipo de cópia é especificado pelo parâmetro `cpType`: se `cpType` for igual a `tObject::DeepCopy`, um novo corpo de vetor é construído e seus elementos são inicializados com os valores dos elementos de `v`; se `cpType` for igual a `tObject::ShallowCopy`, um novo corpo de vetor não é criado; nesse caso, o corpo de `v` é compartilhado.

Veja também Classe `tObjectBody`

Destrutor
```
~tVector();
```
Se o corpo do vetor for utilizado somente por **this**, destrói o corpo do vetor.

Métodos Públicos

Copy
```
tVector Copy(const tVector v, int copy = tObject::ShallowCopy);
```
Cópia de vetores. Torna **this** uma cópia de `v` e retorna **this**. O tipo de cópia é especificado pelo parâmetro `cpType`: se `cpType` for igual a `tObject::DeepCopy`, um novo corpo de vetor é construído e seus elementos são inicializados com os valores dos elementos de `v`; se `cpType` for igual a `tObject::ShallowCopy`, um novo corpo de vetor não é criado; nesse caso, o corpo de `v` é compartilhado.

GetNumberOfElements
```
inline int GetNumberOfElements() const;
```
Retorna o número de elementos do vetor.

IsEqual
```
bool IsEqual(const tVector v, int eqType = tObject::ShallowEqual);
```
Comparação de vetores. Retorna **true** se **this** e `v` são iguais; retorna **false** caso contrário. O tipo de identidade é especificado pelo parâmetro `eqType`: se `eqType` for igual a `tObject::ShallowEqual`, os vetores são iguais se seus corpos forem os mesmos; se `eqType` for igual a `tObject::DeepEqual`, os vetores são iguais se os elementos de seus corpos forem iguais.

Operadores Públicos

operator =
```
tVector operator =(const tVector v);
```
Operador de cópia (*shallow-copy*). Torna **this** uma cópia de `v` e retorna **this**.

Veja também `tVector::Copy`

operator +
```
tVector operator +(const tVector v) const;
```
Retorna o vetor resultante da soma de **this** com o vetor `v`. Se **this** e `v` não possuem o mesmo número de elementos, uma exceção da classe `tXMath` é gerada.

Veja também Classe `tXMath`

operator –
```
tVector operator -(const tVector v) const;
```
Retorna o vetor resultante da subtração de **this** com o vetor `v`. Se **this** e `v` não possuírem o mesmo número de elementos, uma exceção da classe `tXMath` é gerada.

Veja também Classe `tXMath`

operator *
```
tVector operator *(const tMatrix m) const;
```
Retorna o vetor resultante da multiplicação de **this** com a matriz `m`. Se o número de elementos de **this** for diferente do número de linhas de `m`, uma exceção da classe `tXMath` é gerada.

Veja também Classe `tXMath`

```
double operator *(const tVector v) const;
```
Retorna o produto interno de **this** com o vetor `v`. Se **this** e `v` não possuírem o mesmo número de elementos, uma exceção da classe `tXMath` é gerada.

Veja também Classe `tXMath`

```
tVector operator *(double s) const;
```
Retorna o vetor resultante da multiplicação de **this** com o escalar `s`.

operator +=
```
tVector operator +=(const tVector v);
```
Soma **this** com o vetor `v`, armazena o resultado em **this** e retorna **this**. Se **this** e `v` não possuírem o mesmo número de elementos, uma exceção da classe `tXMath` é gerada.

Veja também Classe `tXMath`

operator –=
```
tVector operator -=(const tVector v);
```
Subtrai o vetor `v` de **this**, armazena o resultado em **this** e retorna **this**. Se **this** e `v` não possuírem o mesmo número de elementos, uma exceção da classe `tXMath` é gerada.

Veja também Classe `tXMath`

operator *=
```
tVector operator *=(double s);
```

Multiplica **this** pelo escalar `s`, armazena o resultado em **this** e retorna **this**.

operator ==
```
bool operator ==(const tVector v) const;
```
Operador de comparação (*shallow-equal*). Retorna **true** se **this** e `v` possuírem o mesmo corpo de vetor; retorna **false** caso contrário.

Veja também `tVector::IsEqual`

operator !=
```
bool operator !=(const tVector v) const;
```
Retorna **true** se **this** e `v` possuírem corpos de vetor diferentes; retorna **false** caso contrário.

Veja também `tVector::IsEqual`

operator ()
```
double operator ()(int i) const;
```
Retorna, como um *rvalue*, o `i`-ésimo elemento de **this**. Um *rvalue* é um valor que pode aparecer somente no lado direito de uma expressão de atribuição.

```
double& operator ()(int i);
```
Retorna, como um *lvalue*, o `i`-ésimo elemento de **this**. Um *lvalue* é um valor que pode aparecer no lado esquerdo de uma expressão de atribuição.

Classe tVectorExtractor — warp.h

Um objeto da classe `tVectorExtractor` é um filtro de extração do campo de deslocamentos de um modelo mecânico. A entrada do filtro é um modelo de decomposição por células; a saída do filtro é o mesmo modelo de decomposição por células de entrada. O filtro não transforma a geometria ou a topologia do modelo de entrada; somente extrai o vetor de deslocamentos dos graus de liberdade de cada nó do modelo de entrada e armazena a informação no atributo `Vector` do próprio nó. Um objeto da classe `tVectorExtractor` é um filtro de transformação de atributos, como visto no Capítulo 7.

Veja também `tVertex::Vector`, classes `tNode`, `tMesh`

Construtor Público

Construtor
```
inline tVectorExtractor();
```
Construtor *default*. Inicializa a entrada e a saída do filtro.

Métodos Públicos

Execute
```
void Execute();
```
Executa o filtro.

GetInput
```
inline tMesh* GetInput();
```
Retorna um ponteiro para o modelo mecânico de entrada de **this**.

GetOutput
```
inline tMesh* GetOutput();
```
Retorna um ponteiro para o modelo mecânico de saída de **this**.

SetInput
```
void SetInput(tMesh* mesh);
```
Torna `mesh` o modelo de entrada de **this**.

Classe tVertex — model.h

Um objeto da classe `tVertex` é um vértice global de OSW. Um vértice global é definido por um identificador, posição espacial, cor, valor escalar e valor vetorial, sem quaisquer outras informações topológicas características de um modelo geométrico específico. Os modelos geométricos de OSW implementam *iterators* próprios de vértices globais, conforme discutimos no Capítulo 9.

Veja também Classe `tModel`

Construtores Públicos

Construtores
```
inline tVertex();
```
Construtor *default*. Sem funcionalidade.

```
inline tVertex(const t3DVector& p);
```
Inicializa a posição de **this** com o vetor `p`.

```
inline tVertex(const tVertex& v);
```
Construtor de cópia. Inicializa os atributos de **this** com uma cópia dos atributos de `v`.

Atributos Públicos

Color
```
tColor Color;
```
Cor de **this**.

Veja também Classe `tColor`

Number
```
int Number;
```
Inteiro identificador de **this**,

Position
```
t3DVector Position;
```
Posição de **this**.

Veja também Classe `t3DVector`

Scalar
```
double Scalar;
```
Valor de um campo escalar na posição de **this**. O valor do campo é usualmente definido por um filtro da classe `tScalarExtractor` e utilizado, por exemplo, por um filtro da classe `tContourFilter` para geração de isolinhas do modelo ao qual **this** pertence.

Veja também Classes `tScalarExtractor`, `tContourFIlter`

Vector
`t3DVector Vector;`
Valor de um campo vetorial na posição de **this**, usualmente definido por um filtro da classe `tVectorExtractor` e utilizado, por exemplo, por um filtro da classe `tWarpFilter` para geração da estrutura deformada do modelo ao qual **this** pertence.

Veja também Classes `tVectorExtractor`, `tWarpFIlter`

Método Público

Transform
```
inline void Transform(const t3DTransfMatrix& m);
```
Transforma a posição de **this** com a transformação `m`.

Classe tView — view.h

Um objeto da classe `tView` é um elemento de interface homem-documento. Uma vista tem dois propósitos:

1. Capturar os comandos de entrada do usuário. Uma vista possui uma tabela de comandos que define a interface da vista com o usuário. Cada comando da tabela de comandos da vista é implementado por um método da classe da vista. Os comandos de uma vista podem adicionar, alterar ou eliminar atores, luzes, câmeras ou quaisquer outros dados específicos da cena a qual a vista é associada.

2. Exibir imagens da cena. Uma vista possui um objeto da classe `tRenderer` responsável pela síntese de imagens da cena a qual a vista é associada.

Veja também Classes `tScene`, `tCamera` e `tRenderer`

Construtor e Destrutor Públicos

Construtor
```
tView(tScene& scene, tCamera* camera = 0);
```
Inicializa a vista pertencente à cena `scene`. A imagem da cena exibida pela cena é tomada pela câmera `*camera`. Se `camera` for igual a zero, constrói uma câmera *default* para a vista.

Destrutor
```
virtual ~tView();
```
Destrutor virtual.

Métodos Públicos

GetCamera
```
inline tCamera* GetCamera();
```
Retorna um ponteiro para a câmera utilizada por **this**.

GetName
```
virtual const char* GetName() const;
```
Retorna o nome da vista. Em `tView`, o nome da vista é igual ao nome da câmera da vista.

GetRenderer
```
inline tRenderer* GetRenderer();
```
Retorna um ponteiro para o *renderer* que gera as imagens da cena de **this**.

CmAddLight
```
virtual void CmAddLight();
```
Comando padrão de adição de uma luz à cena de **this**.

CmAzimuth
```
virtual void CmAzimuth();
```
Comando padrão de azimute da câmera de **this**.

CmDeleteLight
```
virtual void CmDeleteLight();
```
Comando padrão de remoção de luzes de **this**.

CmElevation
```
virtual void CmElevation();
```
Comando padrão de elevação da câmera de **this**.

CmPitch
```
virtual void CmPitch();
```
Comando padrão de arfagem da câmera de **this**.

CmRedraw
```
virtual void CmRedraw();
```
Comando padrão de atualização da imagem de **this**.

CmRoll
```
virtual void CmRoll();
```
Comando padrão de rolagem da câmera de **this**.

CmSelectLight
```
tLight* CmSelectLight();
```
Comando padrão de seleção de luzes da cena de **this**.

CmSetAmbientLight
```
void CmSetAmbientLight();
```
Comando padrão de ajuste da luz ambiente da cena de **this**.

CmSetBackground
```
void CmSetBackground();
```
Comando padrão de ajuste da cor de fundo da cena de **this**.

CmYaw
```
virtual void CmYaw();
```
Comando padrão de ginada da câmera de **this**.

CmZoom
```
virtual void CmZoom();
```
Comando padrão de *zoom* da câmera de **this**.

SetCamera
```
void SetCamera(tCamera* camera);
```

Torna `camera` a câmera de **this**. Se `camera` for igual a zero, cria uma câmera padrão para **this**.

Atributos Protegidos

Camera
`tCamera* Camera;`
Ponteiro para a câmera utilizada por **this**.

Renderer
`tRenderer* Renderer;`
Ponteiro para o *renderer* utilizado por **this**.

Scene
`tScene* Scene;`
Ponteiro para a cena proprietária de **this**.

Métodos Protegidos

GetAngle
```
bool GetAngle(double& a, const char* m);
```
Função de entrada de ângulo utilizada em um comando de vista. Exibe a mensagem `m` na janela de comandos e executa a *thread* de requisição de um ângulo, armazenado em `a`. Retorna **true** se a operação foi bem sucedida; retorna **false** se o comando foi cancelado pelo usuário.

GetColor
```
bool GetColor(tColor& c, const char* m);
```
Função de entrada de cor utilizada em um comando de vista. Exibe a mensagem `m` na janela de comandos e executa a *thread* de requisição de uma cor, armazenada em `c`. Retorna **true** se a operação foi bem sucedida; retorna **false** se o comando foi cancelado pelo usuário.

GetDistance
```
bool GetDistance(double& d, const char* m);
```
Função de entrada de distância utilizada em um comando de vista. Exibe a mensagem `m` na janela de comandos e executa a *thread* de requisição de uma distância, armazenada em `d`. Retorna **true** se a operação foi bem sucedida; retorna **false** se o comando foi cancelado pelo usuário.

```
bool GetDistance(double& d, const char* m, t3DVector& b);
```
Função de entrada de distância utilizada em um comando de vista. Exibe a mensagem `m` na janela de comandos e executa a *thread* de requisição de uma distância, armazenada em `d`. A distância é definida em relação ao ponto base `b`. Retorna **true** se a operação foi bem sucedida; retorna **false** se o comando foi cancelado pelo usuário.

GetPoint
```
bool GetPoint(t3DVector& p, const char* m);
```
Função de entrada de ponto utilizada em um comando de vista. Exibe a mensagem `m` na janela de comandos e executa a *thread* de requisição de um ponto, armazenado em `p`. Retorna **true** se a operação foi bem sucedida; retorna **false** se o comando foi cancelado pelo usuário.

ConstructRenderer
`virtual tRenderer* ConstructRenderer();`
Construtor "virtual" de *renderer* da vista. Em `tView`, retorna um objeto da classe `tScanner`.

Classe tViewport — viewport.h

Um objeto da classe `tViewport` é uma área retangular definida por dois pontos P1 e P2, tomados em relação a um sistema de coordenadas de dispositivo.

Construtores Públicos

`inline tViewport();`
Construtor *default*.

`inline tViewport(const tViewport& v);`
Construtor de cópia. Inicializa as coordenadas de P1 e P2 a partir das coordenadas de P1 e P2 de `v`.

`inline tViewport(const tDCPoint& p1, const tDCPoint& p2);`
Inicializa P1 com as coordenadas de `p1` e P2 com as coordenadas de `p2`.

Veja também Classe `tDCPoint`

`inline tViewport(int x1, int y1, int x2, int y2);`
Inicializa P1 com as coodenadas `x1` e `y1` e P2 com as coordenadas `x2` e `y2`.

Atributos Públicos

P1
`tDCPoint P1;`
Ponto, em coordenadas de dispositivo, do canto superor esquerdo da *viewport*.

P2
`tDCPoint P2;`
Ponto, em coordenadas de dispositivo, do canto inferior direito da *viewport*.

Métodos Públicos

Center
`inline tDCPoint Center() const;`
Retorna o centro da *viewport*.

Contains
`bool Contains(const tDCPoint& point) const;`
Retorna **true** se **this** contém o ponto linha `point`.

`bool Contains(const tDCPoint& p1, const tDCPoint& p2) const;`
Retorna **true** se **this** contém a linha definida pelos pontos `p1` e `p2`.

`bool Contains(const tViewport& v) const;`
Retorna **true** se **this** contém a *viewport* `v`.

Height
```
inline int Height() const;
```
Retorna a altura H da *viewport*.

Veja também `tViewport::Width`

Intersects
```
bool Intersects(const tDCPoint& point) const;
```
Retorna **true** se o ponto `point` intercepta **this**; retorna **false** caso contrário.

```
bool Intersects(const tDCPoint& p1, const tDCPoint& p2) const;
```
Retorna **true** se a linha definida pelos pontos `p1` e `p2` intercepta **this**; retorna **false** caso contrário. Utiliza o algoritmo de recorte de Cohen-Sutherland.

```
bool Intersects(const tViewport& v) const;
```
Retorna **true** se a *viewport* `v` intercepta **this**; retorna **false** caso contrário.

IsEmpty
```
inline bool IsEmpty() const;
```
Retorna **true** se *viewport* for vazia; retorna **false** caso contrário.

Veja também `tViewport::SetEmpty`

Normalize
```
void Normalize();
```
Ajusta as coordenadas de P1 e P2 tal que P1 e P2 correspondam, respectivamente, ao canto superior esquerdo e ao canto inferior direito da *viewport*.

SetCorners
```
inline void SetCorners(const tViewport& v);
```
Ajusta as coordenadas de P1 e P2 com as coordenadas P1 e P2 de `v`.

```
void SetCorners(const tDCPoint& p1, const tDCPoint& p2);
```
Ajusta P1 com as coodenadas de `p1` e P2 com as coordenadas de `p2`. Normaliza a *viewport*.

Veja também `tViewport::Normalize`

```
void SetCorners(int x1, int y1, int x2, int y2);
```
Ajusta P1 com as coodenadas `x1` e `y1` e P2 com as coordenadas `x2` e `y2`. Normaliza a *viewport*.

Veja também `tViewport::Normalize`

SetEmpty
```
void SetEmpty();
```
Ajusta as coordenadas de P1 e P2 tal que **this** seja uma *viewport* vazia.

Veja também `tViewport::IsEmpty`

Width
```
inline tDCPoint Width() const;
```
Retorna o largura W da *viewport*.

Veja também `tViewport::Height`

Operadores Públicos

operator =
```
inline tViewport& operator =(consttViewport& v);
```
Operador de cópia. Ajusta as coordenadas de P1 e de P2 com as coordenadas de P1 e de P2 de `v`. Retorna uma referência para **this**.

operator +
```
tViewport operator +(const tViewport& v) const;
```
Retorna uma *viewport* resultante da expansão de **this** com `v`.

operator +=
```
tViewport& operator +=(const tViewport& v);
```
Expande **this** a partir de `v`; retorna uma referência para **this**.

operator ==
```
inline bool operator ==(const tViewport& v) const;
```
Retorna **true** se P1 e P2 são iguais a P1 e P2 de `v`; retorna **false** caso contrário.

operator !=
```
inline bool operator !=(const tViewport& v) const;
```
Retorna **true** se P1 é diferente de P1 de `v` ou se P2 é diferente de P2 de `v`; retorna **false** caso contrário.

Classe tWarpFilter — warp.h

Um objeto da classe `tWarpFilter` representa um filtro de geração da malha deformada de um estrutura. A entrada do filtro é a malha de elementos do modelo mecânico da estrutura. A saída do filtro é um modelo gráfico contendo a malha deformada.

Veja também Classe `tMesh`, classe `tGraphicModel`

Construtor Público

Construtor
```
tWarpFilter();
```
Construtor *default*.

Métodos Públicos

Execute
```
tGraphicModel* Execute();
```
Executa o filtro, se existir o modelo de entrada. Retorna um ponteiro para o modelo gráfico resultante do processo.

GetScale
```
inline double GetScale() const;
```
Retorna o valor de escala utilizado por **this** para deformar a malha do modelo de entrada.

SetInput
```
void SetInput(tMesh* mesh);
```
Torna `mesh` o modelo de entrada de **this**.

SetScale
```
void SetScale(double value);
```
Ajusta o valor de escala de **this** com o valor `value`.

Classe tWriter — writer.h

A classe `tWriter` é uma classe abstrata que descreve a estrutura e a funcionalidade de um escritor genérico (definimos *escritor* no Capítulo 7). A classe declara o método virtual puro privado `Run()`, responsável pela execução do escritor. O método deve ser sobrecarregado nas classes concretas derivadas de `tWriter`.

Construtor e Destrutor Públicos

Construtor
```
tWriter(const char* fileName);
```
Inicializa o nome do arquivo de escrita de **this** com `fileName`.

Destrutor
```
virtual ~tWrite();
```
Destrutor virtual. Libera a memória utilizada para armazenar o nome do arquivo de saída de **this**.

Método Público

Execute
```
void Execute();
```
Abre o arquivo de saída de **this** com o nome de arquivo dado no construtor. Executa o método virtual privado `Run()` e fecha o arquivo de saída de **this**.

Atributos Protegidos

File
```
FILE* File;
```
Ponteiro para o arquivo de saída de **this**.

FileName
```
char* FileName;
```
Ponteiro da cadeia de caracteres que contém o nome do arquivo de saída de **this**.

Classe tXMath — pmath.h

Um objeto da classe `tXMath` representa uma exceção gerada por algum objeto OSW durante a execução de cálculos matemáticos. Um objeto da classe `tFullSystem`, por exemplo, gera uma exceção do tipo `tXMath` se, durante a resolução do sistema, a matriz dos coeficientes for singular. A classe deriva da classe `xmsg`.

Veja também Classes `xmsg` (C++), `tFullSystem`

Construtor Público

Construtor
```
tXMath(const string& msg);
```
Inicializa o objeto base `xmsg` com o parâmetro `msg`.

Classe tXSolver — solver.h

Um objeto da classe `tXSolver` representa uma exceção gerada por um objeto de uma classe derivada da classe abstrata `tSolver` durante a análise numérica de um modelo mecânico. A classe deriva da classe `xmsg`.

Veja também Classe `xmsg` (C++)

Construtor Público

Construtor
```
tXSolver(const string& msg);
```
Inicializa o objeto base `xmsg` com o parâmetro `msg`.

10.4 Definições de Tipos Globais de OSW

Nesse Seção apresentamos, em ordem alfabética, as definições de tipo (**`typedef`**s) globais de OSW.

Tipo t3FNode — node.h

```
typedef tNodeT<3> t3FNode;
```
Nós de um modelo mecânico com 3 graus de liberdade.

Veja também Classes `tNodeT` , `tBE4NQuad`

Tipo t6FNode — node.h

```
typedef tNodeT<6> t3FNode;
```
Nós de um modelo mecânico com 6 graus de liberdade.

Veja também Classes `tNodeT` , `t3NShell`

Tipo tCellIterator — cell.h

```
typedef tDoubleListIterator<tCell> tCellIterator;
```
Iterator de células de um modelo de decomposição por células.

Veja também Classes `tDoubleListIterator` , `tCell` , `tMesh`

Tipo tDCPoint — viewport.h

```
typedef TPoint tDCPoint;
```
Ponto em coordenadas de dispositivo (X, Y).

Veja também Classe `TPoint` (OWL)

Tipo tEdgeIterator — model.h

```
typedef tIterator<tEdge> tEdgeIterator;
```
Iterator de arestas de um modelo genérico.

Veja também Classes `tIterator` , `tEdge`

Tipo tFaceIterator — model.h

typedef `tIterator<tFace> tFaceIterator;`
Iterator de faces de um modelo genérico.

Veja também Classes `tIterator` , `tFace`

Tipo tInternalEdgeIterator — model.h

typedef `tInternalIterator<tEdge> tInternalEdgeIterator;`
Iterator interno de arestas de um modelo genérico.

Veja também Classes `tInternalIterator` , `tEdge`

Tipo tInternalFaceIterator — model.h

typedef `tInternalIterator<tFace> tInternalFaceIterator;`
Iterator interno de faces de um modelo genérico.

Veja também Classes `tInternalIterator` , `tFace`

Tipo tInternalVertexIterator — model.h

typedef `tInternalIterator<tVertex> tInternalVertexIterator;`
Iterator interno de vértices de um modelo genérico.

Veja também Classes `tInternalIterator` , `tVertex`

Tipo tNodeIterator — node.h

typedef `tDoubleListIterator<tNode> tNodeIterator;`
Iterator de nóss de um modelo de decomposição por células.

Veja também Classes `tDoubleListIterator` , `tNode` , `tMesh`

Tipo tPrimitiveIterator — gmodel.h

typedef `tDoubleListIterator<tPrimitive> tCellIterator;`
Iterator de primitivos de um modelo gráfico.

Veja também Classes `tDoubleListIterator` , `tPrimitive`

Tipo tSourcePointIterator — belement.h

typedef `tIterator<tSourcePoint> tSourcePointIterator;`
Iterator de pontos fonte de um objeto da classe `tBESolver`.

Veja também Classes `tIterator` , `tSourcePoint` , `tBESolver`

Tipo tVertexIterator — model.h

typedef `tIterator<tVertex> tVertexIterator;`
Iterator de vértices de um modelo genérico.

Veja também Classes `tIterator` , `tVertex`

CAPÍTULO 11

Exemplos

11.1 OSW-Shell

OSW-Shell[1] é um programa OSW de modelagem de estruturas constituídas de cascas delgadas. Em OSW-Shell, uma casca é geometricamente representada por um modelo de cascas, tal como definido no Capítulo 3. As malhas dos modelos mecânicos da estrutura, automaticamente geradas a partir do modelo geométrico, são constituídas de elementos finitos triangulares de casca, descritos no Capítulo 6. A interface de OSW-Shell é ilustrada na Figura 11.1.

11.1.1 Classes de OSW-Shell

Nessa Seção descreveremos as classes específicas de OSW-Shell, as quais são derivadas das classes das bibliotecas de classes de OSW. Descreveremos, também, os comandos das vistas da aplicação.

Classe tGeoScene — oswshell.h

Um objeto da classe `tGeoScene` representa uma cena de um modelo geométrico de cascas de OSW-Shell. `tGeoScene` deriva diretamente da classe `tScene`. A cena pertence a um documento da classe `tShellDoc` e mantém uma referência para o modelo de cascas manipulado pela cena.

Veja também Classes `tShellDoc`, `tShell`

Construtor Público

Construtor
`inline tGeoScene(tShellDoc& doc);`
Inicializa o objeto base `tScene` com `doc`.

[1] O roteiro de desenvolvimento de OSW-Shell foi apresentado no Capítulo 9.

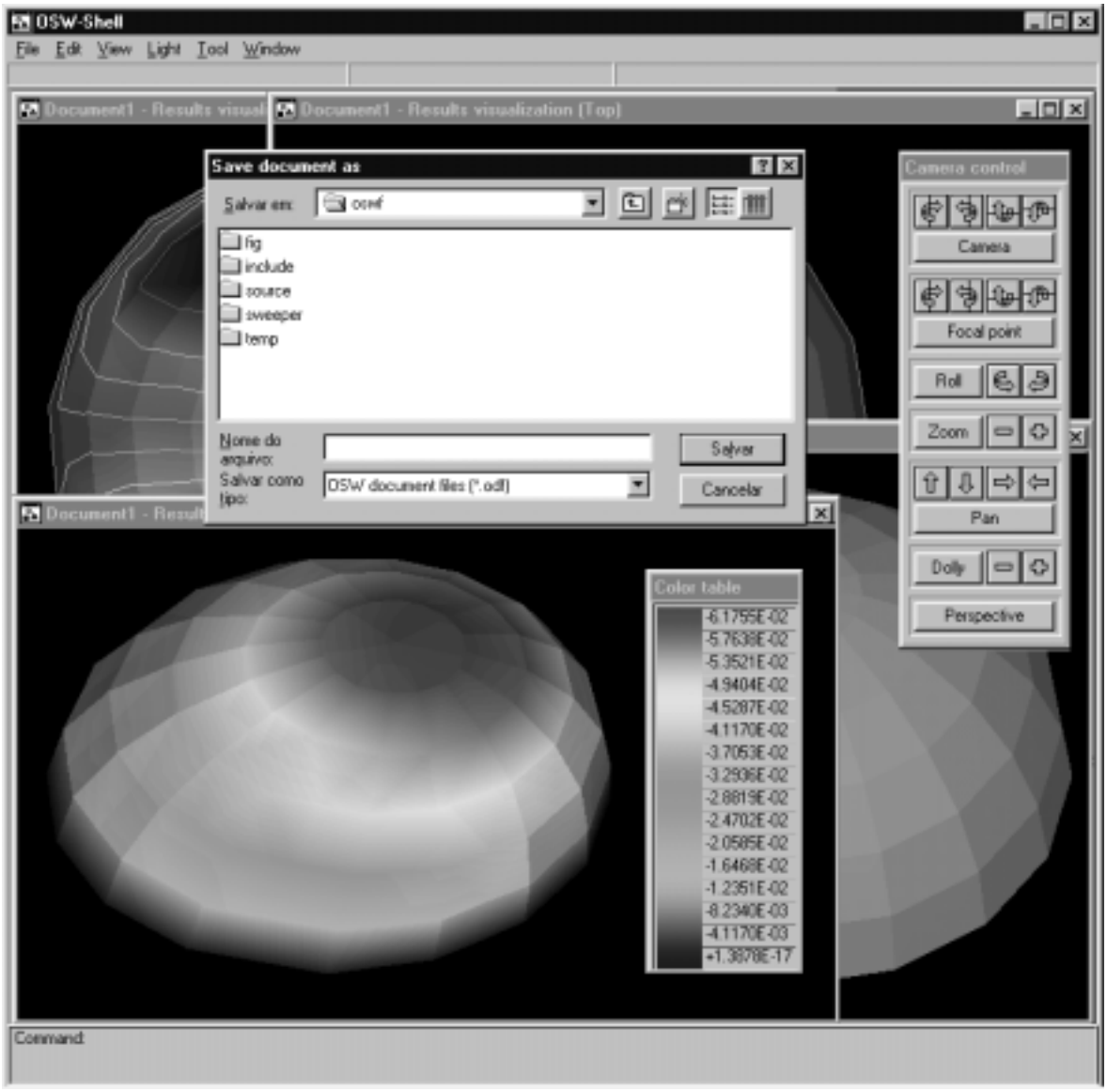


Figura 11.1: Interface de OSW-Shell.

Métodos Públicos

GetSceneName
`const char* GetSceneName() const;`
Retorna o nome da cena. Método virtual declarado na classe base `tScene`.

Veja também Classe `tScene`

GetShell
`inline tShell* GetShell();`
Retorna um ponteiro para o modelo de cascas manipulado pela cena.

Veja também Classe `tShell`

SetShell
`inline void SetShell(tShell* shell);`
Adiciona o modelo de cascas `shell` ao documento da cena. O documento contém

somente um modelo de cascas. Antes de adicionar `shell` ao documento e criar um ator para `shell` em **this**, o método destrói o modelo de cascas do documento, se houver.

Veja também `tScene::AddActor`

Classe tGeoView — oswshell.h

Um objeto da classe `tGeoView` é uma vista de uma cena da classe `tGeoScene`. A vista pode responder a comandos de leitura de um modelo de cascas, de manipulação de vértices e faces de um modelo de cascas e de geração de malhas de elementos finitos a partir do modelo geométrico de uma casca. A classe deriva diretamente da classe `tPolyView`.

Veja também Classes `tGeoScene`, `tPolyView`

Construtor Público

Construtor
```
inline tGeoView(tGeoScene& scene, tCamera* camera);
```
Inicializa o objeto base `tPolyView` com `scene` e `camera`.

Método Público

GetScene
```
inline tGeoScene* GetScene();
```
Retorna um ponteiro para a cena de **this**.

Comandos

box
Cria o modelo de cascas de uma caixa. Solicita a entrada, em uma das janelas de vista da cena de **this**, de um retângulo correspondente à face da caixa no plano de trabalho da vista. Em seguida, solicita a entrada de uma distância correspondente à altura da caixa. O modelo é criado por varredura translacional e adicionado à cena de **this**.

Veja também Classe `tShellSweeper`

create_face
Solicita a seleção, nas vistas da cena de **this**, de um vértice do modelo de cascas manipulado pela cena de **this**. Cria uma face da classe `tShell::tFace` com o vértice selecionado. Em seguida, solicita a seleção dos demais vértices que definem o laço externo da face criada. Adiciona a face ao modelo manipulado pela cena de **this**.

Veja também Classe `tShell::tFace`

create_vertex
Solicita a entrada de um ponto e cria um vértice da classe `tShell::tVertex` no ponto dado. O vértice é adicionado ao modelo de cascas manipulado pela cena de **this**.

Veja também Classe `tShell::tVertex`

cylinder
Cria o modelo de cascas de um cilindro. Solicita a entrada, em uma das janelas de vista da cena de **this**, de uma circunferência correspondente à base do cilindro no plano de trabalho da vista. Em seguida, solicita a entrada de uma distância correspondente à altura do cilindro. O modelo é criado por varredura translacional e adicionado à cena de **this**.

Veja também Classe `tShellSweeper`

cylindrical_shell
Cria o modelo de cascas de uma casca cilíndrica. Solicita a entrada, em uma das janelas de vista da cena de **this**, de um arco correspondente à geratriz da casca cilíndrica. Em seguida, solicita a entrada de uma distância correspondente à altura da casca. O modelo é criado por varredura translacional e adicionado à cena de **this**.

Veja também Classe `tShellSweeper`

delete
Remove o modelo de cascas manipulado pela cena de **this** da cena.

delete_face
Solicita a seleção, nas vistas da cena de **this**, de uma lista de faces do modelo de cascas manipulado pela cena de **this** e remove as faces selecionadas do modelo.

delete_vertex
Solicita a seleção, nas vistas da cena de **this**, de uma lista de vértices do modelo de cascas manipulado pela cena de **this** e remove os vértices selecionados do modelo. As faces incidentes nos vértices selecionados também são removidas.

dome
Cria o modelo de cascas de um domo. Solicita a entrada de um ponto correspondente ao centro do domo e, em seguida, a entrada de uma distância correspondente ao raio do domo. O plano da base do domo é o plano de trabalho da vista corrente da cena de **this**. O modelo é criado por varredura rotacional e adicionado à cena de **this**.

Veja também Classe `tShellSweeper`

generate_mesh
Executa uma caixa de diálogo solicitando o tamanho do elemento da malha a ser gerada. Cria um filtro da classe `tFEShellMeshGenerator`. Ajusta o modelo de entrada do filtro com o modelo de cascas manipulado pela cena de **this**. Ajusta o tamanho do elemento do filtro com o tamanho do elemento capturado na caixa de diálogo e executa o filtro. Se o filtro executar com sucesso, cria uma cena da classe `tMecScene` e adiciona o modelo mecânico de saída do filtro à cena criada. Se ocorrer algum erro durante a geração da malha, uma exceção da classe `xmsg` é gerada.

Veja também Classes `tFEShellMeshGenerator`, `tMecScene`, `xmsg` (C++)

mirror
Solicita, nas vistas da cena de **this**, a entrada de dois pontos correspondentes ao traço de um plano perpendicular ao plano de trabalho da vista corrente da cena de **this**. Aplica uma transformação de reflexão no modelo de cascas manipulado pela cena de **this** no plano dado.

Veja também Classe `t3DTransfMatrix`

move
Solicita, nas vistas da cena de **this**, a entrada de um vetor de deslocamento. Translada o modelo de cascas manipulado pela cena de **this** de acordo com o vetor de deslocamento dado.

Veja também Classe `t3DTransfMatrix`

move_face
Solicita a seleção, nas vistas da cena de **this**, de uma face do modelo de cascas manipulado pela cena de **this**. Solicita a entrada de um vetor de deslocamento e translada os vértices da face selecionada de acordo com o vetor de deslocamento dado.

Veja também Classe `t3DTransfMatrix`

move_vertex
Solicita a seleção, nas vistas da cena de **this**, de um vértice do modelo de cascas manipulado pela cena de **this**. Solicita a entrada de um vetor de deslocamento e translada o vértice selecionado de acordo com o vetor de deslocamento dado.

Veja também Classe `t3DTransfMatrix`

read
Executa uma caixa de diálogo solicitando o nome do arquivo com a descrição de um modelo da classe `tShell`. Cria um leitor da classe `tShellReader`, passando como parâmetro o nome do arquivo capturado na caixa de diálogo. Se o leitor executar com sucesso, adiciona o modelo de cascas de saída do leitor à cena de **this**.

Veja também Classes `tShellReader`, `tShell`

rotate
Solicita a entrada de um ponto correspondente ao centro de rotação. Em seguida, solicita a entrada de um ângulo correspondente ao ângulo de rotação. A direção de rotação é dada pelo VPN da vista corrente da cena de **this**. Rotaciona o modelo de cascas manipulado pela cena de **this** de acordo com os parâmetros dados.

Veja também Classe `t3DTransfMatrix`

scale
Solicita a entrada de um ponto correspondente ao centro base da transformação de escala. Em seguida, solicita a entrada de um ponto correspondente aos valores de escala x,y,z da transformação de escala. Aplica a transformação de escala no modelo de cascas manipulado pela cena de **this** de acordo com os parâmetros dados.

Veja também Classe `t3DTransfMatrix`

set_material
Solicita a seleção, nas vistas da cena de **this**, de uma lista de faces do modelo de cascas manipulado pela cena de **this**. Solicita a seleção de um material no editor de materiais e atribui o material às faces selecionadas.

Veja também Classes `tMaterial`, `tMaterialEditor`

set_tickness
Solicita a seleção, nas vistas da cena de **this**, de uma lista de faces do modelo de cascas manipulado pela cena de **this**. Executa uma caixa de diálogo solicitando a espessura da face e atribui a espessura às faces selecionadas.

sphere
Cria o modelo de cascas de uma esfera. Solicita a entrada de um ponto correspondente ao centro da esfera e, em seguida, a entrada de uma distância correspondente ao raio da esfera. O plano do equador da esfera é o plano de trabalho da vista corrente da cena de **this**. O modelo é criado por varredura rotacional e adicionado à cena de **this**.

Veja também Classe `tShellSweeper`

write
Executa uma caixa de diálogo solicitando o nome do arquivo no qual será armazenada a descrição do modelo de cascas manipulado pela cena de **this**. Cria um escritor da classe `tShellWriter`, passando como parâmetro o nome do arquivo capturado na caixa de diálogo. Ajusta o modelo de entrada do escritor com o modelo de cascas manipulado pela cena de **this** e executa o escritor.

Veja também Classe `tShellWriter`

Classe tMecScene — oswshell.h

Um objeto da classe `tMecScene` representa uma cena de um modelo mecânico de cascas de OSW-Shell. `tMecScene` deriva diretamente da classe `tScene`. A cena pertence a um documento da classe `tShellDoc` e mantém uma referência para um modelo mecânico manipulado pela cena.

Veja também Classes `tShellDoc`, `tShellMesh`

Construtor Público

Construtor
```
inline tMecScene(tShellDoc& doc);
```
Inicializa o objeto base `tScene` com `doc`.

Métodos Públicos

Veja também Classe `tScene`

GetMesh
```
inline tMesh* GetMesh();
```
Retorna um ponteiro para o modelo mecânico manipulado pela cena.

Veja também Classe `tMesh`

GetSceneName
```
const char* GetSceneName() const;
```
Retorna o nome da cena. Método virtual declarado na classe base `tScene`.

SetMesh
```
inline void SetMesh(tMesh* mesh);
```

Adiciona o modelo mecânico `mesh` ao documento da cena e cria um ator para `mesh`. O documento pode conter vários modelos mecânicos, mas **this** manipula somente o modelo mecânico `mesh`.

Veja também `tScene::AddActor`

Classe tMecView — oswshell.h

Um objeto da classe `tMecView` é uma vista de uma cena da classe `tMecScene`. A vista pode responder a comandos de leitura de um modelo mecânico de uma casca, de especificação da vinculação e do carregamento de um modelo mecânico de uma casca e de análise elastostática de um modelo mecânico de uma casca pelo método dos elementos finitos. A classe deriva das classes `tMeshView` e `tDataView`.

Veja também Classes `tMecScene`, `tMeshView`, `tDataView`

Construtor Público

Construtor
`inline tMecView(tMecScene& scene, tCamera* camera);`
Inicializa os objetos base `tMeshView` e `tDataView` com `scene` e `camera`.

Veja também Classes `tMeshView`, `tDataView`

Método Público

GetScene
`inline tMecScene* GetScene();`
Retorna um ponteiro para a cena de **this**.

Comandos

delete_cell
Solicita a seleção, nas vistas da cena de **this**, de uma lista de células do modelo mecânico de cascas manipulado pela cena de **this** e remove as células selecionadas do modelo.

delete_node
Solicita a seleção, nas vistas da cena de **this**, de uma lista de nós do modelo mecânico manipulado pela cena de **this** e remove os nós selecionados do modelo. As células incidentes nos nós selecionados também são removidas.

fix
Solicita a seleção, nas vistas da cena de **this**, de uma lista de nós do modelo mecânico de cascas manipulado pela cena de **this**. Fixa todos os graus de liberdade especificados com o comando `select_dofs` dos nós selecionados do modelo.

Veja também Comando `select_dofs`, classe `tDOF`

load_cell
Solicita a seleção, nas vistas da cena de **this**, de uma lista de células do modelo mecânico de cascas manipulado pela cena de **this**. Executa uma caixa de diálogo solicitando as coordenadas de um ponto e ajusta o vetor de cargas das células selecionadas com as coordenadas do ponto capturado na caixa de diálogo.

Veja também Classe `t3NShell`

read
Executa uma caixa de diálogo solicitando o nome do arquivo com a descrição de um modelo mecânico da classe `tShellMesh`. Cria um leitor da classe `tShellMeshReader`, passando como parâmetro o nome do arquivo capturado na caixa de diálogo. Se o leitor executar com sucesso, adiciona o modelo mecânico de saída do leitor à cena de **this**.

Veja também Classes `tShellMeshReader`, `tShellMesh`

release
Solicita a seleção, nas vistas da cena de **this**, de uma lista de nós do modelo mecânico de cascas manipulado pela cena de **this**. Libera todos os graus de liberdade dos nós selecionados do modelo.

Veja também Comando `select_dofs`, classe `tDOF`

select_dofs
Executa uma caixa de diálogo solicitando a especificação de quais graus de liberdade dos nós do modelo mecânico manipulado pela cena de **this** serão fixados pelo comando `fix`.

Veja também Comando `fix`

set_material
Solicita a seleção, nas vistas da cena de **this**, de uma lista de células do modelo mecânico manipulado pela cena de **this**. Solicita a seleção de um material no editor de materiais e atribui o material às células selecionadas.

Veja também Classes `tMaterial`, `tMaterialEditor`

set_tickness
Solicita a seleção, nas vistas da cena de **this**, de uma lista de células do modelo mecânico manipulado pela cena de **this**. Executa uma caixa de diálogo solicitando a espessura da célula e atribui a espessura às células selecionadas.

solve
Cria um *solver* da classe `tFESolver`. Ajusta o modelo de entrada do *solver* com o modelo mecânico manipulado pela cena de **this**. Se o *solver* executar com sucesso, cria uma cena da classe `tResScene` e adiciona o modelo de saída do *solver* à cena criada.

Veja também Classes `tFESolver`, `tResScene`

unload_cells
Solicita a seleção, nas vistas da cena de **this**, de uma lista de células do modelo mecânico de cascas manipulado pela cena de **this**. Zera o vetor de cargas das células selecionadas do modelo.

Veja também Classe `t3NShell`

write
Executa uma caixa de diálogo solicitando o nome do arquivo no qual será armazenada a descrição do modelo mecânico manipulado pela cena de **this**. Cria um escritor da classe `tMeshWriter`, passando como parâmetro o nome do arquivo capturado na caixa

de diálogo. Ajusta o modelo de entrada do escritor com o modelo mecânico manipulado pela cena de **this**. Executa o escritor.

Veja também Classe `tMeshWriter`

Classe tShellApp — oswshell.h

Um objeto da classe `tShellApp` é uma aplicação OSW de modelagem de cascas, ou seja, OSW-Shell. A classe é diretamente derivada da classe `tApplication`.

Veja também Classe `tApplication`

Construtor Público

Construtor
`inline tShellApp();`
Construtor *default*. Inicializa OSW-Shell.

Veja também `tApplication::Run`

Classe tShellDoc — oswshell.h

Um objeto da classe `tShellDoc` representa um documento de OSW-Shell. O documento pode conter um modelo geométrico de cascas, vários modelos mecânicos e vários modelos gráficos resultantes do processo de visualização dos resultados de análise, além das cenas correspondentes. A classe é diretamente derivada de `tDocument`.

Veja também Classe `tDocument`

Construtor Público

Construtor
`inline tShellDoc(tShellApp& app);`
Inicializa o objeto base `tDocument` com `app`.

Classe tShellMeshReader — oswfread.h

Um objeto da classe `tShellMeshReader` é um leitor de modelos mecânicos de cascas (vimos a definição de *leitor* no Capítulo 7). Um modelo mecânico de cascas é constituído de elementos finitos triangulares da classe `t3NShell`, cujos nós possuem 6 graus de liberdade, como descrito no Capítulo 6. A classe é derivada diretamente de `tMeshReader`.

Veja também Classes `tMeshReader`, `t3NShell`

Construtor Público

Construtor
`inline tShellMeshReader(const char* fileName);`
Inicializa o objeto base `tMeshReader` com `fileName`.

Classe tResScene oswshell.h

Um objeto da classe `tResScene` representa uma cena de pós-processamento de um modelo mecânico de cascas de OSW-Shell. `tResScene` deriva diretamente da classe `tMecScene`. Além de manter uma referência para um modelo mecânico manipulado pela cena, um objeto da classe `tResScene` contém atores que representam o mapa de cores, as isolinhas, a estrutura deformada e as arestas de contorno do modelo mecânico.

Veja também Classe `tMecScene`

Construtor Público

Construtor
```
inline tResScene(tShellDoc& doc);
```
Inicializa o objeto base `tMecScene` com `doc`.

Métodos Públicos

GetSceneName
```
const char* GetSceneName() const;
```
Retorna o nome da cena. Método virtual declarado na classe base `tScene`.

Veja também Classe `tScene`

Classe tResView oswshell.h

Um objeto da classe `tResView` é uma vista de uma cena da classe `tResScene`. A vista pode responder a comandos de visualização do mapa de cores, isolinhas, estrutura deformada e arestas de contorno de um modelo mecânico de cascas. A classe deriva das classes `tColorMapView` e `tDataView`.

Veja também Classes `tResScene`, `tColorMapView`, `tDataView`

Construtor Público

Construtor
```
inline tResView(tMecScene& scene, tCamera* camera);
```
Inicializa os objetos base `tColorMapView` e `tDataView` com `scene` e `camera`.

Método Público

GetScene
```
inline tResScene* GetScene();
```
Retorna um ponteiro para a cena de **this**.

Comandos

extract_scalar_field
Executa uma caixa de diálogo solicitando o campo escalar a ser extraído do modelo mecânico manipulado pela cena de **this**. Cria um filtro da classe `tScalarExtractor`, passando como parâmetro o tipo de campo escalar capturado na caixa de diálogo (deslocamento em x, y, ou z, rotação em x, y ou z, etc.). Ajusta o modelo de entrada do filtro com o modelo mecânico manipulado pela cena de **this** e executa o filtro.

Veja também Classe `tScalarExtractor`

set_contour_parms
Executa uma caixa de diálogo solicitando os parâmetros do filtro de geração de isolinhas do modelo mecânico manipulado pela cena de **this**. Os parâmetros são o valor escalar mínimo, o valor escalar máximo e o número de isolinhas geradas pelo filtro.

Veja também Comando `show_contour`, classe `tContourFilter`

set_warp_scale
Executa uma caixa de diálogo solicitando a escala utilizada por um filtro da classe `tWarpFilter` para geração da malha deformada do modelo mecânico manipulado pela cena de **this**.

Veja também Comando `show_warp`, classe `tWarpFilter`

show_boundary_edges
Exibe ou esconde o modelo gráfico que contém as arestas de contorno do modelo mecânico de cascas manipulado pela cena de **this**. O modelo gráfico com as arestas de contorno é gerado por um filtro da classe `tBoundaryEdgesFilter`.

Veja também Classe `tBoundaryEdgesFilter`

show_color_map
Exibe ou esconde o mapa de cores do campo escalar do modelo mecânico manipulado pela cena de **this**. O campo escalar deve ser extraído do modelo com o comando `extract_scalar_field`.

Veja também Comando `extract_scalar_field`

show_mesh
Exibe ou esconde o modelo mecânico de cascas manipulado pela cena de **this**. O comando permite a visualização da malha original do modelo juntamente com as imagens obtidas do pós-processamento dos resultados de análise do modelo.

show_warp
Exibe ou esconde a malha deformada do modelo mecânico manipulado pela cena de **this**. A malha deformada é gerada por um filtro da classe `tWarpFilter`. O campo vetorial de deslocamentos utilizado pelo filtro é extraído do modelo mecânico manipulado pela cena de **this** por um filtro da classe `tVectorExtractor`.

Veja também Classes `tWarpFilter`, `tVectorExtractor`

11.2 OSW-Solid

OSW-Solid é um programa OSW de modelagem de sólidos elásticos. Em OSW-Shell, um sólido é geometricamente representado por um modelo de sólidos, tal como definido no Capítulo 3. As malhas dos modelos mecânicos de um sólido, automaticamente geradas a partir do modelo geométrico, são constituídas de elementos de contorno quadrangulares planos, descritos no Capítulo 6. A interface de OSW-Solid é ilustrada na Figura 11.2.

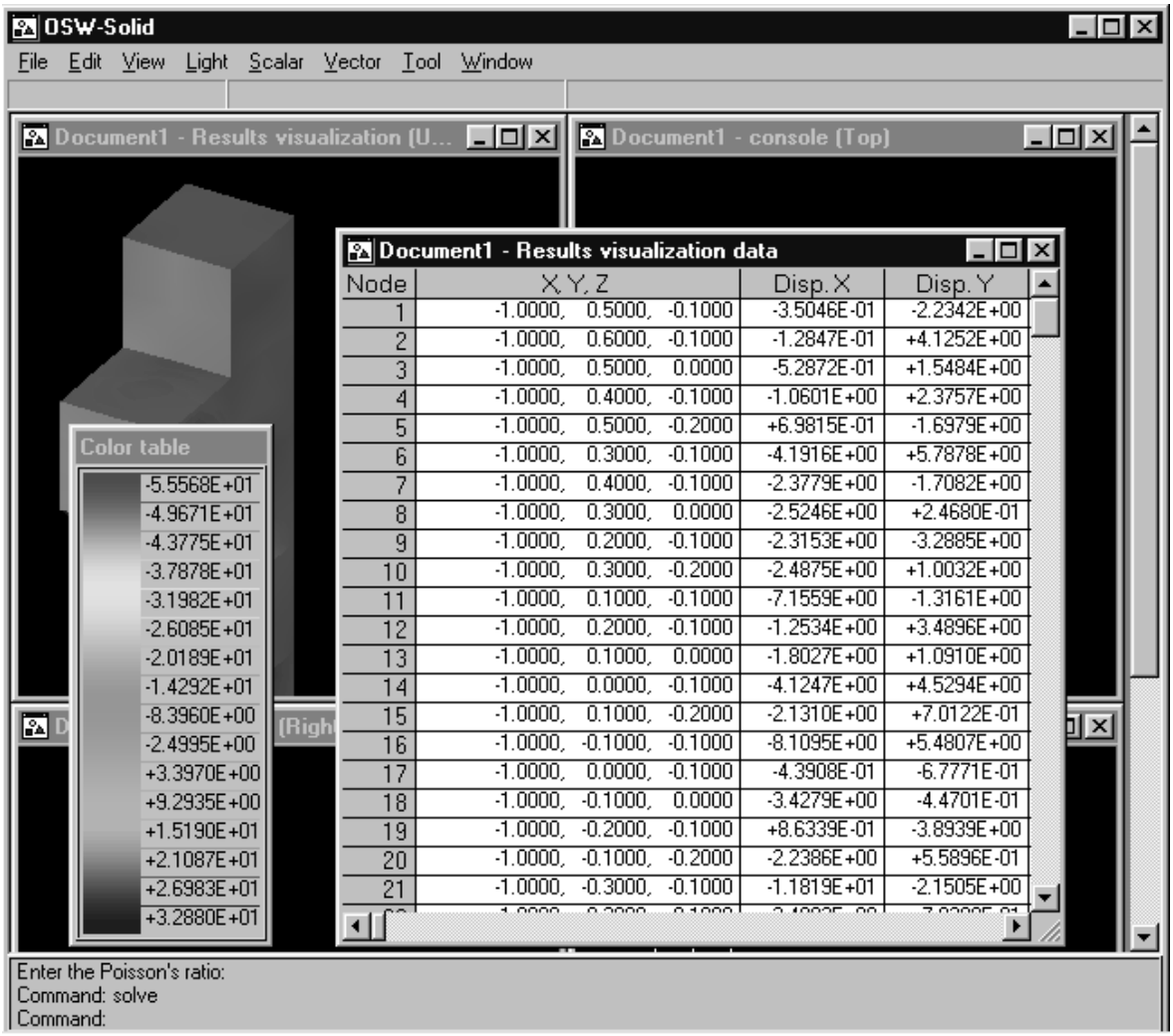


Figura 11.2: Interface de OSW-Solid.

11.2.1 Classes de OSW-Solid

Nessa Seção descreveremos as classes específicas de OSW-Solid, as quais são derivadas das classes das bibliotecas de classes de OSW. Descreveremos, também, os comandos das vistas da aplicação.

Classe tGeoScene — oswsolid.h

Um objeto da classe `tGeoScene` representa uma cena de um modelo geométrico de sólidos de OSW-Solid. `tGeoScene` deriva diretamente da classe `tScene`. A cena pertence a um documento da classe `tSolidDoc` e mantém uma referência para o modelo de sólidos manipulado pela cena.

Veja também Classes `tSolidDoc`, `tSolid`

Construtor Público

Construtor
`inline tGeoScene(tSolidDoc& doc);`
Inicializa o objeto base `tScene` com `doc`.

Métodos Públicos

GetSceneName
```
const char* GetSceneName() const;
```
Retorna o nome da cena. Método virtual declarado na classe base `tScene`.

Veja também Classe `tScene`

GetSolid
```
inline tSolid* GetSolid();
```
Retorna um ponteiro para o modelo de sólidos manipulado pela cena.

Veja também Classe `tSolid`

SetSolid
```
inline void SetSolid(tSolid* solid);
```
Adiciona o modelo de sólidos `solid` ao documento da cena. O documento contém somente um modelo de sólidos. Antes de adicionar `solid` ao documento e criar um ator para `solid` em **this**, o método destrói o modelo de sólidos do documento, se houver.

Veja também `tScene::AddActor`

Classe tGeoView — oswsolid.h

Um objeto da classe `tGeoView` é uma vista de uma cena da classe `tGeoScene`. A vista pode responder a comandos de leitura de um modelo de sólidos, de manipulação de vértices e faces de um modelo de sólidos e de geração de malhas de elementos finitos a partir do modelo geométrico de uma sólido. A classe deriva diretamente da classe `tPolyView`.

Veja também Classes `tGeoScene`, `tPolyView`

Construtor Público

Construtor
```
inline tGeoView(tGeoScene& scene, tCamera* camera);
```
Inicializa o objeto base `tPolyView` com `scene` e `camera`.

Método Público

GetScene
```
inline tGeoScene* GetScene();
```
Retorna um ponteiro para a cena de **this**.

Comandos

box
Cria o modelo de sólidos de uma caixa. Solicita a entrada, em uma das janelas de vista da cena de **this**, de um retângulo correspondente à face da caixa no plano de trabalho da vista. Em seguida, solicita a entrada de uma distância correspondente à altura da caixa. O modelo é criado por varredura translacional e adicionado à cena de **this**.

Veja também Classe `tSolidSweeper`

cylinder
Cria o modelo de sólidos de um cilindro. Solicita a entrada, em uma das janelas de vista da cena de **this**, de uma circunferência correspondente à base do cilindro no plano de trabalho da vista. Em seguida, solicita a entrada de uma distância correspondente à altura do cilindro. O modelo é criado por varredura translacional e adicionado à cena de **this**.

Veja também Classe `tSolidSweeper`

delete
Remove o modelo de sólidos manipulado pela cena de **this** da cena.

generate_mesh
Executa uma caixa de diálogo solicitando o tamanho do elemento da malha a ser gerada. Cria um filtro da classe `tBESolidMeshGenerator`. Ajusta o modelo de entrada do filtro com o modelo de sólidos manipulado pela cena de **this**. Ajusta o tamanho do elemento do filtro com o tamanho do elemento capturado na caixa de diálogo e executa o filtro. Se o filtro executar com sucesso, cria uma cena da classe `tMecScene` e adiciona o modelo mecânico de saída do filtro à cena criada. Se ocorrer algum erro durante a geração da malha, uma exceção da classe `xmsg` é gerada.

Veja também Classes `tBESolidMeshGenerator`, `tMecScene`, `xmsg` (C++)

mirror
Solicita, nas vistas da cena de **this**, a entrada de dois pontos correspondentes ao traço de um plano perpendicular ao plano de trabalho da vista corrente da cena de **this**. Aplica uma transformação de reflexão no modelo de sólidos manipulado pela cena de **this** no plano dado.

Veja também Classe `t3DTransfMatrix`

move
Solicita, nas vistas da cena de **this**, a entrada de um vetor de deslocamento. Translada o modelo de sólidos manipulado pela cena de **this** de acordo com o vetor de deslocamento dado.

Veja também Classe `t3DTransfMatrix`

move_face
Solicita a seleção, nas vistas da cena de **this**, de uma face do modelo de sólidos manipulado pela cena de **this**. Solicita a entrada de um vetor de deslocamento e translada os vértices da face selecionada de acordo com o vetor de deslocamento dado.

Veja também Classe `t3DTransfMatrix`

move_vertex
Solicita a seleção, nas vistas da cena de **this**, de um vértice do modelo de sólidos manipulado pela cena de **this**. Solicita a entrada de um vetor de deslocamento e translada o vértice selecionado de acordo com o vetor de deslocamento dado.

Veja também Classe `t3DTransfMatrix`

read
Executa uma caixa de diálogo solicitando o nome do arquivo com a descrição de um

modelo da classe `tSolid`. Cria um leitor da classe `tSolidReader`, passando como parâmetro o nome do arquivo capturado na caixa de diálogo. Se o leitor executar com sucesso, adiciona o modelo de sólidos de saída do leitor à cena de **this**.

Veja também Classes `tSolidReader`, `tSolid`

rotate
Solicita a entrada de um ponto correspondente ao centro de rotação. Em seguida, solicita a entrada de um ângulo correspondente ao ângulo de rotação. A direção de rotação é dada pelo VPN da vista corrente da cena de **this**. Rotaciona o modelo de sólidos manipulado pela cena de **this** de acordo com os parâmetros dados.

Veja também Classe `t3DTransfMatrix`

scale
Solicita a entrada de um ponto correspondente ao centro base da transformação de escala. Em seguida, solicita a entrada de um ponto correspondente aos valores de escala x,y,z da transformação de escala. Aplica a transformação de escala no modelo de sólidos manipulado pela cena de **this** de acordo com os parâmetros dados.

Veja também Classe `t3DTransfMatrix`

set_material
Solicita a seleção, nas vistas da cena de **this**, de uma lista de faces do modelo de sólidos manipulado pela cena de **this**. Solicita a seleção de um material no editor de materiais e atribui o material às faces selecionadas.

Veja também Classes `tMaterial`, `tMaterialEditor`

sphere
Cria o modelo de sólidos de uma esfera. Solicita a entrada de um ponto correspondente ao centro da esfera e, em seguida, a entrada de uma distância correspondente ao raio da esfera. O plano do equador da esfera é o plano de trabalho da vista corrente da cena de **this**. O modelo é criado por varredura rotacional e adicionado à cena de **this**.

Veja também Classe `tShellSweeper`

write
Executa uma caixa de diálogo solicitando o nome do arquivo no qual será armazenada a descrição do modelo de sólidos manipulado pela cena de **this**. Cria um escritor da classe `tSolidWriter`, passando como parâmetro o nome do arquivo capturado na caixa de diálogo. Ajusta o modelo de entrada do escritor com o modelo de sólidos manipulado pela cena de **this** e executa o escritor.

Veja também Classe `tSolidWriter`

Classe tMecScene — oswsolid.h

Um objeto da classe `tMecScene` representa uma cena de um modelo mecânico de sólidos de OSW-Solid. `tMecScene` deriva diretamente da classe `tScene`. A cena pertence a um documento da classe `tSolidDoc` e mantém uma referência para um modelo mecânico manipulado pela cena.

Veja também Classes `tSolidDoc`, `tSolidMesh`

Construtor Público

Construtor
```
inline tMecScene(tSolidDoc& doc);
```
Inicializa o objeto base `tScene` com `doc`.

Métodos Públicos

Veja também Classe `tScene`

GetMesh
```
inline tMesh* GetMesh();
```
Retorna um ponteiro para o modelo mecânico manipulado pela cena.

Veja também Classe `tMesh`

GetSceneName
```
const char* GetSceneName() const;
```
Retorna o nome da cena. Método virtual declarado na classe base `tScene`.

SetMesh
```
inline void SetMesh(tMesh* mesh);
```
Adiciona o modelo mecânico `mesh` ao documento da cena e cria um ator para `mesh`. O documento pode conter vários modelos mecânicos, mas **this** manipula somente o modelo mecânico `mesh`.

Veja também `tScene::AddActor`

Classe tMecView — oswsolid.h

Um objeto da classe `tMecView` é uma vista de uma cena da classe `tMecScene`. A vista pode responder a comandos de leitura de um modelo mecânico de uma sólido, de especificação da vinculação e do carregamento de um modelo mecânico de uma sólido e de análise elastostática de um modelo mecânico de uma sólido pelo método dos elementos finitos. A classe deriva das classes `tMeshView` e `tDataView`.

Veja também Classes `tMecScene`, `tMeshView`, `tDataView`

Construtor Público

Construtor
```
inline tMecView(tMecScene& scene, tCamera* camera);
```
Inicializa os objetos base `tMeshView` e `tDataView` com `scene` e `camera`.

Veja também Classes `tMeshView`, `tDataView`

Método Público

GetScene
```
inline tMecScene* GetScene();
```
Retorna um ponteiro para a cena de **this**.

Comandos

delete_cell
Solicita a seleção, nas vistas da cena de **this**, de uma lista de células do modelo mecânico de sólidos manipulado pela cena de **this** e remove as células selecionadas do modelo.

delete_node
Solicita a seleção, nas vistas da cena de **this**, de uma lista de nós do modelo mecânico manipulado pela cena de **this** e remove os nós selecionados do modelo. As células incidentes nos nós selecionados também são removidas.

fix
Solicita a seleção, nas vistas da cena de **this**, de uma lista de nós do modelo mecânico de sólidos manipulado pela cena de **this**. Fixa todos os graus de liberdade especificados com o comando `select_dofs` dos nós selecionados do modelo.

Veja também Comando `select_dofs`, classe `tDOF`

load_node
Solicita a seleção, nas vistas da cena de **this**, de uma lista de nós do modelo mecânico de sólidos manipulado pela cena de **this**. Executa uma caixa de diálogo solicitando as coordenadas de um ponto e ajusta o vetor de cargas dos nós selecionados com as coordenadas do ponto capturado na caixa de diálogo.

read
Executa uma caixa de diálogo solicitando o nome do arquivo com a descrição de um modelo mecânico da classe `tSolidMesh`. Cria um leitor da classe `tSolidMeshReader`, passando como parâmetro o nome do arquivo capturado na caixa de diálogo. Se o leitor executar com sucesso, adiciona o modelo mecânico de saída do leitor à cena de **this**.

Veja também Classes `tSolidMeshReader`, `tSolidMesh`

release
Solicita a seleção, nas vistas da cena de **this**, de uma lista de nós do modelo mecânico de sólidos manipulado pela cena de **this**. Libera todos os graus de liberdade dos nós selecionados do modelo.

Veja também Comando `select_dofs`, classe `tDOF`

select_dofs
Executa uma caixa de diálogo solicitando a especificação de quais graus de liberdade dos nós do modelo mecânico manipulado pela cena de **this** serão fixados pelo comando `fix`.

Veja também Comando `fix`

set_material
Solicita a seleção, nas vistas da cena de **this**, de uma lista de células do modelo mecânico manipulado pela cena de **this**. Solicita a seleção de um material no editor de materiais e atribui o material às células selecionadas.

Veja também Classes `tMaterial`, `tMaterialEditor`

solve
Cria um *solver* da classe `tBESolver`. Ajusta o modelo de entrada do *solver* com o modelo mecânico manipulado pela cena de **this**. Se o *solver* executar com sucesso, cria uma cena da classe `tResScene` e adiciona o modelo de saída do *solver* à cena criada.

Veja também Classes `tBESolver`, `tResScene`

unload_nodes
Solicita a seleção, nas vistas da cena de **this**, de uma lista de nós do modelo mecânico de sólidos manipulado pela cena de **this**. Zera o vetor de cargas dos nós selecionadas do modelo.

write
Executa uma caixa de diálogo solicitando o nome do arquivo no qual será armazenada a descrição do modelo mecânico manipulado pela cena de **this**. Cria um escritor da classe `tMeshWriter`, passando como parâmetro o nome do arquivo capturado na caixa de diálogo. Ajusta o modelo de entrada do escritor com o modelo mecânico manipulado pela cena de **this**. Executa o escritor.

Veja também Classe `tMeshWriter`

Classe tSolidApp — oswsolid.h

Um objeto da classe `tSolidApp` é uma aplicação OSW de modelagem de sólidos, ou seja, OSW-Solid. A classe é diretamente derivada da classe `tApplication`.

Veja também Classe `tApplication`

Construtor Público

Construtor
`inline tSolidApp();`
Construtor *default*. Inicializa OSW-Solid.

Veja também `tApplication::Run`

Classe tSolidDoc — oswsolid.h

Um objeto da classe `tSolidDoc` representa um documento de OSW-Solid. O documento pode conter um modelo geométrico de sólidos, vários modelos mecânicos e vários modelos gráficos resultantes do processo de visualização dos resultados de análise, além das cenas correspondentes. A classe é diretamente derivada de `tDocument`.

Veja também Classe `tDocument`

Construtor Público

Construtor
`inline tSolidDoc(tSolidApp& app);`
Inicializa o objeto base `tDocument` com `app`.

Classe tSolidMeshReader oswfread.h

Um objeto da classe `tSolidMeshReader` é um leitor de modelos mecânicos de sólidos (vimos a definição de *leitor* no Capítulo 7). Um modelo mecânico de sólidos é constituído de elementos de contorno da classe `tBE4NQuad`, cujos nós possuem 3 graus de liberdade, como descrito no Capítulo 6. A classe é derivada diretamente de `tMeshReader`.

Veja também Classes `tMeshReader`, `tBE4NQuad`

Construtor Público

Construtor
```
inline tSolidMeshReader(const char* fileName);
```
Inicializa o objeto base `tMeshReader` com `fileName`.

Classe tResScene oswsolid.h

Um objeto da classe `tResScene` representa uma cena de pós-processamento de um modelo mecânico de sólidos de OSW-Solid. `tResScene` deriva diretamente da classe `tMecScene`. Além de manter uma referência para um modelo mecânico manipulado pela cena, um objeto da classe `tResScene` contém atores que representam o mapa de cores, as isolinhas, a estrutura deformada e as arestas de contorno do modelo mecânico.

Veja também Classe `tMecScene`

Construtor Público

Construtor
```
inline tResScene(tSolidDoc& doc);
```
Inicializa o objeto base `tMecScene` com `doc`.

Métodos Públicos

GetSceneName
```
const char* GetSceneName() const;
```
Retorna o nome da cena. Método virtual declarado na classe base `tScene`.

Veja também Classe `tScene`

Classe tResView oswsolid.h

Um objeto da classe `tResView` é uma vista de uma cena da classe `tResScene`. A vista pode responder a comandos de visualização do mapa de cores, isolinhas, estrutura deformada e arestas de contorno de um modelo mecânico de sólidos. A classe deriva das classes `tColorMapView` e `tDataView`.

Veja também Classes `tResScene`, `tColorMapView`, `tDataView`

Construtor Público

Construtor
```
inline tResView(tMecScene& scene, tCamera* camera);
```
Inicializa os objetos base `tColorMapView` e `tDataView` com `scene` e `camera`.

Método Público

GetScene
`inline tResScene* GetScene();`
Retorna um ponteiro para a cena de **this**.

Comandos

extract_scalar_field
Executa uma caixa de diálogo solicitando o campo escalar a ser extraído do modelo mecânico manipulado pela cena de **this**. Cria um filtro da classe `tScalarExtractor`, passando como parâmetro o tipo de campo escalar capturado na caixa de diálogo (deslocamento ou força de superfície em x, y, ou z). Ajusta o modelo de entrada do filtro com o modelo mecânico manipulado pela cena de **this** e executa o filtro.

Veja também Classe `tScalarExtractor`

set_contour_parms
Executa uma caixa de diálogo solicitando os parâmetros do filtro de geração de isolinhas do modelo mecânico manipulado pela cena de **this**. Os parâmetros são o valor escalar mínimo, o valor escalar máximo e o número de isolinhas geradas pelo filtro.

Veja também Comando `show_contour`, classe `tContourFilter`

set_warp_scale
Executa uma caixa de diálogo solicitando a escala utilizada por um filtro da classe `tWarpFilter` para geração da malha deformada do modelo mecânico manipulado pela cena de **this**.

Veja também Comando `show_warp`, classe `tWarpFilter`

show_boundary_edges
Exibe ou esconde o modelo gráfico que contém as arestas de contorno do modelo mecânico de sólidos manipulado pela cena de **this**. O modelo gráfico com as arestas de contorno é gerado por um filtro da classe `tBoundaryEdgesFilter`.

Veja também Classe `tBoundaryEdgesFilter`

show_color_map
Exibe ou esconde o mapa de cores do campo escalar do modelo mecânico manipulado pela cena de **this**. O campo escalar deve ser extraído do modelo com o comando `extract_scalar_field`.

Veja também Comando `extract_scalar_field`

show_mesh
Exibe ou esconde o modelo mecânico de sólidos manipulado pela cena de **this**. O comando permite a visualização da malha original do modelo juntamente com as imagens obtidas do pós-processamento dos resultados de análise do modelo.

show_warp
Exibe ou esconde a malha deformada do modelo mecânico manipulado pela cena de **this**. A malha deformada é gerada por um filtro da classe `tWarpFilter`. O campo vetorial de deslocamentos utilizado pelo filtro é extraído do modelo mecânico manipulado

pela cena de **this** por um filtro da classe `tVectorExtractor`.

Veja também Classes `tWarpFilter`, `tVectorExtractor`

11.3 Figuras Coloridas

Nas páginas seguintes apresentaremos figuras coloridas de alguns resultados obtidos pela aplicação de modelagem utilizada como exemplo no Capítulo 9. As imagens foram capturadas diretamente das janelas de vistas gráficas de OSW-Shell e impressas por um *software* de manipulação de imagens. Apresentaremos, também, algumas figuras coloridas que ilustram a utilização dos elementos de interface de OSW-Shell na modelagem de uma casca simples. Mostraremos, em alguns passos, como definir o modelo geométrico de uma casca, gerar as malhas de elementos finitos, impor as condições de contorno e carregamentos às malhas geradas, analisar os modelos mecânicos e visualizar os resultados de análise.

Figuras Coloridas

Casca cilíndrica. No Capítulo 7 utilizamos uma casca cilíndrica para exemplificarmos os resultados de visualização de escalares e de vetores. No Capítulo 9 empregamos a mesma casca nas ilustrações da interface de OSW-Shell. A seguir, mostraremos algumas imagens coloridas dessa casca cilíndrica. Quando as grandezas não forem referidas a um sistema de medidas específico, utilizaremos uc para denotar unidade de comprimento e uf para denotar unidade de força.

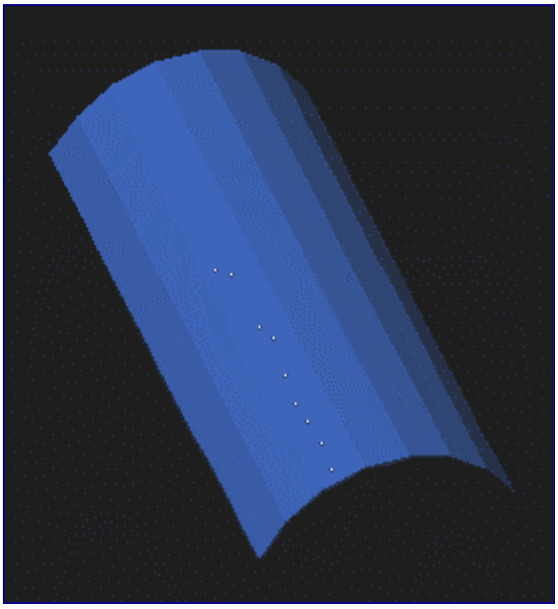


Figura Colorida 1. Modelo geométrico da casca. O modelo geométrico da casca é um modelo de cascas, tal como definido no Capítulo 3. O modelo foi gerado por varredura translacional de um arco de circunferência de raio unitário, centrado na origem, ao longo do eixo z. O ângulo interno do arco é de 120°. O comprimento da casca é igual a 4 uc. A imagem foi sintetizada por um objeto da classe `tScanner`, com tonalização *flat*. A cor da luz ambiente da cena é um tom de cinza. O azul é devido a uma luz puntual sobre o eixo z, situada a 10 uc da origem do sistema global.

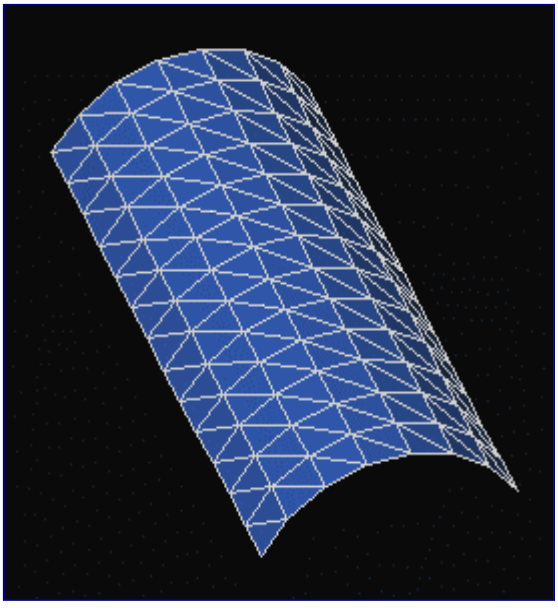


Figura Colorida 2. Modelo mecânico da casca. A geometria do modelo mecânico da casca é representada por um modelo de decomposição por células, tal como visto no Capítulo 3. O modelo foi gerado pelo processo de geração de malhas descrito no Capítulo 6. O tamanho do elemento é igual a 0,3 uc. A malha de elementos finitos do modelo possui 182 células e 112 vértices. A espessura dos elementos é 0,1 uc. O material, hipotético, possui módulo de elasticidade igual a 10.000 uf/uc^2. O coeficiente de Poisson é 0,5.

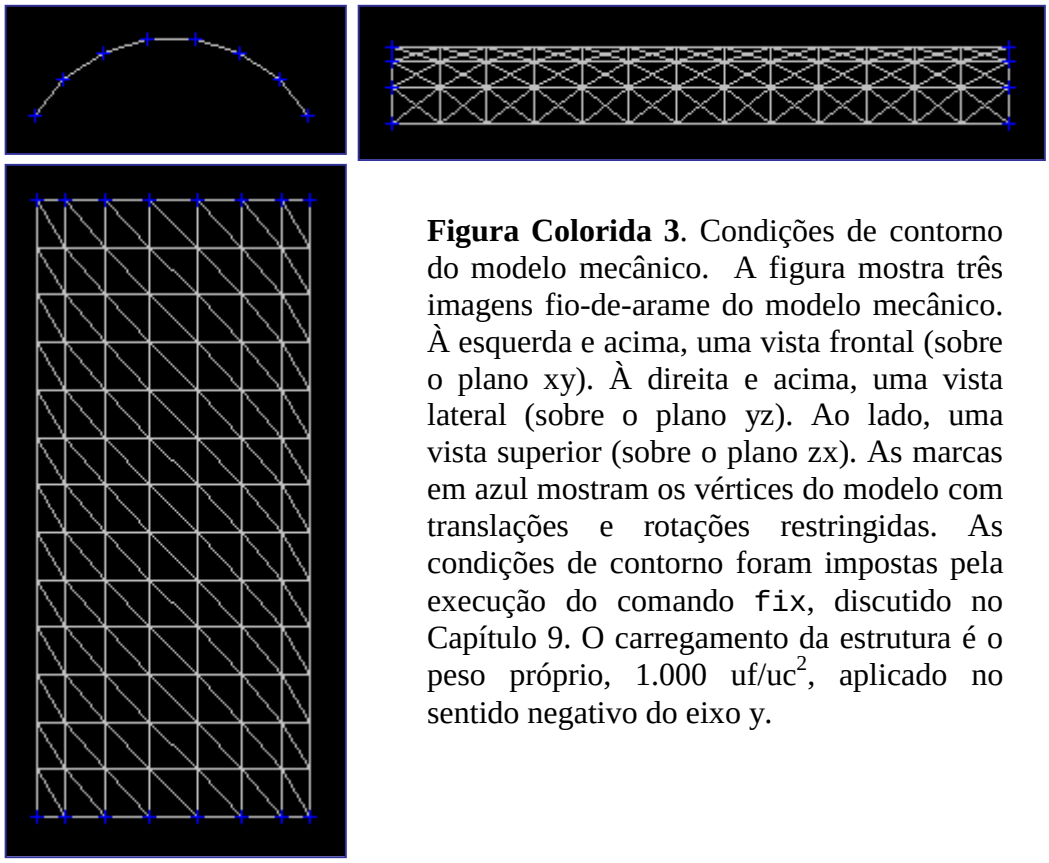


Figura Colorida 3. Condições de contorno do modelo mecânico. A figura mostra três imagens fio-de-arame do modelo mecânico. À esquerda e acima, uma vista frontal (sobre o plano xy). À direita e acima, uma vista lateral (sobre o plano yz). Ao lado, uma vista superior (sobre o plano zx). As marcas em azul mostram os vértices do modelo com translações e rotações restringidas. As condições de contorno foram impostas pela execução do comando `fix`, discutido no Capítulo 9. O carregamento da estrutura é o peso próprio, 1.000 uf/uc^2, aplicado no sentido negativo do eixo y.

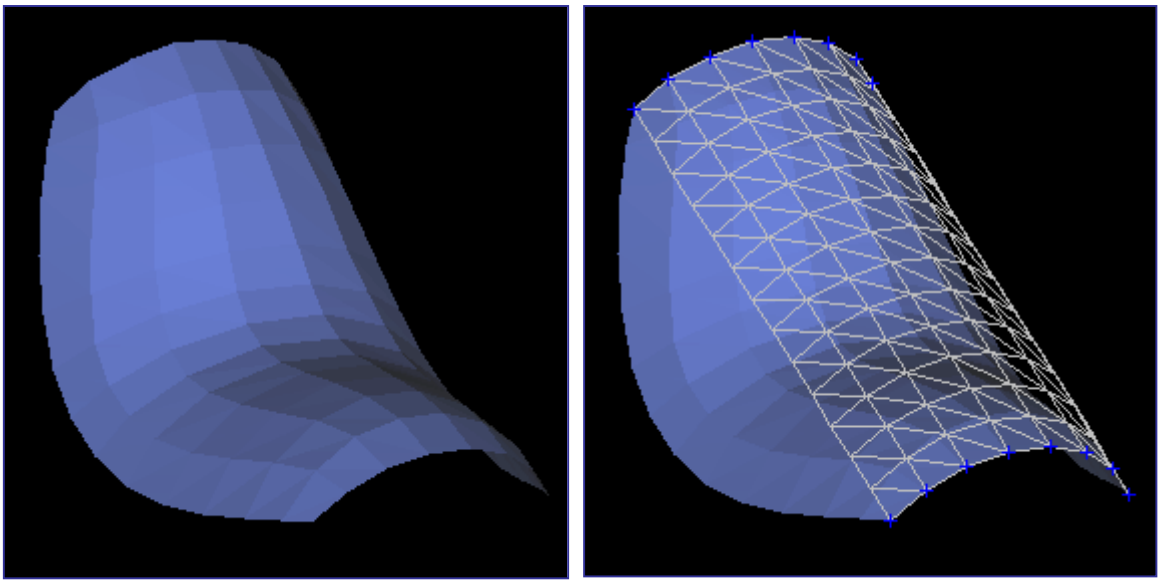


Figura Colorida 4. Estrutura deformada da casca. Após a análise numérica, um filtro da classe `tVectorExtractor` extrai o vetor de deslocamentos dos nós do modelo mecânico. Em seguida, um filtro da classe `tWarpFilter` usa o vetor de deslocamentos para gerar um modelo gráfico que representa a estrutura deformada da casca. O fator de escala utilizado no exemplo é igual a 10. À esquerda, uma imagem do modelo gráfico produzida por um objeto da classe `tScanner`, com tonalização de Gouraud. À direita, um ator adicionado à cena, exibindo a malha do modelo mecânico.

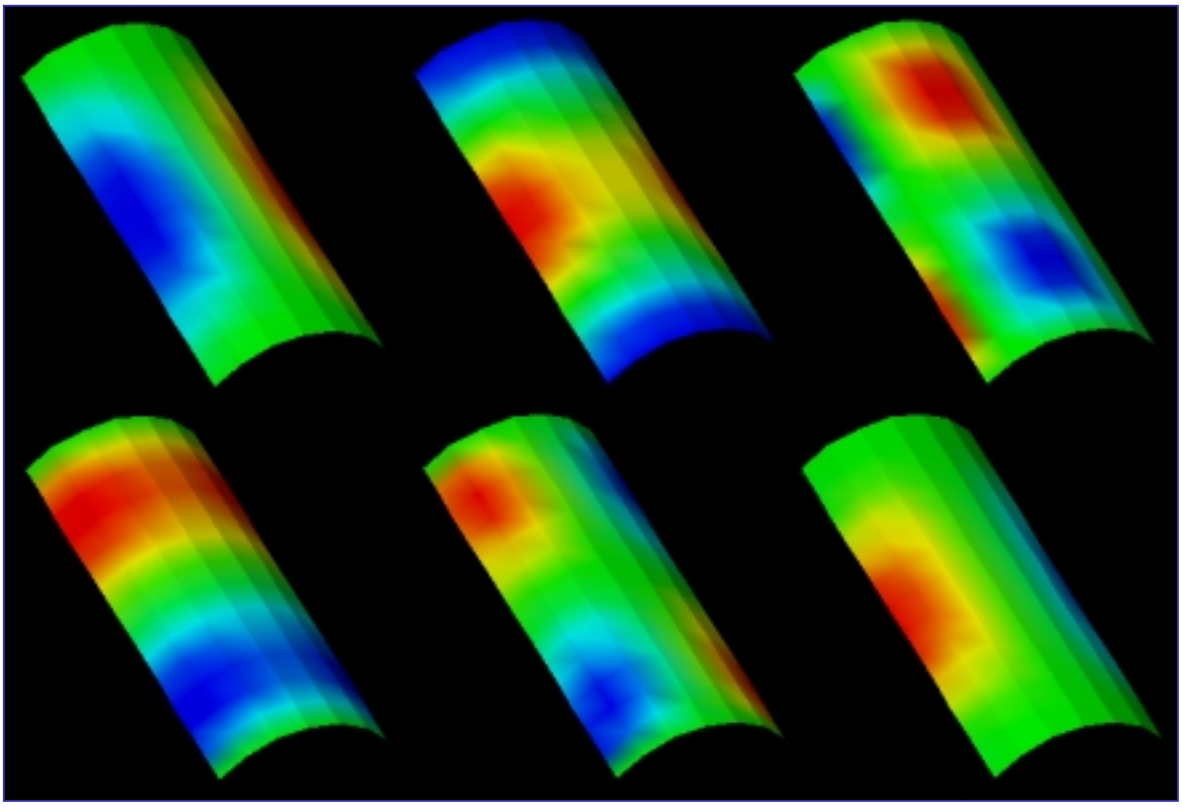


Figura Colorida 5. Mapas de cores. Embora o problema fundamental seja definido em termos de vetores e tensores, algoritmos de transformações de escalares podem ser utilizados para visualização do comportamento da casca. Da esquerda para a direita, acima, mapas de cores para os componentes x,y,z de deslocamento, respectivamente. Abaixo, da esquerda para a direita, mapas de cores para os componentes x,y,z de rotação, respectivamente.

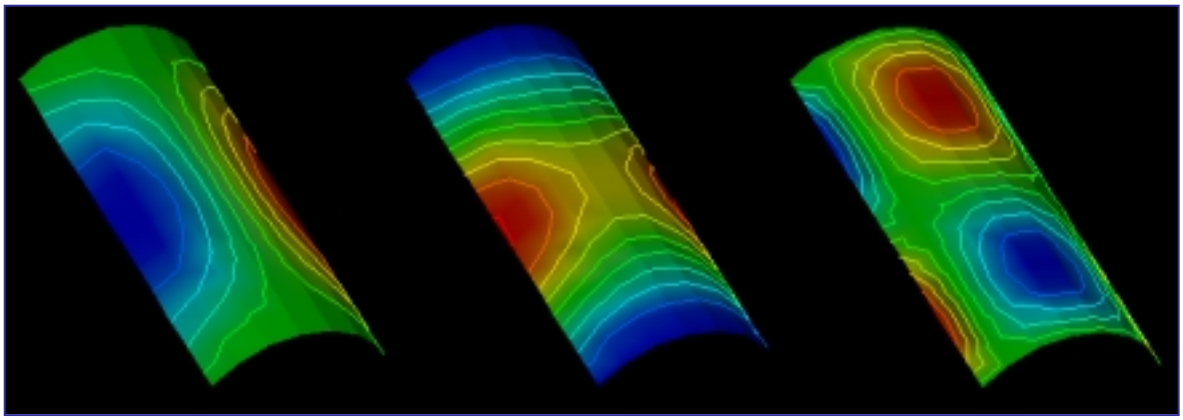


Figura Colorida 6. Mapas de cores e isolinhas. Da esquerda para a direita, mapas de cores e isolinhas para os componentes x,y,z de deslocamento. A imagem mostra que as isolinhas são “contornos” de regiões do mapa de cores. Em visualização tridimensional, as cores do mapa de cores são alteradas pela luz da cena. A figura foi gerada com uma luz puntual de cor cinza claro, para realçar as isolinhas.

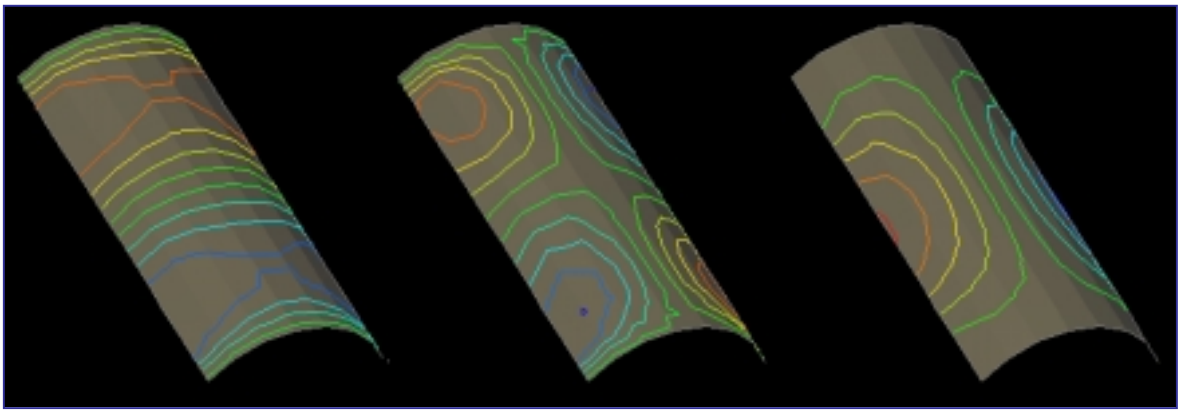


Figura Colorida 7. Mais isolinhas. Da esquerda para a direita, isolinhas dos componentes x,y,z de rotação. Nessas imagens, os mapas de cores não são exibidos. A mensagem `tMapper::UseScalars(false)` foi enviada aos mapeadores dos atores do modelo mecânico da cena. Como resultado, a cor utilizada para iluminação de um vértice do modelo não é dada pelo escalar do vértice e pela tabela de cores do mapeador, mas sim pelas propriedades materiais do objeto, de acordo com o modelo de iluminação definido no Capítulo 7.

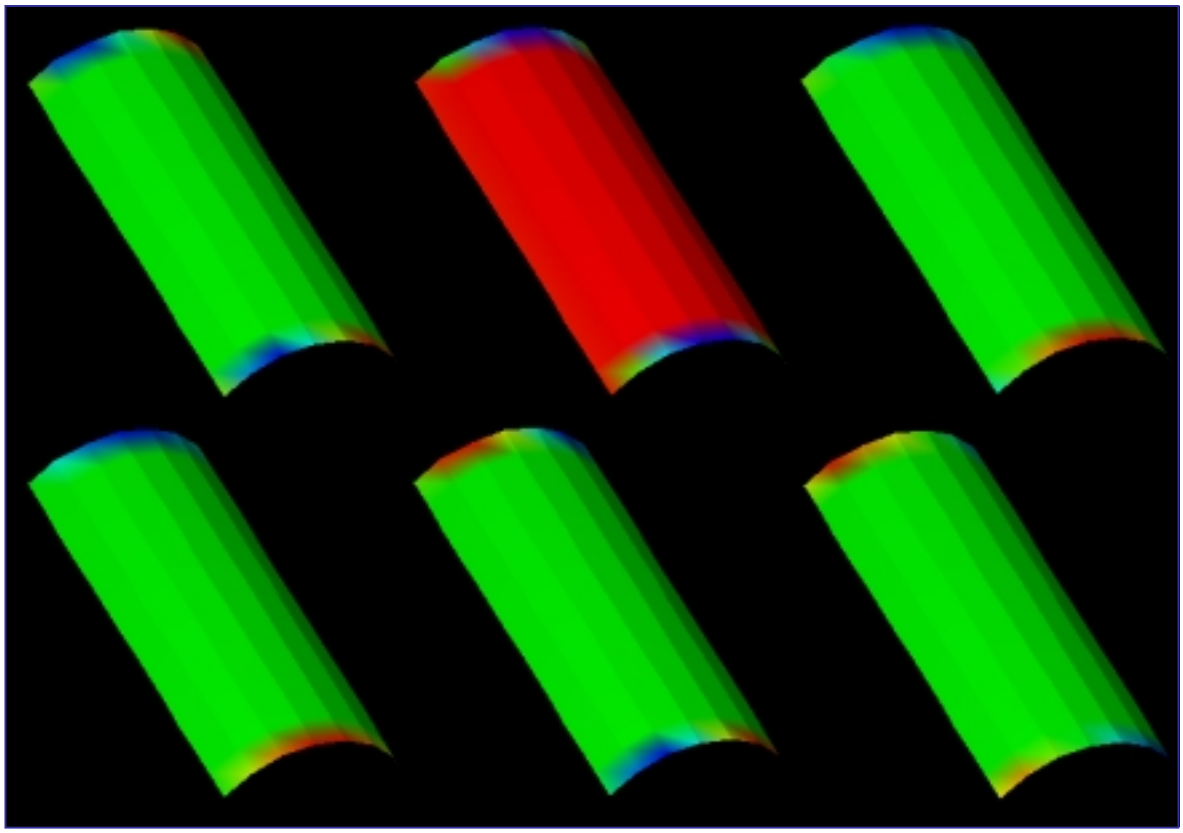


Figura Colorida 8. Mais mapas de cores. Da esquerda para a direita e de cima para baixo, mapas de cores dos componentes x,y,z de forças e componentes x,y,z de momentos, respectivamente.

Cobertura de um galpão. Os dados geométricos para esse exemplo foram obtidos de ZIENKIEWICZ [133]. Os atributos mecânicos adotados, contudo, são distintos. Duas análises foram efetuadas para a mesma malha de elementos e os mesmos carregamentos, mas para condições de contorno diferentes. Algumas imagens são apresentadas a seguir.

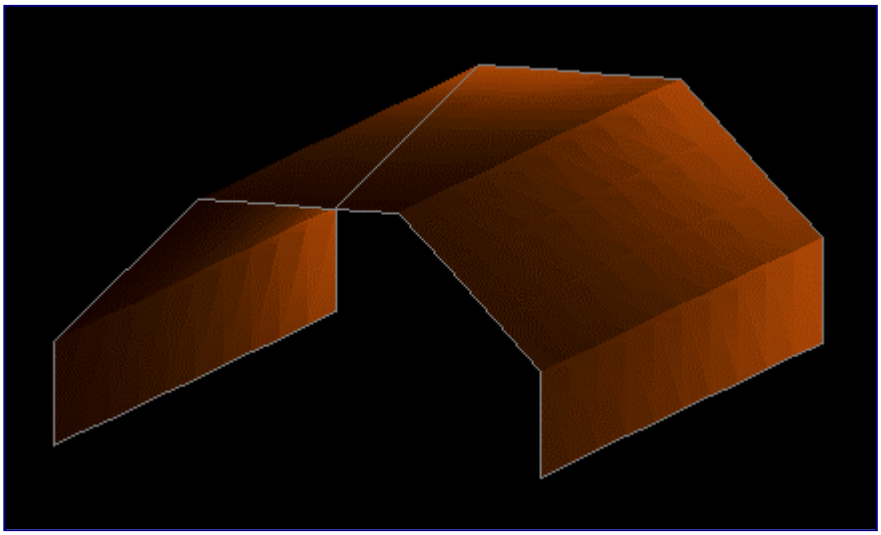


Figura Colorida 9. Modelo geométrico da cobertura. O modelo geométrico da casca foi gerado por varredura translacional de um objeto da classe `tPolyline`, definido no Capítulo 3, com 6 vértices. A cobertura possui 25,2cm de largura, 12,6cm de altura e 38,8cm de profundidade. A imagem foi sintetizada por um objeto da classe `tScanner`, com tonalização *flat*. A cena contém dois atores: um correspondente ao modelo de cascas e outro correspondente às arestas de contorno do modelo mecânico.

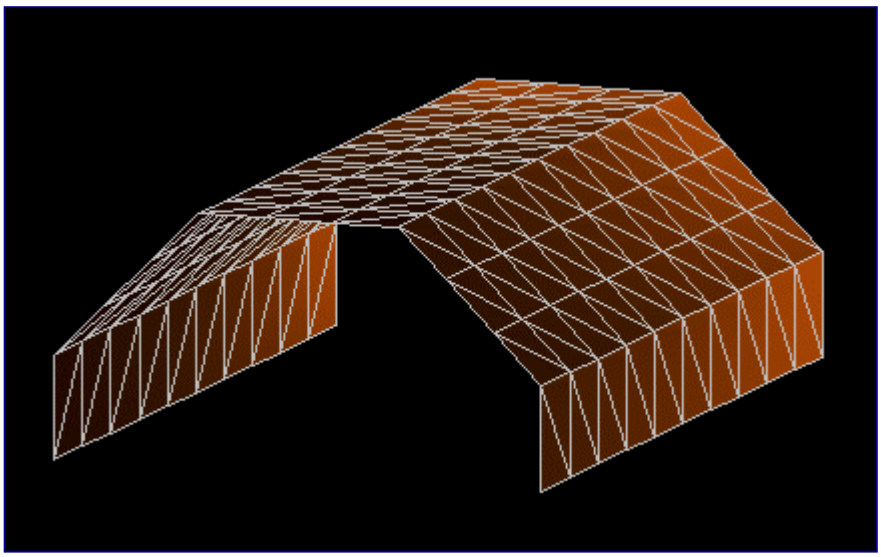


Figura Colorida 10. Modelo mecânico da casca, gerado por um objeto da classe `tFEShellMeshGenerator`. O tamanho do elemento é 4cm. A malha resultante tem 220 células e 132 nós. A espessura da casca é 0,5cm. O material tem módulo de elasticidade de 250kgf/cm^2 e coeficiente de Poisson 0,43.

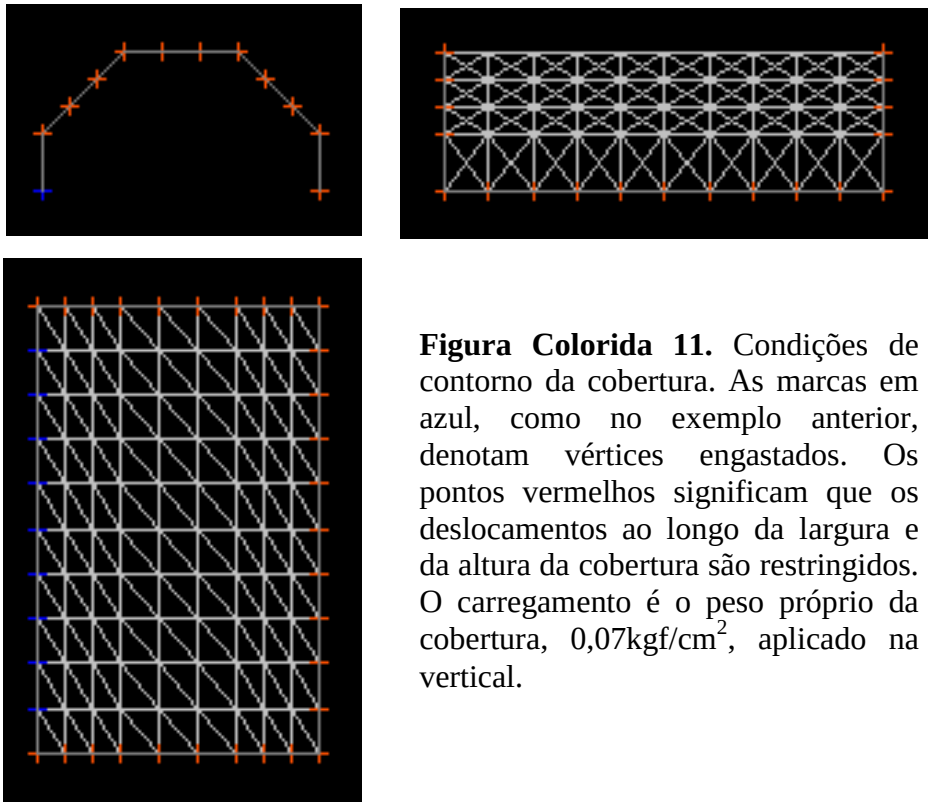


Figura Colorida 11. Condições de contorno da cobertura. As marcas em azul, como no exemplo anterior, denotam vértices engastados. Os pontos vermelhos significam que os deslocamentos ao longo da largura e da altura da cobertura são restringidos. O carregamento é o peso próprio da cobertura, 0,07kgf/cm^2, aplicado na vertical.

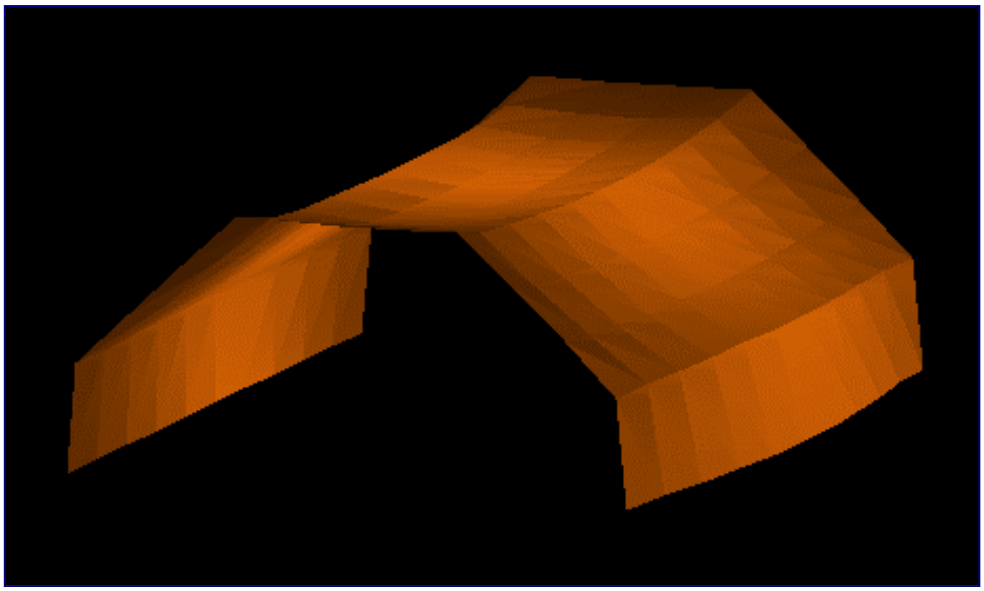


Figura Colorida 12. Estrutura deformada da cobertura. Modelo gráfico com 220 triângulos, gerado por um objeto da classe `tWarpFilter`. A imagem foi sintetizada por um *renderer* da classe `tScanner`, usando tonalização de Gouraud. As silhuetas dos triângulos, resultantes do processo, podem ser observadas na figura. Contudo, o método é comumente empregado porque o mais importante em visualização científica não é realismo visual, mas o significado dos dados de um modelo, como mostrado nessa imagem.

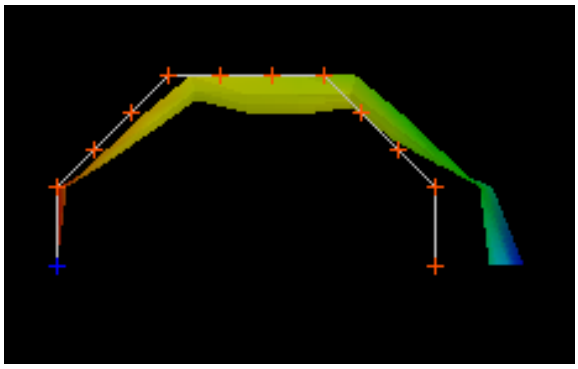


Figura Colorida 13. *Warping* e mapa de cores. Os atores de uma cena podem compartilhar os modelos do documento da aplicação. Como cada ator possui seu próprio `tMapper`, um modelo pode ser exibido de diferentes formas em uma cena, ou em cenas distintas. Na figura acima, a estrutura deformada é colorida com o mapa de cores do deslocamento horizontal da cobertura.

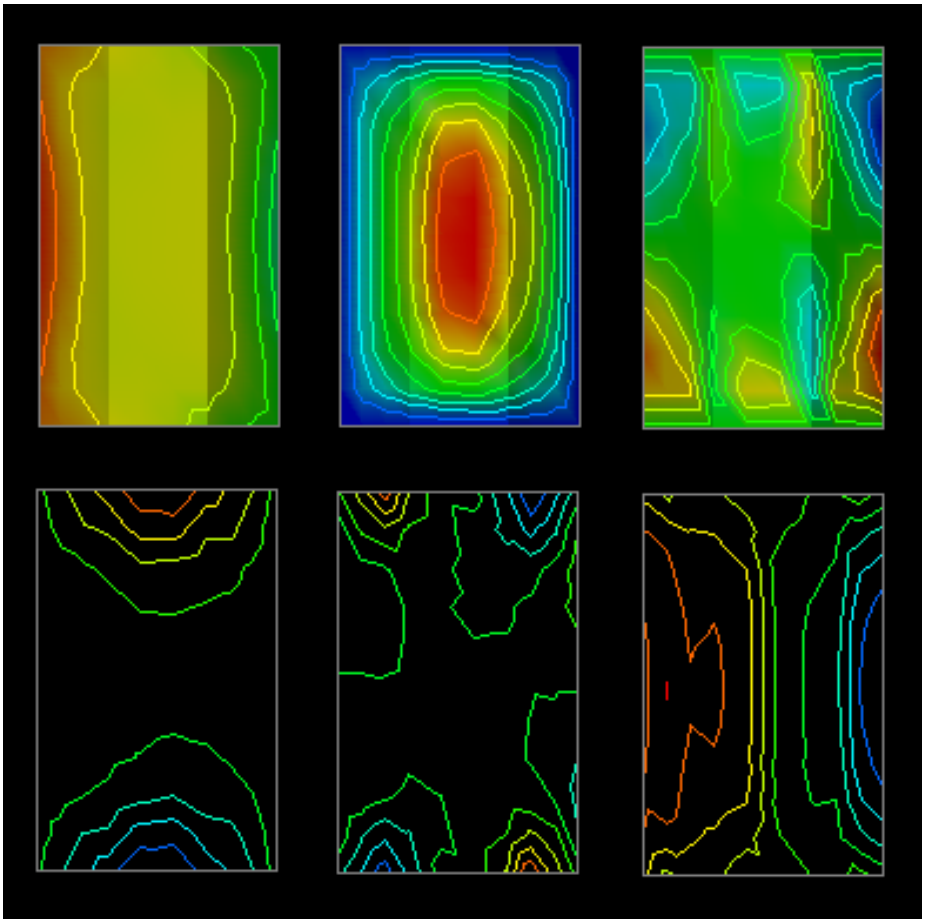


Figura Colorida 14. Isolinhas. Da esquerda para a direita, acima, isolinhas e mapas de cores dos componentes x,y,z de deslocamento, respectivamente. Da esquerda para a direita, abaixo, arestas de contorno do modelo mecânico da cobertura e isolinhas dos componentes z,y,z de rotação, respectivamente.

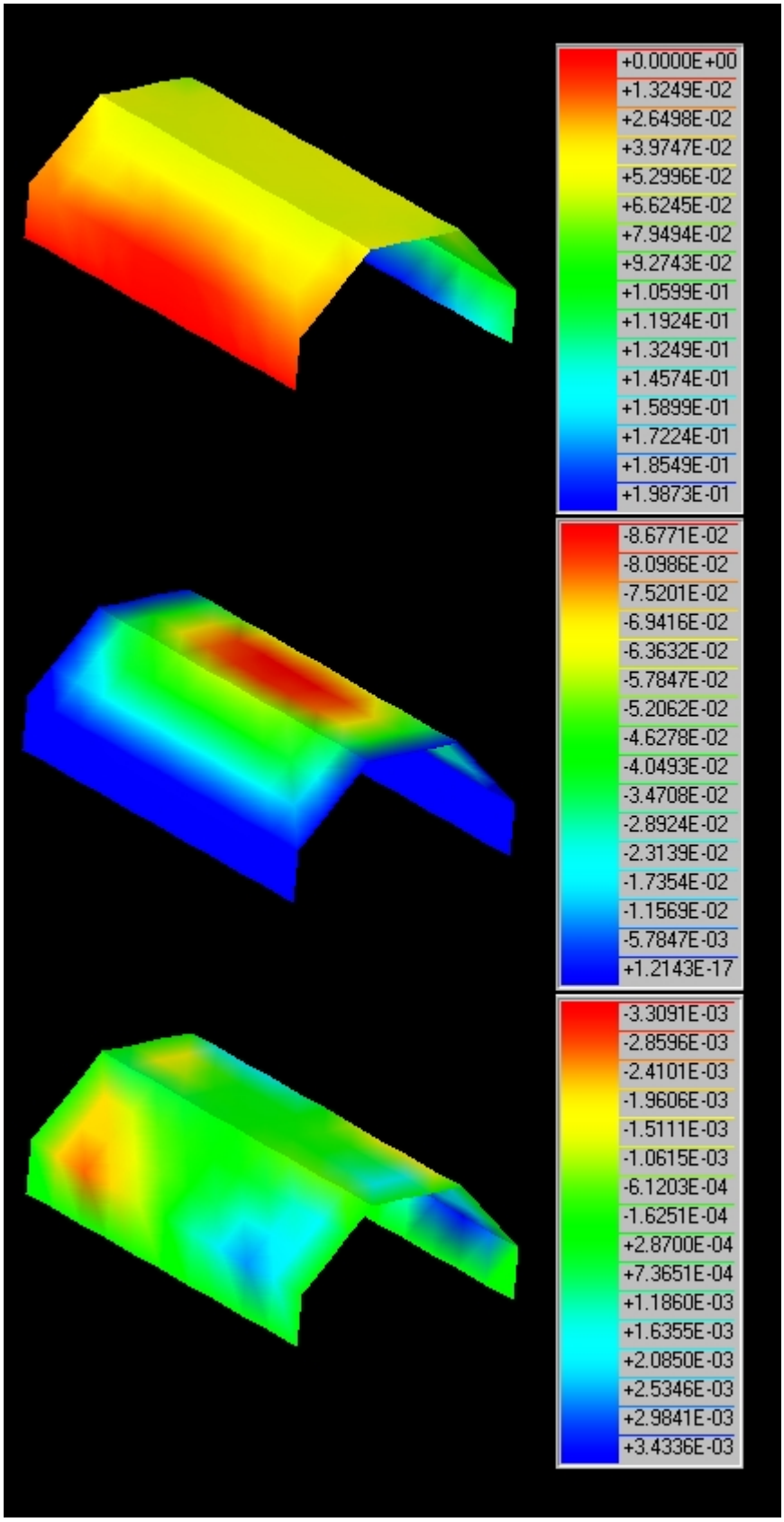


Figura Colorida 15. Janelas de tabelas de cores. De cima para baixo, à esquerda, mapas de cores para os componentes x,y,z de deslocamento, respectivamente. À direita, as tabelas de cores correspondentes. Uma tabela de cores é um objeto da classe `tLookupTable`. Uma tabela de cores padrão possui 256 cores, do vermelho até o azul. As cores são especificadas no modelo HSV, definido no Capítulo 7.

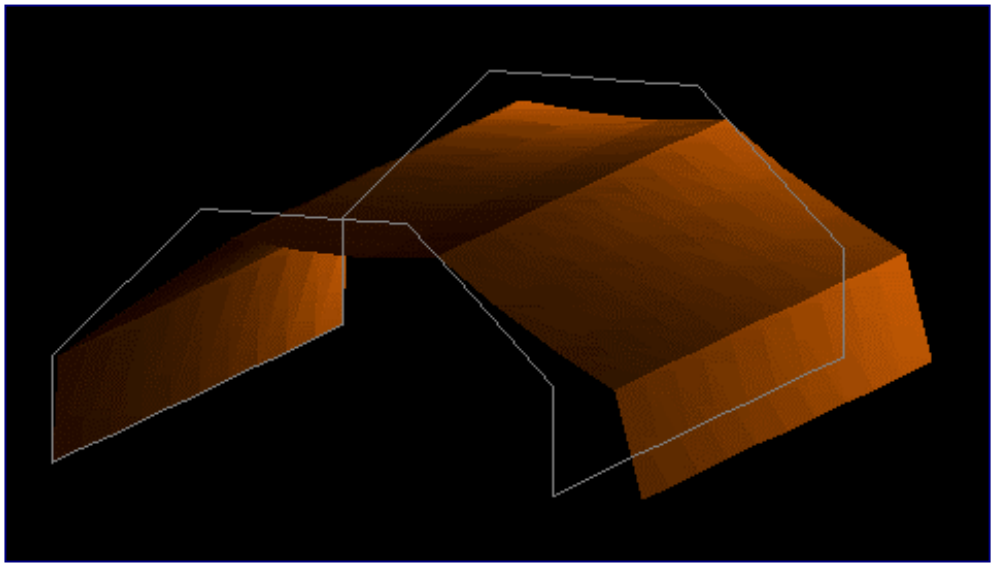
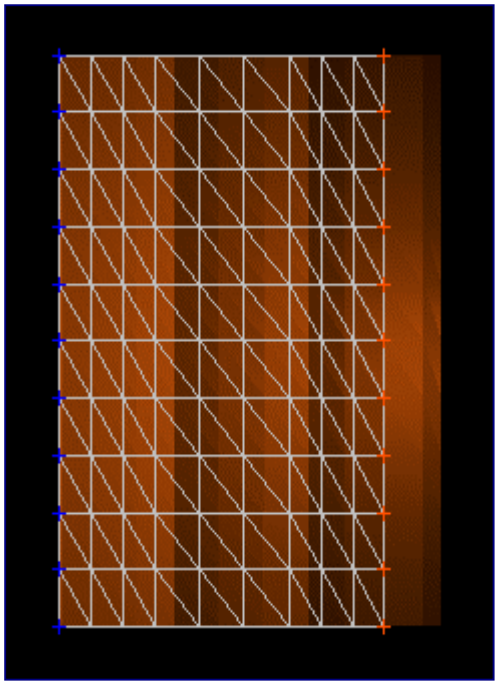


Figura Colorida 16. Outras condições de contorno. A figura mostra as arestas de contorno do modelo mecânico e do modelo gráfico da estrutura deformada. A análise foi feita com os mesmos carregamentos, mesmo material e mesma malha de elementos do exemplo anterior. As únicas alterações estão nos apoios. Somente os nós de cota mais inferior foram restringidos. Os deslocamentos da estrutura deformada, gerada por um objeto da classe `tWarpFilter`, estão em escala 0,15:1. A imagem do modelo gráfico foi produzida por um objeto da classe `tScanner`.



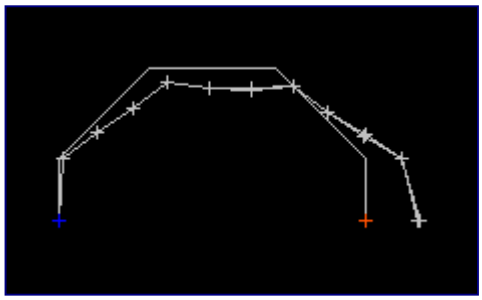


Figura Colorida 17. Outras vistas da estrutura deformada da cobertura. Ao lado, vista superior, com os *blips* azuis e vermelhos indicando os nós apoiados. Acima, vista frontal da cobertura.

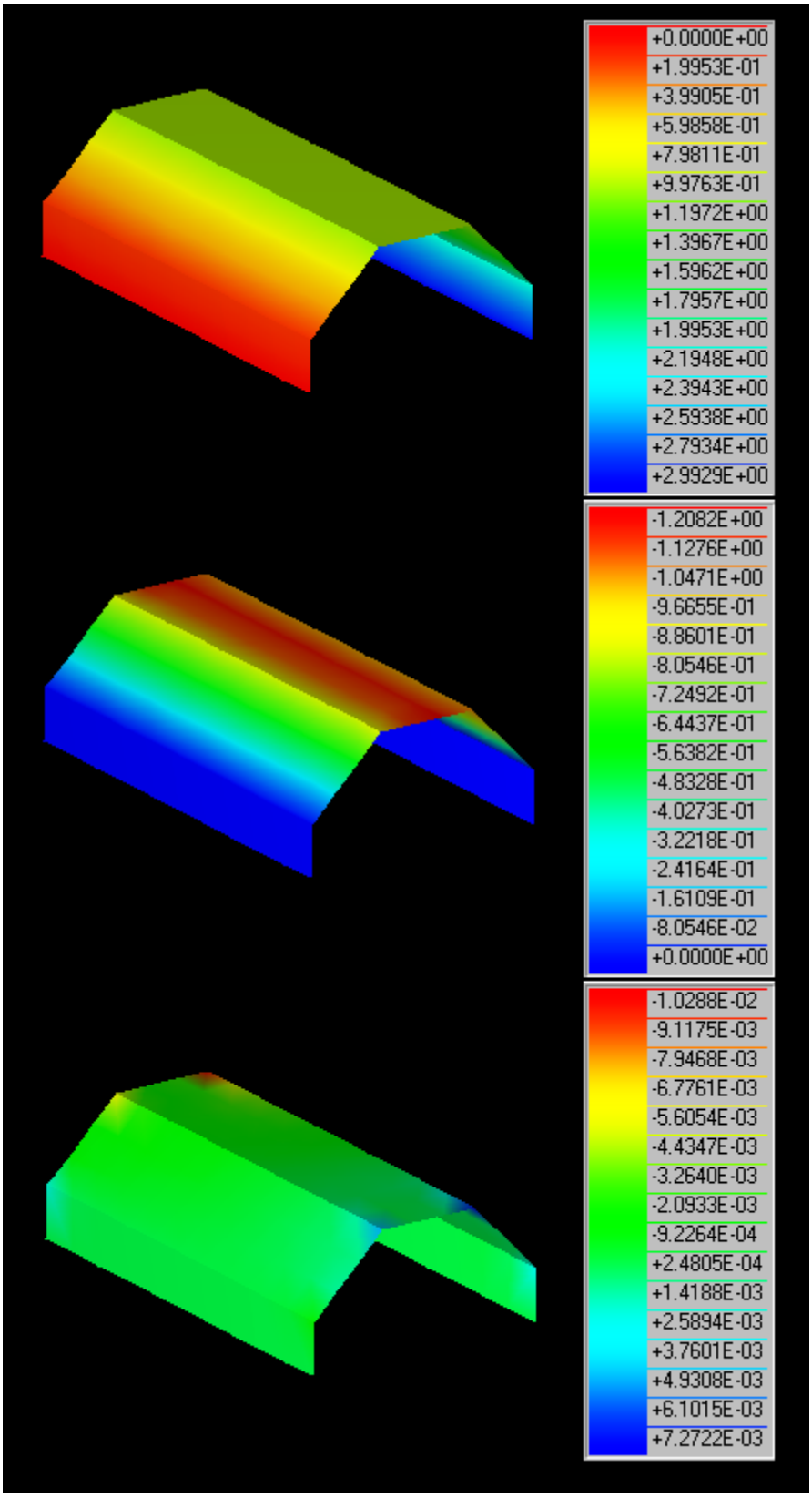


Figura Colorida 18. Mapas de cores e janelas de tabelas de cores. De cima para baixo, à esquerda, mapas de cores para os componentes x,y,z de deslocamento, respectivamente. À direita, as tabelas de cores correspondentes. Apenas 16 dos 256 valores associados às cores da tabela são indicados.

OSW-Shell. OSW-Shell é uma aplicação OSW de modelagem de cascas delgadas. Vimos, no Capítulo 1, que uma aplicação de modelagem em OSW é um programa de visualização e de análise numérica de modelos estruturais. A seguir, apresentaremos algumas Figuras Coloridas que ilustram alguns dos recursos da interface gráfica de OSW-Shell.

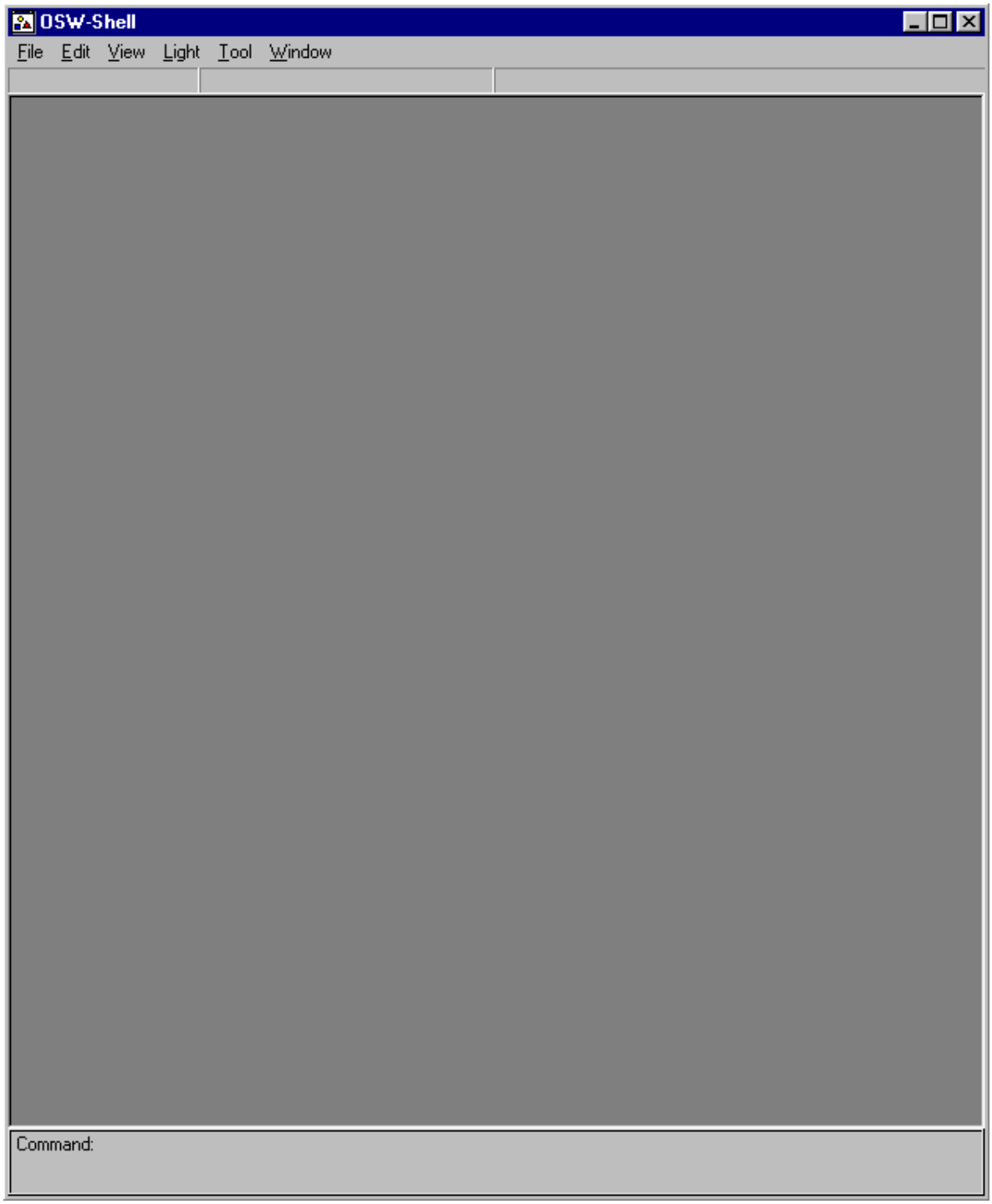


Figura Colorida 19. Janela principal de OSW-Shell. A janela principal contém, na parte superior, a barra de título da aplicação, a barra de menus e a barra de *status*, e na parte inferior, a janela de comandos. Na área cliente da janela principal são exibidas as janelas de vistas das cenas do documento da aplicação.

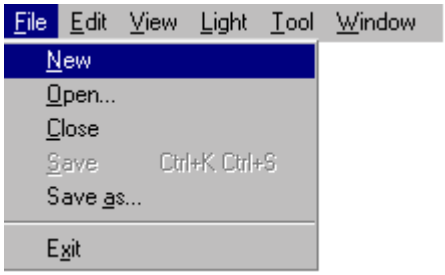


Figura Colorida 20. Criando um novo documento. O comando `new` cria um novo documento da classe `tShellDoc`, como descrito no Capítulo 9.

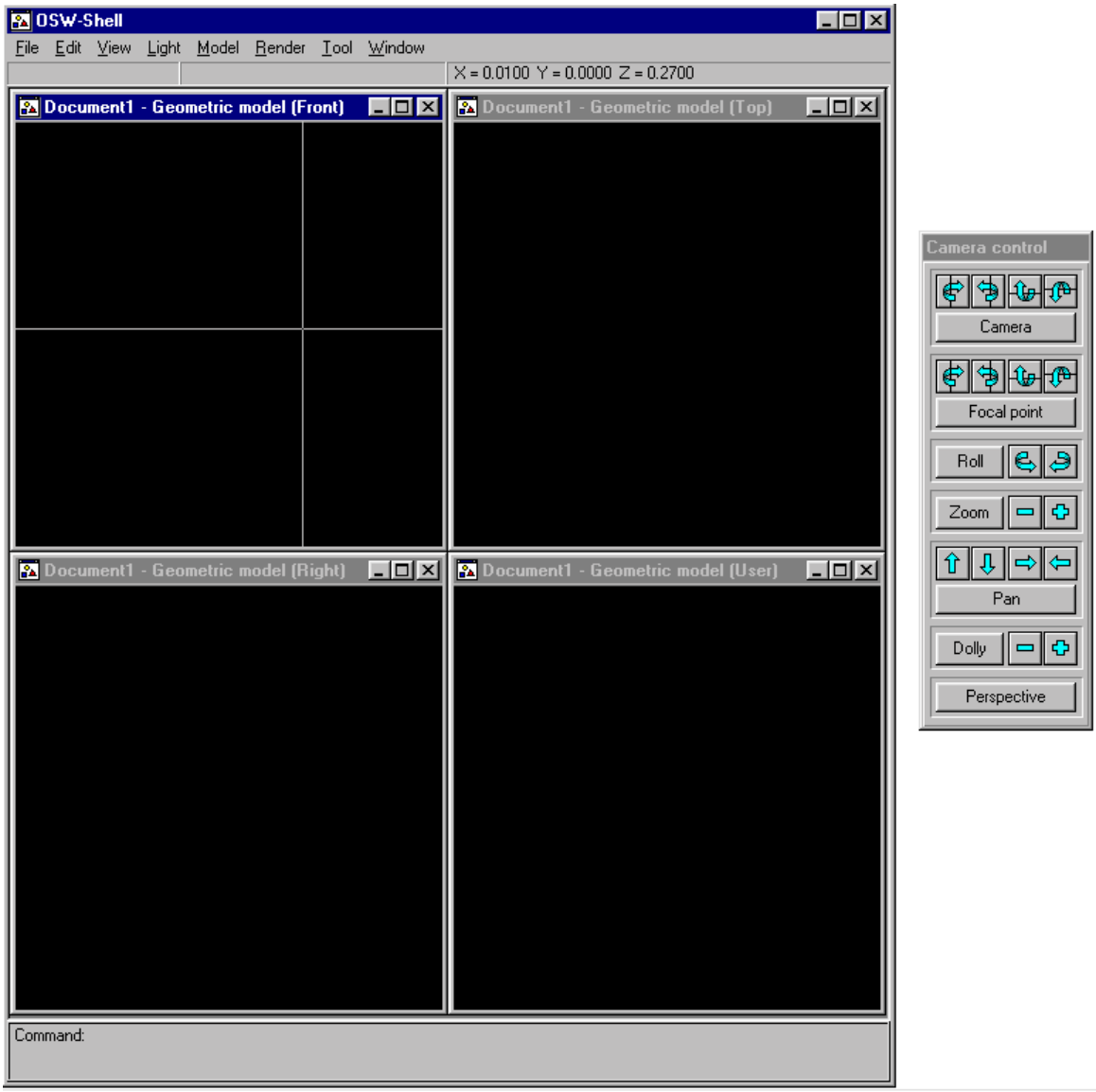


Figura Colorida 21. Janelas de vistas do modelo geométrico de uma casca. Um documento da classe `tShellDoc`, após criado, constrói uma cena da classe `tGeoScene`, a qual, por sua vez, constrói quatro janelas de vistas da classe `tGeoView`, descrita no Capítulo 11. As vistas apresentam projeções ortográficas de frente (*Front*), de cima (*Top*), de direita (*Right*) e isométrica (*User*) do modelo. Qualquer vista pode ser modificada pelos controles da caixa de diálogo *Camera Control*, os quais executam, entre outras, as operações de azimute, elevação, rolagem, arfagem, guinagem e *zoom* definidas no Capítulo 7. *Camera Control* (à direita) é um objeto da classe `tCameraDialog`.

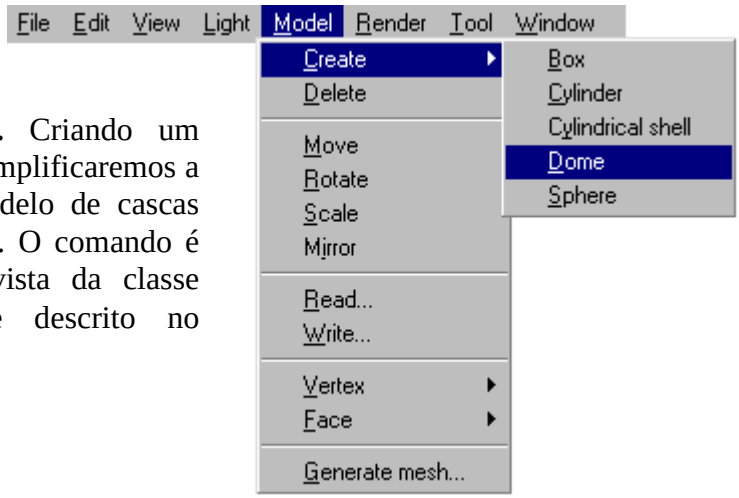


Figura Colorida 22. Criando um modelo de cascas. Exemplificaremos a construção de um modelo de cascas com o comando `dome`. O comando é executado por uma vista da classe `tGeoView`, conforme descrito no Capítulo 11.

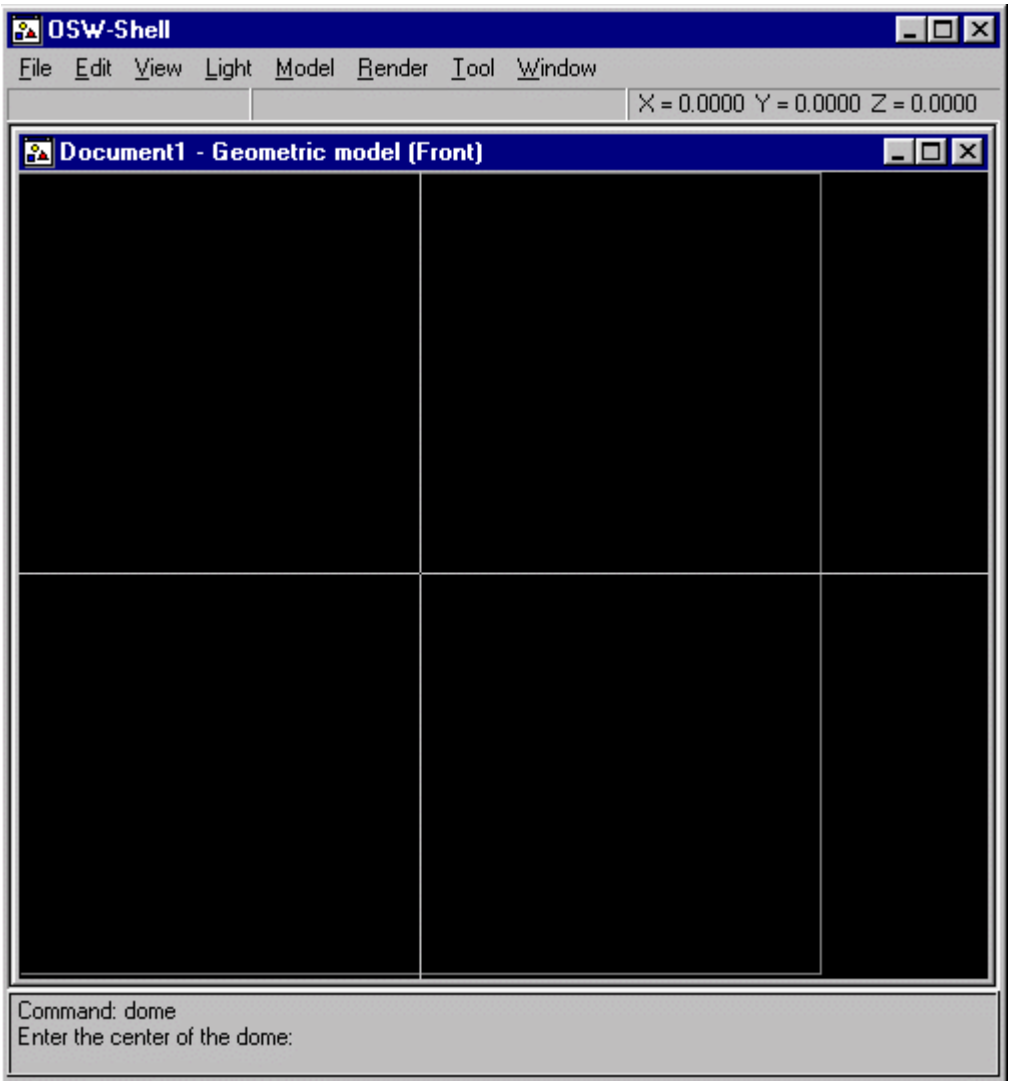


Figura Colorida 23. Trabalhando com vistas. No Capítulo 9, vimos que as vistas de uma cena, além de exibir imagens sintetizadas da cena, podem também ser utilizadas para entrada de parâmetros necessários à execução de um comando. O comando `dome` requer dois parâmetros: o centro do domo e seu raio. Os parâmetros podem ser apontados com o *cursor* em qualquer vista da classe `tGeoView`.

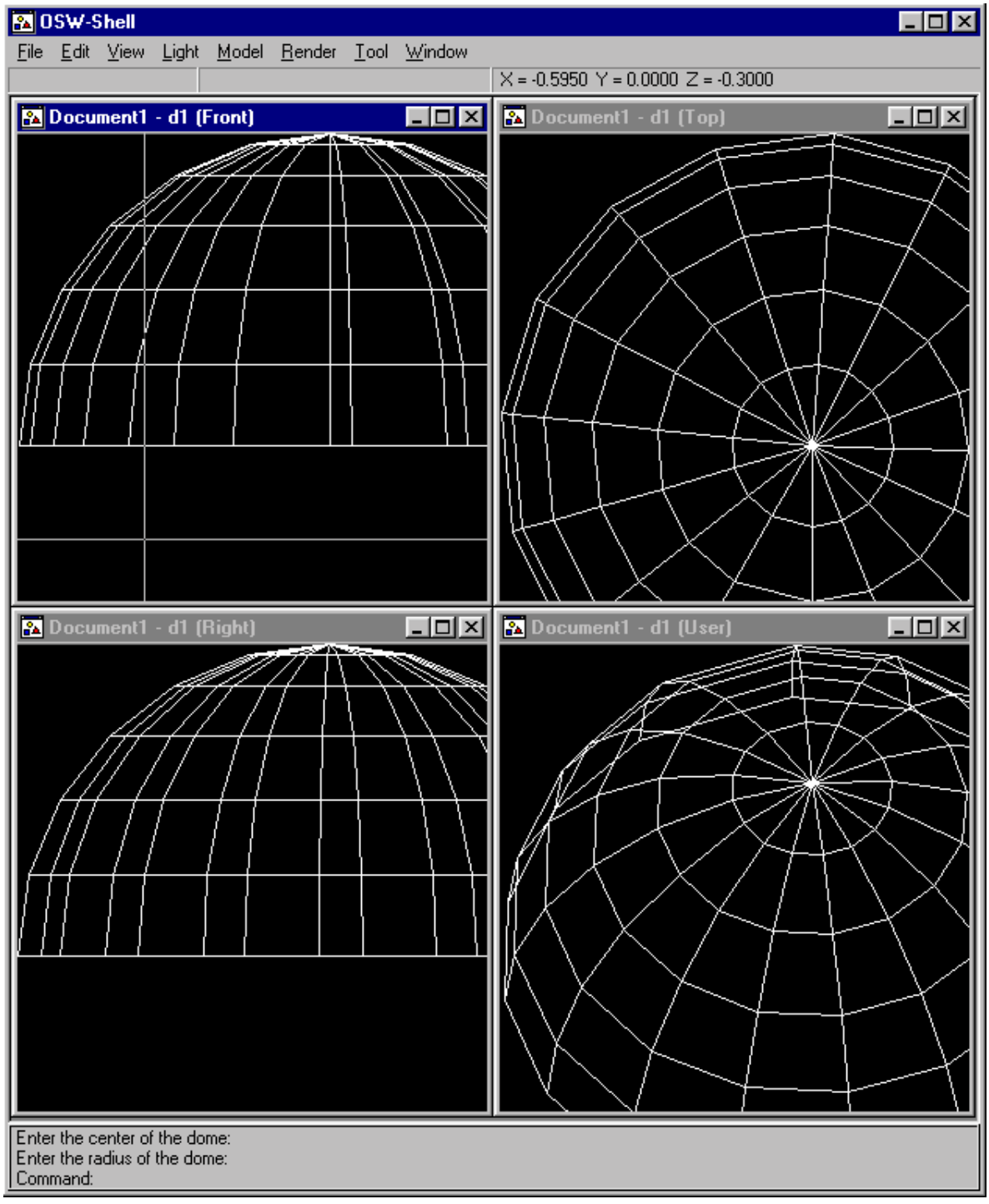


Figura Colorida 24. Imagens *wireframe* do modelo geométrico. O comando `dome` finaliza sua execução construindo um ator (um objeto da classe `tActor`) para o modelo de cascas do domo e adicionando o ator à cena de modelos geométricos do documento. As imagens do ator do modelo são visualizadas nas vistas da cena. Cada vista apresenta sua própria imagem do ator, sintetizada pelo *renderer* da vista (objeto de uma classe derivada de `tRenderer`) em função dos parâmetros da câmara virtual da vista (objeto da classe `tCamera`). Por enquanto, temos apenas imagens *wireframe* do modelo de cascas do domo, construído com centro na origem e raio unitário definido pela distância da origem até o ponto de coordenadas (0,1,0).

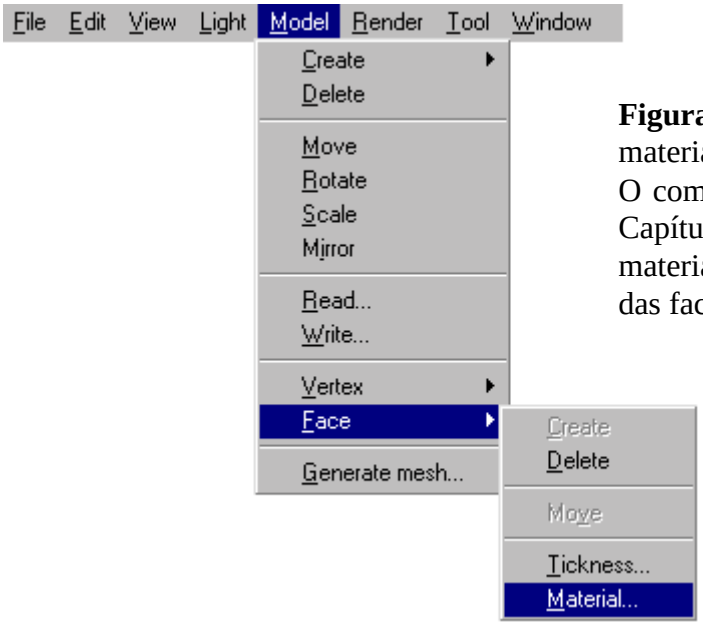


Figura Colorida 25. Definindo o tipo de material das faces do modelo geométrico. O comando `set_material`, descrito no Capítulo 11, permite a definição do material (objeto da classe `tMaterial`) das faces do modelo de cascas.

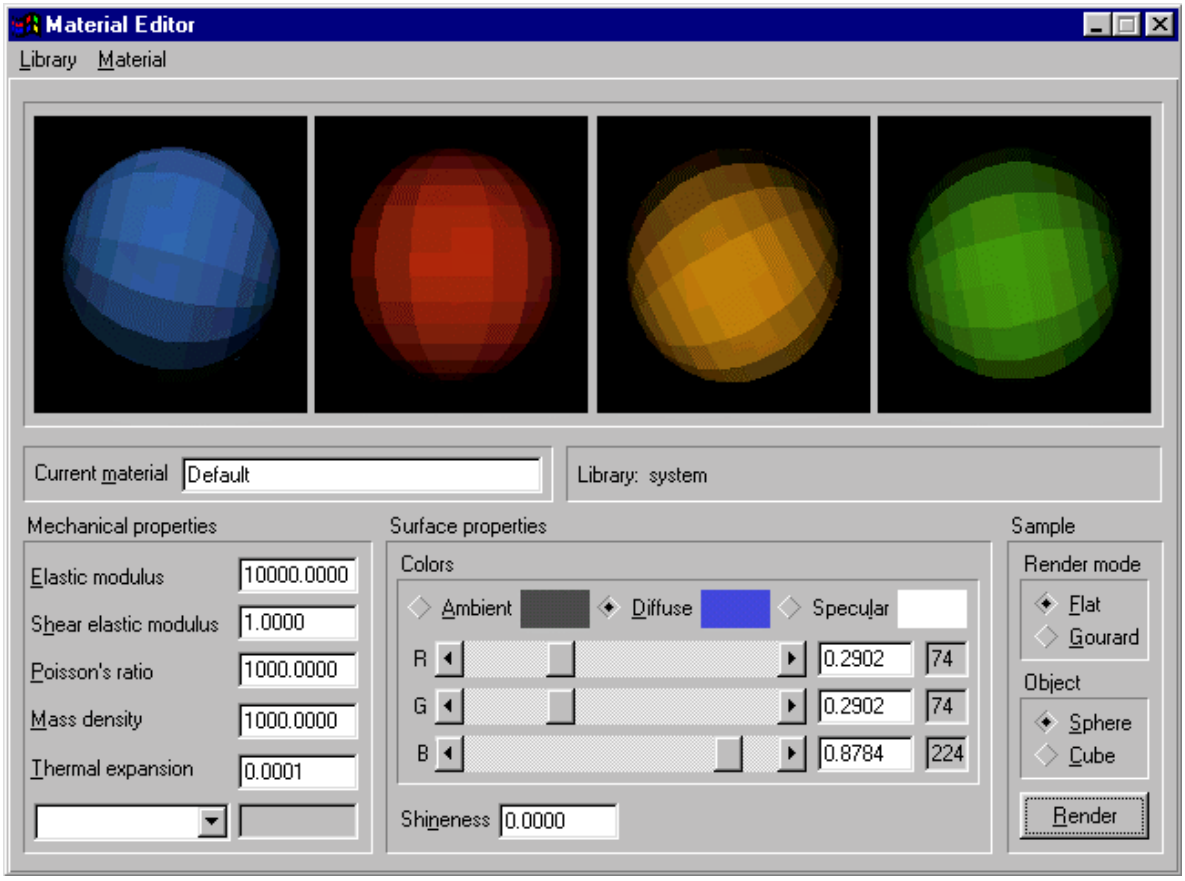


Figura Colorida 26. O editor de materiais. O editor de materiais é um objeto da classe `tMaterialEditor` que permite a especificação das propriedades mecânicas e de superfície de um material, definidas nos Capítulos 6 e 7, respectivamente. O material criado no editor de materiais pode ser armazenado em uma *biblioteca de materiais* e recuperado para uso em outras aplicações de modelagem.

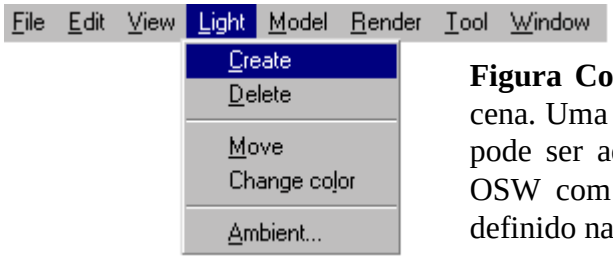


Figura Colorida 27. Acionando luzes à cena. Uma luz (objeto da classe `tLight`) pode ser adicionada a qualquer cena de OSW com o comando `create_light`, definido na classe base `tView`.

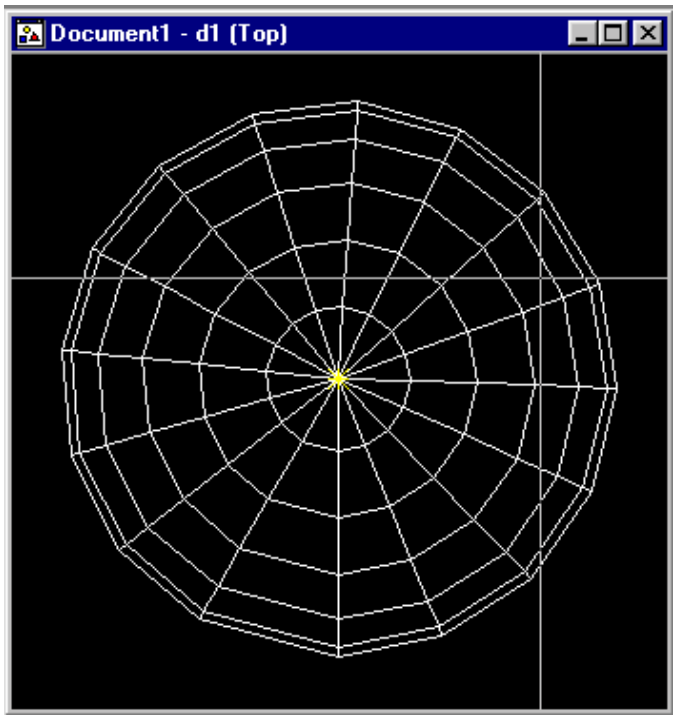
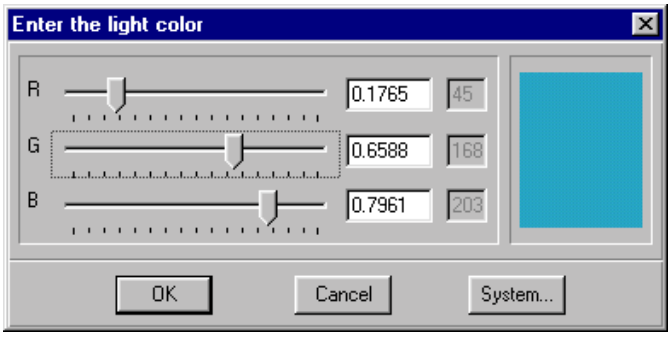


Figura Colorida 28. O comando `create_light` requer o ponto que define a posição da fonte de luz e sua cor. As coordenadas da posição da luz podem ser apontadas com o *cursor* em qualquer janela de vista da cena ou dadas em uma caixa de diálogo da classe `tInputDialog` (acima). Nas vistas da cena, a fonte de luz é representada por um ícone da classe `tDCObject` (à esquerda, no centro).



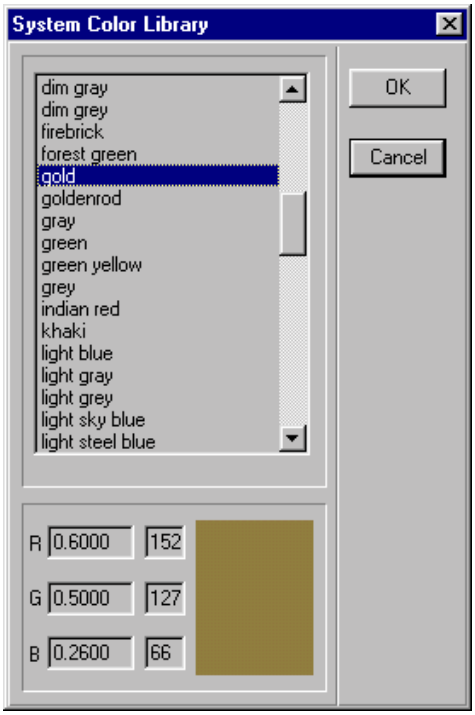


Figura Colorida 29. Definindo a cor da luz. A cor da luz é definida no modelo RGB visto no Capítulo 7. A cor pode ser tomada de uma *biblioteca de cores* de OSW (à direita) ou seus componentes RGB diretamente especificados em uma caixa de diálogo da classe `tColorDialog` (acima).

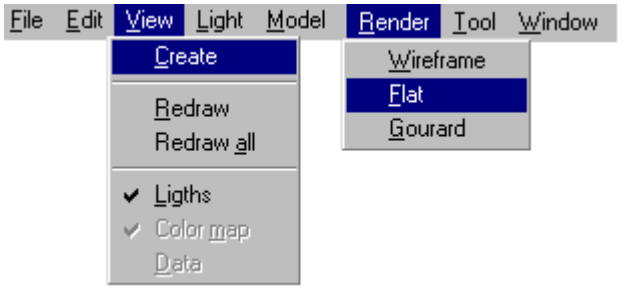


Figura Colorida 30. Criando uma nova vista e gerando uma imagem colorida do modelo geométrico. Uma nova vista de uma cena é criada com `create_view`. Uma imagem colorida do modelo pode ser obtida com o comando `flat`.

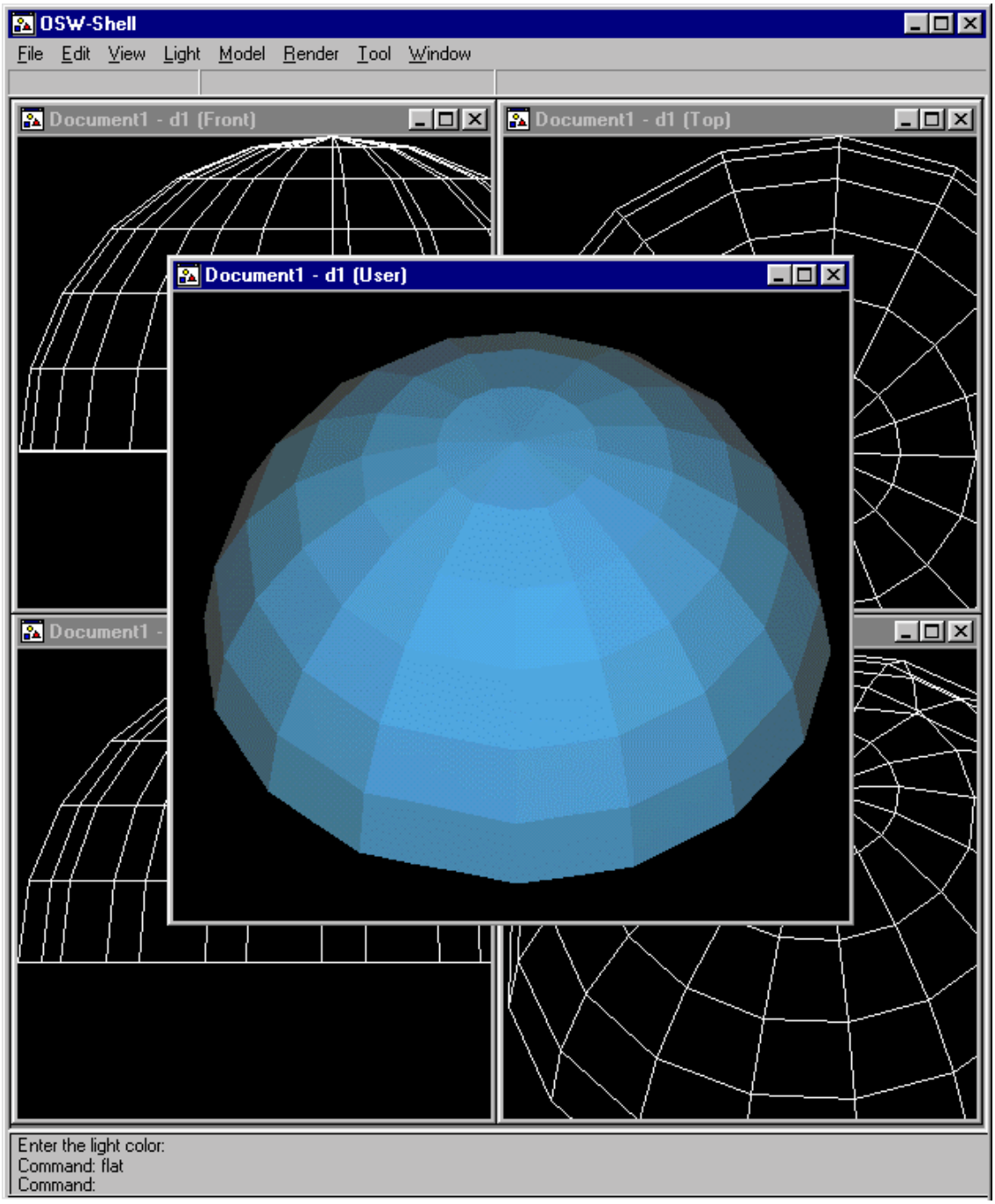
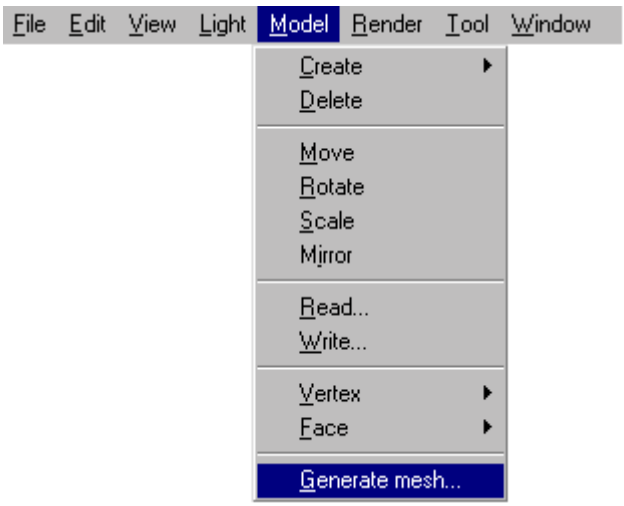


Figura Colorida 31. Imagem colorida com tonalização "*flat*" do modelo geométrico, resultante da execução do comando `flat`. A imagem foi gerada por um *renderer* da classe `tScanner`.



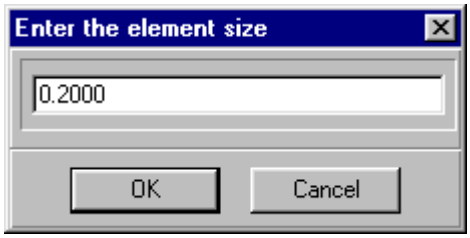


Figura Colorida 32. Gerando a malha de elementos. O comando `generate_mesh` é responsável pela geração da malha do modelo mecânico da estrutura. O comando requer como parâmetro o tamanho do elemento (acima).

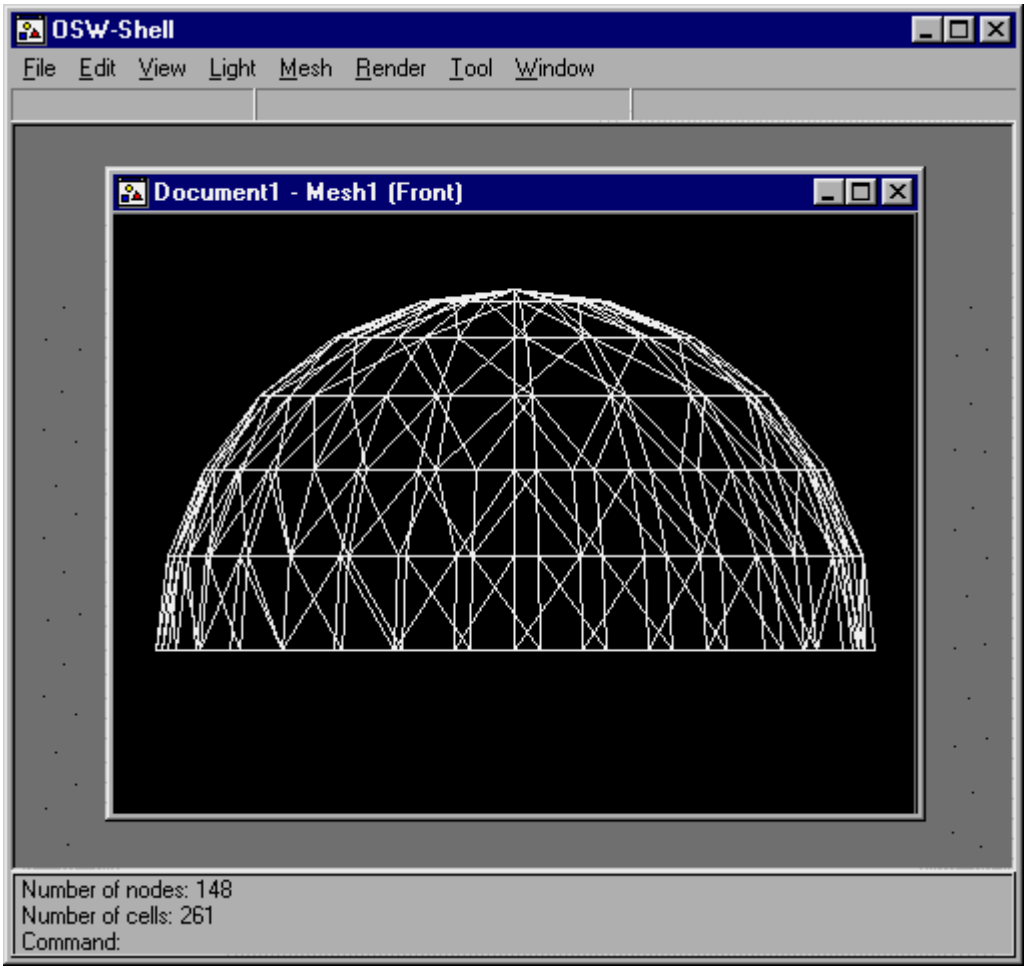


Figura Colorida 33. Malha de elementos finitos do modelo mecânico do domo. A malha é gerada por um objeto da classe `t2DMeshGenerator`, criado pelo comando `generate_mesh`. Após a geração da malha, uma cena da classe `tMecScene`, descrita no Capítulo 11, é construída e adicionada ao documento da aplicação. A malha é exibida em uma vista da classe `tMecView`, também descrita no Capítulo 11. Uma célula da malha "herda" o material da face do domo que a originou. Podemos gerar tantas malhas quanto quisermos, a partir do modelo geométrico do domo.

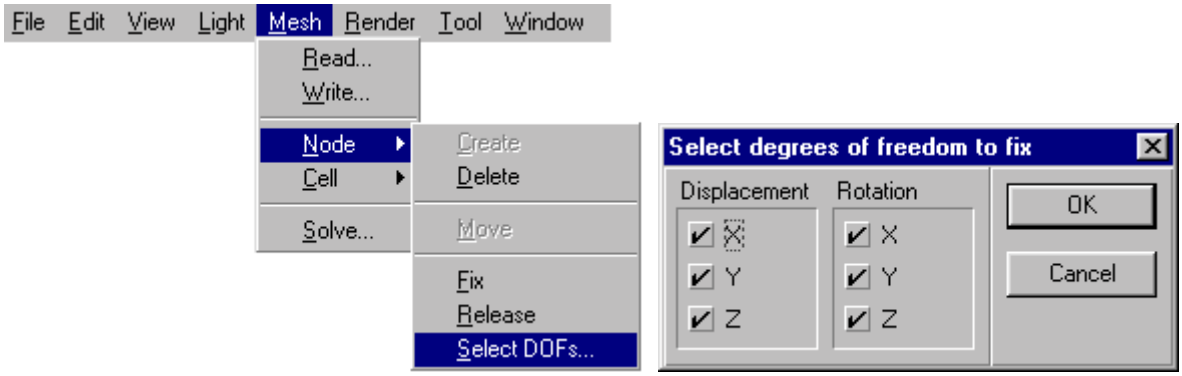


Figura Colorida 34. Definindo a vinculação do modelo mecânico. Os comandos `select_dofs` e `fix`, descritos no Capítulo 11, são utilizados para definição da vinculação da estrutura. O comando `select_dofs` permite a seleção de quais graus de liberdade nodais serão restringidos (deslocamentos e/ou rotações em x,y,z). O comando `fix` permite a seleção dos nós do modelo mecânico aos quais serão aplicadas as restrições escolhidas com o comando `select_dofs`.

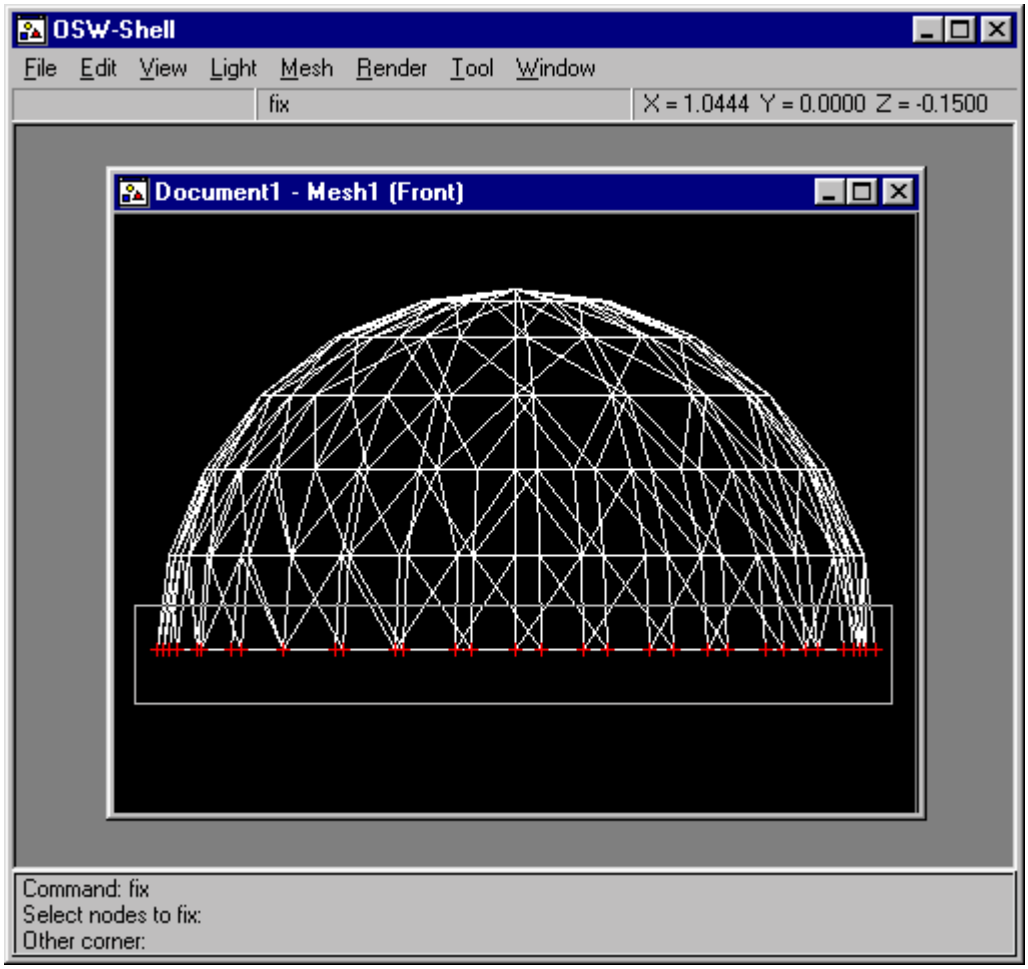


Figura Colorida 35. Fixando os nós do modelo mecânico. A seleção dos nós a serem fixados pelo comando `fix` é efetuada nas janelas de vista do modelo mecânico (objetos da classe `tMecView`). Os métodos de seleção de nós e células de um modelo mecânico são implementados na classe `tMeshView`, da qual `tMecView` deriva. No exemplo, estamos engastando todos os nós da base do domo.

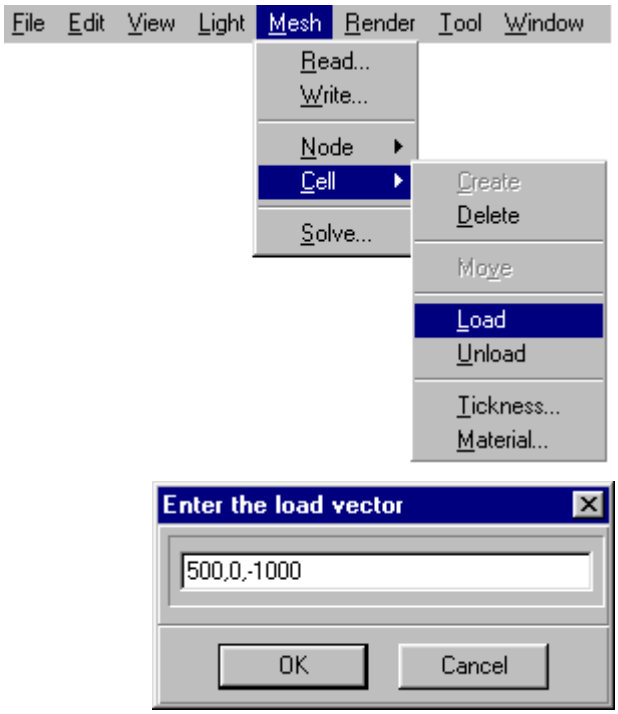


Figura Colorida 36. Aplicando os carregamentos às células do modelo mecânico. O comando `load_cell`, descrito no Capítulo 11, é utilizado para definição do carregamento aplicado às células do modelo mecânico do domo. O comando requer dois parâmetros: o vetor de carga e a lista de elementos aos quais será aplicado o carregamento. O vetor de cargas, igual a (500,0,-1000) u.f. (em relação ao sistema global da estrutura), pode ser diretamente especificado em uma caixa de diálogo da classe `tInputDialog` (à esquerda).

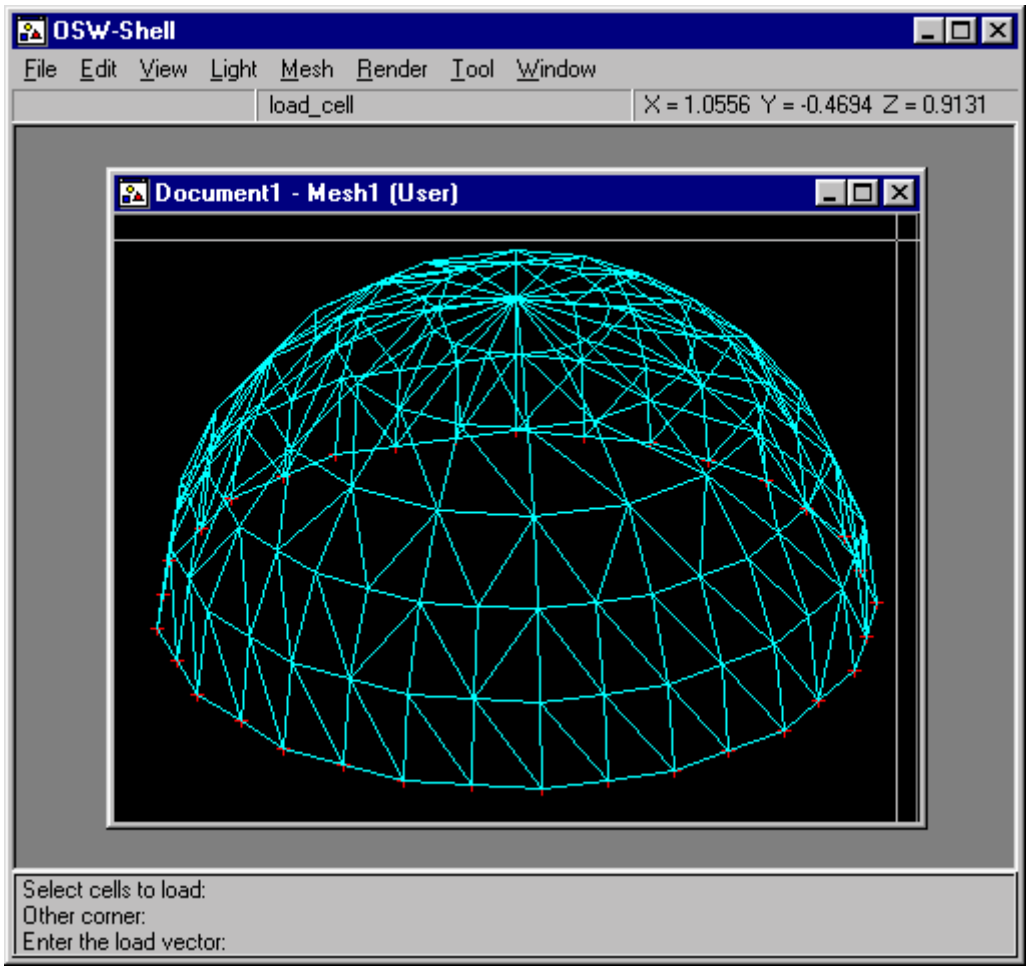


Figura Colorida 37. Carregando as células do modelo mecânico. A seleção dos elementos finitos aos quais serão aplicados o carregamento é efetuada nas janelas de vista do modelo mecânico, tal como a seleção de nós do comando `fix`. No exemplo, estamos aplicando o carregamento a todos os elementos finitos do modelo mecânico do domo.

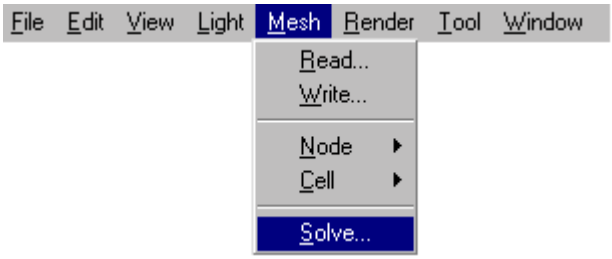
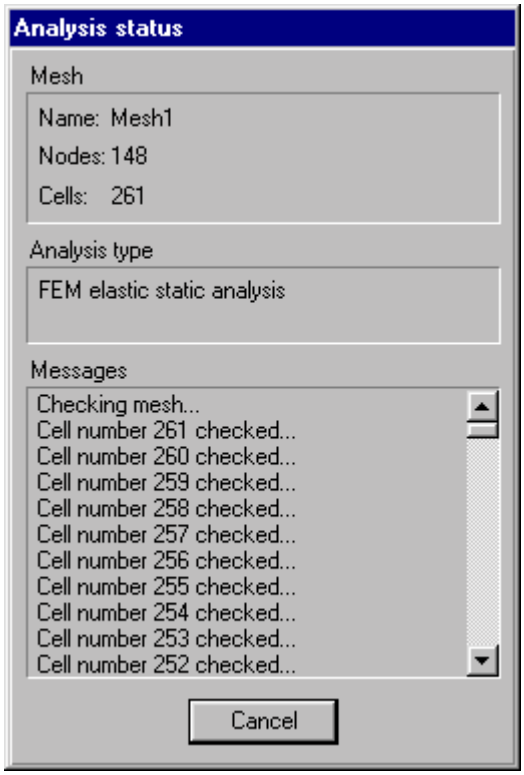
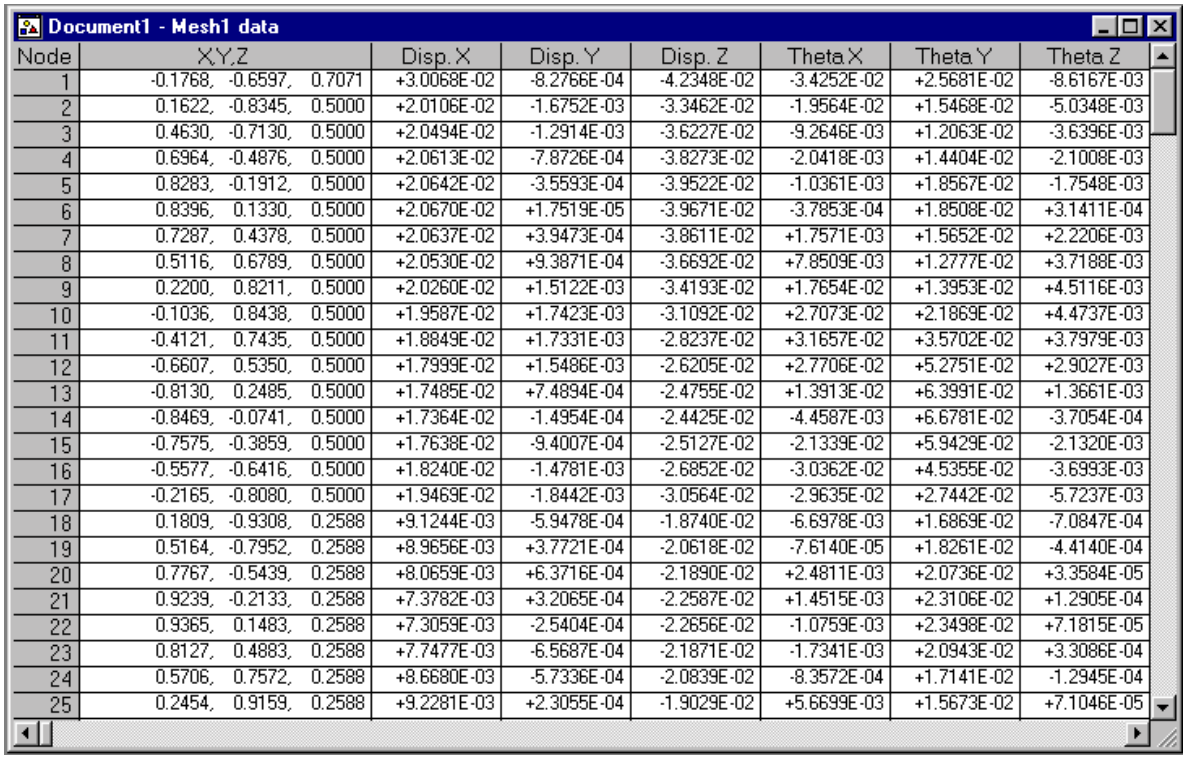


Figura Colorida 38. Analisando o modelo mecânico do domo. A execução da análise numérica do modelo mecânico da estrutura é iniciada com o comando `solve`. O comando cria uma nova linha de execução (*thread*) e um *solver* da classe `tFESolver` (descrita no Capítulo 10). O método `Run()` do *solver* é executado na *thread* criada pelo comando. Dessa forma, podemos realizar outras tarefas em OSW-Shell enquanto a análise numérica estiver sendo efetuada. O *status* da análise é exibida na caixa de diálogo *Analysis Status* (à direita).





Node	X,Y,Z	Disp. X	Disp. Y	Disp. Z	Theta X	Theta Y	Theta Z
1	-0.1768, -0.6597, 0.7071	+3.0068E-02	-8.2766E-04	-4.2348E-02	-3.4252E-02	+2.5681E-02	-8.6167E-03
2	0.1622, -0.8345, 0.5000	+2.0106E-02	-1.6752E-03	-3.3462E-02	-1.9564E-02	+1.5468E-02	-5.0348E-03
3	0.4630, -0.7130, 0.5000	+2.0494E-02	-1.2914E-03	-3.6227E-02	-9.2646E-03	+1.2063E-02	-3.6396E-03
4	0.6964, -0.4876, 0.5000	+2.0613E-02	-7.8726E-04	-3.8273E-02	-2.0418E-03	+1.4404E-02	-2.1008E-03
5	0.8283, -0.1912, 0.5000	+2.0642E-02	-3.5593E-04	-3.9522E-02	-1.0361E-03	+1.8567E-02	-1.7548E-03
6	0.8396, 0.1330, 0.5000	+2.0670E-02	+1.7519E-05	-3.9671E-02	-3.7853E-04	+1.8508E-02	+3.1411E-04
7	0.7287, 0.4378, 0.5000	+2.0637E-02	+3.9473E-04	-3.8611E-02	+1.7571E-03	+1.5652E-02	+2.2206E-03
8	0.5116, 0.6789, 0.5000	+2.0530E-02	+9.3871E-04	-3.6692E-02	+7.8509E-03	+1.2777E-02	+3.7188E-03
9	0.2200, 0.8211, 0.5000	+2.0260E-02	+1.5122E-03	-3.4193E-02	+1.7654E-02	+1.3953E-02	+4.5116E-03
10	-0.1036, 0.8438, 0.5000	+1.9587E-02	+1.7423E-03	-3.1092E-02	+2.7073E-02	+2.1869E-02	+4.4737E-03
11	-0.4121, 0.7435, 0.5000	+1.8849E-02	+1.7331E-03	-2.8237E-02	+3.1657E-02	+3.5702E-02	+3.7979E-03
12	-0.6607, 0.5350, 0.5000	+1.7999E-02	+1.5486E-03	-2.6205E-02	+2.7706E-02	+5.2751E-02	+2.9027E-03
13	-0.8130, 0.2485, 0.5000	+1.7485E-02	+7.4894E-04	-2.4755E-02	+1.3913E-02	+6.3991E-02	+1.3661E-03
14	-0.8469, -0.0741, 0.5000	+1.7364E-02	-1.4954E-04	-2.4425E-02	-4.4587E-03	+6.6781E-02	-3.7054E-04
15	-0.7575, -0.3859, 0.5000	+1.7638E-02	-9.4007E-04	-2.5127E-02	-2.1339E-02	+5.9429E-02	-2.1320E-03
16	-0.5577, -0.6416, 0.5000	+1.8240E-02	-1.4781E-03	-2.6852E-02	-3.0362E-02	+4.5355E-02	-3.6993E-03
17	-0.2165, -0.8080, 0.5000	+1.9469E-02	-1.8442E-03	-3.0564E-02	-2.9635E-02	+2.7442E-02	-5.7237E-03
18	0.1809, -0.9308, 0.2588	+9.1244E-03	-5.9478E-04	-1.8740E-02	-6.6978E-03	+1.6869E-02	-7.0847E-04
19	0.5164, -0.7952, 0.2588	+8.9656E-03	+3.7721E-04	-2.0618E-02	-7.6140E-05	+1.8261E-02	-4.4140E-04
20	0.7767, -0.5439, 0.2588	+8.0659E-03	+6.3716E-04	-2.1890E-02	+2.4811E-03	+2.0736E-02	+3.3584E-05
21	0.9239, -0.2133, 0.2588	+7.3782E-03	+3.2065E-04	-2.2587E-02	+1.4515E-03	+2.3106E-02	+1.2905E-04
22	0.9365, 0.1483, 0.2588	+7.3059E-03	-2.5404E-04	-2.2656E-02	-1.0759E-03	+2.3498E-02	+7.1815E-05
23	0.8127, 0.4883, 0.2588	+7.7477E-03	-6.5687E-04	-2.1871E-02	-1.7341E-03	+2.0943E-02	+3.3086E-04
24	0.5706, 0.7572, 0.2588	+8.6680E-03	-5.7336E-04	-2.0839E-02	-8.3572E-04	+1.7141E-02	-1.2945E-04
25	0.2454, 0.9159, 0.2588	+9.2281E-03	+2.3055E-04	-1.9029E-02	+5.6699E-03	+1.5673E-02	+7.1046E-05

Figura Colorida 39. Janela de resultados de análise. Os resultados numéricos da análise são exibidos em uma janela da classe `tDataWindow`, descrita no Capítulo 10. A janela possui uma coleção de objetos da classe `tColumn`, também descrita no Capítulo 10. Cada objeto coluna da janela é responsável pela exibição de um atributo de um nó do modelo mecânico da estrutura.

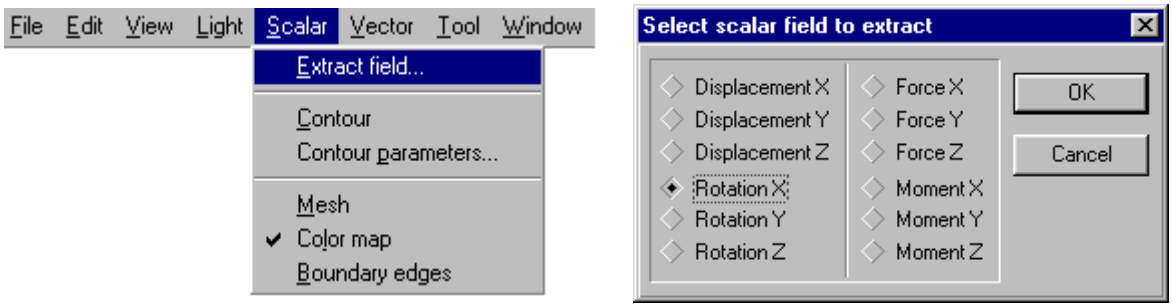


Figura Colorida 40. Visualizando escalares. Concluída a análise numérica, o comando `solve` cria uma cena da classe `tResScene` e adiciona a cena criada ao documento da aplicação. As vistas da cena, objetos da classe `tResView`, executam comandos responsáveis pela visualização dos mapas de cores, das isolinhas e da estrutura deformada do modelo mecânico, a partir do pós-processamento dos resultados da análise numérica. O comando `extract_scalar_field`, descrito no Capítulo 11, cria um objeto da classe `tScalarExtractor` que extrai do modelo mecânico o campo escalar selecionado na caixa de diálogo *Select scalar field to extract*, conforme visto no Capítulo 7.

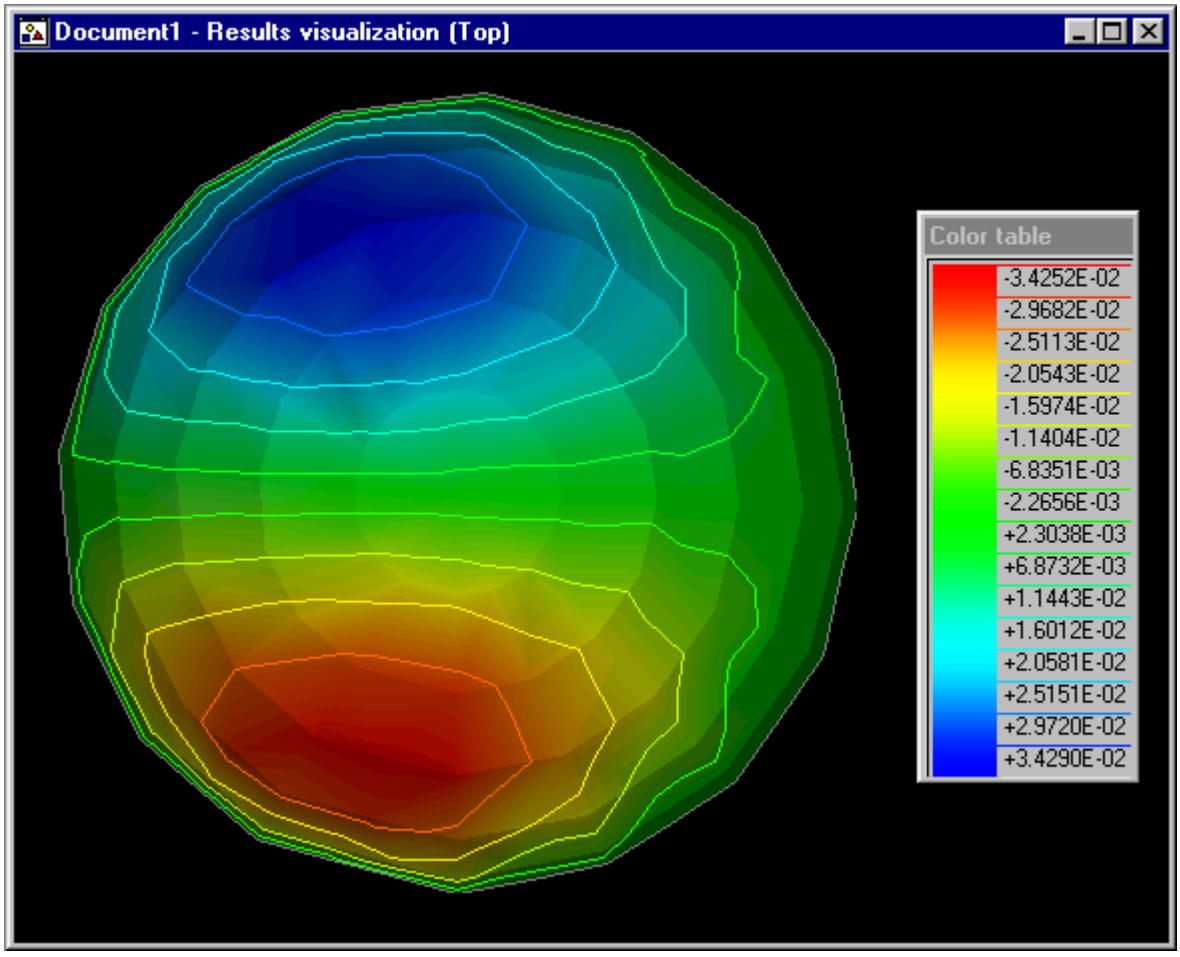


Figura Colorida 41. Mapa de cores, isolinhas e arestas de contorno. Ao contrário das outras cenas de OSW-Shell, as cenas da classe `tResScene` podem conter vários atores. No exemplo, arestas de contorno do modelo e mapa de cores e isolinhas correspondentes à rotação x.

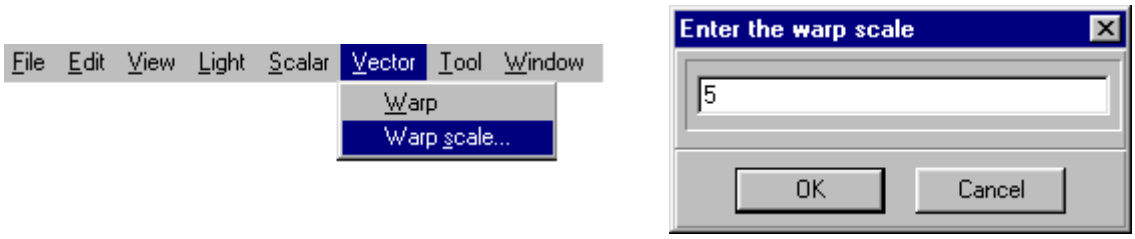


Figura Colorida 42. Visualizando vetores. O comando `show-warp`, descrito no Capítulo 11, exibe a estrutura deformada do modelo mecânico. O comando cria um filtro da classe `tWarpFilter`, responsável pela construção do modelo gráfico da estrutura deformada. A deformação da estrutura é controlada pelo valor de escala do filtro, especificado em um objeto da classe `tInputDialog`.

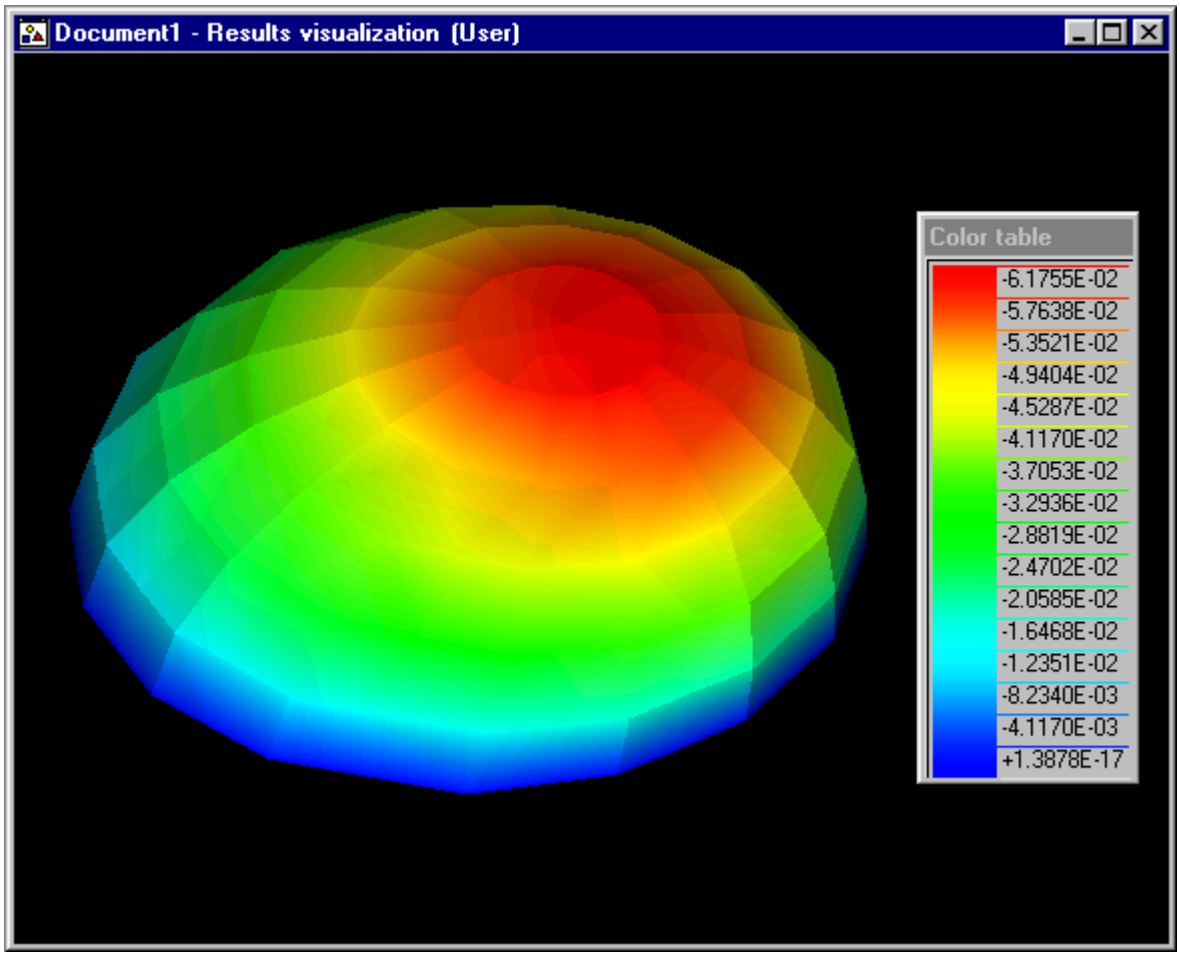


Figura Colorida 43. Estrutura deformada e mapa de cores. Podemos visualizar escalares e vetores conjuntamente. No exemplo, a estrutura deformada do domo e o mapa de cores dos deslocamentos em z. Ao lado, a tabela de cores do mapa de cores.

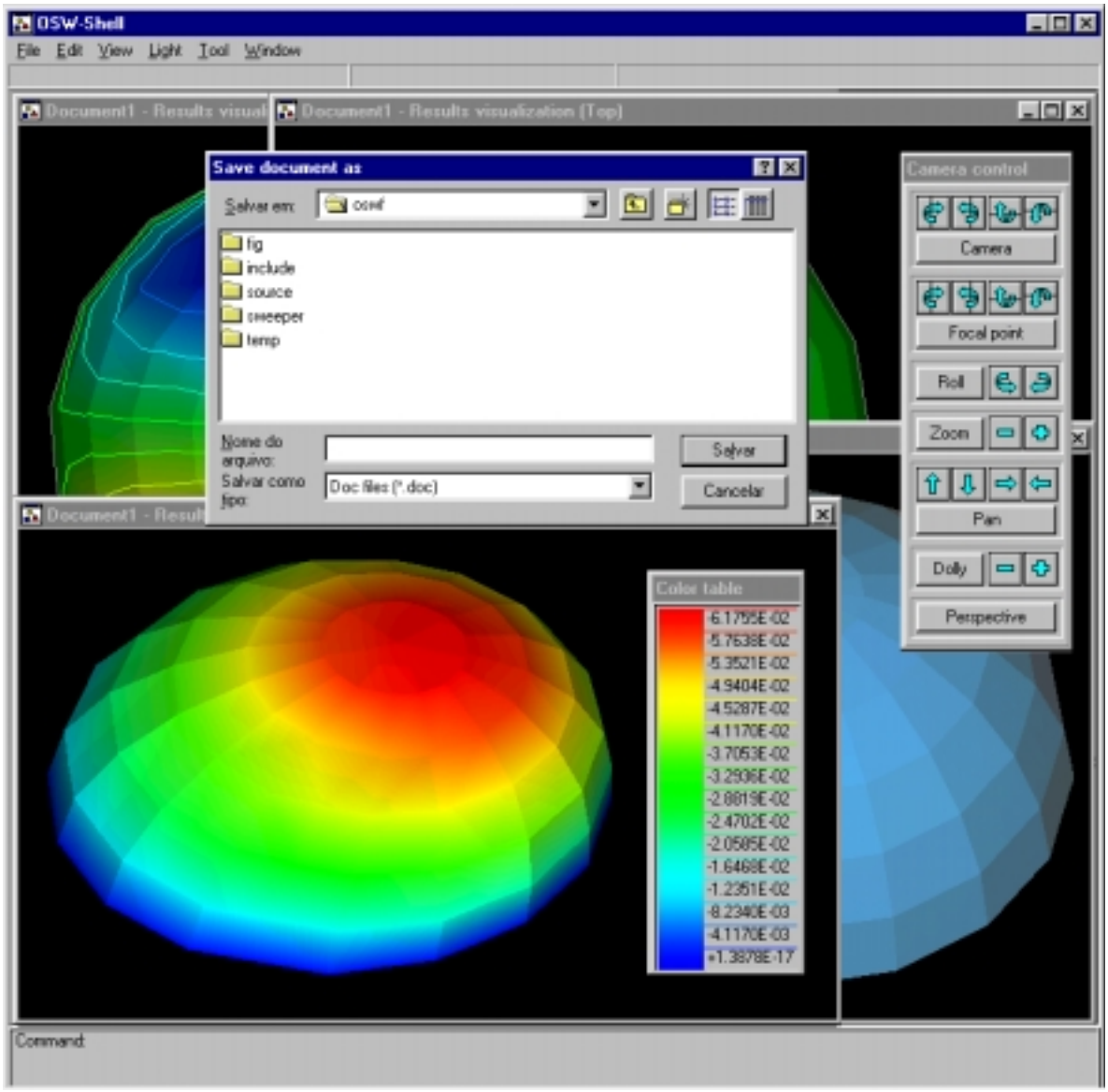


Figura Colorida 44. Salvando o documento. Os objetos pertencentes a um documento OSW, tais como modelos e cenas (juntamente com suas vistas, atores, luzes e câmaras), além do próprio documento, são objetos *persistentes*, isto é, são objetos que sobrevivem à execução da aplicação. O comando `save` permite que o documento seja armazenado em arquivos permanentes e recuperado para posterior utilização.

CAPÍTULO 12

Conclusão

12.1 OSW e o Problema Fundamental

Qual é o comportamento de uma estrutura? Como podemos prever esse comportamento, e com que precisão e rapidez? As perguntas são instigantes. Não somente porque as respostas são, do ponto de vista prático, fundamentais para o engenheiro de estruturas, mas também porque encerram um dos motivos mais fascinantes do espírito humano: a curiosidade intrínseca de tentarmos compreender a natureza. As respostas, contudo, não sabemos, pelo menos exatamente. Por isso, construímos *modelos*.

Para nossos propósitos, modelos podem ser entendidos como representações das características principais de um objeto, criadas com o objetivo de permitir a visualização e a compreensão da *estrutura* e do *comportamento* do objeto antes de sua construção. Em engenharia, os objetos possuem a estrutura definida por superfícies de sólidos de um *modelo geométrico* que descreve, exata ou aproximadamente, suas formas e dimensões. O comportamento do objeto é regido por um conjunto de equações diferenciais de um *modelo matemático* que nos permite prever, sob certas condições, os efeitos de ações externas sobre o objeto. A solução do modelo matemático, baseada em formulações derivadas de um *modelo mecânico* do objeto, pode ser obtida computacionalmente pelo método dos elementos finitos ou método dos elementos de contorno.

Object Structural Workbench, ou OSW, é um sistema de *programação orientada a objetos* destinado à construção de *aplicações de modelagem* em engenharia de estruturas. Uma aplicação de modelagem OSW é um programa C++ de análise numérica e de visualização de modelos estruturais que executa no sistema operacional Windows NT. 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.

Defendemos a tese de que o emprego da tecnologia de objetos em OSW é natural porque observamos, computacionalmente, uma identidade dos conceitos de modelo e objeto, como dados no Capítulo 2 e no Capítulo 8, respectivamente. Dessa forma, pensamos em aplicar as propriedades da programação orientada a objetos — encapsulamento, herança, polimorfismo — na modelagem de estruturas. Além disso, a programação orientada a objetos facilita o desenvolvimento de programas complexos porque

nos permite modelar problemas do mundo real tão próximo quanto possível da visão que temos desse mundo.

Tínhamos três objetivos nesse trabalho:

- *Apresentar os fundamentos matemáticos e computacionais utilizados no desenvolvimento das classes de objetos de OSW.* Os conceitos empregados na construção de OSW são multidisciplinares. Estudamos modelagem geométrica, mecânica do contínuo, métodos numéricos, computação gráfica, visualização científica e programação orientada a objetos. Na primeira parte do texto, procuramos apresentar esses conceitos. Talvez em alguns capítulos tenhamos sido mais eficientes que em outros. Infelizmente, não houve espaço-tempo suficiente para discutirmos os temas com a profundidade que gostaríamos (principalmente tempo). Esperamos, no entanto, que os resumos forneçam idéia das disciplinas tratadas e que as referências bibliográficas apresentadas possam servir de ponto de partida para outras pesquisas. Na próxima seção, apresentaremos algumas sugestões nesse sentido.

- *Descrever as classes de objetos das bibliotecas de classes de OSW.* Descrevemos as bibliotecas de classes no Capítulo 10, na forma de guia de referência. Juntamente com o *software*, disponível no Departamento de Engenharia de Estruturas da Escola de Engenharia de São Carlos, torcemos que esse guia possa ser útil, se não no desenvolvimento de programas baseados nas classes de OSW, pelo menos no desenvolvimento ou como inspiração para o desenvolvimento de outras aplicações orientadas a objetos em engenharia de estruturas.

- *Exemplificar a utilização das classes de OSW na construção de aplicações de modelagem.* No Capítulo 9 apresentamos um roteiro de como utilizar as classes de OSW na construção de um programa de modelagem orientado a objetos. Reconhecemos, contudo, a necessidade de elaboração de um documento em separado que descreva muito mais detalhadamente todos esses passos. No Capítulo 11 mostramos algumas imagens dos resultados gerados por uma aplicação OSW.

Não tivemos como objetivo inventar elementos finitos ou de contorno mais eficientes ou precisos. Embora nossas primeiras indagações tenham nos conduzido à formulação do problema fundamental, nosso principal propósito não foi resolvê-lo para todos os casos possíveis em engenharia de estruturas. Quizemos, sim, inventar um programa de computador que pudesse servir como ferramenta para que outros pesquisadores desenvolvessem seus próprios programas de análise sem ter que, toda vez, recomeçar do zero.

Acreditamos que, juntamente com a análise, a modelagem e a visualização são importantes; a primeira necessária para representar os próprios dados de entrada da análise, a segunda necessária para interpretar mais facilmente os resultados da análise. Mas modelagem e visualização, integradas com análise, resultam em um programa de computador muito complexo. O estudioso em métodos numéricos, sequioso do desenvolvimento de um novo elemento estrutural ou de uma formulação mais eficiente, pode não achar nem um pouco interessante ter de se preocupar com detalhes pertinentes à modelagem ou à visualização. Em caso de necessidade, poderia então utilizar alguns dos vários sistemas comerciais disponíveis, escrevendo processadores responsáveis pela adequação do formato de seus dados com o formato dos dados de entrada e saída desses programas. Por isso escrevemos OSW.

Mas OSW não é uma "caixa preta". Todo o código de OSW é disponível e pode ser modificado de acordo com as necessidades de uma aplicação em particular. Mais que isso, OSW é orientado a objetos. Embora o código possa ser modificado e melhorado, projetamos OSW pensando na reutilização de código, de forma acentuada. Ao invés de reinventar, vamos especializar. Nesse aspecto, a programação orientada a objetos se presta maravilhosamente bem. Utilizando os mecanismos de encapsulamento, herança e polimorfismo, podemos derivar novas classes de objetos das classes já existentes da biblioteca de classes de OSW e implementar estrutura e comportamento especializados às novas classes derivadas.

O estudioso de métodos numéricos pode muito bem desenvolver seu novo elemento estrutural e, então, construir uma classe derivada de `tFiniteElement` chamada, por exemplo, `tMyElement`. Muitos métodos declarados em `tFiniteElement` são virtuais; sobrecarregando apropriadamente esses métodos em `tMyElement`, nosso estudioso poderá facilmente implementar seu novo elemento finito. Para testar o novo elemento, não há necessidade de inventar uma malha de elementos ou funções de montagem e resolução do sistema de equações lineares. Além disso, há alguns recursos de pré-processamento e pós-processamento imediatamente disponíveis.

Esse crescimento pode acontecer, e esperamos que venha a acontecer, em todas as disciplinas abordadas em OSW. Novos métodos de representação geométrica de objetos, novos métodos de geração de malhas, novos algoritmos de visualização, novos modelos matemáticos dinâmicos e não-lineares e, principalmente, novos elementos estruturais.

É claro que a orientação a objetos não resolve todos os problemas de desenvolvimento de programas de computador. Precisamos utilizar muito criteriosamente algumas construções de uma linguagem orientada a objetos porque nossos programas de modelagem são programas de processamento numérico e gráfico. Todo *hardware* é pouco para computação numérica e gráfica. Em um programa orientado a objetos as possibilidades de abstração são tão grandes que, por muitas vezes, podemos esquecer que as implementações das abstrações podem ser ineficientes. Além disso, o projeto de uma hierarquia de classes é complicado. Devemos decidir quais as classes que representam generalizações e fazê-las classes bases. Porém, devemos prever quais serão suas possíveis especializações porque temos que definir a interface da classe e suas funções virtuais. Para isso, temos que ter um conhecimento profundo do problema e suas ramificações.

Por exemplo, consideremos a classe `tFiniteElement`, a qual representa um elemento finito genérico. Vimos no Capítulo 9 que um objeto que representa um elemento finito deve ser capaz de calcular sua própria matriz de rigidez e seu próprio vetor de esforços equivalentes, os quais constituem a contribuição do elemento à solução do problema. Definimos, então, na classe `tFiniteElement`, os métodos virtuais chamados `ComputeStiffnessMatrix()` e `ComputeLoadVector()`. A classe `tFiniteElement` implementa uma versão básica para ambos os métodos, mas como são virtuais, as classes derivadas direta ou indiretamente de `tFiniteElement` podem oferecer versões próprias para os métodos. Fizemos isso para o elemento de casca do Capítulo 6. Em função desses métodos virtuais, escrevemos a classe `tFESolver`, um analisador de estruturas elastostáticas pelo método dos elementos finitos.

Vamos supor, agora, que quiséssemos efetuar a análise dinâmica de uma estrutura por elementos finitos. Poderíamos inventar uma nova classe chamada, digamos, `tFEDynamicSolver`, derivada de `tFESolver`, na qual sobrecarregaríamos os métodos virtuais de `tFESolver` para implementarmos a algoritmo de Newmark, por exemplo. Esse é o procedimento correto em orientação a objetos. Derivamos uma nova classe

e reaproveitamos o código já existente. No entanto, isso não é o bastante. Temos que alterar a classe `tFiniteElement` também, pois quando escrevemos `tFiniteElement` não tínhamos em mente que utilizaríamos elementos finitos em análise dinâmica. Estávamos preocupados somente com análise estática. Agora precisamos não somente modificar a implementação da classe `tFiniteElement`, *mas também sua interface*, a qual passará a contar com um método virtual chamado `ComputeMassMatrix()`.

As alterações na interface de uma classe podem provocar alterações em outras classes de objetos que utilizam a classe. No entanto, essas perturbações no sistema podem ser atribuídas a falhas de análise e projeto, e não ao fato de estarmos programando com objetos. Além disso, todo o sistema de computador possui suas limitações, as quais são minimizadas através de sucessivas revisões e extensões do projeto original. Esperamos o mesmo para OSW.

12.2 Mais Problemas

Construímos OSW guiados por quatro propriedades que julgamos muito importantes em uma aplicação de modelagem estrutural: funcionalidade, extensibilidade, eficiência e facilidade de uso. Vamos analisar criticamente nossos resultados em relação a essas propriedades.

- *Funcionalidade.* As aplicações de modelagem desenvolvidas no trabalho possuem funcionalidade bastante limitada. Desenvolvemos somente elementos estruturais de superfície para análise elastostática de cascas delgadas e de sólidos simples. Os recursos de pré-processamento e de pós-processamento são igualmente simples. Esperamos, contudo, que esses recursos possam ser utilizados como fundação ou inspiração para o desenvolvimento de outras aplicações em engenharia de estruturas.

- *Extensibilidade.* As aplicações OSW são orientadas a objetos e, por isso, acreditamos que sua funcionalidade, embora limitada a princípio, possa crescer muito mais facilmente. Daremos algumas sugestões a seguir.

- *Eficiência.* Fizemos nosso melhor para implementarmos eficientemente os métodos das classes de OSW responsáveis pela computação gráfica e numérica. No entanto, não dispomos de dados quantitativos sobre velocidade de processamento e consumo de memória para compararmos a eficiência das aplicações OSW com outros programas de modelagem estrutural. Não tínhamos esse propósito. Nosso objetivo foi construir um *toolkit* que pudesse ser útil na construção de (eficientes) aplicações de análise e visualização de estruturas. Como discutido no Capítulo 1, podemos empregar o conceito de eficiência de uma maneira mais ampla. Se OSW nos permitir construir programas complexos mais rapidamente, com técnicas que facilitam o reaproveitamento e a extensão do código já existente, então podemos, desse ponto de vista, considerar o sistema eficiente.

- *Facilidade de uso.* Os programas podem ser mais fáceis de usar, mas a construção de interfaces mais amigáveis e com maior número de recursos é uma tarefa bastante complicada. Além disso, o desenvolvimento de uma aplicação gráfica orientada a objetos requer o conhecimento da estrutura e do comportamento de

muitas classes de objetos. Isso pode ser desestimulante, mas não é mais difícil, por exemplo, do que aprender os comandos do sistema UNIX.

OSW pode crescer em todas as direções e sentidos. Apresentaremos, a seguir, algumas sugestões, nas diversas disciplinas abordadas no trabalho.

Modelagem geométrica No Capítulo 3 definimos modelos gráficos, modelos de cascas, modelos de sólidos e modelos de decomposição por células. Discutimos também os processos de varredura translacional, varredura translacional cônica e varredura rotacional utilizados para geração de nossos modelos geométricos. Esses métodos de geração de modelos são adequados para a geração de alguns tipos simples de objetos, tais como esferas, blocos ou cascas cilíndricas. Mas objetos mais complexos, resultantes da união, intersecção e diferença de outros objetos, não podem ser gerados por tais processos. As operações de união, intersecção e diferença de modelos geométricos são chamadas *operações boolenas*. (MÄNTYLÄ [74] apresenta uma implementação de operações *boolenas* para modelos de sólidos com representação por fronteira, ou *b-rep*.) Outras formas de representações geométricas também podem ser acrescentadas a OSW, tais como esquemas de subdivisão de espaço (*octrees*, por exemplo) e CSG (geometria construtiva de sólidos, ou *constructive solid geometry*).

Modelagem matemática No Capítulo 4 apresentamos as equações matemáticas dos modelos matemáticos considerados no trabalho, derivadas da teoria da mecânica do contínuo. Restringimos nossa atenção a sólidos contínuos, homogêneos, isótropos e perfeitamente elásticos, os quais apresentam pequenas deformações e pequenos gradientes de deformações quando submetidos a forças de volume e de superfície estaticamente aplicadas. Elasticidade linear é um bom começo. Agora podemos considerar modelos matemáticos dinâmicos e não-lineares físicos e geométricos de estruturas.

Análise numérica No Capítulo 5 discutimos os princípios do método dos elementos finitos e do método dos elementos de contorno, bem como a aplicação desses métodos à resolução dos modelos matemáticos desenvolvidos no Capítulo 4. Podemos estender a aplicação dos métodos para os casos de análise dinâmica e não-linear. Além disso, há problemas onde nem o emprego individual de elementos finitos nem o emprego individual de elementos de contorno é ideal; nesses casos, o *acoplamento* desses métodos pode conduzir a técnicas mais eficientes de análise, como demonstrado pelos trabalhos de CODA [23] e PAIVA [85].

Modelagem mecânica No Capítulo 6 apresentamos as formulações dos elementos estruturais de nossos modelos mecânicos e descrevemos o processo de geração de malhas implementado em OSW. Implementamos um gerador de malhas de elementos de dimensão topológica 2, somente, porque os elementos finitos de casca e de contorno de sólidos utilizados nos modelos mecânicos, são elementos de superfície. A aplicação do algoritmo é restrita, mas o método é ponto de partida do processo de geração de malhas de elementos de volume por avanço de fronte [68]. Muitos outros tipos de elementos, casos de carregamento e geradores de malhas podem ser incorporados a OSW.

Visualização No Capítulo 7 introduzimos os algoritmos de transformação de dados de um modelo em imagens descritivas da estrutura e do comportamento do modelo. Ainda há muito a fazer. Por exemplo, algoritmos de visualização de vetores tais como *hedgehogs* e *glyphs*, gráficos de deslocamento (*displacement plot*) e linhas de fluxo (*streamlines*). Em visualização de tensores, elipsóides de tensão e linhas de hiperfluxo (*hyperstreamlines*) [101]. A computação gráfica também oferece um campo fascinante e absolutamente útil em engenharia. Por exemplo, um método de *rendering* chamado *radiosidade* [6] pode ser combinado com traçado de raios para obtermos imagens com qualidade (realmente) de fotografia. Em análise dinâmica, seria muito interessante a utilização de *animação* [15] para visualizarmos a estrutura e o comportamento dos objetos no decorrer do tempo. Tudo isso se torna muito mais atrativo, ou, talvez, só seja de fato atrativo, se pudermos utilizar uma interface que nos permita manipular interativamente esses recursos. Atrativo e complexo. Por isso utilizamos a orientação a objetos. (Usar a programação orientada a objetos para resolver um problema complexo não significa que o problema se tornará menos complexo. Acreditamos sim, baseados em nossos resultados, que o emprego da tecnologia de objetos pode nos ajudar a inventar soluções computacionais do problema mais facilmente.)

12.3 Comentários Finais

No Capítulo 9 tentamos mostrar, em seus aspectos gerais, como construir uma aplicação de modelagem orientada a objetos utilizando as classes de objetos de OSW. Vimos que, independentemente dos propósitos de uma aplicação, há componentes que são comuns a todos os programas em OSW. Por exemplo, uma aplicação sempre é um objeto de uma classe derivada de `tApplication`, o qual contém um objeto de uma classe derivada de `tDocument`, o qual contém uma coleção de objetos de classes derivadas de `tScene`, os quais contêm coleções de objetos das classes `tActor`, `tView`, `tCamera`, etc. Por isso, pensamos em inventar o *ambiente de desenvolvimento* (no Capítulo 1 chegamos a afirmar que OSW era constituído de bibliotecas de classes e do ambiente de desenvolvimento), um programa orientado a objetos de auxílio à criação de aplicações OSW. O objetivo do ambiente de desenvolvimento é assistir o usuário no uso dos recursos das bibliotecas de classes de OSW para geração e edição do código-fonte de um programa de modelagem.

Nossa pretensão era de termos um ambiente totalmente integrado, onde pudéssemos editar, compilar, executar e depurar um programa de modelagem. Para tal, necessitaríamos de ferramentas tais como um compilador e um ligador simbólico. Para o futuro, ainda continuamos com essa pretensão. Nos primeiros ensaios de OSW, chegamos até mesmo a elaborar um sistema de execução próprio [80], definido por uma linguagem, um compilador e uma máquina virtual. Conceitualmente interessante, mas uma máquina real processa números e imagens muito mais velozmente que uma máquina virtual (pelo menos mais velozmente que a nossa máquina virtual).

O ambiente de desenvolvimento ainda está em fase de análise e projeto. Pela frente, esperamos ter um bom programa. Em todos os sentidos.

Referências Bibliográficas

[1] ADEY, R.A.; BREBBIA, C.A. *Basic Computational Techniques for Engineers*. London, Pentech Press, 1983.

[2] AHO, A.V.; SETHI, R.; ULLMAN, J.D. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1988.

[3] ALLEN, D.N.G. *Relaxation Methods*. McGraw-Hill, 1955.

[4] ARGYRIS, J.H. *Energy Theorems and Structural Analysis*. Butterworth, 1960.

[5] ARRUDA, R.S.; LANDAU, L.; EBECKEN, N.F.F. Object-Oriented Structural Analysis in a Graphical Environment. In: TOPPING, B.H.V.; PAPADRAKAKIS, M., eds. *Artificial Intelligence and Object Oriented Approaches for Structural Engineering*, Civil-Comp Press, 1994. p.129-138

[6] ASHDOWN, A. *Radiosity: A Programmer's Perspective*. John Wiley & Sons, 1994.

[7] ATKINSON, M.P. et alli. An Approach to Persistent Programming. *The Computer Journal*, v. 26, n. 4, p.360-65, 1983.

[8] BASS, L.; COUTAZ, J. *Developing Software for the User Interface*. Addison-Wesley Publishing Company, 1991.

[9] BAUMGART, B. A Polyhedron Representation for Computer Vision. In: NATIONAL COMPUTER CONFERENCE, 1975. *Proceedings*. p.589-96

[10] BATOZ, J.L.; BATHE, K.J.; HO, L.W. A Study of Three-Node Triangular Plate Bending Elements. *International Journal for Numerical Methods in Engineering*, v.15, p.1771-1812, 1980.

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

[12] BERGAN, P.G.; FELIPPA, C.A. A Triangular Membrane Element with Rotational Degrees of Freedom. *Computer Methods in Applied Mechanics and Engineering*, v.50, p.25-69, 1985.

[13] BERGAN, P.G.; NYGARD, M.K. Finite Elements with Increased Freedom in Choosing Shape Functons. *IJE*, v.20, p.643-663, 1984.

[14] BOMME, P.; ZIMMERMANN, T.H. Towards Intelligent Objects in Finite Element Programming. In: TOPPING, B.H.V., ed. *Advances in Computational Structures Technology*, Civil-Comp Press, 1996, p.107-114

[15] BRADFORD, R.E. *Real-Time Animation Toolkit in C++*. John Wiley & Sons, 1995.

[16] BREBBIA, C.A. *The Boundary Element Method for Engineers*. London, Pentech Press, 1978.

[17] BREBBIA, C.A.; FERRANTE, A.J., eds. *The Finite Element Technique: An Introduction for Engineers*. Porto Alegre, Editora da UFRGS, 1975.

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

[19] BURATYNSKI, E.K. A Fully Automatic Three-Dimensional Mesh Generator for Complex Geometries. *International Journal for Numerical Methods in Engineering*, v.30, p.931-952, 1990.

[20] CARRIJO, E.C. *Aplicação do Elemento Finito DKT à Análise de Cascas*. São Carlos, 1995. Dissertação. (Mestrado) – Escola de Engenharia de São Carlos – USP.

[21] CÉSAR, C.N.L. *Um Módulo para Visualização Eficiente de Sóliods B-Rep*. São Carlos, 1995. Dissertação. (Mestrado) – Instituto de Ciências Matemáticas de São Carlos – USP.

[22] CLOUGH, R.W. The Finite Element in Plane Stress Analysis. In: ASCE CONFERENCE ON ELETRONIC COMPUTATION, 2. Pittsburgh, September, 1960. *Proceedings*.

[23] CODA, H.B. *Análise Tridimensional Transiente de Estruturas pela Combinação entre o Método dos Elementos de Contorno e o Método dos Elementos Finitos*. São Carlos, 1993. Tese (Doutorado) – Escola de Engenharia de São Carlos – USP.

[24] CODA, H.B.; VENTURINI, W.S. Three-Dimensional Transient BEM Analysis. *Computers and Structures*, v.56, n.5, p.751-68, 1995.

[25] COLLINS, R.J. Bandwidth Reduction by Automatic Renumbering. *International Journal for Numerical Methods in Engineering*, v.6, p.345-56, 1973.

[26] CORREA, M.R.S. *Aperfeiçoamento de Modelos usualmente empregados no Projeto de Sistemas Estruturais de Edifícios*. São Carlos, 1991. Tese (Doutorado) – Escola de Engenharia de São Carlos – USP.

[27] COX, B.J. *Object-Oriented Programming*. Addison-Wesley Publishing Company, 1986.

[28] CRUSE, T.A.; RIZZO, F.J. A Direct Formulation and Numeric Solution of the General Transient Elasto-Dynamic Problem - I. *J. Math. Anal. Appl.*, v.22, p.341-55, 1968.

[29] DAVENPORT, J.H. et alli. *Computer Algebra: Systems and Algorithms for Algebraic Computation*. San Diego, Academic Press Limited, 1988.

[30] DIGITALK. *Smalltalk/V 286 Tutorial and Programming Handbook*. Los Angeles, May, 1988.

[31] DEVLOO, P.R.B. Efficiency Issues in an Object-Oriented Programming Environment. In: TOPPING, B.H.V.; PAPADRAKAKIS, M., eds. *Artificial Intelligence and Object Oriented Approaches for Structural Engineering*, Civil-Comp Press, 1994. p.147-151

[32] DOI, A. et alli. Data Visualization Using a General-Purpose Renderer. *IBM Journal of Research and Development*, v.35, n.1, p.44-57, January, 1991.

[33] DUBOIS-PÈLERIN, Y.; ZIMMERMANN, T. Object-Oriented Finite Element Programming: I. Governing Principles. *Computer Methods in Applied Mechanics and Engineering*, n.98, p.291-303, 1992.

[34] DUBOIS-PÈLERIN, Y.; ZIMMERMANN, T. Object-Oriented Finite Element Programming: II. A Prototype Program in Smalltalk. *Computer Methods in Applied Mechanics and Engineering*, n.98, p.361-97, 1992.

[35] DUBOIS-PÈLERIN, Y.; ZIMMERMANN, T. Object-Oriented Finite Element Programming: III. An Efficient Implementation in C++. *Computer Methods in Applied Mechanics and Engineering*, n.108, p.165-83, 1993.

[36] EYHERAMENDY, D.; ZIMMERMANN, T. Object-Oriented Finite Element Programming: Beyond Fast Prototyping. In: TOPPING, B.H.V.; PAPADRAKAKIS, M., eds. *Artificial Intelligence and Object Oriented Approaches for Structural Engineering*, Civil-Comp Press, 1994. p.121-127

[37] FINLAYSON, B.A. *The Method of Weighted Residuals and Variationals Principles*. Academic Press Limited, 1972.

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

[39] GAJEWSKI, R.R. An Object Oriented Approach to Finite Element Programming. In: TOPPING, B.H.V.; PAPADRAKAKIS, M., eds. *Artificial Intelligence and Object Oriented Approaches for Structural Engineering*, Civil-Comp Press, 1994. p.107-113

[40] GAJEWSKI, R.R; LOMPIES, P. Object-Oriented Implementation of Bandwidth, Profile and Wavefront Reduction Algorithms. In: TOPPING, B.H.V., ed. *Advances in Computational Structures Technology*, Civil-Comp Press, 1996, p.115-119

[41] GALLAGHER, R.S. Scientific Visualization: an Engineering Perspective. In: GALLAGHER, R., ed. *Computer Graphics Techniques for Scientific and Engineering Analysis*. CRC Press, 1995. p3-16

[42] GALLAGHER, R.S. Scalar Visualization Techniques. In: GALLAGHER, R., ed. *Computer Graphics Techniques for Scientific and Engineering Analysis*. CRC Press, 1995. p89-128

[43] GALLAGHER, R.S. Future Trends in Scientific Visualization. In: GALLAGHER, R., ed. *Computer Graphics Techniques for Scientific and Engineering Analysis*. CRC Press, 1995. p291-304

[44] GHASSEMI, F. Automatic Mesh Generation Scheme for a Two- or Three-Dimensional Triangular Curved Surface. *Computers and Structures*, v.15, n.6, p.613-26, 1982.

[45] GIBBS, N.E.; POOLE, W.G.; STOCKMEYER, P.K. An Algorithm for Reducing the Bandwidth and Profile of a Sparse Matrix. *SIAM Journal of Numerical Analysis*, v.13, n.2, p.236-50, 1976.

[46] GLASSNER, A.S. et alli. An Introduction to Ray Tracing. In: SIGGRAPH 88. *Course Notes*, August, 1988.

[47] GOLDBERG, A.; ROBSON, D. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Publishing Company, 1983.

[48] GRAY, W.H.; AKIN, J.E. An Improved Method for Contouring on Isoparametric Surfaces. *International Journal for Numerical Methods in Engineering*, v.14, p.451-72, 1979.

[49] HAMILTON, W.R. *Lectures on Quaternions: Containing a Systematic Statement of a New Mathematical Method*. Hodges and Smith, 1843.

[50] HUANG, C.Y.; ODEN, J.T. Gamma2D: A Multiregion/Multiblock, Structured/Unstructured Grid Generation Packed for Computational Mechanics. *Computers and Structures*, v.53, n.2, p.375-410, 1994.

[51] HRENNIKOFF, A. Solution of Problems in Elasticity by the Framework Method. *J. Appl. Mech.*, v.A8, p.169-75, 1941.

[52] JAWSON, M.A. A Review of the Theory. In: BREBBIA, C.A., ed. *Topics in Boundary Element Research - 1*. Berlin, Springer-Verlag, 1985. p.13-40

[53] JEYACHANDRABOSE, C.; KIRKHOPE, J.; BABU, C.R. An Alternative Explicit Formulation for the DKT Plate-Bending Element. *IJE*, v.21, p.1289-93, 1985.

[54] JOHNSON C. *Numerical Solution of Partial Differential Equations by the Finite Element Method*. Cambridge University Press, 1987.

[55] JOY, K.I. et alli. Image Synthesis. In: SIGGRAPH 88. *Tutorial n.9*, August, 1988.

[56] JU, J.; HOSAIN, M.U. Substructuring using an Object-Oriented Approach. In: TOPPING, B.H.V.; PAPADRAKAKIS, M., eds. *Artificial Intelligence and Object Oriented Approaches for Structural Engineering*, Civil-Comp Press, 1994. p.115-120

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

[58] KHOSHAFIAN, S. *Object-Oriented Databases*. John Wiley & Sons, 1993.

[59] KHOSHAFIAN, S.; ABNOUS, R. *Object Orientation: Concepts, Languages, Databases, User Interfaces*. John Wiley & Sons, 1990.

[60] KHOSHAFIAN, S.; FRANKLIN, M.J.; Carey, M.J. Storage Management for Persistent Complex Objects. *Information Systems*, v.15, n.3, p.303-20, 1990.

[61] KONG, X.A.; CHEN, D.P. An Object-Oriented Design of FEM Programs. *Computers and Structures*, v.57, n.1, p.157-166, 1995.

[62] KOYAMADA, K.; NISHIO T. Volume Visualization of 3D Finite Element Method Results. *IBM Journal of Research and Development*, v.35, n.1, p.12-25, January, 1991.

[63] KUPRADZE, O.D. *Potencial Methods in the Theory of Elasticity*. Daniel Davey & Co., New York, 1965.

[64] LEDBETTER, L.; COX, B. Software - IC's. *Byte*, v.10, n.6, June, 1985.

[65] LIANG, Y.D.; BARSKY, B.A. A New Concept and Method for Line Clipping. *ACM Transactions on Graphics*, v.3, n.1, p.1-22, 1984.

[66] LINDLEY, C.A. *Pratical Ray Tracing in C*. John Wiley & Sons, 1992.

[67] LO, S.H.; CHEUNG, Y.K.; LEUNG, Y.T. An Algorithm to Display Three-Dimensional Objects. *Computers and Structures*, v.15, n.6, p.673-83, 1982.

[68] LÖHNER, R.; PARIKH, P. Generation of Three-Dimensional Unstructured Grids by the Advancing-Front Method. *International Journal for Numerical Methods in Fluids*, v.8, p.1131-1149, 1988.

[69] LORENSEN, W.E.; CLINE, H.E. Marching Cubes: a High Resolution 3D Surface Construction Algorithm. *Computer Graphics*, v.21, n.3, p.163-169, 1987.

[70] LU, J. et alli. A Matrix Class Library in C++ for Structural Engineering Computing. *Computers and Structures*, v.55, n.1, p.95-111, 1995.

[71] MACKIE, R.I. Objects, Elements and Complexity. In: TOPPING, B.V.H., ed. *Advances in Computational Structures Technology*. Civil-Comp Press, 1996. p.143-150

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

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

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

[75] MÄNTYLÄ, M. A Modeling System for Top-Down Design of Assembled Products. *IBM Journal of Research and Development*, v.34, n.5, p.636-58, September, 1990.

[76] MCHENRY, D. A Lattice Analogy for the Solution of Plane Stress Problems. *J. Inst. Civ. Eng.*, v.23, p.59-82, 1943.

[77] MENÉTREY, P.; ZIMMERMANN, T. Object-Oriented Non-Linear Finite Element Analysis: Application to J2 Plasticity. *Computers and Structures*, v.49, n.5, p.767-77, 1993.

[78] NEWMARK, N.M. Numerical Methods of Analysis in Bars, Plates and Elastic Bodies. In: GRINTER, L.E., ed. *Numerical Methods in Analysis in Engineering*, Macmillan, 1949.

[79] OWEN, D.R.J.; HINTON, E. *Finite Elements in Plasticity: Theory and Practice.* Swansea, Pineridge Press Limited, 1982.

[80] PAGLIOSA, P.A.; PAIVA, J.B. OSW: Um Analisador de Estruturas Reticulares Orientado a Objetos. In: CONGRESSO IBERO LATINO AMERICANO SOBRE MÉTODOS COMPUTACIONAIS PARA ENGENHARIA, 15., Belo Horizonte, 1994. *Anais.* v.1, p.433-42

[81] PAGLIOSA, P.A.; PAIVA, J.B. Elementos Finitos Orientados a Objetos em OSW. In: JORNADAS SUDAMERICANAS DE INGENIERIA ESTRUCTURAL, 27., Tucumán, Argentina, 1995. *Memorias.* v.3, p.455-66

[82] PAGLIOSA, P.A.; PAIVA, J.B. Elementos de Contorno Orientados a Objetos em OSW. In: JORNADAS SUL-AMERICANAS de ENGENHARIA ESTRUTURAL, 28., São Carlos, 1997. *Anais.*

[83] PAGLIOSA, P.A.; PAIVA, J.B. Modelagem Estrutural Orientada a Objetos em OSW. In: CONGRESSO IBERO LATINO AMERICANO SOBRE MÉTODOS COMPUTACIONAIS PARA ENGENHARIA, 18., Brasília, 1997. *Anais.*

[84] PAIVA, J.B. *Formulação do Método dos Elementos de Contorno para Flexão de Placas e suas Aplicações em Engenharia de Estruturas.* São Carlos, 1987. Tese (Doutorado) – Escola de Engenharia de São Carlos – USP.

[85] PAIVA, J.B. *Formulação do Método dos Elementos de Contorno para Análise da Interação Solo-Estrutura.* São Carlos, 1993. Tese (Livre-docência) – Escola de Engenharia de São Carlos – USP.

[86] PELETEIRO, S.C. *Utilização da Formulação Livre para Desenvolvimento de um Elemento de Membrana com Liberdades Rotacionais.* São Carlos, 1997. Dissertação. (Mestrado) – Escola de Engenharia de São Carlos – USP.

[87] PENNA, M.A.; PATTERSON, R.R. *Projective Geometry and its Applications to Computer Graphics.* Prentice-Hall International, 1986.

[88] PETZOLD, C. *Programming Windows 95.* Microsoft Press, 1996.

[89] PIDAPARTI, R.M.V.; HUDLI, A.V. Dynamic Analysis of Structures using Object-Oriented Techniques. *Computers and Structures*, v.49, n.1, p.149-156, 1993.

[90] PRESSMAN, R.S. *Software Engineering*. McGraw-Hill, 1992.

[91] PRZEMIENIECKI, J.S. *Theory of Matrix Structural Analysis*. McGraw-Hill, 1971.

[92] RAJASEKARAN, S.; VENKATESAN, K.G. A New Coutouring Algorithm. *Computers and Structures*, v.54, n.5, p.953-77, 1995.

[93] RAMALHO, M.A. *Sistema para Análise de Estruturas considerando Interação com Meio Elástico*. São Carlos, 1990. Tese (Doutorado) – Escola de Engenharia de São Carlos – USP.

[94] REQUICHA, A.A.G.; VOELCKER, H.B. *Constructive Solid Geometry*. Tech. Memo, n.25, Production Automation Project, University of Rochester, 1977.

[95] REQUICHA, A.A.G.; VOELCKER, H.B. Boolean Operations in Solid Modelling: Boundary Evaluation and Merging Algorithms. In: IEEE, Jan. 1985. *Proceedings*. v.1, p.30-44

[96] REQUICHA, A.A.G. Representation for Rigid Solids: Theory, Methods and Systems. *Computing Surveys*, v.12, n.4, p.437-63, December, 1980.

[97] RICHTER, J. *Advanced Windows NT*. Microsoft Press, 1994.

[98] ROGERS, D.F. *Mathematical Elements for Computer Graphics*. McGraw-Hill, 1976.

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

[100] RUMBAUGH, J. et alli. *Object-Oriented Modeling and Design*. Prentice-Hall International, 1991.

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

[102] SEZER, L.; ZEID, I. Automatic Quadrilateral/Triangular Free-Form Mesh Generation for Planar Regions. *International Journal for Numerical Methods in Engineering*, v.32, p.1441-83, 1991.

[103] SHEPARD, M.S.; GEORGES, M.K. Automatic Three-Dimensional Mesh Generation by the Finite Octree Technique. *IJE*, v.32, p.709-49, 1991.

[104] SHEPARD, M.S; SCHROEDER, W. Analysis Data for Visualization. In: GALLAGHER, R., ed. *Computer Graphics Techniques for Scientific and Engineering Analysis*. CRC Press, 1995. p61-87

[105] SHUEY, D. et alli. PHIGS: A Standard, Dynamic, Interactive Graphics Interface. *Computer Graphics and Applications*, p.50-7, August, 1986.

[106] SLOAN, S.W. An Algorithm for Profile and Wavefront Reduction of Sparse Matrices. International Journal for Numerical Methods in Engineering, v.23, p.239-51, 1986.

[107] SOUTHWELL, R.V. *Relaxation Methods in Theoretical Phisics*. London, Oxford University Press, 1946.

[108] STEVENS, A. *C++ Database Development*. MIS:Press, 1992.

[109] STEVENS, A. Persistent Objects in C++. *Dr. Dobb's Journal*, n. 195, p.34-44, December, 1992.

[110] STROUSTRUP, B. *The C++ Programming Language*. 2.ed. Addison-Wesley Publishing Company, 1994.

[111] TAKAHASHI, T.; LIESENBERG. H.K.E. Programação Orientada a Objetos. In: ESCOLA DE COMPUTAÇÃO, 7., São Paulo, 1990. *Livro*. São Paulo, IME-USP, 1990.

[112] TANEMBAUM, A.S. *Operating Systems: Design and Implementation*. Prentice-Hall International, 1987.

[113] TANEMBAUM, M.A.; LANGSAM, Y.; AUGENSTEIN, M.J. *Data Structures Using C*. New Jersey, Prentice-Hall International, 1990.

[114] THACKER, W.C. A Brief Review of Techniques for Generating Irregular Computational Grids. *International Journal for Numerical Methods in Engineering*, v.15, p.1335-41, 1980.

[115] THOMAS, D. What's in an Object? *Byte*, v.14, n.3, p.231-40, March, 1989.

[116] THOMPSON, J.F.; WARSI, Z.U.A.; MARTIN, C.W. *Numerical Grid Generation: Foundations and Applications*. North-Holland, 1985.

[117] TIMOSHENKO, S.P.; GOODIER, J.N. *Theory of Elasticity*. McGraw-Hill, 1970.

[118] TRUESDELL, C. *The Elements of Continuum Mechanics*. Springer-Verlag, 1965.

[119] TURNER, M.J. et alli. Stiffness and Deflections Analysis of Complex Structures. *J. Aero. Sci.*, v.23, p.805-23, 1956.

[120] ULBIN, M.; REN, Z.; FLASKER, J. Object Oriented Programming of Engineering Numerical Applications. In: TOPPING, B.V.H., ed. *Advances in Computational Structures Technology*. Civil-Comp Press, 1996. p137-142

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

[122] VENTURINI, W.S. *Um Estudo sobre o Método dos Elementos de Contorno e suas Aplicações em Problemas de Engenharia*. São Carlos, 1988. Tese (Livredocência) – Escola de Engenharia de São Carlos – USP.

[123] WALNUM, C. *3-D Graphics Programming with OpenGL*. Que, 1995.

[124] WATKINS, C.D. et alli. *Photorealism and Ray Tracing in C*. Prentice-Hall International, 1992.

[125] WEAVER, W. *Computer Programs for Structural Analysis*. D. Van Nostrand Company, 1967.

[126] WEAVER, W.; GERE, J.M. *Analysis of Framed Structures*. D. Van Nostrand Company, 1965.

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

[128] WINER, R.S.; PINSON, L.J. *An Introduction to Object-Oriented Programming and C++*. Addison-Wesley Publishing Company, 1988.

[129] WISSKIRCHEM, P. *Object-Oriented Graphics*. Springer-Verlag, 1990.

[130] ZEGLINSKI, G.W.; HAN, R.P.S; AITCHISON, P. Object-Oriented Matrix Classes for use in a Finite Element Code using C++. *International Journal for Numerical Methods in Engineering*, v.37, p.3921-37, 1994.

[131] ZIENKIEWICZ, O.C.; PHILLIPS, D.V. An Automatic Mesh Generation Scheme for Plane and Curved Surfaces by 'Isoparametric' Co-ordinates. *International Journal for Numerical Methods in Engineering*, v.3, p.519-28, 1971.

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

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