PRÉ-PROCESSADOR PARA MODELOS RETICULADOS E PLANOS DO MÉTODO DOS ELEMENTOS FINITOS

Reginaldo Lopes Ferreira

UNIVERSIDADE FEDERAL DE MINAS GERAIS
ESCOLA DE ENGENHARIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DE ESTRUTURAS

PRÉ-PROCESSADOR PARA MODELOS RETICULADOS E PLANOS DO MÉTODO DOS ELEMENTOS FINITOS

Reginaldo Lopes Ferreira

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia de Estruturas da Escola de Engenharia da Universidade Federal de Minas Gerais, como parte dos requisitos necessários à obtenção do título de "Mestre em Engenharia de Estruturas".

Comissão Examinadora:

Prof. Dr.Roque Luiz da Silva Pitangueira
DEES - UFMG (Orientador)

Prof. Dr. Felicio Bruzzi Barros
DEES - UFMG

Prof. Dr. Gray Farias Moita
CEFET - MG

Belo Horizonte, 09 de julho de 2008

Professores ideais são os que se fazem de pontes, que convidam os alunos a atravessarem, e depois, tendo facilitado a travessia, desmoronam-se com prazer, encorajando-os a criarem suas próprias pontes.

*(*Nikos Kazantzakis*)*

Dedico este trabalho aos meus pais, irmão e Juliana.

Índice

Lista de Tabelas

Lista de Figuras

Resumo

Esta dissertação de mestrado descreve um programa de pré-processamento, que é parte componente do sistema **INSANE** (*INteractive Structural ANalysis Environment*), para a criação de modelos do Método dos Elementos Finitos. Com ele é possível combinar elementos de barras, estado plano e placas. O software, implementado em JAVA, foi desenvolvido segundo o paradigma da programação orientada a objetos (POO) e apresenta uma arquitetura em camadas baseada na combinação de três padrões de projeto de software: o padrão *Model-View-Controller* (MVC), que permite a independência entre as camadas modelo e vista; o padrão *Observer* que, através de um mecanismo de propagação de mudanças, permite a sincronização do modelo com os diversos pares "vista-controlador"; e o padrão *Command* que encapsula as ações do programa em classes separadas de forma organizada. Esta segmentação favorece a expansão do programa e a criação de novos comandos. Para permitir diferentes discretizações de um mesmo modelo geométrico, o pré-processador foi dividido em dois módulos: *GEOMETRIA* e *MALHA*. O primeiro permite a criação e a edição de pontos, linhas, curvas e regiões. O segundo disponibiliza recursos para a discretização dos objetos geométricos e a definição de atributos. Sobre as regiões, podem ser geradas malhas de elementos de estado plano e placas, através dos métodos de Mapeamentos Transfinitos (Lofting, Bilinear ou Trilinear). Os dados do modelo geométrico e do modelo de malha são armazenados em estruturas de dados para subdivisão planar baseadas em *Semi-Arestas*. Estas permitem o armazenamento de forma organizada e estabelece as relações de adjacências, possibilitando um rápido acesso aos dados. Operadores de Euler são usados

para criar e manipular as entidades topológicas que compõem o modelo (vértices, arestas e faces) e representá-lo na estrutura de dados. O uso de múltiplas vistas combinado com a aplicação de transformações geométricas e projeções, possibilitam a visualização do modelo de vários ângulos e mais de um modelo podem ser simultaneamente observados. A persistência dos modelos pode ser feita em objetos JAVA ou arquivos XML.

Abstract

This Master's Thesis describes a pre-processing program, which is part of the **INSANE** (*INteractive Structural ANalysis Environment*) system, for the creation of Finite Element models. With it is possible to combine bar, plane state and plate elements. The software, implemented in JAVA, was developed according to the object oriented programming paradigm and has a layered architecture based on a combination of three design patterns: the pattern *Model-View-Controller* (MVC) that allows the independence between the model and view layers; the pattern *Observer* whith, through a mechanism of changing propagation, allows the synchronization of the model with various pairs "view-controller"; and the pattern *Command* which encapsulates the actions of the program in separate classes in an organized way. This segmentation favors the expansion of the program and the creation of new commands. To allow different discretizations of the same geometric model, the preprocessor was divided in two modules: GEOMETRY and MESH. The first one allow the creation and edition of points, lines, curves and regions. The second provides resources for the discretization of geometric objects and their attributes definition. On the regions, meshes of plane state and plates elements can be generated through the Transfinite Mapping (Lofting, Bilinear or Trilinear). The geometric model and the mesh model data are stored in data structures for planar subdivision which uses the half-edge concept. It allows the storage in an organized way and establishes the relations of adjacencies, making it possible a fast access to the data. Euler operators are used to create and to manipulate the topological entities that compose the model (vertices, edges and faces) and to represent it in the data structure. The use

of multiple views, combined with the application of geometric transformations and projections, allows different visualizations of a model and more than one model can be simultaneously observed. The models can be serialized as JAVA objects or XML files.

Agradecimentos

A *DEUS* por ter me dado saúde, coragem e inspiração durante toda a caminhada, permitindo a conquista de mais uma vitória .

A *meus pais* Antônio e Irene, ao meu irmão Reinaldo e a Juliana que me deram apoio e incentivo.

Ao professor *Roque Luiz da Silva Pitangueira*, que soube se fazer de ponte, me incentivando a enfrentar os desafios e contribuindo com o meu conhecimento e aperfeiçoamento profissional.

A toda equipe *INSANE* pelas contribuições dadas a esse e a todo o projeto.

Aos *professores* e *funcionários* do Departamento de Engenharia de Estruturas da $UFMG$ pela disponibilidade, carinho e atenção em todos os momentos a mim dedicados.

Ao CNPq - Conselho Nacional de Desenvolvimento Científico e Tecnológico pelo apoio financeiro na forma de bolsa.

A FAPEMIG - Fundação de Amparo à Pesquisa do Estado de Minas Gerais pelo apoio financeiro na forma de fomento à pesquisa.

Capítulo 1

INTRODUÇÃO

Na resolução dos problemas de engenharia, os modelos matemáticos, normalmente expressos por equações diferenciais, fornecem a solução em qualquer ponto do domínio. Porém, em muitos casos, não se conhece esta solução, chamada de solução analítica. Já os modelos discretos, obtidos a partir de uma aproximação numérica, são empregados com o objetivo de superar estas limitações. Dentre os métodos numéricos existentes, o Método dos Elementos Finitos (MEF) provou ser uma ferramenta poderosa para a análise de estruturas.

O grande desenvolvimento do MEF vem sendo sustentado por programas processadores capazes de trabalhar com modelos complexos e discretizações cada vez maiores. Atualmente, tais processadores são auxiliados por pré-processadores que permitem a criação do modelo geométrico e de elementos finitos, e por pós-processadores que representam as grandezas envolvidas no problema e auxiliam na interpretação dos resultados.

Historicamente, diversos softwares desenvolvidos pela comunidade acadêmica têm se mostrado totalmente dependentes de sistemas operacionais, escritos em linguagens de programação não apropriadas, de difícil expansão e/ou manutenção, com documentação deficiente, entre outras limitações. Estes inconvenientes são creditados à falta de disposição da comunidade de se apropriar das tecnologias emergentes ou mesmo à inexistência das mesmas. Porém, o avanço dos recursos para desenvolvimento de softwares, tais como o paradigma de Programação Orientada a Objeto

(POO); linguagens independentes de plataforma, como JAVA; linguagens de marcação, como XML (eXtensible Markup Language); linguagens de modelagem, como UML (Unified Modelling Language); sistemas de controle de versão, como SVN (Subversion); padrões de projeto de software; entre outros, permite o desenvolvimento de sistemas mais amigáveis (Alvim, 2003).

1.1 Objetivos do Trabalho

1.1.1 Objetivo Geral

O projeto **INSANE** (*INteractive Structural ANalysis Environment*), em realização no Departamento de Engenharia de Estruturas da Universidade Federal de Minas Gerais, disponível em `www.insane.dees.ufmg.br`, visa desenvolver software na área de métodos numéricos e computacionais aplicados à engenharia. Com base em modernos recursos tecnológicos, o projeto trabalha no aprimoramento de modelos discretos, utilizando um ambiente computacional bem segmentado e que permite alterações e ampliações.

Possui módulos para pré-processamento, processamento e pós-processamento implementados em linguagem JAVA, segundo o paradigma da Programação Orientada a Objeto. O pré-processador é responsável pelo fornecimento de recursos para a criação e edição de modelos. O processador representa o núcleo numérico e cuida da solução dos diferentes modelos discretos disponibilizados no sistema. O pós-processador é responsável pela apresentação de resultados como deslocamentos, deformações e tensões.

O presente trabalho acrescenta ao **INSANE** um novo módulo de pré-processamento, expandindo o projeto sem a necessidade de reescrever o que já foi implementado.

1.1.2 Objetivo Específico

O objetivo desta dissertação é a elaboração de um pré-processador gráfico e interativo para facilitar a criação de modelos que combinam elementos finitos de barras, estado plano e placas.

1.2 Organização do Texto

Esta dissertação é constituída de 5 capítulos.

No Capítulo 2 apresentam-se a Fundamentação Teórica e a Revisão Bibliográfica referentes aos conceitos de: programação orientada a objetos *(POO)*; linguagem JAVA; linguagem XML (eXtensible Markup Language); estrutura de dados para subdivisões planares *(semi-arestas)*; padrões de projeto de software *(MVC, Observer e Command)*; geração de malhas bidimensionais; transformações geométricas e projeções.

O Capítulo 3 descreve o Projeto e a Implementação do Pré-Processador com destaque para os requisitos desejados, a arquitetura do sistema, os sub-projetos, as principais classes e os relacionamentos entre elas, a forma como a estrutura de dados de "semi-arestas" foi empregada para a organização dos modelos, a utilização dos operadores de Euler e os recursos adotados da linguagem JAVA.

O Capítulo 4 relaciona os recursos disponibilizados no *Módulo Geometria* e no *Módulo Malha* do pré-processador. São apresentados os comandos para criação e edição da geometria, os recursos para geração de malhas, os comandos para definição de atributos e os comandos para o processamento e a persistência do modelo. Alguns exemplos de modelos gerados com o pré-processador também são apresentados no final desse capítulo.

Finalizando, no Capítulo 5, apresentam-se as contribuições deste trabalho e as sugestões para trabalhos futuros.

Capítulo 2

FUNDAMENTAÇÃO TEÓRICA E REVISÃO BIBLIOGRÁFICA

2.1 Programação Orientada a Objetos

Programação Orientada a Objetos ou, abreviadamente, *POO*, é um paradigma de programação de computadores baseado em classes e objetos, criados a partir de modelos (representações simplificadas de conceitos, pessoas, tarefas, etc), para representar e processar os dados desses modelos.

Em POO, os dados são representados por tipos nativos, característicos da linguagem de programação, por modelos já existentes na linguagem ou por outros criados pelo programador (Santos, 2003).

2.1.1 Classes e Objetos

Camarão e Figueiredo (2003) definem: uma *classe* é um componente de programa que descreve a "estrutura" e o "comportamento" de um grupo de objetos semelhantes - isto é, as informações que caracterizam o estado desses objetos e as ações (ou operações) que eles podem realizar. Uma classe é formada, essencialmente, por *construtores de objetos* dessa classe, *variáveis de instância* e *métodos*.

Os *objetos* de uma classe - também chamados de instâncias da classe - são criados durante a execução de programas. A criação de um objeto consiste na criação de cada

uma das variáveis especificadas na classe. Os valores armazenados nessas variáveis determinam o estado do objeto. Uma variável de um objeto é também chamada de atributo desse objeto.

Objetos podem receber mensagens, ou seja, uma chamada a um método específico para realizar uma determinada operação. A execução de uma chamada a um método de um objeto pode modificar o estado desse objeto (modificar os valores dos seus atributos) e pode retornar um resultado.

2.1.2 Coleções de Objetos

Ainda em Camarão e Figueiredo (2003) encontra-se uma caracterização e diferenciação clara de classes e objetos.

Muitas aplicações para as quais deseja-se desenvolver programas, consistem em sistemas bastante complexos. Uma maneira natural de lidar com a complexidade de um sistema é dividi-lo em subsistemas mais simples, de maneira que o comportamento do sistema, como um todo, possa ser expresso em termos dos comportamentos de seus subsistemas e das interações entre eles.

O desenvolvimento software para modelagem de um sistema envolve fases de análise, projeto e implementação desse sistema. O princípio em que se baseia o paradigma de orientação a objetos, o de que existe uma correspondência entre componentes do sistema e objetos, torna mais simples esse processo. Objetos constituem limites naturais para construções de *abstrações de dados*: todas as informações referentes a uma dada entidade são confinadas em um determinado objeto, que se relaciona com outros objetos mediante uma interface bem definida.

Linguagens de programação mais modernas oferecem recursos para construção de um programa como uma coleção de componentes, com interfaces bem definidas, que especificam as interações entre esses componentes. Um programa nessas linguagens consiste em uma coleção de definições de classes, que descrevem os objetos que implementam entidades de um sistema.

Os conceitos de Abstração, Encapsulamento, Modularidade, Herança e Polimorfismo são detalhadamente descritos em Goodrich e Tamassia (2002), conforme resumo a seguir.

2.1.3 Abstração

Abstrair significa decompor um sistema complicado em suas partes fundamentais e descrevê-las em uma linguagem simples e precisa. A descrição das partes de um sistema implica atribuir-lhes um nome e descrever suas funcionalidades. Por exemplo, a interface gráfica com o usuário de um editor de textos compreende a abstração de um menu "editar" que oferece várias opções de edição de texto incluindo recortar e colar porções de texto ou outros objetos gráficos. Sem entrar em detalhes sobre como uma interface gráfica com o usuário representa e exibe textos ou objetos gráficos, os conceitos de "recortar" e "colar" são simples e precisos. Uma operação de recorte apaga o texto ou gráfico selecionado e o coloca em uma área de armazenamento externa. A operação de colagem insere o conteúdo externamente armazenado em uma localizção específica do texto. Dessa forma, a funcionalidade abstrata do menu "editar" e suas operações de recortar e colar são definidas em uma linguagem precisa o suficiente para ser clara e simples o bastante para "abstrair" os detalhes desnecessários. Essa combinação de clareza e simplicidade traz benefícios a robustez, uma vez que leva a implementações corretas e compreensíveis.

2.1.4 Encapsulamento

Em projeto orientado a objetos, o conceito de encapsulamento estabelece que os diferentes componentes de um sistema de software não devem revelar detalhes internos de suas respectivas implementações. Genericamente, propõe que todos os componentes operem dentro de uma filosofia de conhecer o mínimo necessário sobre os demais.

Uma das maiores vantagens do encapsulamento é que ele oferece ao programador

liberdade na implementação dos detalhes do sistema. A única restrição ao programador é manter a interface abstrata que é percebida pelos de fora. Por exemplo, o programador do código do menu "editar" da interface gráfica com o usuário de um editor de textos pode, em um primeiro momento, implementar as operações de copiar e colar copiando e restaurando telas para a área externa de armazenamento. Mais tarde, pode ficar insatisfeito com essa implementação, uma vez que não permite um armazenamento compacto da seleção e não distingue objetos gráficos de textos. Se o programador tiver projetado a interface das operações de copiar e colar tendo em mente o encapsulamento, trocar a implementação por uma que armazene o texto como texto e os objetos gráficos em uma forma compacta apropriada não irá causar nenhum problema aos métodos que necessitam interagir com esta interface gráfica. Dessa forma, encapsulamento permite a adaptação, porque autoriza a alteração de detalhes de partes de um programa sem afetar de forma negativa outros componentes.

2.1.5 Modularidade

Outro conceito fundamental de projeto orientado a objetos é a *modularidade*. Ela refere-se a uma estrutura de organização na qual os diferentes componentes de um sistema de software são divididos em unidades funcionais separadas. A estrutura imposta pela *modularidade* auxilia a tornar o software reutilizável. Se os módulos do software forem escritos de uma forma abstrata para resolver problemas genéricos, eles podem ser reutilizados em outros contextos quando instâncias do mesmo problema geral surgirem.

2.1.6 Herança

Para evitar código redundante, o paradigma de orientação a objetos oferece uma estrutura hierárquica e modular para reutilização de código através de uma técnica

conhecida como *herança*. Esta técnica permite projetar classes genéricas (*superclasses*) que podem ser especializadas em classes mais particulares (*subclasses*), que por sua vez, reutilizam o código das mais genéricas. A classe genérica define variáveis de instância "genéricas" e métodos que se aplicam em uma variada gama de situações. A classe que especializa uma *superclasse* não necessita fornecer uma nova implementação para os métodos genéricos, uma vez que os herda. Deve apenas definir métodos que são particulares a esta *subclasse*.

2.1.7 Polimorfismo

"Polimorfismo" significa, literalmente, "muitas formas". No contexto de projeto orientado a objetos, entretanto, refere-se à habilidade de uma variável de objeto de assumir formas diferentes. Linguagens orientadas a objetos referenciam objetos usando variáveis referência. Uma variável referência $\boldsymbol{o}$ deve especificar que tipo de objeto ela é capaz de referenciar em termos de uma classe $\boldsymbol{S}$. Isso implica, entretanto, que $\boldsymbol{o}$ também pode se referir a qualquer objeto pertencente à classe $\boldsymbol{T}$, derivada de $\boldsymbol{S}$. Analise agora o que acontece se $\boldsymbol{S}$ define um método *a()* e $\boldsymbol{T}$ também define um método *a()*. A seqüência de ativação de métodos sempre é iniciada com a busca pela classe mais restritiva à qual se aplica. Ou seja, quando $\boldsymbol{o}$ se refere a um objeto da classe $\boldsymbol{T}$ e *o.a()* é invocado, então será ativada a versão de $\boldsymbol{T}$ do método *a()*, em lugar da versão de $\boldsymbol{S}$. Neste caso, diz-se que $\boldsymbol{T}$ sobrescreve o método *a()* de $\boldsymbol{S}$. Por outro lado, se $\boldsymbol{o}$ se refere a um objeto da classe $\boldsymbol{S}$ (que, ao contrário, não é um objeto da classe $\boldsymbol{T}$), quando *o.a()* for ativado, será executada a versão de $\boldsymbol{S}$ de *a()*. Um polimorfismo como esse é útil porque aquele que chama *o.a()* não precisa saber quando $\boldsymbol{o}$ se refere a uma instância de $\boldsymbol{T}$ ou $\boldsymbol{S}$ para poder executar a versão correta de *a()*. Dessa forma, a variável de objeto $\boldsymbol{o}$ pode ser polimórfica, ou assumir muitas formas, dependendo da classe específica dos objetos aos quais está se referindo. Esse tipo de funcionalidade permite a uma classe especializada $\boldsymbol{T}$ estender uma classe $\boldsymbol{S}$, herdar os métodos genéricos de $\boldsymbol{S}$ e redefinir outros métodos de $\boldsymbol{T}$, de maneira que

sejam incluídos como propriedades específicas dos objetos ***T***.

Algumas linguagens orientadas a objetos também oferecem um tipo de polimorfismo "em cascata", que é mais precisamente conhecido como sobrecarga de métodos. A sobrecarga ocorre quando uma única classe ***T*** tem vários métodos com o mesmo nome, desde que cada um tenha uma assinatura diferente. A assinatura de um método é uma combinação entre seu nome e o tipo e a quantidade de argumentos que são passados para o mesmo. Dessa forma, mesmo que vários métodos de uma classe tenham o mesmo nome, eles são distinguíveis pelo compilador pelo fato de terem diferentes assinaturas, ou seja, na verdade são desiguais. Em linguagens que possibilitam a sobrecarga de métodos, o ambiente de execução determina qual método ativar para uma determinada chamada de método que percorre a hierarquia de classes em busca do primeiro método cuja assinatura combine com a do método que está sendo invocado. Por exemplo, imagine uma classe ***T*** que define o método *a()*, derivada da classe ***U*** que define o método *a(x,y)*. Se um objeto ***o*** da classe ***T*** recebe a mensagem "$o.a(x, y)''$, então a versão de ***U*** do método *a()* é ativada (com os dois parâmetros x e y). Assim, o verdadeiro polimorfismo aplica-se apenas a métodos que têm a mesma assinatura mas estão definidos em classes diferentes.

A herança, o polimorfismo e a sobrecarga de métodos suportam o desenvolvimento de software reutilizável. Pode-se estabelecer classes que herdam as variáveis e os métodos de instância genéricos e que podem, a seguir, definir novas variáveis e métodos de instância mais específicos que lidam com os aspectos particulares dos objetos da nova classe.

2.2 Linguagem JAVA

Conforme Santos (2003), existem diversas linguagens de programação orientadas a objeto, cada uma com diferentes características e apelos de mercado, educacionais ou acadêmicos. A seguir, são apresentadas as razões que motivaram a escolha da

linguagem JAVA para a implementação deste trabalho de dissertação.

1. *JAVA é obrigatoriamente orientada a objetos.* Algumas linguagens permitem que funções e variáveis existam em diversos pontos de um programa, como se estivessem desatreladas de qualquer estrutura. Em JAVA, todas as variáveis e métodos devem estar localizados dentro de classes, forçando o uso de orientação a objetos até mesmo em tarefas simples.

2. *JAVA é simples.* A estrutura de programas e classes em JAVA segue a organização de linguagens tradicionais como C e C++, mas sem elementos que tornam programas e programação mais complexos. A simplicidade se reflete também na maneira como arquivos contendo programas em JAVA são compilados e executados: o compilador se encarregará de compilar todas as classes necessárias em uma aplicação automaticamente, sem a necessidade de arquivos adicionais de configuração e inclusão de bibliotecas.

3. *JAVA é portável.* O código-fonte de um programa em JAVA pode ser compilado em qualquer computador, usando qualquer sistema operacional, contanto que este tenha uma *Máquina Virtual JAVA (JVM)* adequada. A independência de plataformas, aumenta a utilidade da linguagem.

4. *JAVA é gratuita.* JAVA possui um ambiente de desenvolvimento gratuito, chamado *JAVA Development Kit* ou *JDK*. Ele contém a máquina virtual JAVA, anteriormente mencionada, e está disponível para cópia no site da Sun Microsystems (http://java.sun.com).

5. *JAVA é robusta.* Administração de memória (alocação e liberação) e o uso de ponteiros, duas das fontes de erros e *bugs* mais freqüentes em programas em C e C++, são administrados internamente na linguagem, de forma transparente para o programador. De maneira geral, programas em JAVA têm restrições no acesso à memória que resultam em maior segurança para os programas sem

diminuir a utilidade dos mesmos. JAVA também tem um poderoso mecanismo de exceções que permite melhor tratamento de erros em tempo de execução dos programas (Santos, 2003).

6. *JAVA tem bibliotecas prontas para diversas aplicações.* As bibliotecas de classes JAVA contêm várias classes que implementam diversos mecanismos de entrada e saída, acesso à Internet, manipulação de strings, poderosas estruturas de dados, utilitários diversos e um conjunto completo de classes para implementação de interfaces gráficas. Essas bibliotecas são padrão de JAVA - qualquer máquina virtual JAVA permite o uso dessas bibliotecas, sem a necessidade de instalar pacotes adicionais (Santos, 2003).

2.3 Linguagem XML

O armazenamento de dados em variáveis e arranjos (vetores e matrizes) é temporário - os dados são perdidos quando uma variável local "sai do escopo" ou quando o programa termina. Arquivos são utilizados para retenção a longo prazo de grandes quantidades de dados, mesmo depois de terminar a execução do programa. Os dados mantidos em arquivos são freqüentemente chamados de dados persistentes (Deitel e Deitel, 2003).

A *linguagem XML (eXtensible Markup Language)* foi adotada na implementação deste trabalho para se fazer a persistência dos dados. Trata-se de um formato universal padronizado de arquivo texto, projetado para escrever e estruturar dados. Seu grande diferencial está na possibilidade de se processar a informação contida no documento, além de ser independente de linguagem de programação ou sistema operacional (Lozano, 2003).

A plataforma de desenvolvimento JAVA oferece todas as API´s (*Aplication Program Interfaces*) necessárias à escrita de programas capazes de ler, criar e editar documentos XML. Tais API´s permitem a leitura e a escrita dos documentos em

arquivos, conexões TCP/IP, strings e outros meios de entrada/saída (Liesenfeld, 2002).

2.4 Estruturas de Dados

O desenvolvimento de modelos geométricos requer o armazenamento de dados que possibilitem a manipulação de construções e operações geométricas de forma estruturada e acessível. Tendo isto em vista, são usadas estruturas de adjacências no tratamento de modelos computacionais.

As estruturas de dados para subdivisões planares, ou simplesmente estruturas de adjacências, são uma forma organizada de armazenamento de dados de modelos geométricos computacionais, possibilitando um rápido acesso e garantindo a consistência destes dados.

2.4.1 Estrutura de dados "Winged-Edge" ou "Arestas-Aladas"

A estrutura de "arestas aladas" tem as *arestas de uma subdivisão planar* como a principal informação do modelo geométrico, embora também faça uso dos vértices e faces para armazenamento de dados. Na Figura 2.1 tem-se o caso da subdivisão de um cubo unitário (Figueiredo e Carvalho, 1991).

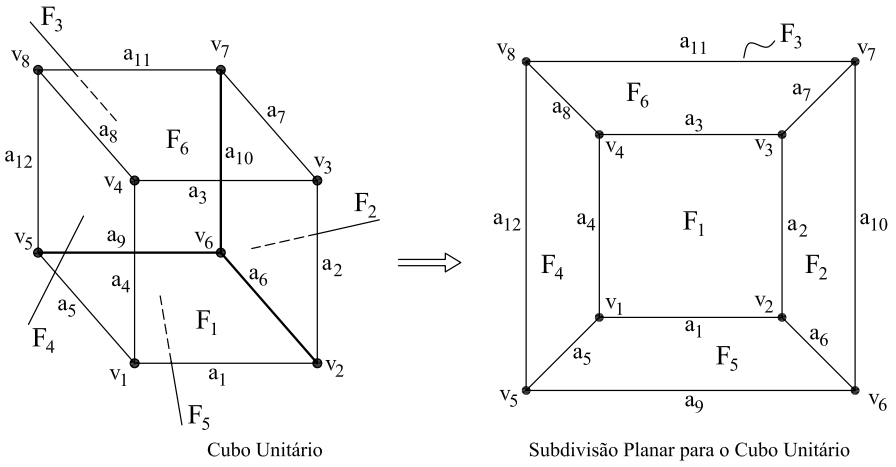


Figura 2.1: Cubo Unitário.

Nesta estrutura, os vértices, as faces e as arestas são organizados em listas sendo cada lista preenchida de acordo com o tipo de elemento a ser representado. A lista de **vértices** armazena as coordenadas (x, y, z) e uma das arestas (**av**) incidente em cada vértice. A lista de **faces** armazena as faces e somente uma de suas arestas (**af**). As **arestas** são os elementos que armazenam o maior número de dados. As tabelas 2.1, 2.2 e 2.3 exemplificam essa organização para a face 1 do cubo unitário da Figura 2.1.

Tabela 2.1: Vértices

Vértices	**Coordenadas**	**av**
v_1	(1,0,0)	a_1
v_2	(1,1,0)	a_2

Tabela 2.2: Faces

Faces	**af**
F_1	a_1

Tabela 2.3: Arestas

Arestas	v_i	v_j	**fccw**	**fcw**	**pccw**	**nccw**	**pcw**	**ncw**
a_1	v_1	v_2	F_1	F_5	a_4	a_2	a_6	a_5
a_2	v_2	v_3	F_1	F_2	a_1	a_3	a_7	a_6
a_3	v_3	v_4	F_1	F_6	a_2	a_4	a_8	a_7
a_4	v_4	v_1	F_1	F_4	a_3	a_1	a_5	a_8

Cada uma das arestas é representada por um par de vértices (vértices $\mathbf{v}_1$ e $\mathbf{v}_2$ na Figura 2.2), que são informados na ordem de orientação da aresta.

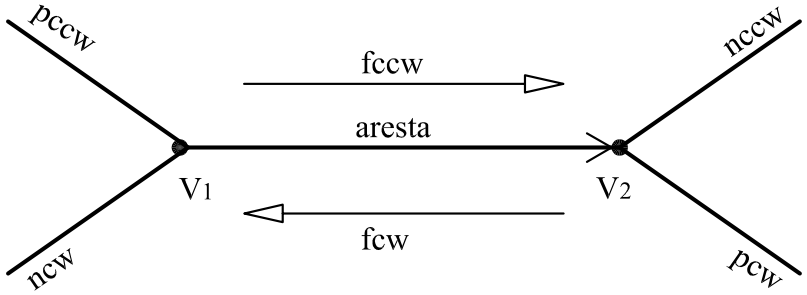


Figura 2.2: Representação da aresta na estrutura *"Winged-Edge"*.

A orientação da aresta proporciona a localização das faces adjacentes, sendo a face à esquerda da aresta chamada de **face counterclockwise** ou **fccw**, pois o sentido da aresta nesta face é anti-horário. Por sua vez, a face à direta é chamada de **face clockwise**, representada por **fcw**, que possui sentido horário de acordo com a orientação da aresta. Seguindo a mesma representação, são guardadas na lista das arestas, as arestas anteriores e posteriores incidentes nos vértices da aresta analisada. Portanto tem-se **previous counterclockwise** (**pccw**), e **next counterclockwise** (**nccw**), as arestas anterior e posterior no sentido da aresta, **previous clockwise** (**pcw**) e **next clockwise** (**ncw**), no sentido contrário ao da aresta.

2.4.2 Estrutura de dados "Half-Edge" ou "Semi-Arestas"

A estrutura de semi-arestas é uma variação da estruturas de "arestas-aladas", diferenciando-se pela sua organização topológica. A principal alteração é a transformação de uma aresta em duas semi-arestas (Figura 2.3).

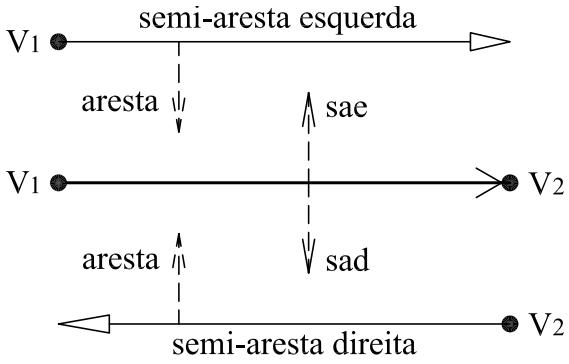


Figura 2.3: Representação da aresta na estrutura *"Half-Edge"*.

Conforme Mäntylä (1987), esta estrutura se desenvolve por uma hierarquia composta de cinco níveis (Figura 2.4), representando as entidades ***Subdivisão Planar***, ***Face***, ***Ciclo***, ***Semi-Aresta*** e ***Vértice***. Cada nível possui uma lista duplamente encadeada (anterior/próximo) para acesso às entidades adjacentes e também possui referências para acesso aos outros níveis.

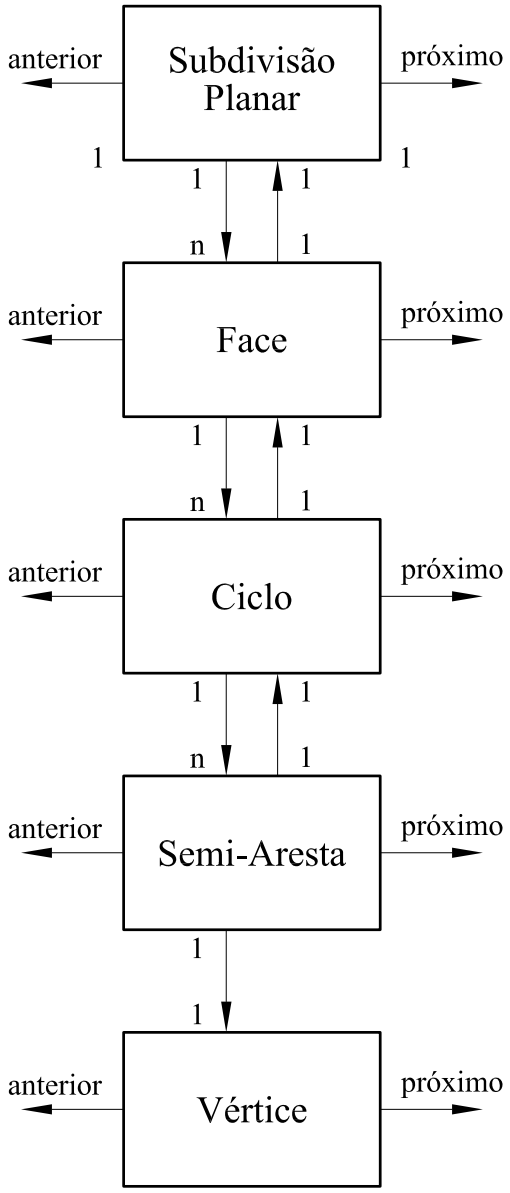


Figura 2.4: Estrutura de *"Semi-Arestas"*.

Na hierarquia anteriormente apresentada não foi feita a conexão entre as faces que compõem o modelo. Para isso, Del Savio et al. (2004) propuseram adicionar o nível intermediário ***Aresta***, conforme Figura 2.5. Este nível possui uma lista duplamente encadeada para as arestas do modelo e uma referência para ***Semi-Aresta***.

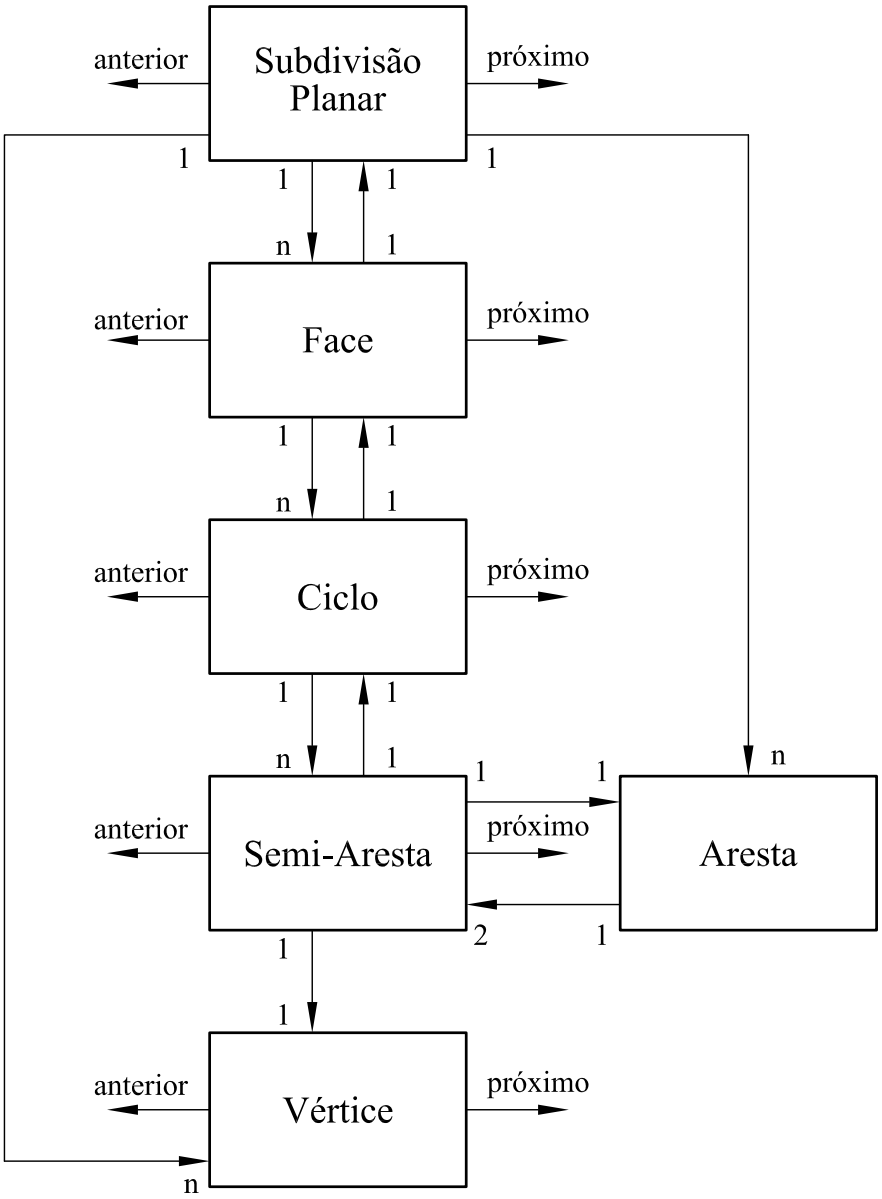


Figura 2.5: Estrutura de *"Semi-Arestas"* modificada.

Sendo assim, a ***Subdivisão Planar*** passa a ter uma referência para a ***Aresta*** e para o ***Vértice***. Já a ***Semi-Aresta*** passa a ter uma referência para a sua respectiva ***Aresta***. Os termos vértice, aresta e face são provenientes do fato de que subdivisões planares representam a topologia (ligação entre os componentes) da fronteira de sólidos cuja forma se assemelha a uma esfera, ou seja, cada ponto da fronteira tem uma vizinhança que é mapeável para um disco aberto 2D.

A estrutura de "Semi-Arestas" permite o armazenamento dos dados de forma organizada, estabelecendo as relações de adjacências entre os componentes do modelo. Seguindo a ordem da hierarquia, possibilita um rápido acesso a esses componentes por consulta simples. As tabelas 2.4, 2.5, 2.6 e 2.7 mostram como são armazenados os dados do modelo ilustrado pela Figura 2.6.

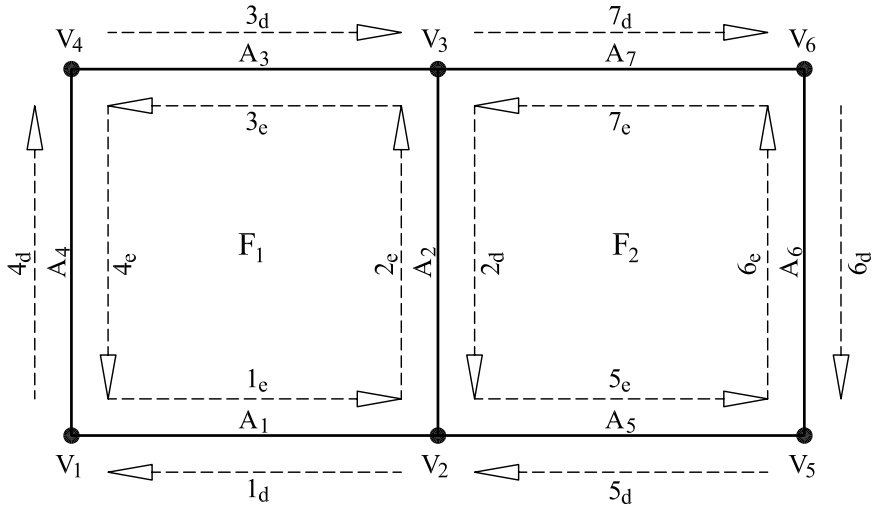


Figura 2.6: Exemplo de um modelo geométrico e seus componentes.

Tabela 2.4: Subdivisão Planar

Faces	**Arestas**	**Vértices**
F_1	A_1	V_1
F_2	A_2	V_2
	A_3	V_3
	A_4	V_4
	A_5	V_5
	A_6	V_6
	A_7	

Tabela 2.5: Faces e Ciclos

Face	**Ciclo**	**Semi-Arestas**
1	1	1_e, 2_e, 3_e, 4_e
2	2	5_d, 6_d, 7_d, 2_d

Tabela 2.6: Semi-Arestas

SA	**Ciclo**	$v_{inicial}$
1_e	1	v_1
2_e	1	v_2
3_e	1	v_3
4_e	1	v_4
5_e	2	v_2
6_e	2	v_5
7_e	2	v_6
2_d	2	v_3

Tabela 2.7: Arestas e Semi-Arestas

Aresta	v_i	v_j	SA_s
A_1	v_1	v_2	1_e, 1_d
A_2	v_2	v_3	2_e, 2_d
A_3	v_3	v_4	3_e, 3_d
A_4	v_4	v_1	4_e, 4_d
A_5	v_2	v_5	5_e, 5_d
A_6	v_5	v_6	6_e, 6_d
A_7	v_6	v_3	7_e, 7_d

A estrutura de dados de "Semi-Arestas" pode armazenar nove relações de adjacências entre os vértices, as arestas e as faces, conforme ilustra a Figura 2.7.

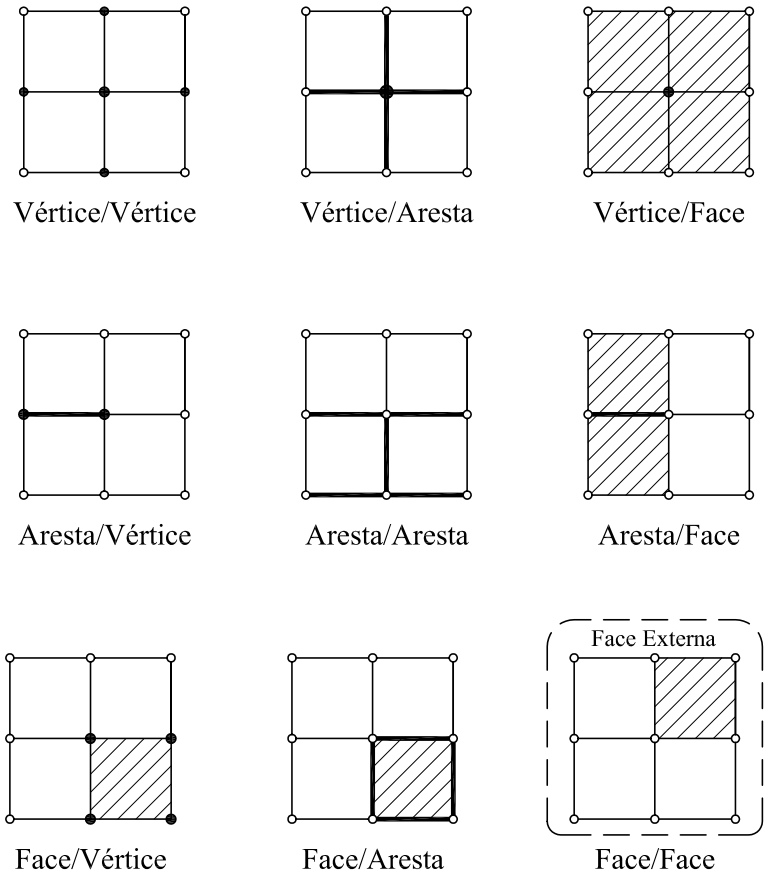


Figura 2.7: Relações de adjacências.

2.4.3 Operadores de Euler

Os operadores de Euler foram originalmente introduzidos por Baumgart (1974), no contexto da estrutura de "arestas aladas" (***winged-edge***). Para discussão dos operadores de Euler no contexto da estrutura de "semi-arestas", basta acrescentar o nível ***Ciclo***. Por conveniência de linguagem, os *ciclos* internos são chamados de *rings* e representam furos numa face. Os *ciclos* que consistem de apenas um vértice, que por sua vez não possui uma aresta, são chamados de *empty loops*.

Os operadores de Euler (Mäntylä, 1987) criam e manipulam as entidades topológicas (*vértices, arestas, faces*) que compõem um modelo sólido, para representá-lo

numa estrutura de dados (Figura 2.8).

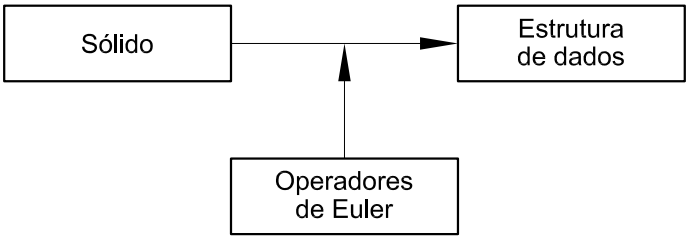


Figura 2.8: Atuação dos operadores de Euler.

Por convenção histórica, os operadores de Euler são normalmente denotados por códigos de nomes abreviados, por exemplo: MEV = *Make Edge, Vertex*. As chaves para esses nomes são usadas conforme indicado na Tabela 2.8

Tabela 2.8: Chaves para os Operadores de Euler

Chave	**Descrição**	**Chave**	**Descrição**	**Chave**	**Descrição**
M	make	V	vertex	H	hole
K	kill	E	edge	R	ring
S	split	F	face		
J	join	S	solid		

Uma seqüência finita de operadores de Euler é capaz de criar uma estrutura de dados segura, de maneira que as relações de conectividade entre as entidades topológicas fiquem consistentes. Para garantir essa consistência, a chamada *fórmula de Euler-Poincaré* (Mäntylä, 1987), dada abaixo, deve ser satisfeita.

$$V - E + F = 2(S - H) + R \tag{2.1}$$

Os operadores de Euler são divididos em duas categorias: os Locais, que alteram a topologia de apenas uma parte do modelo e os Globais, que alteram globalmente a topologia.

2.4.3.1 Operadores para Manipulações Locais

O operador **MVFS** (*Make Vertex, Face, Solid*) adiciona na estrutura de dados apenas um vértice solitário (Figura 2.9). Com isso a nova face possui um ciclo vazio e sem arestas. O "sólido" criado não satisfaz a noção intuitiva de um objeto sólido. Apesar disso, esse operador é útil como condição inicial para a criação de uma estrutura de adjacência. Ele define um "protótipo" que cria o esqueleto de modelos. O operador **KVFS** (*Kill Vertex, Face, Solid*) realiza a função inversa.

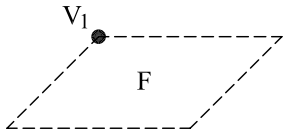
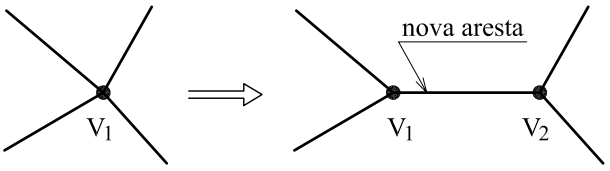


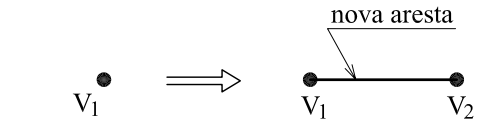
Figura 2.9: Operador MVFS.

O operador **MEV** (*Make Edge, Vertex*) adiciona uma aresta e um vértice na estrutura de dados. Portanto, subdivide o ciclo de arestas de um vértice em dois ciclos pela partição de um vértice em dois, unidos com uma nova aresta (Figura 2.10(a)). Sua aplicabilidade pode ser estendida para vértices solitários (Figura 2.11(b)) e para o caso de um novo vértice ser ligado, por meio de uma nova aresta, a um vértice existente (Figura 2.11(c)). O operador **KEV** (*Kill Edge, Vertex*) realiza a função inversa.

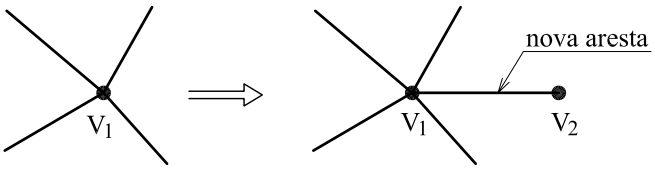


(a): partição de um vértice em dois e inclusão de aresta.

Figura 2.10: Operador MEV - caso a.



(b): vértice solitário unido a outro pela nova aresta.



(c): novo vértice ligado por nova aresta ao vértice existente.

Figura 2.11: Operador MEV - casos b e c.

O operador **MEF** (*Make Edge, Face*) adiciona uma aresta e uma face na estrutura de dados. Juntando dois vértices com uma nova aresta, ele subdivide um ciclo de uma face criando outra face (Figura 2.12). A função inversa é obtida com o operador **KEF** (*Kill Edge, Face*).

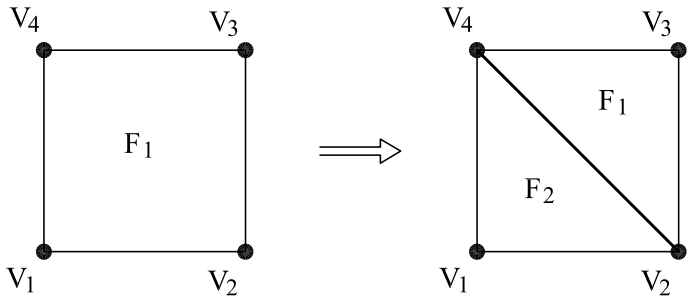
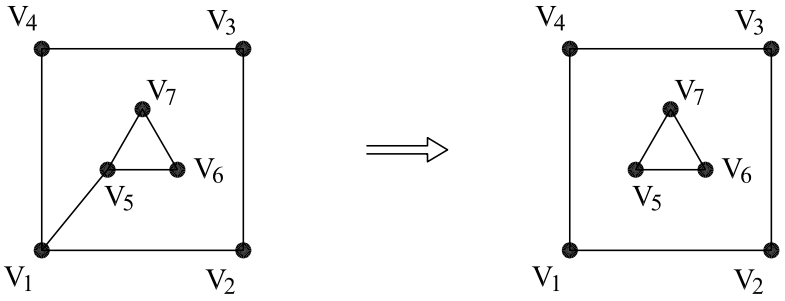
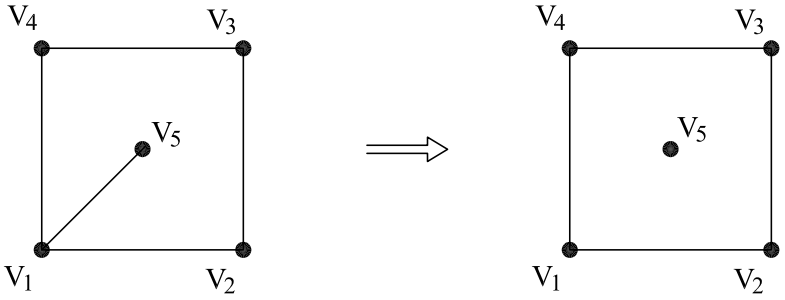


Figura 2.12: Operador MEF.

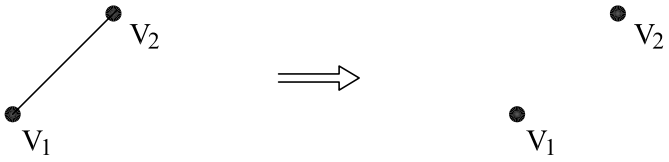
O operador **KEMR** (*Kill Edge, Make Ring*) remove uma aresta e cria um anel na estrutura de dados. Particiona um ciclo em dois novos pela remoção de uma aresta que aparece duas vezes no ciclo (Figura 2.13(a)). Casos especiais que resultam num ciclo vazio estão ilustrados nas Figuras 2.13(b) e 2.13(c). A função inversa é obtida com o operador **MEKR** (*Make Edge, Kill Ring*).



(a): remoção de aresta e partição do ciclo (criação de um anel).



(b): remoção de aresta e criação de um ciclo vazio.



(c): remoção de aresta resultando em vértices solitários.

Figura 2.13: Operador KEMR.

2.4.3.2 Operadores para Manipulações Globais

O operador **KFMRH** (*Kill Face, Make Ring, Hole*) é um operador global (arranjo local de vértices e arestas de uma face não são alterados) e sua atuação está ilustrada na Figura 2.14. A face F_2 é removida da face F_1 e um ciclo interno, composto pelas "semi-arestas" 5_d, 8_d, 7_d e 6_d (seqüência horária), passa a representar um furo em F_1. O operador **MFKRH** reverte seu efeito.

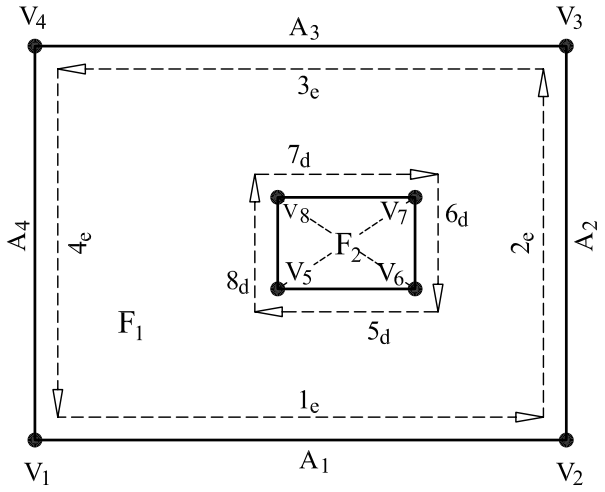


Figura 2.14: Operador KFMRH.

2.5 Padrões de Projeto de Software

No início da década de 90, Gamma, Helm, Jonhson e Vlissides criaram padrões de projeto para o desenvolvimento de software com base em boas soluções de código. Os padrões de projeto determinam nomes, motivações e expõem soluções voltadas para problemas recorrentes na arquitetura e desenvolvimento de sistemas orientados a objeto (Souza e Lima, 2008).

2.5.1 Padrão Model-View-Controller (MVC)

O uso do padrão de projeto *Model-View-Controller (MVC)* permite separar o modelo de sua representação gráfica. A utilização desta metáfora de programação permite que o controle da interação com o usuário e a visualização das respostas sejam implementados independentemente do modelo adotado, minimizando as tarefas de manutenção e expansão da aplicação. A implementação com o padrão *MVC* permite o aperfeiçoamento gradual da aplicação através de mudança de plataforma, criação de diversas vistas sincronizadas com o modelo, substituição ou atualização das diversas vistas e disponibilização on-line do sistema (Brugiolo, 2004).

Existe um ciclo de vida para cada uma das atividades executadas pelo programa. Este ciclo de vida permite que o usuário faça alterações no modelo e visualize o resultado a cada alteração, até que consiga o resultado desejado. O referido ciclo compõe-se de: especificação do usuário, atualização do modelo e visualização.

O padrão de projeto *MVC* pode ser usado para a implementação do ciclo de vida de cada atividade. Nele a aplicação é dividida nos componentes *modelo*, *vista* e *controlador* (Figura 2.15). O *modelo* contém o núcleo dos dados e da funcionalidade do sistema, sendo independente das saídas e entradas de dados. A *vista* apresenta para o usuário as informações armazenadas no *modelo*. O *controlador* é associado a um componente *vista*, ficando responsável pela percepção das entradas do usuário e tradução das mesmas em requisições de serviços para o *modelo* e para a *vista* (Buschmann et al., 1995).

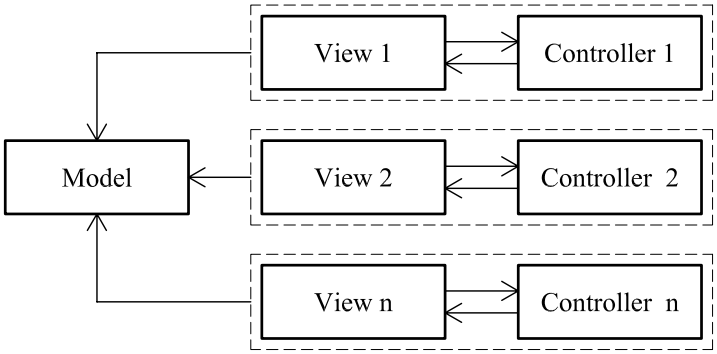


Figura 2.15: Exemplo do padrão *MVC*.

2.5.2 Padrão Command

O uso do padrão *Command* (Gamma et al., 1995) (Figura 2.16) proporciona grande flexibilidade no projeto da interface. Ele permite o encapsulamento de rotinas de execução em objetos, a associação destes objetos aos elementos de interface gráfica com o usuário (botões e menus) e aos dispositivos de entrada (teclado e mouse). O encapsulamento das rotinas de execução possibilita também que a realização de alterações nas mesmas não provoque modificações nas classes existentes (Brugiolo,

2004).

Conforme mostra a Figura 2.16, esse padrão baseia-se em uma classe abstrata chamada *Command*, a qual declara uma interface para execução de operações. Na sua forma mais simples, esta interface inclui o método abstrato *execute()*. As subclasses concretas de *Command* especificam um par receptor-ação através do armazenamento do receptor como uma variável de instância e pela implementação do método *execute()* para invocar a solicitação. A operação *execute()* pode armazenar estados de maneira que o comando possa reverter seus efeitos. Para isso, a interface *Command* deve acrescentar a operação *undo()*. Os comandos executados devem ser armazenados em uma lista histórica (Gamma et al., 1995) que possa ser percorrida para trás (desfazer) e para frente (refazer: aciona *execute()* novamente). O objeto *Client* (exemplo: uma interface gráfica) cria um objeto *ConcreteCommand* (exemplos: *NewModelCommand*, *AddLineCommand*, *RotateCommand*) e o associa a um objeto *Invoker* (botão e/ou item de menu) que é o responsável por disparar a operação *execute()*. Os objetos *Client* e *ConcreteCommand* são associados a um objeto *Receiver* (exemplos: uma área de desenho, um controlador) que tem o conhecimento necessário para poder executar a solicitação.

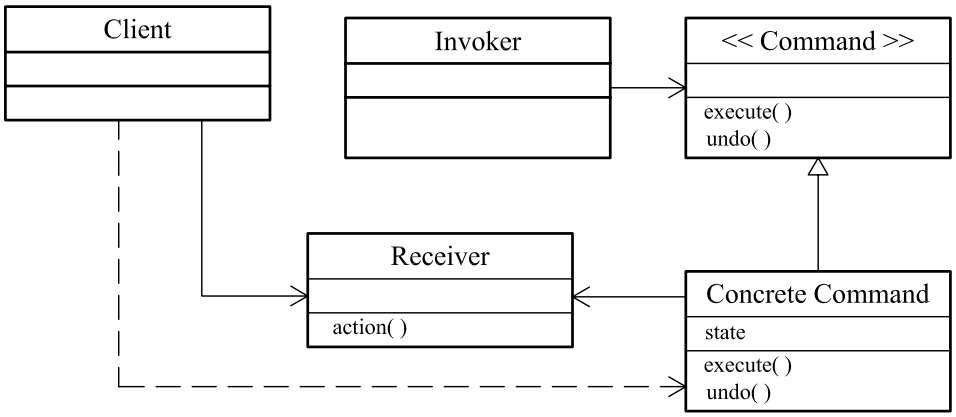


Figura 2.16: Estrutura do padrão *Command*.

2.5.3 Padrão Observer

A utilização da interface *Observer* e da classe *Observable* (Figura 2.17), disponíveis em JAVA, proporciona um mecanismo de propagação de mudanças que garante a consistência e a comunicação entre os pares de componentes controlador e vista (observadores) com o componente modelo (observado). No momento em que um par controlador/vista é criado, o mecanismo de propagação de mudanças efetua seu registro, ligando-o ao modelo do qual ele é dependente.

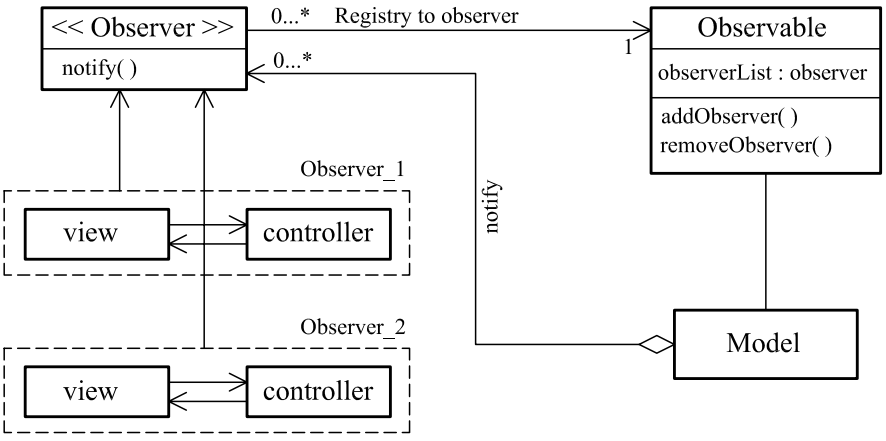


Figura 2.17: Estrutura do padrão *Observer*.

A cada mudança de estado do modelo, o componente *vista* é atualizado. Cada *vista* é associada a um único *controlador* e fornece a este a funcionalidade necessária para manipular a exibição de dados.

2.6 Geração de Malhas Bidimensionais

2.6.1 Mapeamentos Transfinitos

Conforme discutido em Fonseca (1989), os métodos de geração de malhas em domínios bidimensionais podem ser classificados em diretos ou algébricos e indiretos ou de equações diferenciais.

Os métodos diretos ou algébricos são assim chamados porque geram uma malha

sobre o domínio, baseados em algum algoritmo algébrico definido. Podem ser subdivididos em geração de malhas em domínios elementares, geração de malhas por transformação de coordenadas, mapeamentos conformes, mapeamentos isoparamétricos, mapeamentos transfinitos e decomposição de domínio.

Dentre os citados acima, o mapeamento transfinito estabelece sistemas de coordenadas curvilíneas definidos pelo contorno de domínios arbitrários (Figura 2.18). Este método descreve uma superfície aproximada que coincide com a superfície real ou idealizada em um número não enumerável de pontos, propriedade que deu origem ao nome mapeamento transfinito. É capaz de modelar o contorno de superfícies sem a introdução de nenhum erro na geometria do mesmo (Brugiolo, 2004).

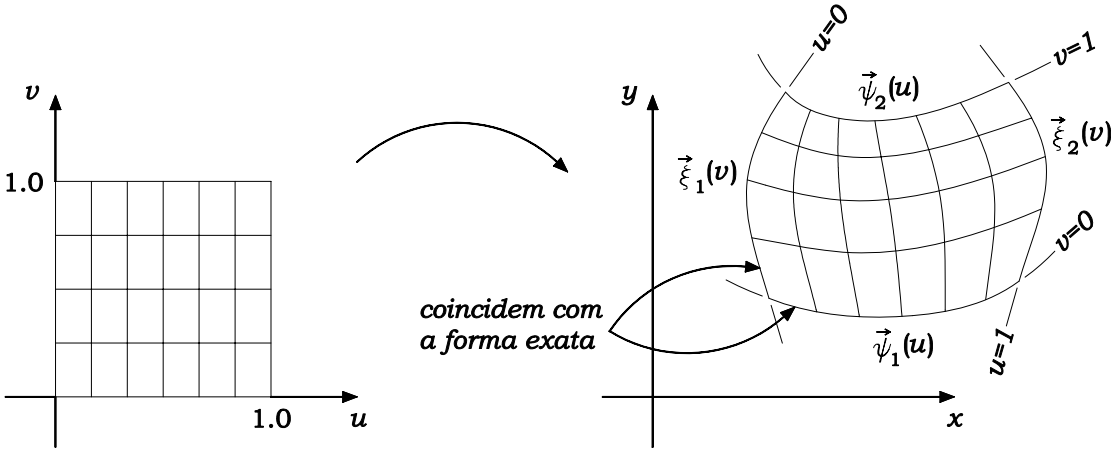


Figura 2.18: Mapeamento Transfinito.

Para descrever este mapeamento é necessário introduzir o conceito de projetor. Um projetor (P) é qualquer operador linear idempotente que mapeia uma superfície real (F) em uma superfície aproximada ($P[F]$), sujeito a certas restrições de interpolação.

O projetor "lofting" é um dos projetores mais simples (Figura 2.19). Ele efetua uma interpolação linear entre duas curvas do contorno, $\overrightarrow{\psi_1}(u)$ e $\overrightarrow{\psi_2}(u)$, de uma região F.

$$P[F] \equiv P(u,v) = (1-v)\overrightarrow{\psi_1}(u) + (v)\overrightarrow{\psi_2}(u) \tag{2.2}$$

para $0 \leqslant u \leqslant 1$ e $0 \leqslant v \leqslant 1$, onde u é uma coordenada paramétrica normalizada ao

longo de $\overrightarrow{\psi_1}$ e $\overrightarrow{\psi_2}$, e v é uma coordenada normalizada que vale zero ao longo de $\overrightarrow{\psi_1}$ e um ao longo de $\overrightarrow{\psi_2}$.

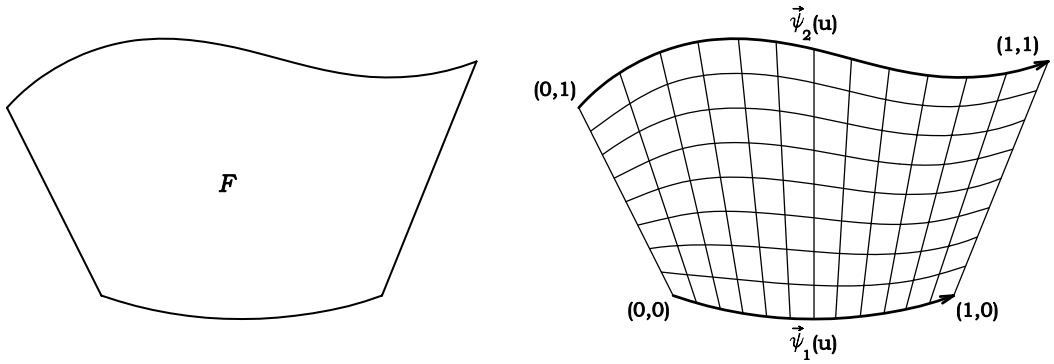


Figura 2.19: Representação do projetor "lofting".

Utilizando-se dois projetores "lofting", um interpolando na direção u e outro na direção v, pode-se fazer combinações para obter projetores mais complexos que irão coincidir com o contorno da região F em todos os pontos. Partindo-se de dois projetores "lofting":

$$P_1[F] \equiv P_1(u,v) = (1-v)\overrightarrow{\psi_1}(u) + (v)\overrightarrow{\psi_2}(u) \tag{2.3}$$
$$P_2[F] \equiv P_2(u,v) = (1-u)\overrightarrow{\xi_1}(v) + (u)\overrightarrow{\xi_2}(v),$$

pode-se definir o projetor soma booleana $(P_1 \oplus P_2)[F]$, de forma que F é mapeada de maneira precisa nos contornos (Figura 2.20):

$$(P1 \oplus P2)[F] \equiv P_1[F] + P_2[F] - P_1P_2[F] = (1-v)\overrightarrow{\psi_1}(u) + \tag{2.4}$$
$$+ v\overrightarrow{\psi_2}(u) + (1-u)\overrightarrow{\xi_1}(v) + u\overrightarrow{\xi_2}(v) - (1-u)(1-v)F(0,0) +$$
$$- (1-u)vF(0,1) - uvF(1,1) - u(1-v)F(1,0)$$

Os projetores P_1 e P_2 são mostrados nas Figuras 2.20(b) e 2.20(c), respectivamente. O projetor produto $P_1P_2[F]$, mostrado na Figura 2.20(d), coincide exatamente com F nos quatro vértices com aproximações lineares nos lados. O projetor $(P_1 \oplus P_2)[F]$ representa um sistema de coordenadas curvilíneas criado pelo mapeamento de um quadrado unitário em F. Este projetor pode ser chamado de

interpolador Lagrangeano bilinear transfinito de F (Figura. 2.20(e)).

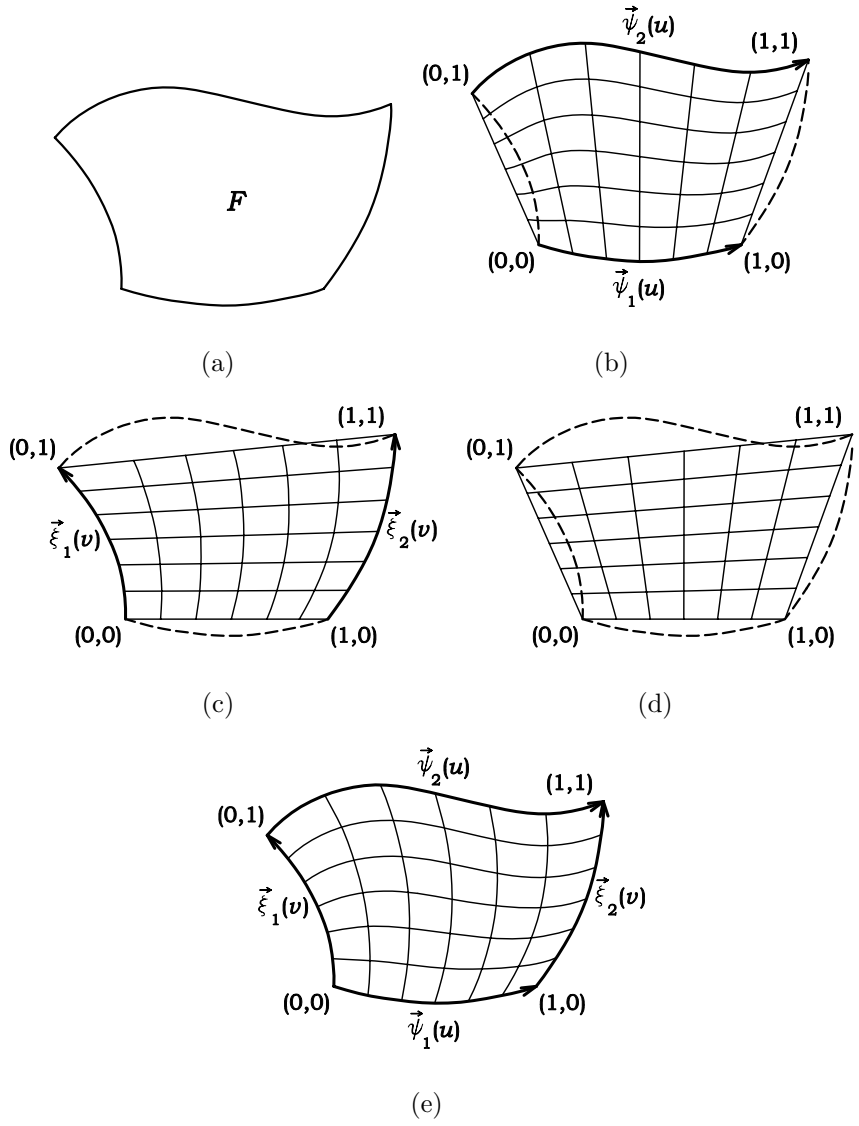


Figura 2.20: Representação de projetores em regiões quadrilaterais.

O mapeamento transfinito bilinear é mais adequado em regiões com contornos curvos nos quatro lados. As linhas interiores da malha possibilitam uma transição suave entre as curvas do contorno.

Através do uso de interpoladores trilineares em um sistema de coordenadas triangulares, obtém-se um projetor capaz de mapear um triângulo unitário em uma região definida por três curvas no contorno.

Para uma região triangular T (Figura 2.21) delimitada por três curvas $\overrightarrow{\psi}(u)$,

$\overrightarrow{\xi}(v)$ e $\overrightarrow{\eta}(w)$, sendo u, v e w o sistema de coordenadas de área, estabelecido no interior da região, define-se o seguinte conjunto de projetores lineares:

$$N_1 \equiv N_1(u,v,w) = \left(\frac{u}{1-v}\right)\overrightarrow{\xi}(v) + \left(\frac{w}{1-v}\right)\overrightarrow{\eta}(1-v) \tag{2.5}$$
$$N_2 \equiv N_2(u,v,w) = \left(\frac{v}{1-w}\right)\overrightarrow{\eta}(w) + \left(\frac{u}{1-w}\right)\overrightarrow{\psi}(1-w)$$
$$N_3 \equiv N_3(u,v,w) = \left(\frac{w}{1-u}\right)\overrightarrow{\psi}(u) + \left(\frac{v}{1-u}\right)\overrightarrow{\xi}(1-u)$$

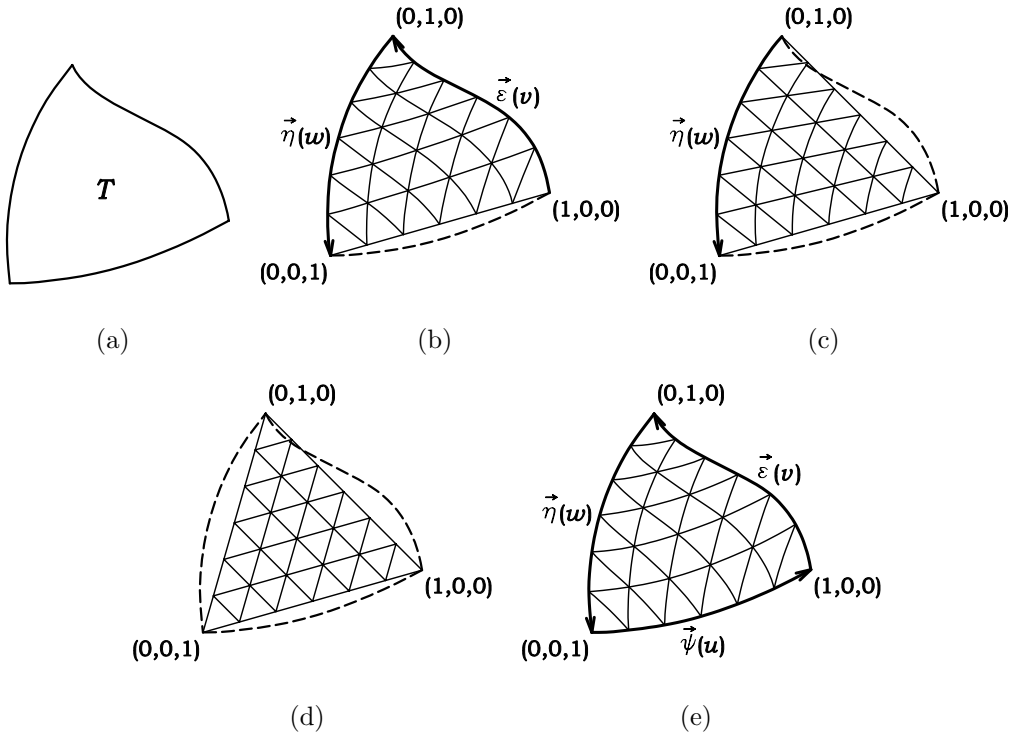


Figura 2.21: Representação de projetores em regiões triangulares.

Estes projetores são interpolações Lagrangeanas lineares entre três pares de curvas de contorno. O projetor N_1 é ilustrado na Figura 2.21(b). O projetor N_1N_2 é ilustrado na Figura 2.21(c). Deve-se observar que $L \equiv N_iN_jN_k$, $i \neq j \neq k$, projeta em um plano passando pelos três vértices de T (Figura 2.21(d)). Pode-se obter ainda seis projetores soma quase-Booleana:

$$N_i \oplus N_j \equiv N_i + N_j - N_iN_j \tag{2.6}$$

com $i \neq j$; $i, j = 1, 2, 3$.

Cada um interpola T de forma exata no contorno. Um projetor trilinear (Figura 2.21(e)) pode ser então definido:

$$S \equiv \frac{1}{2}\left[(N_i \oplus N_j) + (N_i \oplus N_k)\right] = \frac{1}{2}\left[N_1 + N_2 + N_3 - L\right] = \tag{2.7}$$

$$S(u,v,w) = \frac{1}{2}\left[\left(\frac{u}{1-v}\right)\xi(v) + \left(\frac{v}{1-u}\right)\xi(1-u) + \left(\frac{v}{1-w}\right)\eta(w) + \right.$$

$$\left.\left(\frac{w}{1-v}\right)\eta(1-v) + \left(\frac{w}{1-u}\right)\psi(u) + \left(\frac{u}{1-w}\right)\psi(1-w) - u\,\xi(0) - v\,\eta(0) - w\,\psi(0)\right]$$

com $i \neq j \neq k \neq i$.

Os projetores "lofting" e bilinear criam uma partição natural da região, composta de elementos quadrilaterais, enquanto o projetor trilinear cria elementos triangulares.

Inúmeras vantagens tornam o mapeamento transfinito muito adequado para servir a um sistema interativo de geração de malhas de elementos finitos (Brugiolo, 2004):

1. Pontos no interior da malha são gerados por uma expressão fechada, extremamente simples;

2. A quantidade de cálculos requerida para uma malha de pontos não aumenta com a complexidade das curvas do contorno, caracterizando a eficiência computacional;

3. A curvatura das curvas do contorno e o espaçamento dos nós no contorno são refletidos de forma precisa e previsível na malha gerada;

4. As curvas do contorno não necessitam ser funções simples;

5. Ângulos agudos e pontos de inflexão não representam nenhum problema específico;

6. Nenhum erro no ajuste de curvas é introduzido além dos erros de discretização inerentes ao método dos elementos finitos;

7. O uso de geradores automáticos de curvas de contorno reduz os dados de entrada;

8. Em geral, menos regiões e curvas de contorno são requeridas para mapeamentos transfinitos do que para outros métodos;

9. Uma única rotina para cada mapeamento manipula todo tipo de curva de contorno.

A maior desvantagem deste método é a restrição imposta na topologia da malha:

1. Nas regiões "lofting" e bilineares o número de divisões deve ser o mesmo em curvas opostas;

2. Nas regiões trilineares o número de divisões deve ser o mesmo em todas as curvas.

Regiões especiais de transição são necessárias em alguns casos para superar esta dificuldade.

2.7 Transformações Geométricas e Projeções

2.7.1 Sistemas de Coordenadas

Pode-se utilizar diferentes sistemas de coordenadas para descrever os objetos modelados. O sistema de coordenadas serve para dar uma referência em termos de medidas do tamanho e posição dos objetos dentro da área de trabalho. Como exemplo, podem ser citados: o sistema de coordenadas polar (r, θ), o sistema de coordenadas esférico (r, Ψ, θ) e o sistema de coordenadas cilíndrico (r, θ, z).

Um determinado sistema de coordenadas é denominado de *Sistema de Referência* se for um sistema de coordenadas cartesianas para alguma finalidade específica. Para definir um sistema de coordenadas de referência, deve-se especificar dois aspectos principais: a unidade de referência básica e os limites extremos dos valores aceitos para descrever os objetos. Alguns sistemas, como os apresentados a seguir, recebem uma denominação especial.

2.7.1.1 Sistema de Referência do Universo (SRU)

Neste sistema os objetos são descritos em termos das coordenadas do mundo (mínimas e máximas), utilizadas pelo usuário em determinada aplicação.

2.7.1.2 Sistema de Referência do Objeto (SRO)

Neste sistema cada objeto é um miniuniverso individual, ou seja, tem suas particularidades descritas em função de seu sistema, muitas vezes coincidindo o centro do sistema de coordenadas com o seu centro de gravidade.

2.7.1.3 Sistema de Referência do Dispositivo (SRD)

Este sistema utiliza coordenadas que podem ser fornecidas diretamente para um dado dispositivo de saída específico (vídeo, scanner). Assim, nos hardwares, o

sistema de coordenadas depende geralmente da resolução possível e da configuração definida pelo usuário (640x800, 800x600, etc).

2.7.1.4 Sistema de Referência Normalizado (SRN)

Este sistema trabalha com as coordenadas normalizadas, isto é com valores entre 0 e 1, onde $0 \leqslant x \leqslant 1$ e $0 \leqslant y \leqslant 1$, sendo x e y as coordenadas horizontais e verticais possíveis. O SRN serve como um sistema intermediário entre o SRU e o SRD. Sua principal aplicação é tornar a geração das imagens independente do dispositivo, pois as coordenadas do universo são convertidas para um sistema de coordenadas padrão normalizado.

2.7.1.5 Tranformações entre Sistemas de Coordenadas

Aplicações gráficas freqüentemente requerem a transformação de descrições de objetos do SRU para o SRD. Também, muitas vezes, um objeto é descrito em um sistema de coordenadas não-cartesiano e precisa ser convertido para esse sistema.

2.7.2 Transformações em Pontos e Objetos

A habilidade de representar um objeto em várias posições no espaço é fundamental para compreender sua forma. A possibilidade de submetê-lo a diversas transformações é importante em diversas aplicações da computação gráfica (Azevedo e Conci, 2003).

2.7.2.1 Transformação de Translação

Transladar significa movimentar todos os pontos de um objeto. A translação é feita adicionando quantidades às coordenadas dos pontos ($P^{'} = P + T$). Numa representação matricial temos:

$$\text{em 2D} \quad \left[\begin{array}{cc} x' & y' \end{array} \right] = \left[\begin{array}{cc} x & y \end{array} \right] + \left[\begin{array}{cc} T_x & T_y \end{array} \right] \tag{2.8}$$

$$\text{em 3D} \quad \begin{bmatrix} x' & y' & z' \end{bmatrix} = \begin{bmatrix} x & y & z \end{bmatrix} + \begin{bmatrix} T_x & T_y & T_z \end{bmatrix} \tag{2.9}$$

onde, T_x, T_y e T_z são as quantidades transladadas nas direções x, y e z, respectivamente.

2.7.2.2 Transformação de Escala

A transformação de escala consiste em multiplicar as coordenadas de todos os pontos de um objeto por fatores de escada. É importante lembrar que, se o objeto não estiver definido em relação à origem do sistema de coordenadas, essa operação também fará com que o mesmo translade. Se os fatores de escala não forem iguais, o objeto também se deforma.

$$\text{em 2D} \quad \begin{bmatrix} x' & y' \end{bmatrix} = \begin{bmatrix} x & y \end{bmatrix} . \begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix} = \begin{bmatrix} xS_x & yS_y \end{bmatrix} \tag{2.10}$$

$$\text{em 3D} \quad \begin{bmatrix} x' & y' & z' \end{bmatrix} = \begin{bmatrix} x & y & z \end{bmatrix} . \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & S_z \end{bmatrix} = \begin{bmatrix} xS_x & yS_y & zS_z \end{bmatrix} \tag{2.11}$$

onde, S_x, S_y e S_z são os fatores de escala nas direções x, y e z, respectivamente.

2.7.2.3 Transformação de Rotação

Rotacionar significa girar. A rotação de um objeto no plano xy de um ângulo θ, se dá pela multiplicação de suas coordenadas por uma matriz de rotação. Essa operação também resulta em uma translação caso o objeto não esteja definido em relação à origem do sistema de coordenadas.

$$\begin{bmatrix} x' & y' \end{bmatrix} = \begin{bmatrix} x & y \end{bmatrix} . \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \tag{2.12}$$

A rotação de objetos tridimensionais permite sua visualização de diferentes posições e ângulos. A rotação de um objeto 3D é mais simples de ser realizada individualmente sobre cada um dos eixos usando os denominados ângulos de Euler. Esses

ângulos definem a rotação (β, δ, α) de um plano $(yz, xz$ ou $xy)$ pelo giro em torno de um vetor normal a esse plano.

Os sistemas de coordenadas com três eixos ortogonais podem ser descritos por diferentes posições dos seus eixos. A direção positiva será a que obedecer a denominada *"regra da mão direita"* de ordenação dos eixos.

Em 3D tem-se três possíveis matrizes de rotação, uma para cada eixo. Assim, um giro de β graus em torno do eixo x muda as coordenadas de um ponto por:

$$\begin{bmatrix} x' & y' & z' \end{bmatrix} = \begin{bmatrix} x & y & z \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\beta) & \sin(\beta) \\ 0 & -\sin(\beta) & \cos(\beta) \end{bmatrix} \tag{2.13}$$

Um giro de δ graus em torno do eixo y muda as coordenadas de um ponto por:

$$\begin{bmatrix} x' & y' & z' \end{bmatrix} = \begin{bmatrix} x & y & z \end{bmatrix} \cdot \begin{bmatrix} \cos(\delta) & 0 & -\sin(\delta) \\ 0 & 1 & 0 \\ \sin(\delta) & 0 & \cos(\delta) \end{bmatrix} \tag{2.14}$$

Um giro de α graus em torno do eixo z muda as coordenadas de um ponto por:

$$\begin{bmatrix} x' & y' & z' \end{bmatrix} = \begin{bmatrix} x & y & z \end{bmatrix} \cdot \begin{bmatrix} \cos(\alpha) & \sin(\alpha) & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.15}$$

Uma observação interessante é que essas matrizes são todas ortogonais e normalizadas (ortonormais). Outro ponto importante é que ao se definir a rotação tridimensional por combinação de rotações, tem-se uma posição que é dependente da ordem de definição das rotações. Ou seja, os ângulos de Euler não definem as rotações de objetos de maneira comutativa.

O processo de combinar duas ou mais matrizes é chamado de concatenação e é executado multiplicando as matrizes antes de aplicá-las aos pontos.

2.7.2.4 Transformação de Reflexão

A transformação de reflexão em torno de um eixo (ou *flip*), aplicada a um objeto, produz um novo objeto espelhado em relação a esse eixo. No caso de uma reflexão

2D, o espelho pode ser considerado sobre o eixo x ou y. No caso de objetos 3D, a reflexão pode ser em torno de qualquer um dos três planos. Por exemplo, para uma reflexão em torno do plano xz, as coordenadas são transformadas da seguinte forma:

$$\begin{bmatrix} x' & y' & z' \end{bmatrix} = \begin{bmatrix} x & y & z \end{bmatrix} . \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.16}$$

Pode-se também definir uma reflexão em torno de dois eixos, por exemplo x e y. Neste caso, a reflexão é feita em torno da origem do sistema de coordenadas. Cada ponto é transformado da seguinte maneira:

$$\begin{bmatrix} x' & y' & z' \end{bmatrix} = \begin{bmatrix} x & y & z \end{bmatrix} . \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.17}$$

2.7.2.5 Transformação de Cisalhamento

Cisalhamento (*Shearing ou Skew*) é uma transformação que distorce o formato de um objeto. Nela aplica-se um deslocamento aos valores das coordenadas x, y ou z do objeto, proporcional ao valor das outras coordenadas de cada ponto transformado. Uma distoção na direção x, proporcional a coordenada y, pode ser produzida da seguinte maneira:

$$\begin{bmatrix} x & y & z \end{bmatrix} . \begin{bmatrix} 1 & 0 & 0 \\ S & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} x + S_y & y & z \end{bmatrix} \tag{2.18}$$

Onde S é um valor fixo qualquer. Qualquer número real pode ser usado como parâmetro, assim como é possível fazer a distorção em qualquer direção. Por exemplo:

$$\begin{bmatrix} x & y & z \end{bmatrix} . \begin{bmatrix} 1 & a & 0 \\ 0 & 1 & 0 \\ 0 & b & 1 \end{bmatrix} = \begin{bmatrix} x & (xa + y + zb) & z \end{bmatrix} \tag{2.19}$$

distorcerá um objeto ao longo da direção y de forma proporcional à coordenada x, de um valor a e proporcional à coordenada z de um valor b.

2.7.3 Coordenadas Homogêneas

Num sistema de coordenadas homogêneas, um ponto no espaço (x, y, z) é representado por quatro valores (x', y', z', M). A transformação do sistema homogêneo para o cartesiano se dá pela seguinte relação: $(x, y, z) = (x'/M, y'/M, z'/M)$.

Com as coordenadas homogêneas, as operações de translação podem ser escritas como matrizes de transformação. Isso faz com que as transformações geométricas fiquem uniformizadas pelo cálculo matricial e podem ser combinadas por concatenação (multiplicação) de matrizes.

A aplicação da translação em 3D por multiplicação de matrizes pode, nesse sistema de coordenadas, ser escrita na forma:

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} . \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{bmatrix} \tag{2.20}$$

Já uma transformação de escala é dada por:

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} . \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.21}$$

2.7.4 Projeções

As projeções geométricas permitem que os objetos tridimensionais sejam representados em um plano bidimensional. As coordenadas 3D são convertidas em coordenadas 2D de maneira que correspondam a uma visão do objeto de uma posição específica. Para essa representação devem ser considerados os seguintes elementos básicos (Figura 2.22): plano de projeção (superfície onde será projetado o objeto), raios de projeção (retas que passam pelos pontos do objeto e pelo centro de projeção) e o centro de projeção (ponto fixo de onde partem os raios projetantes).

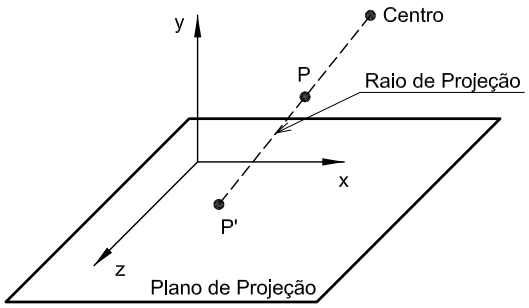


Figura 2.22: Projeção Geométrica.

As projeções geométricas são classificadas conforme o organograma da Figura 2.23 e dependem das relações entre o centro, o plano e os raios de projeção.

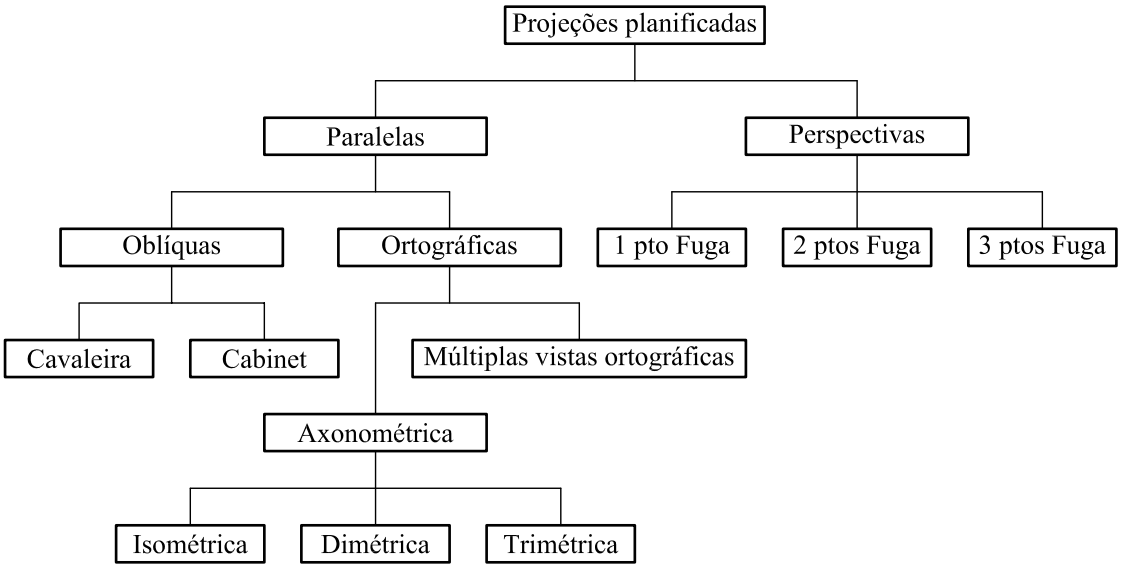


Figura 2.23: Classificação das Projeções Geométricas.

Nas projeções paralelas, o centro de projeção é localizado no infinito e todas as linhas de projeção são paralelas entre si. São tradicionalmente usadas em engenharia e desenhos técnicos.

2.7.4.1 Projeções Paralelas Oblíquas

As projeções oblíquas são produzidas por um conjunto de linhas de projeção inclinadas de um ângulo qualquer em relação ao plano de projeção.

Na projeção *oblíqua cavaleira* (ou cavalier) essas linhas fazem um ângulo de 45 graus com o plano de projeção e com isso os pontos projetados preservam sua medida original nas direções não paralelas a esse plano.

Na projeção *oblíqua cabinet* as linhas de projeção fazem um ângulo de aproximadamente 63,4 graus (tangente=2) de modo a reproduzir os objetos com uma dimensão de metade do tamanho original. Somente a face do objeto paralela ao plano de projeção permanece com o seu tamanho sem distorção (ou com a verdadeira grandeza).

2.7.4.2 Projeções Paralelas Ortográficas

Nas projeções paralelas ortográficas, as linhas de projeção são paralelas entre si e perpendiculares ao plano de projeção. A principal característica dessas projeções é a direção que o plano de projeção faz com as faces principais do objeto a ser projetado.

Nas *múltiplas vistas ortográficas*, o plano de projeção aparece paralelo (a uma distância T) aos planos principais que representam as faces do objeto. Assim ele é mostrado de topo (planta baixa), de frente e de lado (elevações). Para projeções paralelas as matrizes de transformação são:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & T_z & 1 \end{bmatrix} \qquad \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & T_y & 0 & 1 \end{bmatrix} \tag{2.22}$$

em xy em yz em xz

As projeções *axonométricas* consideram as medidas antes e depois das projeções. Dependendo da maneira como essas medidas aparecem no plano de projeção, recebem denominações especiais: isométrica (a mais utilizada), dimétrica ou trimétrica. Na isometria o plano de projeção fica posicionado em relação aos planos do objeto de maneira tal que os três eixos do objeto parecerão ter a mesma mudança nas métricas, como o próprio nome indica (*iso=mesmo, métrica=medida*).

Para obter uma projeção axonométrica usando métodos computacionais, as coordenadas do objeto são transformadas por uma matriz em coordenadas homogêneas:

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} . \begin{bmatrix} \cos(\delta) & \sin(\delta)\sin(\beta) & 0 & 0 \\ 0 & \cos(\beta) & 0 & 0 \\ \sin(\delta) & -\cos(\delta)\sin(\beta) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.23}$$

onde δ é o ângulo de rotação em torno do eixo y e β em torno do eixo x.

Para uma projeção isométrica os valores dos ângulos devem ser: $\delta = 45°$ e $\beta \cong 35.26°$. Nas projeções dimétricas, em vez dos três eixos sofrerem as mesmas mudanças de escala, apenas dois terão a mesma redução. Para um dado ângulo β, o ângulo δ deve ser tal que $\tan\beta = \sin\delta$. Já nas projeções trimétricas, cada eixo sofrerá uma transformação de escala própria.

2.7.4.3 Projeções Perspectivas ou Cônicas

A projeção perspectiva produz uma imagem mais realista do objeto, porém não pode reproduzir suas verdadeiras medidas. As coordenadas dos pontos projetados dependem da distância em relação ao plano de projeção. As matrizes a seguir descrevem as projeções de um ponto de vista sobre um eixo perpendicular ao plano de projeção:

$$\underset{\text{em } x}{\begin{bmatrix} 0 & 0 & 0 & \frac{-1}{f_x} \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}} \qquad \underset{\text{em } y}{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{-1}{f_z} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}} \qquad \underset{\text{em } z}{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & \frac{-1}{f_z} \\ 0 & 0 & 0 & 1 \end{bmatrix}} \tag{2.24}$$

onde f_x, f_y e f_z são distorções perpendiculares aos planos de projeção (Figura 2.24)

Os desenhos em perspectiva são caracterizados pela mudança do comprimento (os objetos são cada vez menores à medida que sua distância ao centro de projeção aumenta) e pelos pontos de fuga (pontos para os quais convergem um conjunto de linhas paralelas).

Pontos de fugas principais são aqueles que aparentam uma interseção entre um conjunto de retas paralelas a um dos eixos principais x, y ou z. As projeções em perspectiva são classificadas pelo número de pontos de fuga principais, ou seja, o número de eixos que o plano de projeção intercepta. Projeções com um e dois pontos de fuga são mais comumente usadas em arquitetura e desenho publicitário. Já aquelas com três pontos de fuga são bem menos usadas, pois adicionam um certo surrealismo a cena (Azevedo e Conci, 2003).

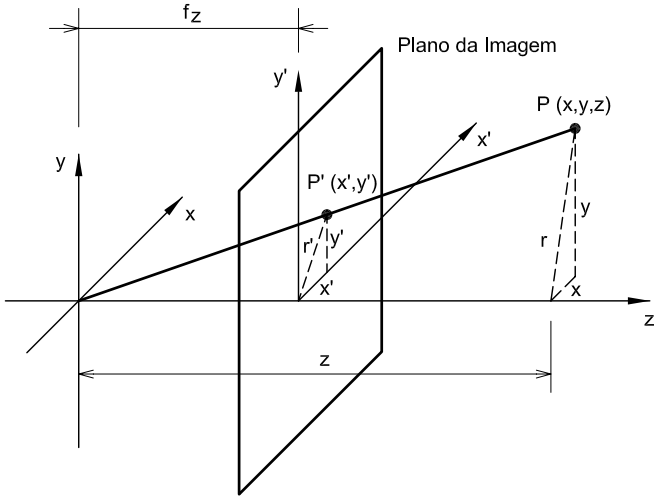


Figura 2.24: Projeção Perspectiva.

2.8 Trabalhos Relacionados

Voltados ao ensino da análise estrutural, existem duas categorias de programas para os estudantes de engenharia:

(i) Programas educacionais e gratuitos. São de fácil utilização e têm como objetivo apresentar ao estudante as técnicas numéricas para análise estrutural. Alguns permitem acesso a diversas etapas intermediárias de cálculo, tornando-se ferramentas importantes para o ensino.

(ii) Programas comerciais com versões educacionais. Apresentam limitações quanto ao número de nós e elementos. São robustos e exigem uma certa experiência e domínio de conceitos para serem utilizados. Objetivam mostrar ao estudante ferramentas reais de trabalho.

Segundo Kaefer (2000), o programa **FTOOL** (disponível em `http://www.tecgraf.puc-rio.br/ftool`) é um sistema gráfico interativo cujo objetivo principal é fornecer ao estudante de engenharia estrutural uma ferramenta para aprender o comportamento de pórticos planos. Pode ser executado em praticamente qualquer plataforma, de micro computadores a estações gráficas de trabalho, bastando recompilá-lo na plataforma desejada e ligá-lo com as bibliotecas gráficas apropriadas (sistema de Interface IUP/LED e o sistema gráfico *Canvas Draw*) desenvolvidas no Tecgraf/PUC-Rio. Algumas das suas características são:

1. Interface gráfica baseada em manipulação direta com menus em cascata e botões;
2. Desenvolvido em linguagem C (código fonte não disponível para avaliação);
3. Não faz separação entre modelo geométrico e modelo de malha;
4. Utiliza a estrutura de dados de "semi-arestas" até o nível *Aresta*.

O programa denominado **PORT 3D** (Ferro, 2001), é um aplicativo para a análise de pórticos espaciais. Desenvolvido em linguagem DELPHI, através do trabalho de um grupo de pesquisa da Faculdade de Engenharia da UNESP. Tem a finalidade educacional, permitindo ao estudante a visualização dos dados da estrutura através de janelas e o acesso às etapas intermediárias de cálculo (`http://www.pp.ufu.br/Cobenge2001/trabalhos/MTE121.pdf`).

Desenvolvidos no Brasil, programas comerciais como o **TQS** (`http://www.tqs.com.br`) e o **Eberick** (`http://www.altoqi.com.br`) são voltados para o projeto de edificações em concreto armado e permitem análise via pórtico espacial.

O CYPECAD (`http://www.cype.com`) também é outro programa comercial para modelagem de estruturas espaciais (concreto e aço). Apesar de possuir alto nível de automação, não permite interferência do usuário nas características do modelo.

A avaliação das aplicações mencionadas anteriormente, serviu de base para o desenvolvimento do novo pré-processador do **INSANE**.

Capítulo 3

PROJETO E IMPLEMENTAÇÃO DO PRÉ-PROCESSADOR

3.1 Requisitos Desejados

Uma vez que a finalidade desta dissertação foi a elaboração de um pré-processador para a criação de modelos tridimensionais, com a possibilidade de combinar elementos, desejou-se os seguintes requisitos para o programa:

1. Pré-Processador com capacidade de combinar elementos de barras, estado plano e placas;
2. Existência de uma interface gráfica e interativa;
3. Independência entre o modelo geométrico e o modelo de malha;
4. Interface com o usuário através de múltiplas janelas, sendo um tipo específico para cada módulo do pré-processador (módulo *Geometria* e módulo *Malha*);
5. Armazenamento da topologia do modelo na estrutura de dados "Half-Edge";
6. Possibilidade de observar os modelos através de várias vistas;
7. Possibilidade de aplicar nas vistas uma transformação geométrica ou uma projeção do modelo;

8. Recursos para seleção de objetos através de seleção individual, por linha, por janela ou com uso de filtros;

9. Comandos para geração de primitivas geométricas, tais como: pontos, linhas, arcos, círculos, curvas cúbicas e curvas quadráticas;

10. Comandos para criação de regiões retangulares, circulares ou poligonais;

11. Geradores automáticos de geometria para: vigas, pórticos planos, pórticos espaciais e grelhas;

12. Comandos para edição da geometria, tais como: apagar, copiar, mover, esticar, quebrar, dividir, rotacionar, espelhar, escalar e inserir furos em regiões;

13. Desenvolvimento de uma classe para cuidar da transição entre o modelo geométrico e o modelo de malha (*Parser GeoToMesh*);

14. Adaptação do gerador de malhas desenvolvido por Brugiolo (2004) para permitir o mapeamento nos planos XY, XZ e YZ e o armazenamento dos dados na estrutura de "Half-Edges";

15. Comandos para permitir a remoção de "nós" e "elementos" após a geração da malha;

16. Diálogo para a escolha do *Modelo de Análise Global*;

17. Diálogos para criação das listas de: materiais, seções transversais, carregamentos e combinações;

18. Diálogos para definição de atributos em nós: restrições, molas, massa, amortecimento, deslocamentos prescritos, ângulo e cargas nodais;

19. Diálogos para definição de atributos em elementos: cargas pontuais, de linha, de área, seção transversal, ordem de integração;

20. Diálogos para definição de atributos exclusivos a elementos de barra: variação de temperatura, carregamento nodal equivalente, liberação nas extremidades, deformações pré-definidas;

21. Existência de um comando para gerar o modelo de elementos finitos após a definição dos atributos no modelo de malha (*Parser MeshToFem*);

22. Comando para exportação do modelo de elementos finitos, em formato XML, antes ou depois do processamento;

23. Diálogo para escolha do tipo de solução, tipo de processamento (Automático ou Interativo) e as formas de persistência (arquivo XML ou arquivo binário).

3.2 Arquitetura do Sistema

A arquitetura das aplicações **INSANE** (Figura 3.1) é baseada em uma combinação dos padrões de projeto de software *Model-View-Controller (MVC)*, *Command* e *Observer*, discutidos no Capítulo 2.

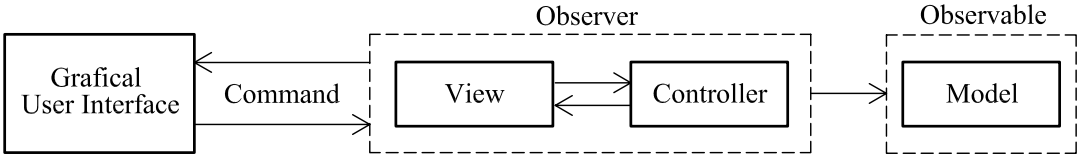


Figura 3.1: Arquitetura do **INSANE**.

A Figura 3.2 apresenta o projeto do pré-processador, uma adaptação à arquitetura das aplicações **INSANE**, mostrando o relacionamento entre os pares *vista-controlador* com o *modelo*.

A interface gráfica possui recursos específicos para a manipulação de cada modelo (Geométrico ou Malha). As *vistas* que observam um *modelo* são atualizadas todas as vezes em que esse *modelo* é modificado através da execução de um comando. Isso

se dá graças ao mecanismo de propagação de mudanças proporcionado pela interface *Observer* e pela classe *Observable*.

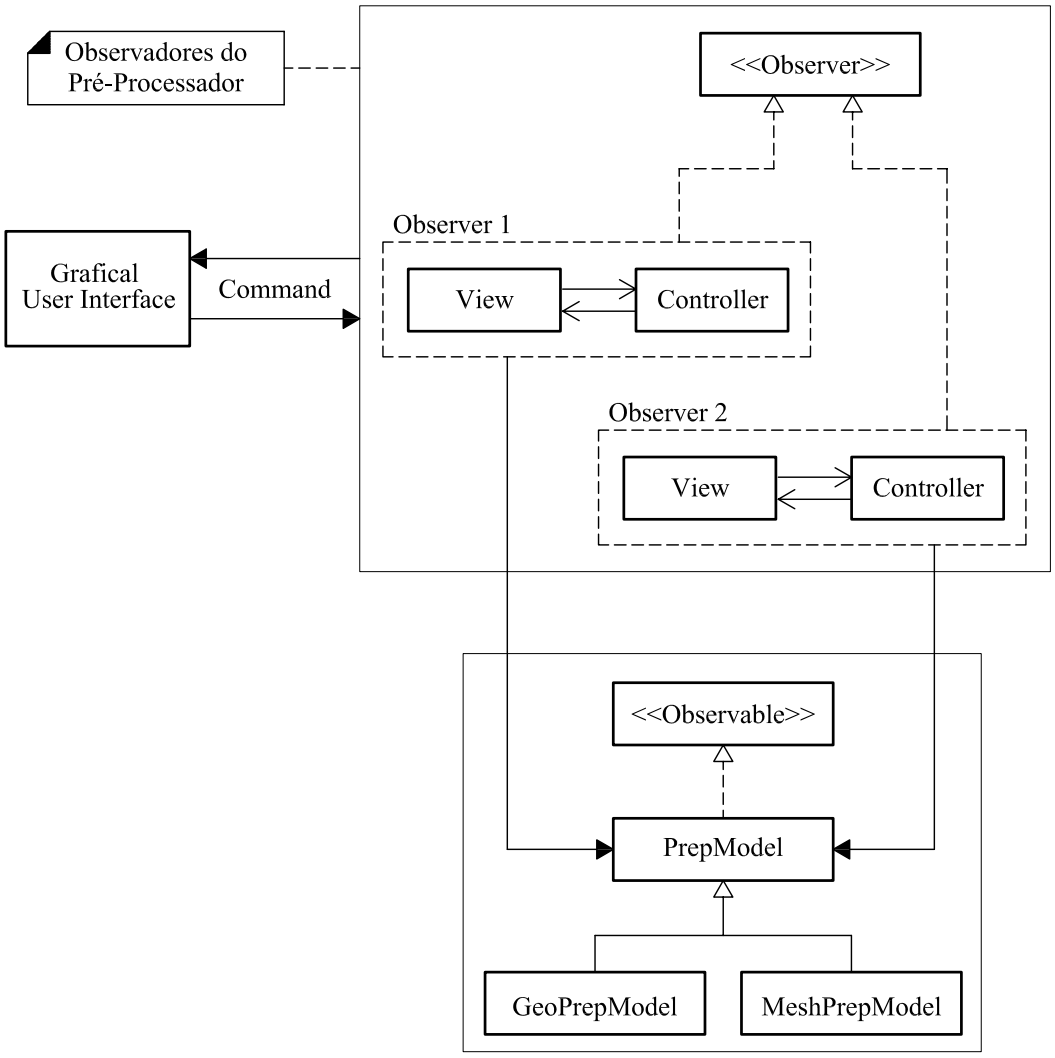


Figura 3.2: Arquitetura do Pré-Processador INSANE.

3.3 Sub-Projetos e Classes Principais

O sistema **INSANE** é dotado de uma estrutura bem segmentada na qual aplicações cliente (módulo aplicação) consomem implementações ofertadas por módulos de serviço. Um relação com todos os módulos está disponível no Apêndice A.

O pré-processador (módulo aplicação) foi dividido em quatro sub-projetos, listados a seguir, e cada um se utiliza de outros sub-projetos (módulos de serviço).

1. **br.ufmg.dees.insane.model.prep** [1] - reúne as classes que são comuns aos modelos do pré-processador (Geométrico e Malha);

2. **br.ufmg.dees.insane.ui.rich.geo** - reúne as classes relacionadas com a interface gráfica do pré-processador geométrico;

3. **br.ufmg.dees.insane.ui.rich.mesh** - reúne as classes relacionadas a interface gráfica do pré-processador de malhas;

4. **br.ufmg.dees.insane.ui.rich.prep** - reúne as classes que são comuns aos sub-projetos *ui.rich.geo* e *ui.rich.mesh*.

Para a apresentação das principais classes que compõem o pré-processador, serão usados ao longo desta seção, diversos diagramas que possibilitem a visualização da estrutura da aplicação. Os diagramas seguem a proposta da UML (Unified Modeling Language), linguagem padronizada para a modelagem de sistemas de software orientados a objetos. A Figura 3.3 apresenta uma legenda dos símbolos utilizados.

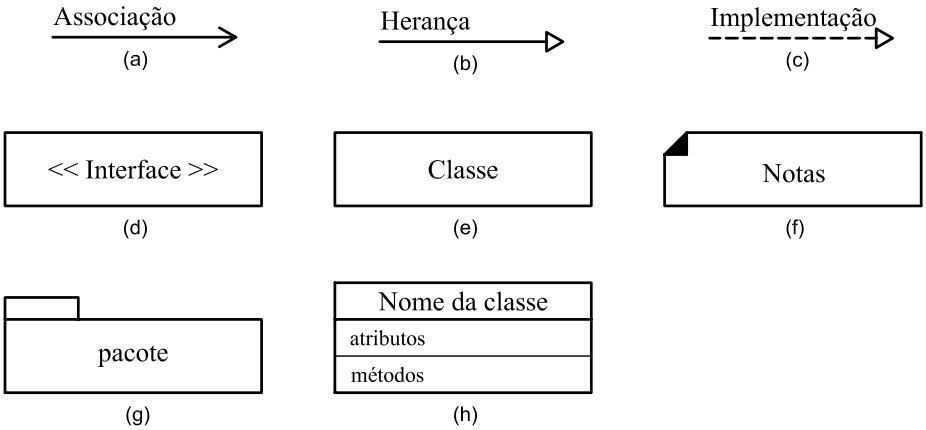


Figura 3.3: Símbolos da linguagem UML.

[1] A nomenclatura adotada segue o seguinte padrão de distribuição de software: br = Brasil; ufmg = Universidade Federal de Minas Gerais; dees = Departamento de Engenharia de Estruturas; insane... = nome do sub-projeto do INSANE.

3.3.1 Classes para o Modelo

As classes que representam o modelo do pré-processador foram organizadas em pacotes, conforme ilustra a Figura 3.4.

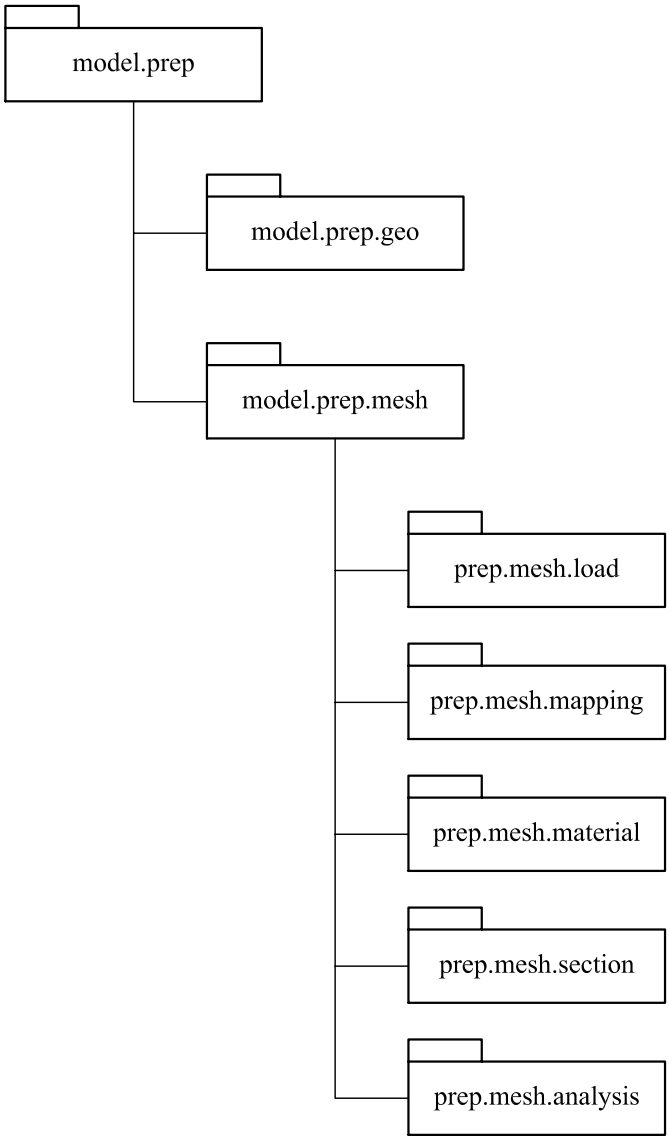


Figura 3.4: Estrutura de pacotes para o modelo do pré-processador.

A Figura 3.5 apresenta a classe *PrepModel* (pacote *model.prep*) e suas classes herdeiras. Ela implementa a interface *Serializable* (pacote *java.io.Serializable*) para que o modelo possa ser persistido e a interface *Observable* (pacote *java.util.Observable*) para que os observadores sejam notificados quando o modelo for atualizado.

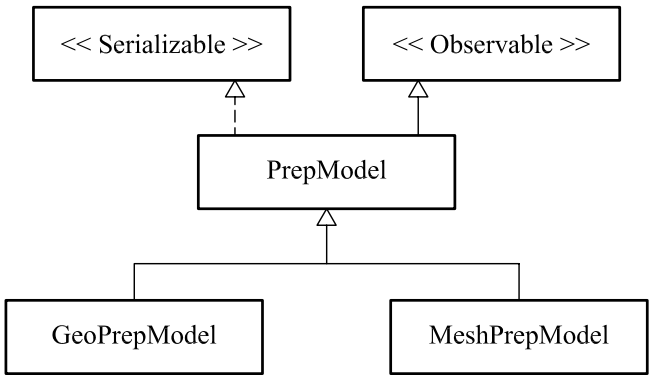


Figura 3.5: Classes do modelo.

Como o modelo do pré-processador foi separado em *GeoPrepModel* (pacote *model.prep.geo*) e *MeshPrepModel* (pacote *model.prep.mesh*), também foi projetado uma interface gráfica (detalhada na próxima seção) com dois módulos: *Geometria* e *Malha*. Cada módulo está associado a um modelo. O primeiro conta com recursos para criação e edição de pontos, linhas, curvas e regiões. O segundo disponibiliza recursos para a geração de malhas e a definição de atributos. Com essa segmentação possibilita-se gerar diferentes malhas a partir de um mesmo modelo geométrico.

A classe *PrepModel* (Figura 3.6) possui um campo da classe *HalfEdgeDataStructure* (projeto *br.ufmg.dees.insane.geometry*), responsável pelo gerenciamento dos dados que compõem a subdivisão planar do modelo.

Para permitir a independência entre os modelos, foram implementadas classes denominadas "parsers" (projeto *br.ufmg.dees.insane.parser*), responsáveis pela tradução dos dados de um modelo para o outro.

Na passagem do módulo *Geometria* para o módulo *Malha*, um "parser" denominado *ParserGeoToMesh* faz uma cópia da estrutura de dados do modelo geométrico para o modelo de malha. Após a geração da malha e a definição de seus atributos (tipos de elemento, seção transversal, material, carregamentos e combinações), outro "parser" denominado *ParserMeshToFem* cuida do preenchimento de um modelo de elementos finitos.

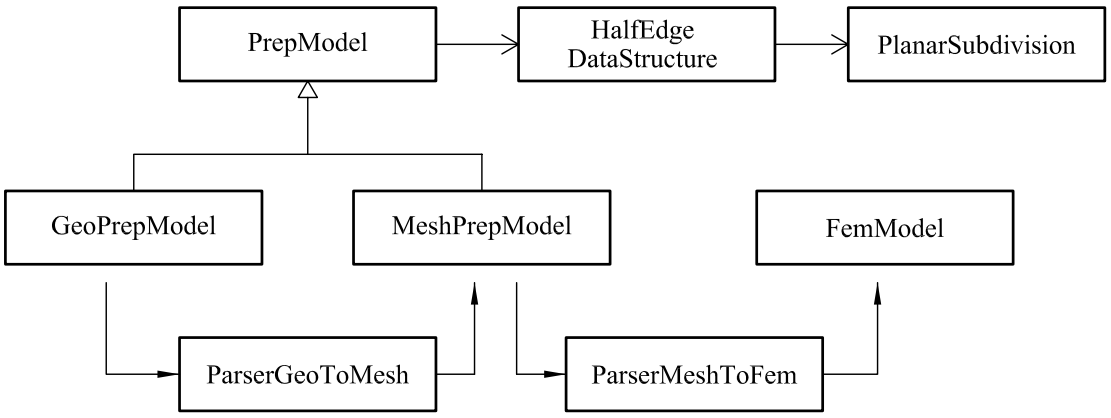


Figura 3.6: Independência entre os modelos.

A Figura 3.7 ilustra a classe *MeshPrepModel* com seus tipos e campos. A criação de classes com nomes semelhantes àqueles do projeto *br.ufmg.dees.insane.model* permitiu a independência entre o modelo de malha e o modelo de elementos finitos (*FemModel*). Os dados relativos à lista de nós e de elementos (e seus atributos) são armazenados, respectivamente, nas listas de Vértices e Faces da Subdivisão Planar.

MeshPrepModel	
Boolean	hasAttributes
String	combinedElemType
String	probDriver
String	globalAnalysisModel
ArrayList<Boundary>	boundaryList
ArrayList<String>	analysisModels
ArrayList<MeshLoadCombination>	loadCombinations
ArrayList<MeshLoading>	loadings
ArrayList<MeshSection>	sections
ArrayList<MeshMaterial>	materials
ArrayList<String>	constitutiveModels
metodos para acesso e modificação dos atributos	

Figura 3.7: Campos da classe *MeshPrepModel*.

A classe *GeoPrepModel* herda todos os campos e atributos de sua classe base.

3.3.2 Classes para a Interface Gráfica

A Figura 3.8 mostra a organização da janela principal do **INSANE**, que representa a interface gráfica com o usuário (GUI - Graphical User Interface).

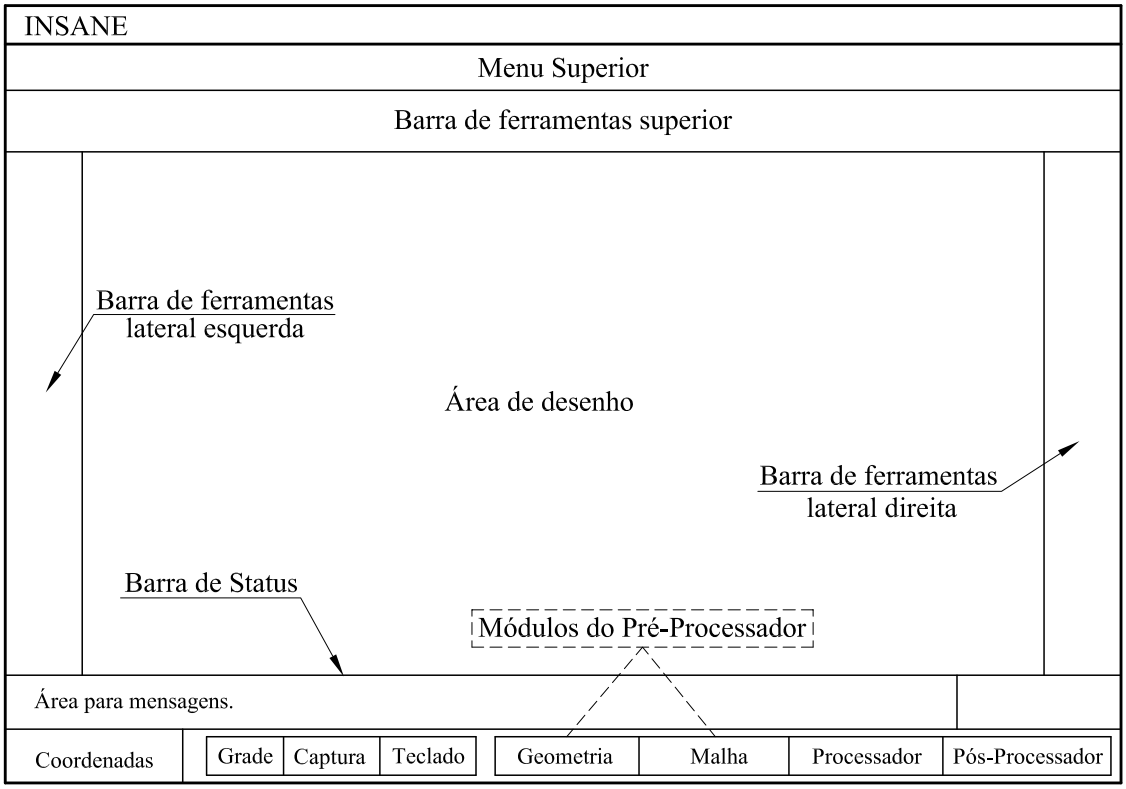


Figura 3.8: Interface gráfica do INSANE.

As Figuras 3.9 e 3.10 apresentam os relacionamentos entre as classes que formam a interface gráfica do **INSANE** e a aplicação dos padrões de projeto de software *MVC*, *Command* e *Observer*.

A classe abstrata *Interface* implementa métodos de *InternalFrameListener* (pacote *javax.swing.event*) para permitir o uso de múltiplas janelas internas (relação um para n). No momento em que uma janela da classe *InternalInterfaceGeo* ou *InternalInterfaceMesh* é ativada, um método chamado *updateOwner()*, definido na classe *InternalInterface* é acionado. Esse método atualiza a interface do pré-processador (GUI) com menus e barras de ferramentas específicos. Na classe de cada janela são declarados vários comandos que são associados a cada item de menu ou botão das

barras, conforme estabelece o padrão *Command*,

A classe *InsaneInternalInterface* complementa o método *updateOwner()* para registrar na barra de status, o módulo corrente (Geometria ou Malha) e o estado de alguns botões (habilitado ou desabilitado).

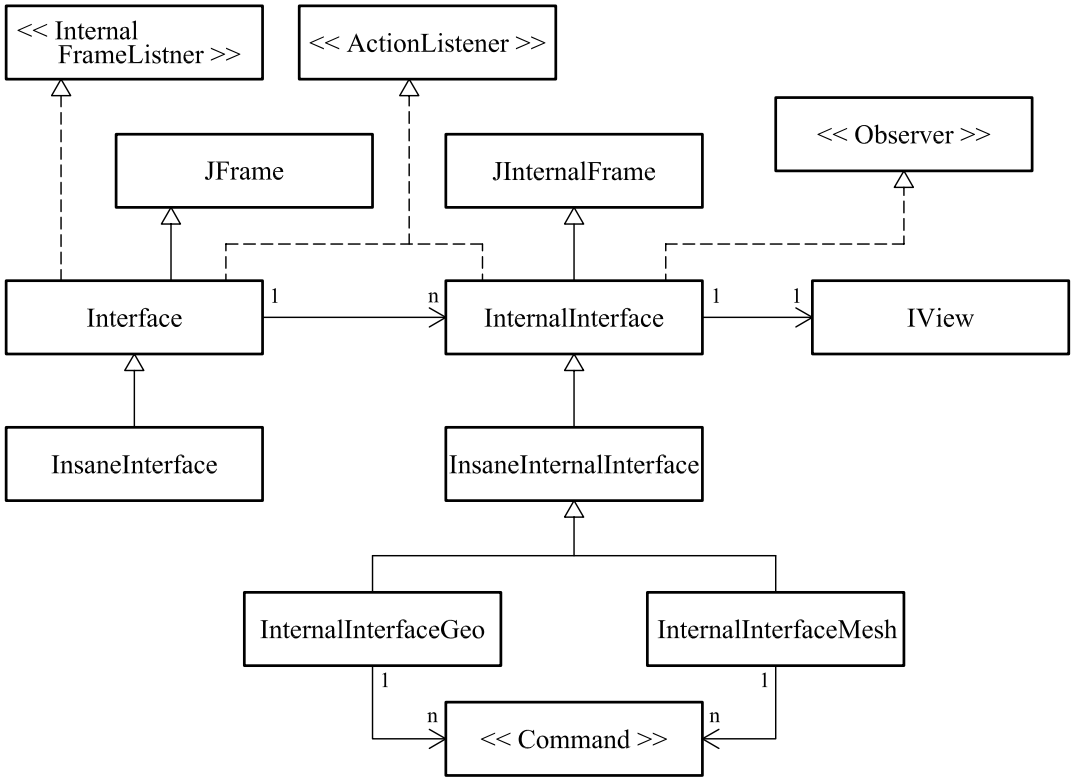


Figura 3.9: Relacionamento entre as classes da interface gráfica.

A *InternalInterface* é herdeira de *JInternalFrame* (pacote *javax.swing.JInternalFrame*) e implementa a interface *ActionListener* (pacote *java.awt.event.ActionListener*) para disparar o método *execute()* de cada comando. Implementa também a interface *Observer* (pacote *java.util.Observer*) para ser notificada quando da atualização do modelo. Ela observa o modelo através de uma vista, representada pela classe *IView* (detalhada na Figura 3.10). A *vista* é formada por uma área de desenho (classe *DrawingArea*), por um estado da vista (classe *ViewState*) e por um controlador (classe *Controller*).

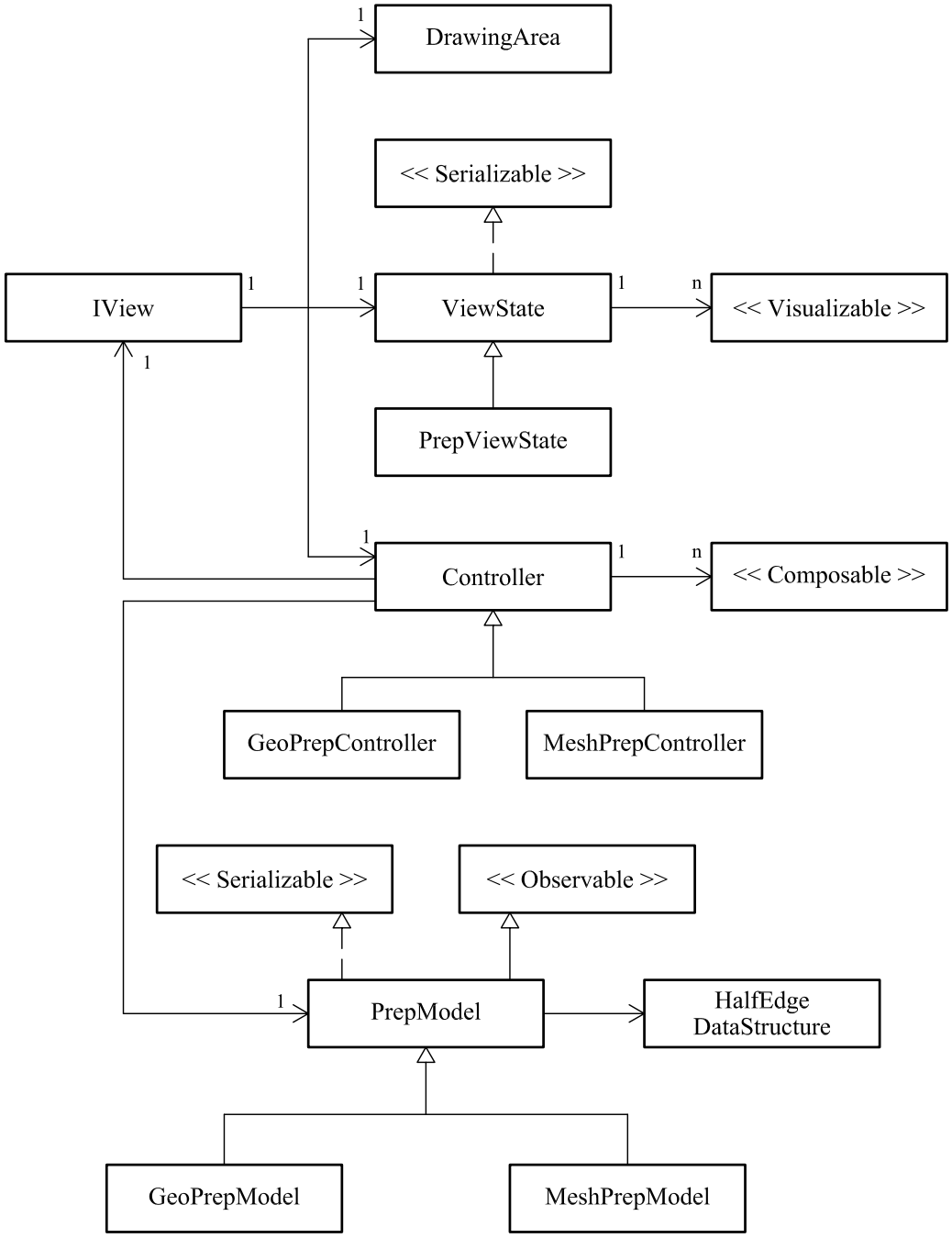


Figura 3.10: A vista e seus componentes.

O estado da vista é o responsável pelo visual dos elementos gráficos e pode ser formado por várias classes que implementam a interface *Visualizable*. O controlador cuida da composição dos desenhos que representam o modelo e pode ser formado por vários "compositores" (classes que implementam a interface *Composable*).

Assim como feito para as janelas, a classe *Controller* também foi segmentada

(Figura 3.10). Quando uma janela da classe *InternalInterfaceGeo* é inserida na interface, um controlador da classe *GeoPrepController* é a ela associado para observar um modelo do tipo *GeoPrepModel* (Figura 3.11).

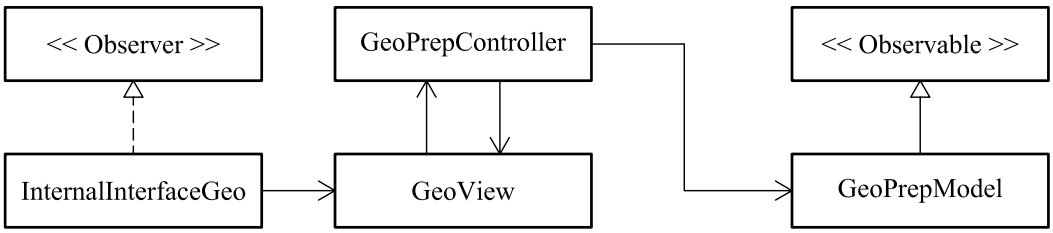


Figura 3.11: Vista-Controlador e Modelo para o módulo *Geometria*.

Da mesma forma, uma *InternalInterfaceMesh* é associada a um *MeshPrepController* para observar um modelo do tipo *MeshPrepModel* (Figura 3.12).

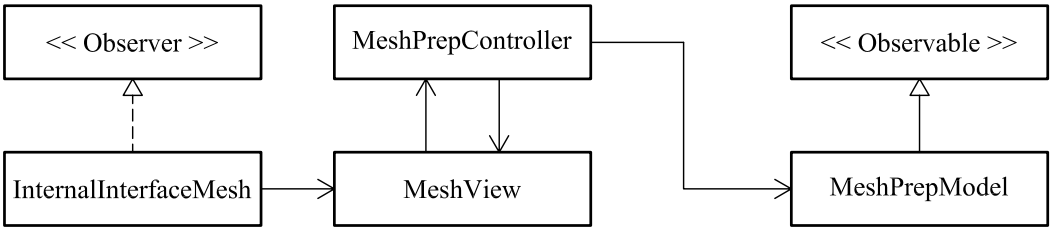


Figura 3.12: Vista-Controlador e Modelo para o módulo *Malha*.

As classes *GeoView* e *MeshView* são herdeiras de *IView* (Figura 3.13).

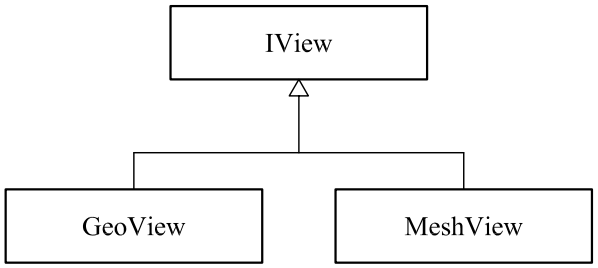


Figura 3.13: Sub-classes de *IView*.

As classes que implementam a interface *Composable* para o controlador geométrico (*GeoPrepController*) e para o controlador de malha (*MeshPrepController*) são

apresentadas nas Figuras 3.14 e 3.15, respectivamente. Cada uma é responsável pela composição de um tipo de desenho (objetos filhos da classe abstrata *Draw*).

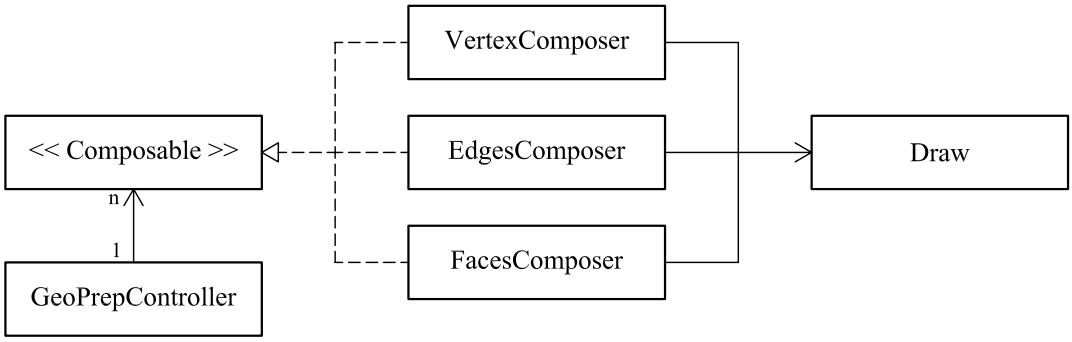


Figura 3.14: Compositores do controlador geométrico.

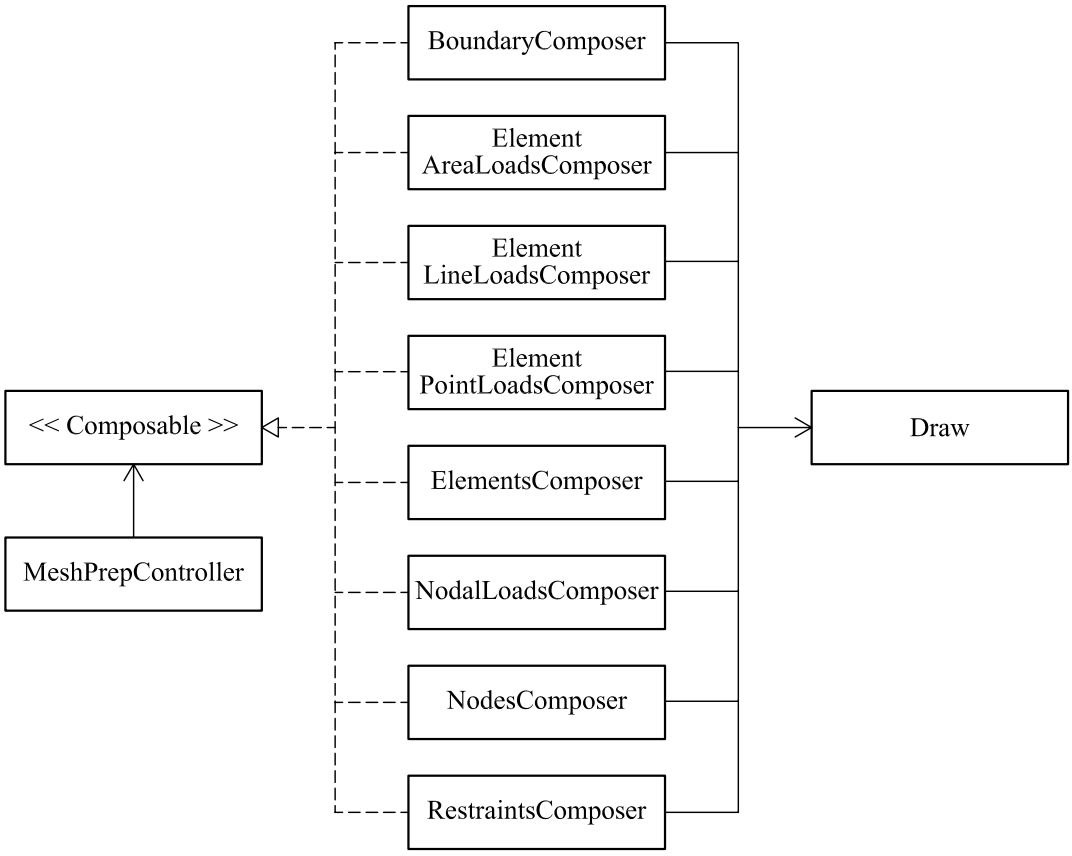


Figura 3.15: Compositores do controlador de malha.

As classes derivadas de *Draw*, utilizadas pelo pré-processador, estão ilustradas na Figura 3.16. Essas classes fazem parte do projeto *br.ufmg.dees.insane.draw*.

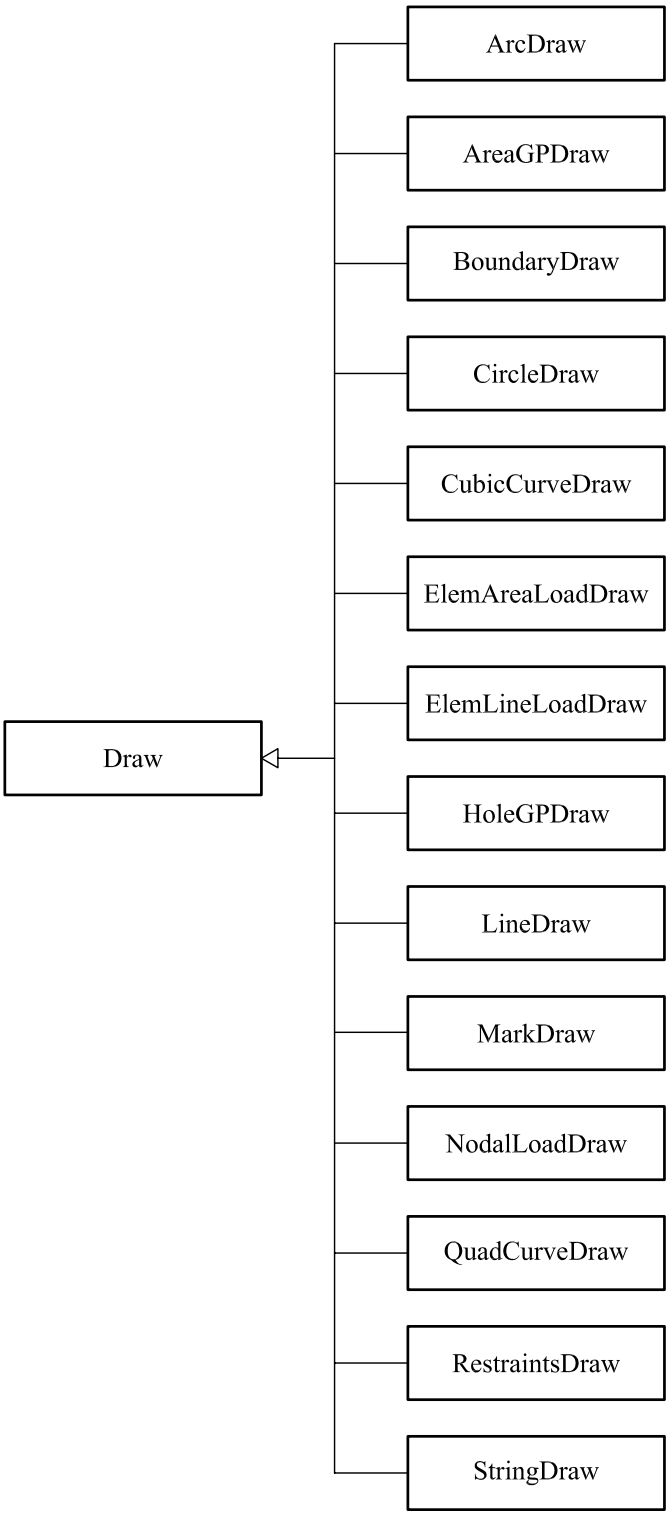


Figura 3.16: Classes de desenho utilizadas pelo pré-processador.

Os vários **Draws** criados pelos compositores são representados na área de desenho (**DrawingArea**) com características específicas (cor, fonte, espessura do traço,

dentre outros). Os atributos de cada desenho são definidos em objetos que caracterizam o estado da vista (classes que implementam a interface *Visualizable*). A Figura 3.17 apresenta as classes responsáveis pelo visual dos desenhos.

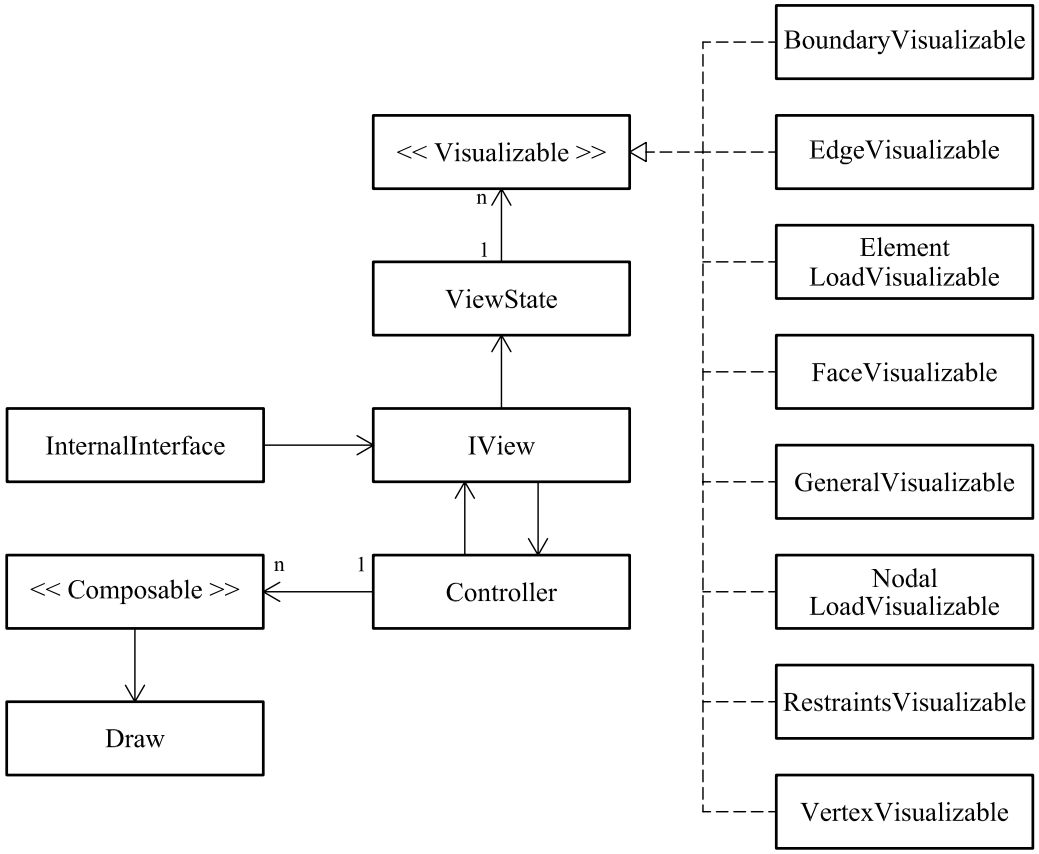


Figura 3.17: Classes que implementam a interface *Visualizable*.

A Figura 3.18 mostra a classe *DrawingArea*. Ela é a responsável pela representação de objetos do tipo *Graphics2D* (sub-classe abstrata da classe *Graphics*) nas janelas internas do pré-processador. Um objeto **Graphics** gerencia um contexto gráfico e desenha pixels na tela para representar os desenhos. Estes objetos possuem métodos para desenhar, manipular fontes, manipular cores e outros (Deitel e Deitel, 2003). A classe *JComponent* (*java.awt.Graphics*) contém o método *paintComponent()* que pode ser utilizado para desenhar imagens gráficas.

A classe *PrintableGridCanvas* possui uma referência para um objeto da classe *Transform* que é a responsável pelas transformações do sistema de coordenadas do

dispositivo gráfico (coordenadas da vista ou coordenadas do mundo) e por estipular os limites do desenho. Também implementa a interface *Printable* (*java.awt.print*) e interfaces relativas a eventos de mouse (*java.awt.event*).

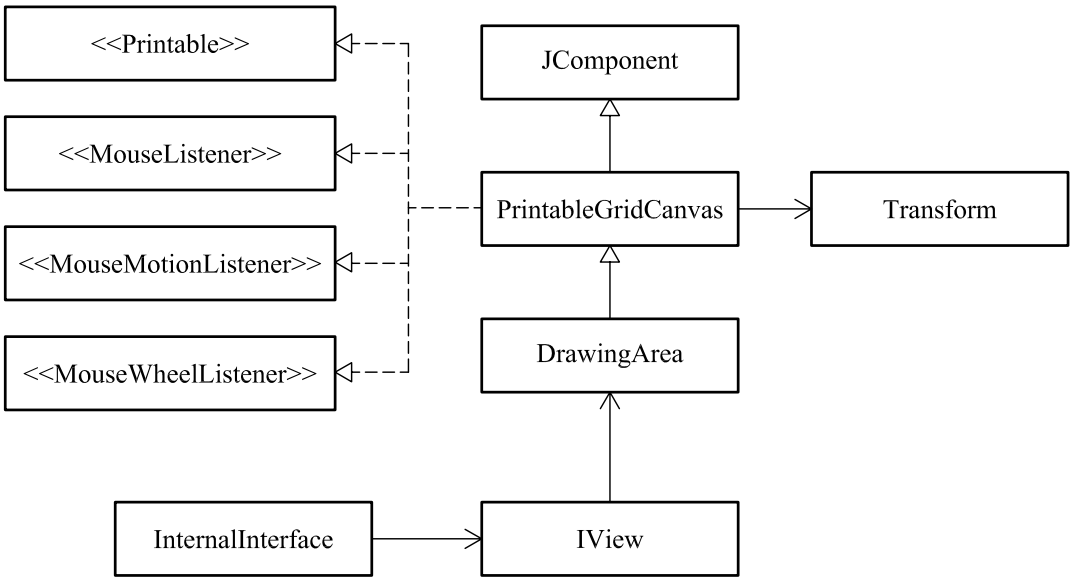


Figura 3.18: Diagrama UML para a área de desenho.

3.3.3 Classes para Transformações Geométricas

Para possibilitar a aplicação de transformações geométricas ou projeções sobre a *vista* (representada pela classe *IView*), o pré-processador utiliza-se de várias classes que foram implementadas no trabalho de Penna (2007) e disponibilizadas no pacote *br.ufmg.dees.insane.geometry.geometricTransformations*.

Os comandos definidos para executar as transformações acessam, através do método *execute()*, o estado da vista (classe *ViewState* na Figura 3.19) para adicionar uma transformação específica (objeto da classe *GeneralTransform*). Esses mesmos comandos solicitam aos controladores que façam a composição dos desenhos. As classes que implementam a interface *Composable* compõem os desenhos utilizando a matriz de transformação resultante (concatenação das várias matrizes) disponível na classe *GeometricTransformation*.

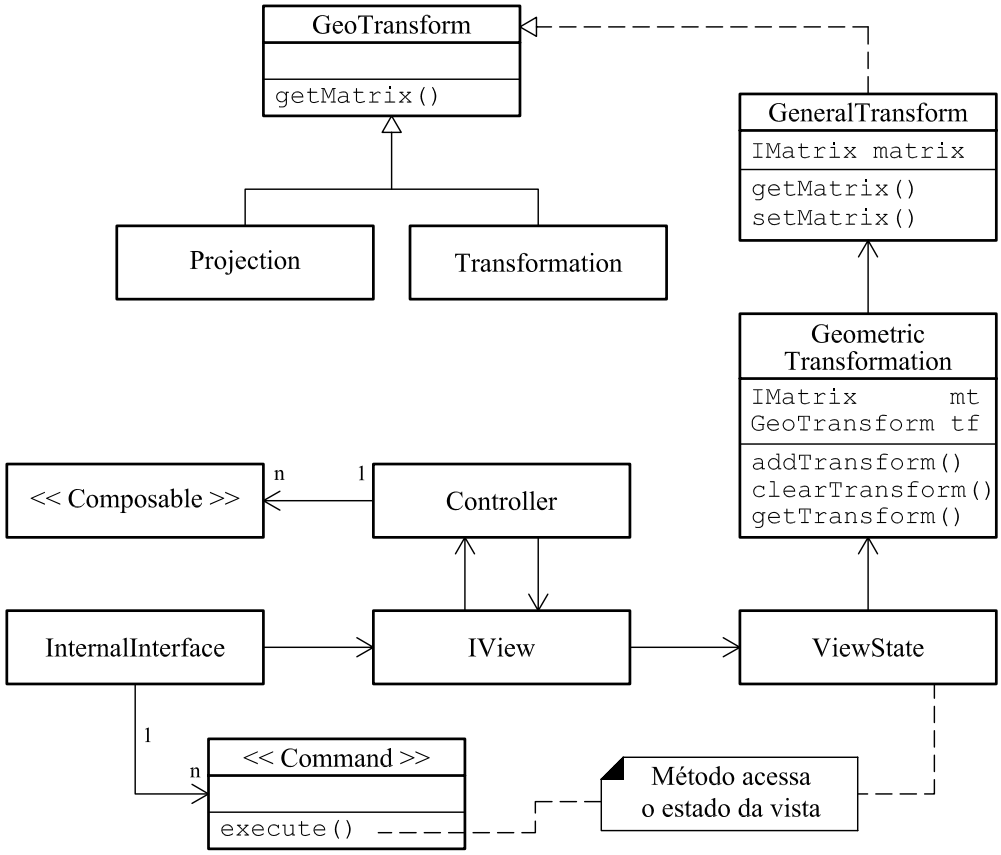


Figura 3.19: Relação entre a vista e as transformações geométricas.

A seguir é apresentado, a título de exemplo, um trecho de código referente ao comando que projeta o modelo no plano XY.

```
public void execute() {
    ViewState state = intFrame.getView().getViewState();
    state.getGeoTransform().clearTransform();
    state.getGeoTransform().addTransform(new Axonometric(0, 0));
    intFrame.getView().getController().compose();
    intFrame.getView().getDrawingArea().repaint();
}
```

As Figuras 3.20 e 3.21 apresentam as classes herdeiras de *Projection* e *Transformation*, respectivamente.

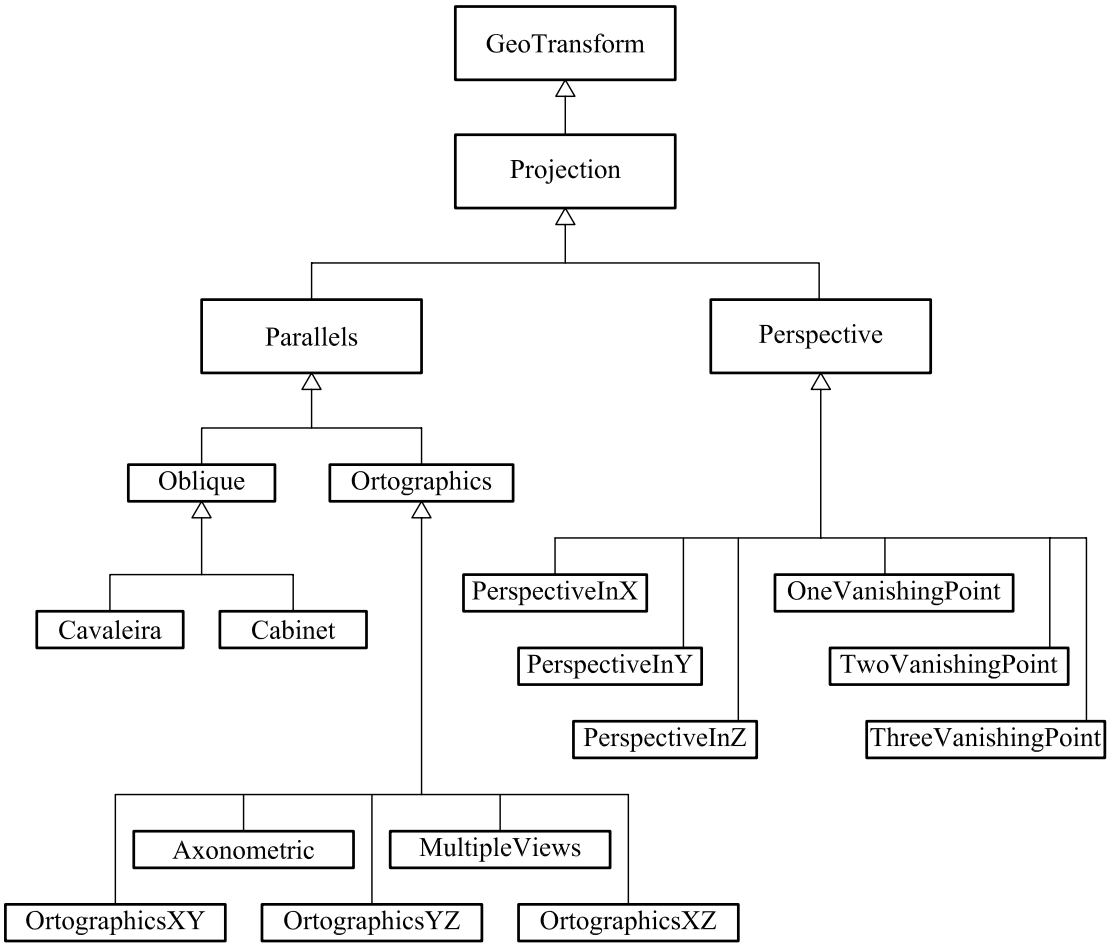


Figura 3.20: Classes herdeiras de *Projection*.

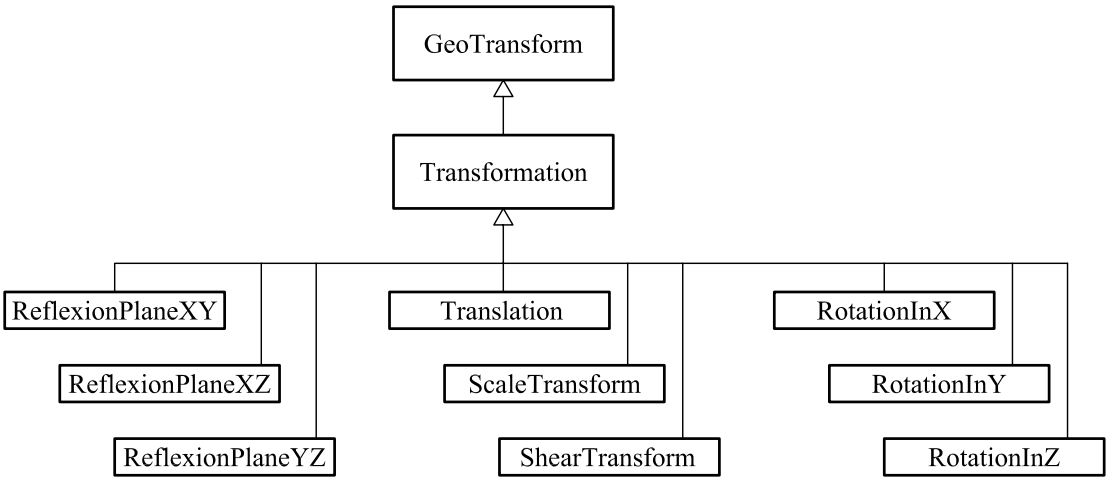


Figura 3.21: Classes herdeiras de *Transformation*.

3.3.4 Classes para a Geração de Malhas

Todas as classes relacionadas com o processo de geração de malhas por Mapeamento Transfinito, implementadas no trabalho de Brugiolo (2004), foram revisadas para possibilitar o mapeamento nos planos XY, XZ e YZ e o armazenamento dos dados na estrutura de "Semi-Arestas". O processo de mapeamento (Figura 3.22) se dá através das seguintes etapas:

1. Seleção e divisão do contorno de uma região (*Face*);
2. Definição de duas, três ou quatro fronteiras (conjunto de vértices);
3. Aplicação de um tipo de mapeamento (Lofting, Trilinear ou Bilinear) conforme o número de fronteiras criadas.

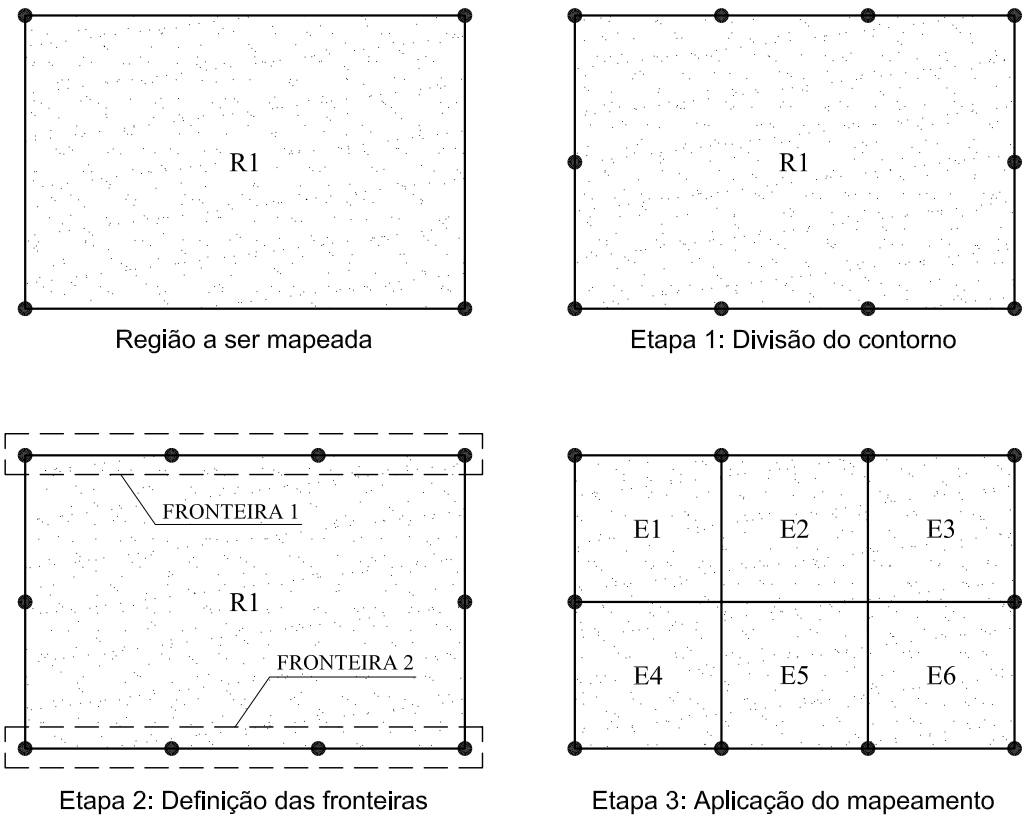


Figura 3.22: Etapas do mapeamento.

A Figura 3.23 apresenta a classe *Mapping* (pacote *model.prep.mesh.mapping*). Ela possui uma lista de objetos da classe *IPoint3d* (projeto *br.ufmg.dees.insane.util*), para armazenar todos os pontos gerados após o mapeamento e uma lista de objetos da classe *MapElement* para armazenar a incidência desses pontos em cada elemento.

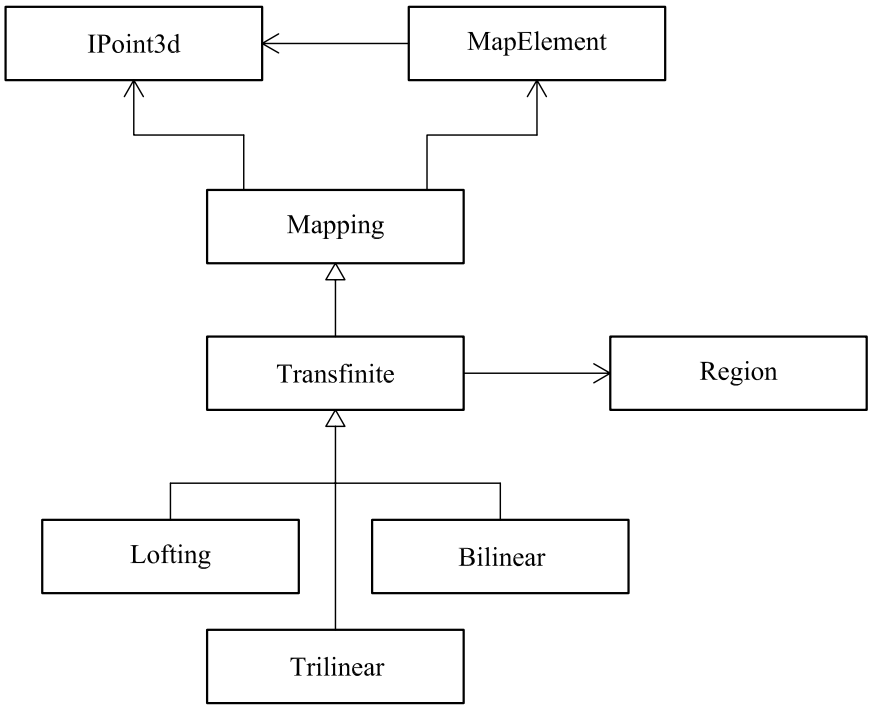


Figura 3.23: Classes do gerador de malhas.

Com a aplicação de um dos sub-tipos de Mapeamento Transfinito (Lofting, Trilinear ou Bilinear) sobre uma região (encapsulada pela classe *Region*), novos *Vértices* são gerados a partir dos pontos mapeados e novas *Faces* são criadas com base na lista de *MapElement*. Esses componentes são, em seguida, adicionados na estrutura de dados do modelo.

A Figura 3.24 detalha a classe *Region*. Ela possui uma referência para a área (*Face*) a ser mapeada e uma lista de fronteiras (objetos da classe *Boundary*). Cada sub-classe de *Transfinite* possui uma referência para uma sub-classe de *Region*, pois dependem do número de fronteiras criadas.

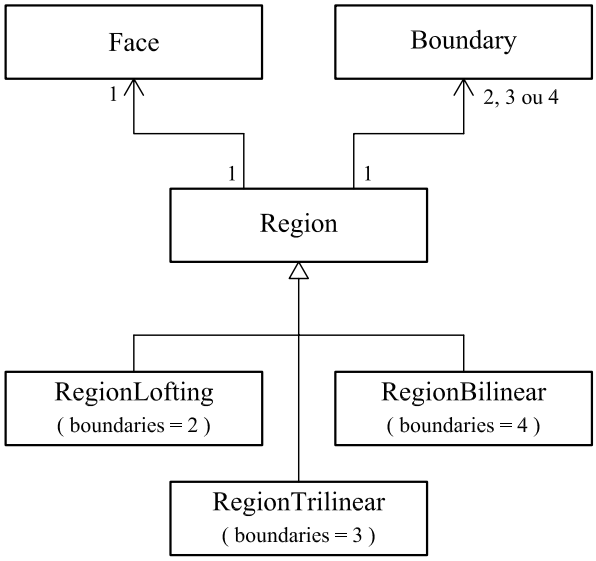


Figura 3.24: Classe *Region*.

As Figuras 3.25 e 3.26 apresentam as classes *Lofting* e *Bilinear*. Ambas possuem classes mais especializadas que realizam o mapeamento com elementos quadrilaterais ou triangulares, lagrangeanos ou serendípticos.

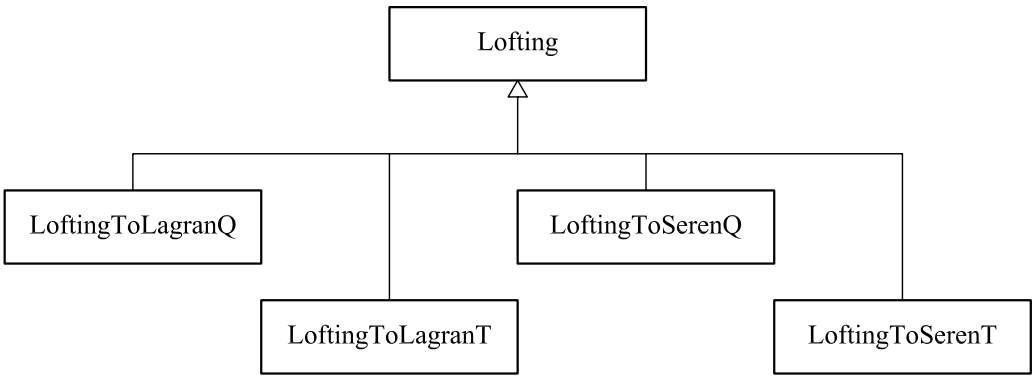


Figura 3.25: Mapeamento Transfinito “Lofting”.

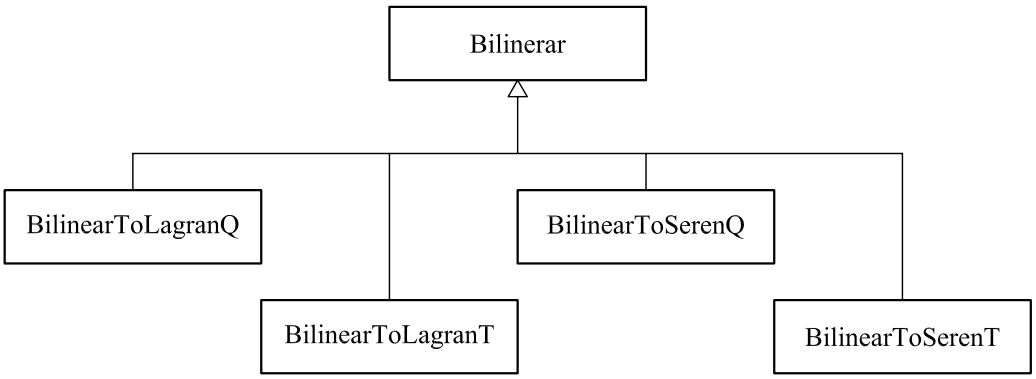


Figura 3.26: Mapeamento Transfinito Bilinear.

A Figura 3.27 apresenta a classe *Trilinear* e suas sub-classes que realizam o mapeamento com elementos triangulares lagrangeanos ou serendípticos.

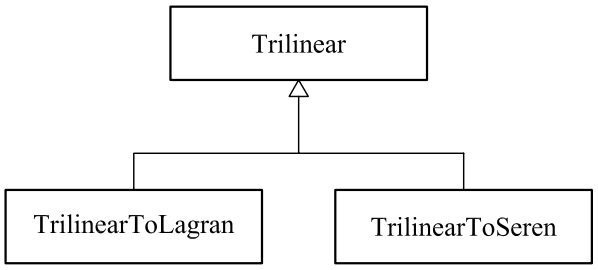


Figura 3.27: Mapeamento Transfinito Trilinear.

3.4 Utilização da Estrutura de Dados "Semi-Arestas"

Esta seção apresenta a maneira como os modelos do pré-processador (*GeoPrepModel* e *MeshPrepModel*) utilizam a estrutura de dados de "semi-arestas", implementada no trabalho de Penna (2007).

Conforme mencionado anteriormente, o modelo geométrico e o modelo de malha possuem uma referência para um objeto da classe *HalfEdgeDataStructure* de forma independente (Figura 3.28). Os vários métodos implementados nessa classe dão acesso à subdivisão planar do modelo (*PlanarSubdivision*) e, consequentemente, acesso às listas de faces, arestas e vértices que formam a geometria.

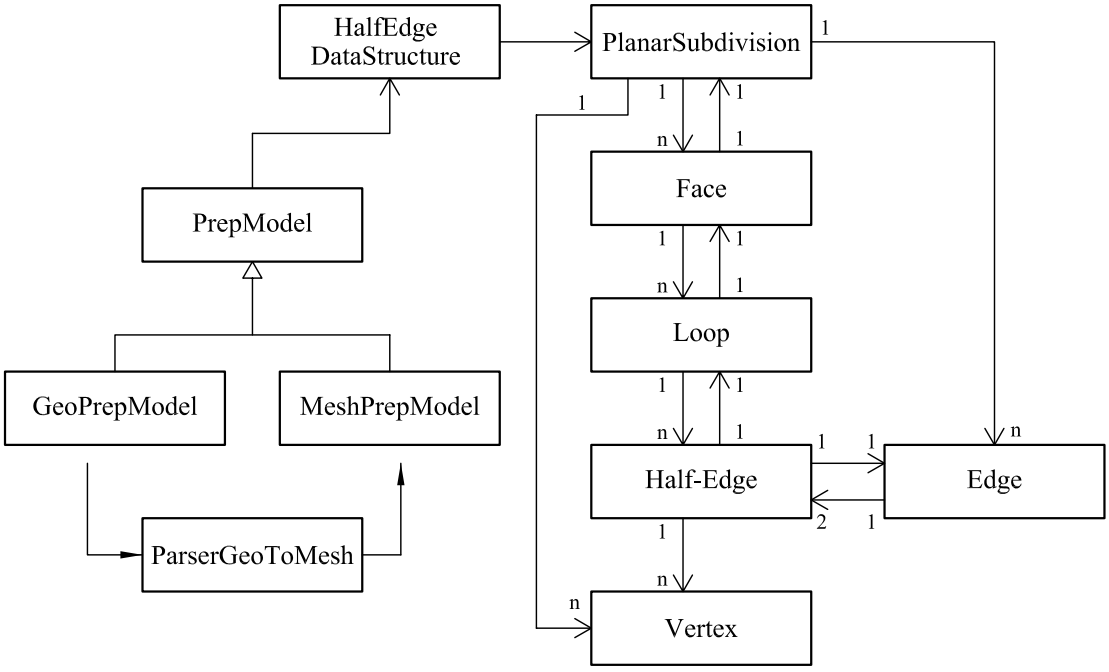


Figura 3.28: A Subdivisão Planar do Modelo.

Na subdivisão planar do *GeoPrepModel*, objetos da classe *Vertex* são utilizados para armazenar as coordenadas (x, y, z) dos vértices geométricos e objetos da classe *Edge* são usados para armazenar as primitivas geométricas tais como: linhas, curvas, círculos e arcos. As regiões formadas por polígonos são representadas por objetos da classe *Face*.

Na interface gráfica, quando se faz a mudança entre os módulos *Geometria* e *Malha* (Figura 3.29), a subdivisão planar do modelo geométrico é clonada e adaptada, pelo *ParserGeoToMesh*, para formar a subdivisão planar do modelo de malha.

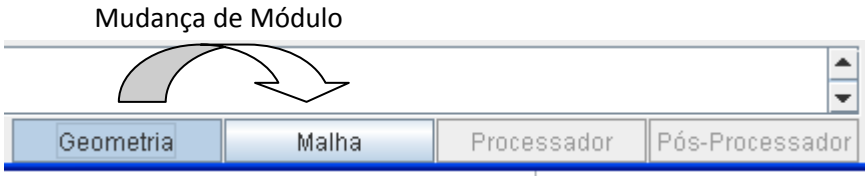


Figura 3.29: Mudança de módulo e ativação do "parser".

Um diálogo (Figura 3.30) é apresentado para que o usuário defina a forma de atuação do "parser". Conforme a escolha, será criada uma *PlanarSubdivision* capaz de representar uma malha formada por elementos de barra, por elementos planos ou com elementos combinados.

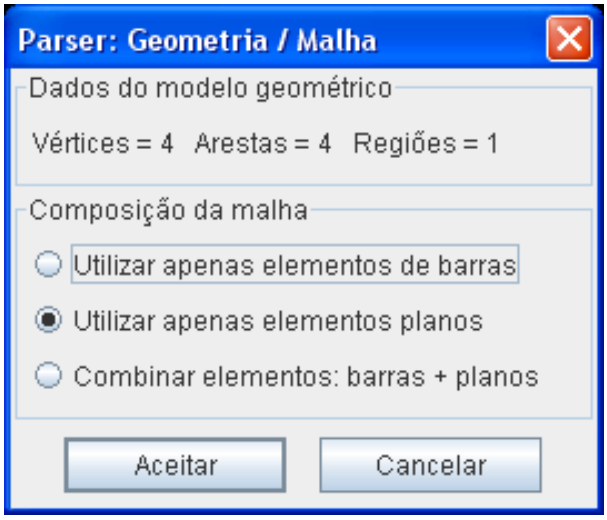


Figura 3.30: Diálogo utilizado pelo *ParserGeoToMesh*.

A Figura 3.31 ilustra como é montada a subdivisão planar do *MeshPrepModel* para representar uma malha formada apenas por elementos de barra. Novos vértices são criados a partir dos existentes no modelo geométrico (caminho 1) e as arestas são transformadas em objetos da classe *Face* (caminho 2) para representar os elementos de barra com 2, 3 ou 4 nós ("open faces").

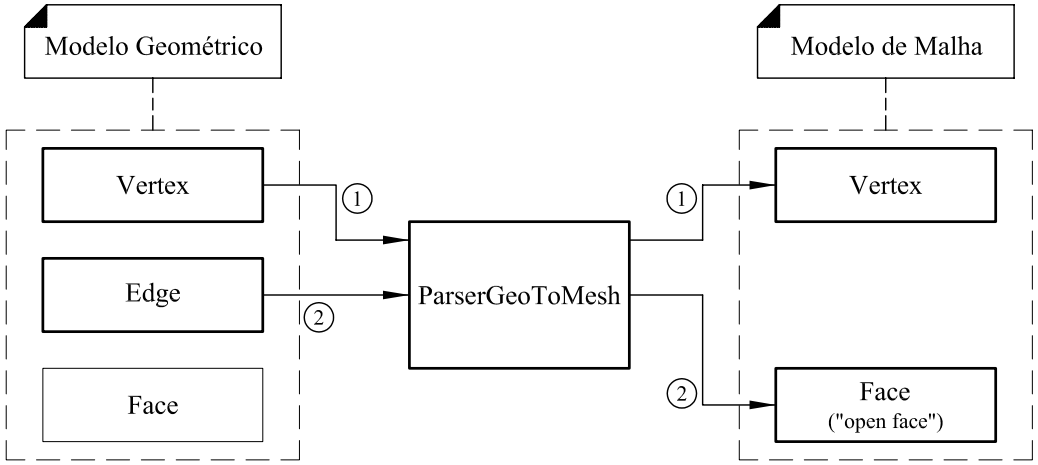


Figura 3.31: Parser *GeoToMesh* para modelos com elementos de barra.

A Figura 3.32 ilustra como é montada a subdivisão planar do *MeshPrepModel* para representar uma malha formada apenas por elementos planos. As arestas são mantidas na lista de arestas da subdivisão planar mas não são transformadas. As faces definidas na geometria continuam sendo objetos da classe *Face* (caminho 3) e representam os elementos planos ("closed faces").

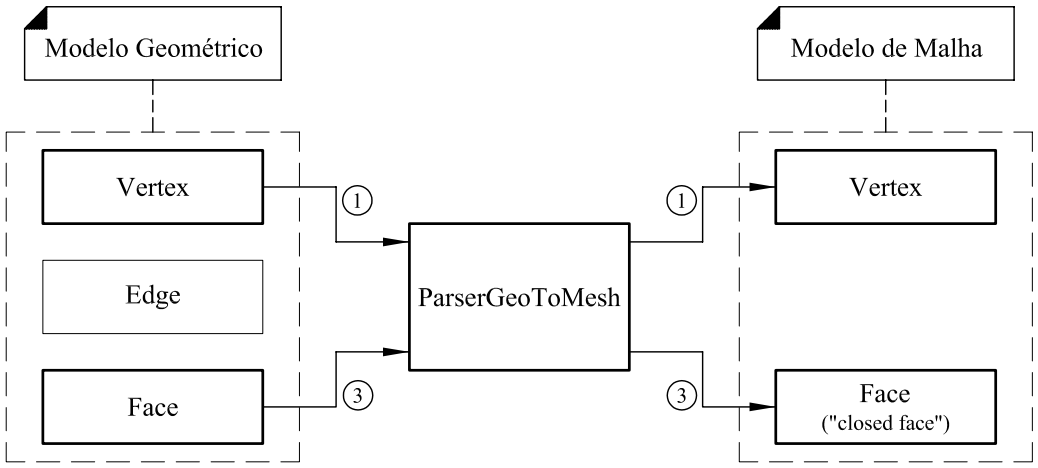


Figura 3.32: Parser *GeoToMesh* para modelos com elementos de planos.

A Figura 3.33 ilustra como é montada a subdivisão planar do *MeshPrepModel* para representar uma malha com elementos combinados. As arestas são transformadas em objetos da classe *Face* (caminho 2) para representar elementos de barra com 2, 3 ou 4 nós ("open faces"). As faces continuam sendo objetos da classe *Face*

(caminho 3) e representam os elementos planos (“closed faces”).

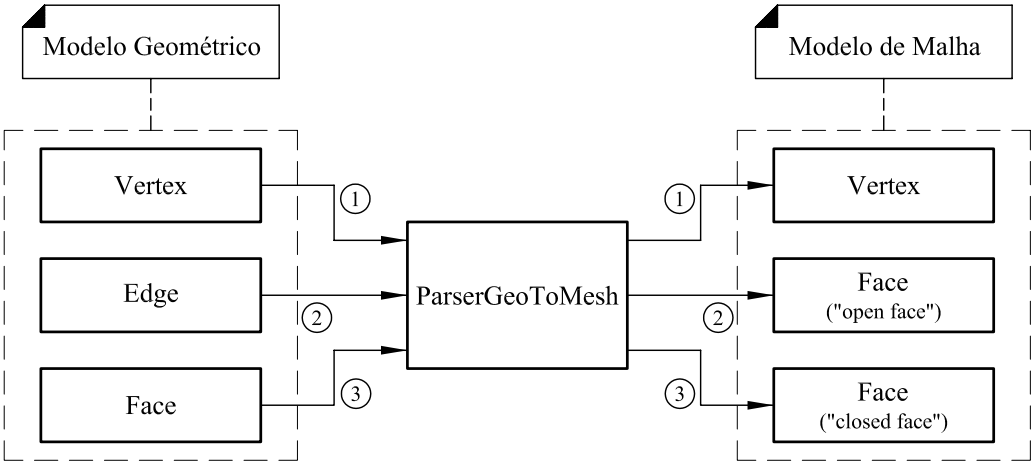


Figura 3.33: Parser *GeoToMesh* para modelos combinados.

3.4.1 Subdivisão Planar: Variáveis das Classes

A classe *Edge* possui uma variável da classe *Curve* (Figura 3.34) para armazenar as primitivas arco, círculo, curva cúbica ou curva quadrática do modelo geométrico.

Edge	
`String`	`id;`
`HalfEdge`	`rightHalfEdge;`
`HalfEdge`	`leftHalfEdge;`
`Vertex`	`vStart;`
`Vertex`	`vEnd;`
`Curve`	`shapeCurve;`

Figura 3.34: Variáveis da classe *Edge*.

A Figura 3.35 apresenta as classes implementadas para essas primitivas.

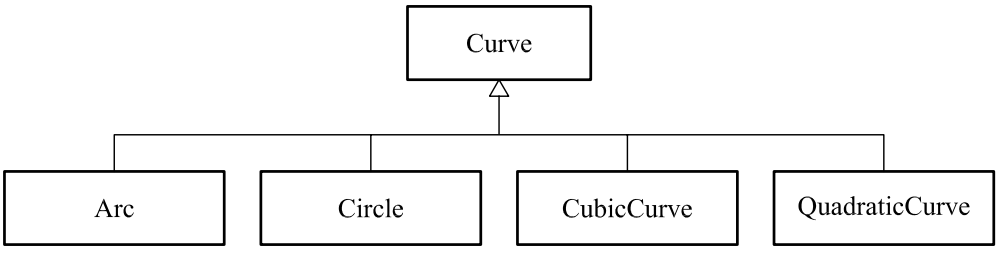


Figura 3.35: Sub-classes de *Curve*.

A Figura 3.36 apresenta as variáveis que foram definidas para as classes *PlanarSubdivision*, *Face*, *Loop*, *HalfEdge* e *Vertex*.

PlanarSubdivision	
String	id;
LinkedList<Face>	faceList;
LinkedList<Edge>	edgeList;
LinkedList<Vertex>	vertexList;

Face	
String	id;
LinkedList<Loop>	loopList;
HashMap<String, Object>	values;
ArrayList<String>	keys;
PlanarSubdivision	planarSubdivision;

Loop	
String	id;
LinkedList<HalfEdge>	halfEdgeList;
Face	face;

HalfEdge	
String	id;
Edge	edge;
Vertex	vertex;
Loop	loop;

Vertex	
String	id;
IPoint3d	coords;
HashMap<String, Object>	values;
ArrayList<String>	keys;

Figura 3.36: Variáveis das classes da Subdivisão Planar.

As classes *Vertex* e *Face* possuem a variável *values* do tipo *HashMap<String, Object>* (pacote *java.util.HashMap*) para permitir o armazenamento de informações

genéricas. A relação de todas as chaves utilizadas pelo modelo do pré-processador podem ser encontradas no Apêndice B.

3.4.2 Representação dos Elementos de Barra

As Figuras 3.37, 3.38 e 3.39 mostram como os elementos de barra com 2, 3 e 4 nós ("open faces") são representados na estrutura de "Half-Edges".

Para cada elemento é criado um objeto da classe *Face* com um único *Loop* que por sua vez, armazena cada *HalfEdge* com seu respectivo *Vertex*. A incidência dos vértices é armazenada no mapa da *Face*, (*HashMap<String, Object>*) numa chave chamada *"INCIDENCE"* associada a um objeto do tipo *ArrayList<String>*.

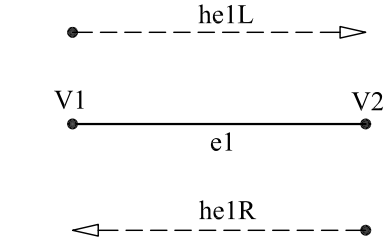


Figura 3.37: Representação do elemento de barra com 2 nós.

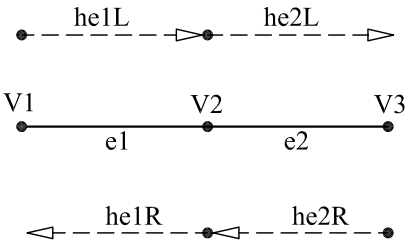
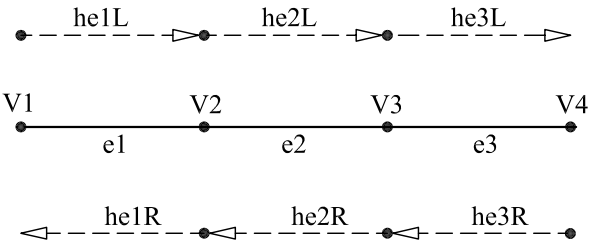


Figura 3.38: Representação do elemento de barra com 3 nós.

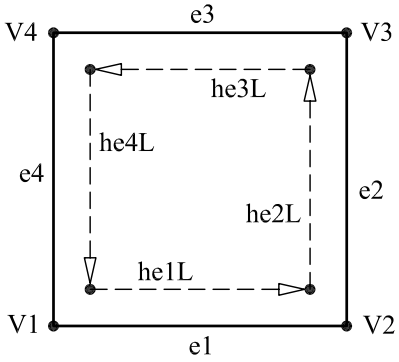


Face 3: Loop 1 [he1L(v1) he2L(v2) he3L(v3) he3R(v4) he2R(v3) he1R(v2)]
Incidence: [V1 V2 V3 V4]

Figura 3.39: Representação do elemento de barra com 4 nós.

3.4.3 Representação dos Elementos Planos

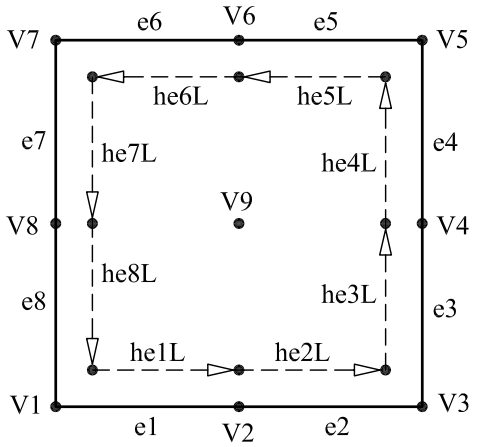
A Figura 3.40 ilustra como os elementos planos, formados apenas por nós no contorno, são representados em objetos da classe *Face*.



Face 4: Loop 1 [he1L(v1) he2L(v2) he3L(v3) he4L(v4)]
Incidence: [V1 V2 V3 V4]

Figura 3.40: Representação de elemento plano com 4 nós (Q4).

No caso de elementos que possuem nós internos (caso do Q9 ilustrado na Figura 3.41), os nós do contorno são armazenados como já ilustrado na Figura 3.40. Como os nós internos não estão associados a nenhuma "semi-aresta", eles são armazenados na chave *"INCIDENCE"* do mapa da *Face* (*HashMap<String, Object>*). Uma chave chamada *"INTVERTEX"*, associada a um objeto do tipo *ArrayList<String>*, também guarda a lista desses nós, referenciados pelos seus identificadores (Ids).

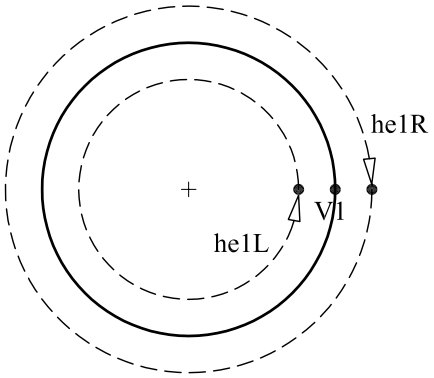


Face 4: Loop 1 [he1L(v1) he2L(v2) he3L(v3) he4L(v4)
he5L(v5) he6L(v6) he7L(v7) he8L(v8)]
Incidence: [V1 V2 V3 V4 V5 V6 V7 V8 V9]
IntVertex: [V9]

Figura 3.41: Representação de elemento plano com 9 nós (Q9).

3.4.4 Representação das Regiões

A Figura 3.42 mostra como uma região circular é representada em um objeto da classe *Face*. O *Loop* é formado pelas "semi-arestas" que compartilham o mesmo *Vertex*.

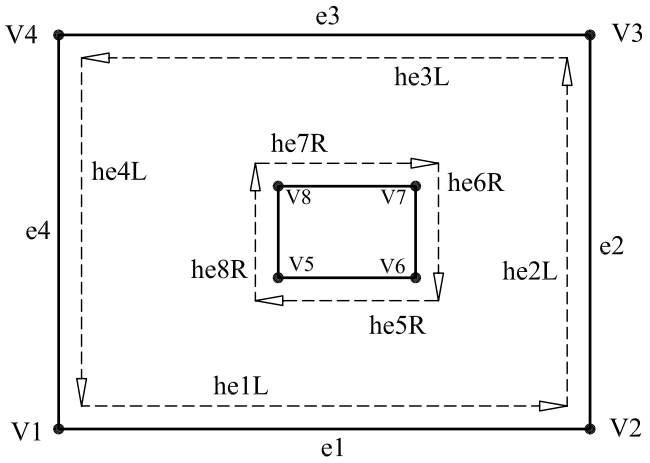


Face 5: Loop 1 [he1L(v1) he1R(v1)]
Incidence: [V1 V1]

Figura 3.42: Representação de uma região circular.

A Figura 3.43 ilustra como é representada, num objeto da classe *Face*, uma região

com furo. A presença de mais de um *Loop*, formado pelas "semi-arestas" ordenadas no sentido contrário às do "loop" mais externo, caracteriza a existência de furos na região.



Face 6: Loop 1 [he1L(v1) he2L(v2) he3L(v3) he4L(v4)]
Loop 2 [he8R(v5) he7R(v8) he6R(v7) he5R(v6)]
Incidence: [V1 V2 V3 V4 V5 V8 V7 V6]

Figura 3.43: Representação de uma região com furo.

A Figura 3.44 ilustra uma região com furo criada no *Módulo Geometria* do pré-processador.

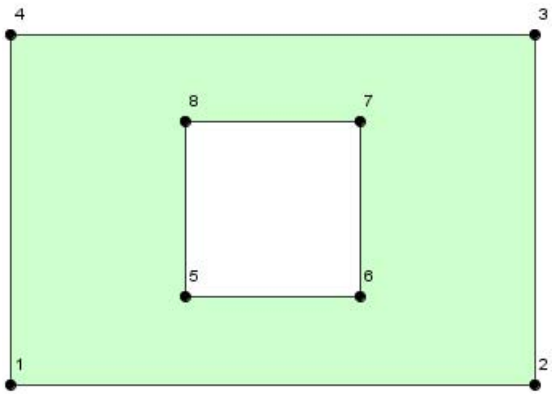


Figura 3.44: Exemplo de uma região com furo.

3.5 Implementação dos Operadores de Euler

Para o desenvolvimento do modelo geométrico e o gerenciamento da estrutura de dados, foram implementados e adaptados da proposta original de Mäntyla (1987), oito Operadores de Euler, conforme indicado na Tabela 3.1.

Tabela 3.1: Operadores de Euler implementados

Chave	**Descrição**	**Categoria**
MV	Make Vertex	Local
KV	Kill Vertex	Local
ME	Make Edge	Local
KE	Kill Edge	Local
MF	Make Face	Local
KF	Kill Face	Local
KFMH	Kill Face Make Hole	Global
MFKH	Make Face Kill Hole	Global

O operador **MV** adiciona um vértice na estrutura de dados a partir de um ponto (x,y,z) fornecido pelo usuário. O operador **ME** cria uma aresta representada por linhas ou curvas. Seus vértices são definidos a partir daqueles já existentes na estrutura de dados. O operador **MF** cria uma face após a seleção de arestas existentes. O operador **KFMH** é um operador global (arranjo local de vértices e arestas de uma face não são alterados). Para cada operador que adiciona uma entidade na subdivisão planar do modelo (vértice, aresta ou face), existe um operador que realiza a operação contrária.

Para permitir a edição da malha, gerada após a aplicação de um Mapeamento Transfinito, foram criados mais dois operadores. O operador **KVF** (Kill Vertex Face) remove um vértice (interno ou do contorno) de uma face e as faces a ele adjacentes. O operador **KFV** (Kill Face Vertex) remove uma face e verifica se esta

possui vértices internos para também removê-los.

A Tabela 3.2 resume as classes que foram implementadas para cada operador. Elas foram agrupadas no projeto *br.ufmg.dees.insane.geometry*.

Tabela 3.2: Classes implementadas para os Operadores de Euler.

Classe	**Função**
OpEulerMV	Criar e adicionar um novo vértice.
OpEulerKV	Remover um vértice e as entidades adjacentes (aresta e face).
OpEulerME	Criar e adicionar uma nova aresta.
OpEulerKE	Remover uma aresta e as faces a ela adjacentes.
OpEulerMF	Criar uma região delimitada por linhas e/ou curvas.
OpEulerKF	Remover uma face.
OpEulerKFMH	Remover uma face interna à outra criando um furo.
OpEulerMFKH	Remover um furo recriando o interior da face.
OpEulerKVF	Remover o vértice selecionado e as faces adjacentes.
OpEulerKFV	Remover a face selecionada e vértices internos.

Instâncias das classes que representam os operadores de Euler são criadas, em tempo de execução do programa, através do método ***execute()***, presente nas classes que implementam a interface ***Command***. A Figura 3.45 ilustra a construção de um operador do tipo *OpEulerMV*.

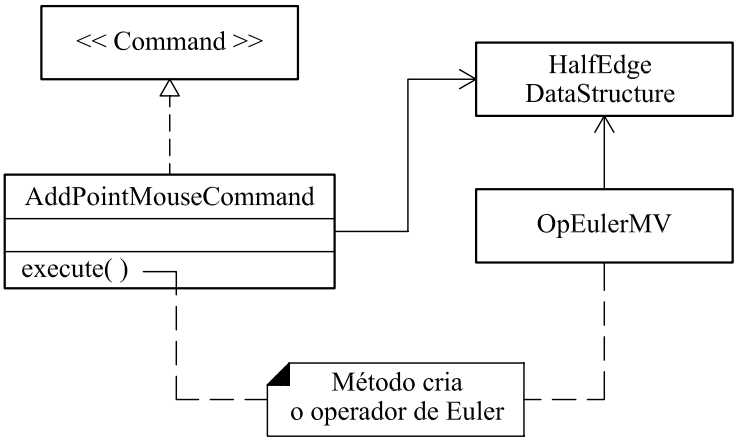


Figura 3.45: Exemplo de construção de um Operador de Euler.

3.6 Recursos JAVA

Para a elaboração do pré-processador foram utilizados vários recursos da linguagem JAVA. Os principais pacotes e classes adotadas foram:

1. Pacotes **java.awt** e **javax.swing**, para concepção da interface gráfica (classes **java.awt.Graphics** e **java.awt.Graphics2D**) e das caixas de diálogo;

2. Pacote **java.util** (classes **java.util.ArrayList**, **java.util.LinkedList** e **java.util.HashMap**), para permitir a manipulação da estrutura de dados da aplicação;

3. Pacote **java.io** para permitir a serialização dos objetos e manipulação de arquivos;

4. Pacote **java.lang** para permitir o uso de classes que constituem recursos básicos da linguagem, necessários à execução de qualquer programa JAVA;

5. Pacote **java.text** para permitir a manipulação de textos, datas, números e mensagens.

Além do ambiente de desenvolvimento **J2SDK** da *Sun Microsystems*, disponivel em `http://www.sun.com`, também foram utilizados: a plataforma **Eclipse** (`http://www.eclipse.org`) para implementação do código; o **Maven** (`http://maven.apache.org`) para promover o gerenciamento de todos os projetos que constituem o **INSANE**, o **Subversion** (`http://subversion.tigris.org`) para permitir o controle de versões e o **Bugzilla** (`http://www.bugzilla.org`) para manter um histórico de erros encontrados no código.

Capítulo 4

RECURSOS DO PRÉ-PROCESSADOR

4.1 Introdução

Neste capítulo são apresentados os recursos disponibilizados no pré-processador, destacando-se os comandos e diálogos definidos nos módulos *Geometria* e *Malha*. A Figura 4.1 ilustra a organização da interface gráfica.

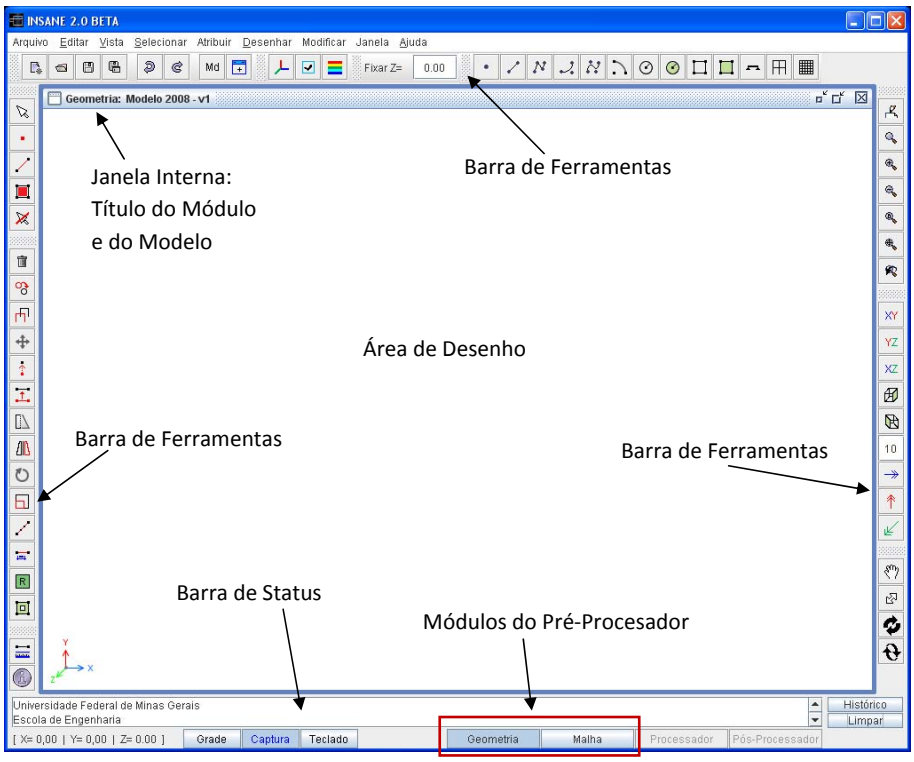


Figura 4.1: Organização da interface gráfica.

Conforme explicado no Capítulo 3, os menus e as barras de ferramentas são definidos na interface no momento em que uma janela do tipo *InternalInterfaceGeo* ou *InternalInterfaceMesh* é ativada.

O pré-processador permite trabalhar com múltiplas janelas internas e, em cada uma, o modelo pode ser visto com uma projeção diferente. A Figura 4.2 ilustra a geometria de um pórtico espacial observado por três vistas.

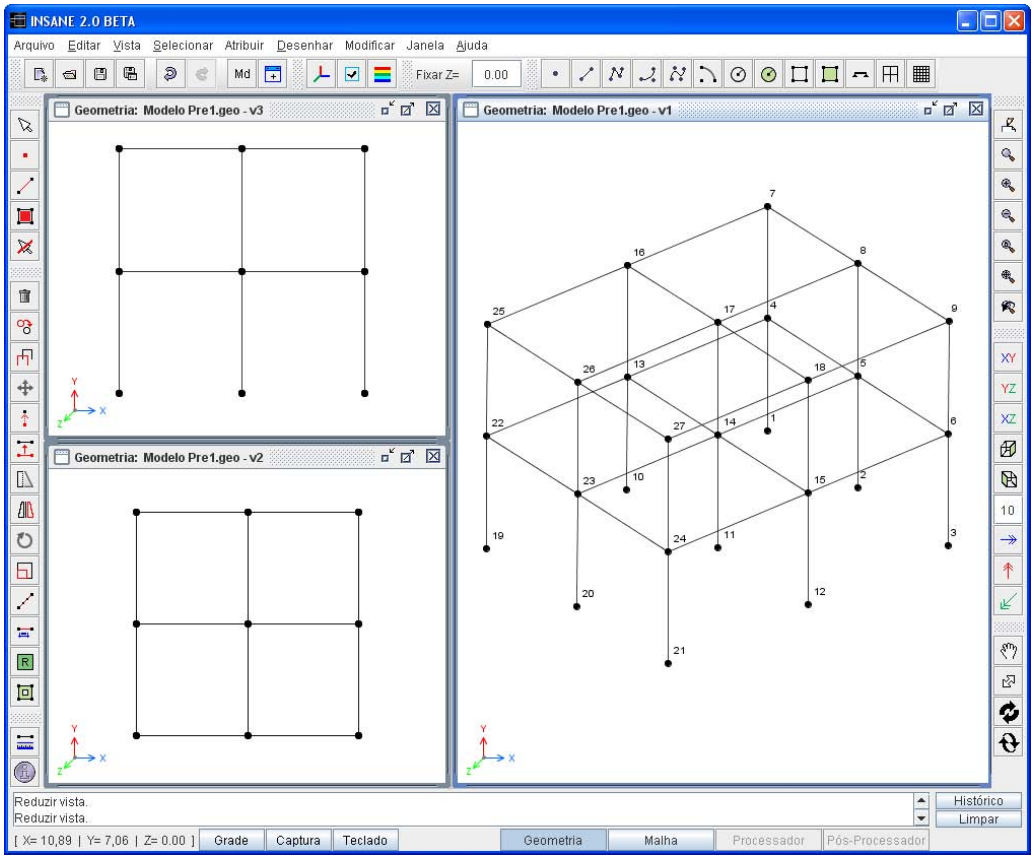


Figura 4.2: Interface com múltiplas janelas.

4.2 Recursos do Módulo Geometria

Para a criação do modelo geométrico, foram disponibilizados os seguintes recursos: seleção específica de objetos; desenho de pontos, retas, curvas cúbicas, curvas quadráticas, arcos, círculos e regiões circulares ou quadrilaterais; gerador de geometria para viga contínua, pórtico plano, pórtico espacial e grelha; vários comandos de

edição; comandos para criação de regiões poligonais (com ou sem furos); comandos para aplicação de transformações geométricas e projeções e comandos para ajuste das preferências do usuário. Esses recursos foram agrupados em menus e barras de ferramentas de acordo com a categoria (Figuras 4.3, 4.4 e 4.5).

Figura 4.3: Menu e barra de ferramentas superior.

Figura 4.4: Barra de ferramentas para primitivas geométricas.

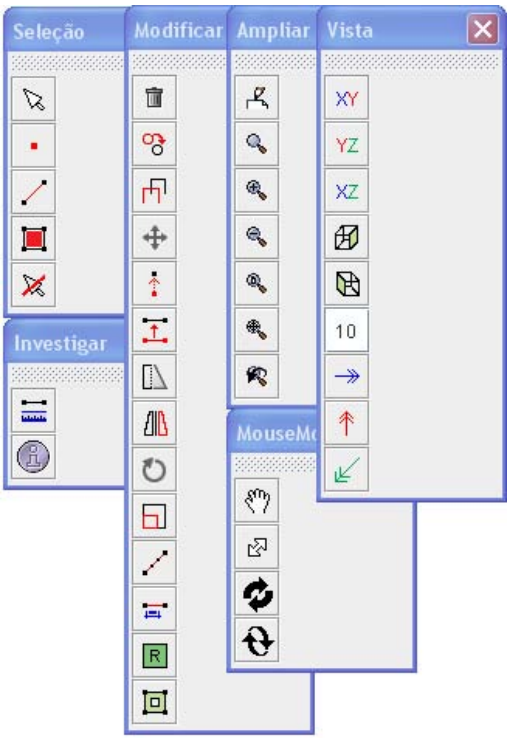


Figura 4.5: Barras de ferramentas laterais.

Quando a opção *Arquivo* ≫ *Novo* (Figura 4.6) é acionada, um novo modelo geométrico é criado (*GeoPrepModel*) e uma janela do tipo *InternalInterfaceGeo* é adicionada na interface para representá-lo. Também é criada uma pasta com o nome do modelo para que esse possa ser persistido.

Figura 4.6: Menu Arquivo.

Através da opção *Arquivo* ≫ *Abrir*, diferentes modelos geométricos (arquivos *.geo*) podem ser abertos simultaneamente.

No *Menu Editar* (Figura 4.7) estão as opções para *Desfazer*, *Refazer* e *Cancelar* comandos.

Figura 4.7: Menu Editar.

No *Menu Vista* (Figura 4.8) estão reunidos comandos para o controle da visualização, aplicação de projeções e perspectivas e transformações sobre a vista.

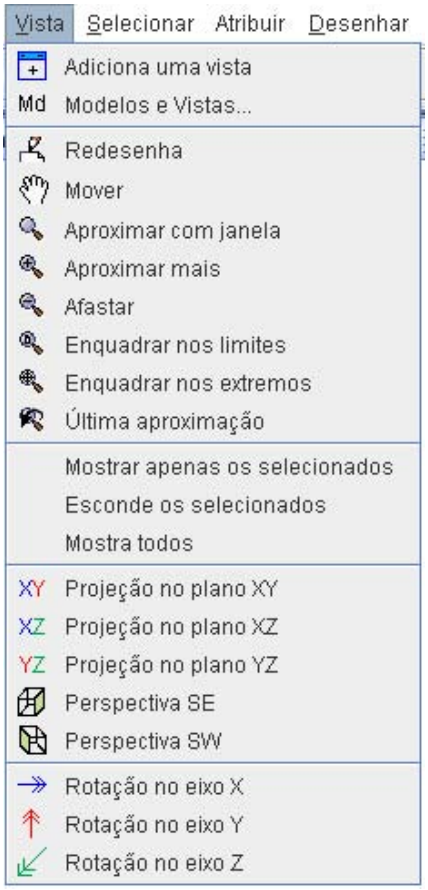


Figura 4.8: Menu Vista.

A opção *Vista* ≫ *Adicionar uma vista* (Figura 4.8) cria novas janelas para o modelo da janela ativa. Já a opção *Vista* ≫ *Modelos e Vistas...*, abre um diálogo (Figura 4.9) para ajudar no gerenciamento das vistas de cada modelo.

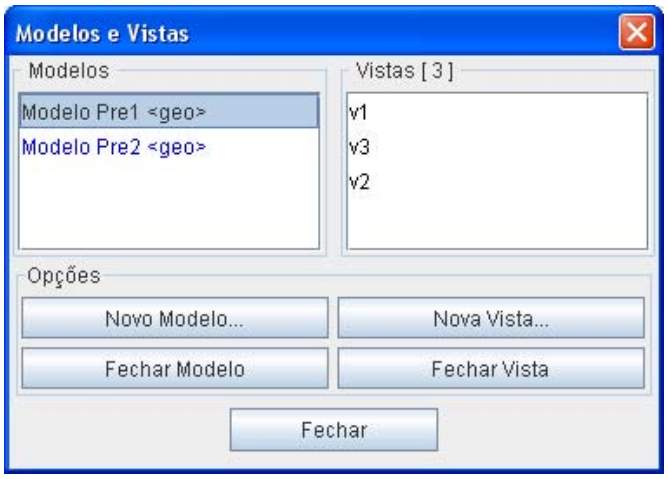


Figura 4.9: Diálogo para gerenciamento de modelos e vistas.

O *Menu Selecionar* (Figura 4.10) oferece diferentes maneiras para seleção de objetos. As teclas F3 a F9 também acionam os modos de seleção.

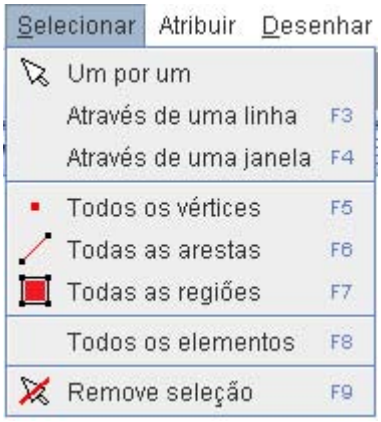


Figura 4.10: Menu Selecionar.

No *Menu Atribuir* (Figura 4.11) estão os comandos relacionados com as preferências do usuário. Os diálogos que são abertos através das opções deste menu são mostrados nas Figuras 4.12 a 4.16.

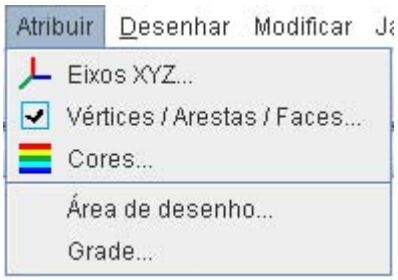


Figura 4.11: Menu Atribuir.

Para especificar o posicionamento do sistema local de coordenadas, utiliza-se o diálogo mostrado na Figura 4.12.

Figura 4.12: Diálogo para posicionamento do sistema de eixos local.

Para especificação dos atributos ou visualização de pontos, linhas e regiões, utiliza-se o diálogo da Figura 4.13.

Figura 4.13: Diálogo para configuração das entidades geométricas.

A atribuição de cores às entidade gráficas também podem ser definidas através do diálogo *Cores* mostrado na Figura 4.14.

Figura 4.14: Diálogo para atribuição de cores.

Através do diálogo *Área de Desenho* (Figura 4.15) pode-se definir as dimensões da área gráfica ou então capturar os limites do enquadramento atual (zoom corrente). Esse diálogo também é apresentado ao usuário quando da criação de um novo modelo geométrico (opção *Arquivo* $\gg$ *Novo*)

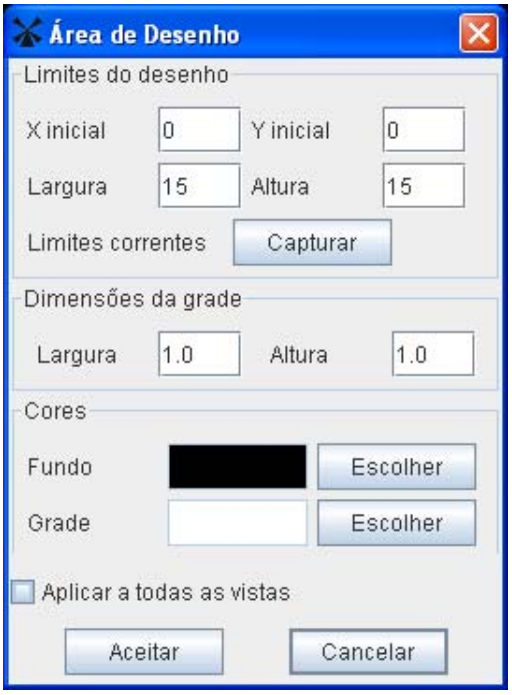


Figura 4.15: Diálogo para configuração da área de desenho.

As dimensões e cor da grade também podem ser ajustadas através do diálogo *Grade* ilustrado na Figura 4.16.

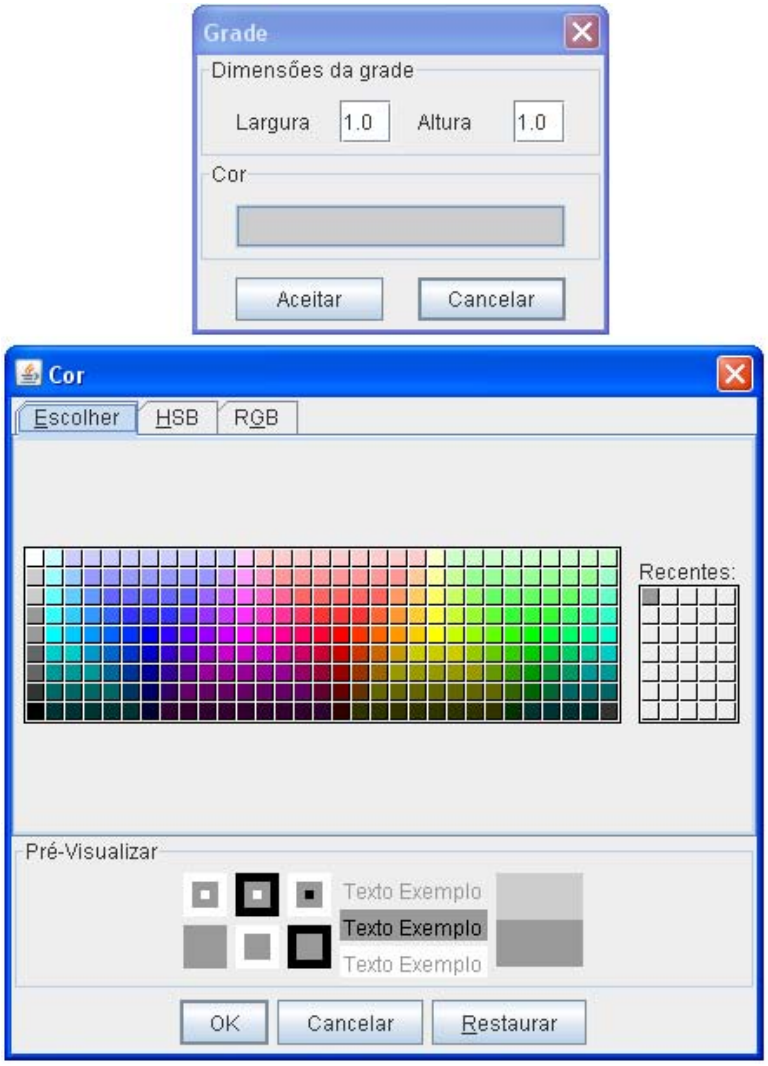


Figura 4.16: Diálogo para configuração da grade.

O *Menu Desenhar* (Figura 4.17) reúne as opções para a geração da geometria. A Figura 4.18 ilustra cada comando.

Figura 4.17: Menu Desenhar.

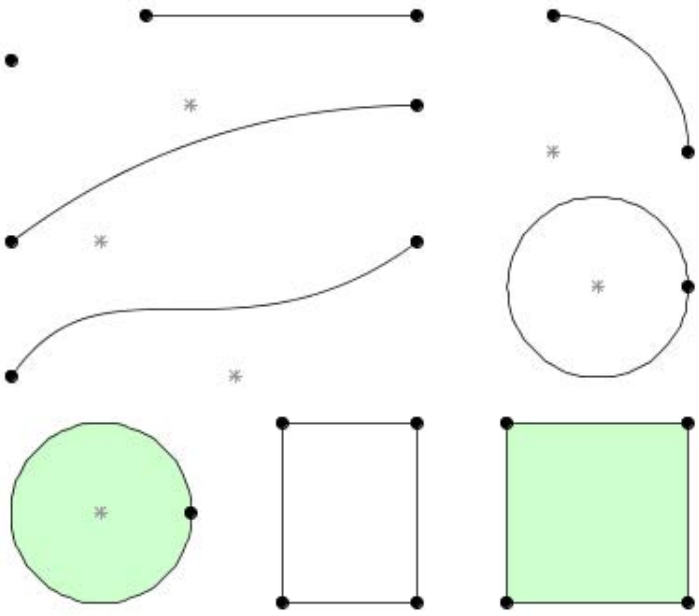


Figura 4.18: Primitivas geométricas.

As opções *Viga...*, *Pórtico...* e *Grelha...* estão detalhadas nas Figuras 4.19 a 4.21.

A Figura 4.19 apresenta o diálogo que auxilia na geração da geometria de vigas contínuas. Basta definir o número de vãos desejados e suas dimensões.

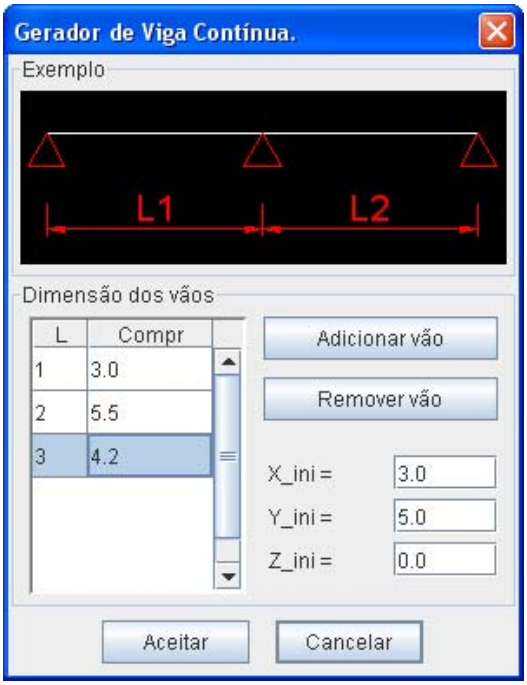


Figura 4.19: Diálogo para gerar geometria de vigas contínuas.

O gerador de pórticos (Figura 4.20) possibilita a geração rápida da geometria de pórticos planos ou espaciais.

Figura 4.20: Diálogo para gerar geometria de pórticos 2D ou 3D.

O gerador de grelhas (Figura 4.21) cria a geometria de uma grelha no plano XZ, formada apenas por arestas ou com a presença de faces. O botão *Atualizar Esp.* recalcula o espaçamento entre as barras após a definição do número de divisões.

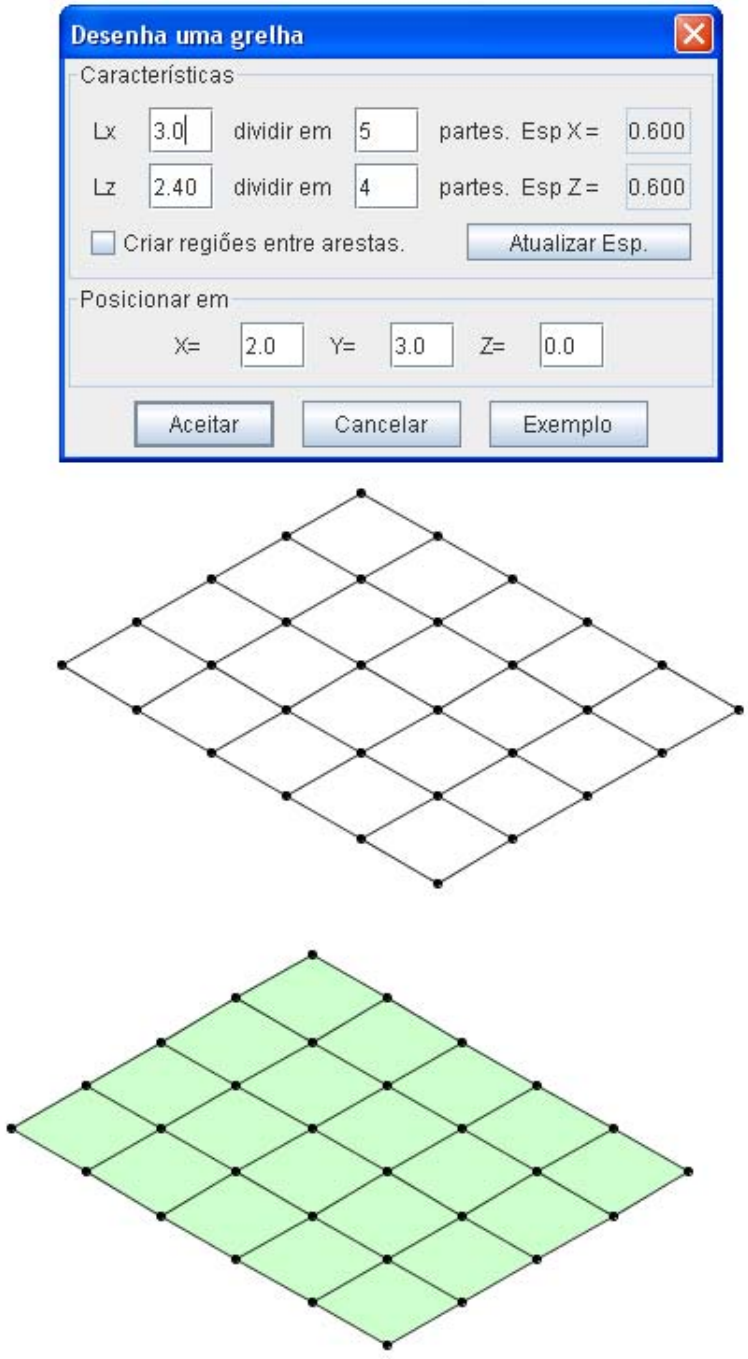


Figura 4.21: Diálogo para gerar geometria de grelhas.

No *Menu Modificar* (Figura 4.22) foram agrupados os comandos que permitem a edição do modelo geométrico. Detalhes de alguns dos comandos estão ilustrados nas Figuras 4.23 a 4.31.

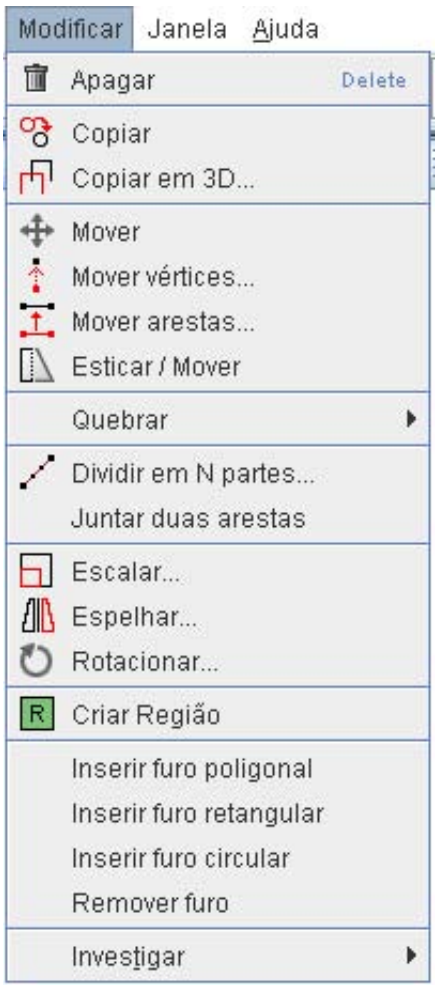


Figura 4.22: Menu Modificar.

O comando *Modificar* ≫ *Apagar* remove do desenho e da estrutura de dados do modelo os vértices, arestas e faces pré selecionados.

Para copiar objetos, pode ser usado o comando *Modificar* ≫ *Copiar* ou então o comando *Modificar* ≫ *Copiar em 3D....* O primeiro faz cópias no plano XY com interação através do dispositivo apontador (mouse). O segundo faz cópias paralelas com auxílio do diálogo *Cópia Paralela* (Figura 4.23).

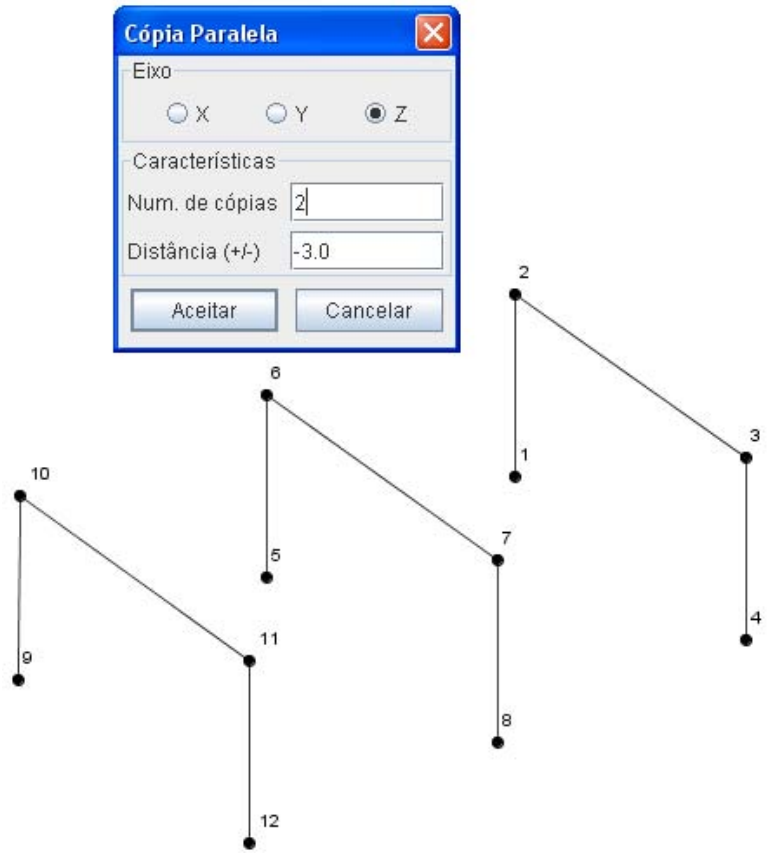


Figura 4.23: Diálogo para cópias paralelas.

Para mover objetos no plano XY, com interação via mouse, pode ser usado o comando *Modificar* $\gg$ *Mover* ou então o comando *Modificar* $\gg$ *Esticar/Mover*. Através das opções *Modificar* $\gg$ *Mover vértices...* (Figura 4.24) e *Modificar* $\gg$ *Mover arestas...* (Figura 4.25), os vértices e as arestas podem ser transladados em qualquer direção.

Figura 4.24: Diálogo *Mover Vértices*.

Figura 4.25: Diálogo *Mover Arestas*.

Com as opções *Modificar* ≫ *Quebrar* (Figura 4.26), as arestas podem ser "quebradas" em dois segmentos. A primeira opção permite separar uma aresta num vértice pré-desenhado sobre ela. A opção *Quebrar* ≫ *a aresta na posição indicada...* conta com o auxílio do diálogo mostrado na Figura 4.27. A terceira opção verifica se existe interseção entre duas arestas e quebra neste ponto, gerando quatro novas arestas.

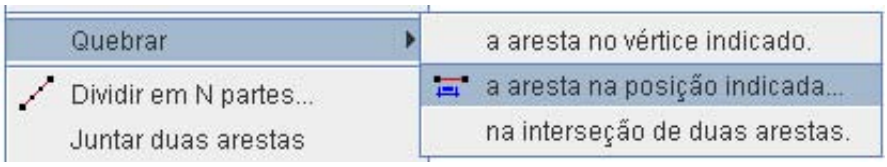


Figura 4.26: Opções do sub-menu *Quebrar*.

Figura 4.27: Diálogo *Quebrar a aresta na posição indicada.*

Os objetos pré-selecionados podem ser divididos em segmentos menores através da opção *Modificar* ≫ *Dividir em N partes....* A Figura 4.28 ilustra a divisão de uma linha e um arco. No caso das arestas que possuem uma curva (curva cúbica, curva quadrática, arco ou círculo), a forma da curva é mantida após a divisão, para que se avalie a aproximação dos segmentos.

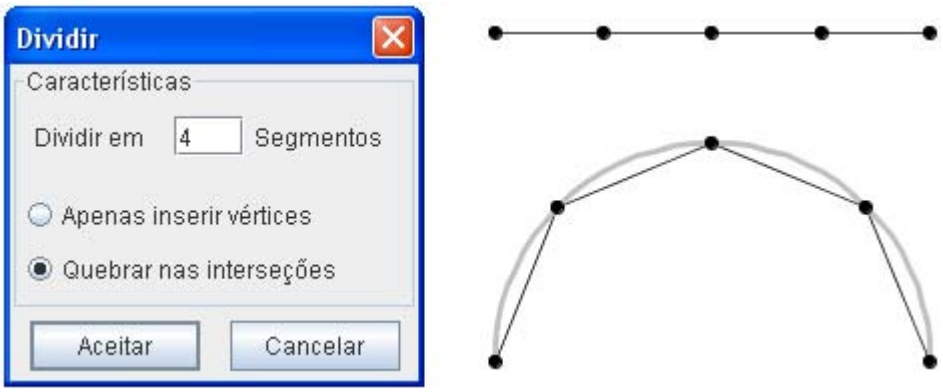


Figura 4.28: Diálogo *Dividir em N partes*.

A opção *Modificar* ≫ *Juntar duas arestas* (rever Figura 4.26), une dois segmentos de aresta no vértice comum.

A Figura 4.29 ilustra o comando *Modificar* ≫ *Escalar*. Pode-se aplicar um fator de escala para cada direção (X, Y ou Z) ou um fator uniforme. Um ponto base também pode ser especificado.

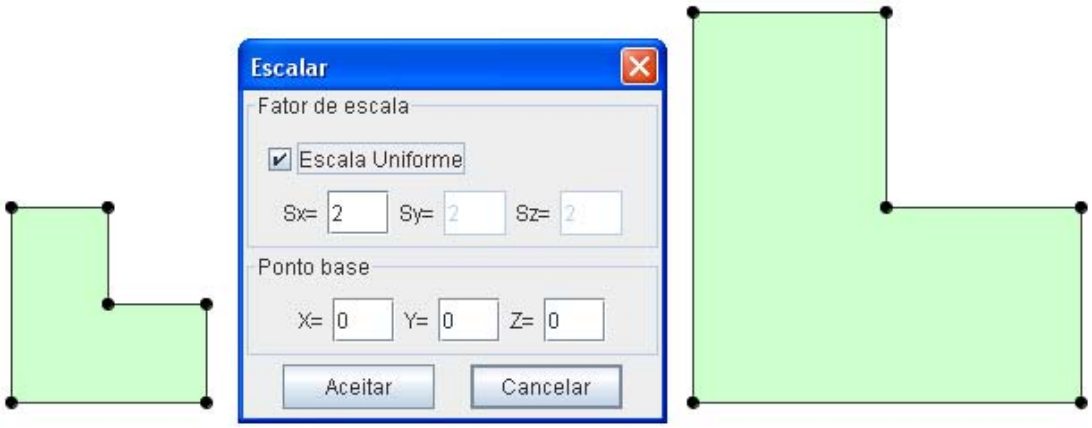


Figura 4.29: Diálogo *Escalar*.

Os objetos pré-selecionados podem ser espelhados em relação aos planos XY, XZ ou YZ. O espelho pode ser posicionado ao longo de um dos eixos (X, Y ou Z) conforme a escolha do plano. A Figura 4.30 ilustra o comando *Modificar* ≫ *Espelhar*.

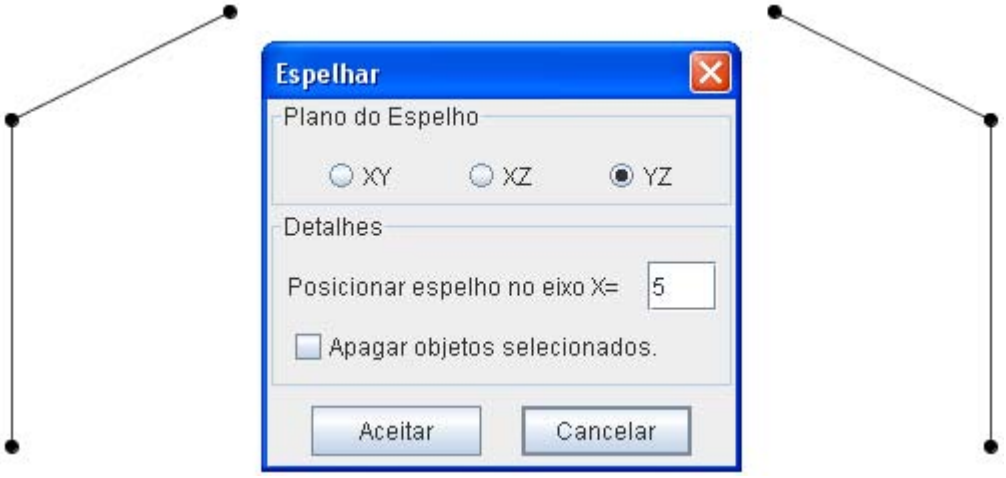


Figura 4.30: Diálogo *Espelhar*.

A Figura 4.31 ilustra o comando *Modificar* $\gg$ *Rotacionar*. Após a escolha do eixo de rotação (X, Y ou Z), define-se a posição (y, z), (x, z) ou (x, y) onde o eixo será fixado.

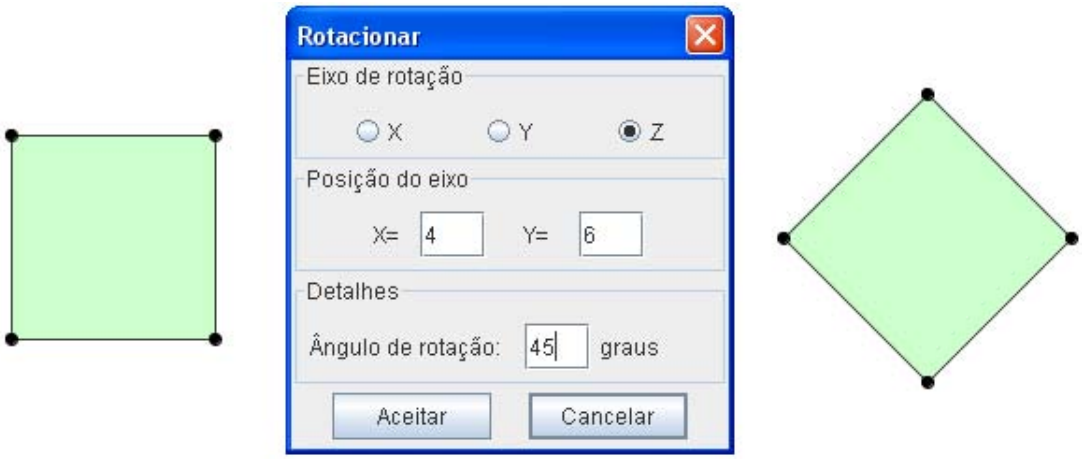


Figura 4.31: Diálogo *Rotacionar*.

Com a opção *Modificar* $\gg$ *Criar Região* (Figura 4.32) podem ser geradas regiões poligonais, após a seleção de arestas que definem um contorno fechado. A Figura 4.33 ilustra o comando.

Figura 4.32: Comando *Criar Região*.

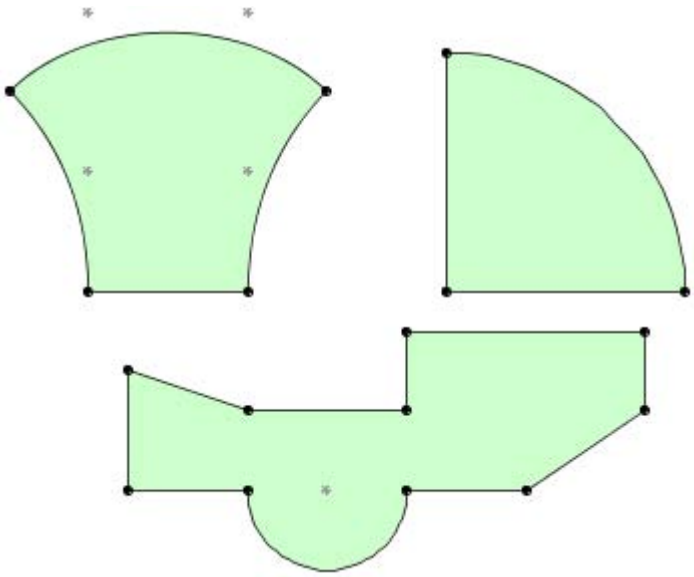


Figura 4.33: Exemplos de regiões poligonais.

Dentro das regiões também podem ser inseridos furos poligonais, retangulares ou circulares (Figura 4.34).

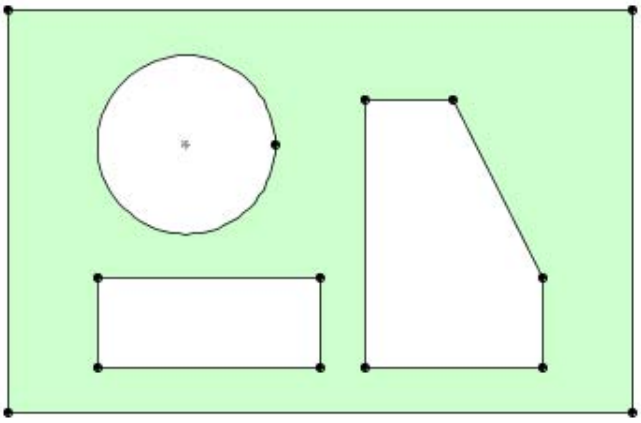


Figura 4.34: Exemplo de uma região com furos.

A opção *Modificar* ≫ *Remover furo* remove o furo selecionado e reconstitui a região (face) original.

A opção *Modificar* ≫ *Investigar* reúne comandos para investigação (Figura 4.35). Pode-se medir a distância entre dois pontos, obter informações sobre os vértices, arestas e faces selecionados ou então gerar uma listagem com os dados do modelo geométrico (Figura 4.36).

Figura 4.35: Comandos de investigação.

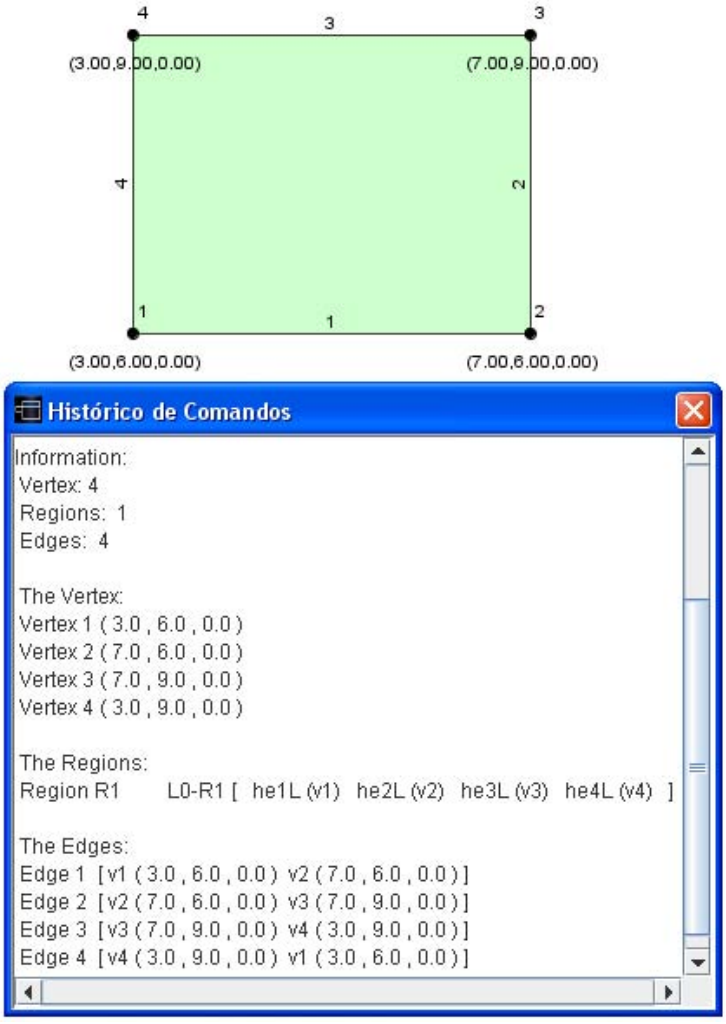


Figura 4.36: Listagem de dados do modelo.

Para ajudar na organização das vistas, inseridas na interface, foram incluídos no *Menu Janela* (Figura 4.37) as opções *Cascata* e *Lado a Lado*.

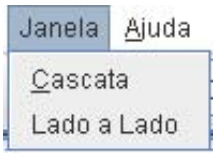


Figura 4.37: Menu Janela.

Para permitir a internacionalização da aplicação, todos os textos presentes nos menus e diálogos do pré-processador foram armazenados em um arquivo específico. Através do *Menu Ajuda* (Figura 4.38) é possível definir o idioma a ser utilizado na interface.

Figura 4.38: Menu Ajuda.

4.3 Mudança do Módulo Geometria para Malha

Na passagem do módulo **Geometria** para o módulo **Malha** é apresentado o diálogo *Parser: Geometria/Malha* para que se defina como o modelo de malha será montado (As diferentes formas de atuação do "parser" foram detalhadas no Capítulo 3). Caso o modelo geométrico possua regiões, é possível compor uma malha com elementos planos ou com elementos combinados (barras e planos). O "parser" *Geometria/Malha* faz uma cópia da estrutura de dados do modelo geométrico para o modelo de malha garantindo, deste modo, a independência entre eles. Conforme ilustra a Figura 4.39, é possível gerar vários modelos de malha para um mesmo modelo geométrico. Para montar um novo modelo de malha basta ativar a janela da geometria e clicar novamente no botão Malha.

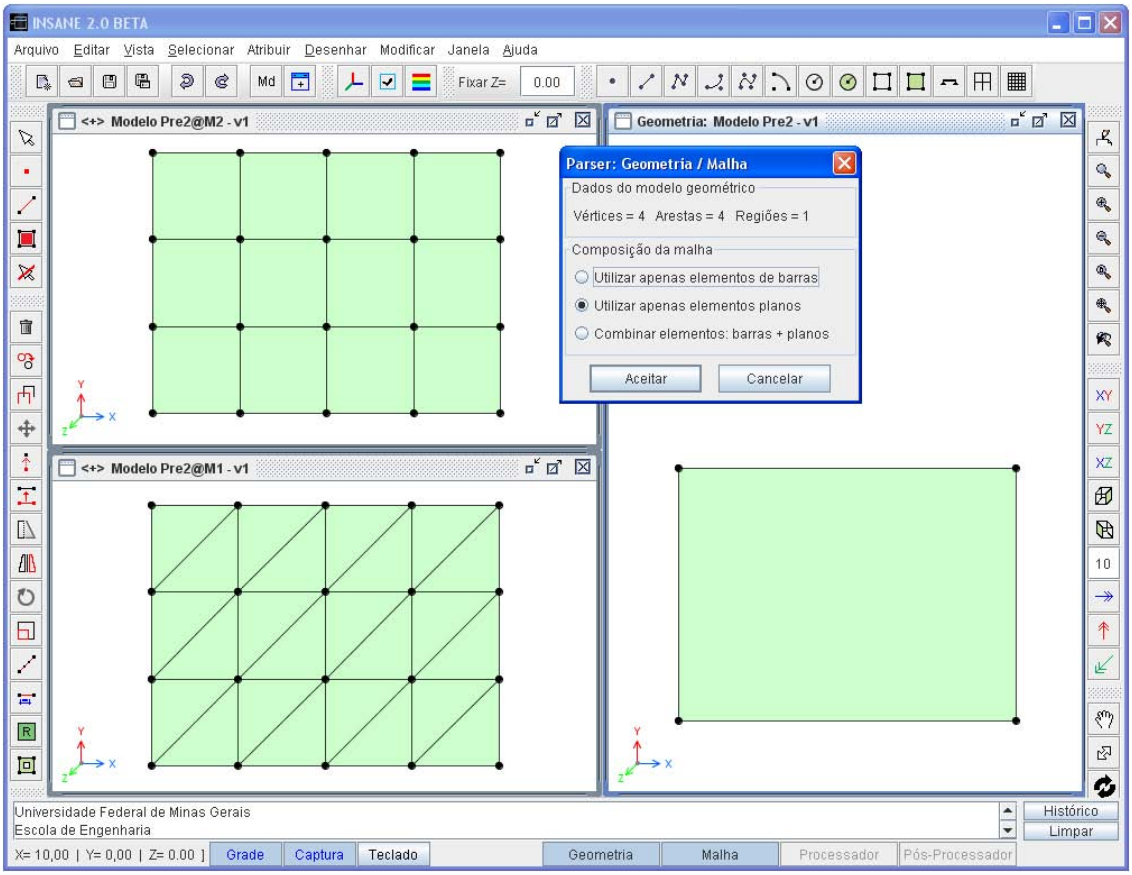


Figura 4.39: Exemplo de modelos de malha para uma mesma geometria.

4.4 Recursos do Módulo Malha

Os recursos do módulo malha foram separados para atender a duas etapas: geração da malha e definição de atributos. A Figura 4.40 apresenta o menu e a barra de ferramentas superior da etapa 1 (malha/geração).

Figura 4.40: Barra de menus e barra de ferramentas do módulo malha/discretização.

A Figura 4.41 apresenta as opções do *Menu Malha*.

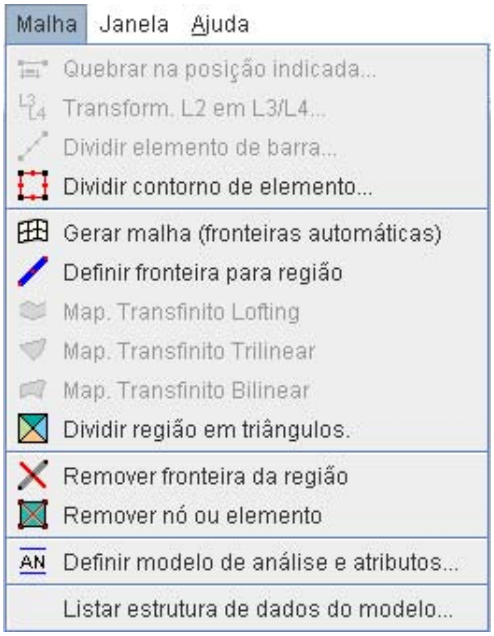


Figura 4.41: Opções do Menu Malha.

As quatro primeiras opções permitem a divisão de linhas e curvas para a discretização com elementos de barra ou para particionamento do contorno de regiões. A primeira opção (*Quebrar na posição indicada...*) já foi detalhada no módulo geometria. A Figura 4.42 ilustra o comando que transforma elementos de barra com 2 nós em elementos com 3 ou 4 nós.

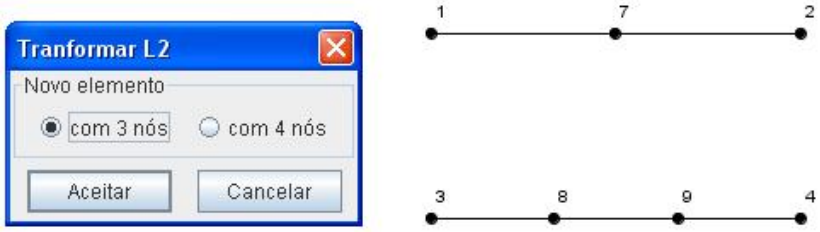


Figura 4.42: Criação de elementos de barra com 3 ou 4 nós.

A Figura 4.43 ilustra a divisão de uma curva em 4 e 6 partes. Cada segmento gerado passa a representar um elemento de barra com dois nós.

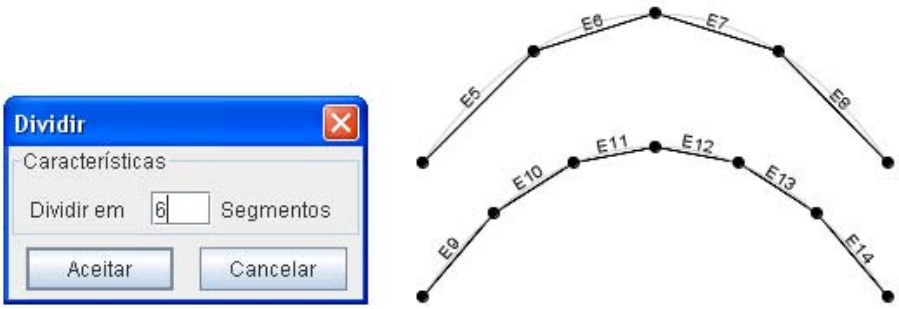


Figura 4.43: Divisão de curvas.

Para a geração de malhas, compostas por elementos paramétricos (Q4, Q8, Q9, T3, T6 ou T10), primeiro é necessário dividir o contorno da região onde será aplicado um dos métodos de Mapeamento Transfinito (Lofting, Trilinear ou Bilinear). A Figura 4.44 ilustra a preparação de uma região.

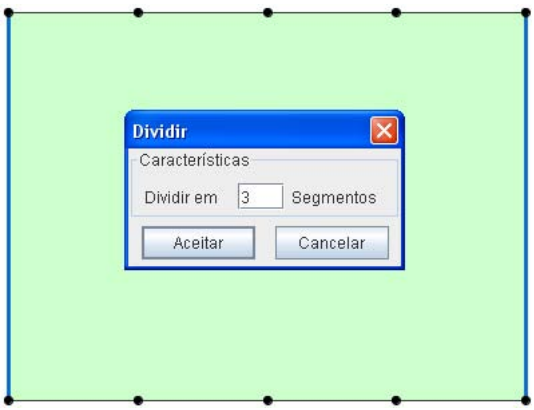


Figura 4.44: Divisão do contorno de uma região.

Caso seja criada no módulo geometria uma região formada por 3 ou 4 arestas, após a divisão do contorno pode-se aplicar a opção *Malha* ≫ *Gerar Malha*. A malha será gerada através de um mapeamento **Trilinear** (região com três fronteiras) ou **Bilinear** (região com quatro fronteiras). A Figura 4.45 ilustra essa opção e o diálogo para a escolha da forma do elemento.

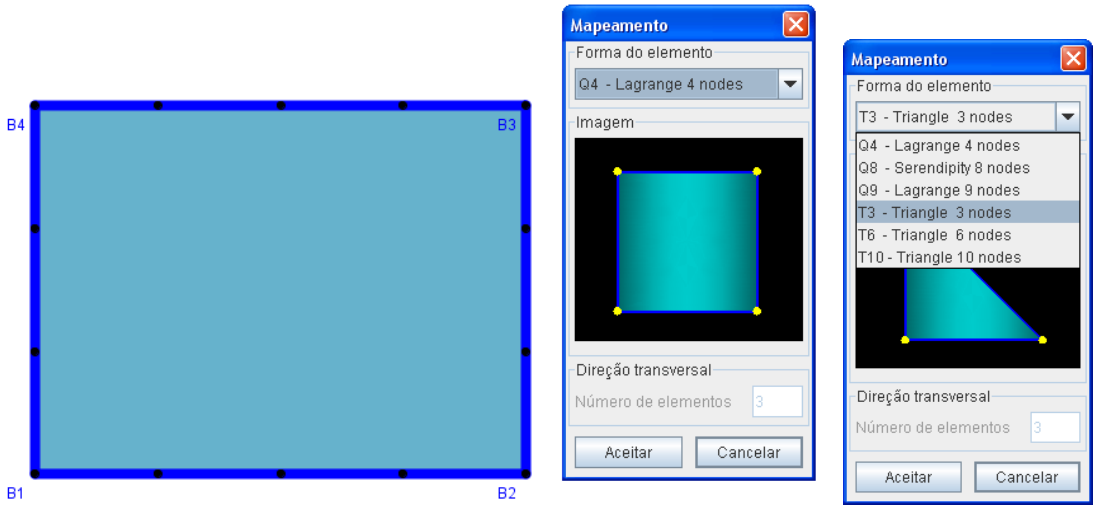


Figura 4.45: Geração automática de fronteiras e escolha do elemento.

Se a malha gerada não atender às expectativas, pode-se desfazer o comando (*Menu Edit* ≫ *Desfazer*) e acioná-lo novamente para escolha de outro elemento. As Figuras 4.46 e 4.47 ilustram uma malha com elementos Q4 e outra com T3.

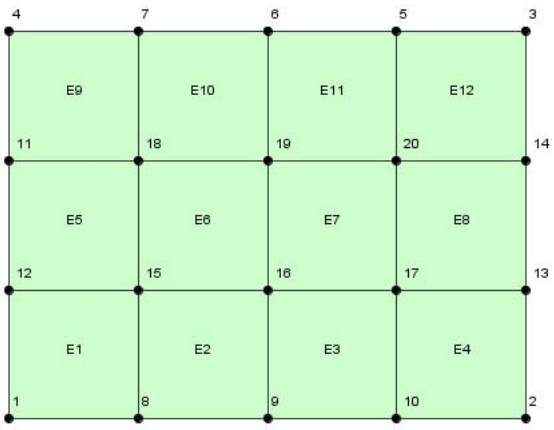


Figura 4.46: Mapeamento com Q4.

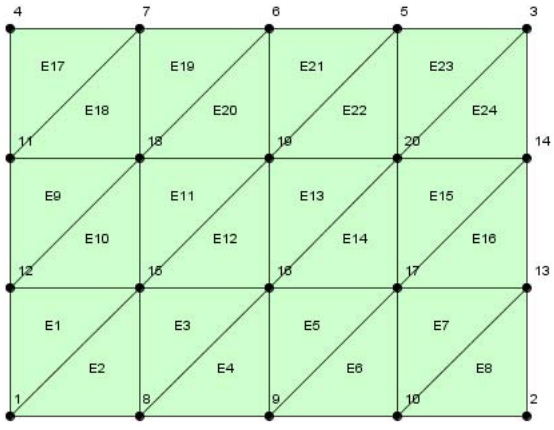


Figura 4.47: Mapeamento com T3.

Para uma região formada originalmente por mais de 4 arestas é necessário dividir o contorno e agrupar os novos segmentos em duas, três ou quatro fronteiras. Desta forma é possível aplicar um dos tipos de Mapeamento Transfinito. A Figura 4.48 mostra a seqüencia de operações possíveis para a geração de uma malha.

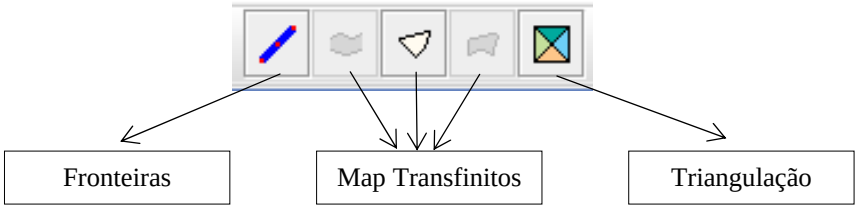


Figura 4.48: ToolBarMapping

Acionando a opção *Malha* ≫ *Definir fronteira...*, antes de selecionar uma região, um diálogo explica as diferentes maneiras de se definir uma fronteira (Figura 4.49).

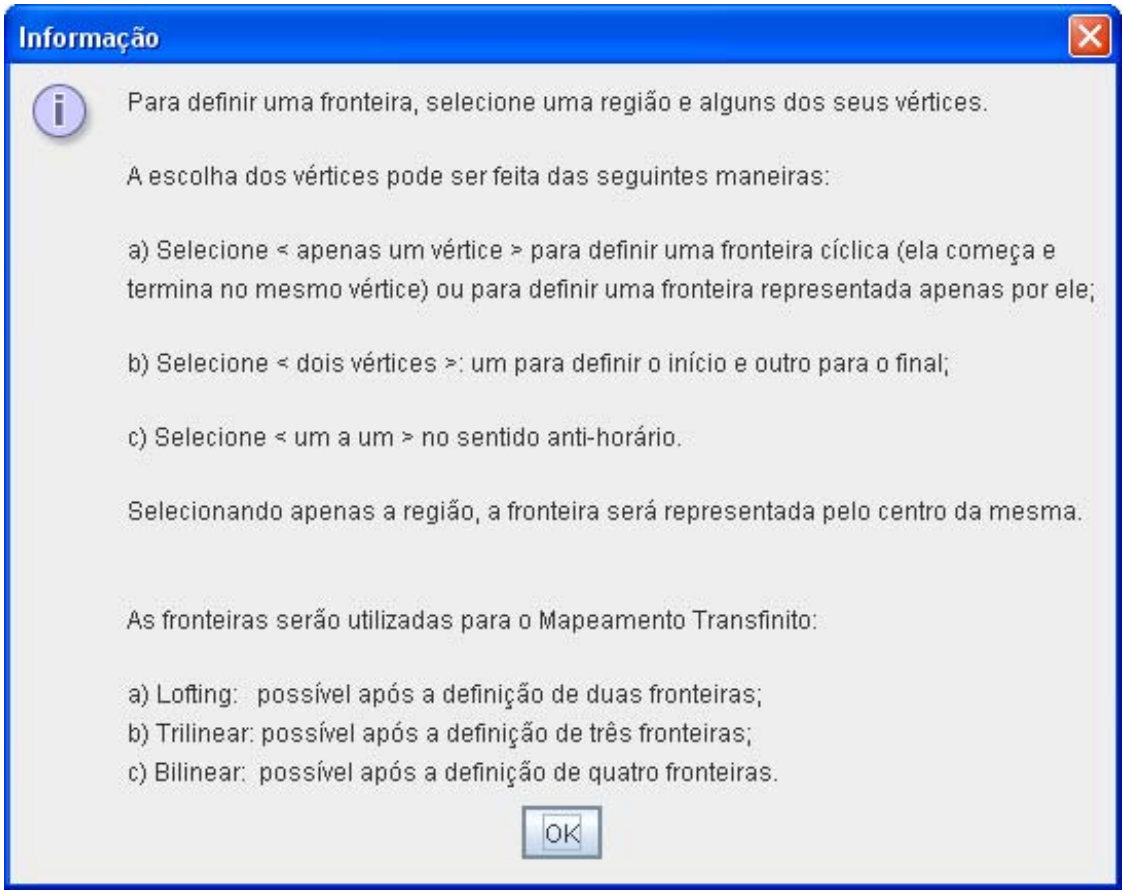


Figura 4.49: Informações sobre definição de fronteiras em uma região.

A Figura 4.50 ilustra a opção *Malha* ≫ *Dividir região em triângulos* que permite gerar elementos do tipo T3 em relação ao centro de uma região ou em relação ao centro de elementos do tipo Q4. Também é possível editar a malha com a opção *Malha* ≫ *Remover nó ou elemento*.

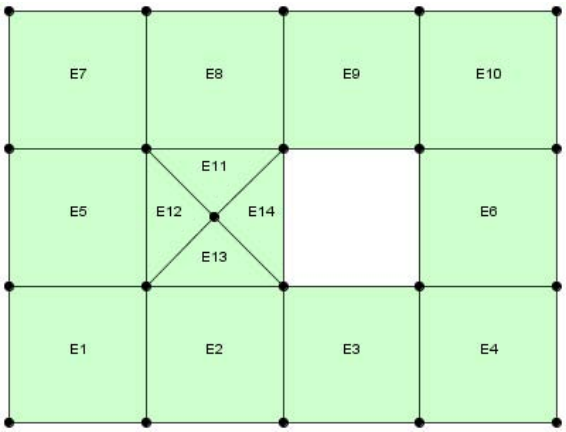


Figura 4.50: Subdivisão com T3 e remoção de elemento.

Após a geração da malha, através da opção *Malha* ≫ *Definir modelo de análise e atributos...*, é apresentado o diálogo **Modelo de Análise** (Figura 4.51).

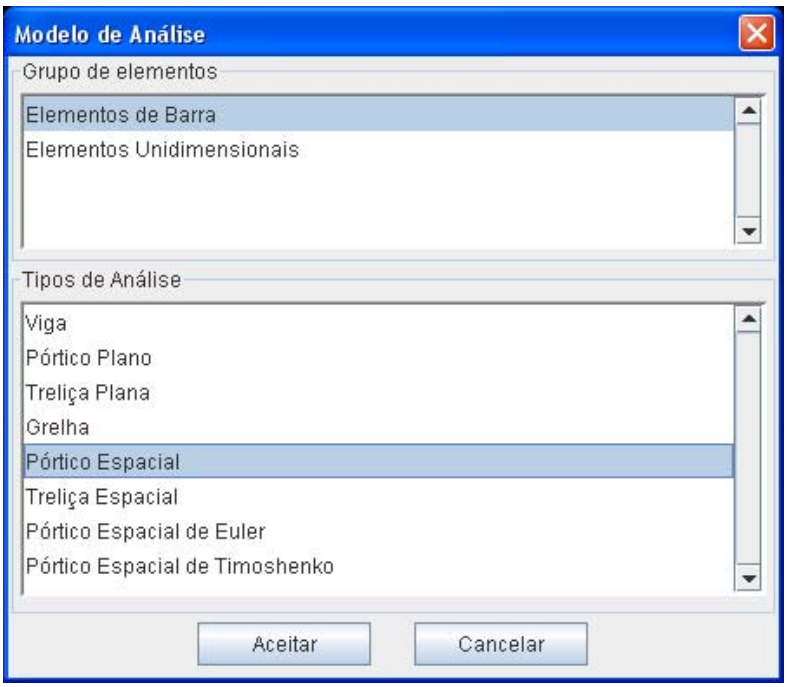


Figura 4.51: Diálogo Modelo de Análise.

Os tipos de análise que são carregados na inicialização do diálogo dependem de como a malha foi composta: por elementos combinados (Tabela 4.1), apenas por elementos de barra (Tabela 4.2) ou apenas por elementos planos (Tabela 4.3).

Tabela 4.1: Tipos de análise para malhas formadas por elementos combinados.

Num.	**Tipo de Análise**
C1	Pórtico Espacial de Timoshenko + Placa Espessa de Reissner Mindlin
C2	Pórtico Espacial de Euler + Placa Fina de Kirchhoff com elemento MZC
C3	Pórtico Plano + Estado Plano de Tensão

Tabela 4.2: Tipos de análise para malhas formadas por elementos de barra.

Num.	**Tipo de Análise**
B1	Grelha
B2	Linha Reta 1D
B3	Linha Reta 2D
B4	Linha Reta 3D
B5	Pórtico Espacial
B6	Pórtico Espacial de Euler
B7	Pórtico Espacial de Timoshenko
B8	Pórtico Plano
B9	Treliça Plana
B10	Treliça Espacial
B11	Viga

Tabela 4.3: Tipos de análise para malhas formadas por elementos planos.

Num.	**Tipo de Análise**
P1	Axissimétrico
P2	Estado Plano de Deformação
P3	Estado Plano de Deformação em Meio Contínuo Micropolar
P4	Estado Plano de Tensão
P5	Estado Plano de Tensão em Meio Contínuo Micropolar
P6	Placa Fina de Kirchhoff com elemento MZC e Integração Numérica
P7	Placa Fina de Kirchhoff com elemento MZC e Integração Analítica
P8	Placa Fina de Kirchhoff com elemento BFS e Integração Numérica
P9	Placa Fina de Kirchhoff com elemento BFS e Integração Analítica
P10	Placa Fina de Kirchhoff com elemento CKZ e Integração Numérica
P11	Placa Fina de Kirchhoff com elemento CKZ e Integração Analítica
P12	Placa Fina de Kirchhoff com elemento Cowper e Integração Numérica
P13	Placa Fina de Kirchhoff com elemento Cowper e Integração Analítica
P14	Placa Espessa de Reissner Mindlin
P15	Placa Fina de Reissner Mindlin com Integração Reduzida
P16	Placa Fina de Reissner Mindlin com Integração Seletiva
P17	Placa Fina de Reissner Mindlin com Deformação de Cisalhamento Imposta
P18	Placa Fina de Reissner Mindlin com Heterosis
P19	Placa Fina de Reissner Mindlin com Heterosis e Integração Reduzida

A Tabela 4.4 apresenta as classes que foram criadas no **INSANE** para representar os **Modelos de Análise** (projeto *br.ufmg.dees.insane.analysisModel*). Quando um tipo de análise é escolhido pelo usuário, o pré-processador, através da classe *MeshAnalysisModel* (detalhada no Apêndice C.3), cria uma instância de uma dessas classes para definir o modelo de análise dos elementos.

Tabela 4.4: Modelos de Análise do **INSANE**.

Num.	**Modelo de Análise**	**Graus de Liberdade**
1	Axisymmetric	Dx, Dy
2	Beam	Dy, Rz
3	BFSKirchhoffPlate	Dz, Rx, Ry, Wxy
4	CKZKirchhoffPlate	Dz, Rx, Ry
5	CowperKirchhoffPlate	Dz, Rx, Ry, Wxx, Wxy, Wyy
6	EulerSpaceFrame	Dx, Dy, Dz, Rx, Ry, Rz
7	Grid	Dy, Ry, Rz
8	Heterosis	Dz, Rx, Ry
9	Line_1D	Dx
10	Line_2D	Dx, Dy
11	Line_3D	Dx, Dy, Dz
12	MZCKirchhoffPlate	Dz, Rx, Ry
13	PlaneFrame	Dx, Dy, Rz
14	PlaneStrain	Dx, Dy
15	PlaneStress	Dx, Dy
16	PlaneTruss	Dx, Dy
17	PolarPlaneStrain	Dx, Dy, Mrz
18	PolarPlaneStress	Dx, Dy, Mrz
19	ReissnerMindlinPlate	Dz, Rx, Ry
20	SpaceFrame	Dx, Dy, Dz, Rx, Ry, Rz
21	SpaceTruss	Dx, Dy, Dz
22	TimoSpaceFrame	Dx, Dy, Dz, Rx, Ry, Rz

Após a definição do tipo de análise, os menus e as barras de ferramentas são alterados (Figura 4.52) e outros recursos são disponibilizados para a definição de atributos. A Figura 4.53 apresenta os recursos do *Menu Modelo*.

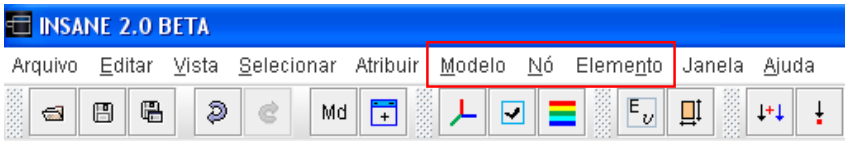


Figura 4.52: Menus do módulo malha/atributos.

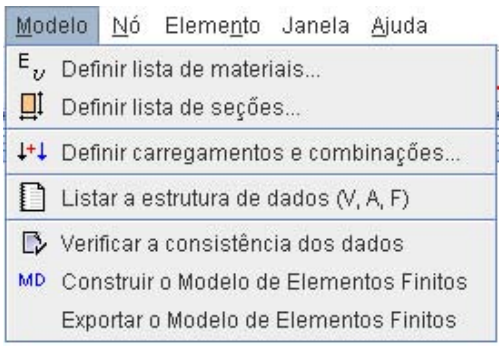


Figura 4.53: Menu Modelo

A opção *Modelo* ≫ *Definir lista de materiais...* abre o diálogo ilustrado na Figura 4.54. Os materiais disponíveis para uso dependem do modelo de análise.

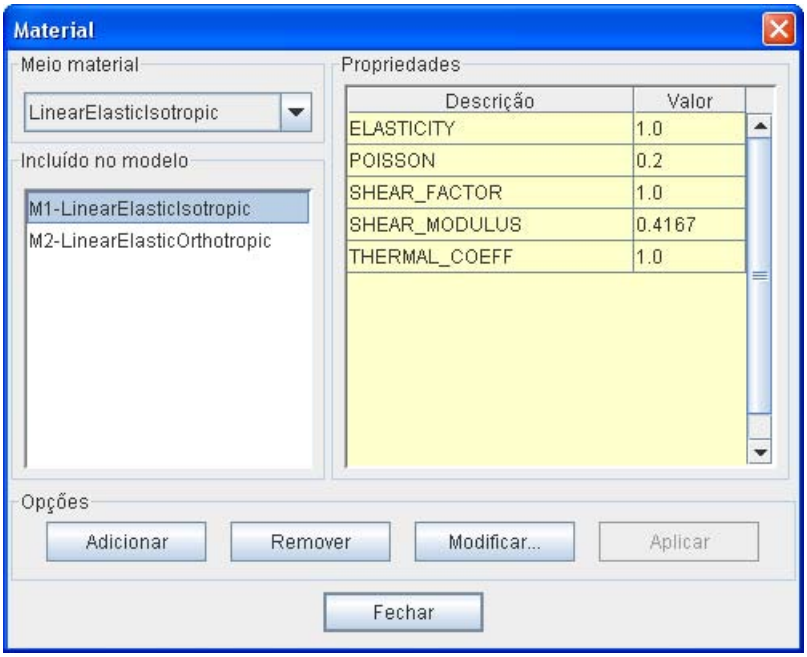


Figura 4.54: Diálogo para definição de materiais.

A opção *Modelo* ≫ *Definir lista de seções...* abre um diálogo (Figura 4.55) para

a definição das seções transversais que serão usadas no modelo. Basta escolher um dos tipos e clicar no botão *Adicionar*.

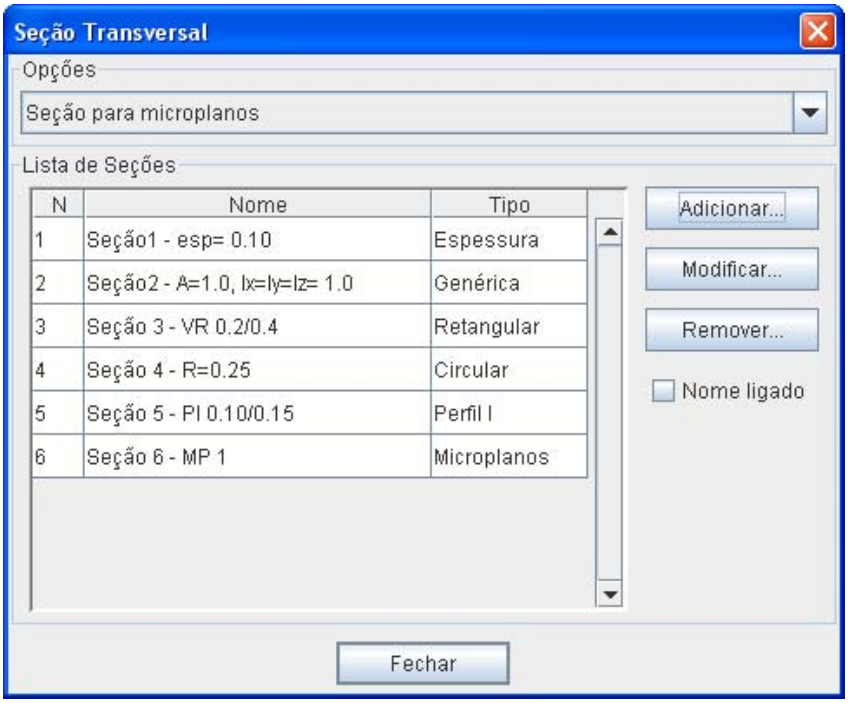


Figura 4.55: Diálogo para definição de seções transversais.

Cada tipo de seção possui um diálogo específico para a definição das propriedades e associação com um material previamente cadastrado. As Figuras 4.56 e 4.57 ilustram os diálogos para seções genéricas e retangulares.

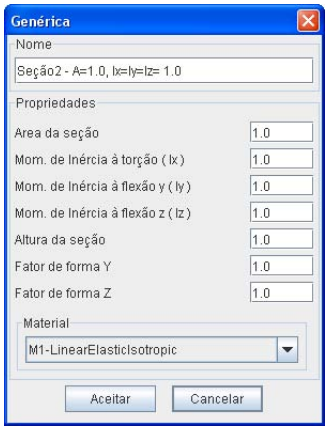


Figura 4.56: Seção genérica.

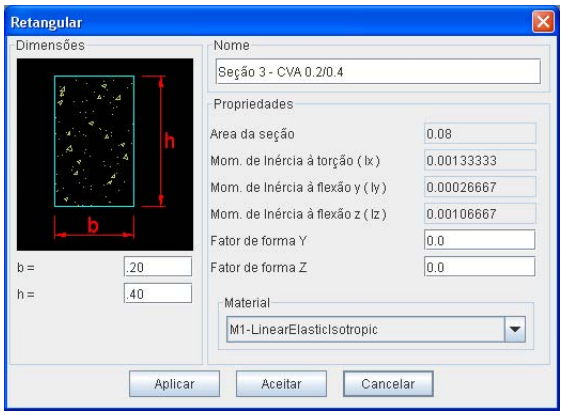


Figura 4.57: Seção Retangular

As Figuras 4.58 à 4.61 ilustram os diálogos para os outros tipos de seção.

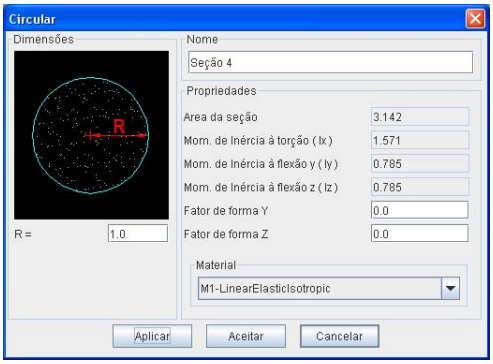


Figura 4.58: Seção Circular.

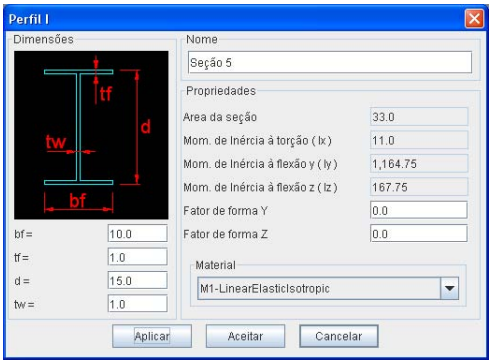


Figura 4.59: Seção Perfil I.

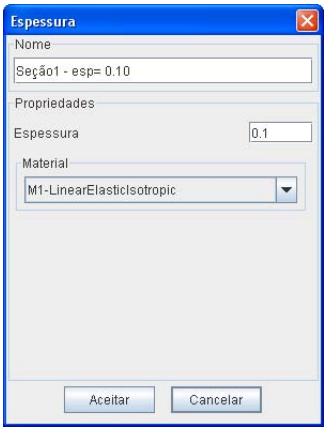


Figura 4.60: Seção Espessura.

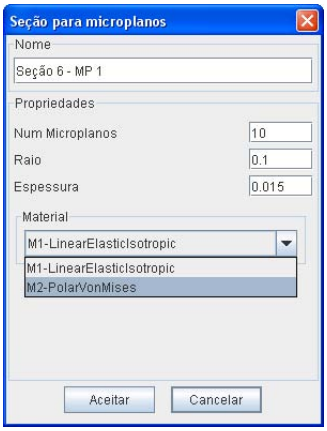


Figura 4.61: Seção para Microplanos.

Pode-se também ativar o campo "*Nome ligado*" (Figura 4.55) para identificar as seções que forem atribuídas a cada elemento, como mostra a Figura 4.62.

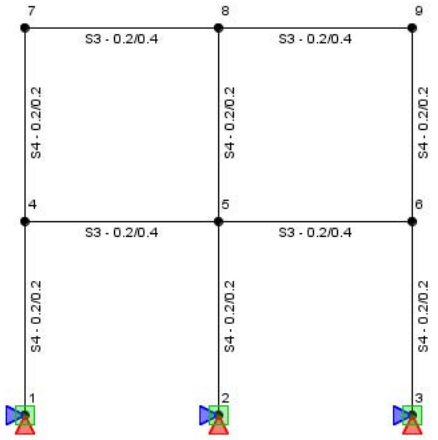


Figura 4.62: Identificação das seções atribuídas aos elementos.

A opção *Modelo* ≫ *Definir carregamentos e combinações...* abre o diálogo ilustrado na Figura 4.63. Através dele pode-se incluir vários casos de carregamento e montar as combinações. A Figura 4.64 detalha a adição de uma combinação.

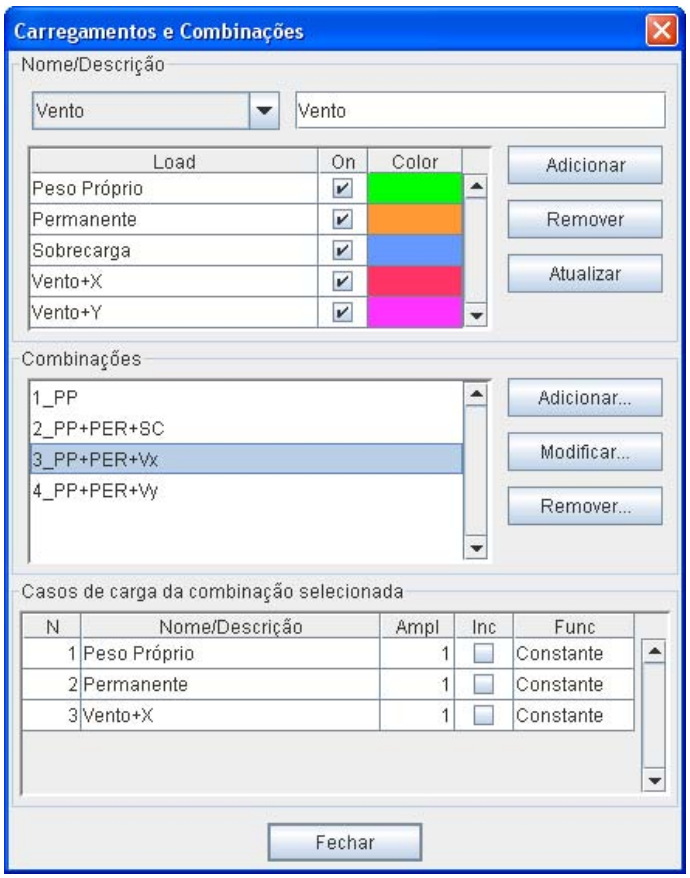


Figura 4.63: Diálogo para definição de carregamentos e combinações.

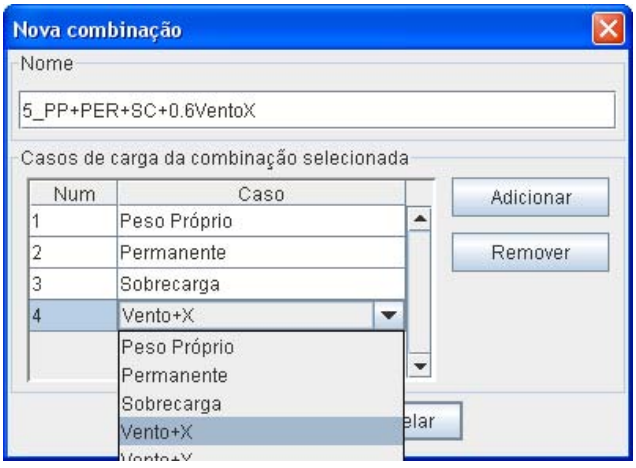


Figura 4.64: Diálogo para adição de combinações.

Uma função escalar pode ser atribuída a um determinado caso de carga. Para isso, basta selecionar uma combinação, clicar no caso desejado e ajustar parâmetros conforme o tipo da função escolhida (Figura 4.65).

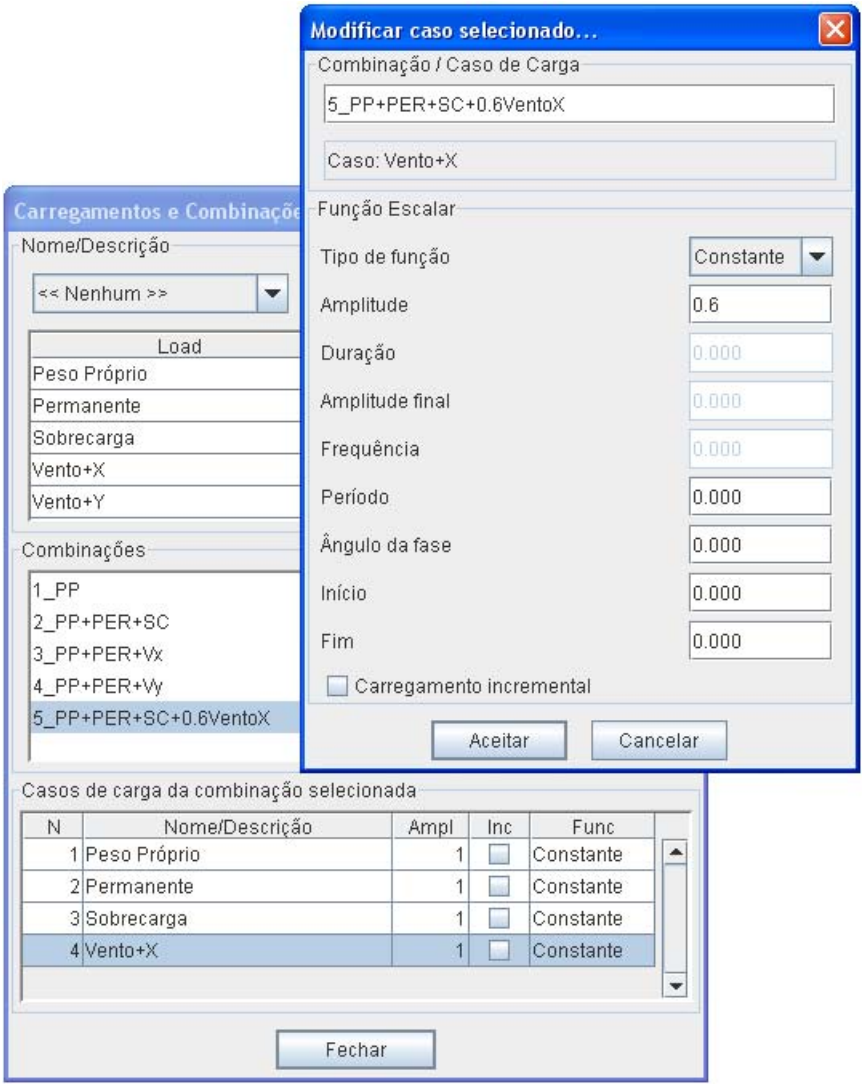


Figura 4.65: Diálogo para definição de uma função escalar.

A Figura 4.66 destaca o *Menu Nó* e suas opções.

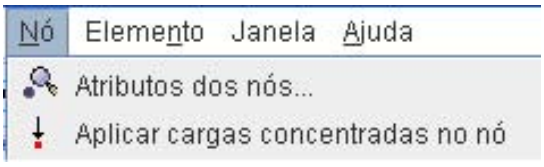


Figura 4.66: Menu Nó.

Para definição de atributos nodais, o diálogo ilustrado na Figura 4.67 reúne as opções: restrições, mola, massa, amortecimento, deslocamentos prescritos e ângulo. Os valores definidos são aplicados nos nós previamente selecionados.

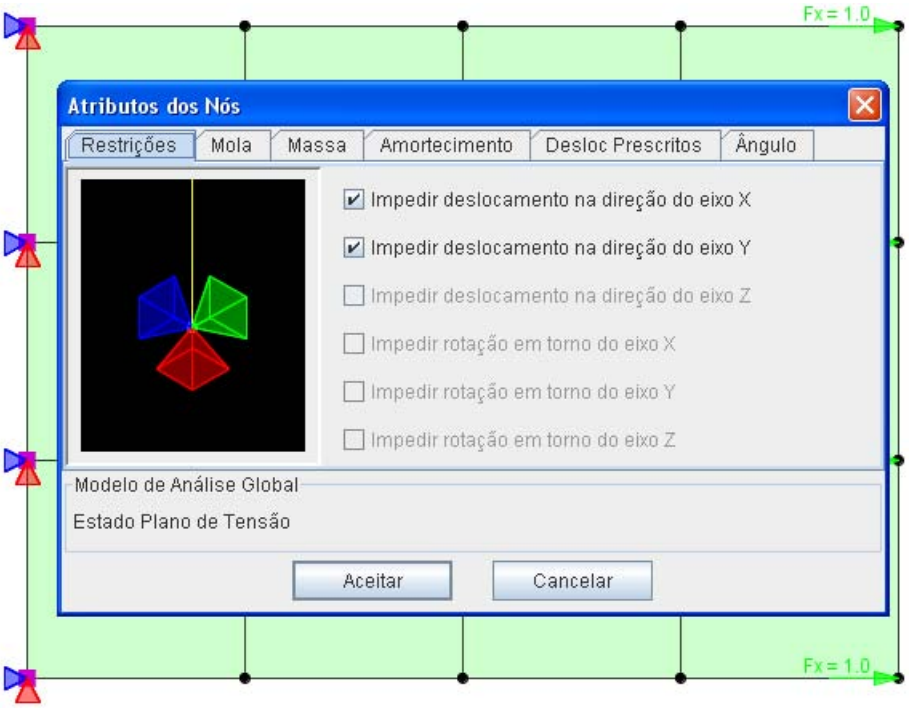


Figura 4.67: Diálogo para definição de atributos nodais.

Para aplicar forças nos nós selecionados, utiliza-se um diálogo onde são apresentados todos os casos de carga previamente cadastrados (Figura 4.68). Os campos (F_x, F_y,...M_z) são liberados conforme o modelo de análise adotado.

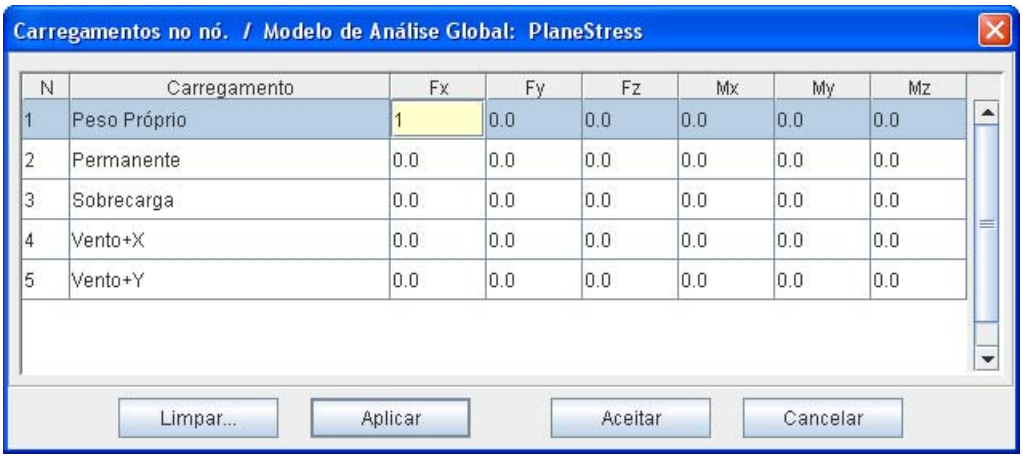


Figura 4.68: Diálogo para definição de cargas nodais.

A Figura 4.69 destaca o *Menu Elemento* e suas opções.

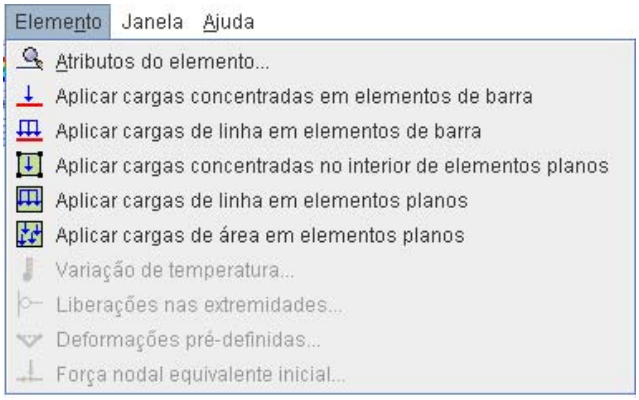


Figura 4.69: Menu Elemento

Para os elementos previamente selecionados, o diálogo ilustrado na Figura 4.70 permite definir uma seção transversal e a ordem de integração. Também mostra o material, o modelo constitutivo, o modelo de análise e o controlador do problema, associados a esses elementos.

Figura 4.70: Diálogo para definição de atributos nos elementos.

As Figuras 4.71 a 4.76 apresentam os diálogos que foram criados para a aplicação de carregamentos nos elementos pré-selecionados.

A Figura 4.71 ilustra o diálogo para definição de cargas pontuais em elementos de barra. Os casos de carregamento previamente criados são apresentados e vários pontos de carga podem ser adicionados para o carregamento selecionado. O botão *"X,Y,Z..."* abre um diálogo (Figura 4.72) para alterar a posição da carga selecionada.

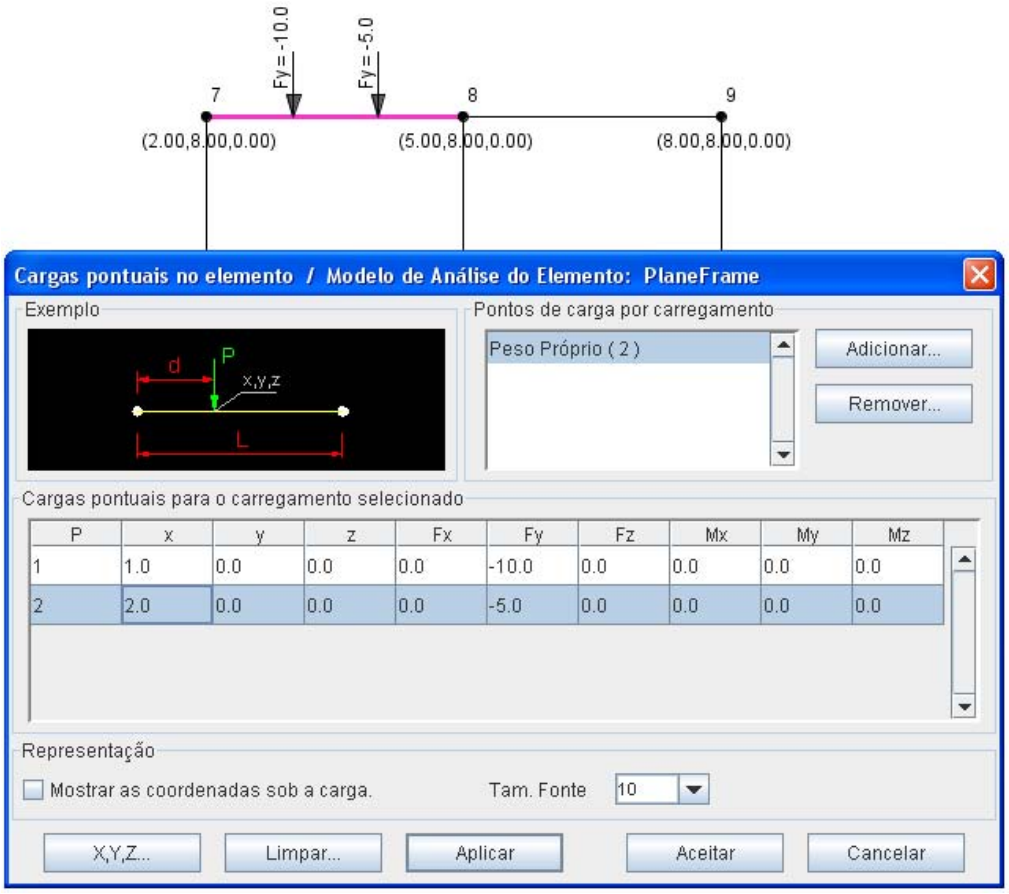


Figura 4.71: Diálogo para definição de carga pontual em elemento de barra.

Figura 4.72: Diálogo para ajustar a posição da carga.

A Figura 4.73 ilustra o diálogo para definição de cargas de linha em elementos de barra. São apresentados os casos de carregamento previamente criados e várias linhas de carga podem ser adicionadas para um mesmo caso. Através do botão *"Adicionar carga pontual"* é possível incluir pontos para compor a linha de carga. O botão *"X,Y,Z..."* abre um diálogo (Figura 4.72) para alterar a posição da carga selecionada.

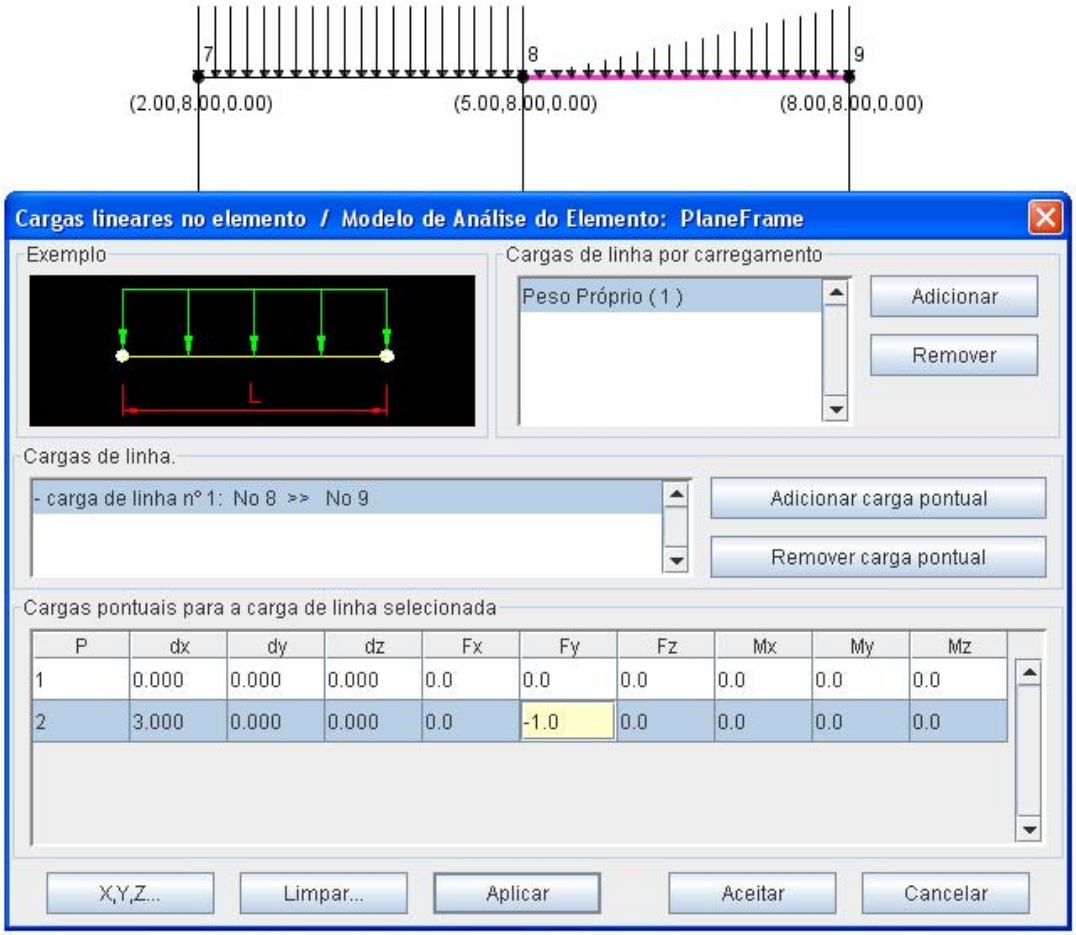


Figura 4.73: Diálogo para definição de carga de linha em elemento de barra.

A Figura 4.74 ilustra o diálogo para definição de cargas pontuais em elementos planos. Vários pontos de carga podem ser adicionados para o carregamento selecionado. Os valores das coordenadas x, y e z podem ser editados. Após o acionamento do botão *"Aplicar"*, os pontos fornecidos são verificados em relação aos limites e ao plano do elemento. Se a opção *"Mostrar as coordenadas sob a carga"* estiver marcada, serão desenhadas as coordenadas de cada ponto.

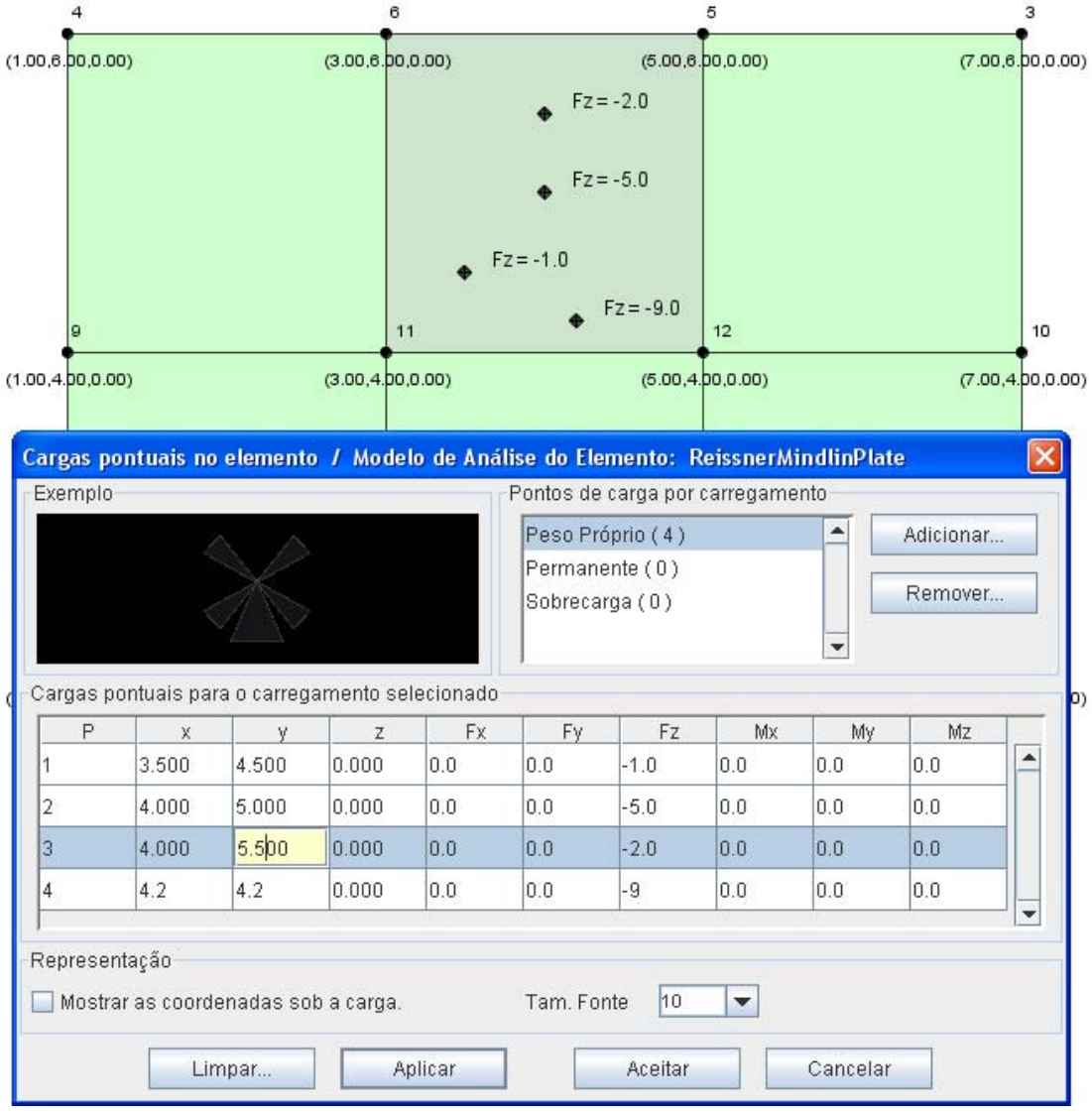


Figura 4.74: Diálogo para definição de carga pontual em elemento plano.

A Figura 4.75 ilustra o diálogo para definição de cargas de linha em elementos planos. Com a opção *"No contorno"* marcada, após o acionamento do botão *"Adicionar carga pontual"*, será apresentado na tabela um ponto de carga para cada vértice do elemento e a carga será aplicada entre os nós escolhidos. Caso a opção *"No contorno"* esteja desativada, poderão ser definidos pontos dentro do elemento e os campos das coordenadas ficam liberados para edição. Neste caso, o carregamento é aplicado apenas no primeiro elemento que foi selecionado.

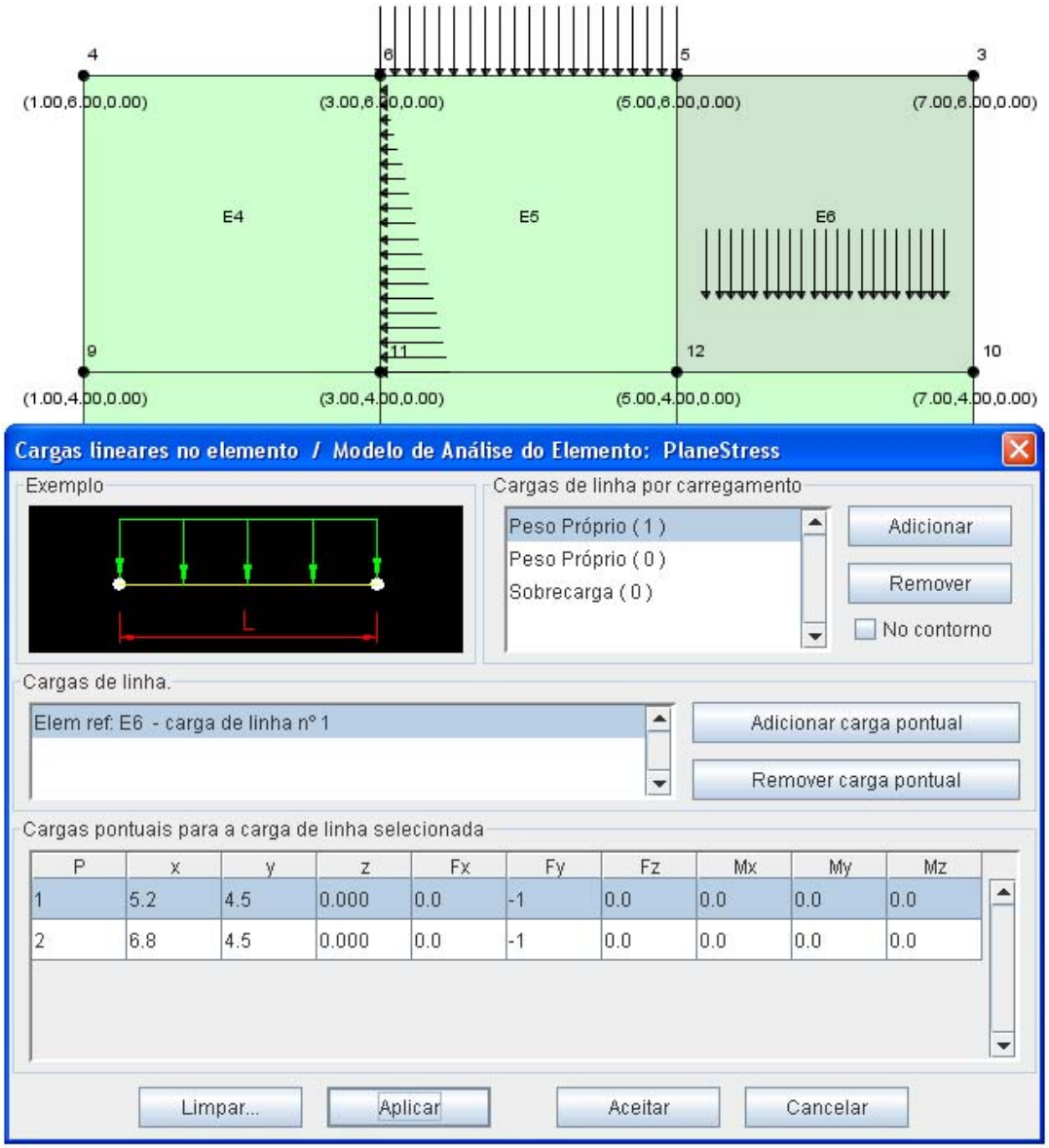


Figura 4.75: Diálogo para definição de carga de linha em elemento plano.

A Figura 4.76 ilustra o diálogo para definição de carga de área. Ativando-se a opção *"Uniforme"*, a carga é lançada em todo o elemento. São apresentadas as coordenadas do centro do primeiro elemento selecionado para a definição das forças. Os outros elementos pré-selecionados também receberão o mesmo carregamento. Caso a opção *"Uniforme"* esteja desativada, poderá ser definida uma área qualquer dentro do elemento (campos das coordenadas ficam liberados para edição). O carregamento é aplicado apenas no primeiro elemento que foi selecionado. Se a opção *"Mostrar as coordenadas sob a carga"* estiver marcada, serão desenhadas setas e coordenadas apenas nos nós do elemento.

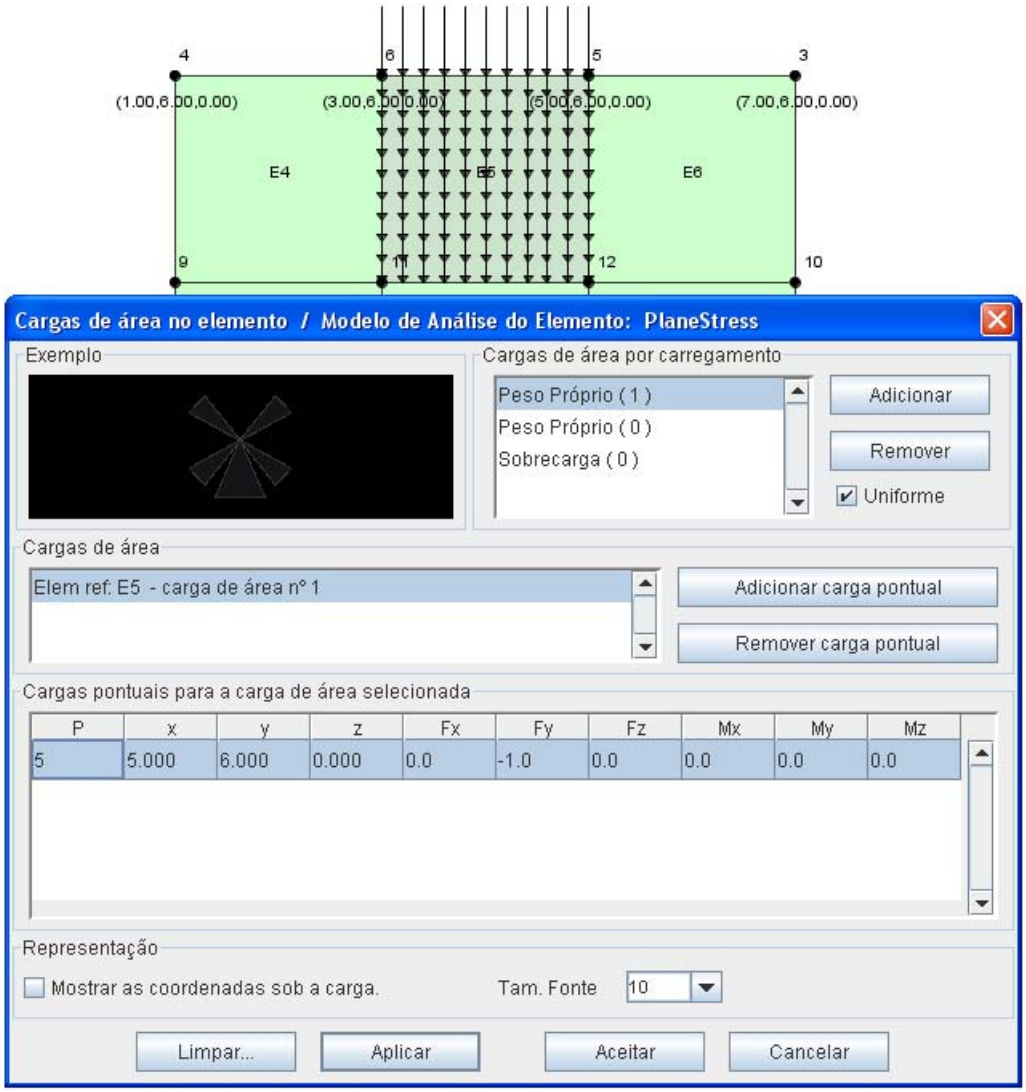


Figura 4.76: Diálogo para definição de carga de área em elemento plano.

Para atributos exclusivos dos elementos tipo **Framed**, foi criado um diálogo específico. As Figuras 4.77 e 4.78 apresentam as guias para definição de variação de temperatura e liberação nas extremidades, respectivamente.

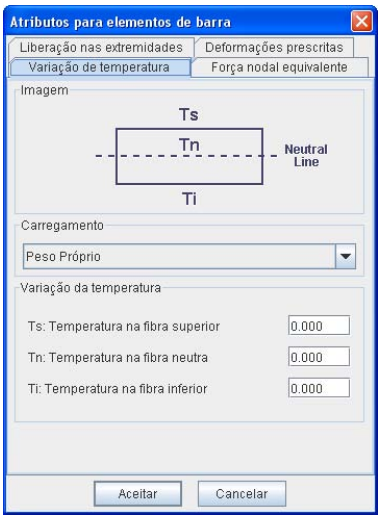


Figura 4.77: Diálogo Variação de temperatura.

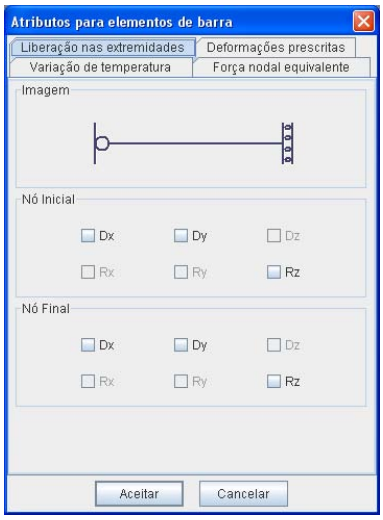


Figura 4.78: Diálogo Liberação nas extremidades.

As Figuras 4.79 e 4.80 apresentam as guias para definição de deformações prescritas e força nodal equivalente, respectivamente.

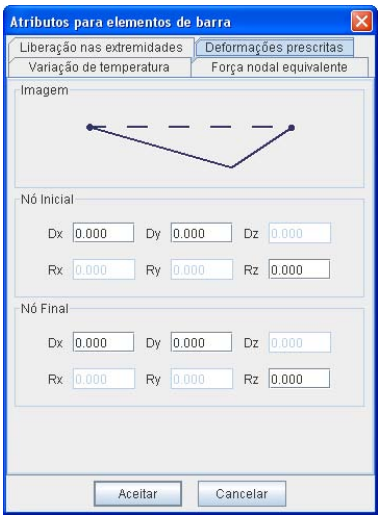


Figura 4.79: Diálogo Deformações prescritas.

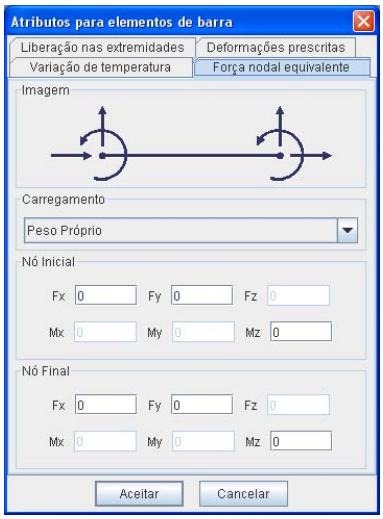


Figura 4.80: Diálogo Carregamento nodal equivalente.

4.5 Criação do Modelo de Elementos Finitos

Uma vez concluída a etapa de definição dos atributos, através da opção *Modelo* ≫ *Construir o Modelo de Elementos Finitos* (rever Figura 4.53), entra em ação outro "parser" chamado *MeshToFem*. Ele é o responsável pela criação de um modelo de elementos finitos (**FEM Model**) a partir das informações presentes no modelo de malha. A Figura 4.81 ilustra a transição entre os modelos.

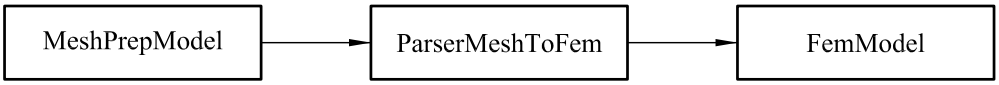


Figura 4.81: Transição entre Mesh Model e FEM Model.

Na falta de dados, um diálogo é apresentado (Figura 4.82) avisando sobre o número de erros encontrados.

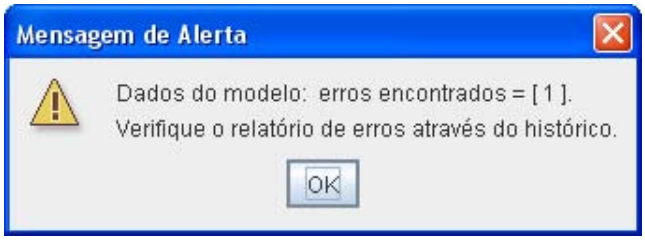


Figura 4.82: Diálogo para informar erros no modelo.

Caso todos os dados presentes no *MeshModel* sejam suficientes para a atuação do "parser", o *FemModel* é criado e o diálogo da Figura 4.83 é apresentado.

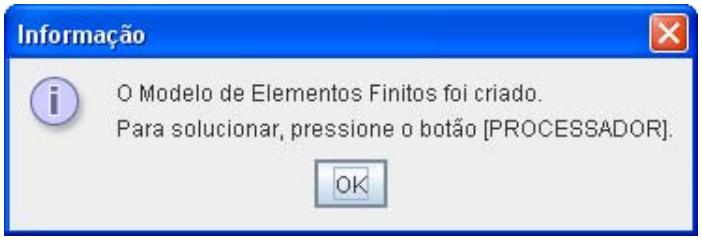


Figura 4.83: Diálogo para confirmar a criação do FEM Model.

4.6 Processamento

Após a criação do modelo de elementos finitos, clicando sobre do botão **Processador** (localizado na barra de status da interface), o pré-processador exibe o diálogo *Solução* (Figura 4.84). Através dele o usuário define o **Tipo de Processador** e o **Tipo de Análise/Solução** que serão empregados para solucionar o modelo.

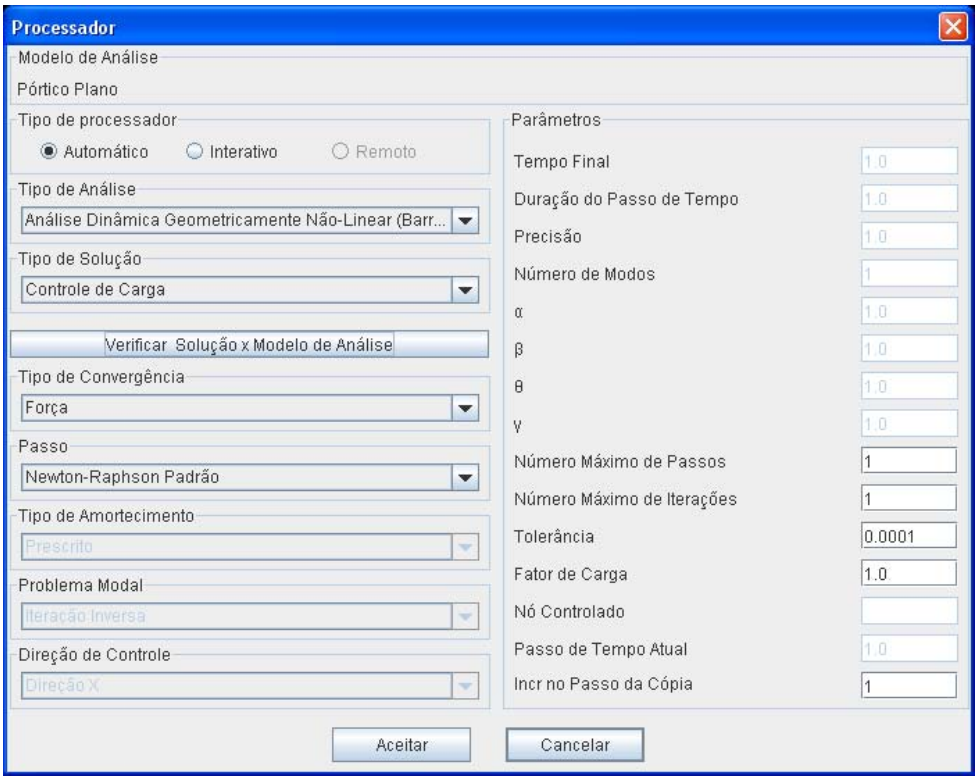


Figura 4.84: Diálogo para definição do processamento.

Optando-se pelo *"Processador Automático"* são disponibilizados seis tipos de análise e para cada tipo é listado as possíveis soluções. Os campos para fornecimento de parâmetros são liberados para preenchimento após o acionamento do botão *"Verificar Solução X Modelo de Análise"*. Neste momento é verificado se o tipo de solução é possível para o modelo de análise global adotado.

Optando-se pelo *"Processador Interativo"* a solução "Linear" é a única possível e vale para todos os modelos de análise. O processador interativo (projeto *br.ufmg.dees.insane.ui.rich.learn*) também representa um módulo de aplicação do

INSANE. Maiores detalhes podem ser encontrados nos trabalhos de Nicoliello (2005) e Fernández (2008).

4.7 Persistência do Modelo

Após o processamento, dentro da pasta onde foi gravado o modelo geométrico, é criada a pasta *"RESULTS"* (Figura 4.85). Nela o modelo de elementos finitos é persistido de duas formas:

(i) Como objeto JAVA em um arquivo com extensão .isn que pode ser acessado por outros segmentos do sistema.

(ii) Em arquivo XML (eXtensible Markup Language). Trata-se de um formato padronizado de arquivo texto escolhido para fazer a ligação entre os aplicativos do **INSANE** e outras aplicações baseadas no método dos elementos finitos.

Figura 4.85: Pastas para a persistência do modelo.

Dentro das sub-pastas de *"RESULTS"* é gravado um arquivo para cada combinação de carregamento que foi definido no modelo. A Figura 4.86 ilustra a persistência de arquivos XML.

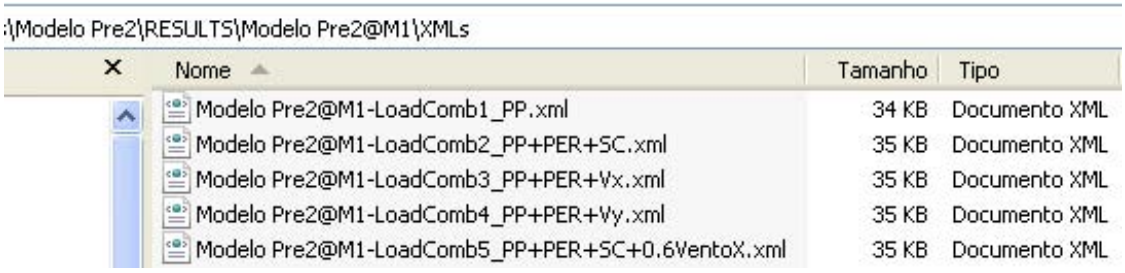


Figura 4.86: Persistência de arquivos XML para cada combinação.

4.8 Exemplos

Esta seção apresenta alguns modelos que foram gerados com os recursos apresentados nas seções anteriores. Os exemplos foram agrupados nas categorias: Modelos Reticulados, Modelos Planos, Modelos de Placas e Modelos Combinados.

4.8.1 Modelos Reticulados

A Figura 4.87 ilustra uma viga contínua com vários tipos de carregamento.

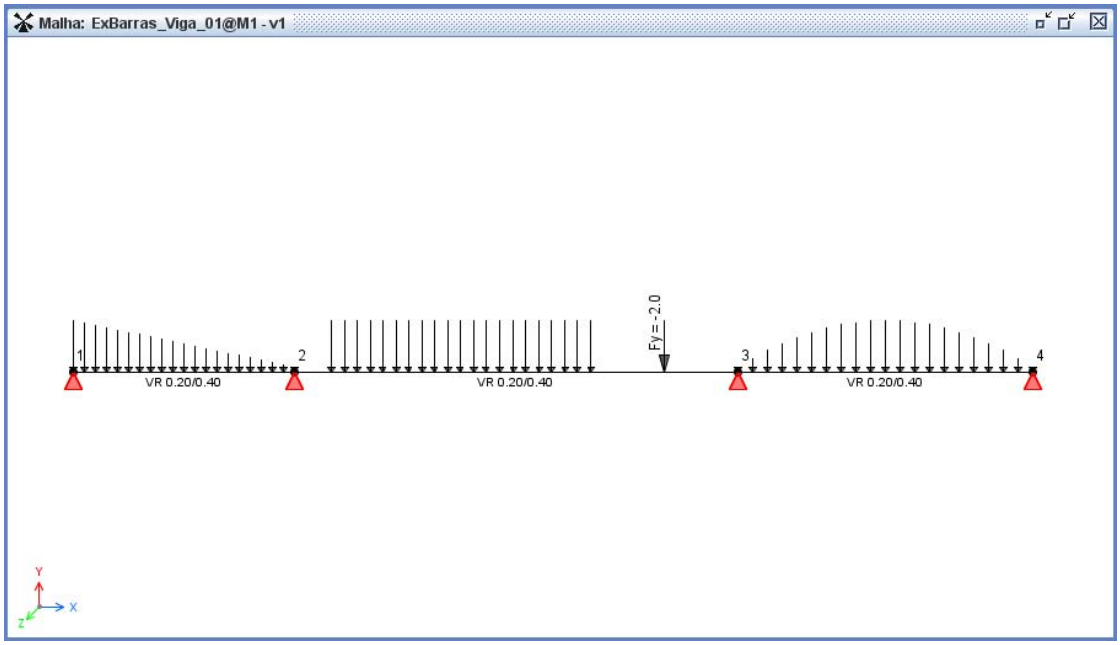


Figura 4.87: Viga contínua.

As Figuras 4.88 e 4.89 ilustram modelos de pórtico plano.

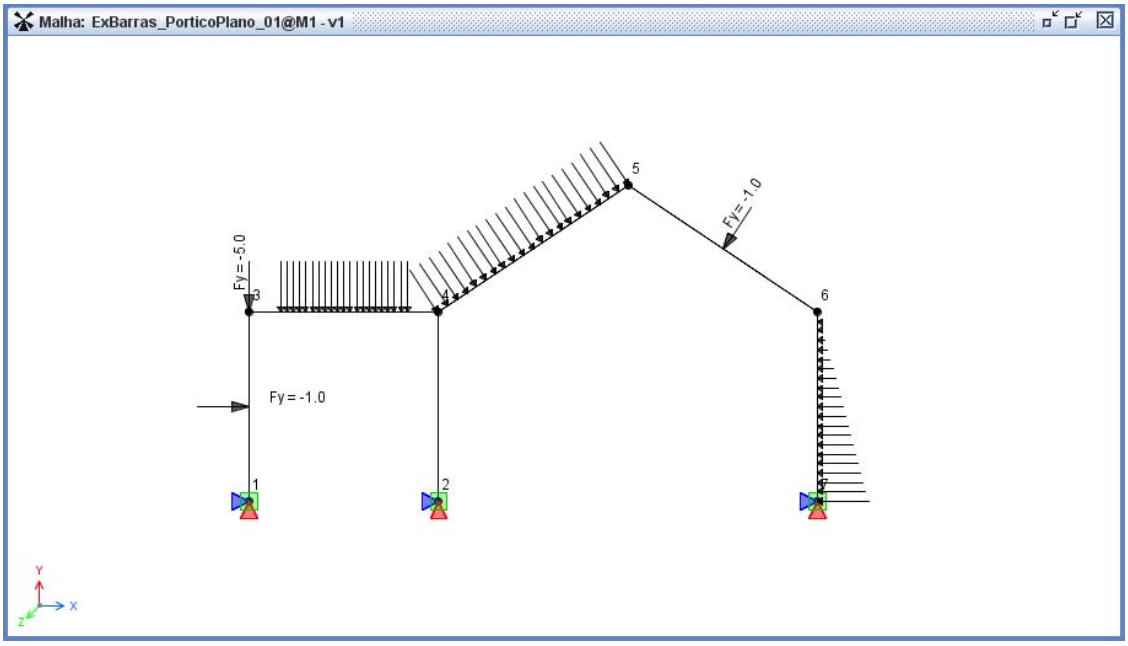


Figura 4.88: Pórtico Plano 1.

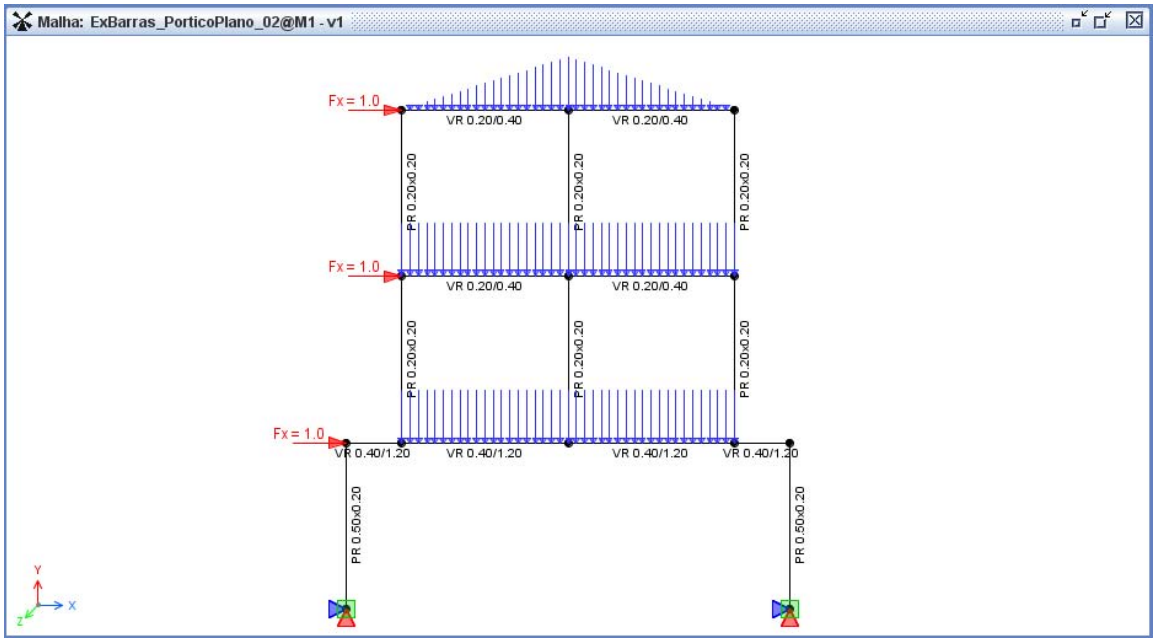


Figura 4.89: Pórtico Plano 2.

A Figura 4.90 ilustra uma grelha gerada através do “Gerador de Grelhas”.

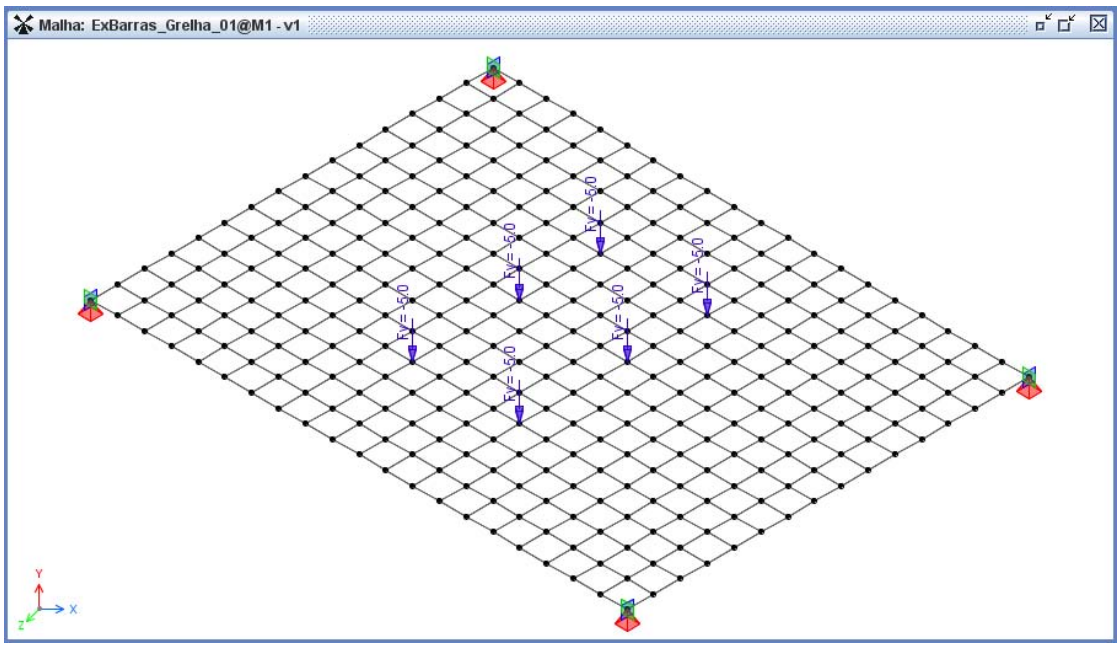


Figura 4.90: Grelha 1.

A grelha da Figura 4.91 foi desenhada no plano XY e depois rotacionada para o plano XZ.

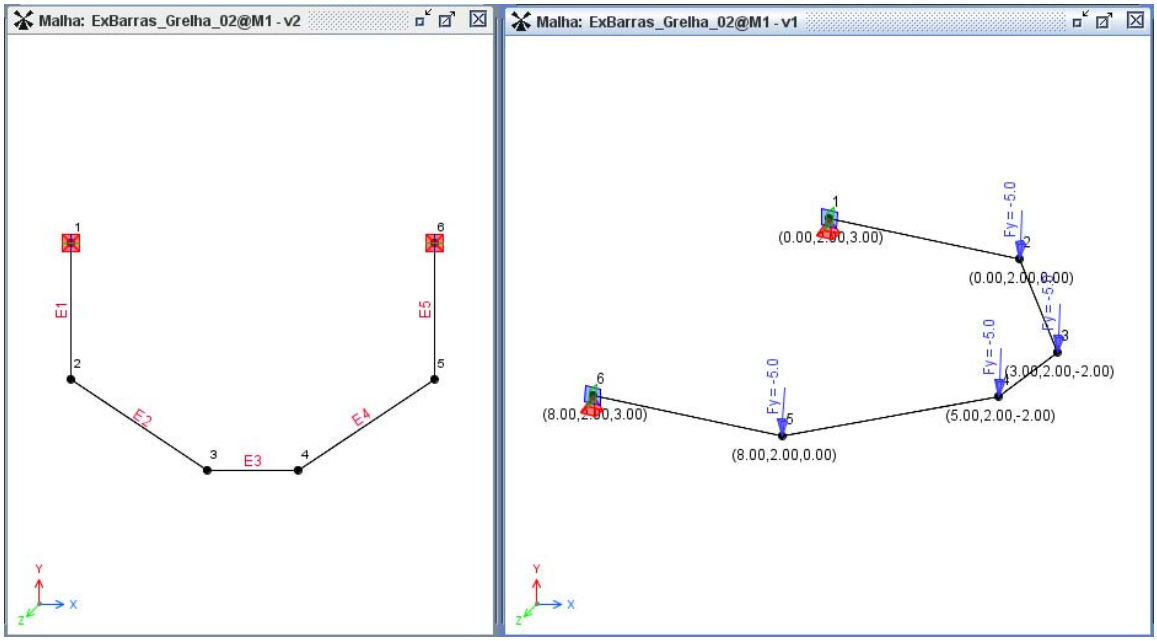


Figura 4.91: Grelha 2.

As Figuras 4.92 e 4.93 ilustram dois modelos de treliça.

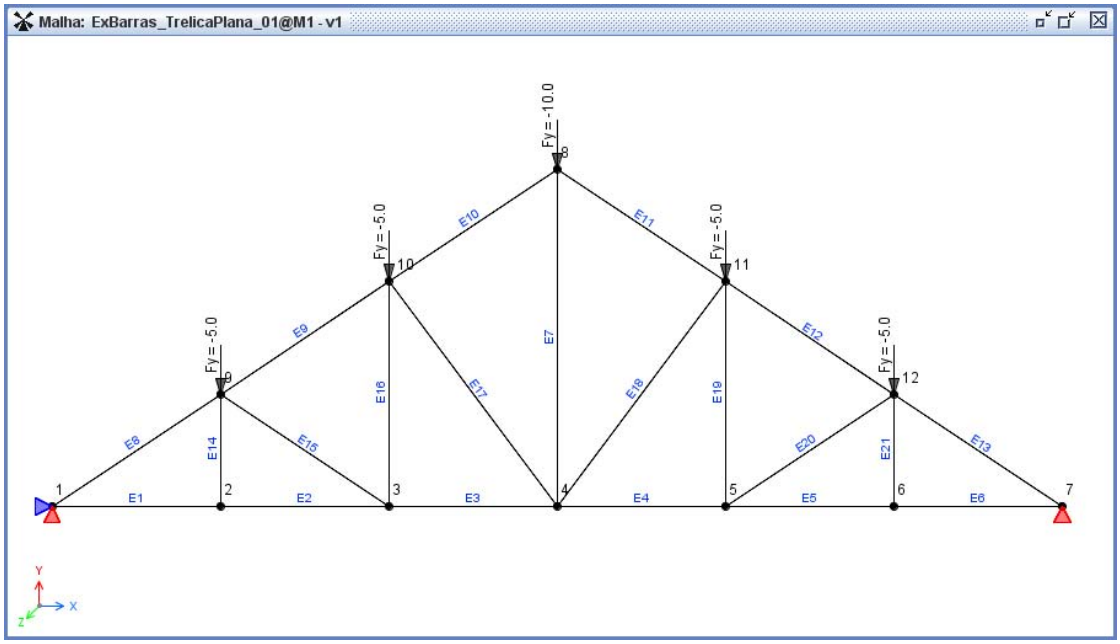


Figura 4.92: Treliça Plana.

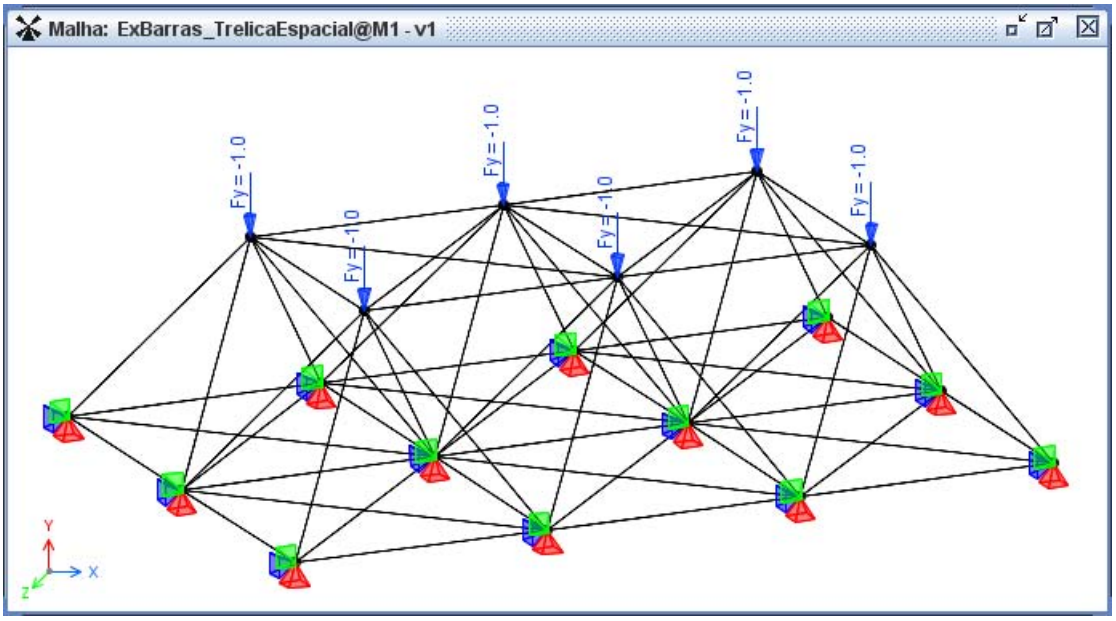


Figura 4.93: Treliça Espacial.

A Figura 4.94 ilustra um pórtico espacial gerado através do "Gerador de Pórticos" e editado com barras inclinadas.

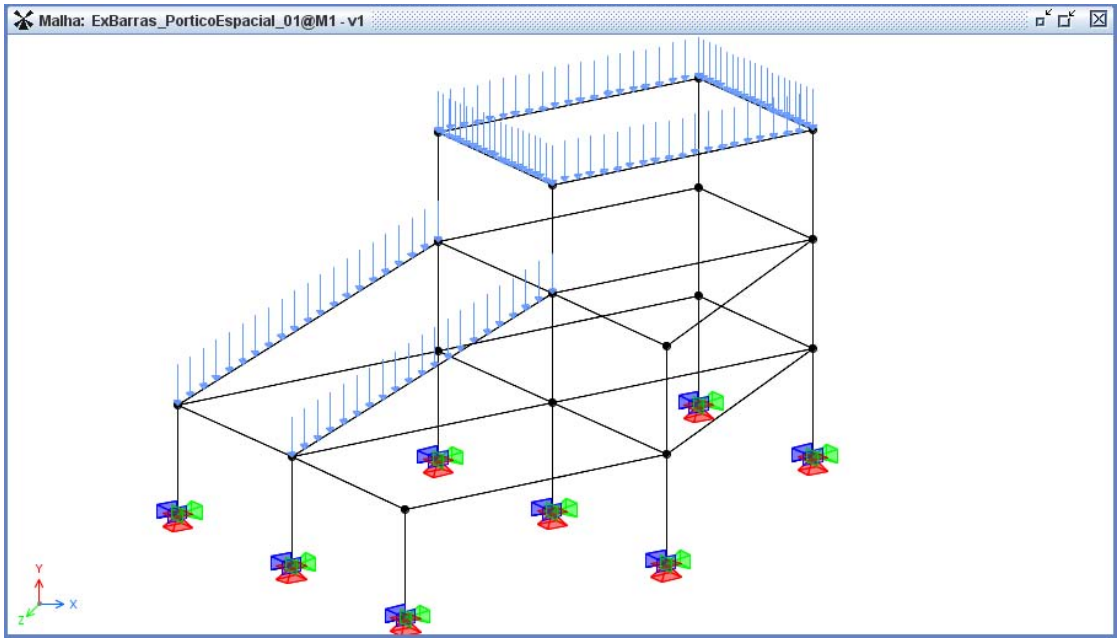


Figura 4.94: Pórtico Espacial 1.

O modelo de galpão apresentado na Figura 4.95 foi gerado com o recurso de "Cópias Paralelas".

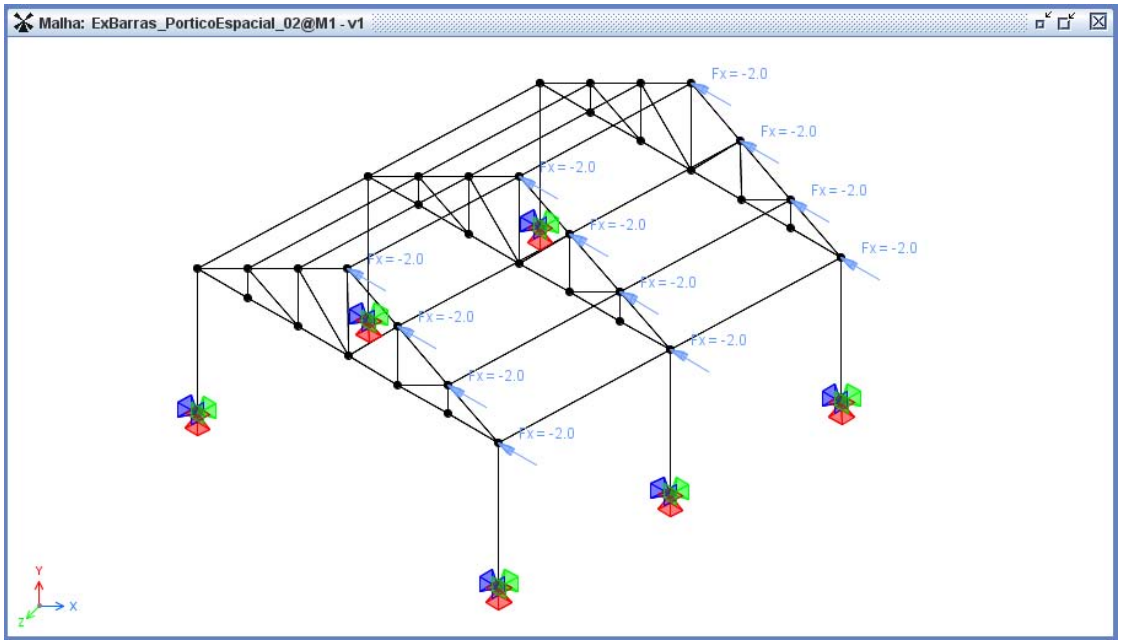


Figura 4.95: Pórtico Espacial 2.

4.8.2 Modelos Planos

As Figuras 4.96 e 4.98 ilustram modelos de vigas altas para análise no Estado Plano de Tensão.

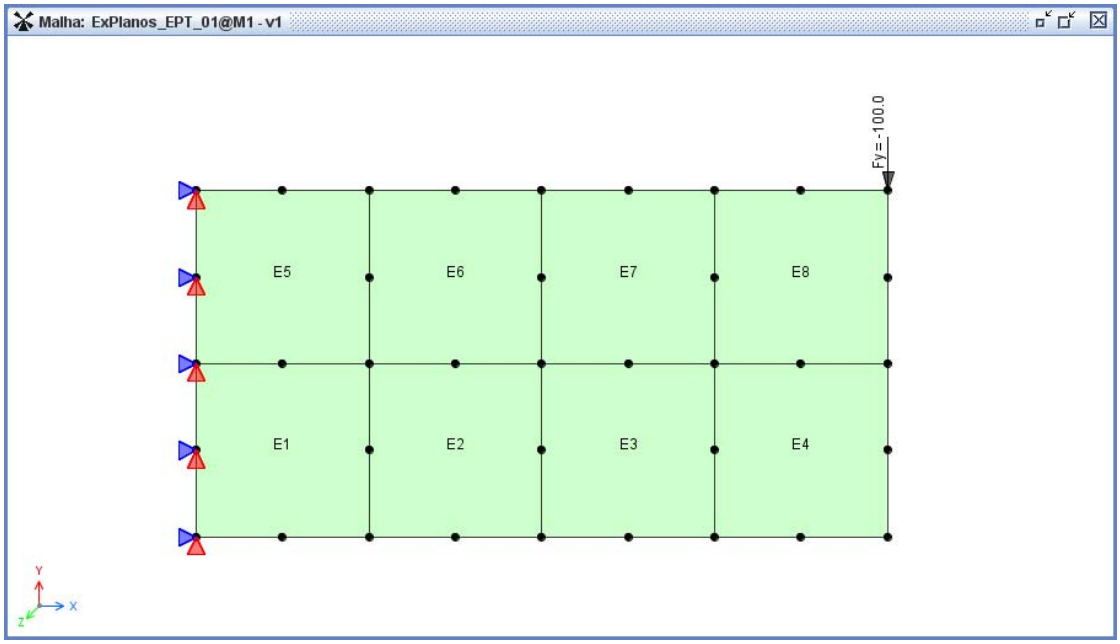


Figura 4.96: Viga Alta - Malha com Q8 e carga concentrada.

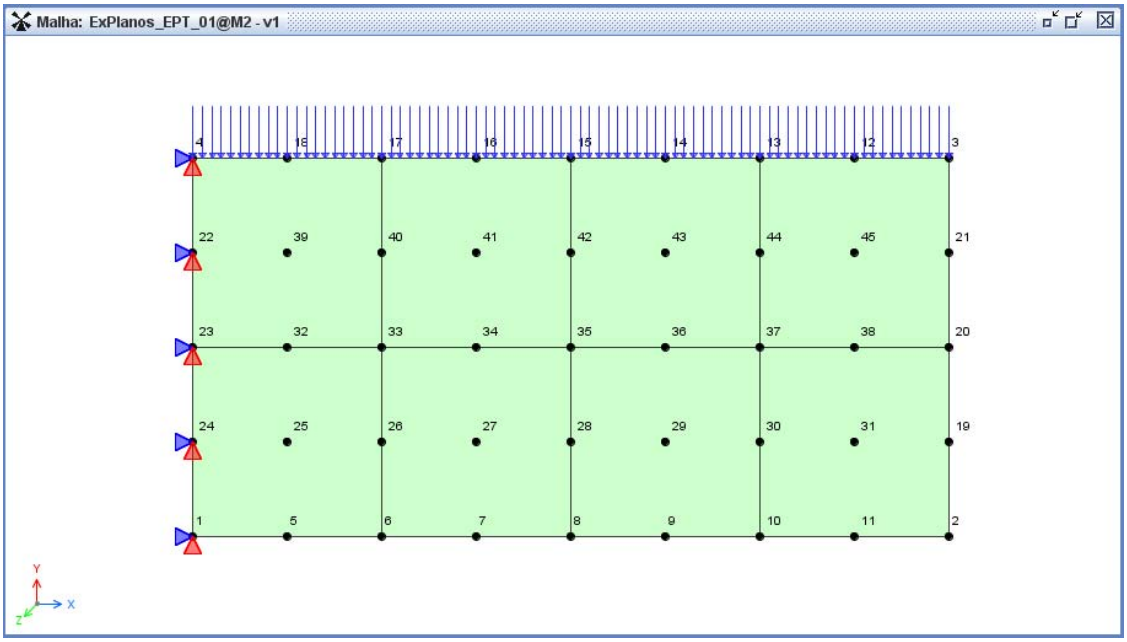


Figura 4.97: Viga Alta - Malha com Q9 e carga distribuída.

A Figura 4.98 ilustra uma chapa com furo para análise de tração.

Figura 4.98: Chapa com furo.

A Figura 4.99 apresenta uma cunha discretizada com elementos T3.

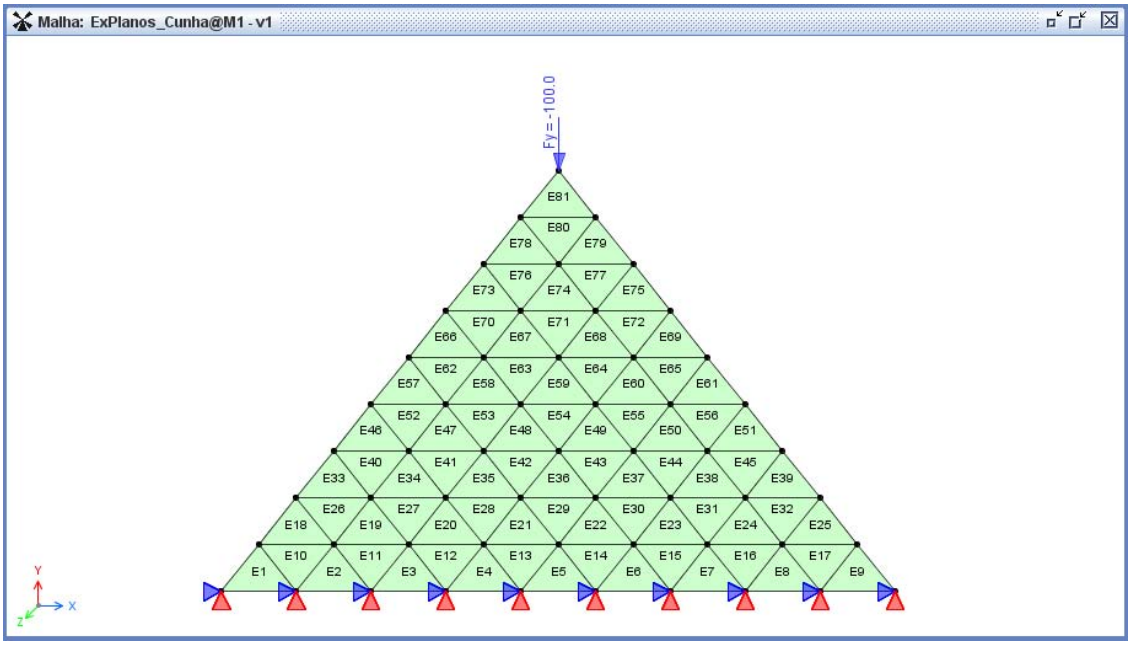


Figura 4.99: Cunha.

A Figura 4.100 ilustra um modelo axissimétrico com elementos Q8.

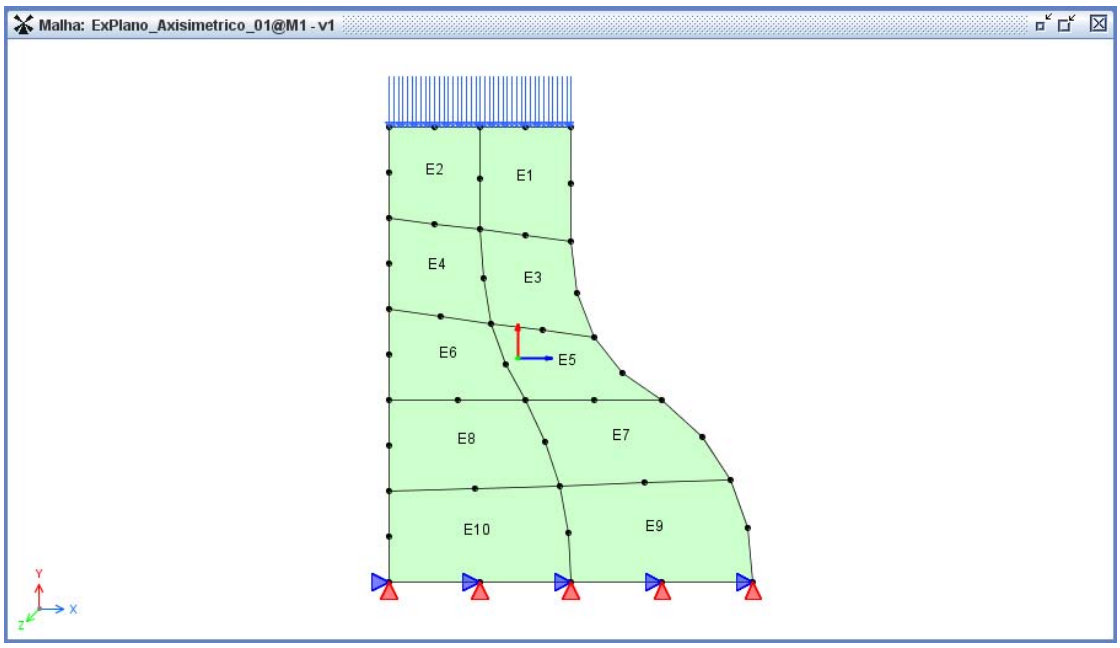


Figura 4.100: Pedestal.

A Figura 4.101 apresenta uma modelo com elementos Q4 para análise no Estado Plano de Deformação.

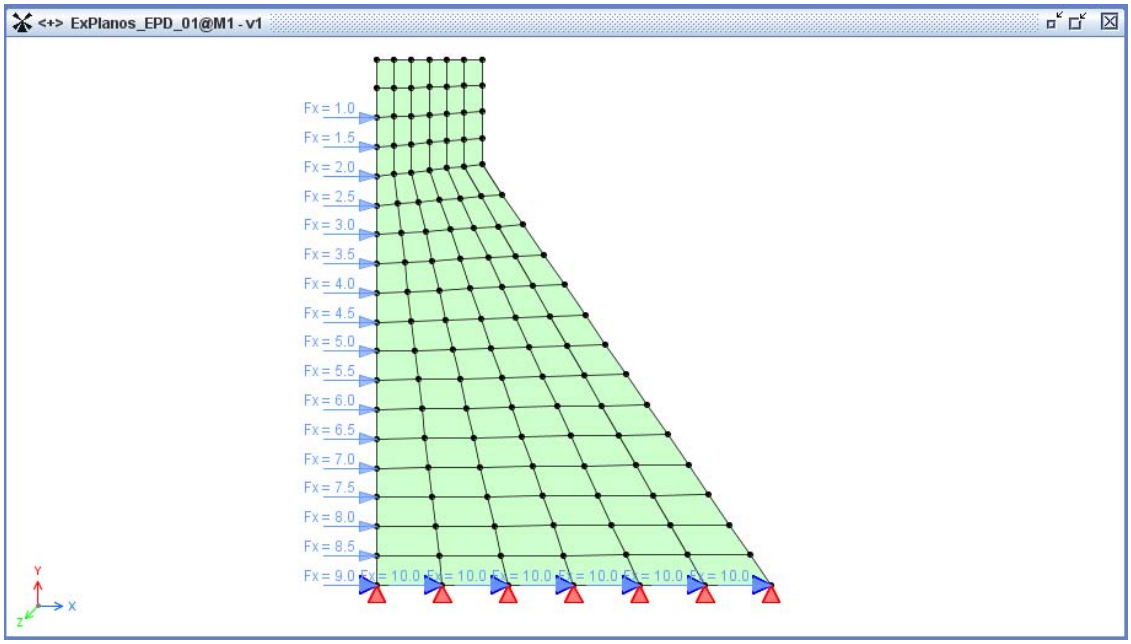


Figura 4.101: Barragem.

4.8.3 Modelos de Placas

As Figuras 4.102 e 4.103 ilustram uma placa anelar discretizada com elementos Q4.

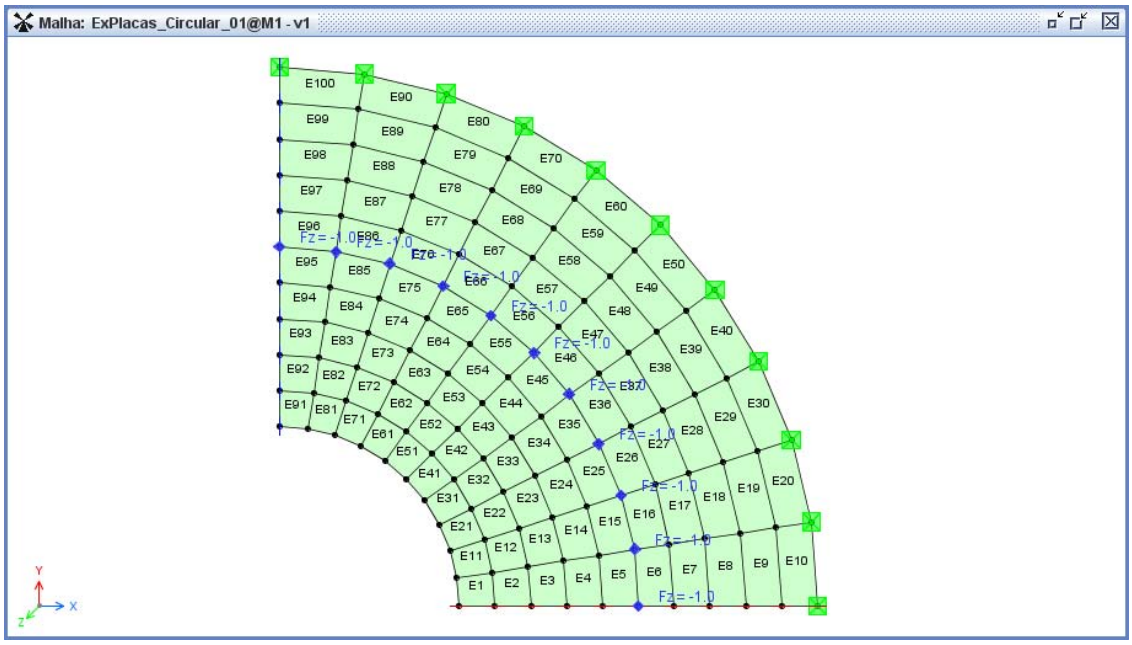


Figura 4.102: Placa Anelar - Malha.

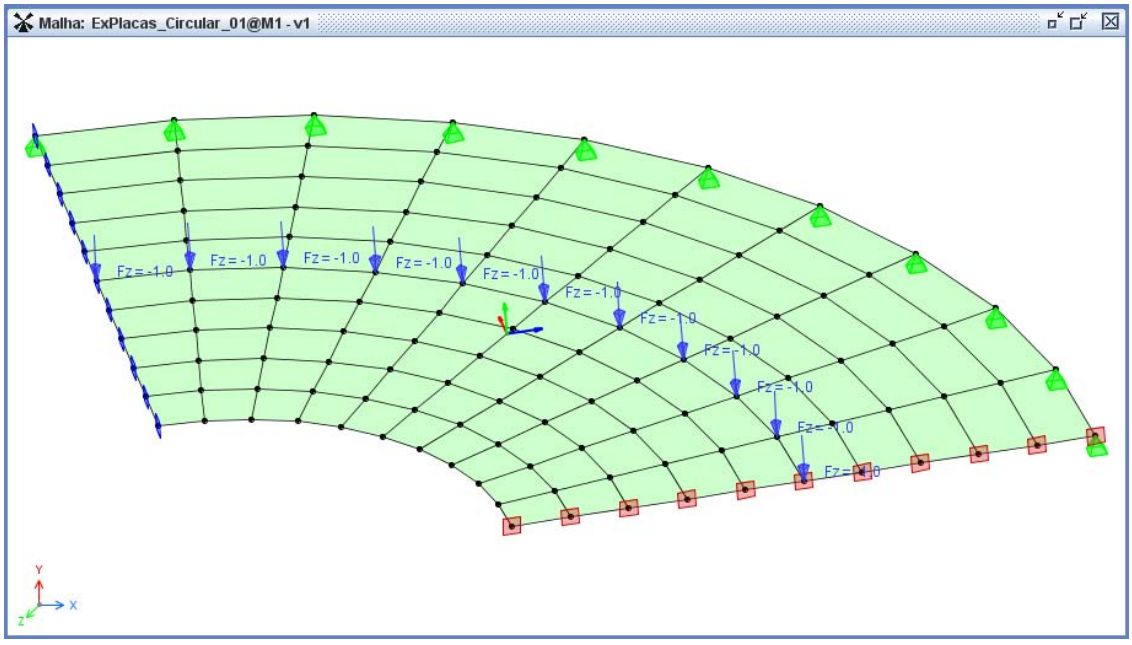


Figura 4.103: Placa Anelar - Detalhe do carregamento.

A Figura 4.104 ilustra um modelo de placa espessa de Reissner Mindlin discretizada com elementos T6.

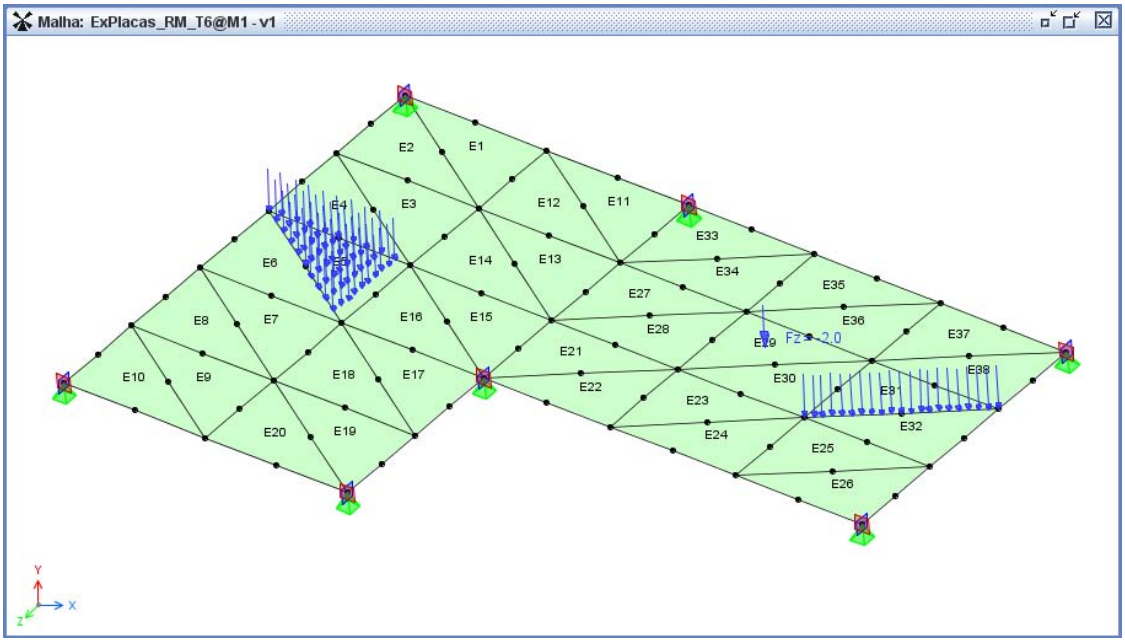


Figura 4.104: Placa de Reissnner Midllin - Modelo 1.

A Figura 4.105 ilustra outro modelo de placa espessa de Reissner Mindlin discretizada com elementos Q4.

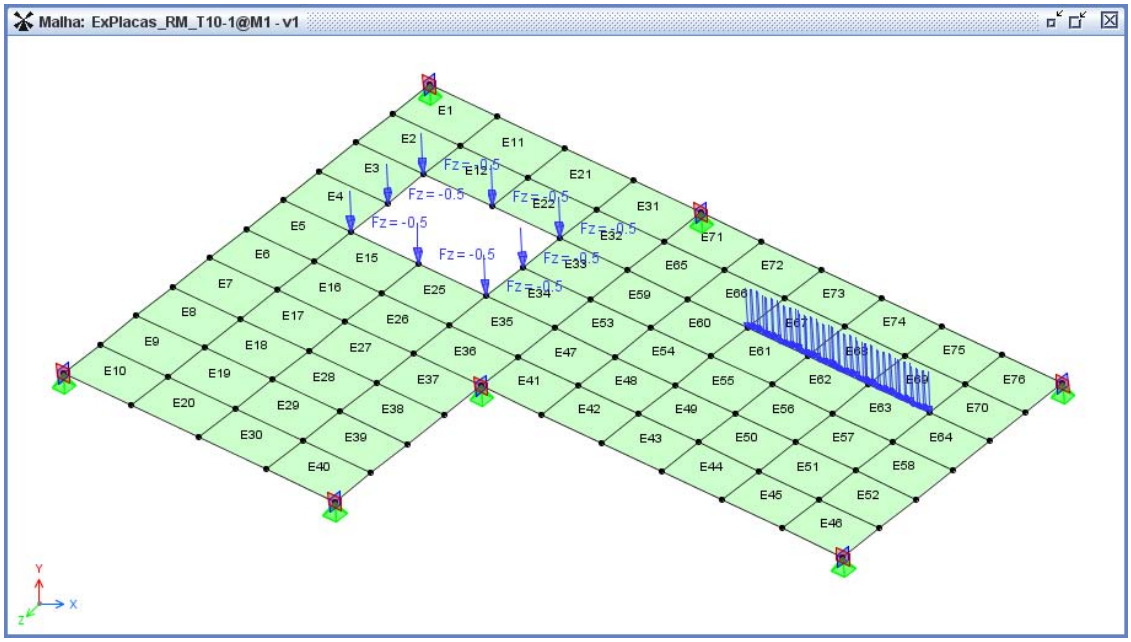


Figura 4.105: Placa de Reissnner Midllin - Modelo 2.

4.8.4 Modelos Combinados

As Figuras 4.106 e 4.107 ilustram um modelo que combina elementos de *Pórtico Espacial de Timoshenko* com elementos de *Placa Fina de Reissner Mindlin*.

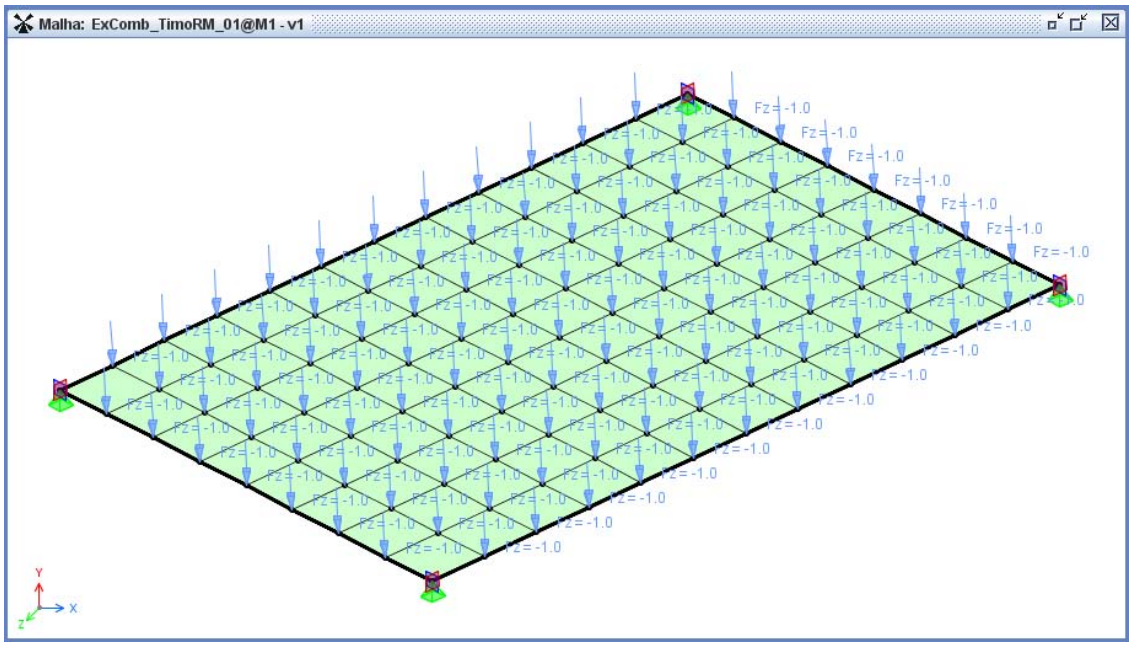


Figura 4.106: Modelo combinado para análise de um pavimento.

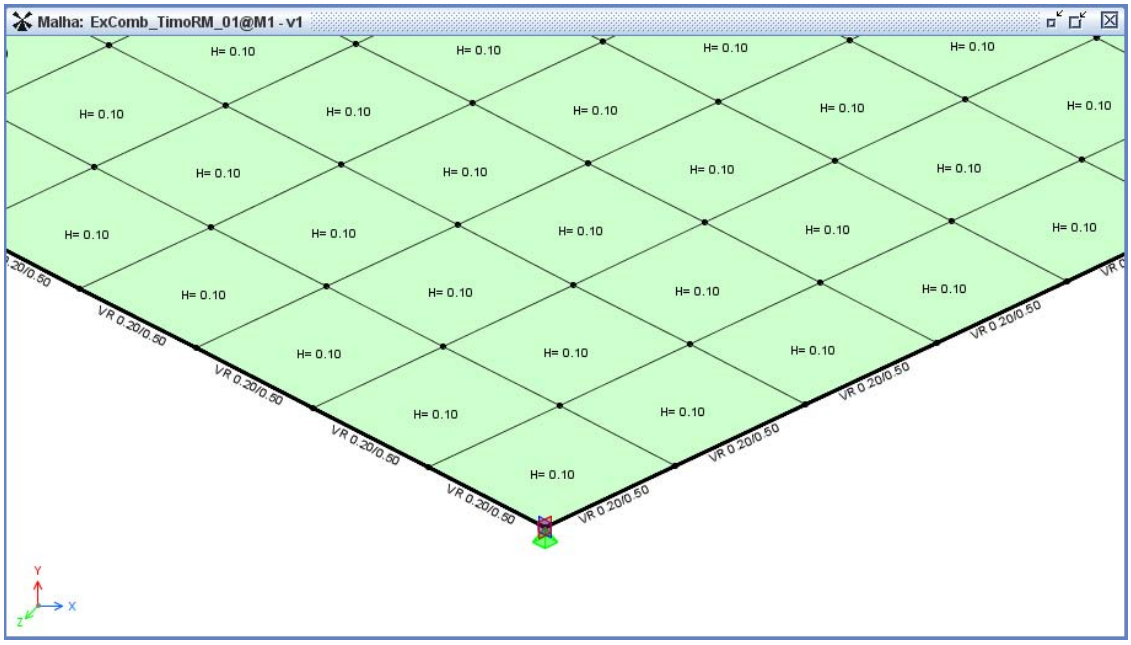


Figura 4.107: Detalhe das seções nos elementos de barra e placa.

As Figuras 4.108 e 4.109 ilustram um modelo que combina elementos de *Pórtico Plano* com elementos de *Estado Plano de Tensões*.

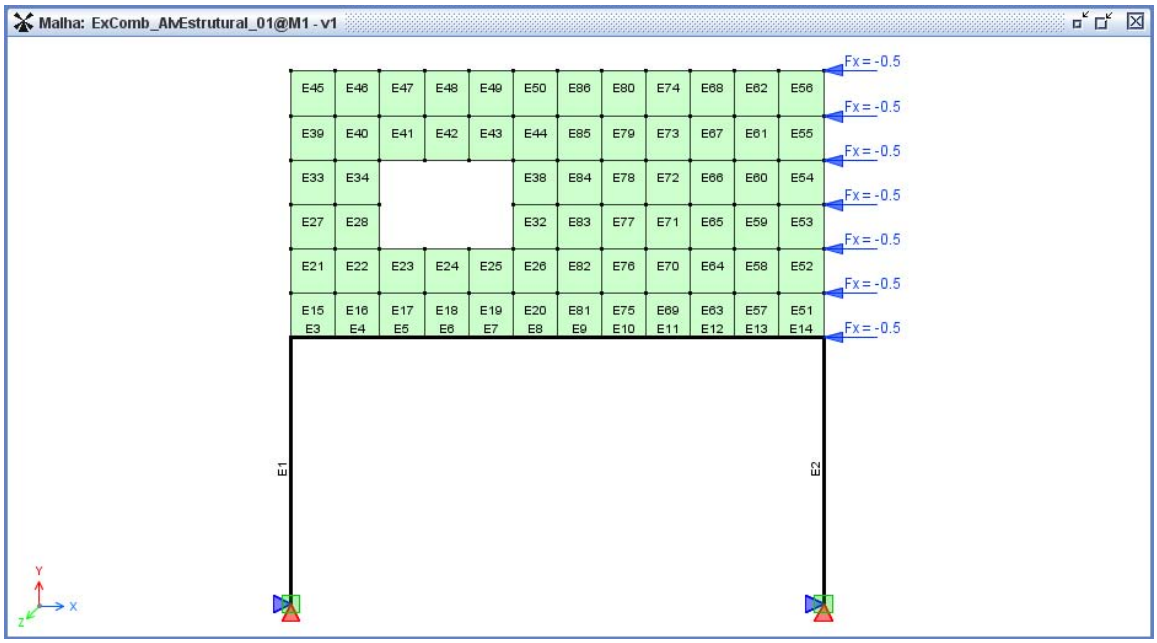


Figura 4.108: Pórtico de concreto mais alvenaria estrutural.

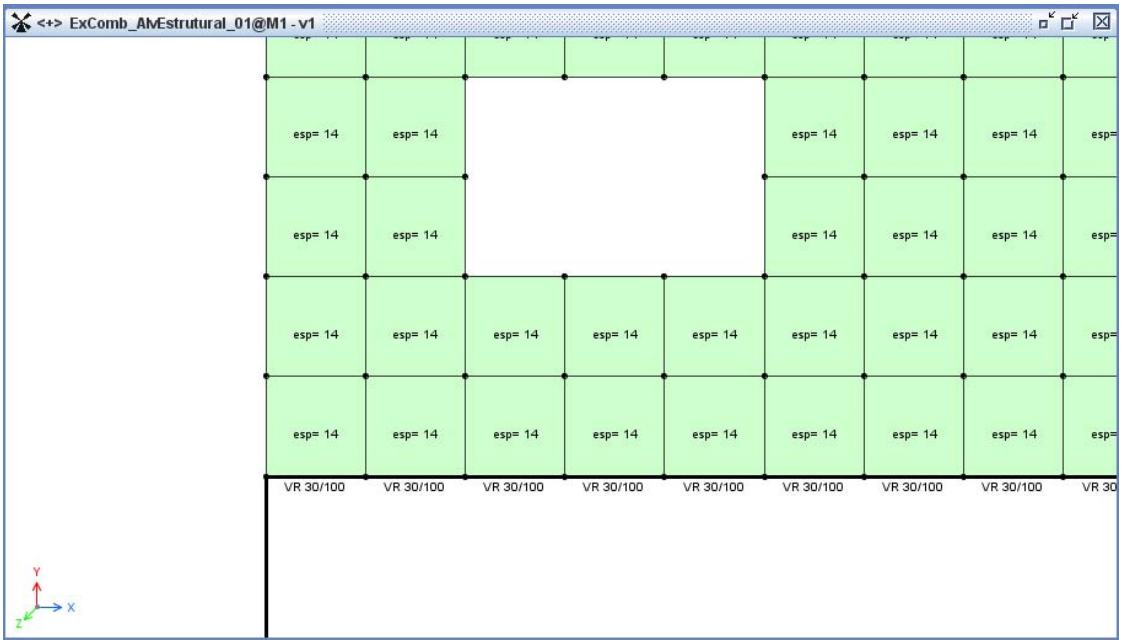


Figura 4.109: Detalhe das seções transversais.

Capítulo 5

CONSIDERAÇÕES FINAIS

A primeira versão do **INSANE** era composta de um pré-processador capaz de criar somente modelos reticulados bidimensionais. Esses modelos podiam ser formados apenas por elementos de barra (com dois nós) para análise de vigas, pórticos planos, treliças planas e grelhas. Também possuía um módulo que permitia a geração de malhas, através de Mapeamento Transfinito, apenas no plano XY.

Com a constante expansão do núcleo numérico do **INSANE** o pré-processador existente tornou-se obsoleto. A partir daí, para analisar um dado problema era necessário a criação de um arquivo XML (Extensible Markup Language), segundo um padrão pré-estabelecido de entrada de dados, contendo as informações do modelo (número de nós e suas coordenadas, número de elementos, materiais, seções, atributos, etc.) e as características da análise. O processamento de um modelo com base nas informações disponíveis em arquivo XML ainda é possível, porém, esse procedimento exige do usuário um bom domínio dessa linguagem e um tempo maior na preparação dos dados.

5.1 Contribuições deste trabalho

O novo pré-processador, objetivo deste trabalho, permite a integração dos atuais recursos disponíveis no **INSANE**. Com ele também é possível a criação de modelos espaciais (pórtico e treliça espacial) e a combinação de elementos finitos de barras (com 2, 3 ou 4 nós), estado plano e placas. Os dados geométricos do modelo, antes armazenados em várias listas (instâncias de classes da API Collection), agora são organizados na estrutura de dados *Half-Edge* (Mäntylä, 1987) o que permitiu estabelecer as relações de adjacências entre os componentes do modelo, possibilitando um rápido acesso a eles. O gerador de malhas desenvolvido por Brugiolo (2004) foi adaptado para permitir a criação de elementos (T3, T6, T10, Q4, Q8, Q9) também nos planos XZ e YZ.

O projeto da interface gráfica assim como todos os comandos e diálogos que foram implementados para essa nova versão, visam proporcionar ao usuário uma interação bem simples e intuitiva, oferecendo facilidades para criação e edição do modelo e proporcionando aumento de produtividade na modelagem.

Além de todos os recursos já mencionados no Capítulo 4, como o modelo de elementos finitos pode ser persistido em objeto JAVA (arquivo binário) ou em arquivo XML, o pré-processador também serve como um "plug-in" para fornecer dados, de forma independente, ao módulo de pós-processamento de modelos bidimensionais não-lineares (Penna, 2007) e ao módulo de serviço WEB (Camara, 2007).

A maneira como os módulos que compõem o pré-processador foram organizados facilita a manutenção do código. Esse módulos foram detalhados no Apêndice C que também destaca as possibilidades de expansão para essa aplicação.

5.2 Sugestões para trabalhos futuros

Algumas sugestões para trabalhos futuros são apresentadas a seguir:

1. Implementação de novas classes para contemplar outras técnicas de geração de malhas como Avanço de Fronteira e Decomposição de Domínio;

2. Implementação de novas classes que suportem modelos sólidos;

3. Utilização de recursos baseados em OpenGL (ambiente para desenvolvimento de aplicações gráficas interativas 2D e 3D) para composição e manipulação de elementos gráficos e imagens de modo a se obter melhor desempenho computacional;

4. Implementação de um diálogo para a criação interativa de seções transversais.

Apêndice A

Sub-projetos do INSANE

Tabela A.1: Sub-projetos do Sistema INSANE

Num	Sub-projeto	Módulo
1	br.ufmg.dees.insane.analysisModel	serviço
2	br.ufmg.dees.insane.assembler	serviço
3	br.ufmg.dees.insane.commons	serviço
4	br.ufmg.dees.insane.continuousPointModel	serviço
5	br.ufmg.dees.insane.draw	serviço
6	br.ufmg.dees.insane.geometry	serviço
7	br.ufmg.dees.insane.help	serviço
8	br.ufmg.dees.insane.images	serviço
9	br.ufmg.dees.insane.langPack	serviço
10	br.ufmg.dees.insane.materialMedia	serviço
11	br.ufmg.dees.insane.model	serviço
12	br.ufmg.dees.insane.model.learn	serviço
13	br.ufmg.dees.insane.model.postp	serviço
14	**br.ufmg.dees.insane.model.prep**	aplicação
15	br.ufmg.dees.insane.parser	serviço
16	br.ufmg.dees.insane.persistence	serviço
17	br.ufmg.dees.insane.solution	serviço
18	br.ufmg.dees.insane.ui.rich.full	aplicação
19	**br.ufmg.dees.insane.ui.rich.geo**	aplicação
20	br.ufmg.dees.insane.ui.rich.learn	aplicação
21	**br.ufmg.dees.insane.ui.rich.mesh**	aplicação
22	br.ufmg.dees.insane.ui.rich.postp	aplicação
23	**br.ufmg.dees.insane.ui.rich.prep**	aplicação
24	br.ufmg.dees.insane.ui.rich.solution	aplicação
25	br.ufmg.dees.insane.ui.web	aplicação
26	br.ufmg.dees.insane.util	serviço
27	br.ufmg.dees.insane.webServiceClient	serviço
28	br.ufmg.dees.insane.xyplot	aplicação

Apêndice B

Chaves criadas em mapas (*HashMap*)

Neste apêndice encontram-se tabelas com a relação das informações armazenadas nos mapas (*HashMap<String, Object>*) das classes *Vertex* e *Face* (ver Capítulo 3).

A Tabela B.1 lista os pares <Chave, Valor> criados no mapa da classe *Vertex*. As chaves são utilizadas pelo modelo de malha (*MeshPrepModel*) para armazenar informações relativas aos atributos dos nós.

Tabela B.1: Chaves criadas no mapa da classe *Vertex*.

Chave (String)	**Tipo**	**O que armazena**
"DOF_LABELS"	StringPointValues	Graus de liberdade do nó
"RESTRAINTS"	BooleanPointValues	Restrições de apoio
"PRE_STATE_VARIABLE"	DoublePointValues	Deslocamentos prescritos
"COEFFICIENTS_A"	DoublePointValues	Coeficientes de massa
"COEFFICIENTS_B"	DoublePointValues	Coeficientes de amortecimento
"COEFFICIENTS_C"	DoublePointValues	Coeficientes de mola
"ANGLE"	DoublePointValues	Ângulos do apoio

A Tabela B.2 lista os pares <Chave, Valor> criados no mapa da classe *Face*. As chaves são utilizadas pelo modelo do pre-processador para armazenar informações relativas às regiões (*GeoPrepModel*) ou aos atributos dos elementos (*MeshPrepModel*).

Tabela B.2: Chaves criadas no mapa da classe *Face*.

Chave	**Tipo**	**O que armazena**
"CENTER"	IPoint3d	Coordenadas (x,y,z) do centro da face
"FACETYPE"	String	Tipo da face: "L2" ou "AREA"
"CORNER_VERTEX"	ArrayList	Lista de vértices nos cantos da região
"SHAPE"	String	Função de forma do elemento
"INCIDENCE"	ArrayList	Lista com os Ids de todos os vértices (ou nós)
"INTVERTEX"	ArrayList	Lista com os Ids dos vértices (ou nós) internos
"ENDVERTEX"	ArrayList	Vértices extremos do elemento de barra (L3 ou L4)
"ANALYSISMODEL"	String	Modelo de análise do elemento
"CROSSSECTION"	String	Tipo de seção transversal
"CONSTMODEL"	String	Modelo constitutivo do elemento
"PROBLEMDRIVER"	String	Tipo de problema associado ao elemento
"ELEMENT"	String	Tipo de elemento (barra, placa, paramétrico)
"LENGTH"	Double	Comprimento do elemento de barra
"PTLOAD"	IValue	Pontos mapeados para desenhar carregamento

Apêndice C

Manutenção e Expansão

Conforme já mencionado na introdução deste trabalho, o **INSANE** é um ambiente computacional bem segmentado (uma relação com todos os sub-projetos está disponível no Apêndice B). Isso permite que alterações em seu código ou a inclusão de novos módulos sejam feitas com simplicidade, sem a necessidade de reescrever toda a aplicação.

O objetivo deste apêndice é detalhar os quatro módulos que compõem o pré-processador (já citados no Capítulo 3), destacando as possibilidades de manutenção e expansão para essa aplicação.

C.1 Organização dos Módulos do Pré-Processador

Todas as classes comuns aos módulos *Geometria* e *Malha* foram agrupadas no sub-projeto **br.ufmg.dees.insane.ui.rich.prep**. Novos comandos, diálogos e características da vista, que possam servir para a interface gráfica do modelo geométrico ou do modelo de malha, devem ser incluídos num dos sub-pacotes mostrados na Figura C.1.

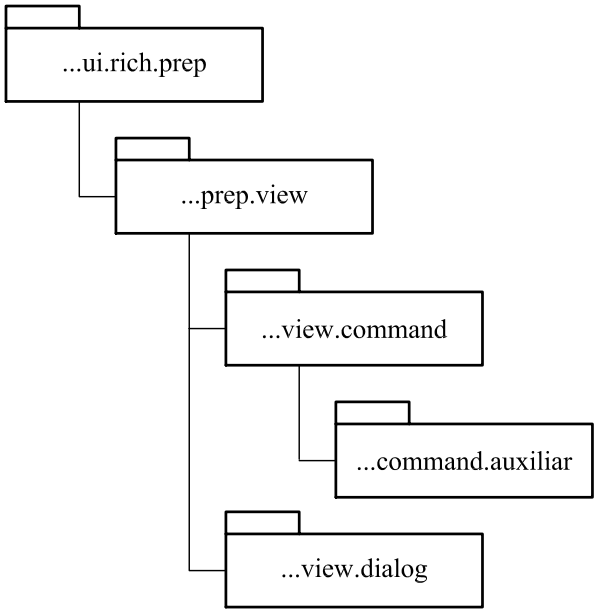


Figura C.1: Pacotes do projeto *...rich.prep*.

O sub-projeto **br.ufmg.dees.insane.ui.rich.geo** reúne as classes relacionadas com a interface gráfica do pré-processador geométrico. Sua estrutura de pacotes está ilustrada na Figura C.2.

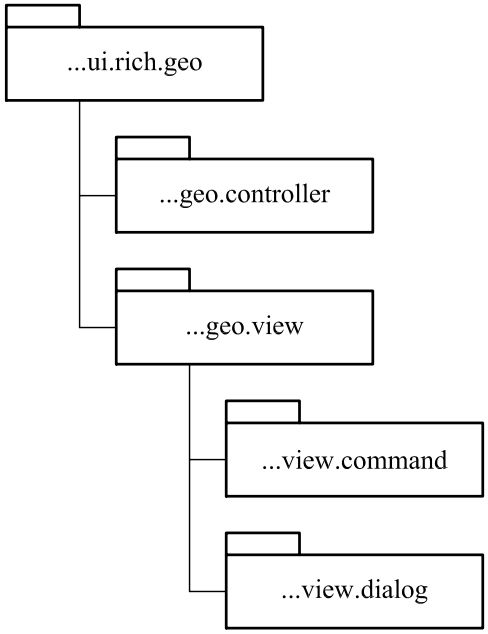


Figura C.2: Pacotes do projeto *...rich.geo*.

Já no sub-projeto **br.ufmg.dees.insane.ui.rich.mesh** estão reunidas as classes

relacionadas à interface gráfica do pré-processador de malhas (Figura C.3).

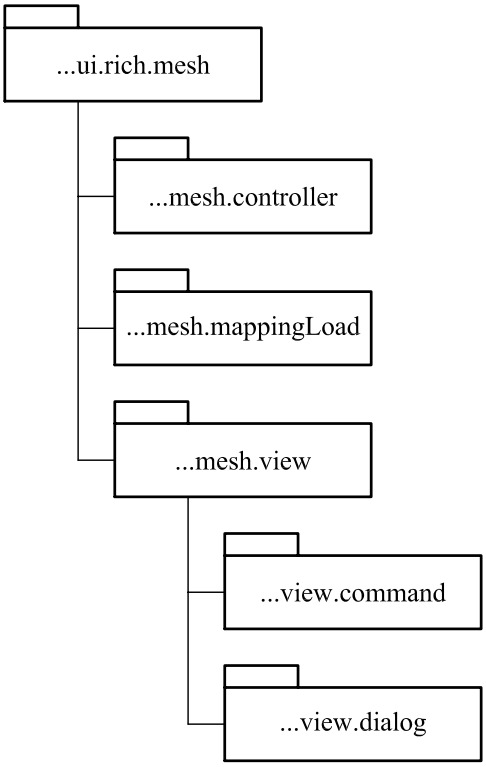


Figura C.3: Pacotes do projeto *...rich.Mesh*.

Em **br.ufmg.dees.insane.model.prep** formam agrupadas as classes que formam os modelos do pré-processador. Manutenções no modelo geométrico (*GeoPrepModel*) devem ser feitas no pacote *...model.prep.geo* (Figura C.4).

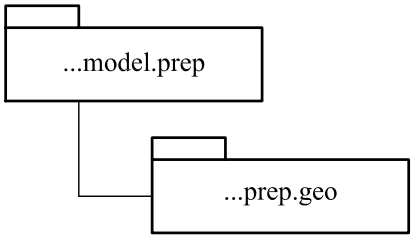


Figura C.4: Pacotes do modelo geométrico.

Para o modelo de malha (*MeshPrepModel*), as alterações devem ser feitas nas classes dos sub-pacotes de *br.ufmg.dees.insane.model.prep.mesh* (Figura C.5).

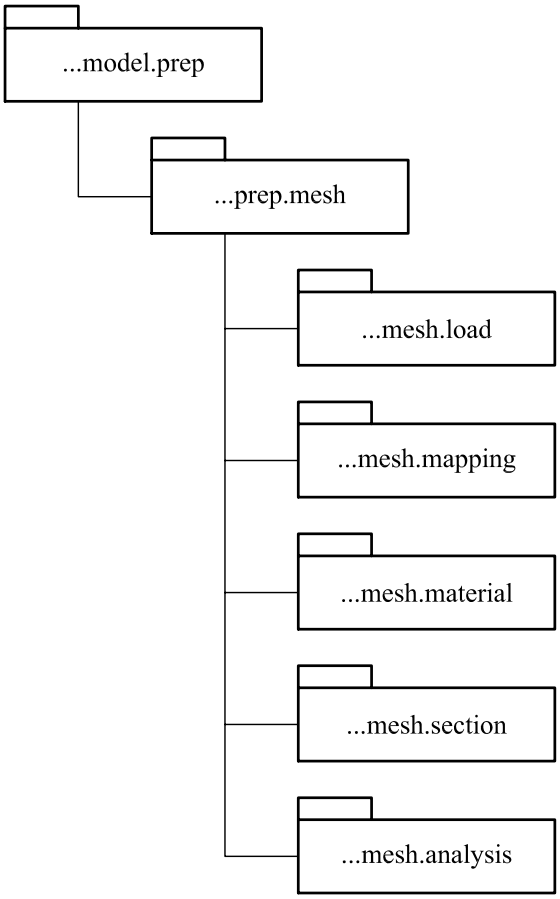


Figura C.5: Pacotes do modelo de malha.

C.2 Internacionalização da Aplicação

No projeto **br.ufmg.dees.insane.langPack** foram agrupados os arquivos responsáveis pela internacionalização da aplicação. Todos os textos que aparecem nos menus e diálogos do pré-processador devem ser incluídos nos arquivos **UiRichPrep_pt.java_native** e **UiRichPrep.java_native**. Em cada arquivo deve ser criado um identificador no formato {“chave”, “texto”}, conforme o exemplo a seguir:

```
Em português:   {"dlgAnalysis_msg1", "Viga"},
Em inglês:      {"dlgAnalysis_msg1", "Beam"},
```

C.3 Inclusão de Novos Tipos de Análise

A classe ***MeshAnalysisModel*** (pacote *...model.prep.mesh.analysis*) possui o método ***getAnalysisTypeList()*** que retorna uma lista com os tipos de análise que foram disponibilizados no pré-processador (já citados no capítulo 4). Para inserir uma referência a um novo tipo, primeiro é necessário criar uma chave nos arquivos de internacionalização (conforme explicado na seção anterior) e depois incluir essa referência no final desse método. Um trecho do código está reproduzido a seguir:

```
public class MeshAnalysisModel {
    private ArrayList<String> idMaterialList;
    private AnalysisModel globalAnalysisModel;
    private ResourceBundle bundle;
    private String  problemDriverBar, problemDriverPlane;
    private String  elementTypeBar, elementTypePlane;
    private String  elementShapeBar, elementShapePlane;
    private String  analysisModelBar, analysisModelPlane;
    private boolean enableLoadButtons;

public ArrayList<String> getAnalysisTypeList() {
    ArrayList<String> anTypeList = new ArrayList<String>();
    anModelList.add(bundle.getString("dlgAnalysis_msg1"));
    anModelList.add(bundle.getString("dlgAnalysis_msg2"));
    ...
    return anTypeList;
}
```

O método ***makeAnalysisModel(String analysisTypeId)***, também presente na classe *MeshAnalysisModel*, recebe como parâmetro um nome que identifica o tipo de análise escolhido pelo usuário e cria uma instância de uma classe que representa o **Modelo de Análise Global**. Também inicializa variáveis que representam atributos dos elementos e monta uma lista com os tipos de materiais compatíveis com o modelo de análise. Um trecho desse método está ilustrado a seguir:

```
public void makeAnalysisModel(String analysisTypeId) {
    ArrayList<String> analysisTypeList = this.getAnalysisTypeList();
    analysisModelBar    = "NULL";
    analysisModelPlane  = "NULL";
    elementTypeBar      = "NULL";
    elementTypePlane    = "NULL";
    problemDriverBar    = "NULL";
    problemDriverPlane  = "NULL";
    elementShapeBar     = "NULL";
    elementShapePlane   = "NULL";
    idMaterialList      = new ArrayList<String>();
    if (analysisTypeId.equalsIgnoreCase(analysisTypeList.get(0))) {
        globalAnalysisModel = new Beam();
        analysisModelBar    = globalAnalysisModel.getLabel();
        elementTypeBar      = "FrameElement";
        elementShapeBar     = "LinearCubic1DCart";
        problemDriverBar    = "Frame";
        enableLoadButtons   = true;
        idMaterialList.add("LinearElasticIsotropic");
    } else if...
```

Para os tipos de análise que combinam elementos, as variáveis mostradas no início do método ***makeAnalysisModel(...)*** são inicializadas para os elementos de barra e para os elementos planos. O trecho de código a seguir ilustra os valores que foram definidos para a combinação *"Plane Frame + Plane Stress"*:

```
// Plane Frame + Plane Stress
if (analysisTypeId.equalsIgnoreCase(analysisTypeList.get(33))) {
    globalAnalysisModel = new PlaneFrame();

    analysisModelBar    = globalAnalysisModel.getLabel();
    elementTypeBar      = "FrameElement";
    elementShapeBar     = "LinearCubic1DCart";
    problemDriverBar    = "Frame";

    analysisModelPlane  = (new PlaneStress()).getLabel();
    elementTypePlane    = "ParametricElement";
    elementShapePlane   = "NULL";
    problemDriverPlane  = "Parametric";

    enableLoadButtons   = true;
    idMaterialList.add("LinearElasticIsotropic");
    }
```

O diálogo **Modelo de Análise Global** (Figura C.6) é criado através do comando ***SetGlobalAnalysisModelCommand*** (pacote *.ui.rich.mesh.view.command*).

Tanto o diálogo quanto o comando utilizam-se da classe ***MeshAnalysisModel***. O trecho de código a seguir destaca a criação do diálogo e a chamada ao método *makeAnalysisModel(...)*.

```
// Show a dialog to choose the analysis model.
GlobalAnalysisModelDialog dlg =
            new GlobalAnalysisModelDialog(intFrame);
if (dlg.getOptionButton() == 0) {
    String gAnlType = dlg.getAnalysisTypeId();
    MeshAnalysisModel meshAnalysis = new MeshAnalysisModel(bundle);
    meshAnalysis.makeAnalysisModel(gAnlType);
    globalAnModel = meshAnalysis.getGlobalAnalysisModel();
    this.setVertexKeys(meshPlnSub);
    this.setFaceKeys(meshPlnSub, meshAnalysis);
```

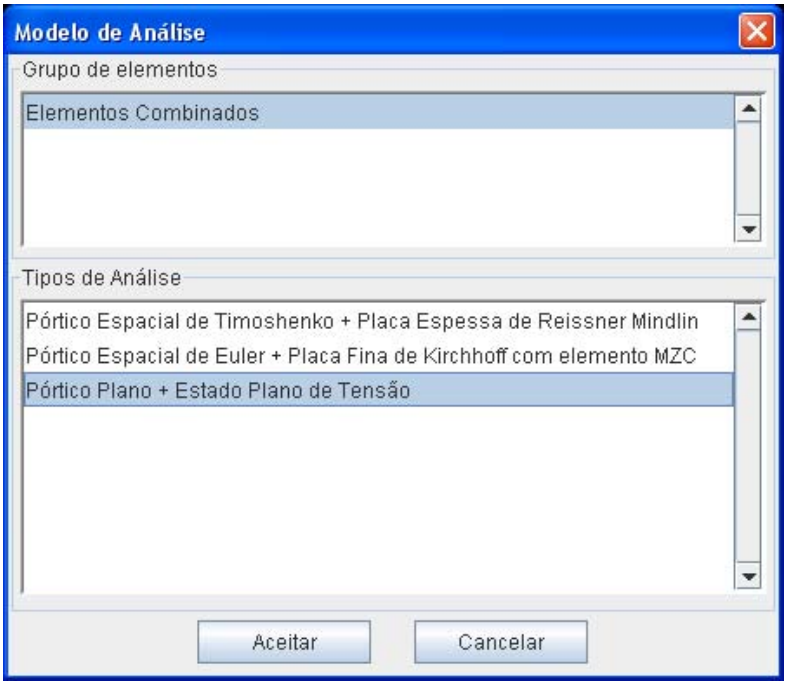


Figura C.6: Diálogo para escolha do Modelo de Análise Global.

O próximo trecho de código destaca o método ***setFaceKeys(...)***. Através dele todas as *Faces* que compõem a Subdivisão Planar do modelo são percorridas. Verifica-se em cada **Face** se a chave *"FACETYPE"* representa um elemento de barra ou plano e então, os valores inicializados em *makeAnalysisModel(...)* são

atribuídos nas chaves *"ANALYSISMODEL"*, *"ELEMENT"*, *"PROBLEMDRIVER"* e *"SHAPE"*.

```
private void setFaceKeys(...) {
    for (int i = 0; i < meshPlnSub.getFaceList().size(); i++) {
        Face face = meshPlnSub.getFaceList().get(i);
        String faceType = (String) face.getValue("FACETYPE");
        if (faceType.equalsIgnoreCase("AREA")) {
            face.setValue("ANALYSISMODEL", analysisModelPlane);
            face.setValue("ELEMENT", elementTypePlane);
            face.setValue("PROBLEMDRIVER", problemDriverPlane);
            if (!elementShapePlane.equalsIgnoreCase("NULL")) {
            face.setValue("SHAPE", elementShapePlane);
            }
        } else {
            face.setValue("ANALYSISMODEL", analysisModelBar);
            face.setValue("ELEMENT", elementTypeBar);
            face.setValue("PROBLEMDRIVER", problemDriverBar);
            if (!elementShapeBar.equalsIgnoreCase("NULL")) {
                    face.setValue("SHAPE", elementShapeBar);
            }
        }
```

Quando as variáveis *elementShapeBar* e *elementShapePlane* são inicializadas no método *makeAnalysisModel(...)* com o valor "NULL", significa que a função de forma do elemento é definida após a geração da malha. Os comandos *MakeAutomaticMesh* e *MappingCommand* (pacote *...ui.rich.mesh.view.command*) definem na chave *"SHAPE"* os valores: "L2", "L3", "L4", "T3", "T6", "T10", "Q4", "Q8" ou "Q9".

C.4 Inclusão de Novos Materiais

No diálogo *Material*, representado pela classe ***ModelMaterialListDialog*** (pacote *...ui.rich.mesh.view.dialog*), o componente *Meio Material* (Figura C.7) é inicializado com os tipos de materiais compatíveis com modelo de análise global adotado.

Figura C.7: Componente para definição do Meio Material.

Um trecho do método ***getMaterialListPanel()***, reproduzido a seguir, ilustra a criação do componente:

```
// ComboBox
    MeshAnalysisModel mam = new MeshAnalysisModel(bundle);
    mam.makeAnalysisModel(meshModel.getGlobalAnTypeId());
    ArrayList<String> idMaterialList = mam.getIdMaterialList();
    cmbMatMedia = new JComboBox();
    for (int i = 0; i < idMaterialList.size(); i++) {
        cmbMatMedia.addItem(idMaterialList.get(i));
    }
```

Referências a novos materiais devem ser incluídas na variável *idMaterialList* que está dentro do método *makeAnalysis(String gAnModelId)*, comentado na seção anterior. Já as classes que representam novos meios materiais devem ser incluídas no pacote *br.ufmg.dees.insane.model.prep.mesh.material*.

Para que uma instância de um novo material seja criada e adicionada ao modelo, quando da ativação do botão *Adicionar* (diálogo *Material*), a referência ao novo meio material também deve ser incluída no método ***addMaterial()***, presente na classe *ModelMaterialListDialog*. O trecho de código a seguir ilustra o método:

```
private void addMaterial() {
    String currentMatMedia = (String)
    cmbMatMedia.getSelectedItem();
    MaterialNameDialog dlg =
        new MaterialNameDialog(..., currentMatMedia);

    if (dlg.getOptionButton() == 0) {
        MeshMaterial mat = null;
        String newName = dlg.getName();
        double shearmod = 0.4167; // = 1.0 / [ 2 x (1 + 0.2) ]

         if (currentMatMedia.equals("LinearElasticIsotropic")) {
         // ( elast, shearmod, ni, shearfactor, thermalcoeff )
            mat = new MeshLinearElasticIsotropic(1.0, shearmod, ...);
            mat.setLabel(newName);
            mat.setMaterialMedia("LinearElasticIsotropic");
            meshModel.getMaterialsList().add(mat);
         } else if (...) {
```

C.5 Inclusão de Novas Seções

Para incluir no diálogo de seções transversais (Figura C.8) mais uma opção de tipo de seção é preciso:

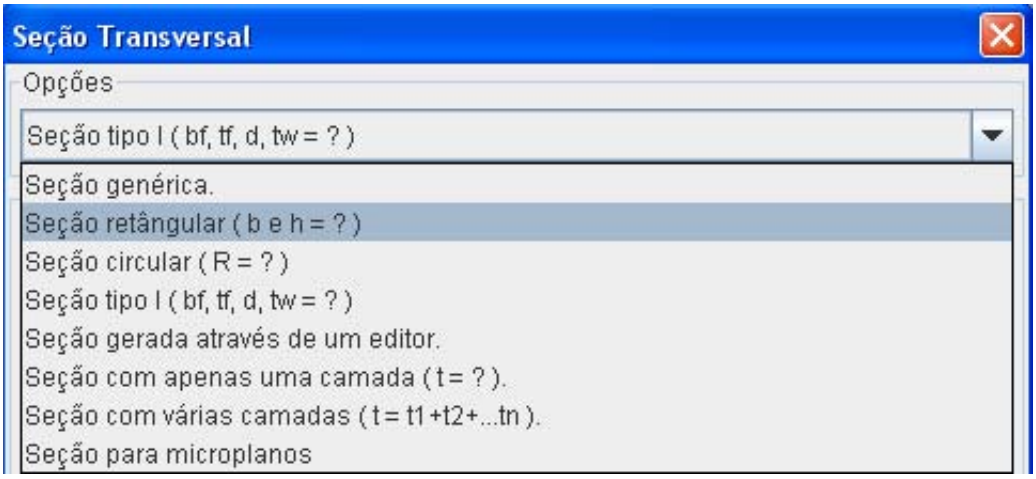


Figura C.8: Opções de seção.

1. Criar no pacote *...model.prep.mesh.section* uma classe para representar a nova seção transversal;

2. Criar no pacote *...ui.rich.mesh.dialog* uma classe para representar o diálogo dessa nova seção;

3. Na classe ***ModelCrossSectionDialog***, presente no mesmo pacote, editar os métodos ***getSectionsModePanel()***, ***addCrossSection()*** e ***modifyCrossSection()*** para incluir a referência do novo diálogo na lista de opções.

O trecho a seguir ilustra o método ***getSectionsModePanel()*** responsável pela criação do componente que exibe os tipos de seção. Devem ser incluídos nos arquivos de internacionalização do pré-processador (ver seção C.2), os textos que identificam cada tipo.

```
private JPanel getSectionsModePanel() {
...
// ComboBox
cmbDegeneration = new JComboBox();
 // Generic section
cmbDegeneration.addItem(bundle.getString("dlgCrossSec_msg16"));
 // Retangular section ( b e h = ? )
cmbDegeneration.addItem(bundle.getString("dlgCrossSec_msg17"));
 // Circular section ( R = ? )
cmbDegeneration.addItem(bundle.getString("dlgCrossSec_msg18"));
 // Shape I ( bf, tf, h, tw = ? )
cmbDegeneration.addItem(bundle.getString("dlgCrossSec_msg19"));
 // Section generate by an editor
cmbDegeneration.addItem(bundle.getString("dlgCrossSec_msg20"));
 // Section with only one layer ( t = ? )
cmbDegeneration.addItem(bundle.getString("dlgCrossSec_msg21"));
 // Section with more layers ( t = t1+t2+...tn )
cmbDegeneration.addItem(bundle.getString("dlgCrossSec_msg22"));
 // Microplane section
cmbDegeneration.addItem(bundle.getString("dlgCrossSec_msg29"));
```

O trecho a seguir ilustra o método ***addCrossSection()***. Quando o botão *Adicionar* do diálogo é pressionado (Figura C.9), esse método cria um sub-diálogo específico conforme o tipo de seção escolhido.

```
private void addCrossSection() {
switch (cmbDegeneration.getSelectedIndex()) {
case 0: // Generic
    SectionGenericDialog dlg0 = new SectionGenericDialog(...);
case 1: // Retangular
    SectionRetangularDialog dlg1 = newSectionRetangularDialog(...);
case 2: // Circular
    SectionCircularDialog dlg2 = newSectionCircularDialog(...);
case 3: // Shape I
    SectionShapeIDialog dlg3 = newSectionShapeIDialog(...);
case 4: // Discrete: not implemented yet
case 5: // Thickness
    SectionThicknessDialog dlg5 = new SectionThicknessDialog(...);
case 6: // Layers (Thickness)
    SectionLayersDialog dlg6 = newSectionLayersDialog(...);
case 7: // Microplane
    SectionMicroplaneDialog dlg7 = newSectionMicroplaneDialog(...);
```

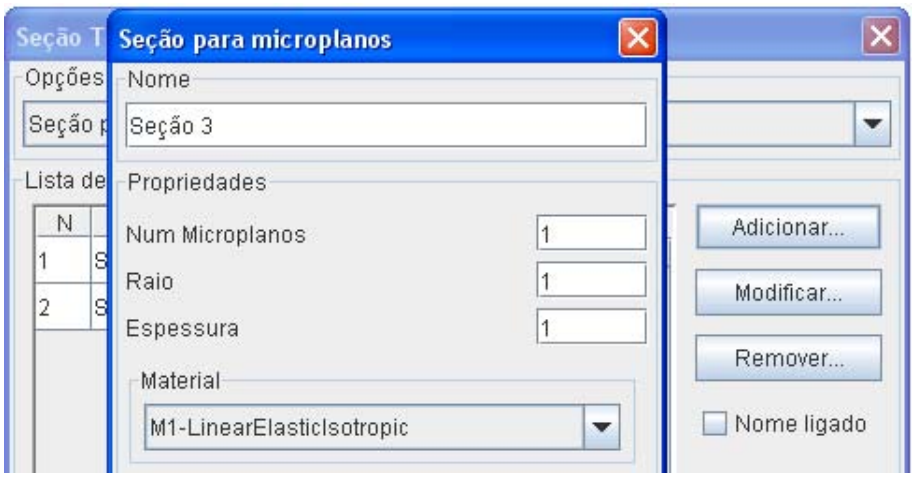


Figura C.9: Diálogo para seção tipo “Microplanos”

C.6 Inclusão de Novos Tipos de Solução

A classe ***ProcessorFullDialog*** (projeto *...ui.rich.full*) representa um diálogo através do qual devem ser informados parâmetros para o processamento do modelo.

As referências para novos *"Tipos de Análise"* (Figura C.10) devem ser adicionadas no método ***getSolutionAnalysisPanel()***.

Figura C.10: Componente para escolha do Tipo de Análise.

Um trecho do método está reproduzido abaixo:

```
private JPanel getSolutionAnalysisPanel() {
    cmbSolutionAnalysis = newJComboBox();
// Linear Static Analysis
cmbSolutionAnalysis.addItem(bundle.getString("dlgSolution_msg1"));
// Physically Non-Linear Static Analysis
cmbSolutionAnalysis.addItem(bundle.getString("dlgSolution_msg2"));
// Geometrically Non-Linear Static Analysis (Framed)
cmbSolutionAnalysis.addItem(bundle.getString("dlgSolution_msg51"));
// Geometrically Non-Linear Static Analysis (TL)
cmbSolutionAnalysis.addItem(bundle.getString("dlgSolution_msg3"));
// Geometrically Non-Linear Static Analysis (UL)
cmbSolutionAnalysis.addItem(bundle.getString("dlgSolution_msg4"));
// Dynamic Linear Analysis
cmbSolutionAnalysis.addItem(bundle.getString("dlgSolution_msg5"));
```

Já as referências para os *"Tipos de Solução"* (Figura C.11), devem ser adicionadas no método ***updateSolutionTypeCombo(String solCase)***. Um trecho do código está ilustrado abaixo:

```
private void updateSolutionTypeCombo(String solCase) {
cmbSolutionType.removeAllItems();
// Linear Static Analysis
if (solCase.equals(bundle.getString("dlgSolution_msg1"))) {
    cmbSolutionType.addItem(bundle.getString("dlgSolution_msg1"));
//Dynamic Linear Analysis
} else if (solCase.equals(bundle.getString("dlgSolution_msg5"))) {
    cmbSolutionType.addItem(bundle.getString("dlgSolution_msg14"));
    cmbSolutionType.addItem(bundle.getString("dlgSolution_msg15"));
    cmbSolutionType.addItem(bundle.getString("dlgSolution_msg16"));
    cmbSolutionType.addItem(bundle.getString("dlgSolution_msg17"));
    cmbSolutionType.addItem(bundle.getString("dlgSolution_msg18"));
    cmbSolutionType.addItem(bundle.getString("dlgSolution_msg19"));
```

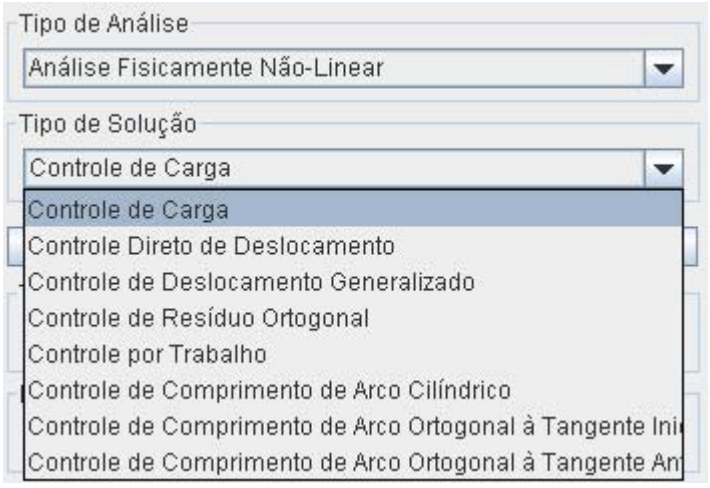


Figura C.11: Componente para escolha do Tipo de Solução.

Com a inclusão de novos tipos de análise e novos tipos de solução, o método ***checkAnalysisModel(...)*** (trecho de código abaixo) também dever ser atualizado.

```
private boolean checkAnalysisModel(...) {
// Linear Static Analysis
if(solutionAnalysis.equals(bundle.getString("dlgSolution_msg1"))) {
    isPossible = true;
    problemDriverKey = "All";
// Physically Non-Linear Static Analysis
} else if
    (solutionAnalysis.equals(bundle.getString("dlgSolution_msg2"))) {
    isPossible = true;
    problemDriverKey = "PhysicallyNonLinear";
```

O botão *"Verificar Solução X Modelo de Análise"* (Figura C.12) dispara o método *checkAnalysisModel(...)* para verificar se a solução é possível para o modelo de análise adotado.

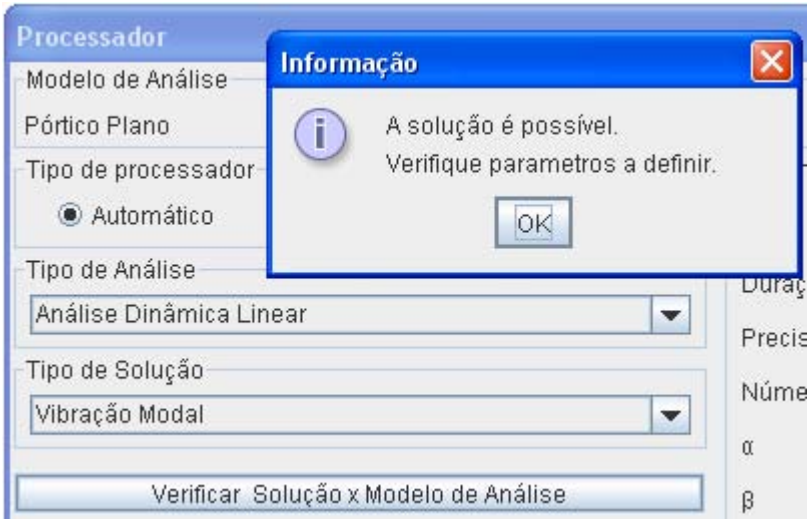


Figura C.12: Compatibilidade entre a Solução e o Modelo de Análise.

Referências Bibliográficas

Almeida, M. L., 2005. Elementos finitos paramétricos implementados em java. Dissertação de Mestrado, Universidade Federal de Minas Gerais, Belo Horizonte.

Alvim, P., 2003. *Open source: Os novos desafios de negócios e a indústria de TI.* Developers Magazine. Citado em Almeida (2005).

Azevedo, E. e Conci, A., 2003. *Computação Gráfica - Teoria e Prática.* Ed. Campus.

Baumgart, B., 1974. Geometric Modelling for Computer Vision. Tese de Doutorado, Stanford University.

Brugiolo, M. A., 2004. Geração de Malhas Bidimensionais de Elementos Finitos Baseada em Mapeamento Transfinitos. Dissertação de Mestrado, UFMG - Universidade Federal de Minas Gerais, Belo Horizonte.

Buschmann, F., Meunier, R., Rohnert, H., Sommerland, P. e Stal, M., 1995. *Pattern-Oriented Software Architecture: A System of Patterns.* Vol. 1, Wiley.

Camara, L. S., 2007. Um Serviço WEB para o Método dos Elementos Finitos. Dissertação de Mestrado, UFMG - Universidade Federal de Minas Gerais, Belo Horizonte.

Camarão, C. e Figueiredo, L., 2003. *Programação de Computadores em JAVA.* Ed. LTC.

Deitel, H. M. e Deitel, P., 2003. *JAVA Como Programar.* 4ª edição, Ed. Bookman.

Del Savio, A. A., Santi, M. R. e Martha, L. F., (2004), Traçado de curvas offset para auxilio na geração de malhas, *in* 'XXV Congresso Ibero Latino Americano de Mecânica Computacional - CILAMCE', Recife, Pernambuco.

Fernández, L. H. S., (2008), Processador interativo para modelos estruturais do método dos elementos finitos, *in* 'VIII Simpósio Mecânica Computacional - SIMMEC', Belo Horizonte, Minas Gerais.

Ferro, N. C. P., (2001), Um aplicativo acadêmico para análise de estruturas reticulares espaciais, *in* 'XXIX Congresso Brasileiro de Ensino de Engenharia - COBEMGE', Porto Alegre, Rio Grande do Sul.

Figueiredo, L. H. e Carvalho, P. C. P., 1991. *Introdução à Geometria Computacional*. IMPA, Rio de Janeiro.

Fonseca, G. L., 1989. Geração de Malhas através de Mapeamentos Transfinitos. Dissertação de Mestrado, PUC-RJ - Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro.

Gamma, E., Helm, R., Jonhson, R. e Vlissides, J., 1995. *Design Patterns - Element of Reusable of Object Oriented Software*. Addi.

Goodrich, M. T. e Tamassia, R., 2002. *Estruturas de Dados e Algoritmos em JAVA*. Ed. Bookman.

Kaefer, L. F., 2000. Desenvolvimento de uma Ferramenta Gráfica para Análise de Pórticos de Concreto Armado. Dissertação de Mestrado, Escola Politécnica da Universidade de São Paulo, São Paulo.

Liesenfeld, R., 2002. *Processando XML em JAVA*. Java Magazine, Edição 2, páginas 48 a 52.

Lozano, F., 2003. *Entenda a tecnologia XML*. Revista do Linux, Edição 48, páginas 42 a 49.

Mäntylä, M., 1987. *An Introduction To Solid Modeling*. Computer Science Press, Helsinki University of Technology.

Nicoliello, R. M., 2005. Aplicação Gráfica Interativa para Ensino do Método dos Elementos Finitos. Dissertação de Mestrado, UFMG - Universidade Federal de Minas Gerais, Belo Horizonte.

Penna, S. S., 2007. Pós-processador para resultados de análise não-linear de modelos do método dos elementos finitos. Dissertação de Mestrado, Universidade Federal de Minas Gerais, Belo Horizonte.

Santos, R., 2003. *Introdução à Programação Orientada a Objetos Usando JAVA*. Ed. Campus.

Souza, W. C. A. e Lima, L. C. A., 2008. *Explicando Padrões de Projeto*. Java Magazine, Edição 56, páginas 38 a 42.