PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

Danilo Silva Bomfim

Uma estratégia de modelagem aberta e extensível para a criação de modelos de subdivisões planares para mecânica computacional

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre em Engenharia Civil pelo Programa de Pós-graduação em Engenharia Civil, do Departamento de Engenharia Civil da PUC-Rio.

Orientador: Luiz Fernando Campos Ramos Martha

Rio de Janeiro

Janeiro de 2022

PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

Danilo Silva Bomfim

Uma estratégia de modelagem aberta e extensível para a criação de modelos de subdivisões planares para mecânica computacional

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós-graduação em Engenharia Civil da PUC-Rio. Aprovada pela Comissão Examinadora abaixo.

Prof. Luiz Fernando Campos Ramos Martha
Orientador
Departamento de Engenharia Civil – PUC-Rio

Prof. Luiz Carlos Wrobel
Departamento de Engenharia Civil – PUC-Rio

Prof. André Maués Brabo Pereira
Universidade Federal Fluminense

Rio de Janeiro, 25 de Janeiro de 2022

Danilo Silva Bomfim

Graduou-se em Engenharia Civil na Universidade Estadual de Santa Cruz (UESC) em 2020. Iniciou o curso de mestrado em Engenharia Civil (área de estruturas) na Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio) em 2020. Atua na linha de pesquisa da Computação Gráfica aplicada a Engenharia Civil e no desenvolvimento de ferramentas para modelagem geométrica de sólidos.

Ficha Catalográfica

Bomfim, Danilo Silva

Uma estratégia de modelagem aberta e extensível para a criação de modelos de subdivisões planares para mecânica computacional / Danilo Silva Bomfim; orientador: Luiz Fernando Campos Ramos Martha. – 2022.

157 f. : il. color. ; 29,7 cm

Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Engenharia Civil e Ambiental, 2022.

Inclui bibliografia

1. Engenharia Civil e Ambiental - Teses. 2. Modelagem de subdivisões planares. 3. Estrutura de dados topológica. 4. Half-Edge. 5. Representação por fronteira. 6. Sólidos 2-manifold. I. Martha, Luiz Fernando Campos Ramos. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Engenharia Civil e Ambiental. III. Título.

CDD:624

Agradecimentos

À Deus que me deu forças e me possibilitou o desenvolvimento deste trabalho.

Ao meu orientador Professor Luiz Fernando Martha pelo estímulo e pelo conhecimento transmitido na orientação deste trabalho.

Ao Professor André Maués Pereira que foi essencial para o desenvolvimento deste trabalho.

À PUC-Rio, pelos auxílios concedidos, sem os quais este trabalho não poderia ter sido realizado.

A todos os membros do grupo de pesquisa que contribuíram para a elaboração deste trabalho.

Aos membros da banca, pelas diversas sugestões feitas na redação final da dissertação.

Aos meus pais, Regivaldo e Alane, pelo apoio e atenção incondicional em todos esses anos.

A todos os amigos e familiares que de uma forma ou de outra me estimularam ou me ajudaram.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

Resumo

Bomfim, D. S.; Martha, L. F. **Uma estratégia de modelagem aberta e extensível para a criação de modelos de subdivisões planares para mecânica computacional**. Rio de Janeiro, 2022. 157p. Dissertação de Mestrado - Departamento de Engenharia Civil, Pontifícia Universidade Católica do Rio de Janeiro.

Este trabalho apresenta uma estratégia de modelagem aberta e extensível, desenvolvida em Python, para a criação de modelos de subdivisões planares. A estratégia se dá na forma de uma biblioteca de modelagem geométrica, denominada HETOOL, desenvolvida no trabalho e de uso genérico, baseada na conhecida e consagrada estrutura de dados topológica Half-Edge. Além de considerar os aspectos topológicos e geométricos da modelagem, a estratégia também permite a configuração pelo usuário final dos atributos de simulação. Essas características, somadas à disponibilização do código fonte, conferem um caráter útil e relevante para o desenvolvimento de ferramentas educacionais para modelagem em mecânica computacional. Para demonstrar a aplicabilidade da estratégia proposta, foi desenvolvido um aplicativo, denominado de *Finite Element Method Educational Computer Program* (FEMEP), que permite a criação de modelos bidimensionais de elementos finitos, com geração de malhas por região, para diversos tipos de simulação de mecânica computacional. O pacote desenvolvido apresenta uma modelagem iterativa e dinâmica que realiza a interseção automática entres os elementos geométricos modelados. O HETOOL oferece várias funcionalidades e facilidades ao usuário, permitindo o uso do pacote mesmo sem o usuário ter conhecimento sobre os conceitos topológicos envolvidos na implementação dessa estrutura de dados. O pacote possibilita a criação e configuração atributos de forma simples e rápida a partir de um arquivo no formato JSON. Essa versatilidade na criação atributos permite a aplicação deste pacote na resolução de vários problemas presentes na engenharia e em outras áreas do meio científico.

Palavras-chave

Modelagem de Subdivisões Planares; Estrutura de Dados Topológica; Half-Edge; Representação por Fronteira; Sólidos *2-manifold*.

Abstract

Bomfim, D. S.; Martha, L. F. (Advisor). **An open and extensible modeling strategy for creating planar subdivision models for computational mechanics**. Rio de Janeiro, 2022. 157p. Dissertação de Mestrado - Departamento de Engenharia Civil, Pontifícia Universidade Católica do Rio de Janeiro.

This work presents an open and extensible modeling strategy, developed in Python, for creating planar subdivision models. The strategy takes the form of a geometric modeling library called HETOOL, developed in the work and of general use, based on the well-known and renowned Half-Edge topological data structure. In addition to considering the topological and geometric aspects of the modeling, a strategy also allows for an end-user configuration of simulation attributes. These characteristics, added to the availability of the source code, provide a useful and relevant tool for the development of educational tools for modeling computational mechanics. To demonstrate the applicability of the proposed strategy, an application was developed, called the *Finite Element Method Educational Computer Program* (FEMEP), which allows the creation of two-dimensional finite element models, with mesh generation per region, for various types of mechanics simulation computational. The developed package presents iterative and dynamic modeling that performs an automatic intersection between the modeled geometric elements. HETOOL offers several functions and facilities to the user, allowing the use of the package even without the user having knowledge about the topological concepts involved in the implementation of this data structure. The package makes it possible to create and configure attributes simply and quickly from a file in JSON format. This versatility in creating attributes allows the application of this package to solve several problems present in engineering and in other areas of the scientific environment.

Keywords

Modeling Planar Subdivisions; Topological Data Structure; Half-Edge; Boundary Representation; 2-manifold Solids.

Sumário

Lista de Figuras

1
Introdução

A modelagem de sólidos se tornou uma parte fundamental para o desenvolvimento de projetos e pesquisas no meio científico e em aplicações práticas da Engenharia. A modelagem de sólidos consiste na representação geométrica de objetos reais a partir do armazenamento das propriedades físicas e geométricas destes elementos.

Várias aplicações nas áreas da Engenharia, Mineração, Geologia, Cartografia e Geografia trabalham com representações simplificadas em modelos bidimensionais, que são essencialmente subdivisões planares do espaço. Como exemplos destas aplicações computacionais tem-se os geradores de malhas de elementos finitos, os simuladores de escavação subterrânea, os simuladores de transformações geológicas, as representações de mapas cartográficos e os sistemas de coordenadas geográficas [1].

A Figura 1.1 ilustra um modelo de uma peça mecânica com condições de apoio e carga aplicada e a Figura 1.2 mostra um modelo de subsuperfície como exemplos de modelos de subdivisão planar. No primeiro exemplo, o domínio do modelo é subdivido em regiões com o objetivo de explorar diferentes tipos de algoritmos de geração de malha em cada região. No segundo exemplo, as regiões estão associadas a diferentes propriedades de camadas geológicas.

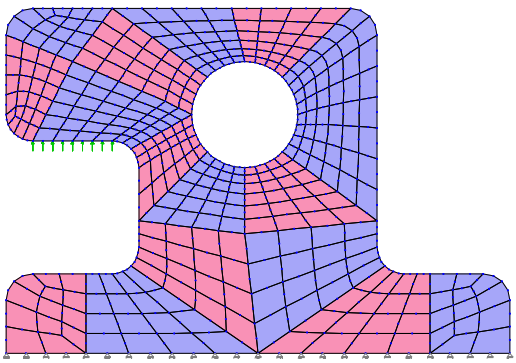


Figura 1.1 - Modelo bidimensional de uma peça mecânica com uma malha de elementos finitos [2]

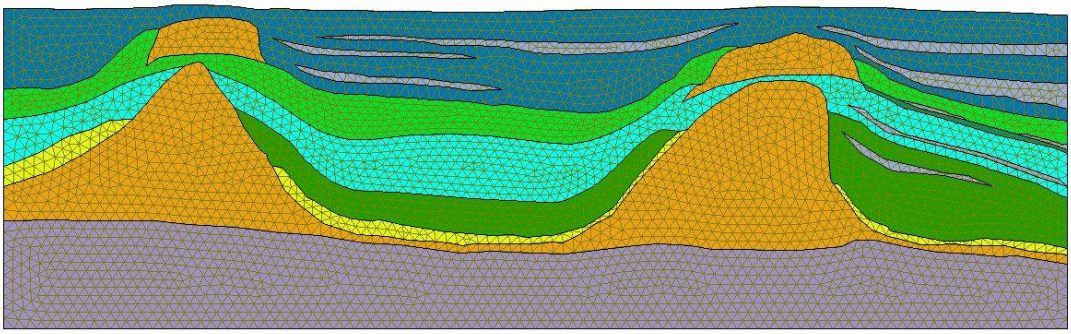


Figura 1.2 - Modelo de subsuperfície como exemplo de subdivisão planar [3]

Um exemplo genérico de subdivisão planar está apresentado na Figura 1.3. Nesta subdivisão, o plano é dividido em três regiões (ou faces) denominadas por f_1, f_2 e f_3 e uma região infinita f_{ext} que não apresenta uma fronteira externa definida. Cada região, mesmo a infinita, é limitada por um conjunto de segmentos de curva (arestas) que, por sua vez, são delimitadas por um par de pontos (vértices) não necessariamente distintos [4].

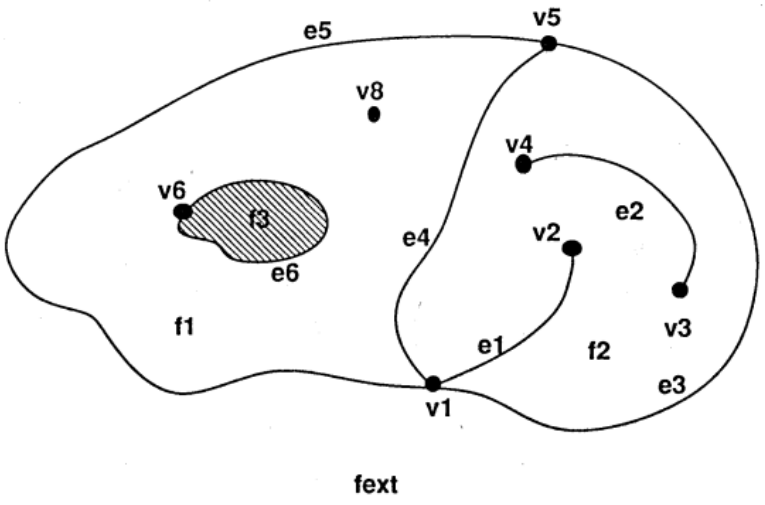


Figura 1.3 - Subdivisão planar [4]

Existem várias técnicas e métodos que podem ser utilizados para o gerenciamento das informações de uma subdivisão planar. Esse conjunto variado de métodos pode se distinguir na eficiência, na quantidade de memória utilizada para o armazenamento dos dados e nos tipos de informações a serem guardadas.

Dentre os tipos de técnicas utilizadas no gerenciamento das informações topológicas e geométricas, as técnicas baseadas na representação de fronteira (B-Rep) de um sólido são bastante utilizadas devido à flexibilidade que oferecem em descrever um sólido em termos de vértices, arestas e faces. Dessa forma, as subdivisões planares podem ser modeladas como a superfície de um sólido planificado. Isso justifica o aparecimento da face infinita que não apresenta uma fronteira externa definida.

Devido às facilidades e vantagens oferecidas pelas técnicas baseadas na representação de fronteira de sólidos, ao longo dos anos foram desenvolvidas várias estruturas de dados baseadas no padrão de representação B-Rep. Dentre essas estruturas de dados que utilizam das técnicas de representação de fronteira para a modelagem dos sólidos, a Half-Edge, desenvolvida por Mäntylä [5], é uma estrutura bem conhecida e bastante utilizada.

A estrutura de dados Half-Edge foi desenvolvida para representar superfícies de sólidos *2-manifold*. É comum definir um sólido *2-manifold* como um sólido cujas fronteiras são superfícies fechadas que são variedades (*manifolds*) de dimensão 2. Uma superfície é denominada 2-variedade ou *2-manifold* (termo mais utilizado na área de modelagem geométrica) quando a vizinhança de qualquer dos seus pontos se parece localmente com um espaço Euclidiano de dimensão 2. Neste trabalho a estrutura de dados Half-Edge é utilizada para representar uma subdivisão planar, que pode ser interpretada como uma superfície *2-manifold* planificada.

Uma das vantagens mais importantes no uso de estruturas de dados topológicas é que elas podem ser aplicadas como base de modelagem para diversos problemas no campo científico. Existem muitas outras vantagens oferecidas pelo uso de estruturas de dados topológicas, como eficiência e velocidade no processamento das informações e economia no uso de memória. Nos últimos anos diversos trabalhos têm sido publicados, tais como [6-16], aplicando ou otimizando estruturas de dados topológicas em projetos e pesquisas cientificas.

Conforme mencionado, diversos problemas podem ser modelados por estruturas de dados topológicas, considerando também as características geométricas do modelo. Entretanto, a topologia e a geometria não são suficientes para modelar completamente um problema de Engenharia ou Geologia. Uma

questão fundamental está nos atributos de modelagem associados às entidades topológicas e geométricas do modelo. A geometria e a topologia constituem os aspectos genéricos de um problema, mas o que realmente caracteriza e diferencia uma aplicação são os seus atributos de modelagem. Neste trabalho, além do tratamento topológico e geométrico de uma subdivisão planar, considera-se o gerenciamento dos atributos de modelagem de maneira configurável e extensível por um cliente final.

1.1
Objetivos e principais contribuições

Este trabalho tem como objetivo principal o desenvolvimento de uma biblioteca aberta e extensível, desenvolvida em Python e denominada HETOOL, para a modelagem genérica de subdivisões planares. Essa biblioteca é baseada na estrutura de dados topológica Half-Edge e possibilita o gerenciamento de atributos de modelagem configurado pelo usuário final. Por ter um código aberto e disponível livremente pela internet, a biblioteca HETOOL pode ser estendida por qualquer programador cliente em suas próprias aplicações e, portanto, pode ser utilizada com fins educacionais nas áreas de computação gráfica e mecânica computacional.

O uso da biblioteca HETOOL tem como principal vantagem o encapsulamento da complexidade associada ao tratamento da topologia de uma subdivisão planar. Dessa maneira, o cliente da biblioteca lida com as entidades topológicas de uma subdivisão planar em um nível que pode ser considerado alto, pois a "costura" topológica das entidades geométricas (pontos, curvas e regiões) é realizada de forma automática e transparente. Neste processo de modelagem de alto nível, o usuário ou programa cliente manipula uma subdivisão planar pela inserção de curvas geométricas e eliminação de segmentos de curvas.

Embora já existam outros pacotes de código aberto voltados para a modelagem de sólidos bidimensionais, poucos apresentam uma facilidade e flexibilidade na modelagem geométrica e na criação de atributos. O uso do pacote permite uma modelagem geométrica iterativa e dinâmica apresentando interseção

automática de curvas e reconhecimento automático de regiões fechadas pelas curvas. A geometria das curvas é definida por linhas retas e linhas poligonais. É responsabilidade do cliente converter curvas paramétricas por linhas poligonais equivalentes.

Além do tratamento topológico e geométrico de uma subdivisão planar, a biblioteca HETOOL também gerencia genericamente atributos de simulação de um cliente. A configuração de um novo atributo é feita de uma forma bem simples a partir de um arquivo no formato JSON. Para adicionar um novo atributo a este pacote basta seguir um padrão específico adotado neste arquivo. A generalização dos atributos presentes neste pacote permite aplicar a biblioteca desenvolvida na modelagem e simulação de vários problemas da mecânica computacional. Além disso, é possível salvar e ler os modelos criados a partir de arquivos no formato JSON.

Para a demonstração do uso dessa biblioteca foi desenvolvido um aplicativo, denominado FEMEP, para a criação de modelos planos com curvas poligonais arbitrárias e várias regiões, e para geração de malhas de elementos finitos bidimensionais em cada região. O FEMEP utiliza o ambiente Qt para a criação da interface gráfica e a biblioteca OpenGL para a visualização do modelo. O modelador foi desenvolvido com viés acadêmico podendo ser utilizado para fins educacionais.

A interface do FEMEP foi parametrizada para aceitar os novos atributos criados o que elimina, dessa maneira, a necessidade de realizar configurações adicionais para a incorporação dos novos atributos. O aplicativo também oferece a capacidade de modelar simultaneamente vários modelos planos a partir de múltiplas telas de modelagem.

Outro aplicativo desenvolvido para a demonstração do uso desse pacote foi o programa *Hetool* que é um aplicativo educacional focado no processo de ensino dos conceitos básicos que envolvem a implementação da estrutura de dados topológica Half-Edge.

1.2
Trabalhos correlatos

A estratégia de modelagem desenvolvida neste trabalho é o resultado de diversos desenvolvimentos anteriores. Desde os trabalhos desenvolvidos por Campos [17, 18] e Cavalcanti [19] em 1991, já se almejava a criação de um ambiente de modelagem capaz de representar uma subdivisão planar arbitrária para servir como estrutura básica de um gerador de malhas bidimensionais de elementos finitos. Cavalcanti [4] em 1992, aprimorou e otimizou o processo de manutenção e representação das subdivisões arbitrarias no espaço Euclidiano bidimensional e tridimensional criando uma metodologia capaz de construir subdivisões espaciais consistentes com a topologia e geometria.

Logo em seguida, vários trabalhos foram desenvolvidos [20-22] visando a criação de um sistema extensível e configurável para simulação de problemas da mecânica computacional. Esses desenvolvimentos tinham como objetivo a criação de um ambiente capaz de realizar simulações de mecânica computacional que seja independente do problema e que possa ser facilmente estendido e configurado.

A biblioteca HETOOL é o resultado da convergência dos trabalhos anteriormente mencionados, utilizando a linguagem de programação Python e outros recursos mais modernos (JSON). A biblioteca HETOOL possibilita a criação de um ambiente de modelagem para representação de subdivisões planares e a extensão da biblioteca através da configuração de novos atributos de modelagem por meio de um arquivo JSON.

Atualmente existem vários *softwares* comerciais, tais como *Abaqus* e *Ansys*, focados na modelagem e simulação de problemas da mecânica computacional. O diferencial da biblioteca desenvolvida neste trabalho com relação a esses programas comerciais se dá na facilidade de modelagem e da disponibilidade de uso. Esses programas comerciais apresentam um processo iterativo de modelagem menos amigável com relação a proposta de modelagem desenvolvida neste trabalho. A biblioteca HETOOL, que é focada na construção de modelos bidimensionais, apresenta uma modelagem dinâmica por meio da interseção automática entre os

elementos geométricos modelados e o reconhecimento automático de regiões fechadas.

Outro diferencial da biblioteca HETOOL com relação a outros *softwares* é a de configuração de novos atributos de modelagem de uma maneira mais simples e rápida, podendo até ser realizado por usuários que não apresentam um perfil de programador. A flexibilização na configuração de novos atributos de modelagem permite a extensão e a aplicação desta biblioteca na simulação de diferentes problemas da mecânica computacional. Além disso, a biblioteca HETOOL é de código aberto o que possibilita modificações e extensões no código desenvolvido pelo cliente programador.

1.3
Organização da dissertação

Esse trabalho foi dividido em sete capítulos. O primeiro capítulo apresenta as motivações para o desenvolvimento do trabalho, as principais características e funcionalidades da biblioteca desenvolvida e a organização da dissertação.

Como a estrutura de dados adotada neste trabalho é baseada em uma estrutura de dados concebida para representar a superfície de um sólido, o Capítulo 2 faz um breve histórico da modelagem de sólidos apresentando os principais sistemas e estratégias de modelagem que surgiram ao longo dos anos.

O Capítulo 3 apresenta conceitos sobre topologia no contexto da modelagem geométrica de sólidos e descreve de forma geral as vantagens da sua utilização. Além disso, neste capítulo é realizado um comparativo entre a modelagem de sólidos *2-manifold* e *não-manifold*.

O Capítulo 4 descreve duas estruturas de dados topológicas (Winged-Edge e Half-Edge) bem conhecidas para representação de fronteira de sólidos *2-manifold*. Nesse capítulo também são apresentados os principais conceitos que envolvem a modelagem de sólidos *2-manifold* e os operadores que manipulam uma estrutura de dados topológica para representação de superfícies *2-manifold*, os chamados operadores de Euler.

O Capítulo 5 aborda sobre as principais características e a arquitetura da biblioteca HETOOL. Nesse capítulo é detalhado o funcionamento do pacote desenvolvido por meio de diagramas. Além disso, o Capítulo 5 apresenta as principais classes, métodos e atributos envolvidos na implementação da biblioteca HETOOL.

O Capítulo 6 apresenta exemplos de utilização da biblioteca HETOOL. É exemplificado como construir o modelo geométrico e como criar e configurar novos atributos a partir dos métodos presentes na biblioteca. Esse capítulo também descreve dois aplicativos com viés acadêmico, denominados FEMEP e *Hetool*, que foram desenvolvidos a partir da biblioteca. Além disso, é fornecido um link para o repositório da biblioteca HETOOL, onde pode ser encontrado o código fonte da biblioteca e exemplos de utilização deste pacote

O Capítulo 7 apresenta as considerações finais e principais contribuições deste trabalho. Nesse capítulo é resumida a importância e aplicabilidade da biblioteca desenvolvida bem como algumas sugestões de trabalhos futuros.

2
Modelagem geométrica de sólidos

A modelagem geométrica de sólidos é uma área central de pesquisa e desenvolvimento com diversas aplicações tais como design e fabricação de produtos, prototipagem eletrônica e programação de robôs [23]. Todas essas aplicações requerem a representação computacional dos objetos reais e o armazenamento das informações relacionadas à geometria e propriedades físicas destes objetos.

Modelos computacionais consistem em informações armazenadas na memória de um computador que podem ser utilizadas para realizar simulações de fenômenos físicos. A simulação computacional de um fenômeno físico envolve a implementação de modelos que sejam capazes de representar os aspectos relevantes da estrutura e do comportamento dos objetos envolvidos no fenômeno. A simulação computacional pode ser dívida em três fases: modelagem, análise e visualização dos resultados (Figura 2.1).

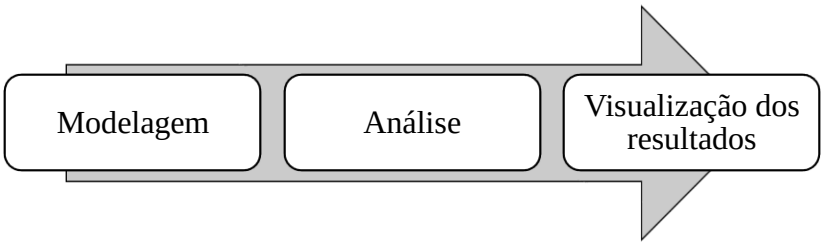


Figura 2.1 - Etapas da simulação computacional

A primeira fase de uma simulação computacional fundamenta-se na construção de um modelo que representa e descreve um objeto real. Diversas áreas do conhecimento produzem modelos para realização de simulações computacionais. Esses modelos podem ser classificados como modelos físicos, modelos matemáticos e modelos geométricos.

Um modelo matemático, em geral, é definido por um conjunto de equações diferenciais formuladas a partir de hipóteses simplificadoras sobre a natureza dos materiais que compõem o objeto. Um modelo físico descreve e caracteriza objetos

reais através de suas propriedades físicas. Já o modelo geométrico apresenta informações que descrevem a posição, dimensão e forma de um objeto real [24].

O processo de modelagem geométrica consiste na criação de representações de objetos reais de forma que seja possível atender a todas as questões geométricas e topológicas de uma maneira simples, rápida e eficiente. Inúmeras aplicações e análises só são possíveis devido ao sistema de modelagem geométrica que está presente em várias áreas como na produção cinematográfica, automotiva, aeroespacial e construção naval [23].

Atualmente, para facilitar a modelagem geométrica, utilizam-se ferramentas que auxiliam na criação e controle dos componentes do sólido modelado. Dentre essas ferramentas utilizadas na modelagem de sólidos, tem-se os programas computacionais de modelagem e as estruturas de dados. Muitos modeladores de sólidos em conjunto com estruturas de dados viabilizam a construção de modelos, o cálculo de propriedades físicas, a realização de análises matemáticas e mecânicas e o gerenciamento das informações dos objetos reais representados.

Conforme indica a Figura 2.1, a partir da construção do modelo, é possível prosseguir para as outras fases da simulação computacional viabilizando a análise e visualização dos resultados que, se referindo a mecânica computacional aplicada a problemas estruturais, pode ser a visualização dos deslocamentos, deformações e tensões.

2.1
Tipos de representação na modelagem de sólidos

A modelagem de sólidos é o resultado histórico de diversos desenvolvimentos convergentes. Esses desenvolvimentos incluem sistemas de desenho automático, gráficos e animação computacional [23]. Ao longo dos anos surgiram vários tipos de sistemas de modelagem, sendo cada um destes sistemas compostos por características particulares que variam de acordo com o nível de complexidade, as limitações do sistema, os tipos e a quantidade de informações a serem armazenadas. A seguir será descrito os principais sistemas de modelagem que surgiram ao longo

dos anos baseado no que cada um desses tipos de sistemas representam [23, 25 e 26].

Uma das primeiras técnicas de modelagem surgiu com o sistema de modelagem por arames (*wireframe*) onde apenas as arestas e os vértices dos objetos eram representados. Esse sistema de modelagem era limitado por permitir a existência de ambiguidades na representação do objeto. Um exemplo disso é mostrado na Figura 2.2 que ilustra um bloco com um orifício chanfrado no centro. Note que não é possível deduzir a direção do furo, pois poderia estar em qualquer uma das três direções principais, estabelecendo assim uma representação ambígua do objeto real.

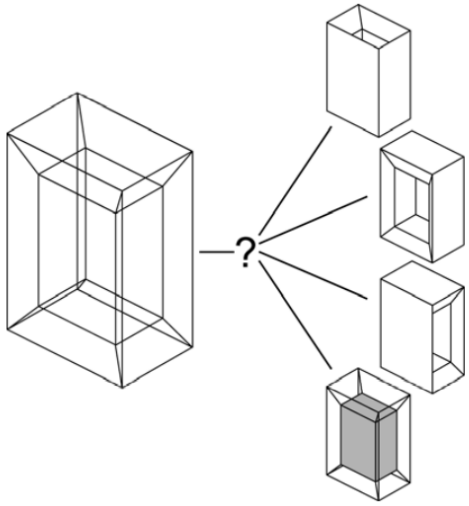


Figura 2.2 - Objeto de estrutura de arame ambíguo [5]

Em seguida, surgiu o sistema de modelagem por superfícies (*Surface modeling*) que era similar a modelagem por arames com a adição da descrição matemática da forma da superfície dos objetos representados. Após o sistema de modelagem por superfícies, surgiu o sistema de modelagem de sólidos (*Solid Modeling*) que é muito utilizada atualmente em diversas aplicações. Esta técnica de modelagem é muito vantajosa em relação as anteriores, essencialmente por garantir que qualquer modelo gerado irá formar objetos com volumes fechados e com contorno definido que se assemelham mais aos objetos reais.

Outra vantagem da modelagem de sólidos é a possibilidade de diferenciar o exterior de um sólido do seu interior, permitindo determinar propriedades físicas inerentes ao objeto como volume e centro de gravidade. Esta última característica amplia a capacidade de aplicação deste sistema de modelagem.

Existem formas distintas de representar as informações de um modelo geométrico pelo sistema de modelagem de sólidos. Os tipos de representação de sólidos mais comumente utilizados são: *Boundary represetantion* (B-Rep), *Cell decompositon* e *Constructive solid geometry* (CSG). A biblioteca desenvolvida neste trabalho utiliza um sistema de representação baseado em modelos de contorno (B-Rep) e, portanto, esse tipo de representação será apresentado com mais detalhes na Seção 2.1.3. Os demais tipos de representação serão brevemente discutidos nas Seções 2.1.1 e 2.1.2 e podem ser obtidos nas referências [23, 25, 27 e 28].

Após o sistema de modelagem de sólidos, surgiu o sistema de modelagem *não-manifold* (*Non-manifold Geometric Modeling*) que elimina algumas limitações presentes na modelagem de sólidos. Além disso, esse sistema possibilita a representação de objetos mais complexos e com estruturas internas. A distinção entre o sistema de modelagem *manifold* e *não-manifold* será abordada com mais detalhes na Seção 3.3.

2.1.1
Representação por decomposição (*Cell Decomposition*)

O método de decomposição (*Cell Decomposition*) consiste na divisão do espaço do objeto em elementos de uma forma simples (geralmente cubos ou tetraedros) e na representação do objeto na forma de uma coleção desses elementos. Este tipo de representação de sólidos desempenha um papel secundário na modelagem geométrica devido a certas limitações. Contudo, modelos de decomposição possuem muitas aplicações importantes, como por exemplo na área de análise numérica e em bancos de dados geográficos.

O esquema de decomposição mais simples é dado por uma subdivisão uniforme do espaço em uma malha de cubos de tamanho específicos. Todos os

cubos que cruzam o interior do sólido são marcados e o sólido é então representado pelo conjunto de cubos marcados. A Figura 2.3 ilustra este tipo de subdivisão. Este tipo de representação é aproximado e quanto menor for o número de cubos maior será a precisão da representação.

Inicialmente, apenas os cubos marcados precisam ser armazenados. A adjacência de dois cubos pode ser representada explicitamente ou pode ser inferida através da localização de um cubo no espaço. Representações deste tipo são comumente utilizadas em análises numéricas para a discretização de domínios.

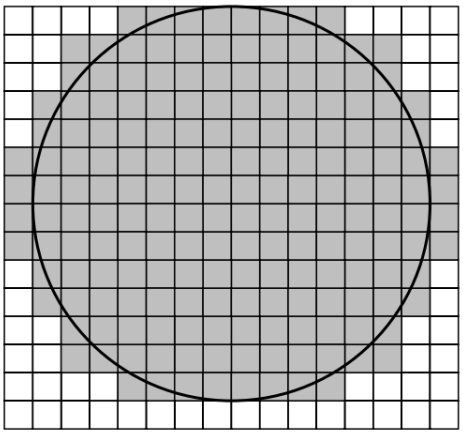


Figura 2.3 - Subdivisão uniforme do espaço [23]

Quando uma precisão maior é requerida, o número de cubos armazenados na memória pode ser muito grande para que essa estrutura de dados seja considerada apropriada. Para evitar esse problema, a utilização da subdivisão hierárquica em árvores, que subdivide o modelo em células de tamanho variável, o que pode ser muito eficaz. Em geral, as células de uma subdivisão hierárquica são quadrados no caso plano, resultando na subdivisão do tipo *Quadtree*, e cubos no caso tridimensional, resultando na subdivisão do tipo *Octree*. A Figura 2.4 mostra a representação de um disco por subdivisão do espaço utilizando o método *Quadtree*.

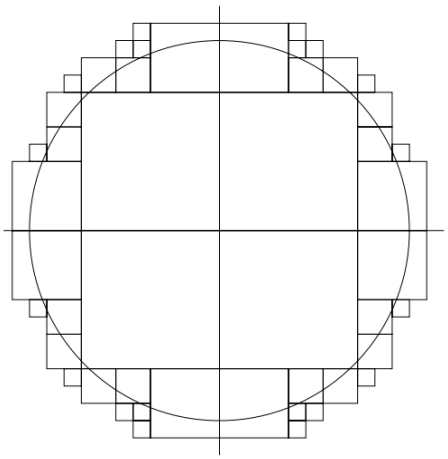


Figura 2.4 - Representação de um círculo por subdivisão do espaço em uma árvore *Quadtree* [23]

Existem alguns tipos de decomposição espacial em que as células não possuem uma relação específica com o espaço de coordenadas. Ao custo de adjacências mais complexas e maior processamento de formas geométricas, é possível subdividir o domínio do objeto em células com formas irregulares. Por exemplo, em um problema analisado pelo método dos elementos finitos, elementos triangulares e tetraédricos são utilizados, conforme ilustra a Figura 2.5, para subdividir um disco com células triangulares.

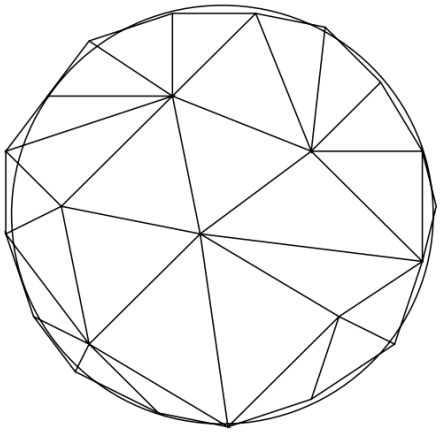
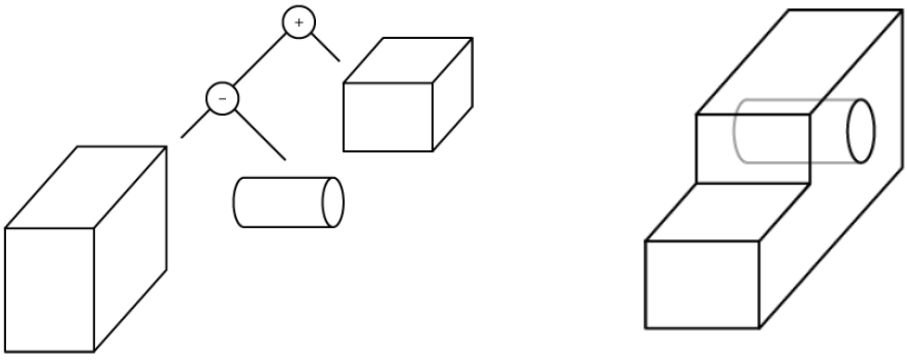


Figura 2.5 - Subdivisão irregular de um disco em células triangulares [23]

2.1.2
Representação por construção (*Constructive Solid Geometry*)

O método de construção (*Constructive Solid Geometry - CSG*) consiste na combinação de sólidos mais simples, chamados sólidos primitivos. A combinação destes sólidos mais simples é realizada por meio de operações booleanas e transformações geométricas aplicadas a eles. Os sólidos primitivos mais comuns utilizadas são os blocos no formato de paralelepípedos, o prisma triangular, a esfera, o cilindro, o cone e o toro. A Figura 2.6 ilustra o processo de modelagem de um sólido utilizando o método CSG.



(a) Sólidos primitivos (b) Modelo final

Figura 2.6 - Exemplo de modelo de construção [29]

Conforme pode ser visto na Figura 2.6.a, o processo de modelagem por construção ocorre por uma união ou diferença de sólidos primitivos que combinados dão origem ao modelo final representado na Figura 2.6.b.

Esses sólidos primitivos são genéricos no sentido de que representam formas parametrizadas que devem ser instanciadas pelo usuário nas dimensões definidas pelos valores dos parâmetros. Por exemplo, para obter um paralelepípedo de comprimentos de borda 1, 1 e 3, especifica-se o bloco (1; 1; 3), onde as dimensões são expressas em unidades a depender das convenções adotadas.

Outros tipos de sólidos primitivos podem ser permitidos, desde que possam ser generalizados da mesma forma, ou seja, a partir de parâmetros cujos valores devem ser definidos pelo usuário. Cada sólido primitivo possui um sistema de coordenadas local (Figura 2.7), a partir do qual os valores dos parâmetros podem ser instanciados. Os sistemas de coordenadas locais relacionam-se com o sistema de coordenadas global através de transformações geométricas que servem para mapear a localização das primitivas no espaço tridimensional.

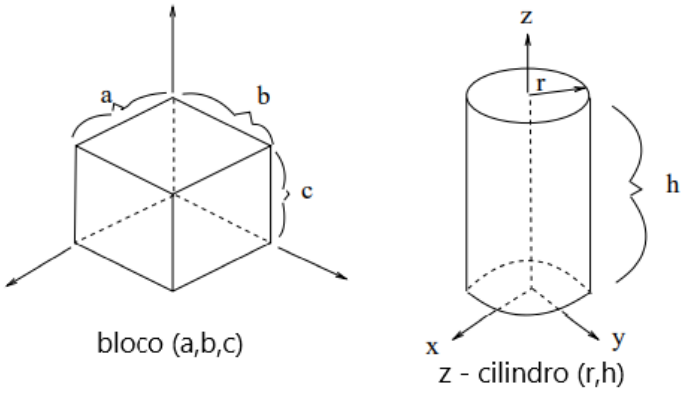


Figura 2.7 - Coordenadas locais para dois sólidos primitivos. Adaptado de [23]

Em geral, as primitivas devem possuir valores finitos para os seus parâmetros, ou seja, apenas sólidos com dimensões finitas podem ser representados. Casos particulares de sólidos primitivos que constituem semiespaços, ou seja, com dimensões infinitas, podem ser tratados no processo de definição de sólidos mais complexos.

2.1.3
Representação de fronteira (*Boundary Representation*)

Historicamente, o sistema *Boundary Representation* (B-REP) surgiu do aprimoramento dos modelos poliedrais usados em computação gráfica para representar objetos com remoção de linhas escondidas [24].

Um modelo baseado na representação de fronteira (modelo de contorno) descreve a superfície orientada de um sólido como uma estrutura de dados composta de vértices, arestas e faces. A convenção desta orientação permite determinar de que lado da superfície o interior do sólido está localizado. Assim, desde que a superfície e a geometria do modelo satisfaçam certos requisitos geométricos e topológicos, esse tipo de representação possibilita descrever o interior e exterior do sólido sem que haja ambiguidade [23].

Os modelos de contorno representam faces em termos de vértices explícitos da estrutura de dados, possibilitando muitas alternativas para representar a geometria e a topologia de um modelo de contorno. O contorno de um sólido tridimensional é uma superfície bidimensional que é usualmente representada como uma coleção de faces. Por sua vez, as faces do modelo são representadas em termos de curvas unidimensionais que definem as fronteiras destas faces [30].

A porção da superfície que forma a face é delimitada em termos de uma curva fechada que está sobre a superfície. Uma face pode ter várias curvas como fronteira desde que elas definam um objeto conexo, a Figura 2.8 ilustra exemplos disso. Os casos (a) e (c) representam duas faces distintas formando conjuntos bidimensionais não conexos. Os casos (d) e (e) representam faces cujas fronteiras se tocam. Usualmente o caso (d) seria considerado como a representação de uma face enquanto o caso (e) seria considerada como a representação de duas faces. O interior de (d) é dito conexo, mas o interior de (e) não é [5 e 24].

Nesse sistema de representação, as fronteiras que delimitam a face são representadas através de um conjunto de arestas. Arestas devem ser escolhidas de modo a ter uma representação conveniente, que em geral, pode ser feita através da utilização de equações paramétricas. Cada aresta é demarcada em termos de dois vértices localizados nas extremidades [5 e 24].

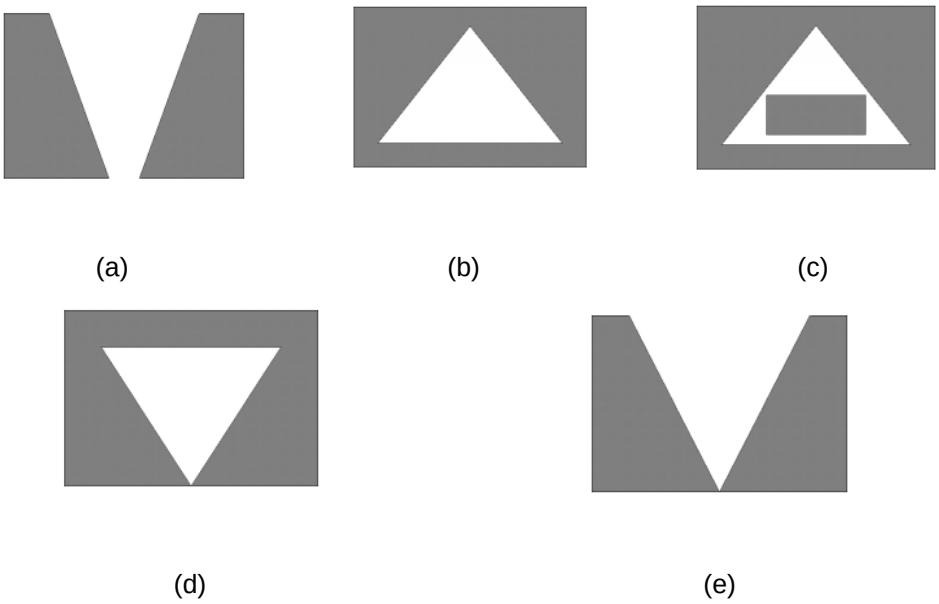


Figura 2.8 - Definição de uma face [24]

As informações topológicas de uma representação de fronteira podem estar centradas nas faces planares (polígonos), nos vértices ou nas arestas. Nas seções a seguir, descreve-se cada um desses tipos de representação a partir dos conceitos obtidos em [5 e 24].

2.1.3.1
Modelos de contorno baseados em polígonos

Um modelo de contorno que possui somente faces planares é denominado um modelo poliedral. Devido a todas as arestas de um poliedro serem segmentos de retas, é possível reduzir consideravelmente a estrutura de dados.

Nesse modelo as faces são representadas como polígonos onde cada polígono consiste em uma sequência de triplas ordenadas. Um sólido consiste em uma coleção de faces agrupadas em termos de uma tabela de identificadores ou uma lista encadeada de faces. Em alguns casos, até mesmo a informação de agrupamento é eliminada e as relações entre as faces são completamente implícitas. Esta representação é muito utilizada em meta-arquivos de sistemas gráficos.

2.1.3.2
Modelos de contorno baseados em vértices

No sistema descrito anteriormente, as coordenadas de um vértice aparecem o mesmo número de vezes do que o vértice aparece em uma face. Esta redundância pode ser removida introduzindo-se vértices como entidades independentes da estrutura de dados. Dessa maneira, para cada face são associados identificadores de vértices.

A Figura 2.9 mostra um cubo com vértices e arestas indicadas. Na Figura 2.10 apresenta-se a lista dos vértices de cada face deste cubo em uma ordem consistente, sendo está no sentido horário quando visto por fora do cubo da Figura 2.9. Esta orientação é útil em muitos algoritmos, por exemplo, na remoção de superfícies ou linhas escondidas. Isto permite a eliminação das faces de trás com base nos vetores normais das faces apontando consistentemente para fora do material. Nesse caso, as faces f_1, f_4 e f_5, indicadas na Figura 2.10, seriam descartadas caso o cubo fosse visualizado na orientação mostrada na Figura 2.9.

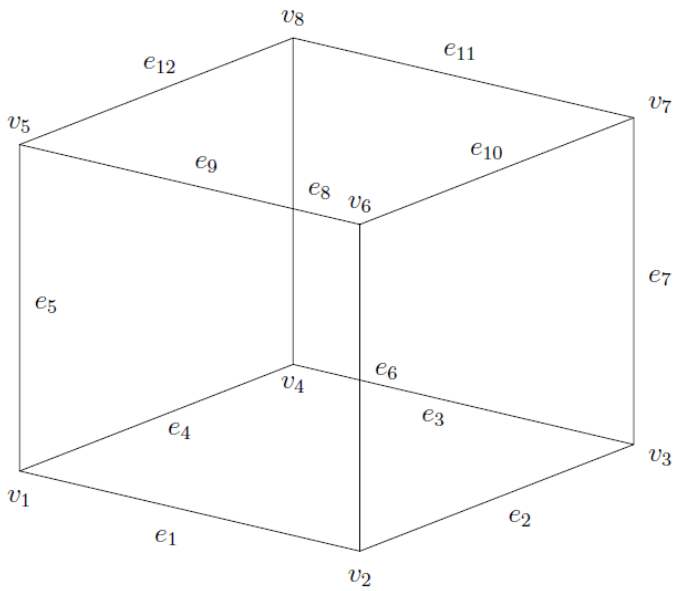


Figura 2.9 - Cubo com vértices e arestas enumerados [24]

Vértice	Coordenadas
v_1	$x_1\ y_1\ z_1$
v_2	$x_2\ y_2\ z_2$
v_3	$x_3\ y_3\ z_3$
v_4	$x_4\ y_4\ z_4$
v_5	$x_5\ y_5\ z_5$
v_6	$x_6\ y_6\ z_6$
v_7	$x_7\ y_7\ z_7$
v_8	$x_8\ y_8\ z_8$

Face	Vértices
f_1	$v_1\ v_2\ v_3\ v_4$
f_2	$v_6\ v_2\ v_1\ v_5$
f_3	$v_7\ v_3\ v_2\ v_6$
f_4	$v_8\ v_4\ v_3\ v_7$
f_5	$v_5\ v_1\ v_4\ v_8$
f_6	$v_8\ v_7\ v_6\ v_5$

Figura 2.10 - Informações topológicas de um modelo de contorno baseado em vértices

Os dados apresentados na Figura 2.10 não incluem qualquer informação geométrica da superfície do cubo da Figura 2.9. Quando todas as faces são planas, suas geometrias estão completamente definidas pelas coordenadas de seus vértices. Por outro lado, se a geometria das superfícies das faces fosse necessária por algum motivo, essa informação poderia ser associada com as faces.

Em geral, no modelo de contorno deve-se escolher como as informações devem ser armazenadas. Esse processo pode ser feito de forma explicita ou implícita (calculada posteriormente). Por exemplo, a inclusão explicita das coordenadas do vértice fornece implicitamente as equações das faces. O tratamento oposto em se armazenar explicitamente as equações das faces e deixar as coordenadas dos vértices de maneira implícita conduziria a um modelo do tipo *half-space* [24].

2.1.3.3
Modelos de contorno baseados em arestas

O modelo de contorno baseado em arestas representa uma face em termos de uma sequência fechada de arestas, essa sequência é usualmente chamada de laço

(*loop)*. Os vértices desse modelo são representados somente através das arestas. Esse tipo de modelo é bastante vantajoso devido ao número de entidades topológicas adjacentes ser limitado. Por exemplo, apenas duas faces incidem em uma aresta e uma aresta apresenta apenas dois vértices.

Esse modelo é muito utilizado devido a simplicidade e vantagens que ele oferece com relação a manipulação das informações topológicas, sobretudo as informações relacionadas com a adjacência dos elementos topológicos do modelo. Neste trabalho, a biblioteca HETOOL desenvolvida foi fundamentada em uma representação de contorno baseada em arestas, conforme é detalhado no Capítulo 5.

Na Figura 2.11, as informações apresentadas indicam a orientação de cada aresta. Por exemplo, a aresta e_1 é considerada orientada no sentido positivo do vértice v_1 para o vértice v_2. Assim, as faces estão orientadas de forma consistente, isto é, suas arestas estão listadas no sentido horário quando vistas por fora do cubo da Figura 2.9 .

Aresta	vértices
e_1	$v_1\ v_2$
e_2	$v_2\ v_3$
e_3	$v_3\ v_4$
e_4	$v_4\ v_1$
e_5	$v_1\ v_5$
e_6	$v_2\ v_6$
e_7	$v_3\ v_7$
e_8	$v_4\ v_8$
e_9	$v_5\ v_6$
e_{10}	$v_6\ v_7$
e_{11}	$v_7\ v_8$
e_{12}	$v_8\ v_5$

Vértice	Coordenadas
v_1	$x_1\ y_1\ z_1$
v_2	$x_2\ y_2\ z_2$
v_3	$x_3\ y_3\ z_3$
v_4	$x_4\ y_4\ z_4$
v_5	$x_5\ y_5\ z_5$
v_6	$x_6\ y_6\ z_6$
v_7	$x_7\ y_7\ z_7$
v_8	$x_8\ y_8\ z_8$

Face	Arestas
f_1	$e_1\ e_2\ e_3\ e_4$
f_2	$e_9\ e_6\ e_1\ e_5$
f_3	$e_{10}\ e_7\ e_2\ e_6$
f_4	$e_{11}\ e_8\ e_3\ e_7$
f_5	$e_{12}\ e_5\ e_4\ e_8$
f_6	$e_{12}\ e_{11}\ e_{10}\ e_9$

Figura 2.11 - Informações topológicas de um modelo de contorno baseado em arestas

No Capítulo 4 são descritos com mais detalhes dois tipos de estruturas de dados de modelos de contorno baseados em arestas, que são as mais utilizadas.

3
Topologia em modelagem geométrica

A topologia é um ramo da matemática comumente correlacionado com a geometria. A palavra topologia originou-se do grego *topos*, lugar, e *logos*, estudo, e é considerada uma das geometrias que estuda as transformações contínuas [31].

A geometria Euclidiana estuda as propriedades geométricas que são invariantes sob transformações rígidas ou isométricas, em outras palavras, que preservam distâncias entre pontos. Dessa forma, na geometria Euclidiana dois modelos são considerados equivalentes ou congruentes se um pode ser obtido do outro por uma ou mais isometrias, por exemplo, como uma translação e rotação no ambiente bidimensional [32].

Dentre as geometrias não-Euclidianas, a topologia caracteriza-se pelo estudo das propriedades dos modelos geométricos que são invariantes sob transformações topológicas. Essas transformações invariantes podem ser exemplificadas por ações tais como encolher, esticar e deformar, conhecidas como homeomorfismos que serão definidos posteriormente [26 e 32].

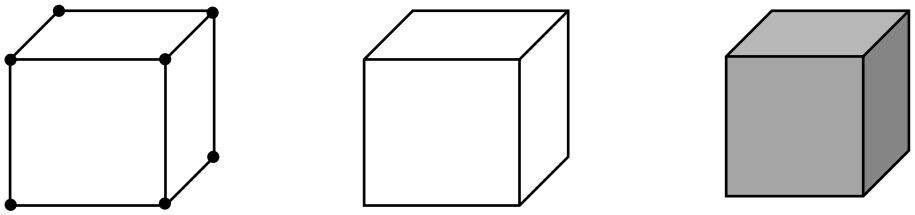
Existem diversas definições para a palavra topologia que variam de acordo com a área de estudo. Há muitas áreas de estudos relacionadas com o termo topologia tais como topologia algébrica, topologia combinatória, topologia diferencial, topologia de conjuntos de pontos, dentre outros tipos.

Diante das diversas áreas de estudo para o termo topologia, o enfoque neste trabalho será dado à topologia referente as entidades primitivas derivadas do modelo geométrico e suas relações de incidência e adjacência (topologia combinatória). Assim, o termo topologia será utilizado apenas para se referenciar a alguns conceitos relacionados com os temas supracitados.

As informações topológicas são incompletas na descrição de um objeto, ou seja, a topologia não possui informações suficientes para que o objeto possa ser modelado. Por outro lado, as informações geométricas de um objeto permitem que ele seja perfeitamente construído [25 e 26].

Logo, não é possível obter todas as informações geométricas a partir da topologia de um objeto, mas o contrário é válido, apesar de nem sempre as informações geométricas estarem numa forma conveniente para que se possam derivar as informações topológicas. As informações topológicas, em geral, auxiliam na representação dos objetos através de um sistema estruturado e organizado das informações necessárias para a modelagem do sólido.

A descrição topológica de modelos geométricos depende das relações de adjacências de suas entidades primitivas. Em geral, a topologia de modelos geométricos é constituída a partir da combinação de três entidades topológicas primitivas: vértices (V), arestas (E) e faces (F) como ilustra a Figura 3.1.



(a) Vértices (b) Arestas (c) Faces

Figura 3.1 - Constituintes básicos de um modelo geométrico

Conforme pode ser visto na Figura 3.1, um vértice é um ponto único associado a uma posição tridimensional única no espaço de modelagem. Uma aresta é um segmento de uma curva limitado por dois vértices. E a face pode ser definida como a região delimitada por um conjunto de vértices e arestas.

As relações de incidência e adjacência são muito úteis em sistemas de modelagem geométrica pois fornecem informações sobre quais elementos topológicos (vértices, arestas e faces) estão situados lado a lado de forma que se tocam. O uso destas relações permite ao sistema de modelagem uma maior fluidez na criação de elementos topológicos, na consulta e manipulação das informações do modelo, o que torna o sistema de modelagem mais organizado e eficiente.

A topologia é utilizada como meio para caracterizar as informações de um modelo geométrico em um conjunto de dados unificado, consistente e organizado

que possibilita obter, de maneira eficiente e rápida, informações relevantes sobre os elementos topológicos sem haver a necessidade de se fazer uma consulta global à geometria do modelo. Na manipulação de uma pequena porção de um sólido em estudo, é muito vantajoso obter diretamente as informações sobre as porções adjacentes sem a necessidade de passar por todas as informações associadas ao objeto. Isto melhora de forma significativa a eficiência computacional da modelagem dos objetos [25 e 26].

Utilizar a topologia como base de um sistema de modelagem consiste em explicitar as informações topológicas e criar uma estrutura de dados e algoritmos que utilizem essas informações. Em geral, os elementos topológicos são organizados segundo uma hierarquia decrescente, colocando os elementos de dimensão superior acima dos elementos de dimensão inferior [25].

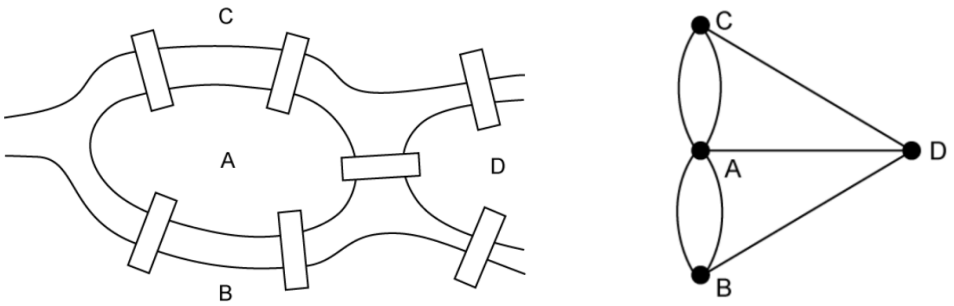
O uso da topologia proporciona ao sistema de modelagem estabilidade, organização sistemática e redução da quantidade de erros numéricos. A estabilidade deriva do fato que os elementos topológicos estão suscetíveis a menos variações do que os elementos geométricos, em outras palavras, existem diferentes formas de representar matematicamente os elementos geométricos enquanto a topologia costuma ser mais estável. Isto implica que um sistema baseado na representação topológica dos elementos sofre menos ajustes no caso da mudança do tipo de representação geométrica adotado [25 e 26].

O uso da topologia permite organizar os elementos hierarquicamente e separar as informações topológicas das informações geométricas tornando o sistema mais organizado e sistemático. A organização do sistema de modelagem é um fator essencial que facilita a criação, consulta, manipulação das informações de um sólido modelado. O uso da topologia também evita erros numéricos durante consultas. Para o uso das informações topológicas não é necessário tolerâncias e aproximações nos algoritmos geométricos, evitando erros de precisão numérica [25 e 26].

3.1
Grafos

A teoria dos grafos tem sido um dos meios mais simples e eficazes para a solução de problemas envolvendo modelagem geométrica. A teoria dos grafos surgiu na cidade de Königsberg em 1736 (atual Kaliningrado da Rússia) pelo matemático suíço Leonhard Euler [27].

Essa cidade possuía 7 pontes que conectavam duas ilhas sobre as quais surgiu o questionamento se era possível fazer um caminho por entre as pontes de modo que se passasse apenas uma vez em cada uma delas. Para resolver este problema Euler propôs um diagrama, conforme apresentado na Figura 3.2.b, contendo vértices e arestas. Posteriormente esse diagrama foi chamado de grafo.



(a) (b)

Figura 3.2 - Pontes de Königsberg em forma de grafo [27]

A seguir serão apresentados alguns conceitos referentes aos grafos que podem ser encontrados em diversas literaturas tais como [25, 26, 27 e 33]. Um Grafo (G) pode ser definido como um conjunto de elementos finitos formado por vértices, arestas (podem ser segmentos retos ou curvos) e uma função de incidência pela qual cada aresta do grafo G está associada a um par ordenado de vértices de G.

Quando grafos são apresentados por diagramas, os vértices são pontos e as arestas são linhas ligando esses pontos (Figura 3.3). Uma aresta é incidente com os vértices que se liga e quando uma aresta incide em um único vértice esta aresta é denominada de laço ou *loop* (e_4). Dois vértices são adjacentes quando estão ligados

por uma única aresta. Caso o vértice não apresenta nenhuma aresta incidente ele é chamado de vértice isolado (v_4).

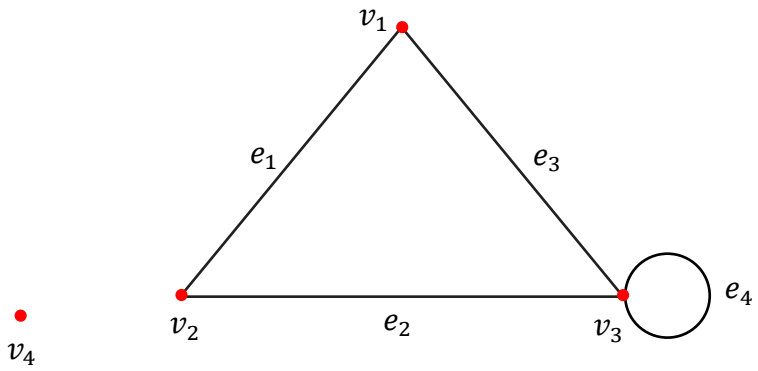


Figura 3.3 - Grafo G

A Tabela 1 mostra informações sobre a incidência das arestas que podem ser extraídas do Grafo G da Figura 3.3:

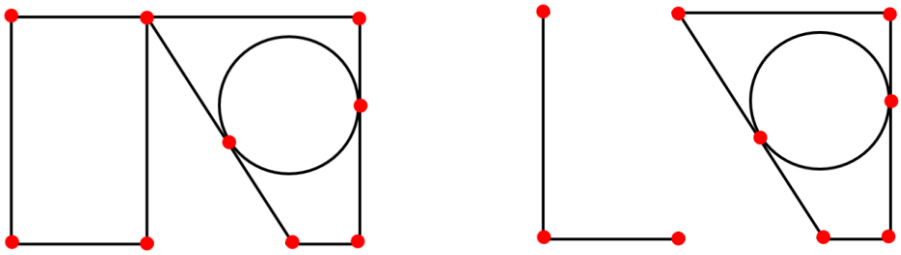
Tabela 1 - Incidência do Grafo G

Aresta	Vértices incidentes
e_1	v_1, v_2
e_2	v_2, v_3
e_3	v_3, v_1
e_4	v_3, v_3

Um caminho em um grafo é definido como uma sequência de vértices tal que cada um de seus vértices há uma aresta para o próximo vértice da sequência. Um grafo é considerado conexo se for possível estabelecer um caminho de um vértice para qualquer outro vértice desse mesmo grafo através do percurso das arestas.

A conectividade de um grafo é determinada pelo número mínimo de vértices que, quando removidos juntamente com suas arestas incidentes, resulta num grafo desconexo. Um grafo desconexo pode ser definido por um grafo que possua pelo

menos dois vértices que não estejam conectados através de algum caminho. A Figura 3.4 ilustra os conceitos de grafo conexo e desconexo.



(a) (b)

Figura 3.4 - (a) grafo conexo; (b) grafo desconexo

3.2
Conceitos topológicos e geométricos

O termo topologia também pode ser definido como o estudo das propriedades que são invariantes por homeomorfismos. Homeomorfismo entre espaços métricos (X, Y) é dado por uma função, $f: X \to Y$, bijetora e contínua cuja sua inversa também é contínua. Essa mesma definição pode ser aplicada de forma análoga aos espaços topológicos [34].

Dois sólidos podem ser considerados homeomorfos quando estes são topologicamente equivalentes, intuitivamente homeomorfismos podem ser encarados como deformações elásticas que preservam as relações de adjacência. A seguir, serão apresentados conceitos fundamentais para o entendimento da dissertação que foram obtidos a partir de [23, 25, 26 e 34].

Espaço métrico é um par (M, d) formado por um conjunto $M \neq \emptyset$ e uma função $d: M \times M \to \mathbb{R}$, que associa a cada par de pontos $x, y \in M$ um número real $d(x, y)$, chamado de métrica ou distância do ponto x ao ponto y, de tal modo que:

1. $d(x, y) > 0 \; se \; x \neq y \, , \forall \, x, y \in M$
2. $d(x, x) = 0, \; \forall \, x \in M$
3. $d(x, y) = d(y, x), \forall \, x, y \in M$ (simetria)

4. $d(x,y) \leq d(x,y) + d(y,z)\ , \forall\ x,y,z\ \in M$ (desigualdade triangular)

O espaço métrico mais conhecido é o espaço Euclidiano. A métrica euclidiana define a distância entre dois pontos como o comprimento que os conecta. Espaço Euclidiano é um espaço vetorial real de dimensão finita munido de um produto interno. Neste trabalho, assuma-se o espaço métrico como o espaço Euclidiano.

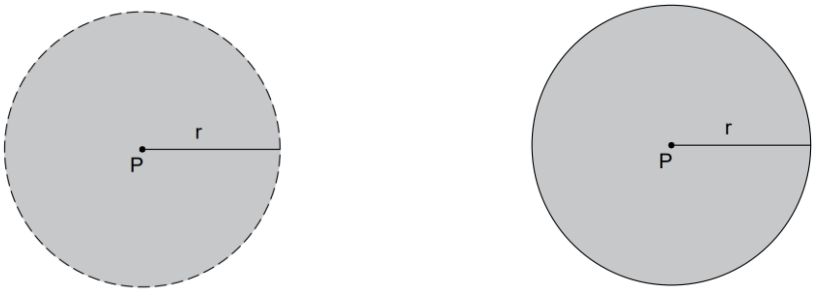
Espaço topológico pode ser definido como um par (X, τ), sendo X um conjunto e τ a topologia em X constituída por uma família de subconjuntos de X com as seguintes propriedades:

1. $\emptyset$ e X pertencem a τ
2. A união de uma família arbitrária de membros de τ pertence a τ
3. A interseção de uma família finita de membros de τ pertence a τ

Seja p um ponto de um espaço métrico (M, d) e $r > 0$ um número real, um disco aberto de centro p e raio r, indicada por $D(p,r)$ na Equação (*1*) são todos os pontos pertencentes a M cuja distância seja inferior a r.

$$D(p,r) = \{x\ \in M |\ d(x,y) < r\} \tag{1}$$

Em outras palavras, um disco aberto, pode ser descrito como a região de um espaço que se localiza no interior de um círculo de raio r centrado em um ponto p, excluindo-se a circunferência que a delimita. Já um disco fechado é descrito por essa região incluindo-se a circunferência que a delimita. A Figura 3.5 ilustra esses conceitos.



(a) (b)

Figura 3.5 - (a) disco aberto; (b) disco fechado.

Um subconjunto de um espaço topológico pode ser dito conexo em arco se para qualquer par de pontos desse subconjunto existe um caminho contínuo entre eles totalmente contido nesse subconjunto. Assim, é possível definir uma superfície como um espaço conexo em arco que é topologicamente bidimensional. Embora uma superfície seja localmente bidimensional, ela pode ser curva e existir geometricamente em um espaço tridimensional.

Uma superfície é dita limitada se toda ela puder estar contida em uma esfera aberta. A fronteira de uma superfície pode ser uma curva aberta ou fechada, ou mesmo um único ponto na superfície. Uma superfície é fechada se ela é limitada e não tem fronteira, como por exemplo uma esfera. Um plano é uma superfície ilimitada e sem fronteira, portanto não é uma superfície fechada.

Uma vizinhança de um ponto p, com respeito a um sólido *S*, no espaço tridimensional pode ser definida como sendo a interseção do sólido *S* com uma bola aberta centrada no ponto p e de raio infinitesimal ε. Se o ponto p estiver sobre superfície do sólido, então sua vizinhança é a metade da porção do espaço que compreende a bola aberta conforme ilustra Figura 3.6. Um ponto de um espaço topológico é considerado um ponto de aderência de um subconjunto desse espaço se toda vizinhança desse ponto contém pelo menos um ponto desse subconjunto.

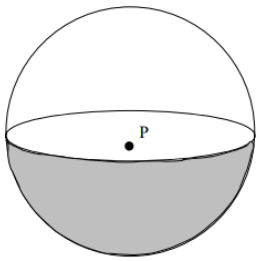


Figura 3.6 - Vizinhança de um ponto na superfície de um sólido fechado [23]

Um subconjunto A de um espaço topológico é fechado quando ele contém todos os seus pontos de aderência. O fecho de um subconjunto A de um espaço topológico, denotado por $F(A)$, é o conjunto de todos os pontos de aderência de A. O interior desse subconjunto denotado por $I(A)$, é o conjunto de pontos de A que não são pontos de aderência do complemento de A. A fronteira de A, denotado por

∂A, é o conjunto de pontos do espaço topológico que são pontos de aderência de A e do complemento de A.

Um subconjunto A de um espaço topológico é conexo quando pode ser dividido em dois subconjuntos B e C com $A = B \cup C$, $B \neq \emptyset$ e $C \neq \emptyset$ de forma que:

1. $\exists b \in B$, com $b \in F(C)$
2. $\exists c \in C$, com $c \in F(B)$

Um subconjunto de um espaço topológico é dito limitado quando ele está contido numa bola aberta. Um conjunto fechado e limitado é dito compacto.

Uma superfície *2-manifold* (variedade de dimensão 2) é uma superfície topologicamente conexa. Uma superfície é dita *2-manifold* se a vizinhança de qualquer ponto na superfície é homeomorfa a um disco aberto bidimensional.

A vizinhança de qualquer ponto de uma superfície *2-manifold*, tal como mostra a Figura 3.6, tem exatamente duas partes: uma no interior e outra no exterior do sólido. Qualquer configuração geométrica que resulte na vizinhança de um ponto com uma parte apenas ou com mais de duas partes não é uma superfície *2-manifold*. Uma superfície planar é um *2-manifold* se for ilimitada e pode ser interpretada como a superfície de uma esfera de raio infinito.

Um grafo pode ser dito imerso em uma superfície se for desenhado na superfície de modo que duas arestas não se cruzem, exceto em seus vértices incidentes. Um grafo planar é aquele que pode ser imerso numa superfície planar. Um grafo também pode ser imerso num espaço tridimensional, contendo ou não superfícies limitadas, desde que as propriedades de não-interseção sejam respeitadas, isto é, desde que nenhum par de elementos se intercepte a não ser em elementos de fronteira de mais baixa dimensão comuns.

Faces são subconjuntos conexos da superfície definida por um grafo imerso em uma superfície. Cada face é um componente conexo do conjunto obtido pela subtração dos vértices e arestas do grafo imerso da superfície. A fronteira de uma face corresponde a todas aquelas arestas e vértices do grafo imerso que tocam a face em toda a sua extensão. A face não contém a sua fronteira.

Uma face é simplesmente conexa quando possui uma fronteira única e conexa. Uma face multiplamente conexa possui uma fronteira formada por dois ou mais componentes desconexos, como no caso de uma face com um orifício interno.

O grafo imerso em uma superfície planar *2-manifold* (ilimitada) é, neste trabalho, referido como uma "subdivisão planar". Uma subdivisão planar *2-manifold* tem uma única face externa infinita. Essa face ilimitada não tem uma fronteira externa e pode ter uma fronteira interna, no caso de um grafo conexo, ou várias fronteiras internas, no caso de um grafo desconexo.

3.3
Modelagem geométrica *manifold* e *não-manifold*

O termo *manifold* na matemática refere-se a um espaço topológico que quando localmente próximo a um ponto se assemelha ao espaço Euclidiano. Um *n-manifold* é um espaço topológico com a propriedade que cada ponto tem uma vizinhança que é homeomorfa a um subconjunto aberto do espaço Euclidiano n-dimensional. Assim, um espaço *1-manifold* incluem linhas fechadas e um espaço *2-manifolds* incluem superfícies fechadas.

A conceituação do espaço topológico *manifold* é essencial para a geometria porque permite que estruturas complexas sejam descritas em termos de propriedades topológicas de espaços mais simples. Por conveniência, toda vez que for utilizado o termo *manifold* nesta dissertação se referirá ao espaço topológico 2-*manifold*.

Em um ambiente tradicional *manifold* de modelagem tridimensional, objetos sólidos apresentam uma fronteira *manifold*, consistindo em retalhos de superfícies (faces), arestas e vértices. Cada superfície separa o interior do sólido do ambiente exterior. Cada aresta é compartilhada por exatamente duas superfícies do sólido [35]. Esse sistema tradicional apresenta certas restrições topológicas que limitam a representação dos objetos. Essas restrições não estão presentes no sistema de modelagem *não-manifold* que amplia a capacidade de representação dos objetos.

A modelagem *não-manifold* no espaço tridimensional pode ser considerada como a decomposição deste espaço em conjuntos de elementos de dimensões zero, um, dois ou três, ou seja, vértices, arestas, faces e volumes, respectivamente [36]. *Não-manifold* é um termo da topologia geométrica que significa permitir que qualquer combinação de arestas, vértices, superfícies e volumes existam em um único corpo lógico [37]. Modelos *não-manifold* têm uma configuração que não pode ser desdobrada em uma peça plana contínua e, portanto, não são fabricáveis e não são fisicamente realizáveis em um único volume [10].

No sistema de modelagem *não-manifold* é possível que várias faces se encontrem em uma aresta, que arestas múltiplas se encontrem em um vértice e que arestas e vértices coincidam [26 e 37]. Além disso, as superfícies dos sólidos não-*manifold* podem apresentar fronteiras entre o interior do objeto e o exterior do ambiente ou entre duas células espaciais dentro do sólido (Figura 3.7.f). Na modelagem *não-manifold* não são exigidas algumas restrições topológicas presentes em sólidos *manifolds* possibilitando a construção de modelos com auto interseção e a utilização de operadores geométricos e topológicos mais complexos.

As representações *não-manifold* herdam em um só tipo de representação as características do outros sistemas de modelagem por arames, por superfícies e a modelagem *manifold*. Tal fator oferece vantagens significativas na criação, implementação e manutenção deste sistema de modelagem [26].

Na representação de objetos do tipo *não-manifold* a dependência entre vértices, arestas e faces entre si e entre seus vizinhos é inexistente. Com isso, não é preciso que um vértice faça parte de uma aresta, ou de uma face, e qualquer modificação aplicada apenas a um deles não necessariamente afetará a sua vizinhança.

De maneira geral, o sistema de modelagem geométrica *não-manifold* é mais flexível, tendo a capacidade de representar uma variedade maior de objetos e consequentemente ter uma maior quantidade de aplicações do que o sistema de modelagem *manifold*, ao custo de uma estrutura de dados mais robusta e complexa.

Para um modelo tridimensional ser considerado *manifold*, a vizinhança de cada ponto tem que ser topologicamente equivalente a um disco aberto, ou seja, a vizinhança de cada ponto dentro do subespaço deve ser capaz de ser deformado em

um disco bidimensional [38 e 39]. Alternativamente, um sólido *manifold* tem uma fronteira tal que qualquer ponto tem uma vizinhança (Figura 3.6) dividida em exatamente duas partes, uma interior ao sólido e outra exterior. Em um espaço tridimensional, qualquer objeto que não puder satisfazer esta definição de *manifold* é um objeto *não-manifold*. A Figura 3.7 abaixo esclarece este conceito distinguindo os objetos em sólidos *manifolds* e *não-manifolds*.

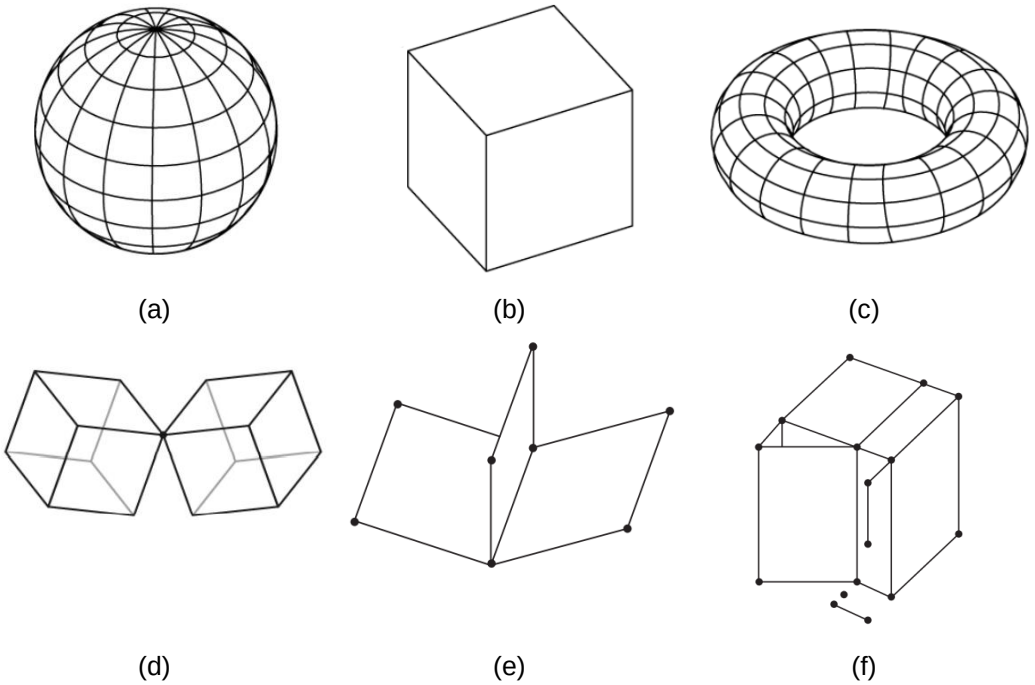


Figura 3.7 - Sólidos *manifolds* e *não-manifolds* [38 e 40]

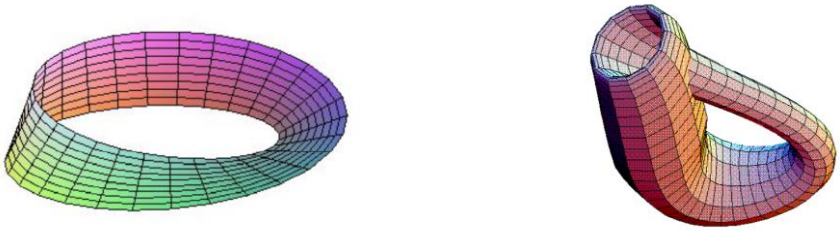
Os sólidos apresentados na Figura 3.7.a, Figura 3.7.b e Figura 3.7.c são classificados como modelos *manifold* pois satisfazem os critérios supracitados e os demais sólidos (Figura 3.7.d, Figura 3.7.e e Figura 3.7.f) não satisfazem esses critérios para todos os pontos e são classificados como modelos *não-manifold.*

Numa representação de sólidos *manifold*, todo ponto numa superfície tem uma vizinhança que é homeomorfa a um disco bidimensional. Isso quer dizer que se analisada localmente numa área suficientemente pequena no entorno de um ponto dado, uma superfície existente num espaço tridimensional pode ser considerada plana. Dessa maneira, deformando essa superfície tridimensional

localmente para um plano, ela não rasga e nem possui pontos coincidentes [25 e 26].

As superfícies de um sólido *manifold* devem ser orientáveis e fechadas para que haja uma nítida distinção entre interior, fronteira e exterior. As superfícies *manifold* de um sólido podem se constituir de diversos pedaços desde que todos esses pedações sejam orientáveis e conectados para formar uma superfície fechada.

De acordo com Hoffman [23], uma superfície *manifold* pode ser dita orientável se for possível distinguir dois lados diferentes. Por exemplo, ao escolher um ponto p de uma superfície e definir arbitrariamente um sentido (horário ou anti-horário) e em seguida movendo-se ao longo de qualquer caminho fechado dessa superfície, mantendo o sentido, existir um caminho tal que seja possível retornar ao ponto p com uma orientação oposta à escolhida, então a superfície é dita não-orientável. Caso contrário, a superfície é orientável. A Figura 3.8 ilustra dois exemplos de superfícies não orientáveis.



(a) (b)

Figura 3.8 - Exemplos de superfícies não orientáveis: a) Faixa de Möbius [31]; b) Garrafa de Klein [24]

A estrutura de dados desenvolvida neste trabalho, concebida para modelar subdivisões planares, utiliza-se do sistema de modelagem *manifold* e, portanto, os itens posteriores estão relacionados com esse tipo de sistema. O sistema *manifold* foi escolhido pois, como mencionado anteriormente, uma subdivisão planar pode ser interpretada como uma superfície *manifold* de um sólido planificado. Além disso, as vantagens proporcionadas pelo sistema *não-manifold* não seriam aproveitadas nas ferramentas desenvolvidas. Além do mais, os modelos *manifold*

armazenam menos memória e a implementação, em geral, é mais simples do que o sistema de modelagem *não-manifold*.

Acrescenta-se como vantagem um formalismo matemático que relaciona as entidades topológicas de um modelo *manifold*. Esse formalismo é apresentado na próxima seção e é a base dos operadores que manipulam uma estrutura de dados *manifold*, conforme visto no próximo capítulo.

3.4
A fórmula de Euler-Poincaré

Seja *S* uma superfície dada como um modelo plano e sejam *V*, *E*, *F* respectivamente o número de vértices, arestas e faces no modelo. Então a soma apresentada na Equação (2) é uma constante que não depende da maneira na qual S foi subdividida a fim de formar o modelo plano [24]. Esta constante é denominada a característica de Euler é denotada por χ.

$$\chi = V - E + F \tag{2}$$

A característica de Euler pode ser expressa, em termos de números de Betti, como:

$$\chi = h_0 - h_1 - h_2 \tag{3}$$

Onde h_0, h_1 e h_2, são denominados os números de Betti do modelo plano. A Equação (3) representa de forma geral a expressão de Euler-Poincaré. Essa fórmula apresenta uma relação quantitativa entre os elementos topológicos e a partir dela é possível deduzir propriedades topológicas da superfície tais como orientação, conectividade e o número de buracos [24].

Na Equação (3), o número de Betti h_0 representa o número de pedaços conexos de uma superfície arbitrária (veja a definição de pedaço conexo na Seção 3.1). O número de Betti h_1 denota a conectividade da superfície (também definido na Seção 3.1). Esse termo representa o maior número possível de curvas fechadas

que podem ser desenhadas sobre a superfície sem separá-la em duas ou mais partes. E o número de Betti h_2 denota a orientação da superfície (explicado na Seção 3.3) [24].

A invariância pode ser generalizada para esses números de Betti. Dessa forma, não importa como uma superfície *S* seja dividida para formar um modelo plano, os números de Betti e todas as características topológicas associadas a eles continuam invariantes [24].

Tendo isso em vista, ao aplicar a equação de Euler-Poincaré no caso mais simples onde o sólido apresenta uma superfície fechada e orientável e sem buracos ou vazios internos, a Equação (3) se traduz em:

$$V - E + F - 2 = 0 \tag{4}$$

Modificações nesta fórmula podem ser feitas para considerar outras situações mais gerais como o caso de a superfície sólida conter orifícios, tais como alças, ou o caso de a superfície conter vazios internos conforme será visto mais adiante.

Uma alça (*handle*) pode ser formada cortando-se dois buracos na superfície do objeto e construindo-se um tubo para ligá-los conforme ilustra Figura 3.9 abaixo. O número de alças é chamado de *genus* da superfície. A ordem (*genus*) de um modelo pode ser definida como o número mínimo de alças que precisam ser adicionadas a uma esfera para que o objeto seja homeomorfo a ela. [23].

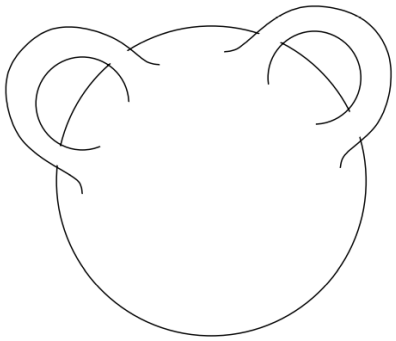


Figura 3.9 - Esfera com duas alças (*genus* = 2) [23]

Considerando a possibilidade de que o sólido tenha alças, mas que permaneça delimitado por uma única superfície conexa e que as faces sejam homeomorfas a um disco fechado a fórmula de Euler-Poincaré, levando em conta *genus* (G), vértices (V), arestas (E) e faces (F), é dada por:

$$V - E + F - 2(1 - G) = 0 \quad (5)$$

Pode-se generalizar ainda mais a fórmula de Euler-Poincaré adicionando a possibilidade de o sólido conter cavidades internas. Essas cavidades são delimitadas por superfícies *manifold* fechadas e separadas, chamadas de cascas (*shells)* ou sólidos. O número de *shells* será denotado por *S* [23].

Além disso, é possível flexibilizar a exigência de que uma face é limitada por apenas uma fronteira totalmente conexa, desde que cada face possa ser mapeada para o plano [23]. Uma fronteira conexa de uma face é denominada *laço* ou *loop* (termo mais utilizado na área e adotado neste trabalho). Um *loop* é formado por uma sequência ordenada e conexa de arestas na fronteira de uma face.

Para ilustrar o conceito de uma face limitada por mais de um *loop*, considere a face da Figura 3.10 que possui quatro *loops* distintos, um externo e três internos. Um dos três *loops* internos é constituído por um único vértice, outro interno é formado por dois vértices conectados por uma aresta. O terceiro *loop* interno é constituído por três vértices conectados por três arestas e o último *loop*, externo, é formado por quatro vértices conectados por quatro arestas.

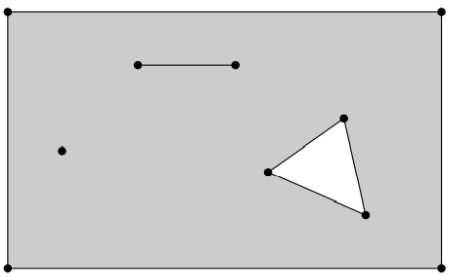


Figura 3.10 - Face com 4 *loops* [23]

Considerando L o número de total de *loops* presentes em todas as faces e S o número de cascas, a fórmula de Euler-Poincaré pode ser generalizada da seguinte forma [23]:

$$V - E + F - (L - F) - 2(S - G) = 0 \quad (6)$$

Embora um sólido *manifold* deva satisfazer a fórmula Euler-Pointcaré, essa não é uma condição suficiente e nem toda superfície que satisfaça a fórmula será uma superfície *manifold*. Por exemplo, um cubo, que é um sólido *manifold*, apresenta seis faces, doze arestas e oito vértices. Da mesma forma, o sólido ilustrado na Figura 3.11 apresenta o mesmo número de vértices, arestas e faces e não é um sólido *manifold*.

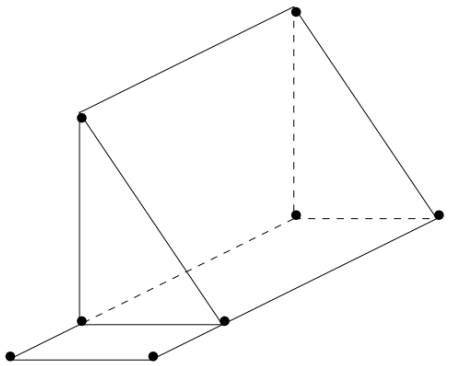


Figura 3.11 - Superfície com 8 vértices, 12 arestas e 6 faces [23]

3.5
Suficiência de uma topologia

Os conceitos apresentados a seguir foram obtidos a partir das referências [23, 25 e 26]. A topologia de um modelo pode ser inconsistente no sentido de existir um sólido múltiplo cujos vértices, arestas, e faces não satisfazem as relações de adjacência. Dessa forma, algumas condições devem ser satisfeitas pelo número de arestas, vértices e faces do modelo para garantir a consistência topológica.

Suficiência topológica é a capacidade de representar de forma completa e não ambígua as adjacências topológicas de um modelo. Na topologia de adjacência existem nove tipos de relações de adjacência possíveis que evolvem os seguintes elementos: vértices (*V*), arestas (*E*) e faces (*F*). Para simplificar essa descrição considera-se apenas o caso em que as faces só apresentem um *loop* (externo). Se uma representação topológica apresentar informações para recriar todas as nove relações de adjacência de forma adequada e sem ambiguidades essa representação pode ser considerada consistente.

A Figura 3.12 ilustra as 9 relações de adjacência possíveis em um modelo. Essas relações foram expressas pelos termos *VV*, *VE*, *VF*, *EV*, *EE*, *EF*, *FV*, *FE* e *FF*. Como mencionado anteriormente, os termos *V*, *E*, *F* denotam respectivamente vértices, arestas e faces. O termo *VE* representa as arestas que incidem em um determinado vértice e o termo *FE* denota as arestas incidentes a uma determinada face. Já o termo *EE* expressa as arestas adjacentes a uma determinada aresta, considerando que duas arestas são adjacentes quando elas compartilham uma face. Os demais termos seguem de forma análoga a lógica supracitada.

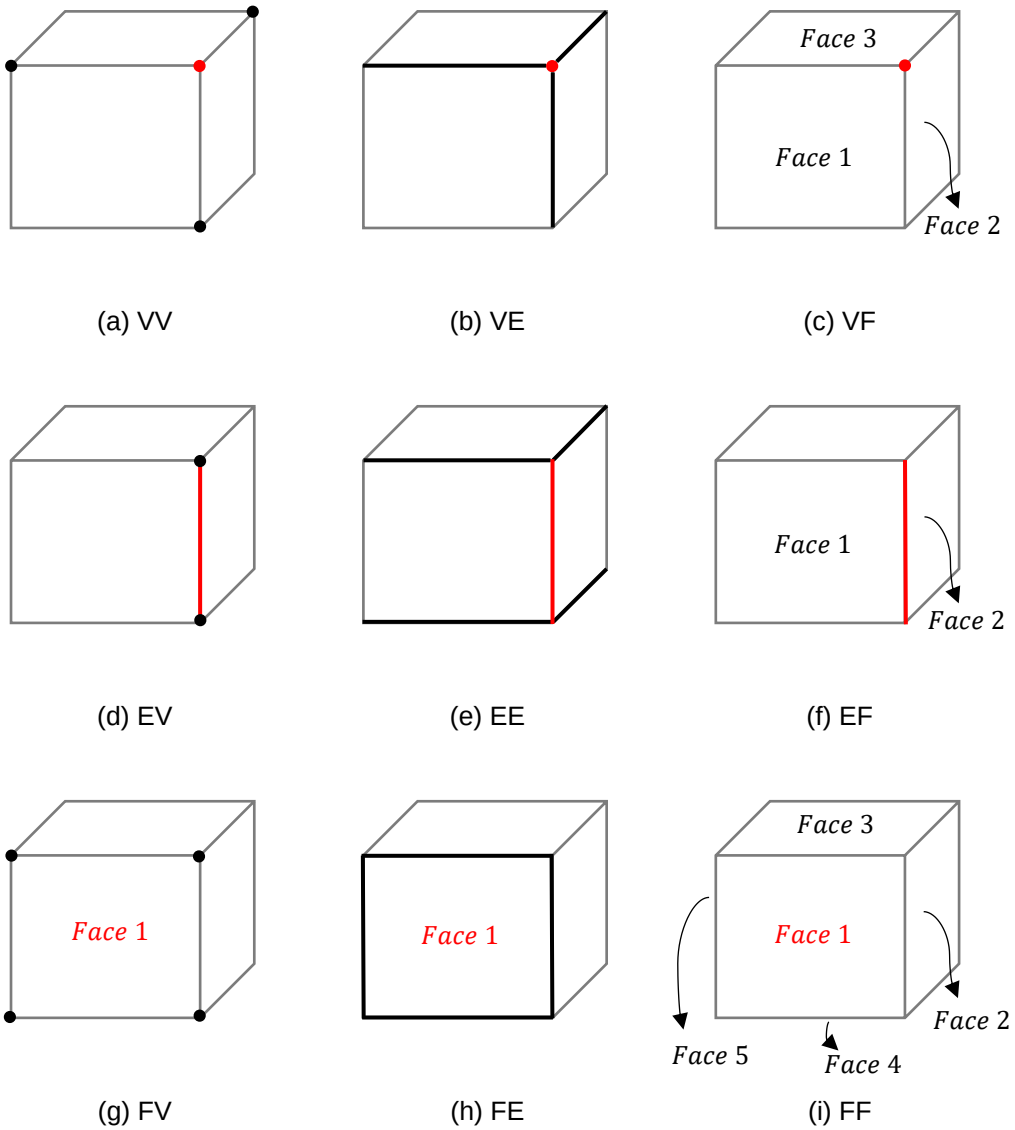


Figura 3.12 - Relações de adjacência

Em geral, não é preciso que um sistema de modelagem armazene todas as nove relações de adjacência, pois isto representaria um gasto adicional e desnecessário de espaço. Dessa forma, é possível armazenar um conjunto mínimo de relações de adjacência de forma que todas as demais relações possam ser derivadas indiretamente e localmente a partir desse conjunto mínimo armazenado.

É interessante observar uma característica das relações de adjacência que têm como elemento central de referência uma aresta. As três relações de adjacência baseadas em aresta têm o número de entidades adjacentes limitado. Isto é, o número

de faces adjacentes a uma aresta é igual a dois (pode ser apenas um, para o caso de uma aresta pendente no interior de uma face), o número de vértices adjacentes a uma aresta é igual a dois (pode ser apenas um, para o caso de uma aresta que é um laço que usa duas vezes o mesmo vértice) e o número de arestas adjacentes a uma aresta (que compartilham uma face com a aresta de referência) é limitado a quatro. Essa característica explica porque a maioria das estruturas de dados topológicas para representação de superfícies *2-manifold* são baseadas em arestas.

No sistema de modelagem *manifold*, restrições devem ser impostas para que os sólidos com superfícies *manifold* sejam fisicamente representados de forma adequada e sem ambiguidades.

A restrição geométrica mais importante na implementação da geometria relativa a uma topologia *manifold* é que as superfícies *manifold* não podem se interceptar a não ser nas suas fronteiras. Em outras palavras, não é permitido que faces se auto interceptem, a não ser em um vértice em comum ou em uma aresta em comum, ou interceptem outras faces. Essa restrição é necessária para manter as superfícies homeomorfas a um disco aberto conforme mencionado na definição de um sólido *manifold*.

De forma geral, sólidos *manifolds* compactos e orientáveis, grafos imersos e conexos e a validade da equação de Euler-Poincaré caracterizam as principais restrições para garantir um domínio topologicamente suficiente para representações *manifold*.

3.6
Domínio topológico

No processo de criação de um sistema de modelagem a definição do domínio topológico é um dos primeiros passos que devem ser tomados. O domínio de uma representação topológica é o conjunto de possibilidades para as quais a representação é válida. Em outras palavras, o domínio representa o conjunto de objetos e processos que podem existir no ambiente a ser desenvolvido. Mais detalhes sobre o domínio topológico podem ser encontrados em [25 e 26].

Uma representação topológica envolve não somente as informações que serão armazenadas e as estruturas de dados utilizadas para organizar estas informações, mas também os tipos de procedimentos que serão disponibilizados no ambiente de modelagem para lidar com estas informações.

Uma representação topológica depende da completa especificação do domínio sobre o qual pretende-se ser utilizado e da total suficiência topológica sobre esse domínio. Logo, o domínio deve ser especificado da forma mais completa possível, definindo uma série de restrições sobre o ambiente desenvolvido. Essas restrições podem ser classificadas em representacionais e procedurais.

As restrições representacionais limitam o que pode ou não ser representável no domínio. Como exemplo, podem existir restrições que limitam o número de cavidades internas de um sólido, o número de *loops* existentes em cada face ou a ordem (*genus*) de uma superfície. Já as restrições procedurais limitam a forma como as condições topológicas vão ser representadas sem restringir o que é ou não representável dentro do domínio. Por exemplo, pode-se limitar a quantidade de alças presentes numa face como sendo zero, sem restringir a possibilidade de representação de uma face com alças, apenas que a alça deve possuir uma fronteira bem definida. As restrições procedurais possibilitam que objetos antes não representáveis possam ser representados desde que estejam de acordo com certas restrições impostas.

4
Estruturas de dados topológicas para representação de sólidos *2-manifold*

A representação e manipulação de modelos geométricos de maneira adequada e eficiente exige a utilização de uma estrutura de dados topológica que possibilite gerenciar todas as informações necessárias para descrever tais modelos. Essa estrutura de dados deve apresentar certas características tais como eficiência e economia no uso de memória [41].

A eficiência de uma estrutura de dados está relacionada com o esforço necessário e a capacidade de interpretar os dados recebidos de forma adequada. O uso da memória de um modelador de sólidos deve ser um dos principais fatores a ser considerado na elaboração do algoritmo da estrutura de dados, pois o uso indiscriminado da memória pode prejudicar a eficiência deste modelador [41].

Uma estrutura de dados topológica é dita completa se ela é capaz de prover as relações de adjacências entre todas as entidades topológicas definidas em tempo ótimo, ou seja, em tempo linearmente proporcional ao número de entidades retornadas [42].

A estrutura de dados pode ser classificada em dois tipos: dinâmica e cinética. Uma estrutura de dados dinâmica apresenta a capacidade de se adaptar a qualquer alteração do modelo. Gold [43] definiu uma estrutura de dados localmente "dinâmica" ou atualizável como uma estrutura que oferece a capacidade de inserção, exclusão, movimentação e navegação local no modelo. Já a estrutura de dados cinética é caracterizada pelas propriedades estáticas da estrutura de dados tais como a falta de suporte para modificações locais, sendo voltada para aplicativos não dinâmicos do usuário final [11].

Nesta seção será discutido apenas as estruturas de dados topológicas baseada na representação de fronteira dos sólidos modelados. Existem várias estruturas de dados topológicas que foram desenvolvidas para criação de modelos de contorno [11 e 44], tais como:

- Winged-Edge (Baumgart [45])
- Half-Edge (Mäntylä [5])
- Quad-Edge (Guibas e Stolf [46])
- Radial Edge (Weiler [26])
- Dual Half-Edge (Boguslawski [27])

A estrutura de dados Radial Edge foi concebida para modelos *não-manifold*. Ela está listada devido a sua importância no grupo de estrutura de dados de representação de fronteira.

A biblioteca HETOOL desenvolvida neste trabalho apresenta uma estrutura de dados baseado na Half-Edge. A Half-Edge é uma estrutura de dados que pode ser dita como um aprimoramento da Winged-Edge. Nas seções a seguir, são fornecidos mais detalhes acerca destas duas estruturas de dados.

4.1
Winged-Edge

A estrutura de dados Winged-Edge, desenvolvida por Baumgart [45] em 1975, é provavelmente a estrutura de dados mais antiga utilizada em modelos de contorno. Ela descreve objetos poliedrais *manifold* através do armazenamento de informações em arestas, vértices e faces. A entidade principal dessa estrutura de dados é a aresta que armazena informações topológicas no formato de listas duplamente encadeadas. Os elementos da estrutura de dados Winged-Edge que são armazenados em cada aresta estão indicados na Figura 4.1.

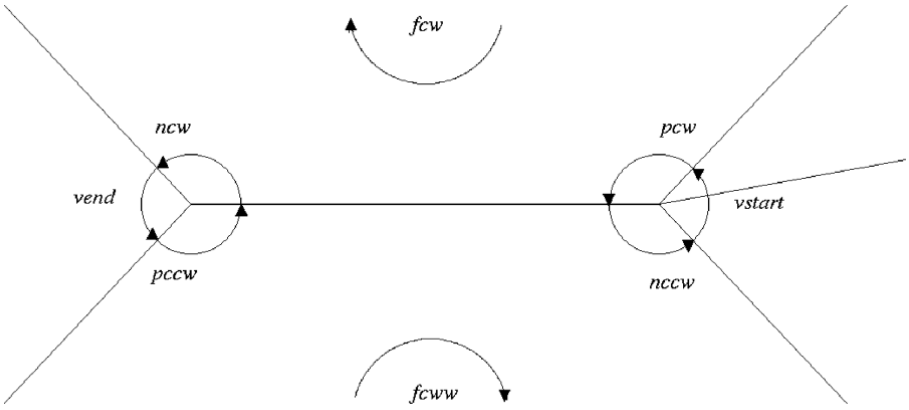


Figura 4.1 - Estrutura de dados Winged-Edge [24]

As informações topológicas armazenadas na estrutura da Winged-Edge consistem, essencialmente, de relações de adjacências de uma determinada aresta com relação a outros elementos topológicos tais como faces, vértices e arestas. O termo Winged-Edge resulta da aparência gráfica das arestas adjacentes, que se assemelham a asas, quando desenhadas em relação à aresta de referência.

Uma vez que cada aresta *e* aparece em exatamente duas faces, exatamente duas outras arestas e' e e'' aparecem adjacentes a *e* em cada uma dessas faces. Além disso, por conta da orientação consistente das faces, *e* está direcionado em orientação positiva (vértice inicial para o vértice final) em uma face e direcionado em orientação oposta na outra face (vértice final para o inicial). A estrutura de dados Winged-Edge baseia-se nessas propriedades estruturais associando identificadores para as duas arestas próximas ao nó da aresta de referência.

Dessa forma, as informações armazenadas na aresta são formadas por identificadores para ambos os vértices da aresta (*vstar* e *vend*), identificadores para ambas as faces que compartilham a aresta (*fcw* e *fcww*) e identificadores para as arestas posteriores (*ncw* e *ncww*) e anteriores (*pcw* e *pcww*) em torno das faces compartilhadas.

Por convenção, estas informações são denominadas por *ncw* para a próxima aresta no sentido horário da face *fcw* e *nccw* para a próxima aresta no sentido anti-horário da face *fcww*. De forma análoga aos termos anteriores, os identificadores

pcw e *pccw* denotam respectivamente, a aresta anterior no sentido horário e a aresta anterior no sentido anti-horário da face [24].

Além das informações armazenadas pelas arestas, cada face armazena um identificador de uma aresta qualquer e um indicador que forneça a orientação desta aresta naquela face (positiva ou negativa). Como qualquer vértice é adjacente a um ciclo ordenado de arestas, cada vértice necessita armazenar somente um identificador de uma aresta qualquer deste ciclo.

Devido a aresta armazenar informações sobre suas duas extremidades, e sobre as duas faces conectadas a ela, as arestas são ditas como não direcionadas e a sua orientação deve ser determinada pela ordenação dos seus vértices. Dessa maneira, é necessário verificar a orientação da aresta no processo de navegação cada vez que se percorrer de uma aresta para a outra o que torna a implementação desta estrutura de dados menos eficiente [27]. A Figura 4.2 apresenta as informações topológicas que podem ser obtidos a partir do cubo da Figura 2.9 utilizando a estrutura de dados Winged-Edge.

Vértice	Coordenadas	Aresta
v_1	$x_1\ y_1\ z_1$	e_1
v_2	$x_2\ y_2\ z_2$	e_2
v_3	$x_3\ y_3\ z_3$	e_3
v_4	$x_4\ y_4\ z_4$	e_4
v_5	$x_5\ y_5\ z_5$	e_5
v_6	$x_6\ y_6\ z_6$	e_6
v_7	$x_7\ y_7\ z_7$	e_7
v_8	$x_8\ y_8\ z_8$	e_8

Face	Primeira aresta
f_1	e_1
f_2	e_9
f_3	e_6
f_4	e_7
f_5	e_{12}
f_6	e_9

Aresta	*vstar*	*vend*	*fcw*	*fccw*	*ncw*	*pcw*	*nccw*	*pccw*
e_1	v_1	v_2	f_1	f_2	e_2	e_4	e_5	e_6
e_2	v_2	v_3	f_1	f_3	e_3	e_1	e_6	e_7
e_3	v_3	v_4	f_1	f_4	e_4	e_2	e_7	e_8
e_4	v_4	v_1	f_1	f_5	e_1	e_3	e_8	e_5
e_5	v_1	v_5	f_2	f_5	e_9	e_1	e_4	e_{12}
e_6	v_2	v_6	f_3	f_2	e_{10}	e_2	e_1	e_9
e_7	v_3	v_7	f_4	f_3	e_{11}	e_3	e_2	e_{10}
e_8	v_4	v_8	f_5	f_4	e_{12}	e_4	e_3	e_{11}
e_9	v_5	v_6	f_2	f_6	e_6	e_5	e_{12}	e_{10}
e_{10}	v_6	v_7	f_3	f_6	e_7	e_6	e_9	e_{11}
e_{11}	v_7	v_8	f_4	f_6	e_8	e_7	e_{10}	e_{12}
e_{12}	v_8	v_5	f_5	f_6	e_5	e_8	e_{11}	e_9

Figura 4.2 - Informações topológicas da estrutura de dados Winged-Edge

4.2
Half-Edge

A estrutura de dados Half-Edge, desenvolvida por Mäntylä [5], funciona essencialmente através de uma entidade abstrata denominada de *half-edge* que

armazena a maioria das informações topológicas presentes na estrutura de dados. A entidade *half-edge* consiste em uma semi-aresta orientada dentro de um laço (*loop*) de uma face. A maioria das consultas topológicas são realizadas utilizando informações armazenadas em tal entidade.

A estrutura de dados de Half-Edge se baseia no fato de que cada aresta é delimitada por exatamente duas faces e isso permite a aresta ser separada em duas semi-arestas que são orientados na direção oposta, tal como indica a Figura 4.3. Isso permite uma orientação consistente das faces, tanto no sentido horário quanto no sentido anti-horário [47].

A estrutura de dados Half-Edge apresenta os seguintes elementos topológicos: sólidos (*shells*), faces (*faces*), laços (*loops*), arestas (*edges*), semi-arestas (*half-edges*) e vértices (*vertices*). A seguir são apresentados detalhes de cada um destes elementos obtidos de [12 e 25]:

- Sólido – O sólido constitui a base de toda a estrutura de dados Half-Edge. A partir do sólido, é possível obter qualquer informação da estrutura de dados topológica como os principais elementos (faces, arestas e vértices) e as relações de adjacência destes elementos. Os principais elementos topológicos estão conectados através de listas duplamente encadeadas contendo identificadores do elemento topológico anterior e posterior ao elemento de referência, conforme pode ser visto na Figura 4.4.
- Face – Este elemento representa a região delimitada por um conjunto de arestas e vértices que formam o contorno da face. As faces podem conter múltiplas fronteiras, de forma que cada face apresenta uma lista de laços sendo cada laço associado a uma fronteira da face. Como todas as faces são planares, um dos *loops* é usualmente chamado de *loop* externo e corresponde a de fronteira externa da face. Já os demais *loops* podem ser chamados de *loops* internos e representam os buracos da face. Além disso, a face armazena os identificadores da face anterior e da face posterior na lista não ordenada duplamente encadeada das faces de um sólido.

- Laço – O *loop*, como já mencionado anteriormente, é o caminho por entre as arestas que pode formar uma fronteira conexa de uma face. Essa entidade possui um identificador para a face que o contém, um identificador para uma das *half-edges* que formam a sua fronteira e identificadores para os *loops* anterior e posterior daquela face.
- Aresta – Para cada aresta do modelo são criadas duas semi-arestas com sentidos contrários, conforme ilustra a Figura 4.3. Esse elemento é composto por identificadores para as duas semi-arestas da aresta e um identificador para elemento geométrico que contém as informações geométricas a respeito dessa aresta. Além disso, essa entidade armazena dois identificadores para a aresta anterior e posterior na lista não ordenada duplamente encadeada de arestas de um sólido.
- Semi-aresta – A *half-edge* consiste em uma semi-aresta de uma aresta que forma um *loop* com as demais semi-arestas que estão orientadas no mesmo sentido dentro de uma determinada fronteira da face. É composta por um identificador para o laço que o contém e um identificador para o vértice da aresta localizado na direção oposta da semi-aresta. Essa entidade possui também identificadores para as *half-edges* anterior e posterior, formando uma lista ordenada duplamente encadeada de *half-edges* de um *loop*.
- Vértice – O vértice apresenta um identificador para o ponto que contem às coordenadas homogêneas do espaço Euclidiano bidimensional ou tridimensional. Há também dois identificadores para os vértices anterior e posterior formando uma lista não ordenada duplamente encadeada dos vértices de um sólido.

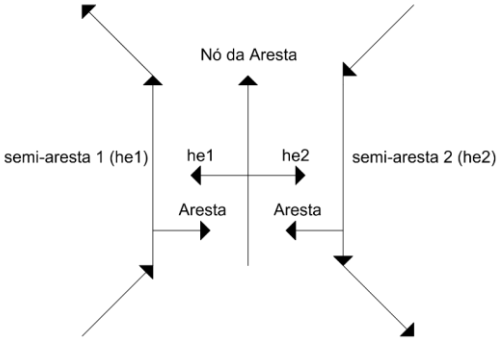


Figura 4.3 - Relação entre a aresta e as semi-arestas da estrutura de dados Half-Edge

A Figura 4.4, ilustra os conceitos mencionados anteriormente e as relações hierárquicas entre os elementos topológicos da estrutura de dados Half-Edge.

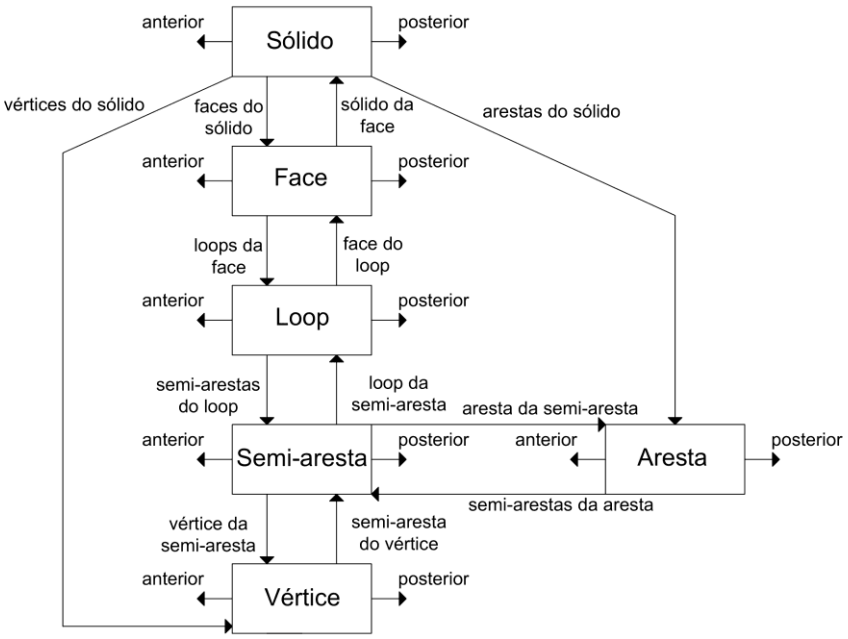


Figura 4.4 - Elementos topológicos da estrutura de dados Half-Edge

A consulta às informações topológicas e geométricas de uma face funciona através da navegação entre as *half-edges* dos *loops* da face (ver Figura 4.5). Cada semi-aresta (*he*) apresenta um indicador para a semi-aresta posterior (*he_next*) e

anterior (*he_prev*) garantindo, dessa maneira, completo acesso as informações topológicas de todos os elementos pertences a face. Conforme ilustra a Figura 4.5, para obter as informações referentes a uma face adjacente, começa-se a consulta pela semi-aresta *mate*, que é paralela a semi-aresta atual, porém em sentido contrário.

Na Figura 4.5, todas as semi-arestas não tracejadas pertencem ao laço da face localizada à esquerda do vértice em destaque. Apenas as semi-arestas não tracejadas podem ser acessadas diretamente pela semi-aresta *he* utilizando a lista duplamente encadeada. As outras semi-arestas podem ser acessadas utilizando as semi-arestas *mate* que são estão representadas pelas setas tracejadas.

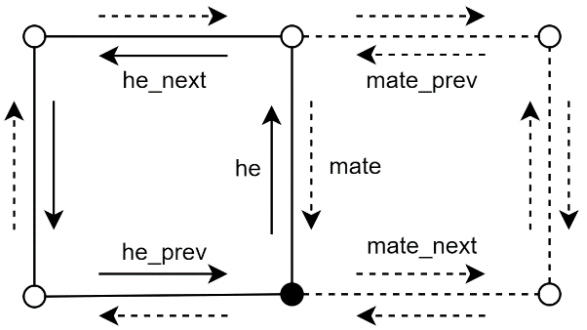


Figura 4.5 - Relação entre as semi-aretas posterior e anterior.

Na estrutura de dados Half-Edge, a orientação do *loop* externo de cada face funciona da seguinte forma: o identificador *next* da semi-areta *he* ou *mate* aponta para a próxima *half-edge* ao redor da face no sentido anti-horário *he_next* ou *mate_next*. Para cada *loop* da face é suficiente armazenar apenas um identificador de uma semi-aresta qualquer do *loop* externo, as demais semi-arestas pertentes ao laço desta face podem ser acessadas através da lista duplamente encadeada quando necessário.

Com a estrutura de dados Half-Edge é possível representar buracos nas faces do sólido a partir dos laços internos de cada face. Como já mencionado, cada face apresenta um identificador para o *loop* externo (a fronteira externa da face) e uma lista de identificadores contendo os *loops* internos que representam as fronteiras internas (buracos) da face [27].

Em geral, todos os laços da face são armazenados em uma lista duplamente encadeada sendo o *loop* externo o primeiro *loop* desta lista. Conforme ilustra a Figura 4.6.a, os laços internos têm a orientação oposta (sentido horário) ao *loop* externo (sentido anti-horário).

Existe também outro método, menos convencional, que pode ser utilizado para a representação destes furos na face. Em vez de manter vários *loops* descrevendo uma face, os laços podem ser conectados em um único *loop* externo contendo semi-aretas como pontes para os *loops* internos conforme indicado na Figura 4.6.b [27]. Este método não é adotado neste trabalho.

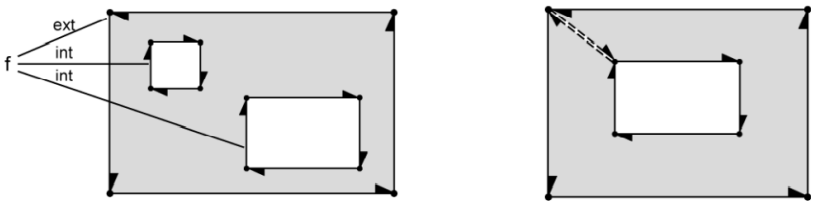


Figura 4.6 - Face contendo buracos representada por: (a) um *loop* externo e vários *loops* internos; (b) Um *loop* externo que é ligado a fronteira do furo por uma ponte de *half-edges* [27]

A Figura 4.7 e a Figura 4.8 exemplificam o funcionamento da estrutura de dados Half-Edge demonstrando as informações que podem ser obtidas para a composição de tabelas que expressam as relações entre as entidades topológicas dessa estrutura de dados.

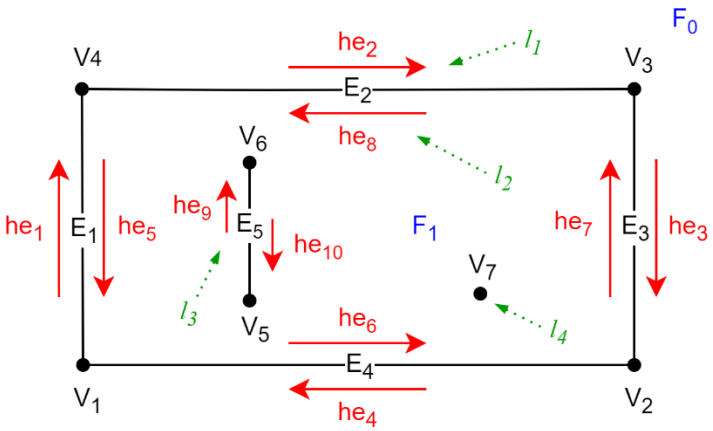


Figura 4.7 - Exemplo da estrutura de dados Half-Edge

Vértices	Coordenadas	Semi-aresta
V_1	$x_1\ y_1$	he_1
V_2	$x_2\ y_2$	he_4
V_3	$x_3\ y_3$	he_3
V_4	$x_4\ y_4$	he_2
V_5	$x_5\ y_5$	he_9
V_6	$x_6\ y_6$	he_{10}
V_7	$x_7\ y_7$	$-$

Arestas	Semi-arestas
E_1	he_1 e he_5
E_2	he_2 e he_8
E_3	he_3 e he_7
E_4	he_4 e he_6
E_5	he_9 e he_{10}

Loops	Semi-aresta	Face
l_1	he_1	F_0
l_2	he_5	F_1
l_3	he_9	F_1
l_4	$-$	F_1

Faces	Loop_externo	Loop_interno
F_0	$-$	l_1
F_1	l_2	$l_3\ e\ l_4$

Semi-aresta	Aresta	Vértice	Loop	he_next	he_prev
he_1	E_1	V_1	l_1	he_2	he_4
he_2	E_2	V_4	l_1	he_3	he_1
he_3	E_3	V_3	l_1	he_4	he_2
he_4	E_4	V_2	l_1	he_1	he_3
he_5	E_1	V_4	l_2	he_6	he_8
he_6	E_4	V_1	l_2	he_7	he_5
he_7	E_3	V_2	l_2	he_8	he_6
he_8	E_2	V_3	l_2	he_5	he_7
he_9	E_5	V_5	l_3	he_{10}	he_{10}
he_{10}	E_5	V_6	l_3	he_9	he_9

Figura 4.8 - Informações da estrutura de dados Half-Edge

A face F_0 é a face infinita que não apresenta uma fronteira externa definida, sendo apenas constituída por *loops* internos. As demais faces podem apresentar tanto *loops* externos como *loops* internos que dão origem aos buracos na face.

Conceitualmente o *loop* l_4 da Figura 4.7 não apresenta *half-edges*. Contudo, pode-se adotar, para facilitar a implantação computacional, a existência de uma "*half-edge* virtual" no *loop* l_4. Na biblioteca HETOOL desenvolvida neste trabalho foi adotado a estratégia da existência de uma *half-edge* virtual no caso de vértices isolados.

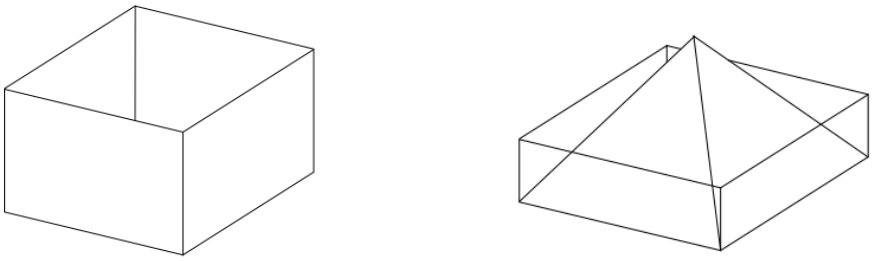
4.3
Validade dos modelos de contorno

A validade de modelos de contorno é em geral muito difícil de se estabelecer. Os critérios de validade dividem-se em restrições geométricas e topológicas. Embora seja possível gerenciar a validade topológica sem grandes custos adicionais, é difícil impor critérios geométricos sem penalizar a velocidade de interação do usuário.

De acordo com Mäntylä [5], a validade de um modelo de contorno, considerando que o objeto sólido representado seja limitado e suas superfícies sejam fechadas e orientáveis, é determinado pelas seguintes condições:

- O conjunto de faces de um modelo de contorno deve formar uma "pele" completa do sólido cobrindo-o totalmente;
- Faces do modelo não devem se interceptar a não ser em arestas ou vértices comuns;
- As fronteiras das faces devem ser polígonos simples que não se auto interceptam.

A primeira condição não permite a representação de objetos "abertos", como a caixa ilustrada na Figura 4.9.a, sejam considerados validos. Já as duas últimas condições excluem objetos que se auto interceptam, como o caso ilustrado na Figura 4.9.b. Além disso, a segunda condição também impossibilita a representação de objetos que não sejam orientáveis porque eles sempre se auto interceptam no espaço tridimensional [5 e 24].



(a) (b)

Figura 4.9 - Modelos de contorno não válidos [24]

A primeira condição está relacionada com a integridade topológica de um modelo de contorno. Essa integridade topológica pode ser imposta apenas por meios estruturais. Em particular, a primeira condição pode ser satisfeita impondo-se com que cada aresta seja adjacente a no máximo duas faces como ocorre nas estruturas de dados Half-Edge. Dessa forma, elimina-se a possibilidade de uma aresta ser o contorno de uma parte da superfície que esteja faltando [5 e 24].

Além das três condições citadas acima, se as informações geométricas contidas nas entidades topológicas forem inapropriadas, isso pode gerar sólidos inválidos. Logo, a integridade geométrica de um modelo de contorno, definida pela segunda e terceira condição, não pode ser imposta apenas por meios estruturais.

Para garantir a integridade geométrica, testes e restrições computacionais podem ser implementados. Esses testes realizam verificações geométricas das informações atribuídas as entidades topológicas e as restrições limitam a liberdade do usuário dando a ele apenas mecanismos e funcionalidades que produzam sólidos válidos [5 e 24].

4.4
Operadores de Euler

O processo de construção de um modelo de contorno, em geral, é feito pela adição incremental de segmentos individuais, começando pela criação de um sólido (*shell*). Assim, os dados armazenados no computador precisam ser constantemente

modificados e atualizados conforme o modelo é construído. No nível mais baixo deste processo, existem operadores que alteram diretamente a estrutura de dados e sua implementação depende diretamente das características e particularidades dessa estrutura de dados [27].

Esses operadores são, em geral, utilizados em programas de modelagem juntamente com uma estruturada de dados. Essa estrutura de dados obtém as interações do usuário e determina os operadores que serão executados como resposta à cada interação de modelagem realizada no programa. Em CAD, onde a modelagem sólida B-Rep é amplamente utilizada, tais operadores são denominados de operadores de Euler que desempenham a função de modificar a estrutura de dados no nível das entidades topológicas [27].

Conceitualmente, de acordo com Hoffman [23], os operadores de Euler podem ser definidos como criadores e modificadores da estrutura de dados topológica de sólidos. Em particular, eles podem criar superfícies fechadas e modificar essas superfícies adicionando ou excluindo faces, arestas e vértices. Além disso, esses operadores também podem modificar o *genus* da superfície adicionando ou deletando *handles* (alças).

Os operadores de Euler são usados, em geral, como uma linguagem de baixo nível em sistemas de modelagem. Eles fornecem um nível de abstração bastante desejável para os algoritmos que são implementados em um nível mais alto. Assim, a representação mais básica (vértice, aresta ou face) pode ser modificada com pouco impacto sobre a implementação do sistema de modelagem [23].

Existem vários tipos de operadores de Euler que são capazes de construir arestas, dividir *loops*, destruir vértices etc. Como já foi mencionado, o número de entidades topológicas (vértices, arestas, *loops*, faces, *genus*, sólidos) em uma estrutura de dados deve satisfazer a fórmula de Euler-Poincaré (Equação (6)) para o modelo representado ser válido. Portanto, a adição de quaisquer entidades por um único operador de Euler deve preservar esta relação [27].

Consequentemente, os operadores de Euler também devem satisfazer a equação de Euler-Poincaré (apresentada na Seção 3.4). Essa restrição impossibilita que a estrutura de dados possua um número negativo de entidades ou que existam interseção entre sólidos sem que nenhuma outra entidade intermediária esteja

presente (aresta ou vértice). Além disso, a equação de Euler-Poincaré garante que um objeto válido apresente pelo menos um vértice e pelo menos uma face [27].

Além de garantir a validade da equação de Euler-Poincaré, os operadores de Euler são implementados de tal maneira que preservem a consistência topológica da estrutura de dados. Dessa maneira, uma vez testada e verificada sua correta implementação, os operadores de Euler constituem uma camada de abstração que encapsula de forma correta todas as costuras topológicas para manipulação de um modelo de superfície *2-manifold*.

Os operadores de Euler em um modelador geométrico são considerados operadores de baixo nível. Para uma construção correta de um modelo, são necessárias a definição e a implementação de funções de nível intermediário que convertam de maneira consistente informações geométricas de pontos e curvas para os argumentos de entrada dos operadores de Euler. Ainda são necessárias funções de alto nível que preparam os dados de entrada do usuário para as funções de nível intermediário. Por exemplo, uma curva inserida deve ser particionada em trechos que se encaixam geometricamente com as faces existentes. Esse particionamento é realizado por uma função de alto nível. Neste trabalho, as funções de nível intermediário e de alto nível da biblioteca de modelagem proposta são descritas no próximo capítulo.

Por convenção histórica, as operações de Euler são tradicionalmente nomeadas por um termo do formulário *mxky*, onde *m* representa a expressão *make* que está associada a construção de certa entidade topológica e *k* representa a expressão *kill* que está associada a destruição de determinada entidade topológica [23]. Os termos mais utilizados estão listados abaixo:

- M – *make* – construir
- K – *kill* – destruir
- S – *split* – quebrar
- J – *join* – encaixar
- V – *vertex* – vértice
- E – *edge* – aresta
- F – *face* – face
- S – *solid* ou *shell* – sólido ou casca

- H – *hole* – buraco
- R – *ring* – anel

A partir do formulário listado é possível inferir que o operador MVFS denota "*make a vertex, a face and a solid*" que significa construir um vértice, uma face e um sólido. Da mesma forma o operador MEV denota "*make an edge and a vertex*" que corresponde a construir uma aresta e um vértice. E o operador MEF denota "*make an edge and a face*" que significa construir uma aresta e uma face. De maneira análoga, outros operadores podem ser derivados do formulário apresentado acima e seguirão a mesma lógica supracitada.

Nas seções seguintes, serão detalhados os operadores mais utilizados na construção de uma modelo geométrico. Estes conceitos podem ser obtidos a partir de [5, 24 e 28].

4.4.1
Operadores MVFS e KVFS

O operador MVFS desempenha o papel de criar um sólido que armazena todas as outras informações da estrutura de dados. Inicialmente, essa estrutura é constituída de uma face e um único vértice. Essa face não apresenta *loop* externo (fronteira externa da face é infinita), possuindo um único *loop* interno que está vazio (não apresenta aresta), ou seja, esse *loop* é formado por apenas um vértice isolado. O efeito do operador MVFS é ilustrado na Figura 4.10.

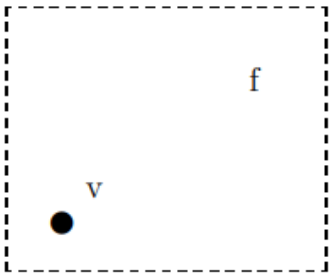


Figura 4.10 - Operador MVFS [24]

Todos os operadores de Euler possuem operadores inversos que desfazem o efeito do operador associado. KVFS é o operador inverso de MVFS. Ele destrói os elementos topológicos da estrutura de dados que foram criados pelo operador MVFS.

4.4.2
Operadores MEV E KEV

O operador MEV apresenta a capacidade de subdividir o ciclo de arestas de um vértice quebrando o vértice em dois e unindo-os a partir de uma nova aresta conforme ilustra a Figura 4.11.a. Com isso, como o próprio nome do operador sugere, um vértice e uma aresta são adicionados à estrutura de dados.

A aplicabilidade de MEV pode ser estendida para vértices isolados subdividindo o vértice em dois novos ligados por uma aresta (Figura 4.11.b). O operador MEV também pode desempenhar a função de ligar um novo vértice com um antigo por meio de uma nova aresta, como ilustrado na Figura 4.11.c.

O operador inverso ao MEV é o KEV que apresenta a capacidade de desfazer quaisquer um dos três casos da Figura 4.11. Dessa maneira, dada uma aresta que conecta dois vértices distintos, o operador KEV apresenta a capacidade de remover a aresta e unificar os vértices em um e combinar seus ciclos de arestas.

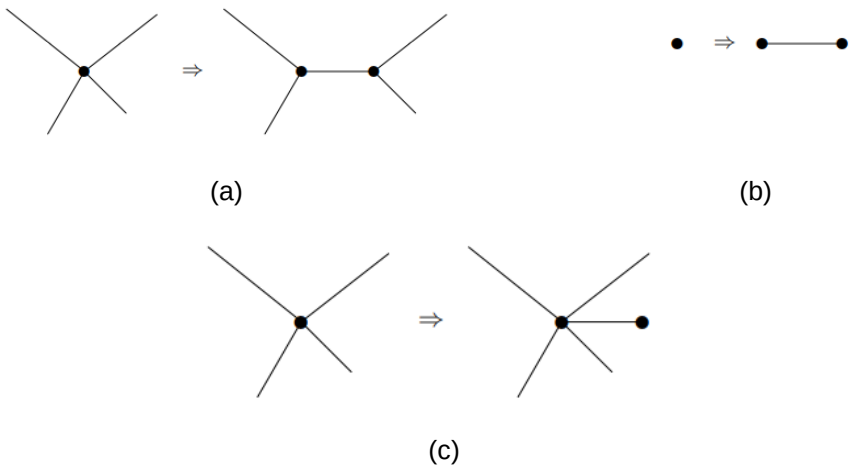


Figura 4.11 - Operador MEV [24]

4.4.3
Operadores MEF E KEF

O operador MEF desempenha o papel de subdividir um *loop* unindo dois vértices com uma nova aresta conforme mostra a Figura 4.12.a. Consequentemente, uma nova aresta e uma nova face são adicionadas a estrutura de dados.

A utilização do operador MEF pode ser estendida para *loops* vazios da mesma maneira como o MEV. Assim, o operador MEF pode ser utilizado para criar uma aresta que forma um laço (*self-loop*) em um vértice isolado separando duas faces, conforme ilustra Figura 4.12.b. Generalizando ainda mais este processo, é sempre possível conectar um vértice a si mesmo em termos de MEF (Figura 4.12.c).

O operador inverso KEF pode desfazer o efeito de MEF em cada um dos casos supracitados. Desse modo, dada uma aresta adjacente a duas faces distintas, KEF é capaz de remover a aresta e unir as duas faces. O *loop* da face resultante é a combinação das fronteiras das faces originais.

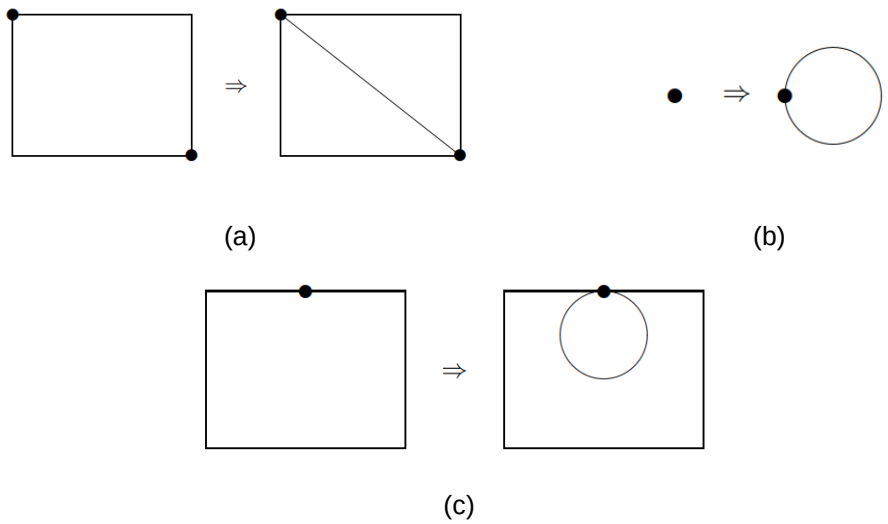


Figura 4.12 - Operador MEF [24]

4.4.4
Operadores KEMR e MEKR

O operador KEMR desempenha a função de subdividir um *loop* em outros dois removendo uma aresta que antes unia estes *loops*. O efeito disto pode ser visualizado na Figura 4.13.a. Em outras palavras, o operador KEMR destrói uma aresta separando o caminho entre as arestas em dois *loops* distintos.

O efeito dessa operação na estrutura de dados consiste na remoção de uma aresta e na adição de um *loop* ou *ring*. Os casos especiais em que um ou ambos os *loops* resultantes são vazios (não apresentam arestas) também estão incluídos neste processo (Figura 4.13.b e Figura 4.13.c). O operador inverso ao KEMR é o operador MEKR que apresenta a capacidade de unir dois *loops* de uma face por meio de uma nova aresta.

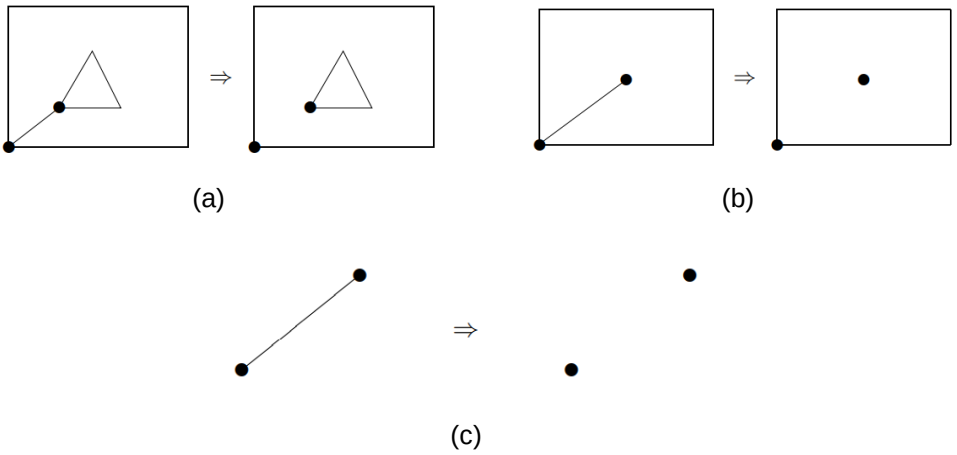


Figura 4.13 - Operador KEMR [24]

4.4.5
Operadores MVSE e KVJE

O operador MVSE apresenta a função de dividir uma aresta em duas através da criação de um novo vértice situado entre essas duas arestas, conforme ilustra a Figura 4.14. Consequentemente, uma nova aresta e um novo vértice são adicionados à estrutura de dados. O operador inverso ao MVSE é o operador KVJE

que apresenta a capacidade de eliminar um vértice unindo duas arestas em uma, ou seja, este operador desfaz tudo o que foi feito pelo operador MVSE.

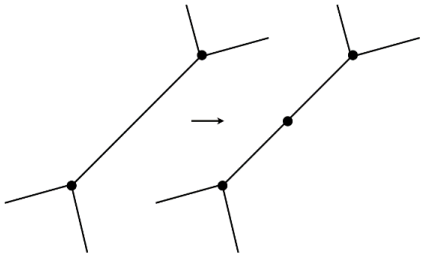


Figura 4.14 - Operador MVSE

4.4.6
Propriedade dos operadores de Euler

A estrutura de dados criada pelo operador MVFS possui um sólido, um vértice e uma face, ou seja, considerando a equação de Euler-Poincaré (Equação (6)) tem-se que $V = F = L = S = 1$ e $G = 0$. Substituindo-se estes valores na mesma equação nota-se que a operação é válida.

Considerando o uso do operador MEV após o uso do MFVS seriam adicionados um vértice e uma aresta a estrutura de dados. Logo os termos da Equação (6) seriam $S = F = E = L = 1$, $V = 2$ e $G = 0$ que ao serem substituídos na expressão de Euler-Poincaré garantem a validade da operação MEV.

Assim como os exemplos de MFVS E MEV mencionados anteriormente, todos os demais operadores de Euler possuem a propriedade de adicionar um certo número de entidades topológicas a estrutura de dados de forma que a equação de Euler-Poincaré continue sendo válida para o modelo representado.

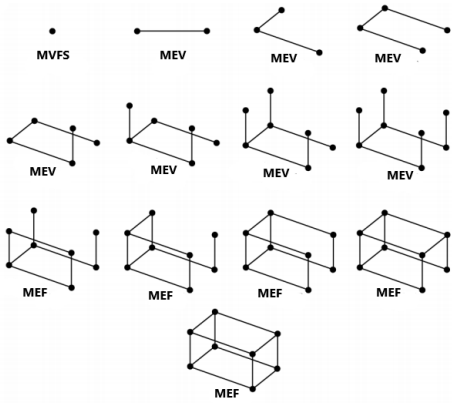
Dessa forma, é possível afirmar que os operadores de Euler são topologicamente consistentes, ou seja, eles não podem adicionar entidades que torne a estruturas de dados topologicamente inválida.

Contudo, todas as entidades sobre os quais os operadores de Euler operam devem existir e ser do tipo apropriado. Por exemplo, o operador KEF pode ser

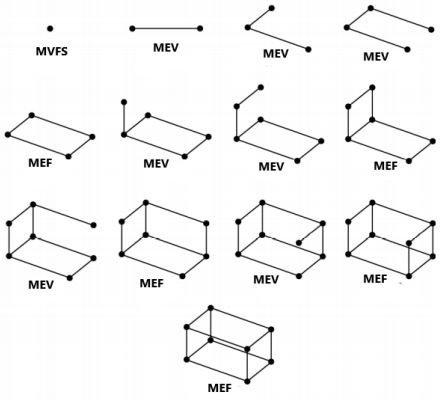
somente aplicado a uma aresta que aparece em duas faces distintas. Assim, devem ser impostas restrições no nível mais alto da estrutura de dados que garanta a validade do uso de cada um dos operadores de Euler.

4.4.7
Exemplos de utilização dos operadores

Existem diferentes sequências possíveis para a utilização dos operadores. A Figura 4.15 exemplifica duas formas distintas de como utilizar os operadores MVFS, MEV e MEF para a construção de um cubo. A Figura 4.15.a mostra um caso em que sete operadores MEV são usados primeiro antes de utilizar qualquer operador MEF. E a Figura 4.15.b mostra um método mais natural ou intuitivo da utilização destes operadores.



(a)



(b)

Figura 4.15 - Construção de cubo usando operadores de Euler: a) os operadores MEV são utilizados antes dos MEF; b) cubo é construído face por face. Adaptado de [27]

5
Biblioteca HETOOL

Neste capítulo são apresentados as principais características e módulos presentes na biblioteca HETOOL. O pacote desenvolvido representa subdivisões planares e utiliza uma estrutura de dados B-Rep (Half-Edge) para a representação da fronteira dos sólidos modelados. Neste trabalho, o termo "sólido" está sendo utilizado para representar uma subdivisão planar. A razão é que de fato a estrutura de dados Half-Edge foi concebida para modelar a fronteira de um sólido.

A biblioteca foi desenvolvida na linguagem de programação Python, que é uma linguagem que vem ganhando bastante espaço no meio acadêmico por ser mais simples e prática. Python é uma linguagem de alto nível e de tipagem dinâmica que exige um menor número de linhas de código se comparado ao mesmo programa escrito em outras linguagens. Essa linguagem de programação contribuiu para criação do sistema de gerenciamento de atributos que funciona a partir de um arquivo JSON. Esse arquivo permite uma comunicação direta com Python e outras linguagens mais modernas, como JavaScript.

A biblioteca HETOOL foi implementada seguindo o padrão da programação orientada a objetos. Esse padrão se baseia no uso de classes, objetos, atributos e métodos. A seguir apresentam-se as definições básicas de cada um destes elementos (mais informações acerca desse assunto podem ser obtidas em [48]).

- Classes são construtores de objetos que apresentam características particulares tais como os métodos que são capazes de realizar e os atributos que estes objetos possuem.
- Objeto é uma instancia gerada a partir de uma classe e é identificado a partir dos métodos e dos atributos que possui.
- Métodos são as funções que objeto pode realizar.
- Atributos representam as variáveis que o objeto possui. Serve para descrever o objeto e armazenar valores específicos.

A estrutura de dados desenvolvida neste trabalho é uma estrutura dinâmica que apresenta processamento automático das interseções entre as entidades geométricas presentes no modelo. Para o uso desta estrutura de dados não é preciso ter conhecimento sobre os conceitos topológicos envolvidos na implementação dela, pois todos os dados de entrada dos métodos de alto nível relacionados com a modelagem lidam apenas com coordenadas e entidades geométricas.

Essa estrutura de dados foi modularizada para gerenciar de forma rápida e eficiente todas as das informações topológicas e geométricas do sólido modelado. Alguns módulos dessa estrutura foram generalizados para facilitar novos incrementos ao código implementado.

Entre os módulos que foram generalizados, destaca-se o módulo responsável pelo gerenciamento de atributos que viabiliza facilmente a criação de atributos com diferentes tipos de propriedades. Outro módulo generalizado é o módulo responsável pelos operadores auxiliares que possibilitam ao usuário desfazer e refazer um conjunto variado de comandos. A generalização destes operadores auxiliares permite de maneira simples a adição de novos comandos que podem ser desfeitos e refeitos.

A biblioteca HETOOL foi desenvolvida seguindo o padrão Model – View – Controller (MVC). O padrão MVC subdivide a estruturas de dados em três subsistemas, conforme pode ser visto na Figura 5.2.

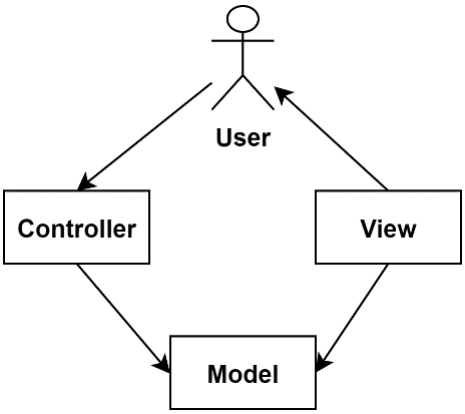


Figura 5.1 - Padrão MVC

Esse padrão apresenta um controlador (*Controller*) que gerencia as interações do usuário (*User*), um modelo (*Model*) responsável por armazenar as informações da estrutura de dados e o visualizador (*View*) que coleta as informações do modelo. Em contraste com outros sistemas onde essas três funcionalidades não são agrupadas, o padrão MVC ajuda a produzir módulos altamente coesos e mais flexíveis [48].

Conforme pode ser visto na Figura 5.1, tanto o controlador quanto o visualizador têm acesso ao modelo. Contudo, o único subsistema capaz de realizar alterações no modelo é o controlador. O visualizador cumpre apenas um papel de leitor do modelo sem realizar alterações neste.

O controlador é responsável por realizar o processamento de toda interação do usuário com os dados já inseridos no modelo, modificando a estrutura de dados topológica e alterando as informações armazenadas no modelo quando necessário.

O visualizador apenas acessa o modelo e obtém os dados em um formato específico para ser transmitido para a classe responsável por renderizar na tela o sólido modelado. O modelo apresenta uma função passiva de armazenar os dados do sólido desenvolvido, ou seja, ela apenas recebe as informações processadas pelo controlador e as guarda.

Os subsistemas criados a partir do padrão MVC, dentro da arquitetura da biblioteca HETOOL, receberam um prefixo *He*. A Figura 5.2 exibe um diagrama de robustez simulando a utilização da biblioteca HETOOL por um aplicativo. Esse diagrama apresenta todos os módulos principais e secundários presentes no pacote desenvolvido.

O *Hecontroller* é a classe central do HETOOL que tem a função de processar todos os dados de entrada e, portanto, essa classe tem acesso a quase todos os módulos da biblioteca conforme pode ser visto no diagrama. O *CompGeom* é um módulo secundário da biblioteca HETOOL que é responsável por computar e analisar as interseções e outras questões geométricas entre os elementos presentes no modelo. Nas seções a seguir serão discutidos mais detalhes acerca dos módulos principais e secundários presentes na biblioteca desenvolvida.

O processo de modelagem controlado pela classe *Hecontroller* trabalha com funções ou operadores em três níveis, conforme detalhado neste capítulo. No nível mais alto, a construção de uma subdivisão planar se dá através da inserção de curvas retas ou poligonais, ou pela inserção de pontos. Essas curvas ou pontos podem interceptar outras curvas existentes no modelo (ou mesmo se auto interceptar) e a decomposição de curvas entrantes em partes que se encaixam na subdivisão planar existente é realizada por funções de alto nível com auxílio do módulo *CompGeom*. Existem também funções de alto nível para eliminação de entidades geométricas da subdivisão planar corrente. A costura topológica dos trechos das curvas entrantes que se encaixam na subdivisão planar é feita por operadores de baixo nível, que são os operadores de Euler e operadores auxiliares. As funções de nível intermediário são responsáveis pela preparação dos dados topológicos e geométricos que constituem os argumentos de entrada dos operadores de Euler.

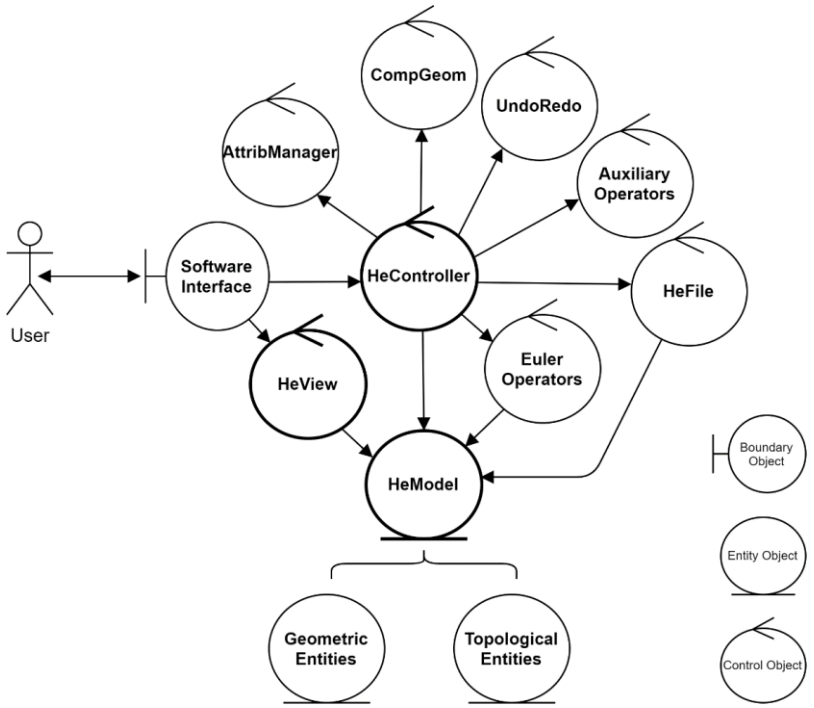


Figura 5.2 - Diagrama de robustez da biblioteca HETOOL

5.1
Classe *HeModel*

O *HeModel* é a classe responsável por armazenar todas as informações presentes na estrutura de dados. Essa classe permite o rápido acesso as entidades geométricas e topológicas por meio de listas. A classe *HeModel* apresenta um atributo que guarda o endereço de memória de um sólido (*shell*) e três atributos (*points*, *segments* e *patches*) que permitem obter todos os pontos, segmentos e regiões do sólido modelado pelo usuário. Uma porção conexa da fronteira de um sólido é denominada casca (*shell*). Por isso, a classe de um sólido (subdivisão planar) é a classe *Shell*, pois essa representa uma casca (*shell*) planificada.

A Figura 5.3 apresenta o diagrama da classe *HeModel* demonstrando a relação desta classe com as entidades geométricas e topológicas presentes na estrutura de dados. O acesso a qualquer entidade topológica pode ser obtida facilmente através do sólido (classe *Shell* explicada no item 5.2) que contêm as listas dos vértices, arestas e faces deste modelo. A Figura 5.4 apresenta os principais atributos e métodos presentes na classe *HeModel.* A seguir serão detalhados alguns métodos desta classe.

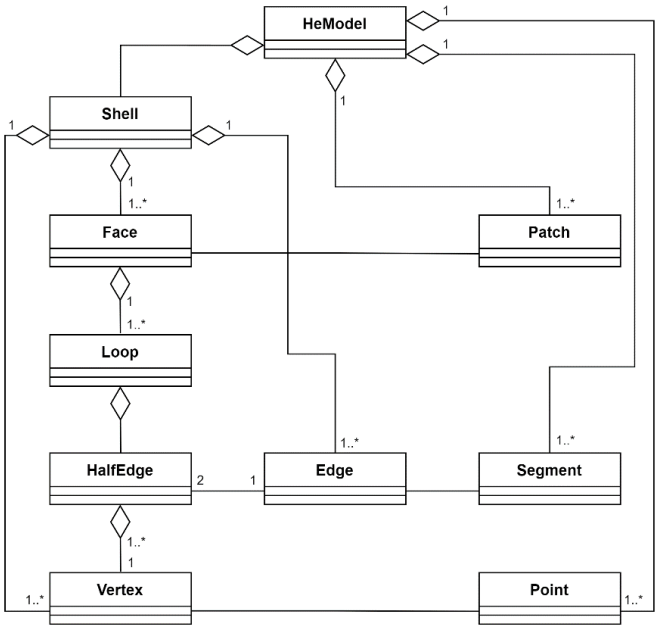


Figura 5.3 - Diagrama da classe *HeModel*

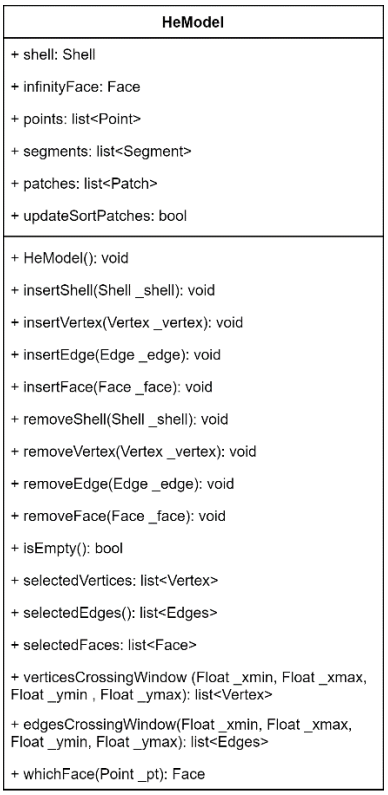


Figura 5.4 - Classe *HeModel*

Os métodos que adicionam e removem as entidades topológicas do modelo ajustam em cada entidade geométrica o atributo que armazena o endereço de memória de sua respectiva entidade topológica. Além disso, cada um destes métodos chama outra função de mesmo nome na classe *Shell* para adicionar ou remover os dados topológicas nas listas presentes no sólido.

O método *isEmpty* é utilizado para verificar se o modelo está vazio. Caso o atributo *shell* da classe *HeModel* aponte para um espaço nulo da memória então o modelo está vazio e nesta circunstância o método retorna o indicador *True.* Caso contrário, o método retorna o indicador *False.*

Os métodos *selectedVertices*, *selectedEdges* e *selectedFaces* percorrem, respectivamente, as listas de vértices, arestas e faces do sólido buscando as entidades que foram selecionadas pelo usuário e armazenando essas entidades em listas. Por fim, cada um destes métodos retorna uma lista contendo as entidades selecionadas.

Os métodos *verticesCrossingWindow* e *edgesCrossingWindow* recebem uma região retangular (limitada por *_xmin*, *_xmax*, *_ymin* e *_ymax*) e retornam, respectivamente, uma lista de vértices e outra de arestas que estão dentro da região retangular ou que cruzam o limite desta região. Esse método é utilizado para obter as entidades do modelo que podem fazer interseção com uma nova aresta inserida pelo usuário.

Por fim, o método *whichFace* é utilizado antes do operador de Euler MVR (explicado na Seção 5.4). Esse método recebe um ponto e retorna a face do modelo a qual este ponto pertence.

5.2
Entidades topológicas e geométricas

As entidades topológicas (*Topological Entities*) são o conjunto de classes que compõem o "esqueleto" da estrutura de dados Half-Edge. Essas entidades topológicas são formadas pelas seguintes classes: *Vertex, Half-Edge, Edge, Loop, Face* e *Shell*. A hierarquia entre essas entidades segue o mesmo padrão explicado anteriormente na Seção 4.2.

Nesta seção serão detalhados os principais atributos e métodos de cada entidade topológica bem como a relação entre as classes desses elementos topológicos. Todos os diagramas apresentados nesta seção são apenas resumos contendo as principais informações de cada classe.

As classes *Vertex, Edge* e *Face* representam as entidades topológicas não abstratas que tem relação direta com as entidades geométricas *Point, Segment* e *Patch* respectivamente (ver Figura 5.3). Assim, cada entidade topológica não abstrata apresenta um atributo que armazena o endereço de memória da sua respectiva entidade geométrica e vice-versa. Já o *Loop* e a *HalfEdge* são entidades abstratas que não apresentam uma classe geométrica correspondente.

O sólido é a entidade geral da estrutura de dados que permite o acesso a todas as outras entidades topológicas existentes. Cada sólido criado é um objeto da classe *Shell* que apresenta um atributo que armazena o endereço de memória de

uma face (utiliza-se a primeira face, também conhecida como face infinita, adicionada no modelo como face de referência do sólido) e três listas constituídas por todos os vértices, arestas e faces deste sólido. A Figura 5.5 demonstra os principais métodos e atributos presentes nesta classe.

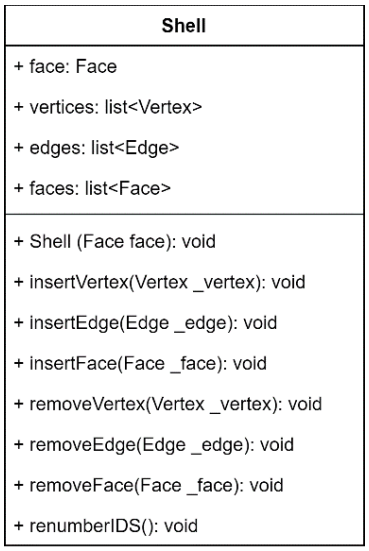


Figura 5.5 - Classe *Shell*

Todos os elementos topológicos armazenados no sólido apresentam dois atributos (*prev* e *next*) que apontam para objetos da mesma classe. A partir desses dos atributos *prev* e *next* é formado uma lista encadeada de objetos da mesma classe. Além disso, todo objeto pertencente ao sólido é constituído por um atributo denominado ID que cumpre o papel de identificador numérico deste objeto.

A classe *Shell* apresenta métodos para inserir ou remover das listas (*vertices, edges* e *faces*) uma determinada entidade topológica. Por meio desses métodos, esta classe faz o controle dos identificadores (IDs) de cada objeto através de contadores específicos. O sólido também possui um método chamado de *renumberIDS* que é utilizado para renumerar todos os IDs das entidades topológicas, sendo utilizado ao salvar o modelo.

Um vértice é um objeto da classe *Vertex* que pode ser criado através de um objeto da classe *Point* e de um objeto da classe *HalfEdge*, conforme ilustra a Figura 5.6.

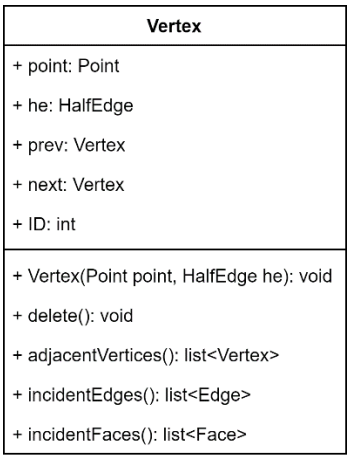


Figura 5.6 - Classe *Vertex*

O atributo *point*, da classe *Vertex*, guarda o endereço de memória para a respectiva entidade geométrica do vértice. A função *adjacentVertices*, como o próprio nome sugere, retorna uma lista contendo todos os vértices adjacentes a um determinado vértice. De forma análoga, os métodos *incidentEdges* e *incidentFaces* são utilizados para obter as listas que contêm respectivamente todas as arestas e faces incidentes a determinado vértice.

Tanto o vértice como as demais entidades topológicas que serão apresentadas posteriormente apresentam o método *delete.* Essa função reorganiza adequadamente a lista encadeada de *prev* e *next* quando uma entidade é removida do modelo, ou seja, quando ela é eliminada pelo usuário.

Uma aresta é criada a partir da classe *Edge* que é formada por um objeto da classe *Segment* e de pôr dois objetos da classe *HalfEdge*. A Figura 5.7 ilustra os principais métodos e atributos pertencentes a classe *Edge*.

De forma análoga ao vértice, toda aresta apresenta três métodos que retornam seus respectivos vértices e faces incidentes bem como as arestas adjacentes. O objeto da classe *Segment* é um segmento que guarda as coordenadas geométricas dos pontos pertencentes a uma determinada aresta.

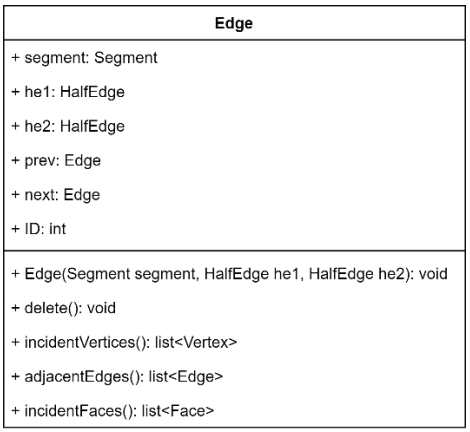


Figura 5.7 - Classe *Edge*

Uma face é um objeto da classe *Face*, composta por um atributo que armazena o sólido que a contêm (*shell*), por um ponteiro para a fronteira externa desta face (*loop*) e por uma lista que contêm o endereço de memória de todas as fronteiras internas pertencentes a esta face (*intLoops*). A Figura 5.8 apresenta os atributos e métodos da classe *Face*.

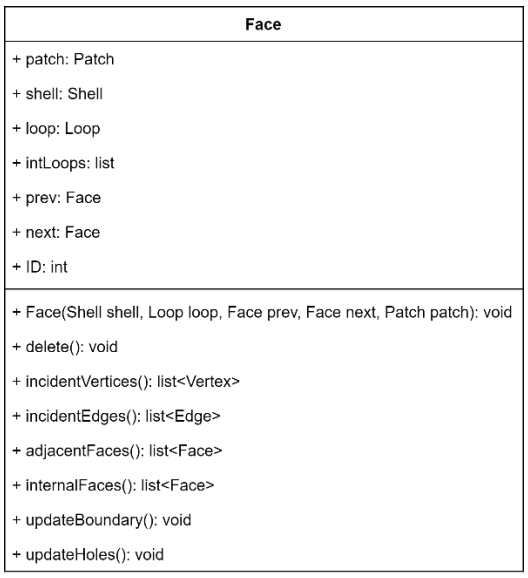


Figura 5.8 - Classe *Face*

Assim como as demais entidades topológicas não abstratas, a classe *Face* apresenta métodos que retornam listas contendo os elementos adjacentes e

incidentes. Além disso, o método *internalFaces* retorna uma lista contendo todas as faces internas.

Os métodos *updateBoundary* e *updateHoles* são utilizados para atualizar, respectivamente, os segmentos da fronteira externa da face e os segmentos das fronteiras de cada buraco presente nessa face. Esses métodos percorrem respectivamente as semi-arestas do *loop* externo e dos *loops* internos (*intLoops*) obtendo os segmentos e a orientação destes segmentos que são armazenados na entidade geométrica (*Patch*).

O *Patch* é o objeto que contém todas as informações geométricas referentes a face. Este elemento apresenta uma lista de segmentos e uma lista com indicadores da orientação desse segmento com relação a orientação do *loop* da face ao qual está relacionado. Além disso, o *Patch* também possui uma lista com informações geométricas dos buracos e outra contendo os dados da orientação dos segmentos que formam a fronteira de cada buraco. Como todas as outras entidades geométricas, o *Patch* também possui um ponteiro para a sua respectiva entidade topológica.

A classe *Loop* representa uma entidade abstrata que não existe geometricamente. A partir deste elemento é possível obter o acesso a um conjunto de semi-arestas que representam o percurso de uma fronteira externa ou interna presente em uma face. O *loop* externo de uma face representa um conjunto de semi-arestas, dos segmentos que formam a fronteira externa desta face, orientadas em sentido anti-horário. Já o *loop* interno é formado por conjunto de semi-arestas, orientadas no sentido horário, pertencentes aos segmentos ou pontos que estão compreendidos no interior da face.

O objeto da classe *Loop* é formado pelo atributo *face* que armazena o endereço de memória da face a qual ele pertence, pelo atributo *he* que aponta para uma das *half-edges* pertencentes ao *loop* e um atributo *isClosed* que indica se o *loop* é fechado ou aberto. A Figura 5.9 mostra um resumo desta classe.

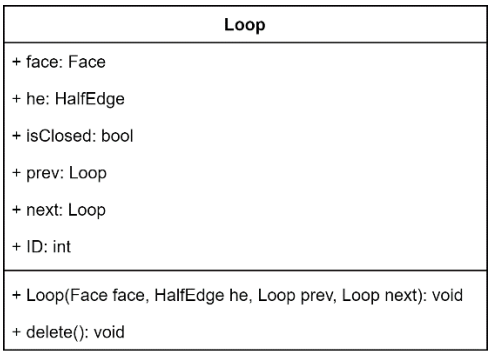


Figura 5.9 - Classe *Loop*

Assim como o *loop*, a semi-aresta também é uma entidade abstrata. Esse elemento é um objeto da classe *HalfEdge*, apresentando o atributo *vertex* que armazena o endereço de memória do vértice ao qual ela está orientada e os atributos *edge* e *loop* que armazenam os ponteiros para a aresta e o laço ao qual pertence. A Figura 5.10 ilustra um resumo do diagrama de classe dessa entidade.

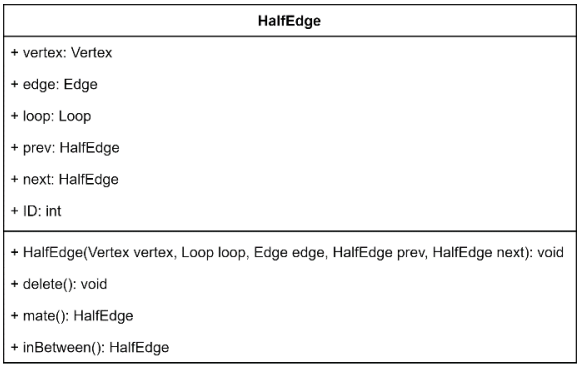


Figura 5.10 - Classe *HalfEdge*

A classe *HalfEdge* é a entidade principal na estrutura de dados, possibilitando a navegação pelas semi-arestas de cada *loop* através do atributo *next (*ou *prev)* que aponta para a próxima semi-aresta do *loop* no sentido anti-horário (ou horário) conforme já explicado na Seção 4.2.

Devido a semi-aresta permitir a conexão com os demais elementos topológicos presentes na estrutura de dados, a maioria das consultas topológicas é realizada por meio desta entidade. Para facilitar ainda mais as consultas topológicas,

a classe *HalfEdge* apresenta o método *mate* (que retorna a semi-aresta par, orientada no sentido oposto, pertencente a mesma aresta) e a função *inBetween* (que retorna a semi-aresta que está entre dois vértices e uma face).

5.3
Classe *HeView*

A classe *HeView* estabelece a comunicação entre a classe *HeModel* e a classe responsável por renderizar a cena do sólido modelado na tela, em geral, conhecido como *Canvas*. A classe *HeView* apresenta a função de leitor do modelo, acessando o objeto da classe *HeModel* e obtendo as informações necessárias para a visualização do sólido desenvolvido pelo usuário. Em seguida, essas informações são transmitidas à classe responsável pela renderização do modelo na tela. Dessa forma, a classe *HeView* é acessado pelo *Canvas* que obtém as informações do modelo para emitir a cena de visualização na tela.

O objeto desta classe é construído através de um *HeModel* que é passado para possibilitar à classe *HeView* o acesso a todas as informações deste modelo. A Figura 5.11 exibe um resumo dos métodos e atributos presentes na classe *HeView*. Todas as funções presentes nesta classe não alteram as informações presentes no modelo. Esses métodos apenas acessam a classe *HeModel* e filtram as informações de acordo com a necessidade. A seguir serão explicados alguns métodos presentes nesta classe.

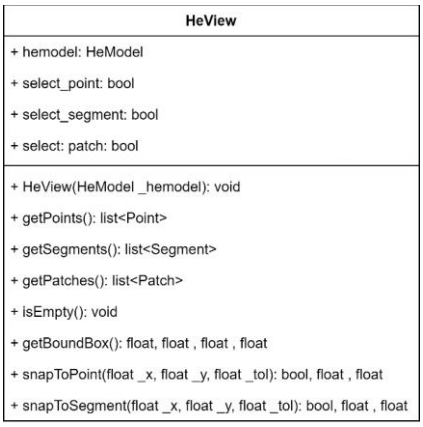


Figura 5.11 - Classe *HeView*

Os métodos *getPoints*, *getSegments* e *getPatches* são utilizados para obter respectivamente os pontos, segmentos e regiões do modelo. O método *isEmpty* é utilizado para verificar se o modelo está vazio. A função *getBoundBox* retorna as coordenadas mínimas e máximas de uma região retangular que engloba todas as entidades do sólido.

As funções *snapToPoint* e *snapToSegment* recebem as coordenadas de um ponto e uma tolerância e retornam um indicador, que sinaliza se existe uma entidade geométrica (segmento ou ponto) próxima as coordenadas (de acordo com a tolerância) fornecidas, e as coordenadas de um ponto ou segmento presente no modelo que estão localizadas a uma distância mais próxima possível das coordenadas dadas.

5.4
Operadores de Euler

Os operadores de Euler realizam ações de modelagem dentro da estrutura de dados, ou seja, a partir desses operadores as entidades topológicas são criadas ou destruídas. Assim, esses operadores apresentam a capacidade de criar sólidos (S), faces (F), laços - *loops* (L), arestas (E), semi-arestas - *half-edges* (HE) e vértices (V). Nesta seção são discutidos alguns detalhes sobre os operadores de Euler implementados na estrutura de dados, mais detalhes acerca destes operadores podem ser encontrados na Seção 4.4. A Tabela 2 apresenta os operadores implementados, o significado e as alterações que cada operador é capaz de realizar na estrutura de dados. Os operadores de Euler inversos trocam os sinais de cada termo da Tabela 2.

Tabela 2 - Operadores de Euler

Operadores	Significado	S	F	L	E	V	HE	Inverso
MVFS	Construir um sólido, uma face e um vértice	+1	+1	+1		+1	+1	KVFS
MVR	Construir um vértice isolado			+1		+1	+1	KVR
MEV	Construir uma aresta e um vértice				+1	+1	+2	KEV
MEF	Construir uma aresta e uma face		+1		+1		+2	KEF
MEKR	Construir uma aresta e destruir um *loop*			-1	+1		+2	KEMR
MVSE	Construir um vértice e dividir uma aresta				+1	+1	+2	KVJE

Nenhum destes operadores implementados faz qualquer tipo de busca global na estrutura de dados. As modificações efetuadas por estes operadores são todas locais e as buscas globais são feitas em um nível superior gerenciado pela classe *HeController*. A inserção e a remoção de entidades do modelo também são realizadas por meio de operadores auxiliares criados em um nível acima deste.

Os operadores de Euler são classificados como de baixo nível porque lidam diretamente com entidades topológicas abstratas, tais como semi-arestas e *loops*, e entidades topológicas físicas, tais como vértices, arestas e faces. As funções de alto nível, por outro lado, lidam com entidades geométricas, tais como pontos e curvas.

Cada operador de Euler é uma classe que apresenta dois métodos além do seu construtor. Esses dois métodos são chamados de *execute* e *unexecute.* O primeiro método tem a função de executar o operador de Euler e o último apresenta o papel de desfazer tudo o que foi feito pelo primeiro.

Esses dois métodos (*execute* e *unexecute*) foram desenvolvidos para possibilitar o usuário desfazer ou refazer qualquer operação criada durante a

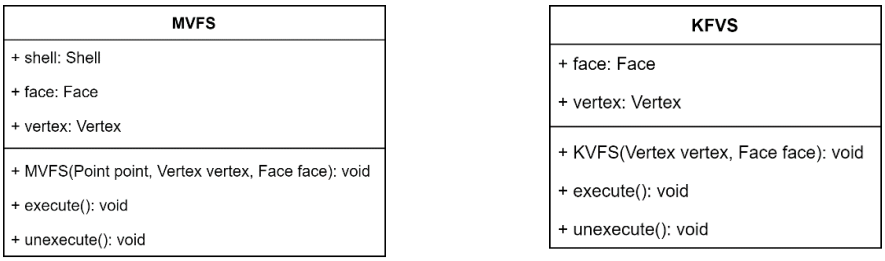
modelagem do sólido (esse assunto é detalhado na Seção 5.6). A função *unexecute* de cada operador cria o respectivo operador de Euler inverso e executa o mesmo.

Por exemplo, o método *unexecute* do operador MEV cria um objeto da classe KEV (inverso de MEV) passando os parâmetros adequados existentes no operador MEV e por fim esse objeto da classe KEV será executado e todas as modificações feitas pelo operador MEV serão desfeitas. Caso seja necessário refazer o operador MEV, deve-se chamar o método *execute* do objeto da classe MEV que todo processo será refeito.

A Figura 5.12 apresenta as classes dos operadores de Euler MFVS e KFVS. O operador MVFS é responsável por adicionar um sólido, um vértice e uma face na estrutura de dados. Este operador é usualmente utilizado para inicializar a estrutura de dados quando o modelo está vazio. O operador inverso ao MVFS é o operador KFVS que remove todas as entidades criadas pelo primeiro operador.

O operador MVFS envolve a criação de uma face que não apresenta fronteira externa, ou seja, sua fronteira externa é infinita. Para a representação da fronteira infinita dessa face foi implementada no algoritmo um "*loop* virtual" que não apresenta semi-arestas. Esse "*loop* virtual" foi criado apenas para padronizar no código implementado que todas a faces tenham um *loop* externo, facilitando a implementação do código e evitando assim a necessidade de conferir a existência de um *loop* externo em cada face.

Deve-se observar que é simples identificar o (único) *loop* virtual de um sólido: é aquele que não possui *half-edge*. Esse fato também serve para identificar a (única) face infinita de um sólido: é aquela que possui um *loop* virtual. Entretanto, nenhum tratamento especial deve ser dado para a face infinita sem *loop* externo (isto é, com um *loop* virtual), pois qualquer construção topológica que fecha uma região nesta face sempre cria a nova face na região fechada. Dessa maneira, a face infinita criada pelo operador MVFS é sempre preservada como a face externa infinita por qualquer outro operador de Euler que manipule faces.



(a) (b)

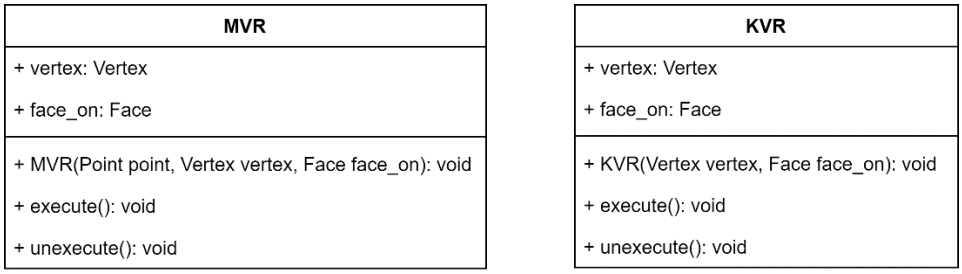
Figura 5.12 - Classes MVFS e KVFS

Todas as classes dos operadores de Euler desenvolvidas apresentam duas formas de criação que se diferenciam de acordo com os seus parâmetros de entrada. A primeira forma de criação tem como parâmetros de entrada informações geométricas, tais como ponto e curva. A segunda forma de criação tem como parâmetros de entrada entidades topológicas que são criadas previamente. A primeira forma é utilizada durante o processo de criação de novos elementos geométricos (pontos e segmentos) resultando na criação de novas entidades topológicas. A segunda forma é utilizada ao se desfazer ou refazer uma ação de modelagem (explicado na Seção 5.6), pois nesses processos não ocorre a criação de novas entidades topológicas físicas.

Um objeto da classe MVFS pode ser criado de duas formas distintas. A primeira forma é passando um ponto (*point*) e neste caso o construtor dessa classe criará os objetos da classe *Shell*, *Face* e *Vertex.* A segunda forma é passando um vértice (*vertex*) e uma face (*face*) e neste caso não é necessário a criação das entidades topológicas previamente citadas. Para a criação do operador KFVS basta indicar o vértice e a face a serem removidos da estrutura de dados.

O método *execute* do operador MVFS e de outros operadores que lidam com a construção de entidades sempre criam *loops* e semi-arestas. Além disso, essa função ajusta as entidades envolvidas de forma adequada através da modificação das propriedades das entidades topológicas, tais como *prev* e *next*. Já o método *execute* do operador KVFS e de outros operadores que lidam com a remoção de entidades destroem os *loops* e semi-arestas criados e ajustam as entidades adjacentes que não foram removidas para manter a consistência da estrutura de dados.

O operador de Euler MVR tem a função de criar um vértice isolado, localizado sobre uma face, e o seu operador inverso apresenta a capacidade de remover este vértice. De maneira análoga a classe MFVS, o operador MVR pode ser criado de duas formas distintas. A primeira forma de criação deste operador é passando um ponto (*point*). A segunda forma é indicando um vértice (*vertex*) e a face (*face_on*) na qual este vértice está localizado. Da mesma maneira, o seu operador inverso é criado passando o vértice (*vertex*) a ser removido e a face (*face_on*) na qual ele está localizado. A Figura 5.13 ilustra um resumo das classes MVR e KVR.

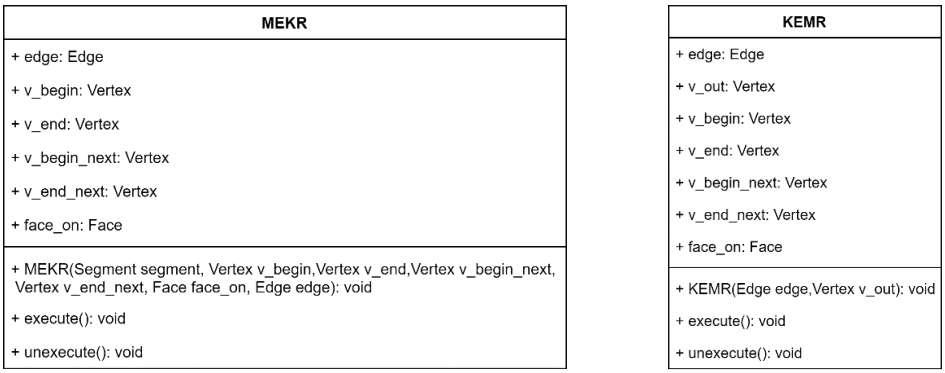


(a) (b)

Figura 5.13 - Classes MVR e KVR

O operador MEKR une dois *loops* distintos por meio da criação de uma aresta que conecta dois vértices (um de cada *loop*). Este operador pode ser criado de duas formas distintas. A primeira forma é indicando o segmento (*segment*) que contém as informações geométricas da aresta a ser criada, o vértice inicial (*v_begin*) e o vértice final (*v_end*) da nova aresta, dois vértices adjacentes ao vértice inicial e final (*v_begin_next* e *v_end_next*) e a face (*face_on*) sobre a qual a aresta está sendo inserida.

Outra forma de criar o operador MEKR é indicando uma aresta em vez do segmento, neste caso não existe a necessidade de criação de uma nova aresta, e todas as outras informações citadas anteriormente. Na Figura 5.14 estão apresentadas as classes MEKR e KEMR.



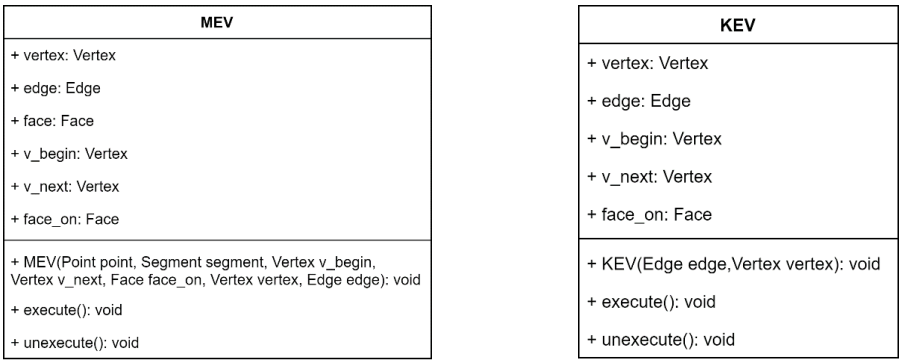
(a) (b)

Figura 5.14 - Classes MEKR e KEMR

O operador KEMR tem a função de remover uma aresta dividindo um *loop* em dois. Para realizar essa operação é necessário indicar a aresta a ser removida e um vértice que incide nesta aresta. Caso a aresta pertença ao *loop* externo da face, o vértice a ser passado deve ser o vértice que permanecerá no *loop* externo após a execução da operação. Durante a execução do operador KEMR são armazenadas todas as informações necessárias (*v_begin*, *v_end*, *v_begin_next*, *v_end_next* e *face_on*) para a utilização do método *unexecute* que cria o operador MEKR.

O operador MEV adiciona um vértice e uma aresta na estrutura de dados. Da mesma forma como outros operadores já citados, o objeto da classe MEV pode ser criado de duas formas distintas. A primeira forma é passando um ponto (*point*) e um segmento (*segment*), e então um novo vértice e uma nova aresta são criadas. A segunda forma é indicando um vértice e uma aresta previamente criados que já contém as informações geométricas necessárias. Além disso, para as duas formas de criação é necessário informar o vértice incidente da nova aresta que já exista no modelo (*v_begin*), o vértice próximo ao novo vértice (*v_next*) e a face (*face_on*) onde a nova aresta está localizada.

A classe KEV cumpre o papel de desfazer todo o processo realizado pelo operador MEV, ou seja, remove a aresta e o vértice que foram adicionados na estrutura de dados. Para criar este operador é necessário indicar a aresta e o vértice que serão removidos. Durante a execução do operador KEV são armazenadas todas as informações necessárias para que seja possível utilizar o método *unxecute* que cria uma operação MEV e a executa. A Figura 5.15 apresenta as classes MEV e KEV.

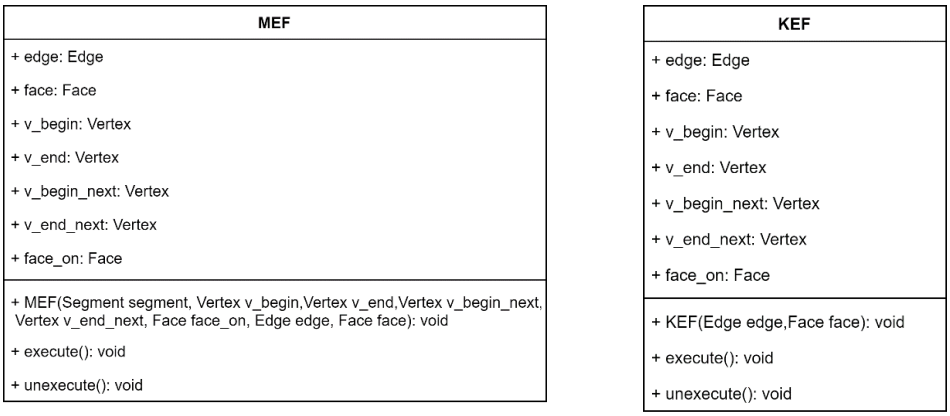


(a) (b)

Figura 5.15 - Classes MEV e KEV

O operador MEF é utilizado para adicionar uma aresta e uma face na estrutura de dados. Este operador pode ser construído de duas formas distintas, a partir de um segmento (e então uma nova aresta e face serão criadas) ou de uma aresta e uma face. Além disso, é necessário indicar o vértice inicial (*v_begin*), o vértice final (*v_end*), dois vértices que são vértices adjacentes ao vértice inicial e final (*v_begin_next* e *v_end_next*) e a face (*face_on*) sobre a qual a aresta está sendo inserida.

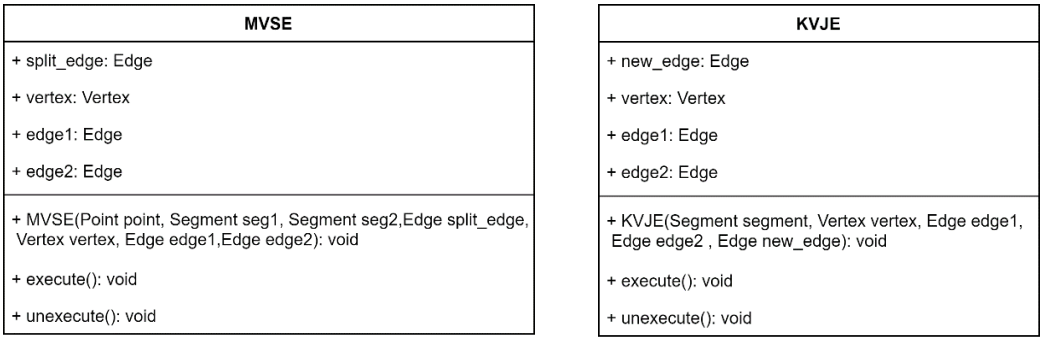
A classe KEF realiza o processo inverso ao MEF, removendo uma aresta e uma face da estrutura de dados. Para criar um objeto da classe KEF basta indicar a aresta e a face que serão removidas. De maneira análoga ao MEV, o operador KEF também armazena as informações necessárias para a utilização do método *unexecute.* A Figura 5.16 ilustra as classes MEF e KEF.



(a) (b)

Figura 5.16 - Classes MEF e KEF

O operador MVSE, cria um vértice dividindo uma aresta em duas. Este operador pode ser criado por meio de um ponto e dois segmentos (neste caso serão criados um vértice e duas arestas) ou através de um vértice e duas arestas que contenham as informações geométricas adequadas do ponto e do segmento a ser divido. Com isso, a aresta inicial é removida e as duas novas arestas são adicionadas na estrutura de dados. Além das informações previamente citadas, para a criação da operação MVSE é necessário informar a aresta (*split_edge*) que será dividida. A Figura 5.17 ilustra o diagrama da classe MVSE bem como de seu operador inverso KVJE.



(a) (b)

Figura 5.17 - Classes MVSE e KVJE

A operação KVJE tem a capacidade de desfazer tudo que é feito na operação MVSE removendo o vértice e unindo a aresta, ou seja, adicionando a aresta previamente removida a estrutura de dados e removendo as duas arestas previamente adicionadas. Esta operação é criada por meio de um segmento ou de uma aresta que contenha as informações geométricas adequadas dos dois segmentos que serão unidos. Além disso, é necessário informar o vértice a ser removido (*vertex*) e as duas arestas que serão unidas (*edge1* e *edge2*).

5.5
Operadores auxiliares

Os operadores auxiliares são um conjunto de classes que cumprem funções especificas dentro da estrutura de dados. Como por exemplo, apresentam a capacidade de adicionar ou remover entidades geométricas do modelo, adicionar ou remover atributos de qualquer entidade topológica e outras funções que colaboram para o funcionamento adequado da estrutura de dados. Além disso, esses operadores auxiliares oferecem uma maior flexibilidade a estrutura de dados possibilitando desfazer ou refazer um conjunto variado de ações de modelagem do usuário.

Em geral, os operadores auxiliares que apresentam a função de inserir ou remover uma entidade do modelo são utilizados após a estrutura de dados sofrer alguma modificação por um operador de Euler. Os operadores auxiliares apresentam o mesmo padrão de classe adotado pelos operadores de Euler. Ou seja, além do seu construtor cada operador auxiliar possui dois métodos denominados de *execute* e *unexecute*. Assim como nos operadores de Euler, esses dois métodos foram criados para possibilitar desfazer ou refazer qualquer comando feito pelo usuário.

Por exemplo, o método *execute* do operador *InsertVertex* insere um vértice ao modelo e o seu método *unexecute* remove o vértice que foi previamente inserido. Os principais operadores auxiliares utilizados na estrutura de dados e a descrição do que fazem estão listados abaixo. A seguir alguns operadores auxiliares serão mais detalhados.

- *InsertVertex* - Insere um vértice no modelo;
- *InsertEdge* - Insere uma aresta no modelo;
- *InsertFace* - Insere uma face no modelo;
- *InsertShell* - Insere um sólido no modelo;
- *RemoveVertex* - Remove um vértice do modelo;
- *RemoveEdge* - Remove uma aresta do modelo;
- *RemoveFace* - Remove uma face do modelo;
- *RemoveShell* - Remove um sólido do modelo;

- *Flip* - Inverte as semi-arestas de uma aresta;
- *MigrateLoops* - Migra os *loops* internos de uma face para outra;
- *CreatePatch* - Transforma um buraco em uma face;
- *DelPatch* - Transforma uma face em um buraco;
- *SetAttribute* - Adiciona um atributo em um vértice ou uma aresta ou uma face;
- *UnSetAttribute* - Remove um atributo em um vértice ou uma aresta ou uma face;
- *DelAttribute* - Deleta um atributo criado pelo usuário;
- *SetMesh* - Adiciona uma malha em uma face;
- *DelMesh* - Remove uma malha de uma face.

O operador *MigrateLoops* é sempre utilizado após a execução do operador MEF. Essa operação transmite os *loops* internos, que necessitam ser migrados, de uma face para uma nova face criada pelo operador MEF. Além disso, este operador também ajusta todos os atributos dos *loops* a serem migrados, tais como os atributos *prev* e *next*, de forma adequada.

O operador *Flip* pode ser utilizado quando for necessário inverter as *half-edges* de determinada aresta para manter a consistência dos dados geométricos com os dados topológicos (que deve estar de acordo com a orientação anti-horária). O algoritmo segue a definição que todo o *loop* da semi-aresta armazenada no atributo *he1* da classe *Edge* apresente a mesma orientação do conjunto de coordenadas geométricas que definem o segmento. A *half-edge* (*he1*) da aresta sempre está orientada na mesma direção do conjunto de coordenadas que definem o segmento.

Caso a orientação do conjunto de coordenadas geométricas do segmento não esteja de acordo com a orientação do *loop* da primeira *half-edge* da aresta (*he1*), as *half-edges* da aresta devem ser invertidas. Este operador pode ser utilizado após alguma operação de Euler que crie uma aresta (mais detalhes sobre esse operador auxiliar serão fornecidos na Seção 5.7.2).

Os operadores *CreatePatch* e *DelPatch* modificam o atributo *isDeleted* do *Patch* de uma face. Esse atributo indica se a face será renderizada na tela ou não. Com isso, esses operadores apresentam a capacidade de transformar a face em um buraco ou um buraco em uma face.

5.6
Classe *UndoRedo*

A classe *UndoRedo*, apresentada na Figura 5.18, é responsável por armazenar e gerenciar uma quantidade específica de comandos feito pelo usuário. A partir dessa classe é possível desfazer e refazer todos os operadores de Euler e operadores auxiliares gerados a partir da interação do usuário com o aplicativo.

Cada comando é representado por um conjunto de operações (formado por operadores de Euler e operadores auxiliares). Esses comandos são armazenados em duas listas. A primeira lista (*undocommands*) é constituída de comandos que podem ser desfeitos pelo usuário e a segunda lista (*redocommads*) é formada por comandos que podem ser refeitos. O objeto da classe *UndoRedo* é criado passando um número que indica o limite (*limit*) de comandos que serão armazenados em cada lista. A seguir são detalhados os principais métodos presentes nessa classe.

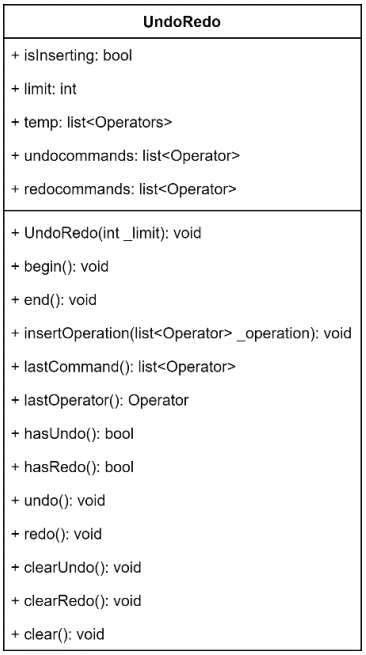


Figura 5.18 - Classe *UndoRedo*

O método *begin* é utilizado como forma de inicialização, permitindo a inserção de operações pelo método *insertOperation*. A função *begin* ajusta o

atributo *isInserting* para *True* possibilitando que as operações sejam armazenadas em forma de pilha em uma lista (*temp*). O atributo *temp* é uma lista temporária responsável por armazena o último comando feito pelo usuário.

Para armazenar os comandos na lista de *undocommands* é necessário finalizar a coleta de operações utilizando o método *end* que ajusta o atributo *isInserting* para *False*. Além disso, essa função também verifica se a lista de *undocommands* ultrapassou o limite ao adicionar um novo comando. Caso o limite de comandos armazenados seja ultrapassado, o primeiro comando adicionado nessa lista é removido.

A função *lastCommand* retorna uma lista contendo todos os operadores utilizados no último comando feito pelo usuário. Já o método *lastOperation* retorna apenas o último operador utilizado no último comando feito pelo usuário.

As funções *hasUndo* e *hasRedo* são utilizadas como forma de verificação que sinalizam, respectivamente, a possibilidade de desfazer ou refazer um comando. Essas verificações são feitas analisando a hipótese da lista de *undocommands* e *redocommands* estarem vazias.

Os métodos *undo* e *redo* atuam gerenciando as listas *undocommands* e *redocommands.* O método *undo* remove o último comando adicionado na lista de comandos que podem ser desfeitos e adiciona este na lista de comandos que podem ser refeitos. De maneira análoga o método *redo* remove o último comando adicionado na lista de comandos que podem ser refeitos e adiciona este comando na lista de comandos que podem ser refeitos.

Nessa classe também existem métodos responsáveis por remover todos os comandos das listas *undocommands* e *redocommands.* A função *clearUndo* remove todos os comandos que podem ser desfeitos e a função *clearRedo* remove todos os comandos que podem ser refeitos. E por fim, a função *clear* remove os comandos de ambas as listas.

5.7
Classe *HeController*

A classe *HeController* gerencia e processa as interações do usuário com a estrutura de dados. Por conta disso, essa classe tem acesso a maioria dos módulos presentes na biblioteca HETOOL. Nessa seção são detalhados os principais atributos e métodos pertences na classe *HeController* conforme é demonstrado na Figura 5.19.

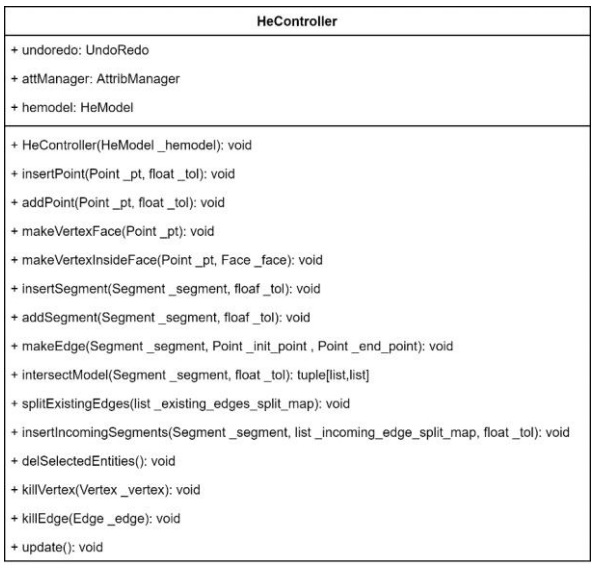


Figura 5.19 - Classe *HeController*

Os principais atributos desta classe são objetos de classes especificas presentes na estrutura de dados. O *undoredo* é uma instancia da classe *UndoRedo* explicada na Seção 5.6. O *attManager* é uma instancia da classe *AttribManager* e age como o gerenciador de atributos da biblioteca HETOOL (mais detalhes sobre essa classe podem ser encontrados na Seção 5.8). O *hemodel* (instancia da classe *HeModel*) é o modelo que será acessado e modificado pelo *HeController*. Esse modelo recebe os dados processados pelo controlador (entidades topológicas e geométricas do sólido) e os armazena.

Os métodos do *HeController*, utilizados na modelagem do sólido, são funções de alto nível que conectam os comandos do usuário com o baixo nível formado pelos operadores de Euler e os operadores auxiliares. Essas funções de alto nível realizam verificações topológicas e geométricas na estrutura de dados.

Os métodos *insertPoint* e *insertSegment* possibilitam, respectivamente, a inserção de pontos e segmentos na estrutura de dados. Já o método *delSelectedEntites* lida com a remoção de entidades do modelo. A Figura 5.20 ilustra um resumo dos métodos da API (*Application Programming Interface*) relacionados com a criação de novas entidades e a conexão entre as funções de alto nível (*High Level*), funções de nível médio ou intermediário (*Middle Level*) e operadores de baixo nível (*Low Level*). Nas seções posteriores, são detalhados os principais métodos da classe *HeController*.

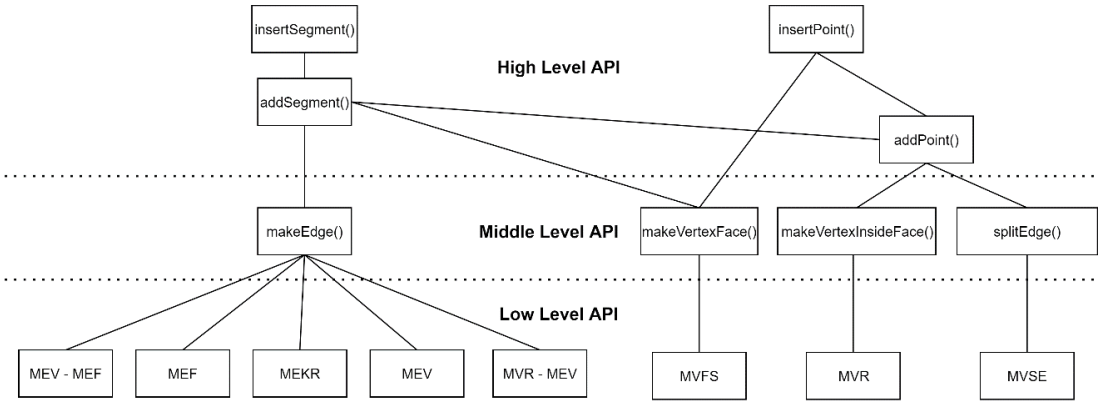


Figura 5.20 - Métodos da API utilizados para adicionar um ponto ou segmento

As funções de alto nível são responsáveis pelas verificações geométricas globais, levando-se em consideração todas as entidades geométricas presentes no modelo. No nível das funções intermediárias, todas as verificações geométricas já foram realizadas e a partir deste ponto realizam-se apenas, caso necessário, verificações topológicas. Por fim, chega-se no baixo nível onde são realizadas modificações locais na estrutura de dados topológica provocadas pelos operadores de Euler.

5.7.1
Método *insertPoint*

O método *insertPoint* recebe um ponto ou uma lista contendo as coordenadas geométricas de um ponto e uma tolerância utilizada em análises geométricas. Inicialmente, esta função verifica se o modelo está vazio. Caso o modelo esteja vazio o método *makeVertexFace* é chamado e dentro dele ocorre a criação da operação de Euler MVFS e de outras operações auxiliares que adicionam um sólido, uma face e um vértice ao modelo. Essa primeira face adicionada ao modelo não apresenta fronteiras externas e, portanto, pode ser denominada de face infinita.

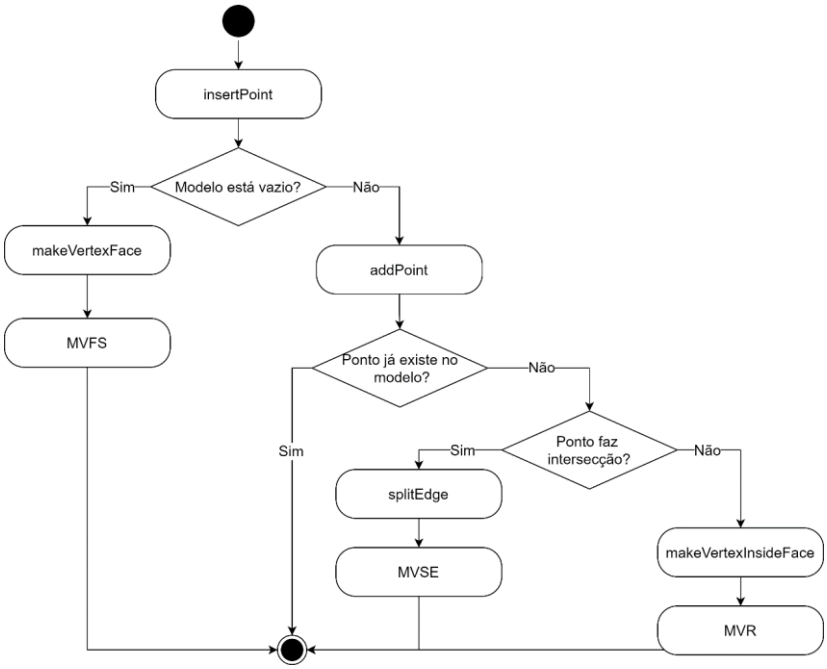


Figura 5.21 - Fluxograma do método *insertPoint*

Caso o modelo não esteja vazio, o método *addPoint* será executado. Esta função verifica se o ponto que o usuário inseriu já existe no modelo. Caso isso ocorra, a operação é encerrada e nenhuma modificação será feita na estrutura de dados. Caso contrário, essa função verifica a possibilidade de interseção desse ponto com as demais entidades geométricas presentes no modelo.

Se o ponto realizar interseção com algum segmento existente no modelo então esse segmento será divido em dois através do método *splitEdge* que cria e executa a operação de Euler MVSE. Além disso, outras operações auxiliares são chamadas realizando a remoção do antigo segmento e adicionando um ponto e dois segmentos (originado pela divisão do segmento removido) ao modelo.

Caso nenhuma das verificações citadas anteriormente forem satisfeitas então o ponto inserido pelo usuário está dentro de uma face. Com isso, o método *makeVertexInsideFace* é chamado para criar e executar a operação de Euler MVR e a operação auxiliar que adiciona o ponto recebido pela função *addPoint* ao modelo.

Após uma das operações de Euler supracitadas e seus respectivos operadores auxiliares serem criados e executados, o método *update* é chamado. Essa função realiza a atualização dos segmentos que compõem o contorno de cada face do modelo e seus respectivos buracos. Essa atualização ocorre através da navegação pelas semi-arestas de cada *loop* da face. Dessa forma, para cada semi-aresta percorrida é armazenado a aresta correspondente e a orientação desta aresta em relação ao sentido do *loop* da semi-aresta. A Figura 5.21 ilustra o funcionamento da função *insertPoint* por meio de um diagrama.

5.7.2
Método *insertSegment*

O método *insertSegment* recebe um segmento ou uma lista contendo as coordenadas geométricas de um segmento e uma tolerância que é utilizada em análises geométricas. Dentro desta função, inicialmente é verificado se o segmento dado realiza interseção com algum trecho pertencente ao próprio segmento, ou seja, se o segmento apresenta auto interseção. Caso exista uma ou mais auto interseções, o segmento adicionado é subdividido em segmentos menores. Em seguida, o método *addSegment* é chamado para cada subsegmento originado das auto interseções. Caso não exista auto interseção, então a função *addSegment* é chamada passando o segmento original.

De forma análoga ao método *insertPoint*, a função *addSegment* também verifica se o modelo está vazio. Caso essa condição for satisfeita, o algoritmo verifica se o segmento é fechado, ou seja, se o ponto inicial do segmento é igual ao ponto final (essa verificação é feita utilizando a tolerância previamente fornecida). Se o segmento for fechado, os métodos *makeVertexFace* e *makeEdge* são chamados. Caso contrário, as funções *makeVertexFace*, *makeVertexInsideFace* e *makeEdge* são chamadas.

Se o modelo não estiver vazio, o algoritmo também verifica a possibilidade do segmento ser fechado. Caso o segmento seja fechado, as funções *addPoint*, *intersecModel*, *splitExistingEdges* e *insertIncomingSegments* são chamadas. Caso contrário, todas as funções acima são chamadas com exceção do método *addPoint*. Todo o processado do método *insertSegment* está ilustrado na Figura 5.22.

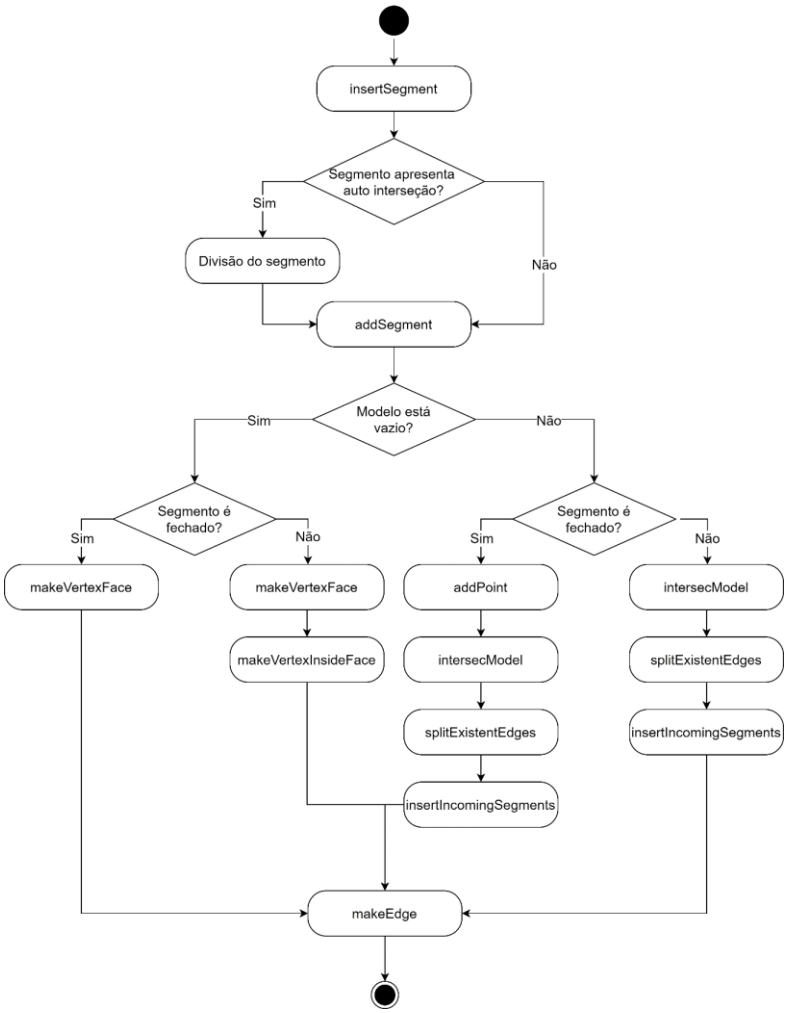


Figura 5.22 - Fluxograma do método *insertSegment*

A função *intersetModel* é responsável por verificar a existência de interseções entre o segmento adicionado pelo usuário e os pontos e segmentos existentes no modelo. Caso existam interseções, os parâmetros de interseção correspondente aos segmentos existentes no modelo e ao segmento recebido pela função *addSegment* são computados.

Esses parâmetros de interseção são então recebidos pelas funções *splitExistentEdges*, que como o nome sugere faz a divisão das arestas existentes no modelo (através da função *splitEdge* já mencionada anteriormente). O método *insertIncomingSegments* também recebe os parâmetros de interseção e subdivide o segmento recebido no método *addSegment.* Por fim, esta função chama o método *makeEdge* passando cada um dos subsegmentos.

O método *makeEdge* recebe um segmento, o ponto inicial (*Pi*) e o ponto final (*Pf*) deste segmento. Essa função realiza a conexão do processamento de alto nível (todas as verificações citadas anteriormente) com os operadores de Euler adequados. A partir deste ponto todas as verificações geométricas de interseção do segmento com o modelo já foram computadas e existem apenas quatro casos possíveis (ver Figura 5.23). Esses casos estão listados abaixo:

- Caso 1: o ponto inicial e final já pertencem ao modelo;
- Caso 2: o ponto inicial pertence ao modelo e o ponto final não pertence;
- Caso 3: o ponto final pertence ao modelo e o ponto inicial não pertence;
- Caso 4: Nem o ponto inicial e nem o final pertencem ao modelo.

Para o primeiro caso, é possível criar e executar os operadores de Euler MEF, MEKR e, se o segmento for fechado, é feito um tratamento especial com os operadores MEV e MEF. Para que os operadores MEF ou MEKR sejam criados, é necessário que o segmento seja aberto. Caso os vértices correspondentes aos pontos *Pi* e *Pf* pertençam ao mesmo *loop*, então é construído o operador MEKR. Caso contrário, o operador MEF é criado. Em ambos os casos existe a possibilidade de construção e execução do operador auxiliar FLIP.

O operador FLIP é criado e executado caso exista a necessidade de inverter as *half-edges* da aresta para manter a consistência dos dados geométricos do

segmento com a orientação do *loop*. A *half-edge* (*he1*) da aresta deve ser a semi-aresta cujo *loop* esteja na mesma orientação do conjunto de coordenadas geométricas que definem o segmento.

Caso o segmento adicionado na função *makeEdge* seja fechado, então ele é dividido em dois trechos e as operações MEV e MEF são criadas e executadas em sequência (existe também a possibilidade de criação de um operador FLIP). Logo, a estrutura de dados não permite a criação de segmentos fechados, sendo cada segmento fechado subdivido em dois segmentos interligados.

Essa limitação ocorre devido ao operador MEF funcionar de forma adequada apenas com a *half-edge* mais à direita dos vértices utilizados na operação. Caso exista dois segmentos fechados ligados ao mesmo vértice, a operação que realiza o cálculo geométrico para pegar essas *half-edges* torna-se inválida devido a orientação das *half-edges* ficarem ordenadas de forma aleatória. Com isso, optou-se por restringir a criação de segmentos fechados durante o processo de modelagem.

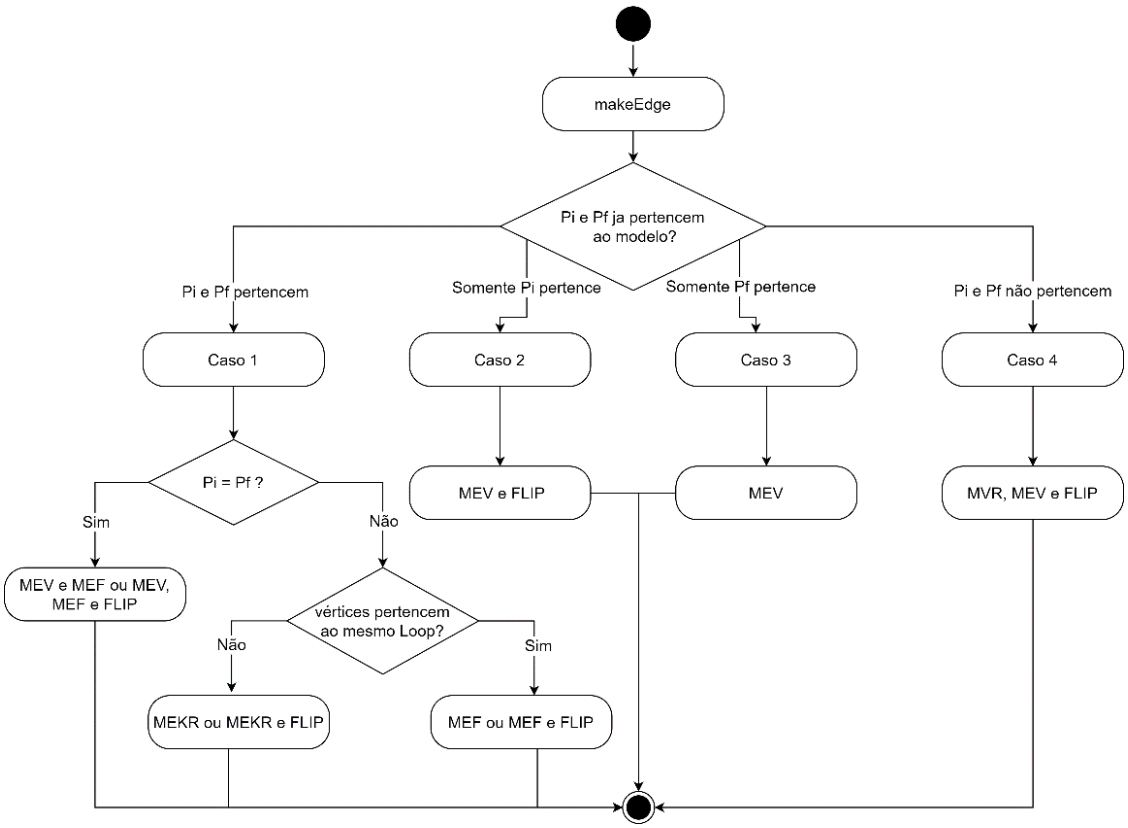


Figura 5.23 - Fluxograma do método *makeEdge*

Para o segundo e terceiro caso, o operador a ser construído será o MEV. No segundo caso, é necessário inverter as *half-edges* da aresta através do operador auxiliar FLIP para manter a consistência dos dados geométricos com os dados topológicos.

Por fim, se nem o ponto inicial e nem o ponto final do segmento pertencer ao modelo (caso 4), são criados os operadores de Euler MVR e MEV. Além disso, utiliza-se o operador FLIP pelo mesmo motivo explicado anteriormente. A Figura 5.23 exibe um resumo da função *makeEdge* que faz a conexão das verificações de alto nível com os operadores de Euler (baixo nível). Para todos os quatro casos são chamados outros operadores auxiliares que adicionam as entidades topológicas e geométricas adequadas no modelo.

5.7.3
Método *delSelectedEntities*

O método *delSelectedEntites* gerencia a remoção das entidades topológicas e geométricas do modelo que foram previamente selecionadas e excluídas pelo usuário. Para isso, a função coleta do modelo todas as entidades selecionadas e em seguida elimina estes elementos em uma ordem predefinida. A ordenação utilizada na remoção das entidades segue as seguintes premissas:

- Todas as arestas são removidas antes dos vértices;
- Para a remoção de um vértice é necessário, antes disso, eliminar as arestas adjacentes ao vértice selecionado (caso existam);
- Caso o vértice selecionado apresente apenas duas arestas adjacentes, o vértice é removido e as duas arestas são unidas por meio do método *joinEdges*. Caso contrário, todas as arestas são removidas e em seguida o vértice é eliminado;
- Ao eliminar uma aresta, os vértices incidentes a esta aresta que não pertencem a nenhuma outra aresta serão eliminados;
- As faces são removidas de forma automática ao eliminar uma aresta de um *loop* fechado;

- Caso a face apresente apenas duas arestas, ao remover o vértice todas as arestas são removidas bem como a face e o vértice;
- É possível remover o visual de uma face para formar um buraco. Essa remoção visual da face consiste em um indicador (*isDeleted*) que determina se a face deve ser renderizada no canvas ou não.

A função *delSelectedEntities* seguindo as premissas previamente citadas, conecta-se com dois outros métodos presentes no *HeController*: *killEdge* e *killVertex*. Estes dois métodos gerenciam, respectivamente, a remoção de arestas e vértices do modelo. A remoção visual das faces ocorre por meio de um operador auxiliar chamado de *DelPatch*.

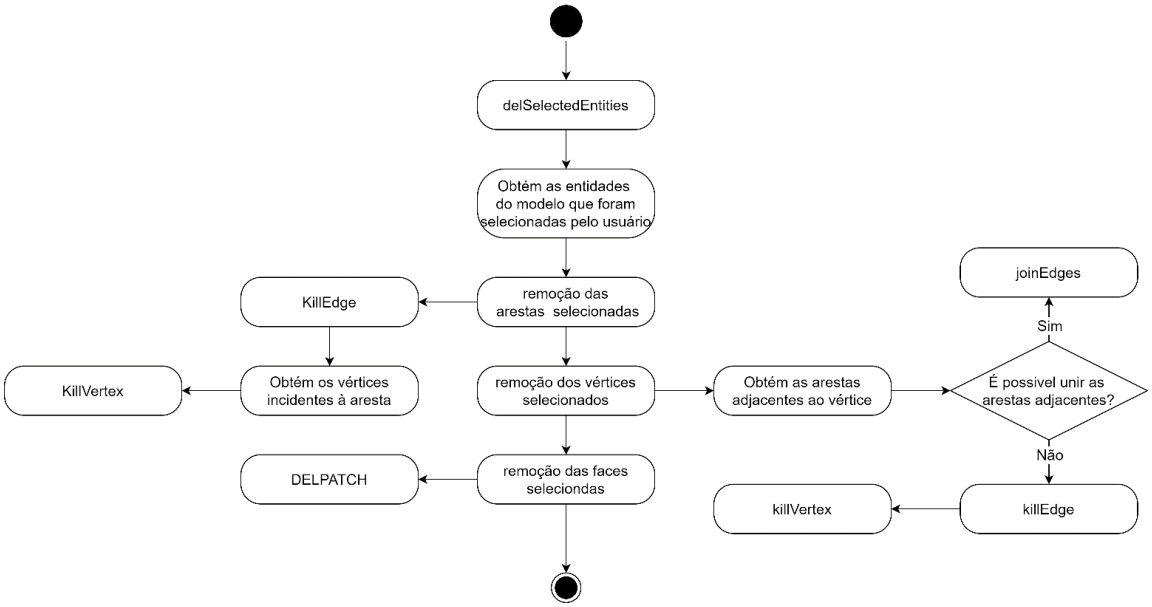


Figura 5.24 - Fluxograma do método *delSelectedEntities*

A função *KillEdge* apresenta dois casos possíveis. O primeiro caso verifica se as *half-edges* da aresta apresentam *loops* distintos. Caso isso aconteça, então o operador KEF é criado e executado junto com operadores auxiliares que removem tanto a aresta selecionada quanto a face. Caso contrário, o operador KEMR é construído e executado realizando a remoção da aresta e a criação de um *loop*. Em

ambos os casos, pode ser necessário chamar o operador auxiliar FLIP para manter a consistência dos parâmetros geométricos da aresta com os dados topológicos.

A função *joinEdges* une dois segmentos incidentes a um vértice que será eliminado. Essa união ocorre por meio da criação de uma polilinha que contém todos os pontos pertencentes aos dois segmentos adjacentes. O operador de Euler responsável por essa operação é o KVJE. Neste método é possível também ocorrer a criação do operador auxiliar FLIP.

Como explicado anteriormente, antes de excluir um vértice é necessário remover todas as arestas incidentes a este elemento. Com isso, no método *killVertex* é verificado se o vértice apresenta arestas incidentes. Caso exista uma ou mais arestas conectadas ao vértice a função é interrompida. Caso contrário, existem duas possibilidades que se conectam com os operadores KVFS ou KVR. Se o número de vértices presentes no modelo for igual a um, o operador KVFS é criado e executado removendo o vértice bem como a face (face infinita) e o sólido. Caso contrário, o operador KVR é chamado removendo apenas o vértice.

5.8
Classe *AttribManager*

A modelagem geométrica de sólidos é genérica e o que diferencia uma aplicação de outra são os atributos de modelagem. Com isso, vem a importância da biblioteca HETOOL ter um mecanismo para a criação e configuração de novos atributos. A classe *AttribManager* é o que torna esse mecanismo possível funcionando como um gerenciador de atributos responsável por armazenar todos os atributos criados pelo usuário. A Figura 5.25 exibe um resumo dos principais componentes pertencentes a esta classe.

Essa classe possui um conjunto de métodos que permitem inserir e remover atributos bem como alterar os valores destes. A classe é composta por duas listas (*prototypes* e *attributes*) que armazenam respectivamente os protótipos de atributos e os atributos criados pelo usuário. Os protótipos de atributos são os diferentes tipos de atributos que podem ser criados pelo usuário.

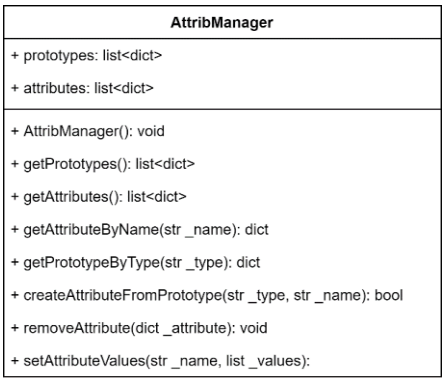


Figura 5.25 - Classe *AttribManager*

Um atributo representa uma cópia de um protótipo de atributo que ao ser criado recebe um nome específico. Cada atributo é criado e transmitido na forma de um dicionário (classe *dict* em Python que apresenta uma estrutura constituída por *chave:valor* muito similar ao formato JSON). Um atributo apresenta propriedades que são formadas pelos diferentes valores e características que este elemento possui. Por exemplo, um atributo de um material pode apresentar duas propriedades tais como módulo de elasticidade e coeficiente de Poisson.

A classe *AttribManager* foi implementada de forma que a adição de um novo protótipo seja realizada de uma forma bem simples e rápida. A lista de protótipos de atributos dessa classe é obtida a partir de um arquivo no formato JSON. Toda vez que a biblioteca for executada, este arquivo será lido pela classe *AttribManager* para a obtenção da lista dos protótipos (*prototypes*) que ficará disponível para a criação de novos atributos.

Dessa maneira, para criar um protótipo basta adicionar um objeto no arquivo JSON seguindo o padrão adotado (ver Figura 5.26). Cada objeto do arquivo JSON é lido como um dicionário em Python. Dicionário em Python são conjuntos formados por *chave: valor*. Assim como em um dicionário convencional os termos (chaves) estão associados a significados (valores), em Python dicionários representam uma estrutura de dados que permite mapear os valores através de suas respectivas chaves.

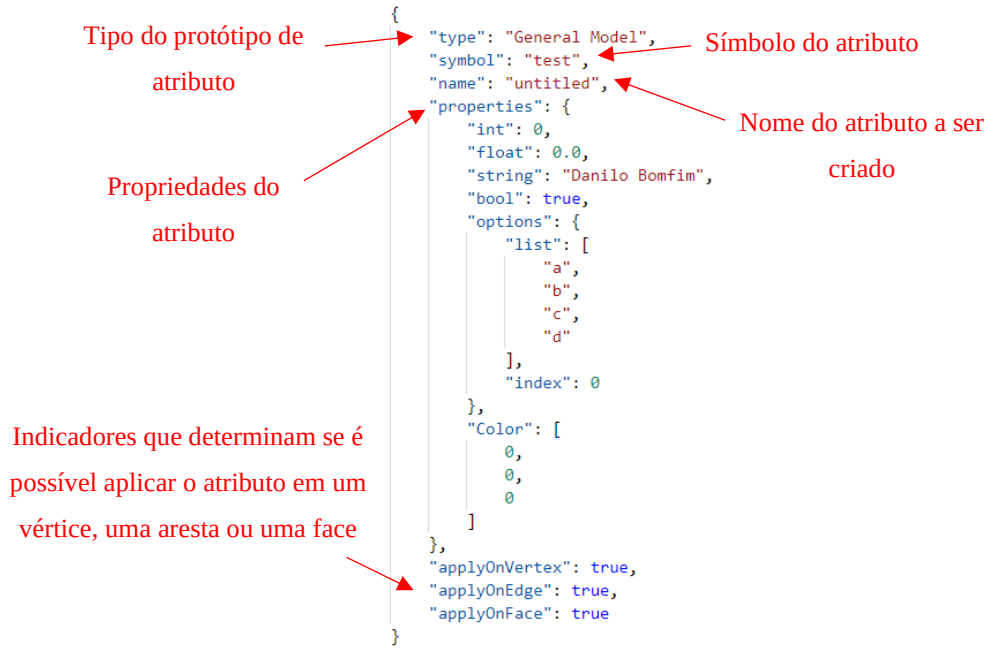


Figura 5.26 - Exemplo de protótipo de atributo no arquivo JSON

O arquivo JSON contém objetos que apresentam uma estrutura composta por *chave:valor*. A partir das chaves é possível filtrar os diferentes valores que cada objeto possui. Dessa forma, foi desenvolvido um padrão de chaves para os protótipos de atributos e a partir deste padrão é possivel configurar novos atributos de maneira bem simples. A chave "*type*" apresentada na Figura 5.26, determina o tipo do protótipo de atributo. Esse tipo pode ser uma carga nodal, carga uniforme, condição de suporte etc.

A chave "*symbol*" indica o símbolo atribuído ao protótipo de atributo. A renderização dos símbolos é realizada a partir de uma classe denominada de *AttribSymbols*. Essa classe apresenta métodos que permitem a configuração dos símbolos associados a cada protótipo de atributo. Esse modulo oferece um conjunto de métodos que são parametrizados pelas entidades geométricas. Esses métodos possibilitam criar e configurar formas básicas tais como quadrados, círculos, arco de círculos, triângulos e setas.

A chave "*name*" apresenta o nome do atributo criado a partir do seu respectivo protótipo. Cada atributo criado apresenta nomes distintos e únicos. O nome é utilizado para localizar e diferenciar cada atributo criado pelo usuário.

Como pode ser visto na chave "*properties*" da Figura 5.26, o padrão adotado no arquivo JSON permite criar atributos com diferentes tipos de valores como por exemplo um número inteiro, um número real, uma palavra e um indicador de verdadeiro ou falso. Além disso, é possível criar uma propriedade que apresente uma lista de valores que limite a opção do usuário final do aplicativo, ou seja, é possível criar uma caixa de opções para valores nos quais aquela propriedade do atributo pode assumir (chave "*options*" presente nas propriedades do atributo da Figura 5.26). Com esse padrão desenvolvido também é possível criar atributos com cores especificas (a coloração é configurável seguindo o padrão RGB).

Por fim, as chaves "*applyOnVertex*", "*applyOnEdge*" e "*applyOnFace*" determinam se é possível aplicar o atributo em um vértice, aresta e face respectivamente. A seguir serão detalhados os principais métodos presentes na classe *AttribManager* que utilizam do padrão previamente explicado.

O método *getPrototypes* retorna uma lista de dicionários que representa todos os protótipos de atributos que o usuário final pode criar. De forma análoga, a função *getAttributes* retorna uma lista de dicionários que representa todos os atributos criados pelo usuário final do aplicativo.

O método *getAttributeByName* é utilizado para obter um atributo específico da lista de atributos criados pelo usuário. Essa função recebe o nome deste atributo e retorna o seu respectivo dicionário. Já a função *getPrototypeByType* retorna o dicionário de um protótipo de um atributo a partir do seu tipo.

A criação de atributos ocorre por meio do método *createAttributeFromPrototype* que recebe o dicionário do protótipo do atributo e o nome do atributo a ser criado. Essa função verifica, na lista *attributes*, se já existe um atributo com o mesmo nome recebido pelo método. Caso não exista, um novo atributo é adicionado na lista de *attributes* e o método encerra retornando *True.* Caso contrário, então o método retorna *False* indicando que não foi possível criar um atributo com este nome.

A função *removeAttribute* é utilizado caso o usuário deseje remover um atributo criado. Esse método recebe o dicionário do atributo a ser removido e exclui o mesmo da lista *attributes*. Para modificar os valores de um atributo é utilizado o método *setAttributesValues* (essa função recebe o nome do atributo e uma lista

contendo todos os valores presentes nas propriedades deste) que altera os valores associados a este atributo.

5.9
Classe *HeFile*

A classe *HeFile* é responsável por salvar, ler e exportar as informações do modelo. Os componentes presentes nesta classe estão apresentados na Figura 5.27. Para salvar o modelo utiliza-se o método *saveFile* que recebe um sólido (*_shell*) no qual contém todas as informações topológicas e geométricas do modelo. Além disso, essa função também recebe uma lista que contêm todos os atributos criados pelo usuário (*_attributes*) e o nome do arquivo a ser salvo (*_filename*).

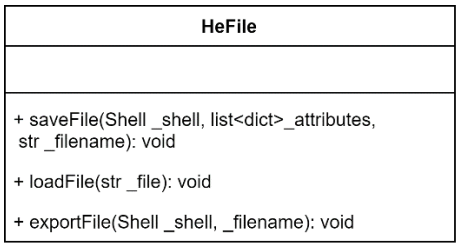


Figura 5.27 - Classe *HeFile*

O modelo é salvo em um arquivo no formato JSON que apresenta uma estrutura constituída por *chave: valor* que é equivalente a estrutura do dicionário em Python. Nesse dicionário existe uma chave correspondente a cada tipo de entidade topológica não abstrata (vértices, arestas e faces) e uma chave para todos os atributos criados pelo usuário. Os valores destas chaves são constituídos por listas de dicionários que contém as informações de cada elemento.

Seguindo o padrão dos dicionários em Python, a estrutura de dados é salva mantendo todas as relações existentes entre as entidades presentes no modelo. Essas relações topológicas do modelo são mantidas por meio da identificação numérica

de cada entidade (ID). Além disso, as informações geométricas são salvas em listas contendo as coordenadas dos pontos de cada vértice e aresta presente no sólido.

Para ler um modelo é utilizado o método *loadFile* que recebe o nome do arquivo a ser lido e processa todas as informações deste, recriando todas as entidades topológicas e as relações entre elas. O método *exportFile* é utilizado para converter as informações do modelo para outros formatos permitindo assim a comunicação com outros aplicativos.

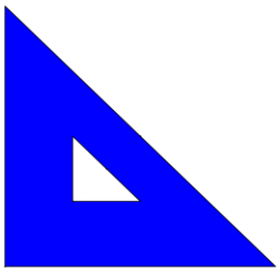
6
Utilização da biblioteca HETOOL

Neste capítulo é discutido e exemplificado como utilizar a biblioteca HETOOL desenvolvida e disponibilizada para utilização publica no repositório: https://gitlab.com/danilosb/hetoollibrary. Este repositório apresenta exemplos de utilização da biblioteca e uma documentação sobre os principais métodos presentes neste pacote. Além disso, são apresentados neste capítulo dois aplicativos desenvolvidos a partir da biblioteca HETOOL.

6.1
Criação do modelo geométrico

A Figura 6.1.a apresenta o trecho do código utilizado para a criação do modelo geométrico demonstrado na Figura 6.1.b. Para a construção do modelo geométrico foi utilizado a função *insertSegment*, detalhada na Seção 5.7.2, passando-se listas de coordenadas geométricas que constituem cada segmento a ser inserido. Em seguida utilizou-se a função *selectPick* para selecionar a região central do modelo geométrico e removê-la através do método *delSelectedEntities*. O sólido da Figura 6.1.b foi gerado através das funções de plotagem disponibilizadas pela biblioteca *matplotlib*. O código completo deste exemplo está disponibilizado no repositório anteriormente indicado.

```
from hetool.include.hetool import Hetool
Hetool.insertSegment([0,0,4,0,0,4,0,0])
Hetool.insertSegment([1,1,2,1,1,2,1,1])
Hetool.selectPick(1.1,1.1,0.01)
Hetool.delSelectedEntities()
```



(a) (b)

Figura 6.1 - Exemplos simples de utilização da biblioteca HETOOL

Como pode ser visto na Figura 6.1.a, a biblioteca possibilita de uma forma bem simples a modelagem de sólidos bidimensionais, sem a necessidade de o usuário ter o conhecimento dos conceitos topológicos envolvidos na implementação dessa estrutura de dados. A maioria das funções disponibilizadas pela biblioteca HETOOL lidam apenas com parâmetros relacionados com a geometria do modelo a ser construído.

A Figura 6.2 apresenta outro exemplo, disponibilizado no repositório, de um disco com fendas. O contorno do disco e de suas fendas foram discretizadas pela subdivisão dos segmentos que compõem o contorno. As funções utilizadas para a construção deste modelo geométrico são as mesmas utilizadas no exemplo anterior, modificando apenas as coordenadas geométricas passadas para os métodos utilizados.

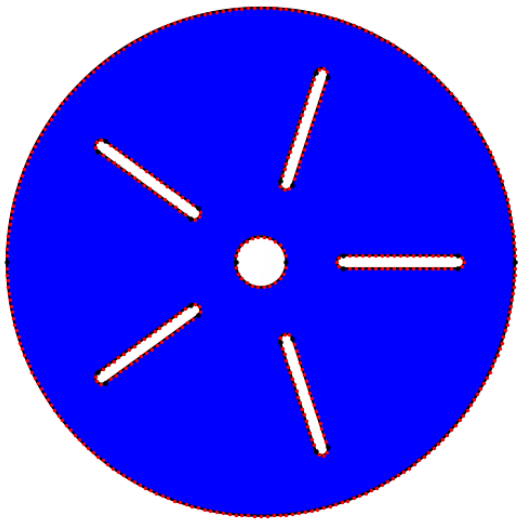


Figura 6.2 - Disco com fendas

6.2
Gerenciamento dos atributos de modelagem

Um dos principais diferenciais da biblioteca HETOOL está no gerenciamento de atributos de modelagem. Como discutido anteriormente, o pacote

desenvolvido disponibiliza funcionalidades que permitem a criação e configuração de novos protótipos de atributos a partir de um arquivo JSON.

As funções disponibilizadas pelo pacote para o gerenciamento de atributos são genéricas, possibilitando que qualquer novo protótipo de atributo seja de igual modo gerenciado a partir destes métodos. Isso é um diferencial relevante, pois permite a aplicação desta biblioteca em áreas distintas do meio científico.

Nesta seção são explicados com mais detalhes o processo para a criação de um protótipo de atributo e para a configuração dos valores do atributo a partir dos métodos presentes na biblioteca HETOOL.

6.2.1
Criação dos protótipos de atributos

Para a criação de um novo protótipo de atributo é necessário acessar o arquivo JSON que contém a lista de todos os protótipos de atributos. O diretório e o nome deste arquivo estão apresentados na Figura 6.3.

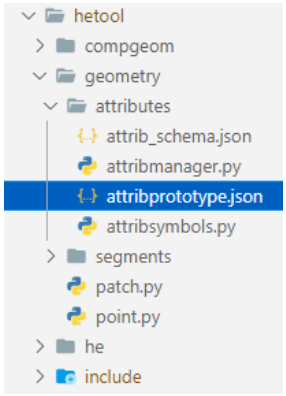


Figura 6.3 - Diretório do arquivo *attribprototype.json*

O arquivo *attribprototype.json* apresenta uma chave *"prototypes"* que fornece o acesso a uma lista de objetos de todos os possíveis atributos que podem ser criados pelo usuário. Para adicionar um novo protótipo deve-se seguir o padrão

utilizado (explicado na Seção 5.8) e adicionar um novo objeto a lista. A Figura 6.4 apresenta exemplos de protótipos de atributos criados a partir do padrão adotado apresentado na Seção 5.8.

Figura 6.4 - Exemplos de protótipos de atributos

A Figura 6.4.a apresenta um protótipo de atributo que determina as propriedades elásticas do material contendo o modulo de elasticidade e coeficiente de Poisson. A chave "*applyOnFace"* deste protótipo determina que todo atributo deste tipo pode ser aplicado nas faces do sólido modelado. De forma análoga, a Figura 6.4.d apresenta um protótipo que determina as condições de suporte de translação da direção X e Y e de rotação na direção Z. As chaves "*applyOnVertex"* e "*applyOnEdge"* determinam que todo atributo deste tipo pode ser aplicado tanto nos vértices como nas arestas do sólido modelado.

A Figura 6.4.b apresenta um protótipo de atributo que determina o valor da carga pontual que pode ser aplicada nos vértices do modelo construído e a Figura 6.4.c apresenta um protótipo de atributo que determina o valor da carga uniforme que pode ser aplicada nas aresta do solido modelado.

Todos os exemplos apresentam uma propriedade do tipo *"color".* Contudo, essa propriedade não é obrigatória. O usuário tem a liberdade de criar um protótipo de atributo com ou sem a cor associada diretamente ao atributo.

A inserção desses objetos, apresentados na Figura 6.4, à lista contida na chave *"prototypes"* do arquivo JSON habilita a criação e configuração desses novos tipos de atributos pelas funções presentes na biblioteca HETOOL.

6.2.2
Criação e configuração dos atributos

Nesta seção é discutido como criar um atributo a partir de um protótipo de atributo e como configurá-lo utilizando as funções disponibilizadas pela biblioteca. Ao utilizar a biblioteca HETOOL, todos os protótipos de atributos contidos na chave *"prototypes*" do arquivo *attribprototype.json* serão lidos e estarão disponíveis como modelos para a criação de atributos. A Figura 6.5 exemplifica como criar um atributo a partir de seu protótipo.

```
Hetool.addAttribute("Material", "M1")
Hetool.addAttribute("Material", "M2")
Hetool.addAttribute("Support Conditions","S1")
Hetool.addAttribute("Concentrated Load","CL1")
Hetool.addAttribute("Uniform Load", "UL1")
```

Figura 6.5 - Criação de atributos a partir dos protótipos de atributos

O método *addAttribute* é utilizado para criar um atributo a partir de um protótipo. Essa função recebe o tipo do protótipo de atributo e o nome para o novo atributo a ser criado, cada atributo criado apresenta um nome único. Após a criação destes atributos, utiliza-se o método *getAttributeByName* para obter o dicionário de cada um dos atributos criados conforme mostra a Figura 6.6.

```
material_1 = Hetool.getAttributeByName("M1")
material_2 = Hetool.getAttributeByName("M2")
support = Hetool.getAttributeByName("S1")
concetratedLoad = Hetool.getAttributeByName("CL1")
uniformLoad = Hetool.getAttributeByName("UL1")
```

Figura 6.6 - Obtenção dos atributos criados

Como já detalhado na Seção 5.8, cada atributo é representado por um dicionário. Logo para realizar modificações nas propriedades do atributo, deve-se acessar as chaves correspondentes aos valores que o usuário deseja modificar. A Figura 6.7 exemplifica a configuração de alguns valores dos atributos anteriormente criados.

A Figura 6.7.a apresenta as configurações das propriedades dos dois materiais criados e a Figura 6.7.b demonstra a configuração das condições de suporte que foram ajustadas para serem fixas nas três direções. A Figura 6.7.c e a Figura 6.7.d apresentam, respectivamente, os valores do carregamento nodal e uniforme. E a Figura 6.7.e apresenta a cor conferida a cada atributo criado seguindo o padrão RGB.

```
material_1['properties']['YoungsModulus'] = 100000
material_1['properties']['PoisonsRatio'] = 0.3
material_2['properties']['YoungsModulus'] = 500000
material_2['properties']['PoisonsRatio'] = 0.25
```

(a)

```
support['properties']['Dx'] = True
support['properties']['Dy'] = True
support['properties']['Rz'] = True
```

(b)

```
concetratedLoad['properties']['Fy'] = -10.0
concetratedLoad['properties']['Mz'] = -2.0
```

(c)

```
uniformLoad['properties']['Qy'] = -3.0
```

(d)

```
material_1['properties']['Color'] = [0.5, 0.5, 0.5]  # cinza
material_2['properties']['Color'] = [0.5, 0.75, 0.3]  # verde escuro
support['properties']['Color'] = [0.0, 0.0, 0.0]  # preto
concetratedLoad['properties']['Color'] = [1.0, 0.0, 0.0]  # vermelho
uniformLoad['properties']['Color'] = [0.0, 1.0, 0.0]  # verde
```

(e)

Figura 6.7 - Configuração dos atributos criados

Após a criação e configuração dos atributos é necessário indicar as entidades geométricas que receberão os atributos criados. Para isso, deve-se selecionar as entidades desejadas e utilizar o método *setAttribute* indicando o nome do atributo a ser associado as entidades geométricas selecionadas. A Figura 6.8.a demonstra o trecho do código utilizado para criar o modelo geométrico e a Figura 6.8.b demonstra como selecionar e associar os atributos as entidades geométricas selecionadas do modelo.

```
Hetool.insertSegment([0, 0, 4, 0])
Hetool.insertSegment([4, 0, 4, 0.5])
Hetool.insertSegment([4, 0.5, 0, 0.5])
Hetool.insertSegment([0, 0.5, 0, 0])
Hetool.insertSegment([2, 0, 2, 0.5])
```

(a)

```
Hetool.selectPick(1.0, 0.25, 0.01)
Hetool.setAttribute("M1")
Hetool.selectPick(3, 0.25, 0.01)
Hetool.setAttribute("M2")
```

(b)

Figura 6.8 - Criação do modelo geométrico e alocação dos atributos

Seguindo o mesmo padrão demonstrado na Figura 6.8.b, é possível selecionar outras entidades geométricas e aplicar os atributos desejados. A Figura 6.9 apresenta o modelo geométrico criado com todos os atributos aplicados a este modelo. O código completo deste exemplo está disponível no repositório anteriormente indicado. Todos os exemplos deste repositório apresentam a função *printHEModel* que gera resultados visuais aos modelos e atributos criados através do pacote *matplotlib* do Python. Os símbolos dos atributos são criados através da classe *AttribSymbols* que apresenta funções parametrizadas pelas coordenadas geométricas de cada entidade. Os exemplos no repositório indicam como utilizar de maneira adequada essas funções.

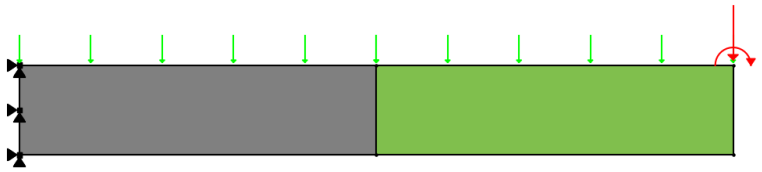


Figura 6.9 - Modelo geométrico simples com os atributos

6.3
Modelador de Sólidos FEMEP

Nesta seção é descrito o desenvolvimento de um modelador de sólidos bidimensionais, denominado *Finite Element Method Educational Computer Program* (FEMEP), que apresenta geração de malhas de elementos finitos por região planar, podendo ser estendido e aplicado na simulação de vários problemas da mecânica computacional. O objetivo da seção é demonstrar a potencialidade de uso da biblioteca HETOOL proposta neste trabalho para criação de aplicativos que trabalhem com modelos de subdivisões planares e que definam atributos de modelagem e simulação próprios.

6.3.1
Características gerais

Para o desenvolvimento da interface gráfica do modelador de sólidos, foi utilizada a ferramenta Qt Designer. O Qt Designer é um ambiente de desenvolvimento integrado multi-plataforma. Esse ambiente reúne características e ferramentas de apoio ao desenvolvimento de aplicativos de maneira fácil e rápida. Assim, o Qt Designer facilitou e otimizou a criação dos botões e janelas presentes na interface gráfica do aplicativo. Para a representação e visualização dos modelos foi utilizada a biblioteca gráfica OpenGL (Open Graphics Library). Essa biblioteca é gratuita e amplamente utilizada no desenvolvimento de aplicativos.

A classe *Canvas* foi desenvolvida a partir da biblioteca OpenGL. Esta classe é responsável por gerenciar os eventos de mouse da janela de modelagem e renderizar o sólido modelado na tela. A tela possui uma câmera que contém os parâmetros de visualização necessários para projetar a cena. Configurações como cores, tipos de linha, modo de renderização e informações de iluminação são configuráveis por meio de métodos e atributos pertences a classe *Canvas*. Essa classe apresenta métodos que permitem a manipulação dos limites da janela de visualização, bem como a manipulação dos eventos de mouse ocorridos dentro do *Canvas*.

No diagrama da Figura 6.10, apresentam-se os principais módulos da biblioteca HETOOL utilizados durante uma interação do usuário com o aplicativo desenvolvido.

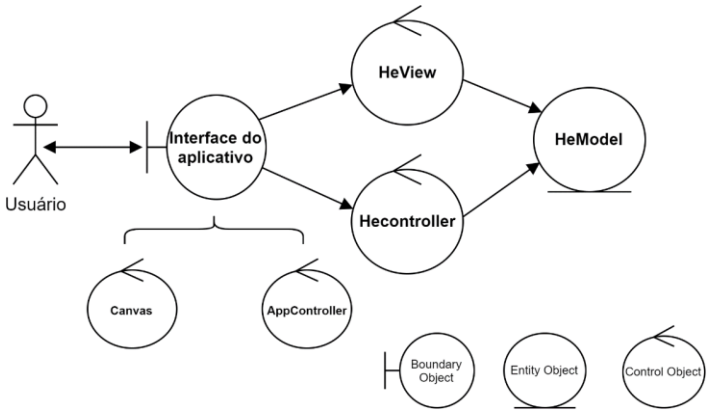


Figura 6.10 - Diagrama resumido da interação do usuário com o modelador de sólidos

Qualquer interação do usuário com os botões e ferramentas da interface passa pela classe *AppController.* Essa classe foi desenvolvida com intuito de realizar um controle eficiente de todas essas interações. O *AppController* contém todos os atributos e métodos que conectam as interações do usuário com os outros módulos presentes no aplicativo como o *Canvas* e o *HeController*. O *AppController* associa todos os botões da interface com as suas respectivas funções.

Como explicado no Capítulo 5, a classe *HeController* apresenta a função de controlar toda a parte de processamento da biblioteca HETOOL. A classe *HeModel* cumpre o papel de modelo salvando todas as informações geométricas e topológicas a respeito do sólido. E a classe *HeView* se comunica diretamente com a classe *HeModel* onde são obtidos os dados necessários para a renderização do modelo e transmitidos para o *Canvas* exibir a cena do sólido na tela.

Como pode ser visto na Figura 6.10, o usuário recebe a resposta para a sua interação por meio de duas formas. A resposta visual pode ser recebida através do *AppController* que fornece informações com relação as ferramentas da interface. Outra forma de resposta visual é através da janela de modelagem, onde as interações dos usuários com a tela são exibidas através de cenas.

6.3.2
Interface

A interface do modelador foi criada objetivando a simplicidade, eficiência e a facilidade de manipulação. Além disso, qualquer novo protótipo de atributo adicionado a biblioteca HETOOL estará disponível pela interface deste modelador sem a necessidade de configurações adicionais. A interface inicial é composta por uma janela de modelagem (canvas) onde será exibido o sólido modelado e por 3 barras de ferramentas, nomeadas por barra principal, secundária e de modelagem, conforme pode ser visto na Figura 6.11.

Além disso, existe um quadro dinâmico, localizado no lado direito do canvas, que exibe opções e configurações de acordo com o botão pressionado. O

aplicativo pode apresentar múltiplas janelas de modelagem possibilitando o desenvolvimento de vários sólidos simultaneamente.

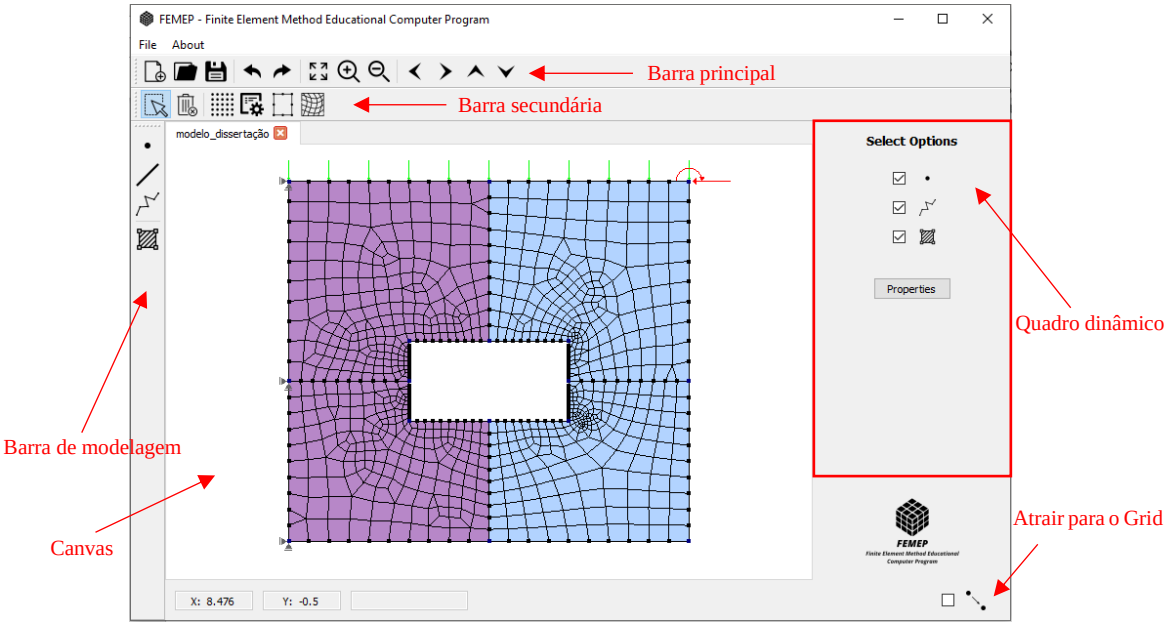


Figura 6.11 - Interface do FEMEP

A barra de ferramentas principal é composta por botões que permitem adicionar um novo canvas, salvar e abrir um modelo, desfazer e refazer comandos. Além disso, está barra apresenta botões que manipulam os limites de visualização da janela do canvas tais como transladar, ampliar, reduzir e ajustar os limites de visualização ao tamanho do modelo.

A barra de ferramentas secundária é composta por botões que possibilitam a seleção de entidades (pontos, segmentos e regiões) e a remoção destas entidades. Além disso, a barra secundaria apresenta o botão *Grid* que tem a função de adicionar uma malha de pontos ao canvas para auxiliar na modelagem. No canto direito inferior da Figura 6.11 existe uma caixa de seleção que apresenta a função de ativar/desativar a atração dos pontos inseridos pelo mouse para a malha de pontos criada pelo botão *Grid*.

A barra de ferramentas secundária também apresenta um botão que habilita a interface do gerenciador de atributos (posicionada no quadro dinâmico). A partir

desse gerenciador é possível criar e deletar atributos bem como associar esses atributos a entidades geométricas presentes no sólido. Conforme pode ser visto na Figura 6.11, esse gerenciador permite adicionar diferentes tipos de materiais, condições de suporte, carregamentos e outros tipos de atributos. A Figura 6.12 apresenta a interface do gerenciador de atributos.

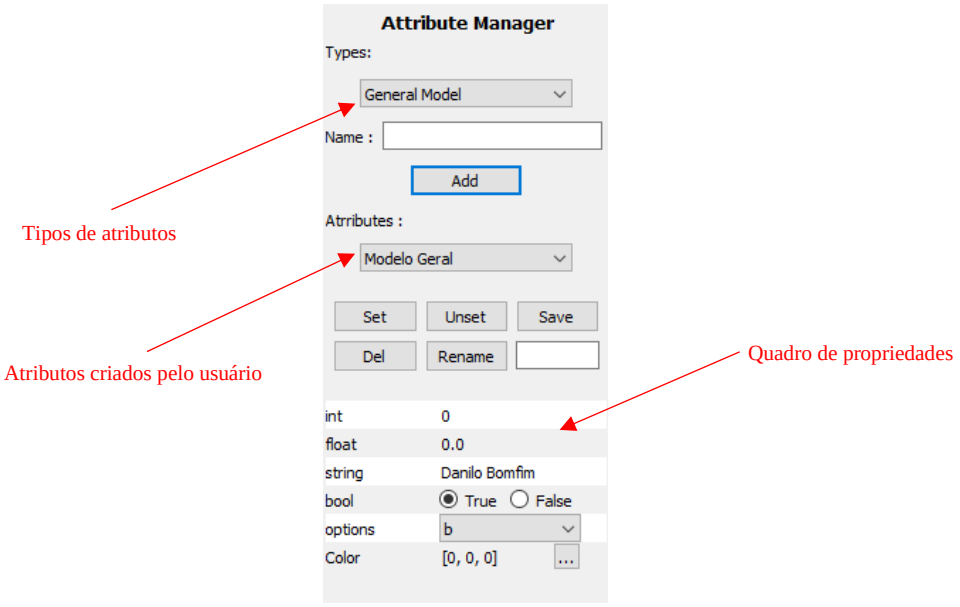


Figura 6.12 - Interface do gerenciador de atributos

Para criar um atributo é necessário selecionar o tipo de atributo desejado e nomeá-lo. Ao apertar o botão *Add*, o atributo será adicionado à lista de atributos criados. Para configurar as propriedades especificas deste atributo é preciso selecioná-lo na lista de atributos. Em seguida, basta alterar os valores deste atributo e salvar. Para associar o atributo a uma entidade geométrica é necessário selecionar a entidade desejada e pressionar o botão *Set*. O botão *Unset* remove o atributo das entidades geométrica selecionadas. O botão *Del* remove um atributo selecionado da lista de atributos e o botão *Rename* renomeia este atributo.

O atributo criado na Figura 6.12 é um atributo genérico com intuito de demonstrar todas as configurações possíveis. No quadro de propriedades apresenta-se todas os tipos de dados que um atributo pode possuir tais como número inteiro, número real, palavra, indicador de verdadeiro ou falso, caixa de opções que limita a escolha do usuário e cor.

A barra de ferramenta secundária também apresenta um botão que habilita a interface de definição do número de subdivisões de um segmento. Essa interface está apresentada na Figura 6.13. Ao número de subdivisões pode ser atribuído uma razão (*Ratio*) que gera uma proporção entre os subsegmentos de forma que o primeiro subsegmento seja *x* vezes maior que o último de acordo com o número da razão.

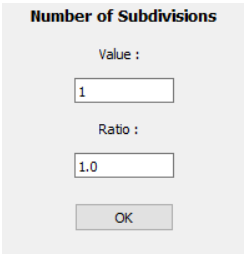
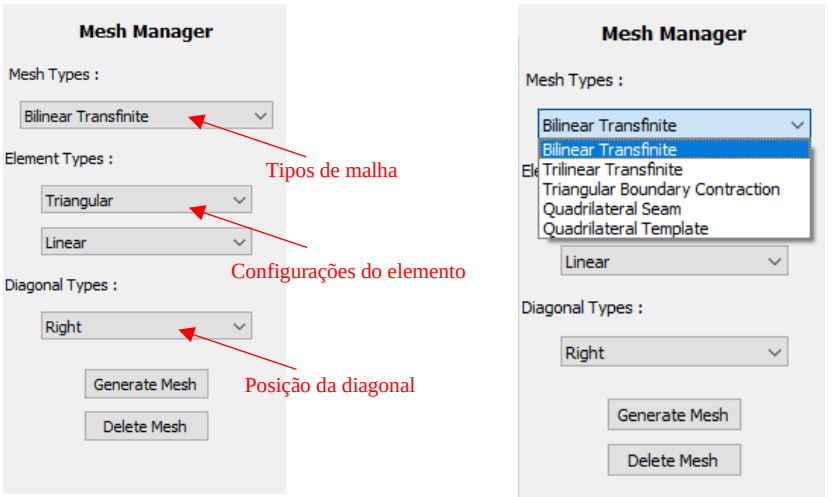


Figura 6.13 - Interface do número de subdivisões

Outro botão presente na barra de ferramenta secundaria é o botão que habilita a interface do gerenciador de malhas apresentado na Figura 6.14. Esse gerenciador permite adicionar e remover diferentes tipos de malhas triangulares e quadrilaterais, estruturadas e não estruturadas. A Figura 6.14.b apresenta a lista de todos os tipos de malhas que podem ser criadas.

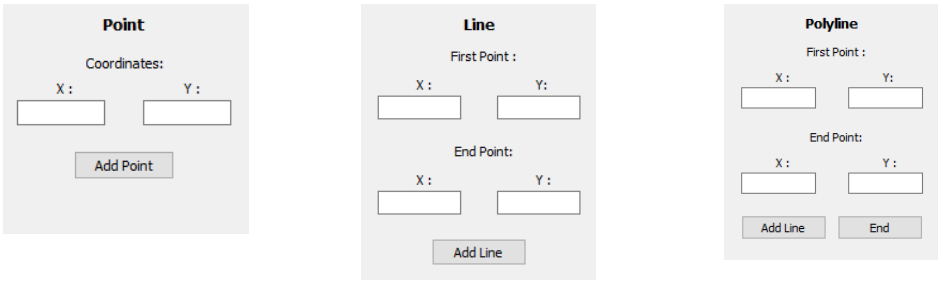


(a) (b)

Figura 6.14 - Interface do gerenciador de malhas

A partir da interface do gerenciador de malhas é possível configurar o tipo de elemento como triangular ou quadrilateral e linear ou quadrático. Para elementos triangulares gerados pelo método de geração de malha transfinito bilinear também é possível configurar a posição da diagonal (esquerda, direita, cruzado e otimizado). Para criar uma malha é necessário selecionar uma face e apertar o botão *Generate Mesh.* Para remover uma malha basta selecionar a face que contêm esta malha e pressionar o botão *Delete Mesh.*

A barra de modelagem apresenta quatro botões que desempenham o papel de criar pontos, linhas, polilinhas e regiões. A inserção de pontos é realizada pressionando o botão do *mouse* na posição desejada. As coordenadas dos pontos criados aparecem no quadro dinâmico. Essas coordenadas podem ser editadas ou mesmo inseridas explicitamente usando o teclado. Para a inserção de coordenadas em modo teclado, as três primeiras funções de modelagem apresentam suas respectivas interfaces (ver Figura 6.15) que são posicionadas no quadro dinâmico (indicado na Figura 6.11). Para inserção de pontos de uma linha poligonal, o usuário entra com as coordenadas do primeiro ponto (*First Point*) e entra com as coordenadas de cada ponto subsequente (possivelmente o último ponto - *End Point*), terminando a inserção de pontos com a opção *End*.

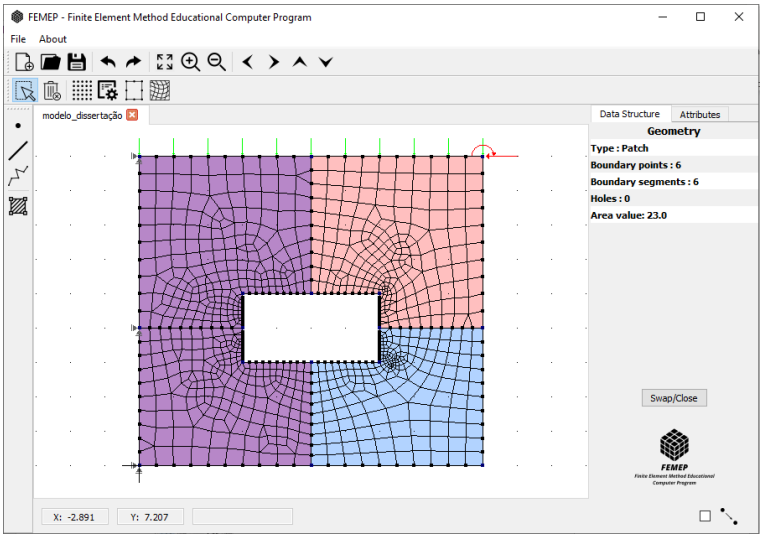


(a) (b) (c)

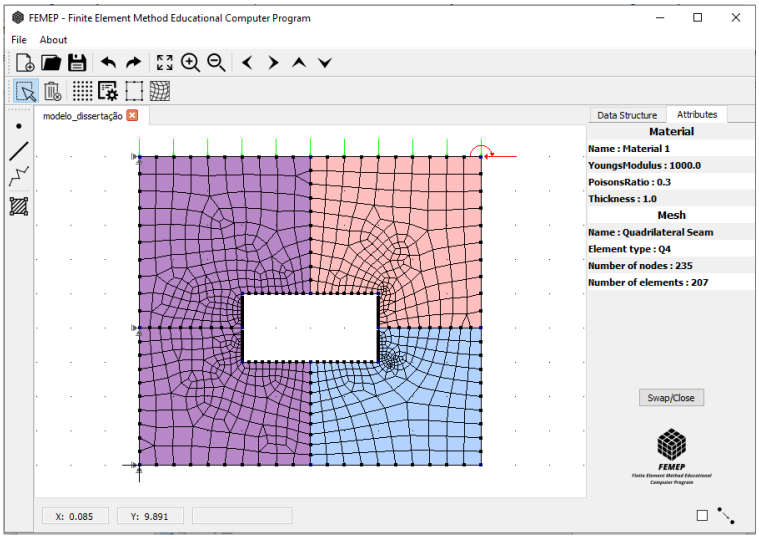
Figura 6.15 - Exibidores dinâmicos das funções de modelagem

A seleção das entidades geométricas modeladas pode ser configurada pela interface de seleção (*Select Options*) demostrada no quadro dinâmico indicado na Figura 6.11. A interface de seleção apresenta três caixas de opção que quando preenchidas permitem selecionar as três entidades geométricas (ponto, segmento ou superfície). Ao desmarcar uma destas caixas, não será possível selecionar o

elemento geométrico associado a caixa de seleção desmarcada. Além disso, esse quadro dinâmico mostrado na Figura 6.11 apresenta o botão *Properties* que demonstra as propriedades e atributos de uma entidade geométrica selecionada, conforme pode ser visto na Figura 6.16 que apresenta as informações sobre uma região planar selecionada.



(a)



(b)

Figura 6.16 - Propriedades da superfície selecionada

A partir do botão *Properties* é possível obter as informações geométricas do elemento selecionado tais como coordenadas dos pontos, comprimento dos segmentos, valor da área de uma região planar, número de pontos e número

segmentos do contorno de uma região, número de buracos presentes em uma região etc. Este mesmo botão também permite obter a descrição de todos os atributos associados com a entidade geométrica selecionada conforme pode ser visto na Figura 6.16.b.

6.3.3
Modelagem de subdivisões planares

Nesta seção é discutido o funcionamento da seleção e remoção de entidades, geração de pontos, geração de segmentos e geração de regiões planares no modelador de sólidos desenvolvido. Para desenho e criação do modelo na interface gráfica, é criado um objeto (*Canvas*) de uma subclasse (*GLCanvas*) da classe *QGLWidget* do Qt, que encapsula o sistema gráfico OpenGL. A classe *QGLWidget* tem funções de *callback* que fazem a interface com o OpenGL e funções de *callback* para tratamento de eventos de *mouse* no canvas. As funções de *callback* devem ser escritas pela aplicação e implementam ações específicas de visualização do modelo e interação com o *mouse*.

Para a coleta inicial de pontos e segmentos foi desenvolvido uma classe chamada de *GeoCollector*. Essa classe possui métodos e atributos que visam a verificação inicial da possível inserção da entidade na estrutura de dados. O principal atributo desta classe é chamado de *geoType* que define o tipo de entidade que está sendo coletada. Dentro dessa classe, existem métodos responsáveis por realizar o controle da quantidade de pontos geométricos que definem cada uma das entidades que podem ser criadas tais como pontos, linhas e polilinhas. Por exemplo, a linha é definida por dois pontos geométricos e a polilinha não apresenta um limite definido para quantidade de pontos.

6.3.3.1
Criação de pontos

Para inserir um ponto no modelo o usuário deve incialmente clicar no botão *Point*. Esse botão gera uma chamada (*on_actionPoint*) no *AppController* que ajusta um atributo do *Canvas* e outro do *GeoCollector*. Esses atributos fazem parte do processo inicial para a coleta de pontos pelo aplicativo.

Para o ajuste do primeiro atributo é chamado a função do *Canvas* *setMouseAction* que modifica o atributo *curMouseAction* (esse atributo define qual ação será executada pelo mouse podendo ser selecionar, coletar ou indefinido). Para a criação de novas entidades geométricas o *curMouseAction* é ajustado para o modo de coleta. O outro atributo modificado é o *geoType*, pertencente a classe *GeoCollector*. Esse atributo é modificado para a coleta da entidade geométrica *Point* através do método *setGeoType*.

Quando o usuário realizar um clique dentro do canvas, será chamada uma função de *callback* da classe *QGLWidget* chamada de *mousePressEvent*. Essa função gerencia os eventos de mouse ocorridos na tela de modelagem. Dentro dessa função, são realizadas verificações relacionadas com o de tipo de ação atribuída ao clique do mouse por meio do atributo *curMouseAction*. Além disso, nessa mesma função é verificado se o ponto a ser adicionado pode ser aproximado para um ponto de um segmento próximo ou para a malha de pontos gerado pela função *Grid*.

Após essas verificações, o método *insertPoint*, pertencente a classe *HeController*, é chamado. Esse método insere o ponto coletado na estrutura de dados. Após o processamento da estrutura de dados, o ponto é adicionado em uma lista de pontos dentro do modelo (*HeModel*) e exibido através da função *PaintGL* do *Canvas* que recebe essa lista de pontos através da classe *HeView*. O fluxograma simplificado do processo realizado pelo modelador para criar um ponto está representado na Figura 6.17.

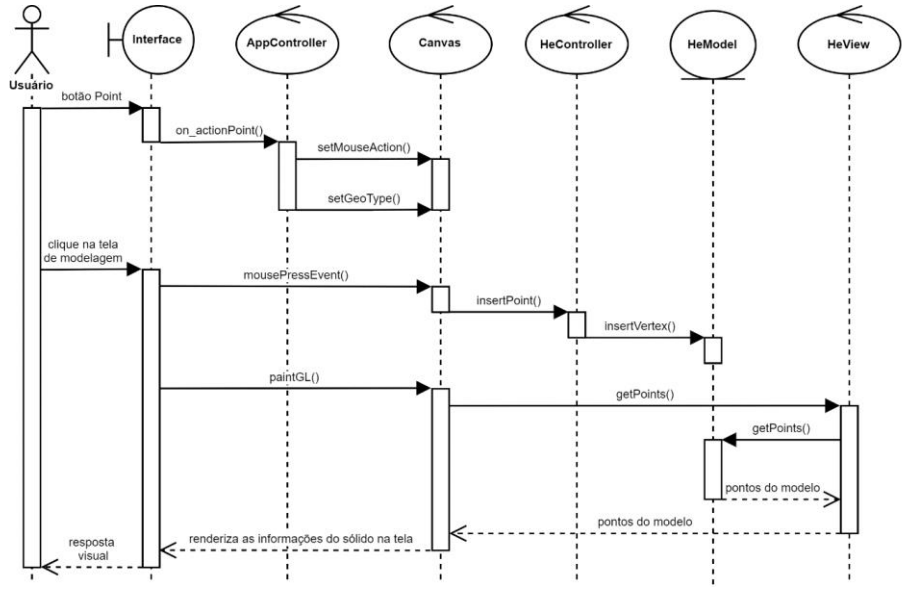


Figura 6.17 - Diagrama de sequência do processo para a criação de um ponto

6.3.3.2
Criação de segmentos

O processo para a geração de segmentos é análogo ao de geração dos pontos. Inicialmente considere que o usuário tenha interagido com o botão *Line* na barra de modelagem. Esse botão gera uma chamada (*on_actionLine*) no *AppController* que ajusta um atributo do *Canvas* e outro do *GeoCollector*. O primeiro atributo modificado no *Canvas*, por meio da função *setMouseAction,* é o atributo *curMouseAction* que é ajustado para a função de coleta. O outro atributo modificado por meio do método *setGeoType é* o atributo *geoType* (pertencente a classe *GeoCollector*) que é ajustado para o tipo *Line.*

Quando o usuário realizar um clique dentro do canvas, a função *mousePressEvent* será chamada. Dentro dessa função, são realizadas verificações relacionadas com o de tipo de ação atribuída ao clique do mouse por meio do atributo *curMouseAction* que já está ajustado para coletar as coordenadas geométricas da linha.

Além disso, nessa mesma função é verificado se os pontos de controle (pontos principais que definem o elemento) do segmento podem ser aproximados para as

coordenadas geométricas de um ponto pertencente a alguma entidade do modelo ou para a malha de pontos gerado pela função *Grid*.

Após o usuário adicionar os pontos de controle do segmento o método *insertSegment*, pertencente ao *HeController*, é chamado. Após o processamento da estrutura de dados, o segmento é adicionado em uma lista de segmentos do *HeModel* e exibido através da função *PaintGL* do *Canvas* que recebe essa lista de segmentos através do *HeView*. O fluxograma do processo realizado pelo modelador para criar um segmento está representado na Figura 6.18.

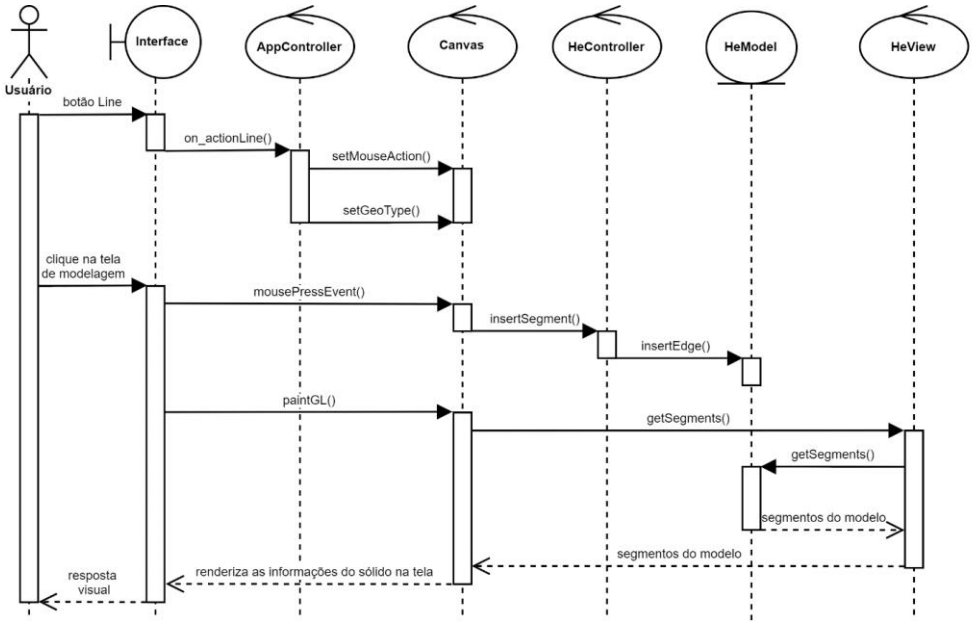


Figura 6.18 - Diagrama de sequência do processo para a criação de uma linha

6.3.3.3
Criação de regiões planares

A geração de regiões planares ocorre de maneira automática. A condição para que uma região seja gerada é que se tenha um conjunto de segmentos conexos que formem um caminho fechado. Após essa condição ser satisfeita, uma nova região planar (*Patch*) é adicionada ao modelo (*HeModel*).

Por fim, o *Canvas* recebe a lista de todas as regiões presentes no modelo através do *HeView* e produz a cena destas regiões na tela. Toda região criada é uma instancia da classe *Patch* que armazena informações sobre os segmentos, orientação dos segmentos e pontos do contorno externo dessa região. Além disso, são armazenadas as mesmas informações citadas anteriormente para os buracos de cada *Patch*.

Para gerar a cena das regiões na tela é feita uma triangulação dos pontos que formam o contorno externo do segmento através de um método já bem conhecido chamado de *Ear Clipping* (mais detalhes sobre esse método podem ser encontrados em [49]). Essa técnica subdivide toda a região planar em um conjunto de triângulos, utilizando apenas pontos no contorno da região (sem criar nenhum ponto interior). Com isso, utiliza-se uma função específica da biblioteca do OpenGL que renderiza os triângulos os quais unidos formam a região desejada. A Figura 6.19 abaixo ilustra este conceito.

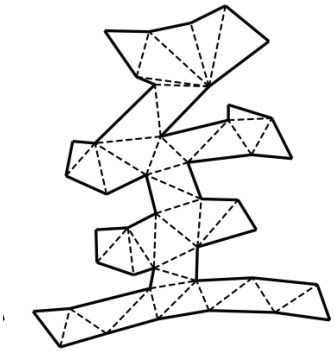


Figura 6.19 - Triangulação de uma região planar pelo algoritmo *Ear Clipping* [49]

6.3.3.4
Seleção e remoção de entidades geométricas

A seleção de entidades se dá através do atributo *isSelected* que é utilizado para verificar se uma entidade geométrica foi selecionada pelo usuário. Todas as classes dos elementos geométricos apresentam esse atributo.

Para remover uma entidade, antes é preciso selecioná-la. Para selecionar um elemento geométrico é necessário clicar no botão *Select* da interface. Esse botão chama a função *setMouseAction* que ajusta o atributo *curMouseAction* do *Canvas* para o modo de seleção.

A partir deste ponto, qualquer clique efetuado na tela de modelagem apresenta a capacidade de selecionar uma entidade geométrica (ponto, segmento ou região planar). Ao selecionar um elemento geométrico, o *Canvas* enviará uma resposta ao usuário através da modificação da cor do elemento selecionado.

A seleção de uma entidade ocorre desde que as coordenadas do clique do mouse estejam suficientemente próximas das coordenadas do elemento geométrico. Em seguida, ao pressionar o botão *Del* da interface as entidades geométricas selecionadas serão removidas pelo *HeController* que removera esses elementos do modelo.

6.3.4
Geração de malhas de elementos finitos

Para a geração das malhas de elementos finitos foi utilizada a biblioteca computacional *mesh 2D* desenvolvida pelo grupo de pesquisa no Instituto Tecgraf/PUC-Rio liderado pelo Prof. Luiz Fernando Martha [50]. Essa biblioteca desenvolvida em C/C++ apresenta um conjunto de métodos que permitem criar malhas de elementos finitos bidimensionais estruturadas e não estruturadas. Essas malhas podem apresentar elementos triangulares ou quadrilaterais e lineares ou quadráticos.

Para o uso dessa biblioteca juntamente com o a estrutura de dados desenvolvida em Python, foi utilizado um aplicativo SWIG (http://www.swig.org) para converter a biblioteca para um formato *pyd* (objeto de extensão) que se comunica com Python. Além disso, foi necessário criar uma camada intermediaria para fazer a comunicação entre as distintas linguagens de programação.

Os métodos presentes no pacote *mesh 2D*, em geral, recebem as coordenadas dos pontos da fronteira da face onde será criado a malha e as configurações

necessárias para a geração de um tipo específico de malha. Em seguida após o processamento das informações, esses métodos retornam as coordenadas de todos os pontos da malha e a relação de conectividade de todos os elementos presentes nesta malha.

A partir dessas informações obtidas pela biblioteca *mesh 2D*, foi utilizado a estrutura de dados desenvolvida para gerar um *HeModel* da malha. Esse novo modelo que apresenta as informações da malha é separado do modelo que guarda o sólido modelado. As figuras abaixo ilustram alguns dos diferentes tipos de malhas que foram geradas a partir do pacote de geração de malhas. Esses sólidos apresentados abaixo foram construídos apenas para demonstrar a capacidade do aplicativo desenvolvido.

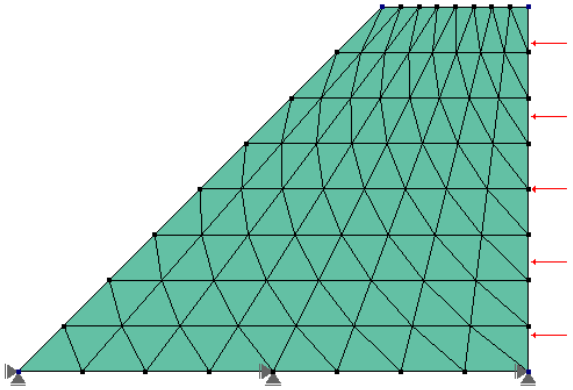


Figura 6.20 - Modelo 1: Malha triangular estruturada

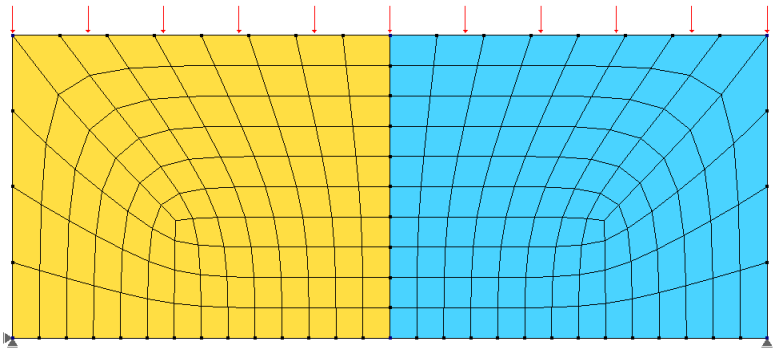


Figura 6.21 - Modelo 2: Malha quadrilateral estruturada

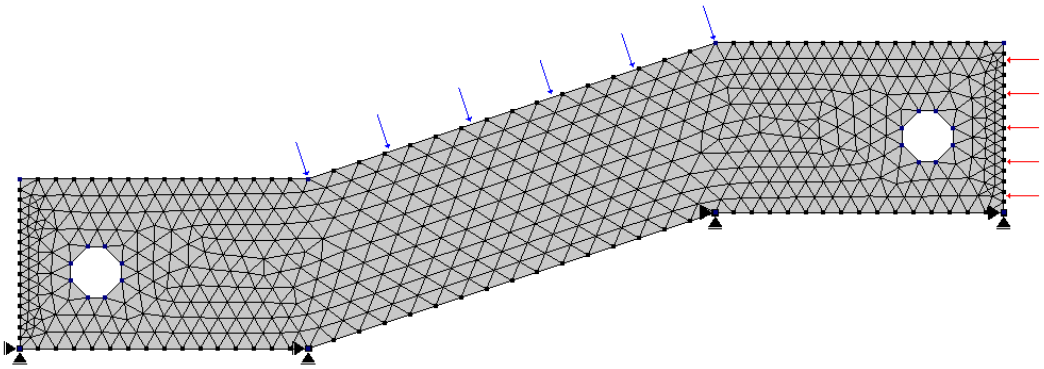


Figura 6.22 - Modelo 3: Malha triangular não estruturada

6.3.5
Exportação do modelo e análise dos resultados

Para a realização da análise dos modelos foi utilizado o programa FEMOOLAB (https://gitlab.com/rafaelrangel/femoolab). O FEMOOLAB é um aplicativo, escrito em MATLAB, de elementos finitos capaz de realizar análises de problemas mecânicos e térmicos. Por meio do modulo *HeFile* da biblioteca HETOOL (ver Seção 5.9), os modelos desenvolvidos no FEMEP podem ser convertidos e exportados para o programa FEMOOLAB, que utiliza um formato de arquivo neutro desenvolvido no Instituto Tecgraf/PUC-Rio (https://web.tecgraf.puc-rio.br/neutralfile).

Para exportar o modelo pelo aplicativo FEMEP deve-se clicar na opção *Export* do menu *File*. Essa ação habilitara um gerenciador de exportação localizado ao lado direito da tela de modelagem (conforme ilustra a Figura 6.23). Ao pressionar o botão *Export* do gerenciador de exportação, o *HeFile* verificará se é possível converter o modelo para o arquivo em formato neutro. Caso nenhum alerta seja emitido pelo aplicativo, o modelo já foi convertido para um arquivo neutro e pode, então, ser utilizado no programa FEMOOLAB para a obtenção dos resultados.

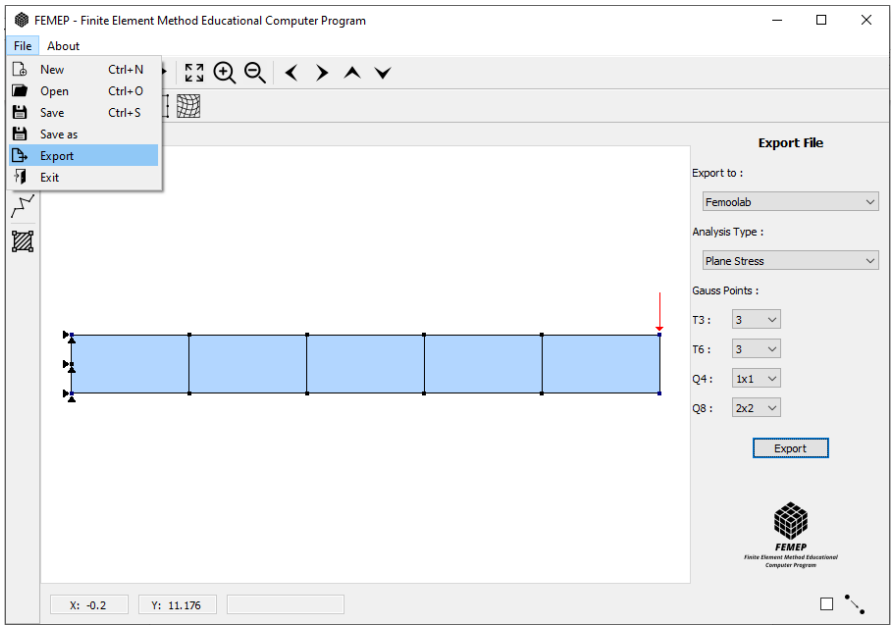
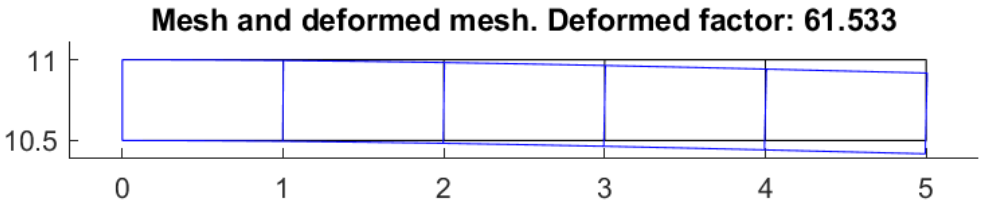


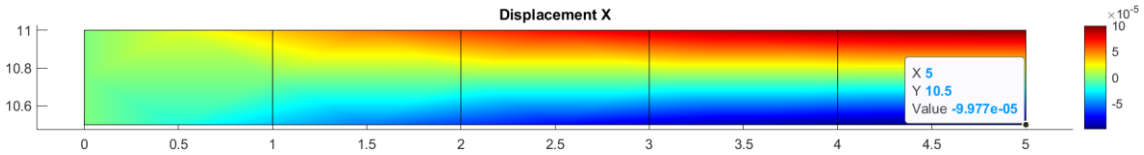
Figura 6.23 - Exemplo 1 de exportação de um modelo

O gerenciador de exportação permite escolher o tipo de análise. O tipo de análise pode ser *Plane Stress* para análises mecânicas ou *Plane Conduction* para análises térmicas. Além disso, é possível configurar o número de pontos de Gauss dos elementos da malha de elementos finitos a ser exportada.

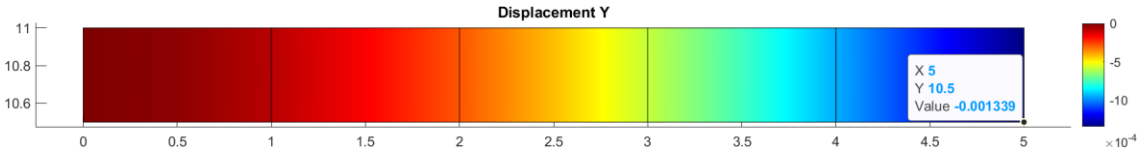
A Figura 6.24 demonstra alguns dos resultados que podem ser obtidos após a exportação do modelo da Figura 6.23 para o aplicativo FEMOOLAB. O modelo apresenta 5 elementos quadrilaterais e quadráticos (elemento Q8), comprimento de 5 metros, largura de 0.5 metros, espessura de 0.1 metros, módulo de elasticidade de 30 Gpa, coeficiente de Poisson de 0.3 e está submetido a uma carga nodal de 1 kN. A Figura 6.24.a apresenta a configuração deformada do modelo exportado, a Figura 6.24.b e a Figura 6.24.c apresentam os deslocamento obtidos na direção *X* e *Y* e a Figura 6.24.d e a Figura 6.24.e demonstram os resultados das tensões normais na direção *X* e *Y*. Os valores dos deslocamentos são dados em metros e das tensões em megapascal.



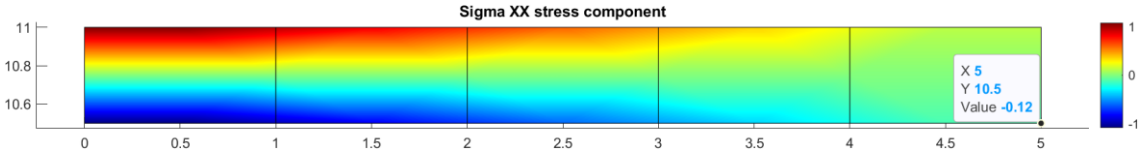
(a) Configuração deformada



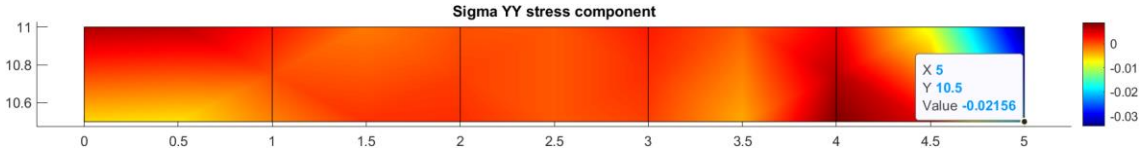
(b) Deslocamento na direção X



(c) Deslocamento na direção Y



(d) Tensão normal na direção X



(e) Tensão normal na direção Y

Figura 6.24 - Resultados obtidos pelo FEMOOLAB para o exemplo 1

Utilizando o mesmo modelo da Figura 6.23 e modificando as condições de contorno é possível analisar o comportamento térmico dessa barra quando submetido a temperatura de 300 kelvin nas extremidades e uma geração de calor interna de 100000 $W.m^{-3}$, conforme ilustra a Figura 6.25. O material deste modelo apresenta condutividade de 1000 $W.m^{-1}.K^{-1}$ e calor específico de 100 $J.kg^{-1}.K^{-1}$.

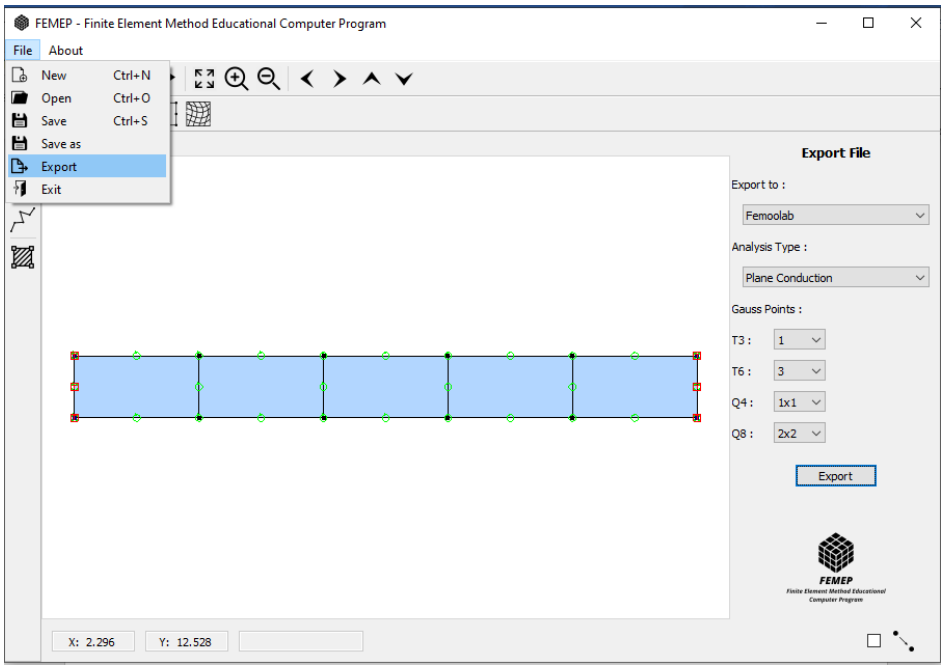
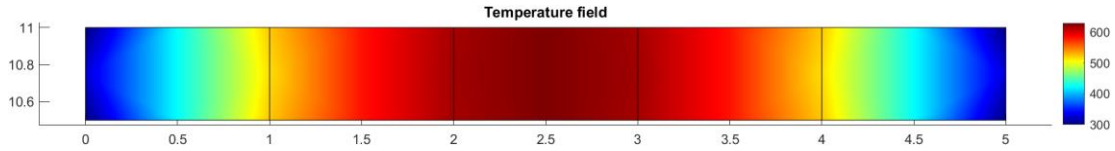
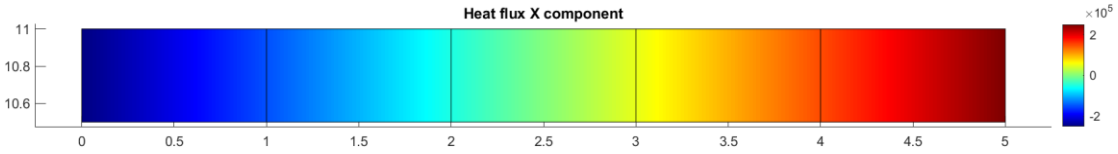


Figura 6.25 - Exemplo 2 de exportação de um modelo

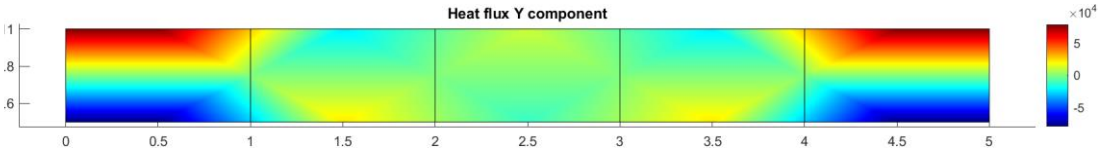
Após a exportação para o programa FEMOOLAB, obteve os resultados apresentados na Figura 6.26. Os resultados apresentados de temperatura estão em Kelvin e do fluxo de calor em $W.m^{-2}$.



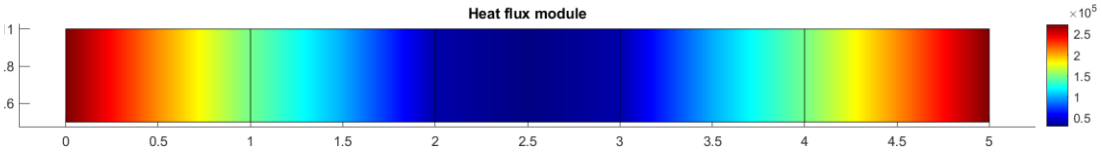
(a) Campo de Temperatura



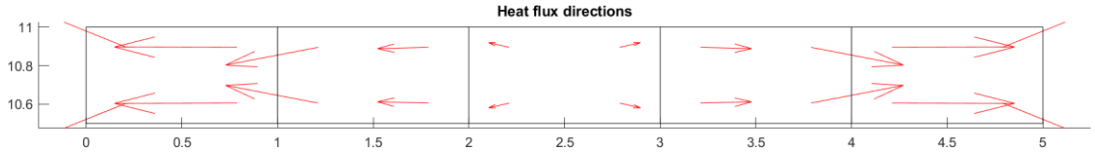
(b) Fluxo de calor na direção X



(c) Fluxo de calor na direção Y



(d) Módulo do fluxo de calor



(e) Direções do fluxo de calor

Figura 6.26 - Resultados obtidos pelo FEMOOLAB para o exemplo 2

6.4
Programa Hetool

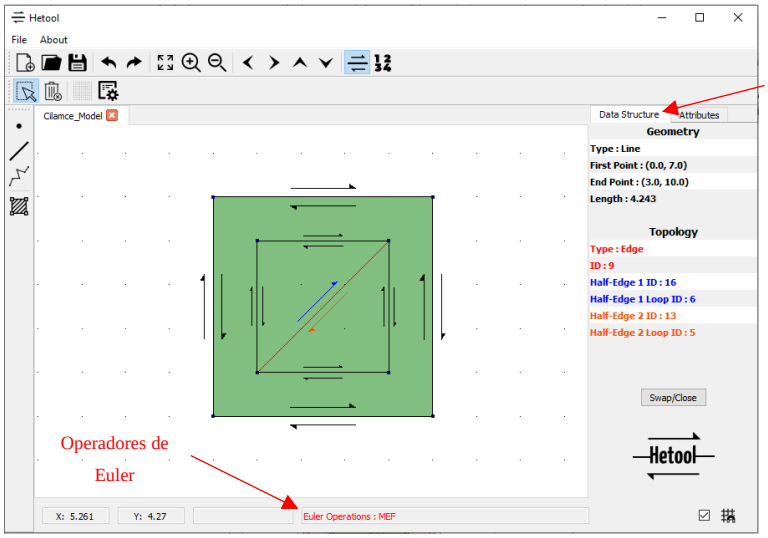
Nesta seção é demonstrado outra aplicação desenvolvida a partir da biblioteca HETOOL. Ao longo do desenvolvimento da biblioteca HETOOL e do aplicativo FEMEP também foi desenvolvido um programa demonstrativo da estrutura de dados Half-Edge focado no processo de ensino dos conceitos básicos que envolvem a implementação dessa estrutura. Esse programa foi publicado com nome de *Hetool* no *XLII Ibero-Latin Congress on Computanional Methods in*

Engineering (CILAMCE – 2021) and 3rd Pan American Congress on Computational Mechanics [51].

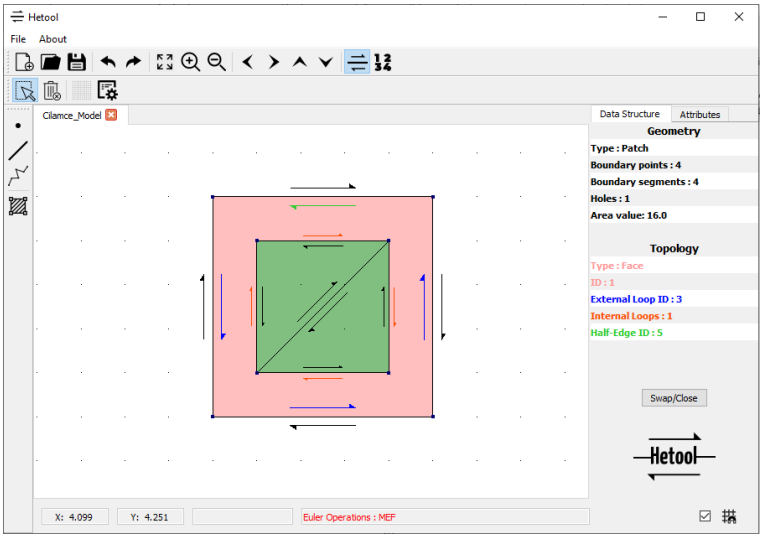
O aplicativo *Hetool* desenvolvido visa melhorar o processo de ensino-aprendizagem da estrutura de dados Half-Edge, tornando essa aprendizagem mais dinâmica, amigável e atrativa ao usuário. A interface do programa pode ser vista na Figura 6.27, onde apresenta-se todas as funcionalidades oferecidas pelo aplicativo, tais como gerenciar os limites de visualização da janela, criar e deletar entidades, salvar e ler modelos. Como pode ser visto nesta figura, durante o processo de modelagem do sólido é possível visualizar as *half-edges* de cada aresta o que viabiliza o aprendizado dinâmico do funcionamento básico da estrutura de dados.

Além disso, é possível selecionar uma das entidades (face, aresta ou vértice) e obter as informações geométricas e topológicas a respeito deste elemento conforme pode ser visto na barra, à esquerda na Figura 6.27, denominada como *Data Structure* (estrutura de dados). Para cada elemento topológico foi atribuído um ID como forma de identificação.

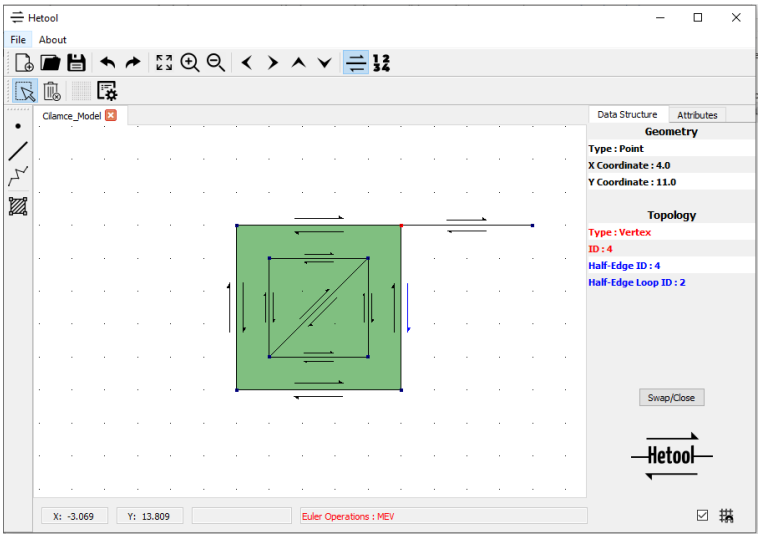
Uma aresta do sólido modelado foi selecionada na Figura 6.27.a, onde é possível obter as informações relacionadas a esta entidade que foi identificada como aresta 9. Na barra *Data Structure*, apresenta-se informações geométricas desta aresta tais como as coordenadas dos pontos iniciais e finais e o comprimento deste elemento. Esta barra também permite obter as informações topológicas pertinentes para o melhor entendimento da estrutura de dados, como as identificações das duas *half-edges* pertences a aresta selecionada e a identificação dos *loops* destas *half-edges*.



(a)



(b)



(c)

Figura 6.27 - Programa Hetool

Na Figura 6.27.b é possível visualizar as informações relacionadas a uma face selecionada que foi identificada como face 1. Na barra de estrutura de dados apresenta-se certas informações geométricas tais como a área da face, número de segmentos e pontos presentes no contorno externo da face selecionada. Ainda nesta barra, o usuário pode obter os dados topológicos da identificação do *loop* externo da face (destacada em azul), número de *loops* internos (destacado em laranja) e a *half-edge* (destacada em verde) que é armazenada pela face 1 e por onde inicia-se a navegação pelas as *half-edges* desta face.

A Figura 6.27.c apresenta as informações de um vértice selecionado que foi identificado como vértice 4. Na barra localizada à esquerda do sólido modelado, o usuário tem acesso as coordenadas do vértice selecionado e as informações topológicas da *half-edge* que é armazenada pelo vértice (destacada em azul).

No canto inferior da interface do aplicativo, existe um quadro denominado de *Euler Operations* (operadores de Euler) que exibe todas as operações de Euler utilizadas durante o processo da modelagem do sólido. Esse quadro auxilia no processo de ensino-aprendizagem dos conceitos básicos relacionados com os operadores de Euler que são uma parte fundamental para o funcionamento da estrutura de dados Half-Edge. Nas Figura 6.27.a e Figura 6.27.b, o quadro exibe a operação MEF que foi o resultado da adição da aresta 9 (aresta selecionada na Figura 6.27.a) à estrutura de dados. Já na Figura 6.27.c esse mesmo quadro exibe a operação MEV que foi o resultado da adição de uma nova aresta ligada ao vértice 4 (vértice selecionado nesta figura). Além disso, todos os outros operadores de Euler implementados na biblioteca HETOOL também podem ser exibidos por esse quadro.

7
Conclusões

Esse trabalho apresenta uma biblioteca voltada para a modelagem de subdivisões planares, denominada HETOOL e desenvolvida em Python, de código aberto e extensível, baseado na bem conhecida estrutura de dados Half-Edge. A biblioteca HETOOL foi desenvolvida pensando na facilidade e rapidez no uso. Esse pacote apresenta várias funcionalidades que permitem criar e gerenciar pontos, segmentos e regiões planares. Essa biblioteca também possibilita atribuir características particulares aos elementos geométricos a partir da aplicação de atributos. HETOOL oferece uma versatilidade na criação de novos atributos o que viabiliza o uso dos sólidos bidimensionais modelados em várias aplicações.

HETOOL é uma biblioteca de modelagem de subdivisões planares dinâmica que realiza a interseção automática entres os componentes geométricos modelados. As informações criadas por este pacote podem ser salvas, lidas e exportadas de uma forma bem simples por meio de um arquivo JSON. No futuro, espera-se aprimorar cada vez mais esta biblioteca oferecendo outras funcionalidades e ampliando o campo de aplicação deste pacote no meio científico.

Como exemplo de uso da biblioteca HETOOL foi desenvolvido um modelador de sólidos bidimensionais com foco na geração de modelos de elementos finitos. Esse modelador apresenta várias características que foram desenvolvidas objetivando a facilidade na manipulação da ferramenta e na modelagem de elementos finitos. Algumas destas características são: interface simples e intuitiva, múltiplas telas de modelagem, criação rápida de novos atributos e interface que se ajusta aos novos atributos criados. Esse modelador pode ser estendido e aplicado na simulação de problemas distintos da mecânica computacional.

7.1
Principais contribuições

A modelagem de sólidos é a base para várias pesquisas e vem cada vez mais sendo utilizada na resolução de problemas da geologia e da engenharia. Para modelagem computacional de sólidos é essencial o uso e gerenciamento de entidades topológicas e geométricas. Essas entidades são partes fundamentais da representação computacional de objetos reais, sendo responsáveis por armazenar as propriedades físicas e as informações geométricas dos objetos representados. Dessa forma, para gerenciar todas estas informações é necessária uma estrutura de dados que atenda a todas as questões geométricas e topológicas. Além disso, essa estrutura de dados deve ser rápida, eficiente e de fácil uso.

HETOOL é uma biblioteca de código aberto que visa contemplar todos os requisitos supracitados para modelos bidimensionais, oferecendo uma estrutura de dados com foco na modelagem de subdivisões planares, possibilitando a execução de um conjunto diversificado de análises devido à flexibilidade na configuração de novos atributos pelo usuário. Além disso, a biblioteca foi implementada e modularizada (seguindo o padrão MVC e da programação orientada a objetos) de forma mais intuitiva e atrativa aos usuários, possibilitando aprimoramentos. A estrutura de dados desenvolvida é dinâmica apresentando processamento automático das interseções entre as entidades geométricas presentes no modelo. Uma grande vantagem dessa estrutura de dados é que o usuário não precisa ter conhecimento sobre topologia para utilizar o código desenvolvido, pois todos os dados de entrada da maioria dos métodos de alto nível lidam apenas com coordenadas geométricas.

Apesar de já existirem outros pacotes de código aberto voltados para a modelagem de sólidos bidimensionais, poucos apresentam uma versatilidade na criação de atributos. HETOOL permite a execução de vários tipos de análises devido a flexibilização e generalização na criação de atributos. Além disso, a configuração de um novo protótipo de atributo utilizando o pacote HETOOL não é complicado. Para um novo protótipo de atributo ser criado basta adicionar um objeto no arquivo JSON seguindo um padrão específico.

Nessa biblioteca também existe um modulo que possibilita configurar os símbolos dos atributos. Esse modulo oferece um conjunto de métodos que são parametrizados pelas entidades geométricas. Esses métodos possibilitam criar e configurar formas básicas tais como quadrados, círculos, arco de círculos, triângulos e setas.

As linguagens de programação modernas como Python, utilizada no desenvolvimento da biblioteca HETOOL, e o JavaScript interpretam um arquivo JSON de forma já incorporada na própria linguagem, criando automaticamente estruturas, dicionários no caso da linguagem Python, que incorporam todos os dados presentes no arquivo JSON.

Todas as informações presentes no modelo podem ser salvas em um arquivo no formato JSON. Esse formato segue uma estrutura simples de *chave: valor* que possibilita uma comunicação rápida com outros aplicativos. Para ler esse arquivo em outros programas, basta filtrar as chaves dos objetos no arquivo JSON e obter as coordenadas geométricas dos pontos e segmentos presentes no modelo. Além disso, todos os atributos criados utilizando a biblioteca também podem ser filtrados e exportados através das chaves presentes no arquivo JSON.

7.2
Sugestões de trabalhos futuros

Tanto a biblioteca HETOOL quanto o modelador de sólidos FEMEP foram desenvolvidos de forma genérica o que permite que essas ferramentas possam ser aplicadas na resolução de vários problemas da mecânica computacional. Como tanto a estrutura de dados como o modelador de sólidos são projetos novos que ainda estão em fase de desenvolvimento, existem muitas funcionalidades que podem ser acrescentadas na biblioteca e no aplicativo.

Uma sugestão de trabalho futuro é implementar, unido ao modelador de sólidos, o processamento e pós-processamento de problemas de elasticidade para fins educacionais. Outros problemas da engenharia que envolvem a modelagem e

simulação também podem ser implementados visando a otimização do processo de ensino-aprendizagem de matérias específicas da engenharia.

Outros tipos de curvas poderiam ser adicionados ao modelador, tais como círculos, arco de círculos, curva de Bézier etc. Para utilizar essas curvas na estrutura de dados é necessário apenas transformar essas curvas em polilinhas equivalentes constituídas por um conjunto de pontos que definem a curva desejada com refinamento adaptativo à curvatura da curva. Além disso, o módulo de computação geométrica da estrutura de dados HETOOL que verifica as interseções geométricas também pode ser otimizado visando checar um menor número de elementos geométricos do modelo.

Outra sugestão seria a ampliação das funcionalidades presentes na biblioteca HETOOL. Existem muitas outras funcionalidades utilizadas na modelagem de sólidos que poderiam ser implementadas na biblioteca. Funcionalidades básicas tais como copiar, colar, recortar, rotação e translação do sólido ou parte do sólido selecionado também seriam incrementos válidos à biblioteca.

Sugere-se também a extensão deste trabalho para qualquer simulação que necessite da modelagem bidimensional de sólidos e que possa ser realizada e incrementada à biblioteca ou ao modelador de elementos finitos. Para isso, seria necessário criar os atributos associados e incrementar o processamento dos dados, incluindo a interface gráfica para a criação e visualização dos atributos. O gerenciamento de todas as informações geométricas e propriedades físicas do modelo será feita de forma automática a partir das ferramentas de apoio já criadas neste trabalho.

Por fim, a extensão deste trabalho para modelagem tridimensional seria muito importante para a área de mecânica computacional, com inúmeras aplicações. Isto é, sugere-se o desenvolvimento de uma biblioteca para modelagem de subdivisões espaciais baseada em uma estrutura de dados topológica com tratamento automático de interseções de superfícies e costura topológica dos retalhos de superfícies resultantes. A estrutura de dados adotada teria que representar sólidos *não-manifold*. Uma possibilidade seria a estrutura de dados Radial Edge desenvolvida por Weiler [26]. Seguindo a linha traçada neste trabalho, a versão tridimensional da

biblioteca também seria implementada em Python com código aberto e com gerenciamento genérico de atributos de simulação.

Referências bibliográficas

1 FILHO, Waldemar Celes. **Modelagem Configurável de Subdivisões Planares Hierárquicas**. Tese (Doutorado em Informática: Ciência da Computação) - Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 1995.

2 MARTHA, L.F.; **Análise Matricial de Estruturas com Orientação a Objetos, 1ª Edição**. Editora GEN LTC, ISBN (versão digital): 978-85-352-8798-1, 2018.

3 MARTHA, L.F. *et al*. FE Adaptive Analysis of Multi-regions Models. **Proceedings of the VI International Conference on Adaptive Modeling and Visualization (ADMOS 2013)**, p. 456-467, Portugal, 2013.

4 CAVALCANTI, Paulo Roma. **Criação e Manutenção de subdivisões no espaço**. Tese (Doutorado em Informática: Ciência da Computação) - Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 1992.

5 MÄNTYLÄ, M.; **An Introduction to Solid Modeling Computer**. Science Press, Rockville, Maryland, 1988.

6 QIAN, Jiang; LUO, Xiaohui; XUE, Yilan; Half-edge composite structure: good performance in motion matching. **Journal of Ambient Intelligence and Humanized Computing**, 2021.

7 ZHOU, G.; YUAN, S.; LUO, S.; Mesh Simplification Algorithm Based on the Quadratic Error Metric and Triangle Collapse, **IEEE Access**, vol. 8, p. 196341-196350, 2020

8 YI, Wenlong *et al*. Computer-aided Geometric Modeling of Plant Cell Shape and Design of Its Topological Retrieval Algorithms*.* **Proceedings of the XXIII International Conference on Soft Computing and Measurements (SCM)***,* p. 174-177, 2020.

9 ZHANG, Yingzhong; LUO, Xiaofang; JIA, Jia. A Compact Face-Based Topological Data Structure for Triangle Mesh Representation. **Computer-Aided Design and Applications**, v. 16, n. 3, p. 539-557, 2019.

10 CHATZIVASILEIADI, Aikaterini *et al*. Characteristics of 3D Solid Modeling Software Libraries for Non-Manifold Modeling. **Computer-Aided Design and Applications**, v. 16, n. 3, p. 496-518, 2019.

11 KARIM, Hairi *et al*. The Potential of the 3D Dual Half-Edge (DHE) Data Structure for Integrated 2D-Space and Scale Modelling: A Review. **Advances in 3D Geoinformation: Lecture Notes in Geoinformation and Cartography.** Springer, Cham, 2017.

12 BRUNO, Hugo B. S. *et al*. Interpretation of Density-Based Topology Optimization Results by Means of a Topological Data Structure. **VI International Symposium on Solid Mechanics - MecSol**, Joinville - SC, 2017.

13 BOGUSLAWSKI, Pawel; GOLD, Christopher. Buildings and terrain unified – multidimensional dual data structure for GIS. **Geo-spatial Information Science**, v. 18, n. 4, p. 151-158, 2016.

14 JAMALI, A., RAHMAN, A.A., BOGUSLAWSKI, P. et al. An automated 3D modeling of topological indoor navigation network. **GeoJournal**, v. 82, p. 157–170, 2017.

15 GOUDARZI, M.; ASGHARI, M.; BOGUSLAWSKI, P.; Rahman, A. A.; Dual Half Edge Data Structure in Database for Big Data in GIS, **ISPRS Ann. Photogramm. Remote Sens. Spatial Inf. Sci**., vol. II-2/W2, p. 41-45, 2015.

16 LIU, Yong-Jin. Exact geodesic metric in 2-manifold triangle meshes using edge-based data structures. **Computer-Aided Design**, v. 45, n. 3, p. 695-704, 2013.

17 CAMPOS, Jorge A.P.; **Geração de malhas de elementos finitos bidimensionais baseada em uma estrutura de dados topológica**. Dissertação de Mestrado (Mestrado em Engenharia Civil) - Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 1991.

18 CAMPOS, J. A. P.; MARTHA, L.F.; GATTASS, M.; Estrutura de Dados Topológica para Geração de Malhas Bidimensionais de Elementos Finitos**, Anais do XI Congresso Brasileiro de Engenharia Mecânica, ABCM**, São Paulo, vol. XIII, p. 137-140, 1991.

19 CAVALCANTI, P.R., CARVALHO, P.C.P.; MARTHA, L.F.; Criação e Manutenção de Subdivisões Planares, **Anais do IV Simpósio Brasileiro de Computação Gráfica e Processamento de Imagens**, USP/SBC, São Paulo, 1991, p. 13-24, 1991.

20 SILVEIRA, E.S.S.S.; **Um sistema de modelagem bidimensional configurável para simulação adaptativa em mecânica computacional**. Dissertação de Mestrado (Mestrado em Engenharia Civil) - Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 1995.

21 CARVALHO, M.T.M.; **Uma estratégia para desenvolvimento de aplicações configuráveis em mecânica computacional**. Tese (Doutorado em Engenharia Civil) - Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 1995.

22 LIRA, W.W.M.; **Um sistema integrado configurável para simulações em mecânica computacional**. Dissertação de Mestrado (Mestrado em Engenharia Civil) - Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 1998.

23 HOFFMANN, C.M.; **Geometric & Solid Modeling: An Introduction. Purdue University**, indiana, 1989.

24 BORTOLOSSI, Humberto J.; **Notas de Aula de um Curso de Modelagem de Sólidos em Computação Gráfica**. Universidade Federal de Fluminense, 2017.

25 ARRUDA, Marcos C.; **Operações booleanas com sólidos compostos representados por fronteira**. Dissertação de Mestrado (Mestrado em Engenharia Civil) - Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2005.

26 WEILER, K.; **Topological Structures for Geometric Modeling**. Doctor of Philosophy (PhD) - Rensselear Polytechnic Institute, 1986.

27 BOGUSLAWSKI, P.; **Modeling and Analysing 3D Building Interiors with The Dual Half-Edge Data Structure**. Doctor of Philosophy (PhD) - University of Glamorgan, 2011.

28 STROUD, Ian. **Boundary Representation Modelling Techniques**. United States of America: Springer, 2006. 788 p. ISBN 978-1-84628-312-3.

29 LEE, K.; **Principles of CAD/CAM/CAE Systems**. Addison-Wesley Longman Publishing Co., Inc., USA, 1999.

30 GOMES, Gilberto. **HeDcpp: estrutura de dados para aplicação em programas de engenharia voltados ao ensino-aprendizagem**. Exacta – EP, São Paulo, v. 12, ed. 2, p. 209-218, 2014

31 LEIVAS, José C. P.; Empregando intuição topológica no ensino de geometria na escola básica**. I Congresso de Educación Matemática de América Central y El Caribe, República Dominicana**, 2013.

32 SPERLING, David. Entre Conceitos, Metáforas e Operações: convergências da topologia arquitetura contemporânea. **Gestão & Tecnologia de Projetos**, São Paulo. v. 3, ed. 2, 2008.

33 WORBOYS, M.; DUCKHAM, M.; **GIS: A Computing Perspective**. CRC Press, 2004.

34 LIMA, Ronaldo F.; **Topologia e Análise no Espaço** $\mathbb{R}^n$. Natal RN, 2013. 244 p.

35 CHATZIVASILEIADI, Aikaterini *et al*. Addressing pathways to energy modelling through non-manifold topology. **Symposium on Simulation for Architecture and Urban Design**, Netherlands, 2018.

36 DYEDOV, V., Ray *et al*. AHF: array-based half-facet data structure for mixed-dimensional and non-manifold meshes. **Engineering with Computers,** v. 31, p. 389-404 ,2015.

37 JABI, Wassim. The potential of non-manifold topology in the early design stages. **Proceedings of the 35th Annual Conference of the**

Association for Computer Aided Design in Architecture, Ohio, US, 2015.

38 BOGUSLAWSKI, P.; GOLD, C.; The Dual Half-Edge- A Topological Primal/Dual Data Structure and Construction Operators for Modelling and Manipulating Cell Complexes. **ISPRS Int. J. Geo-Inf***,* v. 5, n.2, 2016.

39 ELLUL, Claire. **Functionality and Performance – Two Important Considerations When Implementing Topology In 3D**. Doctor of Philosophy (PhD) - University of London, 2007.

40 JABI, Wassim et al. Linking design and simulation using non-manifold topology. **Architectural Science Review,** v. 59, n. 4, p. 323-334, 13 jan. 2016.

41 COSTA, Márcio; PEREIRA, André. Desenvolvimento de Aplicativo Educacional para Determinação de Propriedades Geométricas de Áreas. **Proceedings of the XXXIV Iberian Latin-American Congress on Computational Methods in Engineering**, Pirenópolis - GO, 2013.

42 CELES, Waldemar. **Estruturas de Dados Topológicas: Representação de Malhas**. Rio de Janeiro: Tecgraf (PUC-RIO), 2015.

43 GOLD, C.M.; Data structures for dynamic and multidimensional GIS. **4th ISPRS Workshop on Dynamic and Multi-dimensional GIS**, 2005.

44 BOGUSLAWSKI, P., GOLD, C.; Euler Operators and Navigation of Multi-shell Building Models.**Developments in 3D Geo-Information Sciences. Lecture Notes in Geoinformation and Cartography.** Springer, Berlin, 2010.

45 BAUMGART, B.G.; A polyhedron representation for computer vision. **Proceedings of the National Computer Conference (AFIPS '75)**, Anaheim, CA, USA, 1975.

46 GUIBAS, Leonidas; STOLFI, Jorge. Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams. **ACM Transactions on Graphics**, Estados Unidos, v. 4, n. 2, p. 74-123, 1985.

47 PARIS, Richard. **Modified half-edge data structure and its applications to 3D mesh generation for complex tube networks**. 2013. Dissertação de Mestrado (Master of Engineering) - Universidade de Louisville, 2013.

48 RAMNATH, Sarnath; DATHAN, Brahma. **Object-Oriented Analysis and Design**. London: Springer, 2011. 440 p. ISBN 978-1-84996-521-7.

49 MEI G. *et al*. Ear-Clipping Based Algorithms of Generating High-Quality Polygon Triangulation. **Proceedings of the 2012 International Conference on Information Technology and Software Engineering**. Lecture Notes in Electrical Engineering, v. 212, Springer, Berlin, Heidelberg, 2013.

50 MIRANDA, A.C.O.; MARTHA, L.F.; Uma Biblioteca Computacional para Geração de Malhas Bidimensionais e Tridimensionais de Elementos Finitos. **Proceedings of the XXI CILAMCE – 21st Iberian Latin-American Congress on Computational Methods in Engineering**, Rio de Janeiro, Brazil, v. 3, n. 9, p. 1-15, 2000.

51 BOMFIM, D. S. *et al*. Development of a Python Application Aiming at the Teaching-learning Process of the Half-Edge Data Structure. **Proceedings of the joint XLII Ibero-Latin-American Congress on Computational Methods in Engineering and III Pan-American Congress on Computational Mechanics**, Rio de Janeiro, 2021.