Universidade Federal Fluminense

Escola de Engenharia

Departamento de Engenharia Civil

Danielle Duque Estrada Pacheco

Estruturas de Dados Topológicas para Modelagem Geométrica Bidimensional de Problemas de Engenharia

Niterói

2011

Danielle Duque Estrada Pacheco

Estruturas de Dados Topológicas para Modelagem Geométrica Bidimensional de Problemas de Engenharia

Projeto Final de graduação junto ao curso de Engenharia Civil da Universidade Federal Fluminense, na área de Análise Estrutural.

Orientador: Prof. André Maués Brabo Pereira

Niterói

2011

Sumário

RESUMO

Atualmente, as simulações computacionais vêm adquirindo crescente importância em grande parte das áreas de engenharia. Uma das etapas mais importantes de uma simulação computacional de um problema de engenharia é a geração de um modelo geométrico que represente de forma mais realística e apropriada o problema a ser simulado. Para que seja possível criar e manipular um modelo geométrico é necessário utilizar uma estrutura de dados topológica (preservando a consistência do modelo). Por este motivo, o objetivo deste trabalho consiste da implementação de uma estrutura de dados topológica para utilização em modeladores geométricos bi-dimensionais, com a finalidade de criar e manipular modelos de forma eficiente e consistente, para que futuramente possam ser utilizados em simulações computacionais. Visando atender a essas expectativas, adotou-se a Orientação a Objetos, uma técnica moderna, flexível, eficiente e capaz de criar códigos robustos e reutilizáveis. Aliada a essa técnica, utiliza-se a *Unified Modeling Language* (UML), uma linguagem gráfica que permite a visualização, construção e documentação do desenvolvimento de um sistema computacional orientado a objetos. Para que um programa de análise possa ser utilizado de forma direta e dinâmica, também será desenvolvida uma interface gráfica para a geração e manipulação dos modelos, facilitando a interpretação e a identificação dos conceitos desenvolvidos neste trabalho.

1. INTRODUÇÃO

Com a grande evolução da computação nas últimas décadas, o uso desta tecnologia vem se tornando indispensável na resolução de problemas das mais diversas áreas da engenharia. Problemas estes que, antes da era da computação avançada, eram de solução muito demorada, ou até mesmo impossível, devido à alta necessidade de poder de cálculo envolvida na análise destes.

No que tange a problemas de engenharia em específico, outra dificuldade que muitas vezes se enfrenta, além dos cálculos relacionados ao problema, é a modelagem destes problemas e suas soluções. Estas modelagens muitas vezes são complexas demais para serem desenvolvidas arcaicamente, sem o auxílio de uma máquina. É neste momento que o processo de modelagem computacional vem para facilitar e, principalmente, acelerar a solução de problemas existentes.

Dessa forma, pode-se notar a importância da manipulação de um modelo geométrico e de sua geração correta, as quais estão ligadas diretamente a uma estrutura de dados topológica, que preserve a consistência do modelo.

1.1 Justificativas

Nas últimas décadas, as simulações computacionais vêm adquirindo grande importância nas mais diversas áreas da engenharia, pois é através delas que se torna possível a redução considerável do custo, do tempo e dos recursos consumidos na resolução dos problemas simulados. Porém, sem a utilização de uma estrutura de dados topológica apropriada à geração (assim como a manipulação) eficiente dos modelos, esta se torna uma tarefa inviável. Além disso,

durante o processo de simulação de diversos problemas de engenharia ocorrem mudanças na geometria dos modelos, sendo imprescindível o controle da mesma para atualizar tais modificações de forma correta e eficiente.

Dessa forma, pesquisas na área de estrutura de dados são fundamentais para aumentar a generalidade e eficiência computacional da representação de modelos geométricos.

As estruturas de dados topológicas também possuem grande importância no desenvolvimento de ferramentas educacionais, que visam auxiliar no ensino de tópicos de disciplinas de engenharia. O aprendizado com auxílio de recursos computacionais tem sido empregado com êxito, tanto no Brasil como no exterior, e, nesse contexto, *softwares* educacionais que utilizam ferramentas gráficas vêm ajudando alunos e professores a tornarem o aprendizado mais fácil, rápido e eficiente.

Dessa forma, o uso de recursos computacionais no ensino de disciplinas de engenharia de estruturas trará benefícios aos alunos, ao ajudá-los a visualizar várias situações teóricas importantes e formarem a capacidade de procurar informações e transformá-las em conhecimento.

1.2 Objetivos

O objetivo principal deste projeto consiste na implementação computacional de uma estrutura de dados topológica, que possibilite criar e manipular modelos bidimensionais de forma consistente, para a realização de pesquisas em simulações computacionais e geração de ferramentas para ensino de engenharia. Para realizar

este objetivo, optou-se pela Abordagem Orientada a Objetos (POO) (Booch, 2005) no estado da arte, por ser uma técnica moderna que resulta em códigos robustos e eficientes. Paralelamente, também será desenvolvida uma interface gráfica para o pré-processamento dos modelos, permitindo que o programa de análise possa ser utilizado de forma mais consistente e dinâmica. A implementação dos códigos será feita utilizando a linguagem de programação C++ por, dentre outros fatores, possuir alta flexibilidade, portabilidade, consistência e por fornecer um ótimo suporte ao paradigma da Programação da Orientação a Objetos.

Outro objetivo importante do presente projeto é criação de ferramentas educacionais para o ensino de tópicos de disciplinas de engenharia, onde a primeira ferramenta a ser criada é um modelador simples para geração de modelos geométricos, que futuramente poderá determinar as propriedades geométricas, tais como centróide e momento de inércia, de seções genéricas de vigas e colunas.

Nesse contexto, foi necessário desenvolver o conhecimento das linguagens de programação C e C++, da programação orientada a objetos, além da familiarização com os programas *Microsoft Visual Studio* 2008 e o *toolkit* Qt.

1.3 Organização do Trabalho

Este trabalho está dividido em 7 capítulos. Neste primeiro capítulo, é apresentada a introdução do trabalho, com justificativas, objetivos e a descrição resumida do conteúdo dos demais capítulos.

No capítulo 2, serão apresentados os fundamentos principais de modelagem geométrica, que são conceitos essenciais para o entendimento dos assuntos que

serão descritos nos próximos capítulos. Por sua vez, o capítulo 3, aborda conceitos básicos da Orientação a objetos, bem como sua importância no campo da programação.

No capítulo 4, apresenta-se a principal contribuição deste trabalho que é a arquitetura orientada a objetos da estrutura de dados topológica. Na sequência, trechos de código da implementação dos operadores de Euler são apresentados no capítulo 5 para ilustrar funcionalidades de partes essenciais do programa. No capítulo 6, demonstram-se alguns exemplos práticos de aplicação do programa com a estrutura de dados.

Finalmente, no capítulo 7, serão apresentadas as conclusões e as sugestões para trabalhos futuros.

2. FUNDAMENTOS DA MODELAGEM GEOMÉTRICA

Um modelo é um objeto construído artificialmente com a finalidade de tornar mais fácil a visualização, observação ou análise de outro objeto (Mäntylä, 1988). Diferentes áreas produzem modelos para que fenômenos relativos aos objetos que realmente se deseja estudar possam ser analisados sem a necessidade de que o objeto inclusive exista, ou seja, diretamente observável. Como exemplo, podemos citar modelos físicos, modelos moleculares, modelos matemáticos ou desenhos de engenharia.

Modelos computacionais consistem em informações armazenadas em arquivos ou na memória de um computador que podem ser utilizadas para realizar tarefas semelhantes aos outros tipos de modelos citados acima de forma eficiente e confiável. Para resolução de problemas de engenharia com suporte computacional, a geometria do objeto é a parte mais importante do conjunto de informações que podem ser armazenadas sobre este modelo. Analisar ou testar um modelo, além de ser mais prático, é mais conveniente e econômico do que fazê-lo com o objeto ou o processo real.

Ao conjunto de métodos usados para definir a forma e outras características geométricas de um objeto dá-se o nome de Modelagem Geométrica. Atualmente, esse conceito também é estendido à utilização de computadores para auxiliar a criação, manipulação, manutenção e análise das representações das formas geométricas de objetos no plano ou no espaço (Weiler, 1986).

A modelagem geométrica abrange a área da geometria computacional e estende-se além dessa para o campo da modelagem de sólidos, criando uma importante interação entre a geometria e a computação. Um dos recursos mais

versáteis para a representação e manipulação dos objetos geométricos são as estruturas de dados, as quais são responsáveis pelo armazenamento das informações topológicas de forma eficiente.

Para facilitar o entendimento dos assuntos que serão tratados na próximas seções, alguns conceitos fundamentais devem ser introduzidos, tais como topologia e geometria e superfícies *Manifold* e *Non-Manifold*.

Topologia e Geometria

Pode-se definir a geometria completa de um objeto como sendo o conjunto de informações essenciais para definir a forma deste objeto e a localização espacial de todos os seus elementos constituintes.

A topologia é uma abstração e pode ser definida como um conjunto coerente de informações obtidas a partir da descrição geométrica completa de um objeto. Tal conjunto de informações não varia após a aplicação de transformações afins ao objeto. Ou seja, propriedades que se modificam devido transformações afins não fazem parte deste conjunto. Pode-se notar, então, que a topologia é incompleta na descrição de um objeto, ela não possui informações suficientes para que o objeto possa ser modelado de forma única e completa. Por outro lado, a geometria completa daquele objeto permite que ele seja perfeitamente modelado. Isto significa que não se pode obter todas as informações geométricas a partir da topologia de um objeto, mas o contrário é válido.

O uso da topologia em ambientes de modelagem é sustentado pelas ideias de economia de tempo e complexidade dos algoritmos. Quando a topologia se

caracteriza como um conjunto de informações unificadas, coerentes e organizadas com alto nível de abstração, pode-se obter, de forma simples e rápida, informações de extrema importância relativas aos elementos topológicos sem haver a necessidade de se fazer uma consulta global à geometria do modelo.

Alguns autores (Mäntylä, 1988; Chiyokura, 1988) justificam o uso da topologia baseado em três razões, além da já citada anteriormente.

A primeira razão encontra-se no fato de que a geometria de superfícies curvas ainda é uma área que se encontra em estudos, diversos modelos matemáticos desse tipo de superfícies estão sendo estudados/implementados, enquanto que a topologia já é algo consolidado e é capaz de representar esses tipos de superfícies sem a necessidade de grandes alterações. Sendo possível mudar o enfoque geométrico sem mudar a topologia.

A segunda razão está intimamente ligada a erros numéricos, como erros de arredondamento e falta de precisão. Pois quando trabalhamos com a geometria é necessário a comparação de valores, que pode ocasionar erros numéricos; já quando trabalhamos com a topologia essas relações são ”ligadas” no momento da criação do objeto, propiciando assim uma representação robusta e consistente.

A terceira e última razão encontra-se no fato de que a separação de informações geométricas e topológicas torna a implementação do sistema mais organizada e simples.

Utilizar a topologia como base ou elemento-chave de um sistema de modelagem significa disponibilizar explicitamente as informações topológicas bem como criar uma estrutura de dados e algoritmos que a utilizam baseados nestas informações.

As entidades topológicas que representam as entidades geométricas pontos, curvas e regiões são, respectivamente, vértices, arestas e faces. A Figura 1 ilustra a representação dessas entidades topológicas.

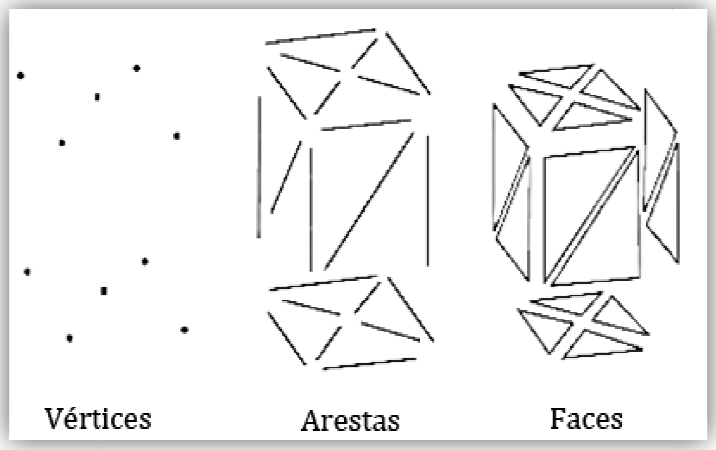


Figura 1 - Entidades topológicas.

As relações de adjacências são informações a respeito de quais entidades topológicas, tais como vértices, arestas e faces, se tocam (e em que ordem). Elas são a base da *topologia de adjacência*, que trata das adjacências físicas dos elementos topológicos imersos no espaço ou nas superfícies dos objetos. Este tipo de topologia é o mais utilizado e é o que será estudado neste trabalho.

- **Superfícies *Manifold* x *Non-Manifold***

O conceito de *manifold* permite caracterizar de forma rigorosa uma importante classe de objetos geométricos de grande interesse no contexto da modelagem geométrica. Uma superfície *manifold* ou *2-manifold* é um espaço topológico onde cada ponto possui uma vizinhança aberta equivalente a um disco bidimensional. Isto quer dizer que, se analisada localmente numa área pequena o suficiente no entorno de um ponto dado, uma superfície existente num espaço tridimensional pode ser considerada “chata” ou plana. Pode-se dizer que deformando a superfície

localmente para um plano, ela não rasga ou passa a possuir pontos coincidentes. Num poliedro *manifold*, cada aresta pertence a exatamente a duas faces. Esta propriedade é fundamental para a utilização de algumas estruturas de dados topológicas, como a *Winged-Edge* e a *Half-Edge*, que serão estudadas nas seções seguintes.

Os objetos que não se enquadram na definição de *manifold* são denominados *non-manifold*. Numa superfície *non-manifold*, a vizinhança ao redor de um ponto qualquer numa superfície não precisa ser um disco bidimensional, ou seja, mesmo analisando localmente uma área bem pequena ao redor do ponto, ela pode não ser "chata" ou plana. Os objetos *non-manifold* são caracterizados por ocorrências topológicas, tais como: um vértice tocando a superfície de um sólido; uma aresta pertencendo a mais de duas faces; múltiplos volumes incidindo sobre um mesmo vértice; objetos com estruturas interiores.

Apesar de não ser objeto de estudo desse trabalho, é importante ressaltar que existe uma estrutura de dados capaz de representar todas as possíveis relações de adjacências que podem ocorrer em superfícies *non-manifold*. Esta estrutura foi desenvolvida por Weiler (1986) e recebeu o nome de *Radial-Edge*. A Figura 2 ilustra um objeto *non-manifold* obtido como resultado da operação booleana de união de dois objetos *manifold*.

2.1 Esquemas de Representação

A modelagem de sólidos procura definir representações que codifiquem o conjunto infinito de pontos que compõem um sólido em uma porção finita da memória do computador, da forma mais genérica possível. As representações que atendem a esta codificação podem ser divididas em três grandes grupos:

Modelos de Decomposição – descrevem sólidos como uma coleção de componentes básicos, combinados entre si pela operação de colagem. Os principais esquemas de representação desse modelo são brevemente descritos em (a) e (b) da Figura 3.

Modelos de Construção – são os mais utilizados no contexto atual para aplicações em CAD[1], os quais representam um conjunto de pontos do espaço tridimensional como uma combinação de conjuntos de pontos primitivos, incluindo operações booleanas (união, interseção e diferença) e as transformações geométricas (rotação, translação e escalamento). O principal esquema de representação desse modelo também se encontra brevemente descrito em (c) da Figura 3.

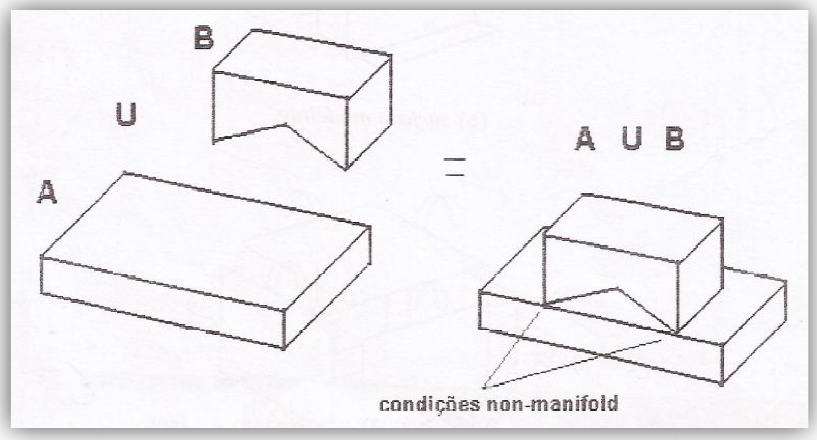
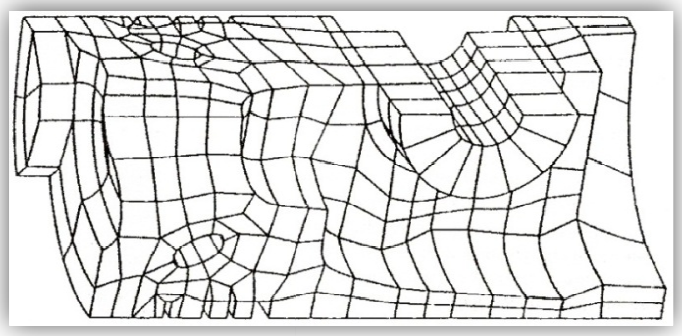


Figura 2 - Um objeto *non-manifold* resultado da união de dois objetos *manifold*.

[1]*Computer Aided Design* ou Projeto Assistido por Computador – compreende a utilização de técnicas e recursos computacionais em projetos de engenharia, arquitetura, etc.

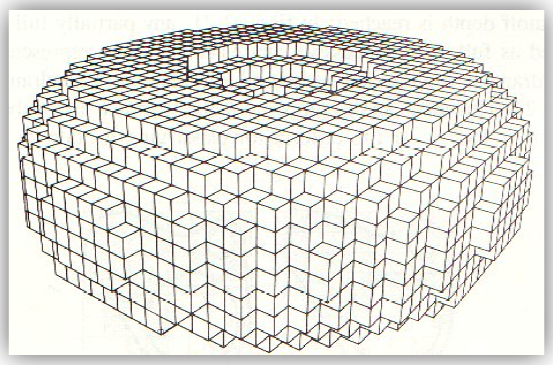
(a) **DECOMPOSIÇÃO CELULAR**



• Mais preciso entre os de decomposição
• Sólido é decomposto em células, sendo que casa uma possui um número arbitrário de lados, que compartilham um ponto, uma aresta, ou uma face.
• Células primitivas são parametrizadas e combinadas pela operação de colagem.
• Uso como representação básica pra métodos de elementos finitos.

(Mäntylä, 1988)

(b) **ENUMERAÇÃO EXAUSTIVA**



• Caso especial da Decomposição Celular, no qual as células são idênticas.

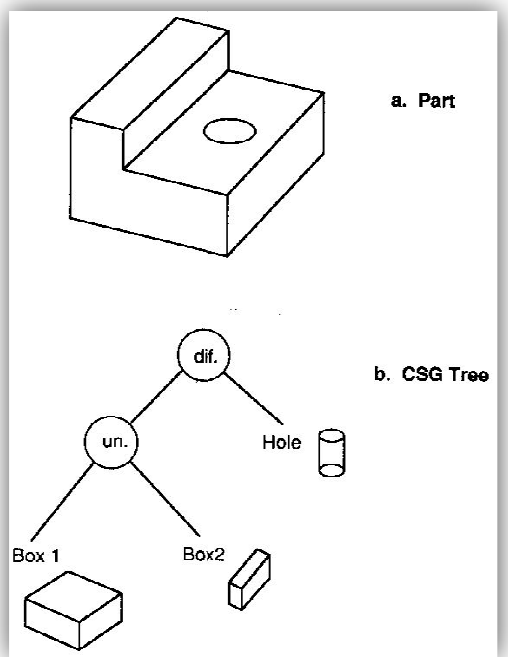
•Usado em modelos onde a precisão é menos importante que a eficiência.

• Sólido é visto como um conjunto de cubos de tamanho fixo (*voxels – volume elements*) localizados em um grid espacial fixo.

• Gera arranjo ordenado de tuplas 3D, que estão ou não ocupadas pelo sólido.

(Mäntylä, 1988)

(c) **GEOMETRIA SÓLIDA CONSTRUTIVA (CSG)**



• Uso bastante difundido, devido ao embasamento matemático, conhecimento de algoritmos e à ampla área de aplicação.

• Sólidos descritos em termos de primitivas, combinadas por operadores booleanos regularizados e movimentos rígidos.

• Utilizam como representação interna árvores binárias, nas quais cada folha é uma primitiva e cada nó interno é um operador booleano (união, interseção ou diferença) ou um movimento rígido.

(Mäntylä, 1988)

Figura 3 - Principais esquemas de representação dos Modelos de Decomposição e Construção.

2.2 Modelos de Fronteira (B-Rep)

Modelos de Decomposição e de Construção analisam os sólidos como um conjunto de pontos e procuram representá-los seja através da discretização destes ou construindo os sólidos a partir de conjuntos mais simples. Em contrapartida, os Modelos de Fronteira representam os sólidos através de uma coleção de faces, limitadas por arestas, que por sua vez são limitadas por vértices (Mäntylä, 1988).

Um B-Rep pode ser definido como um grafo[2] com nós correspondendo às faces, arestas e vértices que definem a fronteira topológica do sólido. A descrição da superfície de um sólido constitui-se de duas partes, uma topológica e outra geométrica. A descrição geométrica permite a imersão dessa superfície no espaço. A descrição topológica diz respeito à conectividade e orientação dos vértices, arestas e faces.

Basicamente, as informações topológicas contêm especificações sobre vértices, arestas e faces de forma abstrata, indicando suas incidências e adjacências. As informações geométricas especificam, por exemplo, as equações das superfícies que contêm as faces, e das curvas que contêm as arestas. O armazenamento explícito das informações topológicas ocasiona um gasto elevado de memória.

Apresentam-se na Figura 4 os componentes básicos de um modelo de fronteira, onde em (a) a superfície do objeto é dividida em um conjunto fechado de faces, cada qual sendo representada pelo seu polígono delimitador em (b), que por sua vez é representado em termos de arestas e vértices em (c).

[2] Um grafo é representado como um conjunto de pontos (vértices), ligados por retas (arestas).

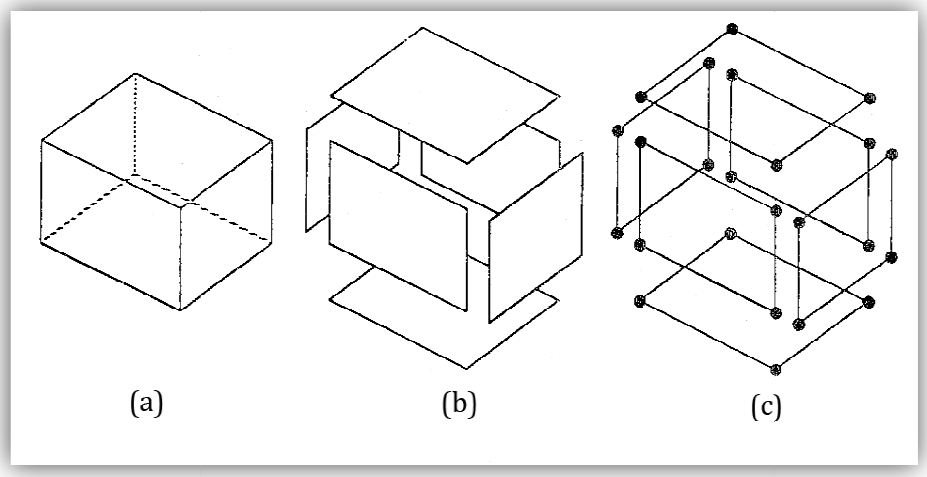


Figura 4 - Componentes básicos de um Modelo de Fronteira.

Um dos principais esquemas de representação desse modelo é o baseado em arestas. Em modelos de fronteira baseados em arestas, a fronteira de uma face é representada em termos de uma sequência fechada e orientada de arestas. Os vértices da face são obtidos através das arestas, igualmente.

Cada aresta possui uma orientação, que se caracteriza pela ordenação dos seus vértices (inicial e final), e as faces são novamente orientadas segundo uma lista ordenada no sentido horário ou anti-horário das suas arestas segundo um observador externo ao sólido. Cada aresta pertence simultaneamente a duas faces, com orientações distintas em cada uma delas.

- **Estruturas de Dados *Winged-Edge***

Uma das primeiras estruturas de dados que adotou a representação da fronteira baseada em aresta foi a *Winged-Edge*, que foi desenvolvida inicialmente por Baumgart, em 1972, para aplicações na área de visão computacional e robótica.

A *Winged-Edge* é capaz de representar superfícies *manifolds* armazenando informações sobre arestas, vértices e faces. As informações armazenadas em cada aresta são formadas de arestas, vértices e faces adjacentes. Dessa forma, é possível ver que a *Winged-Edge* é fortemente baseada na entidade aresta. Para manter a robustez da estrutura de dados topológica é necessário que cada um dos elementos pertencentes às entidades (vértices, arestas e faces) sejam unicamente identificados. O nome *Winged-Edge* resulta da aparência gráfica das arestas adjacentes quando desenhadas em relação à aresta de referência.

Na implementação da estrutura da *Winged-Edge* é necessário orientar arestas e faces, porque levam-se em conta as arestas imediatamente anteriores e imediatamente posteriores à aresta de referência em relação às duas faces que ela limita. Além disso, a aresta de referência é usada uma única vez em seu sentido positivo (do vértice inicial para o vértice final) e uma única vez no seu sentido negativo (do vértice final para o inicial), uma para cada face que ela separa. Pode-se atribuir os qualificativos face esquerda e face direita referentes àquela aresta orientada baseando-se num observador externo ao sólido que tais faces delimitam.

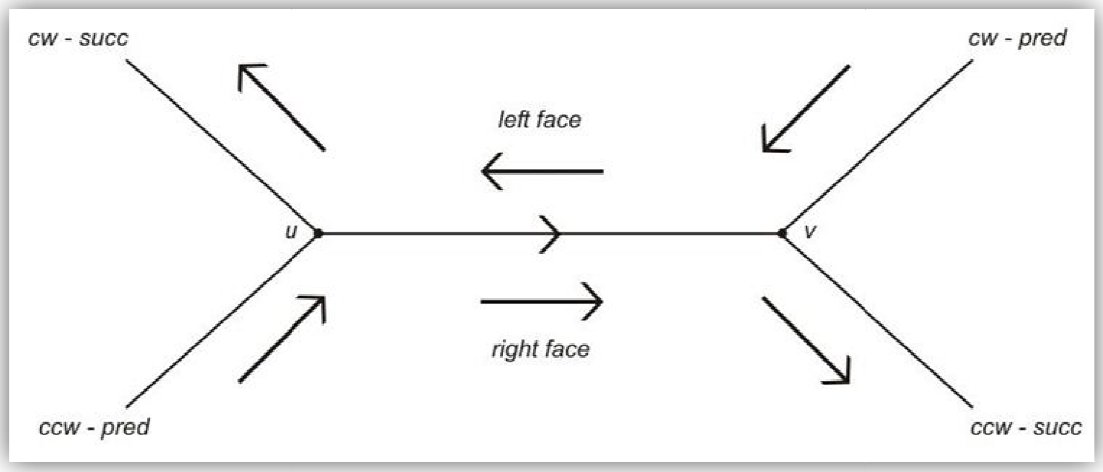


Figura 5 - Estrutura de dados *Winged-Edge.*

Por ser uma estrutura de dados limitada, já que não pode percorrer possíveis furos num modelo, optou-se para construção do modelador a estrutura de dados *Half-Edge*, que será vista na seção 2.4.

2.3 Operadores de Euler

A manipulação de modelos de fronteira, como os apresentados anteriormente, é um tanto quanto complexa para ser realizada diretamente. Quando um modelo está sendo gerado num ambiente de modelagem, diversas operações de criação e remoção de entidades topológicas são constantemente realizadas. No processo de criação de objetos num sistema de modelagem geométrica, cada etapa intermediária requer a inserção, modificação ou eliminação de elementos topológicos, o que pode ser uma tarefa não trivial, já que quando se manipula um elemento normalmente são ocasionadas modificações na estrutura topológica do modelo como um todo, provocando alterações em outros elementos topológicos.

Os operadores de Euler, apresentados originalmente por Baumgart (1975), são um conjunto de operadores topológicos que realizam as operações sobre uma superfície *manifolds* representada pela estrutura *Winged-Edge*. Por exemplo, a criação de elementos conectados como uma face e um vértice, ou uma aresta e uma face, ou uma aresta e um vértice.

Esses operadores são responsáveis pela consistência topológica do modelo, isto é, dada uma coleção de faces, arestas e vértices, esta só será um modelo de fronteira válido caso o número destes elementos satisfaçam a relação de Euler-Poincaré, dada pela seguinte equação:

$$V - E + F = 2(S - H)$$

onde *V*, *E*, *F*, *S* e *H* denotam, respectivamente, o número de vértices, arestas, faces, cascas e furos num sólido.

Para uma topologia do tipo *2-manifold*, que é o foco desse trabalho, esta relação pode ser dada por:

$$V - E + F = 2$$

Verifica-se, portanto, que existe um critério necessário para que se possa garantir a integridade do modelo, seja no plano ou no espaço. Dessa forma, através dos Operadores de Euler, é possível verificar a consistência do modelo, atribuindo-lhe uma relação única e constante entre todas as entidades envolvidas.

2.3.1 Notações e Convenções

Por convenção histórica, os operadores de Euler são preferencialmente denotados por nomes de fácil associação. Seguindo as convenções adotadas por Mäntylä (1988), os operadores serão nomeados da seguinte forma:

M → *make* (criar)	K → *kill* (deletar)
S → *split* (separar)	J → *join* (juntar)
V → *vertex* (vértice)	E → *edge* (aresta)
F → *face* (face)	S → *solid* (sólido)
H → *hole* (furo)	R→ *ring* (anel)

Outro termo importante é o *loop* que pode ser descrito como mais uma entidade topológica e representa o ciclo que as arestas fazem em torno de uma face.

Por exemplo, o operador “*mev*” significa *Make Edge, Vertex*, ou seja, cria uma aresta e um vértice. Neste trabalho, os operadores de Euler serão implementados para a estrutura de dados *Half-Edge* (semi-aresta) e serão denotados em caixa baixa, como no exemplo anteriormente citado.

2.3.2 Operadores Básicos

A seguir, serão descritos os operadores de Euler utilizados no trabalho.

- **MVFS (Make Vertex Face Solid) e KVFS (Kill Vertex Face Solid)**

Este operador cria a partir do nada uma instância da estrutura de dados que possui uma face e um único vértice. A nova face possui um *loop* vazio, sem nenhuma aresta. O "sólido" criado não satisfaz a noção intuitiva de sólido, constituindo uma "forma esquelética", porém é útil como estágio inicial na criação de sólidos B-rep por meio de uma sequência de operadores de Euler.

De maneira similar às operações de manipulação no modelo plano, todos os operadores de Euler terão operadores inversos correspondentes, que podem desfazer o efeito do operador inicial. O inverso do MVFS é o KVFS.

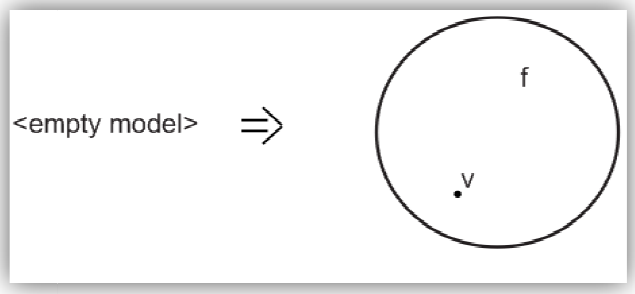


Figura 6 - Operador de Euler MVFS.

- **MEV (Make Edge Vertex) e KEV (Kill Edge Vertex)**

Este operador subdivide um vértice em dois, unindo-os por uma nova aresta, tendo como efeito a adição de um vértice e uma aresta à estrutura de dados. Podem ser utilizados em três situações distintas, como mostrado na Figura 7: (a) partindo da forma esquelética, na qual o *loop* existente é vazio – o vértice sem aresta é dividido em dois, que são unidos pela aresta criada; (b) subdividindo o ciclo de arestas de um vértice em dois ciclos através da divisão de um vértice em dois, juntando-os com uma nova aresta; (c) gerando um novo vértice que é unido a um vértice já existente, através de uma nova aresta criada.

A operação inversa ao MEV é executada pelo KEV, que é capaz de desfazer qualquer um dos três casos acima descritos.

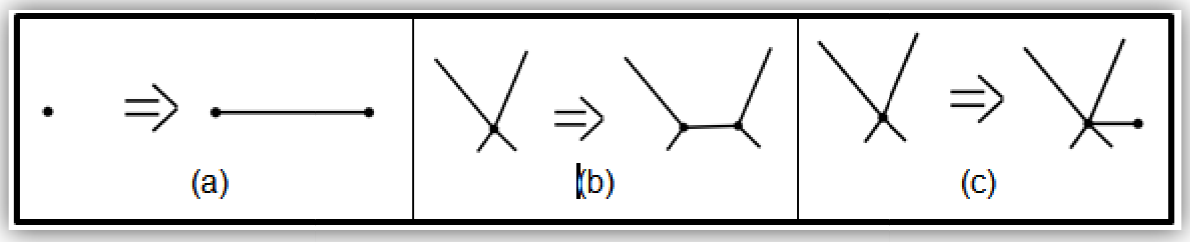


Figura 7 - Operador de Euler MEV.

- **MEF (Make Edge Face) e KEF (Kill Edge Face)**

Este operador subdivide um *loop*, fazendo a ligação de dois vértices por uma nova aresta, ou seja, adiciona uma nova aresta e uma nova face à estrutura de dados.

Além do caso comum mostrado na Figura 8(a), as aplicações do MEF são estendidas para os loops vazios, em analogia direta ao MEV. Então, o resultado da subdivisão de um vértice “solitário” consiste de uma aresta “circular” separando

duas faces. Os vértices iniciais e finais de uma aresta deste tipo são iguais, este caso é admitido pela estrutura de dados *Winged-Edge*, como mostrado Figura 8(b). No caso mais geral, é sempre possível conectar um vértice com ele mesmo através do MEF, como mostrado na Figura 8(c).

A operação inversa do MEF, o KEF, pode desfazer o efeito do MEF em cada um dos três casos. Mais precisamente, dada uma aresta adjacente a duas faces distintas, o KEF é capaz de remover a aresta, e unir as duas faces em uma, de forma que tem seu loop de fronteira é resultado da fusão das duas fronteiras originais.

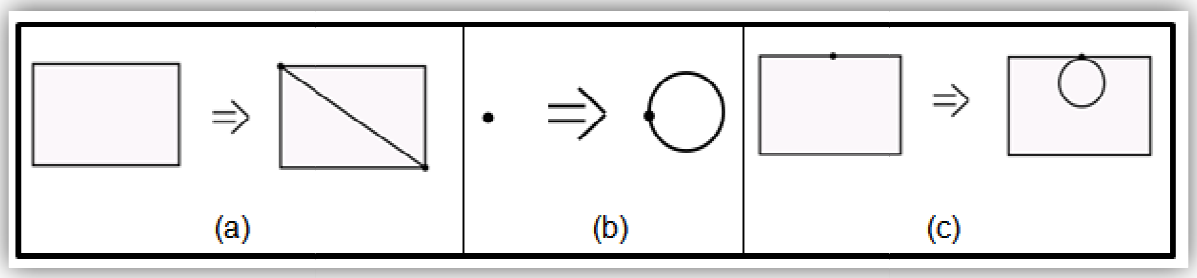


Figura 8 - Operador de Euler MEF.

- **KEMR (Kill Edge Make Ring) e MEKR (Make Edge Kill Ring)**

Os *loops* vazios acima descritos foram utilizados para realizar o modo plano esquelético primitivo do domínio da estrutura de dados do modelo de fronteira. A fim de utilizar completamente estes *loops*, é necessária a apresentação de um operador criado especialmente para se adaptar à criação deles.

O KEMR separa um loop em dois novos através da remoção de uma aresta que aparece duas vezes dentro dele, como mostrado na Figura 9(a). Então, o KEMR divide uma curva de fronteira de uma face em duas curvas de fronteira. Os casos especiais em que um ou ambos os *loops* resultantes estão vazios, também estão inclusos no KEMR, como mostrado na Figura 9(b) e (c).

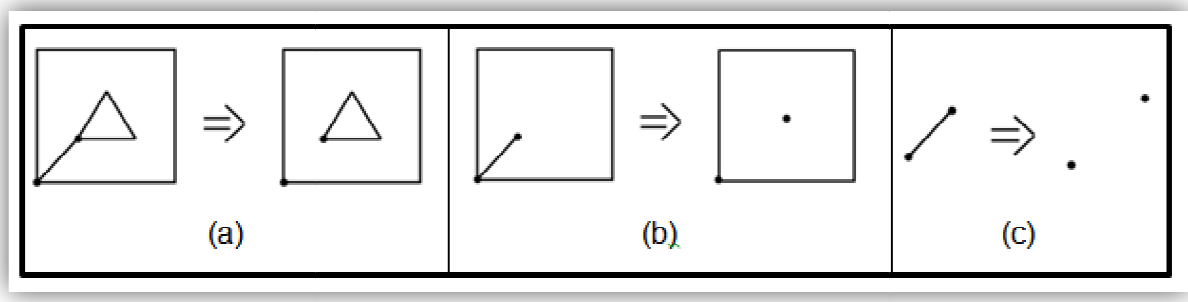


Figura 9 - Operador de Euler KEMR.

2.4 Estrutura de Dados *Half-Edge*

Quando uma estrutura de dados é projetada para armazenar topologias gráficas, ela deve obedecer a três diferentes aspectos:

• suficiência: a estrutura deve ser capaz de criar e recriar todas as relações topológicas de adjacências dos elementos geométricos de um modelo qualquer.

• velocidade: diz respeito à capacidade de estruturas de dados em permitir a manipulação rápida e eficiente de um modelo.

• armazenamento: as implementações devem apresentar o mínimo de informações possíveis, reduzindo ou eliminando o armazenamento de informações redundantes.

Quanto maior for o volume de informações armazenadas maior é a quantidade de memória requerida. Nesse contexto, a estrutura de dados topológica é utilizada para melhorar a relação processamento x armazenamento, ou seja, ela possibilita um melhor aproveitamento da memória do sistema, agindo de forma a reduzir o excesso de informações desnecessárias.

Para muitas aplicações, é necessário que se tenha uma estrutura de dados capaz de representar modelos não-intuitivos, para que se possa armazenar todos os

passos intermediários de uma sequência de operações utilizadas na descrição de um modelo. Na verdade, se modelos não-intuitivos não puderem ser representados, as operações de manipulação de objetos presentes num sistema de modelagem seriam de uso bastante limitado. A própria criação de novas entidades como polígonos, arestas e vértices dentro de um modelo pré-existente pode levar a casos especiais que precisam ser representados.

Uma estrutura de dados completa também precisa de que sejam armazenadas as informações geométricas do modelo, que podem ser somente as coordenadas dos vértices no caso de modelos poliedrais, ou equações de curvas e superfícies no caso de modelos com superfícies curvas.

A estrutura de dados *Half-Edge* (Mäntylä, 1988) é uma variação da estrutura *Winged-Edge* apresentada anteriormente. Ela é implementada segundo uma hierarquia de cinco níveis topológicos: sólido, face, *loop*, *Half-Edge* e vértice. No presente trabalho acrescentou-se a entidade topológica aresta a essa hierarquia, já que o sólido também se comunica com a aresta. A seguir são apresentadas as definições e características destes elementos para a estrutura de dados *Half-Edge* (Mäntylä, 1988):

• **Sólido** – O nó sólido constitui a raiz de toda a estrutura de dados da *Half-Edge*. A partir do ponteiro [3] para um sólido, pode-se acessar quaisquer elementos da estrutura topológica, como faces, arestas e vértices, através de listas duplamente encadeadas[4]. Todos os sólidos também são agrupados por uma lista duplamente

[3] Em linguagens de programação como C++, o ponteiro é um tipo especial de dado que pode armazenar o endereço de memória, sendo utilizado para conectar os nós de uma lista.

[4] Uma lista encadeada é uma representação de uma sequência de objetos na memória do computador. Cada elemento da sequência é armazenado em uma célula da lista: o primeiro na primeira célula, o segundo na segunda célula e assim por diante. Na lista duplamente encadeada, cada célula possui referências tanto para próxima célula quanto para a anterior.

encadeada, ou seja, cada sólido possui ponteiros para o sólido anterior e o sólido posterior.

• **Face** – O nó face representa uma face planar dos objetos representados pela estrutura de dados *Half-Edge*. Numa implementação mais completa e abrangente desta representação, as faces podem conter múltiplas fronteiras, de forma que cada face é associada a uma lista de *loops*, cada um representando uma fronteira da face. Como todas as faces são planares, um dos *loops* pode ser chamado de fronteira externa, enquanto os outros representam os buracos de uma face. Isto pode ser feito através de dois ponteiros, um dos quais aponta para o *loop* externo e outro que aponta para o primeiro *loop* na lista duplamente encadeada que engloba todos os *loops* da face.

• **Loop** – o nó *loop*, como exposto acima, representa uma fronteira conexa de uma face. Possui um ponteiro para a face que o contém, um ponteiro para uma das semi-arestas que formam a sua fronteira e ponteiros para os *loops* anterior e posterior daquela face.

• ***Half-Edge* (semi-aresta)** – O nó *Half-Edge* descreve uma linha que forma um *loop*. Consiste num ponteiro para o *loop* que o contém e um ponteiro para o vértice inicial da linha na direção do *loop*. Possui também ponteiros para as H*alf-Edges* anterior e posterior daquele *loop*, formando uma lista duplamente encadeada de *half-edges* de um loop. Desta forma, o vértice final de uma linha é tido como vértice inicial da próxima *Half-Edge*.

• **Vértice** – O nó vértice contém um vetor de 4 números reais que correspondem às coordenadas homogêneas de um ponto no espaço Euclidiano tridimensional. Há

também dois ponteiros para os vértices anterior e posterior formando uma lista duplamente encadeada dos vértices de um sólido.

A Figura 10 fornece uma visão geral da estrutura de dados *Half-Edge*. Para que esta estrutura esteja completa, no entanto, é necessário introduzir mais um nó, para que as informações topológicas formem um conjunto unificado e não- ambíguo na representação dos objetos. O conceito de semi-aresta (*Half-Edge*) já está bem definido, mas é necessário que se crie um nó de aresta (*edge*) para que se possa avaliar as relações entre as faces de um sólido sem que para isso elas tenham que fazer referência a um mesmo nó de vértice. O nó aresta associa as duas semi-arestas entre si. Ela combina as duas metades de uma aresta para formar a aresta em si. Possui ponteiros para as duas semi-arestas (esquerda e direita, por assim dizer). Há também uma lista duplamente encadeada de arestas, com ponteiros para a aresta anterior e posterior.

Em relação aos elementos anteriormente citados, deve-se acrescentar um ponteiro na estrutura da *Half-Edge* para a aresta que a contém e um ponteiro na estrutura de vértice para uma das *Half-Edges* que emana do mesmo. A Figura 11 mostra um esquema da relação entre as arestas e semi-arestas.

O caso especial de um *loop* vazio (constituído por somente um vértice, mas nenhuma aresta) pode ser representado. Para isto, na estrutura de dados Half-Edge o ponteiro para vértice aponta para este vértice único e o ponteiro para aresta é nulo. Os ponteiros para as semi-arestas anterior e posterior apontam para a própria semi-aresta em questão.

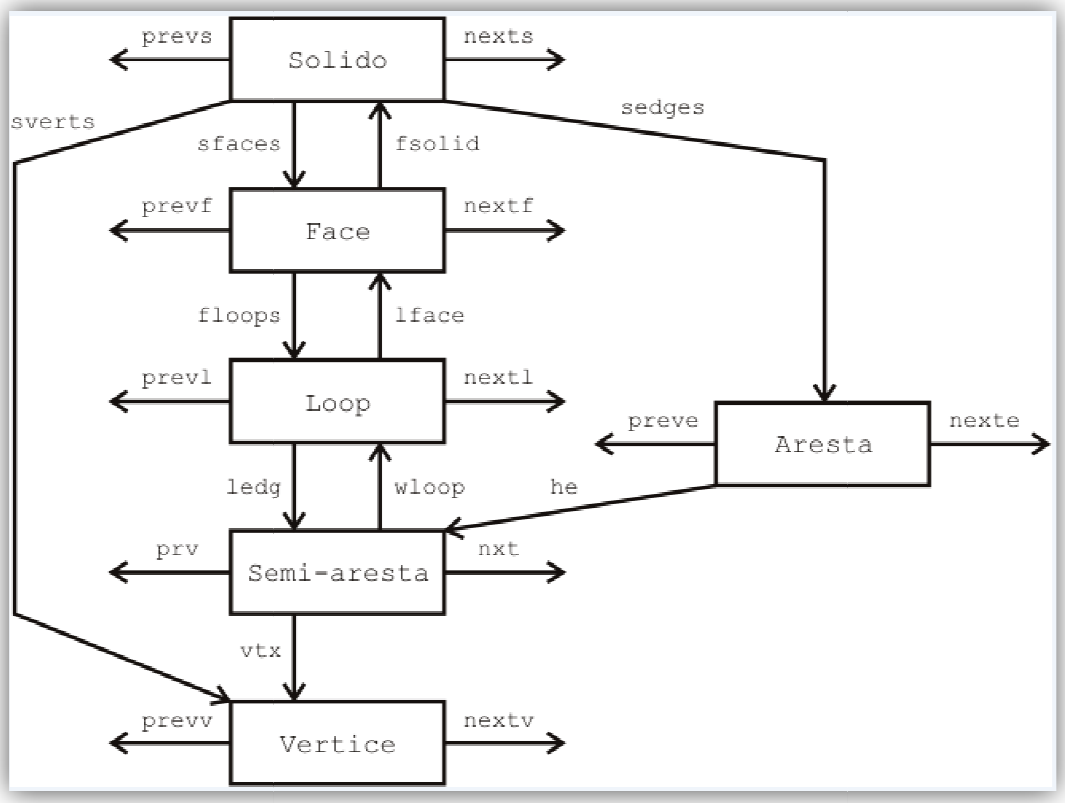


Figura 10 - Estrutura de dados *Half-Edge*.

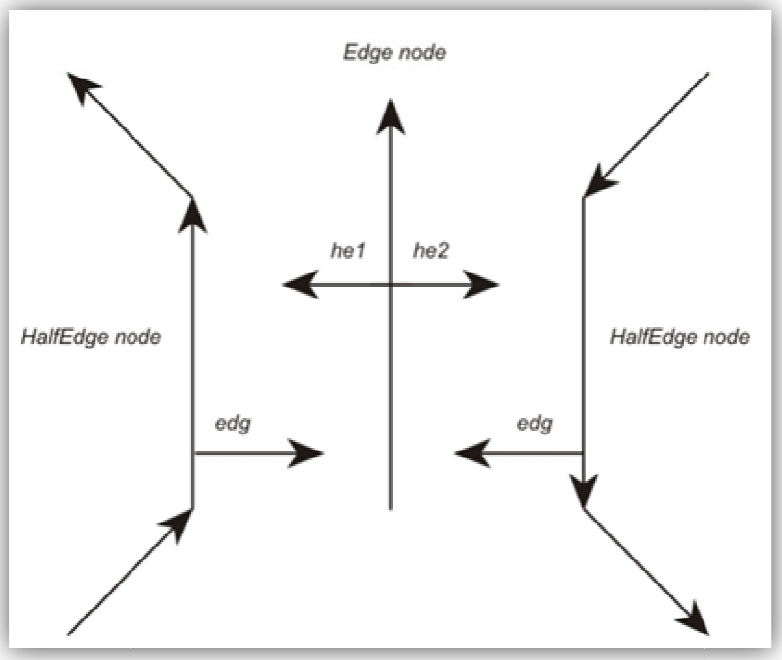


Figura 11 - Relação entre a aresta e semi-aresta na estrutura de dados Half-Edge.

3. FUNDAMENTOS DE ORIENTAÇÃO A OBJETOS E UML

Com o advento do uso da computação, diversas atividades relacionadas à engenharia foram facilitadas através do uso de *softwares*, que possibilitaram maior velocidade na análise e solução de problemas. Porém, com o avanço das técnicas de engenharia, a complexidade destes sistemas cresceu exponencialmente. Em decorrência disso, mostrou-se necessária uma melhor organização e padronização dos métodos de criação de tais *softwares*. Nesse contexto, surge o conceito de modelagem de sistemas.

A modelagem de sistemas é o melhor método para visualizar o projeto do sistema e certificar-se de que os requisitos estão sendo atendidos antes de iniciar a fase de implementação codificação. Os modelos são muito úteis para fazermos uma boa documentação, de forma a implementarmos um sistema robusto, flexível e, principalmente, manutenível. A criação de modelos também possibilita a documentação das decisões tomadas antes da codificação, além de servir como um guia para a construção do sistema.

Segundo Booch (2005), é possível representar um modelo através de uma perspectiva orientada a objetos. Nessa abordagem, o principal bloco de construção de todo sistema é o objeto ou a classe (conceitos de classes e objetos serão vistos na próxima seção). A aplicação da abordagem orientada a objetos em um determinado sistema permite que sejam inseridas ao mesmo inúmeras características positivas, tais como: padronização, extensibilidade, documentação (ter fácil compreensão, ser eficaz, satisfazer o usuário, ser facilmente relembrável), entre outras.

A abordagem orientada a objetos é constituída por basicamente três fases. A primeira fase, denominada Análise Orientada a Objetos (OOA) é a fase onde serão identificados os primeiros objetos que constituem o sistema. Já a segunda fase, conhecida como Projeto Orientado a Objetos (OOD) é a fase onde serão identificadas as relações entre os objetos do sistema. Finalmente a terceira fase, conhecida como Programação Orientada a Objetos (OOP), é a fase onde serão realizadas as implementações propriamente ditas do sistema.

Para entender a arquitetura de um sistema orientado a objetos é imprescindível ter diversas visões necessárias ao desenvolvimento e implantação desses sistemas. Essas visões são sugeridas pela UML, uma linguagem gráfica para visualizar, especificar, construir e documentar os artefatos de um sistema computacional orientado a objetos. De acordo com a UML, deve-se ter uma visão de casos de uso, expondo as exigências do sistema; uma visão de projeto, capturando o vocabulário do espaço do problema e do espaço da solução; uma visão do processo, modelando a distribuição dos processos e linhas do sistema; ima visão de implementação, dirigindo-se à realização física do sistema; e uma visão de distribuição, focando na edição da engenharia de sistema. Cada uma dessas visões pode ter aspectos estruturais, assim como comportamentais. Juntas, essas visões representam as plantas dos sistemas computacionais [Booch, 2005]. A Figura 12 mostra uma síntese dessas visões.

A utilização da UML em arquiteturas orientadas a objetos traz como vantagens:

- ✓ Desenvolvimento de programas de forma rápida, eficiente e efetiva;
- ✓ Revela a estrutura desejada e o comportamento do sistema;

- ✓ Permite a visualização e controle da arquitetura do sistema;
- ✓ Melhor entendimento do sistema que está sendo construído e gerenciamento de riscos.

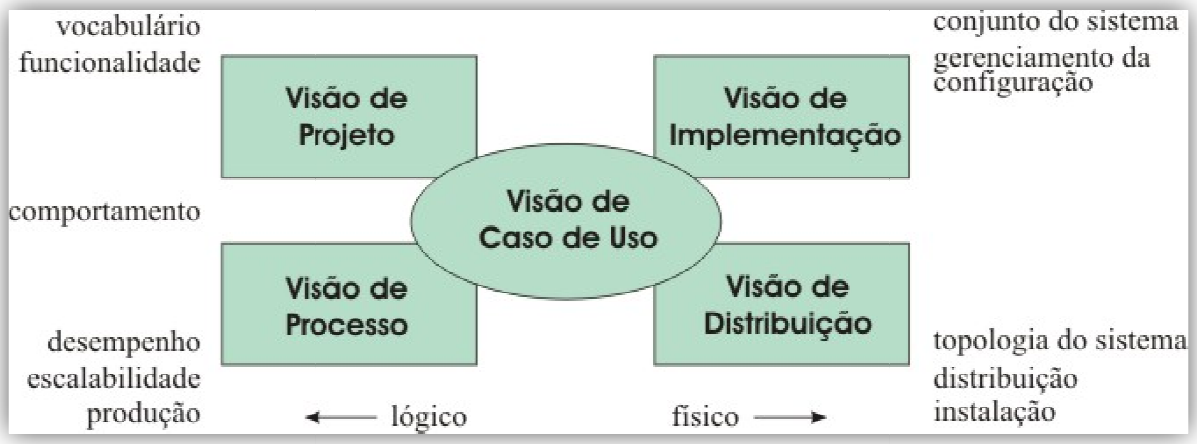


Figura 12 - Visões da arquitetura de um sistema orientado a objetos.

Para um facilitar o entendimento do que será apresentado nesta seção, serão introduzidos alguns conceitos fundamentais da Programação Orientada a Objetos.

3.1 Elementos

- **Objetos**

Segundo Booch (2005), um objeto pode ser definido como uma entidade, ou seja, como qualquer coisa que exista. Tais entidades se utilizam das abstrações da realidade, ou, em outras palavras, os objetos do mundo real são analisados considerando os aspectos relevantes para problema analisado e desconsiderando os aspectos irrelevantes.

Um objeto possui três características:

Identidade: cada objeto possui um nome que o identifica ou distingue dos demais objetos e que permite que outros objetos o reconheçam e o enderecem.

Métodos: cada objeto pode possuir um conjunto de habilidades de processamento que juntas descrevem seu comportamento. Os métodos de um objeto são as funções que este desempenha, ou seja, seu comportamento. Dessa forma, quando um objeto executa um serviço, através do envio e recebimento de mensagens, ele está ativando um de seus métodos.

Atributos: são propriedades e informações características dos objetos.

Como exemplo, pode-se considerar uma disciplina do setor de Estruturas como um objeto. A identidade deste objeto poderia ser “Resistência dos Materiais”. Seus atributos poderiam ser “Carga Horária”, “Pré-requisitos”, “Ementa”, “Alunos Matriculados” e “Notas”. Seus métodos poderiam ser “Inscrever um Novo Aluno”, “Calcular a Média da Turma” e “Registrar Notas dos Alunos”.

Na modelagem de um objeto, a ideia que deve se ter em mente é a da reutilização. O objeto deve ser definido da forma mais completa possível, analisando todos os atributos que o definem e todos os serviços que poderão ser prestados por ele. Dessa forma, este objeto pode ser reutilizado em outros sistemas, sem que seja necessária uma nova modelagem.

Quando se trata de um projeto orientado a objetos, a primeira coisa que deve ser definida é o conjunto de objetos (entidades) envolvidos no sistema, através da especificação de seus comportamentos e relações. Partindo-se dos objetos e relações, definem-se quais serviços cada objeto deverá prestar e como eles devem se comunicar para realizar as funções do sistema.

Um sistema orientado a objetos é considerado uma entidade dinâmica, ou seja, a todo o momento os objetos estão participando das ações que vão sendo realizadas.

- **Classes**

As classes são consideradas elementos fundamentais na composição de sistemas orientados a objetos. Uma classe é uma descrição do conjunto de atributos e métodos que definem um grupo de objetos (Booch, 2005).

São representadas graficamente por um retângulo dividido em três compartimentos. O primeiro deles contém o nome da classe, que é seu identificador e deve ser escolhido de forma a abranger conceitualmente todos os objetos correspondentes a esta classe. Os segundo e terceiro compartimentos compreendem respectivamente os atributos e métodos da classe em questão, como pode ser vista na figura. Esses atributos e métodos devem ser comuns a todos os objetos, instâncias dessa classe.

Uma classe obrigatoriamente terá que possuir um nome, mas não necessariamente deverá possuir atributos ou operações. As classes e os objetos são as principais primitivas ou elementos de composição de *softwares* orientados a objetos. Um sistema orientado a objetos é composto por classes e um conjunto de objetos que colaboram ou interagem para realização dos serviços oferecidos pelo sistema.

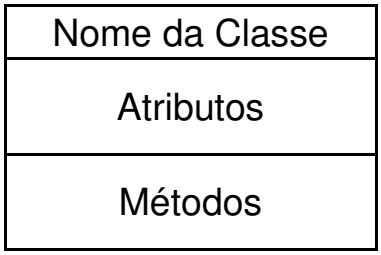


Figura 13 - Características de uma classe.

3.2 Relacionamentos

Geralmente, as classes não estão sós e se relacionam entre si. Dentre os relacionamentos reconhecidos pela UML, a dependência, a agregação e a associação fizeram parte desse trabalho. Esta seção tem o objetivo de explicar como são esses relacionamentos, de acordo com Booch (2005).

- **Dependência**

São relacionamentos de utilização no qual uma mudança na especificação de um elemento pode alterar a especificação do elemento dependente. A dependência entre classes indica que os objetos de uma classe usam serviços dos objetos de outra classe. Esse relacionamento é representado por uma seta tracejada, apontando o item do qual o outro depende, como observado na Figura 14.

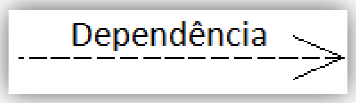


Figura 14 - Representação do relacionamento de dependência.

- **Associação**

Em análise e projeto orientado a objetos, uma associação representa um relacionamento estrutural entre instâncias e especifica que objetos de uma classe estão ligados a objetos de outra classe. Por exemplo, vários alunos podem estar associados a um único professor e um único aluno pode estar associado a vários professores. Neste caso, não existe um relacionamento de posse entre esses objetos. Todos os objetos são independentes. Um aluno pode existir sem a necessidade de um professor, da mesma forma que é possível existir um professor

sem a necessidade da existência de um aluno. Esse relacionamento é representado por uma reta contínua, como sugerido na Figura 15.

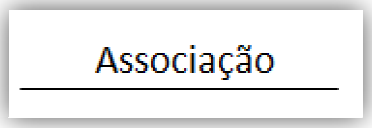


Figura 15 - Representação do relacionamento de associação.

- **Agregação**

É uma associação que reflete a construção física ou a posse lógica. Relacionamentos de agregação são casos particulares dos relacionamentos de associação, e só é necessário distingui-los quando for conveniente enfatizar o caráter “todo-parte” do relacionamento, no qual uma classe representa um item maior (o “todo”), formado por itens menores (as “partes”). Nesse relacionamento, um objeto do tipo todo contém objetos das partes. A agregação, na verdade, é apenas um tipo especial de associação, especificada utilizando uma associação simples com um diamante aberto na extremidade do todo, conforme mostrado na Figura 16.

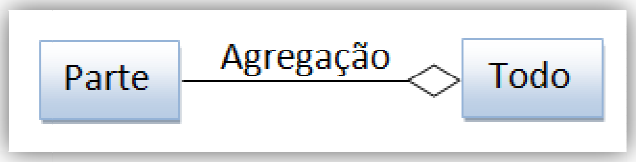


Figura 16 - Representação do relacionamento de agregação.

3.3 Diagramas da UML

De acordo com Booch (2005), um diagrama é a apresentação gráfica de um conjunto de elementos, geralmente representados como gráficos de vértices (itens) e arcos (relacionamentos). Nesse sentido, um diagrama constitui a projeção de um determinado sistema. Uma vez que nenhum sistema complexo pode ser compreendido em sua totalidade a partir de uma perspectiva, a UML define alguns diagramas que permitem dirigir o foco para aspectos diferentes do sistema de maneira independente.

Bons diagramas facilitam a compreensão do sistema que se está desenvolvendo. A UML apresenta uma variedade de diagramas, dos quais serão utilizados nesse projeto os que se seguem abaixo:

1) **Diagrama de Casos de Uso**:

Os diagramas de casos de uso assumem papel importante na modelagem comportamental de um sistema por apresentar uma visão externa de como esses elementos podem ser utilizados no contexto.

Os elementos que compõem um diagrama de casos de uso é um conjunto de casos de uso[5] e atores[6] (um tipo especial de classe) e seus relacionamentos. Esse tipo de diagrama abrange a visão estática de casos de uso do sistema. Tais diagramas são importantes principalmente para a organização e modelagem de comportamentos do sistema. O diagrama de casos de uso é um instrumento

[5] Casos de usos – é um instrumento para descrição das intenções ou requisitos para um sistema computacional. [Booch, 2005].
[6] Atores são representações de entidades externas, mas que interagem com o sistema durante sua execução, através de troca de mensagens. O usuário do sistema, por exemplo, é um ator.

eficiente para a determinação e documentação dos serviços a serem desempenhados pelo sistema. Ele é também um bom meio para a comunicação com os clientes no processo de definição dos requisitos do sistema.

2) Diagrama de Robustez*:

Este diagrama, apesar de não estar declarado entre os diagramas da UML, foi proposto por um de seus idealizadores, Ivar Jacobson.

Para ir dos casos de uso para o projeto detalhado (e, em seguida, para o código), é necessário vincular os casos de uso aos objetos. A ideia básica da análise de robustez é captar, para cada caso de uso, os principais objetos e respectivas relações de comunicação estabelecidas entre os mesmos. Em outras palavras, pode-se usar esse diagrama para assegurar que os casos de uso são suficientemente robustos e consistentes para representar os requisitos do sistema.

Costuma-se categorizar os objetos de um sistema de acordo com o tipo de responsabilidade atribuída a ele. Ivar Jacobson propôs que os objetos fossem divididos em três categorias:

Objetos de Entidade → um objeto de entidade é um repositório para alguma informação manipulada pelo sistema. Esses objetos modelam informações persistentes, sendo tipicamente independentes da aplicação. Os objetos de entidade se comunicam com o exterior por intermédio de outros objetos. Dentro desse contexto, as principais responsabilidades desse tipo de objeto são:

• Informar valores de seus atributos a objetos de controle.

• Criar e destruir objetos parte (considerando que o objeto de entidade seja um objeto todo de uma agregação).

Objetos de Fronteira → esses objetos traduzem os eventos gerados por um ator em eventos relevantes ao sistema. Os objetos de fronteira servem para fazer a comunicação entre o sistema e o ambiente exterior, ou seja, os atores. Nesse contexto, esses objetos são responsáveis por:

• Notificar aos objetos de controle os eventos gerados externamente ao sistema.

• Notificar os atores do resultado de interações entre os objetos internos.

Objetos de Controle → os objetos de controle servem como "ponte de comunicação" entre os objetos de fronteira e objetos de entidade. Esses objetos tem o poder de decidir o que o sistema deve fazer diante da ocorrência um evento externo relevante. Basicamente, os objetos de controle agem como "gerentes" (coordenadores, controladores) dos outros objetos para a realização de um ou mais casos de uso. Dentro desse contexto, as principais responsabilidades desse tipo de objeto são:

• Realizar monitorações, a fim de responder a eventos externos ao sistema (gerados por objetos de fronteira).

• Coordenar a realização de um caso de uso através do envio de mensagens a objetos de fronteira e objetos de entidade.

• Coordenar a criação de associações entre objetos de entidade.

A Figura 17 ilustra como esses objetos devem ser representados.

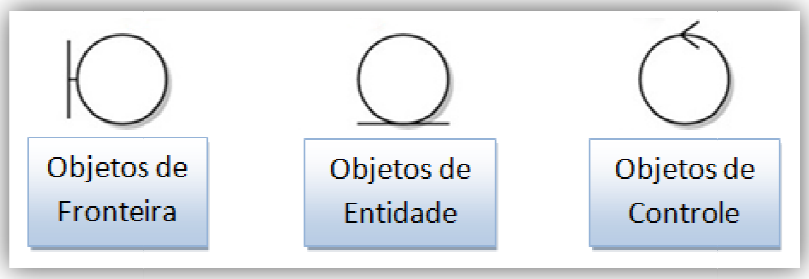


Figura 17 - Representação dos objetos de Fronteira, Entidade e Controle.

3) Diagrama de Classes: é constituído por um conjunto de classes, interfaces e colaborações, bem como seus relacionamentos. Esse tipo de diagrama é encontrado, principalmente, em sistemas de modelagem orientados a objeto e retrata uma visão estática da estrutura do sistema. Os diagramas de classe são importantes não só para a visualização, a especificação e a documentação de modelos estruturais, mas também para a construção de sistemas executáveis.

4) Diagrama de Sequências: é um tipo de diagrama de objetos, ou seja, ele contém como primitiva principal um conjunto de objetos de diferentes classes. O objetivo dos diagramas de sequência é descrever as comunicações necessárias entre objetos para a realização dos processos de um sistema. Os diagramas de sequência tem esse nome porque descrevem ao longo de uma linha de tempo a sequência de comunicações entre objetos. Também é visto como um diagrama de interação, como o próprio nome sugere, exibe uma interação, consistindo de um conjunto de objetos ou papéis, considerando também as mensagens que podem ser trocadas entre eles. O diagrama de interação compreende a visão dinâmica de um sistema. Um diagrama de sequências é um diagrama de interação cuja ênfase está na ordenação temporal das mensagens.

4. ARQUITETURA ORIENTADA A OBJETOS DA ESTRUTURA DE DADOS

A principal originalidade desse projeto refere-se ao tipo de estratégia adotada para implementação computacional da estrutura de dados topológica, que vai possibilitar a criação e manipulação de modelos bidimensionais. Para cumprir esse objetivo, projetou-se uma arquitetura orientada a objetos, aplicando os conceitos apresentados no capítulo 3, na implementação da estrutura de dados topológica que foi introduzida na seção 2.4.

Este quarto capítulo tem o objetivo de mostrar todas as etapas pelas quais teve-se que passar para desenvolver o modelador proposto nesse projeto, etapas essas que serão detalhadas nas seções a seguir.

4.1 Pré-requisitos do sistema

A primeira etapa do projeto foi identificar os requisitos necessários para que o modelador possa satisfazer os objetivos propostos. Tendo em mente que o programa a ser projetado é um programa que pode ser usado para criar e manipular modelos bidimensionais, ou seja, um modelador que gerencie uma coleção de entidades geométricas (pontos ou curvas), pode-se descrever os seguintes requisitos:

• Deve ser possível adicionar um novo ponto (isto é, um vértice, sua correspondente entidade topológica) ou uma nova curva (isto é, uma aresta), assim como manipular essas entidades de forma que seja preservada/mantida a consistência topológica do modelo.

• Deve ser possível adicionar uma curva (aresta) a partir de um vértice (ponto) já existente.

•Deve ser possível adicionar um ponto (vértice) a partir de uma curva (aresta) já existente.

• Deve ser possível, através da manipulação de pontos (vértices) e curvas (arestas), construir uma região fechada (face).

• Deve ser possível, através da combinação de regiões fechadas (faces), construir um modelo.

• Deve ser possível salvar o modelo em uma extensão que mostras as informações topológicas, como as coordenadas dos pontos (vértices).

• Deve ser possível fazer manipulações no modelo geométrico. (Ex: editar, deletar etc.)

• Deve ser possível salvar, abrir um modelo existente, criar um novo, gerenciar mais de um modelo.

• Deve ser possível representar furos no modelo.

• Deve ser possível tratar multi-regiões.

Para utilizar a estrutura de dados topológica desenvolvida, um escopo da interface gráfica (GUI) foi idealizado para auxiliar na identificação de quais ferramentas o modelador deveria oferecer, como pode ser observado na Figura 18. Dessa forma, o modelador deverá apresentar uma barra de *status*, que mostrará as coordenadas de um determinado ponto; uma barra de ferramentas, que mostrará

ícones através dos quais será possível criar modelos geométricos para fazer simulações computacionais, salvá-los, editá-los, entre outras funcionalidades.

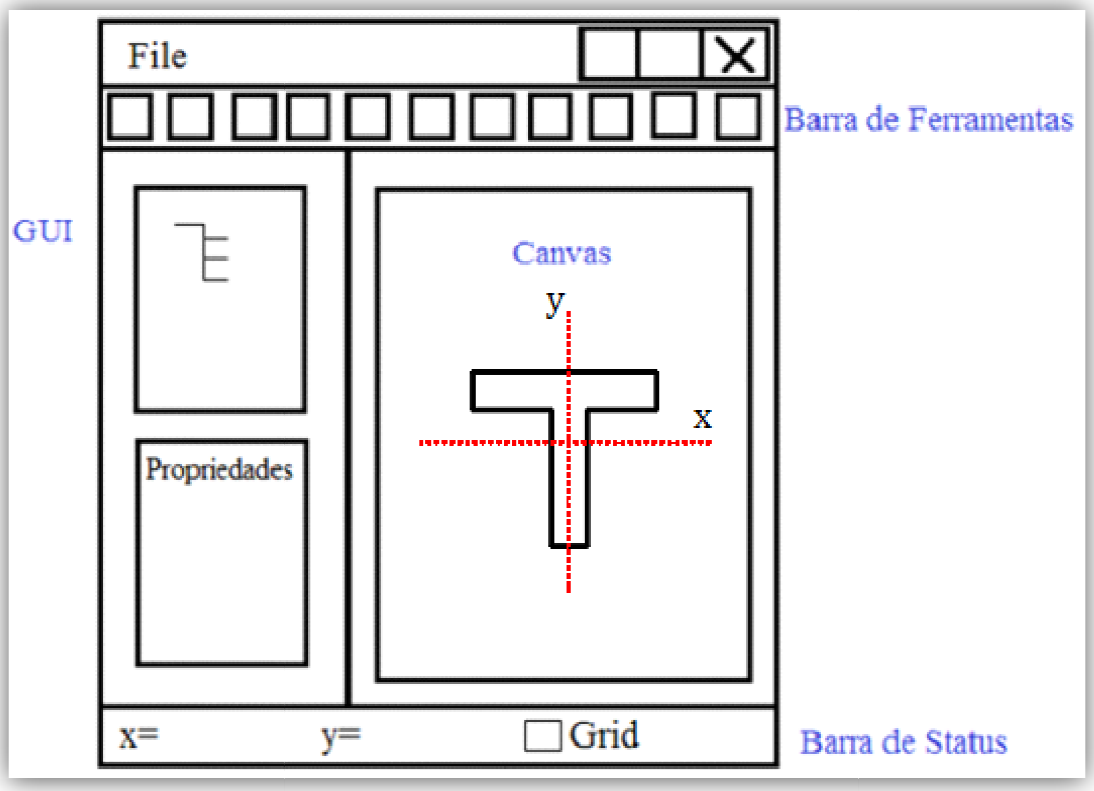


Figura 18 - Escopo da interface gráfica do modelador.

4.2 Diagrama de Casos de Uso

A segunda etapa consiste da construção dos Casos de Uso, que é uma das fases iniciais de um projeto de *software*, pois envolve a determinação dos usos que o sistema terá, ou seja, do que ele deverá fornecer como serviços. O Diagrama de Casos de Uso é um instrumento eficiente para a determinação e documentação dos serviços a serem desempenhados pelo sistema.

Para o modelador proposto no trabalho, o Diagrama de Casos de Uso é o mostrado na Figura 19.

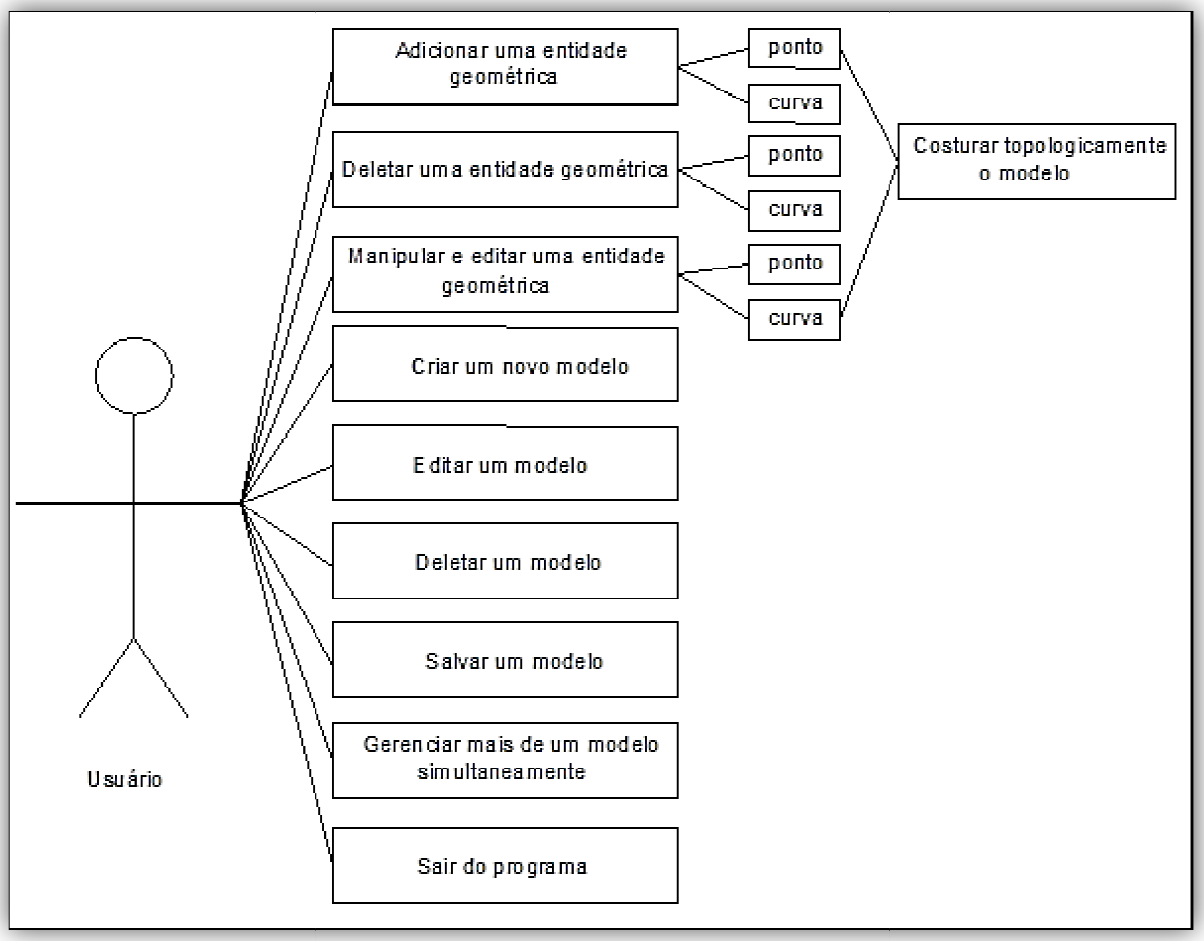


Figura 19 - Diagrama de Casos de Uso

Por exemplo, o caso de uso adicionar uma entidade geométrica é iniciado quando o usuário clica nos ícones "*insert point*" ou "*insert curve*" da interface gráfica. A estrutura de dados detecta que um ponto foi inserido quando o usuário clica no ícone  (o cursor do mouse deve aparecer como uma cruz) e depois clica no canvas. Para inserir uma curva, basta clicar no ícone  e, no canvas, clicar no ponto inicial seguido do ponto final.

4.3 Diagrama de Robustez

Na terceira etapa foi realizada uma leitura inicial dos casos de uso, a partir dos quais foi possível definir farão parte do sistema:

• Um objeto de entidade que representa o atual modelo no qual se está trabalhando (*HEDModel*).

• Um número arbitrário de objetos de entidade, cada uma representando uma das entidades geométricas é topológicas que estão no atual modelo (*GeometricEntity* e *TopologicalEntitiy*).

• Um objeto de fronteira que representa a interface (gráfica ou não) entre o sistema do modelador e o usuário (*GUI* e *HEDUI*).

• Um objeto de fronteira que representa a interface entre o sistema do modelador e o sistema de arquivos no disco (*Output*).

• Um objeto de controle que realiza os casos de uso em resposta aos comandos do usuário, seja na *GUI* ou na *HEDUI* (*HEDController*).

Para o usuário adicionar um ponto, por exemplo, ele informa suas coordenadas através do "*HEDUI*" ou da "*GUI*". Essa informação é transmitida para o "*HEDController*", que envia uma mensagem para o "*GeomChecker*" verificar se aquele ponto já está na estrutura de dados. Para fazer essa verificação, o GeomChecker, envia uma mensagem ao "*HEDModel*", que procura em seu banco de dados a existência de uma "*GeometricEntity*" e sua correspondente "*TopologicalEntity*". Depois da verificação, o "*HEDController*" recebe uma resposta do "*HEDModel*", e decide qual operador de Euler será utilizado, enviando essa

informação para o "*EulerOperator*". Após isso, o "*HEDController*" envia uma mensagem ao "*Output*", com o resultado da inserção do ponto para o usuário.

Para o modelador proposto no trabalho, o Diagrama de Robustez é o mostrado na Figura 20.

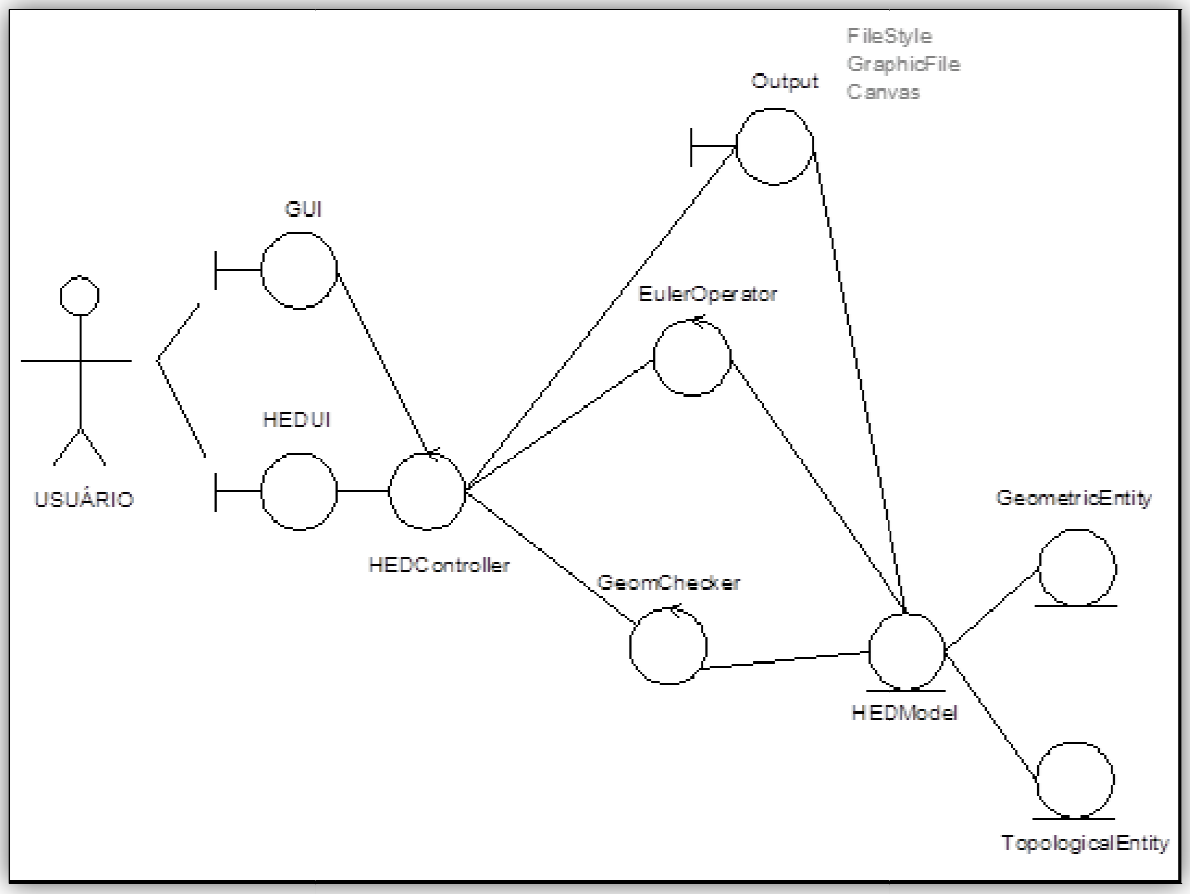


Figura 20 - Diagrama de Robustez

4.4 Diagrama de Sequências

A quarta etapa consiste da elaboração do Diagrama de Sequências. Cada um dos casos de uso descobertos na análise do sistema será realizado por uma sequência de operações que envolvem os vários objetos que compõem o sistema.

Como exemplo, para adicionar um ponto no modelo, o Diagrama de Sequências foi desenvolvido e será mostrado no Anexo 1, para melhorar sua visualização, já que este ficou muito extenso.

4.5 Diagrama de Classes

A quinta etapa consiste da elaboração do Diagrama de Classes. Esse diagrama inclui as classes encontradas durante a análise, além de algumas adicionais descobertas durante o projeto.

Nesse projeto, as classes que surgiram durante o projeto foram provenientes da ramificação das classes "*GeometricEntity*" e "*TopologicalEntity*". São elas: "Curve", "Edge", "Face", "HalfEdge", "*Loop*", "*Point*", "*Solid*", "*Vertex*".

Para o modelador proposto no trabalho, o Diagrama de Classes é o apresentado na Figura 21.

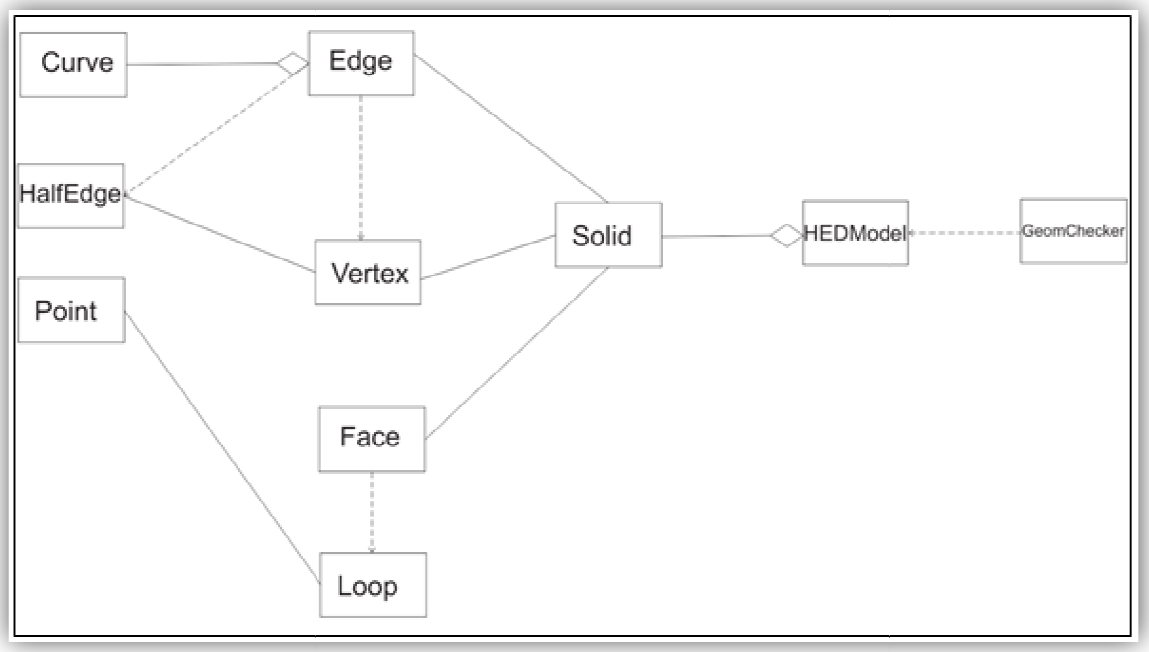


Figura 21 - Diagrama de Classes.

4.5.1 Diagrama de Classes Detalhadas

Esse diagrama apresenta as classes, detalhando seus atributos e seus métodos. Tais informações poderiam ter sido expressas no diagrama anterior, entretanto, para evitar o excesso de informações em um pequeno espaço, optou-se por omitir essas informações na seção anterior e especificá-las nesta seção.

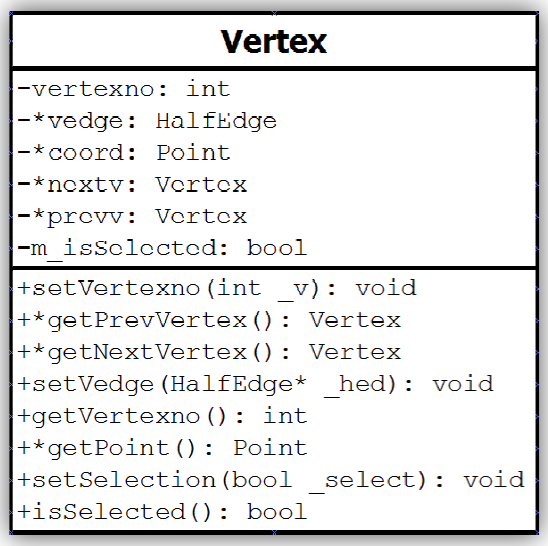


Figura 22 - Projeto de Classes Detalhadas - Vertex.

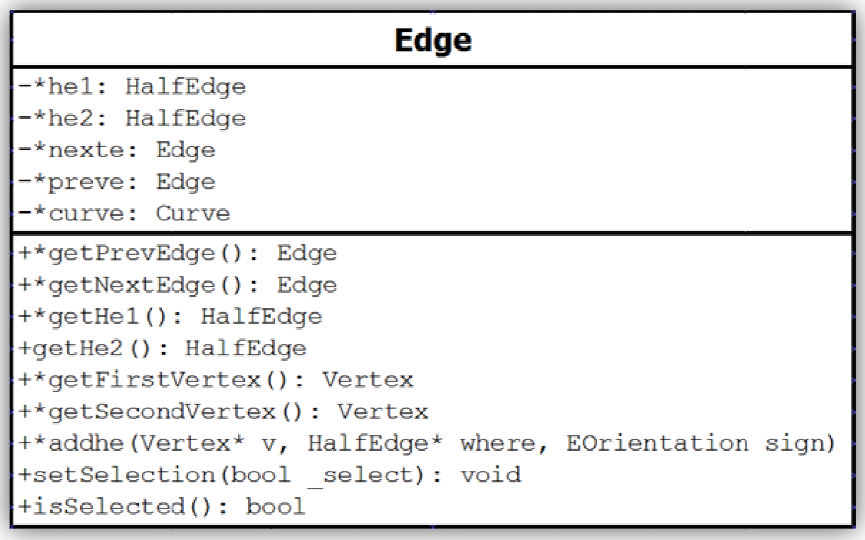


Figura 23 - Projeto de Classes Detalhadas - Edge.

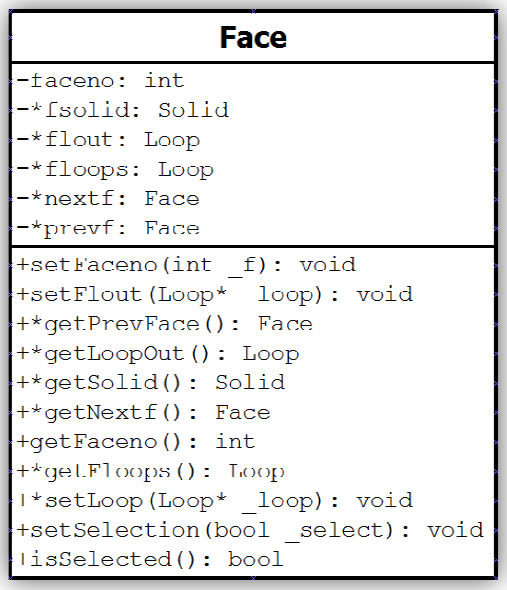


Figura 24 - Projeto de Classes Detalhadas - Face.

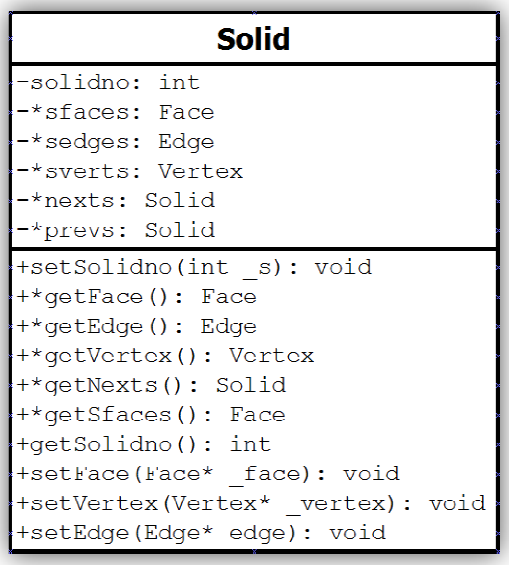


Figura 25 - Projeto de Classes Detalhadas - Solid.

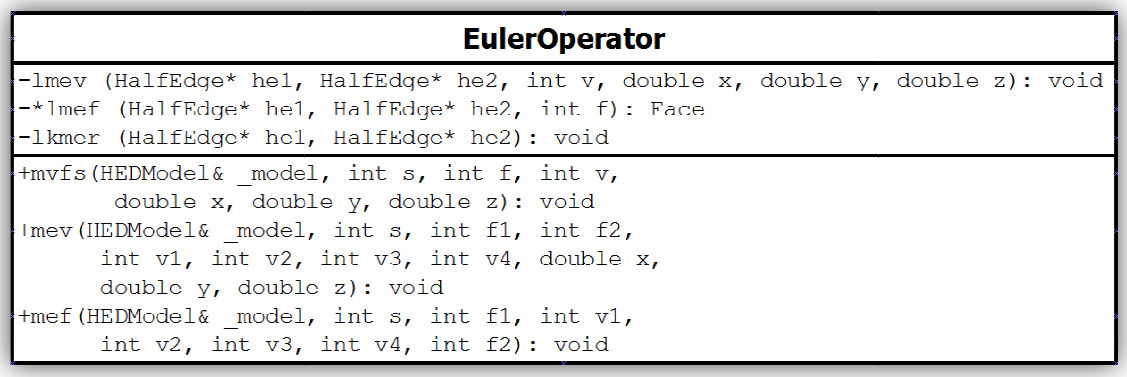


Figura 26 - Projeto de Classes Detalhadas - EulerOperator.

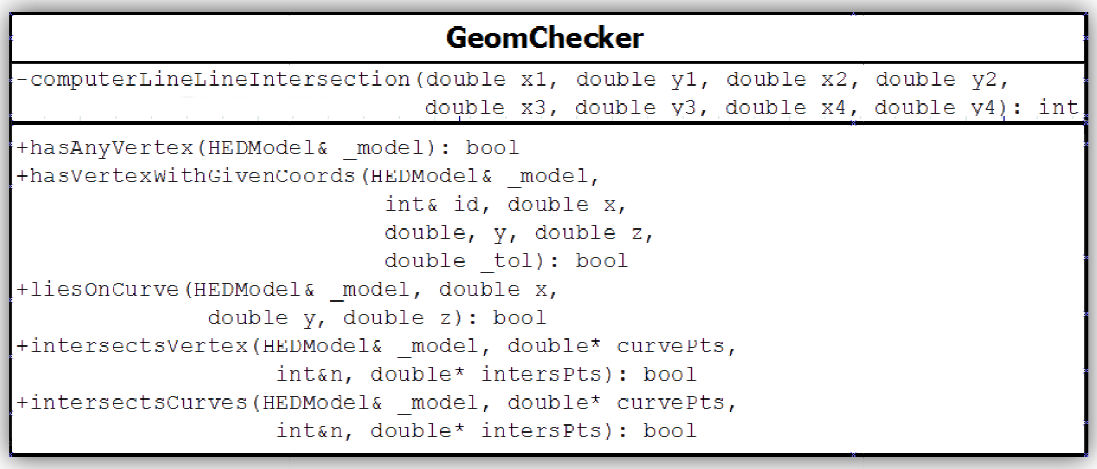


Figura 27 - Projeto de Classes Detalhadas - GeomChecker.

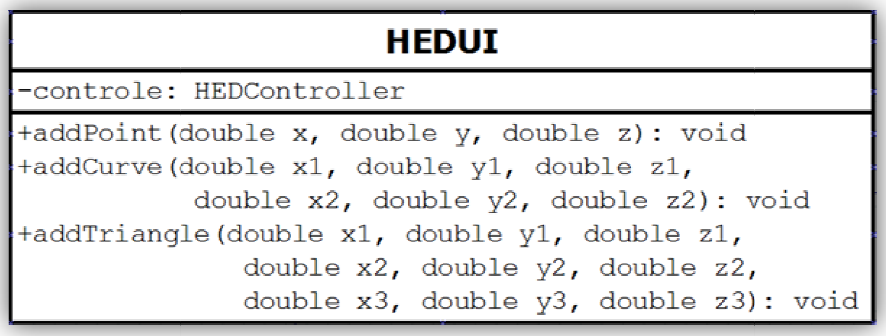


Figura 28 - Projeto de Classes Detalhadas - HEDUI.

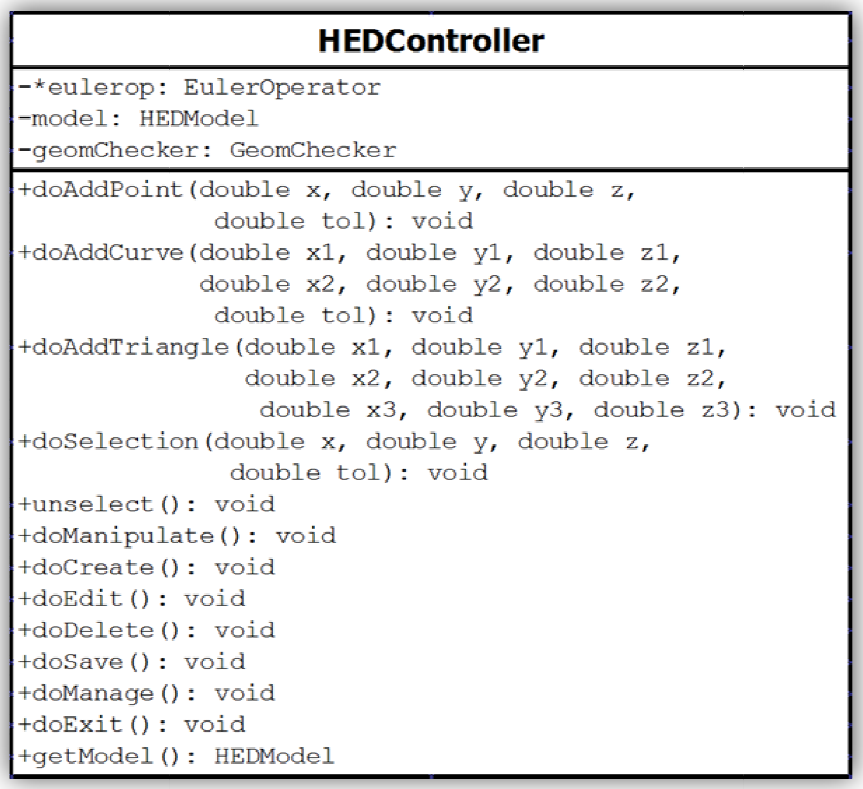


Figura 29 - Projeto de Classes Detalhadas - HEDController.

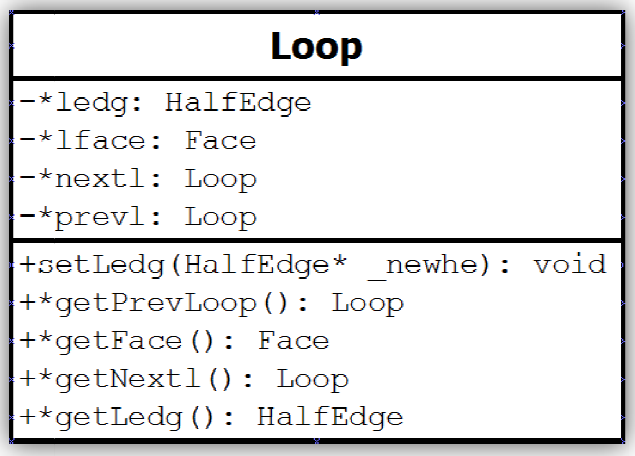


Figura 30 - Projeto de Classes Detalhadas - Loop.

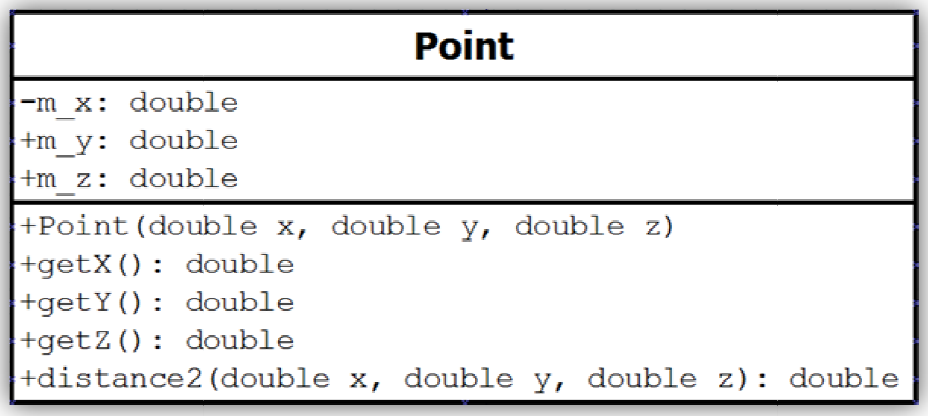


Figura 31 - Projeto de Classes Detalhadas - Point.

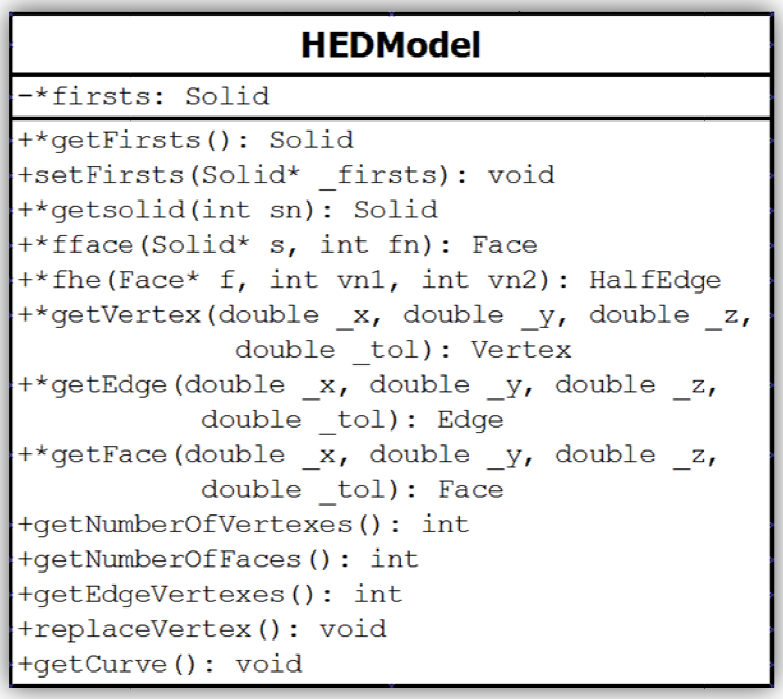


Figura 32 - Projeto de Classes Detalhadas - HEDModel.

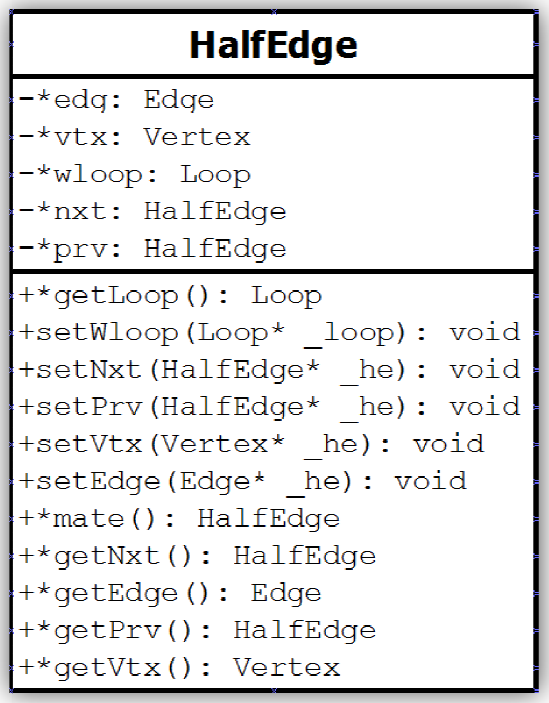


Figura 33 - Projeto de Classes Detalhadas - HalfEdge.

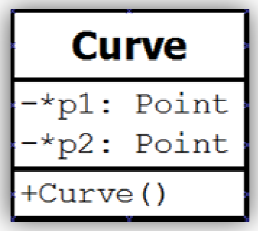


Figura 34 - Projeto de Classes Detalhadas - Curve.

5. IMPLEMENTAÇÃO

Este capítulo foca em trechos de códigos chaves que são importantes para compreensão da fase de programação da arquitetura desenvolvida no capítulo anterior da estrutura de dados topológica da Semi-Aresta. Uma abordagem descrita na linguagem C++, desenvolvido no Microsoft Visual Studio 2008, será utilizada para mostrar (partes chaves) os principais aspectos do escopo do programa.

Nesse projeto, somente os operadores de Euler MVFS, LMEV, LMEF, LKEMR, MEV e MEF foram implementados. Os operadores inversos serão implementados em projetos futuros.

O código de programação referente à implementação do operador de Euler MVFS encontra-se desenvolvido na seqüência.

5.1 Operadores de Euler de baixo nível

- **MVFS**

O primeiro operador de Euler necessário para inserirmos uma entidade topológica é o MVFS, que é o ponto de partida dos operadores. No código abaixo, o processo aloca cada um dos nós: *Solid*, *Face*, *Loop*, e *Vertex* com o nome de "*new*", e um *Half-Edge* com o nome de "*addhe*". Uma nova *Half-Edge* é inicializada corretamente para representar o *loop* "vazio" que consiste de um vértice, porém de nenhuma aresta.

As relações de adjacências entre as entidades são inseridas no segundo bloco de construção.

```
void EulerOperator::mvfs(HEDModel& _model, int s, int f, int v, double x,
double y, double z)
{
	Solid* newsolid = new Solid(_model.getFirsts());
	_model.setFirsts( newsolid );
	Point* point = new Point(x, y, z);
	Face* newface = new Face(newsolid);
	Vertex* newvertex = new Vertex(point,newsolid);
	HalfEdge* newhe = new HalfEdge();
	Loop* newloop = new Loop(newface);

	newsolid->setSolidno(s);
	newface->setFaceno(f);
	newvertex->setVertexno(v);
	newloop->setLedg( newhe );
	newface->setFlout( newloop );
	newhe->setWloop(newloop);
	newhe->setNxt(newhe);
	newhe->setPrv(newhe);
	newhe->setVtx(newvertex);
	newhe->setEdg(NULL);
}
```

- Implementação das relações de adjacências da *Half-Edge*

```
HalfEdge* Edge::addhe(Vertex* v, HalfEdge* where, EOrientation sign)
{
	HalfEdge* he = NULL;
	if( where->getEdg() == NULL )
	{
		he = where;
	}
	else
	{
		he = new HalfEdge();
		where->getPrv()->setNxt(he);
		he->setPrv(where->getPrv());
		where->setPrv(he);
		he->setNxt(where);
	}
	he->setEdg(this);
	he->setVtx(v);
	he->setWloop(where->getLoop());
	if(sign == PLUS)
		this->he1 = he;
	else
		this->he2 = he;

	return he;
}
```

LMEV

O LMEV é o operador de baixo nível responsável pela divisão de vértices.

No código do modelador, se os argumentos *he1* e *he2* do procedimento forem diferentes, o LMEV irá dividir o ciclo de *Half-Edges* do vértice *he1->vtx* em dois ciclos de forma que *he1* seja a primeira H*alf-Edge* em um ciclo, e *he2* é a outra. O identificador *vn*, as coordenadas x y z e o ciclo que começam em *he1* são atribuídos a um novo vértice. Caso *he1* seja igual a *he2*, uma aresta de "suporte" aparece duas vezes em um *loop* e é adicionada a frente de *he1*.

Os procedimentos de manipulações da H*alf-Edge* possibilitam que estas especificações sejam implementadas, sem nenhum código especial, para casos diferentes, de forma suficientemente boa. O processo primeiramente aloca e inicializa o novo vértice e os nós de arestas. Enquanto as *Half-Edges he1* e *he2* são atribuídas a um novo vértice, o *loop while* atualiza os ponteiros de seus vértices. Finalmente, novas *Half-Edges* são inseridas, e o ciclo de ponteiros dos vértices é atualizado. A nova aresta será orientada na direção do vértice antigo.

O código de programação referente à implementação do operador de Euler LMEV encontra-se desenvolvido na sequência.

```
void EulerOperator::lmev(HalfEdge* he1, HalfEdge* he2, int v, double x, y, z)
{
	Loop* loop = he1->getLoop();
	Face* face = loop->getFace();
	Solid* solid = face->getSolid();
	Edge* newedge = new Edge(solid);
	Point* point = new Point(x,y,z);
	Vertex* newvertex = new Vertex(point,solid);
	newvertex->setVertexno(v);
	HalfEdge* he = he1;
	while(he != he2)
	{
		he->setVtx(newvertex);
		he = he->mate()->getNxt();
	}
	newedge->addhe(he2->getVtx(), he1, MINUS);
	newedge->addhe(newvertex, he2, PLUS);
	newvertex->setVedge(he2->getPrv());
	he2->getVtx()->setVedge(he2);
}
```

LMEF

O LMEF é o operador de baixo nível responsável pela divisão de faces.

De maneira similar ao LMEV, se *he1* for diferente de *he2*, o LMEF divide o *loop* de *he1* e *he2* em dois *loops* com uma nova aresta de *he1->vtx* para *he2->vtx*. A H*alf-Edge he2* permanece no *loop* antigo, enquanto *he1* é movida para o novo *loop* que se torna a fronteira mais externa da nova face. No caso especial em que *he1* é igual a *he2*, uma face “circular” com apenas uma aresta é criada.

O código de programação referente à implementação do operador de Euler LMEF encontra-se desenvolvido na sequência.

```
Face* EulerOperator::lmef(HalfEdge* he1, HalfEdge* he2, int f)
{
	Loop* loop = he1->getLoop();
	Face* face = loop->getFace();
	Solid* solid = face->getSolid();
	Face* newface = new Face(solid);
	Edge* newedge = new Edge(solid);
	Loop* newloop = new Loop(newface);

	newface->setFaceno(f);
	newface->setFlout(loop);


	HalfEdge* he = he1;
	while(he != he2)
	{
		he->setWloop(newloop);
		he = he->getNxt();
	}

	HalfEdge* nhe1 = newedge->addhe(he2->getVtx(), he1, MINUS);
	HalfEdge* nhe2 = newedge->addhe(he1->getVtx(), he2, PLUS);

	nhe1->getPrv()->setNxt(nhe2);
	nhe2->getPrv()->setNxt(nhe1);
	HalfEdge* temp = nhe1->getPrv();
	nhe1->setPrv(nhe2->getPrv());
	nhe2->setPrv(temp);

	newloop->setLedg(nhe1);
	he2->getLoop()->setLedg(nhe2);

	return newface;
}
```

LKEMR

O último exemplo de código é o LKEMR, que é o operador de baixo nível responsável pela divisão de um *loop* em dois componentes.

Após um novo nó de *loop* ter sido alocado, as duas H*alf-Edges h1* e *h2*, correspondentes à aresta que será removida, são manipuladas de forma a deixar ambas como componentes distintos, fazendo com que aquela que está associada *h2* se torne um novo *loop*.

O código de programação referente à implementação do operador de Euler LKEMR encontra-se desenvolvido na sequência.

```
void EulerOperator::lkemr(HalfEdge* h1, HalfEdge* h2)
{

	register HalfEdge* h3,*h4;
	Loop* ol= h1->getLoop();
	Loop* nl = new Loop(ol->getFace());

	h3 = h1->getNxt();
	h1->setNxt(h2->getNxt());
	h2->getNxt()->setPrv(h1);
	h2->setNxt(h3);
	h3->setPrv(h2);

	h4 = h2;
	do
	{
		h4->setWloop(nl);
	}
	while((h4 = h4->getNxt()) != h2);
}
```

4.2 Operadores de Euler de alto nível

MEV

```
void EulerOperator::mev(HEDModel& _model, int s, int f1, int f2, int v1,
int v2, int v3, int v4, double x, double y, double z)
{
      Solid* oldsolid = NULL;
      Face* oldface1 = NULL;
      Face* oldface2 = NULL;
      HalfEdge* he1 = NULL;
      HalfEdge* he2 = NULL;

      if((oldsolid = _model.getsolid(s)) == NULL)
      {
       std::cout <<"mev: solid "<< s <<" not found. \n";
      }
      if( (oldface1 = _model.fface(oldsolid,f1)) == NULL)
      {
       std::cout <<"mev: face "<< f1 <<" not found in solid "<< s <<". \n";
      }
      if( (oldface2 = _model.fface(oldsolid,f2)) == NULL)
      {
       std::cout <<"mev: face "<< f2 <<" not found in solid "<< s <<". \n";
      }
      if( (he1 = _model.fhe(oldface1,v1, v2)) == NULL)
      {
       std::cout <<"mev: edge "<< v1 <<" - "<< v2 <<" not found in face "<<
f1 <<". \n";
      }
      if( (he2 = _model.fhe(oldface2,v1, v3)) == NULL)
      {
       std::cout <<"mev: edge "<< v1 <<" - "<< v3 <<" not found in face "<<
f2 <<". \n";
      }
      this->lmev(he1, he2, v4,  x,  y, z);
}
```

MEF

```
void EulerOperator::mef(HEDModel& _model, int s, int f1, int v1, int v2,
int v3, int v4, int f2)
{
      Solid* oldsolid = NULL;
      Face* oldface1 = NULL;
      HalfEdge* he1 = NULL;
      HalfEdge* he2 = NULL;

      if((oldsolid = _model.getsolid(s)) == NULL)
      {
       std::cout <<"mef: solid "<< s <<" not found. \n";
      }
      if( (oldface1 = _model.fface(oldsolid,f1)) == NULL)
      {
       std::cout <<"mef: face "<< f1 <<" not found in solid "<< s <<". \n";
      }
```

```
      if( (he1 = _model.fhe(oldface1,v1, v2)) == NULL)
      {
       std::cout <<"mef: edge "<< v1 <<" - "<< v2 <<" not found in face "<<
f1 <<". \n";
      }
      if( (he2 = _model.fhe(oldface1,v3, v4)) == NULL)
      {
       std::cout <<"mef: edge "<< v3 <<" - "<< v4 <<" not found in face "<<
f2 <<". \n";
      }
      this->lmef(he1, he2, f2);
}
```

6. RESULTADOS

Nesta seção, será apresentado um exemplo de utilização do modelador. A barra de ferramentas do modelador é composta pelos ícones:

Select → 

Move → 

Zoom → 

Insert Point → 

Insert Curve → 

Insert Rectangle (não implementado) → 

Nas figuras que se seguem (da Figura 35 a Figura 46), apresenta-se, passo a passo, a sequência de inserção de arestas na interface gráfica para modelar uma seção T.

Na inserção da primeira curva, os operadores de Euler MVFS e MEV são ativados. A partir da segunda curva até a sétima curva, o operador de Euler MEV. Na inserção da última curva, a que fecha a face, o operador de Euler LMEF

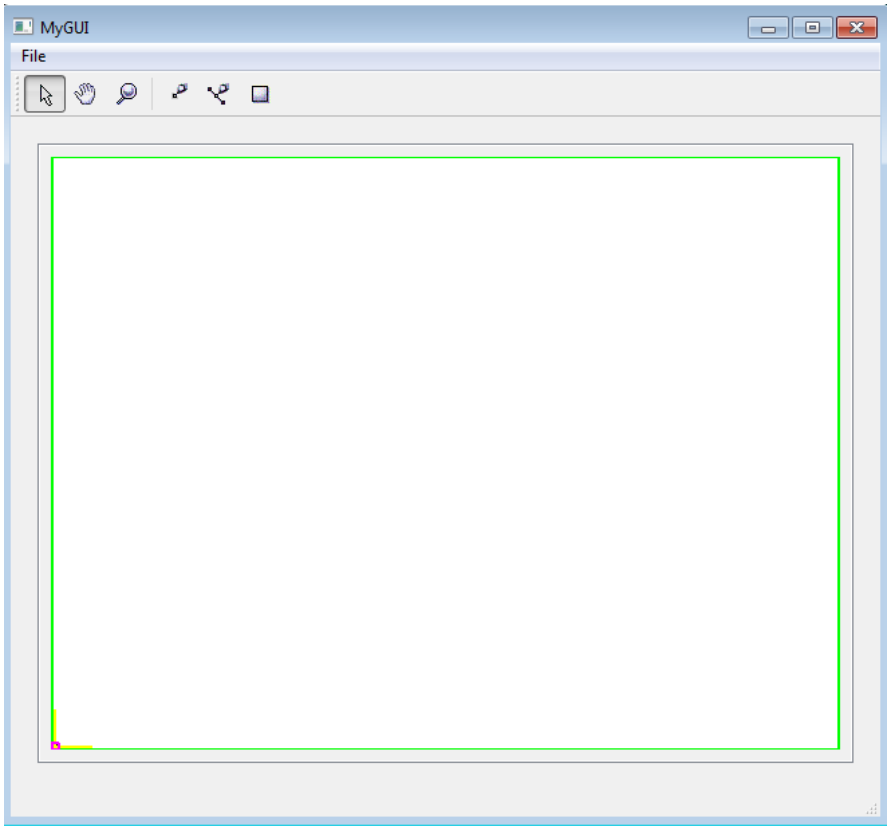


Figura 35 - Interface gráfica do modelador.

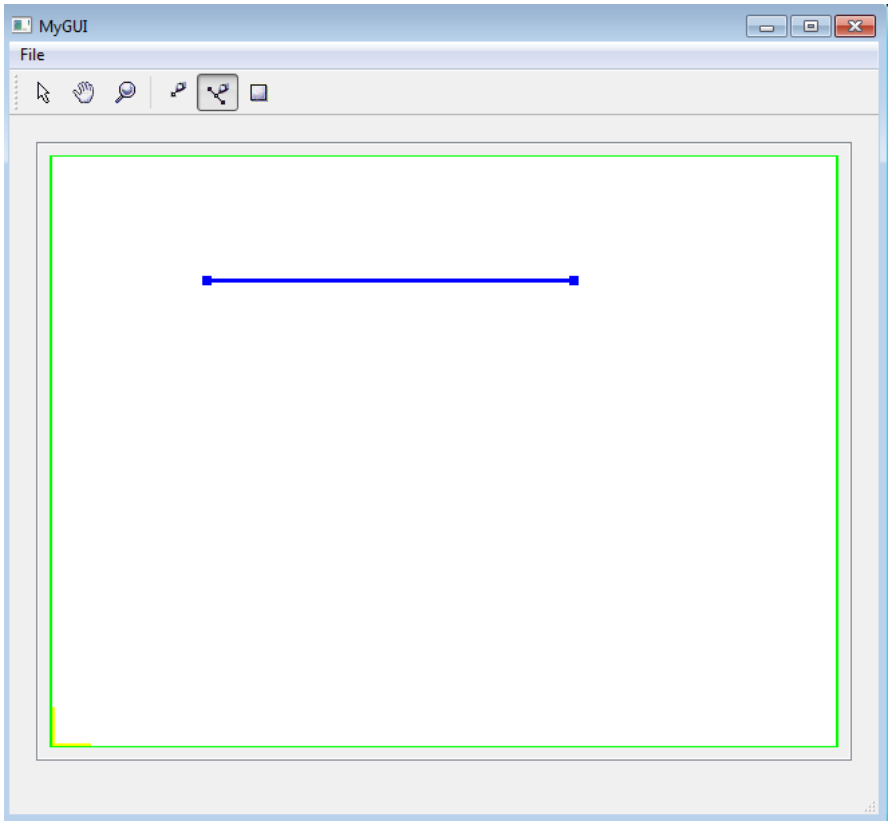


Figura 36 - Inserção da primeira curva.

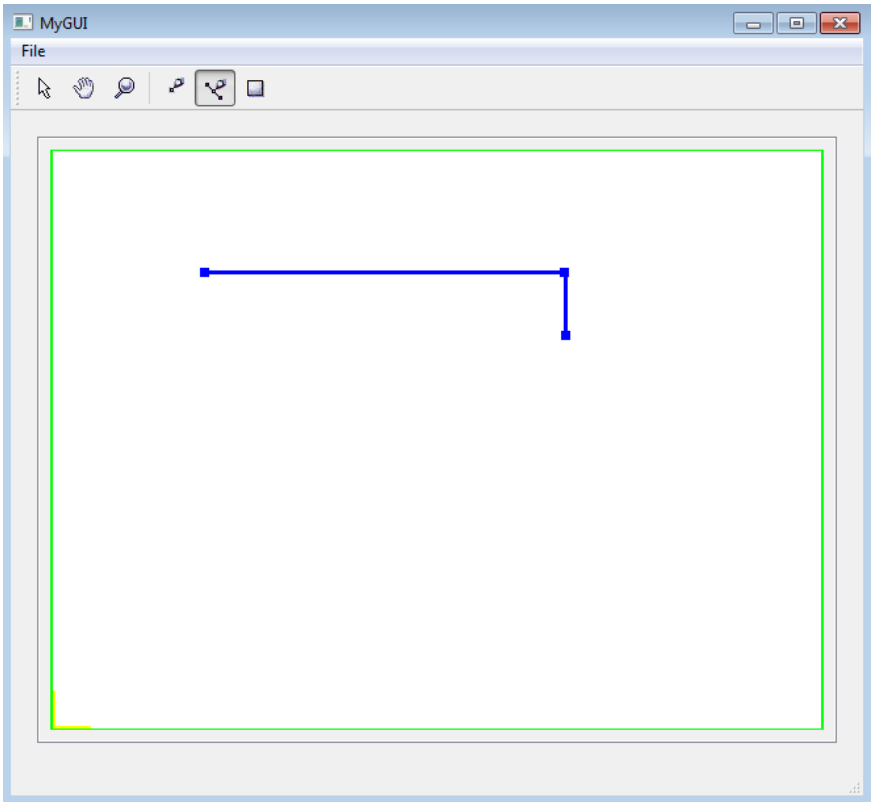


Figura 37 - Inserção da segunda curva.

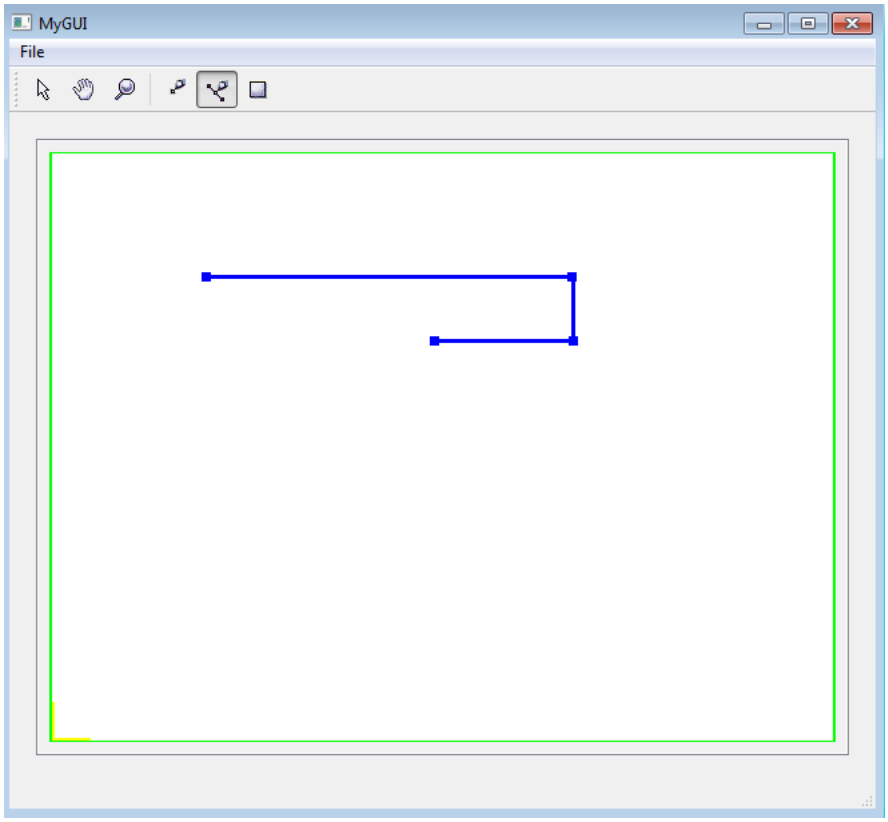


Figura 38 - Inserção da terceira curva.

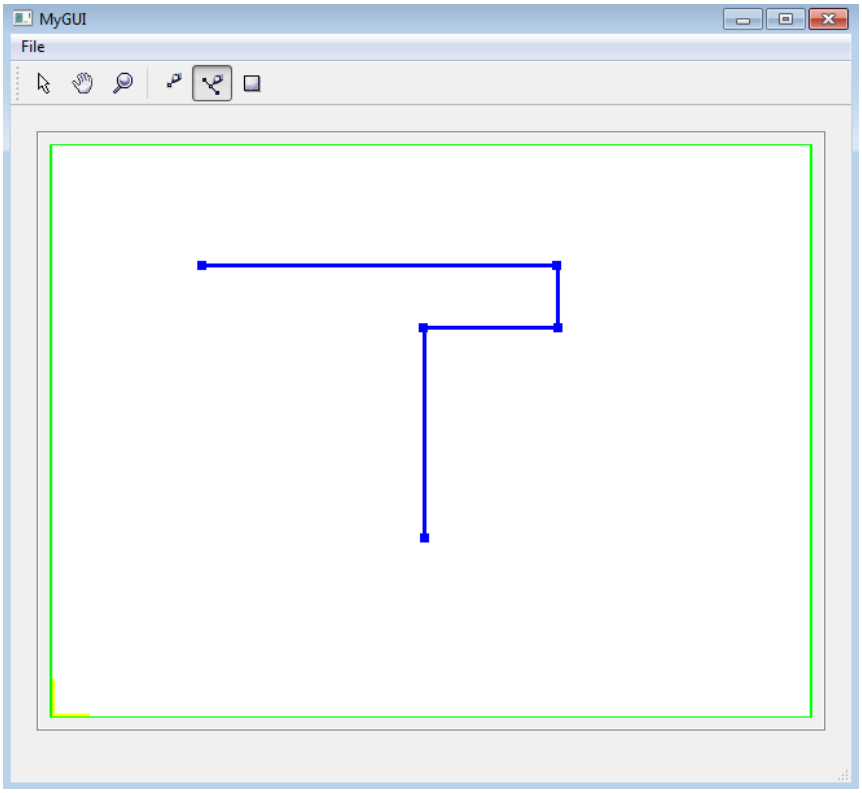


Figura 39 - Inserção da quarta curva.

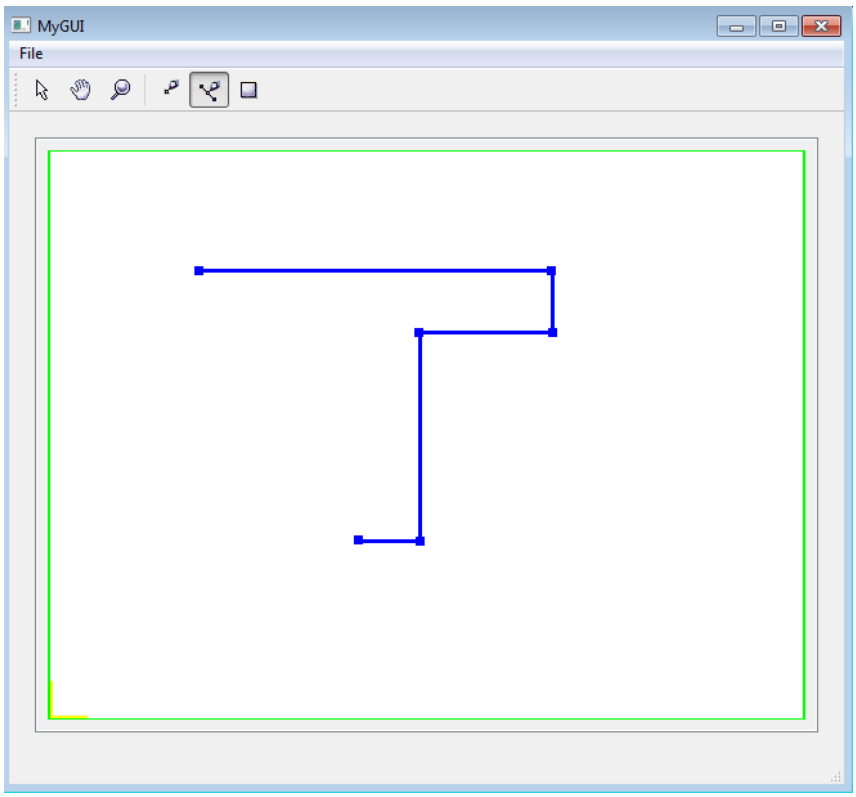


Figura 40 - Inserção da quinta curva.

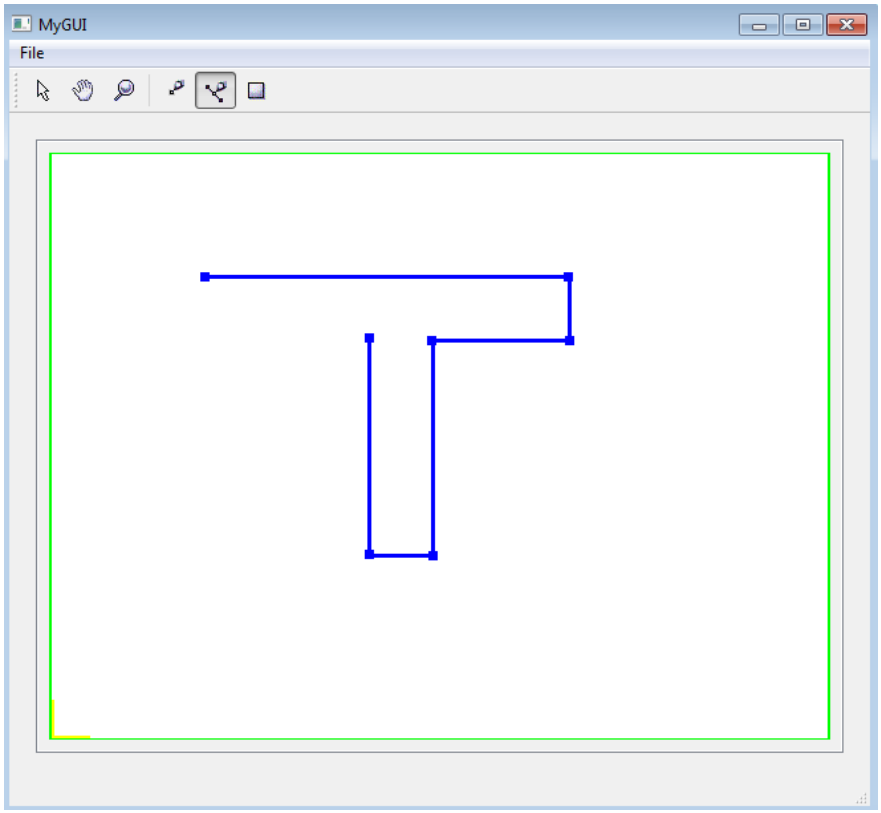


Figura 41 - Inserção da sexta curva.

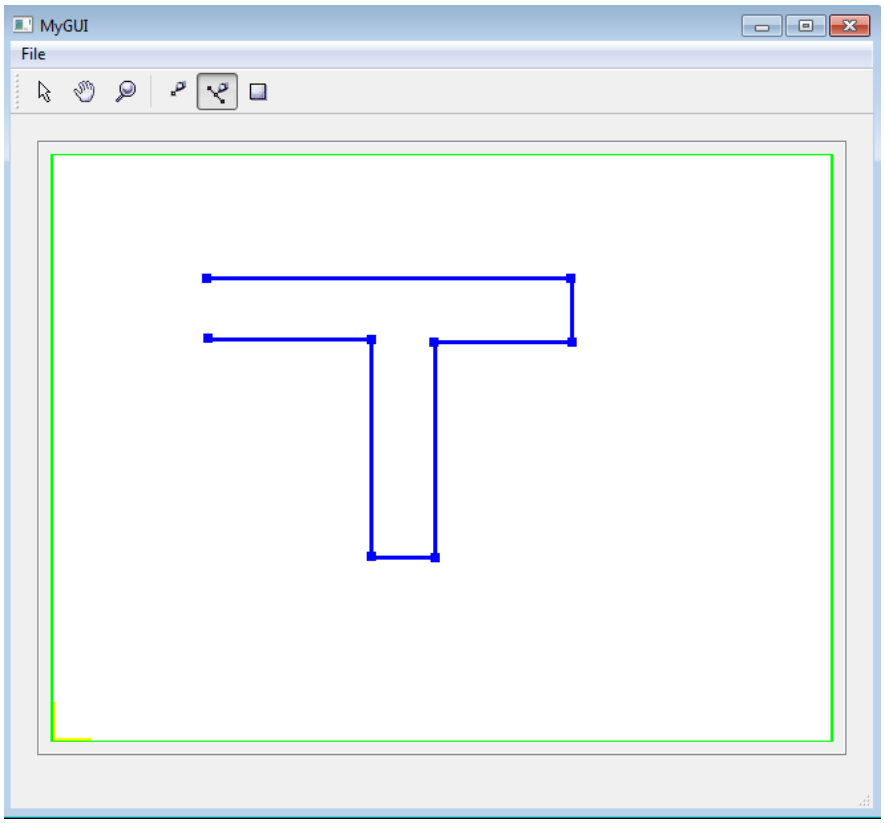


Figura 42 - Inserção da sétima curva.

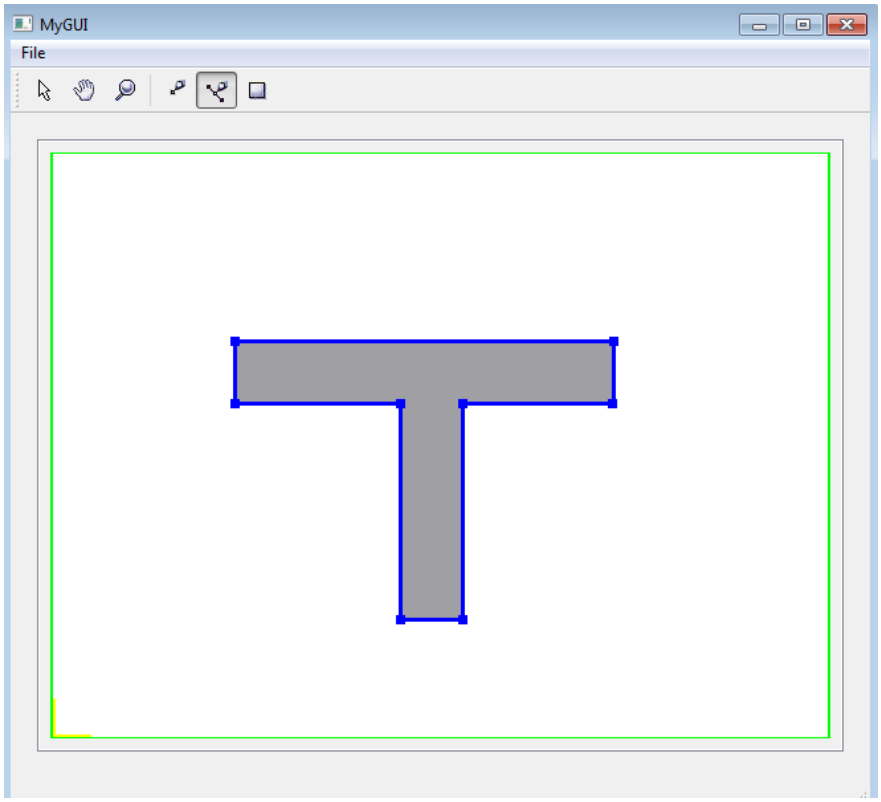


Figura 43 - Inserção da última curva e a detecção da face.

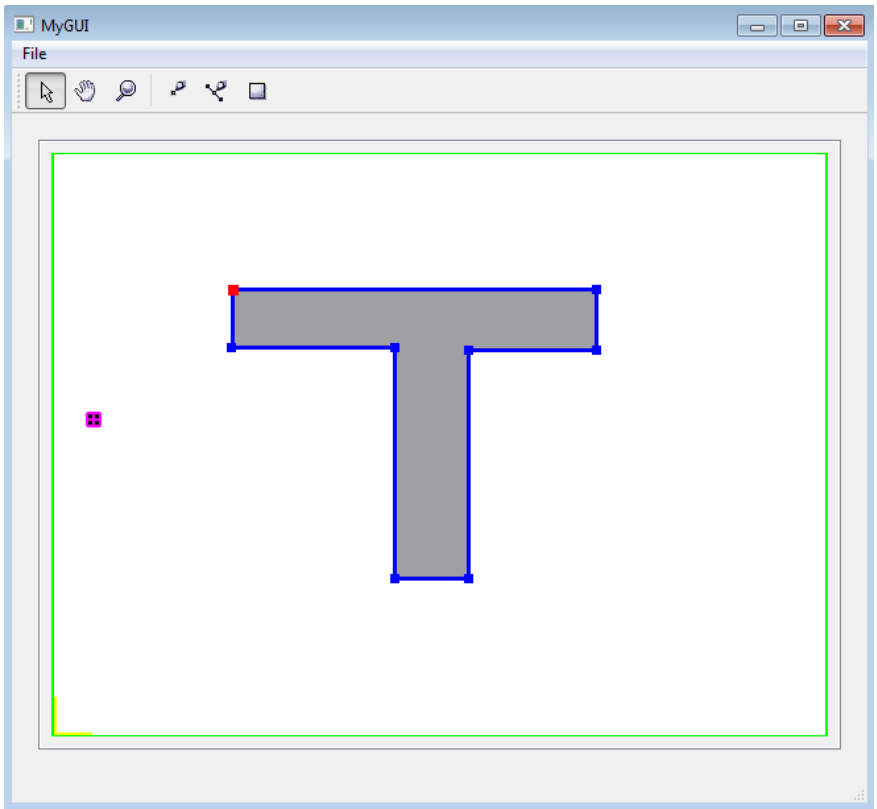


Figura 44 - Seleção de um ponto.

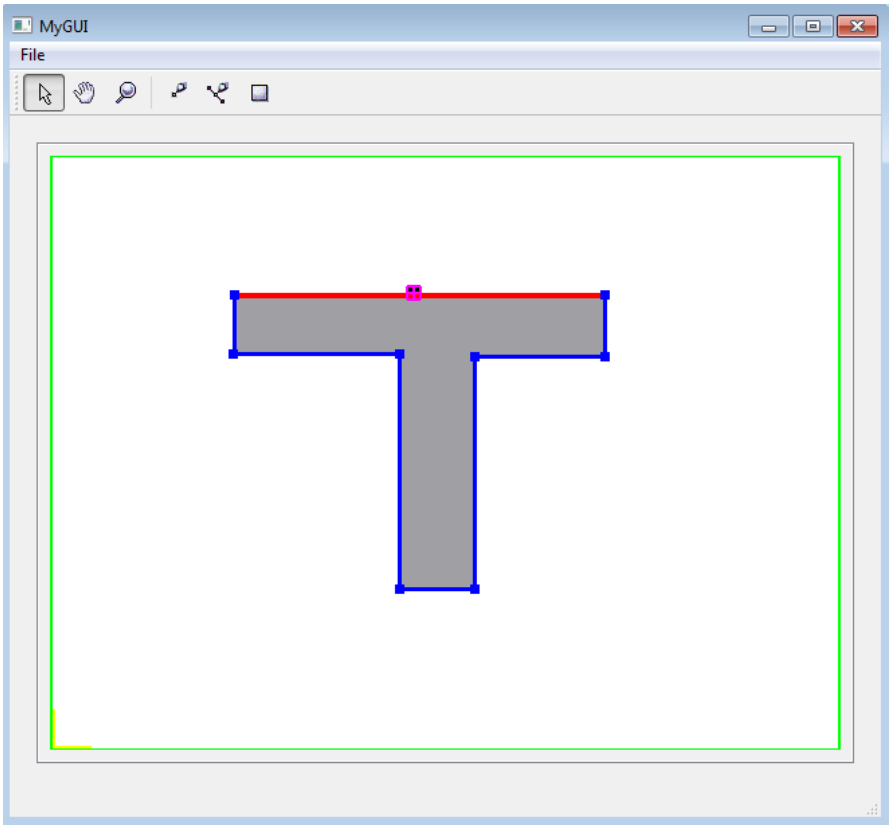


Figura 45 - Seleção de curva.

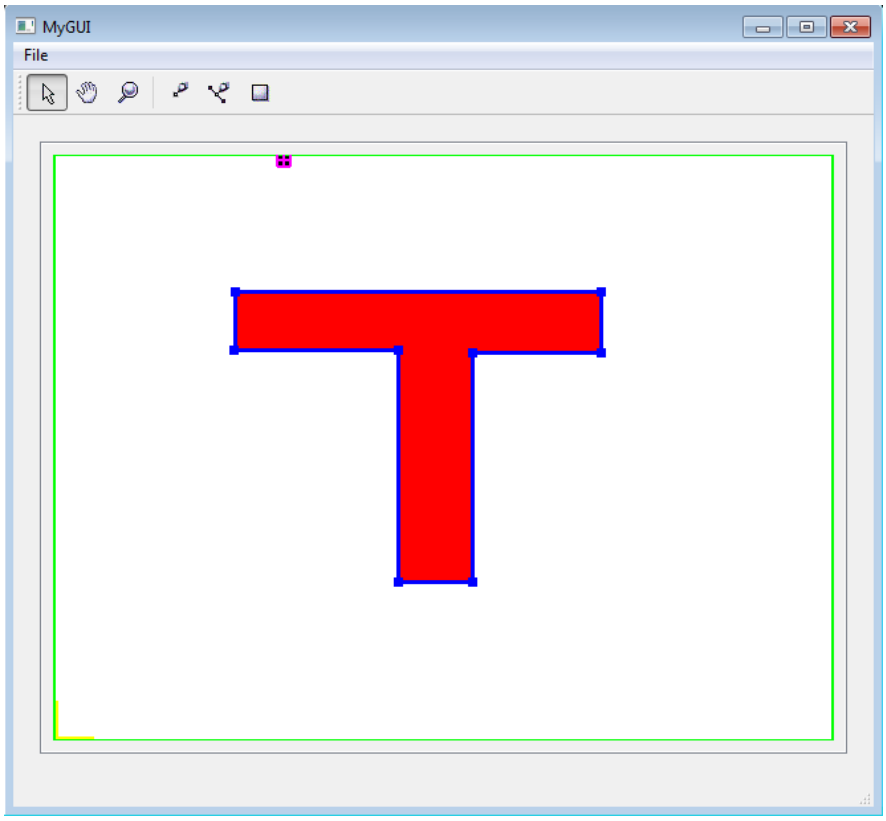


Figura 46 - Seleção de face.

7. CONCLUSÕES

Neste capítulo, faz-se uma síntese final do trabalho, apresentando reflexões acerca do seu teor, dos resultados obtidos e de seu alcance discutindo as conclusões e apresentando as sugestões para trabalhos futuros.

As contribuições obtidas durante o desenvolvimento deste trabalho estão relacionadas à aquisição de conhecimento, construção de *software* e produção científica.

Em relação à aquisição de conhecimento, é importante destacar que para construir o modelador foi necessário um amplo estudo a cerca do desenvolvimento de sistemas de modelagem de sólidos, envolvendo as representações B-rep; a consolidação da análise e programação orientadas a objetos, visando aplicações na Engenharia Civil, além do aprendizado da linguagem de programação C++.

Em relação à construção de *software*, foi necessário um estudo bem aprofundado a cerca da estrutura de dados topológica *Half-Edge*, da implementação do operadores de Euler, bem como da padronização da documentação do sistema, utilizando metodologias de análise e projetos orientados a objetos.

Em relação à produção científica, destaca-se a elaboração, juntamente com o professor orientador André Maués Brabo Pereira, do artigo “Modelagem Orientada a Objetos da Estrutura de dados Topológica Semi- Aresta”, que foi apresentado no Seminário PIBIC 2011, na Universidade Federal Fluminense.

Esses conhecimentos foram imprescindíveis para entender como funciona a estrutura interna de um modelador e certificar o quão importante eles são para a

análise correta dos resultados que os *softwares* utilizados na resolução de problemas de Engenharia oferecem.

Existem diversos aperfeiçoamentos que podem ser incluídos neste modelador, como a implementação de ferramentas que viabilizem cálculo de propriedades geométricas, tais como centróide e momento de inércia, de seções genéricas de vigas e colunas, assim como outras funcionalidades do modelador que ainda não foram implementadas, como deletar um modelo, manipular mais de um modelo simultaneamente, entre outros.

Dessa forma, espera-se que esse trabalho contribua para a realização de projetos futuros, que visem a dar continuidade ao desenvolvimento de modeladores que possam ser utilizados como ferramentas educacionais para facilitar o aprendizado.

REFERÊNCIAS BIBLIOGRÁFICAS

Baumgart, B. G. *A Polyhedron Representation of Computer Vision*. AFIPS National Computer Conference, 1975.

Booch, G.; Rumbaugh, J.; Jacobson, I. *The Unified Modeling Language: User Guide.* Second Edition. Addison Wesley, 2005.

Chiyokura, H. *Solid Modelling with Design base: Theory and Implementation.* San Jose: Addison-Wesley Publishing Company, 1988.

Deitel, H. M. & Deitel, P. J. *C++ Como Programar*. Porto Alegre: Bookman, 2001.

Mäntylä, M. *An Introduction to Solid Modeling*. Computer Science Press, Rockville, 1988.

Weiler, K. *Topological Structures for Geometric Modeling*. Ph.D. Thesis, Rensselear Polytechnic Institute, Troy, New York, 1986.

ANEXO 1

Diagrama de Sequência - Parte 1

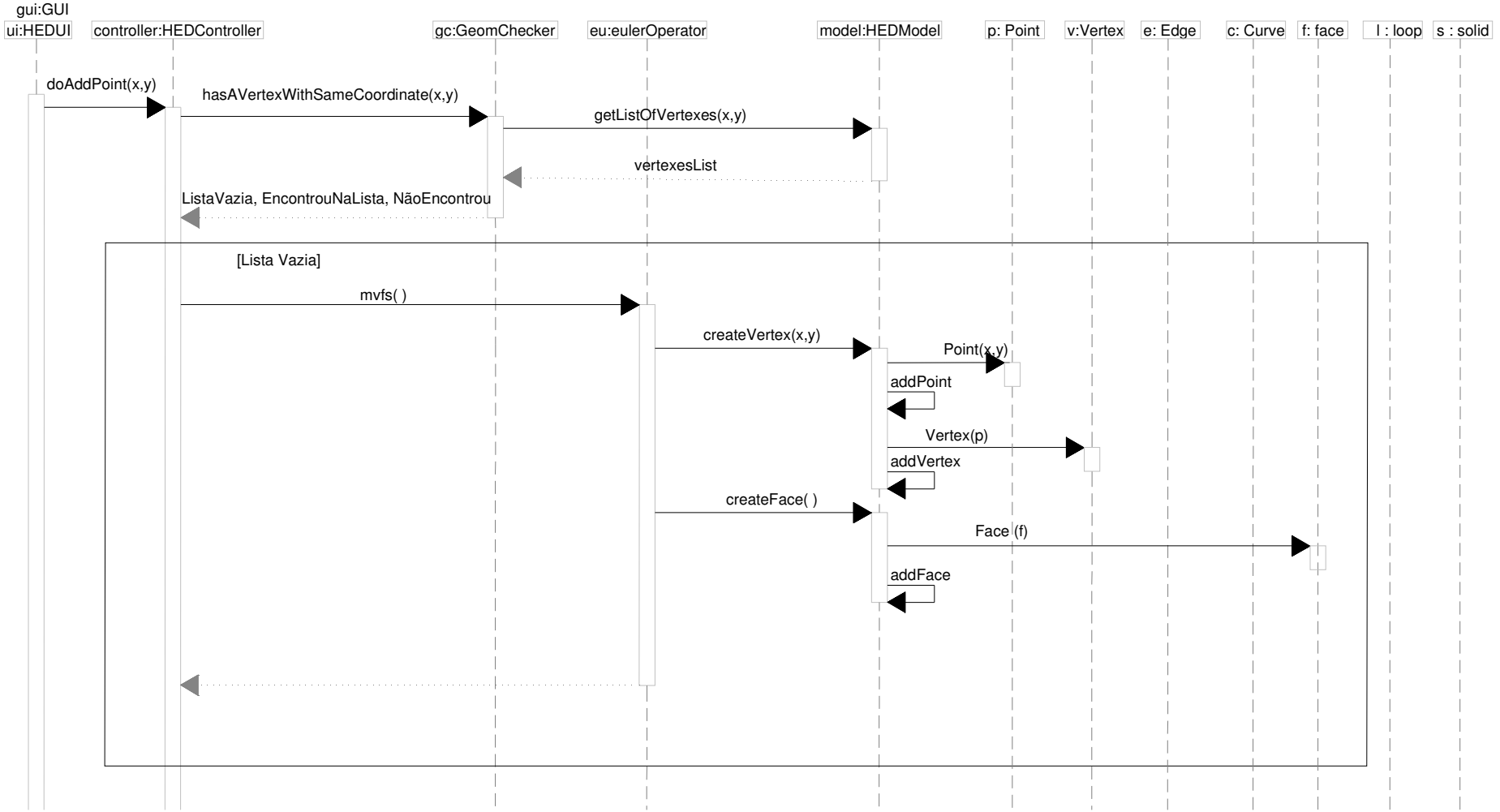


Diagrama de Sequência - Parte 2

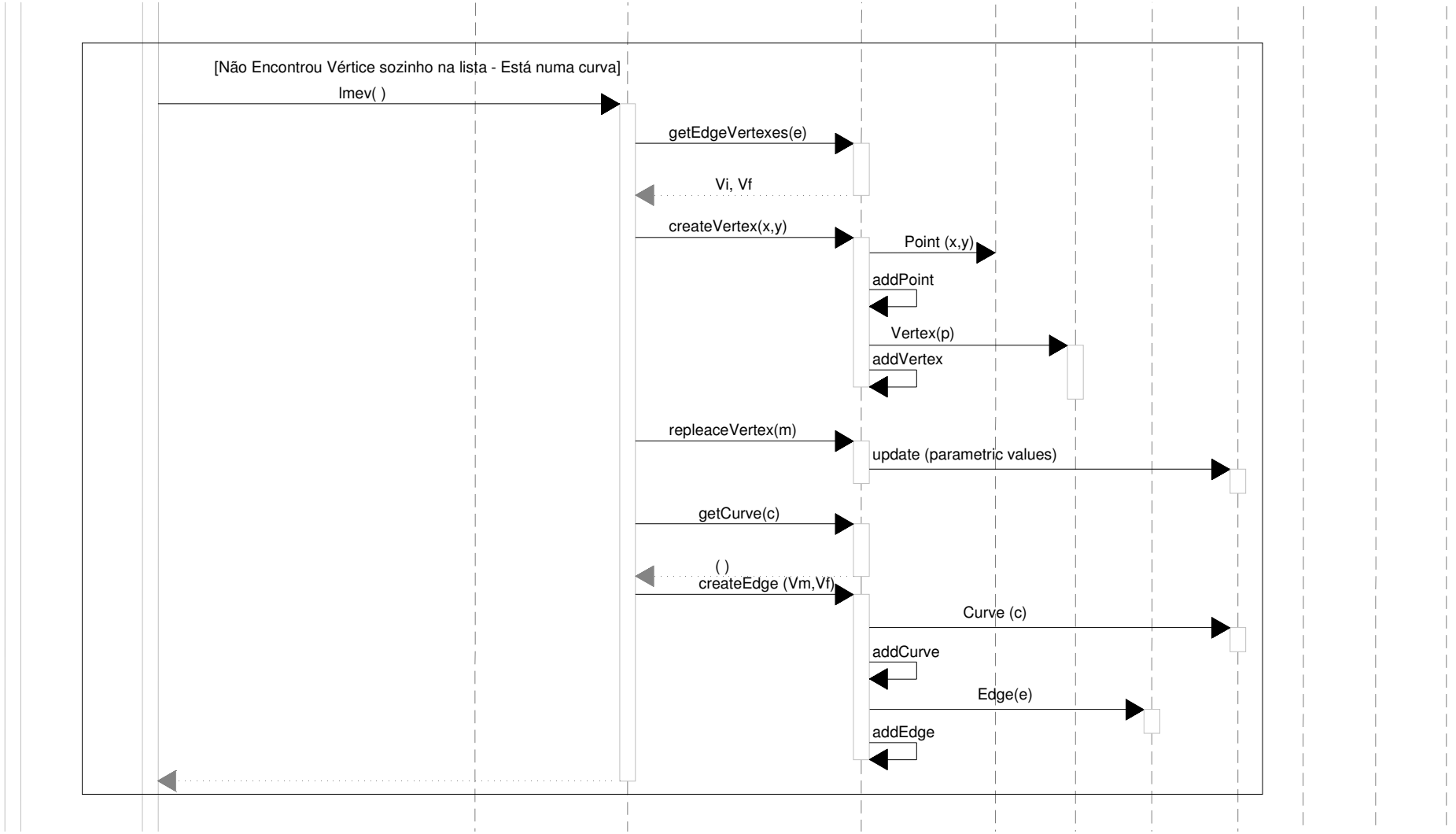


Diagrama de Sequência - Parte 3