FATEC - SP
Faculdade de Tecnologia de São Paulo

Half-Edge: a essência de um modelador 3D para jogos

São Paulo

2009

FATEC - SP
Faculdade de Tecnologia de São Paulo

Half-Edge: a essência de um modelador 3D para jogos

Aluno: Edgard B. Damiani
Grau Pretendido: Monografia submetida como exigência parcial para obtenção do grau de Tecnólogo em Processamento de Dados
Orientador: Gabriel Shammas

São Paulo

2009

Dedico este trabalho à minha esposa, Silvana, e às minhas filhas Stephanny e Amanda (bem-vinda a bordo!).
Esse trabalho também é dedicado a todos aqueles que não se deixam intimidar com as dificuldades e que buscam, corajosa e incessantemente, a Verdade, apesar de todas as tentativas do orgulho humano em ocultá-la ou apresentá-la pela metade.

Agradeço ao professor Gabriel, por ter aceitado orientar um tema tão díspar de sua área de atuação, e à minha família, pela paciência e compreensão.

Lista de Siglas....9

Curriculum Vitae....10

Resumo....11

Abstract....12

INTRODUÇÃO....1

INTRODUÇÃO À MODELAGEM 3D....2

1.1. Origem da modelagem 3D....2

1.2. Tipos de representações 3D....3
1.2.1. Técnicas de modelagem sólida – Classificação....4

COMPONENTES FUNDAMENTAIS DE UMA B-REP....6

2.1. Topologia....6
2.1.1. Objetos sólidos e objetos não-sólidos....7
2.1.2. Homeomorfismo....8
2.1.3. Continuidade....9
2.1.4. Manifold....12
2.1.5. Orientabilidade....14
2.1.6. Superfície – Definição....15
2.1.7. Modelos poligonais como poliedros....17
2.1.8. Característica de Euler (fórmula do poliedro)....18
2.1.9. Fórmula de Euler-Poincaré....18
2.1.10. Operadores de Euler....19

2.2. Geometria....19
2.2.1. Vertex Buffer / Index Buffer....20
2.2.2. A estrutura Winged-Edge....26
2.2.3. Half-Edge: a evolução da Winged-Edge....27

ESTRUTURA HALF-EDGE – PROJETO....31

3.1. Estrutura de classes....31
3.1.1. Classe Solid....32
3.1.2. Classes Face e Loop....32
3.1.3. Classe Vertex....33
3.1.4. Classe HalfEdge....34
3.1.5. Estrutura Half-Edge – Visão geral....35

3.2. Operadores de Euler....35
3.2.1. Operadores de Euler – Diagrama de Classes....36
3.2.2. Macro-operadores de Euler....37

ESTRUTURA HALF-EDGE – IMPLEMENTAÇÃO....42

4.1. Classe Solid....42

4.2. Classe Face....45

4.3. Classe Loop....47

4.4. Classe HalfEdge....49

4.5. Estrutura Point3D....50

4.6. Classe Vertex....51

OPERADORES DE EULER....53

5.1. Operadores construtivos e destrutivos – nomenclatura....53

5.2. Selecionando operadores de Euler....54

5.3. MVFLS....55

5.4. MEV....57

5.5. MEFL....63

5.6. KEML....67

5.7. KFMH....70

CONSTRUINDO E MODIFICANDO A TOPOLOGIA DE UM SÓLIDO COM OPERADORES DE EULER....74

6.1. Criando um cubo com operadores de Euler....74

6.2. Criando uma extrusão no cubo....76

6.3. Testando a consistência do sólido....77

CONCLUSÃO....83

BIBLIOGRAFIA....85

Lista de Figuras

Figura 2.1 – Esquema das pontes de Königsberg.....7

Figura 2.2 – Objetos sólidos e não-sólidos.....8

Figura 2.3 – Transformando uma superfície em outra homeomórfica.....9

Figura 2.4 – O cubo e seu grafo planar correspondente.....10

Figura 2.5 – Cubo vazado e seu grafo planar.....10

Figura 2.6 – Alguns pontos do grafo planar e suas vizinhanças.....11

Figura 2.7 – E se escolhermos um ponto da borda do grafo?.....11

Figura 2.8 – Exemplo de superfície manifold (esquerda) e non-manifold (direita).....13

Figura 2.9 – Orientabilidade do cubo e da fita de Möbius.....14

Figura 2.10 – Duas faces conectadas a uma mesma aresta com direções opostas (do ponto de vista da aresta).....15

Figura 2.11 – Uma superfície poligonal vista como um conjunto de faces separadas (para facilitar a compreensão, apenas as três faces da frente são mostradas).....16

Figura 2.12 – Subcomponentes das B-reps.....17

Figura 2.13 – Superfícies diferentes geradas com o mesmo grupo de vértices.....20

Figura 2.14 – Formação de um triângulo.....21

Figura 2.15 – Conexão dos pontos, formando um cubo.....22

Figura 2.16 – Cubo inicialmente criado pelo modelador fictício.....23

Figura 2.17 – Tentativa de selecionar a face superior do cubo.....23

Figura 2.18 – Os dois triângulos da face superior estão selecionados.....24

Figura 2.19 – Resultado da movimentação da face superior.....25

Figura 2.20 – A estrutura Winged-Edge.....26

Figura 2.21 – Representação de uma meia-aresta e suas conexões.....27

Figura 2.22 – Movimentando uma face e suas adjacências.....28

Figura 2.23 – Uma aresta está conectada a um vértice?.....29

Figura 2.24 – Uma aresta está conectada a uma face?.....29

Figura 3.1 – Diagrama de classes da estrutura Half-Edge.....32

Figura 3.2 – Conexões Face / Loop inicial (esq.) e Loop / HalfEdge inicial (dir.).....33

Figura 3.3 – Conexão Vertex / HalfEdge inicial.....33

Figura 3.4 – Meia-aresta conectada ao vértice de partida (a) e ao vértice de chegada (b).....34

Figura 3.5 – Conexões da classe HalfEdge.....34

Figura 3.6 – Visão global das conexões da estrutura Half-Edge.....35

Figura 3.7 – Diagrama de Classes: operadores de Euler inseridos.....37

Figura 3.8 – Operador MEV aplicado a um vértice do cubo.....38

Figura 3.9 – Chanfro aplicado a uma face do cubo.....39

Figura 3.10 – Grafo planar mostrando a aplicação do chanfro.....40

Figura 5.1 – Operação MVFLS.....56

Figura 5.2 – Operação MEV após uma operação MVFLS.....58

Figura 5.3 – Operação MEV em loops já formados.....59

Figura 5.4 – Operação MEV aplicada em apenas um loop.....62

Figura 5.5 – Desenvolvimento da operação MEFL.....64

Figura 5.6 – Resumo da operação MEFL.....65

Figura 5.7 – Resumo da operação KEML.....68

Figura 5.8 – Aplicação do operador KFMH.....70

Figura 5.9 – Transferência do loop de face2 para face1.....71

Figura 5.10 – Três MEFL consecutivos, gerando três novas faces (mefl1, mefl2 e mefl3).....71

Figura 6.1 – Criação passo-a-passo do cubo.....76

Figura 6.2 – Aplicação de extrusão em uma das faces.....76

Lista de Siglas

E	aresta
F	face
H	*hole*, ou buraco
he	*half-edge*, ou meia-aresta
L	loop
nhe	*new half-edge*, ou nova meia-aresta
R	*ring*, ou loop interno
S	*shell*, ou superfície
V	*vertex*, ou vértice

Curriculum Vitae

Edgard B. Damiani teve sua primeira experiência com programação de jogos em 1987. Vinte e dois anos de experiência com computadores lhe renderam trabalhos nas áreas de web sites, desenvolvimento de sistemas, animação, design gráfico, mas, principalmente, nas duas áreas que moram em seu coração: jogos e ensino. É palestrante na área de desenvolvimento de jogos, além de ser autor de diversos guias e livros publicados pela Novatec Editora, de artigos publicados na revista Digital Designer, além de apostilas de desenvolvimento de jogos utilizadas em escolas especializadas. É professor de Computação Gráfica e desenvolve jogos em C++ utilizando bibliotecas open source.

Resumo

A comunidade de desenvolvimento de jogos testemunha o surgimento de vários game engines desenvolvidos em código-aberto, que tentam preencher a lacuna existente entre a tecnologia e as necessidades dos designers de jogos.

Isso, em si, é uma coisa boa, mas não é o suficiente. A comunidade clama por ferramentas que corroborem o ponto de vista do designer, pois é ele quem enxerga o contexto geral e entende os meandros do jogo. Para realizar esse clamor, em uma época em que os gráficos 3D são a tecnologia padrão, devemos concluir que o desenvolvimento de jogos, seja durante o processo de prototipação, seja após a fase de projeto, inicia sua concretização com a modelagem dos objetos e cenários 3D.

No entanto, mesmo sendo esse o ponto de partida para o processo de concretização do jogo, a modelagem 3D é amplamente negligenciada nas ferramentas de desenvolvimento de jogos, e acredito que a resposta para tal discrepância, na verdade, seja simples: não é fácil se motivar para criar um software consistente de modelagem 3D, em parte porque já existem várias opções (caras!) no mercado, mas também porque a base de conhecimento necessária para construi-lo é esparsa e está espalhada em inúmeros artigos que dizem *o que* ela é, mas não *como* realizá-la.

Este trabalho é uma tentativa de preencher tal lacuna técnica, apresentando, de forma detalhada, a principal estrutura de dados por trás da modelagem 3D para jogos – a Half-Edge. O trabalho inicia com uma discussão não-técnica e intuitiva dos conceitos matemáticos de topologia e geometria necessários à compreensão do problema, seguido por uma implementação em C++ da estrutura de dados e terminando com uma implementação dos operadores de Euler, que são vitais para o processo de modelagem 3D com Half-Edge.

Abstract

The game development community witnesses the rising of many open-source game engines, in an attempt to fill the gap between technology and game designers' needs.

While it's a good thing in itself, it's not enough. The community urges for tools that corroborate and realize the designer's point of view, since it's him who sees the big picture and knows the whereabouts of the game. In order to make it come true, in a time where 3D graphics are the standard, we must conclude that the realization of the game, be it at the prototype stage or after the design process, begins with the modeling of 3D objects and sceneries.

Nevertheless, even if that's the starting point for the realization process, 3D modeling is widely neglected in game developing tools, and I think the answer for that discrepancy is actually simple: it's not easy to motivate yourself in creating a consistent 3D modeling software, partly because there are so many (expensive!) options already in market, but also because the technical background needed to build it is sparse and scattered around many papers that says *what* it is, but not *how* to accomplish it.

This work is an attempt of filling this technical gap, presenting, in a detailed fashion, the main data structure behind 3D game modeling – the Half-Edge. It begins with a non-technical, intuitive discussion about the mathematical concepts of topology and geometry needed to understand the problem in question, followed by a C++ implementation of the data structure and finishing with an implementation of Euler operators, which are vital to the Half-Edge 3D modeling process.

Introdução

Apesar da área de modelagem 3D ter pelo menos 40 anos, a modelagem específica para jogos ganhou força apenas no final da década de 1990. Devido a certos requisitos inerentes aos jogos, além das restrições técnicas do ambiente de execução, faz-se necessário estreitar o escopo da modelagem 3D para um determinado subconjunto que satisfaça, da melhor maneira possível, os requisitos irreconciliáveis de desempenho e realismo.

A modelagem sólida, também conhecida como modelagem poligonal, apresenta o melhor custo-benefício entre desempenho e fidelidade de representação, impondo inúmeras restrições aos tipos de objetos que podem ser representados.

Este trabalho tem como objetivo dissecar a estrutura de dados mais utilizada em técnicas de modelagem 3D voltadas para jogos, a Half-Edge. Além disso, pretende demonstrar o uso dos operadores de Euler para manipular as características topológicas do sólido.

Capítulo 1

Introdução à modelagem 3D

1.1. Origem da modelagem 3D

A origem da modelagem de objetos 2D/3D remonta à indústria naval e, mais recentemente, às indústrias automobilística e aeronáutica. Essas três indústrias tinham um problema em comum: a construção de superfícies curvas; para tanto, utilizavam tiras finas de madeira, facilmente curváveis, denominadas ripas ou *splines*, presas em determinados pontos por peças de metal denominados *nós*. O processo de construir uma superfície a partir de splines é denominado *lofting* (literalmente, 'fazer o teto'); uma superfície pode ser feita a partir de subconjuntos de splines denominados *patches* (remendos).

Devido à complexidade de certos projetos automobilísticos e aeronáuticos, os projetistas sentiram a necessidade de ferramentas auxiliares – daí surgiu o primeiro impulso para a área de modelagem de objetos por computador, com ferramentas chamadas de *Computer-Aided Geometric Design* ou CAGD (Projeto Geométrico Auxiliado por Computador).

Curiosamente, a modelagem geométrica, que é o processo utilizado pelos softwares CAGD, manteve todos os termos citados anteriormente.

1.2. Tipos de representações 3D

A modelagem de objetos 2D/3D se utiliza de diferentes representações dos sólidos, de acordo com as necessidades de cada software:

- Softwares CAD (*Computer-Aided Design*) em geral costumam utilizar *representações aramadas* (*wireframe*) dos objetos, pois essa representação é natural para os projetistas de prancheta, que utilizavam papel e nanquim para desenvolver seus projetos;
- Softwares CAGD costumam utilizar a *modelagem geométrica*, como já foi dito anteriormente; além dela, até recentemente a indústria de filmes 3D também utilizava essas mesmas técnicas para modelar os objetos, cenários e personagens dos filmes. A ideia da modelagem geométrica é gerar superfícies paramétricas, ou seja, superfícies que são criadas e controladas por meio de cálculos matemáticos realizados sobre parâmetros previamente definidos.
- Com o advento e rápida evolução do mundo dos jogos eletrônicos, surgiu a necessidade de se criar uma representação de objetos sólidos que fosse processada com mais rapidez pelos computadores, já que a modelagem geométrica precisa ser convertida em objetos intermediários para serem desenhados na tela. Com isso, a *modelagem sólida* ganhou força – esse tipo de modelagem utiliza, basicamente, poliedros (objetos poligonais) para representar os objetos da vida real. A modelagem sólida tem a grande desvantagem de, por vezes, gerar objetos que são apenas aproximações rudimentares de suas contrapartes reais – no entanto, devido à sua forma de representação e armazenamento interno, os computadores conseguem processá-los de maneira trivial, gerando um ganho enorme em desempenho.

- Por mais que a modelagem geométrica tenha seus méritos na criação de filmes 3D, as superfícies geradas possuem certos tipos de limitações – em especial na hora de deformá-las para animar os objetos – que restringem sua utilidade nesse meio. Com isso, estúdios como a Pixar passaram a usar um novo tipo de representação, denominado *modelagem por subdivisão*, que supre as carências da modelagem geométrica. A ideia básica é modelar os objetos de acordo com os princípios da modelagem sólida, ou seja, manipulando objetos poligonais, mas a representação final desses objetos recebe um tratamento de arredondamento das bordas dos polígonos, tratamento esse criado por uma técnica especial de subdivisão das faces. No final das contas, a modelagem por subdivisão une a flexibilidade da modelagem geométrica com a precisão visual da modelagem geométrica.

O foco deste trabalho recai sobre a modelagem sólida – ou, mais especificamente, sobre uma das estruturas de dados utilizadas para armazenar o sólido: a estrutura *Half-Edge*.

1.2.1. Técnicas de modelagem sólida – Classificação

Para entender onde a estrutura Half-Edge se encaixa no mundo da modelagem sólida, observe a seguinte classificação das técnicas de modelagem sólida, feita por Christoph Hoffmann:

- Modelagem Sólida
    - Constructive Solid Geometry (CSG)
    - Boundary Representation (B-Rep)
        - Winged Edge
        - Half-Edge

A modelagem sólida utiliza, basicamente, duas técnicas de construção dos sólidos:

- *Constructive Solid Geometry* (Geometria Sólida Construtiva) ou *CSG*, que constrói o sólido a partir de sólidos primitivos (cubos, pirâmides, cilindros etc.). A ideia básica é a seguinte: sobrepor sólidos primitivos e realizar entre eles operações de união, intersecção, remoção etc. Essas operações são denominadas operações booleanas, pois são feitas a partir das operações booleanas E, OU, NÃO e seus derivados.
- *Boundary Representation* (Representação de Fronteiras) ou *B-Rep*, que constrói o sólido a partir de manipulações de uma superfície pré-existente. Todas as operações são realizadas sobre uma mesma superfície, ao contrário da CSG, que utiliza várias superfícies separadas (os sólidos primitivos) para construir a superfície final.

Antes de entrarmos no mérito das estruturas de dados utilizadas pelas B-Reps, devemos olhar mais de perto os seus componentes fundamentais, que é o assunto do próximo capítulo.

Capítulo 2

Componentes fundamentais de uma B-Rep

Uma B-Rep é dividida em dois componentes: *topologia* e *geometria*. A topologia se preocupa com a relação entre os subcomponentes que formam a superfície, enquanto a geometria trata da distribuição desses elementos no espaço.

2.1. Topologia

O termo topologia pode ser traduzido como *estudo da superfície*. A topologia é um ramo da Matemática que estuda certas características das superfícies que independem de sua geometria, ou seja, da posição dos elementos no espaço.

A principal utilidade da topologia para o nosso estudo é garantir, de forma matemática, a validade e a consistência das superfícies representadas pela estrutura Half-Edge.

O primeiro estudo topológico foi realizado por Leonhard Euler em seu artigo *Solutio Problematis ad Geometriam Situs pertinentis* (Solução de um Problema relacionado à Geometria de Posição) de 1736. Nesse artigo, Euler aborda o problema que ficou conhecido como As Sete Pontes de Königsberg, mostrado na figura a seguir:

Figura 2.1 – Esquema das pontes de Königsberg

O problema pode ser formulado da seguinte maneira: como atravessar as sete pontes da cidade de Königsberg, de forma a se passar por elas apenas uma vez?

Euler provou que o problema não tem solução; além disso, demonstrou que esse tipo de problema independe do tamanho das pontes ou das distâncias entre elas, dependendo exclusivamente de *como* elas se ligam aos diversos pedaços de terra que as cercam.

Esse exemplo mostra claramente o objeto de estudo da topologia: a conectividade e a continuidade dos elementos que compõem a superfície, sem se preocupar com a sua forma. Tais elementos são chamados de *invariantes topológicas* – a característica de Euler, que será vista mais adiante, é um exemplo de invariante.

2.1.1. Objetos sólidos e objetos não-sólidos

A topologia é um estudo muito vasto, e nem todos os tipos de objetos estudados por ela nos interessam – mais especificamente, estamos preocupados apenas com objetos tridimensionais sólidos, que possuam uma distinção clara entre dois lados: o interior e o exterior.

A figura a seguir mostra quatro objetos: os dois de cima são objetos sólidos, dividindo claramente o espaço tridimensional nos lados de dentro e de fora, enquanto os dois de baixo não são sólidos, tornando ambíguo o nosso conceito de exterior e interior.

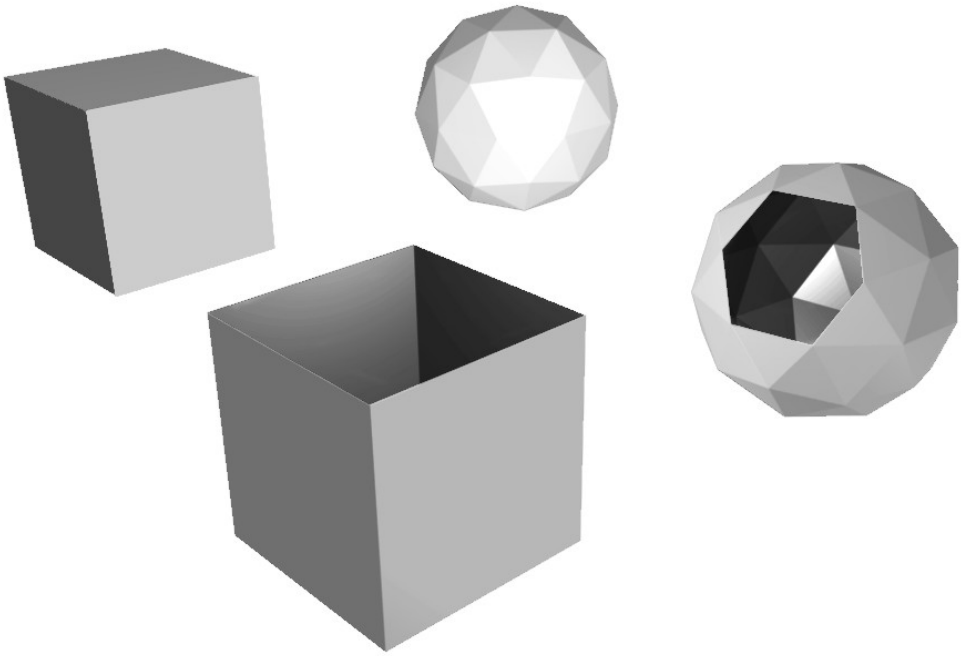

Figura 2.2 – Objetos sólidos e não-sólidos.

2.1.2. Homeomorfismo

Como a topologia não leva em conta a geometria dos objetos, é possível abstrair certas propriedades aparentemente absurdas do ponto de vista geométrico – uma delas é a noção de *homeomorfismo*.

Homeomorfismo significa *mesma forma* – no sentido topológico, duas superfícies são homeomórficas quando uma pode ser transformada na outra sem a necessidade de cortar e recolar a superfície:

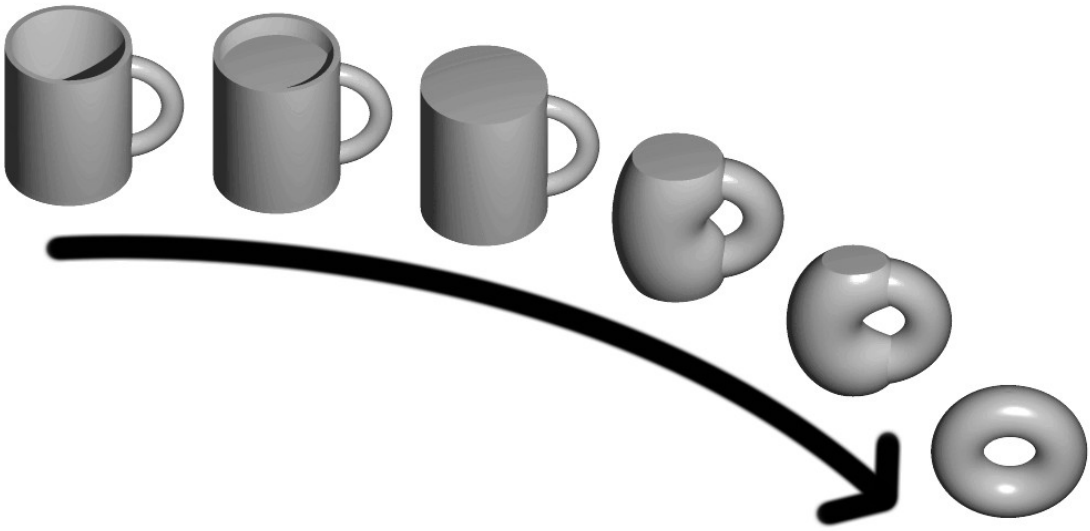

Figura 2.3 – Transformando uma superfície em outra homeomórfica.

Aliás, é por causa dessa propriedade que existe a piada de que um topólogo é uma pessoa que não sabe a diferença entre uma xícara e um donut...

2.1.3. Continuidade

Uma propriedade dos sólidos, importante para o nosso estudo, é a noção de *continuidade*. No entanto, a continuidade é um conceito abstrato e complicado de explicar diretamente em sólidos 3D – por isso, vamos usar o artifício dos grafos planares para compreendê-lo.

Um *grafo planar* é um grafo que, ao ser projetado em um plano bidimensional, não tem arestas cruzando umas sobre as outras – com ele, é possível representar objetos tridimensionais de forma não-ambígua em planos bidimensionais.

Tomemos como exemplo o cubo. Uma maneira de representá-lo como um grafo planar é apresentado na figura a seguir:

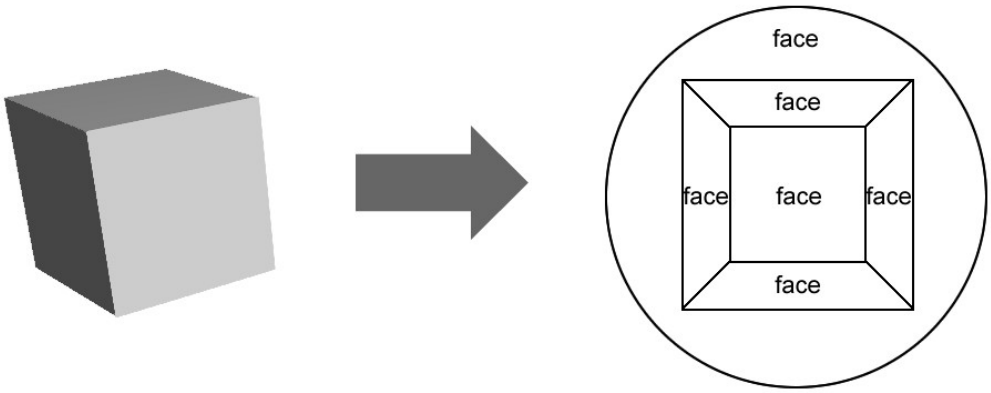

Figura 2.4 – O cubo e seu grafo planar correspondente.

O círculo em torno do grafo indica a face de baixo do cubo, que conecta as quatro faces mais externas do grafo.

De acordo com Sven Havemann, um cubo com um buraco atravessando duas faces opostas poderia ser representado assim:

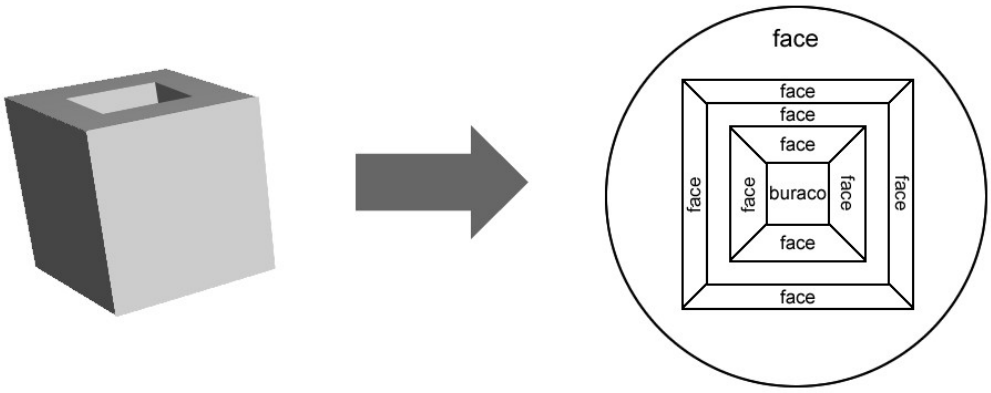

Figura 2.5 – Cubo vazado e seu grafo planar.

Daqui por diante, todas as operações de manipulação realizadas nos sólidos serão representadas preferencialmente na forma de grafos planares.

Voltemos, então, à questão da continuidade.

Observe mais uma vez o grafo planar do cubo, mostrado na figura 2.4. Agora, escolha um ponto qualquer dentro do grafo e desenhe um pequeno disco em torno do

ponto, chamado de *vizinhança* do ponto; se for possível desenhar um disco em torno de qualquer ponto do grafo, sem que o disco saia do grafo, diz-se que *a superfície é contínua*.

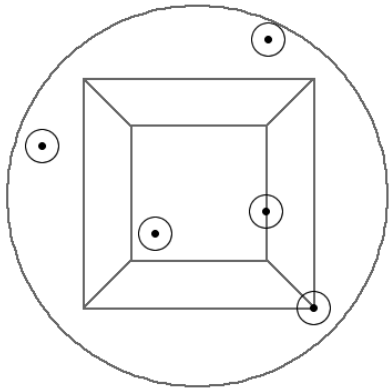

Figura 2.6 – Alguns pontos do grafo planar e suas vizinhanças.

"Mas espere um pouco", você dirá, "se eu escolher um dos pontos da borda do grafo, não será possível desenhar um círculo em torno dele!" – e você terá toda a razão!

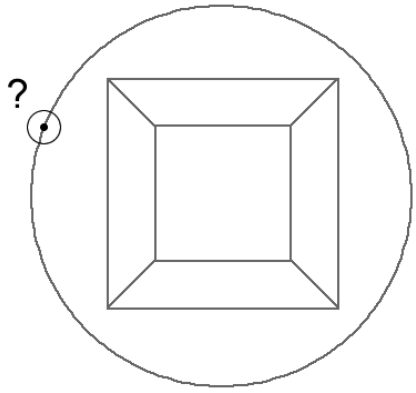

Figura 2.7 – E se escolhermos um ponto da borda do grafo?

No entanto, vale lembrar que, neste contexto, a representação visual do grafo é apenas isso: uma tentativa de representar um objeto 3D em um plano 2D.

Para entender a questão, é importante ter em mente que existe uma relação indissociável entre o objeto 3D e sua representação 2D – portanto, como o grafo foi

construído a partir de um sólido, *a representação 2D não pode ser vista isolada de sua contraparte 3D*.

Observe o círculo que envolve a representação planar do cubo e que representa sua face inferior – a grande questão é: esse limite existe de verdade no cubo 3D? Imagine uma formiga andando na superfície do cubo – se ela continuar seguindo em frente, ela chegará em algum limite que a impeça de continuar andando?

A resposta é *não* – não há barreiras que impeçam seu movimento. Portanto, *o limite mostrado no grafo planar do cubo não existe*, e é fruto da incapacidade de se representar perfeitamente um objeto 3D em um espaço 2D.

Em termos matemáticos, os objetos que estamos estudando são considerados *conjuntos* – ou seja, uma superfície pode ser vista como um conjunto infinito de pontos que a compõe. Se a superfície tiver, de fato, bordas externas, que seriam representadas tal como o círculo no grafo do cubo, significa que existe um limite de deslocamento dos pontos na superfície; essas superfícies seriam consideradas como *conjuntos fechados*.

Já os sólidos que pretendemos representar por meio dos grafos não possuem bordas externas – portanto, não existe um limite para a identificação dos pontos da superfície, tal como uma curva assintótica se aproxima infinitamente de uma reta assíntota, mas nunca a toca. Por isso, esses tipos de superfícies são chamados de *conjuntos abertos*.

Tendo isso em mente, é possível provar que todos os pontos da superfície possuem um disco de pontos à sua volta – afinal, por mais perto que um ponto esteja da borda, sempre existirão pontos mais próximos da borda do que ele. Assim, por menor que seja, sempre haverá um disco de pontos ao redor dele!

É por isso que, para haver continuidade em uma superfície, ela deve ser um

conjunto aberto.

2.1.4. Manifold

Agora que definimos a vizinhança, podemos partir para o conceito de *manifold*. Dado um ponto localizado na superfície de um sólido, analisa-se sua vizinhança para definir a sua continuidade. Se todos os pontos da superfície de um sólido forem contínuos – ou seja, se a vizinhança de todos os pontos for homeomórfica a um disco – significa que a superfície é *manifold*; no entanto, se a vizinhança de algum ponto da superfície do sólido não for contínua, dizemos que a superfície é *non-manifold*.

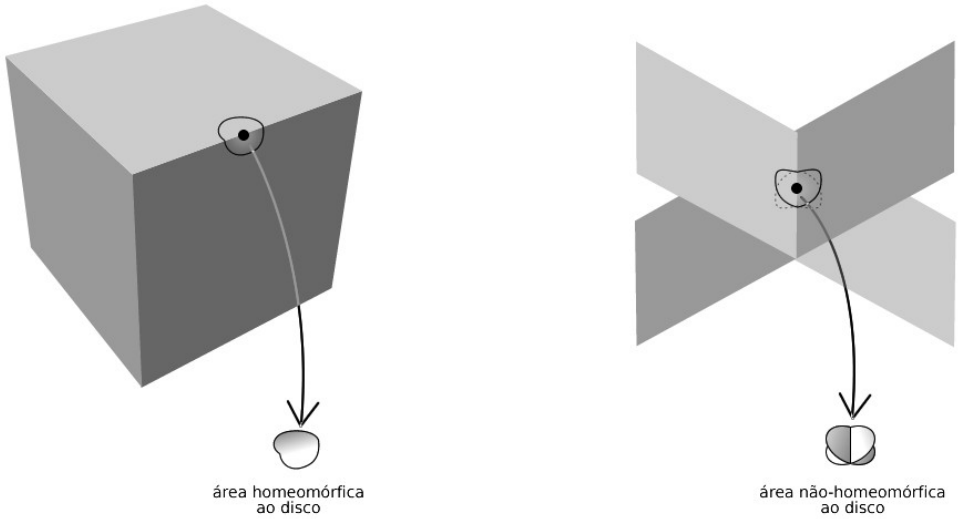

Figura 2.8 – Exemplo de superfície manifold (esquerda) e non-manifold (direita).

Manifolds podem possuir *n* dimensões. Para saber quantas dimensões tem o manifold de um determinado objeto, é preciso observar esse objeto *localmente* – o objeto resultante da observação local indica o número de dimensões do manifold.

Qualquer objeto tridimensional que se enquadre na definição de manifold é um manifold de 2 dimensões, ou *2-manifold*. Para provar isso, basta perceber que a observação da área local ao redor de qualquer ponto da superfície do sólido gera uma

área homeomórfica ao disco, que é um objeto bidimensional.

A distinção entre superfícies manifold e non-manifold é importante para validar as superfícies criadas por meio da modelagem sólida, pois as estruturas de dados e as operações de manipulação apresentadas neste trabalho lidam apenas com superfícies manifold.

2.1.5. Orientabilidade

De forma simplificada, a *orientabilidade* indica se é possível dar uma mesma orientação – ou seja, sentido de rotação – para todas as faces de um sólido; assim, um cubo é orientável, mas uma fita de Möbius não o é – acompanhe a figura para entender o caso:

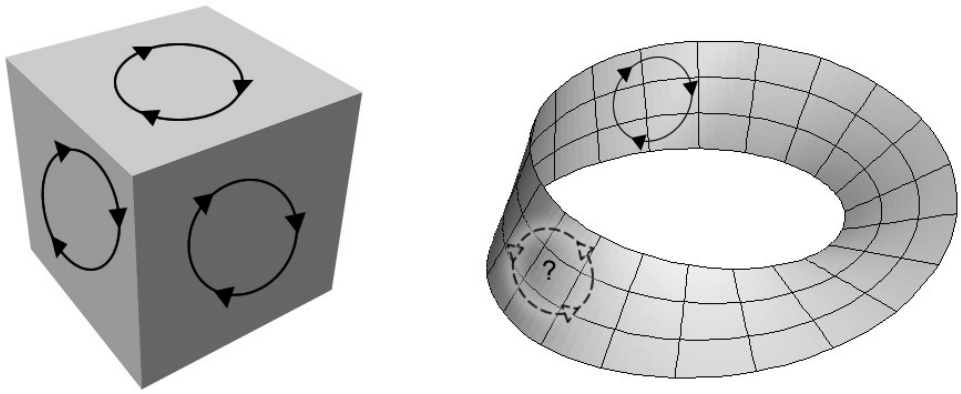

Figura 2.9 – Orientabilidade do cubo e da fita de Möbius.

Perceba que todas as faces do cubo podem ser orientadas consistentemente – por exemplo, no sentido horário – enquanto que a fita de Möbius, *por ter apenas um lado*, ao receber um sentido ao longo de sua superfície, fatalmente acaba retornando no sentido contrário.

A fita de Möbius é um 2-manifold não-orientável, pois qualquer área local de sua superfície pode ser reduzida a um plano bidimensional homeomórfico ao disco. No

entanto, se você parar para pensar a respeito de qualquer localidade da fita, perceberá que é possível passar de um lado do plano ao outro sem atravessar a superfície e sem ultrapassar a borda do objeto – acompanhe a fita, rodando o dedo no sentido horário, até voltar ao mesmo ponto, e você verá que o seu dedo retorna girando no sentido anti-horário!

Dessa propriedade de orientabilidade, é possível concluir que, no caso dos 2-manifolds orientáveis, as faces conectadas a uma mesma aresta terão, relativas a essa aresta, direções opostas, como mostra a figura a seguir:

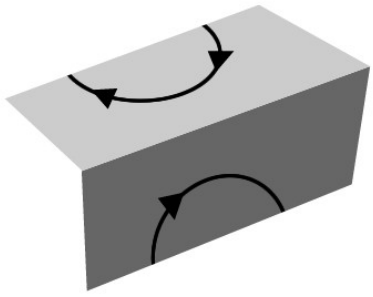

Figura 2.10 – Duas faces conectadas a uma mesma aresta com direções opostas (do ponto de vista da aresta).

2.1.6. Superfície – Definição

Apesar de estarmos usando livremente o termo superfície, chegou o momento de defini-lo. De acordo com Sieradsky, uma superfície é uma estrutura topológica *2-manifold, compacta e conectada*.

O conceito de 2-manifold já foi explicado, então vamos passar para os outros conceitos.

Intuitivamente falando, uma superfície é *compacta* quando ela ocupa uma região finita e limitada em um espaço n-dimensional. Uma esfera, por exemplo, é uma

superfície compacta, pois ocupa uma região finita e limitada no espaço tridimensional; já um cilindro que estenda infinitamente suas extremidades ao longo de seu eixo longitudinal não é uma superfície compacta, pois é infinita em uma de suas dimensões.

O conceito de *conexão* também pode ser entendido intuitivamente. Olhando uma superfície composta por faces poligonais, podemos dizer que ela é conectada quando todas as arestas de todas as faces pareiam com arestas de outras faces quaisquer. Entender uma superfície poligonal como um união disjunta de polígonos, como mostra a figura, ajuda a entender a questão:

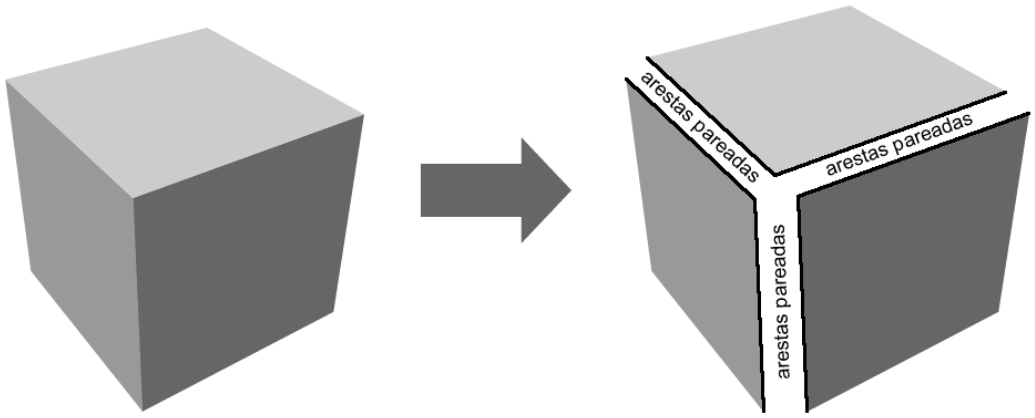

Figura 2.11 – Uma superfície poligonal vista como um conjunto de faces separadas (para facilitar a compreensão, apenas as três faces da frente são mostradas).

Ao separar as faces, fica visível o fato de que elas pareiam as suas arestas com as arestas das faces vizinhas – na prática, nós não vamos tratar de arestas, e sim de meias-arestas, pois elas formam o núcleo da estrutura Half-Edge.

2.1.7. Modelos poligonais como poliedros

De acordo com Hanrahan (1982), um dos objetivos da modelagem sólida é encontrar métodos econômicos de armazenamento de volumes 3D – nesse sentido, armazenar todos os pontos de uma superfície não é viável, e o uso de superfícies paramétricas tornaria muito complexa a criação de determinados sólidos.

É por isso que os modelos representados pelas B-Reps são aproximados por poliedros – assim, é possível regular o grau de aproximação entre o modelo e o objeto real aumentando ou diminuindo a quantidade de faces, arestas e vértices armazenados (a figura 2.12 destaca esses três elementos).

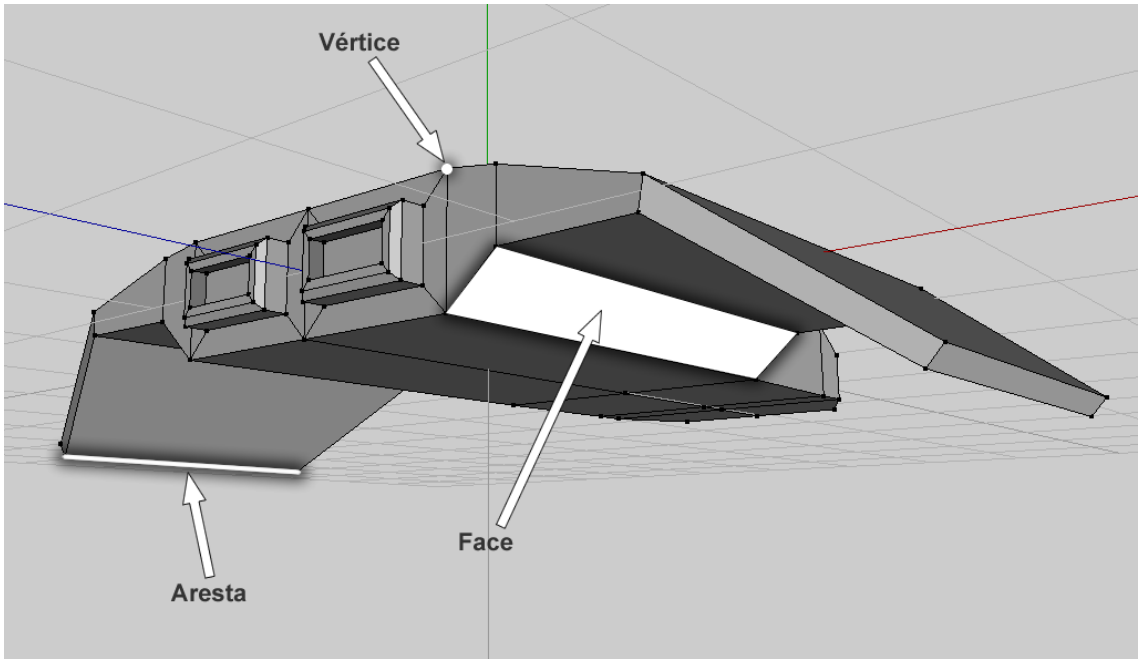

Figura 2.12 – Subcomponentes das B-reps.

Restringir os sólidos que poderemos construir, considerando-os poliedros 2-manifold, compactos e conectados, permite validar matematicamente os sólidos representados pela estrutura Half-Edge por meio de uma propriedade denominada *característica de Euler*, que estudaremos a seguir.

2.1.8. Característica de Euler (fórmula do poliedro)

A característica de Euler é uma invariante topológica, ou seja, é um valor constante para determinadas famílias de poliedros. A fórmula geral para a característica é:

$$x = V - E + F$$

Para poliedros convexos, temos:

$$V - E + F = 2$$

2.1.9. Fórmula de Euler-Poincaré

O matemático francês Henri Poincaré foi o responsável por generalizar a fórmula de Euler, incluindo superfícies que possuam buracos. A fórmula generalizada fica assim:

$$V - E + F - R = 2S - 2H$$

Em que:

- V = nº de vértices
- E = nº de arestas
- F = nº de faces
- R = nº de rings (loops internos)
- S = nº de *shells* (superfícies)
- H = nº de buracos (*holes* ou *genus*)

Levando em conta que

$$R = L - F$$

com L indicando o número total de loops da superfície, podemos substituir o R da equação, ficando com:

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

$$V - E + 2F - L = 2S - 2H$$

É preferível trabalhar com loops em vez de *rings*, pois o loop é um conceito nativo da estrutura Half-Edge.

Como iremos trabalhar apenas com um *shell*, podemos fazer S = 1, chegando na seguinte fórmula:

$$V - E + 2F - L = 2 - 2H$$

2.1.10. Operadores de Euler

Os operadores de Euler manipulam a superfície de um poliedro, mantendo intacta a característica de Euler. Existem inúmeros operadores – no entanto, foi definido um subconjunto de dez operadores que atendem a todas as necessidades de manipulação em modelagem sólida: cinco operadores construtivos e cinco destrutivos.

De acordo com Kettner, os operadores de Euler só funcionam adequadamente com 2-manifolds orientáveis. Isso faz sentido, pois tanto os non-manifolds quanto os 2-manifolds não-orientáveis não respeitam a característica de Euler – mas é importante ressaltar que, apesar dos operadores gerarem sólidos topologicamente consistentes, nem todos os sólidos são geometricamente consistentes.

Para concluir a discussão sobre topologia, podemos concluir que iremos trabalhar com superfícies 2-manifold, compactas, conectadas e orientáveis, que serão constituídas basicamente de faces, vértices e pares de meias-arestas.

2.2. Geometria

A geometria – ou seja, a forma espacial – de uma representação B-Rep depende diretamente da topologia de sua superfície. Não basta saber as coordenadas dos vértices no espaço 3D – é preciso saber, também, como os vértices se conectam entre si, gerando

as arestas e as faces. Para entender melhor o porquê disso, observe a figura a seguir, que mostra como o mesmo grupo de vértices pode gerar superfícies completamente diferentes, dependendo de como eles são conectados:

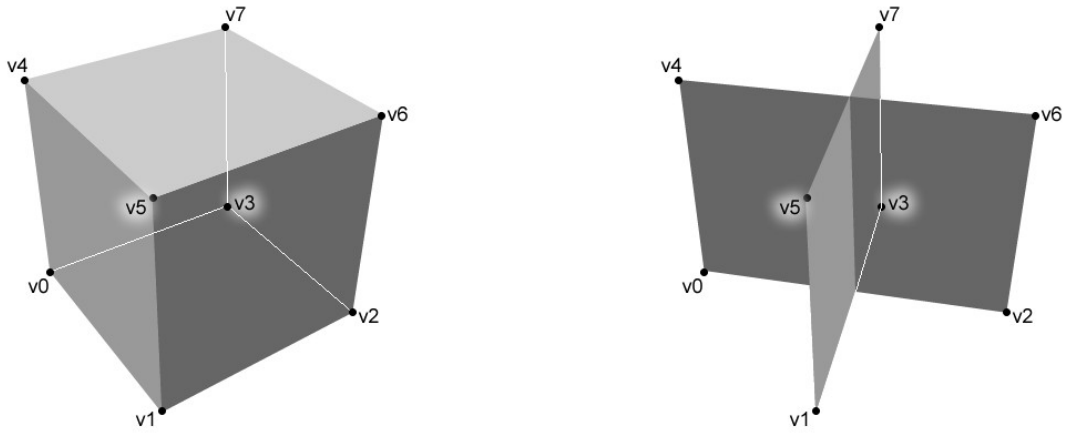

Figura 2.13 – Superfícies diferentes geradas com o mesmo grupo de vértices.

Mas será que conhecer apenas a conexão entre os vértices é o suficiente para podermos manipular uma superfície? Para responder essa pergunta, precisamos olhar mais de perto como os sólidos são representados no computador.

2.2.1. Vertex Buffer / Index Buffer

Um modelo 3D é representado, no nível do hardware, como um conjunto de vértices e de conexões entre esses vértices. Para isso, são usados dois vetores, geralmente denominados *Vertex Buffer* e *Index Buffer*.

O Vertex Buffer armazena as coordenadas dos vértices, enquanto o Index Buffer armazena as conexões entre os vértices – para entender como esse processo funciona, imagine o seguinte Vertex Buffer, descrito em pseudo-C++:

```
VertexBuffer *vertexBuffer = new VertexBuffer({
{-1.0, -1.0, -1.0},
{ 1.0, -1.0, -1.0},
{ 1.0, -1.0,  1.0},
{-1.0, -1.0,  1.0},
{-1.0,  1.0, -1.0},
```

```
{ 1.0,  1.0, -1.0},
{ 1.0,  1.0,  1.0},
{-1.0,  1.0,  1.0}
});
```

A classe VertexBuffer recebe como parâmetro de construção um array bidimensional, em que cada linha representa as coordenadas de um vértice. Se quisermos, por exemplo, criar um triângulo usando os três primeiros vértices, devemos inserir um trio no vetor indexBuffer da seguinte forma:

```
IndexBuffer *indexBuffer = new indexBuffer({{ 0, 2, 1 }});
```

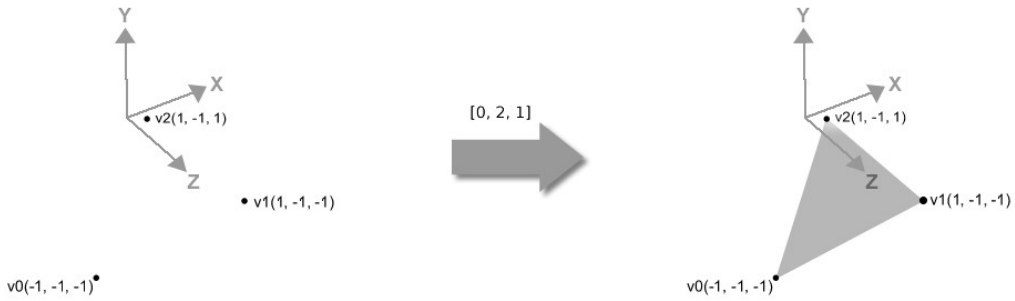

Figura 2.14 – Formação de um triângulo.

A ordem de inserção dos vértices no Index Buffer é importante – no caso do DirectX, eles devem ser inseridos no sentido horário, caso contrário, as faces não serão desenhadas no sentido esperado.

Para criar um cubo usando os vértices inseridos no Vertex Buffer, é preciso criar um Index Buffer semelhante ao mostrado a seguir:

```
IndexBuffer *indexBuffer = new indexBuffer({
{0, 1, 2},
{0, 2, 3},
{0, 4, 5},
{0, 5, 1},
{1, 5, 6},
{1, 6, 2},
{0, 3, 7},
{0, 7, 4},
{2, 7, 3},
{2, 6, 7},
{4, 6, 5},
{4, 7, 6}
});
```

Acompanhe a figura para visualizar a criação dos triângulos:

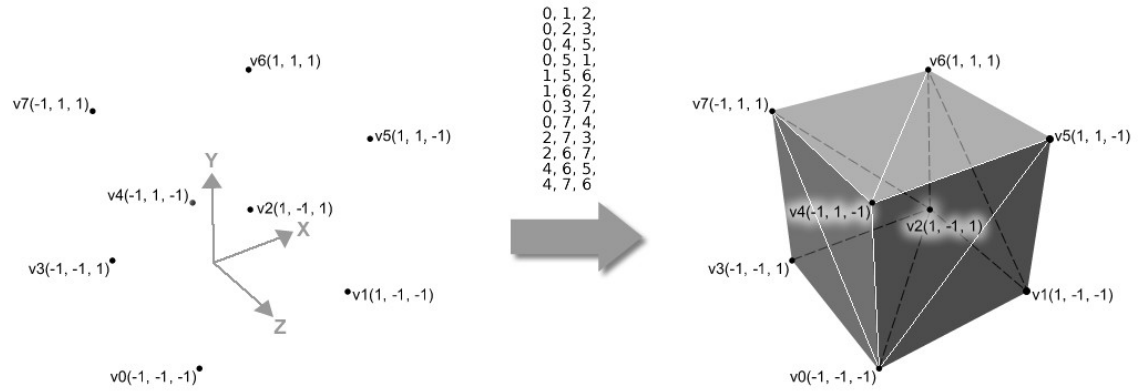

Figura 2.15 – Conexão dos pontos, formando um cubo.

Repare que um mesmo vértice pode fazer parte de vários trios ao mesmo tempo.

Do ponto de vista do hardware, esse tipo de representação é eficiente, pois trabalha com o mínimo de informações possível para se desenhar o modelo 3D na tela. No entanto, existem alguns pontos que devem ser observados:

1. Os índices relacionam os vértices três a três para criar as faces, mas não existe relação entre as faces;
2. As faces, do ponto de vista do hardware, devem possuir três vértices – no caso de um cubo, por exemplo, que possui faces com quatro vértices, é preciso usar duas faces triangulares para criar cada uma das faces quadradas do cubo.
3. Não existe o conceito de aresta – existem apenas faces e vértices.

Imagine, agora, que criamos um modelador 3D baseado apenas nos conceitos de Index Buffer e Vertex Buffer – a figura a seguir mostra um cubo criado inicialmente pelo programa:

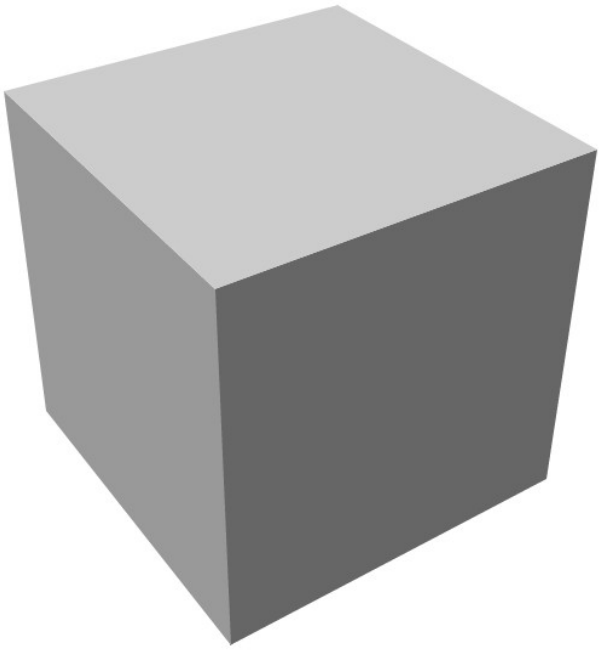

Figura 2.16 – Cubo inicialmente criado pelo modelador fictício.

Suponha que queiramos selecionar a face de cima do cubo – a figura a seguir mostra o que acontece quando clicamos na face superior do cubo:

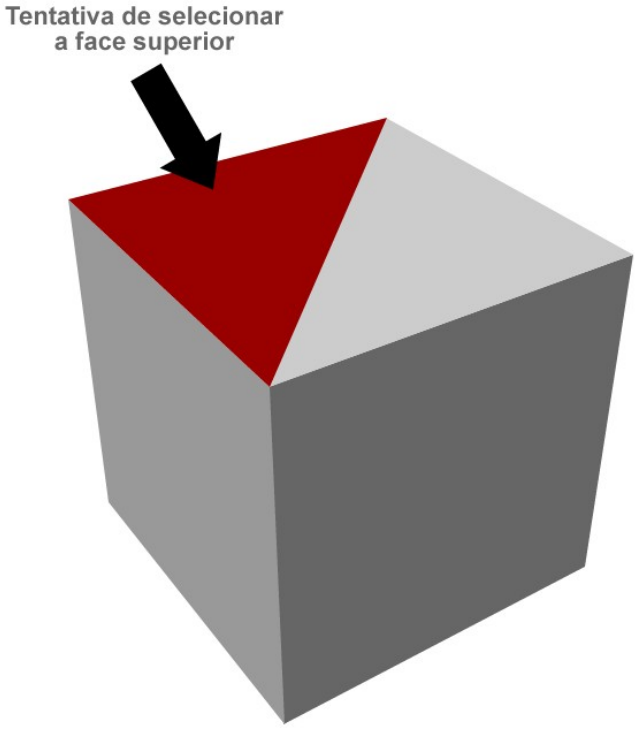

Figura 2.17 – Tentativa de selecionar a face superior do cubo.

Como cada face do cubo é composta, na realidade, de duas faces triangulares, ao clicar em uma das faces do cubo você estará selecionando uma das faces triangulares,

pois, como foi observado, uma face, do ponto de vista do hardware, deve possuir três vértices.

Esse problema pode ser contornado caso o modelador possua alguma maneira de selecionar múltiplas faces – supondo que o nosso modelador possua essa facilidade, selecionamos os dois triângulos da face superior do cubo:

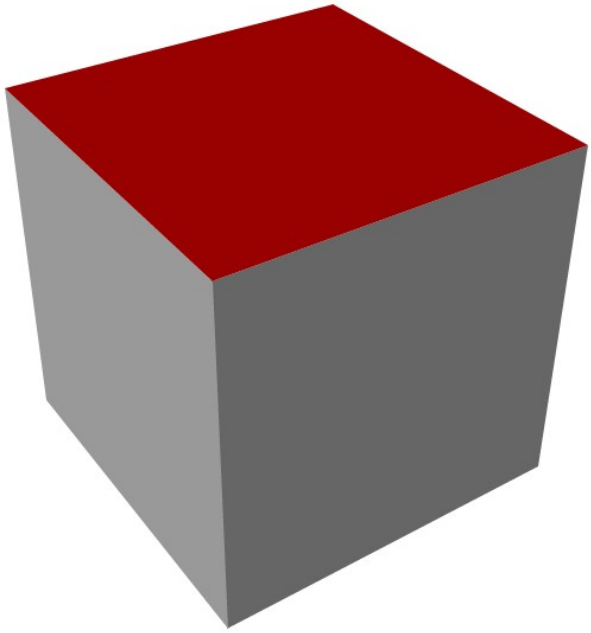

Figura 2.18 – Os dois triângulos da face superior estão selecionados.

Agora que a face superior está completamente selecionada, suponha que você queira movimentá-la para cima, deformando o cubo – veja na figura a seguir o que acontecerá caso você faça isso:

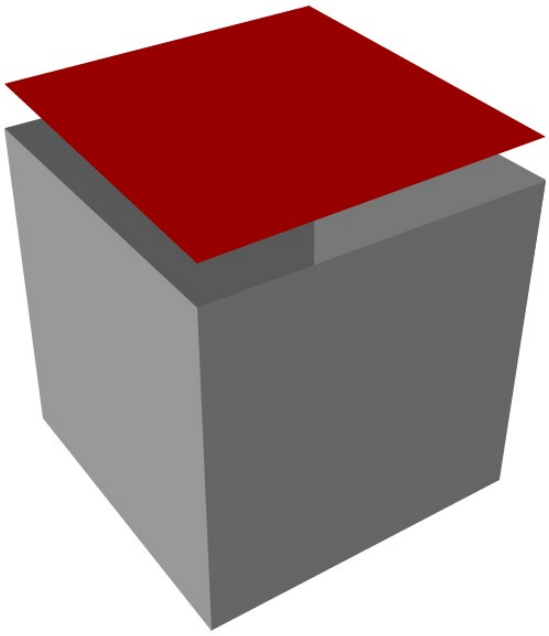

Figura 2.19 – Resultado da movimentação da face superior.

Intuitivamente, sabemos que algo está errado – a idéia era movimentar a face, mas mantendo o sólido íntegro! A primeira observação feita anteriormente explica o ocorrido: como não existe relação entre as faces, o modelador não sabe que deveria movimentar também as faces que compartilham os vértices das faces selecionadas.

O objetivo desse exemplo foi demonstrar a fragilidade do modelo Index Buffer / Vertex Buffer no processo de modelagem 3D: do ponto de vista desse modelo, não existe um sólido, e sim um aglomerado de faces que se tocam em suas bordas, dando a impressão de um sólido!

Podemos concluir que essa representação não é suficiente para um modelador 3D – precisamos, então, buscar uma estrutura de dados alternativa que nos dê flexibilidade suficiente para realizar as ações de manipulação de um sólido.

2.2.2. A estrutura Winged-Edge

Em 1975, Bruce Baumgart publicou o artigo *"Winged-Edge Polyhedron Representation for Computer Vision"*. Nesse artigo, ele apresenta a estrutura de dados Winged-Edge, mostrada na figura a seguir:

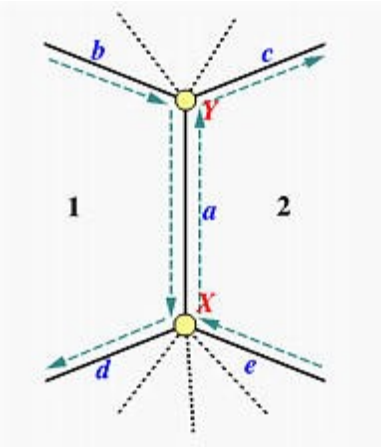

Figura 2.20 – A estrutura Winged-Edge.

Partindo da premissa de que um poliedro é formado por vértices, arestas e faces, a estrutura conta com esses três elementos como sendo os constituintes básicos de um sólido. Como vimos no tópico anterior, para manipular um objeto 3D de forma correta e flexível, é preciso armazenar informações de conectividade entre os diversos elementos básicos que constituem um sólido – em outras palavras, é preciso armazenar as suas informações topológicas.

A estrutura Winged-Edge pretende suprir essa necessidade de informações topológicas da seguinte maneira:

- Um vértice mantém uma referência para uma aresta à qual esteja conectado;
- Uma face mantém uma referência para uma aresta à qual esteja conectado;
- Uma aresta mantém referências para os dois vértices aos quais está conectado, para as duas faces adjacentes a ela e para quatro arestas adjacentes.

Com essas dez referências de conectividade, é possível percorrer toda a topologia de um sólido, dado um vértice, uma face ou uma aresta qualquer.

Apesar de todas as vantagens da Winged-Edge, existe um problema que ela não contempla: se eu tiver uma referência a uma aresta, como saber qual é o sentido que devo seguir? Afinal, a estrutura não guarda referências de sentido de movimento, e é isso que a estrutura Half-Edge busca solucionar [Kettner, 1998].

2.2.3. Half-Edge: a evolução da Winged-Edge

A grande diferença entre as estruturas Winged-Edge e Half-Edge é a divisão das arestas em duas meias-arestas. Assim, cada meia-aresta armazena metade das referências armazenadas na aresta, ou seja: um vértice, uma face e duas meias-arestas (anterior e posterior).

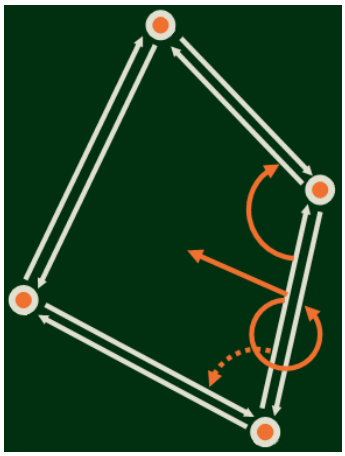

Figura 2.21 – Representação de uma meia-aresta e suas conexões.

Perceba que uma meia-aresta está contida em uma face, e que ela aponta apenas para as meias-arestas que fazem parte dessa mesma face (além da meia-aresta oposta). Assim, fica mais fácil percorrer os loops internos das faces, que é uma das operações mais importantes tanto da Winged-Edge quanto da Half-Edge.

Você se lembra da questão de selecionar uma das faces do cubo e movimentá-la usando a representação via Vertex Buffer / Index Buffer? Pois bem: com a representação Half-Edge, temos as seguintes vantagens:

- As faces podem ser selecionadas como um todo, pois a estrutura Half-Edge não impõe que as faces devem conter apenas três lados;
- As faces adjacentes à selecionada também são movimentadas, pois existem conexões entre elas (via meias-arestas).

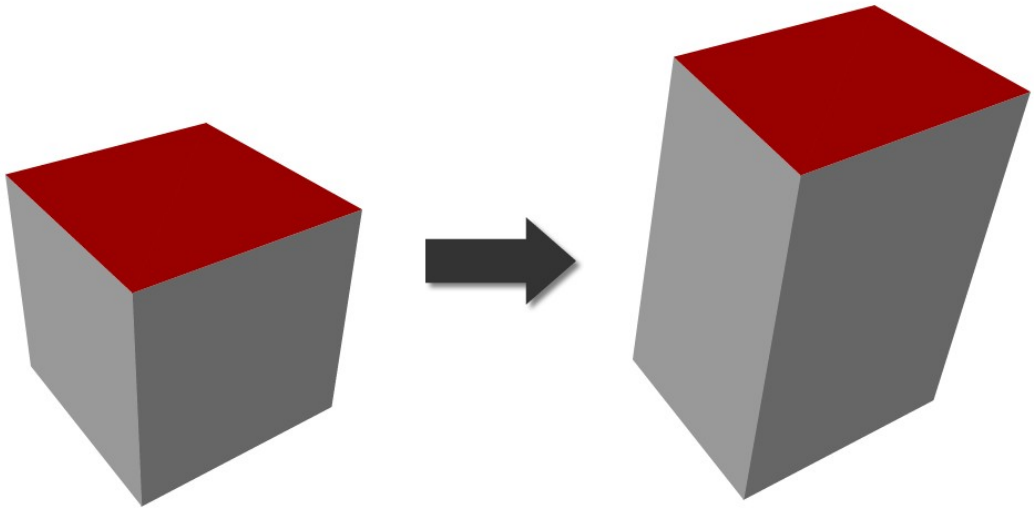

Figura 2.22 – Movimentando uma face e suas adjacências.

A seguir, apresento dois exemplos de como certos problemas são solucionados pela estrutura Half-Edge.

O primeiro exemplo, demonstrado na figura a seguir, mostra o que deve ser feito para saber, partindo de uma meia-aresta, se outra aresta está conectada a um mesmo vértice:

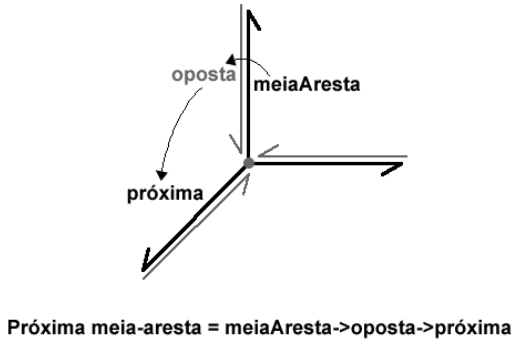

Figura 2.23 – Uma aresta está conectada a um vértice?

Essa técnica de selecionar a próxima meia-aresta conectada a um vértice é fundamental para a construção do algoritmo de certos operadores de Euler (MEV, por exemplo).

O segundo exemplo mostra os passos necessários para descobrir, partindo de uma face, se uma aresta faz parte de um dos loops da face:

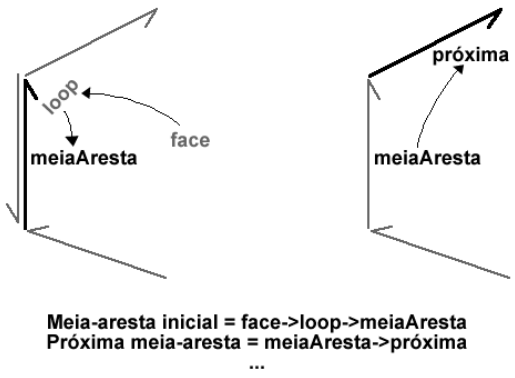

Figura 2.24 – Uma aresta está conectada a uma face?

Tendo em vista que a estrutura Half-Edge possui informações mais precisas de orientação dos loops, justamente por trabalhar com meias-arestas, ela é mais indicada para ser utilizada como estrutura de armazenamento de sólidos 3D em softwares de modelagem, pois as operações de seleção de elementos (faces, arestas ou vértices) dependem fortemente de encontrar os elementos desejados por meio da orientação dos

mesmos, que, em certos casos, é deficiente ou complicado de realizar na estrutura Winged-Edge.

A partir do próximo capítulo, começaremos a dar forma à estrtura Half-Edge, primeiro desenvolvendo as linhas gerais de raciocínio, para, em seguida, criar uma implementação da estrutura em C++.

Capítulo 3

Estrutura Half-Edge – Projeto

A estrutura Half-Edge é composta, inicialmente, por cinco componentes: sólidos, faces, loops, meia-arestas e vértices. Mostrarei um processo de desenvolvimento baseado, principalmente, nos trabalhos de Martti Mantyla e Xianming Chen, mas que apresentará características originais, com o objetivo de suprir certas deficiências encontradas por mim ao longo do caminho.

Antes de entrarmos na implementação da estrutura Half-Edge, vale a pena pararmos para criar um mini-projeto dela – assim, ficará mais fácil visualizar o que queremos construir e quais os objetivos que pretendemos alcançar.

3.1. Estrutura de classes

Em primeiro lugar, vamos olhar as classes básicas da estrutura e as relações entre elas:

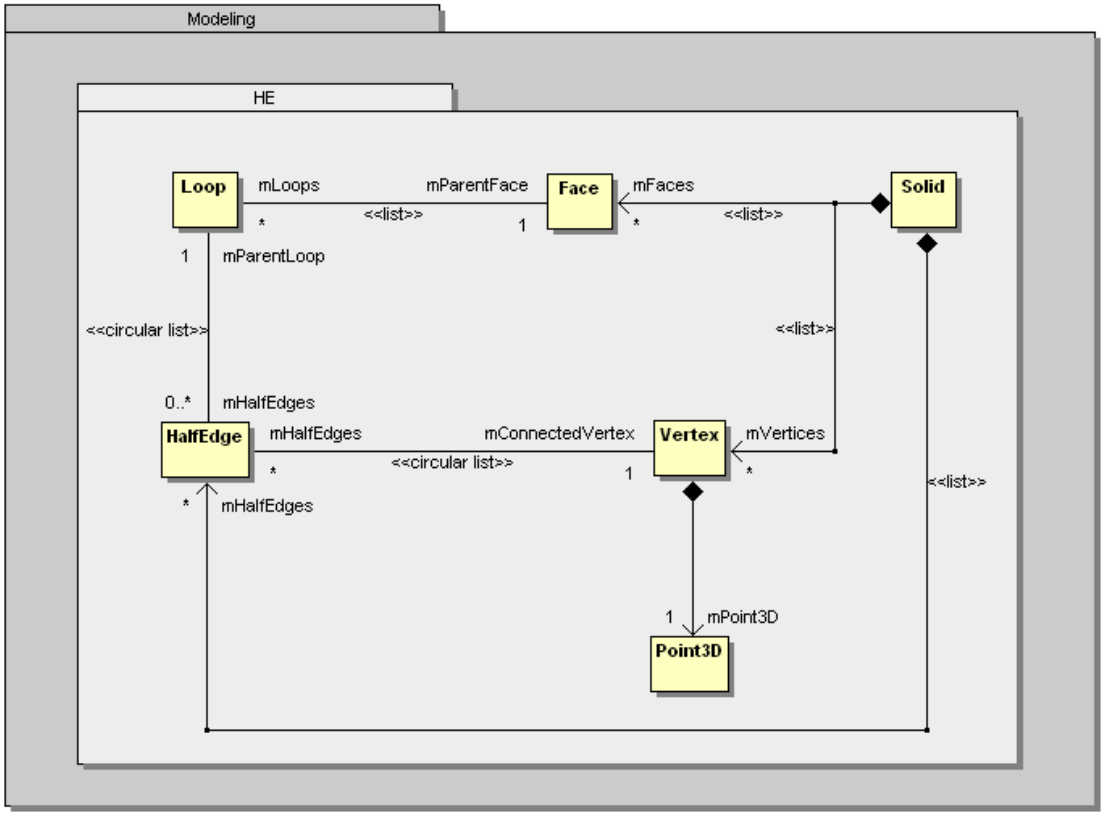

Figura 3.1 – Diagrama de classes da estrutura Half-Edge.

Nota: a classe *HalfEdge* representa uma meia-aresta e não deve ser confundida com a estrutura geral Half-Edge.

3.1.1. Classe *Solid*

A classe *Solid* é a classe básica da estrutura – é por meio dela que todas as outras classes são instanciadas. Ela possui três listas circulares: uma de faces, uma de meias-arestas e outra de vértices – essas listas são úteis, por exemplo, para descobrirmos se um clique do mouse ocorreu sobre um dos elementos a fim de selecioná-lo.

3.1.2. Classes *Face* e *Loop*

A classe *Face* contém uma lista linear de loops; a classe *Loop*, por sua vez, possui uma lista circular de meias-arestas que a compõem – a figura a seguir mostra tais conexões:

Figura 3.2 – Conexões Face / Loop inicial (esq.) e Loop / HalfEdge inicial (dir.).

3.1.3. Classe *Vertex*

A classe *Vertex*, assim como a classe *Loop*, possui uma lista circular de meias-arestas conectadas a ela:

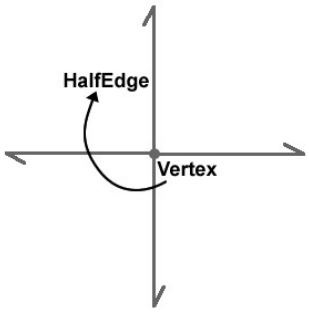

Figura 3.3 – Conexão Vertex / HalfEdge inicial.

De acordo com o diagrama de classes exposto anteriormente, ambas as classes *Loop* e *Vertex* armazenam uma lista circular de meias-arestas – mas onde estão as listas circulares nessas figuras?

Na verdade, as figuras mostram apenas a referência ao elemento inicial de suas listas, não importando se são lineares ou circulares – portanto, interprete as referências como apontadores para a cabeça de cada lista.

3.1.4. Classe HalfEdge

A classe *HalfEdge* possui uma referência tanto para o loop do qual ela faz parte, quanto para o vértice conectado a ela. O vértice ao qual uma meia-aresta está conectada pode ser tanto o vértice de partida quanto o vértice de chegada, como mostra a figura a seguir:

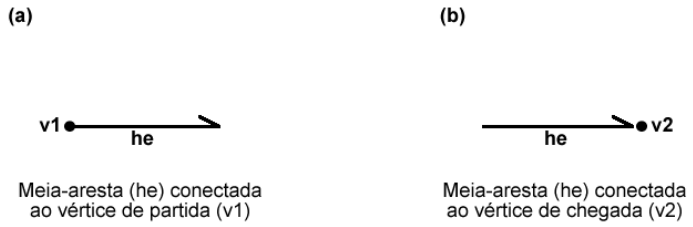

Figura 3.4 – Meia-aresta conectada ao vértice de partida (a) e ao vértice de chegada (b).

Não há motivos óbvios para escolher uma opção em detrimento da outra – ambas são equivalentes. No entanto, é necessário convencionar qual opção será usada, pois a implementação dos algoritmos será levemente modificada dependendo da convenção escolhida.

Para esse trabalho, foi escolhida a opção (a) da figura anterior, ou seja, vamos convencionar que a meia-aresta está conectada ao vértice de partida.

A figura a seguir mostra as conexões básicas da classe *HalfEdge*:

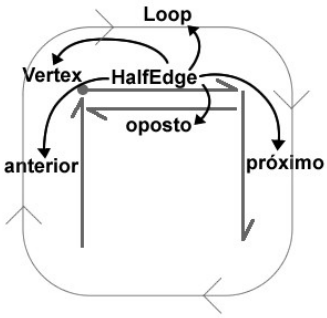

Figura 3.5 – Conexões da classe HalfEdge.

Perceba que toda meia-aresta aponta para três outras meias-arestas: a oposta, a anterior, dentro do mesmo loop, e a próxima, também dentro do mesmo loop.

3.1.5. Estrutura Half-Edge – Visão geral

Agora que definimos isoladamente as conexões de cada elemento, vamos unificá-las para podermos ter uma visão global da estrutura:

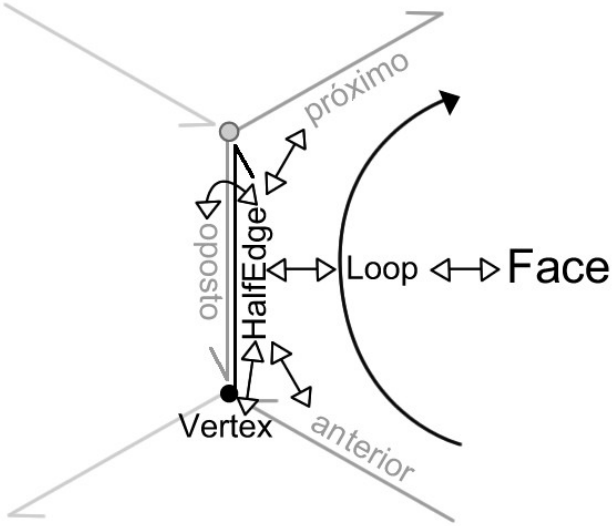

Figura 3.6 – Visão global das conexões da estrutura Half-Edge.

3.2. Operadores de Euler

Vamos, então, continuar projetando a estrutura, adicionando ao diagrama de classes os operadores de Euler que serão implementados pelo sistema. Para restringir

um pouco o escopo deste trabalho, assumirei explicitamente a seguinte posição: a estrutura Half-Edge descrita tem como objetivo manipular o sólido *apenas de forma construtiva*, ou seja, serão implementados apenas os operadores que, de certa forma, contribuam para a construção do sólido, e não para a sua destruição.

Os cinco operadores a serem implementados são:

- MVFLS;
- MEV;
- MEFL;
- KEML;
- KFMH;

Se quiséssemos implementar as modificações destrutivas, teríamos que implementar, também, os seguintes operadores, que são os inversos exatos dos cinco operadores apresentados anteriormente:

- KVFLS;
- KEV;
- KEFL;
- MEKL;
- MFKH;

É importante perceber o seguinte: a essência construtiva ou destrutiva de um operador não pode ser inferida diretamente de seu nome – se assim fosse, KEML deveria ser visto como um operador destrutivo, pois inicia destruindo algo para construir outo elemento em seguida. A verdadeira forma de compreensão dos operadores está no conjunto dos elementos criados ou destruídos – no caso de KEML, significa que o operador deve destruir uma aresta para, em seguida, criar um loop.

3.2.1. Operadores de Euler – Diagrama de Classes

A Figura a seguir mostra o diagrama de classes com as operações inseridas:

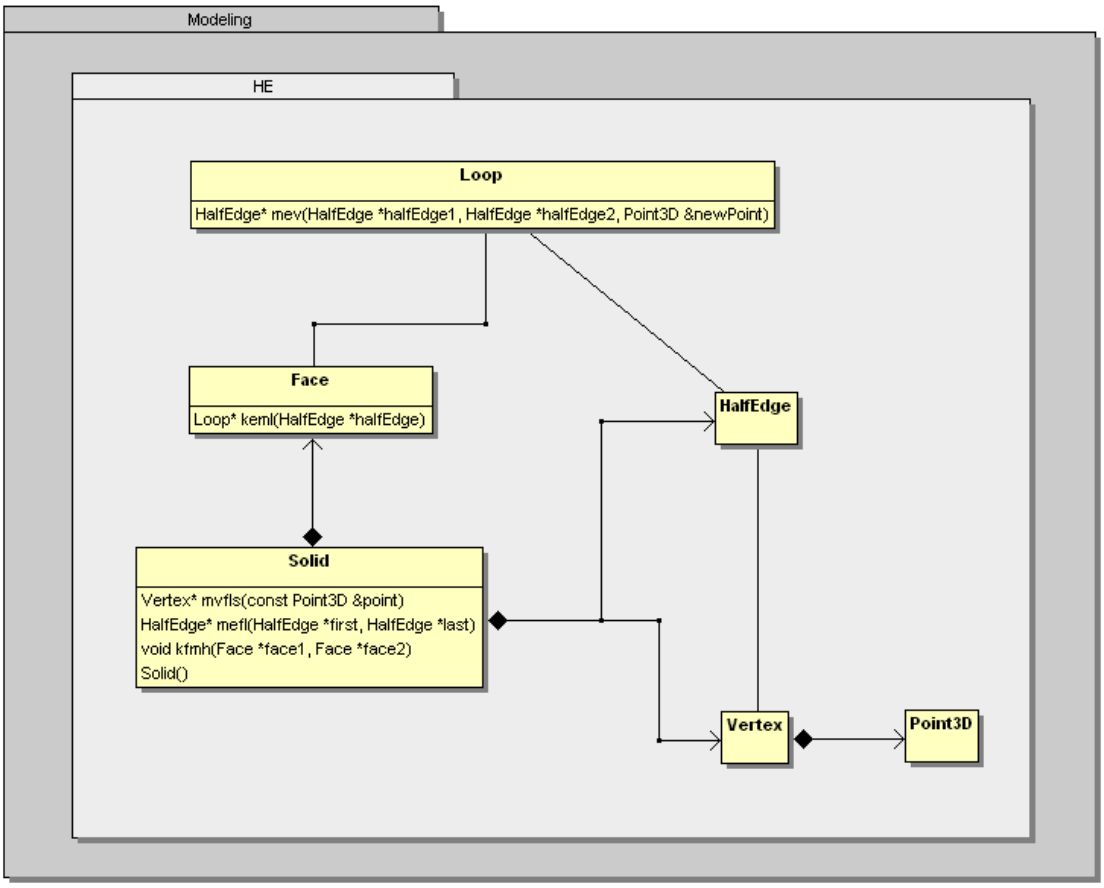

Figura 3.7 – Diagrama de Classes: operadores de Euler inseridos.

Desse diagrama podemos inferir que:

- a classe *Solid* é responsável por três operadores (*mvfls*, *mefl*, *kfmh*),
- a classe *Face* é responsável por um operador (*keml*), e
- a classe *Loop* também é responsável por um operador (*mev*).

Algumas implementações da estrutura Half-Edge adicionam os operadores *keml* e *mev* à classe *Solid*, com o objetivo de facilitar o acesso aos operadores das classes Face e Loop – Xianming Chen é um deles. No entanto, usaremos uma abordagem um pouco diferente dos operadores de Euler.

3.2.2. Macro-operadores de Euler

Como já foi dito anteriormente, a aplicação dos operadores de Euler mantém intacta a invariante topológica denominada *característica de Euler*; no entanto, também foi dito que, mesmo que o sólido permaneça *topologicamente* válido, não significa que ele seja *geometricamente* válido – tome como exemplo a aplicação do operador MEV, tal como mostrado na figura a seguir:

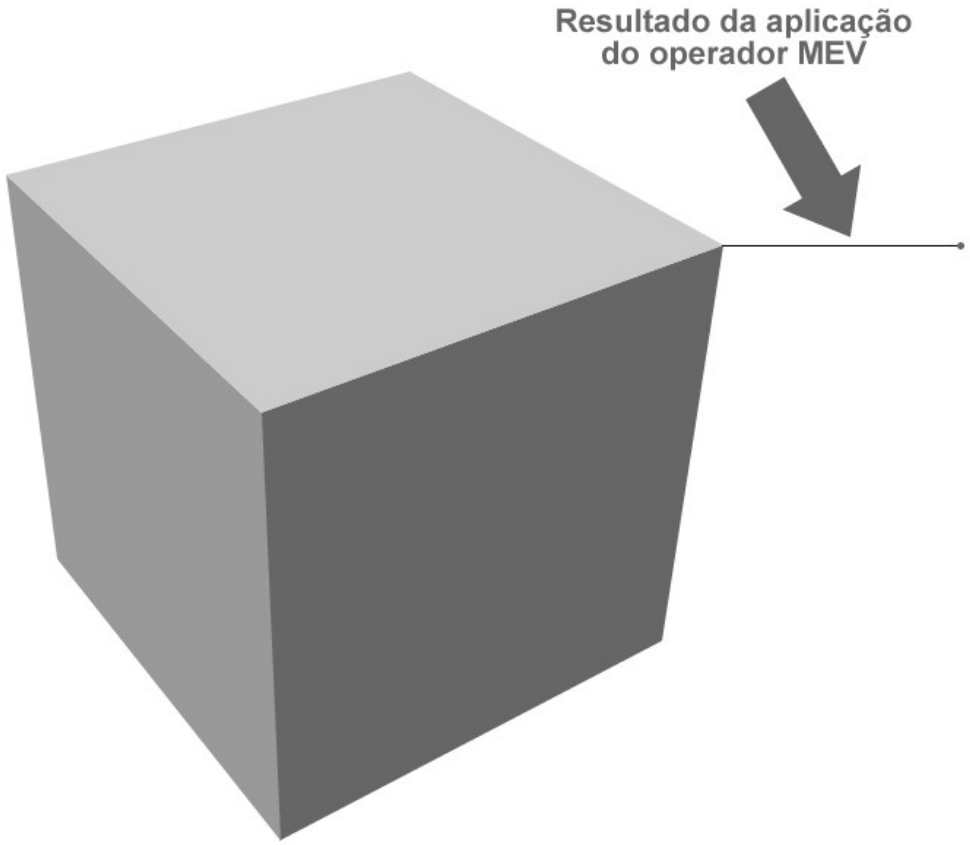

Figura 3.8 – Operador MEV aplicado a um vértice do cubo.

O cubo com uma aresta pendurada é um elemento topologicamente válido, mas deixou de ser um 2-manifold, fugindo, portanto, do conceito de superfície que definimos no capítulo anterior.

Uma maneira de evitar que isso ocorra é proibir o acesso direto aos operadores de Euler, dando ao programador uma interface que contenha apenas *macro-operadores*

que não infrinjam as regras e restrições impostas por nós.

Um macro-operador de Euler é uma sequência finita de operadores de Euler, executados de tal forma que não infrinjam o conceito topológico de superfície. Um macro-operador muito comum é o *chanfro*, mostrado na figura a seguir:

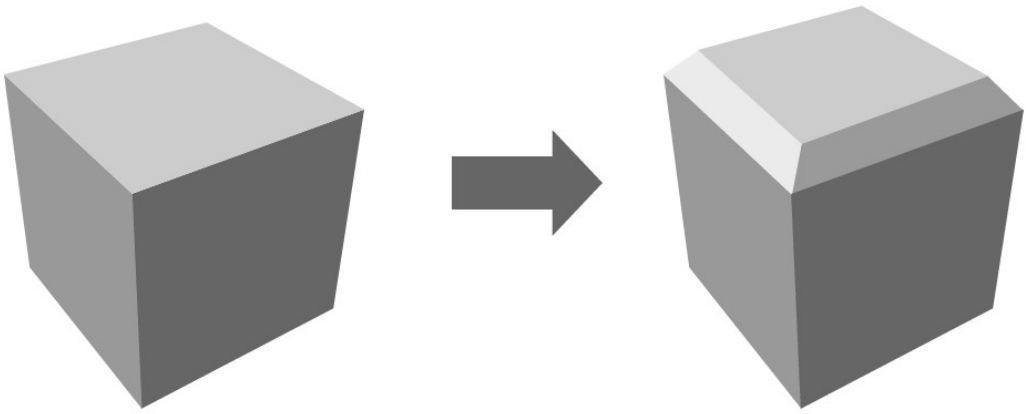

Figura 3.9 – Chanfro aplicado a uma face do cubo.

No caso da figura anterior, o chanfro consiste na seguinte sequência de operadores de Euler: MEV, MEV, MEV, MEV, MEFL, como mostra a figura a seguir:

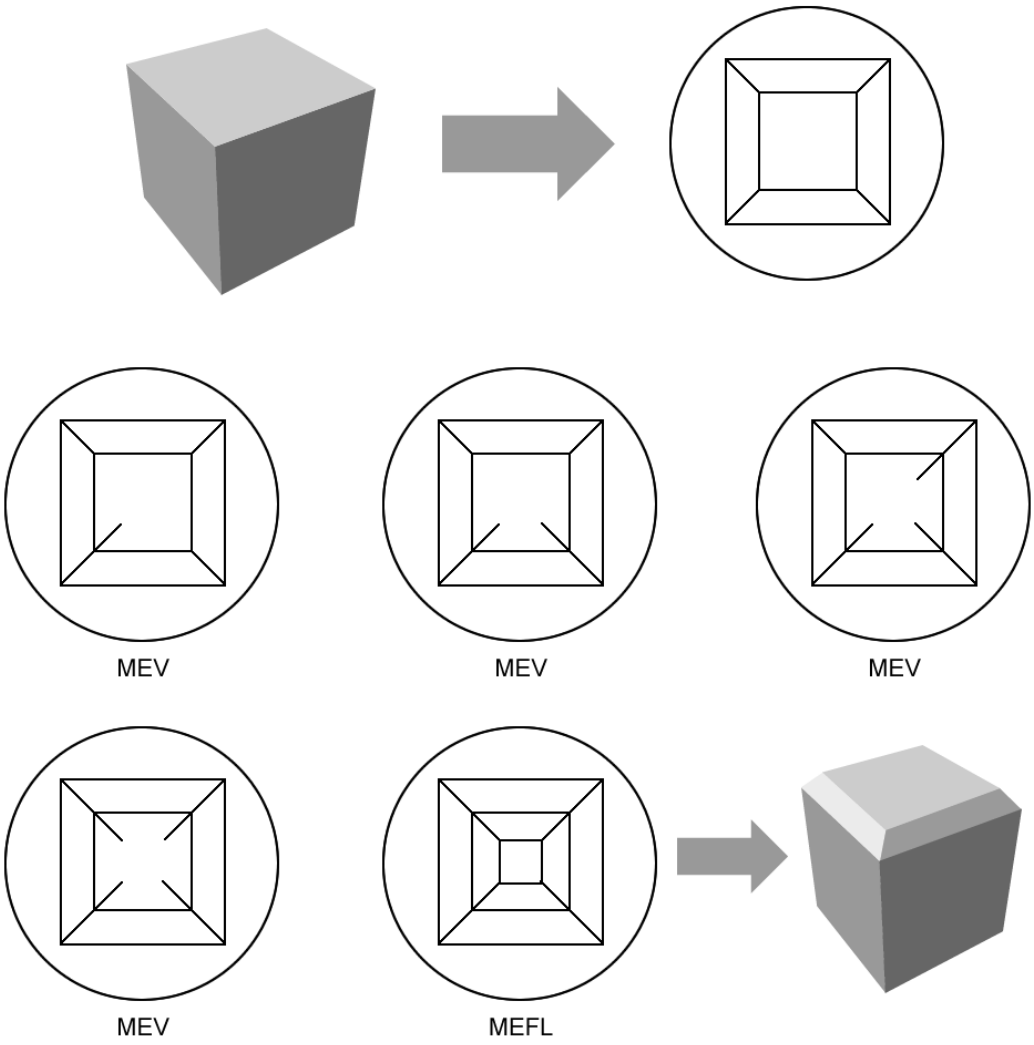

Figura 3.10 – Grafo planar mostrando a aplicação do chanfro.

Usando macro-operações como o chanfro, é possível garantir tanto a validade topológica quanto a validade geométrica da superfície modelada. No entanto, a criação dos macro-operadores está fora do escopo deste trabalho, ficando como um exercício posterior para o leitor.

Para mantermos o trabalho simples, os operadores de Euler poderão ser diretamente acessados – faremos a suposição de que o programador será criterioso na aplicação dos operadores.

E quanto à criação da superfície inicial do sólido – deverá ser feita pelo

programador que utilizará a estrutura ou, tal como as macro-operações, deverá esconder a aplicação dos operadores de Euler? Em minha opinião, a segunda opção é melhor, apesar de ser mais restritiva.

Se permitirmos a aplicação de qualquer sequência de operadores de Euler – partindo sempre de *mvfls*, pois, caso contrário, nem teremos um sólido para manipular! – é possível criar qualquer tipo de sólido inicial, correndo o risco, inclusive, de criar sólidos que não se enquadrem na definição de superfície (2-manifold, orientável, compacta, conectada). No entanto, na prática, o processo de modelagem 3D costuma partir sempre de objetos primitivos – cubos, esferas, cilindros etc. – que são deformados e manipulados para assumir a forma desejada. Por isso, sustento a opinião de que não há a necessidade de começar a criação do sólido a partir do nada – basta dar algumas opções de superfícies primitivas, das quais uma será escolhida como topologia inicial.

Para manter o trabalho simples, a classe *Solid* terá apenas o cubo como superfície primitiva – a escolha do tipo de primitiva será feita no momento da instanciação da classe, por meio de um parâmetro passado ao construtor do objeto.

Capítulo 4

Estrutura Half-Edge – Implementação

Agora que vimos a ideia geral por trás da estrutura Half-Edge, chegou a hora de implementá-la – comecemos, então, criando a estrutura básica de cada classe.

4.1. Classe *Solid*

A classe *Solid* é a classe-mãe da estrutura Half-Edge, engoblando e unificando os demais elementos. Ela possui uma lista circular de faces, uma de arestas e outra de vértices, bem como o próximo valor de identificação de cada um dos subcomponentes (faces, meias-arestas e vértices).

```
namespace Modeling
{
   enum PrimitiveType
   {
      PT_CUBE = 0
   };

   namespace HE
   {
      class Solid
      {
      public:
         Solid(PrimitiveType primitiveType = PT_CUBE);
         ~Solid();

         void        kfmh(Face *face1, Face *face2);
         HalfEdge*   mefl(HalfEdge *first, HalfEdge *last);
         Loop*       mvfls(const Point3D &point);

         void        decreaseEdgeCount() { mEdgeCount--; }
         void        deleteFace(Face *face);
         Face*       getFaceById(int id);
         Face*       getHeadFace() { return mFaces; }
         Vertex*     getHeadVertex() { return mVertices; }
```

```
            int         getEdgeCount() { return mEdgeCount; }
            int         getFaceCount() { return mFaceCount; }
            int         getNextEdgeId() { return mNextEdgeId; }
            int         getNextFaceId() { return mNextFaceId; }
            int         getNextVertexId() { return mNextVertexId; }
            int         getVertexCount() { return mVertexCount; }
            void        increaseEdgeCount() { mEdgeCount++; }
            void        increaseLoopCount() { mLoopCount++; }
            void        increaseVertexCount() { mVertexCount++; }
            void        insertIntoFaceList(Face *face);
            void        insertIntoVertexList(Vertex *vertex);
            int         setNextEdgeId() { return mNextEdgeId++; }
            int         setNextFaceId() { return mNextFaceId++; }
            int         setNextVertexId() { return mNextVertexId++; }
            bool        testConsistency();

         private:
            Face *mFaces;
            Vertex *mVertices;
            int mId, mNextFaceId, mNextEdgeId, mNextVertexId;
            int mEdgeCount, mFaceCount, mVertexCount, mHoleCount, mLoopCount;
         };
      }
   }
```

A seguir, é mostrado o código-fonte dos métodos da classe *Solid* que não foram implementados dentro da declaração da classe (com exceção dos operadores de Euler, que serão mostrado no capítulo seguinte):

```
namespace Modeling
{
   namespace HE
   {
      Solid::Solid(PrimitiveType primitiveType) :
         mId(0), mNextFaceId(0), mNextEdgeId(0), mNextVertexId(0),
               mEdgeCount(0), mFaceCount(0), mVertexCount(0), mLoopCount(0),
   mHoleCount(0),
         mFaces(0), mVertices(0)
      {
#ifdef _DEBUG
         std::cout << "Criando Solido..." << std::endl;
#endif
         if(primitiveType == PT_CUBE)
         {
            Point3D p0(-1.0, -1.0, -1.0);
            Point3D p1( 1.0, -1.0, -1.0);
            Point3D p2( 1.0, -1.0,  1.0);
            Point3D p3(-1.0, -1.0,  1.0);
            Point3D p4(-1.0,  1.0, -1.0);
            Point3D p5( 1.0,  1.0, -1.0);
            Point3D p6( 1.0,  1.0,  1.0);
            Point3D p7(-1.0,  1.0,  1.0);

            Loop *loop = mvfls(p0);

            HalfEdge *he0 = loop->mev(mVertices, p3);
            HalfEdge *he1 = loop->mev(he0, p1);
            HalfEdge *he2 = loop->mev(he1, p2);
            HalfEdge *he3 = mefl(he2, he0->getOppositeHalfEdge());

            HalfEdge *he4 = loop->mev(he0, p4);
            HalfEdge *he5 = loop->mev(he1, p5);
```

```
            HalfEdge *he6 = mefl(he5, he4);
            HalfEdge *he7 = loop->mev(he2, p6);
            std::cout << he7->getVertex()->getId() << "   " <<
               he5->getVertex()->getId() << std::endl;
            HalfEdge *he8 = mefl(he7, he6);
            HalfEdge *he9 = loop->mev(he3->getOppositeHalfEdge(), p7);
            HalfEdge *he10 = mefl(he9, he8);
            mefl(he4, he10);
         }
      }

      Solid::~Solid()
      {
#ifdef _DEBUG
         std::cout << "Destruindo Solido..." << std::endl;
#endif

         Face *prevFace = mFaces;
         while(mFaces)
         {
            mFaces = mFaces->next;
            delete prevFace;
            prevFace = mFaces;
         }

         Vertex *prevVertex = mVertices;
         while(mVertices)
         {
            mVertices = mVertices->next;
            delete prevVertex;
            prevVertex = mVertices;
         }

#ifdef _DEBUG
         std::cout << "Solido destruido." << std::endl;
#endif
      }

      void Solid::deleteFace(Face *face)
      {
         if(mFaces == face) mFaces = face->next;
         if(face->prev) face->prev->next = face->next;
         if(face->next) face->next->prev = face->prev;
         delete face;
      }

      Face* Solid::getFaceById(int id)
      {
         Face *currentFace = mFaces;
         do
         {
            if(currentFace->getId() == id)
               return currentFace;

            currentFace = currentFace->next;
         }
         while(currentFace != mFaces);

         return 0;
      }

      void Solid::insertIntoFaceList(Face *face)
      {
         if(mFaces)
         {
            mFaces->prev = face;
            face->next = mFaces;
```

```
            mFaces = face;
         }
         else
            mFaces = face;
      }

      void Solid::insertIntoVertexList(Vertex *vertex)
      {
         if(mVertices)
         {
            mVertices->prev = vertex;
            vertex->next = mVertices;
            mVertices = vertex;
         }
         else
            mVertices = vertex;
      }

      bool Solid::testConsistency()
      {
#ifdef _DEBUG
         std::cout << "V: " << mVertexCount << "   E: " << mEdgeCount <<
            "   F: " << mFaceCount << "   L: " << mLoopCount << "   H: " <<
            mHoleCount << std::endl;
#endif

         // Fórmula: V - E + 2F - L = 2 - 2H
         int V = mVertexCount, E = mEdgeCount, F = mFaceCount, L = mLoopCount,
            H = mHoleCount;
         if((V - E + 2F - L)) == (2 - (2 * H)))
            return true;
         else
            return false;
      }
   }
}
```

O mais interessante do código-fonte apresentado é o construtor da classe *Solid*, que cria um cubo usando os operadores de Euler, e o método *testConsistency*, que usa a fórmula de Euler-Poincaré para garantir a consistência da estrutura Half-Edge – ambos serão explicados com mais detalhes no Capítulo 6.

4.2. Classe *Face*

A classe *Face* possui uma declaração bem mais simples do que a classe *Solid*:

```
namespace Modeling
{
   namespace HE
   {
      class Face
      {
      public:
         Face(Solid *parentSolid, Vertex *vertex);
         ~Face();

         Loop*    keml(HalfEdge *halfEdge);
```

```
         Loop*    createLoop(Vertex *vertex);
         Loop*    getHeadLoop() { return mLoops; }
         int      getId() { return mId; }
         Solid*   getParentSolid() { return mParentSolid; }
         void     setHeadLoop(Loop *loop) { mLoops = loop; }
         void     setId(int id) { mId = id; }

      private:
         friend class Solid;

         Face *prev, *next;
         int mId;
         Loop *mLoops;
         Solid *mParentSolid;
      };
   }
}
```

O construtor da classe *Face* possui dois parâmetros: um ponteiro para o sólido ao qual pertence e um ponteiro para um vértice – por esse segundo parâmetro, é possível inferir que uma face nunca será criada vazia. Além disso, é de responsabilidade da face criar seus loops internos, fato demonstrado pelo método *createLoop*:

```
namespace Modeling
{
   namespace HE
   {
      Face::Face(Solid *parentSolid, Vertex *vertex) :
         mParentSolid(parentSolid), mId(parentSolid->setNextFaceId()),
         mLoops(0), prev(0), next(0)
      {
#ifdef _DEBUG
         std::cout << "Criando  Face " << mId << "..." << std::endl;
#endif

         createLoop(vertex);
      }

      Face::~Face()
      {
#ifdef _DEBUG
         std::cout << "Destruindo  Face " << mId << "..." << std::endl;
#endif
         Loop *prevLoop = mLoops;
         while(mLoops)
         {
            mLoops = mLoops->next;
            delete prevLoop;
            prevLoop = mLoops;
         }
      }

      Loop* Face::createLoop(Vertex *vertex)
      {
         Loop *newLoop = new Loop(this, vertex);

         if(mLoops == 0)
            mLoops = newLoop;
         else
```

```
            {
               Loop *loops = mLoops;

               // A lista de loops é linear, por isso precisamos
               // descobrir o final dela
               for(; loops->next; loops = loops->next);

               // Uma vez descoberto o final da lista,
               // basta acertar as referências
               loops->next = newLoop;
               newLoop->prev = loops;
            }

            return newLoop;
         }
      }
   }
```

O método *keml* da classe *Face* será mostrado no próximo capítulo.

4.3. Classe *Loop*

A classe *Loop* representa tanto o loop externo das faces quanto os loops internos, denominados *rings* em algumas literaturas:

```
namespace Modeling
{
   namespace HE
   {
      class Loop
      {
      public:
         Loop::Loop(Face *parentFace, Vertex *vertex);
         ~Loop();

         HalfEdge*   createEdge(Vertex *existingVertex, Point3D &newPoint);
         Face*       getParentFace() { return mParentFace; }
         int         getVerticesCount();
         HalfEdge*   getHeadHalfEdge() { return mHalfEdges; }
         bool        hasEdges() { return (mHalfEdges != 0 ? 1 : 0); }
         void        insertEdge(HalfEdge *halfEdge);
         HalfEdge*   mev(HalfEdge *halfEdge1, HalfEdge *halfEdge2,
                         Point3D &newPoint);
         HalfEdge*   mev(HalfEdge *halfEdge, Point3D &newPoint);
         HalfEdge*   mev(Vertex *vertex, Point3D &newPoint);
         void        setHeadHalfEdge(HalfEdge *halfEdge)
                        { mHalfEdges = halfEdge; }
         void     setParentFace(Face *face) { mParentFace = face; }

      private:
         friend class Solid;
         friend class Face;

         Loop *prev, *next;
         HalfEdge *mHalfEdges;
         Face *mParentFace;
      };
   }
}
```

O construtor da classe *Loop* recebe os mesmos parâmetros declarados pelo construtor da classe *Face*. Além disso, a classe possui o método *mev*, mostrando que é responsabilidade do loop executar essa operação (assim como os outros operadores de Euler, o método *mev* será explicado no próximo capítulo):

```
namespace Modeling
{
   namespace HE
   {
      Loop::Loop(Face *parentFace, Vertex *vertex) : mParentFace(parentFace),
         mHalfEdges(0), prev(0), next(0)
      {
#ifdef _DEBUG
         std::cout << "Criando   Loop..." << std::endl;
#endif
      }

      Loop::~Loop()
      {
#ifdef _DEBUG
         std::cout << "Destruindo   Loop..." << std::endl;
#endif

         HalfEdge *currentHalfEdge = mHalfEdges->next;
         HalfEdge *nextHalfEdge = currentHalfEdge->next;
         while(currentHalfEdge != mHalfEdges)
         {
            delete currentHalfEdge;
            currentHalfEdge = nextHalfEdge;
            nextHalfEdge = currentHalfEdge->next;
         }
         delete mHalfEdges;
      }

      int Loop::getVerticesCount()
      {
         HalfEdge *current = mHalfEdges;
         int count = 0;
         do
         {
            count++;
            current = current->next;
         }
         while(current != mHalfEdges);

         return count;
      }

      void Loop::insertEdge(HalfEdge *halfEdge)
      {
         // Insere no final da lista
         mHalfEdges->prev->next = halfEdge;
         halfEdge->prev = mHalfEdges->prev;
         mHalfEdges->prev = halfEdge;
         halfEdge->next = mHalfEdges;
      }
   }
}
```

4.4. Classe *HalfEdge*

A classe *HalfEdge* é o coração da estrutura – nela fica a maior parte das referências para outros elementos que compõem a topologia:

```
namespace Modeling
{
   namespace HE
   {
      class HalfEdge
      {
      public:

         HalfEdge(Loop *parentLoop, Vertex *connectedVertex, int id);
         HalfEdge(Vertex *connectedVertex, HalfEdge *nextHalfEdge, int id);
         ~HalfEdge();

         int         getId() { return mId; }
         HalfEdge*   getOppositeHalfEdge() { return mOppositeHalfEdge; }
         HalfEdge*   getNext() { return next; }
         Loop*       getParentLoop() { return mParentLoop; }
         HalfEdge*   getPrev() { return prev; }
         Vertex*     getVertex() { return mVertex; }
         void        setId(int id) { mId = id; }
         void        setOppositeHalfEdge(HalfEdge *oppositeHalfEdge)
                        { mOppositeHalfEdge = oppositeHalfEdge; }
         void        setParentLoop(Loop *parentLoop);
         void        setVertex(Vertex *vertex) { mVertex = vertex; }

      private:
         friend class Solid;
         friend class Face;
         friend class Loop;

         HalfEdge *prev, *next;
         int mId;
         Vertex *mVertex;
         HalfEdge *mOppositeHalfEdge;
         Loop *mParentLoop;
      };
   }
}
```

O construtor da classe *HalfEdge* possui três parâmetros: um ponteiro para o loop do qual a meia-aresta fará parte, um ponteiro para o vértice conectado à meia-aresta e um número sequencial, utilizado para identificar a meia-aresta em pesquisas lineares:

```
namespace Modeling
{
   namespace HE
   {
      HalfEdge::HalfEdge(Loop *parentLoop, Vertex *connectedVertex, int id) :
         mParentLoop(parentLoop), mVertex(connectedVertex),
         OppositeHalfEdge(0), mId(id), prev(this), next(this)
      {
#ifdef _DEBUG
         std::cout << "Criando    Meia-Aresta " << mId << ", Vertex ID: " <<
            mVertex->getId() << "..." << std::endl;
```

```
#endif

         if(!mVertex->getConnectedHalfEdges())
            mVertex->setHeadHalfEdge(this);
      }

      HalfEdge::HalfEdge(Vertex *connectedVertex, HalfEdge *nextHalfEdge,
         int id) : mParentLoop(nextHalfEdge->getParentLoop()),
         mOppositeHalfEdge(0), mVertex(connectedVertex), mId(id),
         prev(nextHalfEdge->prev), next(nextHalfEdge)
      {
#ifdef _DEBUG
         std::cout << "Criando    Meia-Aresta " << mId << ", Vertex ID: " <<
            mVertex->getId() << "..." << std::endl;
#endif

         // As referências desta meia-aresta foram definidas no preâmbulo;
         // portanto, agora só falta redefinir os ponteiros de nextHalfEdge e
         // de seu predecessor
         nextHalfEdge->prev->next = this;
         next->prev = this;
      }

      HalfEdge::~HalfEdge()
      {
#ifdef _DEBUG
         std::cout << "Destruindo    Meia-Aresta " << mId << "..." <<
            std::endl;
#endif

         mVertex = 0;
         mOppositeHalfEdge = 0;
         mParentLoop = 0;
         prev = next = 0;
      }

      void HalfEdge::setParentLoop(Loop *parentLoop)
      {
         mParentLoop = parentLoop;
         if(!parentLoop->getHeadHalfEdge())
            parentLoop->setHeadHalfEdge(this);
      }
   }
}
```

4.5. Estrutura *Point3D*

A estrutura *Point3D* define um ponto no espaço 3D:

```
namespace Modeling
{
   namespace HE
   {
      struct Point3D
      {
         Point3D() : x(0.0f), y(0.0f), z(0.0f) { }
         Point3D(float tX, float tY, float tZ) : x(tX), y(tY), z(tZ) { }

         float x;
         float y;
         float z;
      };
   }
}
```

4.6. Classe *Vertex*

A classe *Vertex* representa um vértice da topologia:

```
namespace Modeling
{
   namespace HE
   {
      class Vertex
      {
      public:
         Vertex(const Point3D &point);
         Vertex(Loop *parentLoop);
         Vertex(Loop *parentLoop, const Point3D &point);

         ~Vertex();

         int         getId() { return mId; }
         HalfEdge*   getConnectedHalfEdges() { return mConnectedHalfEdges; }
         Point3D&    getPosition() { return mPoint; }
         void        setHeadHalfEdge(HalfEdge *halfEdge)
                        { mConnectedHalfEdges = halfEdge; }
         void        setId(int id) { mId = id; }
         void        setPosition(const Point3D &point);

      private:
         friend class Solid;

         Vertex *prev, *next;
         HalfEdge *mConnectedHalfEdges;
         int mId;
         Point3D mPoint;
      };
   }
}
```

O construtor da classe *Vertex* recebe como parâmetro uma referência ao ponto que será usado como suas coordenadas no espaço 3D:

```
namespace Modeling
{
   namespace HE
   {
      Vertex::Vertex(const Point3D &point) : mId(-1), mConnectedHalfEdges(0),
```

```
            next(0), prev(0)
        {
            mPoint.x = point.x;
            mPoint.y = point.y;
            mPoint.z = point.z;
        }

        Vertex::Vertex(Loop *parentLoop, const Point3D &point) :
            mId(parentLoop->getParentFace()->getParentSolid()->setNextVertexId()),
            mConnectedHalfEdges(0), next(0), prev(0)
        {
            mPoint.x = point.x;
            mPoint.y = point.y;
            mPoint.z = point.z;
        }

        Vertex::~Vertex()
        {
#ifdef _DEBUG
            std::cout << "Destruindo     Vertice " << mId << "..." << std::endl;
#endif
        }

        void Vertex::setPosition(const Point3D &point)
        {
            mPoint.x = point.x;
            mPoint.y = point.y;
            mPoint.z = point.z;
        }
    }
}
```

Todas as classes e estruturas declaradas até aqui representam a definição clássica da estrutura de dados Half-Edge. Perceba que o código é minimalista – apenas o estritamente necessário foi declarado.

No capítulo seguinte, vamos dar uma olhada nos operadores de Euler que serão implementados na estrutura Half-Edge, para, no capítulo final, criarmos um exemplo prático de sua utilização.

Capítulo 5

Operadores de Euler

A estrutura de dados Half-Edge, por si só, apenas armazena e organiza os elementos que compõem um objeto poligonal 3D – para manipulá-los, devemos modificar as conexões dos elementos internos (faces, loops, meia-arestas e vértices).

No entanto, devemos tomar cuidado com o tipo de manipulação realizada – afinal, o sólido deve obedecer à fórmula de Euler-Poincaré para ser considerado um sólido válido.

Por esse motivo, deve-se limitar o conjunto de operações permitidas, de forma a garantir a validade do sólido – tais operadores são chamados de operadores de Euler.

5.1. Operadores construtivos e destrutivos – nomenclatura

Costuma-se dividi-los em dois grupos de operações opostas, com um número variável de operadores.

Os operadores recebem nomes abreviados, usando os seguintes critérios:

- Os operadores criam e/ou destroem elementos, recebendo a letra M (*make*) quando criam elementos e K (*kill*) quando destroem elementos.
- À frente das letras M e K, são colocadas as iniciais dos elementos criados/destruídos: F para face, E (*Edge*) para arestas (duplas de meia-arestas), V

para vértices, L (*Loop*) ou R (*Ring*) para loops, G (*Genus*) ou H (*Hole*) para buracos e S (Shell) para sólidos como um todo.

Portanto, um operador pode ser chamado de K*xyz*M*xyz* ou de M*xyz*K*xyz*. Os operadores mais simples apenas criam ou destroem, como é o caso de MEFL (cria uma aresta, uma face e um loop), MEV (cria uma aresta e um vértice), KEFL (destrói uma aresta, uma face e um loop) e KEV (destrói uma aresta e um vértice). Operadores mais complexos criam alguns elementos e destroem outros, como é o caso de MEKL (cria uma aresta, destrói um loop) e de KFMH (destrói uma face, cria um buraco).

5.2. Selecionando operadores de Euler

De acordo com Martti Mantyla, uma implementação de Half-Edge costuma ter dez operadores de Euler – cinco construtivos e cinco destrutivos; no entanto, para manter o escopo desse trabalho mais restrito, vamos nos concentrar apenas nos cinco operadores construtivos:

- MVFLS (cria um vértice, uma face, um loop e um sólido);
- KFMH (destrói uma face, cria um buraco);
- MEFL (cria uma aresta, uma face e um loop);
- MEV (cria uma aresta e um vértice); e
- KEML (destrói uma aresta, cria um loop).

Xianming Chen adiciona mais um operador, o KFFMH, que, segundo ele, facilita a criação de buracos – no entanto, o operador KFMH também possui essa função, então decidi não incluir o operador KFFMH na nossa discussão, por mais que o trabalho de Chen seja uma das maiores influências do código-fonte apresentado.

Perceba, também, que Chen trabalha com o caso especial de que o sólido é

composto de apenas um *shell* (o S da operação MVFLS), que é suficiente para o desenvolvimento desse trabalho – inclusive, é por esse motivo que não fazemos distinção entre sólidos e *shells*.

Cada operador é realizado por um elemento diferente:

- MVFLS, KFMH e MEFL são realizados pelo sólido.
- MEV é realizado por um loop.
- KEML é realizado por uma face.

Vamos, então, estudar o funcionamento e o código-fonte de cada um deles.

5.3. MVFLS

O operador MVFLS tem como objetivo criar um *shell* – ou sólido, no nosso caso, pois os nossos sólidos sempre conterão apenas um *shell*.

Curiosamente, em todas as literaturas esse operador aparece como MVFS, omitindo a criação do loop. No entanto, isso não é possível, pois um vértice precisa existir dentro do contexto de um loop; além disso, a não-criação do loop violaria a fórmula de Euler-Poincaré:

- Vértices (V): 0 +1
- Arestas (E): 0
- Faces (F): 0 + 1
- Loops (L): 0
- Shells (S): 0 + 1
- Buracos (H): 0

$$V - E + 2F - L = 2 - 2H$$

$$1 - 0 + 2 - 0 = 2 - 0$$

$$3 \neq 2$$

O correto seria:

- Vértices (V): 0 +1
- Arestas (E): 0
- Faces (F): 0 + 1
- Loops (L): 0 + 1
- Shells (S): 0 + 1
- Buracos (H): 0

$$V - E + 2F - L = 2 - 2H$$

$$1 - 0 + 2 - 1 = 2 - 0$$

$$2 = 2$$

Alguns autores deixam implícita a criação do loop, o que eu considero contraproducente para quem está estudando a estrutura Half-Edge; por isso, optei por adicionar o L ao nome da operação, chamando-a de MVFLS.

A figura a seguir mostra visualmente o algoritmo da operação MVFLS:

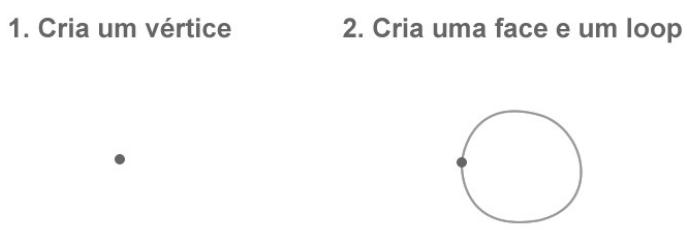

Figura 5.1 – Operação MVFLS.

O código-fonte relativo ao processo da figura é mostrado a seguir:

```
namespace Modeling
{
   namespace HE
   {
   Loop* Solid::mvfls(const Point3D &point)
      {
         // Cria o vértice e atribui um id a ele
         Vertex *newVertex = new Vertex(point);
         newVertex->setId(setNextVertexId());
#ifdef _DEBUG
         std::cout << "Criando     Vertice " << newVertex->getId() << " (" <<
            point.x << ", " << point.y << ", " << point.z << ")..." <<
            std::endl;
#endif

         // Insere o vértice no início da lista de vértices do sólido
         insertIntoVertexList(newVertex);

         // Cria a face
         // NOTA: o loop é criado no momento em que a face é criada
         Face *newFace = new Face(this, newVertex);

         // Insere a face na lista de faces do sólido
         insertIntoFaceList(newFace);

         mVertexCount++;
         mFaceCount++;
         mLoopCount++;

         // Verifica a consistência do sólido
         assert(testConsistency());

         // Retorna o loop que contém o vértice recém-criado
         return newFace->getHeadLoop();
      }
   }
}
```

5.4. MEV

O operador MEV cria um vértice e uma aresta (ou seja, um par de meias-arestas) em um determinado loop – confira a figura a seguir para entender o processo:

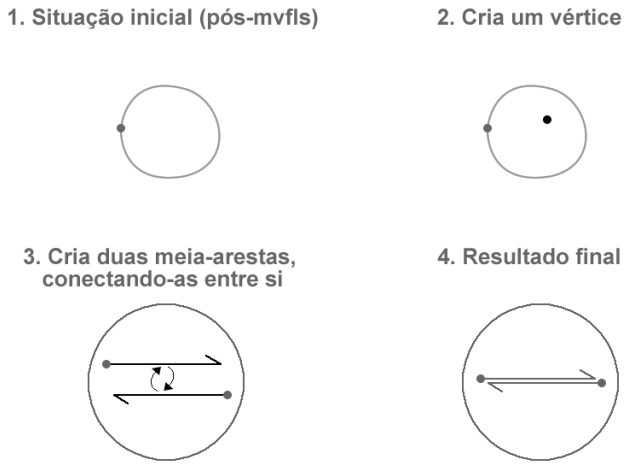

Figura 5.2 – Operação MEV após uma operação MVFLS.

Vamos adicionar os elementos criados pelo operador na fórmula de Euler-Poincaré para verificar se o sólido permanece válido – a título de exemplo, vamos considerar o caso da Figura, em que uma operação MVFLS foi realizada previamente:

Vértices (V): 1 +1

Arestas (E): 0 + 1

Faces (F): 1

Loops (L): 1

Shells (S): 1

Buracos (H): 0

$$V - E + 2F - L = 2 - 2H$$

$$2 - 1 + 2 - 1 = 2 - 0$$

$$2 = 2$$

No entanto, o caso apresentado na Figura é o caso trivial. Na figura a seguir, é mostrado um caso mais complexo, em que a operação MEV é aplicada para criar uma subdivisão em loops já formados (ou seja, que componham faces completas):

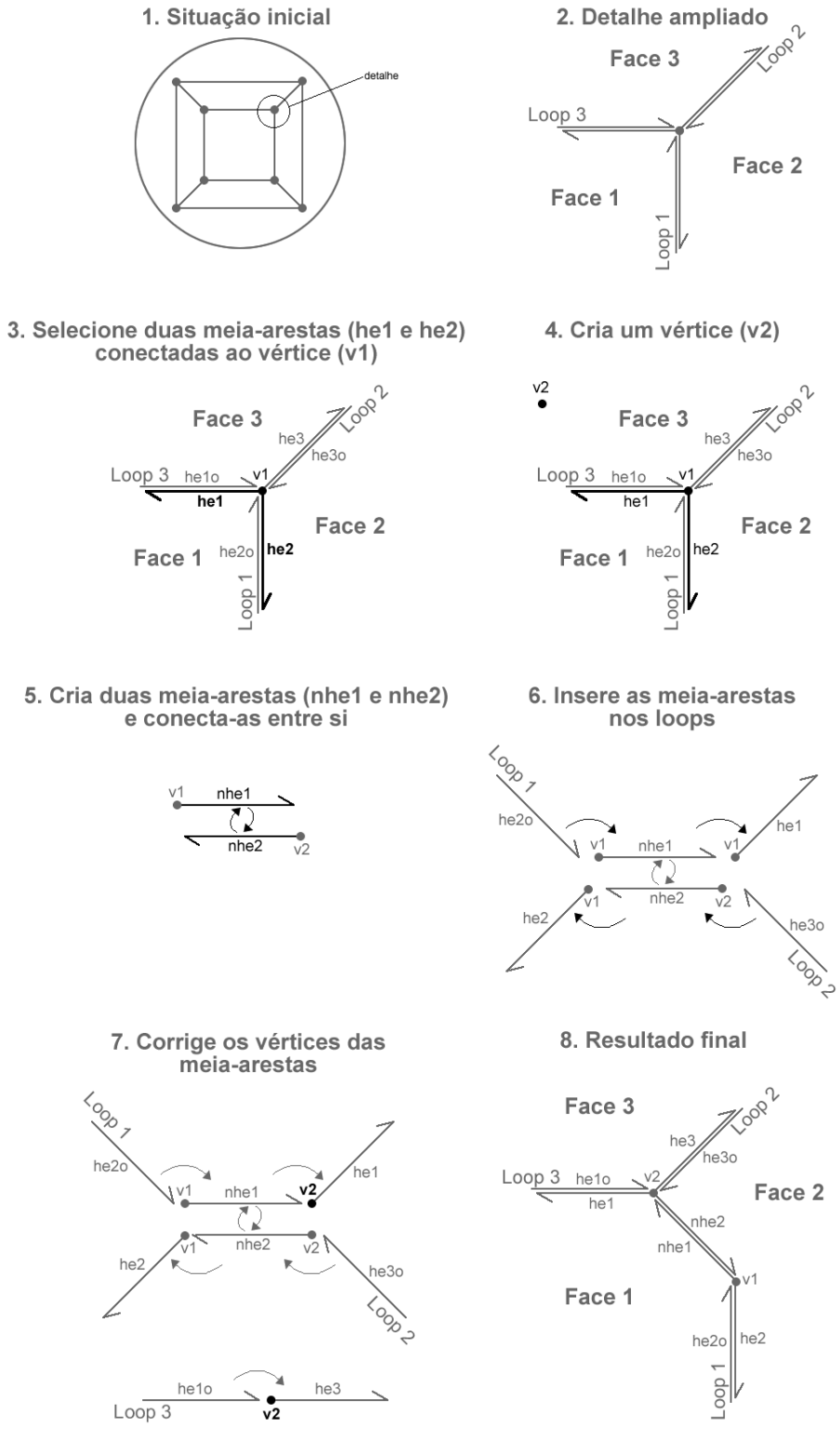

Figura 5.3 – Operação MEV em loops já formados.

O código-fonte do método MEV, para o primeiro e o segundo casos, é mostrado a seguir:

```
namespace Modeling
{
   namespace HE
   {
      HalfEdge* Loop::mev(HalfEdge *halfEdge1, HalfEdge *halfEdge2,
         Point3D &newPoint)
      {
         // Cria o vértice
         Vertex *newVertex = new Vertex(newPoint);
         newVertex->setId(mParentFace->getParentSolid()->setNextVertexId());
         mParentFace->getParentSolid()->insertIntoVertexList(newVertex);

#ifdef _DEBUG
         std::cout << "Criando     Vertice " << newVertex->getId() << " (" <<
            newPoint.x << ", " << newPoint.y << ", " << newPoint.z << ")..." <<
            std::endl;
#endif

         // Cria o par de meia-arestas e conecta-os
         HalfEdge *he2_next = halfEdge2->next;
         int edgeId = mParentFace->getParentSolid()->setNextEdgeId();
         HalfEdge *newHalfEdge1 = new HalfEdge(halfEdge1->getVertex(),
            halfEdge1, edgeId);
         HalfEdge *newHalfEdge2 = new HalfEdge(newVertex, he2_next->prev,
            edgeId);

         newHalfEdge1->setOppositeHalfEdge(newHalfEdge2);
         newHalfEdge2->setOppositeHalfEdge(newHalfEdge1);

         // Subdivide o ciclo de meia-arestas, atribuindo o novo vértice para
         // halfEdge1 e para o oposto da meia-aresta anterior a newHalfEdge2.
         // Se newHalfEdge1 e newHalfEdge2 fizerem parte do mesmo loop,
         // nada acontece.
         for(HalfEdge *halfEdge = newHalfEdge1->next;
            halfEdge != newHalfEdge2;
            halfEdge = halfEdge->getOppositeHalfEdge()->next)
         {
            halfEdge->setVertex(newHalfEdge2->getVertex());
         }

         mParentFace->getParentSolid()->increaseEdgeCount();
         mParentFace->getParentSolid()->increaseVertexCount();

         // Verifica a consistência do sólido
         mParentFace->getParentSolid()->testConsistency();

         // Retorna newHalfEdge2
         return newHalfEdge2;
      }

      // Usado apenas para criar a primeira aresta de um sólido
      HalfEdge* Loop::mev(Vertex *vertex, Point3D &newPoint)
      {
         // Cria o vértice
         Vertex *newVertex = new Vertex(newPoint);
         newVertex->setId(getParentFace()->getParentSolid()->
            setNextVertexId());
         mParentFace->getParentSolid()->insertIntoVertexList(newVertex);

#ifdef _DEBUG
         std::cout << "Criando     Vertice " << newVertex->getId() << " (" <<
         newPoint.x << ", " << newPoint.y << ", " << newPoint.z << ")..." <<
         std::endl;
#endif

         int edgeId = mParentFace->getParentSolid()->setNextEdgeId();
```

```
            // Cria o par de meia-arestas e conecta-os
            HalfEdge *newHalfEdge1 = new HalfEdge(this, vertex, edgeId);
            HalfEdge *newHalfEdge1_opposite = new HalfEdge(newVertex,
               newHalfEdge1, edgeId);

            newHalfEdge1->setOppositeHalfEdge(newHalfEdge1_opposite);
            newHalfEdge1_opposite->setOppositeHalfEdge(newHalfEdge1);

            mParentFace->getParentSolid()->increaseEdgeCount();
            mParentFace->getParentSolid()->increaseVertexCount();

            // Verifica a consistência do sólido
            assert(mParentFace->getParentSolid()->testConsistency());

            // Retorna newHalfEdge1
            return newHalfEdge1;
         }
      }
   }
```

Um terceiro caso ocorre quando queremos adicionar uma aresta dentro de apenas um loop, como é mostrado na figura a seguir:

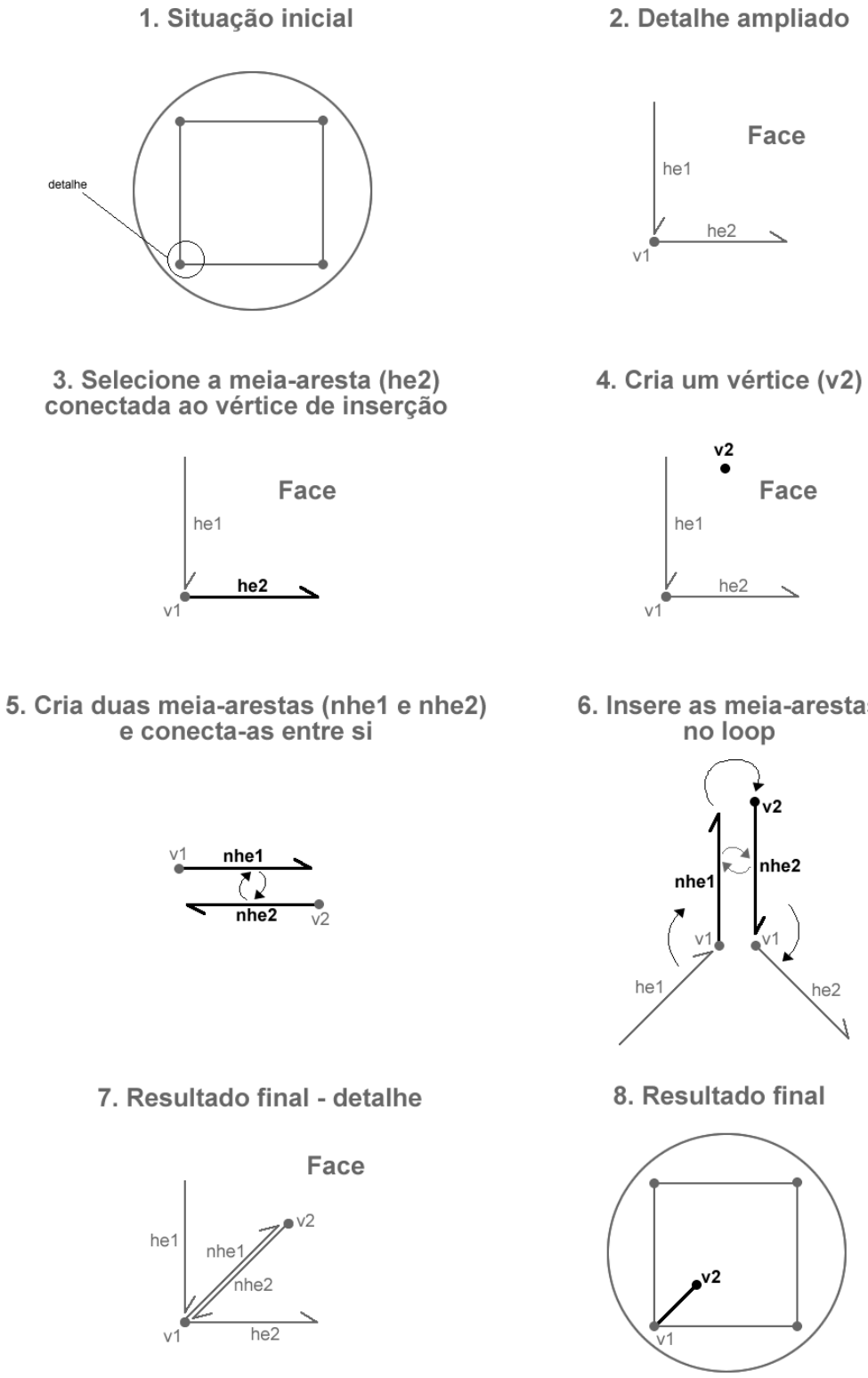

Figura 5.4 – Operação MEV aplicada em apenas um loop.

Esse terceiro caso é um caso especial do método já mostrado – basta passar a mesma meia-aresta nos dois primeiros parâmetros. Portanto, podemos criar um método específico para ele:

```
namespace Modeling
{
   namespace HE
   {
      // Usado quando se quer subdividir internamente um loop; a subdivisão
      // ocorrerá antes de halfEdge.
      HalfEdge* Loop::mev(HalfEdge *halfEdge, Point3D &newPoint)
      {
         return (mev(halfEdge, halfEdge, newPoint));
      }
   }
}
```

5.5. MEFL

O operador MEFL cria uma aresta, uma face e um loop. Geralmente, ele é utilizado após a aplicação repetida de operadores MEV em um loop, com o objetivo de dividi-lo e criar uma nova face; em seguida, algumas meia-arestas são transferidas para o loop recém-criado da nova face. Confira a figura 5.5 para entender melhor o que está acontecendo:

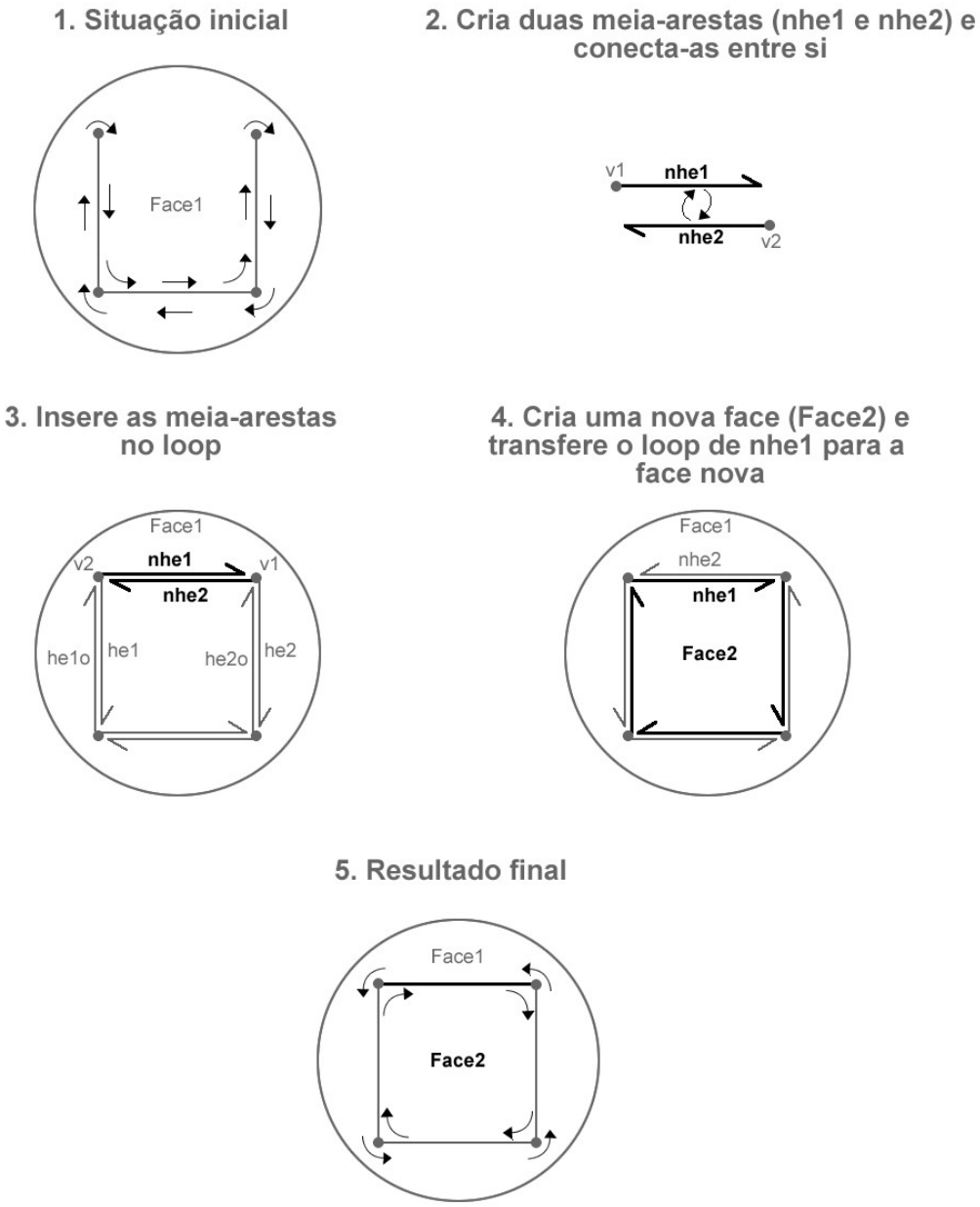

Figura 5.5 – Desenvolvimento da operação MEFL.

Para verificar a validade do operador, considere o caso mostrado na figura a seguir, que é um resumo da figura 5.5:

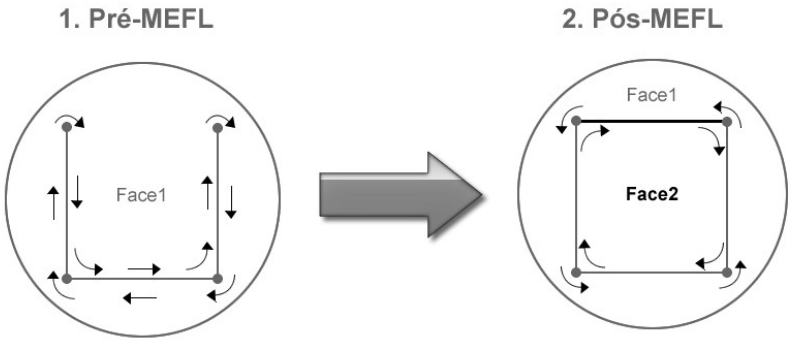

Figura 5.6 – Resumo da operação MEFL.

O sólido possui apenas uma face com um loop; o loop, por sua vez, possuía quatro vértices e três arestas antes da operação – vejamos, então, como a aplicação do operador MEFL irá influir na fórmula de Euler-Poincaré:

- Vértices (V): 4
- Arestas (E): 3 + 1
- Faces (F): 1 + 1
- Loops (L): 1 + 1
- Shells (S): 1
- Buracos (H): 0

$$V - E + 2F - L = 2 - 2H$$

$$4 - 4 + 4 - 2 = 2 - 0$$

$$2 = 2$$

Mais uma vez, a fórmula manteve-se equilibrada.

Vejamos como ficou o código-fonte dessa operação:

```
namespace Modeling
{
   namespace HE
   {
      HalfEdge* Solid::mefl(HalfEdge *halfEdge1, HalfEdge *halfEdge2)
      {
```

```
            int edgeId = setNextEdgeId();

            // Cria o par de meia-arestas e conecta-os
            HalfEdge *he1_prev = halfEdge1->prev;
            HalfEdge *he2_prev = halfEdge2->prev;

            HalfEdge *newHalfEdge1 = new HalfEdge(halfEdge1->getVertex(),
               halfEdge2, edgeId);
            newHalfEdge1->prev = he1_prev;

            HalfEdge *newHalfEdge2 = new HalfEdge(halfEdge2->getVertex(),
               halfEdge1, edgeId);
            newHalfEdge2->prev = he2_prev;

            he1_prev->next = newHalfEdge1;
            he2_prev->next = newHalfEdge2;

            newHalfEdge1->setOppositeHalfEdge(newHalfEdge2);
            newHalfEdge2->setOppositeHalfEdge(newHalfEdge1);

            Face *newFace = new Face(this, halfEdge1->getVertex());
            Loop *newLoop = newFace->getHeadLoop();

            // Insere a face na lista de faces do sólido
            insertIntoFaceList(newFace);

            // Atualiza as referências de face e loop das meia-arestas
            // que serão transportadas
            HalfEdge *halfEdge = newHalfEdge1;
            do
            {
               halfEdge->setParentLoop(newLoop);
               halfEdge = halfEdge->next;
            }
            while(halfEdge != newHalfEdge1);

            // Configura ambas as meia-arestas como sendo
            // as primeiras de seus loops
            newHalfEdge1->getParentLoop()->setHeadHalfEdge(newHalfEdge1);
            newHalfEdge2->getParentLoop()->setHeadHalfEdge(newHalfEdge2);

#ifdef _DEBUG
            std::cout << std::endl;
            std::cout << "--> mefl" << std::endl;
            std::cout << " Loop da face antiga (" << newHalfEdge2->
               getParentLoop()->getParentFace()->getId() << "): ";
            HalfEdge *he = newHalfEdge2->getParentLoop()->getHeadHalfEdge();
            do
            {
               std::cout << he->getId() << "(" << he->getVertex()->getId() <<
                  ")  ";
               he = he->next;
            }
            while(he != newHalfEdge2->getParentLoop()->getHeadHalfEdge());
            std::cout << std::endl;

            std::cout << " Loop da face nova (" << newHalfEdge1->getParentLoop()->
               getParentFace()->getId() << "): ";
            he = newHalfEdge1->getParentLoop()->getHeadHalfEdge();
            do
            {
               std::cout << he->getId() << "(" << he->getVertex()->getId() <<
                  ")  ";
               he = he->next;
            }
            while(he != newHalfEdge1->getParentLoop()->getHeadHalfEdge());
            std::cout << std::endl;
```

```
            std::cout << std::endl;
#endif

            mEdgeCount++;
            mFaceCount++;
            mLoopCount++;

            // Verifica a consistência do sólido
            assert(testConsistency());

            // Retorna newHalfEdge1, que corresponde ao sentido
            // que estava sendo seguido
            return newHalfEdge1;
        }
    }
}
```

5.6. KEML

A função do operador KEML é subdividir um loop *dentro de uma mesma face*, diferentemente de MEFL, que também subdivide um loop, mas cria uma nova face. A figura a seguir mostra o operador em ação:

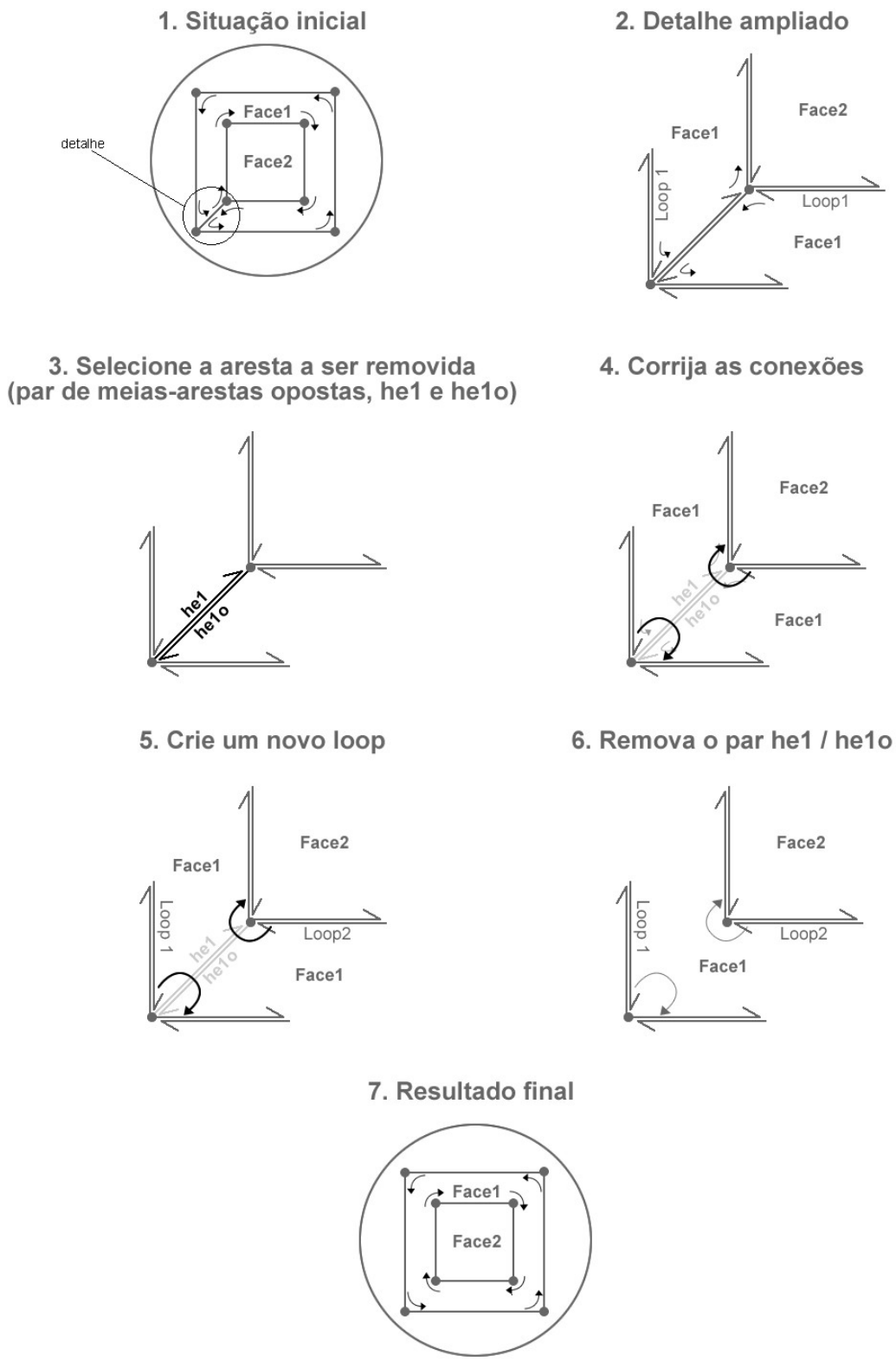

Figura 5.7 – Resumo da operação KEML.

Considerando a figura anterior, vamos ver como a fórmula de Euler-Poincaré se

comporta com a aplicação do operador:

- Vértices (V): 8
- Arestas (E): 9 – 1
- Faces (F): 3
- Loops (L): 3 + 1
- Shells (S): 1
- Buracos (H): 0

$$V - E + 2F - L = 2 - 2H$$

$$8 - 8 + 6 - 4 = 2 - 0$$

$$2 = 2$$

Veja como ficou o código-fonte para esse operador:

```
namespace Modeling
{
   namespace HE
   {
      Loop* Face::keml(HalfEdge *halfEdge)
      {
         // halfEdge1 e halfEdge2 são meias-arestas opostas
         HalfEdge *halfEdge1 = halfEdge;
         HalfEdge *halfEdge2 = halfEdge->getOppositeHalfEdge();

         assert(halfEdge1->getParentLoop() == halfEdge2->getParentLoop() &&
            halfEdge2->next != halfEdge1);

         // Loop antigo
         halfEdge1->prev->next = halfEdge2->next;
         halfEdge2->next->prev = halfEdge1->prev;
         if(halfEdge1->getParentLoop()->getHeadHalfEdge() == halfEdge1 ||
            halfEdge1->getParentLoop()->getHeadHalfEdge() == halfEdge2)
         {
            halfEdge1->getParentLoop()->setHeadHalfEdge(halfEdge1->prev);
         }

         // Loop interno novo
         Loop *loop = 0;
         if(halfEdge1->next == halfEdge2)
         {
            // Como halfEdge1 e halfEdge2 são opostos, o loop novo
            // conterá apenas um vértice após a destruição de ambos
            loop = createLoop(halfEdge2->getVertex());
         }
         else
         {
            loop = createLoop(0);

            // Correção das referências das meias-arestas que
```

```
            // estão conectadas às meias-arestas que serão removidas
            halfEdge2->prev->next = halfEdge1->next;
            halfEdge1->next->prev = halfEdge2->prev;
            loop->setHeadHalfEdge(halfEdge1->next);

            // Inserção das meias-arestas no loop novo
            HalfEdge *halfEdge0 = halfEdge1->next;
            HalfEdge *temp = halfEdge0;
            do
            {
               temp->setParentLoop(loop);
            }
            while((temp = temp->next) != halfEdge0);
         }

         // Exclusão das meias-arestas
         delete halfEdge1;
         delete halfEdge2;

         mParentSolid->decreaseEdgeCount();
         mParentSolid->increaseLoopCount();

         // Verifica a consistência do sólido
         assert(mParentSolid->testConsistency());

         // Retorna o loop recém-criado
         return loop;
      }
   }
}
```

5.7. KFMH

O operador KFMH tem como objetivo transferir os loops de uma face para outra, criando um "buraco" no sólido – observe a figura a seguir:

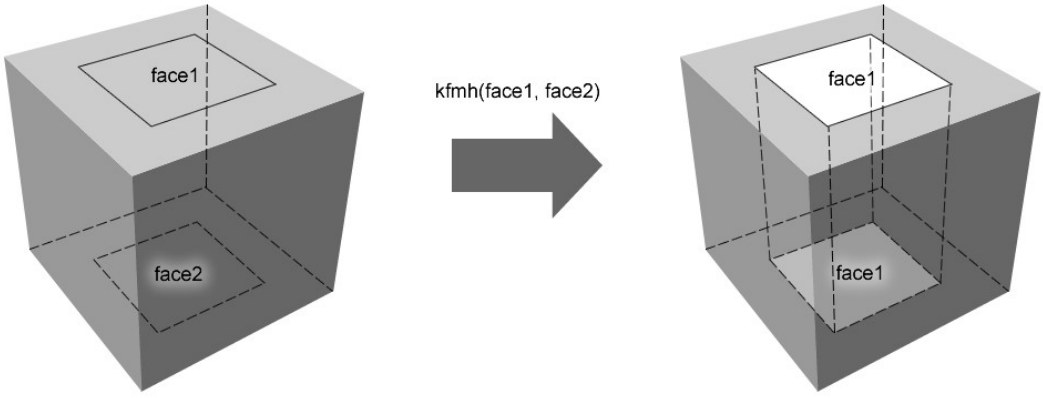

Figura 5.8 – Aplicação do operador KFMH.

No final do processo, a face que teve os loops transferidos é destruída.

Perceba que, após a aplicação do operador no sólido da figura anterior, ocorre

uma conexão das faces de cima e de baixo, já que o loop externo do buraco se encontra na face de baixo e o loop interno foi transferido para a face de cima:

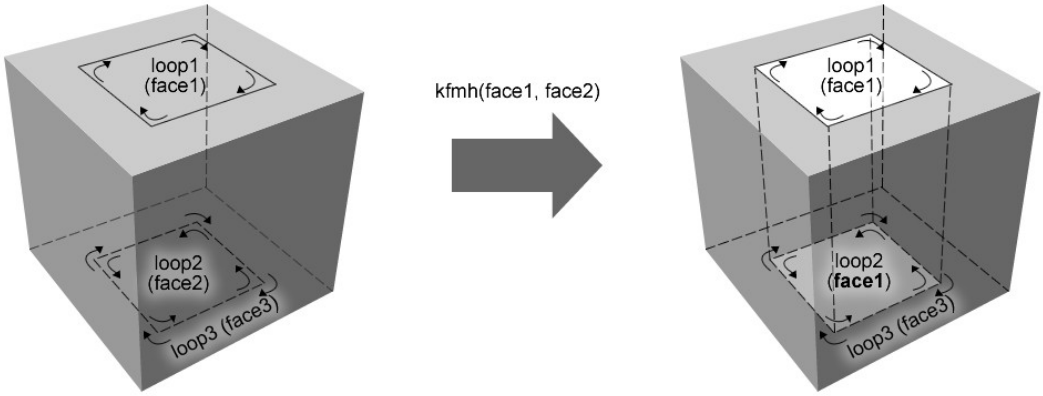

Figura 5.9 – Transferência do loop de face2 *para* face1.

De um ponto de vista prático – criar modelos 3D para jogos, no caso – essa operação isolada não faz sentido. No entanto, se fizermos três operações MEFL consecutivas após a KFMH, teremos um buraco interno, tal como mostra a figura:

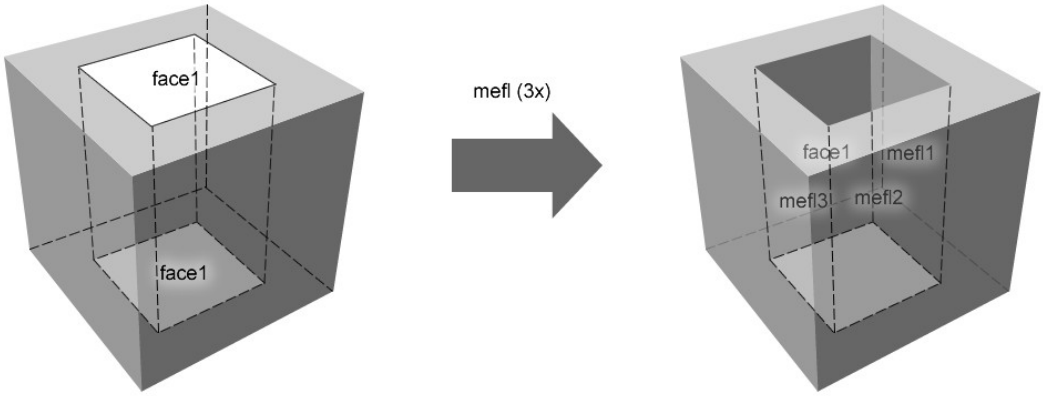

*Figura 5.10 – Três MEFL consecutivos, gerando três novas faces (*mefl1*,* mefl2 *e* mefl3*).*

Confira a fórmula de Euler-Poincaré antes e depois da operação KFMH:

Pré-KFMH

- Vértices (V): 16

- Arestas (E): 20
- Faces (F): 8
- Loops (L): 10
- Shells (S): 1
- Buracos (H): 0

$$V - E + 2F - L = 2 - 2H$$

$$16 - 20 + 16 - 10 = 2 - 0$$

$$2 = 2$$

Pós-KFMH

- Vértices (V): 16
- Arestas (E): 20
- Faces (F): 8 – 1
- Loops (L): 10
- Shells (S): 1
- Buracos (H): 0 + 1

$$V - E + 2F - L = 2 - 2H$$

$$16 - 20 + 14 - 10 = 2 - 2$$

$$0 = 0$$

O código-fonte do operador é mostrado a seguir:

```
namespace Modeling
{
   namespace HE
   {
      void Solid::kfmh(Face *face1, Face *face2)
      {
         assert(! face2->getHeadLoop()->next);
         assert(face1 != face2);

         // Transfere os loops de face2 para face1
         for(Loop *loop = face2->getHeadLoop(); loop; loop = loop->next)
         {
```

```
            loop->setParentFace(face1);
        }

        Loop *next = face1->getHeadLoop()->next;

        // Insere a lista de loops de face2 em face1
        face1->getHeadLoop()->next = face2->getHeadLoop();
        face2->getHeadLoop()->prev = face1->getHeadLoop();

        // Corrige o final da inserção de loops em face1
        Loop *last = face2->getHeadLoop();
        while(last->next) last = last->next;
        last->next = next;
        if(next) next->prev = last;

        // Anula a lista de loops de face2
        face2->setHeadLoop(0);

        // Caso a primeira face da lista de faces do sólido
        // seja face2, faz-se necessário corrigir a lista
        if(mFaces == face2) mFaces = face2->next;
        if(face2->prev) face2->prev->next = face2->next;
        if(face2->next) face2->next->prev = face2->prev;

        // destrói face2
        deleteFace(face2);

        mHoleCount++;

        // Verifica a consistência do sólido
        assert(testConsistency());
      }
    }
  }
```

Uma última observação: Xianming Chen chama essa operação de KFMRH, indicando que, além de um buraco, também é criado um anel (*Ring*), ou seja, um loop interno em uma face. No entanto, como não estamos lidando com *rings* na nossa fórmula de Euler-Poincaré, decidi omitir o R do nome da operação, com o objetivo de evitar futuras confusões. Assim, a operação foi chamada apenas de KFMH, e não KFMRH.

Capítulo 6

Construindo e modificando a topologia de um sólido com operadores de Euler

Agora que temos a implementação das classes da estrutura Half-Edge, bem como dos operadores de Euler, chegou a hora de ver como tudo funciona na prática.

6.1. Criando um cubo com operadores de Euler

Vamos começar revisando o código-fonte do construtor da classe *Solid*, que tem como função principal criar o objeto primitivo (o cubo, no caso):

```
Solid::Solid(PrimitiveType primitiveType) :
   mId(0), mNextFaceId(0), mNextEdgeId(0), mNextVertexId(0),
   mEdgeCount(0), mFaceCount(0), mVertexCount(0),
   mFaces(0), mVertices(0)
{
#ifdef _DEBUG
   std::cout << "Criando Solido..." << std::endl;
#endif
   if(primitiveType == PT_CUBE)
   {
      Point3D p0(-1.0, -1.0, -1.0);
      Point3D p1( 1.0, -1.0, -1.0);
      Point3D p2( 1.0, -1.0,  1.0);
      Point3D p3(-1.0, -1.0,  1.0);
      Point3D p4(-1.0,  1.0, -1.0);
      Point3D p5( 1.0,  1.0, -1.0);
      Point3D p6( 1.0,  1.0,  1.0);
      Point3D p7(-1.0,  1.0,  1.0);

      Loop *loop = mvfls(p0);

      HalfEdge *he0 = loop->mev(mVertices, p1);
      HalfEdge *he2 = loop->mev(he0, p2);
      HalfEdge *he3 = loop->mev(he2, p3);
      HalfEdge *he1 = mefl(he0->getOppositeHalfEdge(), he3);

      HalfEdge *he4 = loop->mev(he0, p4);
      HalfEdge *he5 = loop->mev(he1, p5);
```

```
        HalfEdge *he6 = mefl(he4, he5);
        HalfEdge *he7 = loop->mev(he3, p6);
        HalfEdge *he8 = mefl(he5, he7);
        HalfEdge *he9 = loop->mev(he2, p7);
        mefl(he7, he9);
        // Cuidado, o correto é he6, e não he4!!!
        // Isso acontece porque precisamos de uma aresta conectada
        // ao vértice 4 E QUE ESTEJA DENTRO DO LOOP EM QUESTÃO!
        mefl(he6, he9);
    }
}
```

A figura a seguir, baseada no trabalho de Sven Havemann, mostra a criação passo-a-passo do cubo:

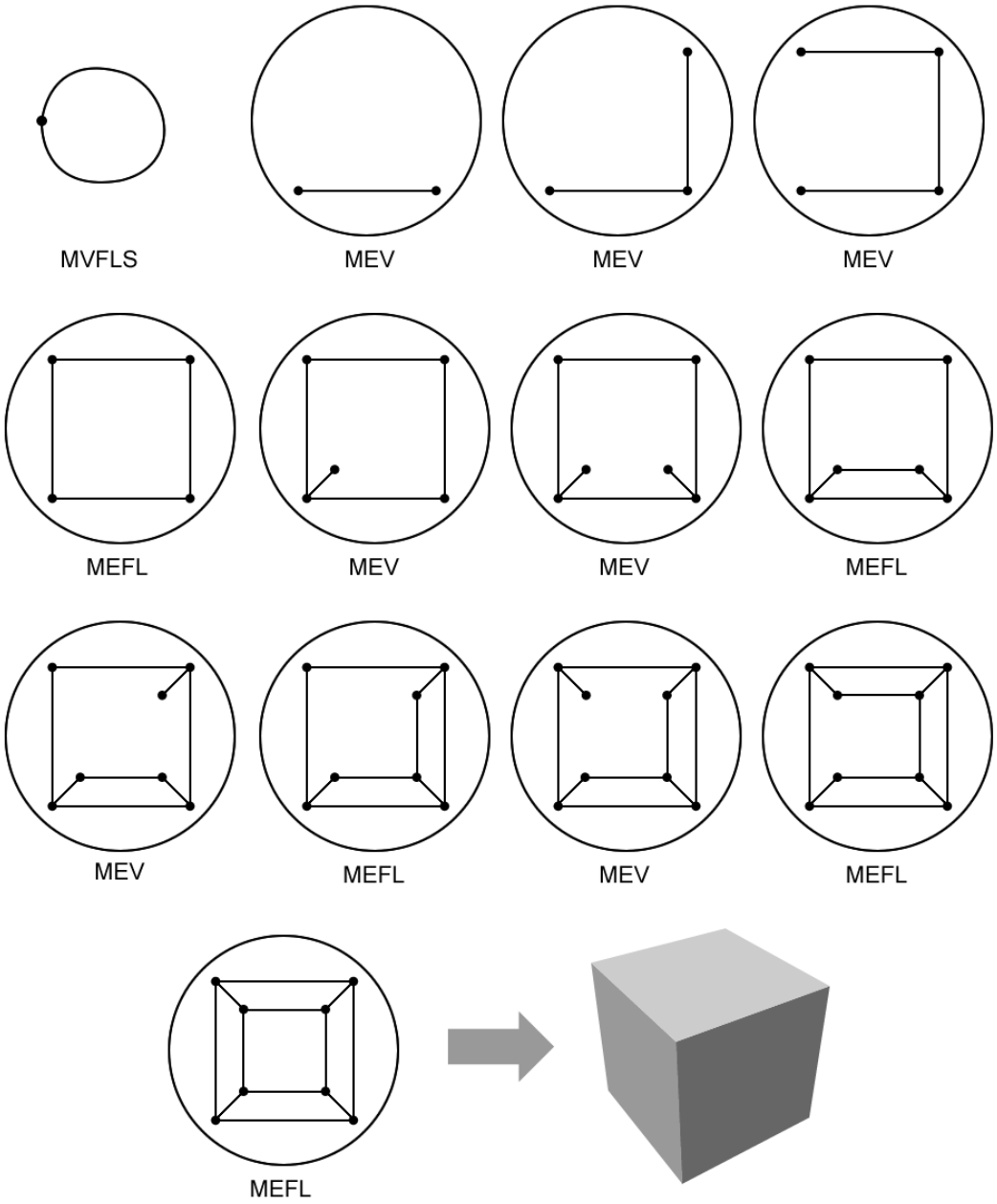

Figura 6.1 – Criação passo-a-passo do cubo.

6.2. Criando uma extrusão no cubo

Agora, vamos realizar, em um das faces, uma operação denominada *extrusão*. A extrusão implica em conectar a face selecionada a uma nova face, criando uma "ponte" de faces entre a face antiga e a recém-criada – observe a figura para entender o processo:

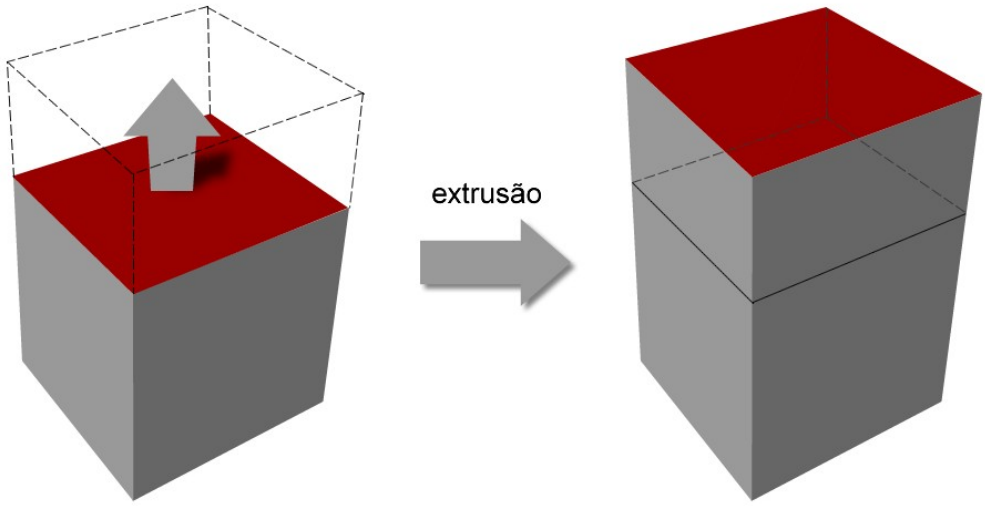

Figura 6.2 – Aplicação de extrusão em uma das faces.

Vamos, então, criar uma aplicação-teste, na qual criaremos o sólido e aplicaremos as modificações necessárias à realização da extrusão:

```
int main()
{
   Modeling::HE::Solid *solid = new Modeling::HE::Solid();

   Modeling::HE::Face *face = solid->getFaceById(5);
   Modeling::HE::HalfEdge *he = face->getHeadLoop()->getHeadHalfEdge();
   std::cout << he->getVertex()->getId() << std::endl;

   Modeling::HE::Vertex *v = he->getVertex();
   Modeling::HE::Point3D p(v->getPosition().x,
                           v->getPosition().y + 2.0f,
                           v->getPosition().z);
   Modeling::HE::HalfEdge *nhe1 = face->getHeadLoop()->mev(he, p);

   he = he->getNext();
   v = he->getVertex();
   p.x = v->getPosition().x;
   p.y = v->getPosition().y + 2.0f;
   p.z = v->getPosition().z;
   Modeling::HE::HalfEdge *nhe2 = face->getHeadLoop()->mev(he, p);
   Modeling::HE::HalfEdge *nhe3 = solid->mefl(nhe1, nhe2);
```

```
   he = he->getNext();
   v = he->getVertex();
   p.x = v->getPosition().x;
   p.y = v->getPosition().y + 2.0f;
   p.z = v->getPosition().z;
   Modeling::HE::HalfEdge *nhe4 = face->getHeadLoop()->mev(he, p);
   Modeling::HE::HalfEdge *nhe5 = solid->mefl(nhe2, nhe4);

   he = he->getNext();
   v = he->getVertex();
   p.x = v->getPosition().x;
   p.y = v->getPosition().y + 2.0f;
   p.z = v->getPosition().z;
   Modeling::HE::HalfEdge *nhe6 = face->getHeadLoop()->mev(he, p);
   Modeling::HE::HalfEdge *nhe7 = solid->mefl(nhe4, nhe6);

   Modeling::HE::HalfEdge *nhe8 = solid->mefl(nhe6, nhe3);

   delete solid;
   return 0;
}
```

Você deve ter reparado que a criação da extrusão possui uma natureza iterativa – isso é um sinal de que toda a operação poderia ser encapsulada em um método, generalizando o processo.

6.3. Testando a consistência do sólido

Se você se recorda do código-fonte dos operadores de Euler, deverá se lembrar da seguinte instrução, que aparece ao final de cada um deles:

```
// Verifica a consistência do sólido
assert(testConsistency());
```

Ao final de cada operador de Euler, o método *testConsistency* da classe *Solid* é invocado, com o objetivo de testar a consistência do sólido modificado – o código-fonte do método é mostrado a seguir:

```
namespace Modeling
{
   namespace HE
   {
      bool Solid::testConsistency()
      {
#ifdef _DEBUG
         std::cout << "V: " << mVertexCount << "   E: " << mEdgeCount <<
            "   F: " << mFaceCount << "   L: " << mLoopCount << "   H: " <<
            mHoleCount << std::endl;
#endif

         // Fórmula: V - E + 2F - L = 2 - 2H
         int V = mVertexCount, E = mEdgeCount, F = mFaceCount, L = mLoopCount,
```

```
            H = mHoleCount;
         if((V - E + 2F - L)) == (2 - (2 * H)))
            return true;
         else
            return false;
      }
   }
}
```

A função *assert* é executada apenas em projetos nos quais a macro *_DEBUG* está definida – portanto, ela só é testada em projetos que estejam em modo de depuração, diminuindo a carga de processamento da estrutura Half-Edge em ambientes de produção propriamente ditos.

O método *testConsistency* utiliza a fórmula de Euler-Poincaré – que foi exaustivamente demonstrada no capítulo anterior – para verificar a consistência do sólido após as transformações decorrentes da aplicação de um operador de Euler. Se a fórmula

$$V - E + 2F - L = 2 - 2H$$

resultar em uma igualdade, significa que o sólido é topologicamente consistente.

Ao executar esta aplicação em modo de depuração, o seguinte texto é gerado:

```
Criando Solido...
Criando     Vertice 0 (-1, -1, -1)...
Criando  Face 0...
Criando   Loop...
V: 1   E: 0   F: 1   L: 1   H: 0
Criando     Vertice 1 (-1, -1, 1)...
Criando    Meia-Aresta 0, Vertex ID: 0...
Criando    Meia-Aresta 0, Vertex ID: 1...
V: 2   E: 1   F: 1   L: 1   H: 0
Criando     Vertice 2 (1, -1, -1)...
Criando    Meia-Aresta 1, Vertex ID: 0...
Criando    Meia-Aresta 1, Vertex ID: 2...
V: 3   E: 2   F: 1   L: 1   H: 0
Criando     Vertice 3 (1, -1, 1)...
Criando    Meia-Aresta 2, Vertex ID: 2...
Criando    Meia-Aresta 2, Vertex ID: 3...
V: 4   E: 3   F: 1   L: 1   H: 0
Criando    Meia-Aresta 3, Vertex ID: 3...
Criando    Meia-Aresta 3, Vertex ID: 1...
Criando  Face 1...
Criando   Loop...

--> mefl
 Loop da face antiga (0): 3(1)  2(3)  1(2)  0(0)
 Loop da face nova (1): 3(3)  0(1)  1(0)  2(2)

V: 4   E: 4   F: 2   L: 2   H: 0
```

```
Criando     Vertice 4 (-1, 1, -1)...
Criando    Meia-Aresta 4, Vertex ID: 0...
Criando    Meia-Aresta 4, Vertex ID: 4...
V: 5   E: 5   F: 2   L: 2   H: 0
Criando     Vertice 5 (1, 1, -1)...
Criando    Meia-Aresta 5, Vertex ID: 2...
Criando    Meia-Aresta 5, Vertex ID: 5...
V: 6   E: 6   F: 2   L: 2   H: 0
Criando    Meia-Aresta 6, Vertex ID: 5...
Criando    Meia-Aresta 6, Vertex ID: 4...
Criando  Face 2...
Criando   Loop...

--> mefl
 Loop da face antiga (0): 6(4)  5(5)  1(2)  4(0)
 Loop da face nova (2): 6(5)  4(4)  0(0)  3(1)  2(3)  5(2)

V: 6   E: 7   F: 3   L: 3   H: 0
Criando     Vertice 6 (1, 1, 1)...
Criando    Meia-Aresta 7, Vertex ID: 3...
Criando    Meia-Aresta 7, Vertex ID: 6...
V: 7   E: 8   F: 3   L: 3   H: 0
6   5
Criando    Meia-Aresta 8, Vertex ID: 6...
Criando    Meia-Aresta 8, Vertex ID: 5...
Criando  Face 3...
Criando   Loop...

--> mefl
 Loop da face antiga (2): 8(5)  7(6)  2(3)  5(2)
 Loop da face nova (3): 8(6)  6(5)  4(4)  0(0)  3(1)  7(3)

V: 7   E: 9   F: 4   L: 4   H: 0
Criando     Vertice 7 (-1, 1, 1)...
Criando    Meia-Aresta 9, Vertex ID: 1...
Criando    Meia-Aresta 9, Vertex ID: 7...
V: 8   E: 10   F: 4   L: 4   H: 0
Criando    Meia-Aresta 10, Vertex ID: 7...
Criando    Meia-Aresta 10, Vertex ID: 6...
Criando  Face 4...
Criando   Loop...

--> mefl
 Loop da face antiga (3): 10(6)  9(7)  3(1)  7(3)
 Loop da face nova (4): 10(7)  8(6)  6(5)  4(4)  0(0)  9(1)

V: 8   E: 11   F: 5   L: 5   H: 0
Criando    Meia-Aresta 11, Vertex ID: 4...
Criando    Meia-Aresta 11, Vertex ID: 7...
Criando  Face 5...
Criando   Loop...

--> mefl
 Loop da face antiga (4): 11(7)  4(4)  0(0)  9(1)
 Loop da face nova (5): 11(4)  10(7)  8(6)  6(5)

V: 8   E: 12   F: 6   L: 6   H: 0
4
Criando     Vertice 8 (-1, 3, -1)...
Criando    Meia-Aresta 12, Vertex ID: 4...
Criando    Meia-Aresta 12, Vertex ID: 8...
V: 9   E: 13   F: 6   L: 6   H: 0
Criando     Vertice 9 (-1, 3, 1)...
Criando    Meia-Aresta 13, Vertex ID: 7...
Criando    Meia-Aresta 13, Vertex ID: 9...
V: 10   E: 14   F: 6   L: 6   H: 0
Criando    Meia-Aresta 14, Vertex ID: 8...
```

```
Criando    Meia-Aresta 14, Vertex ID: 9...
Criando  Face 6...
Criando   Loop...

--> mefl
 Loop da face antiga (5): 14(9)  12(8)  11(4)  13(7)
 Loop da face nova (6): 14(8)  13(9)  10(7)  8(6)  6(5)  12(4)

V: 10   E: 15   F: 7   L: 7   H: 0
Criando     Vertice 10 (1, 3, 1)...
Criando    Meia-Aresta 15, Vertex ID: 6...
Criando    Meia-Aresta 15, Vertex ID: 10...
V: 11   E: 16   F: 7   L: 7   H: 0
Criando    Meia-Aresta 16, Vertex ID: 9...
Criando    Meia-Aresta 16, Vertex ID: 10...
Criando  Face 7...
Criando   Loop...

--> mefl
 Loop da face antiga (6): 16(10)  13(9)  10(7)  15(6)
 Loop da face nova (7): 16(9)  15(10)  8(6)  6(5)  12(4)  14(8)

V: 11   E: 17   F: 8   L: 8   H: 0
Criando     Vertice 11 (1, 3, -1)...
Criando    Meia-Aresta 17, Vertex ID: 5...
Criando    Meia-Aresta 17, Vertex ID: 11...
V: 12   E: 18   F: 8   L: 8   H: 0
Criando    Meia-Aresta 18, Vertex ID: 10...
Criando    Meia-Aresta 18, Vertex ID: 11...
Criando  Face 8...
Criando   Loop...

--> mefl
 Loop da face antiga (7): 18(11)  15(10)  8(6)  17(5)
 Loop da face nova (8): 18(10)  17(11)  6(5)  12(4)  14(8)  16(9)

V: 12   E: 19   F: 9   L: 9   H: 0
Criando    Meia-Aresta 19, Vertex ID: 11...
Criando    Meia-Aresta 19, Vertex ID: 8...
Criando  Face 9...
Criando   Loop...

--> mefl
 Loop da face antiga (8): 19(8)  17(11)  6(5)  12(4)
 Loop da face nova (9): 19(11)  14(8)  16(9)  18(10)

V: 12   E: 20   F: 10   L: 10   H: 0
Destruindo Solido...
Destruindo  Face 9...
Destruindo   Loop...
Destruindo    Meia-Aresta 14...
Destruindo    Meia-Aresta 16...
Destruindo    Meia-Aresta 18...
Destruindo    Meia-Aresta 19...
Destruindo  Face 8...
Destruindo   Loop...
Destruindo    Meia-Aresta 17...
Destruindo    Meia-Aresta 6...
Destruindo    Meia-Aresta 12...
Destruindo    Meia-Aresta 19...
Destruindo  Face 7...
Destruindo   Loop...
Destruindo    Meia-Aresta 15...
Destruindo    Meia-Aresta 8...
Destruindo    Meia-Aresta 17...
Destruindo    Meia-Aresta 18...
Destruindo  Face 6...
```

```
Destruindo   Loop...
Destruindo    Meia-Aresta 13...
Destruindo    Meia-Aresta 10...
Destruindo    Meia-Aresta 15...
Destruindo    Meia-Aresta 16...
Destruindo  Face 5...
Destruindo   Loop...
Destruindo    Meia-Aresta 12...
Destruindo    Meia-Aresta 11...
Destruindo    Meia-Aresta 13...
Destruindo    Meia-Aresta 14...
Destruindo  Face 4...
Destruindo   Loop...
Destruindo    Meia-Aresta 4...
Destruindo    Meia-Aresta 0...
Destruindo    Meia-Aresta 9...
Destruindo    Meia-Aresta 11...
Destruindo  Face 3...
Destruindo   Loop...
Destruindo    Meia-Aresta 9...
Destruindo    Meia-Aresta 3...
Destruindo    Meia-Aresta 7...
Destruindo    Meia-Aresta 10...
Destruindo  Face 2...
Destruindo   Loop...
Destruindo    Meia-Aresta 7...
Destruindo    Meia-Aresta 2...
Destruindo    Meia-Aresta 5...
Destruindo    Meia-Aresta 8...
Destruindo  Face 1...
Destruindo   Loop...
Destruindo    Meia-Aresta 0...
Destruindo    Meia-Aresta 1...
Destruindo    Meia-Aresta 2...
Destruindo    Meia-Aresta 3...
Destruindo  Face 0...
Destruindo   Loop...
Destruindo    Meia-Aresta 5...
Destruindo    Meia-Aresta 1...
Destruindo    Meia-Aresta 4...
Destruindo    Meia-Aresta 6...
Destruindo     Vertice 11...
Destruindo     Vertice 10...
Destruindo     Vertice 9...
Destruindo     Vertice 8...
Destruindo     Vertice 7...
Destruindo     Vertice 6...
Destruindo     Vertice 5...
Destruindo     Vertice 4...
Destruindo     Vertice 3...
Destruindo     Vertice 2...
Destruindo     Vertice 1...
Destruindo     Vertice 0...
Solido destruido.
```

Outras operações de alto nível, semelhantes à extrusão, podem ser realizadas com os operadores de Euler – aconselho que o leitor gaste algum tempo estudando modeladores 3D que possuam modelagem sólida (Wings 3D, 3D Studio Max, Maya, Modo etc.) para entender o processo de modelagem na prática e, também, para tentar

imaginar como as várias operações poderiam ser criadas, utilizando os operadores de Euler como base técnica.

Conclusão

A manipulação de superfícies 3D por parte dos artistas de jogos é fundamental para criar objetos, personagens e cenários para jogos. No entanto, as estruturas utilizadas para armazenar objetos 3D em hardware – em especial a estrutura Index Buffer / Vertex Buffer – não são adequadas para o processo de modelagem, pois elas foram criadas para serem leves, e não flexíveis.

Por isso, é preciso criar uma camada de abstração sobre as estruturas de armazenamento em hardware para que os artistas tenham um mínimo de flexibilidade para criar os objetos 3D – olhando sob esse prisma, a estrutura Half-Edge se encaixa perfeitamente nesse conceito de camada de abstração, pois possui muito mais informações de conectividade entre os elementos que compõem o objeto 3D; além disso, por meio dos operadores de Euler, é possível realizar modificações inimagináveis no nível de abstração do hardware.

Mesmo assim, apenas a estrutura Half-Edge e os operadores de Euler não são suficientes. Os operadores, vistos de forma isolada, nada significam; é preciso agrupá-los em sequência, para que, dessas sequências, emerjam modificações significativas do ponto de vista do artista 3D – a extrusão, demonstrada no capítulo 6, representa uma dessas sequências significativas.

Identificar e codificar um conjunto de macro-operações significativas, embasadas nos operadores de Euler, deve ser o objetivo inicial de qualquer pessoa que pretenda desenvolver algo baseado na estrutura Half-Edge.

O objetivo deste trabalho foi reunir e ordenar o conhecimento necessário para se criar uma estrutura Half-Edge que implemente os cinco operadores de Euler básicos, utilizados em processos construtivos.

Com esse objetivo realizado, além da demonstração da operação de extrusão, fica como exercício posterior para o leitor pesquisar e desenvolver outras macro-operações significativas do ponto de vista do artista 3D.

Bibliografia

BAUMGART, Bruce G. **Winged-Edge Polyhedron Representation for Computer Vision**. 1975. Disponível em: <http://www.baumgart.org/winged-edge/winged-edge.html>. Acesso em: 06 jun. 2008.

CHEN, Xianming. **euler**. 2006. Disponível em: <http://www.cs.utah.edu/~xchen/euler-doc/index.html>. Acesso em: 10 jul. 2008.

CROSSLEY, Martin D. **Essential Topology**. 1. ed. Londres: Springer, 2005. Caps. 2, 3, 4, 5 e 7.

HANRAHAN, Patrick M. **Creating Volume Models from Edge-Vertex Graphs**. ACM SIGGRAPH Computer Graphics, v. 16, n. 3, p. 77-84, 1982.

HAVEMANN, Sven. **Generative Mesh Modeling**. 2005. 303f. Dissertação (Doutorado em Engenharia). Institut für ComputerGraphik, TU Brawnschweig, Alemanha, 2005.

HOFFMANN, Christoph M. **Geometric and Solid Modeling – An Introduction**. 1. ed. São Francisco: Morgan Kaufamnn, 1989. Caps. 1 e 2.

KETTNER, Lutz. **Designing a Data Structure for Polyhedral Surfaces**. Proceedings of the 14th ACM Symposium on Computational Geometry, Minneapolis, Minnesota, p. 146-154, 1998.

MANTYLA, Martti; SULONEN, Reijio. **GWB: A Solid Modeler with Euler Operators**. IEEE Computer Graphics & Applications, v.2, n.7, p. 17-31, 1982.

SIERADSKY, Allan J. **An Introduction to Topology and Homotopy**. 1. ed. Boston: PWS-KENT Publishing, 1992. Cap. 13.