MURILO ANTONIO SALOMÃO GARCIA

ALGORITMO PARA CONVERTER SÓLIDOS CSG EM SÓLIDOS B-REP

Dissertação Apresentada à Escola Politécnica da Universidade de São Paulo para Obtenção do Título de Mestre em Engenharia.

SÃO PAULO

2006

MURILO ANTONIO SALOMÃO GARCIA

ALGORITMO PARA CONVERTER SÓLIDOS CSG EM SÓLIDOS B-REP

Dissertação Apresentada à Escola Politécnica da Universidade de São Paulo para Obtenção do Título de Mestre em Engenharia.

Área de Concentração: Engenharia Mecatrônica

Orientador:
Prof. Dr.
Marcos de Sales Gerra Tsuzuki

São Paulo

2006

Dedico este trabalho à minha mãe que esteve sempre ao meu lado nas horas difíceis e alegres, sempre me ajudando e apoiando minhas decisões.
Mãe,
obrigado por tudo.

Agradecimentos

Agradeço ao meu orientador, Prof. Dr. Marcos de Sales Guerra Tsuzuki, pela ajuda recebida durante a condução deste trabalho.

À minha namorada Larissa, pelo apoio, compreensão e paciência.

Ao meu pai Walter, pela preocupação e pelo apoio.

Aos meus irmãos Valeria e Bernardo, pelo apoio e pelo carinho.

Aos meus amigos do grupo do Bachir, pelo apoio e por não me deixar desistir.

Ao meu amigo Thiago, pelas incontáveis discussões que muito acrescentaram a este trabalho.

Ao meu primo Pedro, pela grande ajuda.

Ao meu amigo Takase, pela ajuda durante este trabalho.

À minha prima Vera, por me tranqüilizar e pelas dicas.

Ao meu amigo Mathias, por me ajudar com o LaTeX.

Ao meu amigo Emiliano, pela ajuda.

Finalmente, a todos os familiares e amigos que direta ou indiretamente me ajudaram na conclusão deste trabalho de mestrado.

A todos vocês,
Obrigado.

Resumo

O objetivo deste trabalho é definir um novo algoritmo para a conversão de um modelo CSG para um modelo B-Rep. Normalmente isto é feito, percorrendo-se a árvore CSG e interpretando-a, ou seja: aplicando-se rotações, translações e escalamento (transformações afins) e determinando-se a união, intersecção ou diferença (operações booleanas) entre dois modelos B-Rep. Nós definimos uma nova abordagem onde o espaço ocupado pelo sólido é determinado como uma representação volumétrica. Após isso, utilizando-se esta representação volumétrica um Modelo Sólido B-Rep é gerado. Um algoritmo assim poderá ser utilizado no futuro para a criação de Modelos Sólidos B-Rep a partir de Imagens Médicas Tridimensionais. Gerando desta forma um modelo sólido completo ao invés de uma Representação por Superfícies, que é como tradicionalmente se tem feito. O Modelo Sólido B-Rep tem a vantagem de permitir a fácil extração de propriedades relativas a massa, como: momento de inércia, centro de massa e volume. Além disso a uma representação por superfícies não garante a geração de um volume fechado (o que é suficiente para a visualização), porém para o uso na engenharia como análise e manufatura por esteriolitografia, é necessário um volume fechado. Um Modelo Sólido B-Rep é uma representação completa para o uso na engenharia. Por fim, posto que a construção de um Modelo Sólido é diferente da construção tradicional de uma Representação por Superfícies, nós tivemos que modificar o algoritmo Marching Cubes para obter o resultado desejado.

Abstract

The purpose of this work is to define a new algorithm for converting a CSG representation into a B-Rep representation. Usually this conversion is done by walking through the CSG tree translating it, which means: applying rotations, translations and scaling(affine transformations) and determining the union, intersection or difference from two B-Rep represented solids. We will define another approach where the space used by the solids is determined as a volumetric representation. Then, using the volumetric representation a B-Rep Solid Model is created. Such an algorithm can be used in the future for creating B-Rep Solid Models from Three Dimensional Medical Images. Generating a full Solid Model instead of the traditional approach of only Surface Representations for 3D medical images, has the advantage that mass properties are easily extracted from a Solid Model, such as: volume, moment of inertia and mass centre. The surface representation does not guarantee that a closed volume is created. It is enough for visualization. However, for engineering purposes as analysis or stereolitography manufacturing, a closed volume is necessary. A B-Rep Solid Model is a complete representation for engineering purposes. The construction of a Solid Model is different from the traditional construction of a surface model; thus we modified the marching cubes algorithm to reach this objective.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Capítulo 1

INTRODUÇÃO

A representação de Modelos sólidos por meio de Constructive Solid Geometry (CSG) permite ao usuário criar objetos sólidos 3D extremamente complexos de forma hierárquica, combinando primitivas geométricas simples por meio de operações Booleanas e transformações afins [2]. Este é um método de modelagem de sólidos muito popular e extremamente poderoso, e é particularmente indicado para o design e manipulação interativa de sólidos.

Tradicionalmente, as primitivas CSG são definidas por objetos analíticos simples como blocos, cilindros e esferas. Alguns algoritmos CSG mais recentes dão suporte também a primitivas, modelos sólidos quaisquer definidos pela superfície de suas fronteiras. Usando representações volumétricas baseadas em voxels, uma extensão maior a esses algoritmos pode incluir objetos extraídos de dados volumétricos usando-se limiares de intensidade. Estes dados volumétricos podem ser obtidos pela digitalização de objetos reais através de métodos como Tomografia Computadorizada, Ressonância Magnética, Imagens de Microscópios ou por meio de amostragem de funções implícitas ou procedurais. Estes modelos CSG expandidos são por vezes chamados de modelos CSG Volumétricos e são muito úteis em aplicações cirúrgicas, exames médicos baseados em imagens e modelagem de fenômenos amorfos [3].

Devido à falta de representação explícita das superfícies das fronteiras dos modelos sólidos, a exibição de CSG em tela não é suportada de forma nativa por dispositivos gráficos padrões. Ainda que muitos algoritmos de renderização CSG interativos tenham sido desenvolvidos no passado [4], eles não podem ser diretamente aplicados quando estão envolvidos dados volumétricos.

Uma possível solução para a renderização de um modelo CSG volumétrico é converter o modelo CSG em um modelo volumétrico baseado em voxel, construir-se o modelo sólido B-Rep e por fim renderizar-se este modelo B-Rep. Este método (conversão de um modelo CSG em um modelo volumétrico) é conhecido como Voxelização CSG. A Voxelização CSG conceitualmente é um problema de se classificar um elemento como pertencente ou não ao objeto CSG para todos pontos de amostragem no volume espacial. A Voxelização CSG é tomada como base para o desenvolvimento deste trabalho, e em seguida gera-se o modelo B-Rep.

Veremos adiante que a Voxelização CSG não será explícita: o modelo volumétrico existirá apenas de forma temporária e estará implicitamente representado na estrutura de dados utilizada pelo algoritmo que criamos. O modelo volumétrico será apenas intermediário, ou seja, não teremos uma estrutura de dados especificamente criada para armazenar-lo (por exemplo uma *octree*). Ainda que uma estrutura deste tipo facilitasse a visualização do modelo volumétrico, este não é o escopo deste trabalho.

O algoritmo que criamos baseia-se no Marching Cubes, mas apresenta, no entanto, três diferenças fundamentais em seu emprego. Tradicionalmente os dados de entrada do algoritmo Marching Cubes são Imagens Médicas Tridimensionais, e em no nosso caso serão modelos CSG. Além disso, enquanto no algoritmo tradicional todo o espaço tem que ser percorrido, no algoritmo proposto apenas a fronteira do sólido é percorrida. Por fim os dados de saída do Marching Cubes são Representações por Superfícies, e no nosso caso serão modelos B-Rep foram ainda efetuadas outras alterações com o intuito de tornar o algoritmo mais eficiente.

Capítulo 2

Representação CSG

Representações baseadas na construção do sólido capturam o processo de construção que define o sólido por uma seqüência de operações que instanciam ou combinam primitivas ou combinam o resultado de construções anteriores. Essas representações usualmente capturam a tentativa de criação de um design do usuário em um alto nível de representação que pode ser facilmente editado e parametrizado. Constructive Solid Geometry (CSG) é a representação por construção mais difundida na modelagem de sólidos. Suas primitivas são sólidos parametrizados, que podem possuir formas simples (como blocos, cones, cilindros e esferas) conforme ilustrado na Figura 2.1 ou formas mais complexas apropriadas para um conjunto de aplicações particulares como, no caso da engenharia mecânica, rasgos de chaveta e furos roscados (Figura 2.2). Resalta-se que as primitivas podem ser instanciadas várias vezes e agrupadas hierarquicamente.

2.1 Transformações Afins

Instâncias de primitivas e grupos podem ser manipulados por meio de transformações afins visando mudar a forma original das primitivas, possibilitando assim que o usuário possa chegar à forma final desejada. Estas transformações afins (conforme Figura 2.3) incluem, entre outras, as seguintes transformações espaciais: rotações, translações e escalamentos uniformes e não uniformes (esticando os eixos por um fator de escala constante) e reflexões (invertendo objetos em relação a um eixo). Estas transformações possuem características em comum, como por exemplo mapear

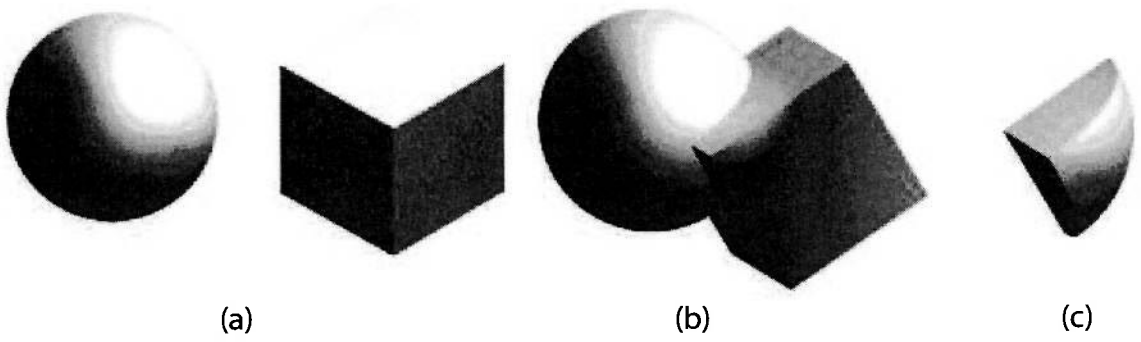


Figura 2.1. Construção de um sólido CSG. Da esquerda para direita, (a) primitivas CSG com formas simples (*bloco* e $esfera$), (b) o *bloco* sofreu transformações afins (rotação e translação) e (c) operação booleana ($bloco \cap esfera$).

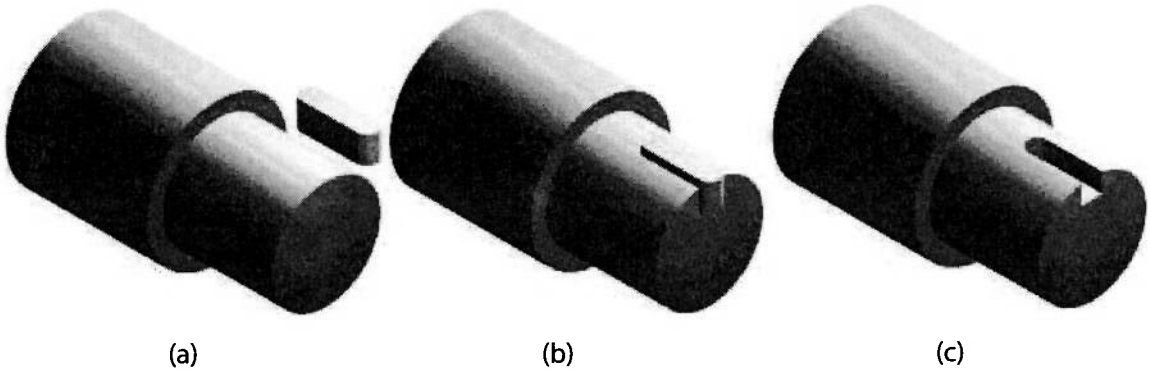


Figura 2.2. Construção de um eixo com um rasgo de chaveta. Da esquerda para direita, (a) primitivas CSG com formas complexas (*eixo* e *chaveta*), (b) a *chaveta* sofreu transformações afins (rotação e translação) e (c) operação booleana ($eixo - chaveta$) para fazer o rasgo da *chaveta*.

linhas em linhas. Note que algumas delas (translação, rotação e reflexão) preservam os comprimentos das arestas e os ângulos entre estas arestas. Outras (como escalamento uniforme), no entanto, preservam os ângulos mas não os comprimentos. Outras ainda (como escalamento não uniforme) não preservam nem os ângulos nem os comprimentos, porém ainda assim mapeiam linhas em linhas.

Todas as transformações acima listadas preservam relações afins básicas, exatamente por isso podem ser classificadas como sendo transformações afins. Por exemplo, dada uma transformação T de um dos tipos acima citados, e dados dois

pontos P e Q, e qualquer escalar α, temos:

$$R = (1 - \alpha)P + \alpha Q \Rightarrow T(R) = (1 - \alpha)T(P) + \alpha T(Q) \tag{2.1}$$

Em outras palavras supondo que R seja o ponto médio da aresta PQ, antes de aplicada a transformação T, então ele continuará a sê-lo após a aplicação da transformação T.

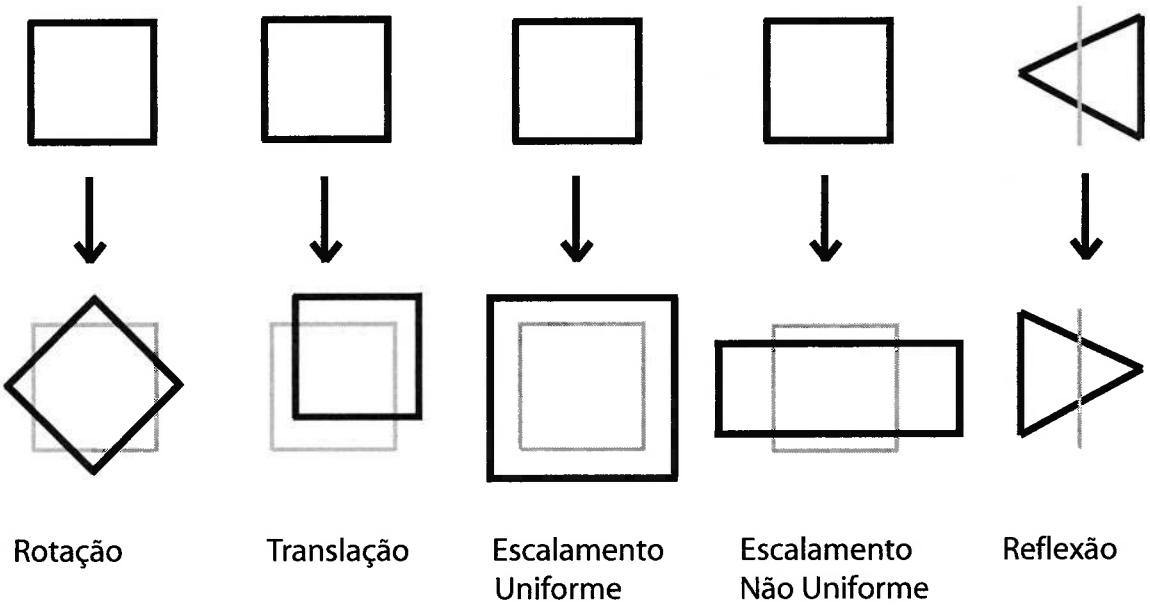


Figura 2.3. Exemplos de transformações afins.

2.2 Operações Booleanas entre Sólidos

Instâncias transformadas podem ser combinadas através de operações Booleanas regularizadas: união, intersecção e diferença. Estas operações Booleanas realizam o respectivo conjunto de operações Booleanas teóricas e, deste modo, transformam o resultado em um conjunto r através da aplicação de operações que isolam o interior do sólido (excluindo a fronteira) seguido de um fechamento topológico (reconstruindo a fronteira do sólido final). Desta forma elas sempre retornam sólidos válidos (embora possivelmente vazios), eliminando possíveis faces internas ao sólido. Ainda que outras operações Booleanas possam ser oferecidas, estas três são convenientes e suficientes, pois dentre as 16 diferentes combinações Booleanas entre dois conjuntos,

A e B, 8 não possuem fronteira, 3 são triviais, e apenas 5 são úteis no âmbito da modelagem de sólidos, são elas:

- $A + B = \{a : a \in A \text{ ou } a \in B\}$,
- $A \cap B = \{a : a \in A \text{ e } a \in B\}$,
- $A - B = \{a : a \in A \text{ e } a \notin B\}$,
- $B - A = \{a : a \notin A \text{ e } a \in B\}$ e
- $(A - B) + (B - A)$ a diferença simétrica

As operações acima citadas estão ilustrado na Figura 2.4.

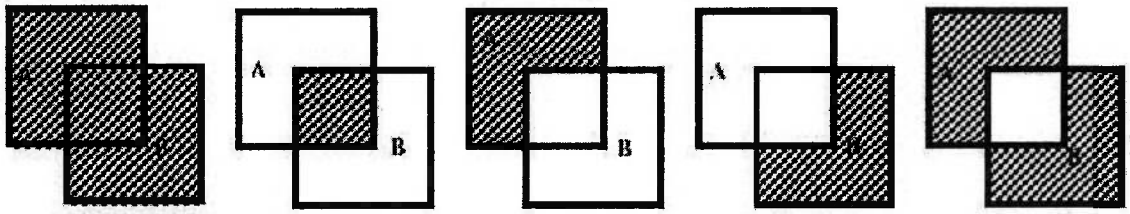


Figura 2.4. As cinco operações Booleanas não triviais para dois conjuntos (da esquerda para a direita): $A + B$, $A \cap B$, $A - B$, $B - A$, e $(A - B) + (B - A)$.

2.3 Árvore CSG

A Figura 2.5 ilustra como uma sintaxe simples pode ser utilizada para especificar um sólido em CSG. O sólido E é definido pela combinação dos sólidos A e B, por meio de uma operação de intersecção. O sólido F é definido pela diferença entre os sólidos C e D. Finalmente, o sólido G é definido pela união dos sólidos E e F. A tradução desta sintaxe nos leva a um grafo com raiz, cujas folhas representam instâncias das primitivas e cujos nós internos representam transformações afins ou operações Booleanas que produzem sólidos. A raiz representa o sólido correspondente ao grafo CSG.

As representações CSG são concisas, sempre válidas no conjunto r do domínio da modelagem, e facilmente parametrizadas e editadas. Muitos algoritmos de modelagem de sólidos trabalham diretamente nas representações CSG através da estratégia

de divisão e conquista, onde resultados calculados nas folhas são transformados e combinados árvore acima de acordo com as operações associadas aos nós intermediários.

As representações CSG não apresentam de forma explícita nenhuma informação a respeito de conectividade, bem como da existência do sólido correspondente. Estas questões topológicas são melhor abordadas por meio de alguma forma de avaliação da fronteira do sólido, onde um B-Rep total ou parcial é derivado por meio de algoritmo do modelo CSG.

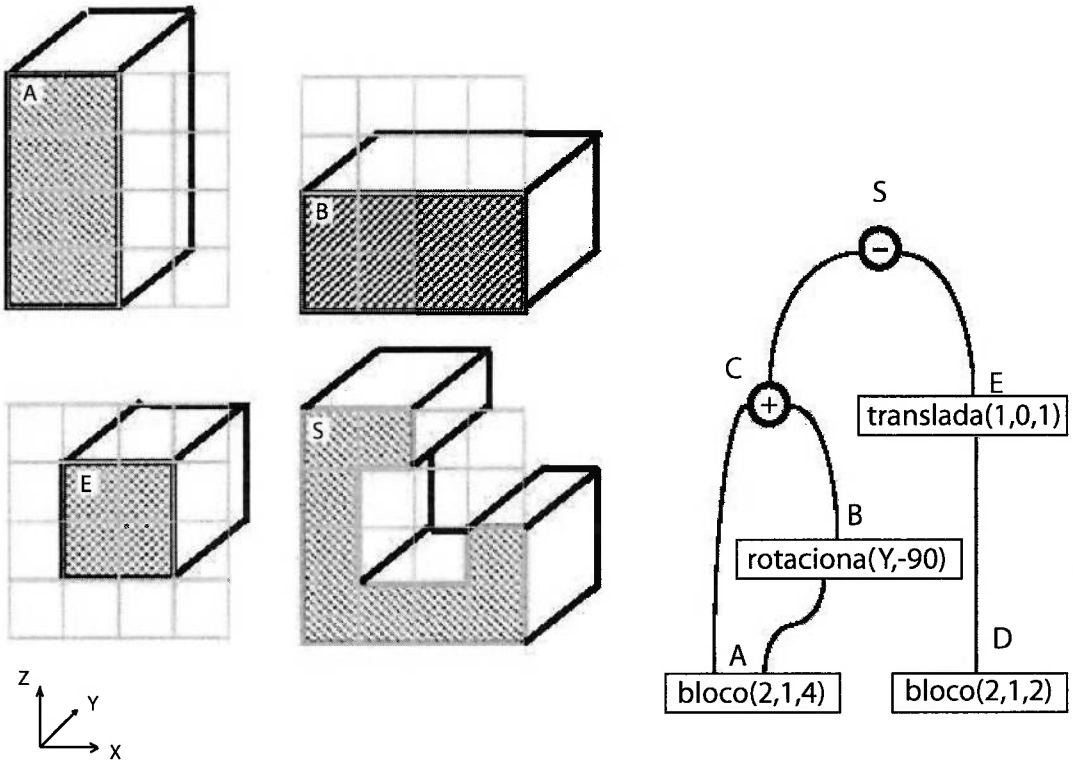


Figura 2.5. As instâncias A, B, e E são mostradas superpostas no mesmo grid de referência. O sólido S foi especificado pela seguinte seqüência de comandos: $A = bloco(2,1,4)$; $B = rotaciona(A, Y - eixo, 90)$; $C = A + B$; $D = bloco(2,1,2)$; $E = translada(D,1,0,1)$; $S = C - E$; O correspondente grafo CSG (direita) possui 2 primitivas nas folhas, 2 nós com transformações, e 2 nós com expressões Booleanas regularizadas.

2.4 Classificação de um Ponto para Sólidos CSG

A árvore CSG pode ser vista como uma descrição implícita da geometria do sólido modelado, que deve ser avaliada para que seja possível gerar a saída gráfica ou para efeito de cálculos. Para isso é preciso que consigamos classificar um dado ponto do espaço como sendo interior ou exterior ao volume ocupado pelo sólido CSG. Como dito antes, a representação CSG é uma árvore. Isso sugere o emprego de métodos de divisão e conquista ou descida em profundidade recursiva para calcular-se a classificação do ponto (como interior ou exterior ao sólido) [5]. O algoritmo está descrito em Algoritmo 1:

Algoritmo 1 Avalia a propriedade P de uma árvore CSG

```
1: /* Avalia a propriedade P de uma árvore CSG */
2: P * Tree_P(CSG_Tree * S, args)
3: {
4: se S->op == <primitiva> então
5:    retorna primitiva_P(S, args)
6: se não
7:    retorna combina_P(Tree_P(S->left, args), Tree_P(S->right, args), S->op);
8: fim se
9: }
10: /* Combina duas avaliações de P com um conjunto de operações OP */
11: P * Combina_P(CSG_Tree * left_P, CSG_Tree * right_P, intOp)
12: { ... }
13: /* Avalia P para a primitiva */
14: P * Primitiva_P(CSG_Tree * S, args)
15: { ... }
```

Tomando o modelo CSG ilustrado na Figura 2.5, e os pontos P e Q indicados na na Figura 2.6, podemos classificá-los, tendo em vista o algoritmo acima, da seguinte maneira:

A árvore será varrida recursivamente até que a primeira primitiva (folha) seja encontrada quando a rotina $P * Primitiva\_P$ é chamada. Neste caso A ($P \in A$),

a próxima primitiva visitada será D ($P \in D$).

Após todas as primitivas da árvore terem sido visitadas verificaremos B, e para tanto devemos levar em conta que $B = rotaciona(A, Y - eixo, 90)$. Na verdade essa transformação já foi aplicada a P quando o algoritmo percorria a árvore descendo em direção à folha A. Aqui é importante notar que a transformação foi aplicada em P e as novas coordenadas de P foram usadas, neste ramo da árvore, deste ponto para baixo. Isso por que não estamos computando o sólido e sim verificando se um ponto é interno ou externo ao sólido. Em outras palavras, neste momento não sabemos como é o sólido, apenas sabemos que, tendo se avaliado a árvore abaixo deste ponto, até agora $P \in B$ e a transformação, na verdade, já tinha sido levada em conta.

Agora temos que classificar P em relação à C, como $C = A + B \Rightarrow P \in C$.

Em relação à E, levando em conta que $E = translada(D, 1, 0, 1)$, conclui-se que $P \notin E$.

Finalmente como $S = C - E$ e $P \in C$ e $P \notin E$ concluímos que $P \in S$.

Abaixo, apresentamos o mesmo raciocínio acima exposto em linguagem matemática. Observe porém que não é esta a forma como o algoritmo trabalha, mas sim como foi acima exposto.

$P \in A$, $P \in B$, $C = A + B$

$\Rightarrow P \in C$

$P \in D$, mas $E = translada(D, 1, 0, 1)$ (vide figura)

$\Rightarrow P \notin E$

$S = C - E$, como $P \in C$ e $P \notin E$

$\Rightarrow P \in S$

De forma análoga podemos concluir que $Q \notin S$.

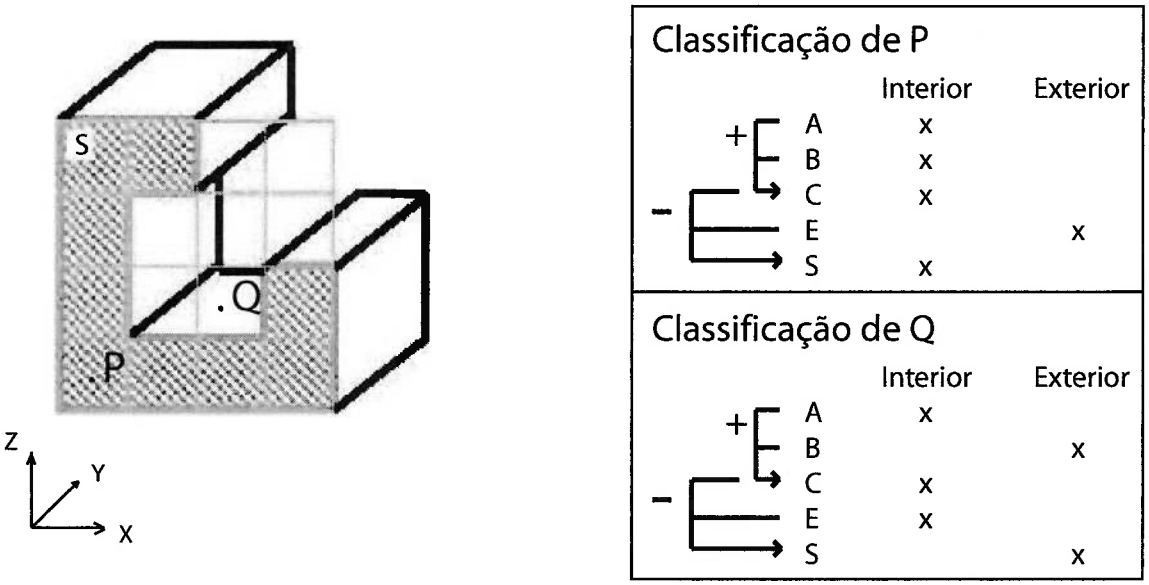


Figura 2.6. Classificação dos pontos P e Q, em relação ao sólido S, no detalhe veja como as operações booleanas afetam a classificação de P e Q em relação á S.

Capítulo 3

Representação B-Rep (Boundary Representation)

Os modelos sólidos B-Rep derivaram dos modelos de poliedros utilizados em computação gráfica na representação de objetos e cenas para remoção de linhas e superfícies ocultas. Eles podem ser encarados como modelos gráficos avançados que tentam resolver alguns problemas através da inclusão de uma descrição completa da superfície da fronteira do objeto.

3.1 Geometria e Topologia do Modelo B-Rep

Existem três entidades primitivas face, aresta e vértice, e a informação sobre a geometria anexada a elas, forma a constituição básica dos modelos B-Rep (Figura 3.1). Além das informações sobre a geometria como equação da face e coordenadas do vértice, um modelo B-Rep deve representar também o relacionamento entre as faces, arestas e vértices.

De acordo com Mäntylä [6], é costume agregar-se toda informação relativas à geometria das entidades sob o termo geometria do modelo de fronteira, e de forma similar, informações relativas a suas interconexões sob o termo topologia. Uma descrição detalhada sobre topologia pode ser encontrada em [7].

Pode-se dizer que a topologia funciona como uma goma onde as informações geométricas são aglutinadas; ou então que "as informações topológicas criam um vigamento no qual as informações geométricas são posicionadas". Como parte da in-

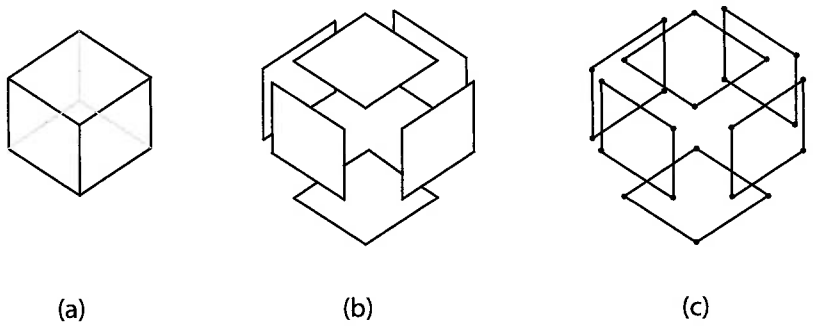


Figura 3.1. Componentes básicos de um modelo por fronteira (B-Rep). Da esquerda para direita, (a) sólido, (b)faces e (c) arestas e vértices.

formação topológica a representação B-Rep precisa armazenar informações de como as faces, as arestas e os vértices de um sólido estão compartilhados (relações de adjacência).

As nove relações de adjacência, definidas pela combinação dois a dois entre os três elementos primitivos da representação B-Rep, podem ser visualizadas na Figura 3.2. Nesta figura, a relação VV é entre um vértice e outros vértices que com este definem uma aresta, a relação AV é entre uma aresta e os dois vértices que a definem, a relação FV é entre uma face e os vértices que a definem, a relação VA é entre um vértice e as arestas limitadas por ele, a relação AA é entre uma aresta e as outras quatro arestas que possuem uma face e um vértice comuns como contorno, a relação FA é entre uma face e as arestas que a limitam, a relação VF é entre um vértice e as faces que o possuem, a relação AF é entre uma aresta e as duas faces que ela separa e finalmente, a relação FF é entre uma face e as outras que a circundam.

3.2 Estruturas de Dados

Nesta seção são apresentadas estruturas de dados capazes de armazenar as informações de geometria e topologia do modelo B-Rep.

A estrutura *winged-edge* mantém as informações de adjacência por meio de ponteiros a vários elementos adjacentes à aresta de referência: duas faces, dois vértices e quatro arestas. Cada uma das quatro arestas compartilha com a aresta de referência um vértice e uma face. (vide Figura 3.3).

Durante um período relativamente longo, apesar da estrutura *winged-edge* ter

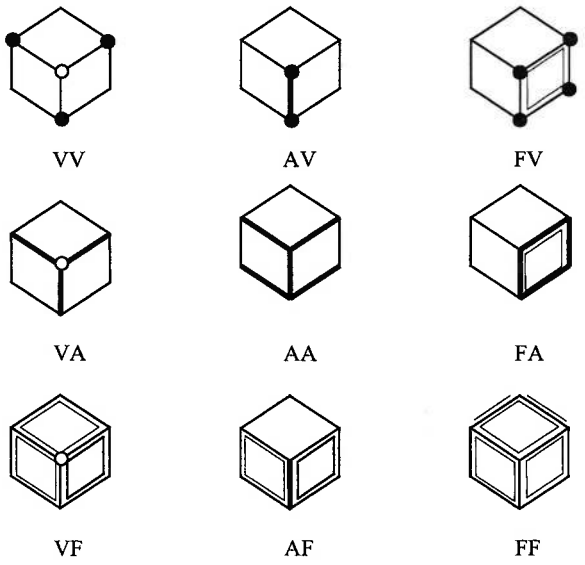


Figura 3.2. Relações de adjacência.

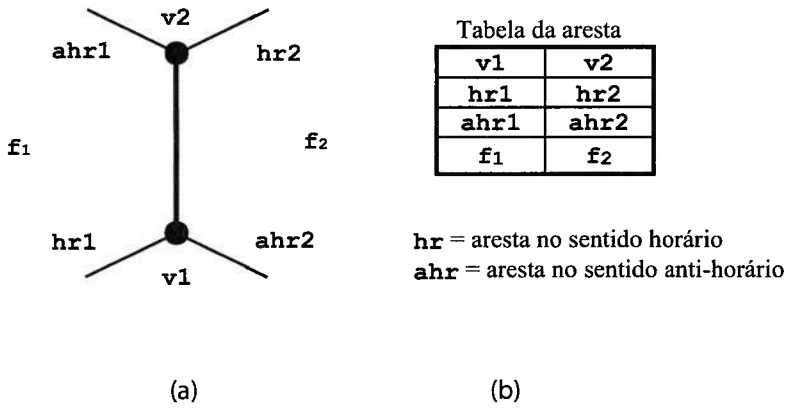


Figura 3.3. Estrutura de dados *winged-edge.*

sido largamente utilizada por pesquisadores de modeladores de sólidos, pouca coisa foi feita para racionalizar analiticamente a sua eficiência. Observe que algumas relações de adjacência envolvem um número variável de elementos. A relação de adjacência FA, por exemplo, requer um número variável de arestas.

A Figura 3.4 ilustra uma pirâmide de quatro faces e possuímos faces com três e quatro arestas adjacentes. Nesta figura é possível observar que as arestas definem um circuito direcional ao redor da face. Por exemplo, a seqüência de arestas a1, a4, a3 e a2 define o circuito direcional das arestas da face f1. Também é possível observar que cada aresta participa em dois circuitos direcionais e em sentidos opostos. Por exemplo, a aresta a1 participa dos circuitos direcionais das faces f1 e f5 em sentidos

complementares.

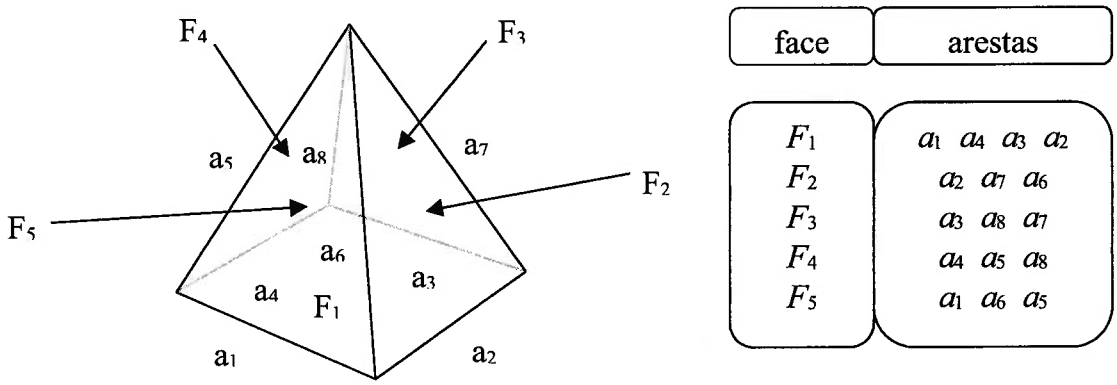


Figura 3.4. Estrutura de dados baseada na relação FA.

Algumas relações de adjacência envolvem um número constante de elementos adjacentes, não importando a situação. A relação de adjacência AV, por exemplo, envolve exatamente dois vértices para cada aresta. É conveniente que as relações de adjacência armazenadas explicitamente na representação possuam um número constante de elementos adjacentes. Este critério aplica-se a três relações de adjacência: AV, AA e AF. Combinando estas três relações de adjacência descobrimos a estrutura *winged-edge*. A estrutura *winged-edge* possui também a relação de adjacência fracional.

Em vez de armazenar todas as arestas pertencentes à relação de adjacência FA para cada face, ela armazena apenas uma aresta pertencente à relação para cada face. A relação de adjacência FA cria um conjunto de armazenamento proporcional ao número de faces e independente do número de arestas associadas às várias faces.

Observou-se que a aresta na estrutura de dados *winged-edge* original [8] desempenhava duas funções principais: representar o circuito de arestas ao redor da face e representar a aresta real. Percorrer o circuito de arestas de uma face é uma operação unidirecional, devido à própria orientação do circuito de arestas.

Na estrutura *winged-edge* cada aresta é utilizada para delimitar duas faces e portanto, cada aresta participa de dois circuitos orientados. Como os dois circuitos possuem orientações opostas entre si, cada aresta é percorrida sempre duas vezes e em direções opostas.

Entretanto, é necessário verificar e determinar a direção em que cada aresta

está sendo percorrida a cada passo do percurso, um processo que aumenta consideravelmente o custo de tempo do procedimento e implicará no desenvolvimento de algoritmos complexos quando estiverem em ambiente curvo. Enfim, por conta dessa deficiência na estrutura de dados, o algoritmo para determinação do circuito de arestas ao redor da face era muito complexo e com muitas regras.

Alguns pesquisadores notaram que separando-se estas duas funções o algoritmo torna-se muito mais simples [9]. Esta separação foi obtida pela divisão de cada *winged-edge* em duas metades.

A conectividade entre ambas as metades é mantida por um ponteiro que referencia a metade oposta. Existem duas possibilidades para realizarmos esta separação: estrutura face-aresta (Figura 3.5) e estrutura vértice-aresta (Figura 3.6).

A definição da estrutura face-aresta foi feita baseando-se na relação de adjacência FA. Em contrapartida, a estrutura vértice-aresta foi baseada na relação de adjacência VA. Em cada estrutura *half-edge* está sendo representada a metade das informações de adjacência da estrutura *winged-edge*.

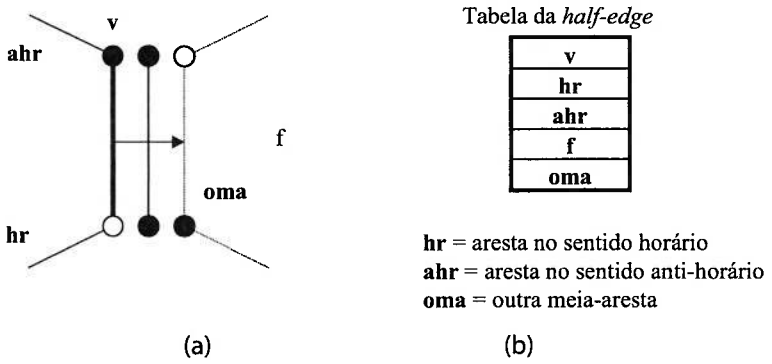


Figura 3.5. Estrutura face-aresta.

Cada *half-edge* possui apenas uma orientação, e cada face possui um circuito direcional de *half-edges*.

Para evitar a duplicidade das informações associadas à aresta (por exemplo, atributo de cor) nas duas *half-edges*, foi proposta a criação de uma nova estrutura para a aresta que associaria a duas *half-edges* que definem a aresta de referência, evitando assim o ponteiro para a outra *half-edge*.

Até este momento, foi assumido que cada face possui apenas um contorno. Entretanto, casos práticos requerem que uma face possua mais de um contorno (como

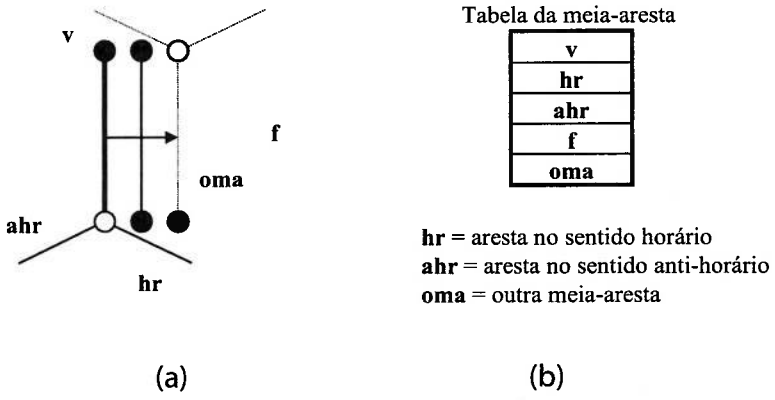


Figura 3.6. Estrutura vértice-aresta.

uma face com furos). Faces com mais de um contorno podem ser simuladas pela técnica de aresta-ponte (*bridge-edge*) no qual uma aresta une os contornos de uma face entre si. A aresta-ponte, portanto, possui a mesma face adjacente em ambas as laterais (vide Figura 3.7(a)).

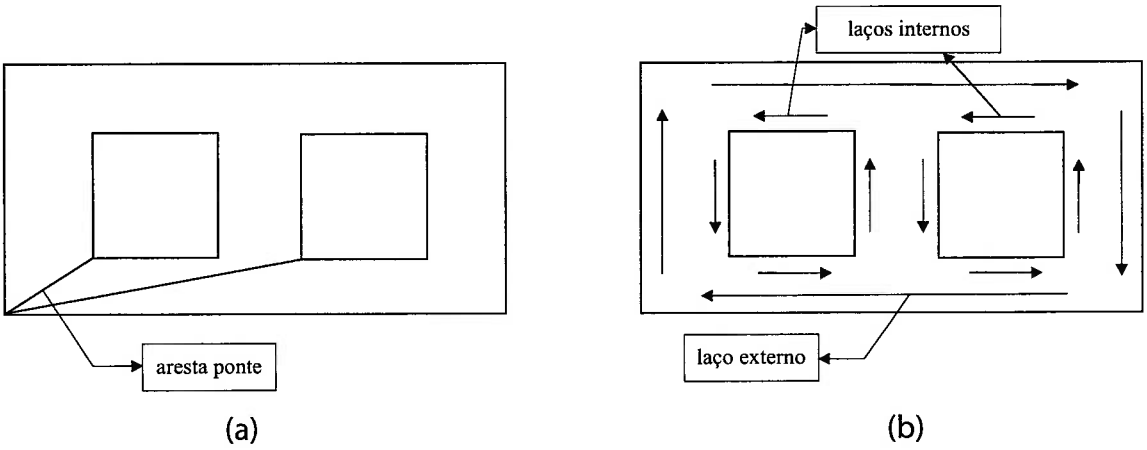


Figura 3.7. Extensão para faces com múltiplos contornos.

Entretanto, a técnica aresta-ponte não é muito eficiente porque será necessário determinar como os contornos devem ser conectados pelas arestas-ponte, o que criará a necessidade de complexos algoritmos para implementar operações de modelagem. Como exemplo, podemos citar as Operações Booleanas que provavelmente interseccionarão as arestas-ponte. Alterações na estrutura *half-edge* de maneira a suportar faces com mais de um contorno não afetarão a estrutura B-Rep ao nível de aresta, mas sim, ao nível de face. Uma técnica muito comum é adicionar uma

estrutura de tamanho fixo chamada laço (loop) que é associada a cada contorno da face. A estrutura laço simplesmente fornece à estrutura face um mecanismo para manter uma lista ligada de ponteiros para os seus múltiplos contornos. Cada face possui um laço externo e zero ou mais laços internos (vide Figura 3.7(b)).

Devido ao formalismo das Operações Booleanas, ao combinarmos dois sólidos por uma Operação Booleana, o resultado será sempre apenas um sólido. Entretanto, mesmo em situações especiais, como a situação exemplificada na Figura 3.8, onde o resultado da Operação Booleana aparenta apresentar dois sólidos, o resultado é considerado como sendo apenas um sólido. Entretanto, nesta situação em especial, considera-se que o sólido resultante possui dois shells. Em outras palavras, para representar esta situação especial, foi criado o elemento shell que representa conjuntos desconexos de faces no espaço. A Figura 3.9 ilustra os elementos primitivos de um modelo sólido. A Figura 3.10 ilustra a hierarquia da estrutura *half-edge* resultante.

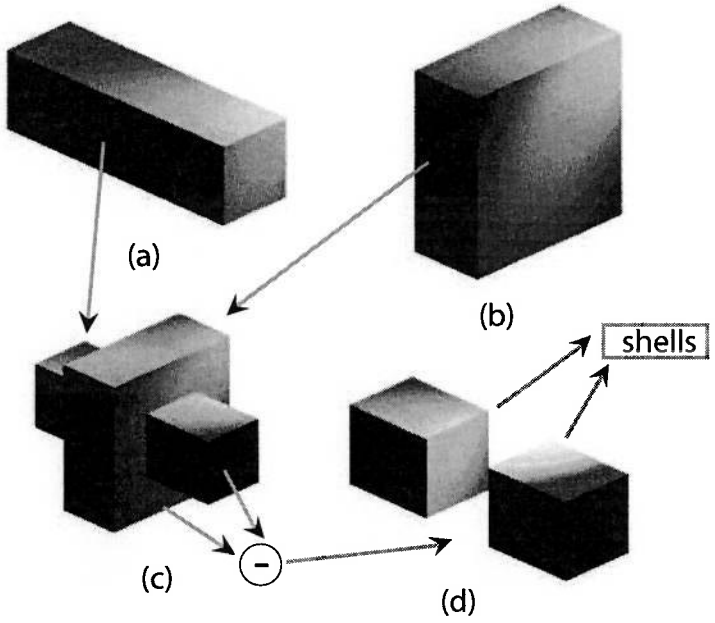


Figura 3.8. Sólido com dois shells. Em (a) e (b) observamos a primitivas, em (c) elas estão posicionadas, em (d) podemos ver o resultado da operação booleana de subtração das primitivas.

Assim com o objetivo de simplificar os algoritmos, a entidade *half-edge* foi criada. Deste modo os modeladores de sólido modernos possuem uma entidade para representar a aresta em si e outra entidade para representar o circuito de arestas

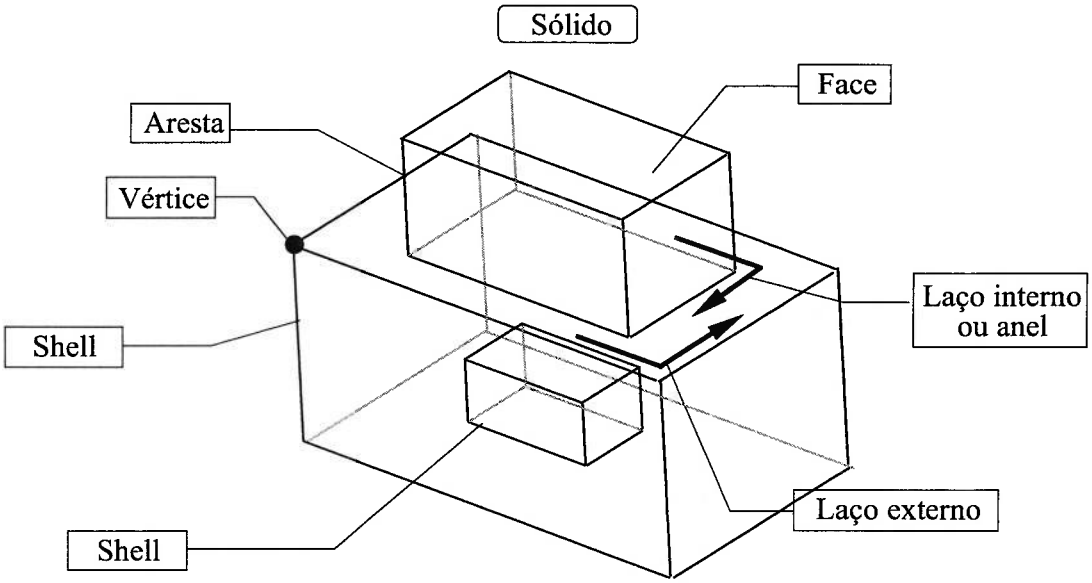


Figura 3.9. Elementos topológicos de um modelo sólido.

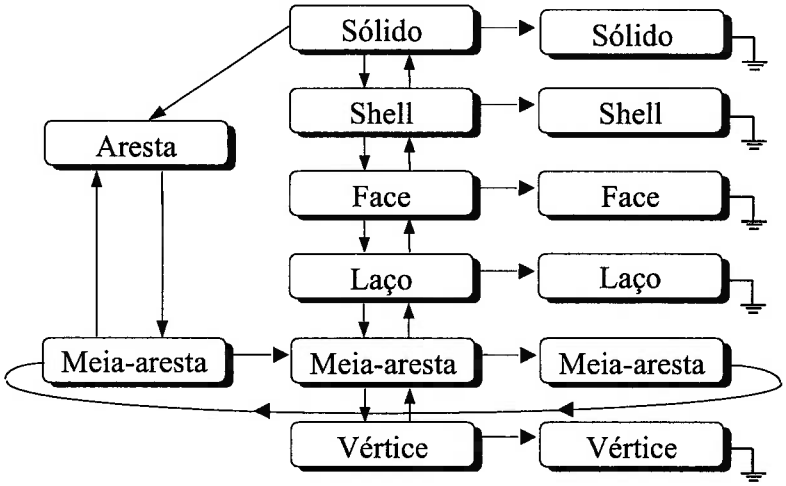


Figura 3.10. Vista hierárquica da estrutura *half-edge*.

ao redor da face. Pelo fato de o modelo B-Rep poder tornar-se mais complexo, se faz necessário a adição de uma nova primitiva: laço. A face pode possuir buracos interiores para representar protuberâncias ou depressões. Neste caso a face possui um laço externo e zero ou mais laços internos.

3.3 Operadores de Euler

Os Operadores de Euler foram introduzidos originalmente por Baumgart [8] juntamente com a estrutura de dados *winged-edge*. Para permitir a manipulação das entidades topológicas e ao mesmo tempo garantir a validade do modelo sólido, os operadores de Euler são utilizados satisfazendo a equação de Euler. A equação de Euler-Poincaré relaciona entidades primitivas que formam o sólido de maneira quantitativa:

$$v - a + 2f = 2(s - h) + l \tag{3.1}$$

Onde v é o número de vértices, a é o número de arestas, f é o número de faces, s é o número de *shells*, h é o número de furos e l é o número de laços. A forma mais conhecida na literatura da equação de Euler-Poincaré supõe ainda a existência de r anéis (ou laços internos) no sólido, onde $r = l - f$. Ficando a equação da seguinte forma:

$$v - a + f = 2(s - h) + r \tag{3.2}$$

Foi provado por Mäntylä [6], que os operadores de Euler formam um conjunto completo de primitivas para modelagem de sólidos $manifold$. Mais precisamente, todos poliedros topologicamente válidos podem ser construídos a partir de um poliedro inicial por meio de uma seqüência finita de operadores de Euler. Eles permitem que a construção do sólido possa ser executada passo a passo, escondendo todos os detalhes de implementação da estrutura de dados. Isto torna os operadores de Euler muito poderosos no âmbito da modelagem de sólidos.

Com a finalidade de facilitar a memorização, os Operadores de Euler estão definidos utilizando a nomenclatura da Tabela 3.1 . Por exemplo, o nome MEV deve ser traduzido por Make Edge, Vertex (Cria uma aresta e um vértice)

Durante o processo de construção de um sólido pela utilização de Operadores de Euler, a validade topológica do mesmo é mantida observando-se a equação de Euler-Poincaré. Entretanto, é comum o agrupamento de Operadores de Euler em uma certa seqüência para mantermos também a geometria válida. É importante observar que não há como manter a geometria válida em todos os estágios da construção. Por

Tabela 3.1. Nomenclatura dos Operadores de Euler.

Símbolo	Significado	Tradução
M	make	cria
K	kill	remove
V	vertex	vértice
E	edge	aresta
F	face	face
S	shell	shell
H	hole	furo
R	ring	laço

isto, os Operadores de Euler devem ser agrupados em seqüências que possuam algum significado.

Vários autores demonstram que seis operadores são suficientes para construir todos os objetos. Enquanto estes seis operadores podem ser escolhidos de várias maneiras, considerações de modularidade e independência criaram apenas pequenas variações na coleção encontrada na literatura. A seguir foram selecionados e estão descritos os seis operadores mais comumente utilizados na literatura, e para cada qual o inverso de seu par, uma representação gráfica para cada operador pode ser observada na Figura 3.11.

1. Criação de Sólidos: um vértice, face e sólido são criados (<**MVSF**> - Make Vertex Solid Face);

2. Remoção de Sólidos: um vértice, face e sólido são removidos (<**KVSF**> - Kill Vertex Solid Face);

3. Divisão de Vértices: um vértice é dividido em dois vértices conectados por uma aresta (<**MEV**> - Make Edge Vertex);

4. União de Vértices: dois vértices vizinhos são unidos em um único e a aresta entre eles é removida. Essa operação é a inversa da divisão de vértices (<**KEV**> - Kill Edge Vertex);

5. Divisão de Faces: a face é dividida em duas por meio da adição de uma nova aresta entre dois vértices de seu laço externo (<**MEF**> - Make Edge Face);

6. União de Faces: duas faces adjacentes são unidas através da remoção da aresta comum à elas. Essa operação é a inversa da divisão de faces (<**KEF**> - Kill Edge Face);

7. Divisão de Laços: um laço é dividido em dois laços, um deles sendo uma nova fronteira interior do laço, através da remoção de uma aresta. Este operador, por conseguinte irá criar um novo laço interno (<**KEMR**> - Kill Edge Make Ring);

8. União de Laços: dois laços, ao menos um sendo um laço interno, são unidos para produzir um só laço. desta forma, um laço interno é removido. Essa operação é a inversa da divisão de laços (<**MEKR**> - Make Edge Kill Ring);

9. Criação de Furos: une duas faces, e cria um furo passante. Este é um operador necessário quando um toróide é criado (<**KFMRH**> - Kill Face Make Ring Hole);

10. Remoção de Furos: cria uma nova face e remove o o furo passante (<**MFKRH**> - Make Face Kill Ring Hole).

11. Criação de um novo *shell*: Transforma o anel de uma face em uma nova face, e todo o conjunto de faces associadas à nova face constituirá um novo *shell* (<**MSFKR**> - Make Shell Face Kill Ring);

12. Remoção de Shells: Transforma uma face em um anel, e o *shell* a qual essa face pertencia é removido. (<**KSFMR**> - Kill Shell Face Make Ring).

Através de um exemplo simples, um bloco retangular com um furo passante retangular, é possível ilustrar a utilização dos Operadores de Euler (Figura 3.12).

3.3.1 Semântica dos Operadores de Euler

Os Operadores de Euler possuem um conjunto de parâmetros de entrada, que permitem definir como será o modelo sólido após a sua aplicação. Assim saberemos onde está o novo vértice e aresta definida pelo Operador de Euler MEV, como por exemplo na Figura 3.13, onde podemos ver como os parâmetros *he*1 (*half-edge* 1) e *he*2 (*half-edge* 2) afetam o resultado da aplicação do operador MEV. O vértice

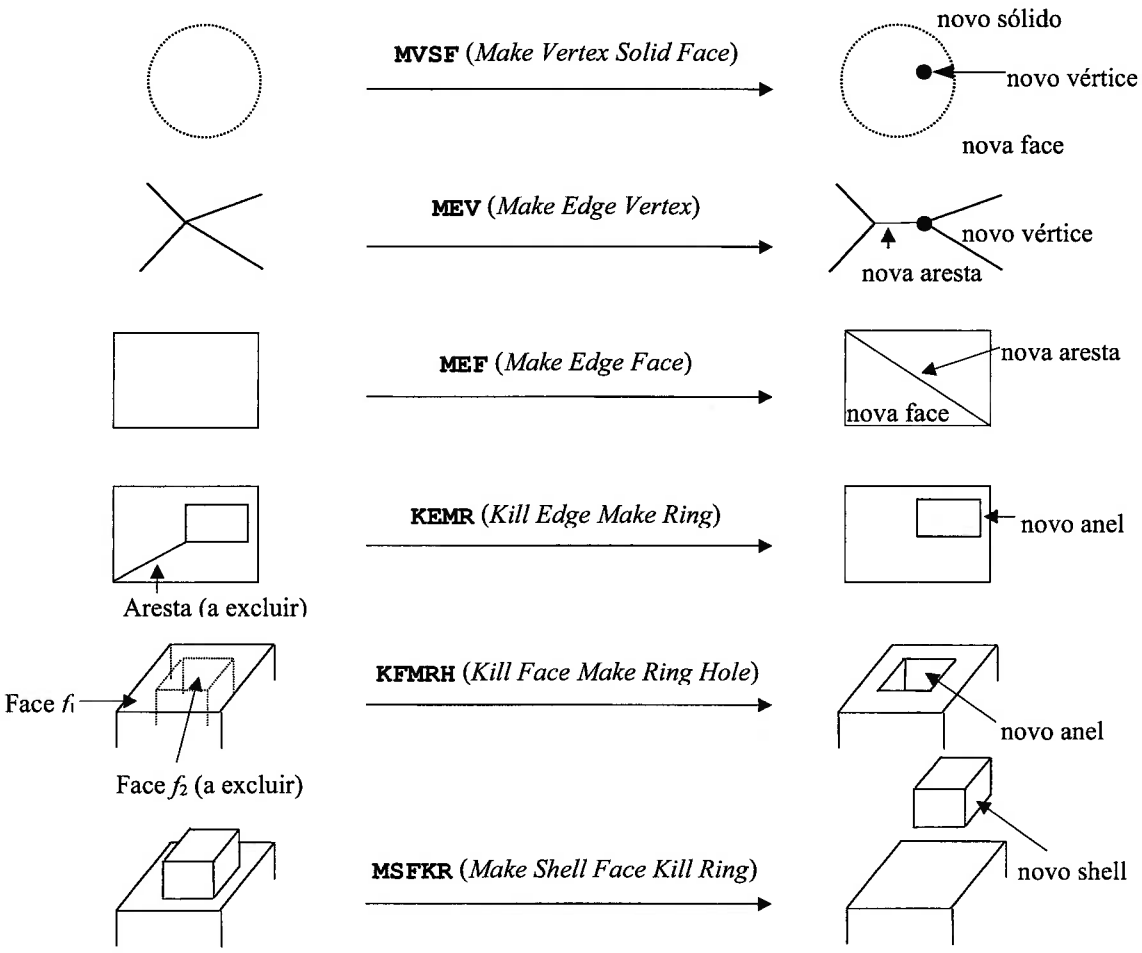


Figura 3.11. Representação gráfica dos Operadores de Euler.

antigo permanece associado à *half-edge* $he2$ para a aplicação deste operador é necessário que as duas *half-edges* ($he1$ e $he2$) estejam inicialmente associadas ao mesmo vértice, elas podem ser diferentes (Figura 3.13 (a)) ou iguais (Figura 3.13 (b)).

A Figura 3.14 ilustra a aplicação do Operador de Euler MEF. A face antiga permanece associada à *half-edge* $he2$, é necessário que as duas *half-edges* ($he1$ e $he2$) estejam inicialmente associadas ao mesmo laço, elas podem ser diferentes (Figura 3.14 (a)) ou iguais (Figura 3.14 (b)).

O Operador de Euler MEKR (Figura 3.15), necessita que os *half-edges* $he1$ e $he2$ estejam na mesma face porém em laços distintos. Ou seja $he1$ é necessariamente diferente de $he2$.

O Operador de Euler KFMRH (Figura 3.16), necessita que sejam fornecidas duas faces, sendo que, a face $f2$ não pode possuir laços internos. A face $f2$ será removida e se transformará em um laço interno de $f1$.

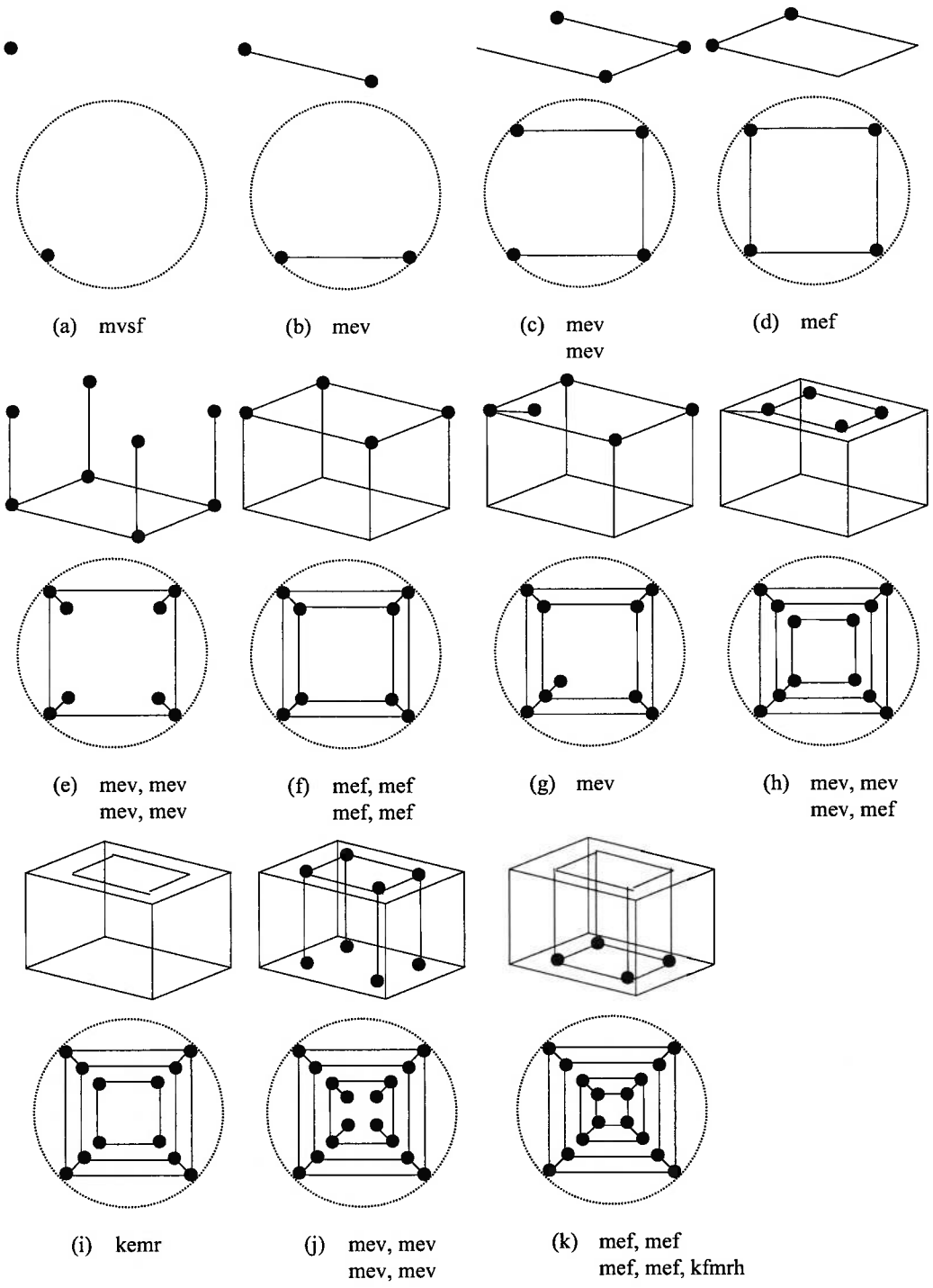


Figura 3.12. Etapas da construção de um bloco retangular com um furo passante retangular.

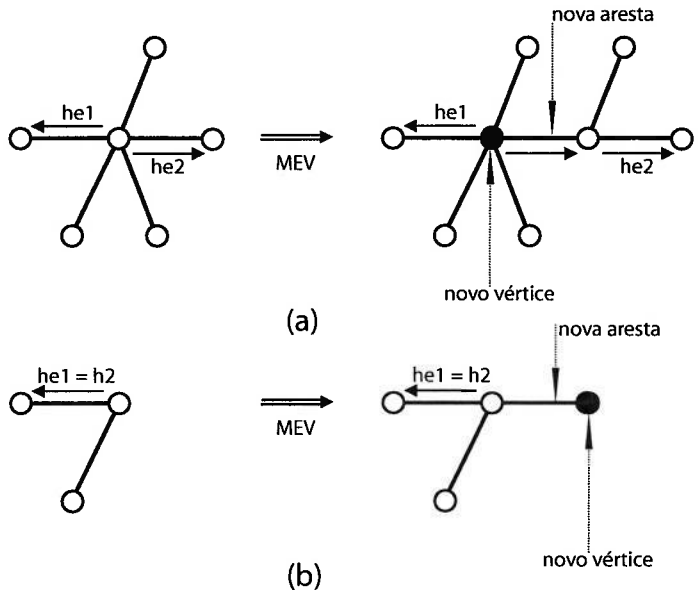


Figura 3.13. Semântica do operador MEV, na figura (a) *he*1 e *he*2 são diferentes e na figura (b) *he*1 e *he*2 são iguais.

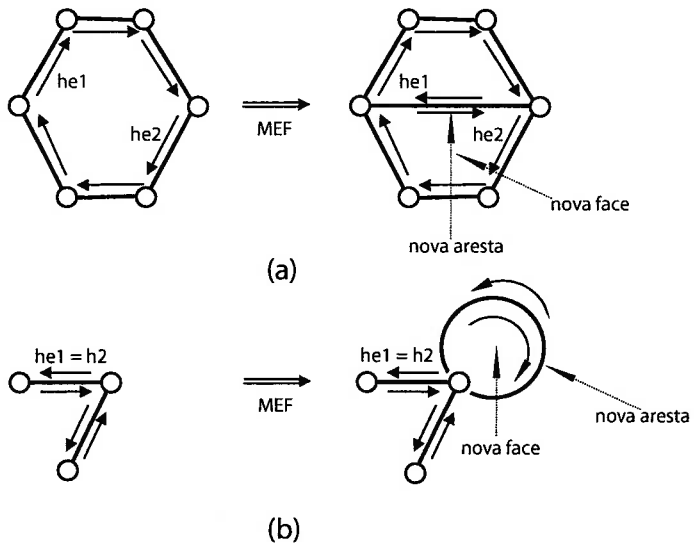


Figura 3.14. Semântica do operador MEF, na figura (a) *he*1 e *he*2 são diferentes e na figura (b) *he*1 e *he*2 são iguais.

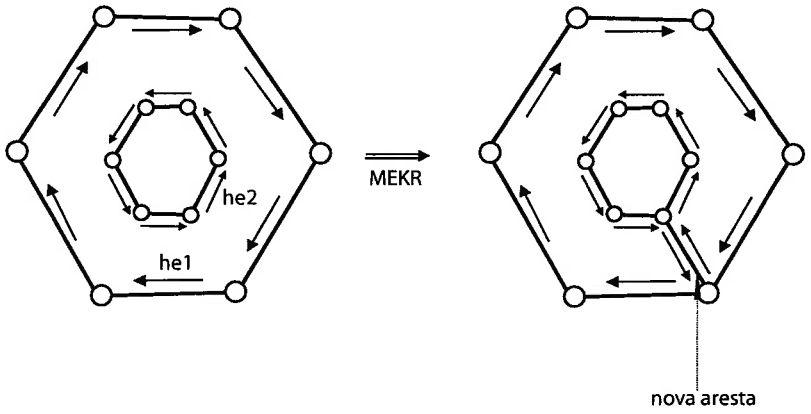


Figura 3.15. Semântica do operador MEKR.

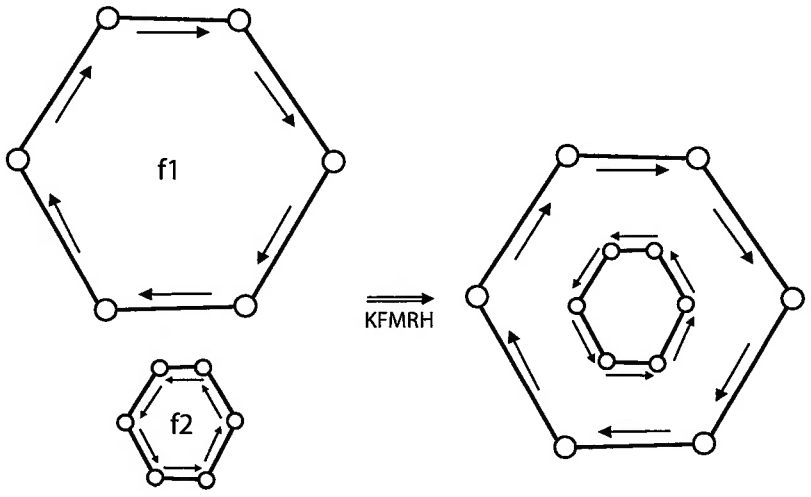


Figura 3.16. Semântica do operador KFMRH.

Capítulo 4

Algoritmo Marching Cubes

Tradicionalmente o Marching Cubes é um algoritmo utilizado para gerar superfícies triângularizadas para fim de visualização a partir de dados volumétricos. Estes dados volumétricos podem ser obtidos através de métodos como Tomografia Computadorizada, Ressonância Magnética, Imagens de Microscópios ou por meio de amostragem de funções implícitas ou procedurais. Atualmente existem estudos para utilizar esse algoritmo na visão estereoscópica [10]. Como originalmente o Algoritmo foi escrito por Lorensen e Cline [11] para ser usado tendo como entrada imagens médicas obtidas por meio dos exames de Tomografia Computadorizada e Ressonância Magnética, pautaremos nossa explicação do método nesse tipo de entrada que são pilhas de imagens 2D em tons de cinza.

O algoritmo Marching Cubes usa a metodologia de divisão e conquista para localizar a isosuperfície, escolhido um isovalor, em um cubo canônico (lado unitário) criado a partir de oito pixels, quatro pixels de cada secção adjacentes (imagens 2D) obtidas no exame [11], vide Figura 4.1.

O algoritmo determina como a isosuperfície intersecta esse cubo, e move (ou marcha) para o próximo cubo. Para achar a intersecção da isosuperfície no cubo, atribui-se o valor *um* para o vértice cujo valor (nível de cinza do pixel) seja maior ou igual ao valor da isosuperfície que estamos construindo. Estes vértices estão localizados dentro (ou na) isosuperfície procurada. Vértices do cubo com valor inferiores ao da isosuperfície buscada recebem valor *zero* e estão localizados fora da isosuperfície. A isosuperfície intersecta as arestas do cubo que possuam um vértice fora da isosuperfície (*zero*) e o outro dentro da isosuperfície (*um*), desta maneira

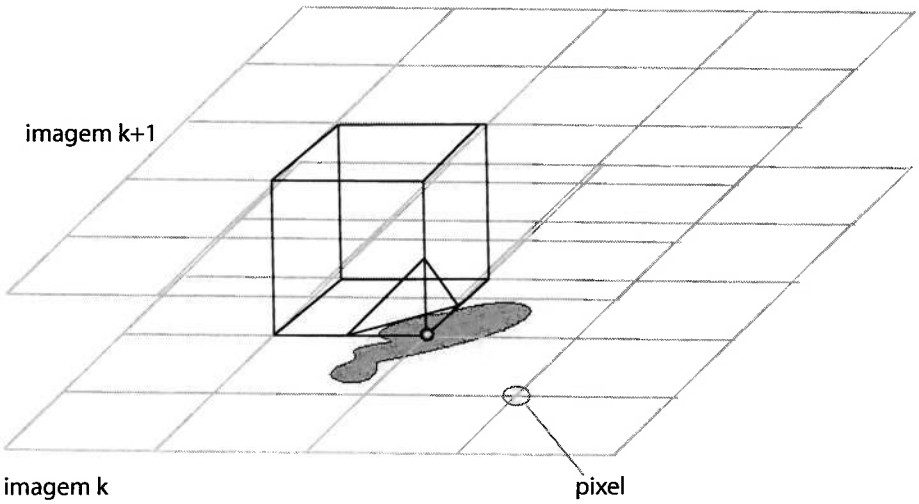


Figura 4.1. Posicionamento do cubo entre duas imagens 2D.

somente cubos que possuam um ou mais vértices com valor *um* e um ou mais vértices com valor *zero* são intersectados pela isosuperfície. Partindo-se destes pressuposto, determina-se geometria da isosuperfície dentro de um cubo, criando-se faces triangulares que dividam o cubo em regiões interiores e regiões exteriores à isosuperfície. Após processar-se todos os cubos intersectados pela isosuperfícies, conecta-se todas estas faces triangulares de todos os cubos na fronteira da isosuperfície obtendo-se assim, uma representação do objeto tridimensional original formada por uma superfície triângularizada.

4.1 Marching Cubes Convencional

Existem 256 possibilidades de configuração do cubo, em cada uma delas, a isossuperfície é triangulada. O algoritmo weaving wall [12] foi proposto junto com todas as 256 possibilidades definidas explicitamente. Este método, entretanto, é muito suscetível a erro. Na implementação convencional do marching cubes, o uso de simetria reduz o número de casos para 15 (vide Figura 4.2). A simetria complementar é definida como a equivalência entre configurações complementares [11]. Duas configurações são definidas como complementar, se a ação do operador lógico NOT sobre todos os pontos e uma configuração se transformar na outra.

A essência do algoritmo marching cubes convencional é que para cada configuração de cubo determinada em uma imagem, uma busca em uma tabela com

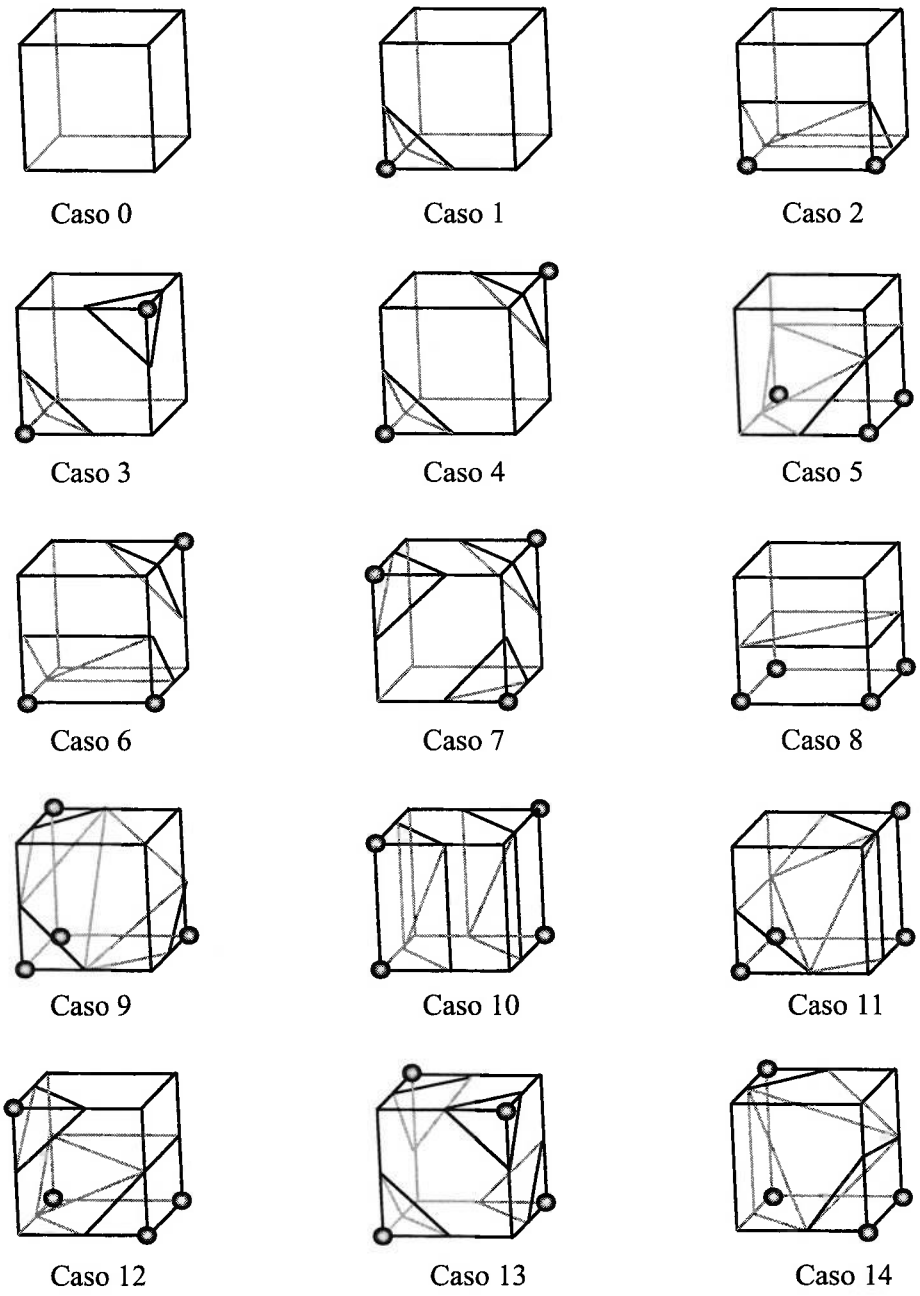


Figura 4.2. Triangulação dos 15 casos de configuração do cubo pré-definidos.

configurações pré-definidas de cubo deve ser feita para determinar uma configuração de cubo pré-definida equivalente ou equivalente ao complementar, para que os triângulos sejam determinados. Para decidir qual configuração de cubo pré-definida é equivalente à configuração de cubo encontrada é necessário rotacionar o cubo de 90 graus em cada uma das direções e compará-lo com as configurações de cubo pré-definidas. A Figura 4.2 exibe as 15 configurações com as triangulações pré-definidas da isossuperfície como definido na referência [1]. As esferas representam voxels da

imagem com valores acima do threshold e as linhas os triângulos produzidos.

O algoritmo marching cubes padrão possui uma falha conhecida [1] bem como um grande número de triângulos produzidos e um custo computacional extra devido às transformações necessárias para encontrar a configuração do cubo com o conjunto pré-definido de triângulos. As diversas configurações facilitam o surgimento de erros, conforme salientado por Delibassis [1].

O uso da simetria que reduz o número de configurações de cubo pode produzir incoerências topológicas, ou furos em determinadas situações de cubos adjacentes conhecido como problema do furo de tipo A. Um certo problema topológico com a ambiguidade entre as conexões dos triângulos é apresentada na literatura. Uma modificação ao algoritmo marching cubes foi proposta de modo a resolver o problema do furo de tipo A, definindo três novas configurações para o cubo. Estas três novas configurações são os complementos para os casos 3, 6 e 7. A Figura 4.3 ilustra a solução para o caso do furo de tipo A.

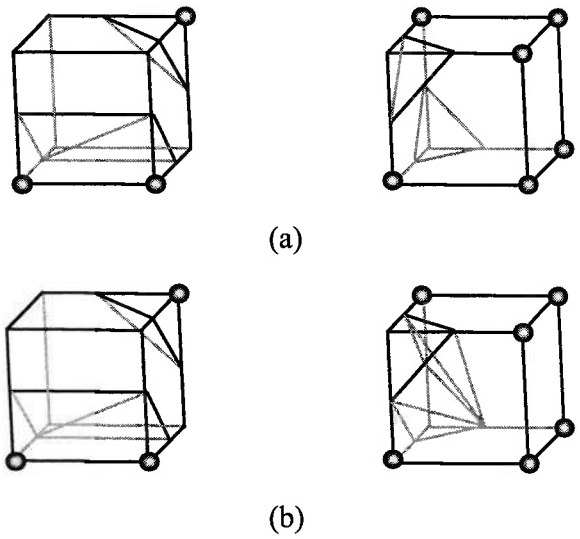


Figura 4.3. (a) Exemplo do problema do furo de tipo A produzido pelo algoritmo convencional do marching cubes, quando o caso 6 (esquerda) e o caso 3 (direita) são adjacentes. (b) Solução para o problema de acordo com as configurações propostas por Delibasis et al. [1].

4.2 Proposta de Delibasis para o Marching Cubes

A tabela com configurações pré-definidas de cubo é uma constante em praticamente todas variações do algoritmo Marching Cubes. Porém o algoritmo proposto por Delibasis et al. [1] não apresenta a tabela com as configurações pré-definidas. Para evitar a comparação com as configurações pré-definidas foi definido um algoritmo para ordenar as intersecções das arestas do cubo de modo a definir polígonos algoritmicamente. Note que a saída do algoritmo de Delibasis é uma superfície formada por polígonos e não triângulos como na implementação original.

Partindo-se dos cubos com seus vértices já classificados como interior ou exterior à isosuperfície, observa-se as configurações das faces dos cubos como ilustrado na Figura 4.4, sempre que uma aresta possui um vértice dentro da isosuperfície e o outro fora a aresta é intersectada pela isosuperfície, e portanto no ponto médio da aresta existe um isoponto (ponto pertencente à isosuperfície). Toma-se o ponto médio pois pressupõe-se que a imagem, inicialmente em tons de cinza, tenha sido binarizada.

4.2.1 Definindo-se a Existência de uma Aresta

Arestas são geradas conectando-se dois isopontos situados numa mesma face do cubo levando-se em consideração as seguintes regras ilustradas na Figura 4.4:

- **Condição 0.** Os dois isopontos devem estar situados na mesma face do cubo;
- **Condição 1.** Os dois isopontos devem compartilhar um vértice adjacente classificado como interno;
- **Condição 2.** Os dois isopontos devem possuir vértices adjacentes (um interno e o outro externo) de forma que um vértice adjacente classificado como interno deve ser adjacente a um classificado como interno e a outro classificado como externo;
- **Condição 3.** Os dois isopontos devem compartilhar um vértice adjacente externo e os outros três vértices da face devem ser classificados como internos.

Os seguintes casos devem ser testados para decidir se uma aresta deve ser gerada ou não, dados dois isopontos.

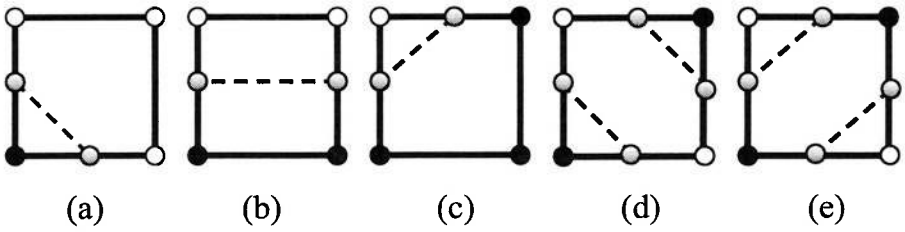


Figura 4.4. Isoponto localizado no ponto médio da aresta do cubo e as condições para existência (vértices em lados opostos da fronteira).

- **Caso 1.** (Condição 0) E (Condição 1), Figura 4.4 situação (a), (d) e (e);
- **Caso 2.** (Condição 0) E (Condição 2), Figura 4.4 situação (b);
- **Caso 3.** (Condição 0) E (Condição 3), Figura 4.4 situação (c);

Tendo definido todos os casos nos quais uma aresta é gerada e adicionadas ao polígono em construção, nós podemos construir um algoritmo onde um isoponto é dado como entrada e o algoritmo tenta localizar o próximo isoponto na seqüência que será o vértice que conectado ao fornecido formará uma nova aresta para gerar o contorno do polígono. Note que o Algoritmo 2 implementa os 3 casos acima e tem por intuito apenas facilitar seu entendimento:

Algoritmo 2 *ProximoIsoponto*($ip1$)

```
1: para todos isopontos não marcados faça
2:    se existe um isoponto ip2 tal que (Condição 0) E ((Condição 1) OU (Condição
      2) OU (Condição 3) então
3:       retorna ip2
4:    se não
5:       retorna nulo
6:    fim se
7: fim para
```

O Algoritmo 2 permite a criação de arestas, este algoritmo fornece os isopontos que formarão as arestas até que o polígono esteja pronto, então o algoritmo retorna *nulo*, linha 5. O algoritmo que se encarrega de fornecer um isoponto e receber o próximo e formar a aresta ou então finalizar o polígono (retorno *nulo*), é o algoritmo principal de Delibasis mostrado no Algoritmo 3.

Algoritmo 3 *AlgoritmoPrincipaldeDelibasis*

1: **para** todos voxels da imagem **faça**
2: **:Passo 0**
3: O cubo de lado unitário é posicionado segundo a Figura 4.1
4: **para** todos arestas do cubo **faça**
5: /* Nesse laço gera-se os isopontos nas arestas do cubo */
6: **se** um vértice do cubo é interno e o outro externo **então**
7: cria-se o isoponto nesta aresta do cubo e coloca-se ele em uma lista
8: **fim se**
9: **fim para**
10: $p = 0$ /* Numeração dos polígonos */
11: **:Passo 1**
12: varre-se alista de isopontos até que o primeiro isoponto não marcado é encontrado
13: **se** não existem mais isopontos não marcados **então**
14: VAI PARA O **Passo 0**
15: **se não**
16: /* Nesse laço gera-se os polígonos */
17: $p = p + 1$
18: marca-se o primeiro isoponto não marcado como $IsopontoAtual$
19: marca-se o $IsopontoAtual$ como pertencente ao polígono p
20: $NovoIsoponto = ProximoIsoponto(IsopontoAtual)$
21: **enquanto** ($NovoIsoponto$!= $nulo$) **faça**
22: $NovoIsoponto = ProximoIsoponto(IsopontoAtual)$
23: $IsopontoAtual = NovoIsoponto$‘
24: marca-se o $IsopontoAtual$ como pertencente ao polígono p
25: **fim enquanto**
26: **fim se**
27: coloca-se o polígono p em uma lista
28: VAI PARA O **Passo 1**
29: **fim para**

Cada cubo pode gerar no máximo 4 polígonos (caso 14, vide Figura 4.2), e nesse caso todos serão triângulos.

Além disso o algoritmo pode gerar polígonos com mais de 3 vértices e que por vezes não estão contidos num plano (vide Figura 4.2, casos 5, 11, 12 e 14).

A ambiguidade presente nos casos mostrados na Figura 4.4 situação (*c*) e (*d*) não é resolvida por Delibasis e é esta ambiguidade que origina o problema do Furo Tipo A (vide Figura 4.3), também presente no algoritmo original.

As deficiências acima expostas não são um problema para efeito de visualização, mas para a criação de um modelo Sólido B-Rep, precisamos gerar uma malha de polígonos topologicamente consistente, uma análise bem detalhada dos problemas de topologia associados ao uso do algoritmo Marching Cubes foi realizada no trabalho de Varadhan et al [13].

Para nosso problema caso fosse gerada uma malha não consistente teríamos que posteriormente aplicar algoritmos para correção da malha como o proposto por Nooruddin [14] para só então poder gerar o modelo sólido. Para evitar esse pós-processamento essas deficiências são corrigidas pelo algoritmo proposto, como será visto adiante.

Capítulo 5

Algoritmo de Conversão de CSG para B-Rep

Neste capítulo partimos do pressuposto que possuímos um sólido CSG, conforme ilustrado no Capítulo 2 e que portanto possuímos uma estrutura de dados em forma de árvore conforme ilustrado na seção 2.3 (Árvore CSG).

A parte inicial do algoritmo é similar ao algoritmo proposto por Delibasis [1], detalhado no Capítulo 4 na seção 4.2, o espaço 3D é discretizado por uma malha de cubos de aresta unitária.

5.1 Isocubo

Começamos a marchar na malha de cubos a partir de um isocubo (cubo intersecionado pela fronteira do sólido), e marcharemos somente de um isocubo para outro, isso será detalhado mais adiante. Desta maneira marcharemos apenas pela fronteira do sólido. Como marcharemos apenas de um isocubo para outro e apenas o processamento destes isocubos diz respeito a este trabalho, deste ponto do texto em diante toda vez que nos referirmos a um *cubo* leia-se *isocubo*, por vezes também utilizaremos a palavra *isocubo*.

Para sabermos se um cubo é um isocubo classificamos todos os vértices que formam esse cubo em relação ao sólido (interior ou exterior), se todos os vértices possuírem a mesma classificação o cubo ou está totalmente dentro ou está totalmente fora do sólido. Se um dos vértices for classificado diferente de qualquer outro vértice

desse mesmo cubo, o cubo é intersecionado pela fronteira do sólido e portanto é um isocubo. Para saber se um vértice é interior ou exterior ao cubo é necessário tomar-se as coordenadas do vértice e utilizar a árvore CSG (vide Capítulo 2 Seção 2.4).

Sabendo se um dado cubo é isocubo ou não podemos prosseguir com nosso algoritmo. Como já explicado no Capítulo 4, o algoritmo de Delibasis gera polígonos e não triângulos e estes polígonos nem sempre estão contidos num plano. Portanto se fez necessário inserir uma etapa de triangularização no algoritmo.

Além de se triangularizar os casos de polígonos não planos, casos 5, 11, 12 e 14 (vide Capítulo 4, Figura 4.2), triangularizamos também os casos 2, 6, 8, 9 e 10, nestes últimos a triangularização não é estritamente necessária pois os polígonos estão contidos num plano, porém ainda sim faremos a triangularização para deixar o algoritmo mais rápido, e para possibilitar futura redução da quantidade de faces, é interessante que todas as faces sejam triangulares.

5.2 Nomeação Única para Isocubos, Isopontos, Arestas e Faces no espaço Volumétrico

5.2.1 Nomeação para os Isocubos

Para facilitar a nossa marcha pelo espaço definimos um algoritmo para associar cada isocubo a um único nome. Para tanto toma-se um vértice do isocubo que representará o isocubo no espaço, convencionou-se tomar-se o vértice com menores valores de coordenadas X, Y e Z. Desta maneira dado um cubo no espaço, o seu nome ou identificador único será expresso pela equação abaixo:

$$isocubo.ID = isocubo.X + isocubo.Y * 256 + isocubo.Z * 256^2 \qquad (5.1)$$

Onde $isocubo.ID$ é o identificador do isocubo e $isocubo.X$, $isocubo.Y$ e $isocubo.Z$ são as coordenadas do isocubo. Observe que o resultado desta equação é sempre um valor inteiro posto que os valores isocubo.X, isocubo.Y e isocubo.Z são sempre inteiros, uma vez que os cubos possuem aresta unitária. Como o espaço discretizado corresponde a um grid onde as coordenadas x, y e z variam de 0 a 255, o identificador

é único. O resultado desta nomeação pode ser observado na Figura 5.1.

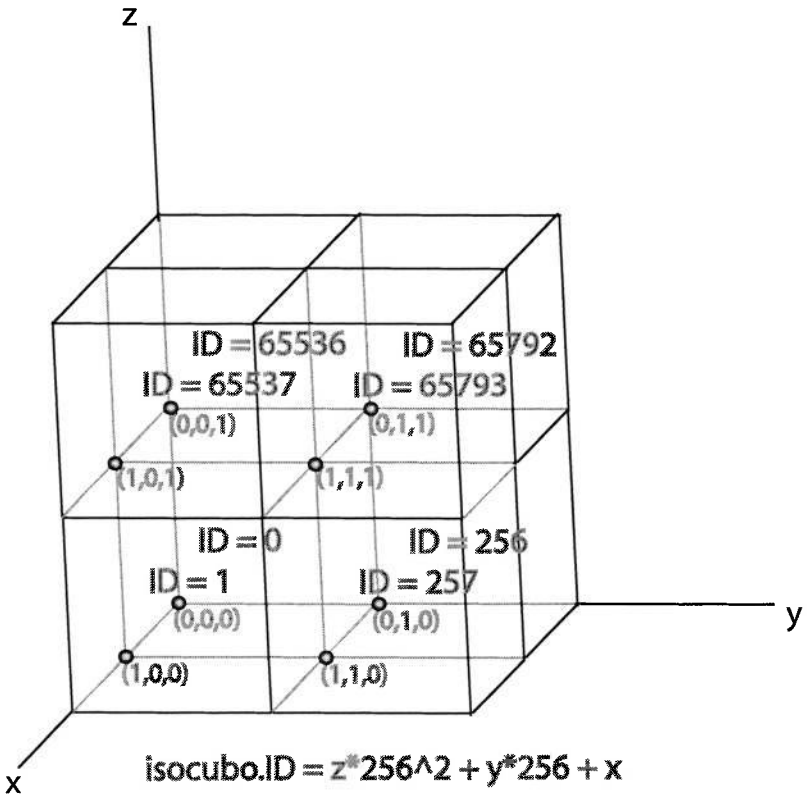


Figura 5.1. Nomeação única para isocubos. Os vértices destacados são os que têm as coordenadas utilizadas na nomeação dos isocubos, assim o isocubo identificado pelo vértice (1,1,1) possui o identificador 65793.

5.2.2 Nomeação para os Isopontos e Arestas

Isocubos vizinhos possuem isopontos comuns, estes isopontos serão os vértices dos triângulos gerados pelo nosso algoritmo. Assim definimos um algoritmo que associa um único nome, à cada isoponto do espaço não importando qual seja o isocubo que está sendo processado. Esse nome ou identificador será utilizado na construção da estrutura B-Rep, pois nela cada vértice possui um identificador único.

Além disso quando mais adiante dizemos que procuramos uma aresta isto é feito por meio de seus vértices (isopontos) que são representados através de seus identificadores, desta forma temos também como identificar uma aresta ou uma *half-edge* de forma inequívoca no espaço.

Para gerar-se este identificador utiliza-se a seguinte equação:

$$isoponto.ID = isoponto.X * 2 + isoponto.Y * 256 + isoponto.Z * 256^2 \quad (5.2)$$

Onde $isoponto.ID$ é o identificador do isoponto e $isoponto.X$, $isoponto.Y$ e $isoponto.Z$ são as coordenadas do isoponto. A coordenada $isoponto.X$ é multiplicada por dois pois o $isoponto.ID$ deve ser um número inteiro e as coordenadas do isoponto sempre possuem valores fracionários onde a parte fracionária sempre vale meio (Ex: 1,5; 120,5...). A Figura 5.2 mostra dois isocubos adjacentes com seus respectivos isopontos (vértices) nomeados de acordo com o algoritmo exposto.

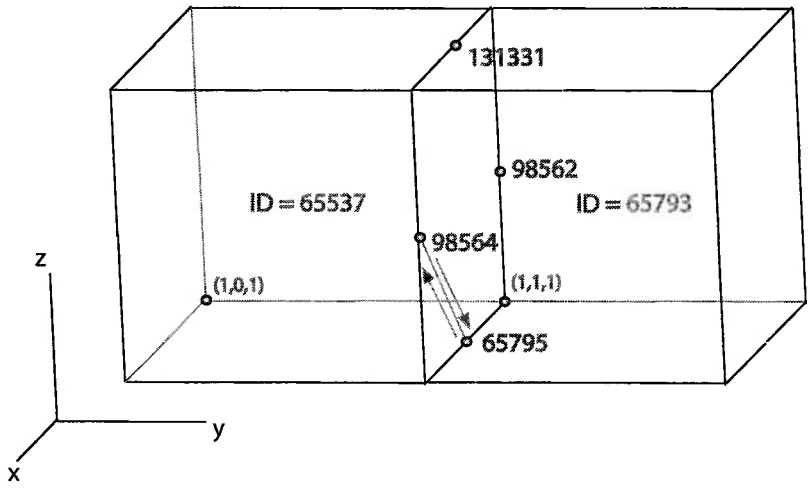


Figura 5.2. Nomeação única para isopontos de dois cubos adjacentes. Os isocubos aqui mostrados são (1,0,1) 65537 e (1,1,1) 65793. Podemos observar os isopontos comuns aos dois isocubos e seus respectivos identificadores. Em detalhe a aresta formada pelos vértices 65795 e 98564 e suas *half-edges* 65795 -> 98564 e 98564 -> 65795.

5.2.3 Nomeação para as Faces

Para nomear-se as faces adotamos uma abordagem diferente, pois não há necessidade de um identificador único uma vez que usamos essa informação apenas localmente. Desta forma nomeamos as faces de acordo com a direção e sentido que ela tem se observada de dentro para fora do isocubo. As 6 faces de um dado isocubo são nomeadas da maneira ilustrada na Figura 5.3.

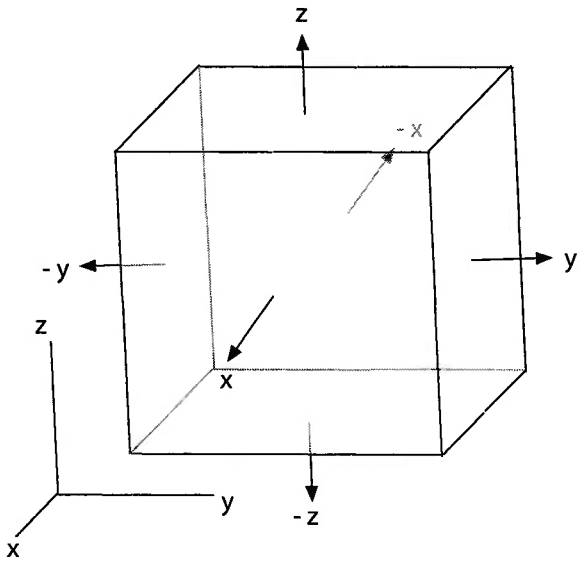


Figura 5.3. Nomeação as 6 faces de um isocubo.

Levando em consideração a nomeação de faces e a nomeação dos isocubos, é possível montar a Tabela 5.1 onde relacionamos a identificação da face e a variação de identificação que somanda à identicação do isocubo atual resulta na identificação do isocubo adjacente a esse tendo como face comum a face indicada.

Tabela 5.1. Relação entre as Faces de um isocubo e a variação de identificação do isocubo.

ID Face	Delta ID Isocubo
x	1
y	256
z	256*256
-x	-1
-y	-256
-z	-256*256

Utilizando essa tabela é fácil notar que se partirmos do isocubo (1,0,1) 65537 e quisermos saber qual isocubo é adjacente a este em relação à face y basta somar 256 a 65537 ou seja teremos o isocubo identificado por 65793 que corresponde ao isocubo (1,1,1) esta é a adjacência que está ilustrada na Figura 5.2.

5.3 Marcha Pelas Faces de Entrada e Saída ou Isofaces

O cerne do algoritmo aqui proposto é a maneira que se marcha de um cubo para outro. Esta marcha é realizada de forma ordenada passa-se de um cubo ao outro através de uma face comum entre eles, desta maneira tira-se o maior proveito possível das informações de adjacência.

Uma face do cubo é considerada face de entrada ou saída, ou seja uma face que leva de um cubo a outro, somente se ela é uma isoface, ou seja é uma face intersecsionada pela fronteira do sólido e nela existe ao menos uma aresta já gerada. Para verificar se uma dada face é uma isoface basta verificar se todos seus vértices têm a mesma classificação em relação ao sólido, se isso for verdade a face não é uma isoface. A Figura 5.4 mostra o cubo processado e suas possíveis faces de saída (x, y e z), a nomeação das faces é feita conforme a explicação apresentada na subseção 5.2.3.

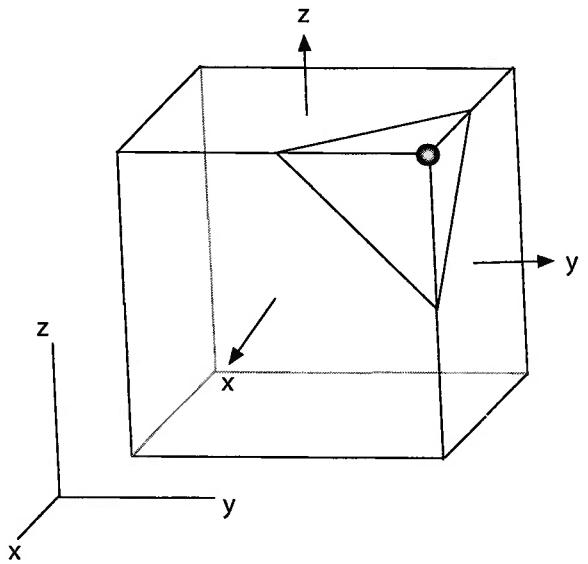


Figura 5.4. Faces de saída (x, y e z) de um cubo já processado.

Assim que o cubo é processado os cubos para onde as faces de saída levam e as faces desses cubos que possuem arestas geradas são empilhados. A informação das faces que possuem arestas geradas é utilizada durante o processamento de cada cubo na parte de geração dos triângulos que será vista mais adiante. Traba-

lhamos com duas pilhas de cubos: a pilha dos cubos que estão sendo processados (*pilha.cubos.atuais*) e a pilha dos que estão conectados às faces de saída dos que estão sendo processados (*pilha.cubos.futuros*). Na Figura 5.5 podemos observar como se daria a marcha se tivéssemos um sólido que englobasse apenas um vértice do cubo. Podemos observar ainda as etapas de construção do sólido. Abaixo descrevemos a marcha:

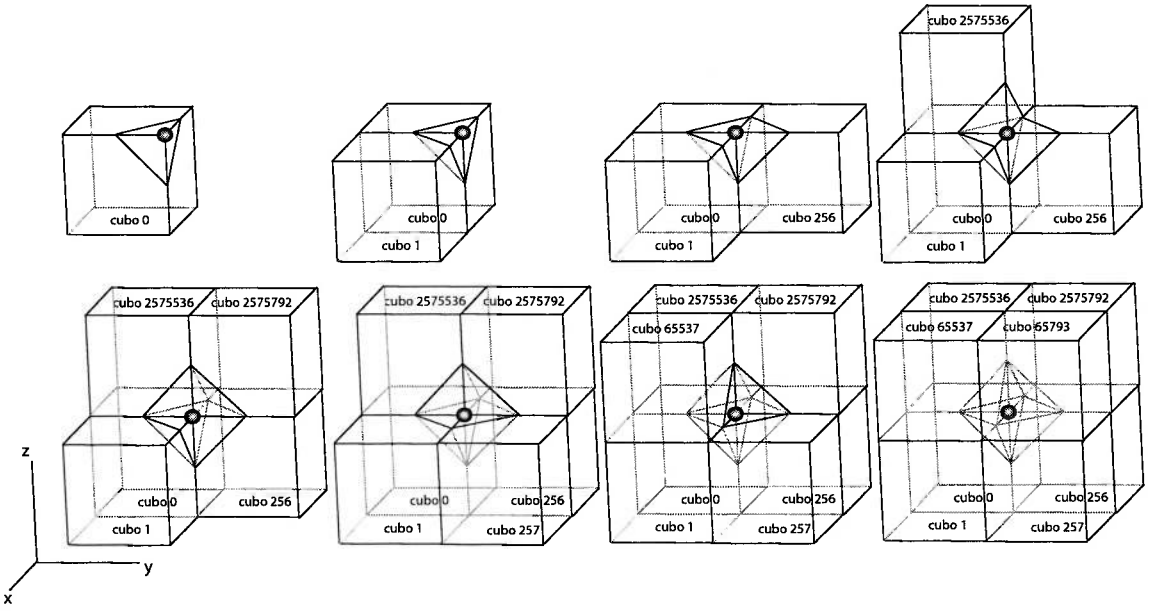


Figura 5.5. Marcha entre os cubos por meio das faces.

- Assim a primeira etapa do algoritmo é encontrar um isocubo qualquer no espaço, denominamos esse cubo de cubo semente para encontrar esse cubo usamos o algoritmo 4 ilustrado no Capítulo 8 Seção 8.1. O cubo semente (*cubo* 0 da figura 5.5) é então empilhado na *pilha.cubos.atuais*, marcado como empilhado e a marcha tem inicio. A *pilha.cubos.atuais* e a *pilha.cubos.futuros* ficam assim:

pilha.cubos.atuais
cubo 0

pilha.cubos.futuros

- Retira-se o primeiro cubo (*cubo* 0, Figura 5.5) da *pilha.cubos.atuais*. Processa-se o *cubo* 0. O *cubo* 0 tem como adjacentes os cubos 1, 256 e 65536, es-

ses cubos e suas faces que já possuem arestas geradas são empilhados na *pilha.cubos.futuros* e marcados como empilhados. As pilhas ficam assim:

pilha.cubos.atuais

pilha.cubos.futuros		
-x, *cubo* 1	-y, *cubo* 256	-z, *cubo* 65536

- Como a *pilha.cubos.atuais* está vazia ela é substituída pela *pilha.cubos.futuros* e a *pilha.cubos.futuros* é eliminada e outra (vazia) criada em seu lugar. As pilhas ficam assim:

pilha.cubos.atuais		
-x, *cubo*1	-y, *cubo* 256	-z, *cubo* 65536

pilha.cubos.futuros

- Retira-se o primeiro cubo (*cubo* 1, Figura 5.5) da *pilha.cubos.atuais*. Processa-se o *cubo* 1. O *cubo* 1 tem como adjacentes os cubos 0 257 e 65537, porém o *cubo* 0 já foi processado, portanto apenas os cubos 257 e 65537 e suas faces que já possuem arestas geradas são empilhados na *pilha.cubos.futuros* e marcados como tal. As pilhas ficam assim:

pilha.cubos.atuais	
-y, *cubo* 256	-z, *cubo* 65536

pilha.cubos.futuros	
-y, *cubo* 257	-z, *cubo* 65537

- Retira-se o primeiro cubo (*cubo* 256, Figura 5.5) da *pilha.cubos.atuais*. Processa-se o *cubo* 256. O *cubo* 256 tem como adjacentes os cubos 0, 65792 e 257, porém o *cubo* 0 já foi processado e o *cubo* 257 está empilhado, portanto as faces que já possuem arestas geradas do cubo 257 precisam ser atualizadas e apenas o cubo 65792 e suas faces que já possuem arestas geradas são empilhados na *pilha.cubos.futuros* e marcado como tal. As pilhas ficam assim:

pilha.cubos.atuais
-z, *cubo* 65536

pilha.cubos.futuros		
-z, *cubo* 65792	-x, -y, *cubo* 257	-z, *cubo* 65537

- Retira-se o primeiro cubo (*cubo* 65536, Figura 5.5) da *pilha.cubos.atuais*. Processa-se o *cubo* 65536. O *cubo* 65536 tem como adjacentes os cubos 0, 65792 e 65537, porém o *cubo* 0 já foi processado e os cubos 65792 e 65537 estão empilhados, portanto as faces que já possuem arestas geradas dos cubos 65792 e 65537 precisam ser atualizadas e nenhum cubo é empilhado. As pilhas ficam assim:

pilha.cubos.atuais

pilha.cubos.futuros		
-y, -z, *cubo* 65792	-x, -y, *cubo* 257	-x, -z, *cubo* 65537

- Como a *pilha.cubos.atuais* está vazia ela é substituída pela *pilha.cubos.futuros* e a *pilha.cubos.futuros* é eliminada e outra (vazia) criada em seu lugar. As pilhas ficam assim:

pilha.cubos.atuais		
-y, -z, *cubo* 65792	-x, -y, *cubo* 257	-x, -z, *cubo* 65537

pilha.cubos.futuros

- Retira-se o primeiro cubo (*cubo* 65792, Figura 5.5) da *pilha.cubos.atuais*. Processa-se o *cubo* 65792. O *cubo* 65792 tem como adjacentes os cubos 65536, 256 e 65793, porém os cubos 65536 e 256 já foram processados, portanto apenas o cubo 65793 e suas faces que já possuem arestas geradas são empilhados na *pilha.cubos.futuros* e marcado como tal. As pilhas ficam assim:

pilha.cubos.atuais	
-x, -y, *cubo* 257	-x, -z, *cubo* 65537

pilha.cubos.futuros
-x, *cubo* 65793

- Retira-se o primeiro cubo (*cubo* 257, Figura 5.5) da *pilha.cubos.atuais*. Processa-se o *cubo* 257. O *cubo* 257 tem como adjacentes os cubos 1, 256 e 65793, porém

os cubos 1 e 256 já foram processados e o cubo 65793 está empilhado, portanto as faces que já possuem arestas geradas do cubo 65793 precisam ser atualizadas e nenhum cubo é empilhado. As pilhas ficam assim:

pilha.cubos.atuais
-x, -z, *cubo* 65537

pilha.cubos.futuros
-z, -x, *cubo* 65793

- Retira-se o primeiro cubo (*cubo* 65537, Figura 5.5) da *pilha.cubos.atuais*. Processa-se o *cubo* 65537. O *cubo* 65537 tem como adjacentes os cubos 1, 65536 e 65793, porém os cubos 65536, 65793 já foram processados e o *cubo* 65793 está empilhado, portanto as faces que já possuem arestas geradas do cubo 65793 precisam ser atualizadas e nenhum cubo é empilhado. As pilhas ficam assim:

pilha.cubos.atuais

pilha.cubos.futuros
-y, -z, -x, *cubo* 65793

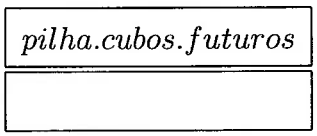
- Como a *pilha.cubos.atuais* está vazia ela é substituída pela *pilha.cubos.futuros* e a *pilha.cubos.futuros* é eliminada e outra (vazia) criada em seu lugar. As pilhas ficam assim:

pilha.cubos.atuais
-y, -z, -x, *cubo* 65793

pilha.cubos.futuros

- Retira-se o primeiro cubo (*cubo* 65793, Figura 5.5) da *pilha.cubos.atuais*. Processa-se o *cubo* 65793. O *cubo* 65793 tem como adjacentes os cubos 65792, 257 e 65537, porém os cubos 65792, 257 e 65537 já foram processados, portanto nenhum cubo é empilhado. As pilhas ficam assim:

pilha.cubos.atuais



- Como a *pilha.cubos.atuais* está vazia ela é substituída pela *pilha.cubos.futuros* e a *pilha.cubos.futuros* é eliminada e outra (vazia) criada em seu lugar. Porém ainda sim a *pilha.cubos.atuais* permanece vazia, portanto a marcha chegou ao fim.

5.3.1 Cubo Intersecionado Mais de Uma Vez Pela Fronteira

Esta situação pode ocorrer quando o cubo for visitado e gerar mais de um polígono, configurações de cubo como ilustradas nos casos 1, 3, 4, 6, 7, 10, 12 e 13 da Figura 4.2 do Capítulo 4 podem levar à esta situação.

Quando situações desse tipo acontecem o cubo é parcialmente processado e será visitado posteriormente para terminar de ser processado, isso ficará claro mais adiante. Entretanto para não haver dúvidas sobre qual aresta foi gerada, armazena-se a aresta na pilha de cubos, como foi detalhado na Seção 5.3, e não a face que possui esta aresta.

Na Figura 5.6, como a *face* 1 possui uma aresta já gerada no processamento do cubo anterior esta face terá suas arestas restantes geradas, por outro lado como a *face* 2 não tem nenhuma aresta já gerada e nenhuma aresta comum com a *face* 1, ela ainda não pode ser gerada. Ou seja o cubo foi parcialmente processado, os cubos adjacentes em cujas faces comuns arestas foram criadas e as arestas geradas são agora empilhados. Além disso o cubo atual, que está sendo processado, deve ser desmarcado como processado assim este cubo poderá ser empilhado para re-processamento, e consequente geração da *face* 2 quando um outro cubo adjacente a ele for processado.

Caso a *face* 2 pertença a um outro shell deste sólido, este cubo não será re-processado, pois não há continuidade entre as fronteiras de shells, e como apenas marchamos pela fronteira do sólido nenhum outro cubo adjacente a este será atingido pela marcha, não possibilitando o re-processamento deste. Isto está previsto no algoritmo e não é um problema, pois uma limitação de nosso algoritmo discutida no Capítulo 7 é o fato de ele só processar um shell por vez.

A mesma lógica é empregada caso o cubo seja interseccionado mais de duas vezes pela fronteira (casos 7, e 13 da Figura 4.2 do Capítulo 4).

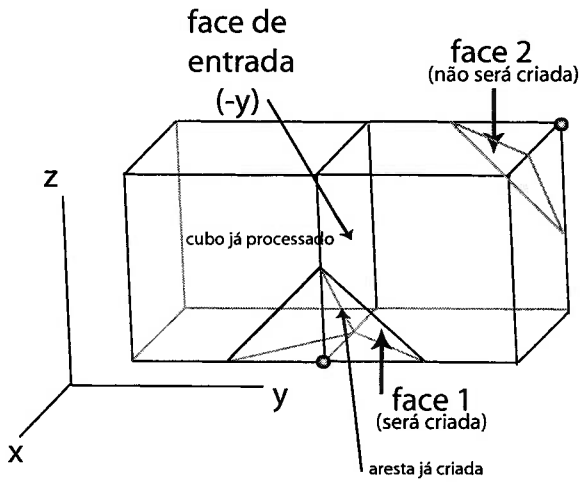


Figura 5.6. Cubo intersecionado mais de uma vez pela fronteira.

Um caso particular desta situação que pode acontecer é quando a face de entrada de um cubo é ambígua, nesse caso o mesmo procedimento é adotado, ou seja apenas as faces que já possuírem arestas geradas anteriormente terão suas arestas restantes geradas, as outras faces serão geradas posteriormente quando o algoritmo retornar a este cubo por meio de outra face de entrada.

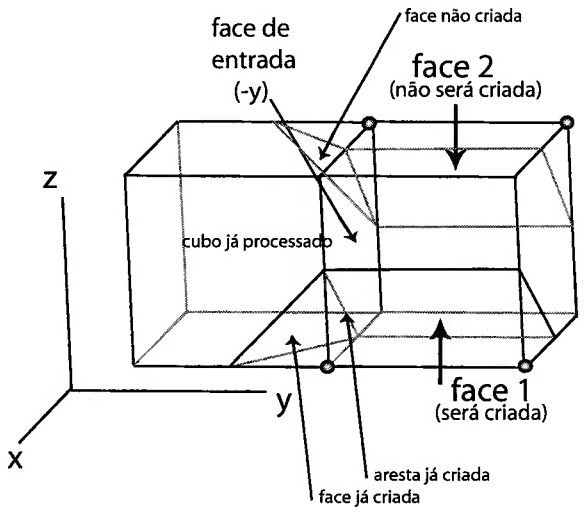


Figura 5.7. Cubo intersecionado mais de uma vez pela fronteira sendo visitado por uma face de entrada ambígua.

Esta situação está ilustrada na Figura 5.7, o cubo indicado como já processado

é um caso como o indicado na Figura 5.6 onde a face de entrada não é ambígua, supondo que no processamento deste cubo somente a face indicada foi gerada ao passarmos para o proximo cubo, usando como entrada uma face ambígua, porém como sabemos qual aresta já foi gerada fica fácil processar esse cubo.

5.4 Geração das Faces Triangulares no Modelo B-Rep

Os operadores de Euler (vide Capítulo 3 Seção 3.3) serão utilizados para gerar as faces triangulares.

O caso mais simples de geração de face triangular ocorre apenas uma vez, quando ainda não existe nada gerado, e isso ocorre logo após o processamento do primeiro cubo.

5.4.1 Geração da Primeira Face Triangular

Para o primeiro vértice deste triângulo utiliza-se o operador <MVSF> (Make Vertex Solid Face). Desta maneira o sólido é criado juntamente com seu primeiro vértice (V1) e sua primeira face, note que esta face só será fechada no final da criação do sólido.

Agora se faz necessário criar o segundo vértice (V2) e a primeira aresta ($\overline{V1V2}$), conectando-se o primeiro vértice ao segundo. Para tanto o operador <MEV> (Make Edge Vertex) é empregado.

Após isso podemos criar o terceiro vértice (V3) e conectá-lo ao segundo (V2) por meio de uma segunda aresta ($\overline{V2V3}$). Para isso empregamos novamente o operador <MEV> (Make Edge Vertex).

Por último precisamos criar a aresta ($\overline{V3V1}$) que conectará o terceiro vértice ao primeiro e finalmente criar a face do triângulo para tanto devemos utilizar <MEF> (Make Edge Face).

De forma resumida temos a seguinte seqüência de operadores:

- **<MVSF>** Make Vertex Solid Face;
- **<MEV>** Make Edge Vertex;

- <**MEV**> Make Edge Vertex;
- <**MEF**> Make Edge Face;

As etapas acima descritas podem ser observadas em detalhes na Figura 5.8.

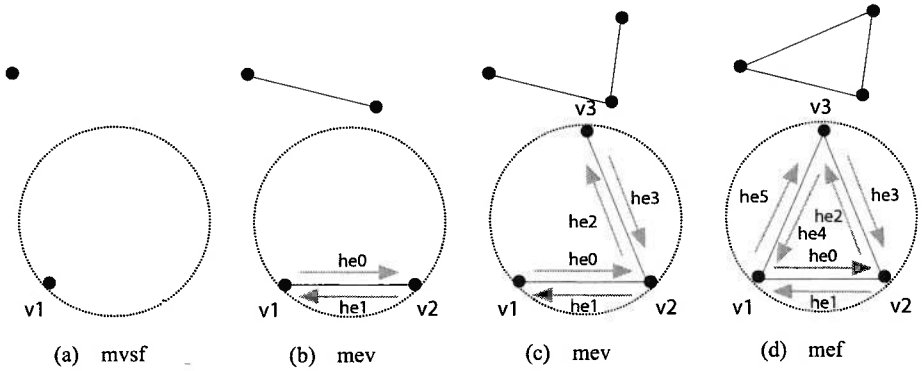


Figura 5.8. Detalhes da criação da primeira face triangular.

5.4.2 Laço de Abertura

Mostramos em detalhes na Figura 5.8 como fica a disposição das *half-edges* (*he*) com a geração da face, o laço formado pelas *half-edges* $he1$, $he5$ e $he3$, é o que chamamos de Laço de Abertura do sólido. É a partir desse laço que o sólido "cresce", ou seja, as *half-edges* contidas nesse laço são usadas na geração das novas faces do sólido. Em alguns casos o sólido pode possuir mais do que um laço de abertura numa etapa intermediária da construção do sólido, porém como essa situação é temporária e com o avanço da marcha estes laços serão unidos, voltando a existir apenas um laço de abertura.

Utilizaremos deste ponto do texto em diante a notação ilustrada na Figura 5.9 quando nos referirmos às *half-edges*. Desta forma quando nos referirmos à $he1->vtx()$ estamos nos referindo ao vértice associado à *half-edge* $he1$ e representado por V1 indicado na figura e quando nos referirmos à $he1->nxt()$ estamos nos referindo à *half-edge* $he2$. Levando em conta a notação acima exposta podemos montar a Tabela 5.2 com alguns exemplos de equivalência entre a notação para *half-edges* e entes geométricos:

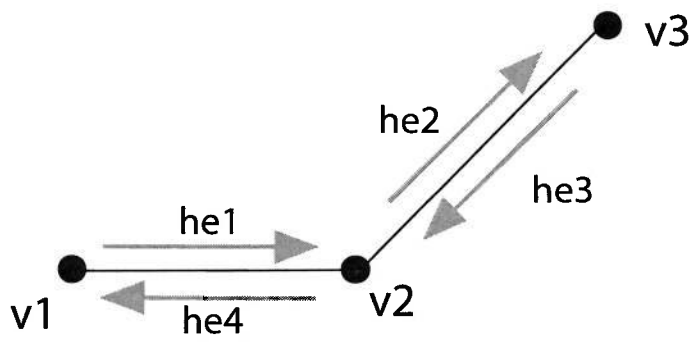


Figura 5.9. Notação empregada no uso das *half-edges*.

Tabela 5.2. Equivalência entre a notação para *half-edges* e os entes geométricos (vide Figura 5.9 para melhor compreensão).

Notação	Ente Geométrico
$he1->vtx()$	V1
$he1->nxt()$	h2
$he2->prv()$	h1
$he1->nxt()->vtx()$	V2
$he2->prv()->vtx()$	V1
$he1->nxt()->nxt()$	h3
$he1->nxt()->nxt()->vtx()$	V3

5.4.3 Geração de Faces Triangulares com Uma Aresta Já Criada

Neste caso a face que temos que criar possui uma aresta em comum com uma face já criada.

Existem duas situações onde isso pode ocorrer: estamos criando um polígono que foi previamente triangularizado por possuir mais do que três vértices (Figura 5.10 (a)), ou estamos gerando uma face que compartilha uma aresta com um cubo vizinho que já foi visitado (Figura 5.10 (b)).

Para os dois casos a maneira que utilizamos para saber que estamos nesta situação é a mesma. A *half-edge he* (Figura 5.10 (a) e (b)) pertence ao laço de abertura e é recuperada por meio da aresta $\overline{V1V2}$, isto é possível para o caso (b) pois estamos supondo que a marcha se deu do *cubo* 1 para o *cubo* 3, e nesse processo a aresta

foi recuperada; no caso (a) temos a aresta pois acabamos de construir a $face$ 1. Em posse de he verificamos se as seguintes expressões são verdadeiras, onde LA é o conjunto de todos os laços de abertura:

$$he- > vtx() = V1 \tag{5.3}$$

$$he- > nxt()- > vtx() = V2 \tag{5.4}$$

$$V3 \notin LA \tag{5.5}$$

Na Figura 5.10 (a) as duas faces pertencem ao mesmo cubo, e na Figura 5.10 (b) as faces pertencem a cubos diferentes, nos dois casos suponha que a $face$ 1 já foi criada e portanto a $face$ 2 já possui uma aresta criada ($\overline{V1V2}$) para tanto o *cubo* 1 já tem que ter sido visitado. Concluímos que a $face$ 2 tem que ser construída conforme descrito acima.

Desta forma já temos dois vértices (V1 e V2) e uma aresta ($\overline{V1V2}$) criados, portanto podemos criar o terceiro vértice (V3) e conectá-lo ao segundo por meio de uma segunda aresta ($\overline{V2V3}$) . Para isso empregamos o operador <MEV> (Make Edge Vertex).

Por último precisamos criar a aresta ($\overline{V3V1}$) que conectará o terceiro vértice ao primeiro e finalmente criar a face do triângulo para tanto devemos utilizar <MEF> (Make Edge Face).

De forma resumida temos a seguinte seqüência de operadores:

- **<MEV>** Make Edge Vertex;
- **<MEF>** Make Edge Face;

As etapas acima descritas podem ser observadas em detalhes na Figura 5.8, onde inicialmente estamos na situação (b) e portanto realizamos as etapas (c) e (d)

5.4.4 Geração de Faces Triangulares com Duas Arestas Já Criadas

A única maneira desta situação ocorrer é quando dois cubos adjacentes ao cubo que está sendo processado já foram visitados e tiveram cada um uma face criada que compartilha uma aresta com o cubo em processamento. Estas arestas pertencem à

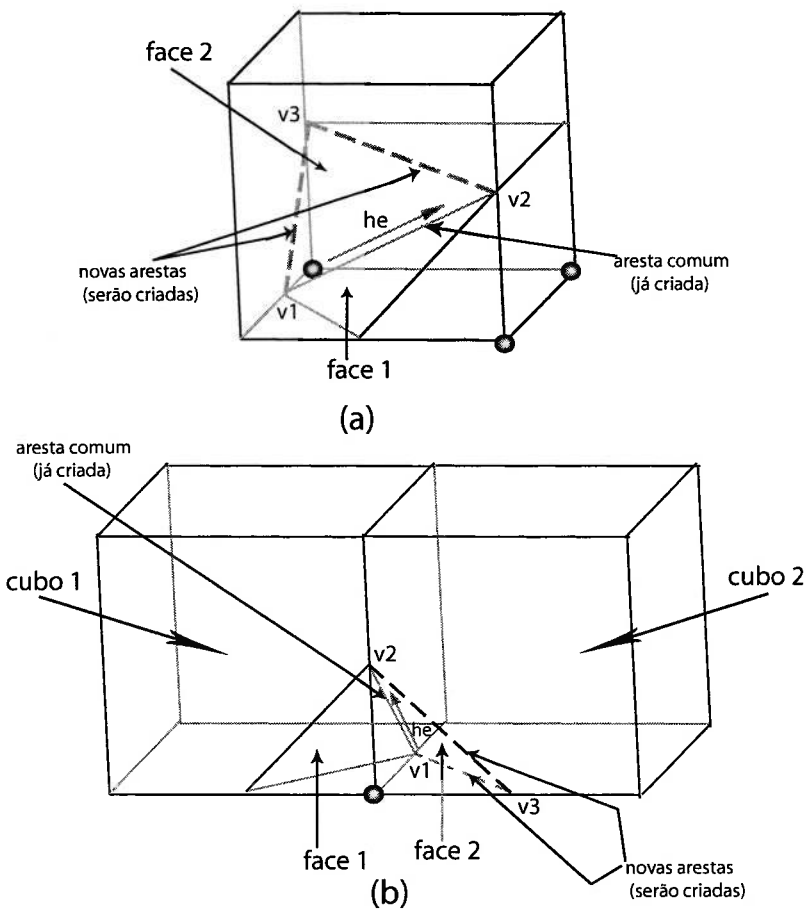


Figura 5.10. Criação de uma face triangular com uma aresta já criada, suponha que a *face* 1 já foi criada e portanto a *face* 2 já possui uma aresta criada, em (a)as duas faces pertencem ao mesmo cubo, e em (b) as faces pertencem a cubos diferentes, tendo o *cubo* 1 um já sido visitado.

face a ser criada, e os três vértices já foram criados, faltado apenas a criação de uma última aresta para fechar a face para tanto devemos utilizar <MEF> (Make Edge Face).

A maneira que utilizamos para saber que estamos nesta situação é a seguinte, a *half-edge he* (Figura 5.11) pertence ao laço de abertura e é recuperada por meio da aresta $\overline{V1V2}$, isto é possível pois estamos supondo que a marcha se deu do *cubo* 1 para o *cubo* 3, e nesse processo a aresta foi recuperada. Em posse de *he* verificamos se as seguintes expressões são verdadeiras:

$$he- > vtx() = V1 \tag{5.6}$$

$$he- > nxt()- > vtx() = V2 \tag{5.7}$$

$$he- > prv()- > vtx() = V3 \tag{5.8}$$

Na Figura 5.11 as três faces pertencem a cubos diferentes, suponha que a *face* 1 e a *face* 2 já foram criadas e portanto a *face* 3 já possui duas arestas criadas ($\overline{V1V2}$ e $\overline{V3V1}$) para tanto o *cubo* 1 e o *cubo* 2 necessariamente já foram visitados.

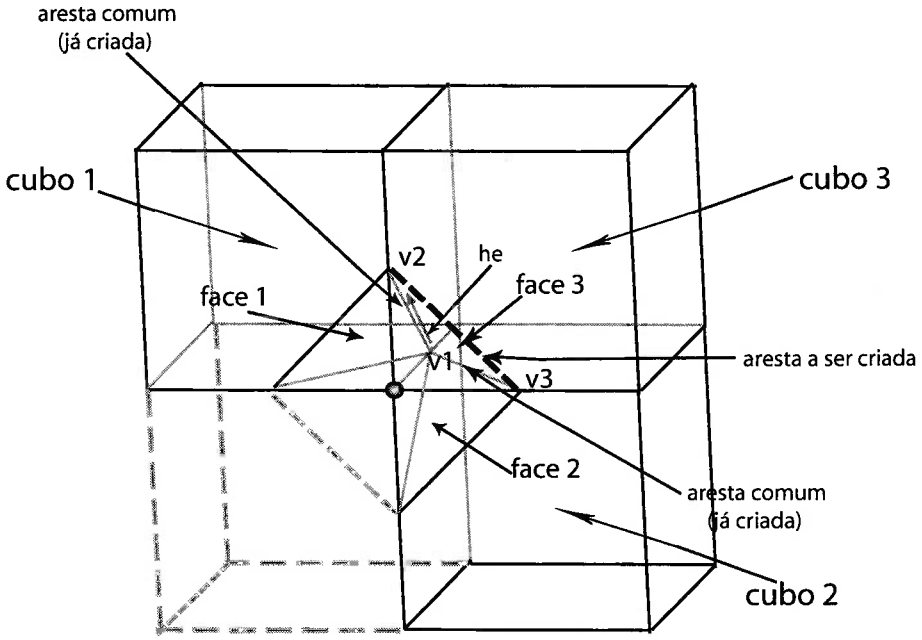


Figura 5.11. Criação de uma face triangular com duas arestas já criadas, suponha que as três faces pertencem a cubos diferentes, suponha que a *face* 1 e a *face* 2 já foram criadas e portanto a *face* 3 já possui duas arestas criadas.

Concluímos que para a *face* 3 ser construída falta somente a criação da última aresta ($\overline{V2V3}$) e da face em si. Abaixo destacamos o operador utilizado:

- <**MEF**> Make Edge Face;

5.4.5 Geração de Faces Triangulares com Três Arestas Já Criadas

Existem situações em que todas as arestas da face são compartilhadas com cubos adjacentes, no caso de todos estes cubos já terem sido visitados e processados, todas as três arestas da face já terão sido criadas. Neste caso nada deve ser feito pois a face já foi indiretamente criada, normalmente isso ocorre quando a última face é adicionada ao sólido. Esta situação é ilustrada na Figura 5.12.

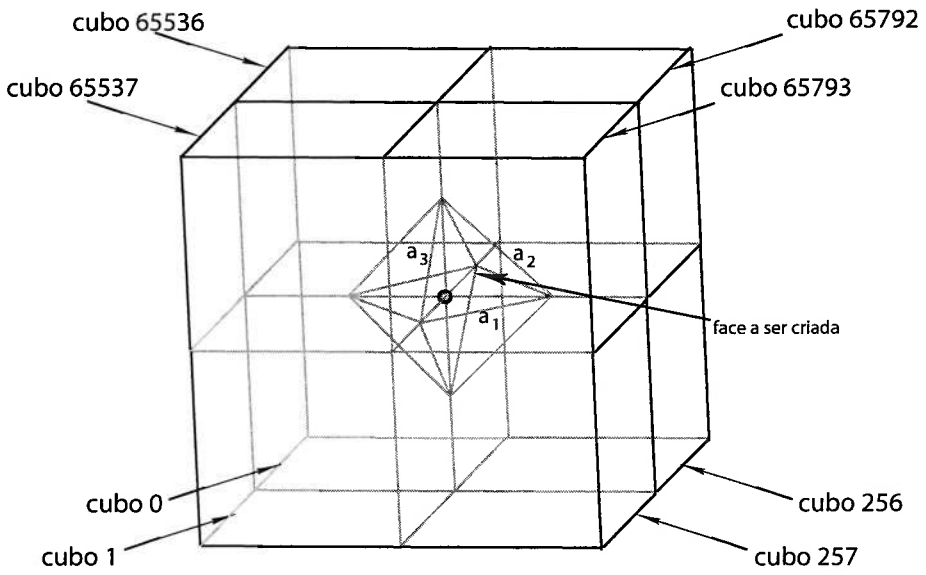


Figura 5.12. Criação de uma face triangular com três arestas já criadas, suponha que as três faces pertencem a cubos diferentes, os cubos 3, 5 e 7 já foram processados e suas respectivas faces criadas e portanto as arestas a_1, a_2 e a_3 também já foram criadas.

A maneira que utilizamos para saber que estamos nesta situação é a seguinte, a *half-edge he* (Figura 5.12) pertence ao laço de abertura e é recuperada por meio da aresta $\overline{V1V2}$, isto é possível pois estamos supondo que a marcha se deu do *cubo* 1 para o *cubo* 3, e nesse processo a aresta foi recuperada. Em posse de *he* verificamos se as seguintes expressões são verdadeiras:

$$he->nxt()->nxt()->nxt() = he \tag{5.9}$$

Ou seja já existe um laço formado pelos vértices V1, V2 e V3, conseqüentemente as arestas e a face já estão criadas.

5.4.6 Geração de Faces Triangulares Três Vértices Já Criados - Caso I

Existe uma segunda situação onde já existem três vértices criados, porém não se verifica a expressão 5.9.

A maneira que utilizamos para saber que estamos nesta situação é a seguinte, a *half-edge he* (Figura 5.13) pertence ao laço de abertura e é recuperada por meio da

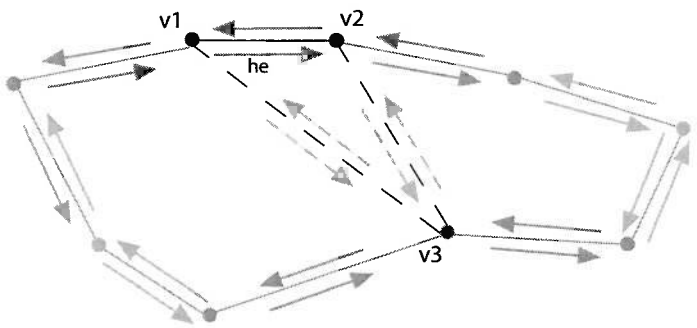


Figura 5.13. Criação de uma face triangular com três vértices já criados Caso I.

aresta $\overline{V1V2}$. Em posse de *he* verificamos se as seguintes expressões são verdadeiras, onde l é o laço de abertura que contém V1 e V2:

$$he- > vtx() = V1 \tag{5.10}$$

$$he- > nxt()- > vtx() = V2 \tag{5.11}$$

$$he- > prv()- > vtx() \neq V3 \tag{5.12}$$

$$V3 \in l \tag{5.13}$$

Ou seja já existem os vértices V1, V2 e V3 de modo que pertençam ao mesmo laço de abertura, porém não existem as arestas $\overline{V2V3}$ e $\overline{V3V1}$.

Desta forma já temos três vértices e uma aresta ($\overline{V1V2}$) criados, portanto podemos conectar o segundo vértice ao terceiro vértice por meio de uma segunda aresta ($\overline{V2V3}$). Para isso empregamos o operador <MEF> (Make Edge Face).

Por último precisamos criar a aresta ($\overline{V3V1}$) que conectará o terceiro vértice ao primeiro e finalmente criar a face do triângulo para tanto devemos utilizar novamente <MEF> (Make Edge Face). Esta opção cria um novo laço de abertura.

De forma resumida temos a seguinte seqüência de operadores:

- **<MEF>** Make Edge Face;
- **<MEF>** Make Edge Face;

5.4.7 Geração de Faces Triangulares Três Vértices Já Criados - Caso II

Existe uma terceira situação onde já existem três aresta criadas, porém também não se verifica a expressão 5.9.

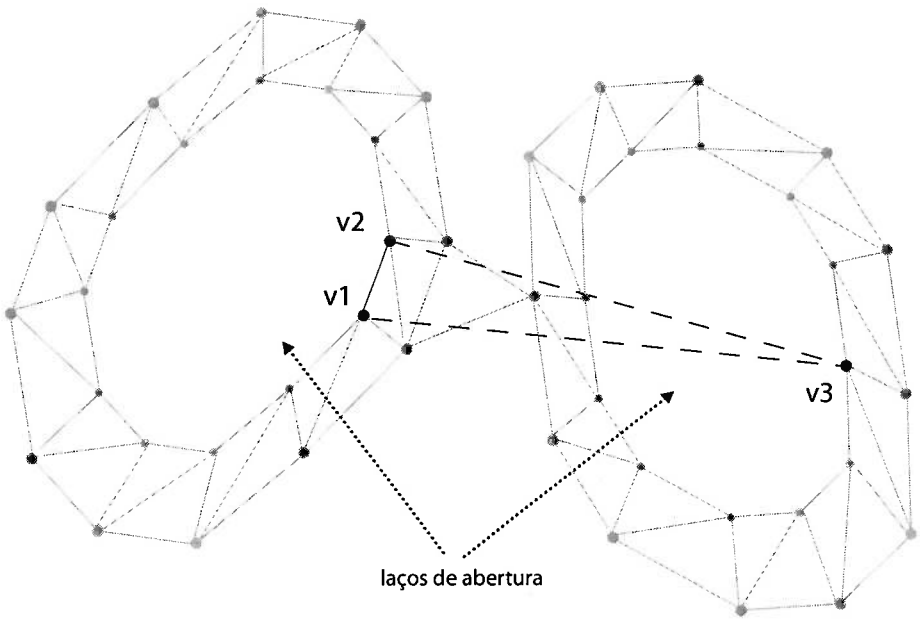


Figura 5.14. Criação de uma face triangular com três vértices já criados Caso II.

A maneira que utilizamos para saber que estamos nesta situação é a seguinte, a *half-edge he* (Figura 5.14) pertence ao laço de abertura e é recuperada por meio da aresta $\overline{V1V2}$. Em posse de *he* verificamos se as seguintes expressões são verdadeiras, onde l é o laço de abertura que contém V1 e V2:

$$he->vtx() = V1 \tag{5.14}$$

$$he->nxt()->vtx() = V2 \tag{5.15}$$

$$he->prv()->vtx() \neq V3 \tag{5.16}$$

$$V3 \notin l \tag{5.17}$$

Ou seja já existem os vértices V1, V2 e V3 de modo que V1 e V2 pertençam ao mesmo laço de abertura, porém V3 pertence a um outro laço de abertura e não existem as arestas $\overline{V2V3}$ e $\overline{V3V1}$.

Desta forma já temos três vértices e uma aresta ($\overline{V1V2}$) criados, porém estamos em uma situação em que o sólido em contrução é topologicamente igual a um toróide ou um tubo, por exemplo, nos quais foram criados mais do que um laço de abertura. Nesse caso em particular os operadores <KFMRH> e <MEKR> precisam ser executados. O segundo operador (<MEKR>) irá criar a segunda aresta ($\overline{V2V3}$).

Por último precisamos criar a aresta ($\overline{V3V1}$) que conectará o terceiro vértice ao primeiro e finalmente criar a face do triângulo para tanto devemos utilizar novamente <MEF> (Make Edge Face). Esta operação transforma dois laços de abertura em um único.

De forma resumida temos a seguinte seqüência de operadores:

- **<KFMRH>** Kill Face Make Ring Hole;
- **<MEKR>** Make Edge Kill Ring;
- **<MEF>** Make Edge Face;

Capítulo 6

Resultados

6.1 Implementação

Na implementação do algoritmo utilizou-se C++ como linguagem de programação, o compilador utilizado foi o Microsoft Visual C++ 6.0, a decisão pelo compilador e linguagem de programação foi pautada pelo fato de a principais bibliotecas utilizadas, o USP Designer e o OpenGL, estarem disponíveis para essa linguagem e compilador.

O USPDesigner, um modelador de sólidos B-Rep (Boundary Representation), utiliza diversos elementos da programação genérica e do STL do C++ com o objetivo de torná-lo uma ferramenta modularizada e reutilizável, na qual um programador pode facilmente incorporar novas funcionalidades. Esta flexibilidade é necessária para que o USPDesigner possa ser utilizado para fins didáticos, pois alunos podem implementar novas features e compreender o funcionamento de um sistema CAD, e para pesquisa aplicada, onde novos algoritmos e estrutura de dados são facilmente incorporados e testados. Dispondo de um ambiente que permite resolver a maioria dos problemas geométricos, o usuário (programador) precisa se preocupar apenas com a lógica relacionada com o seu problema específico.

O sistema criado, que tem o algoritmo em seu kernel, usa como entrada um arquivo de texto com a descrição do sólido CSG, a sintaxe seguinte deve ser utilizada neste arquivo:

Sintaxe Geral do Arquivo :

- **Comentários** Linhas com comentários podem ser inseridas no arquivo, para tanto basta iniciar-se a linha com "//"

  Exemplo: //isto é um comentário

- **Linha de comando** As linhas de comando possuem a seguinte sintaxe:

  <indice> <comando> <parametros>;

- **Fim de Arquivo** O final do arquivo é identificado pela palavra "FIM"

Comandos Os possíveis comandos estão agrupados em comandos relativos às Primitivas, comandos relativos às Transformações Afins e comandos relativos às Operações Booleanas.

Primitivas Os comandos relativos às Primitivas estão listados abaixo:

- **Paralelepípedo**

  sintaxe: <indice> paralelepipedo $v1x$ $v1y$ $v1z$ $v2x$ $v2y$ $v2z$;

  parâmetros: $v1$ e $v2$ são vertices da diagonal principal do paralelepípedo.

  Exemplo: 1 paralelepipedo 0 0 0 2 2 2;

- **Esfera**

  sintaxe: <indice> esfera cx cy cz r;

  parâmetros: cx cy cz são coordenadas do centro da esfera e r o raio.

  Exemplo: 2 esfera 1 0 1 0.7;

- **Cilindro**

  sintaxe: <indice> cilindro r h;

  parâmetros: r é o raio do cilindro e h a altura.

  Exemplo: 12 cilindro 1 2;

- **Cone**

  sintaxe: <indice> cone r h;

  parâmetros: r é o raio do cone e h a altura.

  Exemplo: 110 cone 1 2;

- **Toróide**

  sintaxe: <indice> toroide r rt;

parâmetros: r distância do centro do tubo ao centro do toróide e rt raio do tubo.

Exemplo: 2 toroide 1 0.3;

- **Elipsóide**

  sintaxe: <indice> elipsoide a b c;

  parâmetros: a b c são os semi-eixos do elipsóide.

  Exemplo: 1 elipsoide 1 2 1;

Transformações Afins Os comandos relativos às Transformações Afins estão listados abaixo:

- **Translação**

  sintaxe: <indice> translacao dx dy dz <indice solido>;

  parâmetros: translada de dx dy dz o sólido de índice <indice solido>.

  Exemplo: 18 translacao -1 -1 -1 17; //neste exemplo o sólido 17 é transladado de -1 -1 -1 e o resultado será o sólido 18

- **Rotações**

  sintaxe:

  <indice> rotacaox $tetax$ <indice solido>;

  <indice> rotacaoy $tetay$ <indice solido>;

  <indice> rotacaoz $tetaz$ <indice solido>;

  parâmetros: escala pelos ângulo $teta$ o sólido de índice <indice solido>.

  Exemplo: 19 rotacaox 30 18; //neste exemplo o sólido 18 é rotacionado de 30 graus tendo o eixo x como referência e o resultado será o sólido 19

- **Reflexões**

  sintaxe:

  <indice> reflexaoxy <indice solido>;

  <indice> reflexaoxz <indice solido>;

  <indice> reflexaoyz <indice solido>;

  parâmetros: -

Exemplo: 23 reflexaoxy 22; //neste exemplo o sólido 22 é sofre uma reflexão em relação ao plano xy e o resultado será o sólido 23

Operações Booleanas Os comandos relativos às Operações Booleanas estão listados abaixo:

- **União**

  sintaxe: <indice> uniao <indice solido1> <indice solido2>;

  parâmetros: o sólido <indice solido1> é unido ao sólido <indice solido2>.

  Exemplo: 17 uniao 13 16; //neste exemplo o solido 13 é unido ao solido 16 e o resultado será o sólido 17

- **Intersecção**

  sintaxe: <indice> interseccao <indice solido1> <indice solido2>;

  parâmetros: o sólido <indice solido1> é interseccionado com o sólido <indice solido2>.

  Exemplo: 16 interseccao 14 15; //neste exemplo o solido 14 é interseccionado com solido 15 e o resultado será o sólido 16

- **Subtração**

  sintaxe: <indice> subtracao <indice solido1> <indice solido2>;

  parâmetros: neste caso específico a ordem importa, o sólido <indice solido2> é subtraido do o sólido <indice solido1>.

  Exemplo: 3 subtracao 1 2; //neste exemplo o solido 2 é subtraido do solido 15 e o resultado será o sólido 3

O arquivo é lido e o sólido CSG (Árvore CSG)construído internamente no programa. O algoritmo passa a ser executado obtendo informações do sólido CSG e utilizando a biblioteca para criação do sólido B-Rep e para resgatar informações sobre os laços de abertura, quando necessário. Por fim os triângulos que compõem a fronteira do sólido B-Rep são renderizados por meio da biblioteca OpenGL.

O diagrama da implementação do sistema pode ser observado na Figura 6.1.

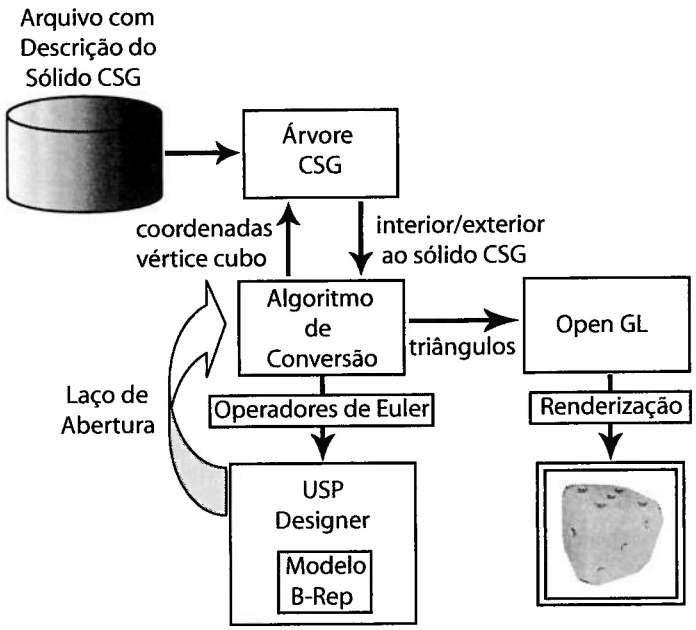


Figura 6.1. Diagrama da implementação do sistema.

6.2 Testes

O algoritmo foi testado para o sólido usado como exemplo no Capítulo 5 Seção 5.3, o resultado pode ser visto na Figura 6.2:

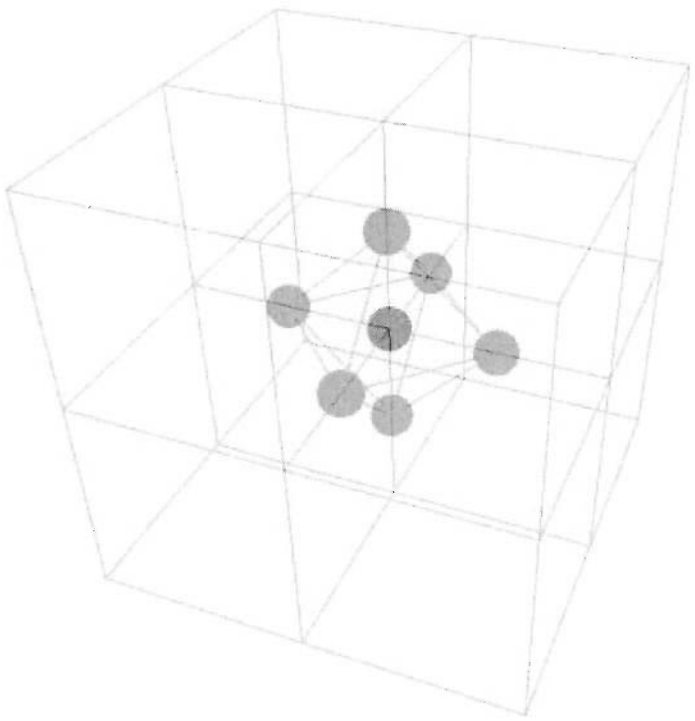


Figura 6.2. O sólido mais simples possível, engloba apenas um vértice do cubo. Foram processados 8 cubos e gerados 8 triângulos.

As etapas da construção do sólido, classificação dos vértices, geração dos isopontos e geração dos polígonos triangularizados, podem ser observadas na Figura 6.3.

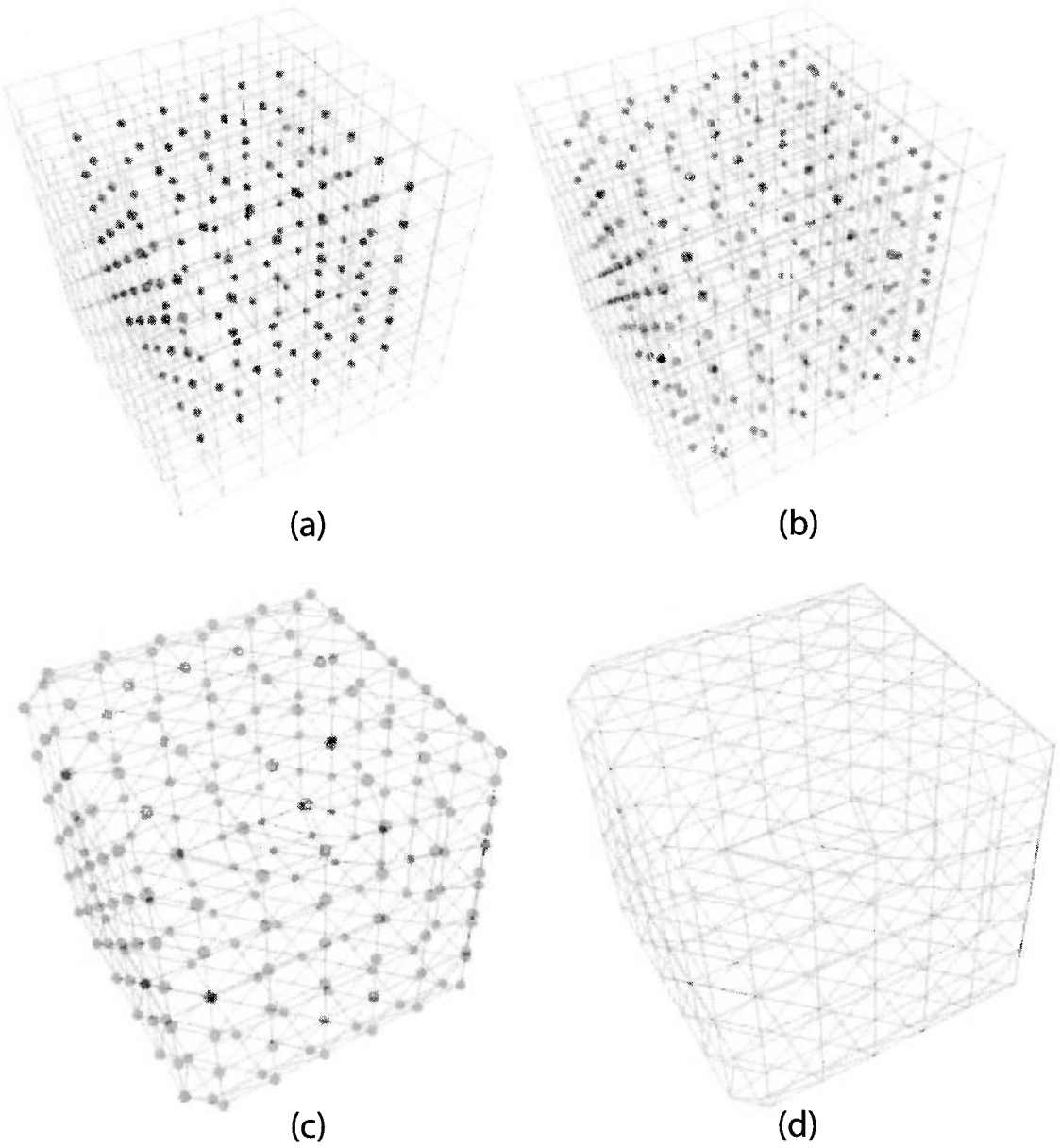


Figura 6.3. Em (a) temos os cubos e os vértices interiores ao sólido, em (b) temos os isopontos nas arestas do cubo, em (c) temos os isopontos conectadas pelas arestas das faces triangulares e em (d) temos as faces do sólido final.

Durante a criação do sólido, mais de um laço de abertura pode ser criado para depois serem unidos, na Figura 6.4 podemos observar esse processo.

O algoritmo foi testado para cada uma das primitivas implementadas, o resultado é mostrado na Figura 6.5.

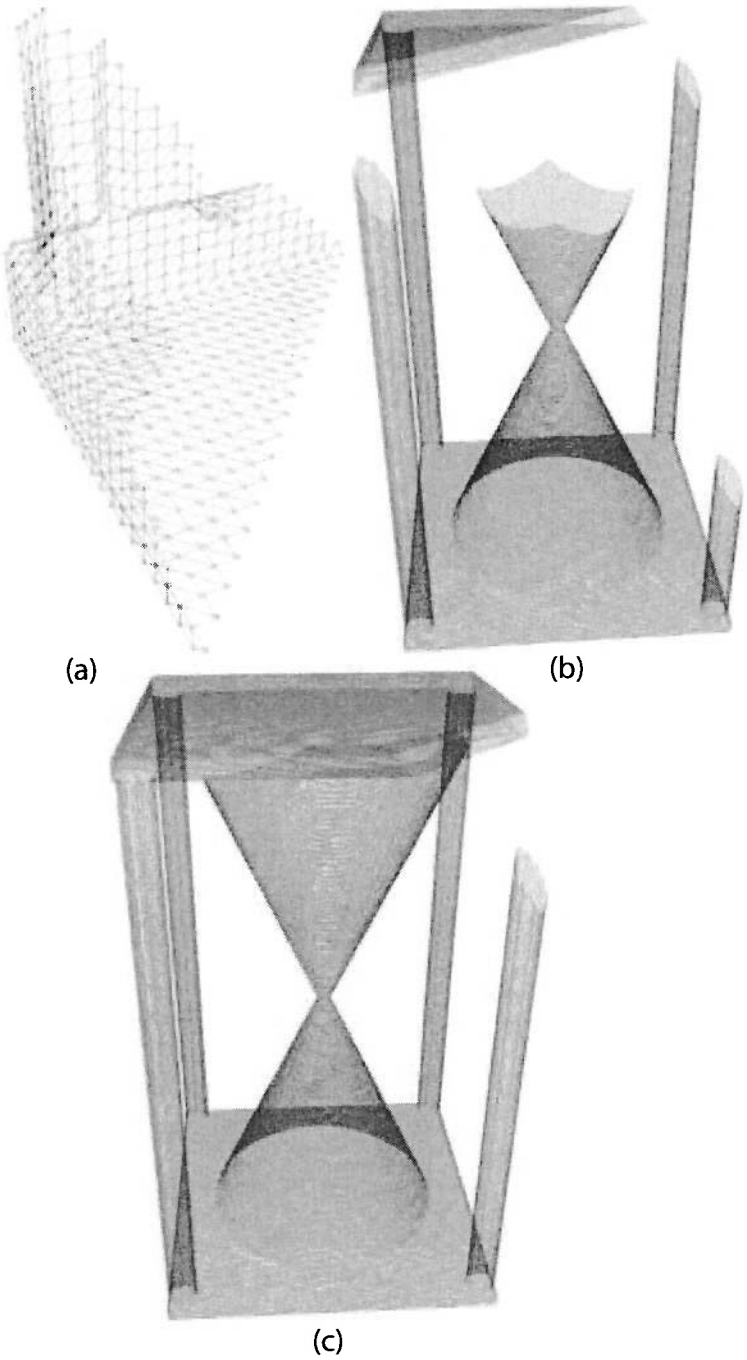


Figura 6.4. Em (a) ampliamos a imagem do início da construção do sólido com apenas 1 laço de abertura criado, em (b) temos uma etapa intermediária da construção do sólido onde 5 laços de abertura foram criados e em (c) temos uma etapa mais avançada da criação do sólido onde parte dos laços de abertura que tinham sido criados foram unidos restando apenas 2 laços de abertura.

As operações booleanas entre um cubo e uma esfera podem ser observadas na Figura 6.6.

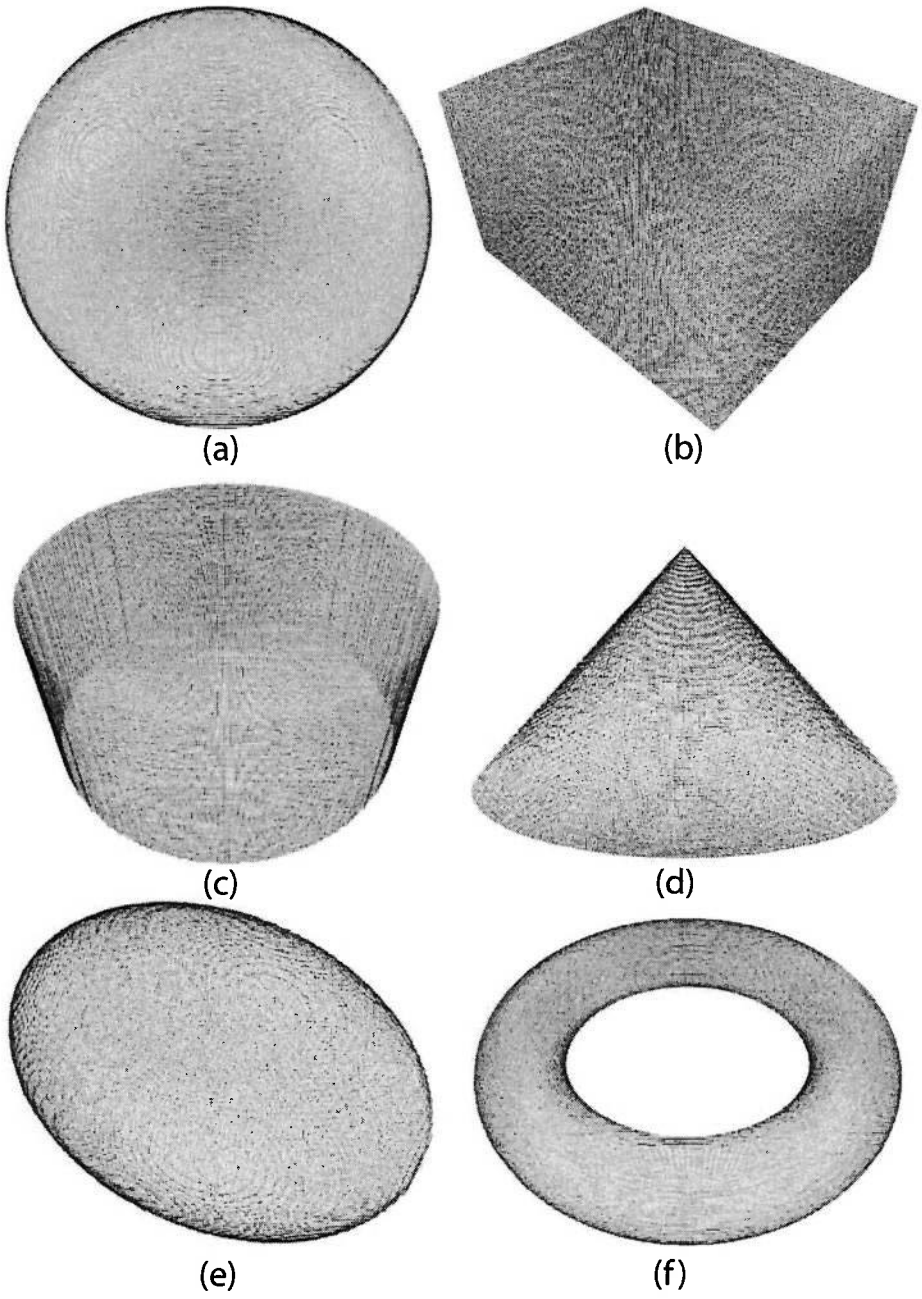


Figura 6.5. Primitivas implementadas: em (a) esfera (120488 cubos processados e 240968 triângulos gerados), (b) paralelepípedo (37602 cubos processados e 75196 triângulos gerados), (c) cilindro (92328 cubos processados e 184648 triângulos gerados), (d) cone (22828 cubos processados e 45648 triângulos gerados), (e) elipsóide (40168 cubos processados e 80328 triângulos gerados) e (f) toróide (90296 cubos processados e 180584 triângulos gerados).

Sólidos mais complexos também foram usados nos testes como os ilustrados nas Figuras 6.7, 6.8, 6.9, 6.10, 6.11 e 6.12.

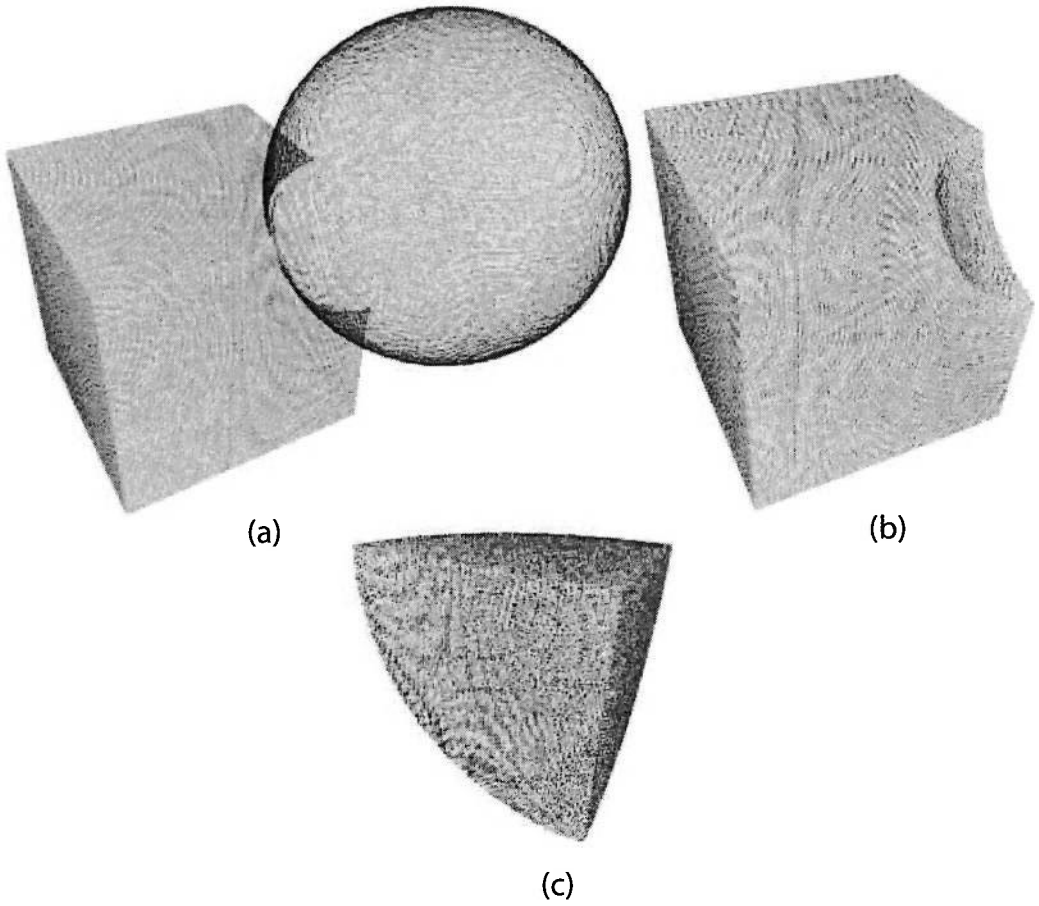


Figura 6.6. Operações booleanas entre um cubo e uma esfera: em (a) união (71849 cubos processados e 143698 triângulos gerados), (b) subtração (38402 cubos processados e 76796 triângulos gerados), (c) intersecção (30596 cubos processados e 61184 triângulos gerados).

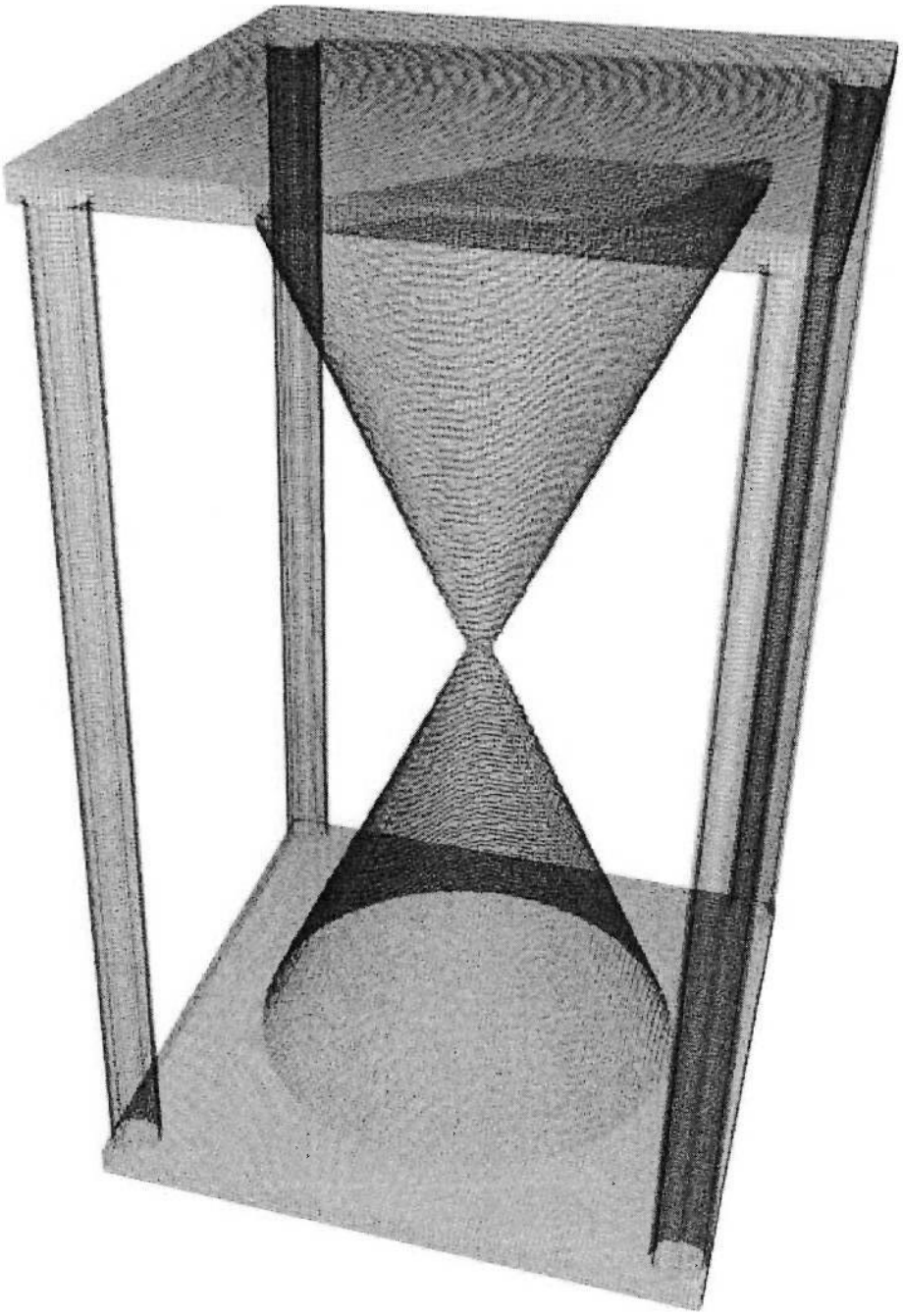


Figura 6.7. Ampulheta formada por paralelepípedos, cilindros e cones. Foram processados 146480 cubos e gerados 292976 triângulos.

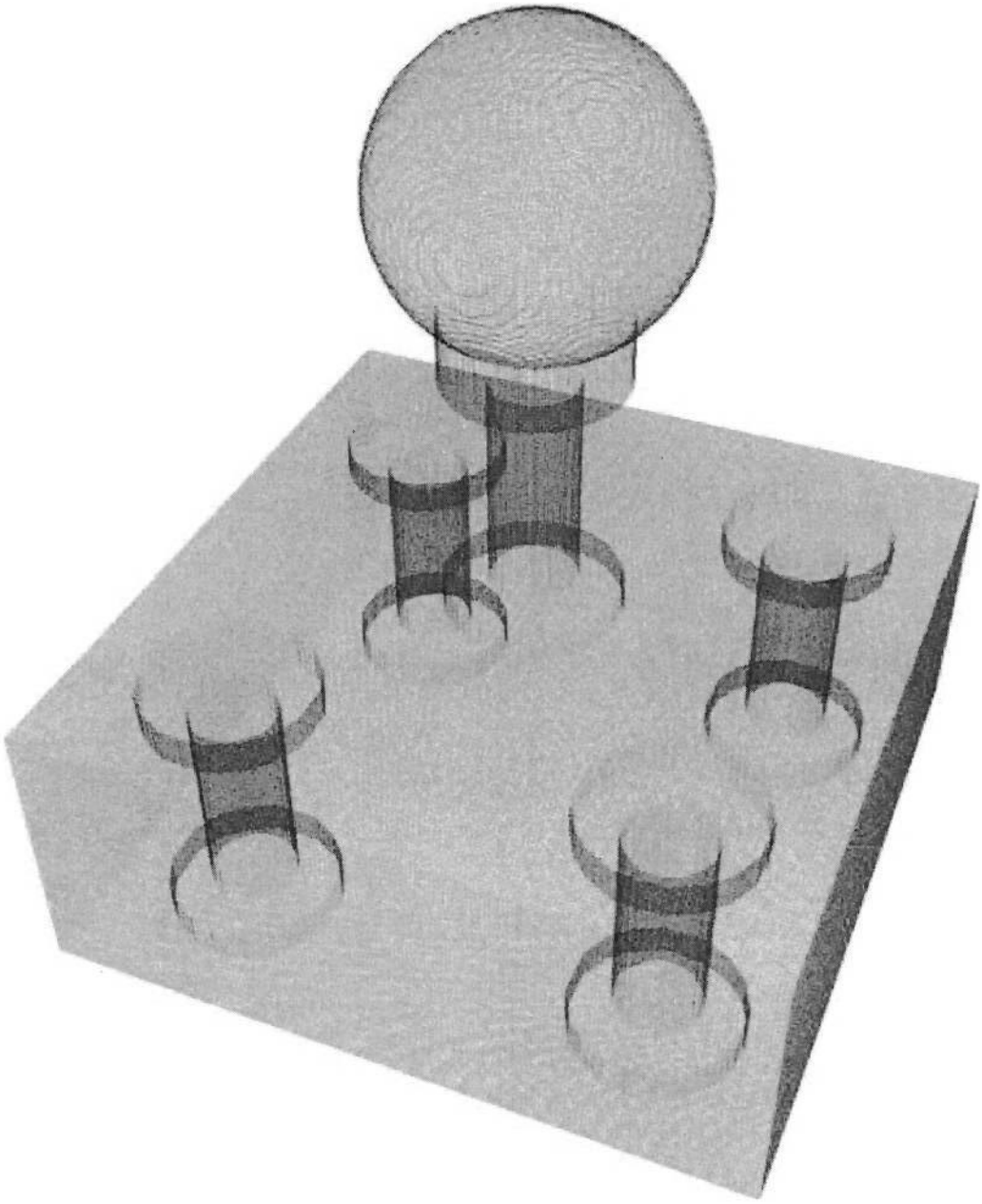


Figura 6.8. Peça formada por um paralelepípedo, alguns cilindros e uma esfera. Foram processados 168674 cubos e gerados 337372 triângulos.

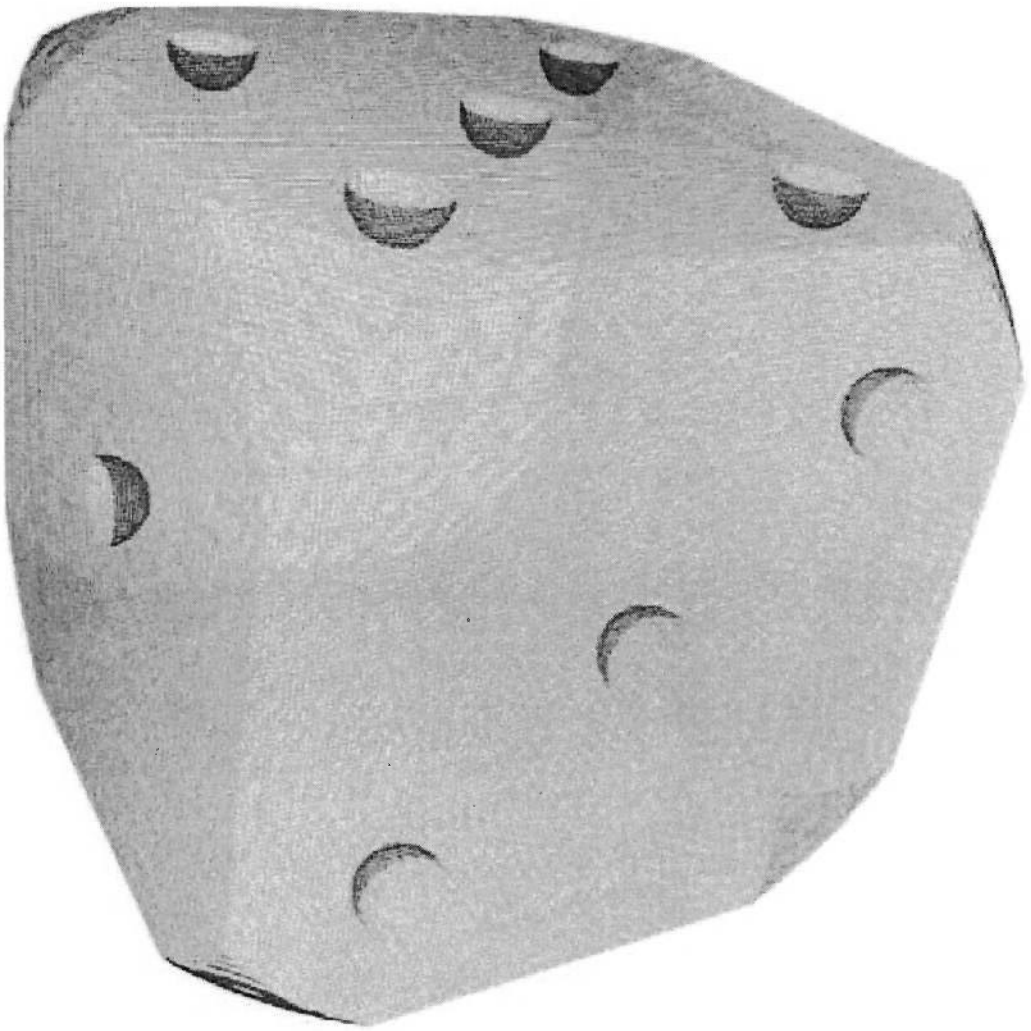


Figura 6.9. Dado formado apenas por um paralelepípedo e algumas esferas. Foram processados 123140 cubos e gerados 246272 triângulos.

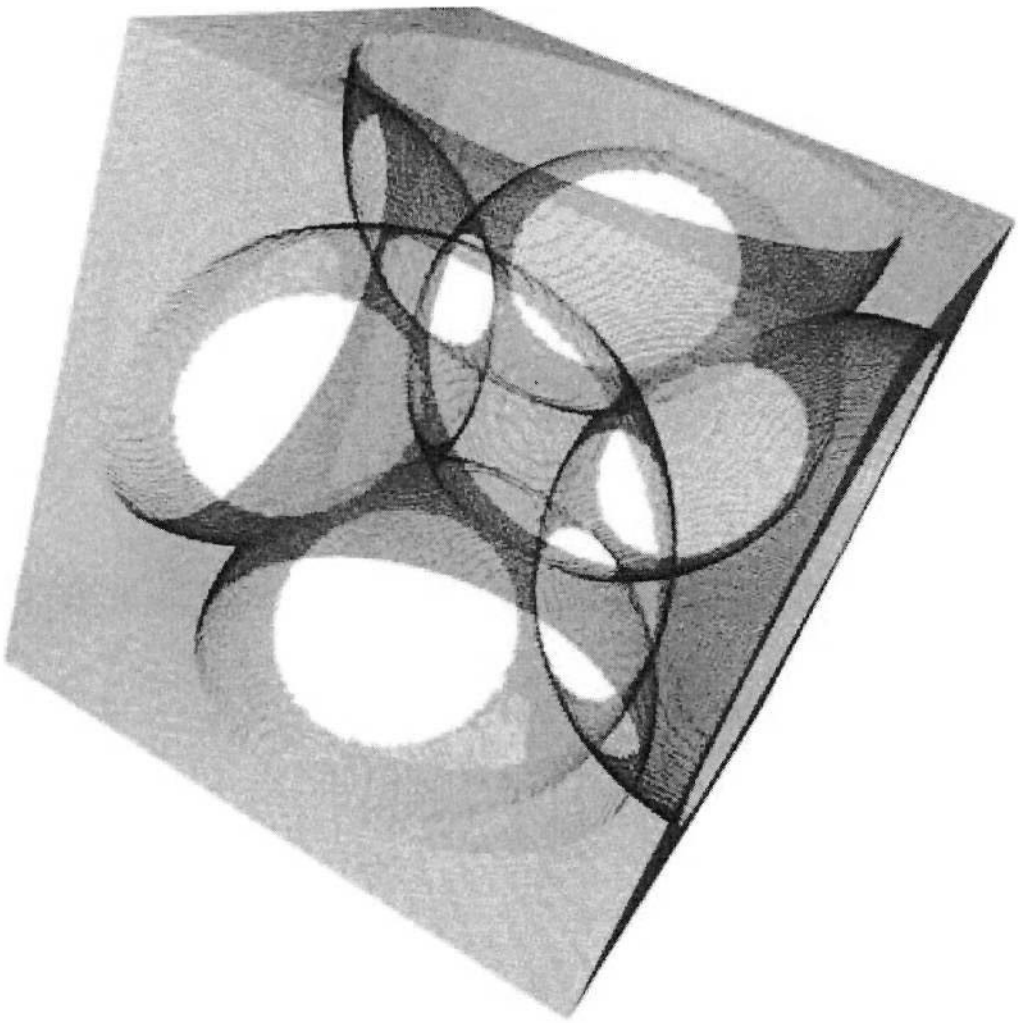


Figura 6.10. Sólido topologicamente complexo obtido pela subtração de 6 esferas de um paralelepípedo. Foram processados 175952 cubos e gerados 356648 triângulos.

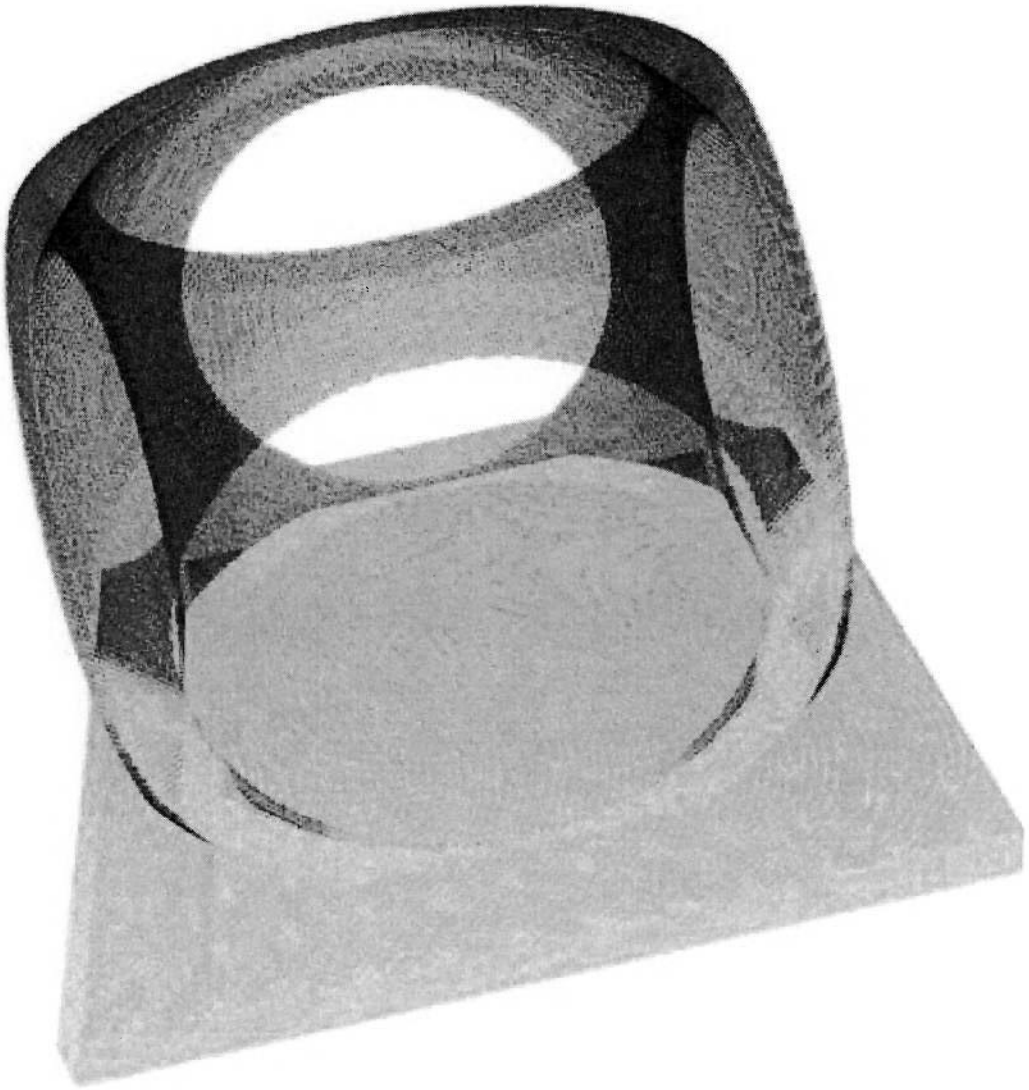


Figura 6.11. Peça formada apenas por paralelepípedos e esferas. Foram processados 277795 cubos e gerados 553632 triângulos.

Figura 6.12. Outro sólido complexo formado por paralelepípedos, esferas e cilindros. Foram processados 45163 cubos e gerados 90919 triângulos.

Capítulo 7

Conclusões

Um novo algoritmo para converter sólidos CSG em sólidos B-Rep foi proposto e implementado. Esse novo algoritmo é uma combinação de voxelização CSG e marching cubes.

Uma vantagem deste algoritmo em relação ao marching cubes é o fato de marchar-se apenas na fronteira do sólido, esta alteração reduziu a complexidade do algoritmo marching cubes de polinomial de primeira ordem para uma complexidade constante.

Outro alteração é a utilização de arestas geradas em um dado cubo no cubo adjacente, desta forma constrói-se um sólido válido pois as possíveis ambigüidades encontradas em uma face são tratadas de maneira igual em cubos adjacentes evitando problemas como furos.

Uma possível melhoria a ser implementada no futuro é a alteração da entrada de dados para que possam ser usados dados volumétricos obtidos pela digitalização de objetos reais através de métodos como Tomografia Computadorizada, Ressonância Magnética, imagens de microscópios ou por meio de amostragem de funções implícitas ou procedurais.

Capítulo 8

Anexos

8.1 O Algoritmo

Abaixo apresentamos, de forma resumida, os algoritmos criados:

Algoritmo 4 *Cubo *AchaSemente(void)*

```
1: /* Nesse laço percorremos do espaço até encontrarmos um cubo que seja um
   isocubo */
2: para x = 0 até x = 255 faça
3:   para y = 0 até y = 255 faça
4:     para z = 0 até z = 255 faça
5:       se cubo(x, y, z) é isocubo então
6:         retorna cubo
7:       fim se
8:     fim para
9:   fim para
10: fim para
11: /* Nenhum isocubo foi encontrado*/
12: retorna nulo
```

Algoritmo 5 *pilha.arestas *GeraTriangulosModeloSolido*(cubo, *pilha.arestas.atuais)

Aplica-se Delibasis, tendo desta maneira seqüencias de vértices (vide Seção 4.2)
- gera-se todos os isopontos (vértices), ainda não gerados
- gera-se a lista de vértices e de arestas de cada polígono, usando também a informação contida na lista de faces do *cubo* que já possuem arestas criadas

aplica-se triangularização nesses polígonos
para cada polígono da lista de polígonos gerados para o *cubo* dado **faça**
  para cada aresta de cada triângulo da lista de arestas do polígono **faça**
    /* Nesse passo gera-se as faces triangulares de acordo com a quantidade de arestas dessa face que já foram geradas (vide Seção 5.4) */
    se a aresta pertence à face do cubo **então**
      se a aresta pertence à *pilha.arestas.atuais* **então**
        retira-se da pilha a aresta
        toma-se o ponteiro para a $halfedge$
        usa-se os laços de abertura e os Operadores de Euler (vide Capítulo 5)
      se não
        usa-se os Operadores de Euler MVSF para criar-se o sólido
        empilha-se a aresta e o ponteiro para a $halfedge$ criada na *pilha.arestas.atuais*
      fim se
    fim se
  fim para
  se (foi gerada ao menos uma face) E (ainda existem polígonos) **então**
    desmarca-se a posição do *cubo* no *array.bitmaps* permitindo nova visita
  fim se
fim para
retorna *pilha.arestas.atuais*

Algoritmo 6 *Algoritmo Principal*

1: /* As pilhas de cubos além de armazenar os IDs dos cubos armazena também, junto a cada cubo, a lista de faces deste que possuem arestas criadas */
2: cria-se a *pilha.cubos.atuais*
3: cria-se a *pilha.cubos.futuros*
4: /* A pilha.arestas.atuais conterá as arestas e ponteiros para *halfedge* que serão usados na criação do modelo B-Rep */
5: cria-se a *pilha.arestas.atuais*
6: /* O *array.bitmaps* contem os cubos que não devem mais ser empilhados*/
7: cria-se o *array.bitmaps*
8: *novocubo* = *AchaSemente*()
9: empilha-se o *novocubo* na *pilha.cubos.atuais*
10: marca-se a posição do *novocubo* no *array.bitmaps*
11: **enquanto** (*pilha.cubos.atuais* não está vazia) **faça**
12: desempilha-se o *novocubo* na *pilha.cubos.atuais*
13: /* Processa o *novocubo* */
14: *pilha.arestas.atuais* = *GeraTriangulosModeloSolido*(*novocubo*)
15: /* Procura-se cubos para os quais as faces do *novocubo* levam */
16: **para** todas *faces* do *novocubo* **faça**
17: **se** *face* é *isoface* **então**
18: **se** o *cubo* para qual a face leva não está no *array.bitmaps* **então**
19: empilha-se o *cubo* para qual a face leva na *pilha.cubos.futuros* empilha-se junto a lista de faces desse cubo que possuem arestas criadas
20: marca-se a posição do *cubo* no *array.bitmaps*
21: **fim se**
22: **fim se**
23: **fim para**
24: /* O conteúdo das pilhas deve ser atualizado */
25: *pilha.cubos.futuros* = *pilha.cubos.atuais*
26: apaga-se a *pilha.cubos.futuros*
27: cria-se uma nova *pilha.cubos.futuros*
28: **fim enquanto**

Referências Bibliográficas

[1] DELIBASIS, K.; MATSOPOULOS, G.; MOURAVLIANSKY, N.; NIKITA, K., *A Novel and Eficient Implementation of The Marching Cubes Algorithm*, **Computerized Medical Imaging and Graphics**, v. 25, p. 343–352, Julho 2001.

[2] HOFFMANN, C. M., **Geometric and Solid Modeling: An Introduction**. Morgan Kaufmann Publishers, 1989.

[3] FANG, S.; LIAO, D., *Fast CSG voxelization by Frame Buffer Pixel Mapping*, em **Proceedings of the 2000 IEEE symposium on Volume visualization**, (Salt Lake City, Utah, United States), p. 43–48, 2000.

[4] THIBAULT, W. C.; NAYLOR, B. F., *Set Operations on Polyhedra Using Binary Space Partitioning Trees*, **SIGGRAPH'87**, v. 21, p. 153–162, Julho 1987.

[5] REQUICHA, A. A. G., *Representation for Rigid Solids: Theory, Methods and Systems*, **Computing Surveys**, v. 12, p. 437–464, Dezembro 1980.

[6] MÄNTYLÄ, M., **An Introduction to Solid Modeling**. New York, NY, USA: Computer Science Press, 1988.

[7] SéGONNE, F., **Segmentation of Medical Images under Topological Constraints**. PhD thesis, Massachusetts, Dezembro 2005, Department of Electrical Engineering and Computer Science, Massachusetts Institute Of Technology.

[8] BAUMGART, B. G., *A Polyhedron Representation for Computer Vision*, em **AFIPS Conf.** (AFIP PRESS, A., ed.), v. 44, (Va), p. 589–596, 1975.

[9] TORIYA, H.; CHIYOKURA, H., **3D CAD Principles and Applications.** Berlin: Springer-Verlag, 1991.

[10] MERCIER, B.; MENEVEAUX, D., *Shape from Silhouette: Image Pixels for Marching Cubes*, **WSCG**, v. 13, p. 112–118, Janeiro-Fevereiro 2005.

[11] LORENSEN, W. E.; E.CLINE, H., *Marching Cubes: A High Resolution 3-D Surface Construction Algorithm*, **Computer Graphics**, v. 21, p. 163–169, Julho 1987.

[12] BAKER, H., *Building Surfaces of Evolution: The Weaving Wall*, **International Journal of Computer Vision**, v. 3, p. 51–71, Maio 1989.

[13] VARADHAN, G.; KRISHNAN, S.; SRIRAM, T.; MANOCHA, D., *Topology Preserving Surface Extraction Using Adaptive Subdivision*, em **Proceedings of the 2004 Eurographics**, (Nice, France), p. 241–250, 2004.

[14] NOORUDDIN, F.; TURK, G., *Simplification and Repair of Polygonal Models Using Volumetric Techniques*, **IEEE Transactions on Visualization and Computer Graphics**, v. 9, p. 191–205, Abril-Junho 2003.