MARCELO SHIMADA

Aritmética Intervalar aplicada em um Modelador de Sólidos B-rep

Dissertação apresentada à Escola Politécnica
da Universidade de São Paulo para obtenção
do título de Mestre em Engenharia

São Paulo
2002

MARCELO SHIMADA

Aritmética Intervalar aplicada em um Modelador de 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 de Sistemas Mecatrônicos

Orientador:
Prof. Doutor
Marcos de Sales Guerra Tsuzuki

São Paulo
2002

Aos meus pais.

Agradecimentos

Agradeço ao Prof. Marcos de Sales Guerra Tsuzuki pelo apoio e orientação.

Agradeço a Fapesp pelo suporte financeiro.

Agradeço a Thiago Martins pelas discussões sobre metodologia de classificação geométrica.

Finalmente, agradeço aos meus pais e aos meus amigos que me ajudaram durante este trabalho.

RESUMO

Neste trabalho é considerada a utilização de aritmética intervalar para aumentar a robustez dos algoritmos de classificação geométrica utilizados na implementação das operações de corte e booleanas em sistemas de modelagem de sólidos. Os algoritmos de classificação geométrica, também conhecidos como testes de incidência, são importantes para manter a consistência entre a topologia e geometria do sólido quando forem executadas as operações de corte e booleanas. Uma falha em um teste de incidência, na qual são realizadas comparações entre valores, pode prejudicar as etapas posteriores do algoritmo das operações de corte e booleanas e conseqüentemente tornar o sólido inconsistente. A aritmética intervalar incorpora os erros de aproximação, eliminando a necessidade de definir uma tolerância fixa para realizar a comparação entre números de ponto flutuante. Entretanto, não é possível converter diretamente os algoritmos que se baseiam em ponto flutuante para algoritmos implementados em aritmética intervalar, sendo necessária uma total reformulação dos algoritmos. Outro item importante é que na determinação dos pontos de intersecção feita em uma etapa da implementação das operações de corte e booleanas, a utilização da arimética intervalar pode em resultar valores com intervalos com dimensões exageradas, o que pode provocar falhas nos algoritmos de testes e incidência. Para conter esta falha, uma correção baseada na geometria é aplicada. São apresentados os conceitos básicos da arimética intervalar, as representações de elementos geométricos utilizando aritmética intervalar, os testes de incidência, conceitos de um modelador de sólidos B-Rep e os algoritmos que implementam as operações de corte e booleanas.

ABSTRACT

In this work, the use of interval arithmetic is considered to increase robustness of geometric classification algorithms in operations of solid modelling systems. The classification algorithms, also known as incidence tests, are important to keep the consistency between topology and solid geometry during the application of cut solid and boolean operations. A incidence test error, where values are compared, can damage the next steps of the cut solid and boolean operations algorithm and then make the solid inconsistent. The interval arithmetic incorporates aproximation errors, so that, eliminates the need of defining a fixed tolerance to do the comparation between floating point numbers. However, it is not possible to directly convert the algorithms using floating point to algorithms using interval arithmetic, so that, there is a need of total reformulation of the algorithms. Another important item is the determination of intersection points that is done in cut solid and boolean operations, the use of interval arithmetic can result values with intervals with large dimensions, and this can cause fails in the algorithm of incidence tests. To deal with this fail, a correction based on the geometry is applied. So, this work will show the basic concepts of the interval arithmetic, the representation of geometric elements using interval arithmetic, the incidence tests, concepts of a B-Rep Solid Modeller and the algorithms for cut solid and boolean operations.

ERRATA

PÁGINA	LINHA	ONDE SE LÊ	LEIA-SE
ix	29	P2 . P1 e P3 . P1	P2 - P1 e P3 - P1
7	11	$f(x) = 2x^3 . 3x^2 + 1$	$f(x) = 2x^3 - 3x^2 + 1$
7	13	$f(x) = x^2(2x . 3)+1$	$f(x)=x^2 (2x - 3)+1$
10	6	v = P1P2 = P2 . P1	v = P1P2 = P2 - P1
10	12	V = ([x3.x2, x4.x1], [y3.y2, y4.y1], [z3.z2, z4.z1])	V = ([x3-x2, x4-x1], [y3-y2, y4-y1], [z3-z2, z4-z1])
13	7	(1.t) * P_1+t * P_2=P(t)	(1-t)* P_1+t *P_2=P(t)
13	10	t * (P_2 . P_1) = P(t). P_1	t*(P_2 - P_1) = P(t)- P_1
13	12	t = (P_3 . P_1)/(P_2 . P_1)	t =(P_3 - P_1)/(P_2 - P_1)
13	15	(P_2 . P_1)	(P_2 - P_1)
13	17	t =[(P_2.P_1)•(P_3.P_1)] / [(P_2.P_1) • (P_2.P_1)]	t =[(P_2-P_1)•(P_3-P_1)] / [(P_2-P_1)•(P_2-P_1)]
15	8	A =[(P2.P1)^(P3.P1)]/2	A=[(P2-P1)^(P3-P1)]/2
15	19	E=[(P2 . P1)•(P3.P1)]	E=[(P2 - P1)•(P3-P1)]
16	10	(P2 . P1) e (P3 . P1)	(P2 - P1) e (P3 - P1)
16	13	P2.P1=[2,6][4,12][0,0]	P2-P1=[2,6][4,12][0,0]
16	14	P3.P1 = [2,4][0,2][0,0]	P3-P1 = [2,4][0,2][0,0]
17	3	P2.P1=(4,12,0).(2,0,0)	P2-P1=(4,12,0)-(2,0,0)
17	4	P2 . P1 = (2, 12, 0)	P2 - P1 = (2, 12, 0)
17	5	P3-P1=(4,2,0).(0,2,0)	P3-P1=(4,2,0)-(0,2,0)
17	13	P2.P1=(6,6,0).(0,2,0)	P2-P1=(6,6,0)-(0,2,0)
17	14	P2 . P1 = (6, 4, 0)	P2 - P1 = (6, 4, 0)
17	15	P3.P1=(4,2,0).(2,0,0)	P3-P1=(4,2,0)-(2,0,0)
17	16	P3 . P1 = (2, 2, 0)	P3 - P1 = (2, 2, 0)
17	17	(P2 . P1) ^ (P3 . P1) = (6, 4, 0) ^ (2, 2, 0) = (0, 0, 4)	(P2 - P1) ^ (P3 - P1)= (6, 4, 0) ^ (2,2,0) = (0, 0, 4)
17	23	(P2 . P1) e (P3.P1)	(P2 - P1) e (P3-P1)
18	2	P2.P1=[2,6][4,12][0,0]	P2-P1=[2,6][4,12][0,0]
18	3	P3.P1=[6,6][2,4][0,0]	P3-P1=[6,6][2,4][0,0]
18	4	P2 . P1 e P3 . P1	P2 - P1 e P3 - P1
18	11	2*A=(P1.P3)^(P2.P3)	2*A=(P1-P3)^(P2-P3)
18	17	A1=[(P2.P1)^(P3.P1)] /2 A2=[(P1.P2)^(P3.P2)] /2 A3=[(P1.P3)^(P2.P3)] /2	A1=[(P2-P1)^(P3-P1)]/2 A2=[(P1-P2)^(P3-P2)]/2 A3=[(P1-P3)^(P2-P3)]/2
19	3	E1=[(P2 - P1)•(P3-P1)] E2=[(P1 . P2)•(P3.P2)] E3=[(P1 . P3)•(P2.P3)]	E1=[(P2 -P1)•(P3-P1)] E2=[(P1 -P2)•(P3-P2)] E3=[(P1 -P3)•(P2-P3)]

PÁGINA	LINHA	ONDE SE LÊ	LEIA-SE
19	3	A1'=[(P2.P1)^(P3.P1)] A2'=[(P1.P2)^(P3.P2)] A3'=[(P1.P3)^(P2.P3)] E1=[(P2-P1)•(P3-P1)] E2=[(P1.P2)•(P3.P2)] E3=[(P1.P3)•(P2.P3)]	A1'=[(P2-P1)^(P3-P1)] A2'=[(P1-P2)^(P3-P2)] A3'=[(P1-P3)^(P2-P3)] E1=[(P2-P1)•(P3-P1)] E2=[(P1-P2)•(P3-P2)] E3=[(P1-P3)•(P2-P3)]
20	14	E1=[(P2-P1)•(P3 - P1)] E2=[(P1.P2)•(P3 . P2)] E3=[(P1.P3)•(P2 . P3)] A3=[(P1.P3)^(P2 . P3)]	E1=[(P2-P1)•(P3-P1)] E2=[(P1-P2)•(P3-P2)] E3=[(P1-P3)•(P2-P3)] A3=[(P1-P3)^(P2-P3)]
20	28	V = [(P2.P1) ^ (P3.P1)] • (P4.P1) /4	V = [(P2-P1) ^ (P3-P1)] • (P4-P1) /4
21	1	V1 = [(P2 . P1) ^ (P3.P1)]•(P4 . P1) /4 V2 = [(P1 . P2) ^ (P3.P2)]•(P4 . P2) /4 V3 = [(P1 . P3) ^ (P2.P3)]•(P4 . P3) /4 V4 = [(P1 . P4) ^ (P2.P4)]•(P3 . P4) /4	V1 = [(P2 - P1) ^ (P3-P1)]•(P4 - P1) /4 V2 = [(P1 - P2) ^ (P3-P2)]•(P4 - P2) /4 V3 = [(P1 - P3) ^ (P2-P3)]•(P4 - P3) /4 V4 = [(P1 - P4) ^ (P2-P4)]•(P3 - P4) /4
34	17	União (operador básico): A ∪ B Subtração: A.B= NS(NS(A)∪B) Intersecção: A ∩ B = NS(NS(A) ∪ NS(B)) Subtração (operador básico): A.B União: A∪B = NS(NS(A).B) Intersecção: A∩B = A.NS(B) Intersecção (operador básico): A∩B União: A ∪ B = NS(NS(A) ∩ NS(B)) Subtração: A.B = A ∩ NS(B)	União (operador básico): A∪B Subtração: A - B = NS(NS(A) ∪ B) Intersecção: A ∩ B = NS(NS(A) ∪ NS(B)) Subtração (operador básico): A-B União: A∪B=NS(NS(A)- B) Intersecção: A∩B = A -NS(B) Intersecção (operador básico): A∩B União: A ∪ B = NS(NS(A) ∩ NS(B)) Subtração: A-B = A∩NS(B)
51	15	v.e + f=2*(s.h) + r	v-e +f=2*(s-h) + r

PÁGINA	LINHA	ONDE SE LÊ	LEIA-SE
51	17	`r = l . f`	`r = l - f`

58	22	`x0 = x1 + t * (x2 . x1)`
		`y0 = y1 + t * (y2 . y1)`
		`z0 = z1 + t * (z2 . z1)`
			`x0=x1 + t * (x2 - x1)`
			`y0=y1 + t * (y2 - y1)`
			`z0=z1 + t * (z2 - z1)`

58	26	`nx * (x1 + t*(x2.x1) )+`
		`ny *(y1 + t*(y2.y1) ) +`
		`nz *(z1 + t*(z2.z1) ) +`
		`nw = 0`
			`nx *(x1 + t*(x2-x1) ) +`
			`ny*(y1 +t*(y2-y1) ) +`
			`nz*(z1 +t*(z2-z1) ) +`
			`nw = 0`

60	9	`( 1.t )*v1 +t*v2 = v(t)`	`( 1-t )*v1+t*v2=v(t)`
60	11	$t * (v_2 . v_1) = v(t) . v_1$	$t * (v_2 - v_1) = v(t) - v_1$
60	13	$t=(v_3 . v_1) / (v_2 . v_1);$	$t = (v_3 - v_1)/(v_2 - v_1);$
60	15	$(v_2 . v_1)$:	$(v_2 - v_1)$:
60	16	$t = [(v_2 . v_1) \bullet (v_3 . v_1)]$ $/ [(v_2 . v_1) \bullet (v_2 . v_1)]$	$t = [(v_2 - v_1) \bullet (v_3 - v_1)]$ $/ [(v_2 - v_1) \bullet (v_2 - v_1)]$

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

B-Rep	– *Boundary Representation*
OpenGL	– *Open Graphic Library*
STL	– *Standard Template Library*

1. INTRODUÇÃO

Atualmente, os sistemas de modelagem de sólidos são amplamente utilizados. Exemplo disto é o seu uso no desenvolvimento de novos produtos, no processo de fabricação para armazenar as informações sobre as peças, durante o projeto e análise de componentes mecânicos através de sistemas de análise por elementos finitos e na área de entretenimento como ferramenta para geração de modelos de objetos e personagens para posterior uso em computação gráfica para filmes ou jogos de computador.

Um componente principal dos modeladores de sólido é a máquina de operações booleanas. Estas operações facilitam a criação de sólidos complexos a partir de sólidos simples. Por uma análise das publicações na literatura é possível observar que o desenvolvimento de algoritmos para implementar operações booleanas envolve a definição de algoritmos de classificação geométrica entre os elementos primitivos (vértice, aresta e face). É necessário determinar se um vértice de um sólido está posicionado sobre algum vértice de outro sólido, ou se uma aresta de um sólido intersecciona alguma aresta de outro sólido definindo um vértice de intersecção. As classificações geométricas conhecidas como testes de incidência, realizam verificações para determinar se um elemento geométrico incide em outro elemento geométrico. Os testes de incidência de maior importância neste trabalho são: a análise de incidência de um vértice com outro vértice, em que é determinado se os dois vértices possuem coordenadas iguais; a análise de incidência de um vértice com um segmento de reta, em que é determinado se existe um ponto pertencente ao segmento de reta com coordenadas iguais ao do vértice que está sendo analisado; e a análise de incidência de vértice com um plano, onde é determinado se existe um ponto pertencente ao plano com coordenadas iguais ao do vértice que está sendo analisado. É comum implementar algoritmos que, por meio de manipulação numérica com uma tolerância, determinam a classificação geométrica. Entretanto, a manipulação numérica de ponto flutuante é susceptível a erros e pode ocorrer instabilidade numérica na determinação de pontos de intersecção.

Nos modeladores de sólidos atuais, valores em ponto flutuante são utilizados para representar elementos geométricos na forma computacional. Entretanto, o ponto

flutuante representa apenas uma quantidade finita de números racionais, enquanto que os números reais que se encontram no intervalo entre dois números racionais não podem ser representados de forma adequada. Ou seja, uma aproximação dos números reais é realizada quando valores numéricos são computados, ocorrendo uma imprecisão devido ao uso de uma representação discreta ser utilizada para representar elementos geométricos contínuos. Como conseqüência desta falta de precisão, podem surgir imprecisões nos testes de incidência e inconsistências entre a geometria e a topologia dos elementos geométricos de um sólido. Ocorrer uma falha em um teste de geometria significa, por exemplo, que durante a determinação de pontos de intersecção entre sólidos na operação booleana, alguns deles podem não ser encontrados, o que pode levar o sistema de modelagem de sólidos a construir sólidos inconsistentes ou até mesmo provocar um travamento do software.

O exemplo apresentado na Figura 1.1 mostra uma seqüência de operações em que podem surgir erros de aproximação. Na Figura 1.1a tem-se dois sólidos **A** e **B** e o ponto **P** em destaque. Em seguida, na Figura 1.1b, foi aplicada a operação de translação ao sólido **A**, de modo que o ponto **P** também foi deslocado. Na Figura 1.1c, o sólido **A** foi rotacionado. Ao ser rotacionado de volta, o ponto **P** do sólido **A** sofreu um erro de aproximação e seu valor ficou diferente do valor inicial. Ao ser aplicado novamente o operador de translação, o ponto que antes tinha valor de **P=(0, 0, 0)**, possui valor de **P = (-0.01, 0, 0)**; computacionalmente diferente do valor inicial.

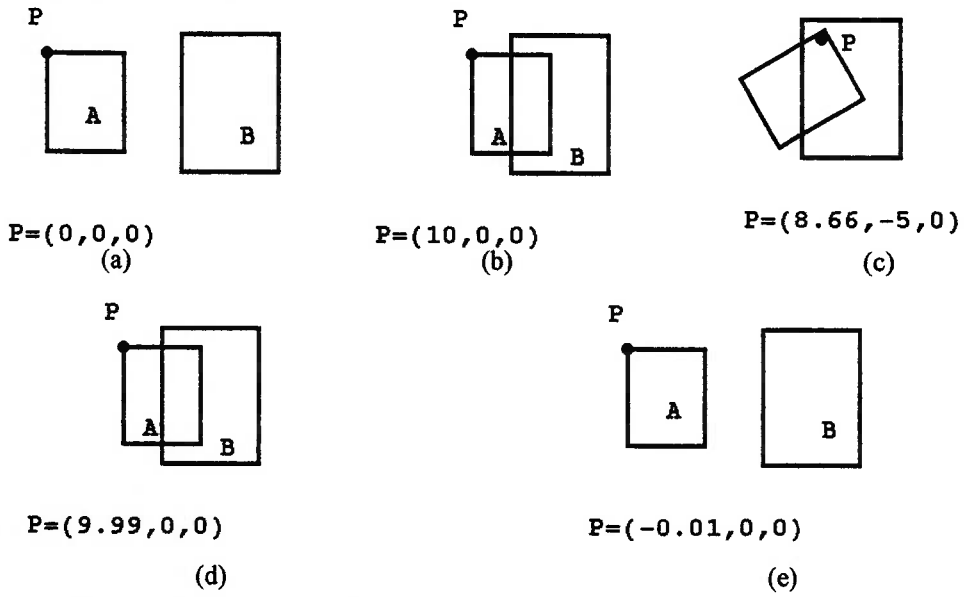
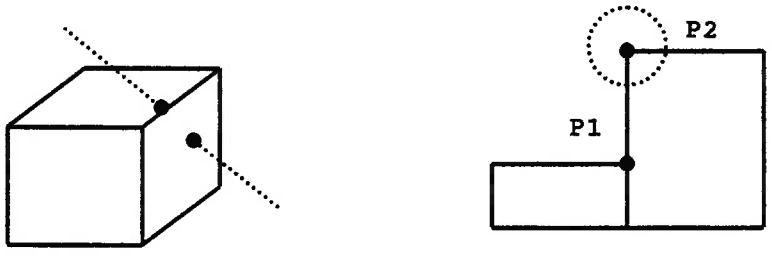


Fig. 1.1. Exemplo de sequëncia de operações para surgir erro de aproximação.

Um dos incovenientes da utilização da tolerância para comparação numérica é que esta tolerância é fixa, enquanto erros de aproximação podem propagar numa seqüência de operações com os elementos geométricos. Deste modo, o erro de aproximação fica maior a cada operação, podendo ficar maior do que a tolerância permitida, comprometendo os testes de incidência.

Exemplos de inconsistências em sólidos são apresentados na Figura 1.2 a e b. Na Figura 1.2 a, uma linha intercepta um sólido, sendo que em uma das faces, o ponto de intersecção está numa aresta e na face adjacente o ponto de intersecção se encontra no interior da face. Deste modo, dois pontos distintos serão adicionados à estrutura de dados do sólido o que irá gerar falhas. Na Figura 1.2 b, dois pontos `P1 = (0, 1, 0)` e `P2 = (0, 1.0099, 0)` estão em destaque, sendo que a tolerância adotada de `0.001` não permite considerar que os dois pontos sejam incidentes. Pode-se imaginar que erros de aproximação seqüenciais fizeram com que a coordenada variasse de `1.0000` para `1.0099`.



(a) (b)

Fig. 1.2. a) linha intercepta sólido de forma inconsistente geometricamente; b) pequena tolerância não permite que pontos sejam considerados incidentes.

Neste trabalho é considerada a utilização de aritmética intervalar (Hu e Patrikalaris, 1996) para aumentar a robustez dos algoritmos de classificação geométrica utilizados nas operações de corte e booleanas. A aritmética intervalar incorpora os erros de aproximação, eliminando a necessidade de definir uma tolerância fixa para realizar a comparação entre números de ponto flutuante. É conveniente salientar que Hu et al (1996) propuseram a aplicação da aritmética intervalar para modeladores de sólidos B-rep. Entretanto, eles não abordaram as implicações do uso da aritmética intervalar de classificação geométrica.

Logo, este trabalho propõe o uso da aritmética intervalar em um modelador de sólido B-rep para aumentar a robustez das operações de corte e booleanas. A aritmética intervalar lidará com os cálculos em ponto flutuante, aumentando a sua robustez.

Uma introdução sobre aritmética intervalar é apresentada na seção 2, em seguida aplicações estão apresentadas na seção 3. Um estudo sobre modeladores de sólidos B-rep e a estrutura de dados são apresentados na seção 4. Os dois principais algoritmos do molelador de sólidos, operador de corte e operações booleanas são apresentados na seção 5. Resultados são apresentados na seção 6. A seção 7 apresenta as conclusões.

2. ARITMÉTICA INTERVALAR ARREDONDADA

Um intervalo convencional consiste em um conjunto de números reais definido por:

$$[a, b] = \{ x \mid a \leq x \leq b \}$$

O intervalo $[a, b]$ é considerado degenerado se $a = b$.

A intersecção de dois intervalos é vazia se $a > d$ ou $c > b$. Do contrário:

$$[a, b] \cap [c, d] = [\, \max(a, c), \min(b, d) \,]$$

A união de dois intervalos que possuem intersecção é obtida por:

$$[a, b] \cup [c, d] = [\, \min(a, c), \max(b, d) \,].$$

Uma ordem de intervalos é definida por $[a, b] < [c, d]$ se e somente se $b < c$.

As comparações em aritmética intervalar devem ser repensadas. Dois intervalos podem ser classificados de três maneiras: certamente (ou rigorosamente) iguais, possivelmente iguais ou certamente não iguais.

Dois intervalos $[a, b]$ e $[c, d]$ são considerados certamente (ou rigorosamente) iguais se $a = c$ e $b = d$. A Figura 2.1 representa esta situação.

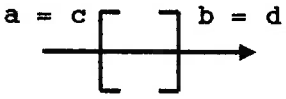


Fig. 2.1. Intervalos certamente (ou rigorosamente) iguais.

Dois intervalos $[a, b]$ e $[c, d]$ são considerados certamente (ou rigorosamente) não iguais se estes intervalos estiverem afastados um do outro de modo a não existir intersecção entre eles. A Figura 2.2 representa esta situação.

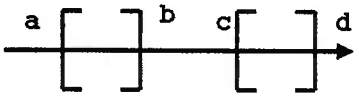


Fig. 2.2. Intervalos certamente (ou rigorosamente) não iguais.

A terceira e última situação: dois intervalos $[a, b]$ e $[c, d]$ são considerados possivelmente iguais se existir intersecção entre eles. A Figura 2.3 representa esta situação.

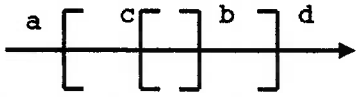


Fig. 2.3.Intervalos possivelmente iguais.

Uma comparação muito realizada em computação é a comparação de um valor com zero. Na implementação com ponto flutuante, um valor é considerado igual a zero, se este valor for menor que a tolerância. Em arimética intervalar, um valor intervalar `[a, b]` é considerado igual a zero se for possívelmente igual a zero, ou seja, o intervalo engloba o zero, de modo que um dos extremos seja zero, ou um extremo é negativo e o outro é positivo, conforme ilusta a Figura 2.4.

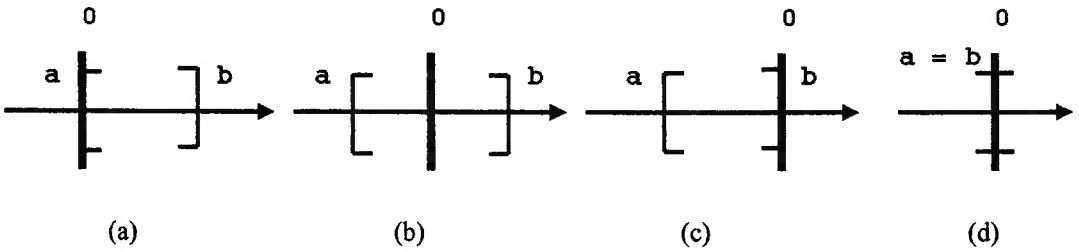


Fig. 2.4. a, b, c)Intervalos possivelmente iguais a zero; d) intervalo certamente (ou rigorosamente) igual a zero.

Logo, as operações utilizando aritmética intervalar são definidas como:

```
[a, b] + [c, d] ≡ [ a + c - ∈, b + d + ∈]
[a, b] -  [c, d] ≡ [ a - d - ∈, b - c + ∈]
[a, b] . [c, d] ≡ [    min(ac, ad, bc, bd) - ∈,
                       max(ac, ad, bc, bd) + ∈]
[a, b] / [c, d] ≡ [    min(a/c, a/d, b/c, b/d) - ∈,
                       max(a/c, a/d, b/c, b/d) + ∈]
```

Onde $\in$ representa a diferença entre o número em ponto flutuante em análise e o número em ponto flutuante imediatamente superior. Quando forem realizadas operações padrões utilizando-se de números intervalares, as extremidades inferior e superior são estendidas para incluir o seu número em ponto flutuante imediatamente anterior e posterior, respectivamente. Então o comprimento do resultado é aumentado de $2 \cdot \in$ e o resultado será confiável nas operações subsequëntes. O valor

de $\in$ depende do número que está sendo analisado, sendo que ele deve ser determinado a cada operação. O algoritmo utilizado está na listagem abaixo: (Hu e Patrikalaris, 1996).

```
typedef union {
      double dp;
      unsigned short sh[4];
} Double;

#define MSW 3      /* 0 if the left-most 16-bit is sh[0] */
                   /* 3 if the left-most 16-bit is sh[3] */

static unsigned short mask[16] = {
      0x0001, 0x0002, 0x0004, 0x0008,
      0x0010, 0x0020, 0x0040, 0x0080,
      0x0100, 0x0200, 0x0400, 0x0800,
      0x1000, 0x2000, 0x4000, 0x8000
};

double ulp2(double x) {
      Double X, U;
      int bit, e1, word;

      X.dp = x;
      X.sh[MSW] &= 0x7ff0;
      U.dp = 0.0;
      if (X.sh[MSW] > 0x0340) U.sh[MSW] = X.sh[MSW] - 0x0340;
      else {
            e1 = (X.sh[MSW] >> 4) -1;
            word = e1 >> 4;
            if (MSW == 0 ) word = 3 - word;
            bit = e1 % 16;
            U.sh[word] |= mask[bit];
      }
      return U.dp;
}
```

Naturalmente, a utilização de aritmética intervalar implica em uma carga computacional maior que a utilização de metodologia com tolerância, o que pode ser notado pela definição das operações aritméticas básicas que necessitam de mais cálculos e verificações. A robustez numérica é adquirida com o preço de uma carga computacional maior.

Em arimética intervalar expressões algébricas equivalentes podem possuir resultados distintos. Naturalmente os resultados são possivelmente iguais mas com intervalos de comprimento diferentes. Como exemplo, a expressão $f(x) = 2x^3 \cdot 3x^2 + 1$, com $x = [1, 2]$, resulta em $[-9,14]$. Enquanto que a expressão equivalente $f(x) = x^2 (2x \cdot 3) + 1$ resulta $[-3, 5]$. É preferivel utilizar expressões algébricas que forneçam os resultados mais estreitos.

Segundo Hu e Patrikalaris (1996), a simetria de incidência e a transitividade de incidência são exemplos de situações onde nota-se claramente a robustez superior

da aritmética intervalar contra a aritmética de ponto flutuante. O problema da simetria de incidência no contexto da aritmética do ponto flutuante significa que um ponto pode ser incidente a outro mas não vice-versa. Em aritmética intervalar, uma vez que o ponto é considerado incidente a outro, este também deve ser incidente ao primeiro.

No problema de transitividade de incidência, devido ao uso da tolerância pré - fixada em aritmética de ponto flutuante, um ponto **A** pode ser incidente a um ponto **B**, e este pode ser incidente a um ponto **C**. Mas o ponto **C** pode não ser incidente ao ponto **A**, pois a tolerância não é grande o suficiente para alcançar o ponto **A**. Em aritmética intervalar, neste caso, os pontos incidentes **A** e **B** são repostos por um novo ponto que cobre estes dois pontos, ou seja, o intervalo aumenta. Deste modo, o ponto **C** torna-se incidente a este novo ponto. A Figura 2.5 ilustra o problema da transitividade de incidência.

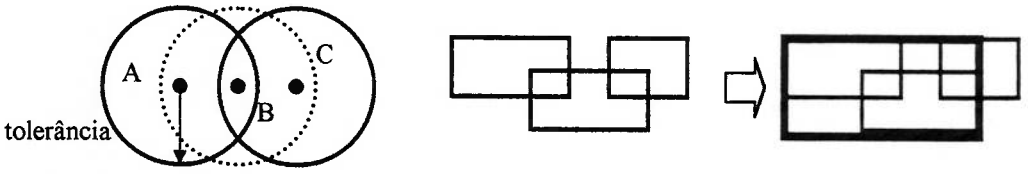


Fig. 2.5. Transitividade de incidência com aritmética do ponto flutuante e aritmética intervalar

Um exemplo simples do uso da aritmética intervalar é caso da divisão do número um por três. Este valor é uma dízima 0.333333..., mas o microcomputador apresentará o seguinte valor (usando tipo "*double*" – dupla precisão):

Ex.: 1/3 = 0.33333333333333331000

Ao se utilizar a aritmética intervalar, esse valor fica armazenado como:

Ex.: 1/3 = [0.33333333333333326000 0.33333333333333337000]

O valor é coerente, pois o valor real é interno ao intervalo.

3. REPRESENTAÇÃO DOS ELEMENTOS GEOMÉTRICOS E TESTES DE INCIDÊNCIA – UTILIZANDO ARITMÉTICA INTERVALAR

Nesta seção estão descritos as aplicações da aritmética intervalar no modelador de sólidos.

Na primeira parte desta seção são estudados as representações dos elementos geométricos vértice, linha e vetor utilizando arimética intervalar. Desta forma, é possível analisar o significado geométrico dos intervalos e sua representação no espaço geométrico.

Na segunda parte são estudados os testes de incidência, que são de grande importância nos algoritmos empregados no modelador de sólidos. Relembrando que os testes de incidência verificam se existe intersecção entre os conjuntos de pontos associados aos elementos geométricos. Os casos estudados são ponto e ponto, ponto e linha e ponto e plano.

3.1. Elemento Geométrico - Vértice Intervalar

Cada coordenada do vértice é representada por um valor intervalar. E cada valor intervalar possui: extremo inferior e extremo superior. A representação bidimensional do vértice intervalar é um retângulo, e a tridimensional, um bloco (Figura 3.1).

```
V1: ( [x1, x2], [y1, y2], [z1, z2] )
```

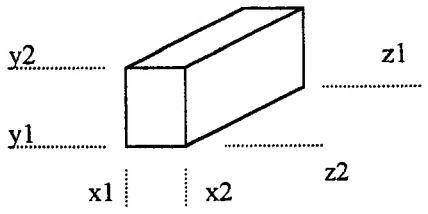


Fig. 3.1. Representação tridimensional de um vértice intervalar.

3.2. Elemento Geométrico - Linha Intervalar

A linha intervalar é formada por dois vértices intervalares. A representação bidimensional da linha intervalar é um polígono de seis lados; e a tridimensional, um poliedro (Figura 3.2).

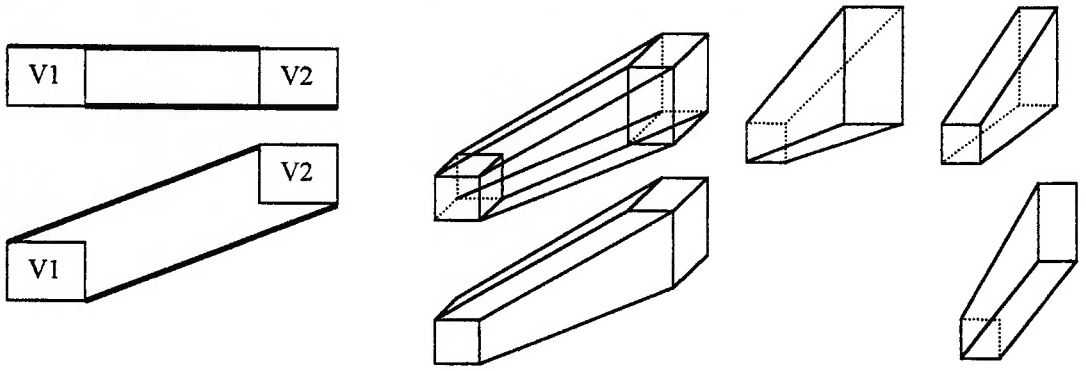


Fig. 3.2. Linha Intervalar bidimensional e tridimensional e vistas ortográficas da linha tridimensional.

3.3. Elemento Geométrico - Vetor Intervalar

O vetor é um elemento aparentemente simples, mas que permite ilustrar a possibilidade do intervalo aumentar em demasia.

Um vetor é definido como:

$$\mathbf{v} = \overline{P1P2} = P2 \;.\; P1$$

sendo:

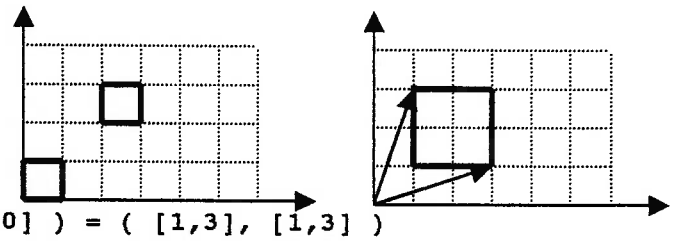
$$P1 = ([x1,x2] , [y1, y2], [z1,z2])$$
$$P2 = ([x3,x4] , [y3, y4], [z3,z4])$$

Logo, utilizando as operações básicas definidas para aritmética intervalar, o vetor v é obtido como sendo:

$$V = ([x3 \;.\; x2,\; x4 \;.\; x1],\; [\; y3 \;.\; y2,\; y4 \;.\; y1] \;,\; [\; z3 \;.\; z2,\; z4 \;.\; z1])$$

Um exemplo 2D é apresentado na Figura 3.3 :

$$P1 = ([0,1] , [\ 0,1])$$
$$P2 = ([2,3] , [\ 2,3])$$



$$v = ([\ 2-1,\ 3-0], [\ 2-1,\ 3-0]) = ([1,3], [1,3])$$

Fig. 3.3. Gráfico do vetor v = ([1,3], [1,3]).

Nota-se que a região que o vetor ocupa é grande. Esta região contém todos os casos extremos de vetores possíveis entre os pontos `P1` e `P2`. Alguns exemplos na Figura 3.4.

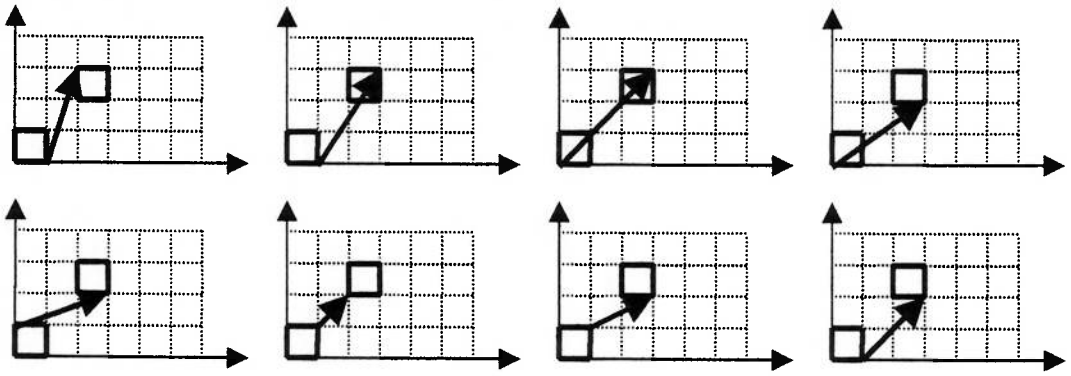


Fig. 3.4. Exemplos do alcance do vetor `v = ( [1,3] , [1,3] )`.

Ao calcular-se `P1` + $\overline{\texttt{P1P2}}$ obtém-se `([1,4],[1,4])` que deveria ser na realidade o mesmo que `P2`. Apesar de serem valores possivelmente iguais, este resultado `([1,4],[1,4])` possui maiores intervalos do que `([2,3],[2,3])` conforme é representado na Figura 3.5.

```
P1 = ( [0,1] , [ 0,1] )
P2 = ( [2,3] , [ 2,3] )
P1P2 = ( [1,3], [1,3] )
P1 + P1P2 =
([0,1],[0,1]) + ([1,3],[1,3]) = ( [1,4],[1,4] )
```

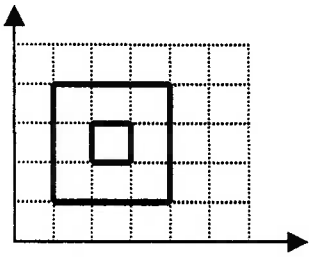


Fig. 3.5. Exemplo de aumento de intervalo.

3.4. Teste de Incidência - Ponto e Ponto

Dados dois pontos `P1` e `P2` (Figura 3.6):

P_1 ($[x_1, x_2]$, $[y_1, y_2]$, $[z_1, z_2]$)
P_2 ($[x_3, x_4]$, $[y_3, y_4]$, $[z_3, z_4]$)

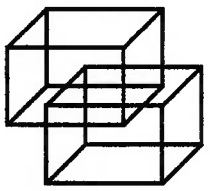


Fig. 3.6. Pontos intervalares P_1 e P_2.

Para que o ponto P_1 seja considerado incidente ao P_2, é necessário que todas estas condições estejam satisfeitas:

- Intervalo $[x_1, x_2]$ tenha algum valor de intersecção com o intervalo $[x_3, x_4]$;
- Intervalo $[y_1, y_2]$ tenha algum valor de intersecção com o intervalo $[y_3, y_4]$;
- Intervalo $[z_1, z_2]$ tenha algum valor de intersecção com o intervalo $[z_3, z_4]$;

Se for confirmada a incidência, um novo ponto P_3 é criado, que engloba os dois pontos **P1** e **P2**, conforme é representado na Figura 3.7.

$$P_3 ([\min(x_1, x_3), \max(x_2, x_4)], [\min(y_1, y_3), \max(y_2, y_4)], [\min(z_1, z_3), \max(z_2, z_4)])$$

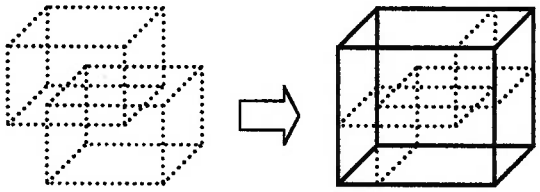


Fig. 3.7. O novo vértice intervalar engloba os dois vértices intervalares **P1** e **P2**.

3.5. Teste de Incidência - Ponto e Linha

Dados três pontos **P1**, **P2** e **P3** representados na Figura 3.8.

$$P_1 ([x_1, x_2], [y_1, y_2], [z_1, z_2])$$
$$P_2 ([x_3, x_4], [y_3, y_4], [z_3, z_4])$$
$$P_3 ([x_5, x_6], [y_5, y_6], [z_5, z_6])$$

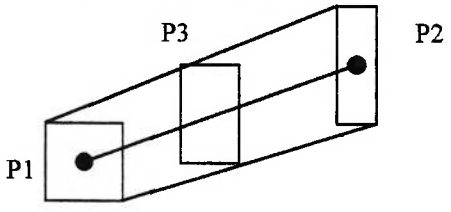


Fig. 3.8. Três Pontos Intervalares.

Este teste de incidência poderia ser feito de várias formas se fosse utilizada aritmética de ponto flutuante. Estes métodos foram aplicados em aritmética intervalar e durante alguns testes utilizando valores númericos, foram detectadas algumas falhas que serão descritas a seguir.

O primeiro método de incidência testado baseia-se em determinar o ponto que pertence a linha que pode ser coincidente com o ponto fornecido.

Para que o ponto P_3 seja considerado incidente à linha formada pelos pontos P_1 e P_2, os seguintes passos devem ser realizados:

- inicialmente supor que o ponto P_3 realmente pertença à linha P_1P_2 e então calcular o parâmetro t. Este parâmetro t, decorre da seguinte equação:

$$(1 . t) * P_1 + t * P_2 = P(t) \qquad \text{onde } 0 \leq t \leq 1$$

Para $t = 0, P(0) = P_1$; para $t=1, P(1) = P_2$

Rescrevendo a equação, tem-se:

$$t * (P_2 . P_1) = P(t) . P_1$$

Para $P(t) = P_3$, obtém-se:

$$t = (P_3 . P_1) / (P_2 . P_1)$$

Como não é definida a divisão de vetores e o valor de t é um valor escalar, multiplicando o numerador e o denominador escalarmente por um outro vetor, é possível obter o valor de t. O vetor escolhido foi $(P_2 . P_1)$, assim:

$$t = [(P_2 . P_1) \bullet (P_3 . P_1)] \; / \; [(P_2 . P_1) \bullet (P_2 . P_1)]$$

- com o valor t obtido, pode-se criar um novo ponto P_{teste}, que está na linha formada por P_1 e P_2. Em seguida, são calculados os valores extremos do intervalo do ponto P_{teste} com base nos extremos de P_1 e P_2 (Figura 3.9) para que o ponto não ultrapasse os limites da linha intervalar. Corrigindo o intervalo para que ele não cresça em demasia.

$$P_{teste} ([x_a, x_b], [y_a, y_b], [z_a, z_b])$$
$$x_a = (1-t) * x_1 + t * x_3$$
$$x_b = (1-t) * x_2 + t * x_4$$
$$y_a = (1-t) * y_1 + t * y_3$$
$$y_b = (1-t) * y_2 + t * y_4$$
$$z_a = (1-t) * z_1 + t * z_3$$
$$z_b = (1-t) * z_2 + t * z_4$$

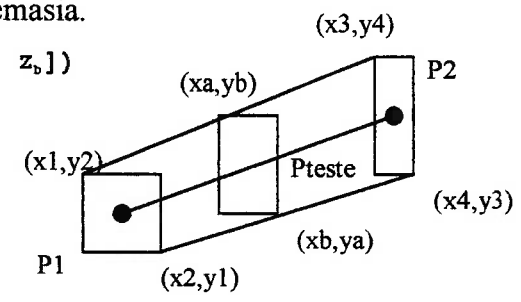
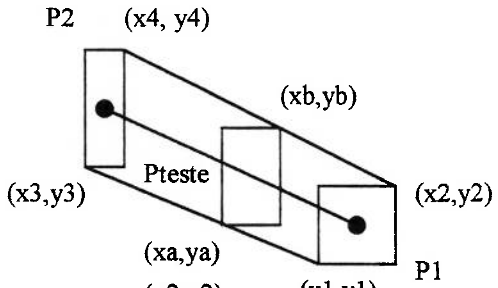


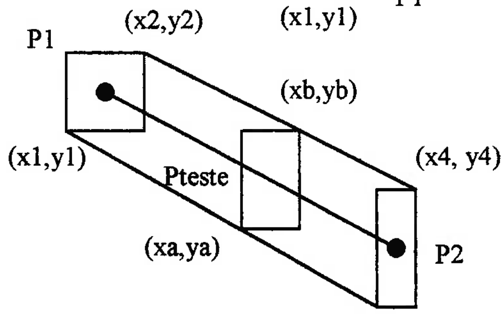
Fig. 3.9. Cálculo dos extremos do ponto intervalar incidente na linha intervalar.

Verificando as outras três posições relativas entre P1 e P2 no caso 2D, obtém-se (Figura 3.10):

$$x_a = (1-t) * x_1 + t * x_3$$
$$x_b = (1-t) * x_2 + t * x_4$$
$$y_a = (1-t) * y_1 + t * y_3$$
$$y_b = (1-t) * y_2 + t * y_4$$
$$z_a = (1-t) * z_1 + t * z_3$$
$$z_b = (1-t) * z_2 + t * z_4$$



$$x_a = (1-t) * x_1 + t * x_3$$
$$x_b = (1-t) * x_2 + t * x_4$$
$$y_a = (1-t) * y_1 + t * y_3$$
$$y_b = (1-t) * y_2 + t * y_4$$
$$z_a = (1-t) * z_1 + t * z_3$$
$$z_b = (1-t) * z_2 + t * z_4$$



$$x_a = (1-t) * x_1 + t * x_3$$
$$x_b = (1-t) * x_2 + t * x_4$$
$$y_a = (1-t) * y_1 + t * y_3$$
$$y_b = (1-t) * y_2 + t * y_4$$
$$z_a = (1-t) * z_1 + t * z_3$$
$$z_b = (1-t) * z_2 + t * z_4$$

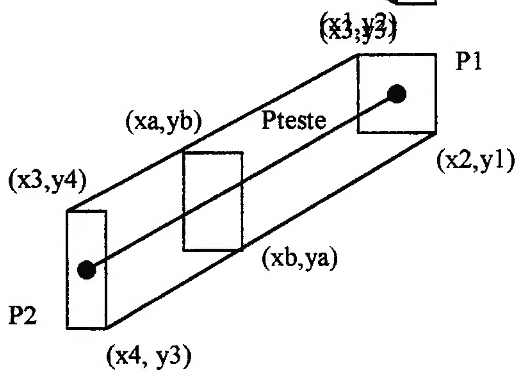


Fig. 3.10. Outras três posições relativas para uma linha intervalar.

Entretanto, sendo a correção feita em aritmética de ponto flutuante pode levar a ser perdido algum valor por arredondamento. Assim, a determinação pode ficar mais robusta utilizando-se valores intervalares para o resultado de xa, xb, ya, yb, za e zb (Figura 3.11).

P_{teste} ($[x_a, x_b]$, $[y_a, y_b]$, $[z_a, z_b]$)

x_a = [xa_min, xa_max]

P_{teste} ([min(x_a),max(x_b)], [min(y_a), max(y_b)], [min(z_a), max(z_b)])

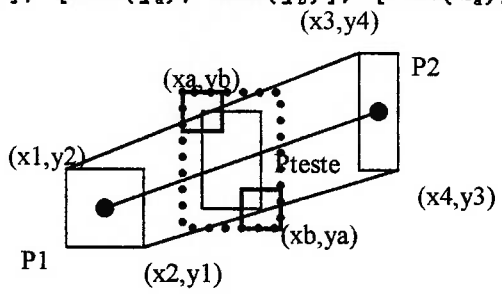


Fig. 3.11. Aumentando os extremos do ponto intervalar.

A segunda técnica para verificar se o ponto **P3** pertence à linha delimitada pelos pontos **P1** e **P2**, é o cálculo da área definida pelo triângulo formado pelos pontos **P1, P2 e P3** (Figura 3.12). Se a área for nula, significa que o pontos são colineares, o que significa que existe a possibilidade do ponto **P3** estar sobre a linha formada por **P2** e **P1**.

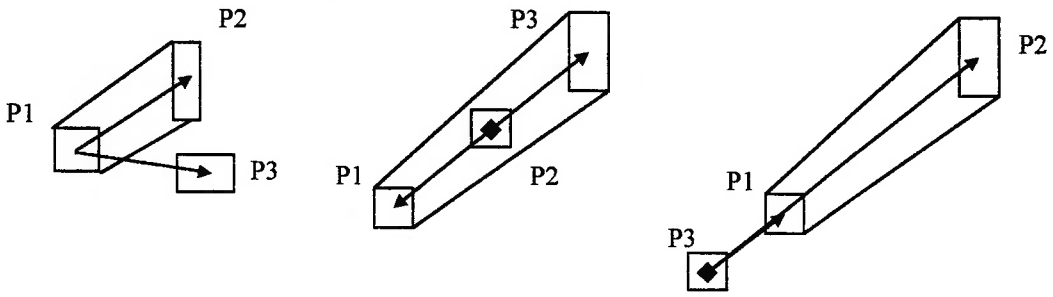


Fig. 3.12. Triângulo formado por **P1**, **P2** e **P3**.

Esta área **A = (Ax, Ay, Az)** é obtida pelo seguinte cálculo:

A = [(P2 . P1) ^ (P3 . P1)] / 2

Em seguida, o produto escalar é aplicado para determinar se o ponto **P3** está entre os pontos **P1** e **P2** ou fora da linha formada. Caso o produto escalar for menor ou igual a zero, o ponto é considerado como estando entre **P1** e **P2**.

E = [(P2 . P1) • (P3 . P1)]

Entretanto, um teste inicial utilizando valores numéricos foi falho. Este teste consistia em fornecer um ponto **P3** fora da linha intervalar, mas muito próximo da linha. O algoritmo indicava que o ponto **P3** estava incidente à linha (Figura 3.13). Que é um resultado incorreto pois **P3** não é incidente a linha **P1P2**.

```
P1: [ 0, 2] [ 0, 2]  [0, 0]
P2: [ 4, 6] [ 6, 12] [0, 0]
P3: [ 4, 4] [ 2, 2]  [0, 0]
```

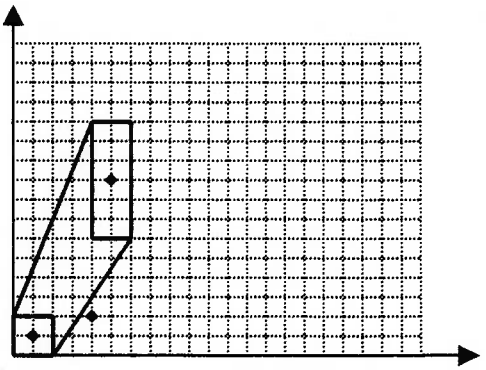


Fig 3.13. Representação do teste de incidente entre uma linha intervalar e um ponto muito próximo.

O valor obtido para `2 * A` foi `([ 0, 0] [0, 0] [ -48, 4])`. Este valor indica que a área é possivelmente igual a zero (`Ax` e `Ay` são zeros, enquanto que `Az` tem extremo inferior negativo e extremo superior positivo, de modo que o valor zero é interno ao intervalo). Ou seja, o resultado indica que o ponto `P3` é incidente à linha formada por `P1` e `P2`. Como os exemplos aqui mostrados estão no plano `xy`, somente é mostrado o valor da área na componente `z` (`Az`), sendo que `Ax` e `Ay` são de valor zero.

Uma explicação encontrada para esta falha foi encontrada na representação gráfica dos vetores `(P2 . P1)` e `(P3 . P1)`. A Figura 3.14 indicou que estes vetores possuem um região de intersecção (ou seja, existe um vetor possivelmente na mesma direção) o que pode ter influenciado no cálculo da área.

Vetores:

```
P2 . P1 = [ 2, 6] [4, 12] [0, 0]
P3 . P1 = [ 2, 4] [ 0 ,2] [ 0, 0]
```

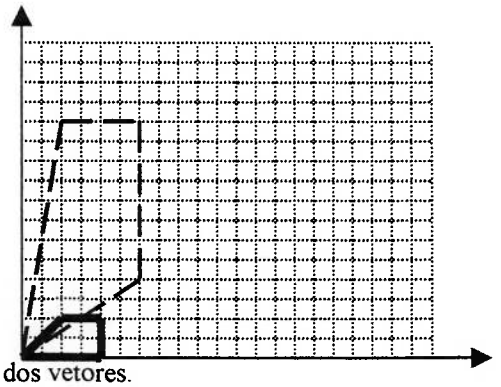
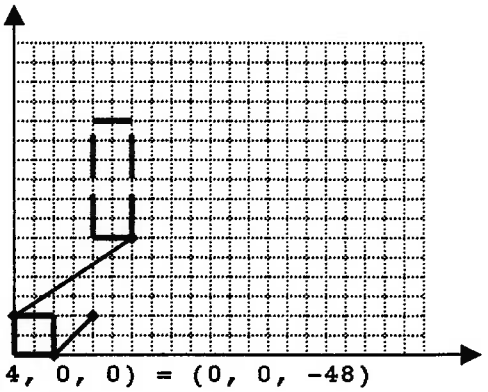


Fig. 3.14. Representação dos vetores.

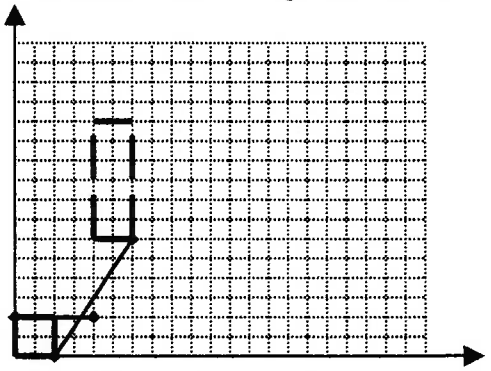
O valor do extremo inferior `2 * Az = .48` foi obtido com os seguintes vetores (Figura 3.15):



```
P3 = (4,2,0)      P1 = (0,2,0)
P2 = (4,12,0)     P1 = (2,0,0)
P2 . P1 = (4, 12, 0) . (2, 0, 0)
P2 . P1 = (2, 12, 0)
P3 - P1 = (4, 2, 0) . (0, 2, 0)
P3 - P1 = (4, 0, 0)
(P2 - P1) ^ (P3 - P1) = (2, 12, 0) ^ (4, 0, 0) = (0, 0, -48)
```

Fig. 3.15. Cálculo de área de um triângulo utilizando "4 pontos".

O valor extremo superior `2 * Az = 4` foi obtido com os seguintes valores (Figura 3.16):



```
P3 = (4,2,0)      P1 = (2,0,0)
P2 = (6,6,0)      P1 = (0,2,0)
P2 . P1 = (6, 6, 0) . (0, 2, 0)
P2 . P1 = (6, 4, 0)
P3 . P1 = (4, 2, 0) . (2, 0, 0)
P3 . P1 = (2, 2, 0)
(P2 . P1) ^ (P3 . P1) = (6, 4, 0) ^ (2, 2, 0) = (0, 0, 4)
```

Fig. 3.16. Cálculo de área de um triângulo utilizando "4 pontos".

Ou seja, a área está aparentemente sendo calculada com base em quatro pontos, sendo que dois pontos "pertencem" ao mesmo ponto intervalar.

Utilizando um ponto `P3 [ 8, 6] [4, 4] [ 0, 0]`, afastado da linha intervalar, verifica-se que o valor `2 * Az = [ -68, +1.4x10E-14 ]`, que é possivelmente igual a zero em aritmética intervalar. O gráfico dos vetores (Figura 3.17) mostra o limite para intersecção entre os vetores `(P2 . P1)` e `(P3 . P1)`.

Vetores:

```
P2 . P1 = [ 2, 6] [4, 12] [0, 0]
P3 . P1 = [ 6,6] [ 2, 4] [0,0]
```

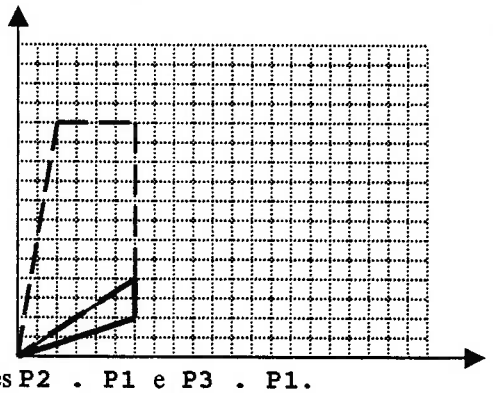


Fig. 3.17. Representação dos vetores `P2 . P1` e `P3 . P1`.

Diminuindo-se um pouco o valor de `P3` para `[8,6] [4, 3.9] [0,0]`, não ocorre a intersecção no gráfico de vetores. Então, a área obtida deverá ser não certamente igual a zero, o que de fato ocorre, pois obteve-se `2*Az=[-68,-0.6]`.

Logo, percebe-se que o uso de técnicas aplicadas em ponto flutuante, requer uma maior atenção quando forem utilizadas em aritmética intervalar.

Notou-se que, calculando-se os vetores com base no ponto `P3`, com `2*A=(P1.P3)^(P2.P3)`, esta técnica "parece" funcionar corretamente, obtendo `2 * Az = [4 ,40]` (área não nula). Logo, dependendo do ponto base em que se realiza o produto vetorial, o resultado é diferente. Os resultados apresentados mostram que `P3` não é colinear com `P2` e `P1` e `P1` é colinear com `P2` e `P3`.

Pode-se imaginar que o método funciona quando são realizados três cálculos de área com base em cada um dos pontos:

```
A1 = [ (P2 . P1) ^ (P3 . P1) ] /2
A2 = [ (P1 . P2) ^ (P3 . P2) ] /2
A3 = [ (P1 . P3) ^ (P2 . P3) ] /2
```

`A1 = 0` e `A2 = 0` e `A3 = 0` $\Rightarrow$ `P1`, `P2` e `P3` são colineares

Se `A1`, `A2` e `A3` forem possivelmente iguais, significa que, realmente, os três pontos `P1`, `P2` e `P3` são colineares. Naturalmente, caso alguma área calculada seja certamente (ou rigorosamente) não igual a zero, significa que os pontos não são colineares intervalares. Nos testes realizados utilizando valores numéricos, este método mostrou-se confiável.

Para determinar se o ponto está fora ou não da linha, são necessários três produtos escalares: um produto escalar `E3` com base no ponto `P3` para determinar se o ponto `P3` está no segmento sobre o segmento `P1P2`, outro `E1` com base no ponto

P1 para verificar se o ponto **P1** está fora do segmento **P2P3** e mais outro **E2** com base no ponto **P2** para verificar se o ponto **P2** está for a do segmento **P1P3**.

$$E1 = [\ (P2 - P1) \bullet (P3 - P1)\ ]$$
$$E2 = [\ (P1 \ . \ P2) \bullet (P3 \ . \ P2)\ ]$$
$$E3 = [\ (P1 \ . \ P3) \bullet (P2 \ . \ P3)\ ]$$

$E3 \leq 0$ e $E1 > 0$ e $E2 > 0 \Rightarrow$ **P3** sobre o segmento **P1P2**.

Se **E3** for menor ou igual a zero, significa que o ponto **P3** está sobre o segmento **P1P2**. Se **E1** for maior que zero, significa que **P1** está fora do segmento **P2P3**. Se **E2** for maior que zero, significa que **P2** está fora do segmento **P1P3**. (vide Figura 3.18a). Se algum destas verificações falhar, então o ponto **P3** não está sobre o segmento **P1P2**.

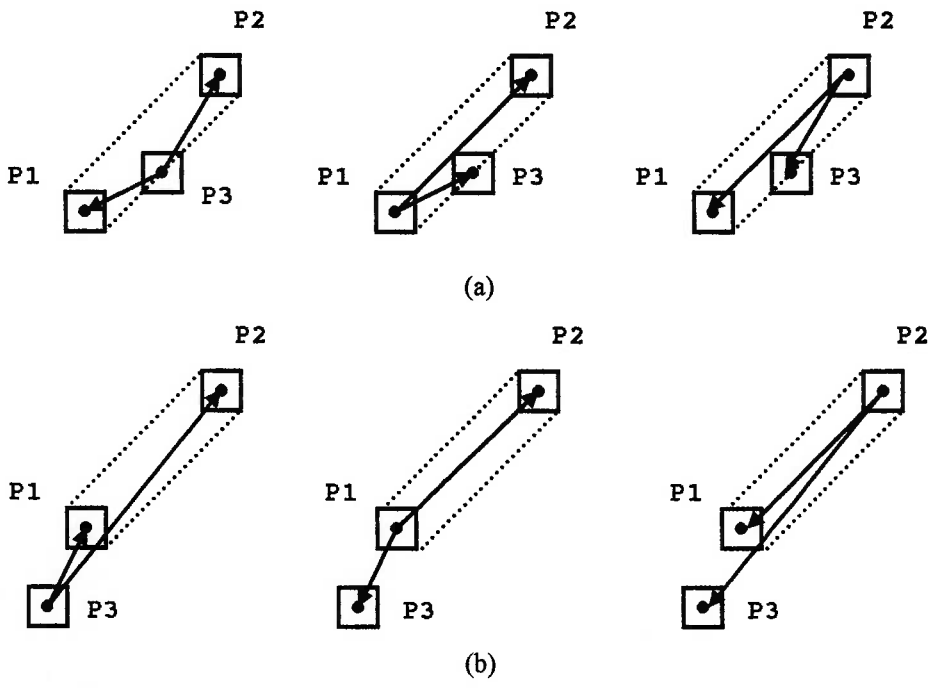


Fig. 3.18. a) Ponto **P3** incidente a linha intervalar **P1P2**; b) Ponto **P3** não incidente a linha intervalar **P1P2**.

Porém, a quantidade de cálculo envolvido é alta: cálculo de três vetores e três produtos vetoriais para determinar se os pontos são colineares e mais três produtos escalares para determinar a posição relativa de **P3** em relação a **P1** e **P2**.

$$A1' = [\ (P2 \ . \ P1) \ \hat{} \ (P3 \ . \ P1)\ ]$$
$$A2' = [\ (P1 \ . \ P2) \ \hat{} \ (P3 \ . \ P2)\ ]$$
$$A3' = [\ (P1 \ . \ P3) \ \hat{} \ (P2 \ . \ P3)\ ]$$

$$E1 = [\ (P2 - P1) \bullet (P3 - P1)\ ]$$
$$E2 = [\ (P1 \ . \ P2) \bullet (P3 \ . \ P2)\ ]$$
$$E3 = [\ (P1 \ . \ P3) \bullet (P2 \ . \ P3)\ ]$$

Na busca de novos métodos para verificar a incidência geométrica, notou-se que invertendo-se a ordem em que as operações eram feitas, poderia-se reduzir a quantidade de cálculos. Inicialmente, o método consistia em: produto vetorial seguido de produto escalar, era comprovado primeiro que os pontos eram colineares para depois verificar se o ponto **P3** estava sobre o segmento **P1 P2**. Agora, verifica-se primeiro os três produtos escalares para determinar a posição relativa de **P1**, **P2** e **P3** (entre os pontos ou afastados dos pontos), e em seguida, caso o ponto **P3** esteja certamente entre os pontos, e **P1** e **P2** sejam certamente extremos; verifica-se o produto vetorial para obter a área que pode ser possivelmente igual a zero. Ou seja, é aplicado o produto escalar com base **P3**, para determinar se **P3** está entre **P1** e **P2**. Em seguida, aplica-se o produto escalar com base **P1** para verificar se **P1** é extremo. O mesmo para **P2**. Se **P3** não é extremo, e **P1** e **P2** são, significa que pode-se verificar se eles são colineares com um único produto vetorial.

```
E1 = [ (P2 - P1) • (P3 - P1) ]
E2 = [ (P1 . P2) • (P3 . P2) ]
E3 = [ (P1 . P3) • (P2 . P3) ]
A3 = [ (P1 . P3) ^ (P2 . P3) ]
```

Na situação em que são utilizados pontos no espaço tridimensional, a área torna-se um valor composto por três coordenadas: **Ax**, **Ay** e **Az**. Naturalmente, a área é possivelmente igual a zero, se **Ax**, **Ay** e **Az** forem todos possivelmente iguais a zero.

3.6. Ponto e Plano

É analisado se um ponto **P4** pertence a um plano formado pelos pontos **P1**, **P2** e **P3**. Observando a metodologia aplicada ao caso de ponto e reta, pode-se utilizar o cálculo de volume para determinar se o ponto pertence ao plano. Se o volume for possivelmente igual a zero, o ponto pertence ao plano.

O volume é obtido pelo produto misto:

```
V = [ (P2 . P1) ^ ( P3 . P1) ] • (P4 . P1) /4
```

Com os resultados das técnicas empregadas no caso de incidência ponto e linha, aplica-se o método de múltiplos cálculos de volume, utilizando-se todos os pontos como ponto base de cálculo:

$$V1 = [(P2 . P1) \hat{} (P3 . P1)] \bullet (P4 . P1) /4$$
$$V2 = [(P1 . P2) \hat{} (P3 . P2)] \bullet (P4 . P2) /4$$
$$V3 = [(P1 . P3) \hat{} (P2 . P3)] \bullet (P4 . P3) /4$$
$$V4 = [(P1 . P4) \hat{} (P2 . P4)] \bullet (P3 . P4) /4$$

Deste modo, o ponto P4 está possivelmente no plano, se **V1**, **V2**, **V3** e **V4** forem possivelmente iguais a zero. Testes iniciais utilizando valores numéricos demonstram que esta técnica funciona. Se os quatro pontos não pertencerem possivelmente ao mesmo plano, um dos volumes é certamente (ou rigorosamente) não nulo. Facilmente nota-se o enorme custo computacional desta técnica: quatro produtos vetoriais, quatro produtos escalares e oito cálculos de vetores.

Outra metodologia, consiste em calcular a equação de face **n** para os pontos **P1**, **P2** e **P3**. Em seguida, aplica-se a equação para o ponto **P4** e se o valor é possivelmente igual a zero, então o ponto P4 está sobre o plano definido por **P1**, **P2** e **P3**.

$$n = (x, y, z, w)$$
$$P4 = (x_4, y_4, z_4)$$
$$x * x_4 + y * y_4 + z * z_4 + w = 0$$

No exemplo da Figura 3.19a é apresentado um plano e na Figura 3.19 b, um plano intervalar, que pode ser subdivido em dois planos **n1** e **n2**. A área hachurada entre estes dois planos está possivelmente sobre o plano intervalar **n**.

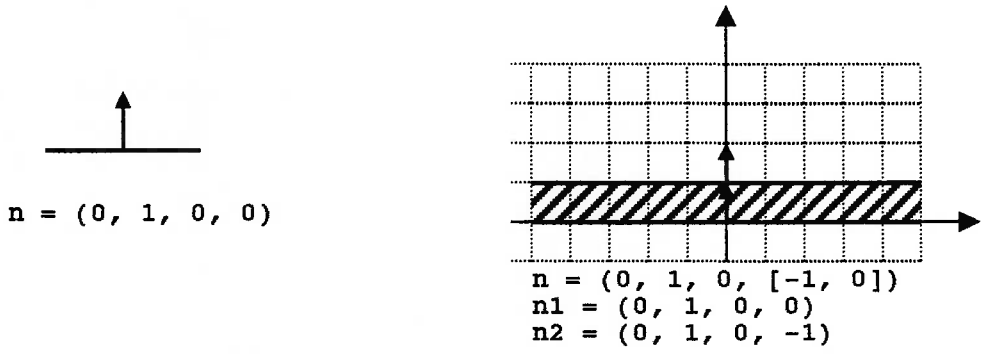
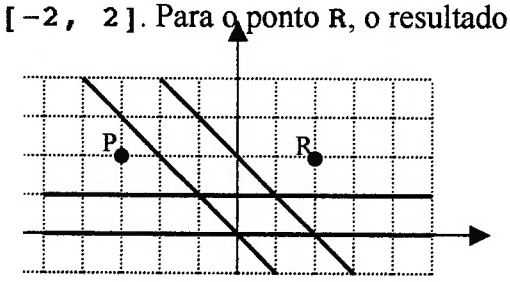


Fig. 3.19 a) Plano **n**; b) Plano Intervalar.

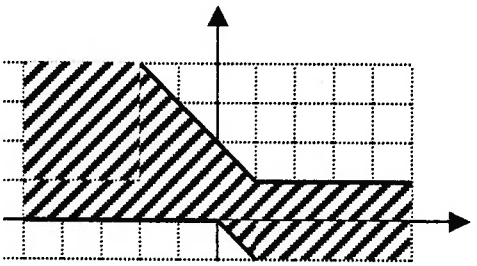
O próximo exemplo na Figura 3.20a possui um plano intervalar **n** formado pelos planos **n1**, **n2**, **n3** e **n4**. A área hachurada está possivelmente sobre o plano intervalar **n**. Na Figura 3.20b, o ponto **P = (-3, 2, 0)** está na área hachurada e

o teste indica que o valor é possivelmente igual a zero, no exemplo o resultado foi `[-2, 2]`. Para o ponto `R`, o resultado foi `[1, 4]`, não está sobre o plano `n`.




```
n = ([0, 1], 1, 0, [-1, 0])
n1 = (0, 1, 0, 0)
n2 = (0, 1, 0, -1)
n3 = (1, 1, 0, 0)
n4 = (1, 1, 0, -1)
```

(a)



```
n = ([0, 1], 1, 0, [-1, 0])
P = (-3, 2, 0)    R = (2, 2, 0)

Teste para o ponto P:
[0,1] * (-3) + 1 * 2 +
[-1, 0] * 0 + [-1, 0] = [-2, 2]
Teste para o ponto R:
[0, 1] * 2 + 1 *2 +
[-1, 0] * 0 + [-1, 0] = [1, 4]
```

(b)

Fig. 3.20 a) Plano intervalar formado por `n1`, `n2`, `n3` e `n4`; b) Exemplo de verificação do pontos `P` e `R` sobre o plano `n`.

Esta última técnica é a mais conveniente, pois já existe uma variável para armazenar a equação da face e é a que foi utilizada neste trabalho.

3.7. Correção para ponto de intersecção intervalar

Uma das situações em que pode surgir erro de aproximação é a determinação de pontos de intersecção entre elementos geométricos (exemplo: linha e face). E a determinação de pontos de intersecção é utilizada pela operação de corte e pelas operações booleanas.

O intervalo do ponto de intersecção calculado utilizando as operações aritméticas definidas em intervalos ultrapassam o limite da linha intervalar, conforme é ilustrado na Figura 3.21 e aumentam rapidamente conforme a quantidade de operações que são aplicadas ao sólido. Em operações que necessitam de pontos de intersecção, surgem falhas nos testes de incidência pois os intervalos estão largos e tornam inútil a adoção da aritmética intervalar para testes de incidência pois geram resultados incorretos.

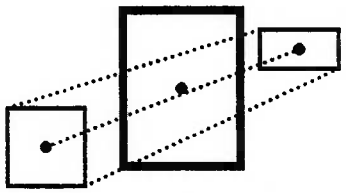


Fig. 3.21. Extremos do ponto de intersecção intervalar ultrapassam limites da linha intervalar.

Logo, deve ser feita uma correção para limitar os intervalos dos pontos de intersecção. Esta correção utiliza-se do parâmetro `t` e as linhas que ligam os extremos `x1, x2, y1, y2, z1, z2, x3, x4, y3, y4, z3` e `z4` dos intervalos dos vértices intervalares para determinar os limites `xa, xb, ya, yb, za, zb` do ponto de intersecção intervalar (Figura 3.22).

$$x_a = (1-t) * x_1 + t * x_3$$
$$x_b = (1-t) * x_2 + t * x_4$$
$$y_a = (1-t) * y_1 + t * y_3$$
$$y_b = (1-t) * y_2 + t * y_4$$
$$z_a = (1-t) * z_1 + t * z_3$$
$$z_b = (1-t) * z_2 + t * z_4$$

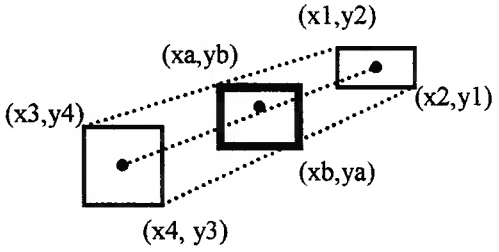


Fig. 3.22. Determinação dos extremos do ponto de intersecção.

Quando ocorre a formação de um novo segmento de reta, não deve ocorrer distorção da direção do novo segmento de reta intervalar. Caso contrário, pode propagar o aumento do intervalo e gerar pontos e linhas intervalares sem utilidade, conforme o exemplo na Figura 3.23.

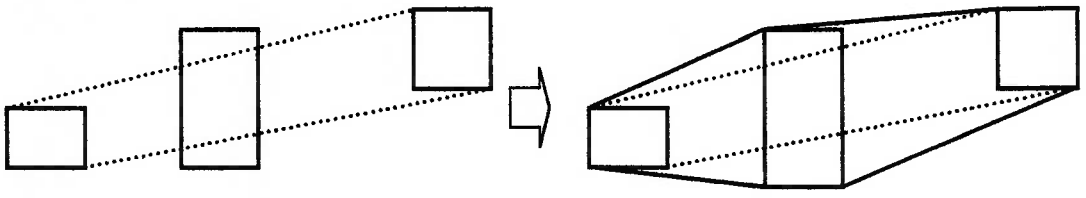


Fig. 3.23. Distorção do segmento de reta intervalar que foi dividido em dois segmentos de retas intervalares.

Neste trabalho a correção é realizada com base na geometria da linha intervalar.

4. MODELADOR DE SÓLIDOS

Nesta seção é apresentado um estudo sobre modeladores de sólidos B-rep e sobre a estrutura de dados utilizada na implementação. São apresentados os elementos hierárquicos necessários para representar geometricamente e topologicamente o sólido em forma computacional.

4.1. Estudo sobre modeladores de sólidos B-rep

Segundo Tsuzuki (1991), a arquitetura de um software modelador de sólidos possui três níveis de abstração:

- Nível Superior: neste nível, estão presentes as rotinas que definem as ferramentas disponíveis ao usuário, permitindo construir, modificar e armazenar os sólidos. Os operadores locais e operações booleanas compõem este nível.
- Nível Intermediário: onde são desenvolvidas ferramentas para implementar as ferramentas do nível superior. Este nível é principalmente composto pelos operadores de Euler, que estão presentes no Anexo A.
- Nível Inferior: onde é desenvolvida a estrutura de dados que forma uma representação apropriada para a manipulação computacional.

A representação B-rep armazena detalhes de como as faces, arestas e vértices se unem para representar um sólido. Um sólido modelado pela representação B-rep deve possuir, por exemplo, a capacidade de descrever como cada face está conectada às suas faces adjacentes, de maneira que um volume totalmente fechado seja definido. Em uma representação B-rep, a informação de adjacência entre elementos está disponível explicitamente, ou seja, não é necessário realizar nenhuma comparação numérica.

Esta informação de adjacência é, geralmente, referenciada informalmente como topologia do sólido modelado. As informações topológicas criam um conjunto de vigas, no qual as informações geométricas são apoiadas. As informações topológicas e as informações geométricas não podem ser tratadas independentemente, pois elas estão profundamente relacionadas. Entretanto, a

topologia é conseqüência da geometria e não vice-versa. A geometria representa, por exemplo, equações de faces e coordenadas de vértices.

A representação B-rep descreve um sólido por superfícies orientadas através de uma estrutura de dados composta por vértices, arestas e faces [HOFFMANN, 1989]. A convenção de orientação permite-nos determinar o interior e o exterior do sólido de forma não ambígua. As normais das faces apontam para onde não existe material. Como exemplo (Figura 4.1.), uma face com um furo é representada com uma borda externa com orientação anti-horária (orientação da normal apontando para "fora da folha", enquanto que o furo está com orientação horária, indicando normal para "dentro da folha").

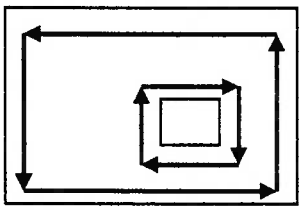


Fig. 4.1. Laço externo representando a borda da face e furo interno com orientação contrária .

Dentre as estruturas de dados, baseadas na aresta como elemento de referência, encontradas na literatura, destacam-se a "*winged-edge*" e a meia-aresta (MÄNTYLÄ, 1988; CHIYOKURA, 1988; TSUZUKI, 1991; LUTZ, 1998).

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. O conjunto de arestas adjacentes é dividido em dois grupos, cada um associado ao circuito de arestas ao redor das faces adjacentes à aresta de referência. Em outras palavras, a estrutura "*winged-edge*" possui ponteiros para as quatro arestas diretamente anteriores e posteriores à aresta de referência, nos ciclos de aresta que contornam as duas faces adjacentes à aresta de referência.

Na estrutura "*winged-edge*", as arestas assumem duas funções principais: dividir o contorno direcional das faces e definir a conectividade entre os elementos primitivos por meio de informações de adjacência da aresta de referência. Porém, do ponto de vista computacional, este é o ponto mais negativo da estrutura "*winged-edge*". Esta deficiência é clara, em particular, quando o circuito direcional de arestas de uma face deve ser obtido pelo procedimento que percorre seqüencialmente todas

as arestas que o compõem. A necessidade deste algoritmo surge com muita freqüência em operações gráficas e geométricas aplicadas ao sólido representado.

Percorrer o circuito de arestas de uma face é uma operação unidirecional, devido à própria orientação do circuito de arestas. Entretanto, na estrutura "*winged-edge*" cada aresta é utilizada para delimitar duas faces; portanto, cada aresta participa em dois circuitos orientados. Segundo a lei Möebius, os dois circuitos possuem orientações opostas entre si, portanto, cada aresta é percorrida exatamente duas vezes e em direções opostas. Em outras palavras, cada aresta pode ser percorrida em duas direções opostas, dependendo da face cujo circuito está sendo analisado. É necessário, entretanto, verificar e determinar a direção em que cada aresta está sendo referenciada a cada passo do percurso, um processo que aumenta consideravelmente o custo do procedimento.

Para resolver esta deficiência, a estrutura meia-aresta foi proposta; onde as duas principais funções da aresta foram separadas. 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.

Na estrutura meia-aresta, cada metade da aresta participa em apenas um circuito de arestas, portanto, cada metade possui apenas uma única orientação. Globalmente, cada aresta de referência é referenciada duas vezes em direções opostas pelos circuitos de arestas que contornam as duas faces adjacentes. A Figura 4.2 ilustra a representação da estrutura meia-aresta.

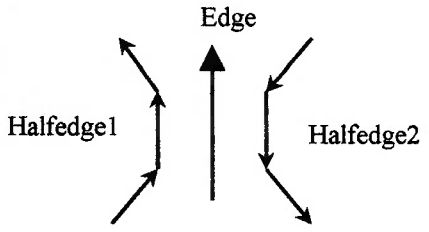


Fig. 4.2. Representação de uma "halfedge".

Chiyokura utilizou uma variação da representação "*winged-edge*", enquanto que Mäntylä utilizou a estrutura meia-aresta.

Segundo Chiyokura (1988), uma estrutura de dados do tipo B-rep possui as seguintes vantagens:

- como as arestas e as faces são representadas explicitamente, uma imagem do sólido tipo fio de arame ("*wireframe*") pode ser obtida rapidamente;
- existência de poucas restrições quanto às operações que podem ser implementadas e utilizadas.

E como desvantagens:

- a estrutura de dados é complexa, o que necessita de uma grande quantidade de memória. Procedimentos para manipulação dos dados internos podem ser complexos;
- dificuldades para modificar sólido já existentes;
- nem sempre, os dados internos representam um sólido válido.

A estrutura do sólido que será utilizada nesta implementação compreende os seguintes elementos:

- Sólido ("*Solid*"): representa o sólido que está sendo modelado; o sólido possui uma ou mais regiões;
- Região ("*Region*"): é possível que existam sólidos formados por conjuntos fechados de faces separados um do outro. Como exemplo, podemos imaginar a operação Booleana União aplicada sobre duas caixas que estão separadas (Figura 4.3) – o sólido resultante possui duas regiões. O elemento Região possui um ou mais "*shells*";

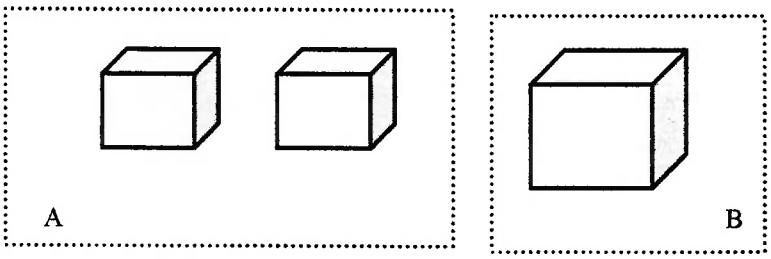


Fig. 4.3. Sólido A, formado por 2 regiões e Sólido B, formado por uma única região.

- "*Shell*": este elemento é um conjunto fechado de faces. Uma região é formada por um "*shell*" externo, e pode ter zero ou mais "*shells*" internos. Como exemplo (Figura 4.4), pode-se imaginar a Operação Booleana Subtração sendo realizada para subtrair uma caixa pequena de uma caixa grande, sem que nenhuma face da caixa menor intercepte alguma face da

caixa maior. É possível aplicar várias subtrações e obter um sólido com vazios internos ("*shells*" internos). Naturalmente, como as normais das faces apontam para onde não existe material, as normais das faces dos "*shells*" internos devem apontar para seu interior (vazio), e as normais das faces do "*shell*" externo apontam para o exterior. O elemento "*Shell*" é formado por faces;

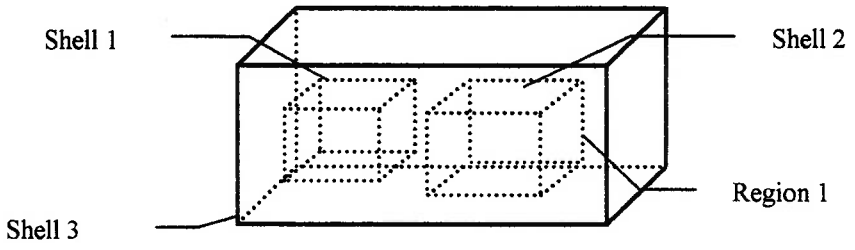


Fig. 4.4. A Região 1 é formado por 3 "shells": o externo ("shell" 3) e dois internos: "shell" 1 e 2.

- Face: elemento que delimita o material do sólido. Pode-se obter a normal da face através do cálculo da equação do plano no qual se encontra a face. A face é formada por laços;
- Laço ("*loop*"): é possível que uma face possua furos. Estes furos são modelados como laços internos. Logo, a face é formada por um laço externo e pode ter zero ou mais laços internos. A Figura 4.5. mostra uma face com dois laços internos. O laço é formado por uma seqüência de meia-arestas;

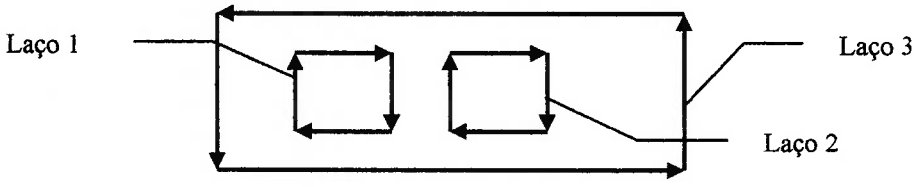


Fig. 4.5. Face formada por 3 laços: laço externo 3 e laços internos 1 e 2.

- Aresta ("*edge*"): este elemento possui informação sobre as meia-arestas que o compõem;

- Meia-aresta ("*Halfedge*"): este elemento possui um ponteiro para um vértice e para a meia-aresta seguinte e a anterior que definem a seqüência de meia-arestas do laço;
- Vértice ("*Vertex*"): elemento que possui uma tripla de coordenadas (x, y, z).

4.2. Estrutura de dados

O modelador de sólidos possui como núcleo a sua estrutura de dados, e deseja-se comparar um modelador de sólidos convencional implementado em aritmética de ponto flutuante e um modelador de sólidos utilizando aritmética intervalar. Para que não fossem implementados dois modeladores de sólidos, foram utilizados "*templates*" e o STL ("*Standard Template Library*" – Ammeraal, 1996) que permitem parametrizar o conteúdo das estruturas de dados (principalmente lista ligada). O uso da aritmética intervalar visa diminuir o erro causado por problemas de precisão numérica (aritmética de ponto flutuante) e instabilidades de cálculos – como os apresentados por Hoffman (1989).

O protótipo foi implementado utilizando o OpenGL como saída gráfica (Woo, 1998). Foram utilizadas algumas ferramentas desenvolvidas para o OpenGL como GLUT ("OpenGL Utility Toolkit") de Kilgard (2000) e o GLUI ("User Interface Library to OpenGL") de Rademacher (1999).

Uma nova estrutura (apresentada na Figura 4.6), baseada na estrutura meia-aresta, descrita por Mäntylä (1988), foi implementada. Uma breve descrição de cada elemento é apresentada a seguir.

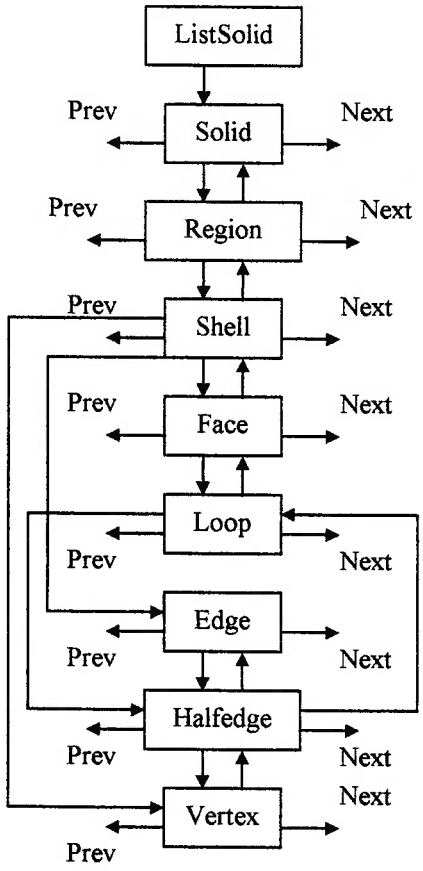


Fig. 4.6. Hierarquia dos elementos da estrutura de dados.

A classe **`ListSolid`** é a que contém a lista dos sólidos. Por existir apenas uma lista de sólidos no modelador de sólidos, esta classe foi implementada como um objeto "*singleton*" de Gabrilovich (1999). O "*singleton*" é um objeto que pode ser instanciado apenas uma vez. Não é possível existir a cópia de um "*singleton*".

```
template <class T>
class TlistSolid {
public:
        * get (identificador)          // fornecido o identificador, é fornecido o
                                       ponteiro para o Elemento (ex.: para Face, usa-
                                       se getFace )
        void addSolid(TSolid<T>)          // adiciona sólido na lista de sólidos
        void delSolid(TSolid<T> ou int)  // remove sólido da lista
        list<TSolid<T> >::iterator lstart_it(void) // retorna primeiro sólido da
lista
        list<TSolid<T> >::iterator lend_it(void)   // retorna último sólido da lista
private:
        list< Tsolid<T> > lsolid;  // lista de sólidos
}
```

A classe **`Solid`** foi implementada de modo convencional, pois podem existir várias instâncias dela e ela não possui ponteiro de retorno para o elemento hierárquico superior, pois só existe uma instância da **`ListSolid`**.

```
template <class T>
class TSolid {
public:
        TSolid()              // constructor
        ~TSolid()             // destructor
        * get (identificador)          // retorna ponteiro
        void addRegion (TRegion<T>)    // adiciona "regions" na lista de regions do
sólido
        void delRegion (TRegion<T>)       // remove regions da lista
        void listRegion (void)            // mostra conteúdo da lista
        TRegion<T> * getRegion (int id)   // fornecido o id, é fornecido o ponteiro
        list<TRegion<T> >::iterator SRegions_it (void) // retorna primeira region
da lista
        list<TRegion<T> >::iterator Sregiend_it (void)  // retorna última region da
lista
private:
        list< TRegion<T> > *sregions      // lista de regions
        int solidno                       // identificador do sólido
}
```

A classe **Region** possui implementação semelhante a da classe **Solid** com um método para retornar o ponteiro de retorno para o sólido que possui o **Region** e um outro ponteiro para o **Shell** externo.

```
template <class T>
class TRegion {
public:
        TRegion()             // constructor
        ~TRegion()            // destructor
        void addShell(TShell<T> )     // adiciona . shells. na lista de . shells.
da região
        void delShell(TShell<T> )     // remove . shells. da lista
        TSolid<T> *RSolid(void)       // retorna ponteiro para o sólido que possui
esta região
        list <TShell<T> >::iterator RShell_it()      // retorna primeiro . iterator.
da lista
        list <TShell<T> >::iterator RShend_it()      // retorna último da lista
        void RSout(TShell<T> )               // seleciona . shell. externo
        TShell<T> * RSout(void)              // retorna ponteiro para o . shell.
externo
private:
        TSolid<T> *rsolids;               // ponteiro de retorno para o sólido
        list<TShell<T> > *rshells;        // lista de . shells.
        int regionno;                     // identificador da região
        TShell<T> *rsout;                 // ponteiro para o . shell. externo
}
```

Os membros hierárquicos da classe **Shell**: face, aresta, vértice, aresta e meia-aresta podem ser representados separadamente do restante da estrutura de dados, conforme a Figura 4.7.

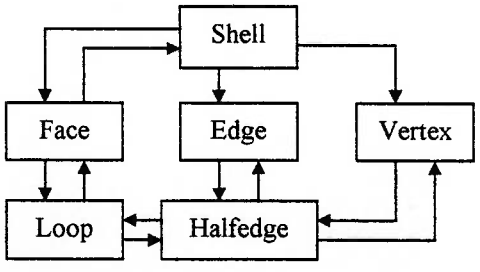


Fig. 4.7. Representação da implementação.

Logo, a classe **Shell** tem o seguinte formato de implementação:

```
template <class T>
class Tshell {
public:
        TShell ()                       // constructor
        ~TShell ()                      // destructor
        void addFace (TFace<T>)
        void addEdge (TEdge<T>)
        void addVertex (TVertex<T>)   // adiciona elementos na lista do Shell
        void delFace (TFace<T>)
        void delEdge (TEdge<T>)
        void delVertex (TVertex<T>)   // remove elementos da lista
        TRegion<T> * SRegions (void)          // retorna ponteiro de retorno para o
Region.
        list<TFace<T> >::iterator SFace_it(void)
        list<TEdge<T> >::iterator SEdge_it(void)
        list<TVertex<T> >::iterator SVert_it(void) // retorna primeiro elemento da
lista
        list<TFace<T> >::iterator SFend_it(void)
        list<TEdge<T> >::iterator SEend_it(void)
        list<TVertex<T> >::iterator SVend_it(void) // retorna último elemento da
lista

private:
        Tregion<T> *sregions            // ponteiro de retorno para o Region
        list <TFace<T> > *sfaces        // lista de faces
        list <TEdge<T> > *sedges        // lista de edges
        list <TVertex<T> > *sverts      // lista de vertex
        int shellno                     // identificador do shell
}
```

A classe **face** possui uma lista de laços e um ponteiro específico para indicar qual o laço externo que delimita a face. Ela está descrita abaixo:

```
template <class T>
class TFace {
public:
        TFace()                 // constructor
        ~TFace()                // destructor
        TLoop<T>* getLoop(),
        TEdge<T>* getEdge(),
        THalfedge<T>* getHalfedge() // retorna ponteiro para o elemento especificado
        void addLoop (TLoop<T>)       // adiciona um laço na lista
        void delLoop (TLoop<T>)       // removece um laço da lista
        Tshell<T>* Fshell(void)       // retorna ponteiro de retorno para o shell
        TLoop<T>* Flout (void)        // retorna o ponteiro para o laço externo
        list<TLoop<T> >::iterator FLoops_it(void)  // retorna primeiro laço da lista
        list<TLoop<T> >::iterator FLend_it (void)  // retorna último laço da lista
private:
        list<TLoop<T> > *floops;      // ponteiro para a lista de laço
        TShell<T> *fshell;             // ponteiro de retorno para o shell
        TLoop<T> *flout;              // ponteiro para o laço externo
        tnVector<T,4> feq;            // equação da face
        int faceno;                   // identificador da face
}
```

A classe **loop** possui um ponteiro de retorno para a face e um para uma meia-aresta, que é início de um ciclo de meia-arestas que formam o laço.

```
template <class T>
class TLoop {
public:
        TLoop()                 // constructor
        ~TLoop()                // destructor
        TFace<T>* Lface(void)         // retorna ponteiro de retorno para a face
        TEdge<T>* LEdg(void)          // retorna ponteiro para . edge.
private:
        TFace<T> *lface;              // ponteiro de retorno para a . face.
        THalfedge<T> *ledg;    // ponteiro para . halfedge.  que compõe o laço
        int loopno;            // identificador do laço
        int length;            // comprimento do laço
}
```

A classe **Edge** possui seu identificador e ponteiros para duas meia-arestas.

```
template <class T>
class TEdge {
public:
        TEdge()                         // constructor
        ~TEdge()                 // destructor
        THalfedge<T> *He1(void)         // retorna ponteiro para a primeira meia-aresta
        void He1(THalfedge<T> *h)       // seleciona ponteiro como sendo primeira meia-
aresta
        THalfedge<T> *He2(void)         // retorna ponteiro para a segunda meia-aresta
        void He2(THalfedge<T> *h)       // seleciona ponteiro como sendo segunda meia-
aresta
private:
        THalfedge<T> *he1;      // ponteiro para a primeira meia-aresta
        THalfedge<T> *he2;      // ponteiro para a segunda meia-aresta
        int edgeno;             // identificador da aresta
}
```

A classe **Halfedge** não é incluída em nenhuma lista, sendo assim, necessita de ponteiros para a próxima meia-aresta e para a meia-aresta anterior. Possui o seguinte formato:

```
template <class T>
class THalfedge {
public:
        THalfedge()             // constructor
        ~THalfedge()            // destructor
        THalfedge<T>* mate(void)        // retorna a outra meia-aresta que, em conjunto
com esta, formam uma . edge.
        TLoop<T>* Wloop()       // retorna o ponteiro para o laço
        TEdge<T>* Edg()                 // retorna o ponteiro para o . edge.
        THalfedge<T>* Nxt()     // retorna o ponteiro para a próxima meia-aresta
        THalfedge<T>* Prv()     // retorna o ponteiro para a anterior meia-aresta
        TVertex<T>* Vtx()       // retorna o ponteiro para o . vertex.
private:
      TLoop<T> *wloop;                  // ponteiro para o laço
      TEdge<T> *edg;                    // ponteiro para o . edge.
      THalfedge<T> *nxt;                // ponteiro para a próxima meia-aresta
      THalfedge<T> *prv;                // ponteiro para a meia-aresta anterior
      TVertex<T> *vtx;                  // ponteiro para o . vertex.
      int halfedgeno;                   // identificador da meia-aresta
}
```

A classe **Vertex** possui a coordenada do vértice **(x, y, z, w)**, seu identificador e um ponteiro para uma meia-aresta.

```
template <class T>
class TVertex {
public:
        TVertex()                       // constructor
        ~TVertex()                      // destructor
        tnVector<T,4> VCoord(void)              // retorna as coordenadas do vértice
        void VCoord(const tnVector<T,4> )       // seleciona as coordenadas para este
vértice
        THalfedge<T> *VEdge(void)               // retorna um ponteiro para uma meia-
aresta que utiliza este vértice
        void VEdge(THalfedge<T> )               // seleciona um ponteiro para meia-
aresta
private:
        THalfedge<T> *vedge;            // ponteiro para uma meia-aresta que utiliza
este vértice
        tnVector<T,4> vcoord;           // coordenadas (x, y, z, w) deste vértice
        int vertexno;                   // identificador do vértice
}
```

5. OPERAÇÕES NO MODELADOR DE SÓLIDOS

Uma vez construídos os operadores de Euler (nível intermediário), é possível começar a construir sólidos de modo coordenado. Entretanto, o trabalho necessário é muito grande. Logo, surge a necessidade dos operadores de alto nível: a operação de corte de sólido ("*CutSolid*") por um plano e as operações booleanas.

É possível observar que a operação de corte de sólido é apenas um passo intermediário para se chegar nas operações booleanas, pois é possível realizar o corte de um sólido com a operação booleana subtração ou intersecção. Em modeladores de sólido B-rep, as operações booleanas são as mais utilizadas e com grande exigência de robustez.

Existem três tipos de operações booleanas: união, subtração (ou diferença) e intersecção. Entretanto, Chiyokura (1988) mostra que é necessária a implementação de apenas um tipo de operação booleana, pois as outras duas podem ser obtidas a partir da operação que foi implementada utilizando o conceito de sólido negativo (indicado como "NS" – "*Negative Solid*"). Logo, dados dois sólidos: **A** e **B**, as operações booleanas podem ser expressas como:

União (operador básico): $\mathbf{A} \cup \mathbf{B}$

Subtração: $\mathbf{A} \,.\, \mathbf{B} = \mathbf{NS}(\, \mathbf{NS}(\mathbf{A}) \cup \mathbf{B} \,)$

Intersecção: $\mathbf{A} \cap \mathbf{B} = \mathbf{NS}(\, \mathbf{NS}(\mathbf{A}) \cup \mathbf{NS}(\mathbf{B}) \,)$

Subtração (operador básico): $\mathbf{A} \,.\, \mathbf{B}$

União: $\mathbf{A} \cup \mathbf{B} = \mathbf{NS}(\, \mathbf{NS}(\mathbf{A}) \,.\, \mathbf{B} \,)$

Intersecção: $\mathbf{A} \cap \mathbf{B} = \mathbf{A} \,.\, \mathbf{NS}(\mathbf{B})$

Intersecção (operador básico): $\mathbf{A} \cap \mathbf{B}$

União: $\mathbf{A} \cup \mathbf{B} = \mathbf{NS}(\, \mathbf{NS}(\mathbf{A}) \cap \mathbf{NS}(\mathbf{B}) \,)$

Subtração: $\mathbf{A} \,.\, \mathbf{B} = \mathbf{A} \cap \mathbf{NS}(\mathbf{B})$

Um sólido negativo tem volume negativo. Logo, implementar uma função para gerar sólido negativo requer apenas inverter a orientação das normais das faces do sólido. Para inverter as normais das faces, é necessário inverter a orientação do

conjunto de meia-arestas que formam a face. Nesta implementação, escolheu-se como operador básico a operação booleana união.

Segundo Mäntylä (1988), os principais problemas que um conjunto de Operações Booleanas podem sofrer são: falta de robustez para lidar com todas as possibilidades de intersecção que podem ocorrer entre os elementos geométricos (faces, arestas e vértices); possibilidade de erro de precisão numérica durante os testes de incidência e determinação de intersecção. É possível solucionar o primeiro problema usando uma extensa análise de casos e situações, realizando testes e aplicando correções. Para o segundo problema, a aritmética intervalar arredondada deverá ajudar a manter a consistência topológica impedindo que o erro de aproximação interfira gravemente na topologia do sólido.

5.1. Algoritmo da operação de corte

Nesta seção é apresentada uma versão simplificada do algoritmo de corte. A versão completa pode ser encontrada no Anexo B. O algoritmo da operação de corte pode ser dividido nas seguintes etapas, para um sólido **A** e um plano de corte **n**:

a) Determinação dos pontos de intersecção das faces do sólido **A** com o plano de corte **n**. Nesta etapa, a correção para arimética intervalar é implementada no momento em que os pontos de intersecção são determinados. Deste modo, se outras operações forem aplicadas a este sólido, a propagação do aumento dos intervalos dos vértices intervalares será diminuída (vide Figura 5.1);

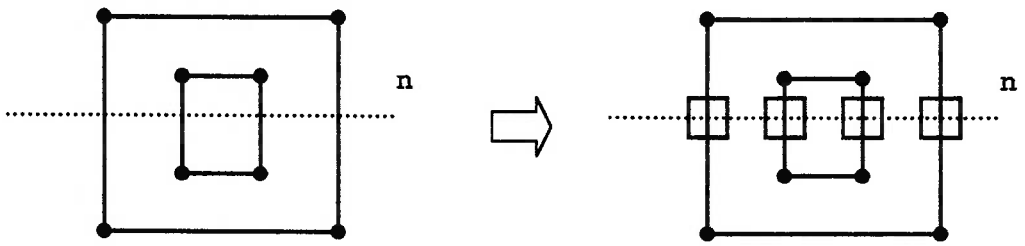


Fig. 5.1. Determinação dos pontos de intersecção entre o plano de corte e as faces do sólido.

b) Os pontos de intersecção determinados na etapa são transformados em vértices na estrutura de dados. Naturalmente, arestas e faces também são inseridas na estrutura (vide Figura 5.2);

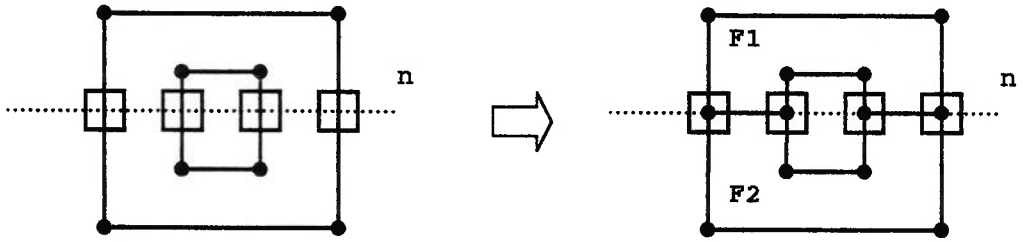


Fig. 5.2. Adição de vértices, arestas e faces na estrutura de dados.

c) As faces que estão abaixo do plano de corte n são inseridos numa lista para que, em seguida, sejam removidas (vide Figura 5.3).

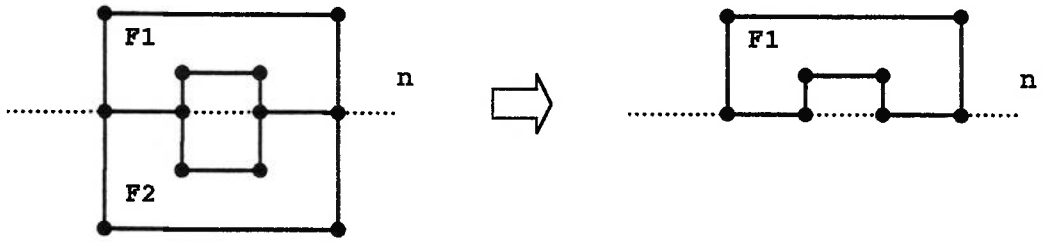


Fig. 5.3. Eliminação das faces abaixo do plano de corte.

d) Após a remoção das faces, ajustes e uma limpeza na estrutura de dados são realizados como etapa final no algoritmo de corte.

5.2. Algoritmo de operações booleanas

Nesta seção é apresentada uma versão simplificada do algoritmo de operações booleanas. A versão completada pode ser encontrada no Anexo C. O algoritmo de operação booleana união pode ser dividido nas seguintes etapas, para dois sólidos **A** e **B**:

a) Determinação dos pontos de intersecção das faces do sólido **A** com as faces do sólido **B**. Nesta etapa, a correção para arimética intervalar é implementada no momento em que os pontos de intersecção são determinados. Deste modo, se outras operações forem aplicadas a este

sólido, a propagação do aumento dos intervalos dos vértices intervalares será diminuída (vide Figura 5.4);

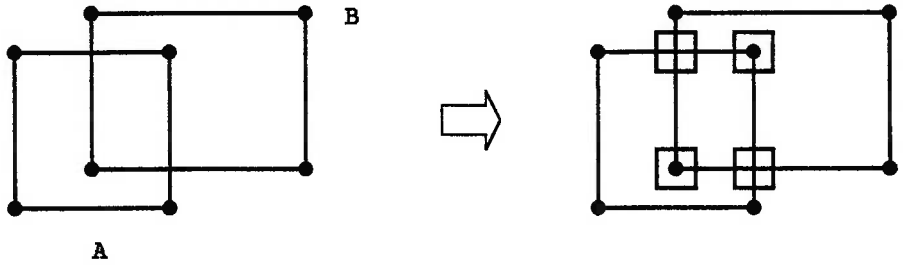


Fig. 5.4. Determinação dos pontos de intersecção entre as faces dos sólidos **A** e **B**.

b) Os pontos de intersecção determinados na etapa são transformados em vértices na estrutura de dados. Naturalmente, arestas e faces também são inseridas na estrutura (vide Figura 5.5);

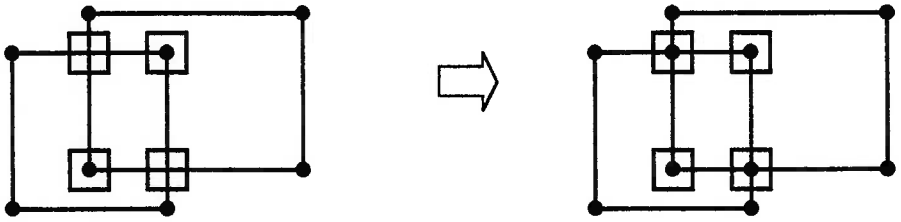


Fig. 5.5. Adição de vértices, arestas e faces na estrutura de dados dos sólidos **A** e **B**.

c) Nesta etapa, as faces dos sólidos **A** e **B** devem ser classificadas para determinar quais devem ser removidas. Neste trabalho, a implementação utilizou uma classificação baseada no trabalho de Chiyokura (1988) (vide Figura 5.6);

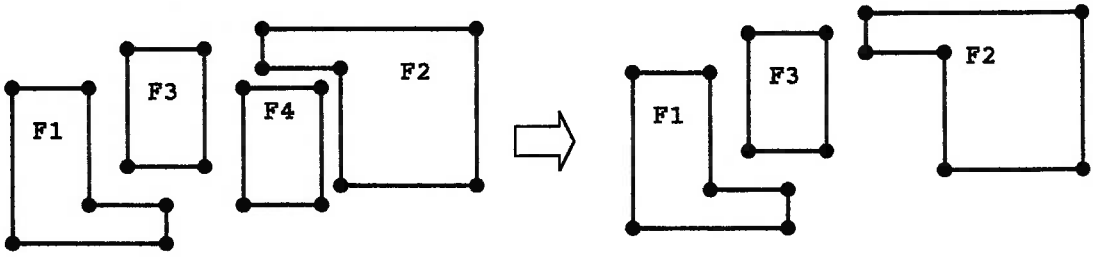


Fig. 5.6. Classificação e remoção das faces internas ao sólido resultante.

d) Após a remoção das faces, ajustes e limpeza da estrutura de dados dos sólidos A e B são realizados para que a etapa final de colagem seja possível. É necessário que as partes em que ocorrem a colagem tenham

topologia idêntica, caso contrário, a colagem dos sólidos não conseguirá ser feita;

e) Esta é a última etapa, em que o sólido B é colado ao sólido A. O que significa que a estrutura de dados do sólido B é transferido para a estrutura de dados do sólido A e as regiões de colagem são removidas da estrutura de dados pois estão internos ao sólido resultante (Figura 5.7).

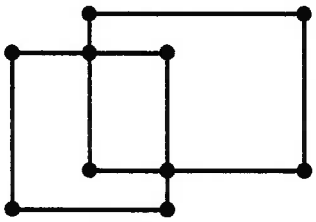


Fig. 5.7. Colagem do sólido **A** com o sólido **B**.

Nas etapas a e b, a aritmética intervalar tem fundamental importância, para impedir que vértices, arestas e faces desnecessários sejam criados, pois elementos desnecessários podem, por acidente, alterar a topologia do sólido e este problema pode ser propagado para os outros elementos geométricos.

Na literatura, o que difere de uma implementação de operação booleana de outra, é o conceito envolvido para realizar a etapa c – classificação. Em geral, as operações booleanas em modeladores B-rep consistem basicamente em determinar pontos de intersecção, criar novos vértices, remover elementos desnecessários e juntar a parte restante do sólido **A** com o restante do sólido **B**. Mas, determinar o que deve ser removido requer a análise local do sólido. Devemos classificar os elementos em: pode ser removido e não pode ser removido da estrutura de dados. A ênfase é para não remover, ou seja, elementos geométricos redundantes são permitidos em etapas intermediárias da operação booleana, pois a etapa final consiste em limpar a estrutura de dados, removendo-os.

A implementação de Mäntylä (1988), utilizava uma classificação baseada na vizinhança dos vértices. Enquanto que a de Chiyokura (1988) e Glaeser (1998), que foi escolhida para este trabalho por ser de mais fácil compreensão e implementação, são baseadas nos diedros que formam a aresta de intersecção.

A versão anterior do USPDesigner é baseada no trabalho de Mäntylä (1988). A experiência adquirida com a implementação e estudo do modelador de sólidos com base no trabalho de Mäntylä (1988), que possui listagens do código em linguagem C, permitiram detectar falhas e falta de consistência no algoritmo de operações booleanas. Assim, neste trabalho procurou-se compreender melhor a proposta de Chiyokura (1988) que não possui listagens ou código apresentados.

6. RESULTADOS

Nesta seção são apresentados dois tipos de resultados: alguns sólidos construídos com o modelador de sólidos USPDesigner e comparações entre os resultados de sólidos em aritmética de ponto flutuante com tolerância fixa e aritmética intervalar. Na Figura 6.1 está uma tomada de tela que mostra a interface gráfica, formada por duas janelas: a principal com o título do programa e a auxiliar.

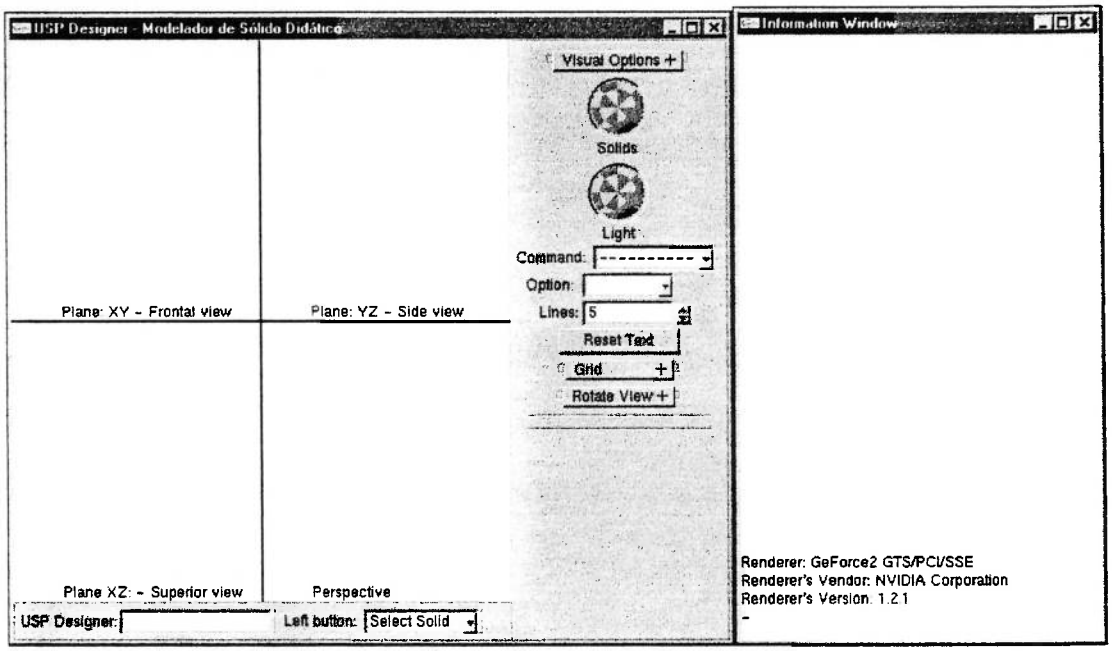
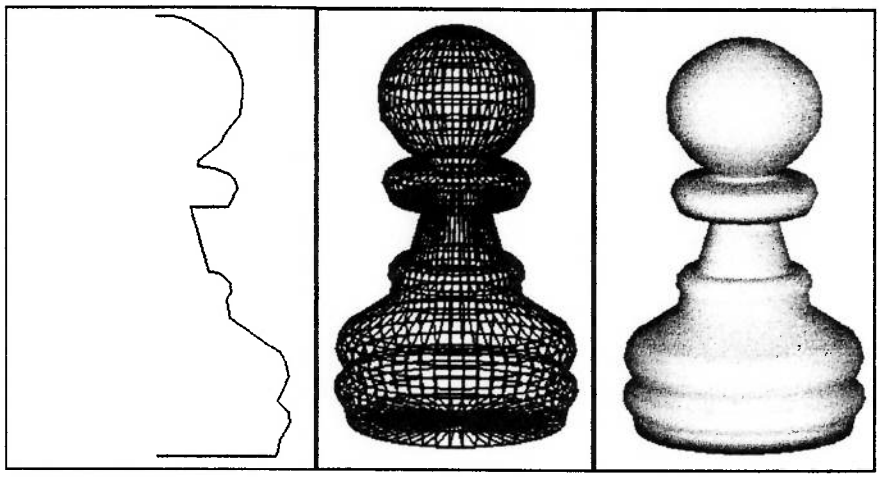


Fig. 6.1. Interface gráfica do Modelador de sólidos.

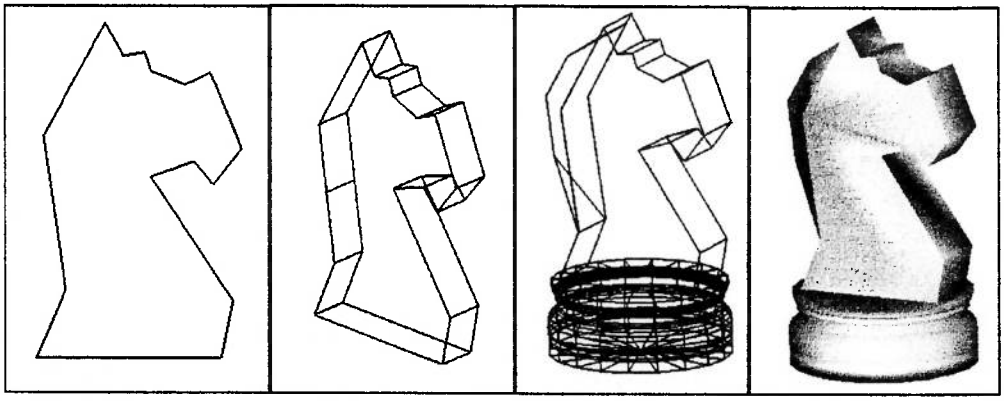
Uma ferramenta utilizada que pode ser utilizada na construção de sólidos, é a definição do sólido por revolução. A Figura 6.2a ilustra o perfil de uma peão de jogo de xadrez que foi utilizado para criar o sólido da Figura 6.2b (2000 faces, 3950 arestas, 1952 vértices). A Figura 6.2c apresenta o modelo sólido com iluminação do OpenGL e visualização "*shading*".



(a) (b) (c)

Fig. 6.2 a) Perfil de uma peão; b) modelo "*wireframe*" c) modelo "*shading*".

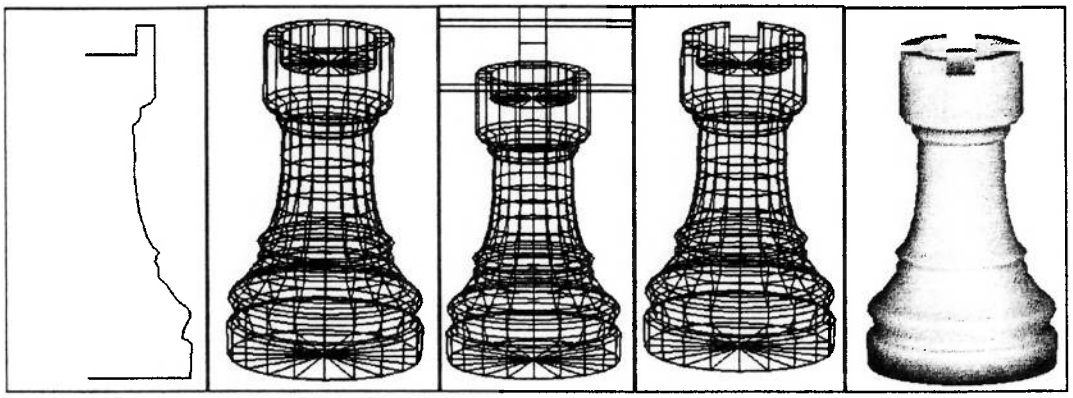
O próximo exemplo utiliza a ferramenta para encompridar o perfil. Na Figura 6.3a temos o perfil de um cavalo de xadrez. Utilizando-se o comando **`extrude`**, é obtido o sólido da Figura 6.3b. Com o perfil da base da peça e com o auxílio do comando de revolução de perfis, o cavalo é completado como mostra a Figura 6.3c, onde a operação booleana união junta a base com o perfil do cavalo.



(a) (b) (c) (d)

Fig. 6.3 a) Perfil da peça cavalo; b) Extrusão do perfil c) Modelo "*wireframe*"; d) Modelo "*shading*".

A próxima peça de xadrez é a torre. Para a sua construção, é necessário o uso da operação booleana de subtração para criar os detalhes da parte superior da peça com o auxílio de dois blocos, conforme é apresentado na Figura 6.4e.



(a) (b) (c) (d) (e)

Fig. 6.4 a) Perfil da peça torre; b) revolução do perfil; c) blocos em posição para aplicar operação booleana subração; d) modelo "*wireframe*"; e) modelo "*shading*".

De modo semelhante, as outras peças de xadrez podem ser construídas: o bispo (Figuras 6.5 a e b) necessita da operação booleana subtração para formar o chanfro; a rainha é definida com o uso de revolução de perfil (Figuras 6.5 c e d); o rei necessita da operação booleana de união para juntar a base com a coroa (Figuras 6.5 e e f).

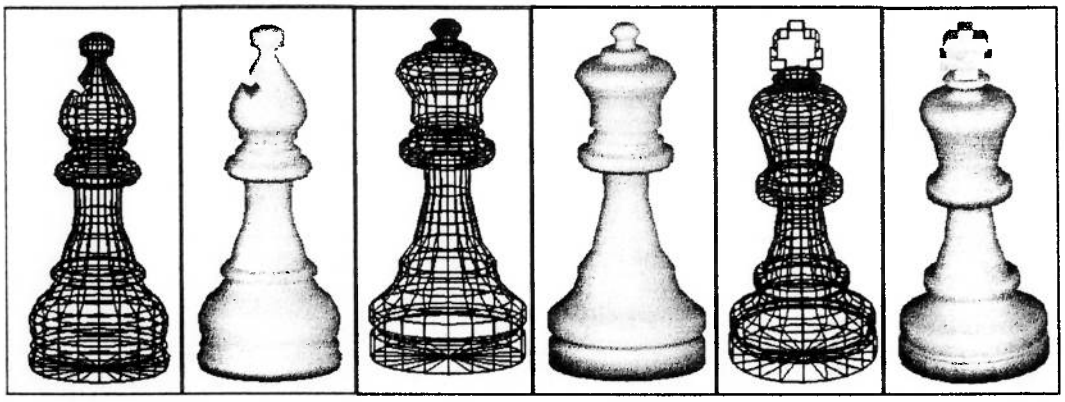


Fig. 6.5 a) Bispo – modelo "*wireframe*"; b) Bispo – modelo "*shading*"; c) Rainha – modelo "*wireframe*"; d) Rainha – modelo "*shading*"; e) Rei – modelo "*wireframe*"; f) modelo "*shading*".

No exemplo da Figura 6.6, foram aplicadas várias operações booleanas de subtração ao sólido.

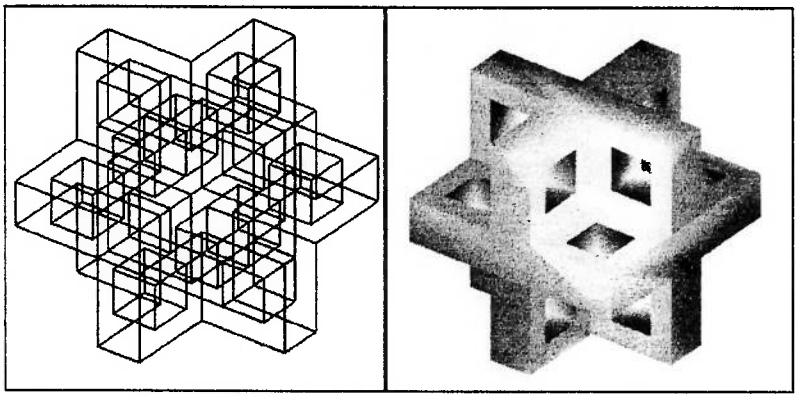
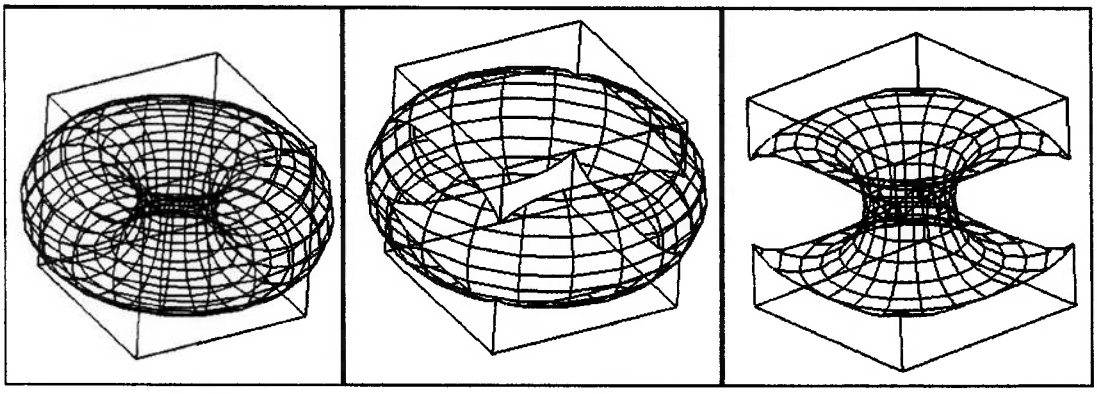
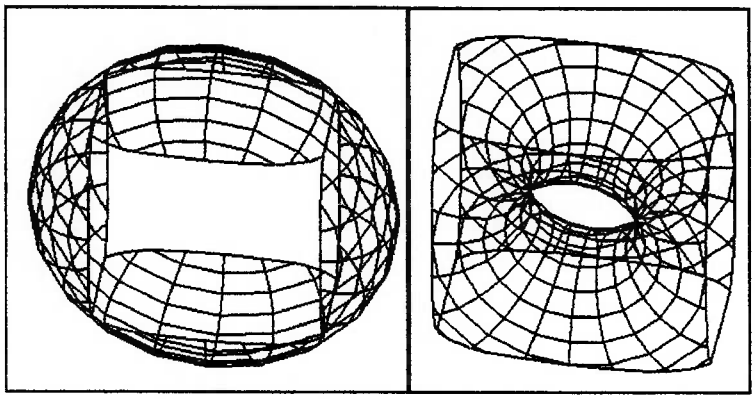


Fig. 6.6 a) modelo "*wireframe*"; b) modelo "*shading*".

Os próximos exemplos focam as operações booleanas. Na Figura 6.7a são apresentados dois sólidos **A** e **B**. Na Figura 6.7b é apresentada a união entre os sólidos **A** e **B**; na Figura 6.7c é apresentada a subtração do sólido **A** de **B**; na Figura 6.7d é apresentada a subtração do sólido **B** de **A** e na Figura 6.7e é apresentada a intersecção entre o sólido **A** e **B**.



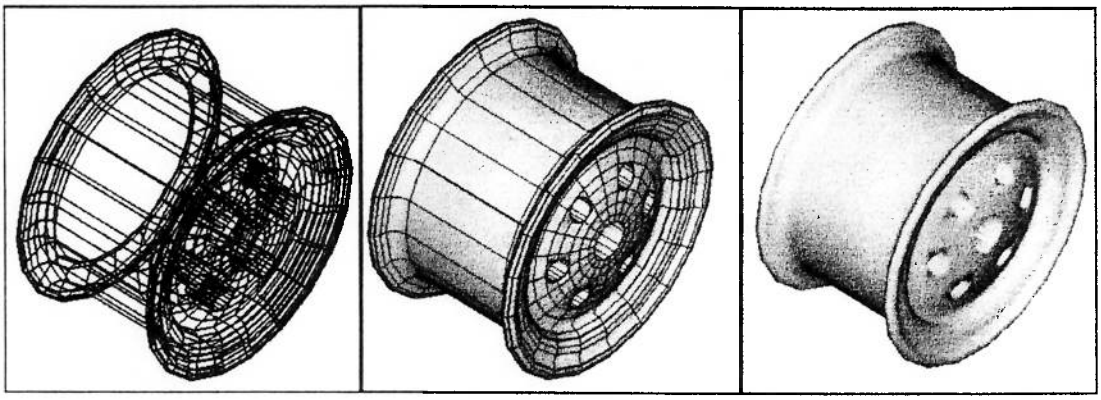
(a) (b) (c)



(d) (e)

Fig. 6.7 a) sólidos **A** e **B**; b) união dos sólidos **A** e **B**; c) subtração do sólido **A** de **B**; d) subtração do sólido **B** de **A**; e) intersecção de **A** e **B**.

O próximo exemplo é apresentado na Figura 6.8. É realizada a revolução de um perfil e, em seguida, utiliza-se a operação booleana de subtração para fazer os furos passantes no sólido.



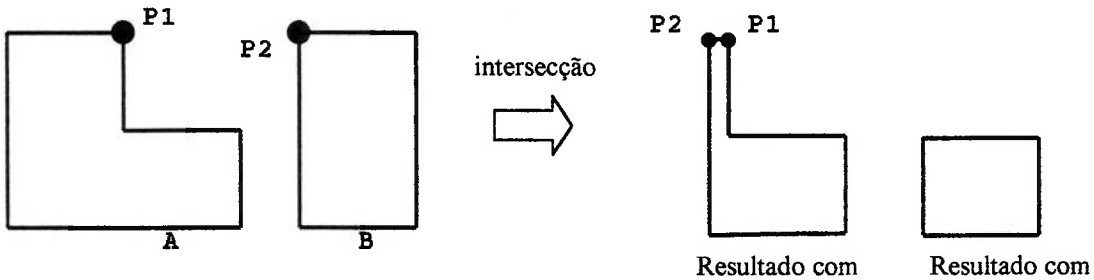
(a) (b) (c)

Fig. 6.8 a)Modelo "*wireframe*"; b) Modelo "*wireframe*" e "*shading*"; c) Modelo "*shading*".

Após esta série de exemplos demonstrando a capacidade de realizar operações com sólidos, os próximos exemplos destacam as situações em que os testes de incidência em aritmética intervalar tem resultados mais consistentes do que os testes de incidência em aritmética em ponto flutuante. Logo, para os próximos exemplos deve-se supor que uma série de operações foram realizadas anteriormente, de modo que as coordenadas do vértice possuem um erro de aproximação alto e a tolerância é de 0.001 para as comparações em ponto flutuante.

Figuras esquemáticas são empregadas no lugar de tomadas de tela por demontrarem melhor as comparações numéricas que são realizadas em aritmética de ponto flutuante e em arimética intervalar.

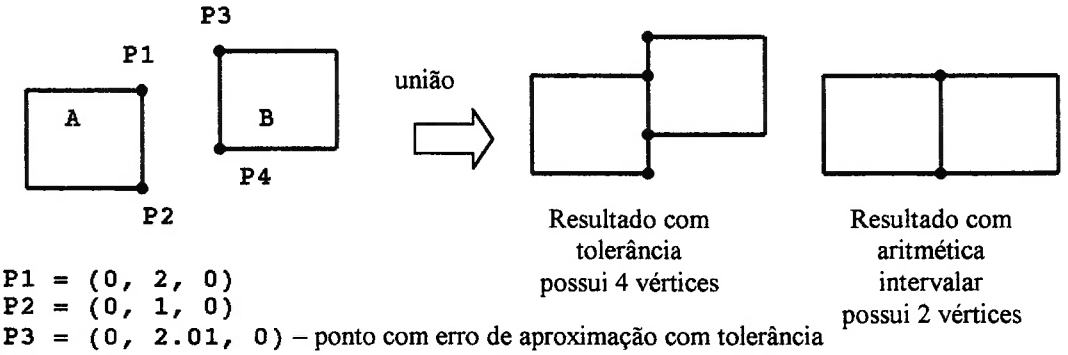
O primeiro exemplo desta série é mostrado na Figura 6.9, em que é aplicado a operação booleana de intersecção entre os sólidos **A** e **B**. O resultado utilizando tolerância em ponto flutuante possui uma prolongação, que foi devido à diferença entre P1 e P2 ser maior que a tolerância. Já o resultado em aritmética intervalar, não possui esta prolongação, já que o erro de aproximação alargou o intervalo para valores acima e abaixo, o que possibilita a igualdade.



P1 = (0, 0, 0)
P2 = (-0.01, 0, 0) – ponto com erro de aproximação com tolerância
P2 = ([-0.01, +0.01], 0, 0) – ponto com erro de aproximação com aritmética intervalar

Fig. 6.9. Comparação de resultados entre aritmética em ponto flutuante com tolerância e aritmética intervalar.

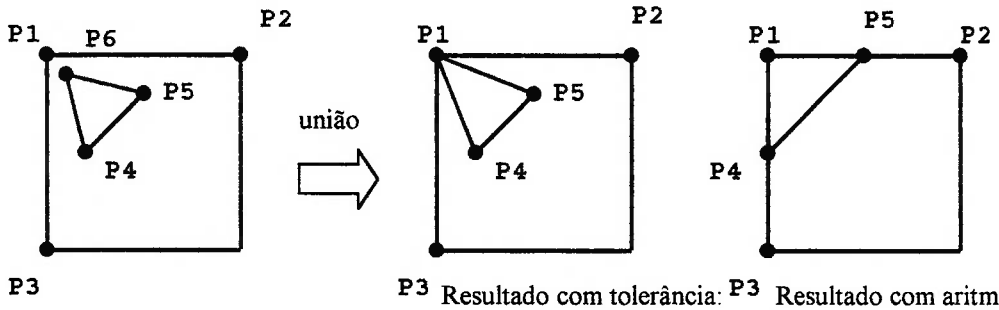
O próximo exemplo na Figura 6.10 mostra a aplicação da operação booleana união com os sólidos **A** e **B**. O resultado utilizando tolerância possui 4 vértices na região de colagem, enquanto que o resultado em aritmética intervalar possui 2 vértices.



P1 = (0, 2, 0)
P2 = (0, 1, 0)
P3 = (0, 2.01, 0) – ponto com erro de aproximação com tolerância
P4 = (0, 1.01, 0) – ponto com erro de aproximação com tolerância
P3 = (0, [2.01, 1.99], 0) – ponto com erro de aproximação com aritmética intervalar
P4 = (0, [1.01, 0.99], 0) – ponto com erro de aproximação com aritmética intervalar

Fig. 6.10. Comparação de resultados entre aritmética em ponto flutuante com tolerância e aritmética intervalar.

O exemplo na Figura 6.11 mostra a aplicação da operação booleana união entre um sólido de perfil triangular e outro retangular. A tolerância não é grande o suficiente para considerar que os pontos **P4** e **P5** são incidentes aos segmentos do sólido de perfil retangular. Logo, o resultado com tolerância apresenta uma topologia diferente do resultado com aritmética intervalar: no primeiro os segmentos de retas **P1P2** e **P1P3** existem, enquanto que no segundo, estes segmentos foram substituídos por **P1P5** e **P5P2** e por **P1P4** e **P4P3**.



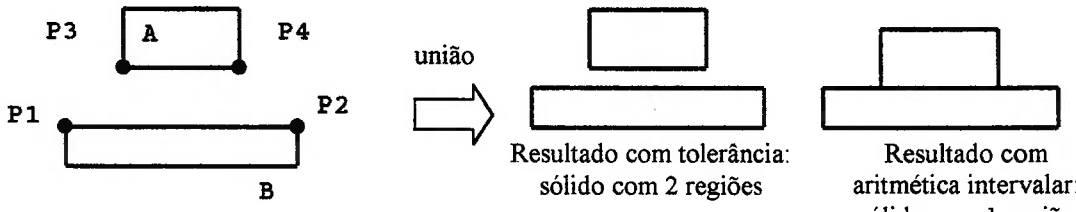
Resultado com tolerância: pontos P4 e P5 não são incidentes aos segmentos P1P2 e P1P3

Resultado com aritmética intervalar: pontos P4 e P5 são incidentes ao segmentos P1P2 e P1P3

`P1 = (1, 2, 0)`
`P2 = (4, 2, 0)` – segmento de reta formado por P1 e P2 tem y = 2
`P3 = (1, -2, 0)` – segmento de reta formado por P1 e P3 tem x = 1
`P4 = (1.01, 0, 0)` – ponto com erro de aproximação com tolerância
`P5 = (3, 1.99, 0)` – ponto com erro de aproximação com tolerância
`P4 = ([0.99, 1.01], 0, 0)` – ponto com erro de aproximação com aritmética intervalar
`P5 = (3, [1.99, 2.01], 0)` – ponto com erro de aproximação com aritmética intervalar
`P6` – ponto incidente ao ponto `P1`

Fig. 6.11. Comparação de resultados entre aritmética em ponto flutuante com tolerância e aritmética intervalar.

Este último exemplo, que se encontra na Figura 6.12, apresenta a aplicação da operação booleana união entre os sólidos **A** e **B**. Uma vez que o teste de incidência ponto - plano informa que os pontos **P3** e **P4** não está no plano formado por **P1** e **P2**, não há como colar os sólidos: o sólido resultante possui duas regiões quando se utiliza a metodologia com tolerância. Utilizando aritmética intervalar, o teste de incidência permite unir as duas regiões.



Resultado com tolerância: sólido com 2 regiões

Resultado com aritmética intervalar: sólido com 1 região

`P1 = (1, 2, 0)`
`P2 = (4, 2, 0)` – segmento de reta formado por P1 e P2 tem y = 2
`P3 = (2, 2.01, 0)` – ponto com erro de aproximação com tolerância
`P4 = (3, 2.01, 0)` – ponto com erro de aproximação com tolerância
`P3 = (2, [1.99, 2.01], 0)` – ponto com erro de aproximação com aritmética intervalar
`P4 = (3, [1.99, 2.01], 0)` – ponto com erro de aproximação com aritmética intervalar

Fig. 6.12. Comparação de resultados entre aritmética em ponto flutuante com tolerância e aritmética intervalar.

7. CONCLUSÕES

Neste trabalho foi apresentado um modelador de sólidos B-Rep implementado com aritmética intervalar arredondada. Para este fim, foi introduzido o conceito de aritmética intervalar e demonstrado que a sua aplicação não é uma simples substituição de tipo, pois os algoritmos precisam ser reformulados, além do que os intervalos podem crescer em demasia tornando-se inúteis. Encontrou-se, então, uma forma de corrigir intervalos demasiadamente grandes baseando-se nas informações redundantes existentes no modelador de sólidos: informações geométricas e topológicas.

Em seguida, foram analizados os casos de incidência entre elementos geométricos, definindo-se novos algoritmos baseados na aplicação de aritmética intervalar. O principal objetivo foi em tornar os resultados mais robustos em relação aos algoritmos baseados em aritmética de ponto flutuante, sendo que uma das principais vantagens foi a validação da propriedade comutativa e transitiva de igualdade entre vértices (algo que não existia em aritmética de ponto flutuante).

Finalmente, foi proposta uma nova estrutura de dados que permitisse a utilização de aritmética intervalar e foram desenvolvidos vários algoritmos baseados nas rotinas de identificação de casos de incidência, culminando na implementação dos operadores de corte e booleanas que fazem uso intenso da identificação de casos de incidência.

O OpenGL se mostrou como uma ferramenta ascessível e de fácil uso. Não apenas gerando resultados com poucas linhas de código, mas utilizando adequadamente recursos disponíveis via hardware de placas de vídeos comercialmente e economicamente acessíveis.

ANEXO A - OPERADORES DE EULER

Conceitualmente, Operadores de Euler podem ser considerados como criadores e modificadores da topologia de superfícies de objetos "*manifolds*". [HOFFMANN, 1989]. Uma superfície "*manifold*" é aquela em que ao redor de todos os pontos dela existe uma vizinhança que é homeomórfica ao plano, isto é, pode-se deformar a superfície "*manifold*" localmente em um plano e não será possível identificar pontos vizinhos separados. Os operadores de Euler são usados como uma linguagem intermediária em sistemas de modelagem de sólido B-rep. São capazes, por exemplo, de adicionar ou retirar vértices, arestas ou faces, simplificando a criação e edição dos sólidos. A consistência topológica é mantida pela geometria e não o inverso. Assim os algoritmos de alto nível processam a geometria e utilizam os Operadores de Euler para garantir a consistência topológica.

O principal objetivo dos operadores de Euler é simplificar a manipulação das complicadas estruturas B-rep. A idéia principal é que a construção dos modelos possa ser realizada passo a passo pelo uso de um conjunto de operadores que manipule a estrutura de dados B-rep e que efetivamente esconda detalhes de implementação da representação.

Os operadores de Euler tornam possível a construção incremental de objetos em uma, duas ou três dimensões de maneira semelhante a desenhar linha a linha. Eles também facilitam alterações locais da forma, característica que é muito conveniente e eficiente para projetistas de modeladores de sólido B-rep.

Tradicionalmente, denota-se os operadores de Euler por um conjunto de letras composto pelas iniciais do que o operador está realizando. Exemplo: *make edge and vertex*, torna-se o operador **MEV**. Cada operador de Euler possui um operador inverso.

Os operadores de Euler utilizados no modelador de sólidos UspDesigner são apresentados abaixo com os seus operadores inversos:

- **MVSF/KSVF** (*Make/Kill Vertex Solid Face*) – este é o primeiro operador a ser aplicado para iniciar a construção de um sólido. Este operador cria os seguintes elementos: um sólido, uma região, um "*shell*", uma face, um laço, um vértice (Figura A.1). Os parâmetros de entrada são as coordenadas do vértice. Para fins

de representação e permitir associar o laço ao vértice que está sendo criado, cria-se uma meia-aresta adicional que permite realizar esta associação. Para definir a existência de uma aresta é necessária uma outra meia-aresta.

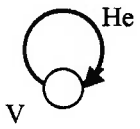


Fig. A.1. Representação de um vértice **V** com uma meia-aresta **He** (sem existência de aresta).

- **MEV/KEV** (*Make/Kill Edge Vertex*) – em geral, após iniciar a construção de um sólido usando o **MVSF**, é possível acrescentar mais vértices. O operador **MEV** cria: uma aresta e um vértice (Figura A.2). A aresta ligará o vértice fornecido como parâmetro de entrada ao novo vértice que será criado;

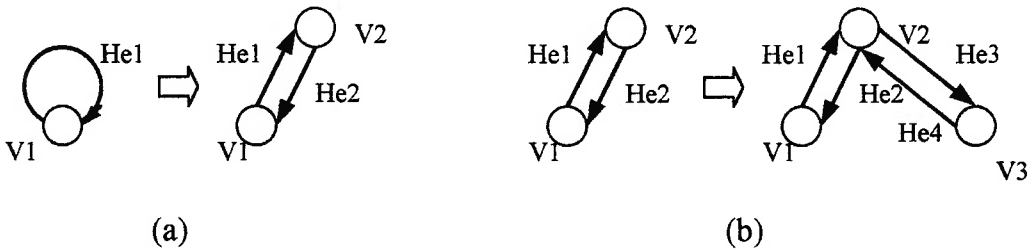


Fig. A.2. (a) **MEV** criando uma meia-aresta; (b) **MEV** criando duas meia-arestas;

- **MEF/KEF** (*Make/Kill Edge Face*) – O operador **MEF** cria uma aresta e uma face (Figura A.3). A aresta ligará um par de vértices (fornecidos como parâmetros de entrada). No exemplo da Figura A.3, originalmente existia uma face **F1** definida por seis meia-arestas. Em seguida, após o uso do Operador **MEF**, existirão duas faces: **F1** (formada pelas meia-arestas em sentido anti-horário **V1 -> V4 -> V3 -> V2**) e **F2** (formada pelas meia-arestas em sentido horário **V1 -> V2 -> V3 -> V4**).

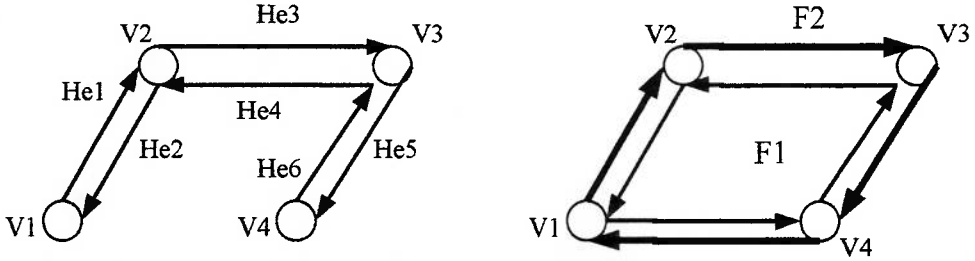


Fig. A.3. Exemplo de utilização do Operador **MEF**, face **F2** foi criada.

- **KEMR/MEKR** (*Kill/Make Edge Make/Kill Ring*) – Em algumas situações, é desejável que uma face tenha laços internos (furos). O operador **KEMR** realiza a primeira etapa na criação de um laço interno: divide um laço existente em duas partes. Ou seja, ao remover uma aresta, ele deixa um vértice isolado dos demais que formavam o laço inicial. E a partir deste vértice isolado, é criado o novo laço. Este operador remove uma aresta, criando um laço. No exemplo apresentado na Figura A.4, o laço externo possui, inicialmente, cinco vértices. Após a aplicação do operador **KEMR**, a face possui dois laços: um com quatro vértices e o outro com um único vértice.

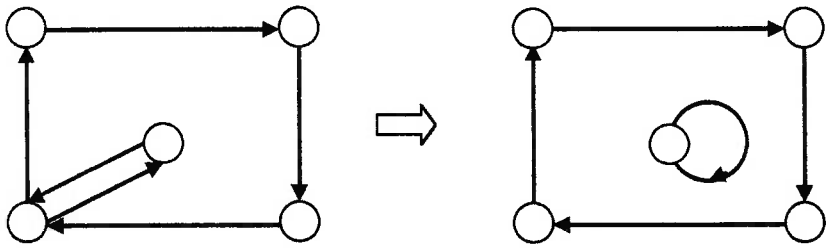


Fig. A.4. Exemplo de utilização do **KEMR**.

- **KFMRH/MFKRH** (*Kill/Make Face Make/Kill Ring Hole*) – O operador **KFMRH** transforma duas faces em uma única face, colocando o laço externo de uma das faces como laço interno da outra face. Utilizado, por exemplo, em casos em que o operador **KEMR** foi aplicado para iniciar um furo e foram adicionados vértices que chegam até a face oposta, e deseja-se que este furo seja passante. Este operador remove uma face e cria um laço na outra face. Um exemplo é apresentado na Figura A.5.

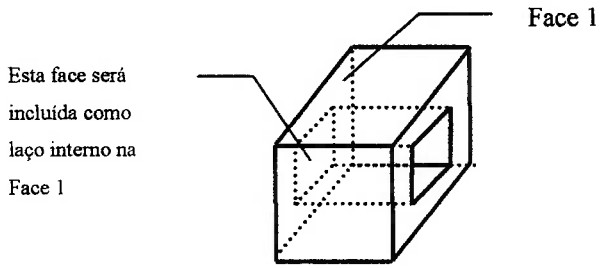


Fig. A.5. Exemplo de utilização de **KFMRH**.

- **KSFMR/MSFKR** (*Kill/Make Shell Face Make/Kill Ring*) – O operador **KSFMR** une dois "*shells*" em um único "*shell*". São removidos uma face, um "*shell*" e, talvez, uma região, e é criado um laço na face incidente (Figura A.6). Naturalmente, todas as informações do "*shell*" a ser removido são transferidas para o outro "*shell*". Na Figura A.6 são apresentados dois exemplos de aplicação do operador **KSFMR**. No Exemplo (a), os dados do "*shell*" 2 (vértices, faces, arestas) são colocados no "*shell*" 1 (e o "*shell*" 2 é então removido) e a face que coincidia com outra do "*shell*" 1, torna-se um laço interno da face do "*shell 1*", deste modo criando um furo não passante do "*shell*" 1. Já no exemplo (b), o "*shell*" resultante não apresenta furos, mas apresenta ainda uma face com um laço interno.

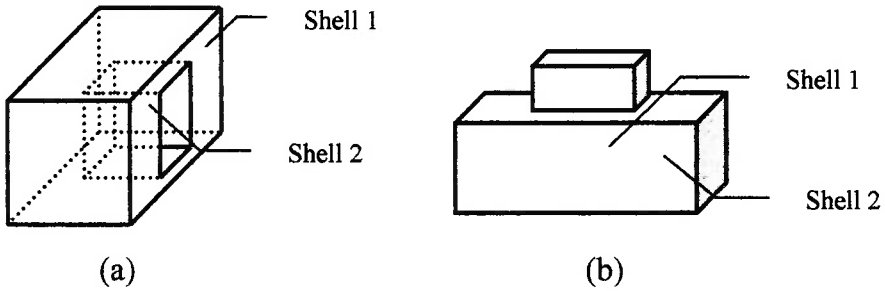


Fig. A.6. (a) a união dos dois "*shells*" gera um furo não passante (b) Utilização de KSFMR.

Durante a construção de modelos com Operadores de Euler, a topologia é mantida válida segundo a equação de Euler – Poincaré, mostrada na equação abaixo:

$$v - e + f = 2 * (s - h) + r$$

Sendo: **s** peças disconexas, **f** faces, **e** arestas, **v** vértices, **h** furos, **l** laços e **r** anéis, tem-se que $r = l - f$.

Ao final de uma seqüência de Operadores de Euler assume-se que a geometria do sólido está correta, mas durante os estágios intermediários não há como manter a topologia e a geometria consistentes devido à presença de faces não planares; que, freqüentemente, não são possíveis de serem representadas por nenhuma forma matemática. Logo, os operadores de Euler não são operadores seguros por si, mas devem ser colecionados em seqüências que forneçam um significado.

Os nomes utilizados foram baseados da implementação de Mäntylä (1988).

ANEXO B - ALGORITMO DE CORTE DE SÓLIDOS – "*CUT SOLID*"

O corte de um sólido pode ser dividido nas etapas que estão representadas na Figura B.1, onde a função principal `CutSolid` chama as demais funções, que realizam tarefas auxiliares no algoritmo principal.

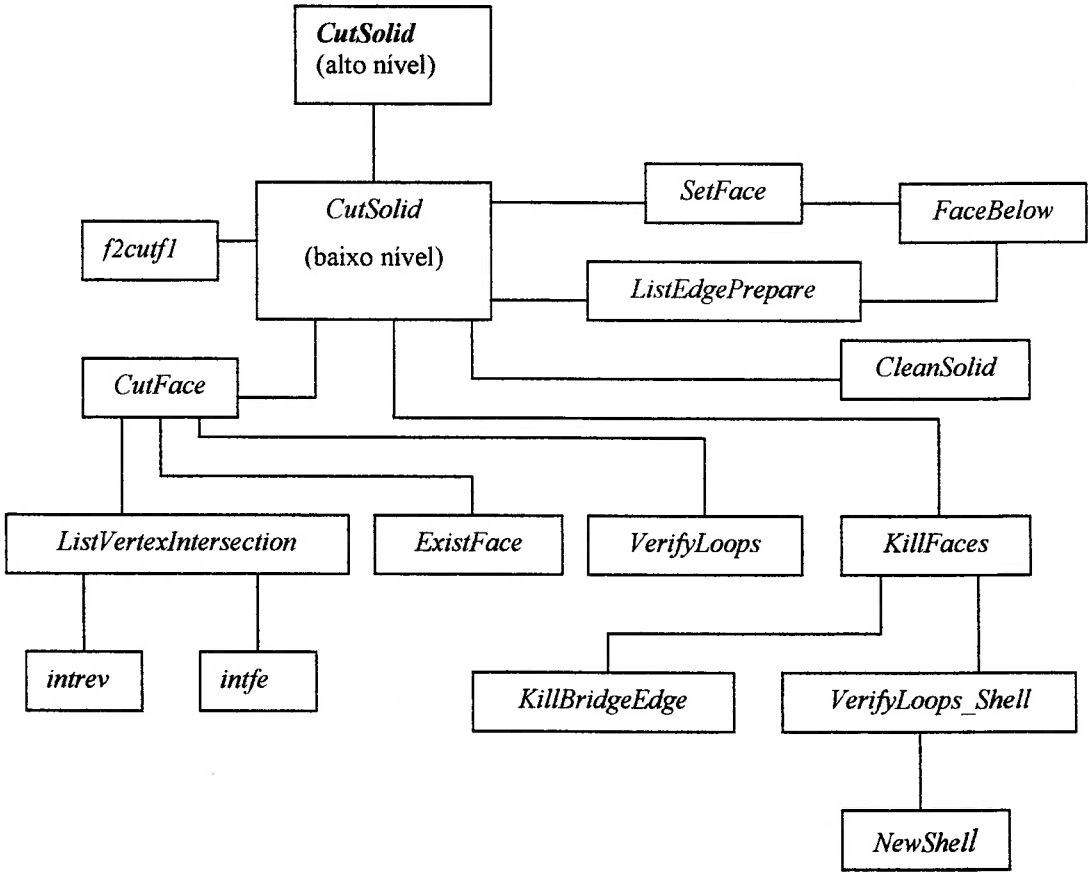


Fig. B.1. Componentes necessários para realizar o algoritmo de corte de sólido.

```
a) CutSolid (     int sn,
                  tnVector<T,4> n)
```

Parâmetros de Entrada:

sn – identificação do sólido a ser cortado;

n – equação do plano de corte;

Função de alto nível de interface com o usuário. Recebe a identificação do sólido e a equação do plano de corte. Utilizando funções internas da estrutura de dados, são obtidos os ponteiros para o sólido e para a face. Estes ponteiros são passados para a etapa seguinte.

```
b) CutSolid (     TSolid<T> *s,
                  tnVector<T,4> n2)
```

Parâmetros de entrada:

s – ponteiro para o sólido a ser cortado;

n2 – equação do plano de corte;

Função que realiza o corte do sólido fornecido pelo ponteiro **s**, pelo plano de corte com normal **n2**.

Internamente, divide-se o processo em:

- cortar todas as faces que possuem intersecção com o plano de corte **n2** (utilizando a função **CutFace**). Nesta parte, todas as faces do sólido são pesquisadas. Uma face **f_it** é considerada como sendo cortada pelo plano de corte, se:
  - não for paralela ao plano de corte;
  - existirem vértices da face que estão acima do plano e corte e outros abaixo do plano de corte (verificação realizada pela função **f2cutf1**);

  Se a face **f_it** for cortada pelo plano de corte, a rotina **CutFace** é acionada.
- As faces que são cortadas pelo plano de corte definem um conjunto de arestas que estão sob o plano de corte. É necessário identificar as arestas que não podem ser removidas e guardá-las na lista de arestas **EdgeList** (a função **ListEdgePrepare** realiza esta tarefa);
- identificar as faces que estão abaixo do plano de corte e acrescentá-las à lista **FaceList** para que sejam removidas (tarefa realizada pela função **SetFace**);
- Remover as faces que estão na **FaceList** (tarefa realizada pela função **KillFaces**);
- Limpar internamente a estrutura de dados do sólido (é utilizada a função **TCleanSolid**);

Verificação em aritmética intervalar: comparação de normais

```
c) f2cutf1 (      TFace<T> *f,
                  tnVector<T,4> n)
```

Parâmetros de Entrada:

f – ponteiro para a face verificada;

n – equação do plano de corte;

A face **f** é considerada cortada pelo plano de corte de normal **n**, se existir algum ponto acima e outro abaixo do plano de corte.

Trabalhando-se apenas com o laço externo da face **f**, procura-se por um vértice que não esteja sobre o plano de corte. Assim que ele for encontrado, ele é classificado como estando acima ou abaixo do plano de corte. Com base neste fato, os demais vértices são verificados. O processo é terminado quando:

- não há mais vértices, ou seja, a face **f** está totalmente acima ou totalmente abaixo do plano (Figuras B.2 b e c);
- encontra-se um vértice com situação diversa em relação ao primeiro vértice não incidente ao plano de corte – neste caso a face **f** é cortada pelo plano de corte (Figura B.2 a);

Fig. B.2. a) face cortada pelo plano; b) face abaixo do plano; c) face acima do plano;

Verificação em aritmética intervalar: incidência de vértice em plano.

```
d) CutFace (      TFace<T> *f1,
                  list< TEdge<T> * > EdgeList,
                  tnVector<T,4> feq)
```

Parâmetro de Entrada:

f1 – ponteiro para a face a ser cortada;

EdgeList – lista de arestas incidentes ao plano de corte;

feq – equação do plano de corte;

A face **f1** será dividida pelo plano de normal **feq**, sendo que as arestas que forem incidentes ao plano, devem ser guardadas na lista de arestas **EdgeList** para serem utilizadas em etapas seguintes.

Inicialmente, os vértices que interseccionam o plano **feq** são determinados. A função **ListVertexIntersection** realiza esta tarefa. Estes vértices são guardados na lista de vértices **VertexList**.

Com os vértices de **VertexList**, é verificado se existe a necessidade de criação de uma nova face com o uso de **ExistFace**, a cada par de vértices.

Existem quatro situações possíveis para classificar as arestas de intersecção:

- os vértices estão em laços diferentes: neste caso, o laço é removido, usando o operador de Euler **MEKR** (Figura B.3), e a nova aresta é guardada em **EdgeList**;

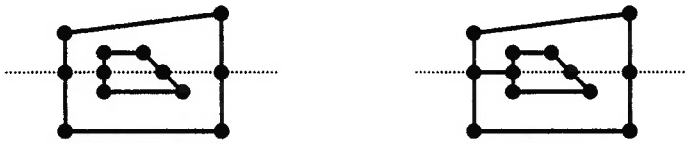


Fig. B.3. Cortando uma face, utilizando o operador de Euler **MEKR**.

- os vértices já estão conectados por uma aresta (Figura B.4): neste caso, a aresta existente é armazenada em **EdgeList**;

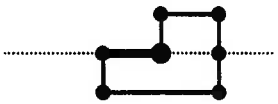


Fig. B.4. Vértices já estão ligados por uma aresta.

- os vértices estão no mesmo laço, não estão conectados por uma aresta e a aresta definida pelos dois vértices é interna a face: neste caso será criada uma nova face, usando o operador de Euler **MEF** (Figura B.5); a nova aresta é guardada em **EdgeList**.

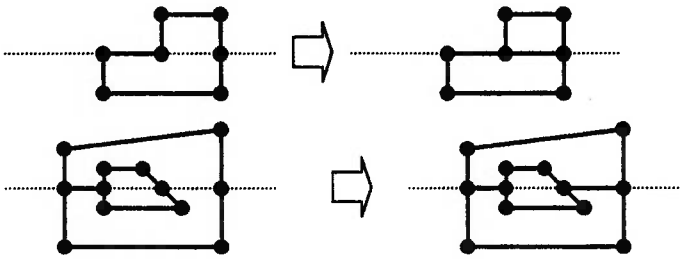


Fig. B.5. Exemplos de formação de novas faces utilizando o operador de Euler **MEF**.

Após criar a nova face, deve-se verificar se os laços internos estão nas faces adequadas (**VerifyLoops** realiza esta tarefa– representado na Figura B.6):

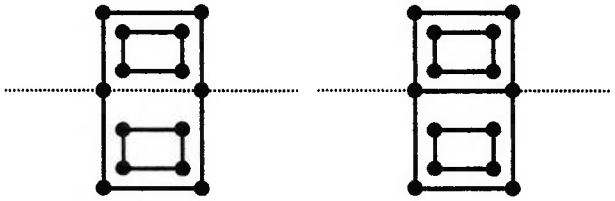


Fig. B.6. O corte desta face gerou uma nova face. Os laços internos devem ser verificados.

- a aresta definida pelos dois vértices não é interna à face: não é realizado nada (Figura B.7).

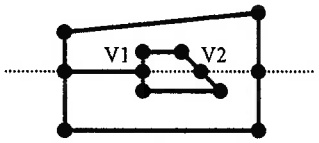


Fig. B.7. Neste exemplo, os vértices **V1** e **V2** não são ligados, pois não existe face entre eles (região de furo da face).

```
e) VerifyLoops (  TFace<T> *f1,
                  TFace<T> *f2)
```

Parâmetros de Entrada:

f1 – ponteiro para a face original;

f2 – ponteiro para a face recém criada;

Verifica se os laços internos de **f1** (face original antes de ser cortada) deveriam estar na face **f2** (que foi criada pelo operador de Euler **MEF**).

Todos os laços internos de **f1** são verificados. Utilizando o procedimento para verificação de vértice em laço (a função **contlv** realiza este teste), é comprovado se o primeiro vértice de um laço interno está no interior do laço externo de **f1**. Se não estiver, o laço interno é transferido para **f2**.

```
f) ListVertexIntersection (  TFace<T> *f1,
                             tnVector<T,4> feq,
                             list< THalfedge<T>* > VertexList)
```

Parâmetros de Entrada:

f1 – ponteiro para a face a ser cortada;

feq – equação do plano de corte;

VertexList – lista contendo os vértices sobre o plano de corte;

A lista de vértices **VertexList** deverá conter os vértices de intersecção da face **f1**, com o plano **feq**.

Para todas as arestas de todos os laços da face, é verificado o ponto de intersecção da aresta com o plano **feq**. A intersecção, se existir, é obtida pela função **intfe**, que retorna as coordenadas **x, y, z** do ponto de intersecção.

Logo, podem ocorrer as seguintes situações:

- um dos vértices da aresta está sobre o plano de corte: não é necessário um novo vértice. Apenas determina-se qual dos vértices está sobre o plano de corte e ele é guardado na **VertexList;**
- nenhum dos dois vértices da aresta está sobre o plano de corte: significa que um novo vértice deve ser criado, pela aplicação do operador de Euler **MEV**. Este novo vértice é guardado na **VertexList**;

Assim que todas as arestas forem percorridas, obtém-se um conjunto de vértices em **VertexList**. Entretanto, é necessário organizá-los para que possam ser utilizados. Para isso, determina-se um dos pontos extremos e a partir de um deles, os outros são organizados em sequência.

Utilizando-se os dois primeiros vértices de **VertexList** (**V1** e **V2**) é possível iniciar uma análise para determinar os extremos. Em seguida é verificado se o próximo vértice é colinear com os dois pontos iniciais, para isto utiliza-se a rotina **intrev**. Esta função retorna um parâmetro **t** da equação: **V = (1 - t) * V1 + t V2**. Se **t = 0, V = V1**; se **t =1, V = V2**; se **t<0**, o ponto **V** está mais afastado do que **V1** (representado na Figura B.8).

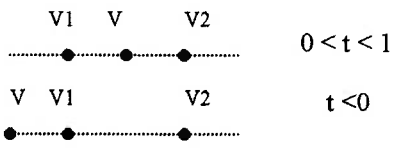


Fig. B.8. O parâmetro **t** e o posicionamento relativo entre os vértices.

Logo, quando: **0 < t < 1**, o vértice está entre **V1** e **V2**, não é necessário fazer nada, em seguida analisa-se o próximo vértice. Quando **t<0**, substitui-se **V1** pelo vértice **V**, para que **V** seja o novo extremo. A verificação continua até acabar os vértices. Deste modo, determina-se um dos extremos de **VertexList**.

No passo seguinte, os vértices são ordenados conforme o parâmetro **t**. Considerando o extremo encontrado no parágrafo acima como fixo, todos os vértices

serão verificados para que o vértice **V2** seja o mais próximo possível de **V1**. Só existem duas situações agora: **t>1** e **0<t<1**. Se **t>1**, o vértice é mais afastado de **V2** (e de **V1** também). Logo, assim que **t<1**, deve-se trocar o vértice para que este seja o novo **V2**. Fazendo iterativamente este processo para todos os vértices, **VertexList** ficará ordenado em relação ao extremo **V1**.

Verificação em aritmética intervalar: incidência de vértices

g) `intfe ( tnVector<T,4> n,`
`THalfedge<T> *he,`
`T x,`
`T y,`
`T z)`

Parâmetros de entrada:

n – equação do plano de corte;

he – meia-aresta que será cortada;

x, y, z – coordenadas do ponto de intersecção entre a meia-aresta e o plano de corte;

Procura-se pelo ponto de intersecção entre o plano de corte e a aresta.

Ponto de intersecção: **(x0,y0,z0)**

Plano de corte: **(nx, ny, nz, nw)**

Aresta formada por dois pontos: **v1 = (x1,y1,z1), v2 = (x2,y2,z2)**, **t (t=0 V = V1, t=1, V = V2)**

O ponto de intersecção pode ser obtido, pela determinação do parâmetro **t**.

O ponto de intersecção pertence ao plano de corte:

```
nx*x0 + ny*y0 + nz*z0 + nw = 0
```

E também à aresta:

```
x0 = x1 + t * (x2 . x1)
y0 = y1 + t * (y2 . y1)
z0 = z1 + t * (z2 . z1)
```

Logo, substituindo-se, obtém-se:

```
nx * (x1 + t * (x2 . x1) ) + ny * (y1 + t * (y2 . y1) ) +
     + nz * (z1 + t * (z2 . z1) ) + nw = 0
```

Isolando-se o parâmetro **t**, obtém-se:

```
t = - (nx * x1 + ny * y1 + nz * z1 + nw) / (nx * (x2-x1) +
     + ny * (y2-y1) + nz * (z2-z1) )
```

Se, durante o cálculo, o parâmetro `t` obtido for menor que 0 ou maior que 1, este parâmetro é inválido, pois não está entre os pontos `v1` e `v2` da meia-aresta.

Após o cálculo do parâmetro `t`, as coordenadas do ponto de intersecção podem ser obtidas aplicando-se as equações para determinar `(x0, y0, z0)`. Entretanto, testes realizados com as Operações Booleanas (que também utilizam esta função), mostram que este ponto possui uma grande tendência para aumentar o intervalo e gerar falhas nos próximos pontos de intersecção que dependem deste ponto de intersecção. Graficamente, o ponto pode ficar muito maior que a linha intervalar (Figura B.9 a). Torna-se necessário ajustar o ponto para que seus intervalos fiquem delimitados à linha intervalar. Este ajuste pode ser feito utilizando a técnica descrita para determinação de incidência entre ponto e linha, através do cálculo dos extremos dos intervalos (apresentado no item 7.2. Ponto e Linha). Ou seja, realiza-se uma correção da propagação dos erros provocados pela aritmética intervalar.

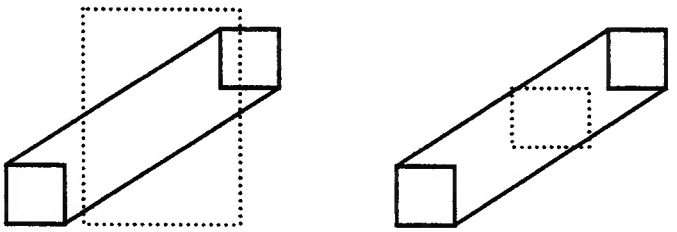


Fig. B.9. a) Ponto de Intersecção Intervalar que está com intervalos incoerentes. b) Ponto de Intersecção com intervalos calculados e ajustados.

```
h) Intrev(  TVertex<T> *v1,
            TVertex<T> *v2,
            TVertex<T> *v3,
            T *t)
```

Parâmetros de entrada:

`v1` – vértice inicial do segmento de reta;

`v2` – vértice final do segmento de reta;

`v3` – vértice que pode estar no segmento de reta formado por `v1` e `v2`;

`t` – parâmetro de reta, se `0 ≤ t ≤ 1`, `v3` pertence a reta formada por `v1` e `v2`;

Inicialmente alguns casos simples são verificados: `t = 0`, ou seja, `v1` e `v3` são vértices incidentes, ou `t = 1` com `v2` e `v3`. Se `v3` não seguir estes casos, verifica-se se `v3` está no segmento `v1v2`.

Para que o ponto `v3` seja considerado incidente à linha formada pelos pontos `v1` e `v2`, são realizadas as seguintes etapas:

- uma vez que na etapa anterior (`ListVertexIntersection`) os pontos calculados são colineares (intersecção de face com plano), é possível calcular o parâmetro `t` por meio da seguinte equação:

  `( 1 . t ) * v1 + t * v2 = v(t)`

  `0 ≤ t ≤ 1` (para `t = 0, v(0) =` P_1 ; para `t=1, v(1) =` P_2)

  Rescrevendo a equação, tem-se: `t *` (v_2 `.` v_1) `= v(t) .` v_1

  Para `v(t) =` v_3, obtém-se:

  `t =` (v_3 `.` v_1) `/` (v_2 `.` v_1);

  Multiplicando escalarmente o denominador e o numerador pelo vetor (v_2 `.` v_1):

  `t = [` (v_2 `.` v_1) `•` (v_3 `.` v_1)`] / [`(v_2 `.` v_1) `•` (v_2 `.` v_1) `]`

- com o valor de `t` obtido, pode-se criar um novo ponto v_{teste}, que está na linha formada por v_1 e v_2. Se v_{teste} for coincidente a `v3` e `0 ≤ t ≤ 1`, significa que `v3` está entre os vértices `v1` e `v2`.

Verificações em Aritmética Intervalar: incidência de pontos.

```
i) ExistFace(     TFace<T> *f,
                  THalfedge<T> *he1,
                  THalfedge<T. *he2,
                  int e)
```

Parâmetros de entrada:

`f` – ponteiro para a face a ser cortada;

`he1` – ponteiro para a meia-aresta – primeiro vértice;

`he2` – ponteiro para a meia-aresta – segundo vértice;

`e` – identificação da aresta (valor de retorno de referência)

A primeira verificação é se os vértices `v1` e `v2` (apontados por `he1` e `he2`) pertencem a laços distintos. A meia-aresta possui em sua estrutura, um ponteiro

específico para indicar à qual laço ela pertence. Se confirmado que pertencem a laços distintos, esta rotina termina e retorna esta situação.

A segunda verificação é se os vértices `v1` e `v2` são ligados por uma aresta. Caso afirmativo, o valor `e` recebe a indentificação da aresta que liga estes vértices e esta rotina termina e retorna esta situação.

Última verificação: existe material entre os vértices `v1` e `v2` ? É realizado o seguinte procedimento:

- somam-se os produtos vetoriais dos vetores formados por vértices consecutivos de arestas consecutivas do segmento de laço iniciando do vértice `v1` até alcançar o vértice `v2` (representado na Figura B.10);

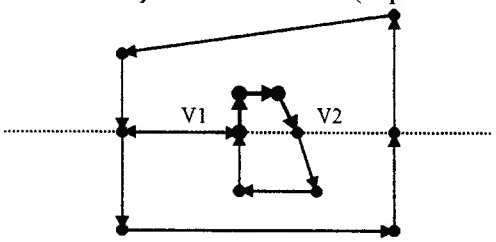


Fig. B.10. Neste exemplo, o conjunto de meia-aresta que parte de `V1` e chega até `V2` possui orientação contrária a da face.

- Se o vetor somatório dos produtos vetoriais tiver o mesmo sentido que a normal da face, então há material entre os vértices `v1` e `v2` – pois ambos pertencem a um laço com circulação positiva. Então esta rotina termina e retorna esta situação.

Neste outro exemplo na Figura B.11, existe material, pois o caminho do vértice `v1` até `v2`, envolve meia-arestas do laço externo (que possuem dimensões maiores, resultando em produtos vetoriais maiores que os produtos vetorias calculados das meia-arestas do laço interno).

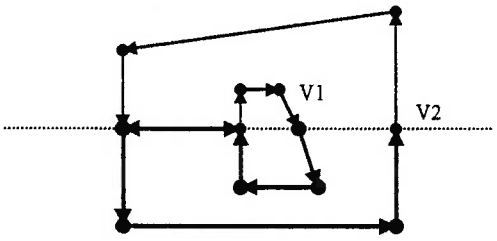


Fig. B.11. O conjunto de meia-aresta que parte de **V1** até **V2** tem a mesma orientação que a face.

Se passar por todas as verificações acima, não existe material e retorna esta situação.

```
j) ListEdgePrepare (    list < TEdge<T>* > ListEdge,
                        tnVector<T,4> n)
```

Parâmetros de Entrada:

ListEdge – lista contendo arestas que são incidentes ao plano de corte

n – equação do plano de corte

Inicialmente a **ListEdge** contém as arestas que estão no plano de corte. A princípio, estas arestas não devem ser apagadas pois farão parte do sólido cortado. Entretanto, existem arestas que estão no plano de corte e podem ser removidas (exemplo na Figura B.12):

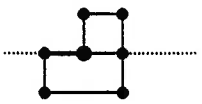


Fig. B.12. A aresta em destaque pode ser removida.

Esta função verifica se as arestas de **ListEdge** não devem ser removidas.

Do exemplo, nota-se que, a aresta em destaque é utilizada por duas faces: uma que está abaixo do plano de corte, e outra que está no plano de corte. Logo, apenas é verificado se as faces que compartilham a aresta estão abaixo ou sobre o plano de corte. Esta verificação é feita pela função **FaceBelow**.

```
k) FaceBelow (    TFace<T> *f,
                  tnVector<T,4> n)
```

Parâmetros de Entrada:

f – ponteiro para a face a ser verificada;

n – equação do plano de corte;

Utilizando-se o laço externo da face **f**, percorre-se todos os vértices deste laço, até que um vértice que esteja acima do plano de corte seja encontrado. Neste momento do algoritmo de corte, não existem mais faces com vértices acima e vértices abaixo do plano de corte; ou todos os vértices da face em análise estão acima ou sobre o plano de corte, ou todos os vértices da face em análise estão abaixo ou sobre o plano de corte.

O primeiro vértice que não estiver sobre o plano de corte, indica a posição da face **f** em relação ao plano de corte. Se após todos os vértices serem analisados e todos os vértices estarem sobre o plano de corte, naturalmente a face **f** está sobre o plano de corte, e portanto, esta face também deve ser removida.

Verificação em Aritmética Intervalar: incidência de vértice em plano

```
l) SetFace( TSolid<T> *s,
            tnVector<T,4> n,
            list<int> ListFace)
```

Parâmetros de Entrada:

s – ponteiro para o sólido que está sendo cortado;

n –equação do plano de corte;

ListFace – lista de faces com os identificadores das faces.

Esta função seleciona as faces que deverão ser removidas do sólido. Ou seja, as faces abaixo do plano de corte.

Todas as faces do sólido **s** são verificadas, percorrendo todas as regiões e em seguida todos os "*shells*". Todas as faces são verificadas utilizando a função **FaceBelow**, e a inclusão da face na lista de faces ocorre caso seja necessário.

```
m) KillFaces(     int sn,
                  list<int> &ListFace,
                  list<int> &ListEdge)
```

Parâmetros de entrada:

sn – identificação do sólido a ser cortado;

ListFace – lista contendo os identificadores das faces a serem removidas;

ListEdge – lista contendo os identificadores das arestas que não podem ser removidas.

Esta função remove as faces da **ListFace**, tomando o cuidado para não remover as arestas da **ListEdge**.

Internalmente, divide-se esta função nas seguintes etapas:

- redução das faces a serem removidas: Nesta redução, os seguintes passos para cada face da **ListFace** são realizados:
  - remover os laços internos das faces: para todos os laços internos, aplica-se o operador de Euler **KEV**, para que os vértices e arestas sejam removidas até que o laço fique com apenas uma meia-aresta. Deste modo, pode-se aplicar o operador de Euler **MEKR**. A face ficará apenas com o laço externo;
  - eliminar as arestas do laço externo da face: primeiro, todas as meia-arestas do laço externo da face são analisadas para verificar se é possível removê-las (é verificado se esta aresta não está em **ListEdge**). Guardam-se as referências das arestas que podem ser apagadas em uma lista auxiliar. Em seguida, esta lista auxiliar é utilizada para, sequencialmente, remover as arestas, aplicando o operador de Euler **KEV**.
- remoção das faces: As faces da estrutura de dados devem ser removidas, aplicando-se o operador de Euler **KEF**. Todas as faces que estão guardadas em **ListFace** são analisadas. Deve-se ter cuidado de não remover arestas que estão na **ListEdge**;
- remoção das arestas pontes que podem se formar durante o processo de remoção de faces – criação de novas faces: Neste momento, ainda deve restar pelo menos uma face na **ListFace**, mas que não pode ser removida pois é formada por arestas que estão na **ListEdge**. Deve-se verificar se esta face deve ser dividida, devido à existência de arestas pontes (exemplo na Figura B.13). A detecção de arestas pontes é feita pela função **KillBridgeEdge**.

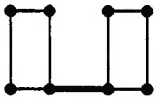


Fig. B.13. A aresta em destaque é uma aresta ponte que liga dois laços e deve ser removida.

- criação de novos "shells": Se existiam arestas pontes, e elas foram removidas, significa que novos laços foram criados, e não faces. Conforme o corte do sólido, talvez seja necessário criar novas faces, novas regiões e novos "*shells*". A função **`VerifyLoops_Shell`** faz esta verificação para uma face. Logo, todas as faces de **`ListFace`** são analisadas por esta função.

```
n) KillBridgeEdge(      TLoop<T> *l,
                        list< TEdge<T> * > &ListEdge)
```

Parâmetros de Entrada:

`l` – laço o qual será verificado a existência de arestas pontes;

`ListEdge` – lista de arestas que não podem ser removidas

Esta função analisa todas as meia-arestas que formam o laço `l`, procurando por arestas pontes. A aresta ponte possui as duas meia-arestas pertencentes ao mesmo laço (exemplo na Figura B.14).

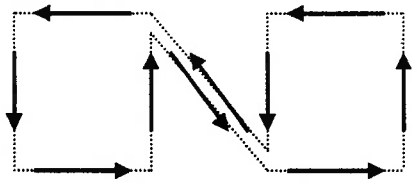


Fig. B.14. As meia-arestas de uma aresta ponte estão no mesmo laço.

Novamente, as arestas que estão na **`ListEdge`** não devem ser removidas. Assim que uma aresta ponte que pode ser removida for encontrada, aplica-se o operador de Euler **`KEMR`**. Outra situação que requer cuidado é exemplificado na aresta ponte da Figura B.15:

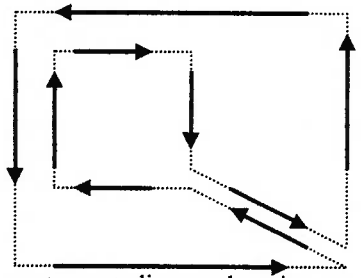


Fig. B.15. Neste caso, a aresta ponte liga um laço interno com o laço externo.

Quando o operador de Euler **KEMR** for aplicado, não existe a garantia de que o ponteiro para o laço externo da face, esteja apontando para o laço maior. Conforme a ordem em que são enviados os parâmetros para o operador de Euler **KEMR**, é possível que, erroneamente, a face considere como laço externo o laço menor (que é interno). Realizando o cálculo da área do laço que será formado após o uso do operador de Euler **KEMR**, pode-se evitar este erro, pois, caso a área retorne valor negativo, significa que é um laço interno, então, os parâmetros são invertidos quando se aplica o operador de Euler **KEMR**. Este cálculo é semelhante ao aplicado na função ExistFace.

Em seguida, aplica-se recursivamente esta função, para o novo laço, e para o laço que foi modificado.

p) `VerifyLoops_Shell ( TFace<T> *f1)`

Parâmetros de entrada:

`f1` – face cujo laços devem ser analisados;

A função `KillBridgeEdge` não cria faces. Para isto, existe a rotina `VerifyLoops_Shell`.

Esta função realiza uma verificação semelhante à função `VerifyLoops`. Ou seja, verifica se os laços internos das faces estão realmente no interior do laço externo (exemplo na Figura B.16).

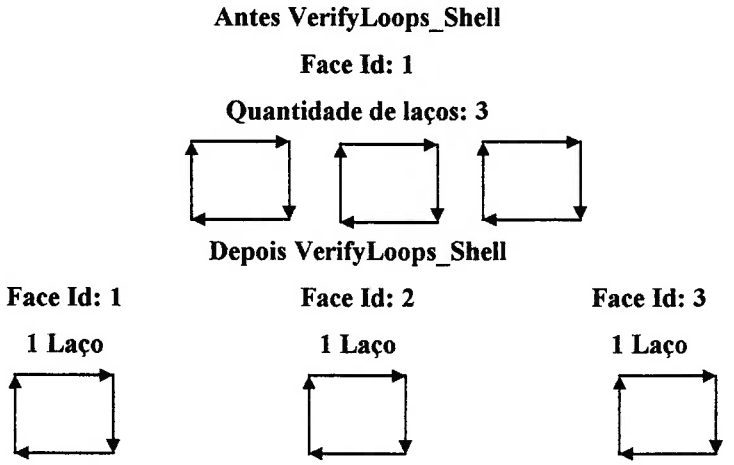


Fig. B.16. Exemplo do resultado esperado da função `VerifyLoops_Shell`.

Utiliza-se a função **`contlv`** para determinar se os primeiros vértices dos laços internos estão no interior do laço externo da face. Se o primeiro vértice do laço não estiver no interior do laço externo da face, significa que o este laço também não deve estar na face. Logo, é aplicado o operador de Euler **`MFKRH`**, que elimina o laço e cria uma nova face. Em seguida, verifica-se, usando a função **`NewShell`**, se existe a necessidade de um novo "*shell*". Caso necessário, deve-se aplicar o operador de Euler **`KFMRH`**, operador reverso de **`MFKRH`**, pois a etapa seguinte necessita de um laço. O operador de Euler **`MSFKR`**, remove o laço, cria um novo "*shell*" e uma nova face.

Verificação em Aritmética Intervalar: incidência de laço em face

```
q) NewShell(      TFace<T> *f1,
                  TFace<T> *f2,
                  list<int> &ListFace)
```

Parâmetros de entrada:

`f1` – face original;

`f2` – nova face criada, que antes era laço de f1;

`ListFace` – lista de faces com os identificadores das faces marcadas

Realiza-se a busca da face **`f2`**, começando da face **`f1`**. Se, durante a procura, for encontrada a face **`f2`**, então as faces pertencem ao mesmo "*shell*". Caso contrário, um novo "*shell*" é necessário (exemplo na Figura B.17 b).

(a) (b)

Fig. B.17. a) não é necessário novo "*shell*"; b) um novo "*shell*" deve ser feito.

Inicialmente, marca-se a face `f1`, guardando-a na `ListFace`. Obtém-se a primeira meia-aresta da face `f1`. Verifica-se a outra face que compartilha a aresta. Se ela for a face `f2`, retorna falso (um novo "*shell*" não é necessário). Se não for, verifica-se se esta face está na `ListFace`. Se estiver, verifica-se a próxima meia-aresta e a sua face compartilhada. Se não estiver, aplica-se a recursão, enviando como parâmetros esta face, a face objetivo `f2` e `ListFace`.

Deste modo, quando forem verificadas todas as meia-arestas e suas faces compartilhadas, é possível não encontrar a face *f2*, o que indica que existe a necessidade de um novo "*shell*". O uso da `ListFace` e de recursão impedem que sejam feitas múltiplas verificações da mesma face.

ANEXO C - ALGORITMO DE OPERAÇÃO BOOLEANA

A operação booleana pode ser dividida em diversas etapas, que estão representadas na Figura C.1, onde estão as principais funções de auxílio ao algoritmo de operação booleana:

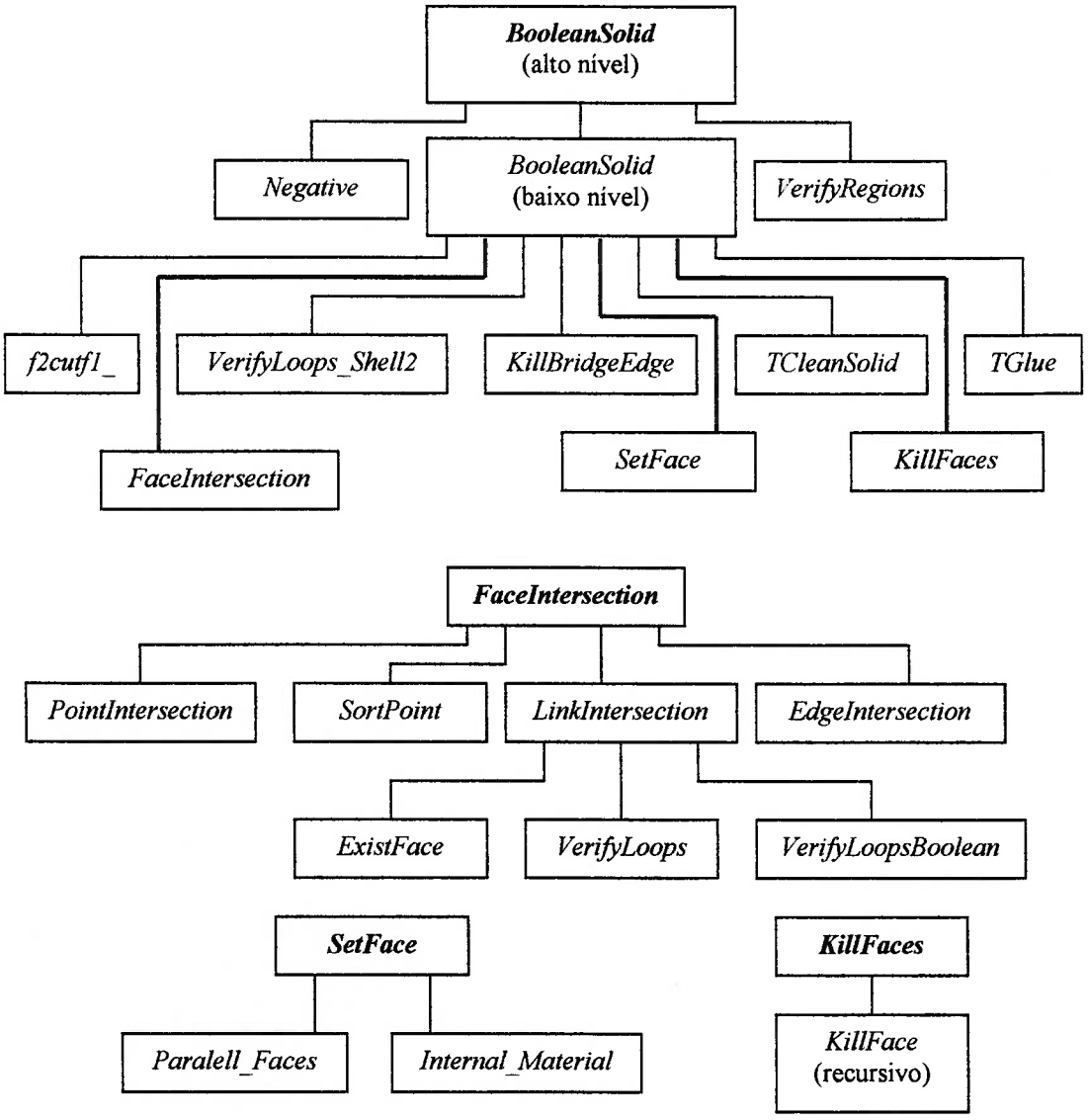


Fig. C.1. Principais Funções do Algoritmo de Operação Booleana.

```
a) BooleanSolid (      int sn1,
                       int sn2,
                       op)
```

Parâmetros de Entrada:

`sn1` – identificação do primeiro sólido;

`sn2` – identificação do segundo sólido;

op – tipo de operação Booleana a ser realizada; pode ser União, Subtração, Intersecção;

Função de alto nível que implementa a interface com o usuário. Recebe os identificadores de dois sólidos entre os quais se deseja aplicar a operação booleana de tipo **op**.

Inicialmente, o algoritmo verifica se estes dois sólidos existem. Em caso verdadeiro, conforme a operação **op**, um procedimento adequado é realizado:

- União: utiliza a função **BooleanSolid** com os ponteiros dos sólidos;
- Subtração: o primeiro sólido é transformado em sólido negativo com o uso da função **Negative**. Em seguida, é utilizada a função **BooleanSolid** com os ponteiros dos dois sólidos. Em seguida, o sólido resultante é enviado para a função **Negative**. Finalmente, verifica-se se existe alguma falha nas regiões do sólido resultante, com o uso da função **VerifyRegions**;
- Intersecção: os dois sólidos são transformados em sólidos negativos pela função **Negative**. Em seguida, é utilizada a função **BooleanSolid** com os ponteiros dos dois sólidos. No passo seguinte, o sólido resultante é enviado para a função **Negative**. Finalmente, verifica-se se existe alguma falha nas regiões do sólido resultante, com o uso da função **VerifyRegions**;

b) Negative(TSolid<T> *s)

Parâmetros de Entrada:

s – ponteiro para o sólido;

Esta função transforma um sólido em um sólido negativo. Ocorre a inversão das normais das faces. Para que isto ocorra, deve-se inverter a orientação dos laços que formam o sólido (Figura C.2).

Logo, todas as arestas devem ser analisadas para que a inversão das meia-arestas ocorra de forma coerente para que seus ponteiros mantenham a integridade da estrutura topológica.

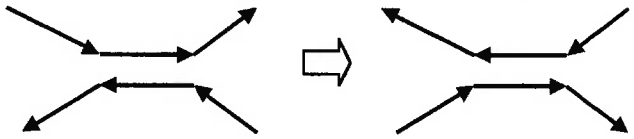


Fig. C.2. Ilustração de mudança de sentido das meia-arestas.

c) VerifiyRegions(TSolid<T> *s)

Parâmetro de Entrada:

s – ponteiro para sólido

No modelador de sólidos, o sólido pode ter regiões com mais de um "*shell*". Ou seja, além do "*shell*" externo, o sólido possui "*shells*" internos.

Esta função, verifica se existe alguma região do sólido em que o "*shell*" externo está com volume negativo. Se encontrado, este "*shell*" negativo deve ser transformado em "*shell*" interno de uma região com "*shell*" externo adequado. A procura deste "*shell*" externo adequado deve parar, quando for encontrado um "*shell*" que engloba o "*shell*" negativo.

```
d) BooleanSolid (       TSolid<T> *s1,
                        TSolid<T> *s2)
```

Parâmetros de entrada:

s1 – ponteiro para o primeiro sólido;

s2 – ponteiro para o segundo sólido;

Função principal do algoritmo. Ela realiza a chamada de outras funções auxiliares.

No decorrer do algoritmo, várias listas com identificadores de elementos geométricos são utilizadas para guardar informações:

- **ListFace1** – guarda os identificadores das faces que deverão ser removidas do primeiro sólido;
- **ListFace2** - guarda os identificadores das faces que deverão ser removidas do segundo sólido;

- **`ListEdge1`** - guarda os identificadores das arestas que não devem ser removidas do primeiro sólido;
- **`ListEdge2`** - guarda os identificadores das arestas que não devem ser removidas do segundo sólido;
- **`ListEdge`** – guarda uma lista formada por uma classe **`DEdge`** que possui dois identificadores que correspondem a arestas do primeiro e segundo sólido que possuem vértices com coordenadas iguais;

Estas listas foram escritas utilizando-se o "*container SET*" do STL. Este "*container*" ordena os elementos no momento de inserção de elementos. Deste modo, as listas estão sempre em ordem crescente em relação ao número de identificação e não existe elemento repetido na lista.

No primeiro passo, devem ser encontrados os pontos e segmentos de reta de intersecção entre os dois sólidos (para estes serem transformados em vértices e arestas). Esta é a etapa que exige maior tempo de processamento, pois devem ser verificadas as intersecções de todas as faces do primeiro sólido com todas as faces do segundo sólido. Na atual implementação, é verificado se a face **`F1`** (do primeiro sólido) e a face **`F2`** (do segundo sólido) possuem normais não colineares. Caso as faces satisfaçam este requisito, realiza-se uma verificação utilizando a função **`f2cutf1_`** para retirar os casos em que as faces tem normais não colineares, mas estão afastadas uma do outra (Figura C.3).

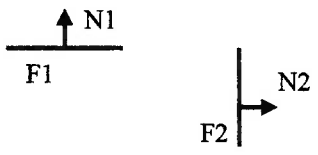


Fig. C.3. Apesar das faces **`F1`** e **`F2`** possuirem normais diferentes, elas estão muito afastadas uma da outra, o que impede a existência de um ponto de intersecção.

Se as faces não estiverem muito afastadas, é feita a chamada da função **`FaceIntersection`**, que determina os pontos e segmentos de reta de intersecção entre as duas faces e os transforma em elementos geométricos da estrutura de dados: vértices e arestas, conforme seja necessário.

Em seguida, utiliza-se a função **SetFace** para determinar quais faces devem ser removidas. É nesta função que está o algoritmo de classificação de diedros, que foi baseado no trabalho de Chiyokura (1988). Duas listas são formadas neste processo: **ListFace1** e **ListFace2**.

No passo seguinte, deve-se determinar o conjunto de arestas que não podem ser removidas quando ocorrer a remoção das faces. Logo, **ListEdge** deve ser dividida em **ListEdge1** (arestas do primeiro sólido) e **ListEdge2** (arestas do segundo sólido). Durante este processo, pode-se verificar se as duas meia-arestas que formam a aresta em questão, são vizinhas a faces que devem ser removidas (que estão em **ListFace1** e **ListFace2**). Em caso afirmativo, significa que, apesar da aresta estar na lista de arestas que não deve ser removida, neste momento, esta aresta em realidade pode ser removida . O exemplo na Figura C.4 demonstra este caso: durante a aplicação de operação booleana de união entre os dois sólido **S1** e **S2** as arestas **E1** e **E2** são criadas e inseridas em **ListEdge**; em seguida, o algoritmo verifica que **E1** é vizinha das faces **F1** e **F2**; o mesmo ocorre para **E2** que é vizinha de **F2** e **F3**; logo, **E1** e **E2** são retirados de **ListEdge** para que nos passos seguintes, estas arestas possam ser removidas, para criar uma única face não plana.

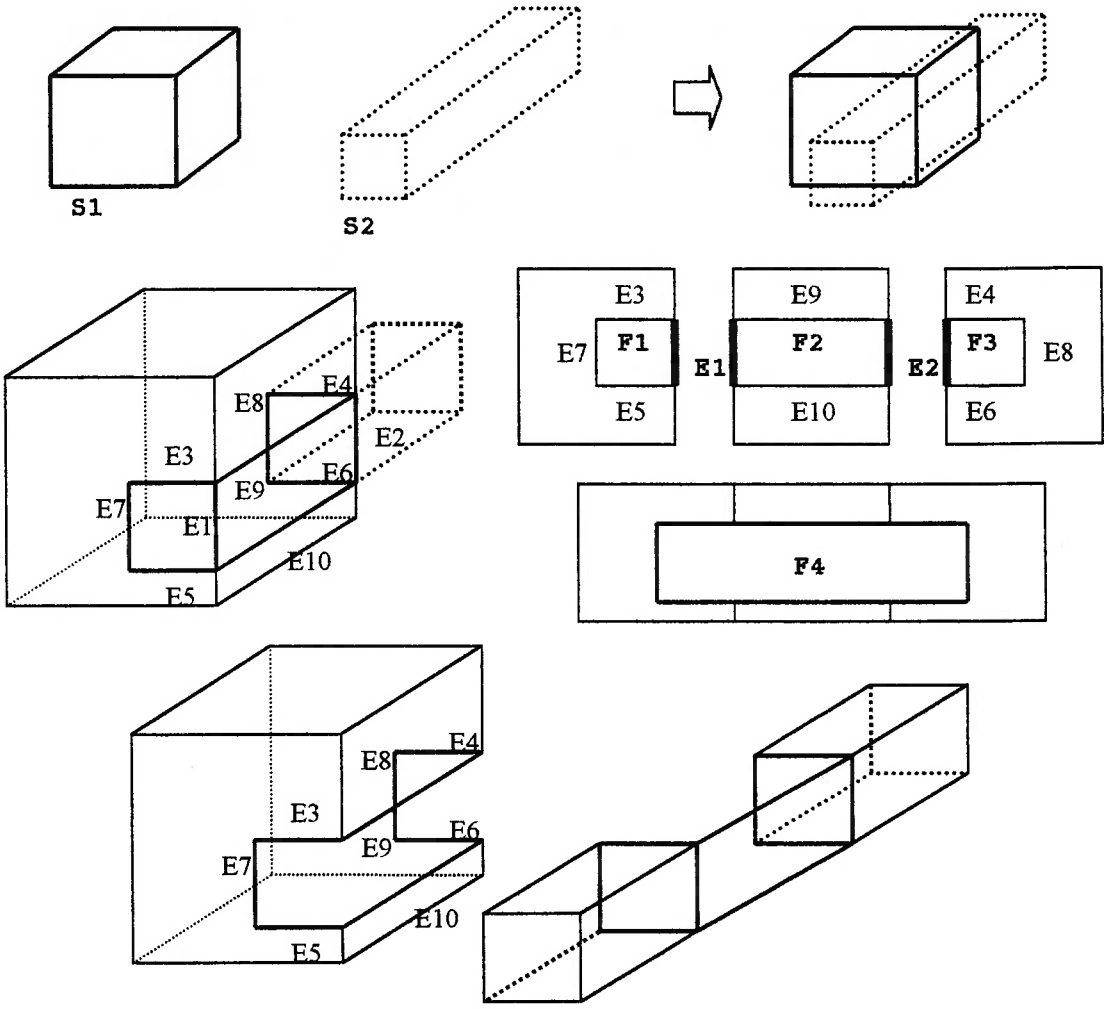


Fig. C.4. Exemplo de verificação de arestas que contornam faces que devem ser removidas.

Agora, faces indesejáveis devem ser removidas, utilizando a função **`KillFaces`**. Envia-se como parâmetro, o ponteiro para o sólido, a lista que contém arestas que não devem ser removidas para este sólido e a lista de faces que devem ser removidas deste sólido. Duas chamadas desta função são realizadas, uma para cada sólido.

Deste ponto, são realizados ajustes nos dois sólidos (alguns destes ajustes foram implementados no algoritmo de corte) para que os sólidos estejam adequados para realizar a colagem. O primeiro ajuste é a remoção de arestas ponte. É verificado se ainda existe alguma face que, apesar de ter seu identificador na lista de faces que devem ser removidas, ainda está na estrutura do sólido. Se for encontrada alguma

destas faces, é chamada a função **KillBridgeEdge** (esta função está descrita no operador de corte).

O próximo ajuste é verificar se existe a necessidade de criar novos "*shells*". Chama-se a função **VerifyLoops_Shell2** (esta função está descrita no operador de corte).

O próximo ajuste remove regiões nulas. Uma região nula possui: "*shell*" nulo com nenhum vértice ou nenhum "*shell*". Regiões nulas podem ser criadas pelo ajuste de criar novos "*shells*" (que também cria novas regiões).

Em seguida, os sólidos tem sua estrutura de dados passadas por uma limpeza, com o uso da função **TCleanSolid**.

O penúltimo passo agora é colar os dois sólidos usando a função **TGlue**. O motivo de tantos ajustes e limpeza é para que a função de colagem funcione corretamente. A topologia da intersecção entre os sólidos precisa estar completamente igual para que a colagem ocorra, caso contrário, o algoritmo não conseguirá colar os dois sólidos. Para ganhar desempenho, esta função **TGlue** foi modificada de modo a só considerar as faces que ainda existem e que deveriam ter sido removidas.

E finalmente, é realizada uma limpeza no sólido resultante.

```
e) f2cutf1_ (     TFace<T> *f,
                  tnVector<T,4> &n)
```

Parâmetros de Entrada:

f – ponteiro para a face que está sendo analisada;

n – equação do plano que pode ou não cortar a face **f** ;

Esta função é semelhante a função **f2cutf1** utilizada no algoritmo de corte de sólido. Ela retorna verdadeiro, caso a face **f** seja cortada pelo plano **n**.

```
f) FaceIntersection (  TFace<T> *f1,
                       TFace<T> *f2,
                       tnVector<T,4> &n1,
                       tnVector<T,4> &n2)
```

Parâmetros de Entrada:

`f1` – ponteiro para face;

`f2` – ponteiro para face;

`n1` – equação da face `f1`;

`n2` – equação da face `f2`.

Esta função obtém os pontos de intersecção entre as faces `f1` e `f2`, cria vértices e arestas e armazena estas informações em `ListEdge`. Para este fim, utiliza-se uma lista auxiliar para armazenar os pontos de intersecção: `PointList`.A determinação dos pontos de intersecção é feita pela função `PointIntersection`.

Os pontos em `PointList` são ordenados pela função `SortPoint`.

Utilizando o conjunto ordenado de pontos de intersecção, devem ser criadas as arestas de intersecção. Para cada ponto de intersecção, analisa-se a sua classificação: se já existe um vértice com as coordenadas do ponto, se o ponto está numa aresta, ou se o ponto está no interior de uma face (Figura C.5 a,b e c).

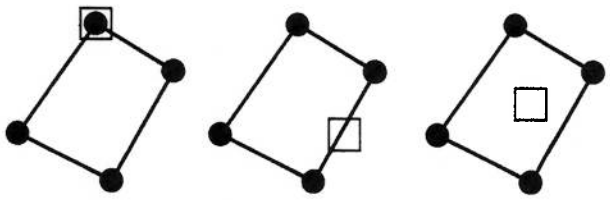


Fig. C.5. (a) vértice com coordenadas do ponto; (b) ponto na aresta (c) ponto no interior do laço;

Se o ponto de intersecção estiver sobre a aresta (Figura C.5b), então, a função `EdgeIntersection` irá criar um novo vértice na aresta em questão (Figura C.6).

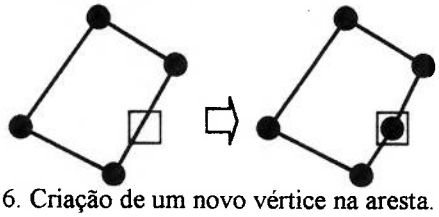


Fig. C.6. Criação de um novo vértice na aresta.

No segundo estágio, unem-se os vértices de intersecção para formarem arestas de intersecção. Utiliza-se a função `LinkIntersection` que retorna o ponteiro para a aresta de intersecção.

Se os dois ponteiros para as duas arestas de intersecção (uma para cada face) forem obtidos, então esta informação é guardada em `ListEdge`. Do contrário,

ocorreu algum erro, pois é improvável existir uma aresta de intersecção em apenas uma das faces.

Repete-se este processo até chegar ao final de **PointList**.

No final, **PointList** é removido pois já não é mais necessário.

```
g) EdgeIntersection(     tnVector<T,4> &p,
                         int &a,
                         THalfedge<T> **hd)
```

Parâmetros de Entrada:

p – coordenadas do ponto de intersecção

a – situação do ponto: 0 – ponto fora da face, 1 – ponto no interior da face, 2 – ponto na aresta, 3 – ponto no vértice;

hd – ponteiro para meia-aresta que tem ligação com o vértice de intersecção.

Esta função só atua no caso em que **a** for igual a 2 (ponto na aresta). Nesta situação, aplica-se o operador de Euler **MEV** para criar um novo vértice (Figura C.6). A variável **a** tem seu valor alterado para 3 (ponto no vértice). O ponteiro **hd** fica direcionado para a meia-aresta que tem ligação com o novo vértice criado.

Nos outros casos, retorna falso.

```
h) LinkIntersection (   TFace<T> *f,
                        int s01,
                        int s02,
                        THalfedge<T> *he1,
                        THalfedge<T> *he2,
                        tnVector<T,4> p1,
                        tnVector<T,4> p2)
```

Parâmetros de Entrada:

f – ponteiro para a face;

s01 – situação do primeiro ponto de intersecção;

s02 – situação do segundo ponto de intersecção;

he1 – ponteiro para a meia-aresta que tem ligação com primeiro vértice de intersecção;

he2 – ponteiro para a meia-aresta que tem ligação com segundo vértice de intersecção;

p1 – primeiro ponto de intersecção;

p2 – segundo ponto de intersecção;

Esta função junta dois vértices de intersecção, criando uma nova aresta e talvez uma nova face.

Podem ocorrer os seguintes casos:

- **s01 = 1 e s02 = 1** (ambos no interior da face)

  Deve-se criar um novo vértice na posição **p1**, com a aplicação do operador de Euler **MEV**. Em seguida, cria-se um novo laço, aplicando-se o operador de Euler **KEMR**. Finalmente, cria-se um novo vértice na posição **p2**, novamente com a aplicação do operador de Euler **MEV** (Figura C.7). Retorna o ponteiro para a nova aresta.

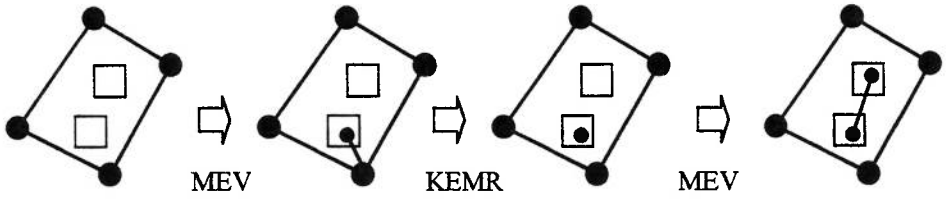


Fig. C.7. Rotina para quando ambos os pontos estiverem no interior da face.

- **s01 = 3 e s02 = 1 ou s01 = 1 e s02 = 3** (um vértice na face e outro no interior da face)

  Apenas aplica-se o operador de Euler **MEV** para criar um vértice na posição no interior da face com o vértice disponível (Figura C.8). Retorna o ponteiro para a nova aresta.

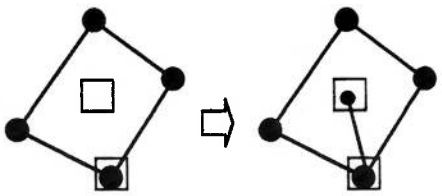


Fig. C.8. Criando um novo vértice e uma nova aresta que une este novo vértice com um vértice já existente.

- **s01 = 3 e s02 = 3** (ambos são vértices sobre o contorno da face)

  Este caso se subdivide em mais casos. É necessário o auxílio da função **ExistFace** (utilizada na operação de corte de sólidos). Conforme o resultado de função **ExistFace**, os vértices de intersecção podem estar:

a) em laços diferentes: aplica-se o operador de Euler **MEKR** e o ponteiro para a nova aresta (Figura C.9) é enviado como parâmetro de retorno desta função;

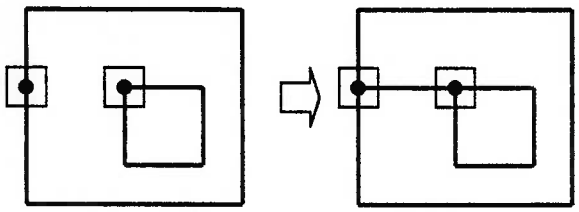


Fig. C.9. Caso como laços diferentes.

b) já existe aresta: retorna o ponteiro para a aresta existente;

c) pertencem ao mesmo laço e existe material entres os vértices: aplica-se o operador de Euler **MEF**, de modo que a nova face tenha a mesma normal que a face antiga (Figura C.10). Verificam-se os laços internos com o auxílio das funções **VerifyLoops** (da operação de corte de sólidos) e **VerifyLoopsBoolean**. Retorna o ponteiro da nova aresta;

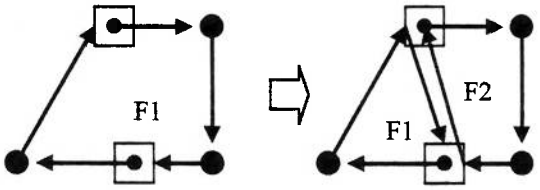


Fig. C.10. Nova face criada, com mesma orientação que a antiga.

É importante destacar que se a nova face tiver a orientação invertida em relação à face antiga, o algoritmo de classificação de diedros não funcionará corretamente pois serão fornecidas normais com sentidos incorretos.

```
i) PointIntersection ( TFace<T> *f,
                       TFace<T> *f2,
                       tnVector<T,4> n,
                       list< tnVector<T,4> > &PointList)
```

Parâmetros de Entrada:

f – ponteiro para a face que será analisada;

f2 – ponteiro para a face do outro sólido;

n – equação da face **f2**;

PointList – lista com os pontos de intersecção.

Esta função é semelhante à utilizada na operação de corte de sólidos. Entretanto, é necessário ter certeza que o ponto de intersecção existe na face **f** e na face **f2**, antes de inserir este ponto na lista de pontos de intersecção **PointList**.

Percorre-se as meia-arestas da face **f** e utilizando a função **intfe** (da operação de corte de sólidos), determina-se se existe ou não o ponto de intersecção entre a meia-aresta e o plano **n**. Se encontrado um ponto de intersecção, verifica-se a situação do ponto na face (utilizando a função **contfv**) para a face **f2**. Se o ponto estiver no interior da face ou numa das arestas ou vértices, então, este ponto é inserido na lista **PointList**.

Neste algoritmo é possível considerar a inclusão de testes de verificação e de imposição de topologia.

```
j) SortPoint(list< tnVector<T,4> > &PointList)
```

Parâmetros de entrada:

PointList – lista que guarda os pontos de intersecção;

É realizada uma ordenação dos pontos de modo semelhante à da operação de corte de sólidos.

```
k) VerifyLoopsBoolean( TFace<T> *f1,
                       TFace<T> *f2)
```

Parâmetros de Entrada:

f1 – ponteiro para a face que foi cortada;

f2 – ponteiro para a nova face criada.

Esta função é semelhante à função **VerifyLoops** (operação de corte de sólidos). Ela é um ajuste que deve ser adotado para situações que não ocorrem no corte de sólidos, mas podem ocorrer em operações booleanas. O exemplo na Figura C.11 mostra uma face com um laço interno. Durante a etapa de criação de novos vértices, arestas e faces, pode ser necessário criar um novo laço nesta face. Como a função **VerifyLoops** não prevê situações em que os laços internos de uma face estão um dentro do outro, e, portanto, no interior do laço externo da face **f1**, não ocorre a mudança do laço para a face **f2**.

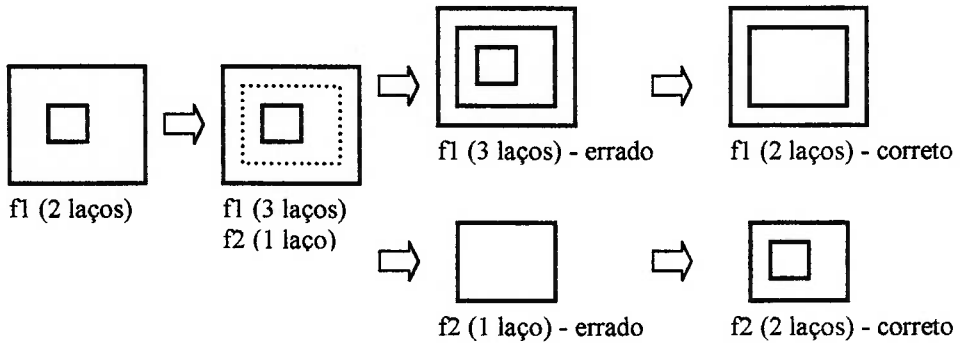


Fig. C.11. Correção para laços internos no interior de laços internos.

É verificado se o primeiro vértice de cada laço interno de `f1` está no interior de algum dos laços de `f1` utilizando-se a função `contlv`. Se estiver, este laço deve ser movido para a face `f2`.

`1) SetFace(void)`

Esta função realiza a classificação das faces do primeiro e do segundo sólido. Ela determina quais faces devem ser removidas.

A lista `ListEdge` contém as arestas de intersecção, o que permite obter as duas faces que formam a aresta. Estas são as faces a serem classificadas.

Utiliza-se uma classificação baseada no trabalho de Chiyokura (1988), em que as informações do diedro formado pelas faces da aresta de intersecção indicam se a face do outro sólido deve ser removida ou não. Na Figura C.12, tem-se os sólidos `S1` e `S2` e, ampliado, um ponto de intersecção (na verdade, uma aresta de intersecção vista de perfil), com as faces que compõem o diedro: `F11` e `F12` do sólido `S1`; e `F21` e `F22` do sólido `S2`.

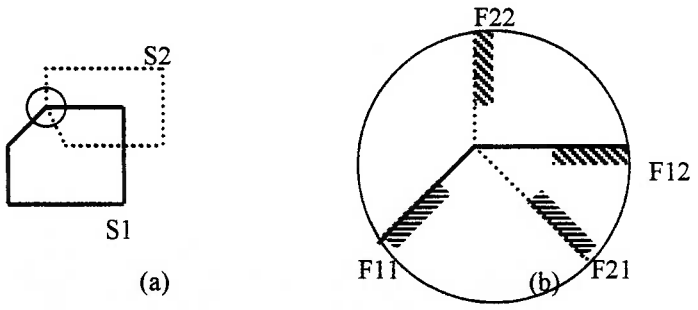


Fig. C.12 a) Sólido `S1` e `S2` b) Diedros dos sólidos `S1` e `S2`;

Para cada aresta de intersecção em um sólido existe outra correspondente no outro sólido. Analisa-se uma aresta por vez, que é confrontada com o diedro do outro sólido (Figura C.13 a,b,c e d)

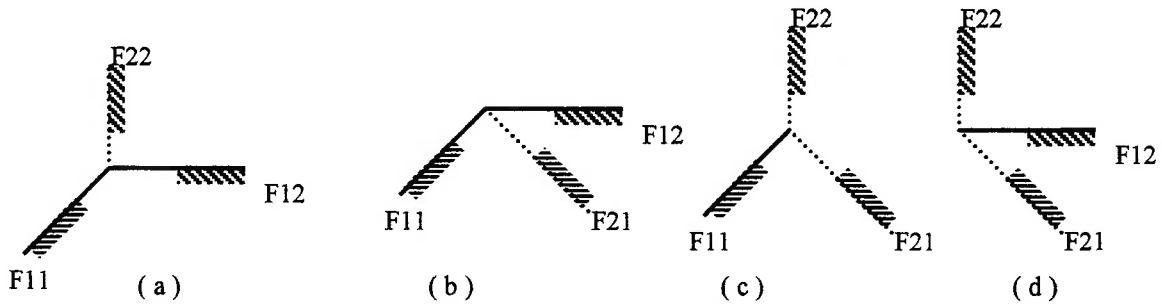


Fig. C.13. a) classifica **F22** baseado no diedro **F11/F12**; b) classifica **F21** baseado no diedro **F11/F12**; c) classifica **F11** baseado no diedro **F21/F22**; d) classifica **F12** baseado no diedro **F21/F22**.

No primeiro estágio da classificação, é verificado se existem faces paralelas com alguma outra face do diedro do outro sólido. Estas faces podem possuir normais de mesmo sentido (Figura C.14 b) ou opostas (Figura C.14 a). Se forem opostas, significa que estas faces irão fazer parte do interior do sólido resultante e portanto, devem ser retiradas. Se possuírem normais com mesmo sentido, significa que definem a superfície do sólido resultante e uma das faces deve permanecer. Por convenção, adotou-se que as faces do sólido **S1** permanecem e as do sólido **S2** são removidas nestes casos.

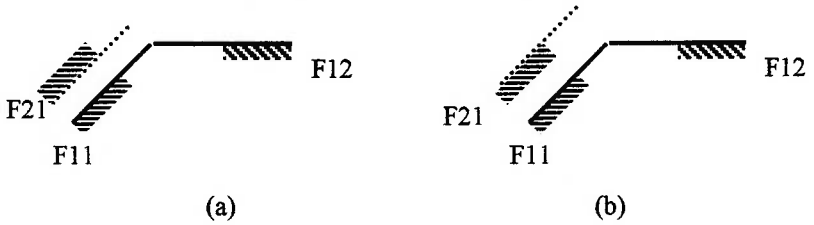


Fig. C.14. a) **F21** e **F11** são faces coplanares com normais opostas – ambas devem ser removidas; b) **F21** e **F11** são faces coplanares com normais de mesmo sentido – uma delas deve ser removida.

A função **Paralell_faces** classifica este tipo de faces.

Após classificar todas as faces paralelas com o outro diedro, classificam-se as faces restantes. Se ao confrontar uma face com o diedro do outro sólido, ela estiver contida no interior do diedro, significa que faz parte de uma área do sólido que deverá ser removida. Naturalmente, se não estiver no interior do diedro, então esta face faz parte da superfície do sólido resultante.

Para determinar se uma face está no interior de um diedro, são necessárias as seguintes informações: um vetor perpendicular à aresta de intersecção e coplanar à face – denominado **V** e as normais das faces do diedro. No exemplo representado na Figura C.15, a face **F22** está sendo classificada com base no diedro **F11/FC.**

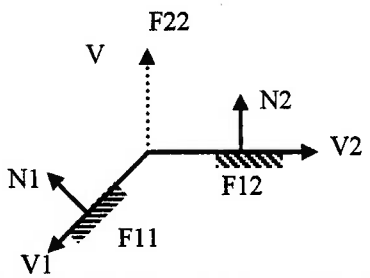


Fig. C.15. Classificação da face **F22** usando o diedro **F11/F12**.

Uma vez que os casos com faces paralelas já foram analisados, restam seis possíveis posições relativas entre a face que está sendo classificada e o diedro correspondente no outro sólido, representadas na Figura C.16.

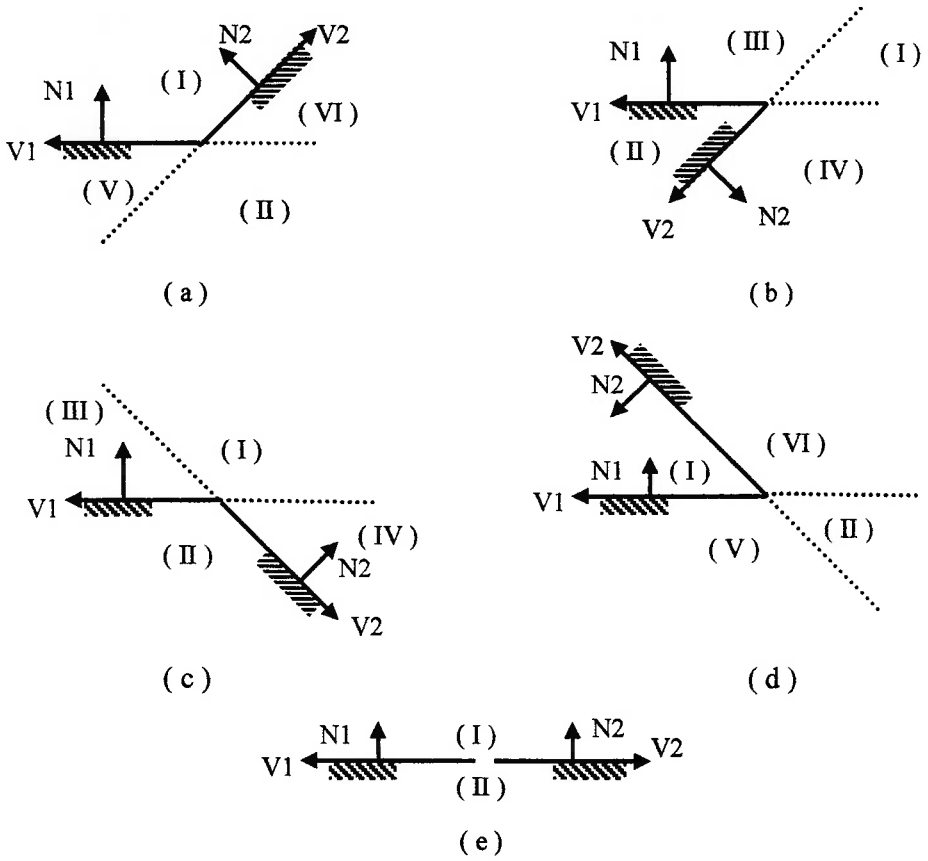


Fig. C.16. Situações possíveis de classificação de face baseado no diedro do sólido oposto.

As regiões que definem o interior do diedro correspondem às marcações **II**, **V** e **VI**. Utilizando o vetor **V**, **N1**, **N2**, **V1** e **V2**, pode-se determinar as condições para se estar nestas regiões (tabela I):

Tabela I – Condições para faces estarem no interior do diedro

(II)	(V)	(VI)
$N1 \cdot V < 0$	$N1 \cdot V < 0$	$N1 \cdot V \geq 0$
$N2 \cdot V < 0$	$N2 \cdot V \geq 0$	$N2 \cdot V < 0$
	$V2 \cdot N1 \geq 0$	$V2 \cdot N1 \geq 0$
	$V1 \cdot N2 \geq 0$	$V1 \cdot N2 \geq 0$

Para as regiões que definem o exterior do diedro (tabela II):

Tabela II – Condições para faces estarem no exterior de diedros.

(II)	(V)	(VI)
N1·V ≥ 0	N1·V ≥ 0	N1·V < 0
N2·V ≥ 0	N2·V < 0	N2·V ≥ 0
	V2·N1 < 0	V2·N1 < 0
	V1·N2 < 0	V1·N2 < 0

A função **Internal_Material** utiliza estas informações para classificar as faces.

```
m) Paralell_faces(      int &fn1,
                        int fn2,
                        bool flag,
                        bool flag1,
                        bool flag2)
```

Parâmetros de Entrada:

fn1 – identificador de face

fn2 – identificador de face

flag – indicador: **flag = verdadeiro**, então as normais das faces **fn1** e **fn2** são iguais e não opostas

Parâmetros de Saída:

flag1 – indicador: **flag1 = falso**, então não é preciso verificar face **fn1** nas próximas etapas

flag2 - indicador: **flag2 = falso**, então não é preciso verificar face **fn2** nas próximas etapas

Estão função trabalha com os casos em que as faces são paralelas. Deste modo, caso as normais das faces sejam iguais (Figura C.17 a), então, a face do primeiro sólido de ser mantida. Do contrário, as duas faces são opostas (Figuras C. 17 b e c) e devem ser removidas (inseridas nas listas **ListFace1** e **ListFace2**), pois delimitam uma região sem material. As variáveis **flag1** e **flag2** devem ser mudadas para falso, pois estas faces já foram classificadas e não precisam ser analisadas pela função **Internal_Material**.

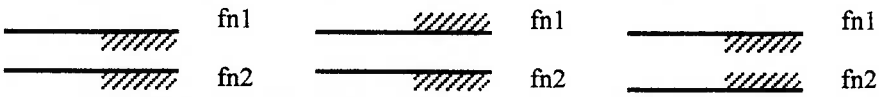


Fig. C.17. a) Normais iguais b) e c) normais opostas das faces **fn1** e **fn2**.

```
n) Internal_Material(   tnVector<T,4> V,
                        tnVector<T,4> n1,
                        tnVector<T,4> n2,
                        tnVector<T,4> v1,
                        tnVector<T,4> v2)
```

Parâmetros de Entrada:

V – vetor coplanar da face sendo classificada

n1 – normal de face **F1** (diedro)

n2 – normal de face **F2** (diedro)

v1 – vetor coplanar da face **F1** (diedro)

v2 – vetor coplanar da face **F2** (diedro)

Parâmetros de Saída:

Verdadeiro – a face deve ser removida.

Os casos restantes da classificação de faces que não envolvam faces paralelas, são analisados nesta função.

São necessários no máximo três verificações: o valor de **s1** = **N1·V**; o valor de **s2** = **N2·V** e **s3** = **N1·V2** ou **s3** = **N2·V1**. Conforme o valor de **s1**, **s2** e **s3**, pode-se determinar em qual região está a face em questão, utilizando-se as informações das tabelas I e II.

```
o) KillFaces(     TSolid<T> *s,
                  set< int,less<int> > &ListFace,
                  set<int, less<int> > &ListEdge )
```

Parâmetros de Entrada:

s – ponteiro para o sólido;

ListFace – lista que guarda os identificadores das faces que devem ser removidas;

ListEdge - lista que guarda os identificadores das arestas de intersecção que não devem ser removidas;

O processo para remover faces foi dividido em duas etapas: a primeira consiste em encontrar um ponteiro para a face que deve ser removida; a segunda

remove recursivamente as faces adjacentes a face que deve ser removida, desde que nenhuma das arestas de intersecção que estão contidas em **ListEdge** sejam removidas.

```
p) KillFace(      TFace<T> *f,
                  set< int,less<int> > &ListFace,
                  set<int, less<int> > &ListEdge)
```

Parâmetros de Entrada:

f – face que deve ser removida;

ListFace – lista que guarda os identificadores das faces que devem ser removida;

ListEdge - lista que guarda os identificadores das arestas de intersecção que não devem ser removida;

Esta função realiza recursivamente a remoção de faces adjacentes à face **f**.

Como primeiro passo, todas as faces adjacentes à **f** são verificadas. Se a aresta compartilhada entre a face adjacente e a face **f** não estiver em **ListEdge** e, a face adjacente não estiver em **ListFace** então, esta face pode ser removida (o identificador da face adjacente é inserido em **ListFace**). Chama-se recursivamente esta mesma função, enviando como parâmetro esta face adjacente.

Quando todas as faces adjacentes forem marcadas para serem removidas passa-se para o segundo passo: remover a face. Este processo é feito de maneira semelhante ao realizado na operação de corte. Vale lembrar que as arestas armazenadas em **ListEdge** devem ser mantidas.

ANEXO D - ARMAZENAMENTO DE DADOS

Após o modelo sólido ter sido finalizado, existem duas formas de armazená-lo para futura utilização. A primeira consiste em guardar em arquivo com os passos que foram seguidos para se chegar no modelo, ou seja, guardando a seqüência de comandos (ex.: "*box*" e "*bool*"). Exemplo:

```
box 0 0 0 10 10 20
rot 1 0 0 45
move 1 -15 0 0
box 0 0 0 10 10 20
rot 2 0 0 45
move 2 15 0 0
box 0 0 -2.5 30 10 15
bool 0 1 3
bool 1 2 0
```

A segunda é pelo armazenamento do sólido final e seus elementos. A principal desvantagem do primeiro tipo de arquivo de armazenamento é o tempo gasto para realizar as operações, pois está sendo reprocessada a história. É interessante armazenar as informações em uma forma próxima da estrutura de dados utilizada para que seja possível carregar rapidamente o sólido. Assim foi definido um arquivo texto em que é possível representar os seguintes elementos:

- Sólido: Representado pela letra **S** e pelo número de identificação do sólido.

  Ex.: Sólido n° 1

  S 1

- Região: Representado pela letra **R** e pelo número de identificação da região.

  Ex.: Região n° 1

  R 1

- "Shell": Pode ser representado pela letra **D** (para "*shell*" interno) ou **T** ("*shell*" externo).

  Ex.: "*Shell*" externo n° 1 e "*Shell*" interno n° 2

  T 1

  D 2

- Vértice: Representado pela letra **V**, seguido do número de identificação e as três coordenadas do vértice em unidade de ponto flutuante (para economia de espaço). Nota-se que, ao armazenar nesta forma de arquivo, o sólido perde o seu conteúdo em aritmética intervalar.
  Ex.: vértice n° 1 com coordenadas (1, 0, 0)
  V 1 1 0 0
- Aresta: Representado pela letra **E**, seguido do número de identificação, em seguida tem-se o número de identificação da primeira meia-aresta, da segunda meia-aresta, o número de identificação do vértice ligado à primeira meia-aresta e o número de identificação do vértice ligado à segunda meia-aresta.
  Ex.: aresta n° 1 formada pela meia-aresta n° 3 e n° 4, usando vértices n° 5 e n° 6
  E 1 3 4 5 6
- Face: Representado pela letra **F** e pelo número de identificação da face
  Ex.: Face n° 1
  F 1
- Laço: Pode ser representado pela letra **X** (laço externo) ou por **L** (laço interno)
  Ex.: Laço externo n° 1 e laço interno n° 3
  X 1
  L 3
- Meia-aresta: Representado pela letra **H**, seguido pela identificação da meia-aresta e pela identificação da aresta a qual pertence a meia-aresta.
  Ex.: Meia-aresta n°3 que pertence a aresta n° 1
  H 3 1

Para que esta informação seja interpretada adequadamente, é necessário que eles sejam apresentadas segundo uma seqüência bem definida no arquivo. Os elementos devem ser definidos na seguinte seqüência: o sólido, as regiões do sólido contendo cada um a sua lista de "*shells*". Cada "*shell*" deve ter abaixo a sua lista de vértices, lista de arestas e lista de faces. Cada face deve ter sua lista de laços e a sequência de meia-arestas que o definem. Abaixo temos um exemplo um sólido

armazenado, representando um poliedro com 4 vértices, 6 arestas e 4 faces - tetraedro:

```
S 1
R 1
T 1
V 1 0 5 0
V 2 4.33013 -2.5 0
V 3 -4.33013 -2.5 0
V 6 0 0 10
E 1 2 1 2 1
E 2 4 3 3 2
E 3 6 5 1 3
E 4 8 7 6 3
E 5 10 9 6 2
E 7 14 13 6 1
F 2
X 2
H 5 3
H 1 1
H 3 2
H 5 3
F 3
X 3
H 9 5
H 8 4
H 4 2
H 9 5
F 4
X 4
H 13 7
H 10 5
H 2 1
H 13 7
F 5
X 5
H 14 7
H 6 3
H 7 4
H 14 7
```

LISTA DE REFERÊNCIAS

AMMERAAL, L.,**STL for C++ programmers**, Wiley,1996.

CHIYOKURA, H., **Solid Modeling with DESIGNBASE: Theory and Implementation**, Addison Wesley Publishing Company, 1988.

GABRILOVICH, E., **Controlling the Destruction Order of Singleton Objects**, C/C++ Users Journal, outubro de 1999, 57p.

GLAESER, G. & STACHEL, H., **Open Geometry: OpenGL + Advanced Geometry**, Springer Verlag., 1998.

HOFFMANN, C.M., **Geometric and Solid Modeling: An Introduction**, Morgan & Kaufmann., 1989.

HU, C.Y.; PATRIKALARIS, N.M., Ye, X., **Robust Interval Solid Modelling Part I: Representations**, Computer Aided Design 28(10), 1996, 807p.

HU, C.Y.; PATRIKALARIS, N.M., Ye, X., **Robust Interval Solid Modelling Part II: Boundary Evaluation**, Computer Aided Design 28(10), 1996, 819p.

KILGARD, M.J., 2000, **GLUT – OpenGL Utility Toolkit**, http://reality.sgi.com/mjk/glut3/, versão 3.7 beta.

LUTZ K., **Designing a Data Structure for Polyhedral Surfaces**. In Proc. of the 14th ACM Symp. on Computational Geometry, Minneapolis, Minnesota, junho de 1998, 146p.

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

RADEMACHER, P., **GLUI – User Interface to OpenGL**, http://www.cs.unc.edu/~rademach/glui/, versão 2.0 beta.

SUTTER, H., **Using auto_ptr Effectively**, C/C++ Users Journal, Outubro de 1999, 63p.

TSUZUKI, M.S.G., **Modelagem de Sólidos: Representação por Fronteira (B-rep)**, XI Congresso Brasileiro de Engenharia Mecânica, dezembro de 1991, 611P.

WOO, Mason et al, **OpenGL Programming Guide**, Addison – Wesley Publishing Company, 1998.

BIBLIOGRAFIA RECOMENDADA

http://nehe.gamedev.net/

http://home.europa.com/~keithr/crossroads/

http://www.cs.unc.edu/~kettner/

http://www.gamedev.net

http://www.gametutorials.com

http://www.nvidia.com

http://www.opengl.org

LUTZ, K., **Using Generic Programming for Designing a Data Structure for Polyhedral Surfaces**. Computational Geometry - Theory and Applications 13, Elsevier 1999, 65p.

RULE, K., **3D File Formats: A Programmers Reference**, Addison – Wesley Developers Press, 1996