5,7 (cinco e sete)

27/08/03

FERNANDO FERNANDES CHAVES
MARCOS TAKESHI KOBAYASHI

Reestruturação da Estrutura de Dados e Implementação de Algoritmos no Modelador de Sólidos Didático USPDesigner

Dissertação apresentada à
Escola Politécnica da
Universidade de São Paulo
para obtenção do título de
Engenheiro

Orientador:
Prof. Doutor
Marcos Sales Guerra Tsuzuki

São Paulo
2003

Agradecimentos

Agradecemos ao Prof. Doutor Marcos de Sales Guerra Tsuzuki pelo apoio e orientação.
Agrademos a Marcelo Shimada e Wang pela ajuda.

Agradecemos a nossos pais, familiares e amigos pela ajuda e apoio sem os quais não conseguiríamos concluir este trabalho.

Resumo

Neste trabalho será realizada uma remontagem da estrutura de dados do modelador de sólidos didático B-Rep USPDesigner, a fim de tornar sua implementação e expansão mais fáceis, melhorando o rendimento do modelador, com uma economia de memória e tempo nos processos de criação, modificação e exclusão de modelos. Para isso será utilizado o conceito de encapsulamento, para que certas partes instáveis dos objetos sejam "escondidas" ao mesmo tempo em que seus acionamentos fiquem implícitos em chamadas simples e seguras. Será também feito uso dos conceitos da linguagem de programação C++ e padrões de projeto. Primeiramente será feito um estudo da estrutura de dados atual do USPDesigner e aprofundaremos os conceitos da linguagem de programação C++. Após este estudo serão iniciadas as modificações na estrutura de dados. Seguindo então uma seqüência de alterações, sendo ela: operadores de Euler, operadores de alto nível, operadores complexos e rotinas de visualização, estas sendo pertencentes à biblioteca OpenGl.

Abstract

In this work, a re-assembly of the data structure of the didactic solids modeler USPDesigner will make its development and expansion easier, allowing better performance to the modeler, with memory and time economy in the processes of creation, editing and deletion of models. For that the concept of encapsulation will be used, so that some unstable parts of the objects are "hidden" but still used through simple and safe calls. Concepts of C++ programming language and project patterns will also be used. At first, an analysis on the actual data structure of USPDesigner and a deeper study at the C++ programming language concepts will be made. Then the data structure modifications will be made followed by a sequence of alterations, in order: Euler operators, high level operators, complex operators, and visualization routines, those belonging to the OpenGl library.

SUMÁRIO

LISTA DE FIGURAS

1. INTRODUÇÃO

Nos dias de hoje, temos que os modeladores de sólidos possuem uma grande importância para o desenvolvimento e estudo de novos produtos. Estes devem ter as informações e desenhos armazenados, conseguindo-se analisar suas atuações e funções através de simulações feitas por computador. Estes modelos além de servirem para simulações podem ser utilizados em computação gráfica para estudo de processos de fabricação e até para entretenimento, se utilizados para construção de jogos.

Tendo em vista estas aplicações fica evidente a necessidade de uma grande fidelidade na modelagem de elementos reais.

Iremos realizar nossos estudos através do modelador de sólidos USPDesigner, criado para fins acadêmicos. Apresentaremos:

- Estrutura de dados do modelador
- Operadores de Euler
- Operadores de alto nível
- Operações booleanas
- Visualização

2. USPDESIGNER

Modelador de sólidos do tipo B-Rep, implementado em Visual C++ 6.0, que utiliza o OpenGL para visualização wireframe ou shading.

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.

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 descrever como cada face está conectada às suas faces adjacentes, de maneira que um volume totalmente fechado seja definido. Em uma representação B-rep, esta informação está disponível explicitamente, ou seja, não é necessário realizar nenhuma comparação numérica.

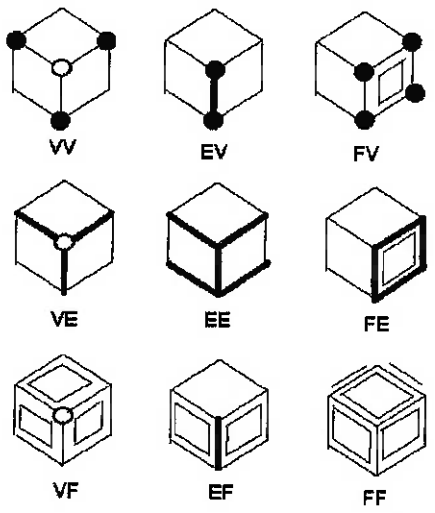


FIG. 1 RELAÇÕES DE ADJACÊNCIA ENTRE VÉRTICE (V), ARESTA (E) E FACE (F).

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.

No USPDesigner a estrutura de dados se baseia na aresta como elemento de referência. Para este tipo de estrutura se destacam a "winged-edge" e a meia-aresta (esta utilizada no USPDesigner).

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.

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 2 ilustra a representação meia-aresta.

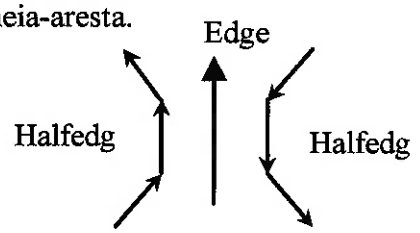


FIG. 2 REPRESENTAÇÃO DE UMA "HALFEDGE".

A estrutura do sólido completa utilizada na implementação do USPDesigner 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 2.3) – o sólido resultante possui duas regiões. O elemento Região possui um ou mais "shells";

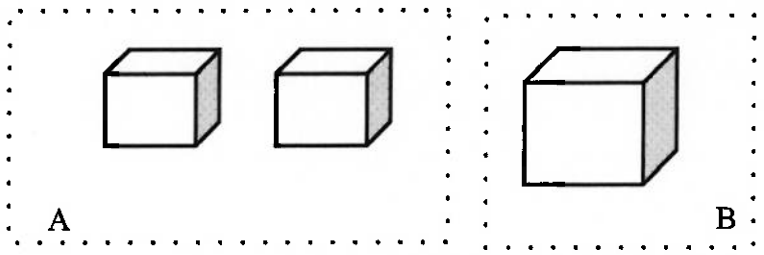
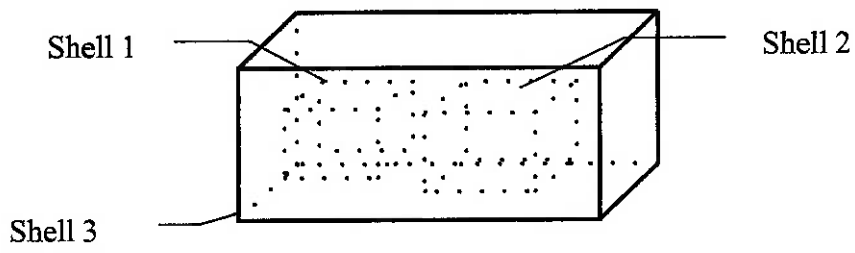


FIG. 3 SÓLIDO A, FORMADO POR 2 REGIÕES E SÓLIDO B, FORMADO POR UMA ÚNICA REGIÃO.

"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 2.4), 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;



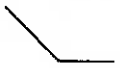


FIG. 4 A REGIÃO 1 É FORMADO POR 3 "SHELLS": O EXTERNO ("SHELL" 3) E DOIS INTERNOS: "SHELL" 1 E 2.

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;

Loop (laço): 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 5 mostra uma face com dois laços internos. O laço é formado por uma seqüência de meia-arestas;

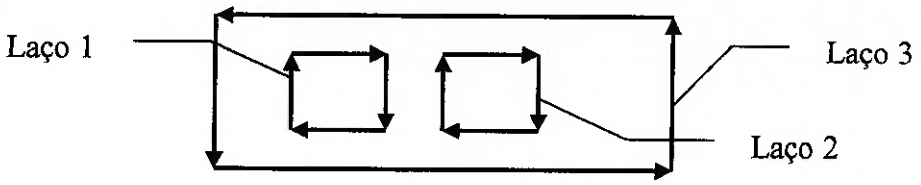


FIG.5 . FACE FORMADA POR 3 LAÇOS: LAÇO EXTERNO 3 E LAÇOS INTERNOS 1 E 2.

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 do laço;

Vértice ("Vertex"): elemento que possui uma tripla de coordenadas (x, y, z).

3. ESTRUTURA DE DADOS

A estrutura de dados do USPDesigner foi implementada criando-se classes para cada um dos elementos considerados como constituintes da estrutura do sólido. Sendo que estes elementos seguem a hierarquia representada na Figura 6.

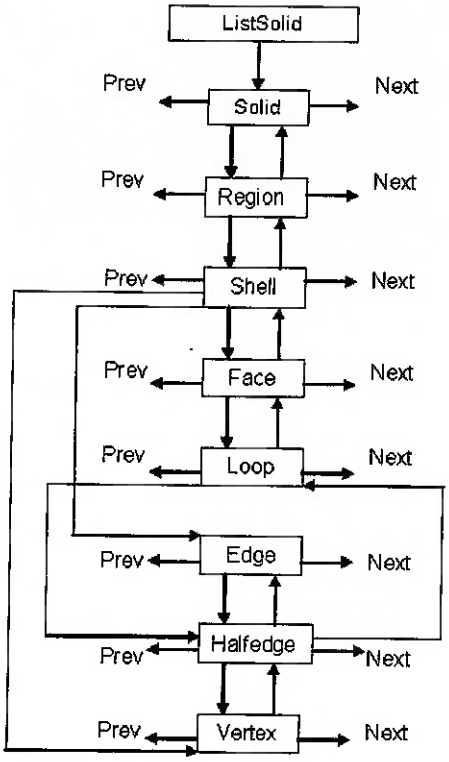


FIG. 6 HIERARQUIA DOS ELEMENTOS DA ESTRUTURA DE DADOS.

Temos:

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 7.

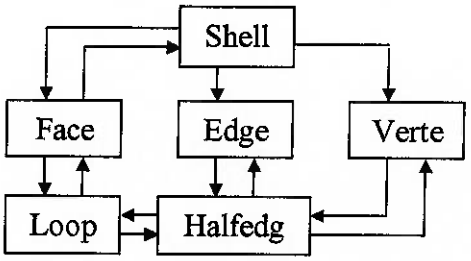


FIG. 7 REPRESENTAÇÃO DA IMPLEMENTAÇÃO.

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
}
```

4. ENCAPSULAMENTO DA ESTRUTURA DE DADOS

A estrutura de dados foi implementada de forma que foram-se criadas listas de elementos. No momento em que se criava um novo elemento, o programa criava uma cópia deste e armazenava-o na lista. Isto gerava uma dificuldade para se terem definidas as relações entre os elementos de hierarquias diferentes. Devido ao fato de que o novo elemento de hierarquia inferior se "comunicava" com a cópia do elemento de hierarquia superior e as cópias dos de mesma hierarquia.

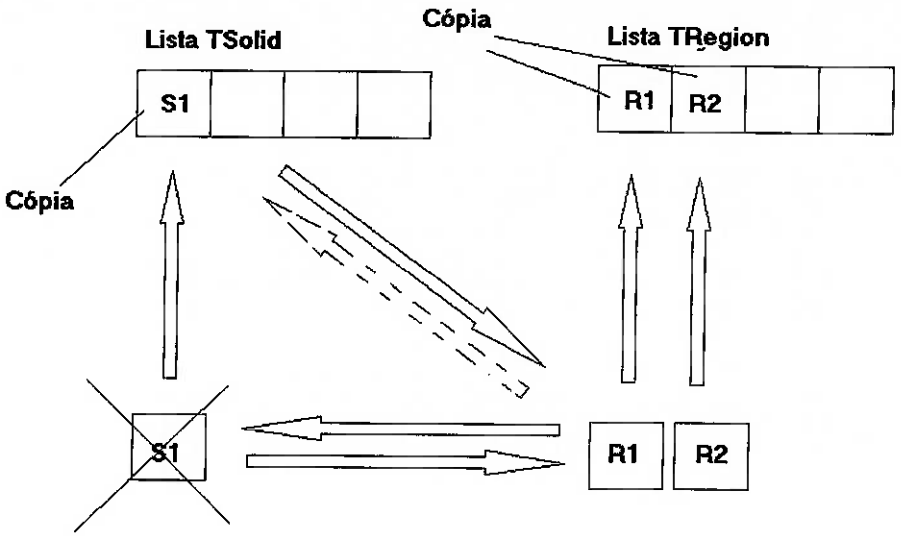


FIG. 8 COMUNICAÇÃO ENTRE OS ELEMENTOS NA ESTRUTURA ANTIGA.

Para aperfeiçoar esta implementação passou-se a utilizar listas de ponteiros e não mais listas de elementos. Estes ponteiros são os endereços dos elementos. Então cada elemento agora possui um ponteiro como referência. Isto gerou uma maior facilidade no relacionamento entre os elementos, pois agora eles se "comunicam" com as listas apenas para obterem os endereços dos outros elementos.

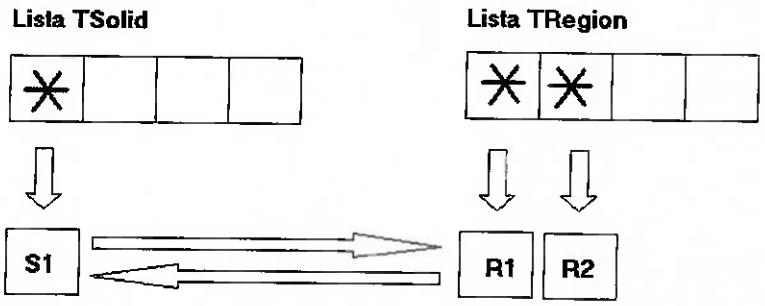


FIG. 9 COMUNICAÇÃO ENTRE OS ELEMENTOS NA ESTRUTURA ATUAL.

Modificou-se a estrutura de dados no seu nível mais baixo. Para isto foram alteradas, acrescidas ou retiradas algumas funcionalidades das classes que definem a estrutura do sólido.

Podemos destacar entre as alterações:

A inclusão das classes na estrutura (exemplo da classe Region):

```
class iterator : public CSmartIterator<TRegion<T> >
class iterator : public CSmartIterator<TShell<T> > {
        public:
          iterator() {};
          iterator(const list<TShell<T> *>::iterator it) :
          CSmartIterator<TShell<T> >(it) {};
          ~iterator() {};
      };
  iterator begin(void) { return  terator(rshells.begin()); }
  iterator end(void) { return iterator(rshells.end()); }

class const_iterator : public CSmartIterator<TRegion<T> >
   class const_iterator : public CSmartIterator<TShell<T> > {
        public:
          const_iterator() {};
          const_iterator(const list<TShell<T>
              *>::const_iterator it) :
          CSmartIterator<TShell<T> >(it) {};
          ~const_iterator() {};
      };
   const_iterator begin(void) const { return
        const_iterator(rshells.begin()); }
   const_iterator end(void) const { return
        const_iterator(rshells.end()); }
```

Esta inclusão possibilitou a modificação das funcionalidades da estrutura de dados, como por exemplo:

```
void listShell(void) {
     cout << "Shells: " << endl;
         list<TShell<T> >::iterator it = rshells->begin();
         for ( ; it != rshells->end() ; it ++)
             cout << (*it).No() << endl;
     };
     int delShell(TShell<T> *s) {
           int sn=s->No();
           return (this->delShell(sn) );
     }
     int delShell(const int &sn){
         list<TShell<T> >::iterator it = rshells->begin();
         for ( ; it != rshells->end() ; it ++)
             if (it->No() == sn) {
                 rshells->erase(it);
                 return SUCESS;
             }
         return ERROR;
     }
```

para

```
     void listShell(void) {
         iterator it = begin();
          cout << "Shells: " << endl;
         for ( ; it != end() ; it ++)
                     cout << it->getId() << " ";
         cout << endl;
     };
      int delShell(TShell<T> *s) { return (this->delShell(s->getId())
            ); } ;
     int delShell(const int &sn){
              iterator it = find(begin(), end(), TShell<T>(sn));
         if (it != end()) {
             rshells.erase(it.GetIterator());
             return 1;
         }
         return 0;
     }
```

Foi então implementado o "encapsulamento " da estrutura de dados utilizada no programa. Ou seja, prevenido o acesso não autorizado a determinados itens de informação ou a características de funcionamento das classes.

O critério chave aqui é separar as partes voláteis das partes estáveis da classe. Encapsulamento põe uma barreira em torno das partes voláteis, impedindo que outros módulos da estrutura acessem as partes voláteis da classe, outros módulos podem acessar apenas as partes estáveis da classe. Isso evita que outros módulos sejam afetados quando as partes voláteis da classe sofrerem alterações. Isto significa que o encapsulamento evita erros quando utilizamos toda a estrutura.

As partes voláteis são as implementações do módulo. Se o módulo é uma classe única, as partes voláteis são normalmente encapsuladas usando as palavras chave private: ou protected. Se o módulo um grupo compacto de classes, o encapsulamento pode ser usado para negar acesso a classes inteiras do grupo.

As partes estáveis são as interfaces. Projetar uma interface limpa e separar a interface da implementação, meramente permite ao usuário usar a interface. Mas ao encapsular (colocar em uma cápsula) a implementação força o usuário a usar a interface.

5. OPERADORES DE EULER

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 10). 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.

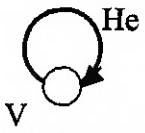


FIG. 10 REPRESENTAÇÃO DE UM VÉRTICE **V** COM UMA MEIA-ARESTA **HE** (SEM EXISTÊNCIA DE ARESTA).

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 11). A aresta ligará o vértice fornecido como parâmetro de entrada ao novo vértice que será criado;

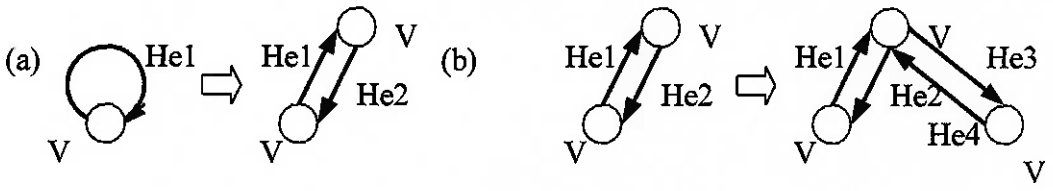


FIG. 11 (A) **MEV** CRIANDO UMA MEIA-ARESTA; (B) **MEV** CRIANDO DUAS MEIA-ARESTAS.

MEF/KEF (Make/Kill Edge Face) – O operador **MEF** cria uma aresta e uma face (Figura 10). A aresta ligará um par de vértices (fornecidos como parâmetros de entrada). No exemplo da Figura 12, 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**).

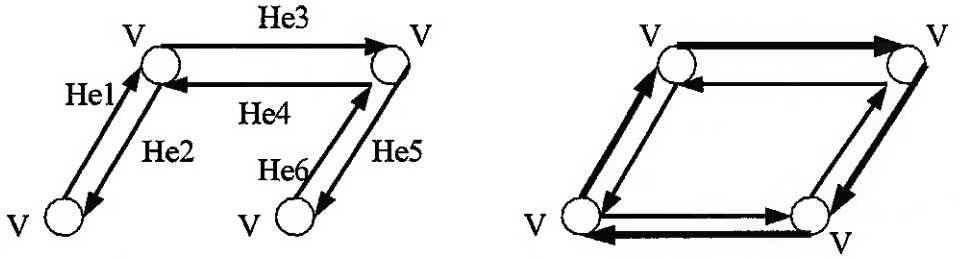


FIG. 12 EXEMPLO DE UTILIZAÇÃO DO OPERADOR **MEF**, FACE **F2** FOI CRIADA.

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 13, 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.

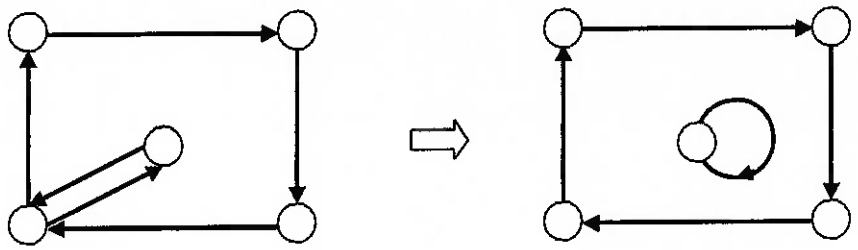


FIG. 13 EXEMPLO DE UTILIZAÇÃO DO **KEMR**.

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 14.

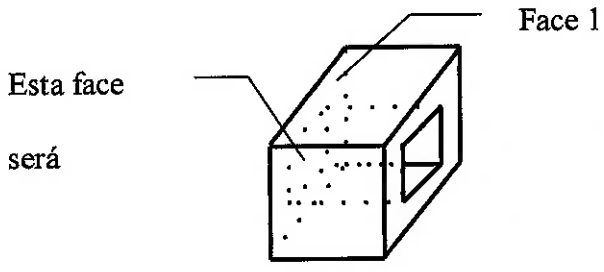


FIG. 14 EXEMPLO DE UTILIZAÇÃO DE **KFMRH**.

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 15). Naturalmente, todas as informações do "shell" a ser removido são transferidas para o outro "shell". Na Figura 15 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.

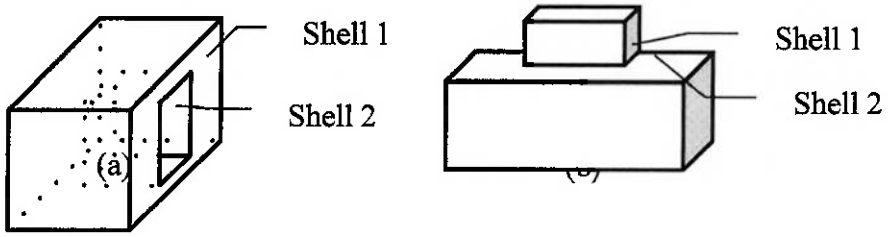


FIG. 15 (A) A UNIÃO DOS DOIS "SHELLS" GERA UM FURO NÃO PASSANTE (B) UTILIZAÇÃO DE **KSFMR**.

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

$$\mathbf{v - e + f = 2 * (s - h) + r}$$

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

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.

6. MODIFICAÇÕES NOS OPERADORES DE EULER

A reestruturação na implementação da estrutura de dados tornou necessária a modificação de todos os níveis de abstração. Por ser o nível intermediário entre a estrutura de dados e o terceiro nível, os operadores de Euler sofreram grandes modificações em sua implementação.

A principal se refere ao interfaceamento entre os níveis, ou seja, alterações nas interfaces utilizadas para comunicação entre os níveis. De modo que, as interfaces foram alteradas por mudanças na estrutura e pelo encapsulamento da estrutura.

Como a estrutura de dados está encapsulada, basta o conhecimento das interfaces vindas desta que são utilizadas nestes operadores, por exemplo, para se obter informações de algum dos elementos. Antes havia a necessidade de se conhecer a hierarquia dos elementos da estrutura de dados. Com o encapsulamento é desnecessária esta informação, tendo em vista que a visualização desta hierarquia na implementação está dentro da parte volátil da estrutura de dados. Ou seja, a obtenção de dados dos elementos da estrutura de dados, que antes era feita por funcionalidades que percorriam toda a hierarquia envolvida entre o elemento referência e o elemento requerido, agora é feita por uma úncia chamada que foi construida em todos os elementos da estrutura de modo que se esconda a hierarquia e o modo de implementação utilizado.

```
list <TFace<T>>::iterator f;
f-> SRegions() ->RSolid() ->No();
para
TShell<T>::iterator_f f;
f->getSolid() ->getId();
```

7. OPERADORES DE ALTO NÍVEL

Este é o nível mais alto de abstração, onde se interage com o usuário através de operações de construir, modificar, armazenar, copiar e mover sólidos. Neste nível, o sólido é descrito por um conjunto de operações de alto nível, que permite o interfaceamento com o usuário.

Temos os operadores locais que permitem que algumas características dos sólidos possam ser modificadas pelo usuário preservando as consistências topológicas e geométrica do local modificado. Por não realizarem alterações em áreas externas à localidade das características na que eles se aplicam, não são realizadas verificações sobre o sólido após a realização de uma operação local.

Um operador local pode ser considerado uma extensão dos operadores de Euler, visto que um operador local ao ser acionado pelo usuário definirá uma sequência de operadores de Euler que a implemente.

Como exemplo da rotina básica de um operador local, temos a cria arco de circunferência, que define a aproximação poligonal de um arco de circunferência, a partir de um ponto fornmecido. O algoritmo que implementa esta rotina pode ser definido utilizadno apenas o operador de Euler MEV. Para se definir um arco de 360° o último operador MEV deve ser substituído por um operador MEF.

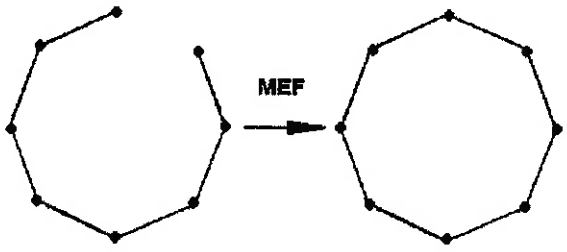


FIG.16 CRIA ARCO DE CIRCUNFERÊNCIA.

Outro exemplo de rotina básica de um operador local temos a extrusão translacional. Esta deve ser aplicada a uma face e será realizada segundo um sentido único. Podendo então ser gerado por este um cubo.

Os sólidos primitivos criados no modelador USPDesigner são o cubo, cilindro, cone, esfera, tubo e torus. Estes são gerados por sequêncis de operadores locais. Como exemplo a rotina de implementação de um cubo.

```
template <class T>
class TBOX {
     TMVSF<T>::instance().high(rn,dn,fn,vn,a,b,c);
     TMEV<T>::instance().high(sn,fn,vn,vn+1,a+x,b,c);
     TMEV<T>::instance().high(sn,fn,vn+1,vn+2,a+x,b+y,c);
     TMEV<T>::instance().high(sn,fn,vn+2,vn+3,a,b+y,c);
     TMEF<T>::instance().high(sn,vn,vn+3,fn,fn+1);
     MSD_lowMakeSweep<T>(fac, 0, 0, z);      //operador local de extrusão
     s->Color(0,0,1);
     s->Kind(BOX);
}
int MSD_lowMakeSweep(TFace<T> *fac, T dx, T dy, T dz) {
for ( ; it!=fac->end(); it++) {
          TMEV<T>::instance().low(scan, scan, maxv++, v->VCoord()[0]+dx,
           v->VCoord()[1]+dy,v->VCoord()[2]+dz);
          while (scan!=first) {
              TMEV<T>::instance().low(scan->Nxt(),     scan->Nxt(),     maxv++,     v-
>VCoord()[0]+dx,
                          v->VCoord()[1]+dy,v->VCoord()[2]+dz);
              TMEF<T>::instance().low(scan->Prv(), scan->Nxt()->Nxt(), maxf++); }
          TMEF<T>::instance().low(scan->Prv(), scan->Nxt()->Nxt(), maxf++);
}
```

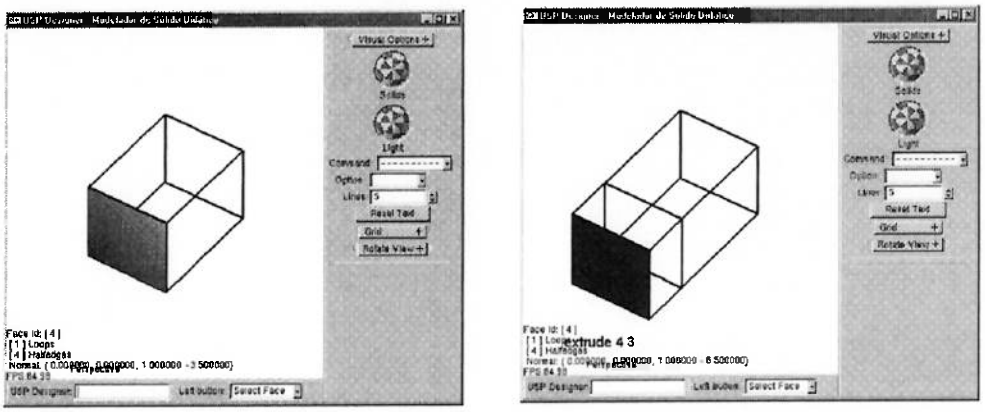


FIG.17 EXTRUSÃO DE UMA FACE DE UM CUBO.

8. OPERAÇÕES BOOLEANAS

Algoritmos para determinar operações entre dois objetos podem ser utilizados por modeladores de representação B-rep. Conceitualmente, estes algoritmos não são difíceis, mas sua implementação requer um substancial trabalho por várias razões. Considerar as várias posições espaciais de incidência das primitivas geométricas em três dimensões pode ser trabalhoso.

A não redundância das informações geométricas reduz a possibilidade de contradição dos dados e portanto aumenta a robustez. As coordenadas dos vértices e as equações dos planos dos modelos poliedrais são exemplos de redundâncias; pois, geralmente, os vértices não estão exatamente sobre os planos, mas podem estar deslocados por uma quantidade muito insignificante. Logo a forma e o contorno das faces podem não ser consistentes.

Para a implementação das Operações Booleanas a manipulação dos modelos dos objetos pode ser dividida em duas fases distintas: cálculos numéricos e modificações na estruturas B-rep. A fase de cálculos numéricos é representada por dois passos: cálculo de todos os pontos de intersecção entre os dois objetos e análise da vizinhança dos pontos de intersecção. A fase de modificação da estrutura B-rep é representada por quatro passos: geração de arestas nulas, análise dos pontos de intersecção e criação dos circuitos de arestas, recorte das arestas nulas e união dos devidos componentes para criar o objeto resultado da operação booleana.

Existem três tipos de Operação Booleana: união, subtração (ou diferença) e intersecção. Entretanto, a literatura pesquisada mostra que é necessário a implementação de apenas um tipo de Operação Booleana, pois as outras duas podem ser realizadas baseada na que for construída, utilizando o conceito de sólido negativo (indicado como "NS" – "Negative Solid").

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, invertemos a orientação do conjunto de meia arestas que formam a face.

Logo, dado dois sólidos: A e B, as operações booleanas podem ser expressas como:

União (operador básico): $A \cup B$

Subtração: $A - B = NS(NS(A) \cup B)$

Intersecção: $A \cap B = NS(NS(A) \cup NS(B))$

Subtração (operador básico): $A - B$

União: $A \cup B = NS(NS(A) - B)$

Intersecção: $A \cap B = A - NS(B)$

Intersecção (operador básico): $A \cap B$

União: $A \cup B = NS(NS(A) \cap NS(B))$

Subtração: $A - B = A \cap NS(B)$

Nesta implementação, escolheu-se como operador básico a Operação Booleana União.

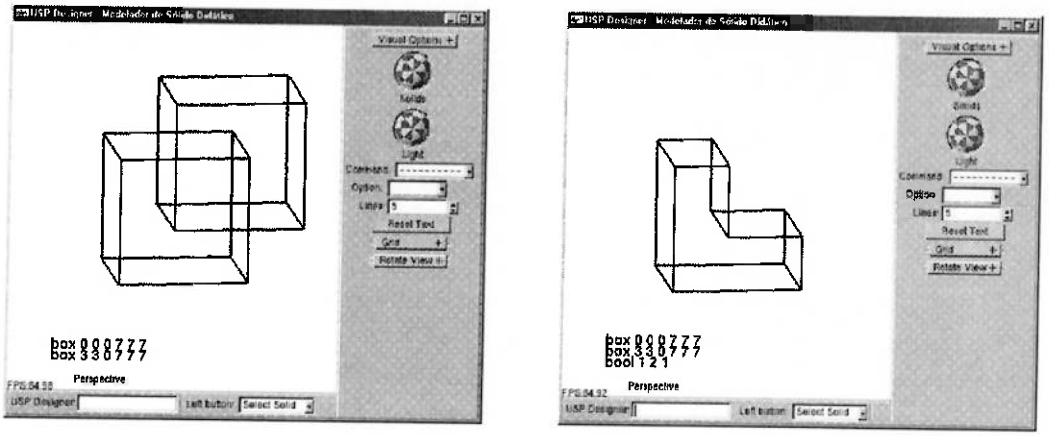


FIG. 18 OPERAÇÃO DE SUBTRAÇÃO DE DOIS CUBOS.

Podemos dividir a Operação Booleana União nos seguintes passos:

1. Determinar os pontos de intersecção entre as faces dos sólidos A e B;
2. Criar novos vértices, novas arestas e novas faces caso sejam necessários;
3. Classificar os elementos geométricos dos sólidos A e B e determinar quais devem ser eliminados da estrutura de dados pois não fazem parte do sólido resultante;
4. Eliminar os elementos geométricos que foram selecionados na etapa anterior;
5. Colar os sólidos A e B, de modo que formem uma única estrutura de dados;
6. Limpar dados redundates (caso ainda existam);

Na literatura, o que difere uma implementação de Operação Booleana de outra, é o conceito envolvido para realizar a etapa 3 – classificação. Em geral, as Operações Booleanas em modeladores "B-rep" consistem basicamente em determinar pontos de intersecção, criar novos vértices, apagar elementos desnecessários e juntar a parte restante do sólido A com o restante do sólido B. Mas, determinar o que deve ser apagado requer análise local do sólido. Devemos classificar os elementos em: pode ser e não deve ser retirado da estrutura de dados. A ênfase é para não apagar, ou seja, elementos geométricos redundantes são permitidos em etapas intermediárias da Operação Booleana, pois a etapa final consiste em limpar a estrutura de dados, retirando-os.

9. MODIFICAÇÕES NAS OPERAÇÕES BOOLEANAS

Como explicado no item anterior, a execução da operação booleana **união** consiste em 6 passos. Pode-se dizer que nos algoritmos de determinação de pontos de intersecção entre as faces, conseqüente criação de vértices e arestas, classificação dos elementos a serem eliminados e efetiva eliminação, as mudanças realizadas após a reestruturação da estrutura de dados limitam-se à adaptação do interfaceamento entre os níveis de programação (operadores de Euler e estrutura de dados) .

É no passo da colagem dos dois sólidos que a simplicidade da execução do programa obtida pelas mudanças estruturais se fazem bastante evidentes. Nesse passo, para cada region do sólido 2 é criada uma nova region no sólido 1 que para guardar suas shells. Quando "migrados" todos os shells de uma region, deleta-se a region, quando "migradas" todas as regions de um sólido, deleta-se o sólido. Depois faz-se uma busca no sólido restante por faces sobrepostas para que sejam removidas. Esses tarefas são feitas por classes do mesmo nível dos Operadores de Eules, por isso a simplificação se dá pelo mesmo motivos: facilidade em se trabalhar a estrutura de dados.

Na estrutura antiga, devido à lógica de criação e ligação dos elementos, muitos dados relativos a Ids precisavam ser guardados para criações de cópias nas listas de elementos do sólido 1 e posterior exclusão da estrutura do sólido 2. No programa novo há apenas uma mudança nos ponteiros e essa migração se dá com poucas linhas de comando.

Estrutura antiga:

```
        dn = d->No();
        rn = d->SRegions()->No(); // save region id to erase after
        TRegion<T> r(newr_Id);
        s1->addRegion(r);

        TRegion<T> *nr = s1->getRegion(newr);
        nr->addShell(*d);
        nr->RSout(nr->RShells() );

        TRegion<T> *rp = s->getRegion(rn );

        rp->delShell(dn);
        d = nr->getShell(dn );

      if (newd != -1) d->No(newd);
```

Estrutura nova:

```
TRegion<T>*r = new TRegion<T>(newr_Id);
s1->addRegion(r);
r->addShell(d);
r->RSout(d);
```

10. PROPRIEDADES

O modelador USPDesigner consegue obter quatro propriedades do sólido, são elas a Área, o Volume, O CG, e os Momentos de Inércia. Sendo que estes três últimos são calculados pela divisão o sólidop em tetraedros e a Área é calculada pela divisão das faces do sólido em triângulos. Estas propriedades nos serviram de base para comprovar a consistência do sólido, bem como sua correta formação.

Estas se relacionam unicamente com a estrutura de dados que ao ser encapsulada fez necessária uma reestruturação no intefaceamento entre estes módulos. A lógica de cálculo das propriedades continua a mesma, entretanto no que se refere a implementação foi alterada.

```
list<TFace<T> >::iterator it = d->SFace_it();
list< TLoop<T> >::iterator it = f->FLoops_it();
for ( ; it!=f->FLend_it(); it ++ )
// exemplo de alterações de implementação no volume
TShell<T>::iterator_f it = d->getFirstFaceIterator();
TFace<T>::iterator it = f->getFirstLoopIterator();
for ( ; it!=f->end(); it ++ )
```

11. VISUALIZAÇÃO

O USPDesigner utiliza bibliotecas de visualização OpenGL.

O modelador USPDesigner possui quatro vistas do sólido (lateral, superior, frontal e perspectiva) e uma tela de informações sobre os comandos utilizados. O usuário pode utilizar os comandos que estão na tela do modelador, ou podem realizá-los na janela de comandos. Selecionando o sólido, face, aresta ou vértice o usuário visualiza suas propriedades computacionais (posição, ID, composição).

As rotinas de visualização se relacionam diretamente com a estrutura de dados e com rotinas de alto nível. Após o encapsulamento da estrutura e modificações nos operadores de alto nível fez-se necessária uma modificação na implementação das rotinas com vista em todo o iterfaceamento entre os módulos.

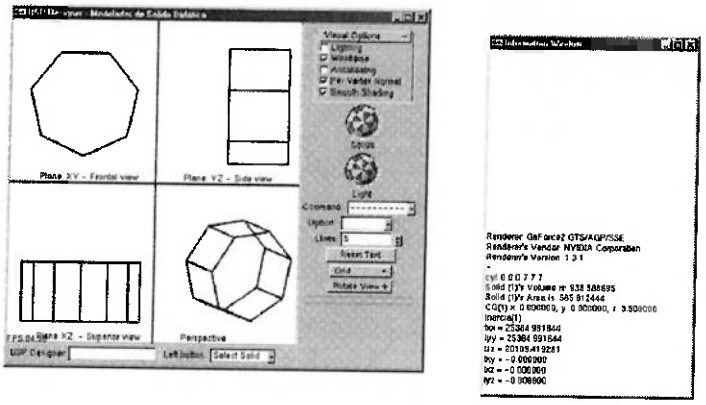


FIG. 19 TELA COM AS QUATRO VISTAS POSSÍVEIS E TELA DE INFORMAÇÕES.

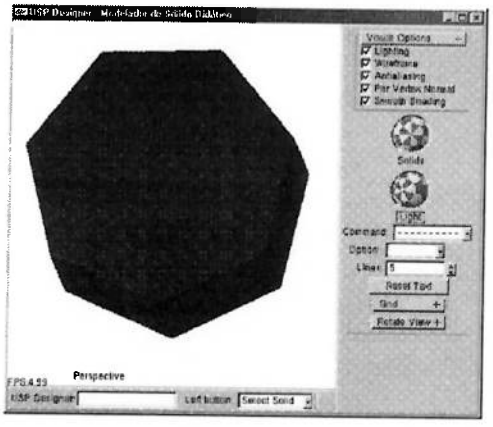


FIG.20 UMA DAS VISTAS INDIVIDUALMENTE.

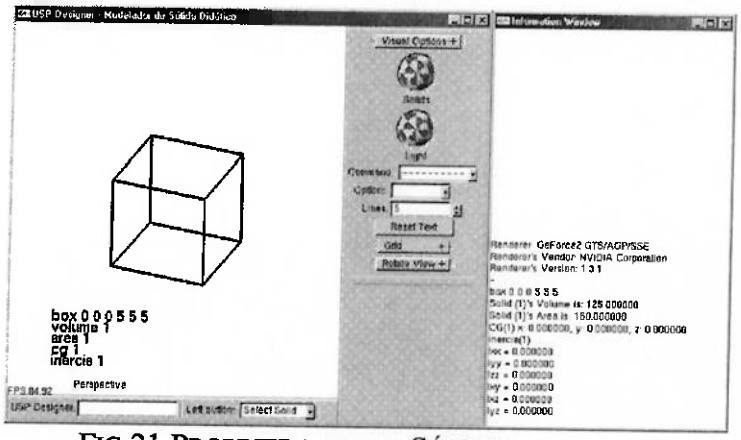


FIG.21 PROPRIEDADES DO SÓLIDO.

12. CONCLUSÕES

Neste trabalho foi apresentado uma nova implementação para a estrutura de dados do modelador de sólidos B-Rep didático USPDesigner. Para este fim apresentou-se um estudo da estrutura original do modelador em todos os níveis e o conceito de implementação do encapsulamento de estruturas.

Foi proposta uma nova estrutura de dados, constituida pelo conceito de encapsulamento, que permiti ao usuário utilizar as interfaces dos módulos sem visualizar a implementação destes. Bem como utilizar as propriedades dos elementos sem precisar conhecer toda a hierarquia da estrutura.

Isto facilitou o entendimento da estrutura e o acréscimo de módulos ou operações, tendo em vista sua implementação. Diminuiu-se linhas de código e tornou mais simples a acessibilidade e visualização pelo ponto de vista de programação.

Esta nova estrutura tornou a utlização de memória mais eficiente, tendo em vista que modificou-se a estrutura das listas utilizadas, não se construindo mais cópias e sim guardando ponteiros mais consistentes dos elementos.

A partir desta nova estrutura encapsulada e mais robusta, modificou-se todos os demais níveis de abstração, sempre auxiliado pela simplificação do interfaceamento entre eles e pela facilidade de visualização da estrutura, chegando até as rotinas de visualização. Estas comprovaram que as melhorias de tempo e utilização de memória impostas pela nova estrutura foram consolidadas e não afetaram a transcrição gráfica dos elementos criados pelo modelador.

Portanto, ganhou-se um modelador com estrutura mais simples, com maior aceitabilidade para mudanças e novas implementações, com ganho em tempo de compilação, memória utilizada e linhas de código.

Apêndice A - ALGORITMO DE OPERAÇÃO BOOLEANA

A operação booleana pode ser dividida em diversas etapas, que estão representadas na Figura 21, onde estão as principais funções de auxílio ao algoritmo de operação booleana:

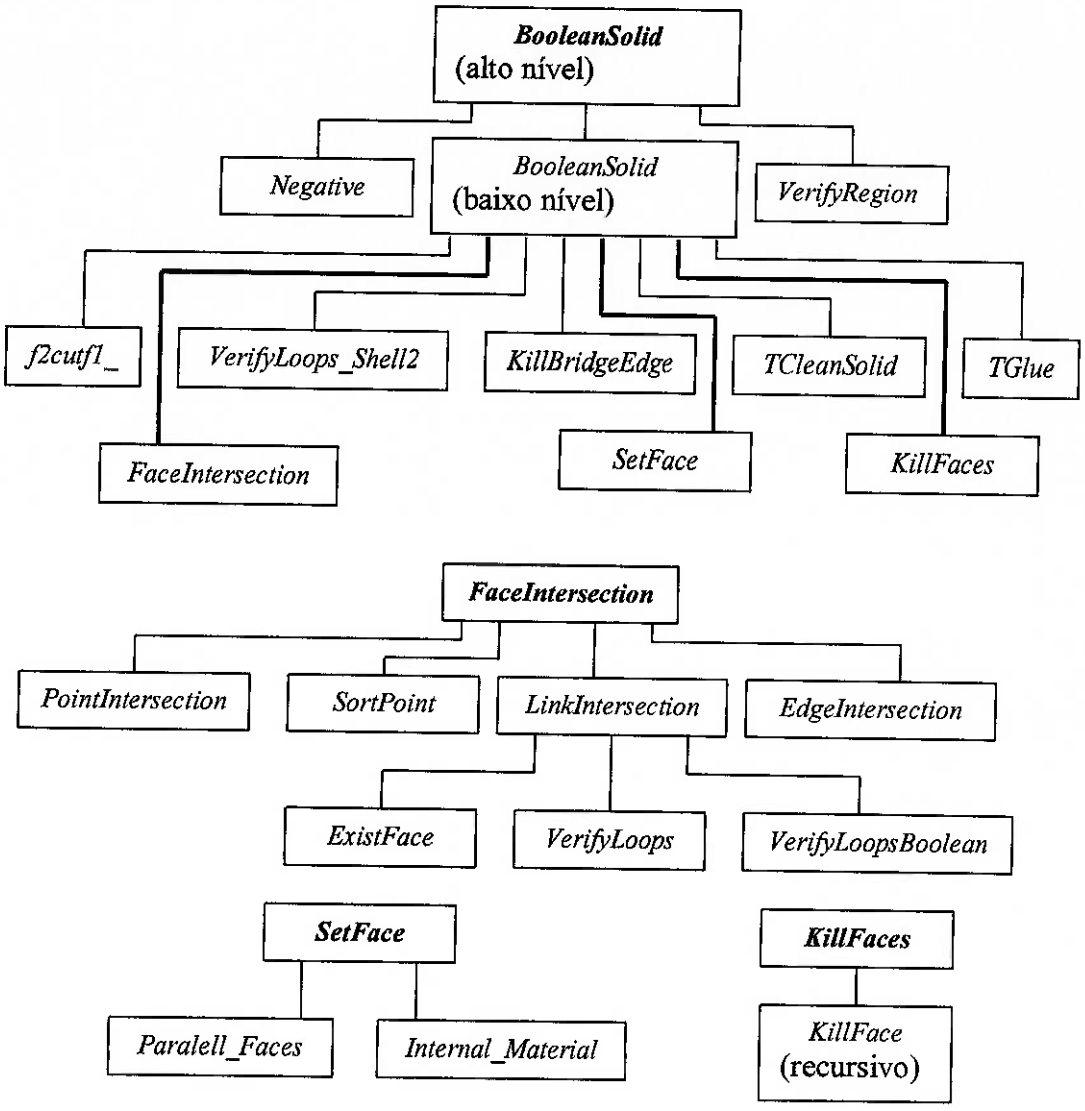


FIG. 22. PRINCIPAIS FUNÇÕES DO ALGORITMO DE OPERAÇÃO BOOLEANA.

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 22).

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.

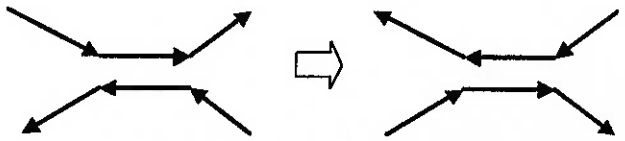


FIG 23. ILUSTRAÇÃO DE MUDANÇA DE SENTIDO DAS MEIA-ARESTAS.

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 23).

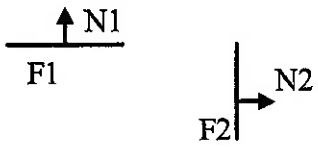


FIG. 24. APESAR DAS FACES F1 E F2 POSSUIREM NORMAIS DIFERENTES, ELAS ESTÃO MUITO AFASTADAS UMA DA OUTRA, O QUE IMPEDE A EXISTÊNCIA DE UM PONTO DE INTERSECÇÃO.

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 24 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.

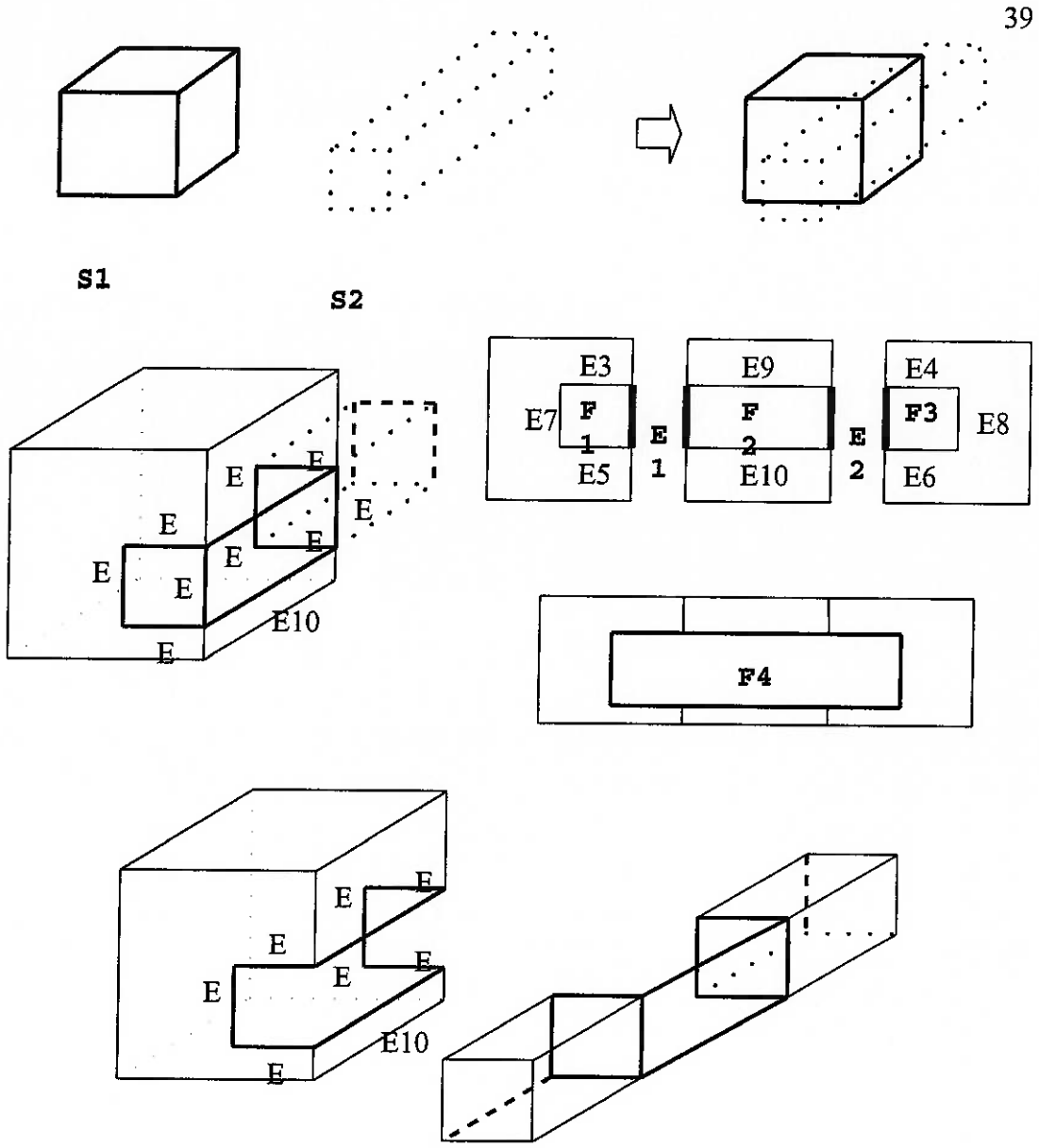


FIG. 25. EXEMPLO DE VERIFICAÇÃO DE ARESTAS QUE CONTORNAM FACES QUE DEVEM SER REMOVIDAS.

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 25 a,b e c).

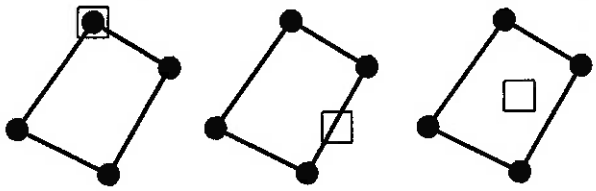


FIG. 26. (A) VÉRTICE COM COORDENADAS DO PONTO; (B) PONTO NA ARESTA (C) PONTO NO INTERIOR DO LAÇO;

Se o ponto de intersecção estiver sobre a aresta (Figura 25b), então, a função EdgeIntersection irá criar um novo vértice na aresta em questão (Figura 26).

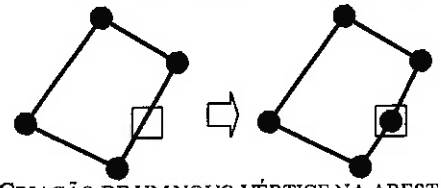


FIG. 27. CRIAÇÃO DE UM NOVO VÉRTICE NA ARESTA.

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 9.6). 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 27). Retorna o ponteiro para a nova aresta.

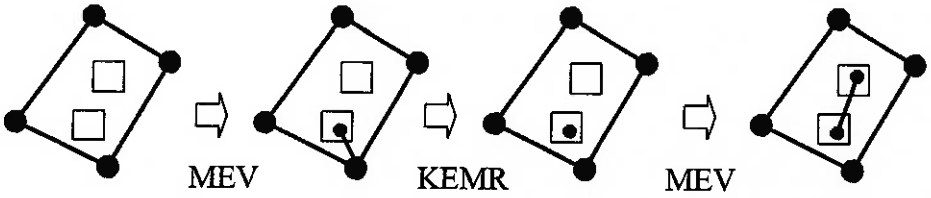


FIG. 28. ROTINA PARA QUANDO AMBOS OS PONTOS ESTIVEREM NO INTERIOR DA FACE.

- 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 28). Retorna o ponteiro para a nova aresta.

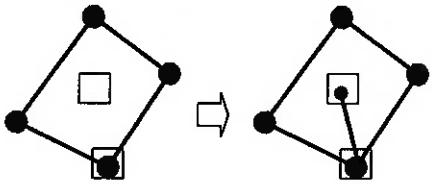


FIG. 29. CRIANDO UM NOVO VÉRTICE E UMA NOVA ARESTA QUE UNE ESTE NOVO VÉRTICE COM UM VÉRTICE JÁ EXISTENTE.

- 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 29) é enviado como parâmetro de retorno desta função;

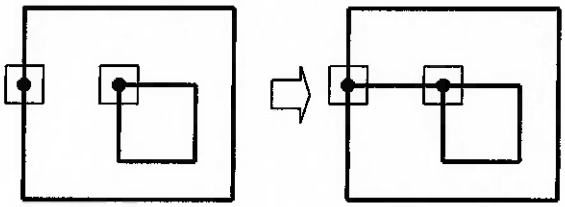


FIG. 30. CASO COMO LAÇOS DIFERENTES.

  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 30). 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;

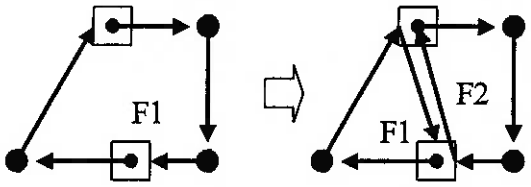


FIG. 31. NOVA FACE CRIADA, COM MESMA ORIENTAÇÃO QUE A ANTIGA.

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 31 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.

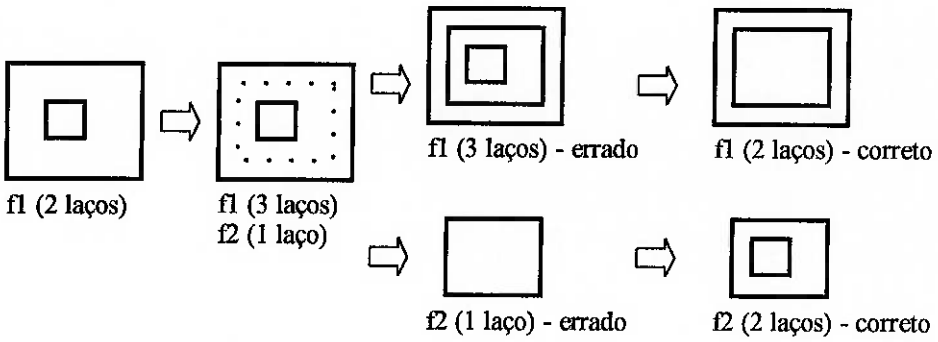


FIG. 32. CORREÇÃO PARA LAÇOS INTERNOS NO INTERIOR DE LAÇOS INTERNOS.

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.

l) 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 32, 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.

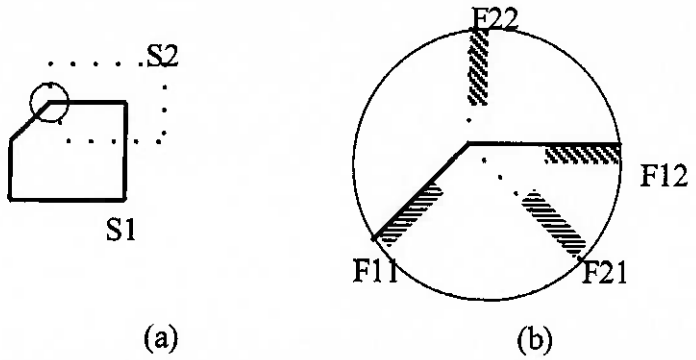


FIG. 33 A) SÓLIDO S1 E S2 B) DIEDROS DOS SÓLIDOS S1 E S2;

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 33 a,b,c e d)

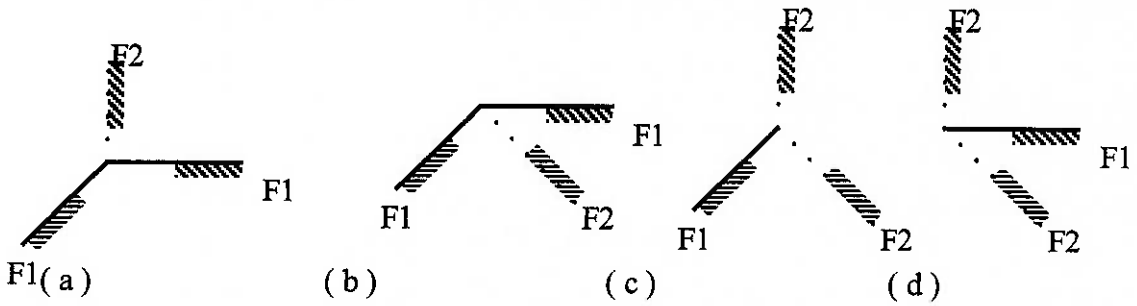


FIG. 34. A) CLASSIFICA F22 BASEADO NO DIEDRO F11/F12; B) CLASSIFICA F21 BASEADO NO DIEDRO F11/F12; C) CLASSIFICA F11 BASEADO NO DIEDRO F21/F22; D) CLASSIFICA F12 BASEADO NO DIEDRO F21/F22.

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 34b) ou opostas (Figura 34 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.

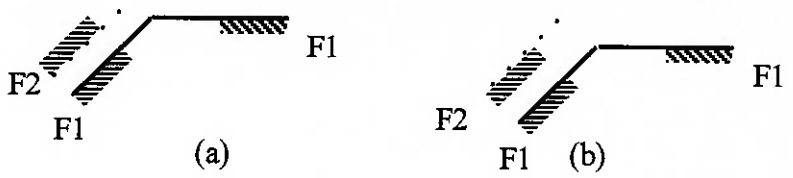


FIG. 35. A) F21 E F11 SÃO FACES COPLANARES COM NORMAIS OPOSTAS – AMBAS DEVEM SER REMOVIDAS; B) F21 E F11 SÃO FACES COPLANARES COM NORMAIS DE MESMO SENTIDO – UMA DELAS DEVE SER REMOVIDA.

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 35, a face F22 está sendo classificada com base no diedro F11/F9.

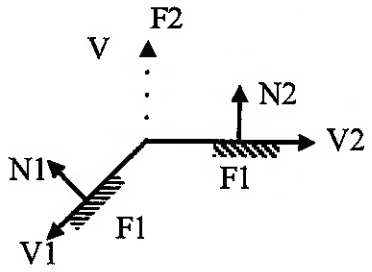


FIG. 36. CLASSIFICAÇÃO DA FACE F22 USANDO O DIEDRO F11/F12.

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 36.

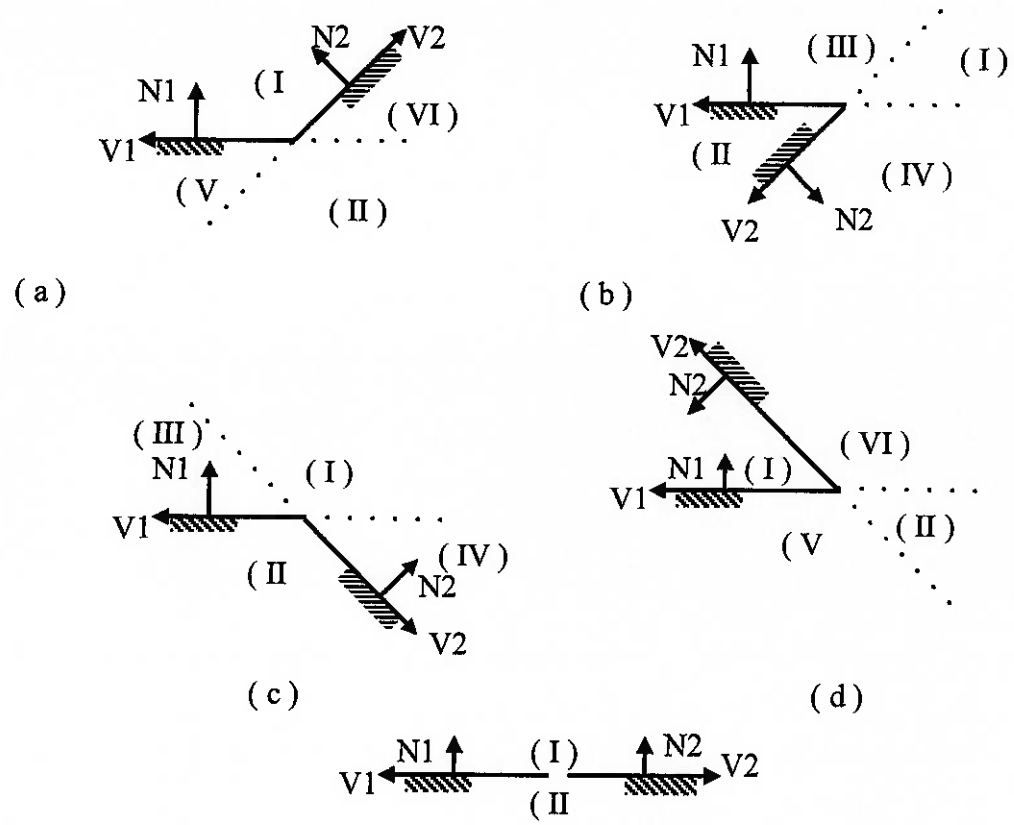


FIG. 37. SITUAÇÕES POSSÍVEIS DE CLASSIFICAÇÃO DE FACE BASEADO NO DIEDRO DO SÓLIDO OPOSTO.

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)
$1 \cdot V < 0$	$N1 \cdot V < 0$	$N1 \cdot V \geq 0$
$N2 \cdot V < 0$	$N2 \cdot V \geq 0$	$N2 \cdot V < 0$
	$V2 \cdot N1 \geq 0$	$V2 \cdot N1 \geq 0$
	$V1 \cdot N2 \geq 0$	$V1 \cdot N2 \geq 0$

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

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

(II)	(V)	(VI)
$1 \cdot V \geq 0$	$N1 \cdot V \geq 0$	$N1 \cdot V < 0$
$N2 \cdot V \geq 0$	$N2 \cdot V < 0$	$N2 \cdot V \geq 0$
	$V2 \cdot N1 < 0$	$V2 \cdot N1 < 0$
	$V1 \cdot N2 < 0$	$V1 \cdot N2 < 0$

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 37 a), então, a face do primeiro sólido de ser mantida. Do contrário, as duas faces são opostas (Figuras 37 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. 38. 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.

Apêndice B - OpenGL

O que é:

OpenGL é uma biblioteca gráfica de baixo nível desenvolvida primariamente para as linguagens C/C++ que disponibiliza ao programador uma pequena lista de primitivos geométricos como pontos, linhas, polígonos e até imagens em bitmap. Além disso, conta com comandos que permitem que o programador "arranje" esses primitivos em duas ou três dimensões, juntamente com comandos que controlam como esses objetos são renderizados no *frame buffer*, permitindo a formação de figuras complexas texturizadas.

Necessidade:

Antigamente, quando somente supercomputadores eram capazes de processar imagens tridimensionais, não havia linguagem universal para tal. Os programas eram feitos especificamente para cada máquina e o máximo que se obtinha de padronização provinha da experiência acumulada de alguns programadores (isso quando os programas não eram feitos "do zero"). Com isso, cada programa tinha sua própria lógica de exibir e trabalhar suas imagens.

Com o desenvolvimento de microprocessadores mais capazes e conseqüente surgimento dos aceleradores gráficos, a computação gráfica tridimensional passou a ser uma realidade também para os computadores pessoais. Isso gerou um enorme aumento na "circulação" de imagens em 3-D. Estas passaram a ser vistas desde em jogos de vídeo-game até em complexos modeladores de acabamento em tempo real.

A partir daí, criou-se a necessidade de se desenvolver padrões de linguagem gráfica que permitisse que programas fossem escritos independentemente do *hardware* ou do sistema operacional em que fossem executados. O OpenGL foi um dos primeiros padrões criados, e é uma dos mais utilizados até hoje.

Vantagens:

A grande vantagem do OpenGL, dada a necessidade para a qual é solução, é que ele pode ser traduzido para qualquer tipo de plataforma. Ele libera o programador de ter que escrever códigos para um equipamento específico. Se o comando dado for executável pelo dispositivo de *hardware*, ele o executa. Senão a biblioteca o executa no processador central.

Outra grande vantagem: a possibilidade de exibir os resultados (imagens) não somente na máquina onde se executou o programa, mas também através de redes, dada sua funcionalidade de ser executado independentemente do sistema operacional, numa relação *servidor-cliente.*

Funções:

Existem cerca de 150 comandos através dos quais o programador define seus objetos e os renderiza. Esses comandos definem os objetos primitivos, suas localizações no espaço 3-D (e outros parâmetros como escala e rotação), propriedades como cor, textura e material e até posição da câmera.

São algumas das principais funcionalidades:

- **Primitivos Geométricos:** A partir de primitivos geométricos, é possível criar qualquer tipo de objeto. A biblioteca oferece pontos, linhas e polígonos. É capaz também de criar objetos utilizando bitmaps.
- **Uso de splines:** *B-splines* podem ser usadas para desenhar curvas entre pontos.
- **Transformações na disposição dos objetos:** permite transladar e rotacionar os objetos em qualquer lugar do espaço 3-D, além de mudar suas formas a até mesmo a posição da câmera.
- **Trabalho com cores:** permite que se trabalhe com cores em modo RGB, ou através de uma *pallete* gráfica.
- **Ocultação de linhas e remoção de faces:** pode não exibir linhas e faces "escondidas" atrás de outras.
- ***Buffer* duplo:** previne que as animações (como rotação) fiquem piscando.
- ***Texture Mapping* (Mapeamento de Textura):** para dar realismo aos objetos renderizados em 3-D. Uma esfera, por exemplo, pode ser exibida como uma "bola" texturizada, e não somente como uma esfera de uma cor só. Isso também ajuda nos efeitos de iluminação.
- ***Antialiasing***: para suavização de retas que às vezes parecem "dentadas" nas bordas.
- **Iluminação**: além dos efeitos com o mapeamento de textura, é possível definir fontes de luz, com posição, intensidade e etc.

- **Transparência nas texturas dos objetos.**
- **Exibição de lista de objetos.**

Bibliotecas auxiliares:

Apesar do OpenGL ser capaz de reproduzir praticamente qualquer tipo de cena tridimensional, existem algumas funcionalidades que requerem a utilização de bibliotecas adicionais. O OpenGL sozinho, por exemplo, não é capaz de interagir com mouse ou com teclado. São algumas bibliotecas auxiliares:

GLU: o uso desta biblioteca já se tornou padrão e ela já vêm incluída no pacote básico do OpenGL. Ela possui funções um pouco mais complexas feitas a partir das funções mais básicas do OpenGL. Por exemplo, pode-se construir um cilindro com apenas um só comando. Também inclui funções adicionais para o trabalho com *splines* e incrementos para operações com matrizes e diferentes modos de projeção.

GLUT: esta biblioteca auxiliar incrementa as funcionalidades para se trabalhar com janelas, teclado e mouse e, assim como o OpenGL, independente da plataforma. Isso é importante, pois assim como as funções gráficas, o *setup* dessas interfaces também é específico de cada sistema operacional e *hardware*. Logo, assim como o OpenGL proporciona uma padronização das funções gráficas, o GLUT proporciona padronização da interface com esses periféricos. Além disso, também adiciona funções mais complexas a partir das funções básicas do OpenGL, como a construção de cones e tetraedro com um só comando.

GLUI: esta biblioteca pode ser considerada uma auxiliar do GLUT (que por sua vez auxilia o OpenGL). Ela fornece controles como botões, *checkboxes* e *spinners* (as esferas usadas para rotação da câmera no USPDesigner).

Apêndice C – ALGORITMO DA VISUALIZAÇÃO

A visualização do USPDesigner é realizada através das ferramentas oferecidas pelo Open GL. As principais funções de auxílio do algoritmo são: *draw*() e *graphic*().

Na função *draw* temos todo o programa necessário para se desenhar os objetos.

```
#define draw_____

void MSD_glVertex3f(const unit &x, const unit &y, const unit &z) {};
void MSD_glNormal3f(const unit &x, const unit &y, const unit &z) {};
double value( const unit &x) {};
void draw_point(TVertex<T> *x) {};
```

Auxiliares para desenhar os vértices.

```
void draw(TVertex<T> *x) {};
```

Desenha os vértices dos sólidos.

```
void draw(TEdge<T> *x) {};
```

Desenha as arestas dos sólidos.

```
void draw_stencil(TFace<T> *x) {};
```

Desenha as faces de acordo com a opção do usuário, neste caso stencil.

```
void draw_hidden(TFace<T> *x) {};
```

Desenha as faces de acordo com a opção do usuário, neste caso hidden.

```
void CALLBACK vertexCallBack( GLdouble *vertex) {}
```

Auxiliar para desenhar os vertices.

```
void draw_tess_nonormal(TFace<T> *x) {};
```

Desenha as faces de acordo com a opção do usuário, neste caso normal.

```
void draw_outline_test(TShell<T> *d) {};
```

Auxiliar para a linha de for a.

```
void draw_outline2(TShell<T> *d) {};
```

Opção de linhas das arestas.

```
void draw_wireframe2(TShell<T> *x) {};
```

Opção de linhas no wireframe.

```
void draw_outline3(TShell<T> *d) {};
```

Opção de linhas das arestas.

```
void draw_outline(TShell<T> *d) {};
```

Desenha as linhas de fora e pode desenhar as faces dependendo da opção do usuário.

```
void draw(TFace<T> *x) {};
```

Percorre os loops dos sólidos a serem desenhados. Manda percorrer os loops de acordo com a opção selecionada pelo usuário.

```
void draw(TShell<T> *x) {};
```

Percorre as faces e as arestas dos sólidos a serem desenhados. Pode mandar percorrer as arestas na opção wireframe ou as faces na opção não-shading.

```
void draw_wireframe(TShell<T> *x) {};
```

Percorre e desenha as arestas dos sólidos na opção wireframe.

```
void draw_shading(TShell<T> *x) {};
```

Percorre as faces dos sólidos a serem desenhados na opção shading.

```
void draw(TRegion<T> *x) {};
```

Percorre as shells dos sólidos a serem desenhados.

```
void draw(TSolid<T> *x) {};
```

Percorre todas as regiões dos sólidos a serem desenhados.

```
void draw() {};
```

Percorre toda a lista de sólidos que será desenhada.

Na função *graphic* temos as opções de visualização par o usuário e a interface com a função *draw*. Nesta função temos as funções *doSolid*, *delSolid* e *loadSolid*.

Para a visualização temos na função *doSolid*:

```
void doSolid(void) {
      draw<double>();

      if (LIGHT);
      if (WIREFRAME);
      if (AXIS) glCallList(AXIS_LIST);
      if (PLANE_XY) glCallList(PLANE_XY_LIST);
      if (PLANE_YZ) glCallList(PLANE_YZ_LIST);
      if (PLANE_XZ) glCallList(PLANE_XZ_LIST);
}
```

Esta função aciona a função *draw* e fornece as referências requeridas para a visualização de acordo com a opção do usuário.

Temos as funções auxiliares de visualização como a *graphic_interface* que define as propriedades da tela de visualização do usuário e as opções permitidas

A função *grid* que define as linhas de referência que o usuário dispõe para visualização.

Temos o *header A1.h* que realiza a interface entre a visualização do programa, com o tipo de visualização dos objetos escolhida pelo usuário com todas

as operações e teclas de funções disponíveis. Além das opções de gráfico definidas na outras funções, temos acréscimo de outras funcionalidades como: escala, rotação, interface com mouse, linha de comandos, etc.

```
#define a1_h_______

void init(void)
{

      glShadeModel(GL_SMOOTH);
      glClearColor(1.0, 1.0, 1.0, 0.0); // sets background
      glEnable(GL_LIGHTING);
      glEnable(GL_NORMALIZE); // recalculates normals after scaling
}

void Redisplay(void) {}

void idle(void) {}

void write_message() {}

void write_select(int x, int y) {}

void write_subtitle(float x, float y, char *message) {}

void light_display(void)

void draw_selection_box(void) {}

void display(void)

void display_select(void) {}

void Scale (int x, int y) {}

void motion(int x, int y)

void rotation(int x, int y)

void mouse(int button, int state, int x, int y){}

void keys(unsigned char key, int x, int y){
      case 's':
      case 'S':
            sprintf(msg,"solid [ %d ] is selected ",
Target_Solid1());
            MSDprint(msg);
            break;

      case 'f':
      case 'F':

int command_maker(char *command, float id[10]) {
      if (!strcmp(command,"line")) {
      if (!strcmp(command,"box")) {
      if (!strcmp(command,"torus")) {
```

```
}

int command_reader(char *st) {}

void button_control(int button_id) {}


void checkbox_control(int id) {
    switch(id) {
    case LIGHTING_ID:
    case ANTIALIAS_ID:
      case SMOOTH_SHADING_ID:
}

void command_list_control(int id) {
}

void subwin_top(GLUI *glui_subwin) {
}

void subwin_bottom(GLUI *glui_subwin) {
}

void subwin_left(GLUI *glui_subwin) {
      glui_subwin->add_statictext("teste1");

void About_display(void) {

void VGA_display(void) {
}

void Generic_keys(unsigned char key, int x, int y) {
}

void AboutWindow(int x, int y, char *name, void function(void)) {
}

void ExtraWindow(void function(GLUI* subwin) ) {
}


void subwin_right(GLUI *glui_subwin) {
command_list_control);
      switch (command_list_viewer) {
      case 1: // primitives
      case 2: // modify solid
      case 3: // modify face
      case 8: // File
      case cylinder_id:
      case sphere_id:
      case torus_id:
      case boolean_id:

void special(int key,int x, int y){
      case GLUT_KEY_F1:
            sprintf(msg,"Full Window - Plane XY Mode", NULL);
      case GLUT_KEY_F2:
      case GLUT_KEY_F5:
```

```
            sprintf(msg,"Full Window - All Views", NULL);
#endif
```

BIBLIOGRAFIA

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

BERRY, J. T., **C+ Programming**, segunda edição, Prentice Hall, 1995.

BORLAND INTERNATIONAL, **Borland C++ Library Reference**, versão 4.0, Borland International, 1993.

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

MÄNTILÄ, M., **An Introduction to Solid Modeling,** Computer Science Press., 1998.

PAPPAS, C. H. & Murray, W. H., **Turbo C++ Completo e Total**, McGraw Hill, São Paulo, 1991.

PORTER, A., **C++ Programming for Windows**, McGraw Hill, 1993.

TSUZUKI, M. S. G., **Apostila de PMC490** – Projeto Auxiliado por Computador, EPUSP, 2002.

SHIMADA, M., **Aritmética Intervalar em um Modelador de Sólidos B-Rep**, Dissertação de Mestrado, EPUSP, 2002.