# DESENVOLVIMENTO DE UM MODELADOR DE SÓLIDOS B-REP UTILIZANDO PROGRAMAÇÃO GENÉRICA

**Marcos de S. G. Tsuzuki**
**Nelson Vogel**
Escola Politécnica da Universidade de São Paulo
Departamento de Engenharia Mecatrônica e de Sistemas Mecânicos
CEP 05508-900, São Paulo, SP, Brasil. E-mail: mtsuzuki@usp.br

**Resumo.** *Neste trabalho é apresentado o USPDesigner, um modelador de sólidos baseado na estrutura B-Rep (Boundary Representation) que utiliza diversos elementos da programação genérica e STL C++. O objetivo é tornar esse modelador uma ferramenta reutilizável, na qual possam ser criadas com facilidade novas features. Estaremos definindo um ambiente onde será possível solucionar uma série de problemas geométricos. O usuário (programador) terá que se preocupar apenas com a lógica relacionada a seu problema específico, aproveitando todas as outras funções existentes no USPDesigner. A principal contribuição deste trabalho está relacionada às features do modelador: trabalhos semelhantes, tais como o CGAL (*The CGAL Consortium, 1999) *e o GrAL (Berti, 2002; Kettner, 2003), oferecem uma reutilização bastante efetiva, porém em nenhum destes trabalhos encontramos uma estrutura de dados tão completa como a que está sendo implementada. Um exemplo interessante desta capacidade diferenciada do USPDesigner é o cálculo do volume apresentado no trabalho, que se torna extremamente simples e eficiente com o uso da estrutura B-Rep e da programação genérica.*

**Palavras-chave:** *Programação genérica, CAD, modelagem de sólidos*

## 1. INTRODUÇÃO

Funcionalidades geométricas são cruciais para uma grande variedade de aplicações, incluindo engenharia mecânica. Por funcionalidades geométricas podemos compreender como sendo processos que englobem Modelagem de Sólidos e Modelagem Geométrica.

Geralmente, processamentos geométricos estão embutidos no contexto de um problema maior. Devido à diversidade dos processos geométricos, diversas ferramentas devem ser combinadas para obter a solução desejada. A implementação de algoritmos geométricos é difícil e demanda um grande tempo, portanto a reutilização é altamente desejável. Infelizmente, implementações tradicionais estão intimamente amarradas a decisões que não permitem a sua utilização em contextos distintos.

Propostas convencionais de implementação de ferramentas geométricas costumam apresentar problemas de eficiência, manutenção, escalabilidade e qualidade, pois estão limitadas a copiar dados via uma API (ou para um arquivo), e acionando uma rotina externa (ou aplicação) implementando a funcionalidade desejada.

A biblioteca CGAL é uma proposta que tornou a reutilização efetiva, mas no CGAL possuímos apenas um conjunto muito básico de elementos de Modelagem de Sólidos. O Modelador de Sólidos USPDesigner (que é o objeto de nosso estudo) utiliza muito do STL C++ e da proposta de padrões de projeto (Gamma et. al., 1994).

Por exemplo, o USPDesigner poderá utilizar aritmética de ponto flutuante ou aritmética intervalar. Caso se deseje uma nova aritmética, será necessário definir um conjunto de classes que possam ser adaptadas ao framework. A fim de garantir o bom desempenho do sistema, estamos incluindo a possibilidade de definir a estratégia de gerenciamento de memória: próprio windows, gerenciamento próprio sem memória virtual e gerenciamento próprio com memória virtual. Também utilizaremos os padrões de projeto Factory, SmartPointer e Iterator que garantem o uso correto e eficiente da memória.

Um outro fator importante neste projeto será a definição de um padrão para que novas funções sejam acrescentadas ao USPDesigner e para isto estamos utilizando os padrões de projeto Command e Mediator. Algumas funções podem requerer uma certa interação com o usuário para que ele defina os parâmetros para o seu acionamento, esta interação também está sendo encapsulada nos padrões criados, de modo que o desenvolvedor fique ocupado apenas nas atividades mais importantes para si.

Caso sejam criados complementos a estruturas de dados, eles devem estar associados a Functors para que não ocorra nenhuma modificação no módulo de UNDO e REDO. O USPDesigner está sendo implementado para o ambiente Windows e possui saída gráfica pelo OpenGl. toda a interação gráfica ocorre através de APIs do OpenGl, tanto para visualização de saída como seleção de elementos (vértices, arestas, faces ou sólidos completos). Esperamos através deste projeto facilitar a sua reutilização, sua manutenção e a criação de novas funções.

Nesse artigo inicialmente será detalhada a estrutura B-Rep, base teórica para a estrutura de dados do modelador. Depois será fornecida uma visão geral do modelador, focando nos requisitos de projeto. Em seguida serão explorados alguns dos recursos que o modelador oferece, iniciando pela interação homem máquina que foi criada por meio de comandos e menus (classes Command e Mediator), o uso de Functors para as funções de UNDO e REDO, e o mapeamento entre os objetos do USPDesigner com a interface OpenGL. Também será explicado o uso do SmartPointer que foi adotado para garantir o uso correto da memória e ponteiros. No final do trabalho é relatado um exemplo prático: o cálculo de volume de um sólido, mostrando como o USPDesigner consegue implementar de maneira bastante simples funcionalidades que são aparentemente complexas.

## 2. ESTRUTURA B-REP

A representação B-Rep armazena detalhes de como as faces, arestas e vértices de um sólido se unem para representar um sólido (Mäntylä, 1988; Tzusuki, 2002). Um sólido representado em modelo B-Rep deve possuir, por exemplo, a capacidade de descrever como cada face está conectada as suas faces adjacentes, de modo a formar um volume totalmente fechado no espaço.

Os três tipos de elementos básicos de um sólido (face, aresta e vértice) e a informação geométrica relacionada aos mesmos formam os constituintes básicos dos modelos B-Rep. Junto com as informações geométricas, como equações do plano das faces e coordenadas de vértice, um modelo B-Rep deve também representar a relação entre as faces, arestas e vértices. Normalmente as informações geométricas dos elementos de um sólido são denominadas por geometria do modelo B-Rep, enquanto as informações sobre o compartilhamento dos elementos básicos são denominadas informalmente por topologia do modelo. Pode-se dizer que a topologia funciona como uma goma onde as informações geométricas são aglutinadas; ou então que “as informações topológicas criam um vigamento no qual as informações geométricas são posicionadas”.

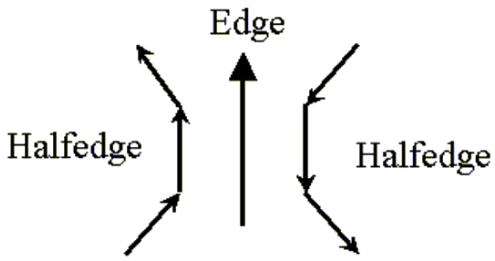


**Figura 1.** Representação através de half-edges.

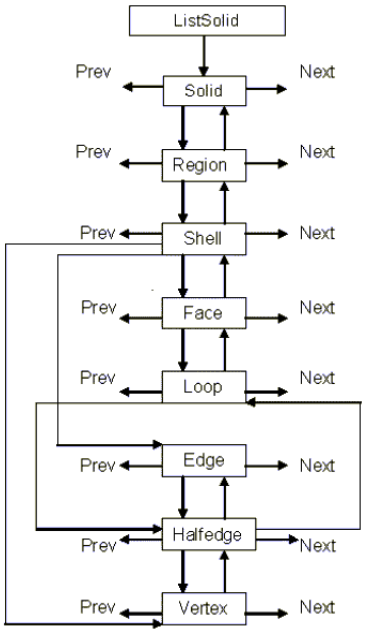


**Figura 2.** Vista hierárquica da estrutura half-edge.

## 2.1. Estrutura de Dados

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 half-edge (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 half-edge, 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 (1) ilustra a representação half-edge.

Em cada estrutura half-edge está sendo representada a metade das informações de adjacência da estrutura winged-edge. Cada half-edge possui apenas uma orientação, e cada face possui um circuito direcional de half-edges. Devido ao formalismo das Operações Booleanas, ao combinarmos dois sólidos por uma Operação Booleana, o resultado será sempre apenas um sólido. Entretanto, mesmo em situações especiais, onde o resultado da Operação Booleana aparenta apresentar dois sólidos, o resultado é considerado como sendo apenas um sólido. Cada uma das partes do sólido resultante é considerada um Shell.

Alterações na estrutura de dados half-edge de maneira a suportar faces com mais de um contorno não afetarão a estrutura B-Rep ao nível de aresta, mas sim, ao nível de face. Uma técnica muito comum é adicionar uma estrutura de tamanho fixo chamada laço (loop) que é associada a cada contorno da face. A estrutura laço simplesmente fornece à estrutura face um mecanismo para manter uma lista ligada de ponteiros para os seus múltiplos contornos. Cada face possui um laço externo e zero ou mais laços internos. A Figura (2) ilustra a hierarquia da estrutura *half-edge* resultante.

## 2.2. Operadores de Euler

Por conterem informações sobre as adjacências entre os elementos primitivos, a estrutura computacional anteriormente exposta é bastante complexa e a sua manipulação exige muitos cuidados para que a consistência dos dados seja mantida. Para contornar este problema, um conjunto de operadores foi desenvolvido com o objetivo de tornar a manipulação das estruturas de dados da representação B-Rep mais intuitiva. Eles permitem que a construção do sólido possa ser executada passo a passo, escondendo todos os detalhes de implementação da estrutura de dados. Estes operadores formam o segundo nível de representação do modelador. A equação de Euler-Poincaré diz que um sólido poliédrico é topologicamente válido se a relação mostrada na Eq. (1) entre as suas quantidades de elementos for verificada:

$$v - e + 2f = 2(s - h) + l \qquad (1)$$

onde $v$ é o número de vértices do sólido, $e$ o número de arestas, $f$ o número de faces, $s$ o número de shells, $h$ o número de furos e $l$ o número de laços. A forma mais conhecida na literatura da equação de Euler-Poincaré supõe ainda a existência de $r$ anéis no sólido, onde $r = l - f$. Resulta na Eq. (2):

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

Vários autores demonstram que seis operadores são suficientes para construir todos os objetos. Enquanto estes seis operadores podem ser escolhidos de várias maneiras, considerações de modularidade e independência criaram apenas pequenas variações na coleção encontrada na literatura. A seguir foram selecionados e estão descritos os seis operadores mais comumente utilizados na literatura.

- **`MVSF`** (*Make Vertex Solid Face*): este operador cria um sólido inicial com apenas uma face e um vértice;
- **`MEV`** (*Make Edge Vertex*): este operador adiciona a um sólido uma aresta e um vértice. A aresta é criada conectando-se um vértice já existente ao novo vértice criado;
- **`MEF`** (*Make Edge Face*): este operador adiciona ao sólido uma aresta e uma face. A face é criada pela divisão de uma face já existente acrescentando-se a nova aresta;
- **`KEMR`** (*Kill Edge Make Ring*): este operador divide o contorno de uma face em dois laços pela remoção de uma aresta-ponte;
- **`KFMRH`** (*Kill Face Make Ring Hole)*: nenhum dos operadores discutidos anteriormente é capaz de modificar as propriedades topológicas globais da estrutura de dados, como dividir um sólido em dois componentes ou criar um furo passante. O operador **`KFMRH`** possui este objetivo;
- **`MSFKR`** (*Make Shell Face Kill Ring*): este é outro operador que manipula informações globalmente. Ele transforma o anel de uma face em uma nova face, e todo o conjunto de faces associadas à nova face constituirá um novo *shell*;

# 3. VISÃO GERAL

No tópico anterior descrevemos a estrutura B-Rep e os operadores de Euler. Esses dois elementos são a base para o nosso modelador, do qual podemos ter uma visão geral na Figura (3).

## 3.1. Requisitos do sistema

Os seguintes requisitos estão sendo considerados durante o projeto:

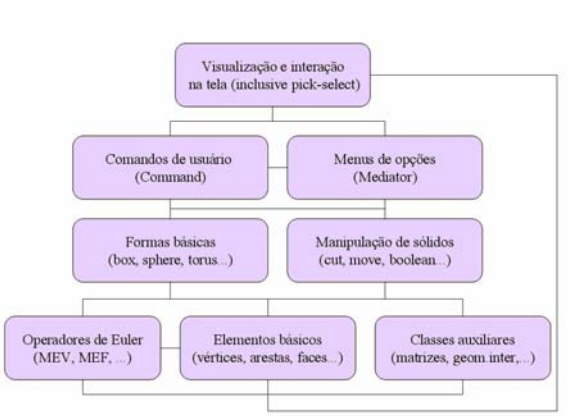


**Figura 3.** Visão geral da arquitetura do USPDesigner.

- **Flexibilidade**: Iniciamos a definição da estrutura de dados do modelador com a utilização da representação B-Rep e o uso das half-edges.
- **Performance**: Diversos elementos foram definidos como objetos Singletons, que garantidamente podem ser instanciados apenas uma vez. O uso intenso de ponteiros é outro fator que contribui para o melhor desempenho do sistema, porém traz também o risco de se utilizar de forma incorreta a memória do sistema. Os SmartPointers (Alexandrescu, 2001) torna o uso de ponteiros mais simples e seguro, associando uma semântica a sua utilização.
- **Reutilização**: O grando sucesso do STL é a principal prova da contribuição que a programação genérica trouxe para o C++. O uso de listas de objetos, de forma independente do conteúdo do objeto, e de Iterator tornam o sistema fácil de aprender, dar manutenção e evoluir com novas features.

### 3.2. Elementos básicos

Cada elemento básico é definido por uma classe. Nesta classe temos uma lista de elementos mais simples. Abaixo, temos a definição da classe TSolid que possui uma lista de elementos regiões. É possível observar que todo elemento possui um identificador para permitir o seu acesso. Nesta classe existem métodos para adicionar, recuperar e remover regiões.

```
template <class T>
  class TSolid : THierarchicalElement<T> {
    public:

      class iterator : public CSmartIterator<TRegion<T> > {
        public:
          iterator() {};
          iterator(const list<PtrRegion>::iterator it) :
            CSmartIterator<TRegion<T> >(it) {};
          ~iterator() {};
      };
      iterator begin(void);
      iterator end(void);
      void addRegion(PtrRegion r);
      int delRegion(PtrRegion r);
      int delRegion(const ID rn);
      PtrRegion getRegion(const ID rn) const;

    Private:
      ID solidno;                 /* solid identifier           */
      list<PtrRegion> sregions;   /* pointer to list of region  */
};
```

### 3.3. Operadores de Euler

Os operadores de Euler são utilizados de maneira bastante intensiva. Para se criar um cubo é necessário acionar inicialmente cinco Operadores de Euler e depois fazer o processo de extrusão, que por si só requer mais dez Operadores. Entretanto, não necessitamos mais que uma instância de cada Operador de Euler. Tendo em vista essa característica, para garantir o bom desempenho do sistema, os operadores de Euler foram implementados na forma de Singleton.

Cada operador de Euler foi implementado em dois níveis: alto e baixo. No nível mais alto, é necessário informar os identificadores dos elementos a serem manipulados desvinculando o seu acesso dos ponteiros. No nível mais baixo é necessário informar os ponteiros para os elementos a serem manipulados. Ambos os níveis não necessários, o primeiro permite programar a nível de interpretação de código, e o segundo nível melhora o desempenho de velocidade por acessar diretamente os ponteiros. A listagem abaixo mostra como exemplo a codificação da classe que implementa o Operador MEV (Make Edge Vertex).

```
template <class T>
  class TMEV: public TEulerOperator<T> {
      typedef auto_ptr<TMEV> TMEVPtr;
      static TMEVPtr & get_instance() {
          static TMEVPtr Singleton(new TMEV);
          return Singleton;
      };
      friend class auto_ptr<TMEV>;
      TMEV(const TMEV &);
      TMEV & operator = (const TMEV &);

    protected:
      TMEV() { };
      ~TMEV() { };

    public:
      static TMEV & instance() { return *get_instance(); };
      static const TMEV * const_instance() { return instance(); };
      int high(const ID sn, const ID fn, const ID v1, const ID v2, const T &x, const T &y, const T &z);
      void low(PtrHalfEdge he1, PtrHalfEdge he2, const ID vn, const T &x, const T &y, const T &z);
};
```

### 3.4. Criando um sólido

Como criar um sólido a partir de Operadores de Euler? Os sólidos mais comuns e conhecidos já estão implementados no modelador, e outros podem ser facilmente desenvolvidos no USPDesigner. Como exemplo, na listagem abaixo é criado um cubo. Para criá-lo é necessário fornecer as dimensões do cubo, o seu posicionamento no espaço tridimensional e o seu identificador.

```
template <class T>
  ID TBox<T>::low(const T &a_, const T &b_, const T &c_,
                  const T &x_, const T &y_, const T &z_, const ID sn) {
    T a = a_ - x_/2; T b = b_ - y_/2; T c = c_ - z_/2;
    T x=x_, y=y_, z=z_;

    TMVSF<T>::instance().high(0, 0, 0, 0, a, b, c);
    TMEV<T>::instance().high(sn, 0, 0, 1, a + x, b, c);
    TMEV<T>::instance().high(sn, 0, 1, 2, a + x, b + y, c);
    TMEV<T>::instance().high(sn, 0, 2, 3, a, b + y, c);
    TMEF<T>::instance().high(sn, 0, 3, 0, 1);

    PtrSolid s = TListSolid<T>::instance().getSolid(sn);
    PtrFace fac = s->getFace(1);
    MSD_lowMakeSweep<T>(fac, 0, 0, z);

    return sn;
}
```

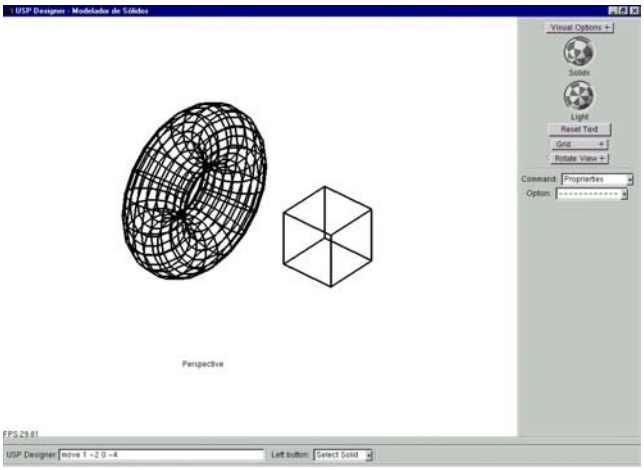


**Figura 4.** Interface do USPDesigner.

## 4. INTERAÇÃO HOMEM-MÁQUINA

O USPDesigner dispõe de um ambiente de visualização e interação com os objetos criados. É nesse ambiente que o usuário aciona os comandos e analisa o resultado das funções. O OpenGl está sendo utilizado por meio das bibliotecas GLUI e GLUT. O OpenGl está sendo utilizado para exibição e seleção dos objetos. Atualmente o software está sendo compilado no ambiente Windows utilizando o Visual C++ 6.0. A Figura (4) mostra a tela do USPDesigner. As funções do USPDesigner podem ser acionadas por duas maneiras: por linha de comando ou por menu. Será mostrado como disponibilizar uma função para o usuário. A classe Command possui esta responsabilidade de disponibilizar a função para ser executada como um comando, e a classe Mediator é responsável por associar o comando a uma posição do menu.

### 4.1. Command

A classe Command (Gamma et. al., 1994) é um padrão de projeto que possibilita criar novos comandos de maneira fácil e consistente. Geralmente, cada comando deve receber um determinado número de parâmetros de entrada. O número de parâmetros e seu tipo variam de acordo com cada comando. Por exemplo, uma linha pode ser criada fornecendo-se as coordenadas dos dois vértices, ou seja, seis números. Uma esfera pode ser criada passando como parâmetro o ponto central e o raio, e o número de divisões, portanto quatro números de ponto flutuante e um inteiro. Desta maneira, a classe Comand encapsula a transferência de parâmetros para acionamento do comando, tanto em quantidade como tipos. Na listagem abaixo podemos observar a definição da classe comando que criar uma esfera.

```
template <class T>
  class TCmdSphere : public TCommand<T> {
    protected:
        typedef TCmdSphere* TCmdSpherePtr;
        TCmdSphere();
        ~TCmdSphere() { };

    public:
        static TCmdSpherePtr pp;
        static TCmdSpherePtr & instance();

        virtual bool verify();
        virtual void run(char *command);
        virtual void run();
};
```

A função **`verify`** contém a lógica que permite a execução do comando, inclusive verificando se os parâmetros foram corretamente definidos. A função **`run(char* command)`** interpreta a linha de comando fornecida pelo usuário, e por último a funçao **`run(void)`** executa efetivamente o comando.

### 4.2. Mediator

Complementando a classe Command, o Mediator tem o papel de criar os menus e janelas de opções, para que o usuário utilize o USPDesigner. A hierarquia de menus deve ser intuitiva para que o usuário possa encontrar facilmente as funções que procura. É importante ressaltar que a hierarquia hoje utilizada é apenas sugestiva e pode ser alterada, até porque quando forem criadas novas funções, será possível incluí-las em menus existentes ou posicioná-la em novos itens de menu. Para incluir um comando na estrutura de menus, basta definir uma classe Mediator. No construtor dessa classe deve ser informado em qual menu e sub-menu o comando deve ser incluso. Na listagem abaixo é ilustrado o Mediator do comando "*sphere*". A função **`run(void)`** é que cria os elementos gráficos para que os parâmetros para a execução do comando sejam fornecidos. Esta função também associa os elementos gráficos criados para que o comando "*sphere*", ao ser executado, recupere os parâmetros corretamente.

```
template <class T>
  class TMedSphere : public TMediator<T> {
    protected:
        typedef TMedSphere* TMedSpherePtr;
        TMedSphere() : TMediator<T>("Primitives","Sphere") {};
        ~TMedSphere() {};

    public:
        static TMedSpherePtr pp;
        static TMedSpherePtr & instance();
        virtual void run(void);
};
```

### 4.3. Pick-Select

No USPDesigner, muitas operações requerem a seleção de um sólido, de uma face, de uma aresta ou de um vértice para que a operação possa ser executada. Por exemplo, para executar uma operação booleana é necessário que dois sólidos sejam fornecidos. Outro exemplo é que o comando para rotacionar um sólido pode ter o eixo de rotação definido pela aresta de um sólido. O recurso de pick-select permite selecionar um sólido, face, aresta ou vértice pelo uso do mouse, e utilizar esse elemento como parâmetro para executar uma operação.

O próprio OpenGl já possui associado a cada elemento um identificador próprio. Assim, foi necessário criar um mapeamento entre o sistema de identificação utilizado pelo OpenGl e o sistema de identificação adotado pelo USPDesigner. No OpenGl todo elemento possui um identificador único, independendo de seu tipo (sólido, face, aresta ou vértice). No USPDesigner cada elemento é definido por dois identificadores: o identificador do elemento e o identificador do sólido ao qual pertence o elemento. A classe pick-select conta com algumas funções que utilizam esse mapeamento para criar as listas do OpenGl, exibi-las na tela em modo visualização, exibi-las na tela em modo de seleção e até destruí-las. Essa classe também é responsável pelo tratamento dos cliques de mouse. Cada vez que o usuário aciona o botão do mouse em cima de um objeto, a função **`pick(int x, int y)`** tem como tarefa interpretar o clique do mouse e reagir a ele.

## 5. FUNCTORS - UNDO E REDO

O Functor é uma abstração que permite desacoplar a comunicação entre objetos. No USPDesigner, a implementação dos Operadores de Euler foi realizada utilizando-se Functors. Os comandos de usuário (classe Command) também foram implementados através de Functors. Com isso, é possível agrupar os Operadores de Euler em listas (listas de UNDO e REDO). Os parâmetros para executar os Operadores de Euler também estão embutidos nos objetos Functors. Assim, é possível executar o UNDO ou REDO sem a necessidade de sabermos quais são os Operadores de Euler que serão executados.

## 6. SMART-POINTERS

O SmartPointer (Alexandrescu, 2001) é um objeto em C++ que simula um ponteiro. Ele imita a sintaxe de uso do ponteiro, em particular os operadores `->` e `*`. A vantagem no uso do SmartPointer é que ele permite executar funções de gerenciamento de memória e travamento de forma transparente ao programador, utilizando a sintaxe a qual ele já está acostumado. O SmartPointer gerencia de certa forma o objeto que está sendo apontado por um ponteiro, e através disso consegue implementar funcionalidades adicionais.

Como o USPDesigner será utilizado por outros desenvolvedores que queiram criar suas próprias funções, é imprescindível proteger o ambiente contra falhas de programação. Um erro bastante comum é esquecer de deletar um objeto depois que o ponteiro passa a apontar para outro objeto. Outra falha bastante freqüente é a cópia de objetos sem necessidade, causando aumento no uso da memória.

Através da implementação do SmartPointer podemos também criar um gerenciamento próprio de memória. É possível agrupar uma determinada quantidade de elementos para serem colocados em um bloco de memória. Evita-se dessa forma a fragmentação de memória bastante comum em aplicações onde existe uma grande quantidade de pequenos elementos sendo criados, como é o caso do USPDesigner.

## 7. EXEMPLO PRÁTICO

A seguir listamos o algoritmo para calcular o volume de um sólido. Ele se baseia no teorema de Green, onde uma integral de volume pode ser transformada em uma integral de superfície se a superfície for coerentemente orientada (Mäntylä, 1988). No algoritmo abaixo, o sólido foi dividido em tetraedros orientados, tendo o seu volume associado a um sinal. Desta forma sólidos côncavos possuem alguns dos tetraedros com volume negativo, e vários outros positivos.

```
Template <class T>
T TVolume<T>::run(PtrLoop l)  {
    PtrHalfEdge he = l->getFirstHalfEdge();
    PtrVertex v1 = he->Vtx();
    tnVector<T,4> aa, bb, cc, vv1;
    T soma=0;
    He = he->Nxt();
    do {
        vv1 = v1->getCoord();
        aa.setDifference(he->Vtx()->getCoord(), vv1);
        bb.setDifference(he->Nxt()->Vtx()->getCoord(), vv1);
        cc.setCross(aa, bb);
        soma += vv1.dot(cc);
    } while ((he = he->Nxt()) != l->getFirstHalfEdge());
    return soma / 6;
}
```

## 8. CONCLUSÃO

Desta maneira foram ilustrados vários conceitos de programação genérica que estão sendo utilizados na implementação desta nova versão do USPDesigner.

## 9. REFERÊNCIAS BIBLIOGRÁFICAS


Alexandrescu, Andrei. Modern C++ Design: Generic Programming and Design Patterns Applied - Addison Wesley Professional, 2001

Berti, Guntram, Generic programming for mesh algorithms: Implementing universally usable geometric components, Fifth World Congress on Computational Mechanics, July 7-12, 2002, Vienna, Austria.

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.

Kettner, Lutz, Using generic programming for designing a data structure for polyhedral surfaces, In: Computational Geometry - Theory and Applications 13, pp. 65-90, Elsevier, 1999.

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

The CGAL Consortium, The CGAL home page, http://www.cgal.org, 1999.


## 10. DIREITOS AUTORAIS



# DEVELOPMENT OF A SOLID MODELLER B-REP USING GENERIC PROGRAMMING


**Marcos de S. G. Tsuzuki**
**Nelson Vogel**
Escola Politécnica da Universidade de São Paulo
Departamento de Engenharia Mecatrônica e de Sistemas Mecânicos
CEP 05508-900, São Paulo, SP, Brasil. E-mail: mtsuzuki@usp.br



**Abstract.** *We are presenting in this work the USPDesigner, a solid modeler based in the B-Rep structure (Boundary Representation) that uses many elements of the generic programming and STL C++. Our goal is to make this modeler a reusable tool, in which a programmer will be able to easially create new features. Therefore, we will be creating a framework in witch it will be possible to deal with most geometric problems. The user (programmer) will have that to be worried only with the logic related to his specific problem, using all the other existing functions in the USPDesigner. Our main contribution with this work is related to the features of the modeller: similar works, such as the CGAL (The CGAL Consortium, 1999) and the GrAL (Berti, 2002; Kettner, 2003), offers an effective reutilization, however none of them offers a data-structure as complete as the one that we are considering. An interesting example for this capacity of the USPDesigner is the volume calculation presented in chapter 8, which becomes extremely simple and efficient with the use of the B-Rep data-structure and the generic programming.*



**Keywords:** *Generic Programming, CAD, solid modeling*