UNIVERSIDADE DE SÃO PAULO
ESCOLA POLITÉCNICA
DEPTO. DE ENG. MECÂNICA

PMC-581

Projeto de uma interface de descrição de modelos sólidos B-rep por meio de operações locais e entrada de dados em 3D

RELATÓRIO FINAL

GLÁUCIO TERRA

ORIENTADOR:

PROF. DR. Marcos de Sales Guerra Tsuzuki

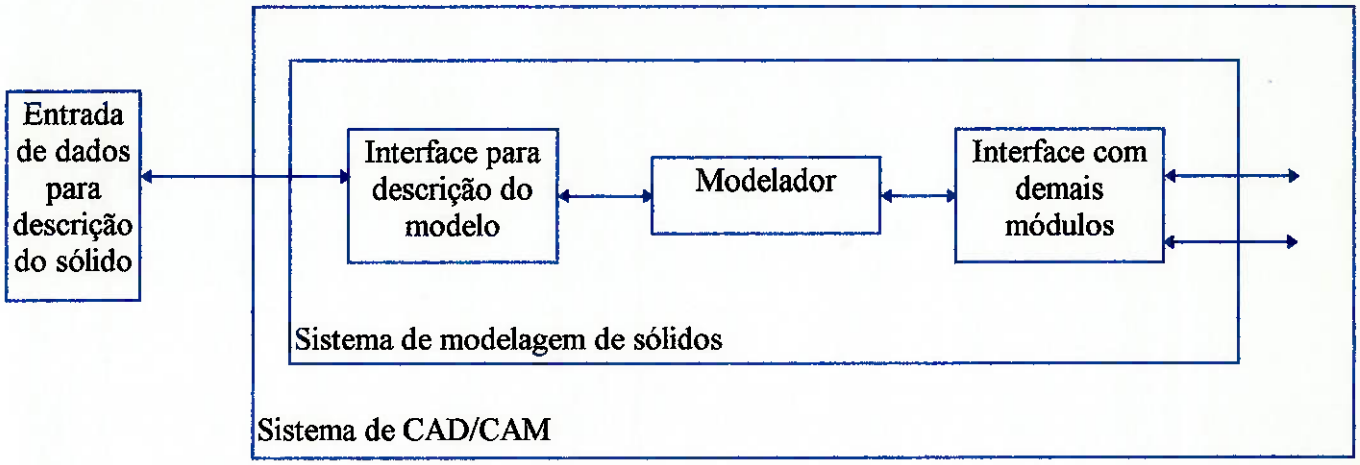
1996

1. Índice

2. Introdução e objetivos

Sistemas de CAD/CAM existentes atualmente possuem , em seu núcleo, um sistema de modelagem geométrica, o qual é capaz de modelar sólidos e responder algoritmicamente questões de natureza geométrica arbitrárias acerca dos sólidos por ele modelados.

Um tal sistema deve ter a seguinte estrutura:



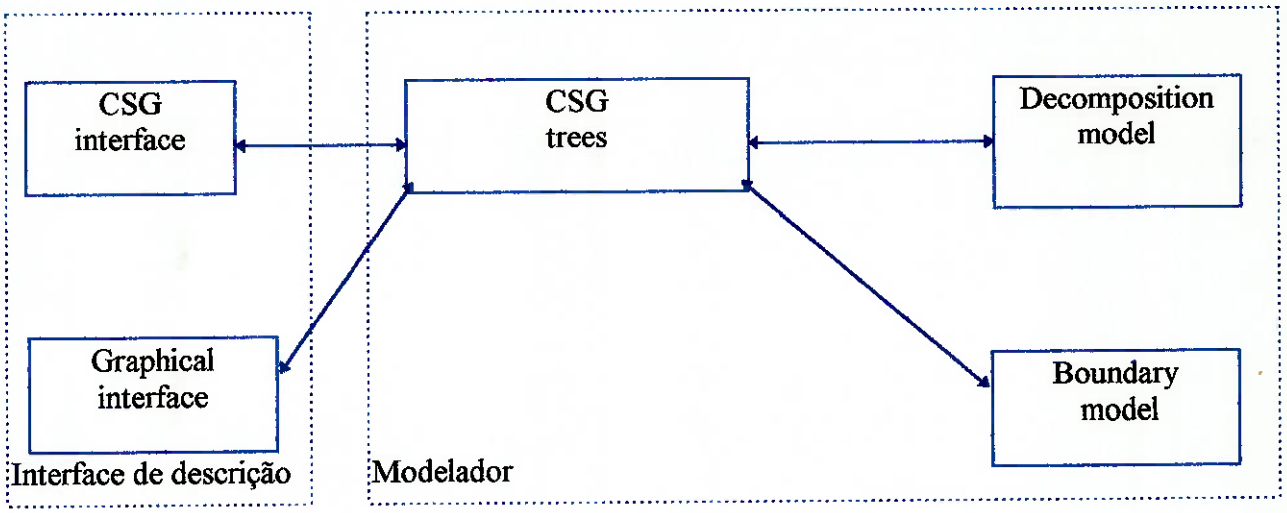
A entrada de dados para o modelador é feita por meio de uma interface de descrição, a qual recebe dados do usuário/memória externa por meio de uma linguagem de descrição ou interface gráfica interativa.

Uma vez entradas, as descrições dos objetos são traduzidas para os esquemas de representação internos do modelador, o qual deve ser capaz de trocar informações com outros módulos através de uma interface interna com os mesmos.

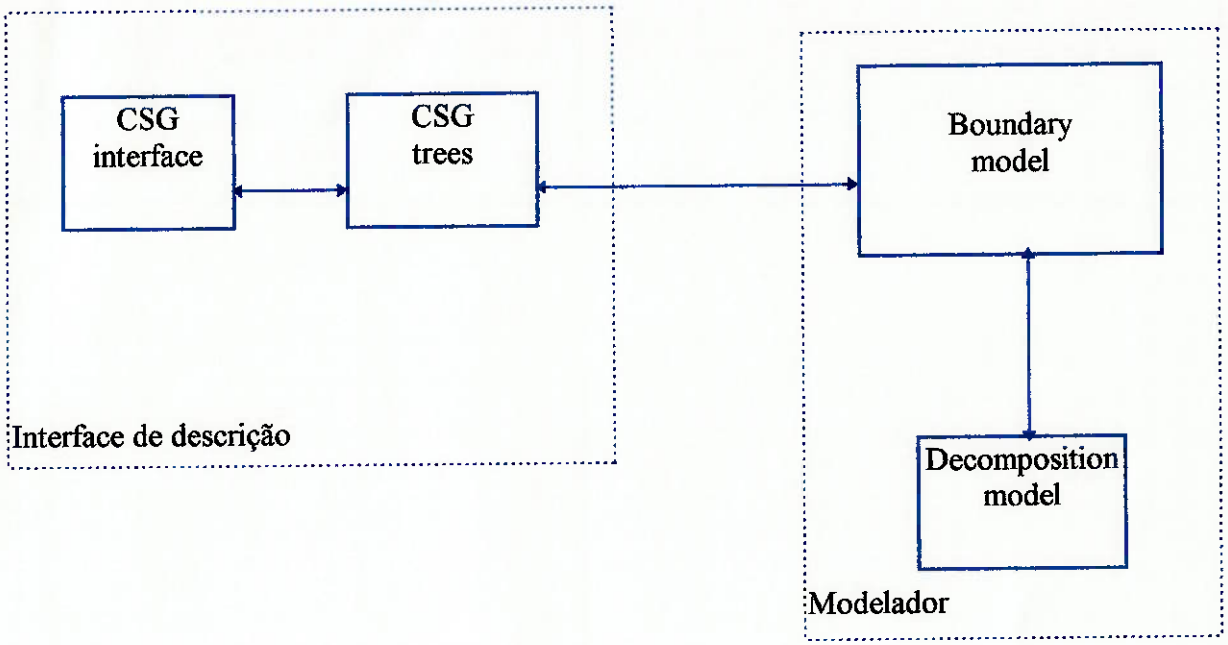
O modelador deve ser capaz de manipular e armazenar informações geométricas por meio de diferentes e coexistentes esquemas de representação, cada um deles adequado para um determinado

tipo de aplicação. Isto implica que o mesmo deve prover algoritmos de conversão entre os vários esquemas de representação por ele suportados. A grande dificuldade que surge é que não são conhecidos algoritmos de conversão entre todos os esquemas de representação; além disso, há problemas de consistência entre as representações, devido ao fato de esquemas de representação distintos possuírem, em geral, espaços de modelagem distintos.

A solução usualmente adotada em modeladores híbridos (isto é, modeladores que suportam diversos esquemas de representação coexistentes) para tais problemas de consistência e falta de algoritmos de conversão consiste em adotar-se uma das representações suportadas como primária, e as demais como secundárias. Seguindo-se esta idéia, a maioria dos sistemas de modelagem de sólidos atualmente existentes adotam uma das seguintes arquiteturas:



(a)



(b)

Modeladores do segundo tipo, que têm modelos B-rep como esquema de representação primário, utilizam uma interface de descrição CSG, o que significa que os sólidos são construídos por meio de operações booleanas sobre um dado conjunto de primitivos. Isto é adequado para sólidos de geometria simples, mas inconvenientes quando sólidos de geometria complexa estão envolvidos. Neste caso, seria interessante se o sólido pudesse ser construído por operações locais, o que simplificaria sobremaneira a descrição do mesmo.

Propõe-se, no presente trabalho, o projeto de uma interface de construção que permita descrever modelos sólidos B-rep a partir de operações locais.

3. Especificações da interface

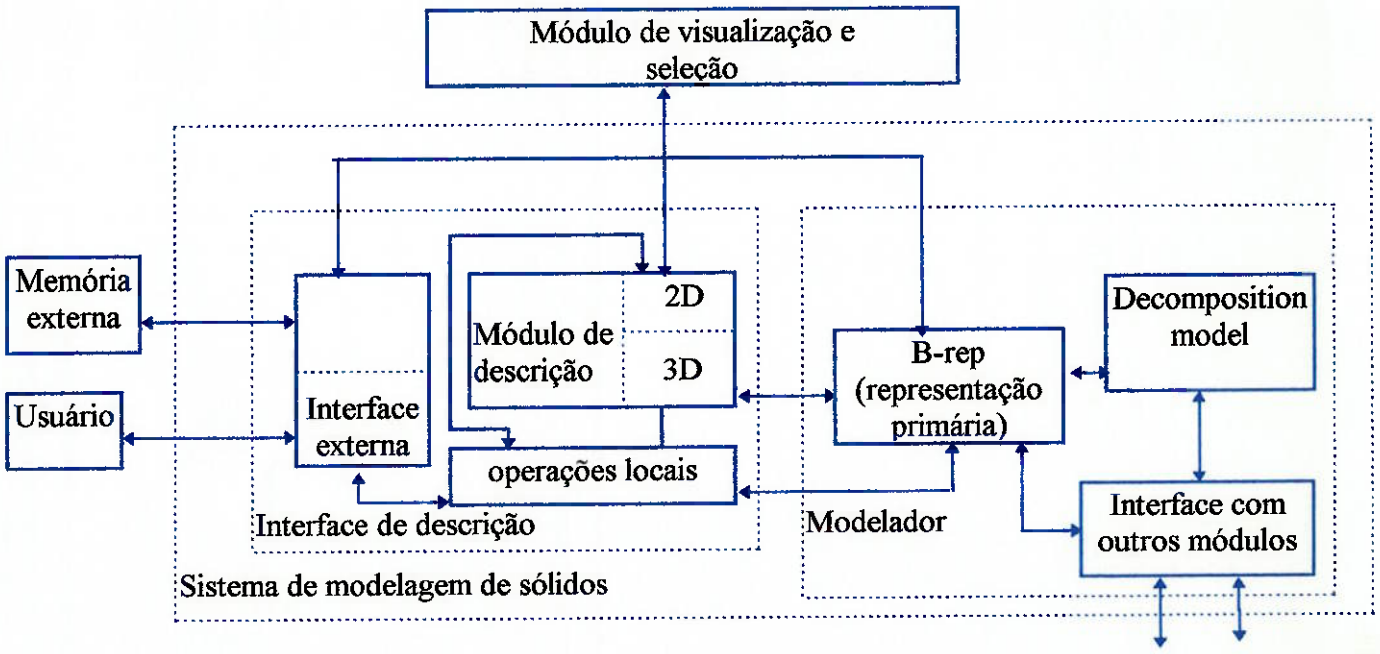
• A entrada de dados para a interface deverá ser feita através de memória externa ou de uma interface gráfica tridimensional, na qual o usuário entrará com dados interativamente por meio de dispositivo apontador ou digitalizador;

• A entrada de dados deverá ser ecoada no terminal de vídeo. Para tal, a interface deverá possuir um esquema de representação interno que permita interface com um módulo de visualização;

• A interface deverá prover módulos para comunicação com o modelador.

4. Solução proposta

Propõe-se desenvolver uma interface de descrição conforme esquematizado abaixo. Adicionalmente, desenvolver-se-á o módulo de visualização, sem o qual a interface não poderá ser testada.



5. Concepção da interface

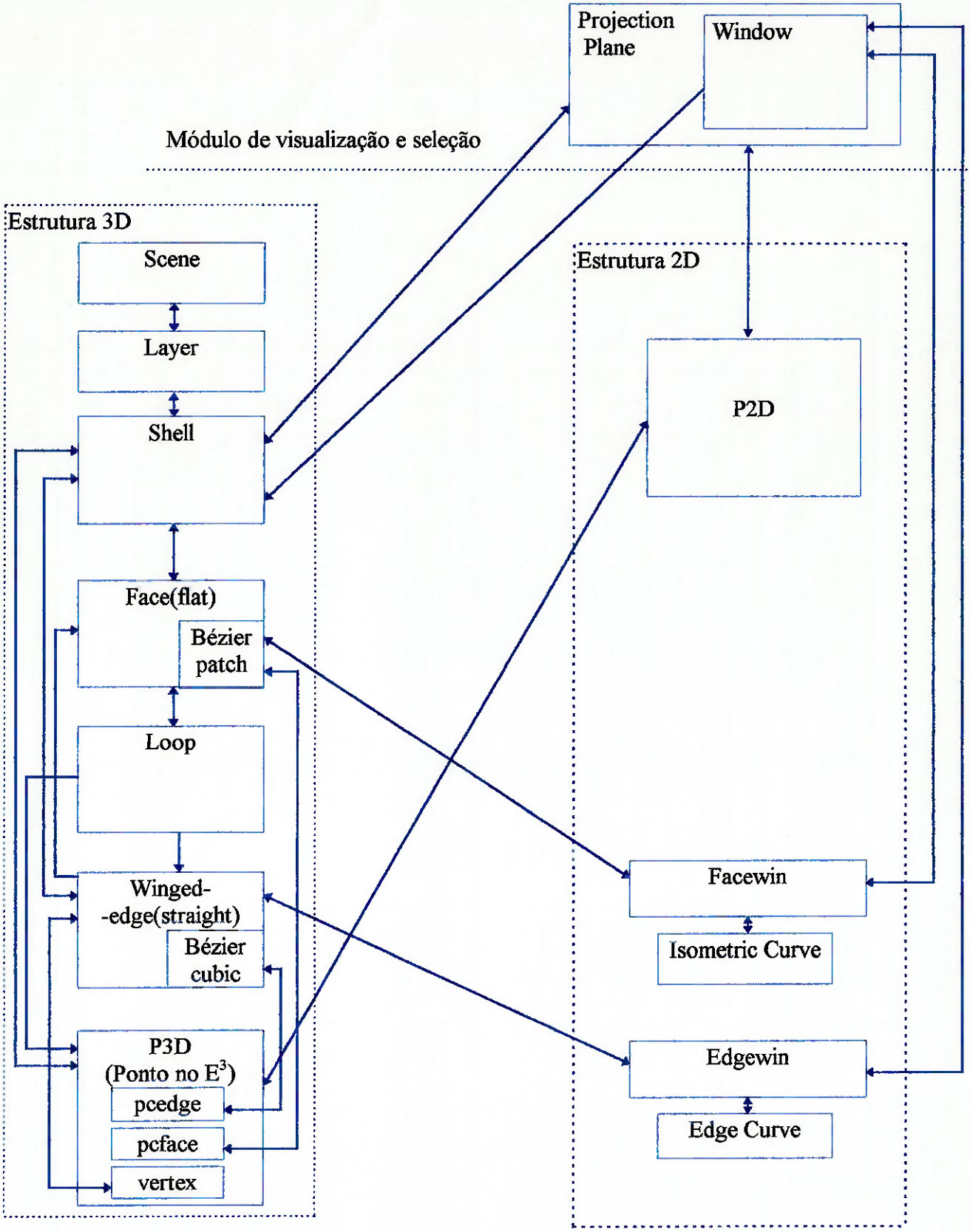
No que segue, será dada uma visão geral do projeto e implementação de cada um dos módulos da interface, conforme esquema do item anterior. Uma descrição em detalhes será feita posteriormente. Os módulos projetados foram desenvolvidos por meio de técnicas de programação orientada a objetos e implementados em C++ (usou-se o ambiente de desenvolvimento *Borland C++ 4.5*, da Borland, e a biblioteca *OWL 2.5*). Alguns algoritmos foram modelados e analisados por meio de *MFG*s.

5.1 Esquema de representação interno

No projeto do esquema de representação interno, optou-se por utilizar um modelo de representação limitada, ao invés do *wireframe* inicialmente proposto. Justifica-se tal escolha por permitir-se uma futura expansão das operações suportadas pela interface (renderização, por exemplo), a um "esforço" de projeto adicional relativamente baixo.

Desta forma, o esquema de representação interno foi implementado através de uma estrutura "*winged-edge*". Para fins de visualização, os sólidos modelados são projetados por uma perspectiva num determinado plano de projeção (note-se que o usuário tem total liberdade para escolher o plano de projeção, bem como os parâmetros da perspectiva) . Para tal, a estrutura tridimensional *winged-edge* é traduzida para uma lista de pontos bidimensionais, por meio de um objeto "*câmera*", que contém os métodos necessários para fazer-se a projeção.

O diagrama a seguir é uma sinopse da estrutura de dados utilizada para implementar o esquema de representação interno da interface.


Projection Plane
Window
Módulo de visualização e seleção
Estrutura 3D
Scene
Layer
Shell
Face(flat)
Bézier patch
Loop
Winged--edge(straight)
Bézier cubic
P3D
(Ponto no E3)
pcedge
pcface
vertex
Estrutura 2D
P2D
Facewin
Isometric Curve
Edgewin
Edge Curve

No esquema acima, cada box em linha cheia representa uma classe. As setas representam interconexões entre as instâncias das classes em questão, feitas por meio de ponteiros ou vetores de ponteiros (no caso de uma dada instância apontar para várias outras de uma mesma classe; por exemplo, um *shell* contém um vetor de ponteiros que apontam para cada uma de suas *faces)*.

5.1.1 Classes da estrutura 3D

Instâncias das classes da estrutura 3D representam objetos tridimensionais no espaço afim euclidiano de três dimensões E^3. Em cada uma destas instâncias estão embutidas estruturas para se descrever a "topologia" dos sólidos (informações de conectividade entre cada um de seus elementos) e estruturas para se descrever a "geometria" dos mesmos (informações relativas à "forma" dos seus elementos).

Um ponteiro global do tipo *scene* (cada uma destas instâncias representa uma cena, um conjunto enumerável de sólidos, organizados em *layers*) é usado para apontar para a cena corrente do aplicativo .

Futuramente, todas estas classes serão feitas serializáveis, de modo a permitir comunicação com memória externa; parte da estrutura necessária para isto já foi projetada e implementada, mas ainda não testada.

Cada sólido (uma instância do tipo shell) aponta para o *layer* que o contém. Além disso, o mesmo contém quatro *containers*: um para o conjunto de faces que compõem o sólido (instâncias do tipo *face*), outro para o conjunto de *winged-edges* e outro para o conjunto de vértices do sólido (instâncias do tipo *P3D*); o quarto *container* contém um conjunto de instâncias do tipo *TMatProjPlane.* Cada uma destas instâncias contém as matrizes de transformação das coordenadas de mundo do sólido (E^3) para as coordenadas associadas a cada um dos planos de projeção . Três tipos de coordenadas são associadas a cada plano de projeção:

• *eye coordinates* (E^3), coordenadas relativas ao sistema de coordenadas da câmera associada ao plano de projeção em questão;

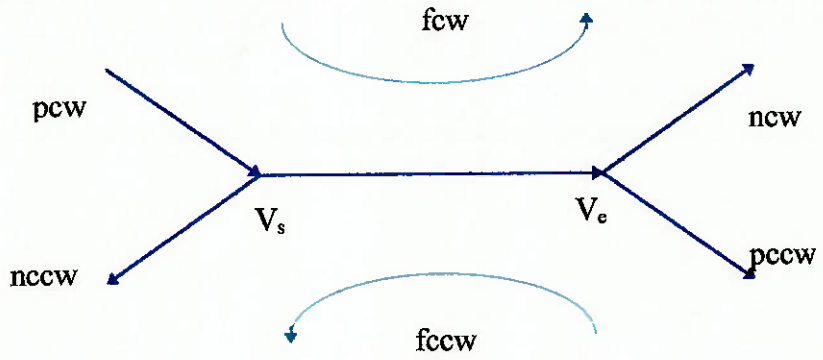
• *projection coordinates* (E^2), coordenadas relativas ao sistema de coordenadas do plano de projeção;

• *view coordinates* (E^2), coordenadas de tela.

Uma instância do tipo *face* representa uma face de algum sólido. Cada face possui um ponteiro para o sólido (*shell*) que a contém, um ponteiro para o seu *loop* externo e um *container* de *loops* internos. Em relação à geometria da face, a mesma poderá ser de dois tipos, a princípio: ou será a face plana ou uma *spline* de *patches* triangulares cúbicas de Bézier. Note que uma face plana pode ser considerada um tipo particular do segundo tipo de face, bastando para isto que os pontos de controle dos triângulos de Bézier estejam todos no mesmo plano; todavia, para fins de desenvolvimento do projeto, bem como por razões de eficiência dos algoritmos e utilização de memória, julgou-se conveniente separar os dois casos. Analogamente, observe que, escolhendo-se os pontos de controle convenientemente, as *patches* triangulares cúbicas de Bézier podem representar partes da superfície de qualquer quádrica, incluindo superfícies esféricas, cônicas, cilíndricas, etc.; no entanto, se for conveniente, tais casos também poderão ser considerados em separado. Até o momento, foram implementadas e testadas apenas as classes de face que descrevem faces planas, permitindo apenas a construção de sólidos poliédricos. Parte da estrutura necessária para descrever outros tipos de face já foi projetada e implementada, no entanto.

Instâncias do tipo *loop* representam os laços (*loops*) internos e externo que compõem cada face. Um *loop* tem um *flag* indicando se o mesmo é um *loop* interno ou externo; a referida classe é feita metamórfica, em alguns métodos (seleção, por exemplo), em função deste *flag*. Um *loop* também tem um ponteiro para a face à qual ele pertence, e um *container* de ponteiros para os *winged-edge*'s do mesmo.

Instâncias da classe *winged-edge* representam arestas do sólido, sua geometria e topologia (conectividade). Um objeto do tipo *winged-edge* pode ser descrito conforme o esquema abaixo:



Onde:

fcw e *fccw* - *face clockwise* e *face counter clockwise* são ponteiros para as duas faces adjacentes ao *winged-edge*;

pcw, ncw, pccw e *nccw* - *previous clockwise, next clockwise, previous counter clockwise* e *next counter clockwise* são ponteiros para os *winged-edge*'s adjacentes pertencentes às faces *fcw* e *fccw.*

Obs.: na implementação da referida estrutura, tomou-se *fccw* como a face com orientação coincidente com a do *edge* em questão, diferentemente do esquema acima.

Além destes ponteiros, um *winged-edge* também possui um ponteiro para o sólido (*shell*) ao qual pertence, e um *container* de instâncias do tipo *edgewin*, cada uma delas correspondendo à projeção da aresta representada pelo *winged-edge* em algum plano de projeção.

De forma análoga ao caso das faces, as arestas também poderão ter dois tipos de geometria: segmentos de reta ou cúbicas de Bézier. Novamente, o primeiro caso também pode ser incluído no segundo, mas considerações idênticas às feitas anteriormente também se aplicam aqui. Do mesmo modo, outros tipos de geometria também poderão ser derivados futuramente, caso seja conveniente.

Finalmente, a classe *P3D* será usada para representar pontos do E^3, e terá três subclasses, conforme o ponto seja um vértice de sólido, ponto de controle de face ou de aresta . Conterá um

container de ponteiros para instâncias da classe *P2D* (cada uma delas correspondendo à projeção do ponto em um dos planos de projeção).

5.1.2 Classes da estrutura 2D

A estrutura 2D possui cinco classes, a saber, *P2D*, *FaceWin*, *EdgeWin*, *IsoCurve* e *EdgeCurve*.

Instâncias do tipo *P2D* representam projeções dos pontos representados pelos objetos *P3D* correspondentes. Conterão um ponteiro para o *P3D* correspondente e outro para o objeto "*projection plane*" correspondente. Tais instâncias são construídas ou modificadas quando o objeto *P3D* correspondente for construído ou modificado. Quando uma instância do tipo *P3D* for construída, o construtor da referida classe deverá chamar uma rotina do objeto *projection plane* associado a cada um dos planos de projeção para calcular as coordenadas da sua projeção naquele plano, e a seguir chamará o construtor da classe *P2D* para construir uma instância da mesma.

As instâncias das classes *TEdgeWin* e *TFaceWin* correspondem a objetos resultantes da operação de *clipping* dos objetos *wedge* e *face* (no caso de uma face cuja superfície seja uma *patch* de Bézier, serão plotadas isométricas da mesma) da estrutura 3D, na *viewport* associada ao plano de projeção correspondente.

Cada objeto *TFaceWin* possui um container de objetos *TIsoCurve*, que por sua vez representa uma das curvas isométricas que serão desenhadas ao desenhar-se a face. Analogamente, cada objeto *TEdgeWin* possui um container de objetos *TEdgeCurve*, que representam cada um dos trechos desconexos da curva na *viewport* correspondente.

5.1.3 Considerações finais sobre o esquema de representação interno

Para finalizar esta seção, algumas considerações importantes sobre o modo como foi concebido o esquema de representação interno se fazem necessárias.

Inicialmente, note-se que muitas estruturas são redundantes, mas extremamente convenientes para alguns algoritmos que foram desenvolvidos e outros que o serão futuramente; além disso, quase todas as ligações (através de ponteiros) são duplas, para fins de rastreabilidade, caso isto seja necessário.

Outra questão relacionada ao uso de grande número de ponteiros é que grande parte dos objetos possui mais de um ponteiro apontando para eles. Isto é uma fonte de *bugs* em potencial, e por este motivo todos estes ponteiros são mantidos sob estrito controle. Para facilitar este controle, todas as classes em questão foram derivadas de uma classe denominada *Tlinkage*, que fornece um conjunto de dados-membro e métodos para que todas as ligações sejam mantidas automaticamente. Assim, por exemplo, quando um objeto de classe derivada de *Tlinkage* é deletado, todos os ponteiros que sejam membros de outros objetos *TLinkage* e que apontam para o referido objeto são automaticamente aterrados, e todos os *containers* são automaticamente atualizados. Esta proposta mostrou-se extremamente útil, não apenas por permitir implementação do código de forma mais sistemática e a prova de erros, mas também por facilitar sobremaneira a manutenção e modificação da estrutura de dados em questão. Tal solução introduz, todavia, um gasto extra de memória.

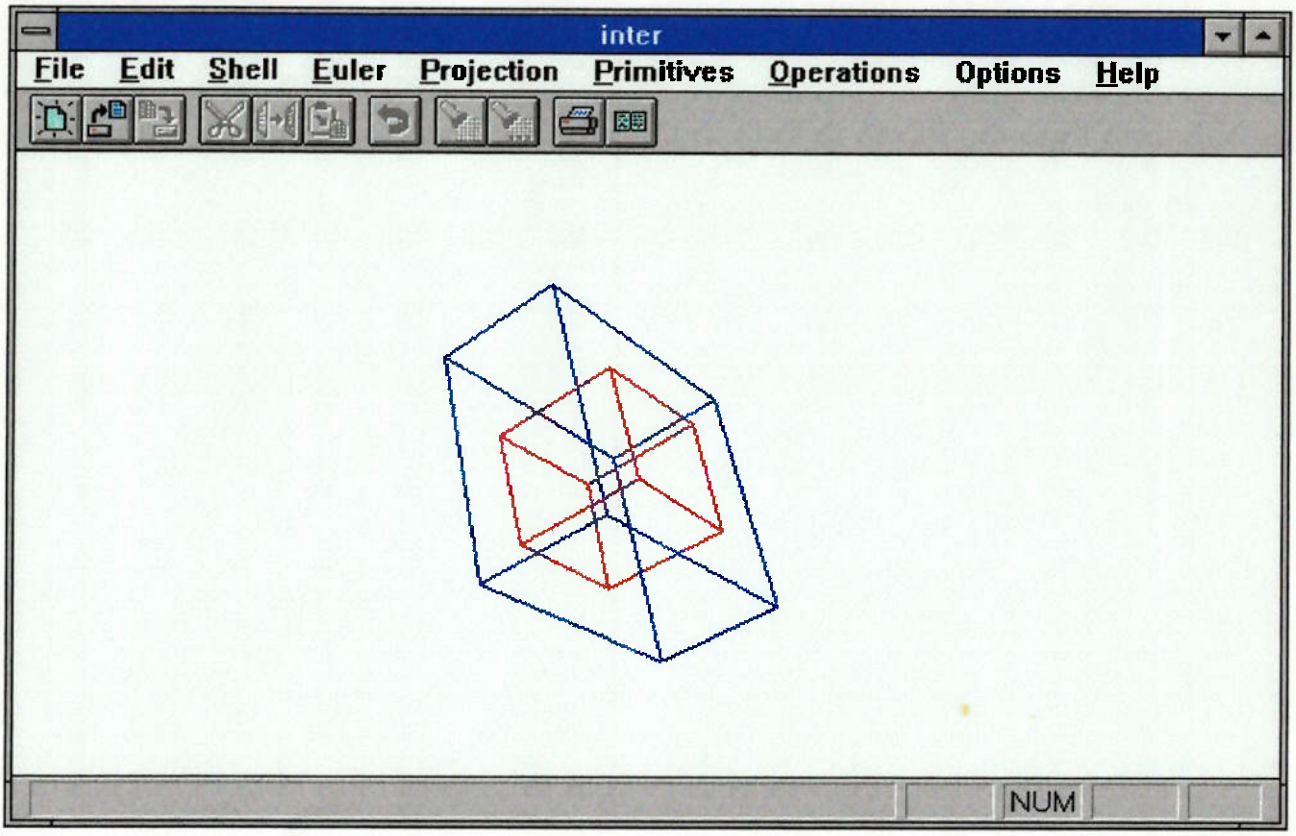
5.2 Módulos de interface externa, visualização e seleção

Conforme especificado no item "3", a entrada de dados é feita interativamente por meio de dispositivo apontador e ecoada no terminal de vídeo. Até o momento, ainda não foi desenvolvida comunicação com memória externa. No entanto, várias classes foram projetadas e implementadas como serializáveis ("*streamable*"), com o intuito de que tal comunicação possa ser feita futuramente sem grandes esforços de projeto. A serialização de alguns objetos permitirá, além disso, implementar uma ferramenta "*undo*".

O aplicativo é executado numa janela, a qual contém uma ou mais *viewports* em que as imagens dos objetos serão ecoadas; cada *viewport* (uma vista), corresponde a um objeto *TWindowData* , que por sua vez é um dado-membro de um objeto *TProjectionPlane*. Ou seja, a cada *viewport* está associado um plano de projeção, havendo liberdade para se posicionar a *viewport*

sobre o referido plano. Até o momento, é permitido o uso de apenas um plano de projeção (apenas uma vista), mas a estrutura para suportar a existência simultânea de vários planos de projeção já está implementada e testada, faltando apenas modificar a implementação das classes de interface *Windows* da OWL. Deve ser usada, provavelmente, uma interface *MDI - doc/view*, ou *SDI - doc/ view*; no último caso, cada cena (*"TScene"*) corresponderá a um documento (*"TDocument"*) e cada plano de projeção corresponderá a uma vista (*"TView"*) do referido documento.

O aplicativo é "orientado" a comandos, ou seja, a cada comando do usuário corresponde uma dada operação de construção, modificação ou visualização de um ou mais sólidos. A entrada de comandos é feita por meio de um sistema de menus e teclas aceleradoras, como o da figura abaixo.



Quando algum comando requisitar a entrada de algum ponto, isto é feito através do mouse: um click no botão esquerdo do mouse gera uma mensagem para que se tome o ponto do plano de construção corrente cuja projeção se encontra na posição do cursor do mouse na *viewport* (plano de

projeção) corrente no momento do click. Um cursor tridimensional facilita a visualização por parte do usuário do ponto cujas coordenadas estão sendo fornecidas.

O usuário tem total liberdade para mudar o plano de construção corrente, bem como para "amarrar" o cursor do mouse a um conjunto particular de pontos (uma linha, uma superfície não plana ou um conjunto discreto de pontos equiespaçados, por exemplo).

Com relação à seleção de objetos, tal operação é feita por meio de uma janela na *viewport* ativa (todos os objetos da *viewport* ativa que estiverem dentro de uma tal janela são selecionados). Para identificar quais os objetos que estão dentro de uma janela na *viewport* ativa, basta percorrer os objetos da estrutura *2D* correspondente e realizar uma operação de *clipping* para a janela de seleção. Ou seja, basta percorrer cada um dos containers *TP2dContainer*, *TEdgeWinContainer* e *TFaceWinContainer* do objeto *TProjectionPlane* associado à *viewport* em questão.

Note que os botões do mouse devem ser sobrecarregados, o que implica que as funções de resposta de mensagem associadas aos mesmos também devem ser. Isto exige o uso de diversos *flags* que indicam o estado do sistema (esperando um comando, esperando a entrada de um ponto, em modo de seleção, entrada no modo de seleção habilitada ou não, etc.); de acordo com o estado destes *flags*, as referidas funções de resposta de mensagem tomam uma ou outra ação.

5.3 Módulo de operações locais

Todas as operações passíveis de serem aplicadas aos sólidos - não apenas operações locais, mas também transformações de corpo rígido (isometrias) - foram implementadas através de objetos de classes serializáveis ("*streamable*"), para futuramente permitir comunicação com memória externa através de linguagem apropriada e implementação do "*undo*".

Num nível mais baixo, foram implementados operadores de Euler para construir os sólidos. No nível seguinte, algumas operações do tipo *sweep*, *lift*, *chamfer*, transformações de corpo rígido - rotação e translação. O conjunto de operações locais implementadas ainda está bastante restrito e precisa ser expandido - e não deve haver dificuldades para tal, uma vez que a estrutura já está montada.

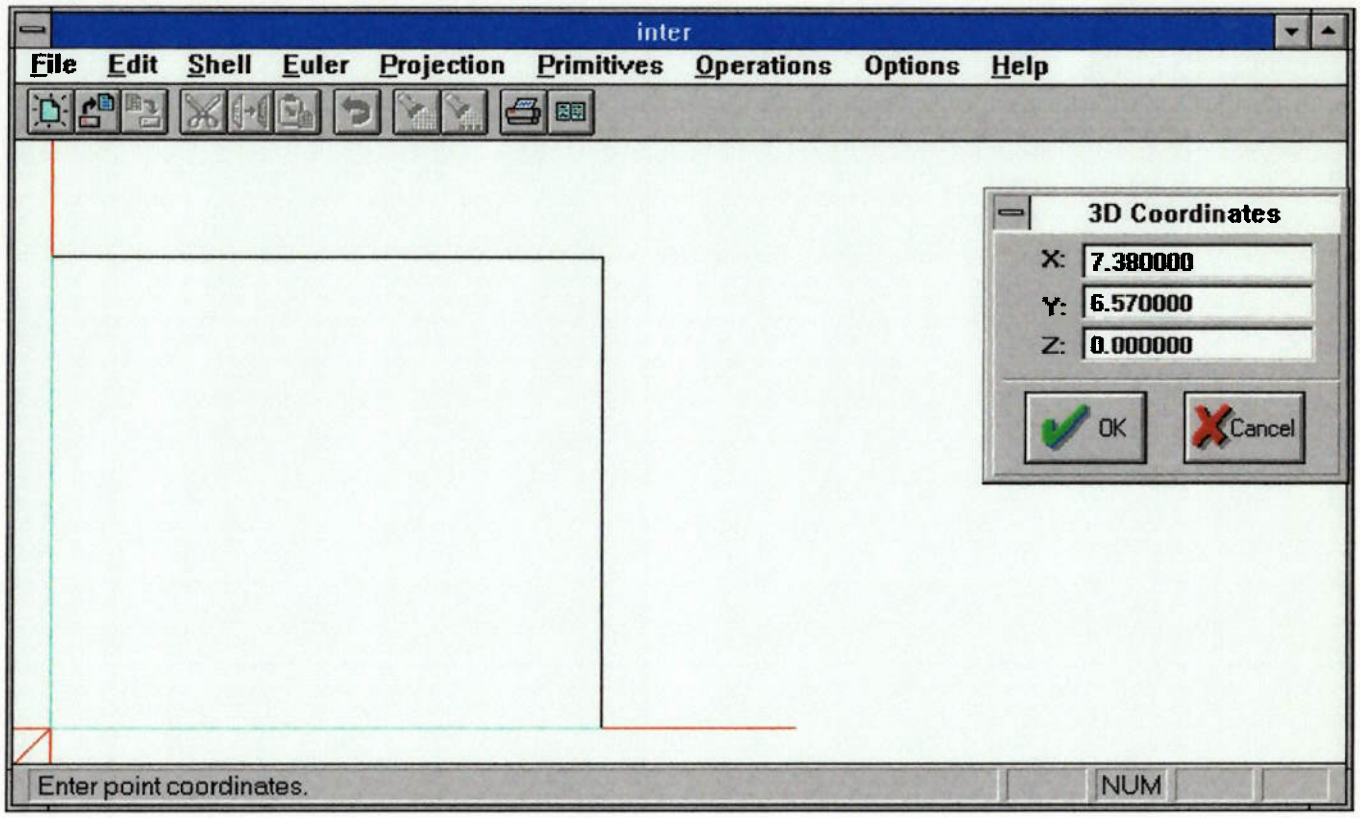
Cada comando que o usuário acessar via menu corresponde a uma ou mais operações locais. O que se propõe é que o usuário tenha , através destes comandos, a maior liberdade possível para efetuar operações tanto de alto como de baixo nível. Assim, o usuário tem acesso, via menu, aos próprios operadores de Euler; ou pode construir um sólido por meio de um primitivo e de operações locais sobre este primitivo.

Assim, por exemplo, o usuário pode acessar os próprios operadores de Euler para construir um sólido; ou pode construir uma face e aplicar a ela um *sweep*, resultando num sólido, ou colá-la a outras faces para também construir um sólido.

6. Como utilizar o software

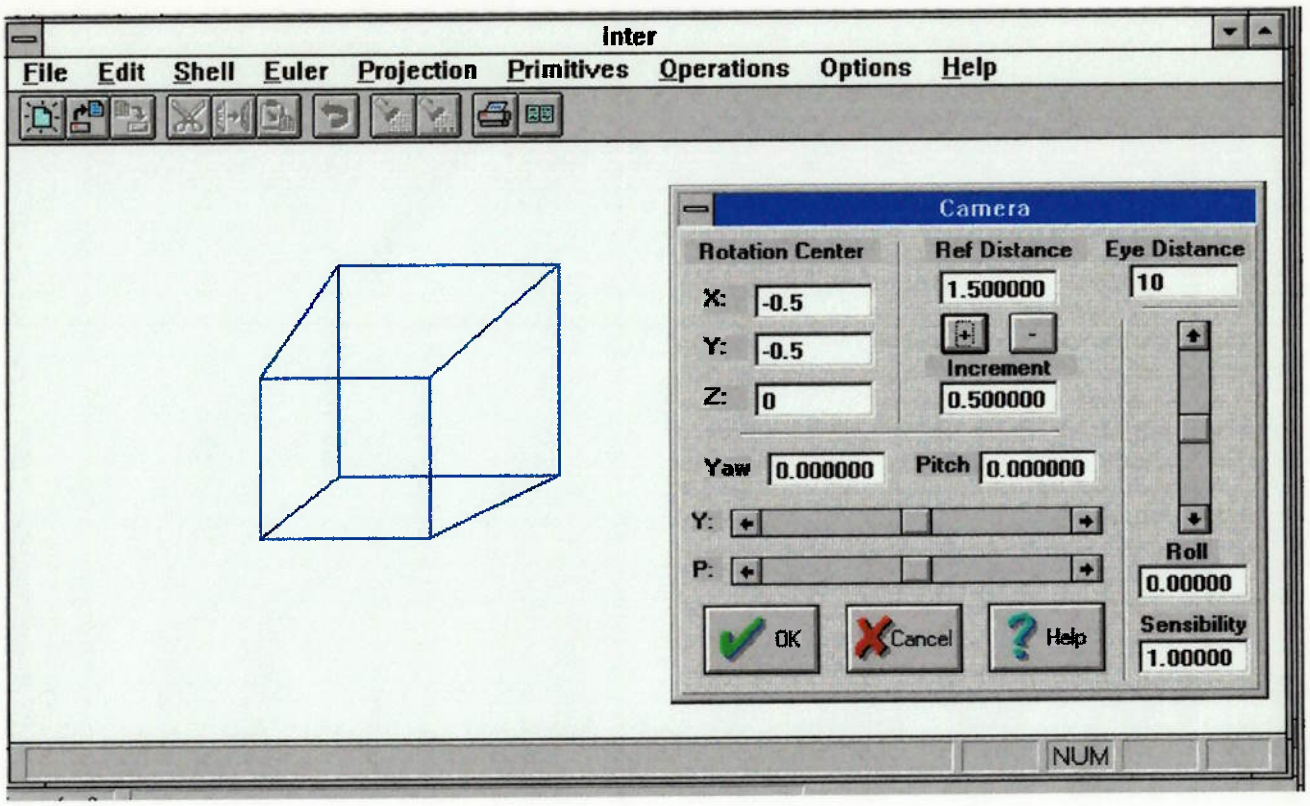
O software é bastante interativo e de fácil utilização, sendo que mensagens na barra de *status* indicam ao usuário o que cada comando faz ou que tipo de argumento ou operação estão sendo esperados. Além disso, qualquer operação pode ser cancelada em qualquer instante, bastando para tal clicar o botão "*cancel*", caso o mesmo esteja exibido, ou selecionar um conjunto vazio quando, por exemplo, em modo de seleção de algum tipo de entidade.

A utilização do software será ilustrada, a seguir, através de um exemplo, mostrando-se cada uma das etapas da construção de um sólido.



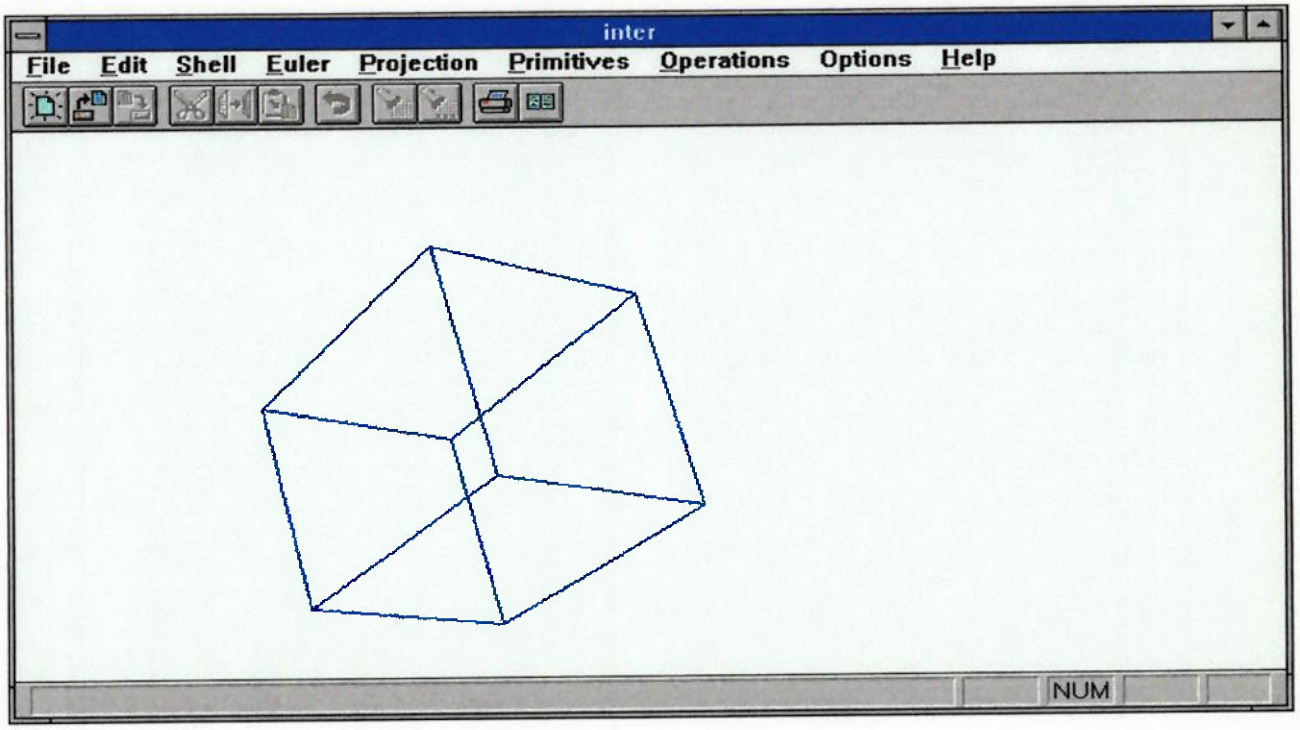
1

Foi selecionado o comando "*Cube*" no menu *"Primitives*"; a seguir, pede-se para que sejam fornecidas as coordenadas do vértice inferior esquerdo frontal do cubo a ser construído. A figura mostra o cursor neste momento, os eixos coordenados e uma caixa de diálogo com a posição atual do cursor. As coordenadas também podem ser dadas numericamente, bastando para tal preencher os *edit boxes* da referida caixa de diálogo. Para cancelar a operação, basta clicar no botão *"cancel*".



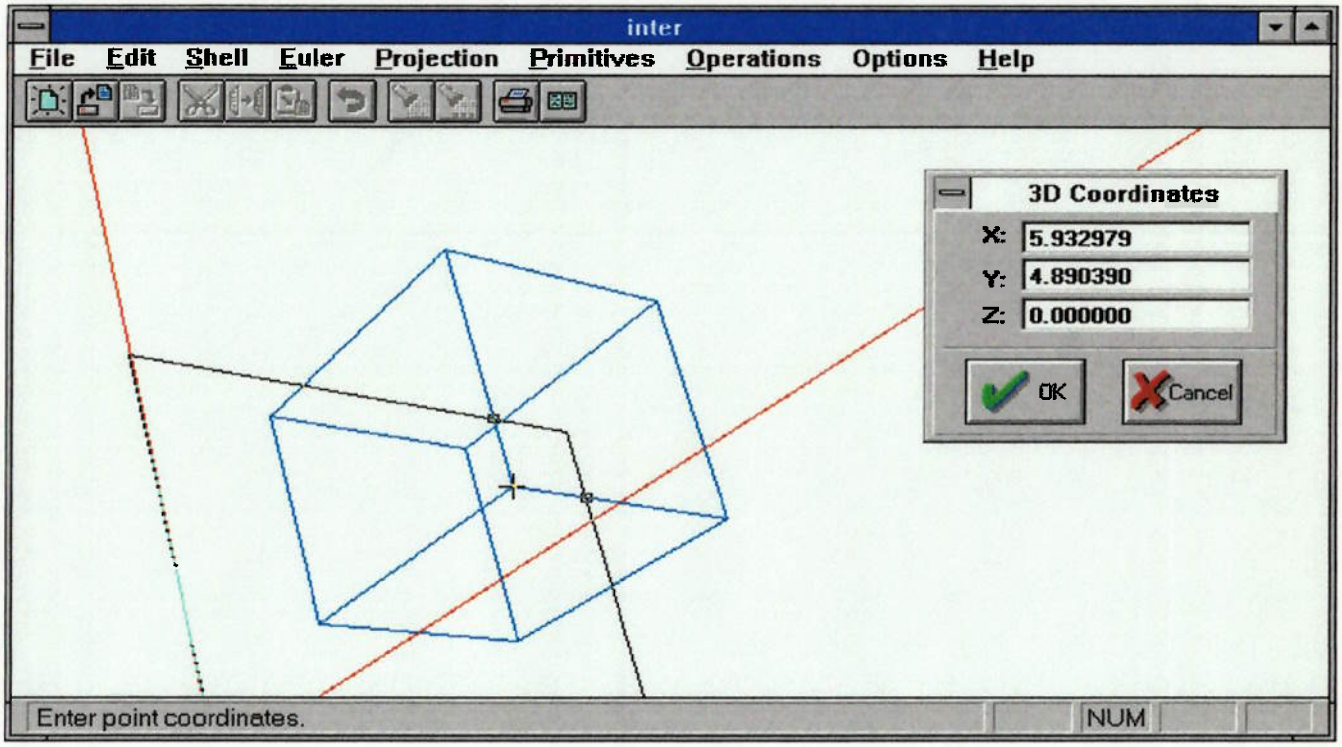
2

Estado do aplicativo após a construção do cubo e chamada ao comando "*camera*" para reposicionamento da câmera associada ao plano de projeção corrente.



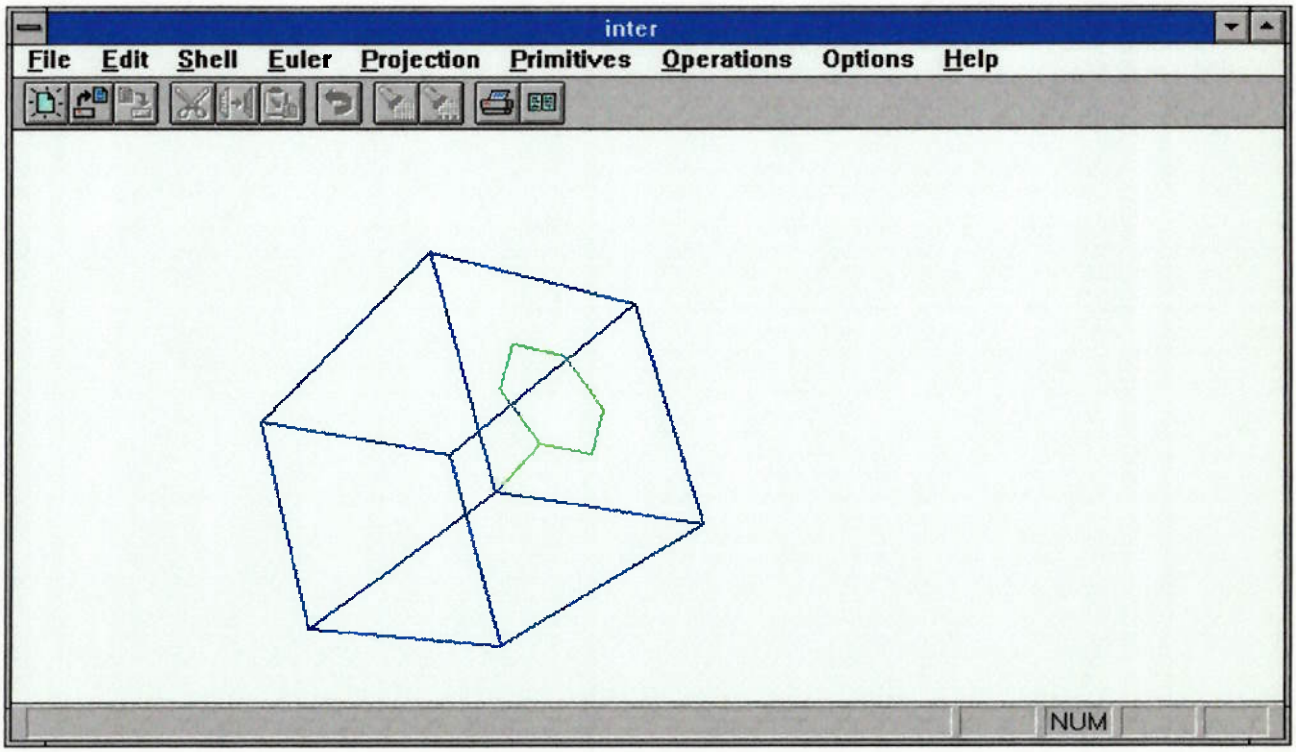
3

Estado do aplicativo após reposicionamento da câmera.



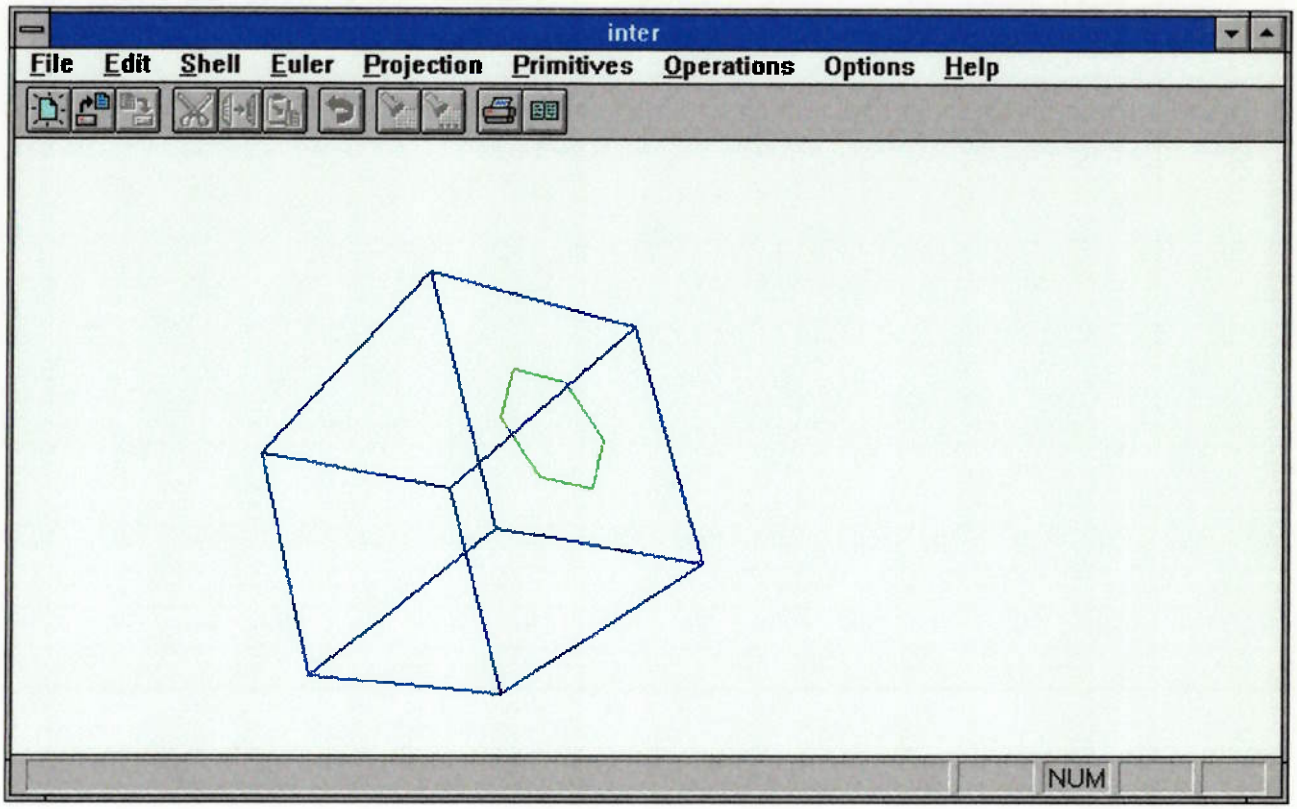
4

Foi selecionado o comando *MEV* (*make edge, vertex*); a seguir, foi selecionado um vértice (vértice marcado com uma cruz, na figura) e o comando pede para que sejam fornecidas as coordenadas do ponto final do *edge*. A figura mostra o estado do software neste instante.

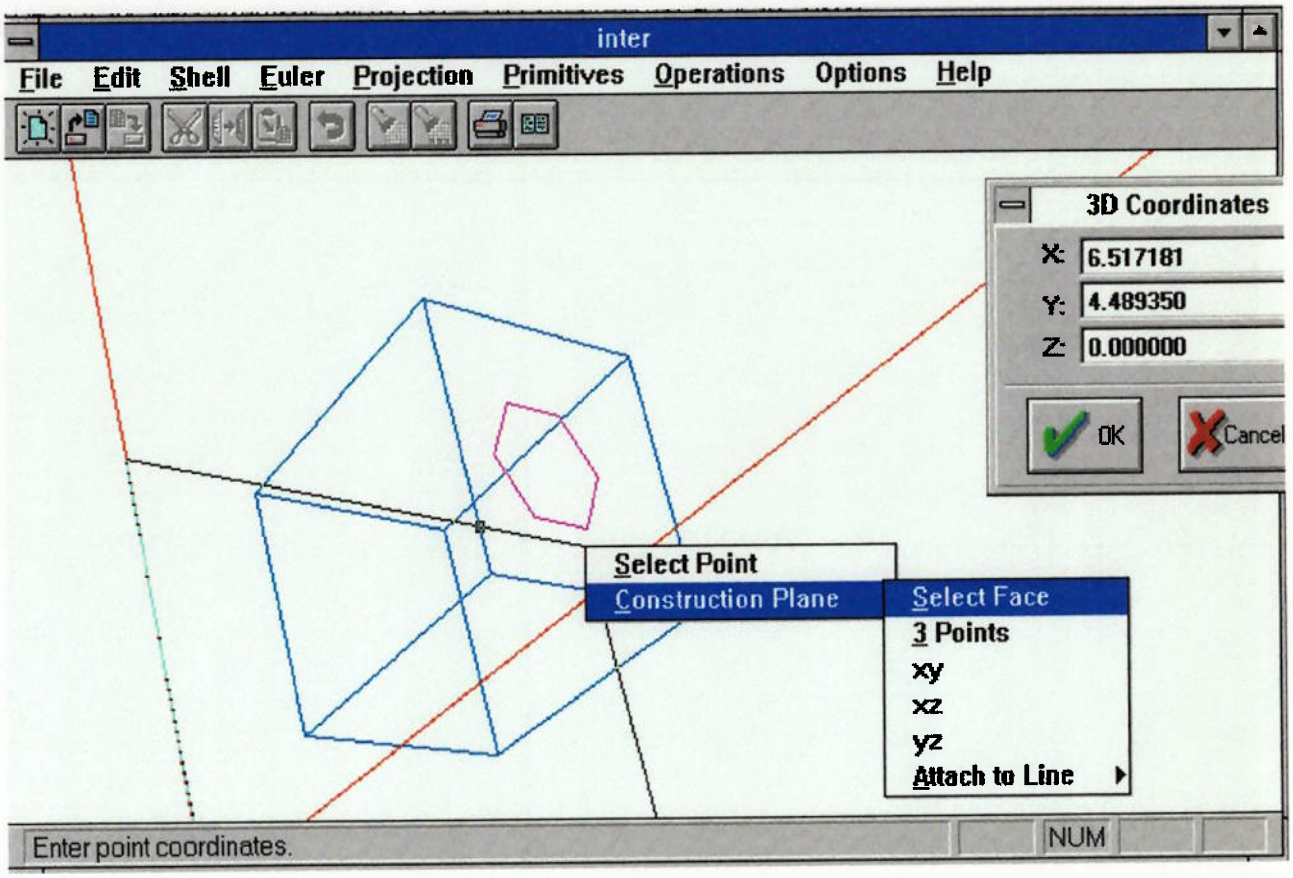


5

Estado após operação *MEV* e construção de uma face hexagonal através do comando *"face"* (menu *"primitives"*).

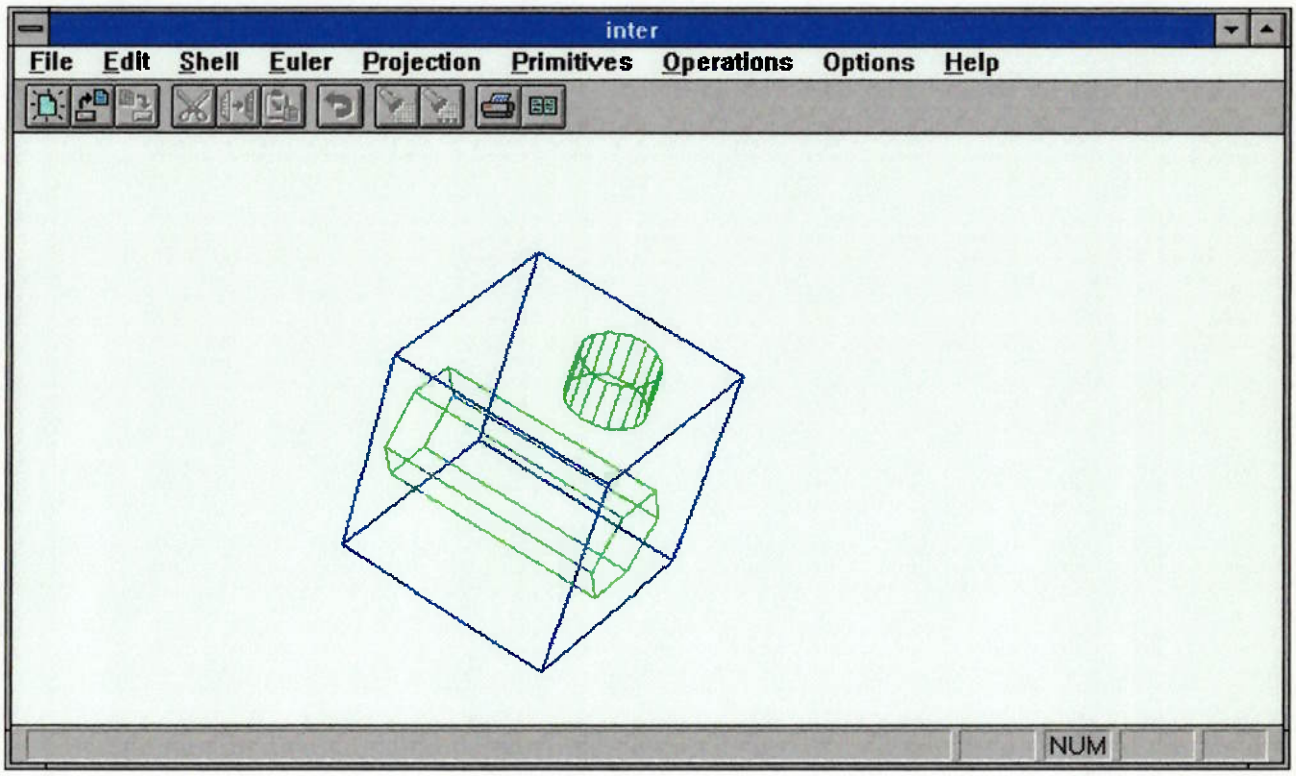


6
Estado após operação *KEMR* (*kill edge, make ring*).



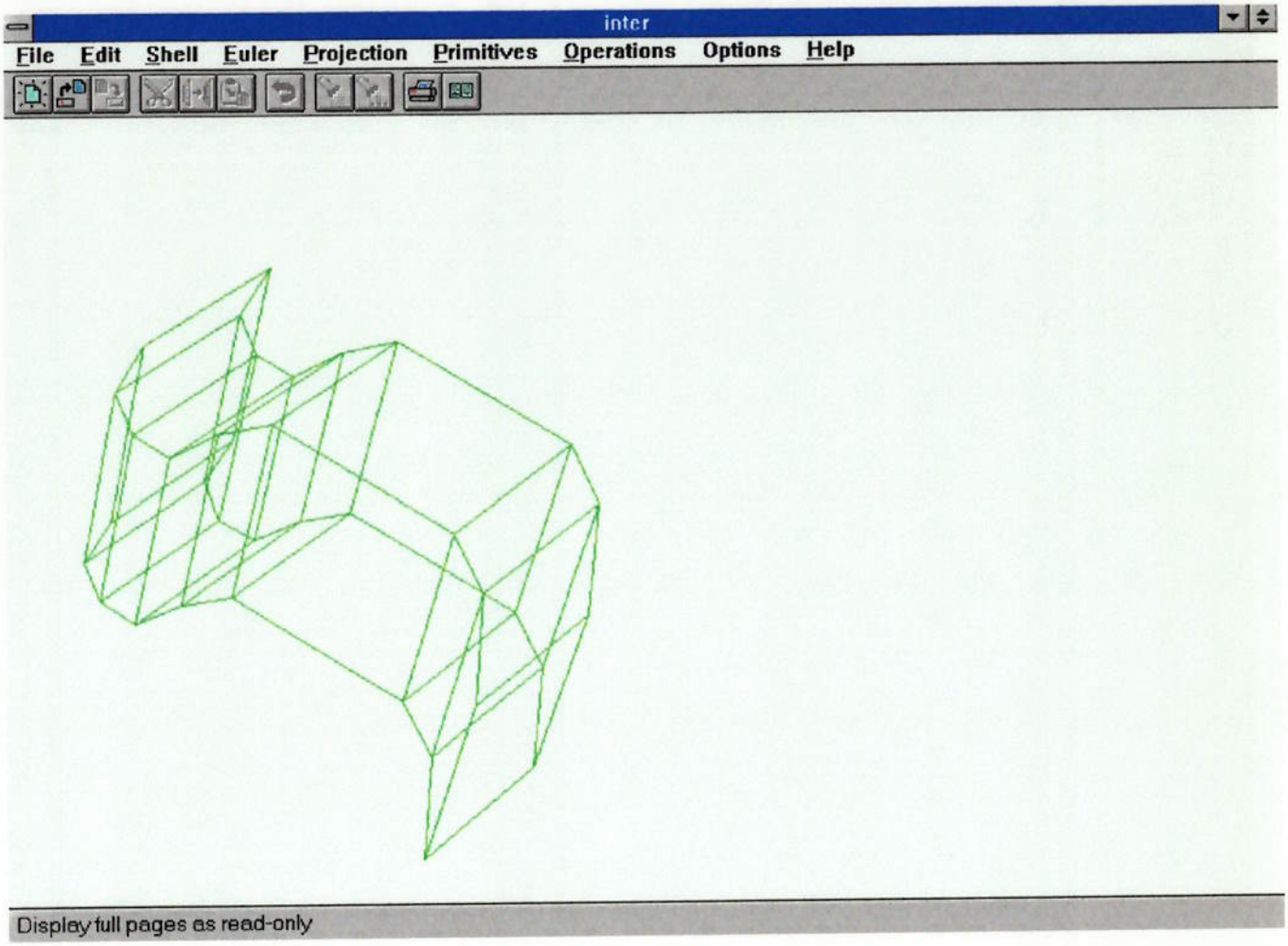
7

Foi chamado o comando *"sweep"* e selecionada a face hexagonal. O comando pede, então, um vetor de translação, que será fornecido através de dois pontos. Um deles foi tomado sobre a referida face, e o outro será tomado sobre a face posterior do cubo.



8

Estado após o *"sweep"* e operação *KFMRH* (*kill face, make ring, hole*), fazendo-se um orifício no cubo. A figura também mostra outra face que foi construída na face superior do cubo e à qual foi aplicado outro *"sweep"*, desta vez para "fora" do cubo. A câmera foi reposicionada para uma posição que permitisse melhor ângulo de visualização.



9 - Outro sólido construído com *sweep*'s.

Finalizar-se-á este pequeno "tutorial" do software com uma sucinta explicação de cada um dos comandos.

6.1 Menu Shell

O menu *"shell"* contém comandos para criar, abrir, fechar e destruir um sólido.

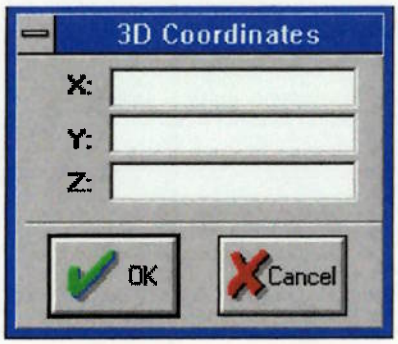
• ***new*** : cria um sólido(*shell*) aberto através do operador *MVS* (*make vertex, shell*).

Um sólido aberto é o que se obtém da remoção de uma face de um sólido fechado, apresentando característica de Euler 1 ($v - e + f = 1$). Os operadores de Euler foram modificados para operar sobre sólidos abertos (isto está formalizado no apêndice "A") ; todos os comandos que chamam tais operadores (incluídos no menu "*Euler*") são desabilitados quando o sólido está fechado. Assim sendo, para aplicar algum operador de Euler sobre algum sólido que esteja fechado, é necessário abrir o mesmo em uma de suas faces com o comando "*open*". Isto foi feito desta forma para facilitar a verificação dos resultados das operações.

O comando *new* pede que sejam fornecidas as coordenados de um ponto como argumento.

Obs.: sempre que for pedido para serem fornecidas as coordenadas de um ponto, isto pode ser feito de três maneiras:

1. Movimentando-se o cursor tridimensional até o ponto desejado e clicando-se o botão esquerdo do mouse. O cursor sempre se movimenta sobre um plano, denominado plano de construção. Inicialmente, o plano de construção é dado por $z = 0$. Para mudá-lo, clique o botão direito do mouse quando estiver no modo de entrada de coordenadas, e aparecerá um menu flutuante como o mostrado numa figura anterior; selecione, então, a opção "*costruction plane*" e a seguir uma das subopções que aparecerem (auto-explicativas).
2. Fornecendo-se as coordenadas numericamente, através da seguinte caixa de diálogo que aparece quando no modo de entrada de coordenadas e que fornece a posição atual do cursor:



3. Selecionando-se um ponto já existente, caso em que serão tomadas as coordenadas do mesmo.

• ***close*** : pede para que seja selecionado um sólido aberto e fecha o mesmo. Antes de fechá-lo, é feita uma verificação e, caso a operação não seja válida (por conduzir a um sólido inválido), a operação é cancelada e gera-se uma mensagem de erro.

• ***open*** : pede para que seja selecionada uma face de um sólido fechado e abre o sólido na referida face.

• ***destroy*** : pede para que seja selecionado um sólido e detrói o mesmo.

6.2 Menu "Euler"

No menu "*Euler*" foram incluídos os seguintes operadores de Euler:

• ***mev:*** make edge, vertex. Deve-se selecionar um ponto (*VStart* do *edge* a ser construído) e fornecer as coordenadas de um outro ponto (*Vend*);

• ***mef:*** make edge, face. Pede como argumentos:

1. seleção de um ponto (*VStart* do *edge* a ser construído);

2. coordenadas de um outro ponto (*Vend*);

3. seleção de um conjunto de *edges* que constituirão, em conjunto com o *edge* a ser construído, o *loop* externo da face a ser construída.

Caso os argumentos fornecidos não sejam consistentes, a operação é cancelada e gera-se uma mensagem de erro. Além disso, como já foi mencionado, qualquer operação pode ser cancelada através do botão "*cancel*", quando em modo de entrada de coordenadas, ou selecionando-se um conjunto vazio, quando em modo de seleção.

• ***kemr:*** kill edge, make ring. Pede como argumentos:

1. seleção do *edge* a ser deletado;

2. seleção da face à qual será incorporado o *ring*;

3. seleção de um conjunto de *edges*, os quais constituirão o *ring* a ser construído.

Analogamente ao comando anterior, caso os argumentos fornecidos não sejam consistentes, a operação é cancelada.

• ***kfmrh:*** kill face, make ring, hole. Pede como argumentos:

1. seleção da face a ser deletada;

2. seleção da face à qual será incorporado o *ring*.

• ***kev:*** kill edge, vertex. Deve-se fornecer como argumentos o *edge* e o *vertex* a serem deletados.

• ***kef:*** kill edge, face. Deve-se fornecer como argumentos o *edge* e a *face* a serem deletados.

• ***mekr:*** make edge, kill ring. Deve-se fornecer como argumentos os vértices inicial e final do *edge* a ser construído, bem como selecionar-se o *ring* a ser deletado.

- ***mfkrh:*** make face, kill ring, hole. Deve-se apenas selecionar o *ring* a ser deletado.

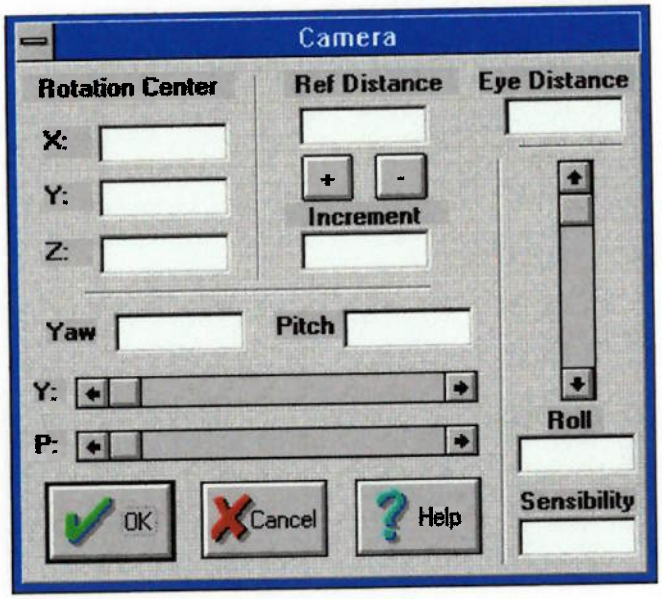
6.3 *Menu "Projection"*

No menu "*projection*" foram incluídos alguns comandos de visualização.

- ***pan:*** desloca a *viewport* ativa sobre o plano de projeção a ela associado. Deve-se fornecer um vetor de translação, através das coordenadas de dois pontos.

- ***zoom:*** aumenta ou diminui o tamanho da *viewport* ativa sobre o plano de projeção a ela associado.

- ***camera:*** chama a caixa de diálogo "*camera*", para que seja alterada a posição da câmera do plano de projeção associado à *viewport* ativa.



6.4 *Menu "Primitives"*

No menu "*primitives*" foram incluídos comandos para construção de sólidos ou faces inteiros. Até o momento, foram desenvolvidos apenas dois comandos:

• ***cube:*** constrói um cubo com faces paralelas aos planos coordenados. Para construir-se um cubo numa posição genérica, basta construir um cubo com este comando e em seguida usar operações de rotação e translação adequadas. Deve-se fornecer dois pontos como argumentos: o primeiro é o vértice inferior esquerdo frontal do cubo, e o segundo é usado apenas para determinar o comprimento do lado do mesmo.

• ***face:*** constrói uma face poligonal regular no plano de construção corrente. Deve-se fornecer como argumentos:

1. dois primeiros vértices da face (o primeiro deve ser um vértice já existente);

2. ponto para indicar o semi-plano do plano de construção corrente em que serão construídos os demais vértices.

O número de lados da face deve ser fornecido através do comando "*general options*" do menu "*options*". O *default* é 6.

Futuramente, novos comandos para construção de primitivos deverão ser desenvolvidos; isto deverá ser feito facilmente, uma vez que os primitivos são construídos a partir de uma mera seqüência de operadores de Euler, e estes já foram implementados.

6.5 Menu "Operations"

Neste menu foram incluídos comandos para efetuar operações sobre sólidos existentes: operações locais e transformações de corpo rígido (rotação e translação).

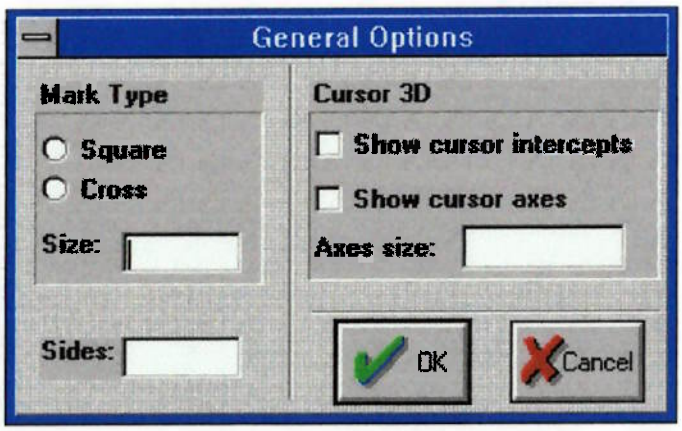
- ***sweep:*** executa um "*sweep*" sobre uma face selecionada. Deve-se fornecer como argumentos uma face e um vetor de translação.

- ***rotate:*** rotaciona um sólido selecionado em torno de um eixo.

Até o momento, o número de operações locais implementadas ainda é muito restrito. Para que a interface seja útil, outros tipos de operações locais devem ser implementados.

6.6 Menu "Options"

Neste menu foram incluídos comandos para alterar parâmetros tais como cor dos objetos a serem construídos, número de lados das faces construídas com o comando "*face*", tamanho das marcas de seleção de pontos e tamanho do cursor de seleção, entre outros.



7. Descrição do código fonte

Uma vez apresentado o funcionamento do software, o passo seguinte será uma descrição em detalhes das principais classes, algoritmos e estruturas de dados usados em sua implementação.

7.1 Pares e ternas ordenadas

O arquivo "`pontos.h`" contém declarações de *templates* (gabaritos) de classes de pares ordenados (`class TParOrdenado<T>`) e ternas ordenadas (`class TTernaOrdenada<T>`, derivada publicamente de `TParOrdenado<T>`).

Foram definidos construtores *default* e um construtor que define o objeto a partir de cada um dos elementos do par ou da terna. Na classe `TTernaOrdenada<T>` também foi definido um construtor que toma como argumento um `TParOrdenado<T>`. Neste caso, constrói-se uma terna ordenada a partir dos dois primeiros elementos de um par ordenado, ficando o terceiro elemento indefinido (ou nulo, caso os elementos da terna sejam números).

Sobrecarregou-se o operador de igualdade em função do tipo de par ou terna ordenada. Assim, se o tipo do par ordenado for `int` (ou seja, `TParOrdenado<int>`), o operador simplesmente testa a igualdade de cada elemento do par (ou da terna). Se o tipo do par ordenado for `float` ou `double`, o operador usa uma função global denominada `compareDouble(...)` usada para testar igualdade de números de ponto flutuante em função de uma tolerância. Se o tipo do par ou terna ordenada for outro, o operador de igualdade sobrecarregado chama o operador de igualdade da classe de base de `TParOrdenado<T>`, `TLinkage` (esta classe será explicada posteriormente).

Alguns `typedef`'s também foram definidos no arquivo "`pontos.h`". Os principais são:

```
typedef TParOrdenado<int> TParInt;
typedef TParOrdenado<float> TParFloat;
typedef TTernaOrdenada<int> TTernaInt;
typedef TTernaOrdenada<float> TTernaFloat;
typedef TArrayAsVector< TParOrdenado<float> > TParFloatContainer;
typedef TarrayAsVector< TTernaOrdenada<float> > TTernaFloatContainer;
```

Para maiores detalhes sobre a implementação destes *templates*, consulte a listagem.

7.2 Matrizes de transformação homogênea

Matrizes de transformação homogênea foram implementadas através da classe `TMatTrans` e de suas derivadas (um sumário sobre matrizes de transformação homogênea encontra-se no apêndice "B").

A classe `TMatTrans` contém uma matriz 4x4 de `float`'s protegida, na qual são armazenados os elementos da matriz de transformação. Foi definido um construtor *default* e outro que define a matriz a partir de um vetor `float[4][4]`. Foi também sobrecarregado o operador `[.]` (índice), para ser possível acessar os elementos da matriz através de uma sintaxe conveniente e de fácil leitura.

Foram derivadas publicamente de `TMatTrans` as seguintes classes:

- `TMatRotx`, `TMatRoty` e `TMatRotz`, matrizes de rotação em torno dos eixos x, y e z, respectivamente. Seus construtores tomam o ângulo de rotação como parâmetro.

- `TMatTranslation`, matriz de translação. Seus construtores tomam como parâmetro três `float`'s ou um objeto `TTernaFloat`, que são as coordenadas do vetor de translação.

- `TMatScaling`, matriz de escalonamento. Seus construtores tomam como parâmetro três `float`'s ou um objeto `TTernaFloat`, que são os fatores de escala nas direções x, y e z.

- `TMatPerspective`, matriz de projeção perspectiva. Seu construtor toma como parâmetro um `float`, distância do centro de projeção ao plano de projeção.

As declarações destas classes encontram-se no arquivo de cabeçalho "`mattrans.h`".

7.3 Funções vetoriais

No arquivo "`vector.h`" encontram-se declaradas várias funções globais para manipulação de vetores:

```
extern float _FAR sqr(float x);
extern double _FAR sqrd(double x);
extern long double _FAR sqrld(long double x);
```

Estas funções dispensam explicações.

```
extern float _FAR norma(const TTernaFloat _FAR &);
extern float _FAR norma(const TParFloat _FAR &);
```

Retornam a norma euclidiana de um ponto passado como parâmetro (`TTernaFloat`).

```
extern TTernaFloat _FAR cross(const TTernaFloat _FAR & p1,const TTernaFloat _FAR & p2);
```

Retorna produto vetorial `p1^p2` .

```
extern float _FAR dot(const TTernaFloat _FAR & p1,const TTernaFloat _FAR & p2);
extern float _FAR dot(const TParFloat _FAR & p1,const TParFloat _FAR & p2);
```

Retornam produto escalar `<p1,p2>`.

```
extern TTernaFloat _FAR versor(const TTernaFloat _FAR & p1);
```

Retorna versor na direção e sentido de `p1`.

```
extern TTernaFloat _FAR operator+(const TTernaFloat _FAR & p1,const TTernaFloat _FAR &
p2);
```

Retorna `p1+p2`.

```
extern TTernaFloat _FAR operator-(const TTernaFloat _FAR & p1,const TTernaFloat _FAR &
p2);
```

Retorna `p1-p2`.

```
extern TTernaFloat _FAR operator-(const TTernaFloat _FAR & p1);
```

Retorna `-p1`.

```
extern TTernaFloat _FAR operator*(float a,const TTernaFloat _FAR & p);
extern TTernaFloat _FAR operator*(const TTernaFloat _FAR & p, float a);
```

Retornam `a.p` (`a` escalar, `p` vetor).

```
extern TTernaFloat _FAR operator/(const TTernaFloat _FAR & p, float a);
```

Retorna `p/a` (`a` escalar, `p` vetor).

```
extern TTernaFloat _FAR operator*(const TTernaFloat _FAR & pin,const TMatTrans _FAR &
m);
```

Retorna `p.m`.

```
extern TMatTrans _FAR operator*(const TMatTrans _FAR & m1,const TMatTrans _FAR & m2);
 /* retorna m1*m2 */
```

Retorna `m1.m2`.

```
extern TMatTrans _FAR transp(const TMatTrans _FAR & m);
```

Retorna transposta de `m`.

```
extern bool _FAR findSegmentsIntersection(TTernaFloat P1, TTernaFloat P2,
          TTernaFloat P3, TTernaFloat P4, TTernaFloat _FAR & Pinter, bool bPlane
=true);
```

Esta função determina a intersecção de dois segmentos `(p1,p2)` e `(p3,p4)`. É usada pela classe do cursor de seleção (`TCursor2D`) para selecionar *edges* na *viewport* ativa. Retorna um valor booleano que indica se encontrou ou não intersecção; caso tenha encontrado, o ponto de intersecção é retornado no parâmetro `Pinter` (caso a intersecção seja um segmento, apenas uma das extremidades do mesmo é retornada, pois isto já é suficiente para os propósitos aos quais esta função se destina no programa). O parâmetro `bPlane` é um flag que indica se os segmentos estão no plano z=0 ou não, pois o algoritmo é diferente em um e outro caso.

O algoritmo é o seguinte:

Consideremos a seguinte parametrização para os segmentos:

$$\begin{aligned} P:[0,1] &\rightarrow E^3 \\ \lambda &\mapsto P1+\lambda\,(P2-P1) \end{aligned}$$

$$\begin{aligned} P^*:[0,1] &\rightarrow E^3 \\ \xi &\mapsto P3+\xi\,(P4-P3) \end{aligned}$$

Então:

$$P(\lambda)=P^*(\xi)\Leftrightarrow$$
$$P1+\lambda\,(P2-P1)=P3+\xi\,(P3-P4)\Leftrightarrow$$
$$\lambda\,(P2-P1)+\xi\,(P3-P4)=P3-P1$$

Fazendo-se

$$P2-P1=\vec{a}$$
$$P3-P4=\vec{b}$$

e $P3-P1=\vec{c}$

devemos resolver a equação vetorial

$$\lambda\,\vec{a}+\xi\,\vec{b}=\vec{c} \quad (*)$$

Caso os vetores estejam no plano, teremos um sistema linear de duas equações e duas incógnitas. Caso não seja este o caso, deveremos achar a projeção de `c` no subespaço formado por `a` e `b`, e depois verificar se ela é solução. Para encontrar a projeção, devemos determinar a matriz Grammiana de `a` e `b` e resolver o seguinte sistema:

$$G \cdot \begin{bmatrix} \lambda \\ \xi \end{bmatrix} = \begin{bmatrix} \langle \vec{a}, \vec{c} \rangle \\ \langle \vec{b}, \vec{c} \rangle \end{bmatrix}$$

onde

$$G = \begin{bmatrix} \langle \vec{a}, \vec{a} \rangle & \langle \vec{a}, \vec{b} \rangle \\ \langle \vec{b}, \vec{a} \rangle & \langle \vec{b}, \vec{b} \rangle \end{bmatrix}$$

é a matriz Grammiana de `a` e `b` e `<.,.>` denota produto escalar.

Resolvido o sistema, devemos verificar se λ e ξ estão entre 0 e 1, e, no segundo caso (vetores fora do plano), verificar se a projeção satisfaz (*).

```
extern bool _FAR findSegPolInter(TTernaFloat P1, TTernaFloat P2,
             const TTernaFloatContainer _FAR & vTerna, TTernaFloat _FAR & Pinter);
```

Esta função determina a intersecção de um segmentos `(p1,p2)` e uma face poligonal plana cujos vértices são dados pelo container `vTerna`. É usada pela classe do cursor tridimensional (`TCursor3D`) para determinar a intersecção do cursor com cada face de cada sólido, de modo a fornecer informações visuais em 3D para o usuário. Retorna um valor booleano que indica se encontrou ou não intersecção; caso tenha encontrado, o ponto de intersecção é retornado no parâmetro `Pinter`. Caso a solução seja indeterminada, o algoritmo retorna `false` e não encontra solução alguma, pois isto não é necessário para os objetivos a que se presta esta função no programa. Seria interessante otimizar o algoritmo proposto, ou mesmo usar outro que seja mais eficiente, pois quando o número de faces é muito grande o programa fica muito lento.

O algoritmo consiste em dividir-se o polígono numa seqüência de triângulos, e percorrer esta seqüência até que se ache a intersecção com algum triângulo ou até que termine a seqüência sem que nenhuma intersecção seja achada. O procedimento para se determinar a intersecção do segmento com um triângulo é o que segue:

Consideremos a seguinte parametrização para o segmento:

$$\begin{aligned} P:[0,1] &\rightarrow E^3 \\ \lambda &\mapsto P1+\lambda\,(P2-P1) \end{aligned}$$

e para o triângulo (dado pelos pontos `T1`, `T2` e `T3`):

$$\begin{aligned} T:[0,1]\times[0,1] &\rightarrow E^3 \\ (\alpha,\beta) &\mapsto T1+\alpha(T2-T1)+\beta(T3-T2) \end{aligned}$$

Então:

$$P(\lambda)=T(\alpha,\beta)\Leftrightarrow$$
$$\alpha(T2-T1)+\beta(T3-T2)+\lambda\,(P1-P2)=P1-T1$$

Analogamente ao caso anterior, isto se resume a resolver

$$\alpha\vec{a}+\beta\vec{b}+\lambda\,\vec{c}=\vec{d}$$

Após resolver o sistema linear 3x3, devemos verificar se α, β e λ estão entre 0 e 1.

7.4 Classes *TPlane* e *TWindowData*

As classes `TPlane` (e derivadas) e `TWindowData` são responsáveis pela interface entre a estrutura `2D` e a interface Windows da *OWL*, além de terem participação no mecanismo de projeção da estrutura `3D` para a estrutura `2D`. Tais classes encontram-se declaradas nos arquivos "`plane.h`" e "`window.h`", respectivamente.

Classe TPlane.

`Objetos da classe` `TPlane` representam planos, como sugerido pelo nome. Esta classe, derivada de `TLinkage` (descrita mais adiante), possui dois membros protegidos, `TTernaFloat p,n;` `p` é

um ponto do plano e `n` é o versor normal ao plano. Há dois construtores públicos: um construtor *default* e outro que constrói o plano a partir de `p` e `n`. A classe também possui as seguintes funções-membro públicas:

```
TTernaFloat getp(void) const;
```

Retorna `p`.

```
TTernaFloat getn(void) const;
```

Retorna `n`.

```
void setp(const TTernaFloat& p);
```

Altera `p`.

```
void setn(const TTernaFloat& n);
```

Altera `n`.

```
bool isOnPlane(const TTernaFloat& p);
```

Verifica se o ponto passado como parâmetro pertence ao plano.

```
float distToPlane(const TTernaFloat& p);
```

Retorna distância do ponto passado como parâmetro ao plano; o sinal é dado pelo semi-plano em que estiver o ponto.

Classe TConstructionPlane

Derivada publicamente de `TPlane`; objetos desta classe representam planos de construção. Cada objeto `TConstructionPlane` está associado a um objeto `TProjectionPlane`, tendo um ponteiro protegido para o mesmo (que entra como parâmetro extra no construtor), além dos membros herdados. Possui os seguintes campos métodos públicos:

```
bool validPointCons;
```

Flag que indica se o ponto construído (coordenadas obtidas de `proj2cons(...)` ou `view2cons(...)`) é válido ou não.

```
void setpTProjPlane(TProjPlane*);
TProjPlane* getpTProjPlane(void) const;
```

Funções para acessar o ponteiro para o plano de projeção associado (`TProjPlane* pTProjPlane`).

```
bool setPlane(TTernaFloatContainer);
bool setPlane(TFlatFaceV*);
```

Alteram os parâmetros do plano (i.e., `p` e `n`) de modo que o plano seja coincidente com o plano de uma face (2a. função) ou com um plano dado por três pontos não colineares (1a. função).

```
TTernaFloat proj2cons(TTernaFloat);
TTernaFloat view2cons(TParInt);
```

view = view coordinates (coordenadas de tela, E^2) do ponto no plano de projeção
proj = projection coordinates (coordenadas de plano de projeção, E^2) do ponto no plano de projeção
cons = world coordinates (coordenadas de mundo, E^3) do ponto a ser construído.

Estas funções convertem as coordenadas de tela ou de plano de projeção de um ponto no plano de projeção, fornecido como parâmetro, em coordenadas de mundo, no presente plano de construção. São usadas para determinar as coordenadas de um ponto no E^3 a partir da posição do cursor tridimensional na tela.

O algoritmo usado na conversão é o seguinte:

Sejam P_0 e `n` os parâmetros que definem o plano de construção (i.e., um ponto do plano e versor normal). Sejam ainda `(eye, ref,up)` o triedro que define a câmera (vide classe `TProjPlane` para maiores detalhes) , P_p o ponto no plano de projeção e P_c o ponto no plano de construção que se deseja encontrar.

O plano de construção é dado por:

$$P_c \in E^3 / \langle P_c - P_0, \hat{n} \rangle = 0$$

A reta `(eye,`P_p`)` pode ser parametrizada por:

$$P: \Re \to E^3$$
$$\lambda \mapsto eye + (P_P - eye) \cdot \lambda \quad (*)$$

Então:

$$P = P_c \Leftrightarrow$$

$$\langle eye + (P_P - eye) \cdot \lambda - P_0, \hat{n} \rangle = 0 \Leftrightarrow$$

$$\lambda = \frac{\langle P_0 - eye, \hat{n} \rangle}{\langle P_P - eye, \hat{n} \rangle}$$

Basta substituir λ em (*) para encontrar o ponto no plano de construção.

Classe TProjPlane

Derivada de `TPlane`; objetos desta classe representam planos de construção. Além dos membros herdados, possui os seguintes membros:

```
/*************************************************************************
        dados-membro protegidos
 *************************************************************************/
TP2dContainer viTP2d;
```

Este container contém ponteiros para todos os `TP2d`'s associados a este plano de projeção. Cada objeto da classe `TP2d` é a projeção de um ponto do E^3 em algum plano de pojeção.

```
bool bProjCoordValid;
```

Flag que indica se as coordenadas de plano de projeção dos elementos da estrutura `2D` associada a este plano são válidas ou não. Quando o plano de plano de projeção ou algum sólido moda de posição, por exemplo, este *flag* é ressetado.

```
bool bPlaneModified;
```

Quando o presente plano de projeção muda de posição no espaço, este *flag* é setado.

```
bool bASolidModified;
```

Quando algum sólido muda de posição no espaço, este *flag* é setado.

```
TFlagSolidContainer vTFlagSolid;
```

Container de `TFlagSolid`'s. Cada objeto da classe `TFlagSolid` contém um ponteiro para um `TShell` e um *flag* indicando se o mesmo mudou de posição no espaço ou não. O container possui um `TFlagSolid` para cada sólido existente.

Este conjunto de *flags* (`bProjCoordValid`, `bPlaneModified`, `bASolidModified` e `vTFlagSolid`) é utilizado para atualizar as coordenadas de plano de projeção dos elementos da estrutura `2D` associada ao presente plano de projeção de uma maneira eficiente e no momento mais adequado.

Se todas as coordenadas fossem atualizadas no momento em que deixam de ser válidas, todos estes *flags* poderiam ser dispensados. Por exemplo, quando um sólido muda de posição no espaço, poder-se-ia atualizar imediatamente todas as coordenadas associadas aos seus pontos. Ou, quando o plano muda de posição, poder-se-ia , de forma análoga, atualizar todas as coordenadas de todos os elementos da estrutura `2D` a ele associada imediatamente. Mas, como estas atualizações consomem tempo, isto pode resultar num efeito visual pobre no momento da operação; quando o usuário rotacionar um sólido, por exemplo, vai desejar ver o sólido sendo rotacionado o mais rápido possível. Para tal, deve-se atualizar as coordenadas de tela imediatamente; as demais coordenadas são atualizadas depois, em segundo plano, quando o usuário não estiver fazendo nada. E, para que se possa obter as coordenadas de mundo de algum vértice do sólido que tiver mudado de posição e ainda não tiver suas coordenadas de mundo atualizadas, por exemplo, será associada uma matriz de transformação ao mesmo descrevendo a mudança de posição, de forma que as coordenadas atualizadas possam ser obtidas à medida em que forem necessárias ("*just in time*"). Além disso, nem todas as coordenadas de tela precisam ser atualizadas imediatamente; apenas aquelas associadas aos planos de projeção visíveis. As demais também podem ser atualizadas em segundo plano, ou, analogamente, "*just in time*", à medida em que forem necessárias.

O algoritmo de atualização das coordenadas em segundo plano é o seguinte:

Para atualizar-se as coordenadas de tela e de plano de projeção, percorre-se um vetor global de `TProjPlane*` e, através do sistema de *flags*, verifica-se se o plano precisa atualizar as coordenadas de plano de projeção a ele associadas. Se for necessário, chama-se uma rotina de atualização de coordenadas do objeto `TProjPlane` em questão, para que a mesma atualize as coordenadas de plano de projeção da estrutura `2D` associada. A necessidade ou não de serem atualizadas as coordenadas de tela também é verificada através de um sistema de *flags* semelhante, presente num objeto

`TWindowData`, membro público de `TProjPlane`, denominado `planeWindowData`. De forma análoga, chama-se uma rotina de atualização de coordenadas do membro `planeWindowData` de `TProjPlane` para que as coordenadas de tela sejam atualizadas, caso necessário. Mais precisamente, a necessidade ou não de se atualizarem as coordenadas de plano de projeção ou de tela é determinada em função de um único *flag*: `bProjCoordValid` (ou `bViewCoordValid` - vide classe `TWindowData` - no caso das coordenadas de tela). Os demais *flags* são utilizados para determinar quais elementos da estrutura `2D` associada devem ter suas coordenadas de tela ou de plano de projeção atualizadas. Por exemplo, quando um sólido mudar de posição no espaço, serão setados em cada um dos planos de projeção os `TFlagSolid`'s correspondentes (além de serem setados `bASolidModified` e ressetado `bProjCoordValid`). Se o plano de projeção não tiver mudado de posição no espaço, ao ser feita a atualização das coordenadas de plano de projeção dos elementos da estrutura `2D` associada, será apenas necessário atualizar as coordenadas dos sólidos que tiverem mudado de posição, não é preciso perder tempo de processamento atualizando coordenadas de sólidos que continuam válidas. Assim, basta percorrer o container de `TFlagSolid`'s e verificar quais sólidos precisam de atualização.

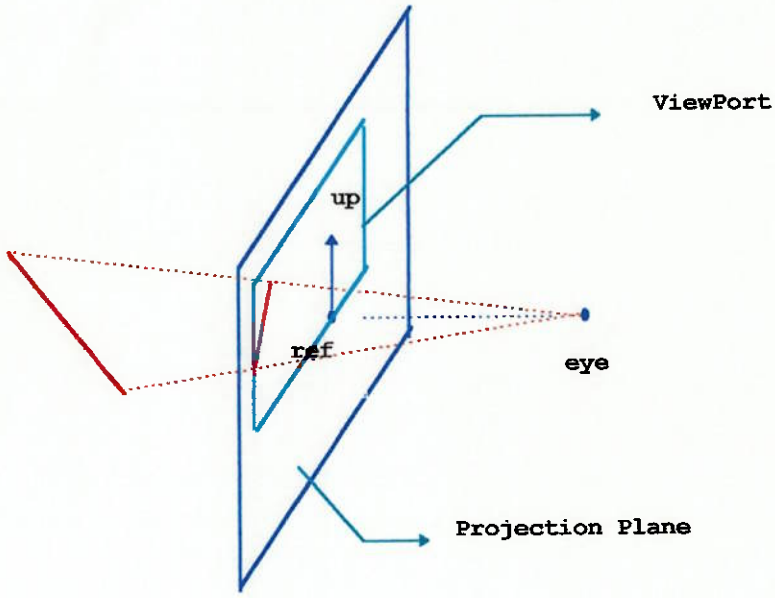
Finalmente, para atualizar-se as coordenadas de mundo da estrutura `3D`, percorre-se o container de `TLayer`'s da cena corrente (apontada pelo ponteiro global `pCurrentScene`), e para cada objeto `TLayer` percorre-se o container de `TShell`'s membro do mesmo. Cada objeto `TShell` possui um *flag* indicando se as coordenadas de mundo do mesmo precisam ser atualizadas ou não; caso necessário, chama-se uma rotina de atualização de coordenadas do objeto em questão.

Apesar de o processamento em segundo plano ainda não ter sido implementado, a estrutura para suportá-lo já está montada, como se pode observar, e, assim sendo, não se deve encontrar dificuldade em implementá-lo.

```
TTernaFloat eye, ref, up;  // (world coordinates)
```

Estes membros protegidos definem a posição da câmera associada a um objeto da presente classe (plano de projeção). *Eye* é um ponto do E^3 no qual se encontra o centro da câmera (i.e., centro da projeção perspectiva. *Ref* é um ponto de referência pertencente ao plano de projeção. O

vetor `(eye - ref)` é sempre normal ao plano de projeção. *Up* é um versor usado para determinar a orientação da câmera em torno do eixo `(eye,ref)`. Vide ilustração abaixo.



A câmera é definida pelo triedro `(eye, ref, up)`. O sistema de coordenadas associado a este triedro (com centro em `ref`, direção *y* dada por `up` e direção *z* dada por `(eye - ref)`, dextrógiro, denomina-se *eye coordinate system* associado ao plano de projeção, e as coordenadas a ele associadas denominam-se *eye coordinates*. A matriz de mudança de base do *eye coordinate system* para o sistema de coordenadas "*absoluto*" (coordenadas de mundo) é dada por uma matriz de transformação homogênea armazenada num membro público `TMarDirInv` de `TProjPlane` denominado `eyeMat`. `TMatDirInv` é uma classe usada para agrupar dois objetos `TMatTrans`, que são uma matriz de transformação homogênea e sua inversa. Para ganhar tempo de processamento, matrizes de transformação homogênea são armazenadas em conjunto com a sua inversa (caso exista inversa), para que não seja necessário inverter a matriz a todo momento que se precisar da sua inversa.

Além da matriz `eyeMat`, há também a matriz de transformação homogênea responsável pela transformação perspectiva. Tal matriz é armazenada no membro público `TMatTrans` de `TProjPlane` denomindado `projMat`. Note que `projMat` não foi declarado da classe `TMatDirInv` porque a

transformação perspectiva não é inversível, logo a matriz de transformação perspectiva não tem inversa. Multiplicando-se as *eye coordinates* de um ponto do E^3 pela matriz de transformação perspectiva `projMat`, obtêm-se as coordenadas da projeção perspectiva do referido ponto no objeto plano de projeção em questão. Tais coordenadas denominam-se *projection coordinates* ou coordenadas de plano de projeção do ponto.

```
virtual void delLinkageTo(TLinkage* pTo);
virtual void delLinkageTo(int i);
```

Estas funções anulam funções "delLinkageTo" da classe base `TLinkage`, e, juntamente com as demais funções herdadas de `TLinkage`, constituem o mecanismo de atualização automática de ligações entre objetos de classes derivadas de `TLinkage`, que será explicado posteriormente.

```
void setEye(TTernaFloat eyeIn);
void setRef(TTernaFloat refIn);
void setUp(TTernaFloat upIn);
void setTriedro(TTernaFloat eyeIn, TTernaFloat refIn, TTernaFloat upIn);
```

Estas funções modificam o triedro `(eye, ref, up)` e redefinem as matrizes `eyeMat` e `projMat` coerentemente; `setTriedro` sempre deve ser chamada a partir da função-membro pública `changePlane(...)`. Ou seja, a posição da câmera só deve ser alterada por esta função, mesmo que a posição do plano de projeção não mude, pois a referida função atualiza o sistema de *flags* para que posteriormente as coordenadas de plano de projeção ou de tela possam ser atualizadas.

As matrizes `eyeMat` e `projMat` são calculadas da seguinte maneira:

$$eyeMat.dir = T \cdot M$$
$$eyeMat.inv = M^t \cdot T^{-1}$$
$$projMat = TMatPerspective\left(\|eye - ref\|\right)$$

onde **M** é a matriz de mudança de base e **T** a matriz de translação do *eye coordinate system* para o sistema de coordenadas de mundo. Usando os índices *w* para coordenadas de mundo, *e* para *eye coordinates* e *p* para coordenadas de plano de projeção, temos as seguintes relações:

$$\vec{x}_e = \vec{x}_w \cdot eyeMat.dir$$
$$\vec{x}_p = \vec{x}_e \cdot projMat$$

Note que as coordenadas de plano de projeção obtidas desta maneira têm sempre a terceira coordenada nula.

```
void setbASolidModified(void);
```

Percorre o container de `TFlagSolid`'s e verifica se há algum *flag* setado, para então atualizar `bASolidModified`.

```
/*************************************************************************
     construtores
 *************************************************************************/
 TProjPlane();                          //construtor default

 TProjPlane(const TTernaFloat& eyeIn,
            const TTernaFloat& refIn,
            const TTernaFloat& upIn);


TProjPlane(const TTernaFloat& eyeIn,
           const TTernaFloat& refIn,
            const TTernaFloat& upIn,
             const TParFloat& llcEye,
              const TParFloat& urcEye,
               const TParInt& llcWin,
                const TParInt& urcWin);
```

Os últimos quatros parâmetros deste construtor são usados para construir o objeto-membro `planeWindowData`, da classe `TWindowData` (explicada mais adiante).

```
/*************************************************************************
        destrutor
 *************************************************************************/

virtual ~TProjPlane();
```

O destrutor deleta todos os objetos `TP2d` da estrutura `2D` associada. Note que, pelo fato de estar sendo utilizado o operador `delete` para destruir-se objetos `TP2d`, todos os objetos deste tipo deverão ser construídos no *heap* (i.e., com o operador `new`), ou no *stack* mas com a condição de que sejam destruídos antes de ser chamado o destrutor do plano de projeção associado. Esta observação vale não apenas para objetos `TP2d`, mas para todos os objetos da estrutura `2D` ou `3D` que forem deletados automaticamente de forma semelhante.

```
/*************************************************************************
        operadores, funções-membro e dados-membro públicos
 *************************************************************************/
```

```
TWindowData planeWindowData;
TConsPlane consPlane;
```

A cada objeto plano de projeção estão associados um objeto plano de costrução (`TConsPlane`) e um objeto `TWindowData`.

```
TMatDirInv eyeMat;
TMatTrans projMat;
```

Matrizes de transformação homogênea já mencionadas.

```
TTernaFloat getEye(void) const;
TTernaFloat getRef(void) const;
TTernaFloat getUp(void) const;

void setbProjCoordValid(bool bIn);
bool getbProjCoordValid(void) const;
void setbPlaneModified(bool bIn);
bool getbPlaneModified(void) const;
bool getbASolidModified(void) const;
void setbFlagSolid(TShell* pShell, bool bIn);
bool getbFlagSolid(TShell* pShell);
```

Funções para acessar os *flags* correspondentes (que são protegidos).

```
TFlagSolid addTFlagSolid(TFlagSolid newTFlagSolid);
TFlagSolid addTFlagSolidAt(TFlagSolid newTFlagSolid, int i);
int findTFlagSolid(TFlagSolid TFlagSolidIn) const;
TFlagSolid& getTFlagSolid(TShell*) const;
TFlagSolid& getTFlagSolid(int i) const;
TFlagSolid removeTFlagSolid(TFlagSolid TFlagSolidIn);
TFlagSolid removeTFlagSolid(int i);
itTFlagSolidContainer itvTFlagSolid;
int getItemsInvTFlagSolid(void);
```

Funções para manipular o container de `TFlagSolid`'s .

```
TTernaFloat world2eye(TTernaFloat ternaIn);
TTernaFloat eye2world(TTernaFloat ternaIn);
TTernaFloat eye2proj(TTernaFloat ternaIn);
TTernaFloat proj2eye(TTernaFloat ternaIn);
TTernaFloat world2proj(TTernaFloat ternaIn);
TTernaFloat proj2world(TTernaFloat ternaIn);
TParInt world2view(TTernaFloat ternaIn);
TParInt eye2view(TTernaFloat ternaIn);
TParInt proj2view(TTernaFloat ternaIn);
```

Funções de conversão de coordenadas. Os nomes são mnemônicos:

world = world coordinates;
eye = eye coordinates;
proj = projection coordinates (coordenadas de plano de projeção);
view = view coordinates (coordenadas de tela).

Note que as coordenadas de plano de projeção são armazenadas numa terna ordenada, ao invés de num par ordenado, como seria razoável de se esperar. A razão disto é que a terceira coordenada (z) das *eye coordinates* é mantida, para que se possam recuperar as *eye coordinates* do ponto a partir das coordenadas de plano de projeção. Ou seja, isto é uma maneira de se transformar a

perspectiva numa aplicação bijetiva (inversível), o que pode ser desejável em alguma situação. Assim, após a mutiplicação

$$\vec{x}_p = \vec{x}_e \cdot projMat$$

atribui-se à cordenada z das coordenadas de plano de projeção a coordenada z das *eye coordinates*. Evidentemente, apenas as duas primeiras coordenadas da terna são, efetivamente, coordenadas de plano de projeção, e apenas elas serão usadas no mapeamento das coordenadas de plano de projeção para coordenadas de tela.

```
void changePlane(TTernaFloat& eyeIn,
                 TTernaFloat& refIn,
                  TTernaFloat& upIn,
                   const TParFloat& llcEye=zeroTernaFloat,
                    const TParFloat& urcEye=zeroTernaFloat,
                     const TParInt& llcWin=zeroTernaInt,
                      const TParInt& urcWin=zeroTernaInt);
```

Redefine a posição do plano de projeção. Caso os quatro últimos parâmetros assumam valores *default*, a posição da *viewport* (i.e., `planeWindowData`) é mantida inalterada, senão a função-membro pública `changeWindow(...)` de `TWindowData` é chamada com estes parâmetros.

```
void changeConsPlane(TTernaFloat p, TTernaFloat n);
```

Redefine a posição do plano de construção associado.

```
void refreshProjCoordinates(void);
```

Atualiza coordenadas de plano de projeção da estrutura `2D` associada.

```
TP2d* addTP2d(TP2d* pnewTP2d);
TP2d* addTP2dAt(TP2d* pnewTP2d, int i);
int findTP2d(TP2d* pTP2d) const;
TP2d* getTP2d(int i) const;
TP2d* removeTP2d(TP2d* pTP2d);
TP2d* removeTP2d(int i);
bool delTP2d(TP2d* pTP2d);
bool delTP2d(int i);
itTP2dContainer itviTP2d;
int getItemsInviTP2d(void) const;
```

Funções para manipular o container de `TP2d`'s.

Classe TWindowData

Cada objeto `TProjPlane` possui um membro público denominado `planeWindowData`. Este membro é da classe `TWindowData`, que tem as seguintes funções principais:

- mapeamento das coordenadas de plano de projeção para as coordenadas de tela;

- manutenção das coordenadas de tela da estrutura `2D` associada ao objeto plano de projeção ao qual um objeto da presente classe (`TWindowData`) pertence;

- comunicação com a interface *Windows* da *OWL* . Provisoriamente, tal comunicação é feita através de um ponteiro para a janela cliente da janela principal do aplicativo. Posteriormente, ao se mudar a interface *Windows* do aplicativo para um modelo *doc/view*, esta comunicação deverá ser feita com um objeto `TView`, não mais com a janela cliente da janela principal. Desta forma, cada objeto `TWindowData` (e, conseqüentemente, cada objeto plano de projeção), estará associado a uma vista de um documento do aplicativo (ou do documento, caso seja mantida a interface *SDI*, o que parece razoável, para limitar o consumo de memória RAM durante a execução do programa). Cada documento deverá, por sua vez, estar associado a um objeto `Tscene`.

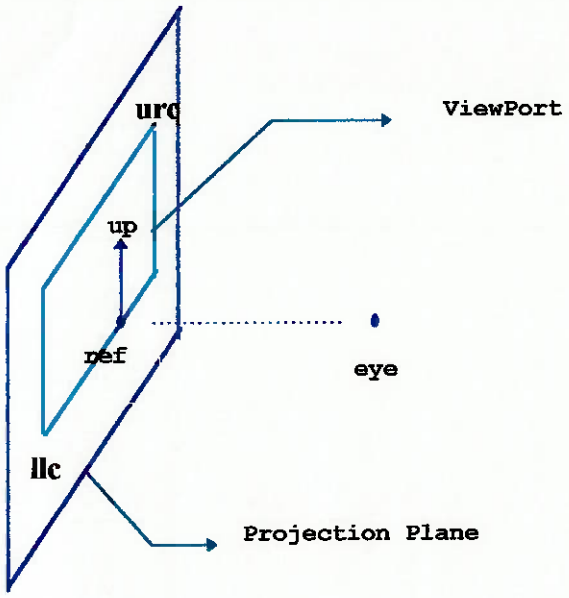
A classe `TWindowData` possui os seguintes membros principais:

```
/************************************************************************
        dados-membro protegidos
 ************************************************************************/
bool bViewCoordValid;
bool bWindowModified;
bool bPlaneModified;
bool bASolidModified;
TFlagSolidContainer vTFlagSolid;
```

Sistema de *flags* semelhante ao da classe `TProjPlane`, usado para atualização das coordenadas de tela, conforme já discutido.

```
TParFloat llcProj, urcProj;
TParInt llcWin, urcWin;
/* view e proj coordinates  do upper right corner e do lower left corner da janela */
```

Através das *projection coordinates* e correspondentes *view coordinates* dos pontos **llc**(*lower left corner*) e **urc**(*upper right corner*) que definem a *viewport*, determina-se o mapeamento das coordenadas de plano de plano de projeção para coordenadas de tela.



O referido mapeamento é feito através da função de conversão de coordenadas `proj2view(...)`, e o mapeamento inverso através de `view2proj(...)`. Estas funções utilizam a matriz de transformação homogênea armazenada no membro público `viewMat`, descrito posteriormente.

```
TProjPlane* pTProjPlane;
```

Ponteiro para permitir comunicação com o objeto `TProjPlane` pai.

```
interWindow* pVisWindow;
```

Ponteiro para permitir comunicação com a janela cliente da janela principal do aplicativo (vide discussão acima).

```
TEdgeWinContainer viTEdgeWin;
TFaceWinContainer viTFaceWin;
```

Containers de ponteiros para todos os objetos `TEdgeWin` e `TFaceWin` da estrutura `2D` associada. Cada objeto `TEdgeWin` é a projeção de um objeto `TWedge` (estrutura `3D`) no plano de projeção pai; analogamente, cada objeto `TFaceWin` é a projeção de um objeto `TBezierFace`. Note que, como a estrutura para suportar faces não-planas ainda não está completamente desenvolvida,

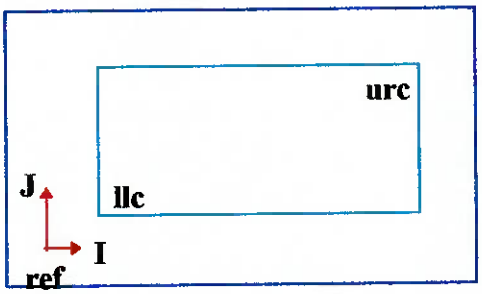
objetos da classe `TBezierFace` não são criados, por enquanto. Conseqüentemente, objetos da classe `TFaceWin` também não são. Não é necessário manter um objeto na estrutura `2D` que represente a projeção de uma face plana. Sendo o propósito de um tal objeto permitir que curvas isométricas da face sejam plotadas numa representação icônica da mesma, dispensa-se isto para faces planas, porque a representação icônica destas consiste apenas nas arestas que limitam o seu contorno (i.e., *loops* internos e externo).

```
virtual void delLinkageTo(TLinkage* pTo);
virtual void delLinkageTo(int i);
/* anulam funções "delLinkageTo" da classe base */
```

Estas funções anulam funções "delLinkageTo" da classe base `TLinkage`, e, juntamente com as demais funções herdadas de `TLinkage`, constituem o mecanismo de atualização automática de ligações entre objetos de classes derivadas de `TLinkage`, que será explicado posteriormente.

```
void setCorners(TParFloat urcProjIn, TParFloat llcProjIn, TParInt urcWinIn, TParInt
llcWinIn);
```

Estas funções alteram as coordenadas de tela e de plano de projeção dos pontos `llc` e `urc` que definem a *viewport* (vide discussão acima). Também redefine, coerentemente, o mapeamento entre coordenadas de tela e de plano de projeção, dado pela matriz de transformação homogênea armazenada no membro público `viewMat`. Esta matriz é calculada da seguinte maneira:



$$s_x = \frac{\langle urc_v, \hat{i} \rangle - \langle llc_v, \hat{i} \rangle}{\langle urc_p, \hat{i} \rangle - \langle llc_p, \hat{i} \rangle}$$

$$s_y = \frac{\langle urc_v, \hat{j} \rangle - \langle llc_v, \hat{j} \rangle}{\langle urc_p, \hat{j} \rangle - \langle llc_p, \hat{j} \rangle}$$

$$\vec{t} = \vec{llc_v} - \left(s_x \cdot \langle llc_v, \hat{i} \rangle, s_y \cdot \langle llc_v, \hat{j} \rangle \right)$$

$$viewMat.dir = TMatScaling(s_x, s_y, 0) \cdot TMatTranslation(t_x, t_y, 0)$$

onde o índice *p* foi usado para coordenadas de plano de projeção (*projection coordinates*) e o índice *v* foi usado para coordenadas de tela (*view coordinates*).

```
/*************************************************************************
        construtores
 *************************************************************************/
TWindowData(TProjPlane* pPlane=NULL); //construtor default

TWindowData(const TParFloat& llcProj,const TParFloat& urcProj,
            const TParInt& llcWin,const TParInt& urcWin,
               TProjPlane* pPlane=NULL);


/*************************************************************************
       destrutor
 *************************************************************************/
virtual ~TWindowData();
```

O destrutor deleta todos os objetos `TEdgeWin` e `TFaceWin` da estrutura `2D` associada, seguindo a mesma filosofia do destrutor da classe `TProjPlane`.

```
/*************************************************************************
       operadores, dados e funções-membro públicas
 *************************************************************************/
 TMatDirInv viewMat, viewCorrectorMat;
```

O objeto `viewMat` armazena a matriz de transformação homogênea através da qual as coordenadas de plano de projeção são mapeadas em coordenas de tela, como já discutido. O membro `viewCorrectorMat` armazena uma matriz de transformação homogênea usada para atualização rápida das coordenadas de tela da estrutura `2D` associada, quando as mesmas deixam de ser válida por causa de um simples deslocamento da *viewport* sobre o plano de projeção (o que ocorre numa operação *zoom* ou *pan*, por exemplo). Vide código de atualização de coordenadas de tela - listagem da função `refreshViewCoordinates(...)` - para maiores detalhes.

```
D2dCursor cursor;
```

Este membro é um objeto da classe `D2dCursor`, descrita mais adiante, e é usado para descrever um cursor bidimensional associado à *viewport*, usado para seleção.

```
void setbViewCoordValid(bool bIn);
bool getbViewCoordValid(void) const;
void setbWindowModified(bool bIn);
bool getbWindowModified(void) const;
void setbPlaneModified(bool bIn);
bool getbPlaneModified(void) const;
bool getbASolidModified(void) const;
void setbFlagSolid(TShell* pShell, bool bIn);
```

```
bool getbFlagSolid(TShell* pShell) const;

TFlagSolid addTFlagSolid(TFlagSolid newTFlagSolid);
TFlagSolid addTFlagSolidAt(TFlagSolid newTFlagSolid, int i);
int findTFlagSolid(TFlagSolid TFlagSolidIn) const;
TFlagSolid& getTFlagSolid(TShell* pShell) const;
TFlagSolid& getTFlagSolid(int i) const;
TFlagSolid removeTFlagSolid(TFlagSolid TFlagSolidIn);
TFlagSolid removeTFlagSolid(int i);
itTFlagSolidContainer itvTFlagSolid;
int getItemsInvTFlagSolid(void);
/* funções para manipular o container de TFlagSolids */

TParFloat getUrcProj(void) const;
TParFloat getLlcProj(void) const;
TParInt getUrcWin(void) const;
TParInt getLlcWin(void) const;

TParInt proj2view (const TParFloat& Pin);
TParFloat view2proj (const TParInt& Pin);
```

Funções de conversão entre *projection* e *view coordinates*, já discutidas.

```
TParInt viewOld2viewNew(const TParInt& Pin);
TParInt viewNew2viewOld(const TParInt& Pin);
/* funções de correção de view coordinates */
```

Estas funções são usadas no mecanismo de atualização rápida de coordenadas de tela, já mencionado.

```
void refreshViewCoordinates(void);
```

Funções de atualização das coordenadas de tela da estrutura 2D associada.

```
void changeWindow(const TParFloat& llcProj,const TParFloat& urcProj,
                  const TParInt& llcWin,const TParInt& urcWin);
```

Esta função redefine a posição da *viewport* sobre o plano de projeção, e altera o mapeamento de coordenadas de plano de projeção para coordenadas de tela coerentemente.

```
void setpTProjPlane (TProjPlane* pIn);
TProjPlane* getpTProjPlane(void) const;

void setpVisWindow (interWindow* pIn);
interWindow* getpVisWindow(void) const;

TEdgeWin* addTEdgeWin(TEdgeWin* pnewTEdgeWin);
TEdgeWin* addTEdgeWinAt(TEdgeWin* pnewTEdgeWin, int i);
int findTEdgeWin(TEdgeWin* pTEdgeWin) const;
TEdgeWin* getTEdgeWin(int i) const;
TEdgeWin* removeTEdgeWin(TEdgeWin* pTEdgeWin);
TEdgeWin* removeTEdgeWin(int i);
itTEdgeWinContainer itviTEdgeWin;
int getItemsInviTEdgeWin(void);
/* funções para manipular o container de TEdgeWin's */

TFaceWin* addTFaceWin(TFaceWin* pnewTFaceWin);
TFaceWin* addTFaceWinAt(TFaceWin* pnewTFaceWin, int i);
int findTFaceWin(TFaceWin* pTFaceWin) const;
TFaceWin* getTFaceWin(int i) const;
TFaceWin* removeTFaceWin(TFaceWin* pTFaceWin);
TFaceWin* removeTFaceWin(int i);
```

```
itTFaceWinContainer itviTFaceWin;
int getItemsInviTFaceWin(void);
/* funções para manipular o container de TFaceWin's */
```

7.5 Classe **TIinkage** ("linkage.h")

A classe `TLinkage` é a classe base de todas as classes das estruturas `2D` e `3D`, das classes `TPlane` e `TWindowData`. Esta classe incorpora funcionalidade aos objetos das classes derivadas para manutenção automática das ligações (ponteiros) entre os mesmos.

Por exemplo, suponha que tenhamos um objeto `TWedge`, com vários outros tipos de objetos apontando para o mesmo. Então, se este objeto for destruído, todos os ponteiros de todos os objetos de classes derivadas de `TLinkage` que apontam para ele serão atualizados (ou seja, aterrados) automaticamente. Além disso, se algum destes objetos possuir um container de ponteiros para objetos `TWedge` (como é o caso dos objetos das classes `TShell`, `TLoop`, e `TP3d`), o ponteiro do objeto `TWedge` a ser deletado será retirado automaticamente destes containers.

Sem este mecanismo automático de atualização de ligações, seria preciso atualizá-las "manualmente", um procedimento sujeito a erros e que envolveria muitas linhas de código.

O inconveniente deste processo de atualização automática de ligações é o consumo de memória, pois cada ponteiro para um objeto de classe derivada de `TLinkage` que participa do mecanismo (há a possibilidade de se excluírem ponteiros do mecanismo, como será visto adiante) acaba tendo que ser armazenado três vezes.

Resumidamente, o mecanismo funciona da seguinte maneira:

Cada objeto de classe derivada de `TLinkage` herda dois containers da mesma: num container, denominado `viTLinkageTo`, são armazenados todos os ponteiros para objetos `TLinkage` para os quais o objeto em questão aponta; no outro container, denominado `viTLinkageFrom`, são armazenados ponteiros para todos os objetos que apontam para o objeto em questão. A atualização automática de ligações é feita quando algum objeto de classe derivada de `TLinkage` está prestes a ser destruído: sendo chamado o destrutor da classe de base para destruir o subobjeto `TLinkage` herdado,

o mesmo faz chamada a um método denominado `delAllLinkages()`, que percorre cada um dos containers mencionados e desfaz as ligações. Para que isto funcione, há dois requisitos:

1. Para que a ligação com algum objeto de classe derivada de `TLinkage` seja incluída no mecanismo de atualização automática, a mesma deve ser feita usando-se a função `newLinkageTo(...)`, função que atualiza o container `viTLinkageTo` do objeto que está apontando e o container `viTLinkageFrom` do objeto que está sendo apontado. Note que a referida função apenas atualiza os containers; a ligação, propriamente dita, deve ser feita normalmente, utilizando-se o operador de atribuição, por exemplo. Como `newLinkageTo(...)` é função protegida, ela deve ser chamada de alguma outra função-membro da classe, ao ser feita a ligação. Isto foi feito propositalmente, com o intuito de que sempre se use uma função de nível mais alto que encapsule as chamadas a `newLinkageTo(...)` para fazer as ligações.

2. Além de chamar a função `newLinkageTo(...)` ao ser feita a ligação, o mecanismo de atualização automático de ligações também exige que seja anulada nas classes derivadas a função virtual `delLinkageTo(...)`. Esta função é chamada pela função `delAllLinkages()`, herdada da classe base. `DelAllLinkages()`, por sua vez, é chamada no destrutor da classe base, conforme já mencionado. A função `delLinkageTo(...)` a ser anulada recebe como parâmetro um `TLinkage*` (ponteiro para `Tlinkage`); este é um ponteiro para o objeto apontado com o qual deverá ser cortada a ligação. A função anulada deverá comparar este ponteiro com cada um dos ponteiros da classe derivada que participam do mecanismo e tomar as medidas cabíveis em cada caso (se o ponteiro for um dado-membro, deverá aterrá-lo; se estiver num container, deverá removê-lo do container, etc.). Vide código fonte da classe `TLinkage` e de alguma classe derivada para uma maior elucidação.

7.6 Classes da estrutura 2D

Classes *TEdgeWin* e *TFaceWin* ("edfawin.h")

Objetos das classes `TEdgeWin` e `TFaceWin` são projeções de objetos das classes `TWedge` e `TBezierFace`, respectivamente. Como já mencionado, ainda não foi desenvolvido completamente o suporte a faces não-planas, por isto objetos da classe `TBezierFace` não são criados, por enquanto, nem da classe `TFaceWin`. As classes `TEdgeWin` e `TFaceWin` são derivadas de uma classe abstrata denominada `TEdgeFaceWin`, que agrupa os elementos comuns às duas primeiras.

Objetos destas classes são sempre criados automaticamente através de uma operação de projeção dos objetos da estrutura 3D. Esta operação de projeção pode ocorrer em duas situações:

- quando o objeto da estrutura 3D é construído; neste caso, o próprio construtor deste objeto faz chamada a uma função membro que o projeta em cada um dos planos de projeção, construindo automaticamente os objetos das estruturas 2D correspondentes;
- quando um novo plano de projeção é criado; neste caso, o construtor do objeto `TProjPlane` faz chamada a uma função membro que "comanda" a cena corrente a se projetar no referido plano. A cena corrente, então, "rola" o comando hierarquia abaixo, ordenando para que cada `TLayer` se projete, que por sua vez ordena que cada `TShell` associado se projete, que por sua vez ordena que cada `TFace`, `TWedge` e `TP3d` componentes se projetem no novo plano de projeção criado.

Os principais membros destas classes são:

Classe TEdgeFaceWin

```
/*************************************************************************
dados-membro protegidos
 *************************************************************************/

TWindowData* pTWindowData;
```

Ponteiro para objeto da classe `TWindowData` do plano de projeção ao qual está associada a estrutura 2D a que pertencerá algum objeto desta classe.

```
TEdgeIsoCurveContainer viTEdgeIsoCurve;
```

Vetor de ponteiros para objetos de classes derivadas da classe abstrata `TEdgeIsoCurve`. Tal classe é a base das classes `TEdgeCurve` e `TIsoCurve` (explicadas mais adiante).

```
virtual void delLinkageTo(TLinkage* pTo);
virtual void delLinkageTo(int i);
/* anulam funções correspondentes da classe base */

int selected;
```

Este inteiro é usado pelo mecanismo de seleção. Cada vez que um objeto `TEdgeWin` é selecionado, este membro é incrementado; se for desselecionado, decrementa-se o referido membro. Na verdade, este membro não deveria estar aqui, e sim nas classes `TWedge` e `TFace`; os objetos `TEdgeWin` e `TFaceWin` são usados pelo mecanismo de seleção, mas os objetos a serem selecionados sempre estão, em última instância, na estrutura `3D`. Colocar este campo aqui foi um erro de projeto, é preciso corrigi-lo.

```
/*************************************************************************
construtores
 *************************************************************************/

TEdgeFaceWin(TWindowData* pTWindowData=NULL);      // construtor default
/*************************************************************************
destrutor
 *************************************************************************/
virtual ~TEdgeFaceWin();
```

Destrói todos os objetos apontados no container `viTEdgeIsoCurve`.

```
/*************************************************************************
operadores, funções-membro públicas
 *************************************************************************/

void setpTWindowData(TWindowData* );
TWindowData* getpTWindowData(void) const;
/* funções para acessar pTWindowData */
TEdgeIsoCurve* addTEdgeIsoCurve(TEdgeIsoCurve* );
TEdgeIsoCurve* addTEdgeIsoCurveAt(TEdgeIsoCurve*, int i);
int findTEdgeIsoCurve(TEdgeIsoCurve*)  const;
TEdgeIsoCurve* getTEdgeIsoCurve(int i) const;
TEdgeIsoCurve* removeTEdgeIsoCurve(TEdgeIsoCurve*);
TEdgeIsoCurve* removeTEdgeIsoCurve(int i);
itTEdgeIsoCurveContainer itviTEdgeIsoCurve;
int getItemsInviTEdgeIsoCurve(void);
void delAllTEdgeIsoCurves(void);
/* funções para manipular o container de TEdgeIsoCurve */

virtual void refreshViewCoordinates(void)=0;
void setVisibility(void);
virtual void draw(TDC& dc) const =0;
virtual void showSelection(TDC&) =0;
virtual void hideSelection(TDC&) =0;
virtual void drawSelection(TDC&) =0;
```

Classe TEdgeWin

Além dos membros herdados de `TEdgeFaceWin`, a classe `TEdgeWin` tem um ponteiro para um objeto `TWedge` (estrutura `3D`) e anula os seguintes métodos virtuais abstratos da classe base:

- `refreshViewCoordinates()`: função usada para atualizar as coordenadas de tela do objeto `TEdgeWin`;
- `draw()`: desenha objetos desta classe;
- `showSelection(...)` e `hideSelection(...)`: seleciona ou desseleciona objetos desta classe.

Classe TFaceWin

Como ainda não foi desenvolvido completamente o suporte a faces não-planas, objetos desta classe não são criados. Assim sendo, não se entrará em maiores detalhes sobre a mesma, além do que já foi dito; consulte a listagem para maior elucidação.

Classes ***TEdgeCurve*** *e* ***TIsoCurve*** *("edfacurv.h")*

As classes `TEdgeCurve` e `TIsoCurve` são derivadas da classe abstrata `TEdgeIsoCurve`. Cada objeto da classe `TEdgeWin` possui um container de ponteiros para objetos da classe `TEdgeCurve`. Cada objeto `TEdgeCurve` é um trecho conexo da projeção do objeto `TWedge` na estrutura `2D`. Analogamente, cada objeto da classe `TFaceWin` possui um container de objetos da classe `TIsoWin`; cada objeto `TIsoWin` é a projeção de uma curva isométrica de uma face (`TBezierFace`) na estrutura `2D`. Objetos desta classe ainda não são criados, por motivos já expostos, por isto não se entrará em maiores detalhes sobre a mesma (vide listagem).

Os principais membros de `TEdgeCurve` são:

```
/*************************************************************************
dados-membro protegidos
 *************************************************************************/

Visibility vis;
```

Membro que indica se o objeto é visível ou não, em função do *layer* a que ele pertence.

```
DPointContainer vDPoint;
```

Container de objetos da classe `Dpoint`. Esta é uma classe derivada da classe `Tpoint` da *OWL*. No caso de `TWedge`'s que sejam segmentos de reta, este container contém as projeções (em coordenadas de tela) de cada extremidade do segmento. Futuramente, para suportar `TWedge`'s com outras geometrias, este container poderá ser usado para armazenar pontos discretos da curva a ser plotada na *viewport*, ou pontos de controle da mesma, por exemplo.

```
TEdgeWin* pTEdgeWin;
```

Ponteiro para o objeto da classe `TEdgeWin` correspondente.

```
virtual void delLinkageTo(TLinkage* pTo);
virtual void delLinkageTo(int i);
/* anulam funções correspondentes da classe base */


/*************************************************************************
operadores, funções-membro públicas
 *************************************************************************/


virtual void setVisibility(void);
```

Determina se o objeto é visível ou não em função da visibilidade do *layer* em que ele se encontra.

```
virtual void draw(TDC& dc) const;
```

Desenha o objeto através dispositivo de contexto passado como parâmetro.

Classe **TP2d** *("p2d.h")*

Instâncias desta classe representam a projeção de um objeto da classe `TP3d`. Como no caso dos objetos das classes `TEdgeWin` e `TFaceWin`, objetos desta classe devem ser sempre criados automaticamente por uma operação de projeção do objeto da estrutura `3D` associado (i.e., `TP3d`). Quando um `TP3d` é criado, o construtor do mesmo chama uma função de projeção que o projeta em todos os planos de projeção, crianndo os `TP2d`'s correspondentes. Analogamente, quando um novo plano de projeção é criado, o construtor do mesmo ordena que a cena corrente se projete sobre ele (vide discussão acima), e todos os `TP3d`'s da cena se projetam sobre o referido plano, criando os `TP2d`'s correspondentes.

Os principais membros desta classe são:

```
/*************************************************************************
dados-membro protegidos
*************************************************************************/

TTernaFloat projCoord;
```

Coordenadas de plano de projeção.

```
DPoint viewCoord;
```

Coordenadas de tela.

```
TP3d* pTP3d;              // ponteiro para o TP3d correspondente
TProjPlane* pTProjPlane; // ponteiro para o plano de projeção correspondente

virtual void delLinkageTo(TLinkage* pTo);
virtual void delLinkageTo(int i);
/* anula funções correspondentes da classe base */



/*************************************************************************
construtores
 *************************************************************************/
TP2d();                          //construtor default
TP2d(TP3d& TP3dpai, TProjPlane& TProjPlanein);
```

Constrói `TP2d` a partir de um `TP3d` e de um objeto `TProjPlane`.

```
/*************************************************************************
operadores, funções-membro públicas
 *************************************************************************/


int toggled;
```

Possui função análoga ao membro `selected` da classe `TEdgeFaceWin`, discutida anteriormente. Cada vez que o ponto é selecionado, incrementa-se este inteiro; cada vez que o ponto é desselecionado, `toggled` é decrementado. Como no caso da classe `TEdgeFaceWin`, este membro também deveria ter sido colocado na estrutura `3D` (i.e., na classe `TP3d`); deve-se corrigir este erro de projeto futuramente (vide discussão sobre isto na classe `TEdgeFaceWin`).

```
void setxProj(float xProj);
void setyProj(float yProj);
void setzProj(float zProj);
void setProjCoord(TTernaFloat pin);
float getxProj(void) const;
float getyProj(void) const;
float getzProj(void) const;
TTernaFloat getProjCoord(void) const;
/* funções para acessar as "projection coordinates" do ponto */
void setxView(int xView);
void setyView(int yView);
void setViewCoord(TParInt pin);
int getxView(void) const;
int getyView(void) const;
TParInt getViewCoord(void) const;
/* funções para acessar as "view coordinates" do ponto */
void setpTP3d(TP3d* pTP3d);
TP3d* getpTP3d(void) const;
/* funções para acessar pTP3d */
void setpTProjPlane(TProjPlane* pTProjPlane);
TProjPlane* getpTProjPlane(void) const;
/* funções para acessar  pTProjPlane */
virtual bool operator==(const TP2d&);
```

Este operador foi sobrecarregado para testar a igualdade entre objetos da classe `TP2d` por meio de comparação entre as suas coordenadas (vide código fonte).

```
operator DPoint()
{return DPoint(getViewCoord());};
```

Operador de conversão que constrói um objeto `DPoint` a partir das coordenadas de tela do objeto `TP2d`.

```
void refreshProjCoordinates(void);
void refreshViewCoordinates(void);
void refreshCoordinates(void);
```

Funções para atualizar coordenadas de plano de projeção e de tela do `TP2d`.

```
void toggleSelection(bool);
```

Caso seja passado *true* como parâmetro, o ponto é selecionado; caso contrário, desselecionado.

```
void showMark(TDC& dc);
```

Mostra uma marca em torno do ponto, caso o mesmo esteja selecionado.

```
void draw(TDC& dc);
```

Usada pela função `Paint(...)` da classe `TWindow` da *OWL* para redesenhar o ponto (se estiver selecionado, desenha uma marca, caso contrário não desenha nada).

7.7 Classes da estrutura 3D

Classe TP3d ("p3d.h")

Objetos desta classe representam pontos do E^3. Seus principais membros são:

```
/*************************************************************************
dados-membro protegidos
 *************************************************************************/

TTernaFloat worldCoord;
```

Coordenadas de mundo do ponto.

```
TP2dContainer viTP2d;
```

Container de ponteiros para todos os objetos `TP2d`'s correspondentes.

```
TShell* pTShell; // ponteiro para o TShell que contém o TP3d
virtual void delLinkageTo(TLinkage* pTo);
virtual void delLinkageTo(int i);
/* anulam funções "dellinkageto" da classe base */



/*************************************************************************
construtores
 *************************************************************************/
TP3d(TShell* pIn=NULL, PointType ptype=nothing);      // contrutor default
TP3d(float x, float y, float z,TShell* pIn=NULL, PointType ptype=nothing);
/* constrói um TP3d com coordenadas x,y,z. O construtor também deverá
```

```
chamar o construtor da classe TP2d, definindo tantas instâncias da referida
classe quantos forem os planos de projeção e estabelecer as ligações com
cada um dos TP2d construídos */
TP3d(TTernaFloat, TShell* pTShell=NULL, PointType ptype=nothing);

/*************************************************************************
destrutor
 *************************************************************************/
virtual ~TP3d();
```

Destrói todos os TP2d's correspondentes.

```
/*************************************************************************
operadores, funções-membro públicas
 *************************************************************************/

void setpTShell(TShell* pTShell);
TShell* getpTShell() const;
/* funções para acessar pTShell */
TP2d* addTP2d(TP2d* pnewTP2d);
TP2d* addTP2dAt(TP2d* pnewTP2d, int i);
int findTP2d(TP2d* pTP2d) const;
TP2d* getTP2d(TProjPlane*) const;
TP2d* getTP2d(int i) const;
TP2d* removeTP2d(TP2d* pTP2d);
TP2d* removeTP2d(int i);
itTP2dContainer itviTP2d;
int getItemsInviTP2d(void);
/* funções para manipular o container de TP2d */

void setxWorld(float x);
void setyWorld(float y);
void setzWorld(float z);
void setWorldCoord(TTernaFloat);
float getxWorld(void) const;
float getyWorld(void) const;
float getzWorld(void) const;
TTernaFloat getWorldCoord(void) const;
```

Funções para acessar as coordenadas de mundo do ponto.

```
operator TTernaFloat()
{return getWorldCoord();};
```

Operador de conversão que constrói uma terna ordenada a partir das coordenadas de mundo do objeto TP3d.

```
float getxEye(TProjPlane*) const;
float getyEye(TProjPlane*) const;
float getzEye(TProjPlane*) const;
TTernaFloat getEyeCoord(TProjPlane*) const;
float getxProj(TProjPlane*) const;
float getyProj(TProjPlane*) const;
float getzProj(TProjPlane*) const;
TTernaFloat getProjCoord(TProjPlane*) const;
int getxView(TProjPlane*) const;
int getyView(TProjPlane*) const;
TParInt getViewCoord(TProjPlane*) const;
```

Funções para acessar *eye*, *projection* e *view coordinates* do ponto em relação ao plano de projeção passado como parâmetro.

```
void projP3d(void);
```

Projeta o objeto TP3d em todos os planos de projeção.

```
void projP3d(TProjPlane*);
```

Projeta o objeto `TP3d` no plano de projeção passado como parâmetro.

```
void refreshWorldCoordinates(void);
void refreshProjCoordinates(void);
void refreshProjCoordinates(TProjPlane*);
void refreshViewCoordinates(void);
void refreshViewCoordinates(TProjPlane*);
```

Atualizam coordenadas do objeto `TP3d`.

Classe **TVertex3d** *("p3d.h")*

A classe `TVertex3d` é derivada de `TP3d` e incorpora funcionalidade extra para representar vértices de objetos `TShell`.

```
/*************************************************************************
dados-membro privados
 *************************************************************************/

TWedgeContainer viTWedge;
```

Container de ponteiros para todas as arestas (`TWedge`'s) ligadas ao vértice.

```
virtual void delLinkageTo(TLinkage* pTo);
virtual void delLinkageTo(int i);
/* anulam funções correspondentes da classe base */
/*************************************************************************
operadores, funções-membro públicas
 *************************************************************************/

TWedge* addTWedge(TWedge* pnewTWedge);
TWedge* addTWedgeat(TWedge* pnewTWedge, int i);
int findTWedge(TWedge* pTWedge) const;
TWedge* getTWedge(int i) const;
TWedge* removeTWedge(TWedge* pTWedge);
TWedge* removeTWedge(int i);
itTWedgeContainer itviTWedge;
int getItemsInviTWedge(void) const;
bool hasTWedgeMember(TWedge* pTWedge) const;
/* funções para manipular o container de TWedge */
```

Classe **TWedge** *("wedge.h")*

A classe `TWedge` é usada para implementar a estrutura "*winged-edge*", já discutida. Seus principais membros são:

```
/*************************************************************************
dados-membro protegidos
 *************************************************************************/

TP3dContainer viTP3d;                    //viTP3d[0]=Vstart; viTP3d[last]=Vend
```

Container de ponteiros para os `TP3d`'s que definem o *edge* (*Vstart*, *Vend* e, possivelmente, pontos de controle intermediários, que podem ser usados para descrever arestas que não sejam segmentos de reta - basta derivar classes desta com funcionalidade extra).

```
TWedge* ppcw;
TWedge* pncw;
TWedge* ppccw;
TWedge* pnccw;
TFace* pfcw;
```

```
TFace* pfccw;
```

Vide explicação sobre a estrutura "*winged-edge*" (item 5.1.1); maiores detalhes na referência [11].

```
TShell* pTShell; //ponteiro para o TShell ao qual pertence o TWedge
TEdgeWinContainer viTEdgeWin;
```

Container de ponteiros para todas as projeções do objeto `TWedge` (i.e., `TEdgeWin`'s).

```
virtual void delLinkageTo(TLinkage* pTo);
virtual void delLinkageTo(int i);
/* anulam funções correspondentes da classe base */


/*************************************************************************
construtores
 *************************************************************************/

TWedge(TShell* pTShell=NULL, EdgeType etype=straight); // construtor default
TWedge(TVertex3d* , TVertex3d* , TShell* pTShell=NULL, EdgeType etype=straight);
/* constrói TWedge com vértices apontados por pvstart, pvend e TShell dado
por pTShell, estabelecendo as ligações com os referidos objetos; deverá cha-
mar o construtor da classe TEdgeWin e estabelecer a ligação com o objeto
correspondente. Define EdgeType  = straight */
TWedge(TVertex3d*[2] , TShell* pTShell=NULL, EdgeType etype=straight);
/* análogo ao construtor anterior; pvertex[1]=vstart e pvertex[2]=pvend */


/*************************************************************************
destrutor
 *************************************************************************/
virtual ~Twedge();
```

Destrói todos os `TEdgeWin`'s associados.

```
/*************************************************************************
operadores, funções-membro públicas
 *************************************************************************/
int countP3d(TP3d*);
```

Retorna o número de ponteiros de um objeto desta classe para o objeto `TP3d` passado como parâmetro.

```
int countWedge(TWedge*);
```

Retorna o número de ponteiros de um objeto desta classe para o objeto `TWedge` passado como parâmetro.

```
int countFace(TFace*);
```

Retorna o número de ponteiros de um objeto desta classe para o objeto `TFace` passado como parâmetro.

```
TEdgeWin* addTEdgeWin(TEdgeWin* pneTWTEdgeWin);
TEdgeWin* addTEdgeWinAt(TEdgeWin* pneTWTEdgeWin, int i);
int findTEdgeWin(TEdgeWin* pTEdgeWin) const;
TEdgeWin* getTEdgeWin(TProjPlane* pIn) const;
TEdgeWin* getTEdgeWin(int i) const;
TEdgeWin* removeTEdgeWin(TEdgeWin* pTEdgeWin);
TEdgeWin* removeTEdgeWin(int i);
```

```
itTEdgeWinContainer itviTEdgeWin;
int getItemsInviTEdgeWin(void) const;
/* funções para manipular o container de TEdgeWin */

void setVStart(TVertex3d* );          // = viTP3d[1]
virtual void setVEnd(TVertex3d* );    // = viTP3d[2]
TVertex3d* getVStart(void) const;
TVertex3d* getVEnd(void) const;
void setTP3d(TP3d* pTP3d, int i);
TP3d* getTP3d(int i) const;
itTP3dContainer itviTP3d;
int getItemsInviTP3d(void) const;
/* funções para acessar Vstart e Vend */

void setpTShell(TShell* pTShell);
TShell* getpTShell(void) const;
/* funções para acessar pTShell */
void setpfcw(TFace* pTFace);
void setpfccw(TFace* pTFace);
TFace* getpfcw(void);
TFace* getpfccw(void);
/* funções para acessar pfcw e pfccw */
void setpncw(TWedge* pTWedge);
void setpnccw(TWedge* pTWedge);
void setppcw(TWedge* pTWedge);
void setppccw(TWedge* pTWedge);
TWedge* getpncw(void)  const;
TWedge* getpnccw(void) const;
TWedge* getppcw(void)  const;
TWedge* getppccw(void) const;
/* funções para acessar pncw, pnccw, ppcw e ppccw */
EdgeType gettipo(void);
void setColor(TColor cor);
TColor getColor(void) const;
void projWedge(void);
```

Projeta o objeto `TWedge` (i.e., constrói os `TEdgeWin`'s correspondentes) em todos os planos de projeção.

```
void projWedge(TProjPlane*);
```

Projeta o objeto `TWedge` (i.e., constrói os `TEdgeWin`'s correspondentes) no plano de projeção passado como parâmetro.

```
virtual void constructEdgeCurves(TEdgeWin*);
```

Método usado pelas funções de projeção para construir os objetos `TEdgeCurve` dos `TEdgeWin` correspondentes. Também é usado pelas funções de atualização de coordenadas de tela (vide código fonte para maior elucidação).

```
void showSelection(void);
void showSelection(TProjPlane* pTProjPlane);
void hideSelection(void);
void hideSelection(TProjPlane* pTProjPlane);
```

`ShowSelection(...)` seleciona; `hideSelection(...)` desseleciona. Quando não é passado nenhum parâmetro, chamam as funções de memo nome de todos os `TEdgeWin`'s associados; quando é

passado um plano de projeção como parâmetro, chamam as funções de mesmo nome apenas do `TEdgeWin` correspondente ao tal plano.

```
void refreshViewCoordinates(void);
```
Atualiza coordenadas de tela relativas a todos os planos de projeção.

```
void refreshViewCoordinates(TProjPlane*);
```
Atualiza coordenadas de tela relativas ao plano de projeção passado como parâmetro.

Classe ***TBezierWedge*** *("wedge.h")*

Classe derivada de `TWedge` à qual foi adicionada funcionalidade extra para representar segmentos que não sejam segmentos de reta necessariamente, podendo ser dados por uma cúbica de Bézier, por exemplo. Para tal, basta adicionar os pontos de controle adequados no container `viTP3d`.

Como o suporte a arestas não retilíneas ainda não está completamente desenvolvido, não serão fornecidos maiores detalhes sobre esta classe; consulte o código fonte, caso necessário.

Classe ***TLoop*** *("loop.h")*

Objetos da classe `TLoop` são usados para representar os *loops* internos e externo de cada face. Seus principais membros são:

```
/**************************************************************************
dados-membro protegidos
 **************************************************************************/

Orientation orient;
```
Define orientação do *loop* (positiva se concordante com a orientação do 1o. `TWedge`, negativa caso contrário).

```
LoopType tipo;
```
Define tipo do *loop*: interno ou externo. Algumas funções são feitas metamórficas em função deste dado-membro (i.e., mudam de comportamento em função do tipo do *loop*).

Todas as faces possuem um único *loop* externo, e zero ou mais *loops* internos.

```
TWedgeContainer viTWedge;
```
Container de ponteiros para todos os objetos `TWedge` que compõem o *loop*.

```
TFace* pTFace;
```
Ponteiro para o objeto `TFace` a que pertence o *loop*.

```
TVertex3d* pVStart;   //ponteiro para primeiro vértice do TLoop
void setTipo(LoopType ltype);
virtual void delLinkageTo(TLinkage* pTo);
```

```
virtual void delLinkageTo(int i);
/* anulam funções correspondentes da classe base */

/*************************************************************************
construtores
 *************************************************************************/

TLoop(TVertex3d* pVStart=NULL , TFace* pTFace=NULL,
Orientation orient=pos, LoopType ltype=outer);
TLoop(const TWedgeContainer& viTWedge, TFace* pTFace=NULL,
Orientation orient=pos, LoopType ltype=outer);
/* constrói TLoop a partir de um container de TWedges */

/*************************************************************************
operadores, dados funções-membro públicos
 *************************************************************************/
int countWedge(TWedge* pTWedge);
```

Retorna o número de ponteiros do objeto `TLoop` para o objeto `TWedge` passado como parâmetro (i.e., quantas vezes o `TWedge` aparece no *loop*).

```
TWedge* addTWedge(TWedge* pnewTWedge);
TWedge* addTWedgeAt(TWedge* pnewTWedge, int i);
int findTWedge(TWedge* pTWedge) const;
TWedge* getTWedge(int i) const;
TWedge* removeTWedge(TWedge* pTWedge);
TWedge* removeTWedge(int i);
itTWedgeContainer itviTWedge;
int getItemsInviTWedge(void);
/* funções para manipular o container de TWedge */

Orientation getOrientation(void);
virtual void setOrientation(Orientation);
LoopType getTipo(void);
void setpTFace(TFace* pTFace);
void setpVStart(TVertex3d* pvstart);
void adjustpVStart(void);
TFace* getpTFace(void) const;
TVertex3d* getpVStart(void) const;
virtual Orientation getTWedgeOrientation(TWedge* pTWedge)=0;
virtual void getTernaSeq(TTernaFloatContainer&) const = 0;

bool hasTP3d(TP3d*);
```

Retorna *true* caso o `TP3d` passado como parâmetro pertença ao *loop*, *false* caso contrário.

```
void hideSelection(void);
```

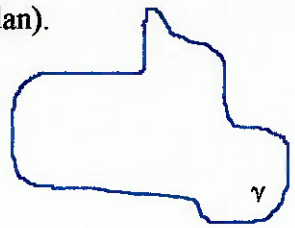
Desseleciona o *loop* (chama o método de mesmo nome de todos os `TWedge`'s do *loop*, desselecionando-os).

```
void showSelection(void);
```

Seleciona o *loop*. Como no caso anterior, também "rola" o comando de seleção hierarquia abaixo (i.e., chama o método de mesmo nome de todos os `TWedge`'s do *loop*, selecionando-os).

Classe **TSimpleLoopV** *("loop.h")*

Derivada da classe `TLoop`, objetos desta classe representam *loops* constituídos de uma curva fechada simples (curva de Jordan).



Curva de Jordan
no plano

Seus principais membros são:

```
/*************************************************************************
construtores
 *************************************************************************/
//contrutor default
TSimpleLoopV(TVertex3d* pVStart=NULL , TFace* pTFace=NULL,
Orientation orient=pos, LoopType ltype=outer);
TSimpleLoopV(const TWedgeContainer& viTWedge, TFace* pTFace=NULL,
Orientation orient=pos, LoopType ltype=outer);
/* constrói TLoop a partir de um container de TWedges */


/*************************************************************************
operadores, dados funções-membro públicos
 *************************************************************************/

TSimpleLoopValidator validator;
```

Este subobjeto membro é usado para verificar a validade do *loop* (i.e., se o *loop* é uma curva fechada simples ou não). Quando o *loop* é construído, métodos da classe `TSimpleLoopValidator` são chamados pelo construtor para verificar se o *loop* é válido; o resultado é armazenado num *flag* público do objeto `validator`, denominado `bIsTLoopValid`. Após construir-se o *loop*, deve-se consultar este *flag* para verificar se o *loop* é válido, e tomar as medidas adequadas caso não seja.

```
virtual void setOrientation(Orientation);
virtual Orientation getTWedgeOrientation(TWedge* pTWedge);
virtual void getTernaSeq(TTernaFloatContainer&) const ;
```

Retorna, no container passado por referência como parâmetro, a seqüência de ternas ordenadas que define o contorno do *loop*.

Classe **TFace** *("face.h")*

Objetos da classe `TFace` (classe base) e de suas derivadas representam faces. Cada face é constituída de um *loop* externo e de zero ou mais *loops* internos.

Os principais membros da classe `TFace` são os seguintes:

```
/*************************************************************************
dados-membro protegidos
 *************************************************************************/
```

```
TShell* pTShell;
```
Ponteiro para o objeto `TShell` ao qual pertence a face.

```
TLoop* pOuterLoop;
```
Ponteiro para o *loop* externo da face.

```
TLoopContainer viTLoop;
```
Container de ponteiros para os *loops* internos da face.

```
virtual void delLinkageTo(TLinkage* pTo);
virtual void delLinkageTo(int i);
/* anulam funções correspondentes da classe base */

/************************************************************************
construtores
 ************************************************************************/

//construtor default
TFace(FaceType ftype = flat);
TFace(TLoop* pOuterLoop, TShell* pTShell=NULL, FaceType ftype=flat);
TFace(const TLoopContainer& , TShell* pTShell=NULL, FaceType ftype=flat);
/* o primeiro TLoop do container é o TLoop externo; os demais, se houver,
são TLoops internos. O construtor também estabelece a ligação com o TShell
correspondente. */


/************************************************************************
destrutor
 ************************************************************************/
virtual ~TFace();
```
Deleta todos os *loops* da face (internos e externo).

```
/************************************************************************
operadores, funções-membro públicas
 ************************************************************************/

virtual TLoop* addTLoop(TLoop* pnewTLoop);
virtual TLoop* addTLoopAt(TLoop* pnewTLoop, int i);
int findTLoop(TLoop* pTLoop) const;
TLoop* getTLoop(int i) const;
virtual TLoop* removeTLoop(TLoop* pTLoop);
virtual TLoop* removeTLoop(int i);
itTLoopContainer itviTLoop;
int getItemsInviTLoop(void) const;
/* funções para manipular o container de TLoops */

TLoop* findTWedgeLoop(TWedge*);
```
Se o `TWedge` passado como parâmetro pertencer à face, retorna um ponteiro para o loop ao qual ele pertence, caso contrário retorna `NULL`.

```
virtual TLoop* setpOuterLoop(TLoop* pTLoop);
TLoop* getpOuterLoop(void) const; // acessa viTLoop[0] (TLoop externo)
void setColor(TColor cor);
TColor getColor(void) const;
FaceType getTipo(void) const;
void setpTShell(TShell* pTShell);
TShell* getpTShell(void) const;
void projFace(void);
```

Projeta a face em todos os planos de projeção. No caso de faces planas, este método não faz nada, pois faces planas não tem elementos correspondentes nas estruturas 2D associadas aos planos de projeção, como já foi explicado.

```
virtual void projFace(TProjPlane*);
```

Projeta a face no plano de projeção passado como parâmetro. Vale a mesma observação anterior caso a face seja plana.

```
virtual void toggleFaceOrientation(void);
```

Método usado para inverter a orientação da face. É sempre chamado pelo método `toggleShellOrientation(...)`, da classe `TShell`, usado para inverter a orientação de um objeto da referida classe (i.e., comutar entre orientação dada por campo normal externo ou interno). `ToggleShellOrientation(...)` comanda todas as faces do sólido a inverterem a sua orientação por uma chamada a `toggleFaceOrientation(...)`; este método, por sua vez, "rola" o comando hierarquia abaixo, ordenando cada *loop* a inverter a sua orientação. Vide código fonte para maiores detalhes.

```
virtual void showSelection(bool showBoundaries = true);
virtual void showSelection(TProjPlane*, bool showBoundaries = true);
virtual void hideSelection(bool hideBoundaries = true);
virtual void hideSelection(TProjPlane*, bool hideBoundaries = true);
```

Métodos usados para selecionar ou desselecionar uma face, análogos aos métodos de mesmo nome das classes `TLoop` e `TWedge`, já estudados. Também usa a filosofia de "rolar" o comando hierarquia abaixo, ordenando para que cada um de seus *loops* e curvas isométricas (no caso de faces não-planas) sejam selecionados ou desselecionados. Quando o *flag* `showBoudaries` for *false*, o comando é "rolado" apenas as curvas isométricas da face (caso existam, i.e., caso a face seja não-plana). Quando o referido *flag* for *true*, o comando também é "rolado" para os *loops* que delimitam a face.

Classe ***TBezierFace*** *("face.h")*

A classe `TBezierFace` foi derivada da classe `TFace` para suportar faces não-planas, dadas por *patches* triangulares de Bézier. O suporte a faces não-planas ainda não foi completamente desenvolvido, como já foi mencionado, por isto objetos desta classe não são criados. Consulte o código fonte para maiores detalhes.

Classe **TFlatFaceV** ("face.h")

A classe `TFlatFaceV` foi derivada de `TFace` para oferecer suporte e verificação à construção de objetos que representem faces planas. O construtor de `TFlatFaceV` chama métodos que verificam a validade da face (por exemplo, se todos os seus pontos estão no mesmo plano). O resultado destas verificações é colocado no *flag* público `bIsTFaceValid`. Assim, após constuir o objeto `TFlatFaceV`, deve-se sempre consultar este *flag* para verificar se a face construída é válida, e tomar as medidas cabíveis (deletar o objeto, por exemplo) caso não seja.

Os principais métodos desta classe são:

```
/*************************************************************************
dados e funções-membro protegidos
 *************************************************************************/

virtual void delLinkageTo(TLinkage* pTo);
virtual void delLinkageTo(int i);
/* anulam funções correspondentes da classe base */

TPlane plane;
```

Subobjeto protegido que descreve o plano da face. Os métodos públicos `getp()`, `setp()`, `getn()` e `setn()` podem ser usados para acessar os parâmetros que definem este objeto (vide classe `TPlane`).

```
bool isLoopOrientationDefined(const TSimpleLoopV&);
```

Método que verifica se a orientação do *loop* passado como parâmetro já está definida (i.e., se existe algum `TWedge` do *loop* que pertença a outra face; caso isto ocorra, a orientação desta face deverá induzir a orientação do referido *loop*, estando a mesma, portanto, já definida). Esta função é chamada pelos métodos sobrecarregados `setLoopOrientation(...)`, descritos logo mais, usados para tentar orientar os *loops* internos ou externo coerentemente com as demais faces do sólido.

```
bool setLoopOrientation(TSimpleLoopV&);
```

Método chamado pela função virtual `setpOuterLoop(...)` para tentar orientar o *loop* externo coerentemente com as outras faces do sólido (caso isto não seja possível, retorna *false*) e calcular o versor normal à face também coerentemente com esta orientação (por chamada ao método `calcNormal(...)`).

```
bool setLoopOrientation(TSimpleLoopV&, TTernaFloat n);
```

Método chamado pela função virtual `AddTLoop(...)` para definir a orientação de um *loop* interno de acordo com a normal à face (passada no parâmetro `n`). Retorna *false* caso não seja

possível orientar o referido *loop* com a orientação induzida pelo campo normal (i.e., caso a orientação do *loop* já esteja definida no sentido contrário, induzida por outra face).

```
TTernaFloat calcNormal(const TSimpleLoopV&);
```

Dado um *loop* plano (passado como parâmetro), calcula versor normal ao plano do mesmo, com orientação induzida pela orientação do *loop*. Retorna terna ordenada com as coordenadas de mundo deste versor. Antes de chamar este método, deve-se verificar se o *loop* é plano, por chamada ao método `isLoopFlat(...)`.

```
TP3dContainer get3NCTP3dInLoop(const TSimpleLoopV&);
```

Retorna um container de ponteiros para objetos `TP3d` com ponteiros para três `TP3d`'s não colineares pertencentes ao *loop*. Caso não haja três pontos não colineares no *loop*, retorna um container vazio.

```
bool isLoopFlat(const TSimpleLoopV&);
```

Retorna *true* caso o *loop* passado como parâmetro seja plano, *false* caso contrário.

```
bool isLoopOnPlane(const TSimpleLoopV&, const TPlane&);
```

Retorna *true* caso o *loop* passado como parâmetro esteja no plano também passado, *false* caso contrário.

Consulte o código fonte para maiores detalhes sobre estes métodos.

```
/*************************************************************************
construtores
 *************************************************************************/


//construtor default
TFlatFaceV(TSimpleLoopV* pOuterLoop=NULL, TShell* pTShell=NULL);

TFlatFaceV(const TLoopContainer& , TShell* pTShell=NULL);
/* o primeiro TLoop do container é o TLoop externo; os demais, se houver,
são TLoops internos. O construtor também estabelece a ligação com o TShell
correspondente. */




/*************************************************************************
dados e funções-membro públicos
 *************************************************************************/

virtual ~TFlatFaceV();
bool bIsTFaceValid;
```

Flag para indicar se o objeto `TFlatFaceV` construído é válido ou não, conforme já explicado.

```
void setp(TTernaFloat);
void setn(TTernaFloat);
TTernaFloat getp(void) const;
TTernaFloat getn(void) const;
```

Estes quatro métodos podem ser usados para acessar os parâmetros que definem o plano da face, como já foi mencionado.

```
/* só deve ser chamado no construtor (este método está protegido)*/
virtual TLoop* setpOuterLoop(TLoop* pTLoop);
virtual TLoop* addTLoop(TLoop* pnewTLoop);
virtual TLoop* addTLoopAt(TLoop* pnewTLoop, int i);
virtual TLoop* removeTLoop(TLoop* pTLoop);
virtual TLoop* removeTLoop(int i);
```

Anulam métodos homônimos da classe base (`TFace`).

```
virtual void toggleFaceOrientation(void);
```

Método usado para inverter a orientação da face, conforme já explicado.

```
virtual void refreshWorldCoordinates(void);
```

Atualiza coordenadas de mundo dos parâmetros que definem o plano da face (i.e., `p` e `n`, vide classe `TPlane`).

Classe ***TShell*** *("shell.h")*

Objetos da classe `TShell` representam sólidos. Seus principais membros são os seguintes:

```
/*************************************************************************
dados-membro privados
 *************************************************************************/

bool bWorldCoordValid;
```

Flag que indica se as coordenadas de mundo dos `TP3d`'s pertencentes ao sólido são válidas ou não. Este *flag* também deve ser usado pelo sistema de atualização de coordenadas em segundo plano, ainda a ser implementado (vide discussão na classe `TPlane`).

```
TLayer* pTLayer;
```

Ponteiro para o objeto `TLayer` a que pertence o sólido.

```
TFaceContainer viTFace;
```

Container de ponteiros para todas as faces que compõem o sólido.

```
TWedgeContainer viTWedge;
```

Container de ponteiros para todos os `TWedge`'s que pertencem ao sólido.

```
TP3dContainer viTP3d;
```

Container de ponteiros para todos os `TP3d`'s que pertencem ao sólido.

```
TMatProjPlaneContainer vTMatProjPlane;
```

Container de objetos da classe `TMatProjPlane`. Cada objeto desta classe possui um ponteiro para um objeto `TProjPlane` e três matrizes de transformação homogênea para transformar as

coordenadas de mundo do sólido em *eye*, *projection* e *view coordinates* relativas ao respectivo plano de projeção.

A princípio, poderia parecer desnecessário armazenar estas matrizes em todo objeto `TShell`. Isto seria verdade se elas fossem as mesmas para todos os sólidos, dependendo apenas do sistema de coordenadas de mundo e dos sistemas de coordenadas associados aos planos de projeção. Mas não é o que ocorre, pois, quando um sólido muda de posição no espaço, as suas coordenadas de mundo não são atualizadas imediatamente (isto deve ser feito em segundo plano, como já foi discutido, vide discussão sobre este assunto na classe `TProjPlane`); ao invés disto, a matriz de transformação homogênea que descreve o movimento do sólido é armazenada. Cada uma das matrizes de transformação homogênea de `vTMatProjPlane` é, então, multiplicada por esta matriz, armazenada no membro público `correctorMat`; é por isto que as referidas matrizes não dependem apenas do sistema de coordenadas de mundo e dos sistemas de coordenadas dos planos de projeção.

As matrizes deste container são atualizadas sempre que o sólido muda de posição no espaço, como discutido no parágrafo anterior, e sempre que algum plano de projeção (ou *viewport* correspondente) é criado ou modificado. Vide código fonte para maiores detalhes.

```
virtual void delLinkageTo (TLinkage* pTo);
virtual void delLinkageTo (int i);
/* anulam funções "delLinkageTo" da classe base */


/*************************************************************************
construtores
 *************************************************************************/

//construtor default
TShell(TFace* pTFace0=NULL, ShellType stype=opened, EditMode mode=off);

/*************************************************************************
destrutor
 *************************************************************************/
virtual ~TShell();
```

Deleta todos os objetos `TFace`, `TWedge` e `TP3d` pertencentes ao sólido.

```
/*************************************************************************
operadores, funções-membro públicas
 *************************************************************************/

int v, e, f, h, r;
```

Estes membros mantêm contagem do número de vértices, *edges*, faces, *holes* e *rings* do sólido, respectivamente; são usados para habilitação dos comandos que aplicam operadores de Euler sobre o sólido.

```
TMatDirInv correctorMat;
```

Este membro é usado para armazenar matrizes de transformação homogênea associadas a movimentos do sólido; como jé foi explicado, deverá fazer parte do mecanismo de atualização de coordenadas em segundo plano.

```
/* funções para manipular o container de TWedge */
TWedge* addTWedge(TWedge* pnewTWedge);
TWedge* addTWedgeAt(TWedge* pnewTWedge, int i);
int findTWedge(TWedge* pTWedge) const;
TWedge* getTWedge(int i) const;
TWedge* removeTWedge(TWedge* pTWedge);
TWedge* removeTWedge(int i);
itTWedgeContainer itviTWedge;
int getItemsInviTWedge(void) const;

/* funções para manipular o container de TFace */
TFace* addTFace(TFace* pnewTFace);
TFace* addTFaceAt(TFace* pnewTFace, int i);
int findTFace(TFace* pTFace) const;
TFace* getTFace(int i) const;
TFace* removeTFace(TFace* pTFace);
TFace* removeTFace(int i);
itTFaceContainer itviTFace;
int getItemsInviTFace(void) const;

/* funções para manipular o container de TP3d */
TP3d* addTP3d(TP3d* pnewTP3d);
TP3d* addTP3dAt(TP3d* pnewTP3d, int i);
int findTP3d(TP3d* pTP3d) const;
TP3d* getTP3d(int i) const;
TP3d* removeTP3d(TP3d* pTP3d);
TP3d* removeTP3d(int i);
itTP3dContainer itviTP3d;
int getItemsInviTP3d(void) const;


void setpTLayer(TLayer* pTLayer);
TLayer* getpTLayer(void) const;
/* funções para manipular o container de TMatProjPlane */
TMatProjPlane addTMatProjPlane(TMatProjPlane newTMatProjPlane);
TMatProjPlane addTMatProjPlaneAt(TMatProjPlane newTMatProjPlane, int i);
int findTMatProjPlane(TMatProjPlane TMatProjPlaneIn) const;
TMatProjPlane& getTMatProjPlane(TProjPlane*) const;
TMatProjPlane& getTMatProjPlane(int i) const;
void setMatEye (TMatDirInv eye, TProjPlane* pPlane);
void setMatProj(TMatTrans proj, TProjPlane* pPlane);
void setMatView(TMatTrans view, TProjPlane* pPlane);
TMatProjPlane removeTMatProjPlane(TMatProjPlane TMatProjPlaneIn);
TMatProjPlane removeTMatProjPlane(int i);
itTMatProjPlaneContainer itvTMatProjPlane;
int getItemsInviTMatProjPlane(void) const;


TTernaFloat worldNew2worldOld(TTernaFloat);
TTernaFloat worldOld2worldNew(TTernaFloat);
TTernaFloat world2eye(TTernaFloat ternaIn, TProjPlane* pPlane);
TTernaFloat eye2world(TTernaFloat ternaIn, TProjPlane* pPlane);
TTernaFloat world2proj(TTernaFloat ternaIn, TProjPlane* pPlane);
```

```
TTernaFloat proj2world(TTernaFloat ternaIn, TProjPlane* pPlane);
TParInt     world2view(TTernaFloat ternaIn, TProjPlane* pPlane);
```

Funções de conversão de coordenadas, análogas às vistas na classe `TProjPlane`.

```
void projShell(void);
```

Projeta o sólido em todos os planos de projeção. Também usa a filosofia de "rolar" o comando hierarquia abaixo, ordenando todos os seus `TP3d`'s, `TWedge`'s e `TFace`'s a se projetarem.

```
void projShell(TProjPlane*);
```

Método análogo ao anterior; projeta o sólido no plano de projeção passado como parâmetro.

```
void adjustMat(void);
void adjustMat(TProjPlane*);
void adjustEyeMat(void);
void adjustEyeMat(TProjPlane*);
void adjustProjMat(void);
void adjustProjMat(TProjPlane*);
void adjustViewMat(void);
void adjustViewMat(TProjPlane*);
```

Métodos usados para atualizar as matrizes de transformação homogênea do container `vTMatProjPlane` (vide discussão acima).

```
void refreshWorldCoordinates(void);
void refreshProjCoordinates(void);
void refreshProjCoordinates(TProjPlane*);
void refreshViewCoordinates(void);
void refreshViewCoordinates(TProjPlane*);
```

Métodos usados para atualização das coordenadas de mundo, de plano de projeção e de tela da estrutura `3D` e estruturas `2D` associadas ao sólido.

```
bool openShell(TFlatFaceV* pTFace=NULL);
```

Abre um sólido fechado na face passada como parâmetro, deletando a mesma. Retorna *true* caso a operação seja bem sucedida, *false* caso contrário.

```
bool closeShell(void);
```

Fecha um sólido aberto, acrescentando uma face ao mesmo. Se for impossível obter um sólido válido pelo acréscimo de uma face, ou se a face a ser acrescentada não for válida, a operação não é feita e o método retorna *false*.

```
void toggleShellOrientation(void);
```

Inverte a orientação do sólido (i.e., comuta entre orientação definida pelo campo normal externo ou interno). Também usa a filosofia de "rolar" o comando abaixo na hierarquia, ordenando todas faces que o compõem a inverterem suas orientações.

```
void showSelection(void);
void showSelection(TProjPlane*);
void hideSelection(void);
void hideSelection(TProjPlane*);
```

Análogas aos métodos homônimos já vistos nas classes `TFace`, `TLoop` e `TWedge`; também usa a filosofia de "rolar" o comando abaixo na hierarquia, ordenando todos os seus `TWedge`'s a serem selecionados ou desselecionados.

Classes **TLayer** *("layer.h")* *e* **TScene** *("scene.h")*

Objetos das classes `TLayer` e `TScene` são usados para representar *layers* e cenas, respectivamente.

Shells são agrupados em *layers*; cada *layer* pode estar num estado visível ou invisível, em função do estado do membro privado `layerMode` (isto ainda não foi completamente implementado, mas, para tal, basta acrescentar algumas poucas linhas de código). Há um ponteiro global, de nome `pCurrentLayer`, que aponta para o *layer* corrente da cena corrente do aplicativo.

Layers, por sua vez, são agrupados em cenas. Há um ponteiro global, denominado `pCurrentScene` (vide declaração de objetos globais em "`globals.h`"), que aponta para a cena corrente do aplicativo. Futuramente, ao se implementar um modelo *doc/view*, cada cena deverá corresponder a um documento, como já foi discutido.

Como o código fonte destas duas classes é curto e de fácil entendimento, não se entrará em maiores detalhes sobre o mesmo.

7.8 Operadores de Euler *("euler.h")*

Os operadores de Euler foram implementados através de classes que derivam da classe base `TEulerOperator` (derivada, por sua vez, da classe `Toperator`).

Como já foi comentado, todas estas classes foram feitas serializáveis ("*streamable*"), de modo a permitir comunicação com memória externa e futuro desenvolvimento de um "*undo*".

Por exemplo, suponha que desejamos aplicar um operador sobre algum sólido, digamos, *mev*. Para tal, basta criar um objeto da classe *mev*, com os devidos parâmetros, e o construtor do mesmo se encarrega de efetuar a operação. A partir daí, cabe ao programador definir o destino do objeto criado. Para implementar-se um "*undo*", o objeto poderia ser colocado numa pilha de operações; sendo necessário desfazer a última operação, bastaria retirar (*pop*) o último objeto "operador" da pilha e executar a operação inversa correspondente (*kev*, caso seja retirado um objeto *mev*, por exemplo), com os devidos parâmetros, que podem ser obtidos a partir dos dados-membros (estado) do objeto que foi retirado da pilha.

O fato de os objetos serem serializáveis permite que uma cena possa ser "armazenada" em memória externa; para isto, basta serializar a pilha de operações para um arquivo. A seqüência de operações seria, então, uma linguagem de descrição da cena.

Caso não se deseje criar um objeto "operador" para efetuar uma dada operação, basta chamar uma função estática que foi colocada em cada uma das classes de operadores de Euler. Assim, para se executar uma operação *mev*, pode-se tanto criar um objeto `TMEV` (o construtor executa a operação) como chamar a função estática `TMEV::MEV(...)`, com os devidos parâmetros. Até o momento, apenas o último método foi testado; o primeiro (criar um objeto "operador"), apesar de projetado e implementado, ainda não foi testado. Mas, como foi discutido no parágrafo anterior, poderá ser necessário para implementar-se um "*undo*" e comunicação com memória externa.

Consulte o código fonte das classes de operadores de Euler para maiores detalhes. Como as classes são curtas e o código de fácil entendimento, o mesmo não será abordado aqui.

7.9 *Interface* **Windows**

A interface *Windows* do aplicativo está centralizada num objeto da classe `TWindow` : a janela cliente da janela principal do aplicativo.

Como já foi dito, esta interface será modificada futuramente, devendo ser implementada por um modelo *MDI - doc/view*, ou *SDI - doc/ view*; neste caso, a cada cena (*"TScene"*) corresponderá

um documento ("*TDocument*") e a cada plano de projeção corresponderá uma vista ("*TView*") do referido documento.

Quando um item de menu ou botão da barra de ferramentas é pressionado, uma mensagem de comando é gerada; tal mensagem é colocada na fila de mensagens do aplicativo, e capturada pelo *loop* de "bombeamento" mensagens do mesmo. Sendo capturada a referida mensagem, o aplicativo pesquisa em cada tabela de respostas de mensagens das classes que estão no caminho de roteamento de comandos (mesagens do tipo `WM_COMMAND`) até encontrar uma função de resposta correspondente em alguma tabela; no caso tal função será encontrada na tabela de resposta da classe `interWindow` , derivada da classe `TWindow` e à qual pertence o objeto janela cliente da janela principal do aplicativo.

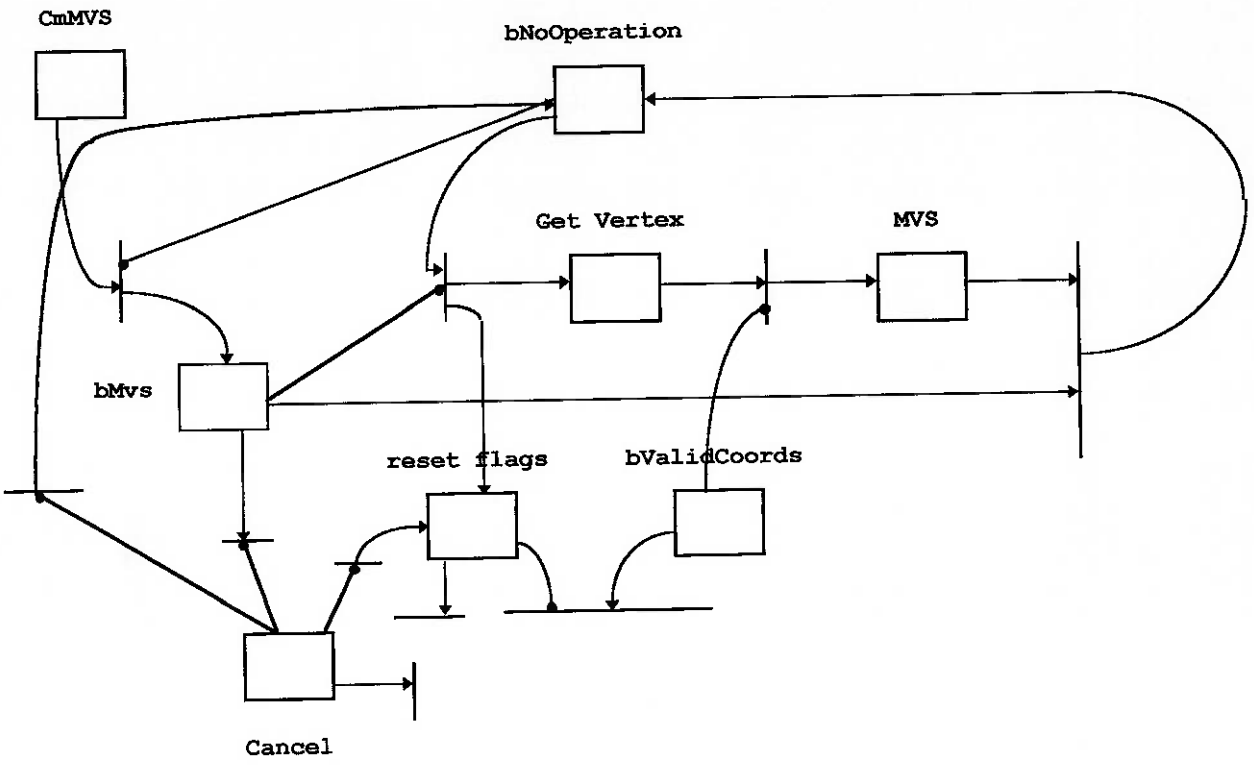
Quase todos os comandos da interface exigem que algum dado seja fornecido como argumento. Por exemplo, pode ser necessário que se forneça as coordenadas de um ponto no E^3, ou que se selecione um vértice, aresta, face, *ring* ou sóido. Assim, ao ser selecionado o item *MEV* do menu *Euler*, uma mensagem de comando com o *ID* do item de menu *MEV* é gerada e colocada na fila de mensagens do aplicativo. Após ser capturada esta mensagem, é chamada a função de resposta de mensagem correspondente, pelo processo descrito no parágrafo acima : o método `CmMEV(...)` da classe `interWindow`, no caso - vide tabela de respostas de mensagens da referida classe. Este método, então, pede que o usuário forneça os argumentos necessários à execução da operação *mev*: seleção de um vértice e entrada das coordenadas de um outro vértice.

A entrada de coordenadas e a seleção de objetos é feita através do *mouse*. Os botões do *mouse* são sobrecarregados; o evento "disparado" por eles é tratado de forma diferente em função do estado do aplicativo. Este estado, por sua vez, é descrito por um conjunto de *flags* globais (variáveis de estado), declaradas no arquivo "`flags.h`" e definidas em "`flags.cpp`".

O código da classe `interWindow` não será discutido aqui; consulte a listagem da referida classe para maiores detalhes. Serão apresentados os digramas ***PFS/MFG*** utilizados para modelar e analisar algumas funções da referida classe, e a seguir as classes usadas para implementar objetos "cursor" e "sistema de coordenadas".

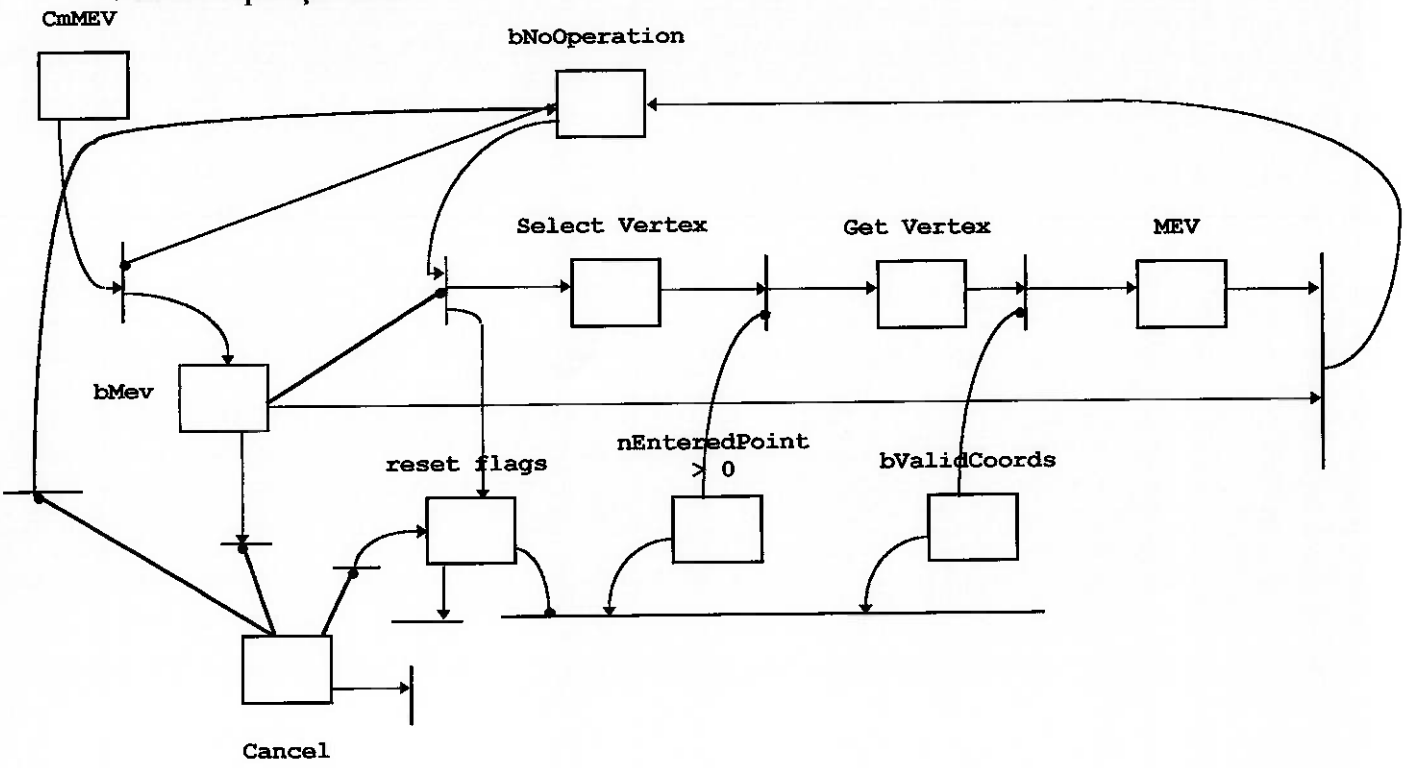
Função ***CmMVS( )***

O método CmMVS(), da classe interWindow, é chamado em resposta ao evento disparado pela seleção do item de menu New do menu Shell. Tal método executa uma operação *mvs* (vide apêndice "A" - operadores de Euler), após pedir para que o usuário forneça os parâmetros necessários. Este método foi modelado pelo seguinte *MFG*:

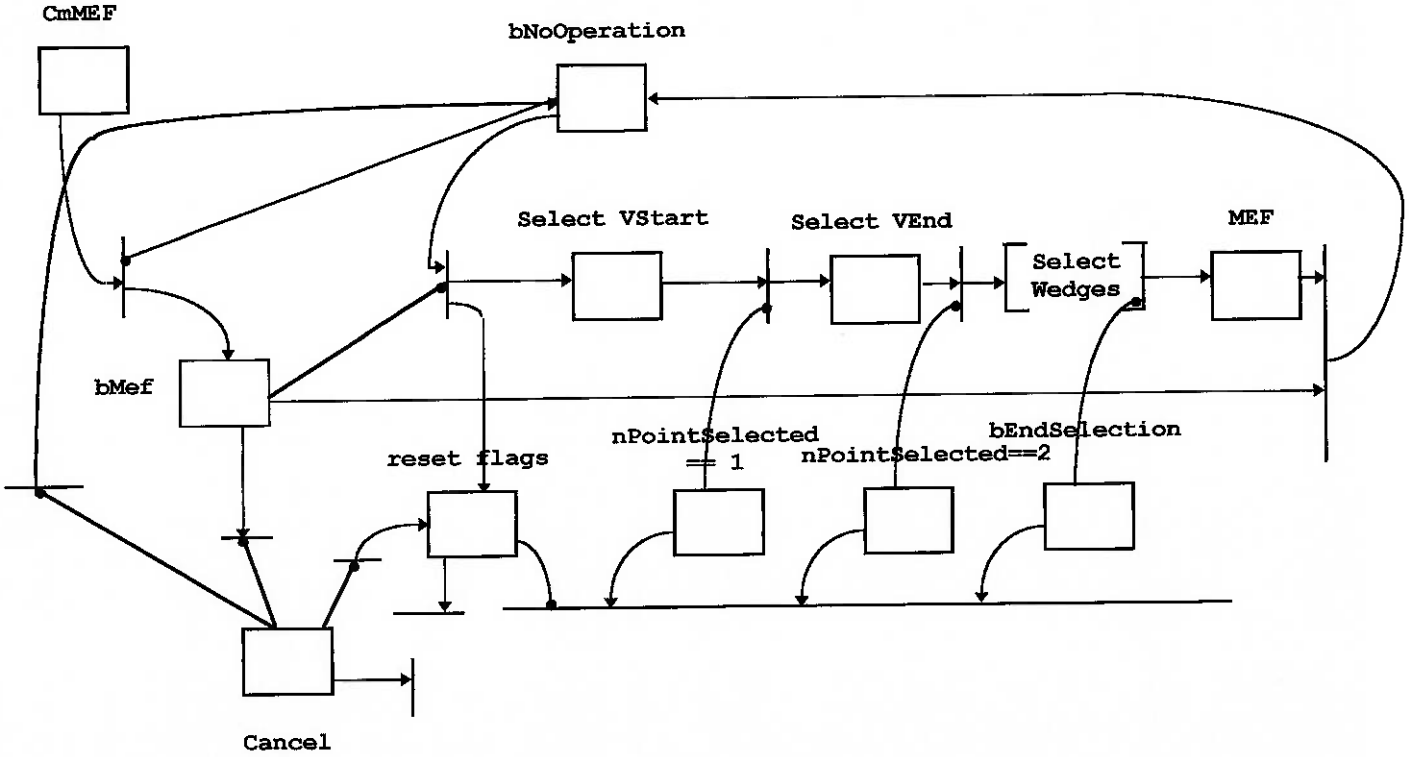


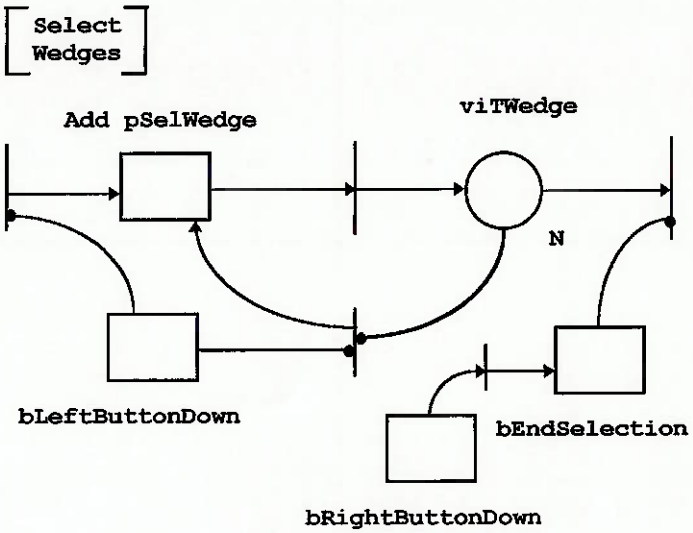
Função **CmMEV( )**

Executa operação *mev*.



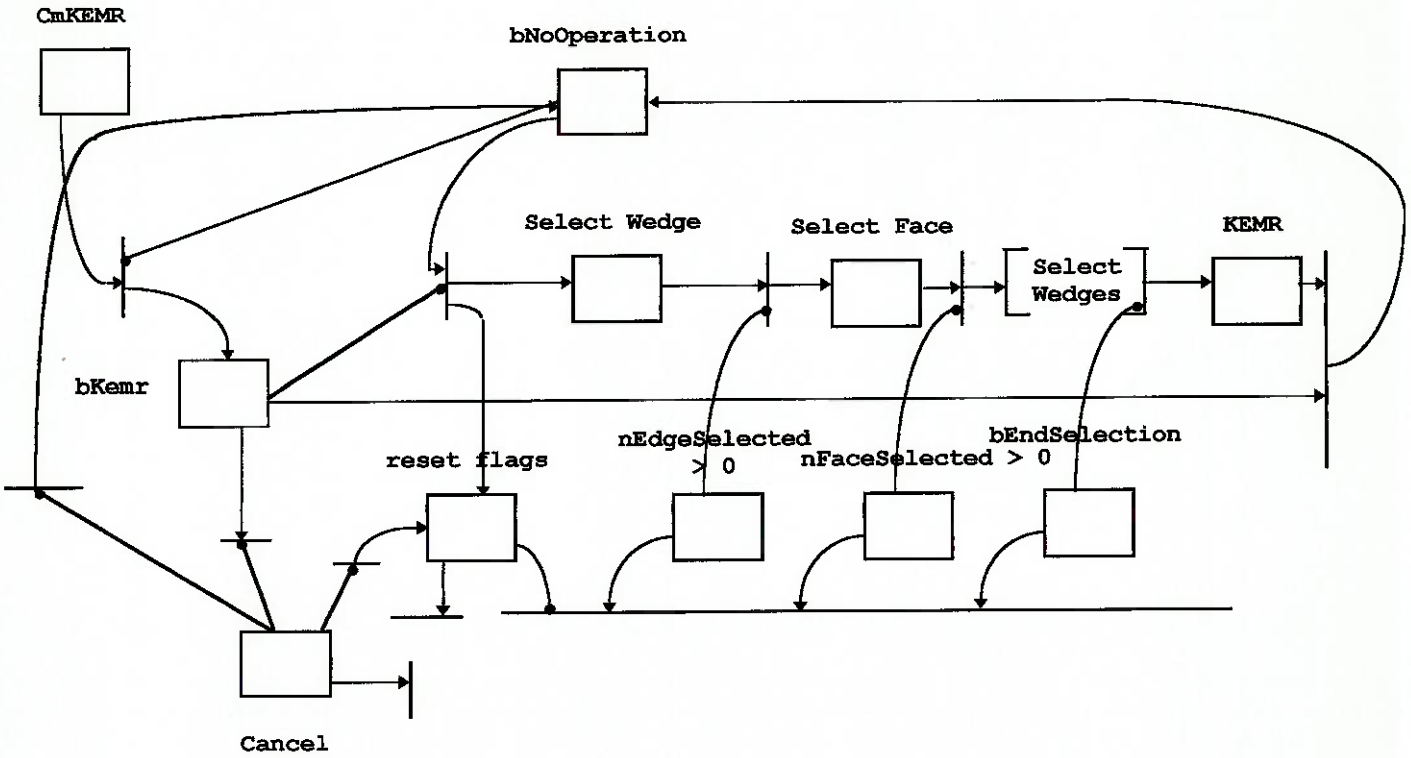
Função **CmMEF( )**





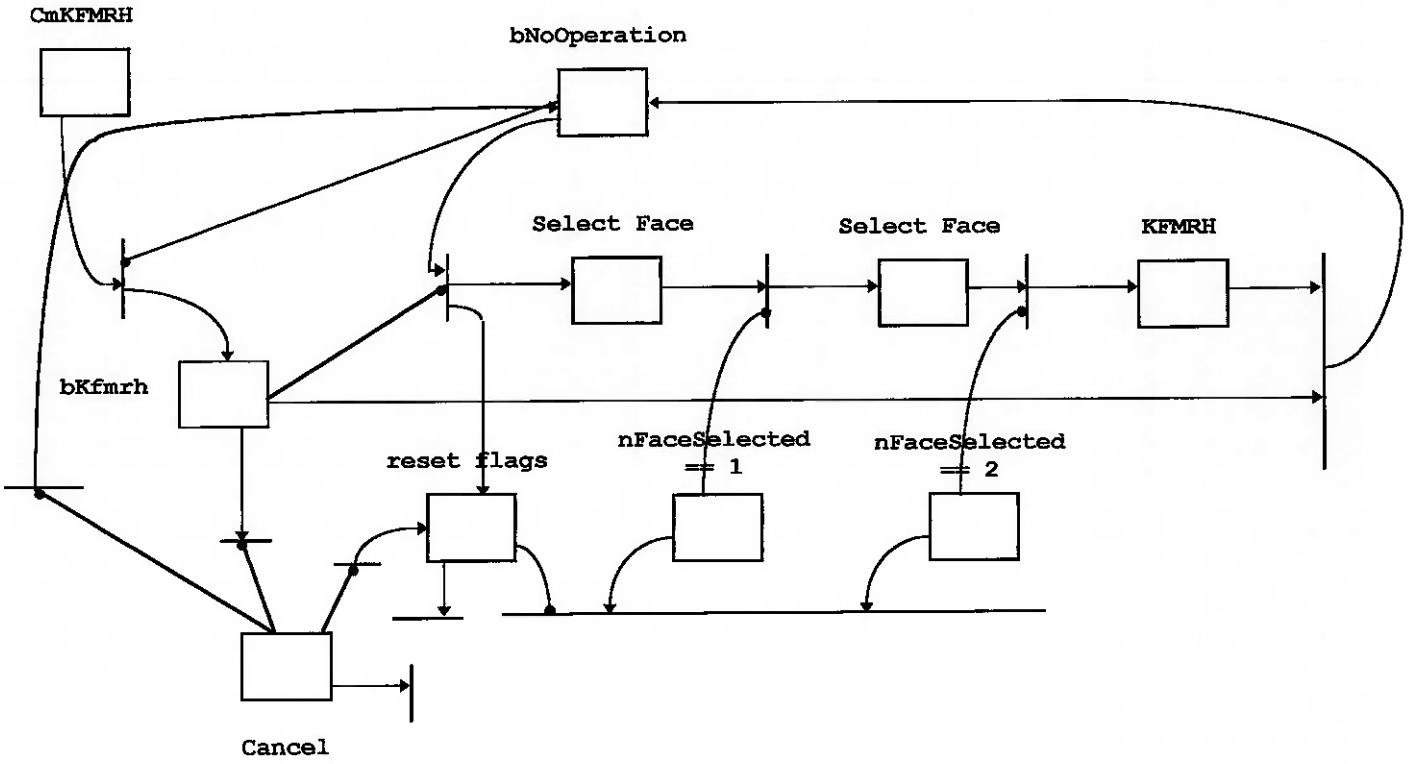
Função ***CmKEMR( )***

Executa operação *kemr*.



Função ***CmKFMRH***

Executa operação *kfmrh*.



Classe ***D3dCursor*** *("cursor.h")*

Cada objeto desta classe, derivada de `DCursor`, é um cursor tridimensional (cursor usado para obter coordenadas de pontos do E^3.

Os principais membros desta classe são os seguintes:

```
/*************************************************************************
        dados-membro protegidos
 *************************************************************************/
```

```
bool findSegCursorIntercept(TTernaFloat P1, TTernaFloat P2, TFace* pTFace);
```

Encontra intersecção entre o segmento `(p1,p2)` e a face passada como parâmetro; o ponto de intersecção é adicionado ao container protegido `intercepts`. Caso não encontre intersecção, retorna *false*. Este método é chamado pela função protegida `findFaceCursorIntercepts(...)`, descrita abaixo.

```
bool findFaceCursorIntercepts(TFace*);
```

Encontra intersecção entre as arestas do paralelepípedo do cursor e a face passada como parâmetro. Na verdade, para que o programa não fique muito lento, por enquanto estão sendo calculadas apenas as intersecções das três arestas ligadas ao "ponto corrente" do cursor.

```
void refreshP3dCoordinates(TP3d&);
```

Método utilizado para atualizar as coordenadas dos vértices do cursor, quando o mesmo muda de posição.

```
TTernaFloat ui, uj, uk;
```

Versores nas direções `i`, `j` e `k`, em relação ao sistema de coordenadas ligado ao cursor.

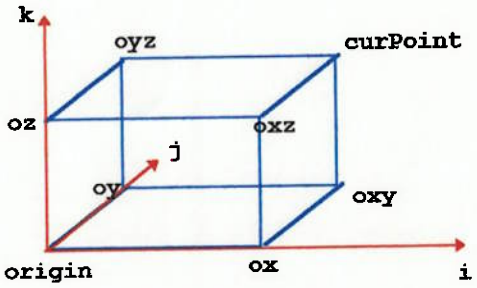
```
TTernaFloatContainer intercepts;
```

Container de ternas ordenadas com as intersecções do cursor com as faces da cena corrente, na posição atual do cursor; quando o cursor é desenhado, é plotada uma marca ao redor de cada uma destas intersecções, de modo a fornecer informações visuais em três dimensões ao usuário.

```
TTernaFloatContainer oldIntercepts;
```

Não utilizado.

```
/*************************************************************************
        dados, operadores e funções-membro públicas
 *************************************************************************/
// todas as coordenadas são absolutas
TP3d curPoint;
TP3d origin;
TP3d ox;
TP3d oy ;
TP3d oz;
TP3d oxy;
TP3d oyz ;
TP3d oxz   ;
```




```
void move(TTernaFloat);
```

Desloca a posição corrente do cursor para o ponto passado como parâmetro, atualiza as coordenadas de todos os pontos do cursor e acha os interceptos com as faces da cena corrente na nova posição.

Os campos e métodos acima são herdados da classe `DCursor`; os campos e métodos seguintes são da classe `D3dCursor`.

```
/*************************************************************************
            construtores
 *************************************************************************/
D3dCursor(TTernaFloat points[4]);
```

Constrói objeto `D3dCursor` a partir das coordenadas dos seus pontos `origin`, `ox`, `oy` e `oz`.

```
/*************************************************************************
        dados, operadores e funções-membro públicas
 *************************************************************************/

virtual void draw(DrawMode = show) const;
virtual void draw(TProjPlane*, DrawMode = show) const;
virtual void draw(TProjPlane*, TDC&, DrawMode = show) const;
```

Métodos usados para desenhar o cursor, semelhantes aos muitos já vistos até aqui. As diferenças entre cada um deles são explicadas no código fonte; o parâmetro `DrawMode` não é usado.

Classe **D2dCursor** *("cursor.h")*

Objetos da classe `D2dCursor` são cursores bidimensionais, usados para seleção. Cada objeto da classe `TWindowData` possui um membro público desta classe, denominado `cursor` (vide classe `TWindowData`). Seus principais membros são:

```
/*************************************************************************
      dados-membro protegidos
 *************************************************************************/
```

```
int tam;
```

Tamanho do cursor.

```
SelType selMode;
```

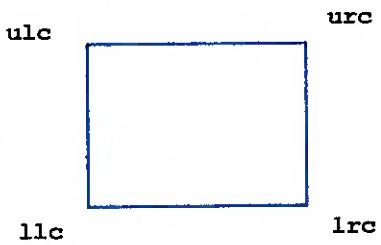
Modo de seleção corrente do cursor. O tipo `SelType` é um `enum` declarado no arquivo "`definit.h`". Consulte na listagem deste arquivo a declaração deste e de vários outros `enum`'s usados no programa.

```
CursorType tipo;
```

Tipo do cursor. Consulte em "`definit.h`" a declaração de `CursorType`.

```
TParInt ulc;
TParInt urc;
TParInt llc;
TParInt lrc;
```

Coordenadas de tela dos vértices do cursor.




```
TWindowData* pTWindowData;
```

Ponteiro para o objeto `TWindowData` a que pertence o cursor.

```
TP3d* pCurSelP3d;
```

Ponteiro para o objeto `TP3d` atualmente selecionado, caso em modo de seleção de pontos.

```
TWedge* pCurSelWedge;
```

Ponteiro para o objeto `TWedge` atualmente selecionado, caso em modo de seleção de *edges*.

```
/*************************************************************************
            construtores
 *************************************************************************/
D2dCursor(TWindowData* pWin = NULL, TParInt point = zeroTernaInt,
          int tamanho = curMarkSize, SelType tsel = edge);       // contrutor default
```

O parâmetro `point` é usado para inicializar o membro público `curPoint`, par ordenado com as coordenadas de tela do ponto corrente do cursor (centro do cursor).

```
/*************************************************************************
        dados, operadores e funções-membro públicas
 *************************************************************************/
TParInt curPoint;
```

Coordenadas de tela do ponto corrente do cursor.

```
TParInt ancora;
```

Coordenadas de tela da âncora (usada para desenhar uma linha elástica à medida que o cursor se move - vide código fonte dos comandos *pan* e *zoom window*, por exemplo).

```
bool contains(TParInt);
```

Método usado por `findSelection()` para verificar se um ponto está dentro da região de seleção.

```
bool findSelection(void);
```

Percorre todos os objetos da estrutura `2D` associada ao cursor até encontrar algum dentro da região de seleção do mesmo; o tipo de objeto a ser selecionado depende do modo corrente de seleção, definido pelo membro protegido `tipo`.

```
TWindowData* setpTWindowData(TWindowData*);
TWindowData* getpTWindowData(void) const;

TP3d* setpCurSelP3d(TP3d*);
TWedge* setpCurSelWedge(TWedge*);
TP3d* getpCurSelP3d(void) const;
TWedge* getpCurSelWedge(void) const;

void changeCursorTam(int);
void changeSelMode(SelType);

void move(TParInt);
void setPos(TParInt);
```

Métodos usados para deslocar a posição corrente do cursor para o ponto passado como parâmetro (coordenadas de tela). Diferenças entre estes dois métodos são explicadas no código fonte.

```
virtual void draw(void) const ;
virtual void draw(TDC&) const ;
```

Desenham o cursor.

```
CursorType getTipo(void) const;
void setTipo(CursorType);
```

Classe **TCoordSys** *("coordsys.h")*

Objetos da classe `TCoordSys` são usados para descrever sistemas de coordenadas. O ponteiro global denominado `pCurrentCoordSys` (vide declaração de objetos globais em "`globals.h`") aponta para o sistema de coordenadas corrente.

Os principais membros desta classe são os seguintes:

```
/*************************************************************************
        dados-membro protegidos
 *************************************************************************/
void setMatChangeBase(TTernaFloat points[4]);
```

Método protegido usado para atualizar a matriz de mudança de base do sistema coordenadas descrito por um objeto desta classe para o sistema de coordenadas de mundo. Esta matriz é armazenada no membro público `matChangeBase`.

```
/*************************************************************************
            construtores
 *************************************************************************/
TCoordSys(TTernaFloat points[4] = defaultTriedro);     // contrutor default
```

Cada objeto `TCoordSys` possui um ponteiro para um objeto da classe `Dcursor`. Os parâmetros passados para o construtor são usados para construir este objeto (vide construtor da classe `DCursor`).

```
/*************************************************************************
       destrutor
 *************************************************************************/
virtual ~TCoordSys();
```

Destrói o objeto da classe `DCursor` correspondente.

```
/*************************************************************************
        dados, operadores e funções-membro públicas
 *************************************************************************/
DCursor* cursor;
```

Ponteiro para objeto da classe `DCursor`, já mencionado.

```
TMatDirInv matChangeBase;
```

Matriz de mudança de base do sistema de coordenadas descrito por um objeto desta classe para o sistema de coordenadas de mundo.

```
TTernaFloat cursorLocalCoord;   // coordenadas locais
```

Coordenadas do cursor (objeto apontado pelo membro público `cursor`) em relação ao sistema de coordenadas descrito por um objeto desta classe.

```
void proj(void);
void proj(TProjPlane*);
```

Chamam métodos homônimos da classe `DCursor`.

```
DCursor* setCursor(DCursor*);
DCursor* getCursor(void);
void draw(DrawMode = show);
void draw(TProjPlane*, DrawMode = show);
void draw(TProjPlane*, TDC&, DrawMode = show);

void move(TTernaFloat);           // coordenadas absolutas
void moveL(TTernaFloat);          // coordenadas locais
TTernaFloat getCursorPos(void);   // coordenadas absolutas
void setCursorPos(TTernaFloat);   // coordenadas absolutas
TTernaFloat getCursorPosL(void);  // coordenadas locais
void setCursorPosL(TTernaFloat);  // coordenadas locais
```

Métodos usados para mudar a posição corrente do cursor associado. As diferenças entre eles são explicadas no código fonte.

```
TTernaFloat local2world(TTernaFloat);
TTernaFloat world2local(TTernaFloat);
```

Funções de conversão entre coordenadas de mundo e coordenadas relativas ao sistema de coordenadas descrito por um objeto desta classe (coordenadas locais ou relativas). Estes métodos utilizam a matriz de mudança de base armazenada no membro `matChangeBase`.

8. Conclusão

Acredita-se que foram atingidos, ao menos parcialmente, os objetivos propostos no início do trabalho, que mostrou-se mais extenso do que a princípio se imaginava. Ainda há muito o que ser desenvolvido, destacando-se os itens seguintes:

1. Projeto e implementação de maior número de operações locais. Apesar de os operadores de Euler implementados permitirem a construção de qualquer sólido poliédrico (vide Apêndice "A"), as operações realizadas pelos mesmos são de nível muito baixo, o que torna o seu uso pelo usuário extremamente difícil. É necessário o desenvolvimento de operações de nível mais alto que facilitem as tarefas do usuário;

2. Apesar de a interface ser gráfica e interativa, como inicialmente proposto, ainda é pouco amigável e deve ser melhorada. O número de planos de projeção (vistas) existentes simultaneamente ainda deve ser ampliado; o acesso aos comandos, por ora apenas através de menus, deve ser facilitado; muitas outras ferramentas ainda devem ser acrescentadas para tornar o uso do software mais fácil e intuitivo.

Além disso, as classes correspondentes aos vários tipos de cursor (*2d*, *3d*), mecanismos de seleção e entrada de coordenadas, sistemas de coordenadas, câmera, etc., deverão ser agrupadas num conjunto de *resources*, os quais poderão ser incorporados a uma linguagem e facilmente transportados para outros sotwares.

3. Devem ainda ser desenvolvidos módulos de comunicação com memória externa e, possivelmente, com outros softwares.

Para finalizar, cabe a observação de que, apesar de ainda haver vários itens a serem projetados e implementados, grande parte da estrutura para suportá-los (operadores de Euler, esquema de representação interno, interface *Windows*, etc.) já foi desenvolvida, o que deverá facilitar sobremaneira o trabalho futuro.

Apêndice A

Observações sobre os operadores de Euler utilizados

Segue-se um resumo sobre operadores de Euler e as modificações introduzidas em sua aplicação no projeto. Maiores detalhes podem ser encontrados na referência [11].

Consideremos o espaço vetorial $\mathbb{R}^6$ sobre $\mathbb{R}$, e os vetores da sua base canônica

$$\vec{V} = (1,0,0,0,0,0)$$
$$\vec{E} = (0,1,0,0,0,0)$$
$$\vec{F} = (0,0,1,0,0,0)$$
$$\vec{H} = (0,0,0,1,0,0)$$
$$\vec{R} = (0,0,0,0,1,0)$$
$$\vec{S} = (0,0,0,0,0,1)$$

O subconjunto do $\mathbb{R}^6$ dado por

$$\vec{P} = v\vec{V} + e\vec{E} + f\vec{F} + h\vec{H} + r\vec{R} + s\vec{S}$$
$$v,e,f,h,r,s \in \Re /$$
$$v - e + f = 2(s - h) + r \qquad (I)$$

é um subespaço vetorial do mesmo, de dimensão 5, como pode ser facilmente verificado.

Consideremos agora o conjunto de todos os sólidos cujas superfícies sejam topologicamente equivalentes a um "*2-manifold*", e cujos modelos planos correspondentes ("*plane models*") tenham característica de Euler χ=2. Designemos por $\boldsymbol{S_F}$ este conjunto (sólidos poliédricos fechados pertencem a ele).

Então, podemos colocar S_F em correspondência biunívuca com um subconjunto discreto do subespaço vetorial (I), obtido pela restrição das coordenadas *v,e,f,h,r* e *s* ao conjunto dos números naturais: a cada modelo plano com

v *vertices*
e *edges*
f *faces*
h *holes*
r *rings*
s *shells*

corresponde o vetor

$$\vec{P} = v\vec{V} + e\vec{E} + f\vec{F} + h\vec{H} + r\vec{R} + s\vec{S}$$

do referido subconjunto.

Sendo B uma base do subespaço (I), a todo sólido de S_F corresponde um único vetor do referido subespaço, dado por uma combinação linear (única, a menos da ordem) dos elementos desta base. Assim, se a cada vetor v desta base fizermos corresponder um operador p sobre o espaço dos modelos planos de S_F, de tal forma que a soma de um vetor com v em (I) e a aplicação de p ao modelo plano correspondente sejam equivalentes, chegaremos um conjunto de operadores através dos quais pode-se construir qualquer modelo plano. Um tal conjunto de operadores é designado de conjunto de *operadores de Euler*.

Geralmente, escolhe-se uma base de (I) formada pelos seguintes vetores (os correspondentes operadores de Euler são indicados ao lado):

Vetor da base						Operador
v	e	f	h	r	s	
1	1	0	0	0	0	MEV
0	1	1	0	0	0	MEF
0	-1	0	0	1	0	KEMR
1	0	1	0	0	1	MVFS
0	0	-1	1	1	0	KFMRH

(II)

Por exemplo, o operador *mev (make edge, vertex)* acrescenta um *edge* e um *vertex* ao modelo plano; o operador *mef (make edge, face)* acrescenta um *edge* e uma *face*, etc. Usa-se a seguinte convenção na nomenclatura dos operadores:

m - make
k - kill
v - vetex
e - edge
f - face
h - hole
r - ring
s - shell

Também são úteis os operadores inversos, que são obtidos quando se toma o simétrico de cada vetor da base (II):

Vetor						Operador
v	e	f	h	r	s	
-1	-1	0	0	0	0	KEV
0	-1	-1	0	0	0	KEF
0	1	0	0	-1	0	MEKR
-1	0	-1	0	0	-1	KVFS
0	0	1	-1	-1	0	MFKRH

No projeto, não foram utilizados operadores de Euler para sólidos do tipo até aqui considerado (i.e., sólidos com superfície topologicamente equivalente a um "*2-manifold*" e com modelos planos com característica de Euler χ=2). Foram usados operadores de Euler para *2-manifold*'s com modelos planos com característica de Euler χ=1. Superfícies poliédricas abertas, obtidas pela remoção de uma face de uma superfície poliédrica fechada, enquadram-se nesta categoria.

Seguindo um raciocínio análogo, o conjunto S_A destes *2-manifold*'s pode ser posto em correspondência biunívuca com um subconjunto discreto do subespaço vetorial de dimensão 5 do $\mathbb{R}^6$ dado por

$$\vec{P} = v\vec{V} + e\vec{E} + f\vec{F} + h\vec{H} + r\vec{R} + s\vec{S}$$

$$v, e, f, h, r, s \in \Re /$$

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

Os operadores de Euler utilizados foram obtidos de maneira análoga:

Vetor da base						Operador
v	e	f	h	r	s	
1	1	0	0	0	0	MEV
0	1	1	0	0	0	MEF
0	-1	0	0	1	0	KEMR
1	0	0	0	0	1	MVS
0	0	-1	1	1	0	KFMRH

(II)

Vetor						Operador
v	e	f	h	r	s	
-1	-1	0	0	0	0	KEV
0	-1	-1	0	0	0	KEF
0	1	0	0	-1	0	MEKR
-1	0	0	0	0	-1	KVS
0	0	1	-1	-1	0	MFKRH

A única modificação introduzida foi no operador *MVFS*, que passou a ser *MVS*, e no correspondente operador inverso.

Apêndice B

Matrizes de Transformação Homogênea

No que segue, serão apresentadas algumas notas sobre matrizes de transformação homogênea. O grande mérito do uso destas matrizes consiste em transformar certas operações como translação em perspectiva em transformações lineares, que podem ser escritas matricialmente e são mais adequadas para tratamento computacional. O assunto pode ser formalizado da seguinte maneira:

Cada ponto **P = (x,y,z)** do $\mathbf{R}^3$ será transformado bijetivamente num subespaço vetorial de dimensão 1 do $\mathbf{R}^4$, designado a partir daqui de *subespaço ampliado de P* $\mathbf{S_P}$, como se segue:

$$(x,y,z) \leftrightarrow \lambda \cdot (x,y,z,1) \quad (I)$$

O conjunto unitário formado pelo vetor $\underline{\mathbf{P}}$ **= (x,y,z,1)** é uma base para $\mathbf{S_P}$; $\underline{\mathbf{P}}$ denomina-se vetor homogêneo associado a **P**.

A seguir, serão definidos os operadores lineares **rotação**, **translação** e **perspectiva** no $\mathbf{R}^4$, de forma que aplicar uma rotação, translação ou perspectiva em um ponto do $\mathbf{R}^3$ seja equivalente a aplicar o correspondente operador **rotação**, **translação** ou **perspectiva** em um vetor qualquer do *subespaço ampliado* deste ponto.

Operador ***translação***

Sendo dado um vetor $\mathbf{t} = (t_x, t_y, t_z)$ do $\mathbf{R}^3$,define-se operador **translação de t**, no $\mathbf{R}^4$, pela seguinte matriz, em relação à base canônica:

$$T_t = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Uma translação de **t** no $\mathbf{R}^3$ pode ser feita por uso deste operador da seguinte maneira:

Seja $\mathbf{P} = (x,y,z)$ um ponto do $\mathbf{R}^3$. Tomemos um vetor qualquer não nulo do seu *subespaço ampliado*, digamos, **(ax, ay, az, a)**, sem perda de generalidade. Aplicando o operador T_t a este vetor, obtemos:

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} ax \\ ay \\ az \\ a \end{bmatrix} = \begin{bmatrix} a(x+t_x) \\ a(y+t_y) \\ a(z+t_z) \\ a \end{bmatrix}$$

Mas $\mathbf{(a(x+t_x) , a(y+t_y) , a(z+t_z) , a)}$ é um vetor do subespaço $\lambda(x+t_x, y+t_y, z+t_z, 1)$, que, por (I), é o *subespaço ampliado* de $(x+t_x, y+t_y, z+t_z)$.

Para simplificar os cálculos, normalmente se toma o vetor homogêneo associado a **P**, apesar de ser possível escolher qualquer vetor do correspondente *subespaço ampliado*, como foi mostrado.

Note que a operação *translação* no $\mathbf{R}^3$, não-linear, foi transformada num operador linear no $\mathbf{R}^4$.

Operadores de ***rotação***

Os operadores de **rotação**, no $\mathbf{R}^4$, são definidos pelas seguintes matrizes, em relação à base canônica:

- rotação em torno de *x*:

$$Rot_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\operatorname{sen}\theta & 0 \\ 0 & \operatorname{sen}\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- rotação em torno de *y*:

$$Rot_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \operatorname{sen}\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\operatorname{sen}\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- rotação em torno de *z*:

$$Rot_z(\theta) = \begin{bmatrix} \cos\theta & -\operatorname{sen}\theta & 0 & 0 \\ \operatorname{sen}\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Para aplicar uma rotação em um ponto **P** do $\mathbf{R}^3$, o procedimento é análogo ao do caso anterior: toma-se um vetor qualquer do correspondente *subespaço ampliado* (para facilitar, escolhe-se o vetor homogêneo $\underline{\mathbf{P}}$), aplica-se o operador de rotação no $\mathbf{R}^4$, e obtem-se o resultado no $\mathbf{R}^3$ utilizando (I).

Operador ***escalonamento***

Uma mudança de escala nos eixos x, y e z, no $\mathbf{R}^3$, segundo os fatores s_x , s_y e s_z , respectivamente, pode ser feita através do operador **escalonamento**, no $\mathbf{R}^4$, definido pela seguinte matriz em relação à base canônica:

$$S(\vec{s}) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Operador ***perspectiva***

Dado um número real não nulo **d**, define-se o operador **perspectiva** pela seguinte matriz, em relação à base canônica do $\mathbf{R}^4$:

$$Pers(d) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -1/d & 1 \end{bmatrix}$$

Seja **P = (x, y, z)** um ponto do $\mathbf{R}^3$ e **P** = **(x, y, z, 1)** o respectivo vetor homogêneo. Então:

$$Pers(d) \cdot \vec{P} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -1/d & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 0 \\ 1 - z/d \end{bmatrix}$$

Através de (I), obtém-se o correspondente resultado no $\mathbf{R}^3$:

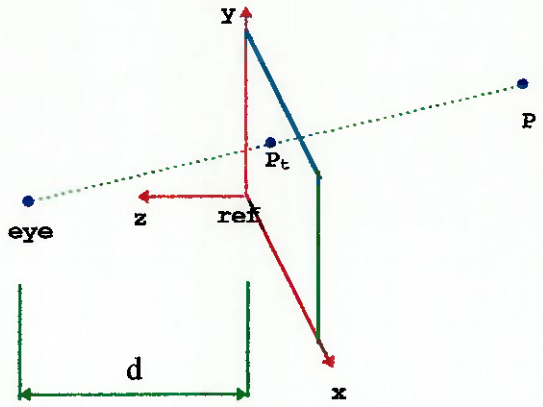
$$P_t = \left(\frac{x}{1 - z/d}, \frac{y}{1 - z/d}, 0 \right)$$

Desta forma, a *transformação perspectiva*, não-linear no $\mathbf{R}^3$, dada por

$$Tp: \Re^3 \rightarrow \Re^3$$

$$(x,y,z) \mapsto \left(\frac{x}{1-z/d}, \frac{y}{1-z/d}, 0 \right)$$

foi transformada num operador linear no $\mathbf{R}^4$.



Resumindo, para aplicar-se uma transformação de **translação**, **rotação**, **escalonamento** ou **perspectiva**, num ponto **P** do $\mathbf{R}^3$, pelo uso de matrizes de transformação homogênea, deve-se seguir os seguintes passos:

1. obter o vetor homogêneo $\underline{\mathbf{P}}$ (ou qualquer outro ponto do *subespaço ampliado* correspondente);

2. aplicar o operador linear **translação**, **rotação**, **escalonamento** ou **perspectiva** no $\mathbf{R}^4$;

3. utilizando (I), obter o resultado correspondente no $\mathbf{R}^3$.

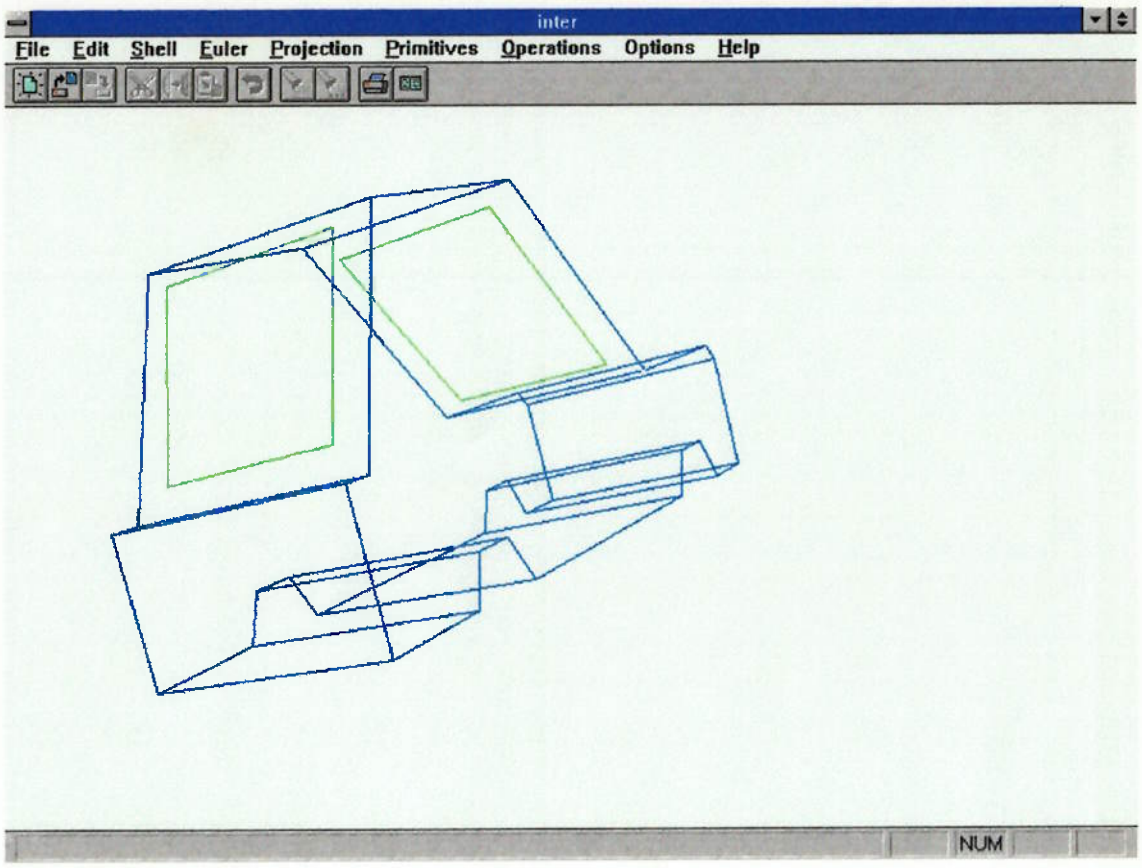
A grande vantagem deste método consiste em tornar transformações não-lineares no $\mathbf{R}^3$ em transformações lineares no $\mathbf{R}^4$. Além de isto ser mais adequado para implementação computacional, uma vez que transformações lineares podem ser escritas sob a forma de multiplicação de matrizes, também facilita os cálculos sobremaneira quando se trata de efetuar composição de transformações.

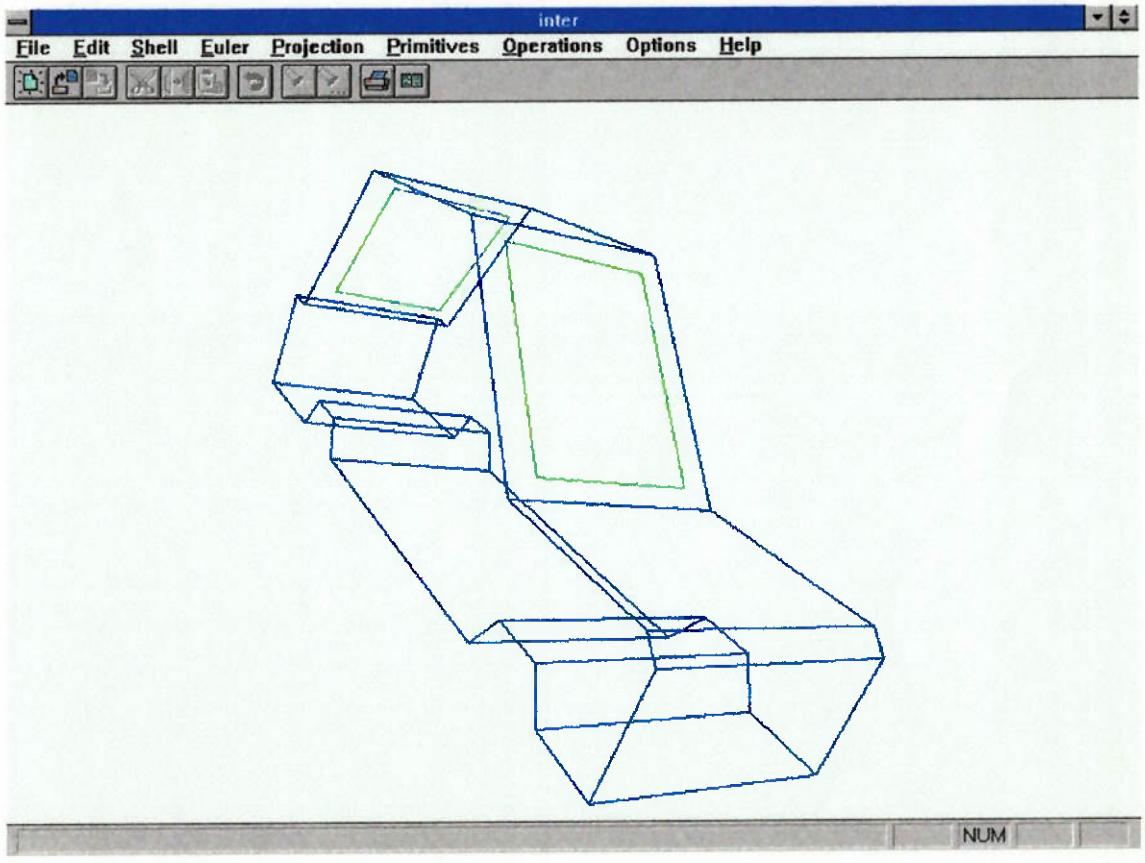
Ora, como sabemos da Álgebra Linear, a matriz de uma transformação linear composta é dada pelo produto das matrizes de cada transformação - isto significa que para efetuar uma composição de transformações, é necessário apenas efetuar uma multiplicação de matrizes.

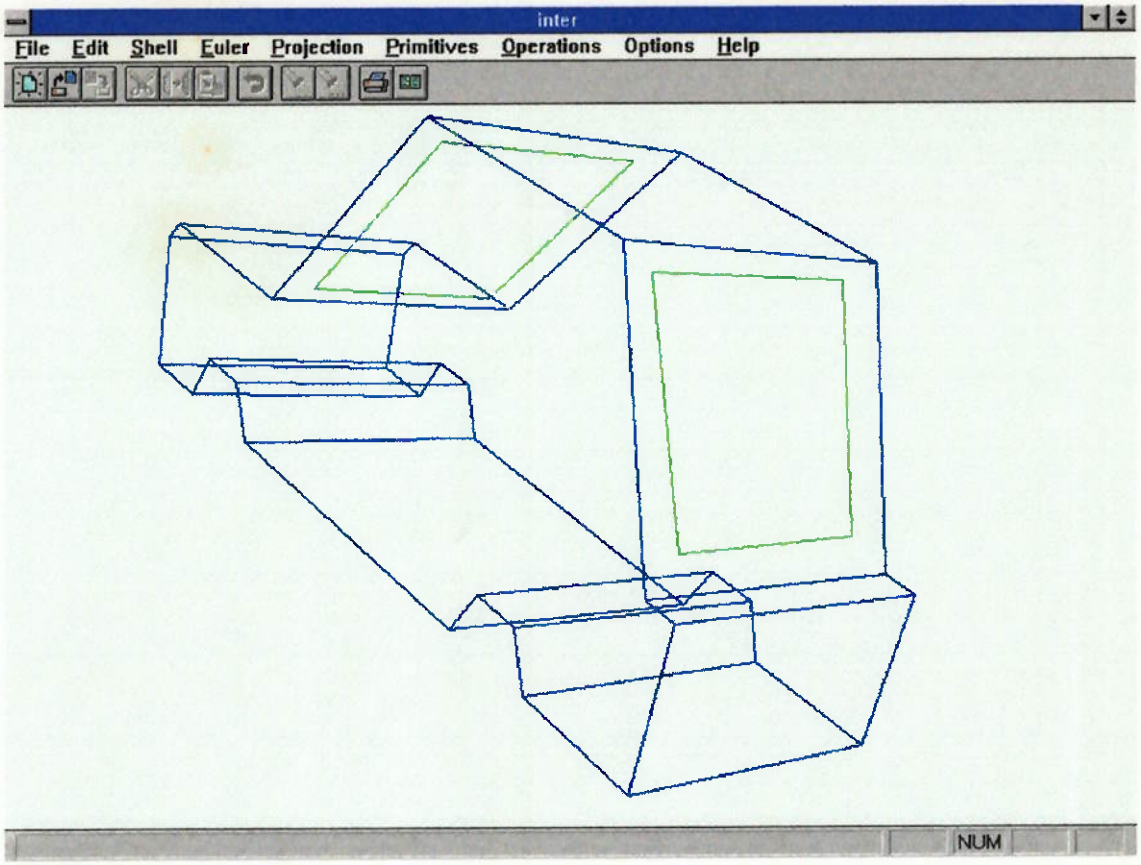
Apêndice C

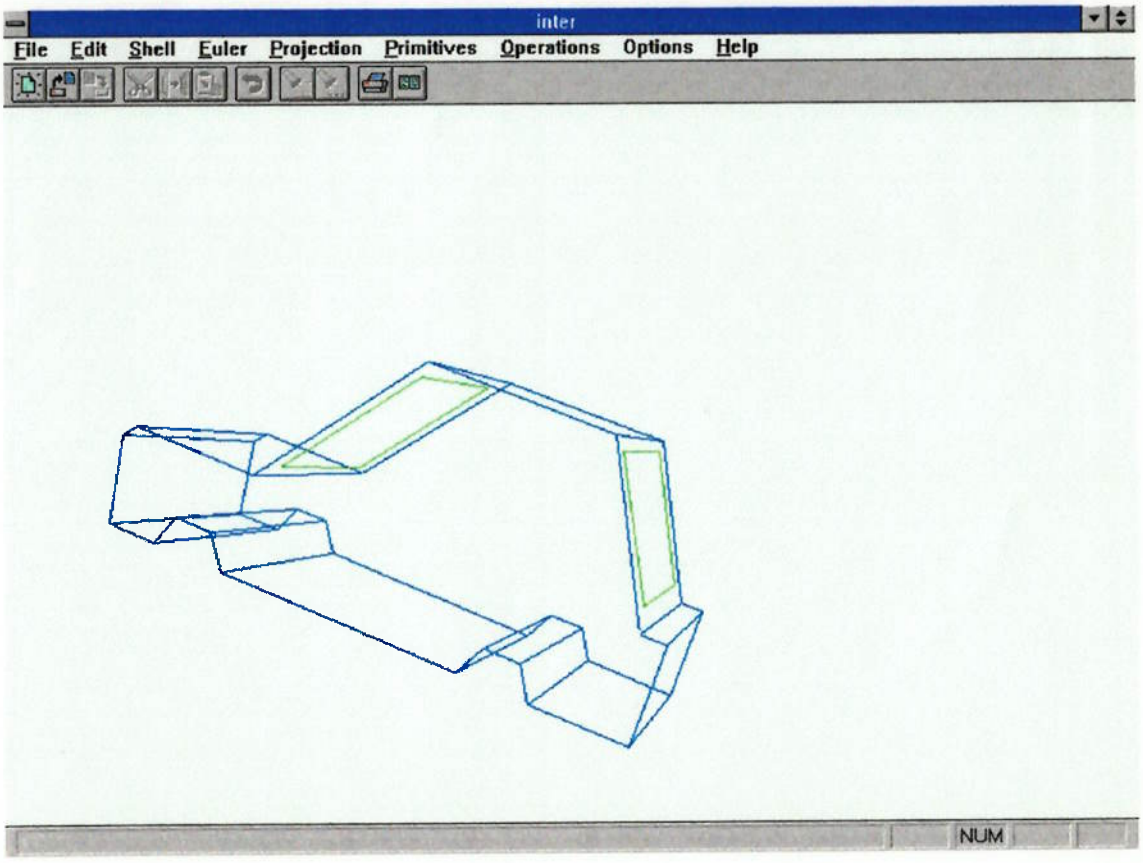
Estudos de Caso

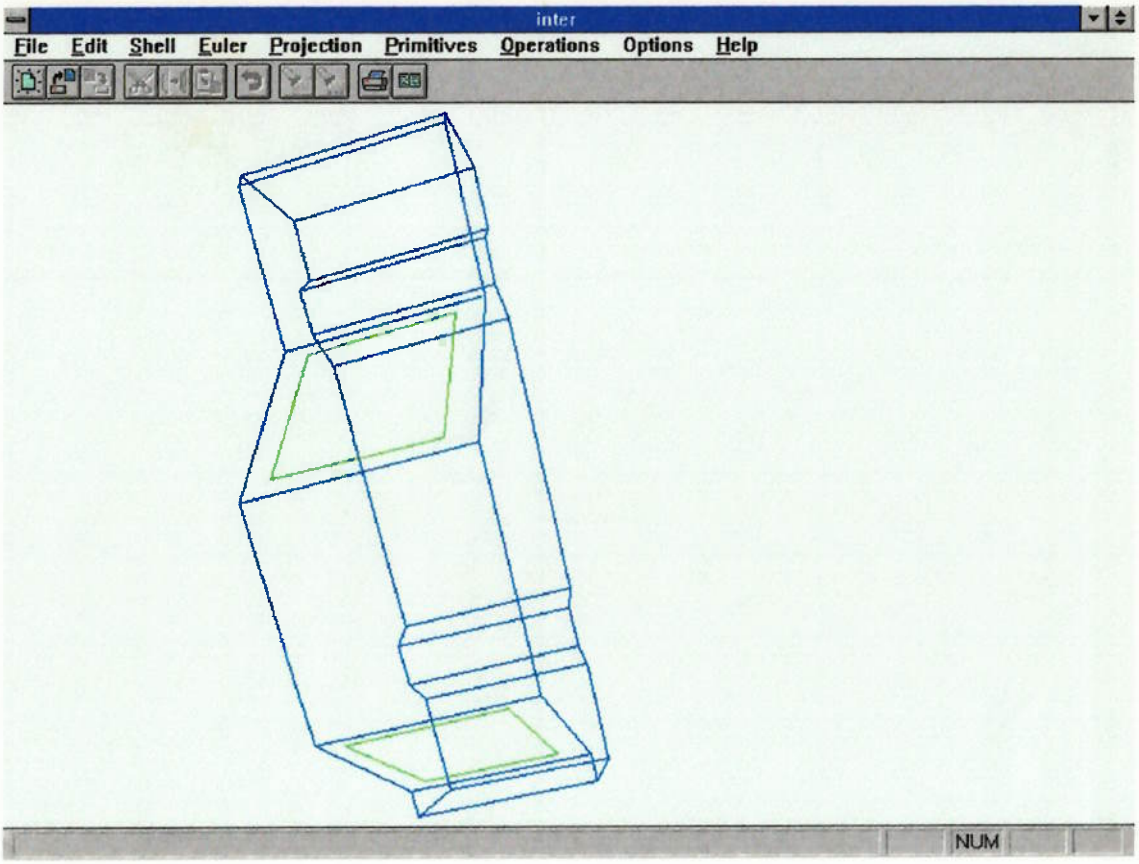
No que segue, serão apresentados alguns modelos sólidos construídos através da interface. Note que, apesar da simplicidade geométrica dos objetos modelados, seria difícil construir-se um modelo dos mesmos sem usar operações locais (i.e., por operações *booleanas*).

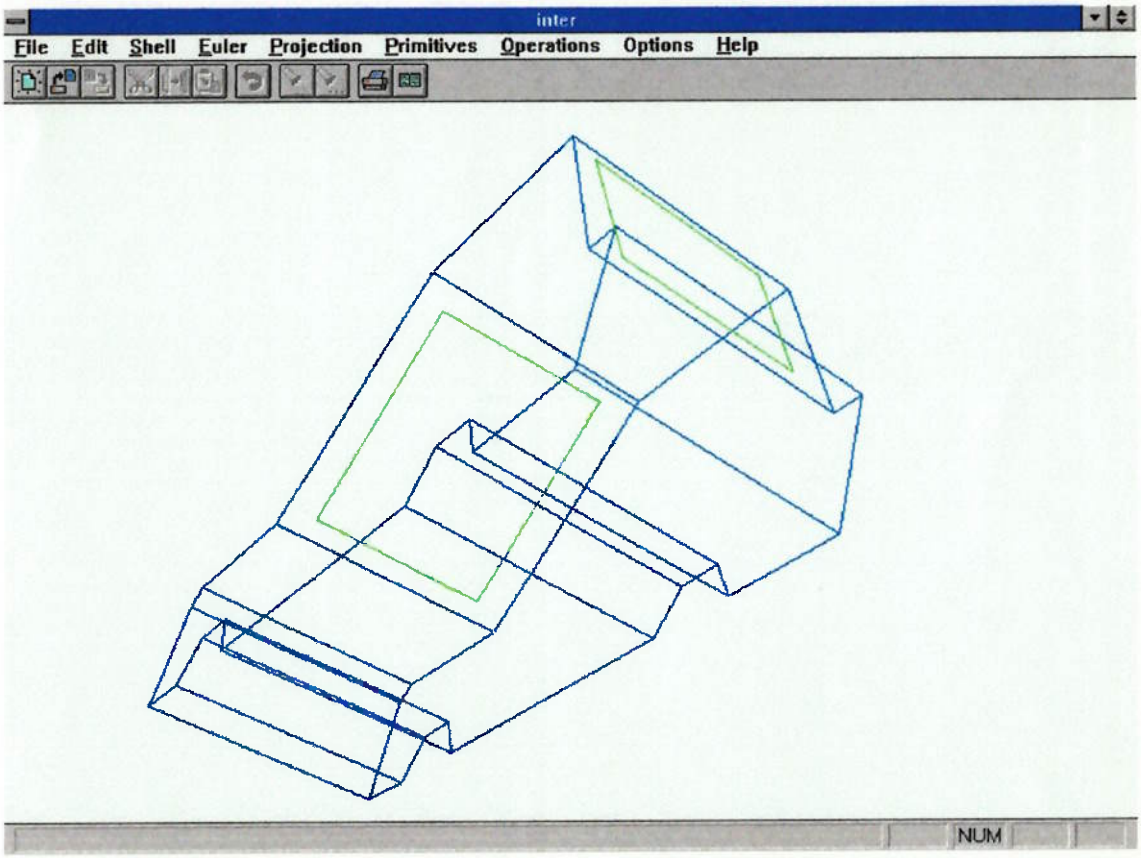

inter
File Edit Shell Euler Projection Primitives Operations Options Help
NUM

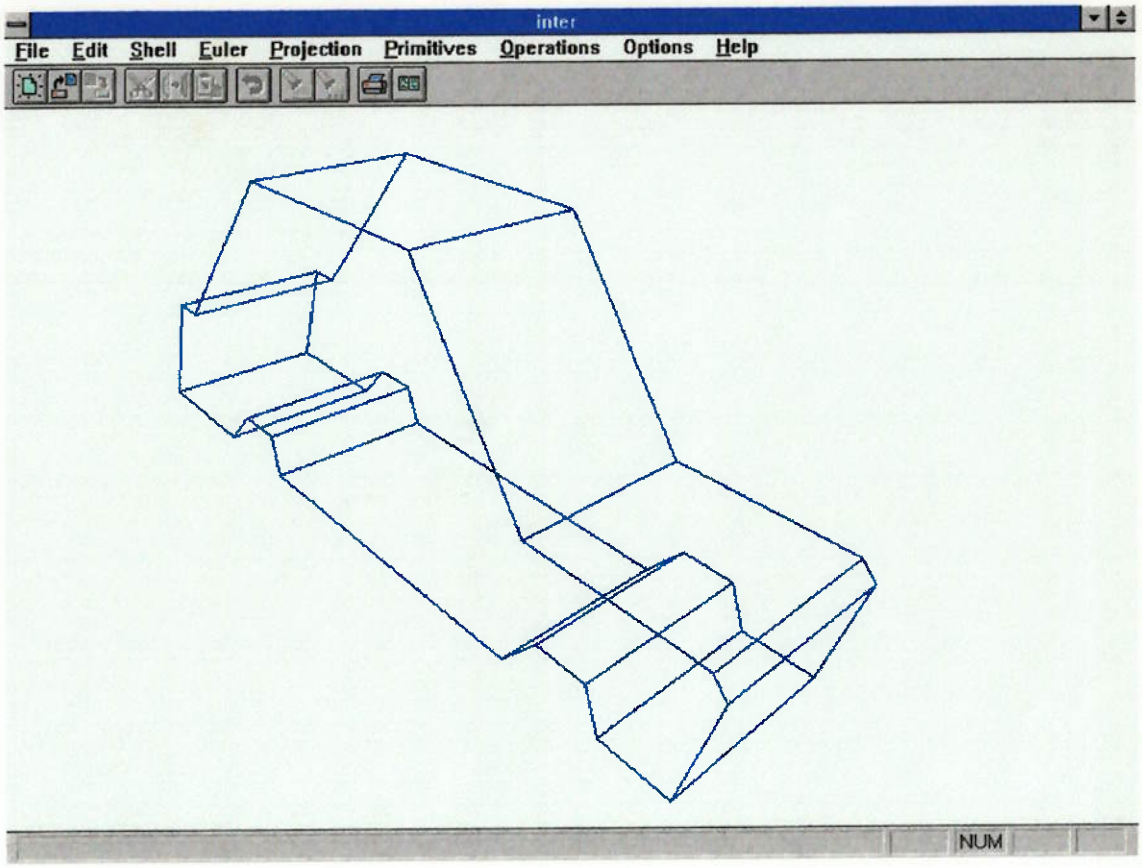

inter
File Edit Shell Euler Projection Primitives Operations Options Help
NUM

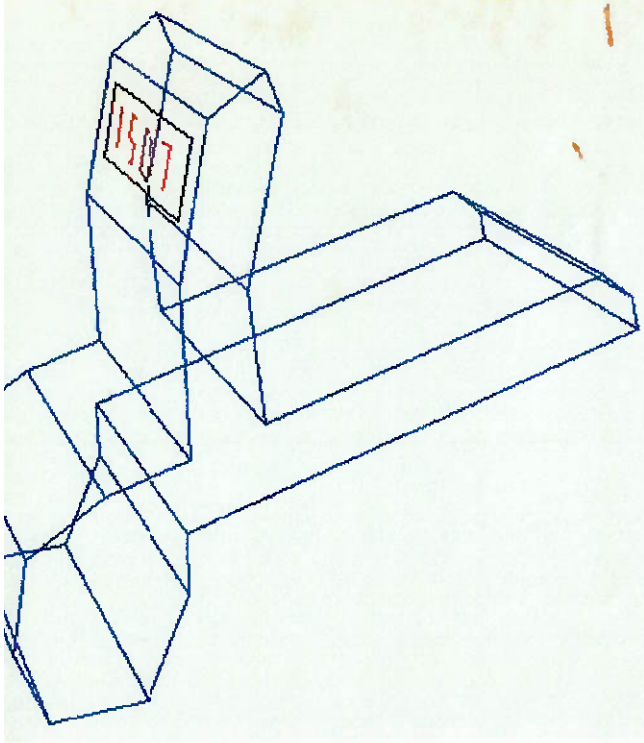

inter
File Edit Shell Euler Projection Primitives Operations Options Help
NUM

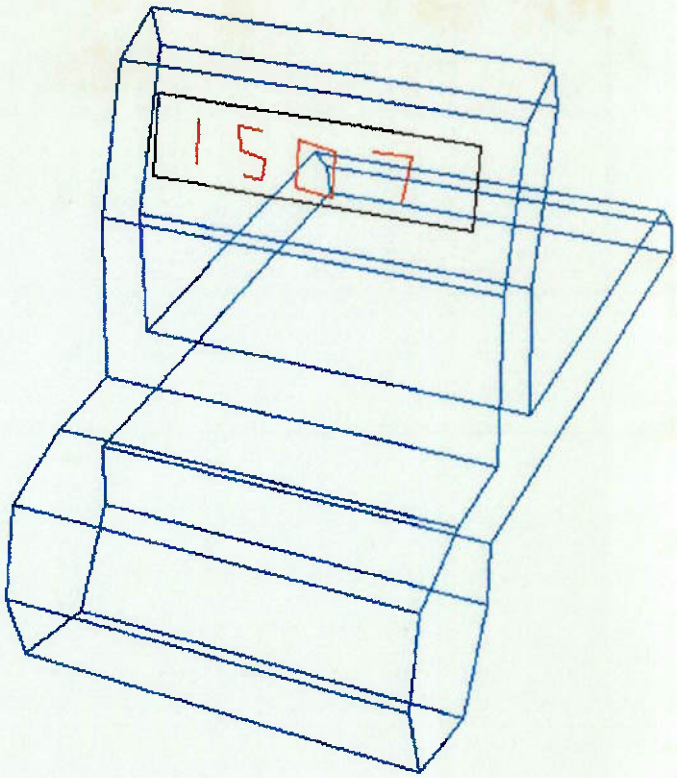

inter
File Edit Shell Euler Projection Primitives Operations Options Help
NUM

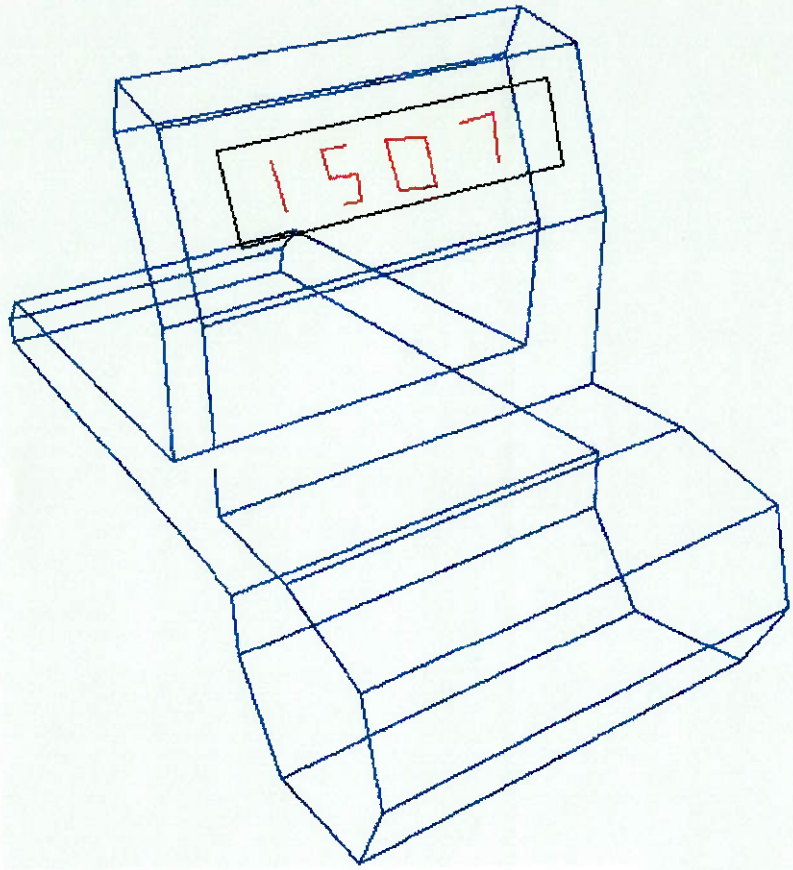

inter
File Edit Shell Euler Projection Primitives Operations Options Help
NUM

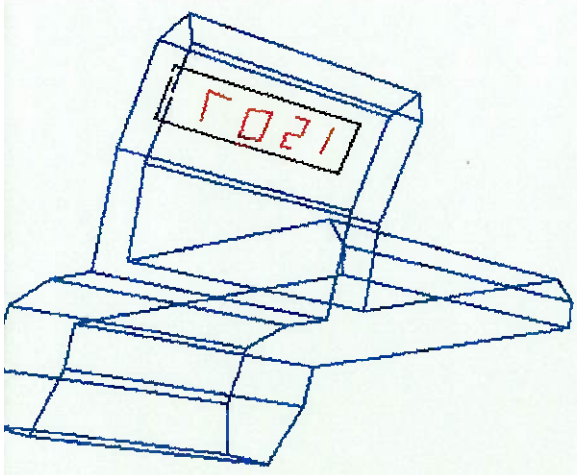

inter
File Edit Shell Euler Projection Primitives Operations Options Help
NUM

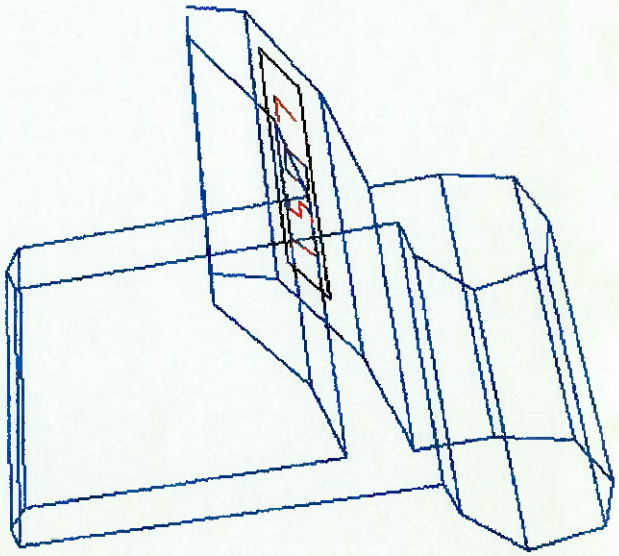

inter
File Edit Shell Euler Projection Primitives Operations Options Help
NUM

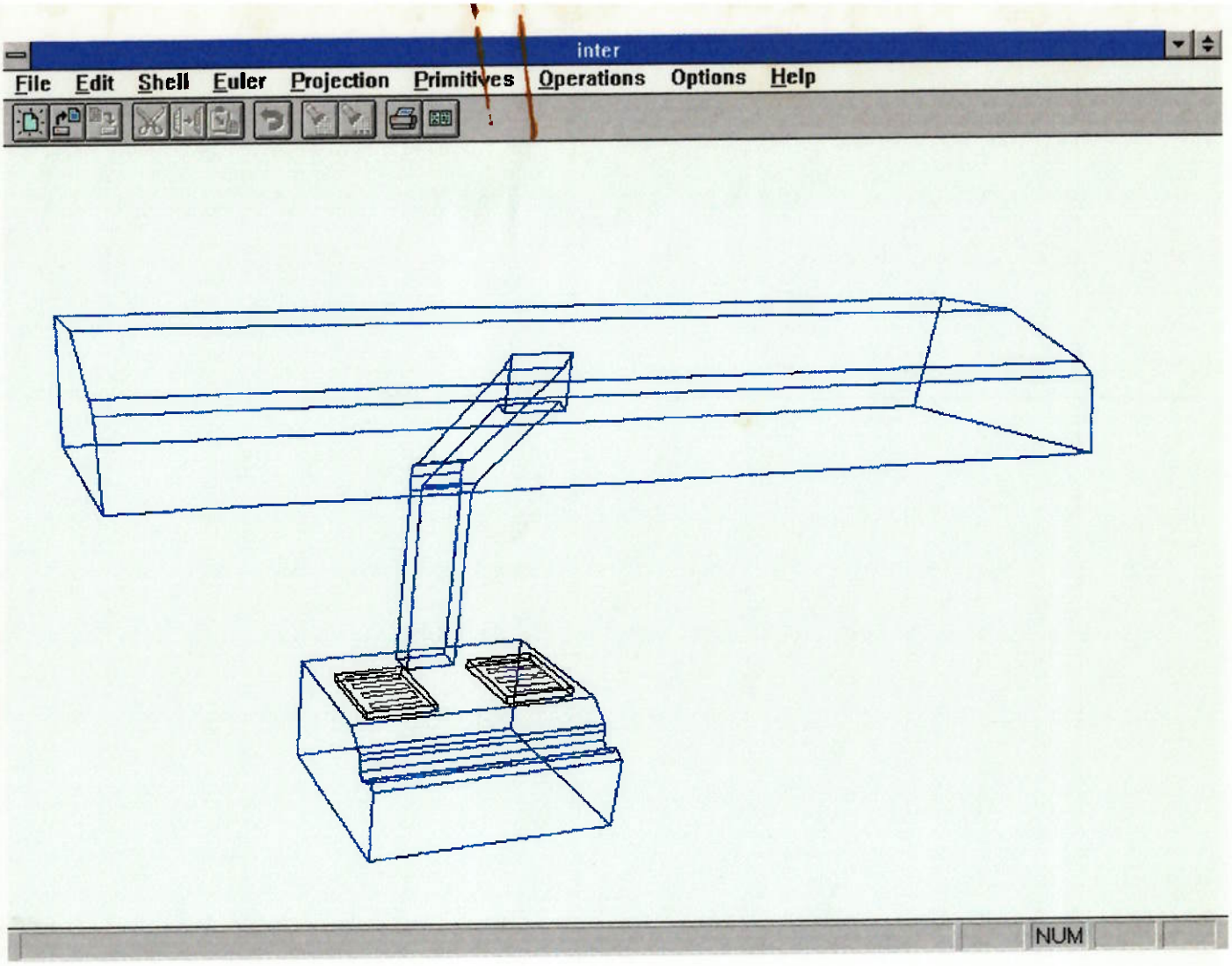

1507

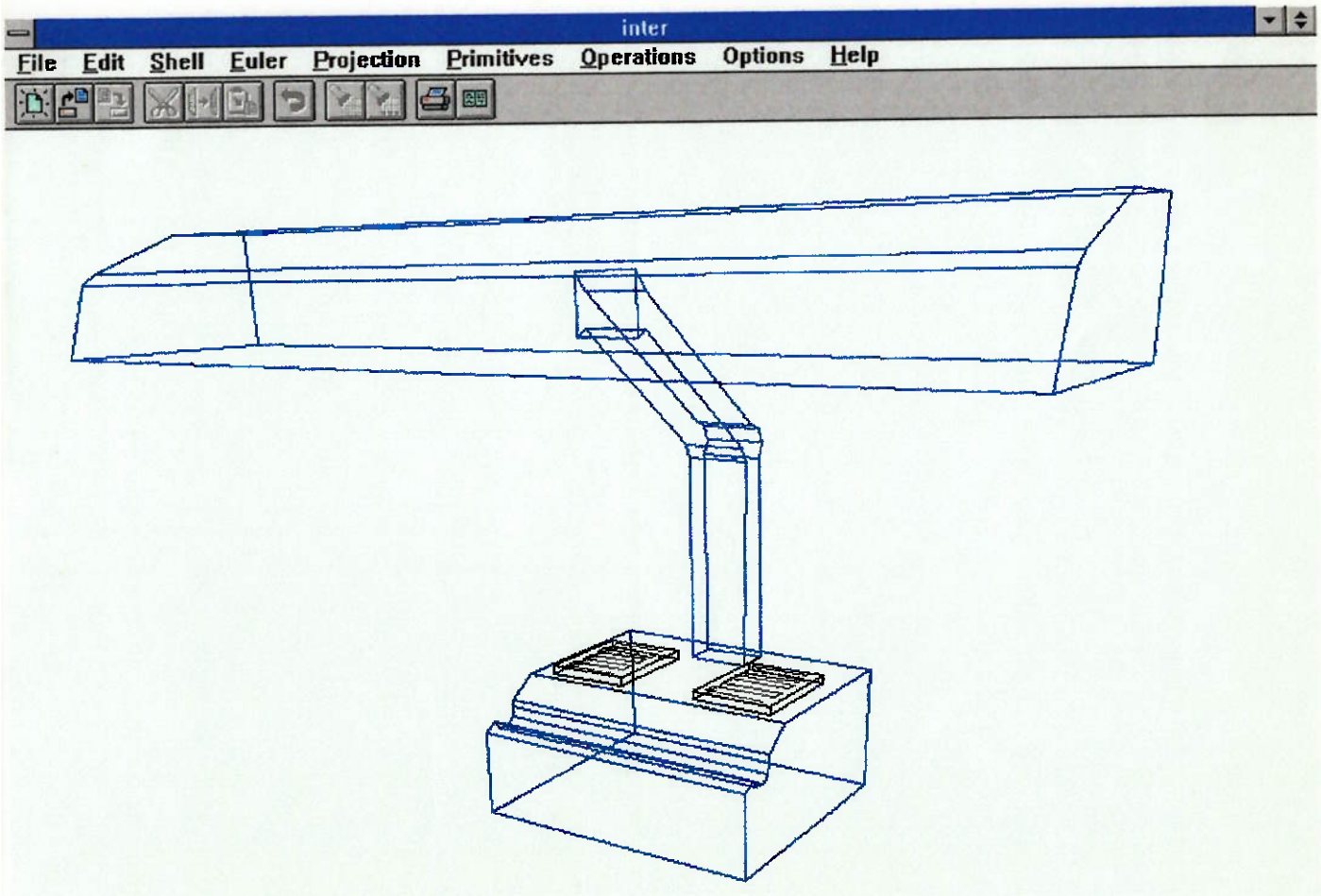

1507

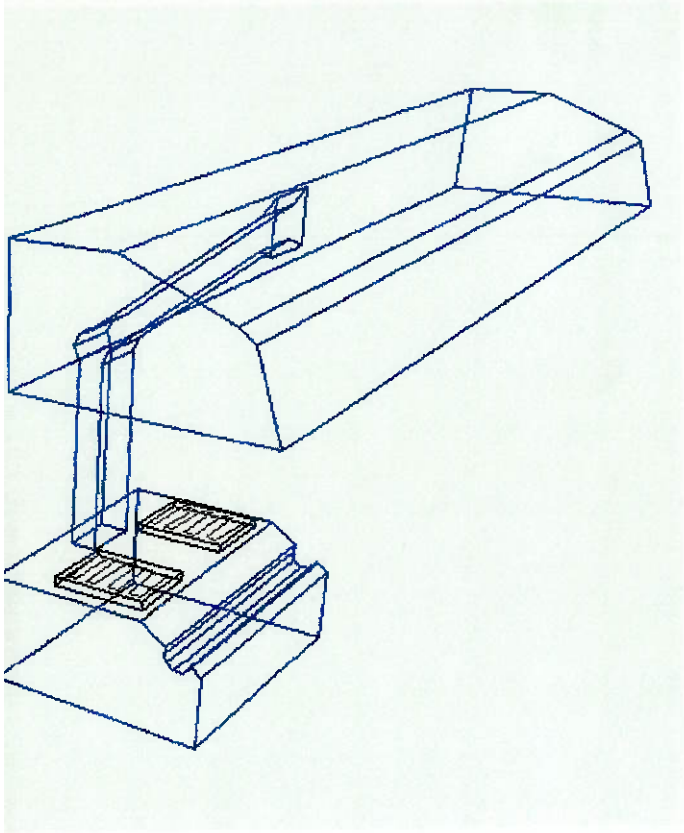

1507

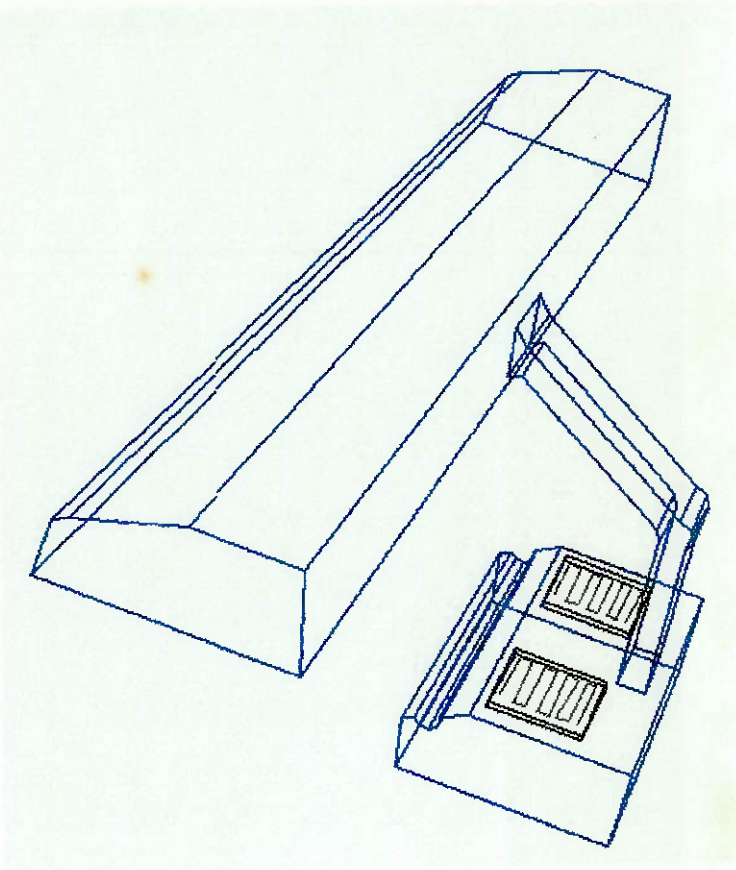

1021

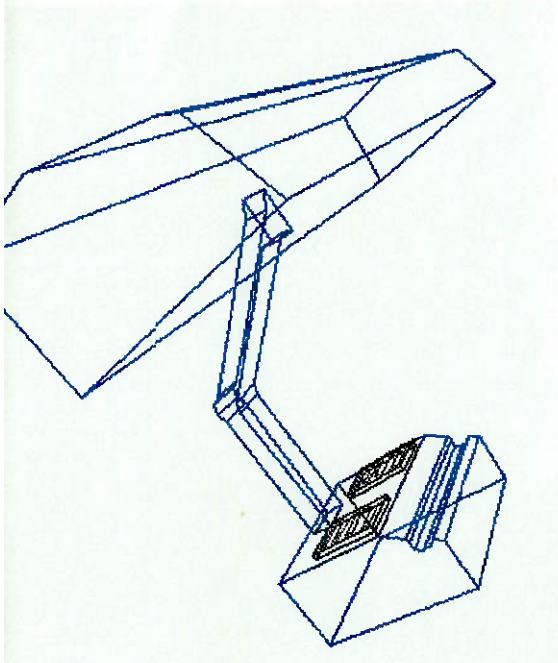


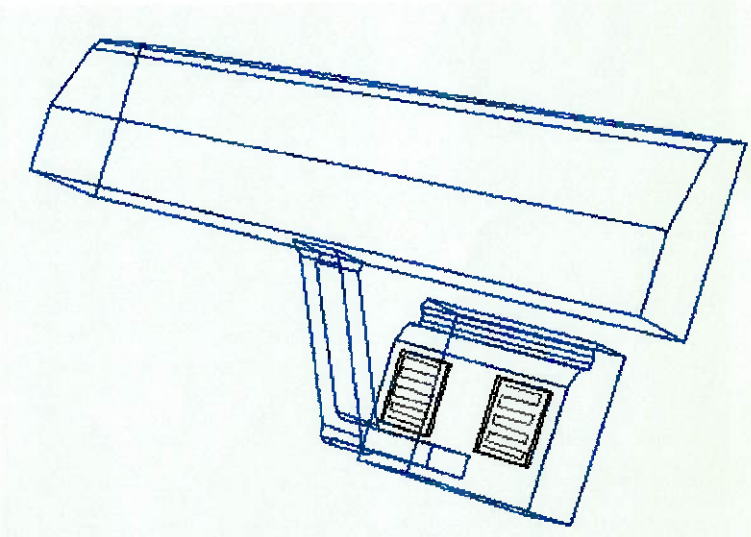

inter
File Edit Shell Euler Projection Primitives Operations Options Help
NUM


inter
File Edit Shell Euler Projection Primitives Operations Options Help









9. Referências Bibliográficas

1 Ritchie, Dennis M. & Kernighan, Brian W. **C, A Linguagem de Programação Padrão ANSI**.

2 Potts, Stephen & Monk, Timothys. **Borland C++ 4.0**. Axcel Books, 1994.

3 Swan, Tom. **Aprendendo C++**. Ed Campus, 1993.

4 Granero, A. F. & Siqueira, J. O. . **Programação Orientada para Objeto em C++ no Ambiente Windows™.** São Paulo, Atlas, 1995.

5 Yao, Paul. **Borland C++ 4.0: programação for Windows™.** São Paulo, Makron Books, 1995.

6 Heiny, Loren. **Windows™ Graphics Programming with Borland C++.** John Wiley & Sons, 1994.

7 Taylor, Philip. **3D Graphics Programming in Windows™.** Addison Wesley, 1994.

8 **Borland Object Windows for C++ 2.0 - Programmer's Guide.** Borland International Inc, 1993.

9 **Borland Object Windows for C++ 2.0 - Reference Guide.** Borland International Inc, 1993.

10 **Borland C++ 4.0 - Library Reference.** Borland International Inc, 1993.

11 Mäntilä, Martti. **An Introduction to Solid Modeling.** Computer Science Press, 1988.