INSTITUTO MAUÁ DE TECNOLOGIA
MAUÁ
■ Centro Universitário
• Escola de Engenharia Mauá
• Escola de Administração Mauá

Introdução à Biblioteca Gráfica

Roberto Scalco

2004 / 2005

Autor

Roberto Scalco

- Mestrando em Engenharia Elétrica pela Universidade Estadual de Campinas (UNICAMP);
- Engenheiro Eletricista com ênfase em Eletrônica pela Escola de Engenharia Mauá (EEM).
  roberto.scalco@maua.br

Software

Delphi ® é marca registrada da Borland®

é padronizado e controlado pelo consórcio ARB (*Architecture Review Board*)

Agradecimentos

Professor Vitor Alex Oliveira Alves

Índice

Lista de Figuras

Lista de Tabelas

1. Introdução

A Computação Gráfica tornou-se mundialmente conhecida devido às indústrias cinematográficas e de entretenimento em geral. Desde os efeitos especiais mais simples desenvolvidos até meados do século passado, até a criação de filmes sem a utilização de atores fizeram com que a Computação Gráfica (ou seus resultados) fossem considerados um show a parte da interpretação dos atores.

O mesmo ocorreu com a evolução dos processadores dos videogames. Os personagens que eram representados apenas por "quadradinhos" tornaram-se mais complexos, chegando até a possuir expressões faciais. Novos efeitos foram adicionados, permitindo que a iluminação e a geração de sombras pudessem ser processadas em tempo real.

Não foram apenas sonhos que incentivaram as pesquisas nesta área. Com os avanços dos recursos médicos novos equipamentos foram desenvolvidos e, consequentemente, algumas técnicas para a geração e tratamento de imagens encontraram outro campo de atuação. Alguns exemplos consistem nos resultados de um exame de ressonância magnética ou topografia da córnea.

Outro uso intensivo da Computação Gráfica está ligado à Visualização Cientifica, ou seja a geração de simulações gráficas das mais diversas áreas da Ciência. Estas simulações são amplamente utilizadas em Mecânica dos Fluídos, Resistência dos Materiais, Eletromagnetismos entre outras.

Para a criação dos efeitos especiais são necessários aplicativos específicos (e caros). Não é o intuito do curso "reinventar a roda"!

Serão apresentadas ferramentas que permitam a criação de aplicativos que manipulem objetos em espaços bidimensionais e tridimensionais. Os aplicativos que serão desenvolvidos possuem característica didática, ou seja, representam uma parte de um problema maior (e mais complexo) que deve ser solucionada.

Para permitir com que os conceitos teóricos sejam aplicados algumas instruções da biblioteca gráfica OpenGL serão apresentadas. Esta biblioteca permite manipular elementos gráficos como vértices, arestas e faces, além de permitir com que sejam definidas as características do observador e de fontes de iluminação existentes na cena, permitindo gerar imagens foto-realísticas.

Por tratar-se de um conjunto de instruções específicas associadas à linguagem de programação, torna-se necessário que seja dedicado algum tempo de estudo em frente ao computador. Como motivação, a criação dos ambientes e seu conteúdo são limitados apenas pela criatividade do programador.

2. Descrição das bibliotecas

2.1. Introdução

Serão apresentadas algumas bibliotecas contendo um conjunto de sub-rotinas que permitirão o uso dos recursos disponíveis das placas gráficas.
Em 1992, a **Silicon Graphics, Inc.** (SGI) criou uma biblioteca para desenvolvimento de aplicativos gráficos 2D e 3D. Esta biblioteca, encapsulada como uma API (*application programming interface*), permite que os recursos do *hardware* gráfico disponível sejam utilizados da mesma maneira por programadores de diversas linguagens.

2.2. OpenGL (*Open Graphics Library*)

OpenGL não é uma linguagem de programação, mas um conjunto de instruções que permite que seja estabelecida uma interface entre o aplicativo desenvolvido e o processador da placa gráfica.
Existem aproximadamente 150 instruções disponíveis que permitem criar elementos primitivos (como vértices, arestas e faces poligonais), realizar transformações geométricas, aplicar efeitos de iluminação e textura, criar animações, processamento de imagem, entre outros.
A *OpenGL Architecture Review Board* (*ARB*) é um consórcio criado em 1992 para administrar o uso e padronização da biblioteca.
Importante: as instruções são iniciadas por **`gl`**, enquanto que as constantes começam por **`GL_`**.
A tabela a seguir mostra as datas de lançamento de cada versão da biblioteca:

Tabela 2.1: Data de lançamento das versões da biblioteca OpenGL

Versão	Data de lançamento
OpenGL v1.0	01.07.1992
OpenGL v1.1	16.05.1995
OpenGL v1.2	16.03.1998
OpenGL v1.2.1	14.10.1998
OpenGL v1.3	14.08.2001
OpenGL v1.4	24.07.2002
OpenGL v1.5	29.07.2003
OpenGL v2.0	07.09.2004

2.3. GLU (*OpenGL Utility Library*)

A GLU é uma biblioteca que acompanha todas as versões da OpenGL. Contém aproximadamente 50 sub-rotinas que utilizam os comandos OpenGL de baixo nível para executar tarefas como configurar as matrizes para projeção, aplicar a tonalização sobre uma superfície ou criar superfícies quádricas e NURBS (*Non Uniform Rational B-Splines*).
Suas sub-rotinas utilizam o prefixo **`glu`**. Nas aplicações desenvolvidas serão utilizadas apenas duas instruções desta biblioteca: **`gluPerspective`** e **`gluLookAt`**.

2.4. GLUT (*OpenGL Utility Toolkit*)

A terceira biblioteca representa um conjunto de ferramentas independentes, uma vez que ao contrário da GLU, não está associada diretamente aos recursos da biblioteca OpenGL. O seu principal objetivo é permitir o uso dos diferentes sistemas de janelas nos aplicativos, além do controle de eventos como, por exemplo, o suporte a *joysticks*. As funções desta biblioteca usam o prefixo **`glut`**.
Como os aplicativos serão desenvolvidos em Delphi, não é necessário que o programador preocupe-se em criar e configurar as janelas com o GLUT.

3. Ambiente de desenvolvimento

3.1. Introdução

Será apresentada uma estrutura do código fonte que permitirá, quando instruções da biblioteca forem utilizadas, que a imagem seja exibida nos aplicativos. Como o escopo da disciplina não inclui a análise detalhada das configurações dos *pixels* ou tratamento de erro no caso de uma placa gráfica não aceitar a biblioteca, esta seção será descrita de maneira breve e sucinta.

3.2. Delphi

Da mesma maneira que o Delphi pode produzir imagens com o auxílio do Canvas, é permitido que imagens foto-realistas sejam criadas com as instruções da biblioteca OpenGL. Para utilizar esta biblioteca, deve-se incluí-la na lista de bibliotecas que serão utilizadas pelo aplicativo em desenvolvimento:

```
uses OpenGL;
```

Desta maneira, as instruções pertencentes à OpenGL e a GLU estarão disponíveis para que sejam utilizadas em comum harmonia com as instruções da linguagem **Object Pascal**.

3.3. Configurações iniciais

Para permitir que a biblioteca OpenGL funcione em um aplicativo desenvolvido em Delphi é necessário que algumas configurações sejam realizadas, alterando as características associadas às estruturas do *pixels*. Estas configurações devem ser realizadas assim que o programa for criado:

```
procedure TForm1.FormCreate(Sender: TObject);
var pfd: TPixelFormatDescriptor;
    FormatIndex: integer;
begin
fillchar(pfd,SizeOf(pfd),0);
{configura as características do formato dos pixels}
with pfd do
  begin
  nSize := SizeOf(pfd);             {define o tamanho da estrutura}
  nVersion := 1;                    {a versão atual do descritor é 1}
  dwFlags := PFD_DRAW_TO_WINDOW;
  iPixelType := PFD_TYPE_RGBA;
  cColorBits := 24;                 {suporta 24 bits de cores}
  cDepthBits := 32;                 {32 bits de profundidade do eixo Z}
  iLayerType := PFD_MAIN_PLANE;
  end;
glDC := getDC(handle);           {Associa o Device Context com o glDC}
FormatIndex := ChoosePixelFormat(Canvas.Handle,@pfd);     {mapeamento}

{ajusta as características dos pixels seguindo as configurações acima}
SetPixelFormat(glDC,FormatIndex,@pfd);

{permite que as configurações dos pixels sejam aplicadas ao glDC}
GLContext:=wglCreateContext(glDC);
wglMakeCurrent(glDC,GLContext);
end;
```

Como pode ser observado, é necessário definir as variáveis globais **glDC** e **GLContext**.

```
var
  Form1: TForm1;
  glDC: HDC;
  GLContext: HGLRC;
```

Para encerrar o aplicativo é necessário apagar os contextos utilizados.

```
{os comandos a seguir servem para finalizar as ferramentas utilizadas}
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
{o valor ZERO, indica que este contexto não será mais utilizado}
wglMakeCurrent(glDC,0);

{após isso é permitido excluir este contexto}
wglDeleteContext(GLContext);
end;
```

Deve-se notar que em momento algum foram utilizadas instruções pertencentes à OpenGL.

3.4. Componente OGLPanel

É possível utilizar componentes que encapsulam as instruções de configuração da biblioteca. Um destes componentes foi criado por Mustafa Kasap e é distribuído gratuitamente: o OGLPanel. Caso esteja instalado no ambiente Delphi as configurações apresentadas no item 3.3 não são necessárias.
Visualmente o componente representa uma janela sobre o formulário onde são desenhadas as primitivas da biblioteca OpenGL.
Um ponto importante consiste na diferença entre o componente e a biblioteca OpenGL. Como apresentado anteriormente, as instruções para configuração não pertencem à OpenGL. Desta maneira, também é necessário declarar o uso da biblioteca OpenGL:

```
uses OpenGL, OGLPanel;
```

4. Estrutura da biblioteca OpenGL

4.1. Introdução

Antes que sejam apresentadas as instruções da biblioteca, é necessário conhecer sua arquitetura básica, além de algumas características e peculiaridades.
Nesta seção, será apresentada a seqüência de operações para o processamento dos vértices e *pixels*, bem como conceitos da máquina de estados utilizada e o padrão da sintaxe das instruções.

4.2. *Rendering pipeline*

O diagrama a seguir mostra a seqüência de operações necessárias para que os dados sejam processados. Embora o diagrama seja uma representação simplificada, é possível compreender quais serão os passos necessários para que o programador possa manipular corretamente os dados vetoriais e digitais.
Cada um dos processos será descrito a seguir:

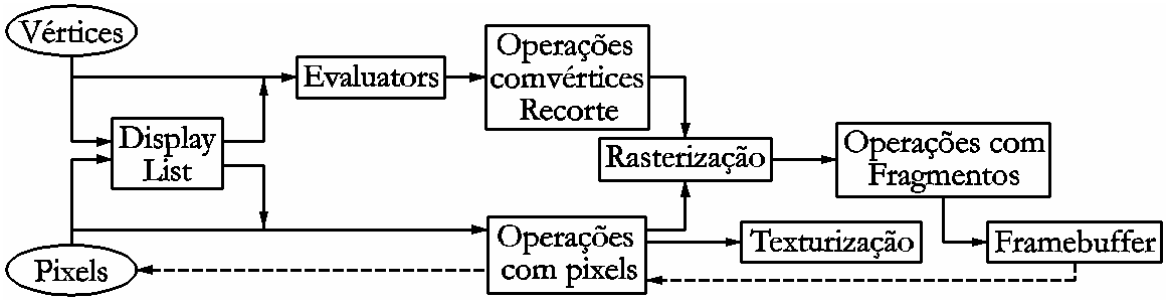


Figura 4.1: *Rendering pipeline*

- **Vértices:** representam os dados geométricos, como os vértices, arestas e faces poligonais;
- ***Pixels*:** representam os dados digitais, ou seja, o conjunto de elementos de uma imagem;
- ***Display List*:** conjunto de instruções pré-definidas que podem ser executadas a qualquer momento. Uma vez que as instruções sejam definidas, não podem mais ser alteradas;
- ***Evaluators*:** Habilita o uso das funções de base para que as curvas e superfícies possam ser definidas a partir de um conjunto de pontos de controle;
- **Operações com vértices:** Converte os vértices para a forma matricial, além de projetá-los para o plano da tela (coordenadas da imagem);
- **Recorte:** Define a janela limitante dos objetos projetados no plano da tela;
- **Operações com *pixels*:** Os *pixels* provenientes da memória sofrem processamento, como mudança de escala, mapeamento em um espaço de textura e outros processamentos diversos. Após o processamento, os *pixels* são novamente armazenados na memória;
- **Texturização:** Quando habilitada, permite que a aplicação da textura tenha prioridade no processamento do *hardware*;
- **Rasterização:** Conversão dos vetores e *pixels* em fragmentos. Cada fragmento representa um *pixel* do *framebuffer*, considerando a sua resolução espacial;
- **Operações com fragmentos:** Permite a aplicação dos *texels*[1] sobre os fragmentos; calcula efeitos como neblina e transparência, além da aplicação de algoritmos de visibilidade;
- ***Framebuffer*:** Conjunto de *pixels* que estão aptos a serem exibidos, após a realização de alguns testes.

[1] *Texture element*

4.3. *Framebuffer*

Como mencionado, o *framebuffer* aplica alguns testes aos *pixels* antes que estes possam ser exibidos. O *framebuffer* compreende os *buffers* de cor, profundidade, seleção e acumulação.

- ***Buffer* de cor (*Color Buffer*):** Composto por quatro *buffers*, dois referentes à visão estérea (direita e esquerda) e dois referentes ao sistema *double-buffer* (frente e trás), que processa uma imagem enquanto outra está sendo exibida. Esta técnica permite que as animações sejam exibidas de maneira contínua, ou seja, a imagem não "pisca";
  Para garantir que o *double-buffer* funcione normalmente, deve-se informar ao *pixel format descriptor* (*pfd*), nas configurações iniciais, utilizando a linha a seguir:

  ```
  dwFlags := PFD_DRAW_TO_WINDOW or PFD_DOUBLEBUFFER;
  ```

  Quando o componente **OGLPanel** for utilizado, deve-se atribuir o valor **TRUE** à característica **DoubleBuffer** pertencente à propriedade **pfdFlags**;

- ***Buffer* de profundidade (*Depth Buffer*):** Armazena o valor da profundidade, em relação ao observador, da cada um dos *pixels.* O *depth buffer* aplica o algoritmo *Z–buffer*;

- ***Buffer* de seleção (*Stencil Buffer*):** Utilizado para desenhar objetos dentro de uma região previamente selecionada, semelhante a uma pintura com spray sobre uma máscara;

- ***Buffer de acumulação (Accumulation Buffer)*:** Permite que as imagens sejam criadas de maneira sobreposta, antes da exibição. Desta maneira, pode-se criar o efeito da perda de foco (*motion blur*);

4.4. Sintaxe das instruções

Para diferenciar as instruções e constantes pertencentes à OpenGL é adicionado o prefixo **`gl`** para instruções e **`GL_`** para constantes antes do nome do comando.
É possível especificar as características dos parâmetros de algumas instruções utilizando alguns caracteres após o nome do comando.

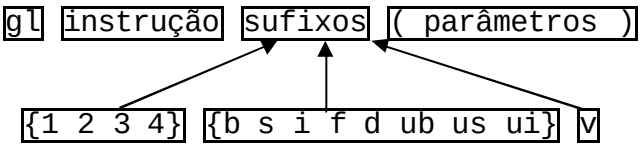


Figura 4.2: Sintaxe das instruções

O primeiro sufixo consiste no número de dimensões do espaço utilizado. Normalmente são utilizados os espaços contendo o sistema de coordenadas do objeto, do mundo, de tela, de textura e de cor. Estes valores pertencem ao intervalo de 1 até 4.

Tabela 4.1: Número de parâmetros das instruções

Sufixo	Componentes	Exemplos
1	Espaço de dimensão 1	(u) ou (s) ou (i)
2	Espaço de dimensão 2	(x,y) ou (u,v) ou (s,t)
3	Espaço de dimensão 3	(x,y,z) ou (s,t,r) ou (R,G,B)
4	Espaço de dimensão 4	(x,y,z,w) ou (s,t,r,q) ou (R,G,B,A)

O segundo sufixo consiste no tipo de dado, ou seja, especifica se as informações contidas no parâmetro são inteiras, reais e suas variações.

Tabela 4.2: Tipo de dados das instruções

Sufixo	Descrição dos tipos de dado	Tipo de dado
b	Inteiro com 8 bits	GLbyte
s	Inteiro com 16 bits	GLshort
i	Inteiro com 32 bits	GLint, GLsizei
f	Ponto flutuante com 32 bits	GLfloat, GLclampf
d	Ponto flutuante com 64 bits	GLdouble, GLclampd
ub	Inteiro não negativo com 8 bits	GLubyte, GLboolean
us	Inteiro não negativo com 16 bits	GLushort
ui	Inteiro não negativo com 32 bits	GLuint, GLenum, GLbitf

O terceiro parâmetro é opcional. Caso não seja explicitado, será considerado que os dados são formados por um conjunto de valores separados por vírgula, sendo que a quantidade de elementos é especificada pelo primeiro sufixo.
Caso seja utilizado o sufixo **v**, o parâmetro esperado é o endereço do primeiro elemento de um vetor, sendo que a quantidade de elementos é especificada pelo primeiro sufixo. Em Delphi, isto é feito indicando o nome da variável indexada antecedida pelo caractere @;

Tabela 4.3: Indicação de dados vetoriais das instruções

Sufixo	Arranjo dos dados	Exemplo
	Vários valores separados por vírgula	glColor3f(0.5, 0.5, 1.0)
v	Uma variável indexada	glColor3fv(@cor)

4.5. Máquina de estados

A OpenGL atua como uma máquina de estados, ou seja, o estado atual influencia no próximo estado após a execução da instrução.
Por exemplo: deseja-se desenhar um quadrado e um triângulo, utilizando as seguintes seqüências de vértices orientados, dispostos da seguinte maneira:

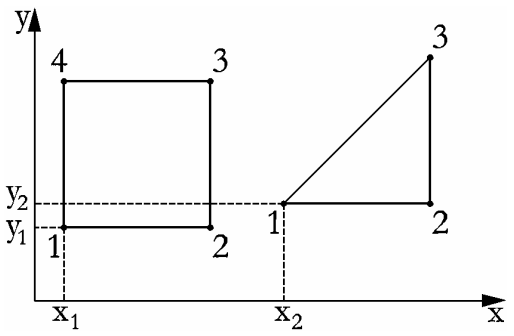


Figura 4.3: Deslocamento relativo

De uma maneira simplificada, podemos descrever a criação deste ambiente utilizando os seguintes passos:

1. Translação (x_1, y_1);
2. Desenhar um polígono cujos vértices são: 1, 2, 3 e 4;
3. Translação (x_2–x_1, y_2–y_1);
4. Desenhar um polígono cujos vértices são: 1, 2 e 3.

Como pode ser observado, é aplicada uma translação relativa ao vértice (x_1, y_1) para desenhar o triângulo. Assim, apenas o deslocamento em relação à última origem definida é realizado. O mesmo é válido para as outras transformações lineares, como rotação e mudança de escala.

Observação: os vértices dos polígonos são definidos em função do sistema de coordenadas do objeto, ou seja, independem da sua localização na cena.

5. Preparando para desenhar

5.1. Introdução

Deve-se lembrar que os conceitos da máquina de estados são aplicados aos vértices que serão desenhados. Desta maneira, algumas características deverão ser definidas antes que as instruções que gerem primitivas sejam executadas.

5.2. O ambiente de desenho

O primeiro passo para que a imagem seja desenhada, consiste na configuração do ambiente de desenho.
A instrução **`glClearColor( )`** recebe quatro valores como parâmetros, sendo que os três primeiros representam as componentes vermelho (**R**), verde (**G**) e azul (**B**) da cor do fundo da imagem. Estes valores reais devem pertencer ao intervalo fechado [0; 1]. Ao quarto parâmetro atribui-se o valor 1, uma vez que representa a componente alfa (a) da transparência, ainda não apresentada. Caso esta instrução não seja declarada, a configuração inicial da OpenGL utilizará o preto (0, 0, 0, 1) como fundo.
A segunda instrução que deve ser declarada consiste em preparar os *buffers* de cor e profundidade para o novo desenho. Para limpar os *buffers* utiliza-se a instrução **`glClear( )`**.
A constante associada ao *buffer* de cor é **`GL_COLOR_BUFFER_BIT`**, enquanto que a constante associada ao *buffer* de profundidade é **`GL_DEPTH_BUFFER_BIT`**. Como serão utilizadas duas constantes pode-se utilizar o operador lógico **OU** para concatená-las.

```
glClear( GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT );
```

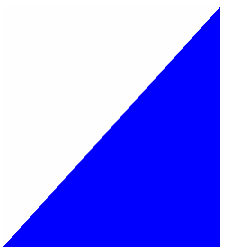
5.3. Tipo de Tonalização

Para que os modelos possam ser exibidos corretamente, deve-se definir qual será o tipo de tonalização que será aplicada. Para alterar o modo de tonalização, utiliza-se a instrução:

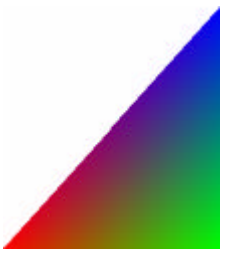
```
glShadeModel( );
```

A biblioteca OpenGL possui dois tipos de tonalização:

- **Constante:** Uma única cor é aplicada sobre todos os vértices de uma face, mesmo que cada um dos vértices que compõem a face possua cores diferentes dos demais. A cor que será aplicada é a cor associada a um dos vértices, dependendo do tipo de primitiva utilizada.
  A tonalização constante está associada à constante **`GL_FLAT`**;
- **Gouraud:** As cores pertencentes ao interior da face serão calculadas pela interpolação das cores dos vértices que compõem a face.
  A tonalização de Gouraud está associada à constante **`GL_SMOOTH`**;



`glShadeModel(GL_FLAT);`



`glShadeModel(GL_SMOOTH);`

Figura 5.1: Tipos de tonalização

5.4. Cores

Quando se deseja desenhar um primitivo com uma cor diferente da cor padrão (branco), utiliza-se a instrução **`glColor*( )`**. O caractere asterisco indica que esta instrução pode possuir sufixos para a identificação das características dos parâmetros, alocados dentro dos parênteses.
As seguintes sintaxes são válidas:

```
glColor3{b s i f d ub us ui} ( VERMELHO, VERDE, AZUL );

glColor4{b s i f d ub us ui} ( VERMELHO, VERDE, AZUL, ALFA );

glColor3{b s i f d ub us ui}v ( @VETOR3 );

glColor4{b s i f d ub us ui}v ( @VETOR4 );
```

Inicialmente serão apenas utilizadas cores contendo três componentes: vermelho (**R**), verde (**G**) e azul (**B**). A componente alfa (a), referente à transparência, será explicada mais adiante. Cada uma das componentes é quantizada em 256 níveis distintos.
Com relação à maneira pela qual os dados serão informados à OpenGL, usualmente utilizam-se os tipos byte (**b**) e ponto flutuante (**f**) para representar as cores. Desta maneira, a instrução **`glColor3b( )`** utiliza valores inteiros pertencentes ao intervalo fechado [0; 255], enquanto que o escopo da instrução **`glColor3f( )`** é definido pelo intervalo fechado [0; 1].
A figura a seguir mostra as cores definidas em um espaço de cores RGB. Neste espaço, o domínio e o contradomínio não assumem valores negativos. Este modelo de cor é descrito por um sistema com três eixos, ortogonais dois a dois, na direção dos vetores:

$$\vec{R} = \{1, 0, 0\}$$
$$\vec{G} = \{0,1, 0\}$$
$$\vec{B} = \{0, 0, 1\}$$

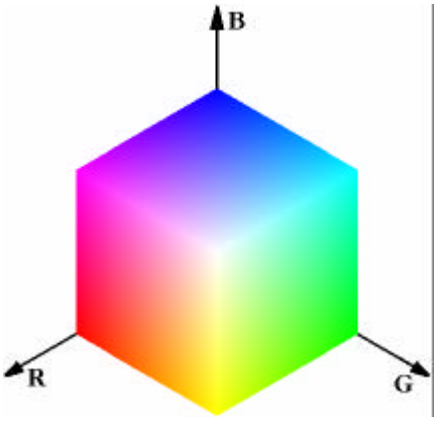


Figura 5.2: Espaço de cores RGB

Nota-se que o preto está na origem do sistema e é definido como sendo o vetor (0, 0, 0) enquanto que o branco é representado pelo vetor (1, 1, 1).

6. Transformações projetivas

6.1. Introdução

Outro elemento que deve ser definido antes que a imagem seja gerada consiste no posicionamento da câmera virtual, ou observador. Este observador também faz parte da cena, embora não possa ser observado.
Esta câmera possui dois conjuntos de parâmetros:

- **Extrínsecos:** Representam as características geométricas da câmera, como a sua posição, direção de observação e alinhamento;
- **Intrínsecos:** Estas características estão relacionadas ao tipo de projeção utilizado.

Para definir os **parâmetros extrínsecos** da câmera virtual, utiliza-se uma instrução da biblioteca GLU. Deve-se lembrar que não é necessário declará-la, pois a GLU está ligada diretamente à biblioteca OpenGL.
Para definir os parâmetros extrínsecos da câmera são necessários dois pontos (posição da câmera e o ponto que será observado) e o vetor *view up*, que define a rotação da câmera em torno do eixo imaginário que indica a profundidade da cena, como mostra a figura a seguir:

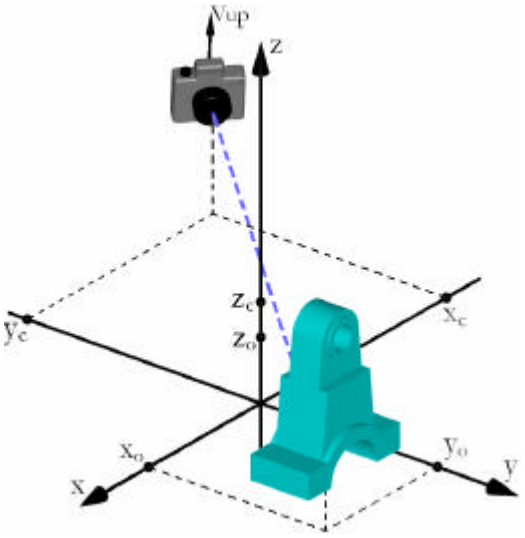


Figura 6.1: Posicionamento da câmera

Para definir estas características na OpenGL, utiliza-se a instrução **`gluLookAt( )`**.

```
gluLookAt( xc, yc, zc, xo, yo, zo, Vupx, Vupy, Vupz );
```

Na maioria dos casos, utiliza-se como vetor *view up* o vetor $\overrightarrow{V_{up}} = (0 \quad 1 \quad 0)$ para ambientes bidimensionais e o vetor $\overrightarrow{V_{up}} = (0 \quad 0 \quad 1)$ para ambientes tridimensionais.

Com o ambiente modelado e a câmera posicionada, é definido um volume de visualização que será projetado no plano da tela. Este volume está diretamente relacionado às características **intrínsecas** da câmera.
Da mesma maneira que são utilizadas matrizes 4x4 para alterar as características geométricas dos vértices, é possível projetá-los $(\Re^3 \rightarrow \Re^2)$ com o uso de matrizes 4x4.

A biblioteca permite manipular as características das matrizes de projeção ou transformação. Para que a matriz de projeção possa ser manipulada, deve-se selecioná-la utilizando a instrução **`glMatrizMode( )`**. A constante associada à projeção é **`GL_PROJECTION`**.

```
glMatrizMode( GL_PROJECTION );
```

Lembrando que o conceito de máquina de estado é aplicado na matriz de projeção, não é possível saber, com as instruções apresentadas até o momento, quais são as características atuais da matriz de projeção. Assim, se for aplicada alguma alteração, o resultado pode ser algo completamente imprevisível.
Para garantir que o resultado seja o esperado, deve-se impor uma condição inicial à matriz de projeção. Como condição inicial, a matriz de projeção deve ser uma matriz identidade 4x4, utilizando a instrução **`glLoadIdentity`**.
Desta maneira, antes de aplicar uma alteração na matriz de projeção, deve-se selecioná-la e impor que seja uma matriz identidade.

```
glMatrizMode( GL_PROJECTION );
glLoadIdentity;
```

Como resultado, a matriz de projeção será:

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Após estas duas instruções, é possível atribuir as características desejadas à matriz de projeção. A OpenGL permite que seja utilizada a projeção ortográfica e a projeção perspectiva.

6.2. Projeção Ortográfica

O volume de visualização da projeção ortográfica é um paralelepípedo. Por tratar-se de uma projeção paralela, a profundidade não influencia nas relações de paralelismo entre os objetos. A figura a seguir mostra o volume de visualização da projeção ortográfica.

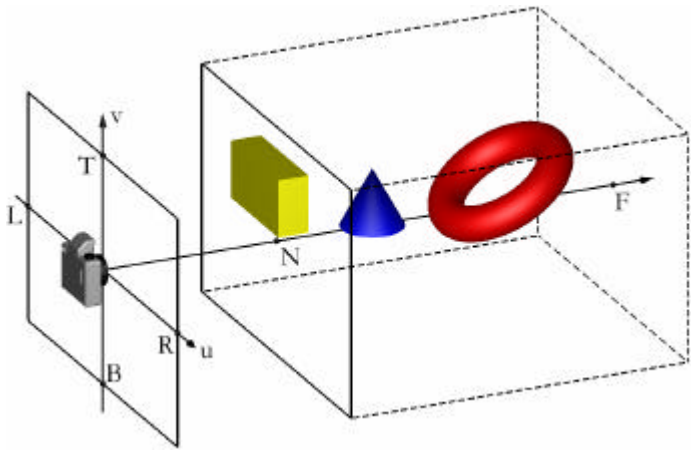


Figura 6.2: Volume de visualização da projeção ortográfica

Com a câmera alocada à frente deste volume, identificado pelo traçado contínuo de uma das faces, um novo sistema de coordenadas é utilizado. Neste sistema a coordenada **z** é definida como a profundidade do volume, na direção de observação da câmera.
Para que a cena seja projetada seguindo as características da projeção ortográfica, utiliza-se a instrução **`glOrtho( )`**. São necessários seis parâmetros para definir o volume de visualização. Estes valores são definidos em coordenadas do **sistema de referência da câmera**: esquerda (**L**), direita (**R**), inferior (**B**), superior (**T**), perto (**N**) e longe (**F**).

```
glOrtho( L, R, B, T, N, F );
```

A projeção ortográfica é feita eliminando a coordenada **z**, como mostra a figura a seguir:

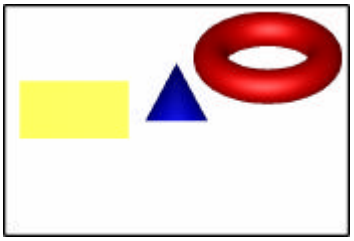


Figura 6.3: Imagem do volume de visualização obtida pela projeção ortográfica

Nota-se que os valores **L**, **R**, **B** e **T** definem uma janela retangular que pode, ou não, conter a origem. Da mesma maneira, a câmera pode apenas observar o volume de visualização, quando **N > 0** ou estar contida nele (**N ≤ 0**). É importante lembrar que a imagem será invertida quando **L > R** ou **B > T** ou **N > F**.
A matriz de projeção definida pela instrução **`glOrtho`** é:

$$P = \begin{bmatrix} \frac{2}{R-L} & 0 & 0 & \frac{R+L}{R-L} \\ 0 & \frac{2}{T-B} & 0 & \frac{T+B}{T-B} \\ 0 & 0 & \frac{-2}{F-N} & \frac{F+N}{F-N} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

6.3. Projeção Perspectiva

O volume de visualização da projeção perspectiva é um tronco de uma pirâmide de base retangular. Na projeção perspectiva, a profundidade irá influenciam nas relações de paralelismo entre os objetos, distorcendo-os à medida que se afastam da câmera. Na OpenGL é possível realizar a projeção perspectiva de duas maneiras distintas, porém equivalentes.
A instrução **`glFrustum`** utiliza os mesmos seis parâmetros da instrução **`glOrtho`** para definir o volume de visualização. Estes valores são definidos em coordenadas do **sistema de referência da câmera** : esquerda (**L**), direita (**R**), inferior (**B**), superior (**T**), perto (**N**) e longe (**F**).

```
glFrustum( L, R, B, T, N, F );
```

A figura a seguir mostra o volume de visualização da projeção perspectiva, utilizando os argumentos da instrução **`glFrustum( )`**.

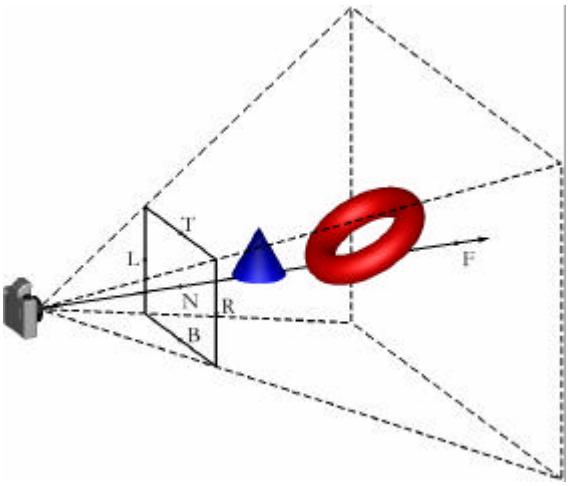


Figura 6.4: Volume de visualização da projeção perspectiva utilizando **`glFrustum`**

Da mesma maneira anteriormente apresentada, os valores **L**, **R**, **B** e **T** definem uma janela retangular que pode, ou não, conter a origem. Entretanto, por tratar-se de um tronco de pirâmide com o vértice localizado na câmera, somente são válidos valores positivos para **N** e **F.**
A matriz de projeção definida pela instrução **`glFrustum`** é:

$$P = \begin{bmatrix} \frac{2\cdot N}{R-L} & 0 & \frac{R+L}{R-L} & 0 \\ 0 & \frac{2\cdot N}{T-B} & \frac{T+B}{T-B} & 0 \\ 0 & 0 & -\frac{F+N}{F-N} & \frac{-2\cdot F\cdot N}{F-N} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Existe uma instrução pertencente à GLU que permite definir um volume de visualização **simétrico** utilizando como parâmetros o ângulo (em graus) entre a direção de observação da câmera e a direção do eixo y da imagem[2]; além da relação entre a largura (**w**) e a altura (**h**) da imagem (*aspect ratio*). O volume de visualização da instrução **`gluPerspective( )`** é mostrado na figura a seguir:

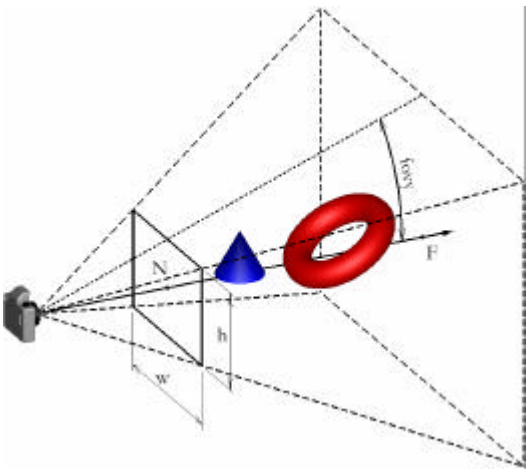


Figura 6.5: Volume de visualização da projeção perspectiva utilizando **`gluPerspective`**

[2] *Field of view in y (fovy)*

A instrução **`gluPerspective`** utiliza como parâmetros o ângulo de abertura entre os eixos **z** e **y** (**fovy**), a relação entre a largura e a altura da imagem (**AR = w / h**) e as distâncias próxima (**N**) e longe (**F**) dos planos ortogonais ao eixo da profundidade.

```
gluPerspective( fovy, AR, N, F );
```

Um cuidado a ser tomado consiste em que o ângulo de abertura deve pertencer ao intervalo aberto (0°; 180°). Além disso, os valores das distâncias **N** e **F** ao longo do eixo da profundidade (**z**) devem assumir somente valores positivos.
A matriz de projeção definida pela instrução **`gluPerspective`** é:

$$P = \begin{bmatrix} \frac{\lambda}{AR} & 0 & 0 & 0 \\ 0 & \lambda & 0 & 0 \\ 0 & 0 & \frac{N+F}{N-F} & \frac{2 \cdot N \cdot F}{N-F} \\ 0 & 0 & -1 & 0 \end{bmatrix}, \text{ sendo que: } \lambda = \operatorname{cotg}\left(\frac{fovy}{2}\right) \text{ e } AR = \frac{w}{h}.$$

A imagem a seguir mostra o resultado da projeção perspectiva do volume de visualização, independente da instrução utilizada, desde que os parâmetros representem o mesmo volume de visualização.

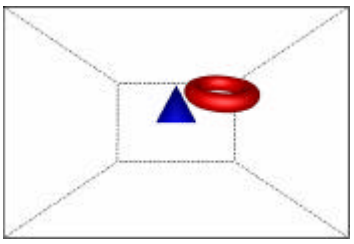


Figura 6.6: Imagem do volume de visualização obtida pela projeção perspectiva

7. Descrição de elementos primitivos

7.1. Introdução

Nesta seção serão apresentadas as principais primitivas geométricas que permitirão ao programador criar elementos gráficos compostos, limitando o resultado exclusivamente pela criatividade.

7.2. Definindo características dos elementos primitivos

É possível alterar o "tamanho" de um ponto, utilizando a instrução **`glPointSize( )`**. Esta instrução possui apenas um parâmetro inteiro que indica o tamanho de um **quadrado**, em *pixels*, que representará o ponto.
De maneira semelhante, é possível alterar a espessura dos segmentos de retas e contornos de polígonos utilizando a instrução **`glLineWidth( )`**. Assim como o "tamanho" do ponto, a espessura da linha é definida como um valor inteiro que representa o número de *pixels*.
Um outro efeito relacionado às linhas consiste em suas características quanto ao tipo: contínua, tracejada, pontilhada, traço-ponto, ou qualquer combinação desejada. A instrução **`glLineStipple( )`** permite que seja criado um padrão qualquer utilizando um conjunto de 16 bits (65536 combinações distintas), que indicam se os *pixels* serão exibidos (1) ou não (0).
O primeiro parâmetro indica qual será o fator de escala inteiro (**fs**) aplicado ao padrão definido por um valor hexadecimal de 16 bits alocado no segundo parâmetro.

```
glLineStipple( fs, padrão_hexadecimal );
```

Para que as características do traçado sejam apresentadas, deve-se habilitar o uso do padrão do traçado das linhas utilizando a instrução **`glEnable ( )`**. A constante associada ao uso do padrão do traçado é **`GL_LINE_STIPPLE`**.

```
glEnable( GL_LINE_STIPPLE );
```

A figura a seguir exemplifica como será o traçado para diferentes parâmetros:

Padrão	fs	Linha
$FFFF	1	
$00FF	1	
$00FF	2	
$E4E4	1	
$AAAA	1	
$AAAA	2	
$AAAA	3	

Figura 7.1: Exemplos de padrões de linha

7.3. Posicionamento de vértices

Para que os objetos possam ser descritos pode-se utilizar uma lista contendo as coordenadas dos vértices que compõem tais objetos, uma outra lista contendo as relações entre estes vértices para que possam compor uma aresta e outra que relacione as arestas para formarem as faces.

Neste momento é de interesse que estes elementos tornem-se mais "reais", podendo ser manipulados sob a forma de valores numéricos resultando em imagens que simulem uma fotografia ou simplesmente movam-se, realizam rotações ou mudem de cor para deleito do usuário.
Uma vez que os vértices e linhas possuam as características desejadas, pode-se criar uma lista ordenada de vértices (semelhante à estrutura *Hard Edge*) limitada pelas instruções **`glBegin( )`** e **`glEnd`**. O argumento da instrução **`glBegin`** será detalhado no item 7.5.
As coordenadas dos vértices serão definidas pela instrução **`glVertex*( )`**. Deve-se lembrar que o caractere asterisco indica que esta instrução pode possuir sufixos para a identificação das características dos parâmetros, alocados dentro dos parênteses. As seguintes sintaxes são válidas:

```
glVertex2{s i f d} ( X, Y );

glVertex3{s i f d} ( X, Y, Z );

glVertex4{s i f d} ( X, Y, Z, W );

glVertex2{s i f d}v ( @VETOR2 );

glVertex3{s i f d}v ( @VETOR3 );

glVertex4{s i f d}v ( @VETOR4 );
```

O uso da instrução **`glVertex*`** somente é válido quando limitada pelas instruções **`glBegin`** e **`glEnd`**.

7.4. Vetores normais

Se o vetor normal uma face planar não for definido, a biblioteca OpenGL considera que o vetor normal de cada vértice é o próprio vetor normal da face que os contém. Este vetor normal pode ser determinado pelo produto vetorial entre os vetores definidos por três vértices **consecutivos** da lista de vértices que definem cada face, ou seja, **a orientação do vetor normal pode ser definida pela seqüência dos vértices**, como mostrado na figura a seguir:

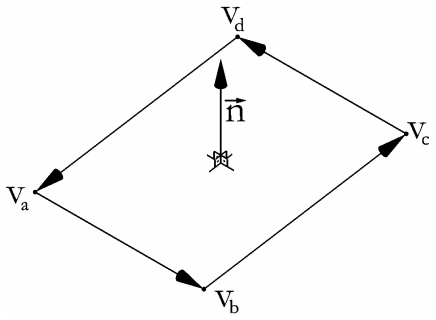


Figura 7.2: Vetor normal de superfície planar

$$\vec{n}_{abc} = \overrightarrow{V_a V_b} \times \overrightarrow{V_b V_c} = (V_b - V_a) \times (V_c - V_b) \qquad \vec{n}_{bcd} = \overrightarrow{V_b V_c} \times \overrightarrow{V_c V_d} = (V_c - V_b) \times (V_d - V_c)$$

$$\vec{n}_{cda} = \overrightarrow{V_c V_d} \times \overrightarrow{V_d V_a} = (V_d - V_c) \times (V_a - V_d) \qquad \vec{n}_{dab} = \overrightarrow{V_d V_a} \times \overrightarrow{V_a V_b} = (V_a - V_d) \times (V_b - V_a)$$

Deve-se lembrar que os vetores devem possui modulo unitário. Para tal, divide-se o vetor pelo seu próprio tamanho:

$$\vec{n} = \frac{\vec{n}_{abc}}{|\vec{n}_{abc}|} = \frac{\vec{n}_{bcd}}{|\vec{n}_{bcd}|} = \frac{\vec{n}_{cda}}{|\vec{n}_{cda}|} = \frac{\vec{n}_{dab}}{|\vec{n}_{dab}|}$$

A correta definição do vetor normal implica na correta visualização das faces quando o modelo de iluminação for utilizado.
É possível gerar efeitos visuais, definido novos vetores normais às faces. Para tal, utiliza-se a instrução **`glNormal*( )`** que será apresentada no item 11.3.

7.5. Desenhando elementos primitivos

Algumas condições ou restrições irão diferenciar os vértices, as arestas e as faces manipuladas computacionalmente dos seus equivalentes matemáticos. Por exemplo, é possível visualizar um ponto na tela do computador, entretanto um ponto matemático é adimensional. Da mesma maneira que as retas, que são infinitas em sua extensão, serão tratadas apenas como segmentos de retas. As arestas serão definidas por polígonos convexos e contínuos.
Para desenhar, deve-se criar uma **lista ordenada** de vértices limitada pelas instruções **`glBegin( )`** e **`glEnd`**. A diferença entre a criação dos elementos ocorre alterando a constante associada à instrução **`glBegin( )`**.
Existem dez constantes que podem ser utilizadas como argumento desta instrução.

- **`GL_POINTS`**
  Desenha um ponto para cada um dos **n** vértices;

- **`GL_LINES`**
  Um segmento de reta é desenhado para cada par de vértices. Estes segmentos não estão interligados. Caso haja um número ímpar de vértices, o último vértice é ignorado;

- **`GL_LINE_STRIP`**
  Desenha uma seqüência de segmentos de reta que ligam o vértice final de um segmento com o vértice inicial do segmento seguinte. Não existe nenhuma restrição com relação à posição dos vértices. Para um conjunto contendo **n** vértices, são desenhados **n–1** segmentos de reta;

- **`GL_LINE_LOOP`**
  Semelhante à constante **`GL_LINE_STRIP`**. Entretanto o último vértice é ligado ao primeiro, resultando em um conjunto de **n** vértices e **n** segmentos de reta.

- **`GL_TRIANGLES`**
  Um triângulo é desenhado para cada três vértices. Estes triângulos não estão interligados. Caso o número de vértices não seja múltiplo de três, os vértices excedentes serão ignorados.

- **`GL_TRIANGLE_STRIP`**
  Desenha um conjunto de triângulos adjacentes. Para um triângulo definido pelos vértices V_i, V_{i+1} e V_{i+2}, o triângulo adjacente é definido pelos vértices V_{i+2}, V_{i+1} e V_{i+3}. O triangulo seguinte é composto pelos vértices V_{i+2}, V_{i+3} e V_{i+4}.
  Deve-se notar que para garantir que todos os vetores normais possuam o mesmo sentido em todos os triângulos, a OpenGL desenha alguns triângulos utilizando um sentido diferente dos dois triângulos adjacentes.

- **GL_TRIANGLE_FAN**
  Desenha um conjunto de triângulos adjacentes. O primeiro vértice definido é comum a todos os triângulos. Os triângulos são definidos da seguinte maneira: V_0, V_1 e V_2; V_0, V_2 e V_3; V_0, V_3 e V_4; e assim por diante.

- **GL_QUADS**
  Semelhante à constante **GL_TRIANGLES**. Para cada conjunto contendo quatro vértices são desenhadas faces quadriláteras **convexas** independentes. Caso o número de vértices não seja múltiplo de quatro, os vértices excedentes serão ignorados.

- **GL_QUAD_STRIP**
  Desenha um conjunto de quadriláteros **convexos** adjacentes. Para um quadrilátero definido pelos vértices V_i, V_{i+1}, V_{i+3} e V_{i+2}, o quadrilátero adjacente é definido pelos vértices V_{i+2}, V_{i+3}, V_{i+5} e V_{i+4}. O quadrilátero seguinte é composto pelos vértices V_{i+4}, V_{i+5}, V_{i+7} e V_{i+6}.

  Deve-se notar que para garantir que todos os vetores normais possuam o mesmo sentido em todos os quadriláteros, a OpenGL desenha alguns quadriláteros utilizando um sentido diferente dos dois quadriláteros adjacentes.

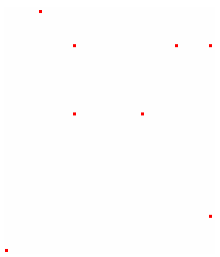
- **GL_POLYGON**
  Permite desenhar um polígono **convexo** composto por **n** vértices e **n** arestas. Deve-se lembrar que são necessários pelo menos três vértices.

Com relação à criação de qualquer tipo de face, seja triangular, quadrilátera ou poligonal, existem algumas restrições que devem ser respeitadas:

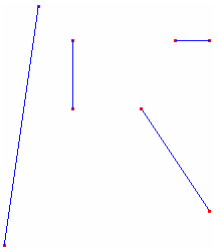
- As arestas que unem os vértices não podem interceptar-se;
- As faces devem ser convexas para que a biblioteca possa aplicar os algoritmos de tonalização e preenchimento sem que o valor dos vértices possa ser dúbio;
- Os polígonos devem ser contínuos em toda a sua região convexa, ou seja, não devem existir buracos;
- Deve-se garantir que a lista de vértices esteja corretamente ordenada para que a biblioteca defina corretamente os vetores normais.

Exemplo 7.1-) O código a seguir desenha oito vértices em um espaço bidimensional. Cada figura foi gerada a partir de um dos dez tipos de elementos primitivos apresentados. Para tal, foi criada uma variável denominada ALGUMA_COISA que está relacionada à respectiva constante. Esta variável representa o parâmetro da instrução **glBegin( )**.

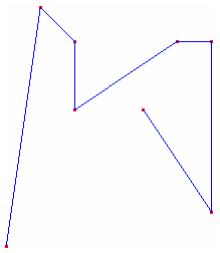
```
glBegin ( ALGUMA_COISA );
  glVertex2f(0.0,0.0);
  glVertex2f(1.0,-1.5);
  glVertex2f(1.0,1.0);
  glVertex2f(0.5,1.0);
  glVertex2f(-1.0,0.0);
  glVertex2f(-1.0,1.0);
  glVertex2f(-1.5,1.5);
  glVertex2f(-2.0,-2.0);
glEnd;
```



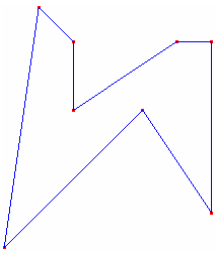
GL_POINTS



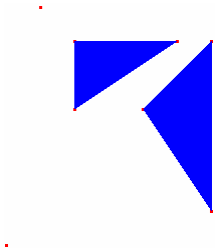
GL_LINES



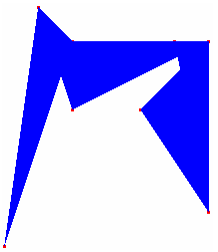
GL_LINE_STRIP



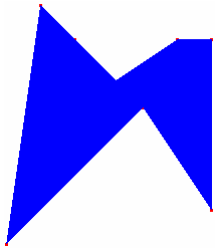
GL_LINE_LOOP



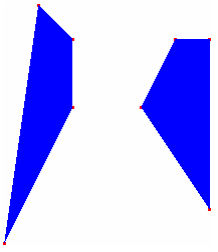
GL_TRIANGLES



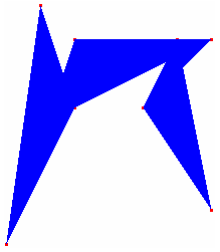
GL_TRIANGLE_STRIP



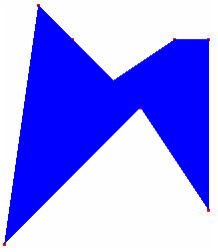
GL_TRIANGLE_FAN



GL_QUADS



GL_QUAD_STRIP



GL_POLYGON

Figura 7.3: Elementos primitivos

Exemplo 7.2-) Elabore um procedimento de um programa em Delphi, utilizando a biblioteca OpenGL, que desenhe um cubo, cuja dimensão do lado é passada como parâmetro. O cubo deve estar centrado na origem e suas faces orientadas de tal maneira que o vetor normal aponte para o exterior do modelo. Cada uma das faces deve possuir uma cor diferente.
Solução:

```
procedure Desenha_cubo(L:real);
begin
glClearColor(1,1,1,1);    // fundo branco

// limpa os buffers de cor e profundidade
glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);

// zera as matrizes
glLoadIdentity;

// define o volume de visualização
glOrtho(-2,2,-2,2,-10,10);

gluLookAt(1,1,1,          // posição do observador ...
          0,0,0,          // ... que está olhando para a origem ...
          0,0,1);         // ... com a cabeça na direção de Z

L := L/2;     // metade do lado do quadrado para cada lado dos eixos
glBegin(GL_QUADS);
  //face Z positivo
  glColor3f(1,0,0);       // vermelho
  glVertex3f(-L,-L,L);
  glVertex3f(L,-L,L);
  glVertex3f(L,L,L);
  glVertex3f(-L,L,L);
  //face Z negativo
  glColor3f(0,1,1);       // ciano
  glVertex3f(-L,-L,-L);
  glVertex3f(-L,L,-L);
  glVertex3f(L,L,-L);
  glVertex3f(L,-L,-L);
  //face X positivo
  glColor3f(0,1,0);       // verde
  glVertex3f(L,-L,-L);
  glVertex3f(L,L,-L);
  glVertex3f(L,L,L);
  glVertex3f(L,-L,L);
  //face X negativo
  glColor3f(1,0,1);       // magenta
  glVertex3f(-L,-L,-L);
  glVertex3f(-L,-L,L);
  glVertex3f(-L,L,L);
  glVertex3f(-L,L,-L);
  //face Y positivo
  glColor3f(0,0,1);       // azul
  glVertex3f(L,L,-L);
  glVertex3f(-L,L,-L);
  glVertex3f(-L,L,L);
  glVertex3f(L,L,L);
```

```
  //face Y negativo
  glColor3f(1,1,0);       // amarelo
  glVertex3f(L,-L,-L);
  glVertex3f(L,-L,L);
  glVertex3f(-L,-L,L);
  glVertex3f(-L,-L,-L);
glEnd;
glFlush;                  // atualiza a tela
end;
```

Utilizando L = 2, a seguinte imagem é exibida:

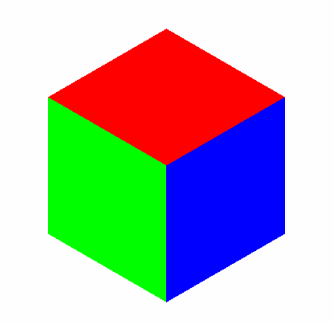
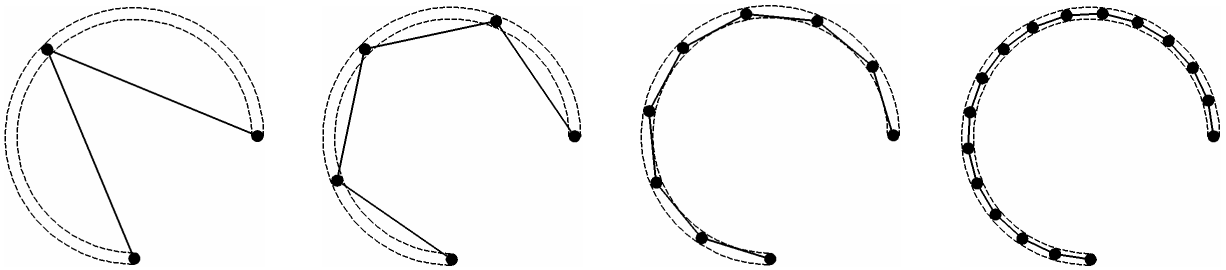


Figura 7.4: Cubo com faces coloridas

7.6. Aproximação linear e planar

Para desenhar uma curva que interpole um determinado número de pontos, pode-se, de maneira grotesca, uni-los por segmentos de retas. Quanto maior a quantidade de pontos conhecidos, melhor será a aproximação da curva. Ou seja, as retas serão tão pequenas que o efeito visual não denunciará a técnica utilizada, como é mostrado na figura a seguir:



Dois segmentos | Quatro segmentos | Oito segmentos | Dezesseis segmentos

Figura 7.5: Aproximação linear

De maneira análoga, pequenos polígonos que podem ser unidos formando superfícies complexas.

Exemplo 7.3-) Elabore um procedimento de um programa em Delphi, utilizando a biblioteca OpenGL, que desenhe a função $f(u,v) = 2 \cdot e^{-u^2} \cdot e^{-v^2}$ definida nos intervalos $[u_{min}; u_{max}]$ e $[v_{min}; v_{max}]$. Os valores limitantes do domínio são informados via parâmetro.

Solução:

Por tratar-se de uma superfície, deve-se utilizar a aproximação planar para representá-la. A projeção de cada uma das faces no plano do domínio da superfície é equivalente a um retângulo de dimensões **du** e **dv**, como mostrado na figura a seguir:

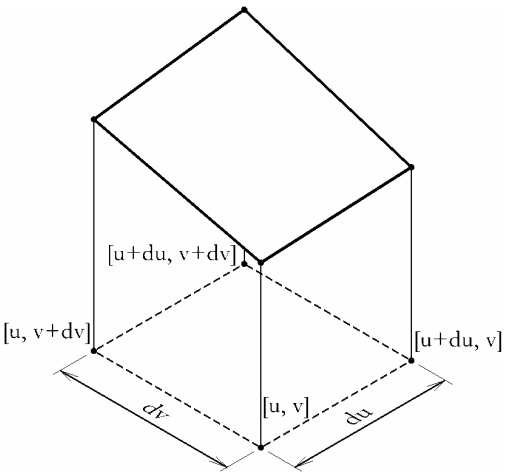


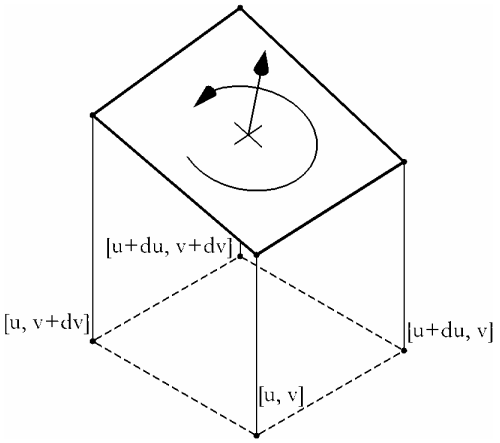
Figura 7.6: Elemento de área

Os valores de **du** e **dv** podem ser quaisquer proporções da extensão do domínio. Neste exemplo cada uma das direções paramétricas será dividida em 20 partes iguais, ou seja:

$$du = \frac{u_{max} - u_{min}}{20} \text{ e } dv = \frac{v_{max} - v_{min}}{20}$$

Para calcular a coordenada **z** de cada um dos vértices $[u \quad v \quad ?]$ das faces, verifica-se qual é o valor associado à função **f** utilizando as coordenadas nas direções paramétricas **u** e **v**. em outras palavras, deve-se utilizar o conjunto de pontos com a característica $[u \quad v \quad f(u,v)]$.

Para garantir que os vetores normais das faces possuam sempre a componente $\mathbf{n_z}$ positiva, todas as faces devem ser definidas utilizando-se uma seqüência de vértices no mesmo sentido, seguindo a regra da mão direita. Para qualquer que seja o valor de **u** e **v**, pode-se definir a face utilizando a seguinte **seqüência** de vértices:



Primeiro vértice:
$[u \quad v \quad f(u,v)]$

Segundo vértice:
$[u+du \quad v \quad f(u+du,v)]$

Terceiro vértice:
$[u+du \quad v+dv \quad f(u+du,v+dv)]$

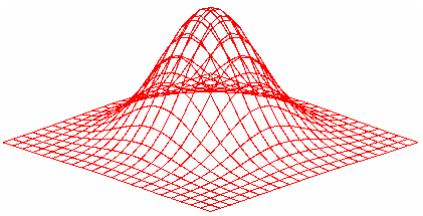
Quarto vértice:
$[u \quad v+dv \quad f(u,v+dv)]$

Figura 7.7: Seqüência dos vértices

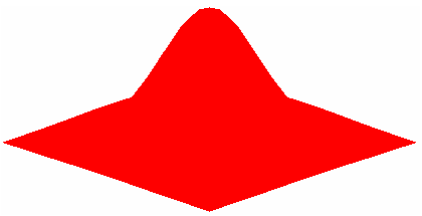
Uma vez que o problema foi corretamente modelado (definida a função, o seu domínio e a seqüência de vértices que serão utilizados) implementa-se no computador:

```
function f(u,v:real):real;
begin
f := 2*exp(-sqr(u))*exp(-sqr(v));
end;

procedure desenha_funcao(umin,umax,vmin,vmax:real);
var u,v,du,dv: real;
begin
glClearColor(1,1,1,1);
glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);

glLoadIdentity;
glOrtho(-3,3,-3,3,-10,10);
gluLookAt(1,1,0.5,0,0,0,0,0,1);

glColor3f(1,0,0);
u := umin;                // valor inicial de u
du := (umax-umin)/20;     // incremento da direção u
dv := (vmax-vmin)/20;     // incremento da direção v
repeat
  v := vmin;              // valor inicial de v
  repeat
    // desenha uma face
    glBegin(GL_LINE_LOOP);
      glVertex3f(u,v,f(u,v));
      glVertex3f(u+du,v,f(u+du,v));
      glVertex3f(u+du,v+dv,f(u+du,v+dv));
      glVertex3f(u,v+dv,f(u,v+dv));
    glEnd;
    v := v + dv;          // incremento na direção v
  until v >= vmax;
  u := u + du;            // incremento na direção u
until u >= umax;
glFlush;                  // atualiza a tela
end;
```

A execução do código acima exibe uma curva em *wireframe*. Entretanto é possível desenhá-la como uma superfície preenchida, alterando o valor do parâmetro de **GL_LINE_LOOP** para **GL_QUADS**.



GL_LINE_LOOP



GL_QUADS

Figura 7.8: Aproximação planar

Exemplo 7.4-) Elabore um procedimento de um programa em Delphi, utilizando a biblioteca OpenGL, que desenhe uma esfera de raio informado via parâmetro e cujo centro localiza-se na origem.

Solução:

De maneira semelhante ao exemplo anterior, devem-se determinar as dimensões dos retângulos que farão a aproximação planar e a orientação dos vértices.
As direções paramétricas **u** e **v** do exemplo anterior coincidem com o sistema de referência cartesiano. Entretanto, para modelar uma esfera é mais fácil utilizar o sistema de coordenadas esférico.

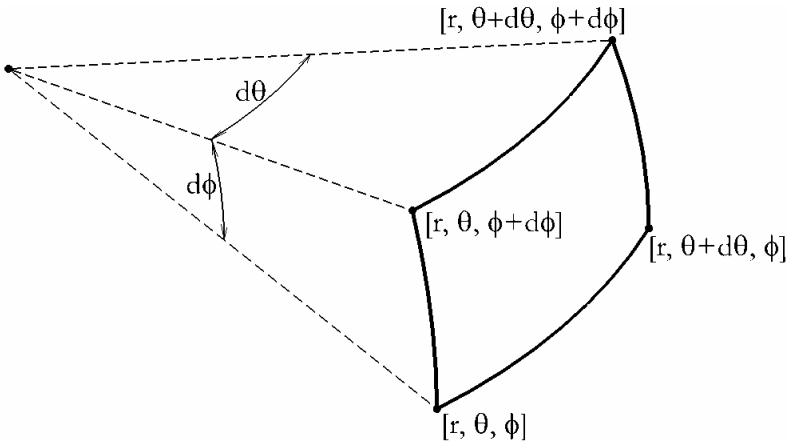


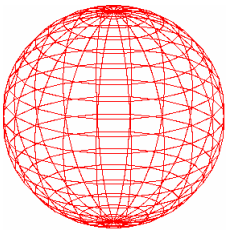
Figura 7.9: Elemento de área

Para que o vetor normal aponte para fora da esfera, utiliza-se a seguinte ordem dos vértices: $[r \quad \theta \quad \phi]$, $[r \quad \theta+d\theta \quad \phi]$, $[r \quad \theta+d\theta \quad \phi+d\phi]$ e $[r \quad \theta \quad \phi+d\phi]$.

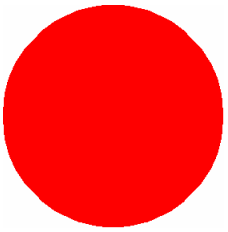
```
// Tipo de dados para armazenar as coordenadas de pontos 3D
type TPonto3d = Array [0..2] of GLfloat;

// Converte as coordenadas do sistema esférico para o cartesiano
function e2c(r,t,f:real):TPonto3d;
begin
e2c[0] := r*cos(t)*cos(f);     // X
e2c[1] := r*sin(t)*cos(f);     // Y
e2c[2] := r*sin(f);            // Z
end;

procedure desenha_esfera(r:real);
var t,f,dt,df: real;
    ponto: TPonto3d;
begin

glClearColor(1,1,1,1);         // fundo branco
glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);

glLoadIdentity;
glOrtho(-3,3,-3,3,-10,10);
gluLookAt(1,1,0.5,0,0,0,0,0,1);

glColor3f(1,0,0);
t := 0;                        // valor inicial de teta
dt := 2*pi/20;                 // incremento do ângulo teta
df := pi/20;                   // incremento do ângulo fi
repeat
  f := -pi/2;                  // valor inicial de fi
```

```
  repeat
    glBegin(GL_LINE_LOOP);      // desenha uma face
      ponto := e2c(r,t,f);      // converte de esférico para cartesiano
      glVertex3fv(@ponto);      // define o vértice utilizando um vetor
      ponto := e2c(r,t+dt,f);
      glVertex3fv(@ponto);
      ponto := e2c(r,t+dt,f+df);
      glVertex3fv(@ponto);
      ponto := e2c(r,t,f+df);
      glVertex3fv(@ponto);
    glEnd;
    f := f + df;                // incremento no ângulo fi
  until f > pi/2;
  t := t + dt;                  // incremento no ângulo teta
until t > 2*pi;
glFlush;                        // atualiza a tela
end;
```

A execução deste procedimento resulta em:



GL_LINE_LOOP



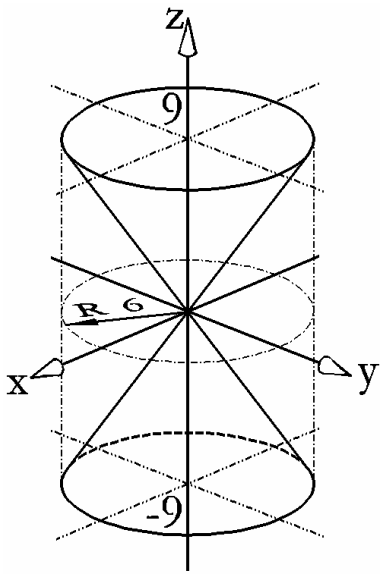
GL_QUADS

Figura 7.10: Aproximação planar

Nota-se que a estrutura da sub-rotina é muito semelhante às apresentadas anteriormente. As variações significativas são referentes à lista de vértices (limitada por **`glBegin`** e **`glEnd`**).
Outro detalhe importante consiste em que sempre foram utilizados quatro vértices para definir as faces, sejam as faces do cubo ou as pequenas faces das superfícies. Por tratar-se de uma aproximação **planar**, apenas duas coordenadas destas faces são alteradas. Assim, mantendo-se uma coordenada "constante" (com as mesmas características, como no segundo caso em que a coordenada **z** era uma função dos pontos **u** e **v**), define-se a face no sentido anti-horário para que os vetores normais sempre indiquem o lado externo (visível) do objeto.

7.1-) Elabore um procedimento de um programa em Delphi, utilizando a biblioteca OpenGL, para representar graficamente cada uma das superfícies a seguir:

a.) Um cilindro de raio 2 e altura 6.
b.) Um prisma de altura **h** de base elíptica. As dimensões máximas da elipse são **2· a** e **2· b** nas direções **x** e **y**.
c.) Um parabolóide de revolução, de altura **h**, cuja intersecção com o plano **XZ** é descrita pela função $z(x,y) = 2 \cdot x^2$ e a intersecção com o plano **YZ** é dada por $z(x,y) = 2 \cdot y^2$.
d.) Um elipsóide cujas dimensões máximas são **2· a**, **2· b** e **2· c**.
e.) O cone da figura a seguir



7.2-) Para determinar as equações de **x** e **y** no exercício anterior, foi utilizado o valor da cota **Z**. Quais seriam os efeitos em tais equações se o valor **Z** fosse substituído por $|Z|$?

Dica: Utilize cores diferentes para cada ponto da superfície.

7.3-) Indique quais são os pontos em comum e descreva uma maneira para otimizar o código, sem a necessidade de repeti-lo para cada uma das superfícies do exercício anterior.

7.4-) Utilizando as instruções da OpenGL, represente graficamente o logotipo da Mauá em três dimensões, sendo que o lado do quadrado maior e profundidade do sólido devem ser informados pelo usuário (na dúvida, olhe para a parte de baixo desta folha!).

8. Curvas e superfícies de Bézier

Para representar as curvas e superfícies de Bézier não é necessário utilizar linhas ou polígonos para aplicar a aproximação linear ou planar, embora estes conceitos sejam utilizados para que instruções específicas possam ser executadas de maneira correta.
Os responsáveis pelo cálculo das funções polinomiais de base são os *Evaluators*.

8.1. Curvas de Bézier

Sabe-se que as curvas de Bézier são descritas por funções polinomiais calculadas a partir de um conjunto de pontos de controle. Computacionalmente, os pontos de controle podem ser definidos, por exemplo, por uma variável indexada bidimensional contendo o número de pontos de controle necessários e a dimensão do espaço (no caso 3: **x**, **y** e **z**).

```
const num_pts = 5;  // utilizar no máximo np = 8
var pt_ctrl: array[1..num_pts,1..3] of GLfloat;
```

A definição de um *evaluator* unidimensional (direção paramétrica **u**) é realizada quando a instrução **glMap1*( )** é executada.

```
glMap1{f d}( TIPO_PC, UMIN, UMAX, AGRUPAMENTO, QUANT_PC, @PONTOS );
```

O primeiro parâmetro da instrução **glMap1*( )** identifica qual é o significado dos pontos de controle (vértices do espaço tridimensional, homogêneo, cor ou textura). Para as aplicações que serão desenvolvidas, os pontos de controle representam vértices definidos no $\Re^3$. Desta maneira, será utilizada a constante **GL_MAP1_VERTEX_3**.

Os dois valores seguintes representam o escopo do parâmetro **u**, ou seja, $u \in [u_{min}; u_{max}]$.

Normalmente o domínio da direção paramétrica **u** é definido pelo intervalo fechado $[0;1]$.

Embora a variável que contenha os pontos de controle tenha sido definida como uma variável indexada bidimensional o último parâmetro informa o endereço da primeira coordenada (**x**) do primeiro ponto de controle, tratando a informação de maneira seqüencial (como uma variável indexada unidimensional). Para que a instrução **glMap1*** possa discernir as coordenadas, o quarto parâmetro (agrupamento) deve indicar a dimensão do espaço (no caso 3), ou seja, a distância entre os valores da coordenada x_i e a coordenada x_{i+1}.

O quinto parâmetro indica a quantidade de pontos de controle que serão considerados. É possível utilizar uma quantidade de pontos menor do que o tamanho máximo da variável.
No último parâmetro deve ser indicado o endereço da primeira coordenada do primeiro ponto de controle, definidos previamente.
Para o exemplo apresentado anteriormente, temos a seguinte definição do *evaluator*.

```
glMap1f( GL_MAP1_VERTEX_3, 0, 1, 3, num_pts, @pt_ctrl );
```

Para habilitar o *evaluator* deve-se utilizar a instrução **glEnable( )**, utilizando o mesmo parâmetro referente ao significado dos pontos de controle (**GL_MAP1_VERTEX_3**).

```
glEnable( GL_MAP1_VERTEX_3 );
```

Para definir o domínio da direção paramétrica **u**, utiliza-se a instrução **glMapGrid1*( )**. Esta instrução necessita de três parâmetros: a quantidade de subdivisões do domínio e seus limites inferior e superior, normalmente $[0;1]$.

```
glMapGrid1{f d} ( SUBDIVISOES, UMIN, UMAX );
```

Para desenhar a curva, utiliza-se a instrução **`glEvalMesh1( )`**, cujos parâmetros representam o tipo de elemento utilizado no traçado (**`GL_POINT`** ou **`GL_LINE`**) e os extremos do domínio de **u**, em subdivisões:

```
glEvalMesh1( TIPO_ELEMENTO, 0, SUBDIVISOES );
```

O exemplo a seguir utiliza 15 subdivisões do domínio [0; 1]:

```
glMapGrid1f(15,0,1);
glEvalMesh1(GL_LINE,0,15);
```

Exemplo 8.1-) Representar graficamente a curva de Bézier definida pelos pontos: $[0 \quad 0 \quad 0]$, $[1 \quad 1 \quad 0]$, $[2 \quad -1 \quad 0]$ e $[3 \quad 0 \quad 0]$.

Solução:

```
procedure desenha_curva_de_Bezier;
var pc: array[1..4,1..3] of GLfloat;
begin
// definir os pontos de controle
pc[1,1] := 0; pc[1,2] := 0;  pc[1,3] := 0;
pc[2,1] := 1; pc[2,2] := 1;  pc[2,3] := -1;
pc[3,1] := 2; pc[3,2] := -1; pc[3,3] := -1;
pc[4,1] := 3; pc[4,2] := 0;  pc[4,3] := 0;

glClearColor(1,1,1,1);
glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);
glLoadIdentity;
glOrtho(-0.5,3.5,-1.5,1.5,-10,10);

// determinar 3 polinômios (Px, Py e Pz), definidos no domínio u=[0;1]
glMap1f(GL_MAP1_VERTEX_3,0,1,3,4,@pc);
glEnable(GL_MAP1_VERTEX_3);

// configurar as características da linha
glColor3f(1.0, 0.0, 0.0);
glLineWidth(3);
glMapGrid1f(20,0,1);         // definir o domínio u com 20 subdivisões
glEvalMesh1(GL_LINE,0,20);   // desenhar a curva com 20 subdivisões
glFlush;                     // atualizar
end;
```

O resultado apresentado é:

Figura 8.1: Curva de Bézier

8.1-) Elabore um procedimento que represente graficamente uma curva de Bézier, descrita pelos seguintes pontos de controle: $[1 \quad 1 \quad 0]$, $[1 \quad 3 \quad 1]$, $[3 \quad 3 \quad -1]$, $[3 \quad 1 \quad 0]$ e $[2 \quad 2 \quad 0]$. Devem ser desenhados os pontos de controle, bem como os segmentos de reta tracejados que os interligam.

8.2. Superfícies de Bézier

De maneira análoga à curva de Bézier, podem-se descrever as superfícies de Bézier por funções calculadas a partir de um conjunto de pontos de controle. Computacionalmente, os pontos de controle podem ser definidos, por exemplo, por uma variável indexada tridimensional contendo o número de pontos de controle em cada uma das direções paramétricas **u** e **v**, além da dimensão do espaço (no caso 3: **x**, **y** e **z**).

```
const npu = 3;  // utilizar no máximo np = 8
      npv = 3;
var pt_ctrl: array[1..npu,1..npv,1..3] of GLfloat;
```

Para que o *evaluator* bidimensional (direções paramétricas **u** e **v**) seja definido, utiliza-se a instrução **`glMap2*( )`**.

```
glMap2{f d}( TIPO_PC, UMIN, UMAX, AGRUP_U, Q_PC_U, VMIN, VMAX,
                  AGRUP_V, Q_PC_V, @PONTOS );
```

O primeiro parâmetro da instrução identifica qual é o significado dos pontos de controle. De maneira semelhante a que foi utilizada para representar as curvas de Bézier, será utilizada a constante **`GL_MAP2_VERTEX_3`**.

Os dois valores seguintes representam o escopo do parâmetro **u**, ou seja, $u \in [u_{min}; u_{max}]$.

O quarto parâmetro indica a distância entre os valores da coordenada x_i e a coordenada x_{i+1} relacionadas à direção paramétrica **u**.

O parâmetro seguinte indica a quantidade de pontos de controle que serão considerados na direção paramétrica **u**.

Os próximos quatro valores possuem características semelhantes aos quatro anteriores, entretanto relacionados à direção paramétrica **v**.

No último parâmetro deve ser indicado o endereço da primeira coordenada do primeiro ponto de controle, definidos previamente.

Para o exemplo apresentado anteriormente, temos a seguinte definição do *evaluator.*

```
glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3*npv, npu, 0, 1, 3, npv, @pt_ctrl);
```

Para habilitar o *evaluator* deve-se utilizar a instrução **`glEnable( )`**, utilizando o mesmo parâmetro referente ao significado dos pontos de controle (**`GL_MAP2_VERTEX_3`**).

```
glEnable( GL_MAP2_VERTEX_3 );
```

Para definir o domínio nas direções paramétricas **u** e **v**, utiliza-se a instrução **`glMapGrid2*( )`**. Esta instrução necessita de três parâmetros para cada direção: a quantidade de subdivisões do domínio e seus limites inferior e superior, normalmente [0; 1].

```
glMapGrid2{f d}( SUBDIVISOES_U, UMIN, UMAX, SUBDIVISOES_V, VMIN, VMAX );
```

Para desenhar a superfície, utiliza-se a instrução **`glEvalMesh2( )`**, cujos parâmetros representam o tipo de elemento utilizado no traçado (**`GL_POINT`** ou **`GL_LINE`** ou **`GL_FILL`**) e os extremos do domínio de **u** e **v**, em subdivisões:

```
glEvalMesh2( TIPO_ELEMENTO, 0, SUBDIVISOES_U, 0, SUBDIVISOES_V );
```

Quando a superfície for preenchida (**`GL_FILL`**) e houver alguma fonte de luz, é necessário habilitar (**`glEnable( )`**) o uso dos vetores normais, associados à constante **`GL_AUTO_NORMAL`**.

Os exemplos a seguir utilizam 15 subdivisões do domínio [0; 1] para as duas direções paramétricas: o primeiro cria uma superfície utilizando apenas linhas (*wireframe*):

```
// superfície em wireframe
glMapGrid1f(15,0,1,15,0,1);
glEvalMesh1(GL_LINE,0,15,0,15);
```

O segundo desenha uma superfície renderizada e pronta para receber um modelo de iluminação:

```
// superfície renderizada
glMapGrid1f(15,0,1,15,0,1);
glEvalMesh1(GL_FILL,0,15,0,15);
glEnable(GL_AUTO_NORMAL);
```

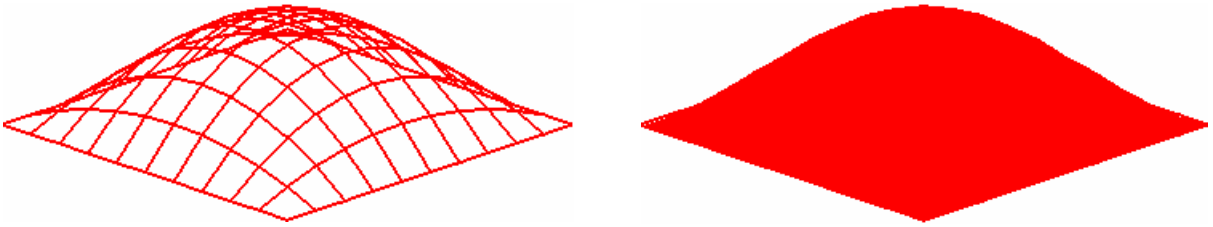
Exemplo 8.2-) Representar graficamente a superfície de Bézier definida pelos pontos: $[-4\ \ -4\ \ 0]$, $[-4\ \ 0\ \ 0]$, $[-4\ \ 4\ \ 0]$, $[0\ \ -4\ \ 0]$, $[0\ \ 0\ \ 9]$, $[0\ \ 4\ \ 0]$, $[4\ \ -4\ \ 0]$, $[4\ \ 0\ \ 0]$ e $[4\ \ 4\ \ 0]$.

Solução:

```
procedure desenha_superficie_de_Bezier;
var qu,qv: integer;
    pc: array[1..3,1..3,1..3] of GLfloat;
begin
// definição dos pontos de controle
pc[1,1,1] := -4;  pc[1,2,1] := -4;  pc[1,3,1] := -4;
pc[1,1,2] := -4;  pc[1,2,2] := +0;  pc[1,3,2] := +4;
pc[1,1,3] := +0;  pc[1,2,3] := +0;  pc[1,3,3] := +0;

pc[2,1,1] := +0;  pc[2,2,1] := +0;  pc[2,3,1] := +0;
pc[2,1,2] := -4;  pc[2,2,2] := +0;  pc[2,3,2] := +4;
pc[2,1,3] := +0;  pc[2,2,3] := +9;  pc[2,3,3] := +0;

pc[3,1,1] := +4;  pc[3,2,1] := +4;  pc[3,3,1] := +4;
pc[3,1,2] := -4;  pc[3,2,2] := +0;  pc[3,3,2] := +4;
pc[3,1,3] := +0;  pc[3,2,3] := +0;  pc[3,3,3] := +0;

glClearColor(1,1,1,1);
glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);
glLoadIdentity;
glOrtho(-6,6,-6,6,-15,15);
gluLookAt(1,1,0.5,0,0,0,0,0,1);

// determinar as funções, definidos no domínio u=[0;1] e v=[0;1]
glMap2f(GL_MAP2_VERTEX_3, 0,1,9,3, 0,1,3,3, @cp);
```

```
glEnable(GL_MAP2_VERTEX_3);

// configurar as características da linha
glColor3f(1,0,0);
glLineWidth(2);

// subdivisões da malha
qu := 10;
qv := 10;

glMapGrid2f(qu,0,1,qv,0,1);          // definir a malha dos domínios
glEvalMesh2(GL_lINE,0,qu,0,qv);      // desenhar a superfícies

// ou
//glEvalMesh2(GL_FILL,0,qu,0,qv);
//glEnable(GL_AUTO_NORMAL);

glFlush;                             // atualizar
end;
```

O resultado apresentado é:



GL_LINE **GL_FILL**

Figura 8.2: Superfície de Bézier

8.2-) Elabore um programa em Delphi que desenhe as superfícies de Bézier $S(u,v)$ para os seguintes pontos de controle.

a.) $P_{ij} = \begin{bmatrix} [-15 \;\; 0 \;\; 15]^T & [-15 \;\; 5 \;\; 5]^T & [-15 \;\; 5 \;\; -5]^T & [-15 \;\; 0 \;\; -15]^T \\ [-5 \;\; 5 \;\; 15]^T & [-5 \;\; 5 \;\; 5]^T & [-5 \;\; 5 \;\; -5]^T & [-5 \;\; 5 \;\; -15]^T \\ [5 \;\; 5 \;\; 15]^T & [5 \;\; 5 \;\; 5]^T & [5 \;\; 5 \;\; -5]^T & [5 \;\; 5 \;\; -15]^T \\ [15 \;\; 0 \;\; 15]^T & [15 \;\; 5 \;\; 15]^T & [15 \;\; 5 \;\; -5]^T & [15 \;\; 0 \;\; -15]^T \end{bmatrix}$

b.) $P_{ij} = \begin{bmatrix} [0 \;\; 0 \;\; 2]^T & [1 \;\; 0 \;\; 3]^T & [2 \;\; 0 \;\; 1]^T & [3 \;\; 0 \;\; 2]^T \\ [0 \;\; 3 \;\; 2]^T & [1 \;\; 3 \;\; 3]^T & [2 \;\; 3 \;\; 1]^T & [3 \;\; 3 \;\; 2]^T \\ [0 \;\; 4 \;\; 2]^T & [1 \;\; 4 \;\; 3]^T & [2 \;\; 4 \;\; 1]^T & [3 \;\; 4 \;\; 2]^T \end{bmatrix}$

9. Transformações geométricas

9.1. Introdução

As transformações geométricas de translação, rotação e mudança de escala são representadas computacionalmente sob a forma de uma matriz quadrada 4x4.
É comum que o efeito desejado somente possa ser obtido quando algumas matrizes de transformação forem aplicadas de maneira sucessiva, ou seja, multiplicando-as seqüencialmente. Com a biblioteca, não é necessário criar sub-rotinas que calculem a multiplicação de matrizes. Ao executar as instruções que serão apresentadas, a multiplicação com a matriz anterior (novamente a máquina de estados!) é feita automaticamente.

9.2. Preparando as matrizes de transformação

Lembrando que a biblioteca OpenGL permite manipular as características das matrizes de projeção ou transformação, é necessário selecionar a matriz desejada para que esta possa ser alterada. Como visto anteriormente, a instrução **`glMatrizMode( )`** permite selecionar a matriz referenciada pelo parâmetro. A constante associada à matriz de transformação geométrica é **`GL_MODELVIEW`**.

```
glMatrizMode( GL_MODELVIEW );
```

Deve-se lembrar que o conceito de máquina de estado é aplicado na matriz de transformação geométrica. Desta maneira deve-se garantir que as multiplicações resultem na matriz desejada.
Para garantir que o resultado seja o esperado, deve-se impor uma condição inicial à matriz de transformação geométrica: esta deve ser uma matriz identidade 4x4. Para tanto, utiliza-se a instrução **`glLoadIdentity`**.
Assim, antes de aplicar uma alteração na matriz de transformação geométrica, deve-se selecioná-la e impor que seja uma matriz identidade.

```
glMatrizMode( GL_MODELVIEW );
glLoadIdentity;
```

Como resultado, a matriz será:

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Um cuidado adicional deve ser tomado com estas instruções, uma vez que as matrizes serão constantemente selecionadas e pode-se aplicar a instrução **`glLoadIdentity`** por engano em uma matriz fazendo com que retorne a condição de matriz identidade.
Uma vez que a matriz de transformação geométrica está selecionada e com valor inicial imposto (matriz identidade) as transformações de translação, rotação e mudança de escala podem ser aplicadas.

9.3. Translação

Para alterar a posição dos próximos elementos que serão desenhados, utiliza-se a instrução **`glTranslate*( )`**.

```
glTranslate{f d}( DX, DY, DZ );
```

Na execução desta instrução a matriz de transformação geométrica da OpenGL é multiplicada por:

$$T(dx, dy, dz) = \begin{bmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

O trecho do programa a seguir exemplifica o uso desta instrução:

```
...
glBegin(GL_LINES);
  glVertex3f(0,0,0);
  glVertex3f(1,2,1);
glEnd;

glTranslate(1,-2,0);

glBegin(GL_LINES);
  glVertex3f(0,0,0);
  glVertex3f(1,2,1);
glEnd;
...
```

O resultado da execução deste trecho é:

Figura 9.1: Translação

Deve-se notar que as retas desenhadas possuem as mesmas coordenadas, entretanto, com a mudança do referencial, o resultado mostra duas retas distintas.

Observação: Este efeito mostra que os vértices dos objetos podem ser definidos exclusivamente no sistema de coordenadas do objeto e estes modelos podem ser distribuídos (rotacionados ou sofrer alteração no tamanho) sem que o programador preocupe-se com os novos vértices.

9.4. Rotação

Para rotacionar os elementos que serão desenhados, utiliza-se a instrução **`glRotate*( )`**.

```
glRotate{f d}( ANGULO, X, Y, Z );
```

Esta instrução rotaciona os elementos de um ângulo, em **graus**, no sentido anti-horário em relação ao eixo representado pelo vetor definido pela origem do sistema e o ponto $[X \quad Y \quad Z]$. Desta maneira, os valores de **X**, **Y** e **Z** podem assumir qualquer valor real, entretanto é comum utilizar apenas os valores **0** e **1**, uma vez que apenas a direção é importante.

Quando se utiliza a rotação em relação à apenas um dos eixos, as seguintes matrizes são utilizadas.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & -\text{sen}(\alpha) & 0 \\ 0 & \text{sen}(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} \cos(\beta) & 0 & \text{sen}(\beta) & 0 \\ 0 & 1 & 0 & 0 \\ -\text{sen}(\beta) & 0 & \cos(\beta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} \cos(\theta) & -\text{sen}(\theta) & 0 & 0 \\ \text{sen}(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```
glRotate(alfa,1,0,0);   glRotate(beta,0,1,0);   glRotate(teta,0,0,1);
```

Quando o eixo de rotação não coincidir com algum vetor da base canônica, a matriz de rotação é calculada por:

$$v = \begin{pmatrix} X & Y & Z \end{pmatrix}^T \qquad u = \frac{v}{|v|} = \begin{pmatrix} X' & Y' & Z' \end{pmatrix}^T$$

$$S = \begin{bmatrix} 0 & -Z' & Y' \\ Z' & 0 & -X' \\ -Y' & X' & 0 \end{bmatrix}$$

$$M = u \cdot u^T + \cos(\text{ANGULO}) \cdot \left(I - uu^T\right) + \text{sen}(\text{ANGULO}) \cdot S$$

$$R = \begin{bmatrix} & & & 0 \\ & M & & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

O trecho do programa a seguir exemplifica o uso desta instrução:

```
...
glBegin(GL_LINES);
  glVertex3f(0,0,0);
  glVertex3f(1,1,0);
glEnd;

glRotate(30,0,0,1); // rotação de 30° em relação ao eixo Z

glBegin(GL_LINES);
  glVertex3f(0,0,0);
  glVertex3f(1,1,0);
glEnd;
...
```

O resultado da execução deste trecho é:

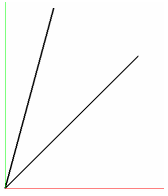


Figura 9.2: Rotação

9.5. Mudança de escala

Para alterar o tamanho dos elementos que serão desenhados, utiliza-se a instrução **`glScale*( )`**.

```
glScale{f d}( SX, SY, SZ );
```

Na execução desta instrução a matriz de transformação geométrica da OpenGL é multiplicada por:

$$S(Sx, Sy, Sz) = \begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

No caso dos parâmetros serem negativos, ocorre o espelhamento.
O trecho do programa a seguir exemplifica o uso desta instrução:

```
...
glBegin(GL_LINES);
  glVertex3f(0,0,0);
  glVertex3f(1,0.5,2);
glEnd;

glScale(1,1,0.5);

glBegin(GL_LINES);
  glVertex3f(0,0,0);
  glVertex3f(1,0.5,2);
glEnd;
...
```

O resultado da execução deste trecho é:

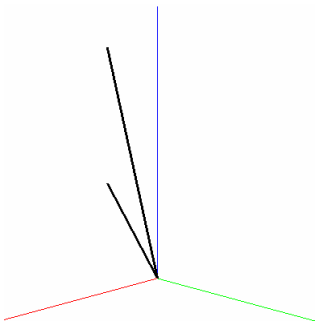


Figura 9.3: Mudança de escala

9.6. Manipulação da pilha

Como pode observado, quando uma instrução relacionada às transformações geométricas é executada, a matriz de transformação da OpenGL é alterada. Entretanto, é comum aplicar alguma alteração na matriz de transformação temporariamente, desejando retornar ao estado anterior após desenhar algum objeto.
Para duplicar a matriz atual, ou seja, adicionar uma cópia no topo da pilha, utiliza-se a instrução **`glPushMatrix`**. Para remover a matriz do topo da pilha utiliza-se **`glPopMatrix`**.

O trecho a seguir mostra o uso da pilha, manipulando a matriz de transformação (rotação de 30° em torno do eixo **Z**) e desenhando o primeiro elemento. Depois é criada uma cópia da matriz que sofrerá alteração (mudança de escala) para que o triângulo possa ser desenhado. Quando a instrução **`glPopMatrix`** é executada a matriz do topo da pilha (mudança de escala) é destruída, retornando para a situação anterior (rotação de 30°).

```
...
glMatrixMode(GL_MODELVIEW);  // seleciona a matriz de transformação
glRotate(30,0,0,1);          // rotação de 30° em torno de Z
glBegin(GL_LINES);
  glVertex3f(0,0,0);
  glVertex3f(1,1,0);
glEnd;

glPushMatrix;                // duplica a matriz atual
  glLoadIdentity;            // define a matriz como identidade
  glScale(1.5,0.5,1);        // mudança de escala na nova matriz
  glBegin(GL_TRIANGLES);
    glVertex3f(0,0,0);
    glVertex3f(1,0,0);
    glVertex3f(0.5,0.8,0);
  glEnd;
glPopMatrix;                 // destrói a matriz atual

// a matriz que está no topo da pilha é a matriz de rotação
glTranslate(0.5,0,0);        // translação
glBegin(GL_LINES);
  glVertex3f(0,0,0);
  glVertex3f(1,1,0);
glEnd;
...
```

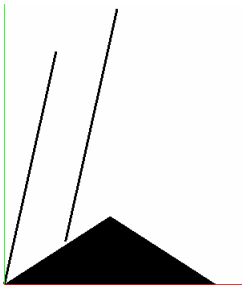


Figura 9.4: Manipulação da pilha

9.1-) Represente graficamente o logotipo da Mauá em três dimensões, sendo que o lado do quadrado maior e profundidade do sólido devem ser informados pelo usuário. Utilize as transformações geométricas de rotação e mudança de escala para simplificar o processo.

10. Atualização do *Framebuffer*

Todas as instruções apresentadas não afetam a imagem do monitor diretamente. Para que a imagem do *framebuffer* seja finalizada e exibida, deve-se utilizar a instrução **`glFlush`**. Nos exemplos apresentados anteriormente, a última instrução do código era **`glFlush`**.
Para criar animações, as imagens não podem ser atualizadas pela instrução **`glFlush`** pois sua atualização demora tempo suficiente para que os olhos percebam que a tela foi apagada e novamente desenhada, ou seja, a imagem "pisca".
Para solucionar o problema, utiliza-se o recurso ***Double Buffer***. No caso da configuração do *pixel format descriptor* (*pfd*) na criação do formulário, acrescenta-se esta opção ao **`dwFlags`** da seguinte maneira:

```
dwFlags := PFD_DRAW_TO_WINDOW or PFD_DOUBLEBUFFER;
```

Quando o componente **OGLPanel** for utilizado, a configuração apresentada é substituída atribuindo o valor **TRUE** à característica **DoubleBuffer** pertencente à propriedade **pfdFlags**;
Uma vez que o *double buffer* está ativo, a instrução **`glFlush`** não deve ser utilizada.
Para atualizar a imagem utiliza-se a instrução **`SwapBuffers( )`**, indicando como parâmetro o *Device Context* utilizado:

```
SwapBuffers( Form1.glDC );
```

No caso em que o componente **OGLPanel** esteja sendo utilizado, a instrução que substitui **`glFlush`** é:

```
OGLPanel1.Swap;
```

Para que as imagens sejam alteradas, é comum utilizar um componente **Timer** para executar a rotina de desenho de tempos em tempos.

10.1-) Utilizando o logotipo desenhado no **exercício 9.1**, faça com que o observador tenha uma trajetória circular ao redor do modelo. O centro do objeto deve ser o centro das atenções do observador.

11. Modelo de iluminação local

11.1. Introdução

A biblioteca OpenGL permite com que o programador utilize efeitos de iluminação nos objetos modelados seguindo os conceitos da máquina de estados apresentado anteriormente. Desta maneira, as configurações desejadas são definidas antes da execução das instruções que representam graficamente os elementos do modelo.
Todos os cálculos envolvendo iluminação feitos pela OpenGL consideram apenas o modelo de iluminação local, ou seja, se o programador desejar efeitos como a radiosidade ou *Ray-Tracing* terá que executar todos os cálculos "manualmente".

11.2. Características dos materiais

As características dos materiais podem ser definidas utilizando a instrução **`glMaterial*( )`**. Como a biblioteca OpenGL utiliza os conceitos da máquina de estados, deve-se lembrar que todos os atributos alterados por esta instrução são válidas para todos os vértices que serão representados em seguida, até que uma nova característica seja definida.
A sintaxe mais geral desta instrução é apresentada a seguir, uma vez que diversas características podem ser manipuladas.

```
glMaterial{i f} ( orientacao_face, propriedade, valor );
glMaterial{i f}v ( orientacao_face, propriedade, @VETOR4 );
```

O primeiro parâmetro referencia-se à definição do vetor normal de cada face, ou seja, se a face foi definida por uma lista de vértices seguindo o sentido anti-horário (vetor normal para fora da superfície) ou não.
A tabela a seguir mostra a relação das constantes da OpenGL com o sentido de definição dos vértices.

Tabela 11.1: Direção do vetor normal

Sentido	Constante
Anti-horário	**`GL_FRONT`**
Horário	**`GL_BACK`**
Não sei / Ambos	**`GL_FRONT_AND_BACK`**

Como as faces que são construídas **normalmente** seguem o sentido anti-horário, utiliza-se a constante **`GL_FRONT`**.
É possível que a cor da superfície, definida pela instrução **`glColor*( )`**, tenha participação na reflexão dos raios luminosos. Para tal, habilita-se a propriedade **`GL_COLOR_MATERIAL`** da seguinte maneira:

```
glEnable ( GL_COLOR_MATERIAL );
```

Coeficiente de reflexão ambiente

O coeficiente de reflexão ambiente ($\mathbf{k}_a$) representa a fração da luz ambiente que será refletida. Este fator é caracterizado por valores pertences ao intervalo fechado $[0;1]$.

Deve-se lembrar que a luz pode ser decomposta em quatro canais (vermelho, verde, azul e alfa). Estes canais devem ser representados por esta grandeza.

```
ka[1] := ka_R;
ka[2] := ka_G;
ka[3] := ka_B;
ka[4] := ka_A;
```

A constante da OpenGL que representa o coeficiente de reflexão ambiente é **GL_AMBIENT**.

```
glMaterialfv ( orientacao_face, GL_AMBIENT, @ka );
```

Coeficiente de reflexão difusa

O coeficiente de reflexão difusa ($\mathbf{k_d}$) representa a fração da luz ambiente que será refletida. Deve-se lembrar que se a superfície possuir apenas este coeficiente de reflexão, classifica-se esta superfície como Lambertiana. Este fator é caracterizado por valores pertences ao intervalo fechado $[0;1]$.
Os quatro canais da cor devem ser definidos em uma variável indexada unidimensional.

```
kd[1] := kd_R;
kd[2] := kd_G;
kd[3] := kd_B;
kd[4] := kd_A;
```

A constante da OpenGL que representa o coeficiente de reflexão difusa é **GL_DIFUSE**.

```
glMaterialfv ( orientacao_face, GL_DIFUSE, @kd );
```

Coeficiente de reflexão especular

O coeficiente de reflexão especular ($\mathbf{k_s}$) representa a fração da luz especular refletida pelo objeto. Este fator é caracterizado por valores pertences ao intervalo fechado $[0;1]$.
O coeficiente de reflexão especular é representado computacionalmente por um vetor do espaço RGBα.

```
ks[1] := ks_R;
ks[2] := ks_G;
ks[3] := ks_B;
ks[4] := ks_A;
```

A constante da OpenGL que associada ao coeficiente de reflexão especular é **GL_SPECULAR**.

```
glMaterialfv ( orientacao_face, GL_SPECULAR, @ks );
```

Expoente da reflexão especular

O expoente da reflexão especular (**n**) representa a fração da luz especular refletida pelo objeto. Este fator é caracterizado por valores pertences ao intervalo fechado $[0;128]$.
A constante da OpenGL que associada ao expoente da reflexão especular é **GL_SHININESS**.

```
glMaterialf ( orientacao_face, GL_SHININESS, n );
```

Coeficiente de emissão

É possível simular a emissão de luz a partir de uma superfície. Para tal, associa-se à superfície uma constante conhecida como coeficiente de emissão ($\mathbf{k_e}$). Este coeficiente é representado por um vetor com quatro valores pertences ao intervalo fechado $[0;1]$.

```
ke[1] := ke_R;
ke[2] := ke_G;
ke[3] := ke_B;
ke[4] := ke_A;
```

Para aplicar a emissividade sobre a superfície, deve-se utilizar a constante **GL_ EMISSION**.

```
glMaterialfv ( orientacao_face, GL_EMISSION, @ke );
```

11.3. Vetores normais

Para o cálculo da luminância ou brilhância é necessário conhecer o vetor normal da superfície no vértice analisado. Deve-se lembrar que as superfícies modeladas computacionalmente utilizam, na maioria dos casos, a aproximação planar.
Se o vetor normal uma face da aproximação planar não for definido, a biblioteca considera que o vetor normal de cada vértice é o próprio vetor normal da face que os contém, conforme apresentado no item 7.4.
Para garantir que todos os vetores utilizados pela OpenGL possuam modulo igual à unidade, habilita-se a característica **GL_NORMALIZE**, utilizando a seguinte instrução:

```
glEnable ( GL_NORMALIZE );
```

Entretanto, é possível definir o vetor normal de um vértice ou até mesmo um conjunto de vértices. Para tal, utiliza-se a instrução **glNormal*( )**. Lembrando que a OpenGL utiliza os conceitos da máquina de estados, assim que a instrução **glNormal*( )** for utilizada, todos os vértices e faces representadas posteriormente possuirão o mesmo vetor normal até que a instrução seja utilizada novamente, alterando os vetores normais dos vértices e faces que serão desenhados em seguida.
As seguintes sintaxes são válidas:

```
glNormal3{b s i f d} ( Nx, Ny, Nz );

glNormal3{b s i f d}v ( @VETOR3 );
```

Os parâmetros **Nx**, **Ny** e **Nz** são as componentes que definem o vetor normal, no sistema de coordenadas do mundo.
O efeito da instrução **glNormal*( )** somente pode ser observado quando o modelo de iluminação for utilizado.

11.4. Tipos de fonte de luz

Dentre os tipos de fonte de luz que foram apresentados, a OpenGL disponibiliza três tipos: a fonte pontual, a fonte distante e a fonte refletora (*spot*).
As três fontes podem ser definidas pela mesma instrução: **glLight*( )**. Devido à abrangência desta instrução, seus parâmetros serão apresentados associados às grandezas e propriedades as quais estão relacionados.

A sintaxe mais geral para a instrução **`glLight*( )`** é a seguinte:

```
glLight{i f} ( identificao_fonte, propriedade, valor );
glLight{i f}v ( identificao_fonte, propriedade, @VETOR3 );
glLight{i f}v ( identificao_fonte, propriedade, @VETOR4 );
```

A principal diferença entre as sintaxes está diretamente ligada às características da propriedade, ou seja, se a propriedade for um valor escalar, utiliza-se a primeira sintaxe.

No caso em que a propriedade for definida por um vetor, utiliza-se uma das duas últimas instruções, dependendo exclusivamente da quantidade de valores associados ao vetor.
A biblioteca OpenGL permite que sejam utilizadas até oito fontes de luz distintas ao mesmo tempo. Para tal, identificam-se as fontes utilizando as constantes de **`GL_LIGHT0`** a **`GL_LIGHT7`** no primeiro parâmetro. Nos exemplos a seguir todas as configurações serão aplicadas na primeira fonte de luz, ou seja, **`GL_LIGHT0`**.

Fonte pontual ou puntiforme

Este tipo de fonte emite raios luminosos em todas as direções, a partir de um ponto conhecido. Este ponto deve ser representado computacionalmente por um vetor, considerando a coordenada homogênea.

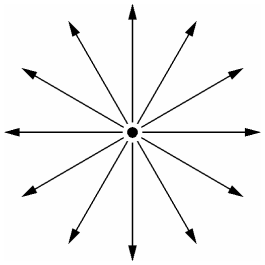


Figura 11.1: Fonte de luz pontual

Para definir a **posição** desta fonte, atribui-se a um vetor os valores das coordenadas tridimensionais, em coordenadas homogêneas. Deve-se lembrar que um ponto possui a coordenada W = 1;

```
Posicao[1] := Xfonte;
Posicao[2] := Yfonte;
Posicao[3] := Zfonte;
Posicao[4] := 1;
```

Uma vez definido o ponto que caracteriza a posição da fonte, aplica-se esta propriedade ao modelo da OpenGL utilizando a constante **`GL_POSITION`**, como mostrado a seguir.

```
glLightfv ( GL_LIGHT0, GL_POSITION, @Posicao );
```

Fonte distante

O fluxo luminoso flui de maneira uniforme em uma direção específica, simulando os raios provenientes de uma fonte como, por exemplo, o sol. Neste tipo de fonte não é considerada a atenuação devido à distância.

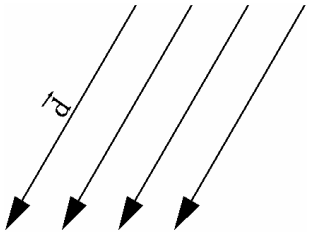


Figura 11.2: Fonte de luz distante

Para definir a **direção** desta fonte, atribui-se a uma variável indexada contendo os valores das coordenadas tridimensionais do vetor que caracteriza estes raios luminosos, em coordenadas homogêneas. Deve-se lembrar que um vetor possui a coordenada W = 0;

```
Direcao[1] := Xfonte;
Direcao[2] := Yfonte;
Direcao[3] := Zfonte;
Direcao[4] := 0;
```

Uma vez definida a direção da fonte, aplica-se esta propriedade ao modelo da OpenGL utilizando a constante **`GL_POSITION`**. Embora não exista um ponto que emite a radiação, o vetor diretor é distinguido de um ponto pela componente W = 0.

```
glLightfv ( GL_LIGHT0, GL_POSITION, @Direcao );
```

Fonte refletora (*spot*)

Este tipo de fonte simula um feixe cônico apontado para uma direção $\vec{d}$. Possui um ângulo de abertura q_{max} em relação ao eixo de simetria.

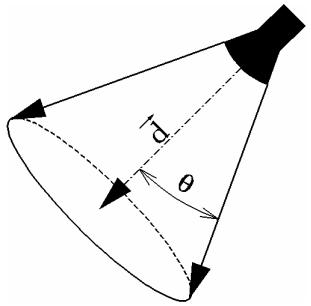


Figura 11.3: Fonte de luz *spot*

Para definir a **posição** desta fonte, atribui-se a um vetor os valores das coordenadas tridimensionais, em coordenadas homogêneas. Deve-se lembrar que um ponto possui a coordenada W = 1;

```
Posicao[1] := Xfonte;
Posicao[2] := Yfonte;
Posicao[3] := Zfonte;
Posicao[4] := 1;
```

Uma vez definido o ponto que caracteriza a posição da fonte, aplica-se esta propriedade ao modelo da OpenGL utilizando a constante **`GL_POSITION`**, como mostrado a seguir.

```
glLightfv ( GL_LIGHT0, GL_POSITION, @Posicao );
```

Outra característica da luz refletora consiste na **direção** que será iluminada. Desta forma, deve ser preenchido um vetor contendo as três coordenadas do vetor diretor. Note que este vetor não deve ser representado em coordenadas homogêneas

```
Direcao[1] := DirX;
Direcao[2] := DirY;
Direcao[3] := DirZ;
```

Para aplicar a direção de iluminação ao modelo da OpenGL, utiliza-se a constante **GL_SPOT_DIRECTION**, como mostrado a seguir.

```
glLightfv ( GL_LIGHT0, GL_SPOT_DIRECTION, @Direcao );
```

O **ângulo** de abertura, em graus, pode ser definido utilizando:

```
glLightf ( GL_LIGHT0, GL_SPOT_CUTOFF, angulo );
```

Este ângulo deve pertencer ao intervalo fechado $[0;90]$, em graus. Caso o ângulo seja $\theta = 180°$, a fonte comporta-se como uma fonte pontual.
A última característica da fonte refletora consiste no **expoente** associado ao co-seno ângulo de abertura. Quanto maior for este expoente, mais concentrada será a luz no interior da área iluminada. Este valor deve pertencer ao intervalo $0 \le n \le 128$.

$$f_{dec} = \cos^{n}(\theta)$$

```
glLightf ( GL_LIGHT0, GL_SPOT_EXPONENT, n );
```

11.5. Componentes da luz

Deve-se lembrar que o modelo de iluminação local utiliza três abordagens para definir a luz: a luz ambiente, a luz difusa e a luz especular.
Da mesma maneira que a definição do tipo da luz utilizou a instrução **glLight*( )**, os atributos das componentes ambiente, difusa e especular também deverão utilizar esta instrução.

Luz ambiente

A luz ambiente considera que uma fonte de luz, sem posição definida, ilumina igualmente todos os pontos da cena. Esta suposição é valida, pois esta luz será refletida diversas vezes por vários elementos que compõem a cena, caracterizando esta iluminação uniforme.
Sabendo que $I = k_a \cdot I_a$, deve-se definir as componentes vermelha, verde, azul e transparente da brilhância $\mathbf{I_a}$.

```
Luz_Ambiente[1] := Ia_R;
Luz_Ambiente[2] := Ia_G;
Luz_Ambiente[3] := Ia_B;
Luz_Ambiente[4] := Ia_A;
```

Para aplicar estas constantes ao modelo de iluminação da OpenGL, utiliza-se a constante **GL_AMBIENT**.

```
glLightfv ( GL_LIGHT0, GL_AMBIENT, @Luz_Ambiente );
```

Um cuidado a ser tomado consiste no fato de que, mesmo com todas as luzes desativadas, a biblioteca considera que uma quantidade mínima de luz ambiente esteja iluminado a cena.
Para garantir que não haja interferência desta componente, deve-se definir uma luz de intensidade nula e depois aplicá-la à luz ambiente padrão:

```
Luz_Ambiente_Padrao[1] := 0;
Luz_Ambiente_Padrao[2] := 0;
Luz_Ambiente_Padrao[3] := 0;
Luz_Ambiente_Padrao[4] := 1;
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, @Luz_Ambiente_Padrao);
```

Luz difusa

A componente difusa da luz é refletida igualmente para todas as direções. Deve-se lembrar que as superfícies que possuem tais características são denominadas superfícies Lambertianas.

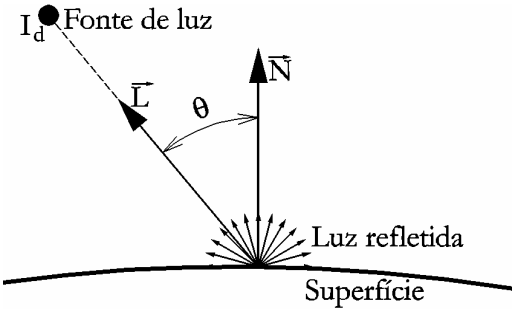


Figura 11.4: Iluminação difusa

Sabendo que $I = k_d \cdot I_d \cdot (\vec{N} \cdot \vec{L}) = k_d \cdot I_d \cdot \cos(\theta)$, devem ser definidas as componentes vermelha, verde, azul e transparente da brilhância $\mathbf{I_d}$.

```
Luz_Difusa[1] := Id_R;
Luz_Difusa[2] := Id_G;
Luz_Difusa[3] := Id_B;
Luz_Difusa[4] := Id_A;
```

Para aplicar estas constantes ao modelo de iluminação da OpenGL, utiliza-se a constante **`GL_DIFFUSE`**.

```
glLightfv ( GL_LIGHT0, GL_DIFFUSE, @Luz_Difusa );
```

Luz especular

A componente especular pode ser observada em superfícies brilhantes, polidas ou lustradas. O efeito visual corresponde a uma região de alta concentração de luz. Esta região possui a mesma cor da componente principal da luz proveniente da fonte. Quando se observa este "ponto de brilho" nota-se que sua posição é diferente para observadores posicionados em locais distintos.

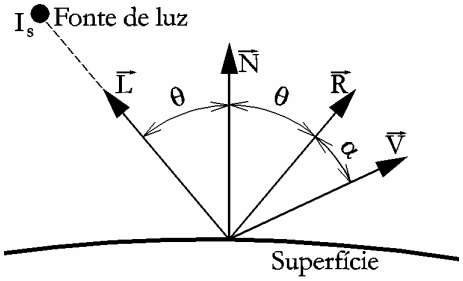


Figura 11.5: Modelo de Phong para a iluminação especular

O cálculo da componente especular foi modelado por Phong e pode ser descrito pela seguinte equação: $I = k_s \cdot I_s \cdot (\vec{V} \cdot \vec{R})^n$. Para realizar este cálculo devem-se definir as componentes vermelha, verde, azul e transparente da brilhância $\mathbf{I_s}$, além do expoente da reflexão especular, associado ao material e apresentado anteriormente.

```
Luz_Especular[1] := Is_R;
Luz_Especular[2] := Is_G;
Luz_Especular[3] := Is_B;
Luz_Especular[4] := Is_A;
```

Para aplicar estas constantes ao modelo de iluminação da OpenGL, utiliza-se a constante **GL_SPECULAR**.

```
glLightfv ( GL_LIGHT0, GL_SPECULAR, @Luz_Especular );
```

11.6. Atenuação atmosférica

A medida em a luz afasta-se da fonte, fatores atmosféricos atenuam sua intensidade. Esta atenuação é descrita pela seguinte equação:

$$f_{at} = \frac{1}{a \cdot d^2 + b \cdot d + c}$$

Os fatores de atenuação quadrático, linear e constante podem ser definidos associando os valores desejados às propriedades **GL_QUADRATIC_ATTENUATION**, **GL_LINEAR_ATTENUATION** e **GL_CONSTANT_ATTENUATION**, seguindo a sintaxe:

```
glLightf ( GL_LIGHT0, GL_QUADRATIC_ATTENUATION, a );

glLightf ( GL_LIGHT0, GL_LINEAR_ATTENUATION, b );

glLightf ( GL_LIGHT0, GL_CONSTANT_ATTENUATION, c );
```

11.7. Habilitando o modelo de iluminação e as fontes de luz

Uma vez que as características do material e das fontes de luz estão definidas, deve-se habilitar a exibição do modelo de iluminação local. Para tal, habilita-se a propriedade **GL_LIGHTNING** da seguinte maneira:

```
glEnable ( GL_LIGHTNING );
```

Esta instrução apenas permite que o modelo de iluminação local possa ser utilizado. Ou seja, como resultado, aparecerá uma tela preta, sem nenhum objeto desenhado.
Embora as características da fonte estejam definidas, a fonte não está ligada. Para habilitar cada uma das fontes, utiliza-se a instrução **`glEnable( )`** para cada fonte identificada pelas constantes pertencentes ao intervalo **`GL_LIGHT0`** a **`GL_LIGHT7`**.
O exemplo a seguir habilita a primeira fonte de luz: **`GL_LIGHT0`**.

```
glEnable ( GL_LIGHT0 );
```

Para desabilitar a fonte de luz utiliza-se:

```
glDisable ( GL_LIGHT0 );
```

11.1-) Crie um aplicativo que permita aplicar o modelo de iluminação local sobre um plano $(Z = 0)$, uma esfera $\left(x^2 + y^2 + z^2 = 1\right)$ e uma superfície $\left(S(u,v) = e^{-u^2} \cdot e^{-v^2}\right)$. Utilize uma fonte de luz pontual e uma refletora, utilizando as seguintes características:

Pontual	**Valor**
Posição	$[0 \quad 0 \quad 2]^T$
I_a	$[1 \quad 0 \quad 0]^T$
I_d	$[0 \quad 1 \quad 0]^T$
I_s	$[0 \quad 0 \quad 1]^T$
a	2
b	0
c	0

Spot	**Valor**
Posição	$[0 \quad 0 \quad 2]^T$
Direção	$(0 \quad 0 \quad -1)^T$
θ	20°
n	60
I_a	$[1 \quad 0 \quad 0]^T$
I_d	$[0 \quad 1 \quad 0]^T$
I_s	$[0 \quad 0 \quad 1]^T$

11.2-) Aplique o modelo de iluminação na cena desenvolvida no seu trabalho de Computação Gráfica.

12. Transparência

12.1. Introdução

A transparência utilizada pela OpenGL é definida para um modelo simplificado, desconsiderando o efeito da refração. Ou seja, utiliza-se apenas uma combinação linear das cores dos elementos interceptados pelo raio de luz.
Deve-se lembrar que no espaço de cores utilizado, a quarta coordenada (a) representa a fração da luz pode ser transmitida através do material.

12.2. Aplicação da transparência

Um cuidado deve ser tomado ao aplicar a transparência em elementos que compõem uma cena na OpenGL. O primeiro passo consiste em representar todos os elementos opacos, independente da sua posição em relação à câmera.
Assim que os elementos opacos estiverem desenhados, o programador deve inserir na cena os elementos com transparência.
A OpenGL aplica a transparência utilizando a expressão desenvolvida por Kay e Greenberg:

$$\boxed{I = \alpha \cdot I_1 + (1-\alpha) \cdot I_2, \quad 0 \le \alpha \le 1}$$

Sabendo que:

- **I**: representa o vetor de cores resultante do *pixel* analisado;
- $\mathbf{I_1}$: representa o vetor de cores do elemento transparente;
- $\mathbf{I_2}$: representa o vetor de cores do elemento que está diretamente atrás do primeiro elemento.
- a: é o parâmetro que indica o índice de transparência do primeiro elemento.

Como pode ser observado, a brilhância do primeiro elemento possui uma função de concordância (*blending function*), no caso a. De maneira semelhante, a segunda brilhância também possui uma função ponderadora, no caso $1-a$.
Estas funções devem ser informadas à OpenGL, utilizando a instrução **`glBlendFunc`**. A função a está associada à constante **`GL_SRC_ALPHA`** enquanto que a função $1-a$ está associada à constante **`GL_ONE_MINUS_SRC_ALPHA`**.

```
glBlendFunc ( GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA );
```

Uma vez definidas as funções, deve-se habilitar o uso da transparência, associado à constante **`GL_BLEND`**.

```
glEnable ( GL_BLEND );
```

Quando o programador definir uma cor para os elementos que serão desenhados futuramente, o valor da quarta coordenada pode ser utilizado livremente, dentro do intervalo fechado $[0;1]$, ou $[0;255]$ para valores inteiros.

```
glColor4{i f} ( R, G, B, A );

glColor4{i f}v ( @VETOR4 );
```

Deve-se lembrar que a transparência deve ser desabilitada assim que os elementos transparentes forem desenhados, utilizando:

```
glDisable ( GL_BLEND );
```

12.1-) Aplique a transparência nas superfícies do **exercício 11.1** do capítulo anterior. A representação da superfície deve considerar um fator de transparência, variando de 0 a 100%.

13. Referências

JACOBS, J. Q. **Delphi developer´s guide to OpenGL**. Texas: Wordware Publishing. 1999 450p

KAY, D.S.; GREENBERG, D. Transparency for Computer Synthesized Images. In: SIGGRAPH'79, Chicago, 1979. **Anais**. p158-164.

PHONG, B. T. Illumination for Computer Generated Pictures. Communications of the ACM, v.18, n.6, p.311-317, 1975.

SCALCO, R. Introdução à Computação Gráfica – Notas de Aula. São Caetano do Sul, S.P. : E.E.M., 2005

WOO, M.; NEIDER J. DAVIS T. **OpenGL Programming Guide – The Official Guide to Learning OpenGL – Version 1.2**. 2 ed, Addison-Wesley Pub Co, 1996 704p

WRIGHT Jr, R. S.; SWEET M. R. **OpenGL SuperBible**. 2 ed, Waite Group Press, 1999

<www.uol.com.br/aurelio>. Acesso em 11.02.04

<www.opengl.org>. Acesso em 12.01.05

<www.sgi.com/software/opengl/>. Acesso em 11.02.04

<pyopengl.sourceforge.net/documentation/manual/gluPerspective.3G.html>. Acesso em 25.02.04

Anexo A – Resolução dos exercícios

7.1-) Elabore um procedimento de um programa em Delphi, utilizando a biblioteca OpenGL, para representar graficamente cada uma das superfícies a seguir:
a.) Um cilindro de raio 2 e altura 6.
Solução:

```
procedure desenha_cilindro;
var t,z,dt,dz: real;
begin
glClearColor(1,1,1,1);
glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);
glLoadIdentity;
glOrtho(-3,3,-3,8,-10,10);
gluLookAt(1,1,0.5,0,0,0,0,0,1);

glColor3f(0,0,0);
t := 0;
dt := 2*pi/20;
dz := 6/7;
repeat
  z := 0;
  repeat
    glBegin(GL_LINE_LOOP);
      glVertex3f(2*cos(t),2*sin(t),z);
      glVertex3f(2*cos(t+dt),2*sin(t+dt),z);
      glVertex3f(2*cos(t+dt),2*sin(t+dt),z+dz);
      glVertex3f(2*cos(t),2*sin(t),z+dz);
    glEnd;
    z := z + dz;
  until z > 6;
  t := t + dt;
until t > 2*pi;
glFlush;
end;
```

b.) Um prisma de altura **h** de base elíptica. As dimensões máximas da elipse são **2· a** e **2· b** nas direções **x** e **y**.
Solução:

```
procedure desenha_prisma_base_eliptica (a,b,h: real);
var t,z,dt,dz: real;
begin
glClearColor(1,1,1,1);
glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);
glLoadIdentity;
glOrtho(-3,3,-3,8,-10,10);
gluLookAt(1,1,0.5,0,0,0,0,0,1);

glColor3f(0,0,0);
t := 0;
dt := 2*pi/20;
dz := h/7;
repeat
  z := 0;
  repeat
    glBegin(GL_LINE_LOOP);
```

```
      glVertex3f(a*cos(t),b*sin(t),z);
      glVertex3f(a*cos(t+dt),b*sin(t+dt),z);
      glVertex3f(a*cos(t+dt),b*sin(t+dt),z+dz);
      glVertex3f(a*cos(t),b*sin(t),z+dz);
    glEnd;
    z := z + dz;
  until z > h;
  t := t + dt;
until t > 2*pi;
glFlush;
end;
```

c.) Um parabolóide de revolução, de altura **h**, cuja intersecção com o plano **XZ** é descrita pela função $z(x,y) = 2 \cdot x^2$ e a intersecção com o plano **YZ** é dada por $z(x,y) = 2 \cdot y^2$.

Solução:

```
procedure desenha (h: real);
var a,t,z,dt,dz: real;
begin
glClearColor(1,1,1,1);
glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);
glLoadIdentity;
glOrtho(-3,3,-3,8,-10,10);
gluLookAt(1,1,0.5,0,0,0,0,0,1);

glColor3f(0,0,0);
t := 0;
a := 2;
dt := 2*pi/20;
dz := h/10;
repeat
  z := 0;
  repeat
    glBegin(GL_LINE_LOOP);
      glVertex3f(z*cos(t),z*sin(t),a*sqr(z));
      glVertex3f(z*cos(t+dt),z*sin(t+dt),a*sqr(z));
      glVertex3f((z+dz)*cos(t+dt),(z+dz)*sin(t+dt),a*sqr(z+dz));
      glVertex3f((z+dz)*cos(t),(z+dz)*sin(t),a*sqr(z+dz));
    glEnd;
    z := z + dz;
  until z > h;
  t := t + dt;
until t > 2*pi;
glFlush;
end;
```

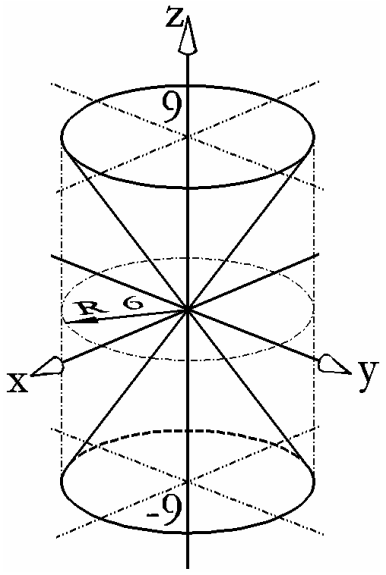
d.) Um elipsóide cujas dimensões máximas são **2· a**, **2· b** e **2· c**.

Solução:

```
procedure desenha_elipsoide(a,b,c:real);
var t,f,dt,df: real;
begin
glClearColor(1,1,1,1);
glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);

glLoadIdentity;
```

```
glOrtho(-3,3,-3,3,-10,10);
gluLookAt(1,1,0.5,0,0,0,0,0,1);

glColor3f(1,0,0);
t := 0;
dt := 2*pi/20;
df := pi/20;
repeat
  f := -pi/2;
  repeat
    glBegin(GL_LINE_LOOP);
      glVertex3fv(a*cos(t)*cos(f), b*sin(t)*cos(f),c*sin(f));
      glVertex3fv(a*cos(t+dt)*cos(f), b*sin(t+dt)*cos(f),c*sin(f));
      glVertex3fv(a*cos(t+dt)*cos(f+df),b*sin(t+dt)*cos(f+df),
                  c*sin(f+df));
      glVertex3fv(a*cos(t)*cos(f+df), b*sin(t)*cos(f+df),c*sin(f+df));
    glEnd;
    f := f + df;
  until f > pi/2;
  t := t + dt;
until t > 2*pi;
glFlush;
end;
```

e.) O cone da figura a seguir



Solução:

```
procedure desenha_cone;
var a,t,z,dt,dz: real;
begin
glClearColor(1,1,1,1);
glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);
glLoadIdentity;
glOrtho(-20,20,-20,20,-20,20);
gluLookAt(1,1,0.5,0,0,0,0,0,1);

glColor3f(0,0,0);
t := 0;
```

```
a := 1.5;
dt := 2*pi/20;
dz := 18/10;
repeat
  z := -9;
  repeat
    glBegin(GL_LINE_LOOP);
      glVertex3f(a*z*cos(t),a*z*sin(t),z);
      glVertex3f(a*z*cos(t+dt),a*z*sin(t+dt),z);
      glVertex3f(a*(z+dz)*cos(t+dt),a*(z+dz)*sin(t+dt),z+dz);
      glVertex3f(a*(z+dz)*cos(t),a*(z+dz)*sin(t),z+dz);
    glEnd;
    z := z + dz;
  until z > 9;
  t := t + dt;
until t > 2*pi;
glFlush;
end;
```

7.2-) Para determinar as equações de **x** e **y** no exercício anterior, foi utilizado o valor da cota **Z**. Quais seriam os efeitos em tais equações se o valor **Z** fosse substituído por $|Z|$?

Dica: Utilize cores diferentes para cada ponto da superfície.

Solução:

```
...
    glBegin(GL_LINE_LOOP);
      glVertex3f(a*abs(z)*cos(t),a*abs(z)*sin(t),z);
      glVertex3f(a*abs(z)*cos(t+dt),a*abs(z)*sin(t+dt),z);
      glVertex3f(a*abs(z+dz)*cos(t+dt),a*abs(z+dz)*sin(t+dt),z+dz);
      glVertex3f(a*abs(z+dz)*cos(t),a*abs(z+dz)*sin(t),z+dz);
    glEnd;
...
```

Há inversão de 180° o cone quando Z < 0.

7.3-) Indique quais são os pontos em comum e descreva uma maneira para otimizar o código, sem a necessidade de repeti-lo para cada uma das superfícies do exercício anterior.

Solução:

Como podemos resolver estes exercícios utilizando duas repetições, podemos especificar os parâmetros da instrução **`glvertex*( )`** como funções. Desta maneira, somente o código de cada função deve ser reescrito.

7.4-) Utilizando as instruções da OpenGL, represente graficamente o logotipo da Mauá em três dimensões, sendo que o lado do quadrado maior e profundidade do sólido devem ser informados pelo usuário (na dúvida, olhe para a parte de baixo desta folha!).

Solução:

```
Procedure desenha_Logo (Lado, Prof: real);
var divs, t, dt, dL1, dL2, L1, L2: real;
begin
glClearColor(1,1,1,1);
glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);

glLoadIdentity;
//glOrtho(-Lado,Lado,-Lado,Lado,-50,50);  // tamanho constante
glOrtho(-6,6,-6,6,-50,50);
gluLookAt(Lado,2*Lado,3*Lado,0,0,0,0,0,1);
```

```
//gluLookAt(Lado*cos(ang),Lado*sin(ang),Lado,0,0,0,0,0,1);

glColor3f(0,0,1);

glBegin(GL_QUADS);
  // quadrado maior
  glVertex3f(Lado/2,-Lado/2,0);
  glVertex3f(Lado/2,+Lado/2,0);
  glVertex3f(Lado/2,+Lado/2,Prof);
  glVertex3f(Lado/2,-Lado/2,Prof);

  glVertex3f(-Lado/2,-Lado/2,0);
  glVertex3f(-Lado/2,-Lado/2,Prof);
  glVertex3f(-Lado/2,+Lado/2,Prof);
  glVertex3f(-Lado/2,+Lado/2,0);

  glVertex3f(-Lado/2,Lado/2,0);
  glVertex3f(-Lado/2,Lado/2,Prof);
  glVertex3f(+Lado/2,Lado/2,Prof);
  glVertex3f(+Lado/2,Lado/2,0);

  glVertex3f(-Lado/2,-Lado/2,0);
  glVertex3f(+Lado/2,-Lado/2,0);
  glVertex3f(+Lado/2,-Lado/2,Prof);
  glVertex3f(-Lado/2,-Lado/2,Prof);

  // quadrado menor
  glVertex3f(Lado/(2*sqrt(2)),-Lado/(2*sqrt(2)),0);
  glVertex3f(Lado/(2*sqrt(2)),+Lado/(2*sqrt(2)),0);
  glVertex3f(Lado/(2*sqrt(2)),+Lado/(2*sqrt(2)),Prof);
  glVertex3f(Lado/(2*sqrt(2)),-Lado/(2*sqrt(2)),Prof);

  glVertex3f(-Lado/(2*sqrt(2)),-Lado/(2*sqrt(2)),0);
  glVertex3f(-Lado/(2*sqrt(2)),-Lado/(2*sqrt(2)),Prof);
  glVertex3f(-Lado/(2*sqrt(2)),+Lado/(2*sqrt(2)),Prof);
  glVertex3f(-Lado/(2*sqrt(2)),+Lado/(2*sqrt(2)),0);

  glVertex3f(-Lado/(2*sqrt(2)),Lado/(2*sqrt(2)),0);
  glVertex3f(-Lado/(2*sqrt(2)),Lado/(2*sqrt(2)),Prof);
  glVertex3f(+Lado/(2*sqrt(2)),Lado/(2*sqrt(2)),Prof);
  glVertex3f(+Lado/(2*sqrt(2)),Lado/(2*sqrt(2)),0);

  glVertex3f(-Lado/(2*sqrt(2)),-Lado/(2*sqrt(2)),0);
  glVertex3f(+Lado/(2*sqrt(2)),-Lado/(2*sqrt(2)),0);
  glVertex3f(+Lado/(2*sqrt(2)),-Lado/(2*sqrt(2)),Prof);
  glVertex3f(-Lado/(2*sqrt(2)),-Lado/(2*sqrt(2)),Prof);
glEnd;

t := 0;
divs := 200;
dt := 2*pi/divs;
repeat
  // cilindro maior
  glBegin(GL_QUADS);
    glVertex3f(Lado/2*cos(t),Lado/2*sin(t),0);
    glVertex3f(Lado/2*cos(t),Lado/2*sin(t),Prof);
```

```
    glVertex3f(Lado/2*cos(t+dt),Lado/2*sin(t+dt),Prof);
    glVertex3f(Lado/2*cos(t+dt),Lado/2*sin(t+dt),0);
  glEnd;
  // cilindro menor
  glBegin(GL_QUADS);
    glVertex3f(Lado/(2*sqrt(2))*cos(t),Lado/(2*sqrt(2))*sin(t),0);
    glVertex3f(Lado/(2*sqrt(2))*cos(t),Lado/(2*sqrt(2))*sin(t),Prof);

glVertex3f(Lado/(2*sqrt(2))*cos(t+dt),Lado/(2*sqrt(2))*sin(t+dt),Prof);

glVertex3f(Lado/(2*sqrt(2))*cos(t+dt),Lado/(2*sqrt(2))*sin(t+dt),0);
  glEnd;
  t := t + dt;
until t > 2*pi;

glColor3f(0.3,0.3,1);
t := -pi/4;
dt := pi/2/divs;
L1 := -Lado/2;
L2 := -Lado/(2*sqrt(2));
dL1 := Lado/divs;
dL2 := Lado/(sqrt(2))/divs;
repeat
  // tampa maior
  glBegin(GL_QUADS);
    glVertex3f(Lado/2,L1,Prof);
    glVertex3f(Lado/2,L1+dL1,Prof);
    glVertex3f(Lado/2*cos(t+dt),Lado/2*sin(t+dt),Prof);
    glVertex3f(Lado/2*cos(t),Lado/2*sin(t),Prof);
  glEnd;
  // tampar menor
  glBegin(GL_QUADS);
    glVertex3f(Lado/(2*sqrt(2)),L2,Prof);
    glVertex3f(Lado/(2*sqrt(2)),L2+dL2,Prof);

glVertex3f(Lado/(2*sqrt(2))*cos(t+dt),Lado/(2*sqrt(2))*sin(t+dt),Prof);
    glVertex3f(Lado/(2*sqrt(2))*cos(t),Lado/(2*sqrt(2))*sin(t),Prof);
  glEnd;
  // tampa maior
  glBegin(GL_QUADS);
    glVertex3f(Lado/2,L1,0);
    glVertex3f(Lado/2,L1+dL1,0);
    glVertex3f(Lado/2*cos(t+dt),Lado/2*sin(t+dt),0);
    glVertex3f(Lado/2*cos(t),Lado/2*sin(t),0);
  glEnd;
  // tampa menor
  glBegin(GL_QUADS);
    glVertex3f(Lado/(2*sqrt(2)),L2,0);
    glVertex3f(Lado/(2*sqrt(2)),L2+dL2,0);

glVertex3f(Lado/(2*sqrt(2))*cos(t+dt),Lado/(2*sqrt(2))*sin(t+dt),0);
    glVertex3f(Lado/(2*sqrt(2))*cos(t),Lado/(2*sqrt(2))*sin(t),0);
  glEnd;
  L1 := L1 + dL1;
  L2 := L2 + dL2;
  t := t + dt;
```

```
until t >= pi/4-dt/2;

t := pi/4;
L1 := Lado/2;
L2 := Lado/(2*sqrt(2));
repeat
  // tampa maior
  glBegin(GL_QUADS);
    glVertex3f(L1,Lado/2,Prof);
    glVertex3f(L1-dL1,Lado/2,Prof);
    glVertex3f(Lado/2*cos(t+dt),Lado/2*sin(t+dt),Prof);
    glVertex3f(Lado/2*cos(t),Lado/2*sin(t),Prof);
  glEnd;
  // tampa menor
  glBegin(GL_QUADS);
    glVertex3f(L2,Lado/(2*sqrt(2)),Prof);
    glVertex3f(L2-dL2,Lado/(2*sqrt(2)),Prof);

glVertex3f(Lado/(2*sqrt(2))*cos(t+dt),Lado/(2*sqrt(2))*sin(t+dt),Prof);
    glVertex3f(Lado/(2*sqrt(2))*cos(t),Lado/(2*sqrt(2))*sin(t),Prof);
  glEnd;
  // tampa maior
  glBegin(GL_QUADS);
    glVertex3f(L1,Lado/2,0);
    glVertex3f(L1-dL1,Lado/2,0);
    glVertex3f(Lado/2*cos(t+dt),Lado/2*sin(t+dt),0);
    glVertex3f(Lado/2*cos(t),Lado/2*sin(t),0);
  glEnd;
  // tampa menor
  glBegin(GL_QUADS);
    glVertex3f(L2,Lado/(2*sqrt(2)),0);
    glVertex3f(L2-dL2,Lado/(2*sqrt(2)),0);

glVertex3f(Lado/(2*sqrt(2))*cos(t+dt),Lado/(2*sqrt(2))*sin(t+dt),0);
    glVertex3f(Lado/(2*sqrt(2))*cos(t),Lado/(2*sqrt(2))*sin(t),0);
  glEnd;
  L1 := L1 - dL1;
  L2 := L2 - dL2;
  t := t + dt;
until t >= 3*pi/4-dt/2;

t := 3*pi/4;
L1 := Lado/2;
L2 := Lado/(2*sqrt(2));
repeat
  // tampa maior
  glBegin(GL_QUADS);
    glVertex3f(-Lado/2,L1,Prof);
    glVertex3f(-Lado/2,L1-dL1,Prof);
    glVertex3f(Lado/2*cos(t+dt),Lado/2*sin(t+dt),Prof);
    glVertex3f(Lado/2*cos(t),Lado/2*sin(t),Prof);
  glEnd;
  // tampa menor
  glBegin(GL_QUADS);
    glVertex3f(-Lado/(2*sqrt(2)),L2,Prof);
    glVertex3f(-Lado/(2*sqrt(2)),L2-dL2,Prof);
```

```
glVertex3f(Lado/(2*sqrt(2))*cos(t+dt),Lado/(2*sqrt(2))*sin(t+dt),Prof);
    glVertex3f(Lado/(2*sqrt(2))*cos(t),Lado/(2*sqrt(2))*sin(t),Prof);
  glEnd;
  // tampa maior
  glBegin(GL_QUADS);
    glVertex3f(-Lado/2,L1,0);
    glVertex3f(-Lado/2,L1-dL1,0);
    glVertex3f(Lado/2*cos(t+dt),Lado/2*sin(t+dt),0);
    glVertex3f(Lado/2*cos(t),Lado/2*sin(t),0);
  glEnd;
  // tampa menor
  glBegin(GL_QUADS);
    glVertex3f(-Lado/(2*sqrt(2)),L2,0);
    glVertex3f(-Lado/(2*sqrt(2)),L2-dL2,0);

glVertex3f(Lado/(2*sqrt(2))*cos(t+dt),Lado/(2*sqrt(2))*sin(t+dt),0);
    glVertex3f(Lado/(2*sqrt(2))*cos(t),Lado/(2*sqrt(2))*sin(t),0);
  glEnd;
  L1 := L1 - dL1;
  L2 := L2 - dL2;
  t := t + dt;
until t >= 5*pi/4-dt/2;

t := 5*pi/4;
L1 := -Lado/2;
L2 := -Lado/(2*sqrt(2));
repeat
  // tampa maior
  glBegin(GL_QUADS);
    glVertex3f(L1,-Lado/2,Prof);
    glVertex3f(L1+dL1,-Lado/2,Prof);
    glVertex3f(Lado/2*cos(t+dt),Lado/2*sin(t+dt),Prof);
    glVertex3f(Lado/2*cos(t),Lado/2*sin(t),Prof);
  glEnd;
  // tampa menor
  glBegin(GL_QUADS);
    glVertex3f(L2,-Lado/(2*sqrt(2)),Prof);
    glVertex3f(L2+dL2,-Lado/(2*sqrt(2)),Prof);

glVertex3f(Lado/(2*sqrt(2))*cos(t+dt),Lado/(2*sqrt(2))*sin(t+dt),Prof);
    glVertex3f(Lado/(2*sqrt(2))*cos(t),Lado/(2*sqrt(2))*sin(t),Prof);
  glEnd;
  // tampa maior
  glBegin(GL_QUADS);
    glVertex3f(L1,-Lado/2,0);
    glVertex3f(L1+dL1,-Lado/2,0);
    glVertex3f(Lado/2*cos(t+dt),Lado/2*sin(t+dt),0);
    glVertex3f(Lado/2*cos(t),Lado/2*sin(t),0);
  glEnd;
  // tampa menor
  glBegin(GL_QUADS);
    glVertex3f(L2,-Lado/(2*sqrt(2)),0);
    glVertex3f(L2+dL2,-Lado/(2*sqrt(2)),0);

glVertex3f(Lado/(2*sqrt(2))*cos(t+dt),Lado/(2*sqrt(2))*sin(t+dt),0);
```

```
    glVertex3f(Lado/(2*sqrt(2))*cos(t),Lado/(2*sqrt(2))*sin(t),0);
  glEnd;
  L1 := L1 + dL1;
  L2 := L2 + dL2;
  t := t + dt;
until t >= 7*pi/4-dt/2;

glFlush;
end;
```

8.1-) Elabore um procedimento que represente graficamente uma curva de Bézier, descrita pelos seguintes pontos de controle: $[1 \quad 1 \quad 0]$, $[1 \quad 3 \quad 1]$, $[3 \quad 3 \quad -1]$, $[3 \quad 1 \quad 0]$ e $[2 \quad 2 \quad 0]$. Devem ser desenhados os pontos de controle, bem como os segmentos de reta tracejados que os interligam.

Solução:

```
procedure desenha_curva_de_Bezier;
var pc: array[1..5,1..3] of GLfloat;
    nsi, i: integer;
begin
pc[1,1] := 1;  pc[1,2] := 1;  pc[1,3] := 0;
pc[2,1] := 1;  pc[2,2] := 3;  pc[2,3] := 1;
pc[3,1] := 3;  pc[3,2] := 3;  pc[3,3] := -1;
pc[4,1] := 3;  pc[4,2] := 1;  pc[4,3] := 0;
pc[5,1] := 2;  pc[5,2] := 2;  pc[5,3] := 0;

glClearColor(1,1,1,1);
glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);

glLoadIdentity;
glOrtho(-2,2,-3,1,-10,10);
gluLookAt(1,1,0.5,0,0,0,0,0,1);

glMap1f(GL_MAP1_VERTEX_3,0,1,3,5,@pc);
glEnable(GL_MAP1_VERTEX_3);

nsi := 30;  // número de sub intervalos
glColor3f(0,0,0);
glLineWidth(5);
glMapGrid1f(nsi,0,1);
glEvalMesh1(GL_LINE,0,nsi);

glColor3f(1.0, 0.0, 0.0);
glLineWidth(1);
glEnable(GL_LINE_STIPPLE);
glLineStipple(1,$00FF);
glBegin(GL_LINE_STRIP);
  for i := 1 to 5 do
    glVertex3fv(@pc[i]);
glEnd;
glDisable(GL_LINE_STIPPLE);

glPointSize(5);
glBegin(GL_POINTS);
  for i := 1 to 5 do
    glVertex3fv(@pc[i]);
glEnd;
```

```
glFlush;
end;
```

8.2-) Elabore um programa em Delphi que desenhe as superfícies de Bézier $S(u,v)$ para os seguintes pontos de controle.

a.) $P_{ij} = \begin{bmatrix} [-15 \;\; 0 \;\; 15]^T & [-15 \;\; 5 \;\; 5]^T & [-15 \;\; 5 \;\; -5]^T & [-15 \;\; 0 \;\; -15]^T \\ [-5 \;\; 5 \;\; 15]^T & [-5 \;\; 5 \;\; 5]^T & [-5 \;\; 5 \;\; -5]^T & [-5 \;\; 5 \;\; -15]^T \\ [5 \;\; 5 \;\; 15]^T & [5 \;\; 5 \;\; 5]^T & [5 \;\; 5 \;\; -5]^T & [5 \;\; 5 \;\; -15]^T \\ [15 \;\; 0 \;\; 15]^T & [15 \;\; 5 \;\; 15]^T & [15 \;\; 5 \;\; -5]^T & [15 \;\; 0 \;\; -15]^T \end{bmatrix}$

Solução:

```
procedure desenha_superficie_de_Bezier;
var pc: array[1..4,1..4,1..3] of GLfloat;
    nsi, i, j: integer;
begin

pc[1,1,1] := -15;  pc[1,1,2] := 0.0;  pc[1,1,3] := 15;
pc[2,1,1] := -15;  pc[2,1,2] := 5.0;  pc[2,1,3] := 5;
pc[3,1,1] := -15;  pc[3,1,2] := 5.0;  pc[3,1,3] := -5;
pc[4,1,1] := -15;  pc[4,1,2] := 0.0;  pc[4,1,3] := -15;

pc[1,2,1] := -5;   pc[1,2,2] := 5.0;  pc[1,2,3] := 15;
pc[2,2,1] := -5;   pc[2,2,2] := 5.0;  pc[2,2,3] := 5;
pc[3,2,1] := -5;   pc[3,2,2] := 5.0;  pc[3,2,3] := -5;
pc[4,2,1] := -5;   pc[4,2,2] := 5.0;  pc[4,2,3] := -15;

pc[1,3,1] := 5.0;  pc[1,3,2] := 5.0;  pc[1,3,3] := 15;
pc[2,3,1] := 5.0;  pc[2,3,2] := 5.0;  pc[2,3,3] := 5;
pc[3,3,1] := 5.0;  pc[3,3,2] := 5.0;  pc[3,3,3] := -5;
pc[4,3,1] := 5.0;  pc[4,3,2] := 5.0;  pc[4,3,3] := -15;

pc[1,4,1] := 15;   pc[1,4,2] := 0.0;  pc[1,4,3] := 15;
pc[2,4,1] := 15;   pc[2,4,2] := 5.0;  pc[2,4,3] := 5;
pc[3,4,1] := 15;   pc[3,4,2] := 5.0;  pc[3,4,3] := -5;
pc[4,4,1] := 15;   pc[4,4,2] := 0.0;  pc[4,4,3] := -15;

glClearColor(1,1,1,1);
glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);
glLoadIdentity;
glOrtho(-30,30,-20,25,-30,30);
gluLookAt(1,0.7,1,0,0,0,0,0,1);

glMap2f(GL_MAP2_VERTEX_3,0,1,3*4,4,0,1,3,4,@pc);
glEnable(GL_MAP2_VERTEX_3);

nsi := 7;  // número de sub intervalos
glColor3f(0,0,0);
glLineWidth(5);
glMapGrid2f(nsi,0,1,nsi,0,1);
glEvalMesh2(GL_LINE,0,nsi,0,nsi);
glFlush;
end;
```

b.) $P_{ij} = \begin{bmatrix} [0\ 0\ 2]^T & [1\ 0\ 3]^T & [2\ 0\ 1]^T & [3\ 0\ 2]^T \\ [0\ 3\ 2]^T & [1\ 3\ 3]^T & [2\ 3\ 1]^T & [3\ 3\ 2]^T \\ [0\ 4\ 2]^T & [1\ 4\ 3]^T & [2\ 4\ 1]^T & [3\ 4\ 2]^T \end{bmatrix}$

Solução:

```
procedure TForm1.Button1Click(Sender: TObject);
var pc: array[1..4,1..3,1..3] of GLfloat;
    nsi, i, j: integer;
begin
pc[1,1,1] := 0;  pc[1,1,2] := 0;  pc[1,1,3] := 2;
pc[2,1,1] := 1;  pc[2,1,2] := 0;  pc[2,1,3] := 3;
pc[3,1,1] := 2;  pc[3,1,2] := 0;  pc[3,1,3] := 1;
pc[4,1,1] := 3;  pc[4,1,2] := 0;  pc[4,1,3] := 2;

pc[1,2,1] := 0;  pc[1,2,2] := 3;  pc[1,2,3] := 2;
pc[2,2,1] := 1;  pc[2,2,2] := 3;  pc[2,2,3] := 3;
pc[3,2,1] := 2;  pc[3,2,2] := 3;  pc[3,2,3] := 1;
pc[4,2,1] := 3;  pc[4,2,2] := 3;  pc[4,2,3] := 2;

pc[1,3,1] := 0;  pc[1,3,2] := 4;  pc[1,3,3] := 2;
pc[2,3,1] := 1;  pc[2,3,2] := 4;  pc[2,3,3] := 3;
pc[3,3,1] := 2;  pc[3,3,2] := 4;  pc[3,3,3] := 1;
pc[4,3,1] := 3;  pc[4,3,2] := 4;  pc[4,3,3] := 2;

glClearColor(1,1,1,1);
glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);

glLoadIdentity;
glOrtho(-3,3,-1,3,-30,30);
gluLookAt(1,2,0.5,0,0,0,0,0,1);

glMap2f(GL_MAP2_VERTEX_3,0,1,3*3,4,0,1,3,3,@pc);
glEnable(GL_MAP2_VERTEX_3);

nsi := 10;  // número de sub intervalos
glColor3f(0,0,0);
glLineWidth(5);
glMapGrid2f(nsi,0,1,nsi,0,1);
glEvalMesh2(GL_LINE,0,nsi,0,nsi);
glFlush;
end;
```

9.1-) Represente graficamente o logotipo da Mauá em três dimensões, sendo que o lado do quadrado maior e profundidade do sólido devem ser informados pelo usuário. Utilize as transformações geométricas de rotação e mudança de escala para simplificar o processo.

Solução:

```
procedure desenha_Logo (Lado, Prof: real);
var contS, contR, divs, t, dt, dL1, dL2, L1, L2: real;
begin
glClearColor(1,1,1,1);
glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);

glLoadIdentity;
//glOrtho(-Lado,Lado,-Lado,Lado,-50,50);  // tamanho constante
```

```
glOrtho(-6,6,-6,6,-50,50);
gluLookAt(Lado,2*Lado,3*Lado,0,0,0,0,0,1);
//gluLookAt(Lado*cos(ang),Lado*sin(ang),Lado,0,0,0,0,0,1);

contS := 0;
repeat
  contR := 0;
  repeat
    glRotate(90,0,0,1);
    glColor3f(0,0,1);
    glBegin(GL_QUADS);
      // quadrado maior
      glVertex3f(Lado/2,0,0);
      glVertex3f(Lado/2,+Lado/2,0);
      glVertex3f(Lado/2,+Lado/2,Prof);
      glVertex3f(Lado/2,0,Prof);
      glVertex3f(0,Lado/2,0);
      glVertex3f(0,Lado/2,Prof);
      glVertex3f(+Lado/2,Lado/2,Prof);
      glVertex3f(+Lado/2,Lado/2,0);
    glEnd;
    t := 0;
    divs := 200;
    dt := 2*pi/divs;
    repeat
      // cilindro maior
      glBegin(GL_QUADS);
        glVertex3f(Lado/2*cos(t),Lado/2*sin(t),0);
        glVertex3f(Lado/2*cos(t),Lado/2*sin(t),Prof);
        glVertex3f(Lado/2*cos(t+dt),Lado/2*sin(t+dt),Prof);
        glVertex3f(Lado/2*cos(t+dt),Lado/2*sin(t+dt),0);
      glEnd;
      t := t + dt;
    until t > pi/2;

    glColor3f(0.3,0.3,1);
    t := 0;
    dt := pi/4/divs;
    L1 := 0;
    L2 := 0;
    dL1 := Lado/2/divs;
    dL2 := Lado/(2*sqrt(2))/divs;
    repeat
      // tampa maior
      glBegin(GL_QUADS);
        glVertex3f(Lado/2,L1,Prof);
        glVertex3f(Lado/2,L1+dL1,Prof);
        glVertex3f(Lado/2*cos(t+dt),Lado/2*sin(t+dt),Prof);
        glVertex3f(Lado/2*cos(t),Lado/2*sin(t),Prof);
      glEnd;
      // tampa maior
      glBegin(GL_QUADS);
        glVertex3f(Lado/2,L1,0);
        glVertex3f(Lado/2,L1+dL1,0);
        glVertex3f(Lado/2*cos(t+dt),Lado/2*sin(t+dt),0);
        glVertex3f(Lado/2*cos(t),Lado/2*sin(t),0);
```

```
      glEnd;
      L1 := L1 + dL1;
      L2 := L2 + dL2;
      t := t + dt;
    until t >= pi/4-dt/2{??};

    t := pi/4;
    L1 := Lado/2;
    L2 := Lado/(2*sqrt(2));
    repeat
      // tampa maior
      glBegin(GL_QUADS);
        glVertex3f(L1,Lado/2,Prof);
        glVertex3f(L1-dL1,Lado/2,Prof);
        glVertex3f(Lado/2*cos(t+dt),Lado/2*sin(t+dt),Prof);
        glVertex3f(Lado/2*cos(t),Lado/2*sin(t),Prof);
      glEnd;
      // tampa maior
      glBegin(GL_QUADS);
        glVertex3f(L1,Lado/2,0);
        glVertex3f(L1-dL1,Lado/2,0);
        glVertex3f(Lado/2*cos(t+dt),Lado/2*sin(t+dt),0);
        glVertex3f(Lado/2*cos(t),Lado/2*sin(t),0);
      glEnd;
      L1 := L1 - dL1;
      L2 := L2 - dL2;
      t := t + dt;
    until t >= pi/2-dt/2;
    contR := contR + 1;
  until contR = 4;
  contS := contS + 1;
  glScale(1/sqrt(2),1/sqrt(2),1);
until contS = 2;
glFlush;
end;
```

11.1-) Crie um aplicativo que permita aplicar o modelo de iluminação local sobre um plano $(Z = 0)$, uma esfera $\left(x^2 + y^2 + z^2 = 1\right)$ e uma superfície $\left(S(u,v) = e^{-u^2} \cdot e^{-v^2}\right)$. Utilize uma fonte de luz pontual e uma refletora, utilizando as seguintes características:

Pontual	**Valor**
Posição	$[0 \quad 0 \quad 2]^T$
I_a	$[1 \quad 0 \quad 0]^T$
I_d	$[0 \quad 1 \quad 0]^T$
I_s	$[0 \quad 0 \quad 1]^T$
a	2
b	0
c	0

Spot	**Valor**
Posição	$[0 \quad 0 \quad 2]^T$
Direção	$(0 \quad 0 \quad -1)^T$
θ	20°
n	60
I_a	$[1 \quad 0 \quad 0]^T$
I_d	$[0 \quad 1 \quad 0]^T$
I_s	$[0 \quad 0 \quad 1]^T$

12.1-) Aplique a transparência nas superfícies do **exercício 11.1** do capítulo anterior. A representação da superfície deve considerar um fator de transparência, variando de 0 a 100%.

Solução (11.1 e 12.1):

```
unit Modelos;
interface
procedure eixos;
procedure Plano;
procedure esfera;
procedure superficie;

implementation
uses OpenGL;

procedure eixos;
begin
glBegin(GL_LINES);
  glColor3f(1,0,0);  glVertex3f(0,0,0);  glVertex3f(2,0,0);
  glColor3f(0,1,0);  glVertex3f(0,0,0);  glVertex3f(0,2,0);
  glColor3f(0,0,1);  glVertex3f(0,0,0);  glVertex3f(0,0,2);
glEnd;
end;

procedure Plano;
var i,j : real;
begin
glBegin(GL_QUADS);
i := -1;
repeat
  j := -1;
  repeat
    glVertex3f(i,j,0);
    glVertex3f(i+1/20,j,0);
    glVertex3f(i+1/20,j+1/20,0);
    glVertex3f(i,j+1/20,0);
    j := j + 1/20;
  until j >= 1;
  i := i + 1/20;
until i >= 1;
glEnd;
end;

procedure esfera;
type TPonto3D = array [1..3] of GLfloat;
function e2c(r,t,f:real):TPonto3D;
  begin
  e2c[1] := r*cos(t)*sin(f);
  e2c[2] := r*sin(t)*sin(f);
  e2c[3] := r*cos(f);
  end;
var r,t,f,dt,df: real;
begin
dt := 2*pi/20;
df := pi/20;
r := 1;
```

```
t := 0;
repeat
  f := -pi;
  repeat
    glBegin(GL_QUADS);
      glVertex3f(e2c(r,t,f)[1],e2c(r,t,f)[2],e2c(r,t,f)[3]);
      glVertex3f(e2c(r,t+dt,f)[1],e2c(r,t+dt,f)[2],e2c(r,t+dt,f)[3]);
      glVertex3f(e2c(r,t+dt,f+df)[1],e2c(r,t+dt,f+df)[2],
                  e2c(r,t+dt,f+df)[3]);
      glVertex3f(e2c(r,t,f+df)[1],e2c(r,t,f+df)[2],e2c(r,t,f+df)[3]);
    glEnd;
    f := f + df;
  until f>= pi;
  t := t + dt;
until t >= 2*pi;
end;

procedure superficie;
type TPonto3D = array [1..3] of GLfloat;
function sup(x,y:real):real;
  begin
  sup := exp(-sqr(x))*exp(-sqr(y));
  end;
var x,y,dx,dy: real;
begin
dx := 2/20;
dy := 2/20;
x := -2;
repeat
  y := -2;
  repeat
    glBegin(GL_QUADS);
      glVertex3f(x,y,sup(x,y));
      glVertex3f(x+dx,y,sup(x+dx,y));
      glVertex3f(x+dx,y+dy,sup(x+dx,y+dy));
      glVertex3f(x,y+dy,sup(x,y+dy));
    glEnd;
    y := y + dy;
  until y>= 2;
  x := x + dx;
until x >= 2;
end;
end.
```

```
...
var
  Form1: TForm1;
  alpha: real;
  Pos, Ia, Id, Isp, La: array [1..4] of GLfloat;
  dir: array [1..3] of GLfloat;
  ang, n: GLfloat;
  ka, kd, ks, ke: array [1..4] of GLfloat;

implementation
uses modelos;
```

```
{$R *.dfm}

procedure desenha;
begin
glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);
glLoadIdentity;
glOrtho(-3,3,-3,3,-100,100);
gluLookAt(1,1,1,0,0,0,0,0,1);
eixos;
ka[1] := 0.1; ka[2] := 0.1; ka[3] := 0.1; ka[4] := 1;
kd[1] := 0.45; kd[2] := 0.45; kd[3] := 0.45; kd[4] := 1;
ks[1] := 0.5; ks[2] := 0.5; ks[3] := 0.5; ks[4] := 1;
ke[1] := 0.0; ke[2] := 0.0; ke[3] := 0.0; ke[4] := 1;
n := 27;
glMaterialfv(GL_FRONT, GL_AMBIENT, @ka);
glMaterialfv(GL_FRONT, GL_DIFFUSE, @kd);
glMaterialfv(GL_FRONT, GL_SPECULAR, @ks);
glMaterialfv(GL_FRONT, GL_EMISSION, @ke);
glMaterialf(GL_FRONT, GL_SHININESS, n);

glEnable(GL_ALPHA_TEST);
glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);
glEnable(GL_BLEND);
  case Form1.RadioGroup1.ItemIndex of
    0: begin
       glColor4f(1,1,1,alpha);
       Plano;
       end;
    1: begin
       glColor4f(1,1,1,alpha);
       esfera;
       end;
    2: begin
       glColor4f(1,1,1,alpha);
       superficie;
       end;
    end;
glDisable(GL_BLEND);
Form1.OGLPanel1.Swap;
end;

procedure TForm1.RadioGroup1Click(Sender: TObject);
begin
desenha;
end;

procedure TForm1.CheckBox1Click(Sender: TObject);
begin
if CheckBox1.Checked
  then glEnable(GL_LIGHT0)
  else glDisable(GL_LIGHT0);
desenha;
end;

procedure TForm1.CheckBox2Click(Sender: TObject);
begin
```

```
if CheckBox2.Checked
  then glEnable(GL_LIGHT1)
  else glDisable(GL_LIGHT1);
desenha;
end;

procedure TForm1.CheckBox3Click(Sender: TObject);
begin
if CheckBox3.Checked
  then glEnable(GL_LIGHT2)
  else glDisable(GL_LIGHT2);
desenha;
end;

procedure TForm1.SpinEdit1Change(Sender: TObject);
begin
alpha := (100- SpinEdit1.Value)/100;
desenha;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
La[1] := 0.0; La[2] := 0.0; La[3] := 0.0; La[4] := 1;
glLightModelfv(GL_LIGHT_MODEL_AMBIENT,@La);

ia[1] := 1; ia[2] := 0.0; ia[3] := 0.0; ia[4] := 1;
id[1] := 1; id[2] := 0.0; id[3] := 0.0; id[4] := 1;
isp[1] := 1; isp[2] := 0.0; isp[3] := 0.0; isp[4] := 1;
glLightfv(GL_LIGHT0, GL_AMBIENT, @Ia);
glLightfv(GL_LIGHT0, GL_DIFFUSE, @Id);
glLightfv(GL_LIGHT0, GL_SPECULAR, @Isp);
// definindo o tipo da fonte
Pos[1] := 0; Pos[2] := 0; Pos[3] := 2; Pos[4] := 1;
glLightfv(GL_LIGHT0, GL_POSITION, @Pos);
ia[1] := 0.0; ia[2] := 1; ia[3] := 0.0; ia[4] := 1;
id[1] := 0.0; id[2] := 1; id[3] := 0.0; id[4] := 1;
isp[1] := 0.0; isp[2] := 1; isp[3] := 0.0; isp[4] := 1;
glLightfv(GL_LIGHT1, GL_AMBIENT, @Ia);
glLightfv(GL_LIGHT1, GL_DIFFUSE, @Id);
glLightfv(GL_LIGHT1, GL_SPECULAR, @Isp);
// definindo o tipo da fonte
Pos[1] := 0; Pos[1] := 0; Pos[3] := 1; Pos[4] := 0;
glLightfv(GL_LIGHT1, GL_POSITION, @Pos);
ia[1] := 0.0; ia[2] := 0.0; ia[3] := 1; ia[4] := 1;
id[1] := 0.0; id[2] := 0.0; id[3] := 1; id[4] := 1;
isp[1] := 0.0; isp[2] := 0.0; isp[3] := 1; isp[4] := 1;
glLightfv(GL_LIGHT2, GL_AMBIENT, @Ia);
glLightfv(GL_LIGHT2, GL_DIFFUSE, @Id);
glLightfv(GL_LIGHT2, GL_SPECULAR, @Isp);
// definindo o tipo da fonte
Pos[1] := 0.5; Pos[2] := 0; Pos[3] := 3; Pos[4] := 1;
dir[1] := -0; dir[2] := 0; dir[3] := -1;
ang := 20;
n := 60;
glLightfv(GL_LIGHT2, GL_POSITION, @Pos);
glLightfv(GL_LIGHT2, GL_SPOT_DIRECTION, @dir);
```

```
glLightf(GL_LIGHT2, GL_SPOT_CUTOFF, ang);
glLightf(GL_LIGHT2, GL_SPOT_EXPONENT, n);
glLightf(GL_LIGHT0,GL_QUADRATIC_ATTENUATION,2);
glLightf(GL_LIGHT0,GL_LINEAR_ATTENUATION,0);
glLightf(GL_LIGHT0,GL_CONSTANT_ATTENUATION,0);

glEnable(GL_LIGHTING);
glEnable(GL_COLOR_MATERIAL);
alpha := 1;
end;
end.
```