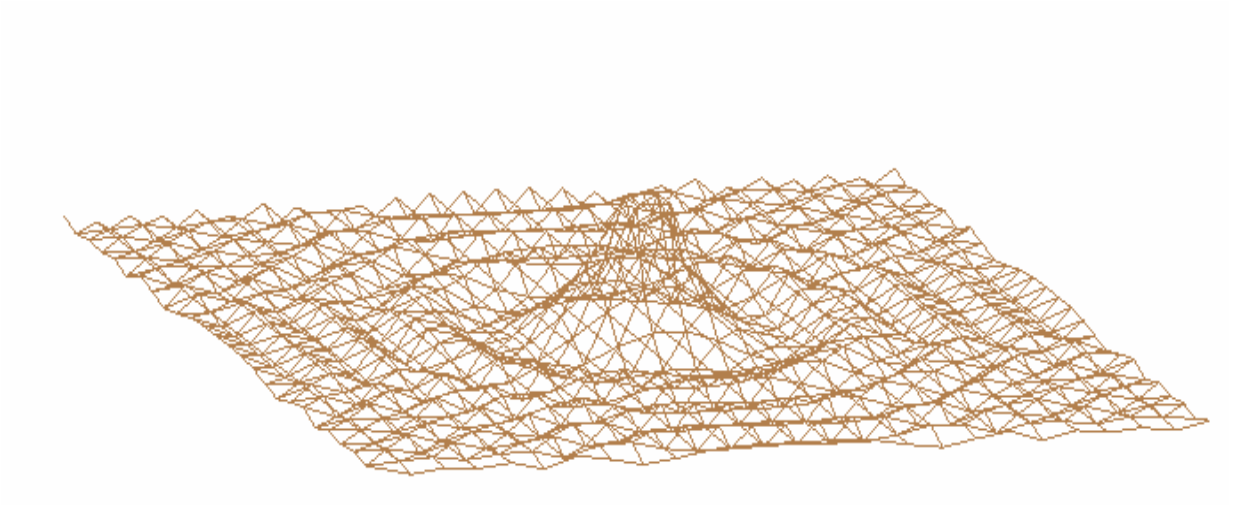
Universidade Federal de Minas Gerais

Programa de Aprimoramento Discente em Modelagem Geométrica Computacional

Curso Básico de OpenGL



Dezembro 2003

Universidade Federal de Minas Gerais
Av. Antônio Carlos, 6627 – Pampulha
31.270-901 – Belo Horizonte – MG – Brasil

Curso Básico de OpenGL

Apostila criada como complemento do "Curso Básico de OpenGL", oferecido pelo PAD Modelagem Geométrica Computacional, apoiado pela PROGRAD e pelo Departamento de Matemática - UFMG.

Componentes PAD Modelagem Geométrica Computacional:

Bernardo Santos bernardo-santos@ufmg.br
Flávio Castro foliver@dcc.ufmg.br
Guilherme Braz guilherme@linuxplace.com.br
Leonardo Catão leocatao@uai.com.br

Pedro Israel psisrael@hotmail.com
Samuel Andrade alemaost@uol.com.br
Wlamir Belchior wlamirm@yahoo.com.br

Orientadores:

Ricardo Takahashi – Departamento de Matemática
Denise Burgarelli – Departamento de Matemática
Rodney Biezuner – Departamento de Matemática
Rodrigo Carceroni – Departamento de Ciência da Computação
Renato Mesquita – Departamento Engenharia Elétrica

Dezembro 2003

Introdução

Sobre a apostila
Sobre a OpenGL

Capítulo 1 – Criando uma janela e um volume de visualização

1.0 - Criando uma janela passo-a-passo
1.1 - Definindo um volume de visualização
1.2– Redimensionamento de Janelas

Capítulo 2 – Modelagem Geométrica 2D e 3D

2.0 - Desenhando primitivas da OpenGL
2.1 - Coloração
2.2 - Transformações geométricas
2.3 - Desenhando primitivas da GLUT
2.4 - Removendo superfícies escondidas
2.5 - Criando objetos transparentes
2.6 - Criando Hierarquias de Transformações

Capítulo 3 – Interação e Animação

3.0 - Interação através do teclado
3.1 - Interação através do mouse
3.2 - Animação

Capítulo 4 – Uso de Iluminação

4.0 - Habilitando o uso de iluminação
4.1 - Definindo propriedades

Capítulo 5 – Tópicos básicos sobre aplicação de textura

5.0 – Compreendendo a aplicação de textura
5.1 – Aspectos teóricos básicos
5.2 – Aspectos práticos básicos

Capítulo 6 – Tópicos básicos sobre fractais

6.0 - Introdução
6.1 - Implementação de figuras auto-semelhantes
6.2 - Adição de adereços ao fractal
6.3 - Usando números aleatórios
6.4 - Considerações finais

Apêndice

I - A linguagem C
II - Como instalar a OpenGL no Dev C/C++
III - Guia para consultas rápidas (funções mais utilizadas)
IV - Seleção de Sites

Introdução

Sobre a apostila

Esta apostila foi desenvolvida pelo grupo de estudos sobre modelagem geométrica computacional, apoiado pela UFMG através do programa PAD (Programa de Aprimoramento Discente), como um complemento para o curso oferecido durante a Semana do Conhecimento.

O objetivo deste trabalho é capacitar pessoas com um pequeno conhecimento prévio em programação a, em poucos dias, desenvolver programas simples de computação gráfica, utilizando a biblioteca OpenGL. Devido ao tempo curto proposto para aprendizagem, não será dada ênfase à teoria envolvida nos processos de modelagem. Em vez disso, o tempo será integralmente concentrado na parte prática dos mesmos.

Ao longo da apostila, serão apresentados programas-exemplos ou passagens destes que usam a teoria explicada, além de propostas de exercícios utilizando-os. Estes programas podem ser encontrados no CD-ROM disponível junto com esta apostila ou pelo site do grupo. Os exercícios propostos devem ser feitos à medida em que forem solicitados pelo palestrante; e qualquer eventual dúvida deve ser enviada para *cursodeopenglufmg@yahoogrupos.com.br*. O e-mail será respondido o mais breve possível. Não se esqueça de antes se inscrever na lista, mandando um e-mail para *cursodeopenglufmg-subscribe@yahoogrupos.com.br.*

Este material divide-se em cinco capítulos subdivididos em tópicos. No primeiro capítulo será dada a base para a criação de uma janela e um volume (ou plano) de visualização adequado. Estes passos são extremamente importantes, pois formam os pilares para a construção de qualquer programa utilizando o OpenGL.

O capítulo dois compreende o assunto de maior ênfase dado na apostila: a modelagem geométrica. Após seu estudo, o leitor deve ser capaz de modelar objetos 2D e 3D, tais como pontos, retas, planos, triângulos, esferas, cubos, paralelepípedos, dentre outros. E ainda: utilizando estes objetos básicos e aproveitando as ferramentas que permitem a realização de transformações geométricas e hierarquia de transformações, o leitor estará apto a representar praticamente qualquer cena desejada, tal como uma casa, um carro ou até mesmo estruturas mais complexas como um robô.

A parte de interação usuário-programa, utilizando a OpenGL e suas bibliotecas auxiliares, é extremamente simplificada, e o capítulo três se encarrega de fornecer informações para tal. Além disso, será explicada uma forma simples de fazer animação sem depender de nenhum tipo de interação.

O capítulo quatro trata de um recurso utilizado para otimizar o realismo de qualquer cena: a iluminação. Este assunto é de constante pesquisa entre programadores do mundo inteiro e, de fato, apresenta-se como um diferencial entre os trabalhos de computação gráfica. Ele é o último capítulo sobre tópicos utilizando a OpenGL e a sua leitura deverá ser suficiente para criar uma iluminação simples em qualquer cena.

Os apêndices fornecem suporte para o aprendizado: uma introdução à linguagem C, um guia de como instalar a OpenGL no Dev C++, uma seleção dos melhores sites de OpenGL encontrados na “web” (muito importante para quem pretende se aprofundar no assunto), um guia para consultas rápidas sobre as funções mais utilizadas da OpenGL, GLUT e GLU.

Sobre a OpenGL

A OpenGL (Open Graphics Library) foi desenvolvida em 1992 pela Silicon Graphics, maior empresa de computação gráfica e animação em todo o mundo. Por ser uma ferramenta portável e rápida, é amplamente utilizada na construção de jogos eletrônicos, tal como Counter Strike, Quake, dentre outros. É importante não confundir: OpenGL não é uma linguagem de programação, e sim uma eficiente API (Aplication Programming Interface). Quando se diz que um programa é escrito em OpenGL significa que são feitas uma ou mais chamadas às suas funções.

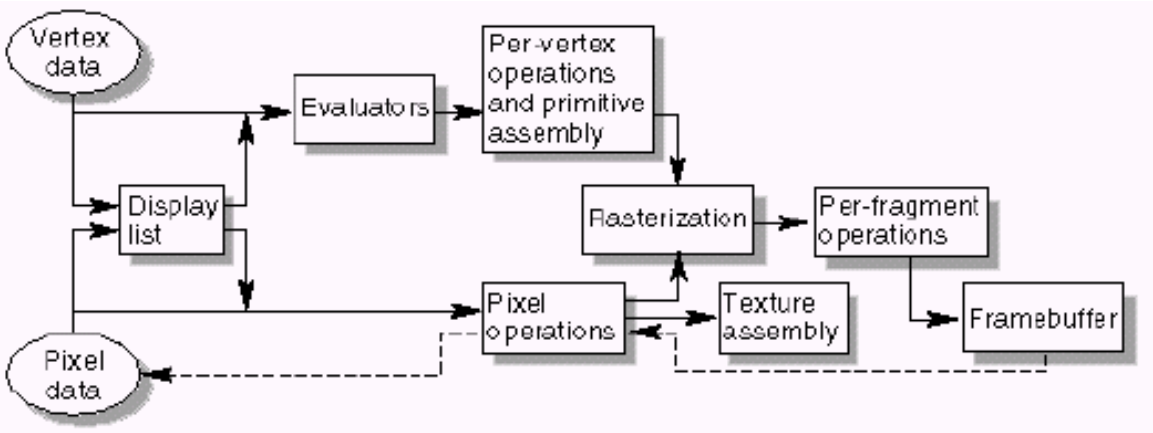
A OpenGL é considerada uma ferramenta relativamente simples, já que, para se alcançar a aparência desejada, é preciso apenas determinar os passos que devem ser feitos para tal finalidade, poupando assim bastante tempo na construção de qualquer programa. Para isso, são oferecidas várias primitivas de baixo nível, sistemas de iluminação, coloração, textura e diversos outros recursos. Já o sistema de gerenciamento de janelas, em nossa apostila, será feito pela biblioteca GLUT (OpenGL Utility Toolkit). Além disso, outra biblioteca será também incorporada aos nossos programas: a GLU (OpenGL Utility library) por nos fornecer várias funções auxiliares, principalmente primitivas gráficas como superfícies quádricas e Splines.

Devido às funcionalidades providas pela OpenGL, esta API tem se tornado um padrão largamente utilizado em diversos setores. Possuir rotinas estáveis, boa documentação disponível e ser de fácil aprendizado só ajudou a concretizar este fato.

A maioria das implementações, em OpenGL, segue uma ordem de operações chamada OpenGL Pipeline. Apesar de não ser uma regra de como a OpenGL é implementada, o Pipeline fornece um guia para entender melhor como e quando a OpenGL realiza suas operações.

Talvez não fique claro, de princípio, o conceito de Pipeline, muito menos sua aplicação, mas à medida em que as lições forem sendo lecionadas, será interessante consultá-lo.

Abaixo, um esquema muito simples e didático do Pipeline da OpenGL:



As funções OpenGL seguem um padrão em relação às suas notações. De forma geral, podemos esquematizar da seguinte maneira:

gl {nome da função}{número de variáveis}{tipo de variáveis}{forma vetorial}(arg 1, arg 2 ...., arg n);

Nem todas as funções seguem exatamente esse formato. Por exemplo, funções que não recebem parâmetros, como "glFlush()", ou que só podem receber um tipo, como "glClearColor()", não possuem nenhum tipo de referência aos argumentos.

Exemplos:
gl Color 3 f => Color é o nome da função, 3 é o número de variáveis passadas para a função e f o tipo de variável
gl Light fv => Light é o nome da função e fv é o tipo de variável (float vector)

A OpenGL funciona como uma máquina de estado, ou seja, uma parte da memória é reservada para guardar o estado atual das diversas variáveis envolvidas no processo de renderização. Então, ao se desenhar algo, todos esses valores armazenados na memória são usados para sua representação na tela, ou seja, primeiro é necessário fornecer todas as características do objeto para só depois desenhá-lo. Isto não é muito intuitivo de início, mas sua praticidade se torna evidente em pouco tempo.

Exemplo: Caso você queira desenhar um quadrado azul rotacionado de 45 graus, primeiro é necessário especificar a cor, depois rotacioná-lo (ou vice-versa, neste caso a ordem dessas alterações não importa) e somente depois desenhá-lo efetivamente.

Capitulo 1 - Criando uma janela e um volume de visualização

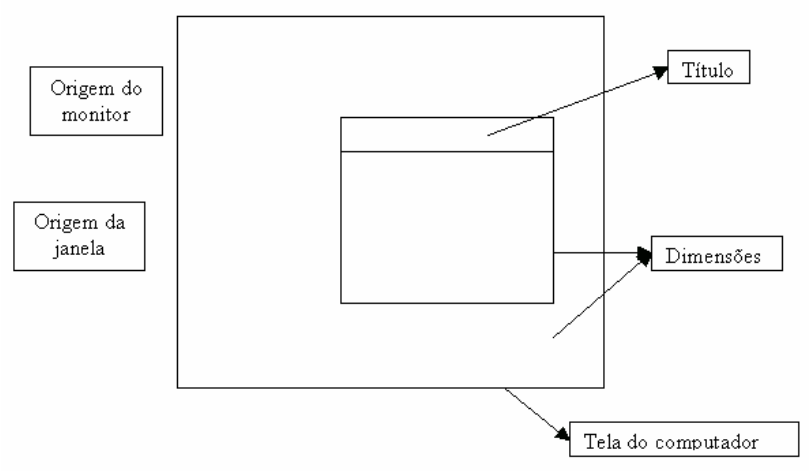
1.0 – Criando uma Janela

Para o processo de criação de janela, utilizamos a biblioteca GLUT, como já foi dito anteriormente. A seguir, será apresentado um trecho, usado para criar a janela, que foi retirado do primeiro programa exemplo:

```
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
glutInitWindowSize(450, 450);
glutInitWindowPosition(50, 50);
glutCreateWindow("Minha Primeira Janela");
```

Obs.: Vale lembrar que o GLUT oferece ferramentas extremamente simples e portáveis para a criação de janelas. Para um trabalho mais detalhado e completo, recomenda-se o uso de outras bibliotecas ou até mesmo o processo de criação de janelas do próprio sistema operacional.

A janela deverá ser apresentada da seguinte forma:



- Comandos apresentados:

glutInit(&argc, argv): inicializa o GLUT. Esta chamada tem que vir antes de qualquer outra da biblioteca GLUT.
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB): determina o sistema de display: o argumento GLUT_SINGLE significa que estamos usando um buffer simples (buffer duplo pode ser usado para uma animação, por exemplo, como veremos mais na frente) e GLUT_RGB siginifica que o sistema de cores será dado pelo sistema RGB (Frações de Vermelho, Verde e Azul, respectivamente).
glutInitWindowSize(450, 450) e **glutInitWindowPosition(50, 50)**: dizemos que nossa janela terá um tamanho de x=450 e y=450 (comprimento e altura, respectivamente, em pixel) e estará na posição x=50 e y=50, onde o (0,0) é o canto superior esquerdo da tela.
glutCreateWindow("Primitivas OpenGl"): cria uma janela efetivamente com o título "Primitivas OpenGL". Perceba que primeiro demos todas as coordenadas da janela para só depois criá-la. Isto porque, como já vimos antes, a OpenGL é uma máquina de estado. Não preocupe com os outros comandos ainda. Por enquanto, saber como criar uma janela é um passo bem grande.

Mas, apenas essas funções não são suficientes para se criar uma janela. Perceba que criamos uma função “INIT()”. Nela existe a função **glClearColor (vermelho, verde, azul, alpha)**: na qual se define a cor de fundo da janela. É importante saber que estes parâmetros devem variar de zero a um e que tal função apenas define a cor de preenchimento e não executa tal.

Em seguida temos a função **glutDisplayFunc(função)**: é responsável por chamar a função “DISPLAY(void)” a cada vez que o estado da cena é alterado. Caso você não se recorde da noção de estado, é aconselhável retornar à seção anterior.

É na função **DISPLAY(void)** que estão contidas as informações necessárias a respeito da cena e do desenho que será feito na janela criada. Como explicamos anteriormente, em OpenGL usamos matrizes para operar sobre nossa cena. Sendo assim, devemos sempre definir em qual matriz estamos trabalhando e, para isso, existe a função **glMatrixMode(nome da matriz)**. Em nosso caso, estamos trabalhando com a GL_PROJECTION, que está relacionada com tudo aquilo que diz respeito ao volume de visualização, e a matriz MODEL_VIEW, que está relacionada com o desenho em si. Muitas vezes, quando selecionamos uma matriz, é necessário que a multipliquemos pela matriz identidade. O efeito disso é como se estivéssemos simplesmente anulando toda a matriz. Isso ficará mais claro em capítulos posteriores. Para usar a matriz identidade, basta chamar a função “**glLoadIdentity()**”.

Note que, quando estamos operando sobre a matriz de projeção, definimos nosso volume de visualização usando a função “**glOrtho(parâmetros)**”. Os valores destes parâmetros

são os da posição de planos em relação à origem absoluta do sistema. Tudo aquilo que for desenhado fora deste volume não será exibido na tela; portanto, é requerida atenção ao se passar valores ao programa. Em seguida, temos a função **glClear (parâmetros)**, que é responsável por "limpar" o fundo da janela. Para tal, é necessário também que o parâmetro assuma o valor de GL_CLEAR_BIT.

A última função a ser vista nesta seção é a **glutMainLoop()**. Ela é responsável por manter todas as funções da GLUT ativas para qualquer mudança de estado ocorrida no sistema. Ou seja, é ela quem gerencia os "loops" do nosso programa.

Não deixe de consultar o apêndice de cores ao final desta apostila. Nele, você encontrará várias combinações de números que geram diferentes cores.

- Exercício:

Altere os parâmetros numéricos das seguintes funções:

glutInitWindowPosistion (posição x, posição y);
glutWindowSize (largura, altura);
glClearColor (vermelho, verde, azul, alpha);

1.1– Definindo um espaço de visualização

Abaixo, segue um trecho retirado do programa exemplo 2:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-10,10,-10,10,-10,10);
//gluPerspective(45,1,0,15);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glClear(GL_COLOR_BUFFER_BIT);
gluLookAt(0,10,0,0,0,0,1,0,0);
```

A função **glMatrixMode()** define qual matriz será alterada daí pra frente. Se o argumento passado for GL_PROJECTION, os próximos comandos irão alterar a matriz de projeção. Se for GL_MODELVIEW, irão alterar a matriz de modelamento e visão. Todos os programas daqui para frente farão alterações nessas duas matrizes.

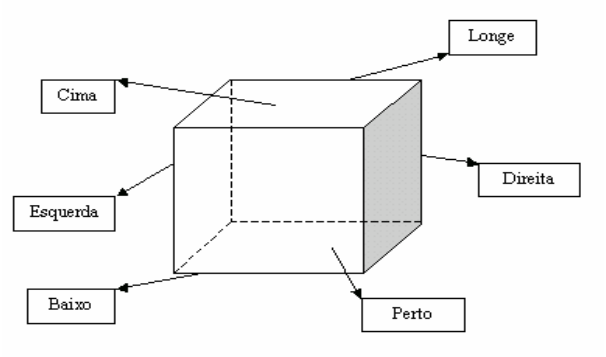
Obs.: SEMPRE desenhe e posicione a câmera na MODELVIEW e SEMPRE defina o volume de visualização na PROJECTION. Assim, vários erros são evitados. Ainda existe uma outra matriz, a GL_TEXTURE, mas não será vista nesse curso.

O **glLoadIdentity()** serve para transformar a atual pilha de matrizes em identidade. Isso também evita alguns erros indesejáveis, se colocado logo após o comando "glMatrixMode()".

O volume de visualização é definido logo após as especificações da janela. Ele definirá tudo que irá aparecer na tela e como irão aparecer. Essas duas características nos dão motivos de sobra para dar atenção redobrada a essa parte.

Há quatro funções para especificar esse volume, duas da glu e duas da opengl:

*__glOrtho(esquerda, direita, baixo, cima, perto, longe)__: define um volume ortogonal, ou seja, um paralelepípedo. Tudo que estiver fora desse sólido não será renderizado. Não há nenhum tipo de perspectiva em profundidade (os objetos não aparentam menores à medida em que se afastam do observador). Este tipo de visualização é bastante utilizada em projetos arquitetônicos e outras aplicações nas quais a medida real do objeto seja mais importante que uma visualização mais realística.



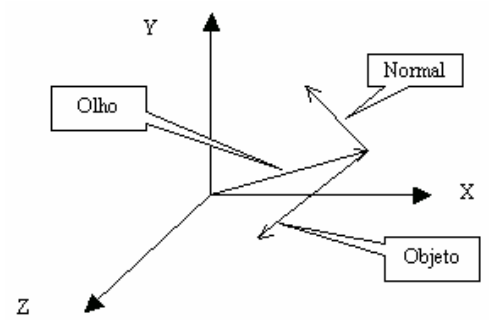
*__glFrustum(esquerda, direita, baixo, cima, perto, longe)__: define um volume na forma de um tronco de pirâmide. Isto permite uma visualização diferente, possibilitando noção de profundidade. Os parâmetros são idênticos ao do glOrtho, exceto que os quatro primeiro se referem ao plano mais perto do observador (note que os planos "perto" e "longe" nesse caso são diferentes).

*__gluPerspective(ângulo,distorção, perto, longe)__: cria um tronco de pirâmide, exatamente como glFrustum. A diferença são os parâmetros: nessa função, o primeiro parâmetro é o ângulo do vértice da pirâmide. Sendo assim, quanto maior o valor, maior o campo de visão (o ser humano tem um ângulo de visão igual a 46 graus. Mais do que isso, somente com a ajuda de lentes chamadas de "grande angular", enquanto valores menores são obtidos com lentes chamadas "teleobjetivas"). O segundo parâmetro define a razão largura/altura da pirâmide. Para não distorcer a sua imagem, coloque esse valor sempre igual a 1. Os outros dois parâmetros são os mesmos das outras funções anteriores. Apesar de parecer mais complicado, esse comando é mais simples que o glFrustum e por esse motivo será utilizado em todos os programas em que for desejado uma projeção em perspectiva.

*__gluOrtho2D (esquerda, direita, baixo, cima)__: por último, esse comando cria um plano ao invés de um volume de visualização. Note que só é necessário passar coordenadas bidimensionais para a função. Se um programa não gerar imagens tridimensionais, essa é a melhor maneira de se especificar onde será renderizada a cena. Por esse motivo, a grande maioria dos programas daqui pra frente usará esse comando.

Outro ponto importante a ser definido em um programa é onde a "câmera" estará e para onde ela estará apontando. Isso pode ser feito através do comando:

gluLookAt(olhox, olhoy, olhoz, <=Posição da câmera
pontox, pontoy, pontoz, <=Objeto
normalx, normaly, normalz) <=Normal à câmera



Para uma demonstração simples, no programa exemplo está o comando glOrtho e o gluPerspective definindo de formas diferentes o volume de visualização para o desenho da frente de um cubo. Descomente as linhas de forma a ver o efeito que cada volume produz. Mude também os parâmetros de gluLookAt como exercício.

Obs.: um cubo, ao ser visto de frente, sem deformação devido à distância, seria visualizado de que forma? E com a deformação? Verifique os resultados através do programa exemplo.

1.2– Redimensionamento de Janelas

Imagine duas situações: revelar uma foto num filme 3 por 4 e num filme 3 por 6. O resultado é ter, na segunda situação, uma foto alongada em um dos eixos por uma escala 6/4. Situação parecida acontece quando redimensionamos a janela em que estamos trabalhando: analogamente, nosso filme foi redimensionado também. Para evitar tais distorções, a função **glutReshapeFunc(Redimensiona)** é utilizada. Ela avisa ao programa que, ao ser realizada qualquer alteração na janela criada pela GLUT, a função "Redimensiona()" será chamada. Esta função tem que, necessariamente, receber dois inteiros: largura e altura da nova configuração da tela, respectivamente. A função abaixo é usada para ajustar o volume de visualização quando a tela é redimensionada.

```
void Redimensiona(int x, int y)
{
  if (y==0)y=1;
  glViewport(0,0,x,y);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  if (x>=y)  gluOrtho2D(1*(float)x/y,4*(float)x/y,1,4);
  else  gluOrtho2D(1,4,1*(float)y/x,4*(float)y/x);
}
```

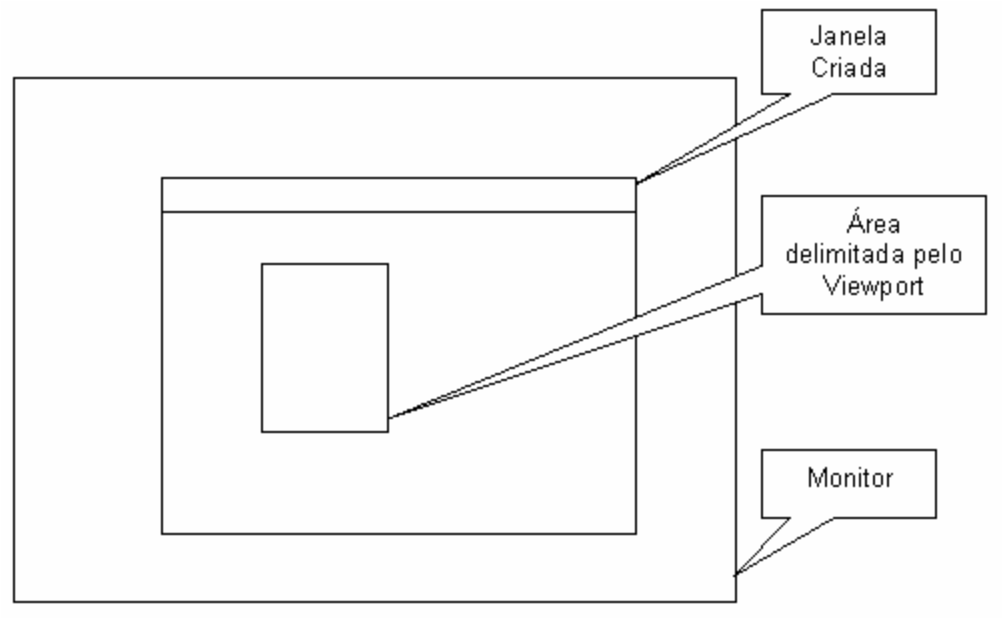
Para evitar a distorção das imagens quando aumentamos, por exemplo, o comprimento da janela, devemos provocar uma distorção igual no nosso volume (ou plano) de visualização. Veja bem as últimas linhas da função abaixo:

se x>=y (comprimento for aumentado), é provocada uma deformação no volume de visualização de x/y no seu comprimento.
se y<x, a deformação é de y/x na altura.

Ex.: Suponha que temos um quadrado de lado unitário dentro de uma janela de tamanho 2 por 2. Então, redimensionamos a tela para um tamanho 4*2. Se não tiver nenhuma função para

controlar alterações na janela, o quadrado se torna um retângulo com o comprimento igual a 4. Mas, se usarmos a função acima, provocamos uma distorção no volume de visualização igual a no quadrado, ficando nosso volume com o comprimento igual ao dobro do anterior. Resultado: nenhuma distorção no desenho é percebida.

A função **glViewport ()** define onde, na tela, será renderizada a imagem. É como se você, após preparar a câmera, escolher o alvo e tirar a foto, fosse escolher qual o tamanho da foto que você quer. Os parâmetros são: as coordenadas do canto inferior esquerdo da tela e as coordenadas do canto superior direito.



Se nenhuma função de redimensionamento for chamada, o "default" da GLUT é ajustar apenas o glViewport.

Obs.: Perceba que não precisamos especificar nosso volume de visualização na função “Desenha()”. Apenas na “Redimensiona()” é suficiente.

Capítulo 2 – Modelagem Geométrica 2D e 3D

2.0 – Desenhando primitivas do OpenGL

A biblioteca OpenGL é capaz apenas de desenhar figuras simples, tais como linhas retas, pontos e polígonos convexos. Essas figuras são denominadas primitivas e, através delas, podemos montar formas mais complicadas, como uma pirâmide. Para desenharmos uma primitiva, necessitaremos basicamente de quatro comandos. Logo abaixo, vemos como eles devem ser dispostos no código:

```
glBegin(parâmetros);
glColor3ub(parâmetros);
glVertex3f(parâmetros);
glEnd();
```

A função **glVertex3f()** é responsável por receber quais os vértices que irão delimitar as primitivas. Sua sintaxe é simples, glVertex3f (coordenada x, coordenada y, coordenada z). A função glColor3ub é o seletor de cores do sistema de desenho. Em seguida, temos os comandos **glBegin()** e **glEnd()**, que sempre trabalham em pares. Por exemplo, temos as seguintes linhas de códigos:

```
glBegin(parâmetro);
glVertex3f(-600.0, -600.0 , 0.0);
glVertex3f(600.0, -600.0 , 0.0);
glVertex3f(600.0, 600.0 , 0.0);
glVertex3f(-600.0, 600.0 , 0.0);
glEnd();
```

Tudo o que houver entres eles será desenhado na janela criada, conforme o parâmetro passado para a função glBegin().
O que queremos dizer é que será esse parâmetro que irá definir quais tipos de primitivas que serão desenhadas.Tal parâmetro pode assumir os seguintes valores:
GL_POINTS, GL_LINES , GL_LINE_LOOP , GL_LINE_STRIP, GL_QUADS, GL_QUAD_LOOP, GL_QUAD_STRIP, GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_POLYGON.
Segue-se uma breve descrição de cada uma das primitivas.

GL_POINTS: Cada vértice é tratado como um ponto. O enésimo vértice define o enésimo ponto. N pontos são desenhados.

GL_LINES: Cada par de vértices gera um segmento de linha independente. Vértices 2n-1 e 2n definem o segmento de linha n. N/2 linhas são desenhadas.

GL_LINE_STRIP: Desenha um grupo de segmentos de linhas conectados do primeiro ao último vértice. Vértices n e n+1 definem linha n. N-1 linhas são desenhadas.

GL_LINE_LOOP: Desenha um grupo de segmentos de linhas conectados do primeiro ao último vértice, voltando então ao primeiro. Vértices n e n+1 definem linha n. A última linha, no entanto, é definida pelos vértices N e 1. N linhas são desenhadas.

GL_TRIANGLES: Cada trio de vértices gera um triângulo independente. Vértices 3n-2, 3n-1 e 3n definem o triângulo n. N/3 triângulos são desenhados.

GL_TRIANGLE_STRIP: Desenha um grupo de triângulos conectados. Um triângulo é desenhado para cada vértice definido depois dos dois primeiros. Vértices n, n+1 e n+2 delimitam o triângulo n. N-2 triângulos são desenhados.

GL_TRIANGLE_FAN: Desenha um grupo de triângulos conectados. Um triângulo é desenhado para cada vértice definido depois dos dois primeiros. Vértices 1, n+1 e n+2 definem triângulo n. N-2 triângulos são desenhados.

GL_QUADS: Cada grupo de quatro vértices é tratado como um quadrilátero independente. Vértices 4n-3, 4n-2, 4n-1 e 4n definem o quadrilátero n. N/4 quadriláteros são desenhados.

GL_QUAD_STRIP: Desenha um grupo de quadriláteros conectados. Um quadrilátero é definido para cada par de vértices definido após o primeiro par. Vértices 2n-1, 2n, 2n+2 e 2n+1 definem o quadrilátero n. N/2-1 quadriláteros são desenhados. Note que a ordem dos pontos utilizada para construir o quadrilátero é diferente da função anterior.

GL_POLYGON: Desenha um simples polígono convexo. Todos os vértices são usados para representá-lo.

Obs.: Para definir o tamanho do ponto a ser desenhado, utilize a função **glPointSize(int tamanho)** e para escolher a espessura da reta, use **glLineWidth(int tamanho)**. Esses dois valores são dados em pixels.

2.1 - Coloração

Quando estamos trabalhando com a OpenGL é desejável atribuirmos diferentes cores para os objetos que estamos desenhando. Para tal, temos a função glColor3ub (vermelho, verde, azul), como já observado anteriormente. Estes parâmetros são valores inteiros que podem variar de 0 a 255 e, de acordo com a combinação de valores, obteremos uma cor diferente. Uma forma de se visualizar a combinação dos números seria a de se usar a palheta de cores disponíveis no Windows e no Linux.

Além de definir a cor, temos também que escolher os tipos de efeitos de preenchimento, utilizando a função **glShadeModel(parâmetro)**. Eles se referem ao modo de preenchimento do desenho, com a cor que você determinou. Temos basicamente dois tipos de preenchimento: o "flat" e o "smooth". O preenchimento "flat" irá colorir o objeto com a cor corrente, enquanto o "smooth" irá fazer um interpolação (degradê) de cores que estão associadas a cada vértice do objeto.

Observe o código abaixo:

```
void INIT(void)
{
 glClearColor(0.0, 0.0, 0.0, 0.0);
 glShadeModel(GL_SMOOTH);
}
void DISPLAY(void)
{
  …
 glBegin(GL_QUADS);
  glColor3ub(255, 0, 0);
  glVertex3f(-100.0, -100.0 , 0.0);
  glColor3ub(255, 255, 0);
  glVertex3f(100.0, -100.0 , 0.0);
  glColor3ub(0, 255, 0);
  glVertex3f(100.0, 100.0 , 0.0);
  glColor3ub(0, 0, 255);
```

```
  glVertex3f(-100.0, 100.0 , 0.0);
glEnd();
…
}
```

Estamos desenhando um quadrado e para cada um dos vértices estamos determinando uma cor diferente. Como definimos, na função “INIT()”, que o sistema de preenchimento era “smooth”, teremos um quadrado degradê na janela. Caso fosse desejado obter um quadrado com apenas uma cor, bastava retirarmos todos “glColor3ub”, excetuando o primeiro.

Logo abaixo, segue uma tabela com algumas combinações numéricas para a obtenção de cores.

Cor	vermelho	verde	azul
vermelho	255	0	0
Verde	0	255	0
Azul	0	0	255
magenta	255	0	255
Ciano	0	255	255
amarelo	255	255	0
Branco	255	255	255
Preto	0	0	0

A maioria dos programas que se encontram em outros materiais, utilizam o comando glColor3f. Definir o tipo de argumento como "f" ao invés de "ub" faz com que a escala de cores varie entre 0 e 1, ao contrário de 0 a 255. Escolhemos esse último por ser mais freqüente em paleta de cores.

2.2- Transformações Geométricas

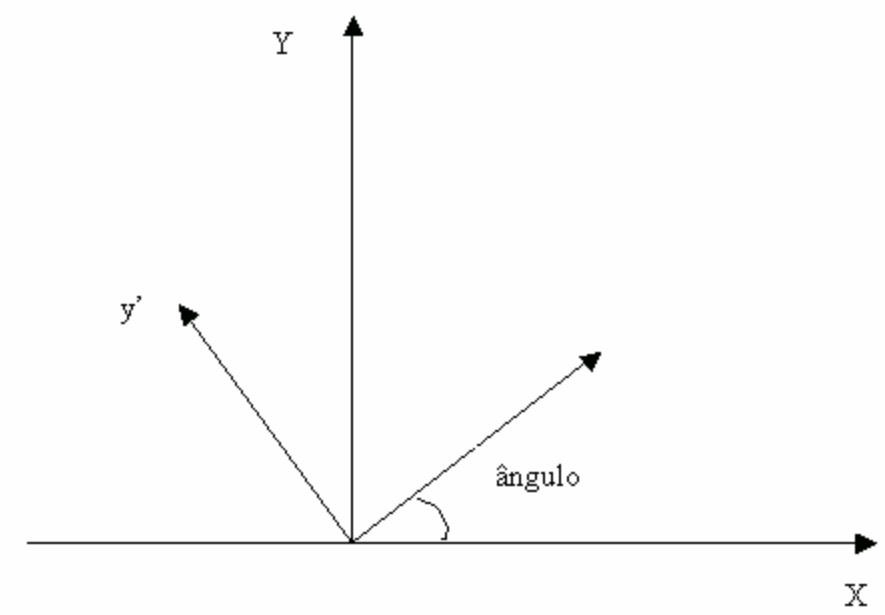
Você deve ter encontrado alguma dificuldade no exercício 2 do capítulo anterior. Isso porque, até o momento, estamos trabalhando sem a utilização de transformações geométricas. Quando estamos gerando uma cena, é necessário dispor os objetos que a compõem de uma forma bem definida. Utilizando as transformações geométricas, podemos posicioná-los e orientá-los de forma precisa.

Basicamente, temos três transfomações geométricas pré-definidas na OpenGL: translação, rotação e escala. Para ficar bem claro a importância disso, analisemos a seguinte situação: no exercício anterior de número dois, foi pedido o desenho de um losango eqüilátero. Mas o que vem a ser um losango eqüilátero? É um quadrado rotacionado de um ângulo de quarenta e cinco graus. No módulo em que são tratadas as transformações, temos exatamente essa situação, porém, usamos uma transformação de rotação para simplificar nosso trabalho. Analisemos então.

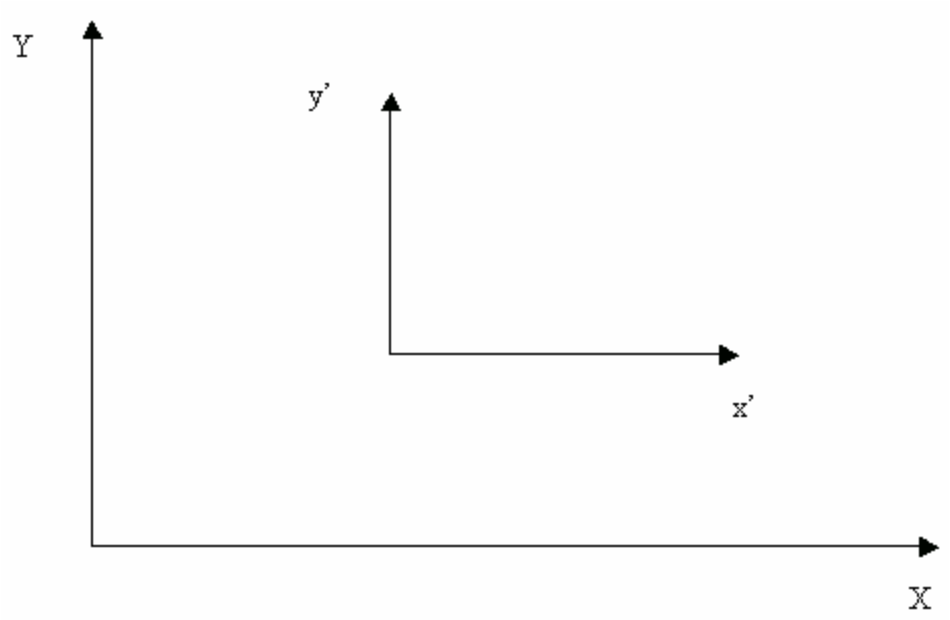
```
glRotatef(45.0, 0.0, 0.0, 1.0);
glColor3ub(255, 0, 0);
glBegin(GL_QUADS);
  glVertex3f(-100.0, -100.0 , 0.0);
  glColor3ub(255, 255, 0);
  glVertex3f(100.0, -100.0 , 0.0);
  glColor3ub(0, 255, 0);
```

```
  glVertex3f(100.0, 100.0 , 0.0);
  glColor3ub(0, 0, 255);
  glVertex3f(-100.0, 100.0 , 0.0);
glEnd();
```

Recordemos o conceito de máquina de estado: é necessário darmos as condições nas quais o desenho será feito. Por isso, aplicamos a rotação de quarenta e cinco graus no sistema, antes de realizar o desenho do quadrado. A função que promove a rotação do sistema coordenado em um ângulo qualquer é "**glRotatef(ângulo, componente x, componente y, componente z)**", onde "ângulo" é o valor, em graus, a ser rotacionado em torno de um vetor definido por componente x, componente y e componente z. A figura a seguir demonstra o efeito de rotacionarmos o sistema de coordenadas em torno do eixo "z".



A função que realiza a translação é "**glTranslatef(posição x, posição y, posição z)**". A origem do sistema será transladada para a posição fornecida como parâmetro para glTranslatef. Observe a figura abaixo:

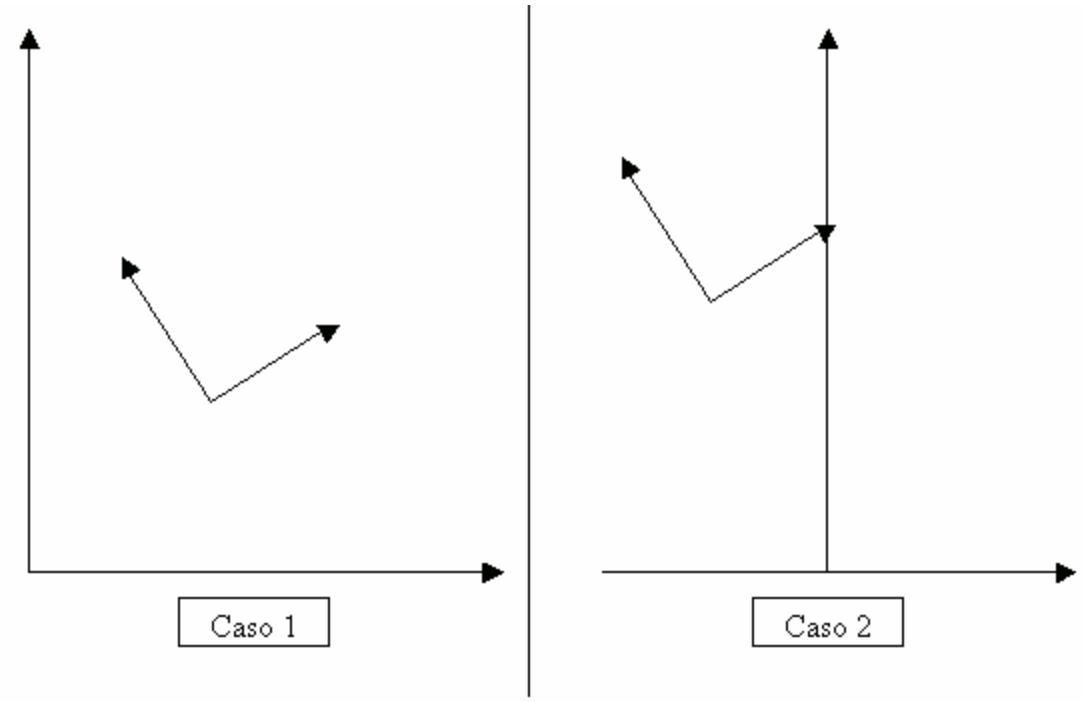


Por fim, temos a função "**glScalef(fator x, fator y, fator z)**", que altera a escala dos eixos conforme o fator multiplicativo fornecido. O resultado é o de encolhimento ou de alongamento das figuras no sentido do(s) eixo(s). Note que, se alguns dos parâmetros for 0, estaremos simplesmente eliminando essa coordenada.

Obs.: Um erro bastante comum é considerar que todas as transformações geométricas são comutativas. Estudando as transformações disponíveis na OpenGL, chegamos à seguinte conclusão:

Rotação / Translação: não comutativa
Rotação / Escala: comutativa
Escala / Translação: não comutativa

Abaixo, um pequeno exemplo mostrando translação/rotação e rotação/translação:



- Exercícios

1- Execute as tarefas descritas no módulo de transformações geométricas.
2- Faça o seguinte desenho: visão superior de um avião qualquer usando as primitivas e as transformações.

2.3 – Desenhando primitivas da GLUT

A biblioteca GLUT é capaz de realizar diversas tarefas; dentre elas, desenhar certos tipos de figuras mais complexas, em três dimensões, e que denominamos primitivas da GLUT. O módulo de primitivas da GLUT contém um código exemplo em que ilustramos algumas dessas primitivas. Vamos analisar algumas primitivas mais úteis. Basicamente, para cada primitiva temos duas variantes: a primeira, em que o resultado é um objeto representado em forma de uma malha de linha ("wire"); e a segunda, em que o objeto é representado em forma sólida ("solid"). Sendo assim, iremos analisá-las aos pares.

Cubo:
glutWireCube (tamanho da aresta);
glutSolidCube (amanho da aresta);
Cone:
glutWireCone (raio, altura, número de planos, fatias horizontais);
glutSolidCone (raio, altura, número de planos, fatias horizontais);
Esfera:
glutWireSphere (raio, número de planos, fatias horizontais);
glutSolidSphere (raio,número de planos, fatias horizontais) ;
Toróide (Rosquinha):
glutWireTorus (raio interno, raio externo, número de planos, fatias horizontais);
glutSolidTorus (raio interno, raio externo, número de planos, fatias horizontais);

O que a GLUT faz para gerar essas figuras é, na verdade, desenhar uma série de planos, aproximando assim para o objeto escolhido. Isso é feito a partir dos parâmetros número de planos e número de fatias; este último divide a figura no determinado número de fatias. Para cada fatia, haverá uma quantidade de planos conforme o parâmetro "número de planos". Sendo assim, se desejarmos, por exemplo, uma esfera "bem redonda", iremos necessitar de passar valores relativamente altos para os parâmetros, número de planos e fatias horizontais. Repare que, quanto maior o valor desses parâmetros, maior será o esforço computacional para gerá-los. Portanto, seja prudente. Repare que podemos usar as funções que desenham cones para criar pirâmides. Para isso, basta passarmos o valor três para o parâmetro "número de planos". Existem diversas funções para se desenhar outras primitivas como dodecaedros, porém, elas não são tão úteis quanto as já apresentadas.

- Exercícios

1 - No módulo correspondente, altere os parâmetros de cada primitiva e observe os resultados. Observe também se houve uma redução na velocidade de rotação dos objetos.
2 - Gere um tetraedro.

2.4 – Removendo Superfícies Escondidas

Apesar do título desta seção, não iremos remover nada, mas sim utilizar uma ferramenta que cria um efeito visual que irá nos permitir diferenciar objetos que estão na frente de outros. Um exemplo seria mais conveniente para demonstrar a importância desse efeito. Imagine dois cubos cujas faces sejam de cores distintas e opacas. Chamemos de cubo 1 aquele em que não há remoção de superfícies escondidas, e de cubo 2 aquele em que esta existe. É muito mais simples e intuitivo imaginar a situação do cubo 2, pois é ela que normalmente ocorre no cotidiano. Não vemos aquilo que está atrás de um outro objeto a menos que este seja transparente. A situação do cubo 1 é bem mais complicada, chegando a ser imprevisível neste caso.

Para evitarmos a situação do cubo 1, a OpenGL nos fornece alguns dispositivos.

```
void INIT(void)
{
  glClearColor(0.0, 0.0, 0.0, 0.0);
  glEnable(GL_DEPTH_TEST);
  glShadeModel(GL_FLAT);
}
void DISPLAY(void)
{
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(-800.0, 800.0, -800.0, 800.0, -800.0, 800.0);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  funçãodesenhacubocolorido();
  glutSwapBuffers();
```

```
}
void main(int argc, char** argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
  glutInitWindowSize(450, 450);
  glutInitWindowPosition(50, 50);
  glutCreateWindow("Aulas em OpenGL");
  INIT();
  glutMainLoop();
}
```

No código acima, foram acrescentados três novos elementos que permitem fazer a remoção de superfícies escondidas. Em "**glutInitDisplayMode(parâmetros)**", acrescentamos um novo buffer "GLUT_DEPTH", que é o buffer de profundidade. Ele é o principal responsável por viabilizar o efeito que desejamos. Além desta modificação, incluímos também a "**glEnable(GL_DEPTH_TEST)**", que habilita o teste de profundidade; e, em "**glClear(parâmetros)**", acrescentamos GL_DEPTH_BUFFER_BIT, que faz com que os pixels do buffer de profundidade assumam a cor definida em "**glClearColor(parâmetros)**".

Devemos observar que é fundamental adicionar esses elementos ao código-fonte quando estamos trabalhando com uma cena em três dimensões. Sem a diferenciação da profundidade dos objetos, a cena irá perder, sem dúvida alguma, a sua realidade.

2.5 - Criando objetos transparentes

Neste capítulo, veremos como podemos adicionar transparência a um objeto. Em seguida, temos um exemplo de como podemos criar tal característica utilizando a função **glBlendFunc().**

```
void Inicio(void)
{
glClearColor(1.0,1.0,1.0,1.0);
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);
glEnable(GL_DEPTH_TEST);
}
```

O primeiro argumento de "glBlendFunc" se refere ao objeto da frente (tomando o observador como referência). O argumento "GL_SRC_ALPHA" significa que o objeto mais próximo será responsável por uma fração igual ao quarto argumento do glColor no cálculo total das cores que serão exibidas na tela. Analogamente, o segundo argumento se refere ao objeto mais longe e o cálculo é feito da mesma forma.

Vamos calcular a cor total usada para criar a cena em um ponto onde podemos ver uma esfera, um octaedro e um prisma, ou seja, as 3 cores se combinam. Começaremos os cálculos de dentro pra fora. A esfera azul no interior é totalmente opaca, não tem transparência. Por isso, 100% da sua cor será levada em conta nos cálculos abaixo. Como o octaedro tem alpha=0.2, apenas 20% da sua cor é utilizada nos cálculos: 0.2 * (0,1,0) = (0,0.2,0). Os outros 80% são ditados pelo esfera no interior, ou seja, 0.8 * (0,0,1) = (0,0,0.8). Logo, a cor total calculada, até agora, vale: (0,0.2,0) + (0,0,0.8) = (0,0.2,0.8). Como o paralelepípedo verde tem alpha=0.2 também, apenas 20% da sua cor é utilizada nos cálculos: 0.2 * (0,1,0) = (0,0.2,0).

Novamente, 80% das cores dependem dos objetos em seu interior. Logo, a cor final calculada vale: (0,0.2,0) + 0.8(0,0.2,0.8) = (0,0.36,0.64).

Este é um exemplo simples de transparência. São apenas 3 elementos compondo uma cena estática. Neste caso, é importante desenhar SEMPRE os objetos opacos primeiro. Tomando este cuidado, garantimos que nossa cena terá o resultado desejado.

Obs.: É importante lembrar de ativar o cálculo de transparência, através de glEnable(GL_BLEND).

- Exercícios

1 - Descomente as 4 linhas do início da função display. Será desenhado um paralelepípedo totalmente opaco logo acima da nossa “bandeira”, usando exatamente esses valores calculados acima como sua cor.
2 - Faça um teste: troque a ordem dos objetos dessa cena e veja o que acontece.

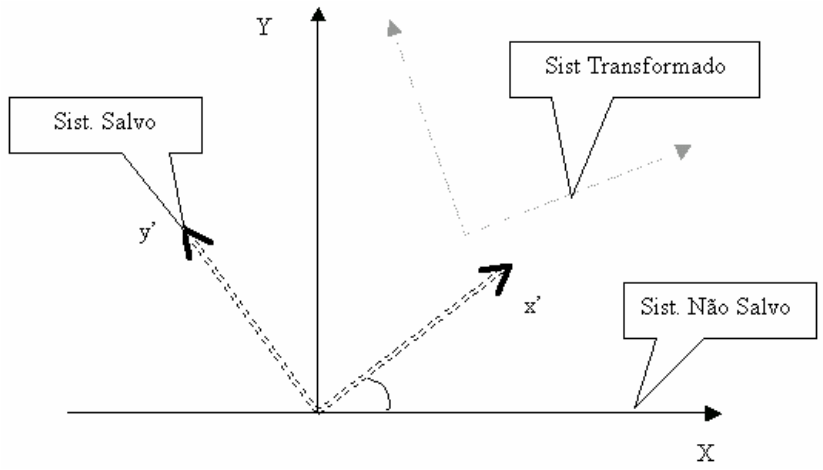
2.6 - Criando Hierarquias de Transformações

À medida em que nossas cenas vão ganhando mais e mais elementos, se torna árduo o processo de determinarmos a posição de todos eles no espaço. Mas isso só ocorre se não utilizarmos uma importante ferramenta da OpenGL.

Para ficar mais claro, partiremos de um exemplo: suponha um prédio qualquer onde existem vários andares, salas, e dentro delas, móveis. Em vez de especificar as coordenadas de uma cadeira qualquer do prédio em relação a uma origem absoluta, como a porta de entrada do prédio, é mais conveniente e sistemático fornecer as coordenadas em relação ao cômodo em que ela se encontra. Este, por sua vez, tem suas coordenadas em relação ao andar no qual se encontra, cuja posição é determinada usando a origem absoluta. É este tipo de organização que utilizamos para facilitar o desenho de um sistema complexo.

Outra aplicação bastante interessante é o modelamento de uma estrutura com alguns eixos articulados, como um braço de robô. Olhando o programa exemplo dessa seção, percebemos que essa hierarquia é feita através da disposição correta das funções **glPushMatrix()** e **glPopMatrix()**. A “grosso modo”, essas funções servem, respectivamente, para salvar e carregar o estado atual da matriz de transformação. Logo abaixo, temos um exemplo genérico de como trabalhamos com os pares de funções glPushMatrix() e glPopMatrix():

```
void Display(void)
{
  glRotatef(ângulo, 0.0, 0.0, 1.0);
  glPushMatrix();
    glTranslatef(deslocamento_x, deslocamento_y, 0.0);
    ...
    “desenhos-1”
   ...
  glPopMatrix();
   ...
    “desenhos-2”
   ...
}
```



Ao analisarmos o código juntamente com a figura, percebemos que primeiro realizamos uma rotação no sistema inicial, que não foi salvo. Isto significa que, caso queiramos voltar a desenhar neste sistema inicial, será necessário aplicarmos uma rotação de mesmo valor, porém, com sentido contrário. Em seguida, salvamos o sistema rotacionado para, logo após, aplicarmos uma translação. A partir daí, realizamos desenhos de quaisquer objetos (os quais chamamos de "desenhos-1"), o que provavelmente irá ocasionar uma mudança drástica no sistema. Feito isso, chamamos a função glPopMatrix(), que retorna automaticamente para o sistema salvo anteriormente, ou seja, o sistema rotacionado. Mais uma vez, realizamos desenhos de quaisquer objetos, os quais chamamos de "desenhos-2", dentro do sistema rotacionado. Como salvamos o estado do sistema, não foi necessário aplicar uma transformação contrária para retorná-lo ao que era antes.

Com este tipo de lógica, é possível então relacionar um sistema relativo a um outro sistema também relativo, gerando então uma espécie de hierarquia de sistemas de coordenadas. No trecho a seguir, temos um exemplo de uma hierarquia de sistemas coordenados. Este exemplo é, na verdade, uma parte de um dos programas didáticos que fazem parte deste curso.

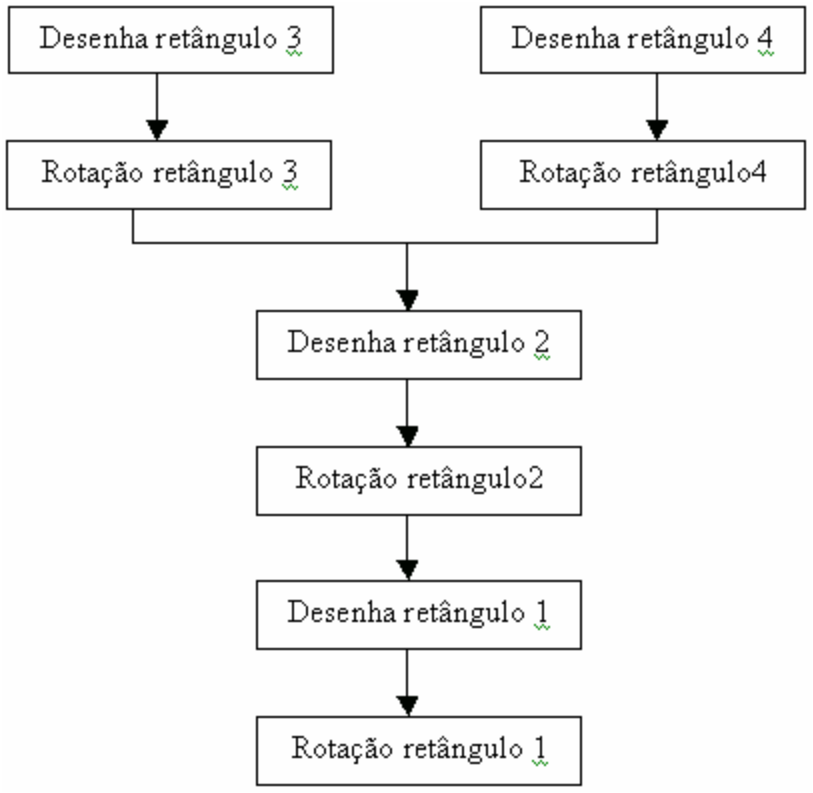
O trecho do programa desenha os 3 últimos retângulos. Com base nas informações dadas acima, o que é de se esperar quando rotacionarmos o Retângulo 2? Acertou se você respondeu que o 3 e o 4 também irão rodar em torno da origem relativa dele. E quando rotacionarmos o 3? Nada acontece com o 2 (ele é desenhado antes dessa rotação) e nada acontece com o 4, porque tem um "glPopMatrix()" separando os dois, ou seja, as modificações na variável rotacao_retangulo_3 só altera o próprio Retângulo 3. O mesmo acontece com o Retângulo 4. Dessa forma, os dois últimos retângulos estão no mesmo nível da hierarquia.

```
glPushMatrix();  //Salvando estado atual de transformações
  glColor3ub(150,150,150);
  glTranslatef(10, 0, 0);
  glRotatef(rotacao_retangulo_2, 0, 0, 1);
  Retangulo(10,4);

  //Desenhando o Retângulo 3
  glPushMatrix();  //Salvando estado atual de transformações
    glColor3ub(100,100,100);
    glTranslatef(10, 0, 0);
    glRotatef(rotacao_retangulo_3+45, 0, 0, 1);
    Retangulo(10,4);
```

```
  glPopMatrix();//Carregando estado de transformação do PushMatrix
correspondente(retângulo3)

  //Desenhando o Retângulo 4
  glPushMatrix();  //Salvando estado atual de transformações
    glColor3ub(100,100,100);
    glTranslatef(10, 0, 0);
    glRotatef(rotacao_retangulo_4-45, 0, 0, 1);
    Retangulo(10,4);
  glPopMatrix();//Carregando estado de transformação do PushMatrix
correspondente(retângulo4)
glPopMatrix();//Carregando estado de transformação do PushMatrix correspondente(retângulo
2)
```

Essas funções, assim como o “glBegin” e “glEnd”, funcionam aos pares. Para evitar erros, sempre tenha um “PopMatrix” para cada “PushMatrix”.

A hierarquia deste programa funciona da seguinte forma: rotacionando o retângulo 1 (alterando o valor do float rotacao_retangulo_1), todos os outros são rotacionados juntos, com o eixo de rotação na origem dele. Rotacionando o retângulo 2, os de hierarquia maior são rotacionados juntos e assim por diante. A cor do retângulo indica sua hierarquia (quanto maior, mais escuro). Segue um pequeno esquema da hierarquia no programa exemplo.



Capítulo 3 - Interação e Animação

3.0 - Interação através do teclado

Esta seção irá tratar da relação usuário/programa através do teclado, o que possibilita uma importante interatividade durante a execução do programa. Ações como girar um objeto ou mudar a sua cor requerem uma interface para serem ativadas, e normalmente, usamos o teclado. Para isso, usamos recursos combinados das bibliotecas GLUT e OpenGL. Observe o seguinte código:

```
GLfloat angulox=0.0;
void DISPLAY()
{
 glMatrixMode(GL_PROJECTION);
 glLoadIdentity();
 glOrtho(-80.0, 80.0, -80.0, 80.0, -80.0, 80.0)
 glRotatef(angulox,1.0, 0.0, 0.0);
 glMatrixMode(GL_MODELVIEW);
 glLoadIdentity();
 ....
}
void KEYBOARD(unsigned char tecla, int x, int y)
{
   switch(tecla)
   {
      case 'x':
      angulox=angulox+5.0;
      if(angulox > 360.0)
      angulox=angulox-360.0;
      glutPostRedisplay();
      break;
      default;
      break;
   }
}
void main(int argc, char** argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
  glutInitWindowSize(450, 450);
  glutInitWindowPosition(50, 50);
  glutCreateWindow("Aula sobre OpenGL");
  INIT();
  glutKeyboardFunc(KEYBOARD);
  glMainLoop();
```

```
}
```

Repare que dentro de "main" temos a função '**glutKeyboardFunc(nome da função com as condições de interação)**". Esta função é responsável por monitorar todo o teclado, excetuando o teclado numérico e certas teclas especiais como as setas. Toda vez em que o usuário pressiona uma tecla, a "glutKeyboardFunc" irá passar para a função " KEYBOARD" três parâmetros, dentre eles o "botão". Com esse parâmetro, podemos ativar determinadas ações sobre nossa cena. No nosso caso, estamos rotacionando nosso volume de visualização através do incremento de uma variável global que está associada a um "glRotatef", operando sobre a matriz de projeção. Enquanto a tecla 'x' estiver pressionada, a variável "angulox" será incrementada e conseqüentemente o volume de visualização será rotacionado. Para que isso possa ser visto na tela do computador imediatamente, é necessário que chamemos a função **glutPostRedisplay()**, ela é responsável por redesenhar a cena toda vez que 'x' for pressionado.

3.1 – Interação através do mouse

Uma outra maneira importante de interação usuário/programa é o uso do mouse. A lógica de trabalho é parecida, mudando apenas algumas particularidades. Vamos usar um código ilustrativo, e, em seguida, faremos algumas observações.

```
 void DISPLAY()
{
 glMatrixMode(GL_PROJECTION);
 glLoadIdentity();
 glOrtho(-80.0, 80.0, -80.0, 80.0, -80.0, 80.0);
 glMatrixMode(GL_MODELVIEW);
 glLoadIdentity();
 ....
 ....
}
void MOUSE(unsigned char tecla, int estado,  int x, int y)
{
  switch (tecla)
{
  case GLUT_LEFT_BUTTON:
     if (estato == GLUT_DOWN)
       função qualquer;
       glutPostRedisplay();
 break;
  case GLUT_MIDDLE_BUTTON:
     if (estado == GLUT_DOWN)
     função qualquer;
     glutPostRedisplay();
  break;
   case GLUT_RIGTH_BUTTON:
     if (estado == GLUT_UP)
     função qualquer;
     glutPostRedisplay();
```

```
  break;

   default:
   break;
  }
}
void main(int argc, char** argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
  glutInitWindowSize(450, 450);
  glutInitWindowPosition(50, 50);
  glutCreateWindow("Aula sobre OpenGL");
  INIT();
  glutMouseFunc(MOUSE);
  glMainLoop();
}
```

Percebemos que, tal como na função "glutKeyboardFunc", a função "**glutMouseFunc(nome da função com as condições de interação)**" passa para a função "MOUSE" os mesmos parâmetros, acrescentando porém, o parâmetro estado. Toda vez que trabalhamos sobre o mouse, a função da GLUT correspondente irá passar os novos parâmetros para a função "MOUSE". Vamos analisar detalhadamente cada parâmetro: "botão" indica qual dos três botões está sendo usado, podendo assumir um dos três seguintes valores: GLUT_LEFT_BUTTON, GLUT_RIGTH_ BUTTON, GLUT_MIDDLE_BUTTON; o parâmetro "estado" indica se o botão está pressionado(GLUT_DOWN) ou não-pressionado(GLUT_UP); por fim, temos os parâmetros "x" e "y", que recebem os valores correspondentes às coordenadas em que se localiza o ponteiro do mouse.

Com isso, temos uma base para podermos criar uma série de operações sobre nossa cena. Poderíamos escrever, por exemplo, uma função que translada um objeto para um local da janela determinado pelo clique do mouse, ou então que um certo objeto mude de cor quando um dos botões do mouse for pressionado, etc.

3.2 – Animação

Um dos objetivos da computação gráfica é, sem dúvida, gerar animações e simulações que seriam muito difíceis, ou até mesmo impossíveis, na vida real. Com as duas ferramentas apresentadas anteriormente, demos um grande passo para podermos criar animações controladas. Vejamos agora como poderíamos criar uma cena com animação um pouco mais automatizada.

A idéia é basicamente a mesma: será necessário escrevermos uma função na qual será incrementada uma variável qualquer, sendo que esta estará associada a uma transformação geométrica, ou frações do RGB, etc. Vejamos um trecho ilustrativo de um código fonte, semelhante ao anterior.

```
void ANIME()
{
  angulox=angulox+5.0;
```

```
   if(angulox > 360.0)
      angulox=angulox-360.0;
}
void DISPLAY()
{
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glRotatef(angulox, 1.0, 0.0, 0.0);
  glOrtho(-80.0, 80.0, -80.0, 80.0, -80.0, 80.0);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
 ....
 ....
}
void MOUSE(unsigned char tecla, int estado,  int x, int y)
{
  switch (tecla)
  {
   case GLUT_LEFT_BUTTON:
     if (estato == GLUT_DOWN)
     glutIdleFunc(ANIME);
   break;
   case GLUT_MIDDLE_BUTTON:
     if (estado == GLUT_DOWN)
     glutIdleFunc (NULL);
   break;

   default:
   break;
  }
```

Neste exemplo, rotacionamos o nosso volume de visualização no sentido horário a cada clique do botão esquerdo e, conseqüentemente, tudo aquilo que está dentro dele. Caso pressionemos a tecla do meio, o quadrado irá interromper o giro. Porém, neste código, há a inclusão de uma nova função da GLUT, a "**glutIdleFunc(nome da função que controla a animação)** ". Esta função tem como objetivo gerar um "loop" infinito na função em que ela tem como parâmetro, caso nenhum outro evento esteja se processando no momento. Isso significa que o volume de visualização irá girar infinitamente, a menos que a tecla do meio seja pressionada.

- Exercício

1 – Estude o módulo de animação, ele é semelhante ao código exemplo desta seção, logo após execute o programa.

Capítulo 4 – Uso de Iluminação

4.0 - Habilitando o uso de iluminação

Um recurso muito importante e amplamente utilizado para aumentar o realismo de uma cena em computação gráfica é a utilização de luz. A OpenGL oferece uma maneira relativamente simples de acrescentá-la em um programa. Este mecanismo será visto abaixo.

Para habilitar o uso de iluminação é necessário acrescentar **glEnable (GL_LIGHTING)**, já que, por default, ela está desativada. Se fizermos apenas isso, nossa cena ficará escura, pois não ativamos nenhuma fonte de luz específica. Para fazê-lo, outro glEnable precisa ser acrescentado, mas com o argumento GL_LIGHTn, onde n é um inteiro identificador da fonte de luz e varia de 0 a 7. Assim, **glEnable(GL_LIGHT0)** ativa a luz de número 0.

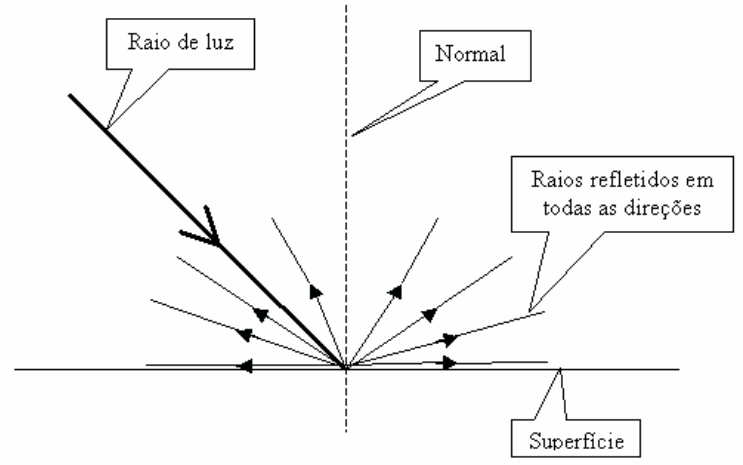
Perceba que não especificamos nenhuma propriedade da luz ou do objeto ainda. Assim, utilizamos os valores "defaults" da OpenGL.

4.1 - Definindo propriedades

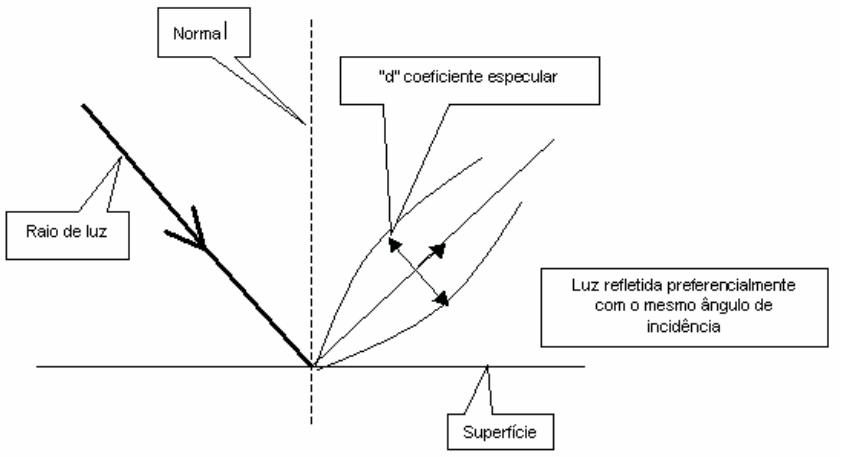
A OpenGL utiliza algumas aproximações para representar a luz, de modo a otimizar a velocidade de processamento requerida, mantendo uma qualidade relativamente alta de seus gráficos. Um desses recursos para modelagem é considerar que existem 3 componentes básicos de luz: a luz ambiente, a difusa e a especular (existe ainda um quarto componente, a emissiva, que não será vista nesse curso).

A propriedade ambiente é uma luz que ilumina toda a cena de maneira uniforme. Esta componente é uma aproximação das sucessivas reflexões ocorridas pela luz em um ambiente. Por exemplo, se ligarmos uma lanterna num quarto totalmente escuro, perceberemos que, mesmo onde não a estamos apontando, ocorre uma pequena iluminação aproximadamente uniforme, resultado das sucessivas colisões dos raios de luz com as paredes.

A propriedade difusa é a mais notável entre as três: ela define a quantidade de luz que, ao encontrar a superfície de um material, é refletida em todas as direções. Por isso, é a maior responsável pela cor final do objeto.



E, por último, a propriedade especular define a quantidade de luz que, ao encontrar a superfície de um material, é refletida preferencialmente seguindo o mesmo ângulo de incidência. Por isso, está intimamente ligada com o brilho dos objetos. Por exemplo, metais têm alta especularidade, ao contrário de um tapete, que é praticamente zero. Espelhos são considerados especulares perfeitos, por refletirem especularmente quase 100 % dos raios que os atingem.



Em seguida, temos uma breve explicação de como determinamos essas propriedades: Primeiramente, definimos os vetores que nos dirão as propriedades da luz e do material. Ambos têm as propriedades ambiente, difusa e especular. Além disso, é necessário especificar a posição da luz e o coeficiente de especularidade do material (define a concentração dos raios refletidos especularmente).

Os vetores que definem as propriedades ambiente, difusa e especular têm 4 valores, sendo que os 3 primeiros são as componentes RGB e o último a componente alpha, que pode servir para tornar o objeto transparente. A intensidade final da luz é calculada tomando como base as propriedades da luz e do material, através da seguinte fórmula:

At = Al*Am
Dt = Dl*Dm
Et = [El*Em]^sh

Em suma, a intensidade de uma determinada componente que atinge o observador é o produto das propriedades desta componente da luz e do material. A única exceção é a parte especular que é elevada ao coeficiente de especularidade (sh).

Se várias fontes de luz estiverem atuando, basta somar as propriedades de todas e efetuar a multiplicação normalmente, como abaixo:

At = (Al1+Al2...+Aln)*Am
Dt = (Dl1+Dl2...+Dln)*Dm
Et = [(El1+El2...+Eln)*Em]^sh

O vetor que define a posição da luz também tem 4 valores: os 3 primeiros são as coordenadas x,y,z da luz e o quarto valor define se a luz é local (usando-o igual a 1) ou "no infinito" (usando-o igual a 0). Este tipo de representação é conhecida como homogênea e é

utilizada diversas vezes em computação gráfica.

```
GLfloat material_ambiente[]= {1.0,1.0,1.0,1.0};
GLfloat material_difusa[]=   {0.7,0.4,0.0,1.0};
GLfloat material_especular[]= {1.0,1.0,1.0,1.0};
GLfloat material_brilho=      30.0;

GLfloat luz_posicao[]=  {0.0,6.0,8.0,1.0};
GLfloat luz_ambiente[]= {0.1,0.1,0.1,1.0};
GLfloat luz_difusa[]=   {1.0,1.0,1.0,1.0};
GLfloat luz_especular[]= {1.0,1.0,1.0,1.0};
```

Após definir todos os vetores, iremos associar cada propriedade com seu respectivo valor.

No caso das propriedades do material, a linha de comando tem a forma:
gl + Material + {tipo de varável} {forma vetorial}, e os argumentos são, respectivamente:
(face a ser renderizada, propriedade, *valores).

```
glMaterialfv(GL_FRONT, GL_AMBIENT, material_ambiente);
glMaterialfv(GL_FRONT, GL_DIFFUSE, material_difusa);
glMaterialfv(GL_FRONT, GL_SPECULAR, material_especular);
glMaterialf(GL_FRONT, GL_SHININESS, material_brilho);
```

No caso da luz, a linha de comando é da forma: gl + Light + {tipo de variável} {forma vetorial}, e os argumentos são, respectivamente: (luz a ser trabalhada, propriedade, *valores).

```
glLightfv(GL_LIGHT0, GL_POSITION, luz_posicao);
glLightfv(GL_LIGHT0, GL_AMBIENT, luz_ambiente);
glLightfv(GL_LIGHT0, GL_DIFFUSE, luz_difusa);
glLightfv(GL_LIGHT0, GL_SPECULAR, luz_especular);
```

Existem diversas outras propriedades que a OpenGL possibilita determinar, como propriedade emissiva de um material, índice de atenuação da luz, ângulo de corte da luz etc, mas estas não serão vistas nesse curso.

Perceba que o cálculo da iluminação, como foi falado no início da seção, depende do ângulo de incidência dos raios de luz sobre a face do material iluminado. A OpenGL calcula esse ângulo automaticamente, desde que você especifique o vetor normal unitário à superfície. Isso é feito através do comando **glNormal3f(normalx, normaly, normalz)**. Veja um exemplo abaixo:

```
glBegin(GL_TRIANGLES);
  glNormal3f(0,1,0);
  glVertex3f(0,0,0);
  glVertex3f(1,0,0);
  glVertex3f(0,0,1);
```

glEnd();

Um triângulo foi desenhado no plano y=0, portanto, sua normal é (0,1,0). Caso a normal passada fosse não unitária, a luz teria um comportamento imprevisível. Para evitar esse tipo de erro, a função “glEnable” tendo como argumento GL_NORMALIZE normaliza automaticamente seu vetor. Não é necessário especificar as normais das primitivas da GLUT e da GLU: elas já estão devidamente calculadas.

É possível passar as propriedades do material de uma forma mais rápida e barata, utilizando a função “glColor3f()”. Para isso, temos que inserir a linha **glEnable(GL_COLOR_MATERIAL)**. Assim, quando escolhemos a cor do objeto, na verdade, estamos definindo as propriedades difusa e ambiente de seu material. Se quisermos alterar outras propriedades através do “glColor”, devemos colocar, antes do “Enable”, a função **glColorMaterial(face do objeto, propriedade(s) da luz)**. Por exemplo, se colocarmos glColorMaterial(GL_FRONT_AND_BACK, GL_SPECULAR), a propriedade a ser alterada através do comando “glColor3f()” é a especular, ao invés das definidas por default.

Capitulo 5 - Tópicos Básicos sobre Aplicação de Textura

5.0 - Compreendendo a aplicação de textura

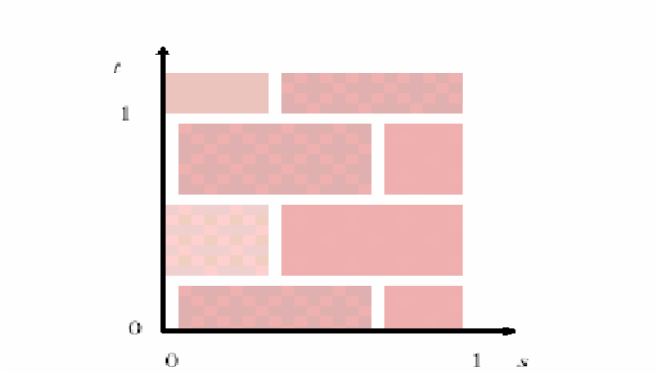
Entende-se como aplicação de textura a adição de uma imagem a um polígono, como, por exemplo, adicionar uma seqüência de imagens de tijolos a um grande retângulo para simular um muro. Pense no esforço necessário para desenhar cada tijolo utilizando as primitivas da OpenGL. Com texturas, essa tarefa é muito simples, além de conferir grande realismo à cena.

Dois passos são importantes nesse processo: a carga da imagem pelo computador, que transforma os dados de cores da foto em dados de cores aplicáveis à cena, e a aplicação, propriamente dita, desses dados ao polígono em questão.

Quanto à carga, pode-se utilizar uma rotina da biblioteca glaux.h para leitura de imagem de extensão .bmp. Como essa biblioteca é um pouco antiga, daremos ênfase à carga de arquivos de extensão .raw, pois esses últimos utilizam as bibliotecas que foram adotadas até então. Carregadores para arquivos de imagem com extensão .tga ou .jpg podem ser facilmente encontrados na internet.

5.1 - Aspectos teóricos básicos

O mapeamento de textura consiste em, inicialmente, aplicar os dados de cor obtidos da imagem em pontos de um plano com coordenadas S (abscissa) e T (ordenada). Esse plano serve para mapear os pontos da imagem, ou seja, sabendo que a imagem possui largura e altura e que os vértices que a delimitam são (0,0), (0,1), (1,1), (1,0) no plano (S,T), todos os pontos internos possuem uma localização e seu respectivo dado de cor. Os dados de cor da textura são chamados de **texels**.



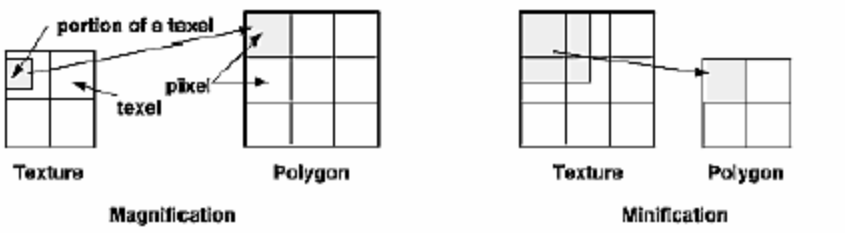
É importante ressaltar que o plano (S,T) não está diretamente relacionado com as faces do polígono ou prisma a ser desenhado, pois essas últimas dependem de três coordenadas (x,y,z). Na verdade, o que o OpenGL faz é uma correspondência entre as coordenadas da textura mapeada e as coordenadas da face, aplicando os dados de cor em pontos no espaço com coordenadas (x,y,z).

Para isso, basta fornecer a correspondência entre os vértices da imagem de textura e do polígono; tudo o que for delimitado por esses vértices será mapeado no plano de textura e terá uma imagem na face delimitada do polígono. Caso a face em questão corte perpendicularmente um dos eixos coordenados, a figura geométrica possui uma das coordenadas da face constante. Logo, para descrever essa face (seu formato, dimensão, etc.) precisamos conhecer as outras duas coordenadas dos vértices e verificar a figura geométrica determinada no plano por esses vértices. O polígono que delimita a textura mapeada deve possuir a mesma forma da figura geométrica da face, para que ocorra perfeita correspondência entre a imagem mapeada e seu local de aplicação. Do contrário, seria como aplicar uma imagem quadrada em um trapézio sem que haja deformação da imagem. Intuitivamente percebe-se que isso é impossível!

Por exemplo, se queremos aplicar os tijolos acima em um retângulo, podemos fornecer a seguinte correspondência:

```
glTexCoord2f(0.0,0.0);glVertex3f(-40.0,-10.0,0.0);
glTexCoord2f(1.0,0.0);glVertex3f(40.0,-10.0,0.0);
glTexCoord2f(1.0,1.0);glVertex3f(40.0,10.0,0.0);
glTexCoord2f(0.0,1.0);glVertex3f(-40.0,10.0,0.0);
```

Independentemente do tamanho do retângulo, ocorre a transferência dos dados de cor da textura, que formam uma imagem quadrada, para pontos do retângulo. Como não aplicamos um quadrado em outro quadrado, evidentemente a imagem ficou dilatada em uma direção. Nesse caso, entram em ação os filtros de imagem. Observe o esquema:



Fonte: The Redbook

Suponha, inicialmente, que um dos dado de cor da textura (texel) utilize exatamente um pixel na imagem desenhada. Sendo assim, imaginando a textura como formada por uma seqüência de pontos de cor (texels) e a imagem desenhada no monitor como um conjunto de pixels, nesse caso a correspondência entre as dimensões da imagem de textura e da imagem desenhada seria perfeita e não ocorreria deformação. Mas, e se queremos diminuir a imagem desenhada, ou mesmo ampliá-la? Veja pelo diagrama anterior que quando aumentamos uma imagem, como se dispuséssemos de uma lupa, uma pequena parte de um texel preencheria totalmente um pixel, para que assim a imagem se magnifique. O contrário é válido para a minificação. O OpenGL realiza cálculos para estimar qual seria o melhor dado de cor a ser aplicado no pixel. Pode ser uma média ponderada dos texels que estão ocupando uma posição de correspondência mais próxima ao centro do pixel em questão (GL_LINEAR) ou simplesmente o texel mais próximo do centro do pixel (GL_NEAREST). Dependendo da situação, podemos escolher entre velocidade de processamento ou qualidade da imagem.

5.2 - Aspectos práticos básicos

Inicialmente, apresenta-se um programa comentado que ilustra a aplicação de uma textura de tijolos a um retângulo. Não é de interesse ressaltar a função dos comandos de carga de textura. O enfoque seria a utilização das funções OpenGL para manipulação das imagens de textura, discutindo unicamente o efeito da alteração dos parâmetros mais significativos.

- Abra o projeto Textura01_raw.dev

Para que a carga de uma imagem .raw genérica funcione, forneça corretamente as dimensões da figura (largura e altura) na função Carregador_de_RAW. No caso, os valores para a figura tijolo.raw já estão ajustados. A dimensão profundidade, normalmente, assume o valor três e é relevante apenas para a carga. Um valor menor do que três deforma a figura.
Faça as seguintes alterações no projeto:

- Experimente alterar as dimensões do retângulo usando o glScalef, abaixo da matriz MODELVIEW. Note que a imagem é modificada, mas mantém seus padrões característicos.
- Experimente um glScalef por um fator maior que 1 na matriz TEXTURE.
- Mantendo sua alteração anterior, procure por glTexParameteri e altere as constantes GL_CLAMP para GL_REPEAT.

Observe que a parede se ajusta. Quando é utilizado um scale, por um fator maior do que 1, na matriz de textura, a área mapeada excede o quadrado unitário. O que o OpenGL coloca fora desse quadrado? Se as constantes GL_CLAMP estão acionadas para ambos os eixos S e T, a última fileira de dados de cor, tanto horizontal quanto vertical da textura, se repete ao longo do plano (S,T). Foi essa informação que mapeamos e enviamos ao polígono.

- Tente, mantendo o scale na matriz TEXTURE, alterar na função glTexParameteri(), um dos eixos para GL_CLAMP e o outro para GL_REPEAT.

A função glTexParameteri() define os filtros de imagem utilizados e a forma como será aplicada a imagem no plano de textura.

- Procure a função glTexEnv() e alterne a constante GL_DECAL para GL_MODULATE.
- Pressione R ou shift+R no teclado. Tente o mesmo com as teclas G e B.

É possível modificar as cores da textura fornecendo um parâmetro float referente a cada coordenada de cor (R,G,B,A) que varia entre 0.0 e 1.0. A constante GL_DECAL define que a textura será aplicada conservando suas cores originais. Utilizando GL_MODULATE, podemos fornecer novos valores para o vetor glColor4f(), o que modifica a cor resultante final.

- Abra o projeto Textura02_raw.dev

A seguir encontram-se os comandos necessários para habilitação de dois ou mais tipos de textura:

- GLuint vetor_de_texturas[2] recebe dois inteiros que, cada um, identificam a imagem a ser utilizada. A constante TOTAL_DE_TEXTURAS identifica o número de texturas possíveis de serem habilitadas.
- Comparando com o programa anterior, a chamada para Carregador_de_RAW foi generalizada, para que várias imagens possam ser carregadas por essa função. Um inteiro adicional identifica a linha "ocupada" pela textura no vetor_de_texturas[].
- A função glGenTextures atribui à textura um valor inteiro que a identifica. Ela recebe como parâmetros o número total de texturas a serem habilitadas e, além disso, guarda no vetor_de_texturas[] o inteiro correspondente àquela imagem.
- glBindTexture efetivamente cria a imagem de textura com os atributos determinados (modo de aplicação, filtros, mipmaps, etc.).

A carga da imagem é efetuada uma única vez, com a função Carregador_de_RAW, o que permite ao programador "descarregar" dados referentes a uma das duas imagens mapeadas por meio da função glBindTexture(). Essa função determina uma textura corrente, ou seja, até que ela seja chamada novamente, toda textura aplicada será aquela que foi "descarregada" por essa função.

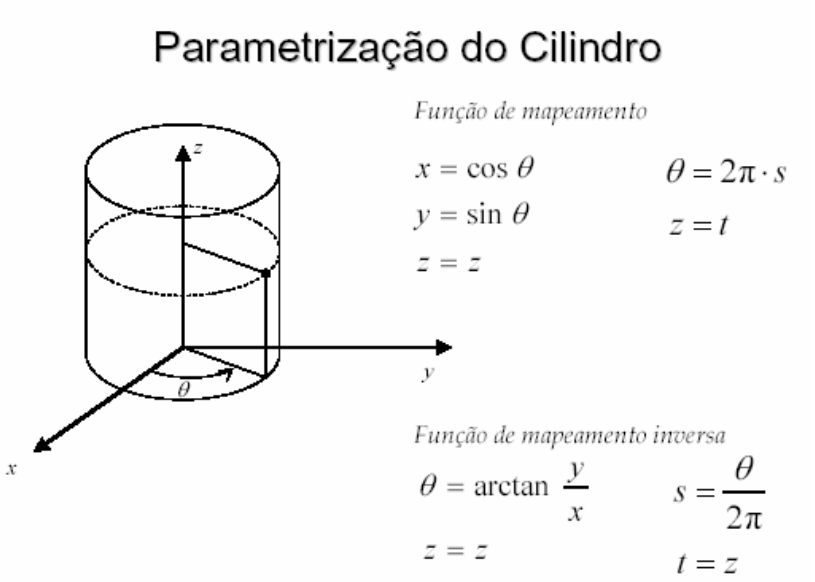
Repare que as transformações referentes ao mapeamento de textura, como scales e translates na matriz GL_TEXTURE foram separadas para cada imagem.

- Altere os parâmetros dos scales e translates e verifique que uma transformação na textura do ratinho não interfere na textura de tijolos.
- Abra o projeto Textura03_raw.dev
- Altere o valor do vetor_de_texturas[] de zero para 1 e mude o filtro de GL_LINEAR para GL_NEAREST na chamada a função Carregador_de_RAW. Observe como a imagem perde em resolução.

Segue-se um exemplo de aplicação de textura em um cilindro gerado pelas primitivas do OpenGL, onde é possível observar a utilização da parametrização de uma superfície em função de duas variáveis. Assim, é possível relacionar os pontos que compõe a superfície com as coordenadas S e T do plano de textura. A idéia na verdade é a mesma: aproximar o cilindro por uma seqüência de faces planas, determinar a correspondência entre os vértices das coordenadas de textura e das faces aproximadas.

- Abra o projeto Textura04_raw.dev

Observe o seguinte esquema:



Essa figura descreve a relação entre as coordenadas de textura (S,T) e os vértices das faces aproximadas do cilindro (x,y,z).

Por fim, destaca-se o uso da função gluBuild2DMipmaps(), que racionaliza o custo computacional ao utilizar um maior número de bytes para pontos da imagem mais próximos do observador, diminuindo gradativamente o número de bytes associados com imagens reduzidas, ou seja, mais distantes do observador. Os filtros de minificação associados ao mipmaps seriam:

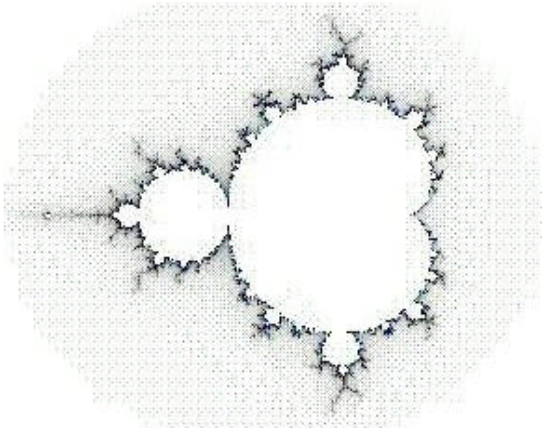
- GL_LINEAR_MIPMAP_LINEAR;
- GL_LINEAR_MIPMAP_NEAREST;
- GL_NEAREST_MIPMAP_NEAREST;
- GL_NEAREST_MIPMAP_LINEAR

Esses filtros determinam o modo de transição entre regiões da imagem que utilizam maiores e menores valores de bytes de cor. A transição entre essas regiões pode ser suavizada com a chamada LINEAR ao final da constante, resultando em uma imagem "smooth".

Capítulo 6 – Tópicos básicos sobre fractais

6.0 - Introdução

Fractais são basicamente formas geométricas determinadas por padrões de figura que se repetem infinitamente. Exemplos:



Nas figuras temos o triângulo de Waclaw Sierpinsky e uma figura do conjunto de Mandelbrot. Vemos claramente no triângulo que cada parte repete exatamente o todo, esse tipo de fractal é chamado de auto-semelhante e é o qual daremos ênfase.

Podemos observar na natureza que existem várias formas que se aproximam da definição de fractais. É o exemplo de árvores, brócolis, arbustos ou até mesmo montanhas podem ser definidas com fractais que utilizam variáveis aleatórias.

A partir de fractais é possível obter diversas figuras bastante complexas (como as figuras acima) mas com uma definição algorítmica/matemática bem simples.

6.1 – Implementação de figuras auto-semelhantes

Como guia para fractais será usado um fractal simples, bidimensional, que cria uma espécie esqueleto de árvore, o código desse fractal poderá ser obtido no módulo1 sobre fractais.

Como dito anteriormente, um fractal é uma estrutura repetida infinitamente. Na prática, na implementação computacional, a repetição infinita da estrutura básica determinaria um estouro de memória. Para então possibilitar a implementação computacional de um fractal são usadas condições de parada.

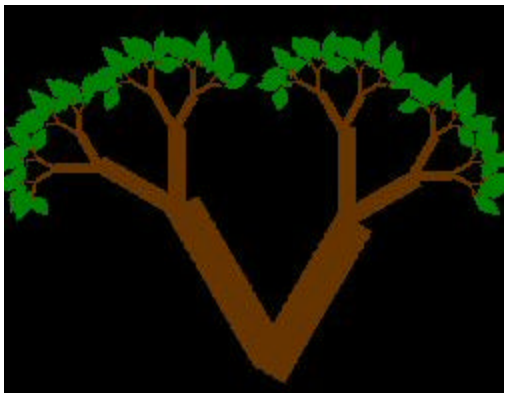
No caso da árvore gerada acima a função que gera o fractal (Arvore) recebe dois parâmetros de entrada. O primeiro parâmetro determina qual o tamanho do menor galho, e o segundo qual o tamanho do galho inicial. O algoritmo implementa uma repetição da estrutura em V diversas vezes e a cada iteração o tamanho do galho utilizado para desenhar é reduzido de um fator qualquer (no código o fator é 1/5) e em seguida comparado com o valor usado como parâmetro do menor galho. Quando o tamanho atual for menor ou igual ao parâmetro menor a repetição para. Essa parada é determinada pelo comando if que compara esses dois valores antes de fazer qualquer outra coisa.

Para estruturar as repetições em um fractal pode-se comumente usar a recursividade. Uma chamada recursiva é caracterizada quando uma função chama ela mesma. A recursão facilita muito a implementação do fractal porque podemos definir apenas a estrutura básica do fractal e fazer em seguida uma nova chama da função com os parâmetros de entrada alterados conforme se deseja.

Para a nossa árvore a recursão é usada da seguinte maneira: desenha-se o braço direito, posiciona-se para desenhar o próximo galho e chama a função árvore novamente com o parâmetro do galho inicial reduzido. Isso fará com que o programa desenhe todos os galhos do lado direito com a redução progressiva dos galhos até que o parâmetro do menor galho seja alcançado. Em seguida posiciona-se para desenhar o braço esquerdo, posiciona-se para desenhar o próximo galho e faz-se novamente a chamada da função com redução do tamanho do galho. Serão desenhados então todos os galhos do lado esquerdo. Os galhos do lado direito que foram desenhados anteriormente prosseguirão no algoritmo e também desenharão seus respectivos braços esquerdo.

Uma dica para facilitar a implementação é pensar na estrutura básica. Desenha-se a estrutura básica, depois posicione o local no código onde deverão ser feitas as chamadas recursivas que é onde a estrutura básica aparecerá novamente.

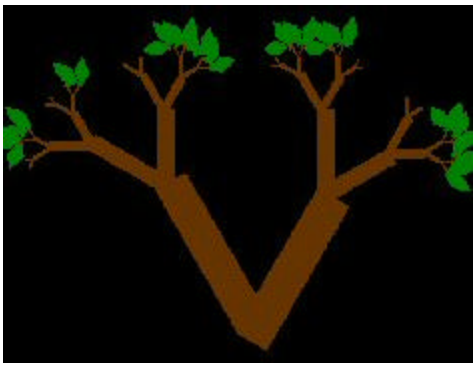
6.2 - Adição de adereços ao fractal



Para transformar nossa antiga árvore seca numa árvore verdinha, cheia de folhas, basta um processo bem simples. A adição de adereços pode ser feita apenas com a adição de uma estrutura condicional que testa o tamanho do galho atual. Nesta árvore o tamanho do galho atual é comparado com o tamanho do galho menor multiplicado por dois. Quando o galho atual já é menor ou igual ao menor é feita algumas transformações para desalinhar as folhas e, em seguida, é chamada a função que faz a folha. As folhas em desalinho conferem um aspecto mais natural à figura. A folha provém da função Folha definida previamente através de um conjunto de pontos dispostos num triagle_fan.

O mesmo processo poderá ser feito para outros fractais comparando os valores que determinam a condição de parada, ou algum outro parâmetro do fractal que é alterado de acordo com o andamento das iterações. Efeitos de mudança de cor, textura, etc, podem ser obtidos desta maneira.

6.3 - Usando números aleatórios



É sabido que na natureza os fenômenos não ocorrem de forma uniforme. Há tantos fatores na natureza que influenciam determinado evento que não é possível descrê-lo de forma analítica, para melhor delimita-lo pode-se fazer através da matemática probabilística.

No exemplo da árvore não são utilizadas nenhuma função de distribuição de probabilidade, nem qualquer outra forma matemática que descreveria com muito melhor fidelidade o que realmente ocorre no evento tratado (onde há galhos e onde não há). Apenas atribuímos uma probabilidade de 90% de uma nova chamada da função, onde seria desenhado outro galho.

Essa probabilidade é implementada através de uma variável que pode assumir valores aleatórios. Em C, os valores aleatórios são obtidos através da função rand(), mas essa função retorna um número de 0 a RAND_MAX, por isso, para obtermos um número aleatório no intervalo [0,1] devemos dividir o resultado por RAND_MAX. È o que é feito na função aleatorio().

Atribuindo então um valor para a variável aleatória, comparamos seu valor com 0,1. Se a variável é maior então se faz a nova chamada da função.

Essa mesma variável poderia ser introduzida para termos uma variação aleatória do tamanho do galho da nova chamada, da inclinação de cada galho, etc... possibilitando obter no final um fractal totalmente aleatório.

É importante ressaltar que o OPEN_GL realiza os cálculos de renderização da cena toda vez que há uma alteração de estado, e a função glutDisplayFunc() é chamada para redesenhar a cena. Movimento da posição do observador, maximização da janela, alteração no tamanho da janela, pressionamento de teclas do mouse ou teclado quando usando as funções de interação, caracterizam mudanças de estado. Quando os cálculos são refeitos os valores da variável aleatória serão diferentes do anterior e uma figura diferente é gerada a cada mudança de estado. Para contornar esse problema é utilizada a semente. A semente é um valor inicial que a variável aleatória assume e a partir dela calcula os demais valores, assim a seqüência de valores aleatórios é sempre a mesma a partir de quando é dada essa semente. Então, para evitar o problema proposto anteriormente, devemos atribuir uma semente antes da chamada da função que desenha o fractal utilizando valores aleatórios. Desta forma a figura desenhada é sempre a mesma não importando a mudança de estado. A figura terá uma relação determinística com a semente, para cada semente uma figura diferente. Experimente retirar a semente e faça mudanças de estado, observe o resultado, é divertido...

6.4 - Considerações finais

Todos os conceitos aqui apresentados poderão ser utilizados para a construção de fractais muito mais complexos, que utilizem o espaço 3D (para a visualização, já que a dimensão de um fractal é assim determinada) ou apresentam diversas colorações; use a criatividade...

A base de todo fractal é a mesma, a repetição, por isso não são necessários grandes códigos (em tamanho) para construir figuras bem complexas, mas pensar na lógica da construção não é muito fácil.

Exemplos de fractais construído pelo PADmod poderão ser observados nos arquivos de exemplo.

Apêndices

I - A linguagem C/C++

Como já dito a biblioteca OpenGL, assim como suas auxiliares, são implementadas utilizando a linguagem C. Sendo assim, é muito conveniente que você escreva o seu programa utilizando C ou C++. O propósito deste apêndice, é o de fornecer apenas a sintaxe de algumas estruturas lógicas básicas destas linguagens.

Declaração de variáveis

Tipo	Declaração	Exemplo
Inteiro	int "nome da variável"	int contador
real ou ponto flutuante	float "nome da variável"	float produto

Caractere	char "nome da variável"	char inicial_do_nome

Declaração de estruturas de armazenamento

Tipo	Declaração	Exemplo
Vetor	tipo "nome da variável"[dimensão]	float vet[3]
Matriz	tipo "nome da variável"[dimensão] [dimensão]	char mat[4][6]

- Estruturas condicionais

Estrutura condicional simples “se”:

```
if(teste)
{
 ....
  ações
 .....
 }
```

Estrutura condicional composta “se – senão - então”:

```
if(teste)
{
 ....
  ações
       .....
 }
else if(teste)
{
 ....
  ações
 .....
 }
 else
 {
 ....
  ações
  }
```

- Estruturas de repetição

Enquanto:

```
while(condição de parada - teste)
{
 ....
  ações
 .....
 }
```

Faça enquanto:

```
do
{
 ....
  ações
 .....
 }while(condição de parada - teste)
```

Repetição composta:

```
for(condição inicial; condição de parada; incremento)
{
 ....
  ações
 .....
}
```

- Funções de Entrada e Saída

Aquisição de valores via teclado:

```
cin>>nome_da_variável;
```

Envio de valores para a tela:

```
cout<<"sua frase"<<nome_da_variável;
```

Para mais informações sobre a linguagem C/C++ consulte nosso site, lá estão disponíveis diversos links.

II - Como instalar o OpenGL no Dev C/C++

A OpenGL é uma biblioteca altamente portável, sendo assim pode ser instalada em diversos sistemas operacionais. Como, sobretudo no Brasil, a maior parte dos usuários de microcomputadores trabalha em ambiente Windows, resolvemos desenvolver este apêndice para auxiliar na instalação da biblioteca em um compilador C/C++ "open source".

Primeiramente faça o download do compilador Dev C/C++ no site da empresa Bloodsheed (http://www.bloodshed.net/devcpp.html), e em seguida instale-o. Realize o download do arquivo "OpenGL.zip" e os das bibliotecas auxiliares em nosso site ou em http://mywebpage.netscape.com/PtrPck/glutming.zip e descompacte-os. A partir daí siga os seguintes passos:

1. Copie o arquivo "glut32.dll" para a pasta "C:\Windows/System" ou "C:\Windows/System 32", dependendo da versão do seu Windows. Normalmente no Windows 98 se copia na pasta "System", enquanto no Windows XP na pasta "System 32".

2. Copie todos os arquivos de extenção ".o" e ".a", para a pasta "C:\Arquivos de programas\Dev-C++\Lib".

3. Copie todos os arquivos de extenção ".h", para a pasta "C:\Arquivos de programas\Dev-C++\Include\Gl".

4. Toda vez que for compilar um código em OpenGL não se esqueça de acrescentar dentro da opção "Project Options" os seguintes itens na caixa "Futher object files or linker options":

   -lopengl32 -lglut32 -lglu32

III - Guia para consultas rápidas (funções mais utilizadas)

Nesta seção apresentam-se algumas das funções OpenGL (incluindo GLUT) utilizadas nos programas de exemplo:

- **void glutInitWindowPosition(int x, int y);**
- **void glutInitWindowSize(int width, int height);**

Define a posição e dimensões da janela a utilizar

```
glutInitWindowPosition(0, 0);
glutInitWindowSize(500, 500);
```

- **void glutInit(int *argcp, char **argv);**

Inicializa a glut.

```
void main(int argc, char **argv)
{
   glutInit(&argc, argv);
```

...

- **#define GLUT_RGB**
  #define GLUT_RGBA
  #define GLUT_INDEX
  #define GLUT_SINGLE
  #define GLUT_DOUBLE
  #define GLUT_DEPTH
  void glutInitDisplayMode(unsigned int mode);

Indica o modo de apresentação a utilizar. RGA, RGBA, INDEX: modo de cor, SINGLE, DOUBLE: utilização de buffer para animação, DEPTH: utilização de Z buffer.

glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);

- **void glutMainLoop(void);**

Entra no ciclo principal de execução da GLUT. A partir deste momento é feita a interação com o usuário e serão chamadas as funções de usuário respectivas.

glutMainLoop();

- **int glutCreateWindow(const char *title);**

Cria uma nova janela com o título indicado

glutCreateWindow("Color Cube");

- **void glutSwapBuffers(void);**

Troca o buffer utilizado (para animação).

glutSwapBuffers();

- **void glutDisplayFunc(void (*)(void));**
  void glutReshapeFunc(void (*)(int width, int height));
  void glutKeyboardFunc(void (*)(unsigned char key, int x, int y));
  void glutMouseFunc(void (*)(int button, int state, int x, int y));
  void glutMotionFunc(void (*)(int x, int y));
  void glutPassiveMotionFunc(void (*)(int x, int y));
  void glutIdleFunc(void (*)(void));

Permitem definir funções que serão chamadas em resposta a certos eventos. Indicar **NULL** para limpar uma definição. Display: é chamada quando a janela é apresentada, Reshape: quando são alteradas as dimensões da janela, Keyboard: quando é premida uma tecla, MouseFunc: quando o usuário move o mouse com um (ou mais) botão precionado, PassiveMouseFunc: quando o usuário move o mouse, Idle: chamada quando o sistema está à espera do usuário.

```
/* definição da função do usuário */
void myReshape(int w, int h)
{
...
}
...
/* indicá-la à GLUT */
glutReshapeFunc(myReshape);
...
/* exemplos de outras definições de função do usuário: */
void myDisplay() ...
void myKeyboard(unsigned char key, int x, int y) ...
void myMouse(int button, int state, int x, int y) ...
void myMotion(int x, int y) ...
void myPassiveMotion(int x, int y) ...
void myIdle() ...
```

- **void glutWireSphere(GLdouble radius, GLint slices, GLint stacks);**
  void glutSolidSphere(GLdouble radius, GLint slices, GLint stacks);
  void glutWireCone(GLdouble base, GLdouble height, GLint slices, GLint stacks);
  void glutSolidCone(GLdouble base, GLdouble height, GLint slices, GLint stacks);
  void glutWireCube(GLdouble size);
  void glutSolidCube(GLdouble size);
  void glutWireTorus(GLdouble innerRadius, GLdouble outerRadius, GLint sides, GLint rings);
  void glutSolidTorus(GLdouble innerRadius, GLdouble outerRadius, GLint sides, GLint rings);
  void glutWireDodecahedron(void);
  void glutSolidDodecahedron(void);
  void glutWireTeapot(GLdouble size);
  void glutSolidTeapot(GLdouble size);
  void glutWireOctahedron(void);
  void glutSolidOctahedron(void);
  void glutWireTetrahedron(void);
  void glutSolidTetrahedron(void);
  void glutWireIcosahedron(void);
  void glutSolidIcosahedron(void);

Permitem desenhar facilmente um conjunto de elementos utilizando faces sólidas (Solid) ou desenhando apenas as arestas (Wire).

```
glutSolidSphere(1.0,  15.0, 15.0);
```

- **#define GLUT_BITMAP_9_BY_15**
  #define GLUT_BITMAP_8_BY_13
  #define GLUT_BITMAP_TIMES_ROMAN_10
  #define GLUT_BITMAP_TIMES_ROMAN_24
  #define GLUT_BITMAP_HELVETICA_10
  #define GLUT_BITMAP_HELVETICA_12

#define GLUT_BITMAP_HELVETICA_18
void glutBitmapCharacter(void *font, int character);

Permite desenhar texto com fonte de mapa de bits. A posição é automaticamente atualizada.

glutBitmapCharacter(GLUT_BITMAP_TIMES_ROMAN_24, 'A');

- **#define GL_COLOR_BUFFER_BIT**
  #define GL_DEPTH_BUFFER_BIT
  #define GL_ACCUM_BUFFER_BIT
  #define GL_STENCIL_BUFFER_BIT
  void glClear (GLbitfield mask);

Limpa buffers.

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

- **void glLightf(GLenum light, GLenum pname, GLfloat param);**
  void glLighti(GLenum light, GLenum pname, GLint param);
  void glLightiv(GLenum light, GLenum pname, const GLint *params);
  void glLightfv(GLenum light, GLenum pname, const GLfloat *params);

Define parâmetros de uma determinada luz.
O parâmetro **light** indica a luz e pode ser: **GL_LIGHT0**, **GL_LIGHT1**, etc...
Para **glLightf(...)** e **glLighti(...)**, **pname** pode ser: **GL_SPOT_EXPONENT**, **GL_SPOT_CUTOFF**, **GL_CONSTANT_ATTENUATION**, **GL_LINEAR_ATTENUATION** ou **GL_QUADRATIC_ATTENUATION**.
Para **glLightfv(...)** e **glLightiv(...)**, **pname** pode ser: **GL_AMBIENT**, **GL_DIFFUSE, GL_SPECULAR, GL_POSITION, GL_SPOT_DIRECTION, GL_SPOT_EXPOENT, GL_SPOT_CUTOFF, GL_CONSTANT_ATTENUATION**, **GL_LINEAR_ATTENUATION** ou **GL_QUADRATIC_ATTENUATION**.

```
static GLfloat light_ambient[]={0.0, 0.0, 0.0, 1.0};
static GLfloat light_diffuse[]={1.0, 1.0, 1.0, 1.0};
static GLfloat light_specular[]={1.0, 1.0, 1.0, 1.0};
static GLfloat light_position[]={2.0, 2.0, 2.0, 0.0};
glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
```

- **void glMaterialf(GLenum face, GLenum pname, GLfloat param);**
  void glMateriali(GLenum face, GLenum pname, GLint param);
  void glMaterialiv(GLenum face, GLenum pname, const GLint *params);
  void glMaterialfv(GLenum face, GLenum pname, const GLfloat *params);

Define características das superfícies a desenhar.
O parâmetro **face** indica qual a face que estamos a definir e pode ser: **GL_FRONT**, **GL_BACK** ou **GL_FRONT _AND_BACK**.
Para **glMaterialf(...)** e **glMateriali(...)**, **pname** só pode ser **GL_SHINNESS**.
Para **glMaterialfv(...)** e **glMaterialiv(...)**, **pname** pode ser: **GL_AMBIENT**, **GL_DIFFUSE, GL_SPECULAR, GL_EMISSION, GL_SHINNESS, GL_AMBIENT_AND_DIFFUSE** ou **GL_COLOR_INDEXES**.

```
static GLfloat mat_specular[]={0.5, 1.0, 1.0, 1.0};
static GLfloat mat_diffuse[]={1.0, 1.0, 1.0, 1.0};
static GLfloat mat_ambient[]={1.0, 1.0, 1.0, 1.0};
static GLfloat mat_shininess={50.0};
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
glMaterialf(GL_FRONT, GL_SHININESS, mat_shininess);
```

- **void glEnable(GLenum cap);**
  void glDisable(GLenum cap);

Permite ativar ou desativar parâmetros globais do OpenGL.
Alguns valores possíveis para **cap**:
GL_COLOR_MATERIAL: as características dos materiais são derivadas da cor atual (ver **glColorMaterial**)
GL_DEPTH_TEST: ativa a utilização do buffer de profundidade (remoção de faces escondidas)
GL_LIGHT*i*: ativa a luz *i*.
GL_LIGHTING: ativa a utilização das luzes para o cálculo das cores finais de cada vértice.
GL_LINE_SMOOTH: desenha linhas suavizadas (anti-aliasing)
GL_NORMALIZE: normaliza os vetores indicados em **glNormal**.
GL_POINT_SMOOTH: equivalente a **GL_LINE_SMOOTH** mas para pontos.
GL_POLYGON_SMOOTH: equivalente a **GL_LINE_SMOOTH** mas para poligonos.
GL_SMOOTH: ativa sombreamento suave (interpolação entre vértices).

```
glEnable(GL_SMOOTH); /*ativar shading suave */
glEnable(GL_LIGHTING); /* ativar luzes */
glEnable(GL_LIGHT0);  /* ativar luz 0 */
glEnable(GL_DEPTH_TEST); /* ativar teste z buffer */
glEnable(GL_COLOR_MATERIAL); /* ativar materiais a partir da cor */
```

- **void glClearColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha);**

Indica a cor de fundo a ser utilizada por **glClear(...).**

```
glClearColor (0.0, 0.0, 0.0, 0.0);
```

- **void glFlush(void);**

Obriga o OpenGL a completar todas as operações pendentes. **glFlush(...)** deve ser chamada quando é necessário garantir que a imagem está completamente definida na tela, por exemplo antes de aguardar por entrada do usuário.

```
glFlush();
```

- **void glPushMatrix(void);**
  void glPopMatrix(void);

Permitem guardar temporariamente a matriz de transformação atual. Após a instrução **glPopMatrix()** todas as operações que alteram a matriz atual (ex: **glTranslatef(...)**, **glLoadIdentity()**) efetuadas após o último **glPushMatrix()**, são ignoradas.

```
glPushMatrix();
glScalef(1.0,5.0,3.0);
gluWireCube(1.0);
glPopMatrix();
```

- **void glOrtho(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far);**

Define a área de visualização para uma projeção ortogonal. De notar que os parâmetros **near** e **far** são indicados relativamente ao usuário, ou seja **near** deve ser menor que **far**. Os parâmetros **left**, **right**, **top** e **bottom** devem ter em conta as dimensões atuais do viewport utilizado para que as imagens não apareçam distorcidas. Todos os objetos, ou seções dos mesmos, fora do volume de visualização indicado não são apresentadas na janela.

```
glViewport(0, 0, w, h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
if (w <= h)
   glOrtho(-2.0, 2.0, -2.0 * (GLfloat) h / (GLfloat) w, 2.0 * (GLfloat) h / (GLfloat) w, -10.0, 10.0);
else
   glOrtho(-2.0 * (GLfloat) w / (GLfloat) h, 2.0 * (GLfloat) w / (GLfloat) h, -2.0, 2.0, -10.0, 10.0);
glMatrixMode(GL_MODELVIEW);
```

- **void glFrustum(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble znear, GLdouble zfar);**

Permite definir uma perspectiva. Os parâmetros **left**, **right**, **top**, **bottom** e **znear** definem a área de visualização mais próxima do observador, o parâmetros **zfar** define o plano de corte mais distante. De notar que tanto **znear** como **zfar** tem de ser maiores que zero. Da mesma forma que **glOrtho**, estes parâmetros são indicados relativamente ao observador.

```
glViewport(0, 0, w, h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
if (w <= h)
   glFrustum(-2.0, 2.0, -2.0 * (GLfloat) h / (GLfloat) w, 2.0 * (GLfloat) h / (GLfloat) w, 10.0, 30.0);
else
```

```
   glFrustum(-2.0 * (GLfloat) w / (GLfloat) h, 2.0 * (GLfloat) w / (GLfloat) h, -2.0, 2.0, 10.0, 30.0);
glTranslatef(0.0, 0.0, -20.0);
glMatrixMode(GL_MODELVIEW);
```

- **void gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble zNear, GLdouble zFar);**

Permite definir uma perspectiva equivalente a **glFrustum**. O parâmetro **fovy** indica o ângulo (em graus) de abertura vertical, **aspect** é a relação entre a largura e a altura do viewport. **znear** e **zfar** definem os planos de corte de profundidade. Tal como em **glFrustum** devem ser ambos maiores que zero.

```
glViewport(0, 0, w, h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
if (w <= h)
   gluPerspective(90.0 * (GLfloat) h / (GLfloat) w, (GLfloat) w / (GLfloat) h, 10.0, 30.0);
else
   gluPerspective(90.0, (GLfloat) w / (GLfloat) h, 10.0, 30.0);
glTranslatef(0.0, 0.0, -20.0);
glMatrixMode(GL_MODELVIEW);
```

- **void gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble centerx, GLdouble centery, GLdouble centerz, GLdouble upx, GLdouble upy, GLdouble upz);**

Permite alterar o ponto de vista de uma perspectiva. **eye...**. são as coordenadas do observador, **center...** as coordenadas do centro da cena e **up...** indica o topo da cena (permitindo rodar a vista). Deve-se notar que após **gluLookAt** os parâmetros **znear** e **zfar** utilizados em **glFrustum** ou **gluPerspective** são relativos ao observador indicado em **gluLookAt** o que pode ter alguns resultados nesperados.

```
glViewport(0, 0, w, h);
glMatrixMode(GL_PROJECTION); >
glLoadIdentity();
if (w <= h)
   gluPerspective(90.0 * (GLfloat) h / (GLfloat) w, (GLfloat) w / (GLfloat) h, 10.0, 30.0);
else
   gluPerspective(90.0, (GLfloat) w / (GLfloat) h, 10.0, 30.0);
gluLookAt(0.0, 0.0, -20.0, 0.0, 6.0, -20.0, 0.0, 0.0, -1.0);
glTranslatef(0.0, 0.0, -20.0);
glMatrixMode(GL_MODELVIEW);
```

- ***void glTranslatef(GLfloat x, GLfloat y, GLfloat z);***
  void glScalef(GLfloat x, GLfloat y, GLfloat z);
  void glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z);

Multiplicam a matriz de transformação atual pelas matrizes de translação (glTranslatef), escala (glScalef) ou rotação (glRotatef). No caso de glRotatef o ângulo é indicado em graus. Na prática

os objetos desenhados após estas instruções são deslocados, esticados ou rodados de acordo com a operação realizada. Deve-se notar que é importante a ordem pela qual executamos as instruções, no exemplo a seguir o ponto atual será (3.0, 2.0, 0.0), no entanto se efetuássemos as operações por ordem inversa seria (1.5, 2.0, 0.0).

```
glScalef(2.0,1.0,1.0);
glTranslatef(1.5,2.0,0.0);
```

IV – Seleção de Sites

Esta seção oferece uma seleção dos considerados melhores sites para se aprender OpenGL. Os assuntos abordados vão desde um curso de C (necessário para o entendimento dos códigos) até a criação de jogos em primeira pessoa. É importante ressaltar a importância dessa parte da apostila para a continuidade dos estudos após o curso.

01) Site oficial do OpenGL. Lá você encontra diversos materiais para download, além das últimas notícias sobre o programa.

http://www.opengl.org
Idioma: Inglês
Classificação: *****

02) Site do curso de C oferecido pela EE UFMG. Muito bom para se aprender a linguagem utilizada em toda a apostila.

http://www.ead.eee.ufmg.br/cursos/C/
Idioma: Português
Classificação: *****

Site com Assunto Separado por Tópicos

03) Ótimo tutorial para iniciantes de uma professora da PUC RS.
O endereço abaixo leva para um menu com 17 opções de tutoriais, desde Introdução até Luzes e Malhas de Polígonos.

http://www.inf.pucrs.br/~manssour/OpenGL/index.html
Biblioteca: GLUT
Idioma: Português
Classificação: ****

04) Neste tutorial há bastante exemplos (todos comentados e explicados) e alguns exercícios para treinamento.

http://dca.ufrn.br/~ambj/ele435/opengl/index.html
Biblioteca: GLUT
Idioma: Português

Classificação: ***

05) Site de outro professor da PUC RS. Não possui um conteúdo muito abrangente, mas os assuntos que são tratados são abordados de forma muito clara e objetiva. Além disso, oferece uma grande ajuda na hora de instalar as bibliotecas utilizadas.

http://www.inf.pucrs.br/~pinho/CG/Aulas/OpenGL/Transparencias/Transparencias.html
Biblioteca: GLUT
Idioma: Português
Classificação: ****

06) Mais um ótimo site para iniciantes. Trata o OpenGL de forma clara, dando tópicos sobre os cálculos envolvidos nas chamadas das funções. A parte de projeção e luz é muito bem explicada.

http://www.comp.ufla.br/~bruno/aulas/cg/monte-mor/44.htm
Biblioteca: GLUT
Idioma: Português
Classificação: *****

07) Este site dá uma breve explicação sobre alguns assuntos, como projeções e transformações geométricas.

http://www.cesec.ufpr.br/~mcunha/opengl/opengl00.html
Biblioteca: GLAUX
Idioma: Português
Classificação: *

08) Este endereço nos fornece vários tutoriais para iniciantes, além do link para baixar um excelente livro em formato .pdf que ensina, passo-a-passo a modelagem de um ser humano caminhando.

http://www.dev-gallery.com/programming/
Biblioteca: GLUT
Idioma: Inglês
Classificação: *****

09) Site com 14 seções sobre OpenGL. Sua ênfase é luz, volume de visualização e outros recursos gráficos. Infelizmente, utiliza o tk ao invés do GLUT, mas nada que atrapalhe a didática significantemente.

http://www.eecs.tulane.edu/www/Terry/OpenGL/Introduction.html
Biblioteca: tk
Idioma: Inglês
Classificação: **

Sites Voltados para a Construção de Jogos Virtuais

10) Este site possui diversos materiais que são necessários para se aprender OpenGL: tutoriais, livros, exemplos etc. O enfoque principal é a criação de jogos.

http://nehe.gamedev.net/
Biblioteca: GLAUX
Idioma: Inglês
Classificação: *****

11) O próximo tutorial é baseado principalmente no material disponível no site "NeHe Productions". Os textos foram traduzidos e adaptados para utilizarem a GLUT. Infelizmente, nem tudo ainda está traduzido e alguns links falham.

http://www.inf.ufsc.br/~awangenh/CG/apostilas/openGL/opengl3D.html
Biblioteca: GLUT/GLAUX
Idioma: Português/Inglês
Classificação: ***

12) Página muito boa contendo diversos materiais completos. É´ necessário criar um usuário e uma senha para ter acesso a todo o site. A ênfase, novamente, é na implementação de jogos, mas é fornecido diversas informações de computação gráfica no geral, inclusive OpenGL.

http://www.unidev.com.br/
Biblioteca: GLUT
Idioma: Português
Classificação: *****

13) Esta página também dá ênfase aos jogos. São 50 programas exemplo começando do mais básico (desenhar um triângulo branco na tela) e terminando com uma aplicação no jogo "Quake" (aliás, feito totalmente em OpenGL). Infelizmente, não utiliza a biblioteca GLUT, o que torna o sistema de gerenciamento de janelas bem mais complicado.

http://www.gametutorials.com/Tutorials/opengl/OpenGL_Pg1.htm
Biblioteca: sem biblioteca auxiliar para gerenciamento de janelas.
Idioma: Inglês
Classificação: *****

Downloads

14) Este endereço nos leva a uma página que permite o download de um .zip. Este arquivo contém 7 executáveis que ajudam na vizualização de algumas das propriedades seguintes: neblina, posição e propriedades da luz, propriedade do material, projeções, transformações, texturas e formas (primitivas OpenGL).

http://www.xmission.com/~nate/tutors.html
Idioma: Inglês
Classificação: *****

15) Um dos sites mais importantes, já que disponibiliza o RedBook tanto para download

quanto para consulta online. Pena que ainda utiliza o GLAUX (adicionando o biblioteca GLUT, retirando a GLAUX e alterando 3 linhas do main, concertamos este "problema").

http://fly.cc.fer.hr/~unreal/theredbook/
Biblioteca: GLAUX
Idioma: Inglês
Classificação: ****

Sites Teóricos

16) Este site, essencialmente teórico, explica através de equações e exemplos vários dos recursos gráficos possíveis no OpenGL. Pena que não tem nenhum exemplo ou referência a ele. Ainda assim, este tutorial é um recurso importante para improvisar o realismo usando qualquer programa de computação gráfica.

http://www.directnet.com.br/users/val/tutor/tutor.html
Idioma: Português
Classificação: ***

Consultas Rápidas

17) Oferece um guia para consultas rápidas das funções mais utilizadas no OpenGL, GLUT e GLU.

http://www.dei.isep.ipp.pt/cg/opengl/
Biblioteca: GLUT
Idioma: Português
Classificação: ***

18) Outro guia para consultas rápidas. Este, ao contrário do anterior, é extremamente completo, contendo mais de 100 funções do OpenGL (o que representa quase todas elas) bem explicadas além de vários comandos Glu. Não oferece material para GLUT.

http://www.deec.uc.pt/~peixoto/dcg/opengl/opengl.html
Biblioteca: sem GLUT
Idioma: Inglês
Classificação: ****