UNIVERSIDADE ESTADUAL DO PIAUÍ - UESPI
CAMPUS PARNAÍBA

MINICURSO SOBRE:

OPENGL BÁSICO

Josué Machado Mota

PARNAÍBA – PI
29-04-08

SUMÁRIO

SOBRE O OPENGL

Desenvolvida em 1992 pela Silicon Graphics, o OpenGL (Open Graphics Library) é uma interface de software multiplataforma destinada a manipulação de hardware gráfico, escrita em linguagem C. Hoje, OpenGL devido ser bastante rápido e portável, tem conquistado diversas áreas, tais como: aplicações científicas, software de modelagem e jogos (2D/3D).

OpenGL não é uma linguagem de programação, no entanto, é uma eficiente API (Application Programming Interface) gráfica com mais de 150 funções distintas disponíveis para manipulação de imagens. Atualmente, OpenGL integra-se a diversos ambientes: Windows, Unix Solaris, Linux e MacOS. Possui implementações para diversas linguagens de programação: C/C++, Java, Object Pascal (Delphi), etc. Isto demonstra claramente porque OpenGL tornou-se um padrão na área de Computação gráfica.

Ao desenvolver uma aplicação fazendo uso de suas funções, dizemos que a aplicação foi desenvolvida em linguagem x baseada em OpenGL, já que esta não é tida como uma linguagem de programação.

Qual linguagem de programação devo usar? Em muitos dos casos C/C++, por também ser multiplataforma, possuir alto desempenho e permitir otimizações de baixo nível, e também por ser a linguagem mais utilizada no desenvolvimento de jogos atualmente. Os exemplos contidos neste documento foram escritos todos em linguagem C, utilizando o Dev-C++ em ambiente Windows.

Sendo apenas uma API gráfica multiplataforma, não possui manipulação de janelas nem iteração com o usuário, já que o tratamento destas difere de sistema para sistema. Em se tratando de aplicações de pequeno ou médio porte, a utilização da biblioteca GLUT (OpenGL Utility ToolKit) é uma ótima opção pois, também é multiplataforma e possui diversas funções para manipulação de janelas e eventos para a iteração do usuário com sua aplicação. Caso sua aplicação seja voltada para muitas animações, o melhor caminho é estudar como funcionam as funções de criação de janela e tratamento de eventos do próprio sistema operacional, o que muita das vezes não é algo agradável.

Outra biblioteca é a GLU, tida como um padrão utilizado na maioria das aplicações baseadas em OpenGL devido possuir funções para criação de superfícies curvas, quádricas e manipulação do posicionamento da cena e do observador.

UMA MÁQUINA DE ESTADOS

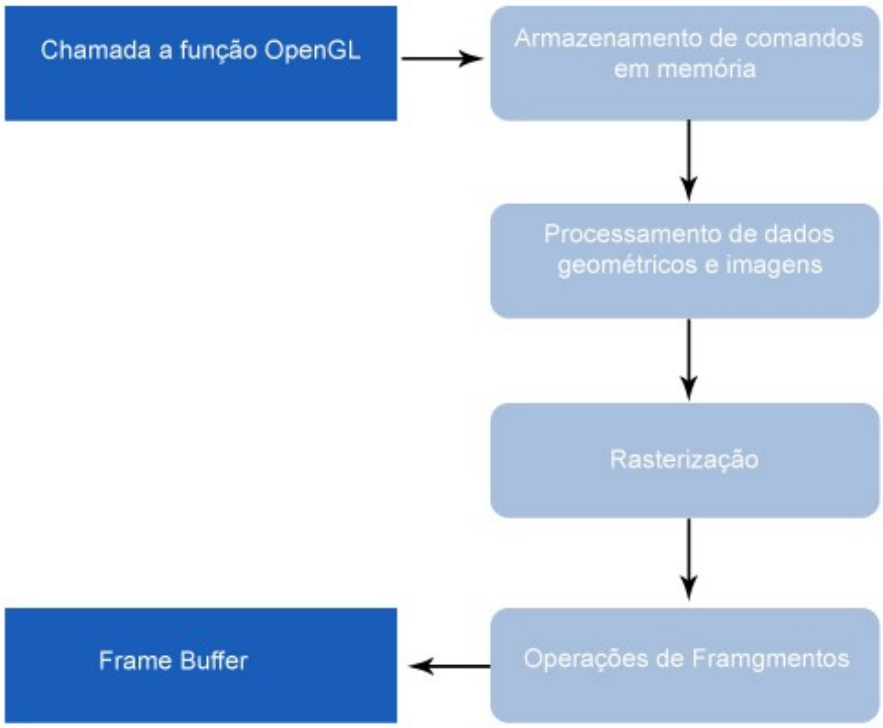
OpenGL dispõe de diversas variáveis para manipulação de estados, tais como: coloração, posicionamento, transformações, iluminação, transparência, etc. Isto faz com que o programador não precise desenhar sempre toda a cena e sim, fazer as mudanças necessárias para que obtenha o resultado desejado. Ao iniciar a aplicação, todos os estados possuem seus valores default e ao decorrer de sua execução não se alteram ao menos que se faça uma chamada a uma função capaz de realizar a troca para aquele determinado estado.

Um exemplo simples seria: ao iniciar uma aplicação baseado em OpenGL, por default sua cor de fundo é preto e de desenho é branco, logo, todos os objetos a serem desenhados na tela teriam a cor branca por preenchimento, ao menos que a cor de desenho seja alterada para uma outra cor.

Muitas destas variáveis de estados podem ser habilitadas e desabilitadas, como exemplo temos a iluminação da cena, podemos habilitá-la com o comando `glEnable (GL_LIGHTING)` e desabilita-la com `glDisable (GL_LIGHTING)`.

PIPELINE DO OPENGL

Em todo e qualquer comando de desenho, OpenGL segue sempre uma ordem de operações para que a imagem seja exibida ao usuário final. Esse processo recebe o nome de PipeLine de Renderização do OpenGL, "o núcleo do OpenGL". Simplificando, o PipeLine seria:



Chamada a função OpenGL – é a chamada feita à função dentro de seu programa;
Armazenamento dos comandos em memória – suas instruções são armazenadas em memória como qualquer outra instrução;
Processamento de dados geométricos e imagens – cálculo de vértices, transformações e imagens aplicadas a cada um dos vértices;
Rasterização – conversão dos dados geométricos em pixels;
Operações de fragmento – cálculo dos efeitos especiais como: profundidade, transparência, reflexos, neblina, etc;
Frame Buffer – Cria-se então o frame buffer na memória. Este guarda todas as informações necessárias dos pixels para que possam ser exibidos ao usuário final;

TIPOS DE DADOS

Para ajudar ainda mais sua portabilidade de código, OpenGL possui seus próprios tipos de dados. Os identificadores fazem lembrar muito a linguagem C. A tabela a seguir demonstra os tipos mais utilizados no OpenGL:

Tipos de dado	Linguagem C	OpenGL
Inteiro de 8 bits	char	**GLbyte**
Inteiro de 16 bits	short int	**GLshort**
Inteiro de 32 bits	int	**GLint**
Ponto flutuante de 32 bits	float	**GLfloat**
Ponto flutuante de 64 bits	double	**GLdouble**
Inteiro 8 bits sem sinal	unsigned char	**GLubyte**
Inteiro 16 bits sem sinal	unsigned short	**GLushort**
Inteiro 32 bits sem sinal	unsigned int	**GLuint**

Note que sempre há o prefixo GL em seus tipos e abreviação do unsigned para u.

É importante ressaltar que a organização em suas nomenclaturas são sempre muito bem definidas, tanto em seus tipos de dados quanto nas nomenclaturas de constantes enumeradas e funções.

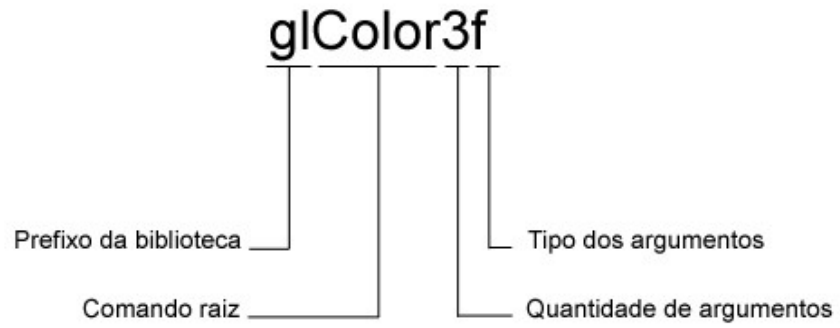
SINTAXE DAS FUNÇÕES OPENGL

Como foi dito anteriormente, OpenGL possui um padrão bem definido em todos os seus comandos. Todas suas funções são formadas pelo seguinte critério:

```
<prefixo da biblioteca> <comando raiz> <qtd de argumentos> <tipo dos argumentos>
```

Sendo que a quantidade e tipos de argumentos podem não existir.

Importante lembrar que as bibliotecas GLU e GLUT por fazerem parte do padrão OpenGL, suas funções também seguem o mesmo critério.



Prefixo da biblioteca – normalmente variam entre as três bibliotecas default do OpenGL: gl (OpenGL), glu (GLU) e glut (GLUT);

Comando raiz – podemos dizer que é onde encontramos o nome real (sentido) da função;

Quantidade de argumentos – variam sempre entre: 2, 3 e 4;

Tipos de argumentos – é tida apenas a abreviação dos tipos aceitos pela função: f (float), i (int), ub (unsigned byte), etc;

Então, é possível deduzir que a função glColor3f teria por protótipo:

```
void glColor3f (float var1, float var2, float var3);
```

Nas funções também pode vir um "v" em seu final que significa vetor, isso implica que iremos passar como parâmetro apenas o endereço do vetor, exemplo:

```
float cores [3];
glColor3fv (&cores);
```

TRATAMENTO DE EVENTOS

Como foi visto no início, OpenGL não trata eventos e muito menos manipula janelas, todo esse processo só é possível graças a GLUT. Ela possui funções para manipular janelas e trata eventos por meio de *funções callback*, estas funções são chamadas sempre que ocorre um evento no sistema tais como: pressionamento de uma tecla, redimensionamento de uma janela, clique ou movimento do mouse e intervalo de tempo no sistema (timers).

As principais funções em diversos casos são:

```
void glutDisplayFunc ( xxx );
```

chamada sempre que um pixel da janela necessita ser atualizado;
Protótipo para a função xxx:
void xxx (void);

```
void glutReshapeFunc ( xxx );
```

chamada sempre que a jenela sofrer um redimensionamento;
Protótipo para a função xxx:
void xxx (int height, int wirth);

```
void glutKeyboardFunc ( xxx );
```

chamada sempre que uma tecla ASCII for pressionada (A, f, ENTER...);
Protótipo para a função xxx:
void xxx (unsigned char key, int posX, int posY);

```
void glutSpecialFunc ( xxx );
```

chamada sempre que uma tecla especial for pressionada (Home, F1...);
Protótipo para a função xxx:
void xxx (int specialKey, int posX, int posY);

```
void glutIdleFunc ( xxx );
```

chamada sempre que nada estiver acontecendo;
Protótipo para a função xxx:
void xxx (void);

Neste documento utilizaremos apenas as funções: glutDisplayFunc, glutReshapeFunc e glutKeyboard. Estas são as funções básicas para iteração com o usuário em qualquer aplicação.

ESTRUTURA BÁSICA DE UMA APLICAÇÃO

Um programa baseado em OpenGL deve conter um mínimo de informações para que possa ser executado sem problemas. Normalmente alguns passos são seguidos:

1. Declaração de arquivos de cabeçalho;
2. Configurar a janela e abri-la;
3. Registrar funções para determinados eventos (funções callback);
4. Inicializar os estados do OpenGL;
5. Entrar no loop de tratamento de eventos;

Deixando um pouco os conceitos, vamos à pratica. O exemplo a seguir demonstra toda essa estrutura.

Exemplo 01:

```
// arquivos de cabeçalho
#include <stdlib.h>
#include <GL\gl.h>
#include <GL\glu.h>
#include <GL\glut.h>

// função de desenho
void draw ()
{
   glClear (GL_COLOR_BUFFER_BIT);

   glFlush ();
}

// função para inicializar as variaveis de estado
void init ()
{
   glClearColor (0.0, 0.0, 0.0, 0.0);
}

// função principal
int main ()
{
   // configurando e exibindo uma janela
   glutInitDisplayMode (GLUT_SINGLE | GLUT_RGBA);
   glutCreateWindow ("Exemplo 01: exibindo uma janela");

   // registrando função de callback
   glutDisplayFunc (draw );

   // inicialização das variáveis de estado
   init ();

   // loop de tratamento de eventos
   glutMainLoop ();

   return 0;
}
```

Analisando o código:

Arquivos de cabeçalho – nesse trecho são adicionados os arquivos necessários para a execução de um programa: gl.h, glu.h e glut.h.

Configurando e exibindo uma janela – glutInitDisplayMode configura o modo de como a janela vai ser tratada: se utilizará buffer simples ou duplo, qual sistema de cores utilizará, se utilizará buffers auxiliares, etc. Em nosso exemplo estamos utilizando buffer simples e o RGBA (Red, Green, Blue e Alpha) como sistema de coloração. glutCreateWindow abre uma janela tendo por título a string passada por parâmetro;

Registrando funções de callback – glutDisplayFunc registra a função passada por parâmetro como sendo a função callback de desenho, logo, sempre que um pixel necessitar ser atualizado (ou "repintado"), será chamada a função registrada, neste exemplo, a função draw;

Inicialização das variáveis de estados – normalmente é utilizada uma função para tal tarefa, em nosso exemplo nomeamos esta função de init devido servir apenas para inicializar variáveis.

Loop de tratamento de eventos – nesse trecho o programa entra em um loop "infinito" esperando que algum evento ocorra e se houver, chama a função de callback responsável pelo tratamento do evento;

glClear – função responsável por "limpar" buffers, em nosso caso, a utilizamos para limpar o buffer de coloração (GL_COLOR_BUFFER_BIT). Está função normalmente é utilizada sempre antes dos comandos de desenhos;

glFlush – função responsável por forçar a exibição de todos os comandos de desenho pois, nem sempre todos o comandos chegam a ser exibidos na tela, então temos a necessidade de força-los a ser "transferidos" para a tela;

glClearColor – está função possui quatro argumentos: Red, Green, Blue e Alpha. Os três primeiros definem a cor que o fundo da janela irá assumir quando for limpo e Alpha corresponde ao nível de transparência, por default todos são possuem valor zero;

PRIMITIVAS OPENGL

OpenGL em sua forma básica, suporta apenas primitivas simples como: pontos, segmento de retas, triângulos, quadrados e polígonos. No entanto, fazendo bom uso destas primitivas podemos chegar a desenhos mais complexos, vários exemplos disto são postos nas bibliotecas GLU e GLUT que são capazes de desenhar objetos complexos apenas com primitivas simples do OpenGL, como por exemplo, desenhar esfera, cubo, cone, etc.

Todas as primitivas são compostas unicamente por vértices, sendo necessário definir que primitiva os vértices irão formar. As primitivas são formadas pelo seguinte código:

```
glBegin ( <tipo da primitiva> );
   < vértices da primitiva >
glEnd ();
```

Tipo da primitiva – pode assumir os seguintes valores: GL_POINTS, GL_LINES, GL_LINE_LOOP, GL_LINE_STRIP, GL_QUADS, GL_QUAD_STRIP, GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN e GL_POLYGON;

Vértices da primitiva – nesta sessão devemos indicar os vértices para que a primitiva seja formada. Cada vértice é indicado pela função `glVertex` que pode receber dois ou três parâmetros: glVertex2f ou glVertex3f;

Como exemplo, vamos exibir uma janela que contenha um quadrado em seu centro. Alterando algumas funções do exemplo anterior, temos:

Exemplo 02:

```
// função de desenho
void draw ()
{
   glClear (GL_COLOR_BUFFER_BIT);

   // alterando a cor de desenho para vermelho
   glColor3f (1.0, 0.0, 0);

   // iniciando o desenho do quadrado
   glBegin (GL_QUADS);
      glVertex2f (-0.5, -0.5);
      glVertex2f ( 0.5, -0.5);
      glVertex2f ( 0.5,  0.5);
      glVertex2f (-0.5,  0.5);
   glEnd ();

   glFlush ();
}
```

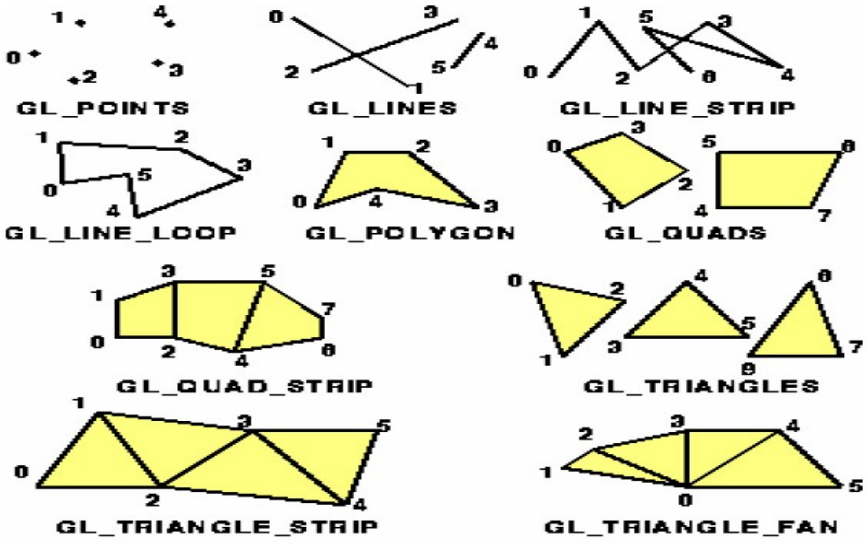
Analisando o código:

glColor3f – função com três parâmetros do tipo float (GLfloat): Red, Green e Blue, sendo que, cada valor varia entre 0.0 e 1.0. Esta é responsável por mudar a cor atual de desenho;

glBegin – função que marca o início da primitiva passada por parâmetro. Todos os vértices (glVertex) que estiverem entre glBegin e glEnd, formarão a primitiva passada em glBegin;

glVertex2f – função com dois parâmetros do tipo float que formarão um ponto no espaço, por ter apenas duas variáveis (X e Y) os vértices estarão em um espaço 2D. Na realidade, glVertex2f é o mesmo que glVertex3f, isto devido ao OpenGL considerar a terceira coordenada zero; então, glVertex2f (1.0, 1.0) é o mesmo que glVertex3f (1.0, 1.0, 0.0);

A seguir temos uma ilustração para cada tipo de primitiva existente no OpenGL:



COLORAÇÃO

No exemplo anterior podemos perceber que trabalhamos em modo RGBA (glutInitDisplayMode (... | GLUT_RGBA)), pois bem, o modo RGBA é bem conhecido devido sua utilização em diversas aplicações gráficas (gimp, photoshop, CorelDraw, ...) e sabemos que sua cor final é obtida através da "mistura" das cores primárias: vermelho, verde e azul. Em OpenGL, diferente das muitas aplicações gráficas, não trabalha com valores de 0 à 255 para cada cor primária, mas sim, entre 0.0 e 1.0. A variável Alpha tem grande importância, se tratando de transparência, seu valor também varia de 0.0 à 1.0 e refere-se ao grau de opacidade do objeto.

Em se tratando de preenchimento de objetos, OpenGL possui dois modos:

GL_SMOOTH – a cor final é o resultado de uma interpolação entre as cores de cada vértice;

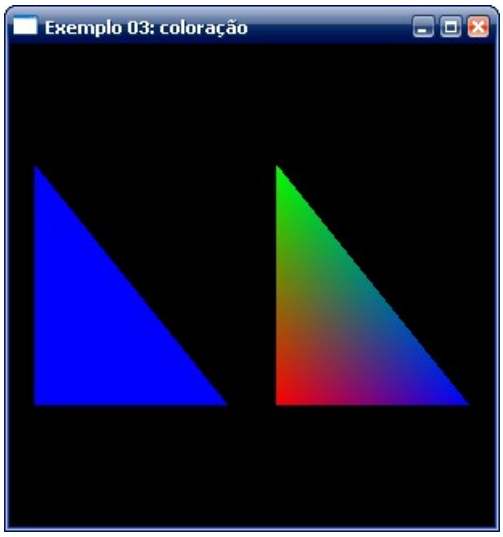
GL_FLAT – a cor final é exatamente a mesma cor do ultimo vértice a ser desenhado;

O modo de preenchimento é definido por meio da função glShadeModel (<modo>).

Alterando a função de desenho draw do exemplo anterior faremos uma demonstração simples. Nosso exemplo irá desenhar dois triângulos com o mesmo tamanho e as mesmas cores nos vértices, tendo apenas sua posição e o modo de preenchimento diferente (o esquerdo será preenchido com GL_FLAT e o direito com GL_SMOOTH), vejamos:

Exemplo 03:

```
// função de desenho
void draw ()
{
   glClear (GL_COLOR_BUFFER_BIT);

   // desenhando o triangulo da esquerda
   glShadeModel (GL_FLAT);
   glBegin (GL_TRIANGLES);
      glColor3f  ( 1.0,  0.0,  0.0);
      glVertex3f (-0.9, -0.5,  0.0);
      glColor3f  (-0.1,  1.0,  0.0);
      glVertex3f (-0.9,  0.5,  0.0);
      glColor3f  ( 0.0,  0.0,  2.0);
      glVertex3f (-0.1, -0.5,  0.0);
   glEnd ();

   // desenhando o triangulo da direita
   glShadeModel (GL_SMOOTH);
   glBegin (GL_TRIANGLES);
      glColor3f  ( 1.0,  0.0,  0.0);
      glVertex3f ( 0.1, -0.5,  0.0);
      glColor3f  ( 0.0,  1.0,  0.0);
      glVertex3f ( 0.1,  0.5,  0.0);
      glColor3f  ( 0.0,  0.0,  1.0);
      glVertex3f ( 0.9, -0.5,  0.0);
   glEnd ();

   glFlush ();
}
```



Analisando o código:

Pelos comandos utilizados, podemos perceber que cada vértice pode assumir sua própria cor e que o modo de preenchimento do OpenGL é que faz toda a diferença;

Importante ressaltar que, pelo que podemos ver, o modo smooth é bem mais interessante e por isso, é tido como default, ou seja, não há necessidade de fazer uma chamada à função glShadeModel dentro da função init por exemplo.

CAMPO DE VISULIZAÇÃO

Para utilizarmos o potencial 3D (e até mesmo 2D) do OpenGL, obrigatoriamente temos que definir uma porção de coisas pois, o que fazer para que uma porção do mundo 3D (imagem) seja exibida em 2D (tela do PC)?

Todo o controle de visualização do espaço OpenGL é feito através de duas pilhas:

Modelview – responsável pela modelagem e visualização da cena;

Projection – responsável pela projeção da cena: ortogonal ou perspectiva;

Por ser manipulada basicamente por pilhas, OpenGL possui grande facilidade de construir modelos hierárquicos, marcando o conceito de máquina de estado;

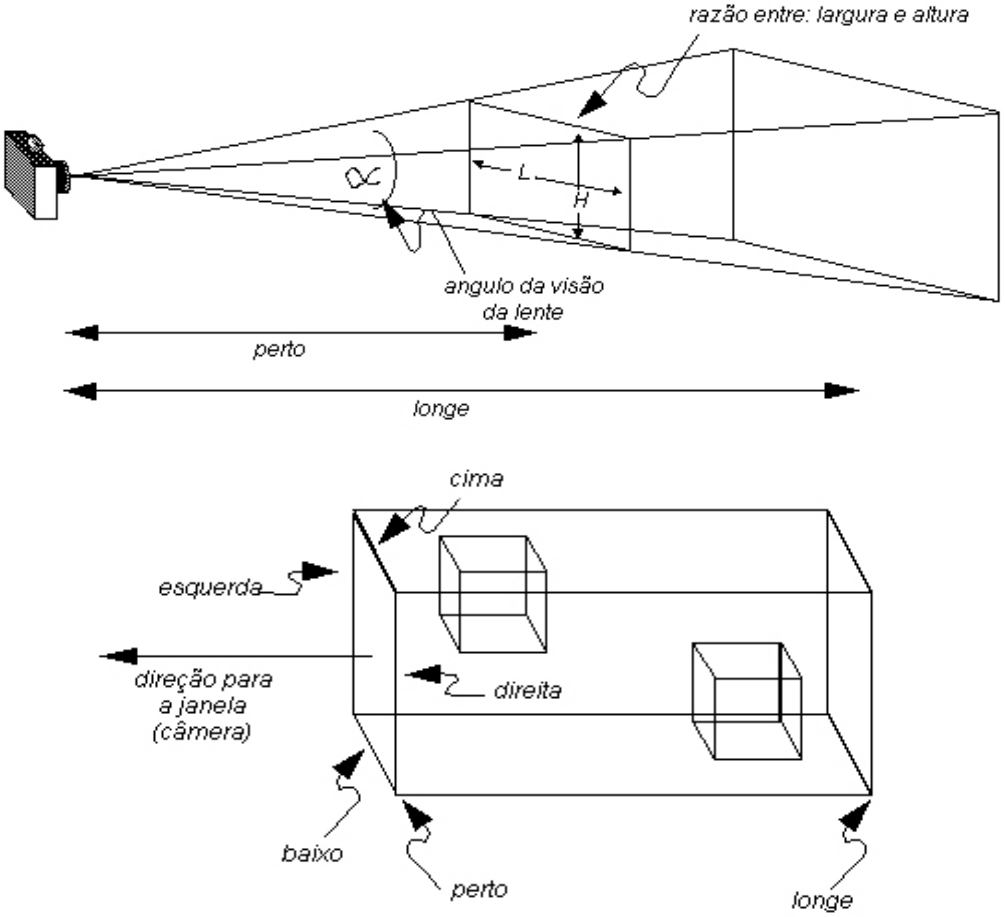
Para definirmos qual pilha iremos trabalhar devemos chamar à função glMatrixMode (<pilha>); sendo que, pilha em seu modo básico pode ser GL_MODELVIEW ou GL_PROJECTION.

Como dito agora pouco, OpenGL suporta dois tipos de projeção:

Ortogonal – define um volume de visualização onde a projeção do objeto não é afetada pela sua distância em relação ao observador (câmera); utilizada bastante em aplicações voltada à arquitetura, onde as medidas são o que realmente importam;

Perspectiva – define um volume de visualização onde a projeção do objeto é reduzida à medida que ele é afastado do observador; se encaixa perfeitamente em nossa realidade, portanto é o mais utilizado em jogos, animações, filmes, etc;

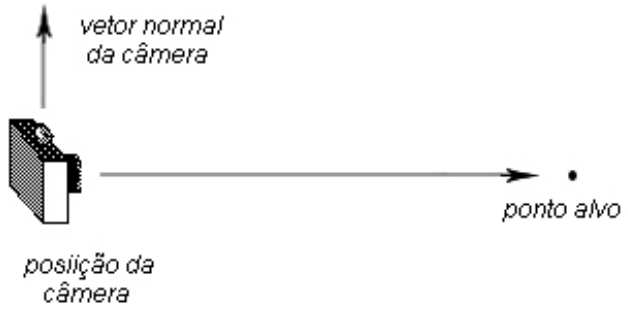
A seguir temos uma ilustração da projeção perspectiva e ortogonal respectivamente:



Além de definirmos um campo de visualização, temos que definir a posição da câmera, seu alvo e sentido, pois ao definir o campo de visualização a posição da câmera é a mesma do alvo, ou seja, não é gerada nenhuma imagem de saída a não ser a própria cor de

fundo da janela. Para configurar tais variáveis a GLU nos oferece uma função bem simples e poderosa:

gluLookAt – esta função recebe seis (6) parâmetros, os três primeiros formam a coordenada da câmera, os três seguintes a posição do alvo e os últimos definem um vetor normal para o sentido da câmera.



A seguir veremos o código necessário para que possamos manipular o campo de visualização e o posicionamento da câmera:

```
// configura o campo de visualização
void confCamera ()
{
   glMatrixMode (GL_PROJECTION);
   glLoadIdentity ();
   gluPerspective (45, 1.0, 0.1, 100);
   //glOrtho (-2, 2, -2, 2, 1, 5);
   glMatrixMode (GL_MODELVIEW);
   glLoadIdentity ();
   gluLookAt (1,1,2, 0,0,0, 0,1,0);
}
```

glLoadIdentity – função responsável por multiplicar a matriz atual pela identidade. Serve para simplificar (anular) a matriz atual voltando ao estado default;

gluPerspective – função responsável por dar uma projeção perspectiva aos objetos; o primeiro parâmetro refere-se ao ângulo de curvatura da lente da câmera, o segundo é normalmente a razão entre a largura e altura da janela (distorção da lente) e o terceiro e quatro correspondem respectivamente à menor e maior distância visualizável, referente à câmera;

glOrtho – função responsável por dar uma projeção ortogonal aos objetos; seus parâmetros correspondem respectivamente: esquerda, direita, baixo, cima, próximo e longe, isso referente à própria janela;

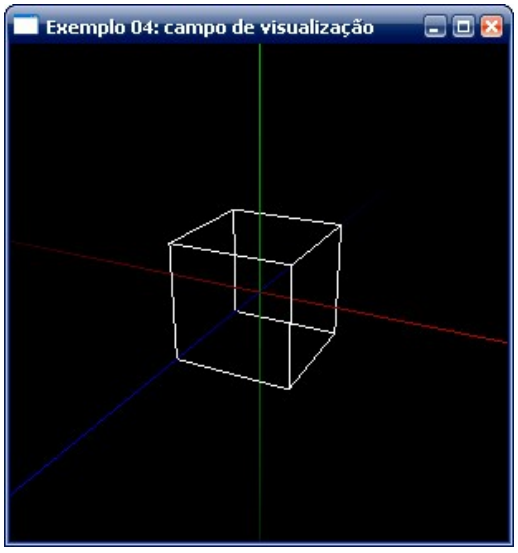
Importante dizer que, nunca, devemos usar gluPerspective e glOrtho juntos. Na realidade nem poderia, como ter uma projeção ortográfica e perspectiva ao mesmo tempo? Isso seria algo ilógico.

Como demonstração, montaremos uma cena em que é desenhado um plano cartesiano e um cubo (desenhado apenas por linhas) no centro do plano. Para tal, alteraremos o código anterior para o seguinte:

Exemplo 04:

```
// função de desenho
void draw ()
{
   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

   // desenhando o plano cartesiano 3D
   glBegin (GL_LINES);
      // eixo X
```

```
      glColor3f ( 0.0, 0.0, 0.0);
      glVertex3f(-2.0, 0.0, 0.0);
      glColor3f ( 1.0, 0.0, 0.0);
      glVertex3f( 2.0, 0.0, 0.0);
      // eixo Y
      glColor3f (0.0, 0.0, 0.0);
      glVertex3f(0.0,-2.0, 0.0);
      glColor3f (0.0, 1.0, 0.0);
      glVertex3f(0.0, 2.0, 0.0);
      // eixo Z
      glColor3f (0.0,  0.0, 0.0);
      glVertex3f(0.0,  0.0,-2.0);
      glColor3f (0.0,  0.0, 1.0);
      glVertex3f(0.0,  0.0, 2.0);
   glEnd ();

   // desenhando um cubo com linhas brancas
   glColor3f (1.0, 1.0, 1.0);
   glutWireCube (0.5);

   glFlush ();
}

// configura o campo de visualização
void confCamera ()
{
   glMatrixMode (GL_PROJECTION);
   glLoadIdentity ();
   gluPerspective (45, 1.0, 0.1, 100);
   //glOrtho (-2, 2, -2, 2, 1, 5);
   glMatrixMode (GL_MODELVIEW);
   glLoadIdentity ();
   gluLookAt (1,1,2, 0,0,0, 0,1,0);
}

// função para inicializar as variaveis de estado
void init ()
{
   glClearColor (0.0, 0.0, 0.0, 0.0);
   confCamera ();
   glEnable (GL_DEPTH_TEST);
}
```



Analisando o código:

glClear – função já abordada anteriormente. Responsável por "limpar" buffers, em nosso caso, como estamos utilizando o potencial 3D do OpenGL a pintura da tela não mais depende apenas da coloração dos pixels pois, como diferenciar qual pixel está na frente do outro, já que o espaço é 3D? Para isso OpenGL possui o depth buffer (buffer de profundidade) que armazena a informação do eixo Z (profundidade) para cada pixel possibilitando diferir qual está na frente ou atrás de acordo com a posição da câmera e do alvo; então, GL_DEPTH_BUFFER_BIT refere-se a este buffer;

desenhando o plano cartesiano 3D – aqui definimos a primitiva de linhas GL_LINES para cada par de vértice formar uma linha (o eixo) e atribuímos uma cor diferente para cada eixo: X (vermelho), Y (verde) e Z (azul). Sendo que, cada linha é iniciada pela cor preta (número negativo) até sua cor real (número positivo);

glutWireCube – função da GLUT que desenha um cubo predefinido formado por arestas. Seu parâmetro representa o raio;

ConfCamera – função definida para configurar o campo de visualização e posicionamento da câmera. Definimos esta função no início deste tópico;

glEnable (GL_DEPTH_TEST) – habilita o teste de profundidade para que um objeto distante não fique por cima de um que se encontra mais próximo em relação à câmera.

Neste exemplo podemos perceber claramente a noção de perspectiva da cena devido ao posicionamento da câmera.

Problema encontrado: ao redimensionarmos a janela percebemos que o cubo não mais pode ser chamado de cubo, isso devido ao tamanho das arestas que em muitos casos não aparentam ter o mesmo tamanho. O problema é que a janela precisa ter altura e largura iguais, o que raramente acontece depois da janela ser redimensionada. Uma solução para esse problema seria usar a função callback para redimensionamento da janela...

CONFIGURANDO A CENA AO REDIMENSIONAR A JANELA

Para evitar a distorção da cena quando sua janela for redimensionada temos uma solução simples, obtendo o tamanho da janela (largura e altura) podemos conseguir a razão entre a largura e altura, o que lembra a lente da câmera! No segundo argumento da função gluPerspective temos a razão entre a largura e altura da janela que na verdade serão os valores da janela atual.

A GLUT possui uma função callback para o redimencionamento da janela e possui dois parâmetros: largura e altura. Logo, para cada redimensionamento temos que redefinir a razão destes valores e atualiza-los na função gluPespective.

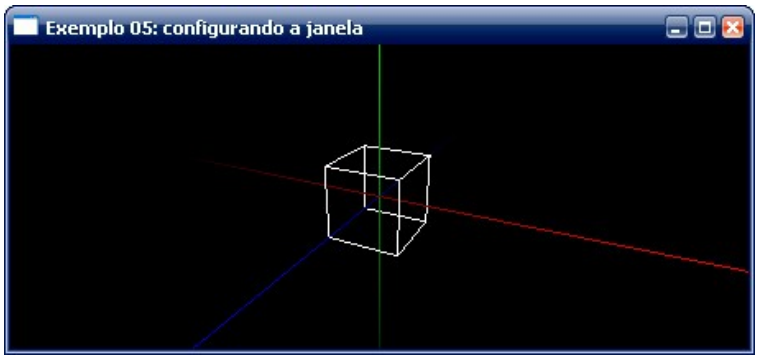
Também teremos que fazer uso de uma função:

glViewport – função responsável por definir qual o tamanho da "foto" que aparecerá na janela, bastante útil quando se deseja exibir quadros de exibições diferentes em uma única janela. Seus dois primeiros parâmetros correspondem ao canto inferior esquerdo da foto e os dois últimos ao superior direito. Em se tratando de uma janela com uma única foto, temos sempre os seguintes valores: 0, 0, widthWindow e heightWindow.

Tendo por base o exemplo anterior, a configuração ficaria mais ou menos assim:

Exemplo 05:

```
// guardará a razão atual da largura e altura da janela
float aspecto = 1.0;

// configura o campo de visualização
void confCamera ()
{
   glMatrixMode (GL_PROJECTION);
   glLoadIdentity ();
   gluPerspective (45.0, aspecto , 0.1, 100);
   glMatrixMode (GL_MODELVIEW);
   glLoadIdentity ();
   gluLookAt (1,1,2, 0,0,0, 0,1,0);
}

// funçao para quando houver redimensionamento da janela
void resize (int width, int height)
{
   if (height == 0)
      height = 1;
   aspecto = (float) width / height;
   glViewport (0, 0, width, height);
   confCamera ();
}

// função principal
int main ()
{
   // ...
   // registrando função de callback
   glutDisplayFunc (draw );
   glutReshapeFunc (resize);
   // ...
}
```


Exemplo 05: configurando a janela

OBJETOS PREDEFINIDOS DA GLUT

Além de gerenciar eventos e manipular janelas, a GLUT possui algumas funções predefinidas para desenhar: esfera, cubo, cone, etc. Suas funções para desenhos obedecem ao seguinte critério: glut + <modo> + <objeto>; Sendo que <modo> - varia entre Wire (desenhado por arestas) e Solid (desenhado por polígonos).

Os objetos mais utilizados são:

Cubo:

```
glutSolidCube (raio);
```

Cone:

```
glutSolidCone (raio, altura, qtd de planos, qtd de fatias);
```

Esfera:

```
glutSolidSphere (raio, qtd de planos, qtd de fatias);
```

Toróide (rosquinha):

```
glutSolidTorus (raio interno, raio externo, qtd de planos, qtd
de fatias);
```

Chaleira:

```
glutSolidTeapot (raio);
```

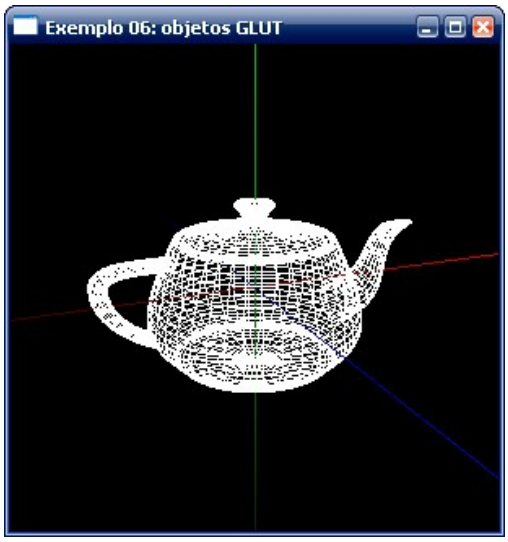
Para exemplificar um de seus objetos iremos desenhar uma cena com o objeto clássico da GLUT por conter em diversos tutoriais, o Teapot.

Nos exemplos anteriores vimos que podemos definir onde a câmera e alvo são posicionados. No entanto, a configuração da câmera foi feita na função de inicialização init, e se quiséssemos uma cena dinâmica onde a câmera pudesse se locomover na cena com a iteração do usuário? Uma solução seria definirmos variáveis para representar a posição da câmera e configura-la quando o usuário interagir com o programa. Neste exemplo teremos uma função callback de teclado, pois utilizaremos as teclas X, x, Y, y, Z e z para alterar a coordenada nos eixos correspondentes da câmera. Vejamos uma implementação desta idéia:

Exemplo 06:

```
// velocidade com que a camera se desloca
#define SPEED_CAMERA  0.1

// guarda a posição atual da camera
float CamX = 1.0;
float CamY = 1.0;
float CamZ = 2.0;

// configura o campo de visualização
void confCamera ()
{
   glMatrixMode (GL_PROJECTION);
   glLoadIdentity ();
   gluPerspective (45.0, aspecto , 0.1, 100);
   glMatrixMode (GL_MODELVIEW);
   glLoadIdentity ();
   gluLookAt (CamX,CamY,CamZ, 0,0,0, 0,1,0);
}

// função responsável por eventos de teclado
void keyboard (unsigned char key, int posX, int posY)
{
   switch (key) {
      case 27 :
         exit (0);
         break;
```

```
      case 'X' :
         CamX += SPEED_CAMERA;
         break;
      case 'x' :
         CamX -= SPEED_CAMERA;
         break;
      case 'Y' :
         CamY += SPEED_CAMERA;
         break;
      case 'y' :
         CamY -= SPEED_CAMERA;
         break;
      case 'Z' :
         CamZ += SPEED_CAMERA;
         break;
      case 'z' :
         CamZ -= SPEED_CAMERA;
         break;
   }
   // repinta a tela
   glutPostRedisplay ();
}

// função de desenho
void draw ()
{
   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

   confCamera ();

   // ...

   // desenhando o Teapot (chaleira)
   glColor3f (1.0, 1.0, 1.0);
   glutWireTeapot (0.5);

   glFlush ();
}

// função principal
int main ()
{
   // ...
   // registrando função de callback
   glutDisplayFunc (draw );
   glutReshapeFunc (resize);
   glutKeyboardFunc(keyboard);
   // ...
}
```



Analisando o código:

Keyboard – função callback para tratamento de eventos do teclado, possui obrigatoriamente três parâmetros, o primeiro é o código ASCII da tecla digitada e os dois seguintes representam a posição do mouse quando a tecla for pressionada (X, Y);

glutPostRedisplay – função responsável por forçar uma nova exibição da cena, ou seja, uma chamada à função callback de desenho;

confCamera – esta função é chamada dentro da função de desenho porque a mudança nas coordenadas da câmera devem ser configuradas à cada exibição da cena;

glutWireTeapot – função da GLUT que desenha uma chaleira predefinida em forma de arestas. Seu parâmetro representa seu raio;

glutKeyboardFunc – função da GLUT para registrar a função callback de teclado. A função registrada é chamada sempre que uma tecla ASCII for pressionada;

SPEED_CAMERA – é uma MACRO com propósito bem intuitivo, determina a velocidade com que a câmera irá se mover;

CamX, CamY e CamZ – armazenam a posição da câmera no espaço 3D.

Interessante seria adicionar variáveis para controlar também o alvo da câmera, como por exemplo: AlvoX, AlvoY e AlvoZ;

TRANSFORMAÇÕES GEOMÉTRICAS

OpenGL em sua forma básica suporta três tipos de operações geométricas: translação, rotação e escalonamento. Em transformações geométricas é que podemos ver realmente o conceito de máquina de estado na prática, pois, uma vez mudado o centro do sistema de coordenadas para outro ponto no espaço, rotacionado ou escalonado, suas mudanças passam a ser consideradas o estado inicial (atual).

Translação:

glTranslatef – função responsável por mudar o posicionamento do centro do espaço. Possui três parâmetros: X, Y e Z. O centro do espaço será movido para a coordenada passada por parâmetro;

Rotação:

glRotatef – função responsável por girar o plano cartesiano 3D em seu cento. Possui quatro parâmetros: Ângulo, X, Y e Z. Ângulo representa a quantidade de graus que será rotacionado e X, Y e Z representam em quais eixos será realizado o giro;

Escalonamento:

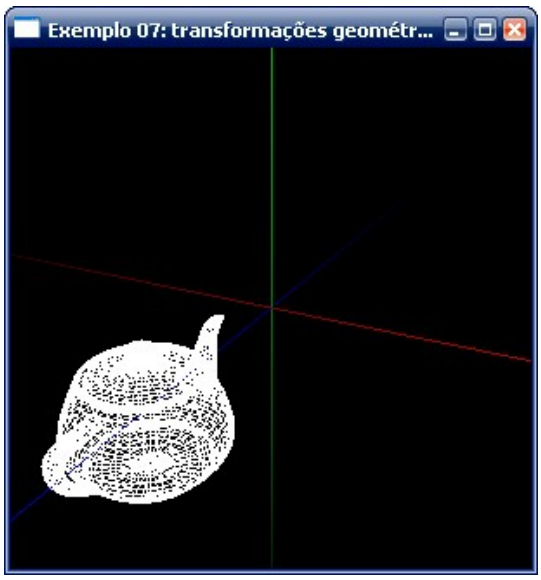
glScalef – função responsável por alterar o tamanho dos eixos do plano 3D. Possui três parâmetros: X, Y e Z. Cada variável está intimamente ligada ao seu eixo correspondente, sendo que internamente, cada valor é multiplicado pelo tamanho de seu eixo. Sendo assim, se quiséssemos um sistema que tivesse o dobro do anterior em todos os eixos teríamos a seguinte chamada: glScalef (2.0, 2.0, 2.0);

Com o exemplo anterior em mãos faremos algumas mudanças, para que tenhamos uma cena dinâmica com operações geométricas, faremos uso da função callback de teclado para termos iteração com o usuário, as teclas utilizadas serão: R, r (rotação), T, t (translação) e E, e (escalonamento):

Exemplo 07:

```
// guarda as transformações para a cena
float RotatAngulo = 0.0;
float TransPosX   = 0.0;
float ScaleEixos  = 1.0;

// função responsável por eventos de teclado
void keyboard (unsigned char key, int posX, int posY)
{
   switch (key) {
      // saindo do programa
      // ...
      // posicionamento da camera
      // ...
      // transformações geometricas
      case 'R' :
         RotatAngulo += 1.0;
         break;
      case 'r' :
         RotatAngulo -= 1.0;
         break;
      case 'T' :
         TransPosX += 0.1;
         break;
      case 't' :
         TransPosX -= 0.1;
         break;
      case 'E' :
```

```
            ScaleEixos += 0.1;
            break;
         case 'e' :
            ScaleEixos -= 0.1;
            break;
      }
      // repinta a tela
      glutPostRedisplay ();
}

// função de desenho
void draw ()
{
      // ...
      glEnd ();

      // realizando as transformações
      glRotatef    (RotatAngulo, 0.0, 1.0, 0.0);
      glTranslatef (TransPosX, 0.0, 0.0);
      glScalef     (ScaleEixos, ScaleEixos, ScaleEixos);

      // desenhando uma chaleira com linhas brancas
      glColor3f (1.0, 1.0, 1.0);
      glutWireTeapot (0.5);

      glFlush ();
}
```



Analisando o código:

Definição de variáveis – definimos três variáveis para controlas as transformações: RotatAngulo, TransPosX e ScaleEixos. Seus nomes são bem sugestivos para suas tarefas.

glRotatef – pela sua chamada implicar dizer que, o sistema de coordenadas sofrerá um giro de RotatAngulo graus nos eixos X, Y e Z. Tendo X e Z valor nulo, e apenas Y um valor diferente de zero (1), o sistema sofrerá o giro apenas no eixo Y da matriz atual;

glTranslatef – desloca o centro de coordenadas para a posição passada por parâmetro. Tendo Y e Z valor nulo, e apenas X um valor diferente de zero (1), o sistema sofrerá um deslocamento apenas no eixo de X da matriz atual;

glScalef – escalona os três eixos atuais em relação aos valores passados por parâmetro, importante observar que todos os parâmetros são os mesmo, logo, os desenhos não irão perder sua proporção;

Importante ressaltar que todas as transformações geométricas atuam sobre a matriz atual, ou seja, as operações devem possuir uma ordem adequada para que se obtenha a cena desejada.

BUFFERS

Sabemos que OpenGL trabalha com buffers para guardar a coloração dos pixels e guardar a coordenada Z (profundidade), no entanto, OpenGL possui outros buffers como por exemplo o stencil de acumulação e alguns outros. Estes servem para diversos motivos, principalmente para efeitos em animações e cenas realísticas.

No exemplo anterior vimos que ao movimentar-mos a posição da câmera ou até mesmo realizar transformações geométricas, fica bem visível a presença de uns tracejados na tela (dependendo do PC), o que não é nada agradável para uma cena.

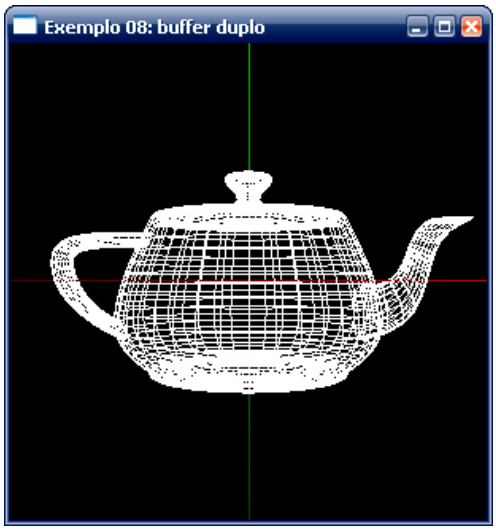
Isto acontece sempre que utilizamos somente um buffer para desenhar a cena (buffer simples), pois, o buffer contido na memória possui o mesmo desenho de sua janela. Na verdade, aparecem os tracejados porque basicamente chegamos a ver o OpenGL "pintando" nossa janela!

Para evitar esse tipo de problema OpenGL suporta trabalhar com buffer duplo, logicamente consome mais memória mas, sua cena fica bem desenhada. Utilizando o buffer duplo, o OpenGL pinta todos os pixels em um buffer auxiliar e somente quando desenha todos pixels faz a troca com o buffer da janela. Isso garante uma cena sem "chuviscos".

Alterando o exemplo anterior veremos como utilizá-lo:

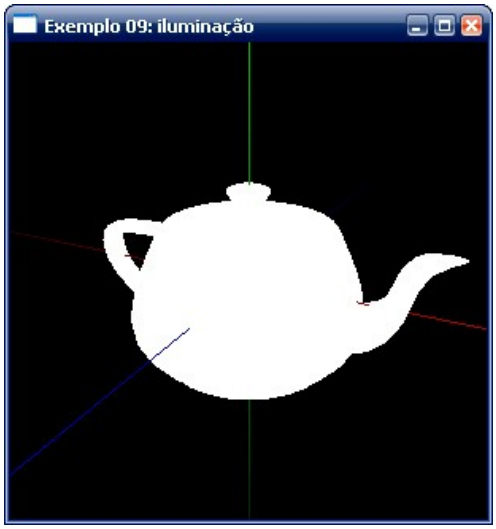
Exemplo 08:

```
// função de desenho
void draw ()
{
   // ...
   glFlush ();
   glutSwapBuffers ();
}

// função principal
int main ()
{
   // configurando e exibindo uma janela
   glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGBA);
   glutCreateWindow ("Exemplo 08: buffer duplo");
   // ...
   return 0;
}
```



ILUMINAÇÃO

Até agora utilizamos desenhos predefinidos da GLUT apenas em modo wire, qual seria o resultado se um objeto fosse desenhado em modo solid? Com o exemplo anterior, alterando apenas o comando glutWireTeapot (0.5) para glutSolidTeapot (0.5) teríamos uma cena como a seguinte:



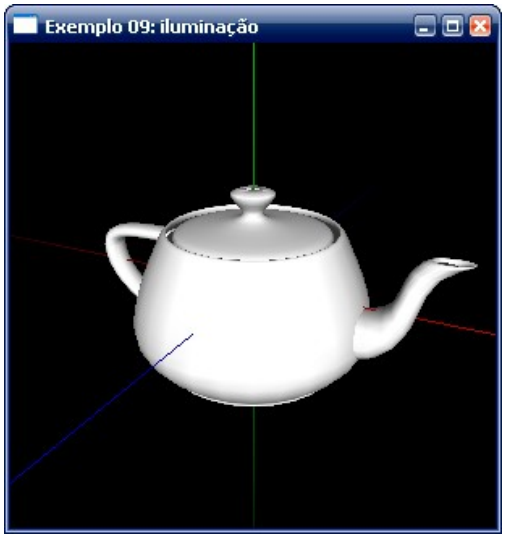
A diferença é que ao invés de linhas, o objeto é desenhado através de polígonos, no entanto, parece ser um desenho 2D, pois não temos a noção de volume do objeto. Isso não acontece apenas no mundo virtual (OpenGL) mas no real, quando estamos no escuro nós chegamos a ver somente o "vulto" do objeto, mas o volume dele fica apenas na imaginação pois, com a presença da luz podemos ver o volume do objeto graças aos à luz que o objeto é capaz de emitir.

OpenGL suporta a utilização de até oito (8) fontes de luz diferente ao mesmo tempo, em nossos exemplos faremos uso de apenas uma dessas. Alterando apenas duas funções do exemplo anterior podemos conseguir uma iluminação em nossa cena:

Exemplo 09:

```
// função de desenho
void draw ()
{
   // ...
   // realizando as transformações
   // ...
   // desenhando uma chaleira com cor branda
   glColor3f (0.9, 0.4, 0.1);
   glutSolidTeapot (0.5);

   glFlush ();
   glutSwapBuffers ();
}

// função para inicializar as variaveis de estado
void init ()
{
   glClearColor (0.0, 0.0, 0.0, 0.0);
   confCamera ();
   glEnable (GL_DEPTH_TEST);
```

```
   // habilitando a iluminação e cor dos materiais
   glEnable (GL_COLOR_MATERIAL);
   glEnable (GL_LIGHTING);
   glEnable (GL_LIGHT0);
}
```



Analisando o código:

glutSolidTeapot – função da GLUT predefinida para desenhar uma chaleira em modo sólido;

glEnable (GL_COLOR_MATERIAL) – habilita a cor dos objetos em iluminação pela função glColor, sem este comando o glColor simplesmente não tem qualquer efeito sobre uma cena iluminada;

glEnable (GL_LIGHT0) – habilita a luz de número zero (0) sobre a cena. OpenGL suporta por padrão até oito fontes de luz, estas variam de GL_LIGHT0, GL_LIGHT1 até GL_LIGHT7;

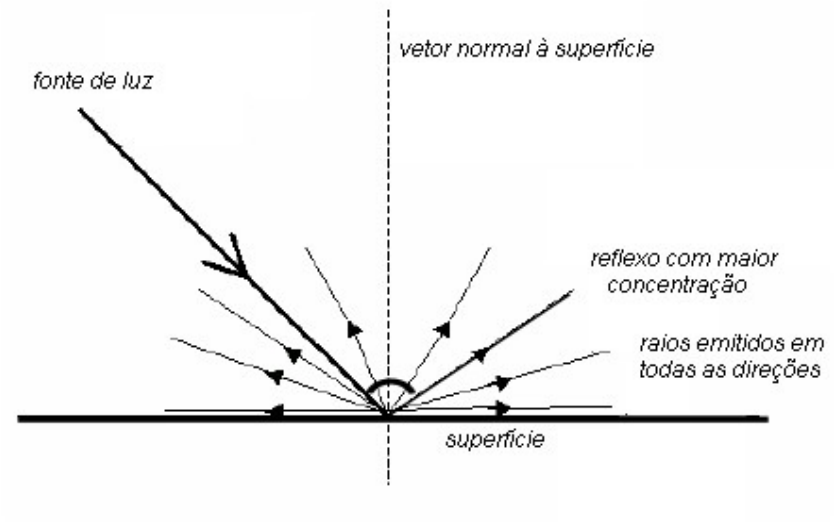
glEnable (GL_LIGHTING) – habilita a iluminação geral da cena. Este comando é sempre utilizado quando pretendemos iluminar a cena, sem ele, mesmo tendo as oito luzes em uso, todas estão apagadas e só acendem quando se habilita GL_LIGHTING;

CONFIGURANDO A ILUMINAÇÃO

Vimos que a iluminação pode ser facilmente habilitada, mas onde se encontra a fonte de luz? Possui alguma cor?

OpenGL controla as fontes de luz por meio de propriedades tais como: posição, cor, intensidade, etc. Algo que devemos esclarecer é que por possuir posição não implica dizer que possui um volume (objeto desenhado), OpenGL apenas guarda uma posição para que a partir dela, calcule-se os reflexos (a cor de cada face) dos objetos sobre a cena.

Como OpenGL sabe se uma superfície n reflete luz à câmera? A seguir temos uma figura demonstrando mais ou menos isso:



Na figura o que mais interessa é que a direção do reflexo de maior concentração, que depende exclusivamente do ângulo formado entre a fonte de luz e a normal da superfície. Ao desenharmos primitivas do OpenGL em cenas iluminadas, para que tenhamos uma reflexão de superfícies adequada temos que dizer ao OpenGL quando estivermos desenhando a primitiva o vetor normal daquela superfície utilizando a função glNormal3f. Lembrando que o vetor normal é composto por apenas uma unidade, ou seja, pela matemática podemos encontrá-lo logo após encontrar as três coordenadas do plano e em seguida, calcular-mos o vetor perpendicular à superfície que muita das vezes não possui uma unidade exata, então temos que redimensiona-lo. Isso tudo é porque o vetor precisa ser de apenas uma unidade, caso contrário, resultados indesejáveis virão!

Ao fazer uma transformação geométrica de escala sabemos que o tamanho do objeto pode aumentar ou diminuir, consequentemente, o vetor normal passa a ser um vetor com tamanho qualquer. Esse problema OpenGL resolve para nós, basta utilizar o comando glEnable (GL_NORMALIZE); que ao escalonar um objeto ele já configura-o, isso se ele contiver realmente um vetor normal.

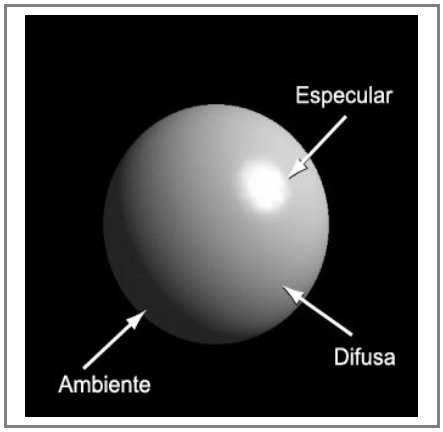
Para configurar as propriedades da iluminação utilizamos a função glLight que possui três parâmetros: Light, Property e Param. Light é a luz a ser configurada, Property é a propriedade a ser posta e Param é o valor para a propriedade Property, este na maioria das vezes é utilizado como um ponteiro. Algumas propriedades da fonte de luz são:

GL_AMBIENT – configura a luz ambiente, luz que ilumina toda a cena (vem de todas as direções);

GL_DIFFUSE – configura a luz difusa (intensidade da luz), vem de um ponto e reflete em todas as direções;

GL_SPECULAR – configura a luz especular, vem de um ponto e reflete em um único ponto;

GL_POSITION – configura a posição da fonte de luz (X, Y e Z);



Importante mencionar que as propriedades: Ambiente e Difusa, "formam" a cor do objeto e Especular o brilho do objeto. Todas as propriedades demonstradas possuem quatro (4) parâmetros, incluindo a posição da luz, este normalmente possui valor 1.0.

Além de definir a fonte de luz, precisamos definir as propriedades dos objetos a serem desenhados, seu potencial de brilho etc. Para configurar os materiais utilizamos a função glMaterial que possui três (3) parâmetros: Face, Property, Param. Face é a face que será utilizada (GL_FRONT, GL_BACK ou GL_FRONT_AND_BACK), Property é a propriedade a ser posta e Param é o valor para a propriedade Property. Utilizamos basicamente duas (2) propriedades para os materiais:

GL_SPECULAR – configura a cor do brilho;

GL_ SHININESS – configura o potencial de brilho do material;

Alteraremos o exemplo anterior para demonstrar e entender melhor os efeitos de iluminação e materiais:

Exemplo 10:

```
// velocidade com que a "luz" se desloca
#define SPEED 0.1
// guardará a razão atual da largura e altura da janela
float aspecto = 1.0;
// guarda a configuração da iluminacao e materiais
float luz_ambiente [] = { 0.1, 0.1, 0.1, 1.0};
float luz_difusa []   = { 0.5, 0.5, 0.5, 1.0};
float luz_especular []= { 0.9, 0.9, 0.9, 1.0};
float luz_posicao []  = { 0.0, 1.0, 0.0, 1.0};
float mat_especular []= { 1.0, 1.0, 1.0, 1.0};
float mat_brilho      = 60.0;

// configura as luzes e os materiais
void confLuzMat ()
{
   // iluminacao
   glLightfv (GL_LIGHT0, GL_AMBIENT,  luz_ambiente);
   glLightfv (GL_LIGHT0, GL_DIFFUSE,  luz_difusa);
   glLightfv (GL_LIGHT0, GL_SPECULAR, luz_especular);
   glLightfv (GL_LIGHT0, GL_POSITION, luz_posicao);

   // materiais
```

```
   glMaterialfv (GL_FRONT, GL_SPECULAR,  mat_especular);
   glMaterialf  (GL_FRONT, GL_SHININESS, mat_brilho);
}

// configura o campo de visualização
void confCamera ()
{
   // ...
   gluLookAt (3,2,3, 0,0,0, 0,1,0);
}

// função responsável por eventos de teclado
void keyboard (unsigned char key, int posX, int posY)
{
   switch (key) {
      // saindo do programa
      case 27 :
         exit (0);
         break;
      // configurando o brilho
      case 'B' :
         if (mat_brilho < 128.0)
            mat_brilho += 1.0;
         break;
      case 'b' :
         if (mat_brilho > 0.0)
            mat_brilho -= 1.0;
         break;
      // posicionamento da fonte de luz
      case 'X' :
         luz_posicao[0] += SPEED;
         break;
      case 'x' :
         luz_posicao[0] -= SPEED;
         break;
      case 'Y' :
         luz_posicao[1] += SPEED;
         break;
      case 'y' :
         luz_posicao[1] -= SPEED;
         break;
      case 'Z' :
         luz_posicao[2] += SPEED;
         break;
      case 'z' :
         luz_posicao[2] -= SPEED;
         break;
   }
   // repinta a tela
   glutPostRedisplay ();
}

// função de desenho
void draw ()
{
   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

   confCamera ();
   confLuzMat ();

   // ...
```

```
   glutSolidTeapot (0.5);

   // desenhando a "luz"
   glColor3f (1.0, 1.0, 0.0);
   glTranslatef (luz_posicao[0], luz_posicao[1], luz_posicao[2]);
   glutSolidSphere (0.1, 20, 20);

   glFlush ();
   glutSwapBuffers ();
}

// função para inicializar as variaveis de estado
void init ()
{
   // ...
   // habilitando a iluminação e cor dos materiais
   glEnable (GL_COLOR_MATERIAL);
   glEnable (GL_NORMALIZE);
   glEnable (GL_LIGHTING);
   glEnable (GL_LIGHT0);
   confLuzMat ();
}
```

MANIPULANDO AS MATRIZES DE TRANSFORMAÇÃO

Quanto às transformações geométricas sabemos que são acumulativas, pois ao fizer uma chamada como glRotatef (10, 0, 1, 0) corresponderia a um giro de 10 graus no eixo Y, caso fizéssemos uma chamada glRotatef (10, 0, 1, 0) em seguida, corresponderia também a um giro de 10 graus no eixo Y, no entanto, este giro em relação ao estado inicial da máquina faria um giro total de 20 graus.

O problema é que nem sempre queremos as transformações em modo acumulativo, em nosso exemplo citado logo a cima, o giro total depende plenamente da primeira transformação para que com a segunda tenhamos um giro com a quantidade de graus desejados.

Como faríamos para ter transformações independentes das outras?

Uma solução seria fazer a transformação desejada e após desenhar o objeto, fazer uma transformação inversa à anterior com os mesmos valores, no entanto, com sinais contrários, por exemplo:

```
// ...
glRotatef (10, 0, 1, 0);
// desenho do objeto ....
glRotatef (-10, 0, 1, 0);
// transformação de 10 graus do eixo Y desfeita!
// ...
```

Apesar de ser uma solução, o código não fica elegante, e utilizar uma transformação apenas para desfazer outra o código acaba ficando "sujo".

OpenGL possui duas funções que permitem salvar e restaurar o estado das transformações, o que resolve nosso problema! Estas funções são: glPushMatrix e glPopMatrix. Utilizadas para Salvar e Restaurar, respectivamente e aplicando ao exemplo anterior teríamos um código muito mais limpo:

```
// ...
glPushMatrix ();
   glRotatef (10, 0, 1, 0);
   // desenho do objeto ....
glPopMatrix ();
// ... transformação de 10 graus do eixo Y desfeita!
// ...
```

Estes comandos servem apenas para as transformações geométricas default do OpenGL: glRotatef, glTranslatef e glScalef.

Tendo por base o exemplo anterior (Exemplo 10), iremos aplicar um pouco deste conceito de salvar e restaurar a matriz de transformação:

Exemplo 11:

```
// ...
float mat_brilho      = 60.0;

// guarda o giro dos objetos
float angulo = 0.0;

// guarda a qtd de lados dos objetos
int lados = 10;

// configura o campo de visualização
void confCamera ()
{
   // ...
   gluLookAt (1,1,5, 0,0,0, 0,1,0);
}

// função responsável por eventos de teclado
```

```
void keyboard (unsigned char key, int posX, int posY)
{
   switch (key) {
      // saindo do programa
      case 27 :
         exit (0);
         break;
      // configurando o giro
      case 'R' :
         angulo += 1.0;
         break;
      case 'r' :
         angulo -= 1.0;
         break;
      // configurando a qtd de lados dos objetos
      case 'L' :
         if (lados < 100)
            lados++;
         break;
      case 'l' :
         if (lados > 3)
            lados--;
         break;
      // configurando o brilho
      // ...
   }
}

// função de desenho
void draw ()
{
   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

   confCamera ();
   confLuzMat ();

   // desenhando o plano cartesiano 3D
   glBegin (GL_LINES);
      // ...
   glEnd ();

   // salvando o estado atual (inicial)
   glPushMatrix ();
      glColor3f (1.0, 0.0, 0.0);
      glTranslatef (-1.5, 0, 0);
      glRotatef (-90, 1, 0, 0);
      glRotatef (angulo, 0, 0, 1);
      glutSolidCone (0.5, 1.0, lados, lados);
   glPopMatrix (); // estado inical recuperado

   // salvando o estado atual (inicial)
   glPushMatrix ();
      glColor3f (0.0, 0.0, 1.0);
      glRotatef (-90, 0, 1, 0);
      glRotatef (angulo, 1, 1, 1);
      glutSolidTorus (0.1, 0.5, 90, lados);
   glPopMatrix (); // estado inicial recuperado

   // salvando o estado atual (inicial)
   glPushMatrix ();
      glColor3f (0.0, 1.0, 0.0);
```

```
      glTranslatef (1.5, 0, 0);
      glRotatef (angulo, 0, 1, 0);
      glutSolidSphere (0.5, lados, lados);
   glPopMatrix (); // estado inicial recuperado

   // desenhando a "luz"
   glColor3f (1.0, 1.0, 0.0);
   glTranslatef (luz_posicao[0], luz_posicao[1], luz_posicao[2]);
   glutSolidSphere (0.1, 20, 20);

   glFlush ();
   glutSwapBuffers ();
}

// função principal
int main ()
{
   // configurando e exibindo uma janela
   glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGBA);
   glutCreateWindow ("Exemplo 11: matrizes de transformação");
   glutFullScreen ();

   // registrando função de callback
   // ...
}
```

Analisando o código:

As variáveis declaradas como angulo e lados servem para guarda a giro dos objetos e o número de lados de cada um, respectivamente;

Alteramos os valores da função gluLookAt apenas para visualizar a cena melhor pois, temos que visualizar três objetos e não mais um, ao mesmo tempo;

Na função main temos o comando glutFullScreen que coloca a janela em tela cheia, muito útil em diversas aplicações;

Note que a primeira chamada à função glPushMatrix na função draw é para desenhar o objeto cone e antes dela não há nenhuma transformação geométrica presente, logo, será salvo uma matriz sem nenhuma transformação. Logo após, é posta uma translação e duas rotações sucessivas e então desenha-se o objeto cone e restaura a matriz salva. Antes de desenhar o objeto seguinte, salvamos a matriz novamente, note que não é utilizado nenhuma transformação de translação pois, quando recuperamos a matriz anteriormente o sistema de

coordenadas voltou ao centro de coordenada então podemos simplesmente desenhar o objeto que ele ficará no centro do plano cartesiano 3D.

Se não estivéssemos utilizado o primeiro glPushMatrix e glPopMatrix tínhamos por resultado o objeto torus sobre o cone, pois não foi aplicada nenhuma translação ao desenhar o torus. Importante lembrar o conceito da máquina de estado, pois, com essa simples alteração, todos os objetos seguintes sofrerão a conseqüência, logo, o objeto sphere assumirá o centro do plano e a "luz" não mais corresponderá à fonte de luz real.

A saída desta idéia ficaria mais ou menos desta forma:

BIBLIOGRAFIA

- The OpenGL Utility Toolkit (GLUT) Programming Interface API, Version 3
  http://www.opengl.org/resources/libraries/glut/spec3/spec3.html
- OpenGL Reference Manual
  http://www.glprogramming.com/blue/
- OpenGL Programming Guide (The Red Book)
  http://www.glprogramming.com/red/