Universidade Federal de Santa Maria
Departamento de Computação Aplicada
Prof. Cesar Tadeu Pozzer
Disciplina: Computação Gráfica
pozzer@inf.ufsm.br
26/01/2021

OpenGL 1.X – Conceitos Básicos

1 Histórico

- Antes do surgimento de **APIs** gráficas
  - Eficiência: Programação em C + *Assembly*
  - Uso exclusivo da CPU
- APIs gráficas 3D antigas
  - PHIGS (*Programmer's Hierarchical Interactive Graphics System*) [1,2]
  - GKS (*Graphical Kernel System*) [3].
- APIs gráficas atuais
  - OpenGL
  - Direct3D
  - Comparativo entre APIs

Versão 1.0 (1992)
Versão 1.1 (1997) – Vertex Array
Versão 1.2 (1998) – Textura 3D
Versão 1.3 (2001) – Multi-textura, etc.
Versão 1.4 (2002) – Mipmap automático, etc.
Versão 1.5 (2003) – Vertex buffer, etc.
Versão 2.0 (2004) – Inclusão da Linguagem de shading
Versão 3.0 (2008) – **112 funções tornaram-se *deprecated*, restando 126 funções.** A programação por shader torna-se obrigatória (http://www.khronos.org/opengl/).
Versão 3.1 (2009) – **As funções deprecadas foram removidas da API.**
Versão 3.2 (2009) – Introduz os contextos *core* e de compatibilidade. Este último para incluir as funções deprecadas, enquanto o contexto *core* trabalha apenas com as novas funcionalidades.
Versão 3.3 (2010) – Lançado simultaneamente com a versão 4.0. Traz recursos avançados para as GPUs antigas. Mas não inclui os novos estágios do *pipeline* programável.
Versão 4.0 (2010) – Novos estágios de pipeline programável. *Tesselation Control* e *Tesselation Evaluation Shader*.
Versão 4.3 (2012) – Inclui o *compute shader*.
Versão 4.5 (2014) – Inclui Direct State Access.

Todas as versões do OpenGL são *backward compatible, até a versão 3.0*.

2 Características da API OpenGL

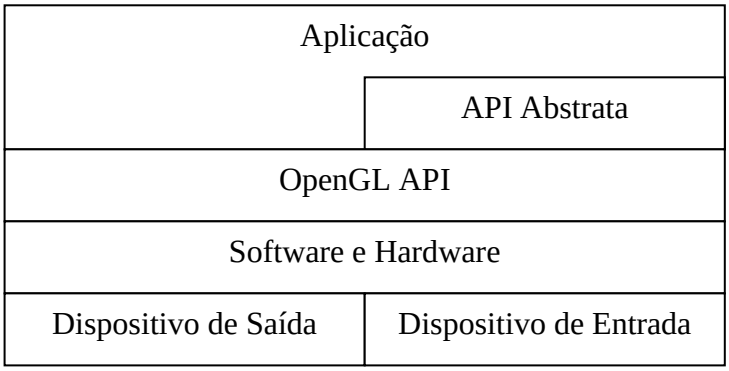
- É uma interface de software (API) com o hardware gráfico (Ver arquivo opengl.h) escrita em linguagem C. **Na versão 3.0 da API, 112 funções tornaram-se *deprecated*, restando 126 funções ativas. A lista das funções pode ser encontrada em: http://www.opengl.org/sdk/docs/man/xhtml/ (OpenGL Version 2.1) ou http://pyopengl.sourceforge.net/documentation/manual-3.0/index.xhtml (OpenGL version 3.0.1).** Veja http://pyopengl.sourceforge.net/documentation/deprecations.html para mais detalhes.

Mesmo com a remoção das funções, ainda é valido estudar a API antiga, pois facilita o entendimento da nova versão do OpenGL. Este documento não aborda versões atuais do OpenGL.

- Baseado em máquina de estados
  - `glClearColor()`
  - `glMatrixMode()`
  - `glEnable()/glDisable()`
  - `glPolygonMode()`
  - `glColor3f()`
- Multi-plataforma;
- Arquitetura bem definida;
- Não oferece comandos para gerenciamento de janelas, nem comandos para geração de modelos 3D complexos. Para definição de janelas, pode-se utilizar funções da biblioteca auxiliar GLUT (*GL Utility Toolkit*);
- Conjunto de primitivas geométricas simples;
  - Básicas: ponto, linha, superfície poligonal (triângulo, quadrado, polígono convexo)
  - Objetos 3D: Superfícies Bézier, quádricas, esferas, poliedros.

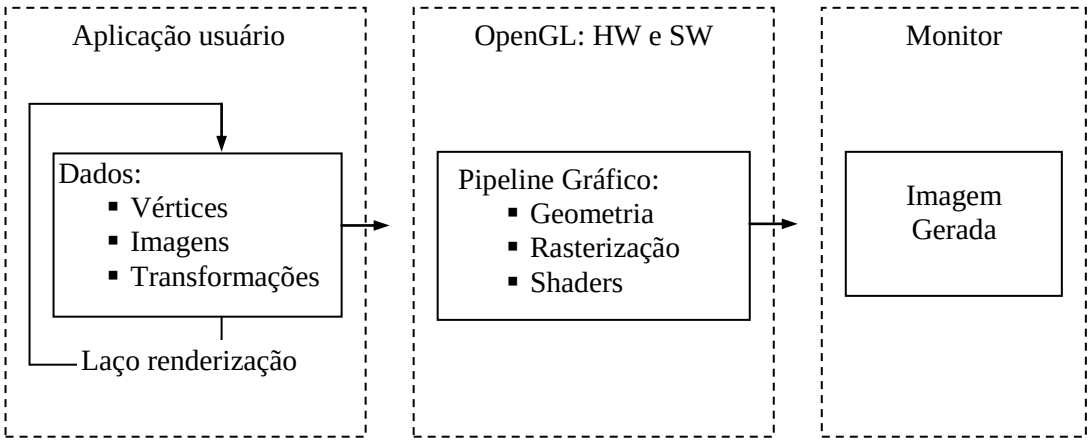
3 Arquitetura da API OpenGL

Uma aplicação desenvolvida em OpenGL pode acessar a API de forma direta ou indireta, como mostrado no seguinte diagrama. Pode haver uma API abstrata sobre o OpenGL que pode tornar a programação em mais alto nível. Pode-se citar como exemplos o Java3D, GLUT, dentre outros. A aplicação do usuário não pode conversar diretamente com o hardware da placa gráfica.



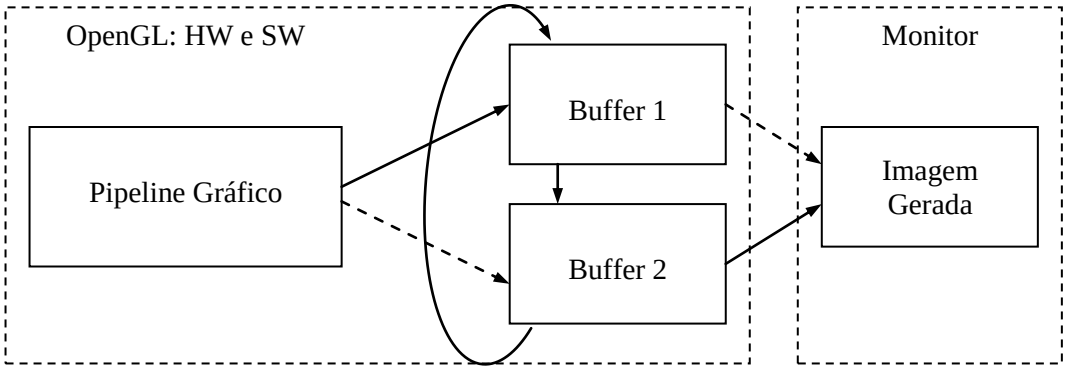
Caso um computador não tiver placa gráfica, ou tiver uma placa gráfica antiga que não oferece muitos recursos, cabe ao driver OpenGL realizar as funcionalidades não oferecidas. Obviamente o driver executa na CPU e isso comprometerá o desempenho da aplicação como um todo.

Um programa OpenGL consiste de um laço infinito onde a cada passo a aplicação, geralmente escrita em linguagem C, passa para a API um conjunto de dados que devem ser processados. Esses dados mudam à medida que o usuário interage com a aplicação.



- Programação em OpenGL x programação em C

Antes dos dados serem exibidos, eles são armazenados em *buffers*. O OpenGL permite a especificação de 1 ou 2 conjuntos de buffers (***Single*** ou ***Double Buffer***). O monitor lê os dados do buffer corrente no momento. O uso de dois *buffers* se faz necessário para evitar que as animações fiquem cortadas ao meio, visto que o monitor pode exibir uma cena animada que está sendo gerada. Usando buffer duplo, quando um buffer está concluído, o OpenGL passas a escrever no outro, liberando o pronto para ser exibido (ver comando `glutSwapBuffers()`).



4 Instalação do OpenGL e Bibliotecas Auxiliares

O OpenGL é disponibilizado via uma API. Dois arquivos definem esta API: gl.h e glu.h. No arquivo gl.h estão definidas as funções incorporadas na API e no arquivo glu.h funções auxiliares (*GL Utility*) do OpenGL. As funções mias usuais da GLU incluem definição de matriz de projeção ortográfica 2D (**`glu`**`Ortho2D()`), função para especificação da posição da câmera (**`glu`**`LookAt()`, **`glu`**`Perspective()`) e funções para manipulação de curvas do tipo NURBS.

Pelo fato do OpenGL ser independente do sistema de janelas, existe uma outra API chamada **GLUT** - *GL Utility Toolkit* (glut.h) que oferece recursos de manipulação de janelas, popup menu, tratamento de eventos de mouse e teclado e algumas primitivas gráficas 3D. Por ser uma API muito difundida, é distribuída juntamente com o OpenGL. A próxima seção oferece mais detalhes dos sistemas de janelas.

Existem outras alternativas para o uso da GLUT. Pode-se também fazer uso da **Freeglut**, cuja interface é muito semelhante (http://freeglut.sourceforge.net/), e é opensource. Outra opção é o uso da própria **API do Windows** para fazer o gerenciamento de janelas e tratamento de eventos. Pode-se também utilizar a API **GLFW** (http://www.glfw.org/), que será apresentada na próxima seção. Outra alternativa é o uso da lib **SDL** (http://www.libsdl.org/). Também oferece recursos para mídias de áudio e vídeo.

Para configurar o OpenGL é necessário que os seguintes arquivos (.h, .lib e .dll) estejam nas seguintes localizações.

Arquivo	**Localização**
gl.h glut.h glu.h	[compiler]\include\GL
opengl32.lib glut32.lib glu32.lib	[compiler]\lib
opengl32.dll glut32.dll glu32.dll	[system]

Todos estes arquivos podem ser colocados juntamente com o projeto. Neste caso deve-se alterar os parâmetros do projeto. Para o Dev-C++, o arquivo opengl32.lib, por exemplo, é chamado libopengl.a. A sintaxe do makefile é semelhante à forma como as libs são especificadas no projeto do Dev-C++.

```
g++ -O0 -g3 -Wall -c *.cpp
g++ -oapp meu_programa.o -L"./" -lglut -lGLU -lGL -lm
```

Para mais detalhes de configuração de parâmetros nos compiladores DEV-C++ e Microsoft .NET, consulte a documentação da API Canvas2D.

4.1 GLFW

A GLFW (http://www.glfw.org/) é ama API aberta para a criação de aplicações baseadas em OpenGL. A maior diferença em relação à Glut é que esta deixa o controle do loop de renderização no código do usuário. Pode-se também fazer uma aplicação sem o uso de callbacks. A distribuição possui uma versão da lib que não exige biblioteca dinâmica (dll) para execução.

Um programa feito com a GLFW tem a seguinte estrutura básica:

```
#include <GL/glfw.h>

int main()
{
	glfwInit();
	glfwOpenWindowHint(GLFW_WINDOW_NO_RESIZE, GL_TRUE);
	glfwOpenWindow(800, 600, 8, 8, 8, 8, 24, 8, GLFW_WINDOW);

	bool running = true;

	while (running)
	{
		// trata eventos

		// Exibe algo na tela

		glfwSwapBuffers();

		running = glfwGetWindowParam(GLFW_OPENED) != 0;
	}
	glfwTerminate();
}
```

Para maiores detalhes, veja demo no site da disciplina ou consulte seu manual.

4.2 GLUT e Sistemas de Janelas

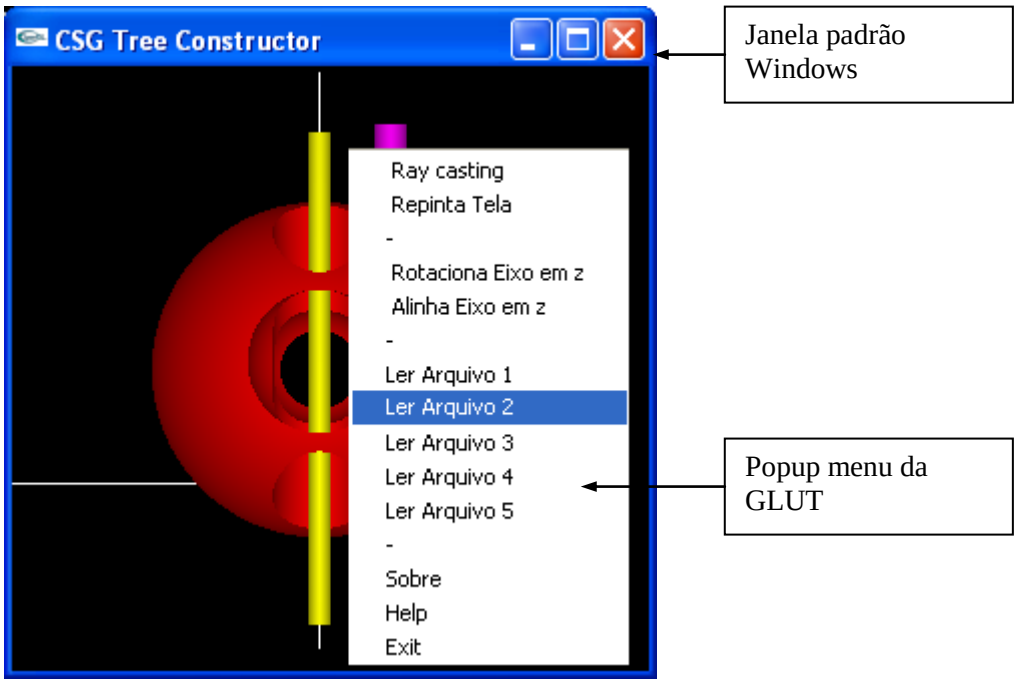
O OpenGL foi projetado para ser independente do sistema de janelas e sistema operacional. Deste modo, não possui comandos para abertura de janelas e leitura de eventos do teclado e mouse. A biblioteca mais difundida para gerenciamento de janelas em OpenGL é a GLUT. Ela é muito limitada quando comparada com outras APIs mas é simples de usar e oferece recursos para construção de aplicativos com interface simples, o que a torna uma atrativa ferramenta de auxílio ao aprendizado do OpenGL.

A GLUT também possui diversas rotinas para especificação de objetos 3D como esfera, torus e o famoso bule de chá chamado *Utha Teapot*.

Funções para gerenciamento de Janelas

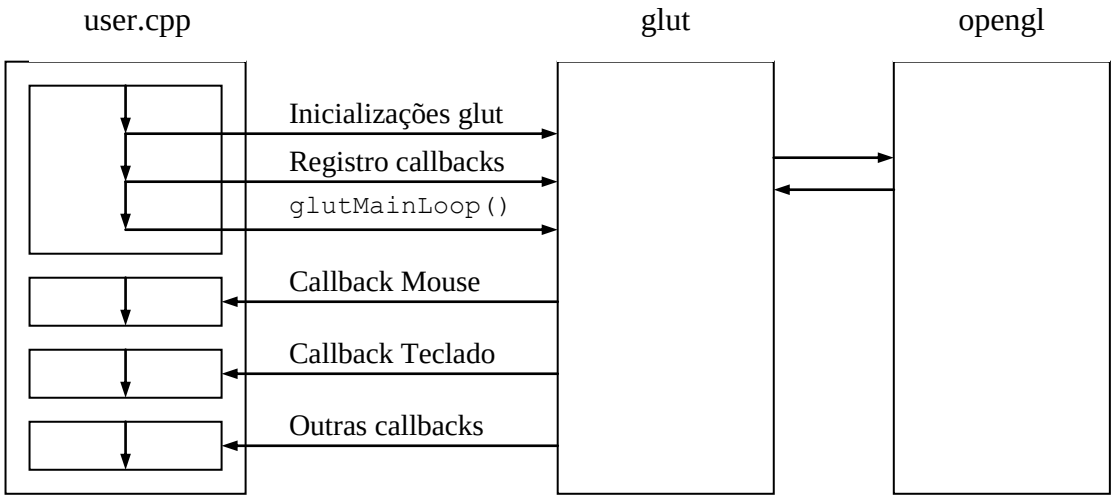
Existem 5 rotinas principais para inicialização de uma janela:

- **`void glutInit(int *argcp, char **argv)`**: Inicializa a GLUT.
- **`void glutInitDisplayMode(unsigned int mode)`**: Especifica o número de buffers a serem utilizados e parâmetros dos buffers: `GLUT_SINGLE`, `GLUT_DOUBLE`, `GLUT_DEPTH`, `GLUT_STENCIL`, `GLUT_ACCUM`).
- **`void glutInitWindowPosition(int x, int y)`**: Especifica a posição do canto esquerdo superior da janela.
- **`void glutInitWindowSize(int width, int height)`**: Especifica o tamanho da janela em pixels.
- **`int glutCreateWindow(const char *title)`**: Cria uma janela com um contexto OpenGL.



Funções de Callback

As funções de *callbacks* são utilizadas para registrar na GLUT funções definidas pelo usuário (ponteiros para função) que vão tratar os eventos da aplicação. Isso é necessário porque a GLUT assume o controle da aplicação tão logo seja chamada a função `glutMainLoop()`, podendo a aplicação do usuário intervir somente quando estes eventos ocorrerem. Os principais comandos são listados a seguir. Para uma lista completa de funções, veja [4, 5, 6].



- **`void glutDisplayFunc(void (*func)(void))`**: Callback mais importante. Usada para especificação da função de desenho. Chamada sempre que o contexto da janela precisa ser reexibido. Todas as rotinas usadas para redesenho da cena devem estar nesta *callback* de display (ver `glutIdleFunc()`).

- **`void glutReshapeFunc(void (*func)(int width, int height))`**: Registra a função para tratar o redimensionamento da janela.

- **`void glutKeyboardFunc(void (*func)(unsigned char key, int x, int y))`** e **`void glutMouseFunc(void (*func)(int button, int state, int x, int y))`**: funções para tratamento de teclado ou botão de mouse.

- **`void glutMotionFunc(void (*func)(int x, int y))`**: trata movimento do mouse enquanto algum botão estiver pressionado.

- **`void glutPassiveMotionFunc(void (*func)(int x, int y))`**: trata movimento do mouse.

- **`void glutKeyboardUpFunc(void (*func)(unsigned char k, int x, int y))`**: tratamento de liberação de teclas pressionadas.

- **`void glutIdleFunc(void (*func)(void))`**: função chamada sempre que não existem evento a serem tratados. Pode ser usada para processamento de tarefas em background ou de animações, por exemplo.

Uma aplicação desenvolvida com a GLUT tem a seguinte estrutura:

```
#include <gl/glut.h>

void init()
{
   //comandos de inicialização do opengl (definição dos estados).
}

void render()
{
   //todos os dados e parâmetros a serem passados ao OpenGL para renderizar cada quadro da
cena devem ser colocados nesta função que está associada à callback display e a callback
idle.
}

void OnReshape(int w, int w)
{
   //ação a ser realizada quando a janela é redimensionada
}

void OnKeyPress()
{
   //ação a ser realizada quando uma tecla é pressionada
}

int main(int argc, char** argv) {

       glutInit(&argc, argv);
       glutInitDisplayMode(GLUT_DEPTH | GLUT_RGBA | GLUT_DOUBLE);
       glutInitWindowSize(640,480);
       glutCreateWindow("Titulo da janela");

       glutDisplayFunc(render);
```

```
        glutIdleDisplayFunc(render);

        glutReshapeFunc(OnReshape);
        glutKeyboardFunc(OnKeyPress);
        OnInit(); //inicializacoes customizadas

        glutMainLoop();
        return 0;
}
```

Na *callbak* `onReshape()` deve-se especificar ações a serem tomadas quando a janela for redimensionada. Nesta situação, os comandos mais comuns a serem utilizados são o de viewport e projeção

A função `render()` é a responsável pela renderização da cena. Antes de gerar cada cena, geralmente deve-se limpar os dados gerados na cena anterior. Para isso existe o comando `glClear()`, que aceita como parâmetros os mesmos buffers que podem ser definidos na inicialização da GLUT pela execução do comando `glutInitDisplayMode()`: `GL_COLOR_BUFFER_BIT`, `GL_DEPTH_BUFFER_BIT`, `GL_ACCUM_BUFFER_BIT` e `GL_STENCIL_BUFFER_BIT`.

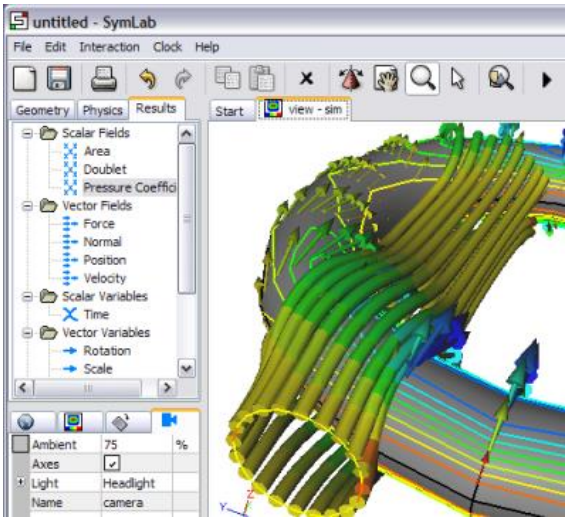
```
void glClear(GLbitfield mask);
void glClearColor(GLclampf r, GLclampf g, GLclampf b, GLclampf a);
```

Ao se utilizar o comando `glClear(GL_COLOR_BUFFER_BIT)`, utiliza-se a cor de limpeza definida pelo comando `glClearColor()`, usualmente chamado na função `OnInit()`.

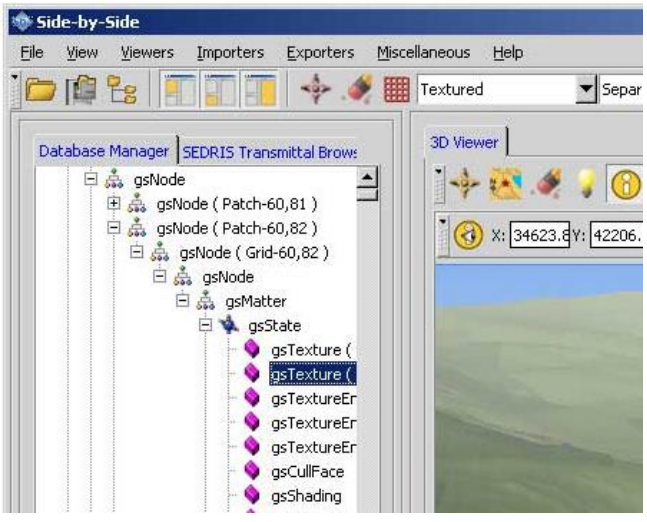
```
void render()
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   renderScene();
   glutSwapBuffers();
}
```

4.3 Outros Sistemas de Janelas

Além da GLUT, existem outras APIs independentes de sistema operacional que oferecem diversos recursos mais avançados para geração de interfaces gráficas. São exemplos as APIs QT [7], GTK [8], wxWidget [9], Fox [10], dentre outras.



wxWidget

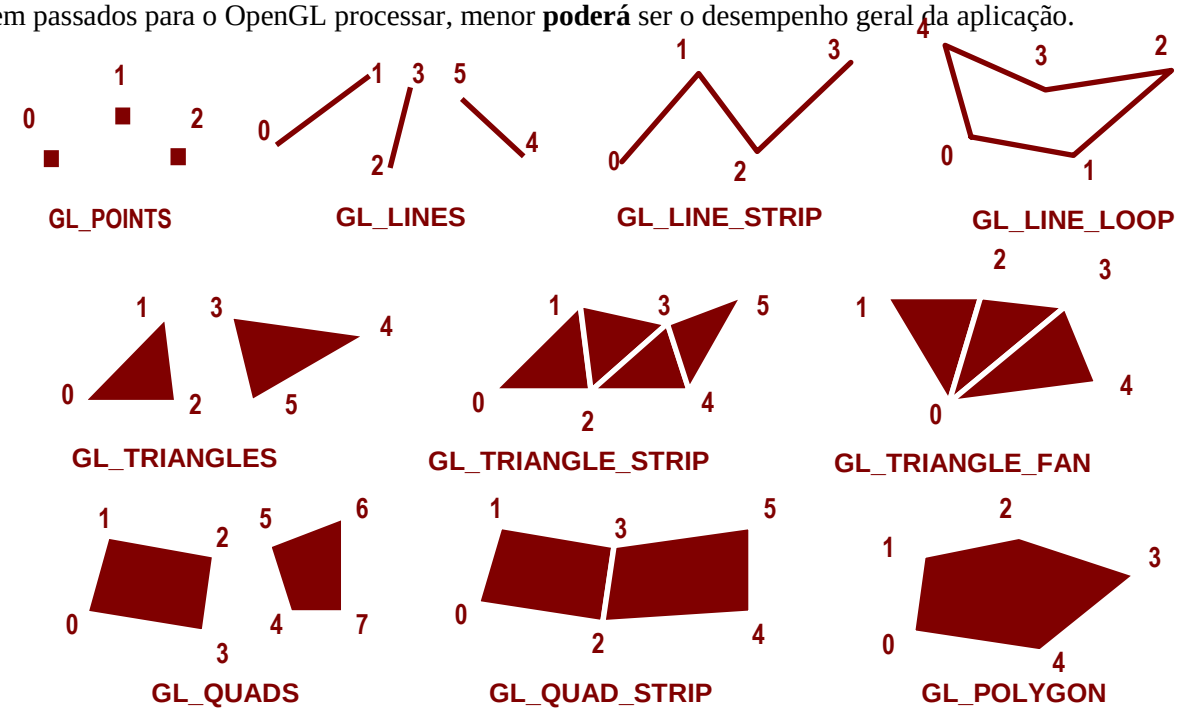


Fox

5 Primitivas Geométricas

O OpenGL oferece uma série de primitivas geométricas para definição dos objetos gráficos. Basicamente existem 3 tipos de primitivas: pontos, linhas ou superfícies (triângulos, quadriláteros e polígonos convexos). A escolha de qual primitiva utilizar vai depender da aplicação.

Deve-se observar que a mesma geometria pode ser definida por vários tipos de primitivas, porém o número de vértices utilizados pode ser diferente. Para definição de malhas poligonais triangulares deve-se preferencialmente utilizar `GL_TRIANGLE _STRIP` ou `GL_TRIANGLE_FAN` ao invés de `GL_TRIANGLES`. Com o uso do comando `GL_TRIANGLE_FAN` pode-se definir por exemplo 3 triângulos com apenas 5 vértices. Utilizando-se `GL_TRIANGLE` seriam necessários 9 vértices. Quanto mais vértices forem passados para o OpenGL processar, menor **poderá** ser o desempenho geral da aplicação.



Primitivas geométricas OpenGL. Extraída de [12]

Para a especificação de vértices, deve-se utilizar o comando do OpenGL **`glVertex*()`**. Este comando, como diversos outros (**`glColor*()`**, **`glNormal*()`**, **`glFog*()`**, **`glLight*()`**, **`glMaterial*()`**, **`glRotate*()`**, **`glTranslate*()`**, **`glScale*()`**), aceitam até 8 diferentes tipos de dados como argumentos. As letras usadas como sufixo para especificar os tipos de dados são mostradas na seguinte tabela.

Deve-se observar que todos os comandos OpenGL iniciam com o prefixo **`gl`** e que todas as constantes iniciam com o prefixo **`GL_`**. A sintaxe do comando **`glVertex*()`** é a seguinte:

```
void glVertex{234}{sifd}[v]{TYPE coords};
```

Este comando especifica um vértice que descreve um objeto geométrico. Suporta até 4 coordenadas (*x*, *y*, *z*, *w*) e no mínimo 2. O sufixo *v* é opcional e é usado para indicar que o argumento é um vetor de dados. Se for usada uma versão que não explicita *z* ou *w*, por default *z=0* e *w=1*.

Sufixo	Tipo de dado	Correspondência com C	Tipo OpenGL
b	Inteiro 8 bits	signed char	`GLbyte`
s	Inteiro 16 bits	short	`GLshort`
i	Inteiro 32 bits	int ou long	`GLint, GLsizei`
f	Float 32 bits	float	`GLfloat, GLclampf`

d	Float 64 bits	double	`GLdouble, GLclampd`
ub	Unsigned 8 bits	unsigned char	`GLubyte, GLboolean`
us	Unsigned 16 bits	unsigned short	`GLushort`
ui	Unsigned 32 bits	unsigned int ou long	`GLuint`

Exemplos:

```
GLdouble pos[] = {0.1, 10, 33.123};
glVertex3dv(pos);
glVertex2d(10, 22.5);
glVertex3d(0.1, 10, 33.123);
glVertex2i(10, 22);
```

Este comando deve ser chamado dentro do par de comandos `glBegin()` e `glEnd()`.

```
void glBegin(GLenum mode);
```

Marca o início de uma lista de vértices que descrevem uma primitiva geométrica. O parâmetro *mode* pode ser uma das 10 primitivas geométricas descritas anteriormente (`GL_POINTS`, `GL_LINES`, ...).

```
void glEnd();
```

Marca o fim de uma lista de vértices.

Dentro do bloco `glBegin() – glEnd()` pode-se também especificar atributos dos vértices, com cor, normal, textura, dentre outros. Os principais comandos são: **`glVertex*()`**, **`glColor*()`**, **`glIndex*()`**, **`glNormal*()`**, **`glTextCoord*()`** e **`glMaterial*()`**.

```
glBegin(tipo_de_prim);
   //define atributos de vértice: cor, normal, textura,...
   //define o vértice
glEnd();

glBegin(GL_TRIANGLE);
    glVertex3d(0.1, 10, 33.123);
    glVertex3d(0.1, 10, 23.123);
    glVertex3d(0.1, 10, 13.123);
glEnd();

glBegin(GL_TRIANGLE);
    for(int i=0; i<30; i++)
       glVertex3dv(data[i]);
glEnd();
```

Considerações:

- Esta forma de especificação dos vértices não impõe uma estrutura de dados. Deve-se chamar uma função para especificar cada vértice
- Insuficiente para primitivas complexas
- Não permite compartilhamento de vértices.

5.1 Pontos e Linhas

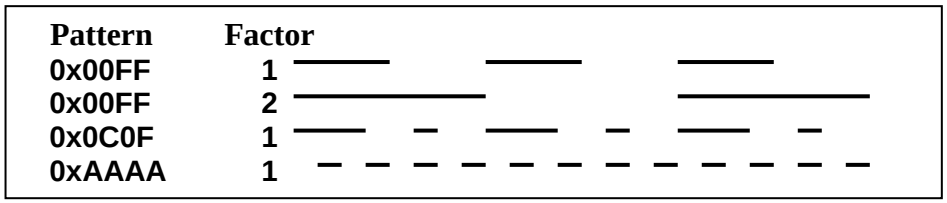
Pode-se também especificar **atributos** às primitivas geométricas, por meio dos comandos

- **`void glPointSize(GLfloat size)`**: determina a largura em pixels do ponto. O tamanho deve ser maior que zero e o default é 1. Os pixels a serem desenhados dependem se o *antialiasing* estiver

habilitado. Se o *antialiasing* estiver desabilitado (default - `glEnable(GL_POINT_SMOOTH)`, larguras fracionárias serão arredondadas para larguras inteiras e será gerado um quadrado de pixels. Com o *antialiasing* ativo, um grupo circular de pixels será gerado, sendo os pixels na borda geralmente tendo intensidades mais fracas.

- **`void glLineWidth(GLfloat width)`**: Determina a largura em pixels de uma linha. O tamanho deve ser maior que zero e o default é 1. Pode-se também gerar linhas com antialiasing (`glEnable(GL_LINE_SMOOTH)`).

- **`void glLineStipple(GLint factor, GLushort pattern)`**: Permite a geração de linhas com padrões configuráveis. O argumento padrão é um conjunto de 16 bits 0's e 1's. O parâmetro `factor` é usado para esticar o padrão gerado. Deve-se habilitar esta característica com o comando `glEnable(GL_LINE_STIPPLE)`.



5.2 Polígonos

Polígonos podem ser desenhados de forma preenchida, apenas com as linhas e contorno (bordas do polígono) ou somente os vértices.

Um polígono tem dois lados (*front* e *back*), que podem ser renderizados diferentemente dependendo do lado que o observador estiver observando. Isso pode ser útil para diferenciar partes que estão dentro ou fora de objetos. O comando `glPoligonMode()` é usado para determinar a forma de desenho de polígonos.

```
void glPolygonMode(GLenum face, GLenum mode);
```

O parâmetro *face* pode assumir `GL_FRONT_AND_BACK`, `GL_FRONT`, ou `GL_BACK`. *mode* pode ser `GL_POINT`, `GL_LINE`, ou `GL_FILL` para indicar se o polígono deve ser desenhado por pontos, linhas ou de forma preenchida (default).

```
glPolygonMode(GL_FRONT, GL_FILL);
glPolygonMode(GL_BACK, GL_LINE);
```

Por conversão, polígonos cujos vértices aparecem em ordem anti-horário na tela são chamados *front-facing*. Pode-se mudar esta convenção com o comando `glFrontFace()` caso os vértices dos objetos forem definidos em ordem horária. Os argumentos podem ser `GL_CCW` *(default) ou* `GL_CW`.

```
void glCullFace(GLenum mode);
void glFrontFace(GLenum mode);
```

Ao se construir um objeto fechado composto por polígonos opacos com uma orientação consistente, nenhum polígono *back-facing* será visível. Neste e em outros casos, pode-se otimizar a renderização removendo-se as faces cuja face da frente não seja visível. O comando `glCullFace()` diz ao OpenGL para descartar polígonos *front-* ou *back-facing*. Os argumentos são `GL_FRONT`, `GL_BACK`, ou `GL_FRONT_AND_BACK`. Deve-se habilitar o recurso de *culling* com o comando `glEnable(GL_CULL_FACE)`.

5.3 Vetores Normais

O OpenGL permite a especificação de normais por faces (polígono) ou por vértices. O comando `glNormal3*()` é usado para a especificação de uma normal. O vetor normal é utilizado para cálculos de **iluminação**.

void glNormal3{bsidf}(**TYPE** nx, **TYPE** ny, **TYPE** nz);
void glNormal3{bsidf}**v**(**const TYPE** *v);

Uma vez setado um valor de normal, este permanece o valor corrente da normal até que outro comando de normal seja chamado. No seguinte exemplo define-se uma normal para cada vértice do polígono. Caso fosse definida somente a normal n_0, todos os vértices $v_0...v_3$ compartilhariam a mesma normal.

```
glBegin (GL_POLYGON);
   glNormal3fv(n0);
   glVertex3fv(v0);
   glNormal3fv(n1);
   glVertex3fv(v1);
   glNormal3fv(n2);
   glVertex3fv(v2);
   glNormal3fv(n3);
   glVertex3fv(v3);
glEnd();
```

Deve-se observar que qualquer superfície possui duas normais, apontando em direções opostas.

- Por convenção, a **normal é o vetor que aponta para fora do objeto** sendo modelado.
- A norma dos vetores normais passados ao OpenGL é irrelevante, porém deve-se observar que antes de aplicar processos de iluminação as normais devem ser normalizadas.
    - De preferência, deve-se passar vetores normalizados.
    - Estes vetores permanecem normalizados exceto se forem aplicados transformações de escala à *modelview*.
    - Transformações de translação e rotação não alteram as normais.
    - Em caso de **transformações não uniformes** como escalas ou especificação de normais não normalizadas, pode-se requisitar que o OpenGL automaticamente as normalize após as transformações. Para isso, deve-se habilitar a normalização com o comando `glEnable(GL_NORMALIZE)`. Esse recurso esta desabilitado por default, visto que requer cálculos adicionais que podem reduzir o desempenho da aplicação.
- As normais são sempre aplicadas sobre **vértices**. Por isso, ao se usar strips (por exemplo GL_QUAD_STRIP), deve-se ter em mente que não se pode aplicar a normal por face, o que proíbe a geração de objetos fechados como cubos, onde deve existir somente uma normal por face.

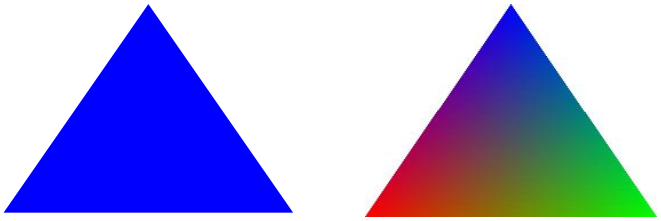
6 Modelos de Shading

Existem vários modelos de iluminação (ou interpolação de cores) de polígonos:

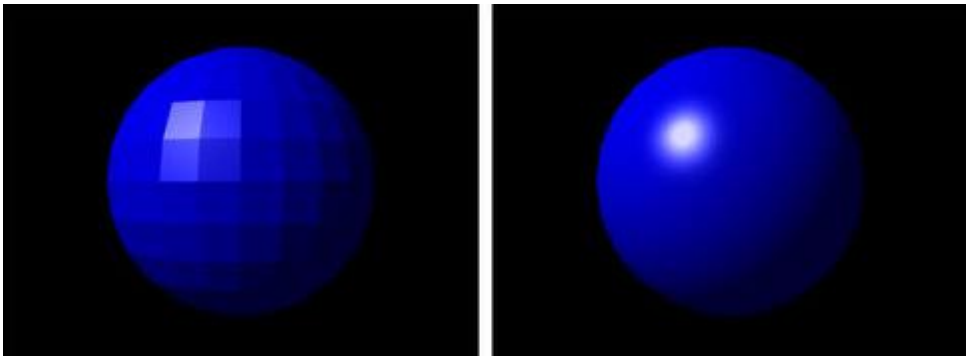
- **Constante**: cálculo de iluminação por polígono. Todo o polígono vai ter a mesma cor, o que em algumas aplicações pode ser desejável, como por exemplo observar a geometria do objeto. É um algoritmo muito eficiente. Esta técnica **é** implementada no OpenGL e pode ser ativada com o comando `glShadeModel(GL_FLAT)`.
- ***Gouraud***: cálculo de iluminação por vértice. Faz interpolação de cor a partir da cor calculada para cada vértice do triângulo. Apesar de rápido, a qualidade depende da resolução da malha. Caso a reflexão especular máxima ocorre no centro de um triângulo grande, ela pode ser descartada complemente, devido a interpolação de cores entre vértices. Para solucionar este problema, deve-se refinar a malha de polígonos. Esta técnica **é** implementada no OpenGL e pode ser ativada com o comando `glShadeModel(GL_SMOOTH)`.

- ***Phong***: cálculo de iluminação por pixel. Aplica-se uma interpolação de normais para cada pixel do triângulo. Tem melhor qualidade, a um custo computacional muito mais alto. Esta técnica **não é** implementada no OpenGL.

As seguintes figuras ilustram a aplicação de dois modelos de shading: `GL_FLAT` (constante) e `GL_SMOOTH` (Gouraud).



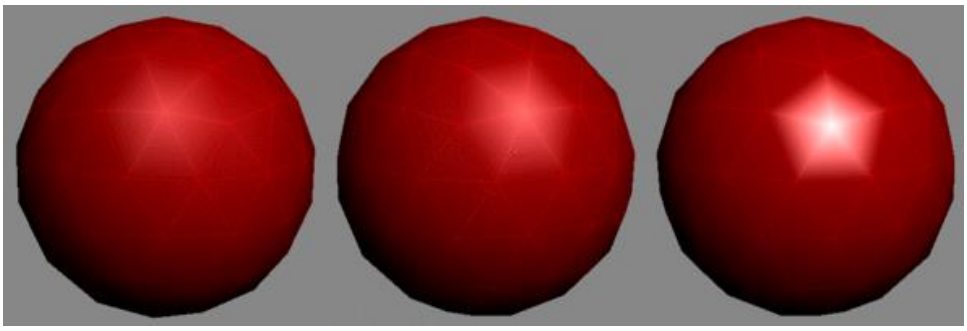
As seguintes figuras ilustram a diferença entre o modelo flat e o modelo *Phong Shading*.



Flat shading ***Phong Shading***

[http://en.wikipedia.org/wiki/Phong_shading]

O algoritmo de *Gouraud*, como mostrado na seguinte figura, faz uma interpolação de cores, gerando resultados diferenciados em função da posição dos triângulos. Esse efeito não ocorre com a técnica *Phong Shading*.



Variação da reflexão especular em função do posicionamento de triângulos.

[http://en.wikipedia.org/wiki/**Gouraud_shading**]

No seguinte exemplo ilustra-se um código OpenGL para geração do triângulo apresentado com variação contínua de cores.

```
glShadeModel(GL_SMOOTH);    // default
glBegin(GL_TRIANGLES);
   glColor3f(1.0,0.0,0.0); // red
   glVertex2f(-1.0,-1.0);
   glColor3f(0.0,1.0,0.0); // green
   glVertex2f(1.0,-1.0);
   glColor3f(0.0,0.0,1.0); // blue
   glVertex2f(0.0,1.0);
glEnd( );
```

Para se gerar o triângulo azul, basta especificar o modelo de *shading* como `GL_FLAT`. Observa-se no segundo exemplo que a especificação de cores por vértice não tem efeito. Leva-se em consideração apenas a última cor especificada.

```
glShadeModel(GL_FLAT);
glColor3f(0.0,0.0,1.0);  // blue
glBegin(GL_TRIANGLES);
   glVertex2f(-1.0,-1.0);
   glVertex2f(1.0,-1.0);
   glVertex2f(0.0,1.0);
glEnd( );
```

```
glShadeModel(GL_FLAT);
glBegin(GL_TRIANGLES);
   glColor3f(1.0,0.0,0.0); // red
   glVertex2f(-1.0,-1.0);
   glColor3f(0.0,1.0,0.0); // green
   glVertex2f(1.0,-1.0);
   glColor3f(0.0,0.0,1.0); // blue
   glVertex2f(0.0,1.0);
glEnd( );
```

Para especificação de cores, deve-se utilizar o comando `glColor*()`. Este comando aceita 3 ou 4 argumentos (RGB e *Alpha*), em coordenadas inteiras ou flutuantes. Caso o parâmetro *Alpha* não for especificado, ele é automaticamente setado para 1.

```
void glColor3{b s i f d ub us ui} (TYPE r, TYPE g, TYPE b);
void glColor4{b s i f d ub us ui} (TYPE r, TYPE g, TYPE b, TYPE a);
void glColor3{b s i f d ub us ui}v (const TYPE*v);
void glColor4{b s i f d ub us ui}v (const TYPE*v);
```

As formas mais comuns de se utilizar este comando são com parâmetros *f* e *ub*, com intervalos entre [0.0, 1.0] e [0, 255], respectivamente. Internamente ao OpenGL, todas as cores são tratadas como valores reais entre [0.0, 1.0].

7 Visualização

O OpenGL oferece três matrizes de transformação: *projection*, *modelview* e *texture* (`GL_MODELVIEW`, `GL_PROJECTION`, ou `GL_TEXTURE`). O link http://www.opengl.org/wiki/Viewing_and_Transformations contém diversos detalhes técnicos muito interessantes sobre esse assunto.

- A **matriz *projection*** (`GL_PROJECTION`) é usada para determinar as transformações de recorte e projeção. A projeção pode ser:
  - projeção ortográfica
  - Projeção em perspectiva
- A **matriz *modelview*** (`GL_MODELVIEW`) é usada para indicar as transformações aplicadas:
  - aos modelos da cena (**transformações de modelos**)
  - posição da câmera (**coordenadas de visualização**). Uma transformação de visualização (câmera) muda a **posição e orientação do *viewpoint*** (ponto de visualização).

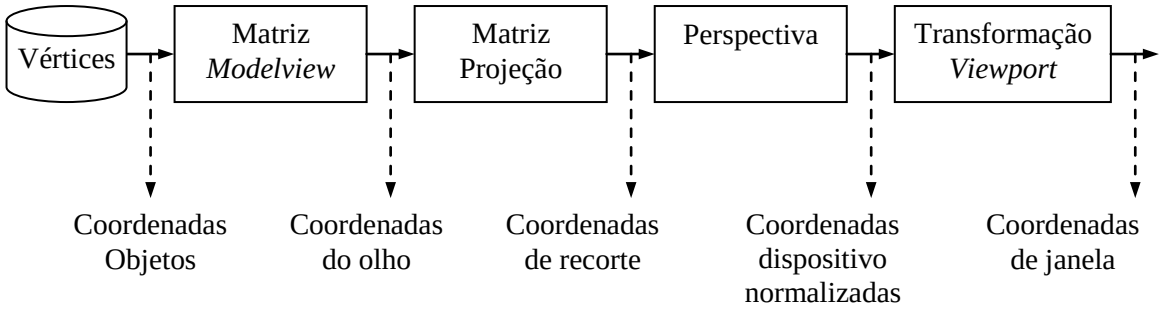
  Como o nome sugere, esta matriz **acumula transformações** geométricas aplicadas a objetos da cena, bem como coordenadas de visualização da câmera sintética (no caso de uma visualização 3D).

Apenas uma matriz pode ser modificada de cada vez. A matriz default é a *modelview*. Todas as matrizes têm dimensão 4x4.

A visualização das cenas é composta por várias transformações: ***modelview***, ***projection*** e ***viewport***.

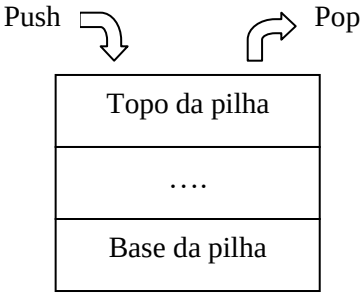
- As transformações de *viewing* devem preceder transformações de modelagem em um programa OpenGL. Devem ser definidas preferencialmente na *callback* `onReshape()`.
- Operações de ***viewport*** e **projeção** podem ser especificadas em qualquer local do código, desde que antes da imagem gerada ser exibida.

A seqüência de operações realizadas para visualização é mostrada na seguinte figura. Detalhes matemáticos destas transformações podem ser encontradas em [11].



- As **coordenadas do olho** são resultantes da aplicação de transformações de modelagem e de câmera sintética.
- As **coordenadas de recorte** são geradas a partir do **volume de visão** gerados pela matriz de projeção em perspectiva ou ortográfica.
- As **coordenada de janela** são geradas pela transformação de *viewport*.

Tanto a matriz *modelview* como a matriz *projection* são na verdade **pilhas de matrizes de transformação** que permitem operações de inserção e remoção de matrizes de transformação. Cada matriz, como já mencionado, tem dimensão 4x4 (em coordenadas homogêneas). Seguindo a filosofia de uma pilha, somente o elemento do topo pode ser modificado.



O recurso de pilha de matrizes de transformações é útil para construção de modelos hierárquicos, no qual objetos complexos são construídos a partir de objetos simples.

- Ao se desenvolver um jogo, por exemplo, transformações aplicadas ao cenário devem também ser aplicadas a todos os elementos da cena.
- Transformações aplicadas sobre personagens, devem levar em consideração a transformação do mundo bem como a transformação do personagem.
- O mesmo pode ser aplicado sobre o braço de um robô: transformações aplicadas à base têm influência em todo o braço. Transformações aplicadas às extremidades, devem também levar em consideração transformações aplicadas em seções anteriores do braço.

Existem diversos comandos para manipulação destas matrizes. Todos estes comandos operam sobre a **pilha ativa**, determinada pelo comando `glMatrixMode()`.

- **`void glMatrixMode`**`(`**`GLenum`** `mode)`: Define a pilha de matrizes ativa. Tem como argumentos `GL_MODELVIEW`, `GL_PROJECTION`, ou `GL_TEXTURE`.
- **`void glPushMatrix`**`(void)`: Insere no topo da pilha uma cópia da matriz que estava no topo até então, ou seja, faz uma duplicação da matriz do topo.
- **`void glPopMatrix`**`(void)`: Remove a matriz do topo da pilha, destruindo seu conteúdo. A matriz que estava uma posição abaixo torna-se o novo topo.
- **`void glLoadIdentity`**`(void)`: Usado para inicializar a matriz topo da pilha ativa com a matriz identidade.
- **`void glLoadMatrix`**`{fd}(`**`const TYPE`** `*m)`: Seta os 16 valores da matriz (4x4) corrente com a matriz especificada em *m*. Para o OpenGL a matriz m deve estar no formato coluna-linha, diferentemente da especificação da linguagem C. Deste modo, sugere-se declarar *m* como sendo *m[16]* ao invés de *m[4][4]*.

- **`void glMultMatrix`**`{fd}(`**`const TYPE`** `*m)`: Multiplica a matriz corrente pela matriz passada em *m*. O valor resultante é armazenado na própria matriz. Toda as multiplicações de matriz no OpenGL operam de forma que a matriz corrente *C* é multiplicada pela matriz argumento *M* da seguinte forma *CM*, que não é igual à *MC* (a propriedade comutativa não se aplica à multiplicação de matrizes).

7.1 Transformações do Modelo

O OpenGL oferece três comandos para transformações de modelos geométricos. Todas as transformações são aplicadas em relação a origem do sistema de coordenadas.

`void glTranslate``{fd}(`**`TYPE`** `x, `**`TYPE`** `y, `**`TYPE`** `z)`: Multiplica a matriz corrente por uma matriz que move (translada) um objeto por valores *x*, *y*, e *z* (ou move o sistema de coordenadas local pelo mesmo valor).

`void glRotate``{fd}(`**`TYPE`** `angle, `**`TYPE`** `x, `**`TYPE`** `y, `**`TYPE`** `z)`: Multiplica a matriz corrente por uma matriz que rotaciona um objeto (ou o sistema de coordenadas local) em sentido anti-horário ao redor de um raio que parte da origem e que passa pelo ponto *(x, y, z)*. O ângulo de rotação é especificado em graus.

`void glScale``{fd}(`**`TYPE`** `x, `**`TYPE`** `y, `**`TYPE`** `z)`: Multiplica a matriz corrente por uma matriz que escala ou reflete um objeto em relação a cada eixo proporcional aos argumentos *x*, *y*, e *z*.

Para se aplicar seqüências de transformações a objetos deve-se observar a ordem em que as operações são aplicadas. Como mencionado anteriormente, a aplicação de uma transformação à matriz corrente ocorre no lado direito, ou seja, se a matriz corrente é *C*, a multiplicação de uma matriz *M* resulta em *CMv*, onde *v* é um vértice do objeto.

Caso for necessário aplicar três transformações *A*, *B* e *C* nesta seqüência, deve-se definir a seguinte seqüência de transformações:

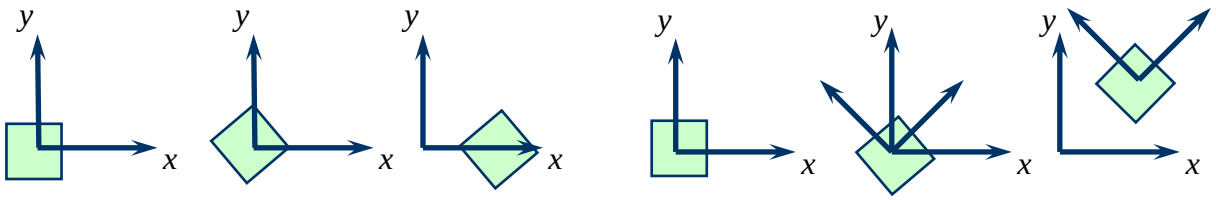
```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();   (Matriz Identidade I)
glMultMatrixf(C);   (Matriz C)
glMultMatrixf(B);   (Matriz CB)
glMultMatrixf(A);   (Matriz CBA)
glBegin();
   glVertex3fv(v);
glEnd();
```

Deste modo, a matriz de transformação contém sucessivamente *I*, *C*, *CB*, *CBA*. O vértice transformado é *CBAv*. Deste modo, a transformação do vértice é *C(B(Av))*, ou seja , *v* é inicialmente multiplicado por *A*, o resultado *Av* por *B* e na seqüência por *C*, ou seja, a **multiplicação ocorre na ordem inversa da especificação**. Na prática, o nó *v* é multiplicado por uma matriz resultante de *CBA*.

A especificação dos comandos de transformação do OpenGL segue um sistema de **coordenadas global fixo**, que exige uma ordem inversa para especificação. Outro sistema é o **sistema local móvel** associado a cada objeto, que faz uso de uma ordem natural das transformações. Neste segundo caso, o sistema de coordenadas é fixo a cada objeto da cena. Se o objeto se mover, o sistema de coordenadas move também. Todas as operações são relativas ao novo sistema de coordenadas. Nos seguintes exemplos ilustra-se os dois sistemas.

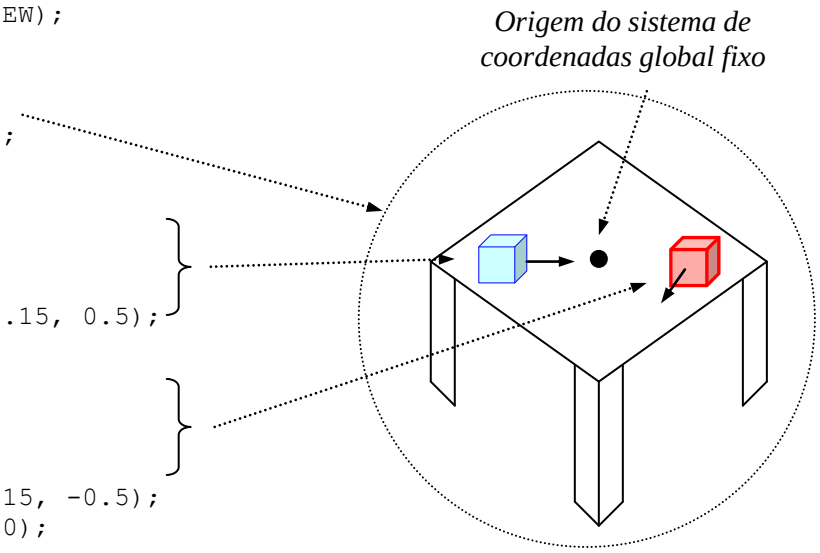
Sistema local móvel	**Sistema global fixo (OpenGL)**

`glMatrixMode(GL_MODELVIEW);`
`glLoadIdentity();`
`glRotatef(45,0,0,1);`
`glTranslatef(10,0,0);`
	`glMatrixMode(GL_MODELVIEW);`
	`glLoadIdentity();`
	`glRotatef(45,0,0,1);`
	`glTranslatef(10,0,0);`



No seguinte exemplo, o primeiro comando de translação tem efeito em toda a cena. Com o comando `glPushMatrix()`, pode-se associar transformações específicas a cada elemento da cena. Após desenhar o primeiro cubo, desempilha-se e volta-se ao estado global para pode então aplicar a transformação ao cubo vermelho.

```
void render()
{
   ...
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity( );

   ...
   glTranslated(0.1, 0, 5);

   renderTable();

   glPushMatrix();
      glColor3f(0, 0, 1);
      glTranslated(-0.5, 0.15, 0.5);
      glutWireCube(0.2);
   glPopMatrix();

   glPushMatrix();
      glColor3f(1, 0, 0);
      glTranslated(0.5, 0.15, -0.5);
      glRotated(90, 0, 1, 0);
      glutWireCube(0.2);
   glPopMatrix();
   ...
```



7.2 Transformações da câmera

Como já mencionado, uma transformação de visualização (câmera) muda a **posição e orientação do *viewpoint*** (ponto de visualização). A câmera sintética possui uma posição, direção de visualização e um vetor *up* que indica a sua rotação em relação ao vetor direção de visualização. A câmera default do OpenGL está posicionada na origem, apontada na direção do eixo *z* negativo, com o vetor *up* alinhado com o eixo *y* positivo.

A transformação de *viewing* deve estar associada a matriz de *modelview*. Existem algumas formas de aplicar transformações de *viewing*:

- Usar um ou mais comandos de transformações de modelos, como `glRotate*()` e `glTranslate*()`;
- Usar o comando `gluLookAt()`.

Supondo como exemplo que se tenha **um objeto centrado na origem**, na mesma posição que a câmera esta definida. Neste caso, mover a câmera para traz (z positivo) ou mover o objeto para frente (z negativo) tem o mesmo efeito. A mesma analogia pode ser feita em relação à rotações.

```
void gluLookAt(GLdouble eyex,    GLdouble eyey,    GLdouble eyez,
               GLdouble centerx, GLdouble centery, GLdouble centerz,
               GLdouble upx,     GLdouble upy,     GLdouble upz)
```

Define uma matriz de visualização que é multiplicada à direita da matriz corrente. O vetor `eye` especifica onde a câmera **está posicionada**, `center` onde a câmera **está olhando** (coordenadas absolutas do sistema global fixo) e `up` **a inclinação** da câmera.

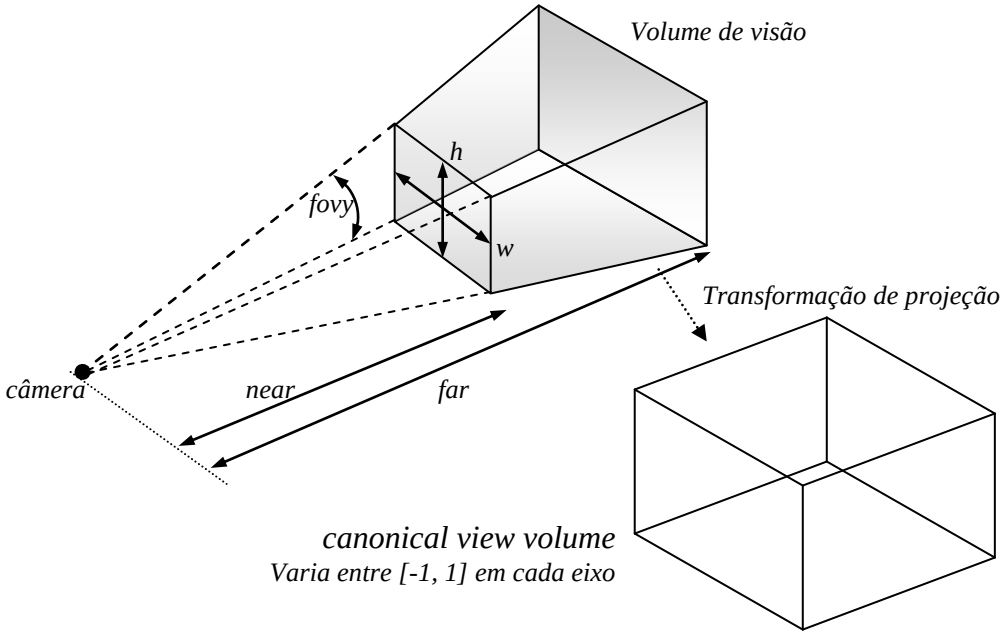
O comando `gluLookAt()` deve ser **chamado antes** das operações de transformações dos modelos da cena, caso contrário ele não terá efeito na visualização da cena.

As transformações aplicadas por este comando mudam a matriz *modelview* de modo que a cena é vista como se a câmera estivesse alinhada ao eixo *z*, apontada em direção ao *z* negativo.

O comando **`glut`**`LookAt()` não está disponível na API do OpenGL visto que é um comando que faz uso de vários comandos básicos do OpenGL como `glRotate*()` e `glTranslate*()`.

7.3 Projeções

O OpenGL oferece comandos para definição de dois tipos de projeção: ortográfica e perspectiva. Estas duas projeções afetam a matriz *projection*, por isso deve-se torná-la ativa com o comando `glMatrixMode(GL_PROJECTION)`. Um ótimo material sobre o assunto pode ser encontrado em http://www.songho.ca/opengl/gl_transform.html#projection.



O objetivo da projeção é a definição do **volume de visão**, que é usado para determinar como um objeto será projetado na tela e quais objetos ou parte de objetos serão removidos da imagem final. Com esta estratégia de recorte, objetos atrás da câmera são automaticamente eliminados, o que não ocorre quanto se utiliza um modelo de câmera sem recorte.

Para aplicar projeção em perspectiva pode-se utilizar o comando `glFrustum()` ou o comando da API auxiliar **`glu`**`Perspective()`. A utilização deste segundo comando é mais intuitiva e por isso descrita neste tutorial.

```
void gluPerspective(GLdouble fovy, GLdouble aspect,
                    GLdouble near, GLdouble far);

void glFrustum(GLdouble left,   GLdouble right,
               GLdouble bottom, GLdouble top,
               GLdouble near,   GLdouble far);
```

O comando `gluPerspective()` cria uma matriz para geração do ***frustum*** (*the portion of a solid – normally a cone or pyramid – which lies between two parallel planes cutting the solid* [Wikipedia]) de visão em perspectiva e a multiplica pela matriz corrente. Para a demonstração das matrizes de transformação, consulte http://www.songho.ca/opengl/gl_projectionmatrix.html.

- ***fovy*** é o ângulo do campo de visão no plano *x*-*z* e deve estar no intervalo [0, 180]. Quanto maior for, maior é a abertura da "lente" da câmera sintética.
- ***aspect*** é a razão de aspecto do *frustum* e é dado por *w*/*h*.
- ***near*** e ***far*** são valores de distância entre o *viewpoint* e os planos de recorte, ao longo do eixo *z* negativo.

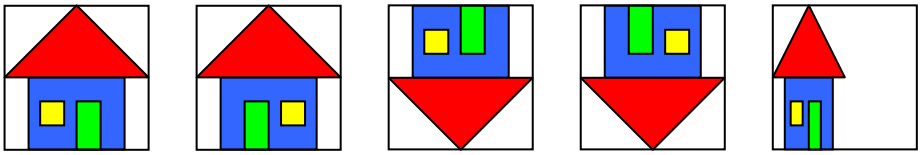
Outra forma de projeção é a ortográfica. Nos seguintes exemplos ilustra-se a utilização de uma matriz de projeção ortográfica para cenas 2D (sem a componente *z*). Nestes exemplos, utiliza-se uma *viewport* do tamanho da janela de desenho.

Utiliza-se o comando `gluOrtho2D()` (ver comando **`glOrtho()`** para projeções ortográficas de cenas 3D) para especificação da matriz de projeção ortográfica.

- A área de recorte tem como parâmetros as coordenadas do canto esquerdo inferior (*left*-*bottom*) e do canto direito superior (*right*-*top*).

```
void gluOrtho2D(GLdouble left,   GLdouble right,
                GLdouble bottom, GLdouble top);

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

```
void onReshape(int w, int h)
{
   ...
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluOrtho2D(0, w, 0, h);    (1)
   gluOrtho2D(w, 0, 0, h);    (2)
   gluOrtho2D(0, w, h, 0);    (3)
   gluOrtho2D(w, 0, h, 0);    (4)
   gluOrtho2D(0, w*2, 0, h); (5)
   glMatrixMode(GL_MODELVIEW);
}
```



1 **2** **3** **4** **5**

Nestes exemplos, as coordenadas da figura variam entre [0, w] em largura e [0, h] em altura, ou seja, proporcionais às dimensões de especificação da ***área de recorte***.

Toda informação gerada pela projeção ortográfica deve ser exibido dentro da *viewport*, com transformações sempre que necessário. Para o exemplo 5, como a projeção gera um volume de visão 2x mais largo que a largura da janela, ao ser exibida na *viewport* esta informação tem que ser reescalada, resultando em uma deformação do objeto.

Em muitos casos é conveniente especificar a figura com coordenadas entre [0, 1] ou [-1, 1]. Neste caso, deve-se mudar a matriz de projeção para a escala apropriada.

```
gluOrtho2D( 0, 1,  0, 1);
gluOrtho2D(-1, 1, -1, 1);
```

O comando `gluOrtho2D()` altera a matriz de projeção, por isso deve-se definir com o comando `glMatrixMode(GL_PROJECTION)` que a matriz corrente é a matriz de projeção. Uma vez configurada a matriz de projeção, volta-se a habilitar a matriz de *modelview* (`GL_MODELVIEW`). Isso indica que as transformações subsequentes irão afetar a *modelview* ao invés da matriz de projeção.

Por default, caso as matrizes de *modelview* e *projection* não forem setadas, elas assumem a matriz **identidade**. Isso pode ser verificado com os seguintes comandos:

```
GLfloat m[16]; //usado para leitura das matrizes
glGetFloatv (GL_PROJECTION_MATRIX, m); //ou GL_MODELVIEW_MATRIX para a modelview
for(int l=0; l<4; l++)
{
   printf("\n");
   for(int c=0; c<4; c++)
   {
       printf(" %.2f", m[l * 4 + c] );
   }
}
```

O seguinte exemplo ilustra a utilização de vários comandos de transformação. Neste exemplo foi utilizado tanto a projeção ortográfica como a perspectiva. O comando `glOrtho(-1,1,-1,1,1,-1)` gera a matriz identidade (*canonical view volume*), e desta forma, não afeta a renderização. Logo, se nenhuma matriz de projeção for definida, o OpenGL assume a matriz identidade, que por usa vez é igual a projeção ortográfica default: `glOrtho(-1,1,-1,1,1,-1)`. Porém, pode-se utilizá-lo em conjunto com `gluPerspective()` em caso de necessitar mudar eixos, como em `gluOrtho2D(-1,1,1,-1)`, que faz a inversão do eixo *y*, ou `gluOrtho2D(1,-1,-1,1)` que faz a inversão do eixo *x*.

```
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

   glMatrixMode(GL_PROJECTION);
   glLoadIdentity( );
   glOrtho(-1,1,-1,1,1,-1); //deve ser chamado antes de gluPerspective.
   gluPerspective(abertura, aspect, znear, zfar);

   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity( );
   gluLookAt(0, 0, 3,  //from. Posicao onde a camera esta posicionada
             0, 0, 0,  //to. Posicao absoluta onde a camera esta vendo
             0, 1, 0); //up. Vetor Up.

   glRotatef ((GLfloat) rz, 1.0f, 0.0f, 0.0f);

   glColor3f(1, 1, 1);
   glBegin(GL_POLYGON);
      glVertex3f(-1, 0, 1);
      ...
}
```

7.4 Viewport

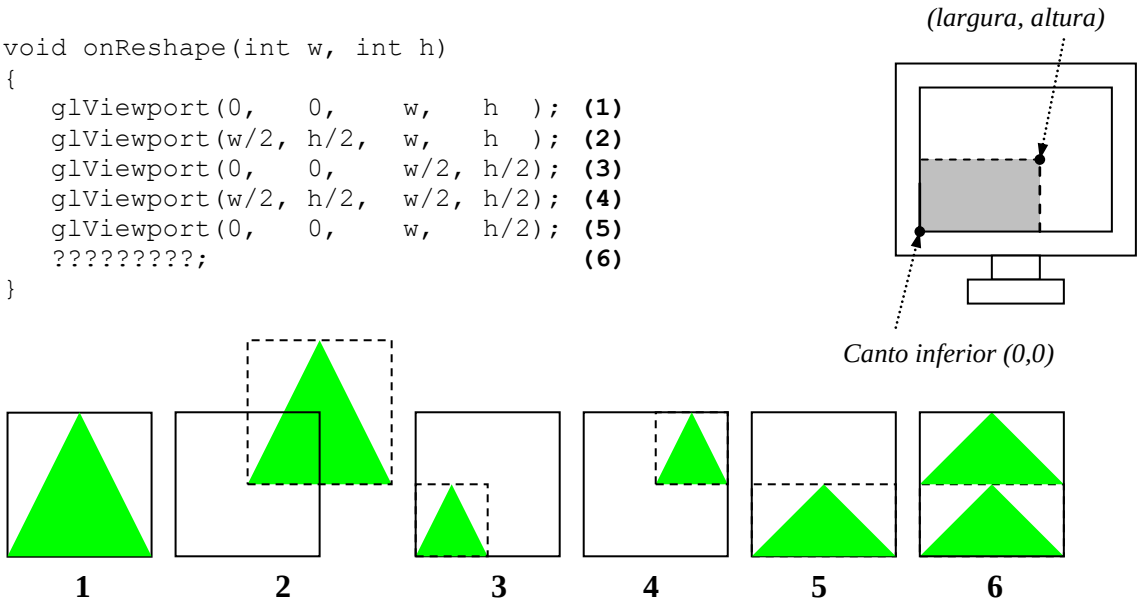
Após os cálculos de projeção, como última etapa do processo aplica-se a transformação de *viewport*. A determinação da *viewport* pode ser dada de várias maneiras, como mostrado no seguinte exemplo.

- Observe que a especificação da *viewport* deve ocorrer preferencialmente na *callback* `onReshape()`.

O comando `glViewport(x,y,width,height)` é usado para determinar a área disponível da janela para desenho.

- Os dois primeiros argumentos indicam a origem da janela e os outros dois a dimensão.
- Esse é o motivo do porque utilizar este comando na *callback* de `reshape()` – sempre que a janela mudar de tamanho, deve-se mudar o tamanho da *viewport*.
- Utilizando-se os parâmetros *w* e *h*, mantém-se a proporcionalidade desejada.
- Este comando somente é necessário caso a *viewport* não for do tamanho da janela.

```
void onReshape(int w, int h)
{
   glViewport(0,   0,    w,   h  ); (1)
   glViewport(w/2, h/2,  w,   h  ); (2)
   glViewport(0,   0,    w/2, h/2); (3)
   glViewport(w/2, h/2,  w/2, h/2); (4)
   glViewport(0,   0,    w,   h/2); (5)
   ?????????;                     (6)
}
```



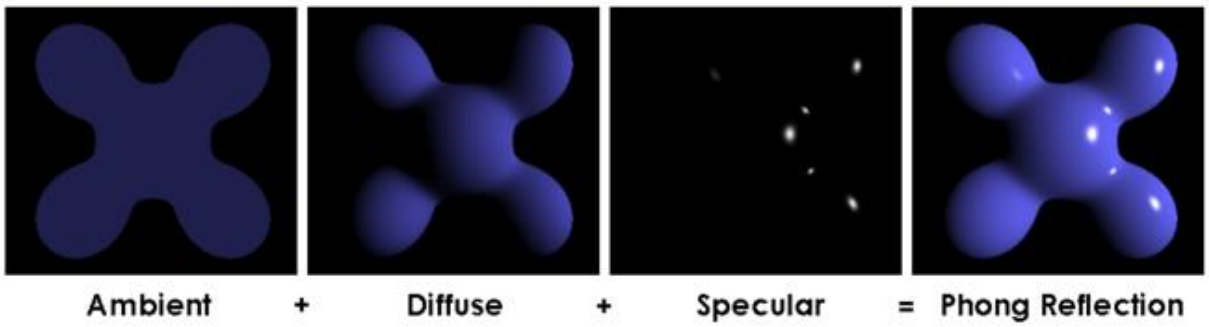
Como pode ser observado nas figuras, caso o tamanho da *viewport* for maior que o tamanho da janela, haverá um recorte dos objetos fora da janela.

- Se a *viewport* for menor que a janela, todos os objetos desenhados serão reescalados para serem exibidos dento da *viewport*.
- Pode-se também exibir duas ou mais *viewports* dentro de uma mesma janela (Figura 6). Neste caso, deve-se definir uma *viewport*, renderizar os objetos, redefinir outra *viewport* e desenhar os novos objetos.

8 Iluminação

O OpenGL permite a definição de fontes luminosas, modelos de iluminação e características dos materiais (o que tem influência direta na reflexão da luz).

A iluminação é fundamental para dar a noção de profundidade. Sem iluminação, uma esfera e um disco possuem a mesma representação. Para se visualizar uma cena 3D sem o uso de iluminação, pode-se utilizar texturas ou fazer a visualização em *wireframe*.



[http://en.wikipedia.org/wiki/Phong_reflection_model]

O OpenGL oferece cálculo da iluminação Ambiente, Difusa e Especular (Gouraud). Cada componente é tratada e configurada separadamente para cada componente RGB, o que permite fazer operações que não ocorrem no mundo real. A cor deve ser especificada para as fontes como para os materiais.

- Na fonte, especifica-se a cor que a conte irá emitir. Se *R=G=B=1*, a fonte irá emitir cor branca, e tem a intensidade máxima. Se os valores forem 0.5, a fonte ainda emitirá branco, porém com apenas 50% da intensidade. Se somente *R=1*, a fonte emitirá cor vermelha.
- No material especifica-se a taxa de reflexão de cada componente. Se *R=1* e *G=0.5*, o material reflete toda luz vermelha incidente e apenas 50% da cor verde e não reflete a cor azul.
- Considerando-se que a fonte emite (L_R, L_G, L_B), que o material reflete (M_R, M_G, M_B), desconsiderando-se todos outros efeitos de reflexão, a luz que chega ao observador é dada por (L_R*M_R, L_G*M_G, L_B*M_B).
- Caso existir mais de uma fonte, caso a somatória de intensidades $L_{R1}+L_{R2}+..L_{Rn}$ for maior que 1, este valor é truncando em 1 (função *clamp*).

Para ser aplicada a iluminação a uma cena necessita-se:

1. Definição dos vetores normais da superfície
2. Ativação e definição da fonte de luz
3. Selecionar um modelo de iluminação
4. Definição de propriedades dos materiais dos objetos da cena.

Para a ativação das fontes de luz (**Passo 1**), deve utilizar os seguintes comandos:

```
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
…
glEnable(GL_LIGHT7);
```

O OpenGL permite a definição de até 8 fontes. Por default a fonte 0 tem cor branca e as demais a cor preto. Para mudar a cor, dentre outros parâmetros da fonte (**Passo 2**), deve-se utilizar o comando `glLightf*v()`.

`void glLight``{if}`**`v`**`(`**`GLenum`** `light, `**`GLenum`** `pname, `**`TYPE`** `*param);`

- **`light`**: Especifica o ID da fonte que deve ser especificada. Pode assumir os seguintes valores: `GL_LIGHT0`, `GL_LIGHT1`, ... , ou `GL_LIGHT7`.
- **`pname`**: define a característica a ser alterada da fonte `light`. Pode-se passar os seguintes argumentos:
  - `GL_AMBIENT`: intensidade RGBA ambiente da luz
  - `GL_DIFFUSE`: intensidade RGBA difusa da luz
  - `GL_SPECULAR`: intensidade RGBA especular da luz
  - `GL_POSITION`: posição (*x*, *y*, *z*, *w*) da luz. w deve ser 0.
  - `GL_SPOT_DIRECTION`: direção (*x*, *y*, *z*) do *spotlight*.
  - `GL_SPOT_EXPONENT`: expoente do *spotlight*.
  - `GL_SPOT_CUTOFF`: ângulo de corte (float). O default é 180 (sem restrição)
  - `GL_CONSTANT_ATTENUATION`: fator de atenuação constante (float).
  - `GL_LINEAR_ATTENUATION`: fator de atenuação linear (float).
  - `GL_QUADRATIC_ATTENUATION`: fator de atenuação quadrático (float).
- **`param`**: ponteiro para o conjunto de dados. Pode ter 1, 3 ou 4 componentes, dependendo do argumento `pname`.

```
GLfloat light_0_position[] = { 1, 1, 1, 0 };
GLfloat light_0_difuse[]   = { 0, 0, 1 };  //BLUE
GLfloat light_0_specular[] = { 1, 1, 1 };  //WHITE
GLfloat light_0_ambient[]  = { 0.2, 0.2, 0.2 };
```

```
glLightfv(GL_LIGHT0, GL_POSITION, light_0_position);
glLightfv(GL_LIGHT0, GL_DIFFUSE,  light_0_difuse);
glLightfv(GL_LIGHT0, GL_SPECULAR, light_0_specular);
glLightfv(GL_LIGHT0, GL_AMBIENT,  light_0_ambient);
```

O OpenGL permite também a especificação de modelos de iluminação (**Passo 3**) com o comando `glLightModel*()`. O modelo é usado para definir a posição do *viewpoint*, definir se a iluminação será aplicada nos dois lados das faces (*front-* e *back-facing*) e para especificação da iluminação global ambiente. Para definir a iluminação global ambiente, que independe de fontes de iluminação, pode-se utilizar o seguinte comando:

```
GLfloat lmodel_ambient[] = { 0.2, 0.2, 0.2, 1.0 };
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodel_ambient);
```

Neste caso, qualquer objeto da cena terá iluminação ambiente, independente da posição e configuração das fontes luminosas. Mesmo com nenhuma fonte habilitada, com este fator de iluminação ainda pode-se visualizar a cena.

O próximo passo consiste na especificação dos materiais dos objetos da cena (**Passo 4**). Para isso existe o comando `glMaterialfv()`.

`void glMaterial``{if}`**`v`**`(`**`GLenum`** `face, `**`GLenum`** `pname, `**`TYPE`** `*param);`

- **`face`**: indica em qual lado da face a propriedade será aplicada. Pode assumir `GL_FRONT`, `GL_BACK`, ou `GL_FRONT_AND_BACK`.
- **pname**: define a característica a ser alterada no material. Pode-se passar os seguintes argumentos:
  - `GL_AMBIENT`: cor RGBA ambiente do material;
  - `GL_DIFFUSE`: cor RGBA difusa do material;
  - `GL_AMBIENT_AND_DIFFUSE`: cor ambiente e difusa do material;
  - `GL_SPECULAR`: cor RGBA especular do material
  - `GL_SHININESS`: expoente especular (float).
  - `GL_EMISSION`: cor RGBA emissiva do material.
  - `GL_COLOR_INDEXES`: índice de cores.
- **`param`**: ponteiro para o conjunto de dados. Pode ter 1, 3 ou 4 componentes, dependendo do argumento `pname`.

```
GLfloat mat_specular[] = { 1, 1, 1, 1 };
GLfloat mat_shininess[] = { 10 };
glMaterialfv(GL_FRONT, GL_SPECULAR,  mat_specular);
glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
```

- → A especificação de parâmetros das fontes e materiais é um trabalho que exige **experimentos e prática**. Não é trivial encontrar uma combinação ideal de forma imediata.
- → Deve-se observar que a iluminação do OpenGL **não gera sombras**. Algoritmos para sombra devem ser implementados separadamente.
- → Se a iluminação é realizada por vértices, que estratégia pode ser utilizada para gerar um cubo, por exemplo, com uma reflexão máxima apenas no centro de uma face?
- → A posição da fonte luminosa (`GL_POSITION`) sofre o efeito da matriz GL_MODELVIEW corrente.

9. Textura

Para dar as superfícies características mais realistas, o OpenGL oferece além de recursos de iluminação, recurso de texturização. Faz-se uso da técnica de *texture mapping* para aplicar a textura sobre a superfície dos polígonos. A idéia é mapear cada face do modelo em coordenadas de textura, geralmente representada por uma imagem, como na seguinte figura. A definição de coordenadas de textura, para cada triângulo do

modelo, que neste exemplo é um modelo de personagem animado 3D (Formato MD2), é feita pelo modelador (artista gráfico).

A textura faz parte do estágio de rasterização do pipeline gráfico do OpenGL. A rasterização é a conversão de **dados geométricos** (polígonos) e **pixels** (textura) em **fragmentos**. Cada fragmento corresponde a um pixel no *framebuffer*.

O mapeamento da textura para os fragmentos **nem sempre preserva as proporções**. Dependendo da geometria do objeto, podem haver **distorções** da textura ao ser aplicada. Também pode acontecer de um mesmo **texel** (*texture element*) ser aplicado a mais de um fragmento como um fragmento mapear vários texels. Para isso **operações de filtragem** são aplicadas.

Uma textura é um mapa retangular de dados (geralmente cores como em uma imagem). O OpenGL oferece algumas maneiras de se especificar textura: pela especificação de coordenadas para cada vértice do modelo ou pela geração automática de coordenadas de textura.

Como o assunto de textura é muito amplo, nesta seção são apresentados alguns comandos com alguns parâmetros possíveis. Para maiores detalhes, consulte [1].

Restrições da textura: A textura deve ter preferencialmente dimensões em potência de 2, como por exemplo: 64x128, 512x512, 128x64, etc.

- A função `glTexImage2D()`, como será visto, não funciona se a textura não seguir esta restrição.
- Outra função equivalente, a `gluBuild2DMipmaps()`, por sua vez, opera com textura de qualquer dimensão.
- Se a textura não for potência de 2, deve-se garantir que a posição de memória do pixel que representa o início de uma linha seja múltipla de 4 (mesmo formato adotado pelo padrão de imagem BMP para imagens que não são potência de 2). Por exemplo, uma imagem em RGB de 3x4 pixels, vai ocupar 12 bytes por linha, ao invés de 9, como esperado. Ver demo de textura.

Para se aplicar textura deve-se inicialmente habilitar a textura, criar a textura, definir parâmetros e aplicar a textura. Para habilitar o recurso de textura utilizam-se os comandos

```
glEnable(GL_TEXTURE);
glEnable(GL_TEXTURE_2D);
```

Cada textura esta associada a um ID que é usado pelo OpenGL para definição da textura ativa. A cada momento existe uma única textura ativa, que é aplicada sobre todos os objetos que possuem coordenadas de textura.

Para criar um ID utiliza-se o comando `glGenTextures(1,&id)` que indica que serão criados 1 índice de textura, e que este índice será armazenado na variável intera `id`.

Par se criar, habilitar e desabilitar a textura, utiliza-se o comando

```
void glBindTexture(GLenum target, GLuint textureName);
```

O parâmetro `textureName` corresponde ao ID da textura. O parâmetro `target` pode assumir `GL_TEXTURE_1D` ou `GL_TEXTURE_2D`.

- Quando este comando for chamado com um ID de textura inteiro diferente de zero pela primeira vez, um novo objeto de textura é criado com o nome passado.
- Se a textura já havia sido criada, ela é então habilitada.
- Se o parâmetro `textureName` for zero, desabilita o uso de textura do OpenGL.

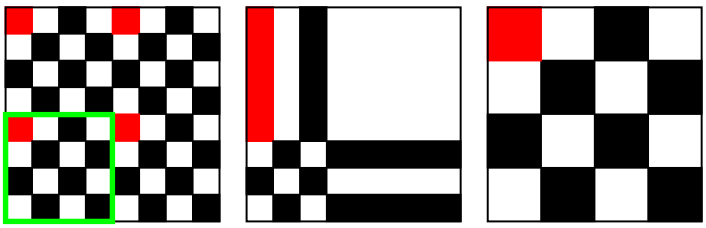
Após a criação da textura (nome), deve-se configurar **parâmetros** de aplicação da textura sobre um fragmento.

```
void glTexParameter{if}(GLenum target, GLenum pname, TYPE param);
```

- `target`: pode assumir `GL_TEXTURE_1D` ou `GL_TEXTURE_2D`.
- `pname` e `param`: formam uma dupla. Para cada valor de `pname`, existem possíveis valores para `param`.

Pname	param
GL_TEXTURE_WRAP_S	GL_CLAMP, GL_REPEAT
GL_TEXTURE_WRAP_T	GL_CLAMP, GL_REPEAT
GL_TEXTURE_MAG_FILTER	GL_NEAREST, GL_LINEAR
GL_TEXTURE_MIN_FILTER	GL_NEAREST, GL_LINEAR, GL_NEAREST_MIPMAP_NEAREST, GL_NEAREST_MIPMAP_LINEAR, GL_LINEAR_MIPMAP_NEAREST, GL_LINEAR_MIPMAP_LINEAR

Os parâmetros `GL_TEXTURE_WRAP_S/T` representam como a textura será aplicada sobre objeto, caso o objeto for maior que a textura. `GL_CLAMP` usa a cor da borda e `GL_REPEAT` replica a textura. `GL_NEAREST` indica que deve ser usado o texel com coordenadas mais próximas do centro do pixel. `GL_LINEAR` faz uma média dos 2x2 texels vizinhos ao centro do pixel.



Os parâmetros `GL_TEXTURE_MAG/MIN_FILTER` indicam como a textura será filtrada. Isso ocorre porque após a projeção, pode ocorrer de parte de um *texel* aplicado sobre um pixel (*magnification*) ou de vários *texels* serem aplicados sobre um único pixel (*minification*). O argumento passado vai indicar como será realizada a média ou interpolação dos valores.

A última etapa consiste na definição dos dados da textura. Para isso existem dois comandos: `glTexImage2D()` e `gluBuild2DMipmaps()`. Mipmapping é uma técnica usada para criar várias copias da textura em diferentes resoluções (1/4+1/16+.... ≈ 1/3, ou seja, aumento de 33% do uso de memória), facilitando assim a escolha da melhor textura a ser aplicada sobre o objeto em função da sua área de projeção (http://en.wikipedia.org/wiki/Mipmap):

```
void glTexImage2D(GLenum target,
                  GLint level,
                  GLint internalFormat,
                  GLsizei width,
                  GLsizei height,
                  GLint border,
                  GLenum format,
                  GLenum type,
                  const GLvoid *pixels);
```

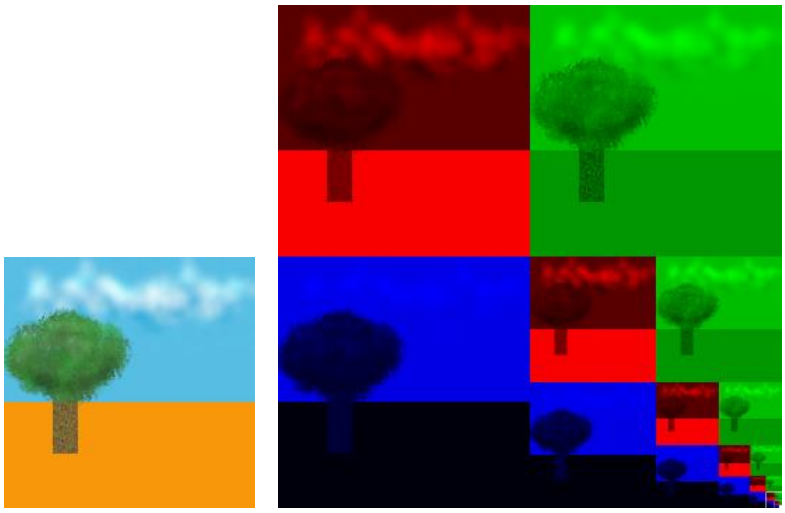
- `level`: comumente deve ser 0. Ver função `gluBuild2DMipmaps()` para geração automática de textura em várias resoluções.
- `internalFormat`: indica como estão armazenados os dados no array `pixels` (imagem). Pode assumir vários valores. O parâmetro mais comum é `GL_RGB`, para indicar que o array é um vetor de coordenadas RGB.
- `width`: largura da imagem em pixels
- `height`: altura da imagem em pixels
- `border`: indica a largura da borda. Valor 0 significa sem borda.
- `format`: Formato dos dados. Use `GL_RGB`.
- `type`: Geralmente deve ser `GL_UNSIGNED_BYTE` para indicar que cada componente RGB utiliza 1 byte não sinalizado.
- `pixels`: contém os dados da imgem-textura. Ver restrições de dimensão de textura.

```
int gluBuild2DMipmaps(GLenum target,
                      GLint components,
                      GLint width,
                      GLint height,
                      GLenum format,
                      GLenum type,
                      void *data);
```

Esta função cria uma série de *mipmaps* e chama a função **`glTexImage*D`**`()` para carregar as imagens. Por isso é uma função `glu`. Vários argumentos são os mesmos da função `glTexImage2D()`**.**

No fórum [https://community.khronos.org/t/how-to-anti-alias-images-shown-as-2d-textures/59657] comentam que algoritmos de antialiasing para textura funcionam apenas para supersampling. Para downsampling, a solução é o uso de Mipmaping (ou uso de técnicas mais complexas com o uso de FBO – Frame Buffer Object). Deve-se observar que a Glut já está descontinuada, e por consequência o recurso de mipmapping pode não estar disponível em outras APIs semelhantes. Outro material bem interessante e completo é [https://learnopengl.com/Advanced-OpenGL/Anti-Aliasing], que aborda inúmeros tópicos sobre OpenGL. Algumas soluções indicam o uso de Extensões do OpenGL, porém nem todas estão disponíveis em todas as placas de vídeo. A solução mais garantida e genérica para criar aplicações que possam rodar em qualquer máquina é a implementação "na mão" do recurso de Mipmapping.

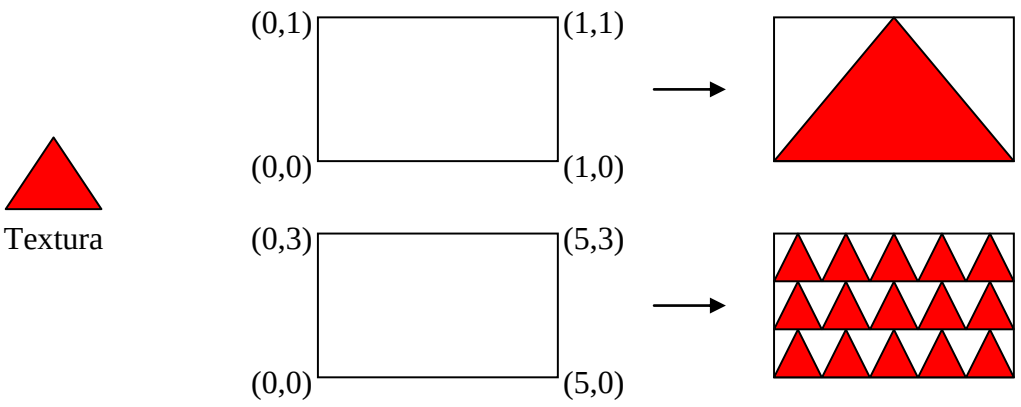
Exemplo de Mipmap: textura original e cópias em escalas de 2 (separada em RGB).



Para se especificar as coordenadas de textura para cada vértice utiliza-se o comando `glTexCoord*()`.

```
void glTexCoord{1234}{sifd}(TYPE coords);
void glTexCoord{1234}{sifd}v(TYPE *coords);
```

Este comando seta as coordenadas de textura corrente (*s*, *t*, *r*, *q*). Vértices definidos na seqüência com `glVertex*()` fazem uso do valor corrente de textura. Geralmente utiliza-se o comando `glTexCoord2f()` para definir as coordenadas *s* e *t*. Coordenadas de textura são multiplicadas pela matriz de textura antes que o mapeamento ocorra. Por default esta matriz é a identidade. A matriz de textura pode conter vários tipos de transformações como rotações, translações, perspectivas, etc. Para aplicar transformações nesta matriz deve-se setar a matriz de textura como corrente com o comando `glMatrixMode(GL_TEXTURE)`. Com isso pode-se simular a rotação de uma textura sobre uma superfície.



Mapeamento da textura. Adaptada de [3]

Um programa que usa textura tem a seguinte estrutura. Para maiores detalhes, veja o demo de textura.

```
int tex_id;
void init()
{
   glGenTextures( 1, &tex_id );
   glBindTexture( GL_TEXTURE_2D, tex_id );

   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
   ...
   glTexImage2D(GL_TEXTURE_2D,...);
```

```
   glEnable( GL_TEXTURE );
   glEnable( GL_TEXTURE_2D );
}

void render()
{
   glBindTexture( GL_TEXTURE_2D, tex_id );

   glBegin(GL_QUADS);
      glTexCoord2f(0, 0);
      glVertex3f(v[0].x, v[0].y, v[0].z);
      glTexCoord2f(3, 0);
      glVertex3f(v[1].x, v[1].y, v[1].z);
      glTexCoord2f(3, 3);
      glVertex3f(v[2].x, v[2].y, v[2].z);
      glTexCoord2f(0, 3);
      glVertex3f(v[3].x, v[3].y, v[3].z);
   glEnd();
}
```

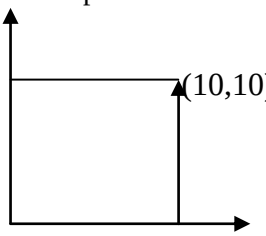
Exemplos de geração de textura:

```
GLuint textureID[MAX_TEXTURES];
glGenTextures( 1, &textureID[0] );
glBindTexture( GL_TEXTURE_2D, textureID[0] );

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
gluBuild2DMipmaps(GL_TEXTURE_2D,
                  GL_RGB,
                  img1->getWidth(),
                  img1->getHeight(),
                  GL_RGB,
                  GL_UNSIGNED_BYTE,
                  data);
glGenerateMipmap(GL_TEXTURE_2D);
```

```
glGenTextures( 1, &textureID[1] );
glBindTexture( GL_TEXTURE_2D, textureID[1] );

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

glTexImage2D(GL_TEXTURE_2D,
             0,
             GL_RGB,
             img1->getWidth(),
             img1->getHeight(),
             0,
             GL_RGB,
             GL_UNSIGNED_BYTE,
             data);
```

Geração automática de coordenadas de textura

O OpenGL permite que as coordenadas de textura sejam criadas automaticamente. Isso é muito útil em diversas aplicações como por exemplo na geração de terrenos onde os vértices formam um reticulado. Considere o caso de um malha poligonal com dimensão 10x10, com origem em (0,0). Independente do número de faces que vai ter, deseja-se aplicar uma única textura sobre toda a malha, sem replicação. Neste caso define-se uma escala 1/10 e especifica-se parâmetros de projeção da textura nos eixos *x* e *z*, como no seguinte exemplo.




```
float escala = 1.0 / 10;
float p1[4] = { escala, 0, 0,      0};
float p2[4] = { 0,      0, escala, 0};
glTexGenfv(GL_S, GL_OBJECT_PLANE, p1);
glTexGenfv(GL_T, GL_OBJECT_PLANE, p2);
```

Como o quadrado tem dimensão de 10, com coordenada inicial em (0,0), se escala valesse 1/20, seria aplicada apenas ¼ da textura. Se escala = 1/5, seriam usadas 4 cópias da textura (com filtro `GL_REPEAT`) para preencher o quadrado.

```
glGenTextures( 1, &id );
glBindTexture( GL_TEXTURE_2D, id);

float escala = 1.0 / 10;
float p1[4] = { escala, 0,      0, 0 };
float p2[4] = { 0,      escala, 0, 0 };
glTexGenfv(GL_S, GL_OBJECT_PLANE, p1);
glTexGenfv(GL_T, GL_OBJECT_PLANE, p2);

glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
gluBuild2DMipmaps(GL_TEXTURE_2D,
                  GL_RGB,
                  img1->getWidth(),
                  img1->getHeight(),
                  GL_RGB,
                  GL_UNSIGNED_BYTE,
                  data);
```

Deve-se habilitar a geração automática de textura e após o uso desabilitar. Maiores detalhes podem ser vistos em [1].

```
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
```

10. Exibição de FPS – Frames Per Second

Passos:

- Calcular o FPS
- Guardar o estado de cada atributos ativo com o comando `glPushAttrib(GLbitfield mask)`. Este comando aceita diversas constantes como argumentos. Os dois mais genéricos são

`GL_ALL_ATTRIB_BITS` e `GL_ENABLE_BIT`. Este comando tem o mesmo efeito do `glPushMatrix()`, só que este opera sobre pilha de atributos.

- Desabilitar todos os atributos com `glDisable()`, para não interferirem na impressão do valor do FPS, que é tratado como uma string:
  - Cor material (`GL_COLOR_MATERIAL`)
  - Coordenadas de textura (`GL_TEXTURE_2D`)
  - Nevoeiro (`GL_FOG`);
  - Iluminação (`GL_LIGHTING`)
  - Z-buffer (`GL_DEPTH_TEST`)
  - Composição (`GL_BLEND`)
- Setar a matriz de projeção como ortográfica
- Habilitar a matriz de *Modelview*,
- Desenhar os caracteres com `glRasterPos2f()`, `glutBitmapCharacter()` e `glutBitmapWidth();`
- Restaurar os atributos com `glPopAttrib();`

Para mais detalhes, veja o demo FPS.

Para calcular o FPS pode-se utilizar o demo FPS disponibilizado nos demos. A classe Frames faz esse cálculo.

```
////////////////////////////////////////////////////////////////////////////////
class Frames {
	clock_t t1, t2;
	long    cont_frames;
	float   fps, fps_old;
public:
	Frames()
	{
		t1 = clock();
		cont_frames = 0;
		fps_old = 20; //valor inicial
	}

	//**********************************************************************
	// Este metodo deve ser chamado uma unica vez a cada atualizacao de tela
	// Retorna o fps da aplicacao. Altere o valor de UPDATE_RATE para diferentes
	// tempos de atualizacao.
	//**********************************************************************
	float getFrames()
	{
		double tempo;

		t2 = clock();
		tempo = (double)(t2 - t1);  //milisegundos
		cont_frames++;
		//apos UPDATE_RATE milisegundos, atualiza o framerate
		if (tempo > 300 ) //0.3 segundos
		{
			t1 = t2;
			fps = cont_frames / (tempo / CLOCKS_PER_SEC);
			cont_frames = 0;
			fps_old = fps;

			return fps;
		}
		//se nao foi decorrido UPDATE_RATE milisegundos, retorna a taxa antiga
		return fps_old;
	}
};
```

11. Modelagem de Objetos

Para se criar cenários em OpenGL pode-se importar modelos já prontos (por exemplo, do 3D Studio Max) ou criar seus próprios modelos, com o uso de primitivas básicas que o OpenGL oferece. Apesar da última ser um processo mais trabalhoso, fica desvinculado de arquivos de dados auxiliares.

Ao se criar objetos modelados com as primitivas do OpenGL deve-se tomar cuidado em relação à orientação das faces. Como ilustrado na seção de Primitivas Geométricas, a ordem que os vértices são definidos é fundamental para se obter uma superfície com os vértices definidos em ordem horária ou anti-horária (*counterclockwise* – padrão do OpenGL)

Caso os vértices forem definidos em ordem errada, surgirão **problemas na iluminação e na remoção de superfícies ocultas** (`glEnable(GL_CULL_FACE);`).

12. Tópicos adicionais

Vários tópicos suportados pelo OpenGL não foram tratados neste material. Para detalhes sobre assuntos como *blending*, *fog*, *antialiasing*, *bitmaps*, *quádricas*, *NURBS*, *picking*, e extensões OpenGL, consulte [1]. Para técnicas de renderização como sombras, efeitos especiais e técnicas de otimização, consulte [2].

13. Referências Bibliográficas

[1] PHIGS. Disponível em: http://en.wikipedia.org/wiki/PHIGS

[2] PHIGS Standard. http://www.itl.nist.gov/iaui/vu/cugini/pvt/hy-top.html

[3] GKS Standard. http://ngwww.ucar.edu/ngdoc/ng4.4/gks/intro.html

[4] Lista comandos OpenGL, GLU e GLUT. Disponível em: http://pyopengl.sourceforge.net/documentation/manual/index.xml

[5] The OpenGL Utility Toolkit (GLUT) Programming Interface API Version 3. http://www.opengl.org/documentation/specs/glut/spec3/spec3.html

[6] Woo, M.; Neider, J.; Davis, T.; Shreiner, D. **OpenGL: Programming Guide**. 5. ed. Massachusetts : Addison-Wesley, 2005.

[7] Qt webpage. Disponível em: http://www.trolltech.com/products/qt

[8] GTK webpage. Disponível em: http://www.gtk.org/

[9] wxWidget webpage. Disponível em: http://www.wxwidgets.org/

[10] Fox Webpage. Disponível em: http://www.fox-toolkit.org/

[11] http://www.cs.queensu.ca/~jstewart/454/notes/pipeline/

[12] Celes, W. Notas de Aula. PUC-Rio, 2006.