OpenGL – Um tutorial

Luis Valente

Instituto de Computação - Universidade Federal Fluminense
lvalente@ic.uff.br

Dezembro, 2004

Resumo

OpenGL é uma biblioteca para modelagem e visualização tridimensional em tempo real, que está disponível para vários sistemas operacionais. Desenvolvida inicialmente pela Silicon Graphics em 1992, hoje é um dos padrões da indústria.

Este trabalho é uma introdução a conceitos básicos do OpenGL, desde a configuração inicial até mapeamento por textura. Assume-se que o leitor possui conhecimento de linguagem C e saiba utilizar algum compilador ou ambiente de desenvolvimento C (como o GCC ou Microsoft Visual C++). O ambiente de desenvolvimento usado como referência é o Microsoft Visual C++, mas os conceitos apresentados são independentes do ambiente e sistema operacional.

Abstract

OpenGL is an API applied in 3D modeling and real-time 3D visualization, which is available for many operating systems. The development of OpenGL was started by Silicon Graphics in 1992, and today it's considered an industry standard.

This work introduces basic OpenGL concepts, from instalation to texture mapping. It's assumed that the reader has knowledged of the C language and is comfortable in using C compilers and/or development environments for C (like GCC or Microsoft Visual C++). Microsoft Visual C++ was adopted as the reference development environment in this work, although the concepts presented here are compiler and operating system independent.

1. Introdução

OpenGL (*Open Graphics Library*) é uma biblioteca (API[1]) para Computação Gráfica 3D e 2D que foi desenvolvida inicialmente por Silicon Graphics Inc. (SGI). Atualmente, é adminstrada pela ARB (*Architecture Review Board*)[2]. A ARB é uma entidade que congrega vários representantes da indústria, como SGI, Intel, NVIDIA e Sun Microsystems.

A API do OpenGL oferece algumas primitivas básicas (como pontos, linhas, triângulos, quadriláteros e polígonos), operações para manipulação do sistema de coordenadas e operações com matrizes (translação, rotação e escala) e efeitos como mapeamento de textura, entre outros comandos.

OpenGL foi projetado para ser utilizado em várias plataformas e sistemas operacionais, como Mac OS, OS/2, Unix, Windows, Linux, OPENStep e BeOS. Pode ser utilizado em diversos ambientes de desenvolvimento de linguagens de programação como Delphi, Visual Basic, C/C++, Java, Fortran e Python, entre outros. Devido a esse propósito, existem várias funcionalidades que não estão disponíveis em OpenGL, porque são dependentes da plataforma nativa. Entre essas funcionalidades, encontram-se janelas (e interface gráfica), fontes para texto e comandos para interação com o usuário (como processamento de eventos do teclado e *mouse*).

A versão atual do OpenGL é 1.5[3]. A biblioteca possui dois componentes principais:

- GL: Representa o núcleo do OpenGL, que agrega as funções principais.
- GLU: É uma biblioteca utilitária, que possui funções diversas para *quadrics*, NURBS e matrizes, entre outras.

Uma característica importante de OpenGL é que o OpenGL é uma API de modo imediato (*immediate mode*). Em APIs de modo imediato, os comandos submetidos alteram o estado do *hardware* gráfico assim que são recebidos. O estado do *hardware* gráfico representa um tipo de configuração que está em uso no momento, e como essa configuração é aplicada internamente pelo *hardware*. Por exemplo, uma aplicação pode utilizar iluminação em uma cena. Desta forma, diz-se que o estado de iluminação está "ligado". Quando esse estado é ligado, o *hardware* é configurado internamente para realizar uma série de operações que têm a ver com essa funcionalidade.

2. Usando OpenGL

No sistema operacional Windows, o OpenGL já está pronto para uso (em modo simulado, por *software*, pelo menos). A implementação da biblioteca está contida em arquivos DLL (opengl32.dll e glu32.dll) no Windows e em arquivos .so (libGL.so, libGLU.so) no Linux.

Para utilizar plenamente os recursos oferecidos pelo *hardware* com o OpenGL, é preciso instalar o *driver* específico da placa gráfica. Esses *drivers* podem ser encontrados nos *sites* dos fabricantes dos *chipsets* da placa gráfica (como NVIDIA[4] ou ATI[5]). No Linux, uma das soluções é instalar a Mesa3D[6], que é uma implementação não-oficial da API do OpenGL. Alguns fabricantes (como a NVIDIA) oferecem drivers nativos para Linux, que podem ser obtidos no *site* da empresa.

1 *Application Programming Interface*
2 http://www.opengl.org/about/arb/
3 A versão 2.0 já está sendo desenvolvida.
4 http://www.nvidia.com
5 http://www.ati.com
6 http://www.mesa3d.org

2.1. Configuração inicial

Esta seção contém instruções de configuração do ambiente de desenvolvimento para criar aplicações com OpenGL.

Grande parte dos compiladores C e C++ disponíveis já possuem os arquivos necessários para se desenvolver programas com OpenGL. Entretanto, a versão do OpenGL que pode ser usada através desses arquivos pode variar bastante. A versão que vêm com o Visual C++ é 1.1, enquanto o MinGW (GCC para Windows) possui a versão 1.3.

2.2. Visual C++

Para usar o OpenGL com o Visual C++, é preciso incluir os seguintes arquivos nos programas:

```
#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>
```

O arquivo gl.h possui as funções principais do OpenGL, enquanto o glu.h possui as funções da biblioteca utilitária.

Os arquivos de ligação com as DLLs são dois: opengl32.lib e glu32.lib (caso se use a GLU). Esses arquivos podem ser incluídos nos projetos da seguinte forma:

- Visual C++ 6: Acesse o menu “Projects” e depois “Settings”. A seguir, acesse a aba “Linker”. No campo “Object/library modules” acrescente “opengl32.lib glu32.lib” (sem aspas).
- Visual C++ 7 (.NET): Acesse o menu “Project” e depois “Properties”. No painel da esquerda, escolha a pasta “Linker” e depois “Input”. No campo “Additional dependencies”, acrescente “opengl32.lib glu32.lib” (sem aspas).

É necessário incluir o arquivo principal do Windows (windows.h), antes dos arquivos do OpenGL ou erros de compilação serão gerados.

2.3. GCC

Para usar o OpenGL com o GCC, é preciso incluir os seguintes arquivos nos programas:

```
#include <GL/gl.h>
#include <GL/glu.h>
```

O arquivo gl.h possui as funções principais do OpenGL, enquanto o glu.h possui as funções da biblioteca utilitária.

Os arquivos de ligação com as DLLs podem ser especificados através dos parâmetros -lopengl32 e -lglu32 (caso se use a GLU), tanto em Windows quanto em Linux.

2.4. Convenções

Todos os comandos do OpenGL seguem uma convenção, que pode ser observada na Figura 2.1.

Figura 2.1 Comando típico do OpenGL (WrSw99)

O comando retratado na figura possui as seguintes características:

- Um prefixo que indica de qual parte da API o comando pertence. Alguns valores possíveis são gl (funções principais), glu (biblioteca utilitária), wgl (funções específicas para Windows), glx (funções específicas para Unix/X11) e agl (funções específicas para Mac OS).
- O nome "principal" do comando. O exemplo da figura representa um comando para se alterar a cor atual.
- O número de argumentos aceitos pela função. A função do exemplo aceita 3 argumentos.
- O tipo dos argumentos. No exemplo, o tipo dos argumentos é ponto flutuante. Dessa forma, a função aceita 3 números de ponto flutuante.

Como um dos objetivos do OpenGL é ser independente de plataforma, são definidos vários tipos de dados, que são descritos na tabela 1.

Sufixo	**Tipo de dado**	**Tipo definido no OpenGL**	**Tipo correspondente na linguagem C**
b	inteiro 8 bits	GLbyte	signed char
s	inteiro 16 bits	GLshort	short
i	inteiro 32 bits	GLint, GLsizei	int, long
f	ponto flutuante 32 bits	GLfloat, GLclampf	float
d	ponto flutuante 64 bits	GLdouble, GLclampd	double
ub	inteiro 8 bits sem sinal	GLubyte, GLboolean	unsigned char
us	inteiro 16 bits sem sinal	GLushort	unsigned short
ui	inteiro 32 bits sem sinal	GLuint, GLenum, GLbitfield	unsigned int, unsigned long
v	array, é usado em conjunto com os outros		

Tabela 2.1 Tipos de dados do OpenGL

É comum que existam diversas variações de comandos, que diferem entre si pelo tipo e número de argumentos aceitos. Exemplos:

```
glColor3f (1.0f, 1.0f, 1.0f);

GLfloat color [3];
glColor3fv (color);
```

3. Sistemas de janelas

Devido à portabilidade do OpenGL, a API não possui funções para lidar com sistemas de janelas, cujas características dependem do sistema operacional usado. Esse é o primeiro problema a ser enfrentado pelo desenvolvedor, já que para usar o OpenGL é preciso abrir uma janela. Eventualmente, a aplicação terá que interagir com o usuário, quando será necessário tratar eventos relacionados com o teclado e o *mouse*.

O desenvolvedor possui duas opções:

- Usar a API específica do sistema operacional. Essa opção torna a aplicação dependente do sistema operacional. Entretanto, é possível utilizar características avançadas (e/ou otimizadas) relacionadas ao sistema sistemas. Obviamente, será preciso aprender a utilizar a API do sistema operacional escolhido.

- Usar algum *toolkit* ou biblioteca que ofereça uma abstração para o sistema de janelas. Existem vários exemplos de ferramentas desse tipo disponíveis na *internet*. Muitas delas estão disponíveis para vários sistemas operacionais e possuem o código aberto. Suas APIs geralmente são bem mais fáceis de aprender (e menos complexas) do que as APIs de sistemas operacionais específicos. Entretanto, por tentar oferecer características que sejam comuns a vários sistemas operacionais, podem ter algumas limitações em termos de funcionalidades.

A opção a se tomar depende dos objetivos da aplicação.

Como o objetivo deste trabalho é ensinar OpenGL somente, a segunda opção foi escolhida aqui. O *toolkit* usado neste trabalho é o freeglut[7].

O freeglut é uma biblioteca de código aberto que está disponível em diversas plataformas. Ela é originária do GLUT (*OpenGL utility toolkit*), que era o *toolkit* padrão para demonstrar conceitos sobre OpenGL. Dessa forma, é possível encontrar na *internet* vários programas com exemplos de utilização de OpenGL escritos com o GLUT. Entretanto, o GLUT não é mais desenvolvido (sua última versão é de 1998).

3.1. Instalação do freeglut

Esta seção assume que o compilador utilizado é o Visual C++. Inicialmente, é preciso ir no *site* (http://freeglut.sourceforge.net/) do freeglut e baixar o código fonte.

7 http://freeglut.sourceforge.net/

3.1.1. Compilação

A seguir, é preciso compilar o código fonte para gerar as DLLs e arquivos .lib. Para isso, descompacte o arquivo para um diretório qualquer (ex: c:\freeglut). Depois, basta abrir o arquivo do workspace (freeglut.dsw). Esse arquivo de projeto foi construído para ser usado com o Visual C++ 6, mas pode ser utilizado pelo Visual C++ .NET sem problemas.

Nesse workspace, constam dois projetos: o freeglut_static e o freeglut (que está selecionado por padrão). O primeiro é usada para se construir uma biblioteca estática e o outro para se construir a DLL. Neste trabalho, será usada a versão em DLL.

Para compilar o projeto, é preciso acessar o menu "Build" e depois "Build freeglut.dll", caso o compilador seja o Visual C++ 6. Se for o Visual C++ .NET, os menus são "Build" e depois "Build freeglut".

Essas instruções geram a versão debug da DLL. Caso se queira gerar a versão release (recomendado), basta acessar os menus "Build", "Set active configuration" e escolher "freeglut – Win32 Release", no Visual C++ 6. No Visual C++ .NET, o caminho é "Build", "Configuration Manager ..." e escolher "Release" em "Active Solution Configuraration". A seguir é preciso construir a DLL como explicado anteriormente.

3.1.2. Configuração

Após a construção da DLL, é preciso configurar o Visual C++ para que seja possível usar o freeglut. Isso pode ser feito com os passos descritos aqui (supondo que o diretório do freeglut seja c:\freeglut).

3.1.2.1. Visual C++ 6

- Acrescentar o diretório "c:\freeglut\include" no Visual C++. Seguir pelos menus "Tools", "Options" e escolher a aba "Directories". Em "Show directories for", escolher "include files". Criar uma nova entrada e acrescentar o nome do diretório.

- Copiar a .lib gerada para um diretório "c:\freeglut\lib" (esse passo não é estritamente necessário, mas conveniente).

- Acrescentar o diretório onde está o freeglut.lib (ex: "c:\freeglut\lib") no Visual C++. Seguir pelos menus "Tools", "Options" e escolher a aba "Directories". Em "Show directories for", escolher "library files". Criar uma nova entrada e acrescentar "c:\freeglut\lib" (ou outro diretório onde esteja o arquivo).

3.1.2.2. Visual C++ .NET

- Acrescentar o diretório "c:\freeglut\include" no Visual C++. Seguir pelos menus "Tools", "Options" e escolher "Projects" no painel da esquerda. A seguir, escolher "VC++ Directories" e em "Show directories for", escolher "Include files". Criar uma nova entrada e acrescentar o nome do diretório.

- Copiar a .lib gerada para um diretório "c:\freeglut\lib" (esse passo não é estritamente necessário, mas conveniente).

- Acrescentar o diretório onde está o freeglut.lib (ex: "c:\freeglut\lib") no Visual C++. Seguir pelos menus "Tools", "Options" e escolher "Projects" no painel da esquerda. A seguir,

escolher “VC++ Directories” e em “Show directories for”, escolher “library files”. Criar uma nova entrada e acrescentar “c:\freeglut\lib” (ou outro diretório onde esteja o arquivo).

3.1.3. Outros compiladores

Para usar o freeglut com outros compiladores, será necessário compilar o código e gerar os arquivos de ligação com o compilador em questão. O código fonte do freeglut já vem com *makefiles* que podem ser usados para compilar o código com o GCC.

Existem versões pré-compiladas do freeglut para o GCC do Linux e Windows (MinGW). Essas versões podem ser obtidas em http://jumpgate.homelinux.net/random/freeglut-fedora/ (pacote RPM para Linux) e http://www.nigels.com/glt/devpak/ (Windows, pacote DevPak para o DevC++[8]).

3.2. Exemplo de uso

O exemplo demonstra um programa simples com freeglut e serve para testar a instalação. Aqui está o exemplo:

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

void OnDisplay ()
 {
   glClearColor (0.0f, 0.0f, 0.25f, 1.0f);

   glClear (GL_COLOR_BUFFER_BIT);

   glFlush ();
 }

int main (int argc, char * argv [])
 {
   glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
   glutCreateWindow (“Exemplo 1”);

   glutDisplayFunc (OnDisplay);

   glutMainLoop ();

   return 0;
 }
```

3.2.1. Arquivos de inclusão

O freeglut possui um arquivo de inclusão principal:

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

8 O DevC++ é uma IDE (Integrated Development Environment) para o GCC do Windows. É gratuita e está disponível em http://www.bloodshed.net/dev/devcpp.html

Esse arquivo permite utilizar todas as funções do freeglut. Esse arquivo já inclui automaticamente os arquivos principais do OpenGL (o gl.h e o glu.h). Dessa forma, não é preciso especificá-los. Não é necessário incluir o arquivo principal do Windows (windows.h).

3.2.2. Função principal

A função principal `main` é o ponto de entrada do programa. Na primeira linha, têm-se:

```
glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
```

Todas as funções do freeglut possuem o prefixo glut, seguindo a convenção adotada pelo OpenGL. A função da primeira linha especifica uma configuração para o OpenGL. Essa configuração é definida por uma combinação de várias *flags*. A configuração do exemplo requisita que seja usado um único *buffer* para a janela e o modo de cores usado seja RGB. No modo `GLUT_SINGLE` (*single-buffered*), todas as operações de desenho são executadas diretamente na janela da aplicação. Esse modo não é adequado para animação, porque o usuário percebe quando a imagem é apagada e desenhada "ao mesmo tempo". Esse problema é conhecido como *flicker*. A alternativa é usar o modo *double-buffered*, que cria dois *buffers*. Nessa configuração, o conteúdo de um dos *buffers* está visível enquanto o outro permanece invisível. Os comandos de desenho são executados no *buffer* que está oculto, e no final da operação, o que está visível torna-se invisíve, e vice-versa. Dessa forma, a animação resultante é suave e sem *flicker*. A constante para esse modo é `GLUT_DOUBLE`.

A próxima linha da função contém:

```
glutCreateWindow ("Exemplo 1");
```

Essa função é responsável por criar a janela. O argumento correponde ao texto que aparecerá na barra de título da janela.

A próxima linha demonstra um conceito importante no *freeglut*:

```
glutDisplayFunc (OnDisplay);
```

Essa função indica ao freeglut que a função `OnDisplay` será usada para desenhar na janela. Essa função é um exemplo das funções *callback* do freeglut. Uma função *callback* é uma função definida pelo usuário e registrada no freeglut, para ser executada na ocorrência de algum evento específico. Nesse exemplo, `OnDisplay` será executada toda vez que for necessário redesenhar a janela.

Existem diversos tipos de eventos que podem ser tratatos através de funções *callback*. Para cada tipo de evento que pode ser tratado, existe uma função específica responsável por registrar uma função *callback*.

A seguir, têm-se a seguinte linha de código:

```
glutMainLoop ();
```

Esse comando requisita que o freeglut inicie o laço principal da aplicação. Nesse laço, ocorre o processamento dos eventos da aplicação (que podem ser tratados através de *callbacks*). O laço principal da aplicação é executado até que seja requisitado o término do programa.

3.2.3. Comandos do OpenGL

A função `OnDisplay` possui vários comandos do OpenGL que serão analisados a seguir. Na primeira linha, têm-se:

```
glClearColor (0.0f, 0.0f, 0.25f, 1.0f);
```

Esse comando estabelece a cor de fundo da janela. No exemplo, a cor de fundo será uma variação de azul. As cores em OpenGL (como será explicado em outras seções) são especificadas no formato RGBA. Nas funções que aceitam argumentos de ponto flutuante, o valor de cada componente varia de 0.0 (ausência) a 1.0 (intensidade máxima).

Após definir a cor de fundo, a tela é preenchida com a cor de fundo (ou seja, é limpa) com o seguinte comando:

```
glClear (GL_COLOR_BUFFER_BIT);
```

O argumento da função indica qual é o tipo de buffer que será afetado. Em OpenGL, um *buffer* é uma área de memória com algumas propriedades especiais. A constante `GL_COLOR_BUFFER_BIT` corresponde à area de memória que contém os *pixels* da janela (ou seja, "a tela"). Existem diversos tipos de *buffers* em OpenGL.

A última linha da função `OnDisplay` é listada a seguir:

```
glFlush ();
```

Esse comando requisita ao OpenGL que execute todos os comandos pendentes. É comum que o OpenGL agrupe vários comandos (internamente, sem conhecimento do usuário) para que sejam executados de uma única vez. O objetivo disso é melhorar o desempenho da aplicação.

4. Primitivas

Uma primitiva é um objeto simples que pode ser usado para a criação de outros objetos complexos. As primitivas disponíveis no OpenGL estão resumidas na figura 4.1.

A especificação de primtivas no OpenGL segue um padrão, demonstrado no seguinte trecho de código:

```
glBegin (nome da primitiva)

  ...  // comandos que especificam os componentes da primitiva

glEnd ();
```

O nome da primitiva é uma constante que indica qual é o tipo de primitiva usado. Essas constantes estão presentes na figura 4.1.

Para se especificar os vértices da primitiva, o seguinte comando é usado:

```
glVertex* ()
```

O * é usado aqui para indicar que existem diversas variações do comando, por exemplo:

```
glVertex3f (10.0f, 40.0f, 0.0f);         // x, y, z, 1
glVertex3i (5, 5, 5);                    // x, y, z, 1
glVertex2d (20.0, 10.0);                 // x, y, 0, 1
glVertex4f (0.0f, 30.0f, 10.0f, 1.0f);   // x, y, z, w
```

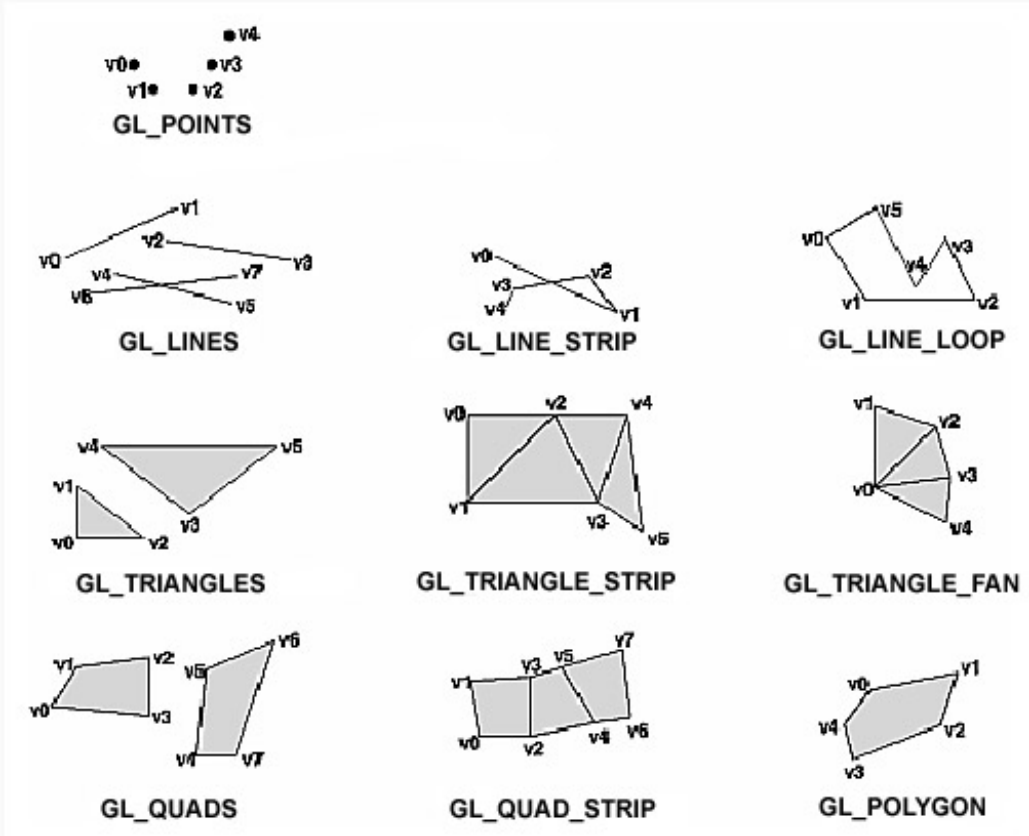

Figura 4.1 Primitivas do OpenGL (NeDa97)

4.1. Pontos

Para especificar pontos simples, é necessário usar a constante `GL_POINTS`.

```
glBegin (GL_POINTS);

  glVertex3f (0.0f, 0.0f, 0.0f);
  glVertex3f (10.0f, 20.0f, -3.0f);
  glVertex3f (4.0f, -10.0f, 1.0f);

glEnd ();
```

O exemplo desenha três pontos na janela. Por padrão, o OpenGL desenha os pontos com tamanho igual a 1 *pixel*. Entretanto, é possível desenhar pontos maiores. Para isso, é preciso usar o seguinte comando:

```
glPointSize (GLfloat tamanho);
```

O maior valor possível depende do *hardware* onde o programa será executado. O único valor garantido a existir é 1.0 (que corresponde a um *pixel*). Para consultar a faixa de valores possíveis, basta usar o seguinte trecho de código:

```
GLfloat faixa [2];
glGetFloatv (GL_POINT_SIZE_RANGE, faixa);

GLfloat min = faixa [0];
GLfloat max = faixa [1];
```

A faixa de valores possíveis para o tamanho dos pontos é apenas uma das propriedades do OpenGL que podem ser consultadas. Existem diversas outras, que podem ser obtidas usando funções como `glGetFloatv` , `glGetIntegerv`, `glGetDoublev` e `glGetBooleanv`.

Outra propriedade interessante que pode ser alterada é a suavização no desenho dos pontos (*anti-aliasing*). O comando para especificá-la é:

```
glEnable (GL_POINT_SMOOTH);
```

O comando `glEnable` é responável por habilitar alguma propriedade do OpenGL (ou seja, "ligar" o estado). O comando correspondente, para desligar, é `glDisable`. Dessa forma, para desligar a suavização no desenho dos pontos, basta usar este comando:

```
glDisable (GL_POINT_SMOOTH);
```

4.2. Linhas

Existem três primitivas relacionadas a linhas em OpenGL que são `GL_LINE`, `GL_LINE_STRIP` e `GL_LINE_LOOP`.

A primitiva `GL_LINE` corresponde a linhas simples:

```
glBegin (GL_LINE)

  glVertex3f (0.0f, 0.0f, 0.0f);
  glVertex3f (10.0f, 10.0f, 5.0f);
  glVertex3f (5.0f, -20.0f, 0.0f);   // descartado

glEnd ();
```

Quando essa primitiva é usada, cada par de vértices é usado para desenhar uma linha. Caso exista um número ímpar de vértices, o último vértice é descartado.

Na primitiva `GL_LINE_STRIP`, os vértices especificados são conectados em sequência, conforme são especificados:

```
glBegin (GL_LINE_STRIP)

  glVertex3f (0.0f, 0.0f, 0.0f);
  glVertex3f (10.0f, 10.0f, 5.0f);
  glVertex3f (5.0f, -20.0f, 0.0f);

glEnd ();
```

Nesse exemplo, existirão dois segmentos de reta.

A primitiva `GL_LINE_LOOP` é semelhante à primitiva `GL_LINE_STRIP`, só que o último vértice especificado é conectado ao primeiro, formando um ciclo.

Por padrão, as linhas desenhadas possuem a espessura igual a 1 *pixel*. Essa propriedade pode ser alterada com o seguinte comando:

```
glLineWidth (GLfloat valor);
```

O único valor garantido a existir é 1.0, para consultar a faixa de valores permitida, o procedimento é semelhante àquele usado para consultar a faixa de valores para os pontos:

```
GLfloat faixa [2];
glGetFloatv (GL_LINE_WIDTH_RANGE, faixa);

GLfloat min = faixa [0];
GLfloat max = faixa [1];
```

Também é possível desenhar as linhas com *anti-aliasing*, usando o comando:

```
glEnable (GL_LINE_SMOOTH);
```

4.3. Polígonos

O OpenGL permite usar três tipos de polígonos: triângulos, quadriláteros e polígonos genéricos. Comum a todos eles, existe uma propriedade importante: a ordenação dos vértices (a ordem em que são especificados) deve ser consistente. As ordenações possíveis são ilustradas na figura 4.2.

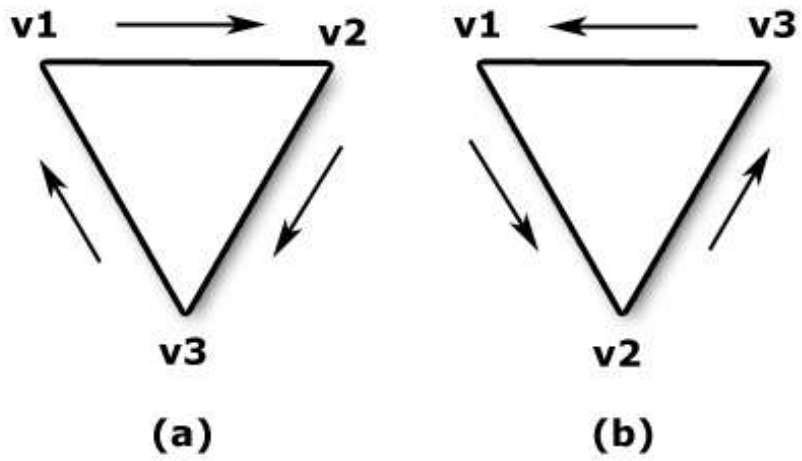

Figura 4.2 Ordenações possíveis. (a) Sentido horário. (b) Sentido anti-horário.

Essa propriedade é importante porque a ordenação dos vértices define o lado do polígono que está "voltado para a frente" e o lado do polígono que está "voltado para trás". O padrão usado pelo OpenGL (e neste trabalho) é que as faces voltadas para frente são especificadas em sentido anti-horário. Esse padrão pode ser alterado com o seguinte comando:

```
glFrontFace (ordem);
```

A ordem pode ser `GL_CW` (sentido horário) ou `GL_CCW` (sentido anti-horário).

Para entender porque a importância dessa especificação, considere o seguinte exemplo: Imagine que em um programa, o usuário visualize um cubo. O cubo é formado por seis polígonos, mas na verdade existem doze faces: seis que estão voltadas para frente e seis que estão voltadas para trás (porque cada polígono possui duas faces). O OpenGL, por padrão, processa as doze faces.

O processo de eliminação das faces que não podem ser vistas pelo usuário (as que estão no interior do cubo, supondo que o usuário esteja for a dele) é conhecido como *backface culling*. Para habilitar o *backface culling*, é necessário usar esse comando:

```
glEnable (GL_CULL_FACE);
```

Os polígonos podem ser desenhados de diversas maneiras: com preenchimento de cor (padrão), em *wireframe* ou só com os vértices.

Para alterar o modo de renderização dos polígonos, usa-se o seguinte comando:

```
glPolygonMode (tipo de face, modo);
```

Os tipos de face aceitos são: `GL_FRONT` (afeta as faces da frente), `GL_BACK` (afeta as faces de trás) e `GL_FRONT_AND_BACK` (afeta ambas). As constantes para os modos de renderização são `GL_FILL` (sólido), `GL_LINE` (*wireframe*) e `GL_POINT` (somente vértices). Dessa forma, para desenhar as faces da frente dos polígonos em modo *wireframe*, basta usar este comando:

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

É importante observar que o modo especificado afeta apenas o tipo de face desejado. Por exemplo, caso se queira desenhar as faces da frente em modo sólido e as de trás como *wireframe*, estes comandos seriam usados (para que ambas sejam visualizadas, o *backface culling* teria que estar desligado):

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

4.3.1. Triângulos

Os triangulos são o tipo de primitiva mais usado em OpenGL (e em gráficos 3D, em geral). No OpenGL, existem três variações sobre o tema: `GL_TRIANGLES`, `GL_TRIANGLE_STRIP` e `GL_TRIANGLE_FAN`.

A primitiva `GL_TRIANGLE` permite desenhar triângulos simples:

```
glBegin (GL_TRIANGLES)

  glVertex2f (10.0f, 10.0f);
  glVertex3f (20.0f, -5.0f);
  glVertex3f (-20.0f, -5.0f);

  glVertex3f (100.0f, 0.0f);    // descartado
  glVertex3f (100.0f, -8.0f);   // descartado

glEnd ();
```

Quando essa primitiva é usada, o OpenGL usada cada trinca de vértices para desenhar um triângulo. Caso exista uma trinca incompleta, esta é descartada.

A primitiva `GL_TRIANGLE_STRIP` é usada para se desenhar uma tira de triângulos (triângulos conectados sequencialmente). Esse esquema pode ser visualizado na figura 4.3.

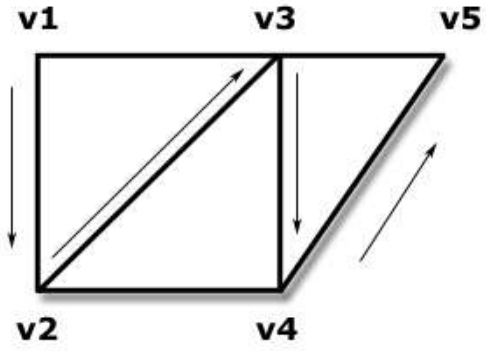

Figura 4.3 Sequência de especificação de vértices para GL_TRIANGLE_STRIP

Os três primeiros vértices são usados para formar o primeiro triângulo. A seguir, cada vértice subsequente é conectado aos dois vértices anteriores. Na figura 4.3, a ordenação dos vértices segue o sentido anti-horário.

A primitiva `GL_TRIANGLE_FAN` é usada para se conectar vários triângulos em torno de um ponto central, tal como um leque ("fan"):

```
glBegin (GL_TRIANGLE_FAN);

  glVertex3f (0.0f, 0.0f, 0.0f);

  glVertex3f (10.0f, -1.0f, 0.0f);
  glVertex3f (10.0f, 9.0f, 0.0f);
  glVertex3f ( 0.0f, 12.0f, 0.0f);
  glVertex3f (-10.0f, 8.0f, 0.0f);

glEnd ();
```

O primeiro vértice especificado na primitiva é o ponto central do leque. Os dois vértices seguintes formam o primeiro triângulo. A partir de então, é preciso somente especificar um novo vértice para formar um novo triângulo (o triângulo será construído usando o novo vértice e dois vértices especificados antes deste).

A figura 4.4. ilustra um leque formado por essa primitiva.

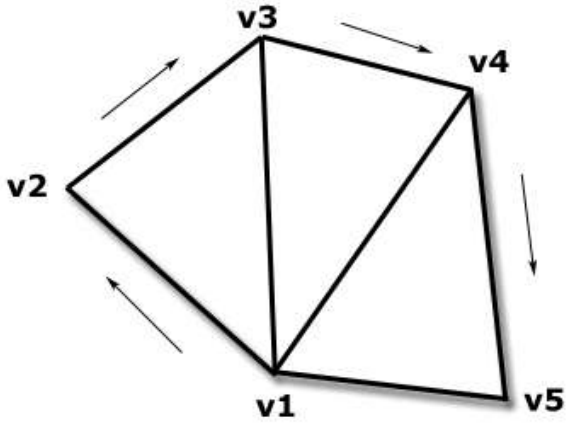

Figura 4.4 Malha gerada por GL_TRIANGLE_FAN

4.3.2. Quadriláteros

As primitivas para quadriláteros em OpenGL são duas: `GL_QUADS` e `GL_QUAD_STRIP`.

O uso de `GL_QUADS` possibilita que sejam gerados quadriláteros simples. O OpenGL utiliza quatro vértices de cada vez para formar um quadrilátero, como neste exemplo:

```
glBegin (GL_QUADS);

  glVertex3f (-10.0f, -10.0f, 0.0f);
  glVertex3f (10.0f, -10.0f, 0.0f);
  glVertex3f (10.0f, 10.0f, 0.0f);
  glVertex3f (-10.0f, 10.0f, 0.0f);

  glVertex3f (-30.0f, -10.0f, 0.0f);  // descartado
  glVertex3f (30.0f, -10.0f, 0.0f);   // descartado

glEnd ();
```

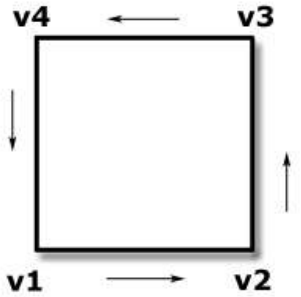

Figura 4.5 GL_QUADS

Caso não existam vértices suficientes para formar um quadrilátero, eles são descartados.

A primitiva `GL_QUAD_STRIP` serve para formar tiras de quadriláteros conectados. Seu modo de uso é quase idêntico ao de `GL_TRIANGLE_STRIP`. Em `GL_QUAD_STRIP` é preciso especificar quatro vértices iniciais (para formar o primeiro quadrado) e mais dois vértices de cada vez para formar os outros quadrados. O uso dessa primitiva é ilustrado na figura 4.6.

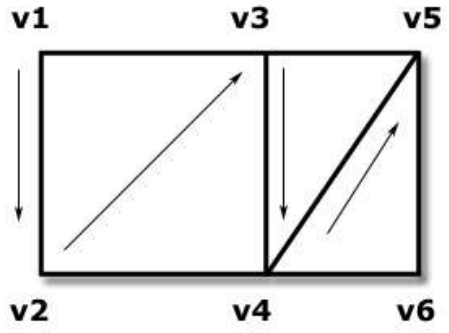

Figura 4.6 Aplicação de GL_QUAD_STRIP

4.3.3. Polígonos genéricos

A construção de polígonos genéricos pode ser feita usando a primitiva `GL_POLYGON`. Não há limites para o número de vértices, mas existem duas regras para o uso de `GL_POLYGON`.

A primeira delas é que o polígono a ser gerado deve pertencer a um único plano (ou seja, todos os vértices devem estar no mesmo plano). A segunda regra é que os polígonos devem ser convexos.

5. Cores e sombreamento

Cores em OpenGL são especificadas como RGBA. O comando para alterar as cores é:

```
glColor* ()
```

Assim como vários comandos, o `glColor` possui diversas variações. Nas versões que aceitam argumentos de ponto flutuante, os valores para cada componente variam entre 0.0 (ausência) e 1.0 (intensidade máxima). Exemplos de uso:

```
glColor3f (1.0f, 1.0f, 1.0f);  // branco
glColor3f (0.0f, 0.0f, 0.0f);  // preto
glColor3f (1.0f, 0.0f, 0.0f);  // vermelho puro
```

Nas variações que trabalham com números inteiros sem sinal, os valores dos componentes variam de 0 (ausência) a 255 (intensidade máxima).

O OpenGL trabalha internamente com RGBA, de forma que se forem especificadas apenas três componntes, o quarto componente (alfa) será 1.0 por padrão.

```
glColor3f (0.0f, 1.0f, 0.0f);       //  verde , alfa = 1.0
glColor4f (1.0f, 1.0f, 0.0f, 0.25f) // amarelo, alfa = 0.25
```

Quando um comando de cor é usado, a cor especificada será usada para todas as primitivas até que uma outra cor seja especificada. Segue um pequeno exemplo:

```
glColor3f (1.0f, 0.0f, 0.0f);

glBegin (GL_TRIANGLES);

  glVertex3f (  0.0f, 10.0f, 5.0f);
  glVertex3f (-10.0f,  0.0f, 5.0f);
  glVertex3f ( 10.0f,  0.0f, 5.0f);

glEnd ();
```

Nesse exemplo, o triângulo será preenchido com a cor vermelha. Entretanto, as cores também podem ser especificadas para cada vértice, como no próximo exemplo:

```
glBegin (GL_TRIANGLES);

  glColor3f  (1.0f, 0.0f, 0.0f);
  glVertex3f (0.0f, 10.0f, 5.0f);

  glColor3f  (0.0f, 1.0f, 0.0f);
  glVertex3f (-10.0f, 0.0f, 5.0f);

  glColor3f  (0.0f, 0.0f, 1.0f);
  glVertex3f (10.0f, 0.0f, 5.0f);

glEnd ();
```

Para entender o que acontece nesse exemplo, é preciso explicar o conceito de modo de sombreamento.

O modo de sombreamento do OpenGL indica como será feita a coloração das primitivas (e preenchimento dos polígonos. Existem dois modos possíveis: o sombreamento suave (Gouraud) e sombreamento fixo. No sombreamento fixo, uma única cor é usada para preencher o polígono. No caso de sombreamento suave, o OpenGL irá realizar uma interpolação entre as cores dos vértices para preencher os polígonos.

O resultado desses dois modos (aplicados no exemplo) pode ser conferido na figura 5.1.

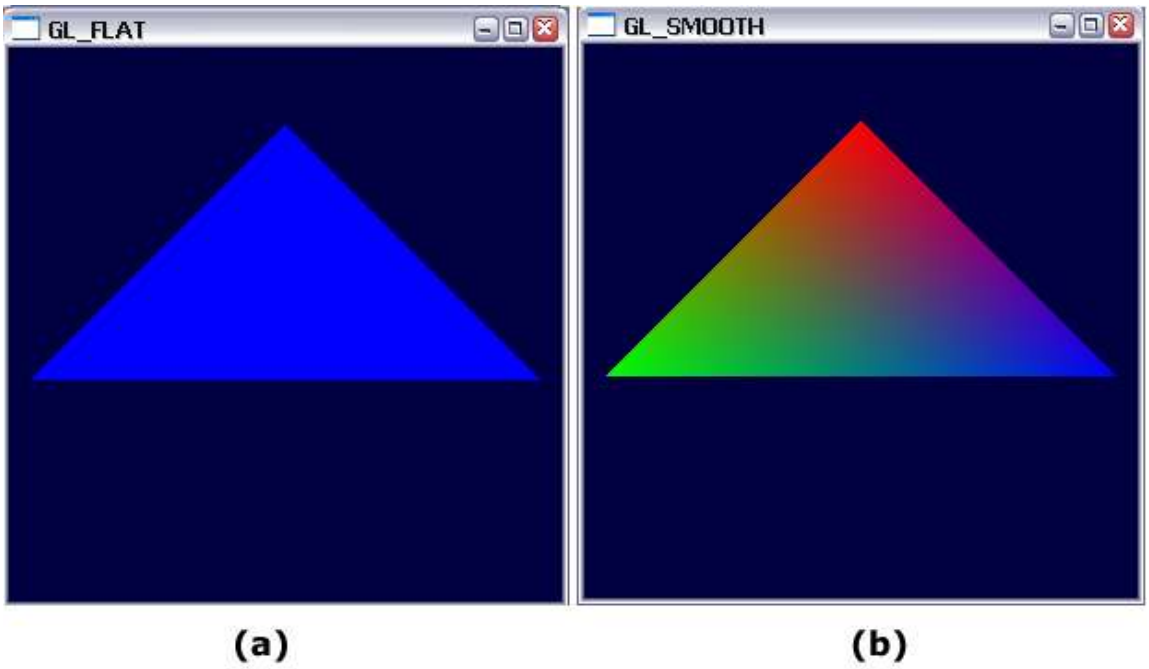

Figura 5.1 Modos de sombreamento. (a) Sombreamento fixo. (b) Sombreamento suave.

Para determinar o modo de sombreamento, o seguinte comando é usado:

```
glShadeModel (modo);
```

O modo de sombreamento suave corresponde à constante `GL_SMOOTH` e o modo de sombreamento fixo corresponde à constante (`GL_FLAT`). O modo padrão usado pelo OpenGL é `GL_SMOOTH`.

6. Z-Buffer

O OpenGL usa o *z-buffer* para determinar a visibilidade dos objetos desenhados na cena. Esse teste ocorre na etapa de rasterização.

O *z-buffer* é um dos vários *buffers* que existem no OpenGL. Tipicamente, a implementação do *z-buffer* é feita no *hardware* da placa gráfica. O *z-buffer* armazena valores que têm a ver com a profundidade dos pixels da tela. Dessa forma, o *z-buffer* possui o mesmo tamanho da janela ou tela da aplicação.

De maneira simplificada, o algoritmo de *z-buffer* é usado da seguinte maneira: no início da aplicação (ou antes de imagem ser construída, por exemplo, no início de um quadro de animação do programa), o *z-buffer* é preenchido ("limpo") com um certo valor (um valor muito grande, por exemplo). Quando um *pixel* precisa ser pintado na tela, o algoritmo consulta o valor correspondente a esse *pixel* que está armazenado no *z-buffer*. Se o valor for menor (o que quer dizer que o novo *pixel* estaria mais perto), o *pixel* é pintado na tela e o valor de profundidade é armazenado no *z-buffer*. Caso esse valor fosse maior do que aquele que lá estivesse, o *pixel* não seria pintado. A consequência da aplicação desse algoritmo é que a cena é desenhada corretamente (em termos de determinação de visibilidade).

Para se usar o *z-buffer* no OpenGL, é preciso requisitar sua criação. No freeglut, basta usar a constante `GLUT_DEPTH` junto com as outras na hora da inicialização:

```
glutInitDisplayMode (GLUT_DOUBLE | GLUT_DEPTH | GLUT_RGB);
```

A seguir, é preciso habilitar o uso do *z-buffer* no OpenGL (que está desligado por padrão). Isso pode ser feito com o seguinte comando:

```
glEnable (GL_DEPTH_TEST);
```

Caso se queira desligar o teste de *z-buffer*, basta usar o comando `glDisable (GL_DEPTH_TEST)`.

Durante o ciclo de vida da aplicação, o *z-buffer* deverá ser zerado toda vez que se desejar criar uma nova imagem. Isso é normalmente feito no mesmo momento em que se limpa a tela. Para realizar essa operação conjunta, basta usar este comando:

```
glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

O funcionamento do *z-buffer* e suas configurações podem ser alteradas com grande flexibilidade, mas esse detalhamento está fora do escopo deste trabalho. Para maiores informações, por favor, consulte a documentação do OpenGL.

7. Transformações e projeções

O OpenGL usa diversos tipos de transformações geométricas na especificação da cena. Essas transformações podem ser dos seguites tipos: *Viewport*, Modelagem, Projeção e Câmera.

O sistema de coordenadas tridimensional usado no OpenGL é o sistema de coordenadas da mão direita, que pode ser visualizado na figura 7.1.

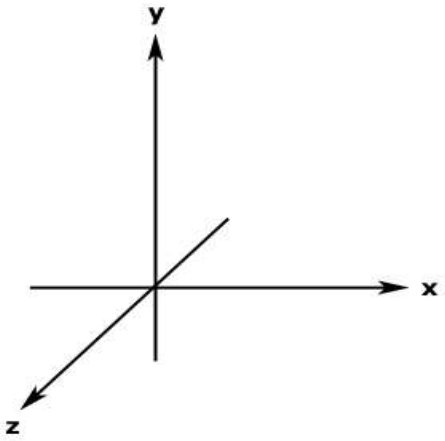

Figura 7.1 Sistema de coordenadas da mão direita

7.1. Viewport

A transformação de *viewport* indica a área da janela onde serão realizadas as operações de desenho. A figura 7.2 ilustra um exemplo de *viewport*.

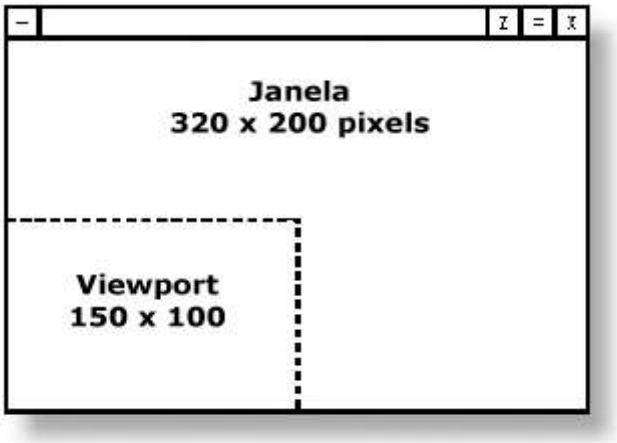

Figura 7.2 Uma viewport (WrSw99)

A origem da *viewport* é o canto inferior esquerdo. Para se determinar a *viewport*, o seguinte comando é usado:

```
glViewport (origem_x, origem_y, largura, altura);
```

Dessa forma, para especificar a *viewport* da figura 7.2, basta submeter este comando:

```
glViewport (0, 0, 150, 100);
```

7.2. Matrizes

O OpenGL usa matrizes para implementar todas as transformações existentes. Normalmente, o usuário não precisa lidar com as matrizes diretamente (embora isso seja possível se for desejado), pois existem vários comandos para manipulá-las.

As transformações de modelagem e projeção são independentes entre si. Internamente, o OpenGL mantém uma matriz para cada um desses tipos.

Em um dado momento, só é permitido usar um dos tipos de matriz. O tipo de matriz usado no momento é determinado pelo seguinte comando:

```
glMatrixMode (nome da matriz);
```

O nome da matriz de modelagem é `GL_MODELVIEW` e o da matriz de projeção é `GL_PROJECTION`. Segue um exemplo:

```
glMatrixMode (GL_PROJECTION);

// a partir deste momento, a matriz afetada é a de
// projeção

...  // comandos que afetam a matriz de projeção

glMatrixMode (GL_MODELVIEW);
```

```
// a partir deste momento, a matriz afetada é a de
// modelagem

...  // comandos que afetam a matriz de modelagem
```

Existem algumas operações pré-definidas que podem ser aplicadas em matrizes. Esses comandos afetam o tipo de matriz selecionado no momento. As operações principais são translação, rotação, escala e carregar a matriz identidade.

A matriz identidade serve para reiniciar o sistema de coordenadas, ou seja, a posição selecionada será a origem, não existirão rotações e a escala será igual a 1 em todos os eixos. O comando para carregar a matriz identidade é:

```
glLoadIdentity ();
```

A operação de translação desloca o sistema de coordenadas pelo número de unidades especificadas. O comando correspondente é:

```
glTranslate* (x, y, z);    // * = f ou d
```

Esse comando é acumulativo (assim como todos os outros), como é descrito neste exemplo:

```
glLoadIdentity ();                    // posição = 0,0,0

glTranslatef (10.0f, 0.0f, 0.0f);     // posição = 10,0,0
glTranslatef (0.0f, 5.0f, -1.0f);     // posição = 10,5,-1
```

A operação de rotação pode ser expressa com este comando:

```
glRotate* (angulo, x, y, z);   // * = f ou d
```

O ângulo é expresso em graus, e os parâmetros `x`, `y`, e `z` representam o eixo de rotação. Dessa forma, para realizar uma rotação de 30º em torno do eixo Z, este trecho de código é usado:

```
glRotatef (30, 0.0f, 0.0f, 1.0f);
```

A operação de escala é feita com o seguinte comando:

```
glScale* (escala_x, escala_y, escala_z);  // * = f ou d
```

Os parâmetros representam os fatores de escala em cada eixo. O valor 1.0 não afeta a escala.

7.3. Modelagem

Como exemplo, suponha que se queria desenhar um triângulo na posição (10, 10, 0). Isso pode ser feito com o seguite trecho de código:

```
glMatrixMode (GL_MODELVIEW);

glLoadIdentity ();
glTranslatef (10, 10, 0);

glBegin (GL_TRIANGLES);
```

```
  glVertex2f (0.0f, 10.0f);
  glVertex2f (-10.0f, 0.0f);
  glVertex2f (10.0f, 0.0f);

glEnd ();
```

Agora, suponha que se queira desenha o mesmo triângulo com uma rotação de -20º em torno do eixo Z:

```
glMatrixMode (GL_MODELVIEW);

glLoadIdentity ();
glRotatef (-20.0f, 0.0f, 0.0f, 1.0f);

glBegin (GL_TRIANGLES);

  glVertex2f (0.0f, 10.0f);
  glVertex2f (-10.0f, 0.0f);
  glVertex2f (10.0f, 0.0f);

glEnd ();
```

7.4. Projeção

O OpenGL possui funções para configurar dois tipos de projeções: projeções ortográficas e projeções em perspectiva.

7.4.1. Projeção ortográfica

A projeção ortográfica pode ser especificada com o seguinte comando:

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

             );
```

Os parâmetros correspondem aos valores dos planos que formam o volume de visão da projeção ortográfica. Esse volume de visão pode ser visualizado na figura 7.3.

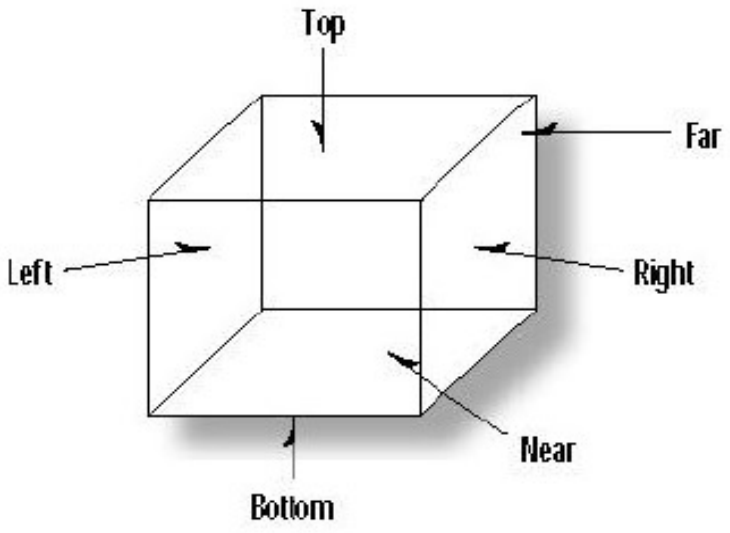

Figura 7.3 Volume de visão da projeção ortográfica (WrSw99)

O volume de visão da projeção ortográfica é um paralelepípedo (ou um cubo), de forma que todos os objetos (primitivas, etc) que estiverem for a desse volume não aparecerão na imagem final.

Por exemplo, suponha que se queria construir uma projeção que possua um comprimento de 20 unidades em todos os eixos, e que a origem esteja no centro da janela[9]. Isso pode ser feito com o seguinte trecho de código:

```
glMatrixMode (GL_PROJECTION);

  glLoadIdentity ();
  glOrtho (-10.0, 10.0, -10.0, 10.0, -10.0, 10.0);

glMatrixMode (GL_MODELVIEW);
  glLoadIdentity ();
```

No caso dos parâmetros `near` e `far`, valores negativos indicam que o plano está atrás do observador (o `near` é negativo porque o exemplo tem como objetivo posicionar a origem no centro do volume de visão).

O exemplo também demonstra uma convenção que é adotada em programas que usam OpenGL. A determinação da projeção é uma tarefa que é realizada poucas vezes. Na maioria dos casos, os comandos de matrizes afetam a matriz de modelagem. Dessa forma, a convenção é assumir sempre que a matriz atual é a de modelagem. Caso se queira alterar algum outro tipo de matriz, altera-se o tipo de matriz e realiza-se as operações desejadas. No final das operações, restaura-se o determina-se que o tipo de matriz é a de modelagem. O objetivo dessa convenção é evitar erros que podem ocorrer quando se altera a matriz errada (por exemplo, alterar a matriz de projeção acidentalmente).

9 Na verdade, que esteja no centro da *viewport*. Nesse exemplo, considera-se que a *viewport* ocupe toda a janela (o que é bastante comum).

Em relação a projeções ortográficas, existe ainda um outro tipo de problema a se tratar. No exemplo, foi criada uma projeção "quadrada". Entretanto, é comum que a janela da aplicação seja retangular. Dessa forma, tem-se o seguinte problema: Suponha que a projeção foi criada como no exemplo. Os eixos X, Y e Z podem representar valores de -20 a 20. Suponha que a janela da aplicação possui tamanho igual a 300x100. A proporção da janela (largura/altura) é diferente da proporção da projeção (que é 1). Sendo assim, as imagens desenhadas na janela serão desproporcionais, conforme pode ser visualizado na figura 7.4.

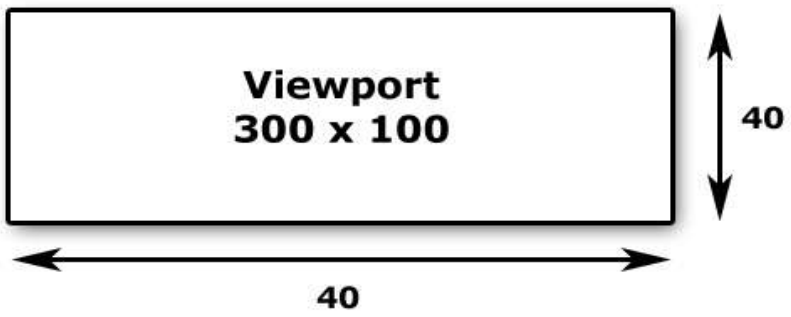

Figura 7.4 Viewport e projeção desproporcional

Para corrigir essa distorção, é necessário levar em consideração a razão de aspecto da janela ao se calcular a projeção. Isso pode ser feito da seguinte forma:

```
// largura e altura da viewport
largura = ...
altura  = ...

glMatrixMode (GL_PROJECTION);

  glLoadIdentity ();

  GLdouble proporcao = largura/altura;

  if (proporcao >= 1.0)
    glOrtho (left * proporcao, right * proporcao, bottom, top,
near, far);
  else
    glOrtho (left, right, bottom / proporcao, top / proporcao,
near, far);

glMatrixMode (GL_MODELVIEW);
  glLoadIdentity ();
```

7.4.2. Projeção em perspectiva

Para definir uma projeção em perspectiva, é necessário especificar os planos que formam o *frustum* (que é o volume de visão dessa projeção). O *frustum* é um volume tridimensional em forma de uma pirâmide imaginária limitada pelos planos delimitadores *near* e *far*, que correspondem a distâncias relativas ao observador, na direção de sua linha de visão. O *frustum* é ilustrado na figura 7.5.

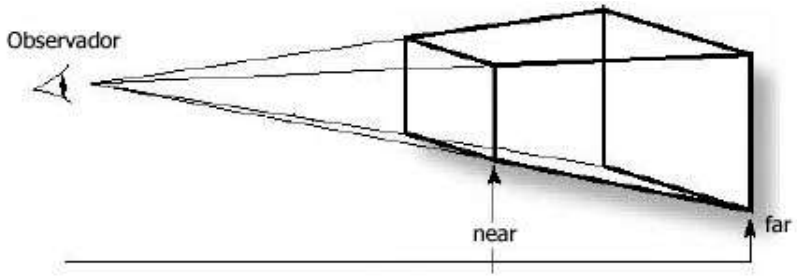

Figura 7.5 Frustum (WrSw99)

Para especificar o *frustum*, existe o seguinte comando do OpenGL:

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

Entretanto, especificar os planos do *frustum* não é conveniente na maioria das vezes. Por essa razão, a GLU oferece uma função para especificar o *frustum* de outra forma:

```
gluPerspective (campo de visão, proporção, near, far);
```

A figura 7.6 ilustra os parâmetros de `gluPerspective`:

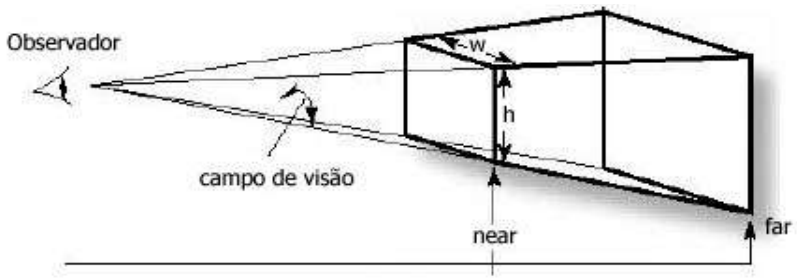

Figura 7.6 Parâmetros de gluPerspective (WrSw99)

O primeiro parâmetro corresponde ao campo de visão vertical (ângulo em graus). O segundo parâmetro corresponde à proporção entre a altura e largura da *viewport*. Os dois últimos parâmetros correspondem aos planos delimitadores *near* e *far*, como na outra função.

7.5. Câmera

A separação do conceito de transformação de câmera, em OpenGL, é apenas uma notação de conveniência. Não existe uma matriz separada para transformação de câmera, na verdade, essa transformação afeta a matriz de modelagem.

A transformação de câmera serve para posicionar e orientar o ponto de vista na cena. O OpenGL oferece uma função para especificar essa transformação:

```
gluLookAt (pos_x , pos_y , pos_z,
           alvo_x, alvo_y, alvo_z,
           up_x  , up_y  , up_z);
```

Os primeiros três parâmetros correspondem à posição do observador (ou da câmera) na cena. Os três seguintes correspondem ao ponto para onde a câmera está apontada. Os três últimos especificam um vetor que indica qual é a direção que pode ser considerada "para cima". Por exemplo, se o usuário está posicionado em (0, 0, 0) e olhando para (0, 0, -20), a direção "para cima" corresponde a (0, 1, 0).

A transformação de câmera deve ser a primeira a ser especificada na renderização da cena:

```
// supondo que o modo de matriz atual seja GL_MODELVIEW

void desenharCena ()
{
  glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  glLoadIdentity ();

  glLookAt (0.0, 0.0, 0.0,
            0.0, 0.0, -20.0,
            0.0, 1.0, 0.0);

  // comandos para posicionar, rotacionar, desenhar, etc
  ...
}
```

Se nenhuma transformação de câmera for especificada, o usuário estará posicionado em (0, 0, 0) e olhando para (0, 0, -1) e a direção considerada "para cima" é o eixo Y.

A transformação de câmera pode ser implementada usando os comandos de manipulação de matrizes (que devem ser usados para que se possa construir outros tipos de transformações).

Por exemplo, suponha que a configuração da câmera seja a padrão. Se for necessário mover a câmera 20 unidades para a esquerda, pode-se executar essa tarefa com este trecho de código:

```
glLoadIdentity ();

glTranslatef (-20.0f, 0.0f, 0.0f);
```

É interessante notar que esse exemplo é equivalente a este outro:

```
glLoadIdentity ();
```

```
glLookAt (-20.0, 0.0, 0.0,
            0.0, 0.0, -1.0,
            0.0, 1.0, 0.0);
```

7.6. Hierarquia de transformações

Suponha que seja necessário desenhar duas esferas em uma cena, como na figura 7.7.

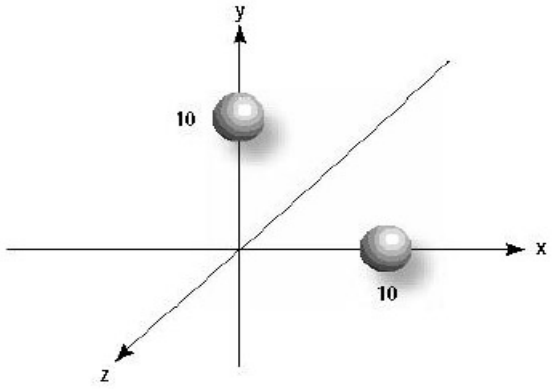

Figura 7.7 Cena com duas esferas (WrSw99)

Como sabe-se que as operações de transformação de matrizes são cumulativas, uma primeira tentativa de se resolver esse caso poderia ser feita da seguinte forma:

```
glLoadIdentity ();

glTranslatef (10.0f, 0.0f, 0.0f);
desenharEsfera ();

glLoadIdentity ();
glTranslatef (0.0f, 10.0f, 0.0f);
```

Entretanto, esse exemplo é inadequado por vários motivos. Primeiramente, quando existem muitos objetos na cena, essa abordagem torna-se confusa e ineficiente (várias chamadas a `glLoadIdentity`). Outro problema é quando alguns objetos dependem de outros. Por exemplo, se uma das esferas girasse em torno da outra, seria bem mais complicado implementar esse modelo se a cada transformação fosse necessário carregar a matriz identidade.

Para resolver esses problemas, o OpenGL fornece pilhas de matrizes. As pilhas podem ser usadas para salvar o valor de alguma matriz, realizar as operações desejadas e depois restaurar o valor da matriz anterior.

O OpenGL mantém uma pilha para cada tipo de matriz existente. Assim como os comandos de manipulação de matrizes, os comandos de manipulação da pilha afetam apenas o tipo de matriz atual. Esses comandos são `glPushMatrix` e `glPopMatrix`.

Aqui está um exemplo de utilização desses comandos:

```
glLoadIdentity ();
```

```
// salva a matriz atual (identidade)

glPushMatrix ();

  glTranslatef (0.0f, 10.0f, 0.0f);
  desenharEsfera ();

glPopMatrix ();

/* a matriz atual volta a ser a identidade */

glPushMatrix ();

  glTranslatef (10.0f, 0.0f, 0.0f);
  desenharEsfera ();

glPopMatrix ();
```

7.7. Outras operações (avançado)

Além das operações implementadas pelo OpenGL, é possível realizar operações personalizadas. O OpenGL possui dois comandos para definir essas operações:

```
glLoadMatrix* (m);   // * = f ou d

glMultMatrix* (m);   // * = f ou d
```

A função `glLoadMatrix*` permite substituir a matriz atual pela matriz especificada como parâmetro. A função `glMultMatrix*` permite multiplicar a matriz atual por uma matriz qualquer.

A definição da matriz m deve ser feita usando um *array* simples:

```
GLfloat matriz [16];
```

O OpenGL trabalha internamente vetores coluna, por isso a matriz `m` é tratada da seguinte forma:

$$\begin{bmatrix} a_0 & a_4 & a_8 & a_{12} \\ a_1 & a_5 & a_9 & a_{13} \\ a_2 & a_6 & a_{10} & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \end{bmatrix}$$

Figura 7.8 Armazenamento interno da matriz

O formato das matrizes usadas em OpenGL é diferente do formato usual de matrizes na linguagem C:

```
float matriz [4][4];      // C convencional
```

O problema é que na linguagem C, as matrizes são armazenadas em memória por ordem de linha. Dessa forma, os quatro primeiros elementos correspondem à primeira linha, os quatro seguintes à segunda linha, e assim sucessivamente. Assim, o elemento m (i, j) da matriz declarada em C representa o elemento a (j, i) da matriz do OpenGL.

Para ilustrar o uso dessas funções, o exemplo a seguir implementa a funcionalidade da função `glTranslatef`:

```
void translate (GLfloat x, GLfloat y, GLfloat z)
{
  GLfloat * m = {1.0f, 0.0f, 0.0f,  0.0f,
                 0.0f, 1.0f, 0.0f,  0.0f,
                 0.0f, 0.0f, 1.0f,  0.0f,
                 x   , y   , z   ,  1.0f};

  glMultMatrixf (m);

}
```

8. Composição de objetos

A maneira mais simples de se desenhar algum objeto seria especificar os seus vértices em alguma função qualquer, como por exemplo:

```
glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)

desenharMeuObjeto ();
```

Existem várias desvantagens em se usar essa abordagem. Por exemplo, se for necessário realizar cálculos complexos para gerar a geometria do objeto, nesse exemplo esses cálculos seriam realizados várias vezes (quando eles poderiam ser feitos uma vez só, como na inicialização do programa).

Com o OpenGL, é possível agrupar comandos para serem executados em uma única chamada, com as *diplay lists*. *Display lists* são listas de comandos do OpenGL pré-processados.

No exemplo do cálculo da geometria do objeto, ao se usar uma display list, somente o resultado final do processo (como os vértices) seria armazenado.

As *display lists* são referenciadas por um identificador, que é um número simples. Entretanto, é preciso requisitar ao OpenGL que gere um identificador válido. Isso pode ser feito da seguinte forma:

```
GLuint minhaLista = glGenLists (1);
```

A variável minhaLista contém o identificador da lista, e o argumento da função `glGenLists` indica quantas listas deverão ser criadas. Caso sejam criadas várias listas, elas poderão ser acessadas sequencialmente, a partir do identificador principal. Por exemplo, caso se queira criar cinco listas:

```
GLuint listas = glGenLists (5);
```

As listas criadas poderão ser acessadas usando `listas`, `listas+1`, `listas+2`, ..., `listas+4`.

Após a criação da lista, é necessário preenchê-la com comandos. Para fazer isso, usa-se:

```
glNewList (minhaLista, GL_COMPILE);

  ... // cálculos, comandos do OpenGL

glEndList ();
```

A constante `GL_COMPILE` indica ao OpenGL que os comandos devem ser agrupados na lista. Uma outra alternativa é usar `GL_COMPILE_AND_EXECUTE`, que indica que os comandos deverão ser armazenados e executados.

Alguns comandos como `glNewList` e `glEndList` não são armazenados na lista (existem outros, que estão descritos na documentação do OpenGL).

Depois que a lista foi definida, é possível executar os comandos armazenados a qualquer momento, com o seguinte comando:

```
glCallList (minhaLista);
```

Ao final da aplicação, é necessário liberar os recursos alocados pela lista, com este comando:

```
glDeleteLists (minhaLista, 1);
```

O primeiro argumento refere-se ao identificador da lista, e o segundo à quantidade de listas que serão apagadas.

Uma observação importante em relação a *display lists* é que as *display lists* são estáticas, ou seja, não é possível alterar os dados armazenados na lista depois que elas forem preenchidas. Entretanto, é possível substituir o seu conteúdo:

```
glNewList (minhaLista, GL_COMPILE);

  ... // novos comandos

glEndList ();
```

9. Iluminação

Para usar iluminação em OpenGL, é preciso seguir alguns passos:

- Habilitar o uso de iluminação;
- Especificar os vetores normais;
- Configurar o material dos objetos;
- Configurar as fontes de luz.

O uso de iluminação está desabilitado por padrão. Para habilitar a iluminação, é necessário usar o seguinte comando:

```
glEnable (GL_LIGHTING);
```

A iluminação pode ser desligada a qualquer momento usando este comando:

```
glDisable (GL_LIGHTING);
```

9.1. Vetores normais

Os vetores normais são usados pelo OpenGL para calcular a incidência de luz sobre uma superfície. O comando para especificar vetores normais é:

```
glVertex3* ()
```

Esses vetores podem ser associados a superfícies inteiras ou a vértices individuais. Para especificar os vetores normais por vértice, basta especificá-los com os vértices questão, como no exemplo a seguir:

```
glNormal3f (0.0f, 1.0f, 0.0f);
glVertex3f (a, b, c);

glNormal3f (0.0f, 0.0f, 1.0f);
glVertex3f (d, e, f);
```

Por questões de otimização, o OpenGL espera que os vetores normais fornecidos sejam unitários. É possível requisitar ao OpenGL que normalize os vetores conforme estes forem submetidos, com o seguinte comando:

```
glEnable (GL_NORMALIZE);
```

Entretanto, isso representa um custo adicional que pode ser evitado.

Outra observação importante é que os comandos `glScale*` afetam o comprimento dos vetores normais, o que pode ocasionar erros no cálculo de iluminação.

9.2. Materiais

Os materiais são propriedades de uma superfície que definem sua aparência. Essas propriedades têm a ver com a componente da luz que eles são capazes de refletir.

Os tipos de material que podem ser especificados no OpenGL são: `GL_AMBIENT`, `GL_DIFFUSE`, `GL_SPECULAR`, `GL_EMISSION` e `GL_SHININESS`.

O `GL_AMBIENT`, `GL_DIFFUSE` e `GL_SPECULAR` especificam a reflectância de luz ambiente, difusa e especular, respectivamente. O `GL_EMISSION` especifica a intensidade de luz emitida pelo material,. O `GL_SHININESS` especifica um valor (“specular exponent”) que tem a ver com a concentração de brilho em uma superfície. Quando maior esse valor, mais concentrado é o brilho especular (a área ocupada por ele é menor).

A primeira alternativa para se especificar materiais é usar a função `glMaterial*`. Essa função recebe três parâmetros: qual é o tipo de face a ser afetado (frente, trás ou ambas), qual é a propriedade a ser alterada e os valores da propriedade.

Como exemplo, tem-se o seguinte trecho de código:

```
GLfloat cinza [] = {0.60, 0.60, 0.60, 1.0};

glMaterialfv (GL_FRONT, GL_AMBIENT_AND_DIFFUSE, cinza);

glBegin (GL_TRIANGLES);

  glVertex3f (0.0f, 10.0f, 30.0f);
  glVertex3f (-50.0f, 0.0f, 30.0f);
  glVertex3f (50.0f, 0.0f, 30.0f);

glEnd ();
```

O exemplo especifica que o material refletido pelo triângulo, sob luz ambiente e difusa é uma variação de cinza. É importante notar que somente a face da frente será afetada. Caso se queira afetar ambas as faces, é necessário usar a constante `GL_FRONT_AND_BACK`.

Para configurar materiais de modo que possuam efeitos especulares, pode ser usado este trecho de código:

```
GLfloat cor [] = {1.0f , 1.0f, 1.0f, 1.0f};

glMaterialfv (GL_FRONT, GL_SPECULAR, cor);
glMaterialf  (GL_FRONT, GL_SHININESS, 128.0f);
```

A variável cor indica qual será a cor refletida pela luz especular. No exemplo, cor indica que todas as cores serão refletidas. O parâmetro `GL_SHININESS` pode variar de 0 a 128.

Caso uma cena possua muitos materias diferentes, o uso de `glMaterial*` pode tornar-se ineficiente. Por isso, existe uma outra alternativa para se especificar propriedades dos materiais.

A outra opção é usar o chamado *color tracking*. Quando o color tracking está ativado, a propriedade do escolhida é determinada com o uso de `glColor*`.

Inicialmente, é preciso habilitar o *color tracking*:

```
glEnable (GL_COLOR_MATERIAL);
```

A seguir, é preciso configurá-lo, com o seguinte comando:

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

O parâmetro `face` indica qual é o tipo de face afetada (`GL_FRONT`, `GL_BACK` ou `GL_FRONT_AND_BACK`). O parâmetro `mode` indica qual é a propriedade do material a ser usada, e pode ser `GL_EMISSION`, `GL_AMBIENT`, `GL_DIFFUSE`, `GL_SPECULAR`, ou `GL_AMBIENT_AND_DIFFUSE` (padrão). Exemplo de uso:

```
glEnable (GL_COLOR_MATERIAL);

glColorMaterial (GL_FRONT, GL_AMBIENT_AND_DIFFUSE);

glColor3f (1.0f, 0.0f, 0.0f);

glBegin (GL_TRIANGLES);
  glVertex3f (0.0f, 10.0f, 30.0f);
  glVertex3f (-50.0f, 0.0f, 30.0f);
  glVertex3f (50.0f, 0.0f, 30.0f);
glEnd ();
```

9.3. Fontes de luz

É possível especificar uma valor de luz ambiente global para a cena. Isso pode ser feito alterando-se o modelo de iluminação usado pela aplicação, que é demonstrado no seguinte trecho de código:

```
GLfloat luz_ambiente [] = {1.0f, 1.0f, 1.0f, 1.0f};

glLightModelfv (GL_LIGHT_MODEL_AMBIENT, luz_ambiente);
```

Por padrão, o valor padrão de luz ambiente para ess modelo é {0.2, 0.2, 0.2, 1.0}. Existem outros modelos que podem ser alterados.

A especificação do OpenGL prevê um número mínimo de oito fontes de luz que podem ser usadas em um programa. Entretanto, nem todas elas necessitam ser implementadas em *hardware* (algumas implementações podem ter apenas uma fonte de luz em *hardware*).

Existem três tipos de fonte de luz no OpenGL: fontes de luz direcionais, fontes de luz pontuais e fontes de luz refletoras (*spot lights*). A especificação do tipo de fonte de luz é feita através da configuração de seus parâmetros (não existe uma diferenciação explícita entre esses tipos).

Aqui está um exemplo de configuração de fonte de luz:

```
GLfloat corAmbiente [] = {0.5f, 0.5f, 0.5f, 1.0f};
GLfloat corDifusa   [] = {1.0f, 1.0f, 1.0f, 1.0f};

GLfloat posicao [] = {0.0f, 0.0f, 50.0f, 1.0f};

glLightfv (GL_LIGHT0, GL_AMBIENT, corAmbiente);
glLightfv (GL_LIGHT0, GL_DIFFUSE, corDifusa);
glLightfv (GL_LIGHT0, GL_POSITION, posicao);

glEnable (GL_LIGHT0);
```

Todas as propriedades da fonte de luz são especificadas pela função `glLight*`. Em linhas gerais, esse comando recebe como parâmetros uma contante que indica a fonte de luz a ser alterada (`GL_LIGHT0`, `GL_LIGHT1`, ..., `GL_LIGHT7`), a propriedade a ser altera e valor dessa propriedade.

No exemplo, foram especificadas as componentes de cor difusa e ambiente da fonte de luz, assim como sua posição. A diferença entre entre fontes de luz direcionais e pontuais é determinada pelo parâmetro de posição. Se o quarto parâmetro (w) for 1.0, significa que a fonte de luz é pontual (e está localizada naquela posição). Caso o valor seja 0.0, significa que a fonte de luz é direcional (ou seja, ela está localizada a uma distância infinita ao longo do vetor especificado como posição).

Por último, é necessário habilitar o uso da fonte de luz desejada.

O valor do componente especular da fonte de luz pode ser alterado com a propriedade `GL_SPECULAR`:

```
GLfloat corEspecular [] = {1.0f , 1.0f, 1.0f, 1.0f};

glLightfv (GL_LIGHT0, GL_SPECULAR, corEspecular);
```

Para a criação de *spot lights*, existem algumas propriedades que podem ser alteradas, conforme é descrito neste exemplo:

```
GLfloat  spotDirection [] = {0.0f, 0.0f, -1.0f};

glLightfv (GL_LIGHT0, GL_SPOT_DIRECTION, spotDirection);
glLightf  (GL_LIGHT0, GL_SPOT_CUTOFF, 60.0f);
glLightf  (GL_LIGHT0, GL_SPOT_EXPONENT, 100.0f);
```

A principal propriedade a ser alterada é o ângulo (em graus) que define o cone de luz, que corresponde à constante `GL_SPOT_CUTOFF`. Essa propriedade é ilustrada na figura 9.1.

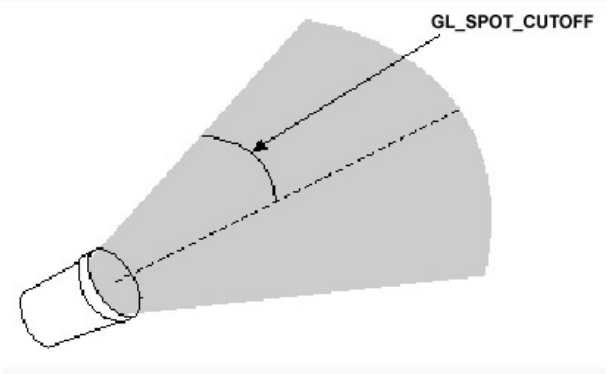

Figura 9.1 Propriedade GL_SPOT_CUTOFF (NeDa97)

Os valores possíveis para essa propriedade variam de 0º a 90º. Existe um valor especial, 180º, que pode ser usado para desabilitar essa propriedade (efetivamente desabilitando esse tipo de fonte de luz). Esse valor é o padrão.

A propriedade `GL_SPOT_DIRECTION` indica qual é a direção do feixe de luz. O valor da propriedade é expresso em coordenadas do objeto (ou seja, em relação à fonte de luz).

A propriedade `GL_SPOT_EXPONENT` indica a intensidade de distribuição da luz. Os valores aceitos variam de 0 a 128. Quanto maior o valor, mais concentrada é a luz no centro do cone. O valor padrão é 0, que resulta em uma distribuição uniforme da luz.

10. Mapeamento de textura

O mapeamento de texturas em OpenGL é feito com os seguintes passos:

- Habilitar o uso de texturas;
- Leitura da imagem em disco (ou geração dos dados da textura através de algum algoritmo);
- Transferir os dados da textura para o OpenGL;
- Selecionar a textura para uso e especificar as coordenadas de textura dos vértices.

Esses passos serão explicados nas seções seguintes.

10.1. Habilitar o uso de texturas

O mapeamento de texturas, por padrão, está desligado no OpenGL. O OpenGL permite o uso de texturas de uma, duas ou três dimensões. Neste trabalho, o enfoque será dirigido ao uso de texturas de duas dimensões.

Para habilitar o mapeamento de texturas de duas dimensões, é necessário usar o seguinte comando:

```
glEnable (GL_TEXTURE_2D);
```

De forma análoga a de outras propriedades, o mapeamento de texturas pode ser desligado com o comando correspondente `glDisable`.

10.2. Identificadores de texturas

Todas as texturas em OpenGL são referenciadas por um identificador (número). Os identificadores podem ser requisitados ao OpenGL com o seguinte trecho de código:

```
GLuint identificador;

glGenTextures (1, & id);
```

O primeiro argumento do comando `glGenTextures` indica a quantidade de identificadores a serem criados enquanto o segundo parâmetro indica onde esses identificadores serão armazenados. Como é possível perceber, vários identificadores podem ser gerados de uma só vez, como é exemplificado a seguir:

```
GLuint texturas [5];

glGenTextures (5, texturas);
```

Para que seja possível usar a textura, é necessário informar ao OpenGL que a textura será usada. Para isso, é necessário usar este comando:

```
glBindTexture (GL_TEXTURE_2D, identificador);
```

A partir desse momento, todas as operações envolvendo texturas fazem referência e/ou afetam a textura representada por `identificador`.

Quando a textura não for mais necessária, será preciso liberar os recursos que foram alocados. Essa tarefa é realizada como no exemplo a seguir:

```
glDeleteTextures (1, & identificador);

glDeleteTextures (5, texturas);
```

O comando `glDeleteTextures` recebe como parâmetros a quantidade de texturas a terem seus recursos devolvidos ao sistema e um local (uma variável ou *array*) que contém os identificadores das texturas.

10.3. Transferência dos dados para o OpenGL

Para transferir os dados da textura para o OpenGL, é necessário usar este comando:

```
void glTexImage2D(GL_TEXTURE_2D,
                  GLint level,
                  GLint components,
                  GLsizei width,
                  GLsizei height,
                  GLint border,
                  GLenum format,
                  GLenum type,
                  const GLvoid *pixels
                 );
```

O parâmetro `level` indica o nível de detalhe da textura. O nível de detalhe é usado para se especificar *mipmaps*. O nível zero (0) corresponde à imagem original.

O parâmetro `components` indica o número de componentes de cor existentes na imagem. Os valores permitidos são 1, 2, 3 ou 4.

Os parâmetros `width` e `height` indicam a largura e a altura da imagem em *pixels*, respectivamente. Uma observação importante é que esses valores precisam ser potência de 2.

O parâmetro `border` indica se a textura possui uma borda. Os valores possíveis são 0 ou 1.

O parâmetro `format` indica qual é o formato dos dados da imagem. Existem várias constantes para representar esses tipos, como `GL_RGB`, `GL_RGBA` e `GL_LUMINANCE` (esses valores estão listados na documentação).

O parâmetro `type` indica qual é o tipo verdadeiro do *array* `pixels`. Por exemplo, se os dados da imagem foram carregados em um *array* de `GLubytes`, o tipo será `GL_UNSIGNED_BYTE` (os outros valores estão listados na documentação.

O último parâmetro corresponde ao *array* que contém os dados da imagem.

Um exemplo simples de uso dessa função é descrito no trecho de código a seguir:

```
/**
 * Carregar uma imagem RGB de 64x64 do disco e transferí-la
 * para o OpenGL
 */

unsigned int * buffer;
```

```
GLuint  id;

...   // alocar buffer para imagem e carrega imagem do disco
      // (isso ainda será visto com mais detalhes)

// habilita o uso de texturas 2D
glEnable (GL_TEXTURE_2D);


// gerar um identificador de textura
glGenTextures (1, & id);


// selecionar a textura para uso
glBindTexture (GL_TEXTURE_2D, id);


// transferir os dados para o OpenGL
glTexImage2D (GL_TEXTURE_2D,
              0,                 // nível de detalhe
              3,                // número de componentes de cor
              64,                // largura
              64,                // altura
              0,                 // borda
              GL_RGB,            // formato
              GL_UNSIGNED_BYTE   // tipo de dados do array
              buffer             // array contendo os
                                 // dados da imagem
              );
```

10.3.1. Mipmaps

A técnica de *mipmaps* foi definida por Lance Williams no artigo *Pyramidal Parametrics* (SIGGRAPH '83). A técnica consiste em se criar várias versões de uma mesma imagem, em resoluções diferentes, como ilustrado na figura 10.1.

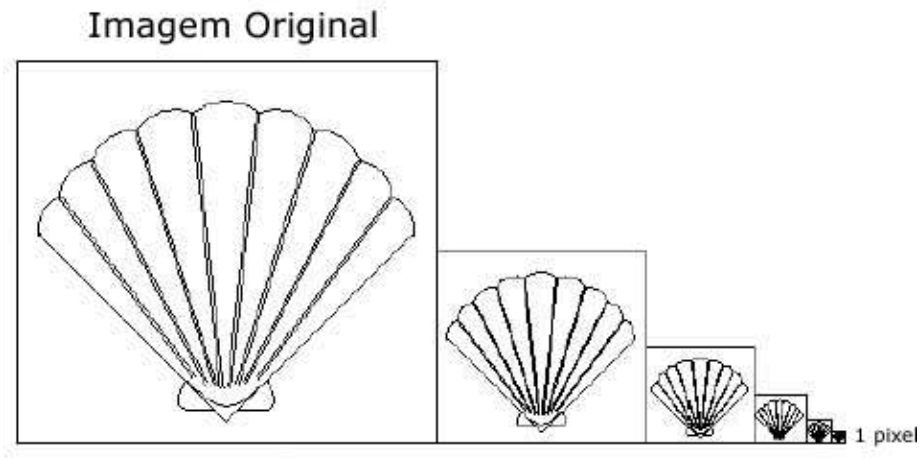

Figura 10.1 Exemplo de uma cadeia de mipmaps (NeDa97)

Esse processo é realizado até que se chegue à menor imagem possível (com 1 *pixel*). As *mipmaps* podem ser aplicadas para mapeamento de texturas em polígonos da seguinte forma: os polígonos que estão distantes do observador usariam versões de menor resolução das texturas, enquanto os que estão próximos do observador utilizariam uma textura de maior resolução.

Algumas vantagens desse método são:

- Melhoria do desempenho da aplicação, porque como as imagens são pré-fltradas, é possível escolher uma imagem que possua tamanho próximo ao do polígono a ser mapeado, reduzindo o custo do processamento em tempo real da interpolação. Adicionalmente, podem ser aplicados filtros mais custosos (que produzam melhores resultados visuais) na etapa de inicialização (geração das *mipmaps*), o que talvez não poderia ser feito em tempo real.
- Diminuição de artefatos visuais causados pelo *aliasing*.

Uma das desvantagens é a quantidade de espaço adicional para se armazenar as imagens.

O OpenGL possui uma função para a geração automática de *mipmaps*, que está definida na GLU. Essa função pode ser usada da seguinte forma:

```
gluBuild2DMipmaps (GL_TEXTURE_2D,
                   components,    // igual ao glTexImage2D
                   width,         // igual ao glTexImage2D
                   height,        // igual ao glTexImage2D
                   format,        // igual ao glTexImage2D
                   type,          // igual ao glTexImage2D
                   pixels         // igual ao glTexImage2D

                   );
```

10.4. Leitura da imagem em disco

O OpenGL não oferece nenhuma funcionalidade para ler e interpretar arquivos de imagem do disco. Essa tarefa é de responsabilidade do desenvolvedor. Entretanto existem diversas bibliotecas disponíveis na internet para auxiliar o desenvolvedor nessa tarefa. Neste trabalho, será usada a biblioteca DevIL para a leitura dos dados em disco.

10.4.1. Instalação da DevIL

A biblioteca DevIL (*Developer's Image Library*) está disponível em http://www.imagelib.org. Essa biblioteca é gratuita e possui o código aberto, está disponível para várias plataformas. Ela foi escolhida por ser fácil de se usar e possuir um modelo de uso parecido com o OpenGL.

Inicialmente, é preciso baixar do site da DevIL a versão pré-compilada da biblioteca (a não ser que ser queira compilá-la para gerar as DLLs). A versão pré-compilada para Windows está pronta para ser usada com o Visual C++.

A seguir, é necessário configurar o compilador para usar os arquivos .h e os arquivos de ligação. As instruções supõem que o arquivo tenha sido descompactado em c:\devil.

Para configurar a DevIL no Visual C++ 6:

- Acesse os menus “Tools”, “Options” e escolher a aba “Directories”. Em “Show directories for”, escolher “include files”. Criar uma nova entrada e acrescente “c:\devil\include”.
- Acesse os menus “Tools”, “Options” e escolher a aba “Directories”. Em “Show directories for”, escolher “library files”. Criar uma nova entrada e acrescentar “c:\devil\lib”.

Para configurar a DevIL no Visual C++ .NET:

- Acesse os menus “Tools”, “Options” e escolher “Projects” no painel da esquerda. A seguir, escolher “VC++ Directories” e em “Show directories for”, escolher “Include files”. Criar uma nova entrada e acrescentar “c:\devil\include”.
- Acesse os menus “Tools”, “Options” e escolher “Projects” no painel da esquerda. A seguir, escolher “VC++ Directories” e em “Show directories for”, escolher “library files”. Criar uma nova entrada e acrescentar “c:\devil\lib”.

10.4.2. Leitura da imagem em disco

Uma vez que DevIL foi configurada corretamente, é possível usar suas funções para ler os dados da imagem do disco, conforme demonstra o trecho de código a seguir:

```
/**
 * Esta função demonstra como carregar um arquivo em disco
 * com a DevIL.
 *
 * Retorna um identificador de textura do OpenGL.
 * Caso seja 0, é porque algum erro ocorreu no processo.
 */

GLuint CarregarImagem (const char * arquivo)
{
  // esta variável representa um identificador para a imagem
  ILuint image;


  // cria um id para a imagem
  ilGenImages (1, & image);


  // seleciona a imagem para uso
```

```
ilBindImage (image);


// determinamos que o 0,0 da figura é o canto
// inferior esquerdo
ilEnable (IL_ORIGIN_SET);

ilOriginFunc (IL_ORIGIN_LOWER_LEFT);


// carregamos a imagem do disco
ilLoadImage (arquivo);


// algum erro no carregamento do arquivo ?
ILenum erro = ilGetError ();

if (erro != IL_NO_ERROR)
  {
      // desaloca recursos usados para carregar a imagem
      ilDeleteImages (1, & image);

      return 0;
  }

// recuperamos dados sobre a imagem
int width         = ilGetInteger (IL_IMAGE_WIDTH);
int height        = ilGetInteger (IL_IMAGE_HEIGHT);
int bytesPerPixel = ilGetInteger (IL_IMAGE_BYTES_PER_PIXEL);
ILenum imageFormat = ilGetInteger (IL_IMAGE_FORMAT);

ILubyte * imageData = ilGetData ();

// envia os dados para o OpenGL
GLuint tex = 0;

// habilita o uso de texturas
glEnable (GL_TEXTURE_2D);

// gera um identificador para a textura
glGenTextures (1, & tex);


// seleciona a textura para uso
glBindTexture (GL_TEXTURE_2D, tex);


// carrega os dados para o OpenGL
glTexImage2D (GL_TEXTURE_2D,
              0,             // nível de detalhe
```

```
                bytesPerPixel, // número de componentes de cor
                width,         // largura
                height,        // altura
                0,             // borda
                imageFormat,   // formato
                GL_UNSIGNED_BYTE  // tipo de dados do array
                imageData      // array contendo os dados
                               // (pixels) da imagem
                );

  // desaloca os recursos usados para carregar
  // a imagem com a DevIL
  ilDeleteImages (1, & image);

  // retorna o identificador de textura
  return tex;

}
```

10.4.3. Outras considerações

Para se usar as funções da DevIL, é preciso incluir o seu arquivo principal:

```
#include <IL/il.h>
```

Antes que as funções da DevIL possam ser usadas, é necessário inicializar a biblioteca. Ao final do programa, é necessário finalizar a biblioteca. Esses comandos correspondem a:

```
ilInit ();     // inicializar a DevIL

ilShutDown (); // finalizar a DevIL
```

Para a ligação com a DLL, é preciso especificar o arquivo de ligação. No Visual C++ 6, isso pode ser feito da seguinte forma:

- Acesse o menu “Projects” e depois “Settings”. A seguir, acesse a aba “Linker”. No campo “Object/library modules” acrescente “devil.lib” (sem aspas).

Já no Visual C++ .NET, o procedimento é este:

- Acesse o menu “Project” e depois “Properties”. No painel da esquerda, escolha a pasta “Linker” e depois “Input”. No campo “Additional dependencies”, acrescente “devil.lib” (sem aspas).

10.5. Coordenadas de textura

Para efetivamente aplicar a textura nos polígonos, é preciso especificar as coordenadas de textura.

O sistema de coordenadas de textura (2D), é ilustrado na figura 10.2.

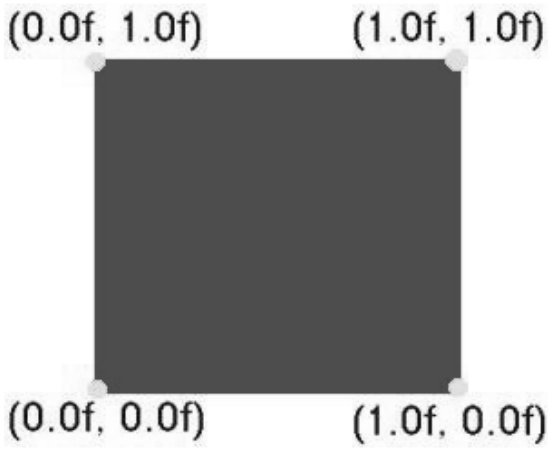

Figura 10.2 Sistema de coordenadas de textura 2D

As coordenadas de textura 2D são especificadas com este comando:

```
glTexCoord2* (s, t)
```

O eixo s corresponde ao eixo horizontal da figura 10.2 e o eixo t corresponde ao eixo vertical da mesma figura. No exemplo a seguir, uma textura é mapeada em um quadrado:

```
glBindTexture (GL_TEXTURE_2D, id);

glColor3f (1.0f, 1.0f, 1.0f);

glBegin (GL_QUADS);

  glTexCoord2f (0.0f, 0.0f);
  glVertex3f  (-10.0f, -10.0f, 0.0f);

  glTexCoord2f (1.0f, 0.0f);
  glVertex3f (10.0f, -10.0f, 0.0f);

  glTexCoord2f (1.0f, 1.0f);
  glVertex3f (10.0f, 10.0f, 0.0f);

  glTexCoord2f (0.0f, 1.0f);
  glVertex3f (-10.0f, 10.0f, 0.0f);

glEnd ();
```

Como visto na figura 10.2, as coordenadas de textura variam de 0.0 a 1.0 nos dois eixos. Entretanto, caso sejam especificados valores fora dessa faixa, ocorrerá uma destas alternativas: a textura será "truncada" (`GL_CLAMP`) ou a textura será repetida pelo polígono (`GL_REPEAT`, que é o padrão). A figura 10.3 ilustra esses resultados:

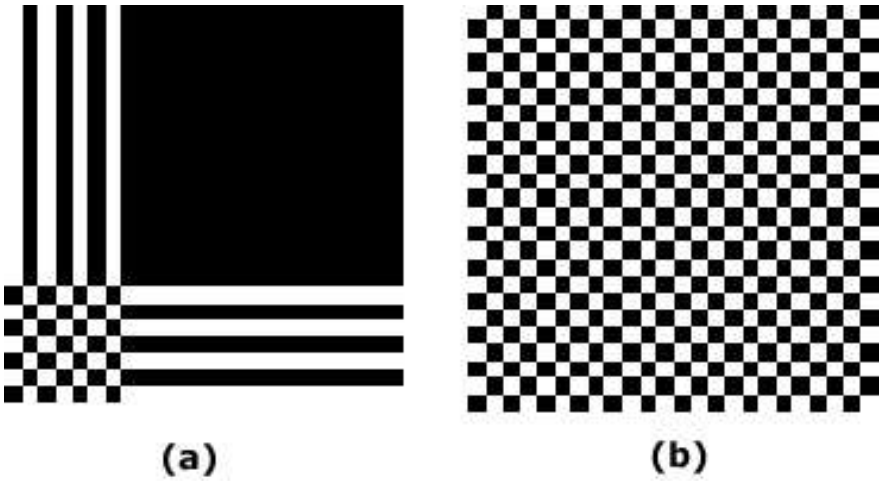

Figura 10.3 Resultados (NeDa97) (a) Truncamento. (b) Repetição (padrão)

Esse comportamento pode ser redefinido utilizando-se o seguinte trecho de código:

```
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, modo);
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, modo);
```

A constante `GL_TEXTURE_WRAP_S` indica que a alteração será efetuada somente para o eixo S, enquanto que a constante `GL_TEXTURE_WRAP_T` indica que o eixo T será o afetado. Dessa forma, é possível determinar comportamentos diferentes para cada um dos eixos.

10.6. Outros parâmetros

Existem diversos parâmetros que afetam a aparência de textura que podem ser configurados. Um dos mais interessantes é aquele que determina qual será o filtro utilizado para interpolar a textura sobre o polígono.

Os filtros disponíveis podem ser de dois tipos: filtros de redução ou filtros de ampliação. Os filtros de redução são usados quando a textura é maior do que o polígono a ser mapeado, enquanto o filtro de ampliação é usado quando o polígono é maior do que a textura. Os filtros podem ser especificados com este trecho de código:

```
glTextureParameteri (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
filtro);

glTextureParameteri (GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
filtro);
```

A constante `GL_TEXTURE_MIN_FILTER` indica que o filtro que será alterado é o filtro de redução. Já o filtro de ampliação é representado pela constante `GL_TEXTURE_MAG_FILTER`.

Os filtros existentes são os seguintes:

- `GL_NEAREST`: Utiliza o *texel* mais próximo (vizinho) do *pixel* que está sendo mapeado. É o filtro de melhor desempenho, porém é o de menor qualidade. Pode ser usado como filtro de ampliação ou redução.

- `GL_LINEAR`: Utiliza interpolação linear. Provê melhores resultados visuais, mas é mais custoso em termos computacionais. Pode ser usado como filtro de redução e ampliação. Esse filtro é o selecionado por padrão.

- `GL_NEAREST_MIPMAP_NEAREST`: Escolhe a *mipmap* com tamanho semelhante ao do polígono que está sendo mapeado e usa o mesmo critério que `GL_NEAREST` para escolher o *texel* da textura.

- `GL_NEAREST_MIPMAP_LINEAR`: Escolhe a *mipmap* com tamanho semelhante ao do polígono que está sendo mapeado e usa o mesmo critério que `GL_LINEAR` para escolher o *texel* da textura.

- `GL_LINEAR_MIPMAP_NEAREST`: Escolher duas *mipmaps* com tamanhos mais próximos ao do polígono que está sendo mapeado e usar o filtro `GL_NEAREST` para escolher o *texel* de cada textura. O valor final é uma média entre esses dois valores.

- `GL_LINEAR_MIPMAP_LINEAR`: Escolher duas *mipmaps* com tamanhos mais próximos ao do polígono que está sendo mapeado e usar o filtro `GL_LINEAR` para escolher o *texel* de cada textura. O valor final é uma média entre esses dois valores.

Outro parâmetro que pode ser alterado indica o modo como a textura será afetada pelas cores dos polígonos e informações de iluminação calculadas. O modo padrão (`GL_MODULATE`) indica que essas informações sejam interpoladas com os dados da textura. Já o modo `GL_DECAL` indica que o mapeamento deve ignorar essas informações (a textura é mapeada como se fosse um "adesivo").

Esses modos podem ser determinados com o comando:

```
glTexEnvi (GL_TEXTURE_ENV, GL_TEXTURE_MODE, modo);
```

11. Objetos mais complexos

O OpenGL oferece meios de desenhar objetos mais complexos do que linhas e polígonos, como curvas de Bèzier e NURBS.

Esta seção explica um outro tipo de objeto que pode ser desenhado em OpenGL: as superfícies quadráticas. Com o uso de superfícies quadráticas, é possível desenhar cilindros, discos e esferas.

As superfícies quadráticas são representadas em OpenGL pelo tipo de dados `GLUquadricObj`. Inicialmente, é preciso criar um objeto desses para uso:

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

GLUquadricObj * quadric = gluNewQuadric ();
```

Quando não for mais necessário usar o objeto `quadric`, será necessário liberar os recursos alocados para a superfície:

```
gluDeleteQuadric (quadric);
```

Existem várias propriedades da superfície que podem ser configuradas através de comandos da GLU.

O comando `gluQuadricStyle` afeta o modo como o objeto é desenhado. Os modos possíveis são `GLU_FILL` (sólido), `GLU_LINE` (*wireframe*), `GLU_SILHOUETTE` (desenha somente a silhueta) e `GLU_POINT` (desenha somente os vértices). A sintaxe desse comando é:

```
gluQuadricStyle (quadric, estilo);
```

O comando `gluQuadricNormals` controla o modo como os vetores normais do objeto são gerados. O modo `GLU_NONE` indica que os vetores normais não devem ser gerados. O modo `GLU_FLAT` indica que os vetores normais devem ser gerados para cada face somente, o que resulta em uma aparência facetada para o objeto. Já o modo `GLU_SMOOTH` indica que os vetores normais devem ser gerados para cada vértice, para que a aparência resultante seja suave. Os vetores normais afetarão a aparência do objeto caso a iluminação esteja sendo usada. O comando pode ser usado da seguinte forma:

```
gluQuadricNormals (quadric, modo);
```

Outra propriedade relacionada com iluminação indica qual é a orientação usada para os vetores normais. Existem duas opções: os vetores podem apontar para fora do objeto ou para dentro do objeto. As opções correspondentes são `GLU_OUTSIDE` e `GLU_INSIDE`. Geralmente, escolhe-se gerar os vetores normais apontando para dentro do objeto quando se deseja posicionar o observador dentro desse objeto. O comando para alterar essa propriedade é:

```
gluQuadricOrientation (quadric, valor);
```

As superfícies quadráticas também podem texturizadas. Existe um comando para determinar se as coordenadas de textura do objeto devem ser geradas automaticamente, como é exemplificado a seguir:

```
gluQuadricTexture (quadric, flag);
```

As opções possíves são `GLU_TRUE` (gerar as coordenadas) e `GLU_FALSE` (não gerar as coordenadas).

11.1. Cilindros

Os cilindros podem ser desenhados com o seguinte comando:

```
void gluCylinder (GLUquadricObj * qobj,
                  GLdouble baseRadius,
                  GLdouble topRadius,
                  GLdouble height,
                  GLint    slices,
                  GLint    stacks
                 );
```

Os parâmetros `baseRadius` e `topRadius` indicam os raio da base do cilindro e do topo do cilindro, respectivamente. O parâmetro `height` indica qual é a altura do cilindro. O parâmetro `slices` indica qual é o número de "faces" na lateral do cilindro. Já o parâmetro `stacks` indica quantas divisões existem ao longo do corpo do cilindro. Esses parâmetros são ilustrados na figura 11.1.

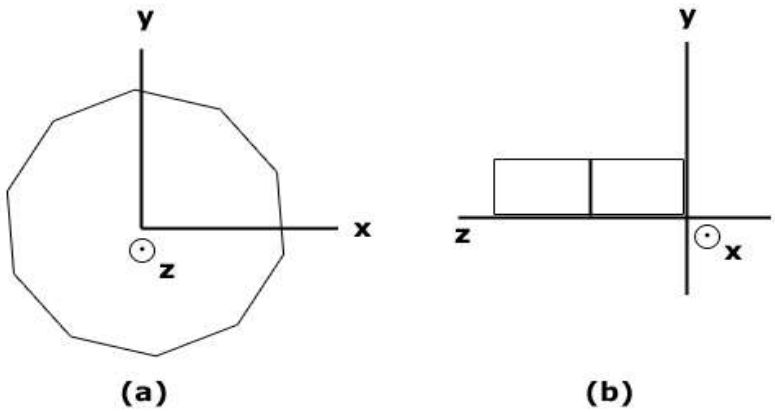

Figura 11.1 Parâmetros de gluCylinder. (a) slices. (b) stacks.

A base do cilindro é desenhada em z = 0, e o topo estará em z = `height`. As extremidades do cilindro não são preenchidas, ou seja, o cilindro não é fechado.

11.2. Discos

Os discos podem ser desenhados com a seguinte função:

```
void gluDisk (GLUquadricObj * qobj,
              GLdouble innerRadius,
              GLdouble outerRadius,
              GLint    slices,
              GLint    loops
             );
```

Os parâmetros `innerRadius` e `outerRadius` correspondem aos raios interno e externo do disco. O parâmetro `slices` indica quantos lados o disco possui. Esses parâmetros são ilustrados na figura 11.2.

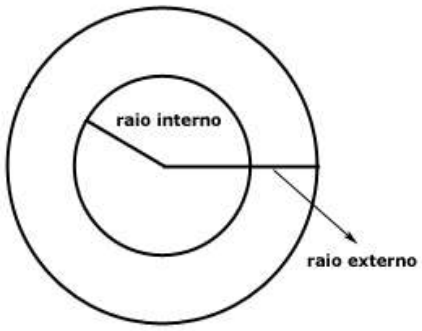

Figura 11.2 Parâmetros raio interno e externo de gluDisk

Finalmente, o parâmetro `loops` indica o número de círculos concêntricos que devem ser desenhados entre os raios interno e externo.

O disco é desenhado no plano z = 0.

11.3. Discos incompletos

A função `gluDisk` permite desenhar discos completos. É possível desenhar discos incompletos (que não possuem uma circunferência completa). Para isso, usa-se esta função:

```
void gluPartialDisk (GLUquadricObj * qobj,
                     GLdouble innerRadius,
                     GLdouble outerRadius,
                     GLint    slices,
                     GLint    loops,
                     GLdouble startAngle,
                     GLdouble sweepAngle
             );
```

A diferença entre `gluDisk` e `gluPartialDisk` é nesta última função, uma fração do disco é desenhada, com início no ângulo `startAngle` e fim no ângulo `startAngle+sweepAngle`. O ângulo `startAngle` é especificado no sentido horário quando visto a partir do topo do disco.

O disco é desenhado no plano z = 0.

11.4. Esferas

Para se desenhar esferas, é utilizado o seguinte comando:

```
void gluSphere (GLUquadricObj * qobj,
                GLdouble radius,
                GLint slices,
                GLint stacks
                );
```

A esfera é desenhada de forma que o seu centro coincida com a origem. O parâmetro `radius` define raio da esfera. O parâmetro `slices` indica o número de divisões da esfera em torno do eixo Z (como longitude). Já o parâmetro `stacks` indica o número de divisões da esfera ao longo do eixo Z (como latitude). A figura 11.3 ilustra os parâmetros `slices` e `stacks`.

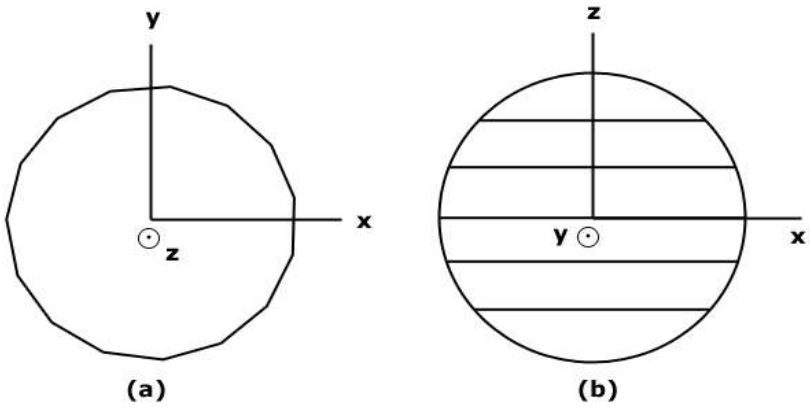

Figura 11.3 Parâmetros de glSphere. (a) slices. (b) stacks.

12. Exemplo com freeglut

Esta seção apresenta um esqueleto de programa que pode ser usado como ponto de partida para o desenvolvimento de aplicações simples com freeglut e OpenGL. Nesse programa, um quadrado vermelho será animado e desenhado em uma janela. Quando o quadrado colidir contra as bordas da parede (na verdade, da *viewport*), sua velocidade será alterada de modo que o objeto irá quicar. O usuário poderá encerrar a aplicação usando a tecla ESC do teclado.

O primeiro passo é incluir os arquivos necessários e declarar as funções que tratarão eventos no freeglut, como é feito neste trecho de código:

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

GLfloat quad_x = 0.0f;
GLfloat quad_y = 0.0f;
GLfloat quad_z = 0.0f;

GLfloat vx = 0.01f;
GLfloat vy = 0.005f;
GLfloat vz = 0.008f;

void OnUpdate ();
void OnResize (int width, int height);
void OnRender ();
void OnKeyPress (unsigned char key, int x, int y);

void InitGL ();
```

O único arquivo que precisa ser incluído é o do freeglut. A posição atual do quadrado é armazenada nas variáveis `quad_x`, `quad_y` e `quad_z`. O ponto de referência é o centro do quadrado. A velocidade do quadrado é armazenada nas variáveis `vx`, `vy` e `vz`.

As funções `OnUpdate`, `OnResize`, `OnRender` e `OnKeyPress` são usadas para tratar eventos. A atualização da animação é realizada em `OnUpdate`. A função `OnResize` é responsável por tratar o evento que ocorre quando a janela é redimensionada. As operações de desenho são feitas em `OnRender`, e a função `OnKeyPress` é responsável por tratar o teclado.

Finalmente, na função `InitGL`, é estabelecida uma configuração inicial para o OpenGL.

O programa principal é definido no seguinte trecho de código:

```
void main ()
{
  glutInitDisplayMode (GLUT_DOUBLE | GLUT_DEPTH | GLUT_RGB);
  glutCreateWindow ("Exemplo de animação");

  glutDisplayFunc  (OnRender);
  glutReshapeFunc  (OnResize);
  glutKeyboardFunc (OnKeyPress);
  glutIdleFunc     (OnUpdate);

  InitGL ();

  glutMainLoop ();

}
```

12.1. Função principal

A primeira linha do programa requisita ao freeglut que sejam usados dois *buffers* para a janela, um *z-buffer* e modo de cores RGB. Como a aplicação usa uma animação, o modo *double-buffered* é necessário para evitar o efeito de *flicker*.

As funções responsáveis por tratar eventos são especificadas ao freeglut logo depois, como repetido aqui:

```
  glutDisplayFunc  (OnRender);
  glutReshapeFunc  (OnResize);
  glutKeyboardFunc (OnKeyPress);
  glutIdleFunc     (OnUpdate);
```

A função `glutReshapeFunc` especifica qual é a *callback* usada para se tratar o evento de redimensionamento da janela. É importante que esse evento seja tratado para que se possa atualizar a *viewport* e a projeção usada pelo programa quando a janela mudar de tamanho (o OpenGL não faz isso automaticamente).

Uma das funções disponíveis para especificar a *callback* responsável tratar eventos do teclado é `glutKeyboardFunc`.

Em `glutIdleFunc`, é especificada uma função que será executada quando não houver outros eventos a serem tratados (em outras palavras, essa função é executada quando a aplicação está ociosa).

Logo adiante, os estados iniciais do OpenGL são determinados e a aplicação inicia seu laço principal de execução.

12.2. Inicialização

Na etapa de inicialização, são submetidos comandos que determinam funcionalidades que serão usadas no restante da aplicação. A função correspondente é descrita a seguir:

```
void InitGL ()
{
  glEnable (GL_DEPTH_TEST);
  glEnable (GL_CULL_FACE);

  glClearColor (0.0f, 0.0f, 0.0f, 1.0f);

  glMatrixMode (GL_MODELVIEW);
  glLoadIdentity ();

}
```

Essa função requisita que o testes de *z-buffer* e eliminação de faces escondidas (*backface culling*) sejam habilitados.

A seguir, a cor de fundo da janela é estabelecida. (preto).

Finalmente, determina-se que o tipo de projeção usado a partir deste momento será a matriz de modelagem.

12.3. Deteminação da projeção

A projeção usada pelo programa é determinada pela função `OnResize`:

```
void OnResize (int width, int height)
{
  glViewport (0, 0, width, height);

  glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();

    glOrtho (-12.0f, 12.0f, -12.0f, 12.0f, -12.0f, 12.0f);

  glMatrixMode (GL_MODELVIEW);
    glLoadIdentity ();

}
```

A primeira linha determina que a viewport usada pelo programa corresponde à janela inteira. A projeção usada pelo programa é uma projeção ortográfica com as mesmas medidas em todos os eixos.

Durante o ciclo de vida da aplicação, esse evento é executado pelo menos uma vez (depois que o laço principal de execução é iniciado).

12.4. Atualização

A função `OnUpdate` atualiza a posição do quadrado e verifica se existiu alguma colisão contra as bordas da janela (*viewport*, na verdade). Caso exista, a posição do quadrado é corrigida e a sua velocidade é invertida (no eixo onde ocorreu a colisão). O quadrado possui aresta de tamanho igual a 4 unidades.

```
void OnUpdate ()
{
  quad_x += vx;  quad_y += vy;  quad_z += vz;

  if (quad_x > 10.0f || quad_x < -10.0f)
    vx = - vx;

  if (quad_y > 10.0f || quad_y < -10.0f)
    vy = - vy;

  if (quad_z > 10.0f || quad_z < -10.0f)
    vz = - vz;

  glutPostRedisplay ();

}
```

A função `glutPostRedisplay` informa ao freeglut que é necessário redesenhar o conteúdo da janela. Se essa função não for usada, não será possível visualizar as alterações.

12.5. Teclado

A função `OnKeyPress` recebe como parâmetros a tecla que foi pressionada e a posição do mouse no momento em que a tecla foi pressionada.

```
void OnKeyPress (unsigned char key, int x, int y)
{
  switch (key)
    {
      case 'q': case 'Q' :
        exit (0); break;
    }

}
```

Existem outras funções para se tratar eventos do teclado, que podem ser consultadas na documentação do freeglut ou do GLUT.

12.6. Renderização

A função responsável pela renderização inicia com a requisição para que a tela e o *z-buffer* sejam limpos.

A seguir, a matriz identidade é carregada, para que o sistema de coordenadas seja reiniciado para a origem.

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

  glLoadIdentity ();

  glTranslatef (quad_x, quad_y, quad_z);

  glColor3f (1.0f, 0.0f, 0.0f);

  glBegin (GL_QUADS);

    glVertex3f (2.0f, 2.0f, 0.0f);
    glVertex3f (-2.0f, 2.0f, 0.0f);
    glVertex3f (-2.0f, -2.0f, 0.0f);
    glVertex3f (2.0f, -2.0f, 0.0f);

  glEnd ();

  glutSwapBuffers ();

}
```

A seguir, é necessário posicionar o quadrado, o que é feito com o comando `glTranslatef`. O quadrado é, então, especificado.

A função `glutSwapBuffers` requisita que o *buffer* usado para as operações de renderização seja exibido na tela.

13. Conclusões

OpenGL é uma API muito rica e cheia de recursos. Este trabalho aborda apenas uma parte do que poderia ser usado.

Existem tutoriais e outros recursos para se aprender mais sobre OpenGL em vários lugares na internet. Aqui estão alguns deles:

- Site oficial do OpenGL: http://www.opengl.org
- Nate Robins – Programas interativos com demonstração de vários conceitos sobre OpenGL: http://www.xmission.com/~nate/tutors
- NeHe – Tutoriais variados sobre OpenGL: http://nehe.gamedev.net
- GameTutorials – Vários tutoriais sobre OpenGL, com enfoque para programação de jogos: http://www.gametutorials.com
- JOGL – Biblioteca para uso OpenGL em Java: https://jogl.dev.java.net
- Dephi3D – OpenGL e programação 3D com Delphi: http://www.delphi3d.net

14. Referências Bibliográficas

OGL04	OpenGL, site oficial: http://www.opengl.org, 2004
WrSw99	Wright, Richard S.; Sweet, Michael – "OpenGL SuperBible Second Edition", Waite Group Press, 1999
NeDa97	Neider, Jackie; Davis, Tom – "OpenGL Programming Guide, 2nd Edition", Addison-Wesley, 1997
FGLUT04	freeglut, site oficial: http://freeglut.sourceforge.net, 2004
DEVIL04	DevIL, site oficial: http://www.imagelib.org, 2004