

Capítulo 13

Desenhos Tridimensionais

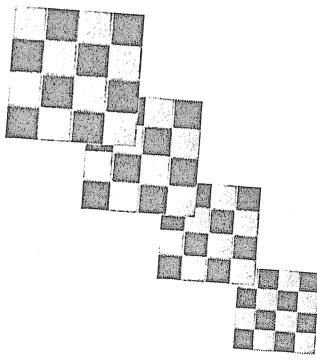


Figura 12.7 – Fazendo *pixel zoom* em imagens (ExemploImageZoom.cpp).

Neste capítulo vamos ver como fazer desenhos tridimensionais utilizando desde primitivas gráficas da biblioteca OpenGL até sólidos simples definidos na biblioteca GLUT (tais como esfera e cubo). Também se demonstra como utilizar a biblioteca `bibutil` para carregar e exibir objetos criados por programas de modelagem.

13.1 Primitivas Gráficas em Três Dimensões

A forma de representação de objetos 3D mais utilizada em Computação Gráfica consiste na especificação de uma malha de faces poligonais. Nesse caso, uma malha de polígonos representa uma superfície discretizada por faces planas, que podem ser triângulos (preferencialmente) ou quadrados. A estrutura de dados mais usada para armazenar essa malha é uma tabela de vértices e uma tabela de faces, que podem ser armazenadas em dois vetores. Conforme ilustra a Figura 13.1, os dados armazenados na tabela de faces correspondem a índices para a tabela de vértices – dessa forma, evita-se a duplicação de informações.

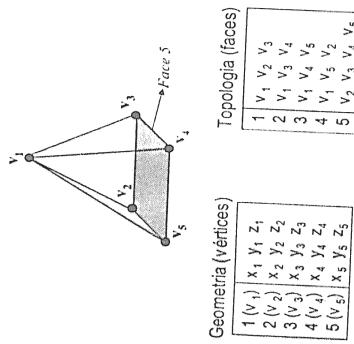


Figura 13.1 – Exemplo da representação de uma pirâmide por meio de uma tabela de vértices e uma tabela de faces.

A utilização de primitivas gráficas em três e duas dimensões é praticamente igual, seguindo as regras já apresentadas no Capítulo 8. A principal diferença está na especificação dos vértices, pois em três dimensões deve-se atribuir um valor para a coordenada z. Desse modo, a lista de vértices passada entre as chamadas de `glBegin(<argumento>)` e `glEnd()` deve ter três ou quatro parâmetros (ver seção 8.1). As constantes que podem ser passadas como argumento para a função `glBegin` são as mesmas, ou seja: `GL_POINTS`, `GL_LINES`, `GL_LINE_STRIP`, `GL_POLYGON`, `GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`, `GL_QUADS` ou `GL_QUAD_STRIP`.

Qualquer uma dessas constantes pode ser usada quando se está trabalhando em três dimensões, uma vez que é possível desenhar os pontos e as faces que compõem um objeto 3D utilizando triângulos ou quadriláteros (preenchidos ou não), como ilustra a Figura 13.2. Nessa figura, está sendo exibido o chamado *wireframe* de um bule de chá (ou teapot). *Wireframe* (aramado) consiste na representação de somente um objeto através de suas arestas [Foley, 1990]. Esse tipo de representação somente é usado para visualização de objetos quando não é necessário um grande grau de realismo ou em algumas aplicações especiais.

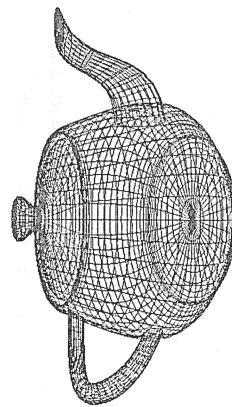


Figura 13.2 – Exemplo da visualização em *wireframe* de um objeto em 3D.

O exemplo seguinte demonstra como a estrutura apresentada pode ser implementada de forma simplificada.

Exemplo3DPiramide.cpp

A primeira providência é definir algumas estruturas para armazenar vértices e faces. A estrutura de vértices é simplesmente uma coordenada 3D:

```
029. // Define um vértice
030. typedef struct {
031.     float x,y,z; // posição no espaço
032. } VERT;
```

Já na estrutura de faces é preciso armazenar o total de vértices sendo utilizados por cada uma e um outro vetor, que contém os índices para cada vértice. Note que neste exemplo simples estamos assumindo que nenhuma face terá mais do que quatro vértices, ou seja, trataremos apenas triângulos e quadriláteros:

```
034. // Define uma face
035. typedef struct {
036.     int total; // total de vértices
037.     int ind[4]; // índices para o vetor de vértices
038. } FACE;

A última estrutura é para representar um objeto — aqui se guardam: apontadores para o vetor de vértices, para o vetor de faces e também a quantidade de faces, uma vez que essa informação não existe em nenhum outro local:
```

```
040. // Define um objeto 3D
041. typedef struct {
042.     VERT *vertices; // aponta para os vértices
043.     FACE *faces; // aponta para as faces
044.     int total_faces; // total de faces no objeto
045. } OBJ;
```

Cabe ressaltar que normalmente essas informações são carregadas de um arquivo no disco (ver seção 13.3), porém, como o objetivo aqui é apenas demonstrar como essas estruturas são utilizadas, o objeto será definido dentro do programa.

Primeiramente, criamos o vetor de vértices:

```
048. VERT vertices[] = {
049.     { -1, 0, -1 }, // 0 canto inf esquerdo tras.
050.     { 1, 0, -1 }, // 1 canto inf direito tras.
051.     { 1, 0, 1 }, // 2 canto inf direito diant.
052.     { -1, 0, 1 }, // 3 canto inf esquerdo diant.
053.     { 0, 2, 0 }, // 4 topo
054. };
```

À seguir criamos o vetor de faces. Observe que no caso das faces triangulares, o último índice de vértice não é utilizado e por isso inicializamos este com -1:

```
057. FACE faces[] = {
058.     { 4, { 0,1,2,3 }}, // base
059.     { 3, { 0,1,4,-1 }}, // lado traseiro
060.     { 3, { 0,3,4,-1 }}, // lado esquerdo
061.     { 3, { 1,2,4,-1 }}, // lado direito
062.     { 3, { 3,2,4,-1 }}, // lado dianteiro
063. };
```

Por fim, definimos o objeto em si:

```
065. // Finalmente, define o objeto pirâmide
066. OBJ piramide = {
067.     vertices, faces, 5 };
```

A pergunta que pode surgir agora é: por que fazer tudo isso? Por que não desenhar simplesmente as faces diretamente no código? Há vários motivos para criar esse conjunto de estruturas. O primeiro – e mais importante – é separar o modelo 3D do código que desenha o mesmo. Portanto, se for necessário trocar o modelo não será necessário modificar o código. O segundo motivo é eficiência no armazenamento: se substituirmos as estruturas por um conjunto de chamadas de desenho, potencialmente vários vértices serão repetidamente informados – no caso da pirâmide, cada vértice é compartilhado por pelo menos três faces. Isso também criaria um problema considerável no caso da necessidade de alterar o modelo: todas as ocorrências de cada vértice teriam que ser modificadas.

Partindo dessas razões, escrevemos então uma função que recebe um objeto genérico, indicado por sua estrutura devidamente preenchida, e traça o contorno de todas as suas faces (usando GL_LINE_LOOP):

```
069. // Desenha um objeto em wireframe
070. void DesenhaObjetoWireframe(OBJ *objeto)
071. {
072.     OBJ *obj = objeto;
073.     // Percorre todas as faces
074.     for(int f=0; f < obj->total_faces; ++f)
075.     {
076.         glBegin(GL_LINE_LOOP);
077.         // Percorre todos os vértices da face
078.         for(int v=0; v < obj->faces[f].total; ++v)
079.             glVertex3f(obj->vertices[faces[f].ind[v]].x,
080.                         obj->vertices[faces[f].ind[v]].y,
081.                         obj->vertices[faces[f].ind[v]].z);
082.     }
083.     glEnd();
084. }
```

Para desenhar um objeto, basta agora chamar a função `DesenhaObjetoWireframe` e passar um apontador para o objeto desejado (se houver mais de um). Isso é feito na função `callback` de desenho:

```
086. // Função callback de redesenho da janela de visualização
087. void Desenha(void)
088. {
089.     // Limpa a janela de visualização com a cor
090.     // de fundo definida previamente
091.     glClear(GL_COLOR_BUFFER_BIT);
092.     // Altera a cor do desenho para preto
093.     glColor3f(0.0f, 0.0f, 0.0f);
095. }
```

Este exemplo também introduz um conceito muito importante: comandos de navegação, que, a partir de agora, serão incluídos em praticamente todos os demais programas deste livro. Esses comandos são os seguintes:

- Movimento do mouse + botão esquerdo pressionado: rotaciona o objeto em torno dos eixos x e y.
- Movimento do mouse + botão direito pressionado: translação do objeto
- Movimento vertical do mouse + botão do meio pressionado: aproxima/afasta o objeto do observador.
- Teclas `Home` e `End`: alteram o ângulo utilizado na projeção perspectiva, ou seja, realiza `zoom in` e `zoom out`.

A implementação desses comandos é realizada em funções diferentes: para rotacionar o objeto, utilizam-se duas variáveis – `rotX` e `rotY` – que são empregadas na função que posiciona o observador (`PosicionaObservador`). Nessa mesma função também se empregam as variáveis `obsX`, `obsY` e `obsZ`, que informam a posição do objeto em relação ao observador.

```
103. // Função usada para especificar a posição do observador virtual
104. void PosicionaObservador(void)
105. {
106.     // Especifica sistema de coordenadas do modelo
107.     glMatrixMode(GL_MODELVIEW);
108.     // Inicializa sistema de coordenadas do modelo
109.     glLoadIdentity();
110.     // Posiciona e orienta o observador
111.     glTranslatef(-obsX,-obsY,-obsZ);
112.     glRotatef(rotX,1,0,0);
113.     glRotatef(rotY,0,1,0);
114. }
```

Já o ângulo de visão (variável `angle`) é empregado na função `EspecificaParametrosVisualizacao`:

```
116. // Função usada para especificar o volume de visualização
117. void EspecificaParametrosVisualizacao(void)
```

```

118. {
119.     // Especifica sistema de coordenadas de projeção
120.     glMatrixMode(GL_PROJECTION);
121.     // Inicializa sistema de coordenadas de projeção
122.     glLoadIdentity();
123.
124.     // Especifica a projeção perspective(angulo,aspecto,zMin,zMax)
125.     gluPerspective(angle,fAspect,0.1,1200);
126.
127.     PosicionaObservador();
128. }

129. void GerenciaMovim(int x, int y)
130. {
131.     // Função callback para eventos de movimento do mouse
132.     #define SENS_ROT 5.0
133.     #define SENS_OBS 15.0
134.     #define SENS_TRANSL 30.0
135.     void GerenciaMovim(int x, int y)
136.     {
137.         // Botão esquerdo ?
138.         if(bot==GLUT_LEFT_BUTTON)
139.         {
140.             // Calcula diferenças
141.             int deltax = x.ini - x;
142.             int deltay = y.ini - y;
143.             // E modifica ângulos
144.             rotY = rotY_ini - deltax/SENS_ROT;
145.             rotX = rotX_ini - deltay/SENS_ROT;
146.         }
147.     }
148. }

149. void GerenciaMouse(int button, int state, int x, int y)
150. {
151.     // Função callback para eventos de botões do mouse
152.     if(state==GLUT_DOWN)
153.     {
154.         // Salva os parâmetros atuais
155.         x_ini = x;
156.         y_ini = y;
157.         obsX_ini = obsX;
158.         obsY_ini = obsY;
159.         obsZ_ini = obsZ;
160.         rotX_ini = rotX;
161.         rotY_ini = rotY;
162.         bot = button;
163.     }
164.     else bot = -1;
165. }

166. void GerenciaMouse(int button, int state, int x, int y)
167. {
168.     // Botão direito ?
169.     if(button==GLUT_RIGHT_BUTTON)
170.     {
171.         // Calcula diferença
172.         int deltax = x_ini - x;
173.         int deltay = y_ini - y;
174.         // E modifica distância do observador
175.         obsZ = obsZ_ini + deltax/SENS_OBS;
176.     }
177. }

178. void GerenciaMovim(int x, int y)
179. {
180.     // Botão do meio ?
181.     if(button==GLUT_MIDDLE_BUTTON)
182.     {
183.         // Calcula diferença
184.         int deltax = x_ini - x;
185.         int deltay = y_ini - y;
186.         // E modifica posições
187.         obsX = obsX_ini + deltax/SENS_TRANSL;
188.         obsY = obsY_ini - deltay/SENS_TRANSL;
189.     }
190. }

```

A atualização da posição e da rotação é feita pelas *callbacks* de mouse. Para tanto, utiliza-se primeiramente a *callback* que recebe eventos quando um ou mais botões forem pressionados (`GerenciaMouse`). Nessa função, salva-se uma cópia dos valores iniciais de rotação e translação, bem como da posição atual do mouse. Também se armazena o número do botão que foi pressionado:

169. // Função callback para eventos de botões do mouse

170. void GerenciaMouse(int button, int state, int x, int y)

171. {

172. if(state==GLUT_DOWN)

173. {

174. // Salva os parâmetros atuais

175. x_ini = x;

176. y_ini = y;

177. obsX_ini = obsX;

178. obsY_ini = obsY;

179. obsZ_ini = obsZ;

180. rotX_ini = rotX;

181. rotY_ini = rotY;

182. bot = button;

183. }

184. else bot = -1;

185. }

186. s

Durante o movimento em si, a GLUT chama a função `GerenciaMovim`, a qual recebe as coordenadas atualizadas do mouse. Primeiramente, determina-se qual botão foi pressionado: se foi o botão esquerdo, calculam-se as diferenças (em x e y) entre a posição atual e a inicial (salva na função `GerenciaMouse`), e atualizamos as variáveis que controlam a rotação do objeto. A constante `SENS_ROT`, definida antes da função, permite especificar a sensibilidade do movimento: quanto maior o valor, mais o mouse terá que ser deslocado para afetar a rotação. É necessário haver esse controle, pois as diferenças são calculadas em pixels, ou seja, os valores são muito elevados para serem usados diretamente.

187. // Função callback para eventos de movimento do mouse

188. #define SENS_ROT 5.0

189. #define SENS_OBS 15.0

190. #define SENS_TRANSL 30.0

191. void GerenciaMovim(int x, int y)

192. {

193. // Botão esquerdo ?

194. if(bot==GLUT_LEFT_BUTTON)

195. {

196. // Calcula diferenças

197. int deltax = x.ini - x;

198. int deltay = y.ini - y;

199. // E modifica ângulos

200. rotY = rotY_ini - deltax/SENS_ROT;

201. rotX = rotX_ini - deltay/SENS_ROT;

202. }

No segundo trecho, trata-se o botão direito: nesse caso, basta calcular a diferença apenas em y e atualizar a variável `obsZ`. Aqui se utiliza outra constante, `SENS_OBS`, também para definir a sensibilidade do movimento:

203. // Botão direito ?

204. else if(button==GLUT_RIGHT_BUTTON)

205. {

206. // Calcula diferença

207. int deltax = x_ini - x;

208. // E modifica distância do observador

209. obsZ = obsZ_ini + deltax/SENS_OBS;

210. }

Por fim, o último trecho considera o botão do meio: aqui o processo é praticamente o mesmo do primeiro trecho, mas atualizam-se as variáveis que controlam a translação em x e y: `obsX` e `obsY`. Porém, note que as diferenças são somadas em x mas subtraídas em y, de modo que se o mouse for deslocado para baixo, o objeto também será movido na mesma direção. A constante de sensibilidade empregada aqui é denominada `SENS_TRANSL`:

211. // Botão do meio ?

212. else if(button==GLUT_MIDDLE_BUTTON)

213. {

214. // Calcula diferenças

215. int deltax = x_ini - x;

216. int deltay = y_ini - y;

217. // E modifica posições

218. obsX = obsX_ini + deltax/SENS_TRANSL;

219. obsY = obsY_ini - deltay/SENS_TRANSL;

220. }

```

221. PosicionaObservador();
222. glutPostRedisplay();
223. }

A outra função relevante é que trata eventos de teclas especiais, onde se altera o
zoom por meio das teclas Home e End:

155. // Função callback para tratar eventos de teclas especiais
156. void TeclasEspeciais (int tecla, int x, int y)
157. {
158.     switch (tecla)
159.     {
160.         case GLUT_KEY_HOME: if(angle>=10) angle -=5;
161.             break;
162.         case GLUT_KEY_END: if(angle<=150) angle +=5;
163.             break;
164.     }
165.     EspecificaParametrosVisualizacao();
166.     glutPostRedisplay();
167. }

```

O resultado desse programa pode ser visualizado na Figura 13.3.

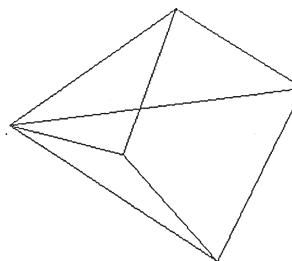


Figura 13.3 – Desenhando uma pirâmide em wireframe (Exemplo3DPiramide.cpp).

13.2 Objetos Predefinidos da GLUT

Para facilitar o desenho de sólidos simples, tais como cones, esferas e cubos, a biblioteca GLUT fornece um conjunto de sólidos predefinidos que dispensam a especificação de um conjunto de vértices e faces. Essas funções estão listadas a seguir [Schreiner, 2004].

```
void glutWireCube (GLfloat size)
```

Função usada para desenhar o wireframe de um cubo cujo tamanho é passado como parâmetro,

```
void glutWireTeapot (GLfloat size)
```

Função usada para desenhar o wireframe de um bule de chá, cujo tamanho é passado como parâmetro;

```
void glutWireSphere (GLfloat radius, GLint slices, GLint stacks)
```

Função usada para desenhar o wireframe de uma esfera representada por um conjunto de faces. O primeiro parâmetro (*radius*) corresponde ao raio da esfera. Os parâmetros *slices* e *stacks* indicam, respectivamente, o número de subdivisões em torno do eixo z (como se fossem linhas longitudinais) e o número de subdivisões ao longo do eixo z (como se fossem linhas latitudinais).

As intersecções dessas linhas formam as faces da esfera;

```
void glutWireCone (GLfloat radius, GLfloat height, GLint slices, GLint stacks)
```

Função usada para desenhar o wireframe de um cone. De forma similar à esfera, os parâmetros indicam, respectivamente: o raio da base do cone (*radius*), a altura do cone (*height*), *slices* e *stacks*;

```
void glutWireTorus (GLfloat innerRadius, GLfloat outerRadius, GLint nsides,
Glut rings)
```

Função usada para desenhar o wireframe de um torus. Os parâmetros indicam: o raio interno (*innerRadius*), o raio externo (*outerRadius*), o número de seções que serão utilizadas para formar o torus (*rings*) e o número de subdivisões para cada seção (*nsides*). Estes dois últimos parâmetros são usados para especificar as faces que compõem o objeto;

```
void glutWireIcosahedron (void)
```

Função usada para desenhar o wireframe de um icosaedro que possui um tamanho predefinido;

```
void glutWireOctahedron (void)
```

Função usada para desenhar o wireframe de um octaedro que possui um tamanho predefinido;

```
void glutWireTetrahedron (void)
```

Função usada para desenhar o wireframe de um tetraedro que possui um tamanho predefinido;

```
void glutWireDodecahedron (GLfloat radius)
```

Função usada para desenhar o wireframe de um dodecaedro cujo raio é especificado por parâmetro.

Por exemplo, na Listagem 7.1 – cujo resultado da execução aparece na Figura 7.12 – utilizamos a função *glutWireCube* para traçar um cubo. Empregando o mesmo

programa-fonte e substituindo a chamada dessa função por `glutWireTorus`, podemos facilmente traçar um *torus*. Para ilustrar a diferença dos parâmetros passados para essa função, a Figura 13.4 demonstra a utilização dos seguintes valores como parâmetros (nesta ordem): (a) (20, 35, 20, 40) e (b) (20, 35, 10, 8).

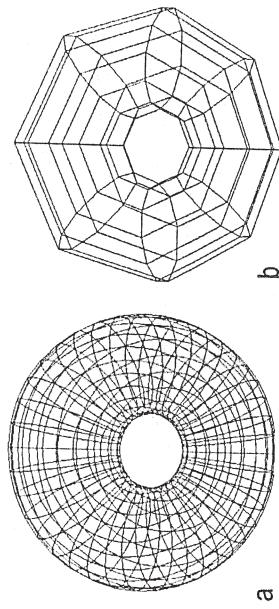


Figura 13.4 – Desenhando um *torus* com diferentes parâmetros (`Exemplo3DTorus.cpp`).

Todas essas funções também podem ser usadas para desenhar objetos sólidos, em vez de exibir apenas o seu *wireframe*. Para tanto, basta substituir a palavra *Wire* do nome da função por *Solid*. Por exemplo, se substituirmos a chamada à função `glutWireTorus` por `glutSolidTorus`, a imagem gerada é a apresentada na Figura 13.5. Entretanto, nesse caso, é necessário utilizar as técnicas de iluminação descritas no Capítulo 14 para que o objeto pareça realmente 3D.

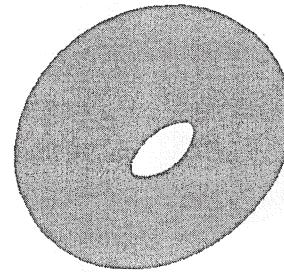


Figura 13.5 – Desenhando um *torus* por meio da função `glutSolidTorus` (`Exemplo3DTorus.cpp`).

13.3 Carregando e Desenhando Objetos

Dependendo da necessidade, pode ser inviável desenhar a cena 3D de uma aplicação utilizando apenas primitivas ou objetos predêfinitos da GLUT. Nesse caso, freqüentemente se utilizam modeladores mais sofisticados, que oferecem diversos recursos para a criação de objetos com maior complexidade. Existem diversos sis-

temas desse tipo, porém geralmente eles têm um custo considerável – alguns são as mesmas ferramentas utilizadas por empresas de efeitos especiais ou mesmo para criar filmes animados.

Todavia, há exceções: para a criação dos modelos utilizados pelos exemplos deste livro foi empregado o modelador Blender (<http://www.blender3d.org>). Pode-se definir o Blender como um sistema integrado de ferramentas, multiplataforma e código livre, capaz de modelar objetos, animar e gerar imagens com qualidade considerável. Ele é também capaz de importar e exportar objetos em diferentes formatos.

Um desses formatos é o **OBJ**, criado pela *Alias/Wavefront* e descrito no apêndice C. De maneira simplificada, um arquivo **OBJ** contém uma lista de vértices (**v**), uma lista de vetores normais (**vn**) e uma lista de faces (**f**). A Listagem 13.1 mostra um exemplo de arquivo nesse formato o qual possui diversas vantagens; as informações são armazenadas como texto ASCII, o arquivo é facilmente interpretável e finalmente, diversos sistemas são capazes de gerar e importar objetos assim codificados.

Listagem 13.1 – Exemplo de arquivo descrivendo um objeto 3D no formato **OBJ**.

```
# Blender OBJ File: tacaz2.blend
# www.blender.org
mtllib tacaz2.mtl
o Curve_Mesh
v -0.279468148947 0.434727907181 2.95124181093e-07
v 0.0217014085501 -0.323404312134 0.00838867100477
v -0.287736922503 0.3540815413 3.321268877e-07
v 0.278044521809 0.101156517863 0.115169532597
v -0.202977135777 -0.565464437008 0.202977397713
v -0.306754589081 0.25540657654 2.95576626286e-07
...
vn -0.9111379754543 0.163915902376 0.377516239882
vn -0.990027487278 0.14087445438 -1.35354475308e-16
vn -0.9546635005207 0.297682374716 -1.30519783298e-16
vn 0.906372725964 -0.193769291043 0.375422269106
vn 0.693705320358 -0.193767488003 0.693705320358
vn 0.672276139259 -0.3099831347 0.672276139259
...
usemtl Vidro
f 79//1 40//2 42//3
f 119//4 184//5 174//6
f 4//7 41//8 291//9
f 399//10 8//11 10//12
f 506//13 27//14 35//15
f 384//16 36//17 323//18
...
...
```

Um leitor do formato OBJ foi implementado como parte da `bibut1` (também descrita no Apêndice C), permitindo a utilização de objetos poligonais, com materiais diversos, vetores normais (ver Capítulo 14) e texturas (ver Capítulo 16). O exemplo apresentado a seguir mostra como se pode exibir o *wireframe* de um objeto carregado a partir de um arquivo OBJ.

ExemploWireframe3D.cpp

Seguindo o mesmo princípio demonstrado na seção 13.1, a `bibut1` define estruturas semelhantes para o armazenamento de vértices, faces e objetos. Porém, normalmente o usuário não precisa se preocupar com esses detalhes, bastando chamar a função para carga de objetos – denominada `CarregaObjeto`. Isso é feito durante a inicialização do programa (função `Inicializa`):

```

169. void Inicializa (void)
170. {
171.     char nomeArquivo[30];
172.
173.     // Define a cor de fundo da janela de visualização como branca
174.     glColor3f(1.0f, 1.0f, 1.0f);
175.
176.     // Inicializa a variável que especifica o ângulo da projeção
177.     // perspectiva
178.     angle=54;
179.
180.     // Inicializa as variáveis usadas para alterar a posição do
181.     // observador virtual
182.     rotX = 0;
183.     rotY = 0;
184.     obsX = obsY = 0;
185.     obsZ = 10;
186.
187.     // Lê o nome do arquivo e chama a rotina de leitura
188.     printf("Digite o nome do arquivo que contém o modelo 3D: ");
189.     gets(nomeArquivo);
190.
191.     // Carrega o objeto 3D
192.     objeto = CarregaObjeto(nomeArquivo, false);

```

A função retorna um apontador para a estrutura OBJ preenchida com as informações lidas do arquivo.

À seguir, é escolhido o modo de desenho, por meio da função `SetaModoDesenho`. A `bibut1` é capaz de desenhar objetos em *wireframe* (`w`), como sólidos (`s`) ou como sólidos utilizando texturas (`t`, se houver texturas associadas ao objeto, ver

Capítulo 16). No momento, como desejamos apenas exibir o *wireframe*, a função é chamada com o parâmetro `w`:

```

195.     SetaModoDesenho('w'); // 's' para sólido
196. }
```

A partir desse momento, o objeto pode ser desenhado pela chamada à `Desenha-Objeto`, o que ocorre dentro da *callback* de desenho:

```

029. void Desenha(void)
030. {
031.     // Limpa a janela de visualização com a cor
032.     // de fundo definida previamente
033.     glClear(GL_COLOR_BUFFER_BIT);
034.
035.     // Altera a cor do desenho para preto
036.     glColor3f(0.0f, 0.0f, 0.0f);
037.
038.     // Desenha o wireframe do objeto 3D lido do arquivo com a cor corrente
039.     DesenhaObjeto(objeto);
040.
041.     // Executa os comandos OpenGL
042.     glFlush();
043. }
```

O restante do código desse exemplo é muito semelhante ao apresentado na seção 13.1. A única diferença é que ele requer a digitação do nome do arquivo que contém a descrição do objeto 3D. Por exemplo, imaginando que o arquivo `taca2.obj` contém a descrição de uma taça 3D (já apresentado na Listagem 13.1), o resultado visual será aquele ilustrado na Figura 13.6.

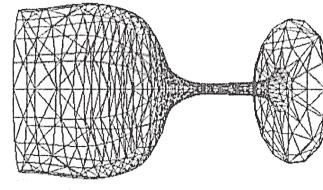


Figura 13.6 – Desenhando uma taça em *wireframe* a partir de um arquivo no formato OBJ (ExemploWireframe.cpp).

Para desenhar um objeto cujas faces sejam preenchidas, deve-se substituir a primitiva `GL_LINE_LOOP` por `GL_POLYGON`, `GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`, `GL_QUADS` ou `GL_QUAD_STRIP`, conforme a organização e o número de vértices de cada face que compõe o objeto. No caso do exemplo anterior, basta passar `s` (sólido) como parâmetro para a função `SetaModoDesenho` e a bibutí 1 utilizará a primitiva `GL_POLYGON`. Entretanto, esse procedimento não é suficiente para exibir uma imagem 3D. Conforme demonstra a Figura 13.7, é fácil notar que utilizando `GL_POLYGON` a imagem gerada ainda parece 2D. Isso ocorre porque as faces “preenchidas” devem ser utilizadas somente quando se está trabalhando com iluminação, pois, além da inserção da coordenada `z`, é necessário aplicar técnicas de realismo para que os objetos pareçam realmente 3D. Por exemplo, para que uma esfera vermelha pareça realmente uma esfera (e não um círculo plano), é necessário que suas faces sejam preenchidas com cores diferentes, simulando um efeito de iluminação. Este faz com que as faces que recebem a luz diretamente são exibidas com uma cor mais clara, e as faces que recebem menos luz são exibidas com uma cor mais escura. O Capítulo 14 explica detalhadamente como trabalhar com iluminação em OpenGL.

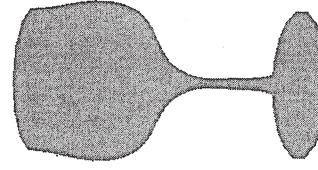


Figura 13.7 – Desenhando uma taça sólida a partir de um arquivo no formato OBJ (ExemploWireframe.cpp).

13.4 Desenhando Superfícies Paramétricas

Na seção 8.4 foi apresentado o processo de utilização dos avaliadores *Bézier* para o desenho de curvas. No entanto, a OpenGL também suporta avaliadores bidimensionais (isto é, que dependem de duas variáveis), com o objetivo de traçar superfícies paramétricas *Bézier* [Hearn, 2004].

Os avaliadores bidimensionais são definidos pela função `glMap2f` (ou `glMap2d`):

```
void glMap2f(GLenum target, GLfloat u1, GLfloat u2, GLint ustride, GLint uorder,
             GLfloat v1, GLfloat v2, GLint vstride, GLint vorder, GLsizei *points)
```

Função OpenGL para definir os pontos de controle e o tipo de um avaliador *Bézier* bidimensional. O parâmetro *target* define o significado dos pontos de controle, de acordo com a Tabela 13.1; os parâmetros *u1/v1* e *u2/v2* definem os intervalos para as variáveis de controle *u* e *v*; os parâmetros *ustride* e *vstride* indicam quantos valores *float* existem entre cada elemento do vetor no sentido horizontal e vertical; os parâmetros *uorder* e *vorder* devem conter a quantidade de elementos nos dois sentidos. Finalmente, o parâmetro *points* deve ser um apontador para a primeira coordenada do primeiro ponto de controle.

Tabela 13.1 – Significado dos pontos de controle em `glMap2f/2d`

Constante	Significado
<code>GL_MAP2_VERTEX_3</code>	Coordenadas <i>x, y e z</i> .
<code>GL_MAP2_VERTEX_4</code>	Coordenadas <i>x, y, z e w</i> .
<code>GL_MAP2_INDEX</code>	Índice de cor.
<code>GL_MAP2_COLOR_4</code>	Cor como RGBA.
<code>GL_MAP2_NORMAL</code>	Vetor normal.
<code>GL_MAP2_TEXTURE_COORD_1</code>	Coord. de textura <i>s</i> .
<code>GL_MAP2_TEXTURE_COORD_2</code>	Coord. de textura <i>s e t</i> .
<code>GL_MAP2_TEXTURE_COORD_3</code>	Coord. de textura <i>s, t e r</i> .
<code>GL_MAP2_TEXTURE_COORD_4</code>	Coord. de textura <i>s, t, q e q</i> .

Novamente, os avaliadores desejados devem ser habilitados por meio de `glEnable` e o parâmetro correspondente. O primeiro passo, como no exemplo de curvas, é definir os pontos de controle em uma matriz:

ExemploSuperfície.cpp

```
027. GLfloat pontos[4][4][3] = {
028.     {{0.0, 0.0, 0.0}, {0.3, 0.5, 0.0}, {0.7, 0.5, 0.0}, {1.0, 0.0, 0.0}},
029.     {{0.0, 0.0, 0.3}, {0.3, 0.5, 0.3}, {0.7, 0.5, 0.3}, {1.0, 0.0, 0.3}},
030.     {{0.0, 0.0, 0.7}, {0.3, 0.5, 0.7}, {0.7, 0.5, 0.7}, {1.0, 0.0, 0.7}},
031.     {{0.0, 0.0, 1.0}, {0.3, 0.5, 1.0}, {0.7, 0.5, 1.0}, {1.0, 0.0, 1.0}}
032. };
```

Repare que cada linha da matriz tem uma coordenada diferente, mas as demais coordenadas são as mesmas da curva apresentada no Capítulo 8. Isso foi feito para facilitar o entendimento, uma vez que pode ser complicado visualizar mentalmente os pontos de controle no espaço. Dessa forma, a superfície gerada é simplesmente uma extrusão da curva original. Depois, os pontos são enviados para a OpenGL na função de inicialização:

```

212. void Inicializa (void)
213. {
214.   // Define significado dos pontos de controle
215.   glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4, 0, 1, 12, 4, &pontos[0][0][0]);
216.   // Ativa geração de coordenadas
217.   glEnable(GL_MAP2_VERTEX_3);

```

Observe a utilização da constante `GL_MAP2_VERTEX_3`, já que o objetivo é apenas obter a posição de cada ponto intermediário. A chamada à `glMap2f` deve ser analisada com cuidado, especialmente com relação aos parâmetros `ustride` e `vstride`:

```

91Map2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4, 0, 1, 12, 4, &pontos[0][0][0]);

```

Note que é utilizado o valor 3 para `ustride`, ou seja, na coordenada `u` o elemento seguinte está a três `floats` de distância na estrutura (pois a matriz armazena coordenadas `x,y,z`). Porém, o valor de `vstride` é 12, já que, para chegar no próximo elemento na direção da coordenada `v`, é preciso passar por todos os valores `float` da linha atual (ou seja, 4 valores * 3 coordenadas = 12 `floats`). Já tanto os parâmetros `uorder` como `vorder` são sempre 4, representando as dimensões da matriz.

Analogamente ao caso de curvas, a função responsável pela avaliação de uma *Bézier* bidimensional é `glEvalCoord2f`:

```

void glEvalCoord2f(GLfloat u, GLfloat v)

```

Função OpenGL para realizar a avaliação da *Bézier* por meio de todos os avaliadores definidos por `glMap2f` e ativados por `glEnable`. Os parâmetros `u` e `v` indicam os valores a serem passados para a *Bézier*, normalmente entre 0 (início da curva) e 1 (final da curva). Há também a variação `glEvalCoord2d`, para valores *double*.

No exemplo, a função é chamada na rotina *callback* de desenho (`Desenha`):

```

041. void Desenha(void)
042. {
...
050.   // Calcula incremento de acordo com o total
051.   // de pontos intermediários
052.   float delta = 1.0/(float)prec;
053.
054.   // Traça a superfície
055.   for(float j=0; j<=1.0f; j+=delta)
056.   {
057.     glBegin(GL_LINE_STRIP);
058.     for(float i=0; i<=1.0f; i+=delta)
059.       glEvalCoord2f(i,j);
060.   }

```

```

061.   glEnd();
062.   for(float i=0; i<1.0f; i+=delta)
063.     glEvalCoord2f(j,i);
064.   glEnd();
065. }

```

Mais uma vez, empregamos a variável `prec` para definir a quantidade de pontos intermediários `e`, consequentemente, para calcular o incremento a ser aplicado em cada ponto (`delta`). Entretanto, o processo de desenho é também um pouco mais complexo: não basta simplesmente percorrer os pontos ao longo de `u` e `v`, pois queremos traçar linhas em ambos os sentidos. Assim, é preciso fazer uma repetição definida pela variável `j` e dentro desta, fazer duas repetições idênticas utilizando a variável `i` – porém alternando o uso de `i` e `j` na chamada à função `glEvalCoord2f`.

O trecho final serve apenas para desenhar os pontos de controle:

```

067.   // Muda a cor para vermelho
068.   glColor3f(1.0f, 0.0f, 0.0f);
069.
070.   // Define tamanho de um ponto
071.   glPointSize(5.0);
072.
073.   // Desenha os pontos de controle
074.   glBegin(GL_POINTS);
075.   for(int i=0; i<4; ++i)
076.     for(int j=0; j<4; ++j)
077.       glVertex3fv(pontos[i][j]);
078.   glEnd();
079.
080.   // Executa os comandos OpenGL
081.   glFlush();
082. }

```

Aqui também pode ser usada uma grade, com o objetivo de simplificar consideravelmente o código de desenho. Mais uma vez, considerando que a distância entre os pontos intermediários é sempre a mesma, a função `glMapGrid2f` define uma grade bidimensional:

```

void glMapGrid2f(GLint un, GLfloat u1, GLfloat u2, GLint vn, GLfloat v1, GLfloat v2)

```

Define uma grade bidimensional de `un` x `vn` pontos intermediários, espacados igualmente, entre `u1/v1` e `u2/v2`. Também existe a variação para *double*: `glMapGrid2d`.

Capítulo 14

Realismo

O procedimento de desenho é agora realizado pela função `glEvalMesh2`:

```
void glEvalMesh2(GLenum mode, GLint i1, GLint j1, GLint j2)
```

Aplica a grade bidimensional definida anteriormente a todos os avaliadores Bézier habilitados. O parâmetro *mode* indica o tipo de primitiva a ser desenhada (`GL_FILL`, `GL_POINT` ou `GL_LINE`); os parâmetros *i1/j1* e *i2/j2* indicam o intervalo de pontos intermediários a ser utilizado (de 0 ao valor passado nos parâmetros *un/vn* durante a chamada de `glMapGrid2f`). A primitiva `GL_FILL` solicita o desenho de polígonos, em vez de linhas ou pontos.

Portanto, para desenhar a superfície com linhas, basta substituir o conjunto de repetições mostrado anteriormente pelas seguintes chamadas:

```
glMapGrid2f(prec,0,1,prec,0,1);
glEvalMesh2(GL_LINE,0,prec,0,prec);
```

A Figura 13.8 demonstra o resultado do programa, onde mais uma vez apresenta-se como a quantidade de pontos intermediários influencia na qualidade da superfície. Os comandos de teclado são os mesmos, ou seja, + para aumentar e - para reduzir a quantidade de pontos intermediários.

No mundo real, para que seja possível enxergar objetos em um ambiente, é fundamental que exista pelo menos uma fonte de luz. De forma simplificada, objetos são visíveis porque refletem e absorvem raios de luz. Portanto, a noção de uma cena 3D com realismo está intimamente ligada à idéia de iluminação, pois os pontos na superfície de um objeto iluminado possuem diferentes tonalidades de acordo com a luz recebida. Como na Computação Gráfica procura-se simular o que acontece na realidade, é necessário incluir fontes de luz no ambiente virtual 3D e descrever como será a interação dessas fontes com os demais objetos da cena.

Neste capítulo são abordadas as idéias necessárias para entender o funcionamento das funções OpenGL que permitem gerar imagens 3D com um certo grau de realismo. Na seção 14.1 são descritos alguns conceitos fundamentais para se trabalhar com luzes em ambientes 3D, enquanto as funções OpenGL correspondentes são apresentadas na seção 14.2. Um exemplo completo de como implementar um programa utilizando iluminação é introduzido na seção 14.3. O cálculo de vetores normais, essencial para determinar a cor de cada ponto que forma a superfície de um objeto, é discutido na seção 14.4. Já a seção 14.5 apresenta detalhes sobre a utilização de cores em OpenGL. Os comandos para incluir e posicionar múltiplas fontes de luz, além da maneira como ocorre a interação dos raios de luz com os objetos, são abordados, respectivamente, nas seções 14.6 e 14.7.

Finalmente, é introduzido o programa-exemplo Sal3D – uma aplicação completa e que será construída passo-a-passo ao longo dos demais capítulos. Outras técnicas de realismo, tais como a geração de objetos transparentes e a aplicação de texturas, são descritas, respectivamente, nos capítulos 15 e 16.

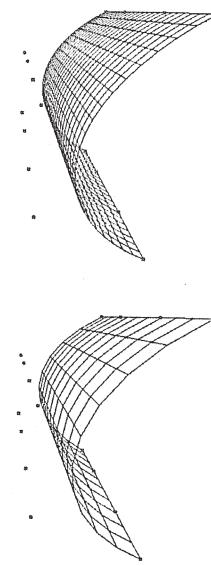


Figura 13.8 – Desenhando superfícies paramétricas com menos (esquerda) e mais (direita) pontos intermediários (ExemploSuperfície.cpp).

Assim como as curvas, também é possível definir superfícies baseadas em NURBS, mas, nesse caso, o processo é muito mais complexo – porém oferecendo mais facilidades de manipulação e combinação de várias superfícies.