



Computação Gráfica

**Trabalho Prático CG - 2017/18**

**Fase 1 - Primitivas Gráficas**

Relatório de Desenvolvimento

André Pereira  
(A79196)

Filipe Miranda  
(A78992)

José Alves  
(A78178)

9 de Março de 2018

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>2</b>
<b>2</b>	<b>Análise e Especificação</b>	<b>3</b>
2.1	Descrição informal do problema . . . . .	3
2.2	Especificação dos Requisitos . . . . .	3
2.2.1	Dados . . . . .	3
2.2.2	Pedidos . . . . .	3
<b>3</b>	<b>Concepção/desenho da Resolução</b>	<b>4</b>
3.1	Gerador . . . . .	4
3.1.1	Plano . . . . .	4
3.1.2	Caixa . . . . .	5
3.1.3	Esfera . . . . .	6
3.1.4	Cone . . . . .	6
3.1.5	Toroid (Quadrangular) . . . . .	7
3.1.6	Exemplo do Gerador . . . . .	7
3.2	Motor . . . . .	9
3.2.1	Exemplo do Motor . . . . .	9
<b>4</b>	<b>Conclusão</b>	<b>10</b>
<b>A</b>	<b>Código do Programa</b>	<b>11</b>

# Capítulo 1

## Introdução

O trabalho prático **Primitivas Gráficas** surge inserido na disciplina de Computação Gráfica e corresponde à primeira fase do trabalho prático desta unidade curricular. Versa sobre o uso da ferramenta **GLUT** e divide-se essencialmente em duas componentes, a criação de um gerador e a criação de um motor. Ao longo deste documento podemos encontrar a exposição do problema (a sua descrição informal, os dados do problema e o que foi pedido que concebêssemos), o desenho da resolução do problema, alguns problemas de implementação que nso surgiram e alguns resultados/testes. Por fim, apresenta-se uma breve conclusão e o código do programa.

## Estrutura do Relatório

Este relatório inicializa-se com uma breve introdução ao trabalho prático. No capítulo 2 é feita uma análise do problema proposto, onde se apresenta o enunciado do trabalho e a especificação dos seus requisitos. De seguida, são apresentadas as estratégias para a resolução do mesmo, onde no capítulo ?? são especificadas as decisões tomadas, os problema que surgiram ao longo da realização do trabalho prático e os resultados obtidos. Finalizando, no capítulo 4 encontram-se as conclusões a que chegamos e o trabalho futuro que poderá vir a ser efetuado.

## Capítulo 2

# Análise e Especificação

### 2.1 Descrição informal do problema

Esta fase do trabalho requer dois programas: um que gere ficheiros com a informação dos modelos (onde, nesta fase, apenas são gerados os vértices para os modelos) e outro que lê um ficheiro de configuração, escrito em XML, que exhibirá os modelos.

### 2.2 Especificação dos Requisitos

#### 2.2.1 Dados

Para criar os ficheiros modelo, o programa tem que receber como parâmetros o tipo da primitiva gráfica, especificações para a criação do modelo e o ficheiro de destino onde serão guardados os vértices. O formato do ficheiro é arbitrário e pode conter informação adicional que assista a leitura do mesmo. Após isso, o programa/mecanismo recebe um ficheiro de configuração, escrito em XML. Nesta fase, o ficheiro XML contém apenas a indicação de quais os ficheiros previamente gerados é que serão carregados.

#### 2.2.2 Pedidos

Nesta fase são pedidas as seguintes primitivas gráficas:

- Plano (um quadrado no plano XZ, com centro na origem e feito com dois triângulos)
- Cubo (requer dimensões X, Y e Z e, opcionalmente, o número de divisões)
- Esfera (requer raio, slices e stacks)
- Cone (requer raio da base, altura, slices e stacks)

## Capítulo 3

# Concepção/desenho da Resolução

### 3.1 Gerador

O gerador recebe como argumentos a figura que vai desenhar (em inglês) e os devidos parâmetros de cada uma, onde, posteriormente irá escrever os pontos necessários num ficheiro para desenhar a figura. Cada ficheiro gerado é nomeado consoante a figura que foi passada como parâmetro e, em termos de conteúdo, os pontos que foram escritos estão organizados em blocos de três. O gerador está abilitado a lidar com anomalias que possam ter ocorrido no input e informa o utilizador com a respetiva mensagem de erro. Por exemplo, caso seja introduzida uma figura que o gerador não reconhece (ou que esteja mal escrita) é enviada a mensagem "Main: Shape not found!".

#### 3.1.1 Plano

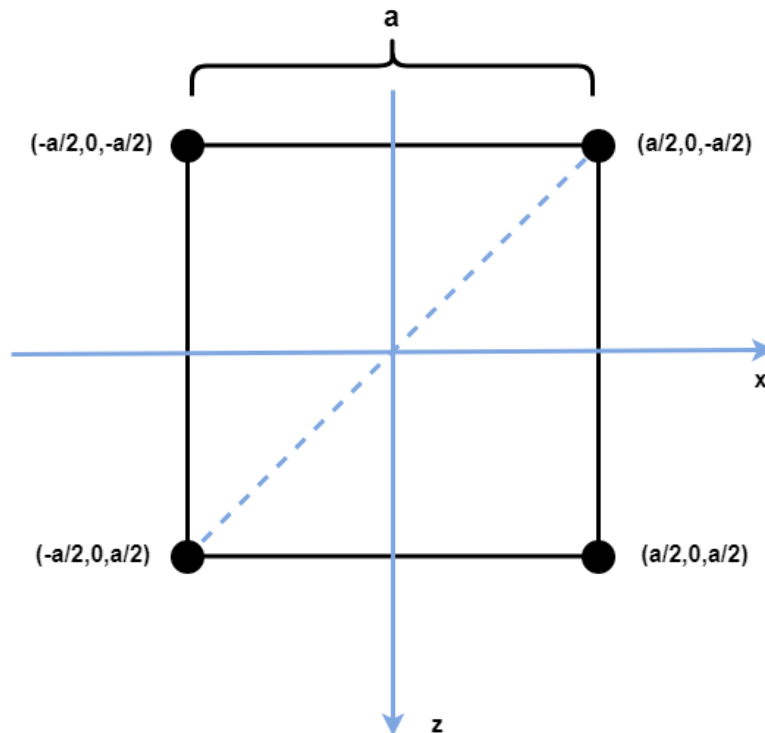


Figura 3.1: Plano

O plano é finito e centrado na origem do referencial. Uma vez que o plano é um objeto geométrico a duas dimensões (e na especificação dos requisitos é pedido que o quadrado esteja no plano XZ), a sua componente do eixo yy é sempre zero. Como se encontra exemplificado na figura 3.1, o plano pode ser desenhado através de dois triângulos. Sendo a origem do referencial o centro do plano e sabendo quanto mede um dos lados do quadrado, os pontos são determinados consoante o que está apresentado na 3.1 e são desenhados tendo em conta a regra da mão direita. No gerador, a função **void plane(float x)** recebe o comprimento de um dos lados e gera um ficheiro com os pontos para desenhar o plano.

### 3.1.2 Caixa

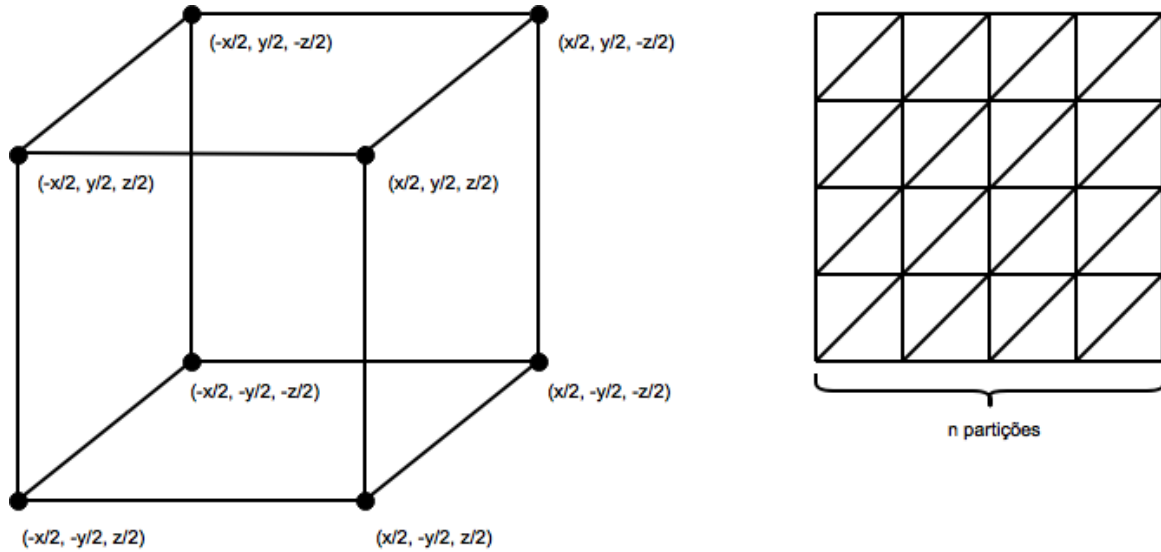


Figura 3.2: Caixa

Uma caixa encontra-se centrada na origem do referencial e contém 3 medidas: o comprimento, a largura e a altura. A figura é composta por seis faces, onde cada uma delas é um plano finito, e pode ou não ter partições.

- Se o número de partições for dado como zero ao gerador, então cada uma das faces será desenhada conforme o plano da Figura 3.1, onde cada uma dela pode ser contruída com dois triângulos. É necessário passar as dimensões como parâmetros ao gerador e, conforme esses valores, é desenhado o modelo da caixa, tendo em conta as coordenadas que se encontram na Figura 3.2. A função **void box(float x, float z, float y, int divisions)** encarrega-se de escrever os pontos num ficheiro com o mesmo nome para desenhar a caixa (neste caso, divisions é igual a zero).
- Se tiverem sido passadas n partições como parâmetro, então cada face estará dividida em n partes iguais vertical e horizontalmente. Consequentemente, cada uma das partições da face é constituída por dois triângulos e o processo aplica-se a todas as divisões da face e a todas as faces da caixa.

### 3.1.3 Esfera

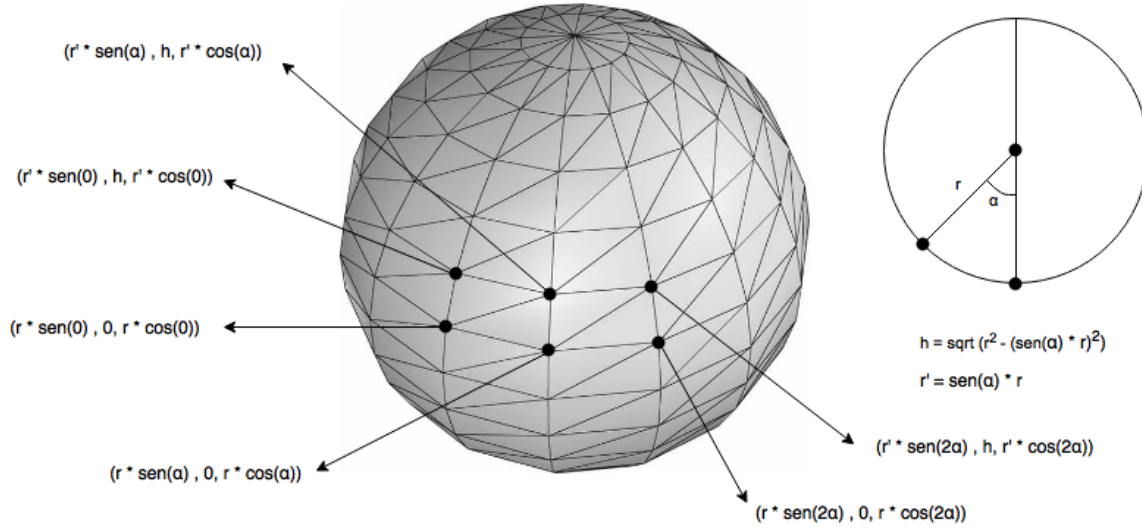


Figura 3.3: Esfera

A esfera é uma superfície fechada de tal forma que todos os pontos dela estão à mesma distância de seu centro (que neste caso se encontra na origem do referencial). No gerador, a função **void sphere(float radius, int slices, int stacks)** recebe a medida do raio da esfera, o número de slices e o número de stacks que a constituem. Sabendo o raio da esfera e o número de slices que foram passados como argumento para o gerador, cada ponto é determinado consoante o esquema que se encontra na Figura 3.8. O corpo da esfera é desenhado consoante o número de stacks que foi passado como argumento, onde são desenhadas novas circunferências duas a duas a partir do centro, de diâmetro consecutivamente mais pequeno à medida que vão chegando aos limites da esfera. Ao percorrermos todas as slices e todas as stacks, estas interseitam-se sempre, formando quadrados que podem, mais uma vez, ser divididos em dois triângulos, como se pode ver na Figura 3.8.

### 3.1.4 Cone

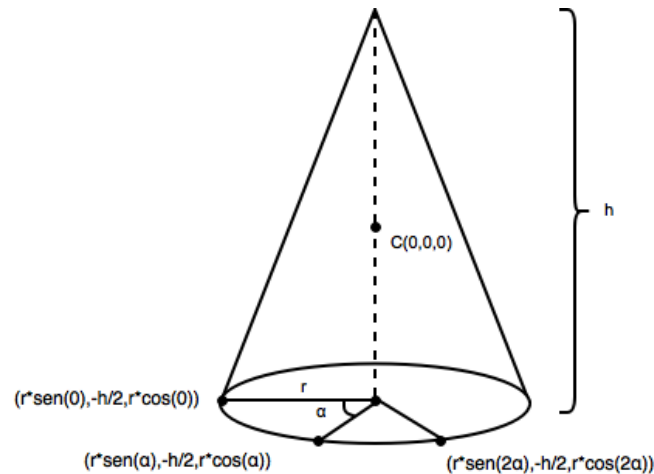


Figura 3.4: Cone



No gerador, a função **void cone(float radius, float height, int slices, int stacks)** recebe a medida do raio da circunferência que forma a base, a altura do cone, o número de slices e o número de stacks. Para determinar os pontos do cone, foi preciso tratar da base da figura e do seu corpo. Tendo em conta que a origem se encontra no centro da figura, a coordenada y de todos os pontos da base será -h/2. Sabendo o raio da circunferência e o número de slices que foram passados como argumento para o gerador, cada ponto é determinado consoante o esquema que se encontra na Figura 3.4, onde  $\alpha = \frac{2\pi}{slices}$ ,  $\forall 0 < \alpha \leq 2\pi$ . O corpo do cone é desenhado consoante o número de stacks que foi passado como argumento, onde, ao longo da altura do cone, são desenhadas novas circunferências, de diâmetro consecutivamente mais pequeno, e é feita a junções das laterais.

### 3.1.5 Toroid (Quadrangular)

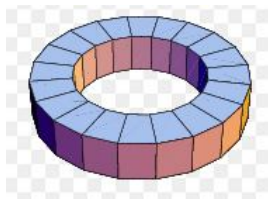


Figura 3.5: Toroid

Foi ainda, acrescentada a geração de um modelo adicional. Um tipo de toroid que é composto por n anéis, em que cada um desses é composto por 4 lados, com uma determinada altura e um determinado comprimento. Os parâmetros para a geração deste modelo, são o raio desde o centro até à camada interior da figura, o raio até à camada exterior do modelo, o número de anéis e a altura de cada anél.

Para a geração desta figura geométrica, iteramos sobre o número de anéis. Sabemos o ângulo de cada anél ( $360^\circ / \text{rings}$ ). Portanto para cada iteração criamos 4 lados (o topo a base e o lado interior e exterior), a altura desde o topo e a base é constante, é passada como argumento na função. A distância desde a camada interior até à exterior é calculada como a diferença entre o segundo e o primeiro raio passados como parâmetro. O comprimento dos lados interiores e exteriores também são facilmente calculados a partir do ângulo de cada anél.

Estamos assim em condições de gerar um modelo similar ao da figura 3.5.

### 3.1.6 Exemplo do Gerador

De seguida, apresentamos um exemplo do funcionamento do gerador, quando aplicado a um cone de raio 2, altura 5, com 10 slices e 10 stacks. Como se pode ver na Figura 3.6 foi criado um ficheiro *cone* onde foram escritos os pontos necessários para desenhar o cone, onde é possível ver uma amostra dos pontos na Figura 3.7.

```
if (strcmp(argv[1], "cone") == 0) {  
    if (argc != 6) {  
        printErrorArgs(argv[1]);  
        _exit(0);  
    }  
    printf("cone\n");  
    generate(argv[1], argv + 2, arg  
    return 0;  
}  
if (strcmp(argv[1], "sphere") == 0)  
    if (argc != 5) {  
        printErrorArgs(argv[1]);  
        _exit(0);  
    }  
    printf("sphere\n");  
    generate(argv[1], argv + 2, arg  
    return 0;  
}  
printf("Main: Shape not found!\n");  
return 0;
```

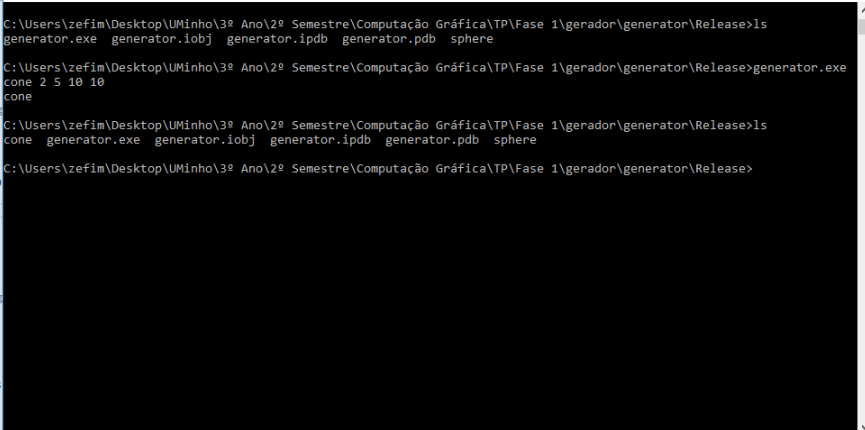


Figura 3.6: Execução do gerador aplicado a um cone

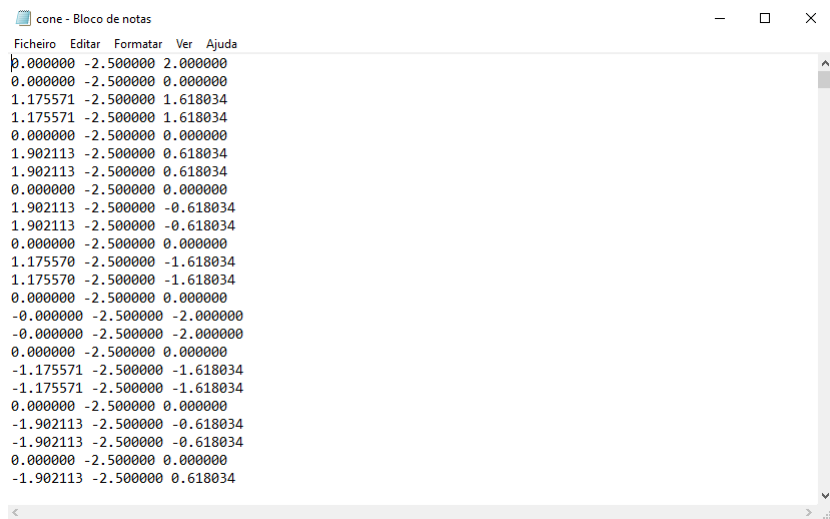


Figura 3.7: Conteúdo parcial do ficheiro *cone*

## 3.2 Motor

Para executar o motor, é-lhe passado como argumento um ficheiro de configuração XML do género da figura 3.8. O motor, juntamente com o parser de XML, lê o ficheiro e vai tentar carregar os ficheiros com extensão .3d que foram gerados pelo gerador. Ao carregar estes ficheiros, o motor lê cada linha (constituída por três valores, que correspondem às coordenadas de um ponto) e faz glVertex dessas coordenadas, onde, a cada triplo de coordenadas que encontrar vai desenhar um triângulo. Como forma de auxílio à leitura, denotamos que está presente o número de vértices que cada ficheiro contém no início. O motor permite ainda a rotação das figuras desenhadas através da movimentação da câmara, com recurso ao teclado, nomeadamente com as teclas "Left", "Right", "Up", "Down". É possível ainda, afastar ou aproximar a câmara dos modelos desenhados permitindo, respetivamente, "F1" e "F2".

### 3.2.1 Exemplo do Motor

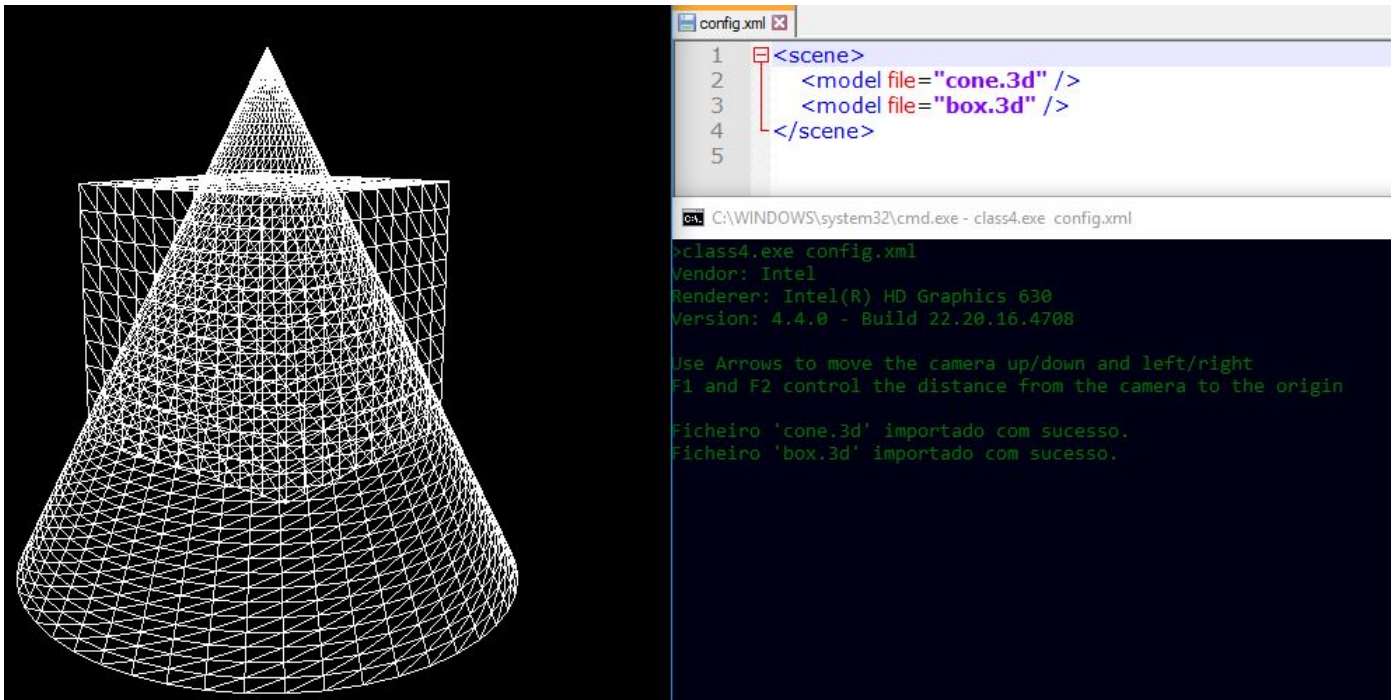


Figura 3.8: Arranque do motor

## Capítulo 4

# Conclusão

O objetivo desta primeira fase do trabalho prático de Computação Gráfica consistiu no desenvolvimento de dois programas: um gerador de ficheiros cujo conteúdo corresponde a todos os pontos que formam um determinado modelo geométrico e um motor que leia ficheiros de configuração que permita expor esses mesmos modelos. De maneira a atingir este objetivo, inicialmente foi feito um estudo aos conteúdos que têm sido abordados nas aulas teóricas e práticas, de maneira a sabermos como deveríamos proceder para o desenvolvimento destes programas. A primeira parte consistiu em desenvolver algoritmos que desenhasssem as diversas figuras geométricas, semelhante ao mecanismo que está por trás do **GLUT**, enquanto que a segunda se focou em implementar estes algoritmos a um gerador que recebe os parâmetros de cada uma das primitivas gráficas e, posteriormente, gere todos os pontos necessários num ficheiro para as desenhar, ao passar por um motor que leia ficheiros escritos em XML. Nesta fase, como extra, pretendíamos implementar a primitiva gráfica do Torus, porém, devido a problemas de implementações no desenvolvimento do motor e com as bibliotecas do tinyxml, concebemos um toroid, que acaba por ser uma espécie de Torus mais fácil de obter. Fora isso, achamos que conseguimos atingir o nosso objetivo e expressamos o nosso contentamento em finalizar a primeira fase de um trabalho prático que se revela complexo e que exige muitas horas de trabalho.

## Apêndice A

# Código do Programa

Lista-se a seguir o código do programa que foi desenvolvido para gerar cada uma das primitivas gráficas requeridas.

---

```
1 void plane(int x = 1) {
2
3     if ((x = abs(x)) == 0) x = 1;
4
5     glBegin(GL_TRIANGLES);
6
7
8     glVertex3f(-x / 2, 0, x / 2);
9     glVertex3f(x / 2, 0, x / 2);
10    glVertex3f(x / 2, 0, -x / 2);
11
12    glVertex3f(x / 2, 0, -x / 2);
13    glVertex3f(-x / 2, 0, -x / 2);
14    glVertex3f(-x / 2, 0, x / 2);
15
16    glEnd();
17 }
18
19 void box(float x, float z, float y, int divisions = 0) {
20
21
22     if ((divisions = abs(divisions)) == 0) divisions = 1;
23
24     float xaux = -x / 2;
25     float zaux = z / 2;
26     float yaux = y / 2;
27
28     //frente
29
30     glBegin(GL_TRIANGLES);
31     glColor3f(0.5, 0.8, 1);
32
33     for (int j = 0; j < divisions; j++) {
34
35         for (int i = 0; i < divisions; i++) {
36
37             glVertex3f(xaux, yaux, z / 2);
38             glVertex3f(xaux, yaux - (y / divisions), z / 2);
39             glVertex3f(xaux+(x/divisions), yaux - (y / divisions), z / 2);
```

```

40
41         glVertex3f(xaux, yaux, z / 2);
42         glVertex3f(xaux + (x / divisions), yaux - (y / divisions), z / 2);
43         glVertex3f(xaux + (x / divisions), yaux, z / 2);
44
45         xaux += (x / divisions);
46     }
47
48     xaux = -x / 2;
49     yaux = yaux - (y / divisions);
50 }
51 glEnd();
52
53
54
55 yaux = y / 2;
56 xaux = -x / 2;
57 zaux = z / 2;
58
59 //atras
60 glBegin(GL_TRIANGLES);
61 glColor3f(0.5, 0.8, 1);
62
63 for (int j = 0; j < divisions; j++) {
64
65     for (int i = 0; i < divisions; i++) {
66
67         glVertex3f(xaux + (x / divisions), yaux, -z / 2);
68         glVertex3f(xaux + (x / divisions), yaux - (y / divisions), -z / 2);
69         glVertex3f(xaux, yaux - (y / divisions), -z / 2);
70
71         glVertex3f(xaux, yaux - (y / divisions), -z / 2);
72         glVertex3f(xaux, yaux, -z / 2);
73         glVertex3f(xaux + (x / divisions), yaux, -z / 2);
74
75         xaux += (x / divisions);
76     }
77
78     xaux = -x / 2;
79     yaux = yaux - (y / divisions);
80 }
81 glEnd();
82
83
84 xaux = x / 2;
85 zaux = z / 2;
86 yaux = y / 2;
87
88 //dir
89 glBegin(GL_TRIANGLES);
90 glColor3f(0.5, 0.8, 1);
91
92 for (int j = 0; j < divisions; j++) {
93
94     for (int i = 0; i < divisions; i++) {
95
96         glVertex3f(xaux, yaux, zaux);
97         glVertex3f(xaux, yaux - (y / divisions), zaux);
98         glVertex3f(xaux, yaux - (y / divisions), zaux - (z / divisions));

```

```

99
100         glVertex3f(xaux, yaux, zaux);
101         glVertex3f(xaux, yaux - (y / divisions), zaux - (z / divisions));
102         glVertex3f(xaux, yaux, zaux - (z / divisions));
103
104         zaux -= (z / divisions);
105     }
106
107     zaux = z / 2;
108     yaux = yaux - (y / divisions);
109 }
110 glEnd();
111
112
113
114
115 xaux = -x / 2;
116 zaux = z / 2;
117 yaux = y / 2;
118
119 // esq
120 glBegin(GL_TRIANGLES);
121 glColor3f(0.5, 0.8, 1);
122
123 for (int j = 0; j < divisions; j++) {
124
125     for (int i = 0; i < divisions; i++) {
126
127
128         glVertex3f(xaux, yaux, zaux - (z / divisions));
129         glVertex3f(xaux, yaux - (y / divisions), zaux);
130         glVertex3f(xaux, yaux, zaux);
131
132         glVertex3f(xaux, yaux, zaux - (z / divisions));
133         glVertex3f(xaux, yaux - (y / divisions), zaux - (z / divisions));
134         glVertex3f(xaux, yaux - (y / divisions), zaux);
135
136         zaux -= (z / divisions);
137     }
138
139     zaux = z / 2;
140     yaux = yaux - (y / divisions);
141 }
142 glEnd();
143
144
145
146
147 xaux = -x / 2;
148 zaux = -z / 2;
149 yaux = y / 2;
150
151 // topo
152 glBegin(GL_TRIANGLES);
153 glColor3f(0.5, 0.8, 1);
154
155 for (int j = 0; j < divisions; j++) {
156
157     for (int i = 0; i < divisions; i++) {

```

```

158
159         glVertex3f(xaux, yaux, zaux);
160         glVertex3f(xaux, yaux, zaux + (z / divisions));
161         glVertex3f(xaux + (x / divisions), yaux, zaux + (z / divisions));
162
163         glVertex3f(xaux, yaux, zaux);
164         glVertex3f(xaux + (x / divisions), yaux, zaux + (z / divisions));
165         glVertex3f(xaux + (x / divisions), yaux, zaux);
166
167         xaux += (x / divisions);
168     }
169
170     xaux = -x / 2;
171     zaux = zaux + (y / divisions);
172 }
173 glEnd();
174
175
176 xaux = -x / 2;
177 zaux = -z / 2;
178 yaux = -y / 2;
179
180 //base
181 glBegin(GL_TRIANGLES);
182 glColor3f(0.5, 0.8, 1);
183
184 for (int j = 0; j < divisions; j++) {
185
186     for (int i = 0; i < divisions; i++) {
187
188         glVertex3f(xaux, yaux, zaux);
189         glVertex3f(xaux + (x / divisions), yaux, zaux + (z / divisions));
190         glVertex3f(xaux, yaux, zaux + (z / divisions));
191
192         glVertex3f(xaux, yaux, zaux);
193         glVertex3f(xaux + (x / divisions), yaux, zaux);
194         glVertex3f(xaux + (x / divisions), yaux, zaux + (z / divisions));
195
196         xaux += (x / divisions);
197     }
198
199     xaux = -x / 2;
200     zaux = zaux + (y / divisions);
201 }
202 glEnd();
203
204 }
205
206 void cone(float radius, float height, int slices, int stacks) {
207
208     float angulo = (2 * M_PI) / slices;
209     float baseaux = -height / 2;
210     float raioaux = radius;
211     float raioaux2 = radius - (radius / stacks);
212
213     //DESENHA BASE
214     glBegin(GL_TRIANGLES);
215
216     glColor3f(1, 0.5, 0.5);

```



```

217
218
219     for (int i = 0; i<slices; i++) {
220         glVertex3f(radius*sin(angulo*i), baseaux, radius*cos(angulo*i));
221         glVertex3f(0.0f, baseaux, 0.0f);
222         glVertex3f(radius*sin(angulo*(i + 1)), baseaux, radius*cos(angulo*(i + 1)));
223     }
224
225     glEnd();
226
227
228
229     //DESENHA STACKS
230     for (int j = 0; j < stacks-1; j++) {
231
232         glBegin(GL_TRIANGLES);
233
234         glColor3f(0.5, 0.8, 1);
235
236         for (int i = 0; i < slices; i++) {
237
238
239             glVertex3f(raioaux*(sin(angulo*i)), baseaux, raioaux*(cos(angulo*i))
240             );
241             glVertex3f(raioaux2*(sin(angulo*(i + 1))), baseaux + (height /
242             stacks), raioaux2*(cos(angulo*(i + 1))));
243             glVertex3f(raioaux2*(sin(angulo*i)), baseaux + (height / stacks),
244             raioaux2*(cos(angulo*i)));
245
246             glVertex3f(raioaux*(sin(angulo*i)), baseaux, raioaux*(cos(angulo*i))
247             );
248             glVertex3f(raioaux*(sin(angulo*(i + 1))), baseaux , raioaux*(cos(
249             angulo*(i + 1))));
250             glVertex3f(raioaux2*(sin(angulo*(i + 1))), baseaux + (height /
251             stacks), raioaux2*(cos(angulo*(i + 1))));
252
253         }
254
255         glEnd();
256
257         baseaux += height / stacks;
258         raioaux = raioaux2;
259         raioaux2 -= (radius / stacks);
260     }
261
262     glBegin(GL_TRIANGLES);
263
264     glColor3f(0.5, 0.8, 1);
265
266     for (int i = 0; i < slices; i++) {
267
268         glVertex3f(raioaux*(sin(angulo*i)), baseaux, raioaux*(cos(angulo*i)));
269         glVertex3f(raioaux2*(sin(angulo*(i + 1))), baseaux + (height / stacks),
270         raioaux2*(cos(angulo*(i + 1))));
271         glVertex3f(raioaux2*(sin(angulo*i)), baseaux + (height / stacks), raioaux2*(

```

```

269         cos(angulo*i)));
270     glVertex3f(raioaux*(sin(angulo*i)), baseaux, raioaux*(cos(angulo*i)));
271     glVertex3f(raioaux*(sin(angulo*(i + 1))), baseaux, raioaux*(cos(angulo*(i +
272         1))));
273     glVertex3f(raioaux2*(sin(angulo*(i + 1))), baseaux + (height / stacks),
274         raioaux2*(cos(angulo*(i + 1))));
275 }
276
277 glEnd();
278
279 }
280
281 void sphere(float radius, int slices, int stacks) {
282     float alfa = M_PI / stacks;
283     float alfaux = M_PI / stacks;
284     float altura = 0;
285     float altura2 = sin(alfa) * radius;
286     float angulo = (2 * M_PI) / slices;
287     float raioaux = radius;
288     float raioaux2 = sqrtf((radius*radius) - (altura2 * altura2));
289
290     if (stacks % 2 == 1) {
291
292         alfa = alfa / 2;
293         altura = sin(alfa) * radius;
294         altura2 = sin(alfa + alfaux) * radius;
295         raioaux = sqrtf((radius*radius) - (altura * altura));
296         raioaux2 = sqrtf((radius*radius) - (altura2 * altura2));
297
298         glBegin(GL_TRIANGLES);
299
300         glColor3f(0.5, 0.8, 1);
301
302         for (int i = 0; i < slices; i++) {
303
304             glVertex3f(raioaux*(sin(angulo*i)), -altura, raioaux*((cos(angulo*i)
305                 )))
306             glVertex3f(raioaux*(sin(angulo*(i + 1))), -altura, raioaux*((cos(
307                 angulo*(i + 1)))))
308             glVertex3f(raioaux*(sin(angulo*(i+1))), altura, raioaux*((cos(angulo
309                 *(i+1)))))
310
311             glVertex3f(raioaux*(sin(angulo*i)), -altura, raioaux*((cos(angulo*i)
312                 )))
313             glVertex3f(raioaux*(sin(angulo*(i + 1))), altura, raioaux*((cos(
314                 angulo*(i + 1)))))
315             glVertex3f(raioaux*(sin(angulo*(i ))), altura, raioaux*((cos(angulo
316                 *(i )))))
317         }
318     }
319     glEnd();

```

```

319         alfa += alfaux;
320         stacks--;
321     }
322 }
323
324
325 //DESENHA STACKS
326 for (int j = 0; j < (stacks / 2)-1; j++) {
327
328     glBegin(GL_TRIANGLES);
329
330     glColor3f(0.5, 0.8, 1);
331
332     for (int i = 0; i < slices; i++) {
333
334         glVertex3f(raioaux*(sin(angulo*i)), altura, raioaux*((cos(
335             angulo*i)))));
336         glVertex3f(raioaux2*(sin(angulo*(i + 1))), altura2, raioaux2
337             *((cos(angulo*(i + 1)))));
338         glVertex3f(raioaux2*(sin(angulo*(i))), altura2, raioaux2*((
339             cos(angulo*(i)))));
340
341         glVertex3f(raioaux*(sin(angulo*i)), altura, raioaux*((cos(
342             angulo*i)))));
343         glVertex3f(raioaux*(sin(angulo*(i + 1))), altura, raioaux*((
344             cos(angulo*(i + 1)))));
345         glVertex3f(raioaux2*(sin(angulo*(i + 1))), altura2, raioaux2
346             *((cos(angulo*(i + 1)))));
347
348     }
349     glEnd();
350
351     glBegin(GL_TRIANGLES);
352
353     glColor3f(0.5, 0.8, 1);
354
355     for (int i = 0; i < slices; i++) {
356
357         glVertex3f(raioaux2*(sin(angulo*i)), -altura2, raioaux2*((
358             cos(angulo*i)))));
359         glVertex3f(raioaux*(sin(angulo*(i + 1))), -altura, raioaux
360             *((cos(angulo*(i + 1)))));
361         glVertex3f(raioaux*(sin(angulo*(i))), -altura, raioaux*((cos
362             (angulo*(i)))));
363
364         glVertex3f(raioaux2*(sin(angulo*i)), -altura2, raioaux2*((
365             cos(angulo*i)))));
366         glVertex3f(raioaux2*(sin(angulo*(i + 1))), -altura2,
367             raioaux2*((cos(angulo*(i + 1)))));
368         glVertex3f(raioaux*(sin(angulo*(i + 1))), -altura, raioaux
369             *((cos(angulo*(i + 1)))));
370
371     }
372     glEnd();

```

```

366
367
368         alfa += alfaux;
369         altura = altura2;
370         altura2 = sin(alfa) * radius;
371         raioaux = raioaux2;
372         raioaux2 = sqrtf((radius*radius) - (altura2 * altura2));
373     }
374
375
376
377     glBegin(GL_TRIANGLES);
378
379     glColor3f(0.5, 0.8, 1);
380
381     for (int i = 0; i < slices; i++) {
382
383
384         glVertex3f(raioaux*(sin(angulo*i)), altura, raioaux*((cos(angulo*i))));
385         glVertex3f(raioaux*(sin(angulo*(i + 1))), altura, raioaux*((cos(angulo*(i +
386             1)))));
387         glVertex3f(raioaux2*(sin(angulo*(i + 1))), altura2, raioaux2*((cos(angulo*(i
388             + 1)))));
389
390         glVertex3f(raioaux2*(sin(angulo*i)), -altura2, raioaux2*((cos(angulo*i))));
391         glVertex3f(raioaux*(sin(angulo*(i + 1))), -altura, raioaux*((cos(angulo*(i +
392             1)))));
393         glVertex3f(raioaux*(sin(angulo*(i))), -altura, raioaux*((cos(angulo*(i))));
394     }
395
396     glEnd();
397
398
399
400 }
401
402 void toroid_square(float r0, float r1, float thick, int sides, int rings) {
403
404
405     float alfa = (2 * M_PI) / rings;
406
407
408     float altura = 0;
409     float altura2 = sin(alfa) * radius;
410     float angulo = (2 * M_PI) / sides;
411     float raioaux = radius;
412     float raioaux2 = sqrtf((radius*radius) - (altura2 * altura2));
413
414
415     //DESENHA STACKS
416
417     glBegin(GL_TRIANGLE_STRIP);
418
419     glColor3f(0.5, 0.8, 1);
420
421

```

```

422     for (int j = 0; j <= rings; j++) {
423
424         for (int i = 0; i <= sides; i++) {
425
426             glVertex3f(r0*cos(alfa*j), r0*sin(alfa*j), thick);
427             glVertex3f(r1*cos(alfa*j), r1*sin(alfa*j), thick);
428
429
430         }
431
432     }
433     glEnd();
434     glBegin(GL_TRIANGLE_STRIP);
435     for (int j = 0; j <= rings; j++) {
436
437         for (int i = 0; i <= sides; i++) {
438
439
440             glVertex3f(r1*cos(alfa*j), r1*sin(alfa*j), -thick);
441             glVertex3f(r0*cos(alfa*j), r0*sin(alfa*j), -thick);
442
443
444         }
445
446     }
447     glEnd();
448
449     glBegin(GL_TRIANGLE_STRIP);
450     for (int j = 0; j <= rings; j++) {
451
452         for (int i = 0; i <= sides; i++) {
453
454
455             glVertex3f(r1*cos(alfa*j), r1*sin(alfa*j), -thick);
456             glVertex3f(r0*cos(alfa*j), r0*sin(alfa*j), -thick);
457
458
459         }
460
461     }
462     glEnd();
463
464     glBegin(GL_TRIANGLE_STRIP);
465     for (int j = 0; j <= rings; j++) {
466
467         for (int i = 0; i <= sides; i++) {
468
469
470             glVertex3f(r1*cos(alfa*j), r1*sin(alfa*j), thick);
471             glVertex3f(r1*cos(alfa*j), r1*sin(alfa*j), -thick);
472
473
474         }
475
476     }
477     glEnd();
478
479
480     alfa += alfa;

```

```
481
482     altura = altura2;
483     altura2 = sin(alfa) * radius;
484     raioaux = raioaux2;
485     raioaux2 = sqrtf((radius*radius) - (altura2 * altura2));
486
487
488 }
```

---

---