Computação Gráfica (3º ano de LCC)

Trabalho Prático (Fase 1) — Grupo 1

Relatório de Desenvolvimento

Bruno Dias da Gião (A96544) Bruno Miguel Fernandes Araújo (A97509)
João Luís da Cruz Pereira (A95375)

7 de março de 2024

Conteúdo

Lista de Figuras

1 Introdução

Este relatório serve como um apoio ao trabalho ,tem explicações sobre os raciocínios usados na primeira fase do projeto da unidade curricular Computação Gráfica.

Nesta fase tivemos de definir duas aplicações:

- *generator*: que após a receção dos parâmetros necessários para a criação de um modelo, cria um ficheiro com a representação dos vértices deste modelo;
- *engine*: que, a partir de um ficheiro de configuração, escrito em *XML*, desenha um espaço com os modelos que já foram previamente gerados.

Demos uso dos módulos *tinyxml2* para a leitura do ficheiro de configuração e do *GLUT* para a representação gráfica.

1.1 Estrutura do Relatório

- §2- Descrevemos a implementação da aplicação *generator*.
- §3 - Apresentamos a implementação da aplicação *engine*.
- §4 - Conclusão.
- §5 - Imagens tiradas dos modelos gerados a partir dos ficheiros *XML* dos testes e um exemplo.

2 *Generator*

Esta aplicação foi desenvolvida com o objetivo de gerar ficheiros do formato *.3d* que serão eventualmente usados pela aplicação *engine*.

O generator é capaz de gerar modelos para cinto tipos de primitivas, uma delas como extra, a torus.

Começaremos por analisar individualmente o raciocíno por detrás destas:

2.1 Sphere

Para a definição de uma esfera são necessários os seguintes parâmetros:

- O seu raio (*radius*);
- O número de divisões verticais (*slices*);
- O número de divisões horizontais (*stacks*);
- O ficheiro que será aberto (*file*).

Como variáveis iniciais temos:

- As coordenadas do ponto que foram calculadas (px,py,pz).
- α, β (inicializadas a 0) e as diferenças de α e β que serão necessárias nos cálculos dos pontos. Respetivamente, *alpha*, *beta* ,*alpha_ diff*, *beta_ diff*.
- alpha_diff é inicializado com a divisão entre o intervalo deste (360°) e o número de slices.
- beta_diff é inicializado com a divisão entre o intervalo deste (180°) e o número de stacks.

```
int32_t gen_sphere(float radius, int32_t slices, int32_t stacks, char*file)
{
  FILE* output = fopen(file, "w+");
  char buff[512];
  float px, py, pz, alpha_diff = 2 * M_PI / slices,
        beta_diff = M_PI / stacks, alpha = 0, beta = 0;
  size_t b_read;
```

Para o cálculo dos pontos temos dois ciclos, um que itera até o número de slices, que acaba na reatribuição de 0 a beta e incrementando o valor de alpha com alpha_ diff permitindo que alpha, ao longo deste ciclo, tenha todos os valores possíveis no seu intervalo de 360°.

Temos então um outro ciclo interno que itera até o número de stacks e terá os cálculos para 6 pontos (2 triângulos) que seguem a seguinte primitiva:

$$px = radius \times \cos(\beta) \times \sin(\alpha)$$

$$py = radius \times \sin(\beta)$$

$$pz = radius \times \cos(\beta) \times \sin(\beta)$$

Para termos então os pontos necessários iremos somar beta_diff , alpha_diff ou ambos dependendo do requerido e por fim incrementar a β, beta_diff. É de notar que β tem um decremento de 90^{o} pois queremos que ao longo do ciclo este varie entre $-90 < \beta < 90$ tal como foi lecionado.

`if(j!=0){}` e `if(j!=stacks-1){}` foram necessários para evitar a sobreposição de triângulos, quando estamos no topo ou no fundo da esfera.

```
  for (int i = 0; i < slices; i++) {
    for (int j = 0; j < stacks; j++) { /* ifs */
      if(j!=0){
        b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n",
              radius * cosf(beta - M_PI_2) * cosf(alpha),
              radius * sinf(beta - M_PI_2),
              radius*cosf(beta-M_PI_2)*sinf(alpha));
        fwrite(buff, sizeof (int8_t),b_read, output);
        b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n",
              radius * cosf(beta - M_PI_2 + beta_diff)
              * cosf(alpha + alpha_diff),
              radius * sinf(beta - M_PI_2 + beta_diff),
              radius*cosf(beta-M_PI_2+beta_diff)*
              sinf(alpha + alpha_diff));
        fwrite(buff, sizeof (int8_t),b_read, output);
        b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n",
              radius * cosf(beta - M_PI_2)
              * cosf(alpha + alpha_diff),
              radius * sinf(beta - M_PI_2),
              radius*cosf(beta-M_PI_2)
              *sinf(alpha + alpha_diff));
        fwrite(buff, sizeof (int8_t),b_read, output);
      }
      if(j!=stacks-1){
        b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n",
              radius * cosf(beta - M_PI_2) * cosf(alpha),
              radius * sinf(beta - M_PI_2),
              radius*cosf(beta-M_PI_2)*sinf(alpha));
        fwrite(buff, sizeof (int8_t),b_read, output);
        b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n",
              radius * cosf(beta - M_PI_2 + beta_diff) * cosf(alpha),
              radius * sinf(beta - M_PI_2 + beta_diff),
              radius*cosf(beta-M_PI_2 + beta_diff)*sinf(alpha));
```

```
        fwrite(buff, sizeof (int8_t),b_read, output);
        b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n",
              radius * cosf(beta - M_PI_2 + beta_diff)
              * cosf(alpha + alpha_diff),
              radius * sinf(beta - M_PI_2 + beta_diff),
              radius*cosf(beta-M_PI_2+beta_diff)*
              sinf(alpha + alpha_diff));
        fwrite(buff, sizeof (int8_t),b_read, output);
      }
      beta += beta_diff;
    }
    beta = 0;
    alpha += alpha_diff;
  }
  fclose(output);
  return 0;
}
```

2.2 Cone

Para a definição de um cone são necessários os seguintes parâmetros:

- O raio da sua base (*radius*).
- A sua altura (*height*).
- O número de divisões verticais (*slices*).
- O número de divisões horizontais (*stacks*).
- Por fim o ficheiro que será aberto (*file*).

Para o cálculo dos pontos da figura começamos por ter dois *nested loops* onde percorremos cada *stack* (nível) e todas as *slices* no mesmo. Começamos por desenhar a base que será composta por um triângulo por cada slice. Os pontos são calculados através de coordenadas polares com centro na origem e a diferença de ângulo entre cada slice é obtido através da divisão do ângulo total pelo número de slices. Ou seja,

$$px = rad \times \sin(\alpha)$$

$$pz = rad \times \cos(\alpha)$$

Analisando agora o caso de ser a última *stack* será apenas um triângulo que liga ao centro com a altura desejada, o método de calcular as coordenadas é o mesmo que na base.

No caso de não ser a última *stack* (base e restantes stacks) iremos querer desenhar um quadrado composto por 2 triângulos. Os mesmos são obtidos da mesma forma que as restantes coordenadas mas tendo em conta a diferença no valor do y para além da diferença do ângulo.

No final de cada iteração do ciclo mais interior, o ângulo é incrementado por *angle_diff*. No caso do ciclo exterior, o ângulo volta a 0, o valor de y é incrementado por *y_diff* (valor da altura dividido pelo número de stacks) e o raio é decrementado por *xz_diff* (valor do raio dividido pelo número de stacks).

```
int32_t gen_cone(float radius, float height, int32_t slices, int32_t stacks, char*file)
{
FILE* output = fopen(file, "w+");
char buff[512];
size_t b_read;
int i, j, r = 0;
```

```
float y=0;
float angle = 0;
float cur_rad = radius;
float angle_diff = 2*M_PI/slices;
float xz_diff = radius/stacks;
float y_diff = height/stacks;

for(i=0; i<stacks; i++) {
    for(j=0; j<slices; j++) {
        // Bottom face
        if (i==0) {
            b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", cur_rad*sinf(angle), 0.f, cur_rad*cosf(angle));
            fwrite(buff, sizeof (int8_t),b_read, output);
            b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", 0.f, 0.f, 0.f);
            fwrite(buff, sizeof (int8_t),b_read, output);
            b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", cur_rad*sinf(angle+angle_diff), 0.f, cur_rad*cosf(angle+angle_diff));
            fwrite(buff, sizeof (int8_t),b_read, output);
        }

        if (i==stacks-1){
            b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", cur_rad*sinf(angle+angle_diff), y, cur_rad*cosf(angle+angle_diff));
            fwrite(buff, sizeof (int8_t),b_read, output);
            b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", 0.f, height, 0.f);
            fwrite(buff, sizeof (int8_t),b_read, output);
            b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", cur_rad*sinf(angle), y, cur_rad*cosf(angle));
            fwrite(buff, sizeof (int8_t),b_read, output);
        }
        else {
            b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", (cur_rad-xz_diff)*sinf(angle+angle_diff), y+y_diff,
                    (cur_rad-xz_diff)*cosf(angle+angle_diff));
            fwrite(buff, sizeof (int8_t),b_read, output);
            b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", (cur_rad-xz_diff)*sinf(angle), y+y_diff,
                    (cur_rad-xz_diff)*cosf(angle));
            fwrite(buff, sizeof (int8_t),b_read, output);
            b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", cur_rad*sinf(angle+angle_diff), y,
                    cur_rad*cosf(angle+angle_diff));
            fwrite(buff, sizeof (int8_t),b_read, output);

            b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", (cur_rad-xz_diff)*sinf(angle), y+y_diff,
                    (cur_rad-xz_diff)*cosf(angle));
            fwrite(buff, sizeof (int8_t),b_read, output);
            b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", cur_rad*sinf(angle), y,
                    cur_rad*cosf(angle));
            fwrite(buff, sizeof (int8_t),b_read, output);
            b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", cur_rad*sinf(angle+angle_diff), y,
                    cur_rad*cosf(angle+angle_diff));
            fwrite(buff, sizeof (int8_t),b_read, output);
        }
        angle += angle_diff;
    }
```

```
    angle = 0;
    y += y_diff;
    cur_rad -= xz_diff;
}
fclose(output);
return r;
}
```

2.3 Plane

Para a definição de um *Plane* são necessários os seguintes parâmetros:

- O seu comprimento (*full_size*).
- O número de divisões (*divs*).
- Por fim o ficheiro que será aberto (*file*).

Começamos por ter dois *nested loops* cada um a percorrer o número de divisões que o plano tem (pois o mesmo será quadrado). Por cada iteração teremos um quadrado composto por dois triângulos.

Começamos com o x a metade do comprimento e o z como o inverso de x, ou seja, o inverso da metade do comprimento. Teremos então, como ponto inicial, o ponto com o maior valor de x e o menor valor de z.

Em cada iteração do ciclo mais interior iremos calcular os pontos através da somas e subtração de *off* (comprimento a dividir pelo número de divisões). No final, subtraímos o valor de *off* ao x para passarmos para o próximo quadrado a ser desenhado.

No final de cada iteração do ciclo mais exterior, voltamos a colocar o valor de x como metade do comprimento (voltar ao início) e incrementamos o z por *off* (próxima linha de quadrados).

```
int32_t gen_plane(float full_size, int32_t divs, char* file)
{
  FILE* output = fopen(file, "w+");
  char buff[512];
  size_t b_read;
  float x = full_size/2, z = -x, off=full_size/divs;
  int i,j, err = 0;

  for (i=0; i < divs; i++) {
    for (j=0; j < divs; j++) {
      //curr.x = x = i * div_len - div_len + off;
      b_read = sprintf(buff, "%.3f 0.000 %.3f\n", x, z+off);
      fwrite(buff, sizeof (int8_t),b_read, output);

      b_read = sprintf(buff, "%.3f 0.000 %.3f\n", x, z);
      fwrite(buff, sizeof (int8_t),b_read, output);

      b_read = sprintf(buff, "%.3f 0.000 %.3f\n", x-off, z);
      fwrite(buff, sizeof (int8_t),b_read, output);

      b_read = sprintf(buff, "%.3f 0.000 %.3f\n", x-off, z);
      fwrite(buff, sizeof (int8_t),b_read, output);

      b_read = sprintf(buff, "%.3f 0.000 %.3f\n", x-off, z+off);
      fwrite(buff, sizeof (int8_t),b_read, output);

      b_read = sprintf(buff, "%.3f 0.000 %.3f\n", x, z+off);
      fwrite(buff, sizeof (int8_t),b_read, output);
```

```
29
30        x -= off;
31      }
32      x = full_size/ 2;
33      z += off;
34    }
35
36    fclose(output);
37    return err;
38  }
```

2.4 Box

Para a definição de uma box são necessários os seguintes parâmetros:

- O seu comprimento (l).
- O número de divisões para cada aresta (d)
- Por fim o ficheiro que será aberto (*file*).

Seguindo um raciocínio similar ao plano, iremos fazer novamente dois *nested loops* sobre as mesmas circunstâncias mas desta vez para cada duas faces da *Box*.

Começamos pela base e o pelo topo da *Box*, faces paralelas ao plano *x0z*.

Seguidamente a face de trás e da frente, faces paralelas ao plano *y0x*.

Por fim a face da direita e da esquerda, faces paralelas ao plano *y0z*.

Apesar de realmente ter cálculos diferentes, o raciocínio por detrás deste é similar ao do plane.

```
1  int32_t gen_box(float l, int32_t d, char* file)
2  {
3    FILE* output = fopen(file , "w+");
4    char buff[512];
5    size_t b_read;
6    int32_t i, j, r = 0;
7    float x = -l/2;
8    float y = l/2;
9    float z = -l/2;
10   float diff = l/d;
11
12     // Bottom and Top Faces
13   for (i=0; i<d; i++){
14     for (j=0; j<d; j++) {
15
16
17       b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", x, y, z);
18       fwrite(buff, sizeof (int8_t),b_read, output);
19       b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", x+diff, y, z+diff);
20       fwrite(buff, sizeof (int8_t),b_read, output);
21       b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", x+diff, y, z);
22       fwrite(buff, sizeof (int8_t),b_read, output);
23
24       b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", x, y, z);
25       fwrite(buff, sizeof (int8_t),b_read, output);
26       b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", x, y, z+diff);
```

```
27        fwrite(buff, sizeof (int8_t),b_read, output);
28        b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", x+diff, y, z+diff);
29        fwrite(buff, sizeof (int8_t),b_read, output);
30
31        b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", x+diff, -y, z+diff);
32        fwrite(buff, sizeof (int8_t),b_read, output);
33        b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", x, -y, z);
34        fwrite(buff, sizeof (int8_t),b_read, output);
35        b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", x+diff,-y, z);
36        fwrite(buff, sizeof (int8_t),b_read, output);
37
38        b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", x, -y, z+diff);
39        fwrite(buff, sizeof (int8_t),b_read, output);
40        b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", x, -y, z);
41        fwrite(buff, sizeof (int8_t),b_read, output);
42        b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", x+diff, -y, z+diff);
43        fwrite(buff, sizeof (int8_t),b_read, output);
44
45        z += diff;
46      }
47      z = - l/2;
48      x += diff;
49    }
50
51    x = -l / 2;
52    y = -l / 2;
53    z = l / 2;
54
55      // Front and Back faces
56    for (int i=0; i<d; i++){
57      for (int j=0; j<d; j++) {
58
59        b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", x+diff, y+diff, z);
60        fwrite(buff, sizeof (int8_t),b_read, output);
61        b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", x, y, z);
62        fwrite(buff, sizeof (int8_t),b_read, output);
63        b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", x+diff, y, z);
64        fwrite(buff, sizeof (int8_t),b_read, output);
65
66        b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", x, y+diff, z);
67        fwrite(buff, sizeof (int8_t),b_read, output);
68        b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", x, y, z);
69        fwrite(buff, sizeof (int8_t),b_read, output);
70        b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", x+diff, y+diff, z);
71        fwrite(buff, sizeof (int8_t),b_read, output);
72
73        b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", x, y, -z);
74        fwrite(buff, sizeof (int8_t),b_read, output);
75        b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", x+diff, y+diff, -z);
76        fwrite(buff, sizeof (int8_t),b_read, output);
77        b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", x+diff, y, -z);
78        fwrite(buff, sizeof (int8_t),b_read, output);
79
80        b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", x, y, -z);
81        fwrite(buff, sizeof (int8_t),b_read, output);
82        b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", x, y+diff, -z);
83        fwrite(buff, sizeof (int8_t),b_read, output);
84        b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", x+diff, y+diff, -z);
85        fwrite(buff, sizeof (int8_t),b_read, output);
```

```
86
87        y += diff;
88      }
89      y = -l/2;
90      x += diff;
91    }
92
93    z = - l/2;
94    y = - l/2;
95    x = l/2;
96
97      // Right and Left faces
98    for (int i=0; i<d; i++){
99      for (int j=0; j<d; j++) {
100
101       b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", x, y, z);
102       fwrite(buff, sizeof (int8_t),b_read, output);
103       b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", x, y+diff, z+diff);
104       fwrite(buff, sizeof (int8_t),b_read, output);
105       b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", x, y, z+diff);
106       fwrite(buff, sizeof (int8_t),b_read, output);
107
108       b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", x, y, z);
109       fwrite(buff, sizeof (int8_t),b_read, output);
110       b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", x, y+diff, z);
111       fwrite(buff, sizeof (int8_t),b_read, output);
112       b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", x, y+diff, z+diff);
113       fwrite(buff, sizeof (int8_t),b_read, output);
114
115       b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", -x, y+diff, z+diff);
116       fwrite(buff, sizeof (int8_t),b_read, output);
117       b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", -x, y, z);
118       fwrite(buff, sizeof (int8_t),b_read, output);
119       b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", -x, y, z+diff);
120       fwrite(buff, sizeof (int8_t),b_read, output);
121
122       b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", -x, y+diff, z);
123       fwrite(buff, sizeof (int8_t),b_read, output);
124       b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", -x, y, z);
125       fwrite(buff, sizeof (int8_t),b_read, output);
126       b_read = snprintf(buff, 512, "%.3f %.3f %.3f\n", -x, y+diff, z+diff);
127       fwrite(buff, sizeof (int8_t),b_read, output);
128
129       y += diff;
130     }
131     y = -l/2;
132     z += diff;
133   }
134
135   fclose(output);
136   return 0;
137 }
```

2.5 Torus

Para a definição de um torus são necessários os seguintes parâmetros:

- O raio interno (*inner_radius*).

- O raio externo (*outer_ radius*).
- O número de divisões verticais(*slices*).
- O número de divisões horizontais(*stacks*).
- Por fim o ficheiro que será aberto (*file*).

Para o cálculo dos pontos utilizamos dois *nested loops*, um para cada slice, e outro para cada stack. Para obtermos as coordenadas dos pontos utilizamos:

$$px = (inner_radius + outer_radius * cos(beta - \pi) * cos(alfa)$$

$$py = outer_radius * sin(beta - \pi)$$

$$pz = (inner_radius + outer_radius * cos(beta - \pi)) * sin(alfa)$$

Utilizamos também a diferença entre cada *slice* para calcular a diferença entre cada *beta* e a diferença entre *stacks* para calcular a diferença entre cada *alfa*.

No final de cada iteração do ciclo mais interior incrementamos o beta por *beta_ diff*. No caso do ciclo mais exterior o *beta* é colocado a 0 e o alfa é incrementado por *alfa_ diff*.

```
int32_t gen_torus(float inner_radius, float outer_radius,
      int32_t slices, int32_t stacks, char* file)
{
  FILE* output = fopen(file, "w+");
  float alfa = 0, beta = 0;
  float alfa_diff = 2 * M_PI / slices;
  float beta_diff = 2*M_PI / stacks;
  float px, py, pz;
  char buff[512];
  size_t b_read;
  int32_t rr = 0;

  for (int i = 0; i < slices; i++) {
    for (int j = 0; j < stacks; j++) {

      px = (inner_radius + outer_radius * cos(beta -M_PI_2)) * cos(alfa);
      py = outer_radius * sin(beta - M_PI_2);
      pz = (inner_radius + outer_radius * cos(beta - M_PI_2)) * sin(alfa);
      b_read = sprintf(buff, "%.3f %.3f %.3f\n", px, py, pz);
      fwrite(buff, sizeof (int8_t),b_read, output);

      px = (inner_radius + outer_radius * cos(beta - M_PI_2 + beta_diff)) * cos(alfa);
      py = outer_radius * sin(beta - M_PI_2+ beta_diff);
      pz = (inner_radius + outer_radius * cos(beta - M_PI_2 + beta_diff)) * sin(alfa);
      b_read = sprintf(buff, "%.3f %.3f %.3f\n", px, py, pz);
      fwrite(buff, sizeof (int8_t),b_read, output);

      px = (inner_radius + outer_radius * cos(beta - M_PI_2 )) * cos(alfa+alfa_diff);
      py =  outer_radius * sin(beta - M_PI_2);
      pz= (inner_radius + outer_radius * cos(beta - M_PI_2)) * sin(alfa+alfa_diff);
      b_read = sprintf(buff, "%.3f %.3f %.3f\n", px, py, pz);
      fwrite(buff, sizeof (int8_t),b_read, output);

      px = (inner_radius + outer_radius * cos(beta - M_PI_2 + beta_diff)) * cos(alfa);
      py =  outer_radius * sin(beta - M_PI_2 + beta_diff);
      pz = (inner_radius + outer_radius * cos(beta - M_PI_2 + beta_diff)) * sin(alfa);
      b_read = sprintf(buff, "%.3f %.3f %.3f\n", px, py, pz);
      fwrite(buff, sizeof (int8_t),b_read, output);
```

```
39
40      px = (inner_radius + outer_radius * cos(beta - M_PI_2 + beta_diff)) * cos(alfa +
    alfa_diff);
41      py =  outer_radius * sin(beta - M_PI_2 + alfa_diff);
42      pz = (inner_radius +outer_radius * cos(beta - M_PI_2 + beta_diff)) * sin(alfa +
    alfa_diff);
43      b_read = sprintf(buff, "%.3f %.3f %.3f\n", px, py, pz);
44      fwrite(buff, sizeof (int8_t),b_read, output);
45
46      px = (inner_radius + outer_radius * cos(beta - M_PI_2)) * cos(alfa + alfa_diff);
47      py = outer_radius * sin(beta - M_PI_2);
48      pz = (inner_radius + outer_radius * cos(beta - M_PI_2)) * sin(alfa + alfa_diff);
49      b_read = sprintf(buff, "%.3f %.3f %.3f\n", px, py, pz);
50      fwrite(buff, sizeof (int8_t),b_read, output);
51
52      beta += beta_diff;
53    }
54    beta = 0;
55    alfa += alfa_diff;
56
57  }
58  fclose(output);
59  return rr;
60 }
```

3 *Engine*

Relativamente ao **Motor**, temos que

1. Fazemos *parse* de um ficheiro XML que será o *input* do nosso motor;
2. Fazemos *parse* das primitivas num *path* fixo - `/prims/`;
3. Finalmente podemos inicializar o ambiente OpenGL e iniciar o ciclo principal do programa.

Procederemos então a explicar cada uma destas fases detalhadamente.

3.1 Parse do ficheiro XML

Fazendo uso da biblioteca **tinyxml2** [1], podemos extrair a informação dos atributos de cada elemento XML, que será onde, maioritariamente, se encontram as informações que desejamos. A informação de cada atributo é armazenada numa struct global `world` composta por structs representativas de cada elemento do *input*.

De notar também o modo como são armazenados os nomes dos ficheiros .3d, isto é, num vetor de struct prims que é constituido por um nome, um contador de quantas vezes vai ser desenhado num dado grupo e que grupo constitui.

Também extendemos o XML com a possibilidade de, opcionalmente alterar o título da janela e a posição inicial da janela (caso contrário inicia com "CG-PROJ" - 0, 0, respetivamente).

```
1 float alfa = 0.0f, beta = 0.0f, radius = 5.0f;
2 float camX, camY, camZ;
3 int timebase, time, frames=0;
4 float fps;
5 int cur_mode = GL_LINE, cur_face = GL_FRONT;
6 GLuint vertex_count, vertices;
```

[1] ver `https://github.com/leethomason/tinyxml2`

```
7
8  struct triple {
9    float x,y,z;
10 };
11 struct prims {
12   int count;
13   int group;
14   char name[64];
15 };
16 struct {
17   struct {
18     int h,w,sx,sy;
19     char title[64];
20   } win;
21   struct {
22     struct triple pos;
23     struct triple lookAt;
24     struct triple up; /* 0 1 0 */
25     struct triple proj; /* 60 1 1000*/
26   } cam;
27   std::vector<struct prims> primitives;
28 } world;
29
30 typedef std::vector<struct triple> Primitive_Coords;
31 std::vector<Primitive_Coords> prims;
32 static int not_in_prims_g(const char* f, int* i, int g, int N)
33 {
34   int r = 1;
35   for (*i=0; *i< N && r; *i+=1) {
36     if (!strcmp(f, world.primitives[*i].name) &&
37         world.primitives[*i].group == g)
38         r = 0;
39   }
40   return r;
41 }
```

```
1  int xml_init(char* xml_file)
2  {
3    XMLDocument doc;
4    XMLElement* world_l, *window, *cam, *posi, *lookAt, *up, *proj,
5      *group_R, *gr, *mod, *models;
6    int i = 0, g, j, rs = 0;
7    const char* f;
8    const char* tit;
9    char tmp[1024];
10   doc.LoadFile(xml_file);
11
12   world_l = doc.RootElement();
13   if (world_l) {
14     window = world_l->FirstChildElement("window");
15     if (!window) {
16       return -1;
17     }
18     world.win.w = window->IntAttribute("width");
19     world.win.h = window->IntAttribute("height");
20     world.win.sx = 0;
21     world.win.sy = 0;
22     world.win.sx = window->IntAttribute("x");
23     world.win.sy = window->IntAttribute("y");
```

```
    tit = window->Attribute("title");
    if (tit)
      strcpy(world.win.title, window->Attribute("title"));
    else
      strcpy(world.win.title, "CG-PROJ");
    cam = world_l->FirstChildElement("camera");
    if (!cam) {
      return -2;
    }
    posi = cam->FirstChildElement("position");
    if (!posi) {
      return -3;
    }
    world.cam.pos.x = posi->FloatAttribute("x");
    world.cam.pos.y = posi->FloatAttribute("y");
    world.cam.pos.z = posi->FloatAttribute("z");
    printf("%.3f %.3f %.3f\n", world.cam.pos.x,
           world.cam.pos.y,
           world.cam.pos.z);
    lookAt = cam->FirstChildElement("lookAt");
    if (!lookAt) {
      return -4;
    }
    world.cam.lookAt.x = lookAt->FloatAttribute("x");
    world.cam.lookAt.y = lookAt->FloatAttribute("y");
    world.cam.lookAt.z = lookAt->FloatAttribute("z");
    printf("%.3f %.3f %.3f\n", world.cam.lookAt.x,
           world.cam.lookAt.y,
           world.cam.lookAt.z);
    up = cam->FirstChildElement("up");
    if (up) {
      world.cam.up.x = up->FloatAttribute("x");
      world.cam.up.y = up->FloatAttribute("y");
      world.cam.up.z = up->FloatAttribute("z");
    }
    else {
      world.cam.up.x = 0.f;
      world.cam.up.y = 1.f;
      world.cam.up.z = 0.f;
    }
    printf("%.3f %.3f %.3f\n", world.cam.up.x,
           world.cam.up.y,
           world.cam.up.z);
    proj = cam->FirstChildElement("projection");
    if (proj) {
      world.cam.proj.x = proj->FloatAttribute("fov");
      world.cam.proj.y = proj->FloatAttribute("near");
      world.cam.proj.z = proj->FloatAttribute("far");
    }
    else {
      world.cam.proj.x = 60.f; // fov
      world.cam.proj.y = 1.f; // neAR
      world.cam.proj.z = 1000.f; //far
    }
    printf("%.3f %.3f %.3f\n", world.cam.proj.x,
           world.cam.proj.y,
           world.cam.proj.z);
    group_R = world_l->FirstChildElement("group");
    if (group_R){
```

```
      for (gr=group_R, g=0;
           gr;
           gr = gr->FirstChildElement("group"),
           g += 1) {

        models = gr->FirstChildElement("models");
        if (models) {
          for (mod=models->FirstChildElement("model");
               mod;
               mod=mod->NextSiblingElement("model")
               ) {

            if (mod){
              f = mod->Attribute("file");
              if ( not_in_prims_g(f, &j, g, i)) {
                struct prims tmp_p;
                strcpy(tmp, "../../prims/");
                strcpy(tmp_p.name, f);
                strcat(tmp, tmp_p.name);
                strcpy(tmp_p.name, tmp);
                tmp_p.count = 1;
                tmp_p.group = g;
                world.primitives.push_back(tmp_p);
                i += 1;
              }
              else
                world.primitives[j].count+=1;
            }
          }

        }
      }

    }
  }
  return i;
}
```

3.2 Parse do ficheiro 3d (Primitivas)

Já os ficheiros 3d, como são ficheiros de texto simples, podemos limitar-nos às bibliotecas standard de C, nomeadamente a sub-rotinas como: `fopen, fclose, fgets` e `strtok`.
Armazenamos as coordenadas de cada primitiva num vetor de vetores de triplos (i.e. uma struct constituida por 3 floats)

De forma a evitar leituras desnecessárias de ficheiros .3d, utilizamos um vetor auxiliar que armazena o nome dos ficheiros já lidos e o grupo a que pertencem, e, finalmente, armazenamos *count* vezes as coordenadas da primitiva.

```
void read_words (FILE *f, int top) {
  char line[1024];
  char* num;
  int i = 0, j=0;
  float x,y,z;
  while (fgets(line,sizeof(line), f) != NULL) {
    num = strtok(line, " \n");
    x = atof(num);
    num = strtok(NULL, " \n");
    y = atof(num);
```

```
    num = strtok(NULL, " \n");
    z = atof(num);

    prims[top].push_back({.x=x,.y=y,.z=z});
  }
}

int read_3d_files(int N)
{
  FILE* fd;
  int i,j;
  int flag = 0;
  struct tmp_s { int g; char*name;};
  std::vector<struct triple> aux;
  std::vector<struct tmp_s> already_read;
  for (i=0; i<N; i++) {
    flag = 0;
    for (j=0; j<already_read.size() && !flag;j++)
      if (!strcmp(world.primitives[i].name, already_read[j].name)&&
        world.primitives[i].group == already_read[j].g)
        flag = 1;
    if (!flag) {
      fd = fopen(world.primitives[i].name, "r");
      if (!fd) {
        return -1;
      }
      for (j=0; j < world.primitives[i].count; j++)
        prims.push_back(aux);
      read_words(fd, i);
      fclose(fd);
      already_read.push_back({.g=world.primitives[i].group,
                .name=world.primitives[i].name});
    }
  }
  return 0;
}
```

3.3 OpenGL

O código relativo ao desenho de gráficos através do ambiente OpenGL é análogo ao código concebido em aulas, com a diferença que este terá de ser agnóstico à primitiva, limitando-se a desenhar triângulos em modo direto.

```
int main(int argc, char **argv)
{
  int res = 0, i;
  if (argc < 2) {
    return 1;
  }
  res = xml_init(argv[1]);
  if (res < 0)
    return res;
  i = res;
  res = read_3d_files(i);

// init GLUT and the window
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_DEPTH|GLUT_DOUBLE|GLUT_RGBA);
  glutInitWindowPosition(world.win.sx,world.win.sy);
```

```
  glutInitWindowSize(world.win.w,world.win.h);
  glutCreateWindow(world.win.title);
  timebase = glutGet(GLUT_ELAPSED_TIME);

// Required callback registry
  glutDisplayFunc(renderScene);
  glutIdleFunc(renderScene);
  glutReshapeFunc(changeSize);

// Callback registration for keyboard processing
  glutKeyboardFunc(processKeys);
  glutSpecialFunc(processSpecialKeys);


  // init GLEW
#ifndef __APPLE__
  glewInit();
#endif


//  OpenGL settings
  /*glEnableClientState(GL_VERTEX_ARRAY);
  glGenBuffers(1, &vertices);*/
  glEnable(GL_DEPTH_TEST);
  glEnable(GL_CULL_FACE);

  //spherical2Cartesian();

  printInfo();

// enter GLUT's main cycle
  glutMainLoop();

  return 1;
}
```

```
void drawfigs(void)
{
  int i, j;
  glBegin(GL_TRIANGLES);
  for (i = 0; i<prims.size(); i++) {
    for (j=0; j<prims[i].size();j++) {
      glVertex3f(prims[i][j].x,
           prims[i][j].y,
           prims[i][j].z);
    }
  }
  glEnd();

}

void renderScene(void) {

  // clear buffers
  char fps_c[64];
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  // set the camera
  glLoadIdentity();
```

```
  gluLookAt(world.cam.pos.x, world.cam.pos.y, world.cam.pos.z,
    world.cam.lookAt.x, world.cam.lookAt.y
    , world.cam.lookAt.z,
    world.cam.up.x, world.cam.up.y, world.cam.up.z);
  /*gluLookAt(camX, camY,camZ,
      0.f,0.f,0.f,
      0.f,1.f,0.f);
  */

  glPolygonMode(GL_FRONT, GL_LINE);

  glBegin(GL_LINES);
    // X axis in red
    glColor3f(1.0f, 0.0f, 0.0f);
    glVertex3f(-100.0f, 0.0f, 0.0f);
    glVertex3f( 100.0f, 0.0f, 0.0f);
    // Y Axis in Green
    glColor3f(0.0f, 1.0f, 0.0f);
    glVertex3f(0.0f, -100.0f, 0.0f);
    glVertex3f(0.0f, 100.0f, 0.0f);
    // Z Axis in Blue
    glColor3f(0.0f, 0.0f, 1.0f);
    glVertex3f(0.0f, 0.0f, -100.0f);
    glVertex3f(0.0f, 0.0f, 100.0f);
    glColor3f(1.f,1.f,1.f);
  glEnd();

  frames++;
  time = glutGet(GLUT_ELAPSED_TIME);
  if (time - timebase > 1000) {
    fps = frames * 1000.0 / (time - timebase);
    timebase = time;
    frames = 0;
  }


  sprintf(fps_c, "%s-%d",world.win.title, (int)fps);

  glutSetWindowTitle(fps_c);

  drawfigs();
  // End of frame
  glutSwapBuffers();
}
```

```
void changeSize(int w, int h) {

  // Prevent a divide by zero, when window is too short
  // (you cant make a window with zero width).
  if(h == 0)
    h = 1;

  // compute window's aspect ratio
  float ratio = w * 1.0 / h;

  // Set the projection matrix as current
  glMatrixMode(GL_PROJECTION);
  // Load Identity Matrix
  glLoadIdentity();
```

```
15    // Set the viewport to be the entire window
16    glViewport(0, 0, w, h);
17
18    // Set perspective
19    gluPerspective(world.cam.proj.x,ratio, world.cam.proj.y ,world.cam.proj.z);
20
21    // return to the model view matrix mode
22    glMatrixMode(GL_MODELVIEW);
23  }
```

4 Conclusão

Concebemos um gerador de coordenadas das primitivas pedidas e do *Torus* e um **motor** de gráficos capaz de, conforme pedido, interpretar um ficheiro XML e as coordenadas geradas pelo gerador, conforme pedido no enunciado do projeto.

4.1 Trabalho Futuro

Para além do conteúdo requisito das fases ainda por realizar, temos certos objetivos pessoais que gostavamos de realizar ao longo da elaboração do projeto, nomeadamente:

- A implementação de 3 tipos de câmeras que podem ser predefinidas no input XML e alteradas ao longo da execução do engine, particularmente, uma câmera explorador, uma câmera primeira pessoa e uma câmera terceira pessoa[2].
- A separação do desenho das primitivas pelos grupos definidos no input XML, algo que já é feito quando este input é interpretado.
- Otimização do engine através do desenho das primitivas usando VBOs ou VBOs com indexação.
- Criação de mais tipos de primitivas como, por exemplo, o teapot ou prismas.

[2]ver Eve Online - `https://www.eveonline.com/`

5 Resultados

Os resultados obtidos para cada teste fornecido e para um exemplo do torus.

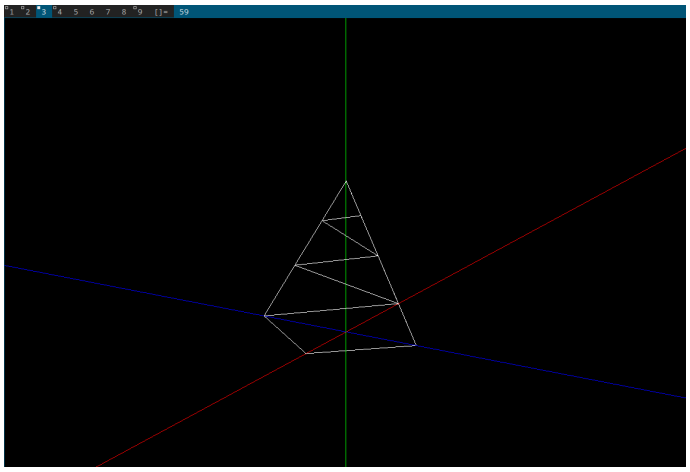

Figura 1: Teste 1

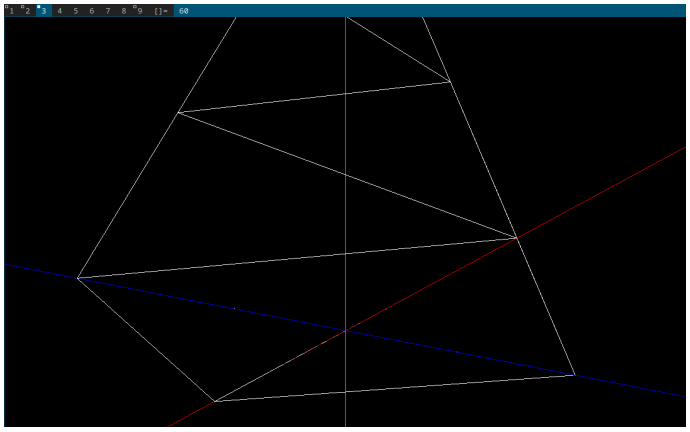

Figura 2: Teste 2

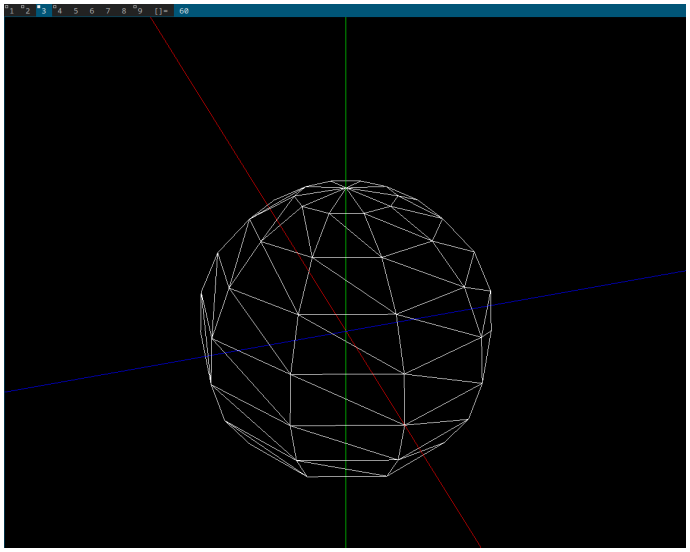

Figura 3: Teste 3

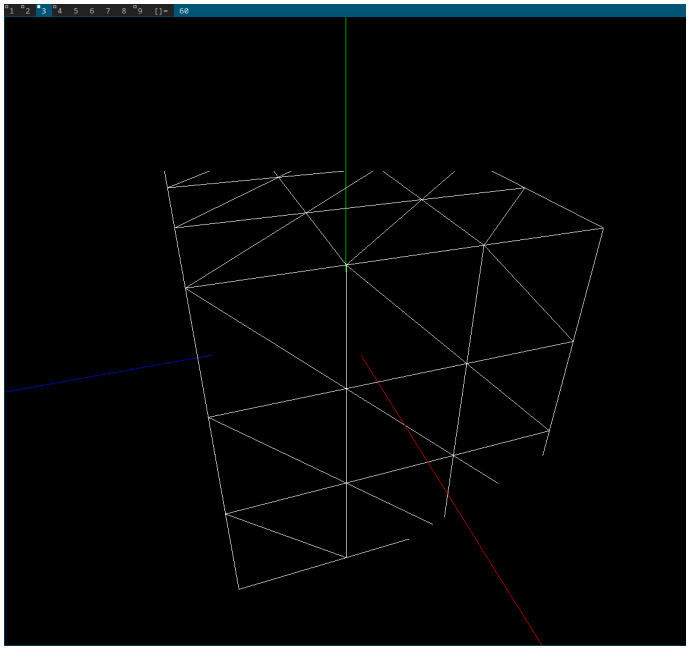

Figura 4: Teste 4

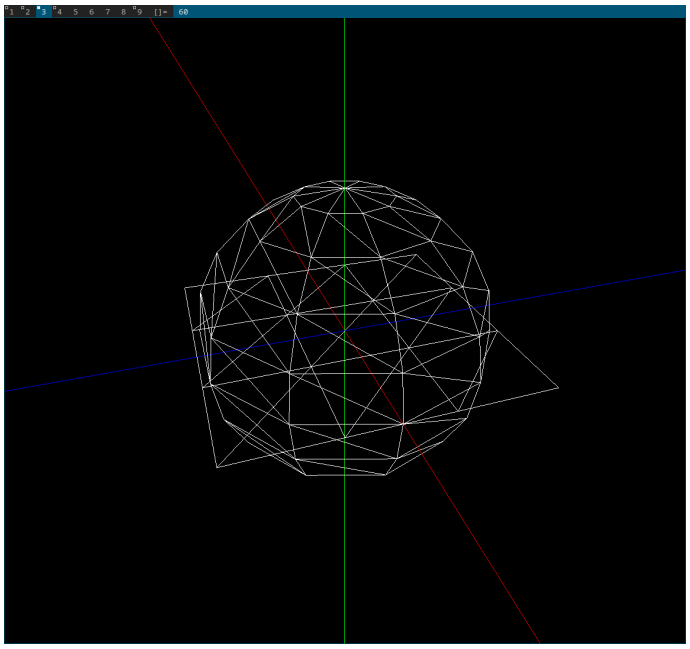

Figura 5: Teste 5

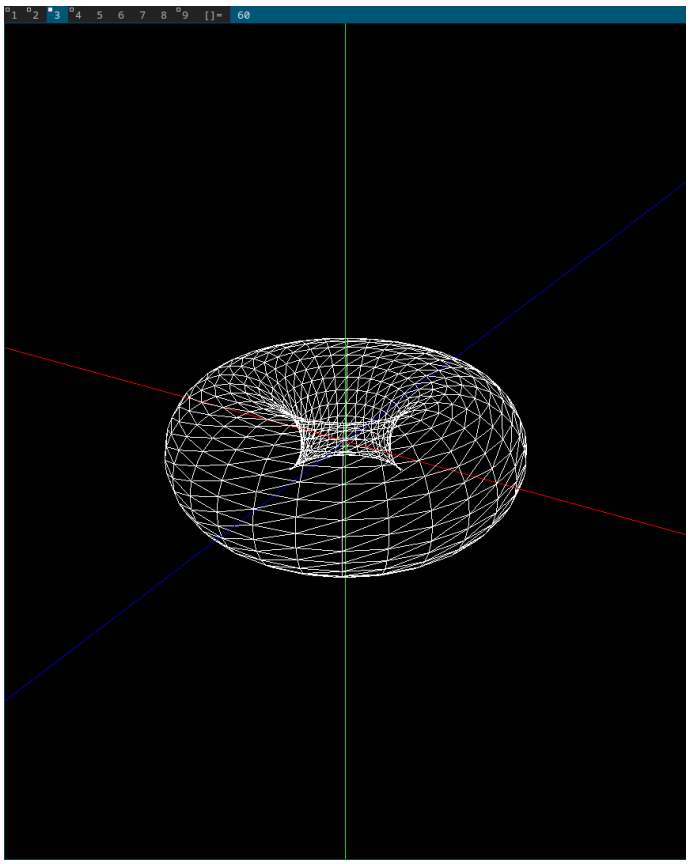

Figura 6: Torus