

Computação Gráfica (3<sup>o</sup> ano de LCC)  
**Trabalho Prático (Fase 1) — Grupo 3**  
Relatório de Desenvolvimento

André Lucena Ribas Ferreira (A94956)      Carlos Eduardo da Silva Machado (A96936)  
Gonçalo Manuel Maia de Sousa (A97485)

10 de março de 2023

### **Resumo**

Este relatório aborda a solução proposta para o enunciado da 1ª fase do Trabalho Prático da Unidade Curricular "Computação Gráfica".

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>2</b>
1.1	Estrutura do Relatório . . . . .	2
<b>2</b>	<b><i>Generator</i></b>	<b>3</b>
2.1	Esfera . . . . .	3
2.2	Cone . . . . .	5
2.3	Plano . . . . .	6
2.4	Cubo . . . . .	7
2.5	Torus . . . . .	7
<b>3</b>	<b><i>Engine</i></b>	<b>10</b>
3.1	Classes . . . . .	10
3.2	Parser . . . . .	11
3.3	<i>Engine</i> . . . . .	13
<b>4</b>	<b>Resultados</b>	<b>16</b>
<b>5</b>	<b>Conclusão</b>	<b>19</b>

# Capítulo 1

## Introdução

O presente relatório tem como objetivo apresentar a solução concebida pelo Grupo 3 para a 1<sup>a</sup> fase do Trabalho Prático da Unidade Curricular "Computação Gráfica".

Esta fase consiste em definir duas aplicações: o *generator*, que gera um ficheiro com representação dos vértices para o modelo pretendido; e o *engine*, que, a partir da especificação de um ficheiro de configuração, escrito em *xml*, exhibe uma cena com os modelos já gerados.

Para tal, utilizou-se o módulo *rapidxml* para leitura do ficheiro de configuração, tal como o *GLUT* para a representação gráfica.

### 1.1 Estrutura do Relatório

Para além deste, o relatório compreende diferentes Capítulos. Em 2, descreve-se a implementação da aplicação *generator*. Em 3 apresenta-se a implementação da aplicação *engine*. Em 5 apresenta-se a conclusão do relatório.

## Capítulo 2

# Generator

A aplicação *generator* foi desenhada de forma a gerar ficheiros *model* que podem ser utilizados pela aplicação *engine*, com extensão ".3d".

Esses ficheiros são encabeçados por um valor inteiro que denota a quantidade de vértices que o modelo contém, seguidos por valores binários que representam uma estrutura *tuple*, constituída por três valores de vírgula flutuante, para cada um dos vértices.

Em todos os momentos em que se realizam operações, nomeadamente de rotação, em pontos, utilizam-se múltiplos do ângulo de rotação/fator de translação e aplica-se sempre aos ponto originais. Desta forma, minimiza-se substancialmente o erro de aproximação por se efetuarem cálculos de vírgula flutuante.

A aplicação é capaz de gerar modelos para cinco tipos de primitivas diferentes:

### 2.1 Esfera

Os parâmetros necessários para definir uma esfera são: o seu raio (*radius*), o números de divisões horizontais (*slices*) e de divisões verticais (*stacks*). Inicialmente é fixado um ponto *P* no eixo *y* tal que:

$$P = (0, radius, 0)$$

```
1 tuple<float , float , float >* generate_sphere(float radius , int slices , int stacks , int *
  points_total){
2   *points_total = slices*6*(stacks-1);
3   float alfa_x = M_PI/stacks;
4   float alfa_y = 2*M_PI/slices;
5
6   float pivot_x = 0;
7   float pivot_y = radius;
8   float pivot_z = 0;
```

São posteriormente gerados *stacks* + 1 pontos auxiliares, incluindo *P*, por rotações de *P* de ângulos múltiplos de  $\frac{\pi}{stacks}$  em torno do eixo *Ox*, aos quais chamamos de *master\_line*.

```
1   tuple<float , float , float >* master_line = new tuple<float , float , float >[stacks+1];
2   tuple<float , float , float >* points_array = new tuple<float , float , float >[*points_total];
3   //generate master line
4   int master_line_index = 0;
5   int i;
6   for (i = 0; i < stacks+1; i++) {
7       master_line[master_line_index++] = make_tuple(
8           pivot_x ,
```

```

9         pivot_y*cos(i*alfa_x),
10        pivot_y*sin(i*alfa_x)
11    );
12 }

```

Finalmente, os restantes pontos da esfera são obtidos rodando os pontos do *master\_line* em torno do eixo *y* por ângulos múltiplos de  $\frac{2\pi}{slices}$ , tendo em conta a ordem contrária os ponteiros do relógio que os vértices devem ter para se obter a orientação correta.

Para cada uma das *slices*, ou seja, para cada uma das linhas verticais,

```

1 int index = 0;
2 for (int j = 0; j < slices; j++) {

```

os vértices são gerados primeiro no sentido crescente do ângulo de rotação ao torno do eixo *Ox* e de seguida no sentido decrescente. Apenas se percorre *stacks-1* divisões horizontais de modo a apenas gerar um triângulo tanto para a primeira e para a última.

No sentido decrescente, geram-se os triângulos no sentido contrário aos ponteiros, em relação à *master\_line*:

```

1     for (int i = 0; i < stacks-1; i++) {
2         points_array[index++] = make_tuple(
3             get<0>(master_line[i])*cos(j*alfa_y) + get<2>(master_line[i])*sin(j*alfa_y),
4             get<1>(master_line[i]),
5             -get<0>(master_line[i])*sin(j*alfa_y) + get<2>(master_line[i])*cos(j*alfa_y)
6         );
7
8         points_array[index++] = make_tuple(
9             get<0>(master_line[i+1])*cos(j*alfa_y) + get<2>(master_line[i+1])*sin(j*alfa_y),
10            get<1>(master_line[i+1]),
11            -get<0>(master_line[i+1])*sin(j*alfa_y) + get<2>(master_line[i+1])*cos(j*alfa_y)
12        );
13
14        points_array[index++] = make_tuple(
15            get<0>(master_line[i+1])*cos(((j+1)%slices)*alfa_y) + get<2>(master_line[i+1])*
16            sin(((j+1)%slices)*alfa_y),
17            get<1>(master_line[i+1]),
18            -get<0>(master_line[i+1])*sin(((j+1)%slices)*alfa_y) + get<2>(master_line[i+1])*
19            cos(((j+1)%slices)*alfa_y)
20        );
21    }

```

Enquanto que no sentido crescente, geram-se os triângulos no sentido dos ponteiros, em relação à *master\_line*:

```

1     for (int i = stacks; i > 1; i--) {
2         points_array[index++] = make_tuple(
3             get<0>(master_line[i])*cos(-j*alfa_y) + get<2>(master_line[i])*sin(-j*alfa_y),
4             get<1>(master_line[i]),
5             -get<0>(master_line[i])*sin(-j*alfa_y) + get<2>(master_line[i])*cos(-j*alfa_y)
6         );
7
8         points_array[index++] = make_tuple(
9             get<0>(master_line[i-1])*cos(-j*alfa_y) + get<2>(master_line[i-1])*sin(-j*alfa_y)
10        ),
11        get<1>(master_line[i-1]),
12        -get<0>(master_line[i-1])*sin(-j*alfa_y) + get<2>(master_line[i-1])*cos(-j*
13        alfa_y)
14        );
15
16        points_array[index++] = make_tuple(

```

```

15         get<0>(master_line[i-1])*cos(-((j+1)%slices)*alfa_y) + get<2>(master_line[i-1])*
sin(-((j+1)%slices)*alfa_y),
16         get<1>(master_line[i-1]),
17         -get<0>(master_line[i-1])*sin(-((j+1)%slices)*alfa_y) + get<2>(master_line[i-1])
*cos(-((j+1)%slices)*alfa_y)
18     );
19 }
20 }
21
22
23     return points_array;
24 }

```

De modo a evitar que pontos sejam calculados por rotações de 360 graus é calculado o módulo do índice  $j+1$  com *slices*.

## 2.2 Cone

De modo a construir o cone, precisamos de quatro parâmetros, o raio da base (*bottom\_radius*), a altura (*height*), as divisões verticais (*slices*) e as horizontais (*stacks*). Começamos por definir o ângulo *alfa* que é calculado através  $2 * \frac{\pi}{slices}$  e os passos *division\_height\_step* como  $\frac{height}{stacks}$  e *division\_radius\_step* como  $\frac{bottom\_radius}{stacks}$ .

Iniciamos com a construção da base do cone, com um triângulo para cada *slice* criando três pontos que juntos geram um triângulo. Um deles será sempre a origem, para que a base do cone fique no plano  $xOz$ , enquanto os outros dois serão pontos no perímetro da circunferência, no sentido dos ponteiros do relógio para a orientação da base ser no sentido negativo do eixo  $Oy$ .

```

1 tuple<float , float , float >* generate_cone(float bottom_radius , float height , int slices , int
stacks , int* total_points){
2
3     vector<tuple<float , float , float >> point_array;
4     int i , j;
5     double alfa = 2 * M_PI / slices;
6     double division_height_step = height / stacks;
7     double division_radius_step = bottom_radius / stacks;
8     // Base
9     for (i = 0; i < slices; i++) {
10         point_array.push_back(make_tuple(bottom_radius * sin(alfa * (i + 1)), 0.0f,
bottom_radius * cos(alfa * (i + 1))));
11         point_array.push_back(make_tuple(bottom_radius * sin(alfa * i), 0.0f, bottom_radius
* cos(alfa * i)));
12         point_array.push_back(make_tuple(0.0f, 0.0f, 0.0f));
13     }

```

Posteriormente, consideram-se circunferências menores ao longo da altura do cone, com saltos ditados pelos valores *division\_radius\_step* e *division\_height\_step*, respetivamente. Para cada uma das *slices*, definem-se dois triângulos orientados para fora do sólido geométrico. Para tal, utilizam-se coordenadas análogas à rotação ao redor do eixo  $Oy$ , por se tratar de uma circunferência.

```

1     for (i = 0; i < stacks; i++) {
2         for (j = 0; j < slices; j++) {
3             double bot_height = division_height_step * i;
4             double top_height = bot_height + division_height_step;
5             double bot_radius = bottom_radius - division_radius_step * i;
6             double top_radius = bot_radius - division_height_step;
7
8             // lados

```

```

9         point_array.push_back(make_tuple(bot_radius * sin(alfa * j), bot_height,
bot_radius * cos(alfa * j)));
10        point_array.push_back(make_tuple(bot_radius * sin(alfa * (j + 1)), bot_height,
bot_radius * cos(alfa * (j + 1))));
11        point_array.push_back(make_tuple(top_radius * sin(alfa * j), top_height,
top_radius * cos(alfa * j)));
12
13        point_array.push_back(make_tuple(top_radius * sin(alfa * j), top_height,
top_radius * cos(alfa * j)));
14        point_array.push_back(make_tuple(bot_radius * sin(alfa * (j + 1)), bot_height,
bot_radius * cos(alfa * (j + 1))));
15        point_array.push_back(make_tuple(top_radius * sin(alfa * (j + 1)), top_height,
top_radius * cos(alfa * (j + 1))));
16    }
17 }
18
19 *total_points = point_array.size();
20
21 tuple<float, float, float>* temp = (tuple<float, float, float>*) malloc(sizeof(tuple<float,
float, float>) * point_array.size());
22 for(int i=0; i< point_array.size(); i++){
23     temp[i] = point_array[i];
24 }
25 return temp;
26 }

```

## 2.3 Plano

Os parâmetros necessários para definir um Plano são: o seu comprimento (*length*) e o tamanho da grelha (*grid\_slices*).

Para tal é fixo o ponto  $(\frac{-length}{2}, 0, \frac{-length}{2})$  e os restantes são calculados realizando translações a partir deste por múltiplos do vetor  $(\frac{length}{2}, 0, \frac{length}{2})$ .

```

1     float delta = length/grid_slices;
2
3     int index = 0;
4
5     float referential_x = -length/2;
6
7     float referential_z = -length/2;
8
9     for(int i = 0; i < grid_slices; i++){
10         for (int j = 0; j < grid_slices; j++){
11             {
12                 point_array[index++] = make_tuple(j*delta+referential_x, 0, i*delta+
referential_z);
13                 point_array[index++] = make_tuple(j*delta+referential_x, 0, (i+1)*delta+
referential_z);
14                 point_array[index++] = make_tuple((j+1)*delta+referential_x, 0, (i+1)*delta+
referential_z);
15
16                 point_array[index++] = make_tuple(j*delta+referential_x, 0, i*delta+
referential_z);
17                 point_array[index++] = make_tuple((j+1)*delta+referential_x, 0, (i+1)*delta+
referential_z);
18                 point_array[index++] = make_tuple((j+1)*delta+referential_x, 0, i*delta+
referential_z);
19             }

```



}

## 2.4 Cubo

Os parâmetros necessários para definir um Cubo são: o seu comprimento (*length*) e o tamanho da grelha (*grid\_slices*).

Inicialmente é gerado e armazenado um plano. Para tal é fixado um ponto ( $(\frac{-length}{2}, \frac{length}{2}, \frac{length}{2})$ ) e gerado um plano ed modo análogo ao processo executado para a diretiva do Plano, ajustado para que seja paralelo a  $xOy$

Posteriormente o plano anterior é rodado 180 graus em torno eixo  $y$  e estes valores são também guardados.

```

1 for (int i = 0; i < *points_total/6; i++) {
2     //back face
3     point_array[index++] = make_tuple(-get<0>(point_array[i]), get<1>(point_array[i]), -
4     get<2>(point_array[i]));
5 }
```

As duas faces finais são geradas por rotações das faces já geradas em torno do eixo  $x$  e  $y$  por um ângulo de 90 graus.

```

1 for(int i=0; i < *points_total/3; i++)
2 {
3     point_array[index++] = make_tuple(get<2>(point_array[i]), get<1>(point_array[i]), -
4     get<0>(point_array[i]));
5 }
6 for(int i=0; i < *points_total/3; i++)
7 {
8     point_array[index++] = make_tuple(get<0>(point_array[i]), -get<2>(point_array[i]),
9     get<1>(point_array[i]));
10 }
```

Note-se que as rotações não causam composição de erro porque todas as multiplicações efetuadas são por números inteiros.

## 2.5 Torus

Como figura extra às pedidas no enunciado, decidimos implementar o modelo de um torus, paralelo ao eixo  $xOz$ .

Os parâmetros necessários para definir um Torus são: o raio interno (*inner\_radius*), externo(*outer\_radius*), o número de divisões verticais(*vertical\_divisions*) e horizontais(*horizontal\_division*). Tal como no caso da esfera os pontos do toros são calculados por rotação de um conjunto de pontos. Neste caso o conjunto de pontos forma uma circunferência centrada no ponto ( $\frac{outer\_radius+inner\_radius}{2}, 0, 0$ ).

De modo a gerar este conjunto é fixo o ponto  $(0, \frac{outer\_radius-inner\_radius}{2}, 0)$  e rodado à volta do eixo  $x$  por ângulos múltiplos de  $\frac{2\pi}{vertical\_divisions}$ , partindo sempre da origem o que resulta em rotações simples de uma circunferência, partindo de um eixo.

```

1 float delta_x = 2*M_PI/vertical_divisions;
2
3 float delta_y = 2*M_PI/horizontal_divisions;
4
5 float pivot_x = 0;
6
7 float pivot_y = (outer_radius-inner_radius)/2;
8
9 float pivot_z = 0;
10
11
```

```

12     for (int i = 0; i < vertical_divisions; i++) {
13
14         master_circle[master_circle_index++] = make_tuple(
15             pivot_x,
16             pivot_y*cos(i*delta_x),
17             pivot_y*sin(i*delta_x) + (outer_radius+inner_radius)/2
18         );
19
20     }

```

Posteriormente são efetuadas rotações dos pontos armazenados por ângulos múltiplos de  $\frac{2\pi}{horizontal\_divisions}$  de modo a gerar todos os pontos do toro, em torno do eixo  $Oy$ .

```

1
2 for (int i = 0; i < horizontal_divisions; i++) {
3
4     for (int j = 0; j < vertical_divisions; j++) {
5
6         point_array[index++] = make_tuple(
7             get<0>(master_circle[j])*cos(i*delta_y) + get<2>(master_circle[j])*sin(i
8             *delta_y),
9             get<1>(master_circle[j]),
10            -get<0>(master_circle[j])*sin(i*delta_y) + get<2>(master_circle[j])*cos(
11            i*delta_y)
12        );
13        point_array[index++] = make_tuple(
14            get<0>(master_circle[(j+1)%vertical_divisions])*cos(i*delta_y) + get<2>(
15            master_circle[(j+1)%vertical_divisions])*sin(i*delta_y),
16            get<1>(master_circle[(j+1)%vertical_divisions]),
17            -get<0>(master_circle[(j+1)%vertical_divisions])*sin(i*delta_y) + get
18            <2>(master_circle[(j+1)%vertical_divisions])*cos(i*delta_y)
19        );
20        point_array[index++] = make_tuple(
21            get<0>(master_circle[(j+1)%vertical_divisions])*cos(((i+1)%
22            horizontal_divisions)*delta_y) + get<2>(master_circle[(j+1)%vertical_divisions])*sin(((i
23            +1)%horizontal_divisions)*delta_y),
24            get<1>(master_circle[(j+1)%vertical_divisions]),
25            -get<0>(master_circle[(j+1)%vertical_divisions])*sin(((i+1)%
26            horizontal_divisions)*delta_y) + get<2>(master_circle[(j+1)%vertical_divisions])*cos(((i
27            +1)%horizontal_divisions)*delta_y)
28        );
29        point_array[index++] = make_tuple(
30            get<0>(master_circle[j])*cos(i*delta_y) + get<2>(master_circle[j])*sin(i
31            *delta_y),
32            get<1>(master_circle[j]),
33            -get<0>(master_circle[j])*sin(i*delta_y) + get<2>(master_circle[j])*cos(
34            i*delta_y)
35        );
36        point_array[index++] = make_tuple(
37            get<0>(master_circle[(j+1)%vertical_divisions])*cos(((i+1)%
38            horizontal_divisions)*delta_y) + get<2>(master_circle[(j+1)%vertical_divisions])*sin(((i
39            +1)%horizontal_divisions)*delta_y),
40            get<1>(master_circle[(j+1)%vertical_divisions]),
41            -get<0>(master_circle[(j+1)%vertical_divisions])*sin(((i+1)%
42            horizontal_divisions)*delta_y) + get<2>(master_circle[(j+1)%vertical_divisions])*cos(((i
43            +1)%horizontal_divisions)*delta_y)
44        );

```

```

33         point_array[index++] = make_tuple(
34             get<0>(master_circle[j])*cos(((i+1)%horizontal_divisions)*delta_y) + get
<2>(master_circle[j])*sin(((i+1)%horizontal_divisions)*delta_y),
35             get<1>(master_circle[j]),
36             -get<0>(master_circle[j])*sin(((i+1)%horizontal_divisions)*delta_y) +
get<2>(master_circle[j])*cos(((i+1)%horizontal_divisions)*delta_y)
37         );

```

---

Como o conjunto de pontos auxiliar define uma circunferência, é necessário calcular o módulo do índice com o tamanho total de modo a permanecer dentro do array e fazer com que os pontos iniciais e finais coincidam.

## Capítulo 3

# Engine

Neste capítulo falaremos do motor 3D, que poderá ser constituído maioritariamente por duas partes:

- Classes
- Parsing de XML
- Engine em si

### 3.1 Classes

Antes de prosseguir para a função de *parsing*, é importante referir que criamos três classes que permitem armazenar o conteúdo lido do *xml*, e desse modo, evitar que o ficheiro tenha de ser lido mais do que uma vez. Temos, por isso, uma classe *Window* que guarda a largura (*width*) e altura (*height*) da janela.

```
1 class Window{
2 public:
3     // Valores por Omissao
4     float width = 512;
5     float height = 512;
6 };
```

Além dessa, a classe *Camara* guarda as coordenadas da sua posição (*position*), o ponto para qual a camara está a olhar (*lookAt*), o vetor "up" (*up vector*) e a projeção (*projection*).

```
1 class Camera{
2 public:
3     float position[3] = {1,2,1}; // Valor por omiss o sugerido por nos
4     float lookAt[3] = {0,0,0}; // valor por omiss o sugerido por nos
5     float up[3] = {0,1,0}; // Valor Padrao segundo o ficheiro de exemplo
6     float projection[3] = {60,1,1000}; // Valor Padrao segundo o ficheiro de exemplo
7 };
```

Por fim, possuímos duas classes que se englobam numa. O *model* que guarda um *array* de n-tuplos do tipo *float* com n igual a 3 que representam as coordenadas de cada ponto. O *group* possui um vetor de apontadores para *models*.

```
1 class Model{
2 public:
3     int size;
4     std::tuple<float , float , float >* figure;
5 };
6
```

```

7
8 class Group{
9 public:
10     std::vector<Model*> models;
11 };

```

---

## 3.2 Parser

O *engine* de conseguir ler um ficheiro de configuração xml. Para tal, seguimos a sugestão do link do *StackOverflow* do enunciado do trabalho. Portanto, escolhemos o *rapidxml* como *XML DOM parser*, por ser *open-source* e ser bastante eficiente em relação aos outros *parsers*.

O nosso *parser* é dividido em várias funções, cada uma corresponde ao *parsing* dos campos do *xml* correspondente a cada classe.

Começando pela função principal, nele começamos por abrir o documento *xml* e preparar para começar a percorrer cada um dos nodos e atributos do ficheiro. O primeiro nodo chamamos de *root\_node* e corresponde ao primeiro *node* (*world*). Posteriormente, fazemos a procura da janela, câmara e grupo. Caso esse campo exista no *xml*, chamamos a função de *parsing* respetiva à classe.

```

1 void parser(char* fileName, Window* window, Camera* camera, Group* group)
2 {
3
4     xml_document<> doc;
5     xml_node<> * root_node = NULL;
6
7     // Leitura do Ficheiro
8     ifstream theFile (fileName);
9     vector<char> buffer((istreambuf_iterator<char>(theFile)), istreambuf_iterator<char>());
10    buffer.push_back('\0');
11
12    // Parsing
13    doc.parse<0>(&buffer[0]);
14
15    // Encontrar o nodo raiz aka world
16    root_node = doc.first_node("world");
17
18    xml_node<> *temp;
19
20    // Janela
21    if((temp = root_node->first_node("window")))
22        parse_window(temp, window);
23
24    // Camara
25    if((temp = root_node->first_node("camera")))
26        parse_camera(temp, camera);
27
28    // Grupo
29    if((temp = root_node->first_node("group")))
30        parse_group(temp, group);
31 }

```

---

Tanto na função da janela quanto na função da câmara, tentamos ser o mais o mais genérico possível, no sentido em que se um dos campos não aparecer no ficheiro, usamos o valor por omissão definido na classe, como se pode ver no excerto de código abaixo.

```

1 void parse_window(xml_node<> *window_node, Window* window){
2     xml_attribute<> *temp;

```

```

3
4     if((temp = window_node->first_attribute("width"))){
5         window->width = atof(temp->value());
6
7     if((temp = window_node->first_attribute("height"))){
8         window->height = atof(temp->value());
9     }
10
11 void parse_camera(xml_node<> *camera_node, Camera* camera){
12     xml_node<> *temp;
13
14     if((temp = camera_node->first_node("position"))){
15         xml_attribute<> *attr;
16         if((attr = temp->first_attribute("x"))){
17             camera->position[0] = atof(attr->value());
18
19             if((attr = temp->first_attribute("y"))){
20                 camera->position[1] = atof(attr->value());
21
22                 if((attr = temp->first_attribute("z"))){
23                     camera->position[2] = atof(attr->value());
24                 }
25
26             if((temp = camera_node->first_node("lookAt"))){
27                 xml_attribute<> *attr;
28                 if((attr = temp->first_attribute("x"))){
29                     camera->lookAt[0] = atof(attr->value());
30
31                     if((attr = temp->first_attribute("y"))){
32                         camera->lookAt[1] = atof(attr->value());
33
34                         if((attr = temp->first_attribute("z"))){
35                             camera->lookAt[2] = atof(attr->value());
36                         }
37
38                     if((temp = camera_node->first_node("up"))){
39                         xml_attribute<> *attr;
40                         if((attr = temp->first_attribute("x"))){
41                             camera->up[0] = atof(attr->value());
42
43                             if((attr = temp->first_attribute("y"))){
44                                 camera->up[1] = atof(attr->value());
45
46                                 if((attr = temp->first_attribute("z"))){
47                                     camera->up[2] = atof(attr->value());
48                                 }
49
50                             if((temp = camera_node->first_node("projection"))){
51                                 xml_attribute<> *attr;
52                                 if((attr = temp->first_attribute("fov"))){
53                                     camera->projection[0] = atof(attr->value());
54
55                                     if((attr = temp->first_attribute("near"))){
56                                         camera->projection[1] = atof(attr->value());
57
58                                         if((attr = temp->first_attribute("far"))){
59                                             camera->projection[2] = atof(attr->value());
60                                         }
61

```

62 }

A função do *Group* é ligeiramente diferente às anteriores porque um *group* possui um nodo *Models* com um número arbitrário de *Model*'s, portanto, temos de recorrer a um ciclo para percorrer todos os nodos. Dentro da função, abrimos e lemos cada ficheiro de cada *Model*, que fora gerado pelo *generator*. Como mencionado anteriormente, um ficheiro *.3d* tem um inteiro seguido por um *array* de tuplos. Seguindo essa lógica, primeiro lemos para um inteiro o valor escrito no ficheiro, que simboliza o número de vértices e só depois é que lemos o *array* de uma só vez, pois sabemos o número de pontos, realizando apenas duas operações de leitura. Finalmente, criamos um *Model*, guardamos tudo na estrutura, e esse objeto é, também, guardado dentro do array de *Models* do objeto *Group*.

```
1 void parse_group(xml_node<> *group_node, Group* group){
2     for(xml_node<> *node_models = group_node->first_node("models")->first_node(); node_models
3     ; node_models = node_models->next_sibling()){
4         // Criar fstream e abrir
5         fstream filestream;
6         filestream.open(node_models->first_attribute()->value(), ios::in | ios::binary);
7
8         // Ler inteiro para o n
9         int n;
10        filestream.read((char*)&n, sizeof(int));
11
12        // Ler array de tuplos
13        tuple<float, float, float>* tuples = new tuple<float, float, float>[n];
14        filestream.read((char*)tuples, sizeof(tuple<float, float, float>) * n);
15
16        // fechar o ficheiro
17        filestream.close();
18
19        // Criar o model, guardar os tuplos e o inteiro no model, guardar o model no group
20        Model* model = new Model;
21        model->figure = tuples;
22        model->size = n;
23        group->models.push_back(model);
24    }
25 }
```

### 3.3 Engine

Finalmente, no ficheiro *engine.cpp*, tratamos da parte gráfica. A primeira função é a função *run()* que é chamada no *main.cpp*, e leva como argumentos os objetos criados igualmente no *main*, isto é, Câmara, janela e grupo, além do *argv* e *argc*. A função começa com a passagem da *camera* e do *group* para uma variável global, pois as funções *renderScene* e *changeSize* não possuem argumentos. A seguir, iniciamos o *GLUT* e a janela, o registro de *Callback* com *glutDisplayFunc* e *changeSize* e o *Callback* com a função *processKeys*. Por fim, a opções do *OpenGL* e chamamos o *glutMainLoop*.

```
1 void run(Window* window, Camera* camera, Group* group, int argc, char* argv[]) {
2     camera_global = camera;
3     group_global = group;
4
5     // init GLUT and the window
6     glutInit(&argc, argv);
7     glutInitDisplayMode(GLUT_DEPTH|GLUT_DOUBLE|GLUT_RGBA);
8     glutInitWindowPosition(100,100);
9     glutInitWindowSize(window->height, window->width);
10    glutCreateWindow("3DEngine");
```

```

11
12 // Required callback registry
13 glutDisplayFunc(renderScene);
14 glutReshapeFunc(changeSize);
15
16 // Callback registration for keyboard processing
17 glutKeyboardFunc(processKeys);
18 //glutSpecialFunc(processSpecialKeys);
19
20 // OpenGL settings
21 glEnable(GL_DEPTH_TEST);
22 glEnable(GL_CULL_FACE);
23
24 // enter GLUT's main cycle
25 glutMainLoop();
26 }

```

Na função *renderScene*, ao chamar a *gluLookAt()*, utilizamos os campos do objeto *Camara* e temos duas funções de desenho, uma *drawAxis* que desenha os eixos e uma *draw* que desenha todas as figuras pedidas pelo *xml*. É de realçar que temos uma variável global que indica se o eixo está ativo ou não, e portanto, se estiver a falso, não desenha os eixos.

```

1 void renderScene(void) {
2
3     // clear buffers
4     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
5
6     // set the camera
7     glLoadIdentity();
8     gluLookAt(camera_global->position[0], camera_global->position[1], camera_global->position
9               [2],
10              camera_global->lookAt[0], camera_global->lookAt[1], camera_global->lookAt[2],
11              camera_global->up[0], camera_global->up[1], camera_global->up[2]);
12
13     // Colocar funcoes de desenho aqui
14     if(axis)
15         drawAxis();
16     draw();
17
18     // End of frame
19     glutSwapBuffers();
20 }

```

Na função *changeSize* na *gluPerspective*, colocamos os campos do objeto nos parâmetros.

```

1 void changeSize(int w, int h) {
2
3     // Prevent a divide by zero, when window is too short
4     // (you cant make a window with zero width).
5     if(h == 0)
6         h = 1;
7
8     // compute window's aspect ratio
9     float ratio = w * 1.0 / h;
10
11     // Set the projection matrix as current
12     glMatrixMode(GL_PROJECTION);
13     // Load Identity Matrix
14     glLoadIdentity();

```



```

15
16 // Set the viewport to be the entire window
17 glViewport(0, 0, w, h);
18
19 // Set perspective
20 gluPerspective(camera_global->projection[0], ratio, camera_global->projection[1],
21               camera_global->projection[2]);
22
23 // return to the model view matrix mode
24 glMatrixMode(GL_MODELVIEW);
25 }

```

A função *drawAxis* desenha os três eixos x, y e z com diferentes cores.

```

1 void drawAxis() {
2     glBegin(GL_LINES);
3     glColor3f(1.0f, 0.0f, 0.0f);
4     glVertex3f(-100.0f, 0.0f, 0.0f);
5     glVertex3f( 100.0f, 0.0f, 0.0f);
6
7     glColor3f(0.0f, 1.0f, 0.0f);
8     glVertex3f(0.0f, -100.0f, 0.0f);
9     glVertex3f(0.0f, 100.0f, 0.0f);
10
11    glColor3f(0.0f, 0.0f, 1.0f);
12    glVertex3f(0.0f, 0.0f, -100.0f);
13    glVertex3f(0.0f, 0.0f, 100.0f);
14
15    glEnd();
16 }

```

Em último lugar, a função *draw* mencionada atrás também recorre a uma variável global para decidir se queremos as figuras geométricas a serem desenhadas com o modo *polygon* ou não. O restante, são dois ciclos *for*s que percorrem o *vector* do objeto *Group* e em cada objeto *Model*, percorremos os vértices, ao mesmo tempo que os desenhamos.

```

1 void draw() {
2     glPolygonMode(GL_FRONT_AND_BACK, polygon ? GL_LINE : GL_FILL);
3     glBegin(GL_TRIANGLES);
4     glColor3f(1.0f, 1.0f, 1.0f);
5     for(int i=0; i<group_global->models.size(); i++){
6         for(int j=0; j<group_global->models[i]->size; j++){
7             glVertex3f(get<0>(group_global->models[i]->figure[j]), get<1>(group_global->models[i]
8             )->figure[j]), get<2>(group_global->models[i]->figure[j]));
9         }
10    }
11    glEnd();
12 }

```

## Capítulo 4

# Resultados

Nesta secção apresentamos os resultados obtidos da execução de ambas as aplicações utilizando os ficheiros de teste fornecidos.

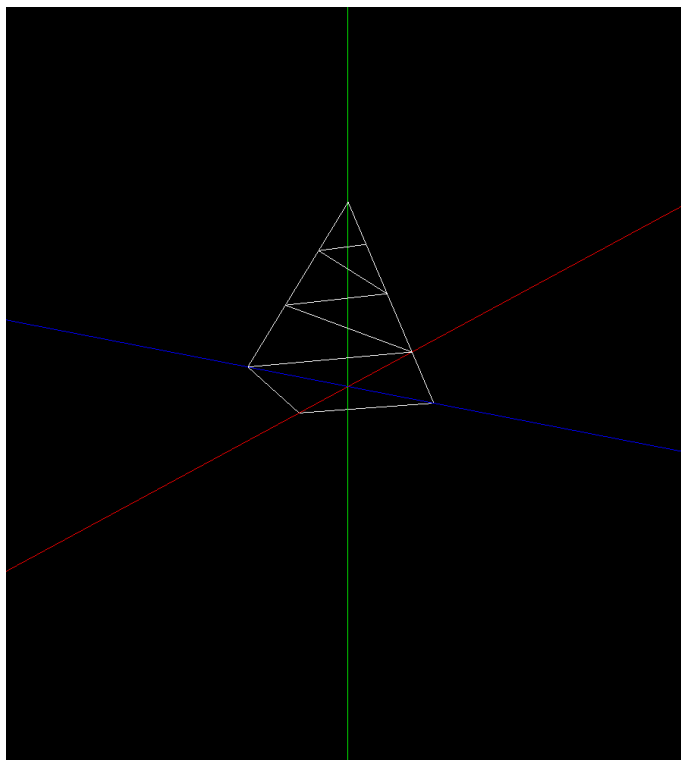


Figura 4.1: Teste 1

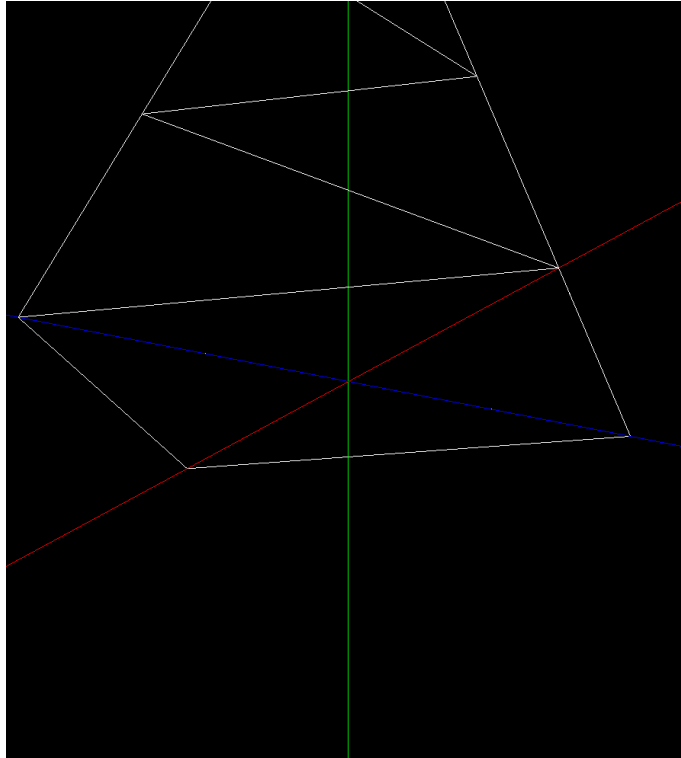


Figura 4.2: Teste 2

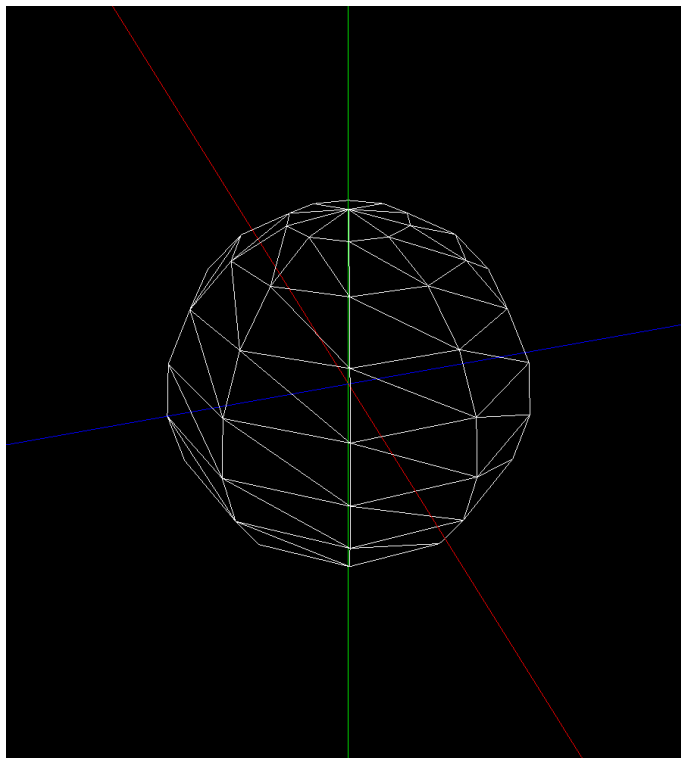


Figura 4.3: Teste 3

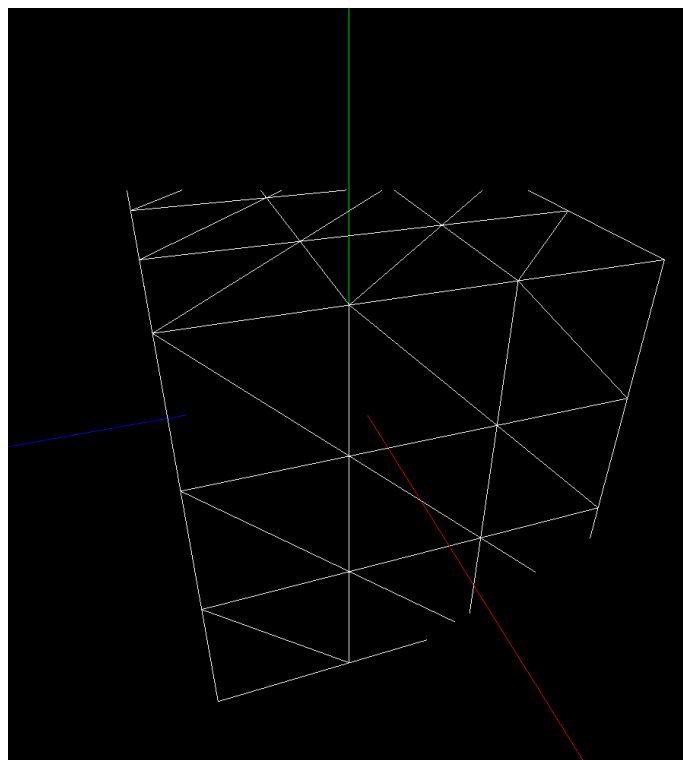


Figura 4.4: Teste 4

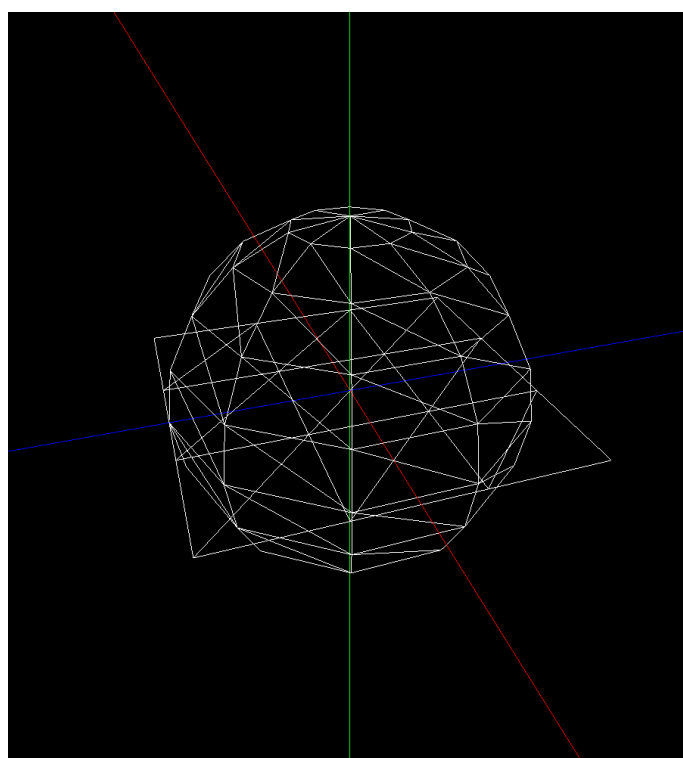


Figura 4.5: Teste 5

## Capítulo 5

# Conclusão

Em suma, ao longo deste relatório foram desenvolvidas duas aplicações, *engine* e *generator*, com sucesso. A aplicação *engine*, que utiliza o *rapixml* para interpretar um ficheiro *xml* fornecido pelo utilizador e gerado pelo *generator*, que é capaz de gerar ficheiros para cinco primitivas gráficas: esfera, cone, plano, cubo e toros.