

# Graphical primitives

Filipe Costa, Miguel Oliveira, Nelson Faria, and José Rodrigues (Grupo 46)

Universidade do Minho, Departamento de Informática, 4710-057 Braga, Portugal  
e-mail: {a85616,a83819,a84727,a85501}@alunos.uminho.pt

## 1 Introdução

Nesta fase 1 do Trabalho Prático foi-nos proposto o desenvolvimento de primitivas gráficas usando o **OpenGL**, sendo essas primitivas um plano, uma caixa, uma esfera e um cone. O **OpenGL** é uma *API* pública usada para o desenvolvimento de aplicações gráficas, nomeadamente aplicações em 3D, sendo que estamos a usá-la em *C++*.

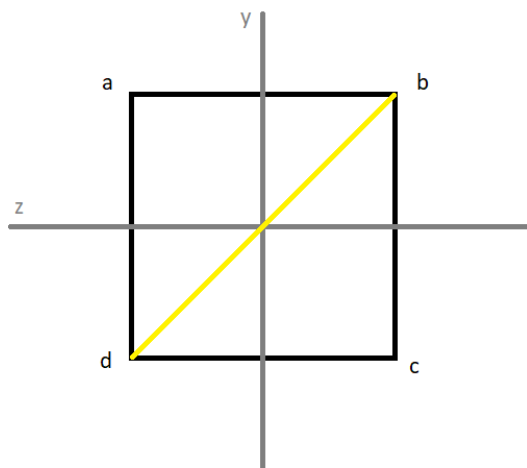
Deste modo, o proposto foi começar a definir os algoritmos necessários ao desenho das 4 primitivas gráficas e escrevê-las num ficheiro. Para isso, definiu-se um programa 'Gerador' que se encarrega de gerar os ficheiros com os respetivos pontos dos vários triângulos necessários ao desenho das figuras. Seguidamente, ao ler-se de um ficheiro escrito em *XML*, pretendia-se que se gerasse as figuras que se encontravam nos ficheiros, cujos nomes desses ficheiros encontram-se no *XML*, sendo que aqui só há a preocupação de se desenhar os triângulos, lendo-se os vértices pela ordem que aparecem no ficheiro. Por fim, na tela, deve-se ter o resultado final das operações anteriores.

## 2 Primitivas Gráficas:

### 2.1 Plano

O plano é neste caso um quadrado, ou seja, apenas necessitámos de conhecer o comprimento do lado desejado. Um plano será obviamente desenhado com base em 4 vértices.

Como pretendemos que o plano fique centrado na origem e ao longo do eixo do *z* e do *y*, e assumindo que o comprimento do lado é igual a *size*, as coordenadas dos vértices são dadas por:



**Figura 1.** Representação dos vértices do plano

O ponto a tem coordenadas:

$$x_a = 0$$

$$y_a = \frac{size}{2}$$

$$z_a = \frac{size}{2}$$

O ponto b tem coordenadas:

$$x_b = 0$$

$$y_b = \frac{size}{2}$$

$$z_b = -\frac{size}{2}$$

O ponto c tem coordenadas:

$$x_c = 0$$

$$y_c = -\frac{size}{2}$$

$$z_c = -\frac{size}{2}$$

O ponto d tem coordenadas:

$$x_d = 0$$

$$y_d = -\frac{size}{2}$$

$$z_d = \frac{size}{2}$$

Decidimos, então, decompor o plano em dois triângulos, sendo os vértices b e d partilhados pelos triângulos.

Temos então o triângulo formado pelos vértices a, b e d:

Decidimos começar por desenhar o vértice d, e fomos, de acordo com a regra da mão-direita (já que desejámos ver o plano), desenhando os restantes vértices, ou seja, em segundo lugar o vértice b e por último o vértice a.

```
fprintf(fp, "%f %f %f\n", 0.0f, -size/2, size/2);  
fprintf(fp, "%f %f %f\n", 0.0f, size/2, -size/2);  
fprintf(fp, "%f %f %f\n", 0.0f, size/2, size/2);
```

**Figura 2.** Representação do triângulo formado pelos vértices a,b e d

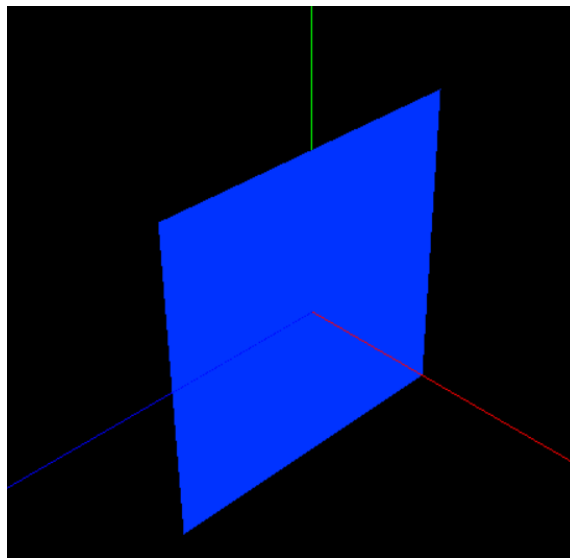
E temos o triângulo formado pelos vértices b, c e d:

Decidimos igualmente começar por desenhar o vértice d, e fomos novamente, em concordância com a regra da mão-direita, desenhando os restantes vértices, ou seja, em segundo lugar o vértice c e por último o vértice b.

```
fprintf(fp, "%f %f %f\n", 0.0f, -size/2, size/2);  
fprintf(fp, "%f %f %f\n", 0.0f, -size/2, -size/2);  
fprintf(fp, "%f %f %f\n", 0.0f, size/2, -size/2);
```

**Figura 3.** Representação do triângulo formado pelos vértices b,c e d

Temos então:

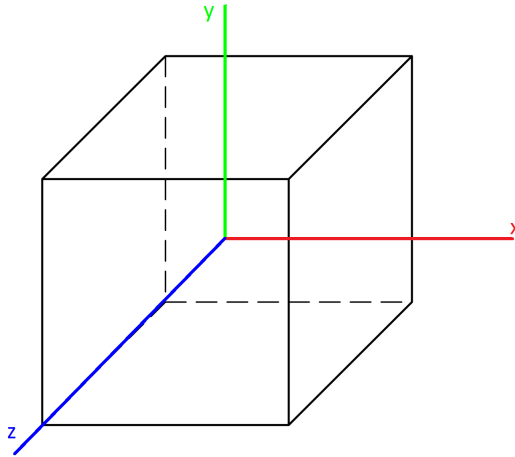


**Figura 4.** Desenho do plano

## 2.2 Caixa

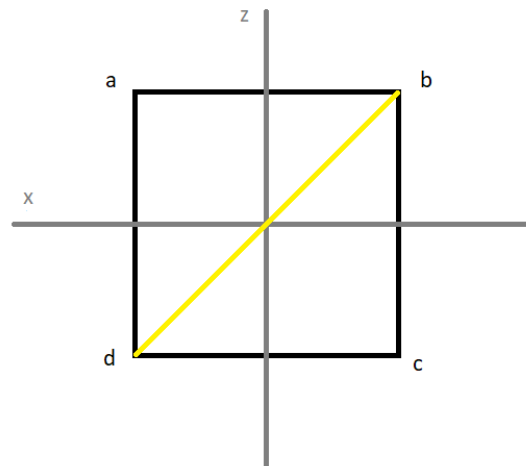
A caixa será centrada na origem e, como é natural, constituída por 6 planos. Tal como concluímos anteriormente, cada plano é constituída por 2 triângulos, ou seja, a caixa será então constituído por 12 triângulos.

Assumindo que  $x$ ,  $y$  e  $z$  são as dimensões desejadas para a construção da caixa, temos que:



**Figura 5.** Representação da caixa

Para construir a tampa e base da caixa temos:



**Figura 6.** Representação dos triângulos que formam a base e a tampa da caixa

Como é evidente, para construir a base todos os vértices do triângulo terão coordenada em  $y = -y/2$  e para construir a tampa todos os vértices do triângulo terão coordenada em  $y = y/2$ , temos então que as restantes coordenadas são dadas por:

O ponto a tem coordenadas:

$$x_a = \frac{x}{2}$$

$$z_a = \frac{z}{2}$$

O ponto b tem coordenadas:

$$x_a = -\frac{x}{2}$$

$$z_a = \frac{z}{2}$$

O ponto c tem coordenadas:

$$x_a = -\frac{x}{2}$$

$$z_a = -\frac{z}{2}$$

O ponto d tem coordenadas:

$$x_a = \frac{x}{2}$$

$$z_a = -\frac{z}{2}$$

Ou seja, temos dois triângulos, um constituído pelos vértices c, b e d e outro constituído pelos vértices d, b e a.

Temos então que a tampa é formada pelos dois triângulos dados por:

```
//Tampa da caixa

fprintf(fp, "%f %f %f\n", -x / 2, y / 2, -z / 2);
fprintf(fp, "%f %f %f\n", -x / 2, y / 2, z / 2);
fprintf(fp, "%f %f %f\n", x / 2, y / 2, -z / 2);

fprintf(fp, "%f %f %f\n", x / 2, y / 2, -z / 2);
fprintf(fp, "%f %f %f\n", -x / 2, y / 2, z / 2);
fprintf(fp, "%f %f %f\n", x / 2, y / 2, z / 2);
```

**Figura 7.** Código relativo à tampa da caixa

E que a base é formada pelos triângulos dados por:

```
//Base da caixa

fprintf(fp, "%f %f %f\n", -x / 2, -y / 2, -z / 2);
fprintf(fp, "%f %f %f\n", x / 2, -y / 2, -z / 2);
fprintf(fp, "%f %f %f\n", -x / 2, -y / 2, z / 2);

fprintf(fp, "%f %f %f\n", x / 2, -y / 2, -z / 2);
fprintf(fp, "%f %f %f\n", x / 2, -y / 2, z / 2);
fprintf(fp, "%f %f %f\n", -x / 2, -y / 2, z / 2);
```

**Figura 8.** Código relativo à base da caixa

Evidentemente, resta codificar os 4 planos que formam as laterais da caixa:

O raciocínio para determinar quais os vértices dos 2 triângulos que constituem cada plano, é idêntico ao utilizado para construir a tampa e a base.

Para a face em que todos os pontos têm coordenada em  $z$  negativa, temos o seguinte código:

```
//Face ao longo do eixo x no z negativo

fprintf(fp, "%f %f %f\n", x / 2, y / 2, -z / 2);
fprintf(fp, "%f %f %f\n", -x / 2, -y / 2, -z / 2);
fprintf(fp, "%f %f %f\n", -x / 2, y / 2, -z / 2);

fprintf(fp, "%f %f %f\n", x / 2, y / 2, -z / 2);
fprintf(fp, "%f %f %f\n", x / 2, -y / 2, -z / 2);
fprintf(fp, "%f %f %f\n", -x / 2, -y / 2, -z / 2);
```

**Figura 9.** Código relativo à face lateral que tem todos os pontos com coordenada  $z$  negativa

Para a face em que todos os pontos têm coordenada em  $z$  positiva, temos o seguinte código:

```
//Face ao longo do eixo x no z positivo

fprintf(fp, "%f %f %f\n", x / 2, y / 2, z / 2);
fprintf(fp, "%f %f %f\n", -x / 2, y / 2, z / 2);
fprintf(fp, "%f %f %f\n", -x / 2, -y / 2, z / 2);

fprintf(fp, "%f %f %f\n", x / 2, y / 2, z / 2);
fprintf(fp, "%f %f %f\n", -x / 2, -y / 2, z / 2);
fprintf(fp, "%f %f %f\n", x / 2, -y / 2, z / 2);
```

**Figura 10.** Código relativo à face lateral que tem todos os pontos com coordenada  $z$  positiva

Para a face em que todos os pontos têm coordenada em  $x$  negativa, temos o seguinte código:

```
//Face ao longo do eixo z no x negativo

fprintf(fp, "%f %f %f\n", -x / 2, y / 2, -z / 2);
fprintf(fp, "%f %f %f\n", -x / 2, -y / 2, z / 2);
fprintf(fp, "%f %f %f\n", -x / 2, y / 2, z / 2);

fprintf(fp, "%f %f %f\n", -x / 2, y / 2, -z / 2);
fprintf(fp, "%f %f %f\n", -x / 2, -y / 2, -z / 2);
fprintf(fp, "%f %f %f\n", -x / 2, -y / 2, z / 2);
```

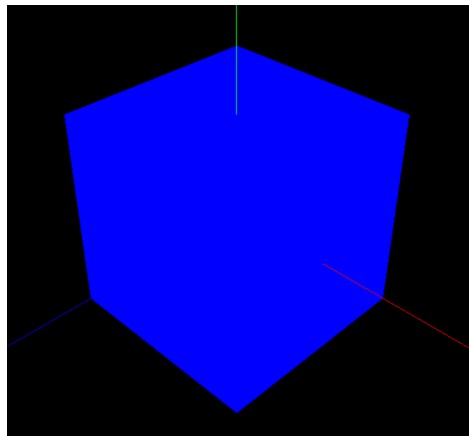
**Figura 11.** Código relativo à face lateral que tem todos os pontos com coordenada  $x$  negativa

Para a face em que todos os pontos têm coordenada em x positiva, temos o seguinte código:

```
//Face ao longo do eixo z no x positivo  
  
fprintf(fp, "%f %f %f\n", x / 2, y / 2, -z / 2);  
fprintf(fp, "%f %f %f\n", x / 2, y / 2, z / 2);  
fprintf(fp, "%f %f %f\n", x / 2, -y / 2, z / 2);  
  
fprintf(fp, "%f %f %f\n", x / 2, y / 2, -z / 2);  
fprintf(fp, "%f %f %f\n", x / 2, -y / 2, z / 2);  
fprintf(fp, "%f %f %f\n", x / 2, -y / 2, -z / 2);
```

**Figura 12.** Código relativo à face lateral que tem todos os pontos com coordenada x positiva

Temos então como resultado final:

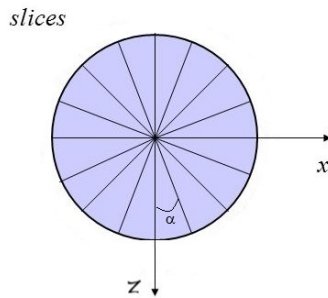


**Figura 13.** Desenho da caixa

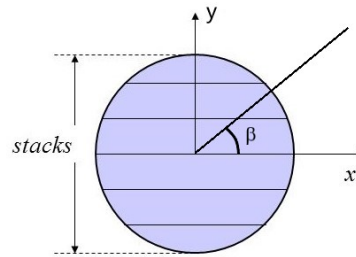
## 2.3 Esfera

Para se construir a esfera foi necessário ter-se a noção de **raio**, **slices** e **stacks**. Os *slices* são o número de fatias/triângulos que se têm ao longo do desenho dos círculos que constituem a esfera, pelo que quantos mais triângulos forem usados para o desenho da esfera, mais perfeitos ficam os círculos, ou seja, a forma da esfera fica mais arredondada. Quanto às *stacks*, são o número de círculos que se encontram todos empilhados de forma a dar a impressão da esfera a 3D. Deste modo, quantas mais *stacks* forem usadas, mais perfeita fica a esfera, uma vez que o espaçamento entre círculos consecutivos será menor.

Deste modo, foi necessário se usar os ângulos  $\alpha$  e  $\beta$  para podermos descobrir os pontos/vértices que constituem a esfera. As figuras a seguir têm uma pequena representação de como é que os ângulos  $\alpha$  e  $\beta$  podem influenciar a descoberta dos pontos dos triângulos que constituem a esfera.



**Figura 14.** Slices



**Figura 15.** Stacks

Decidimos que íamos começar a desenhar cada stack de baixo para cima (utilizando o ângulo  $\beta$ , que não é mais que o ângulo formado no eixo y entre cada stack) e que íamos desenhar cada triângulo/slice que forma cada stack desde o ângulo 0 até ao ângulo  $2\pi$  (utilizando o ângulo  $\alpha$ , que é o ângulo formado em cada slice).

Para tal  $\alpha$  e  $\beta$ :

$$\alpha = \frac{2 * \pi}{slices}$$

$$\beta = \frac{\pi}{stacks}$$

Para determinar o ângulo que permite calcular a altura de cada stack (a que chamamos *angleB*), utilizando o ângulo  $\beta$  que já calculamos anteriormente, temos a seguinte fórmula:

$$angleB = \beta * i - \frac{\pi}{2};$$

Sendo  $i$  uma variável que é incrementada desde 0 até ao número de stacks desejadas.

Para determinar o ângulo que permite desenhar cada slice que forma cada stack, vamos utilizar duas fórmulas:

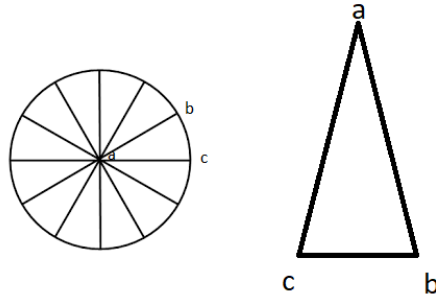
$$angleA = \alpha * j;$$

$$nextAngle = angleA + \alpha;$$

Sendo  $j$  uma variável que é incrementada desde 0 até ao número de slices pretendido.



Dito isto, é simples determinar as coordenadas dos 3 vértices que formam cada triângulo, utilizando as fórmulas para transformar coordenadas esféricas em coordenadas cartesianas:



**Figura 16.** Representação dos pontos que formam cada slice de cada stack

Para o ponto a:

$$\begin{aligned}x_a &= 0 \\y_a &= raio * \sin angleB \\z_a &= 0\end{aligned}$$

Para o ponto b:

$$\begin{aligned}x_b &= raio * \cos angleB * \sin nextAngle \\y_b &= raio * \sin angleB \\z_b &= raio * \cos angleB * \cos nextAngle\end{aligned}$$

Para o ponto c:

$$\begin{aligned}x_c &= raio * \cos angleB * \sin angleA \\y_c &= raio * \sin angleB \\z_c &= raio * \cos angleB * \cos angleA\end{aligned}$$

## 2.4 Cone

Para construir o cone é necessário utilizar o **raio**, a **altura**, as **slices** e as **stacks**. De forma semelhante à esfera, quanto maior for o número de slices e de stacks mais aperfeiçoado fica o cone.

O cone possui um *raio* para o círculo da base, uma *altura*, um número de slices, que indica o número de triângulos presentes num círculo, e um número de *stacks*, que indica o número de círculos que serão usados para desenhar o respetivo cone.

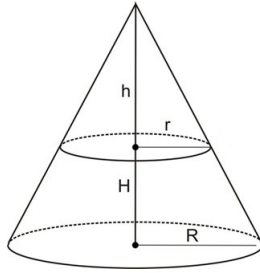
Assim, é necessário saber qual a diferença, relativamente à altura, entre cada círculo. Para tal, sendo **h** a altura e sabendo o número de **stacks**, a *razao* é dada por:

$$razao = \frac{h}{stacks}$$

Por outro lado, em relação ao número de triângulos, temos de saber qual o ângulo onde cada um começa e acaba, que é traduzido pela diferença entre cada lado, que tenha um ponto no centro do círculo, do triângulo. Para isso, sabendo o número de **slices**, a diferença entre cada lado, *ang*, é dado por:

$$ang = \frac{2\pi}{slices}$$

Por fim, precisamos de saber qual o raio que cada círculo tem de ter. Para isso, foi preciso usar a geometria de triângulos, usando a semelhança de triângulos. Assim, se um triângulo pode ser desenhado dentro de outro, é possível saber a dimensão de um lado, através das dimensões do outro triângulo.



**Figura 17.** Semelhança entre triângulos

Tendo isto, sabendo o **R**, raio do círculo da base, **H**, a altura entre a base e o vértice do cone, e o **h**, altura entre o círculo acima da base e o vértice do cone, que é dado por:

$$h = razao * i,$$

sendo o *i* um inteiro de um ciclo *for*, que possui valores pertencentes ao intervalo,  $0 \leq i < stacks$ .

Para calcular o **r**, recorremos ao número de stacks, e ao **R**, raio da base do cone. Para uma *n* stack, o raio é dado fazendo o número de stacks menos o *i* atual do ciclo *for* vezes o raio da base do cone, a dividir pelo número de stacks. Isto, é traduzido pela seguinte expressão:

$$r = \frac{(st - i) * R}{stacks}$$

Em relação aos círculos que constituem o cone, estes são definidos através do mesmo método que os círculos da esfera, desenhados através de ângulos *ahlp*, como foi referido na Figura 14.

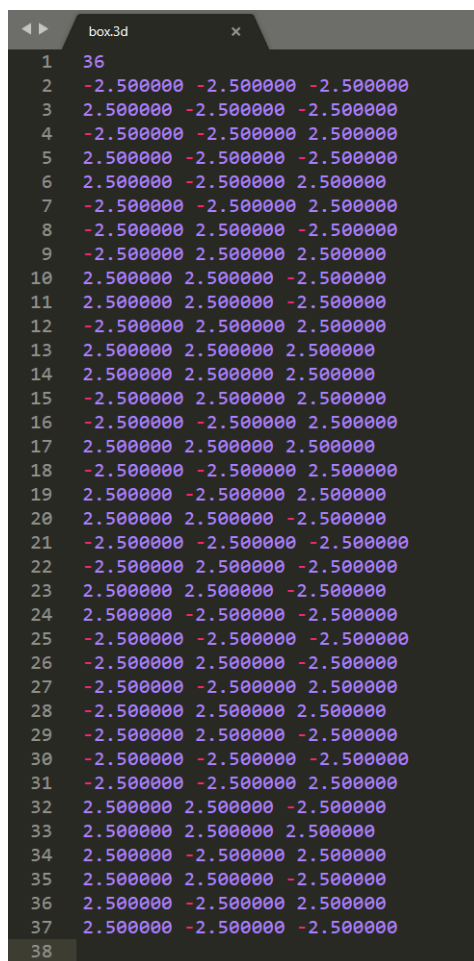
### 3 Generator

O generator é um programa que, ao invés de aplicar os cálculos explicados acima para desenhar diretamente as figuras geométricas tirando partido das ferramentas disponibilizadas pelo *OpenGL*, guarda os vértices pertencentes aos triângulos que resultam dessas contas, que todos juntos formam o desenho, num ficheiro.

O programa generator é capaz de gerar, como referido anteriormente, um plano, uma caixa, uma esfera e/ou um cone. Recebe como argumentos os parâmetros necessários para gerar cada uma das figuras referidas anteriormente para um ficheiro de *output* também especificado como parâmetro.

```
> Generator plane <size> <ficheiro_output>
> Generator box <coord.x> <coord.y> <coord.z> <ficheiro_output>
> Generator sphere <raio> <slices> <stacks> <ficheiro_output>
> Generator cone <raiobase> <altura> <slices> <stacks> <ficheiro_output>
```

Quanto ao formato do ficheiro *output*, optámos por colocar somente aquilo que era necessário para construir as figuras propostas. Deste modo, na primeira linha aparece o número de vértices presentes no ficheiro, nas restantes linhas do ficheiro aparecem os respetivos vértices, como mostra a figura seguinte.



```
1 36
2 -2.500000 -2.500000 -2.500000
3 2.500000 -2.500000 -2.500000
4 -2.500000 -2.500000 2.500000
5 2.500000 -2.500000 2.500000
6 2.500000 -2.500000 2.500000
7 -2.500000 -2.500000 2.500000
8 -2.500000 2.500000 -2.500000
9 -2.500000 2.500000 2.500000
10 2.500000 2.500000 -2.500000
11 2.500000 2.500000 -2.500000
12 -2.500000 2.500000 2.500000
13 2.500000 2.500000 2.500000
14 2.500000 2.500000 2.500000
15 -2.500000 2.500000 2.500000
16 -2.500000 -2.500000 2.500000
17 2.500000 2.500000 2.500000
18 -2.500000 -2.500000 2.500000
19 2.500000 -2.500000 2.500000
20 2.500000 2.500000 -2.500000
21 -2.500000 -2.500000 -2.500000
22 -2.500000 2.500000 -2.500000
23 2.500000 2.500000 -2.500000
24 2.500000 -2.500000 -2.500000
25 -2.500000 -2.500000 -2.500000
26 -2.500000 2.500000 -2.500000
27 -2.500000 -2.500000 2.500000
28 -2.500000 2.500000 2.500000
29 -2.500000 2.500000 -2.500000
30 -2.500000 -2.500000 -2.500000
31 -2.500000 -2.500000 2.500000
32 2.500000 2.500000 -2.500000
33 2.500000 2.500000 2.500000
34 2.500000 -2.500000 2.500000
35 2.500000 2.500000 -2.500000
36 2.500000 -2.500000 2.500000
37 2.500000 -2.500000 -2.500000
38
```

**Figura 18.** Exemplo do ficheiro *output* para o caso de uma caixa(*box*)

## 4 Engine

O programa Engine, que é responsável por mostrar as figuras com o recurso ao *OpenGL*, recebe como *input* um ficheiro *XML* que contém o nome dos ficheiros a desenhar previamente gerados a partir do programa Generator. De seguida, este lê o conteúdo dos ficheiros em que cada linha representa um vértice pertencente a um triângulo. O programa principal agrupa-os em 3 e desenha os triângulos que, ao fim, formam a figura pretendida aquando da chamada do programa.

Nesta parte foi também necessário o uso da biblioteca de funções **TinyXML** para se proceder ao 'parse' (divisão) do ficheiro *XML* (cujo nome dele é *infoXML.xml*) de forma a então se retirarem os respetivos nomes dos ficheiros usados como *input* de vértices.

Quanto ao armazenamento dos vértices na memória, optámos por desenvolver a nossa própria *Estrutura de Dados*, pelo que possuímos assim uma lista de vértices (*ListVertices*) que contém os vértices que estão presentes num dado ficheiro. Caso existam mais ficheiros, serão guardados noutra lista de vértices, uma vez que possuímos um array que, por cada novo ficheiro com vértices, é alocada mais uma lista de vértices para se poder armazenar os vértices da forma associada ao ficheiro. É de salientar que todo o espaço em memória é alocado dinamicamente, pelo que assim é possível se construírem estruturas maiores e gerir o espaço de forma mais eficiente.

Por fim, o desenho das diversas formas geométricas presentes nos ficheiros que fazem parte do ficheiro *XML* foi obtido com o recurso a uma função *standard*, como mostra a figura seguinte:

```
90  /**
91   * Função que desenha os triangulos das formas a partir das listas de vertices
92   */
93  void drawScene() {
94
95      Vertice v,v2,v3;
96      while ((v = nextV(lv[pol])) != NULL) {
97          v2 = nextV(lv[pol]);
98          v3 = nextV(lv[pol]);
99          glBegin(GL_TRIANGLES);
100         glColor3f(0.0f, 0.0f, 1.0f);
101         glVertex3f(getX(v), getY(v), getZ(v));
102         glVertex3f(getX(v2), getY(v2), getZ(v2));
103         glVertex3f(getX(v3), getY(v3), getZ(v3));
104         glEnd();
105     }
106     // Colocar o pointer novamente a 0
107     atualizaPointer(lv[pol]);
108 }
```

**Figura 19.** Código usado para desenhar uma dada primitiva Gráfica a partir de uma lista de vértices que varia dependendo da forma que se pretende visualizar. A variável inteira *pol* serve simplesmente para, caso se pretenda visualizar uma outra forma que eventualmente exista no *array* de listas de vértices, seja possível fazer.

## 5 Conclusão

Com este trabalho, tivemos a oportunidade de explorar o *OpenGL*, nomeadamente a aplicação de princípios da matemática e da Computação Gráfica, de forma a se construírem formas gráficas 3D. De um modo geral, a percepção que conseguimos captar com a realização desta fase 1 relativamente ao desenho das formas baseada somente em triângulos, foi realmente inovadora para nós e ajudou-nos a entender como é que a Computação Gráfica realmente funciona, contudo sempre com a noção que para já o que se fez nesta fase é só o princípio de muito mais que estará para vir nas fases futuras.