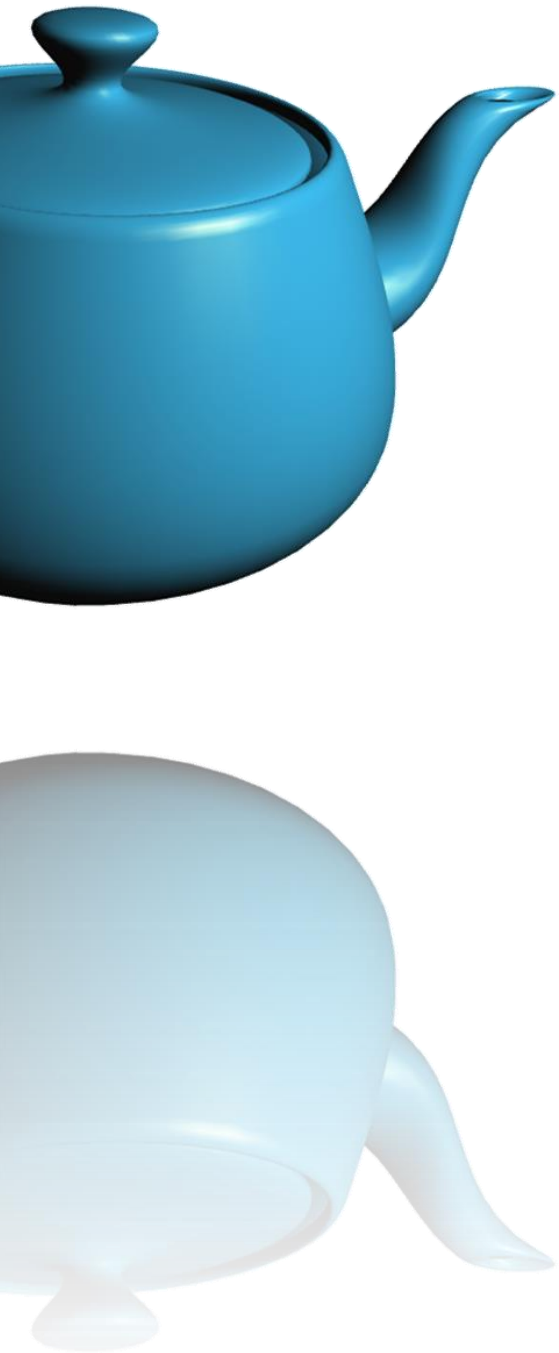
Universidade do Minho

Mestrado integrado em Engenharia Informática

Unidade Curricular de Computação Gráfica



Graphical primitives

Relatório

Trabalho elaborado por:
Carlos Pedrosa - a77320
David Sousa - a78938
Manuel Sousa - a78869

Índice

Índice de ilustrações

Introdução

No âmbito da unidade curricular de Computação Gráfica foi-nos proposto o desenvolvimento de um mini cenário baseado em gráficos 3D. Assim, o projeto encontra-se dividido em 4 fases. Neste relatório pretendemos demonstrar todos os passos que foram efetuados no sentido a cumprir a primeira tarefa proposta.

Nesse sentido, a primeira fase encontra-se divida em duas partes. A primeira prende-se na criação de um programa gerador de ficheiros enquanto que a segunda parte, por sua vez, pressupõe a elaboração de um motor que permita ler um ficheiro de configuração escrito em XML. Por último, o programa terá de desenhar os modelos contemplados no enunciado.

Fase 1

Primitivas gráficas

Por forma a ir de encontro ao proposto pelo enunciado elaboramos as primitivas propostas recorrendo às posições relativas dos vértices dos diferentes modelos.

1. Plano

Para desenhar um plano em OpenGl à custa de vértices, uma vez que, as figuras neste são representadas por triângulos, procedemos à divisão do plano em dois destes, tal como indicado na figura 1.

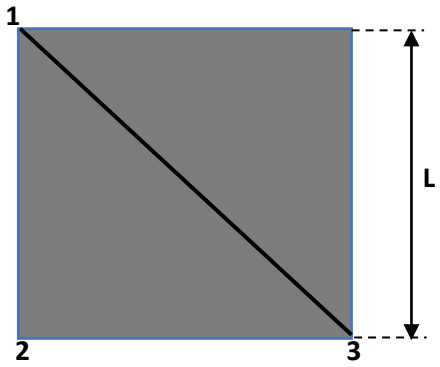


Figura 1 - Divisão de um plano em triângulos

```
void generatePlane(char* l, char* fileName)
```

A função acima indicada permite gerar um plano, esta recebe o tamanho do plano e o nome do ficheiro e opera com base nestes.

```
// Triângulo 1 => Coordenadas do triângulo à esquerda.
file << "" << (-length / 2) << " 0 " << (-length / 2) << "\n";
file << "" << (-length / 2) << " 0 " << (length / 2) << "\n";
file << "" << (length / 2) << " 0 " << (length / 2) << "\n";

// Triângulo 2 => Coordenadas do triângulo à direita.
file << "" << (-length / 2) << " 0 " << (-length / 2) << "\n";
file << "" << (length / 2) << " 0 " << (length / 2) << "\n";
file << "" << (length / 2) << " 0 " << (-length / 2) << "\n";
```

Deste modo, o método usado consiste apenas em determinar as coordenadas que permitiram gerar o plano corretamente, com a orientação adequada à sua visualização.

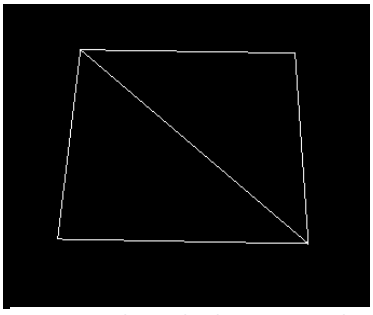
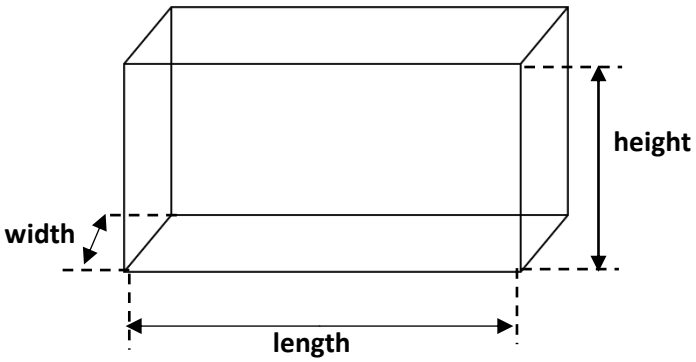


Figura 2 - Plano obtido em OpenGl

2. Caixa

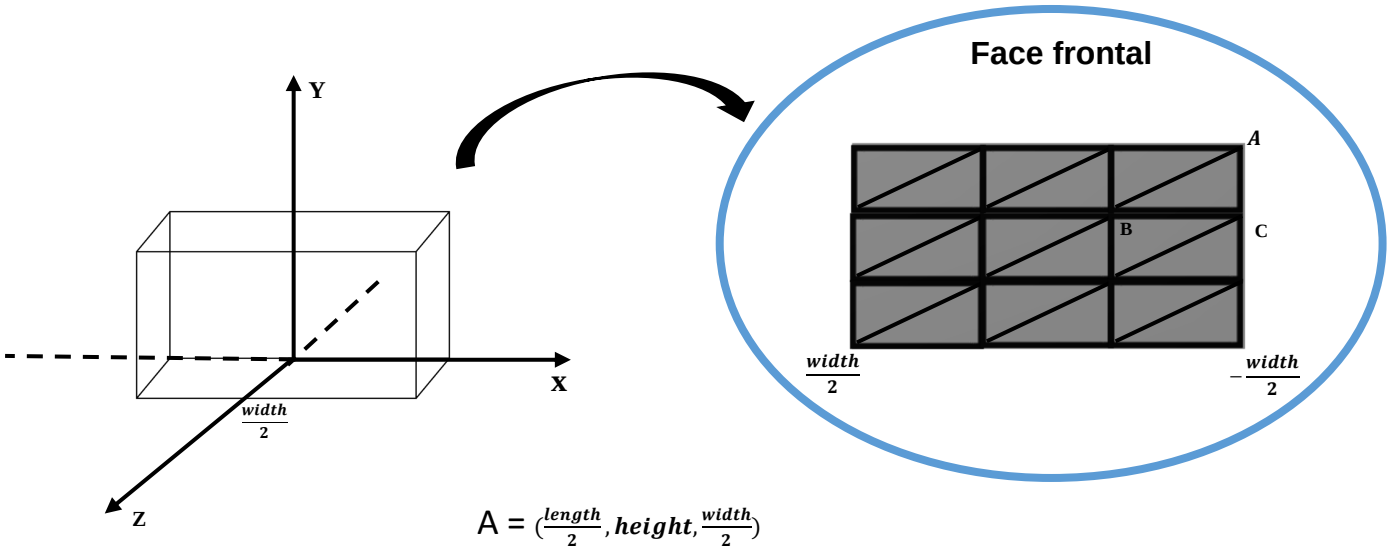
```
void generateBox(char * a, char * b, char * c, int div, char * fileName)
```



Pela figura acima podemos identificar as três medidas principais para o desenho de uma caixa. Estas foram passadas como parâmetros na função *generateBox* por nós criada. Deste modo, temos:

```
double length = atof(a);
double height = atof(b);
double width = atof(c);
```

Por forma a gerar a caixa tivemos de ter em atenção a forma como estavam voltadas as faces desta. Assim, tivemos em consideração como foram desenhados os vértices dos diferentes triângulos, tal como demonstramos na figura seguinte.



$A = (\frac{length}{2}, height, \frac{width}{2})$

$B = (length - \frac{length}{div}, height - \frac{width}{div}, \frac{width}{2})$

$C = (\frac{length}{2}, height - \frac{width}{div}, \frac{width}{2})$

Figura 3 - Desenho da caixa

A caixa foi, assim, desenhada conforme o exemplificado. Cada face ficou assim sujeita a um ciclo *for* que procura iterar dividindo-as segundo a da figura.

```
// Face frontal da caixa.
x = length / 2;
z = width / 2;
for (int i = 0; i < div; i++) {
    y = height;
    for (int j = 0; j < div; j++) {
        file << "" << x << " " << y << " " << z << "\n";
        file << "" << x << " " << (y - divY) << " " << z << "\n";
        file << "" << x << " " << (y - divY) << " " << (z - divZ) << "\n";
        file << "" << x << " " << y << " " << z << "\n";
        file << "" << x << " " << (y - divY) << " " << (z - divZ) << "\n";
        file << "" << x << " " << y << " " << (z - divZ) << "\n";
        y -= divY;
    }
    z -= divZ;
}
```

```
for ( percorrer o número de  divisões vertical) {
        Inicializar uma variável com a medida da caixa que se mantém inalterada;
        for( percorrer o número de divisões horizontal) {
                Percorrer toda a largura/comprimento/altura e ir desenhado os
        triângulos que compõem essa face;
          }
}
```

Figura obtida:

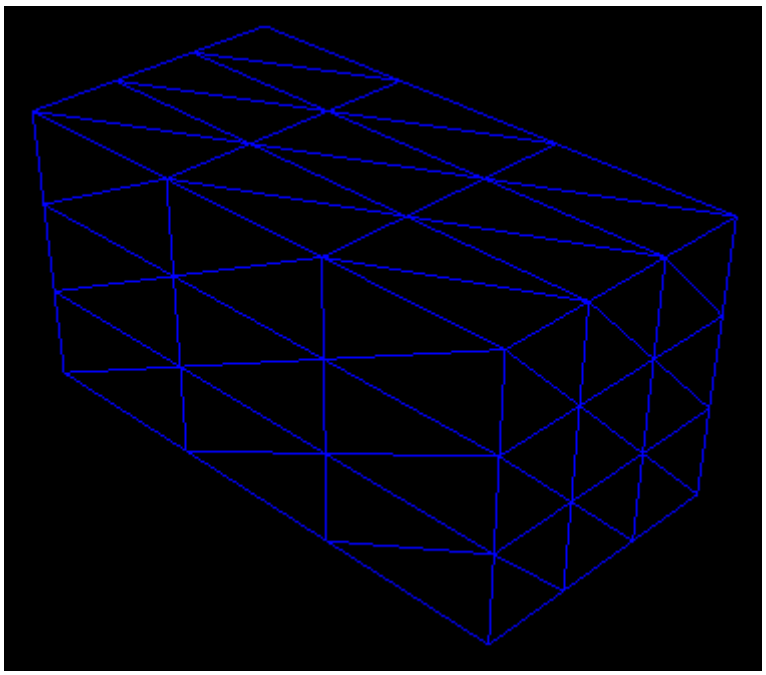
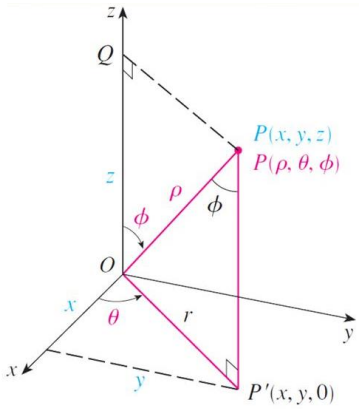


Figura 4 - Caixa obtida em OpenGL

3. Esfera

```
void generateSphere(char* r, char* sl, char* st, char* fileName)
```



$x = r * \sin(\theta) * \sin(\Omega)$
$y = r * \cos(\theta)$
$z = r * \sin(\theta) * \cos(\Omega)$

$\theta \in \{\alpha, \alpha + \Delta(\alpha)\}$
$\Omega \in \{\beta, \beta + \Delta(\beta)\}$

Figura 5 - Coordenadas esféricas

```
int radius = atoi(r);
int slices = atoi(sl);
int stacks = atoi(st);
```

Uma vez mais, construímos uma função que permitisse desenhar um sólido segundo diferentes parâmetros. Neste sentido, a esfera foi traçada com recurso ao raio, o número de *slices* e o número de *stacks*.

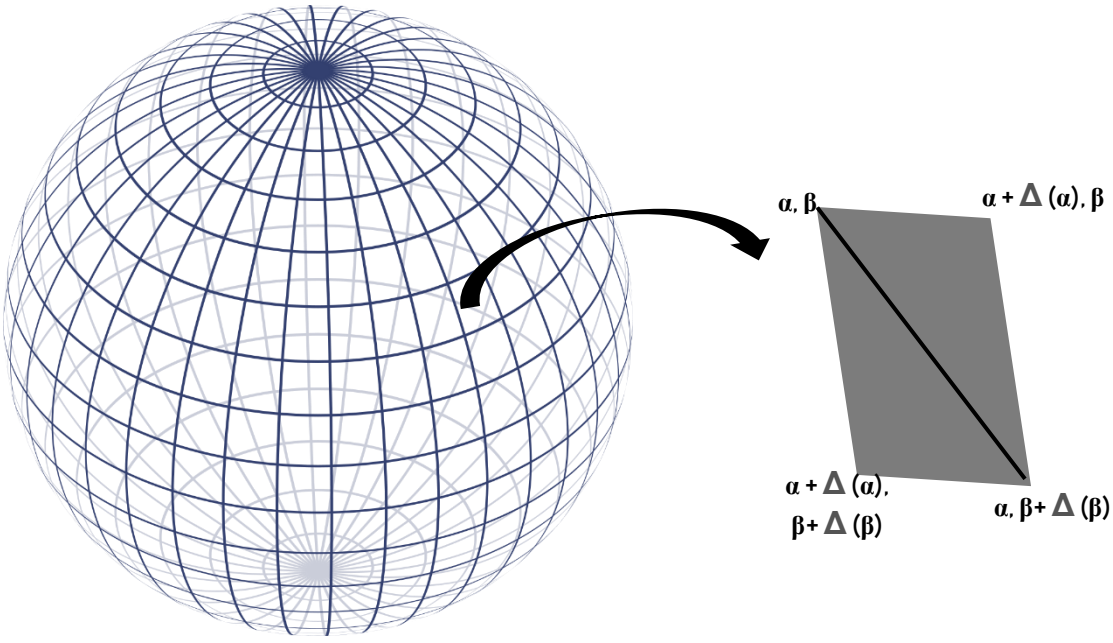


Figura 6 - Demonstração do desenho de uma esfera

```
for (percorrer o número de stacks) {
        Inicializar o alfa em todas as iterações
        for(percorrer o número de slices) {
                Recorrendo às coordenadas polares, desenhar as coordenadas
                correspondentes aos vértices dos triângulos. O ângulo alfa irá variar de
                acordo com uma medida definida: deltaAlfa = 2 * M_PI / slices; O ângulo
                beta irá varias de acordo com deltaBeta = M_PI / stacks;
        }
}
```

Pela figura acima verificámos que as coordenadas de cada segmento da esfera foram obtidas com recurso às coordenadas esféricas.

Assim, inicializamos as variáveis do seguinte modo:

```
double alpha = 0;
double deltaAlpha = (2 * M_PI) / slices;
double beta = 0;
double deltaBeta = M_PI / stacks;
```

Deste modo, os pontos dos diversos triângulos vão ser obtidos à medida que o ângulo vai sendo alterado.

Figura obtida:

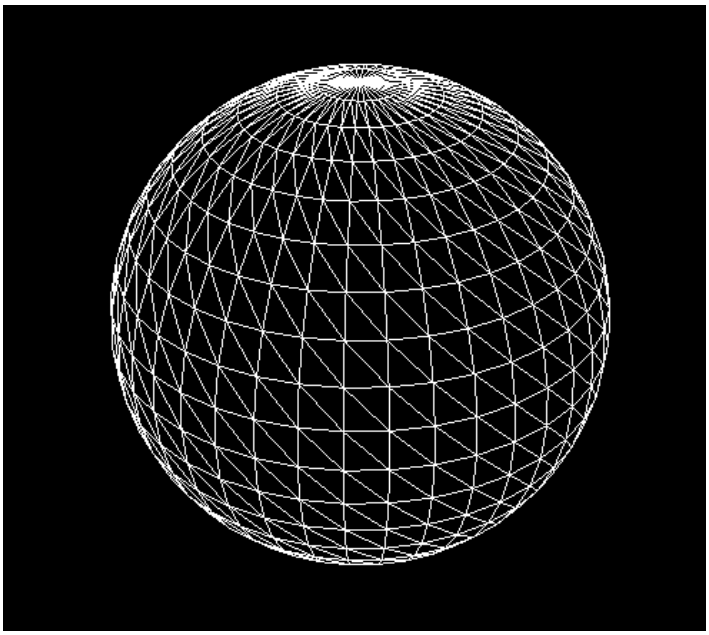


Figura 7 - Esfera obtida em OpenGl

4. Cone

```
void generateCone(char* r, char* h, char* sl, char* st, char* fileName)
double radius = atof(r);
double height = atof(h);
double slices = atof(sl);
double stacks = atof(st);
```

Para desenhar o cone, procedemos de maneira semelhante à efetuada no capítulo anterior. De facto, a base deste é representada por uma circunferência pelo que recorremos novamente ao uso de coordenadas polares.

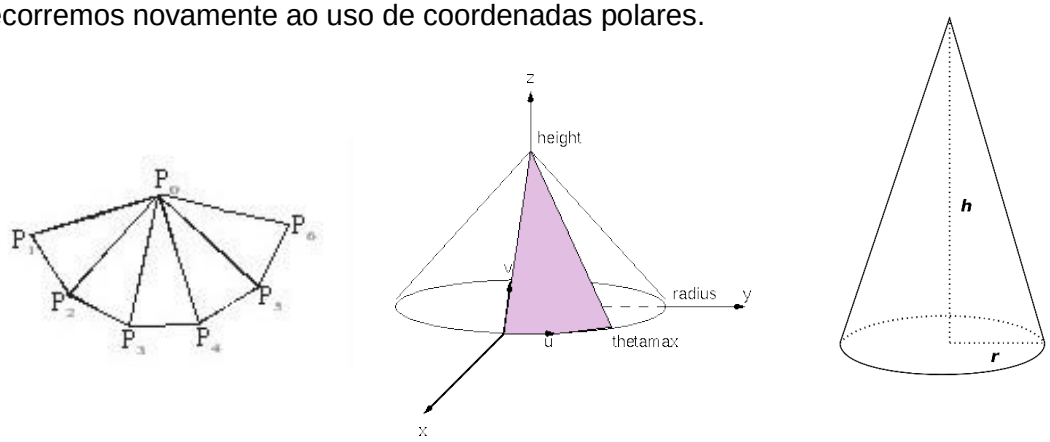


Figura 8 - Demonstração da construção do cone

```
double alpha = 0;
double delta = (2 * M_PI) / slices;
```

Depois de inicializadas corretamente todas as variáveis, passou-se ao desenho das laterais do cone, usando para o efeito o seguinte ciclo *for*:

```
  for (int i = 0; i < slices; i++) {
    file << "" << "0" << " 0 " << "0" << "\n";
    file << "" << (radius * sin(alpha)) << " 0 " << (radius * cos(alpha)) << "\n";
    file << "" << (radius * sin(alpha + delta)) << " 0 " << (radius * cos(alpha + del
ta)) << "\n";
    file << "" << "0 " << height << " 0" << "\n";
    file << "" << (radius * sin(alpha)) << " 0 " << (radius * cos(alpha)) << "\n";
    file << "" << (radius * sin(alpha + delta)) << " 0 " << (radius * cos(alpha + del
ta)) << "\n";
    alpha += delta;
}
}
```

Figura obtida:

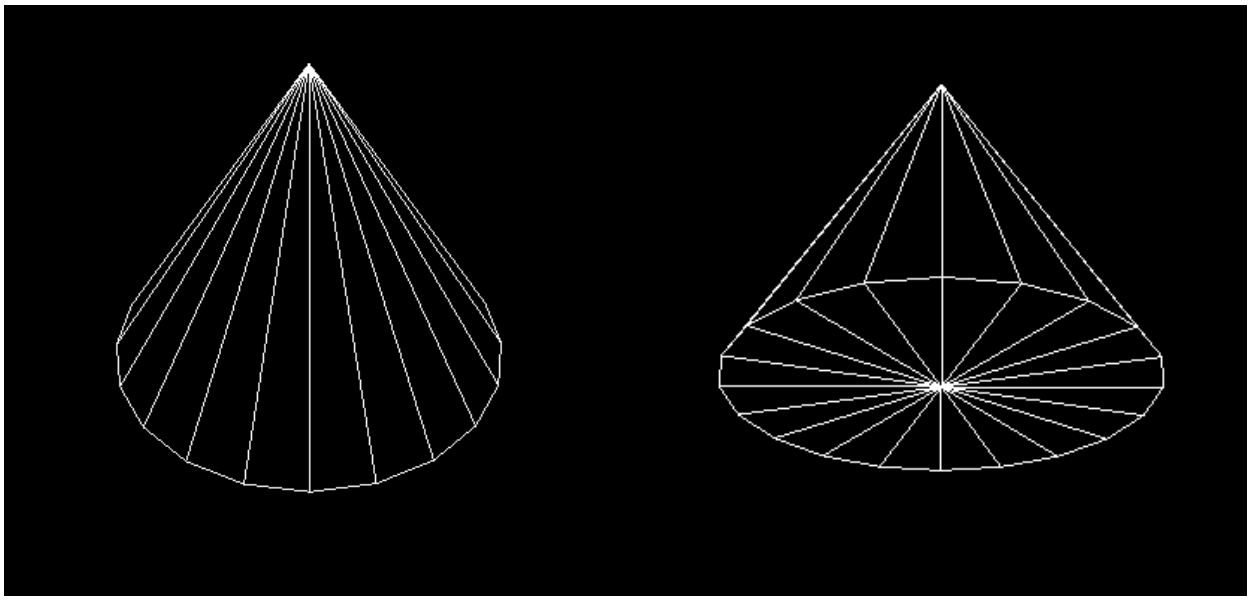


Figura 9 - Cone obtido em OpenGl

Figuras Extras

As figuras extras aqui apresentadas são apenas particularidades dos métodos aqui apresentados.

1. Cilindro

```
void generateCylinder(char* radius, char* ht, char* ss, char* stacks, char* fileName)
```

```
for (int i = 0; i < slices; i++) {
    file << "0 0 0" << "\n";
    file << (r * sin(alpha + delta)) << " 0 " << (r * cos(alpha + delta)) << "\n";
    file << (r * sin(alpha)) << " 0 " << (r * cos(alpha)) << "\n";
    file << (r * sin(alpha + delta)) << " " << height << " " << (r * cos(alpha + delt
a)) << "\n";
    file << (r * sin(alpha)) << " " << height << " " << (r * cos(alpha)) << "\n";
    file << (r * sin(alpha)) << " 0 " << (r * cos(alpha)) << "\n";
    file << (r * sin(alpha + delta)) << " " << height << " " << (r * cos(alpha + delt
a)) << "\n";
    file << (r * sin(alpha)) << " 0 " << (r * cos(alpha)) << "\n";
    file << (r * sin(alpha + delta)) << " 0 " << (r * cos(alpha + delta)) << "\n";
    file << "0 " << height << " 0" << "\n";
    file << (r * sin(alpha)) << " " << height << " " << (r * cos(alpha)) << "\n";
    file << (r * sin(alpha + delta)) << " " << height << " " << (r * cos(alpha + delt
a)) << "\n";
    alpha += delta;
}
```

for (percorrer o número de slices) {

Recorrendo às coordenadas polares, desenhar as circunferências que correspondem à base e ao topo do cilindro. Cada circunferência é assim construída à custa de vários triângulos. Posteriormente, ligar cada uma das circunferências à custa destes. O ângulo alfa irá variar de acordo com uma medida definida: delta = 2 * M_PI / slices;

}

Figura obtida:

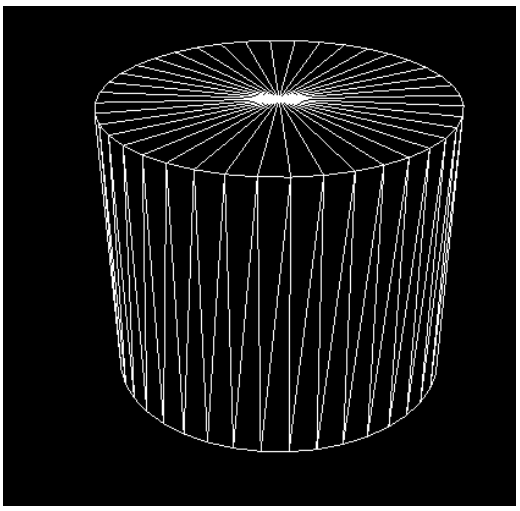


Figura 10- Cilindro obtido em OpenGl

2. Diamante

```
void generateDiamond(char* r, char* fileName)
```

```
double radius = atof(r);
double radiusL = radius + 2;
double alpha = M_PI / 3;
```

Para desenhar o diamante, decompusemos este em várias figuras geométricas, tal como sugerido na figura seguinte.

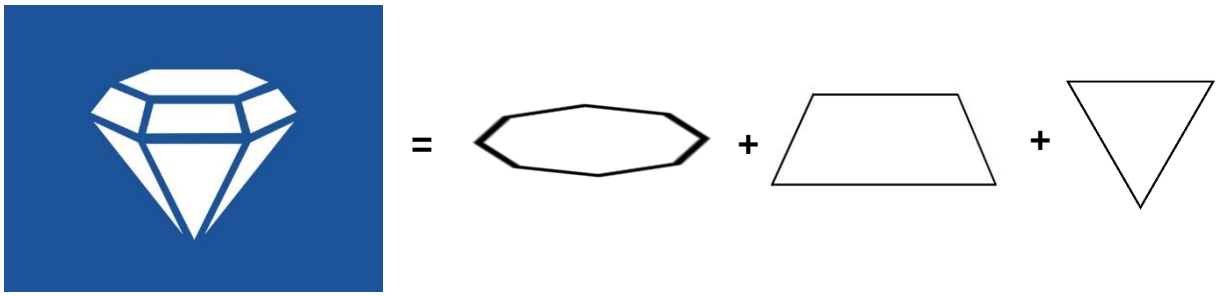


Figura 11 -Decomposição do diamante

Figura obtida:

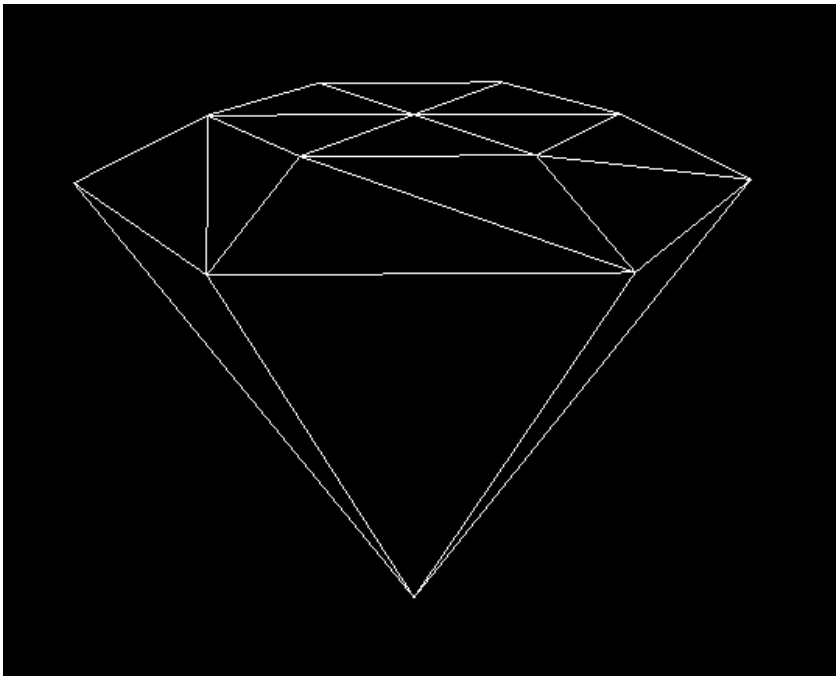


Figura 12 - Diamante obtido em OpenGl

1. Generator

O Generator é composto por uma *main* e por todas as funções que permitem gerar os diferentes sólidos. É na *main* que são recolhidos todos os parâmetros. Estes irão ser usados nas invocações das diversas funções.

```
if (strcmp(argv[1], "plane") == 0) generatePlane(argv[2], argv[3]);
```

No trecho de código acima referido podemos verificar que cada sólido terá a si associado as respetivas funções. Caso se trate, por exemplo de um plano (identificado no primeiro argumento de *Generator*), é chamada a função *generatePlane* com os parâmetros necessários à criação do mesmo.

2. Engine

É importante salientar que o *parser XML* utilizado foi o *tinyxml2*. Para além disso, referimos também o ficheiro *extractCoords* que quando invocado permite extrair de um ficheiro e armazenar em vetores (tuplos de três elementos) os vértices gerados pelo generator para que, posteriormente, seja possível o seu desenho.

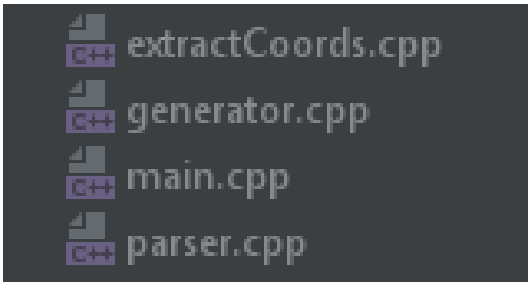


Figura 13 - Estrutura do código

Conclusão

De acordo com o objetivo definido na primeira fase consideramos que este foi definido. De facto, todos os sólidos propostos no enunciado foram desenhados. Para além disso, foi ainda possível adicionar outros extras.

Assim, a presente fase do trabalho permitiu obter um manuseamento mais prático com a ferramenta OpenGl e permitiu perceber como são implementadas alguns dos sólidos que este contém predefinidas.

Bibliografia

- https://simple.wikipedia.org/wiki/Cone#/media/File:Cone_(geometry).svg ;
- https://pt.wikipedia.org/wiki/N-esfera#/media/File:Sphere_wireframe.svg ;
- http://slideplayer.com.br/slide/1234568/ ;
- http://brasilescola.uol.com.br/matematica/volume-paralelepipedo-cubo-cone.htm ;
- http://www.lapiedad.co.cr/paquetes-funerarios-1/diamante .