Computação Gráfica

Trabalho Prático

Luís Miguel Ramos (A83930)
Válter Carvalho (A84464)

6 de Março de 2020

Escola de Engenharia

Grupo nº 26
Mestrado Integrado em Engenharia Informática
Universidade do Minho

Conteúdo

1 Introdução

No âmbito da unidade curricular de Computação Gráfica, foi proposto desenvolver um mini cenário baseado em gráficos 3D.

Este projeto está dividido em quatro fases. O objetivo da primeira fase, que trataremos neste relatório, consiste na criação de um gerador de ficheiro com os vértices do modelo, e na criação de um motor que será capaz de ler o ficheiro XML e exibir a respetiva primitiva gráfica.

Para a criação destas primitivas gráficas tivemos de recorrer a variadas fórmulas que serão especificadas de seguida.

2 Primitivas Gráficas

2.1 Plano

Um plano é constituído por dois triângulos que contêm dois vértices em comum.

O plano gerado poderá estar contido no plano **XZ**, **XY** ou **YZ**. Neste caso, se o valor da variável *axis* for 0, então o plano está contido em **XZ**. Se for 1, o plano está contido em **XY** e caso seja 2, então o plano pertence a **YZ**, como é visível na figura seguinte.

```
// axis: 0->Oxz, 1-> 0xy, 2-> 0yz
void drawPlane(float side1, float side2, int axis) {
```

O plano recebe também duas variáveis *side1* e *side2*, que serão as dimensões do respetivo plano.

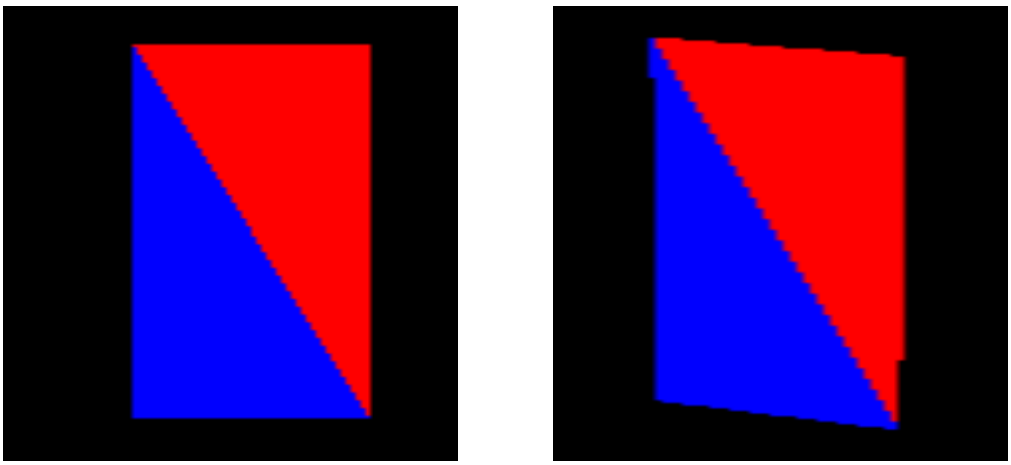


Figura 1: drawPlane(1,3,1)

2.1.1 Fórmulas/Algoritmo

Criou-se duas variáveis novas, sendo cada uma metade das dimensões do plano.

```
Dimensão 1 (c1): side1 / 2.0f

Dimensão 2 (c2): side2 / 2.0f
```

Depois iniciou-se um *switch*, sendo o argumento a variável *axis* responsável por indicar em que plano é que a figura fica.

Consoante o valor da variável *axis*, para calcular os vértices recorremos às variáveis criadas anteriormente *c1* e *c2*.

2.2 Caixa

A caixa é basicamente composta por 3 componentes, que são a dimensão dos X,Y,Z e as divisões (assumimos divisão como "riscos" que dividem a caixa em todas as faces na horizontal e na vertical, por exemplo, uma divisão implica que uma face é dividida em 4 partes com igual área).

```
void drawBox(float x, float y, float z, int d) {
```

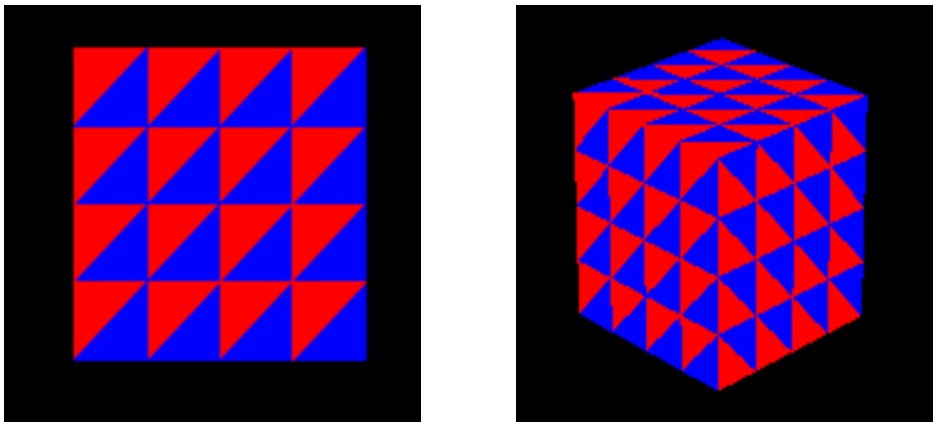


Figura 2: drawBox(2,2,2,3)

2.2.1 Fórmulas/Algoritmo

Essencialmente para desenhar a caixa, guardamos em memória todos os pontos possíveis para cada um dos eixos, tendo em conta o número de divisões. Isto é, para **D** sendo o número de divisões, temos **D**+2 pontos para cada eixo (guardados em array, um para cada dimensão), em que os 2 pontos extra são os da extremidade da caixa. Para além disso, precisamos de saber quanto estes pontos diferem uns dos outros, isto é:

```
float stepX = x/D+1   // Consideramos que x,y,z estão centrados, isto é,

float stepY = y/D+1   // são primeiramente divididos por 2.

float stepZ = z/D+1
```

Para descobrir o valor dos 3 * (**D**+2) pontos, faz-se em cada iteração:

```
xs[i] = x - (stepX*i) // Garantimos a ordem decrescente no array de valores.

ys[i] = y - (stepY*i)

zs[i] = z - (stepZ*i)
```

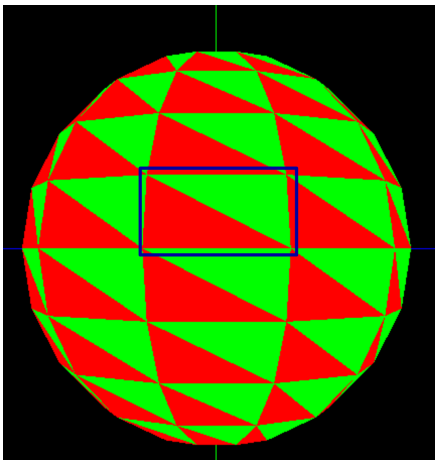
Assim, para cada iteração do ciclo exterior, podemos gerar uma fila de quadrados (construída através de 2 triângulos cada um e geradas no ciclo interior).

2.3 Esfera

A esfera é uma figura sem bases, contém apenas um raio (r).

```
void drawSphere(double r, int stacks, int slices) {
```

De referir que a esfera recebe também uma variável *stacks* que será o número de camadas da esfera. Com isto, tivemos de adaptar o código de forma a que entre cada *stack* e *slice*, a figura passe a ser um plano, ou seja, um conjunto de dois triângulos. É possível verificar isso no quadrado azul assinalado na figura abaixo.



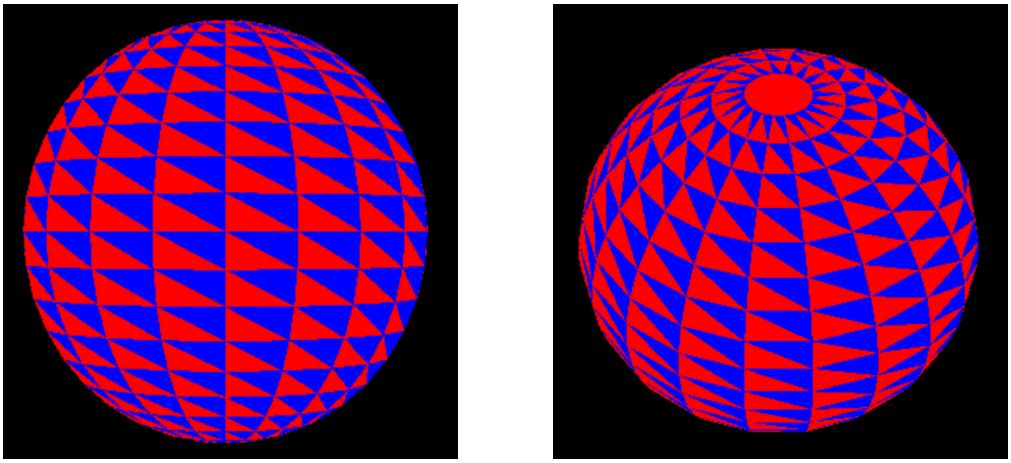


Figura 3: drawSphere(3,20,20)

2.3.1 Fórmulas/Algoritmo

Começou-se por criar duas variáveis, uma para calcular o ângulo de cada fatia e outra para o ângulo de cada camada.

```
Ângulo entre cada fatia (theta): 2 * M_PI / slices;

Ângulo entre cada camada (phi): M_PI / stacks
```

Depois disso iniciamos um ciclo para cada stack, onde calculamos o ângulo da camada a cada iteração e também o da camada seguinte. Assumimos j como a variável incrementada no ciclo

```
Ângulo da camada atual (phi1): phi * j

Ângulo da camada seguinte (phi2): phi * (j+1)
```

Por fim, iniciamos outro ciclo para cada slice, dentro do anterior, e cada iteração calculamos o ângulo da fatia atual, assim como o da próxima fatia. Assumimos n como a variável incrementada no ciclo.

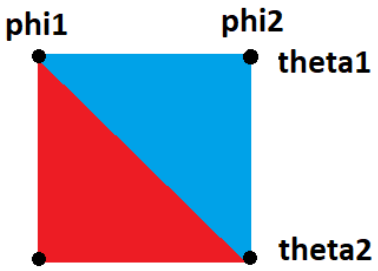
Para calcular o X,Y e Z, recorremos às seguintes fórmulas.

```
Ângulo da fatia atual (theta1): theta * n

Ângulo da fatia seguinte (theta2): theta * (n+1)

X: r * sin(a) * sin (b)
Y: r * cos(b)
Z: r * cos(a) * sin(b)
```

Sendo a e b os valores seguintes, dependendo do local onde o ponto se encontra:

a = $phi1 \parallel phi2$

b = $theta1 \parallel theta2$

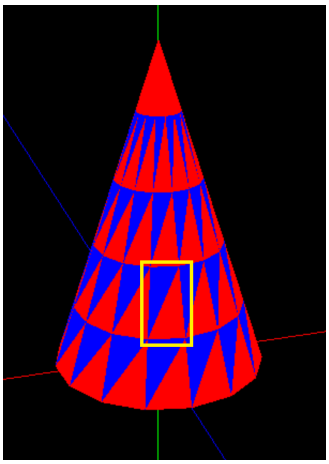


2.4 Cone

O cone é constituído por uma base, com um determinado raio (*radius*) e um determinado número de vértices (*slices*) dessa mesma base, estando esses ligados ao vértice de altura maxima (h), dado também como argumento, como é visível na figura abaixo.

```
void drawCone(float radius, float h, int slices, int stacks) {
```

De referir que o cone recebe também uma variável *stacks* que será o número de camadas do cone. Com isto, tivemos de adaptar o código de forma a que entre cada *stack* e *slice*, a figura passe a ser um plano, ou seja, um conjunto de dois triângulos. É possível verificar isso no quadrado amarelo assinalado na figura abaixo.



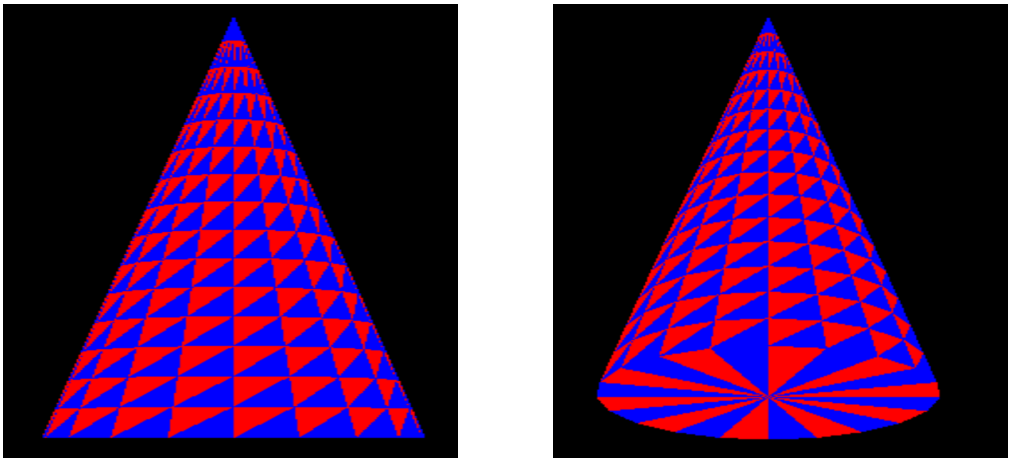


Figura 4: drawCone(2,8,20,15)

2.4.1 Fórmulas/Algoritmo

Começamos por criar três variáveis, uma para o ângulo entre cada fatia, outra para a altura de cada camada e ,por fim, uma para a altura da base do cone.

```
Ângulo entre cada fatia (fat): 2 * M_PI / slices;

Altura de cada camada (cam): h / stacks

Altura da base (alt): -(h / 2)
```

Depois disso iniciamos um ciclo para cada stack, onde calculamos a altura da camada a cada iteração e também a da camada seguinte. Também calculamos o raio da fatia a cada iteração e da iteração seguinte. Assumimos i como a variável incrementada no ciclo

```
Raio da fatia atual (fat1): radius - ((radius / stacks) * i)

Raio da fatia seguinte (fat2): radius - ((radius / stacks) * (i+1))

Altura de camada atual (cam1): alt + (i * cam)
```

```
Altura de camada seguinte (cam2): alt + ((i+1) * cam)
```

Por fim, iniciamos outro ciclo para cada slice, dentro do anterior, e cada iteração calculamos o ângulo da fatia atual, assim como o da próxima fatia. Assumimos n como a variável incrementada no ciclo.

Para calcular o X,Y e Z, recorremos às seguintes fórmulas.

```
Ângulo da fatia atual (j): fat * n

Ângulo da fatia seguinte (k): fat * (n+1)

X: r * sin(a)
Y: alt
Z: r * cos(a)
```

Sendo a,r e alt, os seguintes valores, dependendo do local onde o ponto se encontra:

a = $k \parallel j$

r = $fat1 \parallel fat2$

alt = $cam1 \parallel cam2$

3 Extras

3.1 Cilindro

O cilindro é constituído por duas bases, ambas com o mesmo raio (*radius*) e um determinado número de vértices (*slices*) dessas mesmas bases, estando esses ligados ligados entre si, com comprimento igual altura (*height*), dado também como argumento, como é visível na figura abaixo.

```
void drawCylinder(float radius, float height, int slices, int stacks) {
```

Também recebe o número de *slices* e *stacks*.

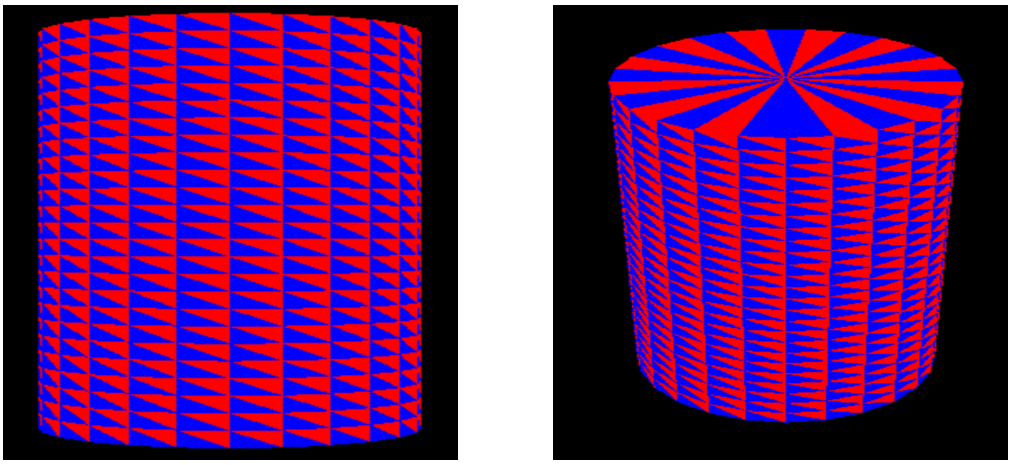


Figura 5: drawCylinder(2,6,25,25)

3.1.1 Fórmulas/Algoritmo

Para desenhar o cilindro, é fundamental saber o ângulo entre *slice*s (para desenhar as bases), assim como a altura de cada *stack*.

```
angle = (2 * M_PI) / slices

h_stack = height / stacks
```

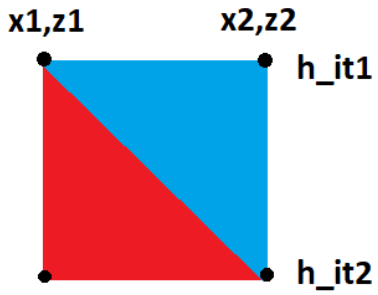
A seguir, usando um processo iterativo (para o número de *slices*), o algoritmo descobre o ponto sobre a *slice* atual e o ponto sobre a próxima, podendo então gerar as bases, porque obtém as componentes X e Z. O Y ou é *-h* ou *h*, dependendo se é a base de cima ou de baixo. Assumimos n como a variável incrementada no processo iterativo.

```
x1 = radius * sin(angle * n)

x2 = radius * sin(angle * (n+1))

z1 = radius * cos(angle * n)

z2 = radius * cos(angle * (n+1))
```

Dentro a iteração das *slices*, pode então desenhar as *stacks*, ou seja, necessita de descobrir a altura da *stack* (j) atual e a da próxima, dadas por:

```
h_it1 = h - (h_stack * j)

h_it2 = h - (h_stack * (j+1))
```

Assim, com tanto X,Y,Z, é possível desenhar todos os pontos para o contexto específico: laterais ou bases.



3.2 Vaso

O vaso é constituído por duas bases, cada uma com um determinado raio (*radius / radius2*). Ambas as bases tem o mesmo número de vértices (*slices*). Esses vértices, estão ligados à base contrária, com um comprimento igual altura (h), dada como argumento, como é visível na figura abaixo.

```
void drawVase(float radius, float radius2, float h, int slices, int stacks) {
```

De referir também que o vaso recebe um numero de *stacks*.

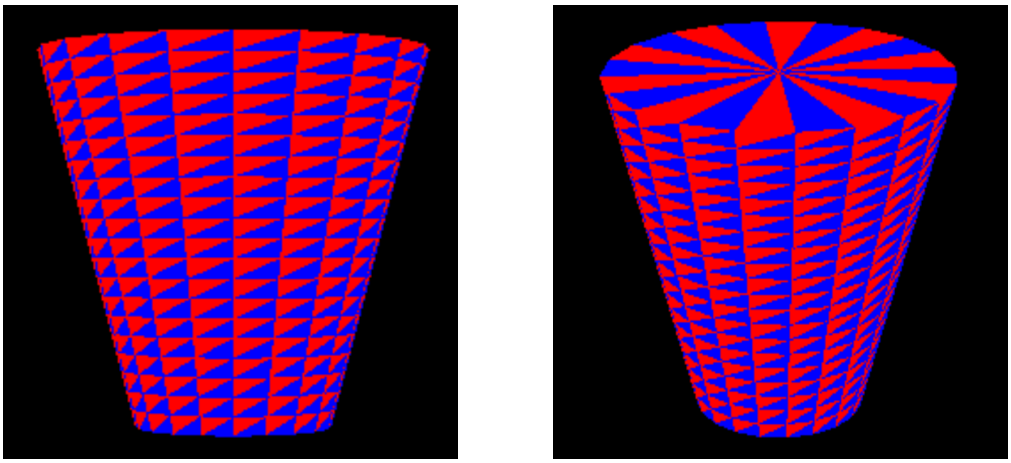


Figura 6: drawVase(2,1,4,20,20)

3.2.1 Fórmulas/Algoritmo

Começamos por criar três variáveis, uma para o ângulo entre cada fatia, outra para a altura de cada camada e ,por fim, uma para a altura da base do vaso.

```
Ângulo entre cada fatia (fat): 2 * M_PI / slices;

Altura de cada camada (cam): h / stacks

Altura da base (alt): -(h / 2)
```

Depois disso iniciamos um ciclo para cada *slice*, onde será calculado cada coordenada da base de baixo do vaso.

```
Ângulo da fatia atual (ang): fat * n

Ângulo da fatia seguinte (k): fat * (n+1)

X: r * sin(a)
Y: alt
Z: r * cos(a)
```

Sendo a, o seguinte valor, dependendo do local onde o ponto se encontra:

a = $k \parallel ang$

Fazemos a mesma coisa para a base de cima, mas com a pequena alteração no Y.

```
Y: -alt
```

Depois disso iniciamos um ciclo para cada *stack*, onde calculamos a altura da camada a cada iteração e também a da camada seguinte. Também calculamos o raio da fatia a cada iteração e da iteração seguinte. Assumimos i como a variável incrementada no ciclo

```
Raio da fatia atual (fat1): radius2 + (radius - radius2) / stacks * i

Raio da fatia seguinte (fat2): radius2 + (radius - radius2) / stacks * (i+1)

Altura de camada atual (cam1): alt + (i * cam)

Altura de camada seguinte (cam2): alt + ((i+1) * cam)
```

Por fim, iniciamos outro ciclo para cada *slice*, dentro do anterior, e cada iteração calculamos o ângulo da fatia atual, assim como o da próxima fatia. Assumimos n como a variável incrementada no ciclo.

Para calcular o X,Y e Z, recorremos às seguintes fórmulas.

```
Ângulo da fatia atual (j): fat * n

Ângulo da fatia seguinte (k): fat * (n+1)

X: r * sin(a)
Y: alt
Z: r * cos(a)
```

Sendo a,r e alt, os seguintes valores, dependendo do local onde o ponto se encontra:

a = $k \parallel j$

r = $fat1 \parallel fat2$

alt = $cam1 \parallel cam2$

4 Gerador

O gerador essencialmente comporta-se da maneira indicada pelo enunciado, isto é, invocamos o executável cujos argumentos dependem do tipo de figura e guarda na pasta *"./files/"* dentro do projeto, dividindo e aproximando a figura em triângulos e guardando todos os estes vértices num único ficheiro. São portanto os comandos válidos:

```
generator sphere <radius> <stacks> <slices> <file>
```

Ex: "generator sphere 5 20 20 Sphere.3d", gerará uma esfera de raio 5, com 20 stacks, 20 slices e grava num ficheiro chamado Sphere.3d.

```
generator box <cx> <cy> <cz> <divisions> <file>
```

Ex: "generator box 5 4 3 0 Box.3d", gerará uma caixa com dimensões 5x4x3 e sem divisões (0). Por outro lado, "generator box 5 4 3 4 Box.3d" gerará uma caixa com 4 divisões. Ambas guardam no ficheiro Box.3d.

```
generator plane <side1> <side2> <axis> <file>
```

Ex: "generator plane 5 4 0 Plane.3d", gerará um plano no eixo xOz (axis = 0) com dimensões 5x4 no plano do referencial e grava no ficheiro Plane.3d. Se axis = 1, gerará no xOy e se axis = 2, gerará no yOz.

```
generator cone <radius> <height> <slices> <stacks> <file>
```

Ex: “generator cone 2 6 10 10 Cone.3d”, gerará um cone com altura 6, raio 2, 10 stacks, 10 slices e guarda no ficheiro Cone.3d.

```
generator cylinder <radius> <height> <slices> <stacks> <file>
```

Ex: “generator cylinder 2 6 10 10 Cylinder.3d”, gerará um cilindro com altura 6, raio 2, 10 stacks, 10 slices e guarda no ficheiro Cylinder.3d.

```
generator vase <radius_top> <radius_bottom> <height> <slices>
<stacks> <file>
```

Ex: “generator vase 5 4 6 10 10 Vase.3d”, gerará um vaso com base de raio 4, abertura do topo com raio 5, altura total de 6 unidades, 10 stacks e 10 slices, gravando no ficheiro Vase.3d.

5 Motor

O motor, nesta fase, apenas lê um ficheiro XML colocado dentro da pasta *"./files/"* dentro do projeto, gerado à mão pelo utilizador, percorrendo todos os ficheiros que foram definidos para serem lidos, dentro deste, e guardando em memória todos os pontos criados pelo gerador para um posterior processamento iterativo de triângulos.

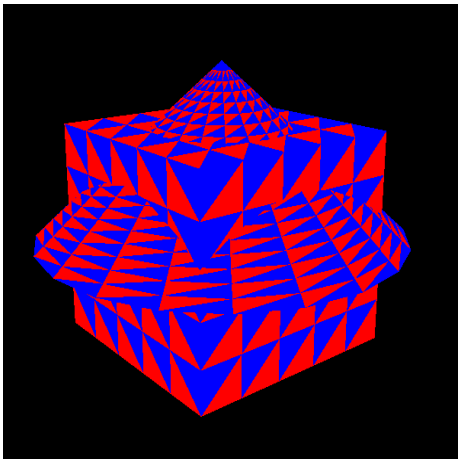
A invocação deste programa é realizada da seguinte forma:

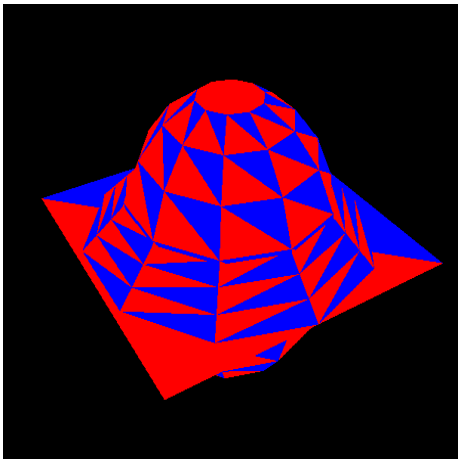
```
engine <file>.xml
```

Alguns exemplos do motor a funcionar:

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <scene>
3      <model file='Box.3d'/>
4      <model file='Cone.3d'/>
5  </scene>
```

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <scene>
3      <model file='Sphere.3d'/>
4      <model file='Plane.3d'/>
5      <model file='Cone.3d'/>
6  </scene>
```





6 Conclusão

Em suma, nesta primeira fase do trabalho, o grupo considera que aprofundou muito o seu conhecimento sobre as ferramentas usadas na unidade curricular, nomeadamente OpenGL.

Foi dada a oportunidade de aprender a construir figuras apenas recorrendo ao uso de triângulos e de alguns algoritmos,e também, aprofundar os conhecimentos da linguagem C++, que será necessária para o resto do projeto.

Por fim, terminada esta primeira fase, o objetivo será começar já a trabalhar na fase seguinte continuando assim aprofundar o conhecimento sobre Computação Gráfica.