



Universidade do Minho
Escola de Engenharia

Computação Gráfica

Trabalho Prático - Fase 1
14 de Março de 2021

Grupo 23

Carlos Filipe Coelho Ferreira, A89542

Joel Salgueiro Martins, A89575

José Carlos Leite Magalhães, A85852

Paulo Ricardo Antunes Pereira, A86475



Conteúdo

1	Introdução	3
2	Gerador	4
2.1	Plano	4
2.2	Caixa	5
2.3	Esfera	8
2.4	Cone	11
3	Motor	14
4	Conclusão	16

1 Introdução

No âmbito da unidade curricular de Computação Gráfica foi proposto desenvolver um motor 3D, baseado num cenário gráfico, cujo propósito passa por explorar todas as suas potencialidades e capacidades. Esta exploração é feita através de diversos exemplos visuais e interativos, sendo utilizadas, para esse efeito, todas as ferramentas abordadas nas aulas práticas da disciplina.

O projeto encontra-se dividido em quatro fases, sendo que este relatório versará sobre a primeira. Esta fase consiste na criação não só de um gerador responsável por conceber modelos, dadas as suas informações, mas também de um motor que será capaz de os analisar, para posteriormente exibir graficamente a primitiva gráfica descrita nos mesmos.

Para a resolução deste problema foram utilizadas diversas fórmulas para as diferentes figuras, sendo que estas serão devidamente fundamentadas nas secções que se seguem.

2 Gerador

O gerador tem como objetivo criar ficheiros XML contendo todos os pontos necessários à criação de um determinado sólido. Sendo esta informação fornecida pelo utilizador, optou-se, de modo a facilitar o processo, por criar dois ficheiros:

- O primeiro - *modelos.cpp* - contém as funções necessárias para desenhar todos os modelos. Quando chamadas, estas encarregam-se de, com os dados devidamente fornecidos, criar a figura pedida e gravar a mesma num ficheiro.
- O segundo ficheiro desta componente - *gerador.cpp* - contém a função *main*, cuja implementação passou pela interpretação de comandos introduzidos pelo utilizador. Quando é inserida informação sobre um determinado sólido, então a função em questão chamará o algoritmo do sólido correspondente, de modo a proceder à sua criação.

Acrescenta-se também que, embora não fosse especificado no enunciado do problema, foi implementada uma função responsável pela criação de ficheiros XML, de modo a tornar o programa mais eficiente e facilitar o uso de vários ficheiros *.3d*.

2.1 Plano

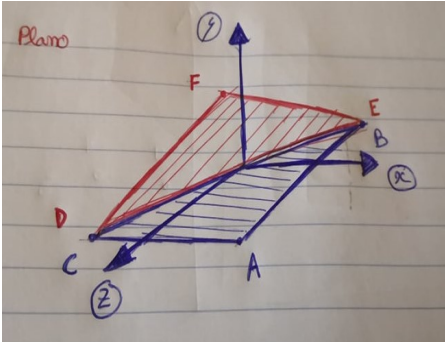
O plano recebe como argumento um inteiro, correspondente ao tamanho do lado do mesmo, e o nome de um ficheiro, para posteriormente ser preenchido, como podemos verificar de seguida:

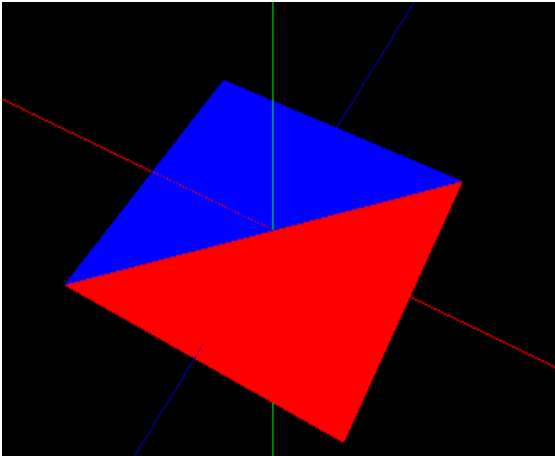
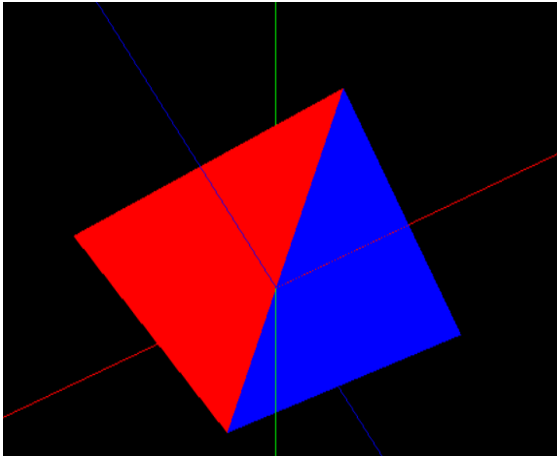
```
void desenhaPlano(float tam, char *ficheiro) {
```

Indo de acordo ao descrito pelo enunciado, o plano estará assente no plano **XoZ**, centrado na origem, e é constituído por dois triângulos, que possuem dois vértices em comum.

Note-se ainda que foi incluído código de modo a que o plano gerado possa ser visto dos dois lados. Para isto, em vez de 2 triângulos, serão criados 4.

De seguida, é possível verificar o rascunho feito pelo grupo, bem como o resultado final:

Rascunho	Pseudo-código
	<pre> //Plano superior //Primeiro triangulo //A fprintf(fd,"%f %f %f\n", tam/2,0.0f,tam/2); //B fprintf(fd,"%f %f %f\n", tam/2,0.0f,-tam/2); //C fprintf(fd,"%f %f %f\n", -tam/2,0.0f,tam/2); //O plano inferior foi deduzido de forma análoga. </pre>

desenhaPlano(1,plano.3d)	
Lado superior	Lado inferior
	

2.2 Caixa

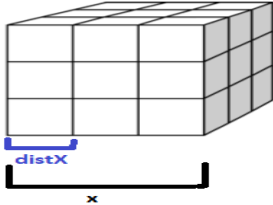
A caixa recebe cinco argumentos: os valores da respetiva dimensão **XYZ**, o número de divisões e o ficheiro de escrita correspondente:

```
void desenhaCaixa(float x, float y, float z, int divisoes, char *ficheiro) {
```

Acrescenta-se que o grupo considerou divisões como traços que dividem a caixa tanto na horizontal como na vertical. Por exemplo, se o número de divisões for 4, então cada face irá ser composta por 25 quadrados e, conseqüentemente, 50 triângulos, como será visto no sólido incluído nesta secção.

Podemos considerar que algoritmo tem duas partes, sendo que a ideia da primeira passa pelo seguinte:

- Criar um *array* para cada dimensão e, após aplicar o número de divisões, guardar no mesmo todos os pontos possíveis resultantes desta divisão. Teremos, então, um total de $nrDivisoes + 2$ pontos, pois, aos já existentes, é necessário adicionar 2 pontos extra, correspondentes à extremidade da caixa.
- Para saber a distância entre pontos, é necessária a introdução de novas variáveis, que armazenem o tamanho das respectivas arestas:

Pseudo-código	Exemplo
<pre>//Seja div = nrDivisões+1 float distX = x / div; float distY = y / div; float distZ = z / div;</pre>	

- Após dividir os valores de x , y e z por 2, de modo a centrar na origem, é necessário encontrar o valor dos $3 * (nrDiv + 2)$ pontos, sendo implementado da seguinte forma:

```
for (int i = 0; i < div + 1; i++) {
    xs[i] = x - (distX * i);
    ys[i] = y - (distY * i);
    zs[i] = z - (distZ * i);
}
```

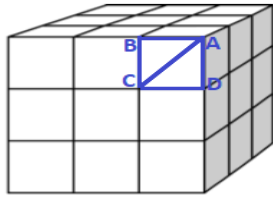
Note-se que, desta forma, é possível garantir que os valores presente nos arrays estão ordenados de forma decrescente.

A segunda parte do algoritmo passa por desenhar cada face, sendo necessário, para isso, a utilização de dois ciclos: o exterior, responsável por percorrer as linhas (cortes horizontais) e o interior, encarregado de percorrer as colunas (cortes verticais).

O processo pode ser resumido de uma forma genérica: estanca-se um canto de uma face sendo que, de seguida, calcula-se o quadrado - mais concretamente, os dois triângulos da sua composição - e percorre-se todos os quadrados da respetiva linha. Aquando o seu término, passa-se então para uma nova linha, até terem sido todas visitadas.

Acrescenta-se ainda que, ao desenhar um determinado quadrado, este pertence a uma face e, por isso, uma das suas coordenadas será constante, podendo ser, consoante

a posição, mínima ou máxima. Tomemos como exemplo a face em que a coordenada **Z** é máxima:

Pseudo-código	Exemplo
<pre> //Percorre as linhas for (int i = 0; i < div; i++) { //Percorre as colunas for (int j = 0; j < div; j++){ CriaTriang(A,B,C); CriaTriang(A,C,D); } } </pre>	 <p>Um diagrama de uma grade 3D com 3 divisões por lado. Uma face específica é destacada com uma borda azul. Os pontos A, B, C e D são marcados nos vértices da face: A no canto superior direito, B no canto superior esquerdo, C no canto inferior esquerdo e D no canto inferior direito. Linhas azuis conectam A-B, B-C, C-D e D-A, formando um retângulo. Além disso, uma linha diagonal azul conecta os pontos A e C.</p>

Analisando a figura e sabendo que os *arrays* estão ordenados por ordem decrescentes, conclui-se que os pontos A,B,C,D são definidos da seguinte maneira:

```

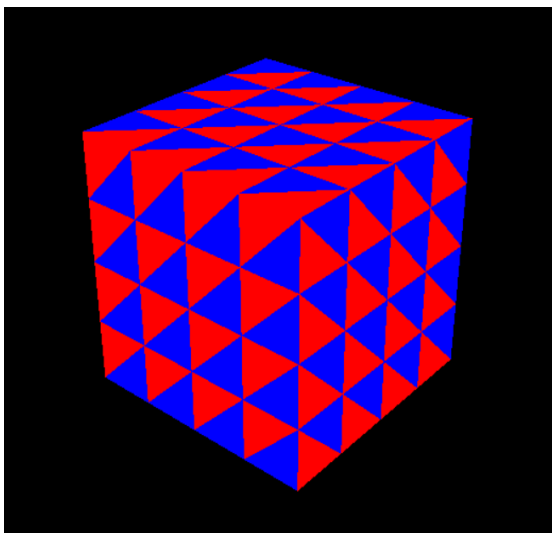
A = (xs[i] , ys[j] , zs[0])
B = (xs[i+1] , ys[j] , zs[0])
C = (xs[i+1] , ys[j + 1] , zs[0])
D = (xs[i] , ys[j + 1] , zs[0])

```

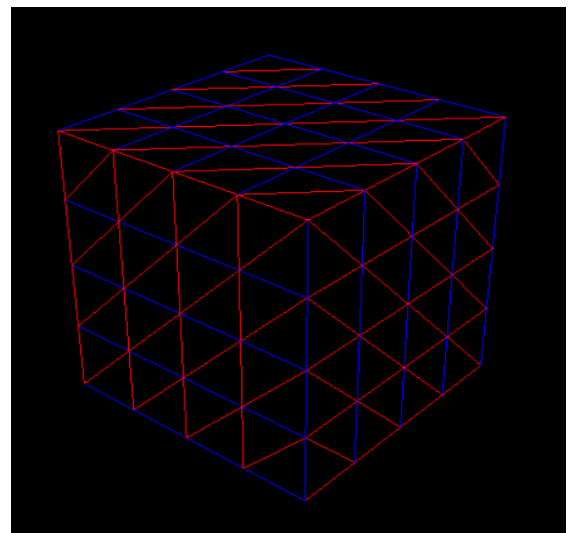
Realça-se a importância da ordem de escrita dos pontos do triângulo, que foi feita de modo a estar em conformidade com a regra da mão direita, abordada nas aulas práticas da disciplina. Por fim, uma aplicação visual do que foi dito:

desenhaCaixa(x=2,y=2,z=2,divisões=3,caixa.3d)

GL Fill



GL Lines



2.3 Esfera

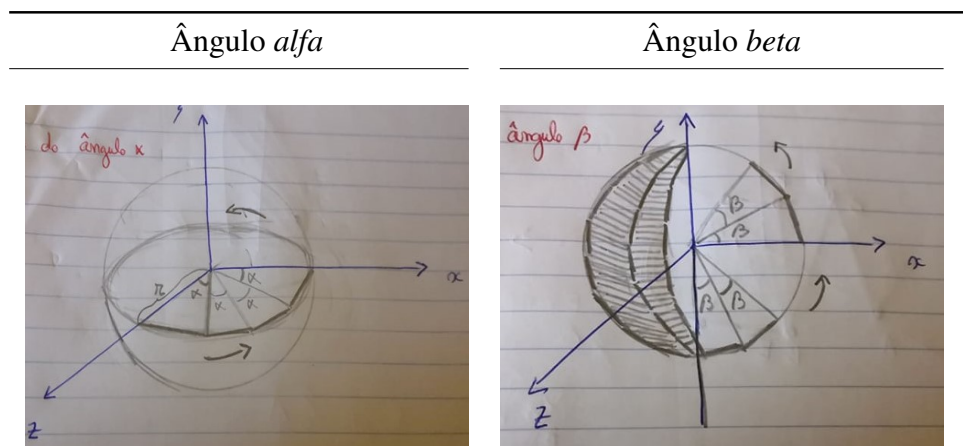
A esfera recebe quatro argumentos: o raio, o número de faces, o número de pilhas e o ficheiro para posterior escrita:

```
void desenhaEsfera(float raio, int faces, int pilhas, char* ficheiro) {
```

Iniciou-se o processo por definir o valor dos ângulos *alfa* e *beta*, correspondendo estes a:

```
//Ângulo de cada fatia:  
float alfa = (2 * M_PI) / faces;  
  
//Ângulo de cada pilha:  
float beta = M_PI / pilhas;
```

O modo de variação dos referidos ângulos pode ser visto no rascunho utilizado pelo grupo:



De seguida, inicia-se um ciclo exterior para cada pilha, onde se procede à atualização do ângulo referente às camadas:

```
for (int i = 0; i < pilhas; i++) {  
    //Ângulo da pilha da atual iteração  
    betaAnterior = betaAtual;  
  
    //Ângulo da pilha seguinte  
    betaAtual += beta;  
}
```

O próximo passo do algoritmo passa por iniciar um novo ciclo interior para cada fatia, onde, analogamente se calcula o ângulo (denominado *alpha*) da fatia atual, bem como o da próxima fatia:

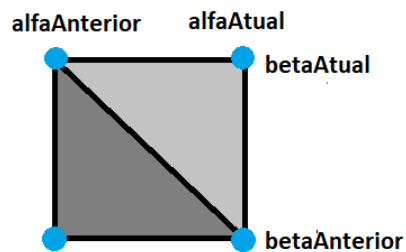

```
for (int j = 0; j < faces; j++) {
    //Ângulo da fatia da atual iteração
    alfaAnterior = alfaAtual;

    //Ângulo da fatia seguinte
    alfaAtual += alfa;
}
```

Por fim, é necessário implementar o processo de cálculo de coordenadas, que é feito da seguinte forma:

```
//Para a coordenada X:
X: raio * sin(alfa) * cos(beta)
//Para a coordenada Y:
Y: raio * sin(beta)
//Para a coordenada Z:
Z: raio * cos(alfa) * cos(beta)
```

O valor de *alfa* e *beta* variam consoante o local, portanto, teremos:



É possível verificar a aplicação geral dos conceitos definidos anteriormente:

Pseudo-código	Output
<pre>for (int i = 0; i < pilhas; i++) { betaAnterior = betaAtual; betaAtual += beta; for (int j = 0; j < faces; j++) { alfaAnterior = alfaAtual; alfaAtual += alfa; CriaTri(A,B,C); CriaTri(C,B,D); } }</pre>	

Os pontos A, B, C e D são definidos da seguinte maneira:

```
//Seja r o raio
//Seja alAnt a variável que representa alfaAnterior
//Seja beAnt a variável que representa betaAnterior
A:(r*sin(alAnt)*cos(beAnt), r*sin(beAnt), r*cos(alAnt)*cos(beAnt));

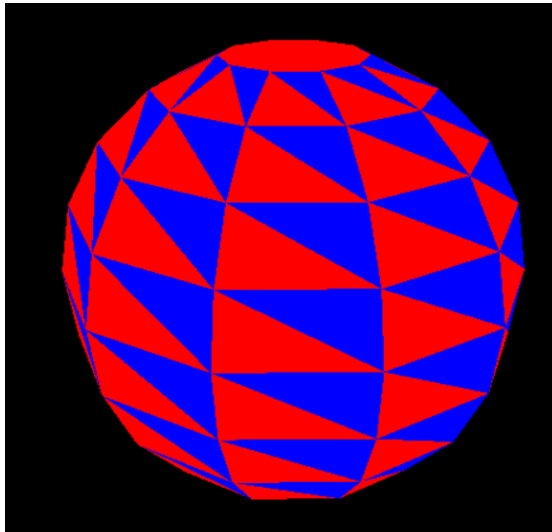
B:(r*sin(alfaAtual)*cos(beAnt), r*sin(beAnt), r*cos(alfaAtual)*cos(beAnt));

C:(r*sin(alAnt)*cos(betaAtual), r*sin(betaAtual), r*cos(alAnt)*cos(betaAtual));

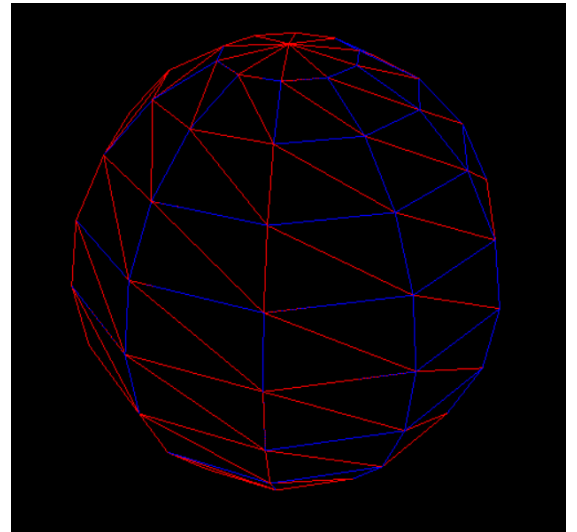
D:(r*sin(alfaAtual)*cos(betaAtual), r*sin(betaAtual), r*cos(alfaAtual)*cos(betaAtual));
```

desenhaEsfera(raio=1,faces=10,pilhas=10,esfera.3d)

GL Fill



GL Line



2.4 Cone

O cone recebe cinco argumentos: o raio da base, altura, o número de faces (cortes verticais), o número de pilhas (cortes horizontais) e o ficheiro de escrita correspondente:

```
void desenhaCone(float raio, float altura, int faces, int pilhas, char* ficheiro) {
```

O algoritmo será essencialmente dividido em duas partes: desenhar a base e desenhar as respetivas laterais. Esta base irá estar assente no plano **XoZ**, e o vértice encontrar-se-á no eixo **Y**.

É necessário, em primeiro lugar, adicionar duas novas variáveis, referentes ao ângulo entre cada face e à altura de cada pilha:

```
//Altura de cada pilha
float alturaCamada = altura/pilhas;

//Ângulo entre cada face
float alfa = (2 * M_PI) / faces;
```

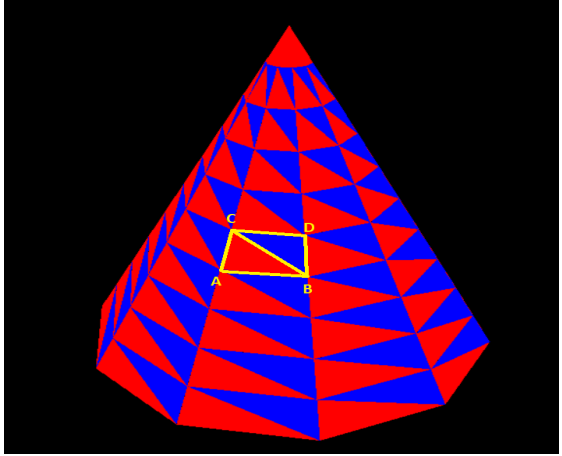
De seguida trataremos da base do cone. É um algoritmo relativamente simples pois sabemos de antemão que a componente **Y** é nula. Para determinar **X** e **Z** aplicou-se o seguinte:

Pseudo-código	Base de um cone
<pre>for (int j = 0; j < faces; j++) { alfaAnterior = alfaAtual; alfaAtual += alfa; CriaTri(A,B,C); } //Sendo A,B,C: A: (raio * sin(alfaAnterior), 0.0f, raio * cos(alfaAnterior)); B: (0.0f, 0.0f, 0.0f); C: (raio * sin(alfaAtual), 0.0f, raio * cos(alfaAtual));</pre>	

Posteriormente iniciam-se dois ciclos:

- O **exterior** para cada pilha, no qual é calculada, a cada iteração, a altura da camada atual e da seguinte. É, também, calculado o raio da fatia atual e da iteração seguinte, seguindo a lógica aplicada na esfera.. Realça-se que estes cálculos são necessários pois à medida que subimos no eixo **Y**, o raio vai diminuindo;

- O segundo ciclo é **interior** para cada face sendo que, em cada iteração, é calculado não só o ângulo da fatia atual, mas também o ângulo da próxima;

Pseudo-código	Output
<pre> //Ciclo referente às pilhas for (int i = 0; i < pilhas; i++) { alturaAnterior = alturaAtual; alturaAtual += alturaFatia; raioAnterior = raio - (raio * alturaAnterior) / altura; raioAtual = raio - (raio * alturaAtual) / altura; //Ciclo referente às faces for (int j = 0; j < faces; j++) { alfaAnterior = alfaAtual; alfaAtual += alfa; CriaTri(A,B,C); CriaTri(B,D,C); } } </pre>	

Os pontos A, B, C e D são deduzidos através das seguintes fórmulas:

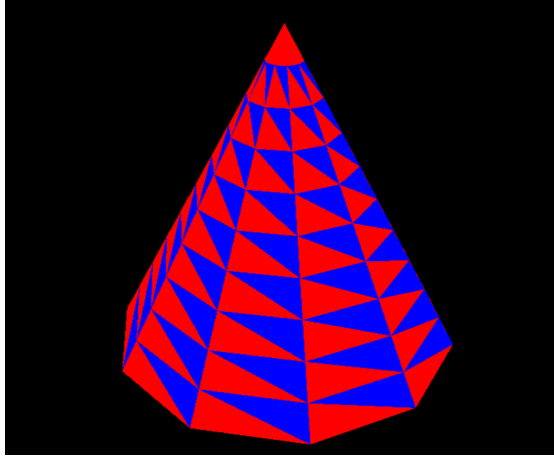
```

//Ponto A
A: (raioAnterior * sin(alfaAnterior), alturaAnterior, raioAnterior * cos(alfaAnterior));
//Ponto B
B: (raioAnterior * sin(alfaAtual), alturaAnterior, raioAnterior * cos(alfaAtual));
//Ponto C
C: (raioAtual * sin(alfaAnterior), alturaAtual, raioAtual * cos(alfaAnterior));
//Ponto D
D: (raioAtual * sin(alfaAtual), alturaAtual, raioAtual * cos(alfaAtual));

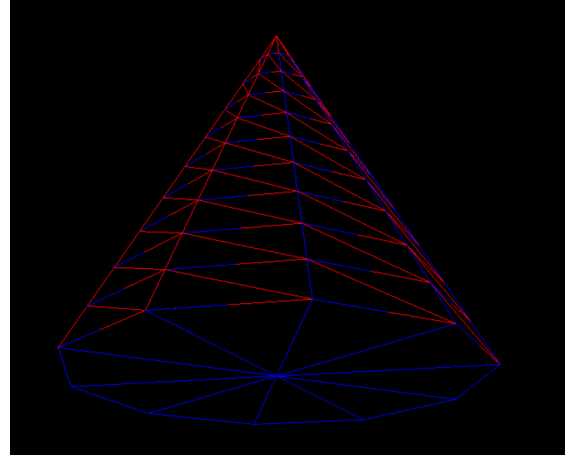
```

```
desenhaCone(raio=2,altura=3,faces=10,pilhas=10,cone.3d)
```

GL Fill



GL Lines



3 Motor

O motor está encarregue de ler um ficheiro XML, criado pelo utilizador, para posteriormente gerar o output *3D*, baseando-se na informação contida nos mesmos. No nosso caso concreto, este ficheiro denomina-se como *primitivas.xml* e encontra-se na pasta *Modelos*, juntamente com os diversos ficheiros com extensão *.3d*.

O processo é simples:

- Inicia-se o processo de leitura do ficheiro XML e de todos os ficheiros nele presentes;
- Para cada ficheiro, é chamada a função *readFiles()*, que povoará o vetor com os respetivos pontos, efetuando corretamente a divisão das coordenadas, atribuindo as mesmas às respetivas componentes (**X**, **Y**, **Z**) na estrutura *Ponto*. Estrutura essa definida por:

```
struct Ponto { float x,y,z; };
```

- Por fim, a função *desenhaPontos()* será chamada para iterar sobre o vetor de pontos sendo que, tal como o nome indica, será responsável por desenhar as várias primitivas;

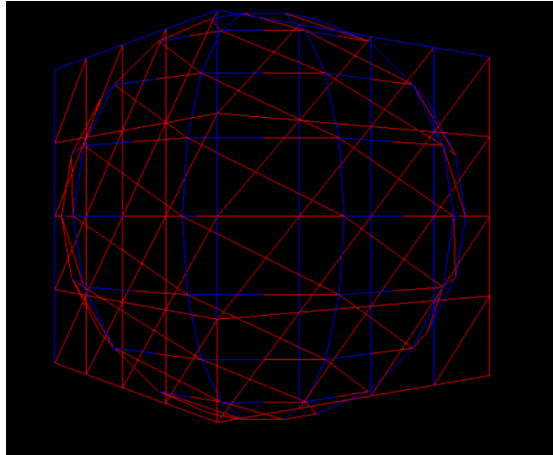
O nosso motor possui ainda algumas funcionalidades leccionadas nas aulas práticas, nomeadamente um conjunto de menus, de modo a aumentar a interação com o utilizador. Vejamos, por exemplo, os controlos referentes à câmara do observador:

w	cima
s	baixo
a	esquerda
d	direita
+	ZOOM out
-	ZOOM in

Primitivas.xml

```
<scene>  
  <model file="esfera.3d"/>  
  <model file="cubo.3d"/>  
</scene>
```

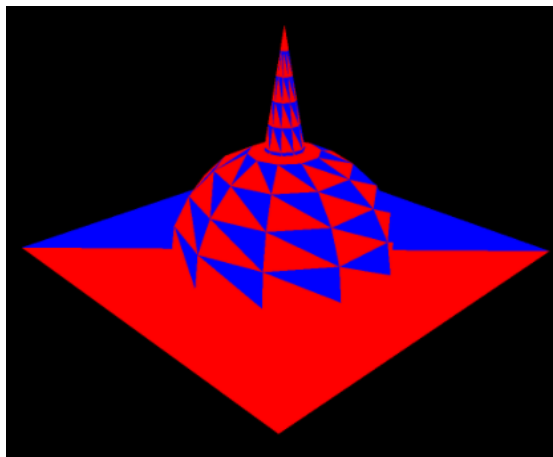
Output



Primitivas.xml

```
<scene>  
  <model file="plano.3d"/>  
  <model file="cone.3d"/>  
  <model file="esfera.3d"/>  
</scene>
```

Output



4 Conclusão

Quanto à parte pedagógica, concluímos que este projeto foi muito enriquecedor para o nosso coletivo, pois este trabalho permitiu que melhorássemos as nossas competências a nível prático relacionadas com a unidade curricular de Computação Gráfica.

O grupo considera bastante produtiva a exploração das ferramentas da disciplina, como o *OpenGL*, bem como a “manipulação” e cimentação de conceitos base como, por exemplo, construir um cubo à base de simples triângulos.

Um dos grandes desafios foi, sem dúvida, aprofundar a linguagem C++ à medida que íamos avançando no projeto, pois as coisas começaram a tornar-se mais complexas e, por isso, exigiu outro nível de conhecimento.

Em suma, o esforço foi grande com o intuito de garantir boas soluções para o enunciado proposto deixando, assim, uma janela aberta e curiosa para as próximas fases.