Universidade do Minho
Escola de Engenharia

Computação Gráfica

Licenciatura em Engenharia Informática

Ano Letivo de 2023/2024

Trabalho Prático

Parte 1- Primitivas Gráficas

Grupo 33

Diogo Gabriel Lopes Miranda (a100839)

Fábio Daniel Rodrigues Leite (a100902)

João Ricardo Ribeiro Rodrigues (a100598)

Sandra Fabiana Pires Cerqueira (a100681)

Março, 2024

C G

Resumo

Este projeto surgiu no âmbito da unidade curricular de Computação Gráfica. Ao longo deste relatório faremos uma pequena introdução e contextualização do problema proposto para esta primeira fase. E de seguida, iremos então explicar o raciocínio utilizado para a resolução do mesmo, bem como expor e discutir os resultados obtidos.

Palavras-chave: Computação Gráfica, GLUT, pontos, vetores, XML, figura geométrica, Generator, Engine, Open GL

Índice

Lista de Figuras

Lista de Tabelas

1. Introdução

1.1 Contextualização

Este relatório surge no âmbito da unidade curricular Computação Gráfica, do 2º semestre, do 3º ano da Licenciatura em Engenharia Informática, da Universidade do Minho. Nele iremos descrever e explicar todo o processo de resolução do problema proposto para esta primeira fase do projeto, que consistiu na criação de duas aplicações: o Generator, que gera arquivos com as informações dos modelos a desenvolver, nesta fase os arquivos contém apenas os vértices para primitivas gráficas das figuras a desenvolver, e o Engine que é responsável por ler um ficheiro de configuração, em XML, e desenhar os vértices das primitivas gráficas geradas pelo Generator.

Para a implementação destas aplicações, foi utilizada a linguagem de programação C++ e utilizada a ferramenta OpenGL.

1.2 Descrição do trabalho proposto

Tal como descrito anteriormente, nesta fase do projeto, temos de desenvolver duas aplicações: um gerador de vértices de primitivas e um motor gráfico.

O gerador de vértices de primitivas terá de ser capaz de gerar as seguintes primitivas, de acordo com argumentos fornecidos específicos das mesmas:

Primitiva	Informações
Plano	Consiste num quadrado no plano XZ, centrado na origem, subdividido nas direções X e Z.
Caixa	Requer dimensão e o número de divisões por aresta, centralizado na origem.
Cone	Requer raio da base, altura, *slices* e *stacks,* a base tem de estar no plano XZ.
Esfera	Requer raio, *slices, stacks* e tem de estar centrada na origem.

Tabela 1- Informações acerca das primitivas a desenvolver

A geração destes vértices, deverá ter como destino um ficheiro .3d, com um formato à escolha. Após gerados, os .3d, e devidamente referenciados num ficheiro XML o Engine terá de ser capaz de dar *display* dos pontos neles contidos.

1.3 Resumo do trabalho a desenvolver

Para o gerador dos vértices das primitivas, o Generator, temos de desenvolver uma função para cada primitiva, em que dados os agrumentos das mesmas, gere os vértices que serão posteriormente interpretados como triângulos, ou seja, cada sequencia de 3 pontos representará um triângulo, seguindo sempre a regra da mão direita.

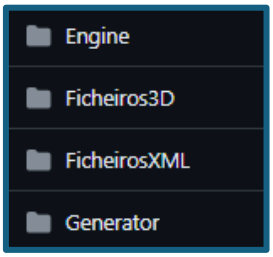
Para guardar os pontos gerados, terá de ser definido, um formato para o ficheiro .3d, que facilite a sua interpretação.

Após tudo isto, o Engine, que é o nosso motor gráfico, terá de ser capaz de interpretar um ficheiro XML, utilizando o OpenGL para da display destes triângulos produzidos pelo generator, e interpretar devidamente os vértices guardados nos .3d. Devendo ainda, armazenar a informação dos modelos importados em memória recorrendo a estruturas de dados, de forma a que os ficheiros XML apenas sejam lidos uma única vez, quando o Engine inicia.

2. Resolução do problema

2.1 Estrutura do Projeto

Tendo em mente tudo aquilo que tínhamos de desenvolver, como descrito da secção 1 deste relatório, o grupo optou por estruturar o projeto da seguinte forma:



- *Figura 1- Estrutura do projeto*

- **Generator:** contém todo o código necessário ao cálculo das coordenadas para a representação das diferentes primitivas (plano, caixa, cone e esfera), bem como a criação e escrita dos pontos nos ficheiros .3d para que possam ser usados no Engine;
- **Engine:** Contém todo o código necessário para que seja possível ver as primitivas;
- **Ficheiros3D:** Onde são guardados os ficheiros .3d gerados pelo Generator
- **FicheirosXML**: Onde estão armazenados os ficheiros XML com as configurações que o Engine tem de ler para dar *display,* bem como o resultado suposto dos mesmos para efeitos de testes.

2.2 Generator

O Generator, foi desenvolvido em C++, e é a aplicação responsável por calcular os pontos das primitivas que são guardados num ficheiro .3d.

Para utilizar o Generator temos de escrever os seguintes comandos, passando ao segundo os argumentos certos:

$ g++ Generator.cpp -o generator

$. /generator [primitiva] [argumentos] [nome do ficheiro .3d]

Onde a primitiva e os argumentos podem ser:

Primitiva	**Argumentos**	**Exemplo de comando**
plane	[Comprimento Divisões]	generator plane 1 3 plane.3d
box	[Comprimento Divisões]	generator box 2 3 box.3d
cone	[Raio Altura Slices Stacks]	generator cone 1 2 4 3 cone.3d
sphere	[Raio Slices Stacks]	generator sphere 1 10 10 sphere.3d

Tabela 2- Estrutura dos comandos e exemplos

Por exemplo, quando executado o comando `` `$ ./generator cone 1 2 4 3 cone.3d` `` , um ficheiro cone.3d seria guardado na nossa pasta *"Ficheiros3D"* , e este ficheiro teria todos os pontos de um cone com raio 1, altura 2, 4 *slices* e *3 stacks*, para futuramente ser usado no Engine.

Estrutura dos ficheiros .3d

Os ficheiros .3d gerados pelo Generator seguem todos uma estrutura específica para facilitar a interpretação dos mesmos. Como se pode observar na figura abaixo, cada linha é composta por coordenadas tridimensionais que representam os 3 vértices de um triângulo. As coordenadas de cada ponto seguem a ordem (x, y, z), onde x, y e z representam as coordenadas nos eixos tridimensionais. Esses pontos são separados por vírgulas, e os vértices estão separados com um ponto e vírgula.

```
-1, 0, -1; -0.333333, 0, -0.333333; -0.333333, 0, -1
-1, 0, -1; -1, 0, -0.333333; -0.333333, 0, -0.333333
-0.333333, 0, -1; 0.333333, 0, -0.333333; 0.333333, 0, -1
-0.333333, 0, -1; -0.333333, 0, -0.333333; 0.333333, 0, -0.333333
0.333333, 0, -1; 1, 0, -0.333333; 1, 0, -1
0.333333, 0, -1; 0.333333, 0, -0.333333; 1, 0, -0.333333
-1, 0, -0.333333; -0.333333, 0, 0.333333; -0.333333, 0, -0.333333
-1, 0, -0.333333; -1, 0, 0.333333; -0.333333, 0, 0.333333
-0.333333, 0, -0.333333; 0.333333, 0, 0.333333; 0.333333, 0, -0.333333
-0.333333, 0, -0.333333; -0.333333, 0, 0.333333; 0.333333, 0, 0.333333
0.333333, 0, -0.333333; 1, 0, 0.333333; 1, 0, -0.333333
0.333333, 0, -0.333333; 0.333333, 0, 0.333333; 1, 0, 0.333333
-1, 0, 0.333333; -0.333333, 0, 1; -0.333333, 0, 0.333333
-1, 0, 0.333333; -1, 0, 1; -0.333333, 0, 1
-0.333333, 0, 0.333333; 0.333333, 0, 1; 0.333333, 0, 0.333333
-0.333333, 0, 0.333333; -0.333333, 0, 1; 0.333333, 0, 1
0.333333, 0, 0.333333; 1, 0, 1; 1, 0, 0.333333
0.333333, 0, 0.333333; 0.333333, 0, 1; 1, 0, 1
```

Figura 2-Formato ficheiros .3d

Por exemplo, na figura, a linha ` -1, 0, -1; -0.333333, 0, -0.333333; -0.333333, 0, -1 `, representa três pontos com as seguintes coordenadas (-1, 0, -1), (-0.333333, 0, -0.333333) e (-0.333333, 0, -1) formam um triângulo, tal como ilustra a seguinte imagem:

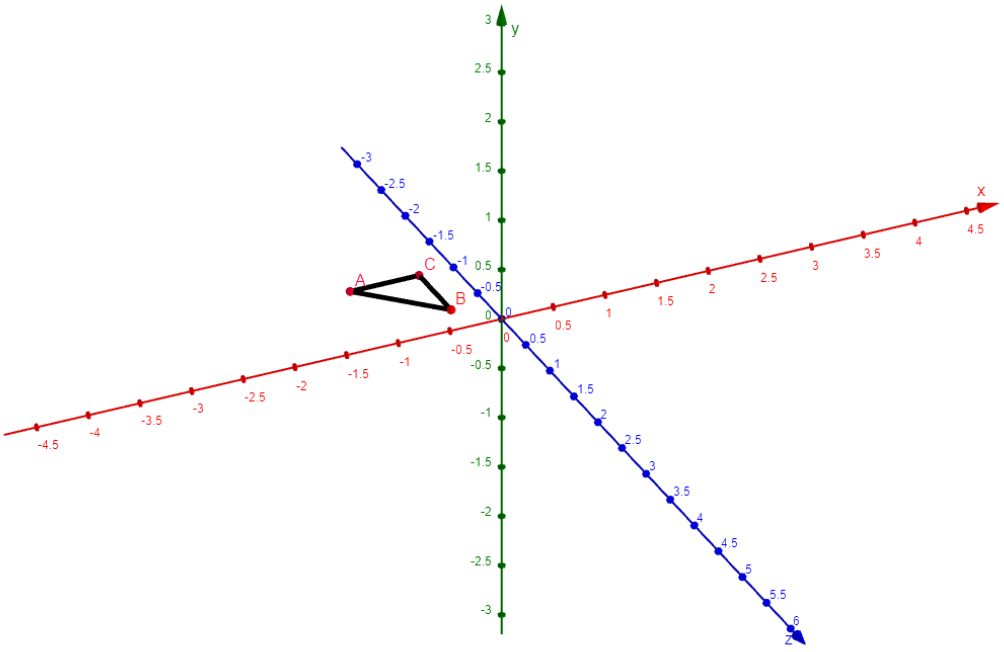


Figura 3-Representação gráfica de uma linha do .3d

2.2.1 Plano

O plano a ser gerado pelo *Generator* deve estar contido no plano XZ e centrado na origem, ponto (0, 0, 0).

Para gerar um plano devem ser fornecidas duas informações cruciais, o seu comprimento e o número de divisões. O comprimento é o tamanho de cada uma das suas arestas. O número de divisões indica em quantas secções deve ser dividida cada aresta.

O plano será constituído pela junção de vários triângulos, sendo que o número destes varia consoante as informações fornecidas à função que o gera. Como referido anteriormente, o plano é dividido em secções, que são constituídas por dois triângulos, formando um quadrado, como podemos visualizar na seguinte figura:

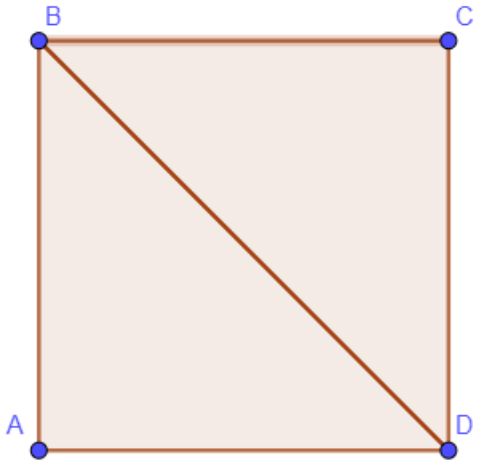


Figura 4 - Ilustração de uma secção

As coordenadas dos pontos utilizados para a primeira secção são dadas pelas seguintes formulas:

$spacing = length \,/\, divison$ → valor das arestas de cada secção (Ex: aresta BC)

A = (-length / 2,-length / 2 + spacing)

B = (-length / 2,-length / 2)

$\mathrm{C} = (-length \,/\, 2 + spacing, -length \,/\, 2)$

$\mathrm{D} = (-length \,/\, 2 + spacing\,, -length \,/\, 2 + spacing)$

De forma a que o plano seja visível quando a câmara se encontra com y>0 (visto de cima), a ordem pela quais os pontos são utilizados para desenhar os triângulos é:

Triângulo 1 – B, D, C

Triângulo 2 – B, A, D

As coordenadas dos pontos da secção seguinte são calculadas deslocando os pontos para a direita, somando à sua coordenada *x* o valor do *spacing*. No caso de já se ter atingido o limite horizontal de secções, é dado novamente os valores iniciais da coordenada *x* e somado o valor do *spacing* à coordenada z, deslocando para a linha abaixo. A figura abaixo demonstra a ordem aplicada na construção.

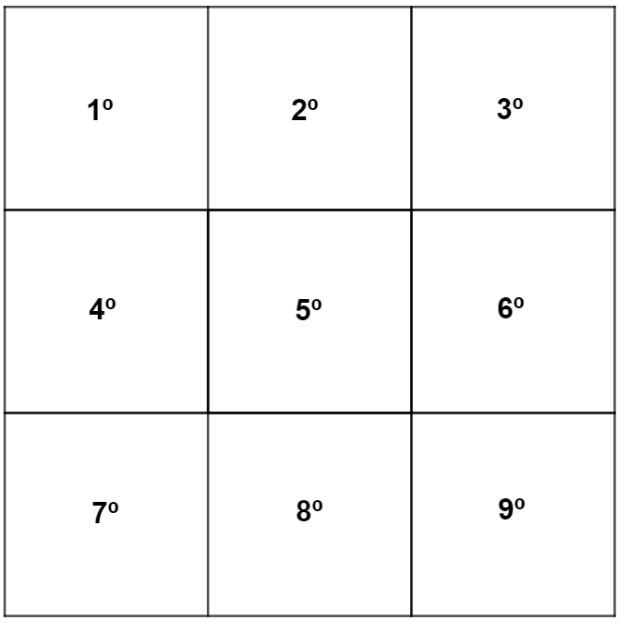


Figura 5 - Ordem de construção do plano

Na figura seguinte encontra-se um exemplo de um plano com *length=1* e *divisions=3*, gerado a partir do seu ficheiro .3d:

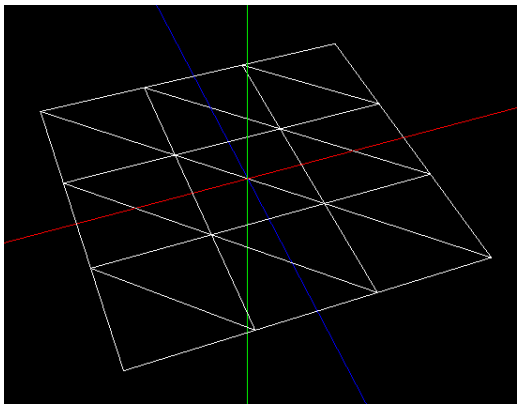


Figura 6 - Exemplo de um plano

De forma a reutilizar a função de gerar o plano para a aplicar à função geradora da caixa, a função de gerar o plano está também preparada para aplicar translações aos pontos e gerar o plano virado para baixo (comportamento padrão é gerar virado para cima). Estas funcionalidades adicionais serão exploradas em detalhe na descrição da função geradora da caixa.

2.2.2 Caixa

Para o cálculo dos pontos da caixa foram utilizadas duas funções auxiliares e a função do plano. As funções auxiliares criadas foram as funções para gerar um plano XY e um plano YZ (raciocínio é o mesmo, mas muda a coordenada igual a zero). A função da caixa recebe dois valores, *dimension* e *divisions*. Em termos práticos têm o mesmo significa que os valores utilizados para os planos.

```
void generatePlaneXZ(float length, int divisions, std::ofstream& outFile, float tx, float ty, float tz, int side)

void generatePlaneXY(float length, int divisions, std::ofstream& outFile, float tx, float ty, float tz, int side)

void generatePlaneYZ(float length, int divisions, std::ofstream& outFile, float tx, float ty, float tz, int side)
```

Figura 7 - Declarações das funções dos planos

Como referido anteriormente no Plano, a função que gera o plano XZ (o mesmo se aplica para as funções dos planos XY e YZ) recebe também valores de translação e um valor que indica se se pretende que a figura esteja voltada para cima ou para baixo.

Os valores de translação (tx, ty, tz) são utilizados para somar às coordenadas obtidas com o cálculo normal, como ilustra a seguinte imagem:

```
float x0 = -length / 2.0f + j * spacing + tx;
float y0 = 0 + ty;
float z0 = -length / 2.0f + i * spacing + tz;
```

Figura 8 - Exemplo da aplicação das translações

A variável *side* é a variável que representa o lado para o qual pretendemos que a figura esteja virada. Dependendo do plano, pode ser cima ou baixo (plano XZ), ou esquerda ou direita (plano XY e YZ). Consoante o seu valor, as coordenadas são calculadas de formas diferentes de forma a alterar o lado para o qual os triângulos estão virados e, por consequente, o plano.

Como se pode visualizar na figura abaixo as variáveis que não possuem o valor 0 foram trocadas, passando, neste caso, o cálculo utilizado para a variável x a ser utilizado para a variável z e vice-versa. Esta mudança é a responsável por a ordem dos pontos utilizados nos triângulos ser trocada, alterando o lado para o qual estão voltados.

```
if (side == 1) {
    for (int i = 0; i < divisions; ++i) {
        for (int j = 0; j < divisions; ++j) {
            // Vértices do primeiro triângulo
            float x0 = -length / 2.0f + j * spacing + tx;
            float y0 = 0 + ty;
            float z0 = -length / 2.0f + i * spacing + tz;
            float x1 = -length / 2.0f + (j + 1) * spacing + tx;
            float y1 = 0 + ty;
            float z1 = -length / 2.0f + (i + 1) * spacing + tz;
            float x2 = -length / 2.0f + (j + 1) * spacing + tx;
            float y2 = 0 + ty;
            float z2 = -length / 2.0f + i * spacing + tz;
```

```
else {
    for (int i = 0; i < divisions; ++i) {
        for (int j = 0; j < divisions; ++j) {
            // Vértices do primeiro triângulo
            float z0 = -length / 2.0f + j * spacing + tz;
            float y0 = 0 + ty;
            float x0 = -length / 2.0f + i * spacing + tx;
            float z1 = -length / 2.0f + (j + 1) * spacing + tz;
            float y1 = 0 + ty;
            float x1 = -length / 2.0f + (i + 1) * spacing + tx;
            float z2 = -length / 2.0f + (j + 1) * spacing + tz;
            float y2 = 0 + ty;
            float x2 = -length / 2.0f + i * spacing + tx;

            writeVertex(outFile, x0, y0, z0, x1, y1, z1, x2, y2, z2);
```

Figura 9 - Excerto de código caso side=1 e side=0, respetivamente

Como pretendemos que o centro do cubo esteja centrado na origem, o valor utilizado nas translações é $length / 2$. O resultado da escolha deste valor é visível na figura abaixo.

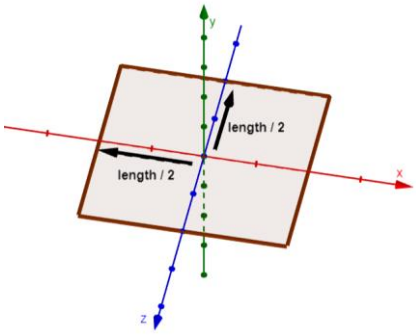


Figura 10 - Demonstração do efeito da translação

```
generatePlaneXY(length, divisions, outFile, tx, ty, tz - length / 2.0f, 0); // Back face
generatePlaneXY(length, divisions, outFile, tx, ty, tz + length / 2.0f, 1); // Front face
generatePlaneXZ(length, divisions, outFile, tx, ty - length / 2.0f, tz, 0); // Bottom face
generatePlaneXZ(length, divisions, outFile, tx, ty + length / 2.0f, tz, 1); // Top face
generatePlaneYZ(length, divisions, outFile, tx + length / 2.0f, ty - length / 2.0f, tz, 1); // Right face
generatePlaneYZ(length, divisions, outFile, tx - length / 2.0f, ty - length / 2.0f, tz, 0); // Left face
```

Figura 11 - Funções utilizadas para gerar a caixa

A figura acima mostra as diferentes translações utilizadas para cada plano de forma a os colocar no local pretendido para cada face. Como é possível visualizar, o último valor passado a cada uma das funções, o valor da variável *size*, alterna entre 0 e 1 nas diferentes utilizações da mesma função, para garantir a correta visualização das faces.

Na figura seguinte encontra-se um exemplo de uma caixa com *dimension=1* e *divisions=3*, gerado a partir do seu ficheiro .3d:

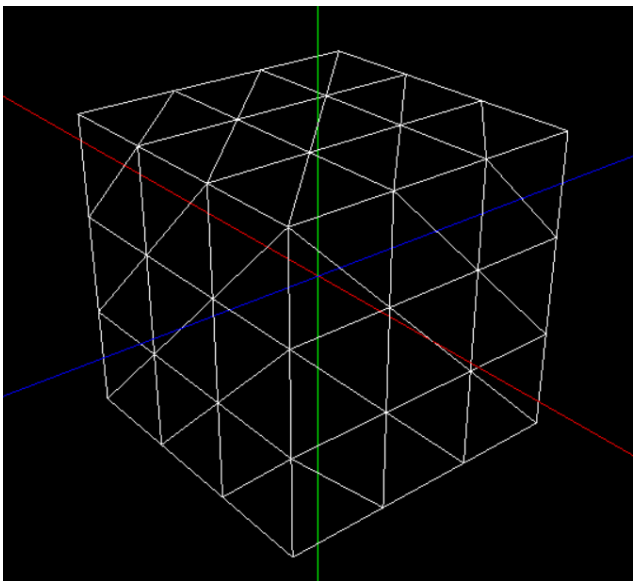


Figura 12 - Exemplo de uma caixa

2.2.3 Cone

Raciocínio inicial

Para ser capaz de gerar os pontos do cone, cuja base está no plano XZ, começámos por analisar bem os parâmetros que o mesmo recebia, que como já referido anteriormente são: o raio da base, a altura, o número de *slices* e o número de *stacks*.

Ou seja, a forma que queremos gerar terá de ser algo do tipo:

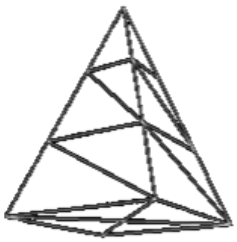


Figura 13-Imagem ilustrativa de um cone

Sendo que o número de divisões verticais e horizontais varia de acordo com as *slices* e *stacks* fornecidas como parâmetro.

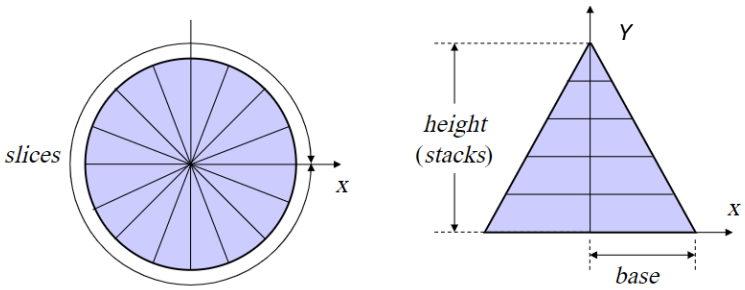
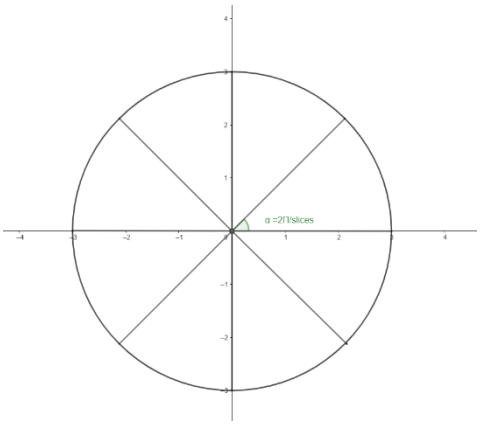


Figura 14-Representação visual das slices e das stacks

Base

Como a base do cone tem de estar no plano XZ, sabemos de antemão que o valor da coordenada y será sempre 0 em todos os vértices. E sabemos ainda que a base é centrada na origem, logo concluímos que um dos vértices dos triângulos que irão constituir a base, será a própria origem (0,0,0).

Uma vez que o número de *slices* é dado, conseguimos calcular qual o angulo (α) entre dois vértices consecutivos da seguinte forma:



$$\alpha = \frac{2\Pi}{slices}$$

Figura 15-Base Cone

Tendo então o ângulo (α) entre cada vértice e o raio (r) da circunferência, recorremos ao uso das coordenas polares, para conseguirmos chegar às coordenas de cada vértice:

$$x = r * sen(\alpha)$$

$$y = r * cos(\alpha)$$

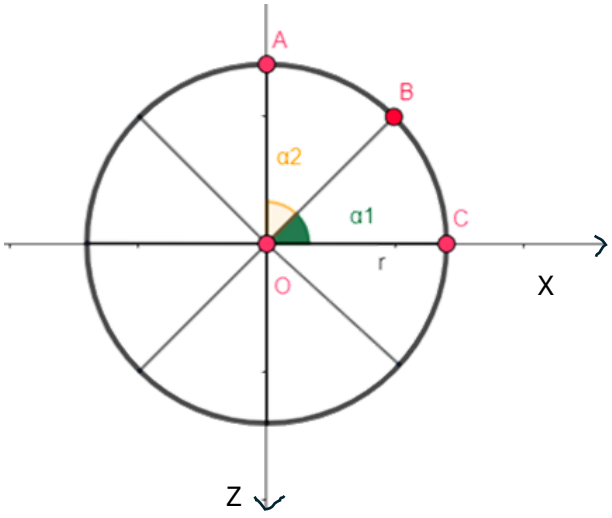


Figura 16-Base Cone

Para definir um triângulo da base do cilindro, como o AOB da Fig.6, ou qualquer outro, recorremos à seguinte sequência de vértices:

Ponto O= (0,0,0)

PontoA= (r * sen ($\alpha 2$),0, r * cos ($\alpha 2$))

PontoB= (r * sen ($\alpha 1$),0, r * cos ($\alpha 1$))

Parte do meio do cone

Após percebermos, como chegaríamos matematicamente aos pontos da base, passámos para o meio do cone, optando por deixar o topo isolado para não haver problemas com sobreposição de pontos.

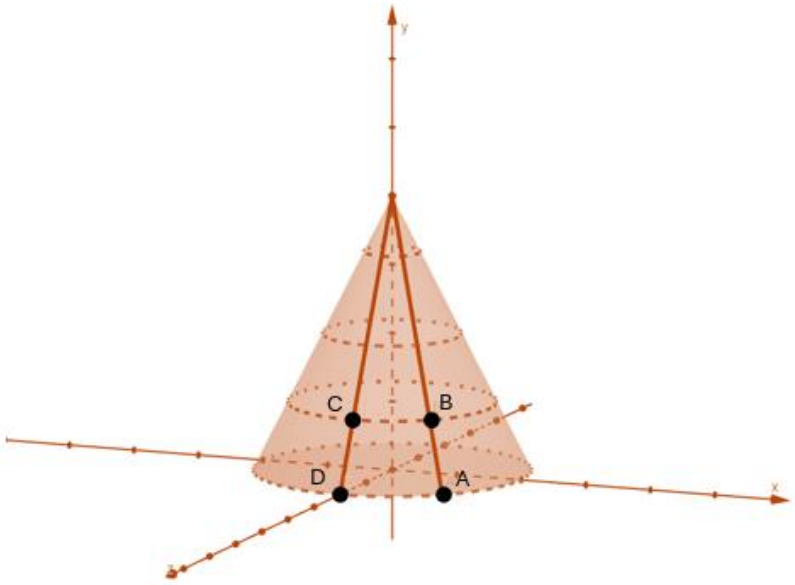


Figura 17- Cone

A nossa abordagem para desenhar esta parte do cone foi inicialmente desenhar uma sequência de triângulos.

Para cada triangulo, as coordenadas dos vértices que se encontram no plano XOZ seriam as mesmas que foram definidas na base, respeitando desta forma o número de slices que o cone terá.

No entanto, cada triangulo é limitado pelo número de stacks, e como as stacks são cortes horizontais, cada triangulo grande passou a ser constituído por vários quadriláteros, como o ABCD da figura acima.

Então desenhamos uma sequência de triângulos em cada stack (dois por cada quadrilátero).

Para calcular a altura de cada de cada *stack* utilizamos a seguinte forma:

$$stackHeight = \frac{height}{stacks}$$

Nota: `height` corresponde à a altura do cone e as `stacks` ao número de stacks pretendidas.

Para além disto, como podemos observar na figura abaixo, de stack para stack o valor do raio varia, raio esse que é necessário para ir calculando os vértices. Para calcular então o novo raio utilizamos a seguinte fórmula:

$$radius2 = radius2 - \left(\frac{radius}{stacks}\right)$$

Nota: `radius´ é o raio do cone e radius2 é o valor do novo raio da stack em que estamos

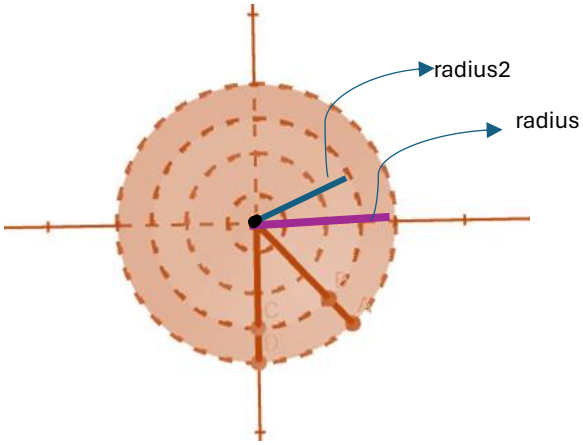


Figura 18-Cone visto de cima

Os nossos triângulos, que formam o quadrilátero, envolvem então informações de duas *stacks*, os triângulos são criados da seguinte forma, respeitando a regra da mão direita:

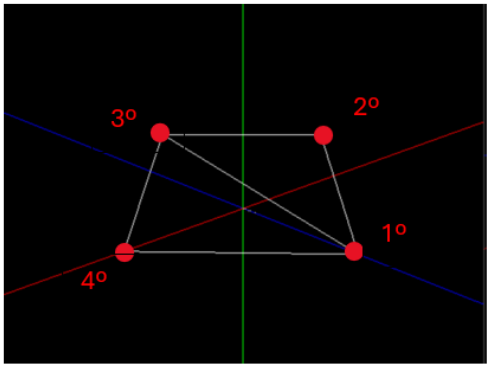


Figura 19-Ordem de criação dos pontos

As coordenadas dos vértices do triângulo 1º 2º 3º são obtidas da seguinte forma:

```
radius * std::sin(alpha2), 0, radius * std::cos(alpha2),
   radius2 * std::sin(alpha2), topHeight, radius2 * std::cos(alpha2),
   radius2 * std::sin(alpha1), topHeight, radius2 * std::cos(alpha1));
```

Figura 20-Obtenção das coordenadas do triangulo 123

As coordenadas dos vértices do triângulo 3º 4º 1º são obtidas da seguinte forma:

```
radius2 * std::sin(alpha1), topHeight, radius2 * std::cos(alpha1),
  radius * std::sin(alpha1), 0, radius * std::cos(alpha1),
  radius * std::sin(alpha2), 0, radius * std::cos(alpha2));
```

Figura 21-Obtenção daas coordenadas do triangulo 341

Este raciocínio é então aplicado nas restantes *stacks:*

```
prevRadius * std::sin(alpha2), bottomHeight, prevRadius * std::cos(alpha2),
   radius2 * std::sin(alpha2), topHeight, radius2 * std::cos(alpha2),
   radius2 * std::sin(alpha1), topHeight, radius2 * std::cos(alpha1));


radius2 * std::sin(alpha1), topHeight, radius2 * std::cos(alpha1),
   prevRadius * std::sin(alpha1), bottomHeight, prevRadius * std::cos(alpha1),
   prevRadius * std::sin(alpha2), bottomHeight, prevRadius * std::cos(alpha2));
```

Figura 22- Obtenção das coordenadas restantes stacks

Topo do cone

Para obter as coordenadas dos vértices dos triângulos da última *stack*, sabemos de antemão que um deles será sempre (0, topHeigth,0), sendo que neste caso a *topHeigth* tem valor igual à altura do cone. E os restantes vértices são iguais aos da *stack* inferior. Sendo, portanto, calculados da seguinte forma:

```
prevRadius * std::sin(alpha2), bottomHeight, prevRadius * std::cos(alpha2),
   0, topHeight, 0,
   prevRadius * std::sin(alpha1), bottomHeight, prevRadius * std::cos(alpha1));
```

Figura 23-Topo do cone

Todo este raciocínio foi então aplicado na definição da criação da função *drawCone*, do *Generator.cpp*, que calcula e escreve num ficheiro .3d os pontos do cone.

2.2.4 Esfera

Para o cálculo dos pontos da esfera, à semelhança do cone, recorremos à utilização de coordenadas esféricas (ϕ, θ, raio), que são traduzidas para coordenadas cartesianas através das seguintes fórmulas:

$$x = raio * \sin\phi * \cos\theta$$

$$y = raio * -cos\ \phi$$

$$z = raio * sin\phi * \sin\theta$$

θ controla a inclinação azimutal da esfera e varia entre 0 e 2π, sendo a sua variação calculada da seguinte:

$$\Delta\,\theta\ = \frac{2\pi}{slices}$$

Da mesma forma, ϕ controla a inclinação polar da esfera e varia entre 0 e π, sendo a sua variação, calculada da seguinte forma:

$$\Delta\,\phi\ = \frac{\pi}{stacks}$$

De forma a conseguirmos calcular os pontos que constituem uma esfera, necessitamos dos seguintes argumentos: raio, slices (camadas verticais) e stacks (camadas horizontais).

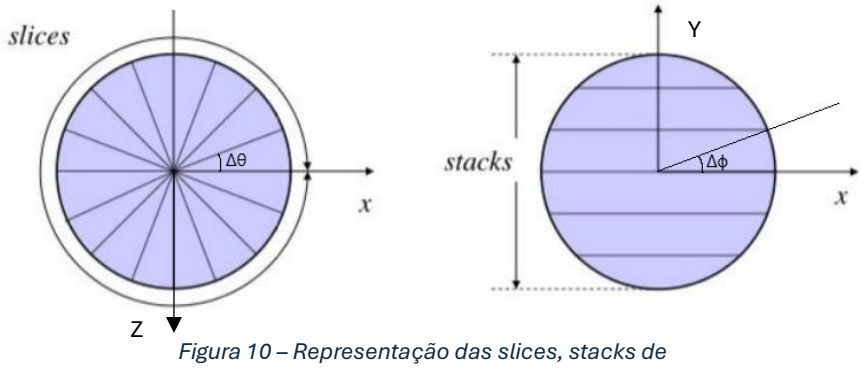


Figura 10 – Representação das slices, stacks de uma esfera e da variação de θ e ϕ.

O desenho das diferentes camadas é feito através de loops consecutivos, onde o loop externo, responsável por delimitar as stacks (camadas horizontais), atualiza o valor do ângulo ϕ. Relativamente ao loop interior, o mesmo é responsável por delimitar as slices (camadas verticais), sendo que esse loop atualiza o valor do ângulo θ.

Desta forma, ao longo das iterações vão sendo construídas cada slice dentro de uma stack, antes de passar para a stack seguinte.

O propósito da variável *"deltaThetaAdjusted"* é ajustar o ângulo θ, ajustando consequentemente o desenho dos pontos e a posição da esfera em relação aos eixos x,y e z.

```
for (int i = 0; i < stacks; ++i) { // ciclo que itera sobre as diferentes stacks

    float phi1 = i * deltaPhi;
    float phi2 = (i + 1) * deltaPhi;

    for (int j = 0; j < slices; ++j) { // ciclo que itera sobre as diferentes slices

        float deltaThetaAdjusted = deltaTheta / 2.0f;
        float theta1 = (j * deltaTheta) - deltaThetaAdjusted;
        float theta2 = ((j + 1) * deltaTheta) - deltaThetaAdjusted;

        // Vértices dos triângulos
        float x1 = radius * sin(phi1) * cos(theta1);
        float y1 = -radius * cos(phi1);
        float z1 = radius * sin(phi1) * sin(theta1);

        float x2 = radius * sin(phi1) * cos(theta2);
        float y2 = -radius * cos(phi1);
        float z2 = radius * sin(phi1) * sin(theta2);

        float x3 = radius * sin(phi2) * cos(theta1);
        float y3 = -radius * cos(phi2);
        float z3 = radius * sin(phi2) * sin(theta1);

        float x4 = radius * sin(phi2) * cos(theta2);
        float y4 = -radius * cos(phi2);
        float z4 = radius * sin(phi2) * sin(theta2);
```

Figura 11-Pedaço de código encarregue de iterar sobre as diferentes stacks e slices.

Relativamente à criação dos triângulos, a cada iteração são gerados 4 pontos diferentes, através da variação dos ângulos, sendo que cada par de vértices são consecutivos na mesma fatia (θ) ou em fatias adjacentes (ϕ), sendo usados para formar dois triângulos. São esses triângulos que compõem a malha triangular que representa a esfera (cada par de triângulos é representado na *Figura 12* por um quadrado).

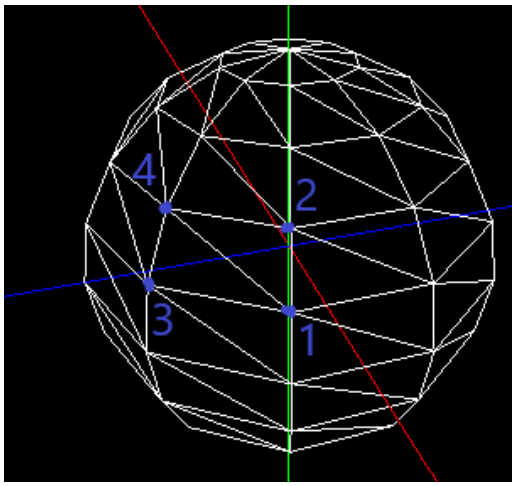


Figura 12 -Representação de 4 vértices gerados, que permitem a construção de dois triângulos.

A esfera é então gerada stack por stack, sendo primeiramente construída a base inferior, seguida das diversas stacks que constituem a parte lateral da mesma, alcançando então a base superior, terminando a construção completa da figura.

Este fenómeno pode ser exemplificado através das seguintes figuras:

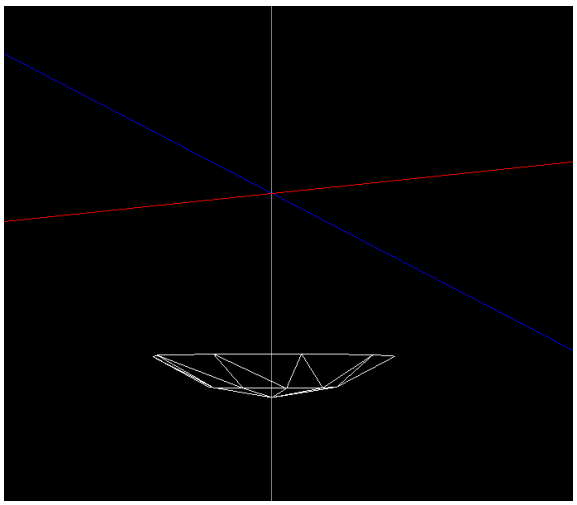


Figura 13 - Desenho da base e da primeira stack lateral.

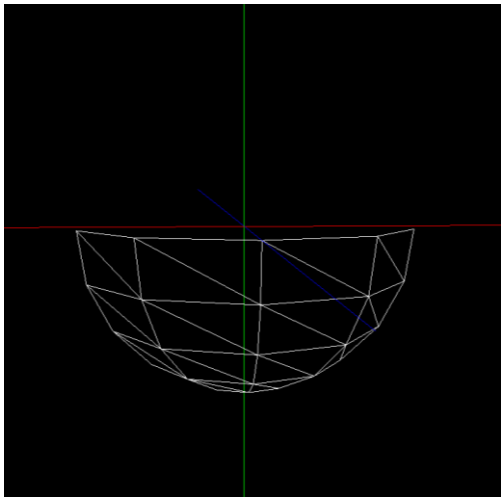


Figura 14 - Desenho de meia-esfera.

2.3 Engine

O *Engine* é, como referido anteriormente, o programa que recebe e lê um ficheiro XML que contém as informações acerca das configurações da posição da câmara e referências aos ficheiros .3D, tal como se pode ver na imagem abaixo:

```
<world>
    <window width="512" height="512" />
    <camera>
        <position x="3" y="2" z="1" />
        <lookAt x="0" y="0" z="0" />
        <up x="0" y="1" z="0" />
        <projection fov="60" near="1" far="1000" />
    </camera>
    <group>
        <models>
            <model file="plane.3d" />
            <model file="cone.3d" />
        </models>
    </group>
</world>
```

Figura 24-Exemplo de ficheiro xml

Para facilitar a leitura do ficheiro XML, recorremos à utilização da biblioteca tinyxml2. Através dela, conseguimos fazer o *parsing* de todas as informações necessárias para configurar e visualizar a figura tridimensional de acordo com as especificações nele contidas.

Foi criada a função *"readxml",* que é responsável por ler o ficheiro xml e guardar as configurações da câmara nele presentes, da seguinte forman:

- **Definições da câmara:**

  -As coordenadas da posição da câmara são guardadas nas variáveis *camX*, *camY* e *camZ*;

  -As coordenadas para onde a câmara está a apontar são guardadas no *lookX*, *lookY* e *lookZ*;

  -As coordenadas do vetor de orientação para cima são armazenadas em *upX*, *upY* e *upZ*.

- **Tamanho da janela de visualização:**

  -Os parâmetros *width* e *height*, utilizados para definir o tamanho da janela de visualização são guardados nas variáveis *windowWidth* e *windowHeight*.

- **Restantes parâmetros:**

  -O *field of view*, o *near* e o *far* são guardados respetivamente nas variáveis *fov*, *near* e *far*.

Para ler os pontos dos ficheiros 3D presentes no XML, desenvolvemos a função "*readPointsFromFile*". Esta função tem a responsabilidade de ler as coordenadas escritas no ficheiro e armazená-las em "*globalPoints*".

```
struct Point3D {
    float x, y, z;
    Point3D(float _x, float _y, float _z) : x(_x), y(_y), z(_z) {}
};

std::vector<Point3D> globalPoints;
```

Figura 25-Estrutura para guardar os pontos

Como apresentado na figura acima, criamos uma *struct* para armazenar as informações de um ponto, a *struct Point3D*. Nesta são guardadas as coordenadas (x, y, z) do ponto. Criamos também uma variável global chamada *globalPoints,* que é um vetor (array dinâmico) constituído por objetos do tipo Point3D. Como referido acima, neste vetor são armazenadas as coordenadas lidas do ficheiro e posteriormente acedido para as obter.

Tendo obtidos as coordenadas dos pontos, passamos agora a desenhar a figura correspondente. A função responsável por essa tarefa é a "*renderScene"*, que utiliza a função "*drawTriangles"* para desenhar os triângulos que compõem a figura. Para tal esta

percorre o vetor *globalPoints,* iterando de 3 em 3 posições, e utilizando os pontos obtidos a cada iteração para desenhar o triangulo respetivo, como apresentado na figura abaixo.

```
void drawTriangles() {
    glBegin(GL_TRIANGLES);
    glColor3f(1.0f, 1.0f, 1.0f);
    for (size_t i = 0; i < globalPoints.size(); i += 3) {
        glVertex3f(globalPoints[i].x, globalPoints[i].y, globalPoints[i].z);
        glVertex3f(globalPoints[i + 1].x, globalPoints[i + 1].y, globalPoints[i + 1].z);
        glVertex3f(globalPoints[i + 2].x, globalPoints[i + 2].y, globalPoints[i + 2].z);
    }
    glEnd();
}
```

Figura 26 - Função "drawTriangles"

Relativamente à câmara, esta pode ser movida utilizando as setas do teclado. Para tal é utilizada a função "*updateCameraPosition",* que é responsável por atualizar a posição da câmara tendo por base os valores *alfa, beta* e *r. Alpha* e *beta* representam os ângulos de rotação horizontal e vertical da câmara, respetivamente, e *r* representa a distância da câmara ao ponto de observação. A função é chamada sempre que é necessário recalcular a posição da câmara, utilizando os valores atualizados das variáveis *alfa, beta* e *r* (resultantes da utilização da função em um momento anterior). A formulas utilizadas para o cálculo das coordenadas da câmara são:

$$camX = r * cos(beta) * sin(alpha)$$

$$camY = r * sin(beta)$$

$$\mathrm{camZ} = \mathrm{r} * \cos(\mathrm{beta}) * \cos(\mathrm{alpha})$$

Para executar o programa e dar display do resultado do teste pretendido, deve ser modificada o campo onde é indicado o ficheiro a utilizar para a função *readXML* para apresentar o ficheiro pretendido. A figura abaixo é um exemplo de executar o teste-_1_5.xml:

```
int main(int argc, char* argv[]) {
    if (argc == 2) {
        readXML(argv[1]);
    }
    else {
        readXML("test_1_5.xml");
    }
    //...
```

Figura 27 - Código necessário para executar o teste "test_1_5.xml"

Alterando esta linha em concreto para executar o teste que pretendemos:

```
readXML("test_1_5.xml");
```

Figura 28 - Linha que deve ser mudada

2.3.1 Outputs do *Engine* e comparação com os Testes

De seguida, iremos apresentar as imagens daquilo que obtivemos *(figuras do lado esquerdo)* após executarmos o *Engine*, colocando nele o nome do XML do qual pretendemos executar o *parsing*. Do lado direito encontram-se as imagens dos testes fornecidos pelos docentes para esta fase do projeto.

Cone (teste_1_2.xml):

Raio=1; Altura=2, Slices=4 e Stacks=4;
Câmara: x="5" y="-2" z="3" e Projection: fov="20" near="1" far="100

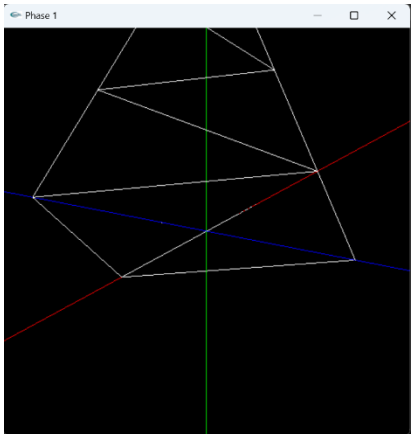


Figura24-Resultado obtido na nossa execução

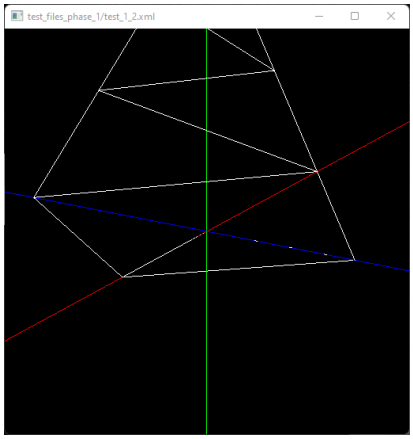


Figura 29-Teste dos professores

Cubo (teste_1_4.xml):

Unidades=2 e Divisão=3;
Câmara: x="3" y="2" z="1" e Projection: fov="60" near="1" far="3.5"

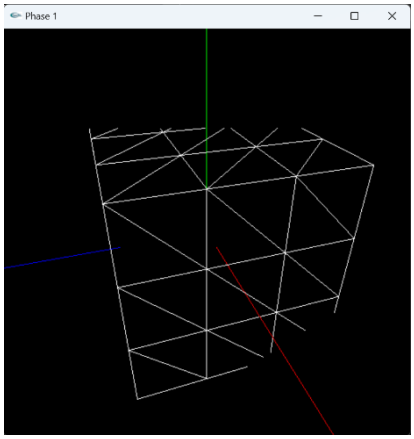


Figura 25-Resultado obtido na nossa execução

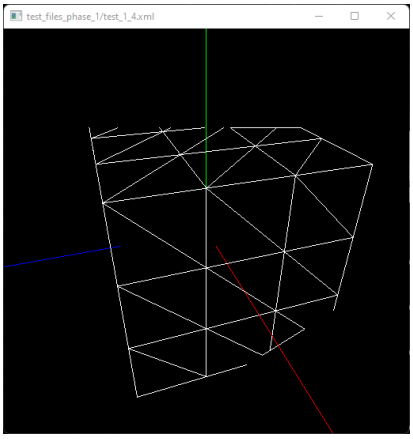


Figura 30-Teste dos professores

Esfera (teste_1_3.xml):

Radius=1, Slices=10 e Stacks=10;
Câmara: x="3" y="2" z="1" e Projection: fov="60" near="1" far="1000"

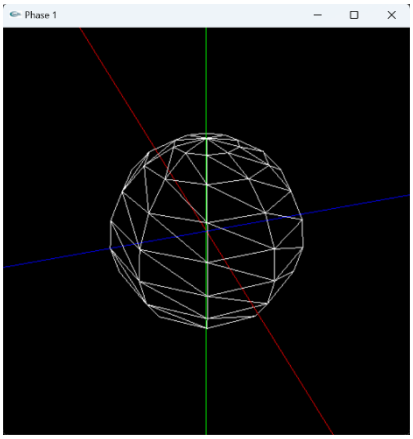


Figura 26-Resultado obtido na nossa execução

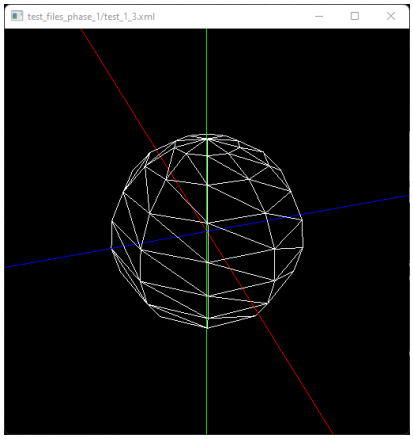


Figura 31-Teste dos professores

Plano (teste_1_5.xml):

Nota: Teste sem as espera;
Lenght=2 e Division=3;
Câmara: x="3" y="2" z="1" e Projection: fov="60" near="1" far="1000"

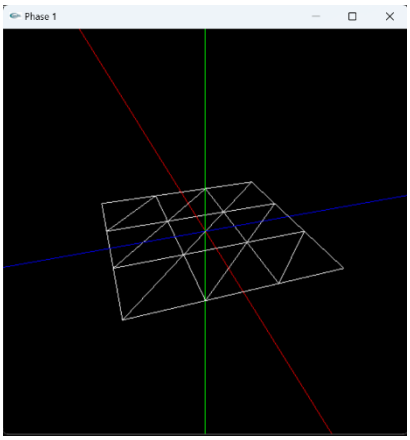


Figura 27-Resultado obtido da nossa execução

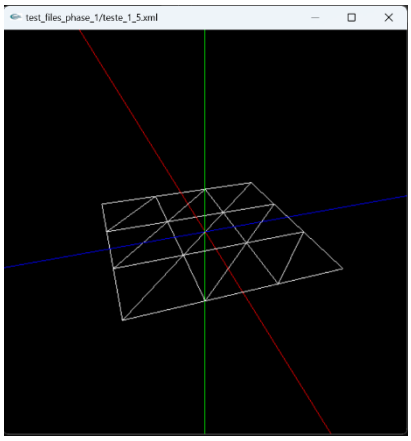


Figura 32-Teste dos professores

Como se pode observar, através da análise das figuras acima apresentadas, os resultados que obtivemos estão todos de acordo com os que foram fornecidos pelos professores, portanto concluímos que o *Generator* e o *Engine* estão definidos como é suposto, logo o trabalho foi executado com sucesso.

3. Extras

3.1 Interação com o sistema e modos de visualização

Com o objetivo de tornar o sistema mais iterativo, e conseguirmos movimentar a câmara ao redor das figuras, definimos as seguintes funções no *Engine*:

```
void processKeys(int key, int xx, int yy) {
    switch (key) {
    case GLUT_KEY_DOWN:
        beta -= 0.1f;
        break;
    case GLUT_KEY_UP:
        beta += 0.1f;
        break;
    case GLUT_KEY_RIGHT:
        alpha += 0.1f;
        break;
    case GLUT_KEY_LEFT:
        alpha -= 0.1f;
        break;
    }
    updateCameraPosition(); // Atualizar a posição da câmara
    glutPostRedisplay();
}
```

Figura 33- Função processKeys

```
void processSpecialKeys(unsigned char key, int x, int y) {
    switch (key) {
    case '+':
        r -= 0.5f;
        break;
    case '-':
        r += 0.5f;
        break;

    case 'f':
        tipo = GL_FILL;
        break;
    case 'l':
        tipo = GL_LINE;
        break;
    case 'p':
        tipo = GL_POINT;
        break;
    }
    updateCameraPosition(); // Atualizar a posição da câmara
    glutPostRedisplay();
}
```

Figura 34-Função processSpecialKeys

Desta forma, conseguimos interagir com o sistema através do teclado da seguinte forma:

+	Zoom in
-	Zoom out
↑	Rotação da câmara para cima
↓	Rotação da câmara para baixo
→	Rotação da câmara para a direita
←	Rotação da câmara para a esquerda
f	Preenche a figura
l	Mostra as linhas da figura
p	Mostra os pontos da figura

Como é possível ver nos comandos acima, nós decidimos implementar a possibilidade de ao pressionar teclas especificas diferentes modos de visualização serem apresentados, tal como se pode ver pelas seguintes figuras:

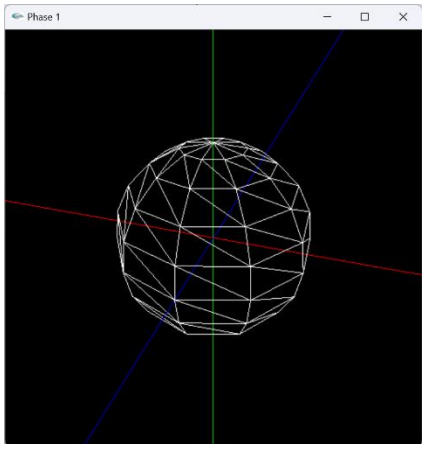


Figura 37- Display de uma primitiva com tecla "l"

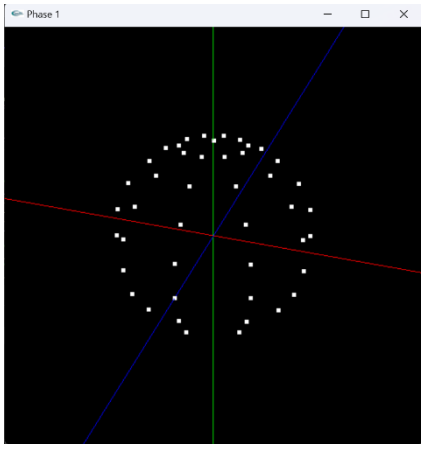


Figura 36- Display de uma primitiva com a tecla "p"

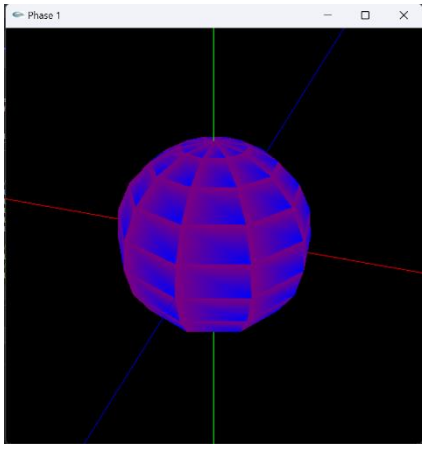


Figura 35-Display de uma primitiva com a tecla "f"

4. Conclusão

Com a realização desta primeira fase do projeto, o grupo foi capaz de por à prova e consolidar os conhecimentos adquiridos ao longo das aulas teóricas e práticas de Computação Gráfica, até à data, assim como adquirir novos conhecimentos, especialmente ao nível do GLUT.

Acreditamos que fomos capazes de atingir todos os objetivos que eram necessários para esta fase, uns com mais facilidade que outros, sentindo dificuldade principalmente na utilização do TinyXml2, para a leitura dos ficheiros XML, uma vez que nenhum dos elementos do grupo era familiarizado com a mesma.

Para além disto, introduzimos ainda alguns extras como a possibilidade de interação no sistema, permitindo a movimentação da câmara, para visualizar as figuras de diversas distâncias e ângulos.

Em suma, o grupo considera que conseguiu nesta fase estabelecer uma base sólida para as restantes fases que se avizinham.