

UNIVERSIDADE DO MINHO  
MESTRADO INTEGRADO EM ENGENHARIA  
INFORMÁTICA

**Trabalho Prático 1**

**Grupo 35**

*André Rafael Olim Martins, A84347*

*Filipe Emanuel Santos Fernandes, A83996*

*José Diogo Xavier Monteiro, A83638*

*Susana Vitória Sá Silva Marques, A84167*

Computação Gráfica  
3º Ano, 2º Semestre  
Departamento de Informática

14 de março de 2021

# Índice

<b>1</b>	<b>Introdução</b>	<b>1</b>
<b>2</b>	<b>Generator</b>	<b>2</b>
2.1	Utilização . . . . .	2
2.2	Plane . . . . .	3
2.3	Box . . . . .	3
2.4	Cone . . . . .	5
2.5	Sphere . . . . .	6
<b>3</b>	<b>Engine</b>	<b>7</b>
3.1	Utilização e Demonstração . . . . .	8
<b>4</b>	<b>Conclusão</b>	<b>15</b>

# 1 Introdução

Nesta UC de Computação Gráfica, foi-nos proposto o desenvolvimento de um projeto dividido em 4 fases que consiste num mecanismo 3D, sendo esta a primeira fase que foca na representação de algumas primitivas gráficas.

Assim, foi decidido que nesta fase se deveria dividir o mecanismo em duas aplicações principais: o *Generator* e o *Engine*. Com estes mecanismos foram criados e representados as primitivas requisitadas: plano, caixa, cone e esfera.

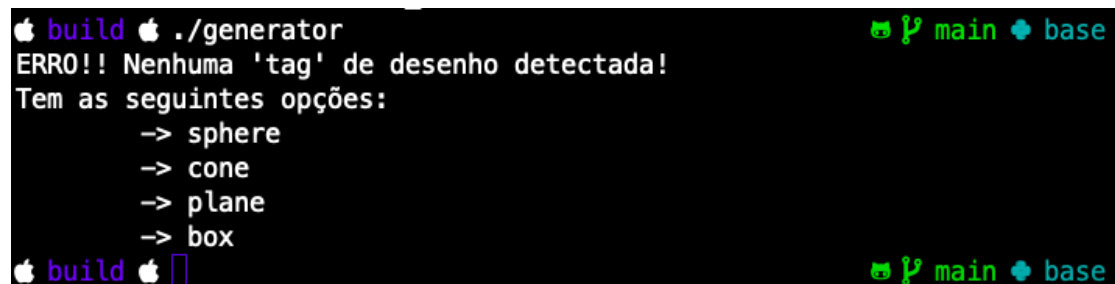
## 2 Generator

Esta aplicação destina-se a gerar vários vértices que constituem as diferentes primitivas gráficas que nos foram requisitadas neste trabalho: plano, caixa, cone e esfera, conforme os parâmetros escolhidos (dimensões, fatias e camadas). Gráficamente, o conjunto de 3 destes vértices correspondem a triângulos, pois estes são a unidade de constituição de todas as primitivas.

A nível de representação dos ficheiros de modelos criados pelo *Generator*, estes apenas contêm em cada linha as coordenadas (x,y,z) de um desses mesmos vértices separadas por espaço. Logo cada conjunto de três linhas descreve um triângulo modelo, pelo que os pontos estão também escritos em *Counter Clock Wise* de acordo com o *standard* de Computação Gráfica, tornando, a seguir, o trabalho do *Engine* mais fácil.

### 2.1 Utilização

Após a criação do executável na diretoria *build* do *Generator* (onde existe já o ficheiro *build.txt* que apenas indica como começar a usar o *Generator*), invocando o comando `./generator` sem nenhum argumento, irá ser apresentado as opções que se poderão usar como primeiro argumento:



```
build ./generator
ERRO!! Nenhuma 'tag' de desenho detectada!
Tem as seguintes opções:
    -> sphere
    -> cone
    -> plane
    -> box
```

Figura 1: Opções de escolha de primitivas

A seguir, se por exemplo, desejarmos criar uma esfera, invocando o comando com apenas 1 argumento: `./generator sphere`, iremos encontrar um menu que nos permite entender quais são os argumentos necessários para a esfera:

```
🍏 build 🍏 ./generator sphere                                🐛 📁 main ➕ base
ERRO!! Número de argumentos errado
Ex: sphere [raio] [camadas] [fatias] [output]
🍏 build 🍏 |
```

Figura 2: Argumentos necessários para a primitiva esfera

Podemos usar este método para qualquer uma das primitivas, para obtermos quais são os argumentos pedidos para criar cada uma delas.

## 2.2 Plane

Para criarmos um plano basta definirmos um quadrado, ou seja, a conjunção de dois triângulos, que neste caso têm de ser desenhados no plano XZ. Estes triângulos têm que ser desenhados numa certa ordem se desejarmos observar a parte de cima (algo comum em Computação Gráfica), ou seja, o desenho dos vértices tem que se dar no sentido oposto ao dos ponteiros do relógio, como já referido anteriormente. Dado que esta primitiva é sempre desenhada da mesma forma, basta apenas definir os dois triângulos mencionados, para observar a vista de cima e desenhar dois triângulos criados de maneira oposta (vértices no sentido contrário) para que ao fazermos uma rotação do plano consigamos ver a vista de baixo também. Ao mesmo tempo, é necessário ter em atenção que foi requisitado que o plano se encontrasse centrado na origem. Desta maneira, as coordenadas de cada ponto correspondem a metade do valor de X e a metade do valor de Z, mantendo a coordenada de Y com o valor 0, por querermos um plano XZ na origem.

## 2.3 Box

O nosso gerador de caixas funciona com base no método de criação de planos, isto porque, uma caixa pode ser considerada como a união de 6 planos distintos. Os planos, tal como vimos previamente, podem ser definidos como a união de pelo menos dois triângulos que tinham sido desenhados sobre o plano XZ, neste caso, teremos que alterar o plano mediante a face que pretendemos gerar.

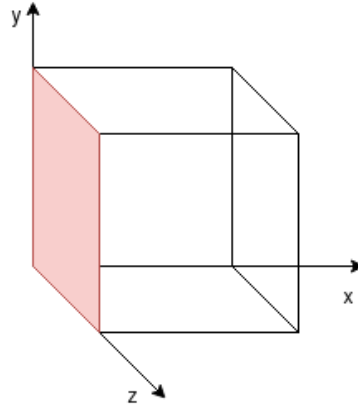


Figura 3: Face  $x=0$

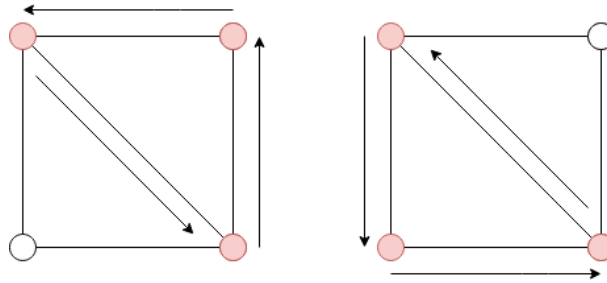


Figura 4: Gerar Plano

Como exemplo do nosso processo de geração, vamos selecionar a face cujo valor  $x$  é 0 (plano XZ). No nosso gerador, decidimos aplicar tanto slices como stacks (divisões verticais e horizontais), mas neste exemplo demonstrativo vamos fazer a geração para quando ambos possuem o valor 1.

Começamos por pré-definir que todos o valor  $x$  dos vértices vão ter o valor 0, porque é esta a face que estamos a representar. Depois, realizamos a repartição em 2 triângulos. Os 3 vértices do triângulo têm que ser escritos no sentido oposto aos ponteiros do relógio (regra da mão direita) para que a normal da figura esteja a "sair" da caixa.

Neste caso simples, o processo é este. Porém, como decidimos aplicar slices e stacks foi introduzida mais complexidade. Assim sendo, continuando a utilizar este exemplo, mas com 5 divisões verticais e 2 divisões horizontais, o processo vai

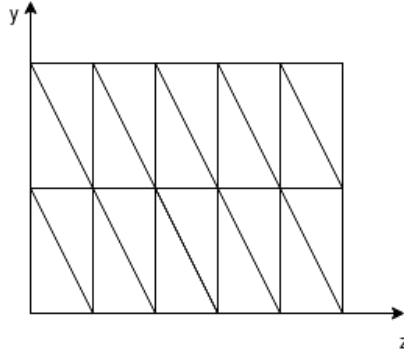


Figura 5: Stacks e Slices no caso da Box

prosseguir de forma semelhante. O único propósito que estes valores vão apresentar será dividir a nossa face em múltiplas sub-faces tanto na vertical, no caso das slices, como na horizontal, no caso das stacks. Neste caso, temos que ter em conta que o tamanho das slices (divisões verticais) da face irá variar mediante o valor de Z e o tamanho das stacks (divisões horizontais) irá variar mediante o valor de Y. Isto, utilizando as seguintes fórmulas:

```
tamanho_divisao_horizontal = y_max / divisao_horizontal;
tamanho_divisao_vertical = z_max / divisao_vertical;
```

Mediante a face, os valores utilizados para a definição dos tamanhos das divisões, tanto horizontais como verticais, irá variar.

## 2.4 Cone

A criação do cone foi pensada em 2 partes: a sua base (circunferência) e o seu lado. As informações pedidas para a construção são: o raio da base, a altura do cone, o número de fatias (verticais) e o número de camadas (fatias horizontais). Como envolve circunferências são usadas coordenadas polares e a sua respectiva conversão para coordenadas cartesianas facilitando assim o processo. A base é feita com triângulos no plano XZ direccionados para o centro da circunferência. O numero de triângulos depende do número de fatias verticais, tendo em conta que quanto maior o seu número, mais "redonda"ficará a circunferência. Para formar o resto do cone, a altura é dividida pelo número de camadas e estas serão formadas



tais como a base, porém o seu raio será menor. Numa fase final são usado triângulos para ligar cada uma destas camadas à sua camada superior e inferior.

## 2.5 Sphere

Para criação da esfera são pedidas três informações: o seu raio, o número de camadas (*stacks*) e o número de fatias (*slices*).

De forma a que as camadas da esfera tenham a mesma altura, é dividido o diâmetro pelo número de camadas desejadas. Assim, a exemplo, a camada do topo tem a altura de raio e a camada a baixo desta tem raio-alturaCamada. Para o cálculo das coordenadas esféricas para a obtenção dos pontos, é usada a seguinte expressão para o cálculo do beta.

```
float getBeta(float height, float radius){  
    return asin(height/radius);  
}
```

A sua construção é realizada de cima para baixo, por camadas. As bases da esfera, topo e base, em relação ao eixo dos Y, têm altura raio e -raio, respetivamente.

Para as faces laterais é calculado o beta da camada atual e da camada a baixo, e depois são usadas as expressões a baixo para o cálculo das coordenadas dos pontos.

```
float esfericaX(float alfa, float beta, float raio){  
    return raio*cos(beta)*sin(alfa);  
}  
  
float esfericaZ(float alfa, float beta, float raio){  
    return raio*cos(beta)*cos(alfa);  
}
```

### 3 Engine

O *Engine* tem como objetivo receber e ler ficheiros escritos em XML (nesses ficheiros apenas se encontra a localização dos ficheiros criados pelo *Generator*), fazendo o *parsing* dos mesmos e carregando-os para memória e interpretando e representando os modelos e o seu conteúdo.

Para a implementação do *Engine* começamos por executar tudo o que aprendemos nas aulas sobre o *Glut* e acrescentamos novos métodos e soluções para o problema em questão.

Para processar o ficheiro XML, usamos como indicado o *TinyXML2* de maneira a explorar o conteúdo destes, extraindo o nome dos ficheiros e as coordenadas dos vértices através de *parsing*.

Tal como o enunciado pede, para guardarmos a informação dos modelos em memória, criamos uma estrutura **Vert** que armazena cada uma das coordenadas x,y,z que identificam um vértice. Para armazenar os diversos vértices que vão ser lidos do ficheiro XML, criamos um vector de *Vert* que nos ajuda a obter uma maior eficiência.

#### Declaração da estrutura de dados:

```
struct Vert {  
    float x;  
    float y;  
    float z;  
};  
  
vector<Vert> vertexes;
```

Passando à fase *render* é apenas uma questão de percorrermos o vector dos *vertexes* desenhando os triângulos correspondentes. O seguinte excerto de código mostra como essa travessia é realizada e como os triângulos são desenhados:

```
glBegin(GL_TRIANGLES);  
for(; i< vertexes.size(); i++) {
```

```

        vertex3f(vertices[i].x, vertices[i].y, vertices[i].z);
    }
    glEnd();

```

### 3.1 Utilização e Demonstração

Para uso do *Engine* com todos os controlos 3D implementados, podemos invocar o comando `./engine help` que nos permitirá visualizar um menu de ajuda, como a seguinte imagem demonstra:

```

build ./engine help main base
-----> Controlos 3D <-----
* Translação: Seta cima, baixo, esquerda, direita
* Rotação: w, a, s, d | W, A, S, D
* Zoom: + | -
* Representação do sólido:
  - por linhas: l | L
  - por pontos: p | P
  - preenchido: f | F
* RESET: r | R
-----><-----
x INT main base

```

É importante referir que para esta aplicação funcionar, terá que existir a criação de um ficheiro XML ( no nosso caso o ficheiro *config.xml* que será passado como primeiro argumento para o *Engine*. Esta criação é feita pelo utilizador, e os ficheiros presentes no mesmo são criados pelo *Generator* e colocados na diretoria *build* do *Engine* (diretoria do executável). No nosso trabalho, temos ainda uma diretoria *models* onde já estão criados alguns modelos com o mesmo nome que no ficheiro XML, para que seja mais fácil testar a aplicação, movendo apenas o modelo que queremos testar para fora dessa diretoria e apenas para dentro do *build* do *Engine*. Assim, após a execução do comando `./engine config.xml` que irá ler e desenhar os modelos, é possível uma iteração com os mesmo utilizando os comandos citados na imagem anterior.

A seguir seguem exemplos gerados de cada uma das primitivas individualmente com o conteúdo do ficheiro *config.xml*:

```
<scene>
  <model file="sphere.3d"/>
  <model file="box.3d"/>
  <model file="box1.3d"/>
  <model file="plane.3d"/>
  <model file="cone.3d"/>
</scene>
```

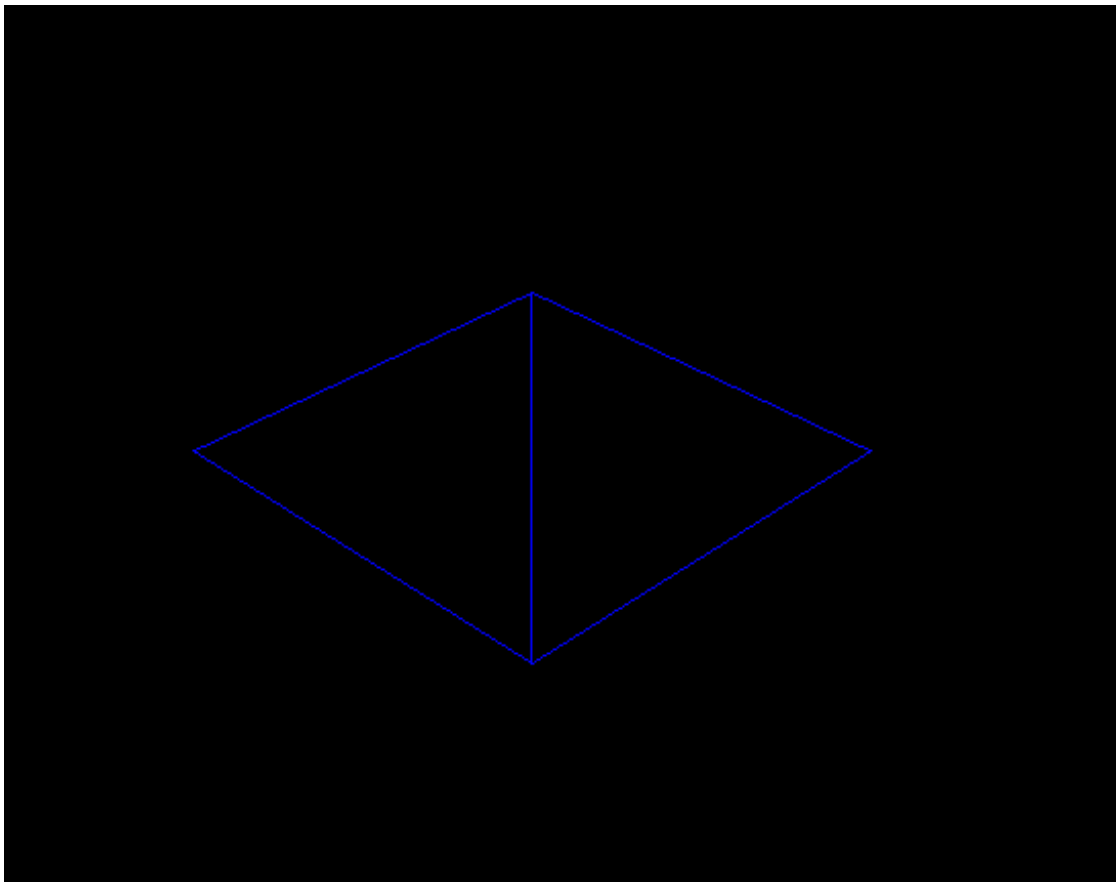


Figura 6: Plano de dimensão 4 (Lado do eixo X=4 e Lado do eixo Z=4)

Comandos utilizados:

```
./generator plane 4 4 plane.3d
```

```
./engine config.xml
```

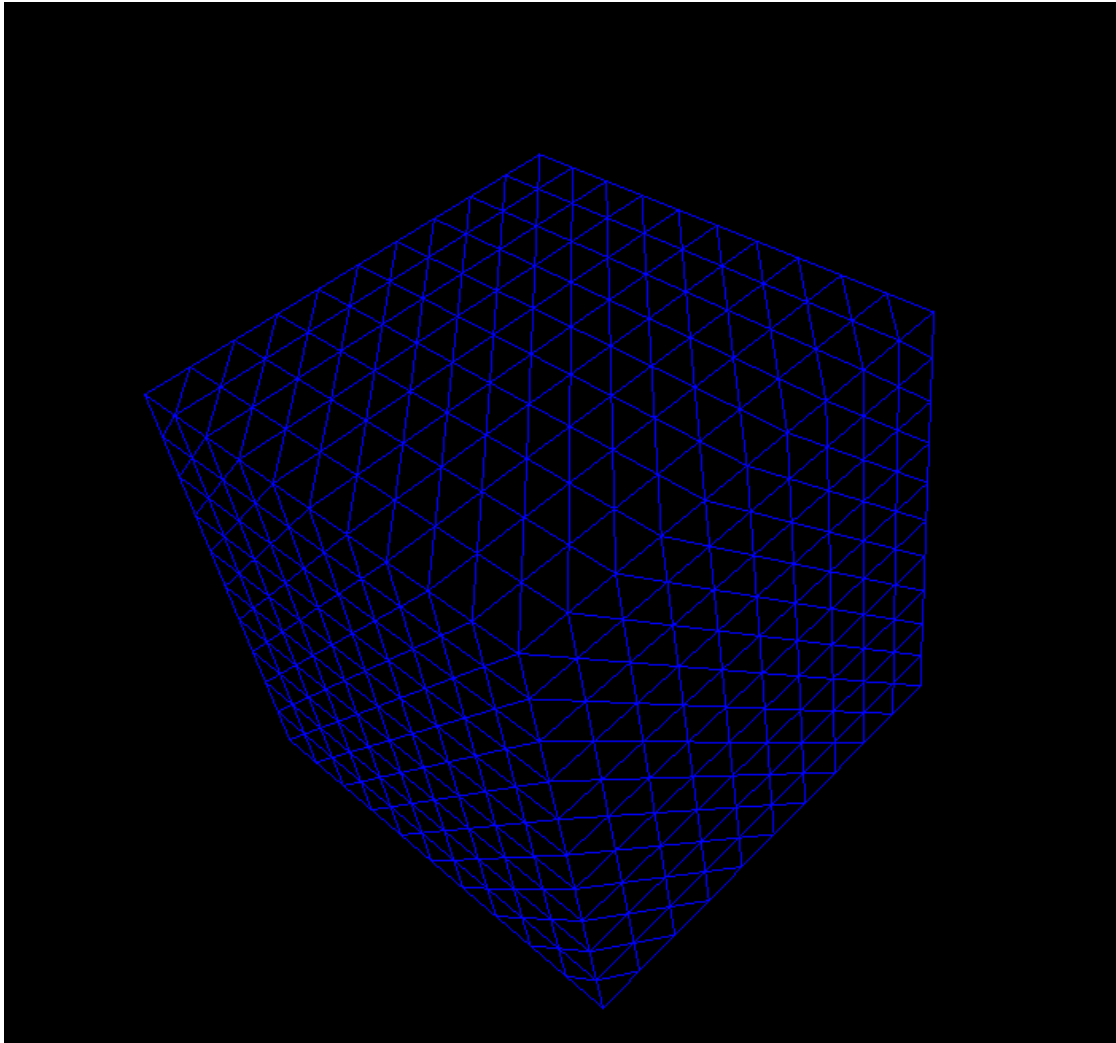


Figura 7: Caixa com 5 unidades de altura, largura e comprimento e com 10 divisões horizontais e 10 divisões verticais

Comandos utilizados:

```
./generator box 5 5 5 10 10 box.3d
```

```
./engine config.xml
```

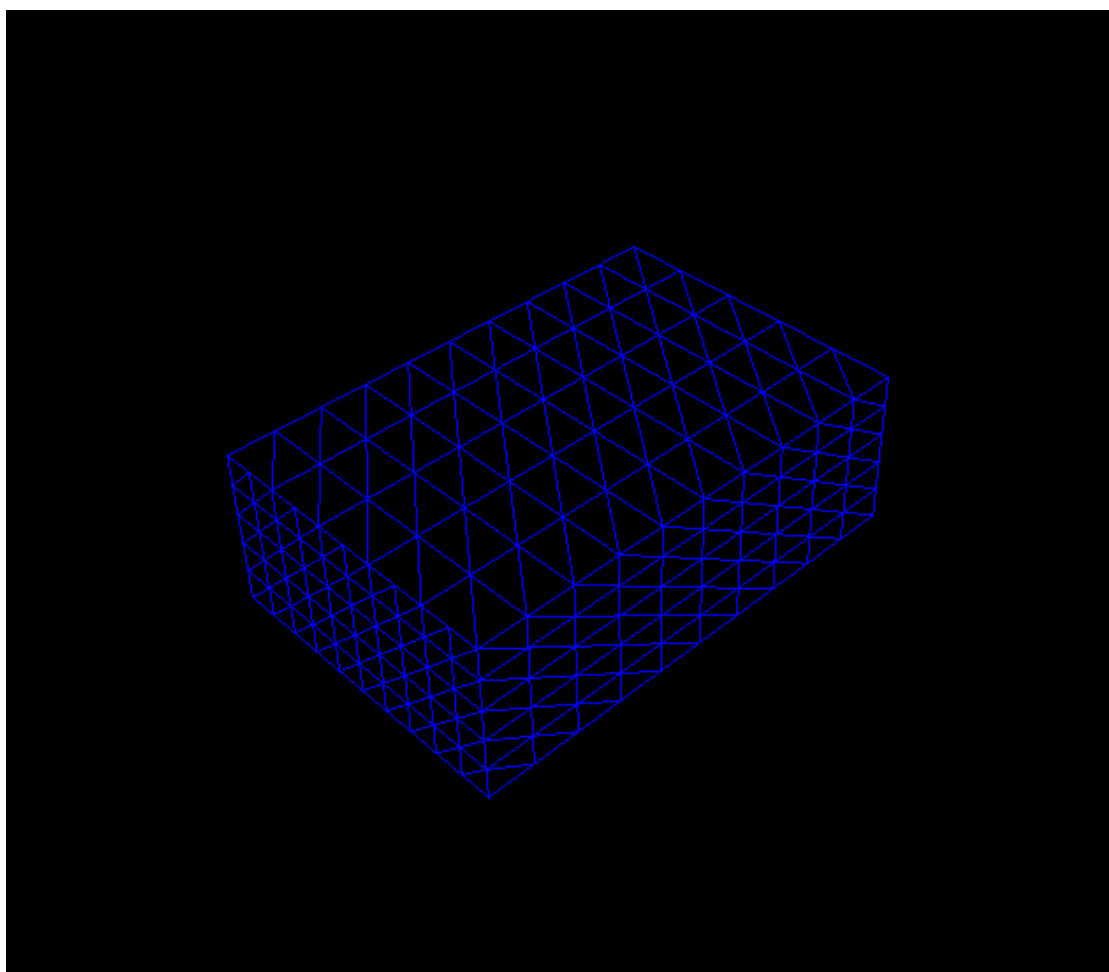


Figura 8: Caixa com 2 unidades de altura, 4 unidades de largura e 6 de comprimento e com 5 divisões horizontais e 10 divisões verticais

Comandos utilizados:

```
./generator box 2 4 6 5 10 box1.3d
```

```
./engine config.xml
```

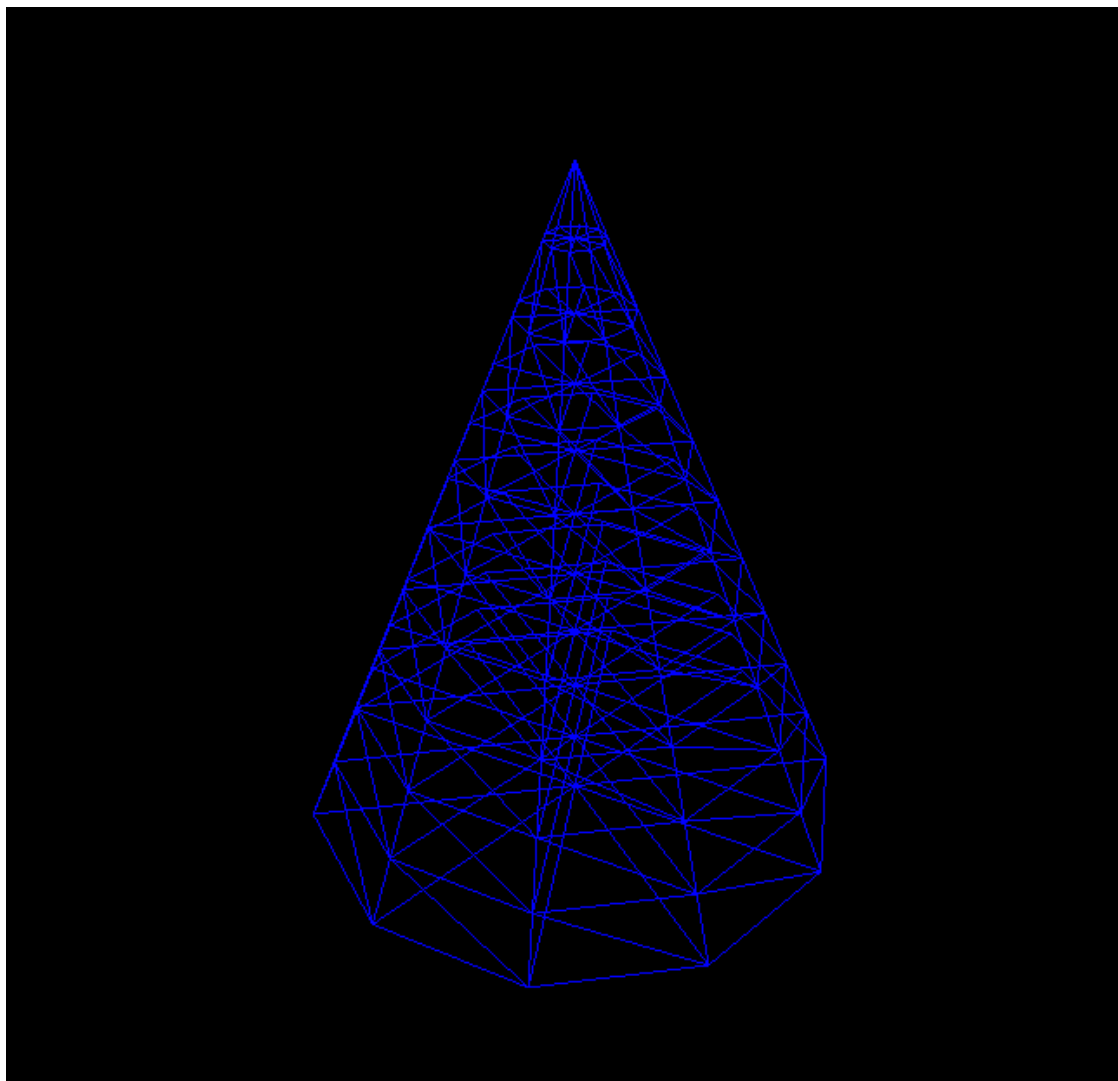


Figura 9: Cone com raio 2, altura 5, 10 fatias e camadas

Comandos utilizados:

```
./generator cone 2 5 10 10 cone.3d  
./engine config.xml
```

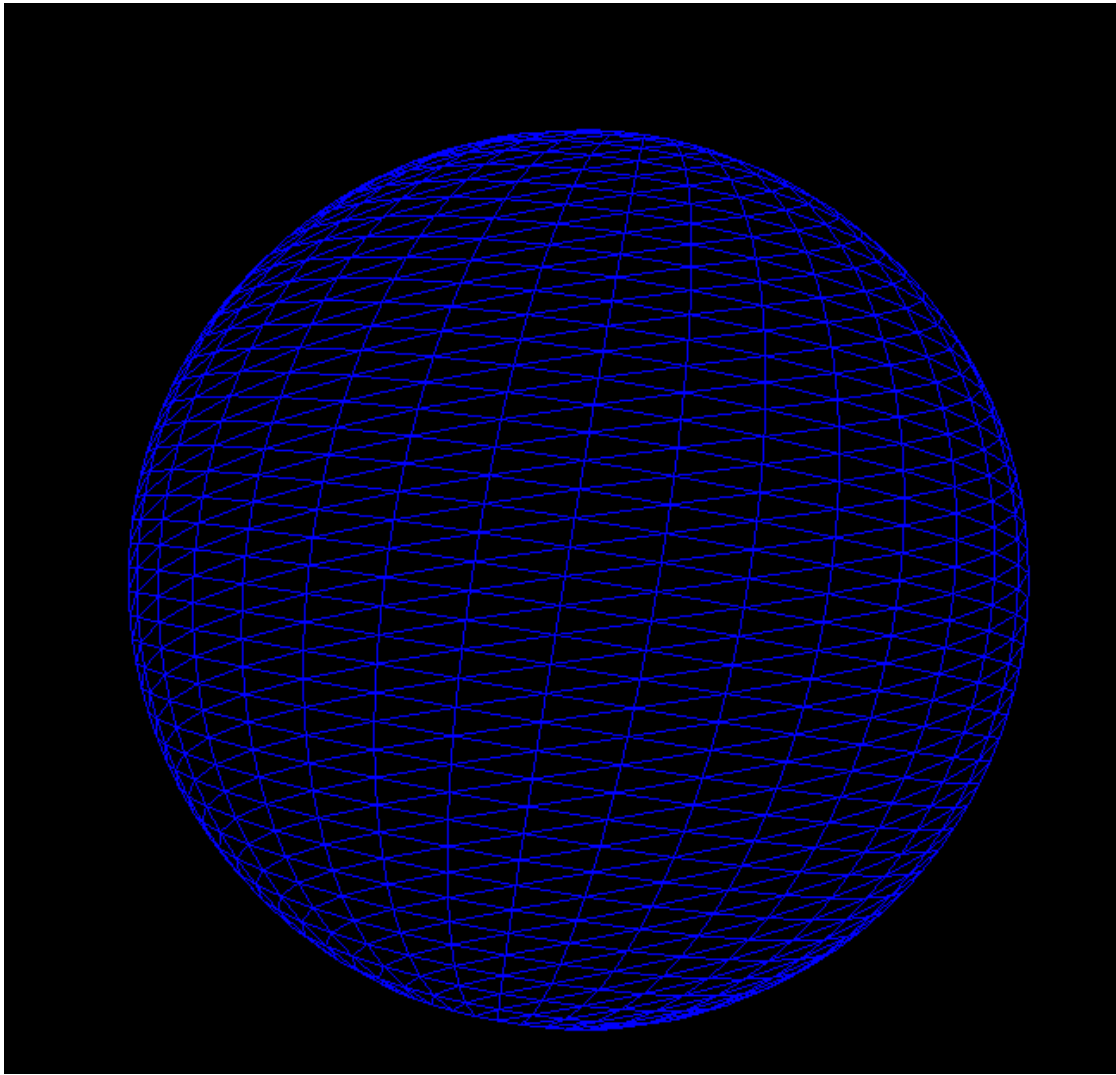


Figura 10: Esfera com raio 5 e 40 fatias e camadas

Comandos utilizados:

```
./generator sphere 5 40 40 sphere.3d  
./engine config.xml
```

Após a visualização dos modelos requisitados, podemos interagir com estes através do teclado. As teclas A,W,S,D permitem efetuar rotações no modelo apresentado, enquanto as setas permitem efetuar translações através do Eixo X e do Eixo Y. Podemos ainda fazer zoom do modelo, diminuindo ou aumentando (atra-



vés da tecla + e da tecla -, respetivamente), mas tendo em atenção para que não haja um aumento ou uma diminuição superior ao desejável para que algo como "olhar para dentro do sólido" não seja possível. Para além disso, é possível representar o modelo por **linhas** (comando L), por **pontos** (comando P) ou o modelo **preenchido** (comando F).

## 4 Conclusão

Na primeira fase deste projeto foi nos pedido que, através dos conhecimentos lecionados até ao momento, desenvolvêssemos dois programas: O Generator, um programa capaz de gerar o conjunto de vértices de uma figura quando introduzidos os parâmetros desejados e o Engine que deverá ser capaz de consumir os diversos ficheiros produzidos pelo Generator para gerar os seus respetivos modelos 3d dinâmicos.

No ponto de vista do grupo, e após uma análise sucinta dos resultados obtivos, achamos que fomos capazes de pôr em uso os conhecimentos lecionados de forma a criar ambos os programas de forma extremamente satisfatória.