



**Universidade do Minho**  
Escola de Engenharia

# **Computação Gráfica**

Fase 1- Primitivas gráficas  
Grupo 7

Braga, Março de 2023

Bernardo Amado Pereira da Costa, A95052  
Eduardo Miguel Pacheco Silva, A95345  
José Carlos Gonçalves Braz, A96168

## *Índice*

<b>1. Introdução .....</b>	<b>3</b>
<b>2. Modelos tridimensionais.....</b>	<b>4</b>
<b>2.1. Plano.....</b>	<b>4</b>
<b>2.2. Caixa.....</b>	<b>5</b>
<b>2.3. Esfera.....</b>	<b>5</b>
<b>2.4. Cone.....</b>	<b>6</b>
<b>3. Generator .....</b>	<b>7</b>
<b>3.1. Estrutura de ficheiro.....</b>	<b>7</b>
<b>4. Engine.....</b>	<b>7</b>
<b>4.1. TinyXml2.....</b>	<b>8</b>
<b>5. Gerar modelos e demais comandos.....</b>	<b>9</b>
<b>6. Conclusão e análise de resultados .....</b>	<b>10</b>

# 1. Introdução

Nesta primeira fase foi-nos pedido para criar primitivas gráficas. Assim sendo, e já com os conhecimentos básicos de programação no âmbito da modelação que temos aprendido nas aulas, pudemos começar a planear e executar o início deste projeto. No que concerne ao planeamento e organização, nesta fase inicial, entre os demais ficheiros programados, temos três essenciais: *geometricShapes*, *generator* e *engine*. O primeiro tem como objetivo definir as classes das primitivas e a própria estrutura de pontos. O *generator* cria objetos geométricos tridimensionais, como plano, caixa, cone e esfera, e grava as suas informações num arquivo 3D. O *engine* é importantíssimo, sendo responsável pela leitura de ficheiros de configuração e renderização das formas geométricas.

## 2. Modelos tridimensionais

### 2.1. Plano

Para a realização do plano e das demais figuras trabalhamos à base de triângulos como visto nas aulas. Primeiramente, definimos os construtores de classe para o mesmo e em seguida criamos vetores de vértices através da função *push\_back* que nos permitem o desenho desta forma geométrica.

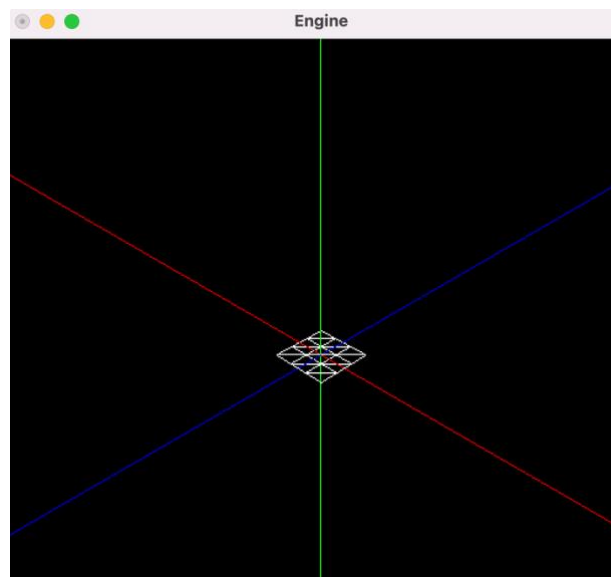


Figura 1-Exemplo de Plano "plane.3d"

## 2.2. Caixa

Após a criação de construtores para o que é a nossa *box* trabalhamos com as vistas de baixo e de cima, frontal e traseira e direita e esquerda, fazendo 6 planos e trabalhando com os 3 diferentes eixos.

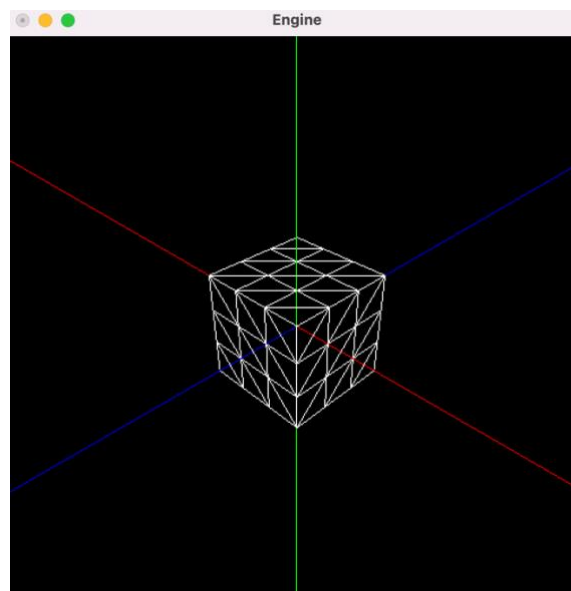


Figura 2- Exemplo da caixa "box.3d"

## 2.3. Esfera

Para a construção da esfera decidimos utilizar coordenadas esféricas devido á facilidade de encontrar um vértice através dos ângulos alfa e beta (e das operações trigonométricas associadas aos mesmos) e dum raio. Sabemos que considerando como ponto de referência o centro da esfera, o raio será constante e os ângulos alfa e beta para o ponto seguinte serão fáceis de encontrar (contando com a colaboração de stacks e slices).

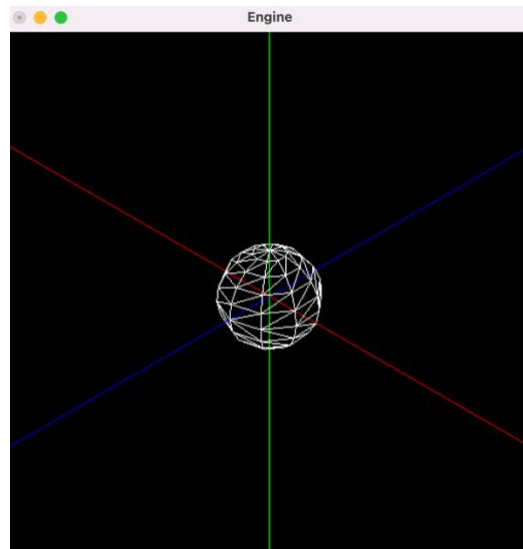


Figura 3- Exemplo de cone "cone.3d"

## 2.4. Cone

A classe possui um método que calcula os pontos que formam o cone. Esse método utiliza as fórmulas matemáticas (onde  $vr$  e  $vh$  são respectivamente a variação do raio e da altura) para calcular os pontos de cada slice e stack do cone e armazena os pontos num vetor de objetos.

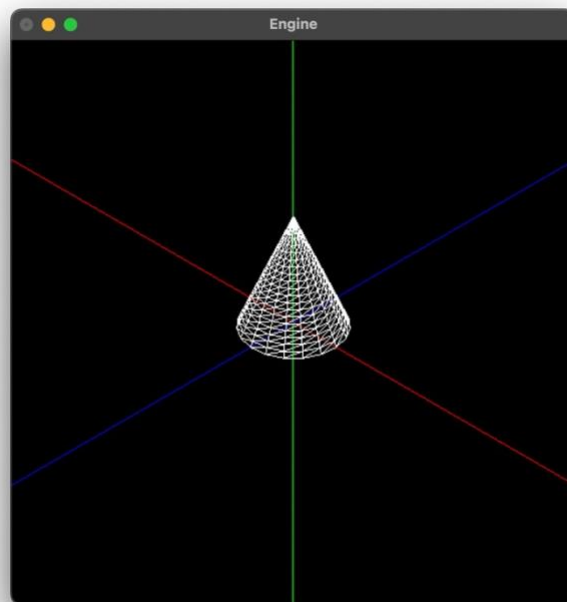


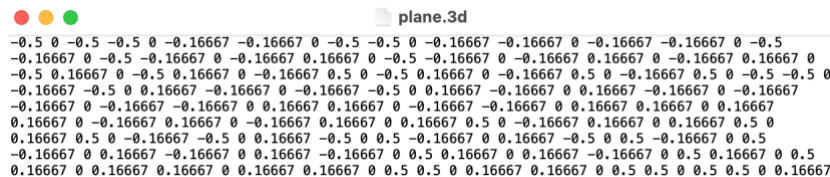
Figura 4- Exemplo de cone "cone.3d"

## 3. Generator

Na função *main*, *GeometricShape* é inicializado com base nos argumentos de linha de comando. Dependendo da forma geométrica especificada, uma nova instância da classe correspondente é criada usando os argumentos fornecidos.

### 3.1. Estrutura de ficheiro

Não existe propriamente separação entre pontos que constituem um vértice, sendo que por exemplo na linha de comandos aparecerão num esquema de linha-a-linha. A identificação do que é um ponto é bastante fácil. Abaixo anexamos um exemplo do ficheiro *plane.3d*. Este contém os pontos pertencentes ao plano.



```
plane.3d
-0.5 0 -0.5 -0.5 0 -0.16667 -0.16667 0 -0.5 0 -0.16667 -0.16667 0 -0.16667 -0.16667 0 -0.5
-0.16667 0 -0.5 -0.16667 0 -0.16667 0.16667 0 -0.5 -0.16667 0 -0.16667 0.16667 0 -0.16667 0.16667 0
-0.5 0.16667 0 -0.5 0.16667 0 -0.16667 0.5 0 -0.5 0.16667 0 -0.16667 0.5 0 -0.16667 0.5 0 -0.5 0
-0.16667 -0.5 0 0.16667 -0.16667 0 -0.16667 -0.5 0 0.16667 -0.16667 0 0.16667 -0.16667 0 -0.16667 0
-0.16667 0 -0.16667 -0.16667 0 0.16667 0.16667 0 -0.16667 -0.16667 0.16667 0.16667 0 0.16667
0.16667 0 -0.16667 0.16667 0 -0.16667 0.16667 0 0.16667 0.5 0 -0.16667 0.16667 0 0.16667 0.5 0
0.16667 0.5 0 -0.16667 -0.5 0 0.16667 -0.5 0 0.5 -0.16667 0 0.16667 -0.5 0 0.5 -0.16667 0.5
-0.16667 0 0.16667 -0.16667 0 0.16667 -0.16667 0 0.5 0.16667 0 0.16667 -0.16667 0 0.5 0.16667 0.5
0.16667 0 0.16667 0.16667 0 0.16667 0.16667 0 0.5 0.5 0 0.16667 0.16667 0 0.5 0.5 0 0.5 0.5 0 0.16667
```

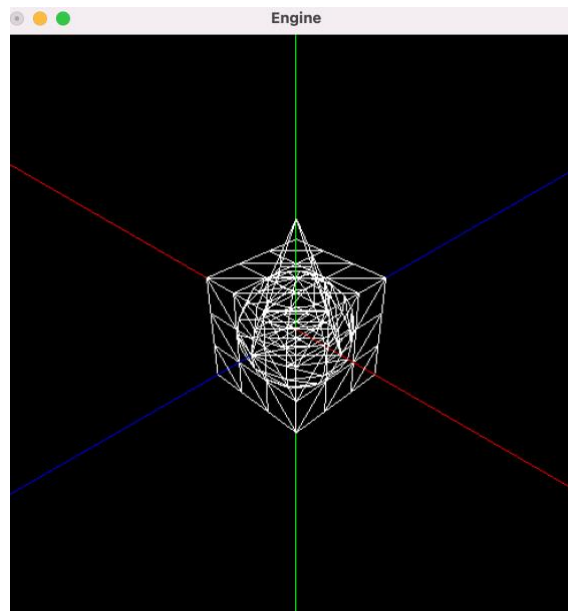
Figura 5- Ficheiro *plane.3d*

## 4. Engine

Neste ficheiro criámos um “*World*” a partir do arquivo .XML de configuração que contém informações sobre a cena a ser renderizada. Em seguida o programa obtém as informações da câmara, projeção e arquivos de objetos a serem renderizados. Quando o utilizador pressiona a tecla “X”, existe alteração entre o modo de preenchimento e o contorno das figuras.

```
<group>
  <models>
    <model file="cone.3d" />
    <model file="sphere.3d" />
    <model file="box.3d" />
    <model file="plane.3d" />
  </models>
</group>
```

Figura 6- Exemplo do *config.xml*



*Figura 7- Exemplo de todas as figuras pretendidas nesta fase*

## 4.1. TinyXml2

De forma a extrair a informação concebida no ficheiro de extensão .XML procuramos uma opção viável e fácil de traduzir o conteúdo com os devidos formatos para que seja possível a criação das figuras com os parâmetros desejados. Dado isso, foi escolhida a biblioteca TinyXml2 que proporciona um leque de funcionalidades que permitem a realização do que foi referido acima.

A forma como foram incutidas as dependências desta biblioteca no nosso projeto estão presentes no ficheiro CMakeLists.txt:

```
# Add the tinyxml2 library
include(FetchContent)
FetchContent_Declare(
  tinyxml2
  GIT_REPOSITORY https://github.com/leethomason/tinyxml2.git
  GIT_TAG 9.0.0
)
FetchContent_MakeAvailable(
  tinyxml2
)
#add_executable(${PROJECT_NAME} engine.cpp geometricShapes.cpp)
add_executable(${PROJECT_NAME} engine.cpp geometricShapes.cpp geometricShapes.h parseXML.cpp)
target_link_libraries(${PROJECT_NAME} tinyxml2::tinyxml2)
```

*Figura 8- fetch das dependências*

Posteriormente, estas funcionalidades são utilizadas nos ficheiros parseXML.cpp e parseXML.h para atribuir os respetivos valores passados pelo utilizador no ficheiro de configuração para uma estrutura que, por sua vez, é utilizada no processo de renderização das figuras geométricas.



## 5. Gerar modelos e demais comandos

Para a geração do ficheiro corremos os seguintes comandos:

```
$ cmake ..
```

```
$/group_project <nome_da_figura> <argumentos>
```

O parâmetro `<nome_da_figura>` pode ser "plane", "box", "cone" ou "sphere", dependendo do tipo de figura que se deseja gerar. Os argumentos necessários para cada figura são os seguintes:

- Para "plane": `<dimensão> <divisão_edge> <nome_do_arquivo>`
- Para "box": `<dimensão> <divisão_edge> <nome_do_arquivo>`
- Para "cone": `<raio> <altura> <fatias> <camadas> <nome_do_arquivo>`
- Para "sphere": `<raio> <fatias> <camadas> <nome_do_arquivo>`

Onde para `<nome_do_arquivo>` se pode entender pelos nomes que o utilizador quiser dar aos ficheiros com a extensão `.3d` que foram gerados. No nosso caso são, respetivamente e seguindo a ordem anterior, *plane.3d*, *box.3d*, *cone.3d* e *sphere.3d*.

Para visualização das figuras alteramos o ficheiro que é referente à execução do programa, alterando a *source* de `generator.cpp` para `engine.cpp`. Com isto, conseguimos, através do seguinte comando: `$/group_project` ter a visualização completa e requerida de qualquer primitiva.

## **6. Conclusão e análise de resultados**

Em resumo, na primeira fase deste projeto de modelação 3D, criamos primitivas gráficas como plano, caixa, esfera e cone. Para isso, utilizamos os conhecimentos de programação que aprendemos nas aulas, criamos as classes das primitivas, geramos objetos geométricos tridimensionais e lemos os arquivos de configuração para renderização das formas geométricas. Utilizamos a biblioteca TinyXml2 para extrair informações do arquivo de configuração e atribuir os valores aos parâmetros das primitivas. Com a realização e consequente conclusão da primeira fase, estamos prontos para as próximas etapas deste projeto.