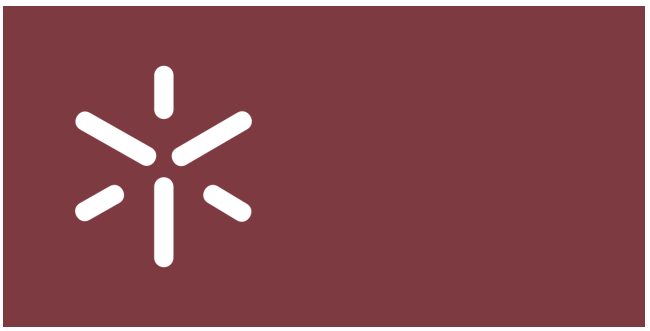

# Universidade do Minho

Escola de Engenharia

Licenciatura em Engenharia Informática

Projeto da U.C. Computação Gráfica

1ª Fase

**Ano letivo 2023/2024**


## Realizado por:

A91672 - Luís Ferreira
A93258 - Bernardo Lima
A100543 - João Pastore
A100554 - David Teixeira

# Conteúdo

# Lista de Figuras

# 1 Introdução

No âmbito da Unidade Curricular de Computação Gráfica, este projeto será desenvolvido utilizando a linguagem de programação *C++* e a biblioteca *OpenGL*. O objetivo é criar um *engine* capaz de gerar e exibir uma variedade de primitivas gráficas. Nesta primeira fase, concentramo-nos na implementação de um **gerador** e de um **motor** com base nas especificações propostas.

# 2 Arquitetura do Projeto

O projeto foi estruturado em diferentes *packages*, visando alcançar uma modularização eficiente e uma organização clara do código. Cada diretoria representa uma parte específica do projeto e contém os componentes necessários para garantir a sua funcionalidade. Nas próximas secções, abordaremos (de uma forma geral) cada um deles.

## 2.1 Generator

O diretório `Generator` abriga o código responsável pelo cálculo das coordenadas dos pontos necessários para representar planos, caixas, esferas e cones. Além disso, é aqui que são criados os ficheiros com a extensão `.3d`, que posteriormente serão utilizados pelo módulo `Engine`.

## 2.2 Engine

Esta diretoria contém o código responsável pela visualização tridimensional dos modelos. Aqui são implementadas as funcionalidades necessárias para renderização e interação com os modelos tridimensionais.

## 2.3 Utils

`Utils` contém estruturas de dados e funções auxiliares utilizadas pelo módulo `Engine` e pelo `Generator`. Estas estruturas e funções são fundamentais para o funcionamento adequado de ambos os módulos, proporcionando uma base sólida para o desenvolvimento do projeto.

## 2.4 TinyXML

Esta diretoria inclui a biblioteca `TinyXML2`, uma ferramenta auxiliar para a leitura de ficheiros *XML*. A biblioteca é essencial para a manipulação de dados *XML* no projeto, que serão utilizados pelo `Engine`.

# 3 Especificação da Arquitetura

## 3.1 Generator

**Generator** é uma ferramenta responsável por calcular os pontos que compõem as diversas primitivas e os colocar num ficheiro `.3d`. Com a sua capacidade de receber argumentos variáveis, pode-se facilmente gerar uma variedade de formas, desde planos simples até estruturas mais complexas. As primitivas suportadas incluem:

- **Plano**: Definido pelo tamanho da aresta e o número de divisões.
- **Caixa**: Determinada pelo tamanho da aresta e o número de divisões em cada face.
- **Esfera**: Especificada pelo raio e o número de fatias horizontais e verticais.
- **Cone**: Caracterizada pelo raio da base, altura e o número de fatias verticais e horizontais.

Além disso, **Generator** cria um ficheiro `.3d` que contém os vértices para construir a primitiva desejada, proporcionando uma maneira eficiente de gerar geometrias tridimensionais complexas.

O formato do ficheiro resultante segue padrões rigorosos: a primeira linha representa o número de pontos necessários, enquanto que as linhas seguintes consistem nas coordenadas dos pontos correspondentes. Cada ponto é especificado numa linha separada, representado por três valores do tipo *float* separados por vírgulas. Cada valor representa as coordenadas $X$, $Y$ e $Z$, respectivamente. Para além disso, cada grupo de 3 pontos forma um triângulo novo. Aqui está um exemplo de um ficheiro `.3d` que gera os vértices necessários para a criação de uma caixa (`box.3d`):

```
324
-1,1,-1
-1,1,-0.333333
-0.333333,1,-1
-0.333333,1,-1
-1,1,-0.333333
-0.333333,1,-0.333333
-1,-1,-1
-0.333333,-1,-1
-1,-1,-0.333333
-0.333333,-1,-1
-0.333333,-1,-0.333333
```

Figura 1: Exemplo de um ficheiro `.3d` para a modelação de uma caixa

Seguidamente, iremos apresentar uma perspectiva do ponto de vista do código criado.

### 3.1.1 Generator.cpp

Para a criação das primitivas, utilizamos o ficheiro `generator.cpp` que obtém os argumentos necessários, lidando assim com possíveis erros de *input*. Uma vez validados os argumentos, recorremos ao `geometry.cpp` para a geração dos pontos das primitivas.

### 3.1.2 Geometry.cpp

Nesta secção, descreveremos as funções responsáveis por gerar as diversas primitivas gráficas.

- **Plano (`generatePlane`) :** Calculamos o número de vértices com base no número de divisões, organizando-os em dois triângulos por divisão quadrada do plano. Isto é feito especificando as coordenadas dos três vértices para cada triângulo. Um total de 6 vértices (2 triângulos com 3 vértices cada) é gerado para cada quadrado da divisão. As coordenadas ($x$, $z$) de cada vértice são calculadas iterativamente, com altura (coordenada $y$) definida como 0 para um plano alinhado com o eixo $XZ$. O tamanho de cada quadrado é calculado dividindo-se o tamanho total do plano (*size*) pelo número de divisões. Isto determina a distância entre os vértices adjacentes na malha. O plano é centralizado ao redor da origem, subtraindo-se metade do tamanho total do plano ($half = size/2.0f$) das coordenadas $x$ e $z$ dos vértices. Isto garante que o plano seja gerado simetricamente em relação ao eixo central.

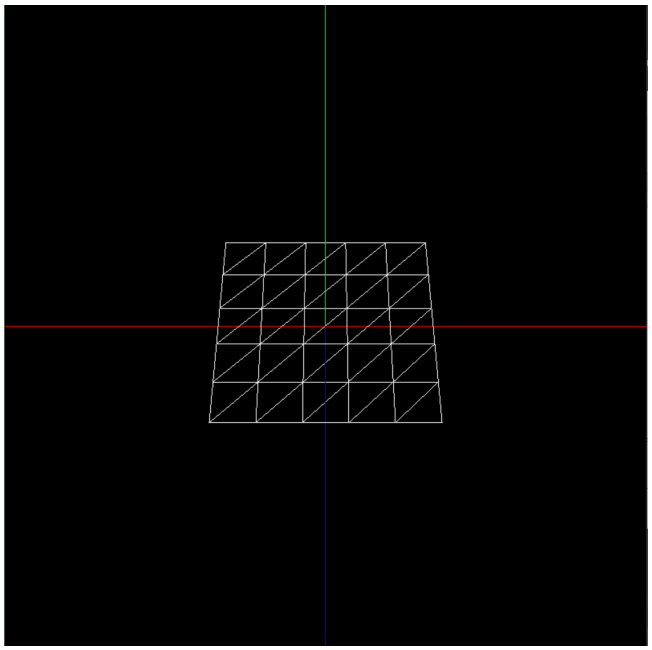

Figura 2: Exemplo de um plano

- **Caixa (`generateBox`) :** Inicialmente, é calculado o tamanho da aresta de cada divisão (*step*) e metade do tamanho da aresta do cubo (*half*), de modo a centrar este na origem do referencial $(0, 0, 0)$. Seguidamente, é calculado o número de vértices da primitiva, multiplicando os 6 vértices de cada quadrado (dois triângulos) pelas 6 faces, multiplicando novamente este valor pelo quadrado das divisões do cubo e guardamos o resultado em ficheiro.
  Referente ao cálculo dos vértices do cubo, este é calculado com base numa limitação em cada face na direção $y$, em que a cada iteração de $x$ é incrementado um multiplo de *step*, sendo a disposição dos triangulos orientada a $x$, em todas as faces do cubo, estando o último eixo limitado à face correspondente:
  - **Topo:** $y = half$, **Base:** $y = -half$
  - **Direita:** $x = half$, **Esquerda:** $x = -half$
  - **Frente:** $z = half$, **Trás:** $z = -half$

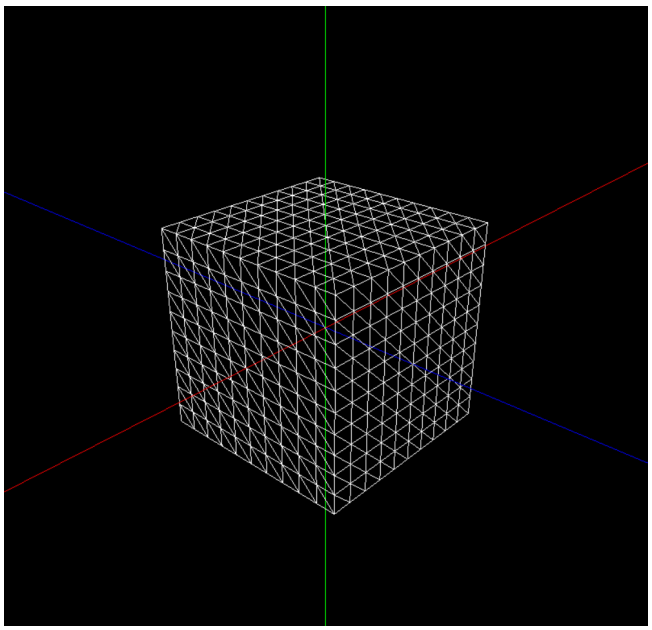

Figura 3: Exemplo de uma caixa

- **Esfera (`generateSphere`) :** O número de vértices é calculado e iteramos através de camadas e fatias da esfera, para calcular as coordenadas de cada vértice utilizando fórmulas esféricas. São utilizadas coordenadas polares esféricas ($\theta$ e $\phi$ para **ângulos polares**) para calcular as posições dos pontos na superfície da esfera. Os valores de $\theta$ variam de $0$ *a* $2\pi$ e os de $\phi$ de $0$ *a* $\pi$.
  Para **cada par de fatias e camadas**, os pontos da superfície da esfera **são calculados** utilizando as fórmulas:

$$x = \sin(\phi)\cos(\theta)$$
$$y = r\cos(\phi)$$
$$z = r\sin(\phi)\sin(\theta)$$

  (onde $r$ é o raio da esfera). Cada par de fatias adjacentes e camadas forma um quadrado na superfície da esfera, que é dividido em dois triângulos para facilitar a renderização gráfica. Isto é feito, conectando-se os pontos calculados para formar triângulos. O código itera sobre todas as fatias e camadas, calculando os pontos e formando triângulos, que são então armazenados ou processados conforme necessário.

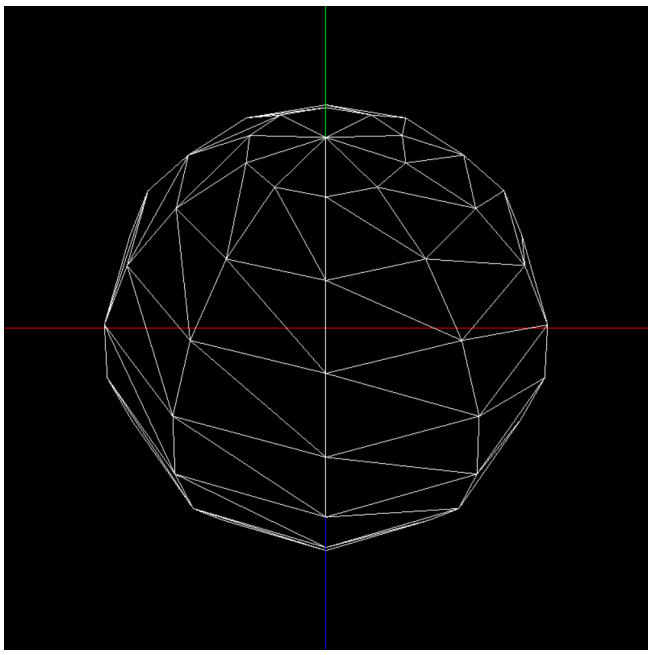

Figura 4: Exemplo de uma esfera

- **Cone (`generateCone`) :** Calculamos o número total de vértices, formando a base e os lados do cone.

  Para cada **fatia**, calculamos os vértices na **borda da base** e conectamos ao **centro** para formar os triângulos da base. Seguidamente, geramos os vértices na borda da base e um vértice no topo para formar os triângulos que compõem a superfície lateral do cone. A base do cone é gerada ao criar um triângulo para cada fatia, com um vértice no centro da base $(0, 0, 0)$ e os outros dois nos pontos ao longo da borda da base, calculados utilizando coordenadas polares com o raio fornecido. Estes triângulos garantem que a base seja renderizada corretamente, com a normal apontada para baixo. O ângulo entre cada fatia (*deltaAngle*) é calculado para distribuir uniformemente os pontos ao redor da base do cone.
  A altura de cada camada (*deltaHeight*) é calculada, embora a variação da altura nas laterais do cone não seja tratada, pois a função cria um cone sólido sem suavizar a transição do raio.
  Os lados do cone são formados conectando o vértice superior (a ponta do cone) a cada par de pontos adjacentes ao longo da borda da base. Isto cria um conjunto de triângulos que cobrem as laterais do cone, com a ponta do cone sendo um vértice comum a todos esses triângulos.
  As coordenadas dos vértices são calculadas utilizando funções trigonométricas, baseadas no ângulo atual (*angle* e *nextAngle*), para posicionar os pontos ao longo da circunferência da base do cone. O vértice superior é fixo no ponto $(0, altura, 0)$.

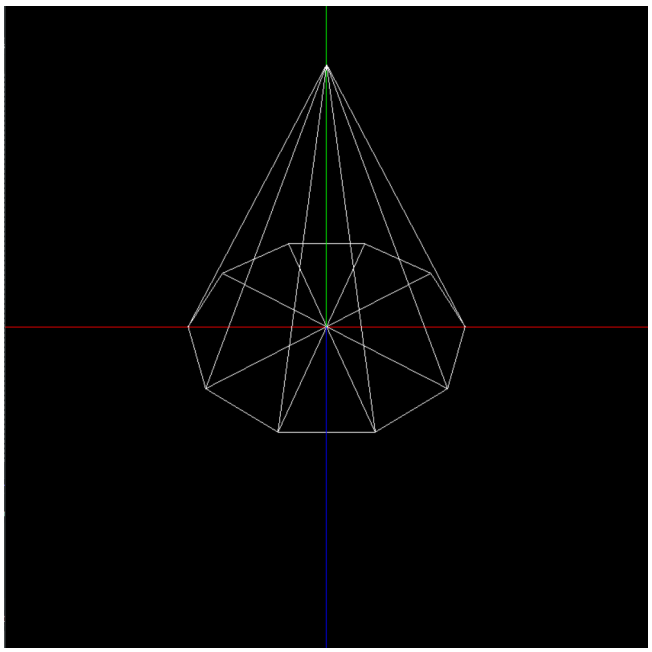

Figura 5: Exemplo de um cone

## 3.2 Engine

### 3.2.1 Engine.cpp

**Engine.cpp** possui a capacidade de receber um ficheiro de configuração *XML*, que possui configurações referentes à **câmara** e a quais ficheiros de primitivas (obtidos pelo *generator*) carregar. Este ficheiro *XML* só deve ser lido uma única vez, no arranque do *engine*.

Em termos de funcionalidades propriamente ditas, **Engine** possui a capacidade de renderizar as primitivas passadas, de acordo com as figuras fornecidas pelos docentes.

## 3.3 Utils

Nesta `package` estão agrupadas diversas estruturas essenciais para operações relacionadas com geometria tridimensional e manipulação de ficheiros.

### 3.3.1 Figura.cpp

O ficheiro `Figura.cpp` contém a implementação de uma estrutura de dados que representa uma figura tridimensional. Esta estrutura inclui uma lista de pontos no espaço. As funcionalidades disponíveis incluem a criação de uma nova figura, adição de pontos a uma figura existente, bem como operações para leitura e escrita de figuras em ficheiros.

### 3.3.2 List.cpp

Implementado no ficheiro `List.cpp`, encontra-se uma estrutura de uma lista genérica em C++. Esta lista é capaz de armazenar dados de qualquer tipo. As suas operações abrangem desde a criação de uma nova lista vazia até à adição de elementos, acesso a elementos em posições específicas, e outras operações típicas de manipulação de listas.

### 3.3.3 Parser.cpp

O ficheiro `Parser.cpp` traz a implementação de um parser *XML*, que utiliza a biblioteca `tinyxml2`. Este parser é responsável por analisar um ficheiro *XML* descrevendo configurações de uma cena 3D, tais como configurações de janela, câmera e modelos 3D. As informações extraídas do *XML* são armazenadas numa estrutura de dados para posterior utilização pelo programa `Engine.cpp`.

### 3.3.4 Ponto.cpp

Para representar pontos no espaço tridimensional, temos o ficheiro `Ponto.cpp`. As funcionalidades disponíveis incluem a criação de um novo ponto, obtenção das suas coordenadas, cálculo de distâncias entre pontos e outras operações relacionadas com geometria espacial.

# 4 Demonstração

## 4.1 Guia de Utilização

Instruções para mover a câmera e alternar entre modos de renderização:

- Pressionar as teclas `A` e `D` para mover a câmera para a esquerda e para a direita, respectivamente.
- Utilizar as teclas `W` e `S` para mover a câmera para cima e para baixo, respectivamente.
- Pressionar as teclas `F`, `L` e `P` para alternar entre os modos de renderização `GL_FILL` (preenchido), `GL_LINE` (linhas) e `GL_POINT` (pontos), respectivamente.

## 4.2 Execução do Programa

Para compilar o projeto, utilizamos um ficheiro *CMakeLists.txt*, que possibilita a compilação através do **CMake**. Após a conclusão do processo de compilação, são gerados dois executáveis: `generator.exe` e `engine.exe`. A execução do `generator.exe` requer a especificação de uma primitiva, juntamente com os seus argumentos correspondentes, seguidos pelo nome do ficheiro `.3d` a ser gerado.
Eis um exemplo de um comando válido :

```
$ ./generator.exe cone 1 2 4 3 cone.3d
```

Para o funcionamento adequado do *engine*, é pedido exclusivamente o fornecimento do ficheiro *XML* que contém informações pertinentes à câmera, juntamente com os ficheiros gerados pelo gerador que se deseja carregar.
Eis um exemplo de um comando válido :

```
$ ./engine.exe /caminho/para/o/xml/test_1_1.xml
```

# 5 Conclusão

A presente etapa do trabalho viabilizou a consolidação dos conhecimentos adquiridos nas aulas teóricas e práticas ao longo das últimas semanas, bem como proporcionou a aquisição de novos conhecimentos. Acredita-se que todos os objetivos delineados para esta fase foram alcançados de forma eficaz, incluindo a correta implementação do **Generator** e do **Engine**. Além disso, foram incorporados elementos adicionais, tais como a capacidade de interação com o sistema, possibilitando a manipulação da câmera. Por fim, entende-se que foram reunidos os elementos necessários para avançar para a próxima fase do projeto.