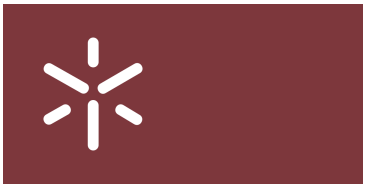

**Universidade do Minho**
Escola de Engenharia
Licenciatura em Engenharia Informática
Licenciatura em Engenharia Informática

## Unidade Curricular - Computação Gráfica

Ano Letivo de 2024/2025

# Fase 1 - Primitivas Gráficas
# Grupo 36

**Tomás Henrique Alves Melo** a104529
**Carlos Eduardo Martins de Sá Fernandes** a100890
**Hugo Rafael Lima Pereira** a93752
**Alexandre Marques Miranda** a104445

CG

Março, 2025

# Resumo

Este relatório tem como objetivo mostrar detalhadamente as aplicações Generator e Engine desenvolvidas, focando principalmente na forma como a construção das primitivas foi pensada, trabalhada e executada de acordo com as melhores decisões possíveis para cada primitiva, garantindo a geração do número mínimo de vértices para a construção de cada uma das primitivas desejadas, e combinação cuidada dos índices para formar corretamente os triângulos que compõem a primitiva em questão. Para além disso, abordará a forma como toda esta fase foi projetada de modo a cumprir todos os requisitos pedidos no enunciado do trabalho prático, como as estruturas de dados que foram criadas e a forma como foram relacionadas umas com as outras, explicação detalhada de funções que sentimos que tenham sido pertinentes para o sucesso desta fase, entre outros tópicos.

**Área de Aplicação**: Geração, manipulação e desenho de modelos 3D com recurso a C++ e OpenGL.

**Palavras-Chave**: Computação Gráfica, Primitivas Geométricas, Modelação 3D, Motor gráfico 3D, OpenGL

# Índice

# Lista de Figuras

# 1. Introdução

No contexto da unidade curricular de Computação Gráfica, foi desenvolvida a primeira etapa do trabalho prático proposto. O objetivo deste projeto é demonstrar o conhecimento adquirido nas aulas teóricas e práticas através da representação de primitivas gráficas, levando em consideração diferentes parâmetros para cada uma.

Este trabalho será dividido em 4 fases. Para concluir esta primeira etapa, utilizámos a biblioteca OpenGL juntamente com a linguagem de programação C++, de modo a desenvolver os programas Generator e Engine.

# 2. Objetivos Impostos

Nesta fase, o grupo teve como objetivo representar 4 primitivas gráficas: plano, caixa, esfera e cone, utilizando diversos parâmetros, como altura, largura, raio, profundidade, bem como o número de slices e stacks. A representação gráfica destas primitivas é feita através do desenho de múltiplos triângulos, sendo necessário calcular os vértices que compõem cada um desses triângulos.

Para alcançar esse objetivo, foi preciso desenvolver duas aplicações distintas: generator e engine.

O generator é responsável por calcular os vértices necessários para formar as primitivas geométricas e salvá-los em ficheiros .3d.

A engine é responsável por ler a informação dos vértices de uma primitiva a partir de um ficheiro de extensão .3d contido no ficheiro XML, e utilizá-los para desenhar a primitiva correspondente.

Esta divisão em duas aplicações permite separar a lógica de geração de vértices da lógica de renderização gráfica, tornando o sistema modular e mais fácil de manter, tal como especificado no enunciado do projeto!

# 3. Arquitetura desenvolvida

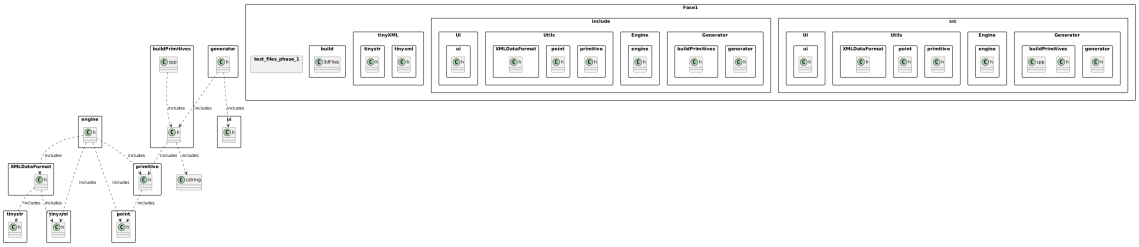

Figura 1: Arquitetura para a Fase 1

O programa está organizado em duas aplicações principais: Generator e Engine, organizadas em duas pastas principais: src e include.

Cada uma dessas pastas contém subdiretorias que armazenam os ficheiros responsáveis pelas funcionalidades específicas de cada aplicação.

A pasta src contém o código-fonte das aplicações com os *packages*:

**Generator**, que reúne os ficheiros responsáveis pela geração das primitivas, incluindo o ficheiro de construção das primitivas (buildPrimitives.cpp) e ainda o ficheiro principal (generator.cpp) que contém a função main, responsável por processar os argumentos fornecidos pelo utilizador e executar as operações necessárias.

**Engine**, que contém os ficheiros responsáveis pela execução do motor gráfico, incluindo o ficheiro principal com a função main, encarregada de interpretar os argumentos do utilizador e executar as ações correspondentes.

**Utils**, que agrupa três ficheiros utilitários que fornecem structs e funções comuns tanto ao Generator quanto ao Engine. Dessa forma, evita-se a repetição de código, promovendo a reutilização e garantindo uma estrutura mais organizada e eficiente.

Para além da pasta src, existe a pasta include, que mantém a mesma organização de subdiretorias (Generator, Engine e Utils), mas destinada exclusivamente aos ficheiros *header* (.h de modo a garantir uma melhor legibilidade para além de que facilita a manutenção do projeto.

Adicionalmente, o projeto utiliza a biblioteca 'tinyXML', armazenada na pasta com o mesmo nome, utilizada para facilitar a leitura de ficheiros XML.

Por fim, existe a pasta 'test_files_phase_1', destinada a armazenar os ficheiros de teste utilizados na fase 1 do desenvolvimento.

Esta arquitetura de projeto, dividida entre src e include, não só organiza as funcionalidades de forma modular, como também permite um desenvolvimento mais flexível e escalável.

# 4. Generator

Esta aplicação tem como função principal criar os vértices e índices que servirão de base para a construção dos triângulos das diversas primitivas geométricas, com recurso a algoritmos específicos para cada tipo de primitiva. Após a geração desses vértices e índices, armazenámo-los num ficheiro com extensão .3d, que, por sua vez, será referenciado dentro de um ficheiro XML.

Para realizar a compilação do projeto, foi utilizado um ficheiro CMakeLists que permite que o projeto seja configurado e construído automaticamente através do CMake.

## 4.1. buildPrimitives

Este ficheiro contém as funções de construção das primitivas propostas (plano, caixa, esfera e cone).

#### 4.1.1. buildPlane

A função buildPlane é responsável por criar os vértices e índices do plano. É gerada uma *mesh* composta por pontos vértices únicos e índices que definem a ordem desses pontos para formar triângulos. Vimos esta abordagem como sendo bastante eficiente, porque reutiliza vértices, de modo a economizar memória e otimiza o desempenho na renderização com OpenGL.

Três estruturas principais são usadas para organizar os dados:

**uniquePoints**: um vetor que armazena apenas os pontos únicos do plano.

**pointIndexMap**: um mapa que relaciona as coordenadas dos pontos com o índice em uniquePoints, garantindo que pontos duplicados não sejam armazenados.

**indices**: um vetor de inteiros que define a ordem dos vértices para formar triângulos na *mesh*.

O cálculo inicial envolve duas variáveis:

**half**: metade do comprimento do plano, usada para centralizar o plano na origem do sistema de coordenadas.

**div_side**: comprimento de cada subdivisão na *mesh*, determinado dividindo o comprimento total pelo número de divisões.

A função então percorre as linhas e colunas da *mesh*, calculando as coordenadas de cada ponto. As coordenadas x e z de cada ponto são determinadas pela posição da linha e coluna na *mesh*. Dependendo do eixo especificado, o plano pode ser orientado em torno dos eixos X, Y ou Z, permitindo criar planos alinhados aos diferentes planos cartesianos (XY, XZ ou YZ). O parâmetro h especifica a altura fixa do plano no eixo escolhido.

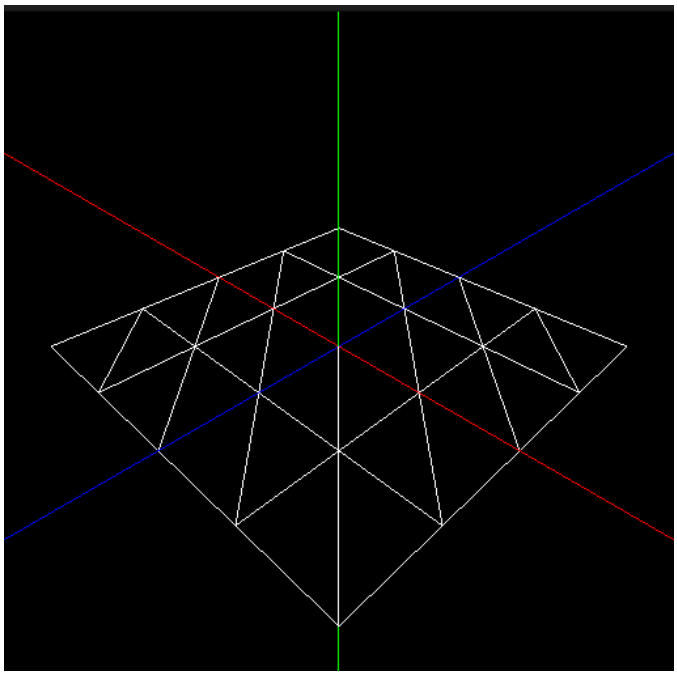

Figura 2: Plane: length = 4, divisions = 4

#### 4.1.1.1. Cálculo de Posições dos Pontos

Para cada ponto da mesh:

- Definimos x e z com base no comprimento total e no número de subdivisões:

$$x = -\frac{\text{length}}{2} + \text{coluna} * \frac{\text{length}}{\text{divisions}}$$

$$z = -\frac{\text{length}}{2} + \text{linha} * \frac{\text{length}}{\text{divisions}}$$

#### 4.1.1.2. Cálculo dos Índices dos Triângulos

Cada quadrado é subdividido em dois triângulos, definidos pelos índices:

- $\text{i1} = \text{linha} * (\text{divisions} + 1) + \text{coluna}$
- $\text{i2} = \text{linha} * (\text{divisions} + 1) + \text{coluna} + 1$
- $\text{i3} = (\text{linha} + 1) * (\text{divisions} + 1) + \text{coluna}$
- $\text{i4} = (\text{linha} + 1) * (\text{divisions} + 1) + \text{coluna} + 1$

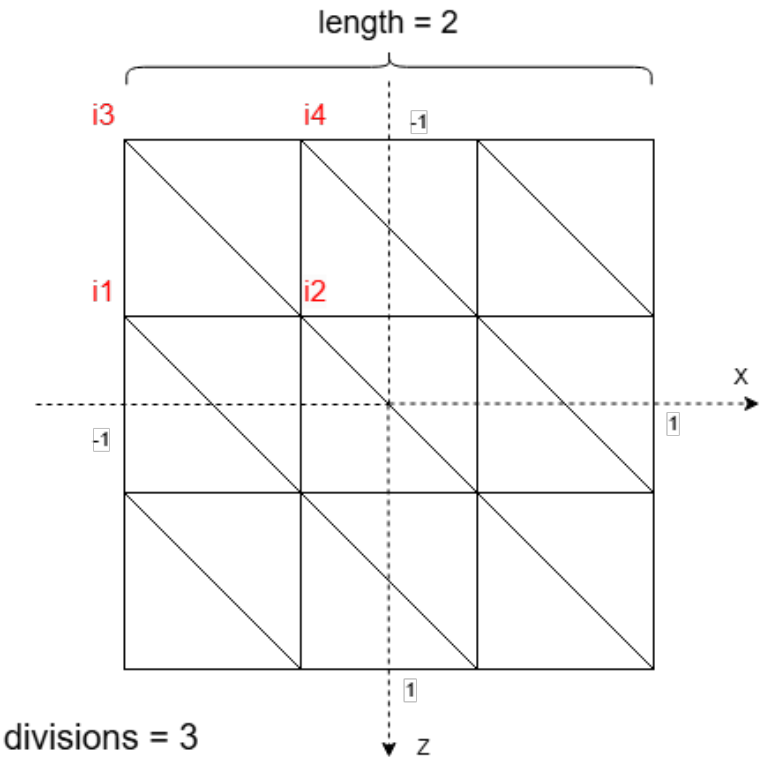


Figura 3: Exemplo da construção de um plano com *length* = 2 e *divisions*= 3

Para além disso o quadrado resultante pode ter a diagonal em duas posições, alterando os indíces na criação dos triângulos, conforme a variável `invertDiagonal`:

- Sem inversão de diagonal:

$$\mathrm{T1} = (\mathrm{i1}, \mathrm{i2}, \mathrm{i3})$$

$$\mathrm{T2} = (\mathrm{i2}, \mathrm{i4}, \mathrm{i3})$$

- Com inversão de diagonal:

$$\mathrm{T1} = (\mathrm{i1}, \mathrm{i4}, \mathrm{i2})$$

$$\mathrm{T2} = (\mathrm{i1}, \mathrm{i3}, \mathrm{i4})$$

#### 4.1.1.3. Inversão das Faces

Se `invertFaces` for verdadeiro, os vértices dos triângulos são rearranjados para que a ordem seja invertida, caso seja preciso. Este parâmetro foi adicionado tendo em vista a variável `invertDiagonal` referida acima, necessária para a construção da *box*.

#### 4.1.1.4. Armazenamento dos Pontos

Após calcular as coordenadas, a função verifica se o ponto já foi adicionado. Para isso, é utilizado o `pointIndexMap`, um mapa que associa as coordenadas tridimensionais do ponto a um índice em `uniquePoints`. Se o ponto ainda não foi armazenado, adicionámo-lo ao vetor `uniquePoints`, e o índice correspondente é armazenado no `pointIndexMap`. Se o ponto já existe, o índice armazenado anteriormente é reutilizado, evitando a duplicação de vértices.

#### 4.1.2. buildBox

A função buildBox é utilizada para criar um cubo 3D subdividido em pequenas células quadradas, aproveitando a reutilização de vértices e índices para otimizar o uso de memória e desempenho na renderização gráfica. A função faz isso ao gerar 6 planos individuais para compor as faces do cubo, utilizando a função buildPlane para construir cada face de maneira eficiente e organizada.

Para centralizar o cubo na origem do sistema de coordenadas, é utilizada a variável `half`, que representa metade do comprimento do cubo. Esta centralização é fundamental para que o cubo seja posicionado simetricamente ao redor da origem.

O cubo é composto por 6 faces: **superior**, **inferior**, **frente**, **trás**, **direita** e **esquerda**.

Cada uma dessas faces é gerada utilizando a função buildPlane. A forma como as faces são construídas e orientadas no espaço 3D é controlada pelo parâmetro `axis`, que representa o eixo normal do plano gerado. A função também utiliza o parâmetro `invertFaces` e `invertDiagonal` para controlar a ordem dos vértices, garantindo que as normais das faces apontam para fora do cubo e que faces consecutivas são invertidas.

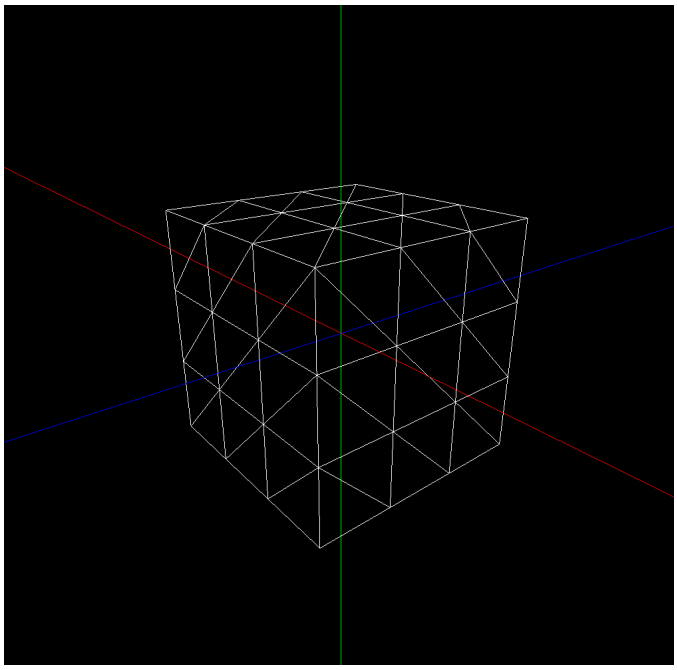

Figura 4: Box: length = 7, divisions = 3

Uma vez que todas as faces foram geradas, a função organiza os pontos e índices para formar a *mesh* completa do cubo. Para isso, são utilizados dois vetores temporários:

**allPoints**: armazena todos os pontos únicos das seis faces.

**allIndices**: define a ordem dos vértices para formar triângulos.

Cada face é processada através da função addFaceToBox, que recebe um Primitive representando uma face do cubo. Dentro dessa função, os pontos e índices da face são extraídos. Para garantir que não haja conflito de índices entre as diferentes faces, um offset é adicionado aos índices de cada face. O offset é calculado com base no tamanho atual de allPoints, garantindo que os índices sejam ajustados corretamente para o vetor global.

Depois de ajustar e adicionar todos os pontos e índices de cada face, a função armazena esses dados na primitiva box utilizando as funções addPoint e setIndices. A `addPoint` adiciona cada ponto único à box, enquanto setIndices define a ordem dos índices para desenhar os triângulos do cubo. Esse uso eficiente de pontos únicos e índices reutilizáveis economiza memória e permite que o OpenGL utilize glDrawElements para renderizar o cubo de forma otimizada.

Por fim, a função retorna a box, que contém todos os dados necessários para renderizar o cubo 3D, incluindo os pontos únicos e a ordem correta dos índices.

#### 4.1.3. buildSphere

A função buildSphere é utilizada para criar uma esfera 3D subdividida em setores (longitude) e stacks (latitude), garantindo a reutilização de vértices e índices para otimizar o uso de memória e o desempenho na renderização gráfica. A função utiliza a parametrização esférica para calcular as coordenadas 3D de cada ponto na superfície da esfera, aproveitando os ângulos de setor (theta) e stack (phi) para gerar os vértices de maneira precisa e eficiente.

Para garantir que a esfera seja fechada corretamente e que não haja vértices duplicados, são utilizados dois polos: o Polo Norte e o Polo Sul. O Polo Norte é posicionado no topo da esfera com coordenadas (0, +radius, 0), enquanto o Polo Sul é colocado na base com coordenadas (0, -radius, 0). Esses polos ajudam a conectar as stacks superior e inferior, evitando problemas de continuidade na *mesh*.

A esfera é composta por:

**Setores** (longitude): São divisões ao longo da circunferência horizontal da esfera, similar às linhas de longitude na Terra. Cada setor representa um ângulo theta, variando de 0 a 2 vezes PI.

**Stacks** (latitude): São divisões ao longo da altura da esfera, similar às linhas de latitude na Terra. Cada stack representa um ângulo phi, variando de 0 a PI.

Cada ponto na superfície da esfera é calculado utilizando as equações paramétricas da esfera:

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

$$y = r\cos(\phi)$$

$$z = r\sin(\phi)\sin(\theta)$$

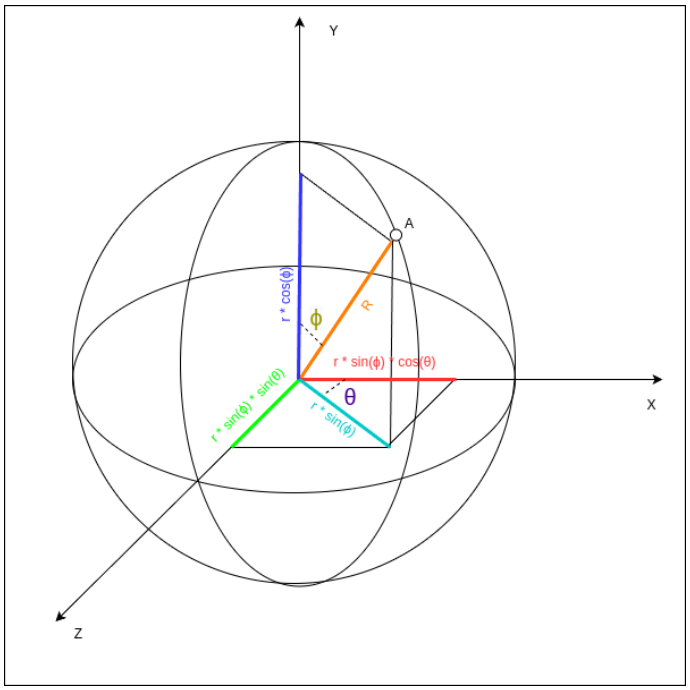


Figura 5: Ponto A na esfera

A função inicia a construção da esfera adicionando o Polo Norte como o primeiro ponto. Em seguida, gera os vértices para as stacks intermediárias (excluindo os polos), iterando de stack = 1 até < stacks, para evitar duplicações nos polos. Em cada stack, são gerados slices vértices ao longo dos setores (theta), garantindo que o último ponto de um setor seja conectado ao primeiro ponto do setor seguinte, formando uma *mesh* contínua.

Após gerar todos os vértices das stacks intermediárias, o Polo Sul é adicionado como o último ponto. Dessa forma, a função evita a repetição de vértices no topo e na base da esfera.

Os índices são organizados em triângulos de forma a conectar os vértices entre as stacks:

- Conexão com o Polo Norte: Cada vértice da primeira stack é conectado ao Polo Norte, formando triângulos que convergem para o topo.
- Conexão entre Stacks Intermediárias: Cada par de stacks adjacentes é conectado por dois triângulos, formando células quadradas na *mesh* esférica.
- Conexão com o Polo Sul: Cada vértice da última stack é conectado ao Polo Sul, formando triângulos que convergem para a base.

Esta organização eficiente dos índices utiliza a propriedade de continuidade angular que acaba por garantir a criação correta da esfera sem criar vértices duplicados, tornando esta a melhor decisão possível na nossa perspetiva.

Após gerar e organizar todos os pontos e índices, a função armazena estes dados na primitiva sphere utilizando as funções `addPoint` e `setIndices`.

Finalmente, a função retorna a esfera que contém todos os dados necessários para renderizar a esfera 3D de maneira eficiente e otimizada.

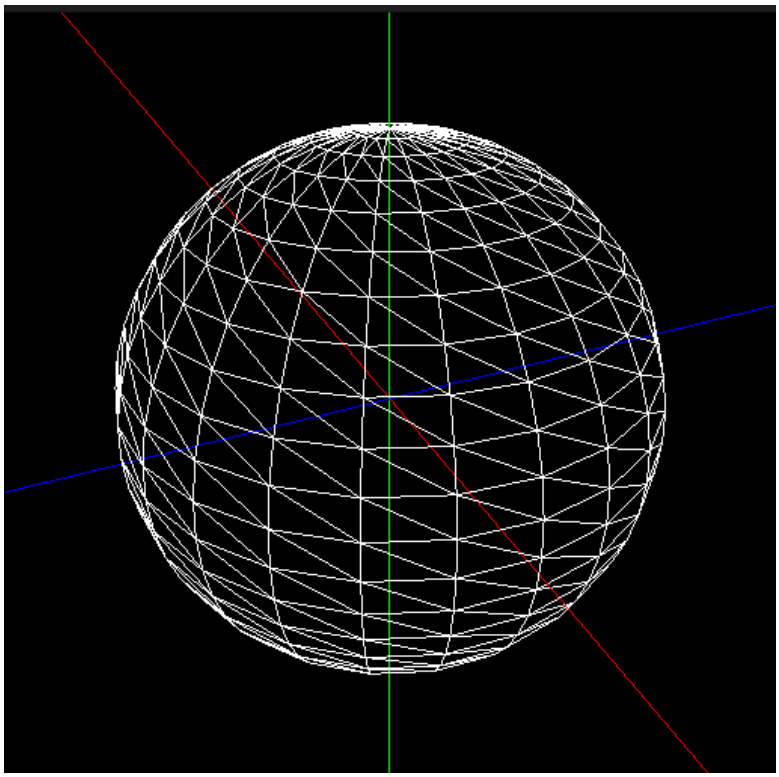

Figura 6: Sphere: radius = 2, slices = 25, stacks = 25

#### 4.1.4. buildCone

A função buildCone é utilizada para criar um cone 3D subdividido em slices e stacks, garantindo a reutilização de vértices e índices para evitar duplicação de dados e otimizar a renderização gráfica, tal como feito ao longo das primitivas anteriormente mostradas. O cone é construído com base na XZ-plane tal como pedido no enunciado e ápice no eixo Y positivo.

Para garantir a correta construção do cone sem vértices repetidos, a função verifica a existência de cada ponto antes de adicioná-lo, reutilizando vértices previamente gerados, evitando que cada stack crie novos pontos para coordenadas já existentes.

O cone é composto por três partes principais:

**Base**: Um círculo no plano XZ, composto por slices pontos ao longo da borda e um único vértice central.

**Corpo**: Camadas de vértices ao longo da altura, diminuindo progressivamente o raio para formar a superfície cónica.

**Topo**: Um único vértice no ápice do cone, conectado aos vértices da última camada.

Cada ponto na superfície do cone é calculado utilizando coordenadas cilíndricas:

- **Base** (y = 0, raio = radius):
  - $\mathrm{x}=\mathrm{radius}\times\cos(\theta)$
  - $\mathrm{z}=\mathrm{radius}\times\sin(\theta)$
  - $\mathrm{y}=0$

- **Corpo** (altura varia de 0 a height):
  - $\mathrm{x}=\mathrm{currentRadius}\times\cos(\theta)$
  - $\mathrm{z}=\mathrm{currentRadius}\times\sin(\theta)$
  - $\mathrm{y}=\mathrm{currentHeight}$

  Onde:

  $\mathrm{currentRadius} = \mathrm{radius}$ * $(1.0$ - $\mathrm{stack}$ / $\mathrm{stacks})$, reduzindo progressivamente o raio.
  $\mathrm{currentHeight} = (\mathrm{stack}$ / $\mathrm{stacks})$ * $\mathrm{height}$, aumentando a altura.

- **Topo** (y = height, x = 0, z = 0):
  - Um único ponto no topo do cone.

A função addUniquePoint é utilizada para evitar que pontos sejam duplicado ao verificar se já existem antes de adicionar novos.

Os índices dos triângulos são organizados para conectar os vértices da melhor forma possível a nosso ver.

**Base**: Cada fatia da base conecta dois vértices da borda ao centro, formando slices triangulares.

**Faces Laterais**: Cada fatia do cone possui dois triângulos por stack, conectando os vértices da camada atual e da próxima.

**Topo**: Cada fatia da última stack conecta-se ao único vértice do topo, formando slices triangulares.

A base do cone situa-se no plano XZ, centrada na origem e o seu processo de criação envolve:

- **Centro da Base**:

O ponto central da base (0,0,0) é adicionado como um ponto único na variável centerIndex. Esta variável será posteriormente utilizada para a criação dos triângulos da base.

- **Cálculo dos Pontos da Circunferência da Base**:

Para cada slice, um ponto na circunferência da base é calculado com recurso a coordenadas polares:

$$\mathrm{x} = \mathrm{radius} * \cos(\mathrm{angle})$$

$$\mathrm{z} = \mathrm{radius} * \sin(\mathrm{angle})$$

- Onde `angle=2π·i` para `i=0,1,…,slices−1`.

Cada ponto é armazenado em points, e o seu respetivo índice em baseIndices.

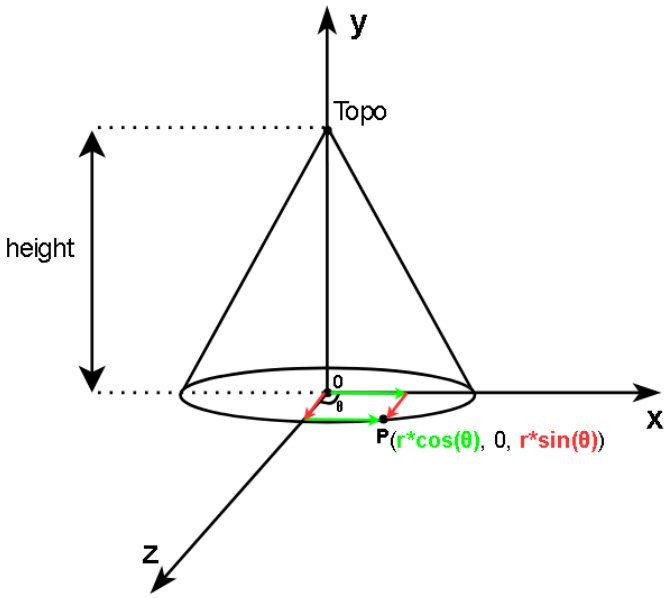


Figura 7: Ponto P na circinfurência da base do cone

**Criação dos Triângulos da Base**:

A base é preenchida com triângulos que conectam o centro da base a cada par de pontos adjacentes na circunferência.

Para cada slice, os índices do centro, do ponto atual e do próximo ponto são adicionados a indices:

$$\text{indices} = [\text{centerIndex},\ \text{baseIndices[i]},\ \text{baseIndices}[(\text{i} + 1)\ \%\ \text{slices}]]$$

Formando assim, a mesh de triângulos que preenche a base.

O corpo do cone é gerado como uma série de camadas (stacks) que conectam a base ao topo. O processo envolve os seguintes passos:

**Cálculo dos Pontos para Cada Camada**:

Para cada camada (stack), a altura (currHeight) e o raio (currRadius) são calculados utilizando interpolação linear:

$$\text{currHeight} = \frac{\text{stack}}{\text{stacks}} * \text{height}$$

$$\text{currRadius} = \text{radius} * \frac{1 - \text{stack}}{\text{stacks}}$$

Para cada slice, os pontos são calculados utilizando coordenadas polares:

$$\text{x} = \text{currRadius} * \cos(\text{angle})$$

$$\text{z} = \text{currRadius} * \sin(\text{angle})$$

Onde `angle=2π·islices` para `i=0,1,…,slices−1`.

Estes pontos são armazenados em points, e os seus índices são armazenados em stackIndices.

**Criação dos Triangulos do Corpo**:

As camadas são conectadas formando células quadradas, que são divididas em dois triângulos cada.

Para cada stack e slice, os índices dos vértices são os seguintes:

- `i1 = stackIndices[stack][slice]`
- `i2 = stackIndices[stack][(slice+1)%slices]`
- `i3 = stackIndices[stack+1][slice]`
- `i4 = stackIndices[stack+1][(slice+1)%slices]`

Dois triângulos são formados para cada célula:

- indices=[i1,i3,i4]
- indices=[i1,i4,i2]

Criando desta forma, uma mesh de triângulos que conecta as camadas.

Como referido anteriormente, o topo do cone é gerado como um único ponto, com coordenadas (0,height,0) e o seu índice é armazenado como `topIndex`. O topo é conectado aos pontos da última camada (stack) para formar os últimos triângulos.

Para cada slice, os índices do topo, do ponto atual e do próximo ponto são adicionados a indices:

$$\text{indices} = [\text{topIndex},\ \text{stackIndices}[\text{stacks}-1][(\text{slice}+1)\%\text{slices}],\ \text{stackIndices}[\text{stacks}-1][\text{slice}]]$$

, criando assim a *mesh* de triângulos que converge para o topo.

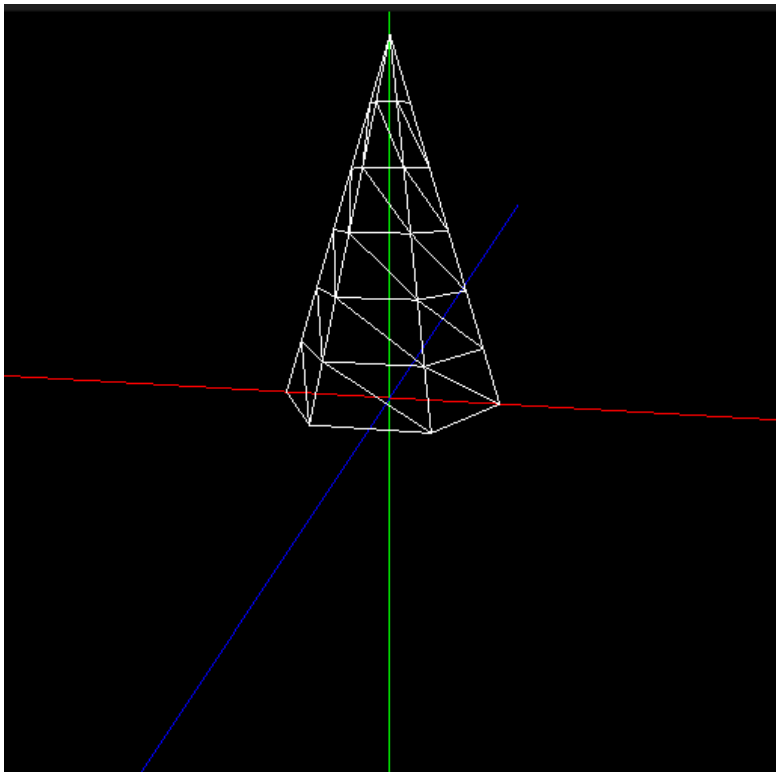

Figura 8: Cone: radius = 1, height = 3, slices = 6, stacks = 6

#### 4.1.5. Criação e Escrita nos Ficheiros .3d

Os ficheiros .3d seguem uma estrutura semelhante para todas as primitivas. Na primeira linha do ficheiro, encontra-se o número total de vértices. Nas linhas seguintes, são listados os vértices em si, sendo cada vértice representado numa linha separada e com as coordenadas separadas por vírgulas. A linha após a escrita de todos os vértices, representa a quantidade de índices que foram gerados para a construção da primitiva, seguido dos índices necessários para formar cada triângulo que compõe a primitiva. A exemplo, o índice 0,3,1 recorre aos pontos 0 (-1,0,-1) 3 (-1,0,0) e 1 (0,0,-1). A figura a seguir ilustra a organização desse formato de ficheiro, por exemplo, no plano:

home > tomas > LEI > 3ANO > 2SEM > CG > CG-24_25 > Fase1 > build > plane.3d

```
Vertices: 9
-1,0,-1
0,0,-1
1,0,-1
-1,0,0
0,0,0
1,0,0
-1,0,1
0,0,1
1,0,1
Indices: 24
0,3,1
1,3,4
1,4,2
2,4,5
3,6,4
4,6,7
4,7,5
5,7,8
```

Figura 9: Exemplo do formato de um ficheiro .3d

# 5. Engine

Além da aplicação Generator, o programa também necessita de outra componente: o Engine, um motor responsável por ler e renderizar triângulos a partir das informações contidas num ficheiro XML.

O funcionamento desta aplicação inicia-se com a leitura de um ficheiro XML, com recurso à biblioteca tinyXML. Este ficheiro é lido apenas uma vez tal como pedido no enunciado. Em seguida, o programa identifica as primitivas mencionadas no XML e acessa os ficheiros .3d correspondentes. Os dados desses ficheiros são então carregados na memória, onde permanecerão armazenados até serem desenhados.

Para organizar e guardar as diferentes primitivas em memória, foram utilizadas as classes Primitive e Point, que apresentam estruturas internas específicas para armazenar as informações necessárias. A classe Point possui variáveis de instância que representam as coordenadas de um ponto no espaço tridimensional, ou seja, armazena os valores de x, y e z de cada vértice.

Além disso, foi desenvolvida a classe Primitive, que contém um vetor de objetos da classe Point e um vetor que contém os índices para os pontos. Desta maneira, cada instância de Primitive representa uma das primitivas geométricas (como plano, caixa, esfera, cone ou cilindro), e as coordenadas dos vértices que formam os triângulos dessas primitivas são armazenadas nesse vetor.

A figura abaixo ilustra a organização das estruturas de dados utilizadas para representar e gerenciar as primitivas em memória.

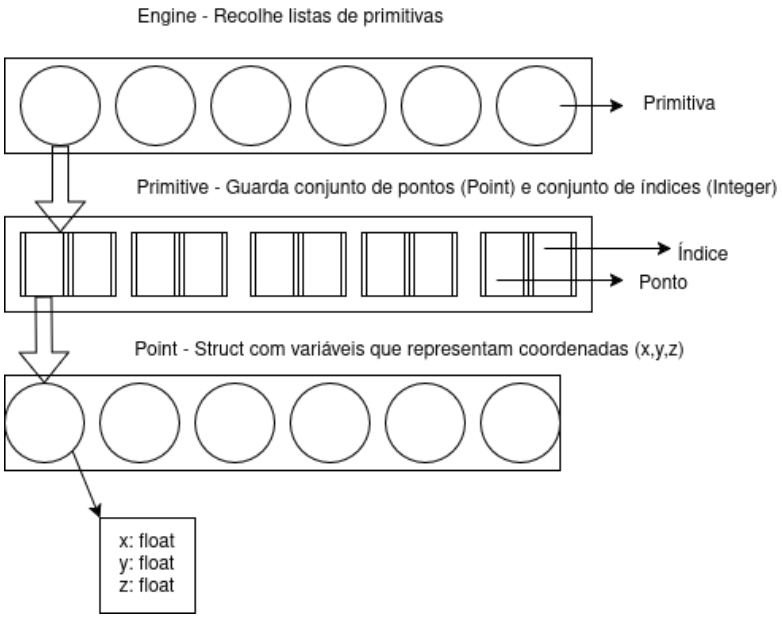


Figura 10: Estrutura da engine

## 5.1. Ficheiros XML

Além de gerar o ficheiro .3d, é necessário criar um ficheiro XML que armazene referências para esses ficheiros gerados. O ficheiro XML contém um apontador para cada ficheiro .3d, associando-o à primitiva

correspondente. Dessa forma, quando o Engine lê o ficheiro XML, ele é capaz de desenhar as primitivas especificadas.

Caso o Generator seja utilizado para criar pontos e o nome do ficheiro já exista, o ficheiro .3d será sobrescrito com as novas informações. No entanto, o XML manterá múltiplas referências ao mesmo nome de ficheiro .3d, o que resulta na sobreposição de duas primitivas idênticas ao serem desenhadas. Abaixo, é apresentado um exemplo de como um ficheiro XML gerado pode ser estruturado:


Figura 11: Exemplo de ficheiro XML

### 5.1.1. Sobreposição de Primitivas

No caso em que mais do que uma primitiva esteja contida no ficheiro XML, estas irão ser dispostas de modo a que se sobreponham como se vê na imagem seguinte:

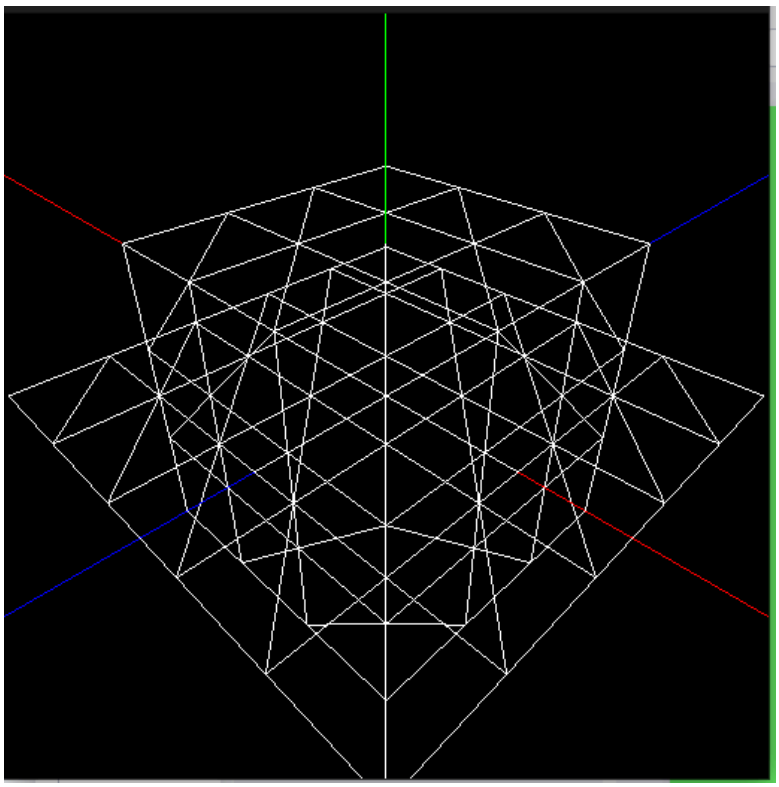

Figura 12: Sobreposição de primitivas

## 5.2. Desenho das primitivas

Na função `drawPrimitives` são usados *Vertex Arrays* para desenhar as primitivas de forma eficiente. Basicamente, cada primitiva tem uma lista de vértices e uma lista de índices, que indicam como esses vértices se ligam para formar triângulos. Em vez de repetir coordenadas, o OpenGL guarda os

vértices num array e usa os índices para definir as ligações. Primeiro, ativa-se o Vertex Array com `glEnableClientState(GL_VERTEX_ARRAY)`, depois passa-se o array de vértices para a GPU com glVertexPointer(3, GL_FLOAT, 0, vertices.data()). Para desenhar, glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, indices.data()) pega nos índices e usa-os para formar os triângulos.

## 5.3. Manipulação da câmera

A manipulação da câmera é feita com coordenadas esféricas, que permitem controlar a posição da câmera em relação ao ponto de interesse (*lookAt*). Inicialmente, os valores que definem a posição da câmera, o ponto de interesse e os parâmetros de visualização são lidos a partir de um ficheiro XML, utilizando o formato XMLDataFormat (classe que criámos para lidar com as operações associadas aos ficheiros XML). Estes valores iniciais são usados para definir a posição da câmera e o ponto de interesse ao iniciar a aplicação.

No sistema de coordenadas esféricas, a posição da câmera é representada por três parâmetros: o raio (distância entre a câmera e o ponto de interesse), o ângulo Beta, que define a inclinação da câmera em relação ao ponto de interesse, e o ângulo Alpha, que define a rotação da câmera ao redor do ponto de interesse.

### 5.3.1. Cálculo das Coordenadas Esféricas

A função computeSphericalCoordinates() calcula os três parâmetros:

- O **raio**(radius) é calculado com base na distância entre a posição atual da câmera e o ponto de interesse.
- O **ângulo Beta** é obtido através do cálculo do seno da diferença de altura entre a câmera e o ponto de interesse, normalizada pela distância (radius).
- O **ângulo Alpha** é calculado utilizando a função atan2, que determina o ângulo horizontal da câmera em relação ao ponto de interesse, baseado nas diferenças de posição ao longo dos eixos X e Z.

### 5.3.2. Atualização da Posição da Câmera

A função updateCameraPosition() atualiza a posição da câmera com base nas coordenadas esféricas. utilizando os valores de Alpha, Beta e radius calculados, a posição da câmera é atualizada com as seguintes fórmulas:

- A coordenada X da câmera é calculada utilizando o raio e os ângulos, levando em conta a rotação horizontal (Alpha) e a inclinação vertical (Beta).
- A coordenada Y é calculada com base no ângulo Beta, ajustando a altura da câmera em relação ao ponto de interesse.
- A coordenada Z é calculada com base no raio e nos ângulos Alpha e Beta, ajustando a posição da câmera no plano horizontal.

Estas fórmulas garantem que a câmera se mova de forma suave ao redor do ponto de interesse, permitindo movimentos circulares tanto ao longo do eixo horizontal quanto do eixo vertical, enquanto mantém a câmera a uma distância constante do ponto de interesse. A forma de como podemos utilizar estas funcionalidades encontra-se no Capítulo 7 deste relatório.

# 6. Utils

Esta diretoria reúne 3 ficheiros (point.cpp, primitive.cpp e XMLDataFormat.cpp) com estruturas de dados para a primitiva, ponto e ficheiro XML, possuindo funções para a formação, manipulação e extração dos dados de ficheiros xml, que serão usados pelo Engine e Generator para o seu funciomaneto.

## 6.1. Point

Para gerir os pontos foi criado um ficheiro point.cpp que define a estrutura e as funções necessárias para criar e manipular pontos no espaço 3D, utilizando variáveis float que representam coordenadas (x, y, z). Estes pontos são usados como vértices na construção das primitivas.

Além da criação de pontos iniciais, o ficheiro inclui funções para acessar as coordenadas (getX, getY, getZ) e para duplicar pontos (dupPoint) que é útil quando se precisa de cópias independentes para evitar modificações não intencionais. A função deletePoint é usada para libetar a memória alocada para um ponto.

A utilidade principal deste ficheiro no projeto é fornecer uma base sólida para a construção de primitivas gráficas uma vez que estes pontos representarão os vértices das primitivas exigidas.

## 6.2. Primitive

O ficheiro primitive.cpp define a estrutura e as funções necessárias para criar, manipular e destruir primitivas gráficas em 3D, que são compostas por conjuntos de vértices (pontos) e índices que definem a ordem dos triângulos.

A estrutura primitive contém dois vetores principais: points, que armazena os pontos (vértices) únicos da primitiva, e indices, que define a ordem dos vértices para formar triângulos. Esta organização permite reutilizar vértices compartilhados por múltiplos triângulos, economizando memória e melhorando o desempenho na renderização.

A criação de primitivas é feita através de duas funções principais. newEmptyPrimitive cria uma primitiva vazia inicializando vetores de pontos e índices vazios. Já newPrimitive permite criar uma primitiva diretamente a partir de um vetor de pontos fornecido como argumento. Ambas as funções utilizam new para alocar memória dinamicamente, permitindo que a estrutura cresça conforme necessário.

Para permitir a exportação e importação das primitivas, o ficheiro inclui as funções fromPrimitiveTo3dFile e from3dFileToPrimitive. fromPrimitiveTo3dFile escreve os pontos e índices da primitiva num ficheiro, organizando os vértices na secção Vertices e os índices na seção Indices. Por outro lado, from3dFileToPrimitive carrega uma primitiva a partir de um ficheiro, reconstruindo os pontos e índices lidos do ficheiro.

O ficheiro também contém funções para adicionar novos elementos às primitivas. A função addPrimitive combina duas primitivas, adicionando os pontos e índices de uma primitiva à outra e a addTriangle cria um triângulo a partir de três pontos fornecidos, enquanto addPoint adiciona um ponto único à primitiva. Há também a função addPoints, que adiciona múltiplos pontos e índices de outra primitiva à atual, facilitando a construção de *meshes* complexas a partir de componentes menores.

Para definir e acessar os dados armazenados, primitive.cpp inclui funções de acesso e modificação. setIndices permite definir a ordem dos índices para a primitiva. getIndices e getPoints retornam, respecti-

vamente, os vetores de índices e pontos armazenados na primitiva. Essas funções encapsulam o acesso aos dados internos, mantendo o código modular e organizado.

A destruição de primitivas é feita através de três funções diferentes: deletePrimitiveSimple, deletePrimitive e deletePrimitive2. Todas liberam a memória alocada dinamicamente, mas com variações no nível de limpeza. deletePrimitiveSimple apenas limpa os vetores e apaga a estrutura, enquanto deletePrimitive percorre todos os pontos e liberta individualmente a memória de cada ponto. deletePrimitive2 é uma versão simplificada que apenas limpa os vetores de pontos e índices.

## 6.3. XMLDataFormat

O ficheiro XMLDataFormat.cpp é responsável por ler, armazenar e manipular dados de configuração para a renderização de cenas 3D, utilizando ficheiros XML como *input*. Os dados são organizados em estruturas específicas que representam parâmetros da janela, configuração da câmera, projeções e modelos 3D a serem carregados.

```
struct Window{
    int width, height;
};

struct PosCamera{
    float cam1, cam2, cam3;
};

struct LookAt{
    float lookat1, lookat2, lookat3;
};


struct Up{
    float up1, up2, up3;
};

struct Projection{
    float fov, near, far;
};

struct XMLDataFormat {
    Window window;
    PosCamera poscamera;
    LookAt lookat;
    Up up;
    Projection projection;

    std::list<std::string> models;

};
```

Figura 13: Structs para a gestão dos ficheiros .XML

O principal objetivo deste ficheiro é facilitar a gestão de configurações de cena, permitindo que as alterações sejam feitas de forma flexível através de ficheiros XML, sem necessidade de modificar o código-fonte.

A estrutura central é XMLDataFormat, que agrupa outras estruturas menores, incluindo:

**Window**: Define as dimensões da janela (width e height).

**PosCamera**: Armazena a posição da câmera no espaço 3D (cam1, cam2, cam3).

**LookAt**: Define o ponto para onde a câmera está a olhar (lookat1, lookat2, lookat3).

**Up**: Define o vetor up da câmera para controlo de orientação (up1, up2, up3).

**Projection**: Armazena os parâmetros de projeção, como campo de visão (fov), plano próximo (near) e plano distante (far).

**models**: Uma lista de strings contendo os caminhos dos ficheiros de modelos 3D a serem carregados.

A criação e inicialização dos dados XML é feita através da função newXMLDataFormat, que aloca dinamicamente um XMLDataFormat e inicializa todos os valores com padrões fixos. Por exemplo, as dimensões

da janela são definidas como 512x512, enquanto as posições da câmera e os parâmetros de projeção são inicializados com 0.0f.

Os dados são extraídos através de funções auxiliares como buildPosCamera, buildLookAtCamera, buildUpCamera e buildProjectionCamera, que convertem os atributos XML em float com atof() e armazenam nos respetivos campos da estrutura.

Além dos parâmetros de câmera e janela, a função também lê modelos 3D a partir da secção models. Os ficheiros são especificados como atributos file em elementos model. Os caminhos desses ficheiros são armazenados na lista models dentro do XMLDataFormat, permitindo que os modelos sejam carregados e renderizados posteriormente na cena.

# 7. Funcionalidades Implementadas

Para esta fase implementámos funcionalidades na parte gráfica como o movimento da câmera, zoom-in e zoom-out, omissão de eixos, entre outras. Abaixo encontram-se todas as funcionalidades implementadas a nível da parte gráfica para melhor visualização, interpretação e compreensão das primitivas.

`W` - Rotacionar a câmera para cima

`A`- Rotacionar a câmera para a esquerda

`S` - Rotacionar a câmera para baixo

`D` - Rotacionar a câmera para a direita

`+` - Aproximar a câmera da origem (Zoom-in)

`-` - Afastar a câmera da origem (Zoom-out)

`X` - Ocultar os eixos x,y, z (Axes)

`F` - Preencher a primitiva (Fill)

`L` - Representar as linhas que constituem a primitiva (Line)

`B` - Representar os pontos que constituem a primitiva (Blank)

`Y` - Modo de cor amarela (Yellow)

## 7.1. Menu de auxílio

Para auxiliar na forma como os comandos devem ser executados com base no tipo de primitiva desejada, foi criado um simples menu de ajuda que indica os parâmetros a passar como argumento ao executar o generator, como visto na imagem seguinte:

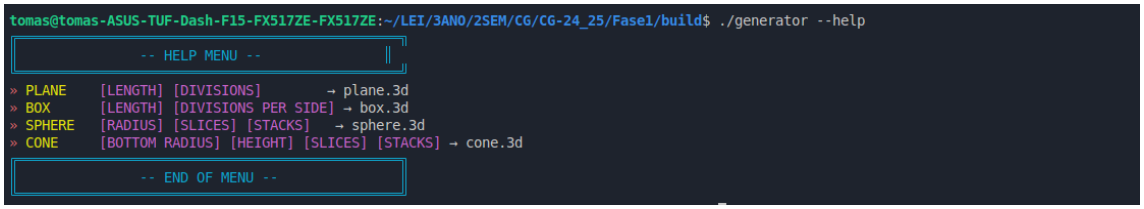


Figura 14: Help menu

## 7.2. Falha ao abrir ficheiro XML

Ainda, para garantir que não sejam referenciados ficheiros XML que não existam, é impressa uma mensagem no terminal caso o ficheiro não seja encontrado (não existir) no path referenciado ao executar o comando de execução da engine.

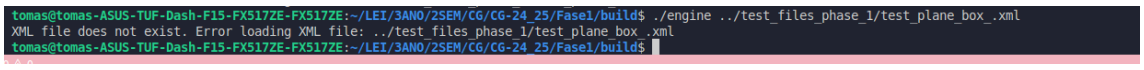


Figura 15: Ficheiro XML não existe

# 8. Forma de Executar e Outputs Gerados

## 8.1. CMake

`cmake ..` - Configura o projeto com base no CMakeLists.txt no diretório pai (..) e gera ficheiros de build para compilação.

`cmake --build .` - Compila o projeto utilizando os ficheiros de build gerados na etapa anterior e gera os executáveis.

```
tomas@tomas-ASUS-TUF-Dash-F15-FX517ZE-FX517ZE:~/LEI/3ANO/2SEM/CG/CG-24_25/Fase1/build$ cmake .. ; cmake --build .
-- Configuring done (0.0s)
-- Generating done (0.0s)
-- Build files have been written to: /home/tomas/LEI/3ANO/2SEM/CG/CG-24_25/Fase1/build
[ 35%] Built target tinyXML
[ 71%] Built target Utils_n_Build_lib
[ 85%] Built target engine
[100%] Built target generator
tomas@tomas-ASUS-TUF-Dash-F15-FX517ZE-FX517ZE:~/LEI/3ANO/2SEM/CG/CG-24_25/Fase1/build$
```

Figura 16: Configuração e compilação do projeto

## 8.2. Generator

Para criar, por exemplo, um plano de 2 unidades de comprimento e 2 subdivisões ao longo de cada eixo no plano XZ deve-se usar o comando:

```
./generator plane 2 2 plane.3d
```

A geração de primitivas, no geral, representa-se por:

```
./generator [primitive_type] [args] [primitive_type].3d
```

O tipo de argumentos varia de acordo com a primitiva. Deve-se consultar o help menu para verificar quais parâmetros cada primitiva precisa.

O output perante esta operação é semelhante entre as diferentes primitivas que são geradas.

```
tomas@tomas-ASUS-TUF-Dash-F15-FX517ZE-FX517ZE:~/LEI/3ANO/2SEM/CG/CG-24_25/Fase1/build$ ./generator sphere 1 10 10 sphere.3d
          EXECUTED SUCCESSFULLY!
» Command: sphere 1 10 10 sphere.3d
             -- END --
```

Figura 17: Geração com sucesso

## 8.3. Engine

A aplicação engine recebe um ficheiro de configuração no formato XML que contém as definições da câmera e indicação do ficheiro .3d. Para tal, no caso da esfera, deve-se executar o comando:

```
./engine ../test_files_phase_1/test_1_3.xml
```

A execução deste comando dará origem à imagem gráfica de uma esfera com raio 1, 10 slices e 10 stacks.

Os ficheiros de teste padrão de extensão XML encontram-se todos na pasta test_files_phase_1, juntamente com novos ficheiros de teste criados por elementos do grupo ao longo das últimas semanas.

O output gerado pela engine encontra-se ao longo do relatório com um exemplo para cada uma das primitivas que nos foram pedidas para desenvolver para esta primeira fase.

# 9. Conclusão

Nesta primeira fase do projeto, foi possível então consolidar e aplicar os conhecimentos adquiridos nas aulas teóricas e práticas de Computação Gráfica, aprofundando a compreensão do funcionamento do OpenGL e da linguagem C++.

Acreditamos que alcançamos todos os objetivos propostos para esta fase, nomeadamente a correta implementação do Generator e do Engine, assegurando a funcionalidade exigida e garantindo sempre o máximo de eficiência possível ao, por exemplo, não gerar pontos repetidos em nenhuma primitiva e garantir apenas a geração do número mínimo de pontos necessários para a construção de cada primitiva gŕafica e otimização na forma de como os índices são construídos para cada uma. Para além disso, foram adicionados alguns extras, como a possibilidade de interação com o sistema, permitindo a movimentação da câmara e diferentes opções de visualização das figuras que nos pode ajudar em fase futuras mesmo não sendo um requisito desta fase.

Apesar das dificuldades, conseguimos superar os obstáculos, apresentando um código robusto, com tratamento de erros, bem estruturado e organizado e funcionalidades estáveis.

Em suma, consideramos que houve um balanço positivo do trabalho realizado nesta fase. Acreditamos que conseguimos reunir todos os elementos necessários para avançar para a próxima etapa do projeto, com uma base sólida de conhecimento e experiência prática.

# Referências

OpenGL Cone & Pyramid. (s.d.). songho.ca. https://www.songho.ca/opengl/gl_cone.html

OpenGL Sphere. (s.d.). songho.ca. https://www.songho.ca/opengl/gl_sphere.html

Why projection matrices typically used with OpenGL fail with Vulkan. (s.d.). Johannes Unterguggenberger. https://johannesugb.github.io/gpu-programming/why-do-opengl-proj-matrices-fail-in-vulkan/