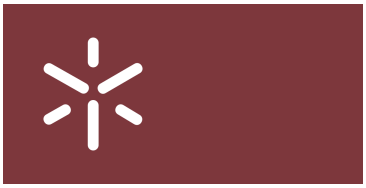

Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática
Mestrado Integrado em Engenharia Informática

Unidade Curricular de Computação Gráfica

Ano Letivo de 2024/2025

Primitivas Gráficas - Fase 1

Afonso Pedreira A104537
Fábio Magalhães A104365
André Pinto A104267

Fevereiro, 2025

CG

Índice

1. Introdução

Este relatório tem como objetivo explicar o trabalho desenvolvido na primeira fase desta UC, aplicando as técnicas apresentadas tanto nas aulas teóricas como nas aulas práticas. Nesta primeira fase, o trabalho consistiu no desenvolvimento de dois módulos:

- ***generator***: O objetivo deste módulo é gerar diferentes figuras com base no input fornecido e guardar as mesmas no ficheiro indicado pelo utilizador.
- ***engine***: O objetivo deste módulo é criar uma *engine* que carrega cenas/modelos a partir de um ficheiro e exibe no ecrã toda a informação.

2. Estrutura do Projeto

O projeto está dividido em dois programas principais: ***generator*** e ***engine***, que trabalham em conjunto para gerar e renderizar figuras gráficas. A estrutura do projeto é organizada de forma a facilitar a modularidade e a reutilização de código.

A arquitetura do projeto é composta pelas seguintes pastas principais:

- **engine**: Esta pasta contém os componentes responsáveis pela renderização das cenas. Inclui elementos essenciais como:
  - **camera**: Define a perspetiva e a posição da visualização da cena.
  - **configuration**: Contém as configurações gerais do sistema, como a `window`, as definições da câmara e os modelos a serem desenhados.
  - **window**: Responsável pela gestão dos parâmetros da janela de visualização.
  - **draw**: Contém a lógica de desenho da cena na janela.
  - **parser**: Este módulo é responsável por fazer o `parse` dos ficheiros de entrada (como `.xml`, `.obj` e `.3d`) e configurar os dados necessários para a renderização.
- **generator**: Esta pasta contém os módulos responsáveis pela geração das figuras geométricas. Cada figura é gerada a partir de parâmetros específicos e guarda num ficheiro `.3d`. Estes ficheiros, posteriormente, serão embutidos em ficheiros `.xml`, que a *engine* utiliza para renderizar as cenas.
- **common**: Esta pasta contém componentes compartilhados entre os dois programas principais. Inclui definições essenciais, como a definição de **Ponto** bem como funções auxiliares para tarefas comuns, como:
  - **salvar em ficheiro**: Funções para escrever os dados gerados em ficheiros de diferentes formatos.
  - **ler de ficheiro**: Funções para ler dados de ficheiros de entrada, como `.xml` e `.obj`.
  - **criar ficheiro 3D**: Funções que permitem a criação e manipulação de ficheiros no formato `.3d`.

A organização modular do projeto permite uma gestão eficiente de cada componente, garantindo a escalabilidade e uma boa base para as próximas fases.

2.1. Generator

A primeira fase deste projeto consistiu no desenvolvimento de uma ferramenta chamada **Generator**, que recebe parâmetros específicos como entrada para gerar diferentes primitivas gráficas. Inicialmente, eram necessárias quatro primitivas: **Caixa**, **Cone**, **Esfera** e **Cubo**. Contudo, o grupo decidiu ir além e, como desafio, optou por desenvolver ainda duas primitivas adicionais: o **Donut** e o **Cilindro**.

Os comandos utilizados pelo **Generator** seguem uma estrutura simples e intuitiva, permitindo a criação flexível das formas desejadas a partir dos parâmetros fornecidos.

- `generator plane <length> <divisions> <output file>`
- `generator sphere <radius> <slices> <stacks> <output file>`

- `generator cone <radius> <height> <slices> <stacks> <output file>`
- `generator box <length> <divisions> <output file>`
- `generator cylinder <radius> <height> <slices> <output file>`
- `generator  donut <innerRadius> <outerRadius> <slices> <stacks> <output file>`

O **generator** é responsável por produzir o conjunto de pontos para cada primitiva gráfica. Todas as primitivas são representadas por triângulos.

Cada grupo de três linhas no ficheiro de saída corresponde a um triângulo, como ilustrado pelo excerto abaixo:

-6 0 -6
-6 0 -3.6
-3.6 0 -6
-3.6 0 -6
-6 0 -3.6
-3.6 0 -3.6
-6 0 -3.6
-6 0 -1.2
-3.6 0 -3.6
-3.6 0 -3.6
-6 0 -1.2
-3.6 0 -1.2

Figura 1: Estrutura de um ficheiro .3d

Em seguida é detalhado o processo de raciocínio para a geração de cada uma das primitivas, explicando as escolhas feitas e os passos seguidos para criar as figuras desejadas:

2.1.1. Figuras

2.1.1.1. Plano

Uso: `generator plane <length> <divisions> <output file>`

Parâmetros necessários para o cálculo dos pontos:
- **Length**: A largura total do plano.
- **Divisions**: O número de divisões a ser feitas na largura do plano.

Processo de geração do plano:

1. **Cálculo do "half"**:Para garantir que o plano fique centrado na origem (0, 0, 0), é necessário dividir a largura total do plano por 2. Isso permite obter uma divisão exata da largura, assegurando que o plano se distribua de forma equilibrada em ambas as direções (positiva e negativa) ao longo dos eixos X e Z.

2. **Espaçamento dos pontos**: Para garantir que os pontos estão igualmente espaçados, a largura total do plano é dividida pelo número de divisões especificadas, obtendo assim o espaçamento correto entre os pontos.

3. **Cálculo das coordenadas dos pontos**: Como o plano está alinhado com os eixos X e Z, a coordenada Y é automaticamente deduzida, sendo sempre igual a zero. Isso ocorre porque o plano está situado no plano horizontal, onde não há variação ao longo do eixo Y. Para calcular corretamente as coordenadas nos eixos X e Z, é necessário levar em conta a posição de cada ponto dentro da divisão do plano. Ou seja, é preciso iterar tanto na direção do X, como na do eixo Z. Assim, **i** representa a travessia pelo eixo X e **j** representa a travessia pelo eixo Z. Dessa forma, para calcular os vértices de cada divisão, aplicamos as seguintes fórmulas:

   firstX = -halfLength + i * spaceBetween

firstZ = -halfLength + j * spaceBetween

secondX = firstX + spaceBetween

secondZ = firstX + spaceBetween

A figura a seguir ilustra e valida os cálculos apresentados anteriormente:

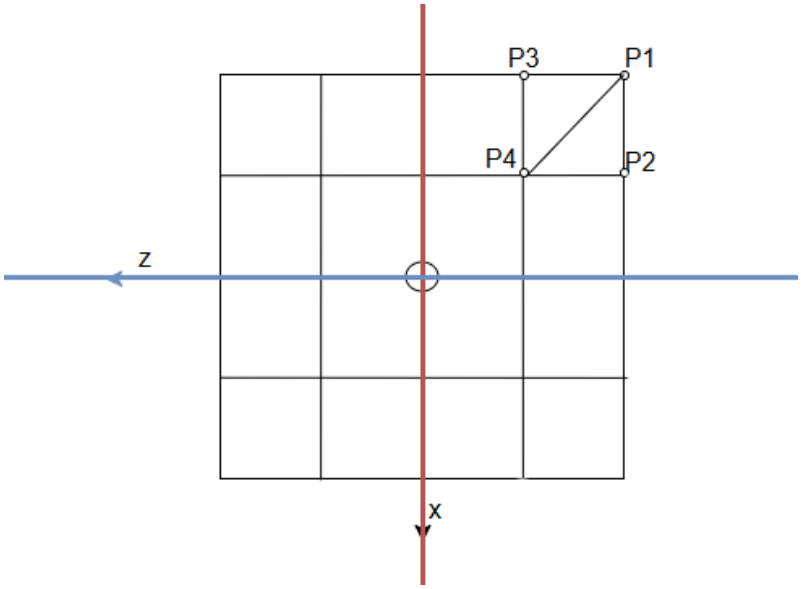

Figura 2: Funcionamento do algoritmo para a geração de um plano

A cada iteração, são gerados dois triângulos, responsáveis pela criação de cada subdivisão do plano. A ordem dos vértices é a seguinte: {P1, P4, P2} e {P1, P3, P4}.

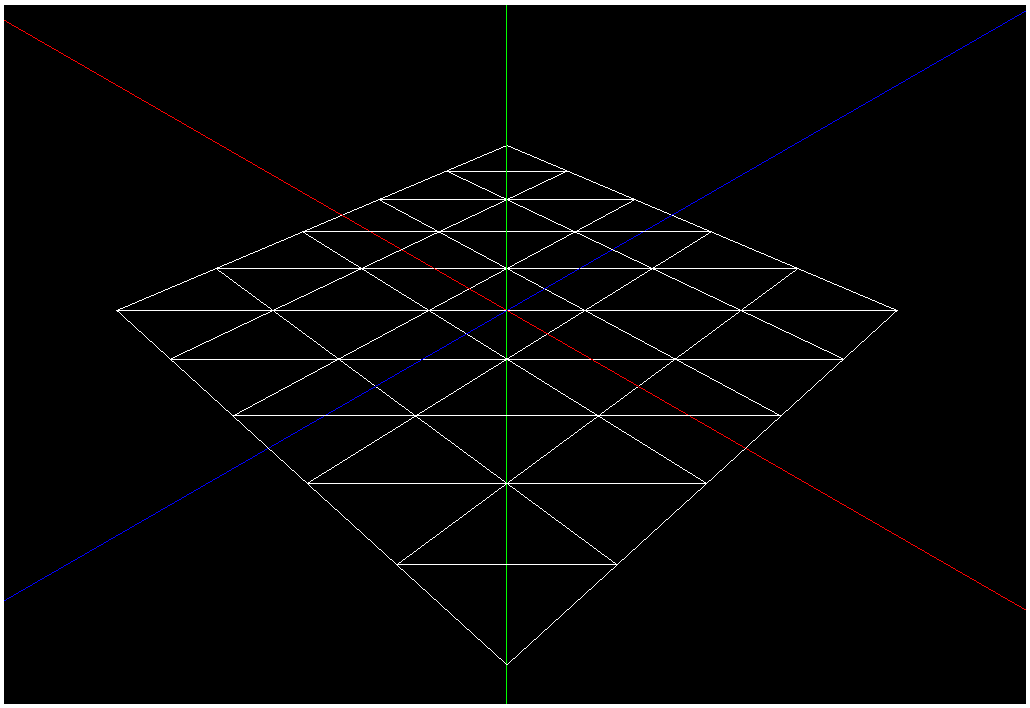

Figura 3: Plano com 5 de largura e 5 divisões

2.1.1.2. Cubo

Uso: `generator box <length> <divisions> <output file>`

Parâmetros necessários para o cálculo dos pontos:

- **Length**: A largura do cubo.
- **Divisions**: O número de divisões a ser feitas na largura do cubo.

Processo de geração do cubo: A geração do cubo, na prática, segue uma lógica muito semelhante à geração do plano mencionada anteriormente. Um cubo é composto por quatro planos de igual largura, ou seja, basta gerar quatro planos com as mesmas dimensões, posicionados em eixos diferentes.

Assim como no plano, na geração do cubo é necessário calcular a metade da largura (halfSize) e o espaçamento entre cada ponto (spaceBetween). Dessa forma, as coordenadas dos pontos são determinadas com base nos loops i e j, sendo que, para cada iteração, é calculada uma subdivisão para cada face. O valor das coordenadas é calculado da seguinte maneira:

firstCoord = -halfLength + i * spaceBetween

firstCoord2 = -halfLength + j * spaceBetween

secondCoord1 = firstCoord1 + spaceBetween

secondCoord2 = firstCoord2 + spaceBetween

positiveHeight = halfLength

negativeHeight = -halfLength

Assim, para cada face do cubo, é necessário manter uma coordenada fixa, que pode ser o valor de `positiveHeight` ou `negativeHeight`, dependendo da face em questão. As outras duas coorde-

nadas são variáveis e são definidas de maneira ordenada, com o uso das variáveis `firstCoord` e `secondCoord`. Isso permite que os pontos de cada face sejam gerados de acordo com as subdivisões feitas ao longo do cubo.

Abaixo, mostramos como as coordenadas são usadas para gerar os pontos de cada face:

Face Frontal (Z fixo) :

- Para a face frontal, a coordenada (z) é fixa em `halfSize` (valor de `positiveHeight`).
- As outras duas coordenadas, (x) e (y), são determinadas com as variáveis `firstcoord`, `firstCoord2`, `secondCoord1` e `secondCoord2`, que posteriormente definem então 2 triângulos.

P1:Ponto (firstCoord,firstCoord2,halfSize)

P2:Ponto (secondCoord1, firstCoord2,halfSize)

P3:Ponto (firstCoord,secondCoord2,halfSize)

P4:Ponto (secondCoord1,secondCoord2,halfSize)

Face Traseira (Z fixo) :

- Para a face traseira, a coordenada (z) é fixa em `-halfSize` (valor de `negativeHeight`).
- As outras coordenadas, (x) e (y), seguem o mesmo raciocínio da face frontal, mas com valores diferentes para os vértices.

Este processo de geração de pontos é repetido para as outras faces do cubo, variando as coordenadas fixas (em (z), (x) ou (y)) e ajustando as variáveis conforme necessário para formar a geometria do cubo.

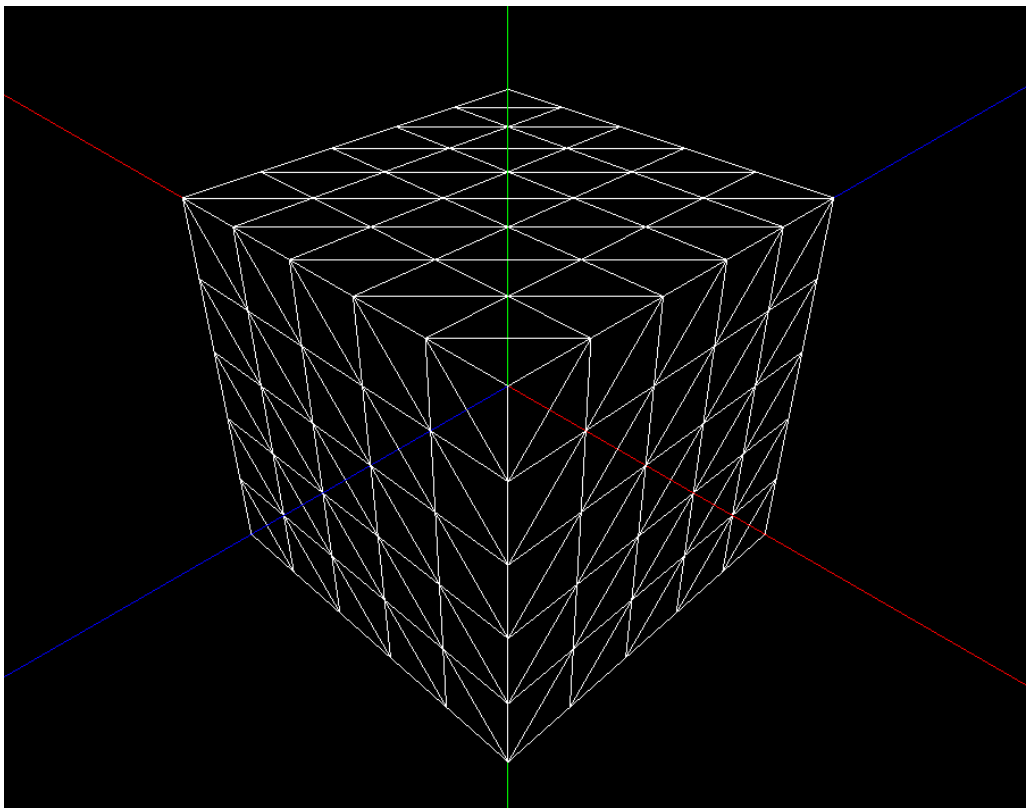

Figura 4: Cubo com 4 de largura e 5 divisões

2.1.1.3. Cone

Uso: `cone <radius> <height> <slices> <stacks> <output file>`

Parâmetros necessários para o cálculo dos pontos:

- **Ângulo de cada `slice`**

  Cada fatia (**slice**) do cone representa uma seção do círculo da base. O ângulo correspondente para cada `slice` é dado por:

$$\theta = \frac{2 * \pi}{\text{slices}}$$

  onde:

  - θ é o ângulo de cada slice,
  - slices é o número total de fatias que compõem a base.

  Esse ângulo é utilizado para calcular a posição dos pontos ao redor da base do cone.

- **Altura de cada `stack`**

  O cone é dividido horizontalmente em camadas (`stacks`). Cada uma dessas camadas possui uma altura constante, dada por:

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

  onde:

  - stackHeight é a altura definida para cada `stack`.
  - height é a altura do cone.
  - stacks é o número de fatias horizontais pretendidas no cone.

Tendo estes dois valores calculados, podemos então avançar para o algoritmo que, para cada `slice`, calcula os pontos das `stacks` ao longo da altura do cone.

2.1.1.3.1. Funcionamento do Algoritmo

Como mencionado anteriormente, este algoritmo é responsável por gerar, de forma sequencial, os pontos ao longo da altura do cone para cada `slice`. A cada iteração, é necessário calcular os seguintes valores, onde `i` é representado por **sliceAtual** e `j` representado por **stackAtual**.

- **Cálculo do Raio para Cada Stack**

O raio do cone diminui progressivamente à medida que subimos ao longo das stacks. Para calcular o raio em cada nível, utilizamos:

$\text{currentRadius} = \text{radius} - \left(\text{stackAtual} * \frac{\text{radius}}{\text{stacks}}\right)$

- **Cálculo das Coordenadas (x, y, z)**

Cada ponto no cone é definido pelas seguintes equações:

$x = \text{currentRadius} * \sin(\text{sliceAtual} * \theta)$

$y = \text{stackAtual} * \text{stackHeight}$

$z = \text{currentRadius} * \cos(\text{sliceAtual} * \theta)$

onde: x e z são as coordenadas no plano horizontal, y representa a altura do ponto, CurrentRadius é o raio calculado para aquela `stack`, $\text{sliceAtual} * \theta$ é o ângulo correspondente à `slice` atual.

Por fim, para facilitar a geração dos triângulos, também são calculados os dois pontos correspondentes à próxima `slice` e à próxima `stack`. Desta forma, é possível definir dois triângulos para representar parte da `slice` usando os pontos calculados.

```
for (int slice = 0; slice < slices; ++slice) {
    for (int stack = 0; stack < stacks; ++stack) {
      const float currentRadius = radius - stack * radius / stacks;
      const float nextRadius = radius - (stack + 1) * radius / stacks;

      const Point bottomLeft = Point(
          currentRadius * sin(slice * angleIncrement), stack * stackHeight,
          currentRadius * cos(slice * angleIncrement));
      const Point bottomRight =
          Point(currentRadius * sin((slice + 1) * angleIncrement),
                stack * stackHeight,
                currentRadius * cos((slice + 1) * angleIncrement));
      const Point topLeft = Point(nextRadius * sin(slice * angleIncrement),
                                  (stack + 1) * stackHeight,
                                  nextRadius * cos(slice * angleIncrement));
      const Point topRight =
          Point(nextRadius * sin((slice + 1) * angleIncrement),
                (stack + 1) * stackHeight,
                nextRadius * cos((slice + 1) * angleIncrement));

      vertices.push_back(topLeft);
      vertices.push_back(bottomLeft);
      vertices.push_back(bottomRight);

      vertices.push_back(topLeft);
      vertices.push_back(bottomRight);
      vertices.push_back(topRight);
    }
}
```

Listing 1: Algoritmo de geração dos pontos do cone

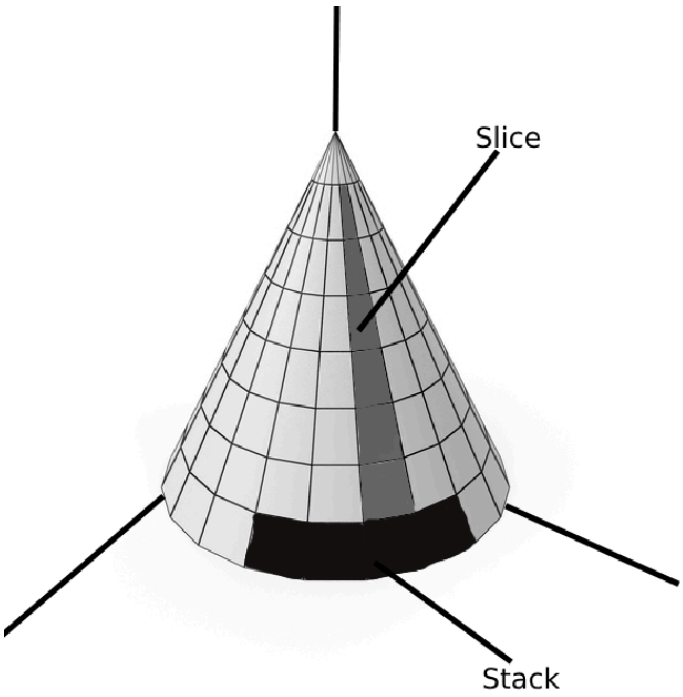

Figura 5: Slice e Stack em um cone

Assim, após a execução do algoritmo para a geração do cone, o resultado obtido é ilustrado na figura abaixo. O cone gerado possui um raio de 2 unidades, uma altura de 4 unidades, com 15 `slices` e 5 `stacks`, conforme especificado:

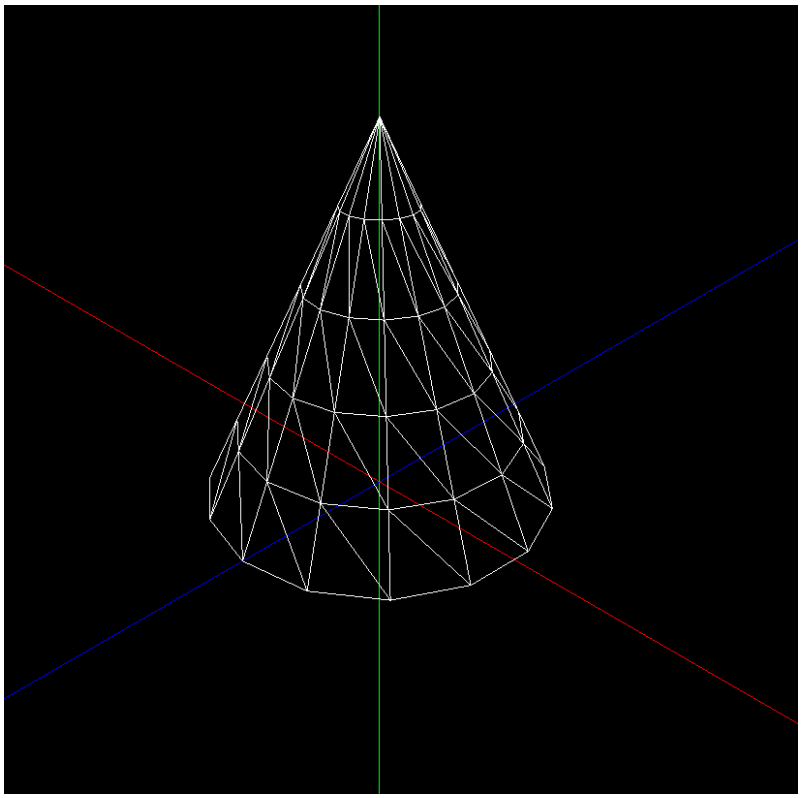

Figura 6: Cone gerado com raio 2, altura 4, 15 slices e 5 stacks.

2.1.1.4. Esfera

Uso: `sphere <radius> <slices> <stacks> <output file>`

O cálculo dos pontos da esfera é semelhante ao do cone, mas com uma diferença essencial: a presença de um ângulo adicional, que decorre das propriedades geométricas da esfera. Esse ângulo permite determinar corretamente as coordenadas cartesianas de cada ponto na sua superfície.

Parâmetros necessários para o cálculo dos pontos da esfera

- **Ângulo de Cada Slice**

  Cada slice representa uma divisão vertical da esfera e o seu ângulo é dado por:

  $$\theta = \text{sliceAtual} * \frac{\pi}{\text{slices}}$$

  onde:

  $\boldsymbol{\theta}$ é o ângulo de cada slice que define a altura dos pontos gerados ao longo da esfera.

  slices é o número total de fatias verticais que compõem a esfera.

- **Ângulo de Cada Stack**

  Cada stack representa uma divisão horizontal da esfera, com o seu ângulo definido por:

  $$\varphi = \text{stackAtual} * \frac{2\pi}{\text{stacks}}$$

  onde: φ é o ângulo de cada stack,que permite determinar a distribuição dos pontos ao longo da superfície da esfera.

stacks é o número total de divisões horizontais da esfera.

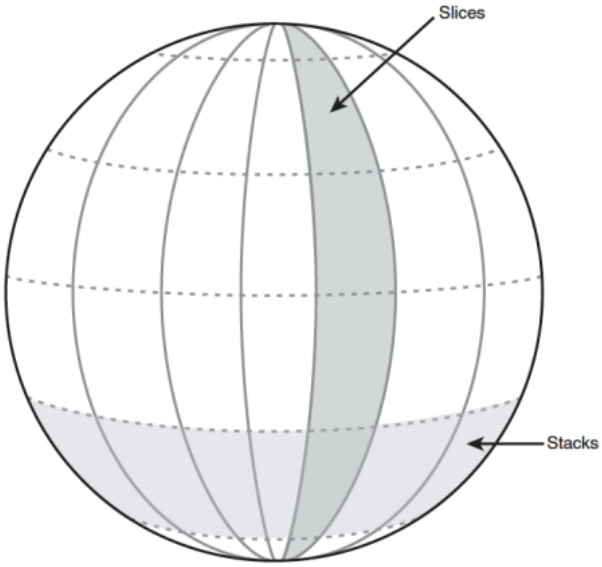

Figura 7: Slices e Stacks em uma circunferência

2.1.1.4.1. Funcionamento do Algoritmo

Este algoritmo é responsável por preencher, para cada secção de uma `slice`, a respetiva `stack` com pontos ao longo da fatia horizontal.

Assim, para cada iteração responsável por incrementar $\boldsymbol{\theta}$, é também incrementado $\boldsymbol{\varphi}$, de modo a gerar todos os pontos da `stack` correspondente, ou seja, da fatia horizontal onde nos encontramos.

- **Cálculo das Coordenadas $(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$**

Para calcular os pontos ao longo da superfície da esfera, utilizamos o sistema de **coordenadas esféricas**, que nos permite converter os ângulos θ e φ em coordenadas cartesianas (x, y, z).

As equações são dadas por:

$$x = \text{radius} * \sin(\theta) * \sin(\varphi)$$

$$y = \text{radius} * \cos(\theta)$$

$$z = \text{radius} * \sin(\theta) * \cos(\varphi)$$

onde:

- $\boldsymbol{x}$ e $\boldsymbol{z}$ representam as coordenadas no plano horizontal,
- $\boldsymbol{y}$ representa a altura do ponto,
- $\boldsymbol{\theta}$ é o **ângulo polar**, que varia de 0 a π e controla a latitude do ponto,
- $\boldsymbol{\varphi}$ é o **ângulo azimutal**, que varia de 0 a 2π e controla a longitude,
- **radius** é o raio da esfera.

As coordenadas esféricas são especialmente úteis para gerar pontos distribuídos uniformemente na superfície de uma esfera, pois permitem a definição precisa da posição de cada ponto apenas com base nos dois ângulos mencionados.

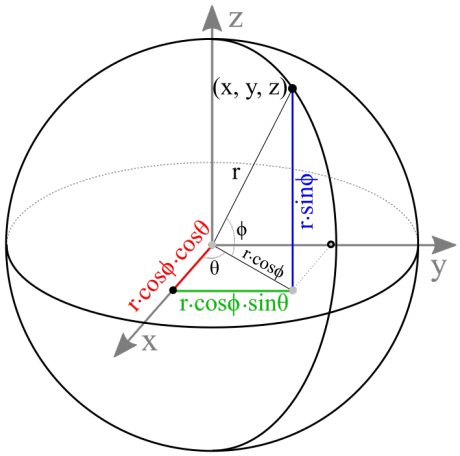

Figura 8: Sistema de coordenadas esféricas

Por fim, à semelhança do que foi feito para o cone, também são calculados antecipadamente os pontos correspondentes à próxima `stack` e à próxima `slice`. Desta forma, é possível formar dois triângulos consecutivos, facilitando a construção da estrutura da esfera.

```
for (int slice = 0; slice < slices; ++slice) {
   float theta1 = static_cast<float>(slice) * static_cast<float>(M_PI) /
                  static_cast<float>(slices);
   float theta2 = static_cast<float>(slice + 1) * static_cast<float>(M_PI) /
                  static_cast<float>(slices);

   for (int stack = 0; stack < stacks; ++stack) {
     float phi1 = static_cast<float>(stack) * 2.0f * static_cast<float>(M_PI) /
                  static_cast<float>(stacks);
     float phi2 = static_cast<float>(stack + 1) * 2.0f *
                  static_cast<float>(M_PI) / static_cast<float>(stacks);

     float z1 = radius * std::sin(theta1) * std::cos(phi1);
     float x1 = radius * std::sin(theta1) * std::sin(phi1);
     float y1 = radius * std::cos(theta1);

     float z2 = radius * std::sin(theta1) * std::cos(phi2);
     float x2 = radius * std::sin(theta1) * std::sin(phi2);
     float y2 = radius * std::cos(theta1);

     float z3 = radius * std::sin(theta2) * std::cos(phi1);
     float x3 = radius * std::sin(theta2) * std::sin(phi1);
     float y3 = radius * std::cos(theta2);

     float z4 = radius * std::sin(theta2) * std::cos(phi2);
     float x4 = radius * std::sin(theta2) * std::sin(phi2);
     float y4 = radius * std::cos(theta2);

     vertices.push_back(Point(x1, y1, z1));
     vertices.push_back(Point(x4, y4, z4));
     vertices.push_back(Point(x2, y2, z2));


     vertices.push_back(Point(x1, y1, z1));
     vertices.push_back(Point(x3, y3, z3));
     vertices.push_back(Point(x4, y4, z4));
   }
}
```

Listing 2: Algoritmo de geração dos pontos da esfera

Assim, após a execução do algoritmo para a geração dos pontos da esfera, o resultado obtido é ilustrado na figura abaixo. A esfera gerada possui um raio de 3 unidades, com 20 `slices` e 20 `stacks`, conforme especificado:

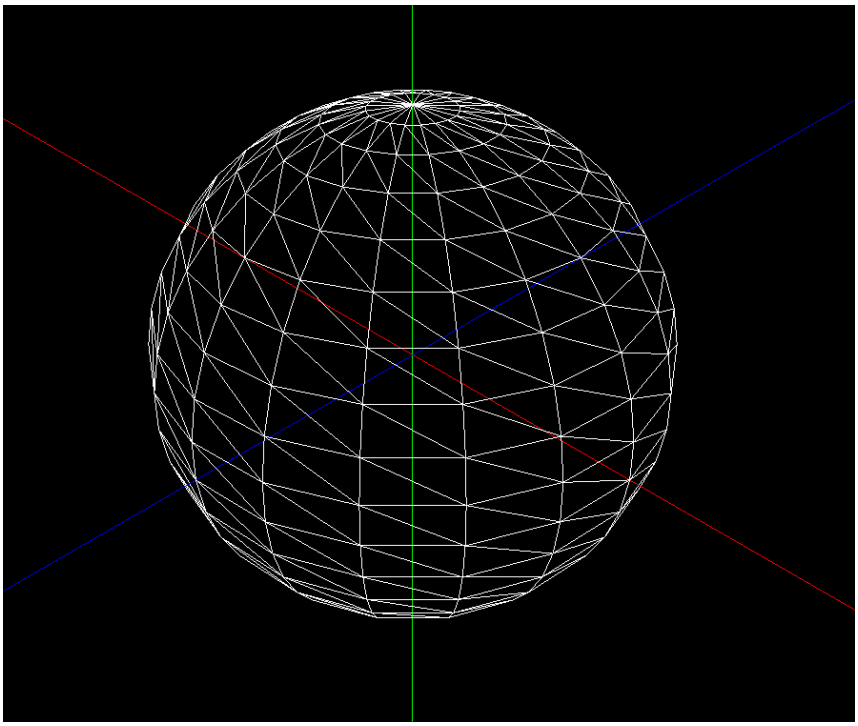

Figura 9: Esfera gerada com raio 3, 20 slices e 20 stacks.

2.1.1.5. Cilindro

Uso: `cylinder <radius> <height> <slices> <output file>`

O cálculo dos pontos do cilindro é relativamente simples, pois estamos a lidar com um objeto de simetria circular e reto, ao contrário da esfera, que tem curvaturas tanto na vertical quanto na horizontal. Para o cilindro, temos basicamente três partes principais a considerar: a base, a parte superior e a face lateral que conecta essas duas partes.

Parâmetros necessários para o cálculo dos pontos do cilindro

- **Ângulo de Cada Slice** Cada slice representa uma divisão da base do cilindro, e o seu ângulo é dado por:

$$\theta = \text{sliceAtual} * \frac{2\pi}{\text{slices}}$$

  onde:

  $\boldsymbol{\theta}$ é o ângulo de cada slice que define a posição dos pontos ao longo da base do cilindro.

  slices é o número total de fatias que dividem ambas as bases.

  A base do cilindro é formada pelas coordenadas no plano $\boldsymbol{y = -\frac{\text{height}}{2}}$, e a parte superior fica em $\boldsymbol{y = \frac{\text{height}}{2}}$.

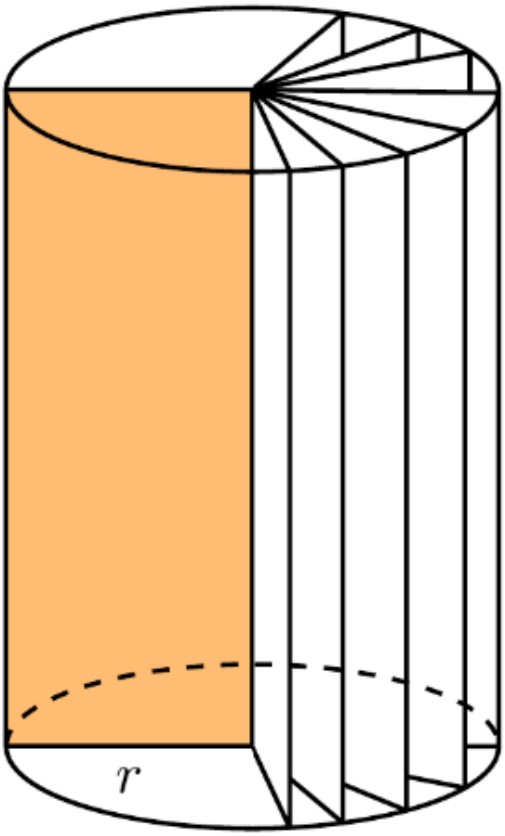

Figura 10: Slices e a disposição dos pontos no cilindro

2.1.1.5.1. Funcionamento do Algoritmo

Este algoritmo preenche as coordenadas dos pontos para cada `slice` na base, além de calcular os pontos para as faces laterais e a parte superior do cilindro. O cálculo dos pontos para cada `slice` envolve a coordenação de duas posições principais: a base inferior e superior do cilindro.

- **Cálculo das Coordenadas (x, y, z)** Para calcular os pontos ao longo da superfície do cilindro, utilizamos um sistema de coordenadas cartesianas, onde a coordenada y varia de $-\frac{\mathbf{height}}{\mathbf{2}}$ para $\frac{\mathbf{height}}{\mathbf{2}}$, e as coordenadas $\boldsymbol{x}$ e $\boldsymbol{z}$ são determinadas pelo ângulo $\boldsymbol{\theta}$.

As equações para as coordenadas cartesianas são:

$$\boldsymbol{x} = \mathbf{radius} * \mathbf{sin}(\boldsymbol{\theta})$$

$$\boldsymbol{z} = \mathbf{radius} * \mathbf{cos}(\boldsymbol{\theta})$$

Estas coordenadas permitem distribuir os pontos ao longo das duas partes do cilindro (base e topo) e também formar a face lateral conectando ambos.

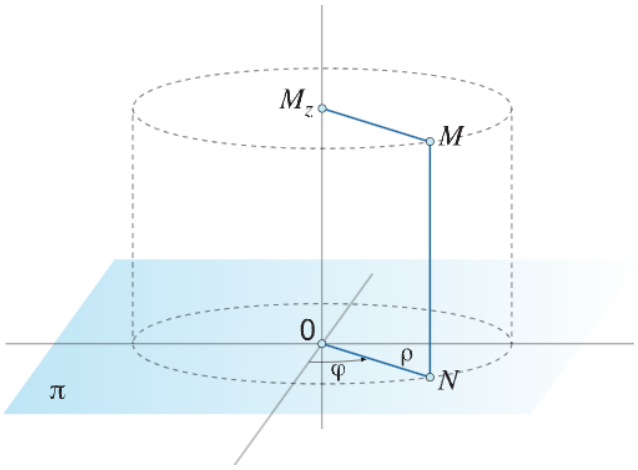

Figura 11: Representação do sistema de coordenadas cilíndricas

Após o cálculo das coordenadas para as bases e a face lateral, o algoritmo vai gerar os triângulos para formar a estrutura do cilindro.

```
 for (int i = 0; i < slices; i++) {

float valueX = radius * sin(newAngle);
float valueZ = radius * cos(newAngle);
vertex.push_back(Point( valueX, -halfHeight,  valueZ));
vertex.push_back(Point(0, -halfHeight, 0));
newAngle += alpha;
valueX = radius * sin(newAngle);
valueZ = radius * cos(newAngle);
vertex.push_back(Point( valueX, -halfHeight,  valueZ));

vertex.push_back(Point( valueX, -halfHeight, valueZ));
vertex.push_back(Point( valueX, halfHeight, valueZ));
newAngle -= alpha;
valueX = radius * sin(newAngle);
valueZ = radius * cos(newAngle);
vertex.push_back(Point( valueX, halfHeight,  valueZ));

vertex.push_back(Point( valueX, halfHeight, valueZ));
vertex.push_back(Point( valueX, -halfHeight,  valueZ));
newAngle += alpha;
valueX = radius * sin(newAngle);
valueZ = radius * cos(newAngle);
vertex.push_back(Point( valueX, -halfHeight,  valueZ));

newAngle += alpha;
valueX = radius * sin(newAngle);
valueZ = radius * cos(newAngle);
vertex.push_back(Point( valueX, halfHeight,  valueZ));
vertex.push_back(Point(0, halfHeight, 0));
newAngle-= alpha;
valueX= radius * sin(newAngle);
valueZ = radius * cos(newAngle);
vertex.push_back(Point( valueX, halfHeight, valueZ));
}
```

Assim, após a execução do algoritmo para a geração dos pontos do cilindro, o resultado obtido é ilustrado na figura abaixo. O cilindro gerado possui um raio de 2 unidades, uma altura de 4 unidades e com 10 `slices`, conforme especificado:

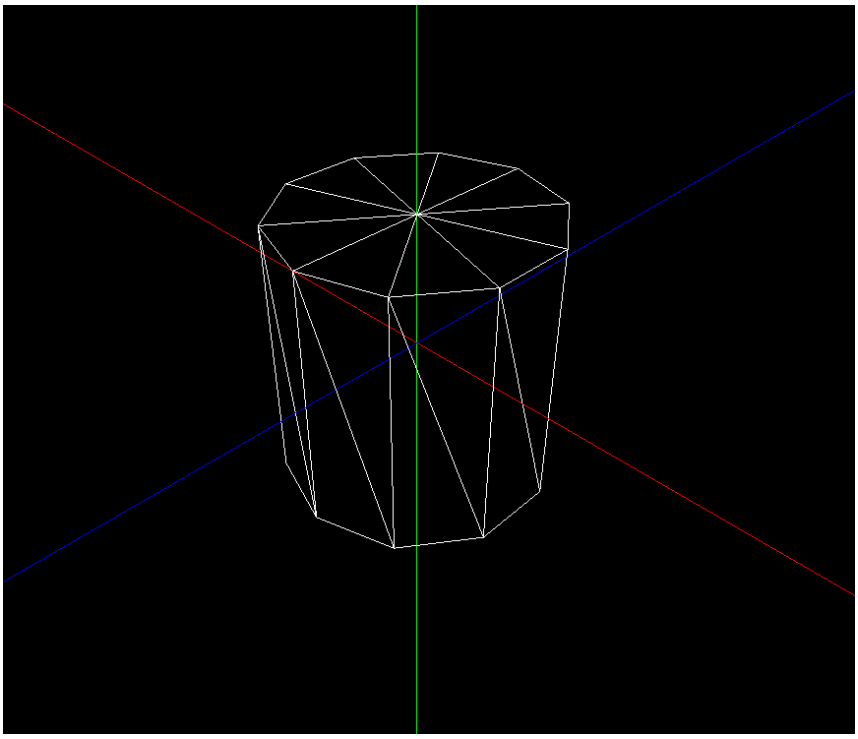

Figura 12: Cilindro gerado com raio 2, 4 de altura e 10 slices.

2.1.1.6. Torus (Donut)

Uso: `generator donut <outerRadius> <innerRadius> <slices> <stacks> <output file>`

O cálculo dos pontos para um torus (ou "donut") envolve a consideração de dois raios: o raio interno e o raio externo. O torus pode ser visualizado como um anel ou "donut", onde o raio externo é a distância do centro do torus até o centro do tubo que o forma, e o raio interno é o raio do próprio tubo.

Parâmetros necessários para o cálculo dos pontos do torus (donut)

- **Raio Interno e Raio Externo**: O raio externo define a forma do "anél" do torus, enquanto o raio interno define a espessura do tubo.

  O torus é formado por uma série de "fatias" (slices) ao longo do anel, e "stacks" que formam o tubo ao redor do anel.

  - **Raio Externo**: A distância do centro do torus até o centro do tubo.
  - **Raio Interno**: O raio do tubo do torus.
  - **Slices**: O número de divisões ao longo do anel do torus.
  - **Stacks**: O número de divisões ao longo do tubo do torus.

  O cálculo dos ângulos para cada slice e stack é feito da seguinte forma:

$$\text{anguloAnel} = \text{stackAtual} * \frac{2\pi}{\text{stacks}}$$

$$\text{anguloTubo} = \text{sliceAtual} * \frac{2\pi}{\text{slices}}$$

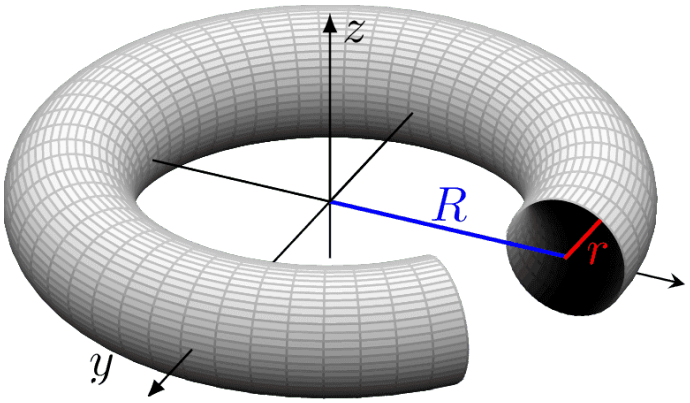

Figura 13: Slices e a disposição dos pontos no Torus

2.1.1.6.1. Funcionamento do Algoritmo

Este algoritmo preenche as coordenadas dos pontos para cada `slice`(lado do tubo) e `stack` (anéis principais do `torus`). Calcula as coordenadas para o `torus` num sistema de coordenadas cartesianas, onde os ângulos dos anéis e dos lados definem a posição dos pontos.

- **Cálculo das Coordenadas (x, y, z)** Para calcular os pontos ao longo do `torus`, utilizamos as equações paramétricas. O raio externo é o parâmetro principal, enquanto o raio interno define a curvatura do tubo.

As equações para as coordenadas cartesianas são:

$$x = \text{outerRadius} + \text{innerRadius} * \cos(\text{sliceAngle}) * \cos(\text{stackAngle})$$

$$y = \text{innerRadius} * \sin(\text{sliceAngle})$$

$$z = \text{outerRadius} + \text{innerRadius} * \cos(\text{sliceAngle}) * \sin(\text{stackAngle})$$

Essas equações geram os pontos ao redor dos dois anéis que formam a estrutura do `torus`.

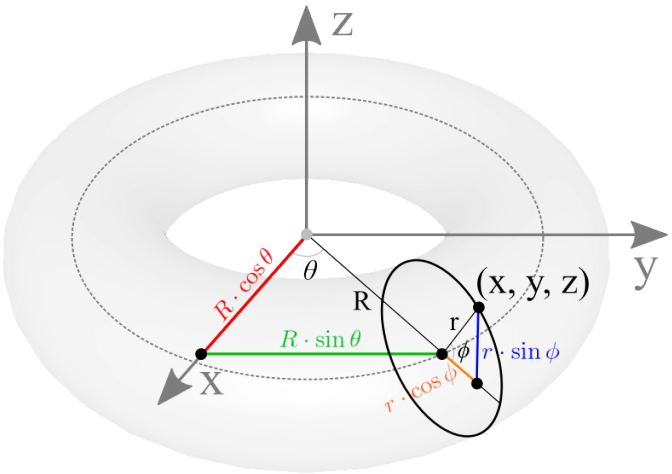

Figura 14: Representação do sistema de coordenadas cilíndricas do Torus (Donut)

Após o cálculo das coordenadas para os lados e os anéis, o algoritmo gera os triângulos que formam a superfície do `torus`.

```
float ringStep = 2.0f * static_cast<float>(M_PI) / numRings;
float tubeStep = 2.0f * static_cast<float>(M_PI) / numSides;

  for (int ring = 0; ring < numRings; ++ring) {
```

```
float ringAngle1 = ring * ringStep;
float ringAngle2 = (ring + 1) * ringStep;

float cosRing1 = cosf(ringAngle1), sinRing1 = sinf(ringAngle1);
float cosRing2 = cosf(ringAngle2), sinRing2 = sinf(ringAngle2);

for (int side = 0; side < numSides; ++side) {
  float tubeAngle1 = side * tubeStep;
  float tubeAngle2 = (side + 1) * tubeStep;

  float cosTube1 = cosf(tubeAngle1), sinTube1 = sinf(tubeAngle1);
  float cosTube2 = cosf(tubeAngle2), sinTube2 = sinf(tubeAngle2);

  float vx1 = (majorRadius + minorRadius * cosTube1) * cosRing1;
  float vy1 = minorRadius * sinTube1;
  float vz1 = (majorRadius + minorRadius * cosTube1) * sinRing1;

  float vx2 = (majorRadius + minorRadius * cosTube2) * cosRing1;
  float vy2 = minorRadius * sinTube2;
  float vz2 = (majorRadius + minorRadius * cosTube2) * sinRing1;

  float vx3 = (majorRadius + minorRadius * cosTube1) * cosRing2;
  float vy3 = minorRadius * sinTube1;
  float vz3 = (majorRadius + minorRadius * cosTube1) * sinRing2;

  float vx4 = (majorRadius + minorRadius * cosTube2) * cosRing2;
  float vy4 = minorRadius * sinTube2;
  float vz4 = (majorRadius + minorRadius * cosTube2) * sinRing2;

  vertices.push_back(Point(vx1, vy1, vz1));
  vertices.push_back(Point(vx2, vy2, vz2));
  vertices.push_back(Point(vx4, vy4, vz4));

  vertices.push_back(Point(vx1, vy1, vz1));
  vertices.push_back(Point(vx4, vy4, vz4));
  vertices.push_back(Point(vx3, vy3, vz3));
}
```

Assim, após a execução do algoritmo para a geração dos pontos do `Donut`, o resultado obtido é ilustrado na figura abaixo. O `Torus`gerado possui um raio maior de 3, raio menor de 1, 15 slices e 15 stacks.

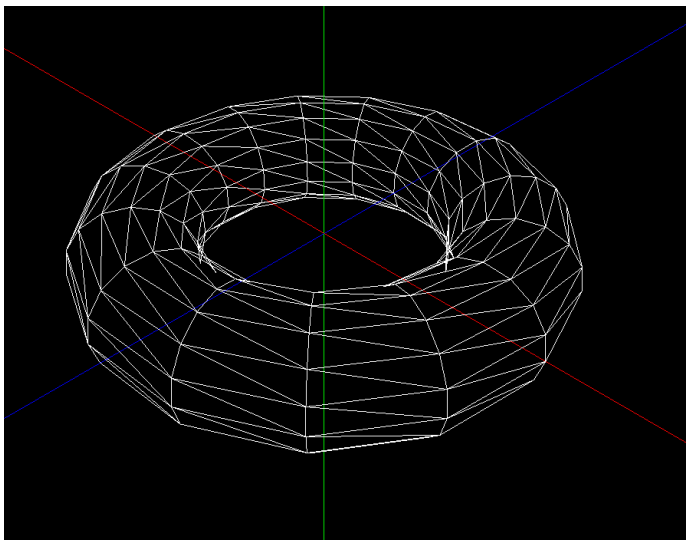

Figura 15: Geração de um Donut com raio maior de 3, raio menor de 1, 15 slices e 15 stacks.

2.2. Engine

Nesta fase inicial, foi também desenvolvido um **engine** capaz de interpretar tanto ficheiros de modelos 3D (.3d, .obj) quanto ficheiros de configuração de cenas (.xml). O objetivo principal deste **engine** é processar estas entradas e produzir a respetiva saída gráfica, ou seja, desenhar os modelos na janela de visualização de acordo com as definições presentes nos ficheiros.

A seguir, é detalhado o funcionamento deste mecanismo, incluindo a leitura dos ficheiros, o `parsing` das configurações e a criação das estruturas fundamentais para a renderização.

2.2.1. Parsing de Ficheiros .3d e .obj

O **engine** desenvolvido necessita de interpretar diferentes tipos de ficheiros para carregar modelos 3D na cena. Entre esses formatos, destacam-se os ficheiros `.3d`, utilizados para armazenar coordenadas de vértices de modelos gerados pelo **generator**, e os ficheiros `.obj`, um formato mais complexo que pode conter informações sobre vértices, normais e faces.

2.2.1.1. Parsing de Ficheiros .3d

O formato `.3d` contém apenas uma lista de vértices representados por coordenadas `(x, y, z)`. O processo de `parsing` é relativamente simples:

1. O ficheiro é aberto e lido linha por linha.
2. Cada linha contém três valores numéricos correspondentes às coordenadas de um ponto.
3. Esses pontos são armazenados num vetor para posterior utilização no `rendering` do modelo.

2.2.1.2. Parsing de Ficheiros .obj

O formato `.obj` é mais complexo, pois contém definições de vértices (v) e de faces (f). O algoritmo de `parsing` funciona da seguinte forma:

1. O ficheiro é lido linha por linha.
2. Se a linha começar com v, significa que contém as coordenadas de um vértice, que é armazenado num vetor de vértices.
3. Se a linha começar com f, significa que contém referências a vértices que formam uma face.
4. Cada índice da face refere-se a um vértice já lido, e a face é reconstruída ordenadamente.

2.2.2. Parsing de Ficheiros XML com RapidXML

Para realizar a leitura e interpretação dos ficheiros `XML`, foi utilizada a biblioteca **`RapidXML`** , uma biblioteca altamente eficiente para manipulação de ficheiros `XML`. Esta permite um `parsing` eficiente, extraindo a informação necessária e estruturando-a de forma acessível.

Durante o processo de `parsing`, os dados do ficheiro `XML` são convertidos para uma estrutura de dados interna, composta por um conjunto de classes que encapsulam e organizam as informações essenciais para a renderização da cena.

A estrutura extraída inclui três componentes principais:

1. **Configuração da Janela**: Os valores dos atributos `width` e `height` são convertidos e armazenados na estrutura **`Window`**.
2. **Configuração da Câmara**: Os nós `position`, `lookAt`, `up` e `projection` são lidos e armazenados na estrutura **`Camera`**, contendo a posição, orientação e parâmetros de projeção.
3. **Leitura dos Modelos**: A estrutura **`Configuration`** também guarda toda a informação referente aos pontos que devem ser desenhados de cada modelo.

```
<world>
  <window width="512" height="512"/>
  <camera>
    <position x="3" y="3" z="3"/>
    <lookAt x="0" y="0" z="0"/>
    <!--  gluLookAt center -->
    <up x="0" y="1" z="0"/>
    <!--  gluLookAt up -->
    <projection fov="60" near="1" far="1000"/>
    <!-- gluPerspective -->
  </camera>
  <group>
    <models>
      <model file="box.3d"/>
    </models>
  </group>
</world>
```

Listing 3: Estrutura de um ficheiro XML

2.2.3. Testes Realizados

Após o desenvolvimento do **engine**, foi realizada uma série de testes disponibilizados pelos docentes para validar a correta interpretação e renderização dos modelos. Estes testes podem ser encontrados na pasta `scenes/testes`. Em seguida, serão apresentados os resultados.

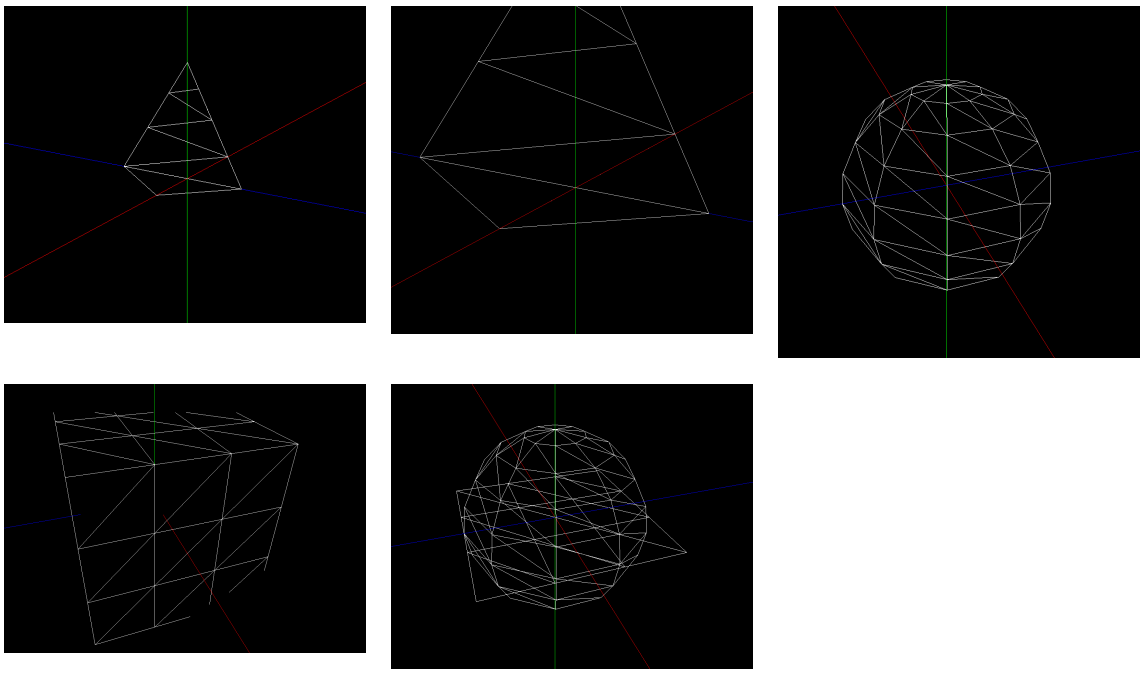

Figura 16: Resultados dos Testes disponibilizados pelos docentes

Adicionalmente, foram realizados testes utilizando ficheiros `.obj` para avaliar a compatibilidade do `parser` com diferentes estruturas e níveis de complexidade dos modelos 3D.Os ficheiros utilizados podem ser encontrados na pasta `models`.

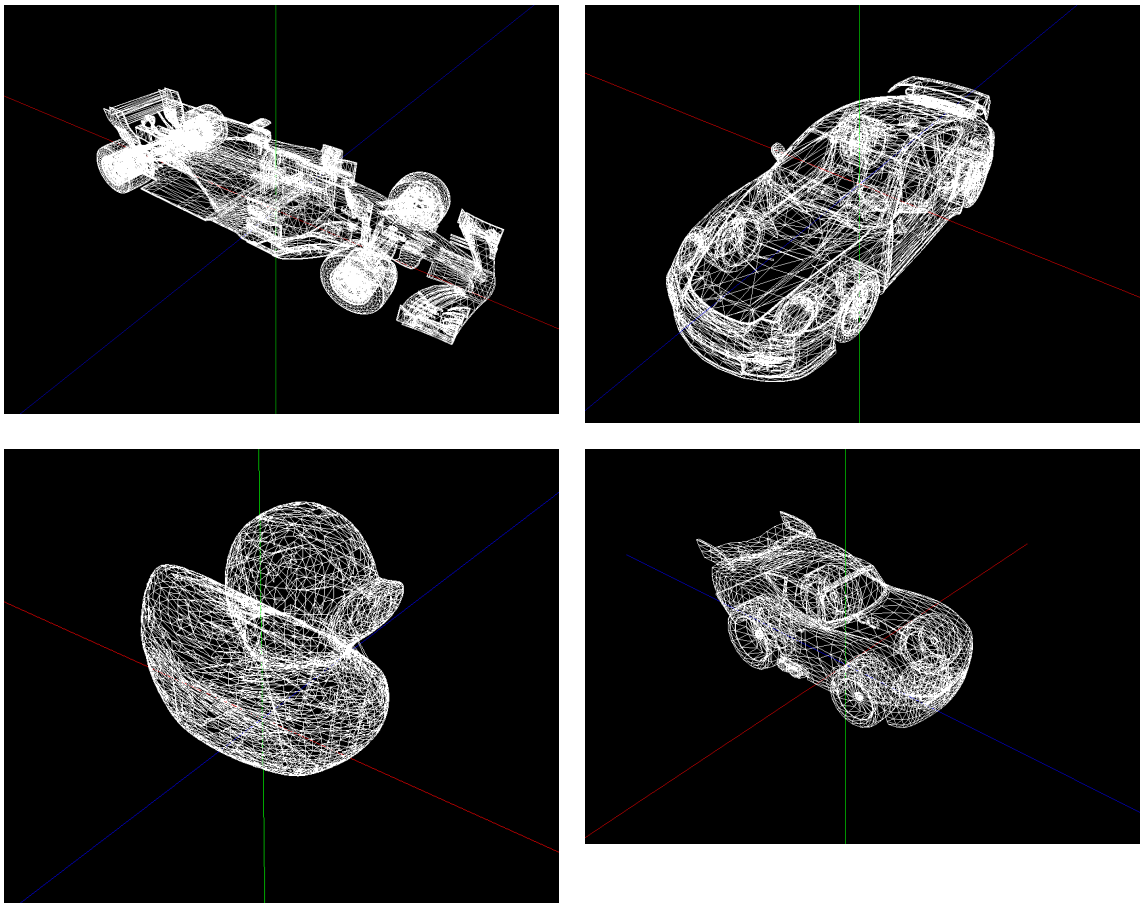

Figura 17: Testes com diferentes ficheiros .obj

3. Funcionalidades Destacadas

O projeto integra diversas funcionalidades que melhoram a experiência de visualização e manipulação dos modelos 3D:

- **Ocultação dos Eixos**: Para uma visualização mais limpa e sem distrações, é possível ocultar os eixos de referência, garantindo um foco total no modelo (“A”).
- **Movimentação**: Permite ao utilizador explorar livremente os modelos, ajustando a posição e o ângulo de visão para uma análise mais detalhada.
- **Zoom Dinâmico**: A funcionalidade de zoom (“M”/"L") possibilita aproximar ou afastar a visualização do modelo, facilitando tanto a inspeção de detalhes minuciosos como a observação da sua estrutura global.
- **Reset**: A funcionalidade de `reset` permite ao utilizador repor os parâmetros padrão definidos durante a inicialização da cena, restaurando a visualização para o estado original (tecla “R”).

Estas funcionalidades proporcionam maior flexibilidade e controlo na interação com os modelos 3D, tornando a experiência mais intuitiva e imersiva.

4. Conclusões e Trabalho Futuro

A implementação desta primeira fase revelou-se bastante satisfatória, permitindo ao grupo adaptar-se ao C++ e ao `OpenGL`, bem como a outras bibliotecas e até a conceitos matemáticos essenciais.

Nas próximas fases, o grupo pretende aprofundar conceitos fundamentais, como **Vertex Buffer Objects (VBOs)** e **indexação**, além de pretendermos desenvolver uma **GUI** que torne a interação mais intuitiva para o utilizador.

Adicionalmente, serão exploradas técnicas de otimização para a **GPU**, nomeadamente o **frustum culling**, que permite renderizar apenas os elementos dentro do campo de visão, reduzindo assim a carga de processamento e melhorando o desempenho.

Bibliografia

- [5] «vcpkg - Open source C/C++ dependency manager from Microsoft». Disponível em: https://vcpkg.io/en/
- [6] «CMake». Disponível em: https://cmake.org/
- [7] «OpenGL - The Industry Standard for High Performance Graphics». Disponível em: https://www.opengl.org/
- [8] «GitHub - Build software better, together». Disponível em: https://github.com/
- [9] Song Ho Ahn, «OpenGL Tutorials». Disponível em: https://www.songho.ca/opengl/
- [10] Dan Gries, «OpenGL Tutorial». Disponível em: https://www.youtube.com/watch?v=heFhbV5J_bY&ab_channel=DanGries

Definição de Ponto

A estrutura `Ponto` representa uma coordenada num espaço tridimensional e é definida como:

```
typedef struct Point {
  float x;
  float y;
  float z;

  Point(float x_val = 0.0f, float y_val = 0.0f, float z_val = 0.0f)
      : x(x_val), y(y_val), z(z_val) {}

} Point;
```