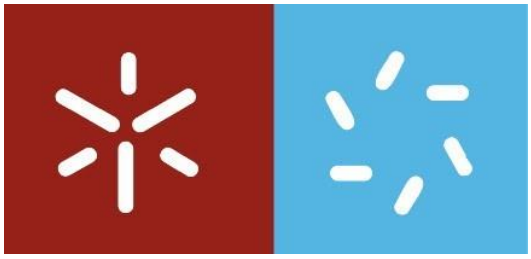


Trabalho Prático - Computação Gráfica

Licenciatura em Ciências da Computação

Fase 1 - Grupo 17

Bruno Neiva (a95311)
Gabriel Antunes (a101101)
Pedro Gonçalves (a101250)
Guilherme Pinho (a105533)

2 de março de 2025

Índice

1. Introdução

Neste trabalho prático da unidade curricular de Computação Gráfica, foi-nos proposto o desenvolvimento de dois programas: o *generator* e o *engine*. O primeiro tem como função gerar um ficheiro contendo as informações dos modelos necessários, ou seja, criar um arquivo com os vértices que compõem o modelo. Já o *engine* é responsável por interpretar um ficheiro de configuração em formato XML e exibir os modelos a partir dessa leitura. Este projeto foi desenvolvido em C++, utilizando a biblioteca OpenGL para a computação gráfica.

2. Generator

2.1. Funcionalidades

O *generator* através de um conjunto de parâmetros efetua os cálculos necessários para produzir a forma geométrica desejada e assim gerar um ficheiro .3d para, posteriormente, ser utilizado no *engine*. No nosso programa implementamos as seguintes primitivas gráficas:

- **Plano:**

  Cria um quadrado no plano XZ, centrado na origem do referencial, recebendo como parâmetros o comprimento das arestas, o número de divisões e o nome do ficheiro (.3d) resultante.

- **Box:**

  Cria um cubo, centrado na origem do referencial, recebendo como parâmetros o comprimento das arestas, o número de divisões em cada face e o nome do ficheiro (.3d) resultante.

- **Esfera:**

  Cria uma esfera, centrada na origem do referencial, recebendo como parâmetros o raio, as slices (divisões na vertical), as stacks (divisões na horizontal) e o nome do ficheiro (.3d) resultante.

- **Cone:**

  Cria um cone com a base paralela ao plano XZ, centrado na origem do referencial, recebendo como parâmetros o raio da base, a altura, as slices (divisões na vertical), as stacks (divisões na horizontal) e o nome do ficheiro (.3d) resultante.

- **Cilindro:**

  Cria um cilindro, com a base paralela ao plano XZ, centrado na origem do referencial, recebendo como parâmetros o raio da base, a altura, as slices (divisões na vertical) e o nome do ficheiro (.3d) resultante.

- **Torus**

  Cria um *torus*, centrado na origem do referencial, recebendo como parâmetros o raio interior, o raio exterior, os anéis (rings), as stacks (divisões na horizontal) e o nome do ficheiro (.3d) resultante.

2.1.1 Plano

1. Calcular o tamanho de cada divisão:

   tamanho_divisoes = dimensao / divisões

2. Calcular o deslocamento do centro do plano:

   dist_centro = dimensao / 2

3. Gerar os triângulos:

   Iteramos sobre as subdivisões em Z (i) e em X (j).

   Nas subdivisões i:

   z1 = (tamanho_divisoes * i) - dist_centro;

   z2 = (tamanho_divisoes * (i + 1)) - dist_centro;

   Nas subdivisões j:

   x1 e x2 seguindo a mesma logica do z e após esse cálculo também desenhamos os triângulos necessários para gerar o plano:

```
// Triangulo 1
```

```
triangulos << x1 << ' ' << 0 << ' ' << z1 << '\n';
triangulos << x1 << ' ' << 0 << ' ' << z2 << '\n';
triangulos << x2 << ' ' << 0 << ' ' << z2 << '\n';

 // Triangulo 2
triangulos << x1 << ' ' << 0 << ' ' << z1 << '\n';
triangulos << x2 << ' ' << 0 << ' ' << z2 << '\n';
triangulos << x2 << ' ' << 0 << ' ' << z1 << '\n';
```

2.1.2 Box

Apresentamos o cálculo da base e do topo pois as laterais seguem a mesmo lógica, mas em vez de calcularmos o x, z, calculámos o y, z e o y, x. E o cálculo da translação fica sempre na variável que não é calculada. Em suma, é gerar os planos descritos acima, mas em planos do eixo diferentes e fazemo-lo duas faces de cada de vez.

1. Calcular o tamanho de cada divisão:

   tamanho_divisoes = dimensao / divisoes

2. Calcular a translação para centrar a box:

   altura = dimensao / 2

3. Criar base e topo:

   Iteramos sobre as subdivisões em Z (i) e em X (j).

   Nas subdivisões i:

z1 = (tamanho_divisoes * i) - dist_centro;

z2 = (tamanho_divisoes * (i + 1)) - dist_centro;

Nas subdivisões j:

x1 e x2 seguindo a mesma logica do z e após esse cálculo também desenhamos os triângulos necessários para gerar a base (y negativo) e o topo (y positivo):

```
// y positivo
// Triangulo 1
   triangulos << x1 << ' ' << altura << ' ' << z1 << '\n';
   triangulos << x1 << ' ' << altura << ' ' << z2 << '\n';
   triangulos << x2 << ' ' << altura << ' ' << z2 << '\n';
// Triangulo 2
   triangulos << x1 << ' ' << altura << ' ' << z1 << '\n';
   triangulos << x2 << ' ' << altura << ' ' << z2 << '\n';
   triangulos << x2 << ' ' << altura << ' ' << z1 << '\n';
// y negativo
// Triangulo 1
  triangulos << x1 << ' ' << -altura << ' ' << z1 << '\n';
  triangulos << x2 << ' ' << -altura << ' ' << z2 << '\n';
  triangulos << x1 << ' ' << -altura << ' ' << z2 << '\n';
// Triangulo 2
  triangulos << x1 << ' ' << -altura << ' ' << z1 << '\n';
  triangulos << x2 << ' ' << -altura << ' ' << z1 << '\n';
  triangulos << x2 << ' ' << -altura << ' ' << z2 << '\n';
```

2.1.3 Esfera

1. Variáveis e cálculos iniciais:

   - *dPhi*: variação do ângulo φ (longitude) por fatia. Ele é calculado como dPhi = 2 * M_PI / slices. Como uma esfera completa tem 360º (ou 2π radianos), o código divide essa circunferência pela quantidade de fatias para determinar o incremento angular de cada fatia.
   - *dTheta*: variação do ângulo θ (latitude) por camada. Ele é calculado como dTheta = M_PI / stacks. O código divide 180º (π radianos) pela quantidade de camadas para determinar o incremento angular de cada camada.

2. Calcular vértices:

   A esfera é gerada através da repetição de dois loops principais:

   - Loop sobre stacks (camadas horizontais):

     angle e *next_angle* determinam as posições latitudinais atuais e da próxima camada (de θ = 0 a θ = π).

- Loop sobre slices (fatias verticais):

  space e *next_space* determinam as posições longitudinais atuais e da próxima fatia (de $\varphi = 0$ a $\varphi = 2\pi$).

Para cada ponto na esfera, as coordenadas cartesianas (x, y, z) são calculadas com base nos ângulos angle (latitude, θ) e space (longitude, φ):

- **x** é calculado usando *radius * sin(angle) * cos(space)*, que projeta a posição no eixo X.
- **y** é calculado com *radius * cos(angle)*, que projeta a posição no eixo Y.
- **z** é calculado com *radius * sin(angle) * sin(space)*, que projeta a posição no eixo Z.

O mesmo cálculo é feito para o próximo ponto, em relação à próxima fatia e camada, com o uso de *next_angle* e *next_space*.

3. Gerar triângulos:

```
// Left Triangles
triangles << x1 << ' ' << y << ' ' << z1 << '\n';
triangles << next_x1 << ' ' << next_y << ' ' << next_z1 << '\n';
triangles << x2 << ' ' << y << ' ' << z2 << '\n';

// Right Triangles
triangles << x2 << ' ' << y << ' ' << z2 << '\n';
triangles << next_x1 << ' ' << next_y << ' ' << next_z1 << '\n';
triangles << next_x2 << ' ' << next_y << ' ' << next_z2 << '\n';
```

2.1.4 Cone

1. Base:

Para calcular quantas slices a base do cone terá, será necessário dividir por 360º (em radianos) pelo número pretendido:

angle = 2 * M_PI / slices;

Gerar os triângulos seguindo a seguinte logica:

Para cada slice(i):

Coordenadas usando os ângulos anteriores:

x = (radius ∗ sin(ang1))

z = (radius ∗ cos(ang1))

next x = (radius ∗ sin(ang2))

next z = (radius ∗ cos(ang2))

Desenhamos os triângulos necessários para gerar a base do cone:

```
triangles << 0 << ' ' << 0 << ' ' << 0 << '\n';
triangles << x2 << ' ' << 0 << ' ' << z2 << '\n';
triangles << x1 << ' ' << 0 << ' ' << z1 << '\n';
```

2. Lateral:

Para calcular quantas stacks o cone terá na lateral ´e necessário fazer o seguinte calculo:

heightBase = height / stacks;

Gerar os triângulos seguindo a seguinte logica:

Para cada stack(i):

Para cada slice (calculada previamente) (j):

Coordenadas do y usando os ângulos anteriores:

y = (height stack ∗ j)

next y = (height stack ∗ (j+ 1))

Coordenadas (x, z) inferiores:

```
x1 = currRadius * sin(angle * i);
 z1 = currRadius * cos(angle * i);
x2 = currRadius * sin(angle * (i + 1));
z2 = currRadius * cos(angle * (i + 1));
```

Coordenadas (x, z) superiores:

```
next_x1 = nextRadius * sin(angle * i);
next_z1 = nextRadius * cos(angle * i);
next_x2 = nextRadius * sin(angle * (i + 1));
next_z2 = nextRadius * cos(angle * (i + 1));
```

Desenhamos os triângulos necessários para gerar a lateral do cone:

```
// Left Triangles
triangles << x1 << ' ' << y << ' ' << z1 << '\n';
triangles << x2 << ' ' << y << ' ' << z2 << '\n';
triangles << next_x1 << ' ' << next_y << ' ' << next_z1 << '\n';

// Right Triangles
triangles << x2 << ' ' << y << ' ' << z2 << '\n';
triangles << next_x2 << ' ' << next_y << ' ' << next_z2 << '\n';
triangles << next_x1 << ' ' << next_y << ' ' << next_z1 << '\n';
```

2.1.5 Cilindro

1. Cálculo das coordenadas dos vértices:

Para cada fatia i do cilindro, o código calcula dois pontos (x1, z1) e (x2, z2) para as posições na base do cilindro, usando a trigonometria:

- x1 e z1 são as coordenadas da base do cilindro para o ponto atual, calculadas como:

  x1 = radius * sin(angle)

  z1 = radius * cos(angle)

- x2 e z2 são as coordenadas da base do cilindro para o próximo ponto (a próxima fatia), calculadas da mesma forma:

  x2 = radius * sin(next_angle)

  z2 = radius * cos(next_angle)

Os ângulos *angle* e *next_angle* são calculados a partir de i (a fatia atual) e i + 1 (a fatia seguinte).

2. Gerar triângulos:

O código cria três conjuntos de triângulos para representar o cilindro:

Base Inferior:

```
// Bottom Base
triangles << x2 << ' ' << 0 << ' ' << z2 << '\n';
triangles << x1 << ' ' << 0 << ' ' << z1 << '\n';
triangles << 0 << ' ' << 0 << ' ' << 0 << '\n';
```

Base Superior:

```
// Top Base
triangles << x1 << ' ' << height << ' ' << z1 << '\n';
triangles << x2 << ' ' << height << ' ' << z2 << '\n';
```

```
triangles << 0 << ' ' << height << ' ' << 0 << '\n';
```

Faces Laterais:

```
// Left Triangle
triangles << x1 << ' ' << 0 << ' ' << z1 << '\n';
triangles << x2 << ' ' << 0 << ' ' << z2 << '\n';
triangles << x1 << ' ' << height << ' ' << z1 << '\n';

// Right Triangle
triangles << x2 << ' ' << 0 << ' ' << z2 << '\n';
triangles << x2 << ' ' << height << ' ' << z2 << '\n';
triangles << x1 << ' ' << height << ' ' << z1 << '\n';
```

2.1.6 Torus

Geramos um torus 3D dividindo-o em uma série de *rings* e *stacks*, e conectando essas fatias com triângulos. O uso das equações paramétricas para calcular as coordenadas garante que a forma do torus seja gerada de maneira precisa e eficiente.

1. Cálculos:

   outerAngle: ângulo (em radianos) para cada anel (anima o círculo grande).

   float outerAngle = 2 * M_PI / rings;

   innerAngle: ângulo (em radianos) para cada "fatias" do círculo pequeno (perfil).

   float innerAngle = 2 * M_PI / stacks;

2. Para cada ring(i):

   Para cada stack(j):

   Coordenadas do y:

   ```
   // y dos priemiros pontos de cada anel
   ```

```
y1 = innerRadius * sin(innerAngle * j);
// y dos segundos pontos de cada anel
y2 = innerRadius * sin(innerAngle * (j + 1));
```

Coordenadas (x,z):

```
x1 = (outerRadious + innerRadius*cos(innerAngle*j)) * sin(outerAngle *
i);
z1 = (outerRadious + innerRadius*cos(innerAngle*j)) * cos(outerAngle0*
i);

nextx1 = (outerRadious + innerRadius * cos(innerAngle * j))* sin
(outerAngle*(i+1));

nextz1 = (outerRadious + innerRadius * cos(innerAngle * j)) * cos
(outerAngle* (i+1));
```

Coordenadas (x,z):

```
x2 = (outerRadious+innerRadius*cos(innerAngle*(j+1)))*sin(outerAngle*
i);
z2 = (outerRadious+innerRadius*cos(innerAngle*(j+1)))*
cos(outerAngle*i);
```

Desenhamos os triângulos necessários para gerar o torus:

```
// Triangulo 1
triangles << x1 << ' ' << y1 << ' ' << z1 << '\n';
triangles << nextx2 << ' ' << y2 << ' ' << nextz2 << '\n';
triangles << x2 << ' ' << y2 << ' ' << z2 << '\n';
// Triangulo 2
triangles << x1 << ' ' << y1 << ' ' << z1 << '\n';
triangles << nextx1 << ' ' << y1 << ' ' << nextz1 << '\n';
triangles << nextx2 << ' ' << y2 << ' ' << nextz2 << '\n';
```

3. Engine

A engine é responsável por ler os ficheiros XML. Esta lê os ficheiros (.3d) gerados pelo generator e gera as figuras pretendidas.

3.1 Leitura do ficheiro de configuração

A função responsável por ler o ficheiro de configuração na nossa engine é:

- void loadXML();

3.2 Desenho dos modelos

No que toca ao desenho dos modelos, desenvolveram-se as seguintes funções:

- void drawAxis():

  Desenha os eixos x, y e z.

- void drawModels():

  Desenha os modelos descritos no ficheiro de configuração.

3.3. Funcionamento da Câmara

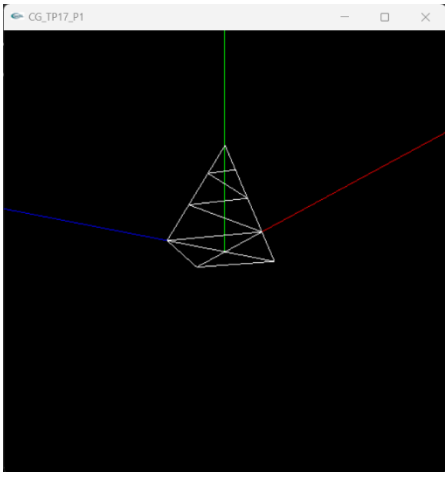
3.3.1 Explorer Mode Camera

A câmara gira em torno do ponto de foco (lookAtX, lookAtY, lookAtZ), como se estivesse presa a uma esfera imaginária.

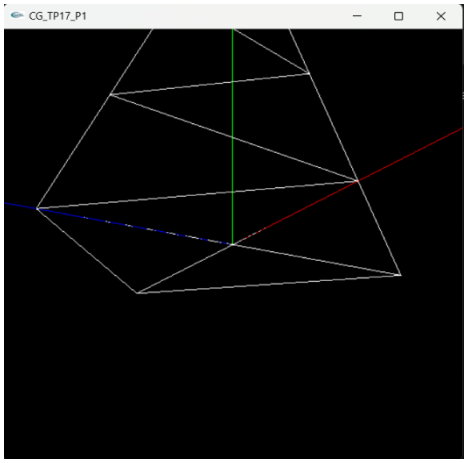
O utilizador pode mover a câmara usando as teclas:

- **W / S** → Move a câmara para cima/baixo ao longo da esfera (controla o ângulo beta).
- **A / D** → Rotação em torno do eixo vertical (alfa).
- **+ / -** → Aproxima ou afasta a câmara do centro (zoom).

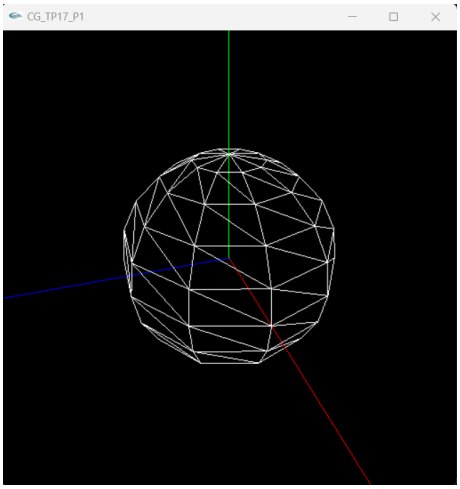
4. Demos



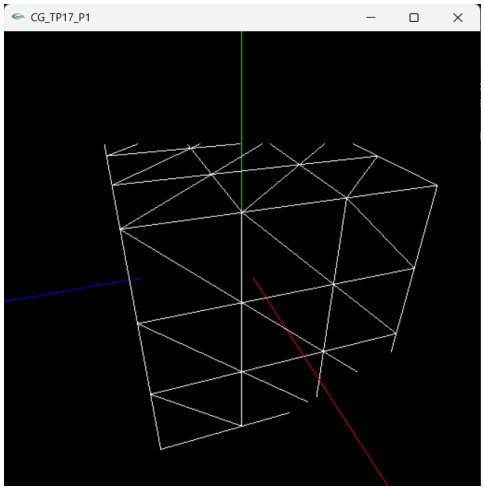
test_1_1



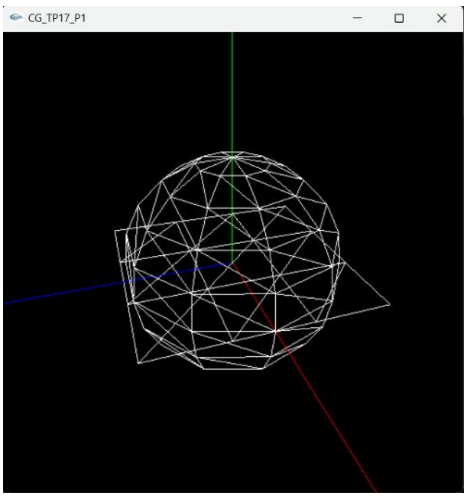
test_1_2



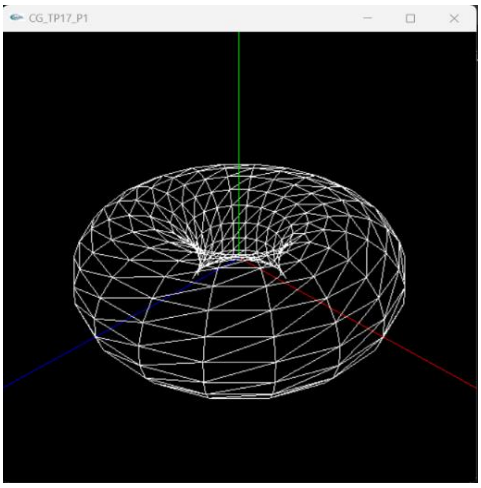
test_1_3



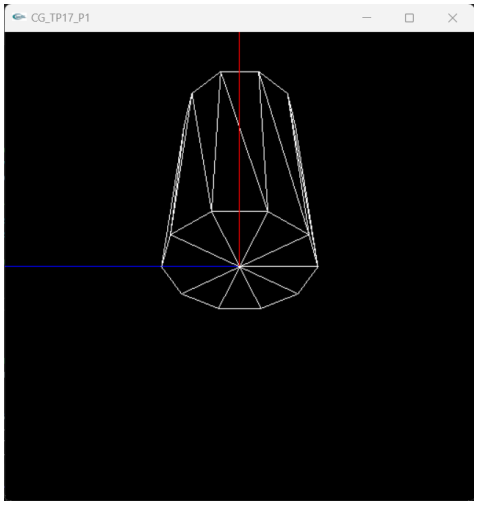
test_1_4



test_1_5



test_1_6



test_1_7

5. Conclusão

Na primeira fase da realização deste trabalho lidamos com situações que nos colocaram à prova e que nunca nos deparamos durante as aulas. No entanto, isso permitiu-nos utilizar esse desafio para procurar o conhecimento necessário para a conclusão deste trabalho. A presente fase permitiu a consolidação da matéria lecionada nas aulas, especificamente na utilização e no domínio de ferramentas e conceitos de computação gráfica e na linguagem de C++