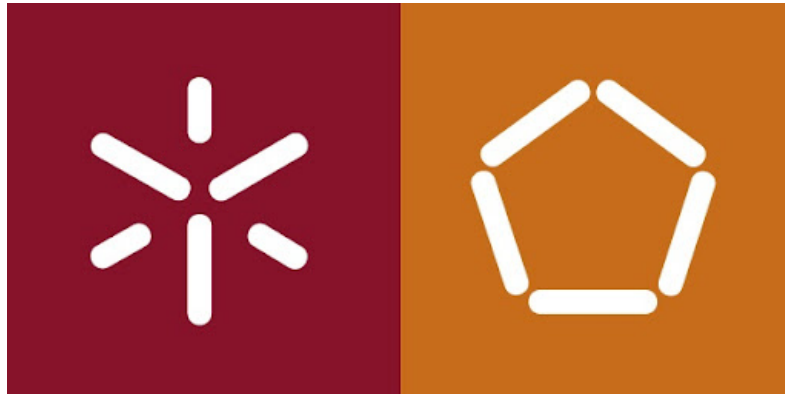


Universidade do Minho

Escola de Engenharia



Graphical Primitives

Computação Gráfica

Relatório do Trabalho Prático

Grupo nº44

Alexandre Eduardo Vieira Martins A93242

José Eduardo Silva Monteiro Santos Oliveira A100547

Pedro Afonso Moreira Lopes A100759

21 de maio de 2023

Índice:

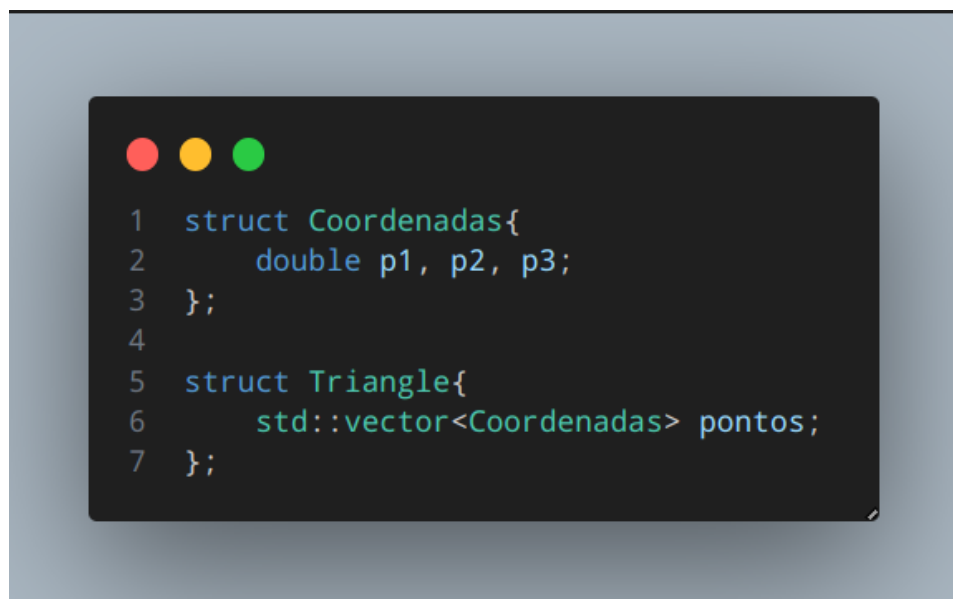
1. Introdução
2. Generator
 - a. Primitivas
 - i. Plano
 - ii. Caixa
 - iii. Cone
 - iv. Esfera
3. Engine
4. Conclusão e trabalho futuro

1.Introdução

Este trabalho foi proposto no âmbito da unidade curricular de Computação Gráfica e tem como objetivo desenvolver um pequeno motor gráfico para representar objetos 3D. Nesta primeira fase do projeto foi pedido para criar duas aplicações, uma para gerar os ficheiros com as informações dos modelos(o generator) e outra que irá ler o ficheiro de configuração, escrito em XML e mostrar os modelos (o engine). Também devem ser implementadas algumas primitivas gráficas como o plano, o cubo, o cone e a esfera.

2.Generator

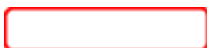
O ficheiro generator.cpp disponibiliza vários métodos para gerar os pontos para desenhar as quatro primitivas. As funções do estilo **generate_** calculam e geram os pontos necessários para cada uma das figuras que vão ser desenhadas, sendo gerados 3 pontos de cada vez, ou seja, triângulo a triângulo, tal e qual como as estruturas abaixo indicam:

A screenshot of a code editor window with a dark background and light-colored text. The code defines two C++ structures. The first structure is named 'Coordenadas' and contains three double variables: p1, p2, and p3. The second structure is named 'Triangle' and contains a vector of 'Coordenadas' objects, named 'pontos'. The code is numbered from 1 to 7.

```
1 struct Coordenadas{
2     double p1, p2, p3;
3 };
4
5 struct Triangle{
6     std::vector<Coordenadas> pontos;
7 };
```

Figura 1. Estruturas Coordenadas e Triangles utilizadas

Com isto feito, todos os triângulos criados vão ser escritos e guardados num ficheiro .3d, para serem lidos pela engine da seguinte forma:



```

cone
0 0 0 ; 6.12323e-17 0 -1 ; 1 0 -0
0 0 0 ; -1 0 -1.22465e-16 ; 6.12323e-17 0 -1
0 0 0 ; -1.83697e-16 0 1 ; -1 0 -1.22465e-16
0 0 0 ; 1 0 2.44929e-16 ; -1.83697e-16 0 1

0 2 0 ; 0.333333 1.33333 -0 ; 2.04108e-17 1.33333 -0.333333
0 2 0 ; 2.04108e-17 1.33333 -0.333333 ; -0.333333 1.33333 -4.08216e-17
0 2 0 ; -0.333333 1.33333 -4.08216e-17 ; -6.12323e-17 1.33333 0.333333
0 2 0 ; -6.12323e-17 1.33333 0.333333 ; 0.333333 1.33333 8.16431e-17
0 2 0 ; 0.333333 1.33333 8.16431e-17 ; 1.02054e-16 1.33333 -0.333333

```

Figura 2. Formato dos ficheiros 3D

Tipo da figura, que vai ser útil depois na engine para saber que tipo de tratamento dar aos pontos do ficheiro.

Secção de pontos, estando em cada linha um triângulo, ou seja, 3 pontos separados por ; . Dependendo da forma, os pontos no ficheiro vão estar separados por secções, sendo cada uma equivalente a uma parte da figura.

A função *main* recebe e interpreta os inputs dos utilizadores para depois serem criados os pontos de cada figura.

2.a. Primitivas

Todas as primitivas são desenhadas a partir de triângulos, utilizando o *GL_TRIANGLES* e 3 pontos definidos pelos cálculos feitos para cada figura, para cada triângulo utilizado para o desenho das tais.

2.a.i. Plano

Para o desenho do plano, necessitamos primeiro de calcular o comprimento de cada um dos *N* divisões que o plano tem. Para isto, dividimos o comprimento dado como argumento de cada lado pelas *N* divisões, sabendo assim agora o comprimento de cada divisão. Com isto feito, fazemos agora 2 ciclos, onde o ciclo interior vai desenhar os triângulos em ordem a *X* e o de fora em ordem a *Z*, desenhando então o plano linha a linha. Para o desenho dos pontos, de forma ao plano estar centrado no centro, os pontos vão ser desenhados de $-\text{comprimento}/2$ até $\text{comprimento}/2$, de acordo com o eixo *XZ*, dando assim o seguinte plano:

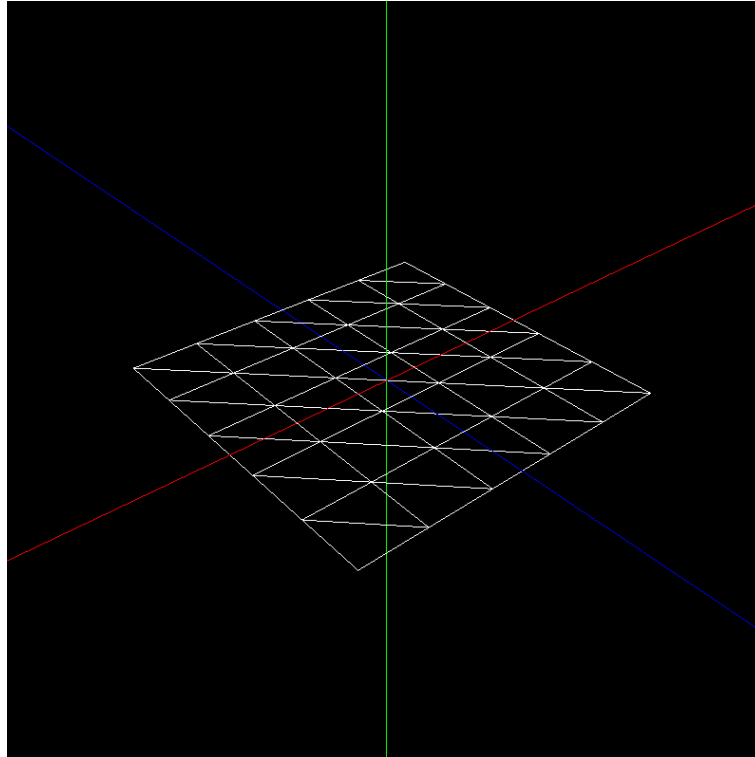


Figura 3. Representação do Plano

2.a.ii. Caixa

Para a construção da caixa, decidimos aplicar o código do plano para cada face, fazendo as alterações necessárias para o desenho correto dos triângulos em cada face. Tal como na face, para deixar a caixa centrada no centro, os pontos são desenhados entre $-\text{comprimento}/2$ e $\text{comprimento}/2$, para todos os eixos, criando cada divisão através do aumento, iteração a iteração, de $\frac{\text{length}}{\text{divisions}}$ resultando na figura seguinte:

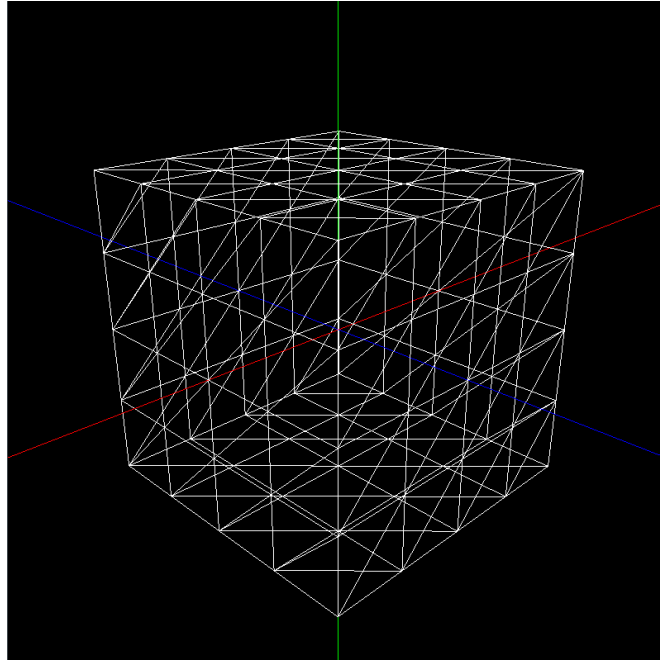


Figura 4. Representação da Caixa.

2.a.ii. Cone

Para o desenho do cone, decidimos, para começar, em inspirar-nos no cilindro para a criação da base.

Com isto feito, passamos então à criação do corpo do cone. Para isto, como reparamos que a primeira stack do cone não tinha 2 triângulos por fatia, decidimos tratar da primeira stack separadamente do resto das stacks de cada fatia. Para isto, fizemos alguns cálculos, como se podem ver aqui em baixo:

$$\textit{altura de uma stack} = \frac{\textit{altura do cone}}{N^{\circ} \textit{ de stacks}}$$

$$\hat{\text{Ângulo}} \beta \textit{ de cada fatia} = \frac{2\pi}{N^{\circ} \textit{ de fatias}}$$

$$\textit{Raio de uma stack} = \frac{\textit{Raio do cone}}{N^{\circ} \textit{ de stacks}} * N^{\text{a}} \textit{ da stack}$$

$$\textit{Altura da Stack } n = \textit{altura do cone} - (n * \textit{Altura de uma Stack})$$

Com estas fórmulas descobertas, podemos assim determinar as coordenadas dos pontos dos triângulos:

$$\begin{aligned}x &= \cos(N^{\circ} \text{ da fatia} * \beta) * \text{Raio de uma stack} \\y &= \text{Altura da Stack } n \\z &= -\sin(N^{\circ} \text{ da fatia} * \beta) * \text{Raio de uma stack}\end{aligned}$$

Com estas fórmulas, foram utilizados 2 ciclos para o desenho do corpo: O de dentro para tratar das stacks da fatia em que se encontra e o de fora para tratar de cada fatia e desenhar a primeira stack do cone.

Outra coisa importante a realçar é que, como não fizemos o cone a partir de uma pilha de círculos cujo raio vai diminuindo, as fórmulas anteriores, dependendo do triângulo que desenham, podem necessitar de um *Raio de uma stack* maior ou *Nº da fatia* maior.

Sendo assim, ficamos assim com a seguinte representação de um cone:

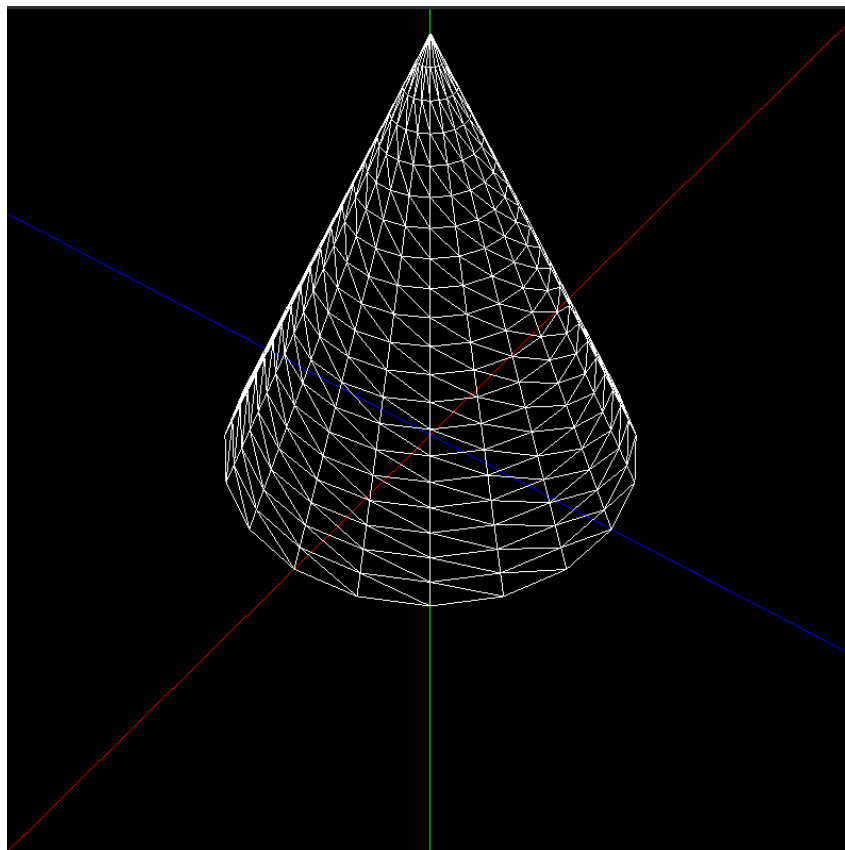


Figura 5. Representação do Cone

2.a.iii. Esfera

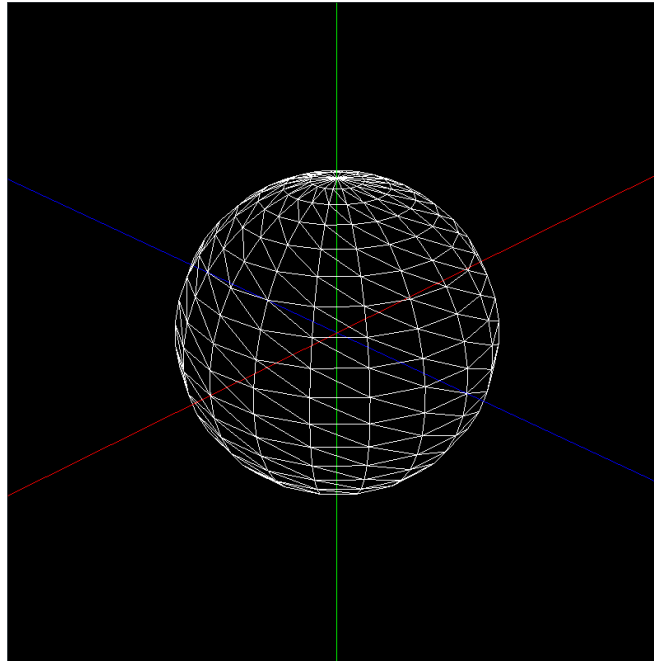


Figura 6. Representação da esfera.

Para o desenho da esfera, são passados três argumentos para a criação da mesma, as stacks, as slices e o raio. A interseção das mesmas irá criar os retângulos que compõem a esfera, e cada um destes irá ser formado por dois triângulos.

Para a escolha dos pontos serão utilizadas coordenadas esféricas. Para isso, serão necessárias duas variáveis, φ e θ , para representar os dois ângulos que pertencem às coordenadas.

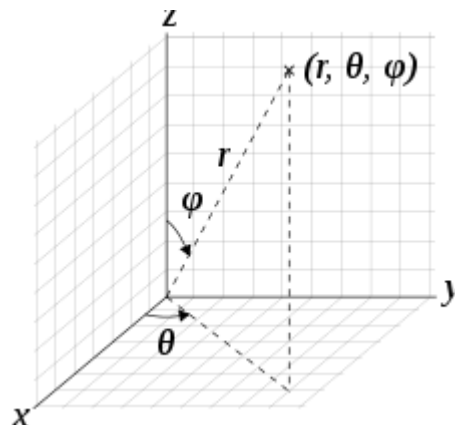


Figura 7. Coordenadas esféricas.

Para serem calculadas as coordenadas de cada ponto será necessário calcular a stack e a slice em que cada ponto se encontra. Estas foram as fórmulas utilizadas para calcular qual a slice e a stack que estão a ser utilizadas:

$$\text{currentSlice} = \frac{\theta \cdot 2\pi}{nrSlices}$$

$$\text{currentStack} = \frac{\varphi \cdot \pi}{nrStacks}$$

Para serem calculadas as verdadeiras coordenadas x, y e z é necessário aplicar 3 fórmulas:

$$x = radius \times \sin(\theta) \times \sin(\phi)$$

$$y = radius \times \cos(\theta)$$

$$z = radius \times \sin(\theta) \times \cos(\phi)$$

3.Engine

Este módulo tem dois componentes:

- O parsing do ficheiro xml
- A render scene.

A função de *parsexml*, tal como o nome indica, faz o parsing dos parâmetros contidos no ficheiro xml e, para auxiliar este processo, é utilizado a biblioteca *tinyxml2*.

No início da função, é feita a leitura do primeiro nó do XML ("world") e, a partir daí, acedemos aos nós filhos com recurso a um ciclo for e funções auxiliares. Cada uma delas está definida para dar o parse dos atributos esperados. Nestas funções, guardamos os valores dos atributos de cada nó em variáveis globais previamente criadas, que serão usadas nas funções subsequentes. Por exemplo, os valores dos atributos "width" e "height" do nó "window" são escritos em variáveis globais para serem usados na função main com o comando *glutInitWindowSize*. Para lidar com os vários nós "model", que podem existir, nesta fase fazemos a leitura dos atributos e colocamos num vetor.

A função *renderScene*, é responsável por chamar as funções responsáveis para abrir o ficheiro .3d e desenhar os triângulos já definidos nesses ficheiros. Nesta função, vamos desenhar os eixos, para ajudar na correção de quaisquer problemas no desenho das formas e, para cada ficheiro encontrado nos nós *model*, vai ser chamada a função *handle_form* para tratar do ficheiro.

Na *handle_form*, vai ser lida a primeira linha, que contém o tipo da forma (mencionado na Figura 2) e, dependendo da forma contida nessa linha, vai ser chamada uma função de desenho para essa figura e, conseqüentemente, uma função de parse dos ficheiros .3d.

Quanto ao parse dos ficheiros, tal como mencionado no capítulo do *Generator*, foram utilizadas essas estruturas para guardar cada triângulo em cada linha num ou mais vetor de triângulos, dependendo da forma, que depois vão ser utilizados, finalmente, para desenhar as figuras em si.

4.Conclusão

Nesta fase, baseado nos resultados obtidos, conseguimos realizar as tarefas pedidas com sucesso, apesar de melhorias serem necessárias para as próximas fases. Entre essas melhorias, o parser do xml não suporta alguns requisitos para fases seguintes e os VBO's poderiam ter sido aplicados nesta fase também mas optamos por deixar isto para fases futuras.