Universidade do Minho

Departamento de Informática

Computação Gráfica

Phase 1 – Graphical primitives

Grupo 39

Gonçalo Araújo Brandão A100663
Maya Gomes A100822
Luís de Castro Rodrigues Caetano A100893

8 de março de 2024

Conteúdos

1 Introdução

Este relatório visa descrever a primeira fase do projeto prático da Unidade Curricular de Computação Gráfica, desenvolvido na linguagem C++. Para este meio iremos recorrer à utilização do *OpenGL* para a construção de alguns modelos 3D, utilizando a biblioteca *GLUT*.

Na totalidade deste projeto prático existem quatro frases. Esta divisão tem como intuito organizar e simplificar o desenvolvimento deste, contribuindo também para a sua melhor compreensão.

Iremos começar este documento mencionando e explicando o problema proposto para esta primeira fase.

Seguidamente, passaremos a dar um contexto mais concreto para os resultados que obtivemos e como lá chegamos, nomeadamente o *Generator*, que gerará um ficheiro com pontos para o modelo especificado e uma *Engine* que lê os modelos do ficheiro XML.

2 Descrição do problema

Esta primeira fase será para desenvolver dois programas, um *Generator* que gerará um ficheiros com vértices para o modelo que for especificado e a *Engine* propriamente dita que lerá um ficheiro escrito em XML, que servirá de ficheiro de configuração e representará graficamente os modelos requeridos.

Tendo isto em conta e com análise do enunciado, confirmamos que os objetivos estipulados para esta fase serão, a nível do *Generator* e da *Engine*:

- *Generator*

  O generator recebe o nome do modelo pretendido a criar, os parâmetros relativos ao modelo em questão e o nome do ficheiro onde os vértices irão ser guardados. Desta forma, ele é responsável por criar um ficheiro que contem as coordenadas dos pontos dos triângulos do modelo que pretendemos desenhar.

  - **Plane**

    O plano é um conjunto de 4 pontos que representará um quadrado centrado na origem no plano XZ.
  - **Box**

    O cubo, ou caixa, segue os mesmos princípios que o plano, sendo que apenas as faces são paralelas aos planos coordenados. Este necessita das dimensões x,y e z e do número de divisões.
  - **Cone**

    Para o cone é necessário o raio da base, a altura e o número de divisões verticais e horizontais.
  - **Sphere**

    Para a esfera necessitamos de dois ângulos, um raio, e do número de divisões verticais e horizontais.

Forma Geométrica	**Parâmetros**
Plano	length, divisions, .3d filename
Caixa	length, divisions, .3d filename
Cone	radius, height, slices, stacks, .3d filename
Esfera	radius, slices, stacks, .3d filename

Tabela 1: Parâmetros das formas geométricas

- *Engine*

  O *Engine* é responsável por ler o ficheiro XML que contém os modelos das primitivas (criados no *Generator*).

  - lê os modelos do ficheiro XML,
  - desenha, utilizando triângulos, as primitivas relativas aos modelos.

3 Estrutura da aplicação

Decidimos organizar cada fase numa diretoria diferente, assim como, cada entidade tem a sua respetiva diretoria com o código fonte.

Figura 1: Models.

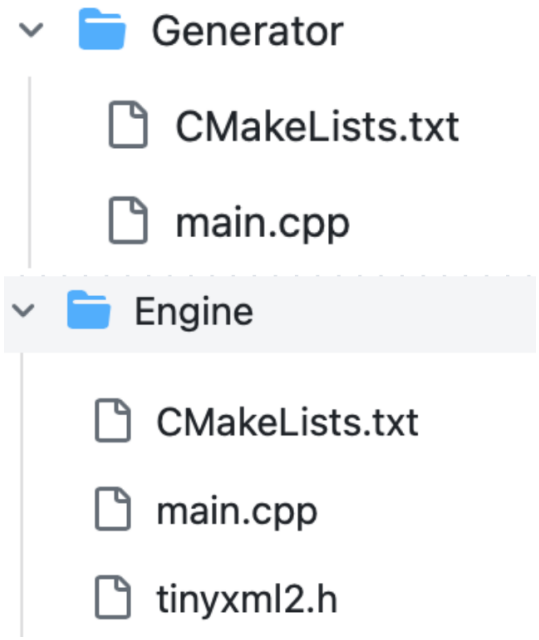

Figura 2: Generator e Engine.

4 Resolução do problema

O problema a ser tratado envolve a representação de vértices num plano a três dimensões. Desta forma, foi criada uma estrutura *Point*, de forma a armazenar as coordenadas de um ponto possuindo assim os parâmetros x, y e z.

```
// Definição da estrutura Point
struct Point {
    float x;
    float y;
    float z;
};
```

Figura 3: Estrutura *Point*.

4.1 Generator

4.1.1 Plane

Num espaço tridimensional, o mínimo para definir um plano são 3 pontos não colineares, sendo que o conjunto destes forma um plano.

Para este projeto, para representar um plano iríamos precisar de 4 pontos já que o nosso plano seria representado por um quadrado no plano XZ.

Portanto, decidimos definir concretamente um dos 4 pontos e definir os outros à custa deste, já que teríamos os dados do mesmo guardados na nossa estrutura. Na figura 4 percebemos como o ponto P1 define todos os outros pontos do plano (P2, P3 e P4).

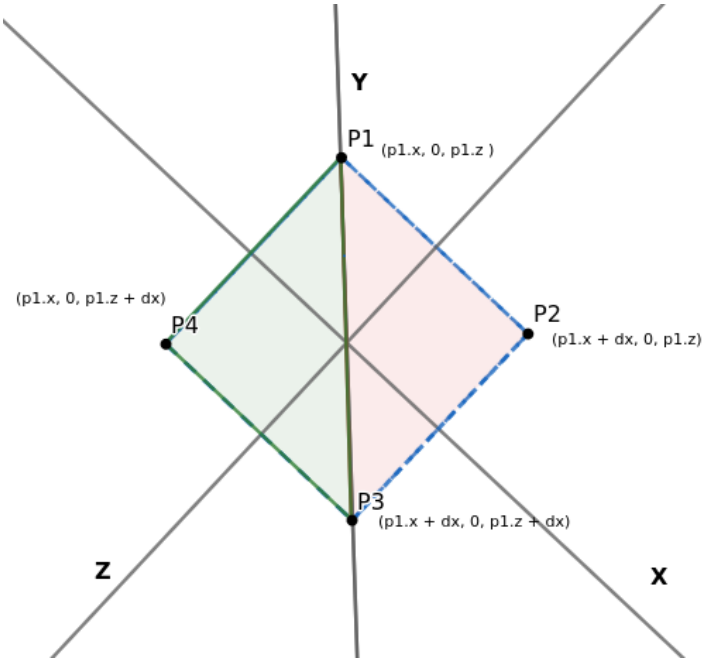

Figura 4: Esquema do plano.

Após a etapa anteriormente descrita, precisávamos de decidir como iríamos gerar o primeiro ponto P1. Por convenção, definimos que o P1 devia ser sempre um ponto do 3º quadrante do plano XZ de forma a que fosse mais intuitivo a definição dos outros pontos do sistema.

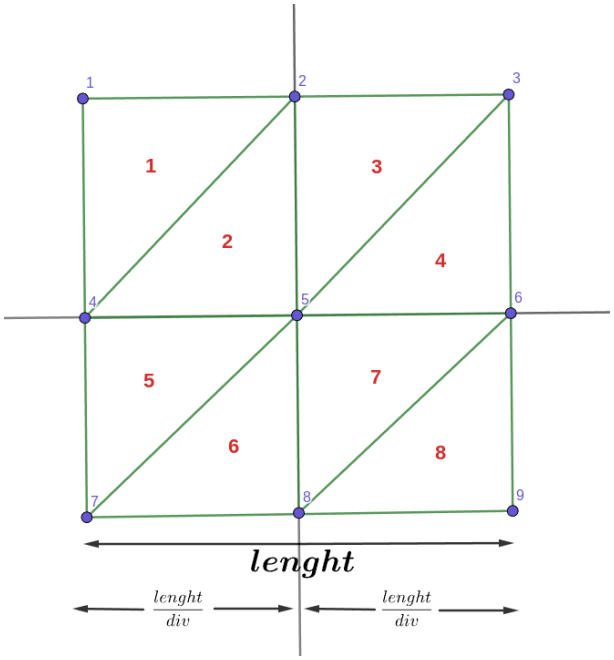

Figura 5: Representação do plano

Para exemplificar isto, damos um exemplo de um plano com 2 divisões, como na figura acima. Para determinarmos o ponto do canto inferior esquerdo (ponto 7 == P1) com o valor de $x = -\frac{lenght}{2}$ e o valor de $z = -\frac{lenght}{2}$, teríamos condições para desenhar os triângulos 5 e 6. Para desenhar o resto do plano, começamos pela divisão de $\frac{lenght}{divisions}$ e incrementamos esse valor a cada iteração até *divisions*. Este processo necessita de dois ciclos for, um para o desenho vertical e outro para o desenho horizontal.

Este processo também garante que o nosso plano está centrado na origem. Para o plano gerado ser no plano XZ o y é sempre igual a 0.

4.1.2 Resultados do plano:

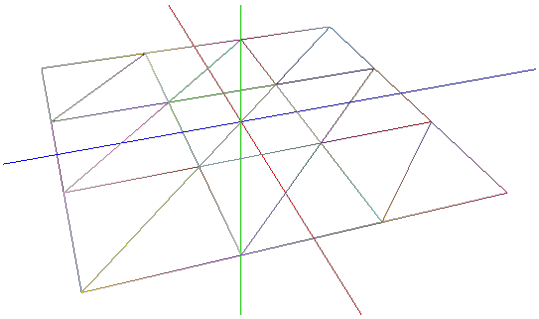

Figura 6: Plano sem preenchimento.

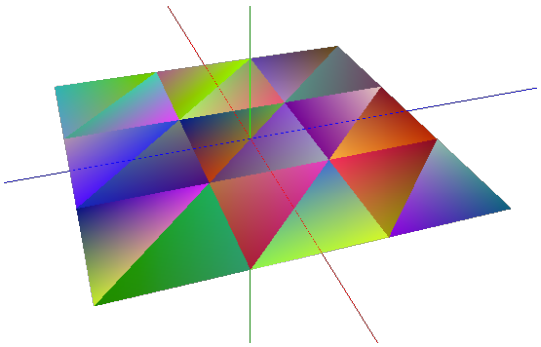

Figura 7: Plano com preenchimento.

4.1.3 Box

Para implementar a caixa decidimos utilizar os mesmo princípios que na elaboração do plano, isto porque a caixa é feita por 6 planos. As únicas diferenças são que alguns planos encontram-se na vertical e outros na horizontal. Sendo que o plano está centrado na origem as suas faces são paralelas aos eixos coordenados e a distância aos mesmos é igual a $\frac{lenght}{2}$.

Desta forma, iremos observar que a cada iteração dos nossos ciclos, as coordenadas são todas incrementadas, à exceção da coordenada da qual queremos o plano respetivo paralelo.

O valor das coordenadas que não é alterado será sempre igual a $-\frac{lenght}{2}$. ou $\frac{lenght}{2}$, consoante se é *Bottom/Top, Front/Back ou Right/Let.*

Para a representação dos pontos para o lado que queremos que estes renderizem, utilizamos a regra da mão direita.

4.1.4 Criação das faces da caixa:

Faces Inferiores e Superiores: Duas iterações aninhadas são usadas para percorrer todas as subdivisões na largura e profundidade da caixa. Pontos são criados para formar cada quadrado na face inferior e superior da caixa. Nas face inferior e superior, cada quadrado é formado por dois triângulos.

Desta forma calcula-se as coordenadas da face superior com: p1 = $(-length \div 2 + i \times dx, length \div 2, -length \div 2 + j \times dx)$;

```
p2 = (p1.x + dx, p1.y, p1.z);
p3 = (p2.x, p1.y, p2.z + dx);
p4 = (p1.x, p2.y, p1.z + dx);
```

No caso da face inferior, só muda o y que passa a ser: $y = (-length \div 2)$.

Exemplo: Assumindo que $length = 2$, com as fórmulas referidas acima para o x e o z conseguimos chegar às divisões representadas na figura 6. Neste caso o y para a face superior seria 1 e para a face inferior seria -1 de forma à caixa estar centrada na origem.

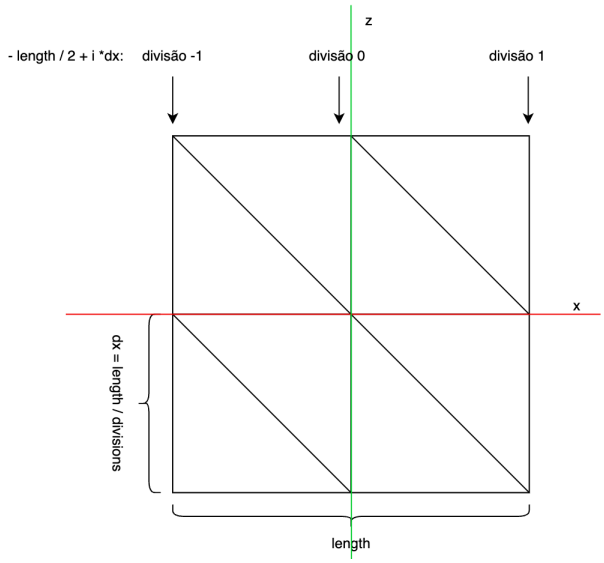

Figura 8: Esquema de uma face da caixa.

Face Traseira e Frontal: Mais uma vez, duas iterações são usadas para percorrer todas as subdivisões na largura e altura da caixa. Desta forma, calcula-se as coordenadas da face traseira com:

p5 =
$(-length \div 2 + k \times dx, -length \div 2 + i \times dx, -length \div 2)$;

```
p6 = (p5.x + dx, p5.y, p5.z);
p7 = (p6.x, p6.y + dx, p6.z);
p8 = (p5.x, p5.y + dx, p5.z);
```

No caso da face frontal, só muda o z que passa a ser:
$y = (length \div 2)$.

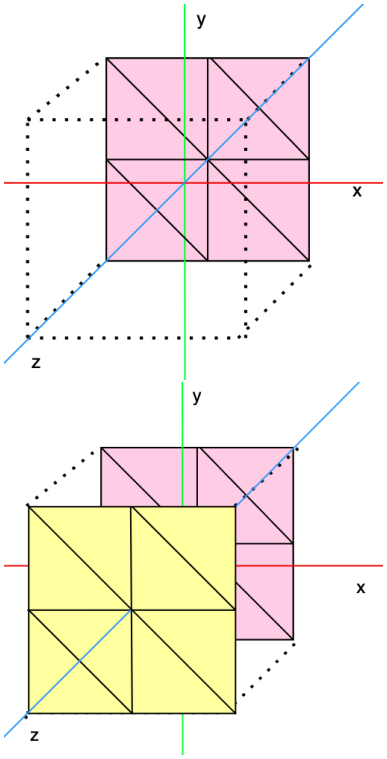

Figura 9: Esquema da face traseira e da face frontal.

Faces da Direita e Esquerda: Mais uma vez, duas iterações são usadas para percorrer todas as subdivisões na profundidade e altura da caixa. Desta forma, calcula-se as coordenadas da face da direita com

p5 =
$(-length \div 2, -length \div 2 + i \times dx, length \div 2 - k \times dx - dx)$;

```
p1 = p5.x, p5.y, p5.z + dx;
p4 = p1.x, p1.y + dx, p1.z;
p8 = p5.x, p5.y + dx, p5.z;
```

No caso da face da esquerda, só muda o x que passa a ser: $x = (length \div 2)$.

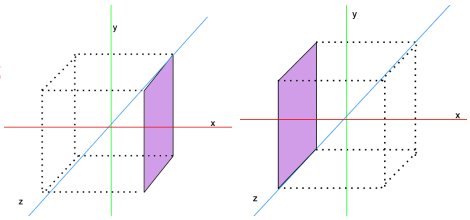

Figura 10: Esquemas das faces da direita e da esquerda.

Como podemos ver, em cada par de faces (frente/trás, direita/esquerda e cima/baixo), apenas uma das coordenadas é modificada para o seu simétrico. Para as faces frontal e traseira, a coordenada alterada é z; para as laterais, é x; e para as superiores e inferiores, é y. Dessa forma, durante o cálculo dos pontos de uma face, podemos simultaneamente obter os pontos da face paralela.

4.1.5 Resultados da caixa:

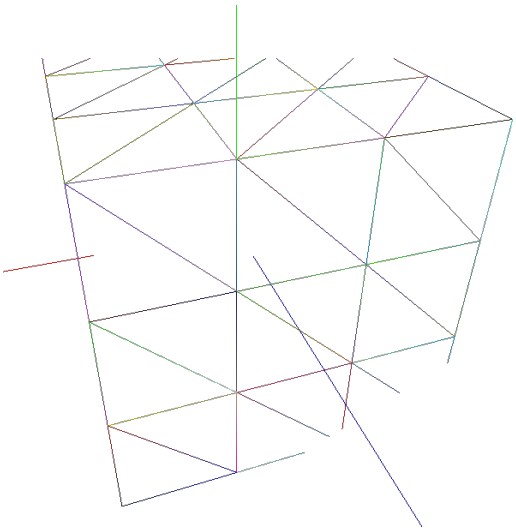

Figura 11: Caixa sem preenchimento.

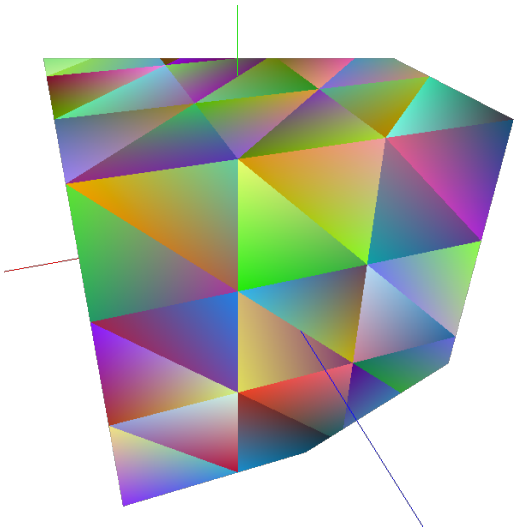

Figura 12: Caixa com preeenchimento.

4.1.6 Cone

Os pontos do cone são calculados utilizando um ângulo *theta* e a distância desse ponto à origem (raio).
Para além disso, serão fornecidos as *slices* e as *stacks*. Cada *slice* define uma seção transversal do cone (divisão horizontal). Visualmente, pode-se imaginar como fatias que cortam o cone perpendicularmente à sua altura. As *stacks* por sua vez, representam a divisão vertical do cone ao longo de sua altura, sendo que cada uma define um segmento vertical do cone, indo da base até o topo.
Iniciando com a base do cone, para os pontos da mesma são utilizados os ângulos θ e θ_2:

- $\theta = \frac{2\times\pi\times j}{slices}$;
- $\theta_2 = \frac{2\times\pi\times(j+1)}{slices}$;

sendo que $j = [0, slices[$. Desta forma serão calculados os pontos p_1, p_2 e p_3, sendo que o ponto p_3 é fixo, pois é o centro da origem do referencial $(0, 0, 0)$.

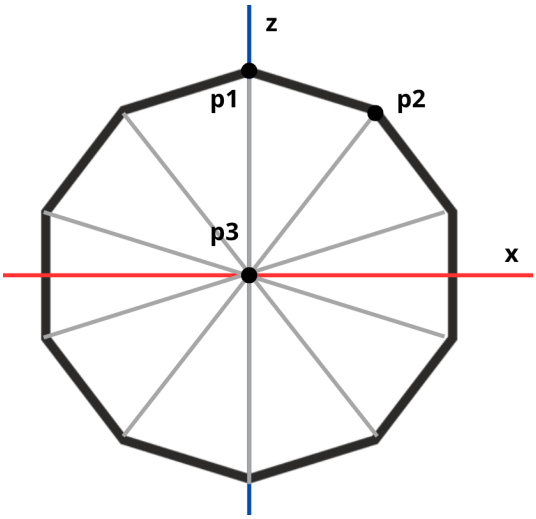

Figura 13: Esquema da base do cone.

Ao longo do *loop*, os ângulos vão variando e são calculados os pontos através de:

- $p1 = (raio \times \sin(\theta), 0, raio \times \cos(\theta))$;
- $p2 = (raio \times \sin(\theta_2), 0, raio \times \cos(\theta_2))$;
- $p3 = (0, 0, 0)$;

Tal como referido anteriormente, os ângulos θ e θ_2 variam entre 0 e 2π radianos e são divididos em *slices* divisões. Assim, foi criado um *loop* e a cada iteração do mesmo, esses ângulos são calculados para determinar as coordenadas dos pontos em cada fatia do cone. A divisão do ângulo é feita dividindo 2π pelo número de *slices*.

A altura total do cone, por sua vez, é dividida em *stacks* divisões (variável *stackHeight*). Desta forma, a cada iteração (de outro *loop*, externo), a altura é ajustada para determinar as coordenadas dos pontos em cada "andar"do cone. A altura é dividida pelo número de *stacks* para garantir uma distribuição uniforme dos pontos ao longo da altura do cone.

A estratégia de desenho dos triângulos consiste em desenhar dois triângulos em cada iteração do *loop* interno (um quadrado).

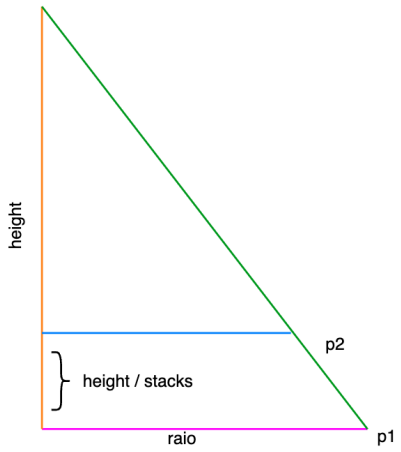

Figura 14: Triângulo referente a figura 7.

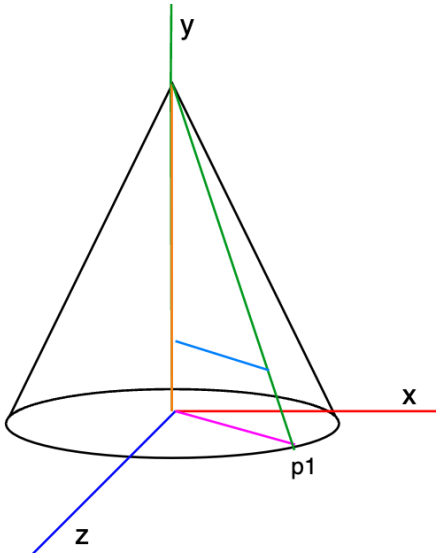

Figura 15: Esquema de um Cone.

$$x = currentRadius \times \sin(\theta) \ ;$$
$$z = currentRadius \times \cos(\theta) \ ;$$

$$currentRadius = raio - i \times deltaRadius;$$
$$deltaRadius = raio \div stacks;$$

O valor do *deltaRadius* é utilizado para calcular o raio de cada esfera dentro do cone (*currentRadius*). Assim, o raio de cada esfera vai diminuindo a medida que "subimos"no cone. Este cálculo garante que a diminuição do raio se distribua uniformemente ao longo da altura do cone.

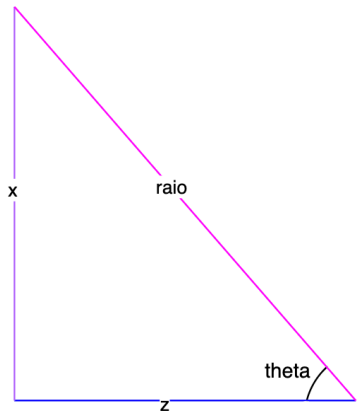

Figura 16: Esquema do triângulo referente à figura 10.

4.1.7 Resultados do cone:

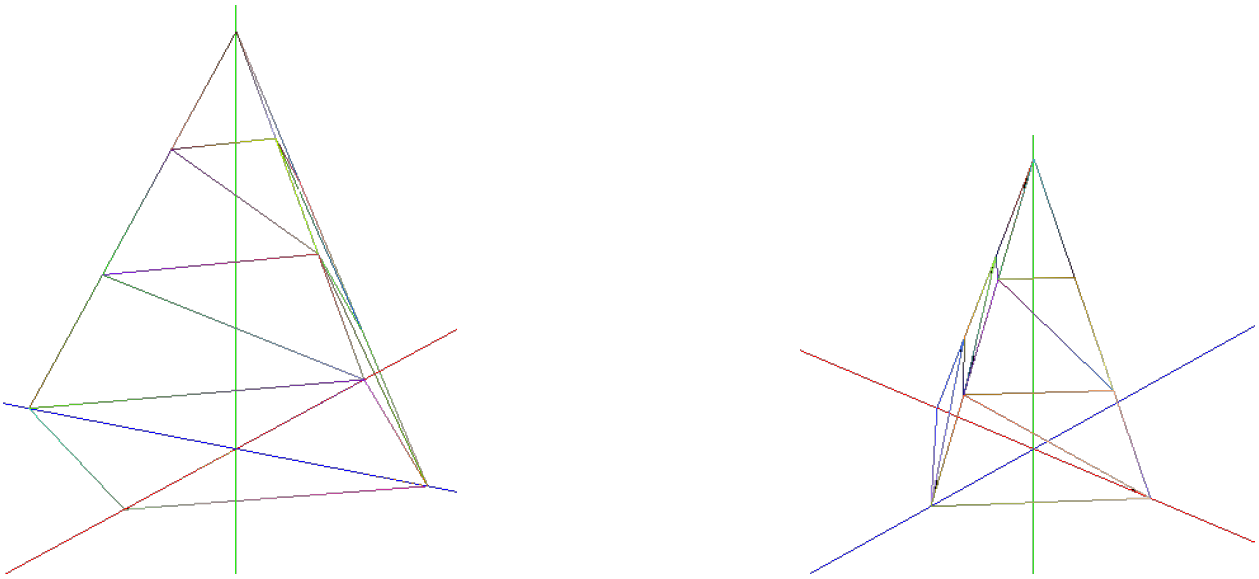

Figura 17: Cones sem preenchimento.

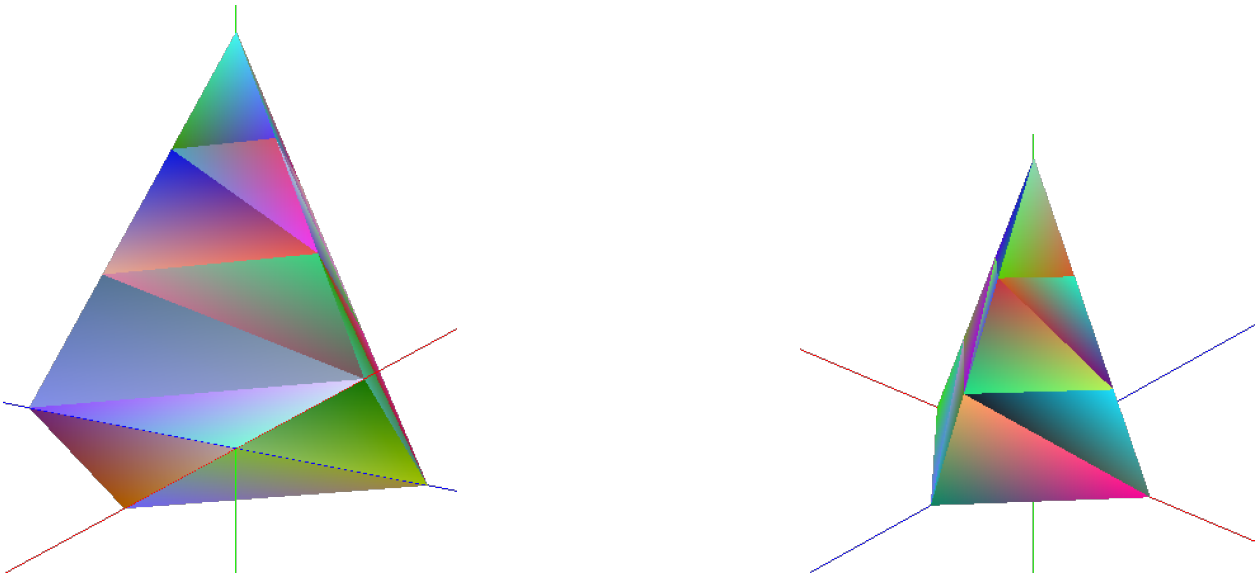

Figura 18: Cones com preenchimento.

4.1.8 Sphere

Para a elaboração de uma esfera precisamos de:

- dois ângulos: α e β
- a distância do ponto à origem (o raio)

Como podemos visualizar na figura 19, o ângulo α varia entre 0 e 2π radianos. O ângulo β, por sua vez, varia entre $-\pi/2$ e $\pi/2$. Para além disso, o ângulo α será dividido em *slices* e o ângulo β em *stacks* que serão fornecidas. Desta forma, sabemos que vamos percorrer *slices* divisões com o ângulo α e *stacks* divisões com o ângulo β.
As divisões horizontais são definidas por:
$\alpha = 2\pi \div slices$
e as divisões verticais são definidas por:
$\beta = \pi \div stacks$
Desta forma, o que altera para cada ponto são os ângulos α e β.

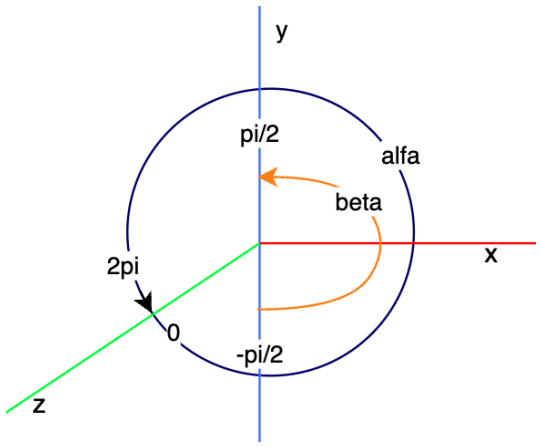

Figura 19: Ângulos utilizados

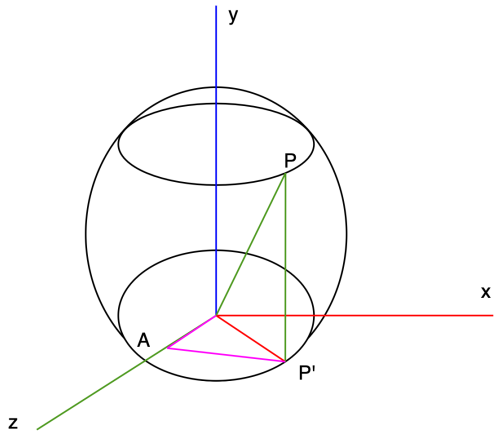

Figura 20: Esquema da esfera e dos triângulos.

Em cada iteração do *loop*, desenhamos dois triângulos que formam uma face da esfera. Desta forma, dentro de um *loop* calcula-se as coordenadas de quatro pontos (p1, p2, p3, p4) que formam dois triângulos adjacentes. Estes compõem uma face da esfera e são usados para definir os vértices dos triângulos.
Desta forma deduzimos as coordenadas do nosso primeiro ponto (p1) P da figura:

$$\begin{aligned} x &= raio \times \cos(\beta) \times \sin(\alpha); \\ y &= raio \times \sin(\beta); \\ z &= raio \times \cos(\beta) \times \cos(\alpha); \end{aligned}$$

Estas coordenadas foram deduzidas pela trigonometria básica:

$$\sin(\beta) = \frac{y}{r} \equiv y = r \times \sin(\beta)$$
$$\cos(\beta) = \frac{r'}{r} \equiv r' = r \times \cos(\beta)$$
$$\sin(\alpha) = \frac{x}{r'} \equiv x = r' \times \sin(\alpha) \equiv x = r \times \cos(\beta) \times \sin(\alpha)$$
$$\cos(\alpha) = \frac{z}{r'} \equiv z = r' \times \cos(\alpha) \equiv z = r \times \cos(\beta) \times \cos(\alpha)$$

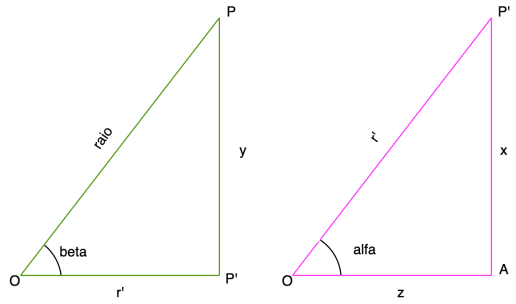

Figura 21: Triângulos referentes à figura 8.

Para calcular as coordenadas dos pontos p2, p3 e p4 em relação ao ponto p1, precisamos de calcular os valores dos seus ângulos. Para o cálculo do ponto p2, em relação ao ponto p1, é alterado o ângulo α aumentando uma unidade na divisão atual. Para o ponto p3, são alterados os dois ângulos, aumentando uma unidade na divisão atual de cada ângulo. Finalmente, para o ponto p4 é alterado o ângulo β aumentando uma unidade na divisão atual do ângulo.
Os triângulos são definidos pelos seguintes conjuntos de pontos:

- p1, p2 e p3
- p1, p3 e p4

Para o cálculo dos ângulos temos que:

- $\alpha = 2\pi \div slices \times a$
- $\beta = \pi \div stacks \times b$

Considerando a = [0,*slices*[e b = [-*stacks*$\div$2, *stacks* $\div$ 2[.
Aplicando as coordenadas esféricas com os diversos valores dos ângulos, obtemos as coordenadas de todos os pontos.

4.1.9 Resultados da esfera:

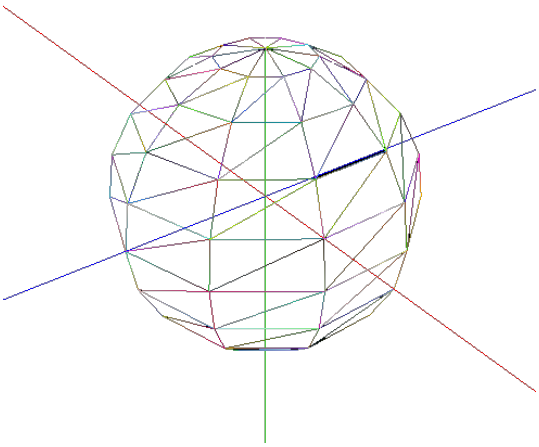

Figura 22: Esfera sem preenchimento.

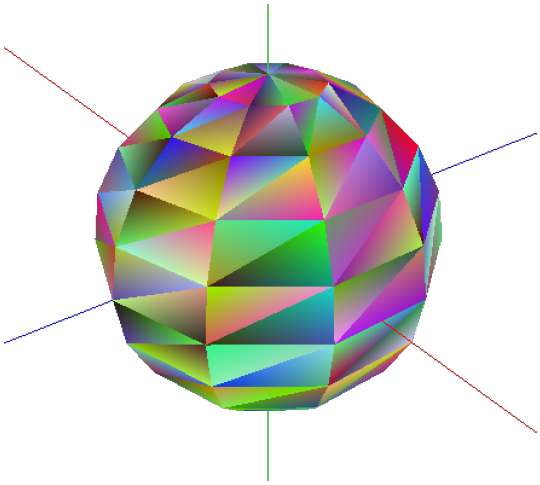

Figura 23: Esfera com preeenchimento.

4.2 Engine

4.2.1 Pasing dos ficheiros XML

Já que a informação sobre os arquivos a serem carregados (arquivos .3d) está contida num ficheiro XML, optamos por usar a biblioteca *TinyXML-2* para analisar esse documento e descobrir os modelos que devem ser desenhados.
Assim, para ler o ficheiro XML, criamos a função *parseXML*. Esta função percorre o ficheiro com o auxílio da biblioteca mencionada anteriormente e vai recolhendo as informações do mesmo. Com os dados recolhidos (*position, lookAt, up e projection*) é preenchida a struct *"Camera"*. Conforme o ficheiro .3d é encontrado, este é armazenado num vetor de strings chamado files `(vector<string> files)` da struct *"World"*.

4.2.2 Pasing dos ficheiros .3d

Depois de receber o nome do ficheiro .3d que contem os nossos pontos na função *parseXML* chamamos a função *readFile*. Nesta função, abre-se o ficheiro .3d e lê-se linha a linha. Em cada linha, divide-se os triângulos usando o caracter '/' como delimitador. Para cada substring, são analisadas as coordenadas x, y e z dos pontos do triângulo, usando o caracter ';' como delimitador. Finalmente, os pontos x, y e z, extraídos de cada triângulo, são armazenados na struct *"Point"*. Este objeto é adicionado ao vetor *points* `(vector <Point> points)` da struct *"World"*.

4.2.3 Desenho das primitivas

Com o objetivo de desenharmos as primitivas elaboradas anteriormente criou-se a função "*drawPrimitives*". Assim, esta função permite-nos desenhar os triângulos das primitivas usando a biblioteca *OpenGL*. Depois de iniciar o processo com o *glBegin(GL_TRIANGLES)* é feito um loop que percorre todos os pontos do `(vector <Point> points)` da struct *"World"*. Sendo que cada ponto é representado por um objeto *Point* com coordenadas x, y e z. Através do *glColor3f* define-se a cor do próximo vértice a ser desenhado. A cor é gerada aleatoriamente usando a função *rand*. Para além disso, é utilizado o *glVertex3f(pt.x, pt.y, pt.z)* de forma a especificar a posição do próximo vértice a ser desenhado.

4.2.4 Câmara

Nesta parte do projeto foram utilizadas várias funções fornecidas nas aulas práticas. Assim, a função *renderScenes*, atravês da função *gluLookAt* define:

- a posição da câmara,
- a posição do ponto de referência que a câmara está a focar,
- o vetor *up*.

Para além disso, calcula e define-se a matriz de visualização corrente, que é usada para posicionar a câmara no mundo 3D. Através da *glPolygonMode* os modelos serão desenhados apenas com linhas ou preenchidos com cor em função do escolhido(variável *linha*). Para além disso, foi criada a função *drawAxes* para desenhar os eixos X, Y e Z.
Quando a janela é redimensionada, a função *changeSize* é chamada e efetua as operações necessárias para ajustar a projeção de acordo com as novas dimensões da janela. Desta forma, a função *changeSize* foi criada para garantir e proporcionar uma experiência de visualização consistente e adequada.
É também importante referir que os parametros da câmara são recolhidos na função *parseXML* referida anteriormente. Para cada elemento (*position, lookAt, up, projection*), a função extrai os valores dos atributos x, y, e z e armazena esses valores nos membros correspondentes da estrutura *Câmara* dentro da estrutura *World*. Assim, essa parte do código é responsável por coletar as informações da câmara do mundo virtual a partir do arquivo XML fornecido.

4.2.5 Opções de teclado

Realizamos algumas funcionalidades para lidar com eventos de teclado. Estas funcionalidades são especificadas pela função *processKeys*, e posteriormente validadas com a *glutKeyboardFunc* do GLUT. Dentro da função *processKeys*, foi criado um switch-case que verifica a tecla que foi pressionada e executa uma ação correspondente. Iremos explicar cada opção:

- **Tecla L:**

  Em vez de visualizar a primitiva com as faces dos triângulos preenchidas com cores aleatórias, é possível fazer com que apenas se visualizam as linhas. Para ativar ou desativar esta opção basta clicar na tecla L.

- **Tecla A:**

  O *Engine* permite utilizar um sistema de eixos tridimensional para facilitar a visualização das primitivas. Como referido anteriormente, para desenhar os eixos foi implementada a *drawAxes*. Para ativar ou desativar a opção de apresentar os eixos, basta clicar na tecla A.

- **Teclas I e O:**

  A câmara tem a possibilidade de se movimentar devido a função *gluLookAt*. Desta forma, para aproximar ou afastar a câmara da origem, basta clicar em I ou O, respetivamente, aumentando e diminuindo o *radius* de uma unidade.

- **Teclas UP, DOWN, LEFT, RIGHT:**

  A função *processSpecialKeys* permite-nos mover a câmara para cima, para baixo, para a direita ou esquerda da figura. Para tal, basta pressionar a tecla UP (para aumentar o ângulo beta), DOWN (para diminuir o ângulo beta), RIGHT (para aumentar o ângulo alpha) ou LEFT (para diminuir o ângulo alpha), respectivamente.

5 Conclusão

O grupo considera que a realização desta fase inicial foi bem-sucedida, visto que todas as funcionalidades requisitadas foram implementadas conforme o enunciado. Essa etapa foi crucial para o desenvolvimento subsequente do projeto, pois proporcionou uma compreensão ampla do funcionamento geral do trabalho com modelos 3D e facilitou a manipulação desses modelos por meio das bibliotecas utilizadas. Além disso, contribuiu significativamente para a familiarização com a linguagem de programação e a construção de figuras que serão úteis nas próximas fases.

Embora a realização desta fase tenha sido relativamente bem-sucedida de uma perspetiva geral, há aspetos que poderiam ter sido melhorados. Embora os programas desenvolvidos tenham fornecido as funcionalidades básicas esperadas, queríamos ter conseguido implementar mais algumas funcionalidades opcionais como o o toro e o cilindro.

Em resumo, a realização desta fase inicial é fundamental para o progresso futuro do projeto. Isso tornará a próxima tarefa relativamente mais simples de realizar e perceber, uma vez que já estamos familiarizados com as ferramentas e a linguagem utilizadas.