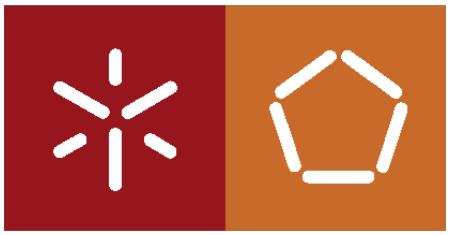

Computação Gráfica

MIEI - 3º ano - 2º semestre
Universidade do Minho

Trabalho Prático - Fase 1

Grupo 11


João Vieira
A76516


Manuel Monteiro
A74036


Bernardo Viseu
A74618


Fernando Pereira
A75496

March 8, 2019

Conteúdo

1 Introdução

1.1 Contextualização

No contexto curricular da disciplina de Computação Gráfica, foi nos proposto a criação de um mecanismo 3D com base num cenário gráfico, tendo por base todos os conhecimentos adquiridos até ao momento nas aulas e utilizando a linguagem de programação C++. Este trabalho prático está dividido em quatro fases, sendo este relatório referente à primeira fase, que tem como objetivo a conceção de algumas primitivas gráficas.

1.2 Resumo

Para esta primeira fase, foi feita a criação de duas aplicações necessárias para o funcionamento do projeto, Gerador e Motor.

O Gerador (*Generator*) é responsável por gerar a informação de cada modelo, guardando os seus vértices num ficheiro específico.

O Motor (*Engine*) tem a função de ler a configuração de um ficheiro XML e exibir os respetivos modelos.

Os modelos gráficos que serão realizados nesta fase são o Plano, Paralelepípedo, Esfera e o Cone. Para além destes, o grupo também fez a elaboração do Cilindro, de maneira a enriquecer o trabalho prático.

2 Arquitetura

Após a análise dos objetivos do trabalho, o grupo de trabalho decidiu estruturar o código do projeto implementando duas principais aplicações, Gerador e Motor. Também foram criadas estruturas para armazenar os vértices de cada modelo, de modo a permitir gerar todos os pontos necessários para cada primitiva.

2.1 Aplicações

Nesta secção são apresentadas as aplicações que permitem realizar a geração e exibição de todos os modelos disponíveis.

2.1.1 Gerador

generator.cpp: Nesta aplicação estão definidas as estruturações das várias primitivas geométricas, de maneira a gerar os respetivos vértices. Cada conjunto de vértices é armazenado num ficheiro 3d.

2.1.2 Motor

engine.cpp: O motor implementado em OpenGL, é o responsável pela visualização de todo o projeto. Este realiza a apresentação de uma janela que exibirá os modelos lidos do ficheiro XML, e também permite toda a sua manipulação através de comandos (teclado).

2.2 Classes

Para facilitar o desenvolvimento e construção das aplicações referidas, decidiu-se criar duas classes: a classe **Vertex** que representa cada ponto, e a classe **Shape** que representa cada forma gerada, ou seja, o seu conjunto de pontos.

2.2.1 Vertex

Vertex.cpp: Classe que guarda um ponto com coordenadas (x,y,z) necessário para o desenho de cada triângulo (3 pontos).

2.2.2 Shape

Shape.cpp: Classe que define a estrutura que guarda o conjunto de pontos de uma determinada forma geométrica, através de um $vector < Vertex* >$.

2.2.3 Models

models.cpp: Classe que contém todos os algoritmos necessários para gerar cada forma geométrica, manipulando o seu conjunto de pontos.

2.2.4 TinyXML2

tinyxml2.cpp: *Parser* em TinyXML escolhido para fazer a leitura dos ficheiros XML e tratamento do seu conteúdo.

3 Primitivas Geométricas

3.1 Plano

Para o plano são apenas necessários 4 triângulos (dois virados para cima, e dois virados para baixo). Sendo assim apenas necessitamos de saber a dimensão do plano.

3.1.1 Algoritmo

Primeiro, o plano a ser definido é centrado na origem, contido no plano XZ. Percebemos então que a variável y é sempre 0.

Como foi referido acima apenas vamos precisar 4 triângulos, mas para além disso, cada par triângulos vai partilhar 2 pontos, que neste caso a reta entre os 2 forma a diagonal do plano. Para chegarmos aos pontos apenas dividimos o tamanho do plano por 2 e obtivemos os pontos: $x = z = size/2$, e o seu simétrico. Para os outros 2, apenas tivemos de trocar os sinais dos valores de x e z, obtendo: $x = -size/2, z = size/2$ e vice versa.

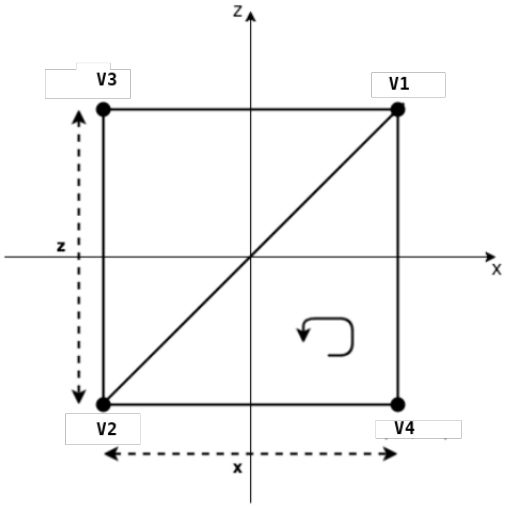

Figura 1: Ilustração da construção do plano

Por fim apenas tivemos de ter atenção a ordem de criação dos vértices para os triângulos serem desenhados, e inverter a ordem para vermos o plano pela parte de baixo.

3.1.2 Modelo3D

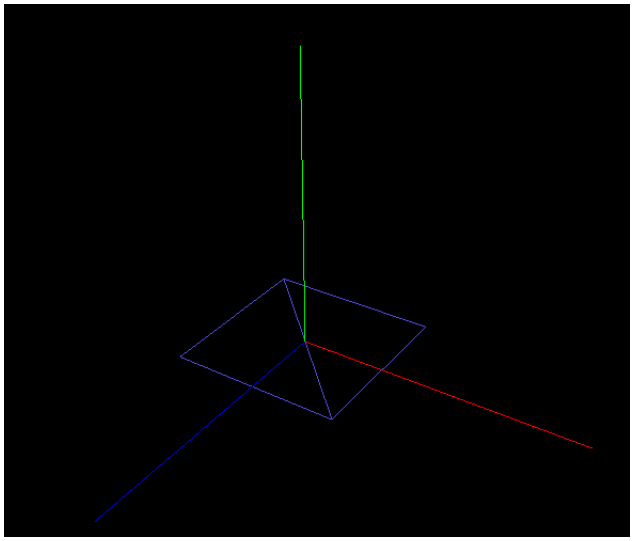

Figura 2: Modelo final do Plano

3.2 Paralelepípedo

Um paralelepípedo é um prisma de 6 faces, e como tal pode ser interpretado como um conjunto de triângulos. Para ser formado, necessitamos de uma largura (x), altura (y) e comprimento (z) e o número de divisões de cada face.

3.2.1 Algoritmo

Como foi dito acima, o Paralelepípedo é um prisma de 6 faces, como tal apenas tivemos de repetir o processo acima descrito, para cada uma das faces. O que realmente é diferente aqui são as divisões que vamos fazer a cada uma das faces. Para calcular os pontos intermédios do plano, primeiro temos que dividir o número de divisões a ser feita por cada uma das faces do prisma, obtendo assim a distância do próximo ponto da divisão da face. Apresentamos então a seguinte imagem para descrever o algoritmo, chamando a atenção de que $varShift = var/ndiv$ tendo $ndiv = 2$:

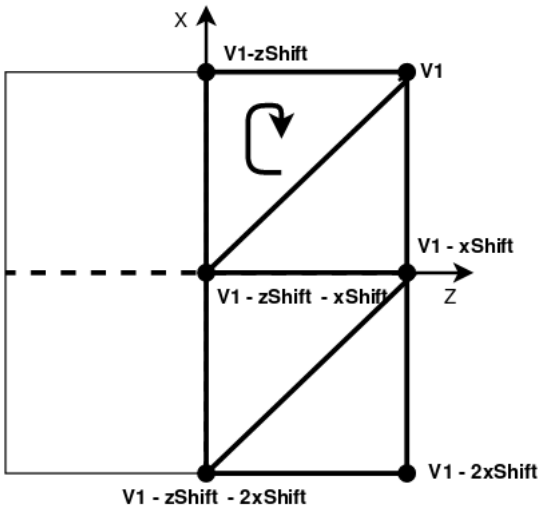

Figura 3: Ilustração da construção do plano do Paralelepípedo

Podemos verificar que para desenhar os triângulos dos quadrados pequenos, começando no vértice $V1$ apenas temos de deslocar o ponto, subtraindo $Shift$ à coordenada que vai sofrer o deslocamento. Lembramos que tal como o plano,há sempre uma variável que se mantém constante, tendo-se que deslocar apenas 2 das 3 variáveis.

Já em termos de código, podemos ver as divisões (quadrados pequenos) de cada plano como uma entrada de uma matriz. Sendo assim, aplicamos 2 ciclos aninhados para aplicar o algoritmo às "linhas"e "colunas"de todas as faces. No caso da ilustração nota-se que quando temos $2varShift$ é realmente a 2^a iteração do ciclo.

Por fim é necessário ter atenção à ordem que se define os pontos, de maneira a apresentar de forma correta, as faces do prisma. No caso da ilustração, como o plano XZ só deve ser visto por baixo, é desenhado da forma apresentada.

3.2.2 Modelo3D

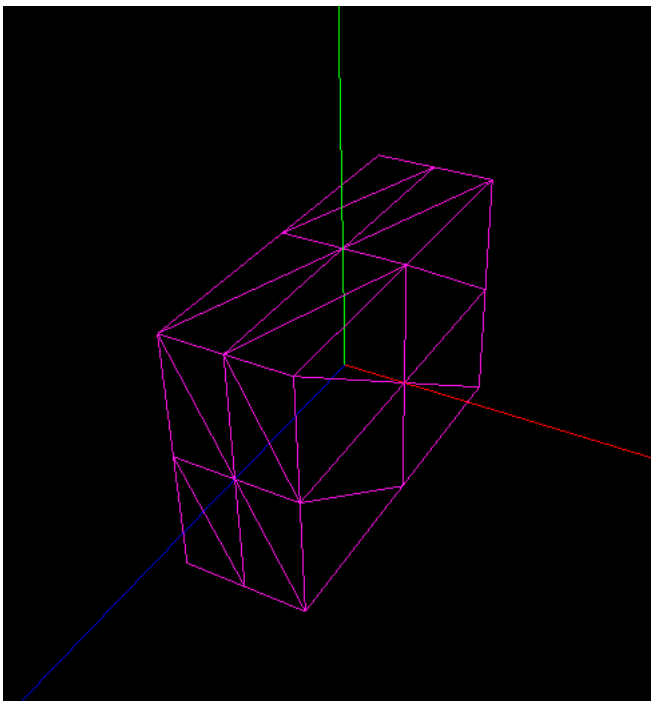

Figura 4: Modelo final do Paralelepípedo

3.3 Esfera

A esfera é um sólido geométrico formado por uma superfície curva contínua formada por pontos equidistantes do seu centro. Para a criação de uma esfera é necessário definir os parâmetros r (raio), $nfatias$ (número de fatias) e $ncamadas$ (número de camadas).

3.3.1 Algoritmo

Antes de prosseguirmos ao desenho dos pontos, temos de perceber o conceito da esfera. Este sólido, é definido por dois ângulos e o seu raio. Um dos ângulos, dando o nome de α é um ângulo giro (2π) que nos dá o deslocamento horizontal da esfera. O segundo,chamando-lhe de β, é um ângulo raso(π) que nos indica o deslocamento vertical da esfera:

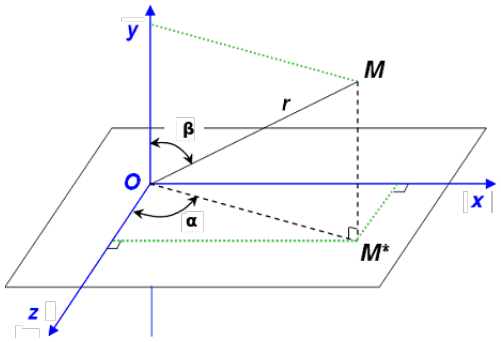

Figura 5: Ilustração dos ângulos que constituem a esfera

Tendo os ângulos, temos agora de dividi-los pelo número de fatias e número de camadas, respetivamente, obtendo assim o deslocamento ($Shift$) horizontal e vertical:

$$\alpha Shift = 2\pi / nfatias$$
$$\beta Shift = \pi / ncamadas$$

Com estes valores de ângulos, temos de converter as coordenadas esféricas, para coordenadas cartesianas, para definir os pontos ficando assim com:

$$x = r * \sin\alpha * \sin\beta$$
$$y = r * \cos\beta$$
$$z = r * \cos\alpha * \sin\beta$$

De seguida percebemos que para cada camada, iremos fazer as fatias todas da mesma e só depois avançamos para a camada seguinte. Por cada fatia, na camada, apenas temos de desenhar dois triângulos. Para isso utilizamos ambos os deslocamentos calculados:

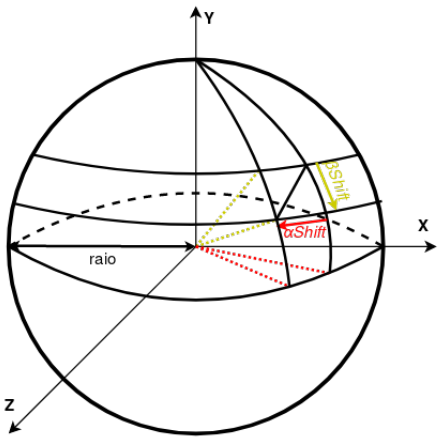

Figura 6: Ilustração da construção da esfera

Agora fazendo para a esfera toda, basta iterar por cada camada, por todas as fatias, aumentando o ângulo inicial (0°) em cada iteração. Depois de finalizar todas as fatias, apenas tem de aumentar o ângulo (β) para refazer o processo de todas as fatias, não esquecendo a ordem de criação dos pontos para apresentar os triângulos de forma correta.

3.3.2 Modelo3D

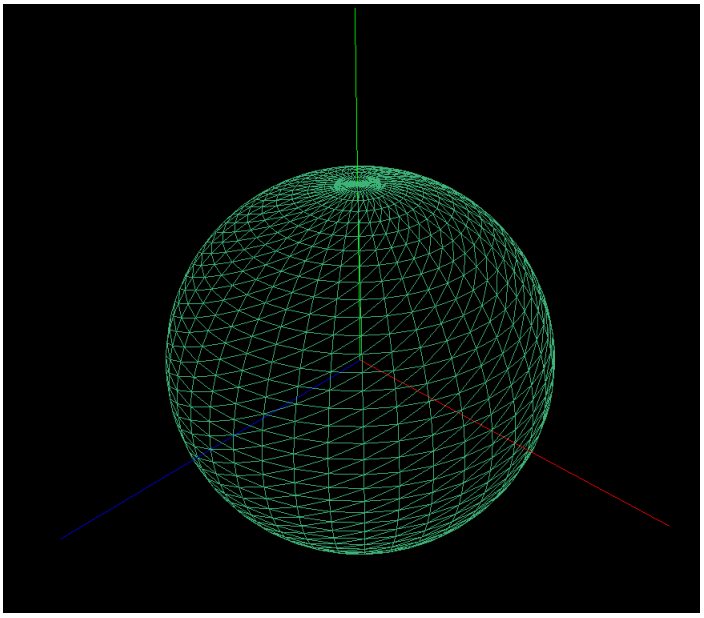

Figura 7: Modelo final de uma esfera

3.4 Cone

O cone é o sólido geométrico que pode ser visto como uma pirâmide, mas em que a sua base é formada por uma circunferência. Mas o cone também pode ser visto como um cilindro, mas com uma diferença, é que no cone, o raio, desde a base até à altura, diminui. Sendo assim, para desenhar o cone temos os parâmetros de raio (r), altura (h), número de fatias ($nfatias$) e número de camadas ($ncamadas$).

3.4.1 Algoritmo

Como foi dito em cima, o cone pode ser visto como um cilindro, pois tem o um ângulo de 2π, que é o que forma a sua curvatura. Então calculamos o deslocamento horizontal, bastando dividir o ângulo pelo número de fatias:

$$\alpha Shift = 2\pi / nfatias$$

Comecemos a desenhar o cone pela sua base. Para isso apenas necessitamos do raio e do deslocamento horizontal. Então percebemos que a base é formada, apenas por um triângulo por fatia, tendo todos os triângulos um vértice em comum, o centro da base. Posto isto apenas temos de iterar pelo número de faias, lembrando que, a base ao ser desenhada no plano XZ, a posição y dos pontos é constante e que a ordem de criação dos pontos foi escolhida de maneira a que a base fosse desenhada quando se estivesse a ver por baixo. Aumentando o ângulo em cada iteração, conseguimos de desenhar a sua base, faltando apenas converter as variáveis x e z de cilíndricas para cartesianas obtendo:

$$x = r * \cos \alpha$$
$$z = r * \sin \alpha$$

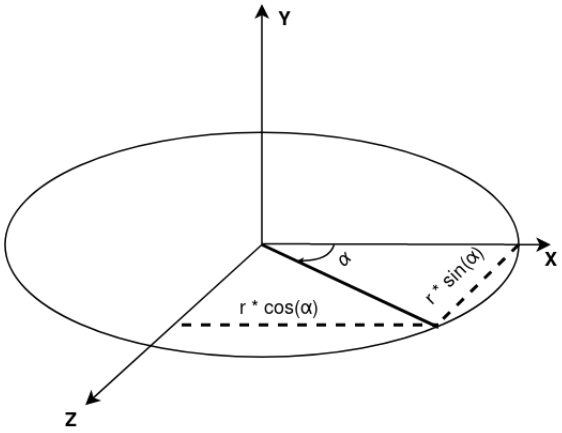

Figura 8: Ilustração da construção da base do cone

Depois o que define o cone é a sua altura e a diminuição do raio. Posto isto, para gerar o resto dos pontos do cone verificamos que temos de efectuar o mesmo processo, para encontrarmos os deslocamentos, feito na esfera. Só que a diferença é que o cone não tem um ângulo de rotação vertical, logo para chegar a esse deslocamento temos de, primeiro, dividir a altura do cone pelas camadas. Mas fazendo isto só obtemos na verdade a variável y do ponto. Reparamos que a diminuição do raio é o factor que nos vai dar as coordenas x e z da seguinte na camada. Então para encontrarmos o factor de diminuição do raio só temos de dividir o raio do cone pelas camadas, obtendo:

$$rShift = r/ncamadas$$
$$\beta Shift = h/ncamadas$$
$$y = \beta Shift$$

Desenhando 2 triângulos em cada fatia de uma camada e tendo atenção à ordem de criação dos pontos obtemos o cone.

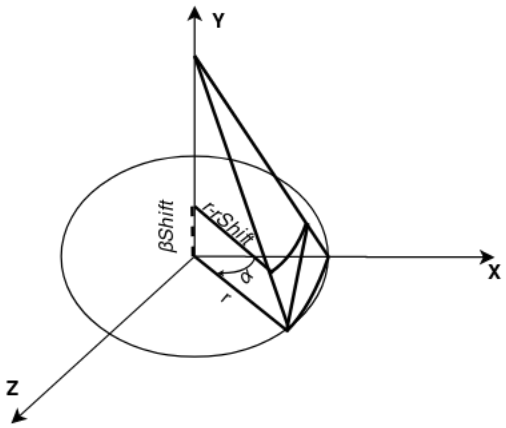

Figura 9: Ilustração da construção do cone

3.4.2 Modelo3D

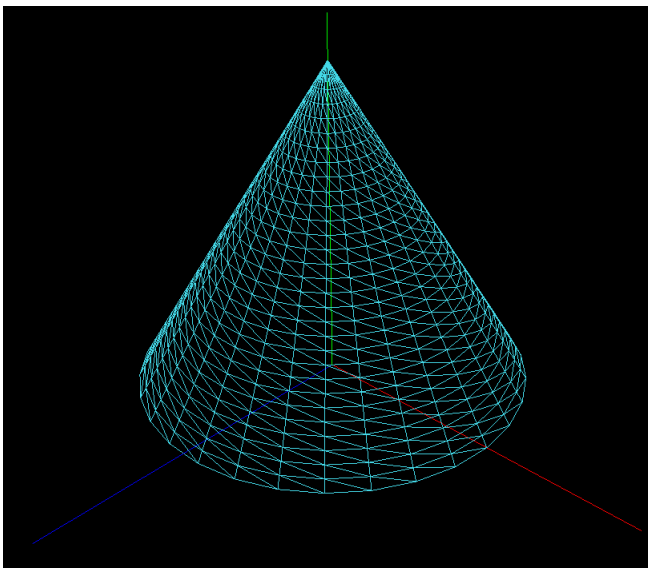

Figura 10: Modelo final de um cone.

3.5 Cilindro (extra)

O cilindro é um sólido geométrico formado por dois círculos (topo e base), e entre o topo e a base existe um alongamento circular, com o mesmo raio. Para a criação de um cilindro é necessário definir os parâmetros r (raio), $height$ (altura) e $nfatias$ (número de fatias).

3.5.1 Algoritmo

Para se desenhar o cilindro definimos um ângulo, aqui denominado por α, que vai ter o valor:

$$\alpha = 2\pi/nfatias$$

E também um outra ângulo β inicializado a 0. O valor deste ângulo vai aumentar segundo o deslocamento na circunferência (topo ou base) de raio dado. Com isto, é possível definir um cilindro desenhando primeiro o circulo do topo, que vai estar centrado no ponto $(x, y, z) = (0, height/2, 0)$. Queremos que este circulo fique orientado para cima, por isso a ordem dos pontos é a seguinte:

$$\begin{aligned}
&Ponto1 : (x, y, z) = (0, height/2, 0)\\
&Ponto2 : (x, y, z) = (r * \cos(\beta + \alpha), height/2, r * \sin(\beta + \alpha))\\
&Ponto3 : (x, y, z) = (r * \cos(\beta), height/2, r * \sin(\beta))
\end{aligned}$$

Depois de desenhado esta fatia do circulo, desenhamos também um triângulo que vai ser metade do alongamento desta fatia para o circulo base, para isto podemos usar dois dos pontos usados para desenhar o triângulo referido anteriormente:

$$\begin{aligned}
&Ponto1 : (x, y, z) = (r * \cos(\beta + \alpha), height/2, r * \sin(\beta + \alpha))\\
&Ponto2 : (x, y, z) = (r * \cos(\beta + \alpha), -height/2, r * \sin(\beta + \alpha))\\
&Ponto3 : (x, y, z) = (r * \cos(\beta), height/2, r * \sin(\beta))
\end{aligned}$$

Depois disto realizamos o mesmo para a parte inferior do cilindro, ou seja, um circulo centrado em $(x, y, z) = (0, -height/2, 0)$ (os pontos são equivalentes aos do circulo do topo), e a outra metade de cada fatia do alongamento do cilindro, com os seguintes pontos:

$$Ponto1 : (x, y, z) = (r * \cos(\beta), height/2, r * \sin(\beta))$$
$$Ponto2 : (x, y, z) = (r * \cos(\beta + \alpha), -height/2, r * \sin(\beta + \alpha))$$
$$Ponto3 : (x, y, z) = (r * \cos(\beta), -height/2, r * \sin(\beta))$$

Depois de desenhada esta fatia completa do cilindro, repete-se isto $nfatias$ vezes, adicionando consecutivamente a β o valor de α.

3.5.2 Modelo3D

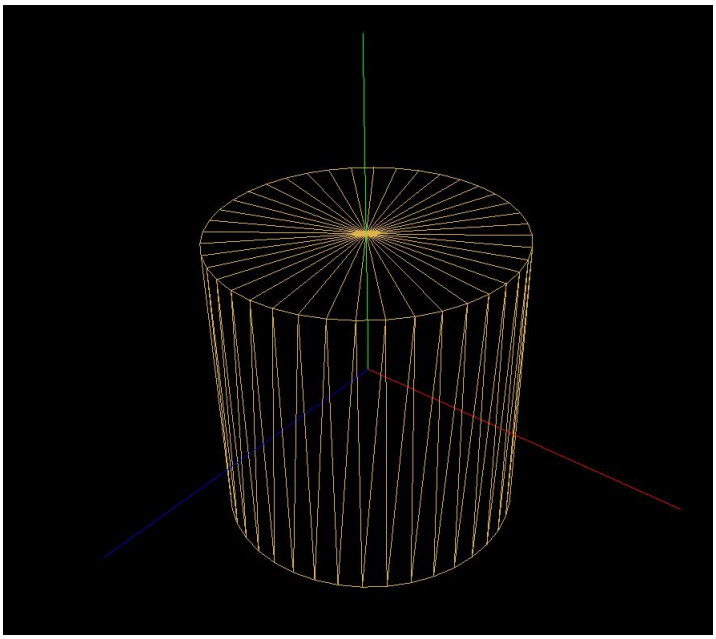

Figura 11: Modelo final de um cilindro.

4 Gerador

4.1 Descrição

O gerador, como já foi referido antes, trata de gerar o ficheiro com os conjuntos de vértices para a forma geométrica pretendida, e de acordo com os parâmetros inseridos.

4.2 Utilização

Em termos de usabilidade do gerador, temos as seguintes instruções disponibilizadas no seu manual (comando ./generator) com os vários parâmetros e opções para cada modelo geométrico:

```
#####################################################
#                  Generator MENU                   #
#     Usage:                                        #
#     ./generate <shape> [options] <file>           #
#                                                   #
#     Shapes & Options:                             #
#        -> plane <size>                            #
#        -> box <width> <height> <length> <divisions> #
#        -> sphere <radius> <slices> <stacks>       #
#        -> cone <radius> <height> <slices> <stacks>  #
#        -> cylinder <radius> <height> <slices>     #
#####################################################
```

Figura 12: Manual do Gerador

5 Motor OpenGL

5.1 Descrição

Como foi referido em cima, o Motor é responsável pela leitura dos ficheiros XML, construindo graficamente os seus respetivos modelos. Na sua implementação tivemos o auxílio da API do OpenGL.

5.2 Utilização

Na imagem apresentada abaixo podemos observar o manual de ajuda do engine, que é acedido através do comando ./engine (sem argumentos).

```
##############################
#         Engine MENU        #
#    Usage:                  #
#    ./engine path_to_XML    #
#                            #
#    KeyBinds:               #
#       w - Rotate up        #
#       s - Rotate down      #
#       a - Rotate left      #
#       d - Rotate right     #
#                            #
#       j - Fill Mode        #
#       k - Line Mode        #
#       l - Point Mode       #
#                            #
#      '-' - Move Cam Back   #
#      '+' - Move Cam In     #
#                            #
#       m - Make axis longer #
#       n - Make axis smaller #
#                            #
#       c - Reset colors     #
##############################
```

Figura 13: Manual do Engine

5.3 Comandos

Como é facilmente percetível pela leitura do Manual do Engine, os comandos **a**, **s**, **d** e **w** servem para rodar os modelos apresentados, sendo assim possível obter vários pontos de vista sobre as figuras geométricas.

Os comandos m e n servem para aumentar e diminuir os eixos do plano, respetivamente.

Ao usar os comandos + e **-**, a câmara aproxima-se e afasta-se das figuras, respetivamente.

Já os comandos **j**, **k** e **l** permitem ter diferentes perspetivas dos modelos, como é possível ver nas imagens abaixo:

Preenchido (comando j)

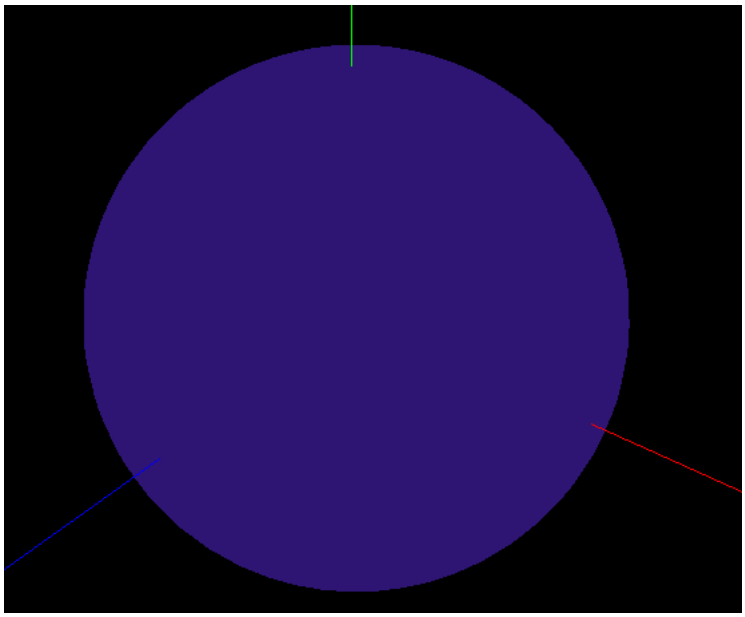

Figura 14: Modelos preenchidos

Linhas (comando k)

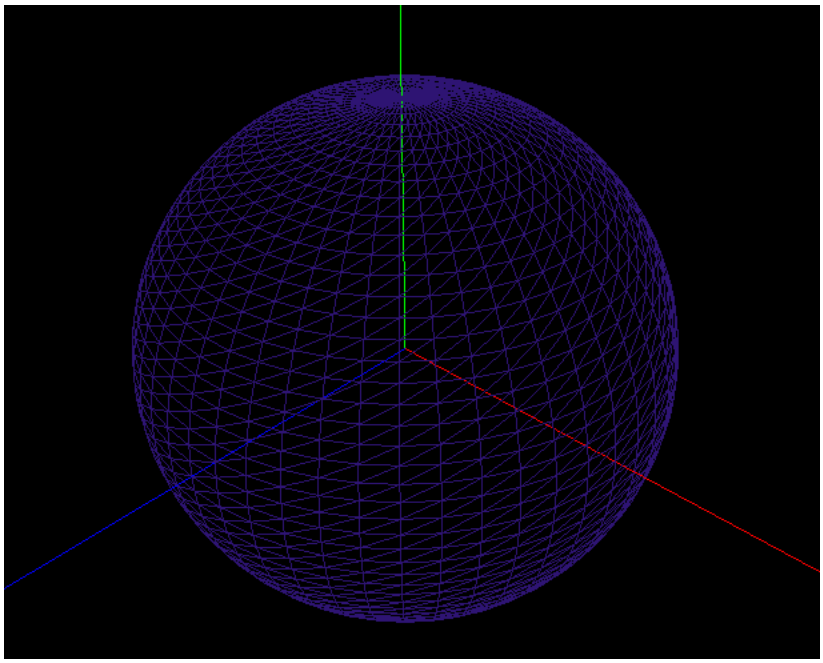

Figura 15: Modelos apresentados com linhas

Pontos (comando l)

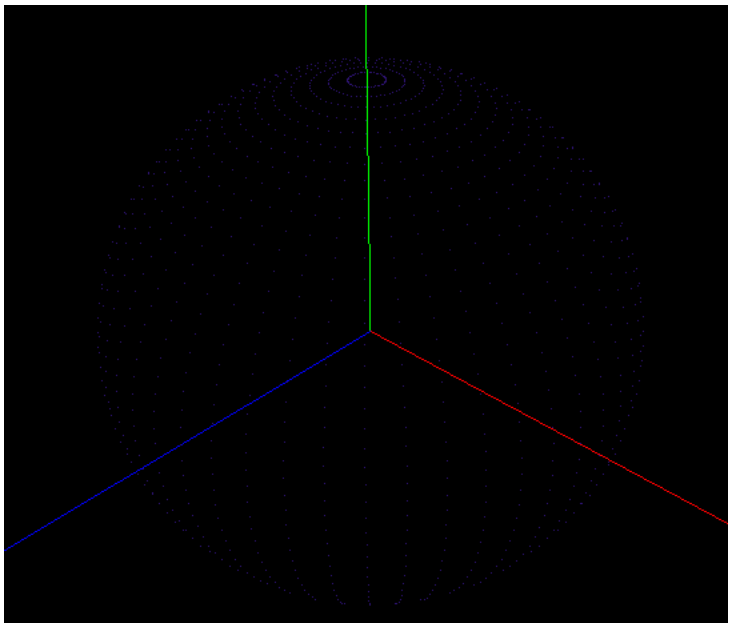

Figura 16: Modelos apresentados com pontos

6 Conclusão

Concluída a primeira fase do trabalho que nos foi proposto, consideramos que todo o processo foi essencial para colocarmos em prática e desenvolvermos conhecimentos na linguagem de programação C++. Também foi importante no que toca à aprendizagem da utilização das ferramentas associadas à computação gráfica que foram o OpenGL e GLUT. Sentimos que a realização desta primeira etapa motivou-nos e serviu como rampa para as restantes etapas e estamos ansiosos para começar a trabalhar na segunda parte do projeto.