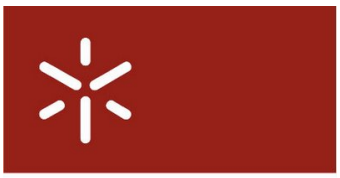


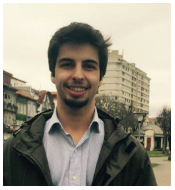
Universidade do Minho

Computação Gráfica

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

Trabalho Prático - Parte 1


Guilherme Guerreiro
A73860


Dinis Peixoto
A75353


Ricardo Pereira
A74185


Marcelo Lima
A75210

22 de Maio de 2017

Conteúdo

1. *Introdução*

1.1 Contextualização

Foi-nos proposto, no âmbito da UC Computação Gráfica, a criação de um mini mecanismo 3D baseado num cenário gráfico sendo que para isso teríamos de utilizador várias ferramentas apresentadas nas aulas práticas entre as quais C++ e OpenGL.
Este trabalho foi dividido em quatro partes, sendo esta a primeira fase que tem como objetivo a criação de algumas primitivas gráficas.

1.2 Resumo

Nesta primeira parte do projeto prático foi necessário a criação de duas aplicações que são essenciais para o seu funcionamento:

- Gerador (*Generator*) - Gerar a informação essencial dos modelos guardando os vértices num ficheiro específicado.
- Motor (*Engine*) - Ler a configuração de um ficheiro XML e exibir os modelos pretendidos.

As primitivas gráficas que serão elaboradas nesta fase são o Plano, Paralelipípedo, Esfera e o Cone. Além destas o grupo decidiu explorar as suas capacidades no âmbito da Computação Gráfica, acabando por desenvolver também um Cilindro.
O objectivo desta fase passa por gerar e exibir estas primitivas gráficas em GLUT.

2. *Arquitetura do código*

Com a análise geral do problema e das várias opções de desenvolvimento, decidimos implementar duas aplicações principais em C++ que nos dispõe uma maior simplicidade na execução das tarefas propostas nesta primeira fase do trabalho prático - **gerador** e **engine** - e as estruturações dos modelos que nos permitem gerar os diversos vértices para a sua exibição - **plane**, **box**, **sphere**, **cone** e **cylinder**.

2.1 Aplicações

Nesta secção são apresentadas as aplicações fundamentais que permitem gerar e exibir os diferentes modelos disponíveis.

2.1.1 Gerador

generator.cpp - Aplicação onde estão definidas as estruturações das diferentes formas geométricas a desenvolver de forma a gerar os respetivos vértices. Esta, apesar de não tão importante, é complementar ao motor.

2.1.2 Motor

engine.cpp - Aplicação que possui as funcionalidades principais. Permite a apresentação de uma janela exibindo os modelos pretendidos e ainda a interação com estes através de comandos que veremos mais adiante.

2.2 Classes

De modo a simplifciar toda a construção das aplicaçãoes em cima referidas, o grupo decidiu criar duas classes de modo a evitar a complexidade que as estruturas em C exigem. Assim, o grupo optou pela criação da classe **Vertex** representativa de cada ponto, e consequentemente a classe **Shape** representativa de cada forma gerada, isto é, um conjunto de pontos.

2.2.1 Vértice/Ponto

Vertex.cpp - Classe que guarda um ponto necessário para a constituição de um triângulo, através da definição das suas coordenadas (x,y,z).

2.2.2 Forma

Shape.cpp - Classe que guarda todo o conjunto de pontos necessários à representação de um determinado modelo, contendo, desta maneira, um *vector<Vertex*>*, ou seja, um conjunto de *Vertex* (vértices).

2.3 Formas

Conjunto de ficheiros constituídos por algoritmos necessários para a criação dos vértices posteriormente utilizados para as diversas formas geométricas disponíveis.

2.3.1 Plano

Plane.cpp - Algoritmo que nos permite obter os vértices necessários para a criação de um plano.

2.3.2 Paralelipípedo

Box.cpp - Algoritmo que nos permite obter os vértices necessários para a criação de um paralelipípedo.

2.3.3 Esfera

Sphere.cpp - Algoritmo que nos permite obter os vértices necessários para a criação de uma esfera.

2.3.4 Cone

Cone.cpp - Algoritmo que nos permite obter os vértices necessários para a criação de um cone.

2.3.5 Cilindro

Cylinder.cpp - Algoritmo que nos permite obter os vértices necessários para a criação de um cilindro.

2.4 Extras

2.4.1 TinyXML2

tinyxml2.cpp - Ferramenta utilizada para fazer o *parsing* dos ficheiros XML de modo a explorar os ficheiros do seu conteúdo.

3. *Primitivas geométricas*

3.1 Plano

Um plano é composto por dois triângulos que partilham dois vértices entre eles. As suas dimensões são dadas pelos valores de X e Z. O plano que se forma está contido no plano XZ, e centrado na origem.

3.1.1 Algoritmo

Para que se possa observar a face do plano voltada para cima é necessário ter em atenção a ordem de criação dos vértices de cada triângulo.

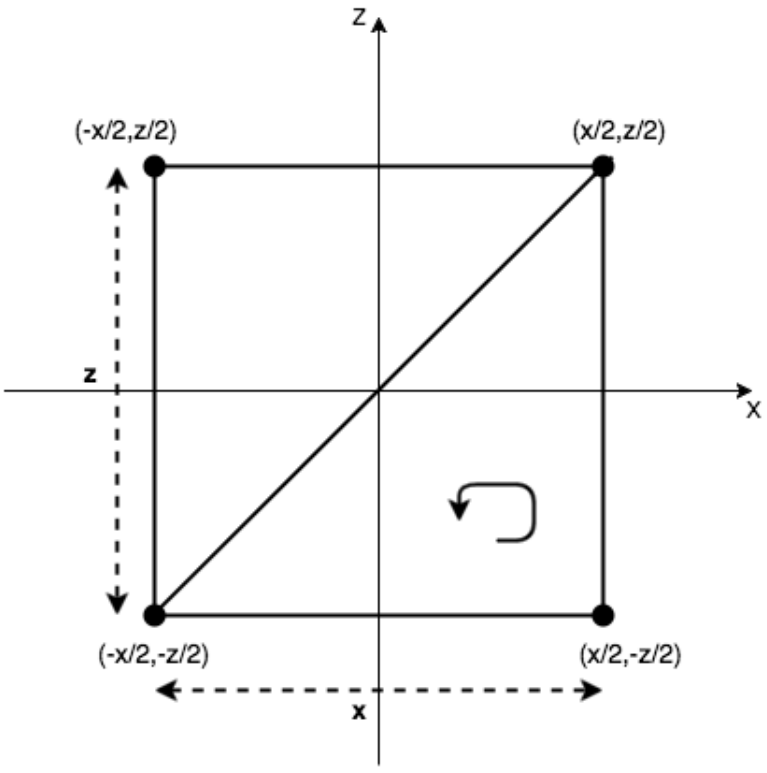


Figura 3.1: Ilustração da construção do plano.

Com a figura acima, é visivel o sentido da ordem de criação dos vértices. De forma a centrar o plano, as coordenadas de cada vértice é obtido com metade do valor de x e z.

3.2 Paralelepípedo

Um paralelepípedo é um prisma com seis faces, e para sua construção são necessários três parâmetros: **largura** (x), **compirmento** (z) e **altura** (y). Para além disso, também é necessário indicar o número de divisões (div).

3.2.1 Algoritmo

Como explicado e ilustrado no plano, para centrar na origem o paralelepípedo usamos metade do valor de **x**, **y** e **z** como coordenadas. Para além disso, a construção de cada face do paralelepípedo segue o mesmo raciocínio do plano, isto é, dois triângulos onde dois vértices são comuns a ambos. No entanto, para implementar as divisões é preciso adotar um algoritmo de construção mais complexo.

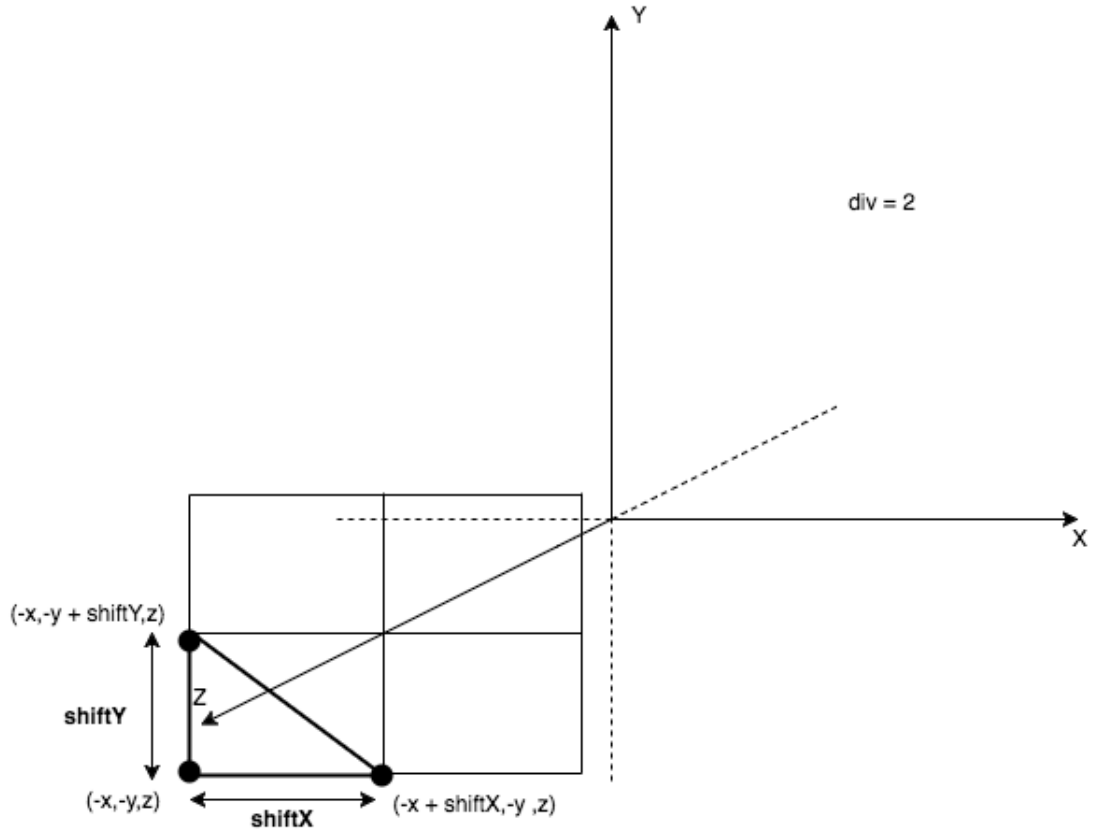


Figura 3.2: Ilustração da construção do paralelepípedo, e do método de divisão.

Como mostra a figura acima, é necessário ter em consideração a divisão que existe, para obtermos os desvios (shifts), de um vértice para outro. Por exemplo, caso a divisão seja de 2, e os valores x = 4, y = 4 e z = 4, os valores dos desvios são: **shiftX** = 4/2 = 2, **shiftY** = 4/2 = 2 e **shiftZ** = 4/2 = 2. A fórmula dos desvios é: shiftP = P/div, sendo P pertencente a X, Y ou Z.
Com a implementação dos desvios torna-se mais fácil a construção do paralelepípedo com divisões. Considerando a face voltada de frente, onde o valor de z, é constante em todos os vértices, é possível obter a mesma face que se obtia com apenas dois triângulos, mas agora com várias divisões. Sendo assim, começamos

por desenhar os triângulos da esquerda para a direita, onde as coordenadas dos vértices, são obtidas através dos desvios calculados. Quando chegar ao fim da linha, incrementamos a altura dos pontos e voltamos a percorrer a linha.
Podemos obter isto numa fórmula (para a face voltada para a frente):

Para todo **i** e **j** menor que **div**:

(-x + (j*shiftX), -y + (i*shiftY), z)
((-x+shiftX) + (j*shiftX), -y + (i*shiftY), z)
(-x + (j*shiftX), (-y+shiftY) + (i*shiftY), z)

(-x + (j*shiftX), (-y+shiftY) + (i*shiftY), z)
((-x+shiftX) + (j*shiftX), -y + (i*shiftY), z)
((-x+shiftX) + (j*shiftX), (-y+shiftY) + (i*shiftY), z)

Onde o **j** percorre em função do eixo dos x, e o **i** com que incremente a altura no final de percorrer a linha.

3.3 Cone

Um cone é um sólido geométrico obtido quando se tem uma pirâmide cuja base é um polígono regular, e o **número de fatias da base tende para infinito**. Os parâmetros para gerar um cone são **r** (raio), **a** (altura), **nfatias** (número de fatias) e **ncamadas** (número de camadas). Quando maior os valores de **nfatias** e **ncamadas**, melhor se torna a curvatura do cone.

3.3.1 Algoritmo

Para a construção do cone, consideramos 3 fases: desenho da base, desenho do plano curvo do cone e por último desenho do topo (bico).

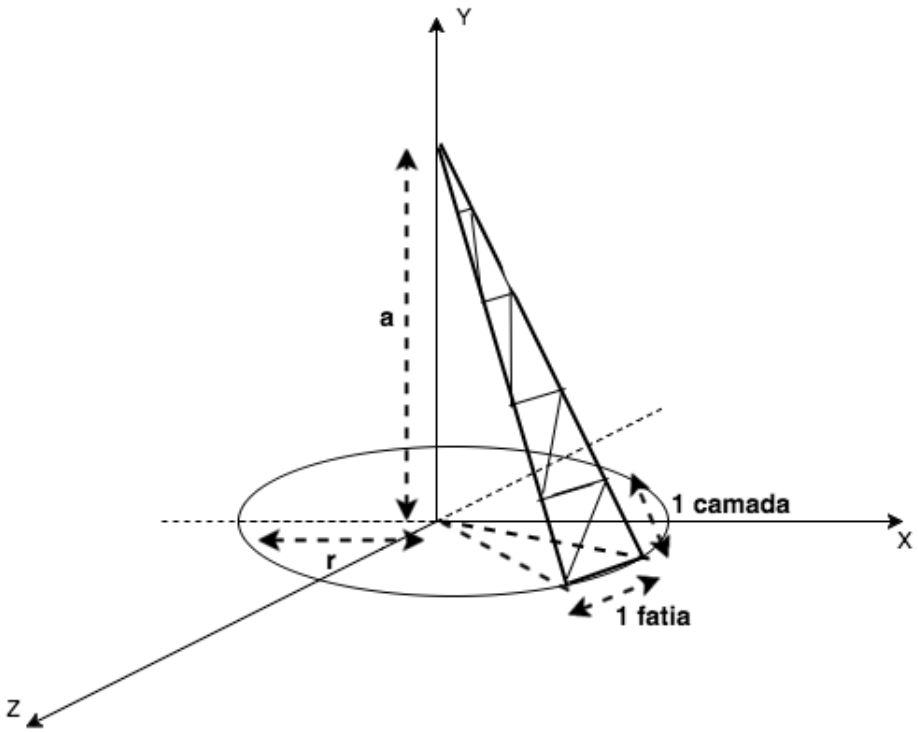


Figura 3.3: Ilustração da construção do cone.

Para desenhar a base, precisamos de fixar um ponto, que neste caso será (0,0,0), para servir de referência para o resto do cone. Este ponto representa o centro do cone, logo, à medida que nos aproximamos da altura do cone, mais próximos estão os pontos do centro.
Para desenhar os lados do polígono regular, que representa a base, é preciso ter em conta o número de fatias (nfatias) dado. Para repartir em fatias equivalentes, dividimos um angulo de 360º pelo número de fatias. Desta forma obtemos a amplitude para uma dada fatia, equivalente a todas as outras. Com esta amplitude, e através das funções cos(x) e sin(x), podemos obter os vértices da base, que seguem a forma de uma circunferência, ou seja, as coordenadas X e Z dos vértices, são obtidas seguindo a fórmula:

x = raio * sin(alfa)
z = raio * cos(alfa)

x = raio * sin(alfa + amplitude)
z = raio * cos(alfa + amplitude)

O ângulo alfa, corresponde ao ângulo do ponto actual, e o ângulo (alfa + amplitude) do próximo ponto, isto é, o valor de alfa começa com 0, e vai acumulando o valor da amplitude de uma fatia, fazendo com que seja possível iterar em forma de uma circunferência.
Para desenhar o plano curvo do cone, seguimos o mesmo raciocínio anterior, no entanto, para conseguirmos garantir que os pontos do vértices tendem para o centro à medida que se desloca no eixo dos y, decrementamos o raio no mesmo sentido,

isto é, à medida que o valor de y aumenta, o valor do raio é descrementado, até que atinge o valor do centro (0). Por fim, quando atingirmos o número de camadas dado, desenhamos o topo do cone. Basicamente, é ligar os vértices ao centro para formar o bico do cone.

3.4 Esfera

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

3.4.1 Algoritmo

Para desenhar uma esfera é necessário considerar algumas variáveis importantes:

> **Deslocamento na horizontal (ângulo)**: corresponde a $2\pi/(nfatias)$;
> **Deslocamento na vertical (ângulo)**: corresponde a $\pi/(ncamadas)$;
> **Altura das camadas**: $raio * sin((\pi/2 - deslocamentoVertical))$.

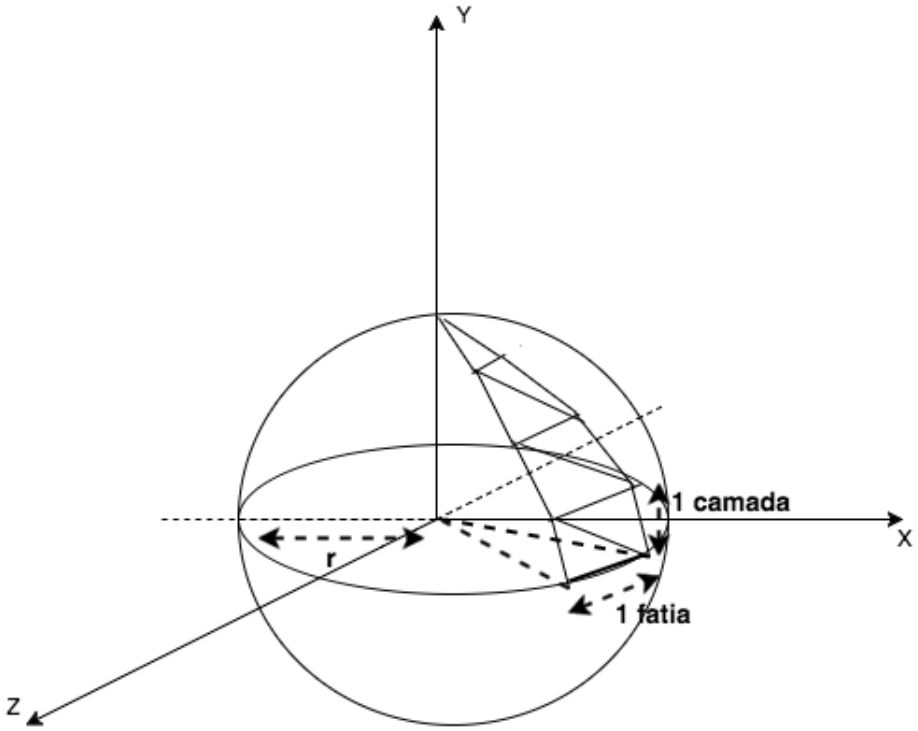


Figura 3.4: Ilustração da construção do esfera.

Com a figura e as variáveis acima, já conseguimos obter um método que nos permite desenhar a esfera. Começamos por desenhar do topo da esfera até ao fundo, uma fatia, isto é, desenhamos fatia a fatia. Com isto, consideramos dois pontos X e Z

actuais, e mais dois pontos X e Z, que correspondem aos proximos pontos. Estes pontos são obtidos em função do raio, funções trigonometricas e o valor do ângulo do deslocamento horizontal. Com estes 3 parâmetros podemos obter os 4 pontos referidos desta forma:

actualX = raio * sin(i*deslocamentoH);
actualZ = raio * cos(i*deslocamentoH);

nextX = raio * sin((i+1)*deslocamentoH);
nextZ = raio * cos((i+1)*deslocamentoH).

Onde i, corresponde às iterações fatia a fatia. Ao fim de obtermos estes pontos basta apenas obter o valor da distância do centro da esfera, este pode ser obtido através da fórmula:

cos(arcsin(altura/raio)).

Por outras palavras, o valor da distância correponde ao valor do cosseno, com amplitude formada pelo triângulo, de hipotenusa igual ao raio e o cateto aposto igual à altura (obtida no início). Desta forma, à medida que andamos no eixo do y, vamos criar uma forma curva (devido aos valores da função cosseno), isto é, do topo da esfera, até a meio, a distância aumenta, no entanto, do meio até ao fundo da esfera, começa a diminuir. Como é claro, é necessário considerar esta distância, para duas alturas diferentes, para conseguirmos obter os vértices, e assim desenhar os triângulos. À medida que se desce no eixo dos y, actualizamos as alturas, desenhando assim a fatia toda até ao fim.

3.5 Cilindro

Um cilindro é um sólido geométrico constituido por duas bases equivalentes, sendo ambas polígonos regulares, onde o **número de fatias das bases tende para infinito**. Os parâmetros para gerar um cilindro são **r** (raio), **a** (altura), **nfatias** (número de fatias) e **ncamadas** (número de camadas). Quanto maiores os valores de **nfatias** e **ncamadas**, melhor se torna a curvatura do cilindro.

3.5.1 Algoritmo

Para a construção do cilindro, consideramos 3 fases, como no exemplo do cone: desenho da base, desenho do plano curvo do cilindro e por último desenho do topo (base).

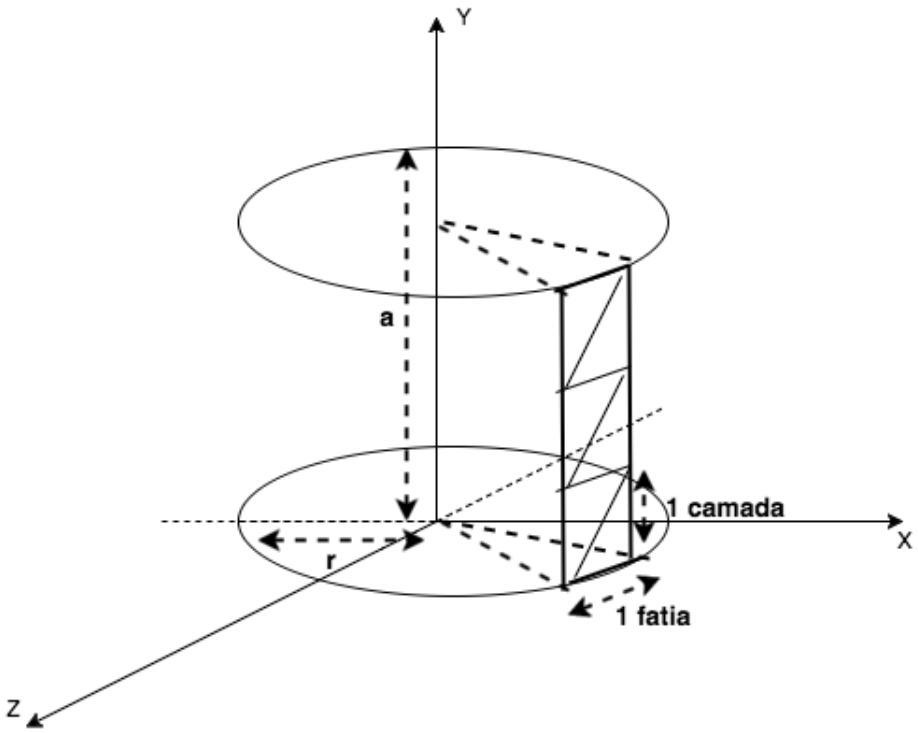


Figura 3.5: Ilustração da construção do cilindro.

O processo de contrução do cilindro é muito idêntico ao do cone, isto é, o método de desenhar a base é igual. O que diferencia é o desenho do plano lateral curvo. Para tal, apenas é necessário manter o raio, ao contrário do cone, à medida que a coordenada y aumenta. Desta forma, conseguimos garantir que à medida que se ande no eixo dos y, o formato da base se mantém. Assim sendo, quando antingirmos o número de camadas, apenas é necessário desenhar novamente a base, uma vez que chegamos ao topo do cilindro.

4. *Generator*

4.1 Descrição

O gerador (ou *generator*), tal como o nome indica, é responsável por gerar ficheiros que contém o conjunto de vértices das primitivas gráficas (plano, caixa, esfera, cone e cilindro) que se pretende gerar, conforme os parâmetros escolhidos (dimensões, e em algumas situações divisões). Estes vértices são conjuntos de 3 pontos que graficamente correspondem a triângulos visto que todas as primitivas têm por unidade de construção o triângulo.

4.2 Usabilidade

De seguida é apresentado o manual de ajuda do *generator*, onde podemos consultar os diferentes comandos e os respectivos argumentos. Este pode ser conseguido através do comando *./generator -help*.

```
#_____________________ HELP _____________________#
|                                                 |
|   Usage: ./generator {COMMAND} ... {FILE}       |
|                      [-h]                       |
|                                                 |
|   COMMANDS:                                     |
| - plane [SIZE]                                  |
|      Creates a square in the XZ plane, centred in the origin.  |
|                                                 |
| - box [SIZE X] [SIZE Y] [SIZE Z] [DIVISIONS]    |
|      Creates a box with the dimensions and divisions specified.|
|                                                 |
| - sphere [RADIUS] [SLICE] [STACK]               |
|      Creates a sphere with the radius, number of slices and    |
|      stacks given.                              |
|                                                 |
| - cone [RADIUS] [HEIGHT] [SLICE] [STACK]        |
|      Creates a cone with the radius, height, number of slices  |
|      and stacks given.                          |
|                                                 |
| - cylinder [RADIUS] [HEIGHT] [SLICE] [STACK]    |
|      Creates a cylinder with the radius, height, number of     |
|      slices and stacks given                    |
|                                                 |
|   FILE:                                         |
| In the file section you can specify any file in which you wish |
| to save the coordinates generated with the previous commands.  |
|                                                 |
#_________________________________________________#
```

Figura 4.1: Manual de ajuda do Generator.

4.3 Demonstração

O funcionamento do gerador, segundo o manual de ajuda anteriomente apresentado, é muito simples. Deve primeiramente ser selecionada qual a figura e respectiva(as) dimensão(ões) a gerar, conforme os diferentes inputs aceites pelo programa, em conjunto com o nome do respectivo ficheiro resultante.

- **Plano**

  plane <dimensão> <ficheiro_resultante>

- **Paralelipípedo**

  box <dimensão_x> <dimensão_y> <dimensão_z> <ficheiro_resultante>

- **Esfera**

  sphere <raio> <nr_fatias> <nr_camadas> <ficheiro_resultante>

- **Cone**

  cone <raio> <algura> <nr_fatias> <nr_camadas> <ficheiro_resultante>

- **Cilindro**

  cylinder <raio> <algura> <nr_fatias> <nr_camadas> <ficheiro_resultante>

O input deverá, portanto, ser semelhante ao exemplo seguinte.

```
$ ./generator plane 4 plane.3d
```

Figura 4.2: Exemplo de input para do Generator.

Posto isto, o gerador irá criar, em caso de inexistência, uma diretoria */files*, onde serão criados todos os ficheiros output do programa. Os ficheiros output podem ter qualquer tipo de extensão dada pelo utilizador, não existe necessidade de ser igual à apresentada no exemplo anterior. Apesar disto, os ficheiros resultantes apresentam todos a mesma estrutura:

$$coordenada_x\ coordenada_y\ coordenada_z \backslash n$$

Em cada linha do ficheiro estão contidos três números em vírgula fluante separados por um espaço, representado um ponto único pertencente a um vértice. O ficheiro resultante do exemplo considerado, seria, portanto, algo idêntico à figura seguinte.

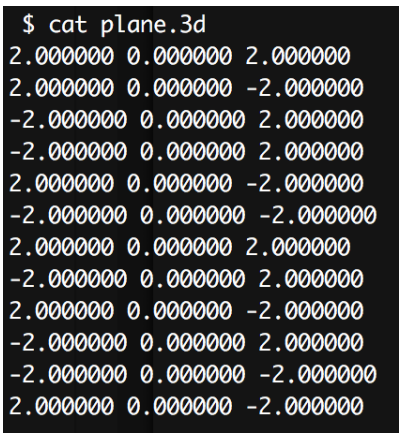


Figura 4.3: Exemplo de output do Generator.

5. *Engine*

5.1 Descrição

O motor (ou *engine*) é responsável por receber ficheiros de configuração escritos em XML. Dentro destes ficheiros contém apenas a indicação de quais os ficheiros que, previamente gerados a partir do generator, serão carregados. Desta forma, depois do motor fazer parsing dos ficheiros gerados, irá interpretar e apresentar graficamente os modelos no seu conteúdo.

5.2 Usabilidade

De seguida é apresentado o manual de ajuda do *engine*, onde podemos consultar os diferentes inputs válidos, assim como os comandos de interação com o modelo. Este pode ser conseguido através do comando *./engine -help*.

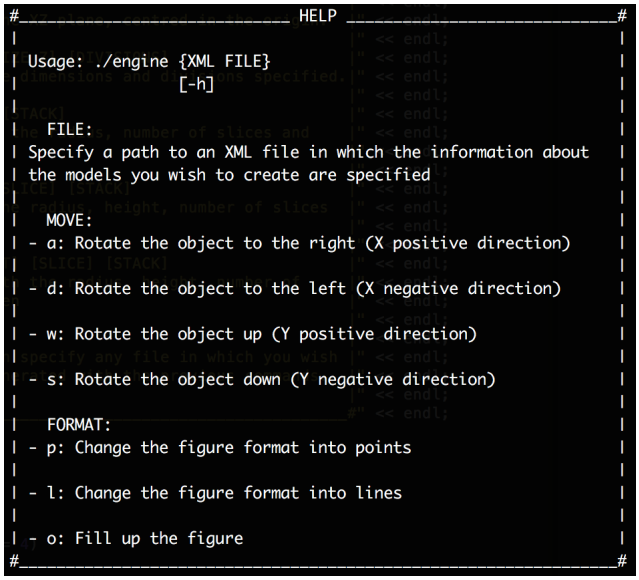


Figura 5.1: Manual de ajuda do Engine.

5.3 Demonstração

O motor, por sua vez, é responsável por ler um ficheiro de configuração, apresentado em XML e apresentar os modelos inseridos neste. É importante ter em conta

que este ficheiro de configuração é criado manualmente pelo utilizador e os ficheiros modelo presentes neste devem ser previamente gerados pelo *generator*. Após interpretados e apresentados os modelos é possível interagir com estes segundo os comandos anteriormente referidos no menu de ajuda.
Segue um exemplo do funcionamento do *engine*, considerando o seguinte ficheiro como input.

```
 $ cat teste.xml
<scene>
        <model file='./CG/files/plane.3d' />
        <model file='./CG/files/sphere.3d' />
</scene>
```

Figura 5.2: Exemplo de um ficheiro de configuração (XML).

Executar o *engine*, neste caso nomeado de *CG_Trabalho*, fornecendo-lhe como ficheiro input o ficheiro anterior.

```
$ ./CG_Trabalho ./CG/files/teste.xml
```

Figura 5.3: Exemplo de input do Engine.

O output deverá ser a apresentação GLUT dos modelos pretendidos, que no caso exemplicado seria algo do género:

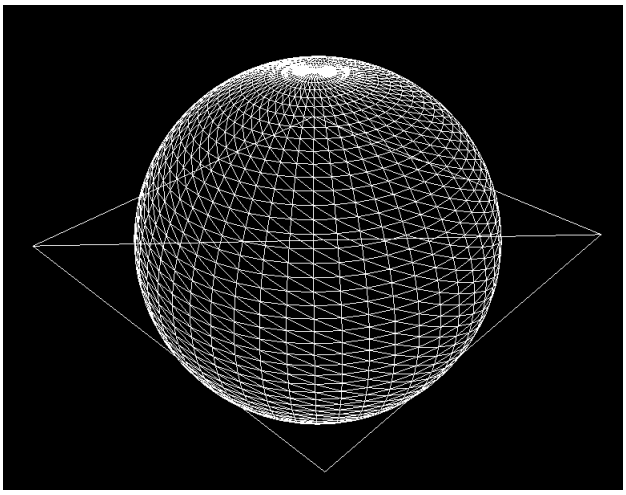


Figura 5.4: Exemplo de output do Engine.

Posteriormente para além de ser possível utilizar os comandos W, A, S e D para rodar os modelos apresentados, é também possível visualizar os modelos de diferentes perspectivas.

Linhas (Comando L)

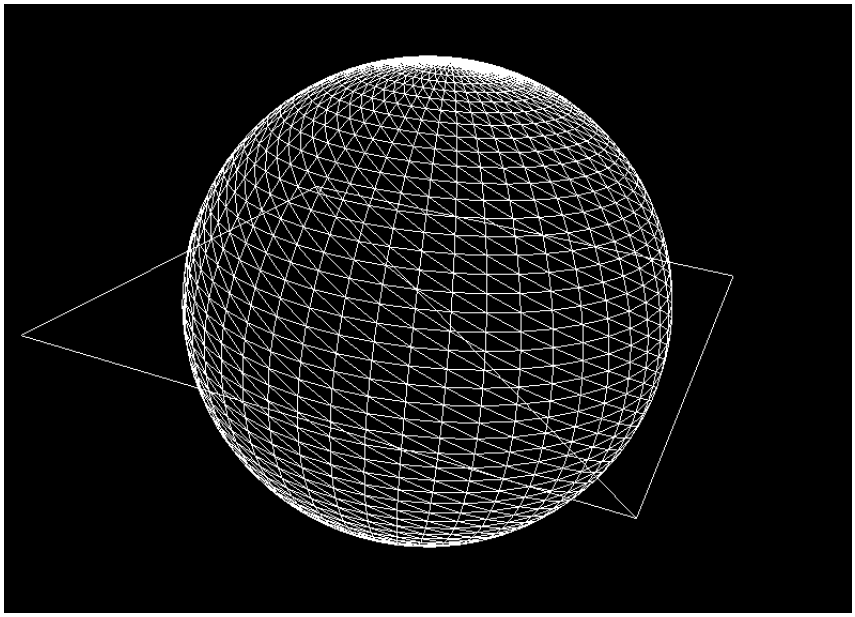


Figura 5.5: Modelos apresentados por linhas.

Pontos (Comando P)

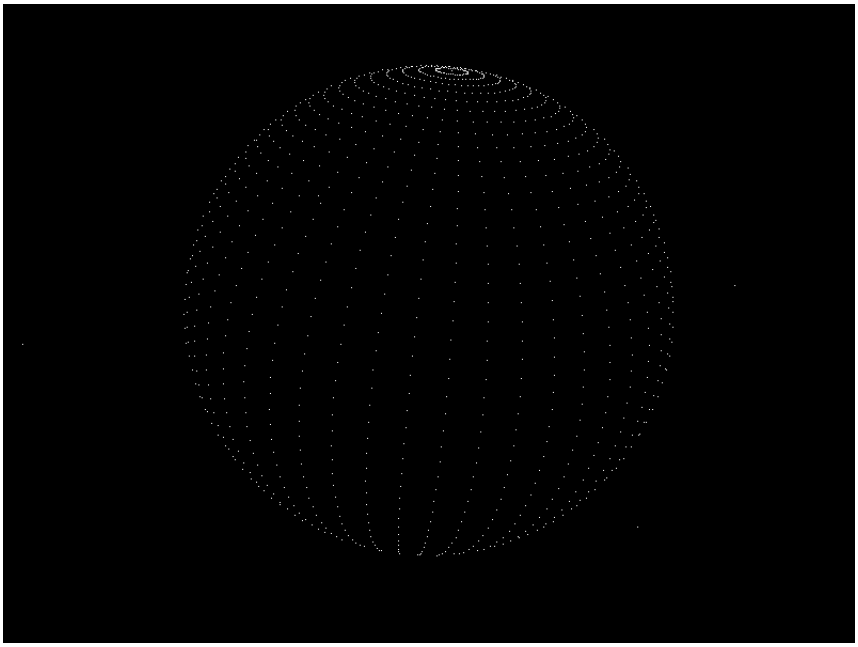


Figura 5.6: Modelos apresentados por pontos.

Preenchido (Comando O)

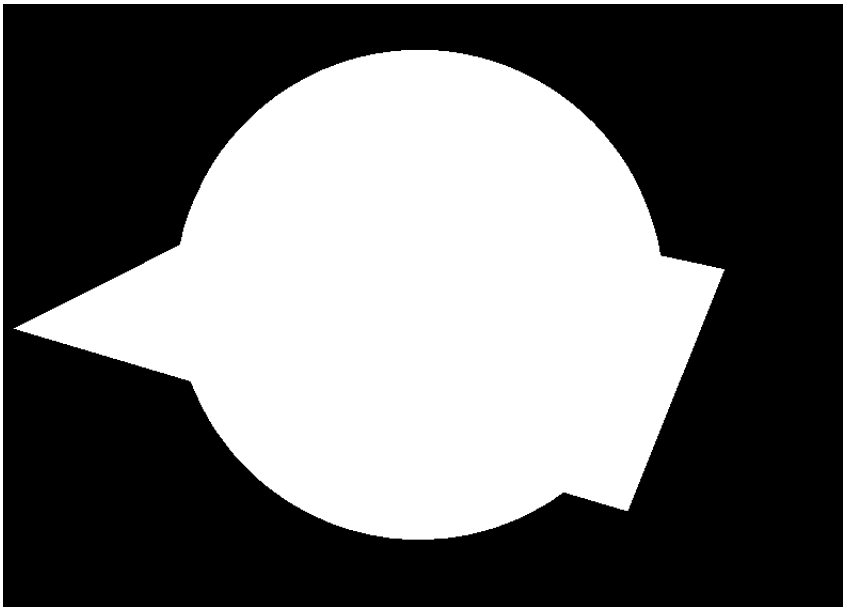


Figura 5.7: Modelos apresentados preenchidamente.

6. *Modelos 3D*

6.1 Plano

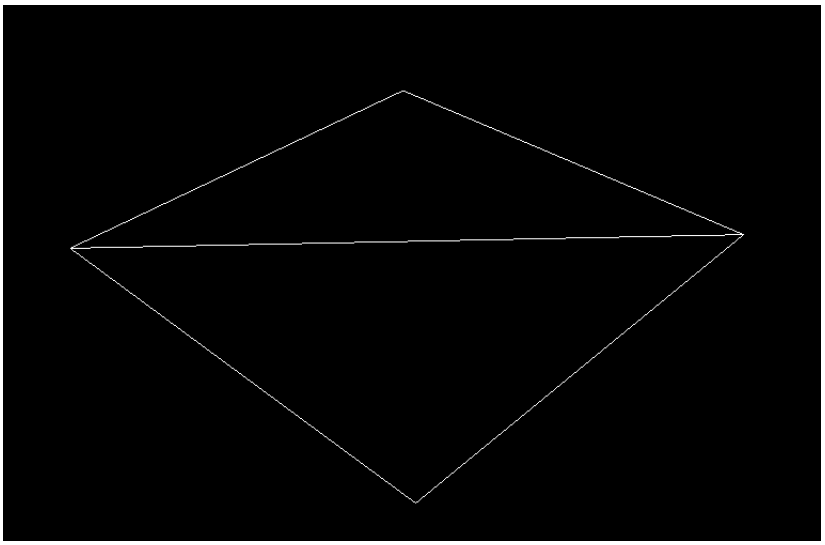


Figura 6.1: Plano com dimensão 4.

6.2 Box

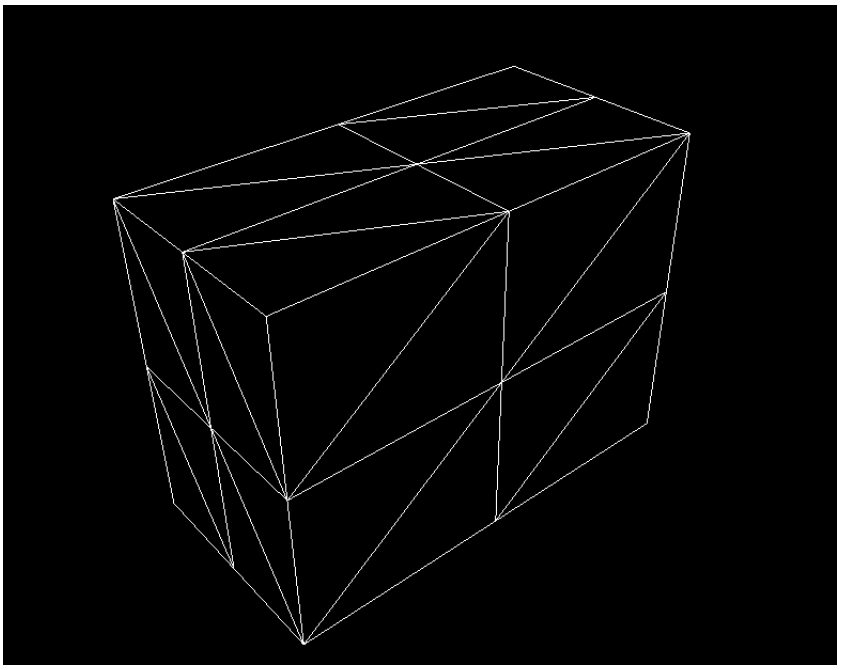


Figura 6.2: Paralelipípedo com dimensões (x,y,z) (2, 3, 4) e 2 divisões.

6.3 Esfera

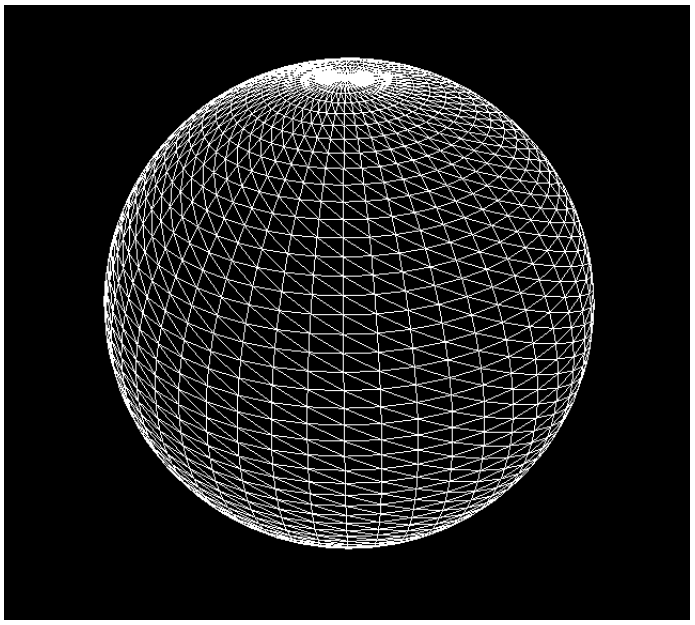


Figura 6.3: Esfera com 2 de raio, 50 fatias e 50 camadas.

6.4 Cone

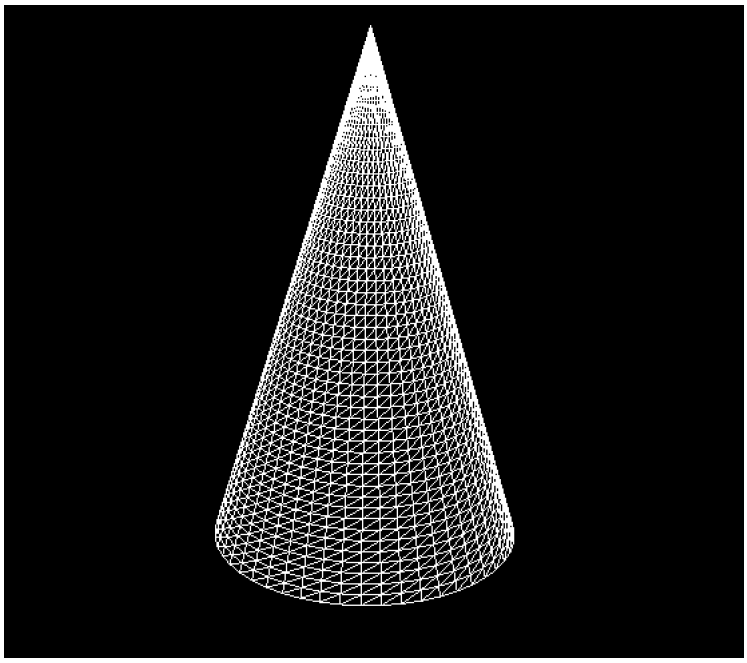


Figura 6.4: Cone com raio 1, altura 3, 50 fatias e 50 camadas.

6.5 Cilindro

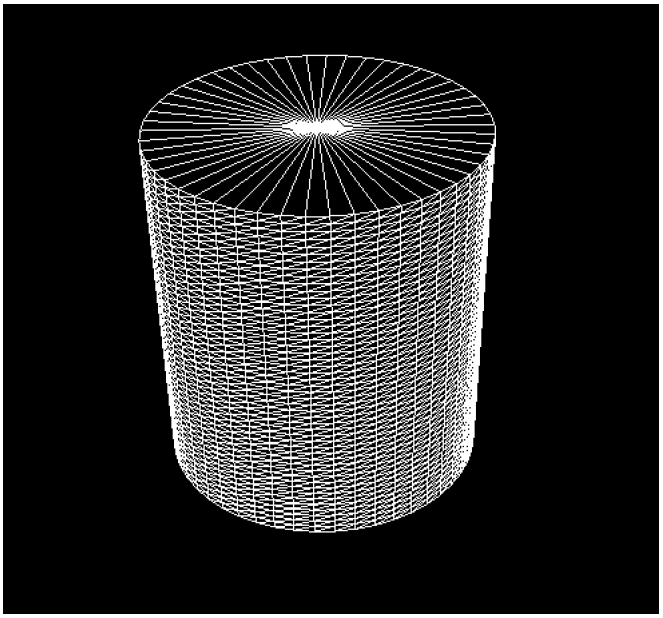


Figura 6.5: Cilindro com raio 1, altura 2, 50 fatias e 50 camadas.

7. Conclusão/Trabalho futuro

A elaboração desta primeira fase do trabalho foi bastante importante, na medida em que nos permitiu ganhar alguma experiência tanto no que toca à utilização e consequente especialização em ferramentas associadas à computação gráfica, como o OpenGL e o GLUT. Ao mesmo tempo permitiu-nos adquirir conhecimentos no que toca à linguagem de programação C++, a qual veio a ser bastante útil e insdispensável para a realização deste trabalho. Deste modo, esperamos que o resultado obtido nesta primeira fase, de algum modo nos motive e sirva de rampa de lançamento para a realização das consequentes fases do projeto.