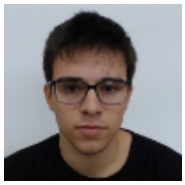
RELATÓRIO PRÁTICO FASE 1

GRUPO 30

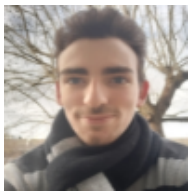
Ana Almeida, A83916

André Figueiredo, A84807



Luís Ferreira, A86265

Rafael Lourenço, A86266



Mestrado Integrado em Engenharia Informática 2019/2020

Computação Gráfica

Universidade do Minho

6 de Março de 2020

Conteúdo

Capítulo 1

Introdução

1.1 Contextualização

Foi-nos proposto, no âmbito da UC Computação Gráfica, a criação de um motor gráfico genérico para representar objetos a 3 dimensões, com o auxílio de duas ferramentas - *OpenGL* e *C++*.
O projeto está dividido em várias fases distintas, sendo que nesta primeira serão implementadas algumas primitivas gráficas, como o plano, o paralelepípedo, a esfera e o cone.

1.2 Resumo

Nesta primeira parte, foi necessária a criação de duas aplicações essenciais para o seu funcionamento:

- **Generator**, gera a informação/os pontos para a criação do modelo e guarda num ficheiro especificado, acrescentando ao ficheiro XML o nome do novo ficheiro;
- **Engine**, motor capaz de ler um ficheiro XML e exibir os modelos lá indicados.

O objetivo desta fase passa por gerar e exibir as primitivas gráficas em *GLUT*.

Capítulo 2

Arquitetura do código

Depois de analisármos o problema proposto e de pensar em várias hipóteses, decidimos implementar duas aplicações em C++, **generator** e **engine**.

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 estruturas das diferentes formas geométricas a desenvolver de forma a gerar os respetivos vértices.

2.1.2 Motor

engine.cpp - Aplicação que possui a capacidade de apresentar a janela, exibindo os modelos indicados no ficheiro *file.xml* e interagir com o ambiente.

2.2 Formas

2.2.1 Plano

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

2.2.2 Paralelepípedo

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

2.2.3 Esfera

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

2.2.4 Cone

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

2.3 Extras

2.3.1 TinyXML

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

Capítulo 3

Primitivas geométricas

3.1 Plano

Um plano de duas faces é composto por 4 vértices e 4 triângulos, 2 voltados para cima e 2 voltados para baixo, isto é, o plano é visto de cima e de baixo.
Decidimos que seria um plano quadrado, assim sendo, as suas dimensões são dadas pelo valor inserido pelo utilizador. O plano gerado 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.

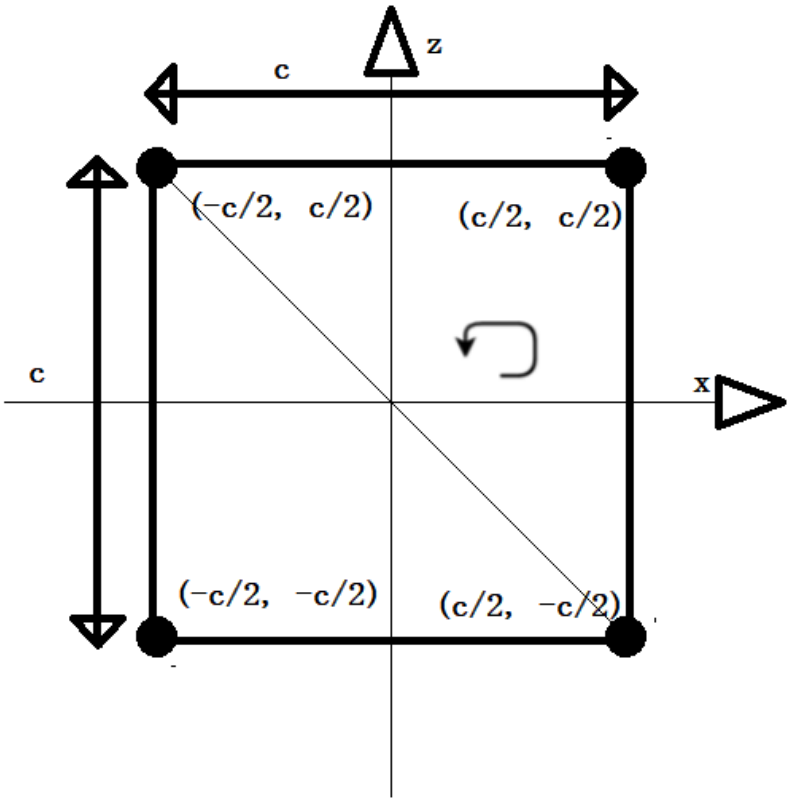


Figura 3.1: Ilustração da construção do plano

Com o auxílio da figura acima facilmente se percebe que o sentido da ordem de criação dos vértices tem de ser no sentido contrário dos ponteiros do relógio para

poder ser vísivel de cima.
Pelo mesmo processo, para ser possível ser visto de baixo, basta inverter a ordem de criação dos vértices, neste caso, no sentido dos ponteiros do relógio.
De forma a centrar o plano, as coordenadas de cada vértice obtido são metade do valor de inserido, neste caso, c.

3.2 Paralelepípedo

Um paralelepípedo é um prisma com seis faces e, para o gerar, são necessários três parâmetros: largura, x, comprimento, z e altura, y. Opcionalmente, também poderá ser indicado o número de divisões em cada eixo - $divX$, $divY$ e $divZ$.

3.2.1 Algoritmo

Como explicado e ilustrado no plano, para centrar na origem, calculámos todos os valores como se fossem $[0; x]$, por exemplo, e subtraíamos metade do valor de x $(x/2)$. Neste caso, o mesmo é feito para as coordenadas yy, onde seria subtraído $y/2$. A geração 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 um pouco mais complexo.

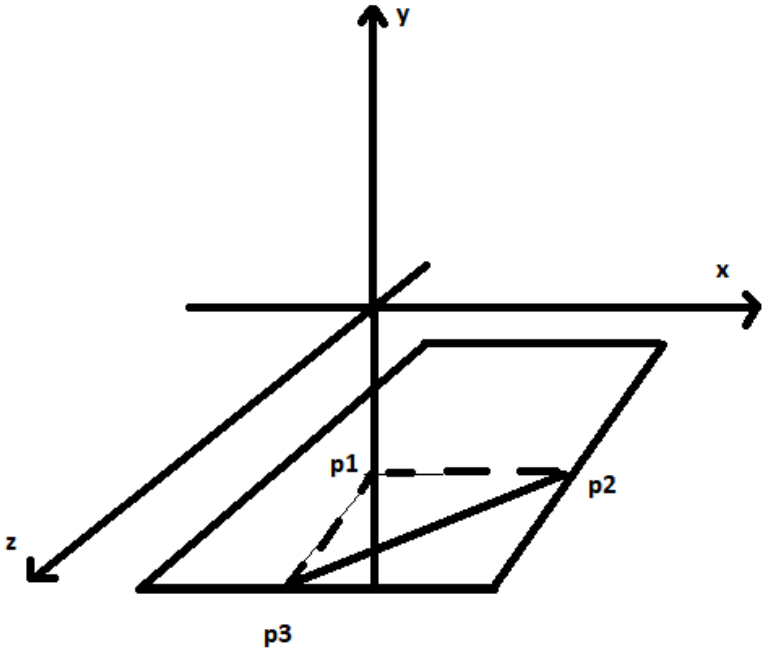


Figura 3.2: Ilustração da construção do paralelepípedo e método de divisão

Como mostra a figura acima, é necessário ter em consideração a divisão que existe para obtermos os desvios de um vértice para outro. Por exemplo, caso a divisão seja 2 para todos os eixos e os valores $x = 4$, $y = 4$ e $z = 4$, isto implica que os valores dos desvios são:

$$x/divX = 2,\ y/divY = 2,\ z/divZ = 2.$$

Com a implementação dos desvios torna-se mais fácil a construção do paralelepípedo com divisões. Considerando as faces paralelas ao plano XOZ, onde o valor de y é constante em todos os vértices duma face, é possível obter a mesma face que se obtinha com apenas dois triângulos, todavia, agora com várias divisões.
Portanto cada quadrado, conjunto dos 2 triângulos desenhados de cada vez, é descrito por:

$$x_i = i/divX * x - (x/2)$$

$$i/divX * x - (x/2) = i * x/divX - (x/2) = i * shift - (x/2)$$

sendo o resultado acima, x_i, então o passo seguinte é:

$$(i+1)/divX * x - (x/2) = (x+1) * x/divX - (x/2) = x * shift - (x/2) + shift = x_i + shift$$

Tal como seria de esperar, o x do ponto p_{i+1} é $x_{i+1} = x_i + shift$.

Sendo assim, começámos por desenhar os triângulos no sentido positivo do eixo dos xx, onde as coordenadas dos vértices são obtidas através dos desvios calculados. Por cada conjunto de 2 triângulos é também escrito o triângulo da face oposta, onde é alterado valor dos yy e escrito na ordem contrária, para poder ser visto na direção oposta. Quando chegar ao fim da linha, passa para a linha seguinte e repete todo o processo.
Podemos colocar isto numa fórmula, para isso, mostrámos o exemplo para as faces voltadas para baixo:

```
for (int k = 0; k < divZ; ++k)
{
        for (int i = 0; i < divX; ++i)
        {
                x1 = i / divX * x -x/2;
                y1 = -y/2;
                z1 = k / divZ * z -z/2;

                x2 = (i + 1) / divX * x -x/2;
                y2 = -y/2;
                z2 = k / divZ * z -z/2;

                x3 = i / divX * x -x/2;
                y3 = -y/2;
                z3 = (k + 1) / divZ * z -z/2;

                x4 = x2;
                y4 = y2;
                z4 = z2;

                x5 = (i + 1) / divX * x -x/2;
                y5 = -y/2;
                z5 = (k + 1) / divZ * z -z/2;

                x6 = x3;
```

```
                y6 = y3;
                z6 = z3;
        }
}
```

Tendo já os pontos da face inferior, podemos tirar os pontos da face superior:

```
y1 = y2 = y3 = y4 = y5 = y6 = y/2;
```

Agora apenas teríamos de inverter a ordem pela qual os pontos eram escritos no ficheiro, pois queremos que a face seja vísivel do lado oposto da face inferior, por exemplo, o ponto 2 seria agora o ponto 3 e vice-versa.
Analogamente, o processo para as faces paralelas ao plano YOZ e XOZ é muito similar, apenas são trocadas váriaveis.

3.3 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 raio, r, número de fatias, *slice* e número de camadas, *stack*.

3.3.1 Algoritmo

Para desenhar uma esfera é necessário considerar algumas variáveis importantes:

- **Deslocamento do ângulo da horizontal,** θ: corresponde a $2\pi/slice$;
- **Deslocamento do ângulo da vertical,** φ: corresponde a $\pi/stack$;

Usámos coordenadas esféricas para descrever a esfera e as suas equações para cada ponto, com um dado r, θ e φ, são:

$$x = r \times \cos(\theta) \times \sin(\varphi)$$

$$y = r \times \cos(\varphi)$$

$$z = r \times \sin(\theta) \times \sin(\varphi)$$

Para que as váriaveis φ e o θ estejam atualizadas, no fim de uma iteração sobre o θ é sempre feito:

$$\theta = \theta + passoTHETA$$

quando é uma iteração sobre o φ é feito :

$$\varphi = \varphi + passoPHI$$

sendo o passoTHETA e o passoPHI, o deslocamento do ângulo da horizontal e da vertical, referidos acima, respetivamente.

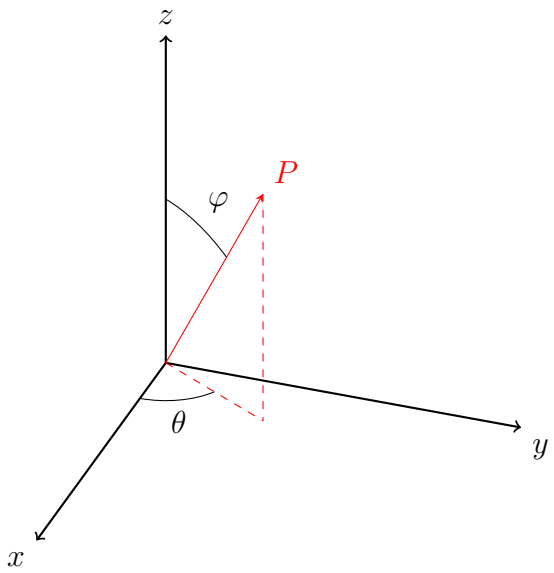


Figura 3.3: Coordenadas esféricas

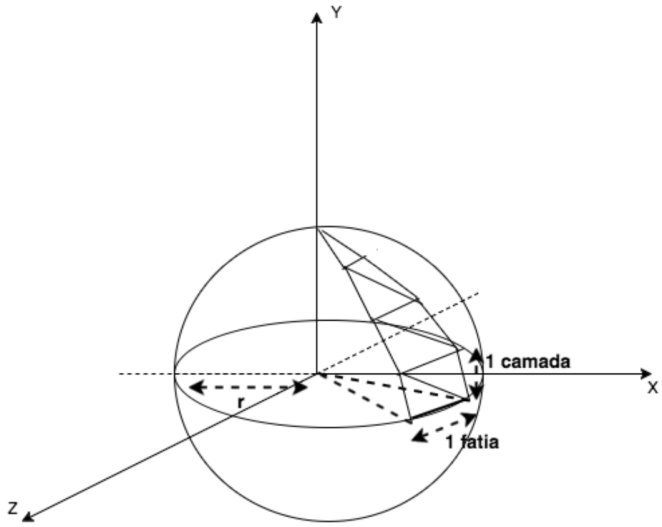


Figura 3.4: Ilustração da construção da esfera

```
phi = 0;
for (int i = 0; i < stack; ++i, phi += passoPHI)
{
        for (int j = 0, theta = 0; j < slice; ++j, theta += passoTHETA)
        {
                x1 = raio * cos(theta) * sin(phi);
                y1 = raio * cos(phi);
                z1 = raio * sin(theta) * sin(phi);

                x2 = raio * cos(theta + passoTHETA) * sin(phi + passoPHI);
                y2 = raio * cos(phi + passoPHI);
                z2 = raio * sin(theta + passoTHETA) * sin(phi + passoPHI);

                x3 = raio * cos(theta) * sin(phi + passoPHI);
                y3 = raio * cos(phi + passoPHI);
                z3 = raio * sin(theta) * sin(phi + passoPHI);

                x4 = x1;
                y4 = y1;
                z4 = z1;

                x5 = raio * cos(theta + passoTHETA) * sin(phi);
                y5 = raio * cos(phi);
                z5 = raio * sin(theta + passoTHETA) * sin(phi);

                x6 = x2;
                y6 = y2;
                z6 = z2;
                }
        }
}
```

Este algoritmo precisou de um pequeno ajuste para não criar triângulos inúteis, isto é, quando está a fazer a última iteração, $i = stack - 1$, ele iria criar um triângulo em que 2 vértices iriam coincidir, criando uma linha ao invés de um triângulo, por isso, nesse caso, não são colocados esses valores para o ficheiro.
E, na primeira iteração, tal como pode ser visto na figura 3.4, o mesmo problema acontece e apenas o triângulo de baixo é preciso, por isso, para a última iteração, quando $i = 0$, apenas um triângulo é colocado no ficheiro.

3.4 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 raio, r, altura, h, número de fatias, *slice* e número de camadas, *stack*. Quanto maiores os valores de *slice* e *stack* mais se aproxima de um cone.

3.4.1 Algoritmo

Para a construção do cone, considerámos duas fases: o desenho da base e o desenho do plano curvo do cone.

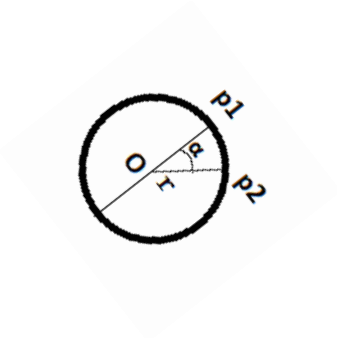


Figura 3.5: Base do Cone

Para desenhar a base, precisámos de fixar um ponto para servir de referência para o resto do cone que, neste caso, será $(0, 0, 0)$. Este ponto representa o centro da base do cone. Portanto, neste exemplo, o triângulo é descrito na ordem O, $p1$ e $p2$, sendo, respetivamente,

$$(0,\ 0,\ 0)$$
$$(r\ \times cos(\theta),\ 0,\ r\ \times sin(\theta))$$
$$(r\ \times cos(\alpha + \theta),\ 0,\ r\ \times sin(\alpha + \theta))$$

O número de triângulos na base estará relacionado com o número de fatias e a amplitude de cada passo será o **deslocamento do ângulo da horizontal,** θ, que corresponde a $2\pi/slice$.

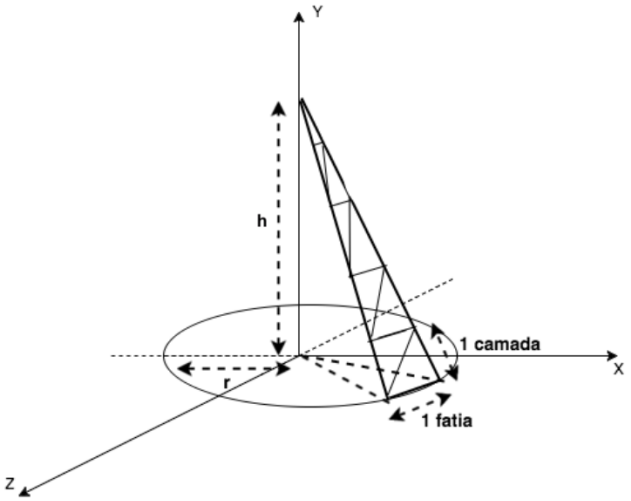


Figura 3.6: Ilustração da construção do cone

O ângulo θ corresponde ao ângulo do ponto atual e o ângulo $(\theta + \alpha)$ ao do próximo ponto, isto é, o valor de θ 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 a outra parte do desenho do modelo, temos de desenhar o plano curvo, ou seja, teremos de iterar por intervalos de altura. Visto que se trata de um cone, os pontos p_2 e p_1 da figura 3.8 estão no mesmo plano yy e os pontos pertencentes ao cone nesse plano formam uma circunferência. Tal como acima, o **deslocamento do ângulo da horizontal**, θ, é $2\pi/slice$ contudo, ao passar para uma stack acima, o raio muda, por isso, temos de descobrir o novo raio para cada stack. O **intervalo entre stacks** é $H/stack$.

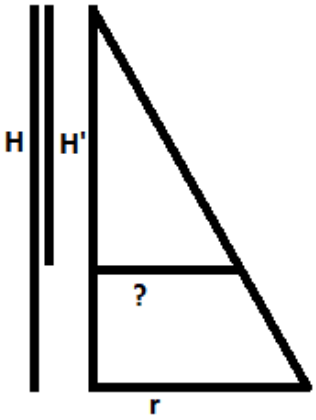


Figura 3.7: Corte do cone

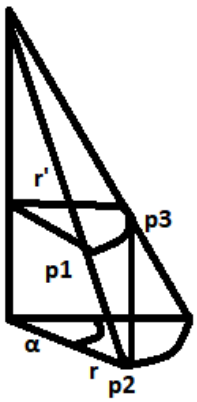


Figura 3.8: Fatia do Cone

Olhando para a figura 3.7 e usando o Teorema fundamental da semelhança de triângulos, podemos descobrir o raio de qualquer stack, sabendo a altura a que se encontra.
Neste caso, o ? seria dado por $H' \times r/H$, visto que o H é a distância do vértice mais "alto"à stack. Sendo y_1 igual a 0 e com um raio igual a r, então a coordenada em y é 0. Isto faz com que a coordenada em y da stack acima seja $(H - H')$ e, portanto

$(H - H') \times r/H$ em que a coordenada y é $0 + H/stack$. Então, tomando a figura 3.8, os pontos **p1**, **p2** e **p3** são, que têm de ser desenhados da seguinte forma para serem vísiveis para o exterior, respetivamente:

$$(r' \times cos(\theta),\ y + H/stack,\ r' \times sin(\theta))$$
$$(r \times cos(\theta),\ y,\ r \times sin(\theta))$$
$$(r' \times cos(\alpha + \theta),\ y + H/stack,\ r' \times sin(\alpha + \theta))$$

```
for (int i = 0; i < stack; ++i)
{
        double d1, d2;
        d1 = (double) raio * (alt - h) / alt;
        d2 = (double) raio * (alt - proxH) / alt;

        angulo = 90;
        proxAng = angulo + passoAngulo;
        for (int alfa = 0; alfa < slice; ++alfa)
        {

                x1 = d2 * cos((double)angulo * PI / 180);
                y1 = proxH;
                z1 = d2 * sin((double)angulo * PI / 180);

                x2 = d1 * cos((double)angulo * PI / 180);
                y2 = h;
                z2 = d1 * sin((double)angulo * PI / 180);

                x3 = d2 * cos((double)proxAng * PI / 180);
                y3 = proxH;
                z3 = d2 * sin((double)proxAng * PI / 180);

                x4 = x2;
                y4 = y2;
                z4 = z2;

                x5 = d1 * cos((double)proxAng * PI / 180);
                y5 = h;
                z5 = d1 * sin((double)proxAng * PI / 180);

                x6 = x3;
                y6 = y3;
                z6 = z3;

                angulo = proxAng;
                proxAng += passoAngulo;
        }
        h = proxH;
        proxH += passo;
}
```

Capítulo 4

Generator

4.1 Descrição

O gerador (*generator*), tal como o nome indica, é responsável por gerar ficheiros que contém o conjunto de vértices das primitivas gráficas (plano, paralelepípedo, esfera e cone) 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 respetivos argumentos. Este pode ser conseguido através do comando $./generator - h$ ou $./generator - help$.

```
./generator -h
#_____________________________ HELP _____________________________#
|                                                                 |
|   Usage: ./generator {COMMAND} ... {OUTPUT 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.                                          |
|                                                                 |
|   OUTPUT 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 Instruções

4.3 Demonstração

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

$ plane <dimensão> <ficheiro resultante>

$ box <dimX> <dimY> <dimZ> [divX] [divY] [divZ] <ficheiro resultante>

$ sphere <raio> <número de fatias> <número de camadas> <ficheiro resultante>

$ cone <raio> <altura> <fatias> <camadas> <ficheiro resultante>

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

```
Fri Mar 06 21:12:21 adduser :Uni$ ./generator box 3 3 3 Engine/Figures/box.3d
```

Figura 4.2: Input para Generator

Existe uma pasta chamada */Figures*, 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:

A primeira linha tem o número de triângulos, enquanto que as restantes são do tipo:

$$c_{x1}\ c_{y1}\ c_{z1}\ c_{x2}\ c_{y2}\ c_{z2}\ c_{x3}\ c_{y3}\ c_{z3}$$

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.

```
Fri Mar 06 21:08:18 adduser :figures$ cat plane.3d
4
5 0 5 5 0 -5 -5 0 5
5 0 -5 -5 0 -5 -5 0 5
5 0 -5 5 0 5 -5 0 5
-5 0 5 -5 0 -5 5 0 -5
```

Figura 4.3: Formato ficheiro do output

Capítulo 5

Engine

5.1 Descrição

Nesta parte do projeto é feito o parse de um Documento XML, no qual está representada uma cena. Nesta primeira fase, o objetivo é produzir os vários modelos que pertencem à cena. Para isso, com o auxílio da biblioteca *tinyxml*, conseguimos obter o nome dos ficheiros correspondentes a cada modelo.
De seguida, criámos uma estrutura auxiliar chamada *"Triangulo"* onde guardámos as coordenadas (x, y, z) dos três pontos que o constituem. Uma vez que uma figura é um conjunto de triângulos, usámos um array dinâmico para a representar. Para guardar as figuras implementámos uma lista ligada.
No fim deste processo, serão desenhadas todas as figuras armazenadas em memória, de forma a obter a cena pretendida.
Desta forma, depois do motor fazer parsing dos ficheiros gerados, irá interpretar e apresentar graficamente os modelos no seu conteúdo.

5.2 Usabilidade

Na página seguinte é apresentado o manual de instruções para manusear a cena produzida. Este menu é apresentado através do comando $./engine - h$ ou $./engine - help$.

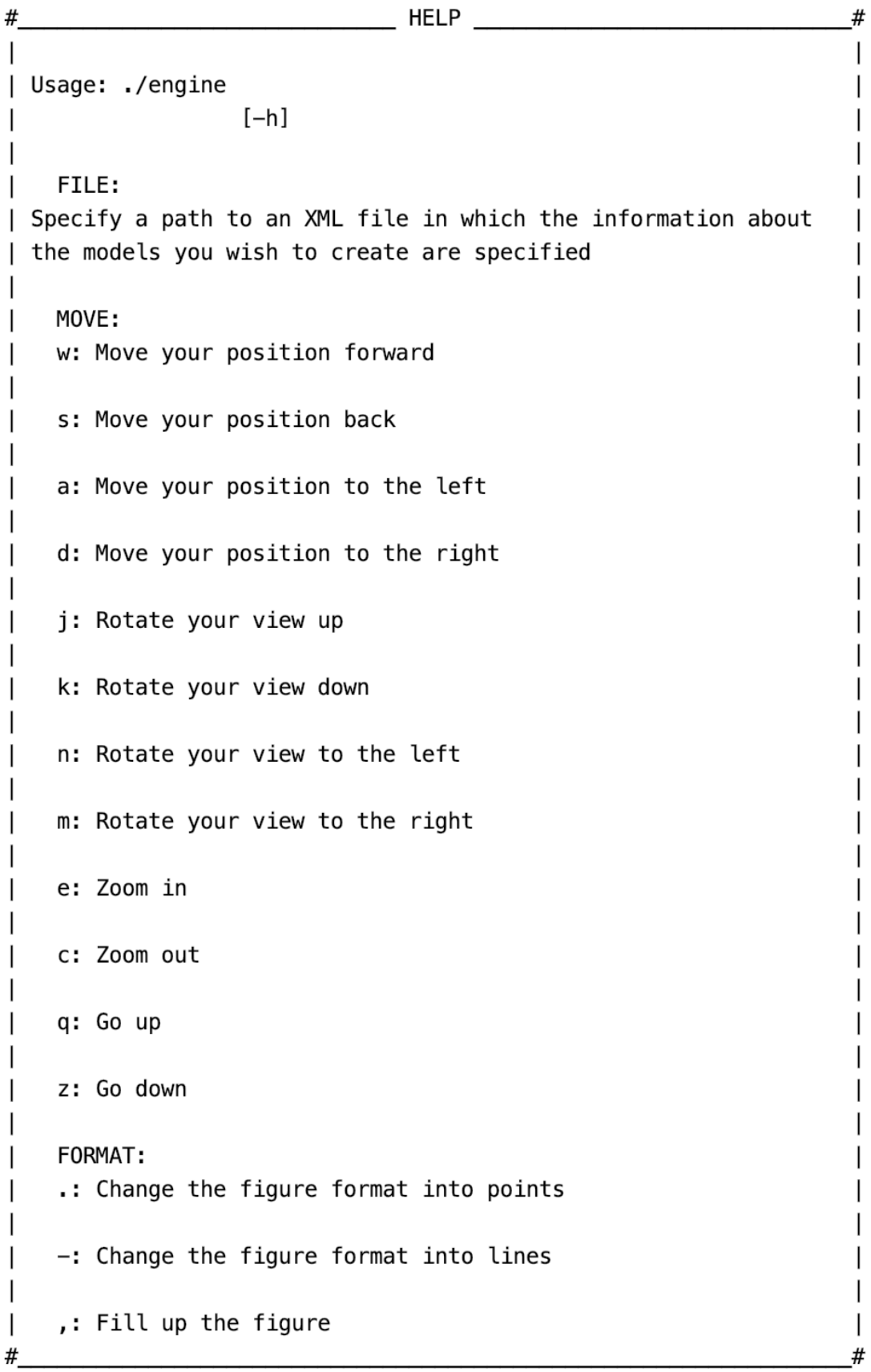


Figura 5.1: Manual de Instruções

5.3 Demonstração

Nesta secção iremos mostrar alguns exemplos funcionamento do engine, considerando o respetivo ficheiro XML.
O motor 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. O output deverá ser a apresentação GLUT dos modelos pretendidos, que no caso exemplicado seria algo do seguinte tipo:

```
<scene>
    <model file="box.3d" />
    <model file="plane.3d" />
    <model file="cone.3d" />
    <model file="sphere.3d" />
</scene>
```

Figura 5.2: Ficheiro XML

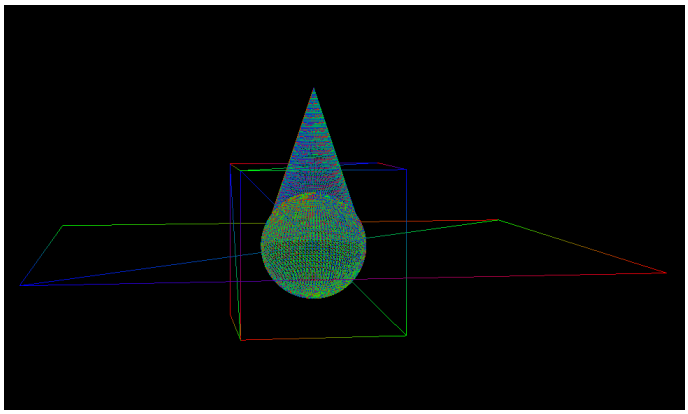


Figura 5.3: Exemplo de output das Primitivas Geométricas

Como se pode verificar no manual de instruções mostrámos, de seguida, que é possível ver o mesmo modelo por pontos, linhas ou preenchido.

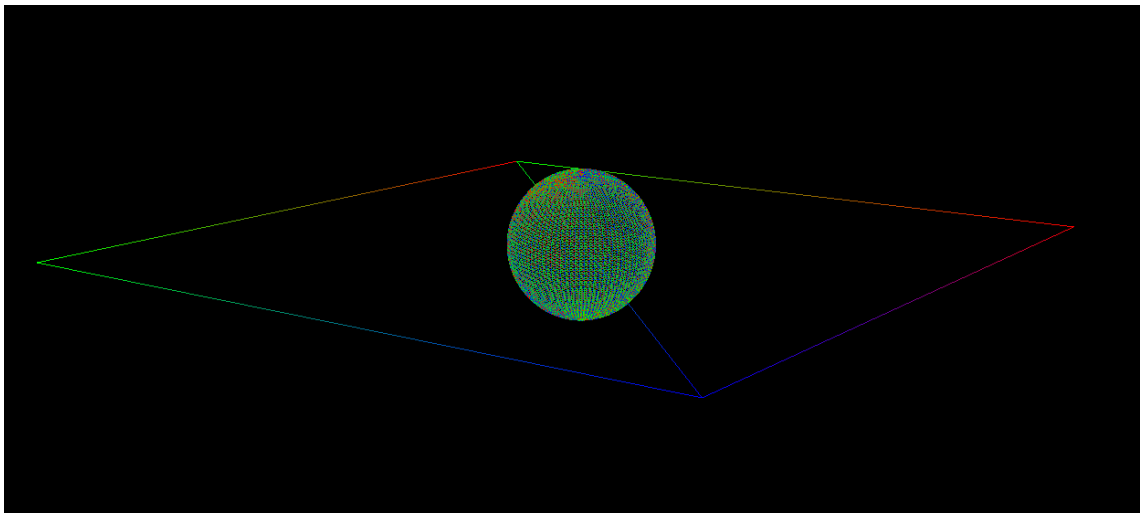


Figura 5.4: Modelos apresentados por linhas

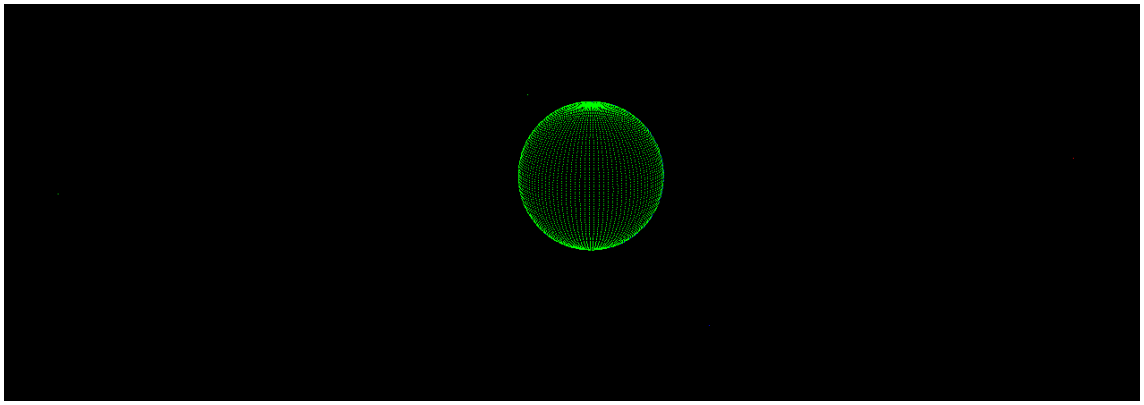


Figura 5.5: Modelos apresentados por pontos

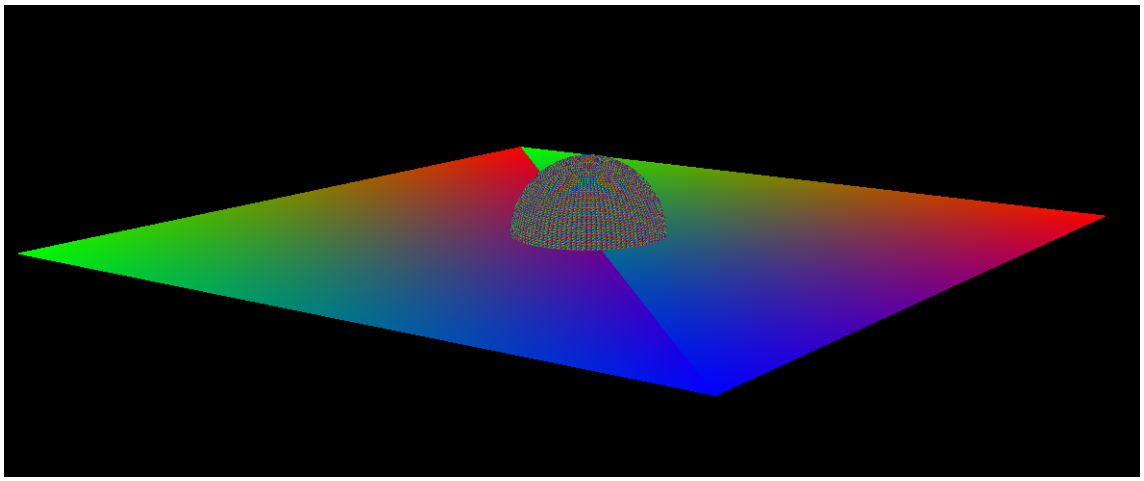


Figura 5.6: Modelos apresentados preenchidamente

Capítulo 6

Modelos 3D

6.1 Plano

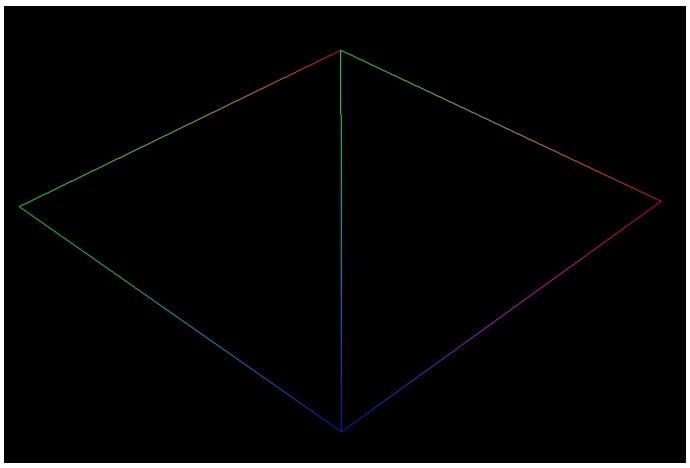


Figura 6.1: Plano com dimensão 10x10

6.2 Paralelepípedo

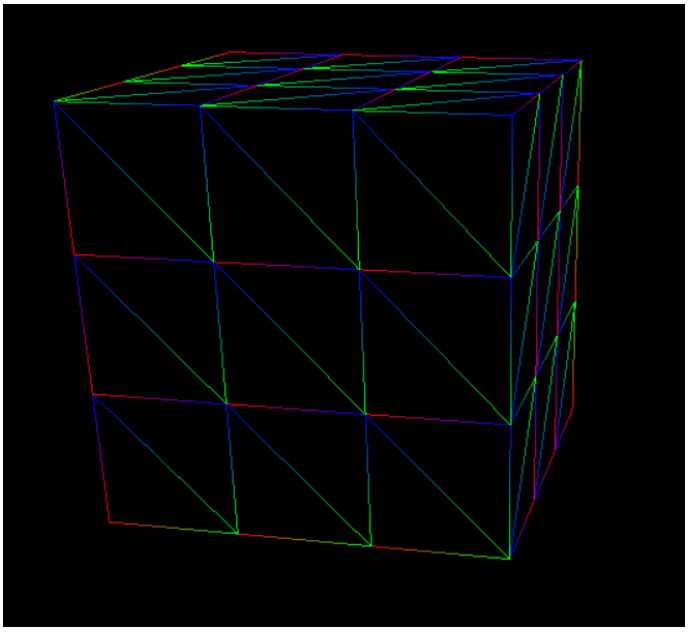


Figura 6.2: Paralelepípedo com 3 de comprimento, 3 de largura, 3 de altura e com 3 divisões

6.3 Esfera

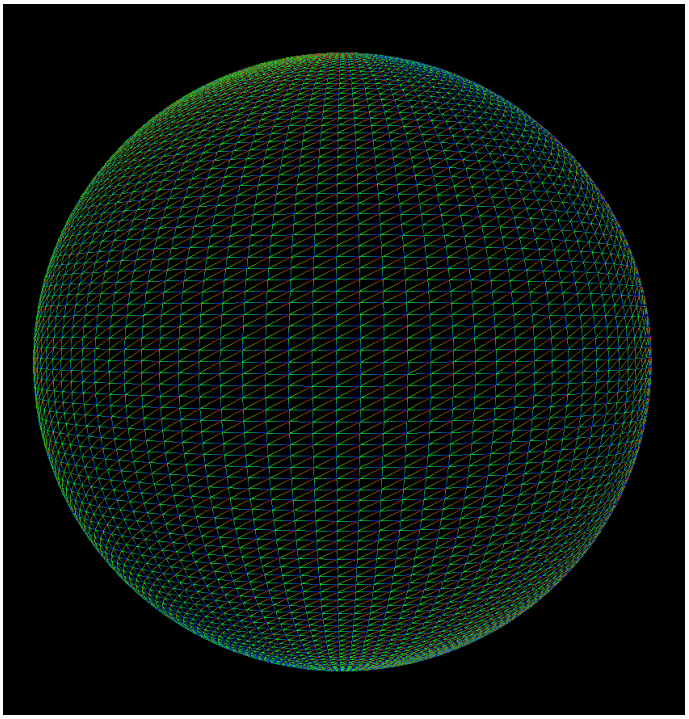


Figura 6.3: Esfera com 1 de raio, 100 fatias e 100 camadas

6.4 Cone

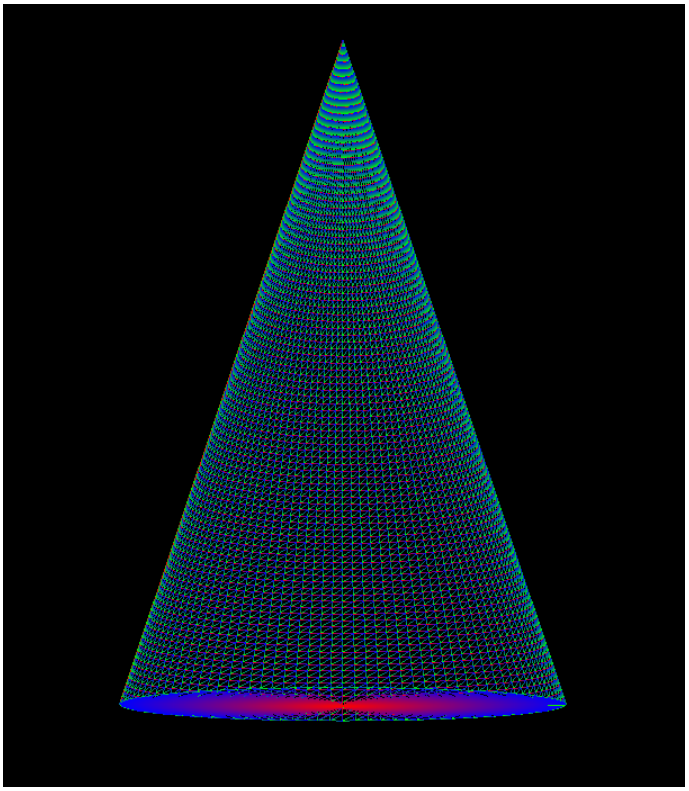


Figura 6.4: Cone com 1 de raio, 3 de altura, 100 fatias e 100 camadas

Capítulo 7

Conclusão

Ao longo desta primeira fase foi possível perceber o modo de funcionamento de um motor gráfico 3D e como o OpenGL atua na criação das diferentes figuras geométricas.
Conseguimos fazer com que todas as primitivas sejam geradas corretamente e que o motor gráfico seja capaz de as representar genericamente.
Fazendo uma análise geral ao trabalho desenvolvido ao longo desta primeira fase, aferimos que foram cumpridos todos os objetivos que nos foram propostos, tendo uma boa base para realizar as restantes fases deste projeto.