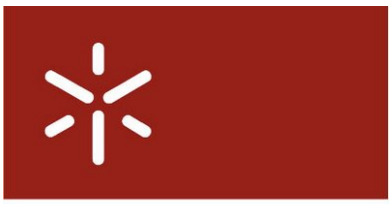


Universidade do Minho

Computação Gráfrica

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

Trabalho Prático

João Leal
A75569

Rui Vieira
A74658

Carlos Faria
A67638

Ricardo Leal
A75411

12 de Março de 2018

Conteúdo

1. *Introdução*

1.1 Contextualização

No âmbito da Unidade Curricular de Computação Gráfica, foi nos prosposto a realização de um mecanismo de 3D onde foram sendo aplicados multiplas ferramentas usadas ao longo do semestre.

Este projeto foi seccionado, em 4 partes, sendo esta a primeira fase que tem como objetivo a criação de algumas primitivas gráficas.

1.2 Resumo

Nesta primeira fase foi necessário a criação de uma aplicação responsável por gerar a informação necessária dos modelos guardando os vertices num ficheiro à qual chamamos de *Generator*, e uma outra aplicação responsável pela leitura da configuração de um ficheiro XML e exibir os respétivos modelos, à qual chamamos de *Engine*.

As primitivas gráficas que serão elaboradas nesta fase são o Plano, Paralelipípedo, Esfera, Cone e o Cilindro sendo a ultima realizada de maneira a explorar capacidades adquiridas nas anteriores.

2. *Arquitetura de Código*

Nesta secção do relatório iremos apresentar a estrutura do mesmo, assim como uma breve introdução às Aplicações, Classes e Primitivas (Modelos) usadas nesta primeira fase.

2.1 Aplicações

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

2.1.1 Generator

generator.cpp - Aplicação onde estão definidas as estruturações das diferentes primitivas geométricas a desenvolver de forma a gerar os respetivos vértices. A aplicação cria uma pasta "files"e guarda os ficheiros gerados dentro da mesma.

2.1.2 Motor

engine.cpp - Aplicação que permite a apresentação de uma janela exibindo os modelos pretendidos, lidos a partir de um ficheiro XML (criado pelo utilizador), e ainda a interação com estes mesmos modelos, através de comandos (teclado e rato) que veremos mais adiante.

2.2 Classes

De maneira a tornar mais fácil, o desenvolvimento das aplicações acima referidas, decidimos criar 2 classes fundamentais, que melhoram a estruturização e a organização do programa. São elas, a Classe **Vertex** e a Classe **Shape**. Além destas duas, construimos uma Classe **Models**, onde estão definidos os algoritmos de criação dos vértices para os modelos que pretendemos gerar.

2.2.1 Vertex

vertex.cpp - Define a estrutura de um vértice, necessário para o desenho de um triângulo. Assim sendo, estão aqui definidas 3 coordenadas: X, Y e Z.

```
#ifndef _VERTEX_H_
#define _VERTEX_H_

#include <string>

class Vertex {

    private:
        float x;
        float y;
        float z;

    public:
        Vertex();
        Vertex(float xx, float yy, float zz);
        Vertex(const Vertex &p);
        Vertex(std::string str);
        float getX();
        float getY();
        float getZ();
        virtual ~Vertex(void);
};

#endif
```

Figura 2.1: Header da classe vertex

2.2.2 Shape

shape.cpp - Define a estrutur aque guarda todos os pontos de um modelo, ou seja, um conjunto de vértices representados na forma *vector<Vertex*>*.

```
#ifndef _SHAPE_H_
#define _SHAPE_H_

#include <vector>
#include "vertex.h"

class Shape {

    private:
        std::vector<Vertex*> vertexes;

    public:
        void pushVertex(Vertex* v);
        void pushShape(Shape* s);
        std::vector<float> *getVertexes();
        void getVertex(int i, Vertex** v);
        void reverse();
        int getSize();
        virtual ~Shape(void);
};

#endif
```

Figura 2.2: Header da classe shape

2.2.3 Models

models.cpp - Guarda todos os algoritmos necessários à criação de cada modelo a querer gerar, com as suas respectivas características.

```
#ifndef _MODELS_H
#define _MODELS_H

#include <vector>
#include <math.h>
#include "../src/shape.h"

Shape* createPlane(float size);
Shape* createBox(float width, float height, float length, int div);
Shape* createCone(float radius, float height, int slices, int stacks);
Shape* createCylinder(float radius, float height, int slices, int stacks);
Shape* createSphere(float radius, int slices, int stacks);

#endif
```

Figura 2.3

2.2.4 TinyXML2

tinyxm2.cpp- Ferramenta utilizada para fazer o parsing dos ficheiros XML de modo a explorar os ficheiros do seu conteudo.

3. *Primitivas Geométricas*

3.1 *Plano*

Um plano pode ser interpretado como 2 triângulos. O Plano necessita de uma dimensão para ser gerado e é centrado na origem, no plano XZ.

3.1.1 Algoritmo

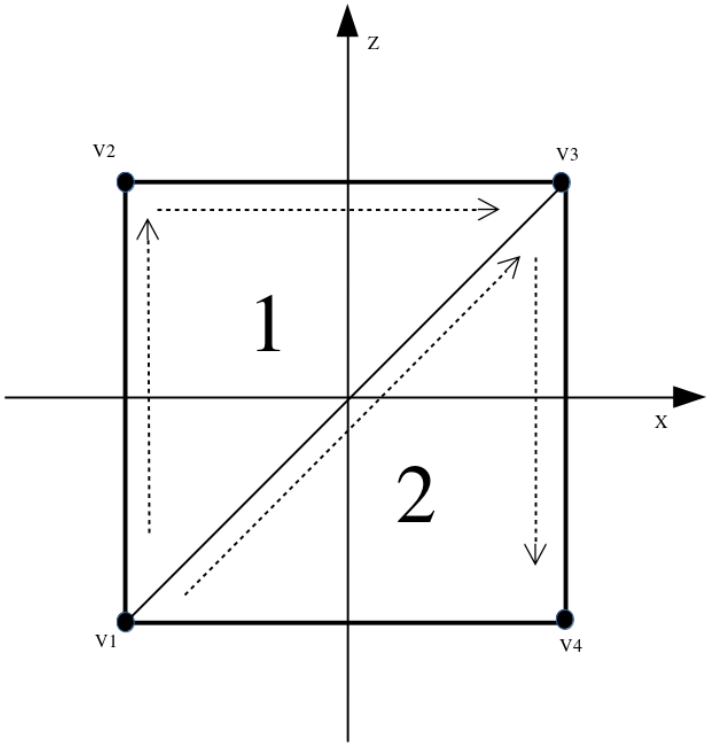
Sendo que, 1 triangulo precisa de 3 vértices para ser definido,verificamos desde já que 2 dos 4 vértices, necessários à definição de um plano, irao ser partilhados por ambos os triângulos. Em seguida, sabendo que este plano iria ser centrado na origem e como este é desenhado em XZ, o Y é uma constante e toma o valor '0'. Tendo isto em conta, prosseguimos para o cálculo das coordenadas X e Z dos vértices do plano, de forma a ficar centrado na origem, que, a partir da dimensão dada pelo utilizador, podem ser facilmente descobertas, da seguinte forma:

v1(x1,0,z1)
x1 = -dimensão/2;
z1 = -dimensão/2;

v2(x2,0,z2)
x2 = -dimensão/2;
z2 = dimensão/2;

v3(x3,0,z3)
x3 = dimensão/2;
z3 = dimensão/2;

v4(x4,0,z4)
x4 = dimensão/2;
z4 = -dimensão/2;



Pela figura apresentada, vemos facilmente a ordem de criação dos triângulos que constituem o plano.

3.1.2 Modelo3D

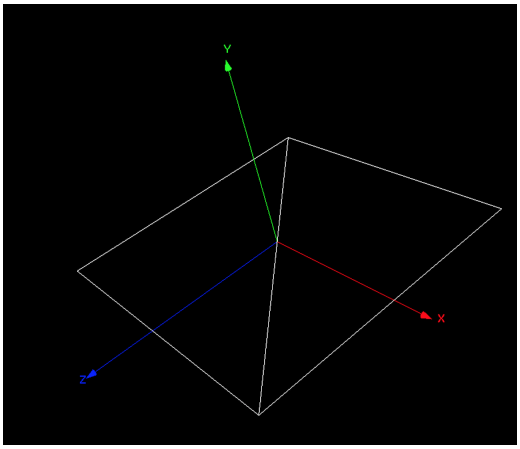


Figura 3.1: Modelo 3D do Plano

3.2 *Paralelipípedo*

Um paralelipípedo, tal como o plano, pode ser interpretado como um conjunto de triângulos. Para este ser gerado, precisamos de uma largura **(X)**, altura **(Y)**, um comprimento **(Z)** e o número de divisões em cada face.

3.2.1 Algoritmo

Para o paralelipípedo, o algoritmo usado é um pouco mais complexo que o do plano, no entanto, é bastante semelhante ao mesmo tempo. Por exemplo, as medidas a serem usadas, X, Y e Z são divididas por 2, de modo a que o cubo seja centrado na origem. Com estas medidas, temos as coordenadas onde as faces da nossa caixa vão ser fixadas. Um paralelipípedo é composto por 3 tipos de faces:

As Faces XY - face da frente e de trás;

As Faces YZ - face da direita e da esquerda;

As Faces XZ - face do topo e da base;

Tendo isto em conta, podemos afirmar que as faces XY, YZ e XZ, não alteram os valores de Z, X e Y, respectivamente. Assim, podemos começar a pensar no cálculo dos vértices do nosso paralelipípedo.

Assumindo que as faces deste podem ser divididas, introduzimos o conceito de deslocamento, para o qual criamos as variáveis *x_shift* , *y_shift* e *z_shift*, que representam o valor que teremos de deslocar, relativamente a cada eixo (a partir do ponto em que estamos) e cujos valores podem ser calculados da seguinte forma:

x_shift = X/divisões;

y_shift = Y/divisões;

z_shift = Z/divisões;

Tomemos como exemplo a face XY, fixa em Z/2. Sabendo os valores dos deslocamentos, pensemos agora de que forma os poderemos aplicar. Tendo em conta as divisões como um limite superior para o número de divisões horizontais (i) e verticais (j), podemos escrever os 4 vértices da seguinte forma:

v1(x1,y1,z)
x1 = -x_pos+(j*x_shift);
y1 = -y_pos+(i*y_shift);

v2(x2,y2,z)
x2 = (-x_pos + x_shift)+(j*x_shift);
y2 = -y_pos+(i*y_shift);

v3(x3,y3,z)
x3 = (-x_pos + x_shift)+(j*x_shift);
y3 = (-y_pos + y_shift)+(i*y_shift);

v4(x4,y4,z)
x4 = -x_pos+(j*x_shift);
y4 = (-y_pos + y_shift)+(i*y_shift);

Para vermos o algoritmo desenvolvido em ação, assim como o uso dos deslocamentos calculados, elaboramos algumas figursa para facilitar a compreensão deste algoritmo:

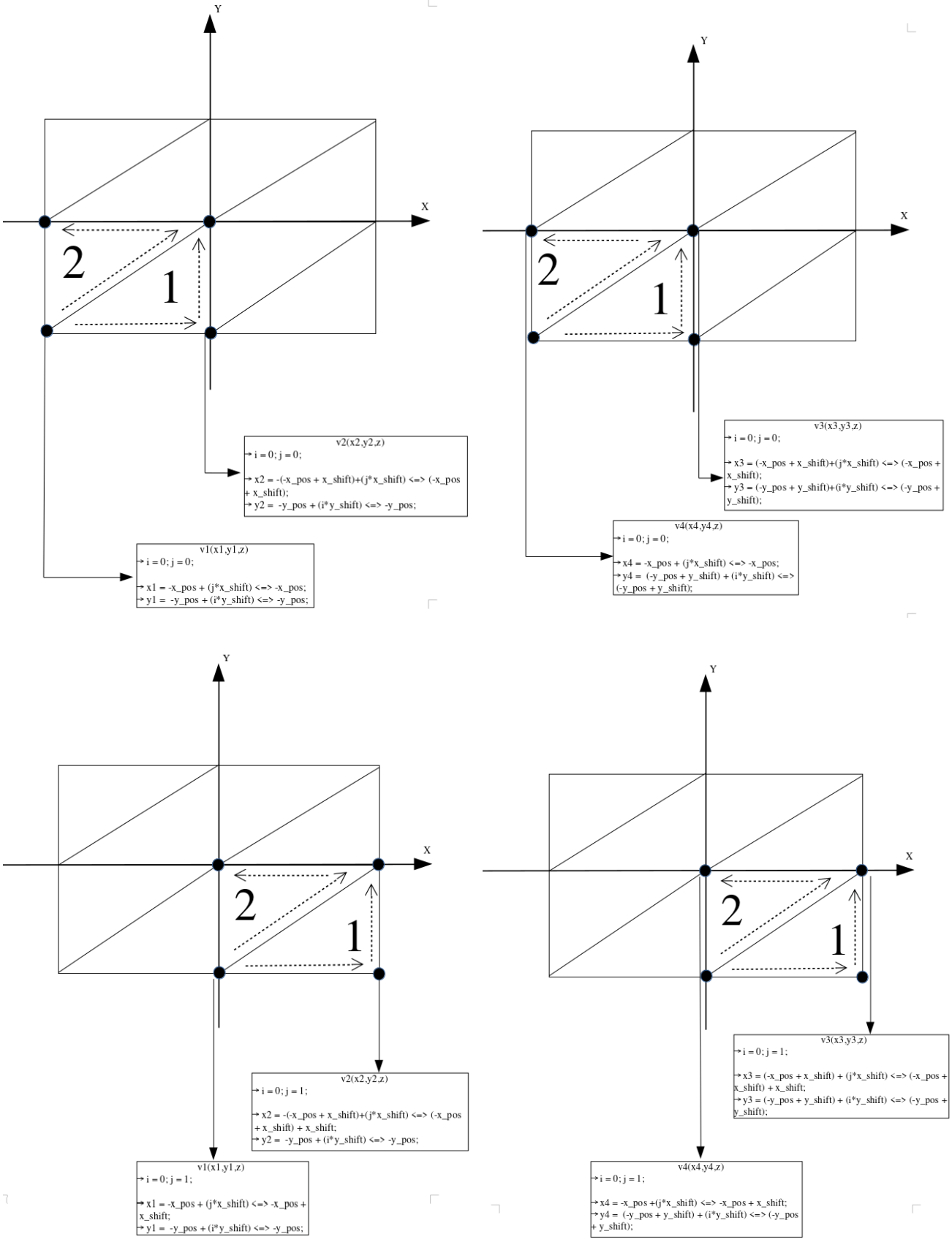


Figura 3.2: Ilustração da contrução do Paralelipípedo

Pelas figuras acima apresentadas vemos facilmente a base do algoritmo usado. São usados dois ciclos *for*, um exterior que percorre cada linha de divisão **'i'** e um interior que percorre cada coluna de divisão **'j'**.

Em primeiro lugar, percorremos a linha inteira, percorrendo todas as colunas geradas pelas divisões introduzidas. Chegando ao fim das divisões, incrementamos o **i** (a altura) e o processo repete-se novamente.

Incrementando j no fim deste ciclo, faz com que j = 2 e, como j = divisões, o ciclo for interior nao será percorrido. Seguindo o ciclo exterior, i incrementa, ou seja, i = 1 e entramos novamente dentro do ciclo for interior, onde j volta novamente a ser 0 e o processo será repetido, desta vez para i = 1, calculando os vertices para uma nova linha.

Os deslocamentos são aplicados da seguinte maneira:

"+ (i*C_shift) ou +(j*C_shift)-> de maneira a deslocarmo-nos à coordenada atual, tendo em conta em que linha ou em que coluna estamos.

"+ C_shift-> de maneira a deslocarmo-nos a uma coordenada futura, vizinha da nossa coordenada atual.

→ + y_shift diz respeito a um ponto, um deslocamento acima do atual.
→ + x_shift diz respeito a um ponto, um deslocamento à direita do atual.
→ + z_shift diz respeito a um ponto, um deslocamento à frente da atual.

Assim sendo, conclui-se que as faces do nosso paralelipípedo são desenhadas linha a linha. Para as faces opostas (fixas nos valores negativos da variável fixa respectiva), e de modo a que estas sejam voltadas para fora, decidimos calcular os vértices da mesma forma, e inverte-los todos no fim.

3.2.2 Modelo3D

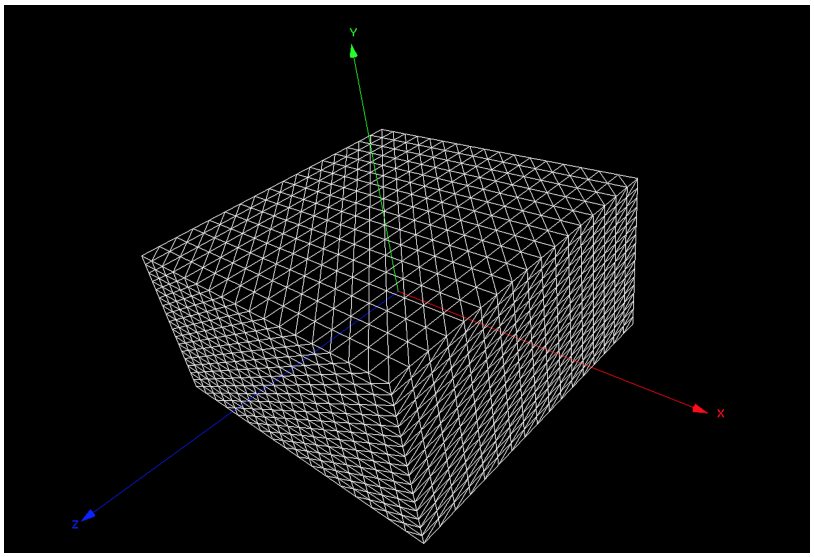


Figura 3.3: Modelo 3D de um Paralelipípedo

3.3 *Cone*

Um cone, assim como as primitivas acima faladas, pode ser também representado com triângulos, mas este, requer um algoritmo, não muito complexo, mas que introduz cálculos trigonométricos para nos ajudar no cálculo das coordenadas dos vértices. Para este ser gerado, um cone necessita de um raio **'r'**, uma altura **'h'**, número de divisões verticais (***slices***) e um número de divisões horizontais (***stacks***).

3.3.1 Algoritmo

O Cone pode ser dividido em duas partes: a ***Base*** e o ***Plano Lateral Curvo***.

Para a *Base*, fixamos uma altura e um ponto inicial, que neste caso será (0,-h/2,0), de modo a ficar centrado na origem, servindo então como referência para o resto do cone. Para desenharmos a circunferência que compõe a base do cone, temos de ter em conta o número de (***slices***) dadas, pois estas dividem a nossa base em *n*. Assim, e sabendo que a circunferência tem 2π radianos (que corresponde a 360^o), podemos dividir essa totalidade angular pelo número de *slices*, obtendo assim o deslocamento angular (α_shift), necessário para o cálculo das coordenadas X e Z da nossa base. Estas, podem ser calculadas da seguinte forma:

- Para o vértice atual:
x = r * sin(α);
z = r * cos(α);

- Para o proximo vértice:
x = r * sin(α + α_shift);
z = r * cos(α + α_shift);

O nosso α começa em 0 e percorre toda a circunferência que compõe a nossa base, acumulando o valor do deslocamento angular (α_shift) a cada iteração (de slice em slice). Assim, os vértices que compõe os triângulos da nossa base, são definidos da seguinte forma: vb1(xb1,-h/2,zb1)
xb1 = r * sin(α);
zb1 = r * cos(α);

vb2(0,-h/2,0)

vb3(xb3,-h/2,zb3)
xb3 = r * sin(α + α_shift);
zb3 = r * cos(α + α_shift);

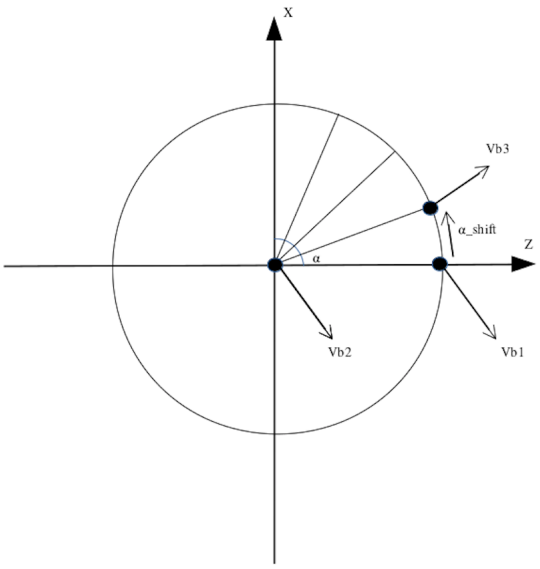


Figura 3.4: Ilustração da construção da base do Cone

Tendo a *Base* definida, falta pensar no *Plano Lateral Curvo* do nosso cone. Ora, sabemos que um cone, converge desde a nossa base até à nossa altura máxima (h/2), onde o raio será 0. Assim, teremos de decrementar o raio de ***stack*** em ***stack***. Concluímos portanto, que para além de um deslocamento angular, precisaremos de calcular um deslocamento radial, com a função de decrementar o raio. Tal pode ser calculado, dividindo o nosso raio, pelo número de *stacks*, obtendo assim o nosso *r_shift*. Além disso, precisamos também de saber o deslocamento vertical necessário, para subir ao longo do nosso eixo Y, que marca a altura da nossa primitiva. Este deslocamento, chamemos de *h_shift* é calculado dividindo h (altura do cone) pelo número de *stacks*.

Portanto:
r_shit = r/stacks;
h_shift = h/stacks;

v1(x1,y1,z1)
x1 = r * sin(α);
y1 = -h/2;
z1 = r * cos(α);

v2(x2,y2,z2)
x2 = r * sin(α + α_shift);
y2 = -h/2;
z2 = r * cos(α + α_shift);

v3(x3,y3,z3)
x3 = (r - r_shift) * sin(α);
y3 = -h/2 + h_shift;
z3 = (r - r_shift) * cos(α);

v4(x4,y4,z4)
x4 = (r - r_shift) * sin(α + α_shift);
y4 = -h/2 + h_shift
z4 = (r - r_shift) * cos(α + α_shift);

Podemos ver pela figura acima representada o processo de criação do nosso *Plano Lateral Curvo*. Primeiro é percorrida a totalidade da circunferência, com o nosso α e o α_shift, onde, por cada *slice* este deslocamento é adicionado ao nosso α, até este chegar novamente a 2π (360º). Atingindo esse ponto, incrementamos a nossa altura e decrementamos o nosso raio atual, e voltamos a percorrer o plano horizontalmente, usando novamente o nosso α (que inicializa no 0).

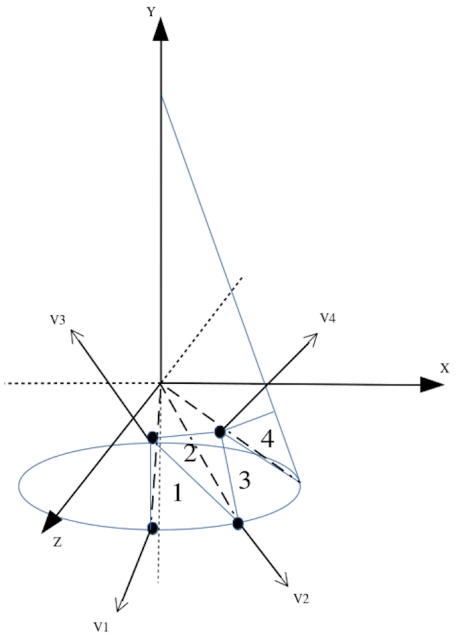


Figura 3.5: Ilustração do plano lateral do Cone (os numeros apresentados no triângulos servem para indicar a ordem de criação dos mesmo)

3.3.2 Modelo3D

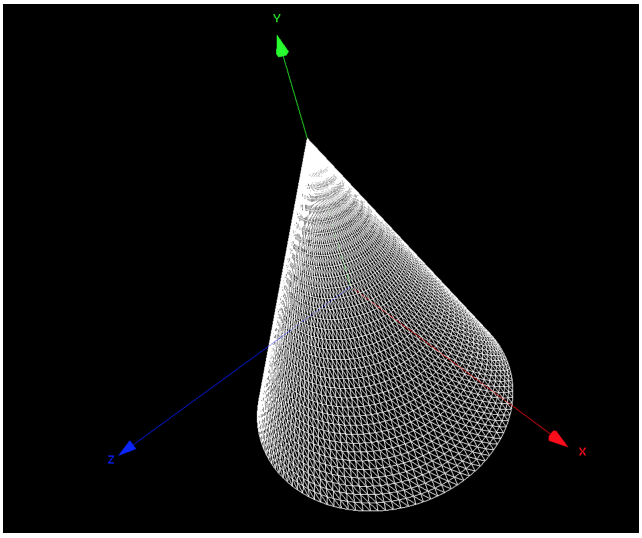


Figura 3.6: Modelo 3D do Cone

3.4 *Esfera*

A Esfera requer um algoritmos mais complexo para a criação dos vértices que a formam, introduzindo novas formulas trigonométricas necessárias. Para esta ser definida necessitamos de um raio **'r'**, um número de divisões verticais **'slices'** e um número de divisões horizontais **'stacks'**.

3.4.1 Algoritmo

Para calcular os vertices que compõem a esfera, necessitamos de calcular os deslocamentos que esta irá usar. Temos 2 deslocamentos possiveis de calcular, o vertical, de 0º a 180º (β*_shift*) e o horizontal, de 0º a 360º (α*_shift*), tais são dados por:

α_shift = 2π / slices;
β_shift = π / stacks;

A esfera começa por ser percorrida a partir de cima do eixo Y.

Assim, e com estes deslocamentos podemos recorrer ao cálculo dos vertices, slice a slice, em que, seguindo o mesmo esquema do cone, em que o ângulo α é incrementado a cada iteração no valor de α_shift e, quando este chegar a 2π (der uma volta de 360º ao longo da esfera), é incrementado o angulo β, recorrendo ao β_shift, e o nosso α é inicializado a partir do 0, novamente, e é mais uma vez percorrida a esfera horizontalmente, até ao nosso ângulo β chegar a π (que corresponde a 180º, a curvatura vertical da esfera).

Para chegar aos valores que cada coordenada desses vértices vão tomar, recorremos ao uso das coordenadas esféricas, onde a partir delas, conseguimos retirar as coordenadas cartesianas das mesmas, através das seguintes fórmulas:

x = r * sin(α) * sin (β);
y = r * cos(β);

z = r * cos(α) * sin (β);

A partir delas, podemos garantir os seguintes cálculos, que geram a esfera, respeitando toda a sua curvatura:

v1(x1,y1,z1)
x1 = r * sin(α) * sin(β);
y1 = r * cos(β);
z1 = r * cos(α) * sin(β);

v3(x3,y3,z3)
x3 = r * sin(α + α_shift) * sin(β);
y3 = r * cos(β);
z3 = r * cos(α + α_shift) * sin(β);

v2(x2,y2,z2)
x2 = r * sin(α + α_shift) * sin(β + β_shift);
y2 = r * cos(β + β_shift);
z2 = r * cos(α + α_shift) * sin(β + β_shift);

v4(x4,y4,z4)
x4 = r * sin(α) * sin(β + β_shift);
y4 = r * cos(β + β_shift);
z4 = r * cos(α) * sin(β + β_shift);

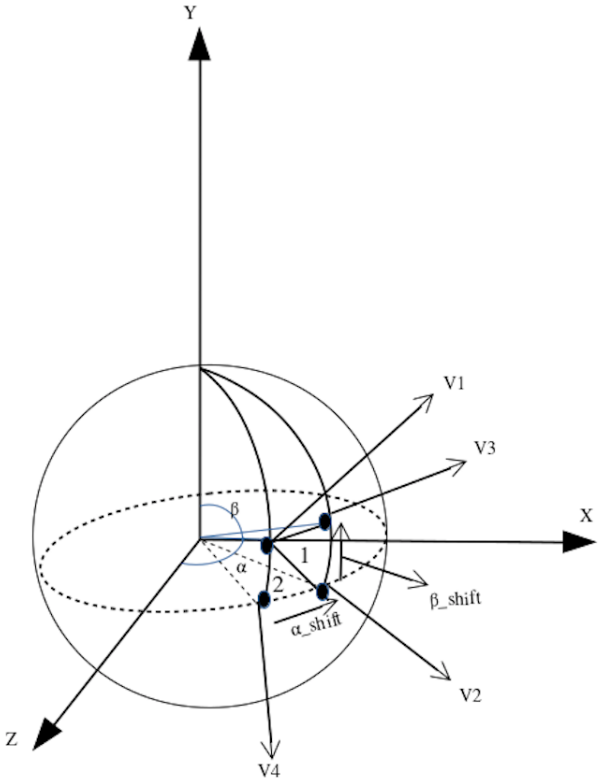


Figura 3.7: Ilustração da construção da esfera

3.4.2 Modelo3D

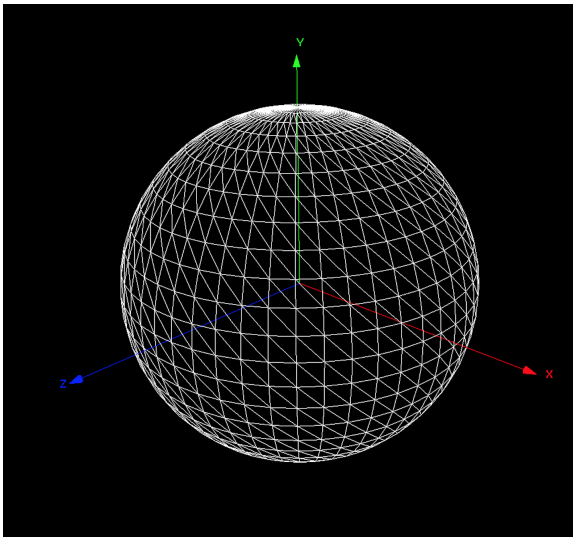


Figura 3.8: Modelo 3D da esfera

3.5 *Extra*

3.5.1 *Cilindro*

O Cilindro, é um sólido muito semelhante ao Cone e, tal como este, necessita de um raio **'r'**, uma altura **'h'**, um número de divisões verticais (***slices***) e um número de divisões horizontais (***stacks***).

3.5.2 Algoritmo

O algoritmo usado para a criação de um cilindro é muito semelhante ao usado para criar um cone, pois a única alteração que precisa de ser feita é do decremento do raio. Num cilindro, o raio mantém-se desde a base ao topo, logo, é desnecessária a criação de um deslocamento radial, pois o nosso raio será o mesmo ao longo da definição de vértices.

3.5.3 Modelo3D

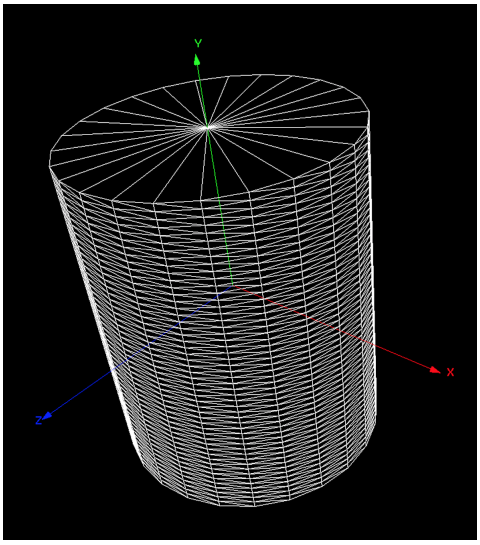


Figura 3.9: Modelo 3D do Cilindro

4. *Generator*

4.1 Descrição

O gerador é responsável por gerar ficheiros quem contém o conjuntos dos vertices das primitivas gráficas que pretendemos gerar, seguindo os parametros escolhidos. Vértices esses que são que são conjuntos de 3 pontos, pois todas as primitivas têm como unidade construção o triângulo.

4.2 Usabilidade

De seguida será apresentado o manual de ajuda do generator que pode ser acedido através do comando *./generator -help*.

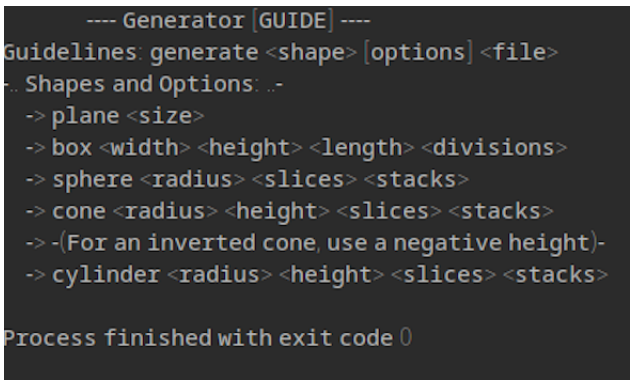


Figura 4.1: Manual de ajuda do generator.

5. *Engine*

5.1 Descrição

O engine é responsavél pela leitura de um ficheiro XML, que indica que ficheiros fazer parsing de modo a interpretar e apresentar graficamente os modelos. Ficheiros esses que anteriormente criados pelo generator.

5.2 Usabilidade

De seguida será apresentado o manual de ajuda do engine que pode ser acedido através do comando *./engine -help*.

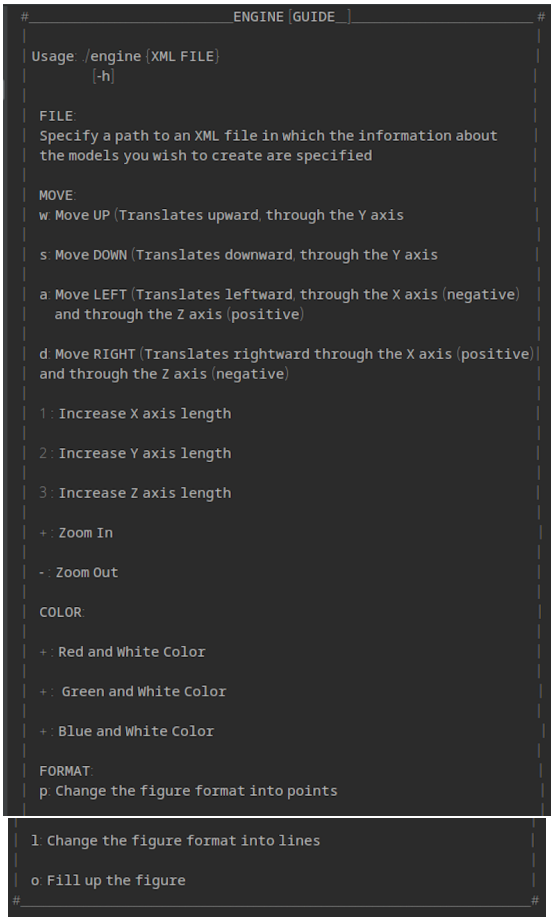


Figura 5.1: Manual de ajuda do engine

6. *Conclusão*

Esta primeira fase do trabalho, na nossa prespetiva foi bastante importante pois permitiu consolidar conhecimentos abordados na UC, familiarizar-nos com as ferramentas usadas em modulação 3D e em simultanio aprender C++. Em suma, esperamos que esta fase nos sirva de motivação para as que seguem pois disfrutamos de a realizar.