

Computação Gráfica

Trabalho Prático - Fase 1

15 de março de 2021



Patrícia Pereira, A89578



Meriem Khammasi, A85829



Jaime Oliveira, A89598



Luís Magalhães, A89528

Conteúdo

1	Abstract	1
2	Introdução	1
3	Organização do Código	1
3.1	<i>Generator</i>	1
3.2	<i>Engine</i>	2
4	Figuras Geométricas	2
4.0.1	PLANE	2
4.0.2	Interpretação do plano no <i>Engine</i>	3
4.0.3	BOX	3
4.0.4	Interpretação da caixa no <i>Engine</i>	6
4.0.5	CONE	7
4.0.6	Base do Cone	7
4.0.7	Parte de cima do Cone	8
4.0.8	Interpretação do cone no <i>Engine</i>	9
4.0.9	SPHERE	10
4.0.10	Interpretação da esfera no <i>Engine</i>	12
5	Parser XML	13
5.1	Ficheiros de configuração	13
5.2	Carregamento dos ficheiros	13
6	Demonstração	14
6.1	Generator	14
6.2	Engine	14
6.3	Guideline	15
7	<i>Engine</i>	15
7.1	Desenho dos pontos das figuras	15
7.2	Posição da Câmara	16
7.3	Inputs do Teclado	16
8	Conclusão	18

1 Abstract

No âmbito da Unidade Curricular de Computação Gráfica, numa fase inicial, foi nos proposto o desenvolvimento de um *Engine* de gráficos 3D com o intuito de formar diferentes formas geométricas, nomeadamente uma caixa, um plano, um cone e uma esfera.

Esta fase é composta por duas aplicações, uma que irá gerar os ficheiros com as informações dos modelos 3D e outra que serve como *Engine* gráfico para a leitura de ficheiros XML, apresentando os modelos 3D solicitados.

Para atingir os objetivos propostos, utilizamos ferramentas como o OpenGL e C++.

2 Introdução

Neste Relatório vamos esclarecer os aspetos mais importantes relativamente à primeira fase do nosso trabalho, onde faremos uma síntese e explicação do código elaborado e dos resultados obtidos.

Iremos também descrever e explicar as abordagens que tomamos quanto à realização do *Generator* e do *Engine* bem como o funcionamento de cada um, apresentando imagens de exemplo para facilitar a compreensão.

Para tal, especificaremos os diversos raciocínios utilizados para conseguir alcançar os objetivos pretendidos.

3 Organização do Código

Após a análise do problema em questão, verificamos que a melhor solução seria a implementação de duas aplicações, o *Generator* e o *Engine*.

3.1 *Generator*

O *Generator* é responsável por gerar os pontos dos triângulos que permitem construir as diversas figuras. Para além disso, nesta classe encontram-se também os diferentes algoritmos para gerar os pontos que permitem obter as figuras pretendidas, através de triângulos. Assim, as primitivas que se pretendem implementar são:

- cone(float raio, float altura, int fatias, int camadas)
- box(float largura, float altura, float comprimento, int d)
- sphere(float raio, int fatias, int camadas)
- plane(float s)

Tendo em conta que cada ponto é constituído por três coordenadas, tornou-se importante criar uma estrutura de dados [Point] para representá-los.

```
typedef struct point {
    float x;
    float y;
    float z;
}Point;
```

Figura 1: Struct Point.

3.2 *Engine*

O *Engine* está encarregue de realizar a leitura de um ficheiro XML, de modo a obter ficheiros .3d que contêm os pontos que permitem gerar as diferentes figuras. Assim, é efetuada uma leitura deste para um vetor de pontos que permitirá exibir as figuras. Será ainda possível interagir com estas utilizando diferentes comandos.

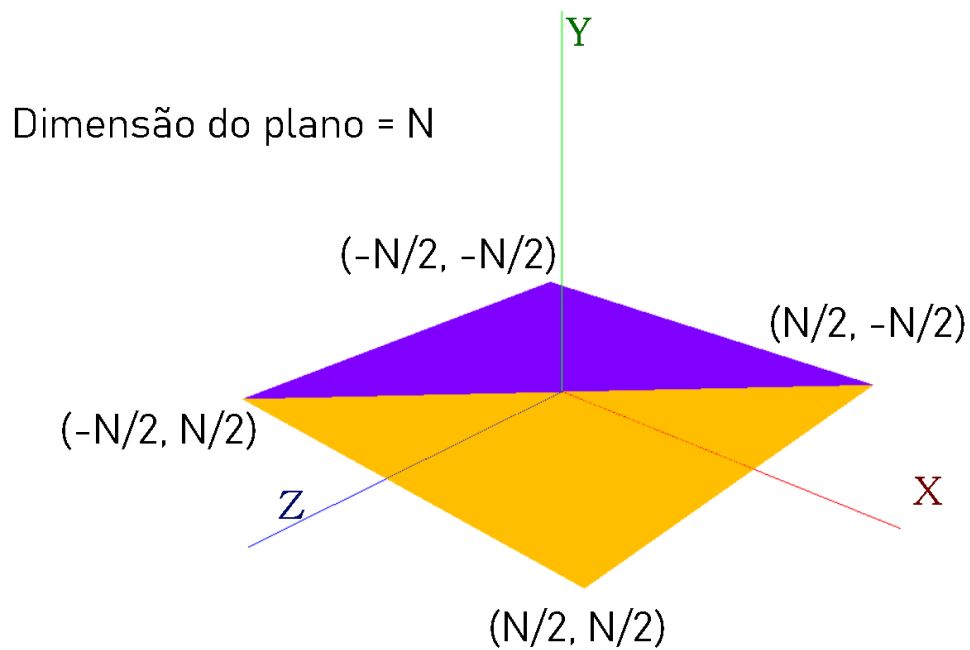
4 Figuras Geométricas

4.0.1 PLANE

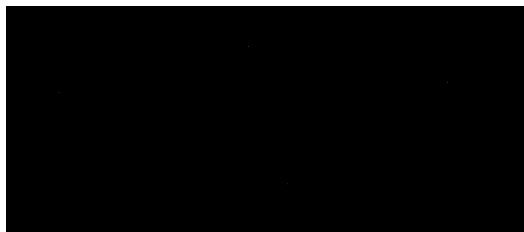
O plano é constituído visualmente por 2 triângulos que em conjunto desenharam um quadrado. Este plano tem origem no centro do referencial e é perpendicular ao eixo Y.

Esta primitiva é criada usando a função "plane" do ficheiro "generator.cpp" e recebe como argumentos um float que representa a dimensão do lado plano quadrangular. Esta função irá desenhar no OpenGL 12 pontos (para fazer um total de quatro triângulos), tendo em conta a regra da mão direita, dois desses triângulos irão constituir a face de cima do plano e os restantes dois vão fazer a face de baixo (isto é feito para que o plano seja visível de todos ângulos).

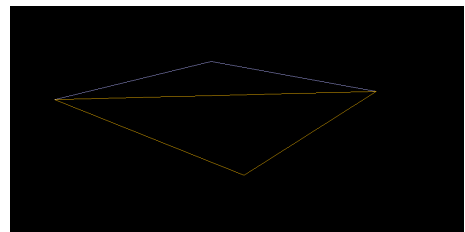
Cada ponto irá variar apenas as suas coordenadas X e Z (Y permanece constante entre todos os pontos), os valores destas duas coordenadas apenas podem ter valores que tenham como valor absoluto metade do valor da dimensão que foi fornecida à função. Desta forma, todos os pontos serão equidistantes à origem, o que resulta na representação de um quadrado.



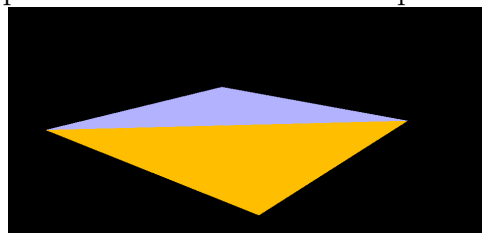
4.0.2 Interpretação do plano no *Engine*



Representação picotada.



Representação por linhas.



Representação completa.

4.0.3 BOX

O processo de desenho da caixa acaba por ser semelhante ao do plano até um certo ponto. O método que usamos para desenhar o plano vai acabar por ser usado várias vezes durante a construção da caixa, no entanto é necessário ter em conta o facto de que a caixa

é constituída por 6 faces e cada face pode conter um certo número de divisões horizontais e verticais (por exemplo: se a caixa for feita com o número de divisões igual a 3, então cada face será dividida em 9 secções iguais).

A função "box", do ficheiro "generator.cpp", recebe os seguintes argumentos:

- a largura da caixa (um float que representa as dimensões da caixa em função do eixo X)
- a altura da caixa (um float representante das dimensões da caixa em função do eixo Y)
- o comprimento da caixa (float que representa das dimensões da caixa mas em função do eixo Z)
- um int "d" (este inteiro indica o número de divisões da caixa)

Para explicar o processo que usamos para desenhar esta primitiva iremos usar como exemplo a construção de uma face (neste caso a face da frente da caixa, que é perpendicular ao eixo Z e aponta para o seu sentido positivo).

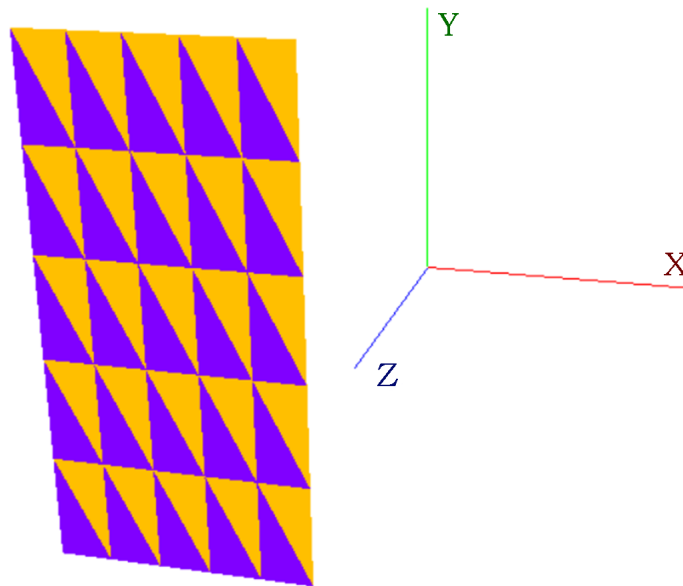


Figura 2: Representação da face da frente

Para começar é importante apontar que cada secção da face vai ser feita de dois triângulos que em conjunto formam um retângulo. As quatro coordenadas necessárias para a representação deste retângulo são dependentes do valor de duas variáveis: uma variável x , que indica a posição dos pontos em relação ao eixo X, e a variável y , que indica a posição dos pontos em função do eixo Y.

O que diferencia cada ponto será o valor adicionado a estas variáveis x e y , para qual será usado duas constantes (a e l).

$$a = altura/d$$

$$l = largura/d$$

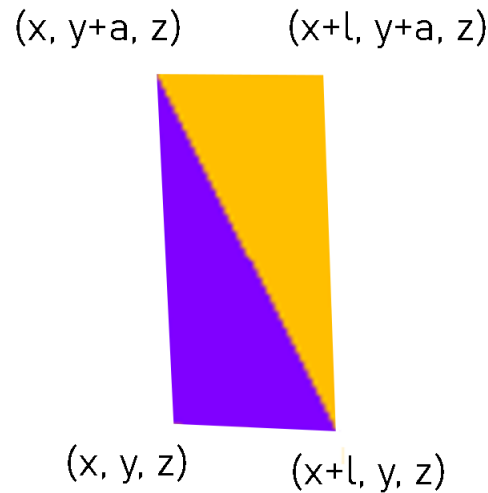


Figura 3: Coordenadas de uma secção da face frontal

Com este conhecimento é possível criar uma estrutura dentro da função capaz de desenhar a face inteira usando dois ciclos (um contido dentro do outro).

O ciclo interior será responsável por desenhar uma linha de secções retangulares ao longo do eixo x , este passo será executado o quantas vezes quanto o número de divisões "d".

O ciclo exterior irá repetir o ciclo interior "d" vezes, mas em cima das linhas que foram desenhadas anteriormente.

```

//frente da box
x = -largura / 2.0f;
y = -altura / 2.0f;
z = comprimento / 2.0f;

for (i = 0; i < d; i++) {
    for (j = 0; j < d; j++) {
        //triângulos amarelos
        ponto(x + l, y, z);
        ponto(x + l, y + a, z);
        ponto(x, y + a, z);

        //triângulos roxos
        ponto(x, y, z);
        ponto(x + l, y, z);
        ponto(x, y + a, z);

        ox += l;
    }
    ox = -largura / 2.0f;
    oy += a;
}

```

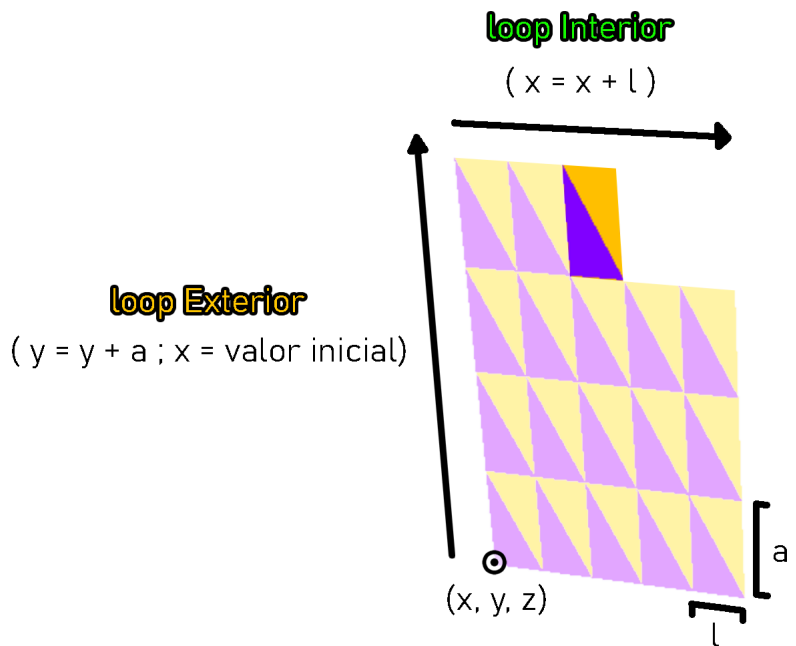
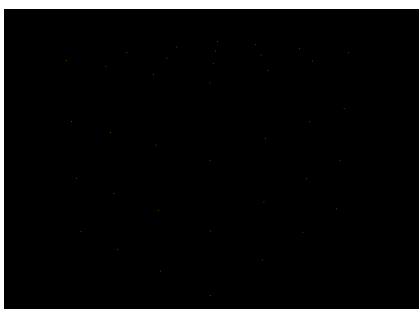


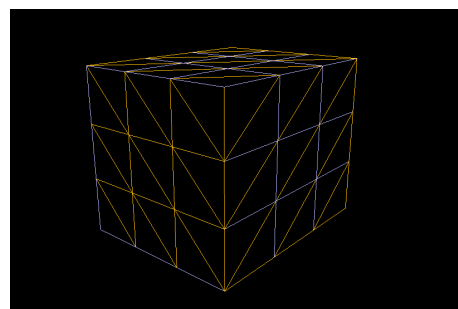
Figura 4: Implementação da estrutura em pseudo-código)

Este processo terá que ser repetido para cada face da caixa. E em cada representação é necessário ter alguns cuidados com as diferenças de certos parâmetros (ordem de desenho dos pontos, a coordenada inicial para o desenho da face inteira e quais as variáveis que devem ser alteradas e quais devem permanecer constantes, etc.).

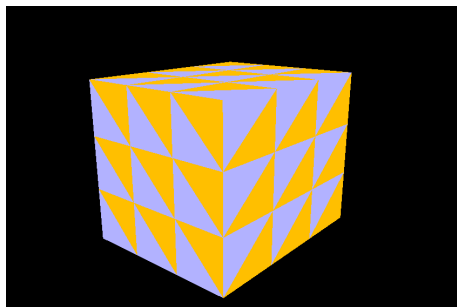
4.0.4 Interpretação da caixa no *Engine*



Representação picotada.



Representação por linhas.



Representação completa.

4.0.5 CONE

Para representar um cone é preciso ter em consideração que ele é composto por uma base circular (que neste caso será apenas uma aproximação dependente do número de slices) com um determinado raio. As respectivas faces laterais (uma por cada slice) ligam a base a um determinado vértice (a distância entre este e a base determina a altura), e cada face será dividida em várias secções dependendo do número de stacks.

4.0.6 Base do Cone

Para desenhar a base desta primitiva foi definida uma estrutura dentro da função "cone" que se foca em desenhar um círculo, mas como foi mencionado anteriormente, será apenas possível desenhar uma aproximação de um círculo que neste caso será um polígono com o número de lados igual ao número de slices.

Esta estrutura baseia-se num ciclo que irá ter o mesmo número de iterações que o número de slices e, por cada iteração será desenhado um triângulo que liga o centro da base a dois pontos de referência. Estes dois pontos de referência são destacados por uma variável destinada apenas a cada um desses pontos. Estas variáveis representam os ângulos formados pelos pontos em radianos. A representação desses pontos será feita da seguinte maneira.

$$A(\cos(ang2), 0, \sin(ang2))$$

$$B(\cos(ang1), 0, \sin(ang1))$$

Tendo sempre em atenção que é obrigatório manter sempre a seguinte regra:

$$ang2 - ang1 = 2\pi/slices$$

No final de cada iteração do ciclo, é adicionado às variáveis $ang1$ e $ang2$ o valor do ângulo (em radianos) formado pelo pontos ACB:

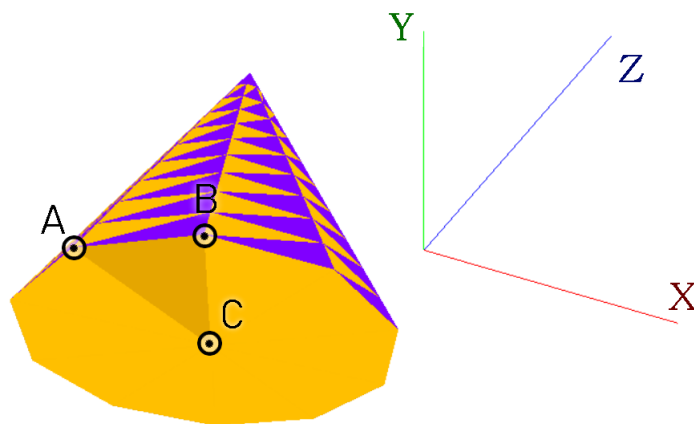


Figura 5: Demonstração da Base do Cone

```

oy = - (altura / 2.0f);
rn = raio;

for (i = 0; i < fatias; i++) {

    p1.x = (cos(ang1) + p.x);
    p1.y = oy + p.y;
    p1.z = sin(ang1) + p.z;
    p2.x = (cos(ang2) + p.x);
    p2.y = oy + p.y;
    p2.z = sin(ang2) + p.z;
    p3.x = p.x;
    p3.y = oy + p.y;
    p3.z = p.z;
    points.push_back(p1);
    points.push_back(p2);
    points.push_back(p3);

    ang1 += (2.0f * M_PI / (float)fatias);
    ang2 += (2.0f * M_PI / (float)fatias);

}

```

Figura 6: Segmento do código do projeto para a base do cone

4.0.7 Parte de cima do Cone

Esta parte de cone irá usar um processo bastante semelhante ao que foi mencionado para a representação de uma face da caixa. Serão implementados na função "cone" dois ciclos (um contido dentro do outro).

O ciclo interior irá desenhar uma stack do cone, cada stack será feita de dois anéis de pontos (semelhante ao foi feito com a base do cilindro), cada anel irá ter um valor de raio diferente.

$$raioF - raioE = raio/stacks$$

O loop exterior vai desenhar novas stacks por cima das que já foram desenhadas (diminuindo o valor dos raios dos anéis de pontos e aumentando a coordenada y dos pontos que vão ser desenhados).

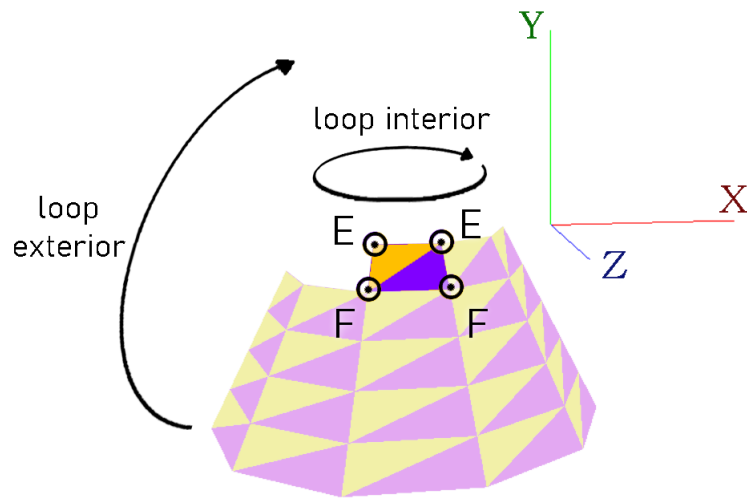
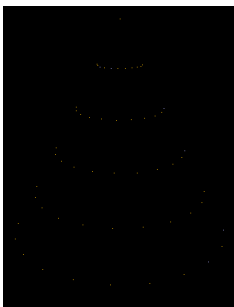
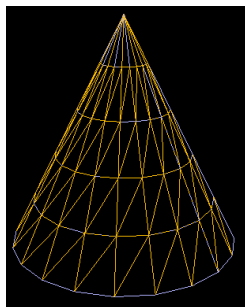


Figura 7: Demonstração da Base do Cone

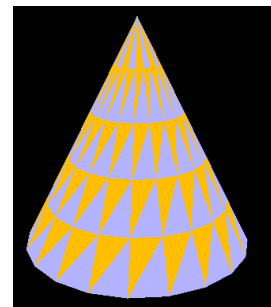
4.0.8 Interpretação do cone no *Engine*



Representação picotada.



Representação por linhas.



Representação completa.

4.0.9 SPHERE

Por último, a primitiva da esfera partilha várias componentes das que foram faladas atrás, a esfera é composta por um certo número de stacks e slices e possui um determinado valor de raio (do tipo float). A esfera também será desenhada usando mesmo método do cone, um ciclo interior para desenharmos uma stack e um ciclo exterior para desenharmos várias stacks uma em cima da outra.

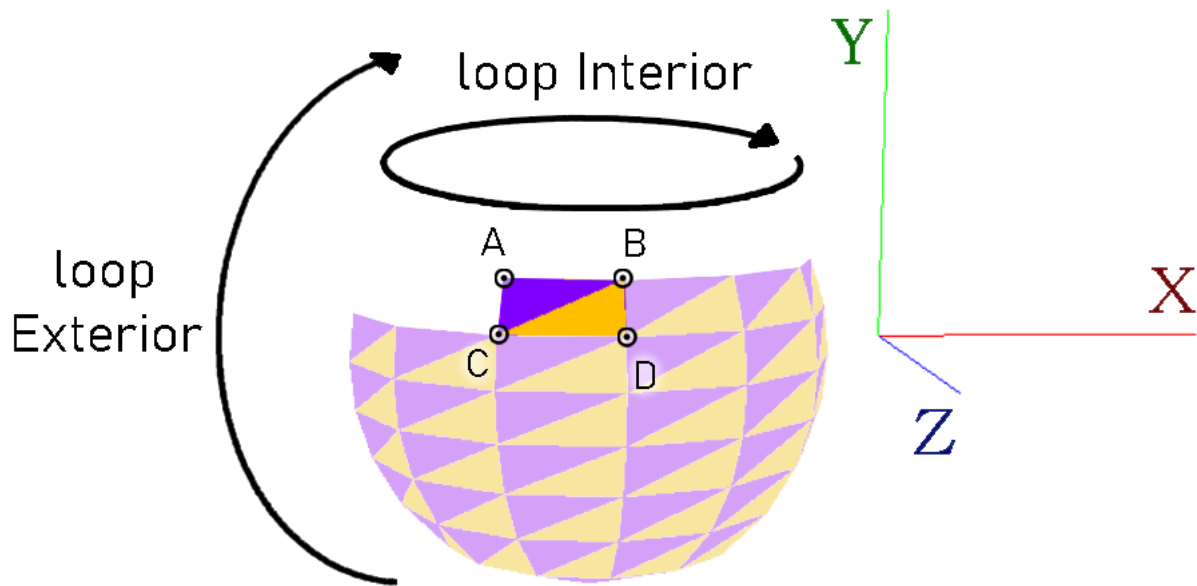


Figura 8: Demonstração da Construção da Esfera

O principal destaque da construção da esfera, é a forma de como algumas operações são alteradas no loop interior e exterior.

O loop exterior é definido logo no início os valores dos raios que serão usados para desenharmos uma stack. Desta vez os valores dos raios dos dois anéis que fazem uma stack são dependentes da expressão matemática que constrói uma esfera.

$$raio^2 = x^2 + y^2$$

O raio corresponde ao raio da esfera valor y corresponde à coordenada Y de todos os pontos de um anel da stack que está a ser desenhada. Se o valor x for posto em evidência, é possível obter o valor dos raios para cada anel da stack.

```
for (i = 0; i < camadas ; i++) {

    or1 = sqrt(raio * raio - oy * oy);
    or2 = sqrt(raio * raio - (oy + a) * (oy + a));
```

Figura 9: Segmento do código do projeto para calcular os valores dos raios da stack

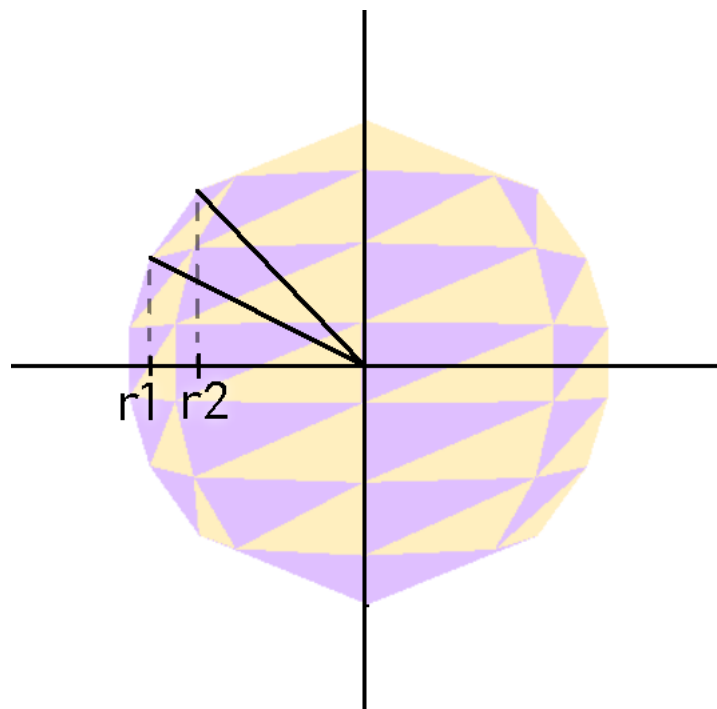


Figura 10: Exemplo do calculo a ser efetuado

Por sua vez o loop interior, em vez de desenhar sempre um par de triângulos junto de cada vez de forma a um retângulo de cada vez. Esta estrutura vai desenhando os triângulos complementares no sentido oposto. A primeira parte do código vai desenhando triângulos de baixo para cima (triângulos roxos), a segunda metade desenha os seus triângulos complementares, mas no sentido oposto que é de cima para baixo (triângulos amarelos).

```

for (j = 0; j < fatias; j++) {

    p1.x = cos(ang2) * or1;
    p1.y = oy;
    p1.z = sin(ang2) * or1;
    p2.x = cos(ang1) * or2;
    p2.y = oy + a;
    p2.z = sin(ang1) * or2;
    p3.x = cos(ang2) * or2;
    p3.y = oy + a;
    p3.z = sin(ang2) * or2;

    points.push_back(p1);
    points.push_back(p2);
    points.push_back(p3);

    p1.x = cos(ang1) * or1;
    p1.y = -oy;
    p1.z = sin(ang1) * or1;
    p2.x = cos(ang2) * or2;
    p2.y = -oy - a;
    p2.z = sin(ang2) * or2;
    p3.x = cos(ang1) * or2;
    p3.y = -oy - a;
    p3.z = sin(ang1) * or2;
    points.push_back(p1);
    points.push_back(p2);
    points.push_back(p3);

    ang1 += 2.0f * M_PI / (float)fatias;
    ang2 += 2.0f * M_PI / (float)fatias;

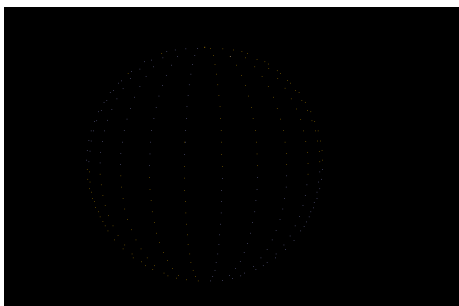
}

```

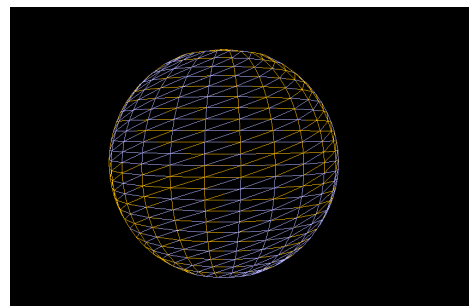


Figura 11: Exemplificação de quais triângulos são desenhados pelo código.

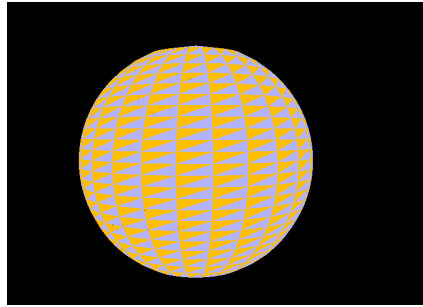
4.0.10 Interpretação da esfera no *Engine*



Representação picotada.



Representação por linhas.



Representação completa.

5 Parser XML

De forma a proceder ao *parsing* do ficheiro XML foi utilizada a biblioteca *tinysql2* que, por ser uma biblioteca bastante completa, eficiente e rápida, consideramos ser adequada para o objetivo pretendido. Deste modo, podemos construir o *engine*, aplicação responsável por realizar a leitura de um ficheiro XML, com o intuito de obter ficheiros 3d. Estes contêm os pontos que permitem gerar as diferentes formas.

Assim sendo, é efetuada a leitura deste, guardando todos os pontos, de forma a exibir as figuras. Será ainda possível interagir com estas através de diversos comandos.

5.1 Ficheiros de configuração

Para esta parte do trabalho foi utilizada a API da *tinysql2*, pegando num ficheiro XML, examinando primeiro onde existe o primeiro elemento com a *tag model* e, de seguida, caso exista, procura o *atribut file*.

```
<scene>
  <models>
    <model file="/home/meriam/Desktop/CG/CGFase1/files/plane.3d"/>
  </models>
</scene>
```

Figura 12: ficheiro XML.

5.2 Carregamento dos ficheiros

Para cada ficheiro encontrado no ficheiro de configuração XML, é invocada a função responsável por abrir o ficheiro. Esta invoca uma segunda função que irá lê-lo e carregar os pontos do respetivo modelo para a estrutura.

Com o objetivo de evitar carregamentos repetidos, os vértices de cada modelo são carregados para uma estrutura que é depois adicionada à estrutura que será representada no final.

6 Demonstração

Para demonstrar as aplicações previamente referidas, iremos demonstrar de que forma somos capazes de executar cada uma delas:

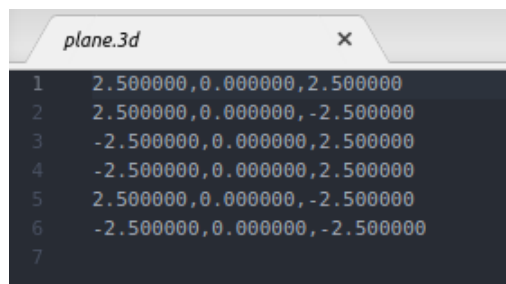
6.1 Generator

```
meriam@meriam-E205SA:~/Desktop/CG/CGFase1/Engine$ cd
meriam@meriam-E205SA:~$ cd Desktop/CG/CGFase1/Generator
meriam@meriam-E205SA:~/Desktop/CG/CGFase1/Generator$ g++ generator.cpp -o generator -lGL -lGLU -lglut
meriam@meriam-E205SA:~/Desktop/CG/CGFase1/Generator$ ./generator sphere 1 25 25 sphere.3d
sphere.3d file was created!
meriam@meriam-E205SA:~/Desktop/CG/CGFase1/Generator$
```

Figura 13: Input/Output Generator.

O projeto inclui uma diretoria "files" com os ficheiros XML, e onde serão criados todos os ficheiros 3d que posteriormente serão lidos através do *Engine*.

Estes tipo de ficheiros têm a seguinte estrutura:



```
plane.3d
1 2.500000,0.000000,2.500000
2 2.500000,0.000000,-2.500000
3 -2.500000,0.000000,2.500000
4 -2.500000,0.000000,-2.500000
5 2.500000,0.000000,-2.500000
6 -2.500000,0.000000,-2.500000
7
```

Figura 14: Exemplo de ficheiro .3d criado.

As linhas correspondem aos pontos dos triângulos. As coordenadas de cada ponto são separadas por virgulas.

6.2 Engine

Os ficheiros XML poderão ser passados como parâmetros. Assim, serão lidos de modo a se obter os ficheiros .3d que contêm os pontos das figuras a serem desenhadas.


```
meriam@meriam-E205SA:~$ cd Desktop/CG/CGFase1/Engine
meriam@meriam-E205SA:~/Desktop/CG/CGFase1/Engine$ g++ engine.cpp tinyxml2.cpp -o engine -lGL -lGLU -lglut
meriam@meriam-E205SA:~/Desktop/CG/CGFase1/Engine$ ./engine sphere.xml
meriam@meriam-E205SA:~/Desktop/CG/CGFase1/Engine$
```

Figura 15: Input/Output Engine.

6.3 Guideline

Executando o comando `./engine -g`, podemos obter o *guideline* apresentado na figura seguinte. Este permite conhecer os comandos de interação com o sistema, tornando o processo mais fácil.

```

|| GUIDE ||

GUIDELINE: ./engine <XML FILE>

FILE:
Specify the path to a XML file in which the information about the
models you wish to create are specified.

MOVEMENTS:

- RIGHT_ARROW_KEY :
  Rotate the camera to right

- LEFT_ARROW_KEY :
  Rotate the camera to left

- DOWN_ARROW_KEY :
  Rotate the camera up ( Y Axe - Positive Direction )

- UP_ARROW_KEY :
  Rotate the camera down ( Y Axe - Negative Direction )

ZOOM:

- i :
  Zoom in

- o :
  Zoom out

FORMAT:

- l :
  Figure format into lines

- p :
  Figure format into points

- f :
  Fill up the figure

|| END ||

```

Figura 16: Guideline.

7 Engine

7.1 Desenho dos pontos das figuras

O *Engine* é responsável pelo carregamento de todas as informações trazidas pelos processos anteriormente explicados. Tem a finalidade de executar as nossas funções de desenho aplicadas aos inputs recebidos. Uma das principais funções é o desenhar os pontos gerados e enviados para o *Engine* como se pode observar no código a baixo.

O método escolhido na realização do trabalho foi a utilização de duas cores diferentes que vão alternando a cada três pontos desenhados.

```

void drawAndColor(void) {
    glBegin(GL_TRIANGLES);
    int i=0;
    bool cor = true;

    for (const Point pt : points) {
        if( i==3 ) {
            cor = !cor;
            i=0;
        }

        if(cor) {
            glColor3f(1, 0.75, 0);
            glVertex3f(pt.x, pt.y, pt.z);
        } else {
            glColor3f(0.7, 0.7, 1);
            glVertex3f(pt.x, pt.y, pt.z);
        }
        i++;
    }
    glEnd();
}

```

Figura 17: Desenho dos pontos recebidos e interpretados.

7.2 Posição da Câmara

Nesta fase foi implementada uma câmara capaz de se mover em torno dos modelos segundo um raio variável.

A câmara é capaz de se deslocar para cima e para baixo, para a esquerda e para a direita e aproximar-se e afastar-se, assemelhando-se ao movimento limitado pela superfície de uma esfera.

Os movimentos da câmara são implementados recorrendo ao uso das seguintes variáveis 'radius', 'alpha' e 'beta'.

```

gluLookAt(radius*cos(beta)*sin(alpha), radius*sin(beta), radius*cos(beta)*cos(alpha),
          0.0, 0.0, 0.0,
          0.0f, 1.0f, 0.0f);

```

Figura 18: Função da câmara.

7.3 Inputs do Teclado

Para uma maior eficiência, constatamos que através das setas direcionais do teclado poderíamos fazer rotações no sentidos das mesmas, sendo modificados os valores das variáveis 'alpha' e 'beta', permitindo alterar a posição da câmara.

```

void specialKey (int k, int i, int j)
{
    (void)i; (void)j;
    switch (k)
    {
        case GLUT_KEY_UP:
            if (beta < (M_PI / 2 - step))
                beta += step;
            glutPostRedisplay();
            break;

        case GLUT_KEY_DOWN:
            if (beta > -(M_PI / 2 - step))
                beta -= step;
            glutPostRedisplay();
            break;

        case GLUT_KEY_LEFT:
            alpha -= step;
            glutPostRedisplay();
            break;

        case GLUT_KEY_RIGHT:
            alpha += step;
            glutPostRedisplay();
            break;

        default:
            break;
    }
}

```

Figura 19: Variação positiva do ângulo beta.

Para fazer zoom in e zoom out recorreremos às teclas correspondentes às letras 'i' e 'o', respetivamente. Alterado-se o valor dos 'radius'.

```

//zoom
case 'o':
    radius += step;
    glutPostRedisplay();
    break;
case 'i':
    radius -= step;
    glutPostRedisplay();
    break;

```

Figura 20: Variação positiva do ângulo beta.

Existe ainda a possibilidade de alterar a forma como visualizamos as figuras: para ver a figura preenchida com duas cores utilizamos a tecla da letra 'f', para ver apenas as arestas representadas recorreremos à letra 'l' e ainda para ver apenas a figura por pontos a letra 'p'.

```

//draw mode
//points
case 'p':
    line = GL_POINT;
    glutPostRedisplay();
    break;

//lines
case 'l':
    line = GL_LINE;
    glutPostRedisplay();
    break;

//full
case 'f':
    line = GL_FILL;
    glutPostRedisplay();
    break;
default:
    break;

```

Figura 21: Variação do formato das figuras.

Além das telas previamente definidas, decidimos implementar uma tecla que nos permita voltar à posição inicial.

```

void keyboard (unsigned char k, int i, int j){
    switch (k){
        case 'r':
            px = 0.0f;
            py = 0.0f;
            pz = 0.0f;
            ry = 0.0f;
            alpha = 0.7f;
            beta = 0.3f;
            step = 0.08f;
            scale = 1.0f;
            glutPostRedisplay();
            break;
        //alteração da rotação do objeto em si
    }
}

```

Figura 22: Tecla r.

8 Conclusão

O trabalho realizado nesta primeira fase foi bastante importante na inicialização desta UC, permitindo-nos acumular experiências na utilização e na especialização de ferramentas e APIs utilizadas em Computação Gráfica, como OpenGL e Glut.

Conseguimos criar um programa que gera as primitivas pretendidas, um *Engine* que processa e desenha os diferentes modelos, interpretando os ficheiros de configuração escritos em *XML*, e também implementar o movimento da câmara usando o input do teclado.

Para além do referido, esta fase permitiu-nos ter uma melhor noção de vértices, tendo sempre em consideração a *regra da mão direita*.

Por fim, consideramos ter cumprido com os objetivos estabelecidos para esta fase e esperamos que o resultado obtido seja a base para a elaboração das seguintes etapas do projeto.