UNIVERSIDADE DO MINHO
ESCOLA DE ENGENHARIA

Computação Gráfica

TRABALHO PRÁTICO - FASE I

Adriana Henriques Esteves Teixeira Meireles A82582
Ana Marta Santos Ribeiro A82474
Carla Isabel Novais da Cruz A80564
Jéssica Andreia Fernandes Lemos A82061

MIEI - 3º Ano - 2º Semestre

Braga, 8 de março de 2019

Índice

1 Introdução

No âmbito da Unidade Curricular de Computação Gráfica, numa fase inicial foi-nos proposto o desenvolvimento de um motor 3D com o intuito de formar diferentes primitivas, nomeadamente um plano, uma caixa, uma esfera e um cone. Para além disso, optamos por construir um cilindro.

Esta fase é composta por duas aplicações: uma que irá gerar os ficheiros com as informações dos modelos 3D (guardando a informação relativa aos vértices) e outra que serve como motor gráfico para a leitura de ficheiros *XML*, apresentando os modelos 3D solicitados.

Para atingir os objetivos propostos, utilizámos ferramentas tais como o *OpenGL* e *C++*.

2 Organização do código

Após a análise do problema em questão verificamos que a melhor solução seria a implementação de duas aplicações, nomeadamente o engine e o generator.

2.1 Generator

Este será responsável por gerar os pontos dos triângulos que permitem construir as diversas figuras. É importante realçar que são passados como parâmetros a figura a desenhar, as suas características, bem como o nome do ficheiro onde os pontos criados deverão ser armazenados. Com o intuito de facilitar a sua construção foi elaborada uma classe *figures*.

2.1.1 Figures

Nesta classe encontram-se os diferentes algoritmos para gerar os pontos que permitem obter os figuras pretendidas, através de triângulos. Assim, as primitivas que se pretendem implementar são:

- plane (size)
- box (length, width, height, divisions)
- sphere (radius, slices, layers)

- cone (radius, height, slices, layers)
- cylinder (radius, height, slices, layers)

Tendo em conta que cada ponto é constituído por três coordenadas, tornou-se imperativo criar uma estrutura para representá-los, tal como é ilustrado de seguida:

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

Para simplificar a construção do plano e da caixa, optamos por recorrer a quadrantes. Assim, definimos quatro quadrantes no eixo positivo dos y e quatro no eixo negativo como podemos observar na figura seguinte:

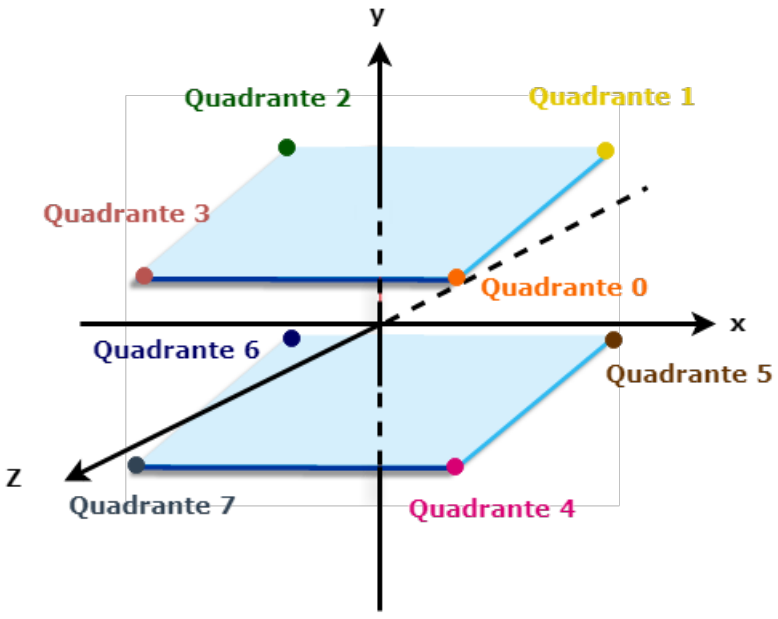

Figura 1. Quadrantes

Uma vez que cada quadrante representa um ponto, decidimos armazenar numa matriz todos estes, como podemos observar de seguida. Desta forma, sabendo o quadrante a que um ponto deverá pertencer conseguimos obter as suas coordenadas através das

dimensões fornecidas.

```
float quadrants[8][3] = {
        { 1,  1,  1}, //Q0
        { 1,  1, -1}, //Q1
        {-1,  1, -1}, //Q2
        {-1,  1,  1}, //Q3
        { 1, -1,  1}, //Q4
        { 1, -1, -1}, //Q5
        {-1, -1, -1}, //Q6
        {-1, -1,  1}, //Q7
};
```

2.2 Engine

Aplicação que 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 formas. Assim, é efetuada uma leitura deste para um vetor de pontos que permitirá exibir as figuras. Será ainda possível interagir com estas através de diversos comandos.

2.2.1 Tinyxml2

De forma a efetuar o parsing dos ficheiros *XML* recorremos à API do *tinyxml2*. Para tal, tivemos em consideração que o ficheiro passado como argumento terá a seguinte estrutura:

```
<scene>
  <models>
     <model file="../../files/exemplo.3d"/>
  </models>
</scene>
```

3 Primitivas Geométricas

3.1 Plane

O plano é representado por dois triângulos, geometricamente semelhantes, mas em posições diferentes.

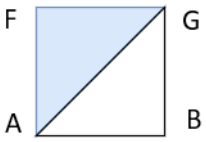

Figura 2. Plano ABFG

Tendo em consideração a regra da mão direita, definimos a ordem dos pontos com que queremos desenhar os triângulos. Deste modo, observamos que o triângulo branco é constituído pelos pontos B, G e A que pertencem aos quadrantes q0, q1 e q3. Semelhante a este, o triângulo azul é definido pelos pontos A, G e F que pertencem aos quadrantes q3, q1 e q2.

Isto origina a junção dos 2 triângulos como é possível ver em baixo.

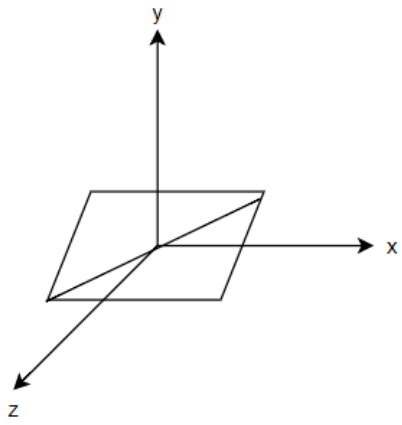

Figura 3. Plane

Para obter os pontos do plano foi usado o seguinte algoritmo:

- Com base nos quadrantes, indicar a sequência pela qual serão alcançados os pontos . Cada algarismo em baixo representa o quadrante.

```
1 int face[6] = {0, 1, 3, 3, 1, 2};
```

- Para se construir dois triângulos são necessários seis pontos (três para cada triângulo). Contudo, o número de pontos a serem feitos são quatro pelo que dois desses pontos são comuns;
- Em cada ciclo é guardado numa variável (no nosso caso j) o ponto gerado no próprio quadrante;
- Com a ajuda dos quadrantes, conseguimos calcular a coordenada de cada ponto. A componente Y mantém-se a zero em todos os pontos, pois a figura encontra-se no plano xOz.

```
for (int i = 0, j; i < 6; i++) {
        j = face[i];
        pt.x = size * quadrants[j][0];
        pt.y = 0;
        pt.z = size * quadrants[j][2];
}
```

3.2 Box

A Caixa é uma figura geométrica composta por 6 faces. Para a sua construção são nos fornecidas as suas dimensões (comprimento, altura e largura) e ainda o número de divisões pretendidas.

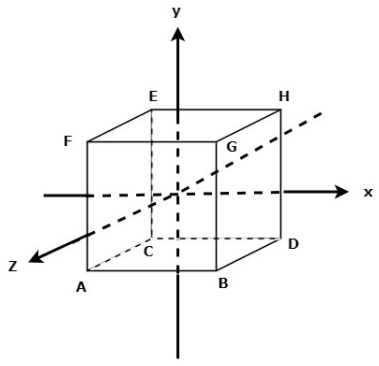

Figura 4. Box

Cada uma das suas faces representará um plano constituído por vários triângulos. O número de triângulos que compõem cada um dos planos, deverá ser determinado pelo número de divisões. Assim, optamos por começar pelo caso mais simples, em que cada plano é composto por dois triângulos, tal como foi explicado na secção anterior. Na eventualidade de serem necessários mais triângulos, estes serão gerados a partir dos mesmos através de um algoritmo que será detalhado de seguida. Para ser mais fácil a observação desta divisão será apresentada como exemplo a base ABFG da figura.

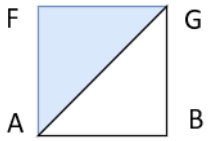

Figura 5. Exemplo do plano ABFG

É importante realçar, que de modo a criar todas as faces, foram geradas uma sequência de 36 pontos sendo 6 por face, dado que cada uma inicialmente é composta por dois triângulos. Por exemplo a face ABFG, é formada por dois triângulos, o GFA e ABG. Estes pontos indicam o quadrante onde o mesmo se situa para que se possa obter as coordenadas de cada um.

3.2.1 Divisões

Nesta subsecção iremos descrever detalhadamente todo o processo de divisão de uma caixa. Para tal, tomaremos como base os dois triângulos representados na figura 5 , que representam um plano. O número de divisões admitido para este exemplo será 2. É importante referir que se considerarmos o número de divisões requerido N, podemos concluir que o número de triângulos necessários para cada base é 2*N*N.

1. Através de cada um dos triângulos principais será gerado um triângulo modelo para ser replicado. A altura do novo triângulo será obtida dividindo-a pelo número de divisões necessárias. O mesmo acontecerá com o comprimento. Assim, neste caso, tanto a altura como o comprimento serão 2.

2. Uma vez construído, este deverá ser replicado na vertical e na horizontal do plano tantas vezes quantas as divisões pretendidas. Para tal, serão utilizados dois ciclos aninhados.

```
for (int i = 0; i < divisions; i++) {
    newP1 = p1, newP2 = p2, newP3 = p3;

    for (int j = 0; j < divisions; j++) {
        //replica triangulo na vertical
    }
    //replica na horizontal
}
```

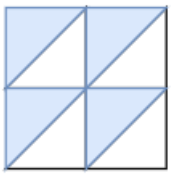

Figura 6. Exemplo da divisão de um plano em 2

3.3 Sphere

A esfera será obtida através da utilização de coordenadas polares. Assim é possível definir o ponto A representado no espaço, sendo o raio a distância deste ponto à origem e em função do α e β apresentados.

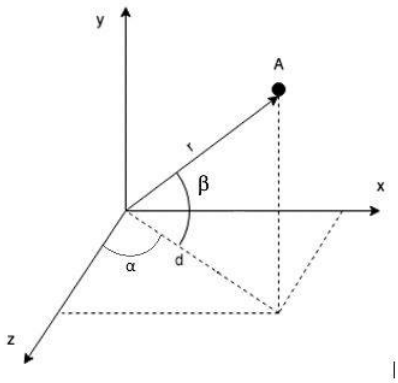

Figura 7. Coordenadas Polares

Assim, através do Teorema de Pitágoras, é possível definir o ponto A com as seguintes coordenadas, A = (d * sin α, r * sen β, d * cos α). Sendo d = r * cos β , ficamos com:

$$A = (r * \cos \beta * \sin \alpha, r * \text{sen } \beta, r * \cos \beta * \cos \alpha)$$

Considerando estes ângulos, para ser possível desenhar a esfera, α estará entre 0 e 2π, pois terá de formar um circunferência completa e tendo em conta a altura da esfera, iremos obter a variação do ângulo β entre -$\pi/2$ e $\pi/2$, indo de uma extremidade à outra. Como input da nossa função irão ser recebidos o número de camadas e fatias. A cada iteração, ter-se-á de calcular quatro novos ângulos para que seja possível a criação da esfera:

$$\alpha = j * 2 * \pi \text{ / slices}$$
$$\alpha_{seguinte} = (j+1) * 2 * \pi \text{ / slices}$$

$$\beta = i * (\pi \text{ / layers}) - \pi/2$$
$$\beta_{seguinte} = (i+1) * (\pi \text{ / layers}) - \pi/2$$

A partir das fórmulas apresentadas, é possível determinar quatro pontos com o auxílio das coordenadas polares previamente calculadas:

$$A = (r * \sin \alpha * \cos \beta, r * \sin \beta, r * \cos \alpha * \cos \beta)$$
$$B = (r * \sin \alpha_{seguinte} * cos\beta, r * \sin \beta, r * \cos \alpha_{seguinte} * cos\beta)$$
$$C = (r * \sin \alpha * \cos \beta_{seguinte}, r * sin\beta_{seguinte}, r * cos\alpha * \cos \beta_{seguinte})$$
$$D = (r * \sin \alpha_{seguinte} * cos\beta_{seguinte}, r * sin\beta_{seguinte}, r * cos\alpha_{seguinte} * cos\beta_{seguinte})$$

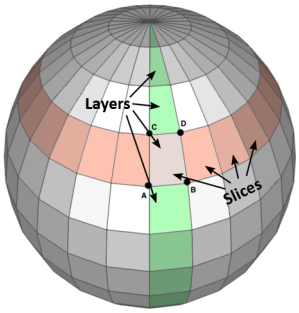

Figura 8. Esfera

Para criar as coordenadas cartesianas através das coordenadas polares, foi criada a função *pointsSphere* recebendo os ângulos e o raio da esfera. Assim, através destas coordenadas e do número de fatias e camadas, é possível construir a esfera por completo. Com isto, são criados dois ciclos:

```
for (int i = 0; i < layers; i++){
        beta = i * (M_PI / layers) - M_PI_2;
        nextBeta = (i + 1) * (M_PI / layers) - M_PI_2;

        for (int j = 0; j < slices; j++){
            alpha = j * 2 * M_PI / slices;
            nextAlpha = (j + 1) * 2 * M_PI / slices;

            p1 = pointsSphere(radius, nextBeta, alpha);
            p2 = pointsSphere(radius, beta, alpha);
            p3 = pointsSphere(radius, nextBeta, nextAlpha);
            p4 = pointsSphere(radius, beta, nextAlpha);

            //draw points
        }
}
```

A cada iteração destes ciclos, são criados quatro pontos que serão utilizados para definir dois triângulos que definirão o plano [ABCD].

3.4 Cone

O cone será uma forma geométrica obtida tendo em conta o raio da base, a altura, o número de camadas bem como o número de fatias, sendo estes os valores passados como argumentos.

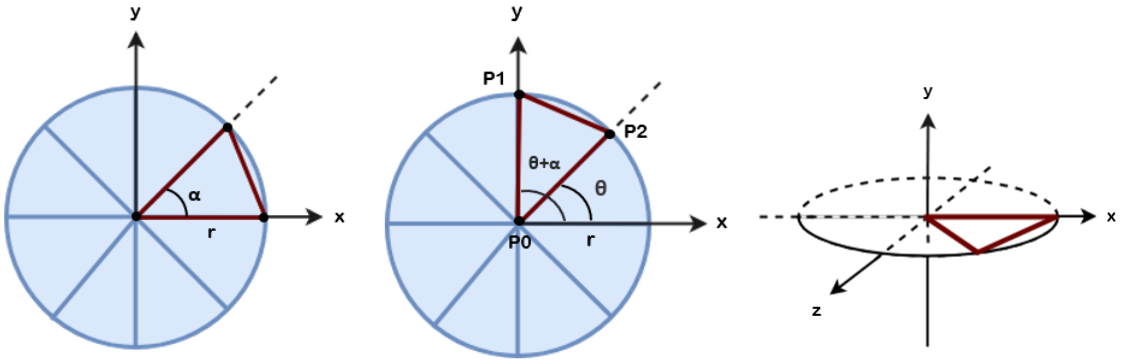

Figura 9. Base do cone

Para a construção deste começamos por criar a base, na qual se tem em consideração

o número de fatias. De modo a dividirmos a base em fatias iguais teremos de repartir 2π pelo número de fatias, obtendo assim o valor de α, que é a amplitude de cada fatia.

$$\alpha = 2\pi \ / \ \text{slices}$$

Através do raio e do ângulo α, conseguimos obter todos os pontos dos triângulos que formam a base do cone.

Como podemos observar na Figura 9 apresentada acima, θ corresponde ao ângulo do ponto atual. Inicialmente o seu valor é 0 e vai sendo acumulado o valor da amplitude de cada fatia, ou seja, obtemos θ através da expressão:

$$\theta = \text{i} * \alpha$$

onde i representa a fatia da base onde nos encontramos. Para obter o ângulo do ponto seguinte somamos ao θ o valor de α, pelo que obtemos a expressão:

$$(\text{i} + 1) * \alpha.$$

Desta forma, o algoritmo apresentado para a construção dos triângulos da base será implementado da seguinte forma:

```
1       alpha = (2 * M_PI) / slices;
2
3       for (int i = 0; i < slices; i++) {
4         teta = i * alpha;
5         tetaNext = (i + 1) * alpha;
6         //draw points
7       }
```

Através das funções cos(x) e sin(x) conseguimos obter as coordenadas de x e z dos pontos dos triângulos, da seguinte forma:

$$\text{x} = \text{raio} * \sin(\text{ângulo})$$
$$\text{z} = \text{raio} * \cos(\text{ângulo})$$

Assim, para o desenho dos triângulos iremos guardar os pontos tendo em consideração a regra da mão direita, ou seja, primeiro a origem seguido do ponto P1 e do P2, como podemos verificar:

```
    //...
    for (int i = 0; i < slices; i++) {
        //...
        p0.x = 0;
        p0.y = 0;
        p0.z = 0;
        p1 = pointsCone(radius, tetaNext, 0);
        p2 = pointsCone(radius, teta, 0);
        //save points
    }
```

Desenhada a base podemos agora passar para a construção do resto do cone. Para tal, seguimos o algoritmo anteriormente elaborado contudo tivemos de ter em conta que os pontos tendem para o centro com o aumento do y. Assim, à medida que o y aumenta o raio irá diminuir até que seja 0.

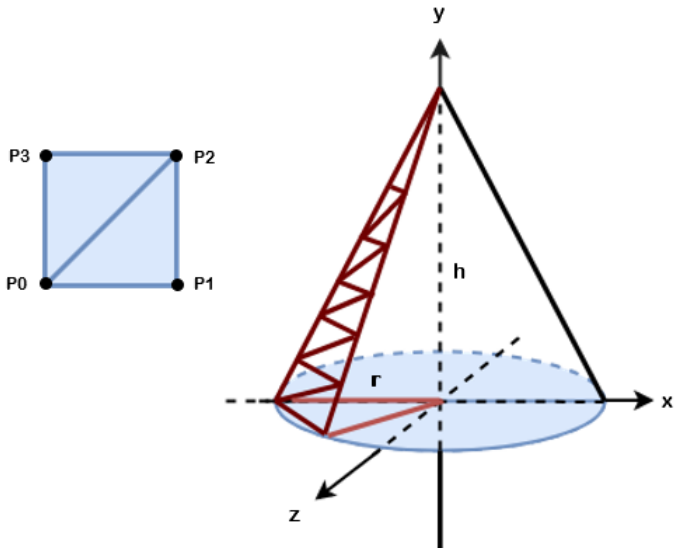

Figura 10. Cone

Como podemos observar nesta figura, o cone é representado por camadas que são divididas em fatias. Através desta divisão verificámos que a figura gerada é semelhante a um quadrado pelo que poderá ser repartido em dois triângulos. Desta forma, para cada camada iremos calcular a altura a que nos encontramos e a altura imediatamente a seguir:

$$\text{altura}_{atual} = i * altura/camadas$$

$$\text{altura}_{seguinte} = altura_{atual} + altura/camadas$$

Para obter os pontos dos triângulos falta ainda calcularmos o raio atual e o raio seguinte:

$$\text{raio}_{atual} = raio - i * raio/camadas$$
$$\text{raio}_{seguinte} = raio - (1+i) * raio/camadas$$

Desta forma, com os valores da altura, raio e ângulo (obtido de forma semelhante indicada anteriormente) é possível calcular as coordenadas dos pontos que constituem a figura referida. Assim, implementamos o algoritmo da seguinte forma:

```
1     scaleHeigth = height / layers;
2     scaleRadius = radius / layers;
3
4       for (int i = 0; i < layers; i++) {
5
6         heigthNow = i * scaleHeigth;
7         radiusNow = radius - i * scaleRadius;
8         radiusNext = radius - (1 + i) * scaleRadius;
9
10        for (int j = 0; j < slices; j++) {
11                teta = j * alpha;
12                tetaNext = (j + 1) * alpha;
13            //draw points
14        }
15     }
```

3.5 Cylinder

Como podemos verificar pela figura 11, a construção do cilindro é bastante semelhante à do cone. Assim, o desenho desta figura geométrica irá ser realizado em três fases, a construção da base, a parte lateral e o topo. Note-se que o desenho da base segue o algoritmo elaborado na secção anterior. A construção da parte lateral difere no sentido em que à medida que o y aumenta o raio irá se manter. De seguida, é desenhado o topo de forma idêntica à base.

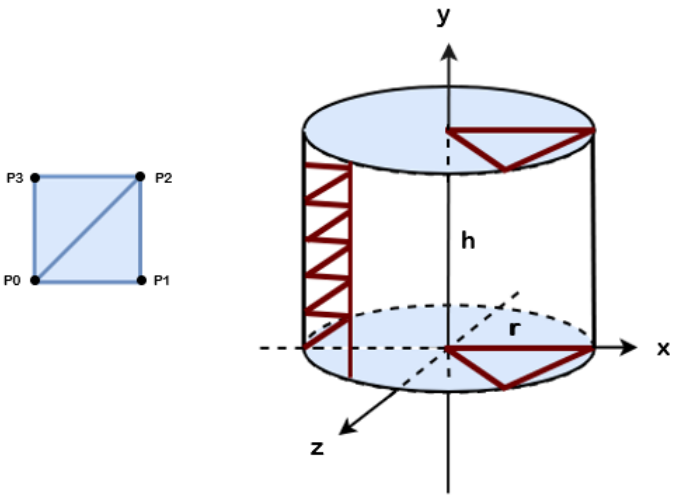

Figura 11. Cilindro

4 Demonstração

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

- **Generator**

```
$ cd generator
$ mkdir build
$ cd build
$ cmake ..
$ make
$ ./generator plane 4 plane.3d
$ ./generator box 4 4 4 20 box.3d
$ ./generator sphere 3 30 30 sphere.3d
$ ./generator cone 2 2 20 20 cone.3d
$ ./generator cylinder 2 2 20 20 cylinder.3d
```

Figura 12. Generator

O projeto inclui uma pasta *files* que contém todos os ficheiros *XML* e onde serão criados todos os .3d que posteriormente serão lidos. Acima são apresentados os vários exemplos.

- **Engine**

Como podemos visualizar na Figura 13, os ficheiros *XML* poderão ser passados como parâmetro. Estes serão lidos de modo a se obter os ficheiros .3d que contêm os pontos das figuras a serem desenhadas.

```
$ cd engine
$ mkdir build
$ cd build
$ cmake ..
$ make
$ ./engine example.xml
```

Figura 13. Engine

4.1 Usabilidade

O menu apresentado na figura seguinte permite conhecer os comandos de interação com o sistema, tornando o processo mais fácil.

```
#_____________________________ HELP _____________________________#
| Usage: ./engine {XML FILE}                                      |
|                 [-h]                                            |
|   FILE:                                                         |
| Specify a path to an XML file in which the information about    |
| the models you wish to create are specified                     |
|                                                                 |
|   ↑ : Rotate your view up                                       |
|                                                                 |
|   ↓ : Rotate your view down                                     |
|                                                                 |
|   ← : Rotate your view to the left                              |
|                                                                 |
|   → : Rotate your view to the right                             |
|                                                                 |
|   F1 : Increase image                                           |
|                                                                 |
|   F2 : Decrease image                                           |
|                                                                 |
|   FORMAT:                                                       |
|   F3: Change the figure format into points                      |
|                                                                 |
|   F4: Change the figure format into lines                       |
|                                                                 |
|   F5: Fill up the figure                                        |
#_________________________________________________________________#
```

Figura 14. Manual de Ajuda

No menu ilustrado, que poderá ser obtido executando ./engine -help ou ./engine -h, constatamos que através das setas podemos fazer rotações no sentido das mesmas. Para tal, são modificados os valores das variáveis alpha e beta, que permitem alterar a posição da câmara.

```
1           //..
```

```
        case GLUT_KEY_UP:
            if (beta < (M_PI / 2 - step))
                beta += step;
            break;

        //case GLUT_KEY_DOWN
        //case GLUT_KEY_LEFT

        case GLUT_KEY_RIGHT:
            alpha += step;
            break;
        //...
```

Existe também a possibilidade de fazer zoom in e out nas figuras, através das teclas F1 e F2, respetivamente. Assim, será alterado o valor do radius.

```
        //...
        case GLUT_KEY_F1:
            radius -= step;
            break;

        case GLUT_KEY_F2:
            radius += step;
            break;
        //...
```

A aplicação permite também alterar o formato das formas. As teclas utilizadas são as F3, F4 e F5, que possibilitam, respetivamente, alterar o desenho para pontos, linhas ou preenchido a cores. É importante referir que a variável line utilizada é passada como argumento na função *glPolygonMode*.

```
        //...
        case GLUT_KEY_F3:
            line = GL_POINT;
            break;
        case GLUT_KEY_F4:
            line = GL_LINE;
            break;
```

```
        case GLUT_KEY_F5:
            line = GL_FILL;
            break;
        //...
```

4.2 Câmara

Tal como na esfera, foi usado o mesmo raciocínio para a posição da Câmara. Assim, a sua localização foi definida através da função *gluLookAt* que recebe as coordenadas da câmara, sítio para onde está a olhar e a sua orientação.

Deste modo, para as coordenadas da câmara, foram usadas as coordenadas polares aplicadas anteriormente, isto é:

$$x = radius*cos(beta)*sin(alfa)$$
$$y = radius*sin(beta)$$
$$z = radius*cos(beta)*cos(alfa)$$

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

4.3 Primitivas

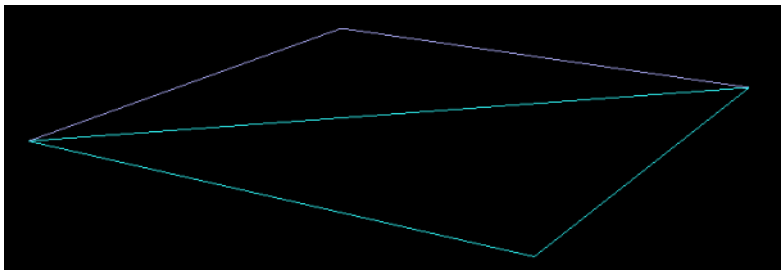

Figura 15. Modelo do Plano apresentado por linhas

Figura 16. Modelo do Plano apresentado por pontos

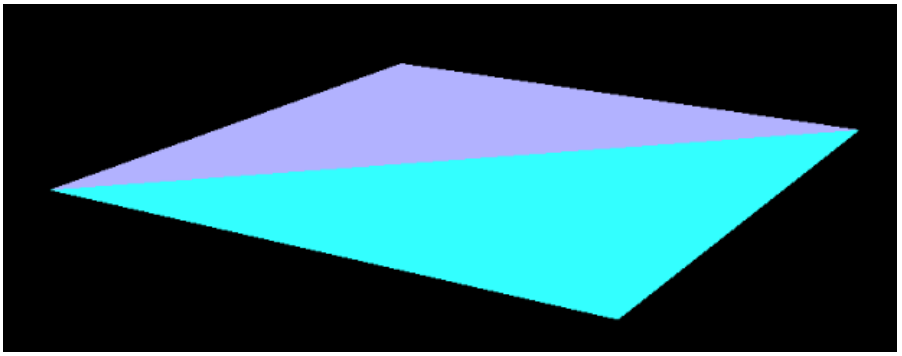

Figura 17. Modelo do Plano preenchido

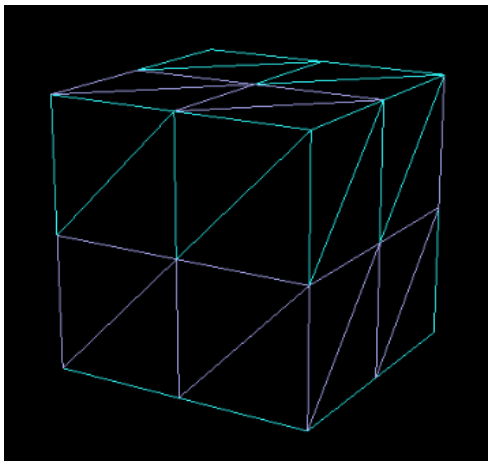

Figura 18. Modelo da Caixa apresentado por linhas

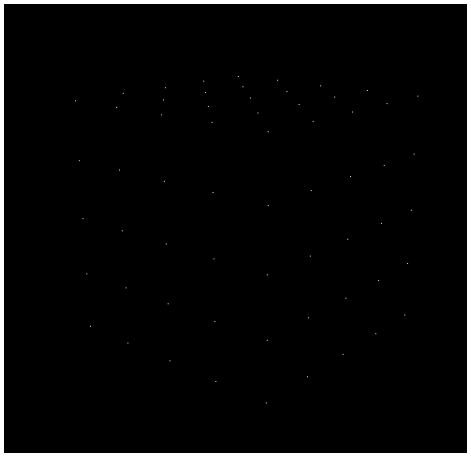

Figura 19. Modelo da Caixa apresentado por pontos

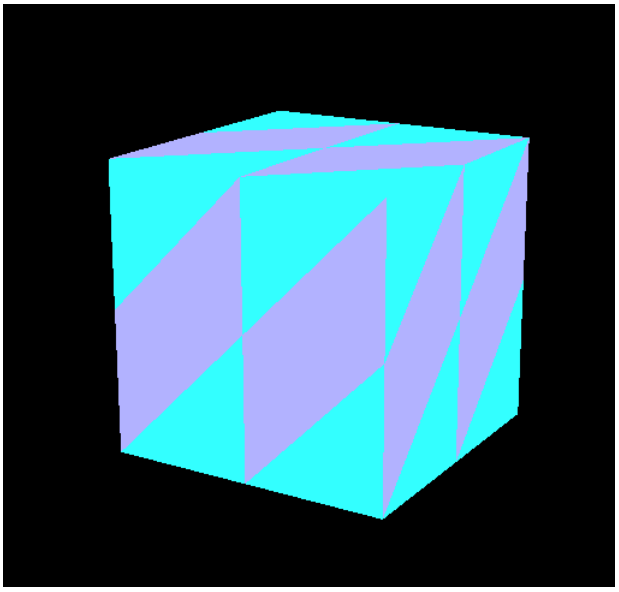

Figura 20. Modelo da Caixa preenchido

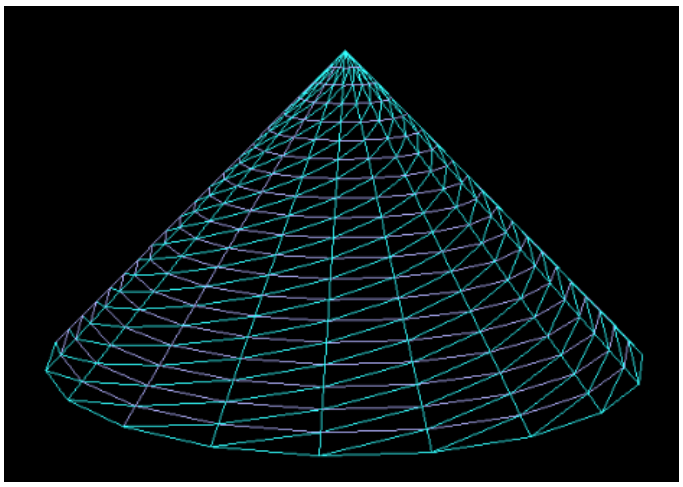

Figura 21. Modelo do Cone apresentado por linhas

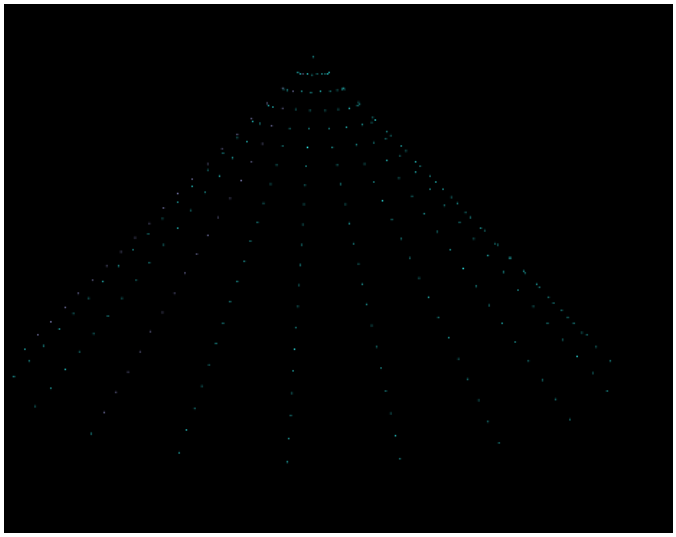

Figura 22. Modelo do Cone apresentado por pontos

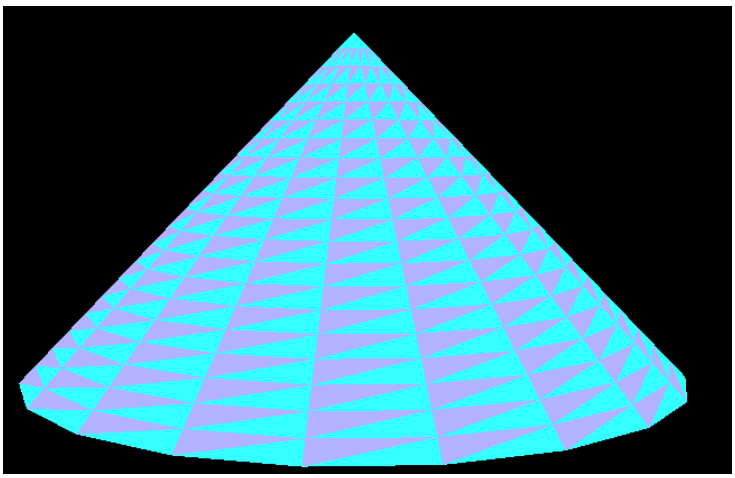

Figura 23. Modelo do Cone preenchido

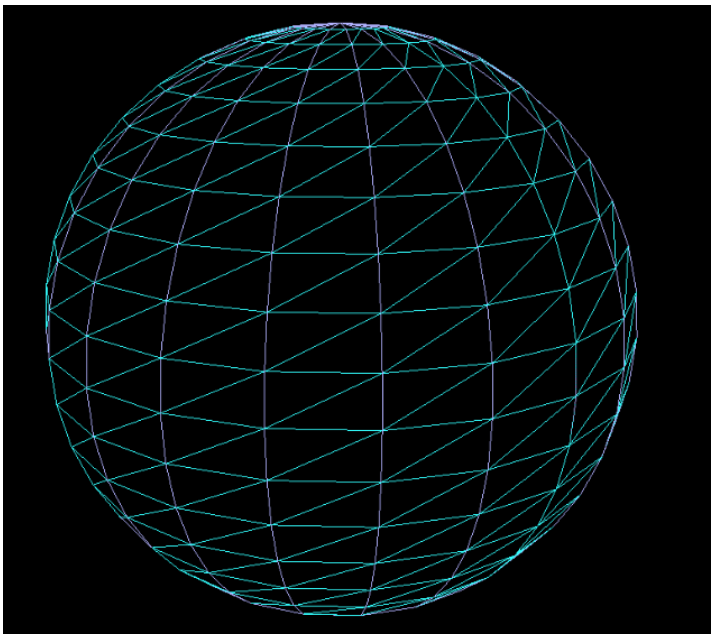

Figura 24. Modelo da Esfera apresentado por linhas

Figura 25. Modelo da Esfera apresentado por pontos

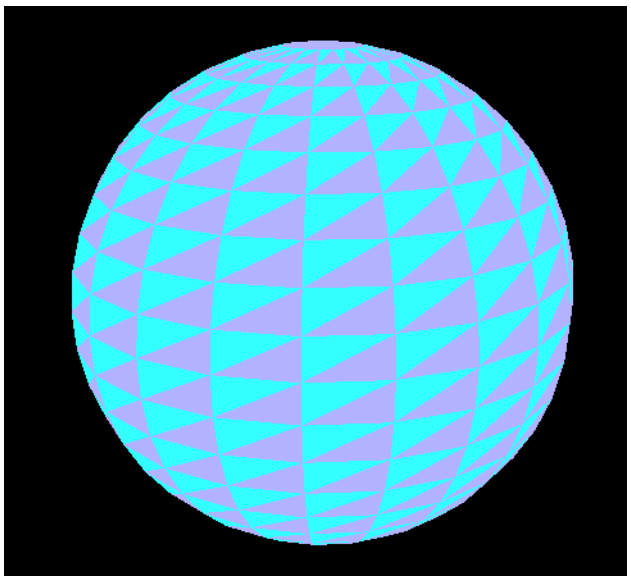

Figura 26. Modelo da Esfera preenchido

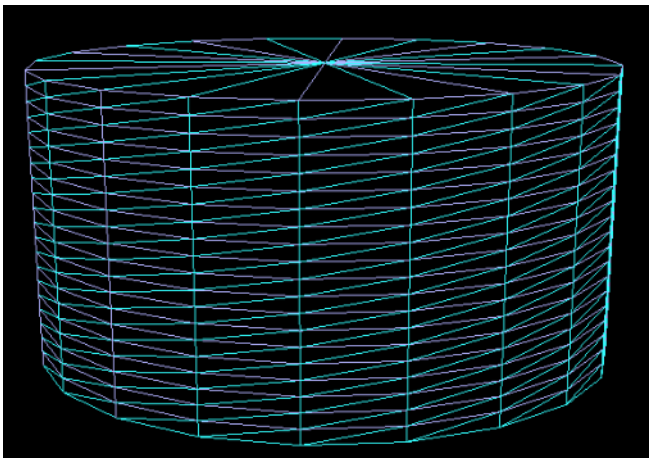

Figura 27. Modelo do Cilindro apresentado por linhas

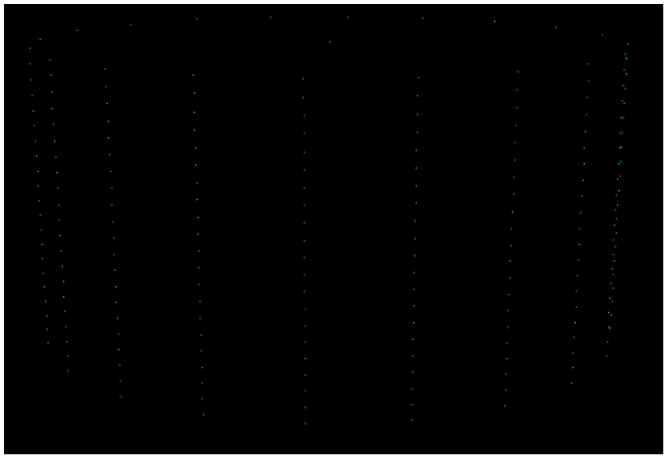

Figura 28. Modelo do Cilindro apresentado por pontos

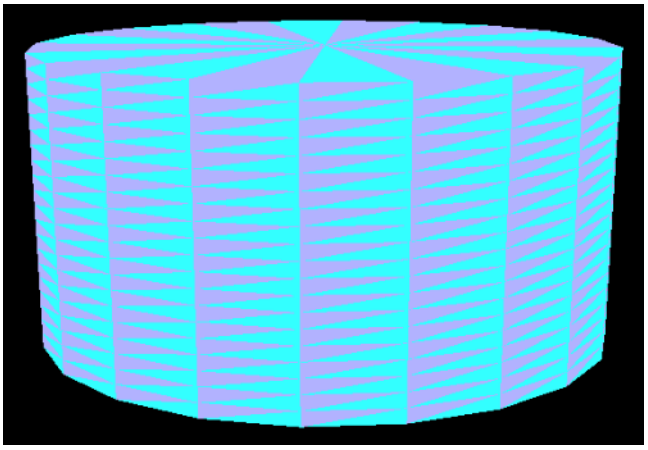

Figura 29. Modelo do Cilindro preenchido

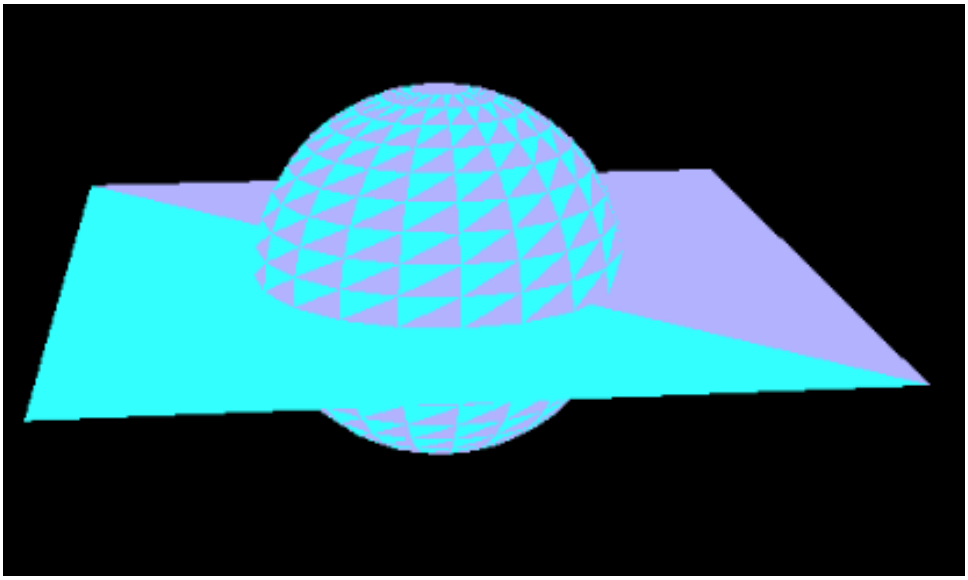

Figura 30. Junção da Esfera com Plano

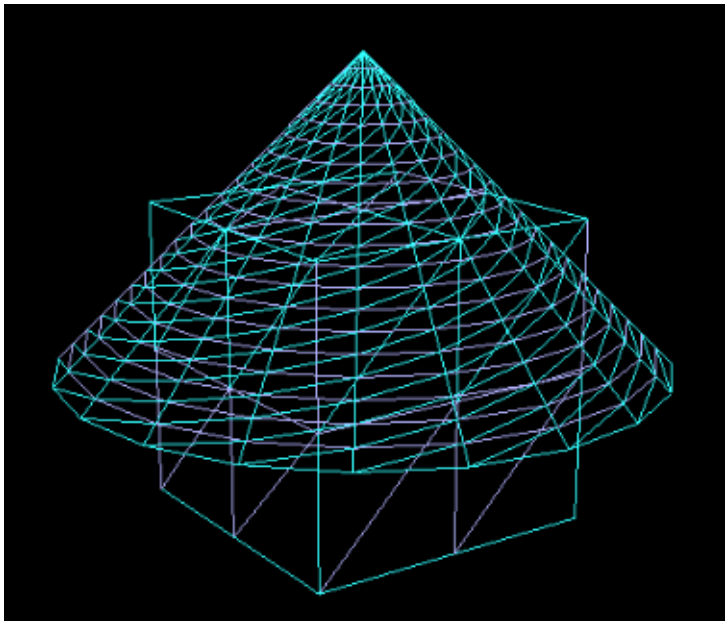

Figura 31. Junção da Caixa com o Cone

Conclusão

A primeira fase deste trabalho permitiu-nos adquirir conhecimentos a nível do tratamento de documentos anotados em *XML* bem como a nível da perceção do espaço 3D para a elaboração das primitivas gráficas. Para além do referido, permitiu-nos ter mais atenção na definição dos vértices (necessário respeitar a regra da mão direita).

Considerámos que foram atingidos os objetivos propostos, isto é, gerar os ficheiros com a informação dos vértices das primitivas e criar um motor que interpretasse o ficheiro de configuração, escrito em *XML*. Também adicionamos a possibilidade de movimentar as figuras sobre si mesmas bem como aproximar ou afastar. Ainda é possível ver apenas os vértices, linhas ou o preenchimento das figuras.

Em suma, esperamos que o resultado obtido nesta fase seja a base para a elaboração das próximas etapas do projeto.