Universidade do Minho

Mestrado Integrado em Engenharia Informática

Computação Gráfica

Grupo 44




Bernardo Mota (A77607)
Gonçalo Duarte (A77508)
Luís Neto (A77763)
João de Macedo (A76268)

Conteúdo

1 Introdução

Este relatório documenta o desenvolvimento da primeira fase do projeto de Computação Gráfica, mostrando a abordagem e as decisões tomadas ao longo desta fase. O objetivo desta fase do projeto é criar um programa que gera pontos 3D baseado em primitivas e um programa que carregue os ficheiros gerados e os desenhe no ecrã.

A primeira parte do relatório detalha a abordagem utilizada para criar o gerador, incluindo as estruturas de dados e os algoritmos utilizadas para gerar os triângulos necessários para representar as primitivas, que são guardadas num ficheiro criado pelo gerador.

A segunda parte do relatório é relativa ao Motor 3D, que lê um ficheiro XML com informação acerca dos modelos e os carrega na memória. Depois de carregar os ficheiros, o motor desenha os modelos no ecrã. São apresentadas as estruturas de dados utilizadas, o processo de carregamento dos dados e do desenho dos modelos.

Finalmente, são apresentadas demonstrações das primitivas e as conclusões para a primeira fase.

2 Gerador

O gerador recebe como argumentos o tipo da primitiva para gerar, os parametros da primitiva e o nome do ficheiro para guardar os pontos 3D.

2.1 Estruturas

Foi utilizada uma struct, Point, cujos parâmetros são três doubles, x, y e z para representar um ponto 3D.

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

2.2 Geração dos Modelos

O programa escreve para um ficheiro de output os pontos 3D que formam os triângulos da primitiva. Os pontos gerados dependem da primitiva passada como argumento ao gerador e aos argumentos das primitivas. A seguir são apresentados os algoritmos utilizados para a geração das diferentes primitivas.

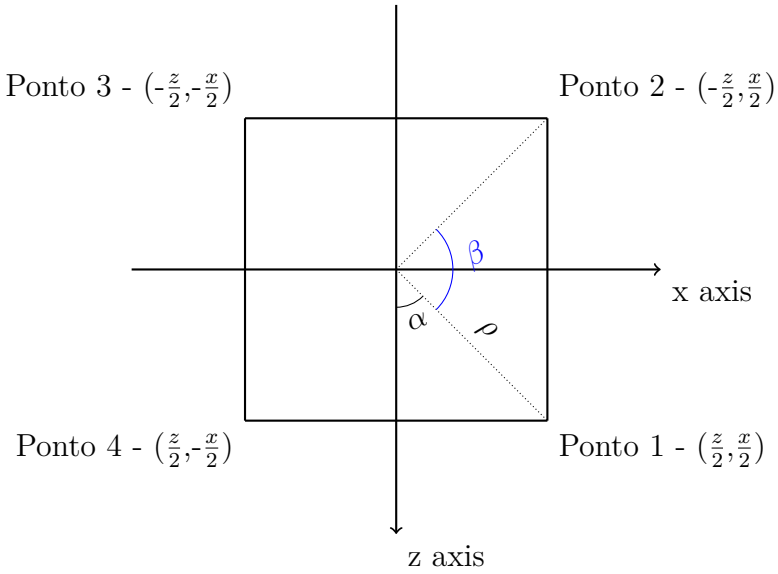
2.2.1 Plane

```
void generatePlane(const string fileName, double size)
```

Para gerar os pontos do plano, centrado na origem, foi utilizado um sistema de coordenadas polares para determinar o valor de cada coordenada:

$$\rho = \frac{size * \sqrt{2}}{2}$$
$$x = \rho * sin(\alpha)$$
$$y = 0$$
$$z = \rho * cos(\alpha)$$

Como o plano no contexto de Computação Grafica é um quadrado, que é descrito por quatro pontos, então:



Observando a figura, num um ponto de vista 2D, é possível ver que o ponto 1 e 3 pertencem á bissectriz dos quadrantes impares e que os pontos 2 e 4 pertencem á bissectriz dos quadrantes pares, que são perpendiculares entre si, logo, $\beta = 90$.

Também é possível observar que a bissectriz dos quadrantes impares forma um angulo de 45 graus com o eixo z, logo, $\alpha = 45°$.

Posto isto, o algoritmo utilizado para gerar os quatro pontos consistiu em calcular

$$\alpha = \frac{(90 * i + 45) * 2\pi}{360}, i \in [0, 4[$$

Após o calculo de α, é feito o calculo de x e z através do sistema de equações referidas anteriormente. Por fim, os pontos são escritos para um ficheiro.

2.2.2 Box

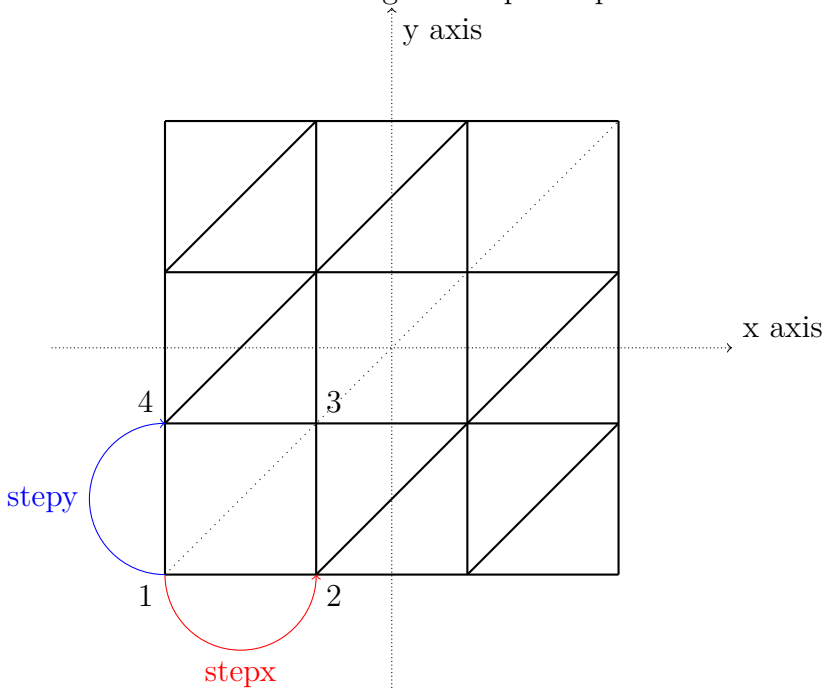
```
void generateBox(const string fileName, double sizex, double sizey,
                 double sizez, int divisions)
```

As 6 faces da primitiva Box foram geradas em pares, pois cada par de faces tem coordenadas identicas (ou seja, a face da frente e de trás diferem apenas no eixo z, que passa a negativo para a face de trás). Para cada par de faces foi criada uma função. Essas função utilizam um algoritmo semelhante diferindo apenas na coordenada que não varia entre cada par.

A função que gera a face da frente e de trás é:

```
void generateBoxFrontBack(double sizex, double sizey, double sizez,
                          double divisions, vector<Point>* points)
```

Para gerar cada divisão da caixa foram gerados quatro pontos:



A variação entre os ponto segundo o eixo do x e y é dada pelas seguintes equações:

$$stepx = \frac{sizex}{divisions}$$

$$stepy = \frac{sizey}{divisions}$$

Assim vem que:

$$Ponto1 = (-\frac{sizex}{2}, \frac{sizey}{2}, \frac{sizez}{2})$$

$$Ponto2 = (-\frac{sizex}{2}, \frac{sizey}{2} - stepy, \frac{sizez}{2})$$

$$Ponto3 = (-\frac{sizex}{2} + stepx, \frac{sizey}{2}, \frac{sizez}{2})$$

$$Ponto4 = (-\frac{sizex}{2} + stepx, \frac{sizey}{2} - stepy, \frac{sizez}{2})$$

Através destes pontos podemos concluir fórmulas para gerar todos os pontos que são necessários para gerar uma face da caixa:

$$Ponto1 = (stepx * i, stepy * j, \frac{z}{2})$$

$$Ponto2 = (stepx * i, stepy * (j-1), \frac{z}{2})$$

$$Ponto3 = (stepx * (i+1), stepy * j, \frac{z}{2})$$

$$Ponto4 = (stepx * (i+1), stepy * (j-1), \frac{z}{2})$$

$$j = \frac{divisions}{2} - n$$

$$i = -\frac{divisions}{2} + n$$

$$j \in \left[-\frac{divisions}{2}, \frac{divisions}{2}\right[$$

$$i \in \left[-\frac{divisions}{2}, \frac{divisions}{2}\right[$$

$$n \in [0, divisions[$$

As fórmulas de geração de pontos das outras faces é análogo às fórmulas acima apresentadas.

2.2.3 Sphere

```
void generateSphere(const string fileName, double radius,
                    double slices, double stacks)
```

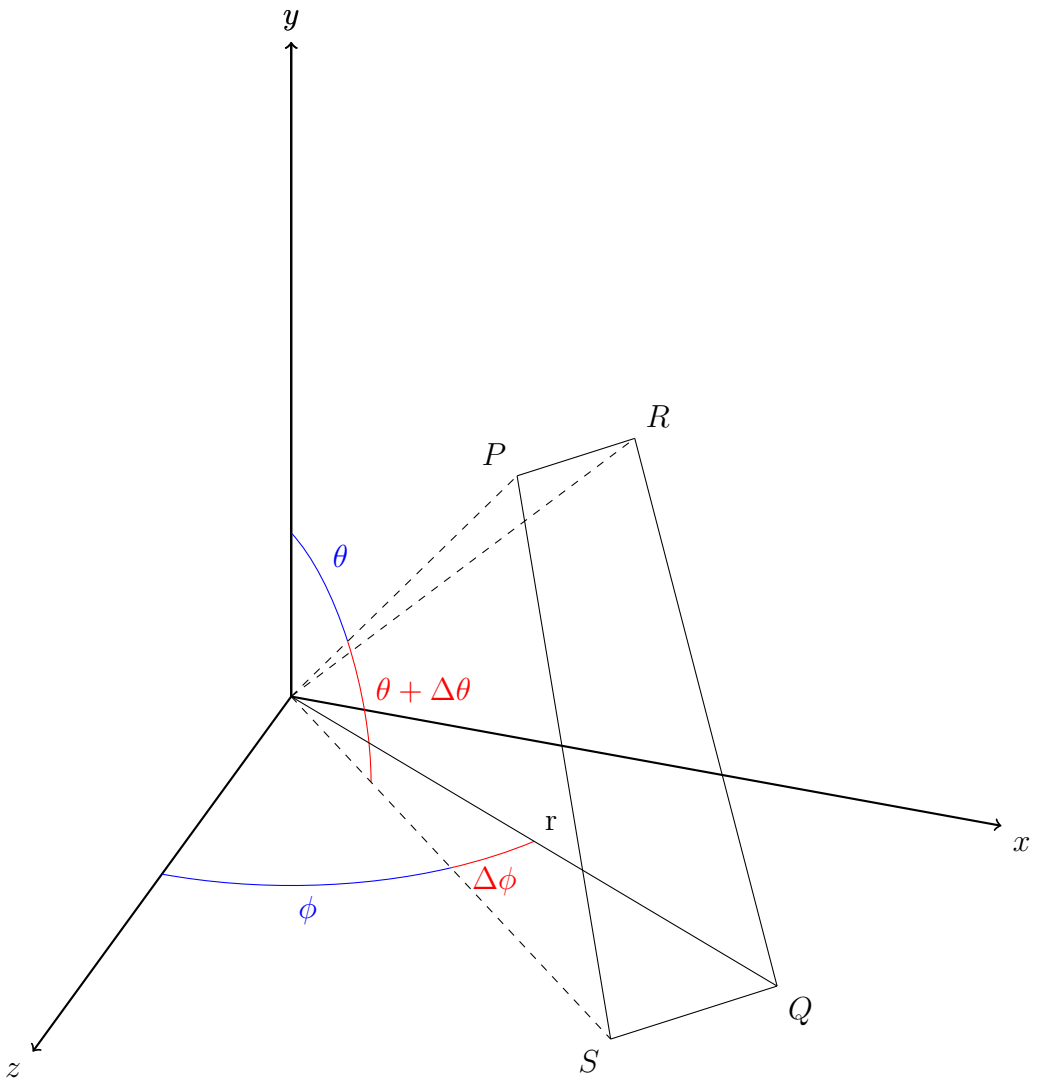
Para gerar os pontos da esfera foi utilizado um sistema de coordenadas esfericas onde as coordenadas x, y, z são dados por:

$$x = r * \sin(\theta) * \sin(\phi)$$

$$y = r * \cos(\theta)$$

$$x = r * \sin(\theta) * \cos(\phi)$$

$$\theta \in [0, \pi[\,, \phi \in [0, 2\pi[$$



Na figura podemos observar os pontos P, R, S e Q que, através das equações referidas anteriormente, têm como coordenadas:

$$P = (x = r * \sin(\theta) * \sin(\phi), y = r * \cos(\theta), x = r * \sin(\theta) * \cos(\phi))$$

$$R = (x = r * \sin(\theta) * \sin(\phi + \Delta\phi), y = r * \cos(\theta), x = r * \sin(\theta) * \cos(\phi + \Delta\phi))$$

$$S = (x = r * \sin(\theta + \Delta\theta) * \sin(\phi), y = r * \cos(\theta + \Delta\theta), x = r * \sin(\theta + \Delta\theta) * \cos(\phi))$$

$$Q = (x = r * \sin(\theta + \Delta\theta) * \sin(\phi + \Delta\phi), y = r * \cos(\theta + \Delta\theta), x = r * \sin(\theta + \Delta\theta) * \cos(\phi + \Delta\phi))$$

É necessario que a esfera tenha *slices* e *stacks*:

$$\Delta\theta = \frac{\pi}{stacks}$$

$$\Delta\phi = \frac{2\pi}{slices}$$

.

Atráves das equações dos pontos da figura foi desenvolvido um algoritmo que desenha todos os pontos da esfera, que consiste em gerar os pontos de cada slice para cada stack, traduzindo-se em :

$$P = (r * \sin(\frac{i * \pi}{stacks}) * \sin(\frac{j * 2\pi}{slices}), r * \cos(\frac{i * \pi}{stacks}), r * \sin(\frac{i * \pi}{stacks}) * \cos(\frac{j * 2\pi}{slices}))$$

$$R = (r * \sin(\frac{i * \pi}{stacks}) * \sin(\frac{j * 2\pi}{slices}), r * \cos(\frac{(i+1) * \pi}{stacks}), r * \sin(\frac{i * \pi}{stacks}) * \cos(\frac{(j+1) * 2\pi}{slices}))$$

$$S = (r * \sin(\frac{(i+1) * \pi}{stacks}) * \sin(\frac{j * 2\pi}{slices}), y = r * \cos(\frac{(i+1) * \pi}{stacks}), * \sin(\frac{(i+1) * \pi}{stacks}) * \cos(\frac{j * 2\pi}{slices}))$$

$$Q = (r * \sin(\frac{(i+1) * \pi}{stacks}) * \sin(\frac{(j+1) * 2\pi}{slices}), r * \cos(\frac{(i+1) * \pi}{stacks}), r * \sin(\frac{(i+1) * \pi}{stacks}) * \cos(\frac{(j+1) * 2\pi}{slices}))$$

$$i \in [0, stacks[\, , j \in [0, slices[$$

De referir que na primeira e última *stack* apenas são desenhados três pontos: SQP e RPS, respetivamente.

2.2.4 Cone

```
void generateCone(const string fileName, double radius,
                  double height, int slices, int stacks)
```

Para gerar os pontos do cone foi considerado que ao dividir o cone em stacks, cada stack tem uma altura (coordenada y do ponto) e um raio diferente, sendo esta relação pode ser expressada por:

$$stack\_relation = \frac{current\_stack}{stacks}$$

$$current\_stack \in [0, stacks]$$

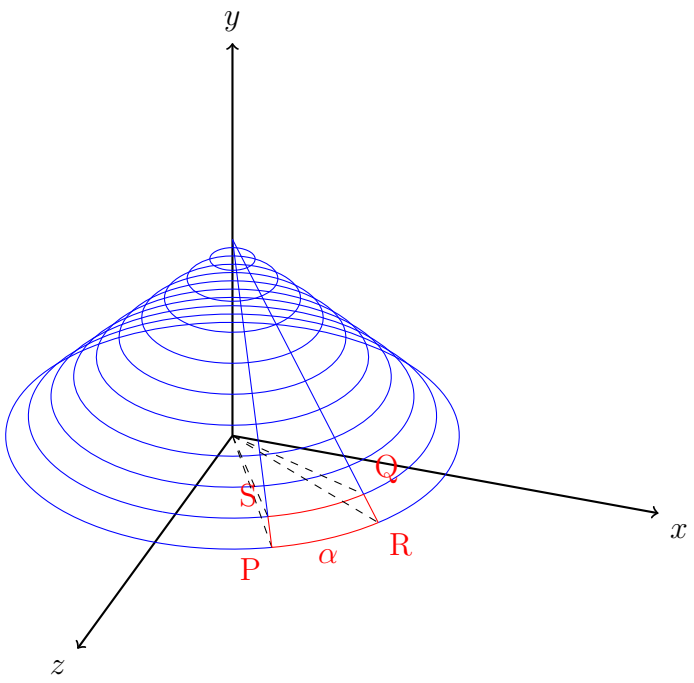
$$raio = radius * (1 + stack\_relation)$$

$$altura = height * stack\_relation$$

As coordenadas x e z de cada ponto são geradas através de um sistema de coordenadas polares:

$$x = raio * \sin(\alpha)$$

$$z = raio * \cos(\alpha)$$



Para desenhar todos os pontos é também necessario saber que α usar, este é dado por:

$$alpha = \frac{current\_slice * 2\pi}{slices}$$

$$current\_slice \in [0, slices[$$

Podemos então dizer que as coordenadas de P, R, S e Q são:

$$P = (raio_1 * \sin(\alpha), altura_1, raio_1 * \cos(\alpha))$$
$$R = (raio_1 * \sin(2\alpha), altura_1, raio_1 * \cos(2\alpha))$$
$$S = (raio_2 * \sin(\alpha), altura_2, raio_2 * \cos(\alpha))$$
$$Q = (raio_2 * \sin(2\alpha), altura_2, raio_2 * \cos(2\alpha))$$

Com a definição de P, R, S e Q podemos inferir um algoritmo para gerar os pontos que consiste em percorrer todas as slices de cada textitstack e calcular os pontos:

$P = (radius * (1 + \frac{current\_stack}{stacks}) * \sin(\frac{current\_slice * 2\pi}{slices}), height * (\frac{current\_stack}{stacks}), radius * (1 + \frac{current\_stack}{stacks}) * \cos(\frac{current\_slice * 2\pi}{slices}))$

$R = (radius * (1 + \frac{current\_stack}{stacks}) * \sin(\frac{(current\_slice+1) * 2\pi}{slices}), (\frac{height * current\_stack}{stacks}), radius * (1 + \frac{current\_stack}{stacks}) * \cos(\frac{(current\_slice+1) * 2\pi}{slices}))$

$S = (radius * (1 + \frac{current\_stack+1}{stacks}) * \sin(\frac{current\_slice * 2\pi}{slices}), (\frac{height * (current\_stack+1)}{stacks}), radius * (1 + \frac{current\_stack+1}{stacks}) * \cos(\frac{current\_slice * 2\pi}{slices}))$

$Q = (radius * (1 + \frac{current\_stack+1}{stacks}) * \sin(\frac{(current\_slice+1) * 2\pi}{slices}), (\frac{height * (current\_stack+1)}{stacks}), radius * (1 + \frac{current\_stack+1}{stacks}) * \cos(\frac{(current\_slice+1) * 2\pi}{slices}))$

De notar que quando são gerados os pontos da primeira stack são também gerados os pontos da base inserindo os pontos ORP no vector de pontos; na ultima stack é apenas inseridos os pontos PRQ.

3 Motor 3D

3.1 Estruturas

- typedef vector<Point> Primitive;
- typedef vector<Primitive> Model;

Foi definido o tipo Primitive como um vetor de Point onde cada Point é um ponto do modelo a desenhar.

Foi definido o tipo Model como um vetor de Primitive que armazena todos os modelos que o motor deve desenhar.

3.2 Ficheiros

3.2.1 Ficheiro Com Pontos Gerados

O ficheiro gerado com o nome passado como argumento ao programa Gerador contem todos os pontos dos triângulos gerados, com as coordenadas de cada ponto separadas por um espaço e com os pontos separados por um newline.

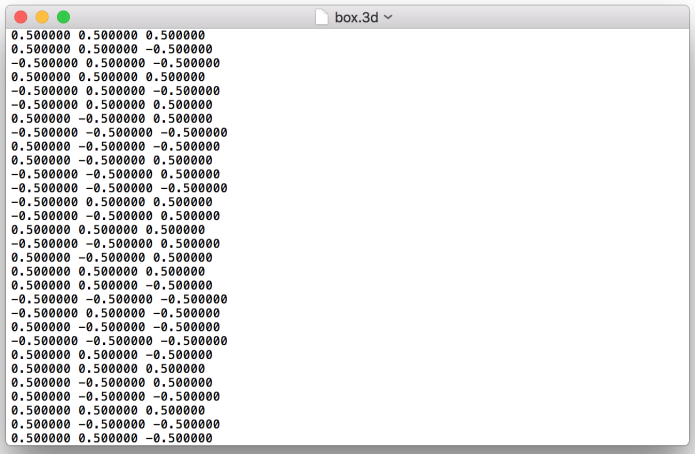


Figura 1: Ficheiro gerado com os argumentos 'box 1 1 1 1 box.3d'

3.2.2 Ficheiro XML

O ficheiro XML é utilizado pelo motor para saber que ficheiros carregar. O ficheiro XML apresentado é utilizado pelo motor para carregar e desenhar os dois modelos.

Listing 1: Ficheiro XML com dois modelos

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

3.3 Carregamento dos Dados

O carregamento dos dados é o primeiro passo para o desenho dos modelos. Os ficheiros com pontos gerados a carregar são indicados no ficheiro XML, cuja localização é passada como argumento ao motor 3D.

```
int loadXML(const char* filename)
```

Através da interface fornecida pela biblioteca TinyXML-2, o ficheiro XML é interpretado, sendo possível extrair o nome dos ficheiros a carregar percorrendo os elementos XML. A tag model contem um atributo file que indica o nome do ficheiro dos pontos do modelo.

Depois de obter o filename do modelo a carregar, a função loadModel carrega os pontos do ficheiro para o tipo Primitive, que é um vector de Point. Cada modelo que é carregado é adicionado a Model, um vector de Primitive.

```
void loadModel(const char* fileName){
    Primitive p;
    ifstream file(fileName);
    if (file) {
        string triangle;
        while (getline(file,triangle)) {
            string sx, sy, sz;
            istringstream st_triangle(triangle);
            st_triangle >> sx >> sy >> sz;
            Point pt = {stof(sx),stof(sy),stof(sz)};
            p.push_back(pt);
        }
    }
    models.push_back(p);
}
```

Após todo o ficheiro XML estar interpretado, todos os pontos de todos os modelos já foram carregados pelo motor 3D.

3.4 Desenho dos Modelos

As seguinte função é utilizada para desenhar todos os modelos que foram carregados do XML.

```
void drawModels(Model models)
```

Esta função percorre todos os modelos desenhando todos os seus pontos.

```
...
    int i = 0;

    for (auto const& primitive: models) {
        glBegin(GL_TRIANGLES);
        for(auto const& pt: primitive){
            if (i < 3) {
                glColor3f(
                ((float)51/255),
                ((float)153/255),
                ((float)51/255));
                i = (i+1);
            }else{
                glColor3f(
                0.0f,
                ((float)200/255),
                0.0f);
                i = (i+1)%6;
            }
            glVertex3f(pt.x, pt.y, pt.z);
        }
        glEnd();
    }
}
```

4 Demonstrações das Primitivas

4.1 Plane

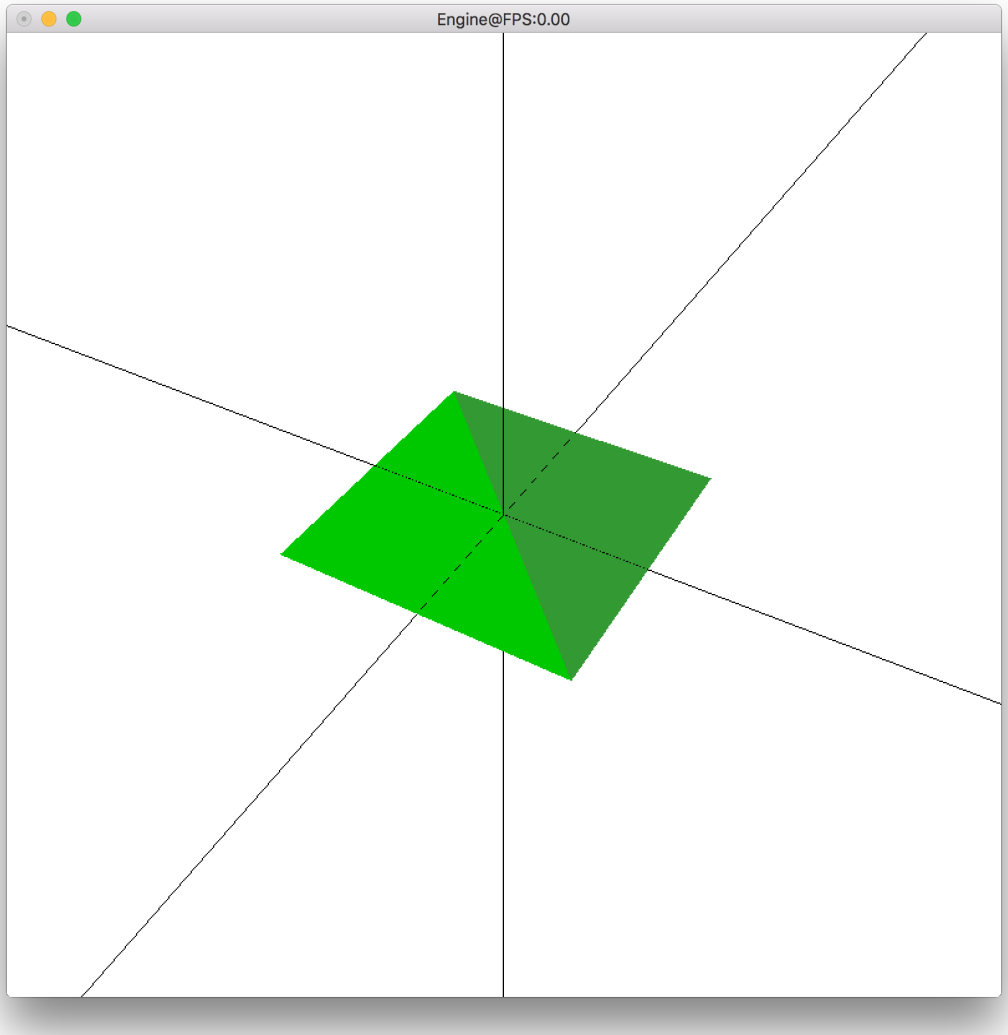


Figura 2: Plane, $size = 1$

4.2 Box

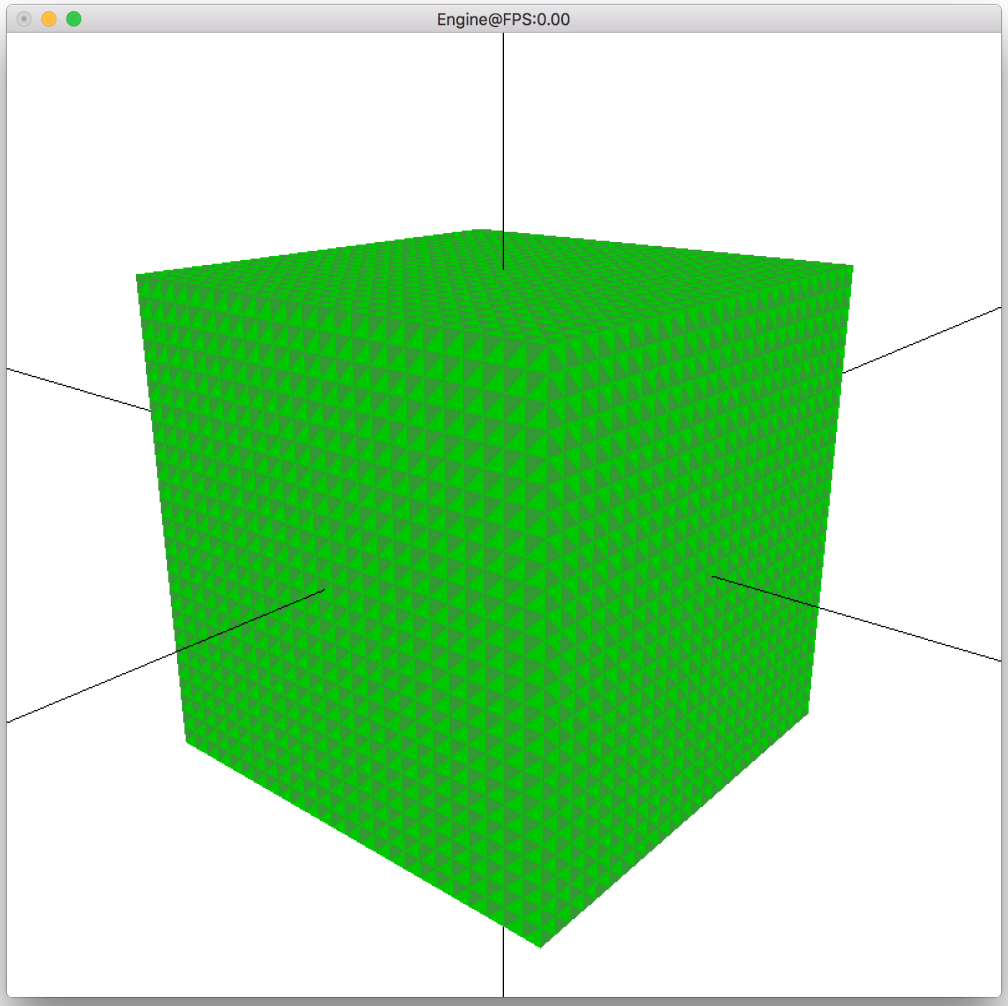


Figura 3: Box, $x = 1, y = 1, z = 1, divisoes = 25$

4.3 Sphere

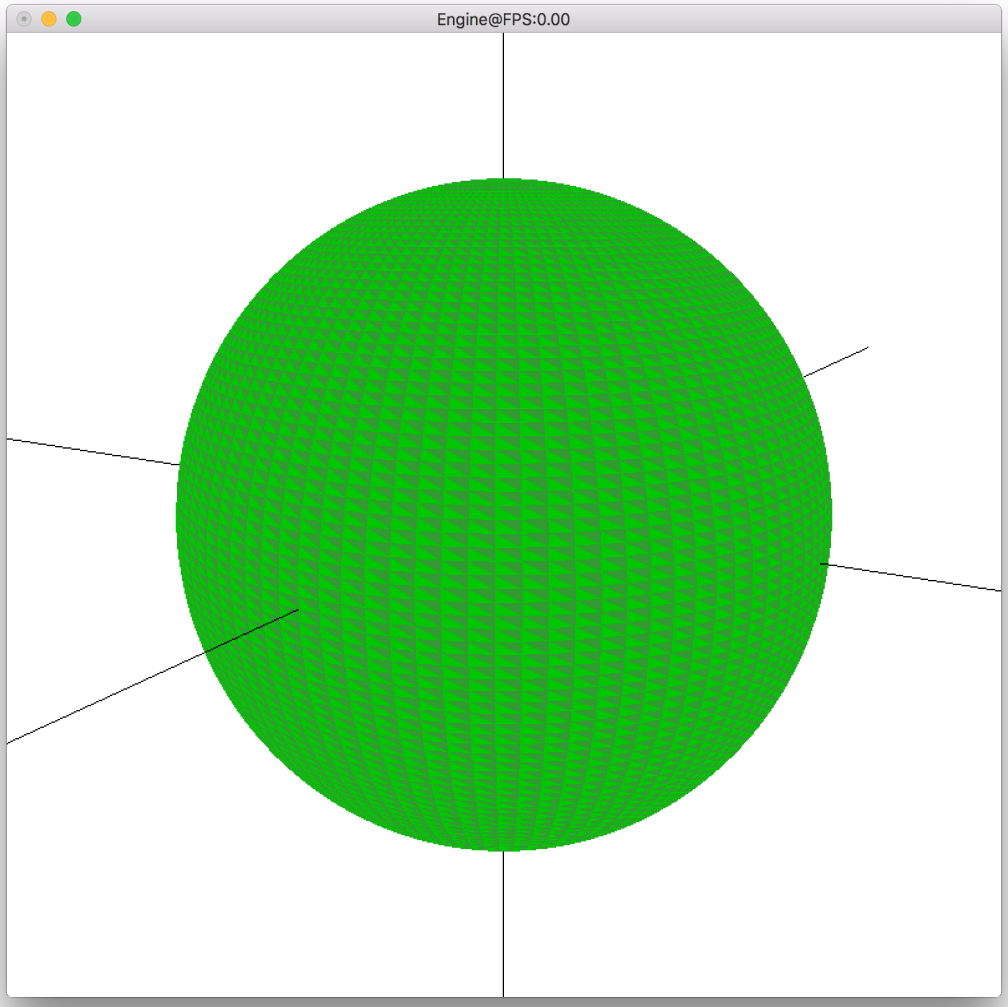


Figura 4: Esfera, $raio = 1, camadas = 100, fatias = 100$

4.4 Cone

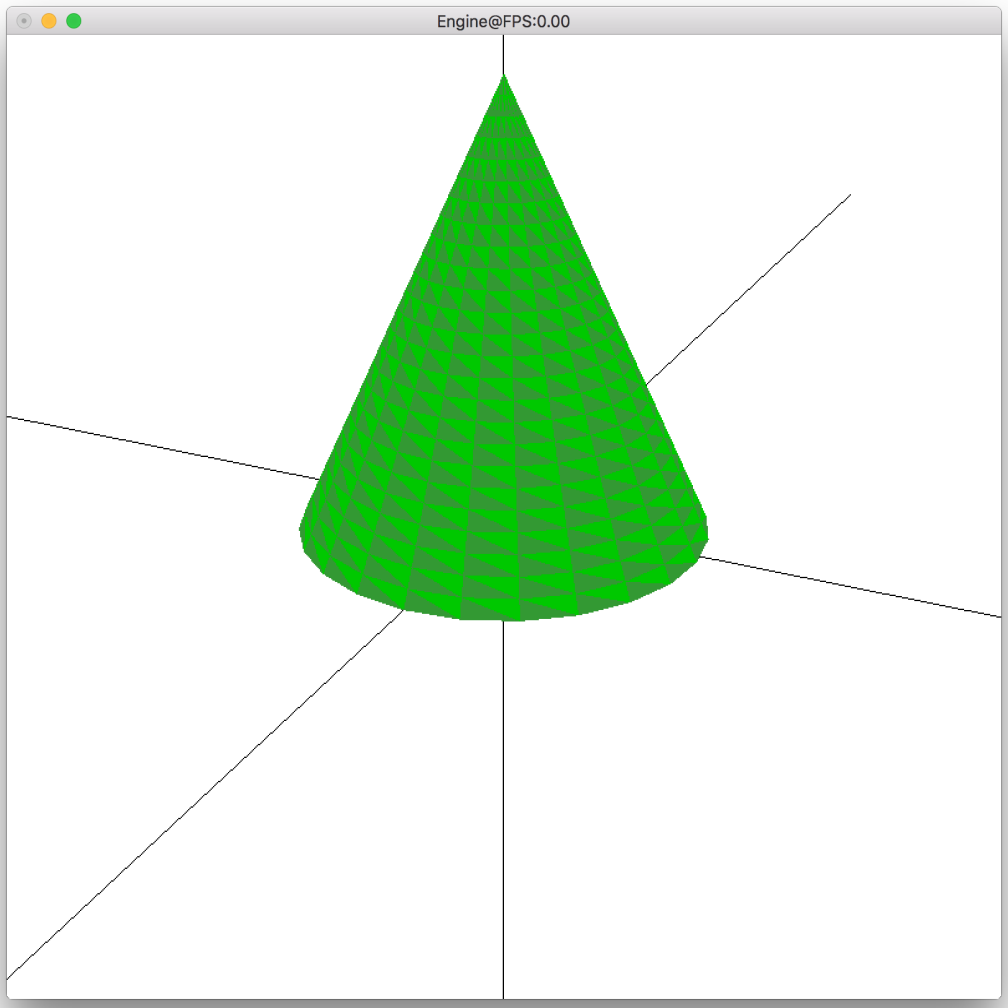


Figura 5: Cone, $raio = 1, altura = 2, camadas = 25, fatias = 25$

5 Conclusões

Foram cumpridos todos os objetivos propostos nesta fase do projeto, implementando a geração dos pontos de todas as primitivas no programa gerador e o carregamento e desenho dos ficheiros gerados no motor 3D. Os ficheiros que são carregados para o motor são indicados no ficheiro XML, que é interpretado pelo motor.

Para além dos objetivos propostos, foi implementado a movimentação da câmara e um contador de FPS, que é indicado no título da janela.