Universidade do Minho
Licenciatura em Ciências da Computação

Unidade Curricular de Computação Gráfica

Ano Lectivo de 2021/2022

Phase 1 – Graphical Primitives

Filipe Azevedo A87969

João Nogueira A87973

Miguel Gonçalves A90416

Rui Baptista A87989

CG

Resumo

Nesta primeira fase iremos aplicar o conhecimento adquirido nas primeiras semanas, nomeadamente utilizando tanto o Generator como o Engine, de forma a ser possível desenhar 4 figuras geométricas, utilizando o GLUT e sendo possível a leitura de um ficheiro XML.

Palavras-Chave: Computação Gráfica, Figuras Geométricas, XML, GLUT, Generator, Engine

Índice

Índice de Figuras

1. Introdução

Nesta fase inicial foi pedido que se realizassem dois sistemas, tendo estes como funções guardar informações relativas à figura que se pretende desenhar e outro onde fosse possível ler, guardar um ficheiro escrito em XML e desenhar o modelo segundo as indicações dadas.

As figuras geométricas requeridas para esta tarefa foram um plano, uma *box*, um cone e uma esfera.

Ao longo do relatório iremos explicar o raciocínio por de trás do cálculo dos pontos para ser possível o desenho das figuras, bem como o restante raciocínio para o funcionamento coeso do programa.

2. Estrutura do Projeto

O nosso projeto está dividido nos seguintes diretórios:

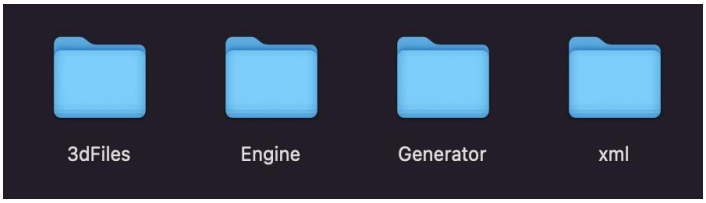

Figura 1 - Estrutura

Iremos explicar cada um deles e as suas respetivas funções, sendo dois deles para guardar ficheiros, 3dFiles e xml, e outros dois os directórios mais importantes onde temos as classes principais, o generator e o engine, que fazem todo o trabalho de geração e leitura dos pontos.

2.1. 3dFiles

Na pasta “3dFiles”, temos os ficheiros 3d gerados pelo generator, sempre que geramos algum ficheiro .3d este é automaticamente criado nessa mesma pasta.

2.2. Generator

Na diretoria Generator temos a classe generator que irá gerar o ficheiro .3d. Para usar o Generetor temos de escrever o comando com certos argumentos:

$ ./Generator [objeto] [argumentos] [nome do ficheiro .3d]

Onde o objeto pode ser:

- Plane
- Box
- Sphere
- Cone

E os respetivos argumentos:

- Plane: [dimensão] [divisão]
- Box: [unidades] [divisão]
- Sphere: [raio] [slices] [stack]
- Cone: [raio] [altura] [slices] [stacks]

Um exemplo de execução seria:

$ ./Generator sphere 1 10 10 sphere.3d

Com esta execução seria gerado um ficheiro chamado Sphere.3d na pasta 3dFiles, este ficheiro iria conter os pontos de uma esfera com 1 de raio, 10 slices e 10 stacks, para ser lido pelo Engine.

O ficheiro .3d gerado na primeira linha vai ter o número de vértices gerados, a qual nos vai auxiliar na leitura do ficheiro na classe Engine, as outras linhas vão ser os pontos gerados escritos 1 ponto por linha com espaços a separar cada coordenada desta forma um ponto (x1, y1, z1) e outro ponto (x2, y2, z2) ficaria escrito no ficheiro da forma:

- x1 y1 z1
- x2 y2 z2
- (…,…,…)

2.2.1. Plano

Para o desenho do plano quadrado em XZ é passado como argumentos a dimensão e divisão

O plano tem n divisões em cada eixo e cada uma dessas divisões está dividida em dois triângulos simétricos com 2 vértices coincidentes.

Construímos os triângulos de forma a que sejam visíveis de cima, ou seja, os vértices foram dados por ordem contrária aos ponteiros do relógio:

Triângulo 1:

A = (size/2 , 0 , size/2)

B = (size/2 , 0 , -size/2)

C = (-size/2 , 0 , -size/2)

Triângulo 2:

A = (size/2 , 0 , size/2)

C = (-size/2 , 0 , -size/2)

D = (-size/2 , 0 , size/2)

2.2.2. Box

A box é constituida por 2 parametros: unidades e divisão. Uma box com 3 divisões refere-se a um cubo em que cada face, está dividida numa matriz (3x3).

Na figura apresentada, o raciocínio por detrás da construção de cada face da box, precisa de 3 variáveis que implementamos(px,py,pz),

Inicialmente, colocamos as variáveis x e z com valor de metade, isto para ser mais fácil fazer a sua divisão no referencial e fazer com que

ele ficasse centrado e com a base em XZ. E utilizamos as 3 variáveis 'extra', para facilitar realizar divisões dos triângulos corretamente.

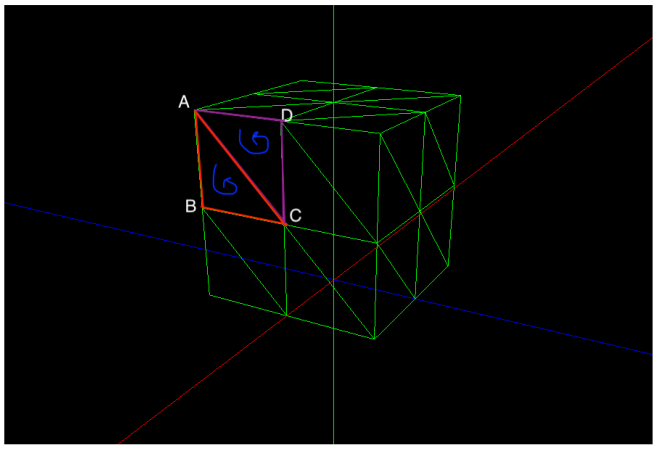

Figura 2 - Box I

Na seguinte figura, temos o exemplo frontal da box. É possível ver que estes 2 triângulos são simétricos e têm 2 vértices coincidentes, e neste caso,
terão sempre a coordenada do eixo Z como positiva, variando assim apenas o valor dos restantes eixos.
Construímos assim os triângulos de forma a que sejam visíveis de cima, sendo os vértices dados por ordem contrária aos ponteiros do relógio:

1º Triângulo :

A = (-x +(i*pX), y-(j*pY), z)

B = (-x +(i*pX), y-(j*pY) -pY, z)

C = (-x +(i*pX) +pX, y-(j*pY) -pY, z)

2º Triângulo :

A = (-x +(i*pX), y-(j*pY), z)

C = (-x +(i*pX) +pX, y-(j*pY) -pY, z)

D= (-x +(i*pX) +pX, y-(j*pY), z)

As variáveis i e j são criadas para variar a zona de divisão que nos encontramos. Isto é, estamos a percorrer criações dentro de dois ciclos,
um ciclo que incrementa a variável i até atingir o valor das divisões e outro ciclo com j, que realiza o mesmo algoritmo.
Com isto, temos as criações dos triângulos de cima para baixo e da esquerda para a direita.

Podemos ver que as variáveis que tivemos que implementar, nos ajudam ao que toca a alterar as coordenadas de cada divisão, pois ao
somarmos estas ao valor atual da variável em questão, saltamos de divisão em divisão dessa coordenada.
No seguinte exemplo, iremos apresentar uma face superior da box. Teremos a variável y, relativa à altura da figura estática e sempre positiva, enquanto as restantes variam.
Mais uma vez, faremos a construção dos diversos triângulos de cima para baixo e da esquerda para a direita.

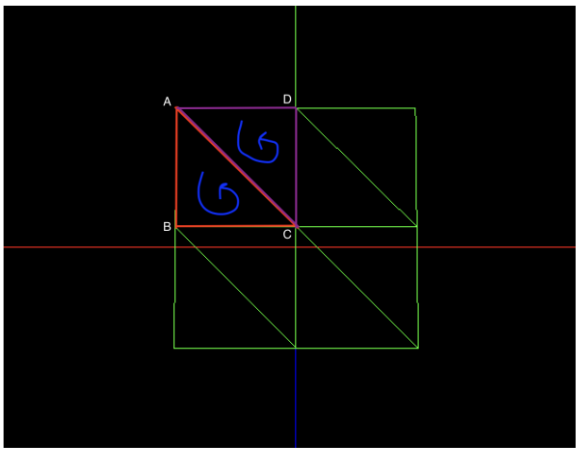

Figura 3 - Box II

1º Triângulo :

A = (-x +(i*pX), y, -z +(j*pZ))

B = (-x +(i*pX), y, -z +(j*pZ) +pZ)

C = (-x +(i*pX) +pX, y, -z +(j*pZ) +pZ)

2º Triângulo :

A = (-x +(i*pX), y, -z +(j*pZ))

C = (-x +(i*pX) +pX, y, -z +(j*pZ) +pZ)

D= (-x +(i*pX) +pX, y, -z +(j*pZ))

Conseguimos ver que o processo, acaba por ser semelhante em ambas as faces. Acaba por ter diferenças, mas depois de percebemos
como temos de olhar para as coordenadas e como o método funciona, fica mais fácil.

2.2.3. Cone

Para desenhar o cone são preciso como argumentos: o raio da base (radius), a altura do cone (height), número de fatias (slices) iguais em que se vai dividir a base, e por fim, número de camadas (stacks).
Construiremos o cone stack a stack, e em cada stack, slice a slice. Cada slice da base equivale a 1 triângulo, cada slice das stacks equivale a dois triângulos, e cada slice da última stack do cone (topo) equivale a 1 triângulo.
Podemos dividir o processo em 3 passos:

1. Base
2. Meio do Cone
3. Topo

2.2.3.1. Base

É essencial calcular o ângulo de cada slice (α = 2π /slices) e a altura de cada stack (height/stacks).

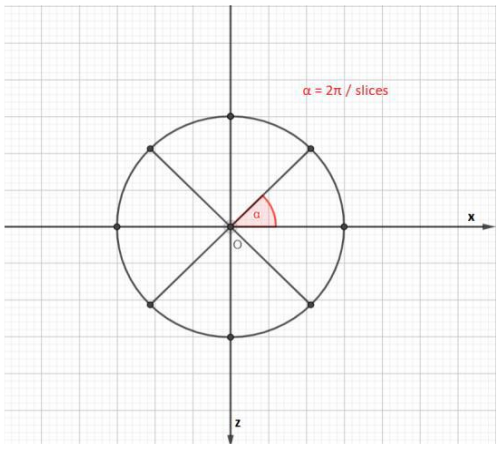

Figura 4 - Cone I

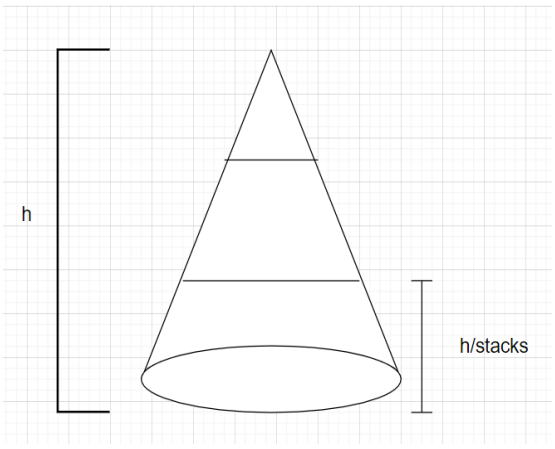

Figura 5 - Cone II

Para todos os triângulos da base é fácil perceber que um dos vértices será a origem (0,0,0), e que os dois outros vértices serão obtidos a partir das coordenadas polares, onde:

px = radius * sin (alpha)

pz = radius * cos (alpha)

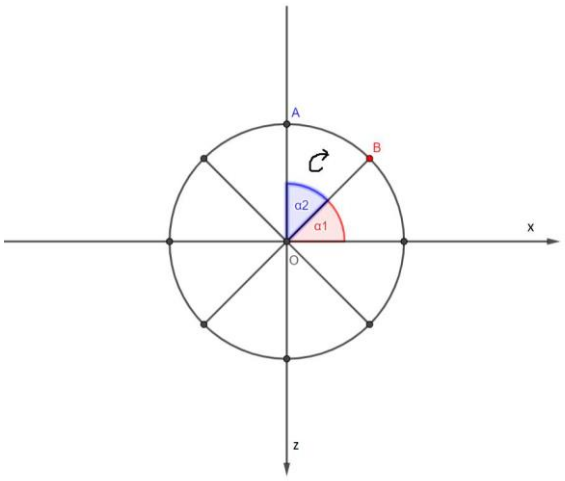

Figura 6 - Cone III

Tal como o triângulo OAB da imagem acima, todos os triângulos serão então da forma:

V1 = (0,0,0)

V2 = (radius * sin(alpha2) ,0, radius * cos(alpha2))

V3 = (radius * sin(alpha1) ,0, radius * cos(alpha1))

2.2.3.2. Meio

Nesta etapa criámos todas as stacks menos a última, pois a última por este método apesar de funcionar cria sobreposição de pontos, por isso é melhor separar das do meio.

Como os triângulos das stacks envolvem informação de duas stacks, ou seja, o topo de uma stack será a base da stack superior e a base de uma stack será o topo da stack inferior.

Os triângulos são criados da seguinte forma:

Triângulo 1:

V2 = (raio2ant * sin(alpha2), alturaBaixo, raio2ant * cos(alpha2))

V1 = (raio2 * sin(alpha2), alturaCima, raio2 * cos(alpha2))

V3 = (raio2ant * sin(alpha1), alturaBaixo, raio2ant * cos(alpha1))

Triângulo 2:

V3 = (raio2ant * sin(alpha1), alturaBaixo, raio2ant * cos(alpha1))

V1 = (raio2 * sin(alpha2), alturaCima, raio2 * cos(alpha2))

V4 = (raio2 * sin(alpha1), alturaCima, raio2 * cos(alpha1))

2.2.3.3. Topo

Para os triângulos da última stack, é fácil perceber que, um dos vértices de todos os triângulos será (0,height,0). Os outros 2 vértices utilizam informação da stack inferior. Sendo assim criados:

Triângulo:

V3 = (raio2ant * sin(alpha2), alturaBaixo, raio2ant * cos(alpha2))

V2= (raio2ant * sin(alpha1), alturaBaixo, raio2ant cos(alpha1))

V1 = (0, height, 0)

1.2.1. Esfera

Para ser possível desenhar a esfera são necessários os seguintes argumentos:

- raio (radius)
- fatias (slices)
- camadas (stacks)

Para o cálculo das coordenadas dos pontos utilizámos coordenadas esférias, da seguinte forma :

- alfa = (2* π) / slices
- beta = π / stacks

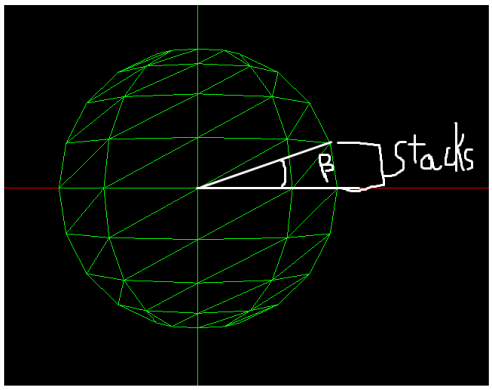

Figura 8 - Esfera I

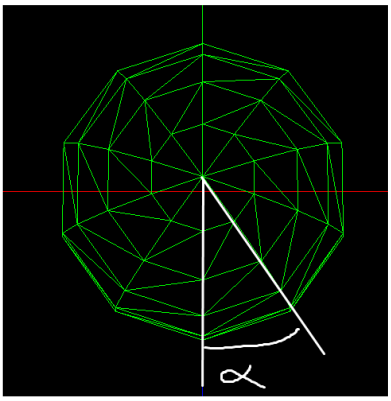

Figura 7 - Esfera II

Transformando agora em coordenadas cartesianas obtemos:«

- x = radius * cos (beta) * sin (alpha)
- y = radius * cos (beta) * cos (alpha)
- z = radius * sin (beta)

Para desenhar os triângulos percorremos cada fatia e começamos por desenhar dois triângulos um para a camada superior e outro para a camada inferior. Caso o número de camadas (stacks) seja superior a 2 percorremos as camadas intermedias e desenhamos dois triângulos de forma a formar um quadrado.

Camada Superior :

- V1 = (radius * cos(π/2) * sin(alpha1), radius * sin(π /2), radius * cos(π /2) *cos(alpha1))
- V2 = (radius * cos(π /2-beta) * sin(alpha1),radius * sin(π/2-beta),radius * cos(π/2-beta) * cos(alpha1))
- V3 = (radius * cos(π /2 - beta) * sin(alpha1+alpha),radius * sin(π /2-beta), radius * cos(π /2 - beta) * cos(alpha1+alpha))

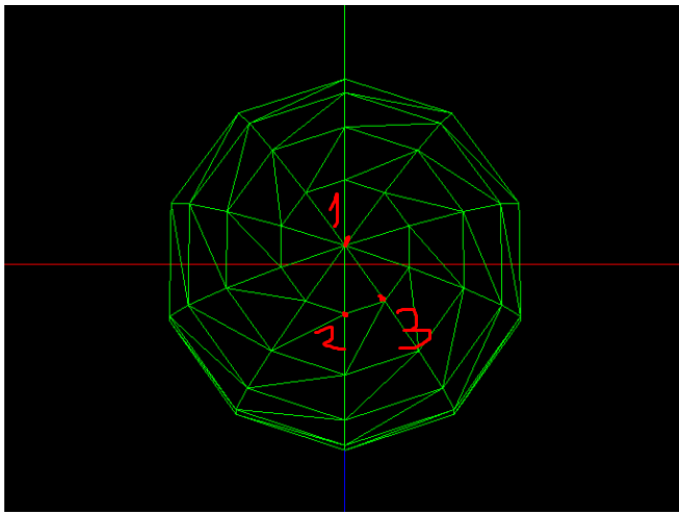

Figura 9 - Esfera III

Camada Inferior :

- V4 = (radius * cos(-π /2) * sin(alpha1), sin(-π /2) * radius, radius * cos(-π /2) * cos(alpha1))
- V5 = (radius * cos((-π /2) + beta) * sin(alpha1 + alpha), radius * sin((-π / 2) + beta), radius * cos((- π / 2) + beta) * cos(alpha1 + alpha))
- V6 = (radius * cos((-π /2) + beta) * sin(alpha1), radius * sin((-π / 2) + beta), radius * cos((-π / 2) + beta) * cos(alpha1))

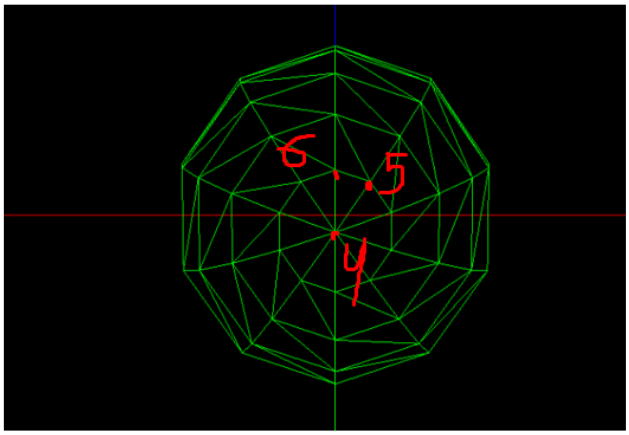

Figura 10 - Esfera IV

Camadas Intermédias :

- V7 = (radius * cos(beta1) * sin(alpha1), radius * sin(beta1), radius * cos(beta1) * cos(alpha1))
- V8 = (radius * cos(beta1 - beta) * sin(alpha1), radius * sin(beta1 - beta), radius * cos(beta1 - beta) * cos(alpha1))
- V9 = (radius * cos(beta1) * sin(alpha1 + alpha), radius * sin(beta1), radius * cos(beta1) * cos(alpha1 + alpha)) V10 = (radius * cos(beta1 - beta) * sin(alpha1+ alpha), radius * sin(beta1 - beta), radius * cos(beta1 - beta) * cos(alpha1+ alpha))

Onde: $0 \leq \text{alpha1} < 2\pi - \text{alpha}$ e $-\pi/2 + \text{beta} \leq \text{beta1} < \pi/2 - \text{beta}$

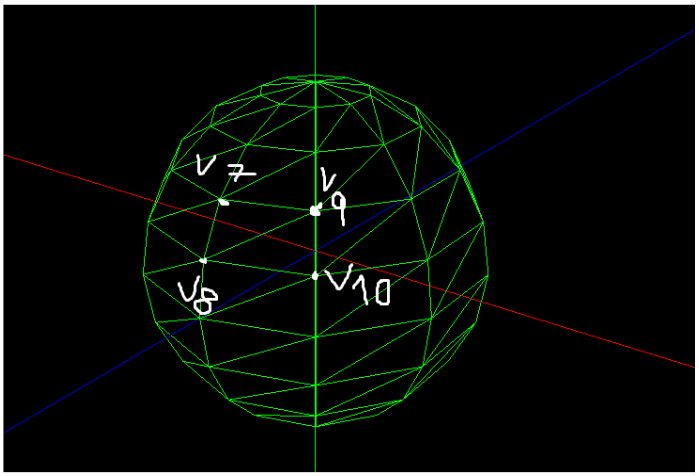

Figura 11 - Esfera V

2.3. Engine

Ao iniciar o Engine podemos lhe passar o ficheiro xml que queremos ler. Este ficheiro xml contém informações da câmera e o ficheiro .3d a ler. As informações respetivas à câmera estão dentro do node <camera> </camera> e o ficheiro .3d a ler dentro de <models> </models>.

Para nos auxiliar na leitura dos ficheiros XML usamos a livraria tinyxml2 que nos foi indicada no enunciado do trabalho, sendo esta uma livraria simples e eficaz para o parsing de ficheiros XML

A partir dessa livraria criamos uma função “readXML” que vai ler o ficheiro xml usando essa livraria, sendo esta função chamada quando o Engine é Iniciado. A função começa por ir ao node <model> e comparar o nome do .3d presente no xml, passando depois a respetiva diretoria a outra função chamada readFile. Depois vai ao node <camera> e vai guardar a informação nas respetivas variáveis.

A função “readFile”, vai ler o ficheiro .3d, lendo a primeira linha que são o número de vértices e fazendo um clico que vai iterar no número dos vértices, que são as linhas do ficheiro pois temos um vértice por linha, e a cada iteração guardamos o vértice (x,y,z) naquela linha em memória numa lista de “Pontos” que é uma simples classe com variáveis (x,y,z).

Finalmente a cada renderScene chamamos a função Draw que vai á lista de pontos, iterando sobre ela, desenhando 3 pontos de cada vez fazendo um triângulo.

2.3.1. Exemplo de output do Engine

- **Cone**

Raio = 1, Altura = 2, Slices = 4 e Stacks = 3

Posição da câmera x = 5, y = -2, z = 3 e fov = 40

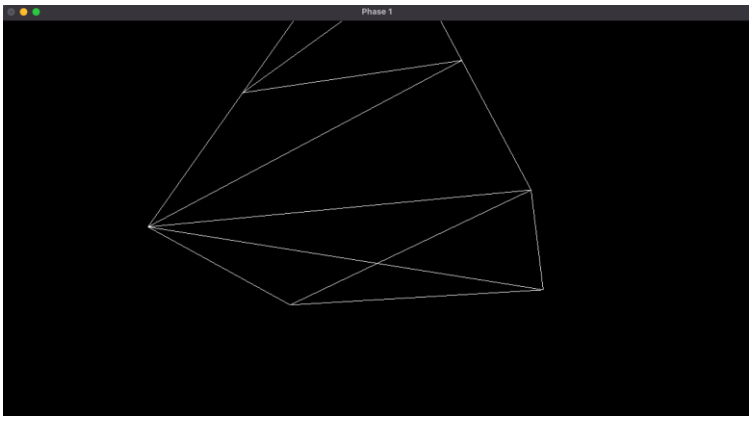

Figura 12 - Cone

- **Sphere**

Raio = 1, Splices = 10 e Stacks = 10

Posição da câmera x = 3, y = 2, z = 1 e fov = 60

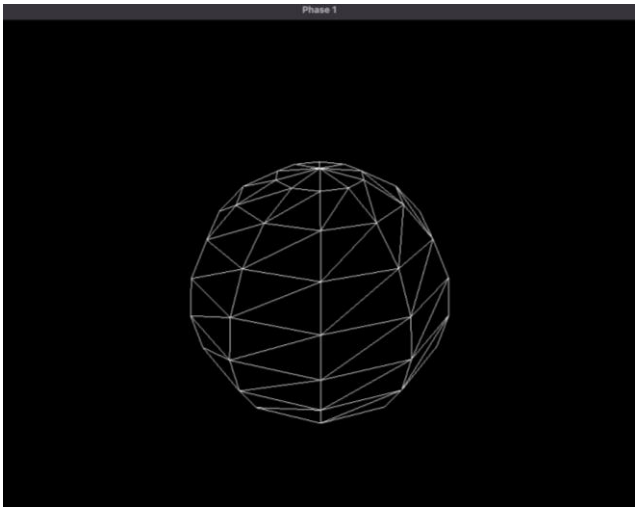

Figura 13 - Sphere

- **Box**

Unidades = 2 e divisão = 3

Posição da câmera x = 3, y = 2, z = 1 e fov = 100

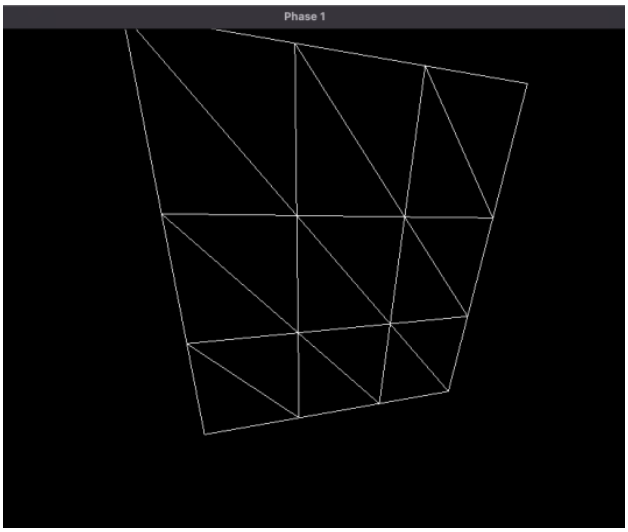

Figura 14 - Box

- **Plane**

Dimensão = 1 e divisão = 3

Posição da câmera x = 3, y = 2, z = 1 e fov = 60

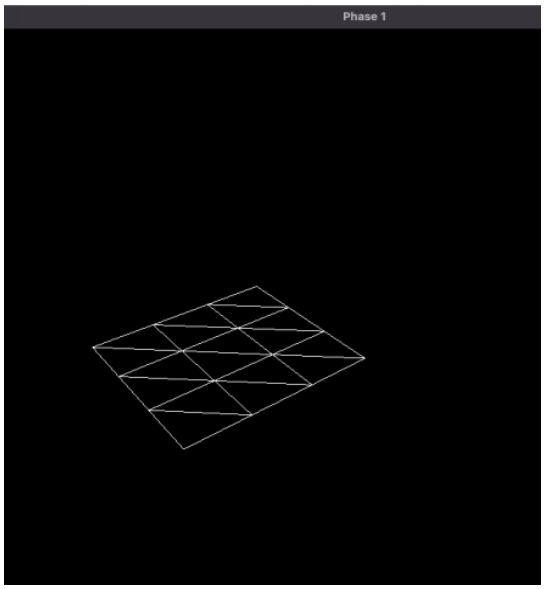

Figura 15 - Plane

3. Extras

3.1 Função Keyboard

- **e**: Adiciona/Remove os eixos XYZ.
- **f**: Modo GL_FILL.
- **l**: Modo GL_LINE.
- **p**: Modo GL_POINT.
- **r**: Muda cor do polígono para vermelho.
- **g**: Muda cor do polígono para verde.
- **b**: Muda cor do polígono para azul.
- **w**: Muda cor do polígono para branco.

```
//funçao das teclas extra
void keyboard(unsigned char key, int x, int y)
{
    if (key == 'e') {
        eixos = !eixos;
    }
    if (key == 'f') {
        tipo = GL_FILL;
    }
    if (key == 'l') {
        tipo = GL_LINE;
    }
    if (key == 'p') {
        tipo = GL_POINT;
    }

    if (key == 'r') {
        v = 1.0f;
        g = 0.0f;
        b = 0.0f;

    }
    if (key == 'g') {
        v = 0.0f;
        g = 1.0f;
        b = 0.0f;
    }
    if (key == 'b') {
        v = 0.0f;
        g = 0.0f;
        b = 1.0f;
    }
    if (key == 'w') {
        v = 1.0f;
        g = 1.0f;
        b = 1.0f;
    }

    glutPostRedisplay();

}
```

Figura 16 - Função Keyboard

3.2 Câmera

Implementamos uma câmera tipo explorardor, onde esta move-se na superfície do polígono a olhar para origem. Para tal criamos duas funções **processMouseButtons** e **processMouseMotion** que calculam a posição da câmera**.**

Ao pressionar o botão esquerdo do rato alteramos o alpha movendo para a esquerda ou direita onde vai aumentar e diminuir respetivamente, e o beta movendo para baixo ou para cima onde vai aumentar e diminuir respetivamente.

Ao pressionar o botão direito alteramos o raio movendo o rato para baixo ou para cima. Quanto menor o raio maior é a proximidade à origem.

4. Conclusão

Com a realização desta primeira entrega foi-nos possível consolidar a matéria dada tanto nas aulas teóricas como a que foi aplicada nas aulas práticas. Fez com que praticássemos as várias funções do **GLUT**, pesquisando ainda funções novas caso houvesse necessidade de as utilizar.

Alcançámos, no nosso ponto de vista, com sucesso os requisitos inicialmente pedidos, eventualmente uns com maior facilidade do que outros, devido também ao grau de dificuldade de cada um deles. Essencialmente esta primeira entrega serviu como uma ótima adaptação à cadeira e aos seus fundamentos, consolidando tudo o que foi aprendido até agora.

Por fim, esperamos continuar com este nível de aprendizagem e desempenho ao longo do semestre, de forma a efetuar com sucesso todas as entregas propostas.