Universidade do Minho
Escola de Engenharia

Computação Gráfica

Licenciatura em Engenharia Informática

Grupo 41

2022/2023

1ª Fase

Alexandra Santos (A94523)
Inês Ferreira (A97040)
José Rafael Ferreira (A97642)
Marta Sá (A97158)

ÍNDICE

INTRODUÇÃO

O objetivo da 1º fase deste trabalho consiste no desenvolvimento de um mecanismo 3D através da utilização de primitivas gráficas, tais como, um plano, uma caixa, uma esfera e de um cone.

Para esta fase pretende-se partir o problema em duas partes criando, o *generator*, capaz de gerar ficheiros com a informação de cada um dos modelos a serem produzidos, nomeadamente o número de vértices e a especificação das coordenadas de cada um deles. Na segunda parte criou-se o *engine* que, por sua vez, é responsável por ler e interpretar o ficheiro XML (passado como argumento), exibindo a respectiva primitiva gráfica.

No presente relatório apresentar-se-ão em detalhe as estratégias aplicadas nas etapas e as funcionalidades que sustentam esta fase do projeto.

GENERATOR

Este é responsável por gerar os vértices que dão origem aos triângulos das diferentes primitivas, através dos respetivos algoritmos. Depois de gerados, os vértices são guardados num ficheiro .3. Por sua vez, este poderá ser marcado num ficheiro XML dentro de uma label designada por *model*.

Para compilar o código executado, faz-se uso de um ficheiro *CMakeLists*, que permite a geração do projeto através do *software CMake*.

Os algoritmos que permitem a geração de cada primitiva (plano, caixa, esfera e cone) são explicitamente apresentados de seguida nos tópicos deste relatório.

É importante realçar que na elaboração das funções responsáveis por desenhar cada primitiva tivemos em consideração a minimização de erros de vírgula flutuante, optando por isso em usar inteiros na maioria dos corpos dos ciclos “for” e inicializando as variáveis no seu conteúdo como *floats*.

REGRA DA “MÃO DIREITA”

Faz todo o sentido abordar a técnica da “Mão Direita” num tópico porque é com o uso correto desta que a norma de qualquer superfície das primitivas segue o sentido pretendido, que inclusive permite a observação, ou não, num determinado ponto de vista dessas mesmas superfícies.

Para explicar os pontos possíveis de observação deve-se supor que tal como as superfícies abaixo se a norma da visão estiver na perpendicular a estas, utilizando a mão direita, ao apontar com todos os dedos à exceção do polegar no sentido indicado pelas setas o sentido para o qual aponta o polegar é paralelo à norma da superfície. Se este apontar para frente do monitor então a superfície seria visível ao executar, no entanto, o mesmo não acontece se apontar para as costas do monitor.

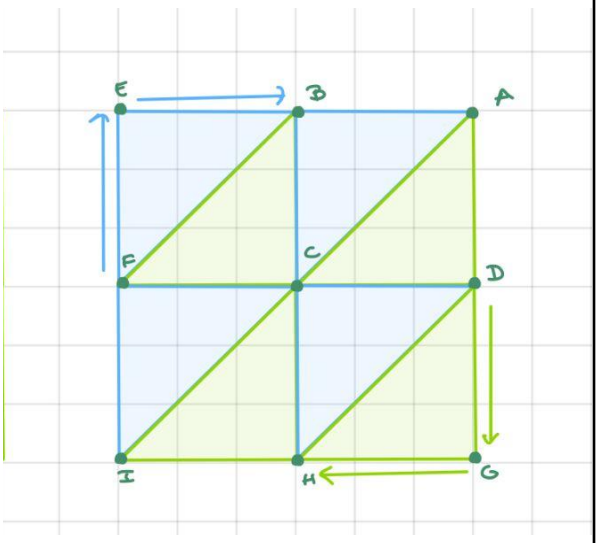

Figura 1: Rotação no sentido horário

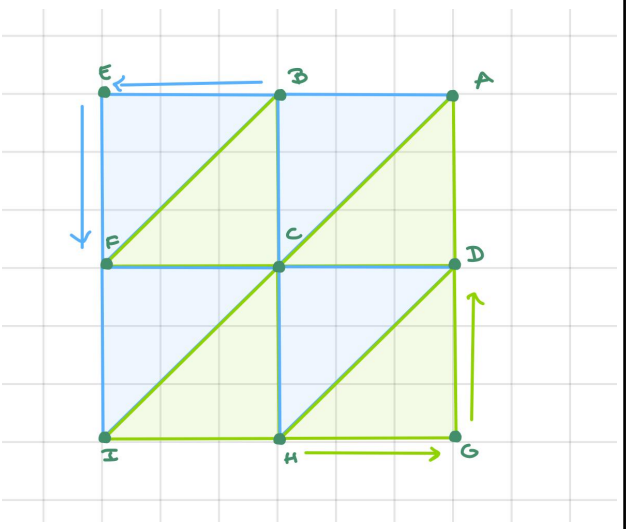

Figura 2: Rotação no sentido anti-horário

Esta regra é importante porque interfere na ordem com que os vértices são desenhados, significa isto que se forem considerados três vértices, que desenham um triângulo através da função GLBegin presente na biblioteca GLut, a ordem com que são desenhados afeta a direção da norma do triângulo. Portanto, como todas as primitivas deste projeto são compostas por vários destes triângulos, os pontos de visibilidade destas são afetados pelo uso desta regra.

PLANO

A função *plane()* dentro do *generator* é responsável pelo desenho do plano que recebe como parâmetros o lado deste (*dimension*), o número de divisões (*divisions*) e o *path* onde será armazenado o seu ficheiro .3d. Desta forma, cria-se a primitiva no plano XZ e armazena-se o total de vértices gerados juntamente com a especificação das coordenadas de cada um no referido ficheiro.

Cálculo dos pontos

A construção do plano inicia-se com o cálculo do número total de vértices necessários para a construção do mesmo. Para isso, utilizou-se a seguinte expressão:

$$vértices = 2 * 3 * divisions * divisions$$

Nesta fórmula de cálculo, 2 corresponde ao número de triângulos por divisão, 3 corresponde ao número de pontos por triângulo e *divisions* corresponde ao número de divisões do comprimento da figura.

Desta forma, numa primeira fase, identificou-se os maiores e menores valores que as coordenadas dos pontos do quadrado podem tomar nos eixos cartesianos. A coordenada de y vai ser sempre igual a 0, visto que o quadrado se encontra no plano XZ. Uma vez que o centro do quadrado é a origem, os valores mínimos e máximos para as coordenadas x e z são, respetivamente, -lado do quadrado/2 e lado do quadrado/2.

Uma vez que o plano, pretendido pelas suas divisões, é constituído por vários quadrados e tendo em conta que cada quadrado pode ser construído à custa de triângulos, decidiu-se calcular para cada um destes quadrados as coordenadas de cada triângulo. Para tal, determina-se o comprimento do lado do triângulo que constitui o plano e a par disto divide-se o tamanho da figura por 2 de forma a determinar os limites do intervalo de variação das coordenadas dos pontos.

De seguida, utiliza-se dois ciclos: o mais externo, fica responsável por incrementar a coordenada de z, desde o valor mínimo ao máximo e um mais interno, que incrementa a variável x também desde o valor mínimo ao máximo. Em ambos os ciclos as variáveis de x e z são progressivamente aumentadas com um valor de incremento que corresponde ao lado de um triângulo que constitui um subquadrado do plano.

Algoritmo do plano

```
ladoTriângulo = dimension / divisions;
 d = dimension / 2;


Para z = -d até z < d faz-se z += ladoTriângulo:
        Para x = -d até  x < d faz-se x += ladoTriângulo:
                Escrevemos o ponto(x + ladoTriângulo, 0, z)
                Escrevemos o ponto(x, 0, z)
                Escrevemos o ponto(x, 0, z + ladoTriângulo)

Para z = d até z > -d faz-se z -= ladoTriângulo:
        Para x = d até x > -d faz-se x -= ladoTriângulo:
                Escrevemos o ponto(x + ladoTriângulo, 0, z)
                Escrevemos o ponto(x, 0, z)
                Escrevemos o ponto(x, 0, z + ladoTriângulo)
```

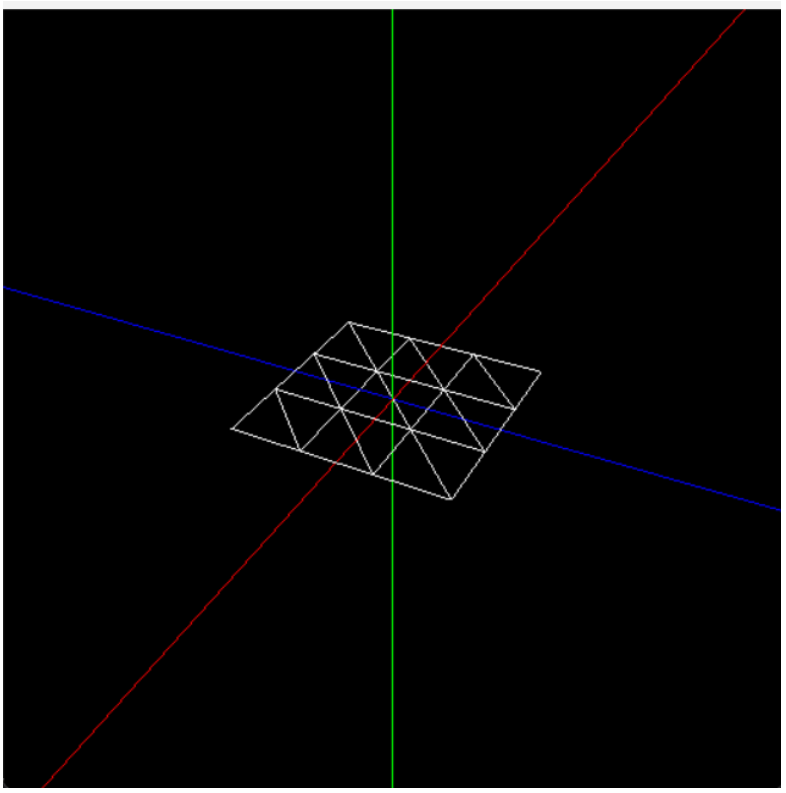

Figura 3: Exemplo de plano para os parâmetros dimension=1 e divisions=3

CAIXA

Para a construção da caixa criou-se a função *box*() que recebe como parâmetros o comprimento de um lado da caixa (*dimension*), o número de divisões (*divisions*) e o *path* para o ficheiro .3d. Assim, esta função irá criar uma caixa (*cubo*), armazenando os pontos gerados no ficheiro recebido.

Cálculo dos pontos

A construção da caixa inicia-se com o cálculo do número total de vértices necessários para a construção da caixa. Para isso, utilizou-se a seguinte expressão:

$$vertices = 2 * 3 * divisions * divisions * 6$$

Sendo que nesta fórmula de cálculo de vértices 2 corresponde ao número de triângulos por divisão, 3 corresponde ao número de pontos por triângulo, *divisions* corresponde ao número de divisões por face e 6 corresponde ao número de faces da caixa.

De seguida, no que toca à construção da caixa, aplicou-se para as 6 faces do cubo o raciocínio referido anteriormente na secção do plano. Tendo em conta que o cubo está centrado na origem podemos concluir que:

- Os pontos da caixa que pertencem à face de cima têm a coordenada y = dimension/2;
- Os pontos da caixa que pertencem à face de baixo têm a coordenada y = -dimension/2;
- Os pontos da caixa que pertencem à face lateral direita têm a coordenada x = dimension/2;
- Os pontos da caixa que pertencem à face lateral esquerda têm a coordenada x = -dimension/2;
- Os pontos da caixa que pertencem à face da frente têm a coordenada z = dimension/2;
- Os pontos da caixa que pertencem à face de trás têm a coordenada z = -dimension/2;

O resto das coordenadas dos pontos e o seu raciocínio pode ser explicada através do algoritmo abaixo apresentado.

Algoritmo da caixa

```
ladoTriângulo = dimension / divisions;
d = dimension / 2;
Face de cima
Para z = -d até z < d faz-se z += ladoTriângulo {
        Para x = -d até x < d faz-se x += ladoTriângulo {
                Escrevemos um triângulo com os pontos:
                (x + ladoTriâgulo, d, z);
                (x, d, z);
                (x, d, z + ladoTriângulo);
        }
}
Para z = d até z > -d faz-se z -= lado {
        Para x = d até x > -d faz-se x -= lado {
                Escrevemos um triângulo com os pontos:
                (x, -d, z);
                (x - ladoTriângulo, -d, z);
                (x, -d, z - ladoTriângulo);
        }
}
Face de baixo
Para z = -d até z < d faz-se z += ladoTriângulo {
        Para x = -d até x < d faz-se x += ladoTriângulo {
                Escrevemos um triângulo com os pontos:
                (x, -d, z);
                (x + ladoTriângulo, -d, z);
                (x, -d, z + ladoTriângulo);
        }
}
Para z = d até z > -d faz-se z -= lado {
        Para x = d até x > -d faz-se x -= lado {
                Escrevemos um triângulo com os pontos:
                (x, -d, z);
                (x - ladoTriângulo, -d, z);
                (x, -d, z - ladoTriângulo);
        }
}
Face de trás
Para y = -d até y < d faz-se y += ladoTriângulo {
        Para x = -d até x < d faz-se x += ladoTriângulo {
                Escrevemos um triângulo com os pontos:
                (x + ladoTriângulo, y, -d);
                (x, y, -d);
                (x, y + ladoTriângulo, -d);
        }
}
```

```
Para y = d até y > -d faz-se y -= ladoTriângulo {
        Para x = d até x > -d faz-se x -= ladoTriângulo {
                Escrevemos um triângulo com os pontos:
                (x - ladoTriângulo, y, -d);
                (x, y, -d);
                (x, y - ladoTriângulo, -d);
        }
}
```

Face da frente

```
Para y = -d até y < d faz-se y += ladoTriângulo {
        Para x = -d até x < d faz-se x += ladoTriângulo {
                Escrevemos um triângulo com os pontos:
                (x, y, d);
                (x + ladoTriângulo, y, d);
                (x, y + ladoTriângulo, d);
        }
}
Para y = d até y > -d faz-se y -= ladoTriângulo {
        Para x = d até x > -d faz-se x -= ladoTriângulo {
                Escrevemos um triângulo com os pontos:
                (x, y, d);
                (x - ladoTriângulo, y, d);
                (x, y - ladoTriângulo, d);
        }
}
```

Face da esquerda

```
Para y = -d até y < d faz-se y += ladoTriângulo {
        Para z = -d até z < d faz-se z += ladoTriângulo {
                Escrevemos um triângulo com os pontos:
                (-d, y, z);
                (-d, y, z + ladoTriângulo);
                (-d, y + ladoTriângulo, z);
        }
}
Para y = d até y > -d faz-se y -= ladoTriângulo {
        Para z = d até z > -d faz-se z -= ladoTriângulo {
                (-d, y, z);
                (-d, y, z - ladoTriângulo);
                (-d, y - ladoTriângulo, z);
        }
}
```

Face da direita

```
Para y = -d até y < d faz-se y += ladoTriângulo {
        Para z = -d até z < d faz-se z += ladoTriângulo {
                Escrevemos um triângulo com os pontos:
                (d, y, z + ladoTriângulo);
                (d, y, z);
                (d, y + ladoTriângulo, z);
        }
}
Para y = d até y > -d faz-se y -= ladoTriângulo {
        Para z = d até z > -d faz-se z -= ladoTriângulo {
                Escrevemos um triângulo com os pontos:
                (d, y, z - ladoTriângulo);
                (d, y, z);
                (d, y - ladoTriângulo, z);
        }
}
```

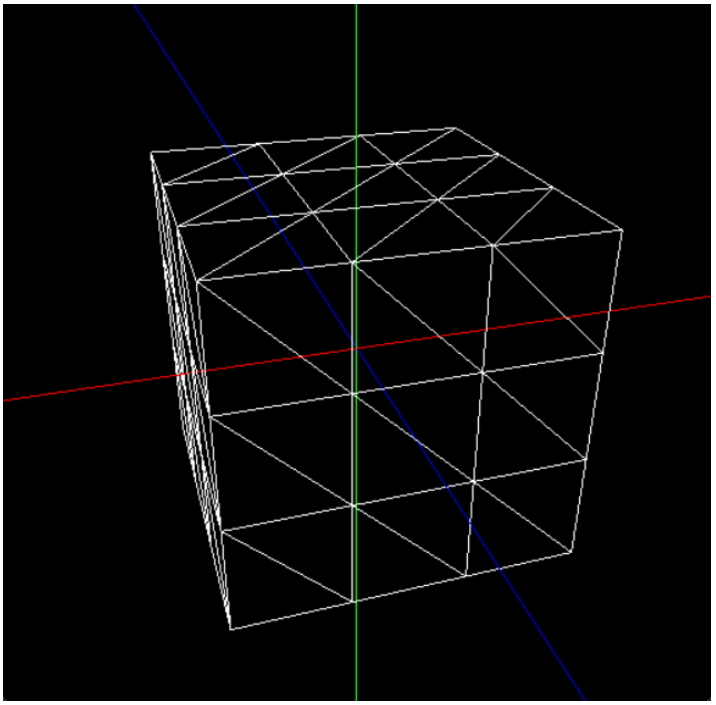

Figura 4: Exemplo caixa

ESFERA

Na construção da esfera, para além do parâmetro do raio (*radius*), foi necessário usar *stacks, slices* e um ficheiro de *output* para armazenar as informações da figura. Para dividir a esfera em *stacks* apenas foi necessário ter em consideração a altura, enquanto que para as *slices* foi necessário realizar as partições a partir do ângulo (*alpha* e *theta*). Ambas as divisões dividem a esfera em secções de iguais dimensões.

Cálculo dos pontos

A implementação da espera inicia-se com o cálculo do número de vértices presentes no cone a partir a seguinte expressão matemática:

$$vértices = 2 * 6 * (stacks / 2) * slices$$

Para representar a esfera, a abordagem baseou-se em representar para cada *stack* os vértices de cada *slice*.

Assim, dentro da esfera, o ângulo *theta* varia entre 0 e 2π (*2 * M_PI*) e o seu é atualizado a cada iteração passando a assumir o valor de *next_theta*. Por sua vez, o valor de *next_theta* é dado pela seguinte expressão:

$$next_{theta} = theta + (2 * M\_PI) / slices$$

Por outro lado, o ângulo *alpha* varia entre 0 e π (*M_PI*) e o seu valor é atualizado a cada iteração passando a assumir o valor de *next_alpha*. Por sua vez, o valor de *next_alpha* é dado pela seguinte expressão:

$$next_{alpha} = alpha + M\_PI / stacks)$$

Deste modo, baseamo-nos nas expressões abaixo para determinar as coordenadas de cada ponto através dos ângulos calculados.

$$pz = r \times \cos\beta \times \cos(\alpha);$$
$$px = r \times \cos\beta \times \sin(\alpha);$$
$$py = r \times \sin(\beta);$$

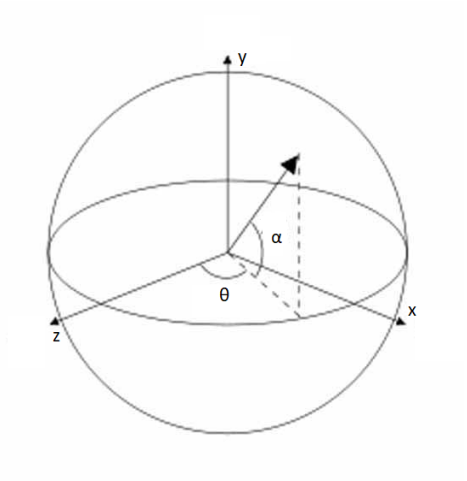

Figura 5: Exemplo do modelo da esfera

Algoritmo da esfera

```
Para i = 0 até stacks fazer i++ {
        next_alpha = alpha + (M_PI / stacks)

        Para j = 0 até slices  fazer j++ {
                next_theta = theta + ((2 * M_PI) / slices)

                Para a metade de cima da esfera
                x = radius * cos(alpha) * sin(theta);
                y = radius * sin(alpha);
                z = radius * cos(theta) * cos(alpha);

                x = radius * cos(alpha) * sin(next_theta);
                y = radius * sin(alpha);
                z = radius * cos(next_theta) * cos(alpha);

                x = radius * cos(next_alpha) * sin(theta);
                y = radius * sin(next_alpha);
                z = radius * cos(theta) * cos(next_alpha);

                x = radius * cos(next_alpha) * sin(next_theta);
                y = radius * sin(next_alpha);
                z = radius * cos(next_theta) * cos(next_alpha);

                x = radius * cos(next_alpha) * sin(theta);
                y = radius * sin(next_alpha);
                z = radius * cos(theta) * cos(next_alpha);
```

```
x = radius * cos(alpha) * sin(next_theta);
y = radius * sin(alpha);
z = radius * cos(next_theta) * cos(alpha);

Para a metade de baixo da esfera
x = radius * cos(alpha) * sin(theta);
y = radius * sin(alpha);
z = radius * cos(theta) * cos(alpha);

x = radius * cos(next_alpha) * sin(theta);
y = radius * sin(next_alpha);
z = radius * cos(theta) * cos(next_alpha);

x = radius * cos(next_alpha) * sin(next_theta);
y = radius * sin(next_alpha);
z = radius * cos(next_theta) * cos(next_alpha);

x = radius * cos(next_alpha) * sin(next_theta);
y = radius * sin(next_alpha);
z = radius * cos(next_theta) * cos(next_alpha);

x = radius * cos(alpha) * sin(next_theta);
y = radius * sin(alpha);
z = radius * cos(next_theta) * cos(alpha);

x = radius * cos(alpha) * sin(theta);
y = radius * sin(alpha);
z = radius * cos(theta) * cos(alpha);
}
}
```

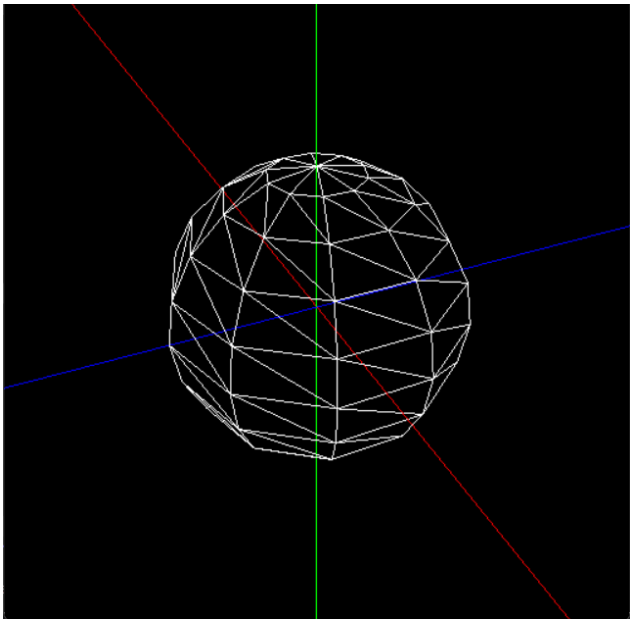

Figura 6: Exemplo de uma esfera

CONE

Para construir o cone, foram necessários os habituais parâmetros do raio (*radius*) e altura (*height*), sendo também necessário usar *stacks* e *slices*. *Stacks* referem-se às camadas do cone, na horizontal, as *slices* por sua vez, correspondem às fatias do cone na vertical.

Cálculo dos pontos

Tal como na esfera, para calcular o número de vértices presentes no cone, é usada a expressão matemática:

$$vertices = 3 * slices + 6 * stacks * slices$$

Para desenhar a primitiva, o algoritmo começa na sua base onde para cada *slice* calcula 3 pontos. De facto, um destes 3 pontos corresponde à origem e todos eles mantêm a coordenada y a 0. As restantes coordenadas

Para dividir o cone em stacks foi necessário ter em atenção a altura, enquanto que para as slices foi necessário focar nas divisões do ângulo alpha. Ambas as divisões dividem o cone em secções de iguais dimensões.

Para representar a lateral do cone, foi usada uma abordagem baseada nas *stacks* de modo a representar em cada uma delas os vértices de cada *slice*. Desta forma, adaptamos as fórmulas utilizadas na espera para calcular *px, py e pz* de forma a atualizar o raio à medida que subimos nas *stacks* e ignorando o valor do cosseno. Assim, para cada *stack* atualizamos um novo valor de y e um novo valor do raio com as seguinte fórmulas:

$$next\_y = y + height / stacks;$$
$$next\_r = r - (radius / stacks);$$

Após a definição destas variáveis, calculamos os pontos dos triângulos da seguinte forma:

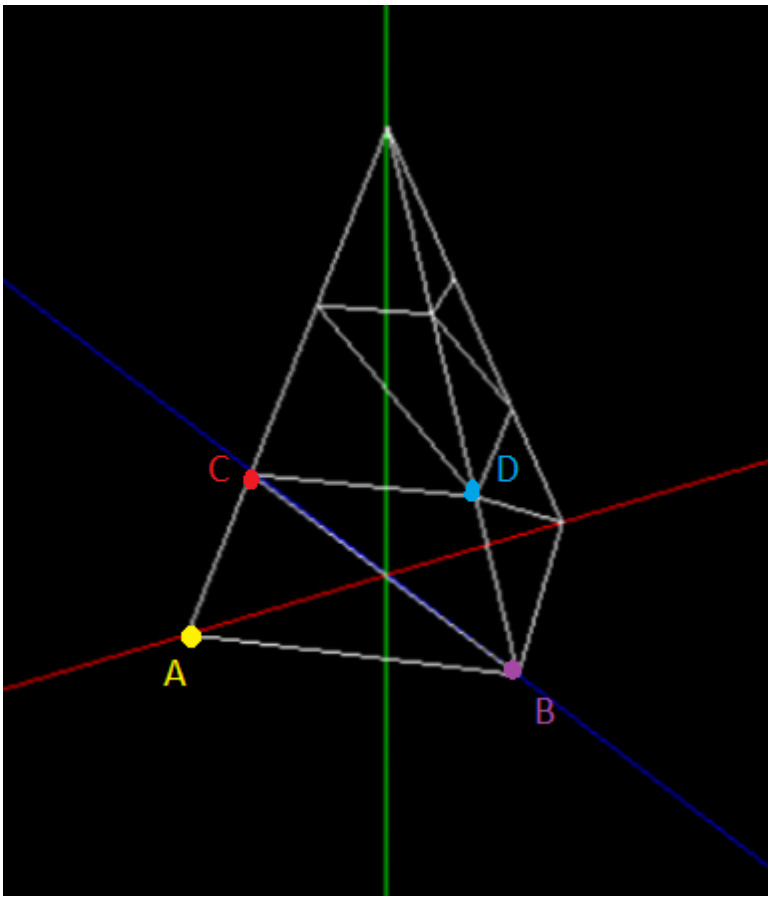

Figura 7: Cone 3D

Começando pelo ponto a A, as suas coordenadas x e z são calculadas da seguinte forma:

$$px = r * sin(j * ((2 * M\_PI) / slices))$$
$$pz = r * cos(j * ((2 * M\_PI) / slices))$$

A partir deste podemos calcular os restantes pontos. O ponto a B é obtido a partir do ponto A incrementando a variável *j* (*slice*):

$$px = r * sin((j + 1) * ((2 * M\_PI) / slices))$$
$$py = r * cos((j + 1) * ((2 * M\_PI) / slices))$$

Por fim, o ponto C é calculado a partir do ponto A determinando o novo valor de y e do raio:

$$px = next\_r * sin(j * ((2 * M\_PI) / slices))$$
$$py = next\_y$$
$$pz = next\_r * cos(j * ((2 * M\_PI) / slices))$$

Os triângulos de cima (no exemplo da figura 7 o triângulo formado pelos pontos B-C-D) foram calculados de forma semelhante.

Algoritmo do cone

Para a base do cone

Para cada i que corresponde a um slice do cone fazer:

```
write << cos((i + 1) * step * M_PI / 180.0) * radius << " " << 0.0f << " " << -sin((i +1)
* step * M_PI / 180.0) * radius << endl;
write << cos(i * step * M_PI / 180.0) * radius << " " << 0.0f << " " << -sin(i * step *
M_PI / 180.0) * radius << endl;
write << 0.0f << " " << 0.0f << " " << 0.0f << endl;
```

Para a lateral do cone

Para cada i que corresponde a uma stack do cone fazer:

```
next_y = y + height / stacks;
next_r = r - (radius / stacks);
Para cada j que corresponde a slice que atravessa a stack i do cone fazer:
    write << r * sin(j * ((2 * M_PI) / slices)) << " " << y << " " << r * cos(j * ((2 *
    M_PI) / slices)) << endl;
    write << r * sin((j + 1) * ((2 * M_PI) / slices)) << " " << y << " " << r * cos((j +
    1) * ((2 * M_PI) / slices)) << endl;
    write << next_r * sin(j * ((2 * M_PI) / slices)) << " " << next_y << " " <<
    next_r * cos(j * ((2 * M_PI) / slices)) << endl;
    write << next_r * sin(j * ((2 * M_PI) / slices)) << " " << next_y << " " <<
    next_r * cos(j * ((2 * M_PI) / slices)) << endl;
    write << r * sin((j + 1) * ((2 * M_PI) / slices)) << " " << y << " " << r * cos((j +
    1) * ((2 * M_PI) / slices)) << endl;
    write << next_r * sin((j + 1) * ((2 * M_PI) / slices)) << " " << next_y << " " <<
    next_r * cos((j + 1) * ((2 * M_PI) / slices)) << endl;
  Depois de terminar cada slice j declara-se:
    r = next_r;
    y = next_y;
```

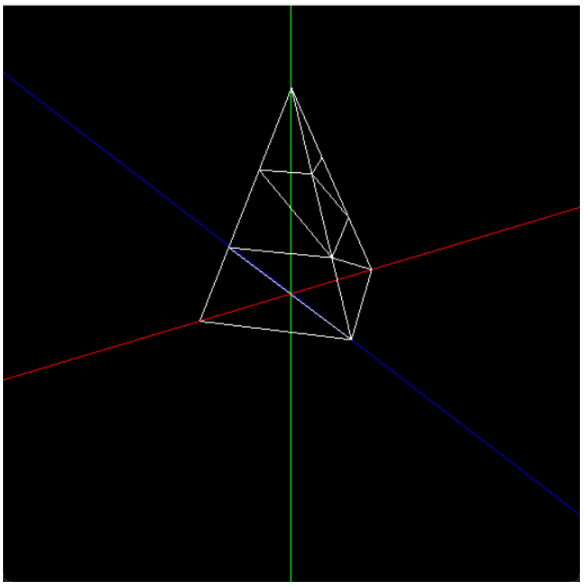

Figura 8: Exemplo cone

ENGINE

Nesta segunda parte do trabalho, desenvolveu-se uma aplicação capaz de ler e desenhar através da informação contida num ficheiro XML. No ficheiro XML está representado o código para obter os modelos a serem desenhados.

Visto que os modelos da aplicação foram gerados pelo *generator* desenvolvido (formato .3d), foi implementado um algoritmo capaz de ler o ficheiro e registar, em memória, todos os pontos.

FICHEIRO XML

Além do ficheiro .3d, é necessário guardar num ficheiro XML a indicação dos ficheiros previamente gerados. Assim, o ficheiro XML irá conter uma referência para cada ficheiro .3d que corresponde a uma primitiva. Deste modo, a leitura do ficheiro XML no *engine* permitirá desenhar a(s) primitiva(s). Para além disso, este ficheiro contém especificações destinadas à configuração da câmara e da janela de visualização. De seguida encontra-se um exemplo de um possível ficheiro XML gerado:

```
<world>
    <window width="512" height="512" />
    <camera>
        <position x="3" y="2" z="1" />
        <lookAt x="0" y="0" z="0" />
        <up x="0" y="1" z="0" />
        <projection fov="60" near="1" far="1000" />
    </camera>
    <group>
          <models>
            <model file="plane_2_3.3d" /> <!-- generator plane 2 3 plane_2_3.3d -->
            <model file="sphere_1_10_10.3d" /> <!-- generator sphere 1 10 10 sphere_1_10_10.3d -->
          </models>
     </group>
</world>
```

Figura 9: Arquivo de configuração XML

FICHEIRO 3D

A estrutura dos ficheiros .3d é análoga para as diferentes primitivas. A primeira linha do ficheiro contém o número de vértices. Nas restantes linhas estão presentes os vértices propriamente ditos, um por linha, e separados por espaço. A seguinte figura pretende ilustrar a estrutura do ficheiro:

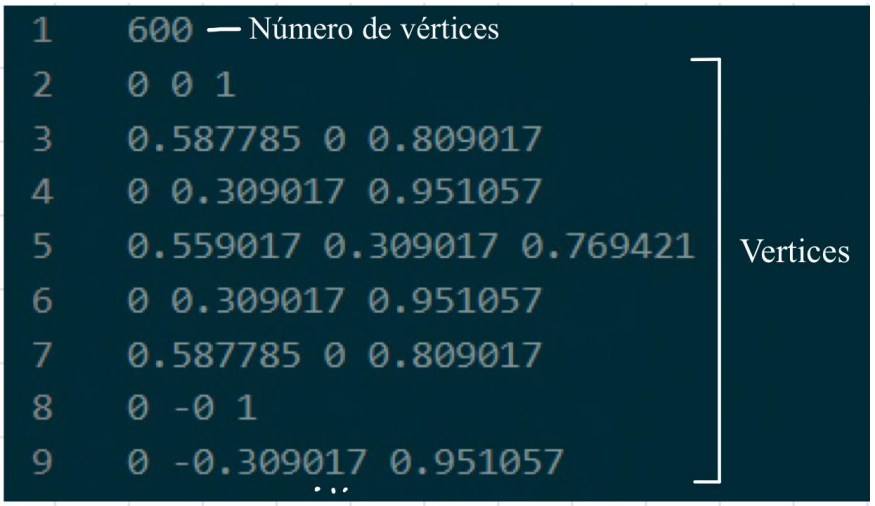

Figura 10: Ficheiro .3d

Junção de primitivas

Durante a leitura do ficheiro XML, caso exista mais do que uma primitiva no seu conteúdo, as imagens geradas vão ser sobrepostas. Abaixo, está representado um exemplo de duas primitivas sobrepostas desenhadas pelo e*ngine*.

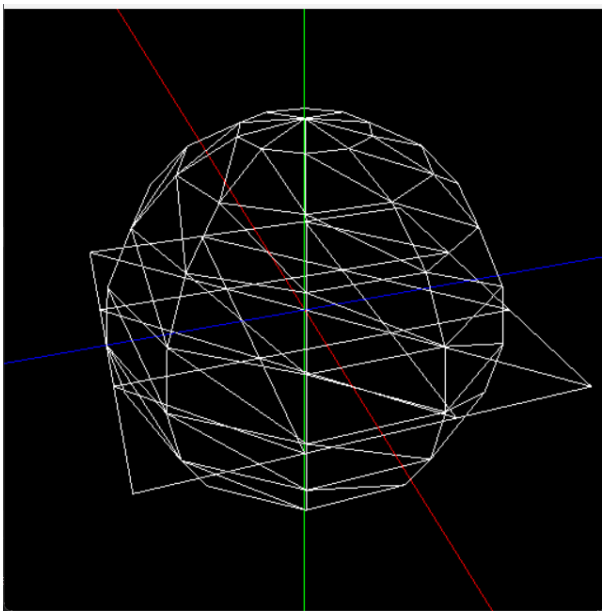

Figura 11: Plano 1 3 e esfera 1 10 10

CONCLUSÃO

Concluída a primeira fase do trabalho prático, antevê-se a obtenção de um bom resultado final tem impacto nas seguintes fases do projeto. Foi, particularmente, interessante compreender o processo de transformação de vários tipos de representações geométricas num plano tridimensional a partir de triângulos que é uma figura relativamente simples e que é usada no plano bidimensional. Esta fase foi útil para explorar algumas potencialidades da biblioteca GLut.

Por outro lado, existiram algumas dificuldades, tais como na implementação do código do referente ao cone, mas também na aplicação para aceder ao ficheiro XML. Apesar das dificuldades, os problemas foram ultrapassados e resolvidos.

Para concluir, consideramos que houve um balanço positivo do trabalho realizado dado que as dificuldades sentidas foram superadas e foram cumpridos todos os requisitos.